

# Exploring Number Representation

## Endianness

- Big-endian: most significant **byte** is at the first address in memory.
  - 0000 0000 0000 0101 = 5
  - First Byte:  $0*2^{15} + 0*2^{14} + 0*2^{13} + 0*2^{12} + 0*2^{11} + 0*2^{10} + 0*2^9 + 0*2^8$
  - Second Byte:  $0*2^7 + 0*2^6 + 0*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$
- Little-endian: least significant **byte** is at the first address in memory.
  - 0000 0000 0000 0101 = 1280
  - First Byte:  $0*2^7 + 0*2^6 + 0*2^5 + 0*2^4 + 0*2^3 + 0*2^2 + 0*2^1 + 0*2^0$
  - Second Byte:  $0*2^{15} + 0*2^{14} + 0*2^{13} + 0*2^{12} + 0*2^{11} + 1*2^{10} + 0*2^9 + 1*2^8$

## Bit Manipulation

- What do the symbols |, &, ~, ^, <<, >> do in C?
- OR |
  - $3 | 5 = 7$  (binary: 0011 | 0101 = 0111)
  - Solve:  $6 | 2 =$
- AND &
  - $3 \& 5 = 1$  (binary: 0011 & 0101 = 0001)
  - Solve:  $8 \& 9 =$
- XOR ^
  - $3 \wedge 5 = 6$  (binary: 0011 & 0101 = 0110)
  - Solve:  $4 \& 7 =$
- NOT ~
  - $\sim 4 = 11$  (binary:  $\sim 0100 = 1011$ )
  - What would the previous example equal if 4 were stored in an unsigned 2 byte variable?
    - e.g.  $4 = 0000\ 0100$
- SHIFT: move all bits over to the right (>>) or to the left (<<) by a specified number of places. Bits shifted off an end are lost. Bits shifted onto an end are either 0, or the correct bit to preserve sign.
  - *Variable >> Number Of Places Or Variable << Number Of Places*
  - $4 \ll 1 = 8$  (binary: 0100 **SHIFT LEFT 1** = 1000)
  - $3 \ll 2 = 12$  (binary: 0011 **SHIFT LEFT 2** = 1100)
  - Solve:  $5 \gg 1 =$

## Number Representation in C

- What is an *int*? A *float*?
- How many bits do each get?
  - Use the *sizeof* function.
  - `sizeof(int); //result is 4 meaning 4 bytes, or 32 bits`
- Whats the difference between unsigned and signed variables?
  - `unsigned int num1 = 4294967295;`
  - `signed int num2 = 4294967295; //what number is this?`
  - `printf("num1: %u, num2: %d", num1, num2); //print them out`
- How would you use printf to display an integer in base 16?
- How would you display an integer in binary?

- Hint: it's not as easy as displaying in hex!
- C doesn't let you easily see the individual bits of a floating point number as stored in memory.
  - `float num = -4.0; //binary: 110000001 000000000000000000000000`
- Nor does C let you perform bit operations on floats
  - `num << 1; //compile error: "invalid operands to binary <<"`
- How would you manipulate the raw bits of a floating point number?
  - Convert the raw bits into an integer type.
  - Why wouldn't this work?
    - `float num = -4.0;`
    - `int inum = num;`
  - You don't want -4 as an integer, you want the actual bits  
11000000100000000000000000000000 stored in an integer.
    - Either tell C to copy the raw memory from the float into the int:
      - `unsigned int inum = 0;`  
`memcpy(&inum, &num, sizeof(num));`
    - Or tell C to reinterpret the floating point number as an integer.
      - `unsigned int * ptr = (unsigned int*) &num;`  
`unsigned int inum = *ptr;`

## SVN Review

- How do you create a repository?
  - `svnadmin create`
- How do you check out a repository into a working directory?
  - `svn checkout`
- How do you add files to your repository?
  - Without a working directory: `svn import`
  - In a working directory: `svn add`
- How do you put all the changes you have made in your working directory back into the repository?
  - `svn commit`