

สถาบันเทคโนโลยีคอมพิวเตอร์

Computer Architecture

การออกแบบและการวิเคราะห์

Design and Analysis

ดร. เกริก ภิรมย์สกุล

Krerk Piromsopa, Ph. D.

ລຳຫັບ ດຸນພກາທິພຍ່າ ເຕີກຫາຍດີດີ ແລະ ເຕີກຫຼູງລາດາ
ຂອບຄຸນລຳຫັບຄວາມໝາຍາອອງຈິວຕ

คำนำ

สถาปัตยกรรมคอมพิวเตอร์ (ภาษาอังกฤษใช้คำว่า Computer Architecture) เป็นพื้นฐานสำคัญที่นักคอมพิวเตอร์ทุกคนต้องรู้ บ่อยครั้งมักจะมีผู้โต้แย้งว่าแม้จะไม่มีความเข้าใจอันดีในสถาปัตยกรรมคอมพิวเตอร์ เรายังอาจจะเป็นนักคอมพิวเตอร์ได้ คำกล่าวนี้คงจะไม่จริงนัก เพราะทุกอย่างที่เป็นฮาร์ดแวร์ของคอมพิวเตอร์ ตั้งแต่คำสั่งที่ใช้ได้ การจัดเก็บข้อมูล การประมวลผล คือสิ่งที่กำหนดโดยสถาปัตยกรรมคอมพิวเตอร์ทั้งหมด ในฐานะวิศวกรคอมพิวเตอร์ และ นักวิทยาศาสตร์คอมพิวเตอร์ การออกแบบ และ การพัฒนา ระบบสถาปัตยกรรมคอมพิวเตอร์เป็นหน้าที่หลักโดยตรง ในฐานะนักพัฒนาซอฟต์แวร์ แม้จะมิได้ออกแบบหรือพัฒนาสถาปัตยกรรมคอมพิวเตอร์โดยตรง แต่ ความเข้าใจอันดีในสถาปัตยกรรมคอมพิวเตอร์ ย่อมช่วยให้สามารถพัฒนาซอฟต์แวร์ได้อย่างมีประสิทธิภาพ

เพื่อให้เห็นภาพชัดเจนยิ่งขึ้น จึงขอยกตัวอย่างประกอบดังนี้ ในการทำลังศึกษาระดับปริญญาเอก ในต่างประเทศนั้นมีการเรียนการสอนวิชา Algorithm (อ่านว่า อัลกอริทึม) ซึ่งอาจารย์ผู้สอนในตอนนั้น ได้ให้ผู้เรียนแข่งขันทำโปรแกรมเป็นโครงงานรายวิชาเพื่อแก้ปัญหาอีนพีบริบูรณ์ (NP-Complete) ให้ได้เร็วที่สุด หากใครเคยศึกษาปัญหาอีนพีบริบูรณ์มาบ้าง ย่อมทราบว่าปัญหานี้ในลักษณะดังกล่าว ไม่สามารถหาคำตอบได้โดยใช้เวลาเป็นพหุนา� กล่าวคือการหาคำตอบที่เหมาะสมนั้นต้องทดสอบวัดค่าจากทุกกรณีที่เป็นไปได้เพื่อให้ได้ทราบคำตอบที่ดีที่สุด เมื่อทราบเจนนั้น ผู้ก่อตัวได้มายกเว่ำ โปรแกรมของเพื่อนร่วมชั้นเรียนแต่ละคน คงจะมีลักษณะไม่แตกต่างกันมาก เพราะแนวทางคร่าวๆ คือ ต้องทำการวนซ้ำเพื่อวิเคราะห์กรณีที่เป็นไปได้ทั้งหมด และ นำคำตอบที่ดีที่สุดออกมานัดสุด ในการแข่งขันครั้งนี้ ผู้ใดใช้ความสามารถพื้นฐานด้านสถาปัตยกรรมคอมพิวเตอร์ของตนเอง เลือกใช้โครงสร้างข้อมูลแบบที่เหมาะสมและสามารถประมวลผลได้รวดสถาปัตยกรรมแบบอินเทลล์ Pentium 3 และ Pentium 4 (ซึ่งเป็นสถาปัตยกรรมที่ดีที่สุดในขณะนั้น) เอาจริงกลุ่มเพื่อนหลายคนในชั้นเรียน จะทำเวลาได้เป็นอันดับที่สองของชั้นเรียน ซึ่งเพื่อนร่วมชั้นเรียนต่างก็เป็นนักศึกษาปริญญาโทและปริญญาเอกที่มีความสามารถสูง ในครั้งนั้นเพื่อนร่วมชั้นที่ได้ผลการทำงานเป็นที่หนึ่งใช้เทคนิคที่แตกต่างกันไปกับวิธีการที่ผู้อื่นใช้ ซึ่งซับซ้อนกว่า เท่าไหร่ยาก และ ในบางกรณีทำงานได้ช้ากว่ามาก แต่เมื่อนับเวลาโดยรวมแล้วสามารถทำได้เร็วกว่าจึงเป็นผู้ชนะไป

ประสบการณ์ในครั้งนั้นจึงให้เห็นว่า หากมีความเข้าใจในสถาปัตยกรรมคอมพิวเตอร์ดีพอ แม้การเขียนโปรแกรมง่ายๆ ในภาษากระดับสูง ก็สามารถจะดึงประสิทธิภาพของหน่วยประมวลผลกลางออกมา ให้ได้ทุกหยด ต่างจากเพื่อนหลายคนในห้องเรียนครั้งนั้น ที่เขียนโปรแกรมลักษณะเดียวกันหรือคล้ายคลึงกันของผม แต่เนื่องจากความเข้าใจในสถาปัตยกรรมคอมพิวเตอร์อาจไม่ถูกซึ่งมากพอ จึงไม่สามารถที่จะดึงความสามารถบางอย่างของหน่วยประมวลผลออกมาใช้ได้ ทำให้ได้ประสิทธิภาพที่ได้แย่กว่าของผมมาก แน่นอนว่าผมมีประสบการณ์ที่คล้ายกันนี้ในหลายกรณี และในหลายโครงการที่ผมเคยมีส่วนเกี่ยวข้อง สิ่งที่ยกขึ้นมาเนี้ยเป็นเพียงกรณีศึกษาอันหนึ่งเท่านั้น (เคยมีนิสิตมาถามผมว่า algorithm ที่เค้าใช้ก็เหมือนกับของผม แต่ทำไมของเค้าถึงช้ากว่าของผมมาก ผมออมยิ้มในใจแล้วตอบนิสิตว่า อีกครั้งหนึ่งอยู่ที่โครงสร้างข้อมูลที่เลือกใช้ และการเลือกใช้คำสั่งที่เหมาะสมด้วย)

ดังนั้นผมจึงอยากให้หนังสือเล่มนี้ส่งข้อความถึง นิสิต นักศึกษา และ นักเรียนทั้งหลาย ว่า ความเข้าใจอันดีในสถาปัตยกรรมคอมพิวเตอร์ นอกจากจะเป็นพื้นฐานสำคัญสำหรับการศึกษา ยังเป็นสิ่ง

จำเป็นสำหรับการทำงานในสาขาวิชาชีพทางด้านคอมพิวเตอร์ทุกสาขา ผมหวังว่า การได้อ่านหนังสือที่ดี และ การได้ทดลอง(ออกแบบ) รวมถึงการวิเคราะห์ที่ดี จะช่วยให้สามารถศึกษาได้อย่างล่องแท้มากขึ้น

หนังสือเล่มนี้พยายามวางแผนพื้นฐานที่สำคัญ เริ่มจากความเข้าใจในสถาปัตยกรรมคอมพิวเตอร์ ไปถึงการวิเคราะห์สถาปัตยกรรมคอมพิวเตอร์ที่หลากหลายแบบต่างๆ อย่างไรก็ตามศาสตร์ด้านนี้มีการเปลี่ยนแปลงอยู่ตลอดเวลา และ สถาปัตยกรรมที่ถูกมองว่าดีในเวลาหนึ่ง มักจะถูกหักล้างในเวลาถัดมาด้วยเทคโนโลยีและความต้องการที่เปลี่ยนไป ผมจึงพยายามจัดวางเนื้อหา ให้เหมาะสมกับการสอนในหนึ่งภาคการศึกษา (ซึ่งสอดคล้องกับวิชา Computer System Architecture ซึ่งเป็นวิชาบังคับในหลักสูตรวิศวกรรมศาสตร์บัณฑิต สาขาวิศวกรรมคอมพิวเตอร์ ของ ภาควิชาวิศวกรรมคอมพิวเตอร์ จุฬาลงกรณ์มหาวิทยาลัย) ซึ่งอาจจะไม่ครอบคลุมถึงสถาปัตยกรรมทุกรูปแบบ ร่วมถึงเนื้อหาในบางด้าน อาจจะไม่ลงรายละเอียดในเชิงลึกมากนัก แต่จะเน้นในเชิงวิเคราะห์และการนำไปใช้เป็นหลัก ทั้งนี้ขออภัยตัวว่า ผมมีได้มองว่าตัวเองเป็นผู้เชี่ยวชาญด้านสถาปัตยกรรมคอมพิวเตอร์แต่อย่างใด (แน่นอนว่าคนที่ทำงานในสายวิชาชีพนี้ หากบอกว่าตัวเองเชี่ยวชาญในด้านใด คงจะเป็นความคิดที่ไม่ค่อยดีนัก เพราะศาสตร์ด้านนี้เปลี่ยนแปลงรวดเร็วทุกวันจนสิ่งที่เราเรียนรู้ พร้อมจะตกรุ่นเดือดุกเวลา) แม้ผมจะมีงานวิจัยและได้ยื่นขอจดสิทธิบัตร (pending) เกี่ยวกับสถาปัตยกรรมคอมพิวเตอร์ที่มีความสามารถด้านความมั่นคงปลอดภัยอยู่บ้างก็ตาม หากแต่ผมเพียงอย่างเดียวที่นักคอมพิวเตอร์รุ่นใหม่โดยเฉพาะกลุ่มที่เป็นคนไทย เข้าใจสถาปัตยกรรมคอมพิวเตอร์และนำไปประยุกต์ใช้ประโยชน์ เพื่อให้สามารถพัฒนาโปรแกรมที่มีประสิทธิภาพมากขึ้น หรืออย่างน้อย ก็สามารถที่จะเลือกใช้สถาปัตยกรรมคอมพิวเตอร์ที่เหมาะสมกับงานที่ใช้ได้

จากการสอบถามส่วนตัวของผม และ ประสบการณ์ที่เกี่ยวข้องกับการเรียนการสอนด้านอาร์แวร์และสถาปัตยกรรมคอมพิวเตอร์กว่าสิบปี (ผมเริ่มสอนวิชาสถาปัตยกรรมคอมพิวเตอร์ครั้งแรกในฐานะอาจารย์พิเศษที่มหาวิทยาลัยเอกชน ตั้งแต่ยังเรียนไม่จบปริญญาโท) ผมจึงอย่างต่อไปที่ต้องการให้สถาปัตยกรรมคอมพิวเตอร์ที่ส่วนหนึ่งได้จากการค้นคว้าและการลองถูกคลองผิด ให้กับ นิสิต นักศึกษา และครุ่นคิดอย่างลึกซึ้ง โดยหวังว่าจากจะเป็นการช่วยให้ นิสิต นักศึกษา สามารถเรียนลัดได้เร็วขึ้นแล้ว ยังจะเป็นการสืบท่องค์ความรู้ด้านสถาปัตยกรรมคอมพิวเตอร์ (ซึ่งปัจจุบัน นิสิต นักศึกษา และนักพัฒนาซอฟต์แวร์รุ่นใหม่ มักไม่ค่อยให้ความสนใจ หรือ ไม่เห็นความสำคัญ) ให้เป็นพื้นฐานสำคัญ และยังคงอยู่ต่อไป

ดร. เกริก ภิรมย์สิغا

ผู้ช่วยศาสตราจารย์
ภาควิชาวิศวกรรมคอมพิวเตอร์
จุฬาลงกรณ์มหาวิทยาลัย

Krerk.P@chula.ac.th

9 มิถุนายน 2557

สารบัญ

1 บทนำ.....	1
1.1 รู้ก่อนเรียน.....	1
1.2 ส่วนประกอบของระบบคอมพิวเตอร์.....	2
1.3 คำจำกัดความ.....	3
1.4 สถาปัตยกรรมชุดคำสั่ง (Instruction Set Architecture).....	3
1.5 โครงสร้างคอมพิวเตอร์ Computer Organization.....	5
1.6 การสร้างระบบคอมพิวเตอร์ (Implementation).....	6
1.7 เป้าหมายของการออกแบบ.....	6
1.8 แบบฝึกหัดท้ายบท.....	8
2 ประสิทธิภาพของระบบคอมพิวเตอร์.....	9
2.1 การวัดประสิทธิภาพของระบบคอมพิวเตอร์.....	9
2.2 Response Time และ Throughput.....	10
2.3 Benchmark.....	11
2.3.1 SPEC Benchmarks.....	12
2.3.2 คุณลักษณะของ benchmark ที่ดี.....	13
2.4 การเบรี่ยนเทียบประสิทธิภาพ.....	13
2.5 การหาเวลา CPU Time.....	16
2.6 กฎของ Amdahl.....	20
2.7 แนวทางการเพิ่มประสิทธิภาพของระบบคอมพิวเตอร์.....	23
2.8 สรุป.....	23
2.9 แบบฝึกหัดท้ายบท.....	25
3 สถาปัตยกรรมชุดคำสั่ง.....	27
3.1 สถาปัตยกรรมชุดคำสั่งในอดีต.....	28
3.1.1 สถาปัตยกรรมแบบ Accumulator.....	29
3.1.2 สถาปัตยกรรมแบบ Stack.....	31
3.1.3 สถาปัตยกรรมแบบ Memory-Memory.....	32
3.1.4 สถาปัตยกรรมแบบ Register-Memory.....	33
3.1.5 สถาปัตยกรรมแบบ Register-Register.....	35
3.2 หมวดคำสั่งและชุดคำสั่ง.....	36
3.3 การจัดการหน่วยความจำ (Memory Organization).....	39
3.3.1 การเรียงข้อมูล (Endian).....	39
3.3.2 การกำหนดตำแหน่งเริ่มต้นของข้อมูล (Memory Alignment).....	40
3.4 การอ้างอิงตำแหน่งหน่วยความจำ (Addressing Mode).....	43
3.5 รูปแบบคำสั่ง (Instruction Format).....	44
3.6 สถาปัตยกรรมชุดคำสั่งที่ดี.....	46
3.6.1 บทบาทของคอมไพล์เตอร์ในการจัดเรียงคำสั่ง (Optimization).....	47
3.7 การเรียกใช้โปรแกรมย่อย และ Exception.....	49
3.7.1 การส่งผ่านและคืนค่าระหว่างโปรแกรมย่อย.....	49
3.7.2 การบันทึกค่า Local Variable และ Register.....	50
3.7.3 Exception และ Interrupt.....	51

3.8 สรุป.....	52
3.9 แบบฝึกหัดท้ายบท.....	53
4 การออกแบบหน่วยประมวลผล แบบ single cycle.....	55
4.1 ขั้นตอนในการออกแบบหน่วยประมวลผลกลาง.....	56
4.1.1 สถาปัตยกรรมชุดคำสั่ง nanoLADA.....	56
4.2 องค์ประกอบภายในสำหรับสถาปัตยกรรม nanoLADA.....	58
4.2.1 ชุดเรจิสเตอร์ (Register file).....	58
4.2.2 Extender.....	60
4.2.3 หน่วยความจำ.....	60
4.3 ทางเดินข้อมูลสำหรับสถาปัตยกรรมชุดคำสั่ง nanoLADA.....	61
4.3.1 ทางเดินข้อมูลสำหรับคำสั่ง ORI และ ORUI.....	63
4.3.2 ทางเดินข้อมูลสำหรับคำสั่ง ADD.....	64
4.3.3 ทางเดินข้อมูลสำหรับคำสั่ง LW.....	64
4.3.4 ทางเดินข้อมูลสำหรับคำสั่ง SW.....	65
4.3.5 ทางเดินข้อมูลสำหรับคำสั่ง BEQ.....	66
4.3.6 ทางเดินข้อมูลของคำสั่ง JMP.....	67
4.3.7 สรุปทางเดินข้อมูลสำหรับสถาปัตยกรรมชุดคำสั่ง nanoLADA.....	68
4.4 สัญญาณควบคุม (Control).....	70
4.5 Microprogram.....	74
4.6 Critical Path และความเร็วสัญญาณนาฬิกา.....	74
4.7 แบบฝึกหัดท้ายบท.....	77
5 หน่วยประมวลผลกลางแบบ Multiple Cycle.....	79
5.1 ประสิทธิภาพของหน่วยประมวลผลกลางแบบ multiple cycle.....	81
5.2 ทางเดินข้อมูลสำหรับสถาปัตยกรรม nanoLADA แบบ multiple cycle.....	83
5.3 สัญญาณควบคุมสำหรับ multiple cycle processor.....	87
5.4 แบบฝึกหัดท้ายบท.....	90
6 การเพิ่มประสิทธิภาพด้วย Pipeline.....	91
6.1 ประสิทธิภาพของ Pipeline กรณีอุดมคติ.....	92
6.2 ปัญหาที่เป็นอุปสรรคต่อการทำ Pipeline.....	93
6.2.1 Structural Hazard.....	94
6.3 Data Hazard.....	97
6.3.1 การแก้ปัญหา Data Hazard ด้วยวิธีการทางซอฟต์แวร์.....	98
6.3.2 การแก้ปัญหา Data Hazard ด้วยวิธีการ hardware forward.....	100
6.3.3 Data Hazard แบบ RAW, WAW, WAR.....	103
6.4 Control Hazard หรือ Branch Hazard.....	104
6.4.1 การแก้ปัญหา Control Hazard ด้วยวิธีการ Branch Delay Slot.....	105
6.5 Data Path.....	107
6.6 สรุป Pipeline.....	108
6.6.1 ประสิทธิภาพของ Pipeline ในทางปฏิบัติ	109
6.7 แบบฝึกหัดท้ายบท.....	110
7 การจัดการหน่วยความจำ.....	111

7.1 Cache.....	111
7.2 หลักการท้องถิน (Locality).....	113
7.3 การจัดการ Cache เป็นต้น.....	115
7.3.1 Direct Mapped Cache.....	116
7.4 ประสิทธิภาพของ Cache.....	118
7.5 การลด Miss Ratio.....	120
7.6 การลด Miss Penalty.....	122
7.7 Block Size.....	123
7.8 Associativity.....	125
7.9 Replacement Algorithm.....	127
7.10 การจัดการ Cache กรณีการเขียนข้อมูล (Write Management).....	128
7.11 โปรแกรมและประสิทธิภาพของ Cache.....	129
7.11.1 การปรับโปรแกรมเพื่อช่วยลด Spatial Locality.....	130
7.11.2 การปรับโปรแกรมเพื่อลด Conflict Miss.....	130
7.12 หน่วยความจำเสมือน (Virtual Memory).....	131
7.12.1 การทำงานของ Virtual Memory.....	132
7.13 การทำงานร่วมกันของ Cache และ Virtual Memory.....	135
7.14 การเหลือมขั้นตอนของ Cache และ Virtual Memory.....	136
7.15 สรุป.....	137
7.16 แบบฝึกหัดท้ายบท.....	139

บรรณรูปภาพ

รูปที่ 1.1: ส่วนประกอบของระบบคอมพิวเตอร์.....	2
รูปที่ 1.2: ความสำพันธ์ของภาษาคอมพิวเตอร์ คอมไฟเลอร์ และ แอสเซมเบลย์.....	4
รูปที่ 1.3: ความสัมพันธ์ระหว่าง software, instruction set architecture และ hardware.....	5
รูปที่ 3.1: ทางเดินข้อมูลของสถาปัตยกรรมแบบ Accumulator.....	30
รูปที่ 3.2: ทางเดินข้อมูลของสถาปัตยกรรมแบบ Stack.....	31
รูปที่ 3.3: ทางเดินข้อมูลของสถาปัตยกรรมแบบ Memory-Memory.....	32
รูปที่ 3.4: ทางเดินข้อมูลของสถาปัตยกรรมแบบ Register-Memory.....	34
รูปที่ 3.5: โครงสร้างทางเดินข้อมูลของสถาปัตยกรรมแบบ Register-Register.....	35
รูปที่ 3.6: การเรียงข้อมูล (a) big endian (b) little endian.....	39
รูปที่ 3.7: การต่อหน่วยความจำเข้ากับหน่วยประมวลผล (a) ความกว้างสายสัญญาณเป็นหนึ่ง (b) ความกว้างสายสัญญาณเป็นสอง.....	41
รูปที่ 3.8: การอ่านค่าแบบมี restricted alignment และ แบบ unrestricted alignment.....	42
รูปที่ 3.9: โครงสร้างการจัดวางหน่วยความจำ(word addressing แสดงในเลขฐาน 16) (a) โปรแกรมภาษา C (b) สถาปัตยกรรมแบบ unrestricted alignment และ (c) สถาปัตยกรรมแบบ restricted alignment.....	42
รูปที่ 3.10: รูปแบบคำสั่ง (instruction format) ในสถาปัตยกรรม LADA.....	45
รูปที่ 4.1: ความสัมพันธ์ระหว่างสัญญาณนาฬิกา วงจรเชิงผสม และ วงจรเชิงลำดับ.....	52
รูปที่ 4.2: register file ขนาด 32×32 บิต สามารถอ่านได้ 2 ชุด และเขียนได้ 1 ชุดพร้อมกัน.....	55
รูปที่ 4.3: การทำงานของ Extender (a) zero extender (b) sign extender (c) zero padding.....	56
รูปที่ 4.4: โครงสร้างหน่วยความจำ.....	57
รูปที่ 4.5: ทางเดินข้อมูลสำหรับ PC \leftarrow PC+4.....	58
รูปที่ 4.6: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง ORI และ ORUI.....	59
รูปที่ 4.7: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง ADD.....	60
รูปที่ 4.8: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง LW.....	61
รูปที่ 4.9: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง SW.....	62
รูปที่ 4.10: ทางเดินข้อมูลส่วน PC เมื่อเพิ่มคำสั่ง BEQ.....	63
รูปที่ 4.11: ทางเดินข้อมูลส่วน PC เมื่อเพิ่มคำสั่ง JMP.....	63
รูปที่ 4.12: ทางเดินข้อมูลสำหรับสถาปัตยกรรม nanoLADA.....	65
รูปที่ 5.1: เปรียบเทียบเวลา (a) คำสั่งที่ไม่ต้องอ่านข้อมูลจากหน่วยความจำ และ (b) คำสั่งที่ต้องอ่านข้อมูลจากหน่วยความจำ.....	75
รูปที่ 5.2: ตัวอย่างการแบ่งขั้นตอนการทำงาน (a) การประมวลผลแบบ single cycle (b) การประมวลผลแบบ multiple cycle.....	76
รูปที่ 5.3: การแบ่งขั้นตอนการทำงาน.....	76
รูปที่ 5.4: การแทรก shadow register เพื่อปรับทางเดินข้อมูลให้รองรับการทำงานแบบ multiple cycle.....	79
รูปที่ 5.5: ทางเดินข้อมูลสำหรับ multiple-cycle processor.....	83

รูปที่ 5.6: state diagram	แสดงการทำงานของหน่วยประมวลผลกลางแบบ multiple cycle.....	84
รูปที่ 6.1: เปรียบเทียบเวลาการทำงานแบบมี pipeline และ ไม่มี pipeline.....		88
รูปที่ 6.2: ปัญหา structural hazard.....		91
รูปที่ 6.3: ปัญหา structure hazard จากการที่แต่ละคำสั่งใช้ช่วง cycle ไม่เท่ากัน		91
รูปที่ 6.4: การแก้ปัญหา structural hazard ด้วยการ stall.....		92
รูปที่ 6.5: การแก้ structural hazard โดยการปรับขั้นตอนการทำงานของคำสั่ง R-type.....		93
รูปที่ 6.6: ปัญหา data hazard.....		94
รูปที่ 6.7: การแก้ปัญหา data hazard ด้วย hardware stall.....		97
รูปที่ 6.8: ตัวอย่างการแก้ปัญหา data hazard ด้วย hardware forward.....		97
รูปที่ 6.9: ตัวอย่างกรณีที่ไม่สามารถ forward ได้.....		98
รูปที่ 6.10: การทำ hardware stall กรณีที่ไม่สามารถทำ hardware forward ได้.....		98
รูปที่ 6.11: Data Hazard แบบ RAW, WAW และ WAR.....		99
รูปที่ 6.12: การสร้าง bypass ขั้นตอนการ write back เพื่อแก้ปัญหา control hazard.....		101
รูปที่ 6.13: การแก้ปัญหา control hazard ด้วย branch delayed slot (แบบ taken) และ (not taken).....		103
รูปที่ 6.14: การแยก register คำสั่งแยกออกจากกันในแต่ละชั้นของ pipeline.....		104
รูปที่ 6.15: การส่งผ่านสัญญาณควบคุมสำหรับ pipeline.....		104
รูปที่ 7.1: ความเร็วและปริมาณหน่วยจดเก็บข้อมูลที่มีในระบบคอมพิวเตอร์.....		108
รูปที่ 7.2: ลำดับการเข้าถึงข้อมูล.....		109
รูปที่ 7.3: ความถี่การอ้างอิงช่วง address ของ GCC.....		110
รูปที่ 7.4: ตัวอย่างการจัดคนเข้าชั้งเก้าอี้ในห้อง (locality) แบบ direct mapped cache.....		112
รูปที่ 7.5: โครงสร้าง Direct Mapped Cache.....		113
รูปที่ 7.6: โครงสร้าง Direct Mapped Cache ที่รองรับการทำงานแบบ Spatial Locality.....		114
รูปที่ 7.7: กรณีมี Cache เพียง 1 entry.....		117
รูปที่ 7.8: การทำ Multilevel Cache เพื่อลด Miss Penalty.....		118
รูปที่ 7.9: block size ต่อ miss penalty.....		120
รูปที่ 7.10: กราฟความสัมพันธ์ระหว่าง block size, miss rate และ miss penalty.....		120
รูปที่ 7.11: ตัวอย่างการเกิด conflict miss ใน 1-way set associative cache.....		121
รูปที่ 7.12: การเพิ่ม set associative เพื่อลดปัญหา conflict miss.....		122
รูปที่ 7.13: โครงสร้างของระบบ Cache แบบ 4-way set associative.....		123
รูปที่ 7.14: การทำงานของ Virtual Memory.....		128
รูปที่ 7.15: การแปลง Virtual Address เป็น Physical Address.....		129
รูปที่ 7.16: การทำงานของ TLB.....		130
รูปที่ 7.17: การเข้าถึงข้อมูลในหน่วยความจำเมื่อมี Cache และ Virtual Memory.....		131

บรรณนิตรารง

ตารางที่ 2.1: เปรียบเทียบเวลาได้จาก benchmark.....	14
ตารางที่ 2.2: การใช้ค่าเฉลี่ยคณิตศาสตร์ถ่วงน้ำหนักเปรียบเทียบเวลาได้จาก benchmark (เปรียบเทียบเมื่อ Program 2 มีการใช้งานคิดเป็น 80% และ Program 1 มีการใช้งานคิดเป็น 20%).....	14
ตารางที่ 2.3: การเปรียบเทียบประสิทธิภาพด้วยอัตราส่วน พร้อมค่าเฉลี่ยคณิตศาสตร์.....	15
ตารางที่ 2.4: การเปรียบเทียบประสิทธิภาพด้วยอัตราส่วน พร้อมค่าเฉลี่ยเรขาคณิต.....	16
ตารางที่ 3.1: addressing mode แบบต่างๆ และการเทียบเคียงกับภาระดับสูง.....	44
ตารางที่ 4.1: สัญญาณควบคุมสำหรับสถาปัตยกรรม nanoLADA.....	68
ตารางที่ 4.2: การทำงานของ ALU สำหรับ nanoLADA.....	68
ตารางที่ 4.3: สัญญาณ alu_ops สำหรับสถาปัตยกรรม nanoLADA.....	69
ตารางที่ 5.1: สัญญาณควบคุมสำหรับหน่วยประมวลผลแบบ multiple cycle.....	84
ตารางที่ 7.1: ประสิทธิภาพของ Replacement Algorithm แบบ LRU และ RR.....	124

บรรณนิสมการ

สมการที่ 2.1 ประสิทธิภาพและเวลา.....	9
สมการที่ 2.2 Speedup.....	9
สมการที่ 2.3 Elapse Time.....	11
สมการที่ 2.4 Cycle Time.....	17
สมการที่ 2.5 CPU Time/instruction.....	17
สมการที่ 2.6.....	17
สมการที่ 2.7 CPU Time.....	17
สมการที่ 2.8 CPU Time.....	17
สมการที่ 2.9 Execution Time.....	21
สมการที่ 2.10 Overall Speedup.....	22
สมการที่ 6.1: CPU Time (pipeline).....	89
สมการที่ 7.1: Memory Access Time (หน่วย cycle).....	116
สมการที่ 7.2: ค่า CPI กรณีที่มี CACHE.....	116

1 บทนำ

1.1 รักก่อนเรียน

เพื่อให้การศึกษาวิชาสถาปัตยกรรมคอมพิวเตอร์ง่ายขึ้น ผู้เรียนควรมีทักษะในการออกแบบระบบจะต้องรู้ว่าจะออกแบบได้ทั้งแบบผสม (Combinational Logic Circuit) และ แบบเชิงลำดับ (Sequential Circuit) นอกจากนี้ผู้เรียนควรจะมีความรู้พื้นฐานทางซอฟต์แวร์พอสมควรเพื่อประกอบความเข้าใจ ก่อสร้างคือ ผู้เรียนควรมีประสบการณ์ในการเขียนโปรแกรม (ไม่จำกัดภาษา) ทั้งนี้ หากผู้เรียนมีประสบการณ์กับภาษาแอสเซมบลี (Assembly) จะช่วยให้สามารถเข้าใจเนื้อหาบางส่วนได้ง่ายและรวดเร็วยิ่งขึ้นเป็นพิเศษ แต่หากไม่มีความคุ้นเคยกับภาษาแอสเซมบลี อย่างน้อยผู้เรียนควรจะเข้าใจหลักการเบื้องต้น เช่น ตัวแปร การจองหน่วยความจำ ประโยคเงื่อนไข และการเรียกใช้งานโปรแกรม ย่อย ในกรณีของพื้นฐานความรู้นี้ คิดว่าไม่น่าจะเป็นประเด็นแต่อย่างไร เนื่องจากการเรียนการสอนทางด้านวิทยาศาสตร์และวิศวกรรมศาสตร์สมัยใหม่ มักจะแทรกองค์ความรู้เกี่ยวกับการเขียนโปรแกรมเข้าไปอยู่แล้ว เช่น นิสิตนักศึกษาที่นี่ในหลายหลักสูตรทางด้านคอมพิวเตอร์ มักจะได้เรียนวิชา Java Programming และนิสิตนักศึกษาที่นี่ที่สองทางสายวิศวกรรม มักจะได้เรียนพื้นฐาน Digital Logic อญ্তแล้ว

เนื้อหาในหนังสือเล่มนี้เรียบเรียงมาจากต่างๆ ทั้งฉบับคลาสิก (ตำนาน) ฉบับอ้างอิง ฉบับอ่านเล่น รวมถึงงานตีพิมพ์และงานวิจัยด้านต่างๆ เนื่องจากสถาปัตยกรรมคอมพิวเตอร์เป็นศาสตร์ที่มีการพัฒนา ปรับปรุงและเปลี่ยนแปลงอยู่เสมอ ผู้เรียนควรค้นคว้าเพิ่มเติมจากแหล่งความรู้อื่นประกอบด้วย

โครงสร้างและสถาปัตยกรรมคอมพิวเตอร์ที่อธิบายในที่นี้ จะอาศัยพื้นฐานจากสถาปัตยกรรมชุดคำสั่งสมมุติ ชื่อ LADA (Light-weighted Architecture for Design and Analysis) เป็นหลักโดย LADA (และ叫做LADA ในบางช่วงบางตอนของหนังสือนี้) เป็นสถาปัตยกรรมคอมพิวเตอร์ประเภท RISC แบบ 32 บิต ที่ผู้เขียนพัฒนาขึ้นเพื่อประกอบการเรียนการสอนโดยเฉพาะ (ผู้เรียนยังไม่ต้องกังวลว่า สถาปัตยกรรมคอมพิวเตอร์ประเภท RISC คืออะไร เมื่อถึงเนื้อหาส่วนที่เกี่ยวข้อง ก็จะเข้าใจเอง) สถาปัตยกรรม LADA มุ่งเน้นให้มีโครงสร้างเข้าใจได้ง่าย ไม่ซับซ้อน เหมาะกับการศึกษาในระดับเบื้องต้น ในส่วนที่เป็นการกล่าวถึงเรื่องต่อ�อดต่างๆ อาจมีการแทรกเนื้อหาของสถาปัตยกรรมอื่นเพื่อประกอบการอธิบายเพิ่มเติมบ้าง การแทรกรายละเอียดของสถาปัตยกรรมเหล่านี้ เน้นเพื่อให้เกิดการเปรียบเทียบและเห็นข้อแตกต่าง ทั้งนี้ผู้เขียนจะไม่แทรกเนื้อหาให้มากเกินไป โดยการสอดแทรกจะขึ้นกับความต้องเนื่องและความเหมาะสมของเนื้อหาเป็นหลัก และในตัวอย่างประกอบคำอธิบาย จะพยายามใช้สถาปัตยกรรม LADA เป็นพื้นฐาน (ยกเว้นกรณีที่ต้องการเปรียบเทียบกับสถาปัตยกรรมอื่น) เพื่อให้เกิดความต้องเนื่องของเนื้อหา

1.2 ส่วนประกอบของระบบคอมพิวเตอร์

ระบบคอมพิวเตอร์โดยทั่วไป ประกอบด้วยหน่วยประมวลผลกลาง แป็นพิมพ์ Mouse จอภาพ เครื่องพิมพ์ ระบบ Multimedia และอุปกรณ์ต่อพ่วงอีกมากมาย โครงสร้างดังกล่าวสามารถถูกจำแนกตามการทำงานได้เป็น

- หน่วยประมวลผลกลาง (Central Processing Unit)
- อินพุต (Keyboard, Mouse, etc...)
- เอ้าท์พุต (Display, Printer, etc...)
- หน่วยความจำ (Memory)
- หน่วยจัดเก็บข้อมูล (Storage, Disk, CD, Tape, etc)
- ระบบเครือข่ายคอมพิวเตอร์ (Computer Network)

การแบ่งในลักษณะนี้ เป็นการแบ่งส่วนประกอบของคอมพิวเตอร์ตามมุ่งมั่งของผู้ใช้งานทั่วไป ในทางสถาปัตยกรรมนิยมแบ่งส่วนประกอบของคอมพิวเตอร์เป็นสามส่วนหลักตามลักษณะของหน้าที่ ได้แก่ หน่วยประมวลผลกลาง หน่วยความจำ และระบบอินพุตเอ้าท์พุต จะสังเกตเห็นว่า อินพุต เอ้าท์พุต และ ระบบเครือข่าย รวมถึงหน่วยจัดเก็บข้อมูล ถูกมองรวมเป็นระบบเดียวกันทั้งหมด ส่วนหนึ่งเนื่องจากในทางสถาปัตยกรรม อุปกรณ์เหล่านี้เชื่อมต่อกับหน่วยประมวลผลกลางผ่านระบบสาย สัญญาณมาตรฐาน I/O (I/O bus) หรืออ่อนกัน ต่างจากหน่วยความจำที่มักจะมีสายสัญญาณเชื่อมต่อเฉพาะสำหรับหน่วยความจำ(memory bus) ดังนั้น หากมองสถาปัตยกรรมคอมพิวเตอร์จะมุ่งมอง ของนักสถาปัตยกรรมแล้ว จะแสดงได้ดังรูปที่ 1.1

รูปที่ 1.1: ส่วนประกอบของระบบคอมพิวเตอร์

ในเบื้องต้น หนังสือเล่มนี้จะมุ่งความสนใจไปที่หน่วยประมวลผลกลางเป็นหลักก่อน จากนั้นจึงจะกล่าวถึงองค์ประกอบรอบข้าง เมื่อส่วนดังกล่าวมีผลกับประสิทธิภาพของระบบหรือส่งผลให้การออกแบบ และการวิเคราะห์แตกต่างไป กล่าวคือ จะเริ่มที่หน่วยประมวลผลกลาง โครงสร้างภายใน และองค์ประกอบต่างๆ ที่ส่งผลต่อประสิทธิภาพ จากนั้นจึงจะกล่าวถึงหน่วยความจำและสถาปัตยกรรมที่เกี่ยวข้องต่อในส่วนท้าย

1.3 คำจำกัดความ

สถาปัตยกรรมคอมพิวเตอร์ (Computer Architecture) ประกอบขึ้นจากคำสองคำคือ สถาปัตยกรรม (architecture) และ คอมพิวเตอร์ (Computer) สถาปัตยกรรมหมายถึง โครงสร้าง องค์ประกอบ หรือ กรอบความคิด ของระบบใดๆ(ซึ่งในความหมายทั่วไป มักจะหมายถึงอาคาร สถานที่ หรือ สิ่งก่อสร้าง) เมื่อนำมาประยุกต์ใช้ในทางคอมพิวเตอร์ จึงหมายถึงโครงสร้างและองค์ประกอบของระบบคอมพิวเตอร์ ในการออกแบบเดียวกัน เราก็นิยมเรียกผู้ออกแบบสถาปัตยกรรมคอมพิวเตอร์ว่า สถาปนิก(architect) เช่นเดียวกันกับสถาปนิกผู้ออกแบบอาคารเช่นกัน

เมื่อประมวลความหมายต่างๆ เข้าด้วยกัน รวมความแล้ว สถาปัตยกรรมคอมพิวเตอร์ หมายถึง โครงสร้างการทำงานของเครื่องคอมพิวเตอร์ โครงสร้างของหน่วยประมวลผลกลาง การแทนข้อมูลระดับเครื่อง สถาปัตยกรรมชุดคำสั่ง (รวมความถึง ภาษาเครื่อง หรือ ภาษาแอสเซมบลี) นอกจากนี้ ยังรวมความถึงการจัดการระบบหน่วยความจำ การเชื่อมต่อระหว่างเครื่องคอมพิวเตอร์และอุปกรณ์ต่อพ่วง การศึกษาวิชาสถาปัตยกรรมคอมพิวเตอร์ จึงเป็นการศึกษาที่มุ่งเน้นถึงโครงสร้างที่จำเป็นต่อการออกแบบ และใช้งานส่วนต่างๆ ของระบบคอมพิวเตอร์ อย่างไร้ตามเมื่อกล่าวถึงคำว่า สถาปัตยกรรมคอมพิวเตอร์ โดยทั่วไปผู้คนมักนึกถึงหน่วยประมวลผลกลางเป็นหลัก ซึ่งเป็นแนวคิดที่ไม่ถูกต้องนัก

อ้างอิงจากความหมายของสถาปัตยกรรมคอมพิวเตอร์แล้ว การศึกษาเรื่องสถาปัตยกรรมคอมพิวเตอร์ จะต้องประกอบไปด้วยอย่างน้อยสามส่วน คือ

- สถาปัตยกรรมชุดคำสั่ง (Instruction Set Architecture)
- โครงสร้างคอมพิวเตอร์ (Computer Organization)
- การสร้างระบบคอมพิวเตอร์ (Implementation)

เนื้อหาในหนังสือเล่มนี้จะครอบคลุมทั้งสามส่วนแต่จะมุ่งเน้นที่ส่วน สถาปัตยกรรมชุดคำสั่งและ โครงสร้างคอมพิวเตอร์เป็นสำคัญ ทั้งนี้เพื่อให้ผู้เรียนเข้าใจอย่างลึกซึ้ง จึงขอขยายความคำจำกัดความของสถาปัตยกรรมชุดคำสั่ง โครงสร้างคอมพิวเตอร์ และ การสร้างระบบคอมพิวเตอร์ ดังต่อไปนี้

1.4 สถาปัตยกรรมชุดคำสั่ง (Instruction Set Architecture)

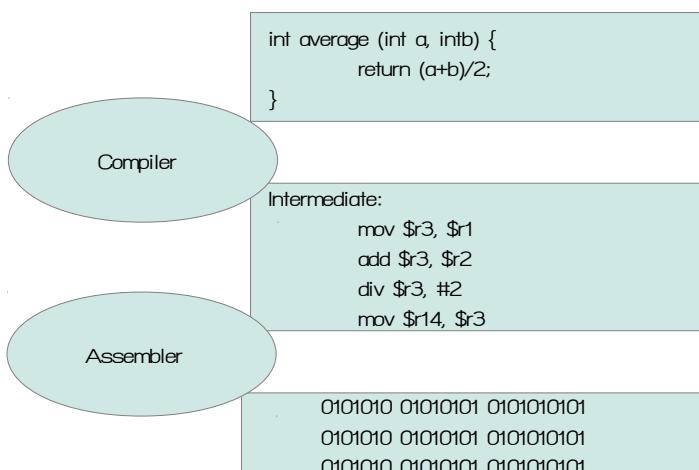
สถาปัตยกรรมชุดคำสั่ง (Instruction Set Architecture) เป็นตัวกลางในการอธิบายถึงโครงสร้างและ

ความสามารถของหน่วยประมวลผลกลางโดยมองจากมุมมองของผู้พัฒนาซอฟต์แวร์และผู้พัฒนาฮาร์ดแวร์ เพื่อประกอบความเข้าใจ เรายังทำความเข้าใจขั้นตอนการพัฒนาซอฟต์แวร์โดยทั่วไปดังนี้

การพัฒนาซอฟต์แวร์โดยทั่วไป ผู้พัฒนาหรือโปรแกรมเมอร์ มักจะพัฒนาโปรแกรมโดยใช้ภาษาชั้นสูง (เช่น ภาษา C หรือ C++) จากนั้นโปรแกรมจะถูกแปลโดยคอมไพล์เตอร์ (compiler) เป็นภาษาแอสเซมบลี (assembly) และโปรแกรมภาษาแอสเซมบลีเหล่านั้นจึงถูกแปลโดยแอสเซมเบลอร์ (assembler) เป็นภาษาเครื่อง หรือ Machine Code ในที่สุด ขั้นตอนดังกล่าวแสดงดังรูปที่ 1.2

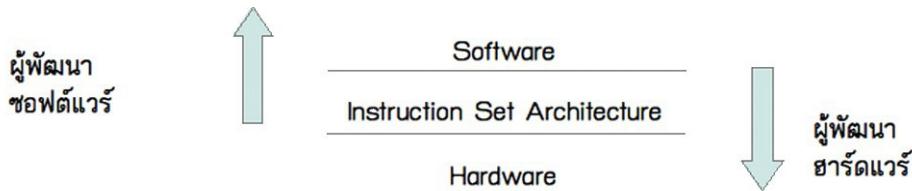
จากลักษณะการพัฒนาซอฟต์แวร์ดังกล่าวพบว่า โปรแกรมที่มีต้นฉบับภาษาชั้นสูงเหมือนกันนั้น เมื่อถูกแปลโดยคอมไпал์เตอร์ และ แอสเซมเบลอร์ แล้วจะได้ภาษาเครื่อง (Machine code) ซึ่งแตกต่างกันไปตามหน่วยประมวลผลกลางที่มีสถาปัตยกรรมชุดคำสั่งแตกต่างกัน ซึ่งอาจกล่าวได้ว่า ชุดคำสั่งภาษาเครื่องเหล่านี้เป็นตัวกลางในการเชื่อมต่อระหว่างฮาร์ดแวร์และซอฟต์แวร์ เพื่อให้เครื่องคอมพิวเตอร์ทำงานตามโปรแกรมที่ต้องการ เราอาจเรียกชุดคำสั่งในภาษาแอสเซมบลีและภาษาเครื่องเหล่านี้ว่า สถาปัตยกรรมชุดคำสั่ง ซึ่งก็คือ โครงสร้างชุดคำสั่งสำหรับสั่งให้หน่วยประมวลผลกลางที่ใช้สถาปัตยกรรมดังกล่าวทำงานตามที่ชุดคำสั่งโปรแกรม กำหนด

ในการพัฒนาซอฟต์แวร์โดยเฉพาะอย่างยิ่งของซอฟต์แวร์ระบบ ผู้พัฒนาจึงควรมีความรู้เกี่ยวกับสถาปัตยกรรมชุดคำสั่งเหล่านี้เพื่อที่จะสามารถพัฒนาซอฟต์แวร์ได้ โดยไม่จำเป็นต้องอาศัยความรู้เกี่ยวกับฮาร์ดแวร์ด้านล่าง (ในระดับล่าง) อย่างไรก็ตามหน่วยประมวลผลกลางที่แตกต่างกัน อาจมีสถาปัตยกรรมชุดคำสั่งที่เหมือนกันได้ ซึ่งนั่นหมายความว่าซอฟต์แวร์ที่ทำงานบนระบบคอมพิวเตอร์เครื่องหนึ่งจะสามารถทำงานได้บนระบบคอมพิวเตอร์ทุกรูปแบบที่มีโครงสร้างสถาปัตยกรรมชุดคำสั่งเหมือนกัน ด้วยอย่างเช่น ซอฟต์แวร์ที่พัฒนาขึ้นสำหรับสถาปัตยกรรม 80486 ของ Intel สามารถทำงานได้บนสถาปัตยกรรม Pentium III ของ Intel หรือ สถาปัตยกรรม Opteron ของ AMD เป็นต้น ทั้งนี้เพราะสถาปัตยกรรมเหล่านี้ แม้จะมีโครงสร้างภายในที่แตกต่างกัน แต่ต่างก็ใช้สถาปัตยกรรมชุดคำสั่งเดียวกัน หรือ สถาปัตยกรรมชุดคำสั่งที่คล้ายคลึงกัน



รูปที่ 1.2: ความสำคัญของภาษาคอมพิวเตอร์ คอมไпал์เตอร์ และ แอสเซมเบลอร์

จากมุ่มมองของผู้พัฒนาอาร์ดแวร์นั้น ผู้พัฒนาอาร์ดแวร์เพียงออกแบบสถาปัตยกรรมชุดคำสั่งและสร้างวงจรประมวลให้สามารถทำงานได้ตามสถาปัตยกรรมนั้น โดยไม่ต้องคำนึงถึงการพัฒนาซอฟต์แวร์ ทำให้ผู้พัฒนาซอฟต์แวร์เป็นอิสระจากอาร์ดแวร์ ความสัมพันธ์ดังกล่าวแสดงได้ในรูปที่ 1.3



รูปที่ 1.3: ความสัมพันธ์ระหว่าง software, instruction set architecture และ hardware

แม้ว่าการอ้างอิงถึงสถาปัตยกรรมคอมพิวเตอร์โดยใช้สถาปัตยกรรมชุดคำสั่งเป็นหลัก จะเป็นประโยชน์ในการพัฒนาซอฟต์แวร์และชาร์ดแวร์ก็ตาม แต่บางกรณีการพัฒนาซอฟต์แวร์และชาร์ดแวร์โดยยึดติดกับสถาปัตยกรรมชุดคำสั่งมากเกินไป ทำให้ไม่สามารถติดคันหรือพัฒนาสถาปัตยกรรมคอมพิวเตอร์แบบใหม่ขึ้นมาได้ อย่างไรก็ตามการที่หน่วยประมวลผลรุ่นใหม่สามารถรองรับการทำงานของสถาปัตยกรรมแบบเดิมได้ก็เป็นนโยบายทางการตลาดที่ต้องยังคง เนื่องจากทันทีที่สถาปัตยกรรมหรือหน่วยประมวลผลกลางรุ่นใหม่ออกวางตลาดก็จะมีซอฟต์แวร์รองรับการทำงานทันทีเช่นกัน ตัวอย่างที่เห็นได้ชัดคือ การที่หน่วยประมวลผลกลางรุ่นล่าสุดของบริษัท Intel ยังคงสามารถประมวลผลโปรแกรมที่พัฒนาบนระบบหน่วยประมวลผลรุ่น 8088 ที่พัฒนาขึ้นมากกว่า 40 ปีที่แล้วได้

1.5 โครงสร้างคอมพิวเตอร์ Computer Organization

โครงสร้างคอมพิวเตอร์ หรือ Computer Organization เป็นการกล่าวถึงโครงสร้างภายในของระบบคอมพิวเตอร์ นั่นหมายความว่า การกล่าวถึงโครงสร้างคอมพิวเตอร์จะต้องเป็นการกล่าวถึง โครงสร้างภายในที่ทำให้หน่วยประมวลผลกลาง และส่วนประกอบต่างๆ สามารถทำงานร่วมสอดคล้องกับสถาปัตยกรรมชุมคำสั่งที่กำหนดได้

หน่วยประมวลผลกลางหรือ CPU เป็นศูนย์กลางในการทำงานของระบบคอมพิวเตอร์ ทุกสิ่งที่กล่าวถึงในสถาปัตยกรรมชุดคำสั่งนั้น หน่วยประมวลผลกลางจะต้องมีความสามารถทำงานได้ตามที่ระบุไว้ หากเปรียบเทียบกับร่างกายเราราจากล่างไว้ด้านบน หน่วยประมวลผลกลางเป็นสมองมนุษย์เรา ทั้งนี้ภายในหน่วยประมวลผลกลาง สามารถจำแนกออกเป็นองค์ประกอบหลักได้ดังนี้

- หน่วยประมวลผลทางคณิตศาสตร์และทางตรรกะ (ALU)
- หน่วยควบคุม (Control Unit)
- หน่วยความจำชั่วคราว (Register)

- ระบบสายสัญญาณสำหรับการเชื่อมต่อต่าง ๆ (BUS)

เหตุที่หน่วยประมวลผลกลางจำเป็นจะต้องมีความสามารถในการจำนั้น เพราะในการคำนวณทุกครั้ง มีความจำเป็นจะต้องใช้การจำประมวลผลการคิดคำนวณเสมอ เพื่อไม่ให้สับสนกับหน่วยความจำทั่วไป ประเภทอื่นๆ เราจึงนิยมเรียกหน่วยความจำชั่วคราวภายในหน่วยประมวลผลกลางนี้ว่า Register (อ่านว่า เรจิสเตอร์)

นอกจากนี้ ในระบบคอมพิวเตอร์ ยังมีระบบอยู่อีกหนึ่งที่อยู่นอกหน่วยประมวลผลกลางอีก เช่น หน่วยความจำหลัก (main memory) และระบบ I/O สำหรับเชื่อมต่อ กับอุปกรณ์ต่างๆ การกล่าวถึง Computer Organization นั้น จะต้องครอบคลุมถึงหน่วยต่างๆ เหล่านั้นด้วยเช่นกัน ในหลังสือเล่มนี้ จะกล่าวถึงรายละเอียดของระบบหน่วยความจำในบทที่ 7

1.6 การสร้างระบบคอมพิวเตอร์ (*Implementation*)

การสร้างระบบคอมพิวเตอร์ หรือ implementation ในที่นี้ หมายถึง การสร้างวงจรและอุปกรณ์เชื่อมต่อต่างๆ ให้เป็นไปตามข้อกำหนดที่ระบุไว้ในโครงสร้างคอมพิวเตอร์ ในการสร้างระบบคอมพิวเตอร์มักจะต้องอาศัยความรู้ทางวิศวกรรมไฟฟ้าควบคู่กับความรู้ทางวิศวกรรมคอมพิวเตอร์ด้วย หากจะแยกออกจากกันให้ชัดเจนคือ โครงสร้างคอมพิวเตอร์จะกล่าวถึงระบบสายสัญญาณและการเชื่อมต่อ โดยไม่สนใจคุณลักษณะทางไฟฟ้า (คุณลักษณะทางไฟฟ้า เช่น แรงดันไฟฟ้า 5 Voltage แทน logic High) หรือโครงสร้างของ transistor ซึ่งบุคคลที่ทำหน้าที่ดังกล่าว มักจะเรียกว่า วิศวกรออกแบบ (design engineer)

หากจะจำแนกการสร้างระบบคอมพิวเตอร์ ออกย่อยอีก อาจจะได้เป็นการออกแบบ layout ของ transistor การสร้างวงจร (นำ transistor มาประกอบกันเป็นวงจร) การตรวจสอบความถูกต้อง (verification) เป็นต้น

เนื้อหาในหนังสือเล่มนี้ จะไม่ครอบคลุมถึงการสร้างระบบคอมพิวเตอร์ในเชิงลึก แต่จะบรรยายการออกแบบโดยใช้ภาษา Verilog HDL² ในการบรรยายการออกแบบ เพื่อประกอบความเข้าใจเป็นหลัก

1.7 เป้าหมายของการออกแบบ

ประเด็นสุดท้ายที่จะกล่าวถึงในบทนี้ คือ เป้าหมายของการออกแบบ กล่าวคือ ทุกสถาปัตยกรรม และการออกแบบ จะต้องมีการแลกเปลี่ยน (tradeoff) คุณสมบัติบางอย่างเพื่อประโยชน์บางอย่าง เช่น ข้อความนี้จะสามารถใช้ได้กับการออกแบบและสถาปัตยกรรมทุกอย่าง ไม่จำกัดเฉพาะ

-
- หากเปรียบเทียบกับการคิดคำนวณต่าง ๆ ของมนุษย์จะพบว่า การคิดคำนวณของมนุษย์ต้องใช้ความจำเช่นกัน เช่น การนึกภาพภาษาแบบเต็กลักษณะเดียวกัน หากเราต้องการคำนวณ 2+3 เราจำต้องนำ 2 และ 3 คำนึงมาใส่ในใจ ซึ่งหมายความว่า แล้วนับต่อจากนับ ซึ่งจะพบว่ามีการใช้ความจำควบคู่กับการคำนวณด้วยเสมอ
 - Verilog HDL เป็นภาษาสำหรับการบรรยายโครงสร้างและการออกแบบชาร์ดแวร์ (Hardware Description Language) มีโครงสร้างคล้ายภาษาซี เป็นที่นิยมใช้สำหรับบรรยายการออกแบบในสหรัฐอเมริกา (นอกจากนี้ นิสิตในหลักสูตรปริญญาบัณฑิต ของ ภาควิชาวิศวกรรมคอมพิวเตอร์ จุฬาลงกรณ์มหาวิทยาลัยทุกคน ยังต้องเรียนภาษา Verilog HDL ด้วย)

สถาปัตยกรรมคอมพิวเตอร์ท่านนี้ ตัวอย่างเช่น รถยนต์ที่ออกแบบให้ประหยัดน้ำมันจะแลกด้วยความสามารถในการเร่งความเร็วเครื่องยนต์ที่ต่ำลง เป็นต้น

ในทางสถาปัตยกรรมคอมพิวเตอร์ ผู้ออกแบบระบบคอมพิวเตอร์ (โดยเฉพาะอย่างยิ่ง หน่วยประมวลผลกลาง) จะเป็นจะต้องแลกเปลี่ยนคุณสมบัติบางอย่าง เพื่อให้หน่วยประมวลผลและระบบคอมพิวเตอร์สามารถทำงานอย่างได้ดีเข่นกัน เช่น หน่วยประมวลผลกลางที่สามารถทำงานด้านกราฟฟิกได้ดีมักจะกินพลังงานทำให้เกิดความร้อนสูง ในขณะที่อุปกรณ์พกพาขนาดเล็กที่ทำงานบนแบตเตอรี่มักจะไม่สามารถประมวลผลงานด้านกราฟฟิกได้มากนัก เป็นต้น

ด้วยเงื่อนไขนี้ การเลือกใช้งานระบบคอมพิวเตอร์จึงจำเป็นจะต้องพิจารณาถึงคุณสมบัติของหน่วยประมวลผลและระบบที่ใช้ ให้เหมาะสมกับงานที่ต้องการทำเช่นกัน เพราะไม่มีสถาปัตยกรรมคอมพิวเตอร์ใดที่ดีพร้อมในทุกด้าน (ซึ่งเป็นจุดประสงค์หนึ่งของการศึกษาสถาปัตยกรรมคอมพิวเตอร์ เช่นกัน)

1.8 แบบฝึกหัดท้ายบท

1. สถาปัตยกรรมคอมพิวเตอร์ คืออะไร
2. ส่วนประกอบของคอมพิวเตอร์ มีอะไรบ้าง
3. ในการพัฒนาโปรแกรมด้วยภาษาชั้นสูง เรามีขั้นตอนในการพัฒนาโปรแกรมต่าง ๆ เหล่านี้ อย่างไร จึงจะทำให้โปรแกรมถูกแปลงเป็นชุดคำสั่งสำหรับทำงานบนหน่วยประมวลผลได้
4. สถาปัตยกรรมชุดคำสั่งคืออะไร เป็นประโยชน์ในการพัฒนาซอฟต์แวร์และฮาร์ดแวร์หรือไม่ อย่างไร จงอธิบาย
5. (ทบทวนความรู้ด้าน logic design) จงสร้างวงจรบวกเลขที่จะบวกเลขขนาด 2 bit และ ตัวทดอีก 1 บิต แล้วให้คำตอบเป็นผลลัพธ์พร้อม ตัวทด
6. (ทบทวนความรู้ด้าน logic design) จงสร้างวงจรบวกเลขแบบอนุกรม (Serial Adder) แบบ Moore และ Mealy Machine

2 ประสิทธิภาพของระบบคอมพิวเตอร์

ประสิทธิภาพ คือ การวัดและการเปรียบเทียบการทำงาน การศึกษาเรื่องประสิทธิภาพของระบบคอมพิวเตอร์จะมุ่งเน้นถึงปัจจัยต่างๆ ที่ส่งผลกระทบต่อการทำงานของระบบคอมพิวเตอร์ โดยมุ่งให้สามารถเลือกใช้ หรือ รู้จักการวัดประสิทธิภาพของระบบ ทั้งนี้ เพื่อใช้ประกอบการศึกษาและการตัดสินใจว่า การพัฒนาประสิทธิภาพของระบบคอมพิวเตอร์ด้วยวิธีการใดได้ ให้ผลในทางปฏิบัติที่ดีขึ้นอย่างไร โดยในที่นี้จะกล่าวถึงในส่วนของการวัดในเชิงปริมาณเป็นหลัก

2.1 การวัดประสิทธิภาพของระบบคอมพิวเตอร์

การวัดประสิทธิภาพของระบบคอมพิวเตอร์ จะใช้เวลาเป็นหลักในการวัด โดยระบบคอมพิวเตอร์ที่ใช้เวลาในการทำงานหนึ่งน้อยกว่า จะมีประสิทธิภาพมากกว่าระบบคอมพิวเตอร์ที่ทำงานโปรแกรมเดียวกัน แต่ใช้เวลาในการทำงานมากกว่า หรืออาจจะกล่าวอีกนัยหนึ่งได้ว่า ประสิทธิภาพเป็นส่วนกลับของเวลา (สมการที่ 2.1) กล่าวคือ ใช้เวลาในการประมวลผลมาก ประสิทธิภาพต่ำ ใช้เวลาในการประมวลผลน้อย ประสิทธิภาพสูง

$$\text{Performance} \left(\frac{1}{\text{seconds}} \right) = \frac{1}{\text{Execution Time} \left(\text{seconds} \right)}$$

สมการที่ 2.1 ประสิทธิภาพและเวลา

การกล่าวว่า “เครื่องคอมพิวเตอร์ X เร็วกว่าเครื่อง Y เป็น n เท่า” หรือ “เครื่องคอมพิวเตอร์ X มีประสิทธิภาพเหนือกว่าเครื่อง Y เป็น n เท่า” นั้น สามารถอธิบายได้ดังนี้

$$\text{Speed up} (n) = \frac{\text{Performance}_x \left(\frac{1}{\text{seconds}} \right)}{\text{Performance}_y \left(\frac{1}{\text{seconds}} \right)}$$

เนื่องจากประสิทธิส่วนเป็นกลับของเวลา ดังนั้นการเปรียบเทียบประสิทธิภาพ อาจคำนวนได้จากเวลา เช่นกัน (สมการที่ 2.2)

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

สมการที่ 2.2 Speedup

เพื่อประกอบการอธิบาย ตัวอย่างที่ 2.1 แสดงการเปรียบเทียบประสิทธิภาพด้วยเวลา จากตัวอย่างจะสังเกตเห็นว่า เครื่องที่ใช้เวลามากกว่าจะมีประสิทธิภาพต่ำกว่า ในตัวอย่างนี้ นอกจากจะสรุปได้ว่า

เครื่อง A (ซึ่งใช้เวลาน้อยกว่า) มีประสิทธิภาพสูงกว่าเครื่อง B เป็น 1.25 เท่าแล้ว ยังสรุปได้อีกว่า เครื่อง A มีเร็วกว่าเครื่อง B อยู่ 1.25 เท่าเช่นกัน

ตัวอย่างที่ 2.1: การเปรียบเทียบประสิทธิภาพผ่านเวลา

จากการทดลองวัดเวลาในการประมวลผลของหน่วยประมวลผลกลางโดยใช้โปรแกรม benchmark ที่กำหนด ผลลัพธ์ที่ได้เป็นดังนี้

เครื่อง A ใช้เวลาในการทำงาน 20 วินาที

เครื่อง B ใช้เวลาในการทำงาน 25 วินาที

เครื่องคอมพิวเตอร์เครื่องใดมีประสิทธิภาพมากกว่ากัน และ มากกว่ากันเป็นกี่เท่า

เครื่อง A มีประสิทธิภาพมากกว่าเครื่อง B เนื่องจากสามารถประมวลผลโปรแกรมทดสอบได้เร็วกว่า

จาก

ได้ว่า

\therefore เครื่อง A มีประสิทธิภาพมากกว่าเครื่อง B เป็น 1.25 เท่า

ประเด็นที่วาย稼้งผู้เรียนหลายคนมักสับสนระหว่างประสิทธิภาพ และ เวลา หลักในการจำคือ หากประสิทธิภาพสูง หมายความว่า เร็ว ใช้เวลาต่ำหรือ น้อย และหากประสิทธิภาพต่ำหมายความว่า จะต้องใช้เวลาสูง หรือ มาก และหลักการนี้ ก็ช่วยให้คิดต่อได้ว่า หากบอกว่า A มีประสิทธิภาพสูงกว่า B อยู่ n เท่า ย่อมหมายความว่า B ใช้เวลามากกว่า A (มากกว่า) ดังนั้น n ย่อมมาจากการใช้เวลาของ B (ซึ่งมีค่ามากกว่า) หารด้วยเวลาของ A (ซึ่งมีค่าน้อยกว่า)

