

CSI333 Lecture 4

C-string format with scanf/printf

```
char myCString[4];
int intVar;
scanf("%3s", &intVar ); /*reads up to 3 chars
and stores them PLUS \0 in the 4-byte var. intVar*/
scanf("%3s", myCString); /*DITTO into the 4-byte
byte array*/
/* DIFFERENT from other output formats! */
printf("%s", &intVar); /* Print a C-String! */
printf("%s", myCString); /* Another C-string */
```

C strings are null-terminated char arrays

- ▶ They go by the address of their first char.
- ▶ In C/C++, with array `char myCharArray[56];`
`myCharArray` (no brackets!) denotes the (const) ADDRESS OF the first character.
- ▶ `myCharArray` is **equivalent to** `&myCharArray[0]`

Bit - Binary Digit

Basic Unit of Information stored and manipulated in our computers.

Computer ¹ hardware stores & manipulates & transmits all data **digitally**: which means with on/off, 0/1, **true/false** (usually) electrical² signals.

A bit (binary digit) is a single unit of memory or transmission that can have only 2 possible values.

¹and much other popular electronic product

²also optical and magnetic

A representation is a coding scheme to give meaning to bit strings (sequences).

Different kinds of data are each represented by different meanings we give to sequences of bits.

Some hardware (printers, keyboards e.g.) embodies particular representations: 01000010 makes standard printers print **B**

The base or radix 2 (binary) number system is used by present computers to represent non-negative integers.

Example: 0001 1100 1000 0110

$$\begin{aligned} &0 \cdot 2^{15} + 0 \cdot 2^{14} + 0 \cdot 2^{13} + 1 \cdot 2^{12} \\ &+ 1 \cdot 2^{11} + 1 \cdot 2^{10} + 0 \cdot 2^9 + 0 \cdot 2^8 \\ &+ 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 \\ &+ 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \end{aligned}$$

$$\begin{aligned} &= (0+0+0+4096)+(2048+1024+0+0)+(256+0+0+0)+(0+4+2+0) \\ &= 7302 \end{aligned}$$

Computer Experts all know their powers of 2:

2^1	2	2^{11}	$2048 = 2Ki$
2^2	4	2^{12}	$4096 = 4Ki$
2^3	8	2^{13}	$8192 = 8Ki$
2^4	$16_{\text{ten}} = 0 \times 10$	2^{14}	$16,384 = 16Ki$
2^5	$32_{\text{ten}} = 0 \times 20$	2^{15}	$32,768 = 32Ki$
2^6	$64_{\text{ten}} = 0 \times 40$	2^{16}	$65,536 = 64Ki$
2^7	$128_{\text{ten}} = 0 \times 80$	2^8	256
2^9	512	2^{10}	$1024 = 1kibi = 1Ki$
2^{20}	$1,048,576 = 1Mi$ $\approx 10^6$	2^{30}	$1,073,741,824 = 1Gi$ $\approx 10^9$

"Thus 1024 bytes of storage is officially a kibibyte, not a kilobyte. However, computer professionals generally dislike this unit (they say it sounds like a cat food) so the ambiguity in the size of a kilobyte persists. The prefix is a contraction of "kilobinary." The symbol Ki-, rather than ki-, was chosen for uniformity with the other binary prefixes (Mi-, Gi-, etc.)." (Russ Rowlett, UNC)

How to convert a number binary; SAME IDEA for converting a number to English!

Let X be a variable initialized with the number. Repeat until done:

1. If $X==0$ write the remaining bits and go home. Otherwise, guess or figure out (by dividing) the largest power of 2 that “fits”

Specifically, what is the largest power of two which is *less than or equal to* X ?

2. Write zeros for any bits that should be zero. (for English, write nothing.) Write the bit for that power of 2.
3. Compute $X = X - (\text{that power of } 2)$.

Example? 7302.

