Lecture Overview

1. Integer Arithmetic (Addition, Subtraction, Multiplication, Division, Sign Extension, Logical Operations)
2. Data Representation (Floating Point)

Unsigned Representation

\[ B2U_w(x_{vec}) = \sum_{i=0}^{w-1} x_i 2^i \]

\[ B2U_4([0101]) = 0.2^3 + 1.2^2 + 0.2^1 + 1.2^0 = 5 \]

\[ B2U_w \] is a bijection:
- associates a unique value to each bit vector of length \( w \)
- each integer between 0 and \( 2^w-1 \) has a unique binary representation as a bit vector of length \( w \)

Class Announcements

1. Questions about Assignment 1?
2. Come meet us at office hours for hands-on help. <2 students show up every hours now.
3. Start Early! Assign 1 is much much harder than Assign 0!
4. Hands-on overview of File I/O and related C Programming aspects relevant to P1 during lecture?
Two’s complement Representation

\[ B2T_w(x_{w:1}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i2^i \]

\[ B2T_4([1011]) = -1.2^3 + 0.2^2 + 1.2^1 + 1.2^0 = -5 \]

\( B2T_w \) is a bijection:
- associates a unique value to each bit vector of length \( w \)
- each integer between \(-2^{w-1}\) and \(2^{w-1}-1\) has a unique binary representation as a bit vector of length \( w \)

Conversion from 2’s complement to unsigned

Rule: The numeric values might change but the bit patterns do not.

\( T2U_w(x) \) equals:
- \( x+2^w \), if \( x < 0 \)
- \( x \), if \( x \geq 0 \)

2’s Complement Addition

Of two signed 2’complement \( w \) bit values \( X \& Y \)

\( X + Y \) equals:
- \( X+Y-2^w \), if \( 2^{w-1} \leq (X+Y) \) Positive overflow
- \( X+Y \), if \( -2^{w-1} \leq (X+Y) < 2^{w-1} \) Normal
- \( X+Y+2^w \), if \( (X+Y) < -2^{w-1} \) Negative overflow

Two’s Complement Addition

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 0 & \text{(Red)} \\
+ & 0 & 0 & 0 & 0 & 0 & 1 & \text{(Red)} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & \text{(Red)} \\
+ & 0 & 0 & 1 & 1 & 0 & 0 & 0 & \text{(Red)} \\
\end{array}
\]
Two's Complement Addition

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

( -2 )

( 1 )

( -1 )

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:

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

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

Carry out from msbs is overflow in unsigned
**Unsigned Overflow Detection**

6-bit examples:

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

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

No Overflow

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

Overflow

- Carry out from msbs is overflow in unsigned

**Two's Complement Overflow Detection**

When adding 2 numbers of like sign

\[
\begin{align*}
&+ \to + \\
&- \to - \\
\text{and the sign of the result is different!}
\end{align*}
\]

\[
\begin{align*}
&+ \quad - \\
&+ \quad + \\
&- \quad + \\
&\text{Overflow!} \quad \text{Overflow!}
\end{align*}
\]

**Addition**

**Overflow detection: 2's complement**

6-bit examples

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

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

**Addition**

**Overflow detection: 2's complement**

6-bit examples

\[
\begin{align*}
&1 \ 1 \ 1 \ 1 \ 1 \ 1 \ (-1) \\
+ &1 \ 1 \ 1 \ 1 \ 1 \ 1 \ (-1) \\
\hline
&1 \ 1 \ 1 \ 1 \ 0 \ (-2)
\end{align*}
\]

\[
\begin{align*}
&1 \ 0 \ 0 \ 0 \ 0 \ 0 \ (32) \\
+ &0 \ 1 \ 1 \ 1 \ 1 \ 1 \ (31) \\
\hline
&1 \ 1 \ 1 \ 1 \ 1 \ 1 \ (-1)
\end{align*}
\]

\[
\begin{align*}
&0 \ 1 \ 1 \ 1 \ 1 \ 1 \ (31) \\
+ &0 \ 1 \ 1 \ 1 \ 1 \ 1 \ (31) \\
\hline
&1 \ 1 \ 1 \ 1 \ 1 \ 0 \ (-2)
\end{align*}
\]
Subtraction
basic algorithm is like decimal...