2.2 Response Time และ Throughput

แม้การใช้เวลาเป็นเกณฑ์จะช่วยให้การวัดประสิทธิภาพทำได้ง่ายขึ้น แต่เนื่องจากเวลาในการทำงานของระบบคอมพิวเตอร์ เพื่อประมวลผลโปรแกรม อาจหมายถึงเวลาในการตอบสนองของเครื่องคอมพิวเตอร์ (Response Time) ได้เช่นกัน เช่น เมื่อผู้ใช้สั่งให้โปรแกรมค้นหาข้อมูล โปรแกรมใช้เวลานานเท่าใด หลังจากเริ่มค้นหา และได้ผลลัพธ์แสลงออกมาก หรือ อาจหมายถึง ความสามารถในการประมวลผลงานต่างๆ ในช่วงเวลาหนึ่ง (Throughput) เช่น เครื่องคอมพิวเตอร์ สามารถประมวลผลโปรแกรมได้ 2 โปรแกรมภายในเวลา 1 วินาที เป็นต้น ซึ่ง Response Time และ Throughput

อาจจะมีได้เป็นไปในทิศทางเดียวกัน

มีความเป็นไปได้ว่าเครื่องที่มี Throughput สูง อาจจะมี Response Time ต่ำได้ เช่น เครื่องคอมพิวเตอร์ประมวลผลงานเสร็จ 5 งานภายใน 10 นาที หากแต่ เวลาที่ใช้ในการประมวลผลงานทั้ง 5 นั้น ก็เฉลี่ยประมาณ 10 นาที เช่นกันโดยการประมวลผลของทั้ง 5 งานเกิดขึ้นพร้อมๆ กัน ในขณะที่ อีกเครื่องหนึ่ง ใช้เวลาในการประมวลผลงานละ 3 นาที หากแต่ทำได้เพียงทีละ 1 งานหมายความว่า จะต้องใช้เวลาทำงานรวมกันทั้ง 5 งานเป็น 15 นาที เป็นต้น จากตัวอย่างดังกล่าว จะพบว่าเครื่องคอมพิวเตอร์เครื่องแรก จะมี throughput มากกว่าเครื่องที่สอง ในการที่เครื่องที่สองจะมี Response Time ต่ำกว่าเครื่องแรก

นอกจากนี้ หากวิเคราะห์การทำงานของซอฟต์แวร์ต่างๆ ภายในเครื่องคอมพิวเตอร์จะพบว่า เครื่องคอมพิวเตอร์ไม่ได้ใช้เวลาทั้งหมดไปกับการประมวลผล แต่เวลาบางส่วนอาจสูญเสียไปเนื่องจากการทำงานของฮาร์ดแวร์ส่วนอื่น เช่น การอ่านข้อมูลจากหน่วยความจำ หรือ ระบบ I/O อื่นๆ ดังนั้น การวิเคราะห์ Response Time หรือ Execution Time จึงสามารถแบ่งย่อยออกได้เป็น

- CPU Time
- Elapse Time

CPU Time คือเวลาที่ CPU ทำงานเพื่อประมวลผลโปรแกรมหนึ่งๆ โดยไม่นับรวมเวลาจากการรือคุยกับการทำงานจากอุปกรณ์ต่อพ่วงที่อยู่ภายนอกหน่วยประมวลผลกลาง

Elapse Time คือเวลาในการประมวลผลโปรแกรมหนึ่งๆ ของระบบคอมพิวเตอร์ ซึ่งรวมถึงเวลาที่ใช้ในการอ่านข้อมูลจาก I/O (I/O Time) และ หน่วยความจำ(Memory Time) ซึ่งอาจจะสรุปได้เป็น สมการอย่างง่ายๆ ว่า

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

สมการที่ 2.3 Elapse Time

ทั้งนี้ในการวัดประสิทธิภาพของระบบคอมพิวเตอร์นั้น บางครั้งจะนิยมวัดด้วย CPU Time แทนการวัดด้วย Elapse Time เนื่องจาก หากวัดประสิทธิภาพโดยใช้ Elapse Time ในการอ้างอิง ค่าของประสิทธิภาพที่วัดได้ อาจแปรเปลี่ยนตามความเร็วของการเข้าถึงข้อมูลในหน่วยความจำ หรือ ความเร็วในการตอบสนองของอุปกรณ์ต่อพ่วงต่างๆ ทำให้บางครั้งไม่สามารถนำค่าที่วัดได้ มาวิเคราะห์ในเชิงเปรียบเทียบ โดยเฉพาะอย่างยิ่ง เนื้อหาในบทเรียนนี้มุ่งเน้นที่ความสามารถของหน่วยประมวลผลกลางเป็นหลัก (จากสถิติพบว่า โปรแกรมส่วนใหญ่จะใช้เวลามากกว่า 45% ในการทำงานงานของ I/O หรือส่วนอื่นๆ ที่ไม่เกี่ยวข้องกับการทำงานของโปรแกรม)

2.3 Benchmark

เนื่องจากการวัดประสิทธิภาพของระบบคอมพิวเตอร์ ต้องอาศัยการวัดเวลาจากการทำงานจริงของ

โปรแกรม เพื่อให้การวัดความสามารถเปรียบเทียบกันได้ จึงจำเป็นจะต้องใช้โปรแกรมทดสอบที่เป็นมาตรฐาน (Benchmark) ทั้งนี้โปรแกรมที่จะใช้ทดสอบ อาจเป็นโปรแกรมใดก็ได้ แต่หากผู้พัฒนาแต่ละราย ทำการทดสอบประสิทธิภาพโดยใช้โปรแกรมทดสอบของตนเอง ผลลัพธ์ที่ได้จากการทดสอบ จะไม่สามารถเปรียบเทียบกันได้ หรือเปรียบเทียบกันได้ยาก

ด้วยเหตุนี้ การเลือกใช้ benchmark ที่เหมาะสม จึงเป็นสิ่งสำคัญต่อการประเมินประสิทธิภาพ แนวทางในการเลือกใช้โปรแกรมสำหรับเป็น benchmark ที่เหมาะสม เช่น

- การใช้โปรแกรมจริงซึ่งใช้ในการทำงานสำหรับการวัดประสิทธิภาพ (real benchmark) เช่น word processing, compilers
- การใช้ **บางส่วนของโปรแกรม** เพื่อวัดประสิทธิภาพ (microbenchmark) เช่น การนำเฉพาะส่วนการคำนวณของโปรแกรมมาใช้ทดสอบ
- การใช้โปรแกรมเฉพาะเพื่อทดสอบบางส่วนของยาร์ดแวร์ (kernel)
- การใช้โปรแกรมสังเคราะห์เพื่อสร้างการประมวลผลสมมุติ (synthetic benchmark)
- การใช้โปรแกรมพื้นฐานทางอัลกอริทึมในการทดสอบ (toy benchmark) เช่น การใช้โปรแกรม quick sort หรือ merge sort ในการทดสอบความเร็ว

จะสังเกตุพบว่า ประสิทธิภาพที่วัดได้จะแตกต่างกันไปตามการเลือก benchmark เช่นกัน เพื่อให้การวัดประสิทธิภาพสามารถเปรียบเทียบกันได้ ไม่มีการโน้มเอียงไปทางใดทางหนึ่ง กลุ่มผู้ผลิตระบบคอมพิวเตอร์จึงได้รวมตัวกันและกำหนด benchmark กลาง ที่เป็นยอมรับสำหรับการทดสอบประสิทธิภาพขึ้น (เรียกว่า SPEC) รายละเอียดสามารถศึกษาได้ในเนื้อหาส่วนถัดไป

2.3.1 SPEC Benchmarks

เพื่อให้การทดสอบมีมาตรฐาน สามารถเปรียบเทียบกันได้ ผู้ผลิตหน่วยประมวลผลกลางและระบบคอมพิวเตอร์จึงได้มีการรวมกลุ่มกัน เพื่อกำหนดประเภทและชุดซอฟต์แวร์ทดสอบที่เหมาะสมสำหรับการวัดประสิทธิภาพของระบบคอมพิวเตอร์ ชุดซอฟต์แวร์ทดสอบเหล่านี้ เรียกว่า SPEC Benchmark suites ซึ่งเผยแพร่และร่วบรวมโดย System Performance Evaluation Cooperative ภายใต้ชื่อซอฟต์แวร์ทดสอบ จจะแบ่งย่อยออกเป็นซอฟต์แวร์ทดสอบด้านต่างๆ เช่น SPEC int สำหรับทดสอบการประมวลผลทั่วไป SPEC web สำหรับทดสอบระบบ Web Server เป็นต้น

ตัวอย่างของชุด benchmark ที่มีอยู่ใน SPEC³ ปัจจุบัน เช่น

- **CPU** สำหรับทดสอบประสิทธิภาพงานที่มีความต้องใช้หน่วยประมวลผลกลางสูง ประกอบไปด้วยชุดโปรแกรมด้าน ปัญญาประดิษฐ์ อัลกอริทึม งานบีบอัดข้อมูล งานบีบอัดภาพ /วิดีโอ การประมวลผลเสียง การแปลงภาษาคอมพิวเตอร์ (compilers และ interpreters) เป็นต้น

³ รายละเอียดเพิ่มเติมสามารถค้นคว้าได้จาก <http://www.spec.org/>

- **Graphics/Workstations** สำหรับทดสอบประสิทธิภาพของการงานทางด้าน Graphics ซึ่งสร้างขึ้นจากความร่วมมือของผู้ผลิตอุปกรณ์ประมวลผลด้าน graphics
- **Java Client/Server** สำหรับทดสอบประสิทธิภาพของ Java Virtual Machine เป็นหลัก ซึ่งจาก Java เป็นภาษาคอมพิวเตอร์ที่ได้รับความนิยมสูงในงานองค์กรนี้ แต่ประสิทธิภาพของ Java ขึ้นอยู่กับความสามารถของ Java Virtual Machine เป็นหลัก ดังนั้น กลุ่มผู้พัฒนา Java Virtual Machine จึงกำหนดมาตรฐานสำหรับการวัดประสิทธิภาพของ Java ขึ้น เพื่อเป็นแนวทางในการพัฒนาคุณภาพของ Java Virtual Machine

2.3.2 คุณลักษณะของ benchmark ที่ดี

เนื่องจาก benchmark ถูกออกแบบเพื่อใช้เป็นตัวแทนในการวัดประสิทธิภาพของระบบคอมพิวเตอร์ ดังนั้น การเลือกว่าจะใช้โปรแกรมชุดใดเป็น benchmark ควรจะเลือกด้วยความระมัดระวัง ไม่เลือกโปรแกรมที่จะส่งผลให้เกิดความเออนเอียงในการประเมินประสิทธิภาพ ด้วยเงื่อนไขดังกล่าวจึงสรุปได้ว่า **benchmark ที่ดี** ควรจะมีคุณสมบัติเบื้องต้น ดังต่อไปนี้

1. เป็นโปรแกรมที่มีอยู่จริง และมีการใช้งานจริง ทั้งนี้หากนำโปรแกรมสังเคราะห์มาใช้ อาจทำให้ผลที่ได้ ไม่สอดคล้องกับลักษณะงานใช้งานจริง
2. สามารถนำมาวัดได้บนชาร์ดแวร์ที่กำหนด ในกรณีนี้หาก benchmark ที่เลือก ไม่สามารถนำมา compiler และประมวลผลได้บนระบบที่ต้องการทดสอบ การทดสอบย่อมจะเกิดขึ้นไม่ได้
3. เป็นตัวแทนของงานในลักษณะคล้ายคลึงกัน และเป็นตัวแทนเทคโนโลยีที่เกี่ยวข้อง ตัวอย่างเช่น นักพัฒนาซอฟต์แวร์ทุกคน มักจะต้องทำการ compile ซอฟต์แวร์ ดังนั้นหากมี benchmark เป็น compiler ที่เป็นที่ยอมรับในงานน้อยอย่างแพร์ helyay ย่อมส่งผลให้การวัดประสิทธิภาพใกล้เคียงกับความเป็นจริงมากขึ้น
4. มีการคำนวณที่ซัดเจนคาดเดาได้ มีบางกรณีที่การทำงานบางอย่างของโปรแกรมไม่สามารถคาดเดาได้ เช่น โปรแกรมมีการทำงานแบบสุ่ม ไม่สามารถประเมินจำนวนครั้งหรือ รอบการทำงานที่แน่นอนได้ โปรแกรมในลักษณะนี้ จะทำให้การวัดผลแต่ละครั้งคลาดเคลื่อน สรุปการทำงานได้ยาก

อย่างไรก็ตาม นี่เป็นเพียงคุณสมบัติบางส่วนเท่านั้น และมีใช้ว่า benchmark อันหนึ่งอันใด จะมีคุณสมบัติครบถ้วน ก็จะมี benchmark ที่เกิดจากการสังเคราะห์ เพื่อทดสอบ ชาร์ดแวร์หรือการคำนวณแบบพิเศษเสมอ ซึ่งอาจไม่ใช้โปรแกรมที่มีการใช้งานจริงแต่อย่างใด

2.4 การเปรียบเทียบประสิทธิภาพ

ถึงตรงนี้ จะเห็นว่าการวัดประสิทธิภาพของระบบคอมพิวเตอร์ทำได้โดยการนำชุดโปรแกรมที่เหมาะสม สม (หรือที่ถูกออกแบบไว้ว่าจะเป็นมาตรฐานในการวัด) มาทดสอบประมวลผลและจับเวลา อย่างไรก็ตาม อาจมีความเป็นไปได้ว่า ผลที่ได้อาจจะไม่เป็นไปในทิศทางเดียวกันเสมอไป เช่น ในชุดโปรแกรม

ที่ทดสอบอาจจะมีโปรแกรมอยู่ 2 โปรแกรม ซึ่งระบบคอมพิวเตอร์หนึ่งอาจจะให้ผลบางค่าดีกว่าอีกระบบหนึ่ง หากเป็นเช่นนี้ การจะสรุปว่าระบบคอมพิวเตอร์ใดมีประสิทธิภาพดีกว่า ย่อมจะทำได้ยาก (ดูตัวอย่างในตารางที่ 2.1)

	Computer A (วินาที)	Computer B (วินาที)	Computer C (วินาที)
Program 1	10	4	5
Program 2	2	10	5
Arithmetic Mean	6	7	5

ตารางที่ 2.1: เปรียบเทียบเวลาได้จาก benchmark

จากตาราง การจะตัดสินว่าคอมพิวเตอร์ระบบใดดีกว่าอย่างจะเป็นไปได้ยาก หากไม่มีข้อมูลเพิ่มเติม คำตอบที่ดูจะเหมาะสมจากค่าเฉลี่ยคณิตศาสตร์⁴ (Arithmetic Mean) อาจจะเป็น Computer C (เมื่อคิดว่าในระบบคอมพิวเตอร์มีการใช้งาน Program 1 และ Program 2 ในสัดส่วนที่ใกล้เคียงกัน) แต่หากมีข้อมูลเพิ่มเติม เช่น ทำการใช้งานส่วนใหญ่มักจะเป็นการใช้งาน Program 2 ถึง 80% การเลือกใช้ Computer A ดูจะเป็นคำตอบที่ดีที่สุด (ตามตารางที่ 2.2) เป็นต้น ซึ่งการสรุปในลักษณะนี้ เป็นการใช้ ค่าเฉลี่ยคณิตศาสตร์แบบถ่วงน้ำหนัก⁵ (Weighted Arithmetic Mean) มาเป็นเครื่องมือช่วยในการตัดสินใจ

	Computer A (วินาที)	Computer B (วินาที)	Computer C (วินาที)
Program 1	10	4	5
Program 2	2	10	5
Weighted Arithmetic Mean	1.8	4.4	2.5

ตารางที่ 2.2: การใช้ค่าเฉลี่ยคณิตศาสตร์ถ่วงน้ำหนักเปรียบเทียบเวลาได้จาก benchmark
(เปรียบเทียบเมื่อ Program 2 มีการใช้งานคิดเป็น 80% และ Program 1 มีการใช้งานคิดเป็น 20%)

ในทางปฏิบัติ อาจจะไม่สามารถสรุปได้ว่าจะมีการใช้งานโปรแกรมใดในสัดส่วนเท่าไหร และการบอก

4 ค่าเฉลี่ยคณิตศาสตร์ (Arithmetic Mean) คำนวณได้จาก
$$\frac{\sum^n x_i}{n}$$

5 ค่าเฉลี่ยคณิตศาสตร์แบบถ่วงน้ำหนัก (Weighted Arithmetic Mean) คำนวณได้จาก
$$\frac{\sum^n w_i \times x_i}{n}$$

ประสิทธิภาพด้วยหน่วยเวลา ยังไม่ใช่การบอกประสิทธิภาพในเชิงเปรียบเทียบที่เหมาะสมอีกเช่นกัน เพราะทั้งนี้ เวลาขึ้นอยู่กับปริมาณงาน (workload) ด้วย เหตุนี้การเปรียบเทียบที่ดีกว่าคือการเปรียบเทียบโดยใช้ Speedup (สมการที่ 2.2) ตัวอย่างการเปรียบเทียบแสดงได้ดังตารางที่ 2.3 ซึ่งการรายงานผลเชิงเปรียบเทียบลักษณะนี้ จะเป็นการรายงาน **เปรียบเทียบกับระบบอ้างอิง (baseline)** เช่น กรณีตัวอย่างในสมมุติให้ Computer A เป็นระบบอ้างอิง

อ้างໄரก์ตาม ยังมีตัวเลขที่เกี่ยวข้องมากกว่า หนึ่งค่าในการเปรียบเทียบ เพราะมีโปรแกรมในชุดของ benchmark อยู่หลายโปรแกรมที่เกี่ยวข้อง เพื่อให้การรายงานผลลัพธ์ดูกันชัดเจน จึงมีแนวคิดที่จะใช้ค่าเฉลี่ยในการสรุปผลแทน ทั้งนี้หากใช้ค่าเฉลี่ยคณิตศาสตร์โดยตรง จะพบว่าค่าเฉลี่ยที่ได้จะไม่สอดคล้องกับข้อมูลที่แท้จริง (ดูตารางที่ 2.3) เพราะค่าเฉลี่ยคณิตศาสตร์ จะเหมาะสมสำหรับการหาค่าเฉลี่ยเฉพาะกรณีข้อมูลที่เปรียบเทียบ มีหน่วยหน่วยเดียวกัน จึงไม่เหมาะสมที่จะนำมาใช้กับข้อมูลที่เป็นอัตราส่วน (ซึ่งไม่มีหน่วย หรือหน่วยตัดกันไปแล้ว) จากตัวอย่างจะเห็นว่า หากนำ A/B มาหาค่าเฉลี่ยจะได้เป็น 1.35 ซึ่งไม่สอดคล้องกับค่าเฉลี่ย A หารด้วย ค่าเฉลี่ย B ซึ่งจะมีค่าเป็น 0.85 (คำนวนได้จาก 6 หารด้วย 7)

	Computer A (วินาที)	Computer B (วินาที)	A/B	Computer C (วินาที)	A/C
Program 1	10	4	2.5	5	2
Program 2	2	10	0.2	5	0.4
Arithmetic Mean	6	7	1.35	5	1.2

ตารางที่ 2.3: การเปรียบเทียบประสิทธิภาพด้วยอัตราส่วน พร้อมค่าเฉลี่ยคณิตศาสตร์

ด้วยเหตุนี้การรายงานผลที่ดีกว่าคือการใช้ค่าเฉลี่ยเรขาคณิต⁶ (Geometric Mean) ทั้งนี้เนื่องจากค่าเฉลี่ยเลขคณิต จะให้ค่าที่เหมาะสมเมื่อข้อมูลอยู่ในลักษณะของอัตราส่วน (เช่น A/B ซึ่งไม่มีหน่วยหรือมีหน่วยเป็นเท่า) ตารางที่ 2.4 แสดงการใช้ค่าเฉลี่ยเรขาคณิตในการประมาณผล จากตัวอย่างจะเห็นได้ว่า การนำค่าเฉลี่ยเลขคณิตมาเปรียบเทียบ (X/Y) ให้ผลลัพธ์เหมือนกับการนำเวลามาเปรียบเทียบโดยตรง (C/B) ซึ่งค่าเฉลี่ยเลขคณิตที่ได้จากการเปรียบเทียบมีค่าเท่ากันเช่นกัน

6 ค่าเฉลี่ยเรขาคณิต (Geometric Mean) คำนวนได้จาก

$$\sqrt[n]{\prod^n x_i}$$

	A (วินาที)	B (วินาที)	A/B (X)	C (วินาที)	A/C (Y)	C/B	X/Y
Program 1	10	4	2.5	5	2	1.25	1.25
Program 2	2	10	0.2	5	0.4	0.5	0.5
Geometric Mean			0.71		0.89	0.79	0.79

ตารางที่ 2.4: การเปรียบเทียบประสิทธิภาพด้วยอัตราส่วน พร้อมค่าเฉลี่ยเรขาคณิต

การเปรียบด้วยด้วยค่าเฉลี่ยเรขาคณิตนี้ เป็นแนวทางมาตรฐานที่ใช้ในการรายงานผลประสิทธิภาพของ SPEC.org⁷ และเป็นวิธีการที่ใช้หัวไปเมื่ออ้างอิงถึงประสิทธิภาพ อย่างไรก็ตามในทางคณิตศาสตร์ยังมีการหาค่าเฉลี่ยแบบอื่นอีก (เช่น ค่าเฉลี่ยข้ามมิค (Harmonic Mean) สำหรับการหาค่าเฉลี่ยกรณีที่ข้อมูลมีหน่วยเป็นอัตรา) ซึ่งมีได้กล่าวถึงในที่นี้

2.5 การหาเวลา CPU Time

ในทางปฏิบัตินั้น บางครั้งการวัดหรือจับเวลา CPU Time ของการประมวลโปรแกรมไม่สามารถทำได้โดยง่าย เนื่องจากบางครั้งเวลาที่รัดໄต้จังกล่า ไม่สามารถแยกໄต้ไว้ เป็น CPU Time หรือ เป็นเวลาการทำงานที่ร่อจาก I/O ดังนั้นการหา CPU Time จึงอาจทำได้โดยการประมาณหรือการนับจำนวนคำสั่งที่ใช้ในการประมวลผล และ หาผลรวมของเวลาที่ใช้ในการประมวลผลคำสั่งทั้งหมด ดังแสดงในตัวอย่างที่ 2.2

จากตัวอย่างที่ 2.2 พบร่วมกัน ทราบว่า เวลาที่ใช้ในการประมวลผลคำสั่งจะคำนวนได้จากผลคูณของจำนวนคำสั่ง (Instruction Count) และเวลาเฉลี่ยในการทำงานของแต่ละคำสั่ง (Average Cycle per Instruction) ในกรณีตัวอย่างนี้ ทุกคำสั่งใช้เวลาเท่ากันคือ 0.1 วินาที ซึ่งในทางปฏิบัติ หากแต่ละคำสั่งใช้เวลาไม่เท่ากัน การประเมินความมีการถ่วงน้ำหนักเพื่อให้ได้เวลาเฉลี่ยที่ใกล้เคียงกับความเป็นจริงหรือทำการคำนวนแยกแล้วนำผลที่ได้มาบวกรวมกันอีกรอบหนึ่ง

ตัวอย่างที่ 2.2: การประมาณ CPU จากการคำนวน

เครื่องคอมพิวเตอร์เครื่องหนึ่งมีเวลาในการประมวลผลคำสั่งแต่ละคำสั่งเป็น 0.1 วินาที หากนำโปรแกรมที่มีจำนวนคำสั่ง 10 คำสั่งมากทำการประมวลผลบนเครื่องดังกล่าว จะใช้เวลาในการทำงานเป็นเท่าไร

$$\text{เวลาในการทำงานทั้งหมด} = 10 \times 0.1 = 1 \text{ วินาที}$$

อย่างไรก็ตาม การบวกเวลาในการประมวลผลของคำสั่งแต่ละคำสั่งนั้น ไม่นิยมบวกเป็นหน่วยวินาที

7 <http://www.spec.org/>

ทั้งนี้เนื่องจากเวลาในการประมวลผลของหน่วยประมวลผลนั้น จะขึ้นอยู่กับสัญญาณนาฬิกาที่ป้อนให้กับหน่วยประมวลผลนั้น (**Clock Rate**) ซึ่งช่วงเวลาเหล่านี้ นิยมเรียกว่า ticks หรือ clock cycles โดย เวลา 1 **clock cycle** คือเวลา 1 คานของการประมวลผล การนับเวลาเป็นหน่วย ticks หรือ clock cycles มีข้อดีที่ทำให้การประเมินค่าไม่จำเป็นต้องยุ่งเกี่ยวกับความถี่สัญญาณนาฬิกาของระบบ ช่วยให้การนับหรือคำนวณสะดวกมากขึ้น โดยเฉพาะเมื่อเครื่องที่ต้องการประเมินมีความถี่สัญญาณนาฬิกาเท่ากัน

ความสัมพันธ์ระหว่าง clock rate และ clock time สามารถแสดงได้ดังสมการที่ 2.4 ด้วย式ที่ 2.4 ตัวอย่างเช่น หน่วยประมวลผลที่มี Clock Rate เป็น 250Mhz จะมี Cycle Time เป็น 4 nanoseconds เป็นต้น (ทั้งนี้ต้องยุ่บลงสมมุติฐานว่า 1 Clock Cycle คือ 1 Cycle Time)

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

สมการที่ 2.4 Cycle Time

นอกจากนี้การประมวลผลคำสั่ง อาจมีได้ใช้เวลาเพียง 1 Clock Cycle Time เท่านั้น จำนวน Cycle ที่ใช้ในการประมวลผลแต่ละคำสั่ง (**Cycle Per Instruction** หรือ CPI) อาจแตกต่างกันไป เช่นการประมวลผลคำสั่งบวกอาจใช้เวลา 1 CPI และการประมวลผลคำสั่งคูณอาจใช้เวลา 3 CPI ดังนั้นหากหน่วยประมวลผลกลางดังกล่าวมีความถี่สัญญาณนาฬิกาเป็น 150Mhz เวลาที่ใช้ในการทำคำสั่งบวกจะเป็น 4 nanoseconds (1^*4) และเวลาที่ใช้ในการทำคำสั่งคูณเท่ากับ 12 nanoseconds (3^*4)

เมื่อนำแนวความคิดทั้งหมดมาประสานเข้าด้วยกัน สามารถสรุปเป็นสมการแสดงความสัมพันธ์ระหว่างเวลาที่ใช้ในการประมวลผล และ ความถี่สัญญาณนาฬิกาได้ดัง สมการที่ 2.5 สมการที่ 2.6 และ สมการที่ 2.7 ตามลำดับ ซึ่งเมื่อนำ สมการที่ 2.5 และ สมการที่ 2.7 จะได้สมการสุดท้ายเป็น

$$\text{CPU Time per instruction (seconds)} = \text{CPI} \times \text{Cycle Time}$$

สมการที่ 2.5 CPU Time/instruction

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

สมการที่ 2.6

ดังนั้นเวลาในการทำงานของ CPU (CPU Time) สำหรับ 1 โปรแกรม จะเป็น

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

สมการที่ 2.7 CPU Time

$$\text{หรือ } CPU\ Time\ (seconds) = Instruction \times CPI \times Cycle\ Time$$

สมการที่ 2.8 CPU Time

จะเห็นว่า เวลาในการประมวลผลสามารถหาได้จากการนับจำนวนคำสั่ง และการคำนวณเบื้องต้น เพื่อประกอบความเข้าใจ ตัวอย่างที่ 2.3 แสดงการหา CPU Time ด้วยการคำนวณจากจำนวนคำสั่ง ความถี่สัญญาณนาฬิกา และจำนวน Cycle ที่ใช้ในการประมวลผลแต่ละคำสั่ง

ตัวอย่างที่ 2.3: การคำนวณหา CPU Time

เครื่องคอมพิวเตอร์เครื่องหนึ่งมีความถี่สัญญาณนาฬิกา 100MHz ใช้เวลาในการประมวลผลโปรแกรมทดสอบซึ่งมีจำนวนคำสั่งทั้งสิ้น 2000 ล้านคำสั่ง เป็นเวลาทั้งสิ้น 20 วินาที หากผู้ออกแบบระบบสามารถพัฒนาให้หน่วยประมวลผลกลางใช้ความถี่สัญญาณนาฬิกาเป็น 200MHz และใช้จำนวน CPI มากขึ้นเป็น 1.2 เท่า หากนำโปรแกรมทดสอบนี้มาทดสอบในหน่วยประมวลผลที่ทำการพัฒนา จะใช้เวลาในการประมวลผลเป็นเท่าไร (กำหนดให้แต่ละคำสั่งใช้เวลาในการประมวลผลเท่ากัน)

ความถี่สัญญาณนาฬิกา 100MHz คิดเป็น

จาก

$$\therefore [CPI] = 1$$

การพัฒนาหน่วยประมวลผล ทำให้ใช้ CPI เพิ่มขึ้นเป็น 1.2 เท่า

$$\therefore [CPI_{ใหม่}] = 1 \times 1.2 = 1.2 \text{ Cycle per instruction}$$

เวลาที่ใช้ในการประมวลผลโปรแกรมทดสอบหลังจากพัฒนาหน่วยประมวลผลกลาง

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

ในบางกรณี ผู้ผลิตระบบคอมพิวเตอร์ (โดยเฉพาะเครื่องคอมพิวเตอร์สมรรถนะสูงในอุตสาหกรรม) นิยมที่จะ

กล่าวถึงความเร็วในการประมวลผลทั่วไป โดยใช้หน่วย MIPS⁸ (Million of Instructions per second) เช่น โปรแกรมมีการทำงานทั้งสิ้น 2000 ล้านคำสั่งภายในเวลา 20 วินาที อาจกล่าวได้ว่า ความเร็วในการประมวลผลของหน่วยประมวลผลดังกล่าวเป็น $2000/20 = 100$ MIPS เป็นต้น ซึ่งมีความหมายว่า หน่วยประมวลผลดังกล่าวสามารถประมวลผลได้ 100 ล้านคำสั่งภายในเวลา 1 วินาที ในเบื้องต้นการบอกความเร็วด้วยหน่วย MIPS อาจดูสอดคล้อง (เป็นส่วนกลับของ) กับการใช้ CPI (Cycles Per Instruction) และ Cycle Time อย่างไรก็ตามหากวิเคราะห์ให้ละเอียดจะพบว่า ค่า MIPS มีได้แสดงถึงประสิทธิภาพที่แท้จริงของระบบแต่อย่างใด เพราะค่า MIPS มีได้ล่าร์วถึงจำนวนคำสั่งภายในระบบเท่านั้น หรือไม่ ก็ต้องใช้เวลา 2.4

ตัวอย่างที่ 2.4: การเปรียบเทียบประสิทธิภาพในหน่วย MIPS

กำหนดให้ เครื่องคอมพิวเตอร์ A มีความเร็ว 10 MIPS ทำงานที่ 33 Mhz และ เครื่องคอมพิวเตอร์ B มีความเร็ว 9 MIPS ทำงานที่ 20 Mhz หากนำโปรแกรมทดสอบมาทำการ compile เพื่อทดสอบบนเครื่อง A ได้จำนวนคำสั่งเป็น 12,000,000 คำสั่ง และ บนเครื่อง B ได้ 10,000,000 คำสั่ง เครื่องคอมพิวเตอร์เครื่องใดมีประสิทธิภาพมากกว่ากัน เมื่อเปรียบเทียบด้วยการทำงานของโปรแกรมทดสอบที่กำหนด

จากคำจำกัดความของ MIPS จะได้ว่า

เนื่องจาก 1 Cycle ใช้เวลาเป็น 1 Cycle Time ดังนั้น ส่วนกลับของผลคูณของ MIPS (n) กับ Cycle Time จะผลลัพธ์เป็น Cycle Per Instruction

(MIPS แทนจำนวนคำสั่งใน 1 วินาที ในขณะที่ 1 Cycle คิดเป็น Cycle Time (วินาที) ดังนั้น ผลคูณของ MIPS และ Cycle Time จึงได้จำนวน (ล้าน) คำสั่งใน 1 Cycle Time)

นำค่าที่ได้มาคำนวณหา CPU Time จะได้ว่า

เครื่องคอมพิวเตอร์ A

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

เครื่องคอมพิวเตอร์ B

จากตัวอย่างดังกล่าว ได้ว่า เครื่องคอมพิวเตอร์ A ใช้เวลาในการประมวลผล 1.2 วินาที ในขณะที่ เครื่องคอมพิวเตอร์ B ใช้เวลาในการประมวลผล 1.11 วินาที

∴ เครื่องคอมพิวเตอร์ B มีประสิทธิภาพสูงกว่าเครื่องคอมพิวเตอร์ A

หากจะสรุปต่อว่า เครื่องคอมพิวเตอร์ B เร็วกว่า เครื่องคอมพิวเตอร์ A กี่เท่า สามารถทำได้โดยนำ สมการที่ 2.2 มาประยุกต์ใช้ ดังนี้

ข้อสรุปที่ได้จากตัวอย่างนี้คือ ค่า MIPS มีค่าตัวชี้วัดประสิทธิภาพที่น่าเชื่อถือแต่อย่างใด

2.6 กฎของ Amdahl

ในการปรับปรุงประสิทธิภาพส่วนใดส่วนหนึ่งของระบบคอมพิวเตอร์ (เช่น การเพิ่มประสิทธิภาพในการบวกเลขของหน่วยประมวลผลกลางให้เร็วขึ้น 2 เท่า) มีได้หมายความว่า เมื่อนำโปรแกรมทดสอบเข้าไป ทดลองประมวล จะได้ประสิทธิภาพเพิ่มขึ้นเป็น 2 เท่า ทั้งนี้เนื่องจากเวลาในการประมวลผลทั้งหมด นั้น ไม่ใช้เวลาที่เกิดขึ้นจากการบวกเลข หากแต่เป็นเวลารวมของการคูณเลข การย้ายข้อมูล หรือการ เปรียบเทียบทางตระร่วมด้วย เพื่อเป็นการประเมินแนวทางการปรับปรุงประสิทธิภาพของระบบ คอมพิวเตอร์ จึงมีแนวคิดในการวัดค่า Speedup เพื่อหากประสิทธิภาพโดยรวมของระบบหลังการ

ปรับปรุง เราเรียกกฎที่ช่วยในการประเมินประสิทธิภาพดังกล่าวว่า กฎของ Amdahl⁹

กฎของ Amdahl (บางตำราเรียกว่า law of diminishing return) ได้กำหนดการวัด Speedup หรือประสิทธิภาพที่เพิ่มขึ้นโดยกล่าวไว้ว่า หากมีการพัฒนาให้ส่วนหนึ่งส่วนใดของระบบมีความสามารถมากขึ้น ผลกระทบของประสิทธิภาพที่เพิ่มขึ้นจากการปรับปรุงส่วนหนึ่งส่วนใดนั้น จะเป็นสัดส่วนของส่วนที่ไม่ได้รับการปรับปรุง รวมกับส่วนที่ได้รับการปรับปรุง

ข้อความนี้ อาจจะทำความเข้าใจได้ยาก แต่หากลองพิจณาดูสมการที่ 2.2 จะพบว่า

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

ทั้งนี้เวลาที่ใช้ในการประมวลผลโปรแกรมทดสอบ หลังจากที่ได้ทำการพัฒนานั้น สามารถหาได้จากความสัมพันธ์ดังสมการที่ 2.9 กล่าวคือ เวลาที่ใช้ในการประมวลผลหลังปรับปรุงประสิทธิภาพ ($\text{Execution Time}_{\text{new}}$) คิดเป็น ผลรวมของเวลาประมวลผลส่วนที่ไม่ได้รับการปรับปรุง ($\text{Execution Time}_{\text{no enhancement}}$) กับ เวลาส่วนที่ได้รับการปรับปรุง ($\text{Execution Time}_{\text{enhancement}}$) หารด้วยอัตราส่วนที่ได้รับการปรับปรุง (enhanced ratio)

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

สมการที่ 2.9 Execution Time

หากวิเคราะห์ต่อไป จะสังเกตุพบว่า ผลรวมของ เวลาประมวลผลส่วนที่ไม่ได้รับการปรับปรุง ($\text{Execution Time}_{\text{no enhancement}}$) กับ เวลาส่วนที่ได้รับการปรับปรุง ($\text{Execution Time}_{\text{enhancement}}$) นั้น ความจริงก็คือ เวลาที่ใช้ในการประมวลผลก่อนปรับปรุงประสิทธิภาพ ($\text{Execution Time}_{\text{old}}$) นั่นเอง หากสมมุติให้ส่วนที่ได้รับการปรับปรุงมีสัดส่วนเป็น P และ อัตราส่วนที่ได้รับการปรับปรุง (enhanced ratio) มีค่าเป็น n เราจะเขียน สมการที่ 2.2 และ สมการที่ 2.9 รวมกันใหม่ได้เป็นกฎของ Amdahl ดังสมการที่ 2.10

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

⁹ กฎของ Amdahl คิดค้นโดย Gene M. Amdahl อดีตผู้ทำงานด้านระบบ Mainframe ของ IBM ซึ่งต่อมาได้ก่อตั้งบริษัท Amdahl Corporation (ปัจจุบันเป็นส่วนหนึ่งของ บริษัท Fujitsu) กฎของ Amdahl มีความสำคัญเป็นรายงานของการพัฒนาระบบสถาปัตยกรรมคอมพิวเตอร์ในปัจจุบัน ที่มา: บางส่วนคัดมาจากราบ wikipedia [3] (สืบค้นข้อมูลเมื่อ มกราคม พ.ศ. 2557)

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

สมการที่ 2.10 Overall Speedup

เพื่อประกอบความเข้าใจ ลองดูตัวอย่างที่ 2.5 ซึ่งเป็นการคำนวณค่า speed up ตามแนวทางของ Amdahl

ตัวอย่างที่ 2.5: การคำนวณ speed up ด้วยกฎของ Amdahl

ในการซื้ออาหารเย็นท่านนั้น จะต้องเสียเวลาไปกับการเดินทางไปและกลับ 10 นาที และ เสียเวลาไปกับการเลือกซื้ออาหาร และชำระเงินในร้านอาหารอีก 5 นาที หากผู้ซื้อสามารถเดินทางได้เร็วขึ้น 2 เท่า ค่า Speedup หรือ ประสิทธิภาพที่เพิ่มขึ้นโดยรวมจะเป็นเท่าใด

เวลาที่ใช้ในการซื้อของแต่เดิม คือ 15 นาที

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

จากตัวอย่างจะสังเกตุได้ว่า แม้จะเดินเร็วขึ้นถึง 2 เท่า แต่ประสิทธิภาพที่ได้โดยรวม เป็นเพียงแค่ 1.5 เท่า เท่านั้น (นี่คือเหตุที่กฎของ Amdahl เรียกว่า Law of Diminish Return กล่าวคือ การปรับปรุงให้บางอย่างเร็วขึ้น มีได้ให้ผลที่เร็วขึ้นเท่ากับแรงที่ออกเสมอไป) เพื่อประกอบความเข้าใจเพิ่มเติม ลองดูตัวอย่างที่ 2.6 ซึ่งเป็นการปรับปรุงประสิทธิภาพในทำนองเดียวกับตัวอย่างที่ 2.5 แต่เป็นการปรับปรุงเวลาในการซื้ออาหารและชำระเงินให้เร็วขึ้นเป็น 3 เท่าแทน

ตัวอย่างที่ 2.6: การประยุกต์ใช้กฎของ Amdahl เพื่อเป็นแนวทางในการปรับปรุงประสิทธิภาพ

ในการซื้ออาหารเย็นท่านนั้น ประกอบไปด้วยเวลาในการเดินทางไปกลับ และ เวลาในการเลือกซื้ออาหารและชำระเงิน หากเวลาในการเลือกซื้ออาหารและชำระเงินเป็น 1 ใน 3 ของเวลาทั้งหมด จงหาค่า Speedup หากผู้ซื้อสามารถเลือกซื้ออาหาร และชำระเงินได้เร็วขึ้น 3 เท่า

เมื่อเปรียบเทียบตัวอย่างที่ 2.5 และตัวอย่างที่ 2.6 จะพบว่า เวลาส่วนใหญ่เสียไปกลับการเดินมากกว่าการเลือกซื้ออาหารและชำระเงิน ดังนั้นหากสามารถเดินได้เร็วขึ้น 2 เท่า ย่อมให้ผลลัพธ์โดยรวมดีกว่าการเลือกซื้ออาหารและชำระเงินเร็วขึ้น 3 เท่า

หลักการเช่นนี้ ถูกนำมาใช้ในการปรับปรุงประสิทธิภาพของระบบคอมพิวเตอร์ เช่นกัน กล่าวคือ หากต้องเลือกปรับปรุงให้ส่วนใดส่วนหนึ่งของระบบเร็วขึ้น เราควรเลือกส่วนที่มีการใช้งานเป็นปริมาณมาก เพื่อให้ได้ประสิทธิภาพโดยรวมดีที่สุดแทนการเลือกส่วนที่ใช้งานน้อย

ประเด็นช่วยจำ ความจริงแล้วกฎของ Amdahl ก็มีที่มาจากการเทียบเวลา Execution Time ตามปกติ ดังนั้น หากสามารถหา Execution Time ได้ (สมการที่ 2.9) แล้วนำมาเทียบด้วยสมการ Speed Up (สมการที่ 2.2) ตามปกติ ก็จะได้ผล เมื่ออนกับการคำนวณด้วยสมการ Overall Speed (สมการที่ 2.10) เช่นกัน

2.7 แนวทางการเพิ่มประสิทธิภาพของระบบคอมพิวเตอร์

จากสมการที่ 2.2 จะเห็นว่า การปรับปรุงเวลาในการทำงานของหน่วยประมวลผลกลาง ทำได้โดยการลดค่า CPI หรือลดค่า Clock Cycle Time หรือลดจำนวนคำสั่งลง ดังนี้

- **Clock Cycle Time** หรือ **Clock Rate** ขึ้นอยู่กับ **โครงสร้างภายในยาร์ดแวร์** และ **เทคโนโลยี** ในการผลิต **transistor**¹⁰
- **CPI** ขึ้นอยู่กับ **โครงสร้างภายในยาร์ดแวร์** และ **โครงสร้างสถาปัตยกรรมชุดคำสั่ง**
- **จำนวนคำสั่ง** ขึ้นอยู่กับ **โครงสร้างสถาปัตยกรรมชุดคำสั่ง** ความสามารถในการแปลงภาษา และ การ Optimize ของตัวแปลงภาษา

2.8 สรุป

พื้นฐานในการวิเคราะห์และวัดประสิทธิภาพของเครื่องคอมพิวเตอร์นั้น เป็นสิ่งสำคัญในการศึกษาระบบสถาปัตยกรรมคอมพิวเตอร์ ทั้งนี้เนื่องจาก เมื่อมีการออกแบบหรือปรับปรุงระบบคอมพิวเตอร์ แล้ว จะต้องสามารถพิสูจน์ หรือ วิเคราะห์ได้ว่า การปรับปรุงดังกล่าวในนั้น ส่งผลต่อการพัฒนาระบบคอมพิวเตอร์ให้ดีขึ้นได้อย่างไร ทั้งนี้ในกรณีเป็นผู้ใช้หรือเลือกซื้อผลิตภัณฑ์ต่าง ๆ ทางคอมพิวเตอร์ ผู้ซื้อควรมีความรู้ความเข้าใจเมื่ออ่านคำแนะนำสำนักค้านั้น ๆ ทราบถึงข้อดีข้อเสีย วิธีทดสอบ โดยไม่หลงคล้อยตามกับคำโฆษณาชวนเชื่อ เพื่อให้สามารถเลือกใช้ระบบคอมพิวเตอร์ที่เหมาะสมกับงานของตน หน่วยประมวลผลกลางบางตัวนั้น อาจมีประสิทธิภาพดี เมื่อประมวลผลทางคณิตศาสตร์แบบจำนวนเต็ม แต่อาจมีประสิทธิภาพแย่เมื่อประมวลผลเลขทศนิยม หรือ Graphics

การวิเคราะห์ประสิทธิภาพในเชิงปฏิบัตินั้น นอกจากระดับความรู้พื้นฐานดังกล่าว ยังต้องอาศัย

¹⁰ ในปี 2014 เทคโนโลยีล่าสุดในการผลิตสารกึ่งตัวนำเกือบที่ขนาด 14 นาโนเมตร ในปี 2004 เทคโนโลยีการผลิตอยู่ที่ 90 นาโนเมตร ในทางปฏิบัติเทคโนโลยีที่เล็กลง ทำให้ transistor มีขนาดเล็กลง ซึ่งมักประหยัดพลังงานมากกว่า และทำงานได้เร็วขึ้นด้วยเช่นกัน

ความรู้ทางสถิติเพื่อประกอบการวิเคราะห์ด้วย เช่น การหาค่าเฉลี่ยทางคณิตศาสตร์ การหาค่าเฉลี่ย ยาโนนิก หรือการ Normalized ผลการทดสอบเพื่อทำการเปรียบเทียบ อย่างไรก็ตามค่าสถิติที่เป็นมาตรฐานสำหรับใช้ในการรายงานผลคือ ค่าเฉลี่ยเรขาคณิต

2.9 แบบฝึกหัดท้ายบท

1. เราจะวัดประสิทธิภาพของเครื่องคอมพิวเตอร์ได้อย่างไร
2. ท่านคิดว่า เหตุใดเราจึงเลือกใช้ CPU TIME ในการวัดประสิทธิภาพของระบบคอมพิวเตอร์
3. ปัจจัยใดบ้างส่งผลกระทบต่อประสิทธิภาพของเครื่องคอมพิวเตอร์
4. เราสามารถเพิ่มประสิทธิภาพของเครื่องคอมพิวเตอร์ได้อย่างไร
5. การเปลี่ยน CPU ใหม่ที่มีความเร็วสูงขึ้นใน จัดเป็นการ ปรับปรุง Response Time หรือ Throughput
6. ความเร็วของสายสัญญาณเครือข่าย 100 Mbps จัดเป็น Response Time หรือ Throughput จงให้เหตุผลประกอบการอธิบาย
7. แม้สัญญาณเครือข่ายจะมีความเร็วสูงถึง 1 Gbps อย่างไรก็ตามการขนส่งข้อมูลนัดใหญ่ที่มีประสิทธิภาพที่สุดคือการนำหน่วยเก็บข้อมูล เช่น Hard Drive จำนวนเยอะเยอะใส่รอนบรรทุก หรือบนส่งทางอากาศ กล่าวคือ การขนส่งดังกล่าว 1 ครั้ง อาจจะขนส่งข้อมูลได้มากกว่า ระบบเครือข่ายที่มีความเร็วที่สุดในปัจจุบันหลายล้านเท่า จัดว่าการขนส่งดังกล่าวให้ Throughput สูง ให้หรือไม่ เพาะเหตุใด
8. จงยกตัวอย่างคุณสมบัติที่ต้อง benchmark มา 4 ข้อ
9. จงเลือกชาร์ดแวร์ที่เหมาะสมสำหรับการทำ Java Application Server คำแนะนำ ดู [SPEC.org](http://www.spec.org/) (<http://www.spec.org/>)
10. คอมพิวเตอร์ 2 ชุด ทำการแปลงโปรแกรมเป็นภาษาเครื่อง 1 ผลเป็นชุดคำสั่งในกลุ่ม A, B และ C ซึ่งคำสั่งในแต่ละกลุ่มมีค่า CPI เป็น 1, 2, 3 ตามลำดับ โปรแกรมที่ได้มีลักษณะดังนี้
 - โปรแกรมที่ได้จากคอมพิวเตอร์ชุดที่ 1 มีคำสั่งอยู่ในกลุ่ม A 5 ล้านคำสั่ง กลุ่ม B 1 ล้านคำสั่ง และ กลุ่ม C 1 ล้านคำสั่ง
 - โปรแกรมที่ได้จากคอมพิวเตอร์ชุดที่ 2 มีคำสั่งอยู่ในกลุ่ม A 6 ล้านคำสั่ง กลุ่ม B 2 ล้านคำสั่ง และ กลุ่ม C 1 ล้านคำสั่ง

โปรแกรมแต่ละชุดใช้เวลาเท่าใด โปรแกรมชุดใดมีประสิทธิภาพมากกว่ากัน หากเปรียบเทียบในหน่วย MIPS โปรแกรมชุดใดจะให้ค่า MIPS สูงกว่า เพราะเหตุใด

11. ในระบบคอมพิวเตอร์หนึ่ง มีการประมาณผลที่เกี่ยวข้องกับการคูณคิดเป็น 15% ของการประมาณผลทั้งหมด ทีมวิศวกรของระบบคอมพิวเตอร์นี้ มีความเห็นเกี่ยวกับการปรับปรุงประสิทธิภาพของระบบคอมพิวเตอร์ดังกล่าว แต่อกอคือเป็น 2 กลุ่ม กลุ่มแรกเห็นว่า ควรจะปรับปรุงวงจรคูณให้มีความเร็วสูงขึ้นสีเท่า กลุ่มที่สองเห็นว่าไม่ควรปรับปรุงวงจรได้แต่ควรไปปรับปรุงคอมไฟเลอร์ให้ลดการใช้การคูณลง หากท่านเป็นหัวหน้าวิศวกรคอมพิวเตอร์ ของระบบดังกล่าว ท่านจะเลือกลงทุนให้วิศวกรกลุ่มไหนดำเนินงาน เพราะเหตุใด จงอธิบาย
12. จากสมการที่ 2.2 และ Feature ต่างๆ ของระบบคอมพิวเตอร์ (Program, Compiler, ISA, Organization, และ Technology) ท่านคิดว่า Feature เหล่านี้ มีผลกระทบต่อประสิทธิภาพของ CPU ในส่วนใด (IC, CPI, Tc) จงอธิบาย
13. เหตุใดการประเมินประสิทธิภาพจึงนิยมใช้ Geometric mean ในการประเมิน จงให้เหตุผลประกอบคำอธิบาย
14. หากชุดโปรแกรมทดสอบประกอบด้วยคำสั่งเกี่ยวกับหน่วยความจำซึ่งมีค่า CPI เป็น 5 อยู่ 40% และคำสั่งที่เกี่ยวกับการ Branch ซึ่งมีค่า CPI เป็น 3 อยู่ 20% ที่เหลือเป็นคำสั่งเกี่ยวกับการประมาณผลทางคณิตศาสตร์ ซึ่งมีค่า CPI เป็น 4 ค่า average CPI ของโปรแกรมทดสอบดังกล่าวจะเป็นเท่าใด

3 สถาปัตยกรรมชุดคำสั่ง

สถาปัตยกรรมชุดคำสั่ง คือสถาปัตยกรรมภาษาที่ใช้สำหรับระบบคอมพิวเตอร์ หรือภาษาเครื่อง ทั้งนี้ ภาษาเครื่องนั้นเป็นภาษาคอมพิวเตอร์ที่มีลักษณะแตกต่างจากภาษาคอมพิวเตอร์ขั้นสูงโดยทั่วไป โดย มักไม่มีคำสั่งการควบคุมที่ซับซ้อน (เช่น while, for, loop) ในหนังสือเล่มนี้มุ่งประเด็นสนใจที่ สถาปัตยกรรมชุดคำสั่งของ LADA (หรือ nanoLADA) ซึ่งเป็นสถาปัตยกรรมชุดคำสั่งสมมุติที่ถูกออกแบบขึ้นมาเพื่อประการเรียนการสอนในหนังสือเล่นนี้โดยเฉพาะ ทั้งนี้เนื่องจาก nanoLADA เป็น สถาปัตยกรรมที่มีชุดคำสั่งน้อยอย่างง่าย ดังนั้น อาจจะมีการยกตัวอย่าง สถาปัตยกรรมอื่นขึ้นมาอ้างอิง เพื่อเปรียบเทียบการทำงาน

ในสถาปัตยกรรมชุดคำสั่งของระบบคอมพิวเตอร์ สิ่งที่ต้องศึกษาและต้องกล่าวถึง ประกอบด้วย

- คำสั่งต่างๆ พร้อมคำอธิบายเรื่อง จำนวน cycle ที่ใช้ และความหมายในการทำงาน ระดับการแลกเปลี่ยนข้อมูลในระดับ rejister ซึ่งนิยมอธิบายด้วยภาษา Register Transfer Language (RTL)
- rejister (Register) หรือที่พากข้อมูลทั่ว千方百ยวในระบบ และ Flag แสดงสถานะ
- ระบบการจัดการหน่วยความจำ การเรียบเรียงและอ้างอิงข้อมูล รวมถึงสถาปัตยกรรมหน่วยความจำ
- รูปแบบคำสั่งภาษาเครื่อง (Instruction Format)
- ระบบ interrupt และ exception

ทั้งนี้ระบบสถาปัตยกรรมชุดคำสั่งนั้น มีความจำเป็นอย่างยิ่งในการพัฒนาอาร์ดแวร์ของระบบ คอมพิวเตอร์ และ ซอฟต์แวร์ระบบ เนื่องจากเป็นข้อตกลงที่ใช้ระหว่างผู้พัฒนาอาร์ดแวร์และ ซอฟต์แวร์ (ดังกล่าวแล้วในบทที่ 1) อย่างไรก็ตามสถาปัตยกรรมชุดคำสั่งมีได้ครอบคลุมถึงการเขียน ภาษาแอสเซมบลีทั้งหมด เพราะในภาษาแอสเซมบลีจะมีการกำหนด Macro หรืออ้างอิงตำแหน่งใน หน่วยความจำด้วย Label แทนที่การอ้างอิงด้วย Address ของหน่วยความจำ เป็นต้น

ก่อนจะเริ่มศึกษาเรื่องสถาปัตยกรรมชุดคำสั่ง มีคำศัพท์ที่ควรทราบในเบื้องต้นดังนี้

- Instructions หรือ Operators (Operations) ในที่นี้หมายถึงคำสั่งที่ต้องการให้หน่วยประมวลผลทำงาน เช่น การสั่งให้บวกเลข จะต้องสั่งด้วยคำสั่ง add เป็นต้น
- Operands¹¹ หรือข้อมูลสำหรับการประมวลผล เช่น การสั่งให้หน่วยประมวลผลกลางทำ

¹¹ ที่มาของคำว่า Operands ไม่มีการบันทึกไว้ชัดเจน แต่เดয์มีอาจารย์ท่านหนึ่ง (อาจารย์ท่านนี้คือ ผศ. กอบกุล เตชะวนิช อตีดอาจารย์ ภาควิชาวิศวกรรมคอมพิวเตอร์ จุฬาลงกรณ์มหาวิทยาลัย) กล่าวไว้ว่า น่าจะมาจากคำว่า Operation และ Land มาผสมกัน หมายถึงที่ที่เกิดการประมวลผล

$5+7$ จะได้ว่า Instruction หรือ Operator คือการบวก (add) และมี Operands เป็น 5 และ 7 เป็นต้น

3. **Memory Address** หรือ ตำแหน่งหน่วยความจำ หมายถึงที่อยู่ของข้อมูลในหน่วยความจำ เพื่อความสะดวกขอให้มองหน่วยความจำแต่ละ byte เป็นตู้เล็กๆ หลายตู้ที่วางต่อกัน โดยแต่ละตู้มีการกำหนดหมายเลขเพื่อความสะดวกในการอ้างอิง
4. **Register** (อ่านว่า เรจิสเตอร์) หมายถึง ที่พักข้อมูลชั่วคราวที่ภายในหน่วยประมวลผลกลาง ในแต่ละหน่วยประมวลผลกลางจะมีการตั้งชื่อสำหรับเรียก register ที่แตกต่างกันไป เพื่อกันความสับสน การอ้างอิงถึงค่า register ในสถาปัตยกรรมชุดคำสั่งนี้ จะเขียนตัวย "\$" เช่น $\$r1$ $\$r2$ เป็นต้น
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 เพราะเป็นค่าที่ถูกระบุเข้าไปเลยในคำสั่ง (คำสั่งสามารถนำค่า 5 และ 7 ไปใช้งานได้เลย โดยไม่ต้องมีการประมวลผลเพิ่มเติม)

เพื่อกันความสับสน การอ้างอิงถึงค่า immediate ในสถาปัตยกรรมชุดคำสั่งนี้ จะเขียนตัวด้วย "#" เช่น

คำสัพท์เหล่านี้ นอกจากจะถูกใช้อ้างอิงในหนังสือเล่มนี้แล้ว ยังเป็นมาตรฐานสำหรับอ้างอิงในสถาปัตยกรรมคอมพิวเตอร์ทั่วไปอีกด้วย ดังนั้นในหนังสือนี้จึงขออ้างอิงถึงคำสัพท์เหล่านี้โดยทับศัพท์โดยตรง เพื่อให้ผู้อ่านคุ้นเคยกับคำต่างๆ

เพื่อให้เข้าใจภาพรวมและเห็นความสำคัญของสถาปัตยกรรมชุดคำสั่ง ในบทนี้จะเริ่มต้นด้วยการกล่าวถึงสถาปัตยกรรมชุดคำสั่งในอดีต การจำแนกหมวดคำสั่งการจัดการข้อมูลในหน่วยความจำ และรูปแบบแบบชุดคำสั่งตามลำดับ

3.1 สถาปัตยกรรมชุดคำสั่งในอดีต

ก่อนที่จะมีการกำหนดสถาปัตยกรรมชุดคำสั่งขึ้นมานั้น ในอดีต หน่วยประมวลผลกลางแต่ละรุ่น จะไม่มีความสอดคล้องกันแต่อย่างใด หมายความว่า หน่วยประมวลผลกลางรุ่นใหม่จะไม่มีความเข้ากันได้ (compatible) กับหน่วยประมวลผลกลางรุ่นก่อนหน้า แม้ว่าจะเป็นหน่วยประมวลผลกลางจากผู้ผลิตรายเดียวกันก็ตาม ผลคือทุกครั้งที่มีการเปลี่ยนรุ่นของหน่วยประมวลผลกลาง จะต้องทำการพัฒนาซอฟต์แวร์ใหม่ทั้งหมด

ดังนั้นหากหน่วยประมวลผลกลางชุดใดมีสถาปัตยกรรมชุดคำสั่งเหมือนกันหรือใกล้เคียงกัน ย่อม

หมายความว่า ซอฟต์แวร์ที่พัฒนาบนระบบหนึ่ง สามารถที่จะนำมาใช้งานได้ในอิทธิพลหนึ่งทันที ตัวอย่างเช่น การที่ซอฟต์แวร์ที่พัฒนาสำหรับหน่วยประมวลผลกลาง 386 ของ intel สามารถทำงานได้ทันทีบนหน่วยประมวลผลกลางรุ่นใหม่ล่าสุด (เช่น intel core i7) โดยไม่ต้องทำการเปลี่ยนแปลงแก้ไขใดๆ ทั้งนี้เพื่อรับหน่วยประมวลผลกลางทั้งสองแบบ ยังคงมีสถาปัตยกรรมชุดคำสั่งที่ใกล้เคียงกัน (เหมือนเดิม) สำหรับคำสั่งทั่วไป เป็นต้น

การอ้างอิง Operand ในคำสั่งเป็นเกณฑ์ การจำแนกสถาปัตยกรรมชุดคำสั่งเป็นกลุ่ม ตามลักษณะการอ้างอิงข้อมูลในการประมวลผล สามารถแบ่งออกได้เป็น สถาปัตยกรรมแบบ **Accumulator** (อ่านว่า แอคคูมูลเตอร์) สถาปัตยกรรมแบบ **Stack** สถาปัตยกรรมแบบ **Memory-Memory** สถาปัตยกรรมแบบ **Register-Memory** และ สถาปัตยกรรมแบบ **Register-Register** (บางตำราจะเรียกว่า สถาปัตยกรรมแบบ Load-Store)

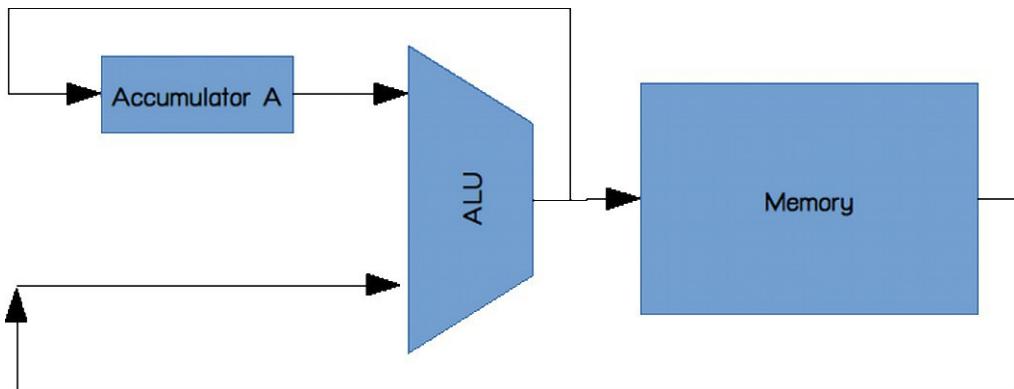
เพื่อประกอบความเข้าใจในกรณีที่อ่านตำราอื่นประกอบในหลายตำรา จะนิยมอ้างอิงถึง สถาปัตยกรรมแบบ Memory-memory และสถาปัตยกรรมแบบ Register-Memory ว่า **Complex Instruction Set Computing (CISC)** ซึ่งเป็นสถาปัตยกรรมที่เน้นให้หน่วยประมวลผลกลางประมวลผลงานที่ซับซ้อนภายในหนึ่งคำสั่ง (คำสั่งมีความซับซ้อน compiler ทำการแปลภาษาได้ด้วย) ในทำนองเดียวกัน หลายตำรา尼ยมเรียกสถาปัตยกรรมแบบ Register-Register ว่า **Reduce Instruction Set Computing (RISC)** ซึ่งเป็นสถาปัตยกรรมที่เน้นให้แต่ละคำสั่งมีขนาดสั้นทำงานกระชับ และการทำงานที่ซับซ้อนจะเขียนอยู่กับ compiler ที่จะต้องทำการแปลคำสั่งที่ซับซ้อนให้เป็นคำสั่งที่สั้น/ง่ายหลายคำสั่ง ทำงานต่อ กัน

อย่างไรก็ตาม ปัจจุบันอยู่ในยุค **Post RISC** ก้าวคือ หน่วยประมวลผลกลางบางแบบ (เช่น intel core i3, i5 i7) แม้จะมีสถาปัตยกรรมชุดคำสั่งแบบ CISC (มีคำสั่งที่มีการใช้งานแบบ Register-Memory) แต่โครงสร้างภายจะมีการแปลงคำสั่งเหล่านี้เป็น micro ops ก่อน (ซึ่ง micro ops เป็นสถาปัตยกรรมแบบ Register-Register) จากนั้นหน่วยประมวลผลกลางจะทำงานด้วยคำสั่ง micro ops แทน จึงไม่อาจกล่าวได้ว่า หน่วยประมวลผลใดเป็น CISC หรือ RISC ได้ชัดเจนนักในปัจจุบัน

ในเบื้องต้น ขอให้ผู้เรียนทำความเข้าใจสถาปัตยกรรมแบบต่างๆ เพื่อเป็นแนวทางในการศึกษาและทำความเข้าใจประสิทธิภาพของหน่วยประมวลผลกลางต่อไป ดังนี้

