**Arithmetic and Logic Unit (ALU)**

หน่วยคำนวณและตรรกะ (Arithmetic & Logic Unit: ALU) คือส่วนประกอบสำคัญของหน่วยประมวลผลกลาง (CPU) ในคอมพิวเตอร์ ทำหน้าที่ประมวลผลข้อมูลทางคณิตศาสตร์และตรรกะ

**หน้าที่หลักของ ALU**

* **การคำนวณทางคณิตศาสตร์**: ALU ทำหน้าที่คำนวณพื้นฐาน เช่น การบวก ลบ คูณ หาร รวมถึงการคำนวณที่ซับซ้อนกว่า เช่น การยกกำลัง การหาราก และการคำนวณทางตรีโกณมิติ
* **การดำเนินการทางตรรกะ**: ALU ทำหน้าที่เปรียบเทียบข้อมูล เช่น การเปรียบเทียบว่าข้อมูลใดมากกว่า น้อยกว่า หรือเท่ากัน รวมถึงการดำเนินการทางตรรกะ เช่น AND, OR, NOT, XOR
* **การจัดการข้อมูล**: ALU สามารถใช้ในการจัดการข้อมูล เช่น การเลื่อนบิต (shift) การหมุนบิต (rotate) และการแปลงรูปแบบข้อมูล

**ส่วนประกอบของ ALU**

ALU ประกอบด้วยวงจรไฟฟ้าที่ซับซ้อน ซึ่งทำงานร่วมกันเพื่อประมวลผลข้อมูล วงจรเหล่านี้รวมถึง

* **วงจรคำนวณ**: ทำหน้าที่คำนวณทางคณิตศาสตร์
* **วงจรตรรกะ**: ทำหน้าที่ดำเนินการทางตรรกะ
* **รีจิสเตอร์**: ใช้เก็บข้อมูลที่ใช้ในการประมวลผล และผลลัพธ์ของการประมวลผล

**การทำงานของ ALU**

ALU ทำงานร่วมกับหน่วยควบคุม (Control Unit: CU) ซึ่งเป็นส่วนประกอบอีกส่วนหนึ่งของ CPU CU จะส่งคำสั่งให้ ALU ทำการประมวลผลข้อมูล โดยข้อมูลที่ใช้ในการประมวลผลจะถูกส่งมาจากหน่วยความจำหลัก (RAM) หรือจากรีจิสเตอร์ภายใน CPU เมื่อ ALU ประมวลผลข้อมูลเสร็จสิ้นแล้ว ผลลัพธ์จะถูกส่งกลับไปยังหน่วยความจำหลัก หรือไปยังรีจิสเตอร์ เพื่อนำไปใช้ในการประมวลผลต่อไป

**ความสำคัญของ ALU**

ALU มีความสำคัญอย่างยิ่งต่อการทำงานของคอมพิวเตอร์ เนื่องจากเป็นส่วนประกอบที่ทำหน้าที่ประมวลผลข้อมูลทั้งหมดของคอมพิวเตอร์ หากไม่มี ALU คอมพิวเตอร์จะไม่สามารถทำงานได้

**Integer Representation**

การแทนจำนวนเต็มในคอมพิวเตอร์เป็นเรื่องสำคัญเพื่อให้คอมพิวเตอร์สามารถจัดเก็บและประมวลผลตัวเลขได้อย่างมีประสิทธิภาพ โดยทั่วไปมีวิธีการแทนจำนวนเต็มอยู่หลายรูปแบบ ซึ่งแต่ละรูปแบบก็มีข้อดีข้อเสียแตกต่างกันไป

**1. Unsigned Integers (จำนวนเต็มไม่มีเครื่องหมาย)**

* ใช้บิตทั้งหมดในการแทนค่าตัวเลข ทำให้สามารถแทนค่าเลขจำนวนเต็มบวกได้ตั้งแต่ 0 ถึงค่าสูงสุดที่บิตนั้นๆ จะสามารถแทนได้
* **ข้อดี:** ง่ายต่อการคำนวณและใช้งาน
* **ข้อเสีย:** ไม่สามารถแทนเลขจำนวนเต็มลบได้

