# CMSC216: Binary Encoding of Data

Chris Kauffman

*Last Updated:*
*Mon Feb 23 06:03:15 PM EST 2026*

# Logistics

## Reading

**Bryant/O'Hallaron Ch 2.1-2.3**

- ▶ Binary Encoding of Data
- ▶ Signed/Unsigned Integers
- ▶ Character Data
- ▶ Optional: Ch 2.4 Floats

**Ch 3.1-7: Assembly, Arithmetic, Control**

### Goals

- ▶ Integer and character encoding
- ▶ Negative Values and Overflow/Underflow
- ▶ Bitwise Operations, Byte-Ordering
- ▶ Assembly Language Basics

## Assignments

- ▶ **Lab04**: GDB and Bit Operations
- ▶ **Project 2**: Posted

# Announcements

## Discussion Review Activity for Engagement Points

▶ In Discussion Wed, staff will give support on Exam 1 review and for students to finish up Project 1

▶ Staff will conduct a short review activity for Engagement Points

▶ The section that has the highest total score will get additional Engagement Points: attend Discussion to make your section the victor!

# Exam 1 Logistics

### Practice + Review

- ▶ Practice Exam 1A and 1B Posted with Solutions
- ▶ Practice 1B was part of discussion on Monday

### Exam 1

- ▶ In-person in class on Thu 19-Feb
- ▶ Exam runs lecture period: 75min
- ▶ Expect 2.5 pages front/back
- ▶ **Open Resource Exam**: review rules for this posted at bottom of course schedule (beneath slides)
  Questions on Open Resource Exam boundaries?

Note: Exam contents ends with C File I/O; no binary encoding (these slides) on Exam 1

# Unsigned Integers: Decimal and Binary

- ▶ Unsigned integers are always positive:
  **unsigned int** i = 12345;
- ▶ To understand their binary encoding, first recall how decimal numbers "work" to encode quantities

## Decimal: Base 10 Example
Each digit adds on a power 10

## Binary: Base 2 Example
Each digit adds on a power 2

$$
\begin{aligned}
80,345 = & 5 \times 10^0 + && 5 \text{ ones} \\
& 4 \times 10^1 + && 40 \text{ tens} \\
& 3 \times 10^2 + && 300 \text{ hundreds} \\
& 0 \times 10^3 + && 0 \text{ thousands} \\
& 8 \times 10^4 && 80 \text{ tens of thousands} \\
& 5 + 40 + 300 + 80,000
\end{aligned}
$$

$$
\begin{aligned}
11001_2 = & 1 \times 2^0 + && 1 \text{ ones} \\
& 0 \times 2^1 + && 0 \text{ twos} \\
& 0 \times 2^2 + && 0 \text{ fours} \\
& 1 \times 2^3 + && 8 \text{ eights} \\
& 1 \times 2^4 + && 16 \text{ sixteens} \\
& = 1 + 8 + 16 && = 25
\end{aligned}
$$

So, $11001_2 = 25_{10}$

# Exercise: Convert Binary to Decimal

Base 2 Example:

$$
\begin{aligned}
11001 = & 1 \times 2^0 + & 1 \\
& 0 \times 2^1 + & 0 \\
& 0 \times 2^2 + & 0 \\
& 1 \times 2^3 + & 8 \\
& 1 \times 2^4 + & 16 \\
= & 1 + 8 + 16 & = 25
\end{aligned}
$$

So, $11001_2 = 25_{10}$

**Try** With a Neighbor

Convert the following two numbers from base 2 (binary) to base 10 (decimal)

- ► 111
- ► 11010
- ► 01100001

**Answers**: Convert Binary to Decimal

$$
\begin{aligned}
111_2 =& 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
=& 1 \times 4 + 1 \times 2 + 1 \times 1 \\
=& 7_{10} \\
11010_2 =& 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\
=& 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \\
=& 26_{10} \\
01100001_2 =& 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 \\
& + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
=& 0 \times 128 + \times 64 + 1 \times 32 + 0 \times 16 \\
& + 0 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 \\
=& 97_{10}
\end{aligned}
$$

