Signed Integers

Hui Chen [a]

[a]CUNY Brooklyn College, Brooklyn, NY, USA

September 6, 2023

Outline

Acknowledgement

The content of most slides come from the authors of the textbook:

Null, Linda, & Lobur, Julia (2018). The essentials of computer organization and architecture (5th ed.). Jones & Bartlett Learning.

Table of Contents

Lesson Objectives

Students are expected to be able to

1. describe the fundamentals of numerical data representation and manipulation in digital computers;
2. convert between various radix systems;
3. *convert and perform arithmetic in signed integer representations*;
4. *explain how errors can occur in computations because of overflow and truncation*;
5. express floating numbers in floating-point representation;
6. recognize the most popular character codes; and
7. describe the concepts of error detecting and correcting codes.

Table of Contents

Signed Integer

Integers can be signed (indicating positive or negative numbers)

- How to we represent signed integers?
- In general, allocate the high-order bit to indicate the sign of a number.
  - The high-order bit is typically on the most significant (often leftmost) bit.
  - Example: 0 is used to indicate a positive number; 1 indicates a negative number.
  - The remaining bits contain the value of the number that can be interpreted in different ways

Signed Integer Representations

Three ways:

- Signed magnitude
- One's complement
- Two's complement
- Excess-M representation (offset binary representation)

Table of Contents

Signed Magnitude Representation

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

```
+3: 0000 0011
-3: 1000 0011
```

Signed Magnitude: Arithmetic Operations

On signed magnitude numbers, they are carried out in much the same way as humans carry out pencil and paper arithmetic.

Humans often ignore the signs of the operands while performing a calculation, applying the appropriate sign after the calculation is complete.

- to discuss binary addition
- how about binary subtraction?

Binary Addition

Consider the four rules (all numbers are binary):

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

How about multiple-digit numbers?

Binary Addition: Example 1

Using signed magnitude binary arithmetic, find the sum of 75_{10} and 46_{10}.

1. convert 75 and 46 to binary, and arrange as a sum, but separate the (positive) sign bits from the magnitude bits; and
2. find the sum starting with the rightmost bit and work left, note and add the carries

Binary Addition: Example 1

Using signed magnitude binary arithmetic, find the sum of 75_{10} and 46_{10}.

```
0   100 1011
0 + 010 1110
-----------
0   111 1001
```

The result is $01111001_2 = 121_{10}$

In this example, we were careful to pick two values whose sum would fit into 7 bits. What if that is not the case?

Binary Addition: Example 2

Using signed magnitude binary arithmetic, find the sum of 107_{10} and 46_{10}.

```
    1 (carry bit)
0     110 1011
0 +   010 1110
------------
0   ? 011 1001
```

We see that the carry from the seventh bit *overflows* and is discarded, giving us the erroneous result: $00111001_2 = 25_{10}$.

Binary Addition: Example 3

Using signed magnitude binary arithmetic, find the sum of -46_{10} and -25_{10} (the two numbers have an identical sign).

Because the signs are the same, all we do is add the numbers and supply the negative sign when we are done.

```
1    010 1110
1 + 001 1001
------------
1    100 0111
```

The result is $11000111_2 = -71_{10}$.

Mixed Sign Addition (or Subtraction)

It is done the similar way, but note:

- The sign of the result gets the sign of the number that is larger.
- Sometimes, we have to "borrow" from the higher order bits.

Mixed Sign Addition (or Subtraction): Example 4

Example: Using signed magnitude binary arithmetic, find the sum of 46_{10} and -25_{10}.

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

Note the "borrows" from the second and sixth bits.

The result is $00010101_2 = 21_{10}$.

Signed Magnitude Representation: Summary

- Signed magnitude representation is to understand, but it requires complex computer hardware (such as keep tracking of carries and borrows)
- It allows two different representations for zero: positive zero and negative zero such as

$$00000000_2 = +0 \tag{1}$$

$$10000000_2 = -0 \tag{2}$$

- To avoid these problems, computers systems actually employ complement systems for numeric value representation.

Table of Contents

Complement Number Systems

Negative values are represented by some difference between a number and its base (i.e., represented by a complement).

For example, the *diminished radix complement* of a non-zero number N in base r with d digits is $(r^d - 1) - N$

In the binary system, this gives us one's complement. It amounts to little more than flipping the bits of a binary number.

One's Complement

One's complement a non-zero number N with d digits is

$$(2^d - 1) - N$$

with which, we represent negative numbers

One's Complement: Example

For example, using 8-bit one's complement representation:

```
+3 = 00000011
-3 = 11111100
```

How?

$$(2^8 - 1) - 3 = 255 - 3 = 252$$

and

$$252_{10} = 11111100_2$$

which represents -3_{10}.

This can be obtained by flipping every bit of 00000011_2!

One's Complement: Advantages

In one's complement representation, as with signed magnitude, negative values are indicated by a 1 in the high order bit.