How to access individual bits in C/C++/Java (1) Bitwise Logical Operators (2) Shift operators

Bitwise Logical Operators

apply to integer type objects (char, short, int, long int and their unsigned counterparts³) and give integer results defined in terms of **bits**.

Advice: Use C/C++ unsigned long int for bitstrings when 32 bits are needed. (long guarantees 32 bits from ANSI C.)

“All unsigned types use straight binary notation, regardless of whether the signed types use 2-s complement, 1-s compl. or sign magnitude ... the sign bit is treated as an ordinary bit.” (Harbison & Steele)⁴

³also bool **after conversion to int**

⁴C/C++ on general purpose computers uses 2-s complement integers, but other sign representations are possible.

Since the LC-3 Instruction Set Architecture

(1) uses (like all others) bit fields in the machine language code,

(2) and has bitwise operations,

C++ bitwise operations are useful to simulate LC-3 in

C++/C/Java.

First, single bit “Boolean” or truth value operations:

Unary “NOT” (complement): (also denoted by \neg , \bar{x} , \sim)

x	NOT(x)
0	1
1	0

Binary “AND,” “(inclusive) OR,” “EXCLUSIVE OR”:

(Binary means “**TWO OPERANDS**” here)

x	y	x AND y	x OR y	x XOR y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Other Symbols: $\wedge, \&, \cdot$ | $\vee, |, +$ | eor, \oplus

Other names for logic values:

- ▶ **1**: true, on, asserted, enabled
- ▶ **0**: false, off, deasserted, disabled

“on, asserted, enabled” etc. are popular in electrical and computer engineering circles.

C++ Bitwise Operators apply Boolean operations to the **individual bits** of the binary integer representations.

```
unsigned char X = 0x0C; // 0000 1100 (binary)
unsigned char Y = 0x0A; // 0000 1010
    (X & Y) == 0x04 // 0000 1000  AND
    (X | Y) == 0x0E // 0000 1110  OR
    (X ^ Y) == 0x06 // 0000 0110  EOR
    (~ X) == 0xF3 // 1111 0011  COMPL
```

Caution: It's a common bug to confuse Bitwise operations with Logical AND, OR, NOT:

`(X && Y) == 1` if `X!=0` and `Y!=0`, 0 otherwise

`(X || Y) == 1` if `X!=0` or `Y!=0`, 0 otherwise

`(! X) == 0` if `X!=0`, 1 if `X==0`

- ▶ In C/C++, consider any non-zero int or pointer as “true,” and 0 as “false.”
- ▶ Many C programmers write `#define true 1` and `#define false 0`
- ▶ In C++/Java, true and false are literals (constants) of type `bool` (boolean in Java).
- ▶ (In C++, non-zero ints or pointers are converted to `bool` true, zero converts to false, true converts to 1 and false to 0).

Shifts For integral X, $\text{AMT}, \text{AMT} \geq 0$

- ▶ $X \ll \text{AMT}$ is X shifted Left AMT bit positions. Zero bits are shifted in from the right.
- ▶ Example: $(0x000F8001 \ll 3) == 0x007C0008$
- ▶ For unsigned X or $X \geq 0$
 $X \gg \text{AMT}$ is X shifted Right AMT bit positions. Zero bits are shifted in from the left.
- ▶ Example: $(0x007C0008 \gg 3) == 0x000F8001$
- ▶ For signed X, $X < 0$, in $X \gg \text{AMT}$,
**whether 0s or 1s are shifted in is IMPLEMENTATION
DEPENDENT!!**

Bitwise Op. Applications

1. Extract or compose bit fields when format is externally defined. (Hardware simulation, device driver software, network packet analysis/synthesis).
2. Work on small, fixed universe subsets efficiently. (HS 7.6, ios flags, Strou. (6.2.4) and p.616-7.)
3. Efficient special case arithmetic operations:

```
if( X & 3 ) { /* X is not a multiple of 4 */ }  
Y = X & (~0x3FF);  
    /* round down to nearest 1K multiple */  
if( (X & 1) == 0 ){ /* X is even */ }
```

How would you sort, using about 1 Megabyte of memory, a few million 7 digit telephone numbers from a disk file once every 1/2 hour, so that we can rapidly tell if a number is assigned?

