M1: Data Representations

1 - Binary Representation

word

- machine-specific grouping of bytes
- For us, word = 4 bytes (as we're using a 32-bit architecture)

1.1 - Binary, Unsigned Integers

1.1.1 - Arithmetic

Works like number addition

If there's overflow, ignore the overflow bits

Ex: Add the 8-bit unsigned binary numbers 241 and 16. What is the result?
241 = 1111 0001 16 = 0001 0000

$$
\begin{array}{r}
1111\ 0001 \\
+\ 0001\ 0000 \\
\hline
1\ 0000\ 0001
\end{array}
$$

overflow
throw away

Result: $0000\ 0001_2 = 1_{10}$

1.1.2 Decimal to Binary

Break into factors of 2

- take the largest power of 2 less than the number, subtract and repeat
- **ie:** 38 ⇒ 32 + 4 + 2 ⇒ 100110

2's Complement

If the leftmost bit is a 0: treat it as unsigned binary

If the rightmost bit is a 1: treat it as unsigned binary and subtract 2^b

2's complement of -x = unsigned representation of 2^b-x

Negate

1. flip all the bits
2. add 1

Decimal ⇒2's Complement

1. Write the positive version binary
2. Negate

2's Complement ⇒ Decimal

1. Negate (flip & add 1)
2. convert to decimal
3. Add negative sign

1.2.2 - Arithmetic

addition and subtraction are the same

- throw away overflow

Multiplication and division need to be handled differently:

- **overflow** occurs when adding numbers if the original two numbers have the same sign, but the result has a different sign

$$\begin{array}{rl} {\scriptstyle 1111\ 1} & \\ 0000\,0100 & (+4) \\ +\,1111\,1101 & (-3) \\ \hline 0000\,0001 & \end{array} \qquad \begin{array}{rl} {\scriptstyle 1111\ 1} & \\ 1111\,1100 & (-4) \\ +\,1111\,1101 & (-3) \\ \hline 1111\,1001 & \end{array}$$

2's complement	Binary
$[-2^{n-1}, 2^{n-1}-1]$	Signed: $[-2^{n-1}, 2^{n-1}-1]$ Unsigned: $[0, 2^{n}-1]$

2 - Hexadecimal NotatioN

$0_{16} = 0_{10}$

$F_{16} = 15_{10}$

Decimal to Hexadecimal

1. Sums of powers of 16
2. Divide by 16 Repeatedly

Number	Quotient	Remainder
3914	244	10
244	15	4
15	0	15

HexaDecimal ⇔ Binary

Each hexadecimal digit is a nibble (4 bits)

- $0_{16} = 0000_2$
- $15_{16} = 1111_2$

What is the hexadecimal -5, represented as a word in two's complement notation?
word = 32 bits $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_2$ $0xFFFFFFFB_{16}$

3 - ASCII Representation

ASCII (American Standard Code for Information Interchange)

- As ASCII is an American standard based on the English alphabet:
  - uses 7 bits [0-127] for 128 unique values
    - 95 of the 128 values are printable
    - The rest are non-printing control codes
  - We use bytes (8 bits) to store ASCII characters, thus wasting one bit per byte

when you press 0 on your keyboard, while it appears on your screen as the symbol we recognize as 0, it is only because the display chose to interpret the value 48 as the symbol 0.

The computer actually stores your keystroke as 48 (in binary of course).

Whether this displays as the digit 0 or the decimal value 48 (or the binary value 00110000), is dependent on how the information is interpreted.

4 - Bitwise Operations

Not (~)

OR (|)

AND (&)

Left-Bit Shift (<<)

- $x << n \Rightarrow x \times 2^n$
- shifting the bits in the given value left by padding additional 0 bits on the right
- any bits that go beyond the fixed size representation are simply discarded

x	n	x << n
0001	0	0001
0001	1	0010
0001	2	0100
0001	3	1000

Right Bit Shift (>>)

- $x >> n \Rightarrow x / 2^n$
- If MSB = 0: pad 0s on the left.
- If MSB = 1: pad 1s on the left

x	n	x >> n
0100	0	0100
0100	1	0010
0100	2	0001
0100	3	0000
1000	0	1000
1000	1	1100
1000	2	1110
1000	3	1111

Ex: Suppose you have a binary sequence that represents an array of boolean values. Each bit is either 1 (“true”) or 0 (“false”). The least significant bit is index 0 of the array, the bit to the left is index 1, and so on
1. Write an expression using bitwise operations that sets index i of the array A[i] = 1/true
A = A | (1 << i)
2. Write an expression using bitwise operations that sets A[i] = b

A = (A & ~(1 << i)) | (b << i)

- first part sets A[i] = 0
- 2nd part sets A[i] = b

Ex

What is printed to the screen?

```c
#include <stdio.h>
int main(void) {
  unsigned char c = 1;
  unsigned char d = 25;
  c <<= 3;
  c |= 1;
  c = c & d;
  printf("%d",c);
  return 0;
 }
```

char = 8 bits

$c = 1_{10} = 00000001_2$

$d = 1_{10} = 00011001_2$

c <<= 3 : 00001000_2

c |= 1 : 00001001_2

c = c : 00001001_2

printf(%d, c) ⇒ 9_{10}