However, complement systems are useful because they eliminate the need for subtraction.

One's Complement: Arithmetic Operations

Using one's complement binary arithmetic, find the sum of 48_{10} and -19_{10}.

1. convert the numbers to binary signed integer
2. add the two numbers
3. with one's complement addition, the carry bit is "carried around" and added to the sum

One's Complement: Arithmetic Operations

Using one's complement binary arithmetic, find the sum of 48_{10} and -19_{10}.

$48_{10} = 00110000_2$ and $19_{10} = 00010011_2$. Then -19's one's complement is 11101100_2.

```
    0 0 1 1 0 0 0 0
+   1 1 1 0 1 1 0 0
-------------------
    0 0 0 1 1 1 0 0
+               1    (carry bit)
-------------------
    0 0 0 1 1 1 0 1
```

The result is $00011101_2 = 29_{10}$

One's Complement: Summary

- One's complement is simpler to implement than signed magnitude despite that the "end carry around" adds some complexity.
- But it still has the disadvantage of having two different representations for zero: positive zero and negative zero, e.g.,

```
0000 0000 = +0
1111 1111 = -0
```

Two's Complement

Two's complement overcomes the problems of positive and negative 0's.

Considering the radix complement of a numbering system where the radix complement of a non-zero number N in base r with d digits is $r^d - N$.

When the base is 2, we have Two's complement $2^d - N$ for d digits.

Two's Complement Representation

To express a value in two's complement representation:

- If the number is positive, just convert it to binary.
- If the number is negative, find the one's complement of the number (easy way: flipping bits) and then add 1.

Example: representing -3 in Two's complement of 8 bits:

1. in 8-bit binary, $3_{10} = 00000011_2$
2. -3's one's complement representation of 8 bits is: 11111100_2
3. Adding 1 to it, we have 11111101_2.

Two's Complement: Arithmetic

- With two's complement arithmetic, all we do is add our two binary numbers.
- Just discard any carries emitting from the high order bit if any.

Two's Complement: Example

For 8-bit binary numbering system, using Two's complement binary arithmetic, find the sum of 48 and -19.

We note: $48_{10} = 00110000_2$ and $-19_{10} = 11101101_2$ in 8-bit Two's complements.

```
   0011 0000
 + 1110 1101
------------
1  0001 1101
^
+--- discard any carry-out bit
```

The result is $48_{10} - 19_{10} = 00011101_2 = 29_{10}$

Excess-M Representation

Also called offset binary representation is a method to represent signed integers using unsigned binary values.

- 0 and 2M: an unsigned binary integer M (called the bias) represents the value 0, while all zeroes in the bit pattern represents the integer 2M
- The integer is interpreted as positive or negative depending on where it falls in the range.

Excess-M Representation: Representing Signed Integers

If n bits are used for the binary representation, we typically select the bias in such a manner that we split the range equally.

So we choose a bias of $2^{n-1} - 1$.

Example, if we were using 4-bit representation, the bias should be $M = 2^{4-1} - 1 = 7$, which we use to represent 0.

Then any unsigned integer less than M represents a negative number while any unsigned integer greater than M a positive number.

The unsigned binary value for a signed integer using excess-M representation is determined simply by adding M to that integer.

Excess-M Representation: Example 1

For example, let's consider excess-7 representation for 4 bit numbers where $M = 2^(4 - 1) - 1$

- the integer 0_{10} is represented as $0_{10} + 7_{10} = 7_{10} = 0111_2$.
- The integer 3_{10} is represented as $3_{10} + 7_{10} = 10_{10} = 1010_2$.
- The integer -7_{10} is represented as $-7_{10} + 7_{10} = 0_{10} = 0000_2$.

To find the decimal value of the excess-7 binary number 11112 subtract 7: 11112 = 1510 and 15 – 7 = 8; thus 11112, in excess-7 is + 810.

Excess-M Representation: Example 2

For example, let's consider excess-7 representation for 4 bit numbers where $M = 2^(4 - 1) - 1$. Given the values, find the decimal values?

Convert to decimal and subtract M from the representation.

$$1111_2 - 7_{10} = 15_{10} - 7_{10} = 8_{10} \tag{3}$$

Comparison of the 4 Representations

Assuming 8-bit binary numbers

Decimal	Binary of abs(·)	Signed Magnitude	One's Complement	Two's Complement	Excess-127
2	0000 0010	0000 0010	0000 0010	0000 0010	1000 0001
-2	0000 0010	1000 0010	1111 1101	1111 1110	0111 1101
100	0110 0100	0110 0100	0110 0100	0110 0100	1110 0011
-100	0110 0100	1110 0100	1001 1011	1001 1100	0001 1011

Table of Contents

Booth's Algorithm

Idea: To do multiplication, replace arithmetic operations with bit shifting to the extent possible.

