Integer Representation

CMPU 224 – Computer Organization
Jason Waterman
Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>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>

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A ^ B</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>

<table>
<thead>
<tr>
<th>A</th>
<th>~A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Bit Vectors and Bit-Level Operations in C

• A bit vector is a string of zeros and ones of some fixed length \( w \).
• C supports bitwise Boolean operations on bit vectors:
  • AND (\&), OR (|), NOT (~), and XOR (^)
• Can be used on any integral data type (char, short, int, long):
  • View arguments as bit vectors
  • Operation is applied bit-wise
• Example:
  • char a = 0x69; char b = 0x55;
  • a & b; // evaluates to 0x41
  • a | b; // evaluates to 0x7D
  • a ^ b; // evaluates to 0x3C
Shift Operations in C

• Left Shift: \( x \ll y \)
  - Shift bit-vector \( x \) left \( y \) positions
    - Throw away extra bits on left
    - Fill with 0’s on right

• Right Shift: \( x \gg y \)
  - Shift bit-vector \( x \) right \( y \) positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0’s on left
  - Arithmetic shift
    - Replicate most significant bit on left

• Undefined Behavior
  - Shift amount < 0 or ≥ data size

**Examples:**

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. ( \gg 2 )</td>
<td>00011000</td>
</tr>
<tr>
<td>Arith. ( \gg 2 )</td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. ( \gg 2 )</td>
<td>00101000</td>
</tr>
<tr>
<td>Arith. ( \gg 2 )</td>
<td>11101000</td>
</tr>
</tbody>
</table>
Logic Operations in C

• No Boolean type in C
  • View 0 as “False”
  • Anything nonzero as “True”
  • Logical operators always return 0 (False) or 1 (True)
  • Early termination

• Logical operators
  • &&, ||, ! (logical AND, OR, NOT)

• Examples (char data type)
  • !0x41 -> 0x00
  • !0x00 -> 0x01
  • !!0x41 -> 0x01
  • 0x69 && 0x55 -> 0x01
  • 0x69 || 0x55 -> 0x01
Today

• How integers (signed and unsigned) are represented in modern computer systems
Representing unsigned integers

• Given $n$ bits to store an integer, we can represent $2^n$ different values

• If we just care about non-negative (aka **unsigned**) integers, we can easily store the values $0, 1, 2, \ldots, 2^n-1$
  • E.g., for 4 bits
    • 0x2 = 2
    • 0xB = 11
    • 0xF = 15 = $2^4-1$
  • Binary number
Representing negative integers

• We have seen how to represent **unsigned integers** (i.e., non-negative integers) as **unsigned binary** numbers
  • Every number between 0 and \(2^w-1\) has a unique encoding as a \(w\)-bit value
  • Addition works as we expect it to

• How do we represent negative integers?

• Three common encodings:
  • Sign and magnitude
  • Ones’ complement
  • Two’s complement
Sign and magnitude

• Use one bit to represent sign, remaining bits represent magnitude

• With $n$ bits, have $n-1$ bits for magnitude
  • E.g., with 4 bits, can represent integers
    -7, -6, ..., -1, 0, 1, ..., 6, 7

• MSB (most significant bit) represents the sign
  • 0 is positive
  • 1 is negative

1011 represents -3

sign: - magnitude: 3
Properties of sign and magnitude

• Advantages:
  • Straight-forward and intuitive

• Issues:
  • Arithmetic operations need different implementations for signed and unsigned numbers
    • E.g., addition, using 4 bits
      • unsigned: $0001 + 1001 = 1 + 9 = 10 = 1010$
      • sign and magnitude: $0001 + 1001 = 1 + -1 = 0$, which is not 1010
  • Two different representations of zero!
    • E.g., using 4 bits, 1000 and 0000 both represent zero!
Ones’ complement

• Negation is performed by performing a bitwise not (~) on the number
  • In other words, flip all the bits in the number
  • For example, using 4 bit numbers:
    • 6 = 0110
    • -6 = 1001

• Like sign and magnitude, first bit indicates whether number is negative
  • If the msb (most significant bit) is 0, treat it like an unsigned binary number
  • If the msb is 1, the number is negative, flip all the bits to see its magnitude

• Using n bits, can represent numbers $2^n - 1$ values
  • E.g., using 4 bits, can represent integers
    -7, -6, ..., -1, 0, 1, ..., 6, 7
Properties of ones’ complement

• We still have two different representations of zero
  • E.g., using 4 bits, 1111 and 0000 both represent zero

• Same implementation of arithmetic operations for signed and unsigned!
  • E.g., addition using 4 bits
    • unsigned: 0001 + 1001 = 1 + 9 = 10 = 1010
    • ones’ complement: 0001 + 1001 = 1 + -6 = -5 = 1010
Two’s complement

• Take the ones’ complement of the number and add one
  • Flip all the bits and add one to the number
• Example: Using 4-bit numbers take the two’s complement of 3

\[
(3_{10})_{0011} \overset{1\text{’s comp}}{\Rightarrow} \frac{1100}{1101} = 1101 (3_{10}) \overset{2\text{’s comp}}{\Rightarrow} \frac{0010}{0011} = 0011 (3_{10})
\]

• Like sign and magnitude and ones’ complement, first bit indicates whether number is negative

