MONTANA
STATE UNIVERSITY

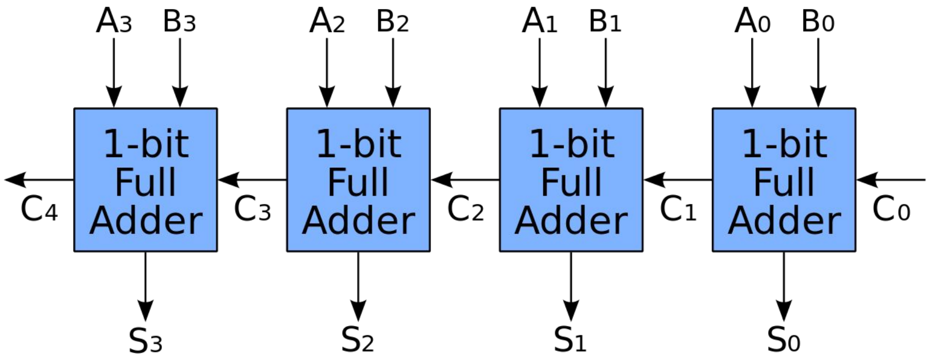
# Bytes, Hex & Integer Representation

•••

Representing and manipulating binary data

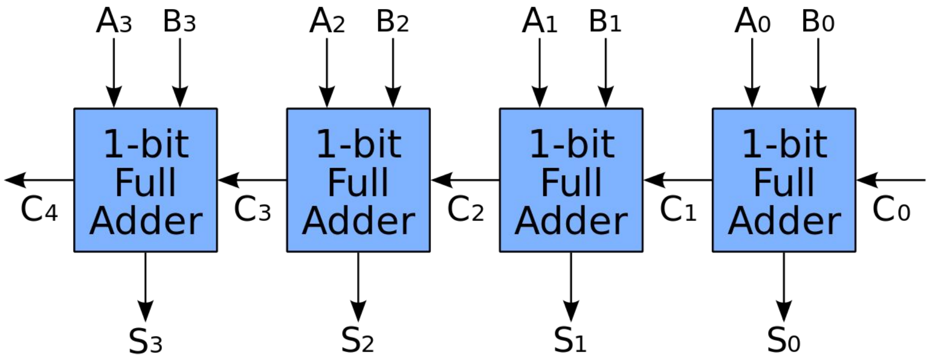
# Last Lecture

- In the last lecture we looked at
  - how binary data is stored in the computer
  - how an operation, addition, is implemented via logical gates



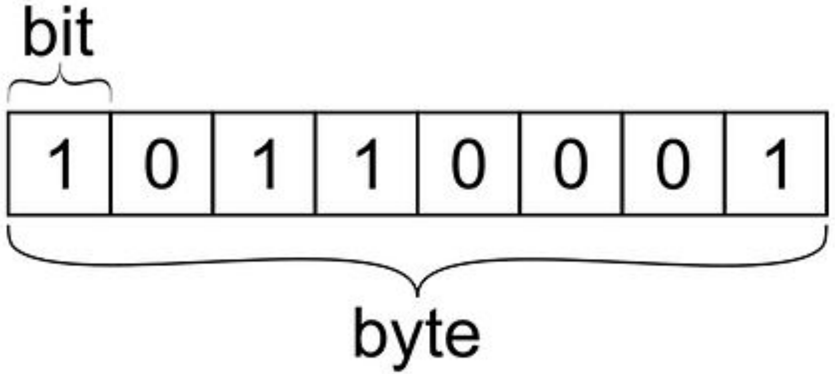
# This lecture

- We are going to move up a level and consider *data representations*
- At the machine leve, everything is *binary*
    - Different interpretive schemes can be imposed on this binary data



# Bytes

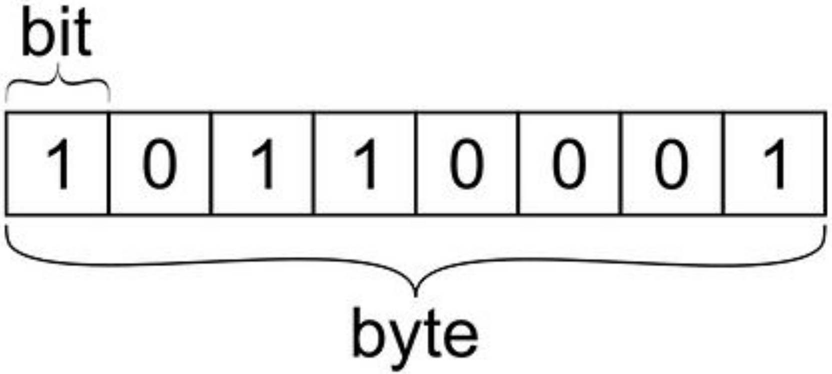
- The smallest group of bits is known as a *byte*
- A byte is 8 bits
- With 8 bits available we can represent 2^8 numbers
    - 0-255 unsigned



