

บทที่ 1 บทนำ

สถาปัตยกรรมคอมพิวเตอร์นั้น หมายถึงโครงสร้างและการทำงานของเครื่องคอมพิวเตอร์ โครงสร้างของโพรเซสเซอร์(หน่วยประมวลผลกลาง) การแทนข้อมูลระดับเครื่อง ภาษาเครื่อง ภาษาแอสเซมบลี สถาปัตยกรรมและการจัดการระบบหน่วยความจำ การเชื่อมต่อระหว่างเครื่องคอมพิวเตอร์และอุปกรณ์ต่อพ่วงต่างๆ รวมถึงโครงสร้างการทำงานของระบบคอมพิวเตอร์แบบขนานอื่นๆ ทั้งนี้ เราสามารถกล่าวโดยสรุปได้ว่า สถาปัตยกรรมคอมพิวเตอร์ คือ ความสามารถ (Function) ในการทำงานของโพรเซสเซอร์ ซึ่งมีความหมายโดยนัยคือ ชุดคำสั่งต่างๆ ที่โพรเซสเซอร์สามารถประมวลผลได้นั่นเอง

เนื้อหาในเอกสารชุดนี้ บางส่วนได้คัดมาจากหนังสือ John L Hennessy, D. A Patterson, "Computer Architecture : A Quantitative Approach 2nd Edition", Morgan Kaufmanns Pub. และบางส่วนเป็นเนื้อหาที่มีการอ้างอิงเพิ่มเติมจากแหล่งความรู้อื่น เพื่อประกอบการอธิบายให้เข้าใจง่ายขึ้น ทั้งนี้สถาปัตยกรรมคอมพิวเตอร์มีการพัฒนา ปรับปรุงและเปลี่ยนแปลงอยู่เสมอ ผู้ศึกษาควรค้นคว้าเพิ่มเติมจากแหล่งความรู้อื่นๆ ประกอบด้วย

โครงสร้างและสถาปัตยกรรมคอมพิวเตอร์ที่อธิบายในเอกสารชุดนี้ จะอาศัยพื้นฐานจากหน่วยประมวลผลกลาง MIPS เป็นหลัก ทั้งนี้เนื่องจาก MIPS เป็นสถาปัตยกรรมคอมพิวเตอร์ประเภท RISC ที่เข้าใจง่าย ไม่ซับซ้อน เหมาะกับการศึกษาในระดับเบื้องต้น นอกจากนี้อาจมีการแทรกเนื้อหาของสถาปัตยกรรมอื่นๆ เพื่อประกอบการอธิบายเพิ่มเติมบ้าง ทั้งนี้ขึ้นกับความต่อเนื่องและความเหมาะสมของเนื้อหา

การศึกษาในวิชาสถาปัตยกรรมคอมพิวเตอร์นั้น ผู้เรียนควรมีทักษะในการออกแบบวงจรตรรกะ หรือ วงจรดิจิทัล โดยความรู้ทางวงจรดิจิทัลนั้น ควรจะสามารถออกแบบได้ทั้งวงจรแบบผสม (Combination Logic Circuit) และ วงจรแบบลำดับ (Sequential Circuit) นอกจากนี้ ผู้เรียนควรมีประสบการณ์หรือเคยศึกษาภาษาแอสเซมบลีของหน่วยประมวลผลกลางตัวใดตัวหนึ่งมาก่อน

โครงสร้างและส่วนประกอบของเครื่องคอมพิวเตอร์

ระบบคอมพิวเตอร์โดยทั่วไปนั้น มักประกอบด้วยหน่วยประมวลผลกลาง Keyboard จอภาพ Monitor เครื่องพิมพ์ ระบบ Multimedia และอื่นอีกมากมาย ทั้งนี้โครงสร้างดังกล่าวสามารถจำแนกตามการทำงานได้เป็น

- หน่วยประมวลผลกลาง (CPU)
- อินพุต (Keyboard, Mouse, etc....)

- เอาท์พุต (Display, Printer, etc...)
- หน่วยความจำ (Ram, Disk Drive, CD, etc...)
- ระบบเครือข่าย

อย่างไรก็ตามส่วนประกอบต่างๆ เหล่านี้จะถูกเชื่อมต่อเข้ากับหน่วยประมวลผลกลาง โดยอาศัยการสื่อสารและควบคุม ผ่านระบบสายสัญญาณต่างๆ (Data path, Control) ซึ่งการทำงานต่างๆ เหล่านี้เป็นสิ่งที่ต้องกล่าวถึงในสถาปัตยกรรมคอมพิวเตอร์

สถาปัตยกรรมชุดคำสั่ง (Instruction Set Architecture)

การพัฒนาซอฟต์แวร์โดยทั่วไปนั้น ผู้พัฒนาหรือโปรแกรมเมอร์ มักจะพัฒนาโปรแกรมโดยใช้ภาษาชั้นสูง (เช่น ภาษา C) จากนั้นโปรแกรมจะถูกแปลโดยคอมไพเลอร์เป็นภาษาแอสเซมบลี และโปรแกรมภาษาแอสเซมบลีเหล่านั้น ก็จะถูกละเอียดโดยแอสเซมเบลอเป็น ภาษาเครื่อง หรือ Machine Code ในที่สุด (ดังแสดงในภาพข้างล่าง)

High-level
language
program
(in C)

```
swap(int v[], int k){
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

C compiler

Assembly
language
program
(for MIPS)

```
swap:
    muli $2, $5, 4
    add $2, $4, $2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31
```

Assembler

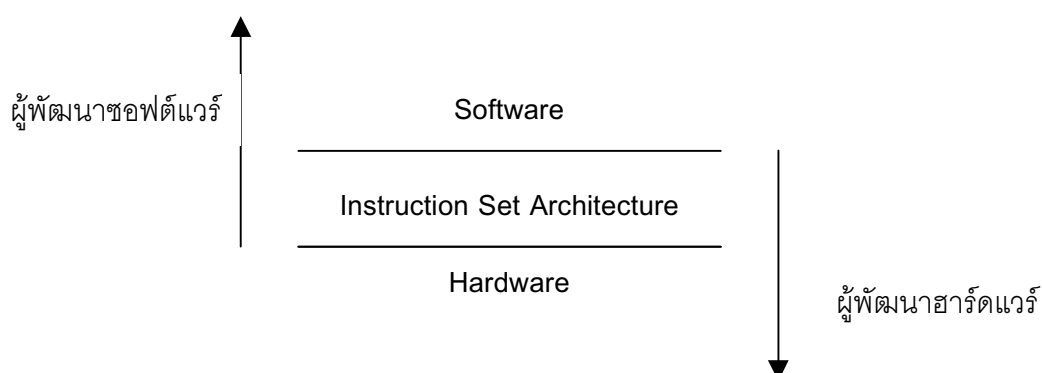
Binary machine
language
program
(for MIPS)

```
000000001010000100000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

จากลักษณะการพัฒนาซอฟต์แวร์ดังกล่าว พบว่าโปรแกรมที่มีต้นฉบับภาษาซีเหมือนกัน นั้น เมื่อถูกแปลโดย คอมไพเลอร์ และ แอสเซมเบลอร์ แล้ว ภาษาเครื่องดังกล่าวจะถูกนำไปประมวลผลโดยหน่วยประมวลผลกลาง เราจึงอาจกล่าวได้ว่า ชุดคำสั่งต่างๆ เหล่านี้ เป็น ตัวกลาง ในการเชื่อมต่อระหว่างฮาร์ดแวร์ และ ซอฟต์แวร์ เพื่อให้สั่งให้เครื่องคอมพิวเตอร์ทำงานตาม โปรแกรมที่ต้องการ เราเรียกชุดคำสั่งต่างๆ ในภาษาแอสเซมบลีและภาษาเครื่อง เหล่านี้ว่า สถาปัตยกรรมชุดคำสั่ง ซึ่งจะใช้อธิบายถึงความสามารถในการทำงานของหน่วยประมวลผลกลาง นี้

ในการพัฒนาซอฟต์แวร์ โดยเฉพาะอย่างยิ่งซอฟต์แวร์ระบบ ผู้พัฒนาจึงควรมีความรู้เกี่ยวกับสถาปัตยกรรมชุดคำสั่งต่างๆ นี้ เพื่อที่จะสามารถพัฒนาซอฟต์แวร์ได้โดยไม่จำเป็นต้องอาศัย ความรู้ทางด้านฮาร์ดแวร์ ทั้งนี้หน่วยประมวลผลกลางที่ต่างกัน อาจมีสถาปัตยกรรมชุดคำสั่งที่ เหมือนกันก็ได้ ซึ่งนั่นหมายความว่าซอฟต์แวร์ที่ทำงานบนระบบคอมพิวเตอร์เครื่องหนึ่ง จะสามารถ ทำงานได้บนระบบคอมพิวเตอร์ทุกระบบที่มีโครงสร้างสถาปัตยกรรมชุดคำสั่งเหมือนกัน

กรณีการพัฒนาฮาร์ดแวร์นั้น ผู้พัฒนาฮาร์ดแวร์ เพียงออกแบบสถาปัตยกรรมชุดคำสั่ง และสร้างวงจรประมวล ให้สามารถทำงานได้ตามสถาปัตยกรรมนั้น โดยไม่ต้องคำนึงถึงการพัฒนา ซอฟต์แวร์ใดๆ



แม้ว่าการอ้างอิงถึงสถาปัตยกรรมคอมพิวเตอร์โดยใช้ สถาปัตยกรรมชุดคำสั่งเป็นหลัก จะเป็นประโยชน์ในการพัฒนาซอฟต์แวร์ และ ฮาร์ดแวร์ก็ตาม แต่บางกรณี การพัฒนาซอฟต์แวร์และ ฮาร์ดแวร์โดยยึดติดกับสถาปัตยกรรมชุดคำสั่งมากเกินไป ทำให้ไม่สามารถคิดค้นหรือพัฒนา สถาปัตยกรรมคอมพิวเตอร์แบบใหม่ๆ ขึ้นมาได้

หน่วยประมวลผลกลาง

หน่วยประมวลผลกลางหรือ CPU นั้น เป็นศูนย์กลางในการทำงานของระบบคอมพิวเตอร์ ทุกสิ่งทุกอย่างถึงในสถาปัตยกรรมชุดคำสั่งนั้น หน่วยประมวลผลกลางจะต้องมีความสามารถทำงานได้ตามที่ระบุไว้ หากเปรียบเทียบกับร่างกายเราอาจกล่าวได้ว่าหน่วยประมวลผลกลางเป็นสมองมนุษย์เรา ทั้งนี้ภายในหน่วยประมวลผลกลางจะประกอบไปด้วย

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

เหตุที่หน่วยประมวลผลกลางจำเป็นจะต้องมีความสามารถในการจำนั้น เพื่อใช้ประกอบการคิดคำนวณต่างๆ (ซึ่งเมื่อเปรียบเทียบกับความคิดคำนวณต่างๆ ของมนุษย์ จะพบว่าการคิดคำนวณของมนุษย์ต้องใช้ความจำเช่นกัน เช่นกรณีการบวกเลขแบบเด็กๆ หากเราต้องการนำ $2+3$ เรายังจะสอนให้เด็กนำ 2 ไว้ในใจ ชูขึ้นมา 3 นิ้วแล้วนับต่อจนครบ ซึ่งจะพบว่ามีการใช้ความจำควบคู่กับการคำนวณด้วยเสมอ)

แบบฝึกหัดท้ายบท

1. สถาปัตยกรรมคอมพิวเตอร์คืออะไร
2. ส่วนประกอบของคอมพิวเตอร์ มีอะไรบ้าง
3. ในการพัฒนาโปรแกรมด้วยภาษาชั้นสูง เรามีขั้นตอนในการพัฒนาโปรแกรมต่างๆ เหล่านั้นอย่างไร
4. สถาปัตยกรรมชุดคำสั่งคืออะไร การกล่าวถึงสถาปัตยกรรมชุดคำสั่งดังกล่าว เป็นประโยชน์ในการพัฒนาซอฟต์แวร์และฮาร์ดแวร์หรือไม่ อย่างไร
5. จงสร้างวงจรบวกเลขที่จะบวกเลขขนาด 2 bit และ ตัวทดอีก 1 บิต แล้วให้คำตอบเป็นผลลัพธ์พร้อมตัวทด
6. จงสร้างวงจรบวกเลขแบบอนุกรม (Serial Adder) แบบ Moore และ Mealy Machine

บทที่ 2 ประสิทธิภาพของระบบคอมพิวเตอร์

ประสิทธิภาพ คือ การวัดและการเปรียบเทียบการทำงาน ทั้งนี้ในการศึกษาเรื่องประสิทธิภาพของระบบคอมพิวเตอร์นั้น เรามุ่งเน้นถึงปัจจัยต่างๆ ที่ส่งผลกระทบต่อการทำงานของระบบคอมพิวเตอร์ โดยมุ่งให้สามารถเลือกใช้ หรือ รู้จักการวัดประสิทธิภาพของระบบ เพื่อใช้อธิบายในหรือหาส่วนถัดไปว่า การพัฒนาประสิทธิภาพของระบบคอมพิวเตอร์ด้วยวิธีการต่างๆ นั้น ให้ผลทางปฏิบัติที่ดีขึ้นอย่างไร

การวัดประสิทธิภาพของระบบคอมพิวเตอร์

การวัดประสิทธิภาพของระบบคอมพิวเตอร์นั้น เราใช้เวลาเป็นหลักในการวัด โดยระบบคอมพิวเตอร์ที่ใช้เวลาในการทำงานโปรแกรมหนึ่งๆ น้อยกว่า จะมีประสิทธิภาพมากกว่าระบบคอมพิวเตอร์ที่ทำงานโปรแกรมเดียวกัน แต่ใช้เวลาในการทำงานมากกว่า หรือเราอาจกล่าวอีกนัยหนึ่งได้ว่า ประสิทธิภาพนั้น เป็นส่วนกลับของเวลา

$$[\text{Performance}] = 1 / [\text{Execution Time}]$$

การกล่าวว่า “เครื่องคอมพิวเตอร์ X เร็วกว่าเครื่อง Y เป็น n เท่า” หรือ “เครื่องคอมพิวเตอร์ X มีประสิทธิภาพเหนือความเครื่อง Y เป็น n เท่า” นั้น สามารถอธิบายเป็นสมการได้ว่า

$$n = [\text{Performance}_x] / [\text{Performance}_y]$$

หรือ

$$n = [\text{Execution Time}_y] / [\text{Execution Time}_x]$$

ทั้งนี้เนื่องจากเวลาในการทำงานของระบบคอมพิวเตอร์ เพื่อประมวลผลโปรแกรมใดๆ นั้น อาจหมายถึงเวลาในการตอบสนองของเครื่องคอมพิวเตอร์นั้นๆ (Response Time) เช่น เมื่อผู้ใช้สั่งให้โปรแกรมค้นหาข้อมูล โปรแกรมใช้เวลานานเท่าใด หลังจากเริ่มค้นหา และได้ผลลัพธ์แสดงผลออกมา หรือ อาจหมายถึง ความสามารถในการประมวลผลงานต่างๆ ในช่วงเวลาหนึ่ง (Throughput) เช่น เครื่องคอมพิวเตอร์ สามารถประมวลผลโปรแกรมได้ 2 โปรแกรม ภายในเวลา 1 วินาที

นอกจากนี้ หากเราวิเคราะห์การทำงานของซอฟต์แวร์ต่างๆ ภายในเครื่องคอมพิวเตอร์ เราจะพบว่า เครื่องคอมพิวเตอร์อาจจะไม่ได้ใช้เวลาทั้งหมดไปกับการประมวลผล แต่เวลาบางส่วนอาจสูญเสียไปเนื่องจากการทำงานของฮาร์ดแวร์ส่วนอื่นๆ เช่น การอ่านข้อมูลจากหน่วยความจำหรือระบบ I/O อื่นๆ ในการวิเคราะห์ Response Time หรือ Execution Time จึงอาจแบ่งออกได้เป็น

- Elapse Time คือเวลาในการประมวลผลโปรแกรมหนึ่งๆ ของระบบคอมพิวเตอร์ ซึ่งรวมถึงเวลาที่ใช้ในการอ่านข้อมูลจาก I/O และ หน่วยความจำ
- CPU Time คือเวลาที่ CPU ทำงานเพื่อประมวลผลโปรแกรมหนึ่งๆ โดยไม่นับรวมเวลาจากการรอคอยการทำงานจากอุปกรณ์ต่อพ่วงที่อยู่ภายนอกหน่วยประมวลผลกลาง

ทั้งนี้ ในการวัดประสิทธิภาพของระบบคอมพิวเตอร์นั้น เรานิยมวัดด้วย CPU Time แทนการวัดด้วย Elapse Time ทั้งนี้เนื่องจาก หากวัดประสิทธิภาพโดยใช้ Elapse Time ในการอ้างอิงค่าของประสิทธิภาพที่วัดได้อาจขึ้นอยู่กับความเร็วของการเข้าถึงข้อมูลในหน่วยความจำ หรือความเร็วในการตอบสนองของอุปกรณ์ต่อพ่วงต่าง ทำให้ค่าที่วัดได้ดังกล่าว ไม่สามารถวิเคราะห์ผลเชิงเปรียบเทียบความสามารถของหน่วยประมวลผลกลางได้ (จากข้อมูลทางสถิติพบว่า เวลาในการประมวลผลของโปรแกรมนั้น มากกว่า 45% เป็นเวลาที่เสียกับการทำงานของ I/O หรือ ส่วนอื่นๆ ซึ่งไม่เกี่ยวข้องกับการทำงานของโปรแกรม)

จากแนวทางในการวัดประสิทธิภาพของระบบคอมพิวเตอร์ดังกล่าว เราจะพบว่าการวัดประสิทธิภาพของระบบคอมพิวเตอร์นั้นจะต้องอาศัยโปรแกรมทดสอบ (Benchmark) และวัดเวลาที่ใช้ในการทำงาน ทั้งนี้โปรแกรมที่จะใช้ทดสอบ อาจเป็นโปรแกรมใดๆ ก็ได้ แต่หากผู้พัฒนาแต่ละรายทำการทดสอบประสิทธิภาพของระบบคอมพิวเตอร์ โดยใช้โปรแกรมทดสอบของตนเอง เวลาหรือผลลัพธ์ที่ได้จากการทดสอบ อาจไม่สามารถเปรียบเทียบกันได้ ดังนั้น จึงได้มีการรวมกลุ่มกันระหว่างผู้ผลิตหน่วยประมวลผลกลางต่างๆ เพื่อกำหนดประเภทของซอฟต์แวร์ที่เหมาะสมสำหรับการวัดประสิทธิภาพของ ระบบคอมพิวเตอร์ (SPEC Benchmark suites) ซอฟต์แวร์ในชุด SPEC นั้นประกอบด้วย ซอฟต์แวร์ที่ใช้ความสามารถของหน่วยประมวลผลค่อนข้างสูง เช่น โปรแกรมวิเคราะห์ทางคณิตศาสตร์ หรือ วิทยาศาสตร์ คอมไพเลอร์ที่มีความสามารถสูง เป็นต้น โดยรายละเอียดเพิ่มเติมสามารถค้นคว้าได้จาก SPEC Benchmark Suites ซึ่งเผยแพร่โดย System Performance Evaluation Cooperative

ตัวอย่าง

จากการทดลองวัดเวลาในการประมวลผลของหน่วยประมวลผลกลาง ในการทำงานของ Benchmark โปรแกรมหนึ่ง ผลที่ได้เป็นดังนี้

เครื่อง A ใช้เวลาในการทำงาน 20 วินาที

เครื่อง B ใช้เวลาในการทำงาน 25 วินาที

เครื่องคอมพิวเตอร์เครื่องใดมีประสิทธิภาพมากกว่ากัน และ มากกว่าเป็นกี่เท่า

เครื่อง A มีประสิทธิภาพมากกว่าเครื่อง B เนื่องจากสามารถประมวลผลโปรแกรมทดสอบได้เร็วกว่า

จาก

$$n = [\text{Execution Time}_y] / [\text{Execution Time}_x]$$

$$= 25/20 = 1.25$$

∴ เครื่อง A มีประสิทธิภาพมากกว่าเครื่อง B อยู่ 1.25 เท่า

การหาเวลา CPU TIME ของโปรแกรมต่างๆ

ในทางปฏิบัตินั้น เราไม่สามารถวัดหรือจับเวลา CPU TIME ของการประมวลโปรแกรมต่างๆ ได้ เพราะเราไม่ทราบเวลาที่ใช้อย่างกล่าว เป็น CPU TIME หรือ เป็นเวลาการทำงานที่รอจาก I/O ดังนั้น การหา CPU TIME จึงอาจทำได้โดยการประมาณหรือการนับจำนวนคำสั่งที่ใช้ในการประมวลผล และ หาผลรวมของเวลาที่ใช้ในการประมวลผลคำสั่งทั้งหมด

ตัวอย่าง

เครื่องคอมพิวเตอร์เครื่องหนึ่งมีเวลาในการประมวลผลคำสั่งแต่ละคำสั่งเป็น 0.1 วินาที หากนำโปรแกรมที่มีจำนวนคำสั่ง 10 คำสั่งมาทำการประมวลผลบนเครื่องดังกล่าว จะใช้เวลาในการทำงานเป็นเท่าไร

$$\therefore \text{เวลาในการทำงานทั้งหมด} = 10 * 0.1$$

อย่างไรก็ตาม การบอกเวลาในการประมวลผลของคำสั่งแต่ละคำสั่งนั้น ไม่นิยมบอกเป็นหน่วยวินาที ทั้งนี้เนื่องจากเวลาในการประมวลผลของหน่วยประมวลผลนั้น จะขึ้นอยู่กับสัญญาณ

นาฬิกาที่ป้อนให้กับหน่วยประมวลผลนั้น (Clock Rate) ซึ่งช่วงเวลาเหล่านี้ นิยมเรียกว่า ticks หรือ clock cycles โดย

$$[\text{Clock Cycle Time}] = 1 / [\text{Clock Rate}]$$

เช่น หน่วยประมวลผลที่มี Clock Rate เป็น 250Mhz จะมี Clock Cycle Time เป็น 4 ns เป็นต้น

นอกจากนี้การประมวลผลคำสั่งแต่ละคำสั่ง อาจมิได้ใช้เวลาเพียง 1 Clock Cycle Time เท่านั้น จำนวน Cycle ที่ใช้ในการประมวลผลแต่ละคำสั่ง (Cycle Per Instruction หรือ CPI) อาจแตกต่างกันไป เช่นการประมวลผลคำสั่งบวกอาจใช้เวลา 1 CPI และการประมวลผลคำสั่งคูณอาจใช้เวลา 3 CPI ดังนั้นหาก หน่วยประมวลผลกลางดังกล่าว มีความถี่สัญญาณนาฬิกาเป็น 150Mhz เวลาที่ใช้ในการทำคำสั่งบวก จะเป็น 4 ns (1×4) และเวลาที่ใช้ในการทำคำสั่งคูณเท่ากับ 12 ns (3×4) เป็น

เมื่อนำแนวความคิดทั้งหมดมาประสานเข้าด้วยกัน สามารถสรุปเป็นสมการแสดงความสัมพันธ์ระหว่างเวลาที่ใช้ในการประมวลผล และ ความถี่สัญญาณนาฬิกาได้ดังนี้

$$[\text{เวลาที่ใช้ในการประมวลผลแต่ละคำสั่ง}] = [\text{CPI ของคำสั่งนั้น}] * [\text{Clock Cycle Time}]$$

หรือ

$$[\text{Seconds / Program}] = [\text{Clock cycles / Instruction}] * [\text{Seconds / Cycle}]$$

ดังนั้นเวลาในการทำงานของ CPU จะเป็น

$$[\text{Execution Time}] = [\text{จำนวนคำสั่ง}] * [\text{เวลาที่ใช้ในการประมวลผลแต่ละคำสั่ง}]$$

จากสมการดังกล่าวพบว่าเราสามารถปรับปรุงเวลาในการทำงานแต่ละคำสั่งได้โดยการลดค่า CPI หรือ ลดค่า Clock Cycle Time หรือ ลดจำนวนคำสั่งลง ดังนี้

- Clock Cycle Time หรือ Clock Rate ขึ้นอยู่กับโครงสร้างภายในฮาร์ดแวร์ และ เทคโนโลยีในการผลิต
- CPI ขึ้นอยู่กับ โครงสร้างภายในฮาร์ดแวร์ และ โครงสร้างสถาปัตยกรรมชุดคำสั่ง

- จำนวนคำสั่ง ขึ้นอยู่กับ โครงสร้างสถาปัตยกรรมชุดคำสั่งและความสามารถในการแปลและ Optimize ของตัวแปลภาษา

ตัวอย่าง

เครื่องคอมพิวเตอร์เครื่องหนึ่งมีความถี่สัญญาณนาฬิกา 100Mhz ใช้เวลาในการประมวลผลโปรแกรมทดสอบซึ่งมีจำนวนคำสั่งทั้งสิ้น 2000 ล้านคำสั่ง เป็นเวลาทั้งสิ้น 20 วินาที หากผู้ออกแบบสามารถพัฒนาให้หน่วยประมวลผลกลางใช้ความถี่สัญญาณนาฬิกาเป็น 200 Mhz แต่ใช้จำนวน CPI มากขึ้นเป็น 1.2 เท่า หากนำโปรแกรมสอบนี้มาทดสอบในหน่วยประมวลผลที่ทำการพัฒนา จะใช้เวลาในการประมวลผลเป็นเท่าไร (กำหนดให้แต่ละคำสั่งใช้ เวลาในการประมวลผลเท่ากัน)

ความถี่สัญญาณนาฬิกา 100 Mhz คิดเป็น Clock Cycle Time = $1/(100 \times 10^6)$ sec

จาก $[CPU\ TIME] = [จำนวนคำสั่ง] \times [CPI] \times [Clock\ Cycle\ Time]$

$$[CPI] = 20 \times 10^8 / 2000 \times 10^6$$

$$\therefore [CPI] = 1\ Cycle$$

การพัฒนาหน่วยประมวลผล ทำให้ใช้จำนวน CPI ใหม่มากขึ้น 1.2 เท่า

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

เวลาที่ใช้ในการประมวลผลโปรแกรมทดสอบหลังจากพัฒนาหน่วยประมวลผลกลาง

$$[CPU\ TIME_{ใหม่}] = 2000 \times 10^6 \times 1.2 \times 1 / (200 \times 10^6)$$

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

ในบางกรณีเรานิยามที่จะกล่าวถึงความเร็วในการประมวลผลทั่วไป โดยใช้หน่วย MIPS (Million of Instructions per Seconds) เช่น โปรแกรมมีการทำงานทั้งสิ้น 2000 ล้านคำสั่งภายในเวลา 20 วินาที อาจกล่าวได้ว่า ความเร็วในการประมวลผลของหน่วยประมวลผลดังกล่าวเป็น $2000/20 = 100$ MIPS เป็นต้น ซึ่งมีความหมายว่า หน่วยประมวลผลดังกล่าวสามารถประมวลผลได้ 100 ล้านคำสั่ง ภายในเวลา 1 วินาที

กฎของ Amdahl

ในการปรับปรุงประสิทธิภาพส่วนหนึ่งส่วนใดของคอมพิวเตอร์นั้น ผลที่ได้จากการปรับปรุง อาจมิได้ส่งผลให้ประสิทธิภาพที่ได้เพิ่มขึ้นแบบเชิงเส้น เช่น การเพิ่มประสิทธิภาพในการบวกเลขของหน่วยประมวลผลกลางให้เร็วขึ้น 2 เท่า มิได้หมายความว่าเมื่อนำโปรแกรมทดสอบเข้าไปทดลอง

ประมวล จะได้ประสิทธิภาพเพิ่มขึ้นเป็น 2 เท่า ทั้งนี้เนื่องจากเวลาในการประมวลผลทั้งหมดนั้น ไม่ใช่เวลาที่เกิดขึ้นจากการบวกเลข หากแต่อาจเป็นการคูณเลข การย้ายข้อมูล หรือ การเปรียบเทียบทางตรรกะก็ได้

กฎของ Amdahl ได้กำหนดการวัด Speedup หรือประสิทธิภาพที่เพิ่มขึ้นไว้ว่า หากเราพัฒนาให้ส่วนหนึ่งส่วนของระบบมีความสามารถมากขึ้นผลของประสิทธิภาพที่เพิ่มขึ้นจากการปรับปรุงส่วนหนึ่งส่วนใดดังกล่าวนั้น จะเป็นอัตราส่วนระหว่างประสิทธิภาพหลังการปรับต่อประสิทธิภาพก่อนการปรับปรุง

$$\text{Speedup} = [\text{ประสิทธิภาพของงานทั้งหมด เมื่อมีการพัฒนา}] / [\text{ประสิทธิภาพของงานทั้งหมด เมื่อไม่มีการพัฒนา}]$$

ทั้งนี้เวลาที่ใช้ในการประมวลผลโปรแกรมทดสอบ หลังจากที่ได้ทำการพัฒนานั้น สามารถหาได้จากความฟังก์ชันดังต่อไปนี้

$$[\text{Execution Time}_{\text{หลังการปรับปรุง}}] = [\text{Execution Time}_{\text{ก่อนปรับปรุง(ไม่มีผลกระทบ)}}] + ([\text{Execution Time}_{\text{ก่อนปรับปรุง(มีผลกระทบ)}}] / [\text{อัตราการพัฒนา}])$$

เช่น ในการซื้อของนั้น จะต้องเสียเวลาไปกับการเดินทางไปและกลับ 10 นาที และ เสียเวลาไปกับการซื้อของ และ ชำระเงินในร้านค้าอีก 3 นาที ดังนั้น หากผู้ซื้อสามารถเดินทางได้เร็วขึ้น 2 เท่า ค่า Speedup หรือ ประสิทธิภาพที่เพิ่มขึ้นจะน้อยกว่า 2 เท่า ซึ่งเราสามารถคำนวณได้ดังนี้

เวลาที่ใช้ในการซื้อของแต่เดิม = 13 นาที

เวลาที่ใช้ในการซื้อของเมื่อเดินทางเร็วขึ้น 2 เท่า = $3 + (10/2) = 8$ นาที

ดังนั้น Speedup หรือ ประสิทธิภาพที่เพิ่มขึ้น = $13/8 = 1.625$ เท่า

สรุป

พื้นฐานในการวิเคราะห์และวัดประสิทธิภาพของเครื่องคอมพิวเตอร์นั้น เป็นสิ่งสำคัญในการศึกษาระบบสถาปัตยกรรมคอมพิวเตอร์ ทั้งนี้เนื่องจาก เมื่อมีการออกแบบหรือปรับปรุงระบบคอมพิวเตอร์แล้ว จะต้องสามารถพิสูจน์ หรือ วิเคราะห์ได้ว่า การปรับปรุงดังกล่าวส่งผลต่อการพัฒนาระบบคอมพิวเตอร์ให้ดีขึ้นได้อย่างไร ทั้งนี้ในกรณีเป็นผู้ใช้หรือเลือกซื้อผลิตภัณฑ์ต่างๆ ทางคอมพิวเตอร์ ผู้ซื้อควรมีความรู้ความเข้าใจเมื่ออ่านคำแนะนำสินค้าต่างๆ ทราบถึงข้อดีข้อเสีย วิธี

การทดสอบ โดยไม่หลงคล้อยตามกับคำโฆษณาชวนเชื่อ เพื่อให้สามารถเลือกใช้งานระบบคอมพิวเตอร์ที่เหมาะสมกับงานของตน หน่วยประมวลผลกลางบางตัวนั้น อาจมีประสิทธิภาพดีเมื่อประมวลผลทางคณิตศาสตร์แบบจำนวนเต็ม แต่อาจมีประสิทธิภาพแย่มากเมื่อประมวลผลเลขทศนิยม หรือ Graphics

การวิเคราะห์ประสิทธิภาพในเชิงปฏิบัตินั้น นอกจากจะอาศัยความรู้พื้นฐานดังกล่าว ยังต้องอาศัยความรู้ทางสถิติเพื่อประกอบการวิเคราะห์ด้วย เช่น การหาค่าเฉลี่ยทางคณิตศาสตร์ การหาค่าเฉลี่ยฮาร์โมนิก หรือ การ Normalized ผลการทดสอบ เพื่อทำการเปรียบเทียบ

แบบฝึกหัดท้ายบท

1. เราจะวัดประสิทธิภาพของเครื่องคอมพิวเตอร์ได้อย่างไร
2. ท่านคิดว่า เหตุใดเราจึงเลือกใช้ CPU TIME ในการวัดประสิทธิภาพของระบบคอมพิวเตอร์
3. ปัจจัยใดบ้างส่งผลกระทบต่อประสิทธิภาพของเครื่องคอมพิวเตอร์
4. เราสามารถเพิ่มประสิทธิภาพของเครื่องคอมพิวเตอร์ได้อย่างไร
5. การเปลี่ยน CPU ใหม่ที่ความเร็วสูงขึ้นใน จัดเป็นการ ปรับปรุง Response Time หรือ Through Put
6. คอมพิวเตอร์ 2 ตัว ทำการแปลโปรแกรมเป็นภาษาเครื่อง ได้ผลเป็นชุดคำสั่งในกลุ่ม A B และ C ซึ่งคำสั่งในแต่ละกลุ่มมีค่า CPI เป็น 1,2,3 ตามลำดับ โปรแกรมที่ได้มีลักษณะ
 - โปรแกรมที่ได้จากคอมพิวเตอร์ชุดที่ 1 มี คำสั่งอยู่ในกลุ่ม A 5 ล้านคำสั่ง กลุ่ม B 1 ล้านคำสั่ง และ กลุ่ม C 1 ล้านคำสั่ง
 - โปรแกรมที่ได้จากคอมพิวเตอร์ชุดที่ 2 มี คำสั่งอยู่ในกลุ่ม A 6 ล้านคำสั่ง กลุ่ม B 2 ล้านคำสั่ง และ กลุ่ม C 1 ล้านคำสั่ง

โปรแกรมแต่ละชุดใช้เวลาเท่าใด โปรแกรมชุดใดมีประสิทธิภาพมากกว่ากัน หากเปรียบเทียบในหน่วย MIPS โปรแกรมชุดใดเร็วกว่ากัน

บทที่ 3 สถาปัตยกรรมชุดคำสั่ง

สถาปัตยกรรมชุดคำสั่ง คือสถาปัตยกรรมภาษาที่ใช้สำหรับระบบคอมพิวเตอร์ หรือภาษาเครื่อง ทั้งนี้ภาษาเครื่องนั้นเป็นภาษาคอมพิวเตอร์ที่มีลักษณะแตกต่างจากภาษาคอมพิวเตอร์ชั้นสูงโดยทั่วไปโดย มักไม่มีคำสั่งการควบคุม (เช่น While, for, loop) ในเอกสารชุดนี้มุ่งประเด็นสนใจที่สถาปัตยกรรมชุดคำสั่งของ MIPS Processor ซึ่งมีการใช้งานบนระบบ Nintendo, Silicon Graphics และ Sony Play Station ทั้งนี้เนื่องจากเป็นสถาปัตยกรรมที่มีชุดคำสั่งน้อยเข้าใจง่าย สิ่งที่ต้องกล่าวถึงและอธิบายในการสถาปัตยกรรมของระบบคอมพิวเตอร์ทั่วไปนั้น ประกอบด้วย

- คำสั่งต่างๆ (ซึ่งนิยมอธิบายด้วยภาษาแอสเซมบลี)
- Register หรือที่ปักข้อมูล และ Flag แสดงสถานะ
- ระบบการจัดการหน่วยความจำ
- รูปแบบคำสั่งภาษาเครื่อง

