CIT237

Appendix G: Binary Numbers and BitWise Operations

December 4, 2019

Reminders / Announcement

- Quiz 8 will be held at the start of class on
  Wednesday, December 11.
- The material covered on Quiz 8 will be:
  - The Lectures of November 25 through December 2.
  - Chapters 19, 20, and 21.
- Project 3: EXTENDED due date is December 9.
- Our last day of class is Monday, December 16.
  - In addition to a lecture, we will be demonstrating Lab solutions during that class.

Appendix Document on Moodle

I have been unable to locate the Appendices for the 9th edition of the textbook on the publisher's website.

- The document I have placed on Moodle is from the 8th edition.

Different Points of View

- We have experienced the computer through the somewhat abstract “C++ Programming Model”.
  - Variables of different types:

    `integer`, `double`, `char`, etc.
  - Various statements:

    `if`, `while`, `class`, etc.
- Have you ever wondered how it all works at a lower level?

“Low-Level” Details of Computers

- At the lowest level, a computer is an electronic device with different kinds of circuits inside it.
- We say that the circuitry is “digital” because it involves distinct states:

“OFF”	vs.	“ON”
“False”	vs.	“True”
“No”	vs.	“Yes”
0	vs.	1

Simple Logic Circuits: NOT

- One input, one output.
- The output is TRUE if the input is FALSE.
- Also called “negation” or an “inverter”.
- We can represent this with a “truth table”:

NOT	0	1	input
	1	0	output

Simple Logic Circuits: AND

- Two inputs, one output
- The output is TRUE if *both* inputs are TRUE

AND	0	1
0	0	0
1	0	1

input

output

input

Simple Logic Circuits: OR

- Two inputs, one output
- The output is TRUE if *either* input is TRUE

OR	0	1
0	0	1
1	1	1

Simple Logic Circuits: Exclusive-OR

- Two inputs, one output
- The output is TRUE if *either* input is TRUE, but not if *both* are TRUE

XOR	0	1
0	0	1
1	1	0

Bitwise Operations

- Bitwise Negation: **`~`**
- Bitwise AND: **`&`**
- Bitwise OR: **`|`**
- Bitwise EXCLUSIVE OR: **`^`**
- Not to be confused with the "Logical" operators "**`&&`**", and "**`||`**"

Used with Integer data types only

- char
- int
- short
- long
- long long
- unsigned char
- unsigned (same as unsigned int)
- unsigned short
- unsigned long
- unsigned long long

Binary Representation of Integers

- A one-byte unsigned integer (**`unsigned char`**), contains 8 bits:

7	6	5	4	3	2	1	0

- Each bit can have a value of 0 or 1, but the meaning (as an integer) depends on position:

128	64	32	16	8	4	2	1

But what about negative numbers?

- We *could* arbitrarily define the leftmost bit as a sign (s), and the remaining bits (v) as the value:

s	v	v	v	v	v	v	v

- The trouble with this is that it now becomes possible to have two distinct values for zero:

0	0	0	0	0	0	0	0	**+ 0**
1	0	0	0	0	0	0	0	**- 0**

Bitwise Negation Operator

- The bitwise negation operator (~) changes all zeros to ones, and ones to zeros:

0	1	1	0	0	1	1	0

becomes

1	0	0	1	1	0	0	1

This value is known as the “one’s complement” of the original number.

Is “One’s Complement” a good way to represent negative numbers?

- If we suggest one’s complement as the representation of negative integers, we have a “negative zero” problem similar to before:

0	0	0	0	0	0	0	0	**+ 0**
1	1	1	1	1	1	1	1	**- 0**

One's Complement and Two's Complement

- Given a binary number: `01100110`
- One's complement
  (invert each bit): `10011001`
  next, add one: `00000001`

sum = "Two's complement": `10011010`

Two’s Complement and Negative Numbers

- The “Two’s complement” of a binary number is obtained by taking the “One’s complement” of that number (inverting each bit) and then adding 1.
- The “Two’s complement” of a number is used to represent the *negative* value of that number:

$$-x = (\sim x) + 1$$

- Is this a *good* way to represent negative numbers?

Repeated Negation Yields Original Number

+102(decimal): 01100110

One's Complement: 10011001

\+ 1

Two's Complement: 10011010 −102 ?

