Data Representation II

CMSC 313
Sections 01, 02

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 high-order 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

2.4 Signed Integer Representation

- There are three 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 of the number in the 7 bits to the right of the sign bit.

3

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

4

2.4 Signed Integer Representation

- Binary addition is as easy as it gets. You need to know only four rules:
```
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.
  - We will describe these circuits in Chapter 3.

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

5

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- First, convert 75 and 46 to binary, and arrange as a sum, but separate the (positive) sign bits from the magnitude bits.

```
0   1001011
0 + 0101110
```

6

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Just as in decimal arithmetic, we find the sum starting with the rightmost bit and work left.

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

7

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
```

8

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.

```
         111
0   1001011
0 + 0101110
       1001
```

9

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    1001011
0 +  0101110
0    1111001
```

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.

10

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 erroneous result: 107 + 46 = 25.

```
1
       1   1 1 1
0    1101011
0 +  0101110
0    0011001
```

11

2.4 Signed Integer Representation

- The signs in signed magnitude representation work just like the signs in pencil and paper arithmetic.
  – Example**:** Using signed magnitude binary arithmetic, find the sum of - 46 and - 25.

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

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

12

2.4 Signed Integer Representation

- Mixed sign addition (or subtraction) is done the same way.
  - Example: Using signed magnitude binary arithmetic, find the sum of 46 and - 25.

$$\begin{array}{lllllllll} & & & \scriptstyle 0 \; 2 & & & & \scriptstyle 0 \; 2 & \\ 0 & & 0 & \cancel{1} & 0 & 1 & 1 & \cancel{1} & 0 \\ 1 & + & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ \hline 0 & & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}$$

- The sign of the result gets the sign of the number that is larger.
  - Note the "borrows" from the second and sixth bits.

13

2.4 Signed Integer Representation

- Signed magnitude representation is easy for people to understand, 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.
- For these reasons (among others) computers systems employ *complement systems* for numeric value representation.

14

2.4 Signed Integer Representation

- In complement systems, negative values are represented by some difference between a number and its base.
- 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.

15

2.4 Signed Integer Representation

- For example, using 8-bit one's complement representation:
  - + 3 is: `00000011`
  - - 3 is: `11111100`
- In one's complement representation, as with signed magnitude, negative values are indicated by a 1 in the high order bit.
- Complement systems are useful because they eliminate the need for subtraction. The 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 is "carried around" and added to the sum.
  - Example**:** Using one's complement binary arithmetic, find the sum of 48 and - 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

- Although the "end carry around" adds some complexity, one's complement is simpler to implement than signed magnitude.
- But it still has the disadvantage of having two different representations for zero: positive zero and negative zero.
- Two's complement solves this problem.
- Two's complement is the radix complement of the binary numbering system; the *radix complement* of a non-zero number N in base r with d digits is $r^d - N$.

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

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

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

- Lets 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, 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.

22

2.4 Signed Integer Representation

- Example:
  - Using two's complement binary arithmetic, find the sum of 107 and 46.
- We see that the nonzero carry from the seventh bit *overflows* into the sign bit, giving us the erroneous result: 107 + 46 = -103.

$$\begin{array}{r} 1\,1\;\;1\,1\,1\;\;\; \\ 01101011 \\ +\ 00101110 \\ \hline 10011001 \end{array}$$

But overflow into the sign bit does not always mean that we have an error.

23

2.4 Signed Integer Representation

- Example:
  - Using two's complement binary arithmetic, find the sum of 23 and -9.
  - We see that there is carry into the sign bit and carry out. The final result is correct: 23 + (-9) = 14.

$$\begin{array}{r} 1 \leftarrow 1\,1\,1\;\;1\,1\,1\;\;\; \\ 00010111 \\ +\ 11110111 \\ \hline 00001110 \end{array}$$

Rule for detecting signed two's complement overflow: When the "carry in" and the "carry out" of the sign bit differ, overflow has occurred. If the carry into the sign bit equals the carry out of the sign bit, no overflow has occurred.

24

2.4 Signed Integer Representation

- 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.
- Trouble arises if an unsigned value "wraps around."
  - In four bits: 1111 + 1 = 0000.
- Good programmers stay alert for this kind of problem.

25

2.4 Signed Integer Representation

- Overflow and carry are tricky ideas.
- 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.

The table on the next slide summarizes these ideas.

26

© Odua Images/ShutterStock, Inc. Copyright © 2014 by Jones & Bartlett Learning, LLC an Ascend Learning Company www.jblearning.com

2.4 Signed Integer 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

27

© Odua Images/ShutterStock, Inc. Copyright © 2014 by Jones & Bartlett Learning, LLC an Ascend Learning Company www.jblearning.com

2.4 Signed Integer Representation

- 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.

28

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.

29

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 left one place, resulting in:

  00000110 (+6)

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

To divide 12 by 4, we right shift twice.

30

Character Codes

2.6 Character Codes

