CS 16 — Fall 2010 — Mid-term exam

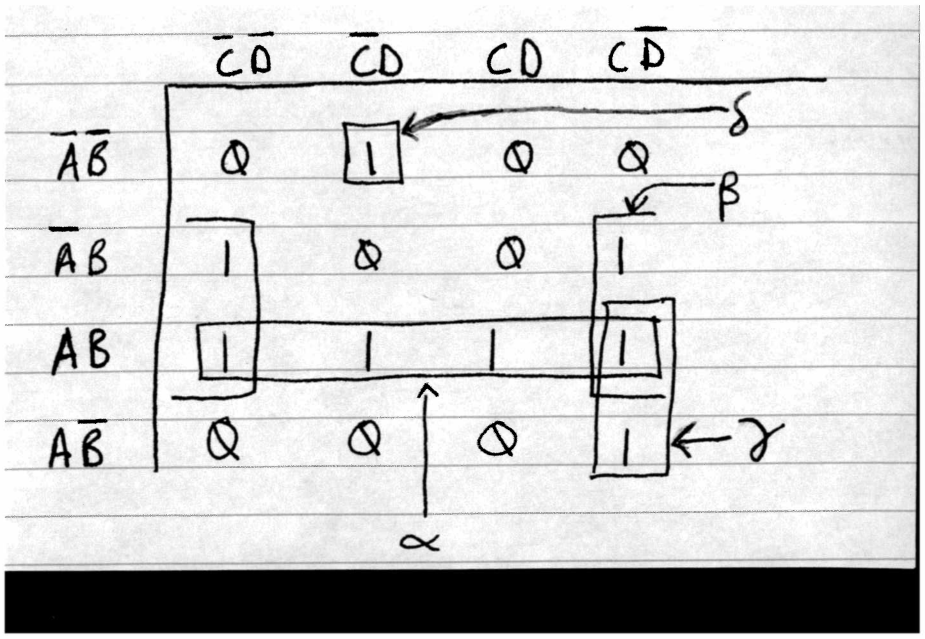
This is a closed-book, closed-note exam. Answer all of the questions **clearly, completely, and concisely**. You have 50 minutes, so be sure to use your time wisely. All work should be written in your blue book. If a question's intent is ambiguous, then **make reasonable assumptions that you can justify, and write them with your answer**.

1. Use a Karnaugh map to simplify the following boolean function:

$$Y = \bar{A}\bar{B}\bar{C}D + \bar{A}B\bar{C}\bar{D} + \bar{A}BC\bar{D} + A\bar{B}C\bar{D} + AB\bar{C}\bar{D} + AB\bar{C}D + ABC\bar{D} + ABCD$$

Draw your rectangles clearly and **express your result as a boolean algebraic equation**—*do not draw a circuit.*

Answer: Below is the Karnaugh map for this function, with rectangles identifying terms that can be simplified:



The rectangles simplify to the following conjunctive terms:

$$\alpha = AB$$

$$\beta = B\bar{D}$$

$$\gamma = AC\bar{D}$$

$$\delta = \bar{A}\bar{B}\bar{C}D$$

Finally, the final, simplified function is formed by disjoining those conjunctive terms:

$$Y = AB + B\bar{D} + AC\bar{D} + \bar{A}\bar{B}\bar{C}D$$

DISCUSSION: Transcription errors were somewhat common, leaving people simplifying the wrong function. What was far more common was failing to see the potential for a second 4-element rectangle. Specifically, everyone found α, but many missed the 2-by-2 rectangle for β, largely because half of that rectangle was already covered by α. However, failing to treat it as a 4-element rectangle left the term $\bar{A}B\bar{D}$, but missing the opportunity to remove $\bar{A}$ from the term.

2. Recall the following feedback function that we developed in class, when first devising a function capable of "remembering" a value, such that its output Q would be **set** ($Q = 1$ when $S = 1$), **reset** ($Q = 0$ when $R = 1$), or retain its old value (Q is whatever it was):

$$Q = \bar{R}S + Q\bar{R}\bar{S}$$

I then claimed that this function was equivalent to a pair of cross-coupled NOR gates. **Show, algebraically, that the function above is equivalent to the cross-coupled NOR gates**.

