Data Lab – Let’s Code

```
/*
 * bitParity - returns 1 if x contains an odd number of 0's
 *   Examples: bitParity(5) = 0, bitParity(7) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 20
 *   Rating: 4
 */
```

```
int bitParity(int x) {
  return 2;
}
```

Bit Operations: Review

- Book Version

~	
0	1
1	0

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

- Truth Tables

P	Q	$P \wedge Q$	$P \vee Q$	$P \uparrow Q$	$P \downarrow Q$	$P \nleftrightarrow Q$	$P \leftrightarrow Q$
T	T	T	T	F	F	F	T
T	F	F	T	T	F	T	F
F	T	F	T	T	F	T	F
F	F	F	F	T	T	F	T
Name (function)		AND (conjunction)	OR (disjunction)	NAND (non-conjunction)	NOR (non-disjunction)	XOR (non-equivalence)	XNOR (equivalence)
where T *means* **true** *and* F *means* **false**							

Strings

- ASCII table (man ascii)

ASCII Table

0 0x00 0000 0000 NUL	1 0x01 0000 0001 SOH	2 0x02 0000 0010 STX	3 0x03 0000 0011 ETX	4 0x04 0000 0100 EOT	5 0x05 0000 0101 ENQ	6 0x06 0000 0110 ACK	7 0x07 0000 0111 BEL	8 0x08 0000 1000 BS	9 0x09 0000 1001 HT	10 0x0a 0000 1010 LF \n	11 0x0b 0000 1011 VT	12 0x0c 0000 1100 FF	13 0x0d 0000 1101 CR \r	14 0x0e 0000 1110 SO	15 0x0f 0000 1111 SI
16 0x10 0001 0000 DLE	17 0x11 0001 0001 DC1	18 0x12 0001 0010 DC2	19 0x13 0001 0011 DC3	20 0x14 0001 0100 DC4	21 0x15 0001 0101 NAK	22 0x16 0001 0110 SYN	23 0x17 0001 0111 ETB	24 0x18 0001 1000 CAN	25 0x19 0001 1001 EM	26 0x1a 0001 1010 SUB	27 0x1b 0001 1011 ESC	28 0x1c 0001 1100 FS	29 0x1d 0001 1101 GS	30 0x1e 0001 1110 RS	31 0x1f 0001 1111 US
32 0x20 0010 0000 SPACE	33 0x21 0010 0001 !	34 0x22 0010 0010 "	35 0x23 0010 0011 #	36 0x24 0010 0100 $	37 0x25 0010 0101 %	38 0x26 0010 0110 &	39 0x27 0010 0111 '	40 0x28 0010 1000 (	41 0x29 0010 1001)	42 0x2a 0010 1010 *	43 0x2b 0010 1011 +	44 0x2c 0010 1100 ,	45 0x2d 0010 1101 -	46 0x2e 0010 1110 .	47 0x2f 0010 1111 /
48 0x30 0011 0000 0	49 0x31 0011 0001 1	50 0x32 0011 0010 2	51 0x33 0011 0011 3	52 0x34 0011 0100 4	53 0x35 0011 0101 5	54 0x36 0011 0110 6	55 0x37 0011 0111 7	56 0x38 0011 1000 8	57 0x39 0011 1001 9	58 0x3a 0011 1010 :	59 0x3b 0011 1011 ;	60 0x3c 0011 1100 <	61 0x3d 0011 1101 =	62 0x3e 0011 1110 >	63 0x3f 0011 1111 ?
64 0x40 0100 0000 @	65 0x41 0100 0001 A	66 0x42 0100 0010 B	67 0x43 0100 0011 C	68 0x44 0100 0100 D	69 0x45 0100 0101 E	70 0x46 0100 0110 F	71 0x47 0100 0111 G	72 0x48 0100 1000 H	73 0x49 0100 1001 I	74 0x4a 0100 1010 J	75 0x4b 0100 1011 K	76 0x4c 0100 1100 L	77 0x4d 0100 1101 M	78 0x4e 0100 1110 N	79 0x4f 0100 1111 O
80 0x50 0101 0000 P	81 0x51 0101 0001 Q	82 0x52 0101 0010 R	83 0x53 0101 0011 S	84 0x54 0101 0100 T	85 0x55 0101 0101 U	86 0x56 0101 0110 V	87 0x57 0101 0111 W	88 0x58 0101 1000 X	89 0x59 0101 1001 Y	90 0x5a 0101 1010 Z	91 0x5b 0101 1011 [	92 0x5c 0101 1100 \	93 0x5d 0101 1101]	94 0x5e 0101 1110 ^	95 0x5f 0101 1111 _
96 0x60 0110 0000 `	97 0x61 0110 0001 a	98 0x62 0110 0010 b	99 0x63 0110 0011 c	100 0x64 0110 0100 d	101 0x65 0110 0101 e	102 0x66 0110 0110 f	103 0x67 0110 0111 g	104 0x68 0110 1000 h	105 0x69 0110 1001 i	106 0x6a 0110 1010 j	107 0x6b 0110 1011 k	108 0x6c 0110 1100 l	109 0x6d 0110 1101 m	110 0x6e 0110 1110 n	111 0x6f 0110 1111 o
112 0x70 0111 0000 p	113 0x71 0111 0001 q	114 0x72 0111 0010 r	115 0x73 0111 0011 s	116 0x74 0111 0100 t	117 0x75 0111 0101 u	118 0x76 0111 0110 v	119 0x77 0111 0111 w	120 0x78 0111 1000 x	121 0x79 0111 1001 y	122 0x7a 0111 1010 z	123 0x7b 0111 1011 {	124 0x7c 0111 1100 |	125 0x7d 0111 1101 }	126 0x7e 0111 1110 ~	127 0x7f 0111 1111 DEL