A discussion is given in Jon Bentley's "Programming Pearls" column in the December 1999 issue of *Dr. Dobb's Journal*.

How to access individual bits in C/C++/Java

Bitwise operations

- ▶ `&` Bitwise AND `int I; 0x8000000 & I` equals 0 if the top-order bit (bit 31) is 0; equals `0x80000000` $\neq 0$ if that bit is 1.
- ▶ `|` Bitwise OR `I = I | 0x8000000;` makes bit 31 of `I` become (or stay) 1. It “sets” bit 31.

How to access individual bits in C/C++/Java

Bitwise operations

- ▶ `&` Bitwise AND `int I; 0x8000000 & I` equals 0 if the top-order bit (bit 31) is 0; equals `0x8000000` $\neq 0$ if that bit is 1.
- ▶ `|` Bitwise OR `I = I | 0x8000000;` makes bit 31 of `I` become (or stay) 1. It “sets” bit 31.
What does `I = I & 0x7FFFFFFF;` do?

How to access individual bits in C/C++/Java

Bitwise operations

- ▶ `&` Bitwise AND `int I; 0x8000000 & I` equals 0 if the top-order bit (bit 31) is 0; equals `0x8000000` \neq 0 if that bit is 1.
- ▶ `|` Bitwise OR `I = I | 0x8000000;` makes bit 31 of `I` become (or stay) 1. It “sets” bit 31.
What does `I = I & 0x7FFFFFFF;` do? “clears” bit 31.
- ▶ `~` Bitwise NOT `~0x7FFFFFFF = ?`

How to access individual bits in C/C++/Java

Bitwise operations

- ▶ & Bitwise AND `int I; 0x8000000 & I` equals 0 if the top-order bit (bit 31) is 0; equals `0x8000000` $\neq 0$ if that bit is 1.
- ▶ | Bitwise OR `I = I | 0x8000000`; makes bit 31 of I become (or stay) 1. It “sets” bit 31.
What does `I = I & 0x7FFFFFFF`; do? “clears” bit 31.
- ▶ ~ Bitwise NOT `~0x7FFFFFFF = ? 0x80000000`

Bit-shift operations

- ▶ `I << k` SHIFTS the bits LEFT `k` positions. `k` can be constant or variable.
- ▶ `I >> k` Guess what?

Addition in Decimal

$$\begin{array}{r} 36 \\ 87 \\ \hline \end{array}$$

Addition in Decimal

$$\begin{array}{r} 1 \quad \text{carries} \\ 36 \\ 87 \\ \hline 3 \end{array}$$

$$\begin{array}{r} 6 \\ 7 \\ \hline 13 \end{array}$$

Addition in Decimal

$$\begin{array}{r} 1 \ 1 \quad \text{carries} \\ 3 \ 6 \\ 8 \ 7 \\ \hline 2 \ 3 \end{array}$$

$$\begin{array}{r} 6 \\ 7 \\ \hline 1 \ 3 \end{array} \quad \begin{array}{r} 1 \\ 3 \\ 8 \\ \hline 1 \ 2 \end{array}$$

Addition in Decimal

$$\begin{array}{r} 1 \ 1 \quad \text{carries} \\ 3 \ 6 \\ 8 \ 7 \\ \hline 1 \ 2 \ 3 \end{array}$$

$$\begin{array}{r} 6 \\ 7 \\ \hline 1 \ 3 \end{array} \quad \begin{array}{r} 1 \\ 3 \\ 8 \\ \hline 1 \ 2 \end{array} \quad \begin{array}{r} 1 \\ 0 \\ 0 \\ \hline 0 \ 1 \end{array}$$