3.1.1 สถาปัตยกรรมแบบ **Accumulator**

สถาปัตยกรรมแบบ **Accumulator** (Accumulator Architecture) นั้นเป็นสถาปัตยกรรมในยุคแรกที่ **ทรานซิสเตอร์ยังมีราคาแพง** ทำให้การสร้างหน่วยประมวลผลจำเป็นต้องออกแบบให้มีหน่วยความจำภายในเฉพาะเท่าที่จำเป็น **Accumulator** นั้นความจริงก็เป็นเพียงเรจิสเตอร์ชุดหนึ่งที่อยู่ภายในระบบเท่านั้น สถาปัตยกรรมชุดคำสั่งในลักษณะที่เรียกว่า **Accumulator** (Accumulator Machine) คือ สถาปัตยกรรมที่มีเรจิสเตอร์สำหรับใช้ในการประมวลคำสั่งในหน่วยประมวลผลกลางเพียงชุดเดียว ซึ่งการคำนวณได้โดยจะต้องใช้ **Accumulator** ร่วมด้วย และผลลัพธ์ที่ได้จะเก็บใน **Accumulator** เสมอ รูปที่ 3.1 แสดงทางเดินข้อมูลของสถาปัตยกรรมแบบ **Accumulator Machine**



รูปที่ 3.1: ทางเดินข้อมูลของสถาปัตยกรรมแบบ Accumulator

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

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

จะเห็นว่าการคำนวณพื้นฐานดังกล่าว ต้องใช้ถึง 10 คำสั่ง ส่วนหนึ่งเนื่องจากนิพักข้อมูลให้ใช้เพียงหนึ่งที่ คือ **Accumulator** ทำให้ทุกครั้งที่จำเป็นต้องมีการคำนวณด้วย **Accumulator** แต่มีค่าอื่นที่จะต้องใช้ค้างอยู่ใน **Accumulator** จะต้องมีการย้ายข้อมูลออกมายังหน่วยความจำก่อนเสมอ (คำสั่งที่ 3 และ 6) กล่าวคือ **Accumulator** เป็นคอขวดของการประมวลผลนั่นเอง

3.1.2 สถาปัตยกรรมแบบ Stack

สถาปัตยกรรมแบบ Stack หรือ Stack Machine นั้น อาศัยหลักการที่ว่า **operands** จะเป็นด้านบนของ **stack** เสมอ ด้วยสมมุติฐานนี้ ช่วยให้แต่ละคำสั่งที่เป็นการประมวลผลมีขนาดเล็ก (เพราะไม่จำเป็นต้องระบุ **operands** อย่างไรก็ตามปัญหาที่เห็นได้ชัดคือ **stack** จะกลายเป็นค่าข้าดของการประมวลผลในทำนองเดียวกับของ **Accumulator Machine** ซึ่งทำให้ไม่สะดวกต่อการทำการประมวลผลแบบบานาน) รูปที่ 3.2 แสดงทางเดินข้อมูลของสถาปัตยกรรมแบบ Stack Machine

รูปที่ 3.2: ทางเดินข้อมูลของสถาปัตยกรรมแบบ Stack

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

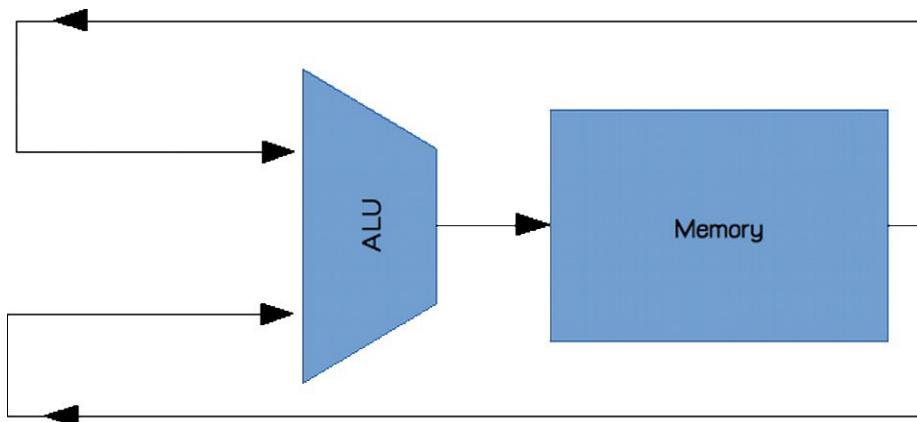
1. PUSH a ; (top of stack) a
2. PUSH b ; (top of stack) b , a
3. MUL ; (top of stack) (a*b)
4. PUSH a ; (top of stack) a, (a*b)
5. PUSH b ; (top of stack) b, a, (a*b)
6. PUSH c ; (top of stack) c, b, a, (a*b)
7. MUL ; (top of stack) (b * c), a, (a*b)
8. SUB ; (top of stack) (a - (b*c)) , (a*b)
9. ADD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

10. POP x ;

เมื่อเปรียบเทียบกับสถาปัตยกรรมแบบ Accumulator จะเห็นว่าจำนวนคำสั่งที่ใช้มีปริมาณไม่แตกต่างกัน อย่างไรก็ตาม คำสั่งของ Stack Machine จะสั้นกว่ามาก (เรียกว่ามี code density ดี เพราะเขียนคำสั่งสั้นแต่ได้ความหมายสมบูรณ์) นอกจากนี้ยังสะดวกต่อการออกแบบชาร์ดแวร์ และสะดวกต่อการทำงานของ compiler ด้วย (เพราะ compiler ต้องสร้าง stack tree ก่อนทำการแปลง code ก่อนอยู่แล้ว)

3.1.3 สถาปัตยกรรมแบบ Memory-Memory

สถาปัตยกรรมแบบ Memory-Memory หรือ Memory-memory Machine เป็นสถาปัตยกรรมที่พบได้ในเครื่อง Mainframe บางระบบ ซึ่งสถาปัตยกรรมในลักษณะนี้สามารถประมวลผลได้โดยไม่ต้องมี rejister หรือ Accumulator ภายในระบบ กล่าวคือ การประมวลผลแต่ละคำสั่งจะทำการอ่านและเขียนค่าจากหน่วยความจำโดยตรง หากจะจำแนกย่อยต่อ บางตัวรายังแบ่ง Memory-memory ออกเป็นแบบ 2 operands และแบบ 3 operands ซึ่งแตกต่างกันที่ความยาวของคำสั่งและตำแหน่งที่ใช้เก็บผลลัพธ์ โดยแบบ 2 operands จะใช้ตัวตั้งตัวหนึ่งในการเก็บผลลัพธ์ และแบบ 3 operands จะต้องระบุทั้งตัวตั้ง ตัวประมวลผล และตัวเก็บผลลัพธ์ 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 i, b, c ; i=b*c
2. SUB j, a, i ; j=a-i = a - (b*c)
3. MUL k, a, b ; k=a*b

4. ADD x, k, j ; $x = k+j = (a*b) + (a - (b*c))$

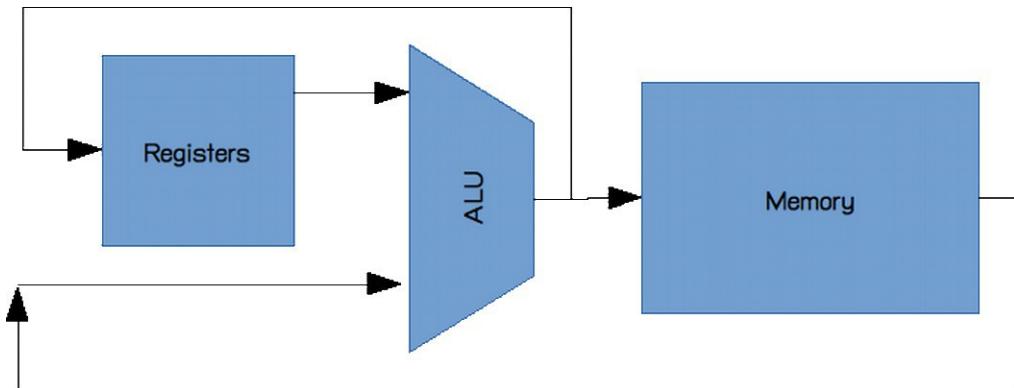
หรือในกรณีสถาปัตยกรรมแบบ 2 operands จะได้ลำดับการทำงานเป็น

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

จะสังเกตุเห็นว่า ไม่ว่ากรณี 3 operands หรือ 2 operands ต่างก็ให้ผลลัพธ์เป็นโปรแกรมที่สั้นกว่าแบบ Accumulator Machine และ แบบ Stack Machines อย่างไรก็ตามข้อเสียของสถาปัตยกรรมในลักษณะนี้คือ ทำให้คำสั่งแต่ละคำสั่งมีการทำงานที่ซับซ้อน เพราการทำงานทุกครั้งต้องมีอ่านเขียนข้อมูลจากหน่วยความจำ ซึ่งทำงานซ้ำกับค่าเรจิสเตอร์มา (เปรียบเทียบเรจิสเตอร์เหมือนค่าตัวเลขที่จำอยู่ในสมองระหว่างการคำนวณกับ หน่วยความจำคือตัวเลขที่เขียนอยู่ในกระดาษทดลอง ซึ่งการอ่าน/เขียนข้อมูลจากกระดาษทดลองย่อมซ้ำกับการคำนวณ)

3.1.4 สถาปัตยกรรมแบบ Register-Memory

สถาปัตยกรรมแบบ Register-Memory หรือ Register-Memory Architecture เป็นสถาปัตยกรรมที่มี Operands ตัวหนึ่งอยู่ในหน่วยความจำ และอีกตัวหนึ่งเป็นเรจิสเตอร์ สถาปัตยกรรมลักษณะนี้พบในสถาปัตยกรรมยุค 1950, 1960, 1970 ซึ่งเป็นยุคที่นิยมให้หน่วยประมวลผลกลางทำงานต่างๆ แทน compiler ซึ่ง intel 386 ที่นิยมใช้กันในปัจจุบันก็จัดอยู่ในกลุ่มนี้ รูปที่ 3.4 แสดงทางเดินข้อมูลของสถาปัตยกรรมในลักษณะนี้ ในเบื้องต้นโครงสร้างจะดูคล้ายกับแบบ Accumulator แตกต่างตรงนี้ สถาปัตยกรรมแบบนี้มีเรจิสเตอร์ให้ใช้มากกว่า 1 ชุด



รูปที่ 3.4: ทางเดินข้อมูลของสถาปัตยกรรมแบบ Register-Memory

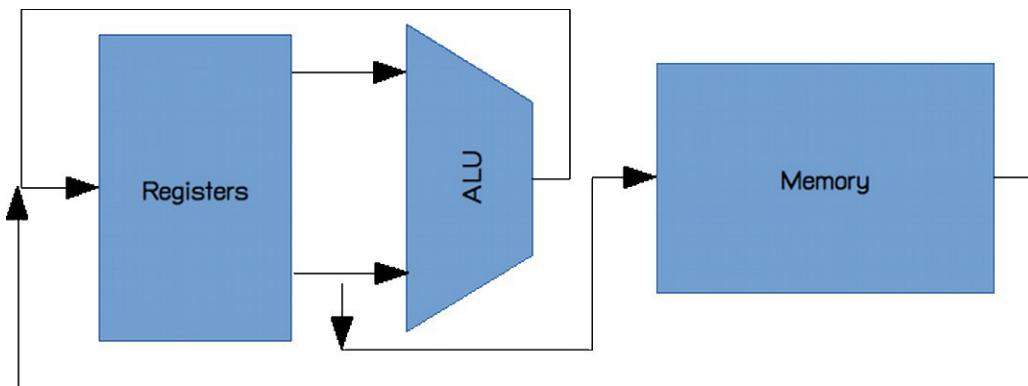
หากต้องการประมวลผล $x = (a * b) + (a - (b * c))$ เมื่อแปลงประโยคดังกล่าวเป็นภาษาเครื่อง จะได้ลำดับการทำงานดังนี้

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

หากวิเคราะห์ในเบื้องต้นจะเห็นการทำงานคล้ายคลึงกับแบบ Memory-Memory ผสมกับแบบ Accumulator หากแต่มีเรจิสเตอร์ภายในให้ใช้มากกว่า **ข้อเสียอย่างหนึ่งที่เห็นได้ชัดคือ operands แต่ละตัวจะใช้เวลาในการเข้าถึงไม่เท่ากัน ตัวหนึ่งจะอยู่ในเรจิสเตอร์ เข้าถึงได้เร็ว ในขณะที่การอ่านค่า operands ในหน่วยความจำจะช้ากว่าเสมอ นอกจากนี้การที่ operand มีการอ้างอิงไม่เหมือนกัน ยังมีศักดิ์เรียกอีกว่า poor orthogonal กล่าวคือ ความหมายและการอ้างอิงของ operand ไม่สอดคล้องกัน มีความเร็วในการเข้าถึงไม่เท่ากัน**

3.1.5 สถาปัตยกรรมแบบ Register-Register

สถาปัตยกรรมแบบ Register-Register หรือ Load-Store Architecture เป็นสถาปัตยกรรมที่แบบได้ในหน่วยประมวลผลกลางรุ่นใหม่ สถาปัตยกรรมแบบ ARM ซึ่งใช้ในโทรศัพท์แบบ smart phone ที่พับได้ทั่วไปในปัจจุบันก็เป็นสถาปัตยกรรมในลักษณะนี้ หลักการพื้นฐานของสถาปัตยกรรมในลักษณะนี้คือ การประมวลผลที่เป็นการคำนวณ จะต้องใช้ operands ที่เป็น rejister เท่านั้น กล่าวคือ จะต้องทำการ load ค่าที่ต้องการใช้ในการคำนวณเข้ามาจำไว้ก่อน จากนั้นเมื่อประมวลผลเสร็จจะ store ค่าที่ได้คืนลงไปในหน่วยความจำ(การทำงานลักษณะนี้จึงเป็นที่มีของชื่อ Load-Store Architecture)



รูปที่ 3.5: โครงสร้างทางเดินข้อมูลของสถาปัตยกรรมแบบ Register-Register

หากต้องการประมวลผล $x = (a * b) + (a - (b * c))$ เมื่อแปลงประโยชน์คดังกล่าวเป็นภาษาเครื่อง 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 เมื่อ ค่าที่ต้องการอยู่ใน寄存器แล้ว การประมวลผลจะทำให้อย่างรวดเร็ว นอกจากนี้ทุกคำสั่งยังมีความยาวเท่ากันทำให้การออกแบบรูปแบบคำสั่งทำได้ง่าย แต่ประสิทธิภาพที่ได้นั้นขึ้นกับการทำงานของคอมพิวเตอร์ และ ในหลายกรณีจะใช้จำนวนคำสั่งมากกว่าเมื่อเปรียบเทียบกับแบบ Memory-memory

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

3.2 หมวดคำสั่งและชุดคำสั่ง

ชุดคำสั่งในหน่วยประมวลผลต่างๆ โดยทั่วไปสามารถจำแนกตามลักษณะการทำงาน หรือการเรียกใช้งาน ได้เป็นหมวดหมู่ดังนี้

1. คำสั่งที่เกี่ยวกับการเคลื่อนย้ายข้อมูล (Data Movement) ใช้สำหรับการอ่านและเขียนข้อมูลจากหน่วยความจำ
2. หมวดคำสั่งการประมวลผลทางคณิตศาสตร์ (บวก ลบ คูณ หาร)
3. หมวดคำสั่งการทำงานทางตรรกะ
4. หมวดคำสั่งควบคุม เพื่อกำหนดเงื่อนไขของการทำงานคำสั่งถัดไป เช่น IF .. THEN .. ELSE รวมถึงการเรียก sub routine ย่อย

นอกจากนี้ สถาปัตยกรรมบางแบบอาจมีหมวดคำสั่งอื่น หรือชุดคำสั่งที่ทำหน้าที่พิเศษอื่น ทั้งนี้จะแตกต่างกันไปตามโครงสร้างและการออกแบบของผู้พัฒนา ด้วยอย่างชุดคำสั่งพิเศษเช่น ชุดคำสั่งที่มีการทำงานในลักษณะของเวคเตอร์ กล่าวคือ 1 คำสั่ง มีการกระทำกับข้อมูลหลายชุด (มีการทำงานแบบขนาน) ซึ่งเหมาะสมกับการประมวลผลด้าน graphics และ multimedia เป็นต้น

ในการประมวลผลแต่ละคำสั่งนั้น หน่วยประมวลผลกลางจะมีขั้นตอนในการดึงคำสั่งจากหน่วยความจำเข้าสู่หน่วยประมวลผลกลาง จากนั้นจึงทำการประมวลผลต่อไป (รายละเอียดเพิ่มเติมดูในบทที่ 5 และ 6) ทั้งนี้คำสั่งที่จะถูกดึงเข้ามานั้นจะถูกกำหนดโดย register พิเศษซึ่งเรียกว่า Program Counter (PC) หรือ สถาปัตยกรรมบางแบบอาจจะเรียก register พิเศษนี้ว่า Instruction Pointer¹² (IP) ดังนั้nlักษณะพิเศษของคำสั่งควบคุมคือ สามารถปรับเปลี่ยน PC ให้ไปยังคำสั่งอื่น ซึ่งมีได้เป็นคำสั่งที่ติดกันได้ เช่น

BEQ \$r0,\$r1,test

¹² 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 (ซึ่งจะกล่าวถึงต่อไป)

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

คำสั่ง BEQ เป็นคำสั่งควบคุมซึ่งจะทำการเปรียบเทียบค่าภายใน register \$r0 และ \$r1 ถ้าเท่ากัน จะไปทำงานที่คำสั่ง SUB และ หากไม่เท่ากันจะทำงานที่คำสั่ง ADD ซึ่งเป็นคำสั่งถัดไปที่ติดกัน

การเขียนโปรแกรมด้วยคำสั่งเหล่านี้ในภาษาแอสเซมบลีเพื่อส่งให้เครื่องคอมพิวเตอร์ทำงานนั้นแต่ละคำสั่งจะประกอบด้วยรหัสคำสั่ง (Opcode) และตัวปฏิบัติการ (Operands) โดย Opcode จะบอกถึงคำสั่งที่ต้องการทำงาน และ ตัวปฏิบัติการบอกถึงหน่วยความจำหรือ Register ที่ใช้ในการทำงานคำสั่งนั้น (operands) เช่น

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

```
AND    $r3, $r0, $r1
```

ADD และ AND นั้นเป็น Opcode ที่บอกให้ทราบว่าคำสั่งที่ต้องการทำงานคือ add และการ AND โดย \$r0, \$r1, \$r2, \$r3 เป็น Operand ที่บอกให้ทราบว่า ให้นำค่าใน register \$r1 และ \$r2 มาบวกกันแล้วเก็บไว้ที่ \$r0 เป็นต้น

สถาปัตยกรรมชุดคำสั่งแบบ LADA ที่ใช้อ้างอิงในหนังสือเล่มนี้ เป็นสถาปัตยกรรมแบบ Register-Register ดังนั้น การประมวลผลทางคณิตศาสตร์จะมี Operand เป็น register เท่านั้น โดย Operand ตัวแรกจะใช้บอก register ที่ใช้เก็บผลลัพธ์ และ Operand ถัดมาอีก 2 ตัวจะใช้บอก Operands ที่เป็นตัวตั้ง (ในที่นี้ขอเรียกว่า source) และ ตัวประมวลผล (ในที่นี้ขอเรียกว่า target) เช่น

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

มีความหมายคือ

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

ทั้งนี้แนวทางหนึ่งอธิบายความหมายของคำสั่งคือ การใช้ (Logical) Register-Transfer Language ก่าว่าคือ การอธิบายความ คำสั่งทำงานอย่างไร โดยระบุว่ามีแลกเปลี่ยนข้อมูลระหว่าง register ได้ในระบบบ้าง เช่น เมื่อต้องการอธิบายการทำงานของคำสั่ง ADD ในคู่มือของสถาปัตยกรรมชุดคำสั่งอาจจะมีการอธิบายดังนี้

คำสั่ง: ADD rd, rs, rt

RTL:

$$\{op, rs, rt, rd\} \leftarrow MEM[PC]$$

```
R[rd] ← R[rs] + R[rt]
```

```
PC ← PC + 4
```

จากตัวอย่าง RTL อธิบายการทำงานของคำสั่ง ADD คือ (1) ค่า {op, rs, rt, rd} ซึ่งเป็นค่าของคำสั่งจะถูกอ่านจาก หน่วยความจำ(Memory) ตำแหน่งที่ซึ่ด้วย PC (ขั้นตอนนี้มักถูกอ้างอิงว่า Fetch เพราะเป็นการอ่านค่าคำสั่ง) จากนั้นในการประมวลผล (2) จะเป็นการนำค่า register ที่ซึ่ด้วย rs และ rt มาทำการบวกกัน แล้วนำผลลัพธ์ที่ได้ ไปเก็บใน register ที่ซึ่ด้วย rd และ (3) พร้อมกันนั้นจะมีการเพิ่มค่า PC ขึ้นอีก 4 (ในกรณีนี้ สมมุติว่าแต่ละคำสั่งมีความยาว 32 บิต หรือ 4 ไบต์)

ดังนั้นกรณีที่พัฒนาซอฟต์แวร์ด้วยภาษาชั้นสูงแล้วมีการประกาศหรือใช้ตัวแปรสำหรับการอ้างอิงค่า Compiler หรือตัวแปลภาษาจะต้องทำการจับคู่ (Associate) ตัวแปล กับ register ในขณะที่มีการประมวลผล (ดู)

ตัวอย่างที่ 3.1: ตัวอย่างการแปลภาษา C เป็น LADA code

ภาษา C:

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

หาก compiler กำหนดให้ A, B, C, E, 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
```

เนื่องจาก LADA มี register ให้ใช้เพียง 32 ตัวเท่านั้น ดังนั้นหากในซอฟต์แวร์ที่พัฒนาจำเป็นต้องมีการอ้างอิงตัวแปลมากกว่า 32 ตัว ข้อมูลเหล่านั้นจะต้องถูกจัดเก็บไว้ในหน่วยความจำ และมีการอ่านค่าขึ้นมาเพื่อประมวลผล เมื่อเสร็จสิ้นแล้วก็บันทึกกลับไปยังหน่วยความจำเข่นเดิม และหากหน่วย

ความจำยังไม่พอเพียงที่จะเก็บข้อมูลดังกล่าว อาจจะต้องบันทึกข้อมูลเหล่านั้นลงใน I/O ต่างๆ เช่น Hard Disk เป็นต้น

3.3 การจัดการหน่วยความจำ (*Memory Organization*)

หน่วยความจำนั้นเป็นเหมือนกับ Array ที่มี 1 มิติขนาดใหญ่ ซึ่งจะอ้างอิงด้วย Address สำหรับ LADA จะใช้ Address ในการอ้างอิงหน่วยความจำ 32 bit ซึ่งหมายความว่า LADA จะมีหน่วยความจำได้ทั้งสิ้น 2^{32} Byte หรือ 4 Gigabyte

ในการอ้างอิงหน่วยความจำนั้นอาจจะเป็นการอ้างถึงครั้งละ 1 Byte (Byte Addressing) หรือ อ้างอิงครั้งละ 2 Byte (Half Word Addressing) หรืออาจจะอ้างอิงครั้งละ 4 Byte (Word Addressing) ก็ได้ ทั้งนี้ขึ้นอยู่กับรูปแบบคำสั่งแต่ละคำสั่ง

นอกจากขนาดของ address และ register ที่เกี่ยวข้องกับการอ้างอิงข้อมูลในหน่วยความจำแล้ว ในการออกแบบสถาปัตยกรรมชุดคำสั่งสำหรับอ้างอิงหน่วยข้อมูลในหน่วยความจำ มีข้อที่ต้องคำนึงถึงอยู่ 3 ประการคือ การเรียงข้อมูล (Endian) และ การกำหนดตำแหน่งเริ่มต้นของข้อมูล (memory alignment)

3.3.1 การเรียงข้อมูล (Endian)

ในการอ้างอิงแบบ Word และ Half-Word สิ่งที่เป็นประเด็นในการวิเคราะห์ประการหนึ่ง คือ ระบบ big endian และ little endian ซึ่งแตกต่างกันไปตามโพรცессอร์สร้างของหน่วยประมวลผลกลางแต่ละตัว ดังรูปที่ 3.6 แสดงการจัดเก็บข้อมูล 0A0B0C0Dh (ฐาน 16) หรือ 168496141 (ฐาน 10) 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

จากรูปข้อแตกต่างที่เห็นได้ชัดคือ ในระบบ big endian นั้น ส่วนที่มีนัยสำคัญต่อทางขวาเมื่อของการเขียนตัวเลขแบบปกติ มักจะอยู่ที่ address สูงในขณะที่ระบบ little endian นั้น ส่วนที่มีนัยสำคัญต่อ มักจะอยู่ที่ address ต่อไปนั้น

ข้อดีข้อเสียของระบบ big endian นั้น ในการจุบันไม่เห็นข้อแตกต่างมากนัก เนื่องจากระบบสาย

สัญญาณสำหรับอ่านข้อมูลมีขนาดใหญ่พอกว่าที่จะอ่านข้อมูลเข้ามาได้ทั้งหมด อย่างไรก็ตาม หากสมมุติว่าการอ่านข้อมูลหนึ่งครั้ง ไม่สามารถอ่านได้ครบทั้ง word (เช่น ต้องการอ่านข้อมูล 32 บิตและอ่านจากสายสัญญาณ 16 บิต) การอ่านข้อมูลจาก address ต่อไปสูงในแบบ big endian จะทำให้เราทราบเครื่องหมาย (ซึ่งมักซ้ายมือเวลาเขียนเลข) ได้ก่อน ช่วยให้การอ่านเลขที่ต้องมีการเบรียบเทียบค่าบ้างครั้งทำได้รวดเร็วขึ้น (เพราะเห็นเครื่อง หรือ นัยสำคัญสูงก็ทราบได้ทันทีว่า ค่าได้มากกว่า นั้นเอง) ในทำนองกลับกัน สถาปัตยกรรมแบบ little endian จะทำให้การประมวลผลบางแบบ (เช่น การบวกผลทางคณิตศาสตร์) ทำได้รวดเร็วกว่า เพราะเมื่ออ่านข้อมูลจาก address ต่อไปสูง สามารถคำนวนเลข (เช่น บวก) จากตำแหน่งที่ไม่นัยสำคัญต่อไปก่อนได้ทันที โดยไม่ต้องรอให้อ่านข้อมูลได้ครบก่อน

ทั้งนี้สถาปัตยกรรม LADA ซึ่งกล่าวถึงในบทเรียนจะมีโครงสร้างแบบ big endian ในขณะที่ หน่วยประมวลผลกลาง Intel ตระกูล x86 ที่พับหัวไปนั้น จะใช้ระบบ little endian ดังนั้นในการพัฒนาซอฟต์แวร์บนสถาปัตยกรรมที่แตกต่างกัน ผู้พัฒนาจึงควรใช้ความระมัดระวังเสมอ (ปัจจุบันมักเป็นหน้าที่ของ compiler)

หมายเหตุ

1. ในสถาปัตยกรรมบางระบบ (เช่น ARM, PowerPC) สามารถจะกำหนดได้ว่าจะใช้ big endian หรือ little endian ได้ในขณะที่ระบบเริ่มทำงาน
2. นอกจากนี้ ยังมีบางสถาปัตยกรรมที่เป็นแบบ Middle-Endian ซึ่งมีการเรียงข้อมูลผสมระหว่าง big endian และ little endian ซึ่งไม่ค่อยพบในปัจจุบัน (จึงมีได้กล่าวถึงในที่นี้)

ประเด็นที่น่าสนใจ หลักการเบื้องต้น ที่จะช่วยลดความสับสนของ Big Endian และ Little Endian คือกรณี Big Endian ข้อมูลที่มีนัยสำคัญสูงจะอยู่ที่ address ต่อไป ส่วนกรณีของ Little Endian นั้น ข้อมูลที่มีนัยสำคัญต่อไปจะอยู่ที่ address ต่อไปกัน กล่าวคือ Big Endian จะสลับนัยสำคัญกับ address นั้นเอง

3.3.2 การกำหนดตำแหน่งเริ่มต้นของข้อมูล (Memory Alignment)

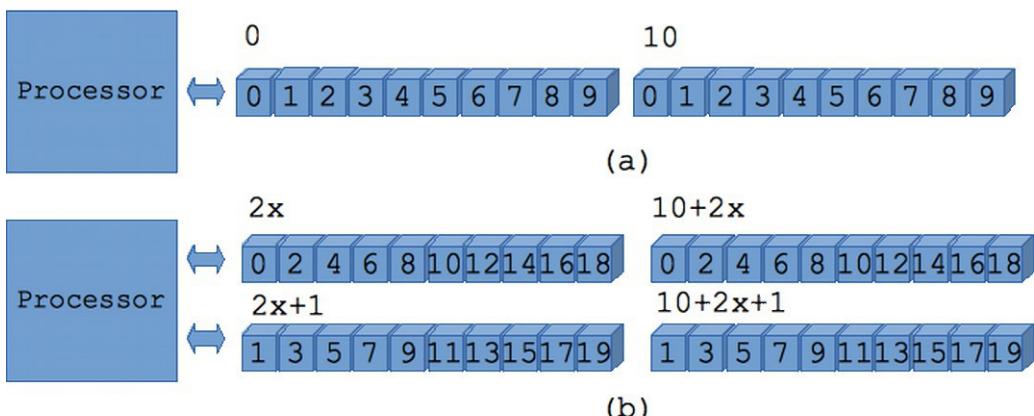
หลักการเบื้องต้นของการกำหนดตำแหน่งเริ่มต้นของข้อมูล หรือ Memory Alignment คือ ตำแหน่งเริ่มต้นของข้อมูล (ที่มีการอ้างอิงแบบ Word หรือ Half-Word) จะต้องเป็นค่าลงตัวของการหารด้วยเลขที่กำหนด เช่น หากกำหนดว่า alignment จะต้องหารด้วย 4 ลงตัว หมายความว่า จะต้องมีการกำหนดตำแหน่งเริ่มต้นเป็น 0h, 4h, 8h , Ch (ฐาน 16) การที่บังสถาปัตยกรรมมีการกำหนดเงื่อนไขในลักษณะนี้ เนื่องจากโครงสร้างของระบบ硬件และสายสัญญาณที่อยู่ภายใต้ระบบ อาจจะไม่เอื้ออำนวยต่อการอ่านหรือเขียนข้อมูลแบบไม่มี alignment

เพื่อประกอบความเข้าใจ ขอยกตัวอย่างการกำหนด alignment อ้างอิงดังนี้.... สมมุติให้กอล์ฟแต่ละกล่องแทนหน่วยความจำ 10 ชุด 1 แตร ในการนำกล่องดังกล่าวมาต่อพ่วงกันเพื่อให้ได้หน่วยความจำที่มากขึ้น สามารถทำได้หลายแบบ โดยแบบหนึ่งคือการต่อແລวติให้ยาวออกไป ซึ่งกรณีนี้ เราจะอ่านข้อมูลได้ทีละ 1 ชุดเท่าเดิม อีกแนวทางหนึ่งคือการนำกล่องดังกล่าวมาต่อข้างกัน เพื่อให้อ่านข้อมูล

ได้ครั้งละ 2 ชุด จะสังเกตว่า ประมาณข้อมูลที่มียังเท่าเดิมคือ 20 ชุด หากแต่ อ่านได้ครั้งละ 2 ชุด (ต่างจากแบบเดิมที่อ่านได้ครั้งละ 1 ชุด) การต่อทั้งสองแบบแสดงดังรูปที่ 3.7

จากรูป แบบแรก (a) เป็นการต่อแบบความกว้างสายสัญญาณเป็นหนึ่ง ซึ่งกรณีนี้หากค่าที่ต้องการใช้ เป็นเป็น half-word (ใช้ที่ละ 2 ชุด) หมายความว่า การอ้างข้อมูลจากหน่วยความจำจะต้องเกิดขึ้นสองครั้ง จึงจะทำการประมวลผลได้หนึ่งครั้ง¹³ ใน การต่อพ่วงหน่วยความจำในลักษณะนี้หน่วยประมวลผล ไม่จำเป็นจะต้องทำ memory alignment เพราะสามารถเข้าถึงข้อมูลได้เพียงที่ละ 1 ชุด แต่ต้องอาศัย การอ่านหลายครั้งจึงจะได้ชุดข้อมูลครบตามที่ register ต้องการ

ในแบบที่สอง (b) เป็นการขยายทางเดินข้อมูลให้กว้างขึ้นเป็นสองชุด หมายความว่า หน่วยประมวลผลสามารถอ่านข้อมูลได้มากขึ้นเป็นสองเท่า การอ่านข้อมูลเพื่อจะนำเข้าสู่ register จะใช้เวลาอ่อนอย่าง (เมื่อเทียบกับแบบ a) อย่างไรก็ตาม หากข้อมูลที่ต้องการอ่านอยู่คนละ block เช่น ต้องการอ่าน ข้อมูลที่เก็บอยู่ใน address 7 และ 8 เข้ามาพร้อมกันย่อมเป็นไปไม่ได้ เพราะตัวชี้ address (จากด้านบน) จะเป็นเลขคณิตชุดกัน การอ่านข้อมูลในลักษณะนี้ จะต้องอ่านสองครั้งและนำข้อมูลมาเรียงลำดับใหม่อยู่ดี (no alignment)

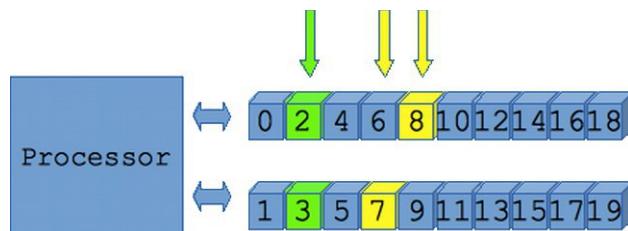


รูปที่ 3.7: การต่อหน่วยความจำเข้ากับหน่วยประมวลผล (a) ความกว้างสายสัญญาณเป็นหนึ่ง (b) ความกว้างสายสัญญาณเป็นสอง

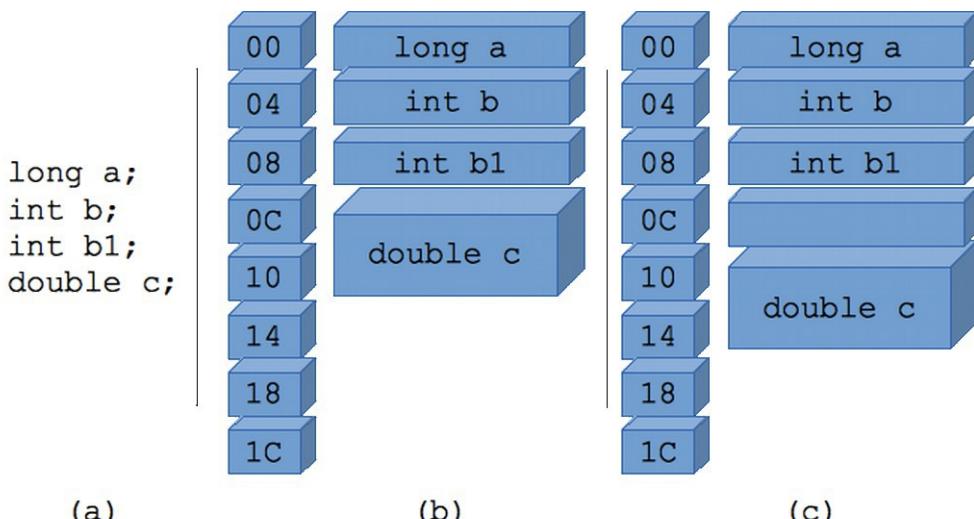
ด้วยเหตุนี้ หากสถาปัตยกรรมชุดคำสั่งสามารถบังคับหรือกำหนดให้ซอฟต์แวร์ต้องอ่านข้อมูลที่มี การทำ memory alignment เสมอ จะช่วยให้การอ่านข้อมูลสามารถอ่านได้เร็วขึ้น เช่น ทุกครั้งที่ประกาศ ตัวแปรที่มีความยาว 2 ชุด เพื่อให้อ่านและเรียกได้ย่างรวดเร็ว ตำแหน่งเริ่มต้นของตัวแปรดักล่า จะต้องหารด้วยสองลงตัว (รูปที่ 3.8 การอ่านค่าแบบมี restricted alignment เช่น ชุดที่ 2 และ ชุด

¹³ ในหน่วยประมวลผลกลางแบบของ intel แม่โคร์งสร้างภายในจะสามารถประมวลผลข้อมูลได้เร็ว แต่ใช้ทางเดินข้อมูลที่แคบเพื่อให้สามารถใช้กับอุปกรณ์และ mother board รุ่นเก่าได้ เช่น 80386SX ซึ่งเป็นสถาปัตยกรรมแบบ 32 บิต แต่ใช้ทางเดินข้อมูลแบบ 16 บิต เพื่อให้สามารถใช้อุปกรณ์ต่อพ่วงของ 80286 ซึ่งเป็นหน่วยประมวลผลกลางรุ่นก่อนหน้าได้ ในทำนองเดียวกับ 8088 ซึ่งเป็นหน่วยประมวลผลกลางแบบ 16 บิตแบบเดียวกับ 8086 แต่ใช้ทางเดินข้อมูลแบบ 8 บิต เพื่อประหยัดค่าใช้จ่าย

ที่ 3 และการอ่านค่าแบบ unrestricted alignment ในชุดที่ 7 และชุดที่ 8 เป็นต้น



รูปที่ 3.8: การอ่านค่าแบบมี restricted alignment และแบบ unrestricted alignment



รูปที่ 3.9: โครงสร้างการจัดการหน่วยความจำ (word addressing แสดงในเลขฐาน 16) (a) โปรแกรมภาษา C
(b) สถาปัตยกรรมแบบ unrestricted alignment และ (c) สถาปัตยกรรมแบบ restricted alignment

ข้อเสียที่เห็นได้ชัดของการที่สถาปัตยกรรมบังคับให้มีการทำ alignment คือ เรื่องของ fragmentation กล่าวคือ มีหน่วยความจำส่วนที่ว่างอยู่ แต่ไม่สามารถใช้ได้ เพราะจะต้องทำ alignment เพื่อให้ตำแหน่งเริ่มต้นหารด้วยเลขที่กำหนดลงตัว เพื่อประกบความเข้าใจ ลองดูตัวอย่าง alignment ของ compiler สำหรับภาษา C แบบ 32 บิตที่พับได้ทั่วไป ในที่นี่ตัวแปรแบบ int, long, float จะมีขนาด 4 ไบต์ ส่วน double มีขนาด 8 ไบต์ หากต้องการประกาศตัวแปร long 1 ชุด , int 2 ชุด และ double 1 ชุด ผลที่ได้ในสถาปัตยกรรมแบบ alignment และ ไม่มี alignment จะเป็นดังรูปที่ 3.9 ข้อสังเกตคือ บนสถาปัตยกรรมที่บังคับ alignment นั้น ข้อมูลแบบ 8 ไบต์ จะต้องเริ่มต้นที่ตำแหน่งที่ตำแหน่งที่ 8 หารด้วยแปดลงตัว ส่วนข้อมูลที่มีขนาด 4 ไบต์ จะต้องเริ่มต้นที่ตำแหน่งซึ่งหารด้วยสิบสองตัว เป็นต้น

3.4 การอ้างอิงตำแหน่งหน่วยความจำ (Addressing Mode)

การอ้างอิงตำแหน่งหน่วยความจำบน LADA แบ่งออกเป็นแบบ **Displacement** (บางตัวจะเรียกชื่อเต็มว่า **Base Displacement**) และ **Immediate** เป็นหลัก ในแบบ Displacement การอ้างอิงตำแหน่งคู่จะประกอบด้วยจำนวนเต็มคู่กับ register เสมอ เช่น 20 (\$r3) จะนำค่าใน register \$r3 มาบวกกับ 32 ก่อนจึงจะใช้อ้างอิงตำแหน่ง Address เช่น หากข้อมูลใน register ที่ \$r2 เป็น 300 คำสั่งดังกล่าวจะนำข้อมูลในตำแหน่งที่ 332 มาเก็บไว้ใน register \$r0

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

ส่วนการอ้างอิงแบบ Immediate คือการนำข้อมูลไปใช้ทันที โดยไม่จำเป็นต้องอ่านข้อมูลจากหน่วยความจำหรือ register (เปรียบเทียบได้กับค่าคงที่ที่ระบุในโปรแกรม) เช่น หากระบุ #4 ในคำสั่ง ADD มีความหมายว่า ให้นำค่า 4 มาบวกกับค่าใน register ที่กำหนด

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

ในสถาปัตยกรรมแบบอื่น อาจมีการอ้างอิงตำแหน่งหน่วยความจำที่ซับซ้อนมากกว่านี้ เช่น ระบบ **Base Index Addressing** หรือระบบ **Base Index Displacement** ซึ่งสามารถอ้างอิงข้อมูลในหน่วยความจำได้หลากหลายรูปแบบ เพื่อประกอบความเข้าใจ ลองดูการอ้างอิงตำแหน่งหน่วยความจำแบบต่างๆ ดังแสดงในตารางที่ 3.1 ทั้งนี้ addressing mode หลายแบบที่แสดงในตาราง มักจะไม่ค่อยพบในสถาปัตยกรรมสมัยใหม่แบบ Load-Store Architecture มาคนนัก เนื่องจากหลายแบบสามารถถูกแทนที่ได้ด้วย Displacement หรือ Register Indirect เมื่อมีการคำนวณตำแหน่งของ address ที่ต้องการอ้างอิงใส่ไว้ใน register ก่อน (โปรแกรมที่ได้มีจำนวนคำสั่งมากขึ้น เพื่อชดเชยกับ addressing mode ที่ขาดหายไป)

นอกจากนี้ยังมีระบบอ้างอิงตำแหน่งหน่วยความจำแบบ PC relative ซึ่งมีการทำงานคล้ายกับ displacement แต่ใช้ PC เป็น register อ้างอิงแทน การทำงานในแบบ PC relative มักใช้กับคำสั่งในกลุ่มประยุคเงื่อนไข เช่น Jump หรือ Branch (เปรียบเทียบได้กับ IF .. THEN .. ELSE, และ SWITCH)

ข้อสังเกตุ

1. หากกำหนดค่า displacement เป็น 0 ผลที่ได้จะเหมือน register indirect
2. หากทำการคำนวณค่าของตำแหน่งที่ต้องการใส่ใน register ก่อน จะสามารถใช้ addressing mode แบบ register indirect แทนได้ทุกแบบ
3. ในแบบ scale เนื่องจาก E และ F เป็น integer ซึ่งมีขนาด 4 byte ดังนั้น ค่า i ซึ่งเป็น index จะต้องคูณ 4 เสมอจึงจะได้ตำแหน่งที่ต้องการ

แม้การมี addressing mode เพียงไม่กี่แบบ แล้วต้องอาศัย compiler และ การคำนวณที่มากขึ้น และผลที่ได้ในทางหนึ่งคือ คำสั่งมีความซับซ้อนน้อยลง และสามารถประมวลผลแบบงานได้มากขึ้น (ซึ่งจะกล่าวถึงในเนื้อหาส่วนถัดไป)

Addressing Mode	ສັນໄຍງ່ ອະນຸຍາ (Assembly)	RTL	ເປົ້າໃຫຍ່ ບກໍບານ ການພຳຕັ້ງ C ¹⁴
1. Immediate	LDI \$r1, #4	\$r1 ← 4	A = 4
2. Register Direct	ADD \$r1, \$r2, \$r3	\$r1 ← \$r2 + \$r3	A = B + C
3. Displacement	LD \$r1, 20(\$r2)	\$r1 ← MEM[100+\$r2]	A = D[100]
4. Register Indirect	LD \$r1, (\$r2)	\$r1 ← MEM[\$r2]	A = *p
5. Index	LD \$r1, (\$r2 + \$r3)	\$r1 ← MEM[\$r1 + \$r2]	A = D[i]
6. Direct	LD \$r1, 1000	\$r1 ← MEM[1000]	
7. Memory Indirect	LD \$r1, @(\$r2)	\$r1 ← MEM[MEM[\$r2]]	
8. AutoIncrement	LD \$r1, (\$r2)+	\$r1 ← MEM[\$r2] \$r2 ← \$r2 + c	A = D[i++]
9. AutoDecrement	LD \$r1, (\$r2)-	\$r1 ← MEM[\$r2] \$r2 ← \$r2 - c	A = D[i--]
10. Scale	LD \$r1, (\$r2)[\$r3*4]	\$r1 ← MEM[\$r2+\$r3*4]	F = E[i]

ตารางที่ 3.1: addressing mode แบบต่างๆ และการเทียบเคียงกับภาษา汇程式ดับสัมผัส

3.5 รูปแบบคำสั่ง (Instruction Format)

คำสั่งภาษาแอสเซมบลีทุกคำสั่ง จะถูกแปลเป็นภาษาเครื่องโดย ภาษาเครื่องจะมีรูปแบบคำสั่งที่เป็นแบบแผน ซึ่งแบบแผนเหล่านี้เรียกว่า รูปแบบคำสั่ง หรือ Instruction Format เนื่องจากภาษาเครื่องนั้นจะประกอบด้วยสายข้อมูลบิต ซึ่งมีเพียง 0 และ 1 มาต่อ กันเท่านั้น รูปแบบคำสั่งที่แปลจากภาษาแอสเซมบลีเป็นภาษาเครื่องจึงใช้การแบ่งสายข้อมูลบิตออกเป็นช่วง แล้วกำหนดความหมายให้แต่ละ

14 ในตัวอย่างนี้ กำหนดให้ A, B, C เป็นตัวแปรเป็น byte (หรือ unsigned char) , D เป็น Array ของ byte , E เป็น Array ของ integer, F เป็น integer (1 integer มีขนาด 4 byte), และ p เป็น ตัวชี้ (pointer), i เป็น index ของ array

ช่วงบิต

กรณีของ LADA นั้น คำสั่งทุกคำสั่งจะมีความยาว 32 บิตเท่ากันหมด สถาปัตยกรรมลักษณะนี้คือ สถาปัตยกรรมที่มีความยาวคำสั่งแบบคงที่ (fixed instruction length¹⁵) สถาปัตยกรรมบางแบบ มีความยาวคำสั่งแตกต่างกันไปในแต่ละคำสั่งตั้งแต่ 8 – 17 byte ซึ่งสถาปัตยกรรมลักษณะนี้เรียกว่า สถาปัตยกรรมที่มีความยาวคำสั่งแบบผันแปร (variable instruction length¹⁶) นอกจากนี้ ยังมี สถาปัตยกรรมบางรูปแบบที่มีความยาวคำสั่งแบบลูกผสม (hybrid instruction length)¹⁷ ซึ่งจะมี โหมดการทำงานพิเศษที่มีความยาวคำสั่งแตกต่างกันไป

ในสถาปัตยกรรม LADA มีรูปแบบคำสั่งทั้งสิ้น 3 รูปแบบเท่านั้น คือ แบบ R, I และ J (ขออ้างอิงถึงโดยเรียกว่า R-format, I-format และ J-format ตามลำดับ) โดยแต่ละรูปแบบจะมีการตีความต่างกันไปตามรูปแบบคำสั่งดังนี้

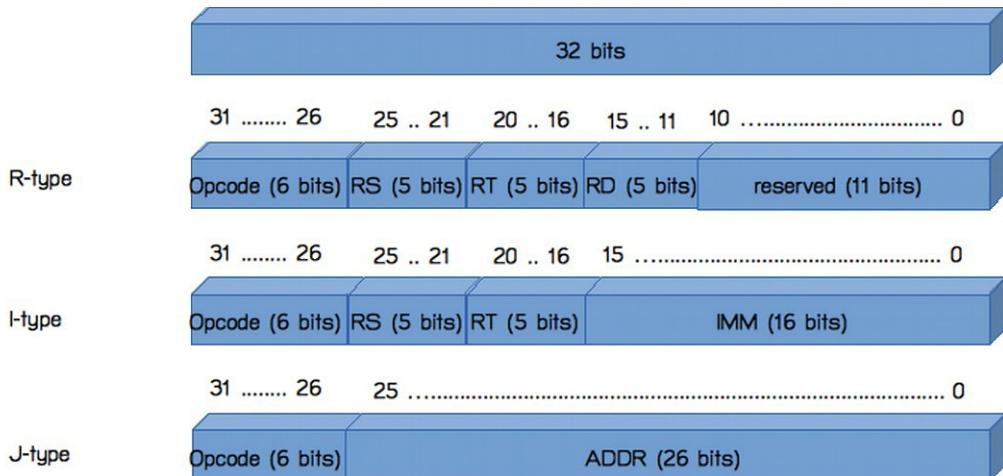
- **R-type** สำหรับการเข้ารหัสคำสั่งที่เกี่ยวกับ register มี field ที่เกี่ยวข้องได้แก่ op, rs, rt, และ rd
- **I-type** สำหรับการเข้ารหัสคำสั่งที่มีการอ้างอิงค่า immediate มี field ที่เกี่ยวข้องได้แก่ op, rs, rt และ immediate
- **J-type** สำหรับการ jump หรือ branch แบบ absolute เป็นหลัก

รายละเอียดรูปแบบคำสั่งที่พบใน LADA แสดงได้ดังรูปที่ 3.10

15 สถาปัตยกรรมที่มีรูปแบบคำสั่งแบบคงที่ (fixed length instruction) มักจะสามารถอ่านคำสั่งได้ภายใน 1 cycle ทำให้การอ่านคำสั่งแต่ละอันใช้เวลาคงที่ ตัวอย่างสถาปัตยกรรมลักษณะนี้ เช่น PowerPC, MIPS, ARM, SPARC เป็นต้น

16 สถาปัตยกรรมที่มีรูปแบบคำสั่งแบบผันแปร (variable length instruction) จะใช้จำนวน cycle ในการอ่านคำสั่ง แต่ละอันไม่คงที่ โดยส่วนแรกของคำสั่งที่อ่านได้มักจะมีบิตที่ใช้บอกความยาวของคำสั่งทั้งหมด ตัวอย่าง สถาปัตยกรรมลักษณะนี้ เช่น Intel x86, VAX เป็นต้น

17 สถาปัตยกรรมที่มีรูปแบบคำสั่งแบบผสม (hybrid length instruction) มักจะเป็นสถาปัตยกรรมแบบ fixed length instruction ที่มีโหมดการทำงานพิเศษที่รับคำสั่งที่มีความยาวเฉพาะ เช่น ARM Thumb ซึ่งเมื่อทำงานในแบบ thumb หน่วยประมวลผลจะทำการประมวลคำสั่งที่มีความยาวแบบ 16 บิตแทน (ปกติ ARM instruction จะมีความยาว 32 บิต)



รูปที่ 3.10: รูปแบบคำสั่ง (instruction format) ในสถาปัตยกรรม LADA

ทั้งนี้การเข้ารหัสคำสั่ง จึงอยู่กับผู้ออกแบบแต่ละรายจะกำหนด (ดูรายละเอียดเพิ่มเติมได้จากคู่มือของโครงสร้างชุดคำสั่งของแต่ละระบบ) เช่น คำสั่ง ADD อาจแทนด้วย 000001 คำสั่ง SUB อาจแทนด้วย 000010 เป็นต้น โดยหลักเกณฑ์ในการกำหนดรหัสต่างนั้น จึงอยู่กับแนวทางในการออกแบบ วงจรเพื่อตีความคำสั่งและสร้างสัญญาณควบคุม

3.6 สถาปัตยกรรมชุดคำสั่งที่ดี

ในปัจจุบันมีสถาปัตยกรรมชุดคำสั่งที่ใช้อยู่พร้อมหลายพื้นที่ หากวิเคราะห์สถาปัตยกรรมชุดคำสั่งที่มีในปัจจุบัน (ปี พ.ศ. 2557) จะเห็นว่ามีสถาปัตยกรรมชุดคำสั่งที่เห็นรอบตัว เช่น สถาปัตยกรรม Intel x86_64 ซึ่งพับในเครื่องคอมพิวเตอร์ส่วนบุคคลทั่วไปและเครื่องตระกูล macintosh ของ apple สถาปัตยกรรม ARM ที่มีการใช้งานพร้อมๆ กันในโทรศัพท์เคลื่อนที่แบบ smart phone และ สถาปัตยกรรมแบบ AVR ที่มีการใช้งานใน Arduino หรือระบบคอมพิวเตอร์แบบฝังตัวส่วนใหญ่ ที่ยกตัวอย่างมาเนี้เป็นเพียงส่วนหนึ่งเท่านั้น

คราวนี้หากจะต้องวิเคราะห์หรือข้อดีข้อเสีย ควรจะมีหลักเกณฑ์ในการประเมินอย่างไรบ้าง หลักในการ評価สถาปัตยกรรมชุดคำสั่งได้เป็นสถาปัตยกรรมชุดคำสั่งที่ดีนั้น มีแนวคิดที่หลากหลาย แต่ละคำรามักให้แบ่งปันในการวิเคราะห์ที่แตกต่างกัน อย่างไรก็ตามจาก การศึกษาจากหลายตำรา พอกจะสรุปเป็นหลักการได้ดังนี้

1. **มีความเข้ากันได้กับสถาปัตยกรรมรุ่นก่อนหน้า** (Portability หรือ Compatibility) กล่าวคือ มีการใช้งานได้นาน ตัวอย่างที่เห็นได้ชัด เช่น intel x86 ซึ่งมีอายุเก่าแก่กว่า 40 ปี ยังคงทำงานได้บนสถาปัตยกรรมรุ่นใหม่ของ intel
2. **มีความครอบคลุม** (Generality หรือ Completeness) กล่าวคือ มีคำสั่งเพียงพอสำหรับการใช้งานทั่วไป เช่น ในยุคหนึ่งหน่วยประมวลผลกลางหรือสถาปัตยกรรมชุดคำสั่งหลักจะไม่มีคำสั่งเกี่ยวกับการคำนวนเลขทศนิยม (floating-point unit) ทำให้การทำงานที่เกี่ยวข้องกับ

Graphics เป็นไปได้ยาก เป็นต้น

3. มีความตรงไปตรงมา (Orthogonal) กล่าวคือ มีการทำงานที่ชัดเจน ไม่มีกลุ่ม register พิเศษหรือความหมายพิเศษมากเกินจำเป็น
4. ช่วยให้สามารถพัฒนาซอฟต์แวร์ได้ง่าย (Ease of Compilation) ทั้งนี้เนื่องจากสถาปัตยกรรมชุดคำสั่งคือมุ่งมองของหน่วยประมวลผลกลางจากผู้พัฒนาซอฟต์แวร์ ดังนั้นคุณสมบัติที่ดีอย่างหนึ่งของหน่วยประมวลผลกลางคือ สามารถพัฒนาซอฟต์แวร์ได้ง่าย
5. ช่วยให้สามารถนำร่องได้ง่าย (Ease of Implementation) ในทำนองเดียวกัน เนื่องจาก ardware จะต้องสอบสวนการทำงานของสถาปัตยกรรมชุดคำสั่ง ดังนั้นสถาปัตยกรรมชุดคำสั่งที่ดีจึงควรทำให้ ardware สามารถทำงานได้ง่าย ยิ่งไปกว่านั้นปัจจุบัน ardware มักมีความเป็นสถาปัตยกรรมเชิงขนาดสามารถทำงานได้หลายคำสั่งพร้อมกัน ซึ่งสถาปัตยกรรมชุดคำสั่งที่ดี จะช่วยให้การประมวลผลแบบขนาด (Streamlined) ทำได้ง่ายขึ้นด้วย

อย่างไรก็ตาม นี่เป็นเพียงคุณสมบัติเบื้องต้นเท่านั้น หากจะขยายความสถาปัตยกรรมที่มีความหลากหลายในการพัฒนาซอฟต์แวร์ อาจจะต้องวิเคราะห์ต่อว่าคอมไพลเลอร์ (compiler) หรือตัวแปลงภาษาที่มีการทำงานอย่างไร และสถาปัตยกรรมลักษณะไหนที่จะช่วยให้ทำการแปลงได้ง่ายขึ้น ซึ่งขั้นตอนการทำงานของคอมไпалเลอร์นั้น อยู่นอกเหนือขอบเขตของหนังสือเล่มนี้ อย่างไรก็ตาม เพื่อประกอบความเข้าใจ จะขออธิบายบทบาทของคอมไпалเลอร์ดังต่อไปนี้

3.6.1 บทบาทของคอมไпалเลอร์ในการจัดเรียงคำสั่ง (Optimization)

เนื่องจากประสิทธิภาพของระบบคอมพิวเตอร์ส่วนใหญ่ ขึ้นกับความสามารถในการจัดเรียงของคอมไпалเลอร์ ดังนั้นหากผู้ออกแบบสถาปัตยกรรมชุดคำสั่งควรมีความเข้าใจในบทบาทของคอมไпалเลอร์เพื่อเป็นข้อมูลประกอบการออกแบบสถาปัตยกรรมชุดคำสั่งที่ดี ทั้งนี้เนื่องจากที่ระบุในส่วนนี้ เป็นเพียงข้อมูลเบื้องต้นเพื่อประกอบการศึกษาเท่านั้น ในทางปฏิบัติการทำการจัดเรียงลำดับมีสองวัตถุประสงค์ (ให้เลือกอย่างใดอย่างหนึ่ง) คือ 1) เพื่อให้ซอฟต์แวร์ที่ได้ทำงานเร็วขึ้น หรือ 2) เพื่อให้ซอฟต์แวร์ที่ได้มีขนาดเล็กลง ซึ่งทั้งสองอย่างมักให้ผลตรงข้ามกันเสมอ (ซอฟต์แวร์ที่เร็วขึ้นมักจะมีขนาดใหญ่ด้วย)

เพื่อให้ซอฟต์แวร์มีประสิทธิภาพมากขึ้น ขั้นตอนหนึ่งในการทำงานของคอมไпалเลอร์คือ การจัดเรียงลำดับ (optimization) ของโปรแกรม โดยการจัดเรียงลำดับแบ่งออกได้เป็นสองลักษณะคือ การจัดเรียงลำดับแบบท้องถิ่น (local optimization) และ การจัดเรียงลำดับแบบรวม (global optimization)

ก่อนจะทำความเข้าใจเรื่องการจัดเรียงลำดับ ควรจะต้องเข้าใจก่อนว่า ในทางคอมไпалเลอร์ จะทำการแบ่งโปรแกรมออกเป็นส่วนย่อยเรียกว่า block โดยส่วนที่เล็กสุดของการทำงานเรียกว่า basic block ซึ่งคือ block ที่มีทางเข้า(ริมทำงาน) เพียงทางเดียวคือด้านบน และทางออกเพียงทางเดียวคือด้านล่าง นั่นหมายความว่าทุกรุ่งที่มีการเปลี่ยนลำดับการทำงาน (เช่น ด้วยประโยคเงื่อนไข IF.. THEN .. ELSE) หรือเรียกโปรแกรมย่อย เป็นอันสิ้นสุด basic block.

จากคำจำกัดความของ basic block ช่วยให้อธิบายความหมายของ การเรียงลำดับแบบท้องถิ่น (local optimization) และ การเรียงลำดับแบบรวม (global optimization) ได้ว่า การเรียงลำดับแบบท้องถิ่น คือการเรียงลำดับภายใน basic block เดียวกันเท่านั้น การเรียงลำดับแบบรวม คือการเรียงลำดับข้าม basic block

ตัวอย่างการจัดเรียงลำดับแบบท้องถิ่น (local optimization)

1. **Common expression elimination** ตัวอย่าง เช่น มี $b+c$ อยู่ในหลายคำสั่ง ดังนี้ หากคำนวณ $b+c$ ไว้ก่อนอาจจะทำให้การคำนวนเร็วขึ้น ($\text{เช่น } a = b + c + d; \text{ และ } y = b + c + e; \text{ หาก } x = b+c \text{ ไว้ก่อน จะได้ว่า } a = x + d; \text{ และ } y = x + e; \text{ ซึ่งทำงานได้เร็วกว่า}$)
2. **Constant propagation** ในกรณีนี้คือ ค่าบางอย่างสามารถแทนที่ได้ด้วยค่าคงที่ ก็ควรจะแทนที่ไปในระหว่างทำการแปลทันที ($\text{เช่น } m = 60; h = 60 * 60; d = 24*m; \text{ days} = \text{seconds} / d; \text{ จะเห็นว่าเป็นการแปลงวินาทีเป็นเวลา หากแต่ตัวแปร } m, h \text{ และ } d \text{ ไม่มีการใช้งานที่อื่นอีก จึงอาจจะแทนที่โดยโยคทั้งหมดได้ด้วย } \text{days} = \text{seconds} / 86400; \text{ เป็นต้น}$)

ตัวอย่างการจัดเรียงลำดับแบบรวม (global optimization)

1. **Global common expression elimination** ทำนองเดียวกันกับ common expression elimination แบบ local หากแต่เป็นการท่าระหว่าง basic block
2. **Code motion** กรณีนี้คือ ค่าบางอย่างอาจไม่เกิดการเปลี่ยนแปลงใน loop (เรียกว่า loop invariant) ดังนั้นหาก **ย้ายคำสั่งกล่าวไปคำนวนก่อนเข้า loop** จะช่วยให้การคำนวนใน loop แต่ละรอบน้อยลง และ code ทำงานได้เร็วขึ้น

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

หากจะวิเคราะห์สั้นๆ จะได้ว่า คุณลักษณะของสถาปัตยกรรมที่ดีจากมุมมองของคอมไพลเลอร์ คือ

1. **มีความเรียบง่าย (Simplicity)** โดยสถาปัตยกรรมมีส่วนประกอบพื้นฐาน (primitive) ให้จากนั้นคอมไพลเลอร์สามารถเอาส่วนประกอบพื้นฐานนั้นมาประกอบกันเพื่อทำงานที่ซับซ้อนได้
2. **มีความตรงไปตรงมา (Orthogonal หรือ Regularity)** ความหมายของแต่ละ field ในการเข้ารหัสคำสั่ง ตรงไปตรงมา ไม่มี mode พิเศษ
3. **มีแนวทางในการเลือกใช้ที่ชัดเจน (Simplify tradeoffs)** ในบางกรณี เราสามารถจะทำงานให้สำเร็จได้มากกว่า 1 วิธี เช่น การกำหนดให้ตัวแปรมีค่าเป็นศูนย์ อาจทำได้โดยการใช้คำ

สั่ง CLR หรือ XOR ข้อมูลเดิมก็ได้ ซึ่งสถาปัตยกรรมที่ดี ควรมีตัวเลือกที่ชัดเจน

4. อนุญาตให้กำหนดค่าตอนแปลภาษา (compile) ได้ เช่น ในขณะที่แปลภาษาอาจทราบได้ทันทีว่าจะต้องนำค่าคงที่เข้าคูณกับตัวแปรเพื่อให้ได้ผลลัพธ์ แต่หากสถาปัตยกรรมไม่รองรับการทำงานของค่า immediate ย่อมเป็นไปไม่ได้ที่จะระบุค่าคงที่ดังกล่าวเข้าไปในโปรแกรม เป็นต้น

3.7 การเรียกใช้โปรแกรมย่อย และ *Exception*

ประเด็นหนึ่งที่สำคัญแต่ไม่เป็นส่วนหนึ่งของสถาปัตยกรรมชุดคำสั่งโดยตรง (มีตัวช่วยในสถาปัตยกรรมชุดคำสั่ง แต่ต้องมีข้อตกลงอื่นเพิ่มด้วย) คือ การเรียกใช้งานโปรแกรมย่อย ซึ่งเป็นส่วนหนึ่งของคำสั่งควบคุม ข้อสำคัญที่ทำให้การเรียกใช้โปรแกรมย่อยแตกต่างจากคำสั่งทั่วไปคือ การที่จะต้องจำ Address สำหรับการ Return กลับมายังที่เดิม ซึ่งสถาปัตยกรรมชุดคำสั่ง มักจะมีตัวช่วยในการเรียกโปรแกรมย่อยเป็นคำสั่ง Call และ Return ซึ่งอาจมีการทำงานดังนี้

ความหมาย \$r13 ← PC + 4;

PC \leftarrow addr;

ແລະ

RET ; (*J-type*)

ความหมาย PC ← \$r13;

ในการนิของสถาปัตยกรรมชุดคำสั่ง LADA ค่า \$r13 เป็น Register ที่ถูกกำหนดให้เก็บค่า Return Address ดังนั้นทุกครั้งที่มีการ Call ค่า Return Address จะถูกนำมายัง Register นี้ บางสถาปัตยกรรมชุดคำสั่ง อาจมี register พิเศษอันอื่น เพื่อทำหน้าที่ดึงกล่าว และในสถาปัตยกรรมบางแบบ จะเลือกใช้ Memory เป็นที่พัก Return Address แทน (เช่น Intel x_86) ซึ่งการเลือกดังสินใจว่าจะใช้แบบใด ย่อมส่งผลต่อประสิทธิภาพโดยรวมของระบบ เช่นกัน

ข้อคิด ในการนับที่ใช้ Register ในการเก็บ return address หากต้องการเรียก sub routine ย่อย่อด้วยเครื่องหมาย recursive จะต้องทำอย่างไรเพื่อให้สามารถจดจำ address ได้เมื่อไหร่ก็ได้

3.7.1 การส่งผ่านและคืนค่าระหว่างโปรแกรมย่อย

นอกจากนี้ ยังมีสิ่งที่ซอฟต์แวร์ต้องการทำในการเรียกใช้โปรแกรมย่อยคือ 1) การส่งผ่านค่า arguments หรือ parameters และ 2) การคืนค่า ซึ่งมีได้มีส่วนไม่แตกล่างไว้ในสถาปัตยกรรมชุดคำสั่งว่า จะต้องส่งผ่านค่า และ คืนค่าอย่างไร เพื่อให้เป็นมาตรฐาน หน่วยประมวลผลกลางมักจะมีคุณสมบัติชุดหนึ่งเรียกว่า Application Binary Interface (ABI) เพื่อกำหนดว่าจะผ่านค่าและคืนค่าอย่างไร

(เดิม มาตรฐานนี้จะกำหนดโดยระบบปฏิบัติการ และ/หรือ คอมไไฟเลอร์แทน)

ลักษณะการส่งผ่าน และคืนค่า� แบ่งเป็นกลุ่มใหญ่ๆ ได้ดังนี้

1. การส่งค่าและคืนค่าผ่าน register กรณีนี้ ABI จะต้องกำหนดมาตรฐานว่า register ได้ใช้สำหรับการคืนค่า และ register ใด (เรียงลำดับอย่างไร) ให้สำหรับการผ่านค่าอย่างไร ตาม register มักมีข้อจำกัดในการส่งข้อมูลขนาดเล็ก หากต้องส่งข้อมูลขนาดใหญ่ จะต้องนำข้อมูลมาใส่ในหน่วยความจำ แล้วส่งผ่าน address ของหน่วยความจำที่ใช้ register แทน
2. การส่งผ่านด้วยหน่วยความจำ ซึ่งจะมีการกำหนดว่า ก่อนการเรียกใช้โปรแกรมย่อๆ จะต้องทำการ push ค่า arguments ลงใน stack และเว้นที่ไว้สำหรับการรับค่าคืน ก่อนจะที่มีการเรียกโปรแกรมย่อๆ (ซึ่งลักษณะนี้เป็นลักษณะที่นิยมใช้โดยทั่วไปสำหรับ intel x_86)

เพื่อประกอบความเข้าใจ ขอให้ผู้เรียนลองทำแบบฝึกหัดท้ายบทเพิ่มเติม

3.7.2 การบันทึกค่า Local Variable และ Register

เนื่องจากในทุกโปรแกรมย่อๆ จะมีการใช้งาน local variable และ register เสมอ ดังนั้นเพื่อให้สามารถลับมาทำงานได้เหมือนเดิม จึงต้องมีการจำค่าดังกล่าวไว้ด้วย ตัวอย่างเช่น ในโปรแกรมหลัก มีการใช้งาน \$r1 แทน a และ \$r2 แทน b อยู่ หากโปรแกรมย่อต้องมีการใช้งาน \$r1 แทน x ด้วย ก่อนการใช้งาน จะต้องมีการจำค่า a ก่อน และ หลังการใช้งานจึงจะคืนค่า a กลับมาที่ \$r1

ประเด็นเพื่อการพิจารณาเมื่อยิ่งว่า หน้าที่ในการจำค่า register ที่อาจมีการเปลี่ยนแปลงในระหว่างการทำงานนั้นเป็นของใคร ระหว่าง โปรแกรมหลักหรือผู้เรียก (caller) และ โปรแกรมย่อๆ หรือผู้ถูกเรียก (callee) อันนี้เป็นอีกปัจจัยที่หนึ่งต้องระบุไว้ใน ABI เพื่อกัน เพื่อให้เห็นข้อดีข้อเสีย ของแต่ละอย่างประกอบการอธิบายดังรูปที่ 3.11

จากรูปที่ 3.11 จะเห็นว่า กรณี Caller เป็นผู้ Save ค่า� Caller จะต้องทราบว่า SUB ใช้ Register อะไรบ้าง ซึ่งหาก Caller ไม่ทราบ ก็จะต้องทำการ save ค่า register ทุกค่า (ເහັນໃນกรณีนี้คือ 32 ค่า) และ หลังจากลับมาจาก SUB จะต้องทำการ POP ทั้ง 32 ค่า ในกรณีกลับกัน หาก Callee เป็นผู้ Save ค่า� Callee สามารถจะบันทึกเพียงค่า Register ที่ตนเองใช้ และเว้นค่าอื่นที่ไม่เกี่ยวข้องไว้ได้เป็นต้น

หากวิเคราะห์ในกรณีนี้ จะเห็นว่า Callee Save อาจจะเป็นตัวเลือกที่ดีกว่าในเชิงของประสิทธิภาพ แต่ สถาปัตยกรรมหลายอัน ก็เลือกใช้ Caller Save แทน ส่วนหนึ่งอาจด้วยเหตุผลว่าสถาปัตยกรรมชุดคำสั่งในอดีต มักมี register ไม่มาก ดังนั้น การจะเก็บค่าโดย Caller จึงมีได้ส่งผลกระทบต่อประสิทธิภาพของระบบมากนัก

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

รูปที่ 3.11: เปรียบเทียบการทำงานระหว่าง Caller Save และ Callee Save

3.7.3 Exception และ Interrupt

Exception และ Interrupt คือ การทำงานที่ไม่ได้เกิดจากโปรแกรมโดยตรง (unprogrammed control flow) กล่าวคือ เป็นสิ่งที่เกิดขึ้นจากเหตุอื่นซึ่งทำให้หน่วยประมวลผลกลางต้องหยุดการทำงานชั่วคราว หลายตำนานยมแยกคำโดยใช้ Exception แทนสิ่งที่เกิดขึ้นจากการทำงานของซอฟต์แวร์ (เช่น การหารด้วยศูนย์ การประมวลผลคำสั่งที่ไม่รู้จัก หรือ การอ่านข้อมูลที่ไม่มีอยู่จริง เป็นต้น) และเรียก Interrupt ว่าเป็นสิ่งที่เกิดจากฮาร์ดแวร์ (เช่น สัญญาณนาฬิกาของระบบมีการเปลี่ยนแปลง มีผู้กด keyboard หรือป้อน input ให้ระบบ เป็นต้น)

หากวิเคราะห์เบื้องต้นจะเห็นว่าการทำงานของ exception หรือ interrupt จะคล้ายคลึงกับการทำงานของการเรียกโปรแกรมย่อย หากแต่ ข้อแตกต่างคือ ตำแหน่งของโปรแกรมย่อนั้น จะเรียกว่า handler และจะต้องมีการกำหนดลงหน้าไว้ก่อน และก่อนการประมวลผล handler จะต้องมีการจำค่า สถานะอื่นๆ (FLAG เช่น zero) เพิ่มเติมจากค่า PC ปัจจุบันด้วย และการจบ exception มักจะต้องทำด้วยคำสั่งพิเศษคือ IRET ซึ่งจะมีการคืนค่า FLAG เพิ่มเติมจากคำสั่ง RET ด้วยเช่นกัน ในทำนองเดียวกับ การเก็บค่า FLAG เหล่านี้ อาจเก็บใน register พิเศษ หรือเก็บไว้ใน stack memory ก็ได้ ทั้งนี้แล้วแต่การออกแบบของสถาปัตยกรรมชุดคำสั่งที่กำหนด ซึ่งมีข้อดีข้อเสียแตกต่างกันไปคล้ายกับกรณีการเก็บ return address เช่นกัน

ในการบอกตำแหน่งของ handler routine มีวิธีการบอกอยู่ด้วยกัน 3 แนวทางใหญ่คือ

1. **vector table** เป็นการสร้างตาราง เพื่อบอกว่า address ของโปรแกรมย่อยสำหรับเป็น handler อยู่ที่ใด (ตารางเก็บ address)
2. **handler table** คล้ายกับวิธีการ vector table เพียงแต่ในตาราง จะเป็น **handler routine** เลย เช่น exception 0 จะไปยัง address 0x0000 , exception 1 จะไปยัง address จะไปยัง address 0x0010 เป็นต้น (1 ช่องของ table มีขนาด 16 byte) (ตารางเก็บ code)
3. **fix entry** มีการกำหนดตำแหน่งตายตัว จะต้องใช้ software ในการจัดการ ซึ่งมักจะมี register พิเศษเพื่อใช้บอกแหล่งที่มาของ exception ควบคู่ด้วย

ข้อคิด ในการนิยอง handler table นั้น ตารางที่ใช้เก็บ routine จะมีขนาดเล็ก (เพียง 16 byte) ซึ่งอาจจะเก็บคำสั่งได้เพียง 4 คำสั่งเท่านั้น หาก handler routine มีขนาดใหญ่มาก จะมีแนวทางในการแก้ปัญหาอย่างไร

3.8 สรุป

สถาปัตยกรรมชุดคำสั่ง เป็นมาตรฐานที่ใช้สื่อระหว่างการพัฒนา hardware และซอฟต์แวร์ ความซับซ้อนของคำสั่งเป็นเพียงปัจจัยหนึ่งในการออกแบบเท่านั้น ทั้งนี้หากคำสั่งมีความซับซ้อนมากขึ้น อาจจะส่งผลให้คำสั่งใช้ CPI มากขึ้น (เนื่องจากวงจรที่มีขนาดใหญ่และมีสถานะรวมถึงขั้นตอนการทำงานมากขึ้น) ทำให้วางจารดังกล่าวไม่สามารถใช้ Clock Rate ที่มีความถี่สูง ปัจจุบันยังไม่มีมาตรฐานเดียวในการกำหนดว่าสถาปัตยกรรมชุดคำสั่งนั้นต้องก่อสร้างสถาปัตยกรรมชุดคำสั่งอื่นหรือไม่อย่างใด การจะบอกว่าสถาปัตยกรรมใดเป็นสถาปัตยกรรมที่ดี ขึ้นอยู่กับมุมมองของผู้ใช้งาน เช่น ในบางกรณีผู้ใช้งานและผู้ออกแบบอาจจะเลือก compatibility เพื่อให้หน่วยประมวลผลรุ่นใหม่สามารถประมวลผลชุดคำสั่งเดิมได้ ซึ่งให้ซอฟต์แวร์เดิมสามารถทำงานบนหน่วยประมวลผลรุ่นใหม่ได้ทันที ขณะที่บางครั้งผู้ออกแบบเลือกที่จะทิ้ง compatibility เพื่อให้ได้ประสิทธิภาพที่ดีกว่า เป็นต้น

3.9 แบบฝึกหัดท้ายบท

1. สถาปัตยกรรมชุดคำสั่งคืออะไร มีสิ่งใดที่ต้องกล่าวถึงบ้าง และเป็นประโยชน์ในการพัฒนาซอฟต์แวร์ และ ฮาร์ดแวร์หรือไม่ อย่างไร
2. Opcode และ Operand คืออะไร
3. หากระบบคอมพิวเตอร์มีการอ้างอิงหน่วยความจำโดยใช้ Address ทั้งสิ้น 32 bit จะสามารถจ้างอิงข้อมูลแบบ Word Address ໄทั้งสิ้นที่ Word
4. ระบบ Big endian และ Little endian มีข้อแตกต่างกันอย่างไร และแต่ละแบบมีข้อดีข้อเสียอย่างไร
5. การที่หน่วยประมวลผลมีระบบการอ้างอิงหน่วยความจำหลากหลายรูปแบบ เป็นประโยชน์หรือไม่ อย่างไร
6. หากสถาปัตยกรรมชุดคำสั่งที่กำหนดให้ มีเพียงระบบการอ้างอิงหน่วยความจำแบบ register indirect และ immediate ให้เท่านั้น เราจะสามารถใช้งานสถาปัตยกรรมชุดคำสั่งดังกล่าวในการพัฒนาโปรแกรมทั่วไปได้หรือไม่ อย่างไร จงอธิบายพร้อมให้เหตุผลประกอบ
7. การที่ Instruction Format ทุกรูปแบบมีความยาวคำสั่งเท่ากันหมด มีข้อดีข้อเสียอย่างไร
8. ภาษาแอกซ์เชมบลีและสถาปัตยกรรมชุดคำสั่งมีข้อเหมือนหรือข้อแตกต่างกันหรือไม่ อย่างไร
9. จากมุมมองของคอมไพลเลอร์ สถาปัตยกรรมชุดคำสั่งที่ดี ควรมีคุณสมบัติอย่างไร
10. จงแสดงการจัดเรียงข้อมูลข้อมูลตามลำดับการจองตัวแลดังกล่าวในภาษา C ลงในหน่วยความจำที่มีการบังคับ 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 Machine จงให้เหตุผลประกอบการอธิบาย

12. ในการเรียกใช้งานโปรแกรมย่อย จงเปรียบเทียบข้อดีข้อเสียระหว่างการจำค่าโดยผู้เรียนเอง และ การจำค่าโดยผู้ถูกเรียก กรณีใดน่าจะทำงานได้อย่างมีประสิทธิภาพมากกว่า จงอธิบายพร้อมยกตัวอย่างประกอบ
13. จากตัวอย่างโปรแกรมที่กำหนดให้ ให้ลองทำการ compile ด้วย “gcc -C -S test.c” แล้ว ลองศึกษาดูว่า โปรแกรมดังกล่าว มีการส่งผ่านค่า parameters และ การคืนค่าอย่างไร (หากเป็นไปได้ ให้ลองทำในระบบปฏิบัติการและหน่วยประมวลผลกลางที่ใช้สถาปัตยกรรมชุดคำสั่งแต่ละรุ่น กันไป 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 register (เช่น \$r13) ในการบันทึกค่า Return Address โปรแกรมย่อยจะต้องมีการจัดการอย่างไร เพื่อให้สามารถเรียกใช้งานโปรแกรมย่อยอื่นหรือทำ Recursive Call ได้อีก จงอธิบายพร้อมยกตัวอย่างประกอบ การอธิบาย
15. จงอธิบายข้อแตกต่างระหว่าง vector table และ handler table พร้อมยกตัวอย่างประกอบ การอธิบาย

4 การออกแบบหน่วยประมวลผล แบบ single cycle

ตามที่ได้ศึกษาถึงสถาปัตยกรรมชุดคำสั่งมาแล้ว เนื้อหาภายในบทนี้จะเป็นการสร้างหน่วยประมวลผลกลางแบบง่าย โดยการสร้างองค์ประกอบพื้นฐาน และนำองค์ประกอบต่าง ๆ มาประกอบเข้าด้วยกัน เป็นหน่วยประมวลผลกลาง โดยมุ่งประเด็นที่ การออกแบบทางเดินข้อมูล (Data path) และการสร้างสัญญาณเพื่อควบคุมองค์ประกอบภายใน เป็นหลัก

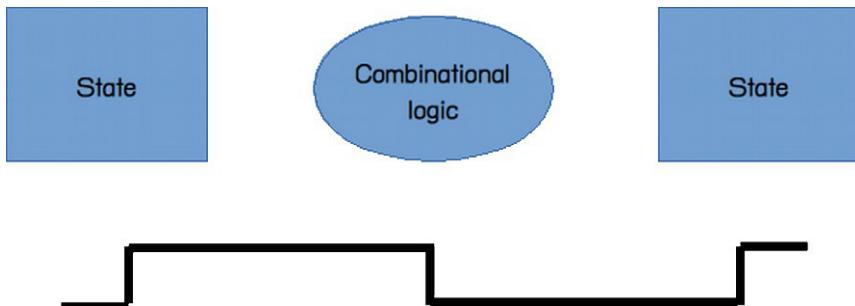
ภายในหน่วยประมวลผลกลาง จะประกอบขึ้นจากการจราจรฟ้าต่าง ซึ่งหากวิเคราะห์ลักษณะของวงจรที่เป็นส่วนประกอบภายในหน่วยประมวลผลกลางตามลักษณะของหน้าที่ จะสามารถจำแนกวงจรได้เป็น 2 กลุ่ม คือ

- **วงจรสำหรับการประมวลผลข้อมูล** หรือ Combinational Logic (วงจรเชิงผสม) ซึ่งเป็นวงจรที่ทำหน้าที่ถอดรหัส ตีความ หรือประมวลผล วงจรเหล่านี้สร้างขึ้นได้ด้วยความรู้ทางวงจรดิจิตอลแบบพื้นฐาน (เช่น K-MAP)
- **วงจรสำหรับการประมวลผลแบบลำดับ** หรือ Sequential Logic (วงจรเชิงลำดับ) ซึ่งเป็นวงจรที่ทำหน้าที่กำหนดการทำงานของส่วนต่างๆ ของระบบให้ทำงานตามขั้นตอนที่เหมาะสม

ในหน่วยประมวลผลกลาง สายสัญญาณที่ใช้ในการเชื่อมต่อระหว่างส่วนต่างๆ ของวงจรสำหรับประมวลผล จะถูกเรียกว่า **ทางเดินข้อมูล** (Data path) ส่วนวงจรที่ใช้ควบคุมการทำงานของทางเดินข้อมูล และกำหนดลำดับสัญญาณให้เป็นไปตามขั้นตอน (สัญญาณควบคุม) 0 จะเรียกว่า หน่วยควบคุม (Control unit)

เพื่อประกอบความเข้าใจ เนื้อหาในบทนี้จะเริ่มต้นจากการสร้างส่วนประมวลผลต่างๆ ภายในหน่วยประมวลผลกลางก่อน จากนั้นจะนำส่วนประกอบต่างๆ มาต่อ กันด้วยทางเดินข้อมูล (Data path) เพื่อให้ข้อมูลสามารถเดินทางได้ และสุดท้ายจึงจะเป็นการออกแบบสัญญาณควบคุม ผลที่ได้หลังจากจบบทนี้คือ หน่วยประมวลผลกลางแบบ 1 cycle (CPI เป็น 1) โดยทุกคำสั่งจะต้องรอการทำงานที่นานพอจะให้ทำงานเสร็จ

เพื่อเป็นการทบทวนความรู้เรื่อง Sequential logic ลองมาทำความเข้าใจ ความสัมพันธ์ระหว่างระบบสัญญาณนาฬิกา และ วงจร (เชิงผสมและเชิงลำดับ) อีกครั้งก่อนที่จะเริ่มเข้าสู่เนื้อหา ดังนี้ สัญญาณนาฬิกาเป็นสัญญาณสำคัญในการให้จังหวะการทำงาน (Synchronize) ของวงจร โดยทั่วไปสัญญาณขาขึ้นของนาฬิกา (positive-edge clock) จะเป็นสัญญาณให้จังหวะการเปลี่ยนสถานะของวงจรเชิงลำดับ (รูปที่ 4.1)



รูปที่ 4.1: ความสัมพันธ์ระหว่างสัญญาณนาฬิกา วงจรเข้าผสาน และ วงจรเชิงลำดับ

ในกรณีของหน่วยประมวลผลกลางแบบ 1 cycle ในบทนี้ หนึ่งคาบ (period) ของสัญญาณนาฬิกา คือเวลาในการประมวลผลหนึ่งคำสั่ง กล่าวคือ ทุกครั้งที่มีขาเข้าของสัญญาณนาฬิกาจะเป็นการเริ่มคำสั่งใหม่เสมอ นั่นหมายความว่า เวลาหนึ่งคาบจะต้องมากพอที่จะทำให้วงจรเข้าผสานที่อยู่ระหว่างกลางนั้นทำงานเสร็จได้ หากเวลาอันน้อยไป อาจทำให้การประมวลผลที่ได้เกิดความผิดพลาด

4.1 ขั้นตอนในการออกแบบหน่วยประมวลผลกลาง

เพื่อความสะดวกในการศึกษาและออกแบบ ในการออกแบบหน่วยประมวลผลกลาง สามารถแบ่งขั้นตอนการออกแบบออกได้เป็น 5 ขั้นตอนดังนี้

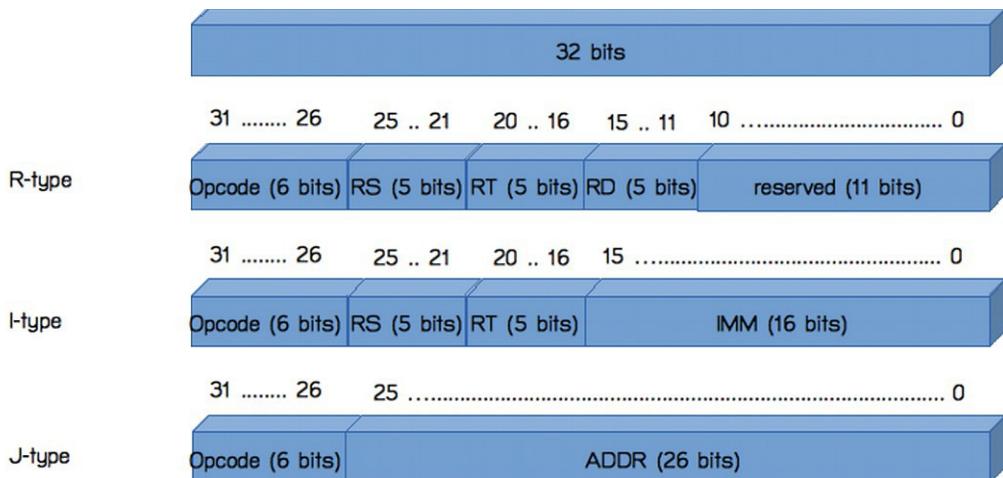
1. วิเคราะห์สถาปัตยกรรมชุดคำสั่ง เพื่อกำหนดส่วนประกอบที่เกี่ยวข้อง
2. ทำการออกแบบส่วนประกอบที่เกี่ยวข้องเพื่อประกอบเป็นทางเดินข้อมูล
3. ประกอบทางเดินข้อมูล
4. วิเคราะห์สัญญาณควบคุมที่เกี่ยวข้อง
5. ออกแบบวงจรควบคุมเพื่อกำหนดสัญญาณควบคุม

จากขั้นตอนดังกล่าว ขั้นต้นจะต้องทำการศึกษาและวิเคราะห์สถาปัตยกรรมชุดคำสั่งที่จะใช้สำหรับหน่วยประมวลผลกลางก่อน เพื่อความสะดวกในการศึกษา ในบทนี้จะยกตัวอย่างการออกแบบทั้งหมดโดยใช้บางส่วนของสถาปัตยกรรม LADA มาเพื่อประกอบการอธิบายดังนี้

4.1.1 สถาปัตยกรรมชุดคำสั่ง nanoLADA

กำหนดให้สถาปัตยกรรม nanoLADA ประกอบด้วยคำสั่งจำนวน 7 คำสั่ง และมีจำนวนรูปแบบคำสั่ง (Instruction format) 3 แบบ คือ R-type, I-type, และ J-type โดยมีรายละเอียดดังต่อไปนี้

รูปแบบคำสั่ง 3 รูปแบบ



ในการตีงคำสั่งที่เกี่ยวข้องการหน่วยความจำ กำหนดให้

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

คำสั่ง

ORI rt, rs, imm	<i>; (I-type)</i>	
ความหมาย	$R[rt] \leftarrow R[rs] \mid \text{zero_ext}(imm);$	$\text{PC} \leftarrow \text{PC} + 4$
ORUI rt, rs, imm	<i>; (I-type)</i>	
ความหมาย	$R[rt] \leftarrow R[rs] \mid \text{zero_pad}(imm);$	$\text{PC} \leftarrow \text{PC} + 4$
ADD rd, rs, rt	<i>; (R-type)</i>	
ความหมาย	$R[rd] \leftarrow R[rs] + R[rt];$	$\text{PC} \leftarrow \text{PC} + 4$
LW rt, rs, imm	<i>; (I-type)</i>	
ความหมาย	$R[rt] \leftarrow \text{MEM}[R[rs] + \text{sign_ext}(imm)];$	$\text{PC} \leftarrow \text{PC} + 4$
SW rt, rs, imm	<i>; (I-type)</i>	
ความหมาย	$\text{MEM}[R[rs] + \text{sign_ext}(imm)] \leftarrow R[rt];$	$\text{PC} \leftarrow \text{PC} + 4$
BEQ rs, rt, imm	<i>; (I-type)</i>	