# Bytes

- Interpreting this value as an *unsigned integer* this value is:

  $(10110001)_2 = (1 \times 2^7) + (0 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = (177)_{10}$



# Bytes

- You will find the limit 255 in some funny places
- CSS color specification
  - Each color in R, G, B can have a value of 0-255

## Example

Define different RGB colors:

```
#p1 {background-color:rgb(255,0,0);} /* red */
#p2 {background-color:rgb(0,255,0);} /* green */
#p3 {background-color:rgb(0,0,255);} /* blue */
```

Try it Yourself »

# ASCII

- ASCII is an example of a representation *imposed* on bytes
- Developed from *telegraph* code
- Work on standard began in 1960
  - Encodes 128 English characters into 7 bit integers

**USASCII code chart**

| $b_7 b_6 b_5$ → | | | | | 0 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1 0 0 | 1 0 1 | 1 1 0 | 1 1 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $b_4$ | $b_3$ | $b_2$ | $b_1$ | Column → / Row ↓ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 0 | 0 | 0 | 0 | NUL | DLE | SP | 0 | @ | P | ` | p |
| 0 | 0 | 0 | 1 | 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 0 | 0 | 1 | 0 | 2 | STX | DC2 | " | 2 | B | R | b | r |
| 0 | 0 | 1 | 1 | 3 | ETX | DC3 | # | 3 | C | S | c | s |
| 0 | 1 | 0 | 0 | 4 | EOT | DC4 | $ | 4 | D | T | d | t |
| 0 | 1 | 0 | 1 | 5 | ENQ | NAK | % | 5 | E | U | e | u |
| 0 | 1 | 1 | 0 | 6 | ACK | SYN | & | 6 | F | V | f | v |
| 0 | 1 | 1 | 1 | 7 | BEL | ETB | ' | 7 | G | W | g | w |
| 1 | 0 | 0 | 0 | 8 | BS | CAN | ( | 8 | H | X | h | x |
| 1 | 0 | 0 | 1 | 9 | HT | EM | ) | 9 | I | Y | i | y |
| 1 | 0 | 1 | 0 | 10 | LF | SUB | * | : | J | Z | j | z |
| 1 | 0 | 1 | 1 | 11 | VT | ESC | + | ; | K | [ | k | { |
| 1 | 1 | 0 | 0 | 12 | FF | FS | , | < | L | \ | l | \| |
| 1 | 1 | 0 | 1 | 13 | CR | GS | - | = | M | ] | m | } |
| 1 | 1 | 1 | 0 | 14 | SO | RS | . | > | N | ^ | n | ~ |
| 1 | 1 | 1 | 1 | 15 | SI | US | / | ? | O | _ | o | DEL |

# ASCII

- ASCII codes are shown at right
- When you work with string literals in C, this is what is actually being stored in memory
- NB: you can treat these as unsigned integers
  - We do that when converting a numeric char to its actual numeric value in the project!

ASCII BINARY ALPHABET

| | | | |
|---|---|---|---|
| A | 1000001 | N | 1001110 |
| B | 1000010 | O | 1001111 |
| C | 1000011 | P | 1010000 |
| D | 1000100 | Q | 1010001 |
| E | 1000101 | R | 1010010 |
| F | 1000110 | S | 1010011 |
| G | 1000111 | T | 1010100 |
| H | 1001000 | U | 1010101 |
| I | 1001001 | V | 1010110 |
| J | 1001010 | W | 1010111 |
| K | 1001011 | X | 1010111 |
| L | 1001100 | Y | 1011001 |
| M | 1001101 | Z | 1011010 |

# Hex

- *Hexadecimal* is another way to represent binary data
    - Unrelated to ASCII
    - Rather, an efficient way to specify groups of 4 bytes
- Uses 16 characters to represent a half byte *(a nibble)*
    - 0-9, then A-F

| DECIMAL | HEX | BINARY |
|---|---|---|
| 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 |
| 10 | A | 1010 |
| 11 | B | 1011 |
| 12 | C | 1100 |
| 13 | D | 1101 |
| 14 | E | 1110 |
| 15 | F | 1111 |

