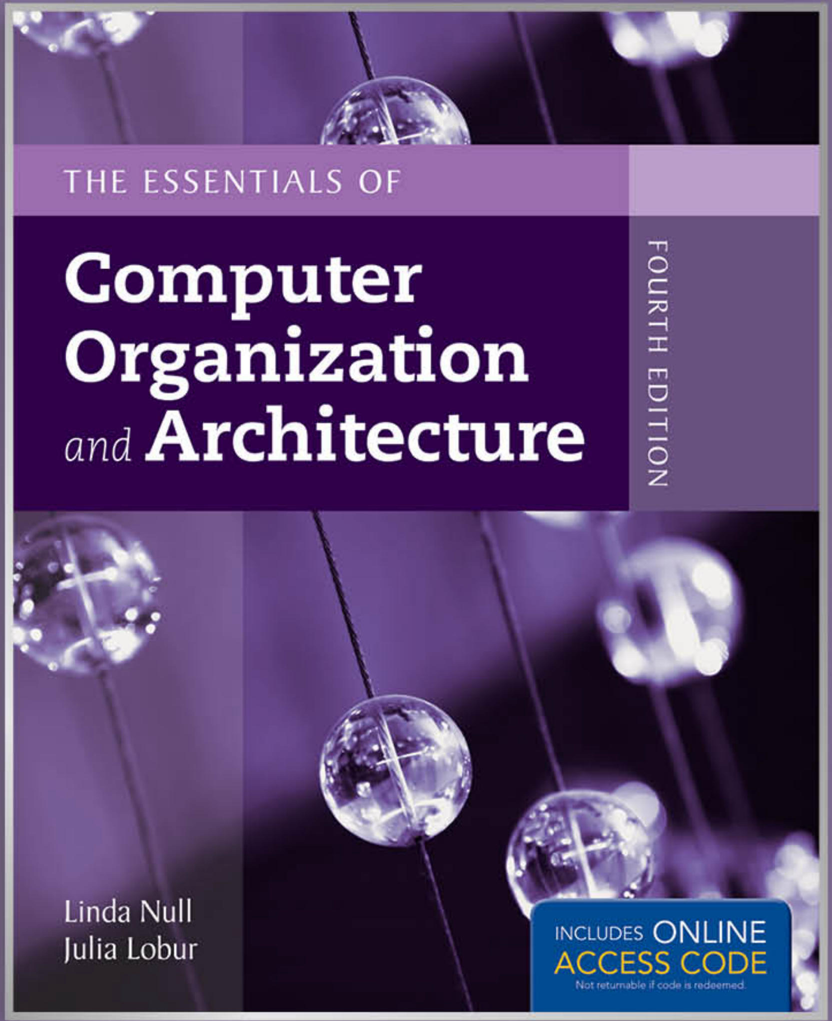
Chapter 2

Data Representation in Computer Systems



2.4 Signed Integer Representation

- The conversions we have so far presented have involved only unsigned numbers.
- To represent **signed integers**, computer systems allocate the **leftmost bit** to indicate the **sign** of a number.
  - The high-order bit is the leftmost bit. It is also called the most significant bit.
  - **0** is used to indicate a **positive number**; **1** indicates a **negative number**.
- The **remaining bits** contain the **value** of the number (but this can be interpreted different ways)

2.4 Signed Integer Representation

- There are three common ways in which **signed binary integers** may be expressed:
  - **Signed magnitude**
  - **One's complement**
  - **Two's complement**

- In an 8-bit word, ***signed magnitude*** representation places the **absolute value (magnitude)** of the number in the **7 bits to the right of the sign bit**.

2.4 Signed Integer Representation

- For example, in 8-bit **signed magnitude** representation:

+3 is: `00000011`

- 3 is: `10000011`

- Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.
  - **ignore the signs of the operands during the calculation, apply the appropriate sign afterwards**.

2.4 Signed Integer Representation

- **Binary addition rules for bits:**

```
0 + 0 =  0     0 + 1 =  1
1 + 0 =  1     1 + 1 = 10
```

- The simplicity of this system makes it possible for digital circuits to carry out arithmetic operations.

Let's see how the addition rules work with signed magnitude numbers . . .