- Assume a mythical "0" starting multiplier bit, process one bit on the multiplier at each step
- Two bits patters:
- 10: shift and subtract, i.e., if the current multiplier bit is 1 and the preceding bit was 0, shift and subtract the multiplicand from the product
- 01: shift and add, i.e., if the current multiplier bit is 0 and the preceding bit was 1, shift and add the multiplicand to the product
- 00: shift only, i.e., if we have a 00 pair, we simply shift.
- 11: shift only, i.e., if we have a 11 pair, we simply shift.

Booth's Algorithm: (Small) Example 1

Compute 3×6 using Booth's Algorithm: 4 bits $\times$ 4 bits, and consider 8 bits product. In the following 3: multiplicand, 6: multiplier

```
                  Mythical
        0011
x       0110          0
------------
     + 0000 shift due to 00 in 011[00] including mythical
    - 0011  shift and subtract due to 01[10]0
   + 0000   shift due to 0[11]00
  + 0011    shift and add [01]100
------------
   00010010
```

The result $= 3_{10} \times 6_{10} = 00010010_2 = 18_{10}$

Booth's Algorithm: (Large) Example 2

```
          00110101
x         01111110
+ 0000000000000000  due to 0111111[00] (mythical 0 added)
+ 11111111001011    due to 011111[10]0
+ 00000000000000    due to 01111[11]00
+ 0000000000000     due to 0111[11]100
+ 000000000000      due to 011[11]1100
+ 00000000000       due to 01[11]11100
+ 0000000000        due to 0[11]111100
+ 000110101         due to [01]1111100
------------------
 1000110100010110
```

In the above -00110101 is in two's complement so that we can use "$+$" instead

Multiplication and Division and Shift Operations

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.

Let's look at some examples.

Shift: Example 1

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.

Shift: Example 2

Multiply the value 11 (expressed using 8-bit signed two's complement representation) by 4.

We start with the binary value for 11:

$$00001011(+11)$$

We shift left *two* places, resulting in:

$$00101100(+44)$$

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

Shift: Example 3

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 left one place, resulting in:

$$00000110(+6)$$

Remember, we carry the sign bit to the left as we shift.

Shift: Example 4

Divide the value 12 (expressed using 8-bit signed two's complement representation) by 4. We start with the binary value for 12:

$$00001100(+12)$$

We shift left *two* places, resulting in:

$$00000011(+3)$$

Remember, we carry the sign bit to the left as we shift.

Table of Contents

Overflow Problem

When we use any finite number of bits to represent a number, we always run the risk of the result of our calculations becoming too large or too small to be stored in the computer.

While we can't always prevent overflow, we can always detect overflow.

In complement arithmetic, an overflow condition is easy to detect.

Two's Complement: Overflow Example

Using two's complement binary arithmetic, find the sum of 107_{10} and 46_{10}.

```
0110 1011
+ 0010 1110
-----------
1001 1001
```

10011001_2 in Two's complement is $-1100111_2 = -103_{10}$

We observe there is a carry bit that goes into the sign bit. Does overflow into the sign bit always mean that we have an overflow error?

Two's Complement: Not an Overflow!

Using two's complement binary arithmetic, find the sum of 23_{10} and -9_{10}.

```
0001 0111
+ 1111 0111
-----------
1 0000 1110
^
+---------- discard the carry-out bit
```

00001110_2 in Two's complement is 14_{10}

The result is correct i.e., no overflow error, but we do observe there is a carry bit that goes into the sign bit!

Detecting Overflow

To detect overflow using Two's Complement: Compare the carry-in and the carry-out

- carry-in: The carry that goes into the sign bit
- carry-out: The carry that goes out of the binary

When carry-in $\equiv$ carry-out $\rightarrow$ no overflow

When carry-in $\neq$ carry-out $\rightarrow$ overflow

Unsigned Integer Wrap-Around

Signed and unsigned numbers are both useful.

- For example, memory addresses are always unsigned.

Using the same number of bits, unsigned integers can express twice as many "positive" values as signed numbers.

Wrap-around problem

- Signed integer: overflow
- Unsigned integer: wraps around, e.g., in 4 bits, $1111 + 1 = 0000$

Always stay alert for overflow and wrap-around problem

Overflow and Carry

Overflow and carry can be tricky.

Signed number overflow means nothing in the context of unsigned numbers, which set a carry flag instead of an overflow flag.

If a carry out of the leftmost bit occurs with an unsigned number, overflow has occurred.

Carry and overflow occur independently of each other.

Overflow or Carry?

The table shows the subtle ideas of overflow and carry

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

Table of Contents

Summary and Q&A

You are expected to be able to

1. *convert and perform arithmetic in signed integer representations*;
2. *explain how errors can occur in computations because of overflow and truncation*;

Any questions on:

- Signed Integer
  - Signed Magnitude
  - One's Complement
  - Two's Complement
  - Excess-M Representation
- Multiplication and Division
- Overflow, wrap-around, and truncation