<table>
<thead>
<tr>
<th>Binary</th>
<th>Ones’ Complement</th>
<th>Two’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-7</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>-6</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>-5</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>-4</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>-2</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>-0</td>
<td>-1</td>
</tr>
</tbody>
</table>
Properties of two’s complement

• Same implementation of arithmetic operations for signed and unsigned
  • E.g., addition, using 4 bits
    • unsigned: 0001 + 1001 = 1 + 9 = 10 = 1010
    • two’s complement: 0001 + 1001 = 1 + -7 = -6 = 1010

• Only one representation of zero!
  • Simpler to implement operations

• Not symmetric around zero
  • Can represent one more negative number than positive numbers
  • Using n bits, can represent numbers $2^n$ values
    • E.g., using 4 bits, can represent integers
      -8, -7, ..., -1, 0, 1, ..., 6, 7

• Most common representation of negative integers in computers
Converting to and from two’s complement

• To encode a negative number in two’s complement in n bits:
  • Compute out the binary notation for the absolute value using all n bits
  • Invert the bits and add 1
  • E.g., to encode -5 using 8 bits
    • 5 = 00000101 using all 8 bits
    • Invert the bits: 11111010
    • Add one: 11111010 + 1 = 11111011
    • -5 encoded in two’s complement using 8 bits is 11111011

• To decode a two’s complement number:
  • If the msb is 0 then number is positive
  • If the msb is 1, then number is negative:
    • invert bits and add 1
    • Value gives you the magnitude of the negative number
Two’s Complement Interpretation

• You can interpret a two’s complement number as having a “negative weighting” in the MSB

<table>
<thead>
<tr>
<th>(-2^3) = -8</th>
<th>(2^2) = 4</th>
<th>(2^1) = 2</th>
<th>(2^0) = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>+4</td>
<td>+2</td>
<td>+1</td>
</tr>
</tbody>
</table>

• For any two’s complement representation
  • The largest negative number will have a 1 in the msb and the rest of the bits 0
  • The largest positive number will have a 0 in the msb and the rest of the bits 1
  • -1 is represented by all ones
Quick two’s complement conversion

• One trick for taking the to two’s complement of a number quickly
  • Flip all the bits, then starting from the lsb, flip all the 1s you see to 0s until you get to a 0, flip that 0 to a 1 and stop

• Example: using 6-bit two’s complement numbers, convert 4 to -4
  • Write out 4 in binary using 6 bits $000100$
  • Flip all the bits $111011$
  • Flip the rightmost 1’s and the first 0 $111100$

• For negative two’s complement numbers, leading 1s are similar to leading 0s for positive numbers

$$
\begin{array}{c}
\frac{3 - 011}{100} \\
-4 + 2 + 1 \\
-4
\end{array} \quad 
\begin{array}{c}
\frac{4 - 010}{1100} \\
-8 + 4 + 2 + 1 \\
-8 + 4 = -4
\end{array} \quad 
\begin{array}{c}
\frac{5 - 010}{11100} \\
-16 + 8 + 4 = -4
\end{array} \quad 
\begin{array}{c}
\frac{6 - 010}{111100} \\
-32 + 16 + 8 + 4 = -4
\end{array}
$$
Signed vs. unsigned in C

- `int x;` // declares `x` as a **signed** integer
- `unsigned int x;` // declares `x` as an **unsigned** integer

- Constants are signed by default
  - `42` // would be stored as a two’s complement number
  - `42u` // would be stored as an unsigned binary number

- C allows conversion between unsigned and signed
  - Most systems follow the rule that the underlying bit representation does not change

```c
int main(void) {
    unsigned char ux = 0xff;
    char x = ux;
    printf("%d %d\n", ux, x);
}
```

**What will this code print?**

`255` and `-1` !!!
Expanding the bit representation of a number

• How do you convert from a smaller integer data type to a larger integer data type (e.g., a short to an int) while retaining the same value?
  • Should always be possible because the larger type will have a wider range of numbers than the smaller type

• To convert an unsigned number to a larger data type:
  • Add leading zeros to the representation
  • Known as zero extension

• To convert a two’s complement number to a larger data type:
  • Perform sign extension
  • Add copies of the msb (the sign bit) of the smaller representation to the extra bits of the larger representation

Example:
Extend a 5-bit number to an 8-bit

<table>
<thead>
<tr>
<th>UNSIGNED</th>
<th></th>
<th>SIGNED</th>
</tr>
</thead>
<tbody>
<tr>
<td>10110</td>
<td>00010110</td>
<td>11101110</td>
</tr>
<tr>
<td>01100</td>
<td>00001100</td>
<td>01100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00001100</td>
</tr>
</tbody>
</table>
Truncating Numbers

• When reducing the number of bits in a number, the smaller number may not be able to correctly represent the larger number
• We reduce the number of bits representing a number by truncating the uppermost (high-order) bits
  • Reinterpret this truncated number
• For unsigned numbers, it is equivalent to performing a mod $2^k$ on the original value where $k$ is the smaller bit width
• A similar property holds for signed numbers except that it then converts the msb into a sign bit
  • Which means it is possible that truncating a signed number changes its sign
• **Rule of thumb**: if the larger number is within the expressible range of the smaller number, the conversion will yield a correct result
What is going on with this cartoon?

They are counting sheep using a 16-bit two’s complement number!