- Calculations aren't useful until their results can be displayed in a manner that is meaningful to people.
- We also need to store the results of calculations, and provide a means for data input.
- Thus, human-understandable characters must be converted to computer-understandable bit patterns using some sort of character encoding scheme.

2.6 Character Codes

- As computers have evolved, character codes have evolved.
- Larger computer memories and storage devices permit richer character codes.
- The earliest computer coding systems used six bits.
- Binary-coded decimal (BCD) was one of these early codes. It was used by IBM mainframes in the 1950s and 1960s.

2.6 Character Codes

- In 1964, BCD was extended to an 8-bit code, Extended Binary-Coded Decimal Interchange Code (EBCDIC).
- EBCDIC was one of the first widely-used computer codes that supported upper *and* lowercase alphabetic characters, in addition to special characters, such as punctuation and control characters.
- EBCDIC and BCD are still in use by IBM mainframes today.

34

2.6 Character Codes

- Other computer manufacturers chose the 7-bit ASCII (American Standard Code for Information Interchange) as a replacement for 6-bit codes.
- While BCD and EBCDIC were based upon punched card codes, ASCII was based upon telecommunications (Telex) codes.
- Until recently, ASCII was the dominant character code outside the IBM mainframe world.

35

2.6 Character Codes

- Many of today's systems embrace Unicode, a 16-bit system that can encode the characters of every language in the world.
  - The Java programming language, and some operating systems now use Unicode as their default character code.
- The Unicode codespace is divided into six parts. The first part is for Western alphabet codes, including English, Greek, and Russian.

36

2.6 Character Codes

- The Unicode codes-pace allocation is shown at the right.
- The lowest-numbered Unicode characters comprise the ASCII code.
- The highest provide for user-defined codes.

Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE

37

Floating Point Formats

38

2.5 Floating-Point Representation

- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
  - For example:

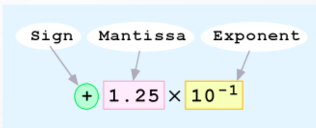
    $0.125 = 1.25 \times 10^{-1}$

    $5{,}000{,}000 = 5.0 \times 10^{6}$

39

2.5 Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



40

2.5 Floating-Point Representation

- Computer representation of a floating-point number consists of three fixed-size fields:

- This is the standard arrangement of these fields.

Note: Although "significand" and "mantissa" do not technically mean the same thing, many people use these terms interchangeably. We use the term "significand" to refer to the fractional part of a floating point number.

41

2.5 Floating-Point Representation

- The one-bit sign field is the sign of the stored value.
- The size of the exponent field determines the range of values that can be represented.
- The size of the significand determines the precision of the representation.

42

IEEE-754 32-bit Floating Point Format

- sign bit, 8-bit exponent, 23-bit mantissa
- normalized as 1.xxxxx
- leading 1 is hidden
- 8-bit exponent in excess 127 format
  - NOT excess 128
  - `0000 0000` and `1111 1111` are reserved
- `+0` and `-0` is zero exponent and zero mantissa
- `1111 1111` exponent and zero mantissa is infinity

UMBC, CMSC313, Richard Chang <chang@umbc.edu>

2.5 Floating-Point Representation

- Example: Express -3.75 as a floating point number using IEEE single precision.
- First, let's normalize according to IEEE rules:
  - 3.75 = $-11.11_2 = -1.111 \times 2^1$
  - The bias is 127, so we add 127 + 1 = 128 (this is our exponent)

1	1	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(implied)

  - Since we have an implied 1 in the significand, this equates to
    $-(1).111_2 \times 2^{(128-127)} = -1.111_2 \times 2^1 = -11.11_2 = -3.75.$

2.5 Floating-Point Representation

- Using the IEEE-754 single precision floating point standard:
  - An exponent of 255 indicates a special value.
    - If the significand is zero, the value is ± infinity.
    - If the significand is nonzero, the value is NaN, "not a number," often used to flag an error condition.
- Using the double precision standard:
  - The "special" exponent value for a double precision number is 2047, instead of the 255 used by the single precision standard.

Memory Organization, “Endian”ness

Random Access Memory (RAM)

- A single byte of memory holds 8 binary digits (bits).
- Each byte of memory has its own address.
- A 32-bit CPU can address 4 gigabytes of memory, but a machine may have much less (e.g., 256MB).
- For now, think of RAM as one big array of bytes.
- The data stored in a byte of memory is not typed.
- The assembly language programmer must remember whether the data stored in a byte is a character, an unsigned number, a signed number, part of a multi-byte number, ...

Common Sizes for Data Types

- A byte is composed of 8 bits. Two nibbles make up a byte.
- Halfwords, words, doublewords, and quadwords are composed of bytes as shown below:

Bit	0
Nibble	0110
Byte	10110000
16-bit word (halfword)	11001001 01000110
32-bit word	10110100 00110101 10011001 01011000
64-bit word (double)	01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101
128-bit word (quad)	01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101 00001011 10100110 11110010 11100110 10100100 01000100 10100101 01010001

5.2 Instruction Formats

- Byte ordering, or *endianness*, is another major architectural consideration.
- If we have a two-byte integer, the integer may be stored so that the least significant byte is followed by the most significant byte or vice versa.
  - In *little endian* machines, the least significant byte is followed by the most significant byte.
  - *Big endian* machines store the most significant byte first (at the lower address).

49

5.2 Instruction Formats

- As an example, suppose we have the hexadecimal number 0x12345678.
- The big endian and small endian arrangements of the bytes are shown below.

Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

50

5.2 Instruction Formats

- Big endian:
  - Is more natural.
  - The sign of the number can be determined by looking at the byte at address offset 0.
  - Strings and integers are stored in the same order.
- Little endian:
  - Makes it easier to place values on non-word boundaries.
  - Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic.

51