# Introduction to Software Development – CS 6010
## Lecture 13 – Bitwise Operations

Master of Software Development (MSD) Program

Varun Shankar

Fall 2023

# Miscellaneous

- Midterm Exam is on Friday in class via Gradescope

# Lecture 13 – Bitwise Operations

- Topics
  - Bit Manipulation
  - Boolean Logic
  - Shifting

# Accessing a single Bit

- Byte == bit[ 8 ].  // Would like to think of a byte as an array of bits…
- Bit firstBit = myByte[ 0 ]. // Unfortunately C++ does not provide this
- To manipulate bits, we will use versions of the logical operators
  - || and &&.  // Boolean operators
- Specifically, we'll use "bitwise" operators that work on all bits in a variable at the same time.
  - | and &.  // Note: Only a single | or & when working with bits!
- Executes a logical Or or And on each bit individually – simultaneously!
- If we are going to work with individual bits, it is almost always a good idea to be specific about the size of variable we are using – i.e.: use types such as uint8_t or int64_t – preferably an unsigned type.

# Bitwise Boolean Operator Examples

- 0 1 1 0  1 0 1 1
- 0 1 0 1  1 0 1 0  Or ( | )
- 0 1 1 1  1 0 1 1

All bits at the same time.

- 0 1 1 0  1 0 1 1
- 0 1 0 1  1 0 1 0  And ( & )
- 0 1 0 0  1 0 1 0

- Exclusive Or ( ^ ) One or the other but not both.
- 0 0  0
- 0 1  1
- 1 0  1
- 1 1  0

- 0 1 1 0  1 0 1 1
- 0 1 0 1  1 0 1 0  Xor ( ^ )
- 0 0 1 1  0 0 0 1

- What is the bitwise Unary Operation?
  - Unary means only has one parameter.
  - ! Is the Not operator for Booleans
  - ~ is the Not operator for bits, affects every bit, just like &, |

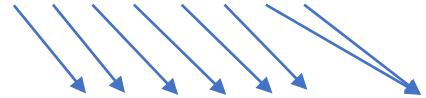
  - int x = 10101111;
  - x = ~x;

# Bitwise Operations on Integers

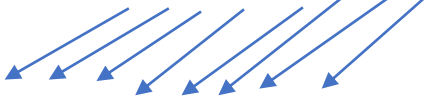
- uint8_t a = 2, b = 4;
- uint8_t c = a | b;
- 010 <– a // You should start to know the 3 (and even 4) bit patterns
- 100 <– b
- 110 <– c == 6
- When we care about what the bits look like, we write them as hex constants:
- int d = 0 x E C; // What is E in bits? C?
- 1 1 1 0 <– E
- 1 1 0 0 <– C
- 1 1 1 0 1 1 0 0 <– d

# Bit Shifting – Move All Bits Left or Right

- Operators << and >>
  - Moves bits to the left or the right (by a given number of bits).
- 0 1 1 0  1 0 1 0 >> 2  // Shift 2 bits to the right – note we fill in with 0s from the left



- 0 0 0 1  1 0 1 0  Lost…// Most significant bit(s) [left most bit(s)] filled in with 0s*
- 0 1 1 0  1 0 1 0 << 3



- 0 1 1 0 0 1 0 1  0 0 0 0.  // Shifted 3 bits to the left – note we also fill in with 0s from the right

- int a = 0001 1010;
- What does shifting to the left mean?  What happens to the number when we shift left?
  - Multiply by 2 (Integer math)
  - But can lose significant bits…
- What does shifting to the right mean?
  - Dividing by 2 (Integer math)

# Bitwise Boolean Identities

- x << 2;
  - Note: This does not change x!
- x = x << 2;
  - Changes x.  Can also be written as:
  - x <<= 2;
- Boolean Identities – x represents a single bit (which could be a 0 or a 1)
  - x | 1 == 1
  - x | 0 == x
  - x & 1 == x
  - x & 0 == 0
- Useful if we want to preserve or change bits. Since all bitwise operations apply to ALL bits in a number simultaneously, we use these identities to be able to keep some bits from changing, and to change others.

