

# CSI333 Lecture 5

# 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 `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.  
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:

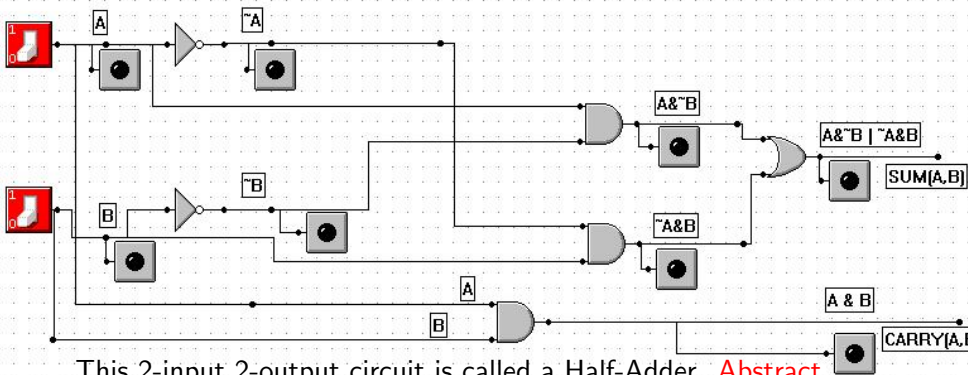
X	Y	$X \text{ OR } Y = X   Y$
0	0	0
0	1	1
1	0	1
1	1	1

(our OR is *Inclusive*)

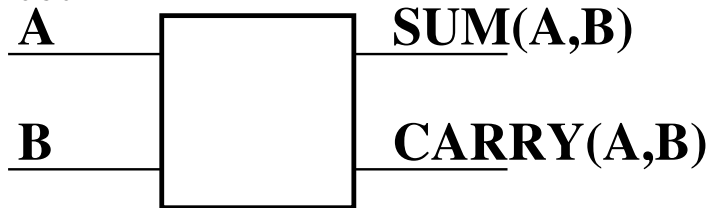
W	$\text{NOT } W = \sim W$
0	1
1	0

## Some Boolean Expressions

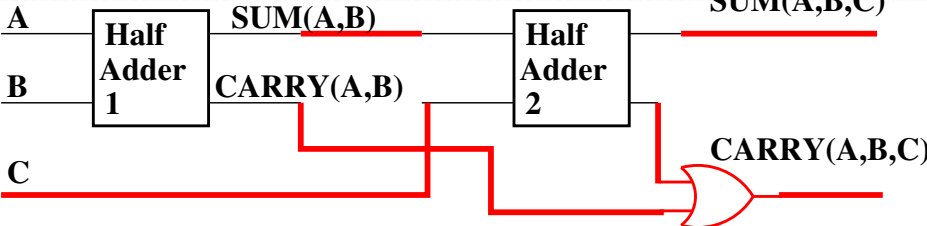
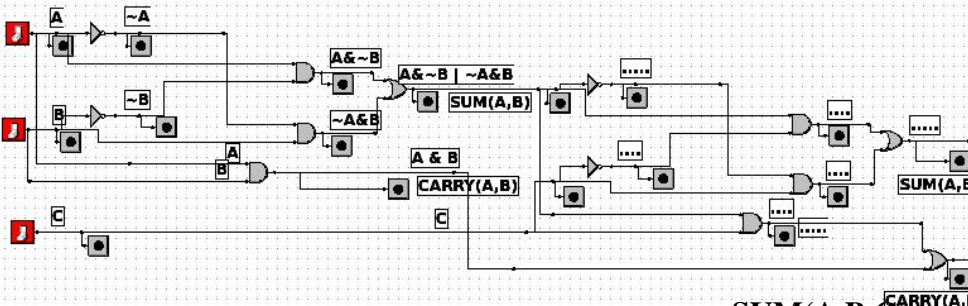
$A$	$B$	$\sim B$	$(A \& \sim B)$	$\sim A$	$(\sim A \& B)$	$(A \& \sim B)   (\sim A \& B)$
0	0	1	0	1	0	0
0	1	0	0	1	1	1
1	0	1	1	0	0	1
1	1	0	0	0	0	0
Two independent Boolean Variables		Intermediate Values				This is <i>Sum</i> !
						Desired Result



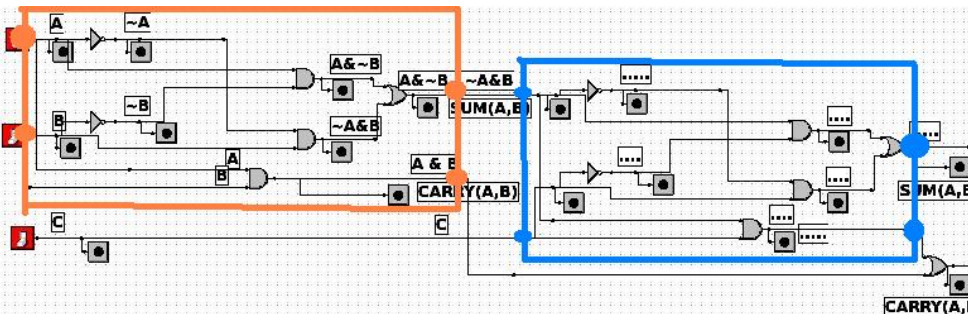
version:



# 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.

## Good Bye for Now; Remainder of Lecture

1. Demonstration of Multimedia Logic software on half and full adder circuits.
2. Presentation of Patt and Patel's Chapter 4 slides on Computer Architecture and intro. to the LC-3 architecture. (This intro. done with the Windows LC-3 simulator was the subject of Lab 02 last week.)
3. "Be a Computer Architect" group problem solving session.