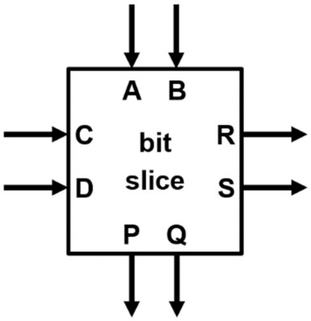
ECE 120: Introduction to Computing Spring 2019 ZJUI

due at START OF LECTURE on Thursday 4 April 2019

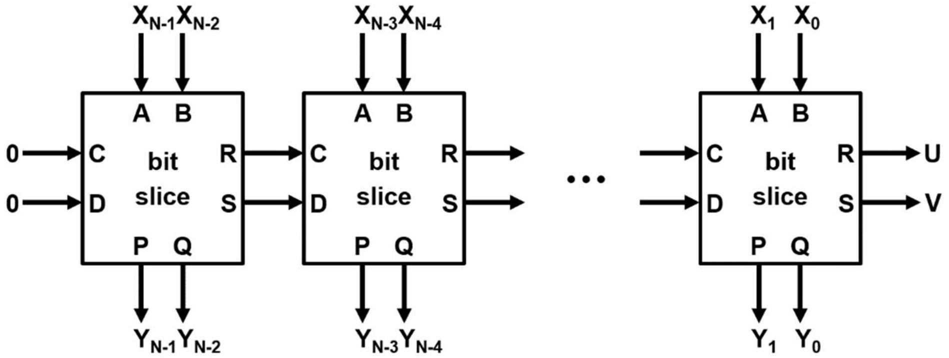
Homework 6: Bit Slicing and Abstraction

1. Bit Slice Analysis

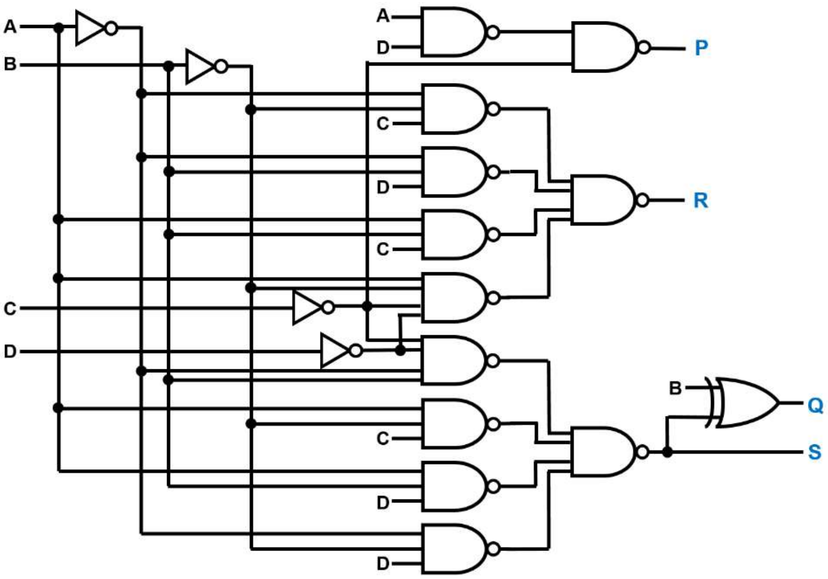


Prof. Lumetta has done it again. He designed a bit-sliced circuit for ECE120, then forgot what the circuit does! Please help him to figure it out. The interface to a single bit slice is shown to the right. Each bit slice consumes two bits of an unsigned number (into inputs A and B) and produces two bits of an unsigned number (from outputs P and Q). Two bits are also passed in from the previous bit slice (more significant bits) as inputs C and D, and two bits are passed out to the next bit slice (less significant bits) as inputs R and S.

For an N-bit unsigned number $X=X_{N-1}X_{N-2}…X_1X_0$, N/2 bit slices are hooked together as shown below to produce unsigned number $Y=Y_{N-1}Y_{N-2}…Y_1Y_0$ and outputs U and V.



A circuit diagram for the bit slice appears to the right.



a. Start by writing a truth table for outputs P, Q, R, and S in terms of inputs A, B, C, and D. Be sure to get the right answer—double-check your work! If you make mistakes here, you will have difficulty understanding the purpose of the circuit.

