King Saud University

CSC 220: Computer Organization

Unit 4
Signed Number Representation

Prepared by:
Md Saiful Islam, PhD

Department of Computer Science
College of Computer and Information Sciences

Overview

- Unsigned Representation
- Representation of signed numbers
  - Signed Magnitude Representation
  - One's Complement Notation
  - Two's Complement Notation
- Two's Complement Addition
- Comparing Signed Number Systems
- Signed Overflow

Unsigned Representation

- Represents positive integers.

  Ex: 8 bit representation of unsigned numbers:

position	7	6	5	4	3	2	1	0
contribution	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
157	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**
1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
10	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

- Addition is simple:

  0 0 0 0 1 0 0 1 + 0 0 0 0 0 1 0 1 = 0 0 0 0 1 1 1 0.

Binary Addition (1 of 2)

- Two 1-bit values

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	10

"two"

Binary Addition (2 of 2)

- Two *n*-bit values
  - Add individual bits
  - Propagate carries
  - E.g.,

```
    1   1
  00010101      21
+ 00011001    + 25
  00101110      46
```

Unsigned Representation …

Advantages:

One representation of zero

Simple addition

Disadvantages

Negative numbers can not be represented.

The need of different notation to represent negative numbers.

Representation of signed numbers

- Is a representation of negative numbers possible?
  - you can not just stick a negative sign in front of a binary number. (it does not work like that)

- There are three methods used to represent negative numbers.
  - Signed magnitude representation
  - One’s complement representation
  - Two’s complement representation

- We will consider two operations
  - How to get –ve number from +ve number
  - How to add two signed Numbers

Signed magnitude representation

- Humans use the signed-magnitude system. We add + or - to the front of a number to indicate its sign.
- We can do this in binary too, by adding a sign bit in front of our numbers.
  - A 0 sign bit represents a positive number.
  - A 1 sign bit represents a negative number.

1101_2	$= 13_{10}$	(a 4-bit unsigned number)
0 1101	$= +13_{10}$	(a positive number in 5-bit signed magnitude)
1 1101	$= -13_{10}$	(a negative number in 5-bit signed magnitude)
0100_2	$= 4_{10}$	(a 4-bit unsigned number)
0 0100	$= +4_{10}$	(a positive number in 5-bit signed magnitude)
1 0100	$= -4_{10}$	(a negative number in 5-bit signed magnitude)

n-bit Representation

For *n* bit representation we use the $(n\text{-}1)^{th}$ bit for the **sign** and remaining bits for **magnitude**

Example:

- Suppose 10011101 is a signed magnitude representation of a 8 bit number.
- The sign bit is 1, then the number represented is negative
- The magnitude is 0011101 with a value $2^4+2^3+2^2+2^0= 29$
- Then the number represented by 10011101 is –29.

position	7	6	5	4	3	2	1	0
contribution	-			2^4	2^3	2^2		2^0
-29	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**

Signed magnitude representation

Exercise 1:

37_{10} has 0010 0101 in signed magnitude notation. Find the signed magnitude of $–37_{10}$ in 8 bits?

Using the signed magnitude notation find the 8-bit binary representation of the decimal value 24_{10} and -24_{10}.

Find the signed magnitude of –63 using 8-bit binary sequence?

Disadvantage of Signed Magnitude

- Addition and subtractions are difficult:

  Signs and magnitude, both have to carry out the required operation.

- There are two representations of 0

  00000000 = $+ 0_{10}$

  10000000 = $- 0_{10}$

  To test if a number is 0 or not, the CPU will need to see whether it is 00000000 or 10000000.

  0 is always performed in programs.

  Therefore, having two representations of 0 is inconvenient.

Ones' complement representation

- In a different representation, ones' complement, we negate numbers by complementing each bit of the number.
- We keep the sign bits: 0 for positive numbers, and 1 for negative.
- The sign bit is complemented along with the rest of the bits.

$1101_2 = 13_{10}$		(a 4-bit unsigned number)
0 1101	$= +13_{10}$	(a positive number in 5-bit ones' complement)
1 0010	$= -13_{10}$	(a negative number in 5-bit ones' complement)

$0100_2 = 4_{10}$		(a 4-bit unsigned number)
0 0100	$= +4_{10}$	(a positive number in 5-bit ones' complement)
1 1011	$= -4_{10}$	(a negative number in 5-bit ones' complement)

Why is it called ones' complement?

