Finite State Machine Design

1. Odd number of 1s checking using serialization

Shown below is a circuit that checks whether the n-bit input \( A = a_{n-1}...a_1a_0 \) contains an odd number of 1s in its binary representation. The circuit produces an output \( p_n \) of 0 if the number of 1s is even, or produces an output \( p_n \) of 1 otherwise. In HW 6, you designed each of the Unit Bit Slices as identical circuits consisting of NAND or NOR gates only.

Re-implement this bit-sliced design using serialization approach presented in Prof. Lumetta's notes set 3.1. Do NOT redraw the gate-level implementation of a Unit Bit Slice from HW 6. Instead use a box called Unit Bit Slice as a building block and add the storage and logic necessary to turn the bit-sliced design into a serial implementation. Assume that you have an input \( F \) to make sure that the Unit Bit Slice starts with the correct data. \( F \) is 1 when \( a_0 \) enters the Unit Bit Slice and it is 0 at all other times.
2. Sequence Counter

Using D flip-flops, design a 3-bit binary counter which counts in the following sequence: 0, 2, 4, 1, 3, 5, and the repeats the cycle. Use don't cares for the unused states (110 and 111). Follow the same steps as in the lecture: describe the states, draw the state diagram and the next-state table, provide the expression and then the circuit diagram. In order to receive full credit, you must include the relevant K-maps.

After you have completed your design, answer the following three questions.

1. Assuming that the counter starts in state 110, which state does the counter enter next?
2. Assuming that the counter starts in state 111, which state does the counter enter next?
3. Again assuming that the counter starts in state 111, describe the long-term behavior of the counter.

Hint: Read the counter examples in Lumetta’s lecture notes 3.1 and 3.2.

3. Software FSM

Finite state machines also play important roles in software design, including digital control system implementation and event-driven software design (most web services, user interfaces, and a growing number of games are designed in this way) as well as parts of compilers. In this problem, you will draw a state transition diagram corresponding to an adventure game. In the game, each “room” is a state, and the input values (0, 1, or 2 for our game) correspond to transitions. Download the program dungeon.c from our class’ web page, play the game, analyze the code, and draw a state transition diagram. Use the room number from the code to label the states, and the input value (0, 1, or 2) to label transitions between states. Note that the game ends in some of the rooms, so your diagram should not have transition arcs leaving these states.

4. Preparation for Lab 5

Read Section 3.3 of the Lumetta course notes, “Design of the Finite State Machine for the Lab.” In a previous homework, you calculated expressions for both the A and P output signals using only NAND as well as only NOR functions.

1. Use the state transition diagram (or, if you prefer, the next state table) provided in the Lumetta course notes to calculate next-state logic expressions for the lab FSM using only NAND gates (start with optimal SOP, then transform the equations to NAND, remembering that both state variables and their complements are available directly from the flip-flops).

In the program FSMsimulate.c replace the output and next-state computations with your NAND expressions. Note that (A NAND B) in C appears as (¬(A & B)); if you use other combinations of bitwise operators, or use any logical operators, you will lose points. Be sure that you remove all but the LSB from each variable using &1, as is already done in the program.

Now replace the test vector input (on line 57) with the bits provided in the Lumetta course notes (under “Testing the Design”). You should include both the initialization sequence as well as the testing sequence from the notes in the new value of TBits. Note that the input value T is taken first from the LSB of TBits, so you will need to reverse the order given in the notes, then translate from binary to hexadecimal (do not use decimal). Your full test sequence should require 15 inputs, so you should have four hex digits. Change the loop to simulate for 15 cycles instead of 21.

The file goldOutput 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 goldOutput 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!

For part (a), turn in a printout of the modified version of FSMsimulate.c containing your NAND-only equations for A, P, and the three next-state variables, $S_2^*, S_1^*$, and $S_0^*$.

2. Repeat part (a) using only NOR gates (start with POS, then transform the equations to NOR). Note that (A NOR B) in C appears as (¬(A | B)); if you use other combinations of bitwise operators, or use any logical operators, you will lose points. Follow the same steps: insert your NOR-based equations into the FSMsimulate.c program, compile it, and check that it produces the same table as appears in goldOutput.

For part (b), turn in a printout of the modified version of FSMsimulate.c containing your NOR-only equations for A, P, and the three next-state variables, $S_2^*, S_1^*$, and $S_0^*$.

Keep a copy of your solution to this problem, as you will need it for Lab 5. You will not receive your graded homework back in time for the lab.
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 FSM on the protoboard in Lab 5. Be sure that you obtain equations in both NAND and NOR form that correctly reproduce the goldOutput file.