```

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

JMP addr ; (J-type)

    ความหมาย PC ← (addr * 4)

```

ในส่วนต่อไป จะเป็นการวิเคราะห์สถาปัตยกรรมชุดคำสั่ง และ การออกแบบค์ประกอบที่เกี่ยวข้องแต่ละส่วน

4.2 องค์ประกอบภายในสำหรับสถาปัตยกรรม nanoLADA

เนื้อหาในส่วนที่ เทียบได้กับขั้นตอนที่สองของการออกแบบหน่วยประมวลผลกลาง จากสถาปัตยกรรมชุดคำสั่งที่กำหนด จะวิเคราะห์ได้ว่าสถาปัตยกรรมดังกล่าวจะต้องมีองค์ประกอบที่เกี่ยวข้องใด้แก่

- register file ขนาด 32 บิตจำนวน 32 ชุด¹⁸ สามารถอ่านได้ 2 ชุดพร้อมกัน (rs || ลงทะเบียนข้อมูลได้ 1 ชุด (rt หรือ rd)
- register พิเศษสำหรับเป็น PC
- ALU มีความสามารถในการ บวก และ or
- ADDER สำหรับประมวลผลค่า PC
- extender มีความสามารถในการทำ sign extender, zero extender และ zero padding

สำหรับ PC จัดเป็นองค์ประกอบพื้นฐานที่สามารถออกแบบได้ด้วย D-flipflop ส่วน ADDER, และ ALU สามารถสร้างได้ด้วยวงจร full adder และ logic gate และ multiplexer ที่; ไปซึ่งผู้มีความรู้ทางด้าน Digital Logic นำเข้าใจอยู่แล้ว จึงขอไม่กล่าวในที่นี้ (หากอ่านถึงตรงนี้แล้ว งง ไม่เข้าใจว่า D-flipflop หรือ full adder คืออะไร คงจะต้องแนะนำให้ไปลองเข้าเรียนในวิชา Digital Computer Logic ซึ่งเป็นวิชาพื้นฐานก่อนเรียนวิชาชนิดอื่นครับ)

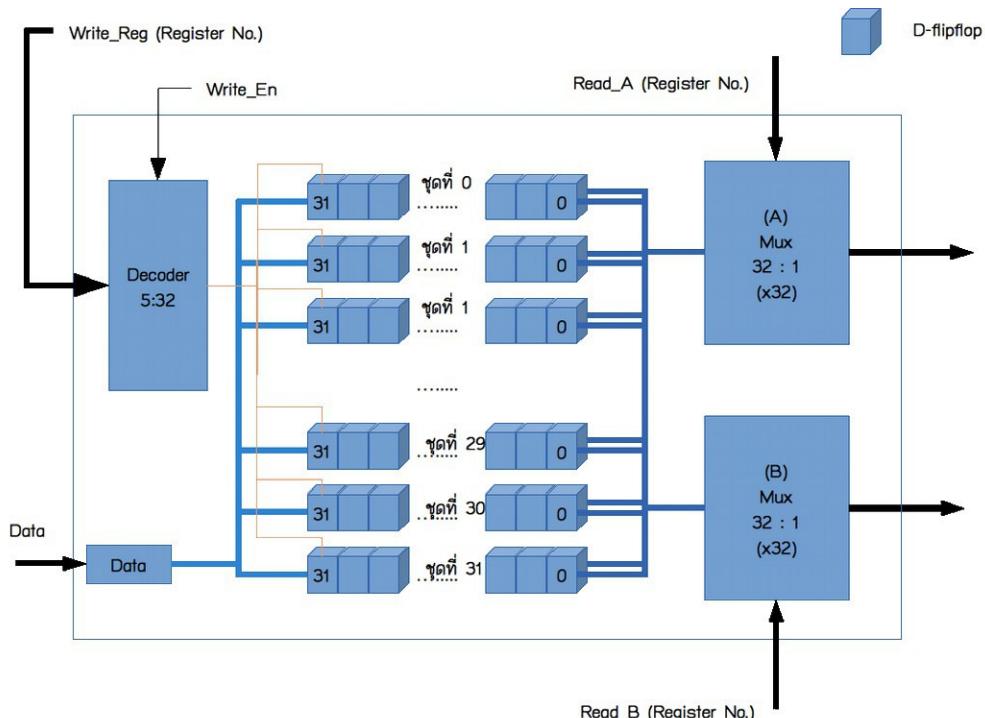
4.2.1 ชุด rejister (Register file)

ทั่วไป เมื่อกล่าวถึง rejister นักคอมพิวเตอร์มักจะนึกถึง D-flipflop ซึ่งมีความสามารถในการเก็บข้อมูล 1 บิต (เช่นกัน จะไม่ออกกล่าวถึงโครงสร้างภายในของ D-flipflop ในที่นี้) ดังนั้นหากต้องการให้มี rejister ขนาด 32 บิต สามารถทำได้ง่ายง่ายโดยการนำ D-flipflop จำนวน 32 ตัวมาต่อเนื่องกัน โดยใช้สัญญาณ enable, read/write ชุดเดียวกัน

¹⁸ เหตุที่มี 32 ชุดเนื่องจากชุดคำสั่งใช้ข้อมูล 5 บิตในการบอกว่าต้องการใช้ข้อมูลจาก register rs, rt ด้วย address ขนาด 5 บิต จึงทำให้สามารถเลือก register ได้ทั้งหมด 32 ชุด (0..31)

สิ่งที่ซับซ้อนขึ้นในที่นี้คือ สถาปัตยกรรมชุดคำสั่งนี้ ต้องการชุดของเรจิสเตอร์ที่ขนาด 32 บิต จำนวน 32 ชุด ที่สามารถอ่านได้ 2 ชุด และ เขียนได้ 1 ชุด พร้อมกัน การออกแบบในที่นี้จึงต้องใช้ ของ D-Flipflop จำนวน 32 ตัวมาต่อหน้ากันเพื่อให้ได้ 1 ชุด และทำลักษณะเดียวกันนี้อีก 31 ชุด (รวมกัน เป็น 32 ชุด) มาวางแผนต่อ กันโดยใช้วงจรเลือก (Multiplexor) แบบ 32 เลือก 1 จำนวน 32×2 ชุด ในการเลือกว่าต้องการอ่านข้อมูลชุดใด ดังแสดงในรูปที่ 4.2 ด้านขวา (ในความเป็นจริง จำนวนสาย สัญญาณจะซับซ้อนกว่านี้มาก แต่ที่นี่เพียงลักษณะสัญญาณที่เกี่ยวข้องเพื่อแสดงให้เห็นลักษณะโครงสร้าง ประกอบการอธิบายเท่านั้น)

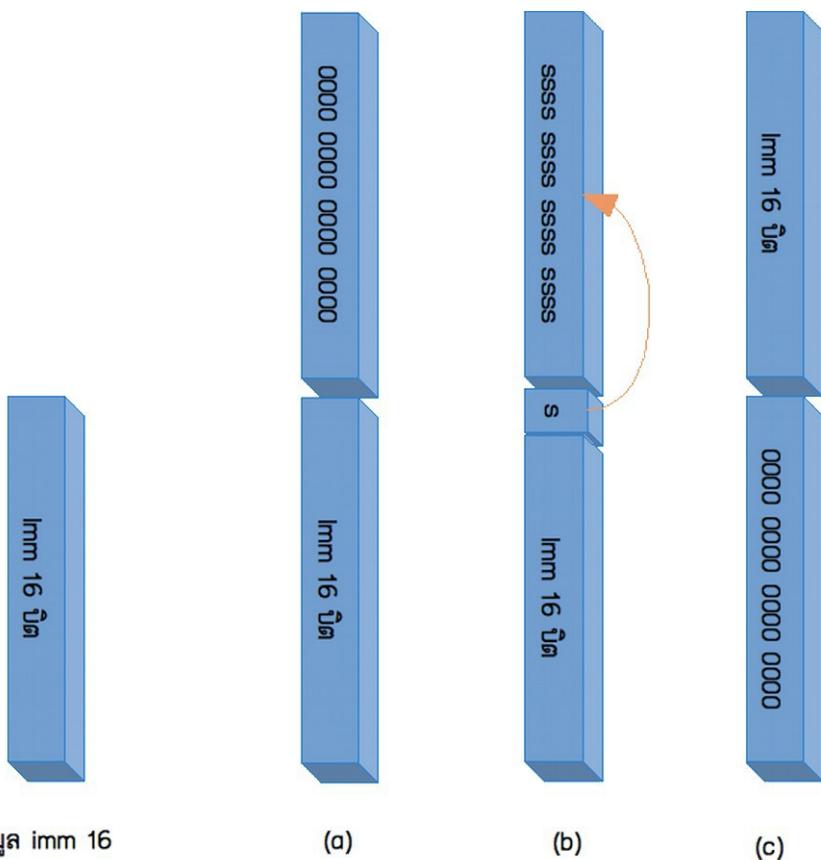
สำหรับกรณีการเขียนค่านั้น จะอาศัยสัญญาณนาฬิกา หรือสัญญาณ Write เป็นตัวกำหนดการ เขียนค่าให้กับเรจิสเตอร์ร่วมกับ Decoder เพื่อเลือกค่าเรจิสเตอร์ ที่ต้องการเขียน ซึ่งลักษณะการทำงานสามารถแสดงได้ดังรูปดังกล่าว เช่น เมื่อต้องการเขียนค่าเรจิสเตอร์ที่ 3 ด้วยข้อมูล 2521 สามารถทำได้โดยการป้อนค่า 3 ที่ Write_Reg และ ค่า 2521 ที่ Data จากนั้นจึงให้จังหวะสัญญาณ Write (ซึ่งสัญญาณเหล่านี้จะสร้างจากการจราบคุณที่จะกล่าวถึงต่อไป)



รูปที่ 4.2: register file ขนาด 32×32 บิต สามารถอ่านได้ 2 ชุด และเขียนได้ 1 ชุดพร้อมกัน จากรูปที่ 4.2 ด้านข้างเป็นการเขียนข้อมูล 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)

4.2.2 Extender

ในที่นี้ Extender จะต้องมีคุณสมบัติในการทำการ extend ได้ 3 แบบตามรูปที่ 4.3 กล่าวคือ (a) zero extender คือการเติม 0 ข้างหน้าจำนวน 16 บิต เพื่อแปลง immediate 16 บิตให้เป็นค่า 32 บิต (b) sign extender กรณีนี้ 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

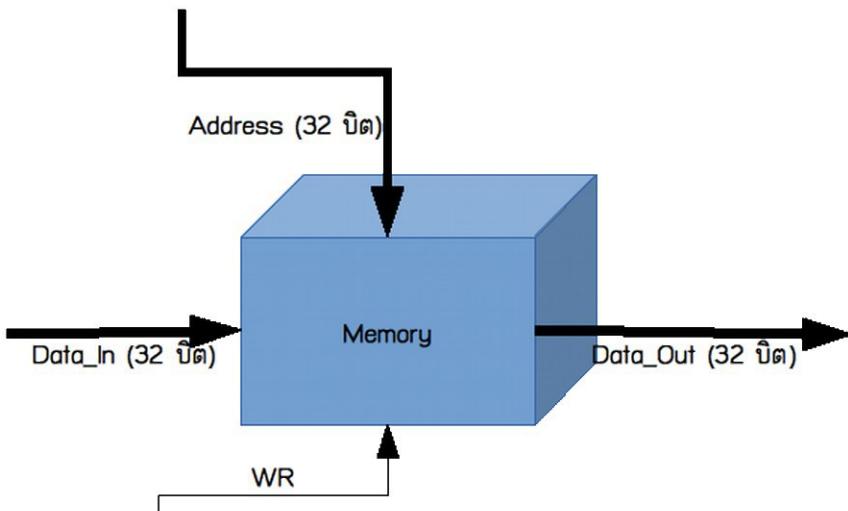
วิธีหนึ่งในการสร้าง extender แบบง่ายๆ โดยการใช้ multiplexor เลือกว่าต้องการผลลัพธ์ชุดใดในการแสดงผล หรืออาจจะใช้ multiplexor ชุดหนึ่งเลือกว่าจะทำ extender หรือ padding และใช้ multiplexor อีกชุดหนึ่งเลือกว่าจะ extend ด้วย zero หรือ sign ก็ได้ (วิธีการออกแบบ ขอให้เป็นแบบฝึกหัดของผู้อ่านในแบบฝึกหัดท้ายบท)

4.2.3 หน่วยความจำ

เพื่อความสะดวก ในบทนี้สมมุติให้หน่วยความจำมีการทำงานที่รวดเร็วไม่จำต้องรอข้อมูลเป็นเวลา

นาน ทั้งนี้ในความเป็นจริงหน่วยความจำที่มีขนาดใหญ่จะใช้เวลาในการเข้าถึงข้อมูลมาก ขนาดที่หน่วยความจำขนาดเล็กมักใช้เวลาในการเข้าถึงข้อมูลน้อย ซึ่งโครงสร้างและการจัดการกับหน่วยความจำ จำกล่าวถึงต่อไปในบทที่ 7

ในตอนนี้ ห้องหน่วยความจำเป็นกล่อง (memory chip) ที่เมื่อส่ง Address เข้าไป จะให้ค่าที่ต้องการออกมายัง Data_Out และ กรณีที่มีสัญญาณ WR จะทำการบันทึกค่าจาก Data_In เข้าไปยัง Address ที่กำหนด รายละเอียดแสดงได้ดังรูปที่ 4.4



รูปที่ 4.4: โครงสร้างหน่วยความจำ

เพื่อความสะดวกในการออกแบบ ในการออกแบบ จึงขอแยกหน่วยความจำออกเป็นสองชิ้นออกจากกัน คือ Program Memory (บางตำแหน่งจะเรียกว่า Instruction Memory) และ Data Memory ทั้งนี้เพื่อให้สามารถอ่านคำสั่งพร้อมกับอ่านและเขียนข้อมูลได้พร้อมกัน (ในทางปฏิบัติ จะเป็นเพียงการแยก CACHE หรือที่พักข้อมูลชั่วคราวเพื่อให้สามารถเข้าถึงข้อมูลจากหน่วยประมวลผลกลางได้พร้อมกันเท่านั้น ซึ่งจะกล่าวถึงต่อไปเมื่อถึงเนื้อหาในส่วนของหน่วยความจำ)

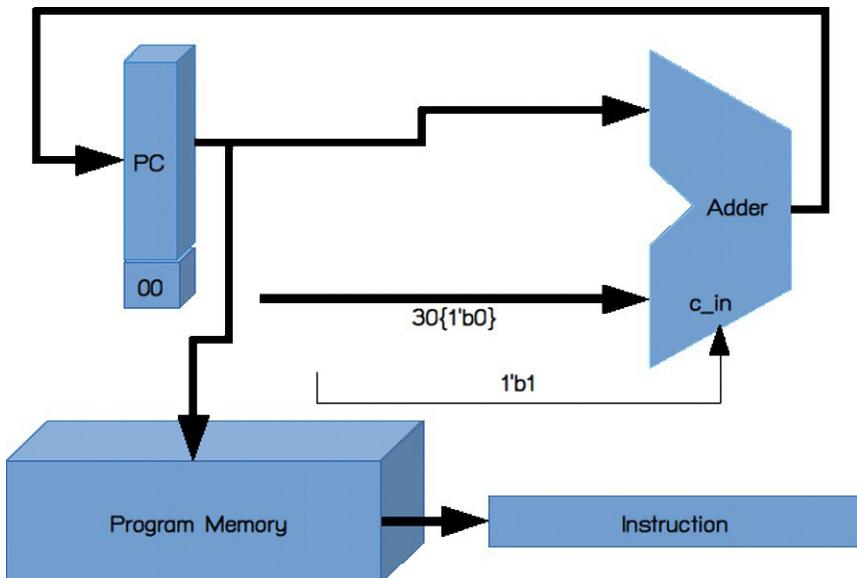
4.3 ทางเดินข้อมูลสำหรับสถาปัตยกรรมชุดคำสั่ง nanoLADA

เนื้อหาในส่วนที่ เทียบได้กับขั้นตอนที่สองของการออกแบบหน่วยประมวลผลกลาง ทางเดินข้อมูล (Data path) หรือ bus ภายใต้หน่วยประมวลผลกลาง เป็นสายสัญญาณ ที่ใช้ส่งผ่านข้อมูลระหว่างองค์ประกอบต่าง ๆ ภายใต้หน่วยประมวลผลกลาง โดยในการออกแบบนั้น สามารถทำได้โดยการนำองค์ประกอบต่าง ๆ มาต่อพ่วงกันในรูปแบบที่เหมาะสม และใช้ Multiplexor เป็นตัวเลือกทางเดินของข้อมูล กรณีที่มีข้อมูลต่างกันหลายทางเลือกที่เป็นไปได้มากกว่าหนึ่งทาง เช่น ค่า PC ซึ่งแสดงถึงคำสั่งต่อไปที่จะถูกดึงขึ้นมาทำงาน อาจจะอยู่ติดกับคำสั่งปัจจุบันคือ PC+4 หรืออาจจะเป็นค่าอื่น ๆ ที่เกิดจากการ Branch จึงใช้ Multiplexor สำหรับการเลือก

อย่างไรก็ตาม การที่จะเลือกว่าควรจะมี Data path เชื่อมต่อระหว่างจุดใดจุดหนึ่งถึงกันนั้น จะพิจารณาโดยอาศัยสถาปัตยกรรมชุดคำสั่งเป็นหลัก โดยพิจารณาจาก Addressing Mode และชุดคำสั่งว่า แต่ละคำสั่งนั้นจะต้องใช้งานค่าประกอบใดบ้าง และ ข้อมูลที่ต้องการใช้ในการประมวลคำสั่งนั้น ๆ ได้มาจากแหล่งใดบ้าง เช่น คำสั่งประมวลด้วย Operand ซึ่งมาจาก register หรือมาจากการคำสั่งโดยตรง ดังนั้น จึงต้องมีทางเดินข้อมูลเพื่อนำข้อมูลจากรีจิสเตอร์ไปป้อนเป็นผลลัพธ์ให้กับ ALU และ ทางเดินข้อมูลที่จะนำข้อมูลในกรณีที่ Operand อยู่ในคำสั่งไปป้อนให้กับ ALU ด้วยเช่นกัน

เพื่อประกอบความเข้าใจและช่วยให้ติดตามได้ง่าย จะยกขั้นตอนการประกอบทางเดินข้อมูลโดยจะนำคำสั่งเข้ามาใส่ให้ลงคำสั่ง โดยเริ่มจากคำสั่ง ADD, ORI, ORUI, LW, SW, BEQ และ JMP ตามลำดับ ทั้งนี้ให้เทียบกับสถาปัตยกรรมชุดคำสั่งในส่วนที่ 4.1.1 เพื่อประกอบความเข้าใจ

ก่อนอื่น ทุกคำสั่งจะมีโครงสร้างการทำงานที่เหมือนกันคือ การ fetch คำสั่ง โดยมี RTL ของ การ fetch คำสั่งคือ $IR \leftarrow MEM[PC]$ และมีขั้นตอนการทำงานตอนท้ายคำสั่งที่คล้ายคลึงกัน คือ $PC \leftarrow PC + 4$ จาก RTL นี้เราสามารถออกแบบทางเดินข้อมูลได้ดังรูปที่ 4.5



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

จากรูปเนื่องจากทุกคำสั่งมีขนาด 4 ไบต์ (32 บิต) ดังนี้ PC จะลงท้ายด้วย 2'b00 เสมอ เพื่อความสะดวก เราจึงใช้ Adder ขนาด 30 บิต และทำการบวก 1 แทนการบวก 4 (หากคิดเป็นเลขฐาน 10 จะได้ว่า $PC \leftarrow ((PC / 4) + 1) * 4$ ซึ่ง มีค่าเหมือนกับ $PC \leftarrow PC + 4$ นั้นเอง แต่เนื่องจากในเลขฐานสอง การคูณสี่คือการ shift left ไป 2 บิต หรือการใส่เลขศูนย์ต่อท้ายสองตัว การหารด้วยสี่หรือคูณด้วยสี่ในลักษณะนี้ จึงทำได้ง่ายมาก ส่วน 30{1'b0}¹⁹ นั้น จะถูกแทนที่ด้วยค่าอื่นต่อไปเมื่อ

19 (บิต 30{1'b0} หมายถึงเลข 0 จำนวน 30 บิต เขียนแบบภาษา VerilogHDL)

เป็นการทำคำสั่ง branch จากนี้ จะเป็นการนำคำสั่งแต่ละคำสั่งมาวิเคราะห์ และประกอบทางเดินข้อมูลให้สมบูรณ์

4.3.1 ทางเดินข้อมูลสำหรับคำสั่ง ORI และ ORUI

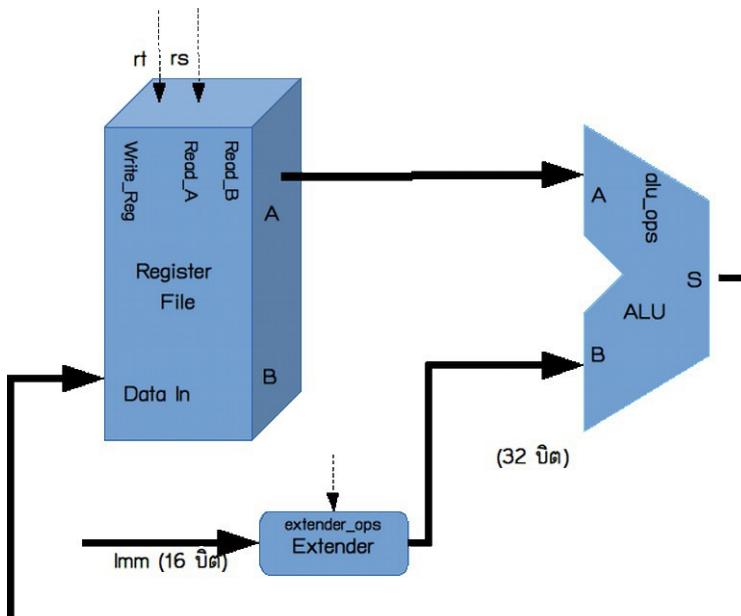
ORI rt, rs, imm ; (I-type)

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

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

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

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

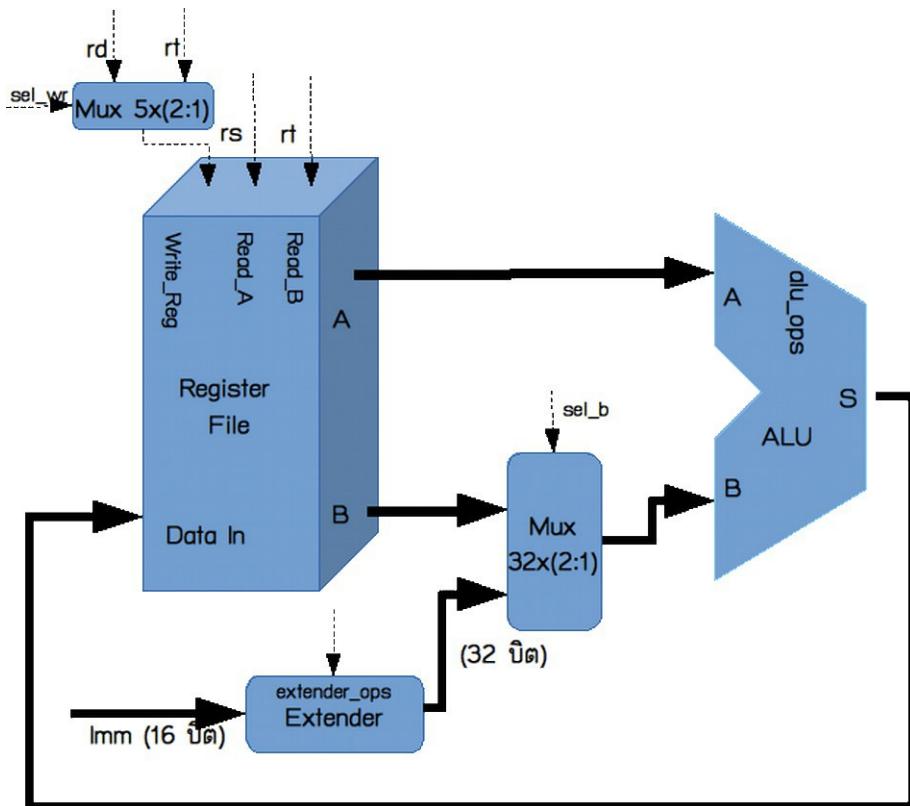
ในทำนองเดียวกันกับคำสั่ง ORI คำสั่ง ORUI มีการทำงานที่คล้ายคลึงกับคำสั่ง ORI มา ก ข้อแตกต่างเพียงแค่การกำหนดให้ Extender ทำการ zero padding แทนที่การทำ zero extending ดังนั้นทางเดินข้อมูลจึงไม่มีการเปลี่ยนแปลง (มีเพียงการเปลี่ยนแปลงของสัญญาณควบคุม extender_ops เท่านั้น)

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 ทำการบวกและนำผลลัพธ์ที่ได้ไปเก็บที่ $R[rd]$ เพื่อให้สามารถทำคำสั่ง ADD ได้ จะต้องมีการปรับทางเดินข้อมูลเพิ่มเติมให้ค่า $R[rt]$ มาป้อนเข้า ALU และให้สามารถเขียนค่าไปยัง $R[rd]$ ได้ ดังรูปที่ 4.7



รูปที่ 4.7: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง ADD

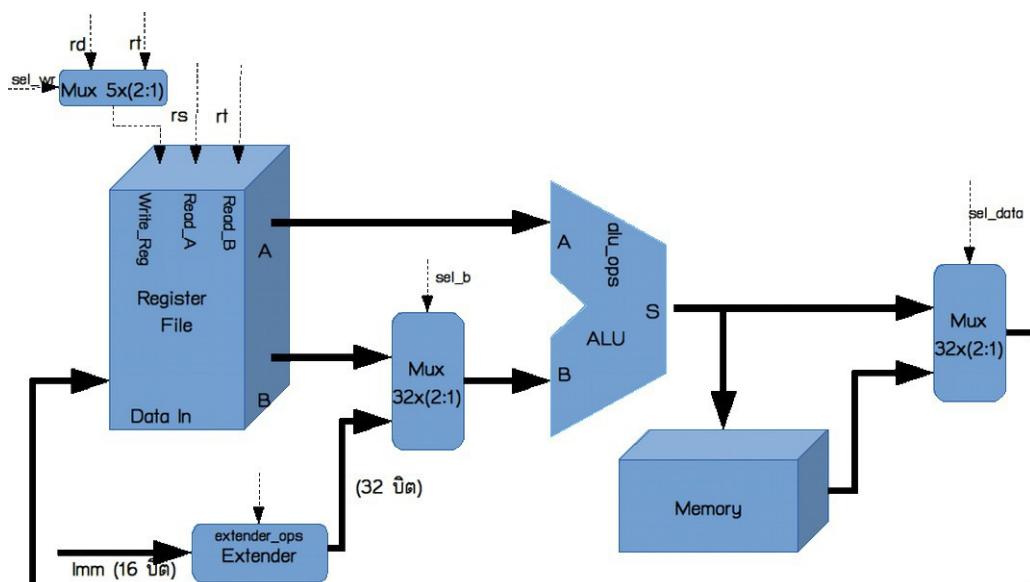
จะสังเกตุเห็นว่ามี Multiplexor 2 เลือก 1 เพิ่มขึ้นมา 2 ชุด โดยชุดแรก ใช้เพื่อเลือกข้อมูลสำหรับป้อนเข้าสู่ B ของ ALU (ควบคุมด้วยสัญญาณ sel_b) และ ชุดที่สอง ใช้สำหรับเลือก register ที่ต้องการเขียนค่า (ควบคุมด้วยสัญญาณ sel_wr)

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 $R[rt]$ มาบวกกับ immediate และนำค่าที่ได้ไปใช้เป็น address เพื่ออ่านค่าจากหน่วยความจำมาเก็บไว้ใน $R[rt]$ ลักษณะจะเห็นว่าทางเดินข้อมูลที่มีอยู่เดิม สามารถทำการเขียนค่าไปยัง $R[rt]$ ได้แล้ว (โดยการเลือก sel_wr ให้เป็นค่าที่หมายถึง) และนอกจากนี้ทางเดินข้อมูลดังกล่าวรยังสามารถประมวลผลค่า $R[rs] + \text{sign_ext}(imm)$ ได้แล้วอีกเช่นกัน เพื่อให้สามารถเขียนนำข้อมูลจากหน่วยความจำมาบันทึกที่ $R[rt]$ จึงทำได้ง่ายเพียง (1) เพิ่ม multiplexer มาใช้เลือกระหว่าง S และ ค่าที่อ่านได้จากหน่วยความจำมาป้อนเป็น Data_in และ(2) นำค่า S มาป้อนเป็น address ให้กับหน่วยความจำ ดังแสดงได้ในรูปที่ 4.8



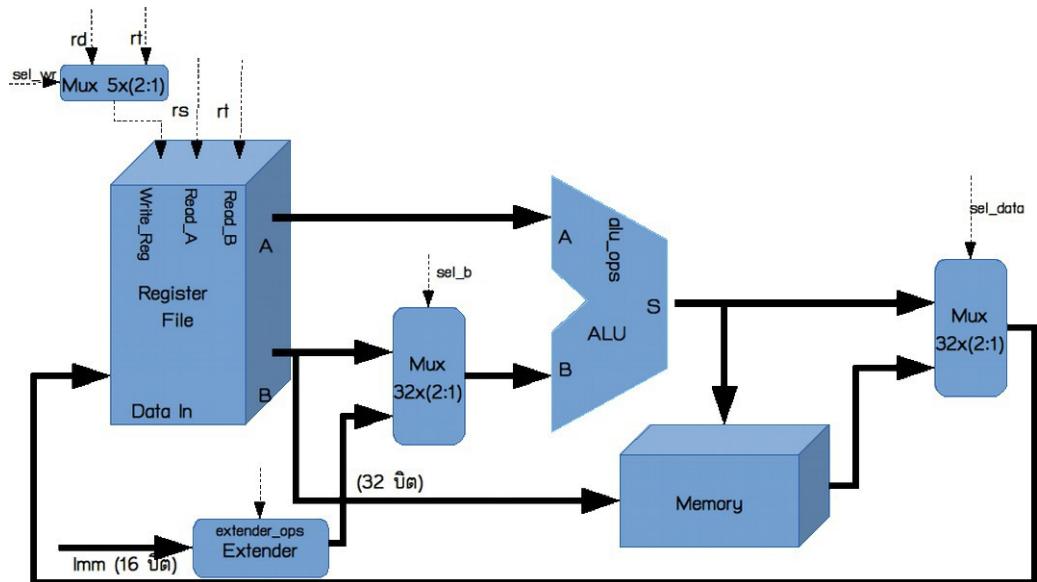
รูปที่ 4.8: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง LW

4.3.4 ทางเดินข้อมูลสำหรับคำสั่ง SW

$SW rt, rs, imm ; (I\text{-type})$

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

คำสั่ง SW มีการทำงานตรงข้ามกับคำสั่ง LW ดื้อเป็นการเขียนข้อมูลแทนที่จะเป็นการอ่านข้อมูล ส่วนการคำนวน address ยังคงมีลักษณะการทำงานเหมือนกัน อ้างอิงจากทางเดินข้อมูลที่มีอยู่เดิม สามารถปรับปรุงให้รองรับการทำงานของคำสั่ง SW ได้โดยการเพิ่มทางเดินจาก B ซึ่งเป็นผลลัพธ์ของ Register ไปยัง Data In ของหน่วยความจำ ดังแสดงได้ในรูปที่ 4.9



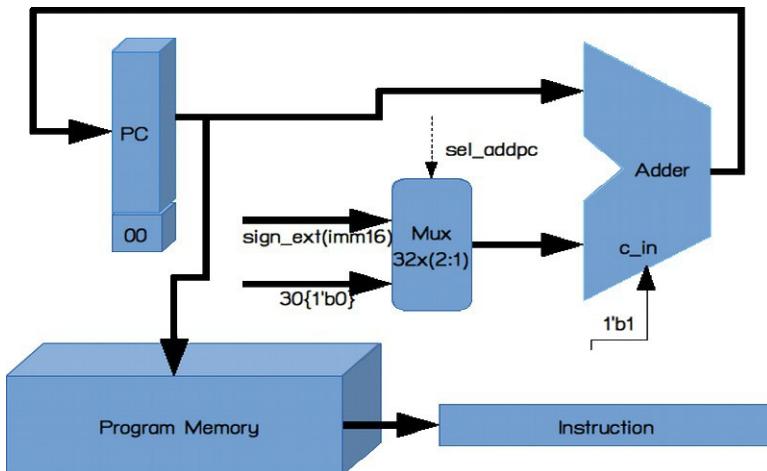
รูปที่ 4.9: ทางเดินข้อมูลเมื่อเพิ่มคำสั่ง SW

4.3.5 ทางเดินข้อมูลสำหรับคำสั่ง BEQ

BEQ rs, rt, imm ; (l-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] มีค่าเท่ากันกับ R[rt] จะทำการบวกค่า PC ด้วย sign_ext(imm) * 4 ในกรณีนี้ หาก ALU สามารถให้ค่า zero เป็น 1 ได้ เมื่อค่า A และ B ซึ่งเป็น Input ของ ALU มีค่าเท่ากัน เรายังสามารถจะนำค่านี้มาใช้ในการเลือกได้ว่า จะต้องทำการบวก PC + 4 ด้วยค่า sign_ext(imm) * 4 หรือไม่ จากทางเดินข้อมูลที่มีอยู่ สามารถเพิ่มเติมปรับปรุงให้รองรับการทำงานของคำสั่ง BEQ ได้ดัง รูปที่ 4.10 โดยในที่นี้ จะแสดงเพียงทางเดินข้อมูลส่วนที่เกี่ยวข้องกับ PC เท่านั้น เพื่อประกอบความเข้าใจ ควรพิจารณารูปที่ 4.5 เทียบกับ รูปที่ 4.10 จะเห็นว่าส่วนที่เพิ่มเติมขึ้นมาคือ Multiplexor (ควบคุมด้วย sel_addpc) สำหรับเลือกค่าที่จะบวกเข้ากับ PC



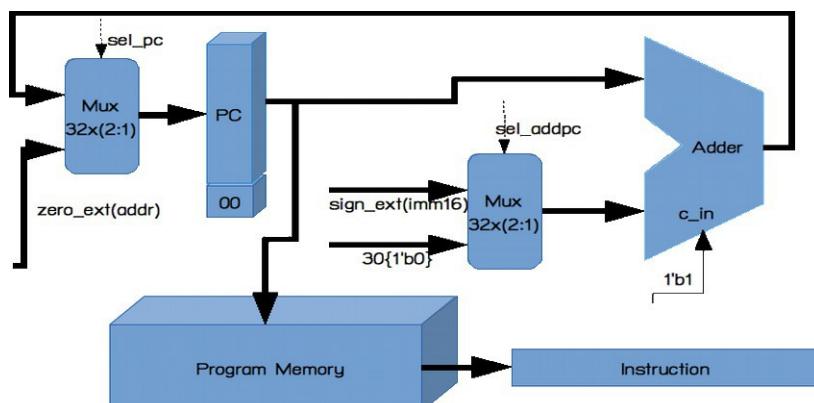
รูปที่ 4.10: ทางเดินข้อมูลส่วน PC เมื่อเพิ่มคำสั่ง BEQ

4.3.6 ทางเดินข้อมูลของคำสั่ง JMP

JMP addr ; (J-type)

ความหมาย $PC \leftarrow (addr * 4)$

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



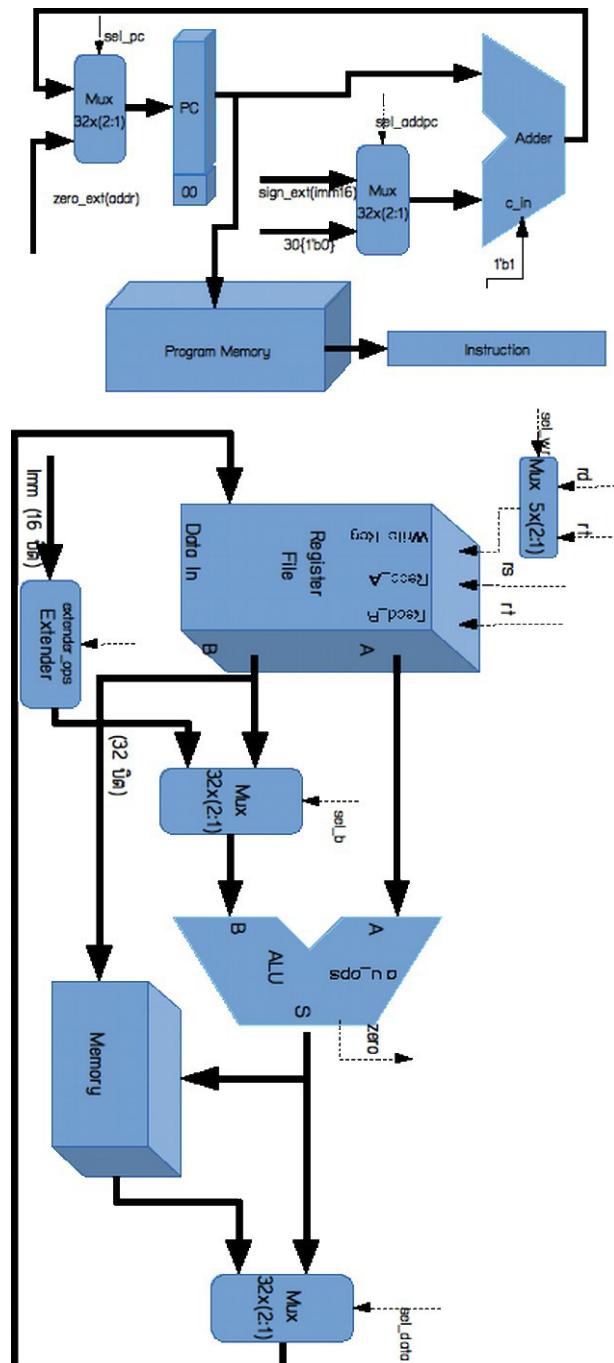
รูปที่ 4.11: ทางเดินข้อมูลส่วน PC เมื่อเพิ่มคำสั่ง JMP

4.3.7 สรุปทางเดินข้อมูลสำหรับสถาปัตยกรรมชุดคำสั่ง nanoLADA

กล่าวโดยสรุป ในการสร้างทางเดินข้อมูลทำได้โดยการนำส่วนประกอบต่างๆ มาวางและลากสายสัญญาณเพื่อเชื่อมระหว่างส่วนประกอบต่างๆ ให้สามารถประมวลผลข้อมูลได้ตามคุณสมบัติที่กำหนด ใน RTL ของแต่ละคำสั่ง หลังจากได้ทางเดินข้อมูลที่สมบูรณ์ (ประมวลผลได้ทุกคำสั่งในสถาปัตยกรรมชุดคำสั่งที่กำหนด) จะเหลือเพียงสัญญาณควบคุมต่างๆ ซึ่งจะต้องทำการสร้างวงจรควบคุม เพื่อควบคุมสัญญาณเหล่านี้ต่อไป ซึ่งทางเดินข้อมูลที่สมบูรณ์แสดงได้ดังรูปที่ 4.12

จากรูปทางเดินข้อมูลจะแสดงด้วยเส้นทึบ และสัญญาณควบคุมจะแสดงด้วยเส้นประ ในที่นี้สัญญาณควบคุมประกอบด้วย

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



รูปที่ 4.12: ทางเดินข้อมูลสำหรับสถาปัตยกรรม nanoLADA

นอกจากสัญญาณเหล่านี้ยัง ยังมีสัญญาณ reg_wr เพื่อบอกให้ register ทำการบันทึกค่า, สัญญาณ mem_wr เพื่อบอกให้หน่วยความจำทำการบันทึกค่า และสัญญาณ alu_ops ซึ่งยังไม่ได้กล่าวถึง โดยกำหนดให้

- **reg_wr** เป็นสัญญาณเพื่อบอกให้ register ทำการบันทึกค่าจาก Data_in ไปยังค่า register ที่กำหนดโดย Write_Reg กำหนดให้ 1 – แทน การบันทึกข้อมูลใหม่
- **mem_wr** เป็นสัญญาณเพื่อบอกให้หน่วยความจำ ทำการบันทึกค่าจาก Data_in ไปยังค่าหน่วยความจำ ที่กำหนดโดย address กำหนดให้ 1 – แทน การบันทึกข้อมูลใหม่
- **alu_ops** เป็นสัญญาณบอกให้ ALU ทำการ and, or, add เป็นต้น (ซึ่งจะกล่าวถึงรายละเอียดในส่วนถัดไป)

อย่างไรก็ตาม ในทางปฏิบัติ มีทางเดินข้อมูลที่เป็นไปได้มากกว่า 1 แบบที่จะให้หน่วยประมวลผลกลางทำงานตามสถาปัตยกรรมชุดคำสั่งที่ต้องการ ซึ่งในกรณีนี้ เน้นที่ความง่ายเป็นหลักเพื่อประกอบการศึกษาเท่านั้น

4.4 สัญญาณควบคุม (Control)

เนื้อหาในส่วนนี้เรียบได้กับขั้นตอนที่สี่ของการออกแบบหน่วยประมวลผลกลาง ดือ **การวิเคราะห์สัญญาณควบคุมที่เกี่ยวข้อง**

หากสังเกตที่ทางเดินข้อมูลที่มีการกำหนดไว้ตามแผนภาพดังกล่าวข้างต้นแล้ว เราจะพบว่ามี Multiplexor สำหรับเป็นตัวเลือกทางเดินข้อมูลอยู่สามสาย而已 แต่ สัญญาณที่จะใช้ป้อนให้กับ Multiplexor เพื่อเลือกทางเดินข้อมูลนี้ เรียกว่า สัญญาณควบคุม ซึ่งจะแตกต่างกันไปตามแต่ละคำสั่ง ดังนั้นที่มาของสัญญาณควบคุม ดือ **การตีความคำสั่งที่อ่านขึ้นมาจาก Program memory** เพื่อใช้เลือกทางเดินข้อมูลสำหรับการทำคำสั่งนั้น

เนื่องจากสถาปัตยกรรมชุดคำสั่ง nanoLADA ที่กำหนดในตอนที่ 4.1.1 มีได้กำหนดค่า opcode มาให้ในที่นี้จึงขอกำหนดค่า opcode ให้กับคำสั่งต่างๆ ดังนี้

คำสั่ง	Opcode
ORI rt, rs, imm	010000
	

คำสั่ง	Opcode
ORUI rt, rs, imm	010001
	
ADD rd, rs, rt	000001
	
LW rt, rs, imm	011000
	
SW rt, rs, imm	011100
	
BEQ rs, rt, imm	100100
	
JMP addr	110000
	

จากค่า opcode ที่กำหนด เราสามารถสร้างตารางค่าความจริง (Truth table) เพื่อกำหนดค่า สัญญาณควบคุมที่เกี่ยวข้องได้โดยพิจารณา opcode (และสัญญาณ zero จาก ALU) เป็น input และ พิจารณาสัญญาณควบคุม เป็น output ของวงจรสร้างสัญญาณควบคุม จะได้ผลตั้งตารางที่ 4.1

ตารางที่ 4.1: 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
ORI 01_0000	x	0	0	1	1	0	1	0
ORUI 01_0001	x	0	0	1	1	0	1	0
ADD 00_0001	x	0	0	0	0	0	1	0
LW 01_1000	x	0	0	1	1	1	1	0
SW 01_1100	x	0	0	x	1	x	0	1
BEQ 10_0100	0 1	0 0	0 1	x	x	x	0	0
JMP 11_0000	x	1	x	x	x	x	0	0

นอกจากนี้ยังมีสัญญาณ alu_ops ซึ่งใช้กำหนดการทำงานของ ALU ให้เป็นไปตามที่ต้องการ เพื่อประกอบการอธิบาย สมมุติให้การทำงานของ ALU เป็นดังตารางที่ 4.2

ตารางที่ 4.2: 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	$S \leftarrow A - B ; z \leftarrow (S==0)$

คราวนี้ ลองสร้างตารางค่าความจริงอีกรังเพื่อกำหนด output ของ alu_ops จะได้ผลเป็นตารางที่ 4.3 ข้อสังเกตุจากตารางนี้ว่า 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 000 และหาก opcode ขึ้นต้นด้วย 100 ค่า alu_ops จะเป็น 10 เป็นต้น ซึ่งหากผู้เรียนมีความเชี่ยวชาญ Digital Computer Logic ก็สามารถที่จะใช้ Karnaugh Map มาทำการสร้างวงจรดังกล่าวได้อย่างรวดเร็ว

ตารางที่ 4.3: สัญญาณ alu_ops สำหรับสถาปัตยกรรม nanoLADA

คำสั่ง Opcode	alu_ops
ORI 01_0000	01
ORUI 01_0001	01
ADD 00_0001	00
LW 01_1000	00
SW 01_1100	00
BEQ 10_0100	10
JMP 11_0000	x

ลองย้อนกลับไปวิเคราะห์ตารางที่ 4.1 อีกครั้งหนึ่ง หากต้องสร้างวงจรเพื่อสร้างสัญญาณควบคุมขึ้นมาจากการนี้ มีแนวทางที่สามารถดำเนินการได้หลายวิธี เช่น การหาผลรวมของผลคูณ (sum of product) รวมถึงการใช้การลดรูปด้วย Karnaugh Map และ Quine–McCluskey algorithm หรือการใช้ PLA หรือ ROM ช่วยในการ decode ซึ่งรายละเอียดดังกล่าวไม่อธิบายในขอบข่ายของหนังสือเล่มนี้ แต่เพื่อประกอบความเข้าใจ จะขอยกตัวอย่างการสร้างสัญญาณควบคุมโดยการทำ sum of product แบบไม่มีการลดรูปดังนี้

กำหนดให้ $op[5..0]$ แทนแต่ละบิตของ Opcode ก่อนอื่นให้ทำการสร้างผลคูณ²⁰ (product) โดยให้

$$G1 \text{ แทนคำสั่ง ORI (01_0000) คือ } G1 = \bar{op}_5 \circ op_4 \circ \bar{op}_3 \circ op_2 \circ \bar{op}_1 \circ \bar{op}_0$$

$$G2 \text{ แทนคำสั่ง ORUI (01_0001) คือ } G2 = \bar{op}_5 \circ op_4 \circ \bar{op}_3 \circ op_2 \circ \bar{op}_1 \circ op_0$$

$$G3 \text{ แทนคำสั่ง ADD (00_0001) คือ } G3 = \bar{op}_5 \circ \bar{op}_4 \circ \bar{op}_3 \circ op_2 \circ \bar{op}_1 \circ op_0$$

$$G4 \text{ แทนคำสั่ง LW (01_1000) คือ } G4 = \bar{op}_5 \circ op_4 \circ op_3 \circ \bar{op}_2 \circ \bar{op}_1 \circ \bar{op}_0$$

$$G5 \text{ แทนคำสั่ง SW (01_1100) คือ } G5 = \bar{op}_5 \circ op_4 \circ op_3 \circ op_2 \circ \bar{op}_1 \circ \bar{op}_0$$

$$G6 \text{ แทนคำสั่ง BEQ (10_0100) คือ } G6 = op_5 \circ \bar{op}_4 \circ \bar{op}_3 \circ op_2 \circ \bar{op}_1 \circ \bar{op}_0$$

²⁰ สำหรับผู้เรียนที่ไม่แน่หรือจำไม่ได้ ผลคูณคือการนำค่าต่างๆ มา AND กัน

G7 แทนคำสั่ง JMP (11_0000) คือ $G7 = op_5 \circ op_4 \circ o\bar{p}_3 \circ o\bar{p}_2 \circ o\bar{p}_1 \circ o\bar{p}_0$

จากค่าผลคูณ G1 .. G7 ที่กำหนด สามารถสร้างสัญญาณควบคุมต่างๆ โดยนำผลคูณที่เกี่ยวข้องมาหาผลบวก²¹ กัน ได้ผลดังนี้

$$sel_{pc} = G7$$

$$sel_{addpc} = G6 + zero$$

$$sel_{wr} = G1 + G2 + G4$$

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

$$sel_{data} = G4$$

$$reg_{wr} = G1 + G2 + G3 + G4$$

$$mem_{wr} = G5$$

หากจะวิเคราะห์ต่อไปอีกนิด จะเห็นว่า sel_wr และ sel_b สามารถใช้สัญญาณ sel_b แทนได้เลย ทั้งนี้เนื่องจากสัญญาณ sel_wr ในตารางมีค่า x (don't care) ในช่องของ G5 การใช้ sel_b แทนจึงมีได้ทำให้ค่าที่ได้ผิดไปแต่อย่างไร

ในปัจจุบัน ทางเลือกที่ดีกว่าคือ การเขียนบรรยายการทำงานด้วย VerilogHDL จากนั้นให้ compiler สร้างจริงจาก code ที่กำหนดแทน

4.5 Microprogram

อีกทางเลือกหนึ่งที่บริษัทผู้ผลิตหน่วยประมวลผลกลางเลือกใช้กันคือ **microprogram** แนวคิด คือ ให้การออกแบบส่วนควบคุมของหน่วยประมวลผลกลางมีลักษณะคล้ายกับการเขียนโปรแกรม ซึ่งช่วยอำนวยความสะดวกในการแก้ไขปัญหาและตรวจสอบข้อผิดพลาดระหว่างการออกแบบ โดยการใช้ **microinstruction** ทำการโหลดลงสู่หน่วยประมวลผลกลาง ซึ่งนอกจากจะอำนวยความสะดวกในการออกแบบแล้ว ยังช่วยให้สามารถปรับปรุงหรือจำลองหน่วยประมวลผลกลางให้ทำงานแทนหน่วยประมวลผลกลางที่มีสถาปัตยกรรมแบบอื่นๆ ได้โดยง่ายอีกด้วย ทั้งนี้หากผู้อ่านมีความสนใจในการออกแบบหน่วยประมวลผลกลางโดยการใช้ microprogram สามารถศึกษาเพิ่มเติมได้จากเอกสารของผู้ผลิตต่างๆ ได้ทั่วไป โดยรายละเอียดและการทำงานของ microprogram จะไม่ออกล่าในที่นี่

4.6 Critical Path และความเร็วสัญญาณนาฬิกา

เมื่อเราได้ทางเดินข้อมูลและนำสัญญาณควบคุมมาต่อพ่วงรวมกันแล้ว ถัดไปตอนของการวิเคราะห์

²¹ ในทำนองเดียวกัน ผลบวกคือการนำค่าต่างๆ มา OR กัน

critical path และการสร้างสัญญาณนาฬิกาในกับหน่วยประมวลผล ในกรณีนี้สัญญาณนาฬิกาจะต้องให้จังหวะในการทำคำสั่งหนึ่งคำสั่งให้เสร็จ ทั้งนี้ข้อควรระวังคือ **ทุกครั้งที่มีสัญญาณนาฬิกามา จะมีการเริ่มคำสั่งใหม่** ดังนั้น เวลาที่เว้นเพื่อให้วางจริงประสมได้ทำงานจำนวนมากเพียงพอเพื่อให้ทุกอย่างทำงานเสร็จสิ้นไม่ว่าจะเป็นคำสั่งเด็กตาม มีชั้นนั้นค่าที่ควรจะเขียนกลับคืนไปยัง register อาจจะผิดพลาดได้ การคำนวณเวลาที่ต้องรอการทำงานของแต่ละองค์ประกอบ จะอาศัยความรู้เรื่อง gate delay ซึ่งเป็นความรู้พื้นฐานเรื่อง digital computer logic ที่ผู้เรียนควรทราบมาก่อนเข่นกัน