**2. Signed Integers (จำนวนเต็มมีเครื่องหมาย)**

* ใช้บิตซ้ายสุด (MSB) เพื่อแทนเครื่องหมาย (+ หรือ -) โดยที่ 0 หมายถึงบวก และ 1 หมายถึงลบ
* มีหลายรูปแบบในการแทนจำนวนเต็มลบ เช่น
  + **Sign-Magnitude:** บิตที่เหลือใช้แทนขนาดของตัวเลข
  + **One's Complement:** แทนค่าลบโดยการกลับบิต (0 เป็น 1 และ 1 เป็น 0) ของค่าบวก
  + **Two's Complement:** แทนค่าลบโดยการกลับบิตของค่าบวกแล้วบวก 1 (เป็นรูปแบบที่นิยมใช้มากที่สุด)
* **ข้อดี:** สามารถแทนเลขจำนวนเต็มได้ทั้งบวกและลบ
* **ข้อเสีย:** การคำนวณอาจซับซ้อนกว่า Unsigned Integers

**3. Floating-Point Numbers (จำนวนทศนิยม)**

* ใช้รูปแบบที่ซับซ้อนกว่าในการแทนเลขทศนิยม โดยแบ่งออกเป็น 3 ส่วน:
  + **Sign bit:** แทนเครื่องหมาย
  + **Exponent:** แทนเลขชี้กำลัง
  + **Mantissa (หรือ Fraction):** แทนส่วนของตัวเลข
* **ข้อดี:** สามารถแทนเลขทศนิยมและเลขที่มีค่ามากๆ หรือน้อยมากๆ ได้
* **ข้อเสีย:** การคำนวณอาจซับซ้อนและใช้เวลานานกว่าจำนวนเต็ม และอาจมีความคลาดเคลื่อนในการคำนวณ

**4. อื่นๆ**

* นอกจากนี้ยังมีรูปแบบการแทนจำนวนเต็มอื่นๆ อีก เช่น BCD (Binary-Coded Decimal) ที่ใช้แทนเลขฐานสิบในรูปแบบเลขฐานสอง

**Sign-Magnitude Representation**

Sign-Magnitude Representation เป็นวิธีการแทนจำนวนเต็มแบบมีเครื่องหมายที่ง่ายที่สุดวิธีหนึ่ง โดยใช้บิตซ้ายสุด (Most Significant Bit: MSB) เพื่อแสดงเครื่องหมายของจำนวนเต็ม และบิตที่เหลือเพื่อแสดงขนาดของจำนวนเต็ม

**หลักการทำงาน**

* **บิตเครื่องหมาย (Sign bit):**
  + 0 หมายถึง จำนวนเต็มบวก
  + 1 หมายถึง จำนวนเต็มลบ
* **บิตขนาด (Magnitude bits):** บิตที่เหลือใช้แทนขนาดของจำนวนเต็มในรูปแบบเลขฐานสอง

**ตัวอย่าง**

สมมติว่าเราใช้เลขฐานสอง 8 บิตในการแทนจำนวนเต็ม

* **+5:** 00000101 (บิตซ้ายสุดเป็น 0 แสดงว่าเป็นจำนวนบวก บิตที่เหลือ 0000101 แทนขนาด 5)
* **-5:** 10000101 (บิตซ้ายสุดเป็น 1 แสดงว่าเป็นจำนวนลบ บิตที่เหลือ 0000101 แทนขนาด 5)

**ข้อดี**

* **เข้าใจง่าย:** เป็นวิธีที่เข้าใจง่ายและตรงไปตรงมา
* **สมมาตร:** จำนวนเต็มบวกและลบที่มีขนาดเท่ากัน จะมีรูปแบบที่สมมาตรกัน (ต่างกันแค่บิตเครื่องหมาย)