Addition in Binary: Single digits

$$\begin{array}{r} 0 \\ 0 \\ \hline 0 \end{array} \quad \begin{array}{r} 0 \\ 1 \\ \hline 1 \end{array} \quad \begin{array}{r} 1 \\ 0 \\ \hline 1 \end{array} \quad \begin{array}{r} 1 \\ 1 \\ \hline 1 \ 0 = 2 \end{array}$$

$$\begin{array}{r} 0 \\ 0 \\ 0 \\ \hline 0 \end{array} \quad \begin{array}{r} 0 \\ 0 \\ 1 \\ \hline 1 \end{array} \quad \begin{array}{r} 0 \\ 1 \\ 0 \\ \hline 1 \end{array} \quad \begin{array}{r} 1 \\ 0 \\ 0 \\ \hline 1 \end{array} \quad \begin{array}{r} 0 \\ 1 \\ 1 \\ \hline 1 \ 0 \end{array} \quad \begin{array}{r} 1 \\ 0 \\ 1 \\ \hline 1 \ 0 \end{array} \quad \begin{array}{r} 1 \\ 1 \\ 0 \\ \hline 1 \ 0 \end{array}$$

$$\begin{array}{r} 1 \\ 1 \\ 1 \\ \hline 1 \ 1 = 3 \end{array}$$

way!

Mathematics requires a binary computer to work this

Express these rules with LOGIC

0	0	1	1	<i>A</i>
0	1	0	1	<i>B</i>
<hr/>				<hr/>
0 0	0 1	0 1	1 0	<i>Carry</i> <i>Sum</i>

Explain logically how *A* and *B* determine *Sum* and *Carry*?

Express these rules with LOGIC

0	0	1	1	<i>A</i>
0	1	0	1	<i>B</i>
<hr/>				<hr/>
0 0	0 1	0 1	1 0	<i>Carry</i> <i>Sum</i>

Explain logically how *A* and *B* determine *Sum* and *Carry*?

Sum = 1 exactly when (*A* = 1 and *B* = 0) or (*A* = 0 and *B* = 1).

Express these rules with LOGIC

0	0	1	1	<i>A</i>
0	1	0	1	<i>B</i>
<hr/>				<hr/>
0 0	0 1	0 1	1 0	<i>Carry Sum</i>

Explain logically how *A* and *B* determine *Sum* and *Carry*?

Sum = 1 exactly when (*A* = 1 and *B* = 0) or (*A* = 0 and *B* = 1).

Carry = 1 exactly when (*A* = 1 and *B* = 1).

Express these rules with LOGIC

0	0	1	1	<i>A</i>
0	1	0	1	<i>B</i>
<hr/>				<i>Carry</i> <i>Sum</i>
0 0	0 1	0 1	1 0	

Explain logically how *A* and *B* determine *Sum* and *Carry*?

Sum = 1 exactly when (*A* = 1 and *B* = 0) or (*A* = 0 and *B* = 1).

Carry = 1 exactly when (*A* = 1 and *B* = 1).

The electronic *devices* called *logic gates* determine output signals like *Carry* from inputs like *A* and *B*. Typical computation time:

0.1 nanosecond = 10^{-10} second.

3 Basic Gates: AND, OR, and NOT

Truth Table for the AND gate (and AND Boolean Operation):

A	B	$A \text{ AND } B = A \& B$
0	0	0
0	1	0
1	0	0
1	1	1

Other useful gates and Boolean Operations:

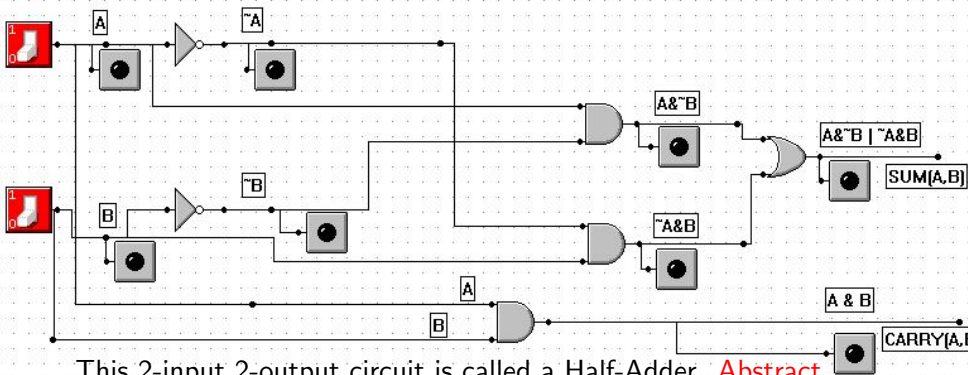
X	Y	$X \text{ OR } Y = X Y$
0	0	0
0	1	1
1	0	1
1	1	1

(our OR is *Inclusive*)

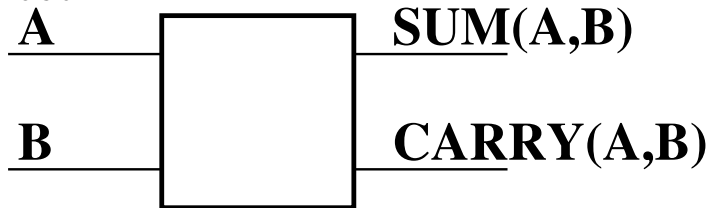
W	$\text{NOT } W = \sim W$
0	1
1	0

Some Boolean Expressions

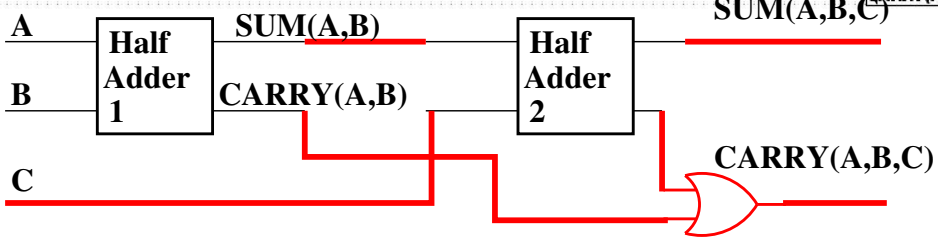
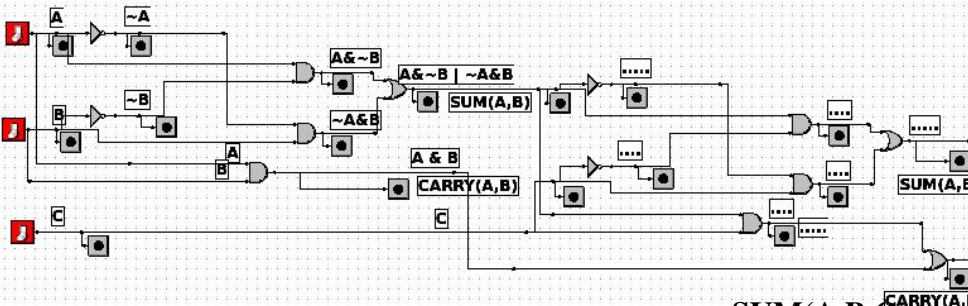
A	B	$\sim B$	$(A \& \sim B)$	$\sim A$	$(\sim A \& B)$	$(A \& \sim B) (\sim A \& B)$
0	0	1	0	1	0	0
0	1	0	0	1	1	1
1	0	1	1	0	0	1
1	1	0	0	0	0	0
Two independent Boolean Variables		Intermediate Values				This is <i>Sum</i> !
						Desired Result