Note: last example ignores leading 0's

## The Other Direction: Decimal to Binary

Converting a number from base 10 to base 2 is easily done using repeated division by 2; keep track of **remainders**

**Convert 124 to base 2**:

$$124 \div 2 = 62 \text{ rem } 0$$
$$62 \div 2 = 31 \text{ rem } 0$$
$$31 \div 2 = 15 \text{ rem } 1$$
$$15 \div 2 = 7 \text{ rem } 1$$
$$7 \div 2 = 3 \text{ rem } 1$$
$$3 \div 2 = 1 \text{ rem } 1$$
$$1 \div 2 = 0 \text{ rem } 1$$

▶ Last step got 0 quotient so we're done.
▶ Binary digits are in **remainders in reverse**
▶ Answer: 1111100
▶ Check:
$0 + 0 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 = 4 + 8 + 16 + 32 + 64 = 124$

# Decimal, Hexadecimal, Octal, Binary Notation

▶ Numbers exist independent of any writing system
▶ Can write the same number in a variety of bases
▶ C provides syntax for most common bases used in computing

|             | Decimal | Binary        | Hex        | Octal     |
|-------------|---------|---------------|------------|-----------|
| Base        | 10      | 2             | 16         | 8         |
| Mathematical| 125     | $1111101_2$   | $7D_{16}$  | $175_8$   |
| C Prefix    | None    | 0b...         | 0x..       | 0...      |
| C Example   | 125     | 0b1111101     | 0x7D       | 0175      |
| printf()    | "%d"    | N/A           | "%x"       | "%o"      |

▶ **Hexadecimal** often used to express long-ish byte sequences
  Larger than base 10 so for 10-15 uses letters A-F
▶ **Examine** number_writing.c and table.c for patterns
▶ **Expectation**: Gain familiarity with doing conversions between
  bases as it will be useful in practice

# Hexadecimal: Base 16

- Hex: compact way to write bit sequences
- One byte is 8 bits
- Each Hex character represents 4 bits
- **Each Byte is 2 Hex Digits**

```
|-----------+----------------+-----|
| Byte      | Hex            | Dec |
|-----------+----------------+-----|
| 0101 0111 | 57 = 5*16 + 7  |  87 |
| 5      7  |                |     |
| 0011 1100 | 3C = 3*16 + 12 |  60 |
| 3    C=12 |                |     |
| 1110 0010 | E2 = 14*16 + 2 | 226 |
| E=14 2    |                |     |
|-----------+----------------+-----|
```

Hex to 4 bit equivalence

| Dec | Bits | 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   |

# Exercise: Conversion Tricks for Hex and Octal

### What tricks are illustrated for Hex/Octal Conversions?

```
|---------+--------------+-----------------+---------------------|
| Decimal | Byte = 8bits | Byte by 4       | Hexadecimal         |
|---------+--------------+-----------------+---------------------|
|      87 |     01010111 | bin: 0101 0111  | 57 = 5*16 + 7       |
|         |              | hex: 5    7     | hex dec             |
|         |              |                 |                     |
|      60 |     00111100 | bin: 0011 1100  | 3C = 3*16 + 12      |
|         |              | hex: 3    C=12  | hex dec             |
|         |              |                 |                     |
|     226 |     11100010 | bin: 1110 0010  | E2 = 14*16 + 2      |
|         |              | hex: E=14 2     | hex dec             |
|---------+--------------+-----------------+---------------------|
| Decimal | Byte = 8bits | Byte by 3       | Octal               |
|---------+--------------+-----------------+---------------------|
|      87 |     01010111 | bin: 01 010 111 | 127 = 1*8^2 + 2*8 + 7 |
|         |              | oct: 1  2   7   | oct   dec           |
|         |              |                 |                     |
|      60 |     00111100 | bin: 00 111 100 | 074 = 0*8^2 + 7*8 + 4 |
|         |              | oct: 0  7   4   | oct   dec           |
|         |              |                 |                     |
|     226 |     11100010 | bin: 11 100 010 | 342 = 3*8^2 + 4*8 + 2 |
|         |              | oct: 3  4   2   | oct   dec           |
|---------+--------------+-----------------+---------------------|
```