2.4 Signed Integer Representation

- Example:
  – Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Convert 75 and 46 **to binary**, and **arrange as a sum**, **separate** the **sign** bits **from magnitude** bits.

```
0     1001011
0  +  0101110
      -------
```

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Find the sum **starting with the rightmost bits** and work left.

```
0    1001011
0  + 0101110
   ---------
           1
```

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- In the second bit, we have a **carry**, so we note it above the third bit.

```
          1
0   1001011
0 + 0101110
  ---------
         01
```

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- The third and fourth bits also give us carries.

```
          1 1 1
0     1 0 0 1 0 1 1
0  +  0 1 0 1 1 1 0
   -----------------
            1 0 0 1
```

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Once we have worked our way through all eight bits, we are done.

```
          1 1 1
0     1 0 0 1 0 1 1
0  +  0 1 0 1 1 1 0
   ----------------
0     1 1 1 1 0 0 1
```

In this example, we were careful to pick two values whose sum would fit into seven bits.

If that is not the case, we have a problem: *overflow*

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 107 and 46.
- We see that the **carry from the seventh bit *overflows* and is discarded**, giving us the wrong result: 107 + 46 = 25.

```
1
          1   1 1 1
  0       1 1 0 1 0 1 1
  0   +   0 1 0 1 1 1 0
      -----------------
  0       0 0 1 1 0 0 1
```

2.4 Signed Integer Representation

- Signs in signed magnitude representation

  Using signed magnitude binary arithmetic, find the sum of - 46 and - 25.

```
       1 1
1    0101110
1  + 0011001
   ---------
1    1000111
```

- The **signs** of the numbers to be added **are both negative**,
- We add the magnitudes and **use the negative sign for the sum**

2.4 Signed Integer Representation

- Mixed sign addition

  Using signed magnitude binary arithmetic, find the sum of 46 and - 25.

```
0   0101110
1 + 0011001
  ---------
```

- **Determine** number with the **larger magnitude**
- **Subtract smaller** magnitude **from larger** magnitude
- The **sign of the number with the larger magnitude** becomes the sign of the sum

  – Note the “**borrows**” from the second and sixth bits.

2.4 Signed Integer Representation

- Mixed sign addition

  Using signed magnitude binary arithmetic, find the sum of 46 and - 25.

$$\begin{array}{ccccccccc} & & & 0 & 2 & & & 0 & 2 \\ 0 & & 0 & \cancel{1} & 0 & 1 & 1 & \cancel{1} & 0 \\ 1 & + & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ \hline 0 & & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}$$

- **Determine** number with the **larger magnitude**
- **Subtract smaller** magnitude **from larger** magnitude
- The **sign of the number with the larger magnitude** becomes the sign of the sum
  - Note the "**borrows**" from the second and sixth bits.

2.4 Signed Integer Representation

- **Signed magnitude** representation is easy for humans, but it **requires complicated computer hardware**.
- Another disadvantage of signed magnitude is that it allows **two different representations for zero**: positive zero and negative zero.

  00000000   10000000

- For these reasons computers systems employ ***complement systems*** for number representation.

2.4 Signed Integer Representation

- In **one's complement representation**, positive numbers are the same as in sign-magnitude, and **negative numbers** are the bit **complement** of the corresponding **positive number**.

	+	–
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	0100	1011
5	0101	1010
6	0110	1001
7	0111	1000

- **negative numbers** are indicated by a **1 in the high order bit.**

- **difference** of two values is found by **adding the minuend to the complement of the subtrahend**.

2.4 Signed Integer Representation

- With one’s complement addition, the **carry bit** (if there is one) is **“carried around”** and added to the sum.
  - Example: Compute 48 - 19

```
 1  1 1
    00110000
    11101100
    --------
    00011100
          +1
    --------
    00011101
```

We note that 19 in binary is **00010011**,

so -19 in one’s complement is: **11101100.**

2.4 Signed Integer Representation

- One's complement is simpler to implement than signed magnitude.
- But it still has the disadvantage of having **two different representations for zero**:

  00000000 11111111

- Two's complement solves this problem.