ทั้งนี้ระบบสถาปัตยกรรมชุดคำสั่งนั้น มีความจำเป็นอย่างยิ่งในการพัฒนาฮาร์ดแวร์ของระบบคอมพิวเตอร์ และ ซอฟต์แวร์ระบบ เนื่องจากเป็นข้อตกลงที่ใช้ระหว่างผู้พัฒนาฮาร์ดแวร์และซอฟต์แวร์ (ดังกล่าวแล้วในบทที่ 1) อย่างไรก็ตามสถาปัตยกรรมชุดคำสั่งมิได้ครอบคลุมถึงการเขียนภาษาแอสเซมบลีทั้งหมด เพราะในภาษาแอสเซมบลีจะมีการกำหนด Macro หรืออ้างอิงตำแหน่งในหน่วยความจำด้วย Label แทนที่การอ้างอิงด้วย Address ของหน่วยความจำ

หมวดคำสั่งและชุดคำสั่ง

ชุดคำสั่ง ในหน่วยประมวลผลต่างๆ นั้นโดยทั่วไปสามารถจำแนกตามลักษณะการทำงาน หรือการเรียกใช้งาน ได้เป็นหมวดหมู่ดังนี้

- หมวดคำสั่งที่เกี่ยวกับการเคลื่อนย้ายข้อมูล (Data Movement) ใช้สำหรับการอ่านและเขียนข้อมูลจากหน่วยความจำ
- หมวดคำสั่งการประมวลผลทางคณิตศาสตร์ (บวก ลบ คูณ หาร)
- หมวดคำสั่งการทำงานทางตรรกะ
- หมวดคำสั่งควบคุม เพื่อกำหนดเงื่อนไขของการทำงานคำสั่งถัดไป เช่น IF..THEN

นอกจากนี้ในโปรเซสเซอร์บางตัว อาจมีหมวดคำสั่งอื่นๆ หรือชุดคำสั่งที่ทำหน้าที่พิเศษอื่นๆ ทั้งนี้จะแตกต่างกันไปตามโครงสร้างและการออกแบบของผู้พัฒนา

ในการประมวลผลแต่ละคำสั่งนั้น หน่วยประมวลผลกลางจะมีขั้นตอนในการดึงคำสั่งจากหน่วยความจำเข้าสู่หน่วยประมวลผลกลาง จากนั้นจึงทำการประมวลผลต่อไป(รายละเอียดเพิ่มเติมดูในบทที่ 5 และ 6) ทั้งนี้คำสั่งที่จะถูกดึงเข้ามานั้นจะถูกกำหนดโดย Register พิเศษซึ่งเรียกว่า

Program Counter (PC) หรือ โพรเซสเซอร์บางตัวอาจจะเรียก Register พิเศษนี้ว่า Instruction Pointer (IP) ดังนั้นลักษณะพิเศษของคำสั่งควบคุมคือ สามารถปรับเปลี่ยน PC ให้ชี้ไปยังคำสั่งอื่น ซึ่งมีได้เป็นคำสั่งที่ติดกันได้เช่น

```
        beq    $s0,$s1,test
        add    $t0,$s1,$s2
test:   sub    $t0,$s1,$s2
```

คำสั่ง beq เป็นคำสั่งควบคุมซึ่งจะทำการเปรียบเทียบค่าภายใน Register \$s0 และ \$s1 ถ้าเท่ากันจะไปทำงานที่คำสั่ง sub และ หากไม่เท่ากันจะทำงานที่คำสั่ง add ซึ่งเป็นคำสั่งถัดไปติดกัน

การเขียนโปรแกรมด้วยคำสั่งต่างๆ ในภาษาแอสเซมบลีเพื่อสั่งให้เครื่องคอมพิวเตอร์ทำงานนั้นแต่ละคำสั่งจะประกอบด้วยรหัสคำสั่ง (Opcode) และ ตัวปฏิบัติการ (Operands) โดย Opcode จะบอกถึงคำสั่งที่ต้องการทำ และ ตัวปฏิบัติการบอกถึงหน่วยความจำหรือ Register ที่ใช้ในการทำงานคำสั่งนั้นๆ เช่น

```
add    $s0,$s1,$s2
and    $t0,$s0,$s1
```

add และ and นั้นเป็น Opcode ที่บอกให้ทราบว่าคำสั่งที่ต้องการทำงานคือ add และการ and โดย \$s0,\$s1,\$s2,\$t0 เป็น Operand ที่บอกให้ทราบว่า ให้นำค่าใน Register \$s1 และ \$s2 มาบวกกันแล้วเก็บไว้ที่ \$s0 เป็นต้น

ชุดคำสั่งบน MIPS นั้น การประมวลผลทางคณิตศาสตร์ต่างๆ จะมี Operand เป็น Register เท่านั้น โดย Operand ตัวแรกจะใช้อธิบาย Register ที่ใช้เก็บผลลัพธ์ และ Operand ถัดมาอีก 2 ตัวจะใช้อธิบาย Operand ที่เป็นตัวตั้ง เช่น

```
add    $s0,$s1,$s2
มีความหมายคือ
$s0 = $s1+$s2
```

ดังนั้นกรณีที่พัฒนาซอฟต์แวร์ด้วยภาษาชั้นสูงแล้วมีการประกาศหรือใช้ตัวแปรต่างๆ Compiler หรือตัวแปลภาษาจะทำการ Associate ตัวแปลต่างๆ กับ Register

เช่น

ภาษา C $A = B + C + D;$
 $E = F - A;$

MIPS CODE `add $t0, $s1, $s2`
 `add $s0, $t0, $s3`
 `sub $s5, $s5, $s0`

เป็นต้น

เนื่องจาก MIPS มีโปรเซสเซอร์ให้เราได้ใช้เพียง 32 ตัวเท่านั้น ดังนั้นหากในซอฟต์แวร์ที่เราพัฒนาจำเป็นต้องมีการอ้างอิงตัวแปรมากกว่า 32 ตัว ข้อมูลเหล่านั้นจะต้องถูกจัดเก็บไว้ในหน่วยความจำ และมีการอ่านค่าขึ้นมาเพื่อประมวลผล เมื่อเสร็จสิ้นแล้วก็บันทึกกลับไปยังหน่วยความจำเช่นเดิม และหากหน่วยความจำยังไม่พอเพียงที่จะเก็บข้อมูลดังกล่าว อาจจะต้องบันทึกข้อมูลเหล่านั้นลงใน I/O ต่างๆ เช่น Hard Disk เป็นต้น

การจัดการหน่วยความจำ (Memory Organization)

หน่วยความจำนั้นเป็นเหมือนกับ Array 1 มิติขนาดใหญ่ ซึ่งจะอ้างอิงด้วย Address สำหรับ MIPS จะใช้ Address ในการอ้างอิงหน่วยความจำ 32 bit ซึ่งหมายความว่า MIPS จะมีหน่วยความจำได้ทั้งสิ้น 2^{32} Byte หรือ 4 Gigabyte ในการอ้างอิงถึงหน่วยความจำนั้นอาจจะเป็นการอ้างอิงครั้งละ 1 Byte (Byte Addressing) หรืออ้างอิงครั้งละ 2 Byte (Half Word Addressing) หรืออาจจะอ้างอิงครั้งละ 4 Byte เป็น Word Addressing ก็ได้ ทั้งนี้ขึ้นอยู่กับรูปแบบคำสั่งแต่ละคำสั่ง ทั้งนี้ข้อควรระวังในการอ้างอิงแบบ Word และ Half Word Addressing คือระบบ Big Indian และ Little Indian ซึ่งจะแตกต่างกันไปตามโครงสร้างของ CPU แต่ละตัว ดังตัวอย่างแสดงการจัดเก็บข้อมูล 0A0B0C0Dh หรือ 168496141 ลงในหน่วยความจำ

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

ระบบ Big Indian

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

ระบบ Little Indian

ทั้งนี้ CPU MIPS ซึ่งกล่าวถึงในบทเรียนจะมีโครงสร้างแบบ Big Indian ในขณะที่ CPU Intel ตระกูล x86 ที่พบทั่วไปนั้น จะใช้ระบบ Little Indian ดังนั้นในการพัฒนาซอฟต์แวร์จึงควรใช้ความระมัดระวัง

ระบบการอ้างอิงหน่วยความจำ (Addressing Mode)

การอ้างอิงหน่วยความจำบน MIPS นั้นจะใช้ระบบ Base Addressing โดยการอ้างอิงตำแหน่งคู่กับ Register เช่น 20(\$s3) จะนำค่าใน Register \$s3 มาบวกกับ 20 ก่อนจึงจะใช้อ้างอิงตำแหน่ง Address เช่น

```
lw    $t0, 32($s2)
```

หากข้อมูลใน Register ที่ \$s2 เป็น 300 คำสั่งดังกล่าวจะนำข้อมูลในตำแหน่งที่ 332 มาเก็บไว้ใน Register \$t0

นอกจากนี้ยังมีการอ้างอิงแบบ Immediate เพื่อนำข้อมูลเหล่านั้นไปใช้ทันทีไม่จำเป็นต้องอ้างอิงข้อมูลจากหน่วยความจำ เช่น

```
add    $t2,$s1,4
```

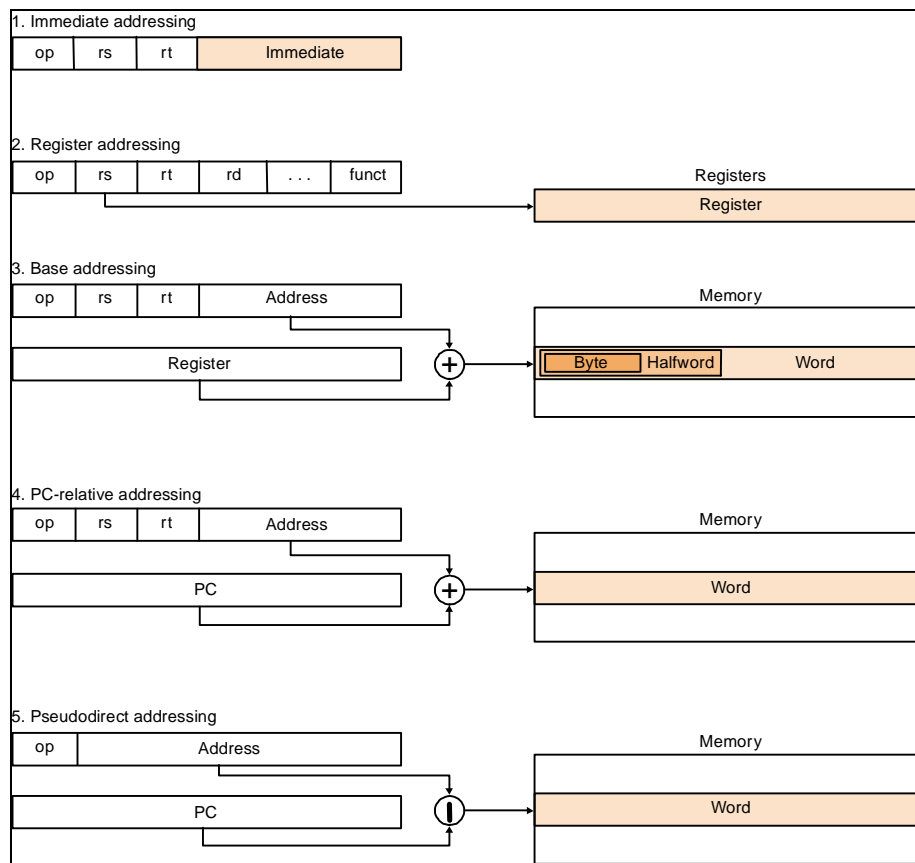
มีความหมายว่า

$$\$t2 = \$s1 + 4$$

บนระบบโพเรสเซอรอื่นๆ อาจมีการอ้างอิงตำแหน่งหน่วยความจำที่ซับซ้อนมากกว่านี้ เช่น ระบบ Base Index Addressing หรือระบบ Base Index Displacement ซึ่งสามารถอ้างอิงข้อมูลในหน่วยความจำได้หลากหลายรูปแบบ

รูปแบบคำสั่ง Instruction Format

คำสั่งภาษาแอสเซมบลีทุกคำสั่งจะถูกแปลงเป็นภาษาเครื่องโดยการแปลงเป็นภาษาเครื่องนี้ มักมีรูปแบบการแปลงที่เป็นแบบแผน ซึ่งเราจะเรียกรูปแบบที่แปลงได้ต่างๆ เหล่านี้ว่า Instruction Format เนื่องจากภาษาเครื่องนั้นจะประกอบด้วย Bit Pattern เพียง 0 และ 1 เท่านั้น รูปแบบคำสั่งที่แปลงจากภาษาแอสเซมบลีเป็นภาษาเครื่องจึงใช้การแบ่ง bit ออกเป็นช่วงๆ แล้วกำหนดความหมายให้ bit ต่างๆ กรณีของ MIPS นั้น คำสั่งทุกคำสั่งจะมีความยาว 32 bits เท่ากันหมด (โพเรสเซอรบางตัวจะมีคำสั่งที่มีความยาวตั้งแต่ 8-17 byte) มีรูปแบบคำสั่งทั้งสิ้น 3 รูปแบบเท่านั้น คือ แบบ R,I และ J โดยแต่ละรูปแบบจะมีการตีความต่างกันไปตามรูปแบบคำสั่ง



ตัวอย่างเช่นการแปลคำสั่ง add \$t0,\$s3,4

000001	01000	10011	00000000000000100
6 bit	5 bit	5 bit	16 bit
Opcode	Rs	Rt	Number

ทั้งนี้แต่ละคำสั่งแต่ละ Register จะถูกอ้างอิงด้วยรหัสใดขึ้นอยู่กับผู้ออกแบบแต่ละรายจะกำหนด (ดูรายละเอียดเพิ่มเติมได้จากคู่มือโครงสร้างชุดคำสั่งของแต่ละเครื่อง) เช่นคำสั่ง Add อาจแทนด้วย 000001 คำสั่ง sub อาจแทนด้วย 000010 เป็นต้น โดยหลักเกณฑ์ในการกำหนดรหัสต่างๆ นั้น ขึ้นอยู่กับแนวทางในการออกแบบวงจรเพื่อตีความคำสั่ง

สรุป

สถาปัตยกรรมชุดคำสั่งนั้น เป็นมาตรฐานที่ใช้สื่อระหว่างการพัฒนาฮาร์ดแวร์และซอฟต์แวร์ ความซับซ้อนของคำสั่งนั้น เป็นเพียงปัจจัยหนึ่งในการออกแบบเท่านั้น ทั้งนี้ หากคำสั่งมีความซับซ้อนมากขึ้น อาจส่งผลให้คำสั่งในการนั้นๆ ใช้ CPI มากขึ้น และ วงจรที่มีขนาดใหญ่ทำให้วงจรดังกล่าวไม่สามารถใช้ Clock Rate ที่มีความถี่สูง ทั้งนี้ปัจจุบันยังไม่มีมาตรฐานใดในการกำหนดว่าสถาปัตยกรรมชุดคำสั่งนั้นดีกว่าสถาปัตยกรรมชุดคำสั่งอื่นๆ หรือไม่

แบบฝึกหัดท้ายบท

1. สถาปัตยกรรมชุดคำสั่งคืออะไร มีสิ่งใดที่ต้องกล่าวถึงบ้าง และเป็นประโยชน์ในการพัฒนาซอฟต์แวร์ และ ฮาร์ดแวร์หรือไม่ อย่างไร
2. Opcode และ Operand คืออะไร
3. หากระบบคอมพิวเตอร์มีการอ้างอิงหน่วยความจำโดยใช้ Address ทั้งสิ้น 32 bit จะสามารถอ้างอิงข้อมูลแบบ Word Address ได้ทั้งสิ้นกี่ Word.
4. ระบบ Big Indian และ Little Indian มีข้อแตกต่างกันอย่างไร
5. การที่หน่วยประมวลผลมีระบบการอ้างอิงหน่วยความจำหลากหลายรูปแบบ เป็นประโยชน์หรือไม่ อย่างไร
6. การที่ Instruction Format ทุกรูปแบบมีความยาวคำสั่งเท่ากันหมด มีข้อดีข้อเสียอย่างไร
7. ภาษาแอสเซมบลีและสถาปัตยกรรมชุดคำสั่งมีข้อเหมือนหรือข้อแตกต่างกันหรือไม่ อย่างไร

บทที่ 4 หน่วยประมวลผลทางตรรกะและคณิตศาสตร์

การสร้างหน่วยประมวลผลทางตรรกะและคณิตศาสตร์เพื่อใช้ในหน่วยประมวลผลขนาด 8 บิต, 16 บิต หรือ 32 บิต แบบพื้นฐานนั้น อาศัยหลักการสร้าง ALU ขนาด 1 บิต แล้วนำมาต่อพวงกันจนได้ขนาดที่ต้องการ ทั้งนี้ความสามารถของ ALU จะต้องเพียงพอที่จะรองรับการทำงานของสถาปัตยกรรมชุดคำสั่งที่จะนำ ALU ดังกล่าวไปใช้ในระบบประมวลผล สิ่งที่ต้องคำนึงถึงในการสร้างวงจรทางคณิตศาสตร์คือ ระบบจำนวนที่จะใช้ในการประมวลผล และเงื่อนไขการคำนวณต่างๆ รวมถึงการสร้างวงจรตรรกะแบบเชิงผสม (Combinational Logic) ภายในบทนี้จะกล่าวถึงระบบจำนวน การแทนค่าเลขต่างๆ ภายใน CPU การแทนจำนวนเต็ม การแทนเลขทศนิยม การบวก ลบ และการคูณเลขเบื้องต้น

ระบบจำนวน และการบวกลบ

การแทนค่าต่างๆ ภายในหน่วยประมวลผลนั้น จะประกอบขึ้นจากบิต หรือ เลข 0 และ 1 เท่านั้น ซึ่งโดยทั่วไปเรามักคุ้นเคยกับระบบเลขฐาน 2 เช่น 10110_2 มีค่าเป็น 22_{10} หรือ -5_{10} มีค่าเท่ากับ -101_2 อย่างไรก็ตาม หากวิเคราะห์ให้ดีเราจะพบว่าการแทนค่าดังกล่าว ไม่สามารถใช้บอกเลขจำนวนเต็มลบภายในหน่วยประมวลผลได้ เพราะในหน่วยประมวลผลจะประกอบด้วย 0 และ 1 เท่านั้น ไม่มีเครื่องหมาย “-” ระบบจำนวนในที่นี้จึงมุ่งเน้นที่การแทนข้อมูลจำนวนเต็มบวกและลบภายในหน่วยประมวลผลกลาง

วิธีการที่ง่ายที่สุดในการแทนจำนวนเต็มบวกและลบภายในหน่วยประมวลผลกลางคือการกำหนดบิตเครื่องหมาย หรือระบบ Sign Magnitude เช่น 0100 แทนเลข 4 และ 1100 แทน -4 (ใช้บิตซ้ายมือสุดแทนเครื่องหมาย 0 แทนเครื่องหมายบวก และ 1 แทนเครื่องหมายลบ) แม้ว่าระบบ Sign Magnitude จะสามารถใช้แทนจำนวนบวกและลบได้ก็ตาม แต่ก็มีข้อเสียใหญ่ที่ไม่สามารถนำค่าที่มาจากเครื่องหมายต่างกันมาบวกลบกันได้ เช่น

Sign Magnitude	เลขฐาน 10
$1001 + 0001 = 1010$	$(-1) + (1) = -2$

จากปัญหาดังกล่าวจึงมีผู้คิดค้นระบบ One Complement ขึ้น เพื่อแก้ปัญหการบวกเลขที่มีเครื่องหมายต่างกัน โดยการแทนจำนวนลบด้วยการ Invert bit ทุกบิต เช่น 0011 มีค่าเป็น 3 จะใช้ 1100 แทนค่า -3 หากทดลองนำเลขที่มีเครื่องหมายต่างกันมาบวกกัน จะได้ผลลัพธ์ดังนี้

One Complement	เลขฐาน 10
$1011 + 0001 = 1100$	$(-3) + (1) = -4$

นอกจากนี้การบวกเลขในระบบ One Complement ยังมีเงื่อนไขเพิ่มเติมคือ กรณีที่การบวกมีตัวทดเกิดขึ้นให้นำตัวทดที่ได้มาบวกกลับในหลักหน่วย ซึ่งเราเรียกการบวกในระบบ One Complement ในลักษณะนี้ว่า One Complement Carry Add Around เช่น

One Complement	เลขฐาน 10
$1011 + 1110 = 11001$	$(-4) + (-1) = -5$
บวกตัวทด	
$1001 + 0001 = 1010$	

หากวิเคราะห์การบวกเลขในระบบ One Complement นั้น จะพบว่ามีความสะดวกสบายและเหมาะสมในการสร้างเป็นวงจรมากกว่าระบบ Sign Magnitude แต่การบวกซึ่งจะต้องสร้างวงจรบวกที่เป็น Carry Add Around นั้นก็ยังซับซ้อนและทำให้วงจรทำงานได้ช้า และเรายังพบอีกว่า 1111 และ 0000 จะแทนค่า -0 และ $+0$ ซึ่งมีค่าเท่ากัน อันเป็นความสับสนและซ้ำซ้อนของระบบจำนวน

ด้วยเงื่อนไขอันยุ่งยากของระบบ One Complement ดังกล่าวทำให้ผู้คิดดัดแปลงระบบจำนวนให้สามารถทำการบวกได้ง่ายยิ่งขึ้น และ ไม่มีการแทนค่าซ้ำซ้อน โดยการนำจำนวนลบในระบบ One Complement ทุกจำนวนมาบวกด้วย 1 ซึ่งผลลัพธ์ที่ได้คือระบบ Two Complement เช่น -2 ในระบบ One Complement จะแทนด้วย 1101 เมื่อใช้เป็นระบบ Two Complement จะแทนด้วย 1110 เป็นต้น นอกจากนี้การบวกจำนวนในระบบ Two Complement นั้น ยังทำได้สะดวกเหมือนการบวกตามปกติ เช่น

Two Complement	เลขฐาน 10
$1010 + 1111 = 11001$	$(-6) + (-1) = -7$
ลบตัวทด	
1001	

ในปัจจุบันหน่วยประมวลผลกลางที่พบจึงมักใช้การแทนจำนวนเต็มด้วย ระบบ Two Complement เพราะความสะดวกในการสร้างวงจรประมวลผล

Binary	Sign Magnitude	One Complement	Two Complement
000 (0)	011 (3)	011 (3)	011 (3)
001 (1)	010 (2)	010 (2)	010 (2)
010 (2)	001 (1)	001 (1)	001 (1)
011 (3)	000 (0)	000 (0)	000 (0)
100 (4)	100 (-0)	111 (-0)	111 (-1)
101 (5)	101 (-1)	110 (-1)	110 (-2)
110 (6)	110 (-2)	101 (-2)	101 (-3)
111 (7)	111 (-3)	100 (-3)	100 (-4)

ตารางเปรียบเทียบระบบจำนวนต่างๆ ขนาด 3 bit

* ข้อสังเกต ระบบที่มีการแสดงเครื่องหมายทุกระบบจะใช้ bit ซ้ายมือสุดในการแสดงเครื่องหมายเสมอ

* บนระบบ Two Complement นั้น กรณีการเพิ่มบิตของการแสดงผลข้อมูล คือการคัดลอกเครื่องหมายไปยังบิตถัดไป เช่น 011 เป็นเลข 3 บิต ที่แสดงค่า +3 แล้ว 0011 จะเป็นเลข 4 บิต ที่แสดงค่า +3 เช่นกัน หรือ 111 เป็นเลข 3 บิต ที่แสดงค่า -1 และ 1111 เป็นเลข 4 บิต ที่แสดงค่า -1 เช่นกัน ดังนั้นเราสามารถทำ sign extend ได้โดยข้อมูลยังคงค่าเดิม

Overflow การตรวจสอบและผลกระทบ

หากสังเกตระบบจำนวนต่างๆ ที่ใช้ในระบบคอมพิวเตอร์จะพบว่า ทุกระบบจำนวนจะแทนข้อมูลได้มากน้อยขึ้นอยู่กับปัจจัยหนึ่งคือ จำนวนบิตที่ใช้เก็บ เช่น ระบบ Two Complement ขนาด 3 bit จะแทนจำนวนได้ตั้งแต่ -4 ถึง +3 ซึ่งเราไม่สามารถแทนข้อมูล +5 ลงในระบบ Two Complement ขนาด 3 bit ได้ เราเรียกการแทนข้อมูลที่เกินความสามารถของระบบจำนวนที่ใช้ว่า Overflow

การเกิด Overflow นั้นเป็นไปได้ทุกระบบจำนวนเมื่อจำนวนหลักที่ใช้ไม่เพียงพอในการแทนค่าที่ได้ นอกจากนี้ Overflow อาจเกิดขึ้นได้จากการบวกลบจำนวน เช่น

Two Complement	เลขฐาน 10
0111 + 0010 = 1001	7 + 2 = (-7)

จากตัวอย่างดังกล่าวจะพบว่า 0111 (7) บวกกับ 0001(1) ควรจะได้คำตอบเป็น 9 แต่ 1001 นั้น บนระบบ Two Complement มีค่าเป็น -7 ซึ่งคำตอบที่ได้ผิดพลาดไป ลักษณะเช่นนี้ ก็เรียกว่า Overflow เช่นกัน

ผลกระทบจากการเกิด Overflow ในการบวกหรือลบจำนวนต่างๆ นั้น ทำให้การประมวลผลที่ได้ผิดพลาด ดังนั้นในการออกแบบหน่วยประมวลผลกลางจึงต้องมีการตรวจสอบ Overflow จากการประมวลผลคำสั่งต่างๆ และ เตือนให้โปรแกรมทราบ เพื่อป้องกันความผิดพลาดจากการประมวลผล ทั้งนี้การเกิด Overflow อาจส่งผลกระทบต่อคำสั่งประเภท Jump หรือ Branch ได้ เพราะการ Jump หรือ Branch แบบมีเงื่อนไขนั้นมักอาศัยการเปรียบเทียบจำนวน เช่น Branch If A Less Than B จะนำ A มาลบกับ B แล้วตรวจสอบเครื่องหมาย ซึ่งหากไม่วิเคราะห์ว่าการลบกันนั้นมี Overflow เกิดขึ้นหรือไม่ อาจทำให้โปรแกรมทำงานผิดพลาดได้

การตรวจสอบ Overflow สำหรับการบวกเลขฐานสองตามปกตินั้น สามารถทำได้โดยการดูตัวทด และ สำหรับการบวกระบบจำนวน Two Complement ทำได้โดยการตรวจสอบเครื่องหมายซึ่งเงื่อนไขการเกิด Overflow นั้น เป็นได้ดังนี้

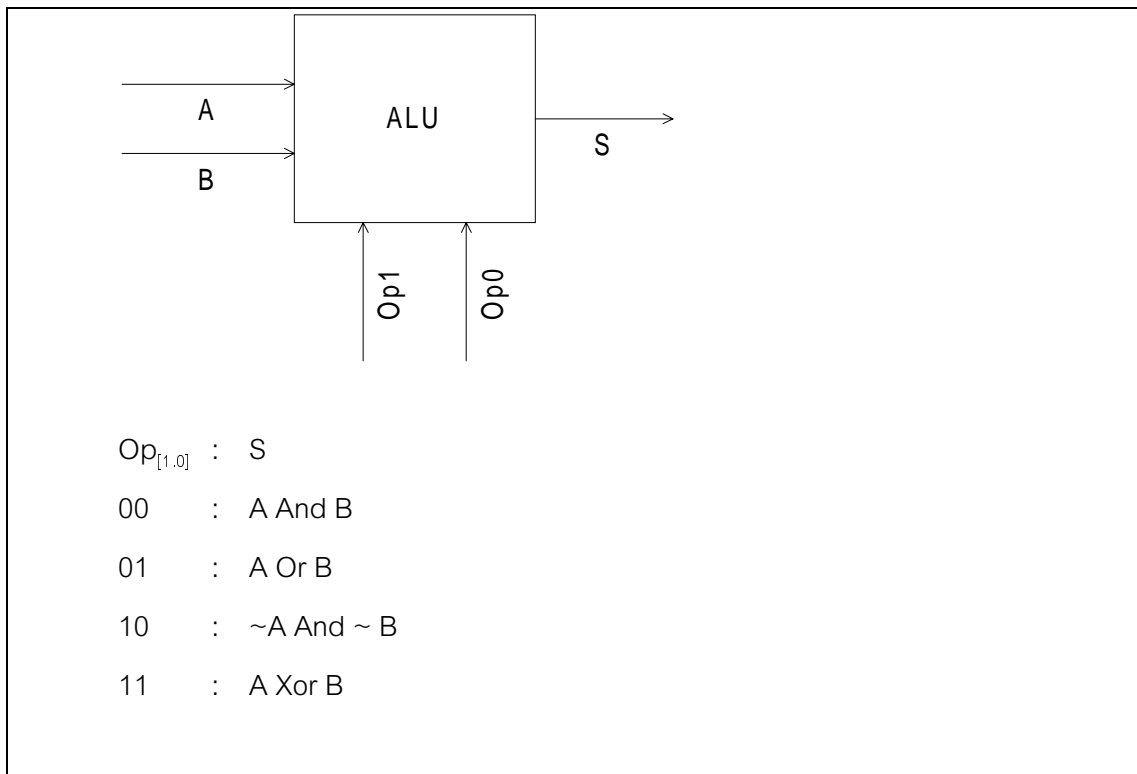
- Overflow จากการบวก จำนวนเต็มบวก ด้วย จำนวนเต็มบวก แล้วได้ผลลัพธ์เป็นจำนวนเต็มลบ
- Overflow จากการบวก จำนวนเต็มลบ ด้วย จำนวนเต็มลบ แล้วได้ผลลัพธ์เป็นจำนวนเต็มบวก
- Overflow จากการลบ จำนวนเต็มบวก ด้วย จำนวนเต็มลบ แล้วได้ผลลัพธ์เป็นจำนวนเต็มลบ
- Overflow จากการลบ จำนวนเต็มลบ ด้วย จำนวนเต็มบวก แล้วได้ผลลัพธ์เป็นจำนวนเต็มบวก

ดังนั้นในการออกแบบ ALU ที่หลักที่มีนัยยะสำคัญมากที่สุด (ซ้ายมือสุด) จึงต้องมีการตรวจสอบ Overflow (ทั้งนี้เนื่องจากเครื่องหมายแสดงที่บิตซ้ายมือสุด)

ALU (Arithmetic Logic Unit)

การสร้าง ALU ขนาด 1 บิตนั้น อาจทำได้โดยอาศัยพื้นฐานจากการสร้างวงจรตรรกะเชิงผสมทั่วไป (Combinational Logic) โดยการเขียนตารางแสดงค่าความเป็นจริง (Truth Table) และ ใช้ K-MAP ในการหาผลรวมของการคูณ (Sum of Product) ดังตัวอย่างต่อไปนี้

จงแสดงการออกแบบ ALU ขนาด 1 bit ซึ่งมีลักษณะดังนี้



แนวทางที่ 1 (Normal Design)

การออกแบบ

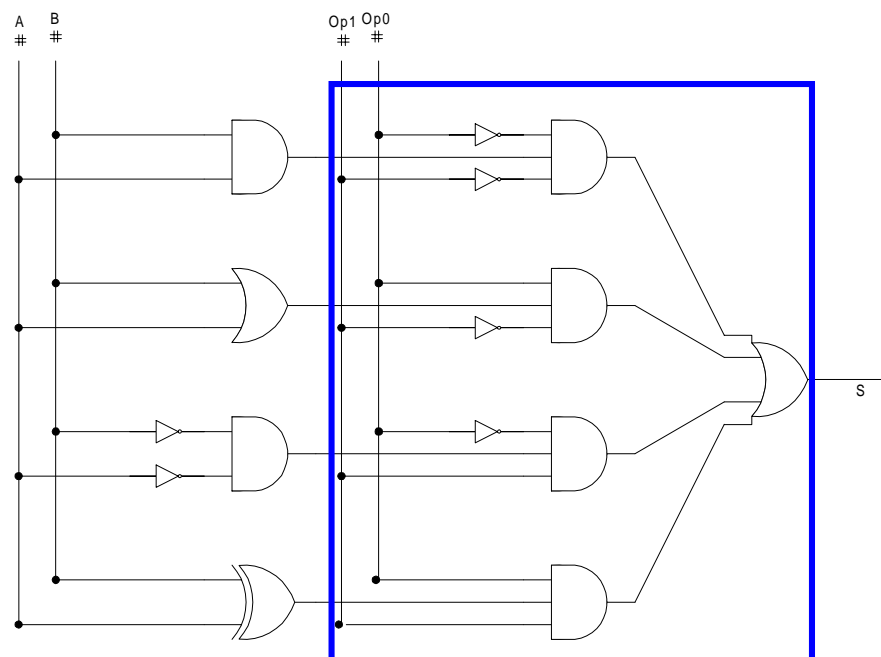
Op ₁ Op ₀ A B	S
0000	0
0001	0
0010	0
0011	1
0100	0
0101	1
0110	1
0111	1
1000	1
1001	0
1010	0
1011	0
1100	0
1101	1

1110	1
1111	0

ทำการวง K-MAP และหาผลลัพธ์					01	0	1	1	1
					11	0	1	0	1
Op1,Op0 / A B					10	1	0	0	0
00						0	0	1	0

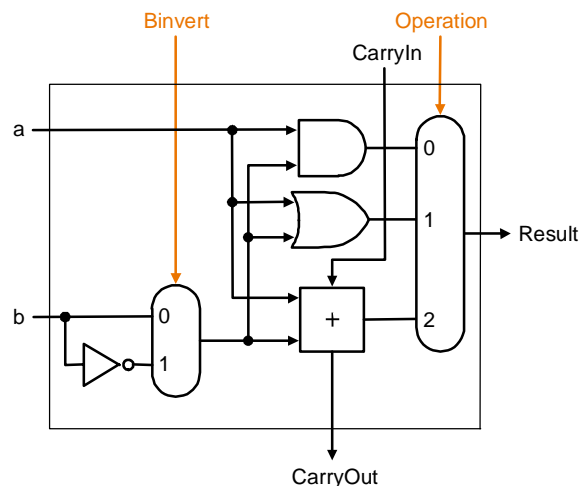
จากตัวอย่างดังกล่าวจะพบว่าวงจรที่ได้จะมีสมการที่ค่อนข้างซับซ้อน

แนวทางที่ 2 (Multiplexor)