# **Answers**: Conversion Tricks for Hex and Octal

► Converting between Binary and Hexadecimal is easiest when grouping bits by 4: each 4 bits corresponds to one hexadecimal digit

```
bin: 0101 0111    bin: 1110 0010
hex: 5    7       hex: E=14 2
```

► Converting between Binary and Octal is easiest when grouping bits by 3: each 3 bits corresponds to one octal digit

```
bin: 01 010 111   bin: 11 100 010
oct: 1  2   7     oct: 3  4   2
```

# Character Coding Conventions

- ▶ Would be hard for people to share words if they interpretted bits as letters differently
- ▶ **ASCII**: American Standard Code for Information Interchange An old standard for bit/character correspondence
- ▶ 7 bits per character, includes upper, lower case, punctuation

| Dec | Hex | Binary | Char | Dec | Hex | Binary | Char |
|-----|-----|----------|------|-----|-----|----------|------|
| 65 | 41 | 01000001 | A | 78 | 4E | 01001110 | N |
| 66 | 42 | 01000010 | B | 79 | 4F | 01001111 | O |
| 67 | 43 | 01000011 | C | 80 | 50 | 01010000 | P |
| 68 | 44 | 01000100 | D | 81 | 51 | 01010001 | Q |
| 69 | 45 | 01000101 | E | 82 | 52 | 01010010 | R |
| 70 | 46 | 01000110 | F | 83 | 53 | 01010011 | S |
| 71 | 47 | 01000111 | G | 84 | 54 | 01010100 | T |
| 72 | 48 | 01001000 | H | 85 | 55 | 01010101 | U |
| 73 | 49 | 01001001 | I | 86 | 56 | 01010110 | V |
| 74 | 4A | 01001010 | J | 87 | 57 | 01010111 | W |
| 75 | 4B | 01001011 | K | 88 | 58 | 01011000 | X |
| 76 | 4C | 01001100 | L | 89 | 59 | 01011001 | Y |
| 77 | 4D | 01001101 | M | 90 | 5A | 01011010 | Z |
| 91 | 5B | 01011101 | [ | 97 | 61 | 01100001 | a |
| 92 | 5C | 01011110 | \ | 98 | 62 | 01100010 | b |

Partial Table of ASCII Codes / Values, try man 7 ascii in a terminal for a full table

# Exercise: Characters vs Numbers

Explain the following program and its output

```
1 // char_ints.c:
2 #include <stdio.h>
3 #include <string.h>
4 int main(){
5   ...
6   char nums[64] = {
7     72, 101, 108, 108, 111, 32,
8     87, 111, 114, 108, 100, 33,
9     0
10  };
11  printf("%s\n",nums);
12  len = strlen(nums);
13  for(int i=0; i<len; i++){
14    printf("[%2d] %c %3d %02X\n",
15           i,nums[i],nums[i],nums[i]);
16  }
17  return 0;
18 }
```

```
>> gcc char_ints.c
>> ./a.out
...
Hello World!
[ 0] H  72 48
[ 1] e 101 65
[ 2] l 108 6C
[ 3] l 108 6C
[ 4] o 111 6F
[ 5]    32 20
[ 6] W  87 57
[ 7] o 111 6F
[ 8] r 114 72
[ 9] l 108 6C
[10] d 100 64
[11] !  33 21
```