เพื่อประกอบความเข้าใจ ลองทำการวิเคราะห์ดังนี้ จากทางเดินข้อมูลในรูปที่ 4.12 จะเห็นว่า มีเวลาที่ต้องรอให้ส่วนต่างๆ ของระบบทำงานเพื่อให้ได้ข้อมูลที่ต้องการดังนี้

1. ตอนเริ่มต้น $\text{Instruction} \leftarrow \text{MEM[PC]}$ จะต้องรอเวลาที่ได้ค่า PC ที่ต้องการ และรอหน่วยความจำ Program Memory อ่านค่า Instruction ที่ต้องการ (สมมุติให้เวลาที่ PC ทำงานเป็น 10ns และเวลาที่หน่วยความจำทำงานเป็น 20ns ตามลำดับ)
2. เมื่อได้ค่าสั่งแล้ว จะต้องรอเวลาที่วงจรควบคุมทำการสร้างสัญญาณควบคุมที่เกี่ยวข้อง (สมมุติให้การทำงานของวงจรควบคุมเป็น 40ns)
3. รอค่า register file เพื่อนำค่า register ที่กำหนดโดย $R[rs]$ ॥ ลชหรือ $R[rt]$ มาแสดงผลที่ทางออก A ॥ ลชB ของ register file ในกรณีที่เป็นแบบ I-type ก็จะต้องรอค่าที่ imm ผ่าน extender เพื่อให้ได้ค่าที่ต้องการด้วย อย่างไรก็ตามเนื่องจาก extender และ register file มีการทำงานที่นานกันได้ ดังนั้น เพียงรอเวลาที่มากที่สุดของข้อมูลชุดใดชุดหนึ่งเท่านั้น (สมมุติให้ register ทำงานที่ 20ns ॥ ลชextender ทำงานที่ 15 ns) ซึ่งในกรณีนี้ max (20ns ,15ns) = 20ns
4. หลังจากได้ค่าที่ต้องการแล้ว ต้องให้ multiplexor ที่ควบคุมด้วยสัญญาณ sel_b ทำการเลือกค่าที่ต้องการ
5. กรณีที่มีการคำนวณค่าโดยใช้ ALU จะต้องรอให้ ALU ทำการประมวลผลจนได้ผลลัพธ์ของมาที่ S (สมมุติให้ ALU ทำงานโดยใช้เวลา 60ns)
6. ในกรณีที่เป็นการอ่านข้อมูลจาก Data Memory จะต้องรอให้หน่วยความจำทำงานเพื่อนำค่าข้อมูลที่ต้องการมาแสดงผล (สมมุติให้ ALU ทำงานโดยใช้เวลา 20ns)
7. หลังจากได้ค่าที่ต้องการแล้ว ต้องรอให้ multiplexor ที่ควบคุมด้วยสัญญาณ sel_data ทำการเลือกค่าที่ถูกต้องเพื่อไปเขียนคืนยัง data_in

นอกจากนี้ยังมีเวลาที่ต้องรอ adder ทำการประมวลผล $\text{PC} \leftarrow \text{PC} + 4$ หรือกรณีอื่น ซึ่งไม่ได้กล่าวถึงในที่นี้ เพราะการทำงานอื่นๆ สามารถทำงานนี้ไปพร้อมกับการประมวลผลขั้นตอนหลักที่กล่าวข้างต้นได้ และมักใช้เวลาน้อยกว่าการทำงานข้างต้น

เพื่อความสะดวกสมมุติให้ multiplexor ทุกชุดทำงานที่ 10ns จะพบว่าหน่วยประมวลผลกลางดัง

กล่าวจะใช้เวลาในการทำงาน 1 คำสั่งมากที่สุด คิดเป็น (1) 10 + 20 + (2) 40 + (3) 20 + (4) 10 + (5) 60 + (6) 20 + (7) 10 คิดเป็นเวลา 190ns

$$\text{clock rate} = \frac{1}{10+20+40+20+10+60+20+10*10^{-9}} = \frac{10^9}{190} = 5.26 \times 10^6 \text{ Hz}$$

หรือสรุปได้ว่า หน่วยประมวลผลกลางดังกล่าวจะทำงานได้ที่สัญญาณนาฬิกา 5.26 Mhz

อย่างไรก็ตามหากวิเคราะห์การทำงานของหน่วยประมวลผลกลางในรูปข้างต้น เราจะพบว่าการประมวลผลแต่ละคำสั่งจะต้องใช้เวลาในการทำงานแท้กันคือ 190ns ทั้งที่ในบางคำสั่งนั้นสามารถทำเสร็จได้ภายในเวลาเพียง 170ns เท่านั้น (เช่น กรณีที่ไม่ต้องอ่านข้อมูลจากหน่วยความจำ) จึงได้มีความพยายามเพื่อลดเวลาในการทำงานดังกล่าวให้รอเท่าที่จำเป็น จากโครงการที่แสดงนี้ยังพบว่ามีองค์ประกอบหลายอย่างที่ทำงานช้ากัน เช่น ALU และ Adder นั้น น่าจะสามารถทำงานแทนกันได้ หรือ Instruction memory และ Data memory น่าจะสามารถใช้องค์ประกอบชิ้นเดียวกันได้ ทั้งนี้เพื่อให้หน่วยประมวลผลกลางที่ไม่มีขนาดเล็กลง จึงเกิดแนวทางในการออกแบบหน่วยประมวลผลกลางในรูปแบบอื่น ๆ อีกมากมาย เพื่อที่จะให้หน่วยประมวลผลกลางมีขนาดเล็กลง หรือ มีความเร็วในการประมวลผลมากขึ้น ซึ่งรายละเอียดและแนวทางจะกล่าวถึงในบทถัดไป

4.7 แบบฝึกหัดท้ายบท

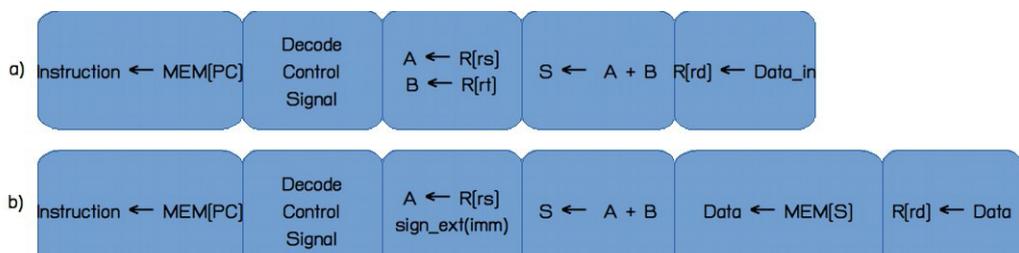
1. จงอธิบายส่วนประกอบต่าง ๆ ภายในหน่วยประมวลผลกลาง
2. จงอธิบายขั้นตอนในการประมวลผลคำสั่งหนึ่งคำสั่งของหน่วยประมวลผลกลาง
3. จงแสดงการสร้าง extender ที่สามารถทำงานได้ตามรูปที่ 4.3
4. จงออกแบบ ALU ขนาด 32 บิตที่มีการทำงานตามตารางที่ 4.2
5. Data path คืออะไร และ Control คืออะไร มีความเหมือนหรือมีความสัมพันธ์กันหรือไม่ อย่างไร
6. จากตารางที่ 4.1 จงแสดงการสร้างสัญญาณควบคุมด้วย PLA
7. จากโครงสร้างภายในของ register file ในรูปที่ 4.2 หากกำหนดให้ d-Flipflop มีค่า delay เป็น 20ns และ multiplexor และ decoder มีค่า delay เป็น 30ns แล้ว โครงสร้างของ register file ดังกล่าวควรจะมีความเร็วในการทำงานเป็นเท่าใด
8. จากตารางที่ 4.2 จงแสดงการสร้างสัญญาณ alu_ops ด้วยวิธีการ sum of products
9. จากตัวอย่างทางเดินข้อมูลสำหรับสถาปัตยกรรม nanoLADA ในรูปที่ 4.12 หากต้องการ
 เ พิ ม คำ ส ง
 SUB rd, rs, rt ;
 ค ว า ม ห ม า ย R[rd] ← R[rs] - R[rt];
 ทางเดินข้อมูลจะต้องมีการเปลี่ยนแปลงหรือไม่ อย่างไร
10. จากข้อ 9 หากกำหนด opcode ของ SUB เป็น 00_0010 จงสร้างตารางแสดงสัญญาณควบคุมที่เกี่ยวข้อง
11. จากหน่วยประมวลผลกลางแบบ single cycle ที่ได้ในบทที่ 4 หากโปรแกรมทดสอบบีบ จำนวนคำสั่งทั้งสิ้น 2 ล้านคำสั่ง หน่วยประมวลผลจะใช้เวลา CPU TIME คิดเป็นเท่าใด
12. การออกแบบหน่วยประมวลผลกลางโดยการใช้ microprogram มีข้อดีข้อเสียหรือไม่ อย่างไร?

5 หน่วยประมวลผลกลางแบบ Multiple Cycle

ลักษณะการออกแบบหน่วยประมวลผลกลางแบบ Multiple Cycle นั้น ช่วยให้เราสามารถพัฒนาหน่วยประมวลผลกลางที่ทำงานได้เร็วขึ้น และในบางกรณีอาจจะมีการแบ่งปันทรัพยากรณ์บางส่วนได้ด้วย สาเหตุที่อาจจะทำงานได้เร็วขึ้นเนื่องจาก **หน่วยประมวลผลกลาง จะไม่จำเป็นต้องรอการทำงานที่ยาวนานที่สุดในทุกรณ์** ส่วนสาเหตุที่อาจแบ่งทรัพยากรณ์ใช้ร่วมกันได้เนื่องจากสามารถใช้ช่องค์ประกอบบางอย่างร่วมกันได้โดยอาศัยช่วงเหลือมของเวลา เช่น หากเบรี่ยนเพียง ALU และ Adder กับเครื่องคิดเลข เมื่อมีคน 2 คน ต้องการใช้เครื่องคิดเลขสำหรับการประมวลผลโดยคนหนึ่ง ต้องการใช้เครื่องคิดเลขสำหรับการบวก และ อีกคนหนึ่งใช้เครื่องคิดเลขสำหรับการคำนวณทั่วไป หากทั้ง 2 คนไม่มีความจำเป็นต้องใช้เครื่องคิดเลขพร้อมกัน จะพบว่าเครื่องคิดเลขเพียงหนึ่งเครื่องก็เพียงพอสำหรับการใช้งานของคน 2 คน โดยการผลัดกันส่งเครื่องคิดเลขไปมาระหว่างคน 2 คน ในทำนองเดียวกัน กรณีของหน่วยประมวลผลกลาง อาจออกแบบให้ใช้ ALU และ Adder สำหรับการประมวลผลค่า PC ได้

นอกจากนี้หากแบ่งการประมวลผลคำสั่งหนึ่ง ออกเป็นช่วงย่อย จะทำให้สามารถข้ามขั้นตอนบางขั้นที่ไม่จำเป็นในการทำงานของคำสั่งนั้นได้ ด้วยเช่น รูปที่ 5.1 (a) เป็นเวลาที่แท้จริงของการทำงานของคำสั่ง ADD ส่วน (b) เป็นเวลาการทำงานของคำสั่ง LW ดังนั้น หากแบ่งการทำงานเป็นส่วนย่อยได้ จะช่วยให้หลายคำสั่งสามารถข้ามขั้นตอนการทำงานของการอ่านข้อมูลจากหน่วยความจำได้

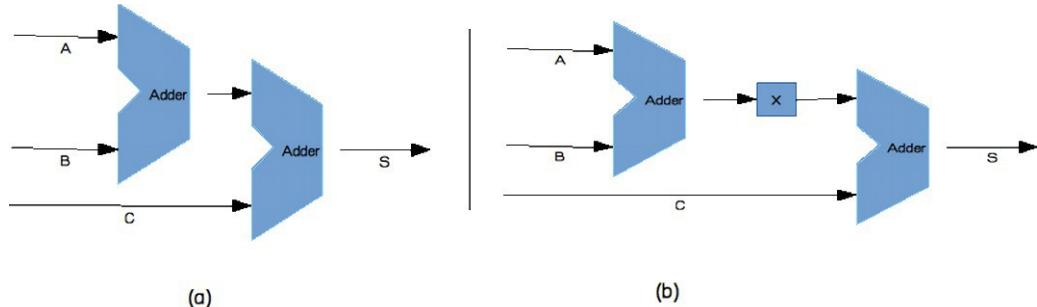
ยิ่งไปกว่านั้นการออกแบบวงจรเพื่อทำงานในช่วงปัญหาที่แ McCart ยังสามารถแก้ไขและตรวจสอบได้ง่ายกว่าการออกแบบวงจรขนาดใหญ่อีกด้วย



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

จากแนวทางในการออกแบบโดยใช้หน่วยประมวลผลบางส่วนร่วมกัน และการออกแบบเพื่อให้การประมวลผลคำสั่งหนึ่งคำสั่งทำงานโดยใช้ช่วง Cycle ของสัญญาณนาฬิกา จึงจำเป็นต้องมีการเพิ่ม register ภายในบางดัว เพื่อใช้ในการส่งผ่านค่าระหว่าง Cycle เพื่อประกอบความเข้าใจ ลองพิจารณากรณีการประมวลผล $S \leftarrow A + B + C$ โดยใช้ adder 2 ชุด หากต้องการแบ่งการ

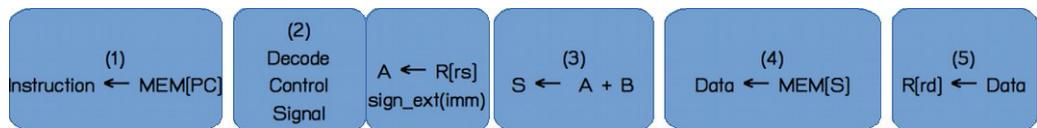
ทำงานดังกล่าวออกเป็น 2 ช่วง สามารถทำได้โดยการเพิ่มตัวแปรเพื่อใช้จำค่าชั่วคราวก่อน เช่น ในจังกกล่าวแรก ให้ทำการประมวลผล $X \leftarrow A + B$ ทำงานก่อน จากนั้นจึงจะทำการประมวลผล $S \leftarrow X + C$



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

หลักของการแบ่งขั้นตอนการทำงานที่ดีคือ การแบ่งให้แต่ละขั้นตอนมีเวลาในการประมวลผลใกล้เคียงกัน ในกรณีของรูปที่ 5.2 นั้น การแบ่งมีความสมบูรณ์เนื่องจากทั้งสองขั้นตอน เป็นการบวกซึ่งใช้เวลาเท่ากันพอดี

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



รูปที่ 5.3: การแบ่งขั้นตอนการทำงาน

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

1. **Instruction Fetch** คือ การประมวลผล $\text{Instruction} \leftarrow \text{MEM}[\text{PC}]$ เป็นการดึงคำสั่งจากหน่วยความจำ(Memory) ตำแหน่งที่ชี้โดย PC เข้าสู่หน่วยประมวลผลกลาง พร้อมกับบวกค่า PC ให้ไปยังคำสั่งถัดไป
2. **Instruction Decode** และ **Register Fetch** คือ การตีความคำสั่งที่ถูกอ่านเข้ามาแล้วเพื่อสร้างสัญญาณควบคุม และ เตรียมข้อมูลใน Register รวมถึง extender ให้พร้อมสำหรับป้อนให้กับ ALU

3. **Execution** ในที่นี้ นอกจากรูปแบบการทำงานของ ALU ตามปกติแล้ว ยังรวมถึง Address Computation และ Branch Completion ด้วย ทั้งนี้การทำงานในขั้นตอน Execution จะแตกต่างกันไปขึ้นอยู่กับคำสั่งที่อ่านเข้ามามาได้ เช่นกรณีคำสั่งที่อ่านเข้ามามาได้เป็นคำสั่งเกี่ยวกับการคำนวน ALU ก็จะทำการคำนวนค่าตามคำสั่ง กรณีเป็นการอ่านข้อมูล ALU ก็จะทำการคำนวนตำแหน่งของหน่วยความจำที่ต้องการอ่านข้อมูล และ กรณีการ Branch ALU ก็จะคำนวนที่อยู่ของคำสั่งต่อไป (PC)
4. **Memory Access** กรณีที่คำสั่งซึ่งอ่านเข้ามามาได้นั้นเป็นคำสั่งเกี่ยวกับการอ่านเขียนข้อมูลจากหน่วยความจำ ในขั้นตอนนี้จะมีการอ่านหรือเขียนข้อมูล
5. **Write back** ซึ่งจะเกิดขึ้นเฉพาะกรณีที่ต้องมีการเขียนข้อมูลคืนกลับไปยัง register เท่านั้น (ในกรณีของ nanoLADA คือคำสั่ง ORI, ORUI, ADD, และ LW)

จากการทำงานใน 5 ขั้นตอนดังกล่าวข้างต้น จะเห็นว่าบางคำสั่ง (เช่น LI, LUI, ADD) จะใช้ 4 ขั้นตอนการทำงานโดยสามารถข้ามขั้นตอน Memory Access ไปได้ ส่วนคำสั่ง SW จะใช้เพียง 4 ขั้นตอนคือ 1 – 4 โดยจะไม่ข้ามตอนของการ Write back และมีเพียง 1 คำสั่งเท่านั้น (LW) ที่ต้องทำงานครบถ้วน 5 ขั้นตอน ส่วนคำสั่งอื่นๆ (BEQ และ JMP) จะทำงานได้โดยใช้เพียง 3 ขั้นตอนหรือน้อยกว่า หากวิเคราะห์อย่างหยาบจะได้ว่า ค่าเฉลี่ยของ CPI ที่ได้ออยู่ที่ต่ำกว่า 4 (คิดจาก $(3 + 4 + 5) / 4$ หารด้วย 3)

ทั้งเรายังไม่สามารถสร้างหน่วยควบคุมได้ทันที เนื่องจากหลักการทำงานข้างต้นมิได้อธิบายถึงที่มาของข้อมูลแต่ละขั้นตอนโดยละเอียด เพราะยังมิได้แสดงให้ทราบว่าจะต้องนำบิตใดของคำสั่งที่อ่านเข้ามาใช้อ้างอิง หรือ นำข้อมูลที่ได้ไปเก็บที่รีจิสเตอร์ใดโดยละเอียด เพื่อให้เห็นขั้นตอนการออกแบบที่ชัดเจน ลองพิจารณาขั้นตอนการออกแบบหน่วยประมวลผลกลางทั้ง 5 ขั้นตอน (จากหัวข้อ 4.1) อีกครั้งหนึ่ง จะพบว่าสิ่งที่เปลี่ยนไปคือ ตั้งแต่ขั้นตอนที่ 3 ซึ่งจะมีการออกแบบทางเดินข้อมูลที่เปลี่ยนไป ส่งผลให้ขั้นตอนที่ 4 และ 5 เปลี่ยนไปด้วย

เนื้อหาในบทนี้ เริ่มต้นด้วยการประเมินประสิทธิภาพของหน่วยประมวลผลกลางแบบ Multiple Cycle สำหรับสถาปัตยกรรมชุดคำสั่ง nanoLADA เปรียบเทียบกับหน่วยประมวลผลกลางแบบ Single Cycle ที่ได้จากบทที่ 4 เพื่อให้เห็นแนวโน้มของประสิทธิภาพว่าจะได้เท่าใด จากนั้น จึงจะกลับเข้าสู่การออกแบบหน่วยประมวลผลกลางโดยเริ่มต้นจากการทำทางเดินข้อมูล และจบที่สัญญาณควบคุม

5.1 ประสิทธิภาพของหน่วยประมวลผลกลางแบบ multiple cycle

เพื่อประกอบความเข้าใจ และ เป็นการวิเคราะห์ว่า หากนีคือการลงทุนเพื่อปรับหน่วยประมวลผลกลางให้เป็นแบบ multiple cycle แล้ว ผลที่ได้จะเป็นเช่นไร ประสิทธิภาพที่ได้คุ้มค่ากับการลงทุนหรือไม่

ก่อนอื่น ลองทบทวนความเข้าใจก่อนว่าเวลาในการประมวลผลของหน่วยประมวลผลกลาง (CPU TIME) สามารถคำนวณได้จากผลคูณของ จำนวนคำสั่ง ค่า CPI และ Cycle Time จากหน่วยประมวลผลกลางแบบ Single Cycle จะได้ว่า CPI เป็น 1 และหากเทียบตัวเลข Cycle Time จาก

บทที่ 4 จะได้ 190ns ลองให้ข้อมูลนี้ประกอบการวิเคราะห์ในตัวอย่างที่ 5.1

ตัวอย่างที่ 5.1: เปรียบเทียบประสิทธิภาพหน่วยประมวลผลกลางแบบ Single Cycle และ Multiple Cycle

กำหนดให้ หน่วยประมวลผลกลางแบบ Single Cycle ค่า Cycle Time เป็น 190ns หากทำการรับปุ่มหน่วยประมวลผลกลางดังกล่าวให้เป็นแบบ multiple cycle และได้ค่า Clock Cycle Time ใหม่เป็น 45ns ค่าเฉลี่ย CPI (average CPI) ควรจะเป็นเท่าใดเพื่อให้อย่างน้อยได้ประสิทธิภาพเท่าเดิม

$$CPU\ Time = \text{instruction count} \times CPI \times \text{Cycle Time}$$

$$CPU\ Time_{single\ cycle} = \text{instruction count} \times 1 \times 190\ (ns)$$

สมมุติว่าการรับปุ่มเดินข้อมูลและวงจรควบคุมทำให้เวลาของ clock cycle time เป็น 45ns (ต่อ 1 ขั้นตอนการทำงาน) จะได้ว่า

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

ดังนั้นเพื่อให้ได้ ประสิทธิภาพอย่างน้อยเท่าเดิม ค่า CPI เฉลี่ย (average CPI) ที่ได้ควรเป็น

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

$$\text{instruction count} \times 1 \times 190 = \text{instruction count} \times \text{average CPI} \times 45$$

$$190 = \text{average CPI} \times 45$$

$$\therefore \text{average CPI} = \frac{190}{45} \approx 4.2$$

จากตัวอย่างจะเห็นว่า หากหน่วยประมวลผลกลางแบบ multiple cycle ที่ได้มีค่าเฉลี่ยของ CPI เป็น 4.2 จะได้ประสิทธิภาพไม่ต่างจากเดิม อย่างไรก็ตามหากหน่วยประมวลผลกลางที่ได้มีค่าเฉลี่ย CPI น้อยกว่า 4.2 นั้นแปลว่าประสิทธิภาพที่ได้ จะต่ำกว่าเดิม

คราวนี้ เรายังคงวิเคราะห์ต่อว่า ค่าเฉลี่ยน CPI ที่ 4.2 มีความเป็นไปได้มากน้อยแค่ไหน หากพิจารณาว่า คำสั่งส่วนใหญ่ใช้ 4 cycle (หรือน้อยกว่า) และมีเพียงคำสั่ง LW เท่านั้นที่ใช้ 5 cycle จะพบว่า ดังนั้น หากมีคำสั่ง LW ไม่เกิน 20% ของคำสั่งทั้งหมด ก็จะได้ค่าเฉลี่ย CPI ที่ 4.2 และ ประกอบความเข้าใจที่มาของค่า 20% ดังกล่าว ลองพิจารณาตัวอย่างที่ 5.2

ตัวอย่างที่ 5.2: การคำนวณค่าเฉลี่ย CPI ของหน่วยประมวลผลกลางแบบ multiple cycle

กำหนดให้คำสั่ง LW มีค่า CPI เป็น 5 ส่วนคำสั่งอื่นๆ ในระบบมีค่า CPI เป็น 4 หากต้องการให้หน่วยประมวลผลกลางมีค่าเฉลี่ย CPI เป็น 4.2 จะต้องมีคำสั่ง LW ไม่เกินกี่เปอร์เซ็นต์ของคำสั่งทั้งหมดที่มีในระบบ

ให้จำนวนคำสั่ง LW ในระบบมีสัดส่วนเป็น k ค่าเฉลี่ย CPI คำนวนได้จาก

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

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

เนื่องจากค่า CPI ที่ต้องการคือ 4.2 ดังนั้น

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

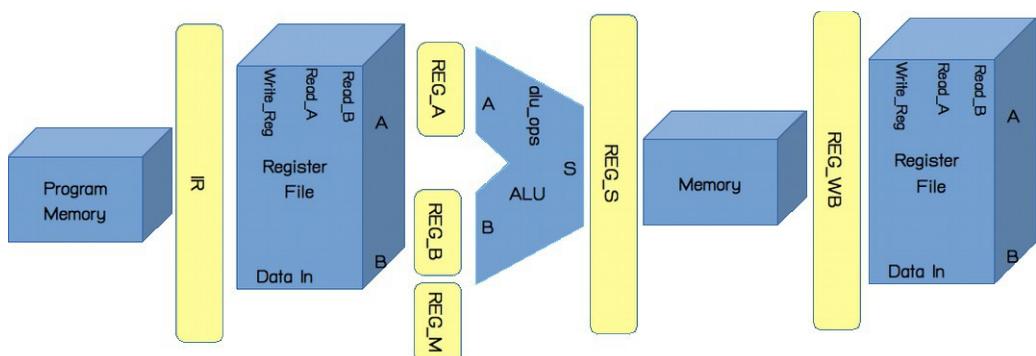
$$4.2 - 4 = k$$

$$\therefore k = 0.2$$

ข้อมูลดังกล่าวเป็นเพียงตัวอย่างเบื้องต้นประกอบการอธิบายเท่านั้น

5.2 ทางเดินข้อมูลสำหรับสถาปัตยกรรม nanoLADA แบบ multiple cycle

เนื่องจากในแต่ละสัญญาณนาฬิกา จะต้องมีการส่งต่อค่าที่จะใช้เป็น input ของวงจรในส่วนตัดไป ดังนั้นแนวทางในการปรับปรุงทางเดินข้อมูลให้รองรับการทำงานจึงตรงไปตรงมาคือ **ทำการเพิ่ม(แทรก) register สำหรับการส่งต่อค่าระหว่างแต่ละสถานะ** รูปที่ 5.4 แสดง register ที่ถูกแทรกเข้าไปเพื่อให้สามารถประมวลผลแบบ multiple cycle ได้ (ในรูปจะตัดสายสัญญาณออก เพื่อให้ดูได้ง่าย)



รูปที่ 5.4: การแทรก shadow register เพื่อปรับทางเดินข้อมูลให้รองรับการทำงานแบบ multiple cycle

จากทางเดินข้อมูลใหม่ จะเห็นว่ามี register ที่เกิดขึ้นมาใหม่อีก 5 ชุด (IR, REG_A, REG_B, REG_S, REG_WB) (เนื่องจาก register เหล่านี้ไม่สามารถเข้าถึงได้ด้วยการเขียนโปรแกรม หลายตำแหน่งเรียกว่า register ในลักษณะนี้ว่าเจริสเตอร์งา หรือ shadow register) ในการออกแบบจึงนิยมอธิบายการทำงานโดยภาษา Register-Transfer language (RTL) สำหรับการทำงานแต่ละขั้น ทั้งนี้ข้อสังเกตคือ RTL ในตอนนี้จะเป็น RTL ที่แสดงการทำงานของ register ที่เกิดขึ้นใหม่เหล่านี้ ซึ่งไม่สามารถเข้าถึงจากโปรแกรมด้วย เพื่อกันความสับสน เราจึงเรียก RTL ที่อธิบายการทำงานของคำสั่งตามสถาปัตยกรรมชุดคำสั่งว่า **Logical RTL** (ตามที่เคยแสดงในบทที่ 4) และจะเรียก RTL ที่อธิบายถึงการทำงานทุกขั้นตอนและแสดงค่า register ที่ซ่อนอยู่ภายในเหล่านี้ว่า **Physical RTL** (ตามความนิยมจะอธิบาย Physical RTL แยกตาม Cycle การทำงานที่เกิดขึ้นด้วย)

จากสถาปัตยกรรมชุดคำสั่ง nanoLADA สามารถอธิบายการทำงานด้วย Logical RTL และ Physical RTL ได้ดังนี้

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

Logical RTL:

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

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

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

Logical RTL:

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

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

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

Logical RTL:

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

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

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

Logical RTL:

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

Cycle	Physical RTL
1	$IR \leftarrow MEM[PC]$ $PC \leftarrow PC + 4$
2	$REG_A \leftarrow R[rs]$ $REG_B \leftarrow zero_ext(imm)$ $REG_M \leftarrow R[rt]$
3	$REG_S \leftarrow REG_A REG_B$
4	$MEM[REG_S] \leftarrow REG_M$
5	$R[rt] \leftarrow REG_WR$

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

Logical RTL:

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

Cycle	Physical RTL
1	IR ← MEM[PC] PC ← PC + 4
2	(ตรวจสอบ decode สัญญาณควบคุมทำงาน)
3	If (R[rs] == R[rt]) THEN PC ← PC + sign_ext(imm) ENDIF

หมายเหตุ ในทางปฏิบัติสามารถตัดแปลง register file ให้แสดงค่า logic 1 หรือ 0 ได้เมื่อทำการเปรียบเทียบค่า R[rs] และ R[rt] ได้ทันที จะช่วยให้สามารถทราบผลการเปรียบเทียบได้ภายใน cycle ที่สอง และไม่จำเป็นต้องใช้ REG_A และ REG_B

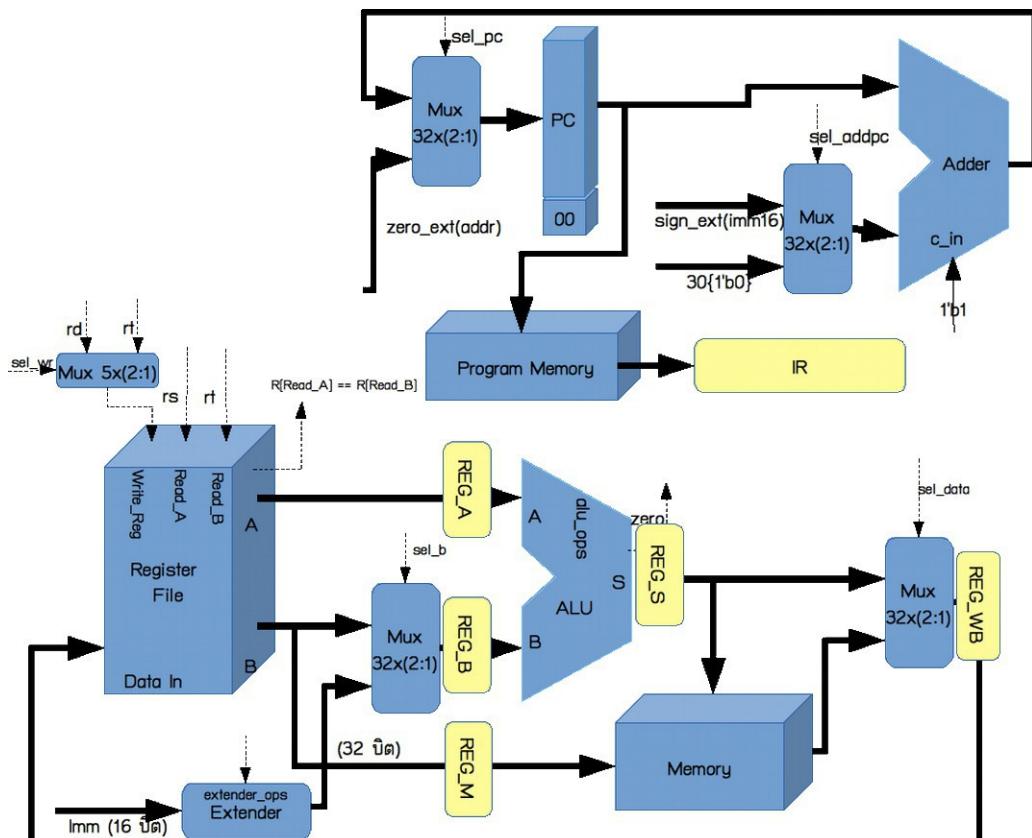
JMP addr ; (*J-type*)

Logical RTL:

```
PC ← (addr * 4)
```

Cycle	Physical RTL
1	IR ← MEM[PC] PC ← PC + 4
2	(ตรวจสอบ decode สัญญาณควบคุมทำงาน)
3	PC ← addr * 4

จาก Physical RTL ที่ได้ ลองทำการสร้าง data path ที่สมบูรณ์ จะได้ทางเดินข้อมูลดังรูปที่ 5.5 ซึ่งโดยหลักการจะคล้ายกับทางเดินข้อมูลแบบ Single Cycle เพียงแต่มี shadow register แทรกขึ้นมาเพื่อเป็นที่พักข้อมูลสำหรับส่งให้ยังหน่วยอื่นประมวลผลในสัญญาณนาฬิกาต่อไป

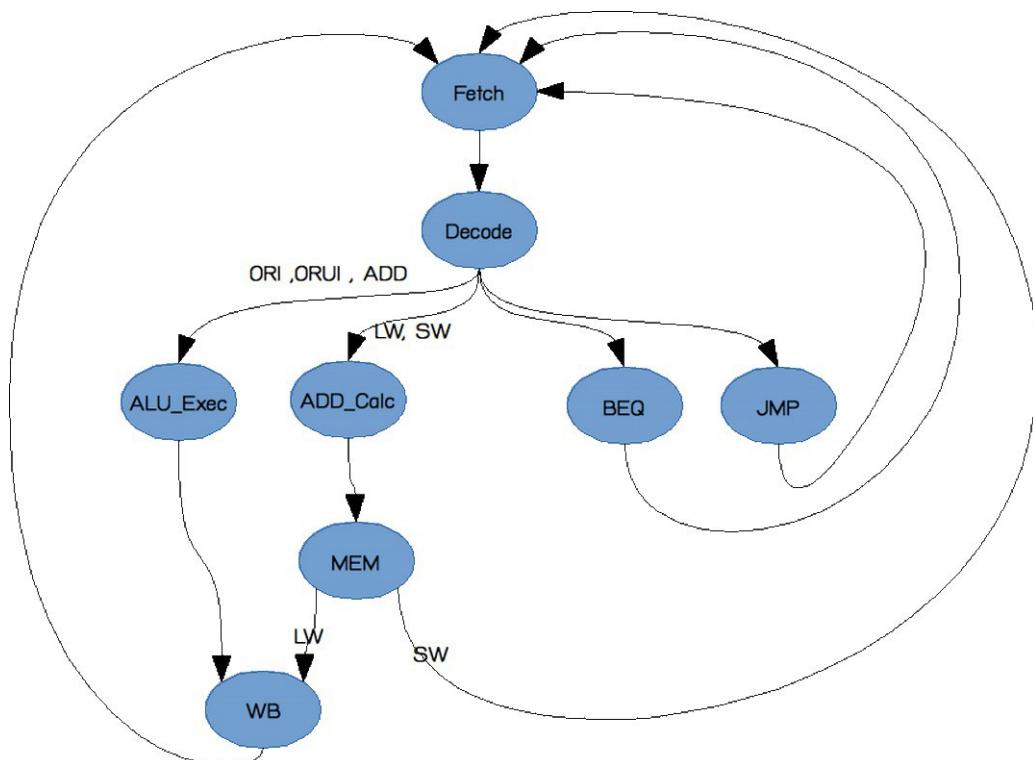


รูปที่ 5.5: ทางเดินข้อมูลสำหรับ multiple-cycle processor

5.3 สัญญาณควบคุมสำหรับ multiple cycle processor

หลังจากที่ได้ทางเดินข้อมูลเป็นอันเสร็จสิ้นการทำงานในขั้นตอนที่สามของการออกแบบ คราวนี้มาถึงขั้นตอนที่สี่ และ ขั้นตอนที่ห้า คือ **การสร้างสัญญาณควบคุม** ในขั้นตอนนี้หากจะวิเคราะห์ให้ดี จะเห็นว่าสัญญาณควบคุมต่างๆ ยังคงเหมือนเดิม เพียงแต่การทำงานจะแยกย่อยไปแต่ละสัญญาณพิเศษ ดังนั้นการออกแบบจะสร้างสัญญาณควบคุมในขั้นตอนนั้นเอง จึงเป็นการสร้างวงจรเชิงลำดับ (sequential circuit) แทน

จากการความรู้เดิมในการออกแบบวงจรเชิงลำดับ เราจะเริ่มต้นการออกแบบโดยการสร้างแผนภาพสถาณะ (State Diagram) หรือ การเขียน ASM Chart เพื่อแสดงการเปลี่ยนไหว้และเงื่อนไขในการเปลี่ยนสถานะต่างๆ สร้างฟังก์ชันเอาท์พุต และ ฟังก์ชันการเปลี่ยนสถานะ ซึ่งแผนภาพสถานะของหน่วยประมวลผลกลางนี้ สามารถแสดงได้ดังรูปที่ 5.6



รูปที่ 5.6: state diagram แสดงการทำงานของหน่วยประมวลผลกลางแบบ multiple cycle

จากรูปที่ 5.6 แสดงเพียงสถานะ (ในทางปฏิบัติ อาจจะสามารถดูบุรวม state บางอันได้อีก) โดยยังไม่มีสัญญาณควบคุมที่เกี่ยวข้อง (หันนี้เพื่อไม่ให้ภาพดูรก จนอ่านไม่ออก) เพื่อให้แนบภาพสถานะสมบูรณ์จะต้องทำการระบุสัญญาณที่เกี่ยวข้องในตารางที่ 5.1

ตารางที่ 5.1: สัญญาณควบคุมสำหรับหน่วยประมวลผลแบบ multiple cycle

สถานะ	สัญญาณที่เกี่ยวข้อง	สถานะถัดไป
Fetch	(wait for PC)	Decode
Decode	IF (op[5..4] == 01) THEN ; I-Type sel_wr \leftarrow 1 sel_b \leftarrow 1 ENDIF (wait for decoder)	ขึ้นกับคำสั่ง ALU_Exec ADD_Calc BEQ JMP
ALU_Exec	IF (opcode in {ORI, ORUI}) THEN alu_ops \leftarrow OR ELSE	WB

สถานะ	สัญญาณที่เกี่ยวข้อง	สถานะถัดไป
	alu_ops \leftarrow ADD ENDIF	
ADD_Calc	alu_ops \leftarrow ADD	MEM
MEM	sel_data \leftarrow 1 IF (op[2] == 1) THEN mem_wr=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 จะสามารถออกแบบวงจรได้โดยการใช้งานจารเริงผสม, ROM หรือ PLA ที่ได้ทั้งนี้ขึ้นอยู่กับผู้ออกแบบ ซึ่งลักษณะของหน่วยควบคุมที่ได้ จะมีอินพุตเป็น Instruction Register และสถานะของหน่วยควบคุม ส่วนเอาท์พุตจะเป็นสัญญาณสำหรับควบคุม Multiplexor และ ALU พร้อมกับการเปลี่ยนสถานะ ซึ่งลักษณะของหน่วยควบคุมนี้สามารถแสดงได้ดังภาพ โดยในที่นี้จะไม่ขอกล่าวถึงรายละเอียดในการออกแบบมากนัก หากผู้อ่านสนใจเพิ่มเติมสามารถศึกษาได้จากการออกแบบจรรยาบรรณทางชีววิทยา

ทั้งนี้ในการออกแบบสัญญาณควบคุมที่แสดงให้ดู ยังไม่ได้รวมการออกแบบเพื่อให้รองรับการเกิด Exception และ Interrupt ซึ่งจะทำให้โครงสร้างหน่วยประมวลผลกลางที่ได้มีความซับซ้อนยิ่งขึ้นไปอีก

5.4 แบบฝึกหัดท้ายบท

- การออกแบบหน่วยประมวลผลกลางโดยใช้ Single Cycle และ Multiple Cycle มีข้อดี ข้อเสียต่างกันอย่างไร จงยกตัวอย่างประกอบ
- ในการประมวลผลคำสั่งแต่ละคำสั่นนั้น หน่วยประมวลผลกลางมีขั้นตอนในการทำงานอย่างไร
- จากรากฐานแossซเมมบลิของสถาปัตยกรรม nanoLADA ที่กำหนดให้ หากออกแบบหน่วยประมวลผลกลางให้ทำงานแบบ Multiple Cycle จงตอบคำถามต่อไปนี้

