

# Fundamentals of Digital Logic Design

ECE/CS 3700

Spring 2007, Solutions to Homework # 4

1. (15 points) **Propagate-generate-delete signals in adders.** In class, we studied the carry look-ahead adder as a means to speed-up carry propagation delay of the ripple carry adder. Look-ahead adders make use of the generate ( $G_i$ ) and propagate ( $P_i$ ) signals to *precompute* whether or not stage  $i$  would output a carry. Similar to using the propagate ( $P_i$ ) and generate ( $G_i$ ) signals, a look-ahead adder can be designed by using a propagate ( $P_i$ ), generate ( $G_i$ ) and delete ( $D_i$ ) signal. Consider the Truth table of the full adder shown in Table I:

TABLE I  
TRUTH TABLE OF A FULL ADDER

| $a$ | $b$ | $c_i$ | $S$ | $c_{i+1}$ | Carry Status |
|-----|-----|-------|-----|-----------|--------------|
| 0   | 0   | 0     | 0   | 0         | Delete       |
| 0   | 0   | 1     | 1   | 0         | Delete       |
| 0   | 1   | 0     | 1   | 0         | Propagate    |
| 0   | 1   | 1     | 0   | 1         | Propagate    |
| 1   | 0   | 0     | 1   | 0         | Propagate    |
| 1   | 0   | 1     | 0   | 1         | Propagate    |
| 1   | 1   | 0     | 0   | 1         | Generate     |
| 1   | 1   | 1     | 1   | 1         | Generate     |

The first two minterms  $m_0, m_1$  correspond to the condition where the carry-out signal gets suppressed (deleted) at  $c_{i+1}$ , independent of the value at  $c_i$ . Prove that:

- The Delete signal  $D_i = a' \cdot b'$ .

**Solution:** Note that  $D_i$  corresponds to the first two minterms. So,  $D_i = a'b'c' + a'b'c = a'b'$ .

- $Sum = P_i \cdot \overline{C_i} + D_i \cdot C_i + G_i \cdot C_i$

**Solution:** From the truth table: i)  $Sum = C_i$  when  $D_i = 1$  or  $G_i = 1$ ; and ii)  $Sum = \overline{C_i}$  when  $P_i = 1$ .

Mathematically:

$$S = C_i(G_i + D_i) + P_i\overline{C_i} \quad (1)$$

- $C_{i+1} = G_i + \overline{D_i} \cdot C_i$ .

**Solution:** Note that  $\overline{D_i} = P_i + G_i$ ; i.e.  $\overline{D_i}$  is the entire function except for the first two points. Moreover we already know that  $C_{i+1} = G_i + P_iC_i$ . So we have to prove that  $C_{i+1} = G_i + \overline{D_i} \cdot C_i = G_i + (G_i + P_i)C_i = G_i + G_iC_i + P_iC_i = G_i + P_iC_i$  which we already know  $= C_{i+1}$ .

2. (10 points) Given that  $A, B, C_i$  are the inputs to a full adder,  $S$  and  $C_o$  are sum and carry-out, respectively, prove that:

- $S = ABC_i + \overline{C_o}(A + B + C_i)$ .

**Solution:** We know that  $C_o$  = majority function of  $A, B, C_i$ . So  $\overline{C_o} = \overline{AB + BC_i + AC_i}$ . Note, in the first mid-term exam, we have already shown that this is  $\overline{C_o} = A'B' + A'C'_i + B'C'_i$ . Moreover, from the truth table, we see that  $S = A'B'C + A'BC' + AB'C' + ABC$ . So substitute  $\overline{C_o}$  in the above equation and we get:  $S = ABC + (A'B' + A'C' + B'C')(A + B + C) = ABC + A'B'C + A'BC' + AB'C' = S$ .

3. (25 points) **Multiplier design.** You are asked to design an array multiplier that multiplies a 4-bit number  $A = (a_3, a_2, a_1, a_0)$  by a 3-bit number  $B = (b_2, b_1, b_0)$ . Consider that pre-designed 4-bit adders are available to you. (Recall that a 4-bit adder adds two four-bit numbers). Design the multiplier using only **two 4-bit adders** and a minimum number of two-input AND/OR/NOT/XOR/XNOR gates. Show a block diagram or a schematic of your design depicting input and output bits clearly. (Solve this problem properly, and you've understood the concept of array multipliers!).

**Solution:** The Schematic of the design is attached as a separate sheet (file) on the webpage. Note how AND gates are used to generate all the bit-products and they're just shifted-and-added as shown.

