Arithmetic Operations

- Addition
- Subtraction
- Multiplication
- Division

Each of these operations on the integer representations:
- Unsigned
- Two's complement
Addition

One bit of binary addition

carry out

+ b

sum bit

a

carry in
<table>
<thead>
<tr>
<th>Carry In</th>
<th>a</th>
<th>b</th>
<th>Carry Out</th>
<th>Sum Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<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>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Addition

- Unsigned and 2's complement use the same addition algorithm
- Due to the fixed precision, throw away the carry out from the msb

```
  00010111
+  10010010
___________
  10101101
```
Two's Complement Addition

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

\[
\begin{array}{cccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
+ & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
\hline
& & & & & & & & & 1 \\
\end{array}
\]
Overflow

The condition in which the result of an arithmetic operation cannot fit into the fixed number of bits available.

For example:

+8 cannot fit into a 3-bit, unsigned representation. It needs 4 bits: 1000
Overflow Detection

- Most architectures have hardware that detects when overflow has occurred (for arithmetic operations).
- The detection algorithms are simple.
Unsigned Overflow Detection

6-bit examples:

0 0 1 1 1 1
+ 0 0 1 1 1 1
_____________
1 0 0 0 0 0

1 1 1 1 1 1
+ 0 0 0 0 0 1
_____________
1 0 0 0 0 0

Carry out from msbs is overflow in unsigned
Unsigned Overflow Detection

6-bit examples:

\[
\begin{align*}
0 & \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \\
+ \quad 0 & \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \\
\hline
0 & \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0
\end{align*}
\]

0  No Overflow

\[
\begin{align*}
1 & \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \\
+ \quad 0 & \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \\
\hline
0 & \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 
\end{align*}
\]

1  Overflow!

\[
\begin{align*}
1 & \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \\
+ \quad 1 & \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \\
\hline
1 & \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 
\end{align*}
\]

1  Overflow!

Carry out from msbs is overflow in unsigned
Two’s Complement Overflow Detection

When adding 2 numbers of like sign

+ to +
- to -

and the sign of the result is different!

+ +
- -

Overflow! Overflow!
Addition

Overflow detection: 2's complement
6-bit examples

111111 ( )
+ 111111 ( )
_________
( )

100000 ( )
+ 011111 ( )
_________
( )

011111 ( )
+ 011111 ( )
_________
( )
Subtraction

basic algorithm is like decimal...

\[
\begin{align*}
0 - 0 &= 0 \\
1 - 0 &= 1 \\
1 - 1 &= 0 \\
0 - 1 &= \_ \text{ BORROW!}
\end{align*}
\]

\[
\begin{align*}
111000 \\
- \ 010110 \\
\hline
101100
\end{align*}
\]
Subtraction

For two’s complement representation

- The implementation redefines the operation:
  \[ a - b \text{ becomes } a + (-b) \]

- This is a 2-step algorithm:
  1. “take the two’s complement of \( b \)”
     (common phrasing for: find the additive inverse of \( b \))
  2. do addition
Subtraction

6-bit, 2’s complement examples

\[ \begin{align*}
001111 & \quad ( ) \\
- \quad 111100 & \quad ( ) \\
\hline
\end{align*} \]

\[ \begin{align*}
000010 & \quad ( ) \\
- \quad 011100 & \quad ( ) \\
\hline
\end{align*} \]
Subtraction

Overflow detection: 2’s complement

If the addition causes overflow, so does the subtraction!

\[
\begin{array}{c}
100000 ( ) \\
- 000010 ( ) \\
\end{array}
\]
Multiplication

0 \times 0 = 0
0 \times 1 = 0
1 \times 0 = 0
1 \times 1 = 1

- Same algorithm as decimal...
- There is a precision problem

\begin{array}{c c c c}
\text{n bits} & \times & \text{n bits} \\
\hline \\
\text{n + n bits may be needed}
\end{array}
In HW, space is always designated for a larger precision product.

\[
\begin{array}{c}
32 \text{ bits} \\
\times
\end{array}
\begin{array}{c}
32 \text{ bits} \\
\hline
64 \text{ bits}
\end{array}
\]
Unsigned Multiplication

