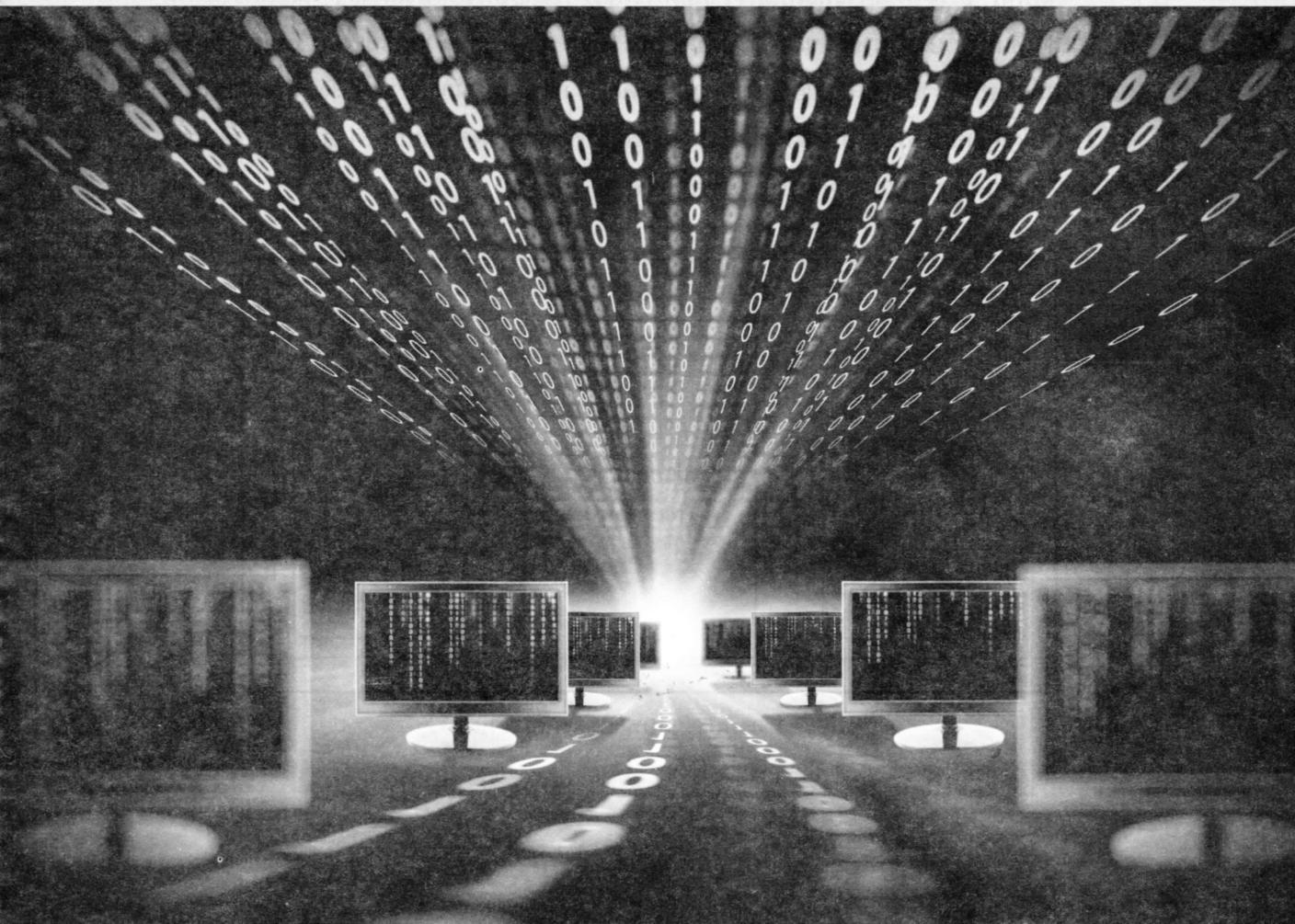


# 3

## สถาปัตยกรรมชุดคำสั่ง (Instruction Set Architecture)

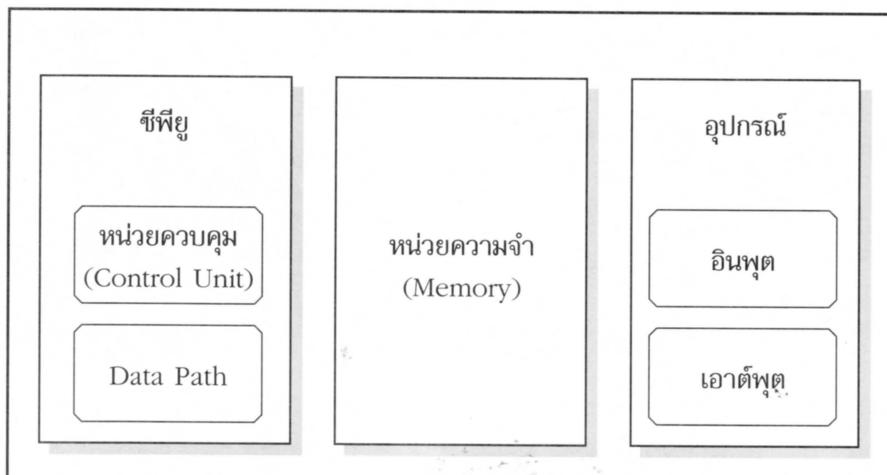


เนื่องจากชุดคำสั่งของเครื่องคอมพิวเตอร์มีความสำคัญมาก เพราะตัวแปลงภาษาจะแปลงโปรแกรมเป็นไบนารีได้เพื่อไปทำงานบนเครื่องคอมพิวเตอร์นั้นๆ โดยไบนารีได้ประกอบด้วยคำสั่งที่มาจากชุดคำสั่งของเครื่องคอมพิวเตอร์นั้น ไบนารีได้ที่ได้จะทำงานได้เร็วหรือไม่นั้น ลักษณะของชุดคำสั่งก็จะมีส่วนสนับสนุน ในบทนี้จะแนะนำความรู้พื้นฐานเกี่ยวกับชุดคำสั่งในด้านของการออกแบบเพื่อให้ได้ชุดคำสั่งที่จะให้ความสะดวกกับตัวแปลงภาษาในการสร้างได้ รวมไปถึงการออกแบบชาร์ดแวร์ที่สอดคล้องกับชุดคำสั่ง ซึ่งจะช่วยให้การทำงานในแต่ละคำสั่นนั้นทำได้เร็ว

ในบทนี้จะกล่าวถึงตัวอย่างของชุดคำสั่งของตรรกะ MIPS รูปแบบต่างๆ และการออกแบบชุดคำสั่งตรรกะนี้ในประเด็นที่เกี่ยวกับประสิทธิภาพ

### ■ 3.1 พื้นฐานของการทำงานของชีพียและคำสั่งแบบ RISC

ในบทที่ 1 เราได้กล่าวถึงไปแล้วว่าคอมพิวเตอร์ประกอบด้วยหน่วยต่างๆ ดังแสดงในรูปที่ 3.1 ในส่วนที่เกี่ยวกับการทำงานของคำสั่งหนึ่งๆ นั้น ได้แก่ ส่วนของหน่วยควบคุม (Control Unit) และ Data Path หน้าที่ของส่วนของหน่วยควบคุม คือ 1) การควบคุมการอ่านคำสั่งจากหน่วยความจำ 2) ให้สัญญาณควบคุมไปยังชาร์ดแวร์ต่างๆ เพื่อให้ทำงานตามคำสั่นนั้น 3) ควบคุมลำดับการทำงานของแต่ละคำสั่งตามผังการทำงานของโปรแกรม

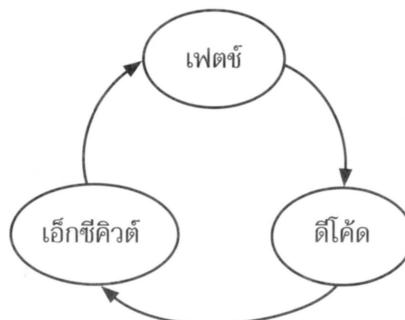


รูปที่ 3.1 องค์ประกอบหลักของคอมพิวเตอร์

สำหรับส่วนของ Data Path นั้น จะเป็นส่วนที่ระบุเส้นทางการไหลของข้อมูลไปยังชาร์ดแวร์หน่วยต่างๆ เพื่อทำงานตามคำสั่ง โดยหน่วยต่างๆ ที่ต้องการข้อมูลมาได้แก่ ALU (Arithmetic Logic Unit) ที่มีหน้าที่ในการคำนวณทางคณิตศาสตร์และลอจิก หรือรวมๆ เรียกว่าหน่วย Functional Unit และส่วน Data Path นั้นรวมไปถึงการเข้ามกันระหว่างหน่วย เช่น เข็มรีจิสเตอร์กับ Functional Unit หรือรีจิสเตอร์เข้ามกับสวิตซ์ เช่น Multiplexer และ Demultiplexer เป็นต้น เพื่อควบคุมการไหลของข้อมูลที่เหมาะสมสำหรับการคำนวณ

โดยทั่วไปแล้ว ในการทำงานคำสั่งหนึ่งๆ จะประกอบด้วย 3 ขั้นตอน ได้แก่ (ดูรูปที่ 3.2 ประกอบ)

1. **เฟตช์ (Fetch)** เป็นการอ่านคำสั่งจากหน่วยความจำเข้ามาในชีพีย์ โดยหน่วยควบคุมจะเป็นตัวควบคุมสัญญาณการอ่านคำสั่งเข้ามา
2. **ดีโคด (Decode)** เพื่อทำการถอดรหัสคำสั่งนั้น หน่วยควบคุมจะส่งสัญญาณเพื่อไปอ่านข้อมูลเข้า (Operand) ที่เหมาะสมมาประมวลผลกับคำสั่งนั้นๆ และจัดส่งสัญญาณที่เหมาะสมไปควบคุมสวิตซ์และควบคุมหน้าที่ของ Functional Unit
3. **ເອັກຊື້ຄົວຕົວ (Execute)** จะทำการคำนวณในหน่วย Functional Unit นั้นๆ เพื่อให้ได้ผลลัพธ์มาจัดเก็บไว้ในรีจิสเตอร์เพื่อใช้ในคำสั่งถัดไป และหน่วยควบคุมก็จะไปทำการเฟตช์คำสั่งถัดไปตามลำดับ



รูปที่ 3.2 ไฉเกิลการทำงานสำหรับการเฟตช์-ดีโคด-ເອັກຊື້ຄົວຕົວ

ในการออกแบบชุดคำสั่ง โดยทั่วไปแล้วจะเน้นความง่ายของชุดคำสั่งเป็นหลัก ความง่ายในที่นี้จะหมายถึง ง่ายต่อการนำไปสร้างชาร์ดแวร์ในระดับล่าง รวมทั้งในส่วนของการวางแผน Data Path

และการสร้างหน่วยควบคุม การออกแบบบุ๊กคำสั่งให้ง่ายนี้หมายถึง จะมีจำนวนรูปแบบคำสั่งไม่มาก และค่อนข้างมีรูปแบบตายตัว มีความยาวที่คงที่ (Fixed Length) ทำให้กระบวนการถอดรหัสคำสั่ง เป็นไปได้่ายตัวย และทำให้ฮาร์ดแวร์ในการถอดรหัสไม่ซับซ้อน นอกจากนี้ การคำนวณของหน่วย ALU จะกระทำการกับໂປເປອແຣນດີທີ່ເປັນຮີຈິສເຕອຣເທົ່ານັ້ນ ດັ່ງນັ້ນ คำสั่งທີ່ເກີຍກັບ ALU ເຊັ່ນ ADD, SUB, OR ເປັນຕົ້ນ ຈະມີໂປເປອແຣນດີປະເກທຮີຈິສເຕອຣເພື່ອຢ່າງເຕີຍ ທຳໃຫ້ຕົ້ນມີคำสั่งປະເກທ Load ເພື່ອທຳການອ່ານຂໍ້ມູນເຂົ້າມາໃນຮີຈິສເຕອຣກ່ອນ ແລະ คำสั่ง Store ເພື່ອທຳການຄ່າຢໍ້າຂໍ້ມູນຈາກຮີຈິສເຕອຣໄປເກີບຍັ້ງหน່ວຍຄວາມຈຳທີ່ຕົ້ນການ ລັກຂະນະບຸດคำสั่งແບບນີ້ເຮົາກວ່າແບບ Reduced Instruction Set Computer (RISC) ນັ້ນເອງ ເພື່ອຄວາມເຂົ້າໃຈ ໃຫ້ພິຈານາໂດັດຕ່ອໄປນີ້

$$X = A + B$$

ໃນການคำນວນປະໂຍຄນີ້ ຄ້າສ້າງອອກມາເປັນคำสั่งແບບຮັບແບບຕັບແອສແຂມບັນລືຕະກູລ RISC ຈາກຈະໄດ້ລັກຂະນະໂດັດ ດັ່ງນີ້

Load R1, A; อ່ານຄ່າຈາກหน່ວຍຄວາມຈຳທີ່ເກີບຕົວແປຣ A ເຂົ້າສູ່ຮີຈິສເຕອຣ R1

Load R2, B; อ່ານຄ່າຈາກหน່ວຍຄວາມຈຳທີ່ເກີບຕົວແປຣ B ເຂົ້າສູ່ຮີຈິສເຕອຣ R2

ADD R3, R1, R2; R3 = R1 + R2

Store X, R3; ເກີບຄ່າຈາກຮີຈິສເຕອຣ R3 ເຂົ້າສູ່ໜ່ວຍຄວາມຈຳທີ່ເກີບຕົວແປຣ X

ລັກຂະນະເຄື່ອງທີ່ຮອງຮັບคำสั่งຮູບແບບນີ້ ບາງຄັ້ງເຮົາເຮືອງວ່າເຄື່ອງແບບ Load/Store ແລະ ສັ້ນເກຕວ່າໃນການກະທຳคำสั่งທາງຄົນຒຕຄາສຕຽງຈະຕົ້ນການໂປເປອແຣນດີ 3 ຕັ້ງດັ່ງໃນคำสั่ง ADD ຂ້າງຕັ້ນ