# Masking

- int x = ?;  // x is any number
- How could we get just the last 4 bits of x?
- int low4BitsOfX = x & ?
- x & 0 x 0 0 0 F  // Written in hex to make it easier to see…
- x & 0000 0000 0000 1111  // Means this in binary
- Note 0 x 0 0 0 F is usually written as 0 x F

- x = 1111 1111 1010 0101
- 0000 0000 0000 1111
- 0000 0000 0000 0101

# Two’s Complement

- int8_t = -1;
- What are the bits?
  - This is a 2’s Complement number.
  - 1 1 1 1  1 1 1 1

    -128 64 32 16 8 4 2 1
  - What are the values of each bit?
  - But in 2’s Complement, the most significant bit has a place value of…
    - -128
  - If the most significant bit is -128, what is the sum of all the other place values?

# Conversion to a bigger type

- int8_t b = -1; // What does b look like (in bits)?
- 1 1 1 1 1 1 1 1
- int i = b; // what happens to i?
- How many bits are in i?
  - 32
- Would i become:
- 0000 0000 0000 0000 0000 0000 1111 1111 // This is wrong
- What is this value? What was b's value?
- 1111 1111 1111 1111 1111 1111 1111 1111 // This is what actually happens.

# Sign Extension

- When you assign a smaller *signed* integer type into a larger *signed* integer type, the new bits copy the top bit of the smaller value.
- This also applies when you right shift a negative number:
- 1 1 0 0 >> 1  // -4 right shifted == -4 / 2 == -2  [Note 1100 is -8 + 4 => -4]

- 1 1 1 0  // -8 + 4 + 2 => -2
- 0xF000 >> 4 equals what?
  - 0x0F00 // WAIT! There was no sign extension here!  Why?
  - 0xF000 (in a vacuum) is not a negative number.  Hex constants are treated as unsigned
  - But if you assign it to a signed number and then shift that int, the signed number will behave as expected
- short i = 0xF000;  // What is the size of a *short*?
  - 2 bytes

- i >> 4 == 0xFF00
- 0xFF00

# Bitwise Operations Rule of Thumb

- Rule of Thumb:
  - Use *explicitly typed* unsigned types.
    - So we can ignore sign extension
  - uint8_t,  uint16_t,  uint32_t,  uint64_t

# Masking

- Think about what bits you wish to change (or read)
- Design a *mask* that represents those numbers.  A mask usually contains ones in the bit locations of interest, and 0s in all other locations.