**ข้อเสีย**

* **มีศูนย์สองรูปแบบ:** มีการแทนเลขศูนย์สองแบบ คือ +0 (00000000) และ -0 (10000000) ซึ่งทำให้สิ้นเปลืองพื้นที่
* **การคำนวณซับซ้อน:** การคำนวณทางคณิตศาสตร์ด้วยวิธี Sign-Magnitude ค่อนข้างซับซ้อน ต้องแยกพิจารณาเครื่องหมายก่อนทำการคำนวณ

**Range Extension**

Range Extension หรือการขยายช่วง คือกระบวนการเพิ่มขนาดของข้อมูล (โดยทั่วไปคือจำนวนบิต) เพื่อให้สามารถแทนค่าตัวเลขได้ในช่วงที่กว้างขึ้น โดยยังคงความถูกต้องของค่าเดิมไว้

**ทำไมต้องขยายช่วง?**

* **การคำนวณ**: ในการคำนวณทางคณิตศาสตร์ ผลลัพธ์ที่ได้อาจมีขนาดใหญ่เกินกว่าที่ตัวแปรเดิมจะสามารถเก็บได้ การขยายช่วงช่วยให้สามารถเก็บผลลัพธ์ได้อย่างถูกต้อง
* **การเปรียบเทียบ**: ในการเปรียบเทียบตัวเลขที่มีขนาดต่างกัน อาจจำเป็นต้องขยายช่วงของตัวเลขที่มีขนาดเล็กกว่า เพื่อให้สามารถเปรียบเทียบได้อย่างถูกต้อง
* **การใช้งานร่วมกับระบบอื่น**: ระบบคอมพิวเตอร์บางระบบอาจใช้ขนาดข้อมูลที่แตกต่างกัน การขยายช่วงช่วยให้ข้อมูลสามารถใช้งานร่วมกันได้

**วิธีการขยายช่วง**

วิธีการขยายช่วงที่พบบ่อยคือ **Sign Extension (การขยายเครื่องหมาย)** ซึ่งใช้กับเลขจำนวนเต็มแบบมีเครื่องหมาย (Signed Integer)

หลักการคือ:

1. คัดลอกบิตเครื่องหมาย (MSB) ของค่าเดิม ไปเติมในบิตที่เพิ่มเข้ามาทางด้านซ้าย
2. ทำซ้ำจนกว่าจะได้ขนาดตามที่ต้องการ

**ตัวอย่าง:**

สมมติว่าเรามีเลขฐานสอง 4 บิต คือ 1011 (-5 ในรูปแบบ Two's Complement)

ต้องการขยายเป็น 8 บิต:

1. บิตเครื่องหมายคือ 1
2. คัดลอก 1 ไปเติมทางซ้าย: 11111011

ค่าที่ได้คือ 11111011 ซึ่งยังคงเป็น -5 ในรูปแบบ Two's Complement แต่มีขนาด 8 บิต

**ข้อควรระวัง**

* การขยายช่วงต้องทำอย่างถูกต้อง เพื่อให้ค่าที่ได้ยังคงถูกต้อง
* การขยายช่วงที่ไม่จำเป็น อาจทำให้สิ้นเปลืองพื้นที่หน่วยความจำ

**Fixed-Point Representation**

Fixed-Point Representation หรือการแทนจำนวนแบบจุดตรึง เป็นวิธีการแทนจำนวนจริง (จำนวนเต็มและทศนิยม) ในคอมพิวเตอร์ โดยกำหนดตำแหน่งของจุดทศนิยมไว้คงที่

**หลักการทำงาน**

* แบ่งบิตที่ใช้แทนจำนวนออกเป็นสองส่วน:
  + **ส่วนจำนวนเต็ม (Integer part)**: บิตทางซ้ายของจุดทศนิยม ใช้แทนจำนวนเต็ม
  + **ส่วนทศนิยม (Fractional part)**: บิตทางขวาของจุดทศนิยม ใช้แทนส่วนทศนิยม
* ตำแหน่งของจุดทศนิยมจะถูกกำหนดไว้ตายตัว ทำให้สามารถแทนจำนวนที่มีทศนิยมได้ แต่มีข้อจำกัดในเรื่องของช่วงของจำนวนที่สามารถแทนได้