01111

* 01101

01101
Unsigned Multiplication

\[ \begin{array}{c}
11111 \\
* \\
\hline
11111 \\
\end{array} \]
Two’s Complement

Slightly trickier: must sign extend the partial products (sometimes!)
OR

Sign extend multiplier and multiplicand to full width of product

And, *use only exact number of lsbs of product*
Multiplication

+       -
\[ \times \ + \quad \times \ + \]
\[ \text{OK} \]

-       +
\[ \times \ - \quad \times \ - \]
\[ \text{find additive inverses} \]
\[ \text{reverse or} \]
\[ \text{sign ext. partial product} \]
\[ \text{OK} \]

\[ \times \ + \quad \times \ + \]
\[ \text{OK} \]
Unsigned Division

\[
\begin{array}{c|c}
11 & 11001 \\
\hline
25/3
\end{array}
\]
Sign Extension

The operation that allows the same 2's complement value to be represented, but using more bits.

\[
\begin{align*}
0 & 0 1 0 1 1 \text{ (5 bits)} \\
\_ \_ \_ & 0 0 1 0 1 1 \text{ (8 bits)} \\
\_ \_ \_ \_ & 1 1 1 0 \text{ (4 bits)} \\
\_ \_ \_ \_ \_ & 1 1 1 1 0 \text{ (8 bits)}
\end{align*}
\]
Zero Extension

The same type of thing as sign extension, but used to represent the same unsigned value, but using more bits

0 0 1 0 1 (5 bits)
_ _ _ 0 0 1 0 1 (8 bits)

1 1 1 1 (4 bits)
_ _ _ _ 1 1 1 1 (8 bits)
## Truth Table for a Few Logical Operations

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X and Y</th>
<th>X nand Y</th>
<th>X or Y</th>
<th>X xor Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</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>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Logical Operations

Logical operations are done **bitwise** on every computer.

Invented example:
Assume that $X, Y,$ and $Z$ are 8-bit variables.

and $Z, X, Y$

If

$X$ is 0 0 0 0 1 1 1 1
$Y$ is 0 1 0 1 0 1 0 1

then

$Z$ is _ _ _ _ _ _ _ _
To selectively **clear** bit(s)

» **clear** a bit means make it a 0

» First, make a **mask**:

  (the generic description of a set of bits that do whatever you want them to)

» **Within** the mask,

  » 1’s for unchanged bits
  
  » 0’s for **cleared** bits

To clear bits numbered 0,1, and 6 of variable X

mask  1 . . 1 0 1 1 1 1 0 0

and use the instruction

and result, X, mask
To selectively \textbf{set} bit(s)

- \textbf{set} a bit means make it a $1$

- First, make a \textbf{mask}:
  - $0$'s for unchanged bits
  - $1$'s for \textbf{set} bits

To set bits numbered 2, 3, and 4 of variable $X$

\begin{verbatim}
mask 0 . . 0 0 0 0 1 1 1 0 0
\end{verbatim}

and use the instruction

or result, $X$, mask
Shift

Moving bits around

1) arithmetic shift
2) logical shift
3) rotate

Bits can move right or left
Arithmetic Shift

Right

sign extension!

Left

0
Logical Shift

Right

Left

Logical left is the same as arithmetic left.
Rotate

Right

No bits lost, just moved

Left
➢ Assume a set of 4 chars. are in an integer-sized variable (X).
➢ Assume an instruction exists to print out the character all the way to the right...

\[
\begin{array}{cccc}
X & 'A' & 'B' & 'C' & 'D'
\end{array}
\]

\[
\text{putc } X \quad \text{(prints D)}
\]

➢ Invent instructions, and write code to print ABCD, without changing X.
Karen's solution

\texttt{rotl X, 8 bits}
\texttt{putc X}  \# A
\texttt{rotl X, 8 bits}
\texttt{putc X}  \# B
\texttt{rotl X, 8 bits}
\texttt{putc X}  \# C
\texttt{rotl X, 8 bits}
\texttt{putc X}  \# D