Lecture 4: Operations on Values
Arithmetic Logic Unit (ALU)

• circuit that performs bitwise operations and arithmetic on integer binary types
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>And</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Or</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Not</th>
<th>~</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exclusive-Or (Xor)</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
General Boolean algebras

- Bitwise operations on words

```
01101001 & 01010101 = 01000001
| 01010101 = 01111101
^ 01010101 = 00111100
~ 01010101 = 10101010
```

- How does this map to set operations?
Exercise: Boolean algebras

• Assume: \( a = 01101101 \), \( b = 01010101 \)

• What are the results of evaluating the following Boolean operations?

  • \( \sim a \)
  • \( \sim b \)
  • \( a \land b \)
  • \( a \lor b \)
  • \( a \oplus b \)
  • \( ((a \oplus b) \land \sim b) \lor (\sim (a \oplus b) \land b) \)
Example: Using Boolean Operations

void f(int *x, int*y){
    *y = *x ^ *y;
    *x = *x ^ *y;
    *y = *x ^ *y;
}
Bitwise vs Logical Operations in C

• Apply to any “integral” data type
  • int, unsigned, long, short, char

• Bitwise Operators &!, |, ~, ^
  • View arguments as bit vectors
  • operations applied bit-wise in parallel

• Logical Operators &&, ||, !
  • View 0 as “False”
  • View anything nonzero as “True”
  • Always return 0 or 1
  • Early termination
Exercise: Bitwise vs Logical Operations

• Assume char data type (one byte)

  • ~0x41
  • ~0x00
  • ~~0x41

  • 0x69 & 0x55
  • 0x69 | 0x55

  • !0x41
  • !0x00
  • !!0x41

  • 0x69 && 0x55
  • 0x69 || 0x55
Bit Shifting

• **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 if you shift amount < 0 or ≥ word size

Choice between logical and arithmetic depends on the type of data
Example: Bit Shifting

- **Unsigned**
  - 0x41 << 4
  - 0x41 >> 4

- **Signed**
  - 41 << 4
  - 41 >> 4
  - -41 << 4
  - -41 >> 4
Addition Example

• Compute 5 + 1 assuming all ints are stored as three-bit unsigned values

• Compute -3 + 1 assuming all ints are stored as three-bit signed values (two's complement)
Addition and Subtraction

• Usual addition and subtraction
  • Like you learned in second grade, only binary
  • Same for unsigned and signed
  • … but error conditions differ
Error Cases

• Unsigned addition:

  \[ x +_w^u y = \begin{cases} 
  x + y & \text{(normal)} \\
  x + y - 2^w & \text{(overflow)} 
\end{cases} \]

  • overflow has occurred iff \( x +_w^u y < x \)

• Signed addition:

  \[ x +_w^t y = \begin{cases} 
  x + y - 2^w & \text{(positive overflow)} \\
  x + y & \text{(normal)} \\
  x + y + 2^w & \text{(negative overflow)} 
\end{cases} \]

  • overflow has occurred iff \( x > 0 \) and \( y > 0 \) and \( x +_w^t y < 0 \)

  or \( x < 0 \) and \( y < 0 \) and \( x +_w^t y > 0 \)
Flags

- A flag is a one-bit value: 1 is “set” and 0 is “unset”
- Flags record conditions of previous arithmetic operations
  - **C**: The carry-out flag from the last bit; indicates unsigned overflow
  - **V**: Indicates if the result, interpreted as a signed value, is erroneous. For addition, this means that the signs of the operands agree and the result has a different sign
  - **Z**: Set if the result is zero
  - **N**: The sign bit of the result; indicates a negative signed result
Multiplication Example

• Compute $5 \times 3$ assuming all ints are stored as three-bit unsigned values

• Compute $-3 \times 3$ assuming all ints are stored as three-bit signed values (two's complement)
Multiplication

• Usual Multiplication
  • Like elementary school, only in binary
  • Product can be two words long; it may be truncated to one word
  • Bit level equivalence for unsigned and signed
Error Cases

• Unsigned multiplication:
  • $x^u_w \cdot y = (x \cdot y) \mod 2^w$

• Signed multiplication:
  • $x^t_w \cdot y = U2T((x \cdot y) \mod 2^w)$
Multiplying with Shifts

C uses << and >>. The arithmetic/logical choice is made according the the operands being signed/unsigned.

Java has no unsigned integers, but it has a third shift >>> for logical right shift.

We can multiply (often faster than with the processor’s multiply instruction) with shifts.

- $x \times 24 = x \times 32 - x \times 8$
  $= (x << 5) - (x << 3)$

Most compilers will generate this code automatically.
Signed Division by a Power of 2

- \( \text{x} \gg \text{k} \) computes \( \text{x} / 2^\text{k} \), rounded towards \( -\infty \)

- C on Intel processors rounds towards 0
  - \(-11 \gg 2 == -3\), but \(-11/4 == -2\)

- Solution: If \( x < 0 \), add \( 2^k - 1 \) before shifting
  - Why does this work?

```c
if (x < 0)
    x += (1 << k) - 1;
return x >> k;
```