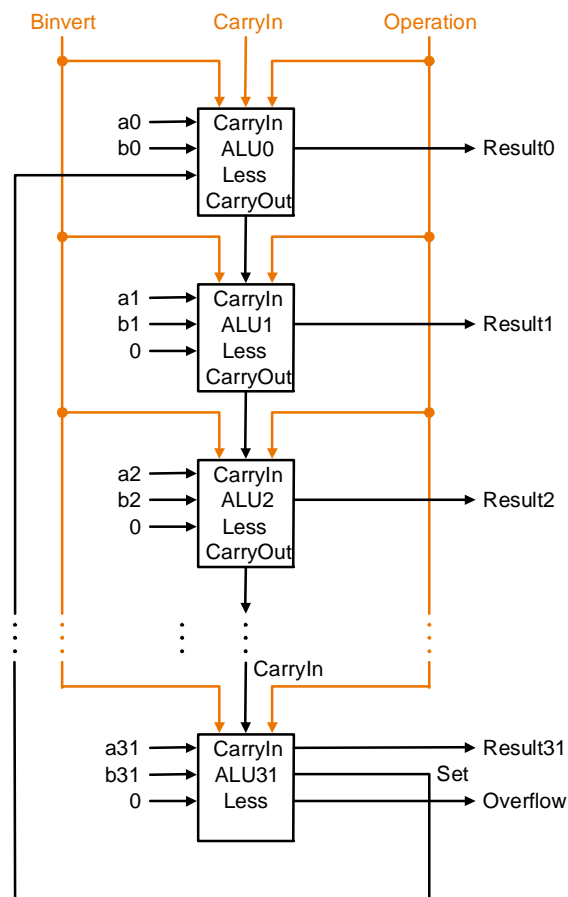
จากภาพจะพบว่าวงจรถัดไป สามารถทำงานได้เหมือนกับวงจรที่ได้จากฟังก์ชันในแนวทางการออกแบบที่ 1 แต่สามารถออกแบบได้ง่ายกว่า โดยการออกแบบวงจรที่สามารถทำงานได้ตามฟังก์ชันที่ระบบตามปกติก่อน เช่น A And B, A Or B, ~A And ~B และ A Xor B จากนั้นจึงใช้ Op[1.0] มาทำการเลือกผลลัพธ์ที่ต้องการโดยเมื่อ Op[1.0] เป็น 00 ผลลัพธ์ที่ S จะเป็น ผลลัพธ์จาก A And B ซึ่งเราจะเรียกววงจรที่ทำการเลือกสายสัญญาณผ่านนี้ว่า Line Selector หรือ Multiplexor นั่นเอง ด้วยหลักการนี้ เราสามารถสร้าง ALU ได้สะดวกมากขึ้น (หลักการนี้สามารถประยุกต์ได้กับการสร้างวงจรระกะทั่วไป)

อย่างไรก็ตาม ALU ที่ใช้งานตามความเป็นจริงนั้น มักจะมีวงจรบวกเลข หรือ Full Adder ซึ่งมีการส่งผ่านตัวทดถึงกัน (Carry In และ Carry Out) ทำให้การสร้าง ALU ที่สามารถบวกเลขได้นั้นมีสายสัญญาณมากขึ้น ดังภาพแสดง ALU 1 Bit ที่พบใน CPU MIPS

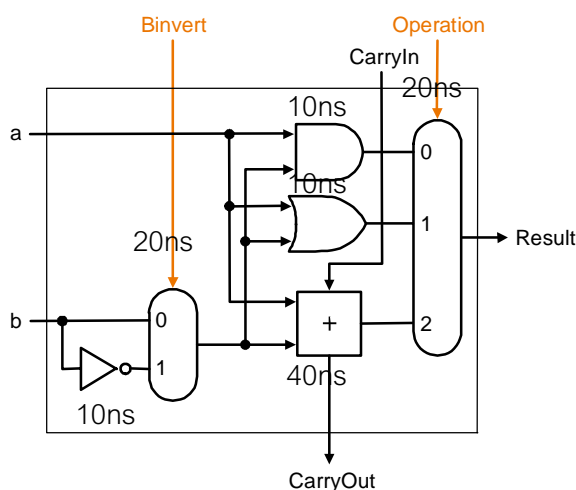


กรณีการลบนั้น เราสามารถใช้การบวกแทนการลบได้ แต่สร้างวงจรสำหรับการทำ Complement ขึ้นมาแทน เช่น $17-2$ มีความหมายเท่ากับ $17+(-2)$ เป็นต้น ทั้งนี้ในบางกรณี อาจมีการสร้างวงจรที่ทำหน้าที่ลบโดยตรงเพื่อใช้งานภายใน CPU ก็ได้ โดยไม่จำเป็นต้องทำการเปลี่ยนเครื่องหมายก่อน แต่จะไม่ขอกล่าวถึงในที่นี้

ในการสร้าง ALU ที่มีขนาดใหญ่มากขึ้นนั้น อาจทำได้โดยการนำ ALU ขนาด 1 Bit มาต่อพ่วงกัน ดังแสดงในภาพ โดยจะต้องทำการลากสายสัญญาณคำสั่งถึงกัน และ โยง Carry Out เข้ากับ Carry In ของ หลักถัดไป ในกรณีที่เป็นหลักสุดท้ายนั้น จะต้องมีการเพิ่มเติมวงจรสำหรับการตรวจสอบ Overflow เข้าไปด้วย



จากพื้นฐานความรู้ในเรื่องการออกแบบวงจรแบบตรรกะ ทำให้เราทราบว่า ประตูตรรกะ (Logic Gate) ต่างก็มีเวลาหน่วงในการประมวลผล (Gate Delay) ทั้งนี้เวลาหน่วงนั้นขึ้นอยู่กับจำนวนอินพุต หากกำหนดให้ Gate ทุกอันมี Delay เป็น 10ns, Multiplexor มี Gate Delay เป็น 20ns และ Full Adder มี Gate Delay เป็น 40ns เมื่อวิเคราะห์เทียบกับวงจรต่อไปนี้ เราจะพบว่า ALU ขนาด 1 บิตนี้จะต้องใช้เวลาในการประมวลผลเป็น 40ns กรณีการประมวลผลคำสั่ง And, Or และ จะใช้เวลา 80 ns ในการประมวลผลการบวกซึ่งมี not ด้วย แต่ในการเปรียบเทียบเวลานั้น เราจะต้องเลือกเวลาที่มากที่สุด (Critical Path) หรือ เลือกทางที่ใช้เวลามากที่สุดมาตั้งเป็นเวลาในการทำงานของวงจร ทั้งนี้เพื่อให้วงจรประมวลผลทุกฟังก์ชันได้ถูกต้อง ดังนั้นในที่นี้เราจึงกล่าวว่า ALU 1 บิตตัวอย่างใช้เวลาในการทำงาน 80ns



เมื่อมีการนำ ALU 1 บิต มาต่อพ่วงกันเพื่อให้เกิด ALU ขนาดใหญ่ ก็จะทำให้เวลาการทำงานของ ALU นั้นเพิ่มมากขึ้นไปด้วย โดยเราอาจประมาณได้ว่าหากนำ ALU ดังกล่าวมาต่อพ่วงกัน 32 บิต จะใช้เวลาในการทำงานประมาณ 80×32 หรือ 2560ns ซึ่งจะส่งผลต่อสัญญาณนาฬิกาที่ใช้กับหน่วยประมวลผลที่มี ALU ตัวนี้เป็นส่วนประกอบด้วย เช่น หาก ALU ใช้เวลาในการทำงานเป็น 2560ns หมายความว่า Cycle ของหน่วยประมวลผลนี้จะต้องมีค่าอย่างน้อยเป็น 2560ns หรือ มี Clock Rate อย่างมากเป็น $1/2560\text{ns} = 0.39\text{.. Mhz}$

ในทางปฏิบัติจึงมีการสร้างวงจรที่ซับซ้อนยิ่งขึ้นเพิ่มให้เวลาในการรอตัวทอดจากหลักก่อนหน้านั้นน้อยลง เช่นวงจร Carry Look Ahead Adder ซึ่งจะไม่ขอกล่าวถึงในที่นี้ ทั้งนี้หากผู้เรียนสนใจสามารถค้นคว้าเพิ่มเติมได้จากหนังสือ Computer Arithmetic ทั่วไป

การคูณ และ วงจรคูณ

การคูณเลขนั้นอาจทำได้โดยการวนลูปบวกหรือการสร้างวงจรเพื่อทำหน้าที่ในการคูณโดยเฉพาะ ทั้งนี้ในที่นี้จะขอกล่าวถึงเพียงการคูณเลขจำนวนเต็มบวกเท่านั้นหากต้องการศึกษาระบบการคูณที่ซับซ้อนและรวดเร็วให้ค้นคว้าจากหนังสือ Computer Arithmetic โดยการคูณในที่นี้จะอาศัยพื้นฐานจากการคูณเลขที่เรียนมาตั้งแต่ชั้นประถม เช่น

*011

010

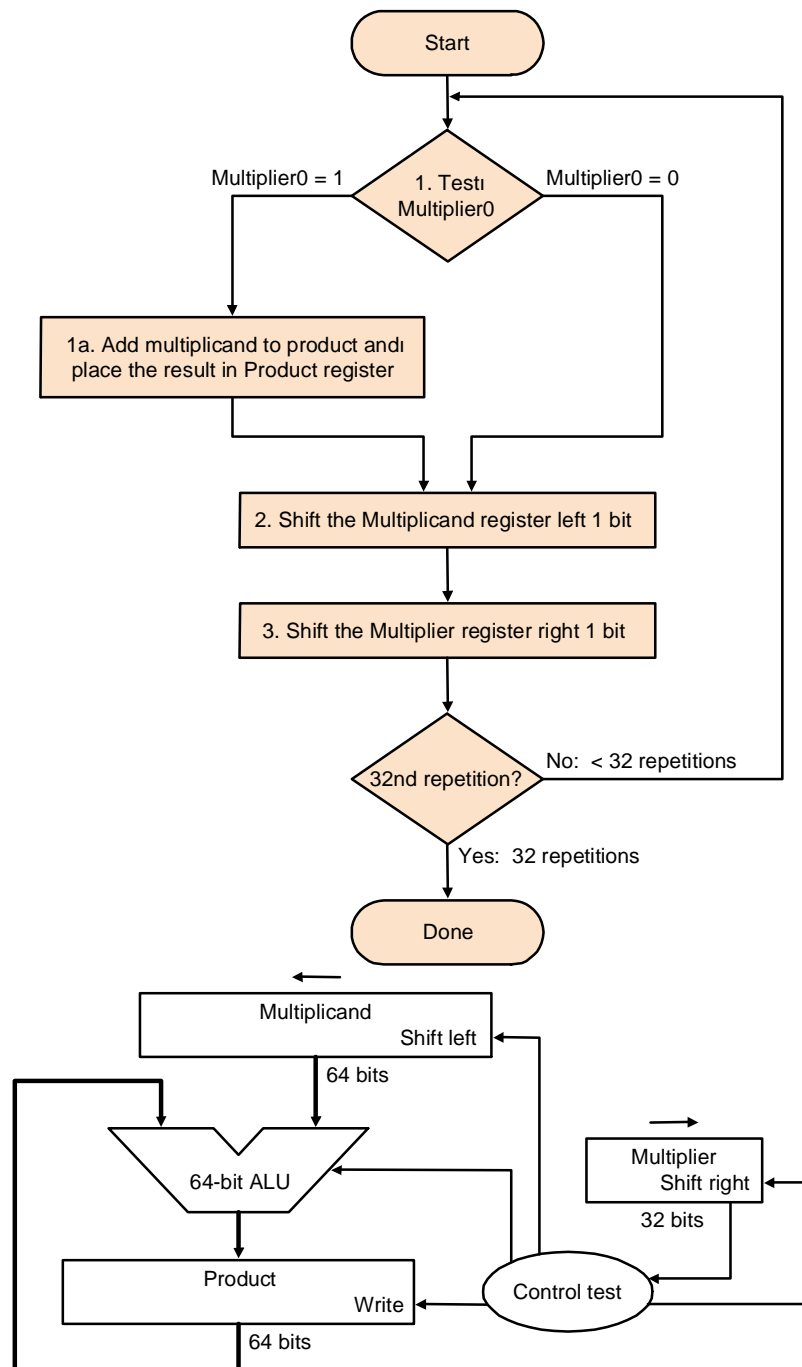
00000

00110

00110

ทั้งนี้ให้สังเกตว่าการคูณเลขฐานสองขนาด 3 บิตนั้นคำตอบที่ได้จะเป็นเลขฐานสองขนาด 6 บิต (ในทำนองเดียวกับการคูณเลขฐาน 10 เช่น 70×20 คำตอบที่ได้คือ 1400 จะเห็นว่าค่าหลักของคำตอบอาจเพิ่มขึ้นได้ถึง 2 เท่า) ดังนั้น เมื่อมีการสร้างวงจรเพื่อทำการคูณจึงต้องเตรียมที่สำหรับเก็บผลลัพธ์ไว้ 2 เท่าเสมอ ในหน่วยประมวลผลกลางทั่วไป นิยมใช้เรจิสเตอร์ 2 ตัวในการเก็บผลลัพธ์ที่ได้จากการคูณ

ตัวอย่างวงจรการคูณแบบที่ 1



กรณีต้องการนำ $1011 * 0101$ ผลลัพธ์ที่ได้จะเป็นดังนี้

ผลลัพธ์ (8bit)	ตัวตั้ง (8bit) <-	ตัวคูณ (4bit) ->
00000000	00001011	0101
00001011	00010110	0010
00001011	00101100	0001
00110111	01011000	0000
00110111	10110000	0000

เริ่มต้นนั้น ให้สังเกตที่ บิต 0 ของตัวคูณ หากเป็น 1ให้นำผลลัพธ์มาบวกกับตัวตั้ง และเก็บไว้ที่ผลลัพธ์ จากนั้นทำการ shift left ตัวตั้ง และ shift right ตัวคูณ กรณีที่ บิต 0 เป็น 0 ให้ทำการ shift ทันทึ ทำเช่นนี้ไปเรื่อยจนครบจำนวนบิต (กรณี CPU เป็นแบบ 4 บิต ก็ให้ทำ 4 ครั้ง)

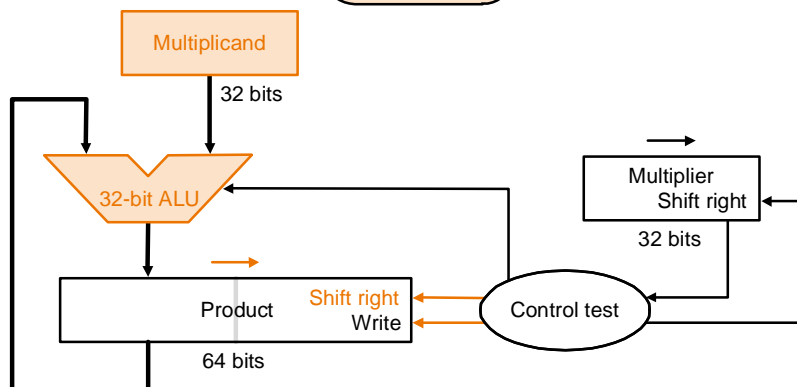
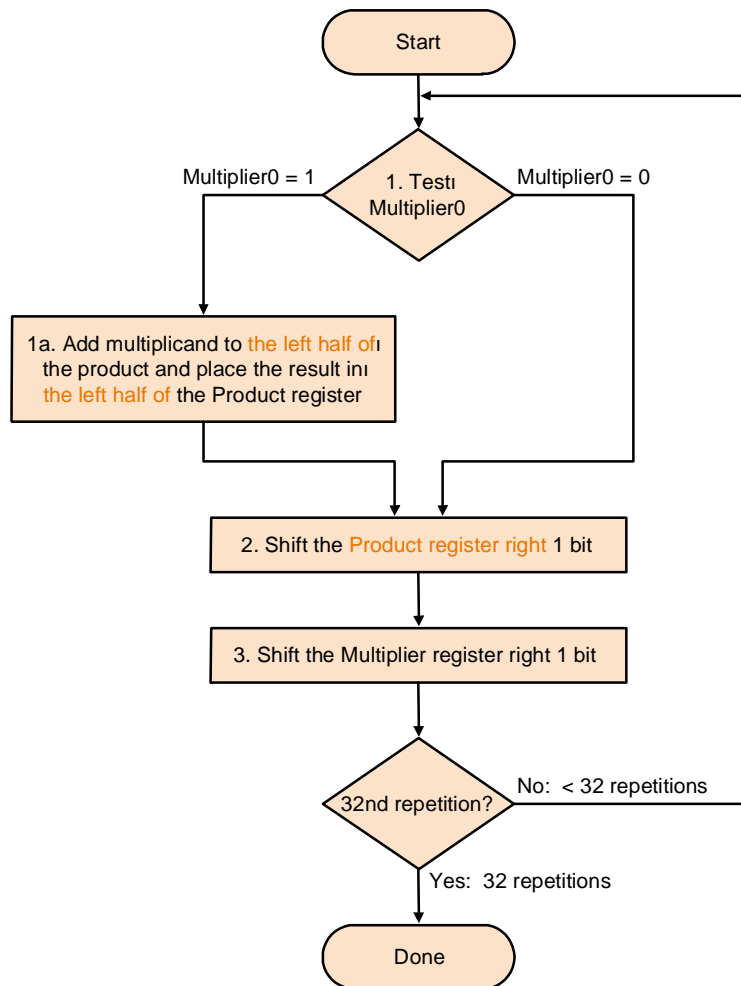
ทั้งนี้วงจรคูณจะต้องมีการทำงานเป็น State Machine โดยอาศัยวงจรควบคุม ต้องทำงานหลาย Cycle อย่างไรก็ตามข้อเสียของวงจรคูณในลักษณะนี้คือจะต้องใช้ เรจิสเตอร์ที่มีขนาดเป็น 2 เท่าของจำนวนบิตใน CPU ถึง 2 ตัว และ ต้องใช้ ALU ที่ประมวลผลได้จำนวนบิตเป็น 2 เท่าเช่นกัน จึงมีผู้ดัดแปลงวงจรคูณดังกล่าวให้กระชับรัดกุมยิ่งขึ้น

ตัวอย่างวงจรการคูณแบบที่ 2

กรณีต้องการนำ $1011 * 0101$ ผลลัพธ์ที่ได้จะเป็นดังนี้

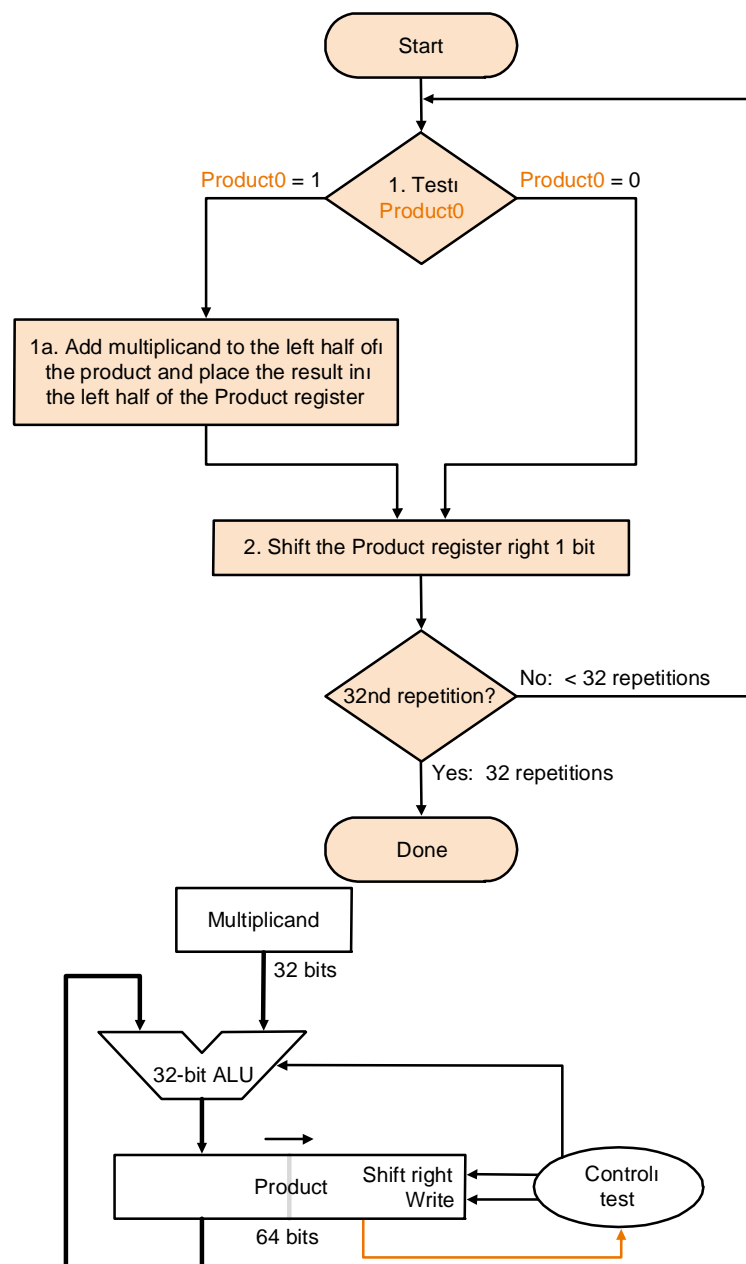
ผลลัพธ์ (8bit) ->	ตัวคูณ (4bit) ->
00000000	0101
10110000	
01011000	0010
00101100	0001
11011100	
01101110	0000
00110111	0000

เริ่มต้นให้ดูที่บิต 0 ของตัวคูณ หากเป็น 1ให้นำ ครึ่งบนของผลลัพธ์บวกกับตัวตั้ง จากนั้นทำการ Shift Right ตัวคูณและผลลัพธ์ บิต 0 ของตัวคูณเป็น 0 ให้ทำการ Shift ได้ทันที ทำเช่นนี้ไปเรื่อยจนกว่าจะครบจำนวนบิตทั้งหมด (เช่นกรณีตัวอย่างเป็น CPU 4 บิต)



จากโครงสร้างในลักษณะดังกล่าวจะพบว่าวงจรคูณในลักษณะนี้จะใช้ทรัพยากรน้อยกว่าแบบแรก และใช้ ALU ที่มีขนาดเท่ากับจำนวนบิตของ CPU อย่างไรก็ตามได้มีผู้คิดค้นดัดแปลงวงจรคูณลักษณะนี้ให้มีขนาดเล็กลงไปอีก ดังแสดงในวงจรคูณแบบที่ 3 ซึ่งเป็นวงจรคูณที่ใช้จำนวน Shift Register น้อยที่สุด

ตัวอย่างวงจรการคูณแบบที่ 2



วงจรคูณในลักษณะนี้จะนำตัวคูณมาเก็บไว้ที่ผลลัพธ์ช่วยให้ประหยัด Shift Register ได้อีก 1 ตัว มีขนาดเล็กและใช้ทรัพยากรน้อยลงไปอีก ทั้งนี้หลักการทำงานทุกอย่างยังคงเหมือนกับวงจรคูณแบบที่ 1 คือการสังเกตที่บิต 0 ว่าเป็น 0 หรือ 1 หากเป็น 1 ให้ทำการบวก บิตครั้งบนกับตัวตั้ง แล้วจึงทำการ Shift Right หากเป็น 0 ให้ทำการ Shift Right ได้ทันที

เช่น กรณีต้องการนำ $1011 * 0101$ ผลลัพธ์ที่ได้จะเป็นดังนี้

ผลลัพธ์(8bit) ->
00000101
10110101
01011010
00101101
11011101
01101110
00110111

อย่างไรก็ดี ได้มีการออกแบบและพัฒนางจรคุณที่มีความเร็วกว่านี้ขึ้นอีกเป็นจำนวนมาก ทั้งนี้วงจรคุณจะทำงานได้รวดเร็วมากน้อยเพียงใดขึ้นกับการออกแบบ และ Algorithm ในการพัฒนาของผู้คิดค้น

ระบบเลขทศนิยม

การแทนค่าเลขทศนิยมนั้น อาจทำได้เหมือนกับการบวกเลขจำนวนเต็มปกติแล้วนับตำแหน่งจุดทศนิยม เช่น $1.25 + 30.50$ อาจทำได้โดยการบวก $125+3050$ ซึ่งได้ผลลัพธ์เป็น 3175 จากนั้นจึงทำการเติมจุดทศนิยม 2 ตำแหน่งหลังตอนแสดงผลเป็น 31.75 (ระบบดังกล่าวนิยมใช้ในการคำนวณจำนวนเงินบนเครื่อง Main Frame หรือระบบคอมพิวเตอร์ยุคแรก ซึ่งเราเรียกการแทนค่าเลขทศนิยมในลักษณะนี้ว่า Pack Decimal)

ทั้งนี้การแทนค่าทศนิยมดังกล่าวไม่เหมาะกับการคำนวณทางคณิตศาสตร์หรือวิทยาศาสตร์ที่ต้องการให้มีการคำนวณที่มีนัยสำคัญสูง ดังนั้นจึงมีการนำการแทนเลขทศนิยมในลักษณะ Scientific Method มาใช้กับการแทนค่าทศนิยมในระบบคอมพิวเตอร์ โดยการเก็บค่าและ เลขชี้กำลังแยกจากกัน เช่น 13590.12345 เมื่ออ้างอิงเป็นระบบ Scientific จะเก็บเป็น $1.359012345 * 10^4$ เป็นต้น

แนวคิดดังกล่าวถูกนำมาใช้ในการแทนค่าเลขทศนิยมบนระบบคอมพิวเตอร์ซึ่งมาตรฐานที่นิยมใช้คือ IEEE754 ซึ่งกำหนดโดย สมาคม IEEE มาตรฐานดังกล่าวนี้ แบ่งการแทนเลขทศนิยม

บนระบบคอมพิวเตอร์เป็น 2 รูปแบบคือ Single Precision และ Double Precision ซึ่งจะมีความละเอียดในการเก็บข้อมูลต่างกัน

- single precision: 8 bit exponent, 23 bit significant
- double precision: 11 bit exponent, 52 bit significant

โดยมีหลักในการคิดคือ $(-1)^{\text{sign}} * (1 + 0.\text{Significant}) * 2^{\text{exponent-bias}}$ โดยกำหนดให้บิตซ้ายมือสุดเป็น บิตเครื่องหมาย bias มีค่าเป็น 127 และ 1023 สำหรับ single และ double precision ตามลำดับ

เช่น ต้องการเก็บเลข -0.75

เลขฐาน 2 จะมีค่าเป็น -0.11 หรือ $-1.1 * 2^{-1}$

ค่า exponent เทียบกับ Bias คือ 126 หรือ 01111110

เขียนเป็น IEEE754 แบบ Single precision ได้ว่า

10111111010000000000000000000000

$(-1)^1 * (1 + 0.1) * 2^{126-127}$

ในหน่วยประมวลผลกลางหลายตัว มักมีวงจรพิเศษที่มีความสามารถในการประมวลผลเลขทศนิยมตามมาตรฐาน IEEE754 หรือ Floating Point Unit เช่น Match Coprocessor เป็นต้น

สรุป

ALU นั้นทำหน้าที่ในการประมวลผลทางคณิตศาสตร์และตรรกะ ซึ่งฟังก์ชันของ ALU นั้นจะต้องรองรับสถาปัตยกรรมชุดคำสั่งของระบบคอมพิวเตอร์นั้นๆ การออกแบบ ALU จะต้องอาศัยความเข้าใจในระบบจำนวนที่ใช้ การแทนค่าของระบบ และพื้นฐานการออกแบบวงจรตรรกะทั่วไป ทั้งนี้ผู้ออกแบบต้องคำนึงถึงประสิทธิภาพของ ALU เป็นหลัก เพราะหาก ALU ทำงานช้าจะทำให้หน่วยประมวลผลกลางทำงานช้า

แบบฝึกหัดท้ายบท

1. จงหาค่าของจำนวนต่อไปนี้ในระบบ Binary, Sign Magnitude, One Compliment และ Two Compliment

- 10001001
- 01010011
- 10101100
- 11111111

2. จงแสดงการคำนวณต่อไปนี้ ในระบบการแทนจำนวนเต็มแบบ One Compliment

- $11001001 + 01011001$
- $01111111 + 00000001$

3. จงแสดงการคำนวณต่อไปนี้ ในระบบการแทนจำนวนเต็มแบบ Two Compliment

- $11001100 + 01111000$
- $01111111 + 01000000$

4. จงอธิบายเงื่อนไขในการเกิด Overflow จากการบวกเลข 2 จำนวน เพื่อเป็นแนวทางในการสร้างวงจร

5. ระบบ Sign Magnitude, One Compliment และ Two Compliment มีข้อดีข้อเสียอย่างไร จงอธิบาย

6. จงแสดงการออกแบบ ALU ขนาด 1 บิต โดยอาศัย Multiplexor ซึ่งมีลักษณะดังต่อไปนี้

Op _[2..0]	S
000	0
001	Not A
010	A Or B
011	A And B
100	A Xor B
101	A
110	A +B
111	1

(ให้ใช้วงจร Full Adder ในการบวก)

7. หาก Multiplexor มี Gate Delay เป็น 10ns และ Gate อื่นมี Gate Delay เป็น 5ns เมื่อนำผลลัพธ์ที่ได้จากการออกแบบ ALU ในข้อ 6 มาต่อเป็น ALU ขนาด 32 bit แล้ว ALU ดังกล่าวจะทำงานได้กับหน่วยประมวลผลกลางที่มี Clock Rate อย่างมากเท่าใด
8. จงแสดงการออกแบบวงจรสำหรับตรวจสอบเงื่อนไขการเกิด Overflow
9. หาก ALU ของท่านมี 16 bit จะสามารถแทนค่าจำนวนเต็มในช่วงใดได้บ้างในระบบ Two Compliment
10. จงแสดงการเก็บ -2.5 ลงในระบบ IEEE754 แบบ Single Precision

บทที่ 5 หน่วยประมวลผลกลาง (ทางเดินข้อมูลและสัญญาณควบคุม)

ตามที่ได้ศึกษาถึงสถาปัตยกรรมชุดคำสั่ง และการออกแบบ ALU อย่างง่ายแล้ว เนื้อหาภายในบทนี้ จะเป็นการนำองค์ประกอบต่างๆ มาประกอบเข้าด้วยกันเพื่อสร้างหน่วยประมวลผลกลาง โดยจะมุ่งประเด็นกล่าวถึงองค์ประกอบต่างๆ ในหน่วยประมวลผลกลาง การออกแบบทางเดินข้อมูล (Data path) และการสร้างสัญญาณเพื่อควบคุมองค์ประกอบต่างๆ ให้ทำงานควบคุมสอดคล้องกัน

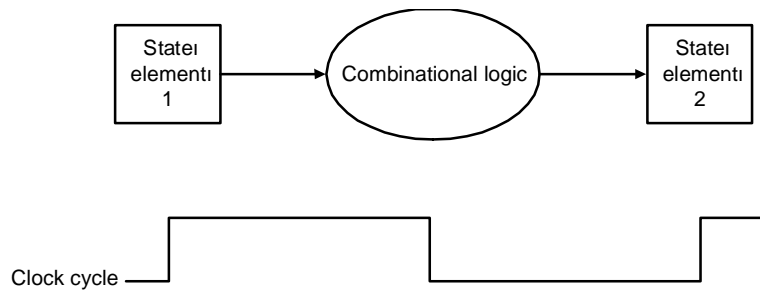
ภายในหน่วยประมวลผลกลางนั้นประกอบขึ้นจากวงจรไฟฟ้าต่างๆ ซึ่งหากเราวิเคราะห์ลักษณะของวงจรที่เป็นส่วนประกอบภายในหน่วยประมวลผลกลางตามลักษณะของหน้าที่ เราจะสามารถจำแนกวงจรได้เป็น 2 กลุ่มคือ

- วงจรสำหรับการประมวลผลข้อมูล หรือ Combinational Logic (วงจรเชิงผสม) ซึ่งเป็นวงจรที่ทำหน้าที่ถอดรหัส ตีความ หรือประมวลผลต่างๆ ซึ่งสร้างขึ้นได้ด้วย K-MAP (เช่น ALU ดังแสดงในบทก่อนหน้า)
- วงจรสำหรับการประมวลผลแบบลำดับ หรือ Sequential Logic (วงจรเชิงลำดับ) ซึ่งเป็นวงจรที่ทำหน้าที่กำหนดการทำงานของส่วนต่างๆ ของระบบให้ทำงานตามขั้นตอนที่เหมาะสม

การกำหนดจังหวะในการทำงานของวงจรเชิงลำดับ

จากพื้นฐานความรู้ในการออกแบบวงจรตรรกะ เราจะพบว่ามีองค์ประกอบที่มีความสามารถในการจดจำค่า อยู่หลายรูปแบบ ซึ่งมีทั้งแบบใช้สัญญาณนาฬิกาเป็นตัวกำหนด (Synchronous) หรือ ไม่มีสัญญาณนาฬิกา (Asynchronous) ซึ่งการออกแบบในลักษณะที่มีสัญญาณนาฬิกาเป็นตัวให้จังหวะนั้น ในปัจจุบันเป็นที่นิยม เนื่องจากสามารถออกแบบได้ง่ายกว่า นอกจากนี้ในระบบคอมพิวเตอร์ นิยมใช้ D-Flipflop มากกว่าการใช้ Latch เนื่องจาก Flipflop จะทำงานที่ขอบของสัญญาณนาฬิกาขึ้นหรือขาลง ในขณะที่ Latch จะทำงานที่ Logic '1' หรือ '0' (รายละเอียดเพิ่มเติมสามารถศึกษาได้จากพื้นฐานวงจรดิจิทัลเชิงลำดับ หรือ Slide ประกอบการบรรยาย)

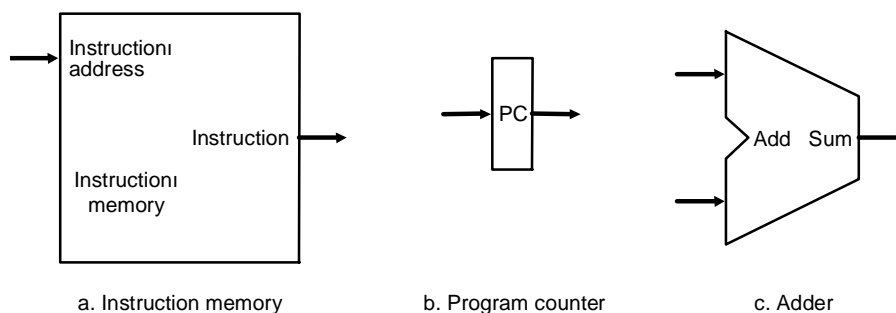
ลักษณะการทำงานโดยทั่วไปของวงจรเชิงลำดับขั้นนั้น จะอาศัยส่วนหนึ่งส่วนใดของสัญญาณนาฬิกาในการให้จังหวะ ซึ่งผู้ออกแบบโดยทั่วไปนิยมใช้ขอบสัญญาณขาขึ้นในการให้จังหวะเปลี่ยนสถานะ อย่างไรก็ตามข้อควรระวังคือ ในการออกแบบนั้น ความยาวของ Clock หรือ คาบ (Period) จะต้องมากพอที่ใช้ให้วงจรเชิงผสมที่อยู่ระหว่างกลางนั้นทำงานเสร็จได้ หากเวลาน้อยไป อาจทำให้การประมวลผลที่ได้ผิดพลาด (ดังแสดงในการหาเวลาการทำงานของ ALU ในบทที่ 4)