ຈົງໆ ແລ້ວບຸດคำสั่งມີຫລາຍຮູບແບບ ເຊັ່ນ MIPS, ARM, IA-32, Sun SPARC, HP PA-RISC, IBM PowerPC, Intel (Compaq), Alpha ເປັນຕົ້ນ ປັຈຈຸບັນຄວາມນິຍມຂອງບຸດคำสั่งຕະກູລ ARM ຈະມີມາກີ່ນເຮືອຍໆ ເນື່ອງຈາກໂພຣເຊີສເຫຼວຣຕະກູລ ARM ໄດ້ຮັບຄວາມນິຍມມາກີ່ນໃນປັຈຈຸບັນ ເພຣະໜີຢູ່ປະເກທນີ້ມັກຖຸກໃໝ່ໃນອຸປກຣນີອີເລັກທຣອນິກລົ້ນນາດເລັກແລະ ຮະບນຟັງຕັ້ງ ຮວມທັງອຸປກຣນີໂທຣຕັພໍ ເຄລື່ອນທີ່ ຂຶ້ງໄດ້ຮັບຄວາມນິຍມມາກີ່ນໃນປັຈຈຸບັນ ໃນນະທີ່ຄວາມແພວ່ຫລາຍຂອງເຄື່ອງເດສກທີ່ອປແລະເຫຼົກໂວຣົກຄ່ອນຂ້າງຈະອູ່ຕົວແລ້ວ

## 3.2 พื้นฐานระบบเลขฐาน 2

ในเครื่องคอมพิวเตอร์จะใช้ระบบเลขฐาน 2 ทั้งหมด และในตัวอย่างขุดคำสั่งที่เราจะใช้อ้างอิงนี้จะเป็นตรารกุล MIPS จะมีความยาวเท่ากัน 32 บิต ดังนั้น จะได้ว่า 1 Word จะยาว 32 บิต ในตารางเป็นตัวอย่างการเข้ารหัสเลขฐาน 2 ในความยาว 32 บิต จะเห็นว่าจะให้ค่าบวกมากสุดไม่เกิน  $2^{32} - 1$  ตารางจะแสดงค่าในฐาน 16, ค่าในฐาน 2 และค่าในฐาน 10

ฐาน 16	ฐาน 2	ฐาน 10
0x00000000	0...0000	0
0x00000001	0...0001	1
0x00000002	0...0010	2
0x00000003	0...0011	3
0x00000004	0...0100	4
0x00000005	0...0101	5
0x00000006	0...0110	6
0x00000007	0...0111	7
0x00000008	0...1000	8
0x00000009	0...1001	9
...		
0xFFFFFFF0	1...1100	
0xFFFFFFF1	1...1101	
0xFFFFFFF2	1...1110	
0xFFFFFFF3	1...1111	

สำหรับเลขลบจะใช้วิธีการ 2's Complement ในการแทนค่า โดยจะทำการกลับบิต (ทำเป็น 1's Complement) และบวกค่า 1 เข้า สมการข้างล่าง เมื่อ X เป็นค่าในบิตใดๆ กรณี n บิต

$$-X = \overline{X_{n-1} X_{n-2} \dots X_0} + 000 \dots 01$$

ถ้าเป็นเลขทศนิยม

$$-X = \overline{X_{n-1} X_{n-2} \dots X_0} + 0.00 \dots 01$$

เช่น เลข  $-5$  ในระบบเลข  $4$  บิต จะทำเป็นเลข  $2$ 's Complement ได้  $\overline{0101}_2 + 1 = 1010_2 + 1 = 1011_2$

สังเกตว่าในระบบเลข  $2$ 's Complement บิตที่  $X_{n-1}$  จะแสดง Sign Bit ถ้าบิตนี้มีค่า  $1$  จะหมายถึงเลขลบ และถ้าเป็นค่า  $0$  จะหมายถึงค่าบวก ประโยชน์ของการใช้เลขระบบ  $2$ 's Complement ที่สำคัญคือ ทำให้การลบเลขง่ายขึ้น เพราะในการลบเลข เช่น  $Y - X$  เราจะใช้วิธีการบวกแทน นั่นคือ  $Y - X$  จะเหมือนกับการบวก  $-X$  เข้ากับ  $Y$  โดยค่า  $-X$  อยู่ในรูปของ  $2$ 's Complement หากจากการคำนวณผลลัพธ์หลังจากการคำนวณแล้ว ในการบวกก็ใช้วงจรการบวก  $1$  ด้วยจะบวก  $1$  เข้ากับผลลัพธ์หลังจากการคำนวณแล้ว ในการบวกก็ใช้วงจรการบวกตามปกติที่มีอยู่ได้ ดังนั้น ในการลบเลขจะใช้ร่วมกับวงจรทำหน้าที่การบวกได้เลย ไม่ได้เพิ่มความซับซ้อนเข้ามาให้กับการสร้างวงจรมากนัก

โดยปกติแล้ว ในการกระทำการบวกมีโอกาสเกิดสิ่งที่เรียกว่า Exception Condition หรือความผิดปกติได้ สาเหตุก็เช่น ตัวเลขที่เครื่องจะต้องแทนค่าที่มีขนาดเล็กหรือมีขนาดใหญ่เกินจำนวนบิตที่ใช้ในเครื่องจะเก็บได้ เนื่องจากจำนวนบิตมีจำกัด กรณีที่ค่าที่จะเก็บใหญ่เกินจำนวนบิตที่มีอยู่ จะเรียกว่าเกิดโอเวอร์ฟอลว์ (Overflow) และถ้าค่าที่จะเก็บเล็กเกินไป ไม่สามารถเก็บได้ด้วยจำนวนบิตที่มีอยู่ จะเรียกว่าเกิดอันเดอร์ฟอลว์ (Underflow) ตัวอย่างที่แสดงเหตุการณ์ โอเวอร์ฟอลว์ ได้แก่ ในระบบเลข  $4$  บิต เลขบวกที่เก็บได้มีค่าไม่เกิน  $0111_2$  หรือ  $7$  ถ้าเป็นเลข  $10$  ก็จะไม่สามารถเก็บได้โดยใช้  $4$  บิต นั่นคือจะเกิดโอเวอร์ฟอลว์นั่นเอง

ตารางค่าความจริงชี้แจงแสดงเงื่อนไขของการเกิดโอเวอร์ฟอลว์เมื่อมีการบวกเลขแบบ  $2$ 's Complement  $Z = X + Y$  เป็นดังนี้

$X_{n-1}$	$Y_{n-1}$	$C_{n-2}$	$Z_{n-1}$	$V$
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	0

จะพบว่าสมการบัญลีนของเงื่อนไขการเกิดโอล์ฟล์วีคือ  $v = \overline{x_{n-1} y_{n-1}} c_{n-2} + x_{n-1} \overline{y_{n-1} c_{n-2}}$  โดยที่  $x_{n-1}$  และ  $y_{n-1}$  เป็นบิต Sign ของเลข X และ Y ตามลำดับ และ  $c_{n-2}$  คือ บิตที่เกิดขึ้นจากการบวกบิต  $x_{n-2}$  และ  $y_{n-2}$  จากสมการของตัวแปร v ข้างต้น จะเห็นได้ว่าเงื่อนไขการเกิดโอล์ฟล์มีแค่ 2 กรณี คือ

$$\text{เมื่อ } x_{n-1} = y_{n-1} = 0 \text{ และ } c_{n-2} = 1$$

$$\text{และเมื่อ } x_{n-1} = y_{n-1} = 1 \text{ และ } c_{n-2} = 0$$

กรณีการเกิดโอล์ฟล์ค่าบวก เมื่อ X และ Y เป็นเลขบวกทั้งคู่และมีบิตที่เกิดขึ้นในตำแหน่งก่อนหน้า ทำให้ได้ค่าที่ใหญ่กว่าที่จะแทนได้ และกรณีการเกิดโอล์ฟล์ค่าลบ เมื่อ X และ Y เป็นเลขลบทั้งคู่ และผลบวกของเลขลบทั้งสองจะเป็นค่าติดลบมากซึ่งจะทำให้บิตหลักเป็น 0

ตัวอย่างเช่น ถ้าในระบบเลข 4 บิต

$$-5 + 6 = 1 \text{ ซึ่งจะตรงกับ } 1011_2 + 0110_2 = 0001_2 \text{ และได้ } C_2 = 1 \text{ ไม่เกิดโอล์ฟล์}$$

$$-5 + 4 = -1 \text{ ซึ่งจะตรงกับ } 1011_2 + 0100_2 = 1111_2 \text{ และได้ } C_2 = 0 \text{ ไม่เกิดโอล์ฟล์}$$

แต่

$$-5 - 6 = -11 \text{ ซึ่งจะตรงกับ } 1011_2 + 1010_2 = 0101_2 \text{ และได้ } C_2 = 0 \text{ ซึ่งเกิดโอล์ฟล์} \\ \text{และ}$$

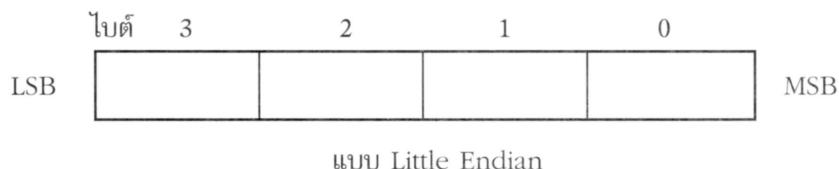
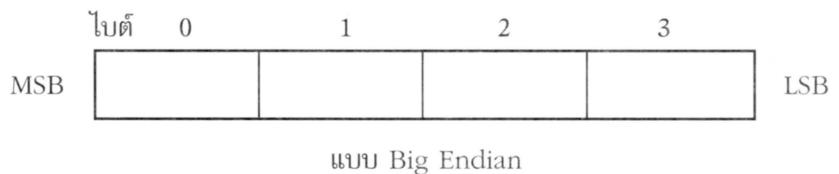
$$6 + 5 = 11 \text{ ซึ่งจะตรงกับ } 0110_2 + 0101_2 = 1011_2 \text{ และได้ } C_2 = 1 \text{ ซึ่งเกิดโอล์ฟล์}$$

นอกจากปัญหาโอล์ฟล์และอันเดอร์ฟล์แล้ว ปัญหาอื่น ได้แก่ การปัดเศษ ซึ่งมีสาเหตุมาจากการจำนวนบิตที่จะเก็บมีจำกัด ตัวอย่างเช่น ในการคูณเลข n บิต 2 ตัว ผลลัพธ์ที่ได้จะไม่เกิน  $2^n$  บิต ถ้าเครื่องเก็บได้แค่ n บิต ก็จะเก็บ n บิตที่มีนัยสำคัญสูงสุด (Most Significant Bit) และทำการเอา n บิตที่เหลือทิ้งไป วิธีการนี้เรียกว่าปัดเศษทิ้ง (Truncation) ข้อผิดพลาดที่เกิดจาก การปัดเศษทิ้งจะขึ้นอยู่กับจำนวนบิตที่ปัดทิ้งนั่นเอง ถ้าเป็นวิธีการปัดขึ้น (Round-up) จะเป็นการบวกค่า  $\frac{r^j}{2}$  (กำหนดให้ r คือฐานของเลข) เข้าไปกับจำนวนทั้งหมดก่อนแล้วทิ้ง n บิตหลังที่มีนัยสำคัญต่ำ (Least Significant Bit) เช่น ค่า 0.346712 ในเลขฐาน 10 ถ้าปัดทิ้งให้เหลือศูนย์ 3 ตำแหน่ง จะบวกค่า 0.0005 หรือ  $\frac{10^{-3}}{2}$  เข้าไปก่อนเป็น  $0.346712 + 0.0005 = 0.347212$  และ เอา 212 ทิ้งไป เหลือแต่ 0.347 ไปเก็บในเครื่อง แต่ถ้าเป็นการปัดเศษทิ้ง จะเก็บ 0.346 ไว้ในเครื่อง

ในการเก็บคำในหน่วยความจำ สมมติว่า 1 Word มีความยาวมากกว่า 1 ไบต์ เช่น ถ้าเป็น 4 ไบต์ จะต้องมีวิธีการจัดเรียงลำดับไปเบ็ดภายนอกใน Word เหล่านี้ด้วย

การเรียงลำดับไบต์ในแต่ละ Word ลักษณะที่เรียงไบต์ลำดับญี่ปุ่นจากซ้ายไปขวาเรียกว่า Big Endian ซึ่งใช้กับเครื่อง IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA เป็นต้น ตั้งแต่แสดงในรูปที่ 3.3 รูปบน ไบต์ 0 จะอยู่ซ้ายมือ

ในทางตรงกันข้าม การเรียงไบต์ในลักษณะที่เรียงไบต์สำคัญที่สุดจากขวาไปซ้ายเรียกว่า Little Endian ซึ่งใช้กับเครื่อง Intel 80x86, DEC Vax, DEC Alpha (Windows NT) เป็นต้น และแสดงในรูปที่ 3.3 รูปล่าง ไบต์ 0 จะอยู่ขวาเมื่อ



รูปที่ 3.3 การจัดเรียงไบต์สำหรับ Little Endian และ Big Endian

ในกรณีของตัวอักษร จะมีการเข้ารหัสตามมาตรฐาน เช่น ASCII (American Standard Code for Info Interchange) ซึ่งเป็นรหัสขนาด 8 บิต ดังแสดงในตารางในหน้าถัดไป (ที่มา: <http://ascii-table.com>)

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	-	o	del

ในการแทนค่าเลขฐานนิยม จะใช้รูปแบบ

$$M \times B^E$$

โดยที่ M คือ จำนวน (Mantissa)