# **Answers**: Characters vs Numbers

## The Whole Array

```c
char nums[64] = {
  72, 101, 108, 108, 111, 32,
  87, 111, 114, 108, 100, 33,
  0
};
```

Lays out a bit pattern at each spot the array; bit pattern is specified with decimal numbers

```c
printf("%s\n",nums);
```

Print the array as though it were "string": an array of characters that is null terminated

## Elements of the Array

```c
printf("[%2d] %c %3d %02X\n",
       i,nums[i],nums[i],nums[i]);
```

Print a single element of the array as

- ► %c : a character (ASCII table lookup for the glyph to draw)
- ► %3d : a decimal number (padding to width 3)
- ► %02X : as a hexadecimal number (with leading 0's if needed and padded with width 2 - *noice!*)

# Unicode

- *World*: Why can't I write

  컴퓨터

  in my code/web address/email?
- *America*: ASCII has 128 chars. Deal with it.
- *World*: Seriously?
- *America*: We invented computers. 'Merica!

- *World*:
- *America*: ... Unicode?
- *World*: But my language takes more bytes than American.
- *America*: Deal with it. 'Merica!

- ASCII Uses 7 bits per char, limited to 128 characters
- UTF-8 uses **1-4 bytes per character** to represent **many** more characters (1,112,064 *codepoints*)
- Uses 8th bit in a byte (**high-order bit**) to indicate extension to multiple bytes
- Requires software to understand coding convention allowing broader language support
- ASCII is a proper subset of UTF-8 making UTF-8 backwards compatible and wildly popular

# Binary Integer Addition/Subtraction

Adding/subtracting in binary works the same as with decimal
EXCEPT that carries occur on values of 2 rather than 10

```
   ADDITION #1              SUBTRACTION #1
     1 11    <-carries                ? <-carries
     0100 1010 = 74          0111 1001 = 121
   + 0101 1001 = 89        - 0001 0011 =  19
   ------------            ------------
     1010 0011 = 163          VVVVVVVVVVVV
                              VVVVVVVVVVVV
   ADDITION #2              VVVVVVVVVVVV
     1111   1 <-carries              x12 <-carries
     0110 1101 = 109         0111 0001 = 119
   + 0111 1001 = 121       - 0001 0011 =  19
   ------------            ------------
     1110 0110 = 230          0110 0110 = 102
```

When 0/1 is represented as Low/High Voltage, one can design
digital circuits that implement arithmetic

# Two's Complement Integers: Representing Negative Values

- ▶ To represent negative integers, must choose a **different** coding system than for positive-only integers
- ▶ The **Two's Complement Encoding** is the most common coding system for signed numbers so we will study it
- ▶ Alternatives exist
  - ▶ Signed magnitude: leading bit indicates pos (0) or neg (1)
  - ▶ One's complement: invert bits to go between positive negative
- ▶ Great advantage of two's complement: **signed and unsigned arithmetic are identical**
- ▶ Hardware folks only need to make one set of units for both unsigned and signed arithmetic

## Summary of Two's Complement

TL;DR: *Most significant bit is a negative power of two.*

```
UNSIGNED BINARY         TWO's COMPLEMENT (signed)
---------------         -------------------------
7654 3210 : position    7654 3210 : position
ABCD EFGH : 8 bits      ABCD EFGH : 8-bits
A: 0/1 * +(2^7) *POS*    A: 0/1 * -(2^7) *NEG*
B: 0/1 * +(2^6)         B: 0/1 * +(2^6)
C: 0/1 * +(2^5)         C: 0/1 * +(2^5)
...                     ...
H: 0/1 * +(2^0)         H: 0/1 * +(2^0)

UNSIGNED BINARY         TWO's COMPLEMENT (signed)
---------               ---------
7654 3210 : position    7654 3210 : position
1000 0000 = +128        1000 0000 = -128
1000 0001 = +129        1000 0001 = -127 = -128+1
1000 0011 = +131        1000 0011 = -125 = -128+1+2
1111 1111 = +255        1111 1111 = -1   = -128+1+2+4+..+64
0000 0000 = 0           0000 0000 =  0         [   +127   ]
0000 0001 = +1          0000 0001 = +1
0000 0101 = +5          0000 0101 = +5
0111 1111 = +127        0111 1111 = +127
```

