2. การแทนข้อมูลในคอมพิวเตอร์

องค์ประกอบคอมพิวเตอร์ใดๆ ขึ้นอยู่กับว่ามีการแทนตัวเลข ตัวอักษร และข้อมูลการควบคุมอย่างไร การสนทนาก็เป็นจริงเช่นกัน: มาตรฐานและอนุสัญญาที่จัดตั้งขึ้นในช่วงหลายปีที่ผ่านมา ได้กำหนดแง่มุมบางประการขององค์กรคอมพิวเตอร์ บทนี้จะอธิบายถึงวิธีการต่างๆ ที่คอมพิวเตอร์สามารถจัดเก็บและจัดการกับตัวเลขและตัวอักษร แนวคิดที่นำเสนอในส่วนต่อไปนี้เป็นพื้นฐานสำหรับการทำความเข้าใจองค์กรและหน้าที่ของระบบดิจิตอลทุกประเภท

หน่วยข้อมูลพื้นฐานที่สุดในคอมพิวเตอร์ดิจิตอลเรียกว่าบิต ซึ่งเป็นการดึงตัวอักษรจากคำว่า BInary digiT: bit ในแง่ที่เป็นรูปธรรมไม่มีอะไรมากไปกว่าสถานะ "เปิด" หรือ "ปิด" (หรือ "สูง" และ "ต่ำ") ภายในวงจรคอมพิวเตอร์ ในปี 1964 ผู้ออกแบบคอมพิวเตอร์เมนเฟรมของ IBM System/360 ได้กำหนดรูปแบบการใช้กลุ่ม 8 บิต เป็นหน่วยพื้นฐานของการจัดเก็บข้อมูลคอมพิวเตอร์ พวกเขาเรียกชุดนี้ว่า 8 บิตต่อไบต์

เวิร์ด (Word) ในคอมพิวเตอร์ประกอบด้วยสองหรือมากกว่าสองไบต์ที่อยู่ติดกัน ซึ่งบางครั้งมีการแก้ไขและเกือบทุกครั้งจะถูกจัดการโดยรวม ขนาดของเวิร์ดหมายถึงขนาดข้อมูลที่จัดการอย่างมีประสิทธิภาพมากที่สุดโดยสถาปัตยกรรมเฉพาะ เวิร์ดอาจมีขนาด 16 บิต, 32 บิต, 64 บิตหรือขนาดอื่นๆ ที่เหมาะสมในบริบทขององค์กรของคอมพิวเตอร์ (รวมถึงขนาดที่ไม่ใช่ทวีคูณแปด) ไบต์ 8 บิตสามารถแบ่งออกเป็นสองส่วน 4 บิตที่เรียกว่า nibbles (หรือ nibbles) เนื่องจากแต่ละบิตของไบต์มีค่าอยู่ภายในระบบการกำหนดตัวเลขตำแหน่งแทนที่ประกอบด้วยเลขฐานสองที่มีค่าน้อยที่สุดจึงถูกเรียกว่า low-order nibble ซึ่งประกอบด้วยสี่บิตล่าง และอีกครึ่งหนึ่งเป็น high-order nibble ที่ประกอบด้วยสี่บิตบน

2.2 ระบบการนับตำแหน่ง

ในช่วงกลางศตวรรษที่สิบหกยุโรปใช้ระบบเลขฐานสิบ (base 10) ที่ชาวอาหรับและฮินดูใช้มาเป็นเวลาเกือบหนึ่งพันปี วันนี้เรารับทราบว่าหมายเลข 243 หมายถึงสองร้อยบวกสี่สิบบวกสามหน่วย แม้ว่าความจริงที่ว่าศูนย์หมายถึง "ไม่มีอะไร" ทุกคนแทบจะรู้ว่ามีความแตกต่างอย่างมากระหว่างการมีสิ่งใดสิ่งหนึ่งและมี 10 อย่าง

แนวคิดทั่วไปที่ซ่อนอยู่เบื้องหลังระบบการกำหนดหมายเลขตำแหน่ง คือค่าตัวเลขแสดงผ่านตัวยกกำลังที่เพิ่มขึ้นของฐาน สิ่งนี้มักถูกเรียกว่าระบบการกำหนดหมายเลขแบบถ่วงน้ำหนัก เนื่องจากแต่ละตำแหน่งจะถูกถ่วงน้ำหนักด้วยตัวยกกำลังของเลขฐาน

ชุดของตัวเลขที่ถูกต้องสำหรับระบบหมายเลขตำแหน่งมีขนาดเท่ากับ radix ของระบบนั้น ตัวอย่างเช่นมีตัวเลข 10 ตัวในระบบเลขฐานสิบ, 0 ถึง 9 และมี 3 ตัว(ฐาน 3) สำหรับระบบที่ประกอบไปด้วย 0, 1 และ 2 หมายเลขที่ถูกต้องมากที่สุดในระบบ Radix คือ จะน้อยกว่าฐานอยู่หนึ่ง radix ดังนั้น 8 ไม่ใช่ตัวเลขที่ถูกต้องในระบบ Radix ใด ๆ ที่มีขนาดเล็กกว่า 9 เพื่อแยกความแตกต่างระหว่างตัวเลขใน radices ที่แตกต่างกันเราใช้ radix เป็นตัวห้อยเช่นใน 33_{10} เพื่อแทนเลข 33 ในฐานสิบ (ในที่นี้ตัวเลขที่เขียน โดยไม่มีตัวห้อยเป็นเลขฐานสิบ) เลขจำนวนเต็มฐานสิบใด ๆ สามารถแสดงได้อย่างชัดเจนในระบบฐานอินทิกรัลอื่น ๆ (ดูตัวอย่างที่ 2.1)

ตัวอย่าง 2.1 ตัวเลขสามจำนวนจะแสดงเป็นตัวยกกำลังของ Radix

$$243.51_{10} = 2 \times 10^2 + 4 \times 10^1 + 3 \times 10^0 + 5 \times 10^{-1} + 1 \times 10^{-2}$$
$$212_3 = 2 \times 3^2 + 1 \times 3^1 + 2 \times 3^0 = 23_{10}$$
$$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{10}$$

ระบบเลขฐานที่สำคัญที่สุดในวิทยาการคอมพิวเตอร์คือเลขฐานสอง และเลขฐานสิบหก อีกหนึ่งเลขฐานที่น่าสนใจคือฐานแปด ระบบเลขฐานสองใช้เฉพาะตัวเลข 0 และ 1; ระบบฐานแปด, 0 ถึง 7 ระบบเลขฐานสิบหกอนุญาตให้ตัวเลข 0 ถึง 9 กับ A, B, C, D, E และ F ที่ใช้เพื่อแทนตัวเลข 10 ถึง 15 ตารางที่ 2ใ1 แสดงเรดิตบางส่วน

2.3 การแปลงระหว่างฐาน

Gottfried Leibniz (1646-1716) เป็นคนแรกที่สรุปแนวคิดของระบบเลขฐานสิบ ที่แปลงไปยังฐานอื่นๆ การเป็นคนที่มีจิตวิญญาณอย่างลึกซึ้ง ไลบนิซกล่าวถึงคุณสมบัติอันสูงส่งของระบบเลขฐานสอง เขามีความสัมพันธ์กับความจริงที่ว่าจำนวนเต็มใดๆ สามารถแทนด้วยชุดของ 1 และ 0 กับแนวคิดที่ว่าพระเจ้า (1) สร้างจักรวาลโดยปราศจากอะไรเลย (0) จนกระทั่งคอมพิวเตอร์ดิจิตอลไบนารีตัวแรกถูกสร้างขึ้นในปลายทศวรรษที่ 1940 ระบบนี้ก็ยังคงไม่ได้มีอะไรมากไปกว่าความอยากรู้ทางคณิตศาสตร์ วันนี้มันเป็นหัวใจของอุปกรณ์อิเล็กทรอนิกส์แทบทุกประเภทที่ต้องอาศัยการควบคุมแบบดิจิตอล

เนื่องจากความเรียบง่ายระบบเลขฐานสอง จึงสามารถแปลงเป็นวงจรอิเล็กทรอนิกส์ได้อย่างง่ายดาย นอกจากนี้ยังเป็นเรื่องง่ายสำหรับมนุษย์ที่จะเข้าใจ ผู้เชี่ยวชาญด้านคอมพิวเตอร์ที่มีประสบการณ์สามารถจดจำเลขฐานสองที่เล็กกว่า (เช่นที่แสดงในตารางที่ 2ใ1) ได้อย่างรวดเร็ว อย่างไรก็ตามการแปลงค่าและเศษส่วนที่มากขึ้นมักจะต้องใช้เครื่องคิดเลขหรือดินสอและกระดาษ โชคดีที่เทคนิคการแปลงนั้นง่ายต่อการฝึกฝน เราแสดงเทคนิคที่ง่ายกว่าในส่วนที่ตามมา

2.3.1 การแปลงหมายเลขทั้งหมด

เริ่มต้นด้วยการแปลงฐานของตัวเลขที่ไม่มีเครื่องหมาย(unsigned numbers) การแปลงหมายเลขที่มีเครื่องหมาย (signed numbers ตัวเลขที่สามารถเป็นบวกหรือลบ) มีความซับซ้อนมากขึ้นและสิ่งสำคัญที่จะต้องเข้าใจเทคนิคพื้นฐานสำหรับการแปลงก่อนที่จะดำเนินการต่อด้วยหมายเลขที่มีเครื่องหมาย

การแปลงระหว่างระบบฐานสามารถทำได้โดยใช้การลบซ้ำหรือวิธีการหารเหลือเศษ วิธีการลบนั้นยุ่งยากและต้องการความคุ้นเคยกับตัวยกกำลังของฐานที่ใช้ เนื่องจากเป็นวิธีที่ใช้งานง่ายกว่าของสองวิธี อย่างไรก็ตามเราจะอธิบายให้ชัดเจนก่อน

ตัวอย่างเช่นสมมติว่าเราต้องการแปลง 538_{10} เป็นฐาน 8 เรารู้ว่า $8^3 = 512$ คือกำลังสูงสุด 8 ที่น้อยกว่า 538 ดังนั้นฐาน 8 ของเราจะเป็นตัวเลข 4 หลัก (หนึ่งสำหรับแต่ละกำลังของ radix: 0 ถึง 3) เราสังเกตว่า 512 เข้าไปครั้งเดียวใน 538 และลบออกโดยมีความแตกต่าง 26 เรารู้ว่ากำลังต่อไปของ 8, $8^2 = 64$ นั้นใหญ่เกินไปที่จะลบได้ดังนั้นเราจึงสังเกตว่า "ตัวยึด" เป็นศูนย์และมองหาจำนวนครั้งที่ $8^1 = 8$ หาร 26 เราเห็นว่ามันไปสามครั้งและลบ 24 เราเหลือ 2 ซึ่งก็คือ 2×8^0 ขั้นตอนเหล่านี้จะแสดงในตัวอย่าง 2.2

ตัวอย่าง 2.2 แปลง 538_{10} เป็นฐาน 8 โดยใช้การลบ

538
$-\ 512 = 8^3 \times 1$
26
$-\ 0 = 8^2 \times 0$
26
$-\ 24 = 8^1 \times 3$
2
$-\ 2 = 8^0 \times 2$
0

$538_{10} = 1032_8$

วิธีการหารที่เหลือเศษนั้นเร็วและง่ายกว่าวิธีการลบซ้ำๆ ใช้แนวคิดที่ว่าฝ่ายที่ต่อเนื่องโดยฐานนั้นในความเป็นจริงการลบอย่างต่อเนื่องโดยตัวยกกำลังของฐาน ส่วนที่เหลือที่เราได้รับเมื่อเราหารด้วยฐานท้ายเป็นตัวเลขของผลซึ่งอ่านจากล่างขึ้นบน วิธีการนี้จะแสดงในตัวอย่าง 2.3

ตัวอย่าง 2.3 แปลง 538_{10} เป็นฐาน 8 โดยใช้วิธีการหารส่วนที่เหลือ

8|538 2 8 divides 538 67 times with a remainder of 2.
8|67 3 8 divides 67 8 times with a remainder of 3.
8|8 0 8 divides 8 1 time with a remainder of 0.
8|1 1 8 divides 1 0 times with a remainder of 1.
0
Reading the remainders from *bottom to top*, we have: $538_{10} = 1032_8$.