B คือ ฐานของเลข (Base) ในที่นี้กำหนดให้เป็นฐาน 2

E คือ ยกกำลัง (Exponent)

โดยทั่วไปแล้ว ให้ B = 2 หรือเป็นเลขฐาน 2 จะได้รูปแบบย่อ คือ (M, E) เช่น ถ้าใช้ 3 บิตสำหรับทั้ง M และ E และสมมติว่า M และ E ใช้การแทนค่าแบบ Sign Magnitude

(X00, XXX) คือ ค่า 0

ค่าบวกที่เล็กที่สุด คือ  $(001_2, 111_2) = 1 \times 2^{-3} = 0.125$

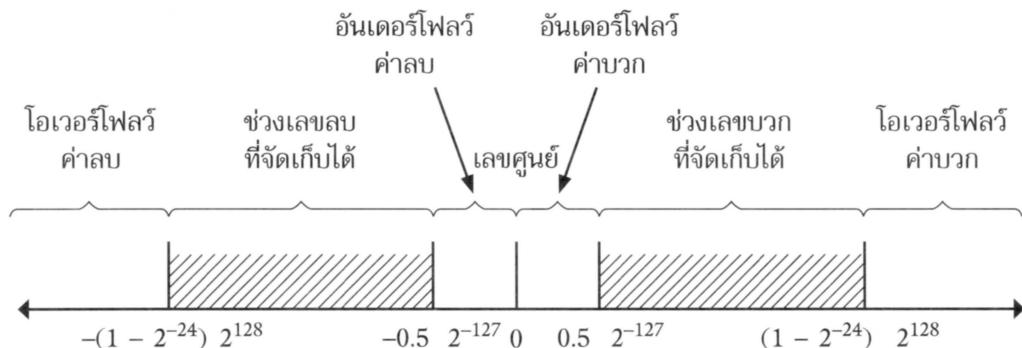
ค่าบวกที่มากที่สุด คือ  $(011_2, 011_2) = 3 \times 2^3 = 24$

ค่าลบที่เล็กที่สุด คือ  $(111_2, 011_2) = -3 \times 2^3 = -24$

ค่าลบที่มากที่สุด คือ  $(101_2, 111_2) = 1 \times 2^{-3} = -0.125$

สังเกตว่าเลขทศนิยมจะต้องการการประมาณค่าในการแทนค่าเพื่อให้อยู่ในรูปแบบดังกล่าว  
ในการแทนค่าลักษณะนี้ ค่าที่แสดงได้จะอยู่ในช่วง  $-2^{k-1}$  ถึง  $2^{k-1} - 1$  หรือ  $-2^{k-1} - 1$  ถึง  $2^{k-1}$  เมื่อ  $k$  คือจำนวนบิตของ Exponent ดังนั้น การแทนค่าข้อมูลแบบทศนิยมจะทำให้มีโอกาสเกิดอันเดอร์ไฟล์ว์หรือโอเวอร์ไฟล์ว์ได้

รูปที่ 3.4 เป็นตัวอย่างสำหรับ 128 บิต จะมีโอกาสเกิดโอเวอร์ไฟล์ว์ค่าลบและค่าบวกที่ค่ามากกว่า  $2^{128}$  และน้อยกว่า  $-(1 - 2^{-24})$  และอันเดอร์ไฟล์ว์ค่าลบและค่าบวกระหว่าง 0 ถึง  $-2^{-127}$  และ 0 ถึง  $2^{-127}$  ช่วงตัวเลขบวกที่จัดเก็บคือตัวเลขระหว่าง  $2^{128}$  ถึง  $2^{-127}$  และตัวเลขลบที่จัดเก็บใน คือระหว่าง  $-(1 - 2^{-24})$  และ  $-0.5$



รูปที่ 3.4 ช่วงของตัวเลขทศนิยม

นอกจากนี้ การแทนค่าเลขทศนิยมต้องมีการอนุมอලัยและมีใช้ Bias หรือ Excess การอนุมอලัยเป็นการทำให้เลขทศนิยมอยู่ในรูปแบบที่เหมือนกันก่อนการจัดเก็บ โดยที่ค่าของ Mantissa จะอนุมอලัยให้ค่าหลังจุดทศนิยมมากกว่า 1 โดยการปรับค่า Exponent ให้ถูกต้อง ตัวอย่าง ตัวเลขที่อนุมอලัยแล้ว ได้แก่  $0.1 \times 10^{19}$  (ในระบบเลขฐาน 10) ซึ่งอาจจะนำอนุมอලัยมาจากการ  $0.001 \times 10^{21}$  จะเห็นว่าในส่วนของ Mantissa นั้น จะต้องเลื่อนไปทางซ้ายเพื่อให้ตัวเลขหลังจุดเป็น 1 และเมื่อเลื่อนจุดไป จะต้องทำการปรับค่า Exponent ในที่นี้จะต้องลดค่า Exponent ลง

ส่วนการใช้ Excess เป็นการบวกค่าคงที่เพิ่มไปในส่วนของ Exponent เช่น Excess-3 ก็จะบวกเพิ่ม 3 ( $011_2$ ) โดยจะบวกเข้าค่าที่มีอยู่ แต่ข้อควรระวังคือ ในการบวกเลขในระบบ Excess-3 โดยใช้วงจรบวกเลขฐาน 2 จะได้ผลลัพธ์ที่มีค่าเกินอยู่ 3 เพราะแต่ละค่าเกิน 3 อยู่แล้ว จึงต้องทำการปรับค่าให้ถูกต้องก่อน เช่น การบวกของ  $5 + 9 = 14$  ในระบบ Excess-3 คือ

$$1000_2 = 5$$

$$\underline{1100}_2 = 9$$

$$1 \quad 0100_2 +$$

บิตทด  $0011_2$  การปรับค่า (Correction)

$$1 \quad 0111_2 = 4 \text{ ใน Excess-3}$$

ได้ผลลัพธ์ คือ 1 4

ในการแทนค่าเลขทศนิยม มักจะใช้มาตรฐาน IEEE 754 สำหรับเลข 32 บิต สำหรับเลข N ใดๆ ได้แก่

N =	Sign	Exponent	Mantissa
		E	M
1		8	23

รูปที่ 3.5 การแทนค่าเลขทศนิยม

โดยจะใช้บิตจำนวน 23 บิตเพื่อเก็บ Mantissa และใช้จำนวน 8 บิตเก็บ Exponent และใช้จำนวน 1 บิตเก็บเครื่องหมาย (Sign) เพื่อบอกว่าเป็นบวกหรือลบ ถ้าค่าเป็น 1 หมายถึงค่าลบ และเป็น 0 หมายถึงค่าบวก

ค่า N คือ  $(-1)^S 2^{E-127} (1.M)$  (โดยที่ Exponent ใช้รูปแบบ Excess-127 หรือ Bias-127)

ค่า E อยู่ระหว่าง  $0 < E < 255$  และ S คือ Sign

ให้ลังเกตว่าในการเก็บแบบนี้ เลข 1 ข้างหน้าในส่วนของ M จะไม่แสดง เช่น เมื่อ  $N = 1.5$

$$\begin{array}{ccccccc} 1 & \underline{0111} & \underline{1111} & & 1000 & \dots & 0 \\ S & & & & 8 \text{ bits} & & 23 \text{ bits} \end{array}$$

ในการแทนค่า มีการใช้ NAN (ย่อมาจาก Not A Number) ซึ่งเกิดจากตัวเลขที่นิยมที่ไม่สามารถแทนค่าได้ เช่น กรณีที่เกิดจากการหารด้วย 0

เมื่อ  $M \neq 0$        $E = 255$  N คือ NAN

เมื่อ  $M = 0$        $E = 255 \text{ N}$  คือ ค่าอินพุตติ้หรือโอล์ฟล์ว

$$\text{เมื่อ } 0 < E < 255 \quad N = (-1)^S \cdot 2^{E-127} \quad (1.M)$$

$$\text{เมื่อ } E = 0 \quad M \neq 0 \quad N = (-1)^S 2^{E-126} (0.M)$$

$$E = 0 \quad M = 0 \quad N = (-1)^S \quad 0$$

ตารางข้างล่างแสดง Single Precision และ Double Precision สำหรับเลขทศนิยมขนาด 32 บิตและ 64 บิต ตามลำดับ ช่อง E หมายถึง ช่วงของค่า Exponent ช่อง F หมายถึง ช่วงของค่า Fraction (Mantissa) ในระบบ Single Precision จำนวนบิตที่ใช้สำหรับ E เป็น 8 บิต และจำนวนบิตที่ใช้สำหรับ Fraction เป็น 23 บิต ส่วนในระบบ Double Precision จำนวนบิตที่ใช้สำหรับ E เป็น 11 บิต และจำนวนบิตที่ใช้สำหรับ Fraction เป็น 52 บิต

Single Precision		Double Precision		ความหมาย
E(8)	F(23)	E(11)	F(52)	
0	0	0	0	เลข 0
0	ไม่เป็น 0	0	ไม่เป็น 0	เลขที่ไม่นอนร์มอลไลซ์ ค่าบวกและลบ
± 1-254	ค่าใดๆ	± 1-2046	ค่าใดๆ	เลขทศนิยม ค่าบวกและลบ
± 255	0	± 2047	0	ค่าอิพินิตี ค่าบวกและลบ
255	ไม่เป็น 0	2047	ไม่เป็น 0	ไม่ใช่ตัวเลข (NAN)

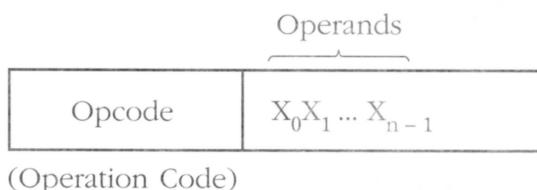
เนื่องจาก Exponent จะเก็บในรูปแบบ Bias เมื่อทำการบวกหรือลบ Exponent จะมีปัญหาทำให้ค่า Bias เกินมา 1 เช่น จึงต้องมีการปรับค่า Bias ศูนย์กลับด้วย

ตัวอย่างเบ็น

### ■ 3.3 รูปแบบชุดคำสั่ง (Instruction Set)

ในส่วนนี้จะกล่าวถึงรูปแบบพื้นฐานของคำสั่ง ลักษณะทั่วไปของชุดคำสั่งแบบต่างๆ รวมทั้ง การเข้าถึงโอเพอแรนด์แบบต่างๆ (Addressing Mode)

โดยทั่วไปแล้ว คำสั่งในภาษาแอสแซมบลีจะอยู่ในรูปแบบ



โดย Opcode ย่อมาจาก Operation Code ซึ่งได้แก่ รหัสของคำสั่งเพื่อบอกว่าเป็น คำสั่งใด และจะใช้เป็นรหัสไปสั่งให้ ALU ทำงาน รวมทั้งสั่งหน่วยควบคุมให้ส่งสัญญาณที่เกี่ยวข้อง และส่งโอเพอแรนด์หรือบางครั้งเรียกว่า Address Field X<sub>i</sub> ซึ่งเป็นชุดของบิต ซึ่งเป็นหมายเลขอริจิสเตอร์หรือตำแหน่งหน่วยความจำ หรืออาจจะเป็นค่าคงที่ (Immediate)

สำหรับขนาดของคำสั่งนี้จะยาวเท่าไรขึ้นอยู่กับสถาปัตยกรรมชุดคำสั่งของเครื่องตระกูลนั้น อย่างเช่นของตระกูล MIPS ที่จะกล่าวถึงต่อไป แต่ละคำสั่งจะมีขนาด 32 บิต และขนาดของ Opcode จะยาว 6 บิตเสมอ แต่สำหรับตระกูลของ 80x86 แต่ละคำสั่งจะมีขนาดไม่เท่ากัน อาจจะตั้งแต่ 8 บิตจนถึง 64 บิต และขนาดของ Opcode เป็น 8 บิต เป็นต้น

ในส่วนของ Address Field หรือโอเพอแรนด์นั้น อาจจะกำหนดได้หลายรูปแบบ โดยจะพิจารณาถึง 1) จำนวนโอเพอแรนด์ และ 2) รูปแบบการเข้าถึงโอเพอแรนด์ หรือที่เรียกว่า Addressing Mode สำหรับเครื่องแต่ละตระกูลก็จะมีทั้ง 2 ข้อนี้แตกต่างกัน เมื่อพิจารณาร่วมกันแล้วจะทำให้เกิดความหลากหลายรูปแบบในการระบุโอเพอแรนด์ของคำสั่งนั้นเอง

ในส่วนของจำนวนโอเพอแรนด์นั้น อาจจะมีจำนวนได้ตั้งแต่ 0 ไปจนถึง 3 ตัว ถ้าไม่มีโอเพอแรนด์เลย ก็จะหมายถึงใช้การสแตกในการทำงาน โดยจะใส่โอเพอแรนด์ไว้ในสแตก และเมื่อคำนวนเสร็จแล้ว ก็จะใส่ผลลัพธ์ไว้ในสแตก กรณีของ 1 โอเพอแรนด์ จะหมายถึงว่าจะไปกระทำกับโอเพอแรนด์ที่ชื่อน้อย ซึ่งอาจจะมีชื่อเฉพาะที่อยู่ภายใต้ เนื่อง อาจจะเป็นakkิวมูลेटอร์ (Accumulator) ในการกระทำต่างๆ โอเพอแรนด์ที่ระบุจะถูกนำไปดำเนินการกับakkิวมูลेटอร์นี้ และผลลัพธ์จะอยู่ใน