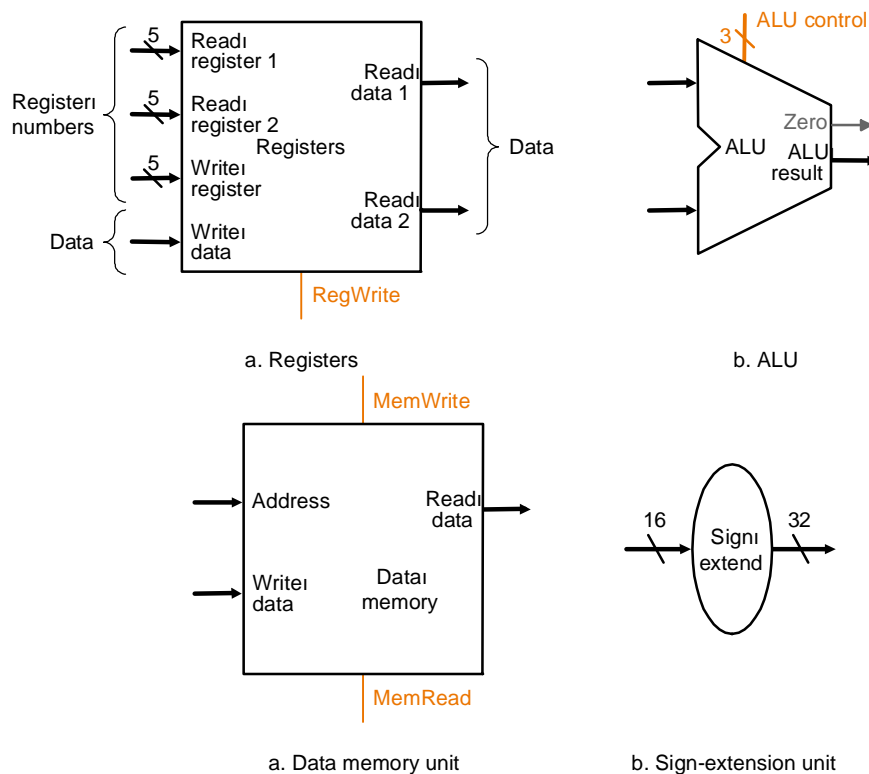


องค์ประกอบต่างๆ ภายในหน่วยประมวลผลกลาง

จากที่กล่าวถึงแล้วในบทที่ 1 นั้น ภายในหน่วยประมวลผลกลาง ประกอบด้วย ALU หน่วยควบคุม หน่วยความจำชั่วคราว (Register) และ ระบบสายสัญญาณต่างๆ เป็นพื้นฐาน นอกจากนี้บนสถาปัตยกรรมสมัยใหม่ยังพบว่ามี Pipeline, Cache หรือ Vector Unit อีก ซึ่งจะกล่าวถึงในบทถัดไป อย่างไรก็ตาม ALU และระบบสายสัญญาณต่างๆ นั้น จะเป็นวงจรเชิงผสม ในขณะที่ Register และ หน่วยควบคุมจะมีองค์ประกอบสำหรับการจำสถานะ (วงจรเชิงลำดับ) เข้ามาเกี่ยวข้องด้วย โดยในที่นี้จะกล่าวถึงโครงสร้างของส่วนต่างๆ ยกเว้น ALU ซึ่งได้กล่าวถึงไปแล้วในบทที่ 4

เมื่อวิเคราะห์ถึงองค์ประกอบต่างๆ ที่จำเป็นสำหรับการออกแบบเพื่อรองรับสถาปัตยกรรมชุดคำสั่งของ MIPS นั้นทั้งหมดมาแสดงพร้อมกัน เราสามารถแสดงส่วนต่างๆ ได้ดังภาพข้างล่าง โดย Instruction memory จะเป็นหน่วยความจำที่ใช้เก็บโปรแกรม PC คือรีจิสเตอร์ที่ใช้บอกตำแหน่งของโปรแกรมที่ต้องการอ่าน Adder ใช้สำหรับบวกค่า PC รีจิสเตอร์เพื่อใช้จำข้อมูล ALU ใช้สำหรับการประมวลผลทางคณิตศาสตร์และตรรกะ Data memory เป็นหน่วยความจำข้อมูล และ Sign extend เพื่อใช้ขยายเครื่องหมายกรณีที่ Operand เป็น Immediate ขนาดเล็กกว่าขนาดของรีจิสเตอร์ (หากผู้อ่านไม่สามารถอธิบายได้ว่าเหตุใดจึงต้องมีองค์ประกอบต่างๆ เหล่านี้ ให้ลองวิเคราะห์เปรียบเทียบกับสถาปัตยกรรมชุดคำสั่งของ MIPS ในบทที่ 3 อีกครั้งหนึ่ง)

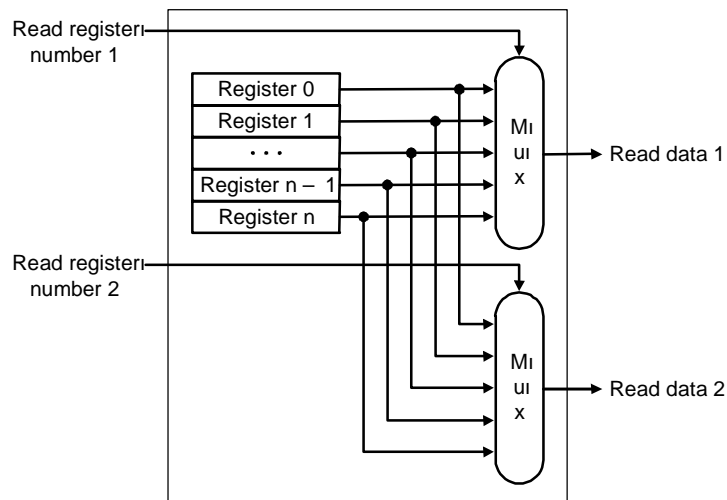




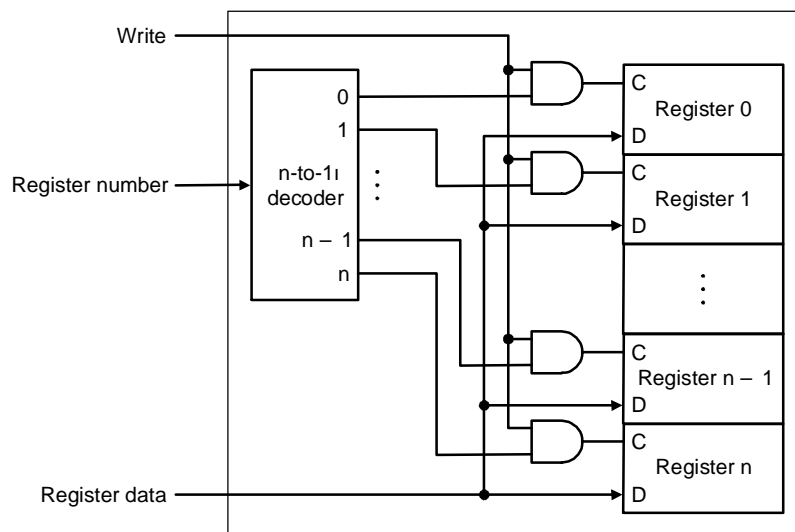
รีจิสเตอร์ (Register)

รีจิสเตอร์นั้น จะทำหน้าที่ในการพักข้อมูลชั่วคราว ซึ่งโดยปกติในเครื่องหนึ่งๆ จะมี รีจิสเตอร์ให้เราใช้งานได้มากกว่าหนึ่งตัว โดยการอ้างถึงรีจิสเตอร์ต่างๆ นั้น จะถูกกำหนด โดยรหัสของภาษาเครื่อง (ดังแสดงแล้วในบทที่ 3) ดังนั้น การออกแบบรีจิสเตอร์จึงอาศัย การออกแบบเป็น Array ของ D-Filpflop หรือ Latch หลายๆ ตัว

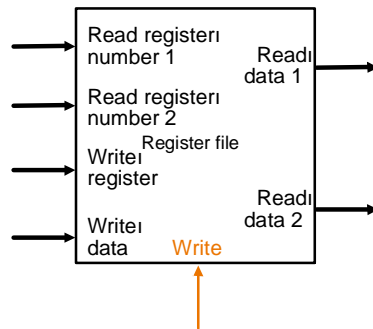
ทั้งนี้ตามโครงสร้างสถาปัตยกรรมชุดคำสั่งของ MIPS นั้น แต่ละคำสั่งสามารถใช้ รีจิสเตอร์ได้ 2 ตัวพร้อมกัน ในการออกแบบรีจิสเตอร์กรณีการอ่านเพื่อใช้งาน จึงสามารถ ทำได้โดยการใช้ Multiplexor 2 ตัว ในการเลือกค่าเพื่อป้อนเป็นอินพุตให้กับ ALU ต่อไป (ดังแสดงในภาพ)



สำหรับการเขียนค่า นั้น จะอาศัยสัญญาณนาฬิกา หรือสัญญาณ Write เป็นตัวกำหนดการเขียนค่าให้กับรีจิสเตอร์ ร่วมกับ Decoder เพื่อเลือกค่ารีจิสเตอร์ ที่ต้องการเขียน ซึ่งลักษณะการทำงานสามารถแสดงได้ดังรูปดังกล่าว เช่นเมื่อต้องการเขียนค่ารีจิสเตอร์ที่ 3 ด้วยข้อมูล 2521 สามารถทำได้โดยการป้อนค่า 3 ที่ Register number และ ค่า 2521 ที่ Register data จากนั้นจึงให้จังหวะสัญญาณ Write (ซึ่งสร้างจากวงจรควบคุม)

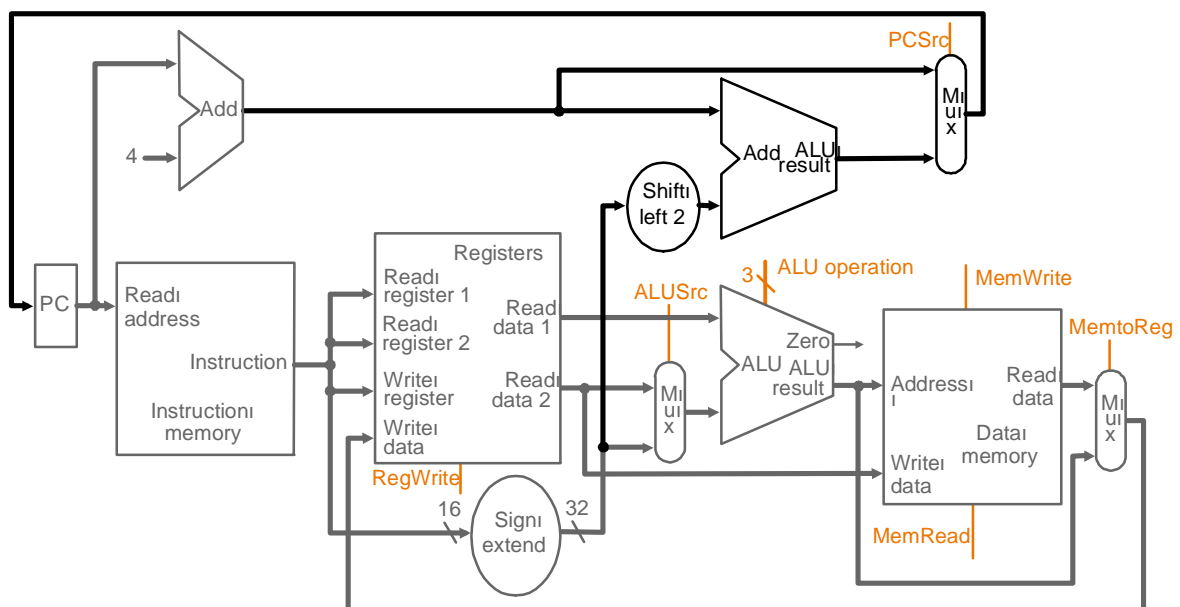


เมื่อนำโครงสร้างของการอ่านและเขียนข้อมูลดังกล่าวมาประกอบรวมกัน ก็จะได้ Register File ที่พร้อมสำหรับการทำงานในหน่วยประมวลผลกลาง นอกจากนี้เราจะพบว่า การอ่านและการเขียนข้อมูลจากรีจิสเตอร์นั้น สามารถกระทำได้พร้อมๆ กัน โดยไม่ทำให้เกิดการประมวลผลข้อมูลผิดพลาด



ทางเดินข้อมูล (Datapath)

ทางเดินข้อมูล(Datapath) หรือ bus ภายในหน่วยประมวลผลกลางนั้น เป็นสายสัญญาณ หรือ ทางที่ใช้ส่งผ่านข้อมูลระหว่างองค์ประกอบต่างๆ ภายในหน่วยประมวลผลกลาง โดยในการออกแบบนั้น สามารถทำได้โดยการนำองค์ประกอบต่างๆ มาต่อพ่วงกันในรูปแบบที่เหมาะสม และใช้ Multiplexor เป็นตัวเลือกทางเดินของข้อมูล กรณีที่ข้อมูลดังกล่าวมีทางเลือกที่เป็นไปได้มากกว่าหนึ่งทาง เช่น ค่า PC ซึ่งแสดงถึงคำสั่งต่อไปที่จะถูกดึงขึ้นมาทำงาน อาจจะอยู่ติดกับคำสั่งปัจจุบันคือ PC+4 หรือ อาจจะเป็นค่าอื่นๆ ที่เกิดจากการ Branch จึงใช้ Multiplexor สำหรับการเลือก



อย่างไรก็ตาม การที่จะเลือกว่าควรมี Datapath เชื่อมต่อระหว่างจุดใดจุดหนึ่งถึงกันนั้น นั้น จะพิจารณาโดยอาศัยสถาปัตยกรรมชุดคำสั่งเป็นหลัก โดยพิจารณาจาก Addressing Mode และ ชุดคำสั่งว่าแต่ละคำสั่งนั้นจะต้องใช้องค์ประกอบใดบ้าง และ ข้อมูลที่ต้องการใช้ในการประมวลคำสั่งนั้นๆ ได้มาจากแหล่งใดบ้าง เช่น คำสั่งต่างๆ จะประกอบด้วย Operand ซึ่งมาจาก

Register หรือ มาจากคำสั่งโดยตรง ดังนั้น จึงต้องมีทางเดินข้อมูลเพื่อนำข้อมูลจากรีจิสเตอร์ไปป้อนเป็นผลลัพธ์ให้กับ ALU และ ทางเดินข้อมูลที่จะนำข้อมูลในกรณีที่ Operand อยู่ในคำสั่งไปป้อนให้กับ ALU ด้วยเช่นกัน

สัญญาณควบคุม (Control)

หากสังเกตที่ทางเดินข้อมูลที่มีการกำหนดไว้ตามแผนภาพดังกล่าวข้างต้นแล้ว เราจะพบว่า มี Multiplexor ซึ่งใช้ในการเลือกสัญญาณต่างๆ อยู่มากมายหลายแห่ง โดยสัญญาณที่จะใช้ป้อนให้กับ Multiplexor เพื่อเลือกทางเดินของข้อมูลดังกล่าวก็คือสัญญาณควบคุมนั่นเอง ที่มาของสัญญาณควบคุมคือ การตีความคำสั่งที่อ่านขึ้นมาจาก Instruction memory เพื่อใช้เลือกทางเดินข้อมูลสำหรับการทำคำสั่งนั้นๆ เช่น คำสั่ง add \$8, \$17, \$18 ซึ่งเมื่อแปลเป็นภาษาเครื่องจะได้คำดังนี้

Instruction Format

000000	10001	10010	01000	00000	100000
Op	rs	rt	Rd	shamt	funct

ค่า rs และ rt จะถูกใช้เพื่อป้อนให้กับ รีจิสเตอร์เพื่อเลือกค่ารีจิสเตอร์ที่ต้องการป้อนเป็นตัวตั้งให้กับ ALU ในขณะที่ rd จะใช้ป้อนให้กับ ALU เพื่อเลือกค่ารีจิสเตอร์ที่ต้องการเขียน ส่วน funct จะใช้ป้อนให้กับ ALU เพื่อสั่งให้ทำการบวก

อย่างไรก็ตาม ในหลายๆ กรณีเราไม่สามารถที่จะนำค่าที่ได้จากชุดคำสั่งไปใช้เป็นสัญญาณควบคุมได้โดยตรง เนื่องจากชุดคำสั่งนั้นมีความซับซ้อนมาก เช่น กรณีของคำสั่ง load หรือ store (lw,sw) ดังตัวอย่างต่อไปนี้

lw \$1, 100(\$2)

100011	00010	00001	0000000001100100
Op	rs	rt	16 bit offset

ALU control input

000	ADD
001	OR
010	add
110	subtract
111	set-on-less-than

จะพบว่า ไม่มีข้อมูลส่วนใดที่พร้อมจะนำมาเป็นตัวป้อนให้กับ ALU เพื่อทำการบวกเลข ดังนั้นจึงจะต้องมีการถอดรหัส หรือผ่านวงจรตีความเพื่อหาค่าสำหรับใช้ป้อนเป็นสัญญาณควบคุมให้กับ ALU หรือทางเดินต่างๆ ดังแสดงได้ตามตารางข้างล่างต่อไปนี้

Instruction Type

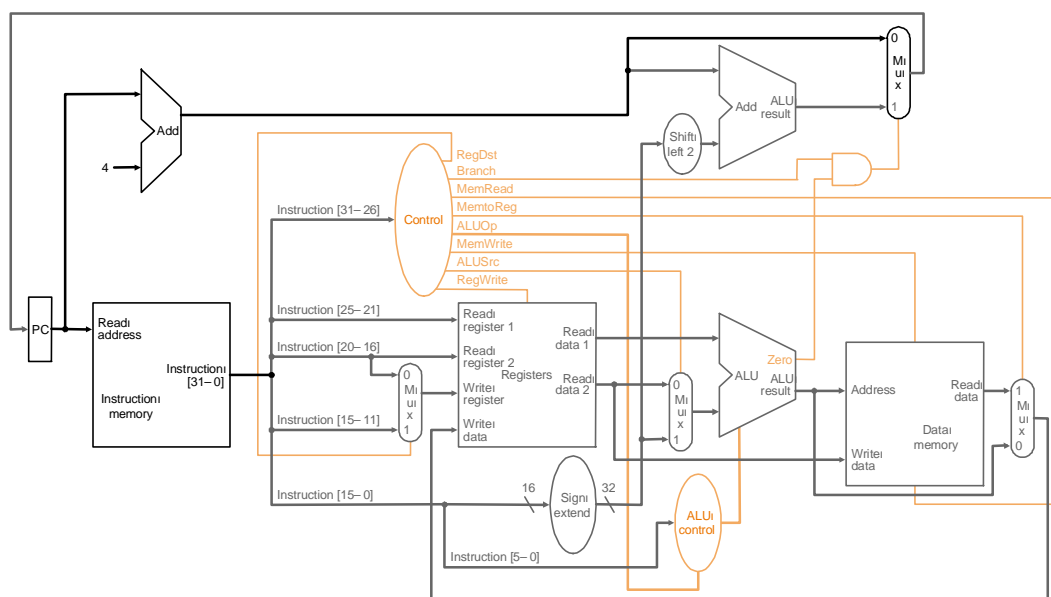
00 = lw, sw

01 = beq,

11 = arithmetic

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

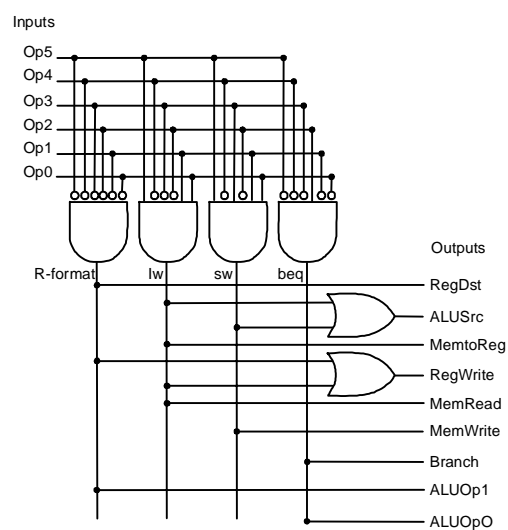
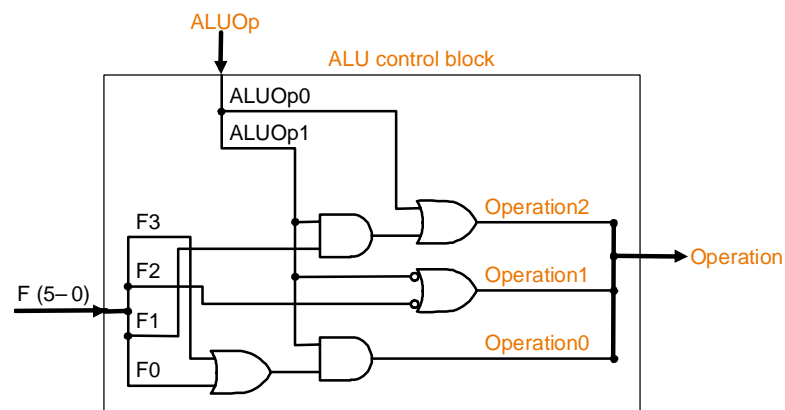
เมื่อประมวลสัญญาณควบคุมที่ใช้ควบคุม ALU เข้ากับสัญญาณควบคุมที่ใช้เลือกทางเดินแล้ว ลักษณะโครงสร้างภายในหน่วยประมวลผลกลางจะซับซ้อนยิ่งขึ้นโดยจะมีวงจรเพื่อถอดรหัสสัญญาณควบคุมการคำสั่งที่อ่านได้ ประกอบขึ้นเพื่อใช้เลือก Multiplexor ต่างๆ



ทั้งนี้ลักษณะของสัญญาณที่ใช้ควบคุมทางเดินข้อมูล ก็จะต้องผ่านวงจรสำหรับการถอดรหัสหรือตีความเช่นกัน โดยการตีความนี้จะขึ้นกับ Addressing Mode และกลุ่มของคำสั่งที่ใช้

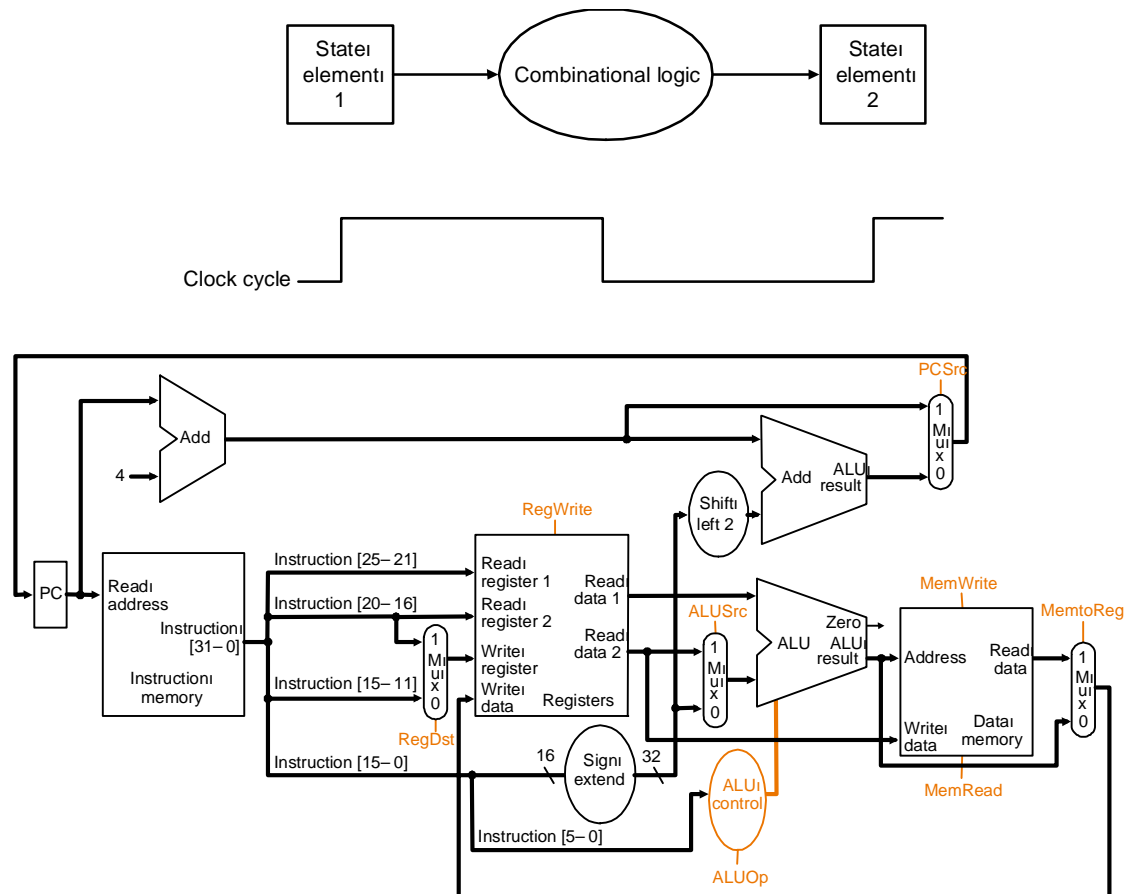
Instruction	Reg Dst	ALU SRC	Mem to Reg	Reg Write	Mem Read	Mem Write	Branch	ALU Op0	ALU Op1
R-Format	1	0	0	1	0	0	0	1	0
LW	0	1	1	1	1	0	0	1	0
SW	X	1	X	0	0	1	0	1	0
BEQ	X	0	X	0	0	0	1	0	1

ซึ่งเราอาจสร้างวงจรถอดรหัสเพื่อสร้างสัญญาณควบคุมได้จากการเขียนตารางค่าความจริง และ K-MAP ซึ่งผลลัพธ์ที่ได้ อาจจะเป็นวงจรในลักษณะดังต่อไปนี้



หน่วยประมวลผลกลางแบบ Single Cycle

เมื่อเราได้ทางเดินข้อมูลและนำสัญญาณควบคุมมาต่อพ่วงรวมกันแล้ว ก็นำมาประกอบกับสัญญาณนาฬิกา เพื่อให้จังหวะในการทำคำสั่งต่างๆ ทั้งนี้ ข้อควรระวังคือเวลาที่เว้นเพื่อให้วงจรเชิงประสมได้ทำงานจำต้องมากเพียงพอเพื่อให้ทุกอย่างทำงานเสร็จสิ้น มิเช่นนั้น ALU อาจจะทำงานผิดพลาด หรือการเขียนข้อมูลอาจจะไม่สมบูรณ์



หากหน่วยความจำทำงาน 10ns ALU และ รีจิสเตอร์ทำงาน 5ns แล้ว เราจะพบว่าหน่วยประมวลผลกลางดังกล่าวจะใช้เวลาในการทำงาน 1 คำสั่ง ประมาณ 35ns หรือคิดเป็น clock rate ประมาณ $1/(35 \times 10^{-9})$ Hz

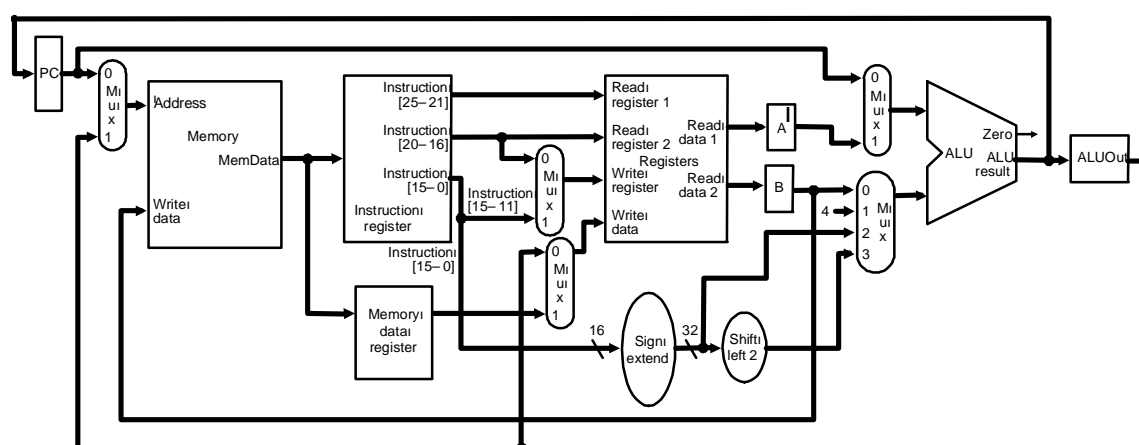
อย่างไรก็ตามหากวิเคราะห์ที่การทำงานของหน่วยประมวลผลกลางในรูปข้างต้น เราจะพบว่าการทำงานแต่ละคำสั่งจะต้องใช้เวลาในการทำงานเท่ากันคือ 35ns ทั้งนี้ ในบางคำสั่งนั้นสามารถทำเสร็จได้ภายในเวลาเพียง 25ns หรือ 30 ns เท่านั้น จึงได้มีความพยายามเพื่อลดเวลาในการทำงานดังกล่าว และ จากโครงสร้างที่แสดงนี้ยังพบว่ามอดูลประกอบหลายอย่างทำงานซ้ำกัน เช่น ALU และ Adder นั้น น่าจะสามารถทำงานแทนกันได้ หรือ Instruction memory และ Data memory น่าจะสามารถใช้มอดูลประกอบขึ้นเดียวกันได้ ทั้งนี้เพื่อให้หน่วยประมวลผลกลางที่ได้มีขนาดเล็กลง จึงเกิดแนวทางในการออกแบบหน่วยประมวลผลกลางในรูป

แบบอื่นๆ อีกมากมาย เพื่อที่จะให้หน่วยประมวลผลกลางมีขนาดเล็กลง หรือ มีความเร็วในการประมวลผลมากขึ้น

หน่วยประมวลผลกลางแบบ Multi Cycle

ลักษณะการออกแบบหน่วยประมวลผลกลางแบบ Multi Cycle นั้น ช่วยให้เราสามารถพัฒนาหน่วยประมวลผลกลางที่มีขนาดเล็กลง เพราะสามารถใช้องค์ประกอบบางอย่างร่วมกันได้ โดยอาศัยช่วงเหลือมของเวลา เช่น หากเราเปรียบเทียบ ALU และ Adder กับเครื่องคิดเลข เมื่อมีคน 2 คน ต้องการใช้เครื่องคิดเลขสำหรับการประมวลผลโดยคนหนึ่ง ต้องการใช้เครื่องคิดเลขสำหรับการบวก และ อีกคนหนึ่งใช้เครื่องคิดเลขสำหรับการคำนวณทั่วไป หากทั้ง 2 คนไม่มีความจำเป็นต้องใช้เครื่องคิดเลขพร้อมกัน เราจะพบว่า เครื่องคิดเลขเพียง 1 เครื่องก็เพียงพอสำหรับการใช้งานของคน 2 คนนี้ โดยการพลัดกันส่งเครื่องคิดเลขไปมาระหว่างคน 2 คน นอกจากนี้หากเราแบ่งการประมวลผลคำสั่ง 1 คำสั่งออกเป็นช่วงย่อยๆ เราอาจจะสามารถข้ามขั้นตอนบางขั้นที่ไม่จำเป็นในการทำงานนั้นๆ ได้ เพื่อให้การทำงานเร็วขึ้น ยิ่งไปกว่านั้นการออกแบบวงจรเพื่อทำงานในช่วงปัญหาที่แคบลง ยังสามารถแก้ไขและตรวจสอบได้ง่ายกว่าการออกแบบวงจรขนาดใหญ่

จากแนวทางในการออกแบบโดยใช้หน่วยประมวลผลบางส่วนร่วมกัน และการออกแบบ เพื่อให้การประมวลผลคำสั่งหนึ่งคำสั่งทำงานโดยใช้หลาย Cycle ของสัญญาณนาฬิกา จึงจำเป็นต้องมีการเพิ่มรีจิสเตอร์ภายในบางตัวเพื่อใช้ในการส่งผ่านค่าระหว่าง Cycle ทำให้ลักษณะของทางเดินข้อมูลนั้นถูกเปลี่ยนไปดังแสดงได้ดังภาพต่อไปนี้



ขั้นตอนในการประมวลผลคำสั่ง

ในการประมวลผลคำสั่งแต่ละคำสั่งนั้น เราสามารถแบ่งการทำงานออกเป็นขั้นตอนย่อยได้ประมาณ 5 ขั้นตอนทั้งนี้ขึ้นอยู่กับโครงสร้างสถาปัตยกรรมคอมพิวเตอร์ และ สถาปัตยกรรมชุด

คำสั่งที่ได้ทำการออกแบบไว้ ซึ่งในกรณีของ MIPS ตามโครงสร้างสถาปัตยกรรมชุดคำสั่งในที่นี้สามารถอธิบายได้ดังนี้

1. Instruction Fetch คือการดึงคำสั่งจากหน่วยความจำ (Memory) ตำแหน่งที่ชี้โดย PC เข้าสู่หน่วยประมวลผลกลาง พร้อมกับบวกค่า PC ให้ชี้ไปยังคำสั่งถัดไป
2. Instruction Decode และ Register Fetch คือการตีความคำสั่งที่ถูกอ่านเข้ามาแล้ว เพื่อสร้างสัญญาณควบคุมส่วนต่างๆ และ เตรียมข้อมูลใน Register ให้พร้อมสำหรับป้อนให้กับ ALU ต่อไป
3. Execution, Memory Address Computation หรือ Branch Completion ทั้งนี้การทำงาน Execution จะแตกต่างกันไปขึ้นอยู่กับคำสั่งที่อ่านขึ้นมาได้ เช่นกรณีคำสั่งที่อ่านขึ้นมาได้เป็นคำสั่งเกี่ยวกับการคำนวณ ALU ก็จะทำให้การคำนวณค่าตามคำสั่ง กรณีเป็นการอ่านข้อมูล ALU ก็จะทำให้การคำนวณตำแหน่งของหน่วยความจำที่ต้องการอ่านข้อมูล และ กรณีการ Branch ALU ก็จะทำให้คำนวณที่อยู่ของคำสั่งต่อไป (PC)
4. Memory Access หรือ R-type Instruction Completion กรณีที่คำสั่งซึ่งอ่านขึ้นมาได้นั้นเป็นคำสั่งเกี่ยวกับการอ่านเขียนข้อมูลจากหน่วยความจำ ในขั้นตอนนี้จะมีการอ่านเขียนข้อมูล และในกรณีคำสั่งที่อ่านขึ้นมาได้ เป็นคำสั่งในการคำนวณเกี่ยวกับรีจิสเตอร์ ในขั้นนี้จะทำการนำผลลัพธ์ที่ได้ไปเก็บในรีจิสเตอร์
5. Write back ซึ่งจะเกิดขึ้นเฉพาะกรณีการอ่านข้อมูลจากหน่วยความจำ คือการนำข้อมูลที่อ่านได้ไปเก็บในรีจิสเตอร์

จากการทำงานใน 5 ขั้นตอนดังกล่าวข้างต้นนั้น เรายังไม่สามารถที่จะนำไปสร้างหน่วยควบคุมได้ทันที ทั้งนี้เนื่องจากหลักการทำงานข้างต้นมิได้อธิบายถึงที่มาของข้อมูลแต่ละช่วงโดยละเอียด เพราะยังมีได้แสดงให้เห็นว่า จะต้องนำบิตใดของคำสั่งที่อ่านขึ้นมาใดมาใช้อ้างอิง หรือนำข้อมูลที่ได้ไปเก็บที่รีจิสเตอร์ใด เป็นต้น ในการออกแบบเราจึงนิยมอธิบายการทำงานโดยอาศัยภาษากลางที่มีความสามารถในการแยกแยะแจกแจงการทำงานได้ในระดับรีจิสเตอร์ หรือ Register-Transfer Language (RTL) ซึ่ง RTL ดังกล่าวนี้นี้สามารถอธิบายการทำงานทั้ง 5 ขั้นตอนข้างต้นได้ดังนี้

1. Instruction Fetch

$IR = Memory[PC];$

$PC = PC + 4;$

ซึ่งมีความหมายว่า นำข้อมูลออกจากหน่วยความจำตำแหน่งที่ชี้โดย PC มาเก็บไว้ที่ รีจิสเตอร์ IR และ นำค่า PC บวกกับ 4 พร้อมๆ กัน

2. Instruction Decode และ Register Fetch

$A = \text{Reg}[\text{IR}[25-21]];$

$B = \text{Reg}[\text{IR}[20-16]];$

$\text{ALUOut} = \text{PC} + (\text{sign-extend}(\text{IR}[15-0]) \ll 2);$

3. การประมวลผลซึ่งแตกต่างกันไปในแต่ละคำสั่ง

Memory Reference:

$\text{ALUOut} = A + \text{sign-extend}(\text{IR}[15-0]);$

R-type:

$\text{ALUOut} = A \text{ op } B;$

Branch:

if $(A == B)$ $\text{PC} = \text{ALUOut};$

4. Memory Access หรือ R-type Instruction Completion

Loads and stores access memory

$\text{MDR} = \text{Memory}[\text{ALUOut}];$

หรือ

$\text{Memory}[\text{ALUOut}] = B;$

R-type instructions finish

$\text{Reg}[\text{IR}[15-11]] = \text{ALUOut};$

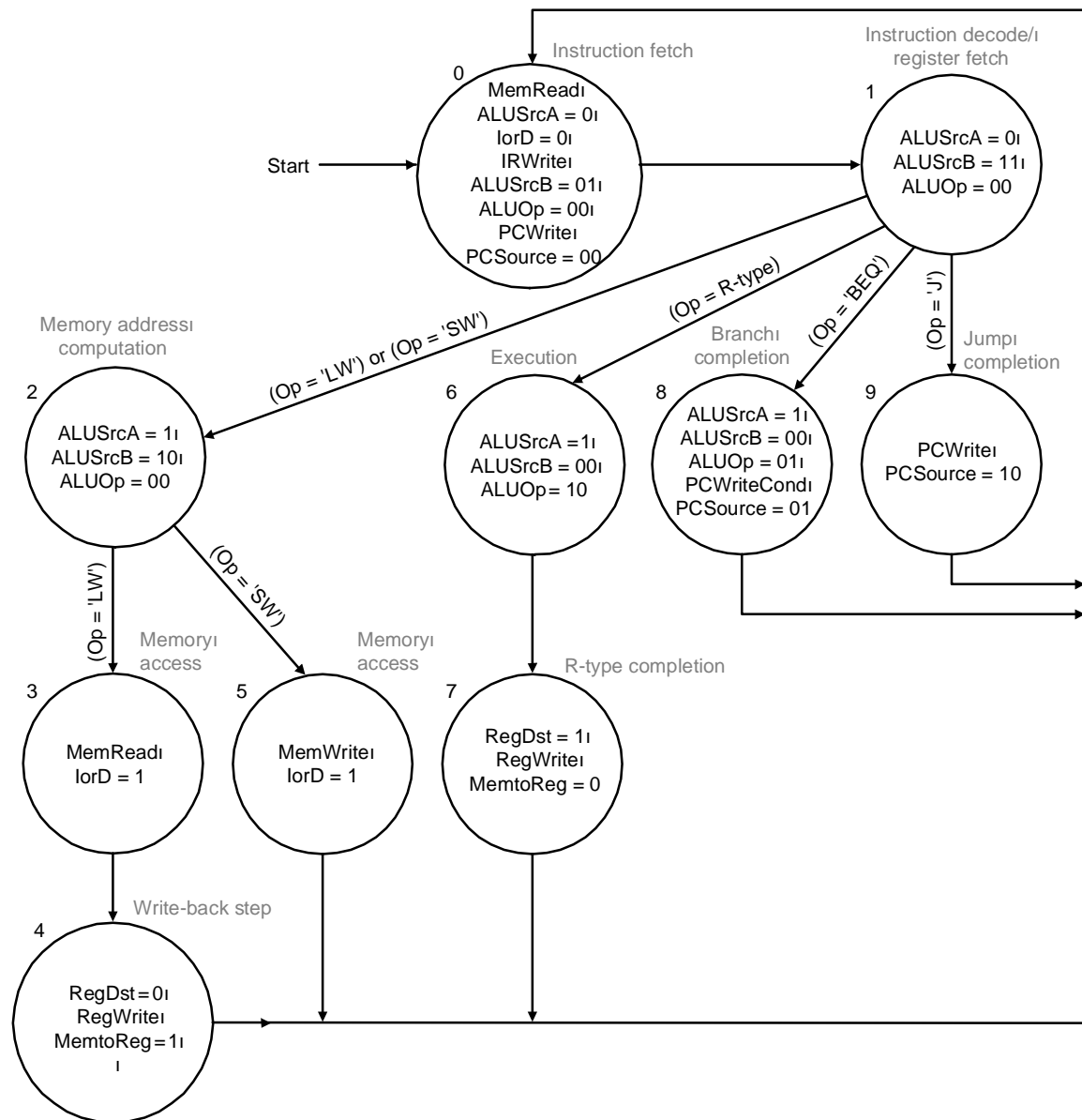
5. Write Back

$\text{Reg}[\text{IR}[20-16]] = \text{MDR};$

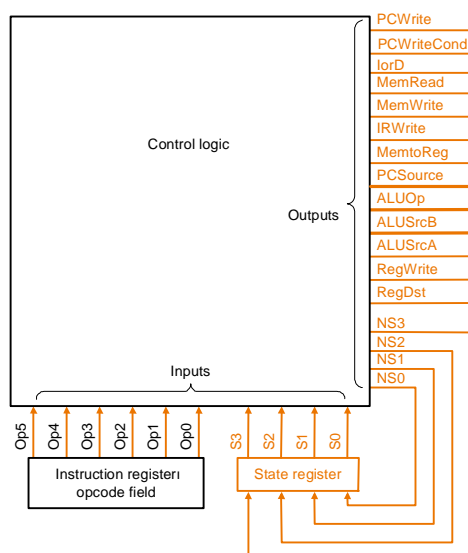
เมื่อนำทุกขั้นตอนมารวมกันและจำแนก RTL ตามลักษณะของกลุ่มคำสั่งที่ทำการประมวลผลแล้ว RTL ดังกล่าว จะสามารถแสดงได้ดังตารางต่อไปนี้

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	$\text{IR} = \text{Memory}[\text{PC}]$ $\text{PC} = \text{PC} + 4$			
Instruction decode/register fetch	$A = \text{Reg}[\text{IR}[25-21]]$ $B = \text{Reg}[\text{IR}[20-16]]$ $\text{ALUOut} = \text{PC} + (\text{sign-extend}(\text{IR}[15-0]) \ll 2)$			
Execution, address computation, branch/jump completion	$\text{ALUOut} = A \text{ op } B$	$\text{ALUOut} = A + \text{sign-extend}(\text{IR}[15-0])$	if $(A == B)$ then $\text{PC} = \text{ALUOut}$	$\text{PC} = \text{PC} [31-28] \parallel (\text{IR}[25-0] \ll 2)$
Memory access or R-type completion	$\text{Reg}[\text{IR}[15-11]] = \text{ALUOut}$	Load: $\text{MDR} = \text{Memory}[\text{ALUOut}]$ or Store: $\text{Memory}[\text{ALUOut}] = B$		
Memory read completion		Load: $\text{Reg}[\text{IR}[20-16]] = \text{MDR}$		

จากความรู้เดิมในการออกแบบวงจรเชิงลำดับ เราจะเริ่มต้นการออกแบบโดยการสร้างแผนภาพสถานะ (State Diagram) หรือ การเขียน ASM Chart เพื่อแสดงการเคลื่อนไหวและเงื่อนไขในการเปลี่ยนสถานะต่างๆ สร้างฟังก์ชันเอาต์พุต และ ฟังก์ชันการเปลี่ยนสถานะ ซึ่งแผนภาพสถานะดังกล่าวของหน่วยประมวลผลกลางนี้ สามารถแสดงได้ดังนี้

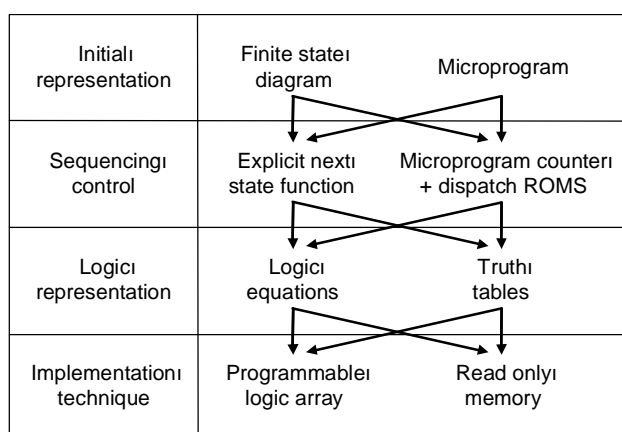


ซึ่งเราอาจจะนำแผนภาพสถานะดังกล่าวมาออกแบบเป็นวงจรได้โดยการใช้วงจรเชิงผสม ROM หรือ PLA ก็ได้ ทั้งนี้ขึ้นอยู่กับผู้ออกแบบ ซึ่งลักษณะของหน่วยควบคุมที่ได้ จะมีอินพุตเป็น Instruction Register และสถานะของหน่วยควบคุม ส่วนเอาต์พุตจะเป็นสัญญาณสำหรับควบคุม Multiplexor และ ALU พร้อมกับการเปลี่ยนสถานะ ซึ่งลักษณะของหน่วยควบคุมนี้สามารถแสดงได้ดังภาพ โดยในที่นี้จะไม่กล่าวถึงรายละเอียดในการออกแบบมากนัก หากผู้อ่านสนใจเพิ่มเติมสามารถศึกษาได้จากการออกแบบวงจรตรรกะเชิงลำดับทั่วไป



Microprogramming

อย่างไรก็ตาม แม้ว่าจะมีแนวทางการออกแบบโดยใช้ ROM หรือ PLA แล้ว แต่การออกแบบหรือแก้ไขหน่วยควบคุมก็ยังคงทำได้ยากลำบาก ดังนั้นจึงเกิดแนวทางในการออกแบบส่วยควบคุมของหน่วยประมวลผลกลางในลักษณะของการเขียนโปรแกรม ซึ่งช่วยอำนวยความสะดวกในการแก้ไขปัญหาและตรวจสอบข้อผิดพลาดระหว่างการออกแบบ โดยการเขียน microinstruction ทำการแปลด้วย microassembler แล้วทำการโหลดลงสู่หน่วยประมวลผลกลาง ซึ่งนอกจากจะอำนวยความสะดวกในการออกแบบแล้ว ยังช่วยให้เราสามารถที่จะปรับปรุงหรือจำลองหน่วยประมวลผลกลางให้ทำงานแทนหน่วยประมวลผลกลางที่มีสถาปัตยกรรมแบบอื่นๆ ได้โดยง่ายอีกด้วย ทั้งนี้หากผู้อ่านมีความสนใจในการออกแบบหน่วยประมวลผลกลางโดยใช้ Microprogram สามารถศึกษาเพิ่มเติมได้จากเอกสารของผู้ผลิตต่างๆ ได้ทั่วไป



แบบฝึกหัดท้ายบท

1. จงอธิบายส่วนประกอบต่างๆ ภายในหน่วยประมวลผลกลาง
2. จงอธิบายขั้นตอนในการประมวลผลคำสั่งหนึ่งคำสั่งของหน่วยประมวลผลกลาง
3. การออกแบบหน่วยประมวลผลกลางโดยใช้ Single Cycle และ Multi Cycle Approach มีข้อดีข้อเสียต่างกันอย่างไร จงยกตัวอย่างประกอบ
4. Data path คือ อะไร และ Control คืออะไร มีความเหมือนหรือมีความสัมพันธ์กันหรือไม่ อย่างไร
5. ในการประมวลผลคำสั่งแต่ละคำสั่งนั้น หน่วยประมวลผลกลางมีขั้นตอนในการทำงานอย่างไร
6. จากภาษาแอสเซมบลีดังกล่าว

```
lw $t2, 0($t3)
```

```
lw $t3, 4($t3)
```

```
beq $t2, $t3, end_program          #assume not
```

```
add $t5, $t2, $t3
```

```
sw $t5, 8($t3)
```

```
end_program: ....                # end of program
```

หากเราออกแบบหน่วยประมวลผลกลางให้ทำงานแบบ Multi Cycle จงตอบคำถามต่อไปนี้

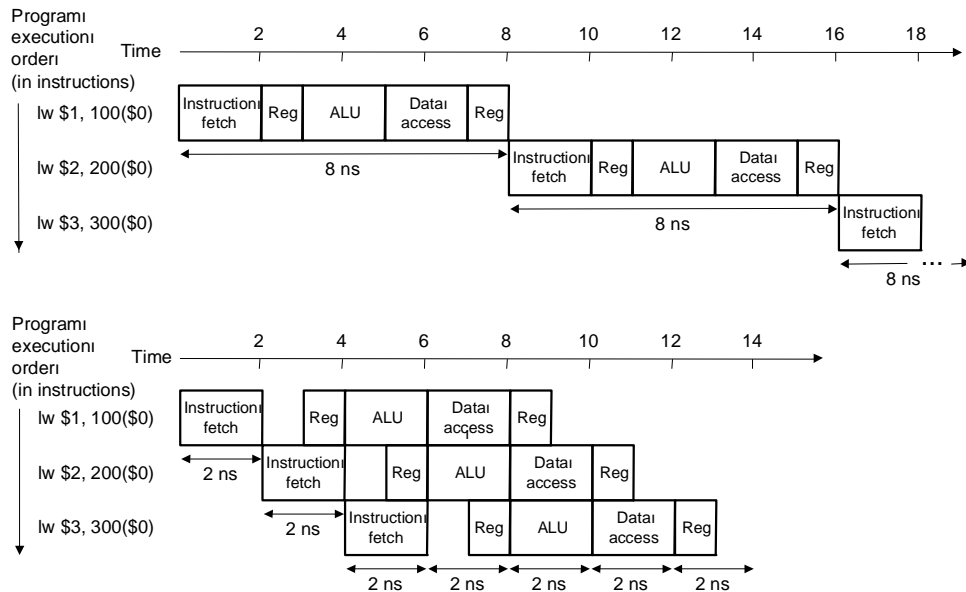
- 6.1. การทำงานในแต่ละคำสั่งจะใช้เวลากี่ Cycle
 - 6.2. ที่ Cycle ที่ 7 หน่วยประมวลผลกลางกำลังทำงานอะไรอยู่
 - 6.3. โปรแกรมดังกล่าวข้างต้นจะใช้เวลาในการทำงานทั้งสิ้นกี่ Cycle และคิดเป็นกี่วินาที
- หาก Cycle Time เป็น 40ns
7. การออกแบบหน่วยประมวลผลกลางโดยใช้ microprogram มีข้อดีข้อเสียหรือไม่ อย่างไร ?

บทที่ 6 การเพิ่มประสิทธิภาพด้วย Pipeline

การทำ Pipeline นั้นเป็นการเพิ่มประสิทธิภาพให้กับหน่วยประมวลผลกลางในลักษณะของการเพิ่ม Throughput กล่าวคือ มีการทำงานหลายคำสั่งพร้อมกันในเวลาหนึ่งๆ ตัวอย่างเช่น สมมติว่าการเตรียมข้าวสาร ประกอบด้วยขั้นตอนย่อย 3 ขั้นตอน คือ ขั้นที่ 1 การร่อนข้าวเปลือกเพื่อคัดสิ่งแปลกปลอมออกที่ปนมากับข้าวเปลือก ขั้นที่ 2 นำข้าวเปลือกที่ได้จากขั้นแรกมาตำเพื่อให้เมล็ดข้าวหลุดออกจากเปลือกข้าว และ ขั้นที่ 3 ทำการแยกเปลือกข้าวออก เพื่อให้เหลือแต่เมล็ดข้าว หากเรามีกระตังสำหรับร่อนแยกสิ่งแปลกปนในขั้นที่ 1 อยู่เพียง 1 อัน มีครกกระตังสำหรับตำให้เมล็ดข้าวแยกออกจากเปลือกเพียง 1 อัน และมีพานะสำหรับทำการแยกเมล็ดข้าวออกจากเปลือกข้าวเพียง 1 อัน หากเรามีคนทำงานเพียง 1 คน อาจจะต้องใช้เวลาในการทำงานทั้งหมด 3 ชั่วโมง แต่หากเรามีคนทำงาน 3 คน แบ่งกันทำงานในแต่ละขั้นตอน แล้วส่งต่อ อาจจะใช้เวลาในการทำงานเพียง 1 ชั่วโมงครึ่ง ทั้งนี้เนื่องจากหากมีคนทำงานเพียง 1 คน จะต้องแยกสิ่งแปลกปนให้เสร็จก่อน จากนั้นจึงจะทำการตำเพื่อให้เมล็ดข้าวหลุดออกจากเปลือกข้าว แล้วจึงมาแยกทั้ง 2 ส่วนออกจากกัน แต่หากมีคนทำงาน 3 คน เมื่อคนแรกแยกสิ่งแปลกปน แล้วส่งข้าวเปลือกที่แยกได้มากให้คนที่ 2 ตำนั่น คนแรกก็สามารถที่จะแยกข้าวเปลือกได้ต่อไป พร้อมกับที่คนที่ 2 กำลังตำข้าวเปลือก และเมื่อคนที่ 2 ตำข้าวเปลือกเสร็จแล้วส่งให้คนที่ 3 แยกเมล็ดข้าวและเปลือกข้าว คนที่ 2 ก็พร้อมที่จะรับข้าวเปลือกที่แยกสิ่งแปลกปนแล้วจากคนแรกมาทำงานต่อได้ ส่งผลให้งานทุกขั้นตอนสามารถทำได้พร้อมๆ กัน เป็นต้น

ในหน่วยประมวลผลกลางนั้น การเพิ่ม Throughput โดยการทำ Pipeline ก็อยู่ในลักษณะเดียวกัน คือ เมื่อคำสั่งแรกถูก Fetch ขึ้น และ ถูกส่งไปทำการ Decode ในช่วงเวลานี้หน่วยประมวลผลสำหรับการ Fetch ก็กำลัง Fetch คำสั่งถัดไปขึ้นมา และเมื่อคำสั่งแรกทำการ Execute คำสั่งที่ 2 ก็อย่างระหว่างการ Decode และ หน่วย Fetch ก็กำลังทำการ Fetch คำสั่งที่ 3 ขึ้นมา ซึ่งส่งผลให้มีคำสั่งที่ถูกทำงานโดยหน่วยประมวลผลกลางมากกว่าหนึ่งคำสั่งในเวลาเดียวกัน ดังแสดงในภาพ

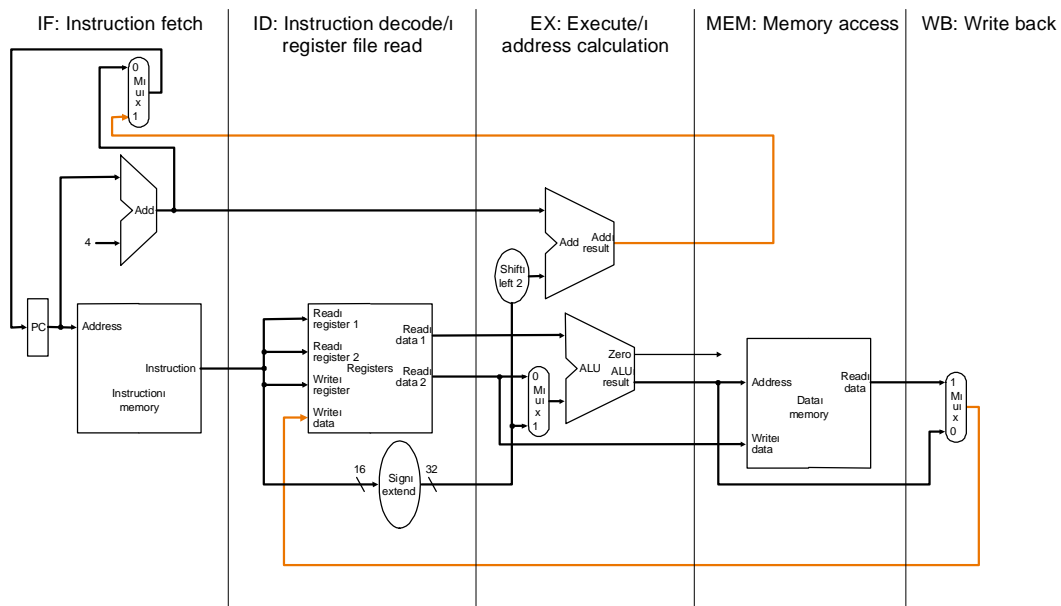
จากภาพจะพบว่าการทำงาน 3 คำสั่งกรณีไม่มี Pipeline นั้น จะใช้เวลาในการทำงานถึง 24 หน่วยเวลา ในขณะที่เมื่อมีการทำ Pipeline จะใช้เวลาเพียง 14 หน่วยเวลาเท่านั้น อย่างไรก็ตาม จะสังเกตได้ว่า แต่ละคำสั่งไม่สามารถ Fetch หรือถูกอ่านขึ้นมาพร้อมกันได้ ทั้งนี้เนื่องจากแต่ละหน่วยของหน่วยประมวลผลกลางสามารถให้บริการได้เพียงหนึ่งคำสั่งในเวลาหนึ่งๆ เท่านั้น



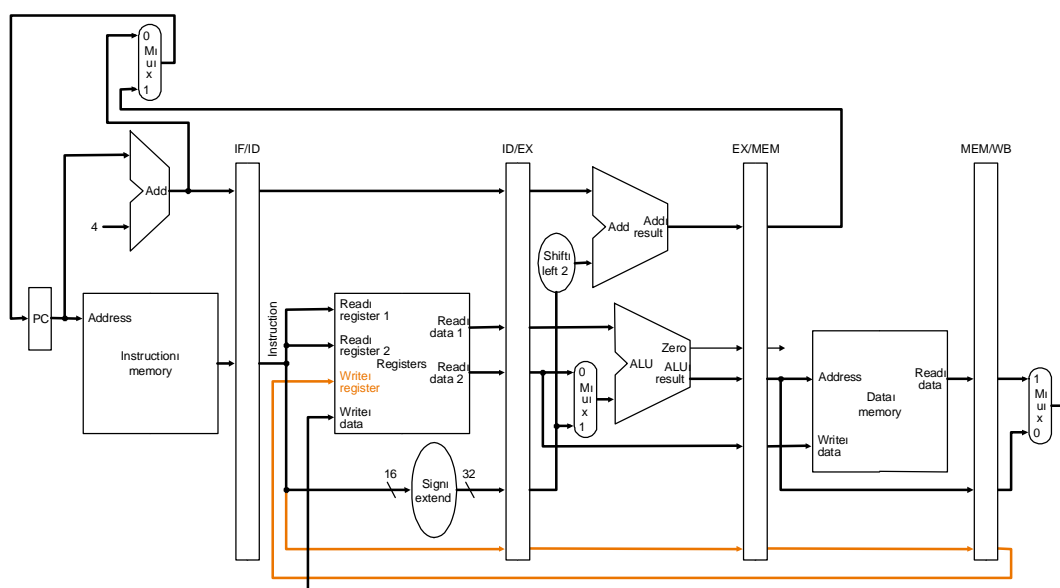
เนื้อหาที่กล่าวถึงในบทนี้จะกล่าวถึงเพียงการสร้างระบบ Pipeline แบบง่าย โดยอยู่บนสมมุติฐานที่ว่าคำสั่งทุกคำสั่งมีความยาวเท่ากัน มีรูปแบบคำสั่งที่จำกัด (3 รูปแบบ) และมีคำสั่งที่เกี่ยวข้องกับหน่วยความจำเพียง Load และ Store เท่านั้น ทั้งนี้หากผู้อ่านต้องการทราบซึ่งสถาปัตยกรรมที่ซับซ้อน สามารถหาอ่านได้ตาม Reference ของผู้ผลิตหน่วยประมวลผลกลางทั่วไป

การปรับเปลี่ยน Datapath และ Control เพื่อการทำ Pipeline

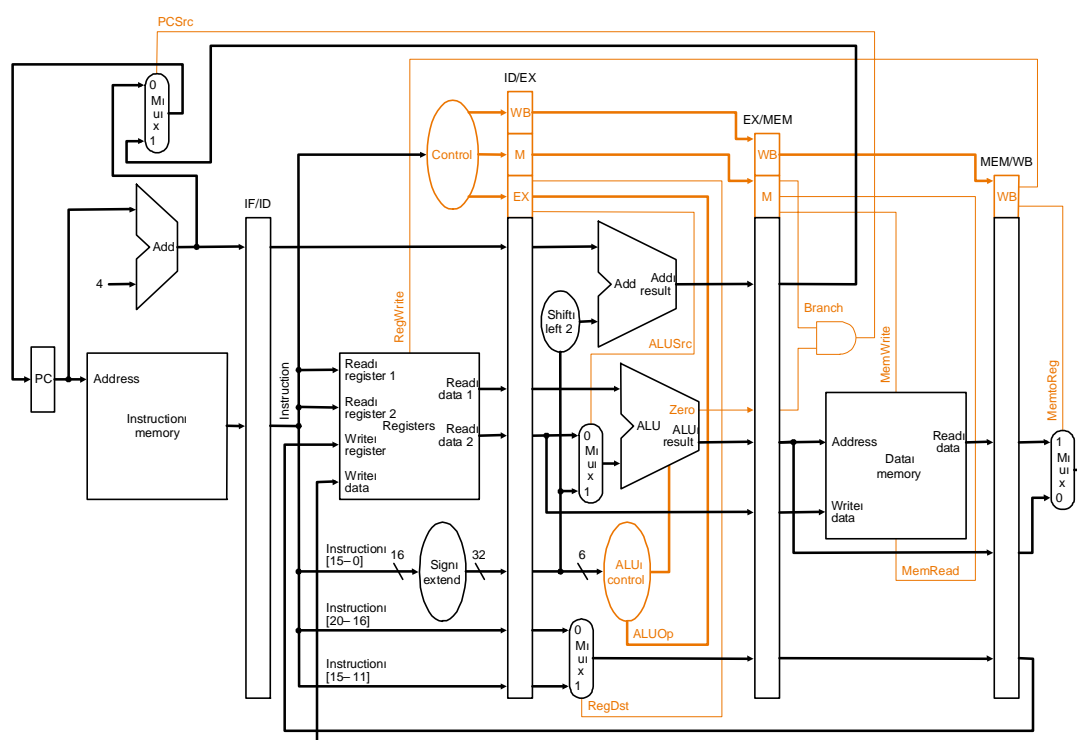
หากเราวิเคราะห์การทำงานในลักษณะของ Pipeline ให้ดีจะพบว่าการทำงาน Pipeline นั้นจะกระทำได้ดีก็ต่อเมื่องานในแต่ละขั้นตอนแยกเป็นอิสระจากกันได้ ดังนั้นในการออกแบบหน่วยประมวลผลกลางให้สามารถทำงานแบบ Pipeline ได้นั้น ปัญหาอย่างแรกคือการทำให้งานในแต่ละขั้นตอนเป็นอิสระจากกัน เช่นกรณีของการเตรียมข่าวสาร หากขั้นตอนที่ 1 และขั้นตอนที่ 3 ต้องใช้กระดิ่งเหมือนกัน แต่เรามีกระดิ่งเพียง 1 อันเท่านั้น เช่นนี้แล้ว ก็คงไม่สามารถที่จะแบ่งการกันทำในลักษณะของ Pipeline ได้ ปัญหาในลักษณะดังกล่าวนี้เรียกว่า Structural Hazard (ดังจะกล่าวถึงเพิ่มเติมในหัวข้อถัดไป) ด้วยเหตุนี้ปัจจัยแรกคือการแยกให้แต่ละขั้นตอนสามารถทำงานได้เป็นอิสระจากกัน ทำให้โครงสร้างภายในของ CPU ที่จะทำ Pipeline นั้นมีอุปกรณ์ที่มากขึ้น และทำงานซ้ำซ้อนกันบ้าง (ซึ่งแนวทางการใช้อุปกรณ์ที่ซ้ำซ้อนกันนี้ จะขัดแย้งกับแนวทางในการออกแบบหน่วยประมวลผลกลางแบบ Multi-cycle ในบทที่ 5 อันเป็นการลดอุปกรณ์ซ้ำซ้อนเพื่อให้ CPU มีขนาดเล็ก) ทั้งนี้เนื่องจากอุปกรณ์ทุกชิ้นต้องทำงานพร้อมกัน ดังแสดงได้ดังภาพ



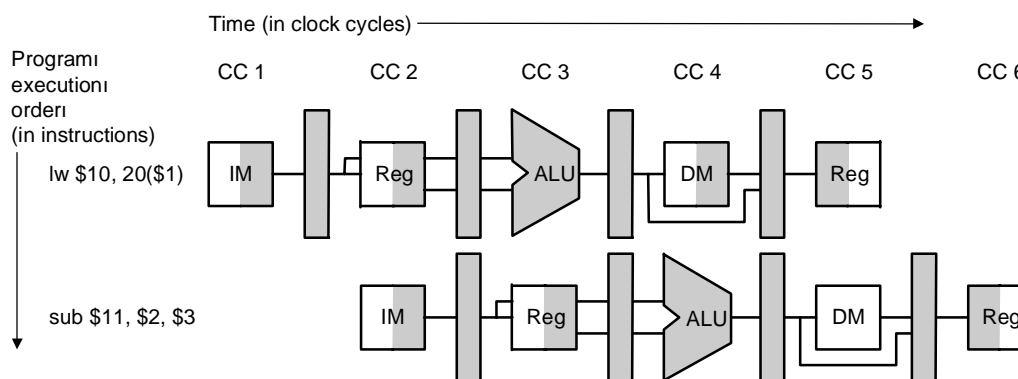
หากเราวิเคราะห์ถึงทางเดินข้อมูลที่เชื่อมต่อระหว่างอุปกรณ์เพิ่มเติม ยังพบอีกว่าการทำงานในบางรูปแบบ หรือ บางคำสั่งของหน่วยประมวลผลกลางจะเป็นอุปสรรคต่อการทำ Pipeline อีก เช่น การทำงานของคำสั่ง Branch และ การเขียนข้อมูลกลับไปใน Register (อันจะถึงในหัวข้อถัดไป) อย่างไรก็ตามปัจจัยอย่างแรกที่ต้องคำนึงถึงคือการให้ข้อมูลซึ่งทำงานในแต่ละขั้นตอนของแต่ละคำสั่ง ไม่ปะปนกัน มิเช่นนั้นเราจะไม่ทราบว่าคุณสมบัติในสายขณะนั้น เป็นของข้อมูลการทำงานของคำสั่งใด ด้วยเหตุนี้ในเบื้องต้นระบบ Data Path ของ Pipeline จึงต้องมีการใส่ Buffer ให้แต่ละส่วนซึ่งประมวลผลในแต่ละขั้นตอนของคำสั่งเป็นอิสระแยกจากกัน ดังแสดงได้ดังภาพ



เมื่อแยกทางเดินข้อมูลให้เป็นอิสระจากกันได้แล้ว อุปสรรคอีกอย่างหนึ่งคือ การแยกสัญญาณควบคุมของแต่ละคำสั่งให้เป็นอิสระจากกัน ดังจะสังเกตได้จากภาพซึ่งพบว่าสัญญาณควบคุมที่ใช้แต่ละขั้นตอนนั้น แต่เดิมถูกรวบรวมเป็นชิ้นเดียวกัน ไม่สามารถแยกออกจากกันได้ เพื่อให้สัญญาณควบคุมในแต่ละขั้นตอนแยกเป็นอิสระจากกันได้ จึงต้องมีการส่งผ่านสัญญาณควบคุมที่ Decode ได้ของแต่ละคำสั่งไปกับแต่ละขั้นตอนของ Pipeline ด้วย ดังแสดงในเส้นทางสีส้มของภาพ อย่างไรก็ตามหากสังเกตในภาพจะพบว่า สัญญาณควบคุมบางอย่างนั้นเป็นของคำสั่งก่อนๆ ซึ่งภายใน Pipeline ก่อนหน้าคำสั่งปัจจุบัน แต่ถูกป้อนกลับมาใช้คำสั่งปัจจุบัน เช่น สัญญาณ Register Write หรือ สัญญาณเลือก PC เป็นต้น



เพื่อเป็นความสะดวกในการอธิบาย Pipeline ในทางปฏิบัติจึงนิยมใช้รูปภาพในการอธิบายแทน โดยแต่ละ Block แสดงการทำงานของ Pipeline ในแต่ละส่วนซึ่งจะขึ้นด้วย Buffer และ ภาพที่มีการแรเงาทางด้านซ้ายจะแสดงถึงการเขียนข้อมูล ส่วนภาพที่มีการแรเงาด้านขวาจะแสดงถึงการอ่านข้อมูล ส่วนองค์ประกอบใดที่มีการใช้งานเพื่อประมวลผลทั้งหมดจะแรเงาทั้งหมด และ กรณีที่องค์ประกอบใดไม่มีการใช้งานจะแสดงเพียงส่วนประกอบเท่านั้น โดยไม่มีการแรเงาใดๆ ทั้งสิ้น



เช่น กรณีของคำสั่ง `lw $10, 20($1)` จะมีการ Fetch หรือ อ่านคำสั่งจาก Instruction Memory มีการอ่านข้อมูลจากเรจิสเตอร์ที่ 1 ใน Cycle ที่ 2 ใช้ ALU ประมวลผลและอ่านข้อมูลจาก Data Memory ใน Cycle ที่ 4 เป็นต้น

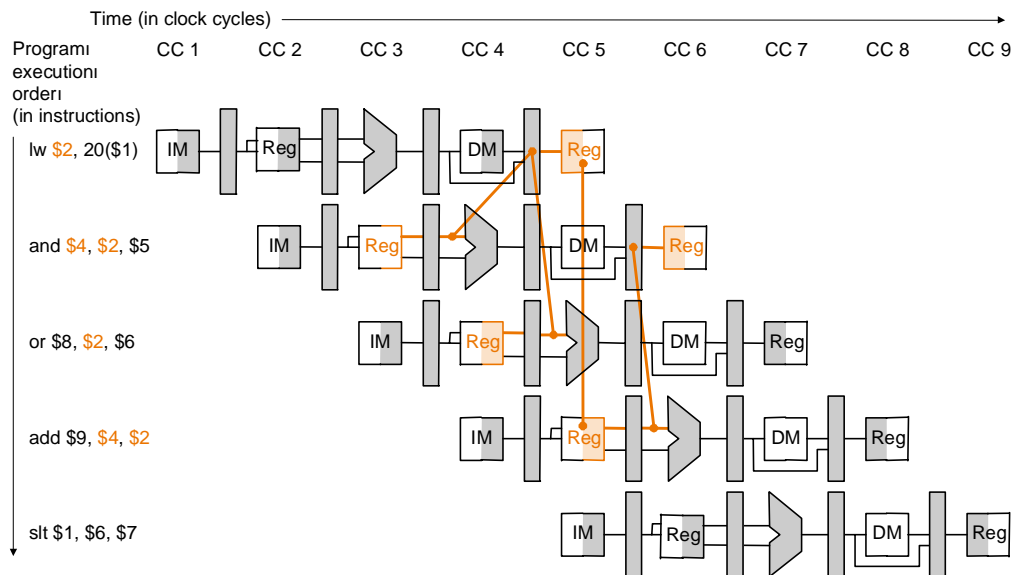
ปัญหาที่เป็นอุปสรรคต่อการทำ Pipeline

ในการทำ Pipeline มีปัญหาหลักที่เป็นอุปสรรคต่อประสิทธิภาพและการทำงานของ Pipeline ทั้งสิ้น 3 ประเด็น คือ ปัญหากรณีโครงสร้างของระบบหรือ Resource ต่างๆ ต้องใช้งานร่วมกัน (Structural Hazard) ปัญหาของการประมวลผลข้อมูลซึ่งมีความเกี่ยวเนื่องกัน (Data Hazard) และ ปัญหาการหาคำสั่งถัดไปของระบบในกรณีการ Jump หรือ Branch (Control Hazard) ซึ่งในที่นี้จะกล่าวถึงปัญหาแต่ละอันและการแก้ปัญหาคร่าวๆ โดยใช้วิธีการพื้นฐาน ดังต่อไปนี้