```

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

```

- การทำงานในแต่ละคำสั่งจะใช้เวลา กี่ Cycle
- ที่ Cycle ที่ 7 หน่วยประมวลผลกลางกำลังทำงานอะไรอยู่
- โปรแกรมดังกล่าวข้างต้นจะใช้เวลาในการทำงานทั้งสิ้น กี่ Cycle และคิดเป็นกี่วินาที หาก Cycle Time เป็น 40ns
- จากรูปที่ 5.6 และตารางที่ 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

6 การเพิ่มประสิทธิภาพด้วย Pipeline

การทำ Pipeline เป็นการเพิ่มประสิทธิภาพให้กับหน่วยประมวลผลกลางในลักษณะของการเพิ่ม Throughput กล่าวคือ มีการทำงานหลายคำสั่งพร้อมกันในเวลาหนึ่ง ตัวอย่างเช่น สมมุติว่าการ เตรียมข้าวสาร ประกอบด้วยขั้นตอนย่อย 3 ขั้น คือ ขั้นที่ 1 การร่อนข้าวเปลือกเพื่อคัดสิ่ง杂质ปลอม ออกที่ป่นมากับข้าวเปลือก ขั้นที่ 2 นำข้าวเปลือกที่ได้จากขั้นแรกมาตำเพื่อให้เมล็ดข้าวหลุดออกจากเปลือกข้าว และ ขั้นที่ 3 ทำการแยกเปลือกข้าวออก เพื่อให้เหลือแต่เมล็ดข้าว หากเรามีกระดัง สำหรับร้อนแยกสิ่ง杂质ปลอมในขั้นที่ 1 อยู่เพียง 1 อัน มีครั้งละเดียวสำหรับต่อให้เมล็ดข้าวแยก ออกจากเปลือกเพียง 1 อัน และมีภาระนี้สำหรับทำการแยกเมล็ดข้าวออกจากเปลือกข้าวเพียง 1 อัน หากมีคนทำงานเพียง 1 คน จะต้องใช้เวลาในการทำงานทั้งหมด 3 ชั่วโมง แต่หากมีคนทำงาน 3 คน แบ่งกันทำงานในแต่ละขั้นตอน แล้วส่งต่อ อาจจะใช้เวลาในการทำงานเพียง 1 ชั่วโมงครึ่ง ทั้งนี้ เนื่องจากหากมีคนทำงานเพียง 1 คน จะต้องแยกสิ่ง杂质ปลอมปนให้เสร็จก่อน จนนั้นจึงจะทำการดำเนิน ให้เมล็ดข้าวออกจากเปลือกข้าว และจึงมีภาระนี้ 2 ส่วนออกจากกัน แต่หากมีคนทำงาน 3 คน เมื่อ คนแรกแยกสิ่ง杂质ปลอมปน แล้วส่งข้าวเปลือกที่แยกได้มากให้คนที่ 2 ดำเนิน คนแรกก็สามารถที่จะแยก ข้าวเปลือกได้ต่อไป พร้อมกับคนที่ 2 กำลังดำเนินการ Fetch คำสั่งที่ 3 ขึ้นมา ซึ่งส่งผลให้มี คำสั่งที่ถูกทำงานโดยหน่วยประมวลผลกลางมากกว่าหนึ่งคำสั่งในเวลาเดียวกัน ดังแสดงในรูปที่ 6.1²²

ในหน่วยประมวลผลกลางนั้น การเพิ่ม Throughput โดยการทำ Pipeline ก็อยู่ในลักษณะเดียวกัน คือ เมื่อคำสั่งแรกถูก Fetch ขึ้น และ ถูกส่งไปทำการ Decode ในช่วงเวลาหนึ่ง ประมวลผล สำหรับการ Fetch ก็กำลัง Fetch คำสั่งถัดไปขึ้นมา และเมื่อคำสั่งแรกทำการ Execute คำสั่งที่ 2 ก็อยู่ระหว่างการ Decode และ หน่วย Fetch ก็กำลังทำการ Fetch คำสั่งที่ 3 ขึ้นมา ซึ่งส่งผลให้มี คำสั่งที่ถูกทำงานโดยหน่วยประมวลผลกลางมากกว่าหนึ่งคำสั่งในเวลาเดียวกัน ดังแสดงในรูปที่ 6.1²²

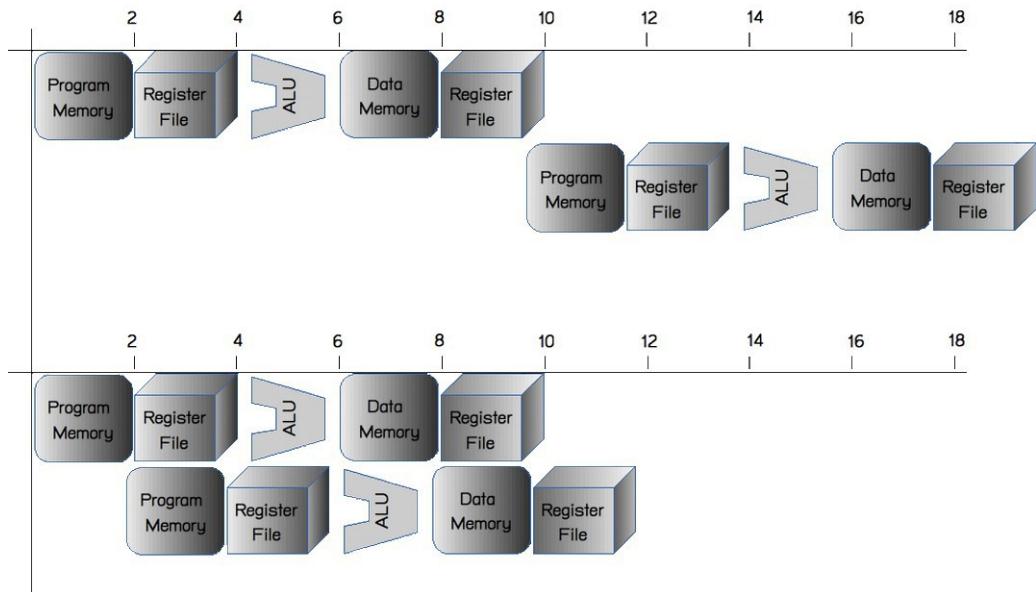
จากรูปจะพบว่า การทำงาน 2 คำสั่งกรณีไม่มี Pipeline นั้น จะใช้เวลาในการทำงานถึง 20 หน่วยเวลา ในการทำงาน 5 ขั้นตอนที่เมื่อมีการทำ Pipeline จะใช้เวลาเพียง 12 หน่วยเวลาเท่านั้น อย่างไรก็ตามจะสังเกตุ ได้ว่า แต่ละคำสั่งไม่สามารถ Fetch หรือถูกอ่านขึ้นมาพร้อมกันได้ ทั้งนี้เนื่องจากแต่ละหน่วยของ หน่วยประมวลผลกลางสามารถให้บริการได้เพียงหนึ่งคำสั่งในเวลาหนึ่งๆ เท่านั้น

จากรูป pipeline ในรูป จะพบว่ากรณี pipeline จะมีการทำงานย่อຢายใน 5 ขั้นตอน ซึ่งหมายถึง เมื่อทำงานเต็มรูปแบบจะมีคำสั่งทำงานพร้อมกัน 5 คำสั่ง ลักษณะนี้ ทางสถาบันตยกรรมคอมพิวเตอร์ จะเรียกว่า pipeline แบบ 5 สถานะ (5-stage pipeline)

เพื่อประกอบความเข้าใจ เนื้อหาในบทนี้จะอธิบายการทำงานและการออกแบบ Pipeline เป็นต้น โดย เริ่มจากการเบรียบเทียบประสิทธิภาพที่ได้รับ ปัญหาที่เป็นอุปสรรคต่อการทำ Pipeline และการ ปรับปรุงหน่วยประมวลผลกลางเพื่อให้รองรับการทำงานของ Pipeline ทั้งนี้อยู่บนสมมุติฐานของ

²² เพื่อความสะดวกในการอธิบายการทำงาน ภายใต้ภาพ จึงใช้การแยกสิ่งที่จะรับ ปัญหาที่เป็นอุปสรรคต่อการทำ Pipeline และการ ปรับปรุงหน่วยประมวลผลกลางเพื่อให้รองรับการทำงานของ Pipeline ทั้งนี้อยู่บนสมมุติฐานของ

สถาปัตยกรรม nanoLADA ที่ว่าคำสั่งทุกคำสั่งมีความยาวเท่ากัน มีรูปแบบคำสั่งที่จำกัด (3 รูปแบบ) และมีคำสั่งที่เกี่ยวข้องกับหน่วยความจำเพียง Load และ Store เท่านั้น ทั้งนี้หากผู้อ่านต้องการทราบซึ่งสถาปัตยกรรมที่ซับซ้อน สามารถหาอ่านได้ตามเอกสารอ้างอิงของผู้ผลิตหน่วยประมวลผลกลางทั่วไป



รูปที่ 6.1: เปรียบเทียบเวลาการทำงานแบบมี pipeline และ ไม่มี pipeline

6.1 ประสิทธิภาพของ Pipeline กรณีอุดมคติ

อ้างอิงถึงการวิเคราะห์ประสิทธิภาพของหน่วยประมวลผลกลางแบบ Multiple Cycle ในตอนที่ 5.1 จะพบว่าหน่วยประมวลผลกลางแบบ Multiple Cycle มีแนวโน้มที่จะมีประสิทธิภาพดีกว่าหน่วยประมวลผลกลางแบบ Single Cycle โดยการผันแปรค่าของ CPI เทียบกับ Clock Cycle Time ที่สั้นลง

สำหรับการทำPipeline แล้ว เวลาที่ใช้ในการทำงานแต่ละคำสั่งนั้น มีได้สั้นลง หากแต่เวลาโดยรวมของการทำงานจะสั้งลง (เป็นการปรับปรุงประสิทธิภาพโดยการเพิ่ม Throughput ดังที่ได้กล่าวไปแล้ว) เพื่อประกอบความเข้าใจลองพิจารณารูปที่ 6.1 และ ตัวอย่างที่ 6.1

จากตัวอย่าง ข้อสังเกตุคือ เมื่อ pipeline เริ่มต้น จนคำสั่งเต็มแล้ว คำสั่งที่เหลือจะใช้เวลาเพียง 1 cycle เพื่อให้ประมวลผลเสร็จ (เพื่อความเข้าใจเพิ่มเติม ขอให้ลองวางแผนการทำงานของ pipeline เพิ่มเติมในแบบฝึกหัดท้ายบท) ดังนั้นสมการทั่วไปสำหรับแสดงเวลาที่ใช้ในการประมวลผลแบบ pipeline จะเป็นสมการที่ 6.1

$$CPU\ TIME_{pipeline} = [no.\ of\ pipeline + (no.\ of\ instructions \times CPI)] \times cycle\ time$$

สมการที่ 6.1: CPU Time (pipeline)

ทั้งนี้ ขอให้นึกอยู่เสมอว่า CPU Time ดังกล่าวคิดกรณี ideal กล่าวคือ pipeline สามารถทำงานได้ต่อเนื่องไม่มีการสอดดูด (ซึ่งจะกล่าวถึงการสอดดูดของ pipeline ต่อไปในหัวข้อ 6.2)

ตัวอย่างที่ 6.1: การเปรียบเทียบประสิทธิภาพ Pipeline กรณีอุดมคติ

จาก pipeline แบบ 5 สถานะที่กำหนดในรูปที่ 6.1 หากมีจำนวนคำสั่งมาก (เช่นเป็นอนันต์) และสมมุติให้การทำงานทุกคำสั่งมี 5 cycle เท่ากัน การทำงานแบบ pipeline แบบอุดมคติ (ต่อเนื่อง ไม่มีการหยุด) และ การทำงานแบบ multiple cycle (ไม่มี pipeline) จะให้ประสิทธิภาพต่างกันเท่ากัน

การทำงานแบบ multiple cycle เมื่อมีจำนวนคำสั่งเป็น k จะได้ว่า

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

การทำงานแบบ pipeline เมื่อมีจำนวนคำสั่งเป็น n จะได้ว่า

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

หากต้องการเปรียบเทียบประสิทธิภาพว่า แบบ pipeline เเร็วกว่าแบบ multiple cycle กี่เท่า จะได้ว่า

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

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

กรณีมีปริมาณคำสั่ง (k) มาจนเป็นอนันต์ จะได้ค่า n ดังนี้

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

\therefore ประสิทธิขึ้นกว่าประมวลผลแบบ pipeline จะเร็วกว่า 5 เท่า (จำนวนขั้นตอนของ pipeline)

6.2 ปัญหาที่เป็นอุปสรรคต่อการทำ Pipeline

หากวิเคราะห์การทำในลักษณะของ Pipeline ให้ดีจะพบว่าการทำ Pipeline นั้น จะกระท้ำได้ก็ต่อเมื่องานในแต่ละขั้นตอนแยกเป็นอิสระจากกันได้ ดังนั้นในการออกแบบหน่วยประมวลผลกลางให้

สามารถทำงานแบบ Pipeline ได้ ปัญหាដอย่างแรกคือ การทำให้งานในแต่ละขั้นตอนเป็นอิสระจากกัน เพื่อประกอบความเข้าใจจะอธิบายเบรียบเทียบปัญหาการเป็นอิสระจากกันในแต่ละขั้นตอนการทำงาน pipeline ด้วยการเตรียมข้าวสาร(ตำข้าว)ในทำงองเดียวกับที่ได้กล่าวนำก่อนเข้าสู่บริการ ดังนี้

ในการเตรียมข้าวสารซึ่งมี 3 ขั้นตอนได้แก่ (1) การร่อนข้าว (2) การตำข้าว (3) การแยกเปลือกข้าว หากขั้นตอนที่ 1 และขั้นตอนที่ 3 ต้องใช้เครื่องมือ(กระดัง)เหมือนกัน แต่มีกระดังเพียง 1 อันเท่านั้น ที่ใช้งานได้ เช่นนี้แล้ว ก็คงไม่สามารถที่จะแบ่งการทำงานในลักษณะของ Pipeline ได้ (ปัญหาในลักษณะดังกล่าวนี้เรียกว่า **Structural Hazard** ดังจะกล่าวถึงต่อไป) นอกจากนี้ยังมีปัญหลักษณะอื่นอีก เช่น จาบงประเทก อาจจะต้องตำและแยกเปลือกข้าวหลายรอบ ดังนั้น ถึงแม้ว่าขั้นตอนที่ 2 จะว่างอยู่ ก็อาจจะไม่สามารถเริ่มตำข้าวชุดใหม่ได้ เพราะต้องรอตำข้าวชุดเดิมให้เสร็จก่อน เป็นต้น

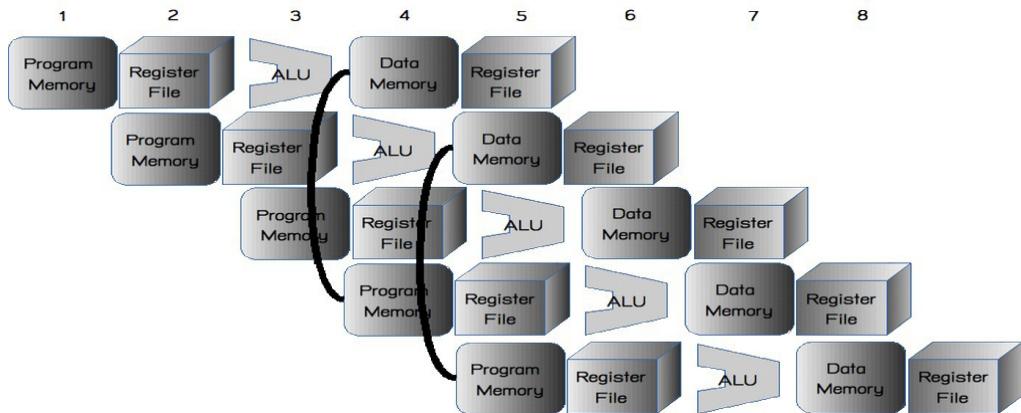
ในการทำPipeline มีปัญหาหลักที่เป็นอุปสรรคต่อประสิทธิภาพและการทำงานของ Pipeline ทั้งสิ้น 3 ประเด็น คือ **ปัญหากรณีโครงสร้างของระบบหรือ Resource** ที่ต้องใช้งานร่วมกัน (**Structural Hazard**) **ปัญหของกการประมวลผลข้อมูลซึ่งมีความเกี่ยวเนื่องกัน** (**Data Hazard**) และ **ปัญหการหาคำสั่งถัดไปของระบบในกรณีการ Jump หรือ Branch** (**Control Hazard**) ซึ่งในที่นี้จะกล่าวถึงปัญหาแต่ละอันและการแก้ปัญหาเบื้องต้น โดยใช้วิธีการพื้นฐาน ดังรายละเอียดต่อไปนี้

6.2.1 Structural Hazard

Structural Hazard เป็นปัญหาที่เกิดจากโครงสร้างของระบบไม่รองรับการทำPipeline เช่น กรณีของหน่วยความจำซึ่งทำหน้าที่เป็นทั้ง Instruction Memory และ Data Memory พร้อมกัน หากเราพิจารณารูปที่ 6.2 จะพบว่าที่ Cycle ที่ 4 นั้น หน่วยความจำจะถูกใช้งานเป็น Instruction Memory เพื่อทำการ Fetch และ Data Memory เพื่อทำการอ่านข้อมูลพร้อมกัน ซึ่งปัญหานี้อาจจะแก้ได้โดยการเพิ่ม Instance ขององค์ประกอบนั้นๆ ให้สามารถให้บริการพร้อมกันได้ นกรณีการเพิ่ม Instance อาจทำได้โดยการแยก Instruction Memory และ Data Memory ให้ต่อ กัน Bus ที่เป็นอิสระจากกัน หรือ แยก Cache ของ Instruction Memory กับ Data Memory ออกจากกัน (ดังจะกล่าวต่อไปในบทที่ 7)

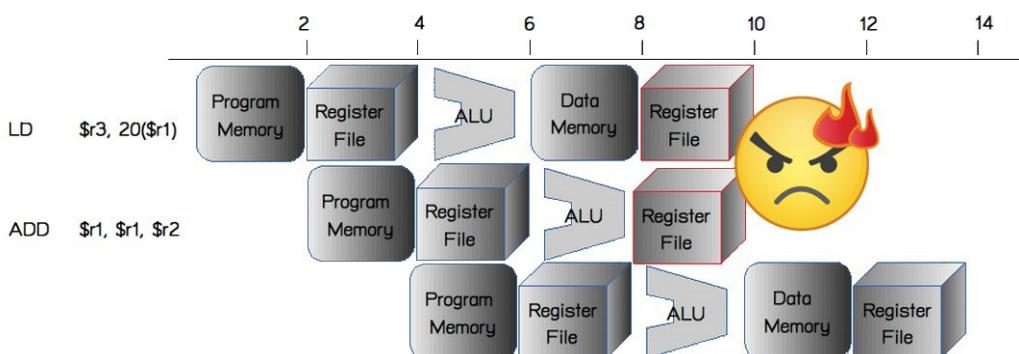
หากเบรียบเทียบ structural hazard กับตัวอย่างการตำข้าวคือ กรณีที่กระดังสำหรับการร่อนข้าว (ขั้นตอนที่ 1) และ สำหรับการแยกข้าวเปลือก (ขั้นตอนที่ 3) เป็นชุดเดียวกัน จึงไม่สามารถดำเนินการพร้อมกันได้ ดังนั้นวิธีการแก้ปัญหา hazard ลักษณะนี้ที่ตรงไปตรงมาที่สุดคือ การเพิ่ม instance ของ resource (ในกรณีนี้คือการเพิ่มกระดังเพิ่มอีก 1 ชุด) เป็นต้น

อีกกรณีหนึ่งของ Structural Hazard ใน nanoLADA คือ กรณีที่หน่วยประมวลผลกลางต้องใช้ ALU (ADDER) เพื่อการบวก PC ในการทำInstruction Fetch และ ทำการ Execute คำสั่งพร้อมกัน หากมี ALU เพียง 1 ชุด จะทำให้ไม่สามารถทำPipeline ได้ ในทำงองเดียวกัน วิธีแก้ไขโดยการเพิ่ม Instance คือเพิ่ม ALU (ADDER) เป็น 2 ชุด ให้ทำงานแยกเป็นอิสระจากกัน



รูปที่ 6.2: ปัญหา structural hazard

นอกจากนี้ ยังมีปัญหา structural hazard อีกแบบหนึ่งซึ่งเกิดจากกรณีที่เวลาในการทำงานไม่เท่ากันของแต่ละคำสั่ง เพื่อประกอบความเข้าใจ ลองดูตัวอย่างในรูปที่ 6.3 จะเห็นว่าคำสั่ง ADD ไม่จำเป็นจะต้องทำงานในชั้นที่ 4 คือ Memory Access ดังนั้น จึงสามารถข้ามไปยังชั้นตอนที่ 5 คือ Write Back ได้เลย ผลที่ได้คือ ขั้นตอนการ Write Back ของคำสั่ง ADD จะตรงกับขั้นตอน Write Back ของคำสั่ง LD ซึ่งมาก่อนหน้าพอดี ทำให้ไม่สามารถทำงานได้ เพราะต้องใช้ resource พร้อมกัน

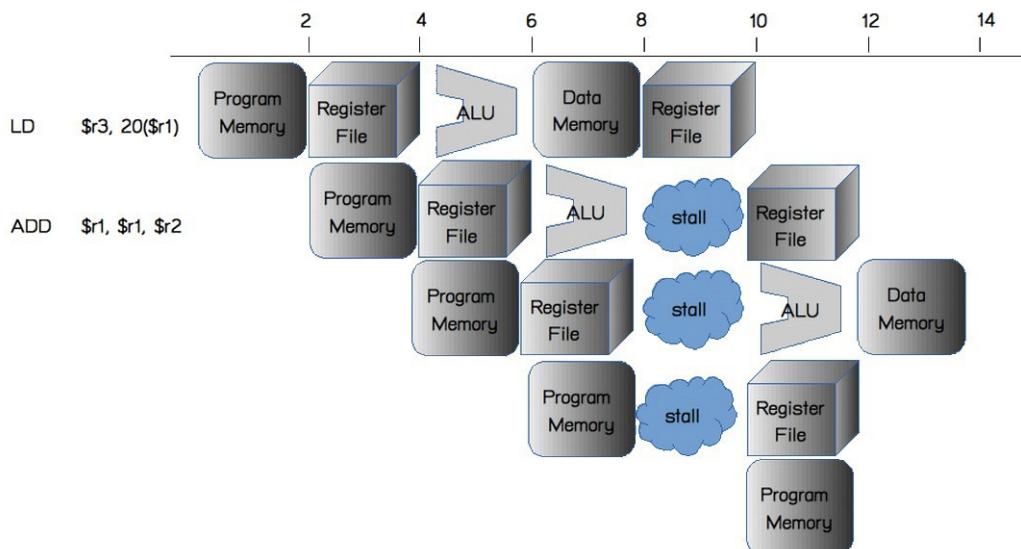


รูปที่ 6.3: ปัญหา structure hazard จากการที่แต่ละคำสั่งใช้นวน cycle ไม่เท่ากัน

เนื่องจากกรณีนี้ ไม่สามารถที่จะเพิ่ม instance ของ resource ได้ (ไม่สามารถเพิ่มชุดของ register file เข้าไปใหม่ได้) ดังนั้น จึงเหลือแนวทางในการแก้ปัญหาเพียงสองแนวทาง คือ การ stall (หยุดการทำงานของ pipeline อันล่างที่ตามมา) และ การปรับขั้นตอนการทำงานของคำสั่ง R-type (ในที่นี้คือคำสั่ง ADD) เพื่อหลีกเลี่ยงปัญหาการเกิด structural hazard

สำหรับการ stall ขอเสียที่เห็นชัดเจนคือ pipeline ด้านล่างจะต้องหยุดทั้งหมด นั่นหมายความว่า การทำงานของ ALU, การอ่าน register file และ การ fetch ข้อมูลจาก program memory จะต้อง

หยุดชะงักไปด้วย ดังแสดงในรูปที่ 6.4 ซึ่งหากมีการ stall ในกลักษณ์นี้มากมาก ย่อมส่งผลเสียต่อประสิทธิภาพโดยรวมของ pipeline แน่นอน

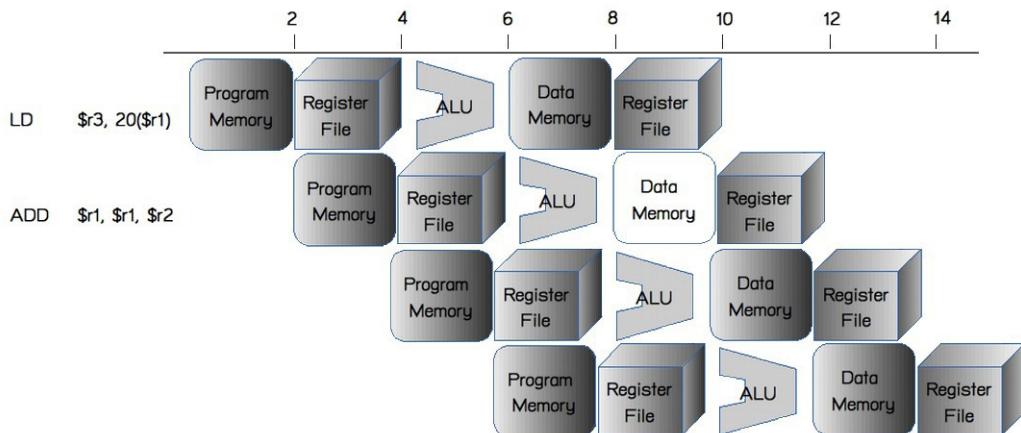


รูปที่ 6.4: การแก้ปัญหา structural hazard ด้วยการ stall

ดังนั้น หากสามารถปรับการทำงานคำสั่ง R-type (คำสั่ง ADD ในที่นี้) ให้สามารถเลี่ยงการเกิด structural hazard ได้ จะเป็นทางออกที่ดีกว่า ในกรณีของสถาปัตยกรรม nanoLADA นี้ สามารถทำได้ง่ายโดยการเพิ่มขั้นตอน Memory Access ให้กับคำสั่ง R-type ทั้งหมด ข้อเสียของวิธีการนี้คือ การทำงานของคำสั่ง R-type (เช่น คำสั่ง ADD) จะเสร็จช้าลงไป 1 cycle แต่หากวิเคราะห์ภาพรวมแล้ว อาจจะมีได้ส่งผลเสียต่อประสิทธิภาพโดยรวมของระบบ เพราะผลที่ได้คือ ไม่มีเกิดการ stall (ไม่มีคำสั่งใดต้องหยุดการทำงาน) และถึงแม่คำสั่งบางคำสั่งจะเสร็จช้าไป 1 cycle แต่ ทุก 1 cycle ก็ยังมีคำสั่งทำงานเสร็จอยู่เหมือนเดิม จึงมีได้ทำให้ประสิทธิภาพแตกต่างไปแต่อย่างใด

เพื่อประกอบความเข้าใจ ลองวิเคราะห์รูปที่ 6.5 จากรูปจะเห็นว่าขั้นตอน Memory Access ของคำสั่ง ADD จะเป็นสิ่งที่ซึ่งเป็นการชี้ว่าไม่มีการอ่านและไม่มีการเขียนข้อมูลเดิมขึ้นแต่อย่างใด ซึ่งการทำงานในระดับ RTL จะเป็นการส่งผ่านค่าที่ได้จาก ALU ต่อไป เพื่อรับการ Write Back ใน clock ถัดไป

ข้อควรคิด อ้างอิงจาก รูปที่ 5.2 ซึ่งแสดงขั้นตอนการทำงานของแต่ละคำสั่ง จะพบว่า นอกจากคำสั่ง R-type ซึ่งมีการทำงาน 4 cycle แล้ว ยังมีคำสั่ง SW, BEQ ฯ ลง JMP ซึ่งมีการทำงานเป็น 4 cycle หรือ 3 cycle ตามลำดับ เหตุใดการทำงานของคำสั่งตั้งกล่าว จึงไม่ส่งผลให้เกิดปัญหา structural hazard (คำตอบขอให้เป็นแบบฝึกหัดท้ายบทสำหรับผู้เรียนได้ขับคิดต่อไป)



รูปที่ 6.5: การแก้ structural hazard โดยการปรับขั้นตอนการทำงานของคำสั่ง R-type

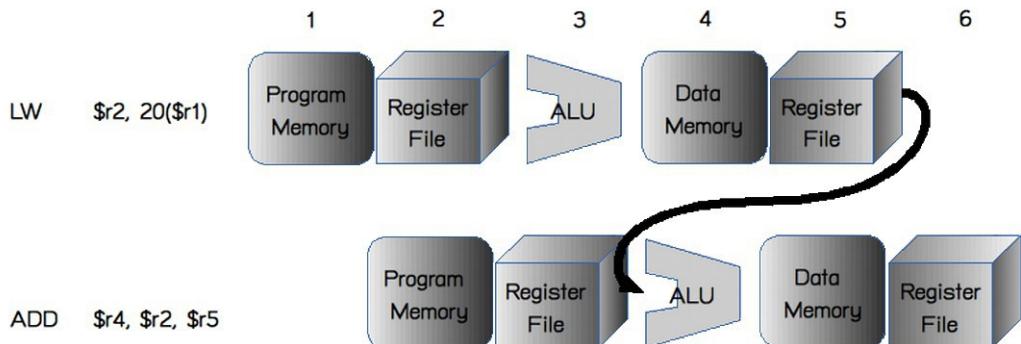
6.3 Data Hazard

Data Hazard นั้น เป็นปัญหาที่เกิดจากการประมวลผลคำสั่งที่มีความเกี่ยวเนื่องกัน (Dependency) โดยหากวิเคราะห์จากรูปที่ 6.6 พบรากурсของการทำงานของคำสั่ง LW อันที่หนึ่งนั้น ข้อมูลที่ได้จากหน่วยความจำตำแหน่งที่ $20 + \$r1$ จะถูกนำมายกเข้าใน rejister \$r2 ใน Cycle ที่ 5 (หากผู้อ่านไม่ทราบว่าแต่ละ Cycle ของแต่ละคำสั่งมีการทำงานอย่างไร ให้คลิกกลับไปพิจารณาขั้นตอนการทำงานในบทที่ 5 อีกครั้งหนึ่ง) ทั้งนี้เนื่องจากเป็นการทำงานที่เกี่ยวข้องกับหน่วยความจำ ซึ่งจะได้ข้อมูลจากหน่วยความจำในตอนจบของ Cycle ที่ 4 ในขณะที่คำสั่งถัดมาต้องการใช้งาน rejister \$r2 และ rejister \$r5 ใน Cycle ที่ 2 ทั้งนี้ข้อมูล rejister \$r2 นั้นจะได้มาจากการทำงานของคำสั่งแรกซึ่งจะเสร็จสิ้นใน Cycle ที่ 5 ซึ่งระบบไม่สามารถนำข้อมูลที่จะได้รับในอนาคตมาประมวลผลได้ดังแสดงด้วยเส้นที่บินในรูป ดังนั้นคำสั่งถัดมาจึงไม่สามารถเริ่มทำงานได้ เพราะข้อมูลที่จำเป็นสำหรับการเริ่มทำงานยังไม่พร้อมให้ใช้งานจนกว่าจะจบการทำงานใน Cycle ที่ 5

เพื่อให้เห็นภาพ ขอเปรียบเทียบกับตัวอย่างโปรแกรมเบื้องต้นดังนี้

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

จากตัวอย่างโปรแกรมนี้ หากการทำงาน $a = 7$ เสร็จสิ้นใน cycle ที่ 5 แต่ เราต้องการทำ $b = a + 5$ ใน cycle ที่ 3 (ขั้นตอนการอ่านค่าจาก register) หมายความว่า หากมีการทำการบวกเกิดขึ้น ผลบวกที่ได้จะผิด เพราะค่า a ที่ถูกนำไปใช้จะเป็นค่าที่ผิด (hazard)



รูปที่ 6.6: ปัญหา data hazard

จากรูปที่ 6.6 หาก register เดียวกันสามารถถูกอ่านและเขียนได้พร้อมกันโดยการประมวลผลข้อมูลถูกต้อง จะพบว่าหากการทำงานของคำสั่ง ADD นั้น ข้ามมาอีก 2 Cycle จะทำให้การประมวลผลของหน่วยประมวลผลกลางถูกต้อง แต่หากจำเป็นจะต้องมีการเขียนค่าลง register ให้เสร็จสิ้นก่อนจึงจะอ่านค่าใหม่ออกมาได้ จะต้องทำให้คำสั่ง ADD เริ่มข้ามไป ให้คำสั่ง ADD เริ่มข้าม 3 Cycle

ดังนั้นการแก้ปัญหาของ Data Hazard แนวทางหนึ่งคือ การแก้ปัญหาด้วยวิธีการทางซอฟต์แวร์โดยการแทรกคำสั่งอื่นๆ ที่ไม่มีการใช้ข้อมูลที่เกี่ยวข้องกันเข้าไประหว่างคำสั่งที่ 1 และ 2 ทั้งนี้ต้องไม่ทำให้การการทำงานของโปรแกรมผิดพลาดไปจากเดิม การแก้ปัญหา hazard ด้วยวิธีการทางซอฟต์แวร์ ในลักษณะนี้เรียกว่า rescheduling กล่าวคือ การโยนปัญหา hazard ไปให้ซอฟต์แวร์แก้ไข

ข้อเสียของการแก้ปัญหา hazard ด้วยวิธีการทางซอฟต์แวร์คือ ทุกครั้งที่มีสถาปัตยกรรมรุ่นใหม่ออก มา อาจจะต้องทำการ compile ซอฟต์แวร์ใหม่เพื่อทำการแก้ปัญหา hazard อีกรอบ ดังนั้นหากสามารถปรับการแก้ปัญหา hazard ให้จัดการด้วยวิธีการทาง硬件ได้ จะช่วยให้สถาปัตยกรรมชุดคำสั่งมี portability มาตรฐานมากขึ้น

เพื่อประกอบความเข้าใจ จะขออธิบายวิธีการแก้ปัญหา data hazard ด้วยแนวทางทางทางซอฟต์แวร์และฮาร์ดแวร์ ดังนี้

6.3.1 การแก้ปัญหา Data Hazard ด้วยวิธีการทางซอฟต์แวร์

การแก้ปัญหา data hazard ด้วยวิธีการทางซอฟต์แวร์คือ การทำให้ซอฟต์แวร์รับรองว่า จะไม่เกิดการอ้างอิงข้อมูลในลักษณะที่ติดกันเกินไปจนทำให้เกิดปัญหา เนื่องจากในปัจจุบันซอฟต์แวร์มักพัฒนาด้วยภาษา率ดับสูง ด้วยเหตุนี้ Complier จึงมีการพัฒนาให้สามารถทำการ Optimize ด้วยวิธีการ code motion²³ เพื่อเรียงคำสั่งให้เหมาะสมกับการทำงานของหน่วยประมวลผลกลางที่มี Pipeline

ในกรณีของรูปที่ 6.6 เนื่องจากโปรแกรมที่ให้มามีเพียงสองคำสั่งจึงไม่สามารถที่จะ reschedule ด้วย

²³ เทคนิคเรื่อง code motion เคยกล่าวถึงแล้วในหัวข้อที่ 3.6.1 อย่างไรก็ตามหน่วยสถาปัตยกรรมชุดคำสั่งบางแบบ บังคับว่า compiler จะต้องทำ code motion เพื่อแก้ปัญหา hazard เสมอ ทั้งนี้อุปสรรคของการทำ code motion คือมักติดอยู่ใน basic block เดียวกัน แม้ไม่สามารถทำงานข้าม block ได้

การเลือกคำสั่งที่อยู่ก่อนหน้าหรือคำสั่งที่ตามมาที่หลังแทรกรเข้ามาแทน แต่หากจำเป็นต้องแทรกรคำสั่งเพื่อให้การทำงานไม่เกิด hazard และ ตัวเลือกของคำสั่งที่เหมาะสมมากที่สุดคือ NOP (No Operation) ซึ่งมีความหมายว่า ไม่มีการประมวลผลใดๆ หรือ ให้อยู่เฉยเฉย ดูตัวอย่างที่ 6.2

ป ร ะ ก อ บ ค า อ ช ิ บ ย

ตัวอย่างที่ 6.2: การแทรกรคำสั่ง NOP เพื่อแก้ปัญหา data hazard

จากตัวอย่างการทำงานของ Pipeline ซึ่งมีปัญหา data hazard ในรูปที่ 6.6 จงทำการแทรกรคำสั่ง NOP เพื่อให้หน่วยประมวลผลกลางสามารถทำงานได้โดยไม่มีปัญหา hazard

จากตัวอย่าง code ที่กำหนดให้มีเพียงสองคำสั่ง คือ

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

หากไม่ต้องการให้เปิด hazard จะต้องให้แก้ไขให้ขั้นตอนการอ่าน \$r2 อยู่ที่ cycle ที่ 6 (สมมุติว่า register จะต้องทำการเขียนค่าใหม่เสร็จก่อน จึงจะทำการอ่านค่าที่ถูกต้องออกมาได้ ดังนั้น จะต้องตันให้ AND ข้าอกไปอีก 3 cycle ดังนี้

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

ข้อควรระวัง จากตัวอย่างจะสังเกตุว่า หากมีการเรียงลำดับคำสั่งในลักษณะที่เป็นการนำคำสั่ง OR \$r5,\$r5,\$r1 มาทำงานก่อนหน้าคำสั่ง AND \$r4,\$r2,\$r5 ได้ เพราะอาจจะทำให้ค่า register \$r5 ที่ใช้ในการประมวลผลผิดพลาดไประหว่างการทำคำสั่ง AND \$r4,\$r2,\$r5

อย่างไรก็ตาม **ข้อเสียของการแทรกรคำสั่ง NOP** เข้าไปนั้น คือ **ทำให้โปรแกรมมีขนาดใหญ่และสินเปลืองเวลาในการทำงานมากขึ้น** ดังนั้นแนวทางที่ดีกว่าคือ **การย้ายหรือเรียงลำดับคำสั่งโดยการนำคำสั่งก่อนหน้าหรือคำสั่งที่อยู่ล้านท้าย มาแทรกโดยยังให้ผลลัพธ์จากการทำงานถูกต้องเหมือนเดิม**

หากมีคำสั่งอื่นให้เลือกได้ในการทำ reschedule (ดัง ตัวอย่างที่ 6.3) จะช่วยให้ประสิทธิภาพที่ได้หลังจากการทำ reschedule ไม่ลดลง (เพราะไม่ต้องมีการแทรกรคำสั่งเพิ่ม เป็นเพียงการเรียงลำดับคำสั่งใหม่) **ข้อควรระวัง** เช่นเดิมคือ จะต้องไม่มีการสลับลำดับจนทำให้ผลลัพธ์ที่ได้ผิดไป

ตัวอย่างที่ 6.3: การทำ rescheduling เพื่อให้ไม่เกิดปัญหา data hazard

จะทำการเรียงลำดับโปรแกรมที่กำหนดให้ดังกล่าวโดยไม่มีการแทรกคำสั่งอื่น (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

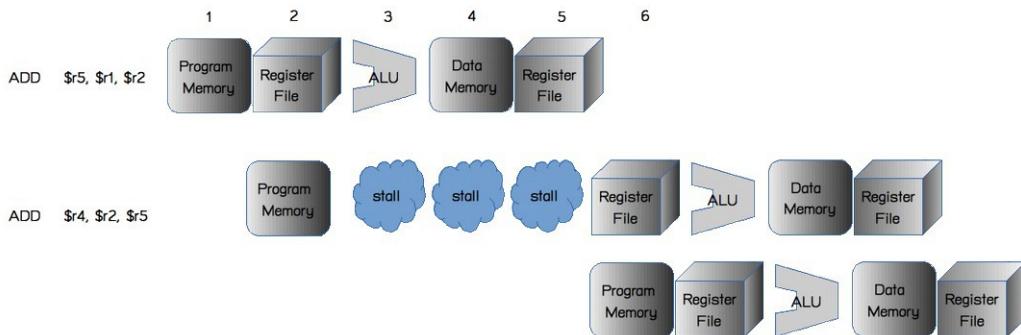
จาก code ที่กำหนดให้ หากทำการวิเคราะห์ลำดับการทำงานจะพบว่า 1 ต้องทำก่อน 2 และ 2 ต้องทำก่อน 3 (เนื่องจากคำสั่งที่ 3 มีการเปลี่ยนแปลงค่า \$r5 หากทำคำสั่งที่ 3 ก่อนคำสั่งที่ 2 อาจทำให้ค่า \$r5 เปลี่ยนแปลงไปได้) ดังนั้นอาจเรียงลำดับคำสั่งใหม่ได้เป็น

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

6.3.2 การแก้ปัญหา Data Hazard ด้วยวิธีการ hardware forward

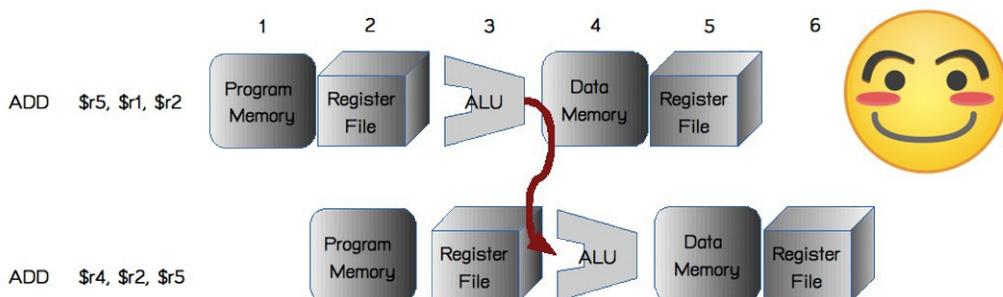
แนวทางการแก้ปัญหา Data Hazard อีกทางหนึ่งคือ การทำชาร์ดแวร์ให้สามารถแก้ปัญหาข้อมูลที่มีความเกี่ยวเนื่องจากเพื่อจะได้มีต้องแก้ไขโปรแกรม ข้อดีคือ ซอฟต์แวร์ซึ่งแต่เดิมเคยทำงานได้ถูกต้องบนหน่วยประมวลผลกลางไม่มี Pipeline สามารถนำมายังหน่วยประมวลผลรุ่นใหม่ที่มี Pipeline ได้ทันทีโดยไม่ต้องแก้ไขหรือ compile ใหม่แต่อย่างใด

การทำชาร์ดแวร์เพื่อแก้ปัญหา Data Hazard วิธีการหนึ่งคือ การออกแบบชาร์ดแวร์ให้ทำการ stall หรือหยุดการทำงานของคำสั่งก่อนหน้าที่จำเป็นต้องรอการประมวลผลให้เสร็จก่อนทำการประมวลผลต่อ (แสดงในรูปที่ 6.7) ซึ่งการแก้ปัญหาในลักษณะนี้ จะเทียบได้กับการแทรกคำสั่ง NOP เข้าไปเพื่อถ่วงเวลาเริ่มต้นการทำงานของคำสั่งถัดมา ในทำนองเดียวกันข้อเสียของการทำ stalled คือเวลาที่เสียไปอย่างไม่เกิดประโยชน์ของการประมวลผล



รูปที่ 6.7: การแก้ปัญหา data hazard ด้วย hardware stall

เพื่อไม่ให้เสียเวลาไปโดยเปล่าประโยชน์ อีกแนวทางหนึ่งในการแก้ปัญหา data hazard คือการทำ hardware forward²⁴ หลักการของการ forward คือการดึงค่าที่ทำการประมวลผลเสร็จออกมาใช้ก่อน แม้ค่านั้นจะยังไม่ถูกเขียนกลับไปก็ตาม เช่น กรณีคำสั่ง ADD \$r5, \$r1, \$r2 ตามด้วยคำสั่ง ADD \$r4, \$r2, \$r5 อยู่ติดกับคำสั่ง ซึ่งข้อมูลของ \$r5 ที่ประมวลผลเสร็จแล้วนั้น จะถูกนำไปเก็บที่ register ใน Cycle ที่ 5 แต่หากวิเคราะห์ให้ดี จะเห็นว่าการประมวลผลทำเสร็จตั้งแต่ cycle ที่ 3 นั้นหมายความว่า ชาร์ดแวร์สามารถที่จะส่งต่อค่าที่ได้จากการประมวลผล ออกมาใช้ก่อนใน Cycle ที่ 4 ได้ โดยจำเป็นไม่ต้องรอให้ข้อมูลถูกนำมาเก็บใน register \$r5 ก่อน ซึ่งชาร์ดแวร์เพื่อช่วยในการทำงานดังกล่าว (เรียกว่า forward unit) จะต้องมี multiplexor สำหรับการสลับค่าน้ำผลลัพธ์จาก ค่าในคำสั่งก่อน กลับมาป้อนให้กับ ALU อีกรอบหนึ่ง (ดังแสดงด้วยเส้นทึบในรูปที่ 6.8)



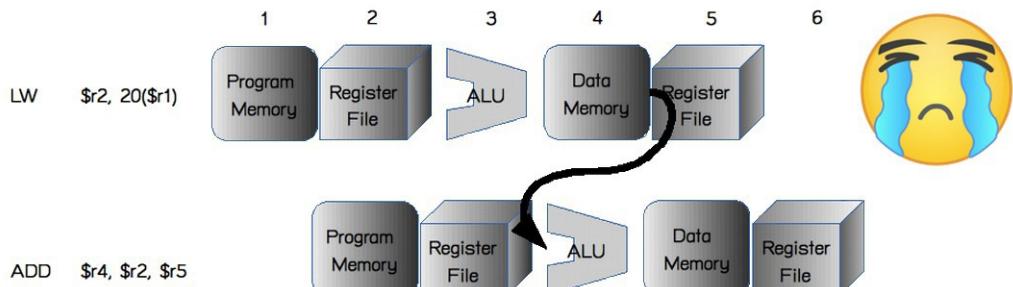
รูปที่ 6.8: ตัวอย่างการแก้ปัญหา data hazard ด้วย hardware forward

จะพบว่า ด้วย hardware forward หน่วยประมวลผลกลางไม่จำเป็นต้องเสียเวลาไปกับการหยุดรอ เพื่อนำค่าของ register \$r5 ไปใช้หลัง Cycle ที่ 5 แต่อย่างไร

อย่างไรก็ตาม มีหลายกรณีที่หน่วยประมวลผลกลางไม่สามารถทำ hardware forward ได้ ตัวอย่างที่ชัดเจนที่สุด เช่น กรณีข้อมูลที่ต้องการใช้ได้มาใน Cycle ที่ 4 จากคำสั่ง LW (แตกต่างจากคำสั่งที่ไว้มัжจะให้ผลลัพธ์ใน Cycle ที่ 3) ซึ่งกรณีคำสั่ง LW (ดูรูปที่ 6.9 ประกอบคำอธิบาย) ดังกล่าว

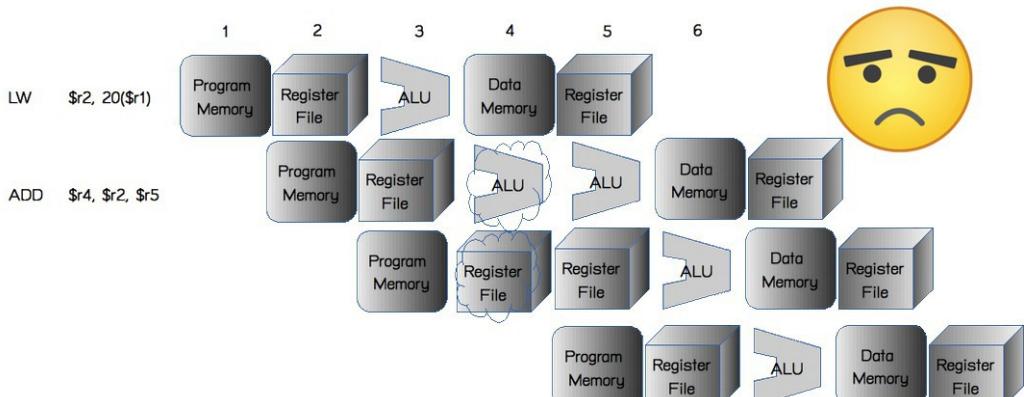
24 บางตำราจะเรียกวิธีการ hardware forward ว่า hardware bypass

จะต้องทำการอ่านค่าจากหน่วยความจำเข้ามาเก็บใน register นั่นหมายความว่า หากจะต้องทำ hardware forward จะต้อง forward จาก cycle ที่ 4 ไปยัง cycle ที่ 5 แทน (เราไม่สามารถ forward จากป้าย cycle ที่ 4 มากังตัน cycle ที่ 4 ได้ ยกเว้นมี time machine)



รูปที่ 6.9: ตัวอย่างกรณีที่ไม่สามารถ forward ได้

เนื่องจากไม่สามารถทำการแก้ปัญหา hazard ได้ด้วยวิธีการอื่น จึงเหลือเพียงวิธีการเดียวคือ การแก้ปัญหา hazard ด้วยการหยุดรอ หรือ stall รูปที่ 6.10 แสดงการทำ hardware stall²⁵ กรณีที่ไม่สามารถ forward ได้



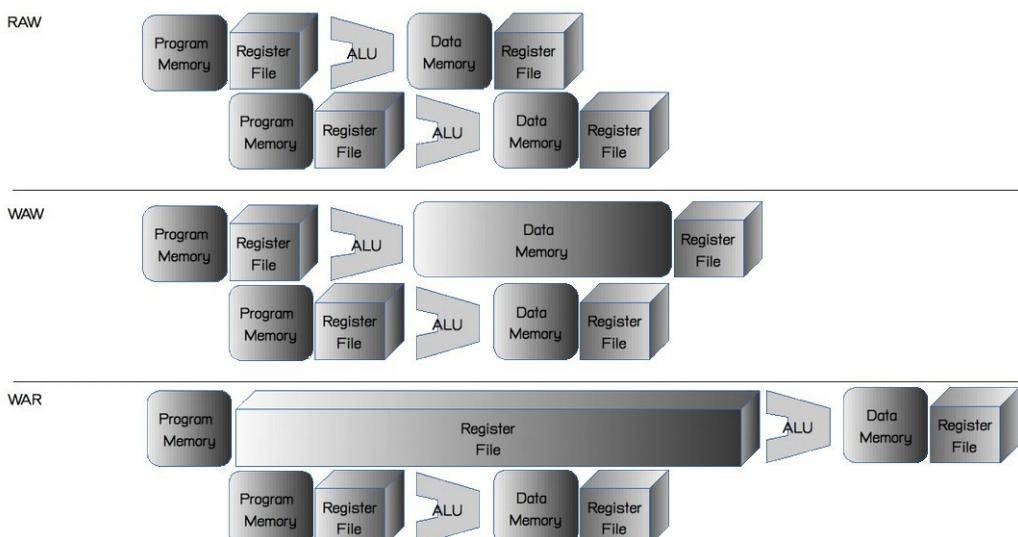
รูปที่ 6.10: การทำ hardware stall กรณีที่ไม่สามารถทำ hardware forward ได้

หากทำการวิเคราะห์ต่อจะพบว่า แม้จะมี hardware forward เพื่อช่วยแก้ปัญหา data hazard แต่การจัดลำดับของคำสั่งที่เหมาะสมสมกัยยังคงช่วยให้ไม่เกิดการ stall ของคำสั่งภายในระบบได้เช่นกัน ดังนั้น การป้องกันการเกิด Data Hazard ที่ดีที่สุดยังคงเป็นการทำงานของซอฟต์แวร์ (Compiler) หรือการทำงานร่วมกันระหว่างฮาร์ดแวร์และซอฟต์แวร์อยู่นั่นเอง

25 การทำ hardware stall ในทางปฏิบัติแนวทางหนึ่งคือ การไม่เปลี่ยนสถานะการทำงาน หรือการให้ทำ stage เดิม ช้าอีก 1 cycle

6.3.3 Data Hazard แบบ RAW, WAW, WAR

เพื่อประกอบความเข้าใจภาพรวมของ Data Hazard ทั้งหมด จะขอทำการวิเคราะห์ปัญหาต่อไปให้ละเอียดมากขึ้น (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)



รูปที่ 6.11: Data Hazard แบบ RAW, WAW และ WAR

- Read-after-Write Hazard เกิดจากค่าที่ต้องการใช้ ยังไม่พร้อมให้ใช้งาน (เช่น การใช้ค่าที่เป็นผลลัพธ์จากการคำนวณในคำสั่งก่อนหน้า) ซึ่ง hazard ลักษณะนี้อาจแก้ได้ด้วยการ forward (รูปที่ 6.8) ดังตัวอย่างที่เคยอธิบายก่อนหน้านี้
- Write-after-Write Hazard เกิดจากการที่การประมวลผลคำสั่งก่อนหน้า ทำงานเสร็จช้ากว่าคำสั่งที่ตามมา เพื่อให้เห็นภาพ ลองพิจารณา code ต่อไปนี้

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

จะเห็นว่า มีความเป็นไปได้ที่ $a \leftarrow data[1000]$ ซึ่งต้องเป็นการอ่านค่าจากหน่วยความจำ อาจจะใช้เวลาในการประมวลผลนาน (หากหน่วยความจำทำงานช้า จนส่งผลให้ $a <- 10$ ซึ่งเป็น addressing mode แบบ immediate ที่ทำงานได้รวดเร็วกว่า เสร็จก่อน หากเกิดเหตุการเขียนข้อมูล อาจทำให้ค่า a ที่ได้ผิดไป และต่อเนื่องให้ค่า b ผิดไปด้วย)

- Write-after-Read Hazard เป็น Hazard แบบที่ **พับได้น้อย** ในสถาปัตยกรรมสมัยใหม่ แต่มีพับได้ทั่วไปในสถาปัตยกรรมที่เป็น variable instruction length เพราะเวลาที่ใช้ในการ decode คำสั่งบางอัน อาจจะยาวกว่าที่ควรจะเป็น ทำให้ค่า operand ที่อ่านได้ อาจจะเป็นค่าใหม่ที่ผิดเพี้ยนไปแล้ว เพื่อให้เห็นภาพมากขึ้น ลองพิจารณา code ต่อไปนี้

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

หากการเตรียมการเพื่อทำการคำนวน $\sin(b)$ ใช้เวลามากกว่าที่ควร จะคำสั่ง $b \leftarrow 10$ ทำงานเสร็จก่อนที่จะมีการอ่านค่า b ในคำสั่งแรก จะทำให้ผลการคำนวนของคำสั่ง $a \leftarrow \sin(b)$ ดังกล่าวผิดไป

โดยการออกแบบ สามารถแก้ไข Write-after-write hazard และ Write-after-read hazard ได้ไม่ยาก กล่าวคือ กรณีของ write-after-write hazard จะต้องทำการยืนยันให้การคำนวนเสร็จตามลำดับเสมอ (in-order completion) ส่วนกรณีของ write-after-read hazard แก้ไขได้โดยการอ่านค่า operand ตั้งแต่ต้น cycle ก่อน

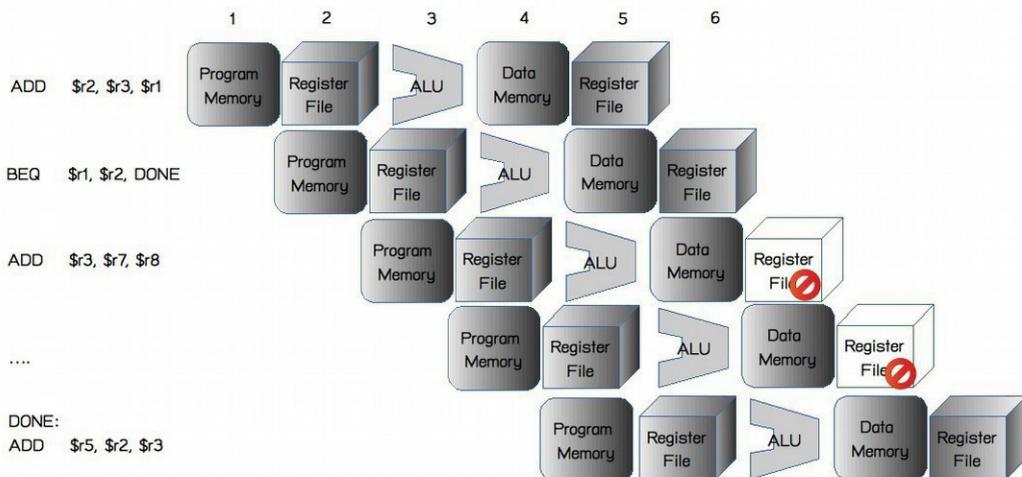
จากลักษณะของ data hazard ดังกล่าว นั่นทำให้สถาปัตยกรรมคอมพิวเตอร์สมัยใหม่ จึงมักมุ่งประเป็นของ data hazard ไปที่ read-after-write เป็นหลัก

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 หรือ การ Branch ของชอฟต์แวร์ เช่น กรณีการทำคำสั่ง BEQ จะทราบที่อยู่ของคำสั่งถัดไป (ทราบว่าจะต้อง branch หรือไม่) เมื่อได้ผ่านการประมวลผลคำสั่งนั้นไปแล้ว 3 Cycle แต่จากโครงสร้างของ Pipeline ซึ่ง จะต้องคำสั่งใหม่ในทุก cycle ทำให้การดึงคำสั่งถัดไปขึ้นมาใน Pipeline อาจเกิดปัญหาว่าเป็นคำสั่งที่ไม่ต้องการใช้งาน (เกิดการ branch)

การแก้ปัญหานี้ สามารถกระทำได้หลายแนวทาง เช่น การสร้างวงจรพิเศษ เพื่อประมวลผลคำสั่งประเภท Branch ให้เสร็จสิ้นภายใน 1 Cycle เพื่อ Pipeline จะได้เริ่มคำสั่งถัดไปที่ถูกต้องขึ้นมาทำงานได้ทันที ซึ่งกรณีนี้ทำให้ต้องออกแบบวงจรที่ซับซ้อน ดังนั้นทางเลือกที่ง่ายกว่า คือ การล้างคำสั่งที่ไม่ต้องการออกจาก Pipeline ให้หมดแล้วค่อยดึงคำสั่งใหม่ขึ้นมา (Flushing) โดยวิธีนี้สามารถทำได้ยากกว่าโดยการเพิ่มเพียงวงจร Flushing เข้าไปเล็กน้อยเท่านั้น อีกทางเลือกหนึ่งที่คล้ายกันคือ การไม่เปลี่ยนค่าคืนไปยัง register (ไม่ทำขั้นตอน write back) ซึ่งผลที่เกิดขึ้นจะเหมือนกับว่าคำสั่งนั้น จะไม่ถูกทำงาน



รูปที่ 6.12: การสร้าง bypass ขึ้นตอนการ write back เพื่อแก้ปัญหา control hazard

วิถีแนวทางในการลดผลกระทบของ Control Hazard ซึ่งทำได้ไม่ยากคือ การเดา (branch prediction) หลักการง่ายๆ คือ หากสามารถเดาได้ว่าจะเกิดการ branch หรือไม่ และเดาได้ถูก จะทำให้ไม่เกิด cycle ของคำสั่งที่ไม่เกิดประโยชน์ (ไม่ต้องทำการ flush) ทั้งนี้การทำงานของ branch prediction ฯ ลฯ hardware ที่เกี่ยวข้องอยู่นอกเหนือขอบเขตของหนังสือเล่มนี้ แต่หลักการทำงานเบื้องต้นคือ (1) เดาว่า branch เสมอ (2) เดาว่าไม่ branch เสมอ หรือ ซับช้อนยิ่งขึ้นด้วย (3) การเก็บสถิติการ branch ล่าสุดไว้ หากครั้งล่าสุดมีการ branch ให้เดาว่าครั้งต่อไปมีการ branch เป็นตัน (ยังมีการทำงานของ branch prediction แบบที่ซับช้อนกว่านี้ ซึ่งมีได้ก็แล้วแต่เงื่อนไข)

ข้อสังเกตคือ ไม่ว่าจะแก้ปัญหา control hazard ด้วยแนวทางใด ก็มีแนวโน้มที่จะเกิด cycle ที่เสียเปล่าไม่เกิดประโยชน์ได้โดยได้ จึงมีความพยายามที่จะให้เทคนิคทางซอฟต์แวร์ช่วย เพื่อแก้ปัญหา stall cycle นี้ ดังจะกล่าวถึงในตอนต่อไป

6.4.1 การแก้ปัญหา Control Hazard ด้วยวิธีการ Branch Delay Slot

ถึงตรงนี้ ข้อสังเกตคือ การแก้ปัญหา Control Hazard ไม่ว่าจะแบบใดก็ตามที่ได้ก้าวมาแล้วคือ จะมีการตั้งคำสั่งที่อาจจะผิดขึ้นมา ทำให้อาจต้องมี cycle ที่ไม่เกิดประโยชน์เกิดขึ้นอยู่ดี ดังนั้นจึงมีการเสนอแนวทางการแก้ปัญหา Control Hazard โดยใช้วิธีการทางซอฟต์แวร์เข้าร่วมเรียกว่า Branch Delay Slot หรือ Delayed Branch

เนื่องจากทุกครั้งที่มีการ branch จะไม่ทราบว่าจะต้อง branch หรือไม่จนกว่าจะผ่านไปถึง cycle ที่ 3 นั้นหมายความว่า คำสั่งเด็กตามที่ถูกดึงขึ้นมา ก่อนจะทราบผลการ branch ต่างก็มีแนวโน้มที่จะเสียเปล่า สมมุติว่า ที่ cycle ที่ 3 หลังจากการทำ branch completion แล้วสามารถทำการตั้งคำสั่งใหม่ (fetch) ที่ถูกต้องได้ทันที ดังนั้น หากสถาปัตยกรรมชุดคำสั่ง ทำการปรับให้ cycle ถัดมาของคำสั่ง

branch ไม่เข้ากับการ branch โดยทำการหน่วงการ branch ออกไป 1 cycle และให้ cycle ที่ตามมาเป็นการทำงานของคำสั่งปกติ จะได้ว่า ไม่ว่าจะมีการ branch เกิดขึ้นหรือไม่ cycle ที่ตามมา จะเป็นการทำงานของคำสั่งที่ต้องมีการทำงานตามปกติเสมอ

เพื่อประกอบความเข้าใจเรื่อง 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)
```