ANSWER: Starting from the cross-coupled NOR gates, we see that the circuit's outputs can be defined (recursively) as:

$$Q = \overline{R + \bar{Q}}$$

$$\bar{Q} = \overline{S + Q}$$

By substituting the latter into the former, we get:

$$Q = \overline{R + (\overline{S + Q})}$$

The following sequence of algebraic steps transforms this function into the given one, above:

$$Q = \bar{R}(\overline{\overline{S + Q}})$$

$$Q = \bar{R}(S + Q)$$

$$Q = \bar{R}(S + Q(\bar{S} + S))$$

(since $\bar{S} + S = 1$)

$$Q = \bar{R}(S + Q\bar{S} + QS)$$

$$Q = \bar{R}(S(1 + Q) + Q\bar{S})$$

$$Q = \bar{R}(S + Q\bar{S})$$

$$Q = \bar{R}S + Q\bar{R}\bar{S}$$

DISCUSSION: There is more than one way to skin this cat, and a number of people came up with different algebraic sequences to show the equivalence. I gave most of the credit if you could get far enough to determine that the essential trick is showing that $S + Q = S + Q\bar{S}$. However, use of truth tables got no credit here, and that was the most common error/mistake/failing. How much credit you lost depended on how deeply you relied on a truth table to do the work of showing equivalence.

3. Consider representing 4-bit signed integers by using *excess-8 notation.* For this notation, all integers are represented by a value that is "in excess" of its true, signed value by 8 positions on the number line. For example, if you see the binary value 1100, the unsigned integer interpretation of that value, in decimal, is 12. However, if it is an excess-8 value, then it is really representing $12 - 8 = 4$. Likewise, $0010 = 2 - 8 = -6$ in excess-8 notation.
Given this form of signed integer representation, **answer the following questions**:
   (a) What is the *range* of values that can be represented?
   (b) Does additon of two excess-8 values "just work"? If not, do you see how to make it work?
   (c) Are there any (other) drawback that make this notation undesirable?

   ANSWER:
   (a) Since the range of unsigned 4-bit integer values is from 0 to 15, then the *range* for 4-bit excess-8 values is from $0 - 8 = -8$ to $15 - 8 = 7$.
   (b) Adding two excess-8 values in the same manner as two unsigned integers does **not** yield the correct result. For example, consider $-4 + 6$. In excess-8 notation, these values are $0100(4 - 8 = -4) + 1110(14 - 8 = 6)$. If we add these two binary values as unsigned integers, we get the 5-bit result 10010, which predictably represents the unsigned sum $4 + 14 = 18$. However, as a 4-bit excess-8 value, the lower 4 bits 0010 represents $2 - 8 = -6$, which is incorrect.
   The fix, however, is quite simple: *Invert the most significant bit of the result.* Thus, in our example, 0010 becomes 1010, which is the unsigned value of 10, or, in excess-8, $10 - 8 = 2$—the correct answer!
   (c) There are a few meaningful drawbacks to this signed integer represenation: First, non-negative values begin with a 1 while negative ones begin with a 0, making the representation of non-negative values fail to match their unsigned counterparts. Second, as a result, the important value 0 is not a collection of four bits whose value is 0, but rather the special pattern 1000. Thus, zero is uniquely represented by one bit-pattern, but it's not the most desirable pattern.

   DISCUSSION: Most everyone correctly identified the range and observed that adding did not "just work." However, many people observed that you need only "add 8" to correct the results of straightforward addition. This answer, by itself, presents a problem of circularity: To get *addition* to work, we need only to perform an extra *addition* operation. Worse, the suggestion invariably failed to state that what was needed was to "add an *unsigned* 8." Those who went on to formulate the correct mechanism for obtaining this result—flipping the most significant bit—got credit; without that observation (or at least some reasoning leading toward that observation), little or no credit was earned.
   The answers for drawbacks were (predictably) all over the map, some earning more credit than others. Many offered vague complaints about *overflow*, failing to notice that (a) overflow is a problem for **any** fixed-sized numeric representation, signed or unsigned, and that (b) overflow is actually quite easy to detect in excess-8 addition. (Do you see how?)
   Another common and erroneous mistake was asserting that extending this notation to larger numbers of bits would be difficult or limiting. This observation is false: Any k-bit signed integer can be represented using excess-2^{k-1} notation. So, 5-bit values would use excess-16, 6-bit would use excess-32, and so on.