วิธีนี้ใช้ได้กับทุกฐาน และเนื่องจากความเรียบง่ายของการคำนวณ จึงมีประโยชน์อย่างยิ่งในการแปลงจากเลขฐานสิบเป็นไบนารี ดังตัวอย่าง 2.4

ตัวอย่าง 2.4 แปลง 147_{10} เป็นไบนารี

2|147 1 2 divides 147 73 times with a remainder of 1.
2|73 1 2 divides 73 36 times with a remainder of 1.
2|36 0 2 divides 36 18 times with a remainder of 0.
2|18 0 2 divides 18 9 times with a remainder of 0.
2|9 1 2 divides 9 4 times with a remainder of 1.
2|4 0 2 divides 4 2 times with a remainder of 0.
2|2 0 2 divides 2 1 time with a remainder of 0.
2|1 1 2 divides 1 0 times with a remainder of 1.
0

อ่านการหารเหลือเศษจากล่างขึ้นบนเรามี: $147_{10} = 10010011_2$

ตัวเลขไบนารีที่มี *N* บิต สามารถแทนจำนวนเต็มแบบไม่มีเครื่องหมายได้ตั้งแต่ 0 ถึง 2^N - 1 ตัวอย่าง 4 บิต สามารถแสดงค่าทศนิยม 0 ถึง 15 ในขณะที่ 8 บิตสามารถแสดงค่า 0 ถึง 255 ช่วงของค่าที่สามารถแทนได้โดยจำนวนบิตที่กำหนดมีความสำคัญอย่างยิ่ง เมื่อดำเนินการทางคณิตศาสตร์กับเลขฐานสอง ลองพิจารณาสถานการณ์ที่เลขฐานสองมีความยาว 4 บิต และต้องการเพิ่ม 1111_2 (15_{10}) ถึง 1111_2 เรารู้ว่า 15 บวก 15 เป็น 30 แต่ 30 ไม่สามารถแทนได้โดยใช้เพียง 4 บิต นี่คือตัวอย่างของเงื่อนไขที่เรียกว่าโอเวอร์โฟลว์(overflow) ซึ่งเกิดขึ้นในการแทนเลขฐานสองแบบไม่มีเครื่องหมาย เมื่อผลลัพธ์ของการดำเนินการทางคณิตศาสตร์อยู่นอกช่วง

ของความแม่นยำที่อนุญาตสำหรับจำนวนบิตที่กำหนด รายละเอียดเพิ่มเติมเกี่ยวกับการล้นจะพูดกล่าวอีกครั้งในหัวข้อ 2.4

2.3.2 การแปลงเศษส่วน

เศษส่วนในระบบฐานใดๆ สามารถแปลงเป็นไปเป็นระบบฐานอื่นๆ โดยใช้กำลังติดลบของฐาน จุดใช้แยกจำนวนเต็มของตัวเลขออกจากเศษเหลือ ในระบบทศนิยมจุดฐานเรียกว่าจุดทศนิยม

เศษส่วนที่มีสตริงหรือสายของตัวเลขแบบวนซ้ำของตัวเลขทางด้านขวาของจุดฐานในฐานหนึ่ง อาจไม่มีลำดับการทำซ้ำของตัวเลขในฐานอื่น ตัวอย่างเช่น 2/3 เป็นเศษส่วนทศนิยมซ้ำ แต่ในระบบฐานสามที่ประกอบไปด้วยสามส่วนมันจะสิ้นสุดเป็น 0.2_3 ($2 \times 3^{-1} = 2 \times 1/3$)

เราสามารถแปลงเศษส่วนระหว่างฐานต่างๆ โดยใช้วิธีที่คล้ายคลึงกับวิธีการลบซ้ำ และวิธีหารเหลือเศษสำหรับการแปลงจำนวนเต็ม ตัวอย่าง 2.5 แสดงวิธีที่เราสามารถใช้การลบซ้ำ เพื่อแปลงตัวเลขจากฐานสิบเป็นฐาน 5

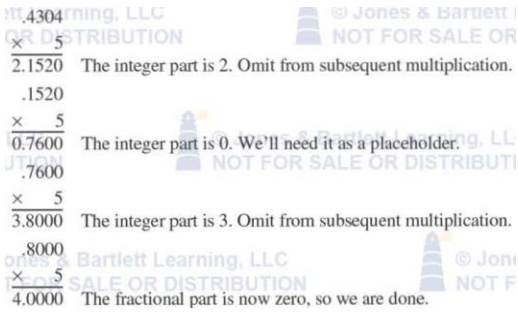
ตัวอย่าง 2.5 แปลง 0.4304_{10} เป็นฐาน 5

$$
\begin{array}{l}
0.4304 \\
\underline{-\ 0.4000} = 5^{-1} \times 2 \\
0.0304 \\
\underline{-\ 0.0000} = 5^{-2} \times 0 \quad \text{(A placeholder)} \\
0.0304 \\
\underline{-\ 0.0240} = 5^{-3} \times 3 \\
0.0064 \\
\underline{-\ 0.0064} = 5^{-4} \times 4 \\
0.0000
\end{array}
$$

อ่านจากบนลงล่าง $0.4304_{10} = 0.2034_5$

เนื่องจากวิธีหารเหลือเศษทำงานร่วมกับตัวยกกำลับบวกของเลขฐาน สำหรับการแปลงจำนวนเต็มมันจึงมีเหตุผลว่าจะต้องใช้การคูณ เพื่อแปลงเศษส่วนเพราะพวกมันแสดงด้วยตัวยกกำลังลบของฐาน อย่างไรก็ตามแทนที่จะมองหาเศษตามที่เราทำด้านบน ก็สามารถใช้เฉพาะจำนวนเต็มของผลหลังจากการคูณด้วยเลขฐาน คำตอบได้จากการอ่านจากบนลงล่างแทนล่างขึ้นบน ตัวอย่าง 2.6 แสดงกระบวนการ

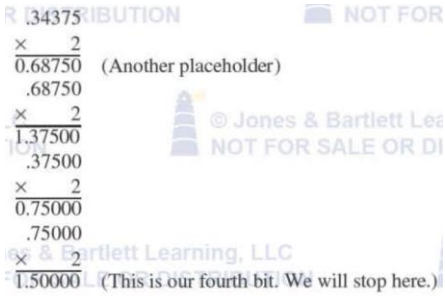
ตัวอย่าง 2.6 แปลง 0.4304_{10} เป็นฐาน 5



อ่านจากบนลงล่างจะได้ $0.4304_{10} = 0.2034_5$

ตัวอย่างนี้แสดงให้เห็นกระบวนการหยุดหลังจากดำเนินการผ่านไปไม่กี่ขั้นตอน บ่อยครั้งที่สิ่งต่างๆ ไม่ได้ผลอย่างเท่าเทียมกันและจบลงด้วยการใช้เศษส่วนวนซ้ำ ระบบคอมพิวเตอร์ส่วนใหญ่ใช้อัลกอริทึมการปัดเศษแบบพิเศษ เพื่อให้ระดับความแม่นยำที่คาดการณ์ได้ อย่างไรก็ตามเพื่อความชัดเจนจะตัด คำตอบของเราเมื่อได้รับความแม่นยำตามที่แสดงในตัวอย่างที่ 2.7

ตัวอย่าง 2.7 แปลง 0.34375_{10} เป็นไบนารี 4 บิตทางด้านขวาของจุดไบนารี



อ่านจากบนลงล่าง $0.34375_{10} = 0.0101_2$ ซึ่งมีสี่ตำแหน่งหลังจุดไบนารี

สิ่งที่ได้อธิบายไว้ สามารถใช้ในการแปลงตัวเลขใดๆ ในฐานใดๆ ไปยังฐานอื่นๆ ได้โดยตรง เช่นจากฐาน 4 เป็นฐาน 3 (ดังในตัวอย่างที่ 2.8) อย่างไรก็ตามในกรณีนี้ส่วนใหญ่จะเร็วกว่าและแม่นยำกว่า เมื่อแปลงเป็นฐาน 10 ก่อนแล้วจึงไปยังฐานที่ต้องการ ข้อยกเว้นอย่างหนึ่งของกฎนี้คือ เมื่อทำการแปลงระหว่างฐานที่มีตัวยกกำลังสองดังที่จะได้แสดงให้เห็นในหัวข้อถัดไป

ตัวอย่าง 2.8 แปลง 3121_4 เป็นฐาน 3

ก่อนอื่นให้แปลงเป็นเลขฐานสิบก่อน:

$$3121_4 = 3 \times 4^3 + 1 \times 4^2 + 2 \times 4^1 + 1 \times 4^0$$
$$= 3 \times 64 + 1 \times 16 + 2 \times 4 + 1 = 217_{10}$$

แล้วจึงแปลงเป็นเลขฐาน 3

3|217	1
3|72	0
3|24	0
3|8	2
3|2	2
0	We have $3121_4 = 22{,}001_3$.

2.3.3 การแปลงระหว่าง Power-of-Two Radices

เลขไบนารีมักถูกแสดงเป็นเลขฐานสิบหก และในบางครั้งแสดงในฐานแปด เพื่อปรับปรุงความสามารถในการอ่าน เนื่องจาก 16 = 2^4 กลุ่มของ 4 บิต (เรียกว่า hextet) จะจดจำได้ง่ายว่าเป็นเลขฐานสิบหก ในทำนองเดียวกันกับ 8 = 2^3 กลุ่ม 3 บิต (เรียกว่า octet) สามารถแทนด้วยเลขฐานแปดเพียงหนึ่งตัว ด้วยการใช้ความสัมพันธ์เหล่านี้ เราสามารถแปลงตัวเลขจากไบนารี่เป็นฐานแปด หรือฐานสิบหกได้ดังนี้

ตัวอย่าง 2.9 แปลง 110010011101_2 เป็นฐานแปดและฐานสิบหก

110 010 011 101 6 2 3 5	Separate into groups of 3 bits for the octal conversion.
	$110010011101_2 = 6235_8$
1100 1001 1101 C 9 D	Separate into groups of 4 for the hexadecimal conversion.
	$110010011101_2 = C9D_{16}$

2.4 การแทนจำนวนเต็มแบบมีเครื่องหมาย

เราได้เห็นวิธีการแปลงจำนวนเต็มแบบไม่มีเครื่องหมายจากฐานหนึ่งไปยังอีก เลขที่มีเครื่องหมายต้องมีการแก้ไขปัญหาเพิ่มเติม เมื่อประกาศตัวแปรจำนวนเต็มในโปรแกรมภาษา การเขียนโปรแกรมจะจัดสรรพื้นที่เก็บข้อมูลที่มีเครื่องหมายด้วยบิตแรกของตำแหน่งหน่วยเก็บข้อมูลโดยอัตโนมัติ ตามแบบแผน 1 ในบิตลำดับสูงแทนจำนวนลบ ที่เก็บข้อมูลอาจมีขนาดเล็กเพียง 8 บิต หรือใหญ่กว่าไบต์ หรือแทนด้วยเวิร์ดขึ้นอยู่กับภาษาที่เขียนโปรแกรมและระบบคอมพิวเตอร์ บิตที่เหลือ (หลังจากเครื่องหมายบิต) จะถูกใช้เพื่อแทนตัวเลข