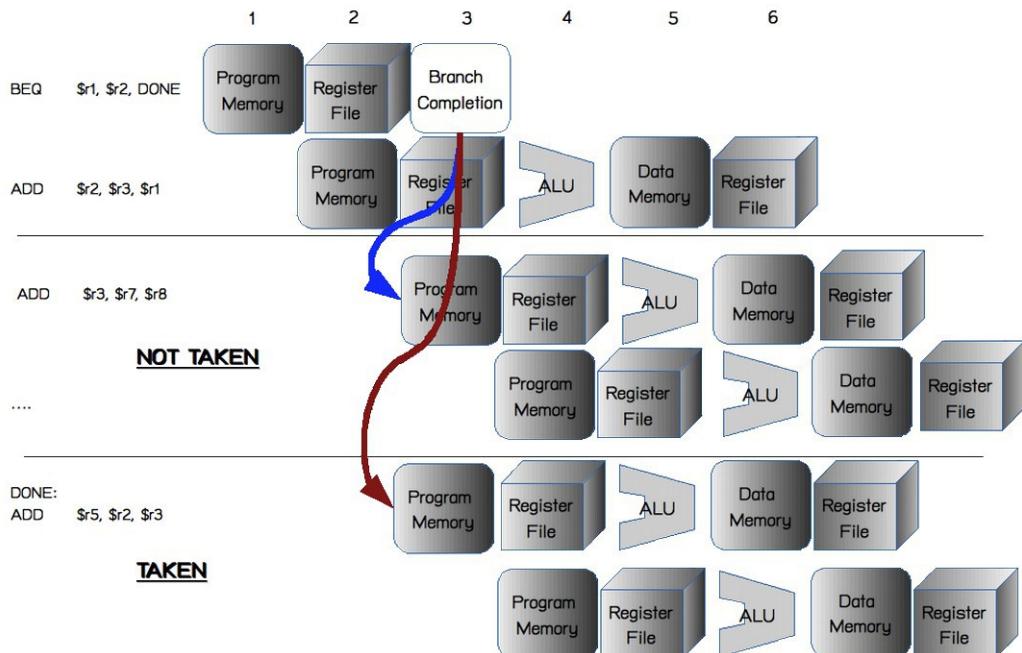
จาก code หากเป็นการทำงานของคำสั่งทั่วไป กรณีที่ BRANCH จะทำงานคำสั่งที่ (9) และกรณีที่ไม่ branch จะต้องทำงานคำสั่งที่ (3) ในกรณีที่มี delayed branch คำสั่งที่ตามหลังมา จะไม่มีผลเกี่ยวกับ branch ดังนั้น หากทำการ reschedule code ดังกล่าวสำหรับองรับการทำงาน delayed branch จะได้เป็น

```
(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 ผังนี้การย้ายคำสั่ง (1) มาไว้หลัง (2) ทำให้ผลที่ได้คือ ไม่มีการ stall เกิดขึ้นอย่างแน่นอน การทำงานของ delayed branch สามารถอธิบายเป็นแผนภาพได้ดังรูปที่ 6.13

จากรูปที่ 6.13 จะเห็นว่าไม่ว่าจะมีการ branch หรือไม่ คำสั่งที่ตามมา จะมีการประมวลผลเสมอ และผลของ branch not taken จะถูก delayed ออกไป 1 คำสั่ง (ซึ่งเป็นที่มาของชื่อ branch delayed slot) และกรณี branch taken ก็สามารถอ่านคำสั่งที่เกิดจาก branch ขึ้นมาได้ทันทีเช่นกัน

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

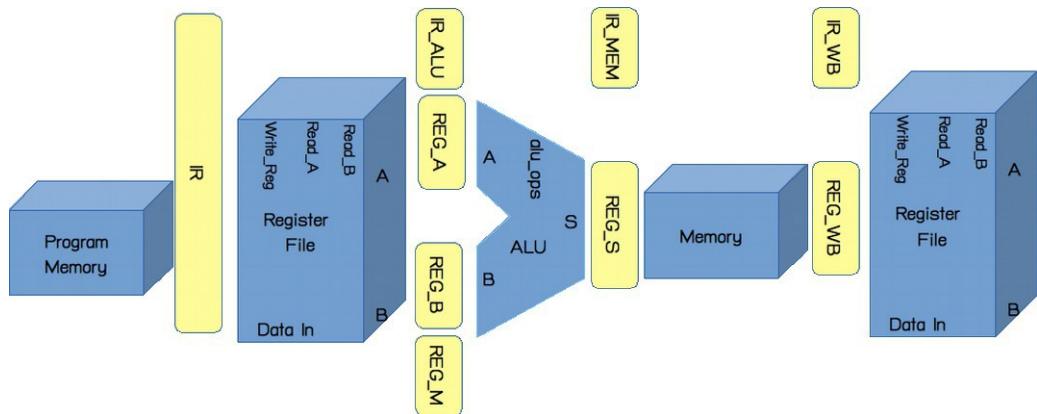


รูปที่ 6.13: การแก้ปัญหา control hazard ด้วย branch delayed slot (แบบ taken) และ (not taken)

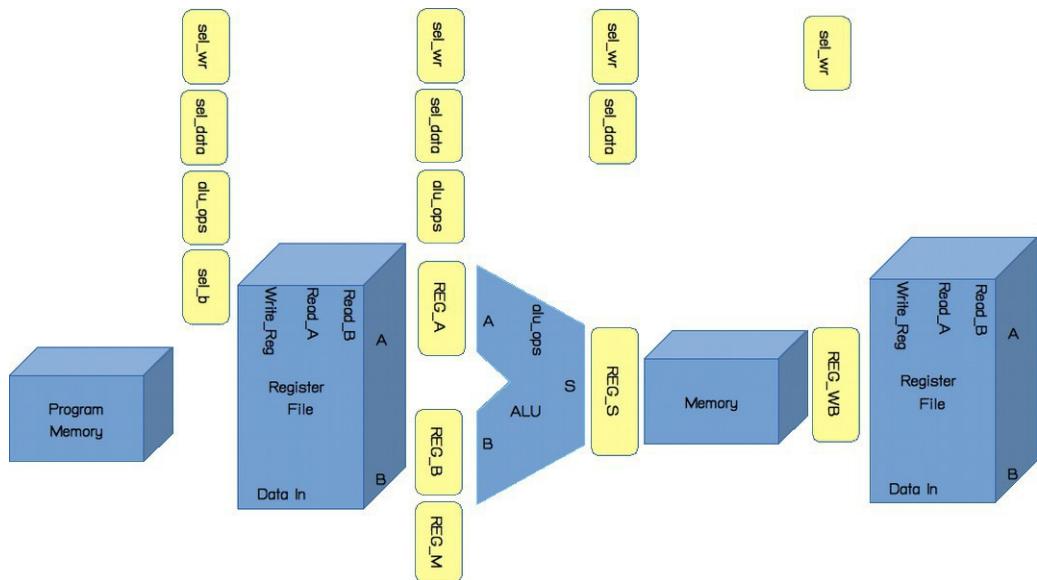
6.5 Data Path

ปัจจัยแรกที่จะทำให้สามารถเริ่มต้นคำสั่งใหม่ได้ทุก cycle คือ การแยกให้แต่ละcycle มีความสามารถทำงานได้เป็นอิสระจากกัน มีคำสั่งที่เกี่ยวข้องเป็นของตนเอง ทำให้โครงสร้างภายในของ CPU ที่ จะ ทำPipeline นั้นมีส่วนประกอบที่มากขึ้นและบางส่วนอาจทำงานซ้ำซ้อนกัน (ซึ่งแนวทางการใช้อุปกรณ์ที่ซ้ำซ้อนกันนี้ จะขัดแย้งกับแนวทางในการออกแบบหน่วยประมวลผลคลาสสิกแบบ Multi-cycle ในบทที่ 5 บางส่วนซึ่งเป็นการลดอุปกรณ์ซ้ำซ้อนเพื่อให้ CPU มีขนาดเล็ก) ทั้งนี้เนื่องจาก อุปกรณ์ทุกชิ้นต้องทำงานพร้อมกัน ดังแสดงในรูปที่ 6.14

เพื่อให้การทำงานแต่ละขั้นเป็นอิสระจากกัน วิธีการที่ง่ายที่สุดคือการใส่ buffer หรือ register พิเศษ เพื่อช่วยส่งต่อสัญญาณที่ต้องใช้ต่อไป จากรูปที่ 6.14 จะเห็นว่ามีการเพิ่ม register คำสั่ง (IR_ALU, IR_MEM, IR_WB) เพื่อแยกกว่าคำสั่งที่ทำงานในต่อๆ กันนี้ต้องของ pipeline นั้น เป็นอย่างไรสั่งอะไร (ในทำนองเดียวกัน เราอาจจะส่งต่อสัญญาณควบคุมแทนก์ได้ เพื่อแสดงว่า สัญญาณควบคุมที่ต้องใช้ ในขั้นตอนดังกล่าว และขั้นตอนถัดไปเป็นอย่างไรบ้าง ดังแสดงในรูปที่ 6.15)



รูปที่ 6.14: การแยก register คำสั่งแยกออกจากกันในแต่ละขั้นของ pipeline



รูปที่ 6.15: การส่งผ่านสัญญาณควบคุมสำหรับ pipeline

เนื้อหาในส่วนนี้จะไม่ขอลงรายละเอียดเกี่ยวกับตัวสายสัญญาณที่เกิดขึ้นจริง เพราะภาพที่ได้จะซับซ้อนจนดูเข้าใจยาก แต่จะขอทิ้งแนวคิดบางส่วนไว้เป็นแบบฝึกหัดท้ายบทแทน

6.6 สรุป Pipeline

การทำ Pipeline เป็นการเพิ่ม Throughput ให้กับหน่วยประมวลผลกลางที่สามารถทำได้เร็วเพื่อมีหน่วยต่างๆ 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 Predictive, Branch Delay Slot หรือ การทำ Dynamic Schedule เป็นต้น นอกจากนี้ ยังมีปัจจัยหนึ่งที่ไม่ได้กล่าวถึงในหนังสือเรื่องของ Exception ซึ่งจะทำให้การ pipeline ยุ่งยากมากขึ้น

แนวทางหนึ่งซึ่งเป็นที่นิยมใช้ในการแก้ปัญหา hazard ของสถาปัตยกรรมชุดคำสั่งสมัยใหม่คือ การทำให้ปัญหาดังกล่าวมองเห็นได้ด้วยซอฟต์แวร์ (หรือ compiler) และบังคับให้ซอฟต์แวร์ (compiler) จัดลำดับการทำงานของคำสั่งเพื่อลีกิคเลี่ยงการเกิด hazard แทน วิธีการนี้ ทำให้ฮาร์ดแวร์ที่ไม่มีความซับซ้อนลดลง แต่เพิ่มเงื่อนไขให้กับ compiler หรือผู้พัฒนาซอฟต์แวร์แทน

6.6.1 ประสิทธิภาพของ Pipeline ในทางปฏิบัติ

เนื่องจากในทางปฏิบัติ Pipeline จะต้องมีการ stall เนื่องจากปัญหา Hazard แบบต่างๆ อยู่บ้าง ดังนั้นหากทำการวิเคราะห์สมการที่ 6.1 ใหม่ เราจะได้ว่า ประสิทธิภาพของ Pipeline ที่แท้จริงจะต้องมีการบวกค่า stall cycle เข้าไปด้วย

$$CPU\ TIME_{pipeline} = ([no.\ of\ pipeline + (no.\ of\ instructions \times CPI)] + stall\ cycle) \times cycle\ time$$

อย่างไรก็ตามหลายตำราอนิยมที่จะคิด stall cycle เข้าเป็นส่วนหนึ่งของ CPI โดยการหาค่า CPI เฉลี่ย ซึ่งเฉลี่ยรวมกรณีที่มีการ stall เข้าไปด้วย และมักจะค่าคงที่ (no. of pipeline) ซึ่งเป็น cycle เริ่มต้นที่ใช้สำหรับการเติม pipeline ให้เต็มตอนเริ่ม จึงทำให้ยังได้สมการ CPU Time เมื่อันเดิมแบบสมการที่ 2.8

6.7 แบบฝึกหัดท้ายบท

1. การทำ Pipeline ช่วยในการเพิ่ม Throughput ได้อย่างไร
2. จadge แผนภาพแสดงการทำงานของ Pipeline (แบบรูปที่ 6.1) เมื่อมีจำนวนคำสั่งเป็น 10 คำสั่ง และทุกคำสั่งมีการทำงาน 5 cycle เท่ากัน
3. จากข้อ 2 จadge ความสัมพันธ์ระหว่างจำนวน cycle ที่ใช้และจำนวนคำสั่ง
4. จadge บายปัญหาซึ่งเป็นอุปสรรคต่อการทำ Pipeline และเสนอแนวทางแก้ไข
5. เหตุใดคำสั่ง SW, BEQ, และ JMP จึงไม่เกิดปัญหา structural hazard แบบคำสั่งคำสั่ง ADD, ORI และ ORUI จadge แผนภาพและให้เหตุผลประกอบการอธิบาย
6. จากภาพแสดงการทำงานของ Pipeline ดังกล่าว จadge แผนภาพการทำงานของ Pipeline เพื่อประมวลผลคำสั่งต่อไปนี้ พร้อมทั้งแสดงการแรงงานที่ถูกต้อง

ADD	\$10, \$0, \$1
LW	\$12, 20 (\$5)
ADD	\$3, \$12, \$12
SW	\$3, 30 (\$10)
ADD	\$10, \$10, \$1

7. จากภาพที่ว่าด้วยข้อ 2 หากหน่วยประมวลผลกลางดังกล่าวมี Hazard ซึ่งสามารถแก้ได้ด้วยการ Stall และ Forward จadge เส้นแสดงตำแหน่งที่ข้อมูลมีการอ้างอิงต่อ กัน (Dependency) และแสดงการเรียงคำสั่งใหม่เพื่อทำการแก้ไขปัญหา Data Hazard ด้วยซอฟต์แวร์
8. เหตุใดการทำงานของหน่วยประมวลผลกลางแบบไม่มี Pipeline จึงไม่เกิดปัญหา Data Hazard จadge อธิบาย
9. จadge การสร้าง Data Path ของสถาปัตยกรรมชุดคำสั่ง nanoLADA เพื่อให้รองรับการทำงานแบบ Pipeline ได้
10. จadge สถาปัตยกรรมชุดคำสั่ง nanoLADA ที่กำหนด จadge การเปรียบเทียบค่า CPI และ Clock Cycle Time ของหน่วยประมวลผลกลางแบบ Single Cycle และ Multiple Cycle และแบบ Pipeline พร้อมทั้งวิเคราะห์ Speed up สูงสุดที่เป็นไปได้
11. จadge บายปัญหา WAR และ WAW พร้อมเขียนแผนภาพประกอบคำอธิบาย

7 การจัดการหน่วยความจำ

ปัญหาหลักของระบบหน่วยความจำ แบ่งออกได้เป็นสองลักษณะคือ 1) ปัญหาระบบประสิทธิภาพ หรือ ความเร็ว และ 2) ปัญหาระบบขนาดของหน่วยความจำที่สามารถใช้งานได้ ซึ่งการกระทำเพื่อปรับปรุงอย่างโดยย่างหนักจะส่องอย่างนี้ มักส่งผลเสียต่อคุณสมบัติอีกด้านหนึ่งเสมอ กล่าวคือ หน่วยความจำขนาดใหญ่จะมีการทำงานที่ช้า ในขณะที่หน่วยความจำที่มีขนาดเล็กจะมีการทำงานที่รวดเร็ว หากจะทำให้ระบบความคิดถูกขึ้นมาอีกนิด จะมีปัญหาเพิ่มอีกเรื่องหนึ่ง คือ 3) ปัญหาระบบราคา แน่นอนว่า หน่วยความจำที่มีความเร็วสูงมักจะมีราคาแพงตามไปด้วย (ทำให้มีได้น้อย) ส่วนหน่วยความจำที่มีความเร็วต่ำ มักมีราคาถูก (ทำให้มีได้เยอะ)

เรื่องความเร็วของหน่วยความจำ และ ขนาดความจุนั้น มีคำอธิบายง่ายๆ ตามหลักการทำงานฟิสิกส์ทั่วไป คือ หน่วยความจำขนาดใหญ่เปรียบเสมือนเพื่อที่ขนาดใหญ่ การจะหาของหรือจัดเก็บข้อมูลในห้องขนาดใหญ่ ย่อมต้องใช้เวลามากกว่าการหาของหรือจัดเก็บข้อมูลบนได้ขนาดเล็ก สำหรับหน่วยความจำถูกเชื่อมเดียวกัน

จากความไม่สอดคล้องกันของความเร็วและขนาด ความยากและท้าทายของเนื้อหาในส่วนนี้คือ การทำให้รู้สึกเหมือนกับว่ามีหน่วยความจำที่ มีความเร็วสูง มีขนาดใหญ่ และ ราคาไม่แพง ผู้เรียนอาจจะรู้สึกว่าเป็นการท้าทายที่ดูจะโลภมาก แต่หากวิเคราะห์ให้ดี บทเรียนที่ผ่านมาก่อนหน้านี้ ล้วนแต่มองว่า หน่วยความจำมี 4Gb (อ้างอิงได้ด้วย 32 บิต) และ มีความเร็วสูง (อ่านหรือเขียนได้ใน 1 cycle)

ในทางสถาปัตยกรรม มีแนวทางการแก้ปัญหาระบบหน่วยความจำโดยใช้ 2 แนวทางร่วมกันคือ (1) **การใช้ระบบ Cache** ซึ่งเป็นการนำหลักการของ locality มาใช้เพื่อให้รู้สึกว่ามีหน่วยความจำที่เร็ว และ(2) **การใช้ระบบหน่วยความจำเสมือน** เพื่อให้รู้สึกเสมือนว่ามีหน่วยความจำอยู่เยอะ กล่าวคือ ระบบ Cache เป็นการเพิ่มประสิทธิภาพในการเข้าหนึ่งหน่วยความจำให้กับหน่วยประมวลผลกลาง (โดยเพิ่มความเร็วในการอ่านข้อมูลจากหน่วยความจำ ส่วนหน่วยความจำเสมือนนั้นช่วยให้การอ้างอิงหน่วยความจำของผู้พัฒนาซอฟต์แวร์ และ ระบบhardtware เป็นอิสระจากกัน ทำให้ผู้พัฒนาซอฟต์แวร์รู้สึกอยู่เสมอกับว่ามีหน่วยความจำอยู่ 4 Gb ตลอดเวลา

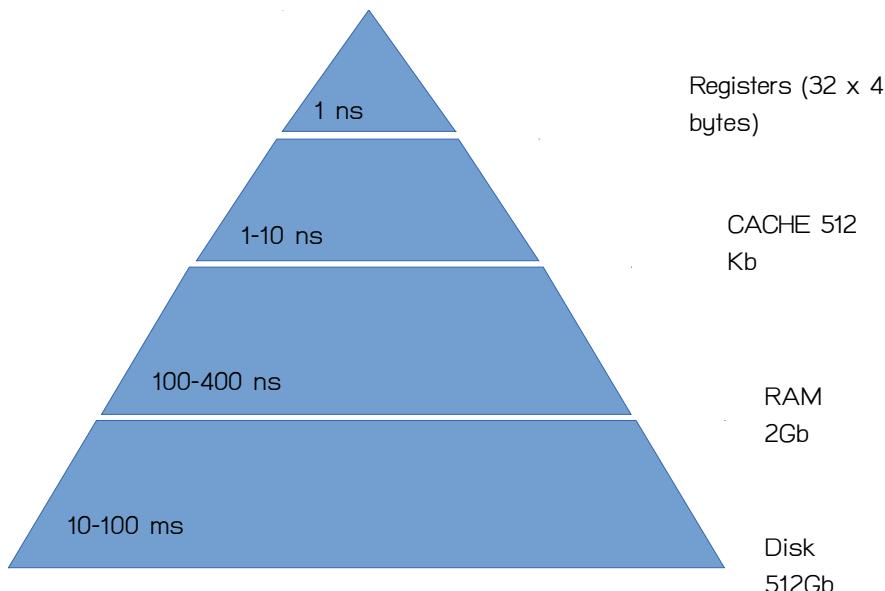
เนื้อหานี้จะเน้นที่พื้นฐานการจัดการระบบ Cache และ ระบบหน่วยความจำเสมือนแบบ Paging เท่านั้น อย่างไรก็ตามยังมีระบบหน่วยความจำเสมือนอีก (เช่น ระบบ Segmentation) ซึ่งมีได้เน้นในที่นี้ ดังนั้นหากผู้อ่านต้องการศึกษาเรื่อง segmentation ควรจะศึกษาเพิ่มเติมจากเอกสารหรือตำราอื่นด้วย

7.1 Cache

Cache นั้นเป็นการเพิ่มประสิทธิภาพในการอ่านและเขียนข้อมูลจากหน่วยประมวลผลกลางให้สามารถกระทำได้รวดเร็วขึ้น โดยอาศัยหลักการของ Locality (ซึ่งจะอธิบายต่อไป) คือ การทำให้ข้อมูลที่ต้องการอ่านเป็นประจำสามารถอ่านได้อย่างรวดเร็ว ด้วยการนำสิ่งที่จำเป็นจะต้องใช้งานบ่อยๆ มาไว้ใกล้ตัว เช่น กรณีที่จำเป็นจะต้องใช้ปากกาบ่อยๆ หากปากการวางอยู่ในที่ซึ่งเราหยิบได้ง่าย การ

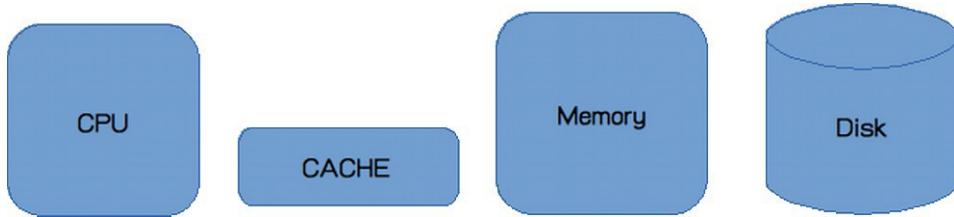
ทำงานของเรารักษาระบบทามได้รวดเร็ว ซึ่งหากปากอยู่ใกล้ตัว (เข่นอยู่ในกระเพาเอกสาร) จะทำให้การทำงานมีประสิทธิภาพด้วยลง เพราะต้องเสียเวลา กับการหยิบปากงานนั้น ในหน่วยประมวลผลกลางก็เช่นเดียวกัน หากการดึงข้อมูลจากหน่วยความจำ (ไม่ว่าจะเป็นการดึงข้อมูลจากหน่วยความจำโปรแกรมหรือหน่วยความจำข้อมูลก์ตาม) สามารถกระทำได้รวดเร็ว อาจจะช่วยให้หน่วยประมวลผลกลางสามารถทำงานได้ที่ Clock Rate ที่สูงขึ้น

ทั้งนี้ เนื่องจากหน่วยความจำที่มีความเร็วสูงนั้นมีราคาแพง ห้าระบบคอมพิวเตอร์ทั้งหมดทำงานด้วยหน่วยความจำที่มีความเร็วสูงทั้งหมดนั้น จะให้เครื่องคอมพิวเตอร์มีราคาสูงมาก แต่ถ้าหากไม่มีหน่วยความจำความเร็วสูงเหล่านี้อยู่เลย ก็จะทำให้การประมวลผลนั้นข้ามจากหน่วยความจำไปยังหน่วยความเร็วต่ำๆ เช่น RAM หรือ Hard disk ที่มีความเร็วต่ำกว่า แต่ถ้าหากมีหน่วยความจำที่มีความเร็วสูง และ ความเร็วต่ำๆ สมกันไป เป็นระดับขั้นในปริมาณที่แตกต่างกันไป หน่วยความจำประเภทใดทำงานช้ามักจะราคาถูก ระบบก็จะมีได้ในปริมาณที่สูง ในขณะที่หน่วยความจำประเภทที่มีความเร็วสูงราคาก็สูงเช่นกัน ก็จะมีในปริมาณน้อย ซึ่ง Cache อันเป็นหน่วยความจำความเร็วสูงอาจจะมีเพียง 512 Kb ส่วนหน่วยความจำหลัก (Ram) อาจจะมีมากถึง 2Gb ॥ และ Hard disk ซึ่งราคาต่อหน่วยค่อนข้างต่ำอาจจะมีมากถึง 512 Gb เป็นต้น ความเร็ว และปริมาณหน่วยจัดเก็บข้อมูลที่มีในระบบคอมพิวเตอร์ แสดงได้ดังรูปที่ 7.1



รูปที่ 7.1: ความเร็วและปริมาณหน่วยจัดเก็บข้อมูลที่มีในระบบคอมพิวเตอร์

ในหน่วยประมวลผลสมัยใหม่ ยังแบ่งการทำงานของ CACHE ออกเป็นหลายระดับชั้น โดยเรียกระดับที่อยู่ใกล้ Register มากที่สุดว่า Level 0 (ซึ่งมักอยู่ในหน่วยประมวลผลกลาง) และเรียกลำดับถัดไปว่า Level 1, Level 2 ออกไปเรื่อยๆ จนถึง RAM ซึ่งในการอ้างอิงข้อมูลในหน่วยความจำ จะอ้างอิงโดยเริ่มจากการหาข้อมูลใน Cache ระดับที่ใกล้กับหน่วยประมวลผลกลางก่อน แล้วจึงไล่ตามลำดับชั้นออกไป (รูปที่ 7.2)



รูปที่ 7.2: ลำดับการเข้าถึงข้อมูล

อย่างไรก็ตาม การมีหน่วยความจำที่มีความเร็วสูงนั้น อาจจะไม่ทำให้ประสิทธิภาพในการอ่านและเขียนข้อมูลเร็วขึ้นเสมอไป หากขนาดระบบการจัดการที่เหมาะสม ตัวอย่างเช่น หากมีตัวอยู่ 1 ตัวขนาด 1×1 ตารางเมตร (ซึ่งเปรียบเทียบได้กับ Cache) และข้อมูลที่ต้องทำงานด้วยเป็นกล่องขนาด 1×1 ตารางเมตร จำนวนหลายกล่อง (เปรียบเทียบได้กับหน่วยความจำหลัก) ซึ่งหากเราทำงานของเราต้องเกี่ยวข้องกับกล่องหลักอัน (และตามหลักของ Cache นั้น เราจะต้องทำงานข้อมูลที่เราต้องการใช้งานอยู่ใน Cache ก่อนจึงจะเริ่มอ่านและเขียนข้อมูล) หากวิเคราะห์ให้ดี จะพบว่า เวลาที่เสียในการยกกล่องขึ้นมาวางและออกจากโต๊ะอาจมากกว่าการเดินไปทำงานที่กล่องเหล่านั้นโดยตรง (เช่น กรณีการทำงานของเราต้องนำของจากกล่องที่ 1 มาเก็บในกล่องที่ 2 และนำของจากกล่องที่ 2 มาเก็บในกล่องที่ 1 ตามลำดับ จะพบว่าเมื่อเราต้องการทำงานกับกล่องที่ 1 นั้น กล่องใบที่ 1 ก็ยังไม่อยู่บนโต๊ะ เราต้องเสียเวลายกกล่องใบที่ 1 ขึ้นมาวางบนโต๊ะก่อน และเมื่อเราทำงานกับกล่องที่ 1 เสร็จแล้วต้องการทำงานกับกล่องใบที่ 2 ก็ต้องเสียเวลายกกล่องใบที่ 1 ลง และยกกล่องใบที่ 2 ขึ้นมาวางแทน จนเมื่อทำงานกับกล่องใบที่ 2 เสร็จ ก็ต้องยกกล่องใบที่ 2 ออก และยกกล่องใบที่ 1 ขึ้นมาวางแทน เป็นต้น) ดังนั้นการมี Cache จึงต้องมีเนื้อนodataที่เหมาะสม และมีระบบการจัดการที่ต้องควบคู่กันไปด้วย

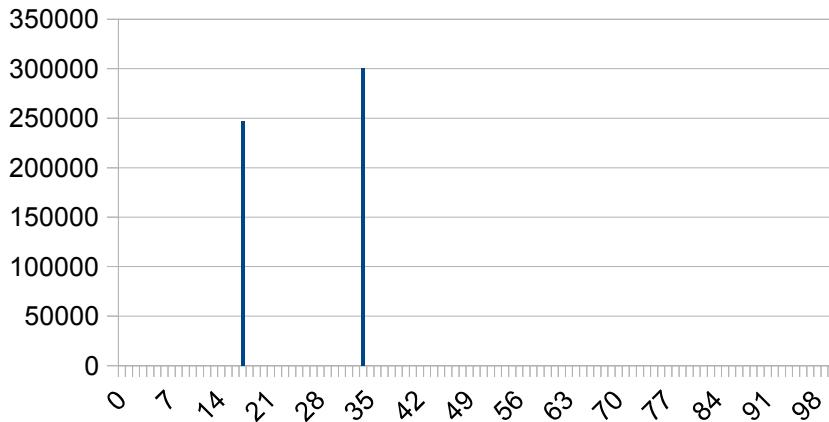
เพื่อประกอบความเข้าใจเรื่องการบริหารจัดการ Cache ที่ดี ก่อนอื่นจะขออธิบายดึงหลักการทำงานของหลักการท้องถิ่น (Locality) ก่อน เพื่อจะได้เป็นข้อมูลประกอบการออกแบบและการบริหารจัดการ CACHE ต่อไป

7.2 หลักการท้องถิ่น (Locality)

หลักการท้องถิ่น หรือ Locality คือการทำให้สิ่งที่น่าจะได้ใช้หรือถูกใช้งานบ่อยๆ มาอยู่ใกล้ตัว ความจริงแล้ว ในชีวิตประจำวันมักพบเห็นการใช้งานของหลักการนี้อยู่ทั่วไป เช่น ผู้ที่ต้องเขียนหนังสือหรือใช้ปากกาบ่อยๆ มักพกปากกาไว้ที่กระเป๋าเสื้อ เพื่อช่วยให้หยิบจับมาเขียนได้อย่างรวดเร็ว หรือ สมุดและหนังสือที่ต้องใช้บ่อย มักวางอยู่บนโต๊ะทำงาน เพื่อให้เข้าถึงข้อมูลได้อย่างทันทีทันใด เป็นต้น ดังนั้น คงจะไม่ต้องขยายความเรื่องความหมายของ Locality มากนัก เพราะว่าเป็นเรื่องใกล้ตัว

ในซอฟต์แวร์คอมพิวเตอร์ทั่วไป หากนำค่าของ address ที่มีการอ้างอิงในโปรแกรมมาทำการวิเคราะห์ จะได้ว่า ในโปรแกรมหนึ่งหนึ่ง มักจะมีช่วงของ address ที่มีการใช้งานสูงอยู่เพียงส่วนหนึ่งเท่านั้น รูปที่ 7.3 เป็นการนำ address ของโปรแกรม gcc จากการ compile โปรแกรมขนาดใหญ่มาก ทำการวิเคราะห์ และ normalize ให้อยู่ในช่วง 0-99 จะเห็นว่าโปรแกรม gcc ใช้เวลาในการทำงานส่วนใหญ่อยู่เพียง 2 ช่วง address เท่านั้น (แทน Y คือ ค่าความถี่) ดังนั้น หากต้องการให้โปรแกรม

นี้ทำงานอย่างรวดเร็ว ประสิทธิภาพสูง เพียงแค่จัดหา Locality ที่มีข่านาดใหญ่พอสำหรับเก็บข้อมูลในสองช่วงนี้ไว้ให้เข้าถึงได้อย่างรวดเร็ว ก็พอแล้ว ไม่จำเป็นต้องมีหน่วยความจำความเร็วสูง ราคาแพงมากเกินความจำเป็น



รูปที่ 7.3: ความถี่การอ้างอิงช่วง address ของ GCC

ถึงแม้ในรูปจะเป็นเพียงตัวอย่างจาก gcc ตาม หากทำการทดลอง หรือสังเกตุเพิ่มเติมจะพบว่า ซอฟต์แวร์ทั่วไป ก็มักจะมีการอ้างอิงถึงข้อมูลทั้งสองรูปแบบนี้เสมอ เช่น พึ่งมีการใช้ตัวแปร a มักจะมีแนวโน้มว่า จะต้องอ้างอิงถึงตัวแปร a อีกในอนาคตอันใกล้ หรือกรณีเป็นข้อมูลแบบ array เมื่ออ้างอิง b[10] แนวโน้มคือ อาจจะอ้างอิง b[11] ต่อไป ด้วยเหตุนี้การบริหารจัดการ Cache ที่ดี จะช่วยให้ข้อมูลที่มีการอ้างอิงทั้งสองรูปแบบสามารถเข้าถึงได้อย่างมีประสิทธิภาพ

อย่างไร้ตาม คำจำกัดความของหลักการท้องถิ่น หรือ Locality ได้ทำการแบบหลักการท้องถิ่นออกเป็น 2 แบบย่อยคือ

1. **หลักการท้องถิ่นเชิงเวลา (Temporal Locality)** กล่าวคือ สิ่งที่ถูกใช้งานบ่อยในช่วงเวลาล่าสุด ควรจะอยู่ใน Locality เช่น ในโทรศัพท์มือถือมักแสดงเบอร์โทรศัพท์ที่พึ่งมีการโทรใช้งานล่าสุด เพื่อให้สามารถโทรใหม่ได้ง่าย นี้เป็นตัวอย่างหนึ่งของ Temporal Locality
2. **หลักการท้องถิ่นเชิงพื้นฐาน (Spatial Locality)** กล่าวคือ ของบางอย่างมักมีการใช้งานที่เกี่ยวกัน เช่น ดินสอมักถูกใช้งานคู่กับยางลบ ดังนั้น หากจะหยิบดินสอขึ้นมาวางบนโต๊ะ ก็ควรที่จะหยิบยางลบขึ้นมาด้วย เป็นต้น

ดังนั้นการออกแบบ Cache ที่ดี จะต้องคำนึงถึงคุณสมบัติของหลักการท้องถิ่นทั้ง 2 ประการนี้ด้วย เช่นกัน

ประเด็นช่วยจำเวลาผ่านหนังสือดึงเรื่องนี้ ผสมชอบยกตัวอย่างเพื่อประกอบความเข้าใจเรื่อง Temporal Locality และ Spatial Locality ดังนี้ ตัวอย่าง Temporal Locality คือ หากขายหนุ่มพึ่งโทรศัพท์ติดต่อหันยิงสาว หมายความว่า เค้าอาจสนใจในตัวເຮືອ ດັ່ງນັ້ນควรวางแผนเบอร์โทรศัพท์ອອນເຮືອໄວ້ໃລ້ຖ້າ ຈະໄດ້ເຂົ້າສິ້ນໄດ້ຫຍີບໃຫ້ໄດ້ง່າຍ ແຕ່หากการທີ່ขายหนุ่ມໂທຣັກສັບທີ່ຕິດຕ່ອງຫຼູງສາວ ເພື່ອຈະເປັນທາງຜ່ານໄປຢັງເພື່ອນສາວອອນເຮືອ ການທີ່ຫຍີບເບົອຮັບອອນເພື່ອສາວທີ່ขายหน່າມໝາຍປອງເຂົ້າມາດ້ວຍ ເພື່ອຈະໄດ້ຫຍີບໂທຣັກສັບທໍາໄດ້ງ່າຍ

7.3 การจัดการ Cache เป็นต้น

เพื่อประกอบการอธิบาย ในระบบ Cache ຈຶ່ງໄດ້ມີການກຳຫັດຄໍາຕົ້ນເພື່ອໃຊ້ອົບາຍລັກຂະນະຂອງ Cache ແລະການອ່ານເຫັນຂໍ້ມູນ ດັ່ງນີ້

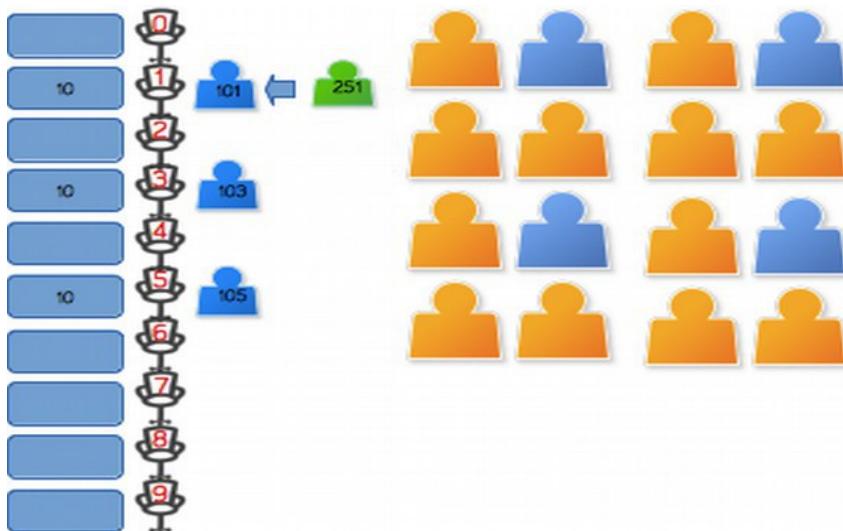
- **Block** คือ การแบ่งขนาดของหน่วยความจำและ Cache เป็นส่วนย่อย ๆ ที่มีขนาดเท่ากัน เพื่อความสะดวกในการจัดการ
- Hit หมายถึง ข้อมูลที่เราต้องการใช้งานนั้นอยู่ใน Cache ซึ่งหากเป็นกรณีการอ่านข้อมูลจะเรียกว่า Read Hit และกรณีการเขียนข้อมูลจะเรียกว่า Write Hit
- Hit Rate หมายถึง อัตราการ Hit ของการอ้างอิงถึงข้อมูล
- **Hit Time** หมายถึง เวลาที่ใช้ในการเข้าถึงข้อมูลกรณี Hit มักมีค่าเป็น 1 Cycle
- Miss หมายถึง ข้อมูลที่เราต้องการจะใช้งานนั้นไม่อยู่ใน Cache ซึ่งเราจะเรียกการ Miss ในกรณีการอ่านข้อมูลว่า Read Miss และ เรียกการ Miss ในกรณีการเขียนข้อมูลว่า Write Miss
- **Miss Rate** (หรือ Miss Ratio) หมายถึง อัตราการ Miss ของการอ้างอิงถึงข้อมูล (มีค่าเป็น $1 - \text{Hit Rate}$) มักมีหน่วยเป็น ค่า Miss ต่อจำนวนคำสั่ง (โอกาสที่ 1 คำสั่งจะเกิด Miss)
- **Miss Penalty** (หากແປລຕາມคำศັບທີ່ຈະເຮັດກວ່າ ค่าປັບປຸງໄໝເຈືອ) หมายถึง เวลาที่ต้องใช้ในการนำข้อมูลจากหน่วยความจำลำดับถัดไปเข้ามาไว้ใน Cache มักมีหน่วยเป็นจำนวน Cycle

ทั้งนี้การแบ่งหน่วยความจำและ Cache ออกเป็น Block นั้นช่วยให้เราสามารถที่จะนำข้อมูลเข้าและออกจาก Cache ได้ง่าย ซึ่งหากเรามองว่า Cache เป็นหน่วยความจำใหญ่เพียงก้อนเดียว โอกาสที่จะเกิดการ Miss และเวลาในการนำเข้าและเก็บข้อมูลของ Cache ย่อมเปลี่ยนแปลงตามขนาดของ Cache ทำให้เราจัดการได้ยาก ประกอบกับลักษณะการทำงานของโปรแกรมโดยทั่วไปมักเกี่ยวข้องกับข้อมูลที่มีลักษณะโดด (ไม่ติดกัน) ซึ่งการแบ่งข้อมูลเป็น Block นี้ช่วยลดโอกาส Miss ของการทำงานกับหน่วยความจำในลักษณะดังกล่าวด้วย ดังจะแสดงได้จากระบบการจัดการ Cache แบบ Direct Mapped ต่อไป

7.3.1 Direct Mapped Cache

การจัดการกับ Cache แบบ Direct Mapped นั้นเป็นระบบการจัดการที่มีพื้นฐานง่ายที่สุด ซึ่งลักษณะโดยสรุปคือ ตำแหน่งในหน่วยความจำทุกตำแหน่งจะมีที่อยู่ใน Cache เพียง 1 ตำแหน่งเท่านั้น เพื่อประกอบความเข้าใจ ขอยกตัวอย่างอธิบายโดยใช้เลขฐาน 10 เป็นตัวอย่างดังนี้

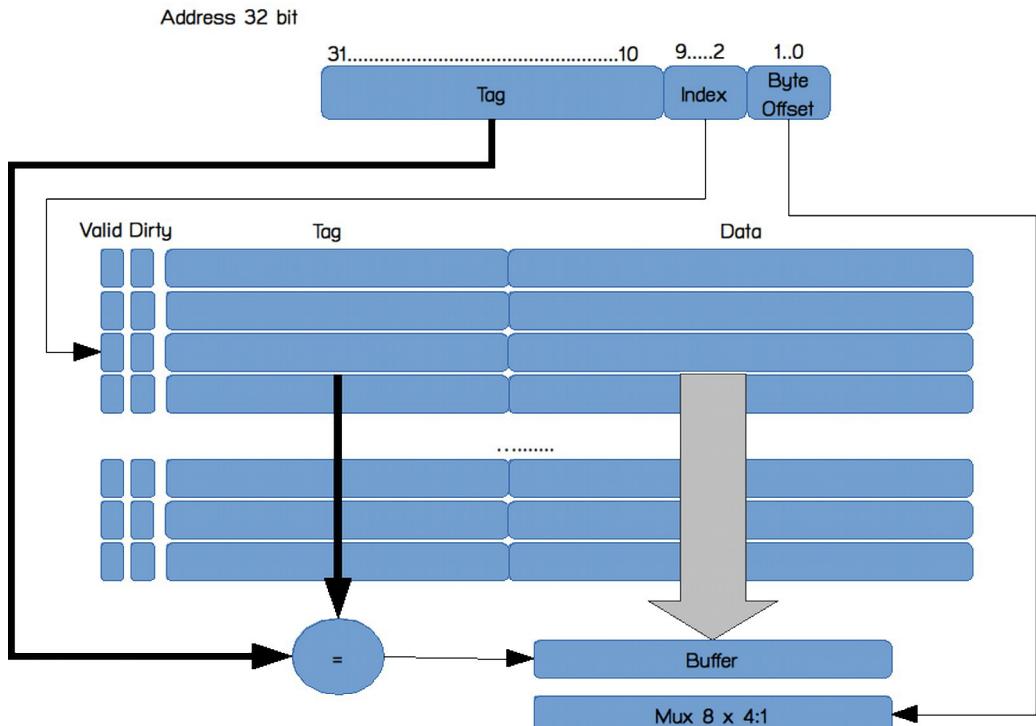
จากรูปที่ 7.4 หากเปรียบเทียบ Cache เป็นเสมือนเก้าอี้ในห้องทำงานซึ่งมีอยู่ 10 ตัว (ติดหมายเลขตั้งแต่ตัวที่ 0 – 9) และในการทำงาน จะต้องทำงานกับคน 1000 คน (โดยแต่ละคนมีเลขประจำตัวของตนเองที่ไม่ซ้ำกัน เริ่มตั้งแต่ 0000 - 9999) เมื่อต้องการทำงานกับคนที่มีเลขประจำตัว 101, 103, 105 ก็จะให้แต่ละคนนั้นเก้าอี้ตัวที่ 1, 3 และ 5 ตามลำดับ ซึ่งหากต้องการทำงานกับคนที่มีเลขประจำตัวเป็น 251 ก็จะต้องให้คนที่มีเลขประจำตัว 101 ออกไปก่อน และวิธีนี้จะทำให้คนที่มีเลขประจำตัว 251 นั่งแทนที่เป็นต้น จะสังเกตว่าตำแหน่งใน Cache ของคนหนึ่งจะเป็นอะไร ขึ้นอยู่กับเลขหลักสุดท้ายของเลขประจำตัว ซึ่งการทำงานของ Direct Mapped Cache ก็จะอยู่ในลักษณะเดียวกัน



รูปที่ 7.4: ตัวอย่างการจัดคนเข้านั่งเก้าอี้ในห้อง (locality) แบบ direct mapped cache

อย่างไรก็ตาม จะเห็นได้ว่า ไม่ว่าจะเป็นคนที่มีเลขประจำตัวเป็น 201, 211, 101, 151 หรือ อีกหลายเบอร์ที่ลงท้ายด้วย 1 นั้น ต่างก็ต้องนั่นที่เก้าอี้ตัวที่หนึ่งเหมือนกัน ซึ่งหากพบร่วมมีคนนั่งอยู่ที่เก้าอี้ตัวที่ 1 แล้ว เราจะทราบได้อย่างรวดเร็วคุณ ที่นั่งอยู่นั้นเป็นใคร (มีเลขประจำตัวอะไร) ด้วยเหตุนี้จึงจำเป็นจะต้องมีการติดป้าย (Tag) เพื่อให้ทราบว่าคนที่นั่งอยู่นั้นมีเลขประจำตัวเป็นอะไร นอกจากนั้นเนื่องจาก Cache ก็เป็นหน่วยความจำขนาดหนึ่งเพื่อแยกแยะให้ทราบว่าข้อมูลในหน่วยความจำนั้นเคยถูกใช้งานมาก่อนหรือไม่ จึงจำเป็นต้องมีการตรวจสอบด้วย Valid bit เพื่อแสดงว่าข้อมูลใน Cache นั้นมีข้อมูลอยู่หรือไม่ (เพื่อให้ทราบว่ามีคนนั่งอยู่หรือไม่ กรณีที่พบว่ามีป้ายติดอยู่เป็น 000 หรือ กรณีเริ่มต้นระบบ) นอกจากนี้กรณีเป็น Cache ของข้อมูลซึ่งมีการเขียนข้อมูลได้ อาจจะมีการเพิ่ม Dirty bit เข้าไปเพื่อแสดงว่าข้อมูลใน Cache entry (บรรทัด) ดังกล่าว มีการเขียนข้อมูลใหม่ลงไปหรือไม่ หากมีการ

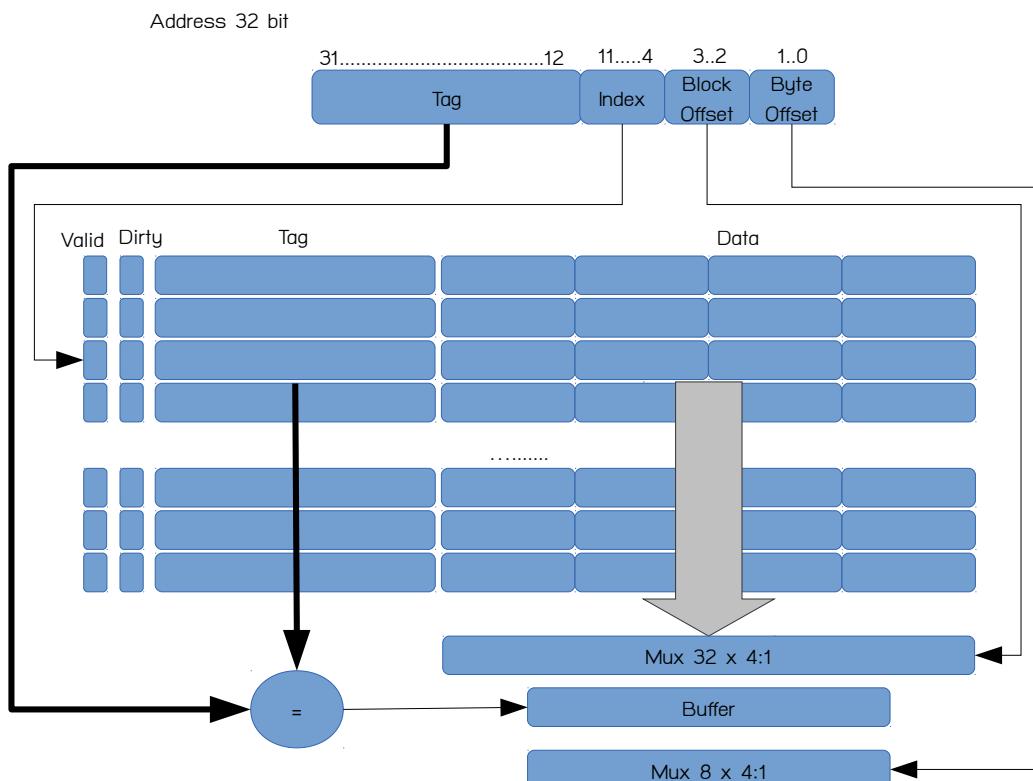
เขียน (มีการตั้งค่า Dirty bit) แสดงว่ามีการเขียนข้อมูล ซึ่งอาจจะต้องมีการนำข้อมูลออกไปเขียนคืนก่อนเมื่อการแทนที่ด้วยข้อมูลใหม่ จากคุณลักษณะดังกล่าว โครงสร้างที่แท้จริงของระบบ Cache จึงมีลักษณะดังรูปที่ 7.5



รูปที่ 7.5: โครงสร้าง Direct Mapped Cache

จากภาพจะพบว่าระบบ Cache ดังกล่าวมี Block ขนาด 4 Byte และ มี Cache ทั้งสิ้น 256 Block (อ้างอิงด้วยจำนวน 8 บิต คือ 9..2) รวมกันแล้วคิดเป็น Cache ขนาด 1 Kbyte การตรวจสอบว่า ข้อมูลที่ต้องการอยู่ใน Cache หรือไม่ ทำได้โดยการตรวจ Tag ของตำแหน่งข้อมูลที่ต้องการอ่านกับ Tag ของ Cache ในตำแหน่งที่ถูก Mapped ด้วย Index หากตรงกัน และ Valid บิต เป็นหนึ่งแสดงว่าข้อมูลที่ต้องการอยู่ใน Cache (หรือ Hit นั่นเอง)

นอกจากนี้ ยังมีการประยุกต์ให้การจัดการข้อมูลในระบบ Cache มีการแบ่ง Block ที่ละเอียดมากขึ้น โดยอาศัยหลักการ Spatial Locality ทำให้ Cache 1 Record (1 บรรทัดตามรูป) มีหลาย Block ซึ่งชี้โดย Block Index ดังแสดงในรูปที่ 7.6



รูปที่ 7.6: โครงสร้าง Direct Mapped Cache ที่รองรับการทำงานแบบ Spatial Locality

จากรูปจะเห็นว่า โครงสร้างดังกล่าวมี Block ขนาด 16 Byte โดยแบ่งออกเป็น 4 block ย่อยในแต่ละ block ย่อย มี 4 Byte ทำการเลือกตัวอย่าง Block offset ขนาด 2 บิต คิดเป็น Cache ขนาด 4 Kbyte

7.4 ประสิทธิภาพของ Cache

ตามคำจำกัดความ การที่ Cache หรือ Locality จะมีประสิทธิภาพที่ดี ขึ้นอยู่กับว่าข้อมูลที่ต้องการใช้ จะมีการ Hit มากน้อยเพียงใด หาก Hit บ่อย หมายถึงข้อมูลที่ต้องการใช้อยู่ใน Locality ย่อมใช้งานได้เร็ว ประสิทธิภาพดี ในมุมกลับคือ หาก Miss บ่อย หมายความว่า Hit สูงนั่นเอง ดังนั้นการพูดว่า ต้องการลด Miss Rate หรือเพิ่ม Hit Rate จึงเป็นเรื่องเดียวกัน

อย่างไรก็ตาม การ Hit และการ Miss นั้นแม้หมายถึง กรณีที่ข้อมูลตำแหน่งที่หน่วยประมวลผลต้องการใช้งานอยู่ใน Cache หรือไม่อยู่ใน Cache โดยไม่แยกแยะว่าเป็นการอ่าน หรือ การเขียน ก็ตาม การ Hit และการ Miss สำหรับการอ่านและการเขียนข้อมูลนั้น ก็ส่งผลต่อประสิทธิภาพและระบบจัดการที่แตกต่างกันไป ดังนี้

- **Read Hit** หมายถึง การ Hit ข้อมูลใน Cache ในกรณีที่หน่วยประมวลผลต้องการอ่าน

ข้อมูลซึ่งสิ่งที่หน่วยประมวลผลกลางจะทำคือ การอ่านข้อมูลจาก Cache ไปใช้ประมวลผล

- **Read Miss** หมายถึง การ Miss ข้อมูลที่หน่วยประมวลผลต้องการอ่าน (ไม่พบใน Cache) ซึ่งสิ่งที่หน่วยประมวลผลต้องทำคือ ต้องหยุดรอ (จ่ายค่าปรับเป็นเวลา) เพื่อให้ระบบจัดการ Cache ทำการดึงข้อมูลจากหน่วยความจำหลักเข้าสู่ Cache ก่อน (ตามคำสั่งที่ก่อนหน้านี้เวลาที่หน่วยประมวลผลต้องหยุดรอ) นิริยากรวมว่า **Miss Penalty** จากนั้นเมื่อระบบจัดการ Cache นำข้อมูลจากหน่วยความจำหลักเข้ามาเก็บใน Cache เรียบร้อยแล้ว จึงดำเนินการตามปกติเหมือน 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) หยุดรอ (Miss Penalty) เพื่อให้ระบบจัดการ Cache นำข้อมูลจากหน่วยความจำหลักเข้ามาเก็บใน Cache ก่อน จากนั้นจึงทำการประมวลผลในลักษณะเดียวกันกับการ Write Hit ข้างต้นต่อไป ลักษณะนี้เรียกว่า **Write Allocate** และ(2) ทำการเขียนข้อมูลต่อไปเลย เมื่อไหร่ก็ได้การ Write Hit ตามปกติ (เพราะไม่มีเหตุผลที่จะต้องนำข้อมูลเข้ามา เพื่อเขียนทับ) ลักษณะนี้เรียกว่า **Write Not Allocate**

ในกรณีของการ Write Hit จะเห็นว่า ทั้ง 2 แนวทางในการจัดการนี้ ก็จะมีข้อดีข้อเสียต่างกันและส่งผลต่อประสิทธิภาพต่างกัน กล่าวคือ Write Back นั้นจะทำงานได้เร็วกว่า เพราะจะเขียนข้อมูลที่ต้องการลงใน Cache จะไม่ต้องกับข้อมูลในหน่วยความจำหลัก และเมื่อต้องนำข้อมูลใน Block นั้นออกจาก Cache จึงจะทำการเขียนข้อมูลดังกล่าวดีนี้ไปยังหน่วยความจำหลัก สำหรับกรณีของการ Write Through²⁶ นั้น หน่วยประมวลผลกลางจะใช้เวลาในการทำงานมากกว่าคือ จะต้องมีการเขียนข้อมูลลงทั้งใน Cache และหน่วยความจำหลักซึ่งจะไม่มีช่วงเวลาที่ข้อมูลในหน่วยความจำหลักมีค่าไม่ตรงกับค่าของข้อมูลใน Cache ด้วยลักษณะดังกล่าว ระบบ Write Back จึงไม่เหมาะสมที่จะใช้งานบนระบบที่หน่วยประมวลผลกลางหลายตัว

ในการนี้ของการ Write Miss การตัดสินใจว่าจะทำ Write Allocate หรือ Write Not Allocate ย่อมส่งผลต่อประสิทธิภาพเพ่นเดียวกัน อย่างไรก็ตาม บ่อยครั้งที่สถาปนิก (ผู้ออกแบบ) มักเลือกใช้ระบบ Write Allocate เพราะเวลาเขียนข้อมูลอาจมีได้เขียนเต็มทุก block ของ Cache ดังนั้น หากไม่นำข้อมูลเข้ามาก่อน (Write Allocate) อาจจะทำให้ข้อมูลใน Cache ผิดพลาดไปได้ อย่างไรก็ตาม ในปัจจุบันมีการเสนอสถาปัตยกรรมการจัดการ Cache แบบมีระบบ sub block กล่าวคือ มี valid bit และ dirty bit แยกกันในแต่ละ block เพื่อจะได้ต้องทำการ allocate แต่ไม่มีผลกระทบ

26 ข้อสังเกตุ การเลือกการทำงานแบบ Write Through ในโครงสร้างของ Cache (รูปที่ 7.5) จะไม่จำเป็นต้องมี Dirty bit เนื่องจากข้อมูลที่เขียนจะมีค่าเหมือนกับข้อมูลใน Cache เพราะมีการเขียนข้อมูลทั้งสองที่

ทบทวนเกี่ยนข้อมูลไม่เต็ม block ด้วย แต่ลักษณะการอคบแบบดังกล่าว อยู่นอกขอบเขตของหนังสือ เล่นี หากผู้เรียนสนใจ สามารถค้นคว้าเพิ่มเติมได้

จากลักษณะการทำงานของ Cache ดังกล่าว จะพบว่า Cache ที่มีประสิทธิภาพนั้นคือ Cache ที่มีระบบการจัดการซึ่งทำให้ Miss Ratio มีค่าต่ำ และ เวลาที่ต้องหยุดเพื่อให้ระบบจัดการ Cache นำข้อมูลจากหน่วยความจำหลักเข้ามาสู่ Cache (Miss Penalty) มีเวลาน้อยที่สุด ซึ่งจากลักษณะดังกล่าวทำให้เวลาที่ใช้ในการประมวลผลของหน่วยประมวลผลของหน่วยประมวลผลกลางนั้นมีความสัมพันธ์กับเวลาที่ต้องหยุดรอเพื่อให้ระบบจัดการ Cache ทำงาน ดังแสดงได้ดังสมการที่ 7.1

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

สมการที่ 7.1: Memory Access Time (หน่วย cycle)

ซึ่งหากวิเคราะห์ต่อไป โดยนำสมการดังกล่าวไป朋นรวมกับสมการเพื่อหาค่า CPU Time จะได้ว่า ค่า Memory Access Time จะถูกownikเข้าไปเป็นส่วนหนึ่งของ CPI ดังแสดงได้ในสมการที่ 7.2

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

สมการที่ 7.2: ค่า CPI กรณีที่มี CACHE

ทั้งนี้ให้ลองนำค่า CPI จากสมการสมการที่ 7.2 กลับเข้าไปแทนค่าในสมการที่ 2.8 จะพบว่า เมื่อมองผิวเผิน CPU time ในกรณีที่มี Cache นั้นจะมากกว่า CPU Time ในกรณีที่ไม่มี ทั้งนี้เนื่องจาก กรณีที่มี Cache เพราะจะมี Stall Cycles ซึ่งเกิดจาก miss ratio x miss penalty เพิ่มขึ้นมา ซึ่งในความเป็นจริงนั้นการมี Cache ช่วยให้ CPU Time มีค่าน้อยลง โดยจะขอเปรียบเทียบดังนี้

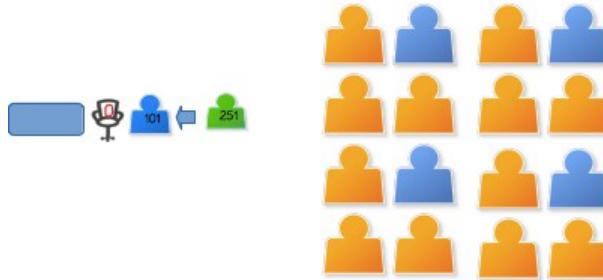
ในกรณีหน่วยประมวลผลกลางแบบ Multiple Cycle หากไม่มีระบบ Cache จะทำให้การเข้าถึงหน่วยความจำทุกครั้ง จะต้องเป็นการเข้าถึงหน่วยความจำที่มีความเร็วต่ำ ซึ่งเวลาในการเข้าถึงนี้ จะเทียบได้กับ miss penalty หรือกล่าวได้อีกนัยหนึ่งว่า miss ratio มีค่าเป็น 1 เสมอ (miss ทุกครั้ง เพราะไม่มี locality ให้คืนหากข้อมูล)

จากการข้างต้นพบว่า การปรับปรุงประสิทธิภาพของระบบ Cache นั้นสามารถทำได้โดยการลด Miss Ratio และ/หรือ ลด Miss Penalty ตามแนวทางที่จะนำเสนอดังนี้

7.5 การลด Miss Ratio

การลด Miss Ratio นั้น คือ การทำให้ข้อมูลที่หน่วยประมวลผลต้องการใช้ มีโอกาสที่จะถูกพบใน Cache มากที่สุด ทั้งนี้ปัจจัยสำคัญหนึ่งคือ การกำหนดให้ Cache มีขนาดเหมาะสม และมีขนาดของ Block Size ที่เหมาะสม ซึ่งจากการจะตอบว่าขนาดของ Cache และ Block Size ที่เหมาะสมเป็นเท่าใดนั้น เป็นสิ่งที่ประเมินได้ยาก เพราะแต่ละโปรแกรมย่อมมีลักษณะของ locality แตกต่างกันไป ทดสอบพบว่า จะมีค่าที่เหมาะสมอยู่ค่าหนึ่ง ๆ สำหรับหน่วยประมวลผลกลางแต่ละตัว ทั้งนี้หาก Block Size มีขนาดใหญ่เกินไป (เมื่อกำหนดให้ Cache มีขนาดเท่าเดิม) ในบางกรณี Cache จะจะมี Miss Ratio มากขึ้นก็ได้ ทั้งนี้เนื่องจากจำนวน Temporal จะลดลง เพื่อประกอบการอธิบาย ลอง

วิเคราะห์ตัวอย่างกรณีสุดขั้วด้านหนึ่งคือ กรณีมี Cache เพียง 1 entry (Block Size = Cache Size) ซึ่งเปรียบเทียบแล้ว จะเหมือนกับว่า เรามีเก้าอี้สำหรับให้คนเข้ามาในห้องเพียงที่เดียว ดังแสดงในรูปที่ 7.7



รูปที่ 7.7: กรณีมี Cache เพียง 1 entry

จากภาพจะเห็นว่าทุกครั้งที่เราต้องทำงาน จะต้องเชิญคนเข้าออกตลอดเวลา จนกล่าวได้ว่า เวลาส่วนใหญ่จะเสียไปกับการให้คนเข้าและออก จึงไม่สามารถที่จะทำอะไรให้เสร็จได้ ปัญหาลักษณะนี้เรียกว่า Ping-Pong effect ซึ่งมีลักษณะคล้ายกับ Trashing ในเรื่องของ Virtual Memory ที่หน่วยประมวลผลกลางยุ่งกับการทำ Page in และ Page out ตลอดเวลา จึงไม่สามารถทำอย่างอื่นให้เสร็จได้

เพื่อให้สามารถแก้ปัญหาเรื่อง Miss Ratio ได้ถูกต้อง ลองวิเคราะห์ต่ออีกนิดหนึ่งว่า Miss เกิดจากสาเหตุใดได้บ้าง เมื่อการ Miss เกิดจากการหาข้อมูลแล้วไม่พบข้อมูลที่ต้องการใช้ใน Locality ดังนั้น Miss จึงเกิดขึ้นได้จาก 3 สาเหตุด้วยกัน ได้แก่

1. **Compulsory Miss** หรือ **Cold Miss** คือ Miss ที่เกิดจากการใช้งานครั้งแรก ซึ่งอาจจะไม่สามารถแก้ไขอะไรได้มากนัก (เปรียบเสมือนกับเริ่มมานั่งเตี้ยทำงาน ยังไม่ได้หยิบของออกจากกระเพามาตั้งบนโต๊ะ ย่อมต้อง Miss เพื่อยิบของครั้งแรก) มีแนวทางที่แก้ไขได้ทางหนึ่งคือการทำ Prefetch กล่าวคือ หยิบของขึ้นมาไว้ก่อนไม่ว่าจะใช้หรือไม่ก็ตาม (รายละเอียดมีได้กล่าวถึงในที่นี้)
2. **Capacity Miss** คือ Miss ที่จาก Cache มีขนาดเล็กไป ตัวอย่างเช่น กรณี Ping-Pong effect ในรูปที่ 7.7 ซึ่งมี Cache น้อยเกินไป ไม่พอกับขนาด Locality ที่จำเป็นต้องใช้ จึงต้องมีการนำข้อมูลเข้าออก วิธีแก้ที่ตรงไปตรงมาที่สุดคือ การเพิ่มขนาดของ Cache (Cache Size²⁷)
3. **Conflict Miss** คือ Miss ที่เกิดขึ้นเมื่อยังมีที่วางเหลืออยู่ใน Cache แต่เนื่องจากระบบบริหารจัดการ ทำให้ไม่สามารถนำข้อมูลนั้นไปวางยัง Cache ที่วางอยู่ได้ เพื่อให้เห็นภาพลองวิเคราะห์รูปที่ 7.4 จะเห็นว่า เมื่อยังมีที่วางอยู่ แต่คุณหมายเลข 251 ก็ไม่สามารถนั่งได้ เพราะหมายเลข 251 นั้นถูกกำหนดต้องให้นั่งที่เก้าอี้หมายเลข 1 ซึ่งมีคนนั่งอยู่ก่อนแล้ว

²⁷ เมื่อจากขนาดของ Cache ที่เหมาะสมจะแตกต่างกันไปในแต่ละโปรแกรม ดังนั้นจึงข้อทิ้งคำダメเรื่องขนาดที่เหมาะสมนี้ ให้เป็นแบบฝึกหัดของผู้เรียน

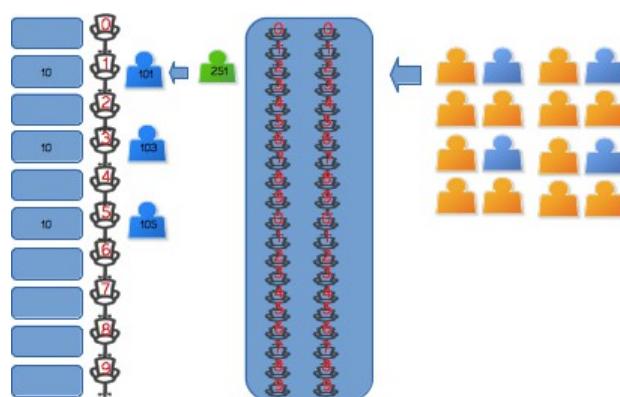
หากสนใจว่า Conflict Miss เป็นหลัก ปัจจัยที่ส่งผลกระทบ Miss Ratio ประกอบด้วย

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)

7.6 การลด Miss Penalty

การลด Miss Penalty คือ การทำให้การโอนถ่ายข้อมูลระหว่าง Cache และ หน่วยความจำหลักสามารถทำได้อย่างรวดเร็ว เนื่องจากในทางปฏิบัติเป็นไปได้ยากที่จะให้ทำให้หน่วยความจำขนาดใหญ่มีความเร็วเพิ่มขึ้น ดังนั้นหลักการที่ดีที่สุดในการลด Miss Penalty คือ การสร้าง Cache เพื่อรับการทำงานหลายระดับ (Multi-Level Cache) โดยเมื่อ หน่วยประมวลผลกลางต้องการหาข้อมูลนั้น หาก Miss ในระดับแรกอาจจะต้องเสีย Miss Penalty ถึง 100 Cycle เพื่อดึงข้อมูลจากหน่วยความจำหลัก แต่หากมีระบบ Cache หลายระดับ เนื่องจากข้อมูลในระดับแรก (Miss) อาจ จะพบข้อมูลใน Cache ระดับถัดไปก็ได้ ทำให้ระบบไม่จำเป็นจะต้องเสีย Miss Penalty ถึง 100 Cycle ทุกครั้งที่มีการ Miss เกิดขึ้น หากวิเคราะห์ให้ดีจะพบว่า การลด Miss Penalty โดยการทำ Multi-Level Cache นั้น คือ การพยายามทำให้ข้อมูลบางส่วนสามารถหยิบใช้งานได้เร็วขึ้น มีได้เป็นการลดค่าตัวเลขโดยตรง เพื่อประกอบความเข้าใจ ดูรูปที่ 7.8 ประกอบคำอธิบายดังนี้



รูปที่ 7.8: การทำ Multilevel Cache เพื่อลด Miss Penalty