# Two's Complement Notes

## Unsigned/Signed Equivalents

```
Unsigned   1000 0110 = 134
Signed     1000 0110 = -121
           = 134 - 256
Unsigned   1111 0001 = 241
Signed     1111 0001 = -15
           =241-256
Unsigned   0011 0011 = 51
Signed     0011 0011 = 51
```

### When/Why X-256?

▶ Leading (leftmost) bit is 1
▶ Counted as 128 in Unsigned
▶ Counts as -128 in Signed
▶ Subtract 256 to compensate

## Negation in Two's Complement

```
int y = -x;
```

▶ Unary Minus operator
▶ **Invert bits, Add 1**
▶ Works for both Pos→Neg and Neg→Pos

```
~ 0110 1000  +104 : negate
-----------
  1001 0111  inverted
+         1
-----------
  1001 1000 = -104

~ 1001 1000 = -104 : negate
-----------
  0110 0111 = +103 inverted
+         1
-----------
  0110 1000 = +104
```

# Exercise: Two's Complement Conversions

Fill in the missing entries in the following table

|  | Bits |  | Hex | Decimal Unsigned | Decimal Signed |
|---|---|---|---|---|---|
|  | 1111 1111 | A: ____ | | B: ___ | C: ___ |
|  | 1001 0110 | | 0x96 | D: ___ | E: ___ |
| F: ____ ____ | | | 0x3E | G: ___ | H: ___ |
|  | 0010 0011 | I: ____ | | 35 | J: ___ |
| K: ____ ____ | | L: ____ | | M: ___ | -35 |

# Answers: Two's Complement Conversions

|    | Bits      |    | Hex    | Decimal Unsigned | Decimal Signed |
|----|-----------|----|--------|------------------|----------------|
|    | 1111 1111 | A: | 0xFF   | B: 255           | C: -1          |
|    | 1001 0110 |    | 0x96   | D: 150           | E: -106        |
| F: | 0011 1110 |    | 0x3E   | G: 62            | H: 62          |
|    | 0010 0011 | I: | 0x23   | 35               | J: 35          |
| K: | 1101 1101 | L: | 0xDD   | M: 221           | -35            |

K / L / M: Converting 35 to -35 decimal/bits can be done via
(-35+255) AND/OR via Invert Bits + 1

# Overflow

- Sums that exceed the representation of the bits associated with the integral type **overflow**
- Excess significant bits are **dropped**
- Addition can result in a sum smaller than the summands, even for two positive numbers (!?)
- Integer arithmetic in fixed bits is a mathematical **ring**

## Examples of Overflow in 8 bits

```
ADDITION #3 OVERFLOW    ADDITION #4 OVERFLOW
 1 1111 111 <-carries    1        1 <-carries
   1111 1111 = 255         1010 1001 = 169
+ 0000 0001 =   1       + 1100 0001 = 193
------------            ------------
 1 0000 0000 = 256       1 0110 1010 = 362
 x drop 9th bit          x drop 9th bit
------------            ------------
   0000 0000 = 0          0110 1010 = 106
```

# Underflow

- **Underflow** occurs in unsigned arithmetic when values go below 0 (no longer positive)
- Pretend that there is an extra significant bit to carry out subtraction
- Subtracting a positive integer from a positive integer may result in a **larger** positive integer (?!?)
- Integer arithmetic in fixed bits is a mathematical **ring**

## Examples of 8-bit Underflow

```
SUBTRACTIION #2 UNDERFLOW
          ?<-carries
   0000 0000 =   0
 - 0000 0001 =   1
 ------------
 VVVVVVVVVVVV
          ?<-carries
 1 0000 0000 = 256 (pretend)
 - 0000 0001 =   1
 ------------
 VVVVVVVVVVVV
   x       2<-carries
 0 1111 1110 = 256
 - 0000 0001 =   1
 ------------
   1111 1111 = 255
```

# Overflow and Underflow In C Programs

▶ See `over_under_flow.c` for demonstrations in a C program.