วิธีการแสดงจำนวนนี้ขึ้นอยู่กับวิธีการที่ใช้ มีสามวิธีที่ใช้กันทั่วไป วิธีการที่ง่ายที่สุดคือ signed magnitude ใช้บิตที่เหลือเพื่อแสดงขนาดของจำนวน ส่วนอีกสองวิธีใช้แนวคิดของการเติมเต็ม(complements) ซึ่งจะได้อธิบายกันในหัวข้อต่อไป

2.4.1 Signed Magnitude

ในหัวข้อนี้ เราได้ละความเป็นไปได้ของการแทนเลขฐานสองสำหรับจำนวนลบ เซตของจำนวนเต็มบวกและลบเรียกว่า เซตจำนวนเต็มแบบมีเครื่องหมายหรือ signed integers ปัญหาของการแทนจำนวนเต็มที่มีเครื่องหมายเป็นค่าไบนารี่ เป็นสัญญาณที่เราควรเข้ารหัสสัญญาณจริงของตัวเลขหรือไม่? การแสดงขนาดแบบเครื่องหมายเป็นวิธีการหนึ่งในการแก้ปัญหานี้ ตามชื่อของมันหมายถึงหมายเลขที่แทนขนาดมีเครื่องหมายเป็นบิตซ้ายสุด (เรียกอีกอย่างว่าบิตลำดับสูงหรือบิตที่สำคัญที่สุด MSB: most significant bit) ในขณะที่บิตที่เหลือแสดงขนาด (หรือค่าสัมบูรณ์) ของค่าตัวเลข ตัวอย่างเช่น เวิร์ด 8 บิต -1 จะแสดงเป็น 10000001 และ +1 เป็น 00000001 ในระบบคอมพิวเตอร์ที่ใช้การแทน signed-magnitude และ 8 บิตเพื่อเก็บจำนวนเต็ม โดยสามารถใช้ 7 บิต สำหรับการแทนค่าจริง ซึ่งหมายความว่าจำนวนเต็มที่มากที่สุดของเวิร์ด 8 บิต สามารถแทน 2^7- 1 หรือ 127 (0 ในบิตลำดับสูงตามด้วย 1 อีกเจ็ดตัว) จำนวนเต็มที่น้อยที่สุดคือ 1 แปดตัวหรือ -127 ดังนั้น *N* bits สามารถแทน -($2^{(N-1)}$ -1) ถึง $2^{(N-1)}$ – 1

คอมพิวเตอร์สามารถคำนวณคณิตศาสตร์ที่เป็นจำนวนเต็มที่แทนด้วยเครื่องหมาย การคำนวณทางคณิตศาสตร์แบบ Signed-magnitude ใช้วิธีการเดียวกับที่มนุษย์ใช้ดินสอและกระดาษคำนวณ แต่อาจทำให้เกิดความสับสนได้ ให้พิจารณาตัวอย่างกฎสำหรับการเพิ่ม:

(1) หากเครื่องหมายเหมือนกันให้เพิ่มขนาดและใช้เครื่องหมายเดียวกันนั้นสำหรับผลลัพธ์

(2) หากสัญญาณแตกต่างกันต้องพิจารณาว่าตัวถูกดำเนินการใดมีขนาดใหญ่กว่า เครื่องหมายของผลลัพธ์นั้นเหมือนกับเครื่องหมายของโอเปอแรนด์ที่มีขนาดใหญ่กว่า และต้องได้ขนาดโดยการลบ (ไม่เพิ่ม) ค่าที่เล็กกว่าจากอันที่ใหญ่กว่า

หากคุณพิจารณากฎเหล่านี้อย่างรอบคอบ นี่เป็นวิธีที่ใช้สำหรับการคำนวณทางคณิตศาสตร์ด้วยมือ

เราจัดเรียงตัวถูกดำเนินการด้วยวิธีการบางอย่าง ขึ้นอยู่กับเครื่องหมาย ดำเนินการคำนวณโดยไม่คำนึงถึงเครื่องหมาย แล้วค่อยพิจารณาใส่เครื่องหมายให้ถูกต้องตามความเหมาะสม เมื่อการคำนวณเสร็จสมบูรณ์ เมื่อสร้างแบบจำลองความคิดนี้ด้วยเวิร์ด 8 บิต จะต้องระมัดระวังที่จะรวมเพียง 7 บิต ขนาดของคำตอบโดยละการกระทำใดๆ ที่เกิดขึ้นแล้วมากกว่าบิตที่มีลำดับสูง

ตัวอย่าง 2.10 บวก 01001111_2 กับ 00100011_2 โดยใช้เลขจำนวนเต็มแบบมีเครื่องหมาย

				1	1	1	1		⇐ carries
0		1	0	0	1	1	1	1	(79)
0	+	0	1	0	0	0	1	1	+ (35)
0		1	1	1	0	0	1	0	(114)

การคำนวณทางคณิตศาสตร์นั้น เหมือนกับการบวกเลขฐานสิบ รวมถึงการดำเนินการจนกระทั่งเราได้บิตที่เจ็ดจากด้านขวา หากมีการทดเข้ามาที่ตำแหน่งนี้จะเกิดสภาพล้น และการทดถูกยกเลิก ส่งผลให้ผลรวมไม่ถูกต้อง แต่ตัวอย่างนี้ไม่มีการล้น

เราพบว่า $01001111_2 + 00100011_2 = 01110010_2$ ในการแทนแบบ signed-magnitude

บิตเครื่องหมายจะถูกแยกออก เนื่องจากมีความเกี่ยวข้องเฉพาะ หลังจากการเติมเต็มเท่านั้น ในกรณีนี้เรามีผลบวกของสองจำนวนบวกซึ่งเป็นบวกล้น (และทำให้เกิดข้อผิดพลาด) จำนวนที่มีเครื่องหมายเกิดขึ้นเมื่อเครื่องหมายของผลลัพธ์ไม่ถูกต้อง

signed magnitude มีบิตเครื่องหมายใช้สำหรับกำหนดว่าค่านั้นเป็นบวกหรือลบ ดังนั้นจึงไม่สามารถ "ดำเนินการ" ได้ หากมีการทดออกจากบิตที่เจ็ด ผลลัพธ์ก็จะถูกตัดทอนเป็นบิตที่เจ็ดล้น โดยให้ผลรวมไม่ถูกต้อง (ตัวอย่าง 2.11 แสดงให้เห็นถึงสภาพการล้นนี้) โปรแกรมเมอร์ที่ชาญฉลาดหลีกเลี่ยงข้อผิดพลาด "ล้านดอลลาร์" โดยการตรวจสอบเงื่อนไขการล้น เมื่อใดก็ตามที่มีความเป็นไปได้น้อยที่สุดที่พวกเขาอาจเกิดขึ้นได้ หากไม่ทิ้งบิตล้นทำให้ผลลัพธ์ที่ผิดมากยิ่งขึ้น จากการรวมของตัวเลขบวกสองตัวที่เป็นค่าลบ (ลองนึกภาพว่าจะเกิดอะไรขึ้นหากขั้นตอนถัดไปในโปรแกรมคือการใช้สแควร์รูทหรือบันทึกของผลลัพธ์นั้น!)

ตัวอย่าง 2.11 บวก 01001111_2 กับ 01100011_2 โดยใช้ signed-magnitude

Last carry overflows and is discarded.	1	←	1	1	1	1	1		⇐ carries
	0		1	0	0	1	1	1	(79)
	0	+	1	1	0	0	0	1	1	+ (99)
	0		0	1	1	0	0	1	0	(50)

เราได้ผลลัพธ์ที่ผิดพลาดจาก 79 + 99 = 50

เช่นเดียวกับการบวก การลบ signed-magnitude จะดำเนินการในลักษณะที่คล้ายกับเลขฐานสิบ โดยใช้ดินสอและกระดาษ ซึ่งบางครั้งก็จำเป็นต้องยืมจากตัวเลขใน minuend

ตัวอย่าง 2.12 ลบ 01001111_2 จาก 01100011_2 โดยใช้เลขคณิต signed-magnitude

			0	1	1	2		⇐ borrows
0		1	+	~~0~~	~~0~~	~~0~~	1	1	(99)
0	−	1	0	0	1	1	1	1	− (79)
0		0	0	1	0	1	0	0	(20)

พบว่า 01100011_2 - 01001111_2 = 00010100_2 ในการแทน signed-magnitude

ตัวอย่าง 2.13 ลบ 01100011_2 (99) จาก 01001111_2 (79) โดยใช้เลขคณิต signed-magnitude

จากการตรวจสอบเราจะเห็นว่า subtrahend 01100011 นั้นมีขนาดมากกว่า minuend ที่ 01001111 ด้วยผลลัพธ์ที่ได้ในตัวอย่างที่ 2.12 เรารู้ว่าความแตกต่างของตัวเลขสองตัวนี้คือ 0010100_2 เพราะ subtrahend นั้นมากกว่า minuend สิ่งที่ต้องทำคือเปลี่ยนสัญลักษณ์ของความแตกต่าง ดังนั้นเราจึงพบว่า 01001111_2 - 01100011_2 = 10010100_2 สำหรับการแทน signed-magnitude

เป็นที่ทราบกันแล้วว่าการลบนั้นเหมือนกับ "การบวกที่ตรงกันข้าม" ซึ่งเท่ากับลบค่าที่เราต้องการลบแล้วบวกแทน (ซึ่งมักจะง่ายกว่าการยืมทั้งหมดที่จำเป็นสำหรับการลบโดยเฉพาะอย่างยิ่ง ในการจัดการกับเลขฐานสอง) ดังนั้นเราต้องดูตัวอย่างที่เกี่ยวข้องทั้งบวกและจำนวนลบ จำกฎสำหรับการบวก: (1) หากเครื่องหมายเหมือนกันให้เพิ่มขนาดและใช้เครื่องหมายเดียวกันนั้นสำหรับผลลัพธ์ (2) หากสัญญาณแตกต่างกันคุณต้องพิจารณาว่าตัวถูกดำเนินการใดมีขนาดมากกว่า เครื่องหมายของผลลัพธ์นั้นเหมือนกับเครื่องหมายของตัวถูกดำเนินการ ที่มีมากขึ้นและขนาดจะต้องได้รับโดยการลบ (ไม่เพิ่ม) อันที่เล็กกว่าจากตัวที่ใหญ่กว่า

ตัวอย่าง 2.14 บวก 10010011_2 (-19) ถึง 00001101_2 (+13) โดยใช้เลขคณิต signed-magnitude

ตัวเลขแรก (augend) เป็นค่าลบ เนื่องจากบิตเครื่องหมายถูกตั้งค่าเป็น 1 จำนวนที่สอง (ตัวบวก) เป็นค่าบวก สิ่งที่ถูกขอให้ทำก็คือการลบ อันดับแรกเราพิจารณาว่าตัวเลขสองตัวใดมีขนาดใหญ่กว่าและใช้ตัวเลขนั้นสำหรับการบวก เครื่องหมายของมันจะเป็นเครื่องหมายของผลลัพธ์

			0	1	2			⇐ borrows	
1		0	0	+	~~0~~	~~0~~	1	1	(–19)
0	–	0	0	0	1	1	0	1	+ (13)
1		0	0	0	0	1	1	0	(–6)

