Number Systems II:

2’s Complement, Arithmetic, Overflow, & Writing Bit-wise Logical & Shifting Code
Logic and arithmetic are two primary ways of manipulating stored state

Computer can do 2 things
1) Store state (How do we interpret stored bits?)
2) Manipulate state (How do we perform meaningful mathematical operations?)
The class consists roughly of 4 quarters: (Bolded words are the big ideas of the course, pay attention when you hear these words)

1. You will build a simple computer processor. Build and create state machines with **data**, **control**, and **indirection**

2. You will learn how high-level language code executes on a processor. Time limitations create dependencies in the state of the processor.

3. You will learn why computers perform the way they do. Physical limitations require **locality** and **indirection** in how we access state.

4. You will learn about hardware mechanisms for parallelism. **Locality**, **dependencies**, and **indirection** on performance enhancing drugs.

- We will have a SPIMbot contest!
Today’s lecture

- Two’s complement signed binary representation
  - Negating numbers in Two’s complement
  - Sign extension

- Bit-wise shift operations
  - Writing bit-wise logical and shifting code

- Two’s complement arithmetic
  - Addition
  - Subtraction
  - Overflow
Two’s complement has asymmetric ranges; there is one more negative number than positive number. Here, you can represent -8 but not +8.
Negating Numbers in 2’s Complement

- To negate a number:
  - Complement each bit and then add 1.

- Example:

  \[
  0100 = +4_{10} \quad \text{(a positive number in 4-bit two’s complement)}
  
  = \quad \text{(invert all the bits)}
  
  = -4_{10} \quad \text{(and add one)}
  
  = \quad \text{(invert all the bits)}
  
  = +4_{10} \quad \text{(and add one)}
  \]

*Sometimes, people talk about “taking the two’s complement” of a number. This is a confusing phrase, but it usually means to negate some number that’s already in two’s complement format.*
Converting 2’s Complement to Decimal

- Algorithm 1:
  - if negative, negate; then do unsigned binary to decimal

- Algorithm 2:
  - Same as with n-bit unsigned binary
    - Except, the MSB is worth $-(2^{n-1})$

- Example:

  \[ \begin{align*}
  1100 &= -4_{10} \quad \text{(a negative number in 4-bit two’s complement)}
  \end{align*} \]
Sign Extension

- In everyday life, decimal numbers are assumed to have an infinite number of 0’s in front of them. This helps in “lining up” numbers.
- To subtract 231 and 3, for instance, you can imagine:

```
  231
- 003
  228
```
- This works for positive 2’s complement numbers, but not negative ones.
- To preserve sign and value for negative numbers, we add more 1’s.
- For example, going from 4-bit to 8-bit numbers:
  - 0101 (+5) should become 0000 0101 (+5).
  - But 1100 (-4) should become 1111 1100 (-4).
- The proper way to extend any signed binary number is to replicate the sign bit.
Sign Extension, cont.
What you need to know for Lab 2.
Review: Bitwise Logical operations

Last time we introduced bit-wise logical operations:

- unsigned char c = a | b;  
  \( \begin{array}{c} 0 \ 1 \ 0 \ 1 \\ 0 \ 0 \ 0 \ 0 \\ 0 \ 1 \ 0 \ 1 \\ 1 \ 1 \ 1 \ 1 \end{array} \)  \text{(bit-wise OR)}

- unsigned char d = a & b;  
  \( \begin{array}{c} 0 \ 1 \ 0 \ 1 \\ 0 \ 0 \ 0 \ 0 \\ 0 \ 1 \ 0 \ 1 \\ 1 \ 1 \ 1 \ 1 \end{array} \)  \text{(bit-wise AND)}

- unsigned char e = a ^ b;  
  \( \begin{array}{c} 0 \ 1 \ 0 \ 1 \\ 0 \ 0 \ 0 \ 0 \\ 0 \ 1 \ 0 \ 1 \\ 1 \ 1 \ 1 \ 1 \end{array} \)  \text{(bit-wise XOR)}

- unsigned char n = ~a;  
  \( \begin{array}{c} 0 \ 1 \ 0 \ 1 \\ 0 \ 0 \ 0 \ 0 \\ 1 \ 0 \ 1 \ 0 \\ 1 \ 0 \ 1 \ 0 \end{array} \)  \text{(bit-wise NOT)}
Bit-wise shifting

- When doing bit-wise logical operations, it can be useful to “shift” bits to the left or right within a word.

- Left shift:

  ```
  unsigned char f = b << 5;
  (left shift)
  ```

  We are shifting bits toward the most significant bit (MSB); we call this a left shift because we think of the MSB being on the left.
Bit-wise shifting, cont.