Finally, some worried that negating a value would be complex in this representation. However, interestingly, negating an excess-8 value requires two familiar steps: (a) flip the bits, and (b) add 1. Sound familiar?

4. Provide a **complete** argument that the NOR logic operator is *universal.* That is, show first that NOT, AND, and OR are sufficient to express **any** logic function; then show that NOR is sufficient to express all three of those.

   ANSWER: First, we must establish the universality of the trio of operators AND/OR/NOT. To establish that universality, we observe that any logic function can be expressed using a truth table. For each entry in the truth table for which the output of the function is 1/TRUE, the corresponding combination of input values can be straightforwardly translated into on *conjunctive term* that is then *disjoined* with the other terms to form a new function in the *disjunctive normal form.* Thus, any logic function can be mechanically translated into *DNF*, which itself uses only AND/OR/NOT operators.
   Second, we can establish that NOR is sufficient to compute each of the AND/OR/NOT operator trio. Specifically:

$$\bar{x} = \overline{x + x}$$

$$x + y = \overline{\overline{x + y}} = \overline{\overline{x + y} + \overline{x + y}}$$

$$xy = \overline{\overline{xy}} = \overline{\bar{x} + \bar{y}} = \overline{\overline{x + x} + \overline{y + y}}$$

   Thus, with simple algebraic transformations (and one use of DeMorgan's Law), each of the trio can be expressed using nothing but NOR operations, thus demonstrating its universality.

   DISCUSSION: Many successfully completed the second portion of this problem by showing the algebra involved and, often, the circuitry that would follow. Either was acceptable in demonstrating the universality of NOR. However, many also struggled to show the universality of AND/OR/NOT, simply claiming that this trio was universal by some declaration of being "fundamental" without justifying the claim nor defining what was fundamental about them. Others attempted to skip any need to argue for the universality of AND/OR/NOT by simply showing how NOR can be used to calculate NAND, and thus relying on our work in class to show that NAND is universal. However, this approach ignores that NAND was only shown to be universal given the assumption that AND/OR/NOT are also universal. Thus, showing the equivalence of NAND is insufficient, since it still requires an explicit demonstration that AND/OR/NOT are universal.

5. **Take home problem to be submitted by Thursday at 5:00 pm:** Is XOR a universal operator? If so, show it; if not, explain/demonstrate/argue/prove why not.

   ANSWER: It cannot be universal. To express AND or OR, one must be able to devise functions whose result for **one** of the input patterns is different from the other **three**. (For example, for AND, only two TRUE inputs yield a TRUE output; the other three input combinations yield FALSE outputs.)
   However, XOR provides one result for **two** of its input patterns, and the other result for the other two. No combination of XOR operations—no composition of XOR with itself—can alter this grouping of two-falses-and-two-trues for the final output. Thus, XOR can be used to perform NOT operations, but no arrangement can yield AND or OR.

   DISCUSSION: Everyone concluded, correctly, that XOR is not a universal operator. Many of you then presented a variety of arguments that were sufficient. Some exhaustively expressed all of the possible combinations of using XOR with the two variables provided as well as the two possible constants, drawing reasonable conclusions about the impossibility of composing XOR with itself successfully.
   There were two prominent errors in justifying the non-universality of this function:

   - Some attempted some non-comprehensive recombinations of XOR with itself, and when they could not construct either AND or OR, they concluded that it couldn't be done. Although the conclusion is correct, the cases and arguments shown were insufficient for drawing that conclusion.
   - Many claimed that XOR could not be used to perform an inversion (that is, the NOT operator). This claim is incorrect: $\bar{x} = x \oplus 1$. This faulty conclusion was most commonly reached by failing to consider the use of constants as inputs.