**ตัวอย่าง**

สมมติว่าเราใช้เลขฐานสอง 8 บิต ในการแทนจำนวนแบบจุดตรึง โดยแบ่งเป็น 4 บิตสำหรับส่วนจำนวนเต็ม และ 4 บิตสำหรับส่วนทศนิยม

* 0101.1010
  + ส่วนจำนวนเต็ม: 0101 (5 ในเลขฐานสิบ)
  + ส่วนทศนิยม: 1010 (0.625 ในเลขฐานสิบ)
  + ดังนั้น 0101.1010 เท่ากับ 5.625 ในเลขฐานสิบ

**ข้อดี**

* **เข้าใจง่าย**: เป็นวิธีที่เข้าใจง่ายและตรงไปตรงมา
* **คำนวณรวดเร็ว**: การคำนวณทางคณิตศาสตร์ด้วยวิธี Fixed-Point ทำได้รวดเร็วกว่า Floating-Point

**ข้อเสีย**

* **ช่วงของจำนวนจำกัด**: เนื่องจากตำแหน่งจุดทศนิยมถูกกำหนดไว้ตายตัว ทำให้ช่วงของจำนวนที่สามารถแทนได้จำกัด
* **ความแม่นยำจำกัด**: จำนวนบิตที่ใช้แทนส่วนทศนิยมมีจำกัด ทำให้ความแม่นยำในการแทนจำนวนทศนิยมมีจำกัด

**Notation**

**Two complement operation**

**Two's Complement คืออะไร?**

* เป็นวิธีการแทนจำนวนเต็มบวกและลบ โดยใช้บิตซ้ายสุด (MSB) เป็นบิตเครื่องหมาย (0 = บวก, 1 = ลบ)
* ใช้ในการคำนวณเลขฐานสองได้อย่างมีประสิทธิภาพ โดยเฉพาะการบวกและการลบ

**หลักการทำงาน**

1. **การแปลงเลขบวกเป็นเลขลบ:**
   * เริ่มต้นด้วยเลขฐานสองที่เป็นค่าบวก เช่น +18 = 00010010
   * ทำการ **Invert** หรือกลับบิตทั้งหมด (0 เป็น 1 และ 1 เป็น 0) จะได้ 11101101
   * จากนั้น **บวก 1** เข้าไปในผลลัพธ์ที่ได้ จะได้ 11101110 ซึ่งเป็นค่า -18 ในรูปแบบ Two's Complement
2. **การแปลงเลขลบเป็นเลขบวก:**
   * เริ่มต้นด้วยเลขฐานสองที่เป็นค่าลบ เช่น -18 = 11101110
   * ทำการ Invert บิตทั้งหมด จะได้ 00010001
   * จากนั้นบวก 1 เข้าไป จะได้ 00010010 ซึ่งเป็นค่า +18

**ข้อดีของ Two's Complement**

* **ไม่มีศูนย์สองรูปแบบ**: มีเลข 0 เพียงรูปแบบเดียว (+0) ทำให้การคำนวณไม่ซับซ้อน
* **การคำนวณง่าย**: การบวกและการลบเลข Two's Complement สามารถทำได้โดยตรง โดยไม่ต้องสนใจเครื่องหมาย
* **ใช้แทนเลขได้กว้าง**: สามารถแทนจำนวนเต็มได้ในช่วงที่กว้าง