ด้วยการรวมบิตเครื่องหมายจะเห็นว่า 10010011_2 - 00001101_2 = 10000110_2 ในการ signed-magnitude

ตัวอย่าง 2.15 ลบ 10011000_2 (-24) จาก 10101011_2 (-43) โดยใช้เลขคณิต signed-magnitude

เราสามารถแปลงการลบเป็นการบวกได้โดยการลบ -24 ซึ่งได้ 24 แล้วสามารถบวก -43 ทำให้มีปัญหาใหม่ที่ -43 +24 อย่างไรก็ตามจากกฎการบวกข้างต้น เพราะขณะนี้เครื่องหมายต่างกัน ต้องลบขนาดที่เล็กกว่าจากขนาดที่ใหญ่กว่า (หรือลบ 24 จาก 43) และทำให้ผลลัพธ์เป็นลบ (เพราะ 43 มีขนาดใหญ่กว่า 24)

		0	2					
	0	+	0	1	0	1	1	(43)
–	0	0	1	1	0	0	0	– (24)
	0	0	1	0	0	1	1	(19)

จำไว้ว่าไม่เกี่ยวข้องกับเครื่องหมาย จนกว่าจะทำการลบ เรารู้ว่าคำตอบแล้วว่าต้องเป็นลบ ดังนั้นจึงจบด้วย 10101011_2 - 10011000_2 = 10010011_2 ในการแทน signed-magnitude

ในตัวอย่างก่อนหน้านี้อาจสังเกตเห็นว่า มีคำถามมากมายที่ต้องถามตัวเอง: จำนวนใดที่ใหญ่กว่า ฉันจะลบจำนวนลบหรือไม่ ฉันต้องยืมจาก minuend กี่ครั้ง? คอมพิวเตอร์ที่ได้รับการออกแบบทางวิศวกรรม เพื่อทำการคำนวณในลักษณะนี้จะต้องทำการตัดสินใจหลายๆ อย่าง (แม้ว่าจะเร็วกว่ามาก) ตรรกะมีความซับซ้อนมากขึ้นโดยข้อเท็จจริงที่ว่าขนาดที่เครื่องหมายมีสองแนวทางสำหรับ 0, 10000000 และ 00000000 (ในทางคณิตศาสตร์ ไม่ควรเกิดขึ้น!) วิธีที่ง่ายกว่าสำหรับการแทนตัวเลขที่มีเครื่องหมายโดยหาวิธีที่ง่ายขึ้นและต้นทุนน้อยลง วงจรมีราคาแพง วิธีการที่ง่ายกว่านี้ขึ้นอยู่กับระบบคอมพลีเมนต์เลขฐาน

2.4.2 Complement Systems

ในทฤษฎีจำนวนเป็นที่ทราบกันมาหลายร้อยปีแล้วว่า เลขฐานสิบทศนิยมหนึ่งตัวสามารถนำมาลบออกจากอีกตัวหนึ่ง โดยการบวกผลต่างของตัวลบจากเก้าที่มีจำนวนหลักเท่าตัวตั้งและบวกกลับเข้าไปด้วยตัวทด สิ่งนี้เรียกว่าการใช้ส่วนเติมเต็มของเก้าหรือคอมพลีเมนต์ (9-Complement) ของตัวลบ หรือโดยทั่วไปเป็นการหาคอม

พลีเมนต์ของฐาน สมมติว่าเราต้องการหา 167 - 52 โดยหาความแตกต่างของตัวลบ 52 จาก 999 ซึ่งเท่ากับ 947 ดังนั้นในเลขเก้าส่วนประกอบเรามี 167 - 52 = 167 + 947 = 1114 ตัวทดจากคอลัมน์ของหลักร้อยจะถูกบวกเข้าไปที่หลักนั่นคือ 167 - 52 = 1_14+1 = 115 วิธีนี้มักเรียกว่า “Casting out 9s” เมื่อถูกขยายไปยังการดำเนินการแบบไบนารี เพื่อลดความซับซ้อนของการคำนวณทางคณิตศาสตร์ ข้อได้เปรียบของระบบคอมพลีเมนต์ที่มีให้มากกว่าวิธี signed magnitude คือไม่จำเป็นต้องประมวลบิตเครื่องหมาย แต่เรายังสามารถตรวจสอบเครื่องหมายของตัวเลขได้อย่างง่ายดายโดยดูที่บิตลำดับสูง

อีกวิธีหนึ่งให้ลองการนึกภาพระบบประกอบ หรือจินตนาการถึงตัววัดระยะทางที่ใช้กับจักรยาน ซึ่งต่างจากรถยนต์ เมื่อถอยหลังจักรยานระยะทางจะย้อนกลับด้วยเช่นกัน สมมติว่าเครื่องวัดระยะทางมีตัวเลขสามหลัก หากเราเริ่มต้นที่ศูนย์และสิ้นสุดด้วย 700 เราไม่สามารถแน่ใจได้ว่าจักรยานเดินไปข้างหน้า 700 ไมล์หรือถอยหลัง 1000-300=700 ไมล์กันแน่ วิธีที่ง่ายที่สุดสำหรับกรณีนี้คือการแบ่งครึ่งเพื่อตัดพื้นที่จำนวนครึ่งโดยใช้ 001-500 สำหรับไมล์บวกและ 501-999 สำหรับไมล์ลบ เรามีประสิทธิภาพลดระยะทางที่เครื่องวัดระยะทางของเราสามารถวัดได้ แต่ตอนนี้ถ้าอ่าน 997 เรารู้ว่าจักรยานถอยหลัง 3 ไมล์แทนที่จะขี่ไปข้างหน้า 997 ไมล์ หมายเลข 501-999 แสดงถึงการเติมเต็มของ Radix (วิธีที่สองของสองวิธีที่แนะนำด้านล่าง) ของหมายเลข 001-500 และใช้เพื่อแสดงระยะห่างเชิงลบ

One's Complement

จากที่กล่าวมาข้างต้น ในเรื่องการคอมพลีเมนต์เลขฐานสิบ ที่สามารถคำนวณได้จากการลบตัวลบ (subtrahend) ออกจากฐานลบหนึ่งซึ่งเท่ากับ 9 ในฐานสิบ กล่าวโดยทั่วไปเมื่อกำหนดตัวเลข N ใดๆที่อยู่ในฐาน r มี d หลัก คอมพลีเมนต์ฐานของ N จะถูกกำหนดได้โดย $(r^d - 1) - N$ สำหรับเลขฐานสิบ, r = 10 และ radix ที่ลดลงคือ 10 -1 = 9 ตัวอย่างคอมพลีเมนต์ของเก้าจากค่า 2468 คือ 9999 - 2468 = 7531 สำหรับการดำเนินการที่เทียบเท่าในไบนารีเราลบจากฐานที่น้อยกว่าหนึ่ง (2) ซึ่งก็คือ 1 ตัวอย่างเช่นส่วนประกอบหนึ่งของ 0101_2 คือ 1111_2 - 0101 = 1,010 แม้ว่าสามารถยืมและนำมาลบกันได้ดังที่กล่าวไว้ข้างต้น แต่จากการทดลองบางอย่างจะทำให้เชื่อว่าการคอมพลีเมนต์เลขฐานสองไม่มีอะไรมากไปกว่าการเปลี่ยน 1s ทั้งหมดเป็น 0s และในทางกลับกัน การกลับบิต(Bit-flipping) แบบนี้ง่ายมากที่จะนำไปใช้กับฮาร์ดแวร์คอมพิวเตอร์

เป็นสิ่งสำคัญที่จะต้องทราบ ณ จุดนี้ว่า ถึงแม้จะสามารถหาส่วนเติมเต็มเก้าของเลขฐานสิบตัวใดๆ หรือ One's complement ของเลขฐานสองใดๆ เราก็สนใจที่จะใช้สัญกรณ์คอมพลีเมนต์เพื่อแทนจำนวนลบ เรารู้ว่าการลบ เช่น 10 - 7 นั้นอาจคิดว่าเป็น "การบวกกับค่าตรงกันข้าม" นั่นคือ 10 + (-7) สัญกรณ์ Complementation ช่วยให้ทำการลบได้ง่ายขึ้น โดยการเปลี่ยนให้เป็นการบวก แต่มันก็ทำให้เรามีวิธีในการแทนจำนวนลบ เนื่องจากเราไม่ต้องการใช้บิตพิเศษเพื่อแสดงเครื่องหมาย (ดังที่ทำในการแทน signed-magnitude) เราจำเป็นต้องจำไว้ว่าหากตัวเลขเป็นลบเราควรแปลงเป็นคอมพลีเมนต์ ผลลัพธ์ควรมี 1 ในตำแหน่งบิตซ้ายสุดเพื่อระบุว่าจำนวนลบ

แม้ว่า one's complement ของตัวเลขจะเป็นค่าทางเทคนิคที่ได้จากการลบตัวเลขนั้นออกจากกำลังสูงสุดของสอง เรามักจะอ้างถึงคอมพิวเตอร์ที่ใช้ one's complement สำหรับตัวเลขลบเป็นระบบประกอบหนึ่ง หรือเป็นคอมพิวเตอร์ที่ใช้เลขคณิต one's complement สิ่งนี้อาจทำให้เข้าใจผิดได้ เนื่องจากตัวเลขบวกไม่จำเป็นต้องคอมพลีเมนต์ เราคอมพลีเมนต์จำนวนลบเท่านั้นเพื่อให้เราสามารถจัดรูปแบบที่คอมพิวเตอร์จะเข้าใจได้ ตัวอย่าง 2.16 แสดงแนวคิดเหล่านี้

ตัวอย่าง 2.16 ให้แสดงการบวก 23_{10} และ -9_{10} ในโหมดไบนารี 8 บิต โดยสมมติว่าคอมพิวเตอร์ใช้การแทนแบบ one's complement

$$23_{10} = +(00010111_2) = 00010111_2$$
$$-9_{10} = -(00001001_2) = 11110110_2$$

จะเห็นว่าแตกต่าง signed magnitude การบวกในระบบ one's complement ไม่จำเป็นต้องคำนึงถึงบิตเครื่องหมายแยกจากบิตอื่นๆ เครื่องหมายจะจัดการตัวเองได้ เปรียบเทียบตัวอย่าง 2.17 กับตัวอย่างที่ 2.10

ตัวอย่าง 2.17 ให้บวก 01001111_2 กับ 00100011_2 โดยใช้ one's complement

				1	1	1	1	⇐ carries	
	0	1	0	0	1	1	1	1	(79)
+	0	0	1	0	0	0	1	1	+ (35)
	0	1	1	1	0	0	1	0	(114)

สมมติว่าเราต้องการลบ 9 จาก 23 เพื่อทำการลบแบบ one's complement จะเริ่มจากต้องแปลงตัวลบก่อน (9) ให้อยู่ในรูป one's complement แล้วนำไปบวกกับตัวตั้ง (23); ถึงตอนนี้ก็สามารถบวก -9 กับ 23 บิตที่มีลำดับสูงจะมีการทดเป็น 1 หรือ 0 ก็จะถูกบวกเข้ากับบิตในลำดับต่ำสุด (สิ่งนี้เรียกว่า end carry-around และผลลัพธ์จากการใช้คอมพลีเมนต์ฐาน)

ตัวอย่าง 2.18 ให้บวก 23_{10} กับ -9_{10} โดยใช้ one's complement

	$1 \leftarrow$	1	1	1		1	1		$\Leftarrow$ carries
		0	0	0	1	0	1	1	1 (23)
	+	1	1	1	1	0	1	1	0 + (−9)
The last		0	0	0	0	1	1	0	1
carry is added								+	1
to the sum.		0	0	0	0	1	1	1	0 14_{10}