Structural Hazard

Structural Hazard เป็นปัญหาที่เกิดจากโครงสร้างของระบบไม่รองรับการทำ Pipeline เช่น กรณีของหน่วยความจำซึ่งทำหน้าที่เป็นทั้ง Instruction Memory และ Data Memory พร้อมๆ กัน โดยหากเราพิจารณาภาพข้างล่างและลองวิเคราะห์ต่อไปจะพบว่าที่ Cycle ที่ 4 นั้น Memory จะถูกใช้งานเป็น Instruction Memory เพื่อทำการ Fetch และ Data Memory เพื่อทำการอ่านข้อมูลพร้อมกัน ซึ่งปัญหานี้อาจจะแก้ได้โดยการเพิ่ม Instance ขององค์ประกอบนั้นๆ ให้สามารถให้บริการพร้อมๆ กันได้ ในกรณีนี้การเพิ่ม Instance อาจทำได้โดยการแยก Instruction Memory และ Data Memory ให้ต่อกับ Bus ที่เป็นอิสระจากกัน หรือ แยก Cache ของ Instruction Memory กับ Data Memory ออกจากกัน (ดังจะกล่าวต่อไปในบทที่ 7)

อีกกรณีหนึ่งของ Structural Hazard ที่ได้ทำการแก้ไขไปแล้วคือ กรณีที่หน่วยประมวลผลกลางต้องใช้ ALU เพื่อการบวก PC ในการทำ Instruction Fetch และ ทำการ Execute คำสั่งพร้อมกัน หากมี ALU เพียง 1 ตัวจะทำให้ไม่สามารถทำ Pipeline ได้ ซึ่งแก้ไขโดยการเพิ่ม Instance หรือ เพิ่ม ALU ให้ทำงานแยกเป็นอิสระจากกัน 2 ตัว เป็นต้น



Data Hazard

Data Hazard นั้น เป็นปัญหาที่เกิดจากการประมวลผลคำสั่งที่มีความเกี่ยวเนื่องกัน (Dependency) โดยหากเราวิเคราะห์จากภาพข้างบน เราจะพบว่ากรณีการทำงานของคำสั่ง LW อันที่หนึ่งนั้น ข้อมูลที่ได้จากหน่วยความจำตำแหน่งที่ 20+\$1 จะถูกนำขึ้นมาเก็บในเรจิสเตอร์ \$2 ใน Cycle ที่ 5 (หากผู้อ่านไม่ทราบในแต่ละ Cycle ของ แต่ละคำสั่งมีการทำงานอย่างไร ให้ลองกลับไปพิจารณาขั้นตอนการทำงานในบทที่ 5 อีกครั้งหนึ่ง) ทั้งนี้เนื่องจากการเป็นการทำงานที่เกี่ยวข้องกับหน่วยความจำ ซึ่งจะได้ข้อมูลออกจากหน่วยความจำในตอนจบของ Cycle ที่ 4 ในขณะที่คำสั่งถัดมาต้องการใช้งานเรจิสเตอร์ \$2 และเรจิสเตอร์ \$5 ใน Cycle ที่ 2 ทั้งนี้ข้อมูลเรจิสเตอร์ \$2 นั้นจะได้มาจากการทำงานของคำสั่งแรกซึ่งจะเสร็จสิ้นใน Cycle ที่ 5 ซึ่งระบบไม่สามารถนำข้อมูลที่จะได้รับในอนาคตมาประมวลผลได้ ดังแสดงด้วยเส้นสีส้มในภาพ

หากเรจิสเตอร์เดียวกันสามารถถูกอ่านและเขียนได้พร้อมกันโดยการประมวลผลข้อมูลถูกต้องนั้น เราจะพบว่าหากการทำงานของคำสั่งที่ 2 นั้น ช้ามาอีก 2 Cycle จะทำให้การประมวลผลของหน่วยประมวลผลกลางถูกต้อง ดังนั้นการแก้ปัญหาของ Data Hazard อาจทำได้ด้วยซอฟต์แวร์โดยการแทรกคำสั่งอื่นๆ ที่ไม่มีการใช้ข้อมูลที่เกี่ยวข้องกันเข้าไประหว่างคำสั่งที่ 1 และ 2 ทั้งนี้ต้องไม่ทำให้การทำงานของโปรแกรมผิดแปลกไปจากเดิม ด้วยเหตุนี้คำสั่งที่นิยมแทรกมากที่สุดคือ nop (No Operation) อันมีความหมายว่าไม่มีการประมวลผลใดๆ หรืออยู่เฉยๆนั่นเอง

จากเดิม

lw \$2,20(\$1)

and \$4,\$2,\$5

แก้ไขเป็น

```
lw    $2,20($1)
nop
nop
and    $4,$2,$5
```

เป็นต้น

อย่างไรก็ตามข้อเสียของการแทรกคำสั่ง nop เข้าไปนั้น ทำให้โปรแกรมมีขนาดใหญ่และสิ้นเปลืองเวลาในการทำงานมากขึ้น ด้วยเหตุนี้จึงมีการพัฒนา Compiler ให้สามารถทำการ Optimize คำสั่งให้เหมาะสมกับการทำงานของหน่วยประมวลผลกลางที่มี Pipeline โดยการเรียงลำดับคำสั่งใหม่ เช่น

จากเดิม

```
lw    $2,20($1)
and    $4,$2,$5
or    $5,$5,$1
add    $6,$6,$1
xor    $9,$9,$9
```

แก้ไขโดยการเรียงคำสั่งใหม่เป็น

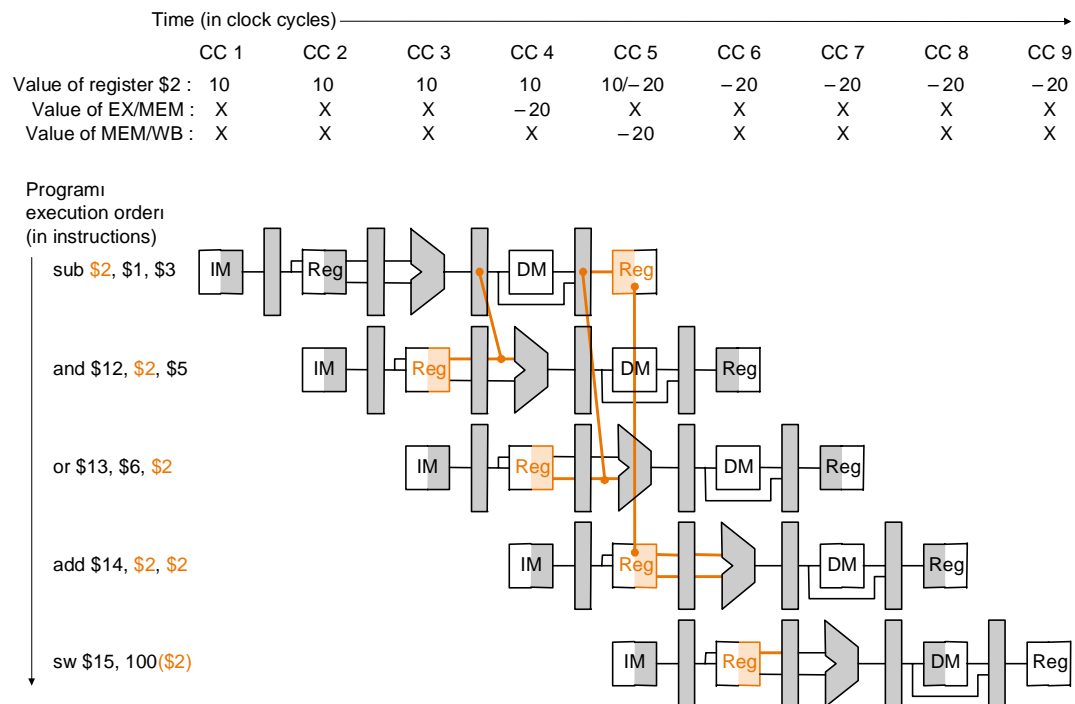
```
lw    $2,20($1)
add    $6,$6,$1
xor    $9,$9,$9
and    $4,$2,$5
or    $5,$5,$1
```

เป็นต้น

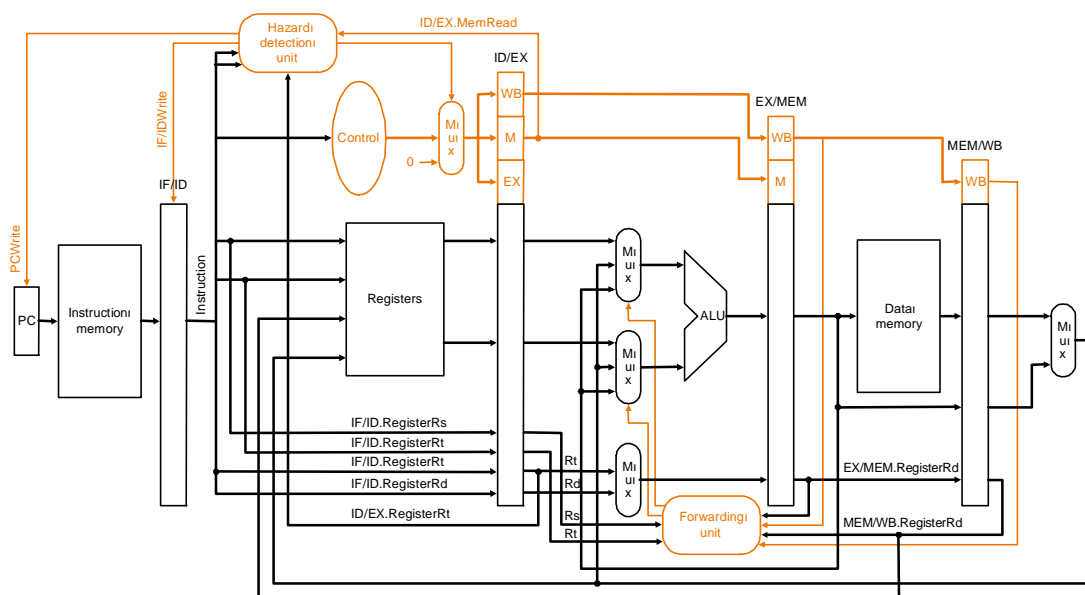
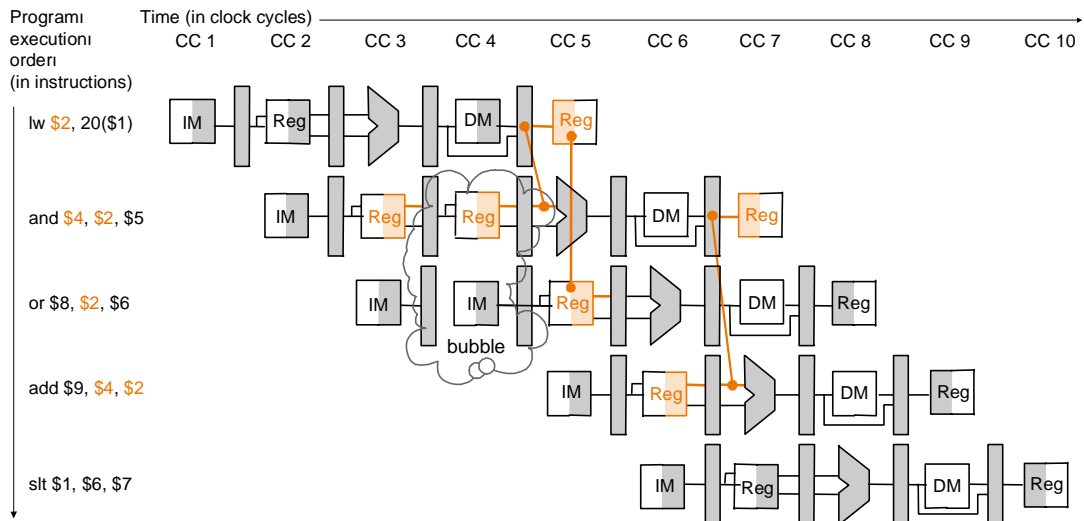
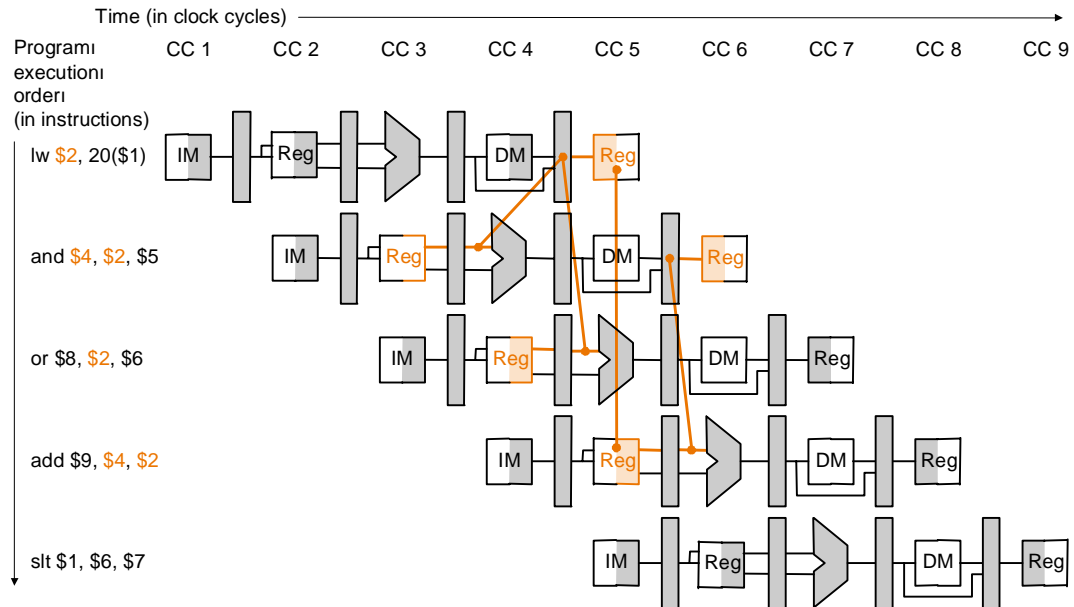
จะสังเกตว่าเราไม่สามารถเรียงคำสั่งในลักษณะที่เป็นการนำคำสั่ง or \$5,\$5,\$1 มาทำงานก่อนหน้าคำสั่ง and \$4,\$2,\$5 ได้ เพราะจะทำให้ค่าเรจิสเตอร์ที่ 5 ที่ใช้ในการประมวลผลผิดพลาดไประหว่างการทำคำสั่ง and \$4,\$2,\$5 ดังเช่น

```
lw    $2,20($1)
or    $5,$5,$1
add    $6,$6,$1
and    $4,$2,$5
xor    $9,$9,$9
```


นอกจากนี้เราจะแก้ปัญหาของ Data Hazard ได้โดยการสร้างฮาร์ดแวร์เพื่อจะได้ไม่ต้องแก้ไขโปรแกรม ซึ่งแต่เดิมเคยทำงานได้ถูกต้องบนหน่วยประมวลผลกลางไม่มี Pipeline โดยอาศัยหลักการของการ forward เช่น กรณีคำสั่ง `sub $2,$1,$3` อยู่ติดกับคำสั่ง `and $12,$2,$5` ซึ่งข้อมูลของ \$2 ที่ถูกต้องนั้น จะถูกนำไปเก็บที่เรจิสเตอร์ใน Cycle ที่ แต่เราสามารถที่จะส่งต่อค่าที่ได้มาไว้ก่อนใน Cycle ที่ 4 ได้ โดยไม่ต้องรอให้ข้อมูลถูกนำมาเก็บในเรจิสเตอร์ที่ 5 เป็นต้น (ดังแสดงด้วยเส้นสีส้ม)



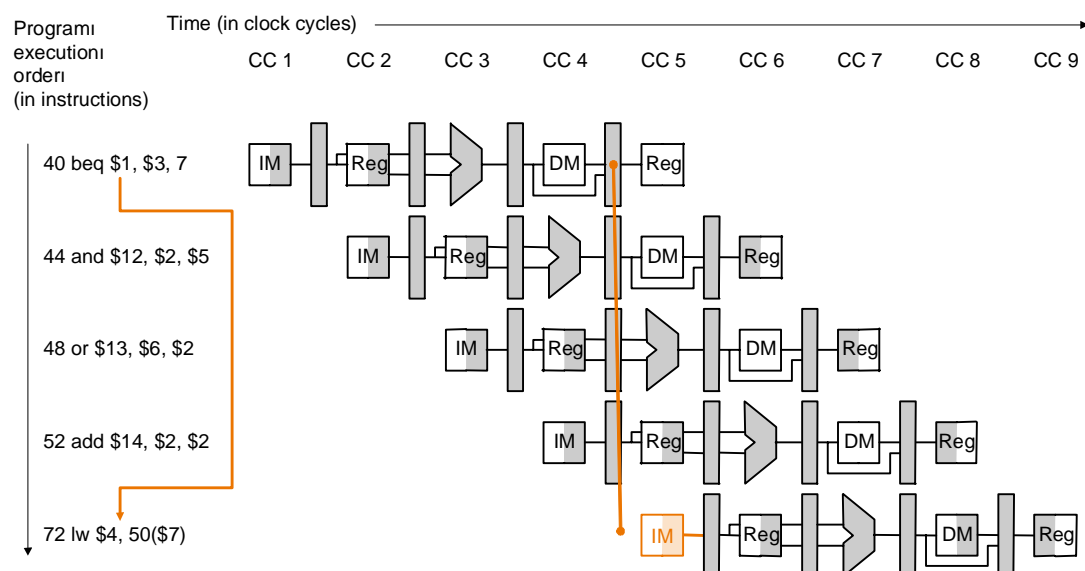
ทั้งนี้พบว่าเราไม่จำเป็นต้องเสียเวลาไปกับการหยุดรอเพื่อนำค่าของเรจิสเตอร์ \$2 ไปใช้ใน Cycle ที่ 5 แต่อย่างไรก็ตามมีหลายๆ กรณีที่เราไม่สามารถ forward ได้เช่นกรณีข้อมูลที่ต้องการใช้ได้มาใน Cycle ที่ 4 แทนที่จะเป็น Cycle ที่ 3 ซึ่งกรณีดังกล่าวจะเกิดขึ้นในเมื่อเป็นการอ่านค่าจากหน่วยความจำเข้ามาเก็บในเรจิสเตอร์เป็นต้น ซึ่งเราต้องแก้ปัญหาดังกล่าวด้วยการหยุดรอเช่นเดิม ทั้งนี้ในกรณีที่การหยุดรอเป็นการหยุดรอที่กระทำโดยฮาร์ดแวร์จะเรียกการหยุดรอในลักษณะนี้ว่า stall ซึ่งการ stall ของฮาร์ดแวร์นั้นคือการทำงานคำสั่งเดิมซ้ำใน Pipeline ถัดมา ด้วยเหตุนี้หน่วยประมวลผลกลางที่แก้ปัญหา Data Hazard ด้วยฮาร์ดแวร์จึงต้องมีวงจรที่คอยตรวจสอบการ forward การ stall และ วงจรที่ทำหน้าที่ forward เสริมขึ้นมา (ดังแสดงได้ดังภาพ)



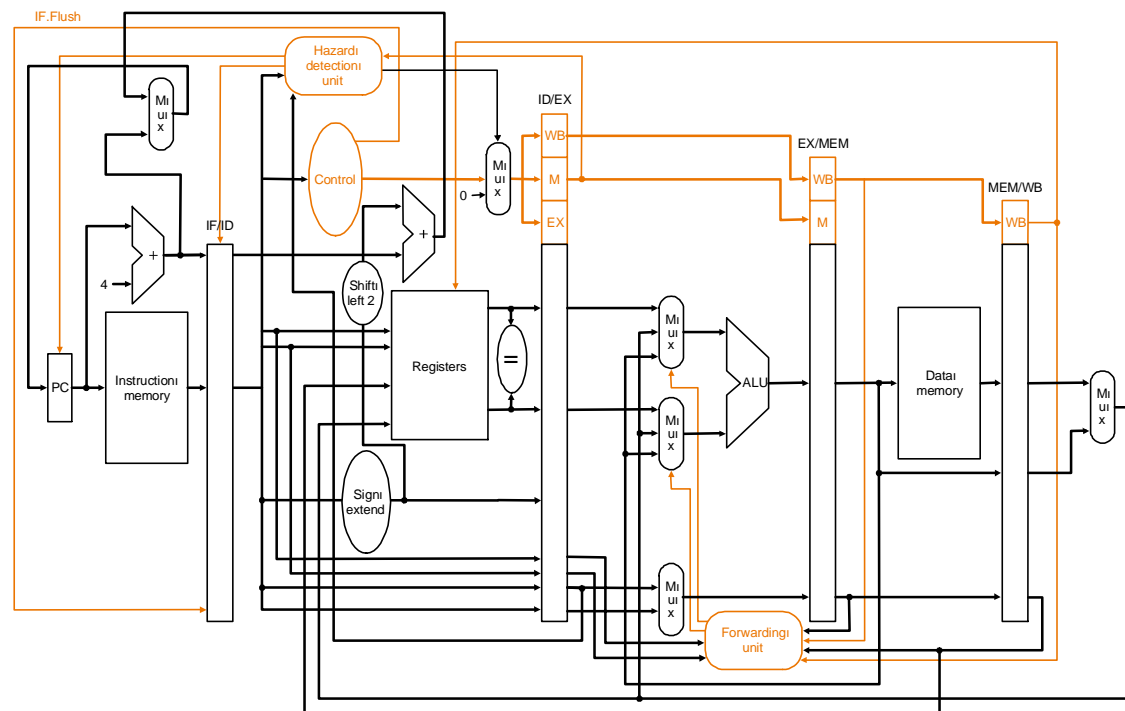
อย่างไรก็ตามเราจะพบว่า แม้จะมีฮาร์ดแวร์สำหรับการ Forward แล้ว แต่การจัดลำดับของรูปแบบคำสั่งที่เหมาะสมก็ยังคงช่วยให้ไม่เกิดการ stall ของคำสั่งภายในระบบได้เช่นกัน ดังนั้นการป้องกันการเกิด Data Hazard ที่ดีที่สุดยังคงเป็นการทำงานของซอฟต์แวร์ หรือ Compiler อยู่นั่นเอง

Control Hazard หรือ Branch Hazard

Control Hazard นั้นเป็นปัญหาในการทำ Pipeline ที่เกิดจากการ Jump หรือ การ Branch ของซอฟต์แวร์ เช่น กรณีการทำคำสั่ง beq จะทราบที่อยู่ของคำสั่งถัดไป (ทราบว่าต้อง branch หรือไม่) เมื่อได้ผ่านการประมวลผลคำสั่งนั้นไปแล้ว 4 Cycle แต่จากโครงสร้างของ Pipeline ซึ่งจะดึงคำสั่งถัดไปขึ้นมาประมวลผลต่อ ทำให้คำสั่งถัดไปที่อยู่ใน Pipeline นั้นเป็นคำสั่งที่ผิดกรณีที่เกิดการ Branch



การแก้ปัญหของ Control Hazard นั้น สามารถกระทำได้หลายแนวทาง เช่น เราอาจเพิ่มวงจรพิเศษเพื่อประมวลผลคำสั่งประเภท Branch ให้เสร็จสิ้นภายใน 1 Cycle เพื่อ Pipeline จะได้คำสั่งถัดไปที่ถูกต้องขึ้นมาทำงานได้ทันที ซึ่งกรณีนี้ทำให้ต้องออกแบบวงจรที่ซับซ้อน ดังนั้นทางเลือกที่ง่ายกว่าคือการล้างคำสั่งที่ไม่ต้องการออกจาก Pipeline ให้หมดแล้วค่อยดึงคำสั่งใหม่ขึ้นมา (Flushing) โดยวิธีนี้สามารถทำได้ง่ายกว่าโดยการเพิ่มเพียงวงจร Flushing เข้าไปเล็กน้อยเท่านั้น ซึ่งเมื่อนำโครงสร้างทั้งหมดของระบบ Pipeline มารวมกัน เราจะได้หน่วยประมวลผลกลางดังภาพ

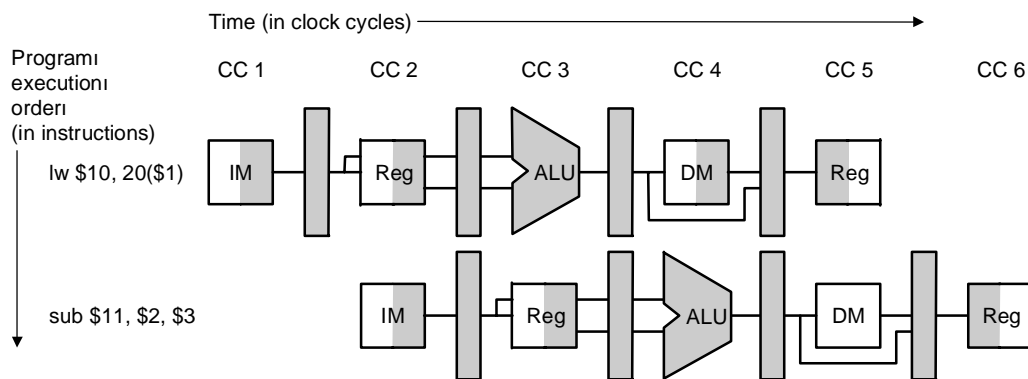


สรุป

การทำ Pipeline เป็นการเพิ่ม Throughput ให้กับหน่วยประมวลผลกลางที่สามารถกระทำได้ง่ายหากหน่วยประมวลผลกลางมีสถาปัตยกรรมชุดคำสั่งที่เหมาะสม โดยการปรับปรุง Data Path และ สัญญาณควบคุมเพียงเล็กน้อยเท่านั้น อย่างไรก็ตามการทำ Pipeline นั้นมีปัญหาที่เป็นอุปสรรคอยู่ 3 ประการคือ Structural Hazard, Data Hazard และ Control Hazard ซึ่งในเอกสารนี้ได้กล่าวถึงแนวทางการแก้ไขแบบพื้นฐานเท่านั้น ในปัจจุบันมีสถาปัตยกรรมที่มีโครงสร้างและการแก้ปัญหา Hazard ที่ซับซ้อนมากกว่านี้ เช่น การทำระบบ Branch Predictive , Branch Delay Slot หรือการทำ Dynamic Schedule เป็นต้น

แบบฝึกหัดท้ายบท

1. การทำ Pipeline ช่วยในการเพิ่ม Throughput ได้ อย่างไร
2. จากภาพแสดงการทำงานของ Pipeline ดังกล่าว



จงแสดงภาพการทำงานของ Pipeline เพื่อประมวลผลคำสั่งต่อไปนี้ พร้อมทั้งแสดงการแรงงาที่ถูกต้อง

```
ADD    $t0,$0,$t1
LW     $t2,20($t5)
ADD    $t3,$t2,$t2
SW     $t3,30($t0)
ADD    $t0,$t0,$t1
```

3. จากภาพที่วาดได้ในข้อ 2 หากหน่วยประมวลผลกลางดังกล่าวมี Hardware สำหรับการ Stall และ Forward จงลากเส้นแสดงตำแหน่งที่ข้อมูลมีการอ้างอิงต่อกัน (Dependency) และ แสดงการเรียงคำสั่งใหม่เพื่อทำการแก้ไขปัญหา Data Hazard ด้วยซอฟต์แวร์
4. เหตุใดการทำงานของหน่วยประมวลผลกลางแบบไม่มี Pipeline จึงไม่เกิดปัญหา Data Hazard จงอธิบาย
5. จงอธิบายปัญหาซึ่งเป็นอุปสรรคต่อการทำ Pipeline และเสนอแนวทางแก้ไข

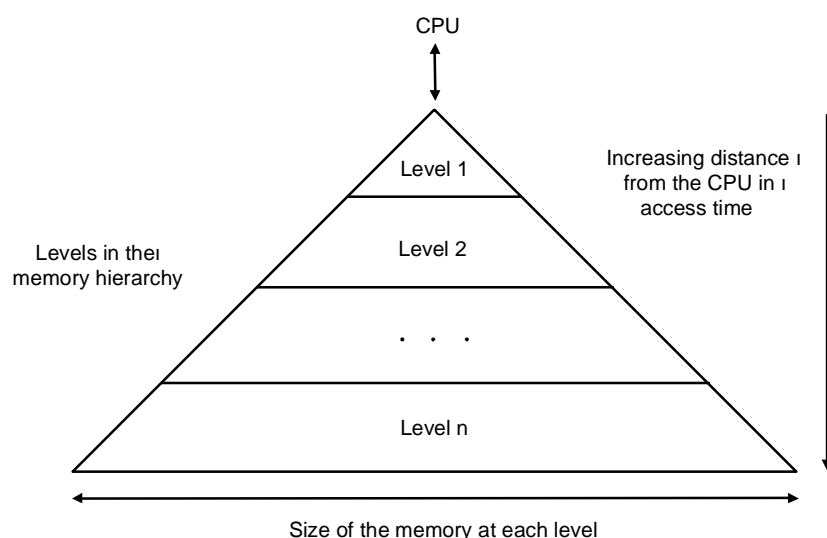
บทที่ 7 ระบบ Cache และ หน่วยความจำเสมือน (Virtual Memory)

ระบบ Cache เป็นการเพิ่มประสิทธิภาพให้กับหน่วยประมวลผลกลาง โดยเพิ่มความเร็วในการอ่านข้อมูลจากหน่วยความจำ ส่วนหน่วยความจำเสมือนนั้นช่วยให้การอ้างถึงหน่วยความจำของผู้พัฒนาซอฟต์แวร์ และ ระบบฮาร์ดแวร์เป็นอิสระจากกัน ซึ่งเนื้อหาในบทนี้จะเน้นที่พื้นฐานการจัดการระบบ Cache และ ระบบหน่วยความจำเสมือนแบบ Paging เท่านั้น อย่างไรก็ตามยังมีระบบหน่วยความจำเสมือนอื่นๆ อีกมากมาย เช่นระบบ Segmentation ซึ่งมีได้กล่าวถึงในที่นี้ แต่เป็นประโยชน์อย่างยิ่งในการใช้พัฒนาซอฟต์แวร์ ดังนั้นผู้อ่านควรจะศึกษาเพิ่มเติมจากเอกสารสามารถของหน่วยประมวลผลกลางต่างๆ เพิ่มเติมด้วย

Cache

Cache นั้นเป็นการเพิ่มประสิทธิภาพในการอ่านและเขียนข้อมูลจากหน่วยประมวลผลกลางให้สามารถกระทำได้รวดเร็วยิ่งขึ้น โดยอาศัยหลักการของ Locality คือการทำให้ข้อมูลที่ต้องการอ่านเป็นประจำนั้นสามารถอ่านได้อย่างรวดเร็ว ซึ่งหลักการของ Locality นั้น เป็นการนำสิ่งที่จำเป็นจะต้องใช้งานบ่อยๆ มาไว้ใกล้ตัว เช่นกรณีที่เราจำเป็นจะต้องใช้ปากกาบ่อยๆ หากปากกาวางอยู่ในที่ซึ่งเราหยิบได้ง่าย การทำงานของเราก็จะสามารถทำได้รวดเร็ว ซึ่งหากปากกาทิ้งไกลตัวจะทำให้การทำงานมีประสิทธิภาพด้อยลง เพราะต้องเสียเวลากับการหยิบปากกานาน ในหน่วยประมวลผลกลางก็เช่นเดียวกัน หากการดึงข้อมูลจากหน่วยความจำ (ไม่ว่าจะเป็นการดึงข้อมูลจากหน่วยความจำโปรแกรมหรือหน่วยความจำข้อมูลก็ตาม) สามารถกระทำได้รวดเร็ว อาจจะช่วยให้อินหน่วยประมวลผลกลางสามารถทำงานได้ที่ Clock Rate ที่สูงขึ้น

อย่างไรก็ตาม เนื่องจากหน่วยความจำที่มีความเร็วสูงนั้นมีราคาแพง ซึ่งหากระบบคอมพิวเตอร์ทั้งหมดทำงานด้วยหน่วยความจำที่มีความเร็วสูงทั้งหมดนั้น จะให้เครื่องคอมพิวเตอร์มีราคาสูงมาก แต่ถ้าหากไม่มีหน่วยความจำความเร็วสูงเหล่านี้เลย ก็จะทำให้การประมวลผลนั้นช้ามากเช่นกัน ด้วยเหตุนี้ระบบคอมพิวเตอร์จึงควรมีหน่วยความจำความเร็วสูงและความเร็วต่ำผสมกันไปเป็นระดับชั้นในปริมาณที่แตกต่างกันไป หน่วยความจำประเภทใดทำงานช้ามักจะราคาถูก เราก็จะได้ในปริมาณที่สูง ในขณะที่หน่วยความจำประเภทที่มีความเร็วสูงราคาแพง ก็จะมีในปริมาณน้อย เช่น Cache อันเป็นหน่วยความจำความเร็วสูงอาจจะมีเพียง 128Kb ส่วนหน่วยความจำหลัก (Ram) อาจจะมีมากถึง 64Mb และ Hard disk ซึ่งราคาต่อหน่วยค่อนข้างต่ำอาจจะมีมากถึง 8 Gb เป็นต้น



อย่างไรก็ตาม การมีหน่วยความจำที่มีความเร็วสูงนั้น อาจจะไม่ทำให้ประสิทธิภาพในการอ่านและเขียนข้อมูลเร็วขึ้นเสมอไป หากเราพิจารณาระบบการจัดการที่เหมาะสม ตัวอย่างเช่น หากเรามีโต๊ะอยู่ 1 ตัว ขนาด 1x1 ตารางเมตร (ซึ่งเปรียบเทียบกับ Cache หรือ Locality) และข้อมูลที่ต้องทำงานด้วยเป็นกล่องขนาด 1x1 ตารางเมตร จำนวนหลายกล่อง (เปรียบเทียบกับหน่วยความจำหลัก) ซึ่งหากการทำงานของเรานั้นต้องเกี่ยวข้องกับกล่องหลายอัน และตามหลักการของ Locality นั้น เราจะต้องทำงานข้อมูลที่เราต้องการใช้งานอยู่ใน Locality ก่อนจึงจะเริ่มอ่านและเขียนข้อมูล หากวิเคราะห์ให้ดีพบว่า เราอาจจะเสียเวลาในการยกกล่องขึ้นมาวางและออกจากโต๊ะมากกว่าการเดินไปทำงานที่กล่องเหล่านั้นโดยตรง (เช่นกรณีการทำงานของเรานั้นต้องนำของจากกล่องที่ 1 มาเก็บในกล่องที่ 2 และนำของจากกล่องที่ 2 มาเก็บในกล่องที่ 1 ตามลำดับ พบว่าเมื่อเราต้องการทำงานกับกล่องที่ 1 นั้น กล่องใบที่ 1 ก็ยังไม่อยู่บนโต๊ะ เราต้องเสียเวลายกกล่องใบที่ 1 ขึ้นมาวางบนโต๊ะก่อน และเมื่อเราทำงานกับกล่องใบที่ 1 เสร็จแล้วต้องการทำงานกับกล่องใบที่ 2 ก็ต้องเสียเวลายกกล่องใบที่ 1 ลง แล้วยกกล่องใบที่ 2 ขึ้นมาวางแทน จนเมื่อทำงานกับกล่องใบที่ 2 เสร็จก็ต้องยกกล่องใบที่ 2 ออก แล้วยกใบที่ 1 ขึ้นมาวางแทน เป็นต้น) ดังนั้นการมี Cache จึงต้องมีขนาดที่เหมาะสม และมีระบบจัดการที่ดีควบคู่กันไปด้วย