▶ **No runtime errors** for under/overflow

▶ Good for hashing and cryptography

▶ Bad for most other applications: system critical operations should use checks for overflow / underflow

▶ Textbook mentions the Ariane Rocket Crash which was due to overflow of an integer converted from a floating point value

> *The Ariane explosion is an instructive example for several reasons.*
> *(1) Software re-use caused the problem subverting the usual wisdom of relying on tested software; hardware changes ALWAYS trump software.*
> *(2) Sometimes computer science IS rocket science*

▶ Assembly provides condition codes indicating when overflow occurs but checking in C is tricky and painful[1]

---

[1]Many compilers like GCC can generate assembly instructions that will detect overflow and abort programs. See the demo program `overflow_detect.c` and GCCs `-ftrapv` option.

# Integer Ops and Speed

▶ Along with Addition and Subtraction, **Multiplication and Division** can also be done in binary

▶ Algorithms are the same as base 10 but more painful to do by hand

▶ This pain is reflected in hardware speed of these operations

▶ The **Arithmetic and Logic Unit (ALU)** does integer ops in the machine

▶ A **clock** ticks in the machine at some rate like 3Ghz (3 billion times per second)

▶ Under ideal circumstances, typical ALU Op speeds are

| Operation | Cycles |
|---|---|
| Addition | 1 |
| Logical | 1 |
| Shifts | 1 |
| Subtraction | 1 |
| Multiplication | 3 |
| Division | >30 |

▶ Due to disparity, it is worth knowing about **relation** between multiply/divide and **bitwise** operations

▶ Compiler often uses such tricks: shift rather than multiply/divide

# Mangling Bits Puts Muscle on Your Bones

Below illustrates difference between logical and bitwise operations.

```
int xl = 12 || 10; // truthy (Logical OR)
int xb = 12 |  10; // 14     (Bitwise OR)
int yl = 12 && 10; // truthy (Logical AND)
int yb = 12 &  10; // 8      (Bitwise AND)
int zb = 12 ^  10; // 6      (Bitwise XOR)
int wl = !12;      // falsey (Logical NOT)
int wb = ~12;      // 3      (Bitwise NOT/INVERT)
```

▶ Bitwise ops evaluate on a per-bit level

▶ 32 bits for int, 4 bits shown

```
Bitwise OR      Bitwise AND     Bitwise XOR     Bitwise NOT
  1100 = 12       1100 = 12       1100 = 12
| 1010 = 10     & 1010 = 10     ^ 1010 = 10     ~ 1100 = 12
-----------     -----------     -----------     -----------
  1110 = 14       1000 =  8       0110 =  6       0011 =  3
```

# Bitwise Shifts

- **Shift** operations move bits within a field of bits
- Shift operations are
  ```
  x = y << k; // left  shift y by k bits, store in x
  x = y >> k; // right shift y by k bits, store in x
  ```
- All integral types can use shifts: `long`, `int`, `short`, `char`
- **Not applicable** to pointers or floating point
- Examples in 8 bits
  ```
  //          76543210
  char x = 0b00010111; // 23
  char y = x << 2;     // left shift by 2
  //    y = 0b01011100; // 92
  //    x = 0b00010111; // not changed
  char z = x >> 3;     // right shift by 3
  //    z = 0b00000010; // 2
  //    x = 0b00010111; // not changed
  char n = 0b10000000; // -128, signed
  char s = n >> 4;     // right shift by 4
  //    s = 0b11111000; // -8, sign extension
  //    right shift >> is "arithmetic"
  ```

