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

Bitwise operations

- & Bitwise AND int I; 0x80000000 & I equals 0 if the top-order bit (bit 31) is 0; equals 0x80000000 ≠ 0 if that bit is 1.

- | Bitwise OR I = I | 0x80000000; 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 ≠ 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 & 0x7FFFFFFFF; 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 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? “clears” bit 31.
- ~ Bitwise NOT \(~0x7FFFFFFF = ?\)
How to access individual bits in C/C++/Java

Bitwise operations

- & Bitwise AND int I; 0x80000000 & I equals 0 if the top-order bit (bit 31) is 0; equals 0x80000000 ≠ 0 if that bit is 1.

- | Bitwise OR I = I | 0x80000000; makes bit 31 of I become (or stay) 1. It “sets” bit 31. What does I = I & 0xFFFFFFFF; 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

\[
\begin{array}{c}
36 \\
87 \\
\hline \\
137
\end{array}
\]
Addition in Decimal

1 carries
3 6
8 7

\[ \underline{3} \]

6
7

\[ \underline{1 3} \]
Addition in Decimal

\[
\begin{align*}
1 & \quad 1 & \text{carries} \\
3 & \quad 6 \\
8 & \quad 7 \\
\hline
2 & \quad 3
\end{align*}
\]

\[
\begin{align*}
6 & \\
7 & \\
\hline
1 & 3
\end{align*}
\]

\[
\begin{align*}
1 & \quad 3 \\
8 & \\
\hline
1 & 2
\end{align*}
\]
Addition in Decimal

\[
\begin{array}{c}
1 & 1 & \text{carries} \\
3 & 6 \\
8 & 7 \\
\hline
1 & 2 & 3
\end{array}
\]

\[
\begin{array}{c}
6 \\
7 \\
\hline
1 & 3 & 1 & 1
\end{array}
\]

\[
\begin{array}{c}
1 \\
3 \\
8 \\
\hline
1 & 2 & 0 & 0 & 0 & 1
\end{array}
\]
Addition in Binary: Single digits

\[
\begin{array}{cccccc}
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 \\
\hline
0 & 1 & 1 & 0 \\
\end{array}
\]

\[10 = 2\]

\[
\begin{array}{cccccc}
0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 \\
\hline
0 & 1 & 1 & 1 & 1 & 0 \\
\end{array}
\]

\[111 = 3\]

Mathematics requires a binary computer to work this way!
Express these rules with LOGIC

<table>
<thead>
<tr>
<th>A</th>
<th>Carry</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Explain logically how $A$ and $B$ determine $Sum$ and $Carry$?
Express these rules with LOGIC

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Carry Sum</td>
</tr>
</tbody>
</table>

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

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

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.
Express these rules with LOGIC

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Carry</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>Carry</td>
<td>Sum</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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):**

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$A$ AND $B = A &amp; B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Carry when adding $A+B$ in binary

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$X$ OR $Y = X \mid Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(our OR is *Inclusive*)

Other useful gates and Boolean Operations:

<table>
<thead>
<tr>
<th>$W$</th>
<th>NOT $W = \sim W$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
### Some Boolean Expressions

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$\sim B$</th>
<th>$(A &amp; \sim B)$</th>
<th>$\sim A$</th>
<th>$(\sim A &amp; B)$</th>
<th>$(A &amp; B) \mid (\sim A &amp; B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Two independent Boolean Variables</th>
<th>Intermediate Values</th>
<th>Desired Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is $Sum$!</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This 2-input 2-output circuit is called a Half-Adder. Abstract version:

\[ \text{SUM}(A,B) \]

\[ \text{CARRY}(A,B) \]
Two abstracted half-adder sub-circuits:
```c
#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 $2^8 = 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 256$ 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);` 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.)