**กรณีพิเศษ: เลข 0**

1. **เริ่มต้นด้วยเลข 0:**
   * เลข 0 ในรูปแบบ Two's Complement คือ 00000000 (สมมติว่าใช้ 8 บิต)
2. **ทำการ Invert (กลับบิต):**
   * กลับบิตทั้งหมดของ 00000000 จะได้ 11111111
3. **บวก 1:**
   * บวก 1 เข้าไปใน 11111111 จะได้ 100000000
4. **เกิด Overflow:**
   * ผลลัพธ์ที่ได้มี 9 บิต ซึ่งเกินขนาดที่กำหนดไว้ (8 บิต) ทำให้เกิด Overflow
5. **ตัด Overflow ทิ้ง:**
   * บิตที่เกินมา (บิตซ้ายสุด) จะถูกตัดทิ้ง ทำให้เหลือ 00000000

**ผลลัพธ์**

* ผลลัพธ์สุดท้ายคือ 00000000 ซึ่งก็คือเลข 0 นั่นเอง

**ทำไมกรณีนี้ถึงสำคัญ?**

* กรณีนี้แสดงให้เห็นว่า เลข 0 ในระบบ Two's Complement มีเพียงรูปแบบเดียว คือ 00000000
* ไม่มีการแทนเลข 0 สองแบบ (+0 และ -0) เหมือนกับระบบ Sign-Magnitude
* ทำให้การคำนวณด้วย Two's Complement ง่ายและมีประสิทธิภาพมากยิ่งขึ้น

**กรณีพิเศษ: เลข -128**

1. **เริ่มต้นด้วยเลข -128:**
   * เลข -128 ในรูปแบบ Two's Complement คือ 10000000 (สมมติว่าใช้ 8 บิต)
2. **ทำการ Invert (กลับบิต):**
   * กลับบิตทั้งหมดของ 10000000 จะได้ 01111111
3. **บวก 1:**
   * บวก 1 เข้าไปใน 01111111 จะได้ 10000000

**ผลลัพธ์**

* ผลลัพธ์สุดท้ายคือ 10000000 ซึ่งก็คือเลข -128 เหมือนเดิม

**ทำไมกรณีนี้ถึงพิเศษ?**

* กรณีนี้แสดงให้เห็นว่า การทำ Two's Complement กับเลข -128 ไม่ได้ทำให้ค่าเปลี่ยนแปลง
* นั่นคือ -(-128) = -128
* เหตุผลคือ ช่วงของเลขที่สามารถแทนได้ในระบบ Two's Complement แบบ 8 บิต คือ -128 ถึง +127
* ดังนั้น -128 เป็นค่าที่ "สุดขั้ว" ของช่วง ทำให้การกลับเครื่องหมายไม่ได้ผลลัพธ์ที่ถูกต้อง

**ข้อสังเกต**

* รูปภาพนี้เน้นย้ำว่า บิตเครื่องหมาย (MSB) ควรจะเปลี่ยนแปลงระหว่างการกลับเครื่องหมาย
* แต่ในกรณีของ -128 บิตเครื่องหมายไม่เปลี่ยนแปลง ซึ่งบ่งชี้ถึงความพิเศษของกรณีนี้

**Overflow Rule**

**Overflow คืออะไร?**

* Overflow เกิดขึ้นเมื่อผลลัพธ์ของการคำนวณเลขฐานสองมีขนาดใหญ่เกินกว่าจำนวนบิตที่ใช้ในการจัดเก็บจะสามารถรองรับได้
* ในระบบ Two's Complement ซึ่งใช้บิตซ้ายสุดเป็นบิตเครื่องหมาย (0 = บวก, 1 = ลบ) Overflow สามารถเกิดขึ้นได้เมื่อ:
  + **บวกเลขบวกสองจำนวน**: ผลลัพธ์ที่ได้มีค่าเกินกว่าค่าบวกสูงสุดที่แทนได้
  + **บวกเลขลบสองจำนวน**: ผลลัพธ์ที่ได้มีค่าเกินกว่าค่าลบต่ำสุดที่แทนได้

**กฎ Overflow สำหรับ Two's Complement**

ในการตรวจสอบว่าเกิด Overflow หรือไม่เมื่อทำการบวกเลข Two's Complement มีกฎง่ายๆ ดังนี้ครับ:

* **ตรวจสอบบิตเครื่องหมาย (MSB) ของตัวถูกดำเนินการทั้งสอง:**
  + ถ้าเครื่องหมายเหมือนกัน (ทั้งคู่เป็นบวกหรือทั้งคู่เป็นลบ) **และ** ผลลัพธ์มีเครื่องหมาย *ต่าง* จากตัวถูกดำเนินการ **แสดงว่าเกิด Overflow**
  + ถ้าเครื่องหมายต่างกัน (ตัวหนึ่งเป็นบวก อีกตัวเป็นลบ) จะไม่มี Overflow

**ตัวอย่าง**

สมมติว่าเราใช้เลขฐานสอง 4 บิต ในระบบ Two's Complement ซึ่งสามารถแทนเลขได้ในช่วง -8 ถึง +7

* **ตัวอย่างที่ 1: การบวกที่ทำให้เกิด Overflow**
  + +7 (0111) + +5 (0101) = +12 (1100)
  + เนื่องจากเครื่องหมายของตัวถูกดำเนินการทั้งคู่เป็นบวก แต่ผลลัพธ์มีเครื่องหมายเป็นลบ แสดงว่าเกิด Overflow
* **ตัวอย่างที่ 2: การบวกที่ไม่ทำให้เกิด Overflow**
  + +7 (0111) + -5 (1011) = +2 (0010)
  + เนื่องจากเครื่องหมายของตัวถูกดำเนินการต่างกัน จึงไม่มี Overflow

**ความสำคัญของกฎ Overflow**

* การตรวจสอบ Overflow ช่วยให้เรามั่นใจได้ว่าผลลัพธ์ของการคำนวณถูกต้อง
* หากเกิด Overflow โดยไม่ตรวจสอบ อาจทำให้ได้ผลลัพธ์ที่ไม่ถูกต้อง ซึ่งอาจนำไปสู่ข้อผิดพลาดในการทำงานของระบบคอมพิวเตอร์

**Subtraction**

**คำอธิบายกฎการลบ**

* **โจทย์**: ในการลบเลขฐานสองตัวลบ (subtrahend) ออกจากตัวตั้ง (minuend)
* **วิธีการ**:\*
  1. **หาส่วนเติมเต็มสองของตัวลบ (subtrahend)**: ทำได้โดยการสลับบิต (0 เป็น 1 และ 1 เป็น 0) แล้วบวก 1
  2. **นำส่วนเติมเต็มสองที่ได้ไปบวกกับตัวตั้ง (minuend)**
  3. **ผลลัพธ์ที่ได้คือผลลัพธ์ของการลบ** (ถ้ามีบิตทดเกินมาให้ตัดทิ้ง)

**ตัวอย่าง (เพิ่มเติม)**

สมมติว่าเราต้องการลบ 0101 (5 ในเลขฐานสิบ) ออกจาก 1011 (11 ในเลขฐานสิบ)

1. **หาส่วนเติมเต็มสองของ 0101:**
   * สลับบิต: 1010
   * บวก 1: 1011
2. **นำ 1011 ไปบวกกับ 1011:**
   * 1011 + 1011 = 10110 (มีบิตทดเกินมา)
3. **ตัดบิตทดทิ้ง:**
   * เหลือ 0110 (6 ในเลขฐานสิบ)

ดังนั้น 1011 - 0101 = 0110

**ทำไมต้องใช้ Two's Complement ในการลบ?**

* **ความสะดวก**: การใช้ Two's Complement ทำให้การลบเลขฐานสองกลายเป็นการบวกเลขฐานสอง ซึ่งง่ายต่อการดำเนินการในวงจรอิเล็กทรอนิกส์ของคอมพิวเตอร์
* **ความสอดคล้อง**: Two's Complement ช่วยให้การบวกและลบเลขฐานสองเป็นไปในทิศทางเดียวกัน ทำให้วงจร ALU (Arithmetic Logic Unit) ใน CPU สามารถรองรับการคำนวณได้ทั้งบวกและลบอย่างมีประสิทธิภาพ