เอกสารคิวมูลเตอร์นี้เขียนกัน สำหรับกรณี 2 โอเปอเรนด์นั้น จะดำเนินการกับโอเปอเรนด์ทั้งสองที่ระบุไว้เลย แต่หลังจากทำเสร็จแล้วนั้น ผลลัพธ์จะอยู่ที่โอเปอเรนด์ตัวใดตัวหนึ่งเสมอ เช่น อาจจะเป็นโอเปอเรนด์ตัวทางซ้ายเสมอ เป็นต้น และในกรณี 3 โอเปอเรนด์ จะเห็นได้ชัดเจนว่าใช้โอเปอเรนด์ 2 ตัวเป็นแหล่งข้อมูลเข้า (Source) ของตัวดำเนินการ และอีกด้วยจะเป็นที่เก็บผลลัพธ์นั่นเอง ในการเลือกใช้รูปแบบใดนั้น จะมีผลต่อการสร้าง Data Path เพื่อส่งข้อมูลจากแหล่งกำเนิดโอเปอเรนด์ไปยัง ALU และการส่งผลลัพธ์ที่ได้ไปเก็บตามรูปแบบที่เหมาะสมอีกด้วย ในกรณีของชุดคำสั่งตรรกะ MIPS จะเป็นรูปแบบ 3 โอเปอเรนด์ และแบบตรรกะ 80x86 จะเป็นรูปแบบ 2 โอเปอเรนด์ เป็นต้น

สำหรับวิธีการเข้าถึงโอเปอเรนด์ในคำสั่งหนึ่งมีหลายวิธี ในที่นี้จะกล่าวถึงวิธีที่นิยมใช้กัน แต่ในชุดคำสั่งบางตรรกะจะมากกว่านี้ เช่น ในกรณีของตรรกะ 80x86 เป็นต้น การมี Addressing Mode หลายแบบก็จะทำให้การสร้าง Data Path ของทางเดินของข้อมูลเข้าสู่ ALU ขับข้อนามากขึ้น เนื่องจากต้องมีทางเลือกมากขึ้น เพราะว่ามี Addressing Mode หลายรูปแบบ และความขับข้อนทางฮาร์ดแวร์นี้อาจจะมีผลกับเวลาของ 1 รอบลัญญาณนาฬิกาของเครื่องได้ ตัวอย่างของ Addressing Mode เช่น

1. Immediate Addressing: เป็นแบบที่โอเปอเรนด์เก็บค่าคงที่ใช้ในการคำนวนของ ALU ได้เลย
2. Direct Addressing: โอเปอเรนด์จะเป็นตัวแปรที่เก็บค่าตำแหน่งของหน่วยความจำ กำหนดในรีจิสเตอร์
3. Indirect Addressing: โอเปอเรนด์เป็นแอดเดรสที่เป็นค่าบอกรอตำแหน่งของข้อมูล
4. Relative Addressing: โอเปอเรนด์จะบอกรอตำแหน่ง แต่บอกรอเป็นค่าสัมพัทธ์กับค่าบางค่า เช่น ค่าใน PC การหาค่าแอดเดรสจริงจะต้องทำการคำนวนเพื่อหาตำแหน่งของข้อมูลจริง (Absolute Address) ซึ่งจะเกิดขึ้นในเวลาทำงานของคำสั่งนั้น ในการคำนวนจะเป็นลักษณะเดียวกับการหา Effective Address นั้นคือสำหรับ Effective Address A หากได้จาก  $R + D$  โดยที่  $D$  คือ Displacement และ  $R$  คือจุดเริ่มต้น

### ข้อดีของการใช้ Relative Addressing คือ

- ข้อมูลเกี่ยวกับแอดเดรสไม่ต้องเก็บในโอเพอแรนด์ของคำสั่งทั้งหมด เพื่อลดความยาวของคำสั่ง เก็บเฉพาะค่าที่สัมพันธ์กับจุดเริ่มต้น หรือบางครั้งเรียกว่า Offset
- เมื่อเปลี่ยนค่า R หรือจุดเริ่มต้น ทำให้ Absolute Address สามารถเปลี่ยนแปลงได้ ในเวลาที่กำลังทำงาน ความสามารถนี้เรียกว่าการย้าย (Relocate) พื้นที่ในการทำงาน โดยการเปลี่ยนเฉพาะค่าจุดเริ่มต้นที่เก็บใน Base Register เท่านั้น
- ค่า Offset เมะะกับการโปรแกรมที่ทำงานเป็นลูปซึ่งใช้พื้นที่ข้อมูลติดกัน เช่น อ้างถึงอาร์เรย์ A[i] ในแต่ละรอบ เช่น ในเข้าถึง A[0], A[1], A[2], ... ตามลำดับของลูป ในลักษณะนี้ เราอาจจะใช้ Base Address เก็บค่าเริ่มต้นแสดงตำแหน่งอาร์เรย์ A และใช้รีจิสเตอร์อีกตัวเก็บค่าตำแหน่งที่เกิดจากอินเด็กซ์ i ตามลูป จะทำให้เปลี่ยนค่าตำแหน่งที่ต้องการอ้างถึงข้อมูลในพื้นที่ถัดไปได้โดยง่าย

ในการนี้ที่ข้อมูลในโอเพอแรนด์มีขนาดเล็กกว่ารีจิสเตอร์ จะต้องมีการขยายโอเพอแรนด์เพื่อทำให้มีขนาดพอติดกับจำนวนบิตในรีจิสเตอร์ เช่น คำสั่งหนึ่งๆ ต้องการโอเพอแรนด์จากรีจิสเตอร์ขนาด 32 บิต ถ้าข้อมูลที่ได้ดึงมาจากพิล์ด Immediate เป็นค่าคงที่มีขนาด 16 บิต ค่าคงที่นี้จะต้องถูกขยายขนาดให้เป็น 32 บิตก่อนที่จะใส่ในรีจิสเตอร์เพื่ออ่านเข้าสู่ ALU และนำไปทำงานได้ วิธีการนี้เรียกว่าการทำ Sign Extension ซึ่งเป็นการเติมค่า 0 หรือ 1 เป็นจำนวน  $n - m$  บิต (เมื่อขนาดของโอเพอแรนด์ที่ต้องการคือ  $n$  บิต และข้อมูลมีขนาด  $m$  บิต โดยที่  $n > m$ ) โดยจะเติม 0 หรือ 1 เข้าไปนั้นขึ้นกับบิต Sign ของข้อมูล ถ้าบิต Sign เป็น 1 แสดงว่าเป็นค่าลบ ดังนั้นจะต้องเติม 1 ไปจำนวน  $n - m$  บิต และถ้าบิต Sign เป็น 0 แสดงว่าข้อมูลนั้นเป็นค่าบวก จึงต้องเติม 0 ไปข้างหน้าจำนวน  $n - m$  บิต

$$S_{n-m} = S....$$

โอเพอแรนด์

จำนวน  $n - m$  บิต

จำนวน  $n$  บิต

ตัวอย่างเช่น

1101 1011 1011 ให้ขยายเป็น 16 บิต จะได้

↑  
บิต Sign = 1

เติมค่า 1 เข้าไปจำนวน  $n - m = 4$

◆—◆  
1111 1101 1011 1011

ในการทำเช่นนี้จะไม่มีผลต่อค่าของข้อมูลแต่อย่างไร เพราะว่าถ้าบิต Sign เป็น 0 การเติม 0 เข้าไปข้างหน้าไม่มีผลต่อค่าของข้อมูล ถ้าบิต Sign เป็น 1 และเติม 1 เข้าไปข้างหน้าในเลขแบบ แบบ 2's Complement ค่าจะไม่เปลี่ยนแปลงเช่นกัน

ในการนี้ของการกำหนดตำแหน่งของหน่วยความจำในโอเพอเรนเตอร์ก็เช่นกัน ดังเช่นใน Addressing Mode แบบ Relative Addressing จะมีการใช้การกำหนด Address Offset ขนาด  $n$  บิต แต่ไฟล์ดีบองโอเพอเรนเตอร์ในคำสั่งอาจจะเก็บเพียง  $m$  บิต แต่เมื่อหลังจากการอุดตรัสร์คำสั่งแล้ว จะมีการทำ Sign Extension เพื่อขยายและเก็บค่าโอเพอเรนเตอร์ให้เป็น  $n$  บิต โดยที่  $m < n$  แล้วจึงนำมานำวกับค่าเริ่มต้นเพื่อหา Effective Address ต่อไป เช่น ถ้าขนาดรีจิสเตอร์เป็น 32 บิต แต่ในคำสั่งรูปแบบ Relative Addressing Mode เก็บส่วนของ Offset มีขนาด 11 บิต ดังนั้น เมื่ออ่านคำสั่งทั้งหมดมา จะต้องขยายไฟล์ Offset นี้จาก 11 บิตมาเป็น 32 บิต จึงเก็บในรีจิสเตอร์และทำการประมวลผลด้วย ALU ต่อไปได้

ในชุดคำสั่ง จะมีการแยกประเภทของคำสั่งเป็นหลายประเภท และการเข้ารหัส Opcode ให้สอดคล้องกับประเภทด้วย ตามปกติสามารถแบ่งได้ดังนี้

1. Data Transfer เป็นกลุ่มคำสั่งเกี่ยวกับการโอนย้ายข้อมูลระหว่างหน่วยความจำและรีจิสเตอร์ เช่น คำสั่ง Load และ Store
2. Arithmetic Instruction เป็นกลุ่มคำสั่งเกี่ยวกับการคำนวณทางคณิตศาสตร์ เช่น ADD, SUB เป็นต้น
3. Logic Instruction เป็นกลุ่มคำสั่งเกี่ยวกับการกระทำการทำทางตรรกะ เช่น AND, OR เป็นต้น

4. Program Control เป็นกลุ่มคำสั่งเกี่ยวกับการควบคุมลำดับการทำงานของโปรแกรม ให้เปลี่ยนค่า PC เพื่อเปลี่ยนลำดับการทำงานของโปรแกรม ได้แก่ คำสั่ง jmp Branch หรือ Jump ทั้งแบบมีเงื่อนไขและไม่มีเงื่อนไข การเรียกโปรแกรมย่ออย และการ Return จากโปรแกรมย่ออย
5. Input/Output เป็นกลุ่มคำสั่งเกี่ยวกับการติดต่อกับภายนอก นอกเหนือไปจากนี้ อาจจะมีคำสั่งเกี่ยวกับการร้องขอการใช้ Operating System Call ต่างๆ ก็ได้

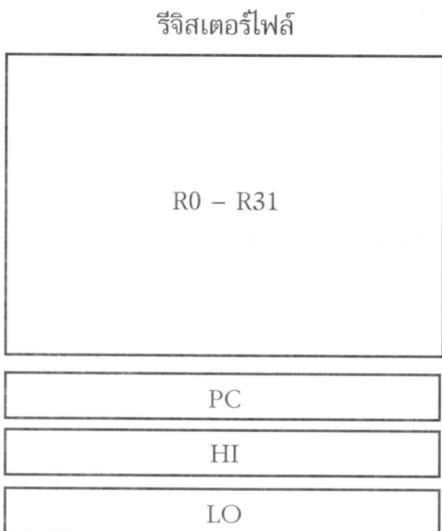
### ■ 3.4 ตัวอย่างของคำสั่งแบบ MIPS

ในคำสั่งตรรกะ MIPS นั้นจะใช้ลักษณะการออกแบบของ RISC โดยจะเน้นให้มีรูปแบบคำสั่ง ไม่มาก แต่ละคำสั่งมีความยาวเท่ากัน มีประเภทคำสั่งน้อย มีรูปแบบ Addressing Mode จำกัด มีคำสั่ง Load/Store ไว้อ่าน/เขียนค่าในหน่วยความจำ และการกระทำการทางคณิตศาสตร์กระทำกับโอเพอเรนเตอร์ที่เป็นรีจิสเตอร์เท่านั้น

ประเภทของคำสั่งจะแบ่งเป็น

1. ประเภทการคำนวณที่ใช้ ALU เช่น การบวก/ลบเลขจำนวนเต็ม การกระทำการทางตรรกะ ลอจิก การเลื่อนบิตไปทางซ้ายหรือขวา เป็นต้น
2. ประเภท Load/Store เป็นคำสั่งเกี่ยวกับการอ่านข้อมูลจากหน่วยความจำเข้าสู่รีจิสเตอร์ และย้ายค่าจากรีจิสเตอร์ไปยังหน่วยความจำ
3. ประเภท Jump และ Branch เป็นคำสั่งเกี่ยวกับการเปลี่ยนค่าในรีจิสเตอร์ที่ควบคุมการทำงาน ลำดับการทำงานของโปรแกรมทั้งแบบมีเงื่อนไขและไม่มีเงื่อนไข รวมทั้ง คำสั่งเกี่ยวกับการเรียกใช้โปรแกรมย่ออย
4. ประเภท Floating Point เป็นการคำนวณสำหรับเลขทศนิยม โดยอาศัยโคโปรเซสเซอร์ (Coprocessor) มากว่า
5. ประเภทการจัดการหน่วยความจำ เป็นคำสั่งเกี่ยวกับการจัดการหน่วยความจำและหน่วยความจำเสี้ยวนในระดับของระบบปฏิบัติการ
6. คำสั่งพิเศษอื่นๆ เช่น คำสั่งเกี่ยวกับหน่วยกราฟฟิก เป็นต้น

สำหรับรีจิสเตอร์มีทั้งหมด 32 ตัว แต่ละตัวมีขนาด 32 บิต เรียกว่าสั้นๆ ว่า R0, R1, ..., R31 มีรีจิสเตอร์พิเศษได้แก่ PC, HI, LO คำสั่งมี 3 รูปแบบ รูปแบบละ 32 บิต การวางแผนการเก็บรีจิสเตอร์แสดงได้ดังรูปที่ 3.6