4. (15 points) **Two's complement numbers.** Suppose that you are given two **3-bit unsigned** numbers  $A[2 : 0], B[2 : 0]$  that have to be added together ( $C = A + B$ ). Suppose, further, that you are given a pre-designed *4-bit adder* that you have to use for this purpose. A 4-bit adder takes inputs  $X[3 : 0], Y[3 : 0]$  and adds them. In order to do the addition correctly, we can take vectors  $A, B$  and concatenate a leading 0 to make them 4-bit vectors and then map the inputs; i.e. in Verilog terms:  $X[3 : 0] = \{1'b0, A[2 : 0]\}$ ; and similarly  $Y[3 : 0] = \{1'b0, B[2 : 0]\}$ . This way,  $n$ -bit unsigned integers can be scaled to larger bits.

However, this technique may not work for 2's complement scheme. So, now you have to answer the following: You are given two 3-bit vectors  $A[2 : 0], B[2 : 0]$  that are already given in **3-bit two's complement form**. You are asked to subtract  $A - B$ . Suppose that you are already given a *4-bit subtractor* (say, the design of Fig. 5.13, pp. 248 in the textbook) that takes  $X[3 : 0], Y[3 : 0]$  and computes  $C = X - Y$ , where  $C$  is a 4-bit two's complement number. You have to use this 4-bit subtractor to subtract the given 3-bit numbers  $A, B$ . How will you scale (or modify) the inputs correspondingly? Show your design/schematic (or a Verilog code, if you wish), and demonstrate the correct functioning of your circuit using an example.

**Solution:** This problem is one of "sign-extension" in two's complement numbers. Whenever we have to scale a 3-bit "positive" number to a 4-bit (or larger) positive number, we just append leading zeros. This way, the number remains positive, and its magnitude doesn't change. For example: 011 in 3-bit two's complement form is +3. If we scale it to 4-bit two's complement, we get 0011 (note the extra leading zero) which is still a positive number and its magnitude does not change.

However, if we take  $X = 111$  in 3-bit two's complement, then  $X = -1$ . If we append a leading zero to  $X$  to make it a 4-bit number, 0111 will become +7, which is wrong. However, if we append 1 to  $X$ , we have 1111 = -1. Therefore,

the concept of sign-extension in two's complement bit-vector scaling is just this: i) Take the most significant bit of vector  $X$ . If it is 0,  $X = +ve$ , else negative. ii) Just append the vector with the sign bits, as many as required.

Another example:  $101 = -3$  in 3-bit two's complement.  $1101$  is also  $-3$ , but in 4-bit two's complement. Similarly,  $111101 = -3$  in 6-bit two's complement!

5. (10 points) **Decoders.** Solve problem 6.1, pp. 343, from the textbook.

**Solution:** Note that  $F = 1$  except for two minterms, minterm 1 and minterm 6. The design is given below in Fig. 1.

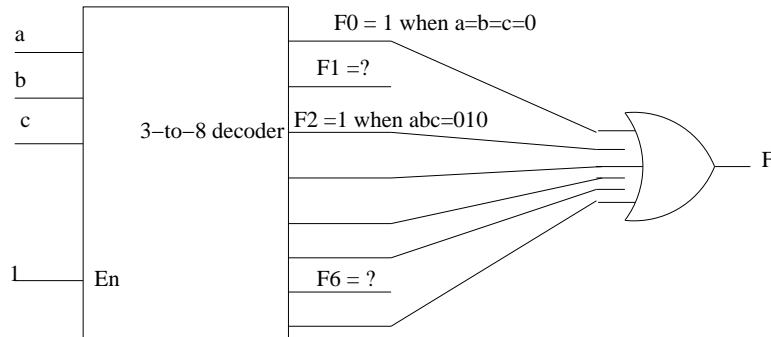


Fig. 1. Design  $F = m(0, 2, 3, 4, 5, 7)$  using a 3-to-8 decoder

Note, when  $(a, b, c) = (0, 0, 1)$  the second output of the decoder  $F1 = 1$ , all other outputs are zero. However, since  $F1$  is not connected to the OR gate, it will not affect the output!!! Same with  $F6$ .

6. (10 points) **Shannon's Expansion and MUXes.** Solve problem 6.5, pp. 343, from the textbook.