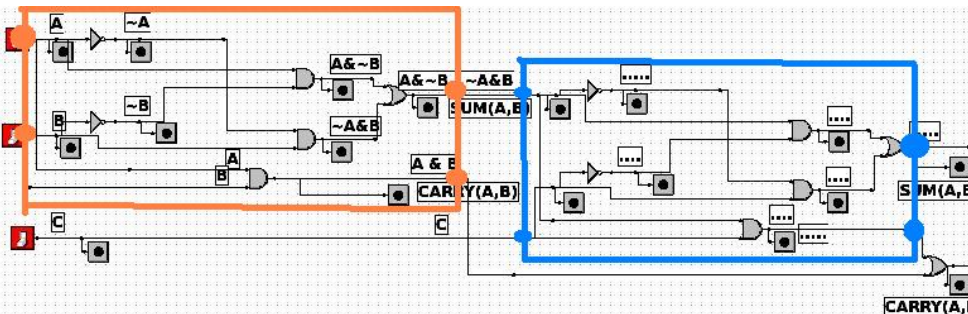
version:



Digital Electronic View



Two abstracted half-adder sub-circuits:



```

#include <stdio.h>
unsigned char N = 0; // 8 bits will be used
for( int i=0; i<1000; i++ )
{
    printf("%d ", N );//Print N as decimal integer.
    N = N + 1;
}

```

overflows or wraps around when N= 255.

0 1 ... 253 254 255 0 1 2 3 ... 255 0 ...
 252 253 254 255 0 1 2 3 ... 229 230 231

Something like this **absolutely must happen** since 8 bits can only distinguish $2^8 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 256$ different data values.

```

#include <stdio.h>
signed char N = 0; // 8 bits will be used
for( int i=0; i<1000; i++ )
{
    printf("%d ", N );//Print N as decimal integer.
    N = N + 1;
}

```

N declared signed this time

```

0 1 2 3 ... 126 127 -128 -127 -126 -125 ...
-3 -2 -1 0 1 2 ... 126 127 -128 -127 ...
-3 -2 -1 0 1 2 ... 126 127 -128 -127 ...
-3 -2 -1 0 1 2 ... 126 127 -128 -127 ...
... -29 -28 -27 -26 -25

```

The reason for the difference is subtle:

- ▶ Standard C `printf("%d", integer value);`
`%d` format ALWAYS uses SIGNED interpretation.
- ▶ C uses “usual argument conversions” on (8 bit) char types to (16 or 32) bit int since the `printf` function doesn't have declared argument types. `printf` is not type-safe!
- ▶ Bits of unsigned char variables are interpreted in unsigned binary for conversion to int.
- ▶ Bits of signed char variables are interpreted in signed binary for conversion to int.

Graphics Homework answer

31e2	0011	0001	1110	0010	00110001111100010		
b5ea	1011	0101	1110	1010	10110101111101010		
f7ca	1111	0111	1100	1010	11110111111001010		
39da	---\	0011	1001	1101	1010	---\	00111001111011010
9ddb	---/	1001	1101	1101	1011	---/	10011101111011011
cec0	1010	1110	1010	0000	1010111010100000		
ed5e	1100	1011	0101	1110	1100101101011110		
210e	0010	0001	0000	1110	0010000100001110		
ffff	1111	1111	1111	1111	1111111111111111		

This is correct! The first 16-bit number 31 is 00110001 The first 2 0's are in the $2^7 = 128$ and $2^6 = 64$ places.

The left-hand end, where we begin reading it, has the BIGGER valued binary digits. That is normal: 2007 means “Two THOUSAND (plus only) SEVEN”,

Wrong Bit-order Endianness!