รูปที่ 7.8 เป็นการทำ Multi-level Cache โดย ให้หันน่วยความจำ(RAM) เปรียบเสมือนคนที่นั่งอยู่ในหอประชุมขนาดใหญ่จำนวน 1,000 คน กรณีที่ต้องการทำงานกับกลุ่มคนดังกล่าว จะมี Cache level 0 อยู่ในห้องทำงานจำนวน 10 ที่นั่ง หากทุกครั้งที่ต้องการเรียกคนจากหอประชุม เข้ามาในห้องทำงาน จะต้องใช้เวลา 100 วินาที หากสามารถทำห้องพักระหว่างกลาง ซึ่งจุดใด 50 ที่นั่งไว้เป็นที่พักก่อน และการเรียกคนจากห้องพักนี้มายังห้องทำงานจะใช้เวลา 50 วินาที นั่งหมายความว่า มีโอกาสเป็นไปได้ที่ จะพบคนคนนั้นให้ห้องพัก ช่วยให้สามารถทำงานได้เร็วขึ้น เพราะไม่ใช่ทุกครั้งที่หาไม่เจอนั่นห้องทำงาน จะต้องวิ่งไปที่หอประชุมเสมอไป ลองดูการวิเคราะห์ในตัวอย่างที่ 7.1

ตัวอย่างที่ 7.1: ประสิทธิภาพของ Multi-level Cache

จากตัวอย่างในรูปที่ 7.8 สมมุติว่า Miss Rate ใน Level 0 เป็น 15% และ Miss Rate ใน Level 1 เป็น 25% ให้ Hit Time ของ Level 0 เป็น 5 วินาที และ Miss Penalty จาก Level 0 ไปยัง Level 1 เป็น 50ns และ Miss Penalty จาก Level 0 และ Level 1 ไปยังหอประชุม คือ 100ns จงแสดง Speed Up ที่ได้จากการเพิ่ม Level 1 เข้าไปยังห้องประมวลผลกลาง

กรณีไม่มี Level 1 จะได้ว่า Access Time เป็น

$$\text{Access Time}_{1\text{level}} = 5 + 0.15 \times 100 (\text{ns})$$

กรณีมี Level 1 จะได้ว่า Access Time เป็น

$$\text{Access Time}_{2\text{level}} = 5 + 0.15 \times (50 + 0.25 \times 100) (\text{ns})$$

คิด Speed Up จะได้เป็น

$$\text{SpeedUp} = \frac{\text{Access Time}_{1\text{level}}}{\text{Access Time}_{2\text{level}}} = \frac{5 + 0.15 \times 100}{5 + 0.15 \times (50 + 0.25 \times 100)}$$

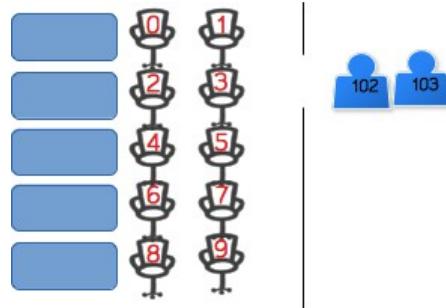
$$\text{SpeedUp} = \frac{20}{16.25} = 1.23$$

จากตัวอย่างจะเห็นว่าได้ประสิทธิภาพเพิ่มขึ้นถึง 23% (คงปรับตัวเลข Miss Rate จะพบว่าไม่ว่าจะปรับ Miss Rate เป็นเท่าไร ก็ยังคงได้ประสิทธิภาพที่ดีขึ้นอยู่ดี)

7.7 Block Size

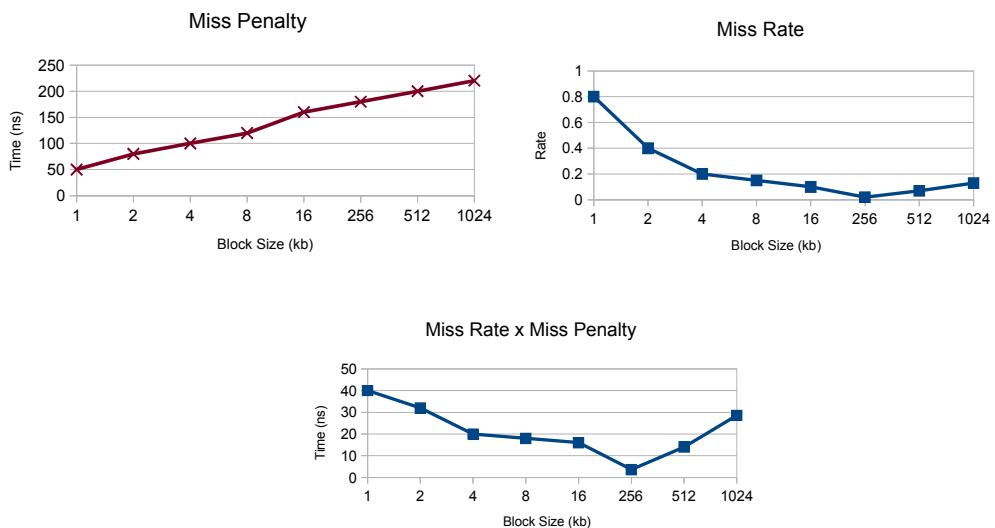
หลักการของ Block Size ที่มีขนาดใหญ่ขึ้นคือ การรองรับ Spatial Locality มากรขึ้น แต่品格ในทางตรงข้ามคือ ทุกครั้งที่มีการ Miss ระบบอาจจะมี Miss Penalty ที่มากขึ้นไปด้วย เนื่องจากในทางปฏิบัติ ทางเดินข้อมูลมีขนาดจำกัด การนำข้อมูลจำนวนมาก เข้ามาภายใน ส่งผลให้ต้องใช้เวลามากขึ้นไปด้วย ลองพิจารณารูปที่ 7.9 จะเห็นว่า มีการเพิ่มขนาดของ block size เป็น 2 แต่เนื่องจากทาง

เดินข้อมูล (ประตุ) ยังคงเข้าได้ที่ละ 1 ทำให้ Miss Penalty มากรึเปล่า



รูปที่ 7.9: block size ต่อ miss penalty

เมื่อนำผลที่ได้มารวมกัน(ดังตัวอย่างกราฟในรูปที่ 7.10) จะเห็นว่า การเพิ่มค่า block size ไปเรื่อยๆ จะทำให้ Miss Penalty เพิ่มขึ้นตามไปเรื่อยๆ แต่จะทำให้ Miss Rate ลดลงเนื่องจากการองรับ Spatial Locality ได้มากขึ้น จนถึงระดับที่ทำให้ Temporal Locality น้อยเกินไป ค่า Miss Rate จึงเพิ่มขึ้น แต่เนื่องจาก เวลาในการเข้าถึงโดยรวม เป็นผลคูณของ Miss Rate และ Miss Penalty ดังนั้น การเลือกขนาดของ Block Size ที่เหมาะสมจึงควรใช้ความระมัดระวัง

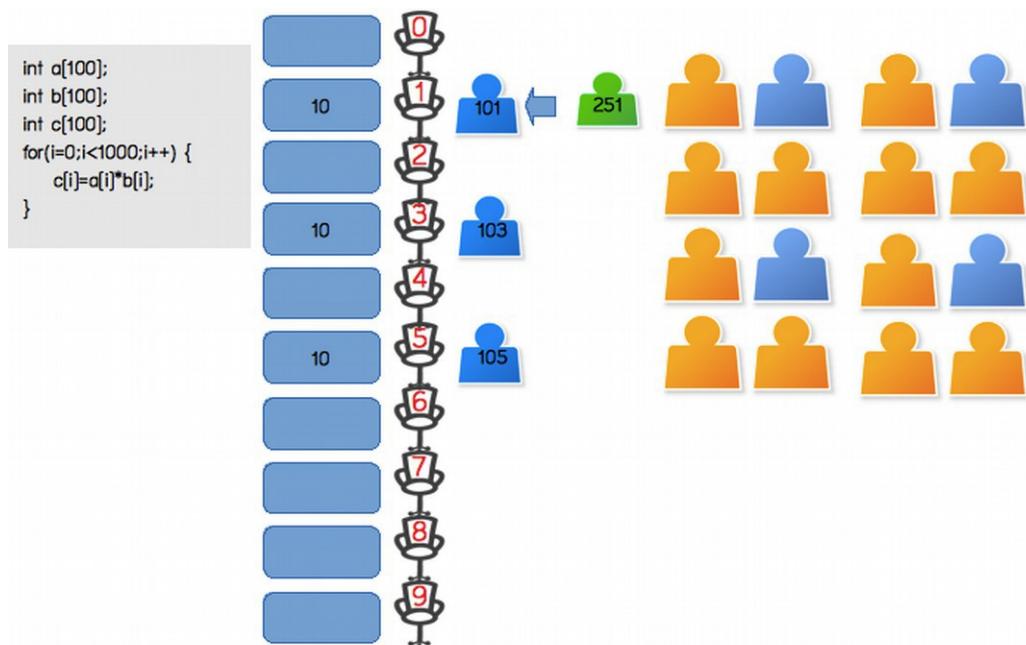


รูปที่ 7.10: กราฟความสัมพันธ์ระหว่าง block size, miss rate และ miss penalty

หมายเหตุ ส่วนใหญ่การออกแบบในปัจจุบัน มักใช้การทดสอบใน simulator เพื่อตรวจสอบผลลัพธ์ ก่อนการสร้างจริง ทั้งนี้ค่าที่แสดงในรูปที่ 7.10 เป็นเพียงค่าสมมุติเพื่อประกอบการอธิบายเท่านั้น

7.8 Associativity

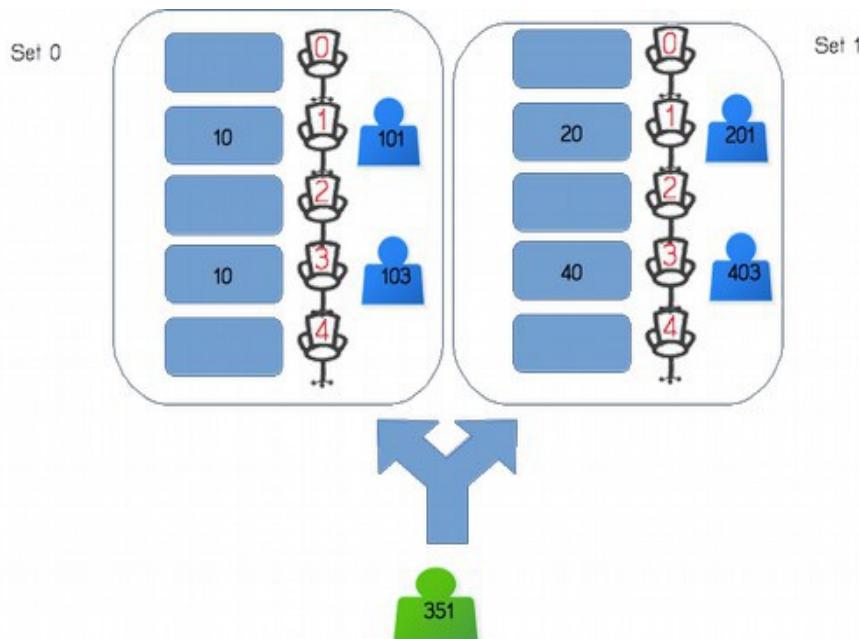
เบื้องต้นนั้นเราได้ศึกษาถึงระบบการจัดการ Cache แบบ Direct Mapped และ ชีงการจัดการ Cache แบบ Direct Mapped นั้นอาจเรียกว่าเป็นการจัดการแบบ One-way set associative cache หรือ การกำหนดให้ 1 ตำแหน่งของข้อมูลในหน่วยความจำหลักนั้นมีที่อยู่ใน Cache ได้เพียง 1 ที่เท่านั้น ปัญหาที่พบคือ มีโอกาสที่เกิด Miss Ratio จาก Conflict Miss สูงมาก รูปที่ 7.11 แสดงตัวอย่างการเกิด conflict miss



รูปที่ 7.11: ตัวอย่างการเกิด conflict miss ใน 1-way set associative cache

จากรูป หาก a เริ่มต้นที่ 100 b เริ่มต้นที่ 200 และ c เริ่มต้นที่ 100 จะเห็นว่า ทุกครั้งที่มีการอ้างอิง ข้อมูล $a[i]$, $b[i]$ และ $c[i]$ จะเกิด conflict miss ขึ้นทุกครั้ง เพราะทั้ง 3 ตัวแปรจะอ้างอิงลงที่ ตำแหน่งเดียวกันใน cache เสมอ ดังนั้น หากสามารถทำการสร้าง ให้มี set associative ที่สูงขึ้น จะได้ว่าโอกาสเกิด conflict ลดลง

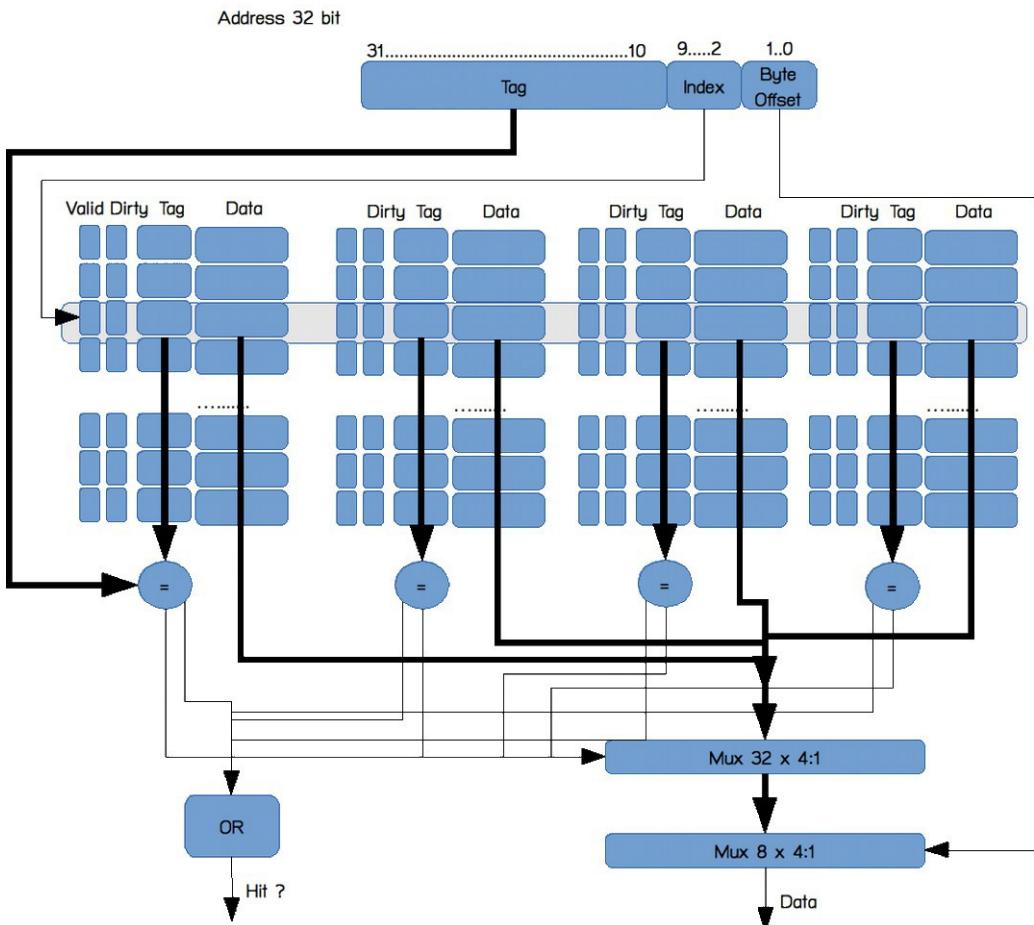
รูปที่ 7.12 เป็นการแก้ปัญหาจากรูปที่ 7.11 โดยการทำเป็น two-way set associative ซึ่งจะเห็นได้ว่า Cache Size โดยรวมมีขนาดเท่าเดิม Block Size มีขนาดเท่าเดิม แต่แบ่งออกเป็น 2 ชุดย่อย ดังนั้น ข้อมูลชุด 101 และ 201 ซึ่งเดิม จะต้องอยู่ที่เดียวกัน ควรวนะจะอยู่พร้อมกันได้ เพราะมีตัวเลือกให้ 2 ที่ อย่างไรก็ตาม หากมี 301 เข้ามา ก็จะต้องทำการเลือกว่า จะเอาข้อมูลชุดใดออก ซึ่งกล่าวถึง ต่อไปในส่วนของ Replacement Algorithm



รูปที่ 7.12: การเพิ่ม set associative เพื่อลดปัญหา conflict miss

อย่างไรก็ตาม แม้ว่าระบบการจัดการแบบมี Associative way มากกว่า 1 ทางจะช่วยลด Miss Ratio จากปัญหา conflict miss ได้ดี แต่การจัดการระบบเพื่อตรวจสอบการ Hit และ Miss ก็ยังยากขึ้นไปด้วย เพื่อประกอบการอธิบาย รูปที่ 7.13 แสดงการทำงานของระบบ Cache ในแบบ Four-way set associative ซึ่งมีลักษณะคล้ายกับ Direct Mapped Cache จำนวน 4 ชุดต่อชานานกัน การตรวจสอบการ Hit นั้นต้องเปรียบเทียบ Tag กับหัว 4 Block และเมื่อมีการ Hit แล้ว ก็จะต้องผ่าน Multiplex เพื่อเลือกข้อมูลที่ต้องการออกมานำใช้ จะเห็นได้ว่า วงจรโดยรวมมีขนาดใหญ่และมีความซับซ้อนยิ่งขึ้น เพราะ การ Hit อาจจะเกิดจากข้อมูลชุดใดก็ได้ใน 4 ชุด และหากมีการอ้างอิงข้อมูลแบบ Byte Offset อีก ก็จะต้องรอให้การเลือกข้อมูลจากชุด associative เสร็จก่อน จึงจะทำการเลือกข้อมูล byte ที่ต้องการได้ ทั้งหมดนี้ ส่งผลให้เวลาจรวจทำงานช้าลง ดังนั้น อาจจะสรุปในเบื้องต้นได้ว่า **Associativity** ส่งผลเสียทำให้ Hit Time ใช้เวลามากขึ้น

อย่างไรก็ดี การทดสอบในสถาปัตยกรรมส่วนใหญ่ใช้ให้เห็นว่า การเพิ่ม Associative Way ช่วยให้ Miss Rate มีค่าลดลงในลักษณะเกือบจะเป็นเชิงเส้น



รูปที่ 7.13: โครงสร้างของระบบ Cache แบบ 4-way set associative

7.9 Replacement Algorithm

อีกปัจจัยหนึ่งที่ส่งผลกระทบต่อ Conflict Miss คือการเลือก Cache Entry ที่ต้องนำออก กรณีเกิด Conflict ซึ่งจะมีผลก็ต่อเมื่อมี Associativity มากรกว่า 1 (เนื่องจาก Direct Mapped Cache มีที่สำหรับอ้างอิงเพียงหนึ่งที่ ดังนั้นจึงไม่สามารถมีตัวเลือกให้เลือกได้)

ปัจุบันมี algorithm ที่เกี่ยวข้องกับ Replacement อยู่หลายรูปแบบ แต่โดยสรุปมักจะเป็นรูปแบบที่คล้ายคลึงหรือดัดแปลงมาจากสองรูปแบบหลักคือ Least Recently Used และ Random Replacement เป็นหลัก ดังนี้

Least Recently Used (นิยมเรียกโดยย่อว่า LRU) เป็น replacement algorithm ที่ใช้การนับว่า ข้อมูลใน Cache ตัวไหนมีการใช้น้อยที่สุด ตัวนั้นควรจะเป็นเหยื่อ (victim) ที่จะต้องถูกเลือกออกไปจาก Cache ดังนั้นในการทำ LRU จะมีต้องมีการเก็บอายุของ Cache โดยทุกครั้งที่มีการอ้างอิงถึง

ข้อมูลตัวใดตัวหนึ่ง อายุของ Cache ตัวอื่นจะต้องเพิ่มขึ้นด้วย ด้วยเหตุนี้การ implement LRU จึงต้องมี บิต สำหรับเก็บอายุของ Cache²⁸ เพิ่มขึ้น ทำให้การทำงานของ Cache ยิ่งซับซ้อนยิ่งขึ้น โดยเฉพาะในกรณีที่มี Associativity หมายทาง ดังนั้นสถาปัตยกรรมหลายแบบ จึงเลือกใช้ Approximate LRU แบบต่างๆ เพื่อลดความซับซ้อนในทางฮาร์ดแวร์แทน

Round Robin²⁹ ด้วยความซับซ้อนทางฮาร์ดแวร์ อิกแนวทางหนึ่งที่นิยมใช้กันคือ การวิยนสลับ กันล่า้วคือ ทุกครั้งที่เกิด conflict แล้วต้องการ victim ในครั้งแรก ก็จะทำการเลือกข้อมูลใน set ที่ 1 เป็น victim ในครั้งต่อไปที่เกิด conflict ก็จะเลือก victim เป็นข้อมูลใน set ที่ 2 วิธีการนี้ช่วยให้ ฮาร์ดแวร์มีความซับซ้อนน้อยลง

เพื่อเป็นการเปรียบเทียบประสิทธิภาพ ตารางที่ 7.1 (ข้อมูลจาก [2]) เป็นการเปรียบเทียบการทำงาน ของ LRU และ RR เมื่อมีโครงสร้าง Cache เมื่อก่อนกัน จากตารางจะเห็นว่า ค่าที่ได้ มีความใกล้เคียงกัน โดย LRU จะให้ผลที่ดีกว่า (ด้วยฮาร์ดแวร์ที่ซับซ้อนกว่ามาก)

ตารางที่ 7.1: ประสิทธิภาพของ 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

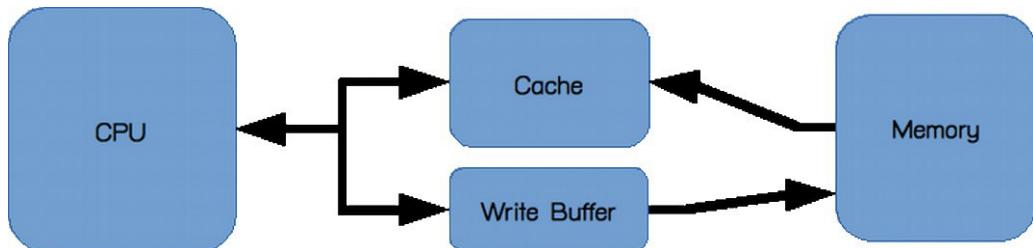
7.10 การจัดการ Cache กรณีการเขียนข้อมูล (Write Management)

ตามที่ได้เกริ่นไว้แล้วก่อนหน้านี้ว่า การเขียนข้อมูลสามารถเลือกได้ว่าจะทำ Write Back หรือ Write Through และเลือกได้ว่าจะทำ Write Allocate หรือ Write No Allocate ซึ่งได้อธิบายไว้ก่อนหน้านี้ ว่า Write Through เป็นทางเลือกที่เหมาะสมมากกว่า ในกรณีที่เป็นการประมวลผลแบบ Parallel หรือมีหลายหน่วยประมวลผล (รวมถึง Multi core) ทำงานกับหน่วยความจำชุดเดียวกัน เพื่อสามารถ ที่จะบริหารจัดการได้ง่ายกว่า ในทำนองเดียวกัน การเลือก Write Allocate จะช่วยให้การบริหาร จัดการง่ายกว่า กรณีที่การเขียนเกิดขึ้นเพียงบางส่วนของ Block ใน Cache

ควรานี้สิ่งที่เป็นปัญหาใหญ่ที่ต้องประสิทธิภาพของ Cache คือ การทำ Write Through นั้น จะ ทำงานได้ช้ากว่า Write Back เพราะจะต้องทำการการเขียนข้อมูลขึ้นทั้งสองแห่งคือ ใน Cache เอง และใน Main Memory ดังนั้น เพื่อให้การเขียนข้อมูลแบบ Write Through เร็วขึ้น จึงมีผู้เสนอให้ทำ Write Buffer เพื่อเป็นวงจรพิเศษสำหรับช่วยเขียนข้อมูลดีนี้ไปยัง Main Memory รายละเอียด สามารถแสดงได้ในรูปที่ 7.14

28 เพื่อให้เห็นถึงความซับซ้อน ขอแนะนำให้ผู้เรียน ลองทำความเข้าใจ LRU และกองเขียนโปรแกรมง่ายๆ เพื่อ จำลองการทำงานของ LRU Cache ดู

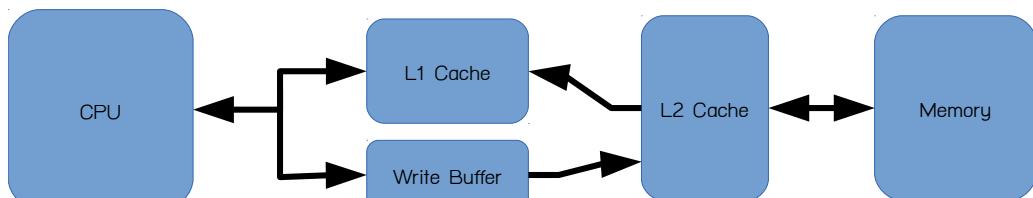
29 Round Robin เป็น Replacement Algorithm ที่นิยมใช้ในสถาปัตยกรรม ARM ซึ่งจากการทดสอบพบว่าให้ผล ใกล้เคียงหรือไม่ด้อยไปกว่า LRU มากนัก



รูปที่ 7.14: การใช้ Write Buffer เพื่อลดเวลาในการเขียนข้อมูลแบบ Write Through

โดยหลักการคือ ทุกครั้งที่มีการเขียน ให้เขียนข้อมูลลงที่ Cache แล้วฝาก Write Buffer ให้นำข้อมูลไปเขียนคืนที่ Main Memory ให้ภายในห้อง แต่ปัญหาที่ตามมาคือ ไม่ว่า Write Buffer จะมีขนาดเท่าใด ก็จะพบว่า Write Buffer จะเต็ม (Saturate) ทำงานไม่ทันการเขียนจากหน่วยประมวลผลกลางเสมอ ทั้งนี้เนื่องจาก การสื่อสารระหว่างหน่วยประมวลผลกลางและ Cache มีความเร็วสูงกว่า การสื่อสารระหว่าง Cache และ Main Memory หาก

ดังนั้น เพื่อยังคงสามารถใช้ Write Through และแก้ (บรรเทา) ปัญหาดังกล่าว คือการใส่ Multilevel Cache กล่าวคือ แบ่งช่วงการทำงานโดยไม่จำเป็นต้องให้ Write Buffer เขียนข้อมูลไปยัง Main Memory โดยตรง เพราะจะใช้เวลามากเกินไป แต่ให้ Write Buffer เขียนข้อมูลลงใน Cache L2 ซึ่งทำงานเร็วกว่า Write Buffer ก็มีโอกาสที่จะเต็มช้าลง (ดู ประกอบคำอธิบาย)



รูปที่ 7.15: การเพิ่ม Level ของ Cache เพื่อบรรเทาปัญหา saturate ของ Write Buffer

ข้อซ่อนอยู่ ลองจินตนาการดูว่า หากต้องวนรอบนึงต้องทำงาน 40 กล่อง ไปปั้นรอบบรรทุก หากคนหนึ่ง มีหน้าที่ยกของลงจากโด๊ะ (เขียนข้อมูลใส่ Cache และ Write Buffer) และอีกคนหนึ่งมีหน้าที่นั่งก่อจ่องจาก Write Buffer ใส่รถเข็นลงพื้นที่ไปใส่ในรถบรรทุก ต่อให้มีคนน้ำหนักใส่รถเข็นแล้ววิ่งไปที่รถบรรทุก 10 คน ก็คงจะทำงานไม่ทันคนที่ยกกล่องลงจากโด๊ะทำงานอยู่ดี เพราะใช้เวลาต่างกันมาก การแก้ปัญหาคือ การแบ่งระยะทางให้สั้นลง และมีคนทำงานเป็นช่วง ก็จะช่วยลดปัญหาการทำงานไม่มันหรือ saturate ลงได้

7.11 โปรแกรมและประสิทธิภาพของ Cache

ในทางปฏิบัติ โปรแกรมจะได้อ่านสังคัด้านประสิทธิภาพจาก Cache อยู่แล้ว โดยไม่จำเป็นต้องทำการปรับปรุงอะไร กล่าวคือ ด้วยโปรแกรมเดิม เพียงเปลี่ยนหน่วยประมวลผลกลางที่มี Cache เพิ่มขึ้น โปรแกรมที่ได้ก็จะทำงานเร็วขึ้นโดยอัตโนมัติ

อย่างไรก็ตาม หากผู้พัฒนาซอฟต์แวร์ เข้าใจโครงสร้างของ Cache ดีพอ ก็สามารถที่จะปรับปรุง code เพื่อให้ได้ประโยชน์จาก Cache มากขึ้นได้ โดยจะขอแยกเป็น ส่วนกรณีคือ กรณีการเขียนโปรแกรมเพื่อใช้ประโยชน์จาก Spatial Locality และ กรณีการเขียนโปรแกรมเพื่อลด Conflict Miss ดังนี้

7.11.1 การปรับโปรแกรมเพื่อช่วยลด Spatial Locality

เนื่องจากโครงสร้าง Cache สมัยใหม่ มักมี block size ที่ใหญ่พอจะเก็บตัวแปรที่อ้างอิงในโปรแกรมทั้วไปได้หลายตัวพร้อมกัน ดังนั้น หากสามารถเขียนการวนรอบ เพื่อให้ได้ประโยชน์จากการทำงานแบบ Spatial Cache การอ้างอิงข้อมูลจาก Array แบบ 2 มิติควรจะอ้างอิงข้อมูลจาก row ก่อน column

เพื่อประกอบความเข้าใจ ตัวอย่างที่ 7.2 แสดงการเขียนโปรแกรมที่ให้ผลเหมือนกัน 2 แบบ อย่างไรก็ตาม หากลองวัดประสิทธิภาพ จะพบว่าเมื่อ data มีขนาดใหญ่ขึ้น Implementation A จะทำงานเร็วกว่า Implementation B มาก เพื่อใน Implementation A จะอ้างอิงถึง $data[i][j]$ และ $data[i][j+1]$ ต่อเนื่องกัน ซึ่งหาก Cache มี Spatial Locality มากรพอ (Block Size มีขนาดใหญ่พอดี) มักจะทำให้ $data[i][j+1]$ เป็นการ Read Hit หากมีการ Read Miss ที่ $data[i][j]$ ในทางตรงกันข้าม Implementation B จะอ้างอิง $data[j][i]$ ต่อตัวย $data[j+1][i]$ ซึ่งไม่ได้อยู่ใกล้เคียงกัน ทำให้ไม่ได้ประโยชน์จาก Spatial แต่อย่างใด

Implementation A	Implementation B
<pre>int data [100] [100]; int sum; for (int i=0; i<100; i++){ for (int j=0; j<100; j++) { sum += data[i] [j]; } }</pre>	<pre>int data [100] [100]; int sum; for (int i=0; i<100; i++){ for (int j=0; j<100; j++) { sum += data[j] [i]; } }</pre>

ตัวอย่างที่ 7.2: การเขียนโปรแกรมเพื่อใช้ประโยชน์จาก Spatial Locality

นี่เป็นเพียงตัวอย่างแบบหนึ่งเท่านั้น ยังมีการเขียนโปรแกรมในลักษณะอื่น เพื่อให้ได้ประโยชน์จาก Spatial Locality อีกซึ่งมีได้ก่อถ่วงใจในที่นี้

7.11.2 การปรับโปรแกรมเพื่อลด Conflict Miss

ในการวางแผนการรับข้อมูลบางรูปแบบ จะมีโอกาสทำให้เกิด conflict miss ได้ง่าย เช่นการใช้ ตัวแปรแบบ Array ที่มีความยาวใกล้เคียงกันหลายชุด เพื่อแทนข้อมูล โอกาสที่ $A[i]$, $B[i]$ และ $C[i]$ จะเกิด conflict miss มักจะสูงกว่าการอ้างอิงตัวแปรที่อยู่ติดกัน (เนื่องจากมี spatial locality) ดังนั้น หากสามารถปรับเปลี่ยนโครงสร้างข้อมูลได้ จะช่วยลดปัญหา Conflict Miss ที่เกิดจากลักษณะนี้ได้

ลองพิจารณา ตัวอย่างที่ 7.3 จะเห็นว่า Implementation Y ให้ผลการทำงานเหมือน Implementation X เพียงแต่ ใน Implementation X จะมีโอกาสเกิด Conflict Miss สูงกว่า ส่วน Implementaion Y ทำการรวม a, b และ c ไว้ใน structure ช่วยให้เกิด spatial locality ของ a, b, c ขณะทำการประมวลผล ซึ่งนอกจากจะเป็นการเลี้ยง conflict miss แล้ว ยังสนับสนุนให้เกิด Hit จากการใช้ข้อมูลแบบ Spatial Locality ด้วย

Implementation X	Implementation 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 7.3: การเขียนโปรแกรมเพื่อเลี้ยงการเกิด conflict miss

ยังมีแนวทางการใช้ data structure อีกหลายแบบที่จะช่วยให้ conflict miss ลดลง รวมถึงเพิ่ม spatial locality ได้ ซึ่งสามารถศึกษาเพิ่มเติมได้ทั่วไป

7.12 หน่วยความจำเสมือน (Virtual Memory)

หน่วยความจำเสมือน ถูกออกแบบมาเพื่อแก้ปัญหาด้านการอ้างอิงหน่วยความจำอย่างน้อย 3 ประการ ได้แก่ 1) การให้ชอฟต์แวร์สามารถอ้างอิงหน่วยความจำได้มากกว่าหน่วยความจำที่มีอยู่จริง (ลดค่าใช้จ่ายด้านฮาร์ดแวร์) 2) การทำให้ชอฟต์แวร์ สามารถ load ไปไว้ที่ไหนก็ได้ซึ่งไม่จำเป็นต้องเป็นที่เดิมเหมือนกับตอนที่พัฒนาหรือชอฟต์แวร์ (Relocatable) 3) การทำให้การข้างอิงหน่วยความจำของแต่ละโปรแกรมเป็นอิสระจากกัน (security)

ความสามารถในการอ้างอิงหน่วยความจำ ในการปรับขนาดมากเกิดกว่าหน่วยความจำจริงที่มีอยู่จริงในระบบเครื่องคอมพิวเตอร์ เกิดขึ้นได้จากการจัดการที่มีการยืมเนื้อที่จากหน่วยจัดเก็บข้อมูลอื่น (เช่น disk) มาทำการลับค่าข้อมูลบางส่วนออก ทำให้ผู้ใช้รู้สึกเสมือนว่าอ้างอิงข้อมูลได้มากกว่าหน่วยความจำที่มีอยู่จริง

คุณสมบัติแบบ Relocatable ตามความหมายดังเดิมคือ การทำให้โปรแกรมดังกล่าว สามารถ load ไปวางที่ address ไหนก็ได้ โดยไม่จำเป็นต้องเป็น address เดียวกันกับตอนที่พัฒนาซอฟต์แวร์ เช่น ในขณะที่พัฒนาโปรแกรมมีการกำหนดว่า ตำแหน่ง 200 เป็นที่สำหรับเก็บข้อมูลตัวแปร แต่หากขณะที่มีการใช้งาน ตำแหน่ง 200 มีการใช้เพื่อเก็บข้อมูลอื่น อาจส่งผลให้โปรแกรมดังกล่าวไม่สามารถทำงานได้ การทำ Relocation ด้วยหน่วยความจำเสมือน หรือ Virtual Memory ช่วยแก้ปัญหานี้ได้โดยการทำให้มีการเปลี่ยนแปลง address ระหว่างการพัฒนาและการใช้งานจริง เพราะ address

ที่อ้างอิงจะเป็น address เสมือน (virtual address) เสมอทำให้โปรแกรมสามารถอ้างอิงได้ด้วย address เดิมตลอดเวลา

ในเรื่องของ security เป็นกรณีต่อเนื่องจาก relocatable คือ ทำอย่างไรให้ หน่วยความจำของ โปรแกรม A และ B เป็นอิสระจากกัน กล่าวคือ หน่วยความจำตำแหน่ง 100 ที่อ้างอิงในโปรแกรม A จะต้องไม่เกี่ยวข้องกับ หน่วยความจำตำแหน่ง 100 ที่อ้างอิงในโปรแกรม B มิใช่นั้น การทำงานของ โปรแกรม B พร้อมโปรแกรม A จะทำให้การประมวลผลผิดพลาดได้ ด้วยระบบ Virtual Address ทำให้ Address ของ A และ B เป็นอิสระจากกัน และแก้ปัญหาโดยปริยาย

7.12.1 การทำงานของ Virtual Memory

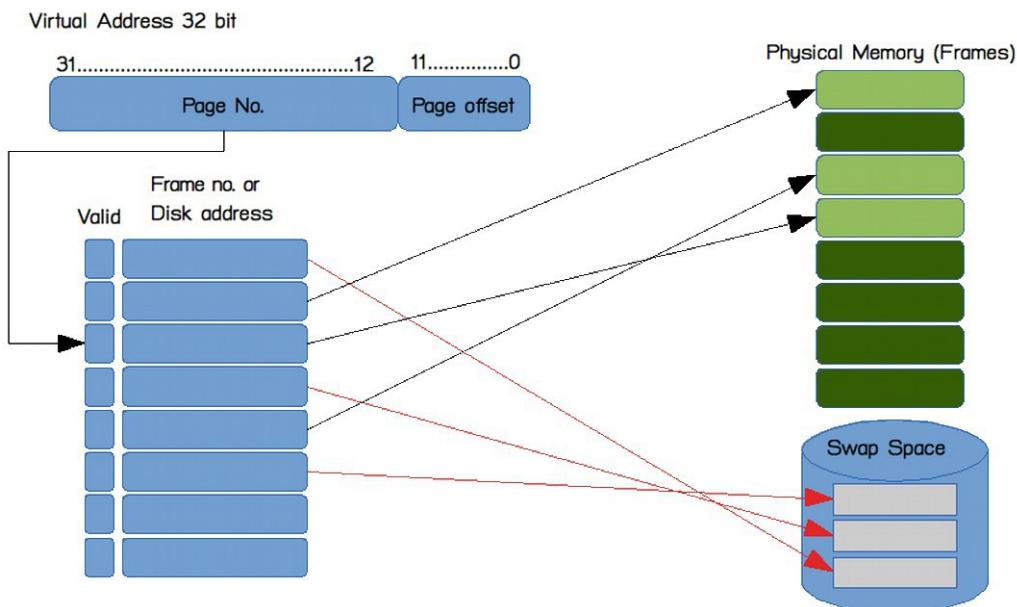
หลักการทำงานของ Virtual Memory มีการทำงานที่คล้ายคลึงกับ Cache บางส่วน เพียงแต่เปลี่ยน คำเรียกให้แตกต่างกัน เพื่อบังคับความสับสน ข้อแตกต่างประการสองประการที่เห็นได้ชัดเจนคือ 1) ในทางโครงสร้าง Cache จะเก็บข้อมูลในตารางโดยตรงเนื่องจากข้อมูลมีขนาดเล็กในขณะที่ Virtual Memory จะมีเพียง address สำหรับอ้างอิงข้อมูลกำหนดอยู่ในตารางเท่านั้น 2) Cache ทำงานโดย ฮาร์ดแวร์ทั้งหมด ในขณะที่ Virtual Memory จะต้องให้ระบบปฏิบัติการช่วยในการนำข้อมูลเข้าและ ออกจากระหว่างหน่วยความจำและ storage (disk) ที่ใช้จัดเก็บข้อมูล

เพื่อความสะดวกในการอธิบาย ขอเริ่มอธิบายคำศัพท์ที่แตกต่างกันระหว่างระบบ Cache และ ระบบ Virtual Memory โดยมีรายละเอียดดังนี้

- **Frame และ Page** เทียบได้กับ block ในระบบ Cache โดยทั่วไปในระบบ โดย Frame จะหมายถึง Physical Page หรือ Page ที่มีข้อมูลพร้อมสำหรับการอ้างอิงอยู่ในหน่วยความจำหลัก และ Page โดยทั่วไปจะหมายถึงข้อมูลอ้างอิงในหน่วยความจำเสมือน
- **Physical Address** คือ Address สำหรับอ้างอิงหน่วยความจำหลัก
- **Virtual Address** คือ Address สำหรับอ้างอิงหน่วยความจำที่ใช้ในโปรแกรม
- **Page Table** เทียบได้กับตารางเก็บค่า Tag, Valid Bit และ Dirty ของ Cache เพื่อช่วยในการค้นหาข้อมูลทำหน้าหลักในการแปลง 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 จะไม่มีตัวเทียบเดียวกันใน Cache หมายถึงที่จัดเก็บข้อมูลที่ยืมมาจากที่อื่น (โดยทั่วไปมักเป็นฮาร์ดดิสก์) สำหรับเป็นที่พักข้อมูลที่นำออกจากหน่วยความจำหลัก (ซึ่งจะอธิบายต่อไป)

เพื่อให้เห็นภาพการทำงานของหน่วยความจำเสมือน ลองวิเคราะห์การที่โปรแกรม ซึ่งต้องการใช้

หน่วยความจำเป็นจำนวนมาก แต่สามารถทำงานได้บนเครื่องที่มีหน่วยความจำน้อยໄດ້ (เช่น โปรแกรมที่ต้องการหน่วยความจำ 40 Mb ระหว่างการประมวลผล อาจจะทำงานได้บนเครื่องที่มีหน่วยความจำเพียง 16 Mb เป็นต้น) ซึ่งการทำงาน สามารถทำได้โดย การใช้งานหน่วยความจำหลัก ให้ทำหน้าที่คล้ายกับ Cache ทุกครั้งที่มีการอ้างอิงข้อมูล และนำข้อมูลทั้งหมดที่ต้องการอ้างอิง (ในกรณีนี้คือ 40 Mb) มาตัดเป็นชุดขนาดเท่ากัน (page) ทำการกำหนดเลขหน้า (page number) แล้วเก็บไว้ใน Swap Space (disk) ทุกครั้งที่ต้องการอ้างอิงข้อมูล (virtual address ที่ต้องการอ่านหรือเขียน) ก็จะเปิดตาราง (Page Table) เพื่อดูว่าข้อมูลที่ต้องการอยู่ที่ส่วนไหนของ disk (ดูที่ page number) และนำข้อมูลนั้นขึ้นมาใส่ในของหน่วยความจำหลัก (frame) ซึ่งอ้างอิงได้ด้วย physical address ในการอ้างอิงข้อมูลครั้งถัดไป จะทำการเปิดตารางเพื่อค้นหากรอบว่าข้อมูลอยู่ใน frame หรืออยู่ใน disk หากอยู่ใน frame ก็สามารถอ้างอิงด้วย physical address ที่ได้จาก page table ได้เลย การทำงานสามารถแสดงได้ดังรูปที่ 7.16

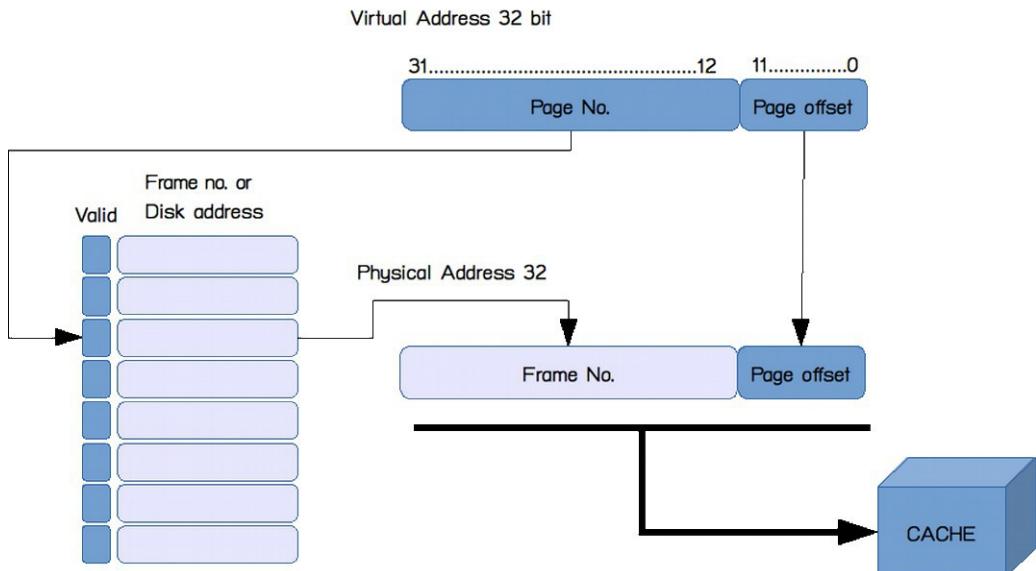


รูปที่ 7.16: การทำงานของ Virtual Memory

จากรูป จะสังเกตว่า Virtual Memory มักมีขนาดใหญ่กว่า Physical Memory ซึ่งบางส่วนจะเก็บอยู่ใน Physical Memory และบางส่วนจะเก็บอยู่ใน Disk (ในทางปฏิบัติ เครื่องคอมพิวเตอร์ขนาดใหญ่จะมีประสิทธิภาพสูงบางเครื่อง อาจมีหน่วยความจำมากพอ จนไม่ต้องมีการทำ swap เลยก็ได้) และ Valid บิต มีเพื่อแสดงว่าข้อมูลดังกล่าว อยู่ใน Swap Space หรือ Physical Memory (ในทางปฏิบัติ จะมี ข้อมูลมูลค่า 0 เพิ่มเติมจาก Valid บิตอีก เช่น Dirty บิต เพื่อบอกว่ามีการเขียนหรือไม่หรือมี Permission เพื่อบอกว่า Page ดังกล่าว สามารถใช้เพื่ออ้างได้บ้าง)

หากจะขยายความขั้นตอนการทำงานของการแปลง Virtual Address เป็น Physical Address ในรูปที่ 7.16 จะได้ดังรูปที่ 7.17 ซึ่งมีขั้นตอนการทำงานคือ นำค่า Page No. ไปมองหาข้อมูลในบรรทัดที่

เกี่ยวข้องเพื่อให้ได้ค่า Frame Number จากนั้นนำค่าดังกล่าวมารวมกับ Offset เพื่อใช้เป็น Physical Address สำหรับการอ่านหรือเขียนข้อมูลจาก Cache ต่อไป



รูปที่ 7.17: การแปลง Virtual Address เป็น Physical Address

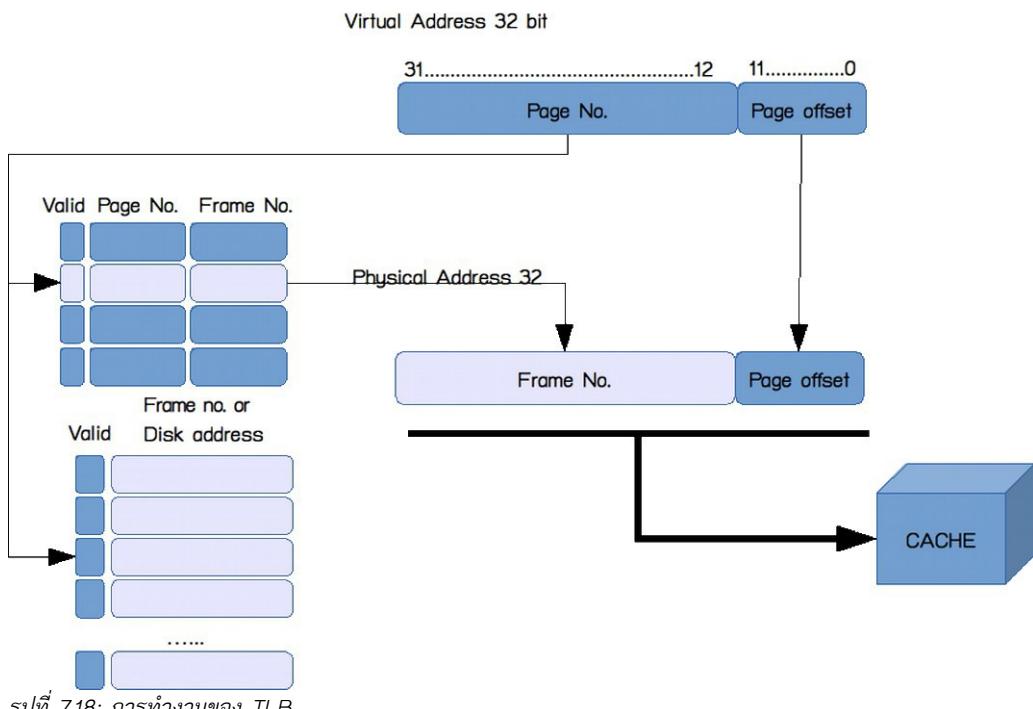
ในกรณีที่ข้อมูลที่ต้องการใช้งานอยู่ใน Swap Space หน่วยประมวลผลกลางจะทำการ Interrupt เพื่อให้ระบบปฏิบัติการดำเนินการนำข้อมูลเข้าออกจากรam ก่อน เนื่องจากในขั้นตอนการนำข้อมูลเข้าและออกจากหน่วยความจำ Physical Memory และ Disk Storage จะต้องถูกจัดการโดยระบบปฏิบัติการ เนื่องจากความต้องการในการใช้หน่วยความจำของแต่ละระบบ และการบริหารจัดการ Disk ของระบบปฏิบัติการมีความแตกต่างกัน หน่วยประมวลผลกลางจึงมีเพียง register พิเศษ ที่จะชี้ไปยังหน่วยความจำตำแหน่งที่เก็บตาราง Page Table และตรวจสอบว่าข้อมูลที่ต้องการอยู่ในหน่วยความจำหลัก (Physical Memory) หรือไม่ และกรณีที่เกิด Page Fault (ข้อมูลที่ต้องการมีได้อยู่ในหน่วยความจำหลัก) หน่วยประมวลผลกลางจะสร้างสัญญาณ Interrupt เพื่อให้ซอฟต์แวร์ทำงาน ซึ่งรายละเอียดเพิ่มเติมสามารถศึกษาได้จากวิชาระบบปฏิบัติการ Operating System

หากโครงสร้างของระบบ Virtual Address และ Physical เป็นดังรูปที่ 7.17 จะพบว่า Page Table ต้องมีจำนวนบรรทัด (Record(S)) ทั้งสิ้น 2^{20} Records (คิดจาก 2^{31-11}) ซึ่งคิดเป็น 1024×1024 บรรทัด หรือ 1 Mega entry และแต่ละ Record นั้นจะประกอบด้วย $(31-11) + 1$ บิต (เนื่องจาก Physical Page Number บอกด้วยบิตที่ 12-31 ของ Physical Address และแต่ละ Record จะต้องเก็บ Valid Bit) ซึ่งเมื่อแปลงข้อมูลเป็นหน่วย Byte จะต้องทำการบัดเตษษขึ้นเพื่อให้เป็นจำนวนเต็มได้ประมาณ 3 Byte ซึ่งในทางปฏิบัติ จะมีการเก็บ permission และข้อมูลสำหรับอธิบาย Page เพิ่มเติมด้วย จึงอาจได้ผลเป็น 4 Byte

ดังนั้นในกรณีนี้ Page Table จะมีขนาดใหญ่ถึง $4 \times 1240 \times 1024$ Byte หรือประมาณ 4Mb และ

เนื่องจาก Page Table นี้ก็ถูกเก็บไว้ในหน่วยความจำหลักเช่นกัน การหาข้อมูลจากตารางขนาด 4Mb ย่อมส่งผลให้การแปลงเลขดังกล่าวทำงานได้ช้า

เพื่อให้การแปลงเลข Virtual Page Number ไปเป็น Physical page Number สามารถกระทำได้รวดเร็วขึ้น จึงได้มีการประยุกต์ใช้ระบบ Cache กับ Page Table ด้วยเช่นกัน เพื่อไม่ให้สับสน จึงเรียก Cache ของ Page Table นี้ว่า Translation Look-aside Buffer (นิยมเรียกอยู่ว่า TLB) ดังแสดงในรูปที่ 7.18



รูปที่ 7.18: การทำงานของ TLB

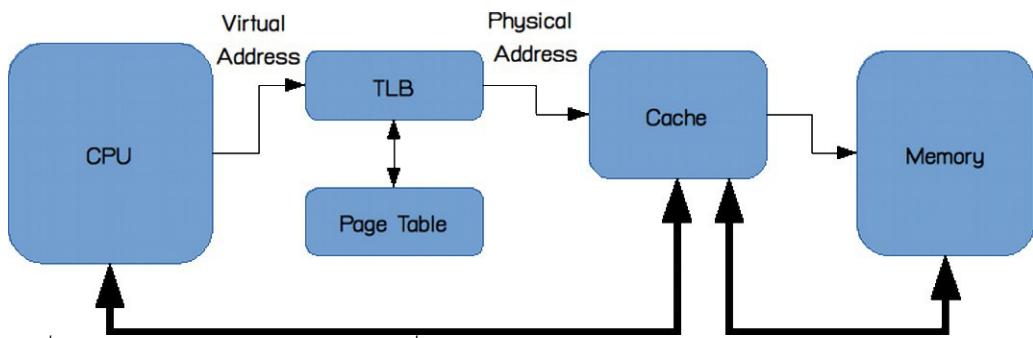
เนื่องจาก TLB มักมีขนาดเล็ก (ไม่เกิน 128 บรรทัด) เมื่อเทียบกับ Page Table ซึ่งมักมี 1 Mega entry หน่วยประมวลผลกลางส่วนใหญ่จึงเลือกที่จะออกแบบให้ TLB ทำงานแบบ Fully Associate

อีกประการหนึ่งคือ การ Write Through ในกรณีของ Page นั้นจะมีค่าใช้จ่ายที่สูงมาก (เนื่องจาก Storage มักจะทำงานช้ามาก) หน่วยประมวลผลกลางส่วนใหญ่จึงเลือกใช้วิธี Write Back เป็นหลัก

7.13 การทำงานร่วมกันของ Cache และ Virtual Memory

เมื่อระบบ Virtual Memory และระบบ Cache ทำงานพร้อมกันทั้งหมดภายในหน่วยประมวลผลกลาง ก็จะทำให้ขั้นตอนในการประมวลผลข้อมูลจากของหน่วยประมวลผลกลางนั้นชั้บช้อนยิ่งขึ้น โดยเริ่มตั้งแต่หน่วยประมวลผลกลางได้รับ Virtual Address จากชอฟต์แวร์ ก็จะทำการแปลงข้อมูลให้เป็น Physical Address โดยค้นหาจาก Page Table ซึ่งการค้นหานี้จะเริ่มต้นจาก Cache

ของ Translation Table ก่อน หากไม่พบก็จะทำการดึงข้อมูลจาก Page Table ตำแหน่งที่ต้องการเข้ามาไว้ใน Cache ของ Translation Table เมื่อได้ Physical Address แล้ว หากข้อมูลที่ต้องการไม่อยู่ภายใน Physical Memory ก็จะสร้างสัญญาณ Page Fault เพื่อให้ซอฟต์แวร์ระบบปฏิบัติการจัดการดึงข้อมูลเข้ามาไว้ใน Physical Memory ต่อไป เมื่อได้ Physical Address แล้ว หน่วยประมวลผลกลางจะค้นหาข้อมูลตำแหน่งดังกล่าวใน Cache ของข้อมูล และทำการอ่านหรือเขียนข้อมูล ส่วนกรณีการ Miss นั้น ระบบจัดการ Cache ข้อมูลจะทำการ Stall CPU แล้วดึงข้อมูลจากหน่วยความจำหลักเข้ามาเก็บใน Cache เพื่อประมวลผลตามขั้นตอน ขั้นตอนการทำงานแสดงได้ดังรูปที่ 7.19



รูปที่ 7.19: การเข้าถึงข้อมูลในหน่วยความจำเมื่อมี Cache และ Virtual Memory

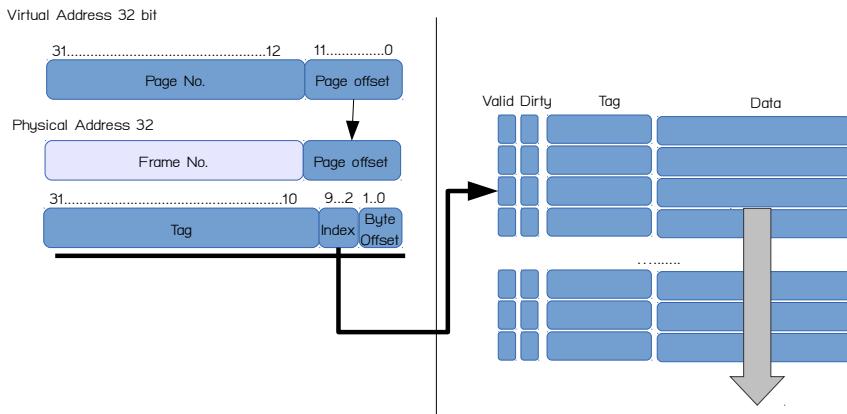
7.14 การเหลือมขั้นตอนของ Cache และ Virtual Memory

จากรูปที่ 7.19 จะพบว่าขั้นตอนการทำงานในการอ่านหรือเขียนข้อมูลนั้นค่อนข้างซับซ้อน และกินเวลา ดังนั้น จึงมีแนวคิดว่า เป็นไปได้หรือไม่ที่จะเหลือมขั้นตอนการทำงานของ Cache และ TLB (รวมถึง Page Table) ให้เร็วขึ้น ค่าตอบที่ได้คือ เป็นไปได้หากมีเงื่อนไขบางอย่างเป็นจริง ดังนี้

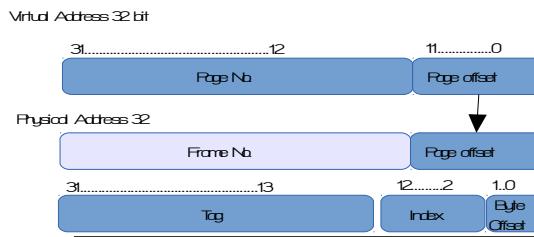
ลองพิจารณาการแปลง Address และการนำค่าที่ได้ไปอ้างอิงข้อมูลใน Cache ตามรูปที่ 7.20 หากการเปลี่ยนแปลง จาก Virtual Address ไปเป็น Physical Address ไม่ทำให้ค่า Index เพื่อใช้ในการค้นหาข้อมูลใน Cache เปลี่ยนแปลงไป (ค่า Cache Index อยู่ภายใต้ขอบเขตของ Page Offset) ย่อมเป็นไปได้ที่การหาข้อมูลใน Cache จะสามารถนำข้อมูล Page Offset จาก Virtual Address ไปใช้ได้ก่อน โดยไม่จำเป็นต้องรอให้มีการคำนวณค่า Physical Address (เพราะค่าที่ได้จะเหมือนเดิม)

อย่างไรก็ตาม ประเด็นหนึ่งที่ตามมาคือ หากต้องการเพิ่ม Cache ให้มีขนาดใหญ่มากขึ้นจนจำนวนบิตที่ได้เกินขอบเขตที่กำหนด จะทำอย่างไร จากตัวอย่าง (รูปที่ 7.21) จะเห็นว่าการเปลี่ยนแปลงจาก Page Number เป็น Frame Number อาจส่งผลให้ค่า Cache Index เปลี่ยนแปลงไปได้ กรณีนี้ หากยังคงต้องการใช้โครงสร้าง Cache ในลักษณะนี้ อาจจะต้องแก้ปัญหาด้วยวิธีการทางซอฟต์แวร์แทนเนื่องจาก ระบบปฏิบัติการเป็นผู้กำหนด Frame Number ให้ แต่ละ Page ดังนั้น หากสามารถบังคับให้การเปลี่ยนแปลงจาก Page Number เป็น Frame Number ไม่มีการเปลี่ยนแปลงนี้เกิดขึ้น ก็จะยัง

คงทำการเหลือมเวลาการเข้าถึงข้อมูลได้เหมือนเดิม



รูปที่ 7.20: การแปลง Virtual Address และการอ้างอิง Address ของ Cache



รูปที่ 7.21: โครงสร้าง Address ที่ไม่สามารถทำการเหลือมเวลาได้

ทั้งนี้ การบังคับด้วยซอฟต์แวร์ อาจจะไม่ใช่ทางออกที่ดีนัก ดังนั้น หากต้องการเพิ่มขนาดของ Cache โดยให้โครงสร้าง Address ในการอ้างอิงข้อมูลใน Cache ไม่เปลี่ยนแปลงไปจากเดิม ทางเลือกที่ดีกว่าคือ การเพิ่มด้วยการขยาย Associativity แทน (เช่น เพิ่มจาก 2-way set เป็น 4-way set) ด้วยวิธีการนี้ จะได้ Cache มาซึ่ง มี Conflict Miss น้อยลง และยังได้ประสิทธิภาพในการค้นหาข้อมูลโดยใช้ offset จาก Virtual Address ได้อีกด้วย

7.15 สรุป

ระบบ Virtual Memory และ ระบบ Cache เป็นระบบที่ช่วยเสริมประสิทธิภาพของหน่วยประมวลผลกลางในการทำงานที่เกี่ยวข้องกับหน่วยความจำ โดยระบบ Cache จะช่วยอำนวยความสะดวกให้หน่วยประมวลผลกลางสามารถอ่านข้อมูลจากหน่วยความจำหลักได้เร็วขึ้น ซึ่งผลที่ได้คือหน่วยประมวลผลกลางจะสามารถทำงานได้ที่ Clock Rate ที่สูงขึ้น เพราะไม่ต้องรอการทำงานของหน่วยความจำ อย่างไรก็ตามประสิทธิภาพของระบบนั้นจะดีหรือไม่ขึ้นอยู่กับค่า Miss Ratio

กับ ค่า Miss Penalty ด้วย สำหรับระบบ Virtual Memory นั้น ช่วยให้โปรแกรมเมอร์สามารถพัฒนาซอฟต์แวร์ได้เป็นอิสระโดยไม่ต้องสนใจระบบหน่วยความจำของเครื่องนั้น ๆ และยังให้ซอฟต์แวร์ที่ต้องการหน่วยความจำสูง สามารถทำงานบนเครื่องที่มีหน่วยความจำลักษณะนี้ ได้

ในด้านการทำงาน Cache เป็นการทำงานที่ไม่เข้ากับซอฟต์แวร์ (ทำงานได้โดยไม่ต้องมีซอฟต์แวร์ช่วย) ในขณะที่ Virtual Memory จะต้องอาศัยการทำงานของหน่วยประมวลผลกลางร่วมด้วย

ในหนังสือเล่มนี้ไม่ได้กล่าวถึงหน่วยความจำเสมือนในลักษณะอื่น (เช่น segmentation หรือ Overlay) เพื่อเป็นการจัดการที่ไม่ได้เป็นที่นิยมใช้แพร่หลายทั่วไป แต่หากผู้เรียนสนใจจะเอิดทางด้านนี้ ขอแนะนำให้ศึกษาเพิ่มเติมยิ่งขึ้นไปอีก

7.16 แบบฝึกหัดท้ายบท

1. หลักการของ Locality ช่วยให้หน่วยประมวลผลกลางสามารถทำงานเร็วขึ้นได้อย่างไร
2. จงอธิบายหลักการทำงานของ Temporal Locality และ Spatial Locality พร้อมยกตัวอย่างประกอบคำอธิบาย
3. หากท่านเป็นผู้ออกแบบ Cache ท่านจะมีหลักเกณฑ์อย่างไรในการเลือกใช้ระบบ Write Back และ Write Through อย่างไร จงให้เหตุผลประกอบคำอธิบาย
4. มีปัจจัยใดบ้างที่มีผลต่อประสิทธิภาพของ Cache และเราสามารถปรับปรุงประสิทธิภาพของ Cache ได้อย่างไรบ้าง จงอธิบาย
5. โปรแกรมทดสอบอันหนึ่งมีชุดคำสั่งทั้งสิ้น 2 ล้านคำสั่ง หาก Miss Rate เป็น 10% และ Miss Penalty เป็น 6 Cycle หากหน่วยประมวลผลกลางดังกล่าวมี CPI เป็น 1 และมี Clock Rate เป็น 200Mhz จงแสดงการคำนวนเวลาในการประมวลผลโปรแกรมทดสอบของหน่วยประมวลผลกลางดังกล่าว
6. นักออกแบบท่านหนึ่งได้ให้ความเห็นว่า การทำ Multi-level Cache เป็นการ tradeoff ระหว่าง Hit Time และ Miss Penalty ท่านเห็นด้วยหรือไม่ อย่างไร
7. โดยปกติการทำงานของ Cache จะไม่เข้าอยู่กับโปรแกรม (กล่าวคือ โปรแกรมสามารถได้ประโยชน์จากการทำงานของ Cache โดยไม่ต้องทำการแก้ไขอะไร) อย่างไรก็ตาม หากพัฒนาซอฟต์แวร์ที่ต้องสร้างการทำงานของ Cache จะสามารถพัฒนาโปรแกรมเพื่อให้ใช้ประโยชน์จาก Cache ได้ดีขึ้นหรือไม่ อย่างไร จงยกตัวอย่างประกอบคำอธิบาย
8. หากกำหนดโปรแกรมให้ ท่านมีแนวทางในการวิเคราะห์ขนาดของ Cache ขั้นต่ำ ที่เหมาะสมกับการทำงานของโปรแกรมนี้อย่างไร
9. หากเครื่องคอมพิวเตอร์ระบบหนึ่งซึ่งมี Cache ทำงานที่ความเร็ว 2 ns และ หน่วยความจำลักษณะที่ 200 ns แล้ว การเพิ่ม Cache อีกระดับหนึ่งซึ่งมีความเร็ว 20 ns แทรกเข้าไป จะช่วยเพิ่มประสิทธิภาพได้หรือไม่ อย่างไร จงแสดงการคำนวนประกอบการอธิบาย
10. จงแสดงการออกแบบโครงสร้างภายในของ Cache แบบ 8-way set associative (fully associative)
11. จงเขียนโปรแกรม (ด้วยภาษาอังกฤษได้) เพื่อจำลองการทำงานของ Cache ที่มี Replacement Algorithm เป็นแบบ LRU และ RR จากนั้น ให้ลองนำ address ตัวอย่างจากโปรแกรมที่นำไป (หา download ได้ที่ไป) ลงทำการประเมินประสิทธิภาพของ replacement algorithm ทั้งสองแบบ ว่าแบบใดให้ผลดีกว่ากัน

12. การมีระบบ Virtual Memory นั้น เป็นประโยชน์สำหรับผู้พัฒนาซอฟต์แวร์หรือไม่ อย่างไร
13. ในหน่วยประมวลผลกลางหนึ่ง มี Page ขนาด 4 Kbyte ผู้ออกแบบหน่วยประมวลผลกลาง ต้องการสร้าง Cache ขนาด 32 Kbyte เพื่อช่วยให้การอ้างอิงข้อมูลเร็วขึ้น หากต้องการ ออกแบบให้มีการเหลือมการทำงานของ Cache และ การแปลง Virtual Address เป็น Physical Address เพื่อเพิ่มประสิทธิภาพด้วย จะเป็นไปได้หรือไม่ อย่างไร จงแสดงการ ออกแบบ ประกอบการให้เหตุผล

บรรณานุกรม

- [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. Comput.*, 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.