- Let's negate it again:

-102(decimal): 10011010

One's Complement: 01100101

\+ 1

Two's Complement: 01100110 +102

Arithmetic Example: -42 + 37

+42 (32+8+2):		00101010	
One's Complement:		11010101	
	+	1	
Two's Complement:		11010110	-42
-42 (decimal):		11010110	
add 37:	+	00100101	
		11111011	-5 really?
One's Complement:		00000100	
	+	1	
Two's Complement:		00000101	+5

Octal Representation

- Octal (base 8) uses digits 0 through 7
- Each octal digit represents 3 bits:

0	1	1	0	0	1	1	0

000: 0	001: 1	010: 2	011: 3
100: 4	101: 5	110: 6	111: 7

Hexadecimal Representation

- Hexadecimal (called “Hex” for short): base 16
- Each Hex digit represents 4 bits:

0	1	1	0	0	1	1	0

- A Hex digit has a value from 0 through 15 (decimal).

Hexadecimal Digits

Binary	Decimal	HEX
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

Binary	Decimal	HEX
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Hexadecimal Representation

- For example, the binary number:

0	1	1	0	0	1	1	0

is represented by the HEX digits:

6 6

Octal notation in C++

- Octal constants begin with zero.

  For example: `0377`

  is the octal representation of binary

1	1	1	1	1	1	1	1

and has a value of decimal 255.

Hex notation in C++

- Hex constants begin with "0x".

  for example, `0xFF` (or `0xff`)

  is the Hex representation of binary:

1	1	1	1	1	1	1	1

and has a decimal value of 255.

Bitwise AND Operation

- The bitwise AND (&) operation does a logical AND of the corresponding bits of each operand.
- For example: 0x05 & 0x0C

Binary equivalent:	0000	0101
	0000	1100
Logical AND	0000	0100

Bitwise OR Operation

- The bitwise OR (|) operation does a logical OR of the corresponding bits of each operand.
- For example: 0x05 | 0x0C

Binary equivalent:	0000	0101
	0000	1100
Logical OR	0000	1101

Bitwise EXCLUSIVE OR Operation

- The Exclusive OR (^) operation does a logical EXCLUSIVE OR of the corresponding bits of each operand.
- For example: 0x05 ^ 0x0C

Binary equivalent:	0000	0101
	0000	1100
Logical XOR	0000	1001

Bitwise LEFT Shift

- The Left shift (<<) operation does a shift of the bits by the indicated number of bits.
- For example: 0x05 << 2

  Binary equivalent: `0000 0101` (decimal 5)

  Shifted left 2 bits: `0001 0100` (decimal 20)

- Left shift 1 bit: same as multiply by 2.
- Left shift 2 bits: same as multiply by 4.
- Left shift 3 bits: same as multiply by 8.

Bitwise RIGHT Shift

- The Right shift (>>) operation does a shift of the bits by the indicated number of bits.
- For example: 0x50 >> 2

  Binary equivalent: `0101 0000` (decimal 80)

  Shifted right 2 bits: `0001 0100` (decimal 20)

- Right shift 1 bit: same as dividing by 2.
- Right shift 2 bits: same as dividing by 4.
- Right shift 3 bits: same as dividing by 8.

Notes on Shift Operations

- Bit values shifted "off the edge" of the variable are lost.
- On many machines, a right shift of <u>signed</u> integers will replicate the sign bit (left-most bit) rather than fill it in with zeros. (This is often called a "sign extended right shift".)
  - Why do they do this? Because it makes the negative numbers work out nicely. (See next slide.)
- <u>However</u>, behavior of the right shift operation on negative numbers is ***implementation dependent***: check your particular platform documentation if your program depends on this behavior.

Sign Extended Example: -42 >> 1

Right-shift one bit -- same as dividing by 2:

-42 (decimal):	`1101 0110`	
right shift 1:	`1110 1011`	-21 ?

- How can we *verify* this result? Negate it:

(result from above)	`1110 1011`	-21 ?
One's Complement:	`0001 0100`	
+	`1`	
Two's Complement:	`0001 0101`	+21

Relevant Quote

"There are 10 kinds of people in the world: those who understand binary, and those who do not."

(origin unknown)