

# CSE 351

Number Representation & Operators

# Number Bases

- Any numerical value can be represented as a linear combination of powers of  $n$ , where  $n$  is an integer greater than 1
- Example: decimal ( $n=10$ )
  - Decimal numbers are just linear combinations of 1, 10, 100, 1000, etc
  - $1234 = 1*1000 + 2*100 + 3*10 + 4*1$
- We can also use the base  $n=2$  (binary) or  $n=16$  (hexadecimal)

# Binary Numbers

- Each digit is either a 1 or a 0
- Each digit corresponds to a power of 2
- Why use binary?
  - Easy to physically represent two states in memory, registers, across wires, etc
  - High/Low voltage levels
  - This can scale to much larger numbers by using more hardware to store more bits

# Converting Binary Numbers

- To convert the decimal number  $d$  to binary, do the following:
- Compute  $(d \% 2)$ . This will give you the lowest-order bit
- Continue to divide  $d$  by 2, round down to the nearest integer, and compute  $(d \% 2)$  for successive bits
- Example: Convert 25 to binary
  - First bit:  $(25 \% 2) = 1$
  - Second bit:  $(12 \% 2) = 0$
  - Third bit:  $6 \% 2 = 0$
  - Fourth bit:  $3 \% 2 = 1$
  - Fifth bit:  $1 \% 2 = 1$
  - Stop because we reached zero

# Hexadecimal Numbers

- Same concept as decimal and binary, but the base is 16
- Why use hexadecimal?
  - Easy to convert between hex and binary
  - Much more compact than binary

# Converting Hexadecimal Numbers

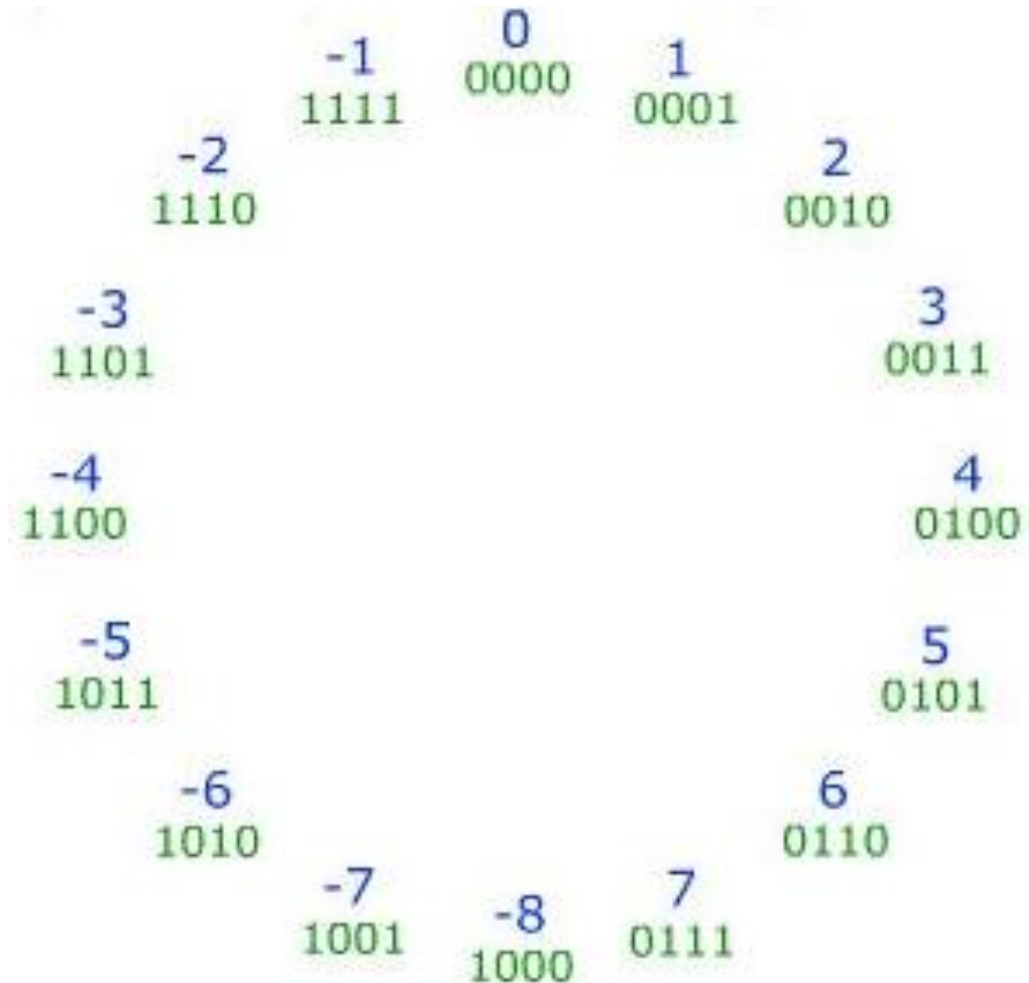
- To convert a decimal number to hexadecimal, use the same technique we used for binary, but divide/mod by 16 instead of 2
- Hexadecimal numbers have a prefix of “0x”
- Example: Convert 1234 to hexadecimal
  - First digit:  $(1234 \% 16) = 2$
  - $1234 / 16 = 77$
  - Second digit:  $(77 \% 16) = 13 = D$
  - $77 / 16 = 4$
  - Third digit:  $4 \% 16 = 4$
  - $4 / 16 = 0$
  - Stop because we reached zero
  - Result: 0x4D2