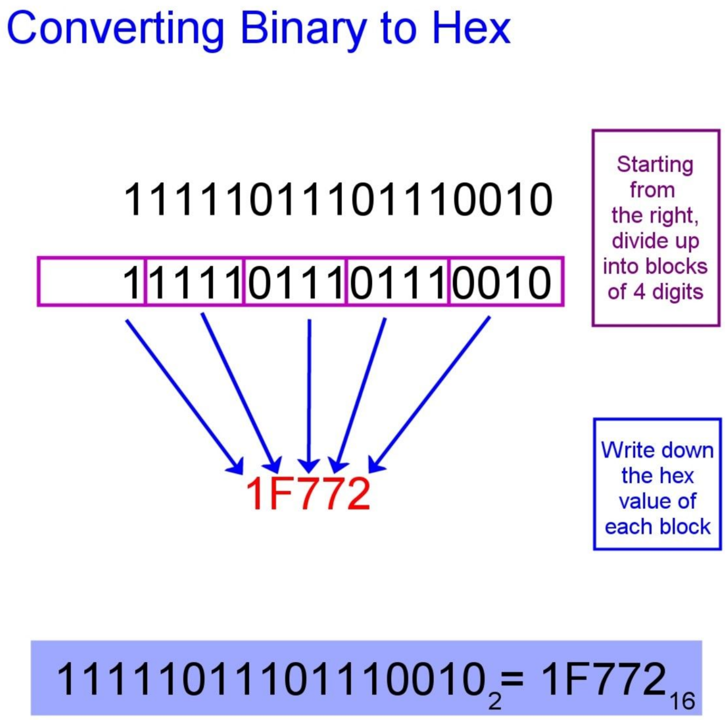
# Hex

- In written representation, hex is typically prefixed with an 0x:
  - 0x00F
- You can use this notation as a literal value in C, Java, etc.

| DECIMAL | HEX | BINARY |
|---|---|---|
| 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 |
| 10 | A | 1010 |
| 11 | B | 1011 |
| 12 | C | 1100 |
| 13 | D | 1101 |
| 14 | E | 1110 |
| 15 | F | 1111 |

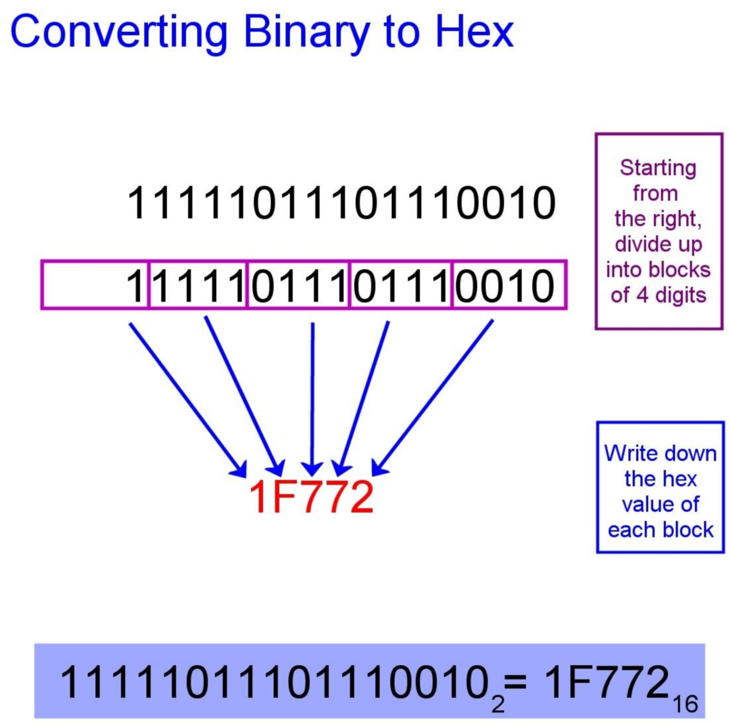
# Converting Binary To Hex

- Pretty straightforward
  - Group in nibbles (groups of 4 bits)
  - 0-9 -> converts to the same numeric character
  - 10-15 -> A, B, C, D, E, F