But, if we write each 8-bit binary value so the LITTLE value digits are at the left-hand end, we get: $31_{\text{hex}} = 1000\ 1100 = 10001100 = 1 \cdot 2^0 + 0 \cdot 2^1 + \dots + 1 \cdot 2^8 + 1 \cdot 2^{16} + 0 + 0$
Now, if we write the bits for each 8-bit number in this fashion:

31	e2	10001100	01000111	1000110001000111
b5	ea	10101101	01010111	1010110101010111
f7	ca	11101111	01010011	1110111101010011
39	da	---\ 10011100	01011011	---\ 1001110001011011
9d	db	---/ 10111001	11011011	---/ 1011100111011011
ce	c0	01110011	00000011	0111001100000011
ed	5e	10110111	01111010	1011011101111010
21	0e	10000100	01110000	1000010001110000
ff	ff	11111111	11111111	1111111111111111

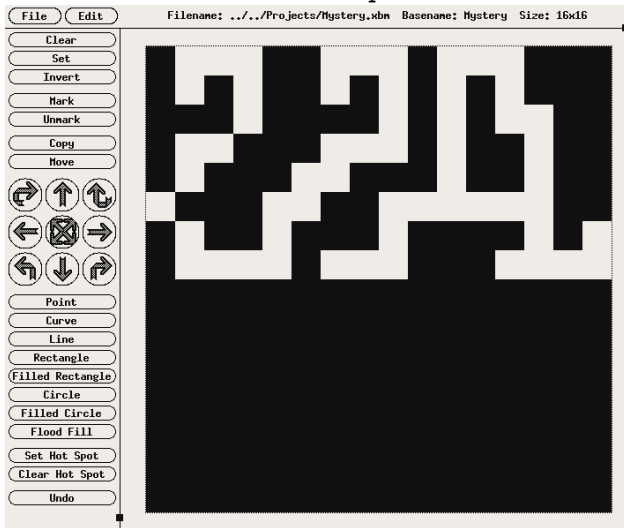
s/0/ / and s/1/M/ result

```
M   MM   M   MMM
M M MM M M M MMM
MMM MMMM M M  MM
M   MMM   M MM MM
M MMM   MMM MM MM
   MMM  MM      MM
M MM  MMM MMMM M
M     M   MMM
MMMMMMMMMMMMMMMM
```

s/0/ / and s/1/M/ result

And how I made it with bitmap:

```
M   MM  M   MMM
M M MM M M M MMM
MMM MMMM M M  MM
M   MMM  M MM MM
M MMM  MMM MM MM
   MMM  MM      MM
M MM MMM MMMM M
M   M   MMM
MMMMMMMMMMMMMMMM
```



Output of bitmap: an ASCII .xbm file of C-code!

```
#define Mystery_width 16
#define Mystery_height 16
static unsigned char Mystery_bits[] = {
    0x31, 0xe2, 0xb5, 0xea, 0xf7, 0xca, 0x39, 0xda, 0x9d, 0x
    0xed, 0x5e, 0x21, 0x0e, 0xff, 0xff, 0xff, 0xff, 0xff, 0x
    0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff};
```

which I edited using emacs to get

```
    0x31, 0xe2, 0xb5, 0xea, 0xf7, 0xca, 0x39, 0xda, 0x9d, 0x
edited into
31e2      ??????????????????
b5ea      ??????????????????
etc
```

... we looked at the data in .xpm file

1. Surprise: .xpm format: C array initializer for an array of bytes, where the bits in each byte (like 0x31 and 0xe2), *taken in LITTLE-ENDIAN BIT ORDER*, represent a left-to-right sub-row of 8 pixels.
2. HAPPY FACT: ALL CPUs do their binary arithmetic on various length words in BIG-ENDIAN BIT ORDER. (Familiar big-endian digit order example: 2007 is in our millennium).
3. (but not all graphics standards!)
4. SAD FACT: MULTI-BYTE numeric data (from memory, files, network apps) is interpreted with different BYTE ORDER (“ENDIAN-NESS”) by DIFFERENT CPU’s.
Big Endian: SPARC from SUN (in itsunix), MIPS (Nintendos)
Little Endian: IA32 (Pentium & clone PC’s)
5. Earth’s current Internet is Big Endian.