# Let’s Do Some Examples

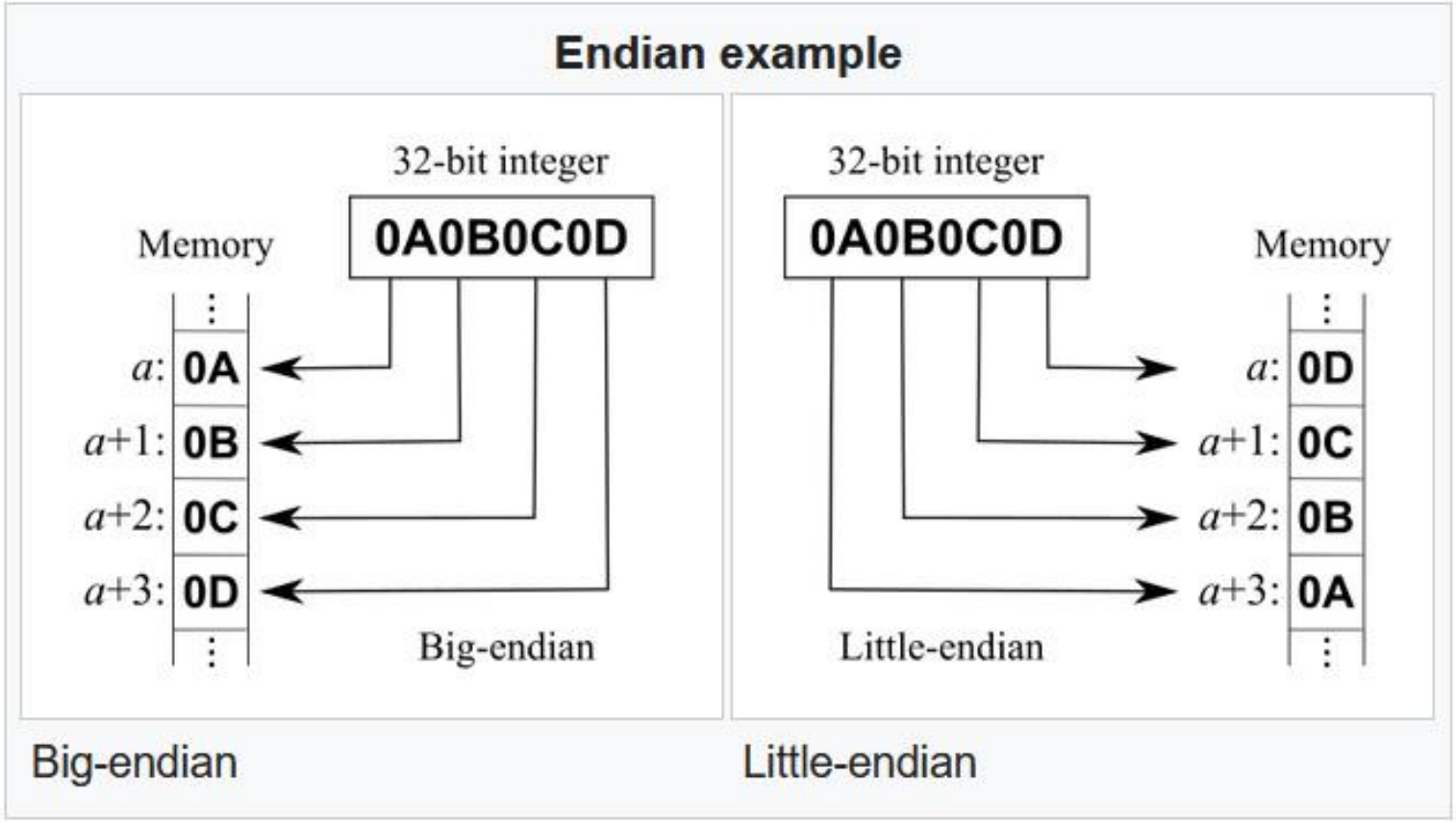
- void printBits( uint32_t num )
  - cout << std::hex << “0x” << num; // To print in Hex
- uint32_t  x  =  0xDEADBEEF;
- uint32_t mask = 0xFF000000;
- How to remove everything but the DE?
  - x = x & mask;
  - x = 0xDE000000
- How to get just the DE?
  - x >> 24
- How to get the BE?
  - y = x & 0x0000FF00 => y = 0x000BE00
  - y = y<<16 =>y = 0xBE000000
  - Y = y>>24 => y = 0x000000BE

  - Could also shift left 16 bits, then shift back right 24 bits.

# Endian-ness

- What date is this:
  - 01/02/03?
  - Is it Jan 2, 2003?  Or is it:
  - Feb 1, 2003?  Or perhaps:
  - Mar 2, 2001?
  - We have to decide what the ordering means.
- Endian-ness refers to the order of bytes as stored in memory.
- This applies to numbers that are stored in multiple bytes.
  - Which “End” comes first when stored in memory…

# Endianness


Endian example
32-bit integer
0A0B0C0D
Memory
a: 0A
a+1: 0B
a+2: 0C
a+3: 0D
Big-endian
32-bit integer
0A0B0C0D
Memory
a: 0D
a+1: 0C
a+2: 0B
a+3: 0A
Little-endian
Big-endian
Little-endian

# Endianness

- We almost never care…
- So when do we care?
  - Loading binary data from a file (that was created on a different architecture)
  - Reading data off of the network (usually that is coming from a machine that is a different architecture than the one we are using)

- Intel x86, ARMv8 – Little Endian
- PowerPC, older ARM – Big Endian

# Today’s Assignment(s)

- Code Review – Book Analyzer
- More catch-up, prep for midterm.
- Bit manipulation assignment tomorrow.