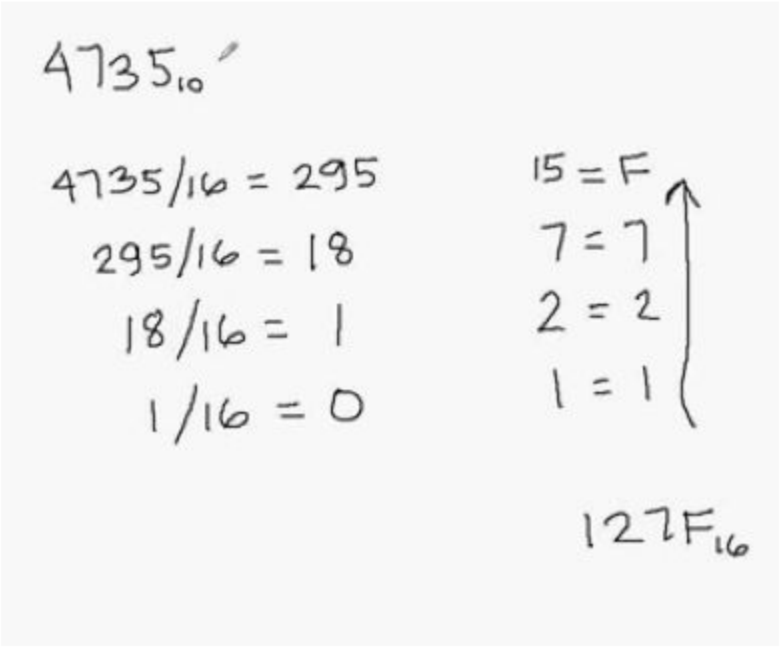
# Converting Decimal To Hex

- A bit more work
  - If number is < 16, just convert it directly
  - Else
    - Divide the number repeatedly by 16, writing down remainders, until value is < 16
    - The last value is the first number in the hex, add remainders in reverse order



# Converting Decimal To Hex

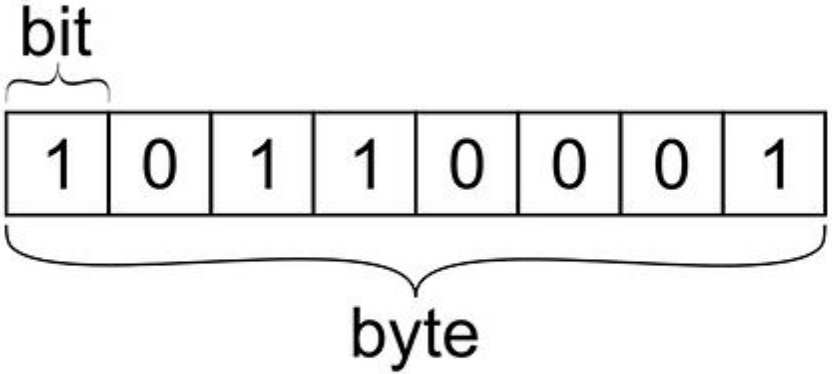
- A bit more work
  - If number is < 16, just convert it directly
  - Else
    - Divide the number repeatedly by 16, writing down remainders, until value is < 16
    - The last value is the first number in the hex, add remainders in reverse order



# Unsigned Integers

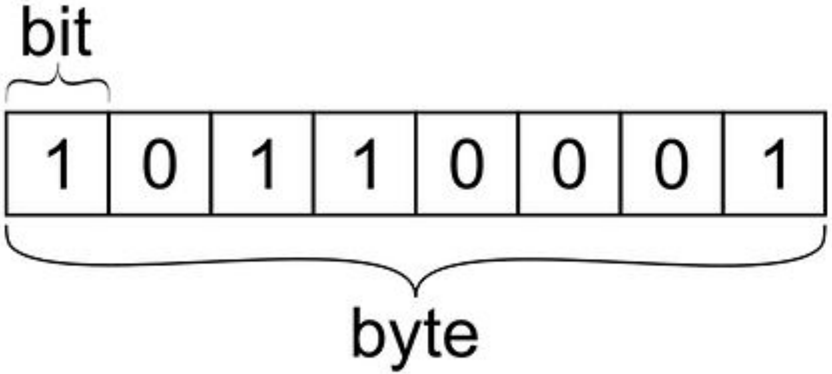
- Recall how unsigned integers in binary works

  $(10110001)_2 = (1 \times 2^7) + (0 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = (177)_{10}$



# Signed Integers

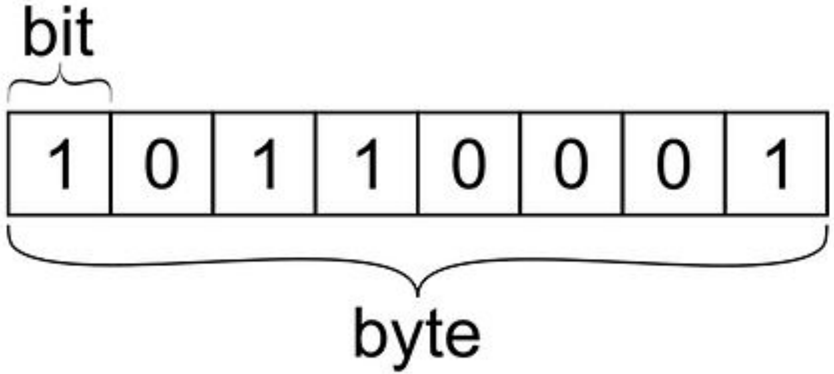
- How could we represent signed integers?
- Option 1: a sign bit
    - By convention, let's say that the most significant bit is the sign bit
    - What number is this using this encoding?



