• I am confused about the process by which we alter the numbers so that they original binary number is biased and will then give the correct result when we use the shortened method.
• http://www.youtube.com/watch?v=4p-9-nK-mwY
Unsigned Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

Standard Addition Function

- Ignores carry output

Implements Modular Arithmetic

$$s = \text{UAdd}_w(u, v) = u + v \mod 2^w$$

$$\text{UAdd}_w(u, v) = \begin{cases} 
    u + v & u + v < 2^w \\
    u + v - 2^w & u + v \geq 2^w 
\end{cases}$$
Visualizing (Mathematical) Integer Addition

- **Integer Addition**
  - 4-bit integers \( u, v \)
  - Compute true sum \( \text{Add}_4(u, v) \)
  - Values increase linearly with \( u \) and \( v \)
  - Forms planar surface
Visualizing Unsigned Addition

- Wraps Around
  - If true sum $\geq 2^w$
  - At most once

```
True Sum
2^{w+1}  \rightarrow  \text{Overflow} \rightarrow 2^w \rightarrow \text{Modular Sum}
```

$UAdd_4(u, v)$
Mathematical Properties

- Modular Addition Forms an *Abelian Group*
  - **Closed** under addition
    \[ 0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1 \]
  - **Commutative**
    \[ \text{UAdd}_w(u, v) = \text{UAdd}_w(v, u) \]
  - **Associative**
    \[ \text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v) \]
  - **0** is additive identity
    \[ \text{UAdd}_w(u, 0) = u \]
  - Every element has additive **inverse**
    - Let \( \text{UComp}_w(u) = 2^w - u \)
    \[ \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \]
Two’s Complement Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

$u + v$

TAdd$_w(u, v)$

TAdd and UAdd have Identical Bit-Level Behavior

- Signed vs. unsigned addition in C:
  ```c
  int s, t, u, v;
  s = (int) ((unsigned) u + (unsigned) v);
  t = u + v
  ```

- Will give $s == t$
TAdd Overflow

**Functionality**
- True sum requires $w+1$ bits
- Drop off MSB
- Treat remaining bits as 2’s comp. integer

<table>
<thead>
<tr>
<th>True Sum</th>
<th>TAdd Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \ldots 001$</td>
<td>000...0</td>
</tr>
<tr>
<td>$2^w - 2^w - 1$</td>
<td>100...0</td>
</tr>
<tr>
<td>$2^w - 1$</td>
<td>011...1</td>
</tr>
<tr>
<td>$-2^w$</td>
<td>-2^w</td>
</tr>
<tr>
<td>$-2^w - 1$</td>
<td>PosOver</td>
</tr>
<tr>
<td>$-2^w - 1 - 1$</td>
<td>-2^w - 1-1</td>
</tr>
<tr>
<td>$0^w \ldots 0$</td>
<td>NegOver</td>
</tr>
</tbody>
</table>
• Practice problems:
  – 2.30
  – 2.31
Visualizing 2’s Complement Addition

- **Values**
  - 4-bit two’s comp.
  - Range from -8 to +7

- **Wraps Around**
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If sum $<-2^{w-1}$
    - Becomes positive
    - At most once
Characterizing $T\text{Add}$

**Functionality**
- True sum requires $w+1$ bits
- Drop off MSB
- Treat remaining bits as 2’s comp. integer

$T\text{Add}_w(u,v) = \begin{cases} 
    u + v + 2^w & u + v < T\text{Min}_w \quad \text{(NegOver)} \\
    u + v & T\text{Min}_w \leq u + v \leq T\text{Max}_w \\
    u + v - 2^w & T\text{Max}_w < u + v \quad \text{(PosOver)}
\end{cases}$
Mathematical Properties of TAdd

- Isomorphic Group to unsigneds with UAdd
  \[ TAdd_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v))) \]
  - Since both have identical bit patterns

- Two’s Complement Under TAdd Forms a Group
  - Closed, Commutative, Associative, 0 is additive identity
  - Every element has additive inverse

\[
T\text{Comp}_w(u) = \begin{cases} 
-u & u \neq T\text{Min}_w \\
T\text{Min}_w & u = T\text{Min}_w 
\end{cases}
\]
Multiplication

- Computing Exact Product of $w$-bit numbers $x, y$
  - Either signed or unsigned

- Ranges
  - Unsigned: $0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - Up to $2w$ bits
  - Two’s complement min: $x \times y \geq (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
    - Up to $2w-1$ bits
  - Two’s complement max: $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$
    - Up to $2w$ bits, but only for $(TMin_w)^2$

- Maintaining Exact Results
  - Would need to keep expanding word size with each product computed
  - Done in software by “arbitrary precision” arithmetic packages
Unsigned Multiplication in C

Operands: \( w \) bits

True Product: \( 2w \) bits

Discard \( w \) bits: \( w \) bits

- Standard Multiplication Function
  - Ignores high order \( w \) bits

- Implements Modular Arithmetic
  \[
  \text{UMult}_w(u, v) = u \cdot v \mod 2^w
  \]
Signed Multiplication in C

Operands: \( w \) bits

True Product: \( 2^w \) bits

Discard \( w \) bits: \( w \) bits

Standard Multiplication Function

- Ignores high order \( w \) bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same
Power-of-2 Multiply with Shift

**Operation**
- \( u << k \) gives \( u \times 2^k \)
- Both signed and unsigned

**Operands:** \( w \) bits

**True Product:** \( w+k \) bits

**Discard** \( k \) bits: \( w \) bits

**Examples**
- \( u << 3 \) \( = \) \( u \times 8 \)
- \( u << 5 - u << 3 \) \( = \) \( u \times 24 \)
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically
Compiled Multiplication Code

C Function

```
int mul12(int x)
{
    return x*12;
}
```

Compiled Arithmetic Operations

```
leal (%eax,%eax,2), %eax
sall $2, %eax
```

Explanation

```
t <- x+x*2
return t << 2;
```

- C compiler automatically generates shift/add code when multiplying by constant
• Practice problems:
  – 2.34