Lecture 5 - Boolean Logic, Bitstrings

Transistor Logic Example:

- Construct a NOR gate. Looks kind of familiar right?
- Remember: pMOS passes a strong 1, so should always be connected to $V_{dd}$
- How do we construct an OR? (add an inverter)
- On your own: design an AND gate
- Circuit symbols for what we’ve seen so far
- Symbols for what we’ve seen so far ($\wedge$, juxtaposition for AND; $\vee$, $+$ for OR)

Logical Expressions contd.

- Note AND is also called the “conjunction” of two expressions
- OR is called the “disjunction”
- Note that OR is inclusive. That is, $x$ OR $y$ is 1 when $x$ and $y$ are both 1
- We also have an exclusive version, appropriately named exclusive OR (XOR)
- Truth table:
  - $x \mid y \mid x$ XOR $y$
  - =$=$
    - $0 \mid 0 \mid 0$
    - $0 \mid 1 \mid 1$
    - $1 \mid 0 \mid 1$
    - $1 \mid 1 \mid 0$
- Here, $x$ XOR $y$ is only one when $x \neq y$
- IMPLY $\rightarrow$ conditional operator
- $X \rightarrow Y$ is 0 iff $X = 0$. Can be read “if $X$, then $Y$.” Note that $X \rightarrow Y$ is the same thing as $\neg X$ OR $Y$ (by definition of implication)
- If we treat true and false numerically, then it also happens to be $X \leq Y$ (try it by making the truth tables)
- Equivalence (also called biconditional) - IFF (if and only if)
- $X$ IFF $Y = 1$ only when $X = Y$ ($X = 0 \& Y = 0$ or $X = 1 \& Y = 1$)
- This also happens to be the opposite of XOR (so it is sometimes called XNOR). Hint: try pronouncing NXOR
- Operator symbols for XOR ($\oplus$), IMPL ($\Rightarrow$, $\rightarrow$), IFF ($\iff$, $\leftrightarrow$)
- Be careful with bars (not) Bar(XY) is not same as Bar(x)Bar(y) (clear when drawn out)

Truth tables for larger expressions

- A truth table for an expression of $n$ variables has $n$ initial columns
- the remaining columns are for specific expressions
- For $n$ (binary) variables, how many rows will we have? ($2^n$)
- Once we have a truth table for an expression, it’s easy to determine logical equivalence of two expressions
• The truth tables are the same!

### Bitstrings

• Let’s be explicit about how we manipulate strings of bits
• The positions of bits are labeled from 0 to n-1 for an n-bit string (reading from right to left)
• For example, if \( X = 0101 \), \( X[0] = 1; X[1] = 0; X[2] = 1; X[3] = 0 \)
• We can specify ranges (substrings) like so \( X[2:1] \). This is inclusive, \( X[2:1] = 10 \)
• We have a few more basic operations
• Let’s state our goals this way: given an arbitrary bit string, we want to be able to know whether or not bit 6 is 1. We’ll need a few operations for that

**Shifting**: We drop bits on left and right, and fill up the holes on left, right
• Example: 1100 left shifted by one is 1000 (or perhaps 1001, depending on the shifting scheme you invent!)
• These are just bitstrings, but what if we interpret them as numbers? What does shifting mean? (division/multiplication by 2)
• What if it’s a negative number? What should our policy be? We want the leading 1s preserved on a right shift correct?
• So we’ll just invent a scheme that fills the holes with 1s instead of 0s. This is called arithmetic shift. Zero fill is called *logical shift*
• Left shifting does not matter whether signed or unsigned, so there’s only zero fill
• Shifting in C is done with the “\( >> \)” operator
• C will pick the right one for you based on the type (e.g. int or unsigned int)

**Bitwise Operators:**
• How do I flip all the bits? Use a bitwise NOT. (in C, the ~ operator, e.g. \( \sim x \)). For \( x[i] = \sim x[i] \)
• NOTE: this is how you can explicitly take 1s complement!
• Bitwise AND: for all \( x[i] \) in \( x \) and \( y[i] \) in \( y \), \( x \& y = x[i] \& y[i] \). In C, represented with ‘\&’
• Similarly, we have bitwise OR and bitwise XOR. In C, ‘|’ and ‘^’ respectively

**Logical Operators:**
• C also has \&\&, ||, !, and != (logical and, or, not, xor, respectively)
• in C, integer 0 represent false (0), and any nonzero integer represents true (1).
• \( x \&\& y \) is true if \( x = 27 \) and \( y = -32 \)
• ! (x \&\& y) is 0
• !!(x \&\& y) is 1
Manipulating Bits

- Back to our question, how do we get bit 6 (whether its 0 or 1?)
- We can use shifts first. Right shift by 6. Bit 6 is now in bit 0’s former position.
- Now all we have to do is bitwise and with 1. If the result is 1, that means bit 6 was set to 1. Otherwise, it wasn’t
- In general, we can use Bit Masks, strings of bits used to help isolate parts of bitstrings
- We’ll rely on some properties: (assume b is a bit)

<table>
<thead>
<tr>
<th></th>
<th>b ^ X</th>
<th>b v X</th>
<th>b ⊕ X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>1</td>
<td>~b</td>
</tr>
</tbody>
</table>

- OK, so given a bit string X, and we want to inspect bit k (X[k]), how do we do it?
- We saw one technique, right shift by k, & with 1
- Another technique, build a mask. The mask has all 0s except in position k. How do we construct such a mask?
- Well, bit k is just 2^k. So we can say our mask M = 2^k. We can then use X & M = S. If S is 0 then bit k was 0, otherwise bit k was 1
- How do we get 2k? In C, we can just left shift 1 by k, e.g. (1 << k)
- How do we set a bit to 1?
- We use the same technique to get an isolated 1 in position k (1 << k)
- Instead of using AND, we can use OR. S = X | M where M again is constructed with (1 << k)
- How do we clear a bit? (AND with ~(1<<k))
- How do we flip a bit? (XOR with (1<<k))

- What about for multiple bits?
- Here’s where the mask really comes in handy. We need to construct a mask with k bits set to 1 (we don’t care about where they are now)
- Let’s just assume they’ll be in bit positions 0 through k-1
- So, for 8 bits and k=4, we’d like to have 00001111
- We really only know how to set the fourth bit though! (1 << k) or (1 << 4). Could we somehow get our string from that?
- Do the algebra! 2^k + x = 00001111 or 00010000 + x = 00001111
- Turns out x here is -1. To build a full mask in bits 0 through k-1, we can just use 2^k - 1 or (1<<k) - 1
- Say we want to flip the top four bits of some bit string. Can we use shifts? (yes, with xor)
- In general: X & M gives us the last k bits
- X <- X | M sets the last k bits to 1
- X <- X & ~M clears the last k bits to 0
- X <- X ^ M flips the last k bits
NOTE: I will post some example C code for bit shifts and bitwise operations and masks