# Shifty Arithmetic Tricks

- ▶ Shifts with add/subtract can be used instead of multiplication and division
- ▶ Turn on optimization: `gcc -O3 code.c`
- ▶ Compiler automatically does this if it thinks it will save cycles
- ▶ *Sometimes* programmers should do this but better to convince compiler to do it for you, **comment** if doing manually

## Multiplication

```
//           76543210
char  x = 0b00001010;  // 10
char x2 = x << 1;      // 10*2
//   x2 = 0b00010100;  // 20
char x4 = x << 2;      // 10*4
//   x4 = 0b00101000;  // 40
char x7 = (x << 3)-x;  // 10*7
//   x7 = (x * 8)-x;   // 10*7
//   x7 = 0b01000110;  // 70
//           76543210
```

## Division

```
//           76543210
char y = 0b01101110;   // 110
char y2 = y >> 1;      // 110/2
//   y2 = 0b00110111;  // 55
char y4 = y >> 2;      // 110/4
//   y4 = 0b00011011;  // 27
char z = 0b10101100;   // -84
char z2 = z >> 2;      // -84/4
//   z2 = 0b11101011;  // -21
//   right shift sign extension
```

# Exercise: Checking / Setting Bits

Use a combination of bit shift / bitwise logic operations to...
1. Check if bit i of `int x` is set (has value 1)
2. Clear bit i (set bit at index i to value 0)

Show C code for this

```c
{
  int x = ...;
  int i = ...;
  if( ??? ) { // ith bit of x is set
    printf("set!\n");
  }

  i = ...;
  ???;
  printf("ith bit of x now cleared to 0\n");
}
```

## Answers: Checking / Setting Bits

1. Check if bit i of int x is set (has value 1)

```
int x = ...;
int mask = 1; // or 0b0001 or 0x01 ...
int shifted = mask << i; // shifted   0b00...010..00
if(x & shifted){         //       x & 0b10...010..01
  ...                    //         ------------------
}                        //           0b00...010..00
```

2. Clear bit i (set bit at index i to value 0)

```
int x = ...;
int mask = 1; // or 0b0001 or 0x01 ...
int shifted  = mask << i;  // shifted   0b00...010..00
int inverted = ~shifted;   // inverted 0b11...101..11
x = x & inverted;          //       x & 0b10...010..01
  ...                      //         ------------------
                           //           0b10...000..01
```

# Showing Bits

- ▶ `printf()` capabilities:
  - %d    as Decimal
  - %x    as Hexadecimal
  - %o    as Octal
  - %c    as Character
- ▶ No specifier for binary
- ▶ Can construct such with bitwise operations
- ▶ Code pack contains two codes to do this
  - ▶ `printbits.c`: single args printed as 32 bits
  - ▶ `showbits.c`: multiple args printed in binary, hex, decimal

- ▶ Showing bits usually involves shifting and bitwise AND &
- ▶ Example from `showbits.c`

```
#define INT_BITS 32

// print bits for x to screen
void showbits(int x){
  for(int i=INT_BITS-1; i>=0; i--){
    int mask = 1 << i;
    if(mask & x){
      printf("1");
    } else {
      printf("0");
    }
  }
}
```

# Bit Masking

- ▶ Semi-common for functions to accept bit patterns which indicate true/false options
- ▶ Frequently makes use of bit **masks** which are constants associated with specific bits
- ▶ Example: Unix permissions might be...

```
#define S_IRUSR 0b100000000 // User   Read
#define S_IWUSR 0b010000000 // User   Write
#define S_IXUSR 0b001000000 // User   Execute
#define S_IRGRP 0b000100000 // Group  Read
...
#define S_IWOTH 0b000000010 // Others Write
#define S_IXOTH 0b000000001 // Others Execute
```