รูปที่ 3.6 การเรียงลำดับรีจิสเตอร์ในรีจิสเตอร์ไฟล์

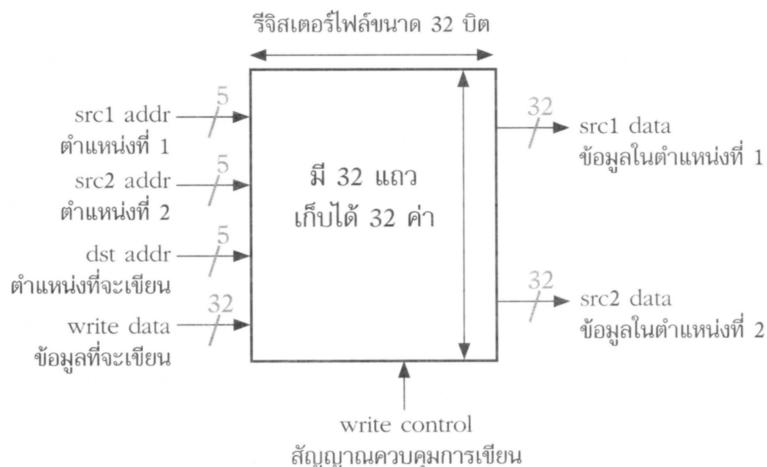
โดยแท้จริงแล้ว แต่ละรีจิสเตอร์จะมีความหมายดังแสดงในตารางด้านล่าง ในช่องแรกจะแสดงชื่อของรีจิสเตอร์ที่กำหนดในคำสั่ง ช่องที่สองจะแสดงลำดับที่ในรีจิสเตอร์ไฟล์ (ตามลำดับการจัดเก็บในรูปที่ 3.6) ช่องต่อไปแสดงหน้าที่ของ การใช้งานโดยทั่วไป

ชื่อ	ตัวที่	การใช้งาน
\$zero	0	เก็บค่าคงที่ 0 (ในชาร์ดแวร์)
\$at	1	จองไว้ (Reserve) สำหรับ Assembler
\$v0 - \$v1	2-3	เก็บค่าส่งคืน (Returned Values)
\$a0 - \$a3	4-7	เป็นอาร์กิวเมนต์ของโปรแกรมย่อย
\$t0 - \$t7	8-15	เก็บค่าชั่วคราว (Temporaries)
\$s0 - \$s7	16-23	จัดเก็บค่าไว้ใช้งาน (Saved Values)
\$t8 - \$t9	24-25	เก็บค่าชั่วคราว (Temporaries)

(ต่อ)

ชื่อ	ตัวที่	การใช้งาน
\$gp	28	เป็นพอยน์เตอร์แบบ Global
\$sp	29	เป็นพอยน์เตอร์ของสแต็ก
\$fp	30	เป็นพอยน์เตอร์ของเฟรม (Frame Pointer)
\$ra	31	เก็บค่าส่งคืนสำหรับ (Return Address) ในชาร์ดแวร์

ลักษณะของรีจิสเตอร์ไฟล์เป็นดังรูปที่ 3.7



รูปที่ 3.7 โครงสร้างรีจิสเตอร์ไฟล์

รีจิสเตอร์ไฟล์จะอ่านได้พร้อมกัน 2 ตัว ข้อมูลจะถูกอ่านจากรีจิสเตอร์ที่กำหนด โดยตำแหน่งทั้งสองกำหนดโดย src1 addr และ src2 addr และข้อมูลที่อ่านได้จากทั้ง 2 ตำแหน่งจะส่งไปยังสายสัญญาณ src1 data และ src2 data ขนาด 32 บิตซึ่งทำให้สามารถอ่านໄວ่เปอร์ແรนด์ได้พร้อมกัน 2 ตัว ทั้งนี้เพื่อสนับสนุนการทำงานของ ALU ที่ต้องการ 2 ໄວ่เปอร์ແรนด์ในการคำนวณต่างๆ สำหรับการเขียนนั้น ตัวรีจิสเตอร์ที่ถูกเขียนจะกำหนดโดย dst addr และข้อมูลที่จะเขียนกำหนดโดยสัญญาณ write data สัญญาณ write control = 1 เป็นสัญญาณขนาด 1 บิตเพื่อควบคุมการเขียน กำหนดให้เป็น 1 ถ้าต้องการระบุว่าต้องการจะเขียนໄไว้ยังรีจิสเตอร์ จะเห็นว่าในแต่ละเลเยนของสัญญาณจะบอกจำนวนบิตที่ส่งໄไว้ เช่น ตัวรีจิสเตอร์จะถูกกำหนดด้วยขนาดเพียง 5 บิต เพราะว่า MIPS มีรีจิสเตอร์ 32 ตัว และข้อมูลที่อ่านได้หรือเขียนลงໄปจะมีขนาด 32 บิต เพราะขนาด Word เท่ากับ 32 บิต

สำหรับรูปแบบของคำสั่งมีเพียง 3 รูปแบบ เพื่อให้ง่ายต่อการถอดรหัสคำสั่ง และมีจำนวน

### 5 Addressing Mode

- R-Type ใช้กับคำสั่งเกี่ยวกับ ALU Operation (ย่อมาจาก Register Type) เพราะว่า ต้องการโไปเปลอแรนด์ทั้ง 2 ตัวเป็นเรจิสเตอร์ และเขียนผลลัพธ์ไปยังเรจิสเตอร์
- I-Type ใช้กับคำสั่งการคำนวนที่ใช้ ALU ที่โไปเปลอแรนด์เป็นค่าคงที่ (Immediate) ใช้ กับคำสั่งกระโดดแบบมีเงื่อนไข (Branch) ซึ่งเป็นการกระโดดแบบสัมพัทธ์ (Relative Jump)<sup>1</sup> และคำสั่ง Load/Store
- J-Type ใช้กับคำสั่งกระโดดแบบไม่มีเงื่อนไข ซึ่งเป็นการกระโดดแบบ Absolute Jump คำสั่ง Call

สรุปรูปแบบคำสั่งของ MIPS ได้ 3 รูปแบบ ดังแสดงในรูปที่ 3.10

คำสั่ง R-Type	Opcode	rs	rt	rd	sa	funct	
คำสั่ง I-Type	Opcode	rs	rt	ค่าคงที่ (Immediate)			
คำสั่ง J-Type	Opcode	ตำแหน่งที่จะกระโดดไป (Jump Address)					

รูปที่ 3.10 รูปแบบคำสั่งของ MIPS

### 3.4.1 ໂທມດກາຮອງແວຕເຕຣສຂອງ MIPS

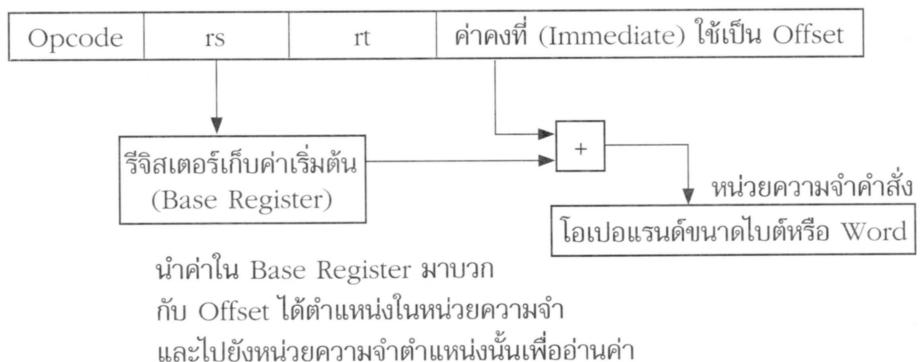
เครื่อง MIPS จะมีหมดการอ้างแอดเดรสดังนี้

1. Register Addressing สำหรับการเข้าถึงโไปเปลอแรนด์ที่เป็นเรจิสเตอร์ จะไปอ่านค่าใน เรจิสเตอร์ที่กำหนด เช่น ระบุไว้ในพาร์ต rs หรือ rt เป็นต้น

<sup>1</sup> โดยกำหนดตำแหน่งที่จะไปสัมพัทธ์กับค่าในเรจิสเตอร์ PC



2. Base Addressing สำหรับการเข้าถึงໂປເປອແຣນດ໌ ໂດຍການນຳຄ່າດໍາແນ່ງ Base ມາບວກກັບ Offset ສູງແບບນີ້ໃຊ້ກັບ I-type ໂດຍ rs, rt ຈະນອກຮີຈິສເຕ່ອຮັດທັນທາງ ແລະຄ່າໃນ rs ຈະເກີນຄ່າ Base Address ແລະຕັ້ງ Offset ເປັນຄ່າຄົງທີ່ຫົ່ວ້ອ Immediate ຄ່າໃນ rs ຈະຄຸກນຳມານຍາຍບົດກ່ອນ ແລ້ວຄ່ອຍນຳມານວກດ້ວຍ ALU ຢ້ອງ Adder



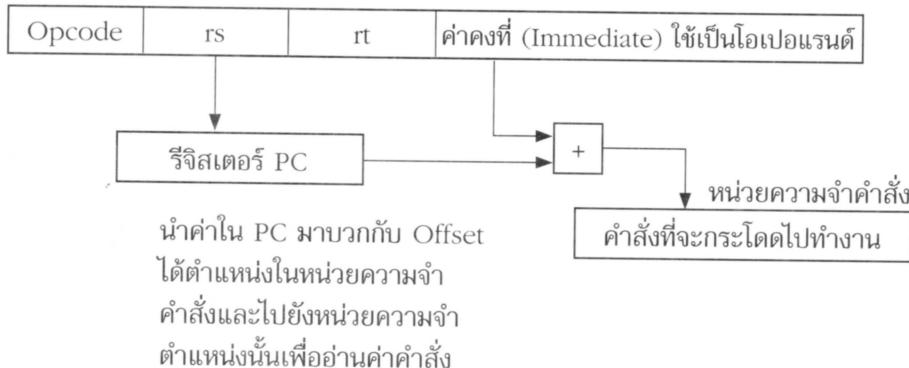
ในรูปแบบนี้จะทำให้เกิดรูปแบบต่างๆ เช่น

- Register Relative (Indirect) เมื่อกำหนดให้ค่าใน Offset เป็น 0 เช่น 0(\$a0)
  - Pseudo-direct เมื่อให้รีจิสเตอร์พิเศษ \$zero ที่บรรจุค่า 0 และ Offset เป็นค่าคงที่ใดๆ เช่น 1000(\$zero)

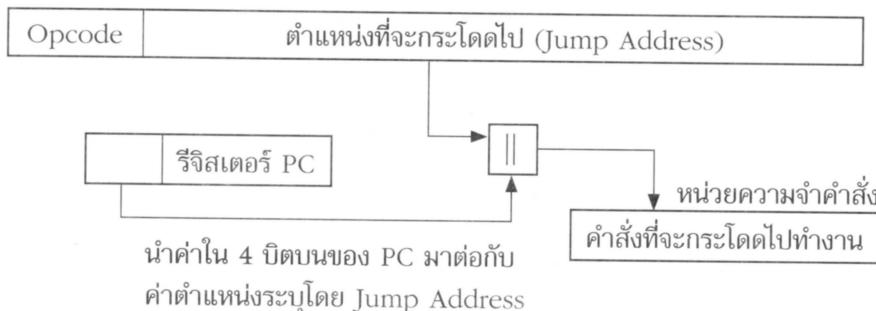
3. Immediate Addressing ใช้เมื่อโวเปอแรนด์เป็นค่าคงที่ขนาด 16 บิต

Opcode	rs	rt	ค่าคงที่ (Immediate) ใช้เป็นโอดีรันด์
--------	----	----	---------------------------------------

4. PC-Relative ใช้สำหรับคำสั่ง Branch แบบ Relative ซึ่งตำแหน่งจะเกิดจากการบวกค่าคงที่ 16 บิตกับ PC + 4 โดยค่า Offset จะถูกขยายบิตก่อน และนำมาบวกกับค่าใน PC ด้วย ALU หรือ Adder และจะบวกตำแหน่งที่จะกระโดดไป



5. Pseudo-direct ใช้กับการกระโดดแบบ Absolute โดยการนำ Address ขนาด 26 บิต มาต่อ กับ ค่าใน PC ที่อยู่ใน 4 บิตบนให้กลายเป็น 32 บิต และใช้ระบุตำแหน่งที่จะกระโดดไปเลย



### 3.4.2 ประเภทของคำสั่ง

รูปแบบคำสั่งของ MIPS มีเพียง 3 รูปแบบ แต่ละรูปแบบมีขนาดเท่ากัน 32 บิต แต่ละรูปแบบใช้กับประเภทคำสั่งเฉพาะอย่าง ในที่นี้จะอธิบายตามรายละเอียดประเภทของคำสั่งและแสดงตัวอย่างคำสั่ง

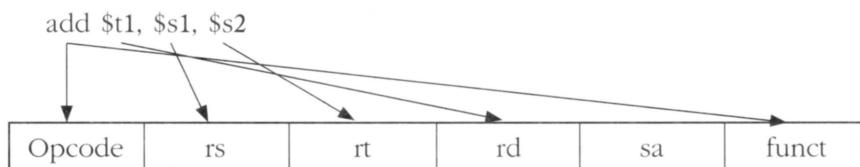
### 3.4.2.1 ประเภท R-Type

คำสั่งประเภท ALU ใช้คำนวน เบ่น บวก ลบ หรือการกระทำทางลอจิกที่ใช้ ALU โดยให้โอเปอเรนด์ทั้ง 3 ตัวเป็นรีจิสเตอร์ทั้งหมด 1 คำสั่งทำเพียง 1 ตัวดำเนินการเท่านั้น เบ่น

```
add $t1, $s1, $s2
sub $t1, $s1, $s2
```

ซึ่งมีความหมาย คือ  $\text{Destination} \leftarrow \text{Source1 op Source2}$

โดยโอเปอเรนด์ตัวแรกจะหมายถึง Destination เสมอ ถ้าขณะนี้ใช้รูปแบบ R-Type ขึ้นจะเป็นคำสั่ง ALU ที่จะกระทำกับโอเปอเรนด์ทั้ง 2 ตัวที่เป็นรีจิสเตอร์ระบุใน Source1 และ Source2 และเก็บผลลัพธ์ไว้ที่รีจิสเตอร์ระบุโดย Destination จากคำสั่ง add ข้างต้น แต่ละฟิลด์จะแบ่งได้ดังแสดงในรูปที่ 3.11