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

\[
\begin{array}{c}
111000 \\
- 010110
\end{array}
\]

\[
\begin{array}{c}
56 \\
22
\end{array}
\]

\[
\begin{array}{c}
100010 \\
34
\end{array}
\]

\[
\begin{array}{c}
-8 \\
22
\end{array}
\]

\[
\begin{array}{c}
-30 \\
22
\end{array}
\]

Subtraction
For two's complement representation

1. The implementation redefines the operation:
   \( a - b \) becomes \( a + (-b) \)
2. 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

2’s Complement Inverse
Additive inverse of a 2’complement w bit value \( X \) equals:

- \( -2^{w-1} \), if \( x = -2^{w-1} \)
- \( -X \), if \( X > -2^{w-1} \)
2's Complement Inverse: Easy Techniques

1) **Toggle all bits and then add 1:**
   E.g. Inverse of 0101 (5) is 1011 (-5)
   Inverse of 1000 (-8) is 1000 (-8)

2) **Toggle all bits until (not including) the rightmost 1 bit:**
   E.g. Inverse of 0111 (7) is 1001 (-7)
   Inverse of 1010 (-6) is 0110 (6)

---

Subtraction

**6-bit, 2's complement examples**

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>001111 (15) - 111100 (-4)</td>
<td>010011 (19)</td>
</tr>
<tr>
<td>000010 (2) - 011100 (28)</td>
<td>100110 (-26)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>001111 (15) - 111100 (-4)</td>
<td>010011 (19)</td>
</tr>
<tr>
<td>000010 (2) - 011100 (28)</td>
<td>100110 (-26)</td>
</tr>
</tbody>
</table>
**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
  
  \[ \begin{array}{c}
  \text{n bits} \\
  \times \text{n bits} \\
  \hline
  \text{n + n bits may be needed}
  \end{array} \]

In HW, space is always designated for a larger precision product.

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

**Unsigned Multiplication**

\[
\begin{array}{c}
01111 \\
\times \text{01101}
\end{array}
\]

\[
\begin{array}{c}
01111 \\
01111 \\
01111 \\
01111 \\
\hline
11000011
\end{array}
\]

128+64+2+1 = 195
00011 = 3
**Unsigned Multiplication**

\[
\begin{array}{cccccc}
& & & & & \uparrow \\
31 & 11111 & 11111 & 11111 & 11111 & 11111 \\
\downarrow & & & & & \\
& & & & & \\
\end{array}
\]

**Unsigned Multiplication**

\[
\begin{array}{cccc}
11111 & 31 \\
31 & 512+256+128+64+1 = 961 \\
& 00001 \quad 1 \\
\end{array}
\]

**Two’s Complement**

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

**OR**

Sign extend multiplier and multiplicand to full width of product

\[
\begin{array}{cccccccc}
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
\end{array}
\]

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

\[ \begin{align*}
\text{OK} & \quad \text{+} \quad \text{×} \quad \text{+} \\
\text{sign ext. partial product} & \quad \downarrow \quad \text{reverse} \\
\text{additive inverses} & \quad \downarrow \\
\text{+} & \quad \text{+} \\
\text{OK} & 
\end{align*} \]

### Unsigned Division

\[ \begin{align*}
11 & \quad 11001 \\
\text{11} & \quad \text{25/3}
\end{align*} \]

### Unsigned Division

\[ \begin{align*}
11 & \quad 11001 \\
\text{11} & \quad \text{25/3}
\end{align*} \]

### Sign Extension

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

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

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

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

1 1 1 0 (4 bits)
1 1 1 1 1 1 1 0 (8 bits)

---

**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)

---

**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 0 0 0 1 0 1 (8 bits)

1 1 1 1 (4 bits)
0 0 0 0 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 $1 . . 1 0 1 1 1 0 0$
and use the instruction
and result, $X$, mask

To selectively **set** bit(s)

- **set** a bit means make it a 1
- First, make a **mask**:
  - 0's for unchanged bits
  - 1's for set bits

To set bits numbered 2,3, and 4 of variable $X$
mask $0 . . 0 0 0 1 1 1 0 0$
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

Left

sign extension!

Logical Shift

Right

Left

Logical left is the same as arithmetic left.

Rotate

Right

Left

No bits lost, just moved

> 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...

\[ X \left[ 'A', 'B', 'C', 'D' \right] \]

\texttt{putc X} \hspace{1em} (prints D)

> Invent instructions, and write code to print ABCD, without changing X.