- Complementing a single bit is equivalent to subtracting it from 1.

x	x'	1 - x
0	1	1
1	0	0

- Similarly, complementing each bit of an *n*-bit number is equivalent to subtracting that number from 2^n-1.
- For example, we can negate the 5-bit number 01101.
  - Here n=5, and 2^5-1 = 11111_2.
  - Subtracting 01101 from 11111 yields 10010.

$$
\begin{array}{rccccc}
 & 1 & 1 & 1 & 1 & 1 \\
- & 0 & 1 & 1 & 0 & 1 \\
\hline
 & 1 & 0 & 0 & 1 & 0
\end{array}
$$

Ones' complement addition

- There are two steps in adding ones' complement numbers.
  1. Do unsigned addition on the numbers, *including* the sign bits.
  2. Take the carry out and add it to the sum.

$$
\begin{array}{rrrrrrl}
 & & 0 & 1 & 1 & 1 & (+7) \\
+ & & 1 & 0 & 1 & 1 & +\ (-4) \\
\hline
1 & & 0 & 0 & 1 & 0 &
\end{array}
\qquad
\begin{array}{rrrrrrl}
 & & 0 & 0 & 1 & 1 & (+3) \\
+ & & 0 & 0 & 1 & 0 & +\ (+2) \\
\hline
0 & & 0 & 1 & 0 & 1 &
\end{array}
$$

$$
\begin{array}{rrrrrl}
 & 0 & 0 & 1 & 0 & \\
+ & & & & 1 & \\
\hline
 & 0 & 0 & 1 & 1 & (+3)
\end{array}
\qquad
\begin{array}{rrrrrl}
 & 0 & 1 & 0 & 1 & \\
+ & & & & 0 & \\
\hline
 & 0 & 1 & 0 & 1 & (+5)
\end{array}
$$

- This is simpler than signed magnitude addition, but still a bit tricky.
- **Two representation of zero** (0000 = +0, 1111 = -0)

Two's Complement representation

- The most used representation for integers.

  All positive numbers begin with 0.

  All negative numbers begin with 1.

Two's complement representation

- Our final idea is two's complement. To negate a number, we complement each bit (just as for ones' complement) and then add 1.

1101_2	$= 13_{10}$	(a 4-bit unsigned number)
0 1101	$= +13_{10}$	(a positive number in 5-bit two's complement)
1 0010	$= -13_{10}$	(a negative number in 5-bit *ones'* complement)
1 0011	$= -13_{10}$	(a negative number in 5-bit two's complement)

0100_2	$= 4_{10}$	(a 4-bit unsigned number)
0 0100	$= +4_{10}$	(a positive number in 5-bit two's complement)
1 1011	$= -4_{10}$	(a negative number in 5-bit *ones'* complement)
1 1100	$= -4_{10}$	(a negative number in 5-bit two's complement)

More about two's complement

- Another way to negate an *n*-bit two's complement number is to subtract it from 2^n.

$$\begin{array}{r r l} & 100000 & \\ - & 01101 & (+13_{10}) \\ \hline & 10011 & (-13_{10}) \end{array} \qquad \begin{array}{r r l} & 100000 & \\ - & 00100 & (+4_{10}) \\ \hline & 11100 & (-4_{10}) \end{array}$$

- You can also complement all of the bits to the left of the rightmost 1.

01101 = $+13_{10}$ (a positive number in two's complement)
10011 = -13_{10} (a negative number in two's complement)

00100 = $+4_{10}$ (a positive number in two's complement)
11100 = -4_{10} (a negative number in two's complement)

Two’s Complement representation …

Example:
11110101 in Two’s Complement (8 bit number)

The most significant bit is 1, hence it is a negative number.

Corresponding number is 00001011 = 8 + 2+1 = 11
the result is then –11.

Two's complement addition

- Negating a two's complement number takes a bit of work, but addition is much easier than with the other two systems.
- To find A + B, you just have to do unsigned addition on A and B (including their sign bits), and *ignore* any carry out.
- For example, we can compute 0111 + 1100, or (+7) + (-4).
  - First add 0111 + 1100 as unsigned numbers.

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

  - Ignore the carry out (1). The answer is 0011 (+3).

Another two's complement example

- To further convince you that this works, let's try adding two negative numbers—1101 + 1110, or (-3) + (-2) in decimal.
- Adding the numbers gives 11011.

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

- Dropping the carry out (1) leaves us with the answer, 1011 (-5).

An algebraic explanation

- For n-bit numbers, the negation of B in two's complement is 2^n - B. (This was one of the alternate ways of negating a two's complement number.)

$$\begin{aligned} A - B &= A + (-B) \\ &= A + (2^n - B) \\ &= (A - B) + 2^n \end{aligned}$$

- If $A \geq B$, then (A - B) has to be positive, and the 2^n represents a carry out of 1. Discarding this carry out leaves us with the desired result, (A - B).
- If $A < B$, then (A - B) must be negative, and 2^n - (A - B) corresponds to the correct result -(A - B) in two's complement form.

Advantages of Two’s Complement Notation

- One representation of zero
  - 0 is represented as 0000 using 4-bit binary sequence.
- It is easy to add two numbers.
  - Subtraction can be easily performed.
  - Multiplication is just a repeated addition.
  - Division is just a repeated subtraction
  - Two’s complement is widely used in ***ALU***

Comparing the signed number systems

- Here are all the 4-bit numbers in the different systems.
- Positive numbers are the same in all three representations.
- There are *two* ways to represent 0 in signed magnitude and ones' complement. This makes things more complicated.
- In two's complement, there is one more negative number than positive number. Here, we can represent -8 but not +8.
- However, two's complement is preferred because it has only one 0, and its addition algorithm is the simplest.