รูปที่ 3.11 การ Map คำสั่งประเภท R-Type

หน้าที่และขนาดของแต่ละฟิลด์ ได้แก่

- Opcode ขนาด 6 บิต ระบุว่าดำเนินการอะไร ในตัวอย่างนี้ทำการบวก
- rs ขนาด 5 บิต ให้ระบุว่าแหล่งข้อมูลสำหรับโอเปอเรนด์ด้านซ้ายของ ALU มาจากรีจิสเตอร์ตัวใด ในที่นี้ คือ \$s1
- rt ขนาด 5 บิต ให้ระบุว่าแหล่งข้อมูลสำหรับโอเปอเรนด์ด้านขวาของ ALU มาจากรีจิสเตอร์ตัวใด ในที่นี้ คือ \$s2
- rd ขนาด 5 บิต ให้ระบุว่าโอเปอเรนด์ที่เก็บค่าผลลัพธ์หลังจากการกระทำ ALU แล้วเก็บไปยังรีจิสเตอร์ตัวใด ในที่นี้ คือ \$t1
- sa ขนาด 5 บิต จะใช้บอกจำนวนบิตที่จะ Shift ให้สำหรับคำสั่ง Shift ไม่ได้ใช้ในคำสั่งอื่นๆ เช่น คำสั่ง add

- funct ขนาด 6 บิต จะใช้บอกรหัสที่เพิ่มเติมจาก Opcode ในตัวอย่างนี้ใช้ร่วมกับ Opcode

ดังนั้น ในวงจรของการถอดรหัสคำสั่งแบบนี้จะอ่านข้อมูลตามบิตข้างต้น และส่งสัญญาณไปควบคุม ALU และการให้ผลของข้อมูลตามจุดเชื่อมต่อต่างๆ ให้กระทำการตาม Opcode ที่ระบุนั้น

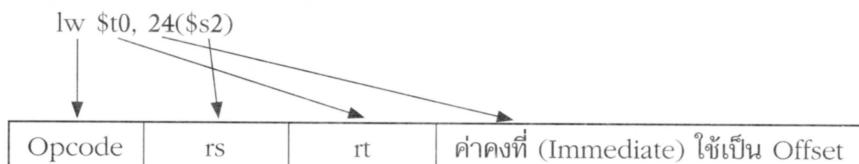
### 3.4.2.2 ประเภท I-Type

ในรูปแบบของ I-Type จะใช้กับคำสั่งหลายประเภท ได้แก่ Load/Store คำสั่ง ALU และ Immediate และคำสั่ง Branch

ต่อไปเป็นตัวอย่างของคำสั่ง Load/Store ซึ่งใช้ I หรือ s นำหน้า ส่วนตัว W ตัวหลังจะย่อมาจาก Word

```
lw $t0, 24($s3) #load word จากหน่วยความจำ
sw $t0, 8($s3)  #store word ไปยังหน่วยความจำ
```

ตำแหน่งของข้อมูลจะได้จาก Base Address + Offset ในที่นี้ Base Address เป็น 24 สำหรับคำสั่งแรก และ Offset เป็นค่าในรีจิสเตอร์ \$s3 ซึ่งเป็นการใช้ Base Addressing Mode นั่นเอง สังเกตว่ารูปแบบนี้เป็นแบบ I-Type ซึ่งค่า Base จะปรากฏอยู่ในพิล็อก Immediate ที่มีขนาด 16 บิต การที่ใช้ 16 บิตจะหมายถึงการอ้างอิงหน่วยความจำได้ระหว่าง  $\pm 2^{13}$  หรือ 8,192 Word ตัวอย่างคำสั่ง load แบ่งเป็นพิล็อกได้ดังรูปที่ 3.12

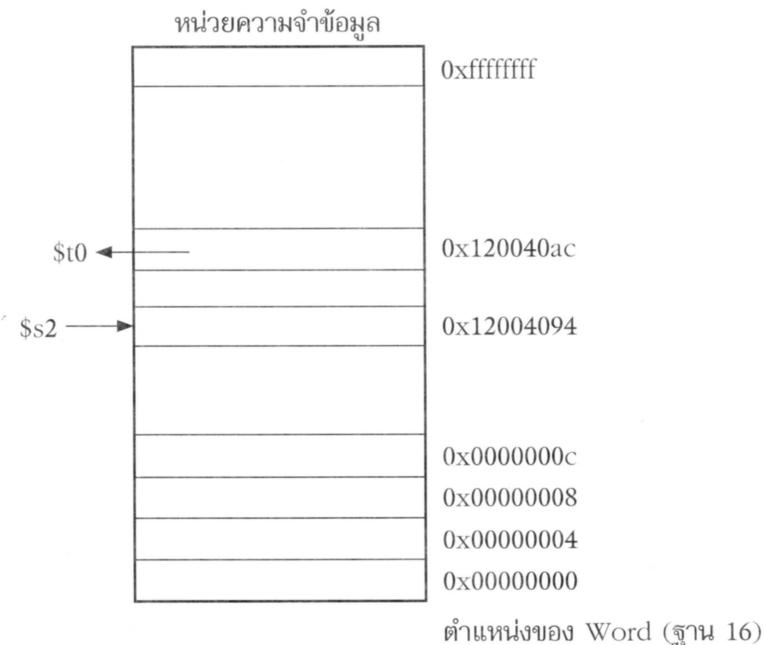


รูปที่ 3.12 ตัวอย่างคำสั่ง I-Type

จากตัวอย่างจะคำนวณค่าตำแหน่งดังนี้

$$\begin{array}{r}
 24_{10} + \$s2 = \dots 0001\ 1000 \\
 + \underline{\dots\ 1001\ 0100} \\
 \hline
 \dots \underline{1010\ 1100} = 0x120040AC
 \end{array}$$

จะหมายถึงการเข้าถึงดังแสดงในรูปที่ 3.13



รูปที่ 3.13 การเข้าถึงหน่วยความจำ

การเรียงไบต์ในหน่วยความจำจะถูกกำหนดโดยรูปแบบ Big Endian สำหรับ MIPS คำสั่งรูปแบบนี้ยังจะสามารถอ่านไบต์หรือจัดเก็บไบต์ได้ด้วยคำสั่ง Load หรือ Store ดังเช่น

```
lb $t1, 1($s3) #load byte จากหน่วยความจำ
sb $t1, 6($s3) #store byte ไปยังหน่วยความจำ
```

ซึ่งจะทำการอ่านเฉพาะไบต์สำคัญที่สุด 8 ไบต์ (Most Significant Byte) จากหน่วยความจำขึ้นมา หรือเปลี่ยนเฉพาะ 8 ไบต์สำคัญไปในหน่วยความจำ สำหรับคำสั่ง Load จะอ่านไบต์เข้าสู่รีจิสเตอร์ และจะขยายบิตให้เต็ม 32 บิต

สำหรับตัวอย่างต่อไปเป็นตัวอย่างคำสั่ง Branch ที่ใช้รูปแบบ I-Type เข่นกัน

```
bne $s2, $s1, Lbl #ไปยัง Lbl ถ้า $s2 ≠ $s1
```

เป็นการกระโดดไปยังตำแหน่งระบุในลาเบล Lbl ถ้าค่าใน s2 ไม่เท่ากับค่าใน s1 และคำสั่ง

```
beq $s2, $s1, Lbl #ไปยัง Lbl ถ้า $s2 = $s1,
```

เป็นการกระโดดไปยังตำแหน่งระบุในลาเบล Lbl ถ้าค่าใน s2 เท่ากับค่าใน s1 ถ้าเป็นคำสั่งในภาษา C

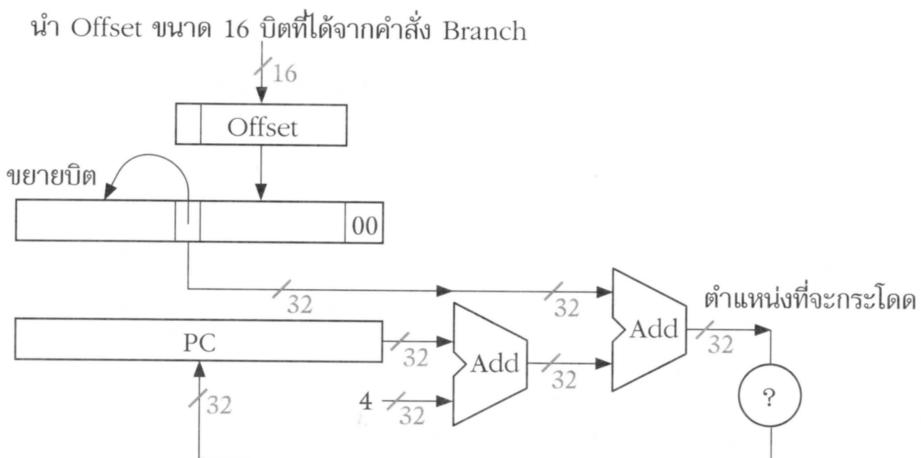
```
if (i=j) h = i + j;
```

ก็จะหมายถึงโค้ด เข่น

```
bne $s0, $s2, Lbl1
add $s3, $s0, $s2
Lbl1: ...
```

คำสั่ง bne ก็ยังใช้รูปแบบ I-Type เหมือนกัน โดย rs จะได้แก่ \$s0, rt จะหมายถึง s2 และ Lbl1 จะเป็นลาเบลที่เก็บตำแหน่งสัมพัทธ์มีขนาด 16 บิต (16-bit Offset) ตามรูปแบบในรูปที่ 3.10

ในคำสั่ง Branch นี้จะเป็นแบบสัมพัทธ์ (Relative) โดยในการหาตำแหน่ง จะนำค่า PC + 4 มาบวกกับค่า Offset ขนาด 16 บิตนี้ ตั้งแสดงในรูปที่ 3.14 และจะทำการเติมคูณ 2 ตัวข้างหลัง ทำให้เป็น 18 บิต และทำการคัดลอกบิต Sign มาเติมข้างหน้าโดยการขยายบิต ทำให้เป็น 32 บิต การใช้ Offset ขนาด 16 บิตนี้จะสามารถกระโดดไปในช่วงตำแหน่ง  $-2^{15}$  ถึง  $+2^{15} - 1$  นับจากตำแหน่ง PC + 4



รูปที่ 3.14 การคำนวณตำแหน่งของคำสั่ง Branch

นอกจากนี้ ยังมีคำสั่งลักษณะเดียวกับการกระໂດด เน่น Set Less Than

```
slt $t0, $s0, $s2 # ถ้า $s0 < $s2 แล้ว
# $t0 = 1 มีเข็มสี
# $t0 = 0
```

แต่จะใช้รูปแบบ R-Type จากนั้นจะสามารถใช้การเปรียบเทียบค่า 0 ได้ในคำสั่ง Branch เช่น

```
slt $at, $s1, $s2 #$at เป็น 1 ถ้า $s1 < $s2
bne $at, $zero, Label
หรือจะใช้คำสั่ง
blt $s0, $s2, Label
```

เป็นการกระໂດถ้า \$s0 < \$s2 ในรูปแบบ I-Type ซึ่งเทียบเท่ากับ 2 คำสั่งนี้

```
slt $at, $s0, $s2      #$at เป็น 1 ถ้า
bne $at, $zero, Label #$s0 < $s2
```

สำหรับการกระໂດยังมีเงื่อนไขแบบอื่นๆ อีก เช่น การเปรียบเทียบน้อยกว่า มากกว่า มากกว่าเท่ากับ ดังนี้

```
ble $s0, $s2, Label; เมื่อ $s0 <= $s2
bgt $s0, $s2, Label; เมื่อ $s0 > $s2
bge $s0, $s2, Label; เมื่อ $s0 >= $s2
```

การใช้คำนวนของ ALU ที่ทำกับค่าคงที่ก็ใช้ I-Type เช่นกัน ในพิลด์ Immediate มีขนาด 16 บิตซึ่งทำให้ระบุค่าคงที่ได้ในช่วงของเลข 16 บิตเท่านั้น หรือ  $+2^{15} - 1$  ถึง  $-2^{15}$

ตัวอย่างเช่น คำสั่งต่อไปนี้

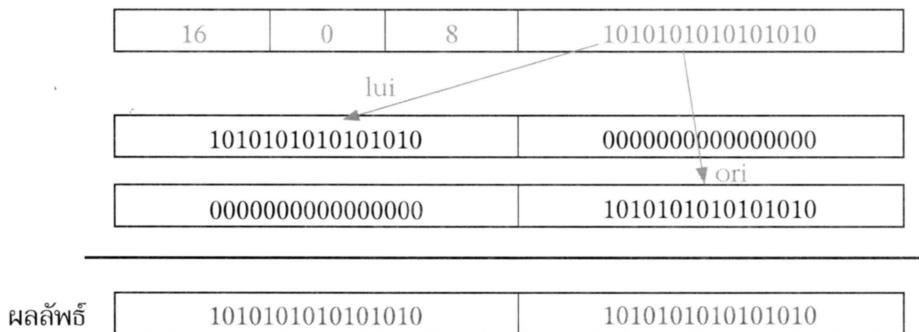
```
addi $sp, $sp, 8      #$sp = $sp + 8
slti $t0, $s1, 15     #$t0 = 1 ถ้า $s1 < 15
```

เป็นคำสั่งที่ใช้ค่าคงที่ขนาดเล็ก

แต่หากต้องการค่าคงที่ขนาดใหญ่ขึ้น ก็จะต้องใช้วิธีการอื่นช่วย เช่น

```
lui $t1, 1010101010101010
ori $t1, $t1, 1010101010101010
```

