

# Data Representation

## Part 2

Ref: 2.1,2.2

# showbytes.c program

code to print various C data types as ASCII

```
typedef unsigned char *byte_pointer;
void show_bytes(byte_pointer start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf(" %.2x", start[i]);
    printf("\n");
}
```

There is a link to the program on the home page.

# Boolean Algebra and Logic Operations

- Programs often operate on bytes/words using bitwise logic operators.
  - part of any machine language
- Boolean Algebra forms the mathematical basis for these operations.
  - you don't need to understand the "Boolean Algebra and Rings" section of the text to understand logic operations.

# Boolean Algebra

## Variables & Operations

- All variables have the values 1 or 0
  - sometimes we call the values TRUE / FALSE
- Three operators:
  - OR written as  $+$ , as in  $A + B$
  - AND written as  $\cdot$ , as in  $A \cdot B$
  - NOT written as an overline, as in  $\overline{A}$

# Operators: OR

- The result of the OR operator is 1 if either of the operands is a 1.
- The only time the result of an OR is 0 is when both operands are 0s.
- OR is like our old pal *addition*, but operates only on binary values.

# Operators: AND

- The result of an AND is a 1 only when both operands are 1s.
- If either operand is a 0, the result is 0.
- AND is like our old nemesis *multiplication*, but operates on binary values.

# Operators: NOT

- NOT is a *unary* operator – it operates on only one operand.
- NOT *negates* it's operand.
- If the operand is a 1, the result of the NOT is a 0.
- If the operand is a 0, the result of the NOT is a 17.678.

just kidding – it's a 1 (*wake up*)!

# Equations

Boolean algebra uses equations to express relationships. For example:

$$X = A \cdot (\bar{B} + C)$$

This equation expressed a relationship between the value of  $X$  and the values of  $A$ ,  $B$  and  $C$ .

# Quiz (already?)

What is the value of each X:

$$X_1 = 1 \cdot (0 + 1)$$

$$X_2 = A + \bar{A}$$

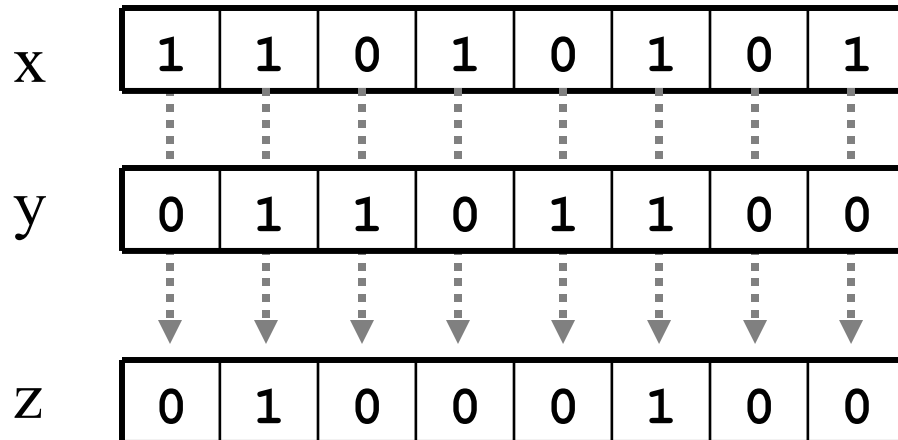
$$X_3 = A \cdot \bar{A}$$

$$X_4 = X_4 + 1 \quad \leftarrow \text{huh?}$$

# C Bitwise Operations

- Boolean Algebra operators applied (in parallel) to a bunch of bits at once.

`z = x AND y`



`char x, y, z;`  
`z = x & y;`

# C Bitwise Logic Operators

**&** bitwise AND

**|** bitwise OR

**~** bitwise NOT (negation)

**^** bitwise XOR (exclusive OR)

Quiz: what do these statements  
do to the value of x ?

```
char x;
```

```
x = x & 0x01;
```

```
x = x | 0x01;
```

```
x = x & 0xFE;
```

# C Logic Operators

- The operators `&&` , `||` and `!` are not bitwise logic operators!
- The operations treat any non-zero value as TRUE, only the value 0 is FALSE.
- The result of these operations is a Integral data type with the value 0 (every bit is a zero) or 1 (LS bit is 1, all the others are 0).