ตัวอย่าง 2.19 ให้บวก 9_{10} กับ -23_{10} โดยใช้ one's complement

The last	0	$\leftarrow$	0	0	0	0	1	0	0	1 (9)
carry is 0		+	1	1	1	0	1	0	0	0 + (−23)
so we are done.			1	1	1	1	0	0	0	1 -14_{10}

จะทราบได้อย่างไรว่า 11110001_2 คือ -14_{10} ในที่นี้ต้องใช้ one's complement กับ 11110001_2 (จำได้ว่ามันต้องเป็นลบเพราะบิตซ้ายสุดเป็นลบ) เมื่อคอมพลีเมนต์ของ 11110001_2 คือ 00001110_2 ซึ่งเท่ากับ 14

ข้อด้อยหลัก one's complement คือยังต้องมีการแทนได้สองแบบสำหรับค่า 0 นั่นคือ 00000000 และ 11111111 ด้วยเหตุนี้และอื่นๆ วิศวกรคอมพิวเตอร์เลิกใช้ one's complement นานมาแล้ว โดยไปใช้การแทนแบบ two's complement

Two's Complement

Two's Complement เป็นตัวอย่างของคอมพลีเมนต์ฐาน เมื่อกำหนดตัวเลข N ในฐาน r ที่มี d หลัก เลขฐานคอมพลีเมนต์ของ N ถูกกำหนดจาก r^d - N สำหรับ N ≠ 0 และ 0 สำหรับ N = 0 คอมพลีเมนต์ฐาน มักจะใช้งานได้ง่ายกว่าคอมพลีเมนต์ฐานลบหนึ่งอย่าง one's Complement จากตัวอย่างของเครื่องวัดระยะทางจักรยานคอมพลีเมนต์ฐานสิบ เมื่อเดินหน้า 2 ไมล์คือ 10^3 - 2 = 998 ซึ่งเราได้ตกลงกันแล้วบ่งชี้ว่ามีระยะลบ (ย้อนกลับ) ในทำนองเดียวกันใน Two's Complement ของ 4-bit คือ 0011_2 คือ 2^4 - 0011_2 = 10000_2 - 0011_2 = 1101_2

เมื่อตรวจสอบอย่างละเอียดก็จะพบว่า Two's Complement นั้นไม่มีอะไรมากไปกว่าคอมพลีเมนต์แล้วบวกเพิ่มด้วย 1 เมื่อต้องการค้นหาคอมพลีเมนต์เลขฐานสองเพียงแค่กลับบิตและบวก 1 สิ่งนี้จะทำให้การบวกและการลบง่ายขึ้นเช่นกัน เนื่องจากตัวลบ (จำนวนที่ถูกคอมพลีเมนต์แล้วบวกเพิ่ม) แต่ไม่ต้องมี ที่ end carry-around ให้ต้องกังวล เพียงแค่ยกเลิกการดำเนินการใดๆ ที่เกี่ยวข้องกับบิตลำดับสูง เช่นเดียวกับ one's complement แต่ Two's Complement จะใช้คอมพลีเมนต์ของจำนวนใดๆ ในขณะที่คอมพิวเตอร์ที่ใช้การแทนแบบนี้เพื่อแสดงตัวเลขติดลบ ก็จะเรียกว่า ระบบ Two's Complement หรือคอมพิวเตอร์ที่ใช้เลขคณิตแบบ Two's Complement สำหรับการแทนจำนวนที่เป็นบวกก็เหมือนกับการแทนแบบอื่นๆ ที่ต่างกันก็คือจำนวนที่ติดลบเท่านั้น ตัวอย่าง 2.20 แสดงแนวคิดเหล่านี้

ตัวอย่าง 2.20 ให้แสดงการแทน 23_{10}, -23_{10} และ -9_{10} ด้วยไบนารี 8 บิตโดยสมมติว่าคอมพิวเตอร์ใช้ Two's Complement

$$23_{10} = +(00010111_2) = 00010111_2$$
$$-23_{10} = -(00010111_2) = 11101000_2 + 1 = 11101001_2$$
$$-9_{10} = -(00001001_2) = 11110110_2 + 1 = 11110111_2$$

เนื่องจากการแทนตัวเลขบวกนั้นเหมือนกันกับ one's complement และ signed-magnitude กระบวนการของการบวกเลขฐานสองที่ทั้งสองจำนวนเป็นจะเหมือนกัน เปรียบเทียบตัวอย่าง 2.21 กับตัวอย่าง 2.17 และตัวอย่าง 2.10

ตัวอย่าง 2.21 บวก 01001111_2 กับ 00100011_2 โดยใช้ Two's complement

				1	1	1	1	⇐ carries	
	0	1	0	0	1	1	1	1	(79)
+	0	0	1	0	0	0	1	1	+ (35)
	0	1	1	1	0	0	1	0	(114)

เมื่อต้องแปลงจากไบนารีไปเป็นเลขฐานสิบ จำนวนที่เป็นบวกนั้นง่าย เช่นในการแปลง two's complement ของ 00010111_2 ให้เป็นฐานสิบ เพียงแปลงเลขฐานสองนี้เป็นเลขฐานสิบเพื่อให้ได้ 23 อย่างไรก็ตามการแปลงจำนวนเลขลบแบบ two's complement จะต้องใช้กระบวนการย้อนกลับที่คล้ายกับการแปลงจากฐานสิบเป็นเลขฐานสอง เมื่อ two's complement ของ 11110111_2 และต้องการรู้ว่ามีค่าเท่าไรในฐานสิบ เรารู้ว่านี่เป็นจำนวนลบ แต่ต้องจำไว้ว่าเป็นการแทนแบบ two's complement จะต้องกลับบิตก่อนแล้วจึงบวก 1 (หา

ส่วนเติมเต็มแล้วบวก 1) ผลลัพธ์คือ: $00001000_2 + 1 = 00001001_2$ นี่เทียบเท่ากับ 9 ในฐานสิบ อย่างไรก็ตาม หมายเลขเดิมที่เราเริ่มต้นมีค่าเป็นลบ ดังนั้นจึงลงท้ายด้วย -9 เป็นทศนิยมเทียบเท่า 11110111_2

สองตัวอย่างต่อไปนี้แสดงวิธีการบวก (และการลบด้วยเหตุนี้เนื่องจากเราลบตัวเลขด้วยการเพิ่มตรงกันข้าม) โดยใช้ two's complement

ตัวอย่าง 2.22 บวก 9_{10} กับ -23_{10} โดยใช้ two's complement

	0	0	0	0	1	0	0	1	(9)
+	1	1	1	0	1	0	0	1	+ (–23)
	1	1	1	1	0	0	1	0	-14_{10}

ให้ลองทำเป็นแบบฝึกหัด โดยให้ตรวจสอบว่า 11110010_2 นั้น -14_{10} โดย two's complement

ตัวอย่าง 2.23 ให้ค้นหาผลรวมของ 23_{10} และ -9_{10} ในโหมดไบนารีโดยใช้ two's complement

1 ←	1	1	1	1	1	1	1	1	⇐ carries	
	0	0	0	1	0	1	1	1	(23)	Discard carry
+	1	1	1	1	0	1	1	1	+ (–9)	
	0	0	0	0	1	1	1	0	14_{10}	

สำหรับการแทนด้วย two's complement การบวกจำนวนที่ติดลบสองตัว จะได้ผลลัพธ์เป็นจำนวนลบตามที่เราคาดหวัง

ตัวอย่าง 2.24 ค้นหาผลรวมของ 11101001_2 (-23) และ 11110111_2 (-9) โดยใช้ two's complement

1 ←	1	1	1	1	1	1	1		⇐ carries	
	1	1	1	0	1	0	0	1	(–23)	Discard carry
+	1	1	1	1	0	1	1	1	+ (–9)	
	1	1	1	0	0	0	0	0	(–32)	

ให้สังเกตว่าการละทิ้งตัวทดในตัวอย่าง 2.23 และ 2.24 นั้น ไม่ได้ทำให้เกิดผลลัพธ์ผิดพลาด การเกิดโอเวอร์โฟลว์ขึ้นหากมีการบวกสองจำนวนที่ไม่ติดลบ ผลลัพธ์ที่ได้เป็นลบ หรือถ้าบวกสองจำนวนที่มากกว่าศูนย์และผลลัพธ์นั้นเป็นบวก เป็นไปไม่ได้ที่จะมีการล้น การแทนแบบ two's complement ถ้าตัวโอเพอร์แรนด์ตัวหนึ่งเป็นบวกและอีกตัวเป็นลบ ก็จะถูกรวมรวมเข้าด้วยกัน

วงจรคอมพิวเตอร์อย่างง่ายสามารถตรวจจับการล้นโดยใช้กฎที่ง่ายต่อการจดจำ สังเกตเห็นได้ทั้งในตัวอย่าง 2.23 และ 2.24 ว่าการทดเข้าไปในบิตเครื่องหมาย (1 ซึ่งถูกทดมาจากตำแหน่งบิตก่อนหน้า เข้ามาในตำแหน่งของบิตเครื่องหมาย) เช่นเดียวกับที่ทดออกไปจากบิตเครื่องหมาย (ค่า 1 จะถูกทิ้ง) เมื่อการทดด้วยค่าที่เหมือนกันก็จะไม่เกิดการล้น แต่ถ้าแตกต่างกันตัวบ่งชี้โอเวอร์โฟลว์จะถูกตั้งค่าในหน่วยคำนวณทางคณิตศาสตร์เพื่อระบุผลลัพธ์ที่ไม่ถูกต้อง

กฎง่ายๆ สำหรับการตรวจสอบการล้นในบิตเครื่องหมาย: ถ้ามีการทดเข้าสู่บิตเครื่องหมายเท่ากับที่ทดออกจากบิตเครื่องหมายจะไม่มีการล้นเกิดขึ้น หากมีการทดเข้าไปในบิตเครื่องหมายต่างจากที่ทดออกจากบิตเครื่องหมายจะมีการล้นเกิดขึ้น (และทำให้เกิดข้อผิดพลาด)

ส่วนที่ยากคือโปรแกรมเมอร์ (หรือคอมไพเลอร์) ต้องตรวจสอบสภาพล้นอย่างสม่ำเสมอ ตัวอย่างที่ 2.25 บ่งชี้ว่ามีการล้นเพราะการทดเข้าสู่บิตเครื่องหมาย (มีการทด 1 เข้า) ซึ่งไม่เท่ากับการทดออก (มีการทด 0 ออก)

ตัวอย่าง 2.25 ให้หาผลรวมของ 126_{10} และ 8_{10} เป็นเลขฐานสองโดยใช้ two's complement

	0 ←	1	1	1	1				⇐ carries
		0	1	1	1	1	1	1	0 (126)
Discard last carry	+	0	0	0	0	1	0	0	0 + (8)
		1	0	0	0	0	1	1	0 (−122???)