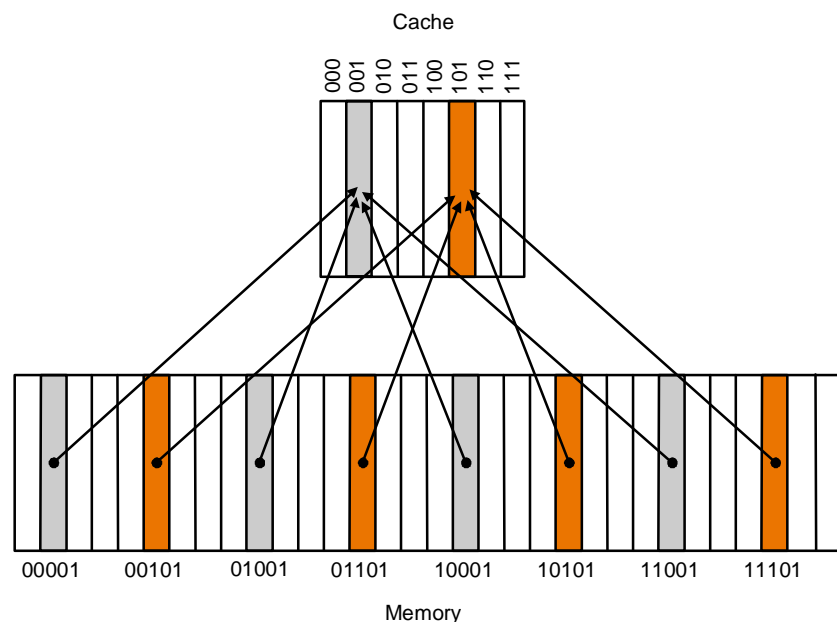
เพื่อความสะดวกในการอธิบายระบบ Cache จึงได้มีการกำหนดคำขึ้นเพื่อใช้อธิบายลักษณะของ Cache และการอ่านเขียนข้อมูล ดังนี้

Block คือ การแบ่งขนาดของหน่วยความจำและ Cache เป็นส่วนย่อยๆ ที่มีขนาดเท่ากัน เพื่อความสะดวกในการจัดการ

Hit หมายถึง ข้อมูลที่เราต้องการใช้งานนั้นอยู่ใน Cache ซึ่งหากเป็นกรณีการอ่านข้อมูลจะเรียกว่า Read Hit และ กรณีการเขียนข้อมูลจะเรียกว่า Write Hit

Miss หมายถึง ข้อมูลที่เราต้องการจะใช้งานนั้นไม่อยู่ใน Cache ซึ่งเราจะเรียกการ Miss ในกรณีการอ่านข้อมูลว่า Read Miss และ เรียกการ Miss ในกรณีการเขียนข้อมูลว่า Write Miss

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 (ดังภาพ)

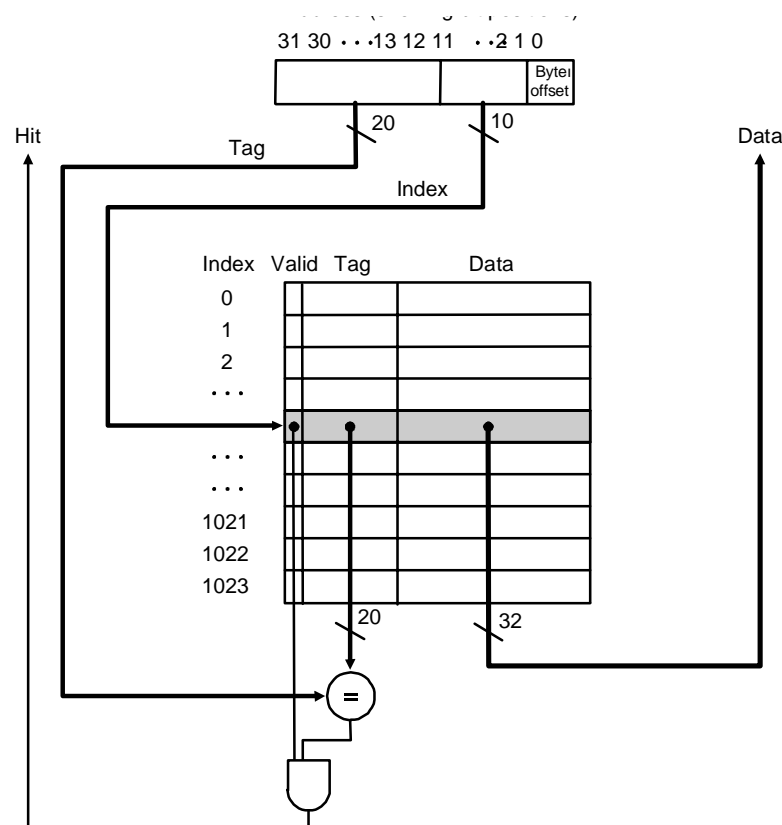


Direct Mapped Cache

การจัดการกับ Cache แบบ Direct Mapped นั้นเป็นระบบการจัดการที่มีพื้นฐานง่ายที่สุด ซึ่งลักษณะโดยสรุปคือ ตำแหน่งในหน่วยความจำทุกตำแหน่งจะมีที่อยู่ใน Cache เพียง 1 ตำแหน่งเท่านั้น หากเราเปรียบ Cache เป็นเหมือนเก้าอี้ในห้องทำงานของเรา ซึ่งมีอยู่ 10 ตัว (ตั้งแต่ตัวที่ 0 – 9) แต่เราต้องทำงานกับคน 1000 คน (โดยแต่ละคนมีเลขประจำตัวของตนเองที่ไม่ซ้ำกัน) เมื่อต้องการทำงานกับคนที่มีเลขประจำตัว 101, 103, 105 ก็จะทำให้แต่ละคนนั้นเก้าอี้ตัวที่ 1, 3 และ 5 ตามลำดับ ซึ่งหากต้องการทำงานกับคนที่มีเลขประจำตัวเป็น 251 ก็จะต้องให้คนที่มีเลขประจำตัว 101 ออกไปก่อน แล้วจึงให้คนที่มีเลขประจำตัว 251 นั่งแทนที่เป็นต้น จะสังเกตว่า

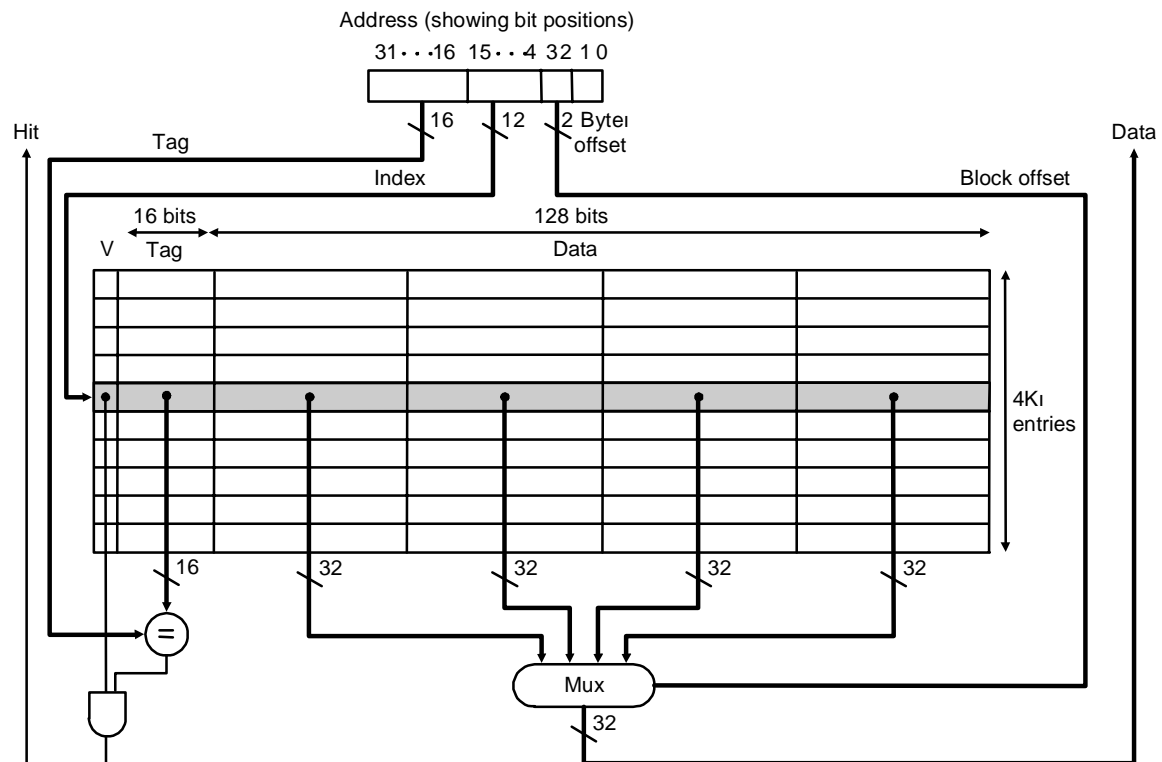
ตำแหน่งใน Cache ของคนต่างๆ นั้น จะเป็นอะไรขึ้นอยู่กับเลขหลักสุดท้ายของเลขประจำตัว ซึ่งการทำงานของ Direct Mapped Cache ก็จะเป็นลักษณะเดียวกัน ในแสดงได้ตามรูปข้างต้น

อย่างไรก็ตามเราจะพบว่า ไม่ว่าจะเป็นคนที่มีเลขประจำตัวเป็น 201, 211, 101, 151 หรืออีกหลายเบอร์ที่ลงท้ายด้วย 1 นั้น ต่างก็ต้องนั้นที่เก้าอี้ตัวที่หนึ่งเหมือนกัน ซึ่งหากเราพบว่ามีคนนั่งอยู่ที่เก้าอี้ตัวที่ 1 แล้ว เราจะทราบได้อย่างไรว่าคนๆ นั้น เป็นใคร(มีเลขประจำตัวอะไร) ด้วยเหตุนี้จึงจำเป็นจะต้องมีการติดป้าย (Tag) เพื่อให้ทราบว่าคนที่นั่งอยู่นั้นมีเลขประจำตัวเป็นอะไร นอกจากนั้นเนื่องจาก Cache ก็เป็นหน่วยความจำชนิดหนึ่งเพื่อแยกแยะให้ทราบว่าข้อมูลในหน่วยความจำนั้น เคยถูกใช้งานมาก่อนหรือไม่จึงต้องมีการตรวจสอบด้วย Valid bit เพื่อแสดงว่าข้อมูลใน Cache นั้นเป็นข้อมูลที่ได้มาจากหน่วยประมวลผลกลาง (เพื่อให้ทราบว่ามีคนนั่งอยู่หรือไม่กรณีที่พบว่าไม่มีป้ายติดอยู่ หรือ กรณีเริ่มต้นระบบ) โครงสร้างที่แท้จริงของระบบ Cache จึงมีลักษณะดังภาพ



จากภาพจะพบว่าระบบ Cache ดังกล่าวมี Block ขนาด 4 Byte และมี Cache ทั้งหมด 1024 Block และการตรวจสอบว่าข้อมูลที่ต้องการอยู่ใน Cache หรือไม่ทำได้โดยการตรวจ Tag ของตำแหน่งข้อมูลที่ต้องการอ่านกับ Tag ของ Cache ในตำแหน่งที่ถูก Mapped ด้วย Index หากตรงกัน และ Valid Bit เป็นหนึ่งแสดงว่าข้อมูลที่ต้องการอยู่ใน Cache (หรือ Hit นั่นเอง)

นอกจากนี้ ยังมีการประยุกต์ให้การจัดการข้อมูลในระบบ Cache มีการแบ่ง Block ที่ละเอียดมากขึ้นโดยอาศัยหลักการ Spatial Locality ทำให้ Cache 1 Record มี หลาย Block ซึ่งชี้โดย Block Index ดังภาพ



Hits และ Miss

การ Hit นั้นหมายถึงกรณีที่ข้อมูลตำแหน่งที่หน่วยประมวลผลต้องการใช้งานอยู่ใน Cache ส่วนการ Miss นั้น ก็จะมี ความหมายตรงข้ามกับ Hit คือ ข้อมูลตำแหน่งที่หน่วยประมวลผลต้องการใช้งานไม่อยู่ใน Cache ซึ่งประสิทธิภาพของ Cache ที่ได้นั้นข้อมูลส่วนใหญ่ควรจะ Hit อยู่ภายใน Cache อย่างไรก็ตามการ Hit และ การ Miss สำหรับการอ่านและการเขียนข้อมูลนั้น ระบบจัดการ Cache วิธีการจัดการในรูปแบบที่แตกต่างกันออกไปดังนี้

Read Hit

หมายถึง การ Hit ข้อมูลใน Cache ในกรณีที่หน่วยประมวลผลต้องการอ่านข้อมูล ซึ่งสิ่งที่หน่วยประมวลผลกลางจะทำก็คือการอ่านข้อมูลจาก Cache ไปใช้ประมวลผล

Read Miss

หมายถึง การ Miss ข้อมูลที่หน่วยประมวลผลต้องการอ่าน (ไม่พบใน Cache) ซึ่งสิ่งที่หน่วยประมวลผลต้องทำคือ ต้องหยุดรอเพื่อให้ระบบจัดการ Cache ทำการดึงข้อมูลจากหน่วยความจำหลักเข้าสู่ Cache ก่อน ซึ่งเวลาหน่วยประมวลผลต้องหยุดรอนี้เรียกว่า

Miss Penalty. จากนั้นเมื่อระบบจัดการ Cache นำข้อมูลจากหน่วยความจำหลักเข้ามาเก็บใน Cache เรียบร้อยแล้ว จึงดำเนินการตามปกติเหมือน Read Hit ต่อไป

Write Hit

หมายถึง การ Hit ข้อมูลที่หน่วยประมวลผลต้องการเขียนใน Cache ซึ่งมีแนวทางในการปฏิบัติได้ 2 ลักษณะ (ทั้งนี้ขึ้นอยู่กับผู้ออกแบบ) คือ Write Back โดยการเขียนข้อมูลลงภายใน Cache เท่านั้น และ การ Write Through คือการเขียนข้อมูลลงภายใน Cache และ หน่วยความจำหลักพร้อมๆ กัน ซึ่งทั้ง 2 แนวทางนี้ก็จะมียกข้อดีข้อเสียต่างกัน กล่าวคือ Write Back นั้นจะทำงานได้เร็วกว่าเพราะจะเขียนข้อมูลที่ต้องการลงใน Cache เท่านั้น แต่มีข้อเสียคือ จะมีช่วงจังหวะหนึ่งที่ข้อมูลใน Cache จะไม่ตรงกับข้อมูลในหน่วยความจำหลัก และเมื่อต้องนำข้อมูลใน Block นั้นออกจาก Cache จึงจะทำการเขียนข้อมูลดังกล่าวคืนไปยังหน่วยความจำหลัก สำหรับกรณีของการ Write Through นั้นหน่วยประมวลผลกลางจะใช้เวลาในการทำงานมากกว่าคือ จะต้องมีการเขียนข้อมูลลงทั้งใน Cache และหน่วยความจำหลักซึ่งจะไม่มีช่วงเวลาที่ข้อมูลในหน่วยความจำหลักมีค่าไม่ตรงกับค่าของข้อมูลใน Cache ด้วยลักษณะดังกล่าว ระบบ Write Back จึงไม่เหมาะที่จะใช้งานบนระบบที่หน่วยประมวลผลกลางหลายตัว

Write Miss

หมายถึง การ Miss ข้อมูลที่หน่วยประมวลผลต้องการเขียนใน Cache ซึ่งหน่วยประมวลผลกลางจะต้องหยุดรอ (Miss Penalty) เพื่อให้ระบบจัดการ Cache นำข้อมูลจากหน่วยความจำหลักเข้ามาเก็บใน Cache ก่อน จากนั้นจึงทำการประมวลผลในลักษณะเดียวกันกับการ Write Hit ข้างต้นต่อไป

ประสิทธิภาพของระบบ Cache

จากลักษณะการทำงานของ Cache ดังกล่าวเราจะพบว่าลักษณะของ Cache ที่มีประสิทธิภาพนั้น คือ Cache ที่มีระบบการจัดการซึ่งทำให้ Miss Ratio มีค่าน้อย และ เวลาที่ต้องหยุดเพื่อให้ระบบจัดการ Cache นำข้อมูลจากหน่วยความจำหลักเข้ามาสู่ Cache (Miss Penalty) มีค่าน้อยที่สุด ซึ่งจากลักษณะดังกล่าวทำให้เวลาที่ใช้ในการประมวลผลของหน่วยประมวลผลกลางนั้นมีความสัมพันธ์กับเวลาที่ต้องหยุดรอเพื่อให้ระบบจัดการ Cache ทำงาน ดังแสดงได้ดังสมการต่อไปนี้

$$\text{execution time} = (\text{execution cycles} + \text{stall cycles}) * \text{cycle time}$$

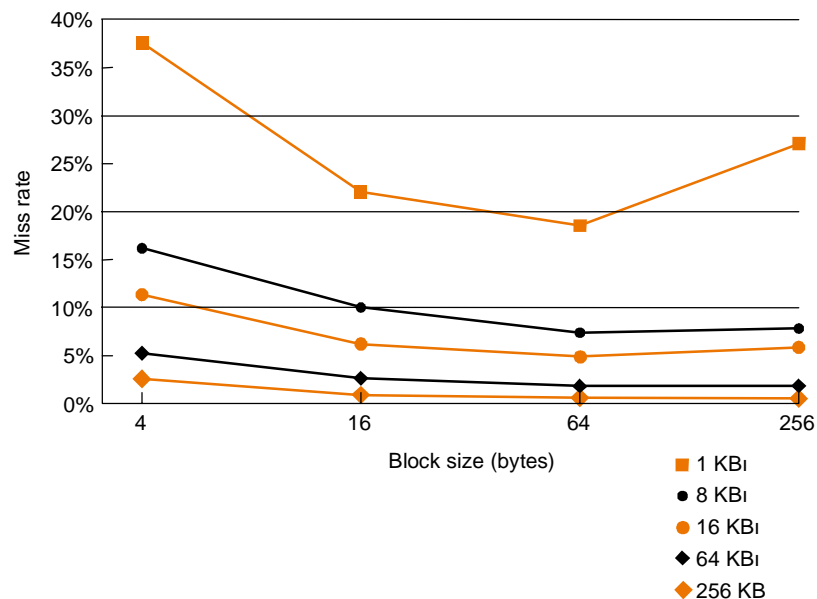
$$\text{stall cycles} = \# \text{ of instructions} * \text{miss ratio} * \text{miss penalty}$$

ทั้งนี้ให้ลองเปรียบเทียบสมการดังกล่าวกับสมการที่พบในบทที่ 2 เมื่อมองแบบผิวเผินจะพบว่า Execution Time ในกรณีที่มี Cache นั้นจะมากกว่า Execution Time ในกรณีที่ไม่มี Cache เพราะจะมี Stall Cycles เพิ่มขึ้นมา ซึ่งในความเป็นจริงนั้นการมี Cache ช่วยทำให้ Cycle Time มีค่าน้อยลง (หน่วยประมวลผลกลางสามารถทำงานได้ที่ Clock Rate ที่สูงขึ้น เพราะเวลาที่ใช้ในการรอข้อมูลจากหน่วยความจำน้อยลง)

จากสมการข้างต้นพบว่า การปรับปรุงประสิทธิภาพของระบบ Cache นั้นสามารถทำได้โดยการลด Stall Cycles ของ Cache ให้น้อยลงซึ่งการลด Stall Cycles นั้น สามารถทำได้โดยการลด Miss Ratio และ ลด Miss Penalty ตามแนวทางต่อไปนี้

การลด Miss Ratio

การลด Miss Ratio นั้น คือการทำให้ข้อมูลที่หน่วยประมวลผลต้องการใช้ มีโอกาสที่จะถูกพบใน Cache มากที่สุด ทั้งนี้ปัจจัยหนึ่งคือการกำหนดให้ Cache มีขนาดของ Block Size ที่เหมาะสม ซึ่งจากการทดสอบพบว่า จะมีค่าที่เหมาะสมอยู่ค่าหนึ่งๆ สำหรับหน่วยประมวลผลกลางแต่ละตัว ทั้งนี้หาก Block Size มีขนาดใหญ่เกินไป ในบางกรณี Cache อาจจะมี Miss Ratio มากขึ้นก็ได้ (ดังแสดงในตาราง) สำหรับกรณีของหน่วยประมวลผลกลาง MIPS นี้ ค่าที่เหมาะสมคือ 64KB



Program	Block size in words	Instruction miss rate	Data miss rate	Effective combined miss rate
gcc	1	6.1%	2.1%	5.4%
	4	2.0%	1.7%	1.9%
spice	1	1.2%	1.3%	1.2%
	4	0.3%	0.6%	0.4%

ดังนั้นการลดค่าของ Miss Ratio ที่ดีที่สุดคือการเลือกใช้ระบบจัดการ Cache ที่เหมาะสม เบื้องต้นนั้นเราได้ศึกษาถึงระบบการจัดการ Cache แบบ Direct Mapped แล้ว ซึ่งการจัดการ Cache แบบ Direct Mapped นั้นอาจเรียกได้ว่าเป็นการจัดการแบบ One-way set associative หรือการกำหนดให้ 1 ตำแหน่งของข้อมูลในหน่วยความจำหลักนั้นมีที่อยู่ใน Cache ได้เพียง 1 ที่ ซึ่งจากการทดสอบพบว่ามีโอกาสที่จะเกิด Miss Ratio ที่สูงกว่าการกำหนดให้ 1 ตำแหน่งของหน่วยความจำหลักมีที่อยู่ใน Cache มากกว่า 1 ที่ เช่น Two-way set associative, Four-way set associative, หรือ ระบบ Eight-way set associative (Fully Associative)

One-way set associative
(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

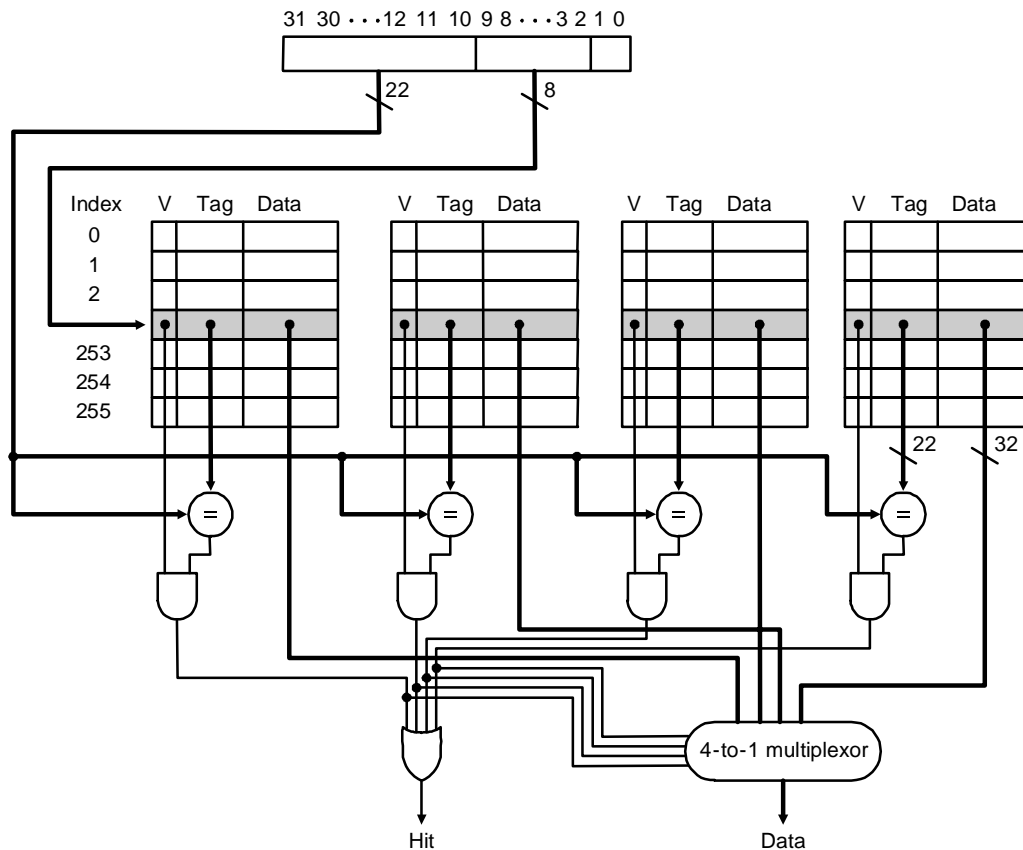
Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

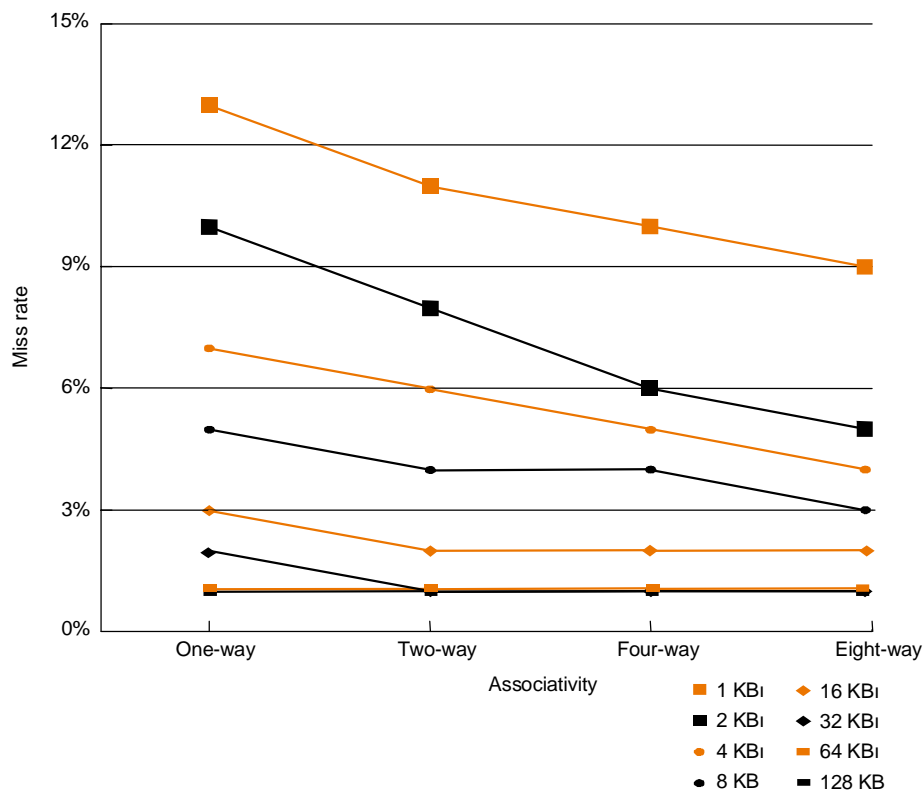
Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

อย่างไรก็ตาม แม้ว่าระบบการจัดการ แบบมี Associative way มากกว่า 1 ทาง จะช่วยลด Miss Ratio ได้ดี แต่การจัดการระบบ Cache เพื่อตรวจสอบการ Hit และ Miss ก็ยุ่งยากขึ้นไปด้วยดังภาพจะแสดงการทำงานของระบบ Cache ในแบบ Four-way set associative ซึ่งจะมีลักษณะคล้ายกับ Direct Mapped 4 อันต่อขนานกัน การตรวจสอบการ Hit นั้นต้องเปรียบเทียบ Tag กับทั้ง 4 Block และ เมื่อมีการ Hit แล้ว ก็จะต้องผ่าน Multiplex เพื่อเลือกข้อมูลที่ต้องการออกมาใช้ ในกรณีที่ต้องการนำข้อมูลออกจาก Cache นั้น ผู้ออกแบบมักนิยมใช้ Least Recent Use (LRU) Algorithm คือเลือกข้อมูลในตำแหน่งที่ถูกใช้งานน้อยที่สุดออกจาก Cache



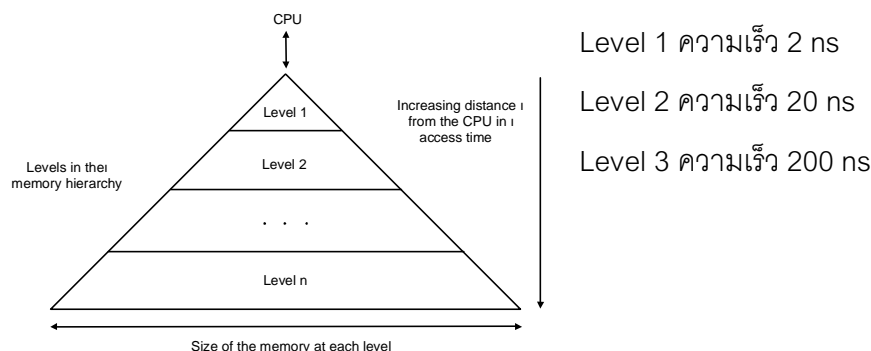
จากการทดสอบพบว่า การเพิ่ม Associative Way ช่วยให้ Miss Rate มีค่าลดลง
ในลักษณะเกือบจะเป็นเชิงเส้น (ดังแสดงได้ดังภาพ)



การลด Miss Penalty

การลด Miss Penalty นั้นคือการทำให้การโอนถ่ายข้อมูลระหว่าง Cache และ หน่วยความจำหลักสามารถกระทำได้อย่างรวดเร็ว ซึ่งหลักการที่ดีที่สุดในการลด Miss Penalty คือการสร้าง Cache เพื่อรองรับการทำงานหลายๆ ระดับ (Multi-Level Cache) โดยเมื่อ หน่วยประมวลผลกลางต้องการหาข้อมูลนั้น หาก Miss ในระดับแรกอาจจะต้องเสีย Miss Penalty ถึง 10 Cycle เพื่อดึงข้อมูลจากหน่วยความจำหลัก แต่หากมีระบบ Cache หลายระดับ เมื่อไม่พบข้อมูลในระดับแรก (Miss) อาจพบข้อมูลใน Cache ระดับถัดไปได้ ทำให้ระบบไม่จำเป็นต้องเสีย Miss Penalty ถึง 10 Cycle ทุกครั้งที่มีการ Miss เกิดขึ้น หากวิเคราะห์ดีจะพบว่า การลด Miss Penalty โดยการทำ Multi-Level Cache นั้น คือการพยายามทำให้ข้อมูลบางส่วนสามารถหยิบใช้งานได้เร็วขึ้น มีได้เป็นการลดค่าตัวเลขโดยตรง

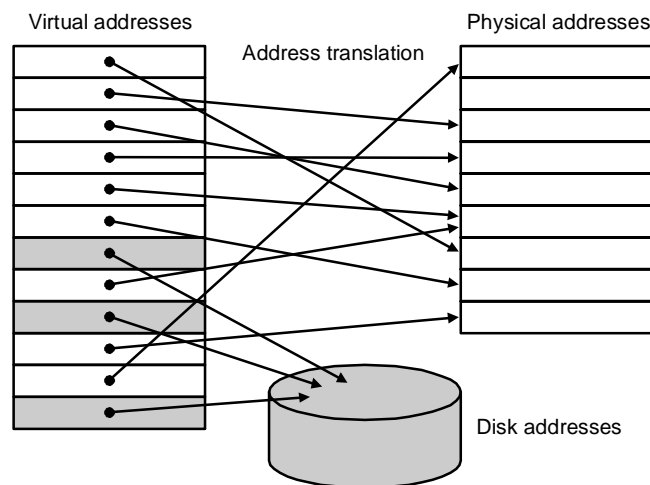
จากการทดสอบพบว่า หากเครื่องคอมพิวเตอร์ระบบหนึ่งซึ่งมี Cache ทำงานที่ความเร็ว 2 ns และ หน่วยความจำหลักทำงานที่ 200 ns แล้ว การเพิ่ม Cache อีกระดับหนึ่งซึ่งมีความเร็ว 20 ns แทนเข้าไป จะช่วยลดค่า Miss Rate จาก 5% เป็น 2% ได้



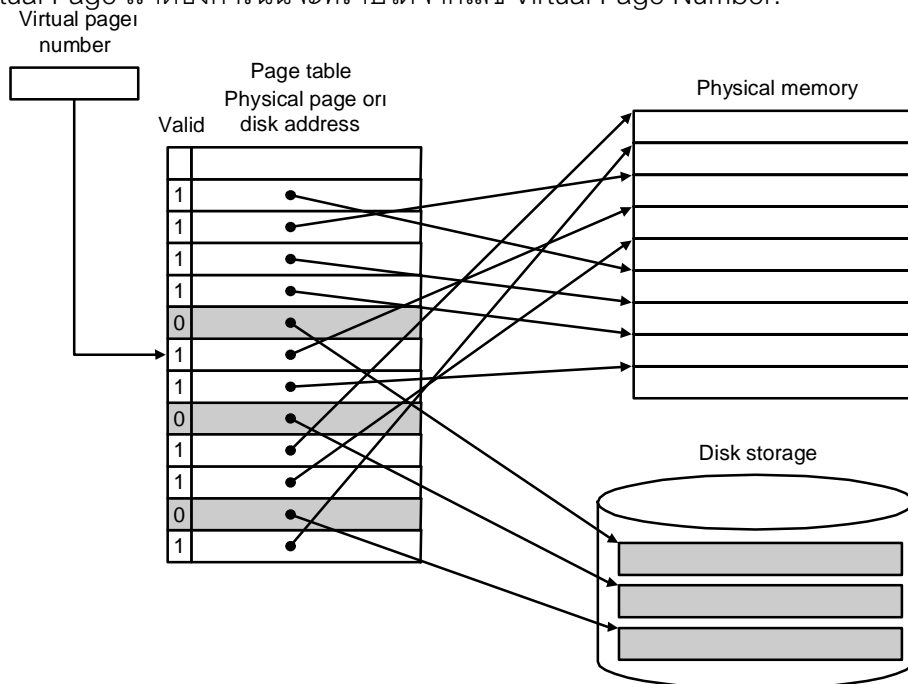
หน่วยความจำเสมือน (Virtual Memory)

หน่วยความจำเสมือนนั้น เป็นหน่วยความจำที่ไม่อยู่จริงในระบบเครื่องคอมพิวเตอร์ หากแต่เป็นเพียงหน่วยความจำที่ใช้อ้างอิงโดยซอฟต์แวร์ทั้งไป ตัวอย่างของระบบหน่วยความจำเสมือนที่พบได้ง่ายที่สุดคือ การที่โปรแกรมซึ่งต้องการใช้หน่วยความจำเป็นจำนวนมาก สามารถทำงานได้บนเครื่องที่มีหน่วยความจำต่ำๆ ได้ (เช่น โปรแกรมที่ต้องการหน่วยความจำ 40 Mb ระหว่างการประมวลผล อาจทำงานได้บนเครื่องที่มีหน่วยความจำหลักเพียง 16 Mb เป็นต้น) โดยการทำงานดังกล่าวนี้ สามารถกระทำได้นี้เนื่องจากหน่วยความจำหลักทำหน้าที่คล้ายกับ Cache ของหน่วยความจำทั้งหมดที่โปรแกรมต้องการ และข้อมูลทั้งหมดของโปรแกรมจะถูกเก็บไว้ในหน่วยความจำสำรองขนาดใหญ่ ซึ่งทำหน้าที่เป็น Swap Space (โดยทั่วไปมักเป็นฮาร์ดดิสก์) ดังแสดงได้ดังภาพ ด้านซ้ายคือ Virtual Memory ด้านขวาคือ Physical Memory และ มี Disk Address

เพื่อทำหน้าที่เป็น Swap Space (จะสังเกตได้ว่า Virtual Memory จะมีขนาดใหญ่กว่า Physical Memory ซึ่งบางส่วนจะเก็บอยู่ใน Physical Memory และ บางส่วนจะเก็บอยู่ใน Disk)

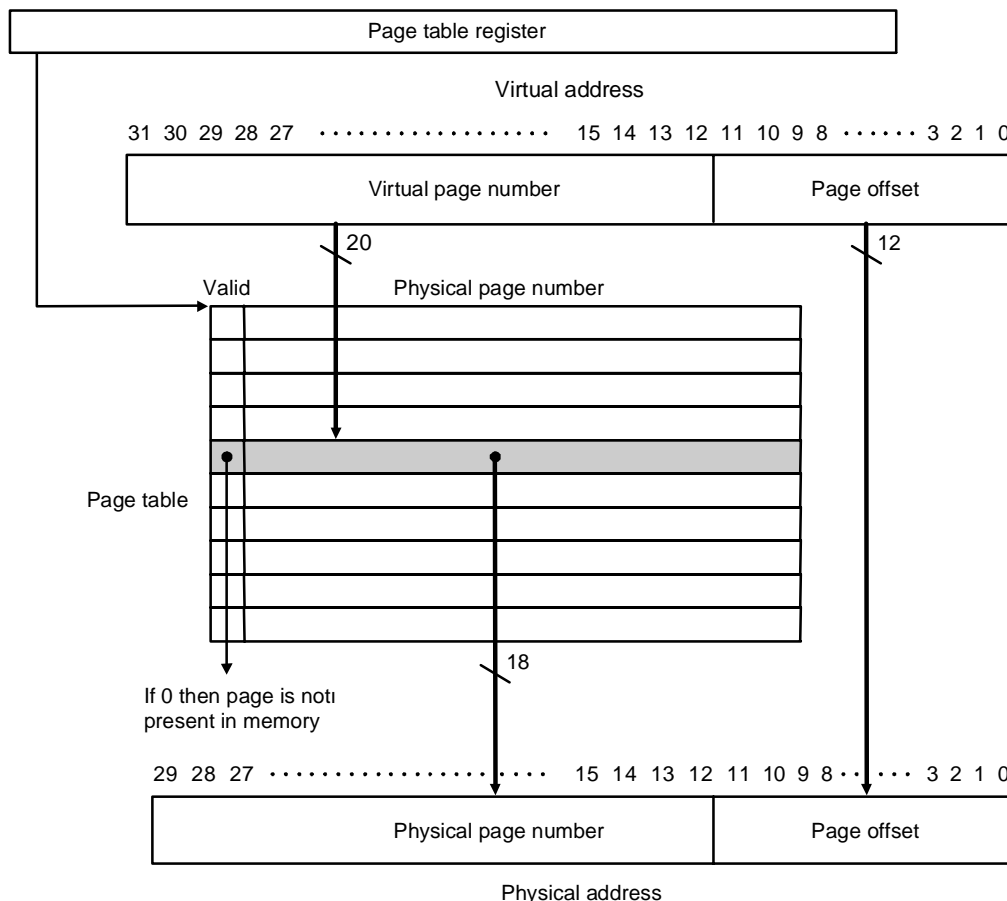


ในการหาข้อมูลของระบบคอมพิวเตอร์ที่มีหน่วยความจำเสมือนนั้น เราจะนำเลข Virtual Page Number มาค้นหาในตาราง Page Table ซึ่งเป็นตารางที่ใช้ในการแปลง Virtual Address ให้เป็น Physical Address โดยตารางนี้จะมีจำนวน Record(s) เท่ากับจำนวน Virtual Page และแต่ละ Record จะจัดเก็บ ตำแหน่งที่ชี้ไปยัง Physical Memory โดยที่ Physical Memory และมีอีก 1 Bit (Valid) ที่จะใช้บอกว่าข้อมูลที่ต้องการนั้น เก็บอยู่ในหน่วยความจำ Physical Memory หรือฝากเป็น Swap อยู่ใน Disk Storage. ซึ่งตำแหน่งของ Record ที่เก็บ Physical Page Number ของ Virtual Page เราต้องการนั้นจะทราบได้จากเลข Virtual Page Number.