**Solution:** For ease of notation, I'm referring to  $w_1 = a, w_2 = b, w_3 = c$ . So we have to design  $f(a, b, c) = m(0, 2, 3, 6) = a'b'c' + a'bc' + a'bc + abc'$  using Shannon's expansion and MUXes. We've seen how MUX decomposition is a direct application of Shannon's expansion!

Let us decompose  $f$  w.r.t. the first variable  $a$ . So,  $f_a = f(a = 1) = 0 + 0 + 0 + bc' = bc'$ . Now let's compute  $f_{a'} = f(a = 0) = b'c' + bc' + bc = b'c' + b = b + c'$ . So, our design is shown below in Fig. 2 for  $f = a \cdot f(a = 1) + a' \cdot f(a = 0)$ . So,  $a$  becomes the control signal of the MUX.

Of course, if you do the decomposition using a different variable, (say,  $b$ ) you'll get a different decomposition.

7. (15 points) **A practical example of Code-Converters.** Braille is a system of raised dots that can be read by a blind person. You are asked to design an *encoder* circuit that converts Binary Coded Decimal (BCD) numbers to Braille. The Braille patterns for the BCD numbers are shown below in Fig. 3.

Derive a minimum sum-of-product form representation for each of the four Braille dot outputs  $X, Y, W, Z$  in terms of a 4-bit BCD number. Denote the 4-bit BCD number as  $ABCD$  where  $A$  is the most significant bit, and  $D$  the least significant bit.

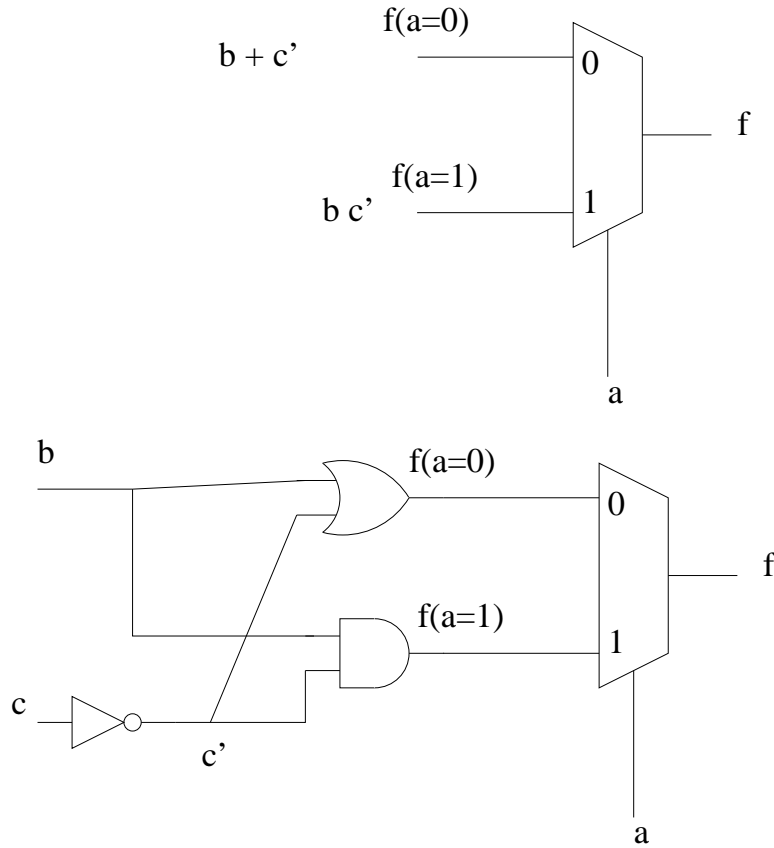


Fig. 2. Design  $F = m(0, 2, 3, 6)$  using a MUX

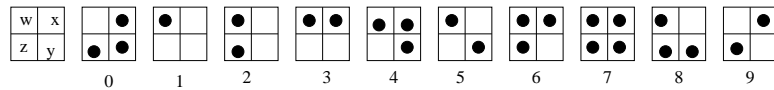


Fig. 3. Braille representation of digits 0 to 9.

**Solution:** The solution is uploaded on the webpage using a different PDF file. Take a look at the truth table and the K-maps and notice the use of don't cares. Actually, Code Converters make a lot of use of Don't care based simplification. This happens because some codes are valid ("care points" in the design space) but some are invalid and should not occur ("don't cares"). In our example, we don't have a separate 1-digit code for 4-bit BCD numbers from 10 to 15 (they are 2-BCD-digit codes!!). Hence the use of Don't cares in simplification.