A	B	C	D	P	Q	R	S
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	1
0	0	1	0	1	0	1	0
0	0	1	1	1	1	1	1
0	1	0	0	0	0	0	1
0	1	0	1	0	1	1	0
0	1	1	0	1	1	0	0
0	1	1	1	1	1	1	0
1	0	0	0	0	0	1	0
1	0	0	1	1	0	0	0
1	0	1	0	1	1	0	1
1	0	1	1	1	1	0	1
1	1	0	0	0	1	0	0
1	1	0	1	1	0	0	1
1	1	1	0	1	1	1	0
1	1	1	1	1	0	1	1

b. Use your truth table from **part (a)** to fill in the table of all bit slice outputs when four copies of the bit slice are connected together and X=01000011.

	A	B	C	D	P	Q	R	S
bit slice for X_7X_6	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**
bit slice for X_5X_4	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**
bit slice for X_3X_2	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**
bit slice for X_1X_0	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**1**

From the table, write the bits corresponding to outputs Y (8 bits), U (1 bit), and V (1 bit). Finally, write the decimal values represented by both X and Y in the 8-bit unsigned representation.

Y = 00010110, U = 0, V = 1, X = 67_{10}, and Y = 22_{10}.

c. Repeat **part (b)** with input X=01111010.

	A	B	C	D	P	Q	R	S
bit slice for X_7X_6	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**
bit slice for X_5X_4	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**1**
bit slice for X_3X_2	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**0**
bit slice for X_1X_0	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**

Y = 00101000, U = 1, V = 0, X = 122_{10}, and Y = 40_{10}.

d. Repeat **part (b)** with input X=0101110111 and five copies of the bit slice.

	A	B	C	D	P	Q	R	S
bit slice for X_9X_8	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**
bit slice for X_7X_6	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**0**
bit slice for X_5X_4	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**0**
bit slice for X_3X_2	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**0**
bit slice for X_1X_0	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**0**

Y = 0001111101, U = 0, V = 0, X = 375_{10}, and Y = 125_{10}.

e. Based on your results from **parts (b)**, **(c)**, and **(d)**, explain how output Y relates to input X.
Y is equal to X/3 rounded down.

f. Based on your results from **parts (b)**, **(c)**, and **(d)**, explain how the outputs UV (two bits) relate to input X.
UV is the remainder of X divided by 3.

g. If we instead choose to think of the bits of X as representing a number in 2's complement, does the design still operate as intended (your explanation in **parts (e)** and **(f)**)? Explain your answer.

No. For example, from the truth table, we know that if ABCD=1100, P and Q are 0 and 1. That means, if we have an X starting with 11 (a negative value), the result will be positive (starting with 01). In fact, Y can never begin with 1, since CD=00 for the first bit slice, and none of the four rows with CD=00 produces P=1.

h. Let's analyze the delay of the bit slice. Fill in the table below with the number of gate delays from each input to each output. Remember that we use the longest path (the one with the most gates) for such measurements. If an input has no effect on a particular output, write "N/A" in the corresponding box in the table. Refer to the bottom of p. 55 of the notes for an example of analysis on the comparator developed in Section 2.4.

		Outputs			
		P	Q	R	S
inputs	A	**2**	**4**	**3**	**3**
	B	**N/A**	**4**	**3**	**3**
	C	**2**	**4**	**3**	**3**
	D	**2**	**4**	**3**	**3**