# Representing Signed Integers

- There are several ways to represent signed integers
- Sign & Magnitude
  - Use 1 bit for the sign, remaining bits for magnitude
  - Works OK, but there are 2 ways to represent zero (-0 and 0)
  - Also, arithmetic is tricky
- Two's Complement
  - Similar to regular binary representation
  - Highest bit has negative weight rather than positive
  - Works well with arithmetic, only one way to represent zero

# Two's Complement

- This is an example of the range of numbers that can be represented by a 4-bit two's complement number
- An  $n$  bit, two's complement number can represent the range  $[-2^{(n-1)}, 2^{(n-1)}-1]$ 
  - Note the asymmetry of this range about 0
- Note what happens when you overflow
- If you still don't understand it, speak up!
  - Very confusing concept





# Bitwise Operators

- NOT:  $\sim$ 
  - This will flip all bits in the operand
- AND:  $\&$ 
  - This will perform a bitwise AND on every pair of bits
- OR:  $|$ 
  - This will perform a bitwise OR on every pair of bits
- XOR:  $\wedge$ 
  - This will perform a bitwise XOR on every pair of bits
- SHIFT:  $\ll, \gg$ 
  - This will shift the bits right or left

# Logical Operators

- With logical operations, a 0 is considered false, anything other than zero is considered true. 1 is produced as output for true.
- NOT: !
  - Evaluates the entire operand, rather than each bit
  - Produces a 1 if the operand is 0, produces 0 otherwise
- AND: &&
  - Produces 1 if both operands are nonzero
- OR: ||
  - Produces 1 if either operand is nonzero

# Common Operator Uses

- A double bang (!! ) is useful when normalizing values to 0 or 1
  - Imitates Boolean types
- Shifts are useful for multiplying/dividing quickly
  - Most multiplications are reduced to shifts when possible by GCC already
  - When writing assembly routines, shifts will be more useful
  - Shifts are also consistent for negative numbers (thanks to sign extension)
- DeMorgan's Laws:
  - $\sim(A|B) == (\sim A \& \sim B)$
  - $\sim(A\&B) == (\sim A | \sim B)$

# Masks

- These are usually strings of 1s that are used to isolate a subset of bits in an operand
  - Example: the mask `0xFF` will “mask” the first byte of an integer
- Once you have created a mask, you can shift it left or right
  - Example: the mask `0xFF << 8` will “mask” the second byte of an integer
- You can apply a mask in different ways
  - To set bits in  $x$ , you can do  $x = x | \text{MASK}$
  - To invert bits in  $x$ , you can do  $x = x ^ \text{MASK}$
  - To erase everything but the desired bits in  $x$ , do  $x = x \& \text{MASK}$

# Application: Symmetric Encryption

- This is an example that shows how XOR can be used to encrypt data
- Say Alice wishes to communicate message  $M$  to Bob
  - Let  $M$  be the bit string: 0b11011010
- Both Alice and Bob have a secret cipher key  $C$ 
  - Let  $C$  be the bit string: 0b01100010
- Alice sends Bob the encrypted message  $M' = M \wedge C$ 
  - $M' =$  0b10111000
- Bob applies  $C$  to  $M'$  to retrieve  $M$ 
  - $M' \wedge C =$  0b11011010
- XOR ciphers are not very secure by themselves, but the XOR operation is used in some modes of AES encryption

# Application: Gray Codes

- Gray Codes encode numbers such that consecutive numbers only differ in their representations by 1 bit
  - Useful when trying to transfer counter values across different clock domains (common in FIFOs)
  - If each wire represents one binary digit, we want to ensure that when the counter increments, the voltage level changes only on one wire
- Let  $n$  be our counter output
  - $(n \gg 1) \wedge n$  will produce a gray coded version of  $n$
- If we receive the gray code  $g$ , we need to convert it to  $n$ :

```
for (int mask = g >> 1; mask != 0; mask >> 1) {  
    g = g ^ mask;  
}
```

- For an example, compile and run [gray\\_code.c](#)

# Lab 1

- **Start early!**
- Intermediate deadline is this Monday
  - Need to finish 4 functions within their respective operator limits
- Tips
  - Work on 8-bit versions first, then scale your solution to work for 32-bit inputs
  - Save intermediate results in variables for clarity
  - **SHIFTING BY MORE THAN 31 BITS IS UNDEFINED!** It will not yield 0

# Example Problems

- Subtract  $b$  from  $a$  without using “-”
  - $(a-b) = a + (-b)$
  - $\sim b + 1 = -b$
  - $(a-b) = a + (\sim b + 1)$
- Create  $0xFFFFFFFF$  using only one operator
  - Limited to constants from  $0x00$  ->  $0xFF$
  - Naïve approach:  $0xFF + (0xFF \ll 8) + (0xFF \ll 16) \dots$
  - Smart approach:  $\sim 0x00 = 0xFFFFFFFF$



# Example Problems

- Replace the leftmost byte of a 32-bit integer with 0xAB
  - Let our integer be  $x$
  - First, we want to create a mask for the lower 24 bits of the image
    - $\sim(0xFF \ll 24)$  will do that using just two operations
  - $(x \& \text{mask})$  will zero out the leftmost 8 bits
  - Now, we want to OR in 0xAB to those zeroed-out bits
  - $(x \& \text{mask}) \mid (0xAB \ll 24)$  will accomplish this
  - Total operators: 5