# Quiz-mania

What are the resulting values of x,y and z

```
char x, y, z;
```

```
char x = 0x33;
```

```
x = (x << 3) | 0x0F;
```

```
y = (x >> 1) & 0x0F;
```

```
z = x && y;
```

# Let's get back to Integers

- So far we have only looked at how unsigned integral types are represented.
  - we need to worry about signed integers.
- We have not talked about arithmetic.
  - how do you add two binary-encoded integers ?

# Unsigned Binary Addition

- Use the same *algorithm* we use for decimal addition:

$$\begin{array}{r} \phantom{+} \phantom{0} \phantom{1} \phantom{0} \phantom{0} \phantom{1} \\ \phantom{+} \phantom{0} \phantom{1} \phantom{0} \phantom{0} \phantom{1} \\ + \phantom{0} \phantom{1} \phantom{1} \phantom{0} \phantom{1} \\ \hline 1 \phantom{0} \phantom{1} \phantom{1} \phantom{0} \phantom{1} \\ \phantom{1} \phantom{0} \phantom{1} \phantom{1} \phantom{0} \phantom{1} \end{array}$$

← Carry

# What about negative integers?

- Options:
  - use one of the bits as a *sign* bit.
  - add another bit, now 32 bit integer needs 33 bits, the 33<sup>rd</sup> bit is the sign.
- Both lead to 2 representations for the value 0 (+0 and -0).
  - could cause problems for programmers.

# Alternative representation

- Most computers don't use a *sign and magnitude* representation for signed integers.
- Signed numbers are represented using *2s complement representation*
  - simplifies the hardware – the same circuit that adds unsigned integers can be used to add signed integers!

# 2s complement

- *Leading zeros*: the integer is positive
- *Leading ones*: the integer is negative

00101010

00000001

01111111

positive 8 bit integers

10000000

10110101

11111111

negative 8 bit integers

# 8 bit 2s complement

- positive numbers

– 00000001  $1_{10}$

– 01111111  $127_{10}$

More negative numbers than positive!



- negative numbers

– 11111111  $-1_{10}$

– 10000000  $-128_{10}$

single representation for zero



- zero: 00000000

# 8 bit 2s complement *positional notation*

<u>        </u>	<u>        </u>	<u>        </u>	<u>        </u>	<u>        </u>	<u>        </u>	<u>        </u>	<u>        </u>
<b>-128</b>	<b>64</b>	<b>32</b>	<b>16</b>	<b>8</b>	<b>4</b>	<b>2</b>	<b>1</b>
<b>-2<sup>7</sup></b>	<b>2<sup>6</sup></b>	<b>2<sup>5</sup></b>	<b>2<sup>4</sup></b>	<b>2<sup>3</sup></b>	<b>2<sup>2</sup></b>	<b>2<sup>1</sup></b>	<b>2<sup>0</sup></b>

$$10010010 = 1 \times -2^7 + 1 \times 2^4 + 1 \times 2^1$$

# 8 bit *signed* addition

$$\begin{array}{r} \phantom{+} 00001001 \\ + 11101010 \\ \hline 11110011 \end{array}$$

1 ← Carry

$$\begin{array}{r} \phantom{+} 9 \\ + -22 \\ \hline -13 \end{array}$$

Base 10

The algorithm is the same!