Decimal	SM	1C	2C
7	0111	0111	0111
6	0110	0110	0110
5	0101	0101	0101
4	0100	0100	0100
3	0011	0011	0011
2	0010	0010	0010
1	0001	0001	0001
0	0000	0000	0000
-0	1000	1111	—
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	—	—	1000

Ranges of the signed number systems

- How many negative and positive numbers can be represented in each of the different four-bit systems on the previous page?

	Unsigned	SM	1C	2C
Smallest	0000 (0)	1111 (-7)	1000 (-7)	1000 (-8)
Largest	1111 (15)	0111 (+7)	0111 (+7)	0111 (+7)

- The ranges for general *n*-bit numbers (including the sign bit) are below.

	Unsigned	SM	1C	2C
Smallest	0	$-(2^{n-1}-1)$	$-(2^{n-1}-1)$	-2^{n-1}
Largest	2^n-1	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$

Signed overflow

- With 4-bit two's complement numbers, the largest representable decimal value is +7, and the smallest is -8.
- What if you try to compute 4 + 5, or (-4) + (-5)?

	0 1 0 0	(+4)		1 1 0 0	(-4)
+	0 1 0 1	+ (+5)	+	1 0 1 1	+ (-5)
	0 1 0 0 1	(-7)		1 0 1 1 1	(+7)

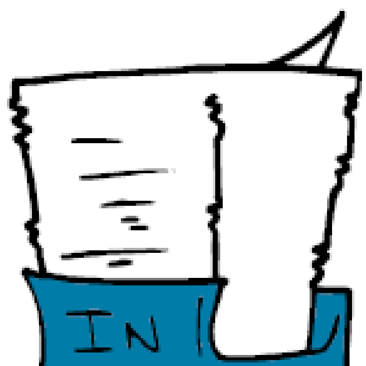
- Signed overflow is very different from unsigned overflow.
  - The carry out is not enough to detect overflow. In the example on the left, the carry out is 0 but there *is* overflow.

Detecting signed overflow

- The easiest way to detect signed overflow is to look at all the sign bits.

$$
\begin{array}{rcccc c r}
 & 0 & 1 & 0 & 0 & & (+4) \\
+ & 0 & 1 & 0 & 1 & & + (+5) \\
\hline
0 & 1 & 0 & 0 & 1 & & (-7)
\end{array}
\qquad\qquad
\begin{array}{rcccc c r}
 & 1 & 1 & 0 & 0 & & (-4) \\
+ & 1 & 0 & 1 & 1 & & + (-5) \\
\hline
1 & 0 & 1 & 1 & 1 & & (+7)
\end{array}
$$

- Overflow occurs only in the two situations above.
  - If you add two *positive* numbers and get a *negative* result.
  - If you add two *negative* numbers and get a *positive* result.
- Overflow can never occur when you add a positive number to a negative number. (Do you see why?)



Overflow

Example1:

$$\begin{array}{rl}
0100000 & \\
0110101_2 & (= 53_{10}) \\
+0101010_2 & (= 42_{10}) \\
\hline
1011111_2 & (= -33_{10})
\end{array}$$

Example2:

$$\begin{array}{rl}
1000000 & \\
1010101_2 & (= -43_{10}) \\
+1001010_2 & (= -54_{10}) \\
\hline
0011111_2 & (= 31_{10})
\end{array}$$

Example3:

$$\begin{array}{rl}
1100000 & \\
0110101_2 & (= 53_{10}) \\
+1101010_2 & (= -22_{10}) \\
\hline
0011111_2 & (= 31_{10})
\end{array}$$

Example4:

$$\begin{array}{rl}
0000000 & \\
0010101_2 & (= 21_{10}) \\
+0101010_2 & (= 42_{10}) \\
\hline
0111111_2 & (= 63_{10})
\end{array}$$

Sign extension

- Decimal numbers are assumed to have an infinite number of 0s in front of them, which helps in "lining up" values for arithmetic operations.

$$\begin{array}{r} 2\ 2\ 5 \\ +\ \ 0\ 0\ 6 \\ \hline 2\ 3\ 1 \end{array}$$

- You need to be careful in extending signed binary numbers, because the leftmost bit is the *sign* and not part of the magnitude.
- To extend a signed binary number, you have to replicate the sign bit. If you just add 0s in front, you might accidentally change a negative number into a positive one!
- For example, consider going from 4-bit to 8-bit numbers.

(+5)	0101	$\longrightarrow$	0000 0101	(+5)
(-4)	1100	$\longrightarrow$	1111 1100	(-4)

Summary

- Data representations are all-important!
  - A good representation for negative numbers can make subtraction hardware much simpler to design.
  - Using two's complement, it's easy to build a single circuit for both addition and subtraction.
- Working with signed numbers involves several issues.
  - Signed overflow is very different from the unsigned overflow we talked about last week.
  - Sign extension is needed to properly "lengthen" negative numbers.