2.4 Signed Integer Representation

- To express a value in two's complement representation:
  - If the number is **positive**, just **convert it to binary** and you're done.
  - If the number is **negative**, find the **one's complement** of the number and then **add 1**.
- Example:
  - In 8-bit binary, 3 is: **`00000011`**
  - -3 using one's complement representation is: **`11111100`**
  - Adding 1 gives us -3 in two's complement form: **`11111101`**.

2.4 Signed Integer Representation

- With two's complement **addition**, all we do is **add** our **two binary numbers**. Just **discard any carries from the high order bit.**
  - Example: Using one's complement binary arithmetic, find the sum of 48 and - 19.

$$\begin{array}{cccccccccc}
1 & & 1 & 1 & & & & & & \\
 & & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
 & + & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
\hline
 & & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1
\end{array}$$

We note that 19 in binary is: **00010011**

so -19 using one's complement is: **11101100**

and -19 using two's complement is: **11101101**

2.4 Signed Integer Representation

- **Excess-M representation** is another way to represent signed integers as binary values.
  - Excess-M representation is intuitive because the binary string with **all 0s represents the smallest number**, whereas the binary string **with all 1s represents the largest number**.

- An unsigned binary integer ***M*** (called the *bias*) **represents the value 0**, whereas **all zeroes** in the bit pattern **represents** the integer ***-M***.

2.4 Signed Integer Representation

- For ***n*-bit patterns**, we choose a **bias** of $M = 2^{n-1} - 1$.
  - For example, if we were using **4-bit** representation, the bias should be $2^{4-1} - 1 = 7$.

2.4 Signed Integer Representation

- The **binary value** of a **signed integer** using excess-M representation is **determined by adding *M* to that integer**.
  - Assuming that we are using **excess-7** representation, the **integer 0_{10} is represented as $0 + 7 = 7_{10} = 0111_2$.**
  - The integer **-7 is represented as $-7 + 7 = 0_{10} = 0000_2$** .
  - To find the **decimal value** of the **excess-7 binary number** 1111_2 **subtract 7**: **$1111_2 = 15_{10}$ and $15 - 7 = 8$**;

2.4 Signed Integer Representation

- Let's compare our representations:

Decimal	Binary (for absolute value)	Signed Magnitude	One's Complement
2	00000010	00000010	00000010
−2	00000010	10000010	11111101
100	01100100	01100100	01100100
−100	01100100	11100100	10011011

Decimal	Binary (for absolute value)	Two's Complement	Excess-127
2	00000010	00000010	10000001
−2	00000010	11111110	01111101
100	01100100	01100100	11100011
−100	01100100	10011100	00011011

2.4 Signed Integer Representation

- When we use any finite number of bits to represent a number, the result of our calculations may become too large or too small to be stored in the computer (**overflow**).
- For **unsigned numbers**, an overflow occurred if a **carry out of the leftmost bit** occurs
- For **signed numbers in complement representation**, an overflow occurred if the **carry in and carry out of the sign bit differs**

2.4 Signed Integer Representation

- Signed numbers in complement representation

Expression	Result	Carry?	Overflow?	Correct Result?
0100 + 0010				
0100 + 0110				
1100 + 1110				
1100 + 1010				

2.4 Signed Integer Representation

- Signed numbers in complement representation

Expression	Result	Carry?	Overflow?	Correct Result?
0100 + 0010	0110	No	No	Yes
0100 + 0110	1010	No	Yes	No
1100 + 1110	1010	Yes	No	Yes
1100 + 1010	0110	Yes	Yes	No

2.4 Signed Integer Representation

- In two's complement we can do **binary multiplication and division by 2** very easily using an ***arithmetic shift*** operation
- A ***left arithmetic shift*** **inserts a 0 in for the rightmost bit** and shifts everything else left one bit; in effect, it **multiplies by 2**
- A ***right arithmetic shift*** shifts everything one bit to the right, but copies the sign bit; it **divides by 2**

2.4 Signed Integer Representation

Example:

Multiply the value 11 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 11:

00001011 (+11)

We **shift left** one place, resulting in:

00010110 (+22)

The sign bit has not changed, so the value is valid.

To multiply 11 by 4, we simply perform a left shift twice.

2.4 Signed Integer Representation

Example:

Divide the value 12 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 12:

00001100 (+12)

We **shift right** one place, resulting in:

00000110 (+6)

(Remember, we carry the sign bit as we shift.)