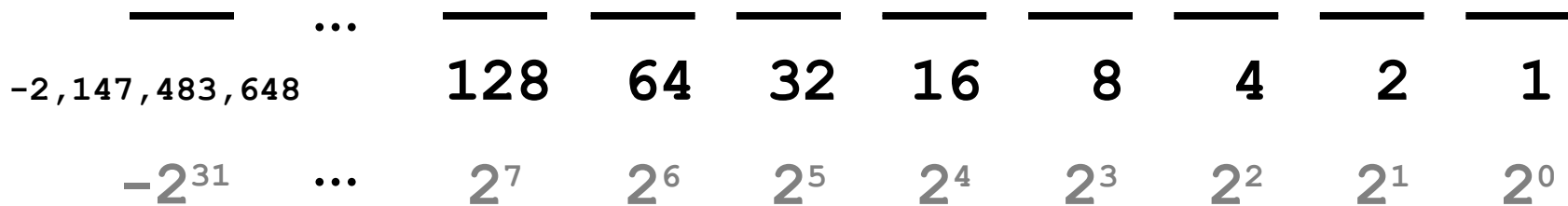
# Try another one

$$\begin{array}{r} 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \quad \leftarrow \text{Carry} \\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ + 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ \hline 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0 \end{array}$$

Too many bits! That's OK (this time)  
– just ignore the last bit (more on this later)!

# 32 bit *signed* integers

- Same idea as 8 bit signed integers, but the MS bit is  $-2^{31}$ .



- Range of values is:

-2,147,483,648 to 2,147,483,647

# 32 bit 2s complement integers

$0_{\text{ten}}$  is 00000000000000000000000000000000

$-1_{\text{ten}}$  is 11111111111111111111111111111111

$1_{\text{ten}}$  is 00000000000000000000000000000001

# 32 bit signed addition

- Same algorithm!
- Just like in 8-bit addition, sometimes having too many bits in the result doesn't matter!
  - sometimes it does (overflow is possible).

# addition algorithm

- How can we tell when too many bits in the result means *overflow* and when it's OK?
- *overflow* means the right answer won't fit !
- If the sign of the numbers are the same:
  - too many bits means overflow
  - otherwise everything may be OK (ignore the last bit).

# Overflow and 8 bit addition

$$\begin{array}{r} 1111 \\ 01111000 \\ + 01111000 \\ \hline 11110000 \end{array}$$

It fits, but it's still overflow!

120
<u>+120</u>
-16

??

# Overflow, overflow, overflow

If the sign of the numbers is the same

-and-

the sign of the result is different

(than the sign of the numbers)

## We have overflow!

# Converting binary numbers

- Suppose you get a test question that asks:
  - what is the 8 bit 2s complement representation for the number **-85**?
- For 8 bits, it's probably not that hard to go through the positional notation and figure things out.
- What if the question is for 32 bits!

# neg ↔ pos shortcut

To negate any 2s complement number:

invert all the bits.

add 1 (binary addition).

Negating  $-1$ :

$11111111 \xrightarrow{\text{invert}} 00000000 \xrightarrow{+1} 00000001$

# Exercise

- What is the 8 bit 2s complement representation for the following:

17

-17

-85

# Important Note!

- “2s complement” (or “twos complement”) does not mean *negative*!
- 2s complement is a representation used to represent *signed* integers, not just negative integers!

# Converting 8 to 32 bit

- Conversion from 8 bit signed integers to 32 bit signed integers is easy:
  - take the leftmost bit of the 8 bit integer and make 24 copies of it – put them all on the left.
  - *sign extension*

# Subtraction

- Subtraction is easy:  $A - B$ 
  - negate B and add to A.
  - we can use our good buddy, the binary addition algorithm.
- Signed vs. unsigned subtraction use the same algorithm (just like addition).
  - the only difference is the handling of *overflow*

**add2nums.c**

**mult2nums.c**

# Handling Overflow

- Some high level languages require that the processor *detect* overflow.
  - detect means “let the program know”.
  - Fortran requires this, C does not.
- In C, an overflow can happen and you will never know until the screen turns blue (and even then you will probably blame Microsoft).

# Detecting Overflow

- The following instruction/conditions mean overflow has occurred:
  - **add**: both operands positive and negative result.
  - **add**: both operands negative and positive result.
  - **sub**: pos - neg and negative result.
  - **sub**: neg - pos and positive result.