ดังการคำนวณดังนี้

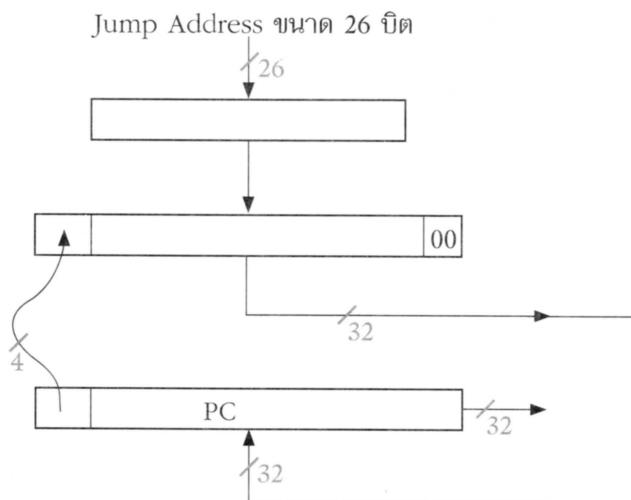


### 3.4.2.3 คำสั่ง J-Type

สำหรับคำสั่ง Jump แบบอื่นๆ ใช้รูปแบบ J-Type ได้แก่

```
j label      #ไปยัง label
```

เป็นการกระโดดแบบไม่มีเงื่อนไข ใช้รูปแบบ J-Type ในรูปที่ 3.10 โดย Jump Address มีขนาด 26 บิต ตำแหน่งขนาด 26 บิตนี้จะถูกนำมาเพิ่ม 0 ข้างหลังอีก 2 ตัว และนำ 4 บิตบนของค่าใน PC มาต่อข้างหน้า ทำให้กลายเป็น 32 บิต ได้ตำแหน่งสำหรับเก็บใน PC คำสั่งกระโดดแบบนี้เป็นแบบใช้แอดเดรสจ共和 หรือเรียกว่า Absolute Address ดังแสดงในรูปที่ 3.15 ซึ่งจะทำให้กระโดดไปยังคำสั่งที่อยู่ไกลได้มากกว่าการกระโดดแบบสัมพัทธ์



รูปที่ 3.15 การนำทำແໜ່ງ Jump Address จากคำสั่ง Jump มาคำนวน

คำสั่งแบบ J-Type นี้จะมีประโยชน์สำหรับการกระโดดแบบมีเงื่อนไขที่ต้องการไปทำແໜ່ງที่ไกลกว่า 16 บิต จะแทนค่าได้ เช่น

`beq $s2, $s1, L1` ถ้า L1 เป็นทำແໜ່ງที่ไกลเกินกว่า 16 บิต

จะต้องเปลี่ยนคำสั่งเป็น

`bne $s2, $s1, L2`

`j L1`

`L2: ...`

โดยใช้ L2 ที่ใกล้กว่ามาเป็นทำແໜ່ງชั่วคราวก่อน และค่อยกระโดดไป L1 ด้วยคำสั่ง j อีกที นอกจากนี้ คำสั่งเกี่ยวกับการเรียกโปรแกรมย่อยก็ใช้ J-Type

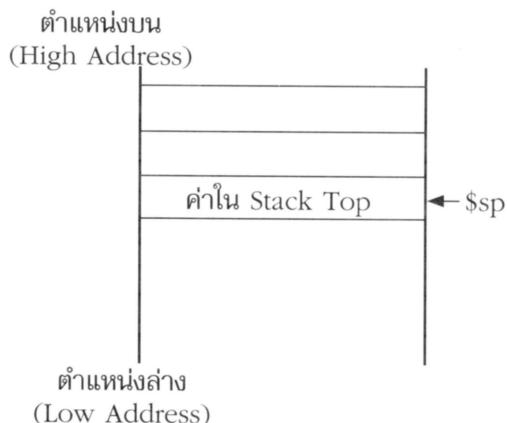
`jal ProcedureAddress #jump and link` เป็นการเรียกโปรแกรมย่อย กำหนดโดยทำແໜ່ງ

ส่วนการ Return จากโปรแกรมย่อยเป็นคำสั่งแบบ R-Type

`jr $ra #return` จากโปรแกรมย่อยไปยังทำແໜ່ງเดิม

ระบุคำสั่งโดยใช้ฟิลด์ Opcode และ funct ในพื้นที่เป็น Jr และโอเปอเรนด์มีเพียงฟิลด์ rs แทนค่าใน \$ra และฟิลด์อื่นไม่ได้ใช้

ในการเรียกโปรแกรมย่อยจะต้องใช้สแต็กประกอบ ดังแสดงในรูปที่ 3.16



รูปที่ 3.16 การเรียงตำแหน่งในสแต็ก

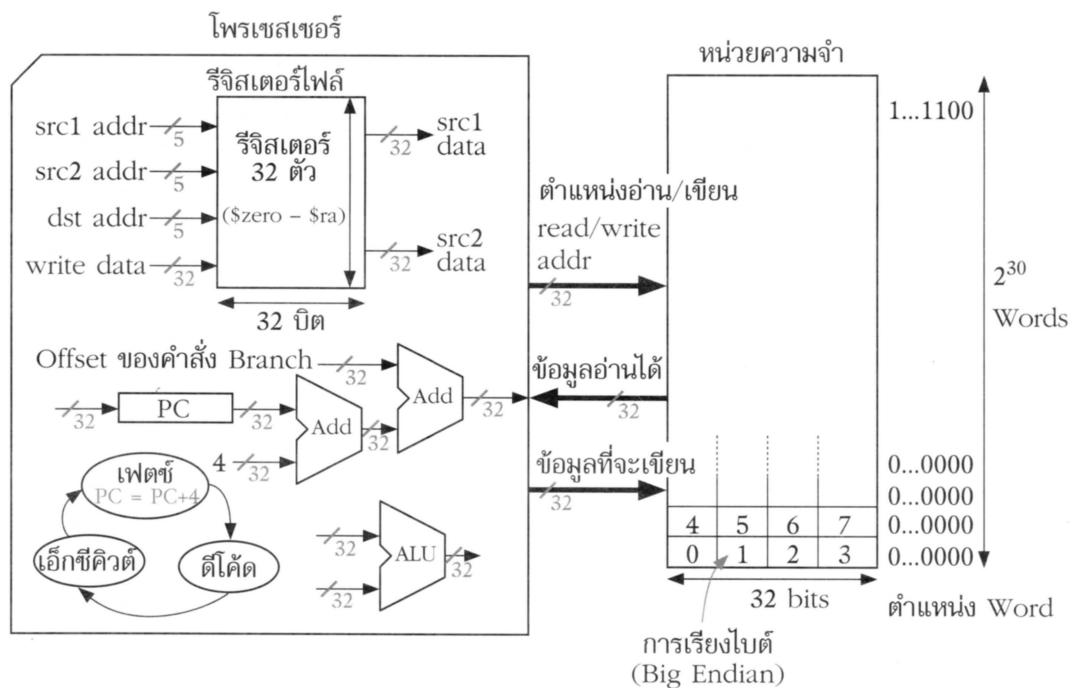
สแต็กต้องมีจิสเตอร์เก็บค่าตำแหน่งสูงสุดของสแต็ก (Stack Top) โดยใน MIPS จะเก็บในรีจิสเตอร์พิเศษชื่อ  $\$sp$  ในการใส่ข้อมูลในสแต็กด้วยการ Push โดยจะต้องลดค่าตำแหน่งที่เก็บใน  $\$sp$

$$\$sp = \$sp - 4$$

และการเอาข้อมูลออกจากสแต็กด้วยการ Pop ค่าออกจากการ Pop โดยจะเพิ่มค่าใน  $\$sp$

$$\$sp = \$sp + 4$$

รูปที่ 3.17 แสดงโครงสร้างของ MIPS โดยรวมเมื่อนำรูปที่ 3.2 และรูปที่ 3.7 มาประกอบกันในโครงสร้างของ MIPS รีจิสเตอร์ไฟล์ และการคำนวณรูปตามรูปแบบคำสั่งต่างๆ และการอ่านข้อมูลเข้าและออกหน่วยความจำ



รูปที่ 3.17 โครงสร้างของ MIPS และองค์ประกอบ

ตารางด้านล่างนี้สรุปคำสั่งต่างๆ ของ MIPS

ประเภท	คำสั่ง	รหัส Opcode	ตัวอย่าง	ความหมาย
Arithmetic (R & I-Type)	add	0 และ 32	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$
	subtract	0 และ 34	sub \$s4, \$s2, \$s3	$\$s4 = \$s2 - \$s3$
	add immediate	8	addi \$s1, \$s2, 7	$\$s1 = \$s2 + 7$
	or immediate	13	ori \$s1, \$s2, 7	$\$s1 = \$s2 \vee 7$

(ต่อ)

ประเภท	คำสั่ง	รหัส Opcode	ด้วย	ความหมาย
Data Transfer (I-Type)	load word	35	lw \$s1, 25(\$s2)	\$s1 = Memory(\$s2+25)
	store word	43	sw \$s1, 25(\$s2)	Memory(\$s2+25) = \$s1
	load byte	32	lb \$s1, 26(\$s2)	\$s1 = Memory(\$s2+26)
	store byte	40	sb \$s1, 26(\$s2)	Memory(\$s2+26) = \$s1
	load upper immediate	15	lui \$s1, 7	\$s1 = $7 \times 2^{16}$
Cond. Branch (I & R-Type)	br on equal	4	beq \$s1, \$s2, L1	if (\$s1 == \$s2) ไปยัง L1
	br on not equal	5	bne \$s1, \$s2, L1	if (\$s1 != \$s2) ไปยัง L1
	set on less than	0 และ 42	slt \$s1, \$s4, \$s3	if (\$s4 < \$s3) \$s1=1 else \$s1=0
	set on less than immediate	10	slti \$s1, \$s4, 6	if (\$s4 < 6) \$s1=1 else \$s1=0
Uncond. Jump (J & R-Type)	jump	2	j 2500	ไปยังตำแหน่ง 10000
	jump register	0 และ 8	jr \$t1	ไปยังตำแหน่ง \$t1
	jump and link	3	jal 2500	go to 10000; \$ra = PC + 4

### ■ 3.5 การพิจารณาประสิทธิภาพของชีพีย์เชิงปริมาณในส่วนของชุดคำสั่ง

พิจารณาตัวอย่างการวัดประสิทธิภาพของชีพีย์ และผลกระทบลักษณะชุดคำสั่งต่อประสิทธิภาพ สมมติว่าสำหรับการออกแบบชุดคำสั่ง มี 2 ทางเลือกสำหรับกรณีคำสั่งแบบ Branch ดังนี้

- ชีพีย์ A ใช้ลักษณะของ Condition Code หรือใช้รีจิสเตอร์ Flag ซึ่งเป็นรีจิสเตอร์พิเศษ อันเดียว ให้เก็บสถานะของการทำงานหลังจากคำสั่งปั๊กบัน ถูกเขตโดยคำสั่งเบรียบเทียบ ก่อนหน้าคำสั่ง Branch ตัวอย่างการเขียนโปรแกรมแบบนี้คือ

CMP X, Y

BEZ L1

หลังจากเบรียบเทียบด้วยคำสั่ง CMP และรีจิสเตอร์ Flag จะถูกเขตโดยย่างเหมาสม เช่น Flag Zero อาจจะถูกเขตเมื่อเบรียบเทียบแล้วทั้ง X, Y เท่ากัน จากนั้นใช้คำสั่ง BEZ ที่เป็นคำสั่ง Branch แบบมีเงื่อนไขไปยังตำแหน่ง L1 เมื่อมีการเขตค่าในรีจิสเตอร์ Flag

- ชีพีย์ B จะใช้คำสั่ง Branch แบบมีเงื่อนไขที่รวมการเบรียบเทียบแล้วในคำสั่งเดียวกัน เช่น

BEZ X, Y, L1

ในคำสั่งนี้ เบรียบเทียบ X เท่ากับ Y หรือไม่ ถ้าใช่ จะไปทำงานที่ตำแหน่ง L1

สำหรับชีพีย์ทั้งสองกำหนดให้คำสั่ง Branch แบบมีเงื่อนไขใช้ 2 ไบเกิล และคำสั่งอื่นๆ ใช้ 1 ไบเกิล ในชีพีย์ A 25% ของคำสั่งทั้งหมดเป็นคำสั่ง Branch แบบมีเงื่อนไข ดังนั้น 25% ของคำสั่งทั้งหมดก็จะเป็นคำสั่งการเบรียบเทียบด้วย เนื่องจากชีพีย์ A ไม่มีการเบรียบรวมอยู่ในคำสั่ง Branch ทำให้คำสั่งของชีพีย์ A อยู่ในรูปแบบที่ง่ายกว่าในแต่ละคำสั่ง ทำให้ Clock Cycle Time ของชีพีย์ A เร็วกว่าของชีพีย์ B อยู่ 25% พิจารณาสมการ CPU Time อย่างทราบว่าชีพีย์ ได้ทำงานเร็วกว่ากัน

ก่อนอื่นจะต้องคำนวณหา CPI ของชีพีย์ A ซึ่งเป็น  $CPI_A = (0.25*2) + (0.75*1) = 1.25$  เพราะว่าคำสั่ง Branch แบบมีเงื่อนไขมีอยู่ 25% และคำสั่งนี้ใช้ 2 ไบเกิล

เราทราบว่า Clock Cycle Time ของชีพียู B คือ  $1.25 \times$ Clock Cycle Time ของชีพียู A เพราะว่าชีพียู A มี Cycle Time ที่เร็วกว่าของชีพียู B อยู่ 25% ดังนั้น

$$\begin{aligned} \text{CPU Time}_A &= \text{Instruction Count}_A \times 1.25 \times \text{Clock Cycle Time}_A \\ &= 1.25 \times \text{Instruction Count}_A \times \text{Clock Cycle Time}_A \end{aligned}$$