Strings: char *

- char *mystring = “Hello World”;
- char is a special kind of short
- Add, subtract, multiply
- Don’t.

Strings: char *

- char *mystring = “Hello World”;
- char is a special kind of short
- Add, subtract, multiply
- Don’t.

Logical Operators

- ||
  - Tests truth of “OR”, returns 0/1
- &&
  - Tests truth of “AND”, returns 0/1
- !
  - Tests for FALSE, returns 0/1

Logical Operators: Continued

- Logical OR of two integers
  - 0x1 | 0x10 == ?
  - 0x1 || 0x10 == ?

```c
#include <stdio.h>

int main() {
        int a = 0x1;
        int b = 0x10;
        int c;

        c = (a || b);
        printf("|| %0x\n", c);
        c = (a | b);
        printf("| %0x\n", c);
        return 0;
}
```

Logical Operators: Continued

- Logical AND of two integers
  - 0x1 & 0x10 == ?
  - 0x1 && 0x10 == ?

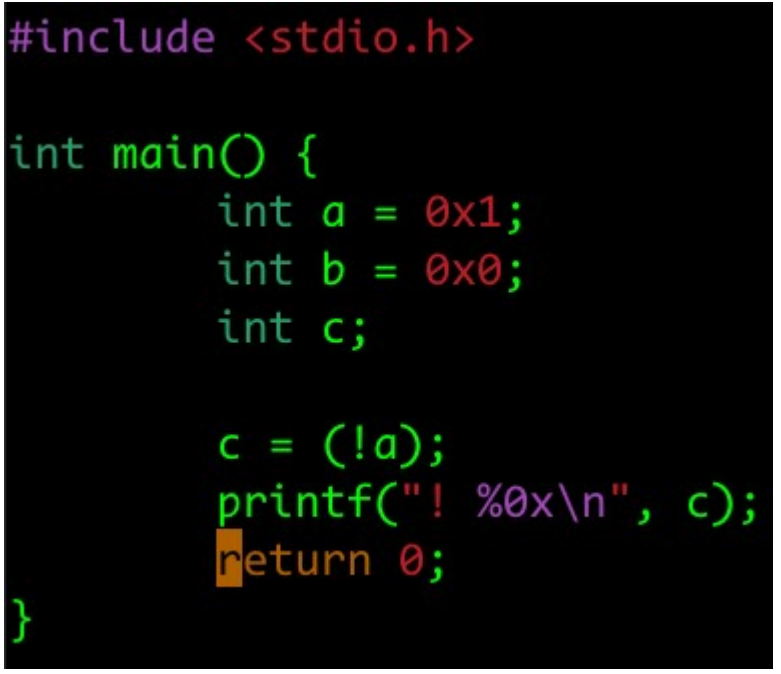
```c
#include <stdio.h>

int main() {
        int a = 0x1;
        int b = 0x10;
        int c;

        c = (a && b);
        printf("&& %0x\n", c);
        c = (a & b);
        printf("& %0x\n", c);
        return 0;
}
```

Logical Operators: Continued

- Logical NOT
  - ! 0x1 == ?
  - ! 0x0 == ?
  - ! 0x100 ==



```c
#include <stdio.h>

int main() {
        int a = 0x1;
        int b = 0x0;
        int c;

        c = (!a);
        printf("! %0x\n", c);
        return 0;
}
```

