CPSC3300 – Computer Systems Organization  
Homework #2 – Boolean Algebra and Adders

Due: 11:59PM Monday, February 10

Submit to Canvas

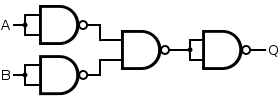
Total 100pts

1. [20pts] Logical completeness
   1. Show that you can use only two-input NAND gates to implement each of the following two-input logic functions, and draw the used NAND gates and wiring.
      1. NOR function

=NOT ()

We know

So: we can get the gates as follows:



* + 1. XOR function

=

= (A. NAND NAND (NAND B)

To reduce one NAND gate, we can simplify it as:

Since:

A NAND = A NAND (B NAND B)

= (A NAND B) NAND B

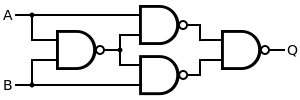
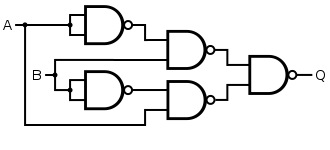
NAND B = (A NAND A) NAND B

= A NAND (A NAND B)

Thus:

(A. NAND NAND (NAND B) =

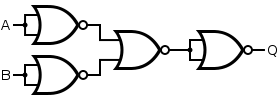
(A. NAND NAND (NAND B)

 or 

* 1. Show that you can use only two-input NOR gates to implement each of the following two-input logic functions, and draw the used NOR gates and wiring.
     1. NAND function

A NOR A =

B NOR B =



* + 1. XOR function

=

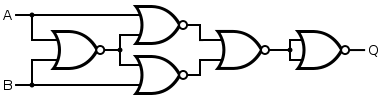
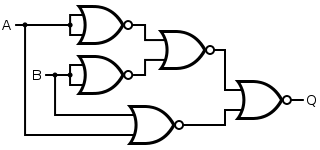
= = NOR B

= =A NOR

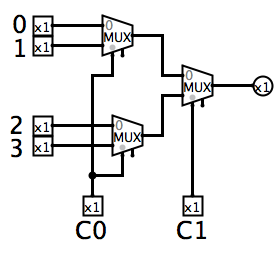
To reduce one NOR gate, we can simplify it as:

NOR B = A NOR A NOR B = A NOR ( A NOR B)

A NOR = A NOR B NOR B = ( A NOR B) NOR B

 or 

1. [10pts] Show how to use 2-1 Muxes to build a 4-1 Mux. Draw the used 2-1 Muxes and the wiring, and mark the 4 inputs and 1 output for the resulting 4-1 Mux.



1. [10pts] Demonstrate by means of truth tables whether the following identities are valid or not:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | | |  | **Output** |
| *A* | *B* | *C* |  |  |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Input** | | |  |  |  | **Output** |
| **x** | **y** | **z** |  |  |  |  |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |

conclusion: NOT TRUE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | | |  | **Output** |
| *A* | *B* | *C* |  |  |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | | |  |  | **Output** |
| *A* | *B* | *C* |  |  |  |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |

conclusion: TRUE

1. [20pts] Prove the identity of each of the following Boolean equations, using algebraic manipulation:

=

=

=

=

= 1

1. [20pts] For the Boolean function O1 and O2, as given in the following truth table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | | | **Output** | |
| **x** | **y** | **z** | **O1** | **O2** |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

* 1. List the minterms for a three-variable function with variables x, y, and z.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input** | | | **minterm** | | | | | | | |
| **x** | **y** | **z** |  |  |  |  |  |  |  |  |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

* 1. Express O1 and O2 in sum-of-product algebraic form.

O1 =  **+**  + ++

O2 **=**  + + +

1. [20pts] In class, we learned the implementation for a 4-bit carry lookahead adder. We can use the same idea and extend to build a 16-bit carry lookahead adder. Denote this implementation as a one-level carry lookahead adder.

In the textbook, Figure 8.6.3 shows a two-level implementation of a 16-bit carry lookahead adder. This adder uses 4-bit carry lookahead adders at the lower level, and uses a carry lookahead unit at the higher level.

Compare these two implementations and provide your explanation why the two-level implementation could be preferred.

The one-level 16-bit carry-lookahead adder using a single 16-bit carry-lookahead unit has the fewest gate delays, however due to the nature that each higher bit requires the propagate and generate signals from all lower bits, there is an increasing number of gates and wider gates used with each additional bit. This greatly increases the transistor count and is impractical as well as inefficient to implement in hardware. Alternatively having four 4-bit carry-lookahead adders in series with the highest carry bit being the carry-in bit for the next higher 4-bit CLA will require much less hardware but will be considerably slower.

The two-level 16-bit carry-lookahead adder introduces an additional 4-bit carry lookahead stage that uses a “super” propagate and “super” generate signal from the four 4-bit carry-lookahead adders at the first stage for the input to generate the carry-in signals for the three higher first-stage carry-lookahead adders. This extra second-stage carry-lookahead unit that uses “super” propagates and “super” generates saves time with respect to having the four 4-bit carry-lookahead adders in series (carry-lookahead “ripple adder”).

The two-level implementation could be preferred as it is more efficient to implement in hardware than a single 16-bit carry-lookahead unit. As far as 16-bit adders with a lower gate count (and number of transistors), the two-level carry-lookahead adder is considerably faster than using the four 4-bit carry-lookahead adders in series with the highest carry bit being the carry-in bit for the next higher 4-bit CLA.

The reason carry lookahead can make carries faster is that all logic begins evaluating the moment the clock cycle begins, and the result will not change once the output of each gate stops changing. By taking the shortcut of going through fewer gates to send the carry in signal, the output of the gates will stop changing sooner, and hence the time for the adder can be less.