เนื่องจากชีพียู B ไม่ใช่คำสั่งการเบรี่ยบเทียบ ดังนั้น  $25\% / 75\% = 33\%$  ของคำสั่งเป็นคำสั่ง Conditional Branch ที่ใช้ 2 ไบเกิล และทำให้ 67% ของคำสั่งทั้งหมดใช้คำสั่งละ 1 ไบเกิล ดังนั้น CPI ของชีพียู B คือ  $\text{CPI}_B = (0.33 \times 2) + (0.67 \times 1) = 1.33$  และทำให้ N หรือจำนวนคำสั่งทั้งหมดในโปรแกรมลดไปเป็น  $0.75 \times \text{Instruction Count}_A$

$$\begin{aligned} \text{CPU Time}_B &= (0.75 \times \text{Instruction Count}_A) \times 1.33 \times (1.25 \times \text{Clock Cycle Time}_A) \\ &= 1.25 \times \text{Instruction Count}_A \times \text{Clock Cycle Time}_A \end{aligned}$$

ได้ว่า

$$\begin{aligned} \text{CPU Time}_A &= \text{Instruction Count}_A \times 1.25 \times \text{Clock Cycle Time}_A \\ &= 1.25 \times \text{Instruction Count}_A \times \text{Clock Cycle Time}_A \end{aligned}$$

ทำให้ทราบว่าโปรแกรมนี้ทำงานบนชีพียู A ใช้ CPU Time เท่ากับเมื่อทำงานบนชีพียู B

จากตัวอย่างข้างต้น ถ้านักสถาปัตยกรรมต้องการแก้ไขการจัดองค์ประกอบทางฮาร์ดแวร์ที่ทำให้ Clock Cycle Time โดยรวมของชีพียู B ตื้นขึ้น โดยกล่าวเป็นว่า Clock Cycle Time ของชีพียู A เร็วกว่าของชีพียู B อยู่ 10% ในกรณีนี้ชีพียูได้เร็วกว่ากัน

ถ้าเปลี่ยน Clock Cycle Time ให้เร็วขึ้น 15% จะได้ว่า Clock Cycle Time ของชีพียู B คือ 1.15 เท่าของ Clock Cycle Time ของ A ในขณะที่ CPU Time ของโปรแกรมนี้บนชีพียู A ยังเหมือนเดิม

$$\begin{aligned} \text{CPU Time}_A &= \text{Instruction Count}_A \times 1.25 \times \text{Clock Cycle Time}_A \\ &= 1.25 \times \text{Instruction Count}_A \times \text{Clock Cycle Time}_A \end{aligned}$$

ได้ว่า

$$\begin{aligned} \text{CPU Time}_B &= (0.75 * \text{Instruction Count}_A) * 1.33 * (1.15 * \text{Clock Cycle Time}_A) \\ &= 1.15 * \text{Instruction Count}_A * \text{Clock Cycle Time}_A \end{aligned}$$

ทำให้สรุปได้ว่าชีพีย์ B ทำงานได้เร็วกว่าชีพีย์ A ในขณะนี้

พิจารณาอีกรอบหนึ่งคือ ถ้าจะเพิ่มการเปลี่ยนแปลงเข้าไปอีกเกี่ยวกับชุดคำสั่ง ถ้าในชุดคำสั่งแบบแรก ทุกคำสั่งจะทำงานได้กับรีจิสเตอร์เท่านั้น โดยที่มีคำสั่ง Load/Store เพื่อถ่ายและรับค่าจากหน่วยความจำให้เท่านั้น

พิจารณา Instruction Mix ที่ประกอบด้วยคำสั่งประเภท ALU คำสั่ง Load/Store และคำสั่งเกี่ยวกับ Branch ดังนี้

ประเภทคำสั่ง	ความถี่ที่ใช้ในโปรแกรม	จำนวนไบเกิลที่ใช้
ALU Op	43%	1
Load	21%	2
Store	13%	2
Branch	23%	2

ในคอลัมน์ความถี่ที่ใช้แสดงสัดส่วนของคำสั่งประเภทนั้นๆ ต่อจำนวนคำสั่งทั้งหมดในโปรแกรม โดยคิดเป็นเปอร์เซ็นต์ ส่วนคอลัมน์จำนวนรอบที่ใช้แสดงจำนวนไบเกิลที่ใช้ในการทำงานคำสั่งประเภทนั้นๆ ถ้าพบว่าในโปรแกรมนี้ 26% ของ ALU Ops เป็นคำสั่งที่ไม่โอเปอเรนด์ที่โหลดเข้ามาแล้วจะไม่ได้ใช้อีก

ถ้ามีการเพิ่มคำสั่งประเภทของ ALU Ops โดยที่โอเปอเรนด์เป็นแบบ Register-Memory และมีจำนวนไบเกิลเท่ากับ 2 ถ้าใช้ชุดคำสั่งใหม่นี้ จะทำให้เพิ่มจำนวนไบเกิลที่ใช้ของคำสั่งประเภท Branch ไป 1 ไบเกิล และไม่มีผลกับ Clock Cycle Time ของระบบ ตามว่าการเปลี่ยนแปลงนี้จะเพิ่มประสิทธิภาพการทำงานของชีพีย์หรือไม่

หา CPI ของชุดคำสั่งเดิมก่อน

$$\text{CPI}_{\text{old}} = 0.43*1 + 0.21*1 + 0.13*2 + 0.23*2 = 1.36$$

ดังนั้น

$$\begin{aligned}\text{CPU Time}_{\text{old}} &= \text{Instruction Count}_{\text{old}} * 1.36 * \text{Clock Cycle Time}_{\text{old}} \\ &= 1.36 * \text{Introduction Count}_{\text{old}} * \text{Clock Cycle Time}_{\text{old}}\end{aligned}$$

และหา

$$\begin{aligned}\text{CPI}_{\text{new}} &= \frac{(0.43 - (0.26*0.43))*1 + (0.21 - (0.26*0.43))*2 + 0.12*2 + 0.24*3 + (0.26*0.43)*2}{1 - (0.26*0.43)} \\ &= 1.91\end{aligned}$$

ทำให้จำนวนคำสั่งทั้งหมดลดไป  $0.26*0.43$  ส่วน เพราะคำสั่งทาง ALU Ops ลดไป 26% และคำสั่ง Load ลดไปประมาณ 26% เข่นกัน และเป็นการเพิ่มคำสั่งแบบใหม่ 26%

และ Clock Cycle Time เพิ่มเติม ดังนี้

$$\begin{aligned}\text{CPU Time}_{\text{new}} &= 0.89 * \text{Instruction Count}_{\text{old}} * 1.91 * \text{Clock Cycle Time}_{\text{old}} \\ &= 1.7 * \text{Instruction Count}_{\text{old}} * \text{Clock Cycle Time}_{\text{old}}\end{aligned}$$

ทำให้เห็นว่าชุดคำสั่งแบบใหม่ไม่ทำให้ประสิทธิภาพดีขึ้น

## ■ 3.6 สรุป

ในการออกแบบสถาปัตยกรรมชุดคำสั่ง (Instruction Set Architecture) นั้น มีความสำคัญต่อประสิทธิภาพของเครื่องคอมพิวเตอร์ โดยเฉพาะในส่วนของ CPI และ Instruction Count และยังสัมพันธ์ไปถึงโครงสร้างชาร์ดแวร์ทั้ง Data Path และหน่วยควบคุม บทนี้ทบทวนถึงพื้นฐานเกี่ยวกับรูปแบบชุดคำสั่งที่มีส่วนประกอบของคำสั่งหนึ่งๆ จำนวนพอดีของโอเปอเรนเตอร์ รูปแบบของโอเปอเรนเตอร์ รวมทั้งวิธีการเข้าถึงใน Addressing Mode และผลกระทบของการออกแบบ Data Path และ Control นอกจากนี้ ยังกล่าวถึงพื้นฐานทางการคำนวณเกี่ยวกับระบบเลขฐาน การเรียงไบต์ และเลขทศนิยม

บทนี้ได้แสดงตัวอย่างของชุดคำสั่งแบบ MIPS ที่เป็นแบบ RISC ลักษณะต่างๆ ของชุดคำสั่งนี้ ตัวอย่างของคำสั่งเบ่งตามรูปแบบ และโครงสร้างองค์ประกอบของ Data Path ตามรูปแบบของ MIPS ที่เกี่ยวข้องกับชุดคำสั่งนี้

## คำถ้ามก้าวยบท

1. ไอเวอร์ไฟล์ของเลขจำนวนเต็มขนาด 8 บิตจะตรวจสอบได้อย่างไร
2. ใน Addressing Mode จำนวนไอเพอแรนด์และความยาวคำสั่งมีผลต่อรอบสัญญาณนาฬิกาอย่างไร
3. ชุดคำสั่งแบบ MIPS ที่เป็นแบบ RISC เป็นอย่างไร
4. Addressing Mode ใน MIPS มีกี่แบบ
5. จำนวนไอเพอแรนด์มีผลต่อความยาวคำสั่งอย่างไร
6. ถ้า Mantissa ขนาด 4 บิต และ Exponent ขนาด 4 บิต ถ้าใช้รูปแบบ 2's Complement สำหรับส่วน Mantissa และ Exponent จะได้ตัวเลขบางที่ใหญ่ที่สุดและที่เล็กที่สุดเท่าไร ตัวเลขลบที่ใหญ่ที่สุดและเล็กที่สุดที่เก็บได้คืออะไร
7. ใน MIPS ถ้าต้องการบวกด้วยค่าคงที่มากกว่า  $2^{16}$  จะต้องเขียนโค้ดอย่างไร
8. รูปแบบคำสั่ง รวมทั้ง Addressing Mode จำนวนไอเพอแรนด์มีผลต่อการถอดรหัสอย่างไร
9. ใน MIPS ถ้าต้องการกระโดยไปกลบกว่า  $2^{16}$  จะต้องเขียนโค้ดอย่างไร
10. จงแทนค่าเลข 12 ด้วยวิธี IEEE-754 แบบ Single Precision
11. การเรียงไบต์แบบ Little Endian และ Big Endian มีความสำคัญอย่างไร MIPS ใช้การเรียงไบต์แบบใด
12. จงเขียนโปรแกรมในรูปแบบ MIPS Assembly ให้ทำงานเทียบกับโค้ดภาษา C ต่อไปนี้
 

```
while (x < 10) {
    y += x*10*x-- + q;
    q = q+10;
}
```

กำหนดให้  $x, y, q$  อยู่ในหน่วยความจำตำแหน่ง 2000, 2004, 2008 ตามลำดับ โดยตั้งกล่าว  
ใช้คำสั่งรูปแบบอะไรบ้าง ขนาดโดยต้องทั้งหมดเท่ากับเท่าไร

13. จงเขียนโปรแกรมในรูปแบบ MIPS Assembly ให้ทำงานเทียบกับโค้ดภาษา C ต่อไปนี้

$$A = B * C + D * E$$

กำหนดให้  $A, B, C, D, E$  อยู่ในหน่วยความจำตำแหน่งตั้งแต่ 2000 เป็นต้นไป โดยตั้งกล่าว  
ใช้คำสั่งรูปแบบอะไรบ้าง ขนาดโดยต้องทั้งหมดเท่ากับเท่าไร

14. จงเขียนโปรแกรมในรูปแบบ MIPS Assembly ที่เรียนมาให้ทำงานเทียบกับโค้ดภาษา C ต่อ  
ไปนี้

```
while (i < 100) {
    A[i] = A[i] + 4;
    i++;
}
```

กำหนดให้อาร์เรย์  $A$  อยู่ในหน่วยความจำตำแหน่งเริ่มต้นที่ 1000 และ  $i$  อยู่ที่ตำแหน่ง 4000  
ตามลำดับ โดยตั้งกล่าวใช้คำสั่งรูปแบบอะไรบ้าง ขนาดโดยต้องทั้งหมดเท่ากับเท่าไร

15. พิจารณาโค้ดต่อไปนี้

$$A = A + B * C$$

จงเขียนโค้ดในรูปแบบต่อไปนี้ สมมติว่ารีจิสเตอร์เป็น  $R0, R1, R2, \dots$

- zero operand
- one operand
- two operands

กำหนดให้ Opcode มีขนาด 4 บิต และโอเปอเรนเดอร์แบบรีจิสเตอร์มีขนาด 8 บิต Memory Address มีขนาด 16 บิต จงบอกขนาดของที่ได้ในแต่รูปแบบในจำนวนของบิต

16. จงเขียนโปรแกรม Quick Sort, Selection Sort, Insertion Sort เพื่อเรียงลำดับข้อมูล [3, 4, 1, 2, 5] เปรียบเทียบจำนวนคำสั่งที่ใช้ในการรันโดยใช้ MIPS และ x86 และจากโปรแกรมนี้ จงคอมpileด้วย gcc และวัดขนาดของโค้ด และนับจำนวนคำสั่งที่ใช้ทำงานทั้งหมด คำนวณ CPU Time จากสูตร และลองเบรียบเทียบเวลา กันเวลาจริงที่จับเวลาได้
17. พิจารณาตัวอย่างโปรแกรมต่อไปนี้ในภาษา C

```
for (i = 0; i < 10; i++)
    A[i] = B[i] + 20;
```

กำหนดให้อาร์เรย์ A อยู่ที่แอดเดรสเริ่มต้น 1000 และอาร์เรย์ B อยู่ที่แอดเดรสเริ่มต้น 2000  
กำหนดให้ตัวแปร i อยู่ที่แอดเดรส 100

17.1 จงแปลงโค้ดดังกล่าวเป็นรูปแบบ Immediate Code ในรูปแบบ Operational Semantic

17.2 จงแปลงโค้ดดังกล่าวเป็นรูปแบบ MIPS Instruction