- ▶ Use them to create options to C functions like

```
int permissions = S_IRUSR|S_IWUSR|S_RGRP;
chmod("/home/kauffman/solution.zip",permissions);
```

# Unix Permissions with Octal

- ▶ Octal arises associated with **Unix file permissions**
- ▶ Every file has 3 permissions for 3 entities
- ▶ Permissions are true/false so a single bit will suffice

▶ `ls -l`: long list files, shows permissions

▶ `chmod 665 somefile.txt`: change permissions of `somefile.txt` to those shown to the right

▶ `chmod 777 x.txt`: read / write / exec for everyone

▶ `chmod` also honors letter versions like `r` and `w`

▶ `chmod u+x script.sh #` make file executable

```
binary      octal
110110101 = 665
rw-rw-r-x somefile.txt
U  G  O
S  R  T
E  O  H
R  U  E
   P  R

Readable chmod version:
chmod u=rw,g=rw,o=rx somefile.txt
```

# Endinaness: Byte ordering in Memory

▶ Single bytes like ASCII characters lay out sequentially in memory in increasing address
▶ Multi-byte entities like 4-byte `ints` require decisions on byte ordering
▶ We think of a 32-bit int like this

```
            Most Signifcant <------> Least Significant
  Binary:   0000 0000  0000 0000  0001 1000  1110 1001
            0    0     0    0     1    8     E    9
  Hex  :    000018E9
  Decimal:  6377
```

▶ There are 2 Options to for ordering multi-byte data in memory
  ▶ **Little Endian**: Least Significant byte at low address
  ▶ **Big Endian**: Most Significant Byte at low address
▶ Example: Integer starts at address #1024

```
            Address
  LittleEnd: #1027      #1026      #1025      #1024
  Binary:    0000 0000  0000 0000  0001 1000  1110 1001
             0    0     0    0     1    8     E    9
  BigEnd:    #1024      #1025      #1026      #1027
             Address
```

# Little Endian vs. Big Endian

▶ Most modern machines use **Little Endian** ordering by default

▶ Some processor (ARM) support both Little / Big Endian BUT and one is chosen at startup and used until turned off

▶ Both Big and Little Endian have (minor) engineering trade-offs

▶ At one time debated hotly among hardware folks: a la Gulliver's Travels conflicts

▶ **Intel Chips** were little endian and have dominated computing for several decades, set the precedent for modern platforms

▶ Big endian byte order shows up in **network programming**: sending bytes over the network is done in big endian ordering

▶ **Examine** show_endianness.c : uses C code to print bytes in order, reveals whether a machine is Little or Big Endian

# Output of `show_endianness.c`

```
 1  // show_endianness.c: Shows endiannes layout of a binary number in
 2  // memory. Intel machines and some ARM machines (Apple M1) are little
 3  // endian so bytes will print least signficant earlier.
 4  #include <stdio.h>
 5
 6  int main(){
 7    int bin = 0b00000000000000000001100011101001;    // 6377
 8    //          |    |    |    |    |    |    |    |
 9    //          0    0    0    0    1    8    e    9
10    printf("%d\n%08x\n",bin,bin);              // show decimal and hex representation of bi
11    char *ptr = (char *) &bin;                 // pointer to beginning of bin
12    for(int i=0; i<4; i++){                     // print bytes of bin from low to high
13      printf("%hhx ", ptr[i]);                  // memory address
14    }                                           // '%hhx' : 1-byte char in hex
15    printf("\n");                               // '%hx'  : 2-byte short in hex
16    return 0;                                   // '%x'   : 4-byte int in hex
17  }
    >> gcc show_endianness.c
    >> ./a.out
    6377
    000018e9
    e9 18 0 0
```

**Notice:** num prints with value 18e9 but bytes appear in reverse
order e9 18 when run on a Little Endian machine: the "littlest"
byte appears earliest in memory