To divide 12 by 4, we right shift twice.

2.4 Signed Integer Representation

- How to implement binary **multiplication by arbitrary number**?
- **Booth's multiplication algorithm** replaces arithmetic operations with bit shifting to the extent possible.

2.4 Signed Integer Representation

Booth’s multiplication algorithm:

- Multiplies two signed binary values in two's complement notation

```
          0011      (multiplicand)
x         0110      (multiplier)
--------------
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

- Multiplies two signed binary values in two's complement notation
- Examines **adjacent pairs** of bits of the multiplier including an **implicit bit 0** below the least significant bit
- **Iterates over these pairs** from least to most significant bit
- If the multiplicand and multiplier are N-bits, then the **product will be 2N-bits**, all bits over 2N are ignored

```
          0011        (multiplicand)
x         0110(0)     (multiplier)
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

For each pair:

- If pair is **10**, **subtract multiplicand** from product and **shift left**
- If pair is **01**, **add multiplicand** to product and **shift left**
- If pair is **00** or **11**, add binary zero and **shift left**

In each step, **fill leftmost bits with 0's for positive numbers and with 1's for negative numbers**

```
        0011     (multiplicand)
x       0110(0)  (multiplier)
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

For each pair:

- If pair is **10**, **subtract multiplicand** from product and **shift left**
- If pair is **01**, **add multiplicand** to product and **shift left**
- If pair is **00** or **11**, add binary zero and **shift left**

In each step, **fill leftmost bits with 0's for positive numbers and with 1's for negative numbers**

```
          0011      (multiplicand)
x         0110(0)   (multiplier)
+ 00000000          (shift)
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

For each pair:

- If pair is **10**, **subtract multiplicand** from product and **shift left**
- If pair is **01**, **add multiplicand** to product and **shift left**
- If pair is **00** or **11**, add binary zero and **shift left**

In each step, **fill leftmost bits with 0's for positive numbers and with 1's for negative numbers**

```
          0011       (multiplicand)
x         0110(0)    (multiplier)
+ 00000000           (shift)
- 0000011            (subtract)
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

For each pair:

- If pair is **10**, **subtract multiplicand** from product and **shift left**
- If pair is **01**, **add multiplicand** to product and **shift left**
- If pair is **00** or **11**, add binary zero and **shift left**

In each step, **fill leftmost bits with 0's for positive numbers and with 1's for negative numbers**

```
         0011       (multiplicand)
x        0110(0)    (multiplier)
+ 00000000          (shift)
- 0000011           (subtract)
+ 000000            (shift)
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

For each pair:

- If pair is **10**, **subtract multiplicand** from product and **shift left**
- If pair is **01**, **add multiplicand** to product and **shift left**
- If pair is **00** or **11**, add binary zero and **shift left**

In each step, **fill leftmost bits with 0's for positive numbers and with 1's for negative numbers**

```
         0011        (multiplicand)
x        0110(0)     (multiplier)
+ 00000000       (shift)
- 0000011        (subtract)
+ 000000         (shift)
+ 00011          (add)
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

For each pair:

- If pair is **10**, **add two's complement of multiplicand** to product and **shift left**
- If pair is **01**, **add multiplicand** to product and **shift left**
- If pair is **00** or **11**, add binary zero and **shift left**

In each step, **fill leftmost bits with 0's for positive numbers and with 1's for negative numbers**

```
          0011        (multiplicand)
x         0110(0)     (multiplier)
+ 00000000            (shift)
+ 1111101             (subtract)
+ 000000              (shift)
+ 00011               (add)
```

2.4 Signed Integer Representation

Booth's multiplication algorithm:

For each pair:

- If pair is **10**, **add two's complement of multiplicand** to product and **shift left**
- If pair is **01**, **add multiplicand** to product and **shift left**
- If pair is **00** or **11**, add binary zero and **shift left**

In each step, **fill leftmost bits with 0's for positive numbers and with 1's for negative numbers**

```
              0011        (multiplicand)
x             0110(0)     (multiplier)
  + 00000000              (shift)
  + 1111101               (subtract)
  + 000000                (shift)
  + 00011                 (add)
    00010010
```

We see that 3 × 6 = 18!