Number Systems (in Binary)

Today's lecture

- **Representing things with bits**
  - N bits gets you 2^N representations
- **Unsigned binary number representation**
  - Converting between binary and decimal
  - Hexadecimal notation
- **Binary Addition & Bitwise Logical Operations**
  - Every operation has a width
- **Two's complement signed binary representation**

Representing things as bits

- **We said everything in computers is bits (e.g., 1's and 0's)**
- **We do this by associating a pattern of bits with a thing**

(like a secret decoder ring…)

Bit pattern	Marine Mammal
0100101	Humpback Whale
0100110	Leopard Seal
0100111	Sea Otter
0101000	West Indian Manatee
0101001	Bottlenose Dolphin

- **This mapping however is rarely stored explicitly**
  - Rather it is used when we interpret the bits.

How many bits to encode N possible things?

- 1 bit can encode 2 possibilities (0, 1)

0

1

Bits = 1	2	...	n
# encodings = 2	4		2^n

Representing Unsigned numbers

- **Unsigned numbers are the set of non-negative numbers:**
  - 0, 1, 2, 3, 4, 5, …

- **We can only represent a range of these numbers**
  - Based on the # of bits we're using to store the range
    - 3 bits ➔ 0 – 7 (*8 representations*)
    - 8 bits ➔ 0 – 255 (*256 representations*)

- **But what encoding should we use?**

Decimal review

Consider 162.375

- **Numbers consist of a bunch of digits, each with a weight:**

1	6	2	.	3	7	5	Digits
100	10	1		1/10	1/100	1/1000	Weights

- **The weights are all powers of the base, which is 10. We can rewrite the weights like this:**

1	6	2	.	3	7	5	Digits
10^{2}	10^{1}	10^{0}		10^{-1}	10^{-2}	10^{-3}	Weights

- **To find the decimal value of a number, multiply each digit by its weight and sum the products.**

$$(1 \times 10^{2}) + (6 \times 10^{1}) + (2 \times 10^{0}) + (3 \times 10^{-1}) + (7 \times 10^{-2}) + (5 \times 10^{-3}) = 162.375$$

Unsigned Binary Number Representation

- **We use the same scheme to represent binary numbers, except the weights are powers of 2.**
- **For example, here is 1101 in binary:**

1	1	0	1	Binary digits, or bits
2^3	2^2	2^1	2^0	Weights (in base 10)

- **The decimal value is:**

$$(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) =$$
$$8 + 4 + 0 + 1 = 13$$

Powers of 2:

$2^0 = 1$	$2^4 = 16$	$2^8 = 256$
$2^1 = 2$	$2^5 = 32$	$2^9 = 512$
$2^2 = 4$	$2^6 = 64$	$2^{10} = 1024$
$2^3 = 8$	$2^7 = 128$	

Decimal Binary

- **The same works with decimal binary, but this is uncommon outside of floating point representations.**
- **For example, here is 1101.01 in binary:**

1	1	0	1	.	0	1	Binary digits, or bits
2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	Weights (in base 10)

- **The decimal value is:**

$$(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) =$$

$$8 + 4 + 0 + 1 + 0 + 0.25 = 13.25$$

Converting decimal to binary

- Decimal integer → binary: Keep dividing by 2 until the quotient is 0. Collect the remainders in *reverse* order.
- Example: 162:

162 / 2 =	rem	0			
/ 2 =	rem	1			
/ 2 =	rem	0			
/ 2 =	rem	0		>>	10100010
/ 2 =	rem	0			
/ 2 =	rem	1			
/ 2 =	rem	0			
/ 2 =	rem	1			

Converting decimal to binary

- **Decimal integer → binary: Keep dividing by 2 until the quotient is 0. Collect the remainders in *reverse* order.**
- **Example: 162.375:**

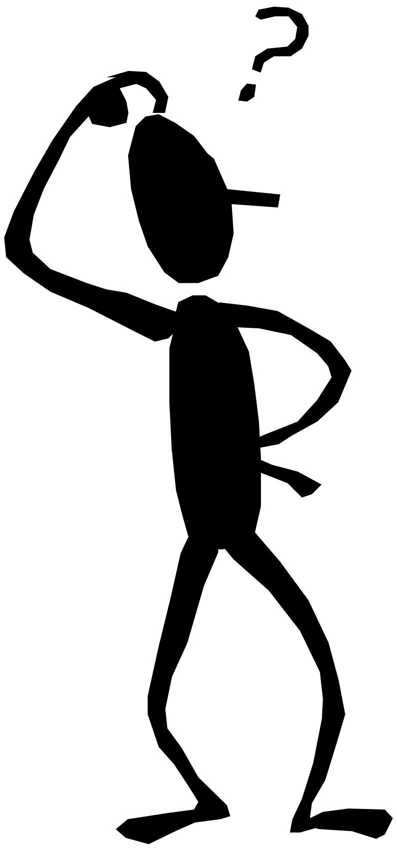
162 / 2 = 81 rem 0
81 / 2 = 40 rem 1
40 / 2= 20 rem 0
20 / 2= 10 rem 0
10 / 2 = 5 rem 0
5 / 2 = 2 rem 1
2 / 2 = 1 rem 0
1 / 2 = 0 rem 1

To convert a fraction, keep multiplying the fractional part by 2 until it becomes 0. Collect the integer parts in forward order.