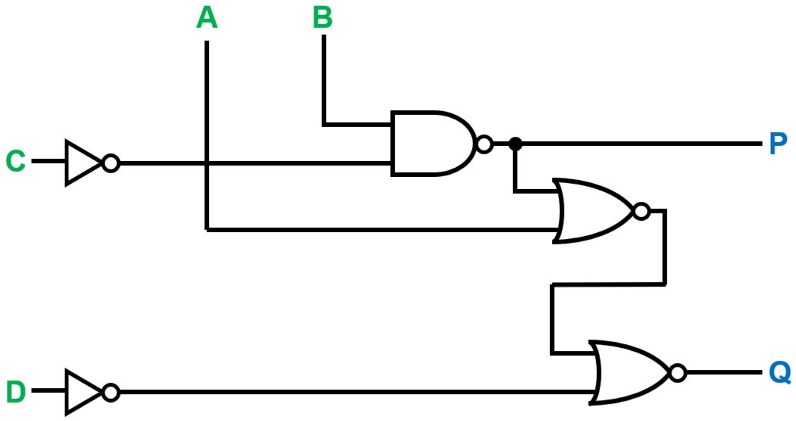
i. Now calculate the number of gate delays needed to perform a computation using N/2 copies of the bit slice on an N-bit unsigned number. Assume that N is even. Assuming that all bits of A and B arrive together, how many gate delays are needed before the last bit slice produces its last output?

Delay from C and D to R and S is more important when it comes to N/2 bit slices. Total delay until the last output (Q) is 3N/2 + 1 gate delays.

2. Debugging a Bit-Sliced Comparator

A friend in ECE120 has designed a bit-sliced comparator for unsigned numbers. The circuit below illustrates a single bit slice. The bits of the numbers to be compared are fed into the A and B inputs. Two additional input bits, C and D, are provided by the bit slice that handles the next most significant bits. In other words, information flows from the most significant to the least significant bits, in the opposite direction as the design in Section 2.4 of the notes. Each bit slice produces outputs P and Q, which are passed to the next bit slice (as inputs C and D, respectively). The representation used for the bits passed between bit slices is as follows: 00 and 10 (both patterns) mean that A < B; 11 means that A > B; and 01 means that A = B. The first bit slice, corresponding to the most significant bit of the numbers being compared, has inputs CD=01 (which means A=B).

Unfortunately, your friend has made a mistake and needs your help to correct the design.



a. Begin by analyzing the circuit to produce a truth table for outputs P and Q as a function of A, B, C, and D.

P = (BC')' = B' + C

Q = (P + A)D (POS form using NOR-NOR)

= (B' + C + A)D

A	B	C	D	P	Q
0	0	0	0	1	0
0	0	0	1	1	1
0	0	1	0	1	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	1	0
0	1	1	1	1	1
1	0	0	0	1	0
1	0	0	1	1	1
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0
1	1	1	1	1	1

b. Copy the bits from your truth table in **part (a)** into two K-maps, one for P, and a second for Q.

P		AB			
		00	01	11	10
	00	1	0	0	1
	01	1	0	0	1
CD	11	1	1	1	1
	10	1	1	1	1

Q		AB			
		00	01	11	10
	00	0	0	0	0
	01	1	0	1	1
CD	11	1	1	1	1
	10	0	0	0	0

c. Using the representation for CD and either your truth table for **part (a)** or your K-maps from **part (b)**, identify the input combination that leads to an incorrect output pattern. There is only one such input combination.
When ABCD=0001, the information from the previous bit slice is "A=B" (CD=01), and the current bits are the same (AB=00), so the output should be "A=B" (again, PQ=01). But in the truth table, PQ=11 in that row.

d. Unfortunately, your friend doesn't believe you. Consider the example A=001000 and B=000100. Using these values as inputs, use your truth table to calculate the value of PQ produced by each of the six bit slices. The final output should be PQ=11, as expected, because A > B.

	A	B	C	D	P	Q
bit slice for A_5B_5	**0**	**0**	**0**	**1**	**1**	**1**
bit slice for A_4B_4	**0**	**0**	**1**	**1**	**1**	**1**
bit slice for A_3B_3	**1**	**0**	**1**	**1**	**1**	**1**
bit slice for A_2B_2	**0**	**1**	**1**	**1**	**1**	**1**
bit slice for A_1B_1	**0**	**0**	**1**	**1**	**1**	**1**
bit slice for A_0B_0	**0**	**0**	**1**	**1**	**1**	**1**

e. Explain why the example in **part (d)** produces a correct answer even though the circuit exercised your so-called "error" From **part (c)**.
Since the error causes the comparator to incorrectly report "A > B" when A = B, comparing numbers such that A > B hides the error. The final answer "A > B" in such cases is indeed correct.

f. Give a counterexample to convince your friend that the design is wrong. In other words, give values of A and B such that the final bit slice's PQ pattern is incorrect, and show the PQ bits produced by each of the bit slices when processing your example.
The answer is wrong whenever A ≤ B and both start with 0 (there are other cases as well). For example, if A is 0011 and B is 0011, the final output should be PQ=01 ("A = B").