1 ถูกทดเข้าไปที่บิตซ้ายสุด แต่มีการทด 0 ออกจากบิตเครื่องหมาย เนื่องจากการทดนี้ มีค่าไม่เท่ากันจึงเกิดการล้น (เราสามารถสังเกตเห็นได้อย่างง่าย จากกรณีที่โอเพอแรนด์ทั้งสองเป็นบวก แต่ผลลัพธ์เป็นลบ) จะกล่าวถึงเรื่องนี้อีกครั้งในหัวข้อ 2.4.6

Two's complement เป็นการแทนแบบคิดเครื่องหมายที่ได้รับความนิยมมากที่สุด อัลกอริธึมสำหรับการบวกและการลบนั้นสามารถดำเนินการได้ง่าย สามารถแทนได้ดีที่สุดสำหรับ 0 (ทั้งหมด 0 บิต) และการแปลงกลับในตัวเอง และสามารถขยายออกไปเป็นบิตจำนวนมากขึ้น มีข้อเสียคือความไม่สมดุลของพิสัย N บิต เช่น N=4 บิต One’s complement และ Signed-magnitude สามารถแทนค่า - 7 ถึง +7 อย่างไรก็ตามการใช้ Two's complement นั้นเราสามารถแทนค่าได้ -8 ถึง +7 ซึ่งมักจะสับสนกับการแทนแบบคอมพลีเมนต์ เหตุผลว่าทำไม +7 จึงเป็นจำนวนมากที่สุดที่สามารถใช้แทนการคอมพลีเมนต์ 4 บิต การแทนแบบ Two's complement ต้องจำไว้ว่าบิตแรกต้องเป็น 0 ถ้าบิตที่เหลือเป็น 1s ทั้งหมด (ทำให้มีค่ามากที่สุดเท่าที่จะเป็นไปได้) 0111_2 ซึ่งก็คือ 7

ในทางตรงกันข้าม ค่าที่น้อยที่สุดเป็นค่าลบควรมีค่าเท่ากับ 1111_2 อันที่จริง 1111_2 มีค่าเท่ากับ -1 (0000+1=-1) แล้ว -8 แทนได้อย่างไรใน Two's complement ที่ใช้ 4 บิต? มันถูกแทนด้วย 1000_2 นี่เป็นจำนวนลบ ถ้ากลับบิต (0111+1=-8)

การคูณและการหารจำนวนเต็ม

หากไม่มีการใช้อัลกอริทึมที่ซับซ้อน การคูณและการหาร จะต้องวนซ้ำหลายรอบเพื่อคำนวณผลลัพธ์ ในที่นี่จะพูดถึงวิธีที่ตรงไปตรงมาที่สุดในการดำเนินการเหล่านี้ ในระบบจริงฮาร์ดแวร์เฉพาะถูกใช้เพื่อเพิ่มประสิทธิภาพ บางครั้งดำเนินการบางส่วนของการคำนวณแบบขนาน ผู้อ่านที่อยากรู้อยากเห็นจะต้องตรวจสอบวิธีการขั้นสูงเหล่านี้ได้จากเอกสารการอ้างอิงที่อ้างถึงในตอนท้ายของบทนี้

อัลกอริทึมการคูณที่ง่ายที่สุดที่ใช้โดยคอมพิวเตอร์นั้น คล้ายกับวิธีการแบบดินสอและกระดาษที่มนุษย์ใช้ ตารางการคูณที่สมบูรณ์ สำหรับเลขฐานสองนั้นไม่สามารถทำให้ง่ายขึ้นได้: ศูนย์คูณจำนวนใดก็ได้ศูนย์ และหนึ่งคูณตัวเลขใดๆ ก็คือตัวเลขนั้น

เพื่อแสดงการคูณคอมพิวเตอร์อย่างง่าย จะเริ่มต้นด้วยการเขียนตัวตั้งหรือตัวถูกคูณกับตัวคูณไปยังพื้นที่เก็บข้อมูลสองแห่งแยกกัน เรายังต้องการพื้นที่เก็บข้อมูลที่สามสำหรับผลคูณ เริ่มจากบิตลำดับต่ำ ตัวชี้จะถูกตั้งค่าเป็นตัวเลขแต่ละหลักของตัวคูณ สำหรับแต่ละหลักของตัวคูณ จะถูกเลื่อนไปทางซ้ายหนึ่งบิตของตัวตั้ง เมื่อตัวคูณคือ 1 ก็จะยกตัวตั้งมาวางที่ตำแหน่งของมันเช่นเดียวกับการคูณฐานสิบ เพื่อให้หาผลรวมสะสมของการคูณ เนื่องจากจะต้องเลื่อนตัวคูณไปทีละบิต การคูณแต่ละตัวคูณต้องใช้พื้นที่การทำงานของตัวตั้งหรือตัวคูณเพิ่มขึ้นเป็นสองเท่า

มีวิธีการง่ายๆสองวิธีในการหารไบนารี: เราสามารถลบตัวตั้งซ้ำจากตัวหารได้ หรือเราสามารถใช้วิธีการลองผิดลองถูกแบบเดียวกันของการหารยาวที่สอนในโรงเรียนประถมศึกษา เช่นเดียวกับการคูณ วิธีที่มีประสิทธิภาพที่สุดที่ใช้สำหรับการหารไบนารี แต่อยู่นอกเหนือขอบเขตของบทนี้ และสามารถอ่านเพิ่มเติมได้ในเอกสารอ้างอิงในตอนท้ายของบทนี้

เมื่อไม่คำนึงถึงประสิทธิภาพของอัลกอริธึม การหารคือการดำเนินการที่สามารถทำให้คอมพิวเตอร์หยุดการทำงานได้เสมอ นี่เป็นกรณีเฉพาะเมื่อพยายามหารด้วยศูนย์ หรือเมื่อโอเพอแรนด์มีขนาดที่แตกต่างกันมาก เมื่อ

ตัวหารมีขนาดเล็กกว่าตัวตั้ง เราจะได้รับเงื่อนไขที่เรียกว่าการหารอันเดอร์โฟลว์(divide underflow) ซึ่งคอมพิวเตอร์มองว่าเทียบเท่ากับการหารด้วยศูนย์ซึ่งเป็นไปไม่ได้

คอมพิวเตอร์มีความแตกต่างระหว่างการหารจำนวนเต็มและการหารทศนิยม การหารจำนวนเต็มคำตอบมาในสองส่วนคือผลหาร(quotient) และเศษเหลือ(remainder) การหารทศนิยมส่งผลให้ตัวเลขที่แสดงเป็นเศษส่วนไบนารี การหารทั้งสองประเภทมีความแตกต่างกัน จึงมีการออกแบบวงจรพิเศษสำหรับแต่ละประเภท การคำนวณเลขทศนิยมจะดำเนินการในวงจรเฉพาะที่เรียกว่า floating-point units หรือ FPU

2.4.4 จำนวน Unsigned เทียบกับ Signed

เราแนะนำการสนทนาของเราเกี่ยวกับเลขจำนวนเต็มไบนารีที่มีตัวเลขที่ไม่ได้ลงนาม หมายเลขที่ไม่ได้ลงชื่อใช้เพื่อแสดงค่าที่รับประกันว่าจะไม่เป็นค่าลบ ตัวอย่างที่ดีของหมายเลขที่ไม่ได้ลงชื่อคือที่อยู่หน่วยความจำ หากค่าเลขฐานสอง 4 บิต 1101 ไม่ได้ลงนามแสดงว่าเป็นเลขฐานสิบ 13 แต่เป็นหมายเลขส่วนเติมของสองที่ลงนามแล้วจะหมายถึง -3 หมายเลขที่เซ็นชื่อใช้เพื่อแสดงข้อมูลที่อาจเป็นบวกหรือลบ

โปรแกรมเมอร์คอมพิวเตอร์จะต้องสามารถจัดการทั้งตัวเลขที่ลงนามและไม่ได้ลงนาม ในการทำเช่นนั้นโปรแกรมเมอร์จะต้องระบุค่าตัวเลขก่อนว่าเป็นตัวเลขที่ลงนามหรือไม่ได้ลงนาม สิ่งนี้ทำได้โดยการประกาศค่าเป็นประเภทเฉพาะ ตัวอย่างเช่นภาษาการเขียนโปรแกรม C มี int และไม่ได้ลงนาม int เป็นประเภทที่เป็นไปได้สำหรับตัวแปรจำนวนเต็มกำหนดจำนวนเต็มลงนามและไม่ได้ลงนามตามลำดับ นอกเหนือจากการประกาศประเภทที่แตกต่างกันหลายภาษามีการดำเนินการทางคณิตศาสตร์ที่แตกต่างกันสำหรับใช้กับตัวเลขที่ลงนามและไม่ได้ลงนาม ภาษาอาจมีคำแนะนำการลบหนึ่งคำสำหรับหมายเลขที่เซ็นชื่อและคำแนะนำการลบแบบอื่นสำหรับหมายเลขที่ไม่ได้ลงชื่อ ในภาษาแอสเซมบลีส่วนใหญ่โปรแกรมเมอร์สามารถเลือกจากโอเปอเรเตอร์การเปรียบเทียบที่ลงนามแล้วหรือโอเปอเรเตอร์การเปรียบเทียบที่ไม่ได้ลงชื่อ

เป็นที่น่าสนใจที่จะเปรียบเทียบสิ่งที่เกิดขึ้นกับหมายเลขที่ไม่ได้ลงชื่อและเซ็นชื่อเมื่อเราพยายามจัดเก็บค่าที่มีขนาดใหญ่เกินไปสำหรับจำนวนบิตที่ระบุ ตัวเลขที่ไม่ได้ลงชื่อเพียงแค่ล้อมรอบและเริ่มต้นใหม่ที่ 0 ตัวอย่างเช่นถ้าเราใช้เลขฐานสองแบบไม่ได้ลงนาม 4 บิตและเราเพิ่ม 1 ถึง 1111 เราได้รับ 0000 คำสั่ง "กลับไปที่ 0" นี้คุ้นเคย - บางทีคุณอาจเคยเห็น รถยนต์ระยะทางสูงที่ฐานเครื่องวัดระยะทางห่อกลับมาอยู่ที่ 0 อย่างไรก็ตามตัวเลขที่ลงนามนั้นอุทิศพื้นที่ครึ่งหนึ่งของพวกเขาให้เป็นจำนวนบวกและอีกครึ่งหนึ่งเป็นตัวเลขติดลบ ถ้าเราบวก 1

เข้ากับเลขเสริมบวกที่ใหญ่ที่สุดของ 4 บิตสอง 0111 (+7) เราจะได้ 1,000 (-8) การขึ้นบรรทัดใหม่ด้วยการเปลี่ยนแปลงเครื่องหมายที่ไม่คาดคิดนี้เป็นปัญหาสำหรับโปรแกรมเมอร์ที่ไม่มีประสบการณ์ทำให้เวลาดีบักหลายชั่วโมง โปรแกรมเมอร์ที่ดีเข้าใจเงื่อนไขนี้และวางแผนที่เหมาะสมในการจัดการกับสถานการณ์ก่อนที่จะเกิดขึ้น

2.5 การแทนจุดทศนิยม

ในการออกแบบคอมพิวเตอร์ มีการแทนค่าจำนวนเต็มใดๆ ที่เพิ่งศึกษาไป ขั้นตอนต่อไปคือการตัดสินใจขนาดของคำหรือเวิร์ดของระบบ หากต้องการให้ระบบมีราคาไม่แพงจะเลือกเวิร์ดโขนาดเล็ก 16 บิต อนุญาตให้บิตมีบิตเครื่องหมาย จำนวนเต็มที่มากที่สุดที่ระบบนี้เก็บได้คือ 32,767 แต่ถ้าต้องการให้เก็บจำนวนเติมได้มากกว่านั้นก็ลองทำให้ขนาดเวิร์ดใหญ่ขึ้นเป็น 32 บิต ตอนนี้เวิร์ดของเรามีขนาดใหญ่พอสำหรับทุกสิ่งที่ทุกคนต้องการนับ แต่ถ้าต้องการแทนตัวเลขเศษส่วนทศนิยม ก็ต้องมาดูวิธีการแทนทศนิยมกัน

