Number Representation & Operators

CSE 351

Section 2

Number Bases

- Any numerical value can be represented as a linear combination of powers of base 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.
  - E.g.: $1234 = 1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \times 1$

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

Decimal to Binary Conversion

To convert the decimal number `d` to binary, do the following:

1. Compute `(d % 2)`. This will give you the lowest-order bit.
2. Divide `d` by 2, round down to the nearest integer, and continue the process to get the higher order bits.

Example: Convert 25_{10} to binary.

First bit:	`25 % 2 = 1`	`(25 / 2) = 12`
Second bit:	`12 % 2 = 0`	`(12 / 2) = 6`
Third bit:	`6 % 2 = 0`	`(6 / 2) = 3`
Fourth bit:	`3 % 2 = 1`	`(3 / 2) = 1`
Fifth bit:	`1 % 2 = 1`	`(1 / 2) = 0`

Since we hit 0, we're done! $25_{10} = 11001_2$.

Exercise

- Convert 1234_{10} to hexadecimal

Binary to Hexadecimal Conversion

Example: Convert 0xA5E2 to binary.

We can convert this number digit by digit:

A	5	E	2
1010	0101	1110	0010

Converting back to hex is the exact same process; break the bit vector into groups of 4 and convert to hex.

0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Exercise

- Convert 4D2 to Binary

Bitwise Operators

- NOT: ~
  - 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: ^
  - This will perform a bitwise XOR on every pair of bits
- SHIFT: <<, >>
  - This will shift the bits right or left
    - logical vs. arithmetic

AND

&	0	1
0	**0**	**0**
1	**0**	**1**

OR

<table>
<tr><th>|</th><th>0</th><th>1</th></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

XOR

^	0	1
0	**0**	**1**
1	**1**	**0**

NOT

~	
0	**1**
1	**0**

Logical Operators

- NOT: `!`
  - Evaluates the entire operand, rather than each bit
  - Produces a 1 if `== 0`, produces 0 if nonzero
- AND: `&&`
  - Produces 1 if both operands are nonzero
- OR: `||`
  - Produces 1 if either operand is nonzero

Exercise

- 4 & 5
- 14 | 25
- 20 ^ 15

Masks

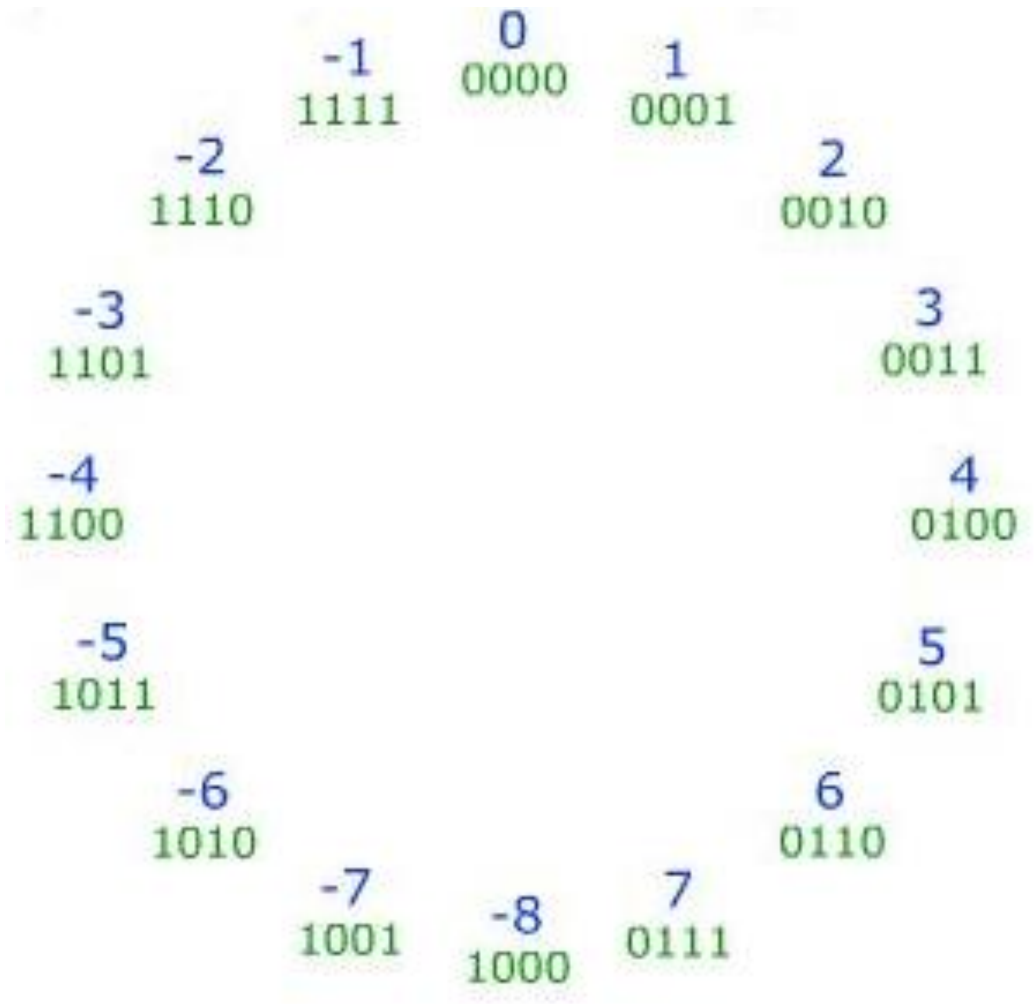
- These are usually strings of 1s that are used to isolate a subset of bits in an operand
  - Example: the mask `0xFF = ...0011111111` will "mask" the first byte of an integer
- Once you have created a mask, you can shift it left or right
  - Example: the mask `0xFF << 8` will "mask" the second byte of an integer
- You can apply a mask in different ways
  - To set bits in `x`, you can do `x = x | MASK`
  - To invert bits in `x`, you can do `x = x ^ MASK`
  - To erase everything but the masked bits in `x`, do `x = x & MASK`

Representing Signed Integers

- Two common ways:
  - <u>Sign & Magnitude</u>
    - Use 1 bit for the sign, remaining bits for magnitude
    - Works OK, but:
      - There are 2 ways to represent zero (`-0` and `0`)
      - Arithmetic is tricky ($4 - 3 \neq 4 + (-3)$)
  - <u>Two's Complement</u>
    - For positives, similar to regular binary representation
    - But, highest bit has a negative weight
    - Solves Sign-and-Magnitude's problems!

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 – there's one more negative number than positive
- Note what happens when you overflow
- If you still don't understand it, speak up!
  - Very confusing concept



Understanding Two's Complement

- There's a simpler way to find the value of a two's complement number, using the handy formula:

`~x + 1 = -x.`

- We can rewrite this as `x = ~(-x - 1)`, i.e. subtract 1 from the given number, and flip the bits to get the positive portion of the number.

Example: `0b11010110`

- Subtract 1: `0b110101`**`10`** `- 1 = 0b110101`**`01`**
- Flip the bits: `0b00101010` $= (32+8+2)_{10} = 42_{10}$
- So the original number we had was -42_{10}.