0.375 x 2 = 0.750
0.750 x 2 = 1.500
0.500 x 2 = 1.000

- **So, 162.375_{10} = 10100010.011_2**

Why does this work?



- This works for converting from decimal to *any* base
- Why? Think about converting 162.375 from decimal to decimal.

  162 / 10= 16 rem 2
  16 / 10= 1 rem 6
  1 / 10= 0 rem 1

- Each division strips off the rightmost digit (the remainder). The quotient represents the remaining digits in the number.
- Similarly, to convert fractions, each multiplication strips off the leftmost digit (the integer part). The fraction represents the remaining digits.

  0.375 x 10 = 3.750
  0.750 x 10 = 7.500
  0.500 x 10 = 5.000

Writing Binary Numbers

- It gets tedious to write 32-bit numbers like:

  10011010111001101011000111111101

- It is even more error prone to copy them.

Hexadecimal (base-16)

- The hexadecimal system uses 16 digits:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

- We can write our 32bit number:

  `1001 1010 1110 0110 1011 0001 1111 1101`

  as

  `0x9AE6B1FD` (C/Java style)

  `32'h9AE6B1FD` (Verilog style)

- Hex is frequently used to specify things like 32-bit IP addresses and 24-bit colors.

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

Binary and hexadecimal conversions

- Converting from hexadecimal to binary is easy: just replace each hex digit with its equivalent 4-bit binary sequence.

$$261.35_{16} = 2 \quad 6 \quad 1 \;.\; 3 \quad 5_{16}$$
$$= 0010 \quad 0110 \quad 0001 \;.\; 0011 \quad 0101_2$$

- To convert from binary to hex, make groups of 4 bits, starting from the binary point. Add 0s to the ends of the number if needed. Then, just convert each bit group to its corresponding hex digit.

$$10110100.001011_2 = 1011 \quad 0100 \;.\; 0010 \quad 1100_2$$
$$= B \quad 4 \;.\; 2 \quad C_{16}$$

Hex	Binary
0	0000
1	0001
2	0010
3	0011

Hex	Binary
4	0100
5	0101
6	0110
7	0111

Hex	Binary
8	1000
9	1001
A	1010
B	1011

Hex	Binary
C	1100
D	1101
E	1110
F	1111

Binary addition by hand (5-bit numbers)

- You can add two binary numbers one column at a time starting from the right, just as you add two decimal numbers.
- But remember that it's binary. For example, 1 + 1 = 10 and you have to carry!

	0	1	0	1	1	Augend	(11)
+	0	1	1	1	0	Addend	(14)
	1	1	0	0	1	Sum	

Most Significant Bit (MSB)

Least Significant Bit (LSB)

Binary addition by hand

- You can add two binary numbers one column at a time starting from the right, just as you add two decimal numbers.
- But remember that it's binary. For example, 1 + 1 = 10 and you have to carry!

The initial carry in is implicitly 0

	1	1	1	0		Carry in
	0	1	0	1	1	Augend
+	0	1	1	1	0	Addend
	1	1	0	0	1	Sum

most significant bit, or MSB

least significant bit, or LSB

Limited representation (4-bit numbers)

- **What if we do that same addition, using only 4-bit numbers**
  - (and where the result can only be 4 bits long…)

	1	0	1	1	Augend	(11)
+	1	1	1	0	Addend	(14)
1	1	0	0	1	Sum	

Overflow

Bitwise Logical operations

- **Most computers also support logical operations like AND, OR and NOT, but extended to multi-bit words instead of just single bits.**
- **To apply a logical operation to two words X and Y, apply the operation on each pair of bits X_i and Y_i:**

	AND		OR		XOR
	1 0 1 1		1 0 1 1		1 0 1 1
AND	1 1 1 0	OR	1 1 1 0	XOR	1 1 1 0
	1 0 1 0		1 1 1 1		0 1 0 1

Bitwise operations in programming

- Languages like C, C++ and Java provide bitwise logical operations:

  & (AND) | (OR) ^ (XOR) ~ (NOT)

- These operations treat each integer as a bunch of individual bits:

  13 & 25 = 9 because 01101 & 11001 = 01001

- Bitwise operators are often used in programs to set a bunch of Boolean options, or flags, with one argument.