For the given comparator:

	A	B	C	D	P	Q
bit slice for A_3B_3	**0**	**0**	**0**	**1**	**1**	**1**
bit slice for A_2B_2	**0**	**0**	**1**	**1**	**1**	**1**
bit slice for A_1B_1	**1**	**1**	**1**	**1**	**1**	**1**
bit slice for A_0B_0	**1**	**1**	**1**	**1**	**1**	**1**

We get output PQ=11, which is incorrect.

g. Explain how to correct the design by giving a correct equation for the incorrect output(s) and indicating which term or factor is missing or has been implemented incorrectly in your friend's design.

The Q value for ABCD=0001 is correct, but the P value should be 0 instead of 1. However, the representation of P allows some flexibility. Specifically, when CD=x0, P=x. So P's K-map is actually:

P		AB			
		00	01	11	10
	00	x	x	x	x
CD	01	0	0	0	1
	11	1	1	1	1
	10	x	x	x	x

We can then write P = C + AB', so we need to replace B's connection to P's NAND gate with a new 2-input NAND gate with inputs A and B'. But then Q needs to be adjusted slightly:
Q = D(P + A + B'), where just the B' term is new (Q before was D(P + A)).

Given the flexibility in P, many answers are possible, of course.

3. Hex Converter

In this problem, you must design circuits to convert ASCII characters representing hexadecimal values into their 4-bit unsigned values. Given a 7-bit ASCII character $C=C_6C_5C_4C_3C_2C_1C_0$, you must produce a 4-bit unsigned number $N=N_3N_2N_1N_0$ and, for part of the problem, a validity signal V indicating that the input character C represents a valid hexadecimal digit.

For reference, ASCII characters x30 through x39 represent digits '0' through '9' and should be converted into unsigned values 0 through 9 (in order). ASCII characters x41 through x46 represent uppercase letters 'A' through 'F' and should be converted into unsigned values 10 through 15 (also in order). Finally, ASCII characters x61 through x66 represent lower-case letters 'a' through 'f' and should also be converted into unsigned values 10 through 15 (again in order). Other ASCII characters do not represent valid hexadecimal digits.

a. Start by finding an expression for V using the approach taken in class to check for upper-case ASCII letters: break the truth table into parts, solve the parts with K-maps, and put the pieces back together with AND and OR. V should be equal to 1 if the ASCII character C represents a valid hexadecimal digit; otherwise, V should be equal to 0. Be sure to choose the better of the SOP and POS solutions for each K-map. After putting the pieces together, a little thought will enable you to simplify your answer slightly. For full credit, your answer must be correct and have area of no more than 22 (use the class' area heuristic).

$V = C_6'C_5C_4T_3 + C_6C_5'C_4'T_4 + C_6C_5C_4'T_6$

But T_4 and T_6 are the same, so $V = C_6'C_5C_4T_3 + C_6C_4'T_4$

T_3 C_1C_0

C_3C_2	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	0	0	0	0
10	1	1	0	0

$T_3 = C_3' + C_2'C_1'$ SOP is better for T_3.

$T_3 = (C_3' + C_2')(C_3' + C_1')$

T_4 C_1C_0

C_3C_2	00	01	11	10
00	0	1	1	1
01	1	1	0	1
11	0	0	0	0
10	0	0	0	0

$T_4 = C_3'(C_2 + C_1 + C_0)(C_2' + C_1' + C_0')$ POS is better for T_4.