Bit Shifts

- >>
  - Shift “right”
  - (make smaller)
- <<
  - Shift “left”
  - (make larger)

Operation	Value 1	Value 2
Argument x	[01100011]	[10010101]
x << 4	[0011*0000*]	[0101*0000*]
x >> 4 (logical)	[*0000*0110]	[*0000*1001]
x >> 4 (arithmetic)	[*0000*0110]	[*1111*1001]

Bit Shifts: Continued

- Padding Zeroes
  - Shifting right fills zeroes in MSb
  - Shifting left fills zeroes in LSb
- Truncating Values
  - Left shift drops MS bit
  - Right shift drops LS bit
- NO ERROR CHECKING: GOOD LUCK HAVE FUN

Integer Representation

Symbol	Type	Meaning	Page
$B2T_w$	Function	Binary to two's complement	100
$B2U_w$	Function	Binary to unsigned	98
$U2B_w$	Function	Unsigned to binary	100
$U2T_w$	Function	Unsigned to two's complement	107
$T2B_w$	Function	Two's complement to binary	101
$T2U_w$	Function	Two's complement to unsigned	107
$TMin_w$	Constant	Minimum two's-complement value	101
$TMax_w$	Constant	Maximum two's-complement value	101
$UMax_w$	Constant	Maximum unsigned value	99
$+^t_w$	Operation	Two's-complement addition	126
$+^u_w$	Operation	Unsigned addition	121
$*^t_w$	Operation	Two's-complement multiplication	133
$*^u_w$	Operation	Unsigned multiplication	132
$-^t_w$	Operation	Two's-complement negation	131
$-^u_w$	Operation	Unsigned negation	125

C Types Maximum Ranges

- Again, CHAR is special
- 32 BIT Definitions

C data type	Minimum	Maximum
[signed] char	−128	127
unsigned char	0	255
short	−32,768	32,767
unsigned short	0	65,535
int	−2,147,483,648	2,147,483,647
unsigned	0	4,294,967,295
long	−2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,295
int32_t	−2,147,483,648	2,147,483,647
uint32_t	0	4,294,967,295
int64_t	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	0	18,446,744,073,709,551,615

Figure 2.9 **Typical ranges for C integral data types for 32-bit programs.**

C Types Maximum Ranges

- Again, CHAR is special
- 64 BIT Definitions

C data type	Minimum	Maximum
[signed] char	−128	127
unsigned char	0	255
short	−32,768	32,767
unsigned short	0	65,535
int	−2,147,483,648	2,147,483,647
unsigned	0	4,294,967,295
long	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long	0	18,446,744,073,709,551,615
int32_t	−2,147,483,648	2,147,483,647
uint32_t	0	4,294,967,295
int64_t	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	0	18,446,744,073,709,551,615

Figure 2.10 **Typical ranges for C integral data types for 64-bit programs.**

Unsigned Storage

- Easy Mode

PRINCIPLE: Definition of unsigned encoding

For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$:

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i \tag{2.1}$$

In this equation, the notation $\doteq$ means that the left-hand side is defined to be equal to the right-hand side. The function $B2U_w$ maps strings of zeros and ones of length w to nonnegative integers. As examples, Figure 2.12 shows the mapping, given by $B2U$, from bit vectors to integers for the following cases:

$$\begin{aligned}
B2U_4([0001]) &= 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 &&= 0+0+0+1 &&= 1\\
B2U_4([0101]) &= 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 &&= 0+4+0+1 &&= 5\\
B2U_4([1011]) &= 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 &&= 8+0+2+1 &&= 11\\
B2U_4([1111]) &= 1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 &&= 8+4+2+1 &&= 15
\end{aligned} \tag{2.2}$$

Two's Compliment

- Reserve MSbit for signedness

PRINCIPLE: Definition of two's-complement encoding

For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$:

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \tag{2.3}$$

The most significant bit x_{w-1} is also called the *sign bit*. Its "weight" is -2^{w-1}, the negation of its weight in an unsigned representation. When the sign bit is set to 1, the represented value is negative, and when set to 0, the value is nonnegative. As examples, Figure 2.13 shows the mapping, given by $B2T$, from bit vectors to integers for the following cases:

$$\begin{array}{lcllcll}
B2T_4([0001]) & = & -0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 & = & 0+0+0+1 & = & 1 \\
B2T_4([0101]) & = & -0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 & = & 0+4+0+1 & = & 5 \\
B2T_4([1011]) & = & -1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 & = & -8+0+2+1 & = & -5 \\
B2T_4([1111]) & = & -1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 & = & -8+4+2+1 & = & -1
\end{array} \tag{2.4}$$