- They are *not* the same as the operators &&, || and !, which treat each integer as a single logical value (0 is false, everything else is true):

  13 && 25 = 1 because true && true = true

Bitwise operations in networking

- IP addresses are actually 32-bit (or 128-bit) binary numbers, and bitwise operations can be used to find network information.
- For example, you can bitwise-AND an address 192.168.10.43 with a "subnet mask" to find the "network address," or which network the machine is connected to.

```
  192.168. 10. 43 = 11000000.10101000.00001010.00101011
& 255.255.255.224 = 11111111.11111111.11111111.11100000
-------------------------------------------------------
  192.168. 10. 32 = 11000000.10101000.00001010.00100000
```

- You can use bitwise-OR to generate a "broadcast address," for sending data to all machines on the local network.

```
  192.168. 10. 43 = 11000000.10101000.00001010.00101011
|   0.  0.  0. 31 = 00000000.00000000.00000000.00011111
-------------------------------------------------------
  192.168. 10. 63 = 11000000.10101000.00001010.00111111
```

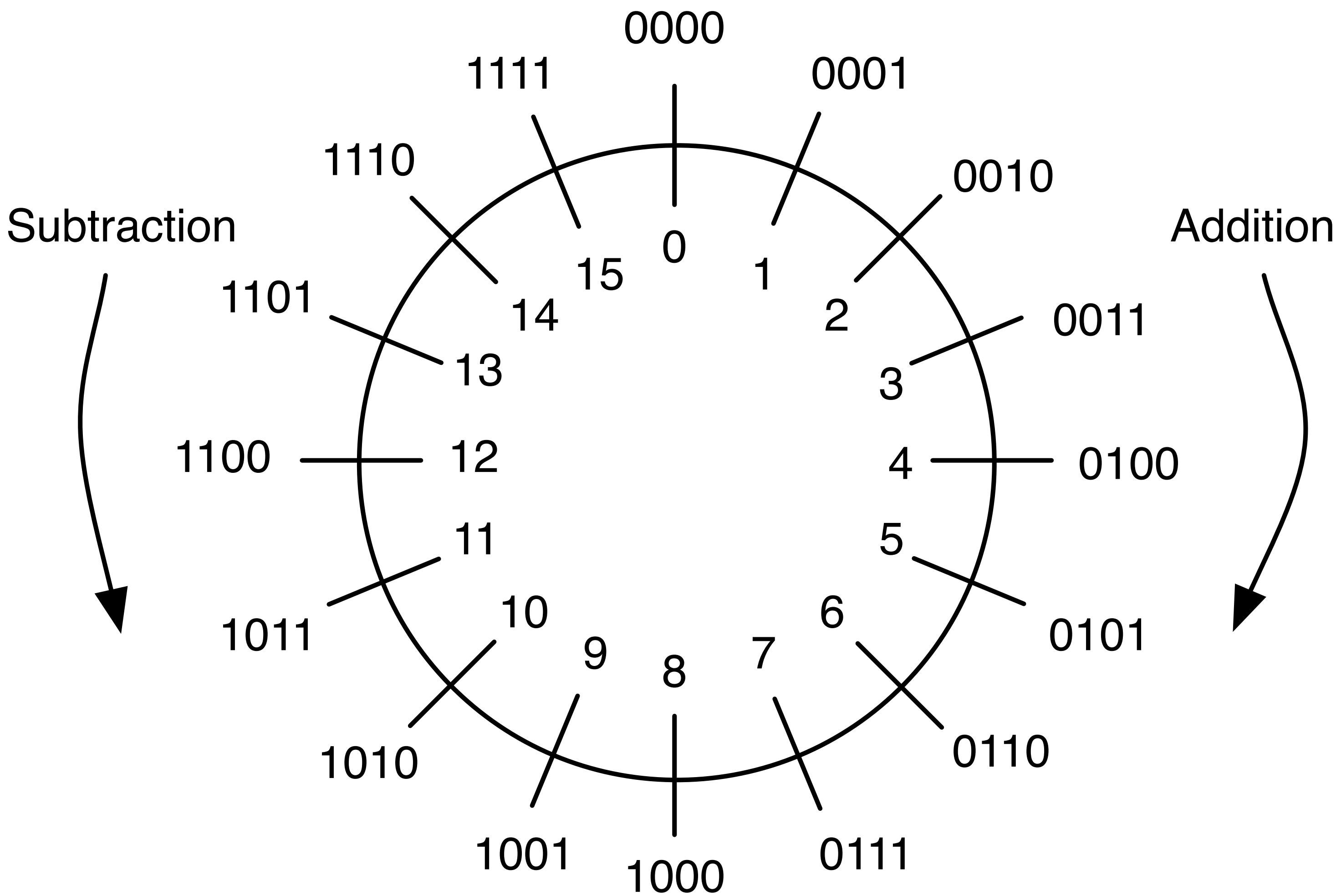
Negative Numbers

- **It is useful to be able to represent negative numbers.**

- **What would be ideal is:**
  - If we could use the **same algorithm to add signed numbers as we use for unsigned numbers**
  - Then our computers wouldn't need 2 kinds of adders, just 1.

- **This is achieved using the 2's complement representation.**

The number wheel (4-bit unsigned #'s)



The number wheel (4-bit 2's complement)

Use MSB to determine the sign of the number it represents (The top bit looks like a sign bit)