$T_4 = C_3'C_2C_0' + C_3'C_1'C_0 + C_3'C_2'C_1$

$V = C_6'C_5C_4(C_3' + C_2'C_1') + C_6C_4'C_3'(C_2 + C_1 + C_0)(C_2' + C_1' + C_0')$
Area = 15 literals + 7 operators = 22

b. **MEASURE AND RECORD HOW LONG IT TAKES YOU TO SOLVE THIS PART OF THE PROBLEM.** When V=0, the number N can be ignored, so the four bits of N are all don't cares for any character C that does not represent a valid hexadecimal digit. Use the abundance of don't cares together with the approach used in **part (a)** to find expressions for N_3, N_2, N_1, and N_0 with minimal area among all SOP and all POS expressions (the expressions that you can find with K-maps). Note that, unlike your answer to **part (a)**, all four expressions should be either SOP or POS. For full credit, your expressions must be correct and have area (again, based on the area heuristic) of no more than 3 for N_3, 9 for N_2, 12 for N_1, and 7 for N_0.

All valid alphabetic hex characters are above 8 and thus have $N_3 = 1$. For numeric hex characters, the digit must be 8 or larger ($C_3 = 1$) to have $N_3 = 1$, but characters above '9' (with $C_6C_5C_4$=011) are not valid and thus can be treated as don't cares.

We can draw a K-map for N_3 as a function of $C_6C_5C_4$:

N_3		C_5C_4			
		00	01	11	10
C_6	0	x	x	C_3	x
	1	1	x	x	1

$N_3 = C_6 + C_3$ (area 3)

As with N_3, N_2 for numeric hex characters ($C_6C_5C_4$=011) is simply C_2. For alphabetic hex characters (C_6C_4=10), we call the function X and draw a K-map (case doesn't matter, of course).

X		C_1C_0			
		00	01	11	10
	00	x	0	1	0
	01	1	1	x	1
C_3C_2	11	x	x	x	x
	10	x	x	x	x

Note that C_2 also covers most of the 1s, so we need add only C_1C_0 and condition it on C_6 (the character is either alphabetic or invalid), giving us

$N_2 = C_2 + C_6C_1C_0$ (area 6…better than Lumetta's answer!)

Let’s handle N_1 the same way. We have C_1 for numeric hex characters ($C_6C_5C_4=011$) and draw a K-map for function Y for alphabetic hex characters ($C_6C_4=10$):

Y		C_1C_0			
		00	01	11	10
C_3C_2	00	x	1	0	1
	01	0	1	x	1
	11	x	x	x	x
	10	x	x	x	x

Here, C_1 is too general. Solving for Y, we obtain:

$Y = C_1'C_0 + C_1C_0'$ OR

$Y = (C_1 + C_0)(C_1' + C_0')$

Y applies for $C_6=1$, and C_1 applies for $C_6=0$, so we can write:

$N_1 = C_6'C_1 + C_6C_1'C_0 + C_6C_1C_0'$ (area 12)

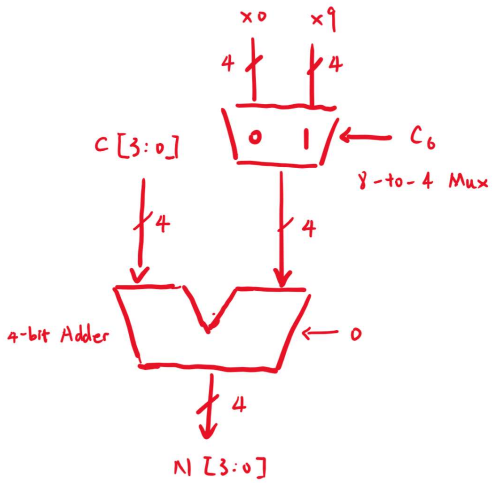
Finally, for N_0, we have C_0 for numeric hex characters ($C_6C_5C_4=011$) and draw a K-map for function Z for alphabetic hex characters ($C_6C_4=10$):

Z		C_1C_0			
		00	01	11	10
C_3C_2	00	x	0	0	1
	01	1	0	x	1
	11	x	x	x	x
	10	x	x	x	x

$Z = C_0'$, so

$N_0 = C_6'C_0 + C_6C_0'$ (area 7)

c. **MEASURE AND RECORD HOW LONG IT TAKES YOU TO SOLVE THIS PART OF THE PROBLEM.** Next, use a 4-bit adder and an 8-to-4 mux (four 2-to-1 muxes with a common control signal) to compute N. Draw your implementation clearly and label it appropriately. Be sure to draw the 8-to-4 mux as a single mux with 4-bit inputs and output. You may not use any additional components nor any additional gates (such solutions will receive no credit).