วิธีที่ง่ายที่สุดสำหรับปัญหานี้คือการรักษาระบบ 16 บิตไว้และพูดว่า "เฮ้เรากำลังสร้างระบบราคาถูกที่นี่ถ้าคุณต้องการทำสิ่งต่างๆ ด้วยตัวคุณเองจงเป็นโปรแกรมเมอร์ที่ดี" แม้ว่าจะฟังดูโอ้อวดในบริบทของเทคโนโลยีในปัจจุบัน แต่มันก็เป็นความจริงในยุคแรกๆ ของคอมพิวเตอร์แต่ละรุ่น ไม่มีสิ่งใดที่ต้องแทนด้วยทศนิยมในเมนเฟรมหรือไมโครคอมพิวเตอร์เครื่องแรก หลายปีที่ผ่านมาการเขียนโปรแกรมอย่างชาญฉลาดทำให้ระบบจำนวนเต็มเหล่านี้ทำหน้าที่เสมือนว่าเป็นระบบจุดทศนิยม

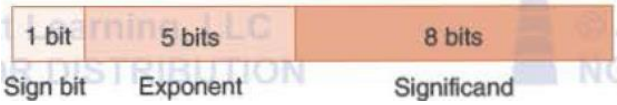
หากคุ้นเคยกับสัญกรณ์ทางวิทยาศาสตร์ อาจกำลังคิดว่าคุณสามารถจัดการกับการดำเนินการเกี่ยวกับ floating-point ได้อย่างไร - คุณจะมีการจำลอง floating-point ในระบบจำนวนเต็มได้อย่างไร ในสัญกรณ์ทางวิทยาศาสตร์ตัวเลขจะถูกแสดงเป็นสองส่วน: ส่วนที่เป็นเศษส่วนและส่วนของตัวยกกำลัง ซึ่งส่วนเศษควรจะยกระดับเพื่อให้สามารถเก็บค่าที่ต้องการได้ ดังนั้นเพื่อแสดง 32,767 ในสัญกรณ์ทางวิทยาศาสตร์ เราสามารถเขียน 3.2767×10^4 สัญกรณ์วิทยาศาสตร์หรือ Scientific number ทำให้การคำนวณด้วยดินสอและกระดาษง่ายขึ้น ซึ่งเกี่ยวข้องกับตัวเลขที่มีขนาดใหญ่มากหรือน้อยมาก นอกจากนี้ยังเป็นพื้นฐานสำหรับการคำนวณเลขทศนิยมในคอมพิวเตอร์ดิจิทัลในปัจจุบัน

2.5.1 โมเดลพื้นฐาน

ในคอมพิวเตอร์ดิจิทัล ตัวเลขจุดทศนิยมประกอบด้วยสามส่วน: บิตเครื่องหมาย ส่วนเลขชี้กำลัง (แทนเลขชี้กำลังด้วยกำลัง 2) และส่วนที่เป็นเศษส่วน (ซึ่งมาดูคำศัพท์ที่เหมาะสม) คำว่า mantissa เป็นที่ยอมรับอย่าง

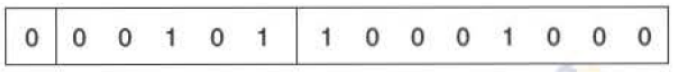
กว้างขวางเมื่ออ้างถึงส่วนที่เป็นเศษส่วน อย่างไรก็ตามหลายคนใช้ข้อยกเว้นของ mantissa เพราะมันยังหมายถึงเศษส่วนของลอการิทึม ซึ่งไม่เหมือนกับส่วนที่เป็นเศษส่วนของจำนวนจุดทศนิยม สถาบันวิศวกรไฟฟ้าและอิเล็กทรอนิกส์ (IEEE: Institute of Electrical and Electronics Engineers) แนะนำคำสำคัญและอ้างถึงส่วนที่เป็นเศษส่วนของจำนวนจุดทศนิยมรวมกับจุดฐานสองโดยนัย นัยแรก (ซึ่งเราจะกล่าวถึงในตอนท้ายของส่วนนี้) น่าเศร้าที่ข้อตกลง mantissa และซิกนิฟิแคนด์(significand) กลายเป็นสิ่งที่ใช้แทนกันได้ เมื่ออ้างถึงส่วนที่เป็นเศษส่วนของจำนวนจุดทศนิยมแม้ว่าจะไม่เทียบเท่าในทางเทคนิคก็ตาม ในที่นี้จะอ้างถึงส่วนที่เป็นเศษส่วนโดยไม่คำนึงว่าจะรวมถึงนัยแรก ตามที่ IEEE ตั้งใจไว้หรือไม่ (ดูหัวข้อ 2.5.4)

จำนวนบิตที่ใช้สำหรับเลขชี้กำลังและซิกนิแคนด์นั้น ขึ้นอยู่กับว่าเราต้องการปรับให้เหมาะสมสำหรับช่วง (บิตเพิ่มเติมในเลขชี้กำลัง) หรือความแม่นยำ (เพิ่มบิตเติมในซิกนิแคนด์แทน) (จะพูดถึงช่วงและความแม่นยำเพิ่มเติมในหัวข้อ 2.5.7.) สำหรับเศษเหลือ(remainder) ในที่นี้จะใช้โมเดล 14 บิต ที่มีเลขยกกำลัง 5 บิต 8 บิต แทนซิกนิฟิแคนด์ และบิตเครื่องหมายอีกหนึ่งบิต (ดูรูปที่ 2.1) แบบฟอร์มทั่วไปมีอธิบายไว้ในส่วน 2.5.2

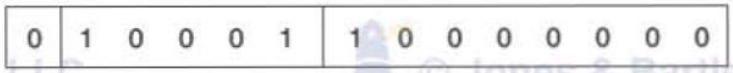


รูปที่ 2.1 แบบจำลองที่ให้แทนเลขทศนิยม

สมมติว่าเราต้องการเก็บเลขทศนิยมที่มีค่าเท่ากับ 17 ไว้ในแบบจำลอง เมื่อ $17 = 17.0 \times 10^{0} = 1.7 \times 10^{1} = 0.17 \times 10^{2}$ เมื่อเทียบกับไบนารี $17_{10} = 10001_2 \times 2^0 = 1000.1_2 \times 2^1 = 100.01_2 \times 2^2 = 10.001_2 \times 23 = 1.0001_2 \times 2^4 = 0.10001_2 \times 2^5$ ถ้าใช้แบบฟอร์มสุดท้ายเศษส่วนของเราจะเป็น 10001000 และเลขชี้กำลังของเราจะ เป็น 00101 ดังแสดงที่นี่:



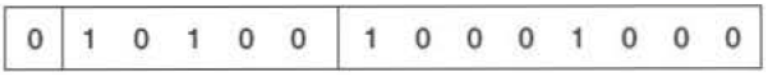
การใช้แบบฟอร์มนี้ เราสามารถเก็บจำนวนความสำคัญได้มากกว่าการแทนจุดคงที่ 14 บิต (ซึ่งใช้จำนวน 14 หลักไบนารี รวมทั้งไบนารีหรือฐาน) หากเราต้องการแทน $65536 = 0.1_2 \times 2^{17}$ ในแบบจำลองทศนิยม 14 บิต จะได้ว่า



ปัญหาหนึ่งที่เห็นได้ชัดของแบบจำลองนี้คือ ไม่ได้ให้เลขชี้กำลังเป็นลบ หากต้องการเก็บ 0.25 จะไม่มีทางทำเช่นนั้น เพราะ $0.25 = 2^{-2}$ และไม่สามารถแทนเลขชี้กำลัง -2 ได้ การแก้ไขปัญหาโดยการเพิ่มเครื่องหมายบิตไปยังเลขชี้กำลัง แต่ปรากฏว่ามีการแทนที่มีประสิทธิภาพมากกว่า สำหรับการใช้เลขชี้กำลังแบบไบอัส (Bias) เพราะสามารถใช้วงจรจำนวนเต็มออกแบบง่ายกว่า โดยเฉพาะสำหรับตัวเลขที่ไม่แทนเครื่องหมาย เมื่อเปรียบเทียบค่าของเลขทศนิยมสองตัว

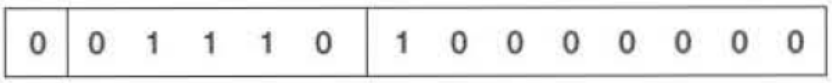
จากหัวข้อ 2.4.3 แนวคิดการใช้ค่าไบแอส คือการแปลงจำนวนเต็มทุกตัวในช่วง ให้เป็นจำนวนเต็มที่ไม่ใช่ค่าลบ ซึ่งจะถูกเก็บไว้เป็นเลขฐานสอง จำนวนเต็มในช่วงเลขชี้กำลังที่ต้องการปรับครั้งแรก โดยการเพิ่มค่า bias คงที่นี้ให้กับแต่ละเลขชี้กำลัง ค่า bias คือตัวเลขที่อยู่ใกล้ช่วงกลางของช่วงของค่าที่เป็นไปได้ที่เลือก เพื่อเป็นตัวแทน 0 ในกรณีนี้จะเลือก 15 เพราะมันอยู่ตรงกลางระหว่าง 0 ถึง 31 (เลขชี้กำลังของมี 5 บิตดังนั้นอนุญาตให้ 25 หรือ 32 ค่า) จำนวนใดๆ ที่มากกว่า 15 ในฟิลด์เลขชี้กำลังแสดงถึงค่าบวก ค่าน้อยกว่า 15 แสดงถึงค่าลบ สิ่งนี้เรียกว่าการแทนค่าส่วนเกิน -15 เพราะเราต้องลบ 15 เพื่อให้ได้มูลค่าที่แท้จริงของเลขชี้กำลัง โดยปกติแล้วเลขยกกำลังของ 0s หรือ 1s ทั้งหมด จะถูกสงวนไว้สำหรับหมายเลขพิเศษ (เช่น 0 หรืออินฟินิตี้) ในรูปแบบง่าย ๆ ของเราเราอนุญาตเลขชี้กำลังของ 0s และ 1s ทั้งหมด

กลับไปที่ตัวอย่างของการจัดเก็บ 17 ที่คำนวณไว้ $17_{10} = 0.10001_2 \times 2^5$ ถ้าปรับเป็นโมเดลที่ใช้ bias เลขชี้กำลังแบบ bias คือ 15 + 5 = 20:

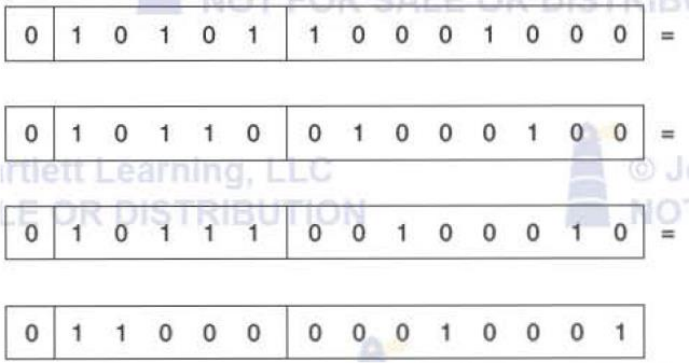


ในการคำนวณค่าของตัวเลขนี้ จะใช้ฟิลด์เลขชี้กำลัง (ค่าฐานสองเท่ากับ 20) และลบ bias (20 - 4) จะได้เลขชี้กำลัง "ของจริง" เป็น 5 ซึ่งส่งผลให้ 0.10001000×2^5

หากเราต้องการเก็บ $0.25 = 0.1 \times 2^{-1}$ เราจะได้:



ยังคงมีปัญหาหนึ่งที่ค่อนข้างใหญ่สำหรับระบบนี้: เนื่องจากไม่ได้มีตัวแทนที่เป็นเอกลักษณ์สำหรับแต่ละหมายเลขทั้งหมดต่อไปนี้เทียบเท่า:

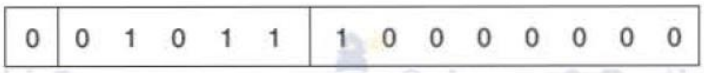


เนื่องจากรูปแบบที่เหมือนกันซึ่งไม่เหมาะสำหรับคอมพิวเตอร์ดิจิทัล ตัวเลขทศนิยมจะต้องเป็นมาตรฐาน - นั่นคือ บิตสุดท้ายของซิกนิฟิแคนด์และต้องเป็น 1 เสมอ กระบวนการนี้เรียกว่าการทำให้เป็นมาตรฐาน(normalization) อนุสัญญานี้มีข้อได้เปรียบ หาก 1 ถูกบอกเป็นนัย จะได้รับความแม่นยำเพิ่มขึ้นเล็กน้อยในซิกนิแคนด์ การทำให้เป็นมาตรฐานทำงานได้ดีสำหรับทุกค่ายกเว้น 0 ซึ่งไม่มีบิตที่ไม่ใช่ศูนย์ ด้วยเหตุผลดังกล่าวโมเดลใดๆ ที่ใช้เพื่อแสดงตัวเลขทศนิยมจะต้องปฏิบัติต่อ 0 เป็นกรณีพิเศษ จะเห็นในหัวข้อถัดไปว่ามาตรฐานทศนิยม IEEE-754 ทำให้เกิดข้อยกเว้นสำหรับกฎการทำให้เป็นมาตรฐาน

ตัวอย่าง 2.34 ให้แสดง 0.03125_{10} ในรูปแบบจุดทศนิยมมาตรฐาน โดยใช้แบบจำลองอย่างง่ายที่มี bias เกิน 15

$0.03125_{10} = 0.00001_2 \times 2^0 = 0.0001 \times 2^{-1} = 0.001 \times 2^{-2} = 0.01 \times 2^{-3} = 0.1 \times 2^{-4}$

เมื่อใช้ไบอัสฟิลด์เลขชี้กำลังคือ 15 - 4 = 11

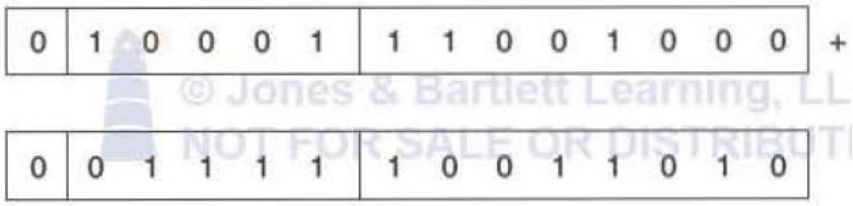


โดยทั่วไปรูปแบบที่เรียบง่าย ยังไม่ได้แสดงจำนวนโดยใช้การทำให้เป็นมาตรฐานสัญกรณ์ที่บอกถึง 1 ซึ่งจะได้กล่าวถึงในหัวข้อ 2.5.4

2.5.2 เลขคณิตของทศนิยม

หากต้องการบวกตัวเลขทศนิยมสองตัวที่แทนด้วย scientific number เช่น $1.5 \times 10^2 + 3.5 \times 10^3$ เราจะเปลี่ยนจำนวนหนึ่งของโอเพอแรนด์ เพื่อให้ทั้งคู่แสดงในฐานตัวชีกำลังเดียวกัน ในตัวอย่าง $1.5 \times 10^2 + 3.5 \times 10^3 = 0.15 \times 10^3 + 3.5 \times 10^3 = 3.65 \times 103$ การเติมจุดทศนิยมและการลบทำงานในลักษณะเดียวกันตามที่แสดงด้านล่าง

ตัวอย่าง 2.35 ให้บวกเลขฐานสองต่อไปนี้ตามที่แสดงในรูปแบบ 14 บิตปกติโดยใช้แบบจำลองอย่างง่ายที่มี bias 15



จะเห็นว่าตัวตั้งปรับตัวยกกำลังสอง และตัวบวก(augend) มีกำลังเท่ากับ 0 กำลัง จัดแนวของตัวถูกดำเนินการทั้งสองนี้บนจุดฐานสองจะได้ว่า

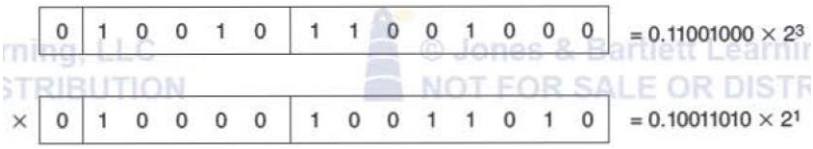
$$\begin{array}{r} 11.001000 \\ +\,0.10011010 \\ \hline 11.10111010 \end{array}$$

การทำให้อยู่ในรูปปกติ ซึ่งยังคงมีเลขชี้กำลังใหญ่ขึ้นและตัดบิตลำดับต่ำ จะได้ว่า

เนื่องจากโมเดลแบบง่าย ต้องการนัยสำคัญที่ทำให้เป็นมาตรฐาน จึงไม่มีวิธีที่จะแทน 0 ซึ่งสามารถแก้ไขได้ง่ายโดยยอมให้สตริงทั้งหมด 0 (เครื่องหมาย 0, 0 เลขชี้กำลังและ 0 ซิกนิฟิคันด์) เพื่อแทนค่า 0 ในหัวข้อถัดไปจะเห็นว่า IEEE-754 ยังสงวนความหมายพิเศษสำหรับรูปแบบบิตบางอย่าง

การคูณและการหารนั้นดำเนินการโดยใช้กฎเลขชี้กำลังแบบเดียวกับที่ใช้กับเลขทศนิยมเช่น $2^{-3} \times 2^{4} = 2^{1}$ เป็นต้น

ตัวอย่าง 2.36 สมมติว่ามี bias 15 บิตคูณด้วย:



การคูณ 0.11001000 ด้วย 0.10011010 ให้ผลคูณเป็น 0.0111100001010000 แล้วคูณด้วย $2^3 \times 2^1 = 2^4$ ซึ่งเท่ากับ 111.10000101 การปรับให้เป็นรูปแบบปกติและจัดเลขชี้กำลังที่เหมาะสม ผลคูณ floating-point คือ:

2.5.3 ข้อผิดพลาดของ Floating-Point

เมื่อคำนวณด้วยดินสอและกระดาษ การแก้ปัญหาตรีโกณมิติหรือคำนวณดอกเบี้ยจากการลงทุน จะเข้าใจอย่างถ่องแท้ว่าเรากำลังทำงานในระบบจำนวนจริง(Real number) เรารู้ว่าระบบนี้มีค่าไม่มีที่สิ้นสุด เพราะด้วยจำนวนจริงใดๆ เราสามารถหาจำนวนจริงอีกจำนวนหนึ่งที่เล็กกว่าได้เสมอ

คอมพิวเตอร์เป็นระบบจำกัด มีที่เก็บข้อมูลจำกัด ซึ่งแตกต่างจากคณิตศาสตร์ในจินตนาการ เมื่อต้องการให้คอมพิวเตอร์ทำการคำนวณจำนวนจริง ซึ่งเป็นการจำลองระบบจำนวนอนันต์ของจำนวนจริงในระบบจำกัดจำนวนเต็ม ในความเป็นจริงสิ่งที่เรามีคือการประมาณของระบบจำนวนจริง ยิ่งใช้จำนวนมากเท่าไหร่การประมาณก็จะยิ่งดีขึ้นเท่านั้น อย่างไรก็ตามมีข้อผิดพลาดบางส่วนอยู่เสมอไม่ว่าจะใช้กี่บิตก็ตาม

ข้อผิดพลาดที่เกิดจากจุดทศนิยมอาจมีความละเอียดอ่อน บางครั้งไม่มีใครสังเกตเห็นได้ ข้อผิดพลาดที่โจ่งแจ้งเช่นตัวเลขล้นหรืออันเดอร์โฟลว์เป็น 1 วินาทีที่ทำให้โปรแกรมขัดข้อง ข้อผิดพลาดเล็กน้อยอาจนำไปสู่ผลลัพธ์ที่ผิดพลาดอย่างรุนแรง ซึ่งมักจะตรวจจับได้ยากก่อนที่จะเกิดปัญหาจริง ตัวอย่างในโมเดลง่ายๆ ของเราสามารถแสดงตัวเลขมาตรฐานในช่วง -0.11111 1112 x 2^{16} ถึง +0.11111111 x 2^{16} เห็นได้ชัดว่าเราไม่สามารถเก็บ 2^{-19} หรือ 2^{128}; เนื่องจากระดับความละเอียดไม่พอ ทำให้ไม่สามารถจัดเก็บ 128.5 ได้อย่างแม่นยำซึ่งอยู่ในขอบเขตของเรา การแปลง 128.5 เป็นไบนารี่จะได้ 10000000.1 ใช้ 9 บิต ระดับนัยสำคัญสามารถทำได้เพียงแปดบิต โดยทั่วไปบิตในลำดับต่ำถูกปล่อยหรือปัดเศษลงในบิตถัดไป อย่างไรก็ตามไม่ว่าจะจัดการอย่างไร ก็จะเกิดข้อผิดพลาดในระบบของเรา

ในการคำนวณข้อผิดพลาดสัมพัทธ์ เป็นการแทนโดยการหาอัตราส่วนของค่าสัมบูรณ์ของข้อผิดพลาดต่อค่าจริงของตัวเลข จากการใช้ตัวอย่าง 128.5 จะพบว่า:

$$\frac{128.5 - 128}{128.5} = 0.00389105 \circ 0.39\%.$$

หากไม่ระวังข้อผิดพลาดดังกล่าว สามารถแพร่กระจายผ่านการคำนวณที่ยาว ทำให้สูญเสียความแม่นยำอย่างมาก ตารางที่ 2.3 แสดงการกระจายข้อผิดพลาดเมื่อเราคูณ 16.24 ด้วย 0.91 ซ้ำโดยใช้แบบจำลอง 14 บิตแบบง่ายของเรา เมื่อแปลงตัวเลขเหล่านี้เป็นไบนารี่ 8 บิต จะเห็นว่ามีข้อผิดพลาดมากมายตั้งแต่เริ่มแรก

อย่างที่เห็นในการทำซ้ำหกครั้ง มีข้อผิดพลาดมากกว่าสามเท่าในผลลัพธ์ การวนซ้ำอย่างต่อเนื่องจะทำให้เกิดข้อผิดพลาด 100% เนื่องจากผลลัพธ์เข้าสู่ 0 ในที่สุดแม้ว่า 14 บิตนี้ มีขนาดเล็กจนเกินความผิดพลาด แต่

ระบบทศนิยมทั้งหมด ทำงานในลักษณะเดียวกัน มีข้อผิดพลาดในระดับหนึ่งที่เกี่ยวข้องเสมอ เมื่อแสดงตัวเลขจริงในระบบจำกัด ไม่ว่าจะสร้างระบบนั้นดีเพียงใด แม้แต่ข้อผิดพลาดที่เล็กที่สุด ก็อาจมีผลเป็นความหายนะ โดยเฉพาะอย่างยิ่งเมื่อคอมพิวเตอร์ถูกใช้เพื่อควบคุมเหตุการณ์ทางกายภาพ เช่นในทางการทหาร และการแพทย์ ความท้าทายสำหรับนักวิทยาศาสตร์คอมพิวเตอร์ คือการค้นหาอัลกอริธึมที่มีประสิทธิภาพสำหรับการควบคุมข้อผิดพลาดภายในขอบเขตของประสิทธิภาพและเศรษฐศาสตร์