# Signed Integers

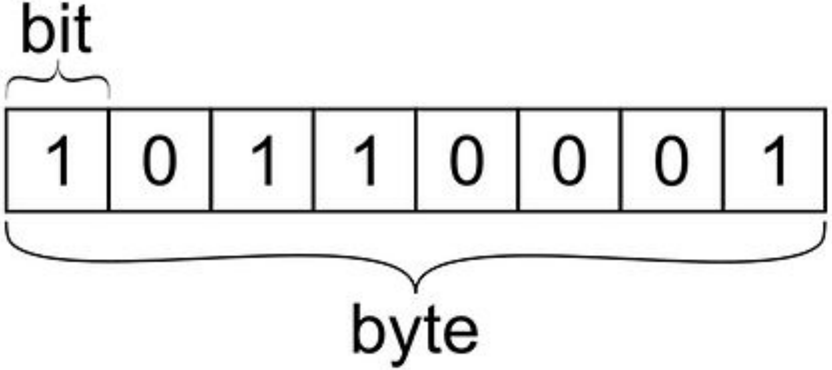
- Interpreting this value as an *unsigned integer* this value is:

  $(10110001)_2 = (-1) * ($
  $(0 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) +$
  $(0 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) +$
  $(1 \times 2^0)\ ) = (-49)_{10}$



# Signed Integers

- *Same bits* as unsigned 177
- The two different *values* we assigned to this bit pattern are the *encoding* we are using to interpret them



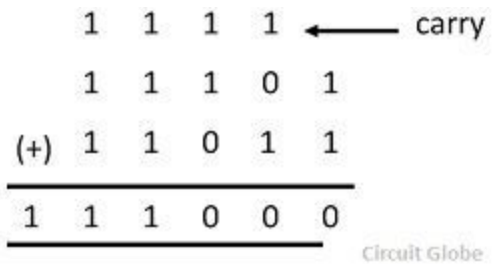
# Signed Integers

- Signed integers typically do not solely use a sign bit
- Instead, they use an encoding known as *2's complement*
- The encoding is shown at right
  - -128 is 10000000
  - -1 is 111111111

| | |
|---|---|
| 00000000 | 0 |
| 00000001 | 1 |
| ... | ... |
| 01111110 | 126 |
| 01111111 | 127 |
| 10000000 | −128 |
| 10000001 | −127 |
| 10000010 | −126 |
| ... | ... |
| 11111110 | −2 |
| 11111111 | −1 |

# Binary Addition

- Why on earth?
- To understand why, you need to understand how binary math works
- Let's look at addition
  - Note the carry bit from our earlier discussion on addition



# Binary Addition

- Let’s add 5 and -3
  - 5 rep: 0000 0101
  - -3 rep: 1111 1101
- Woah, just normal binary math gives us the correct result!
- 2’s complement has this wonderful feature
  - Math with negative numbers works in the same logical manner

$$5 + (-3) = 2$$

$$\begin{array}{rcl} 0000\ 0101 & = & +5 \\ +\ 1111\ 1101 & = & -3 \\ \hline 0000\ 0010 & = & +2 \end{array}$$

# Binary Addition

- Converting a number to 2’s complement:
    - Start with the positive binary rep
    - Subtract 1
    - Flip the bits
- Getting -3
    - Start with 3 = 0000 0011
    - Subtract 1, gives 2 = 0000 0010
    - Flip the bits = 1111 1101

$5 + (-3) = 2$

$$\begin{array}{rl} 0000\ 0101 & = +5 \\ +\ 1111\ 1101 & = -3 \\ \hline 0000\ 0010 & = +2 \end{array}$$

# Binary Finger Counting

- Did you know you can count to 31 on one hand?
- Let’s try it out

# Bytes, Hex & Integer Representation

- We took a look at how binary data can be encoded
- We looked as ASCII, a character encoding
- We looked at hexadecimal encoding
- We looked at integer encoding
- We discussed 2s complement, a scheme for encoding signed integers
    - 2s complement has a really nice property of allowing standard addition to work with negative numbers without a change
- *REMEMBER: IT'S JUST BITS*

MONTANA
STATE UNIVERSITY