d. Based on the implementations shown in the notes and class for the ripple carry adder and the mux (count the XOR as one operator), calculate the area of the two components used in **part (c)**.
4-bit Adder: 14 × 4 = 56
8-to-4 Mux: 7 × 4 = 28

e. Sum the areas from **part (b)** and the areas from **part (d)** to determine the total area required by the two approaches.
part (b): Area = 28
part (d): Area = 84

f. Next, calculate the number of gate delays required for your expressions in **part (b)** and the number of gate delays required for the component-based design in **part (c)**.
part (b): Delay = 2 (four functions are all implemented by 2-level logic)
part (d): Delay = 2 + 2 × 3 + 1 = 9 (mux + c_out × 3 + s)

g. Make a table with two columns and three rows. To the left of the three rows, write, "area," "delay," and "human time." Above the two columns, write "K-maps" and "components." Now fill in the table with the data that you calculated. Which approach do you think is better overall?

	K-maps	Components
Area	28	84
Delay	2	9
Human Time (E.g.)	2.5 Hours	0.5 Hour

Low-level optimization decreased area by 3× and delay by 4.5× but cost around 5× human time. You decide.

4. **Preparation for Lab 3**

Update your Subversion repository to obtain the **`hw6`** subdirectory containing the program **`signals.c`**. The program prints a truth table for four functions (A, P, A_n, and P_n) calculated on three input variables (S_2, S_1, and S_0).

The functions A and P are defined by the truth table at the top of p. 103 of the class notes, in Section 3.3.3. **You need not read this section yet**—we will cover the material later. However, notice that the rows of the table are **not in binary order** and that two input combinations produce don't cares for A and P.

This problem helps you to prepare for Lab 3, in which you must implement the functions A and P on a protoboard using only NAND, NOR, and NOT gates (the functions available with CMOS devices). You may want to read through and finish this entire problem before you decide on the form that you want to use for your implementation (and thus what you turn in for **part (a)**, for example).

a. Using K-maps, derive minimal SOP or POS formulations for the A and P functions. You need only find SOP or POS, not both, but we suggest that you make the same choice for both A and P. Turn in your answers to this part on paper.

b. Now edit your copy of the **`signals.c`** program and implement the expressions that you derived in **part (a)** using C's bitwise AND, OR, and NOT operators. Be sure to mask out any bits above the least significant bit to avoid corrupting the printed values. (You will submit your edited program for **part (d)** below.)

c. Now transform your expressions from **part (a)** to use only NAND and NOR functions (remember that NOT is a 1-input NAND or NOR). **Draw the circuits** using only NAND and NOR gates to produce A and P from S_2, S_1, and S_0. In your lab implementation, complemented inputs (S_2', S_1', and S_0') will be available without the use of inverters, so leave the inverters out of your circuits. Turn in your drawings, **BUT KEEP A COPY FOR LAB 2**. If you want a challenge, try to reduce the number of gates that you need to five. Your answer to **part (d)** will require more typing, but your lab work will be slightly simpler.

d. Again edit your copy of the **`signals.c`** program and implement the expressions that you derived in **part (c)** using only NAND and NOR (you must construct these by hand from C's bitwise AND, OR, and NOT operators). **Assign these new expressions to the variables A_n and P_n. Do not modify your earlier solution to part (b).** Again, be sure to mask out any bits above the least significant bit to avoid corrupting the printed values.

The file **`goldSignals`** contains the correct output (printed table) for your modified program. To compare your results with the correct one, compile your program, say to the executable **`myProgram`**, then execute your code and send the output to a file by typing, "**`./myProgram > myFile`**" (no quotes, and any unused file name will do). Finally, execute "**`diff goldSignals myFile`**" (again, no quotes) to see a list of the differences between the correct output and your output. If the **`diff`** command outputs nothing, your equations are correct!

Commit the modified version of **`signals.c`** back to your Subversion repository for grading.

NOTE: You will not receive solutions for this problem. If you turn in an incorrect solution, you will lose points, but you must find correct expressions for your use in building the A and P functions on the protoboard. Be sure that you obtain expressions that correctly reproduce the **`goldSignals`** file.