- Two kinds of right shift, depends on type of variable:
  - **Unsigned numbers**
    ```
    unsigned char g = f >> 2;
    (right shift logical)
    if unsigned, right shift in zeros
    0 0 1 1 1 0 0 0
    bottom bits disappear...
    ```
  - **Signed numbers**
    ```
    signed char h = f;
    unsigned char i = h >> 2;
    (right shift arithmetic)
    if signed, sign extend MSB
    1 1 1 1 1 1 0 0
    bottom bits disappear...
    ```

Note: `x >> 1` not the same as `x/2` for negative numbers; compare `(-3)>>1` with `(-3)/2`
Bit-shifting has lower precedence than arithmetic but higher than bitwise operators

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher</td>
<td>* / %</td>
<td>Multiplication, division, and modulus</td>
</tr>
<tr>
<td></td>
<td>+ -</td>
<td>Addition and subtraction</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt; &gt;&gt;</td>
<td>Bitwise shifting</td>
</tr>
<tr>
<td></td>
<td>&amp; ^</td>
<td></td>
</tr>
<tr>
<td>Lower</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
</tbody>
</table>
Useful for extracting bits

- We have the unsigned 8-bit word: \( b_7b_6b_5b_4b_3b_2b_1b_0 \)
- And we want the 8-bit word: \( 00000b_5b_4b_3 \)
  - i.e., we want to extract bits 3-5.

- We can do this with bit-wise logical & shifting operations
  - \( y = (x >> 3) \& 0x7; \)

\[
\begin{array}{c|c}
\text{x} & \text{b_7b_6b_5b_4b_3b_2b_1b_0} \\
\hline
\text{x >> 3} & \\
\hline
(x >> 3) \& 0x7 & \\
\end{array}
\]
Useful for merging two bit patterns

- We have 2 unsigned 8-bit words: 
  \[ a_7a_6a_5a_4a_3a_2a_1a_0 \]
  \[ b_7b_6b_5b_4b_3b_2b_1b_0 \]

- And we want the 8-bit word: 
  \[ a_7b_6a_5b_4a_3b_2a_1b_0 \]
Binary addition with 2’s Complement

- You can add two’s complement numbers just as if they are unsigned numbers.
  - Recall, this was the whole reason for this representation

```
0 1 0 1 1 1 11
+ 1 1 1 0 0 + (-4)
```

0 1 0 1 1 1 11
+ 1 1 1 0 0 + (-4)
Subtraction

- We can implement subtraction by negating the 2nd input and then adding:

\[
\begin{array}{cccccc}
0 & 1 & 1 & 0 & 1 & 13 \\
\hline
- & 0 & 1 & 0 & 1 & 0 & -10 \\
\hline
 & 0 & 1 & 0 & 1 & 0 & +(-10)
\end{array}
\]
Why does this work?

- For n-bit numbers, the negation of B in two’s complement is $2^n - B$ (this is alternative way of negating a 2’s-complement number).

  \[
  A - B = A + (-B) \\
  = A + (2^n - B) \\
  = (A - B) + 2^n
  \]

- If $A \geq B$, then $(A - B)$ is a positive number, and $2^n$ represents a carry out of 1. Discarding this carry out is equivalent to subtracting $2^n$, which leaves us with the desired result $(A - B)$.

- If $A < B$, then $(A - B)$ is a negative number $-(B - A)$ and we have $2^n - (B - A)$. This corresponds to the desired result, $(A - B)$, in two’s complement form.
Overflow Review

- Recall that when we add two numbers the result may be larger than we can represent.

  \((in\ 5b\ 2's\ complement\ we\ can\ represent\ -16\ to\ +15)\)

\[
\begin{array}{cccccc}
  & 0 & 1 & 0 & 1 & 1 & \text{Augend} \\
\hline
+ & 0 & 1 & 1 & 1 & 0 & \text{Addend} \\
\hline
  & 1 & 1 & 0 & 0 & 1 & \text{Sum} \\
\end{array}
\]

- The same thing can happen when we add negative numbers.

\[
\begin{array}{cccccc}
  & 1 & 1 & 0 & 0 & 1 & \text{Augend} \\
\hline
+ & 1 & 0 & 1 & 0 & 0 & \text{Addend} \\
\hline
  & 0 & 1 & 1 & 0 & 1 & \text{Sum} \\
\end{array}
\]
“Carry-out” is a procedure, “Overflow” is an interpretation

**Carry-out**
- Occurs at every bit-position
- The process of moving larger numbers to higher bit positions
- Focuses on bit-wise operations

**Overflow**
- Can only be seen after completing an entire mathematical operation
- When the interpretation of a set of bits does not match the expected value after a mathematical operation
- Focuses on representational range (i.e., 4 bits represent 0-15)
How can we know if overflow has occurred?

- The easiest way to detect signed overflow is to look at all of the sign bits.

- Overflow occurs only in the two situations above:
  - If you add two positive numbers and get a negative result.
  - If you add two negative numbers and get a positive result.

- Overflow cannot occur if you add a positive number to a negative number. Do you see why?
Overflow in software (e.g., Java programs)

```java
public class overflow {
    public static void main(String[] args) {
        int i = 0;
        while (i >= 0) {
            i++;
        }  
        System.out.println("i = " + i);
        i--;  
        System.out.println("i = " + i);
        i++;
        System.out.println("i = " + i);
    }
}}
```

Output:
```
i = -2147483648 2^{31}
i = 2147483647 2^{31}-1
i = -2147483648
```