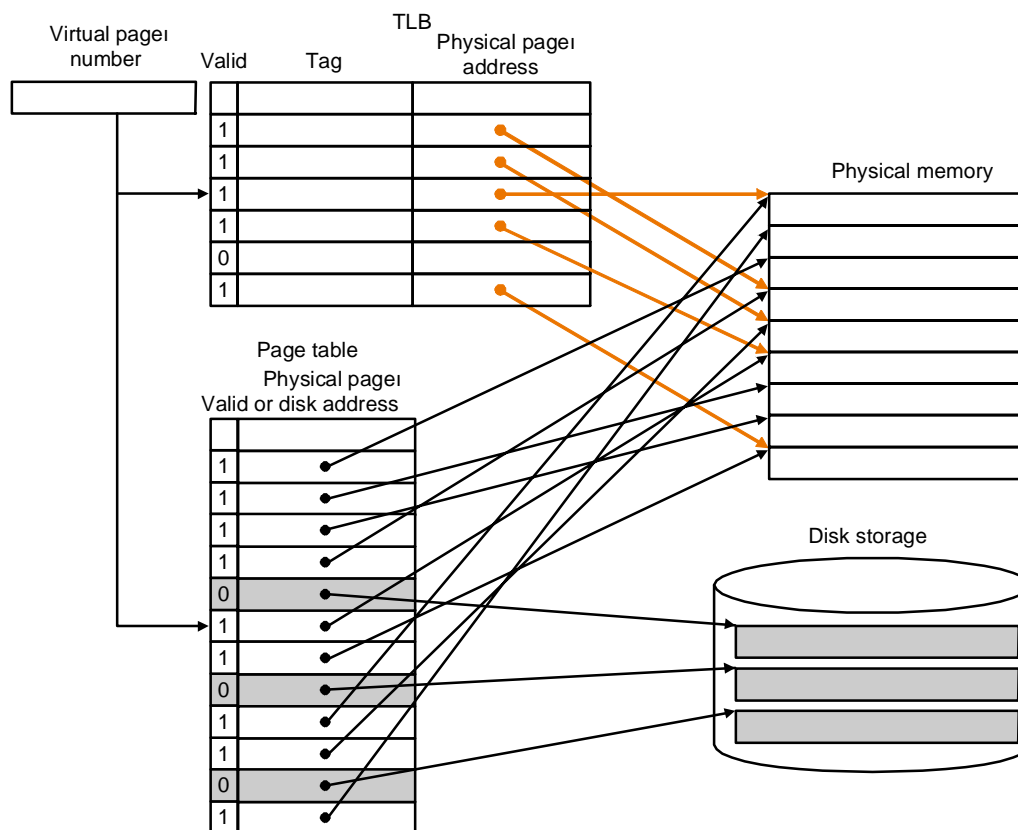
โดยขั้นตอนในการนำข้อมูลเข้าและออกระหว่างหน่วยความจำ Physical Memory และ Disk Storage นั้น จะถูกจัดการโดยซอฟต์แวร์ระบบ หรือ ระบบปฏิบัติการทั้งนี้เนื่องจากความ

ต้องการในการใช้หน่วยความจำของแต่ละระบบมีความแตกต่างกัน หน่วยประมวลผลกลางจึงมีเพียงเรจิสเตอร์พิเศษ ที่จะชี้ไปยังหน่วยความจำตำแหน่งที่เก็บตาราง Page Table และวงจรซึ่งจะทำหน้าที่ตรวจสอบว่าข้อมูลที่ต้องการอยู่ในหน่วยความจำหลัก (Physical Memory) หรือไม่ และกรณีที่เกิด Page Fault (ข้อมูลที่ต้องการไม่ได้อยู่ในหน่วยความจำหลัก) หน่วยประมวลผลกลางจะสร้างสัญญาณ Interrupt เพื่อให้ซอฟต์แวร์ทำงานต่อไป (รายละเอียดเพิ่มเติมสามารถศึกษาได้จากวิชาระบบปฏิบัติการ Operating System)

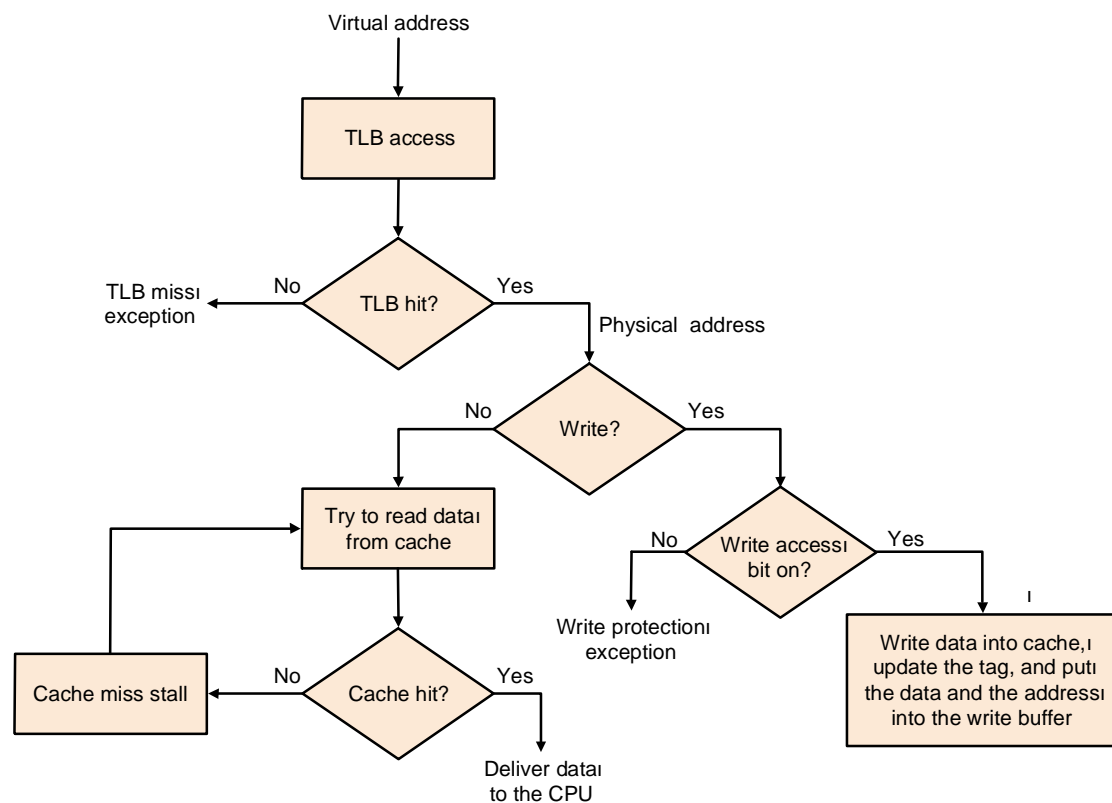


หากโครงสร้างของระบบ Virtual Address และ Physical เป็นดังภาพข้างต้น เราจะพบว่า Page Table นั้นจะมีจำนวน Record(s) ทั้งหมด $2^{(31-11)}$ Records (เนื่องจาก Virtual Page Number บอกด้วย Bit ที่ 12-31 ของ Virtual Address) และ แต่ละ Record นั้นจะประกอบด้วย $(29-11)+1$ Bit หรือ ประมาณ 3 Byte (เนื่องจาก Physical Page Number บอกด้วย Bit ที่ 12-29 ของ Physical Address และแต่ละ Record จะต้องเก็บ Valid Bit ซึ่งใช้บอกว่าข้อมูลอยู่ในหน่วยความจำหลัก หรือ อยู่ใน Swap Space ซึ่งเมื่อแปลงข้อมูลเป็นหน่วย Byte จะต้องทำการปัดเศษขึ้นเพื่อให้เป็นจำนวนเต็มเท่านั้น) ดังนั้นในกรณีดังกล่าวนี้ Page Table จะมีขนาดใหญ่ถึง 3×2^{20} Byte หรือประมาณ 3 Mb และเนื่องจาก Page Table นี้ก็ถูกเก็บไว้ในหน่วยความจำหลักเช่นกัน ดังนั้นเพื่อให้การแปลงเลข Virtual Page Number ไปเป็น Physical Page Number สามารถกระทำได้

รวดเร็วยิ่งขึ้น จึงได้มีการประยุกต์ใช้ระบบ Cache กับ Page Table เช่นกัน (TLB Cache) ดังแสดงได้ดังภาพ



เมื่อระบบ Virtual Memory และระบบ Cache นั้นทำงานพร้อมกันทั้งหมดภายในหน่วยประมวลผลกลาง ก็จะทำให้ขั้นตอนในการประมวลผลข้อมูลจากของหน่วยประมวลผลกลางนั้นซับซ้อนยิ่งขึ้น โดยเริ่มตั้งแต่หน่วยประมวลผลกลางได้รับ Virtual Address จากซอฟต์แวร์ ก็จะทำการแปลงข้อมูลให้เป็น Physical Address โดยค้นหาจาก Page Table ซึ่งการค้นหาจะเริ่มต้นจาก Cache ของ Translation Table ก่อน หากไม่พบก็จะทำการดึงข้อมูลจาก Page Table ตำแหน่งที่ต้องการเข้ามาไว้ใน Cache ของ Translation Table เมื่อได้ Physical Address แล้ว หากข้อมูลที่ต้องการไม่อยู่ใน Physical Memory ก็จะสร้างสัญญาณ Page Fault เพื่อให้ซอฟต์แวร์ระบบปฏิบัติการจัดการดึงข้อมูลเข้ามาไว้ใน Physical Memory ต่อไป เมื่อได้ Physical Address ซึ่งอยู่ใน Physical Memory แล้ว หน่วยประมวลผลกลางจะค้นหาข้อมูลตำแหน่งดังกล่าวใน Cache ของข้อมูล และทำตามขั้นตอนของการ Write Hit หรือ Read Hit ต่อไป ทั้งนี้ขึ้นอยู่กับว่าคำสั่งที่ประมวลผลนั้นเป็นการอ่านหรือเขียนข้อมูล ส่วนกรณีการ Miss นั้น ระบบจัดการ Cache ข้อมูลจะทำการ Stall CPU แล้วดึงข้อมูลจากหน่วยความจำหลักเข้ามาเก็บใน Cache เพื่อประมวลผลตามขั้นตอนต่อไป ลำดับในการทำงานดังกล่าวนี้ อาจแสดงได้ดังแผนภาพต่อไปนี้



สรุป

ระบบ Virtual Memory และ ระบบ Cache เป็นระบบที่ช่วยเสริมประสิทธิภาพของหน่วยประมวลผลกลางในการทำงานที่เกี่ยวข้องกับหน่วยความจำ โดยระบบ Cache จะช่วยอำนวยความสะดวกให้หน่วยประมวลผลกลางสามารถอ่านข้อมูลจากหน่วยความจำหลักได้เร็วขึ้น ซึ่งผลที่ได้คือหน่วยประมวลผลกลางจะสามารถทำงานได้ที่ Clock Rate ที่สูงขึ้น เพราะไม่ต้องรอการทำงาน of หน่วยความจำ อย่างไรก็ตามประสิทธิภาพของระบบนั้นจะดีหรือไม่ขึ้นอยู่กับค่า Miss Ratio กับ ค่า Miss Penalty ด้วย สำหรับระบบ Virtual นั้นช่วยให้โปรแกรมเมอร์สามารถพัฒนาซอฟต์แวร์ได้เป็นอิสระโดยไม่ต้องสนใจระบบหน่วยความจำของเครื่องนั้นๆ และยังให้ซอฟต์แวร์ที่ต้องการหน่วยความสูง สามารถทำงานบนเครื่องที่มีหน่วยความจำหลักน้อยๆ ได้

แบบฝึกหัดท้ายบท

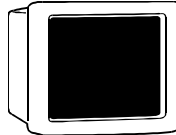
1. หลักการของ Locality ช่วยให้หน่วยประมวลผลกลางสามารถทำงานเร็วขึ้นได้อย่างไร
2. หากท่านเป็นผู้ออกแบบ Cache ท่านจะมีหลักเกณฑ์อย่างไรในการเลือกใช้ระบบ Write Back และ Write Through.
3. มีปัจจัยใดบ้างที่มีผลต่อประสิทธิภาพของ Cache และเราสามารถปรับปรุงประสิทธิภาพของ Cache ได้อย่างไรบ้างจงอธิบาย
4. โปรแกรมทดสอบอันหนึ่งมีชุดคำสั่งทั้งสิ้น 2 ล้านคำสั่ง หาก Miss Rate เป็น 10 % และ Miss Penalty เป็น 6 Cycle หากหน่วยประมวลผลกลางดังกล่าวมี CPI เป็น 1 และมี Clock Rate เป็น 200 Mhz จงแสดงการคำนวณหาเวลาในการประมวลผลโปรแกรมทดสอบของหน่วยประมวลผลกลางดังกล่าว
5. การมีระบบ Virtual Memory นั้นเป็นประโยชน์สำหรับผู้พัฒนาซอฟต์แวร์หรือไม่ อย่างไร

บทที่ 8 การเชื่อมต่ออุปกรณ์ต่างๆ เข้ากับหน่วยประมวลผลกลาง

ในการเชื่อมต่ออุปกรณ์ต่างๆ เข้ากับหน่วยประมวลผลกลางนั้น มีหลากหลายแนวทางที่สามารถทำได้ ทั้งนี้ปัจจัยที่จะใช้เป็นตัวกำหนดว่าการเชื่อมต่อจะเลือกใช้มาตรฐานใด คือ ความเร็วของอุปกรณ์ ลักษณะของหน่วยประมวลผลกลาง และ สภาพแวดล้อมของระบบที่อุปกรณ์เหล่านั้นต่อพ่วงอยู่ เนื้อหาในบทนี้จะกล่าวเพียงแนวทางในการเชื่อมต่ออุปกรณ์ต่างๆ ทางด้านฮาร์ดแวร์และซอฟต์แวร์ (หรือที่เรามักจะคุ้นเคยในลักษณะของ Device Driver)

การเชื่อมต่ออุปกรณ์ทางด้านฮาร์ดแวร์

ในการเชื่อมต่ออุปกรณ์เข้ากับหน่วยประมวลผลกลางนั้น หน่วยประมวลผลกลางบางตระกูลจะต่ออุปกรณ์เหล่านั้นเข้ากับ Memory Bus แล้วมองเสมือนหนึ่งว่าการทำงานกับอุปกรณ์เหล่านั้นเป็นการทำงานกับหน่วยความจำตามปกติ โดยระบบฮาร์ดแวร์จะทำการ Latch ค่าเขียนออกไป และ อ่านค่าอุปกรณ์จากหน่วยความจำตำแหน่งที่กำหนดไปยังค่าอุปกรณ์เหล่านั้น ซึ่งหลักการนี้เราเรียกว่า Memory Mapped I/O ดังแสดงได้ดังภาพ

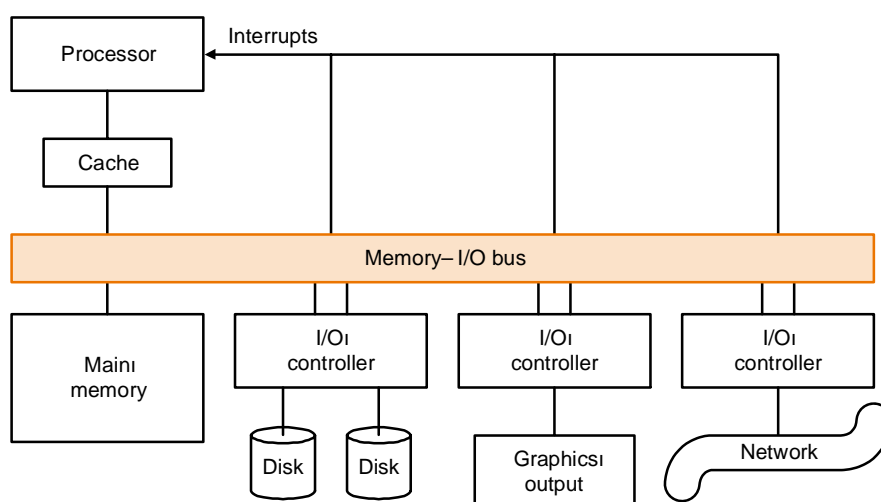
ระบบซอฟต์แวร์	ระบบฮาร์ดแวร์
SW “A”, [B000h]	 ปรากฏตัวอักษร “A” ที่จอภาพ

ซึ่งระบบ Memory Mapped I/O นี้เป็นที่นิยมในระบบคอมพิวเตอร์ยุคแรกๆ แต่เนื่องจากระบบหน่วยความจำนั้นมีลักษณะแตกต่างจากระบบ I/O ดังนั้น จึงได้มีการแยกการจัดการระบบหน่วยความจำออกจากระบบการต่อพ่วงอุปกรณ์ต่างๆ (เนื่องจากหน่วยความจำนั้นมีความเหมาะสมที่จะทำเป็นระบบ Cache ในขณะที่ระบบ I/O นั้นไม่ต้องการ Cache) ดังนั้นจึงเกิดเป็นแนวทางในการเชื่อมต่อหน่วยประมวลผลกลางผ่านระบบ I/O Bus System และ เพื่อกันความสับสน จึงเรียก Address ที่อุปกรณ์ต่อพ่วงอยู่ว่า Port แทน

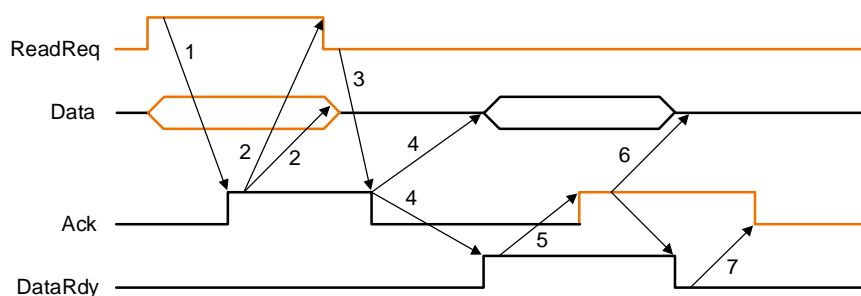
อย่างไรก็ตาม ในบางกรณีนั้นอุปกรณ์ต่อพ่วงอาจจะทำงานเร็วมาก ในขณะที่หน่วยประมวลผลกลางอาจจะทำงาน ซึ่งการโอนถ่ายข้อมูลผ่านระบบ I/O โดยอาศัยหน่วยประมวลผลกลาง อาจจะทำให้ล่าช้า ด้วยเหตุนี้จึงส่งผลให้เกิดระบบ Direct Memory Access (DMA) เพื่อใช้ในการโอนถ่ายข้อมูลระหว่างหน่วยความจำหลักและอุปกรณ์ต่อพ่วงโดยตรง ไม่ต้องผ่านหน่วยประมวลผลกลาง (อุปกรณ์ซึ่งทำงานที่ความเร็วสูงกว่าหน่วยประมวลผลกลางมากๆ นิยมเชื่อมต่อ

ผ่านระบบนี้) อย่างไรก็ตามการเชื่อมต่อในลักษณะนี้จะต้องมีการทำงานของ DMA Controller Unit เข้ามาช่วยในการประมวลผลด้วย

ในกรณีที่อุปกรณ์ต่อพ่วงอยู่บนระบบสายสัญญาณที่มีมาตรฐานแตกต่างจาก ระบบ Bus ของหน่วยประมวลผลกลางนั้น การเชื่อมต่อระหว่างอุปกรณ์ต่อพ่วงและหน่วยประมวลผลกลางจะมีหน่วยประมวลผลพิเศษเป็นตัวประสานซึ่งเรามักจะเรียกหน่วยประมวลผลพิเศษเหล่านี้ว่า Bridge หรือ Controller เช่น อุปกรณ์ต่อพ่วงที่อยู่บนมาตรฐานของระบบ SCSI นั้น ไม่สามารถที่จะต่อพ่วงเข้ากับหน่วยประมวลผลกลางได้โดยตรง การต่อพ่วงจะต้องกระทำผ่าน SCSI Controller เป็นต้น ด้วยเหตุนี้การต่อเชื่อมอุปกรณ์โดยทั่วไป จึงมักกระทำผ่าน Controller ก่อนเสมอ (ดังแสดงได้ดังภาพ)



จากภาพจะพบว่า I/O Controller บางประเภทนั้นจะพ่วงต่อกับอุปกรณ์ได้มากกว่า 1 ชิ้น และหลายๆ กรณีอุปกรณ์เหล่านั้นก็มีความแตกต่างกันมากในแง่ของความเร็ว ซึ่งการสื่อสารระหว่างอุปกรณ์และ Controller ที่มีความเร็วแตกต่างกันนี้หลายกรณีที่อุปกรณ์และ Controller มิได้ทำงานที่สัญญาณนาฬิกาเดียวกัน (Asynchronous) ดังนั้นจึงต้องมีการกำหนดสัญญาณ Hand Shaking หรือ Protocol เพื่อเป็นมาตรฐานในการตกลงการรับส่งข้อมูลระหว่างอุปกรณ์เหล่านี้ ตัวอย่างระบบ Hard Shaking ที่ชัดเจนที่สุดคือการสื่อสารระหว่างเครื่องพิมพ์ (Printer) และ เครื่องคอมพิวเตอร์ ซึ่งไม่ว่าเครื่องคอมพิวเตอร์จะทำงานเร็วเพียงใด ก็ยังสามารถจะสื่อสารข้อมูลกับเครื่องพิมพ์ระบบเก่าที่ทำงานช้าได้เสมอ



ระบบสัญญาณ Hand Shaking โดยทั่วไปนั้น จะประกอบด้วยสัญญาณ Request และ Acknowledge คู่กันเสมอทั้งนี้ หากการสื่อสารระหว่างอุปกรณ์และ Controller เป็นลักษณะ Synchronous นั้นการตรวจสอบความพร้อมของข้อมูลจะทำได้โดยการดูที่สัญญาณนาฬิกาเป็นหลัก (รายละเอียดเพิ่มเติมของการวิเคราะห์ประสิทธิภาพของระบบ Synchronous และ Asynchronous นั้นสามารถหาคำอ่านได้จากหนังสือ Computer Architecture

ลักษณะของระบบสายสัญญาณ (Bus Arbitration)

- Daisy Chain Arbitration คือ การต่ออุปกรณ์ในลักษณะที่พ่วงกันไปเรื่อยๆ (เช่น กรณีของ USB Bus) ซึ่งข้อเสียของ Bus ในระบบนี้คือ เมื่อมีอุปกรณ์ต่อพ่วงมาก ความเร็วในการสื่อสารก็จะเฉลี่ยๆ กันไปให้เท่าๆ กัน
- Centralized Arbitration คือ การต่ออุปกรณ์แบบรวมสายสัญญาณเข้าด้วยกัน (เช่น กรณีของ PCI หรือ ISA Bus)
- Self-selection เป็นระบบ Bus ที่ที่นิยมใช้บนเครื่อง Macintosh (NuBus)
- Collision Detection เป็น Bus ที่มีลักษณะพิเศษคือมีการสื่อสารโดยมีผู้รับผู้ส่งได้มาก 1 ราย (เช่น ระบบ Bus ของ ระบบเครือข่ายคอมพิวเตอร์)

การจัดการอุปกรณ์ทางด้านซอฟต์แวร์

- Polling นั้นเป็นระบบการจัดการอุปกรณ์ต่อพ่วงโดยภาระส่วนใหญ่ อยู่ที่หน่วยประมวลผลกลางเป็นหลัก โดยหน่วยประมวลผลกลางนั้นจะต้องคอยสอบถามอุปกรณ์อยู่เสมอว่า อุปกรณ์มีข้อมูลที่ต้องถูกประมวลผลหรือไม่ เช่น หน่วยประมวลผลกลางต้องการอ่านข้อมูล Hard Disk เมื่อหน่วยประมวลผลกลางสั่งให้ Hard Disk ทำการอ่านข้อมูลแล้ว หน่วยประมวลผลกลางจะต้องคอยตรวจสอบเสมอว่า Hard Disk นั้นอ่านข้อมูลเสร็จหรือยัง โดยหน่วยประมวลผลกลางจะไม่สามารถประมวลผลงานอย่างอื่นได้ ซึ่งข้อเสียของระบบนี้คือ จะทรัพยากรหน่วยประมวลผลจำนวนมาก
- Interrupt นั้น อุปกรณ์จะแจ้งให้หน่วยประมวลผลกลาง ทราบว่าข้อมูลที่หน่วยประมวลผลกลางต้องการนั้นพร้อมแล้ว เช่น กรณีการอ่านข้อมูลจาก Harddisk เมื่อหน่วยประมวลผลกลางสั่งให้ Hard Disk ทำการอ่านข้อมูล ระหว่างที่กำลังรอให้

หน่วย Hard Disk ประมวลผลเพื่อให้ได้ข้อมูลมูลนั้น หน่วยประมวลผลกลางก็สามารถที่จะไปทำงานอย่างอื่นได้ ซึ่งเมื่อ Hard Diskประมวลผลเสร็จ ก็จะทำให้การแจ้งให้หน่วยประมวลผลกลางทราบผ่านระบบ Interrupt โดยขึ้นในการการ Interrupt นั้น หน่วยประมวลผลกลางจะทำการพักคำสั่งที่ต้องการไว้ใน Stack ก่อน แล้วจึงเปิดตาราง Interrupt Vector เพื่อทำงาน Interrupt Routine สำหรับงานนั้นๆ ข้อเสียของระบบนี้คือ การเขียนซอฟต์แวร์บนระบบ Interrupt นั้น มีความซับซ้อนยุ่งยาก

- DMA เป็นระบบประมวลผลที่ใช้กับการเชื่อมต่อฮาร์ดแวร์ที่เป็นระบบ DMA โดยหน่วยซอฟต์แวร์เมื่อได้รับสัญญาณร้องขอการทำ DMA จากอุปกรณ์ต่อพ่วง จะสั่งให้หน่วยประมวลผลกลางเลิกใช้งานหน่วยความจำ เป็นการชั่วคราว จากนั้นเมื่ออุปกรณ์ทำการสื่อสารข้อมูลกับหน่วยความจำเสร็จแล้ว จึงสั่งให้หน่วยประมวลผลกลางทำงานต่อ

ทั้งนี้เมื่อเปรียบเทียบระหว่างระบบ Polling และ ระบบ Interrupt แล้ว เราจะพบว่าการ Polling นั้นเหมาะกับการประมวลผลในลักษณะที่หน่วยประมวลผลกลางต้องทำงานเฉพาะเจาะจงเป็นประจำ ส่วนระบบ Interrupt เหมาะกับระบบที่ทำงานในลักษณะ Time Sharing หรือหน่วยประมวลผลกลางต้องประมวลผลงานหลายๆ อย่างในเวลาเดียวกัน

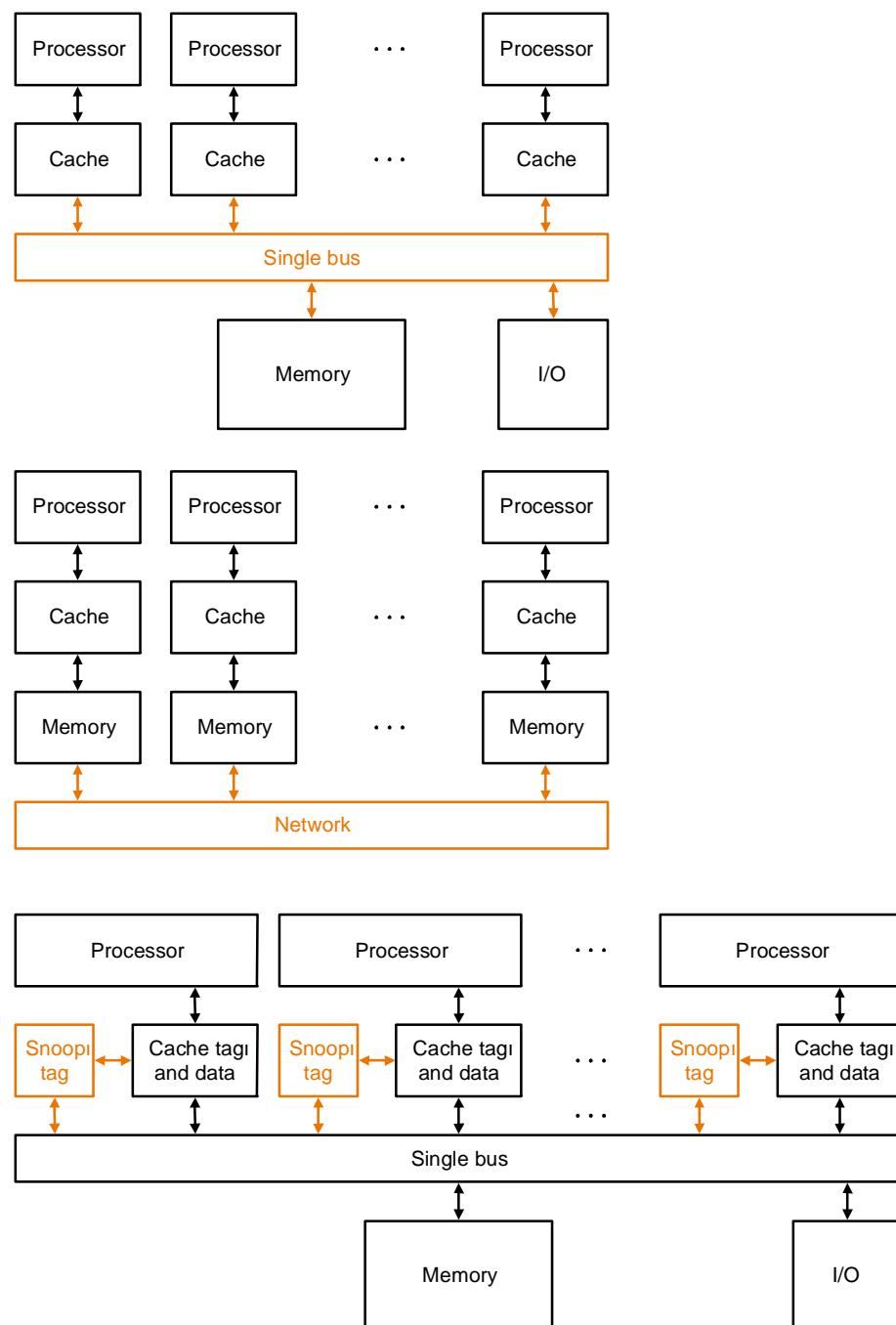
สรุป

ในการออกแบบระบบเพื่อเชื่อมต่ออุปกรณ์เข้ากับหน่วยประมวลผลกลางนั้น มีปัจจัยที่ต้องศึกษามากมาย เช่น ความเร็วในการสื่อสารข้อมูล จำนวนอุปกรณ์ที่สามารถต่อพ่วงได้ ความยาวของสายสัญญาณที่ใช้เชื่อมต่อ ราคา และประสิทธิภาพกรณีที่อุปกรณ์ต่อพ่วงเป็นจำนวนมาก ซึ่งข้อมูลดังกล่าวถึงภายในบทนี้เป็นเพียงรายละเอียดเบื้องต้นเพื่อเป็นพื้นฐานในการศึกษาต่อไปเท่านั้น

แบบฝึกหัดท้ายบท

1. หากท่านเป็นผู้พัฒนาซอฟต์แวร์สำหรับสื่อสารกับอุปกรณ์ชิ้นหนึ่ง ท่านจะเลือกเขียนซอฟต์แวร์โดยใช้ระบบ Polling หรือ ระบบ Interrupt เพราะเหตุใด
2. อุปกรณ์ประเภทใดบ้าง ที่มีลักษณะการสื่อสารกับระบบคอมพิวเตอร์ ในลักษณะ Asynchronous และเพราะเหตุใดจึงเป็นเช่นนั้น
3. การแยกระบบ I/O ออกจากระบบหน่วยความจำเป็นประโยชน์ในการพัฒนาอุปกรณ์คอมพิวเตอร์หรือไม่ อย่างไร
4. หากท่านเป็นผู้พัฒนาอุปกรณ์เชื่อมต่อ ท่านมีหลักเกณฑ์อย่างไรในการเลือกใช้ระบบ Memory Mapped I/O , ระบบ I/O Bus หรือ ระบบ DMA จงอธิบาย

บทที่ 9 สถาปัตยกรรมคอมพิวเตอร์สมัยใหม่ และสถาปัตยกรรมแบบขนาน



แบบฝึกหัดท้ายบท

เอกสารอ้างอิง