University of Washington – Computer Science & Engineering

Spring 2019 Instructor: Justin Hsia 2019-06-04

CSE 369 QUIZ 3

Name: _Perry_Perfect__________________

UWNetID: _perfect______________________

Please do not turn the page until 11:40.

Instructions

- This quiz contains 4 pages, including this cover page.
- Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- The quiz is closed book and closed notes.
- Please silence and put away all cell phones and other mobile or noise-making devices.
- Remove all hats, headphones, and watches.
- You have 40 minutes to complete this quiz.

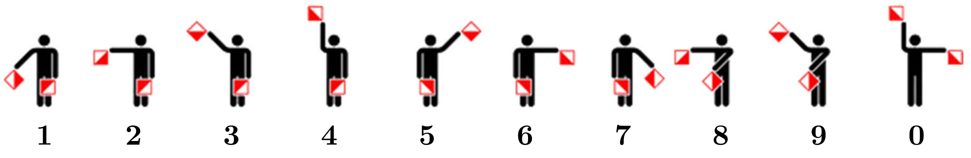
Advice

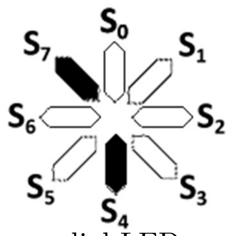
- Read questions carefully before starting. Read *all* questions first and start where you feel the most confident to maximize the use of your time.
- There may be partial credit for incomplete answers; please show your work.
- Relax. You are here to learn.

Question	Points	Score
(1) Flag Semaphore	10	10
(2) Shift Registers	13	13
(3) Parity Bit	9	9
Total:	**32**	**32**

Question 1: Flag Semaphore [10 pts]

We are building a **binary-to-flag semaphore decoder circuit**. A flag semaphore is a textual encoding using the positions of two flags. The semaphores for the digits are shown below on the left. We will use the radial arrangement of 8 LEDs, which are lit/black when the corresponding signal S_i is high/1, to display the flag positions (example below shows "3"). B represents a digit encoded in binary.





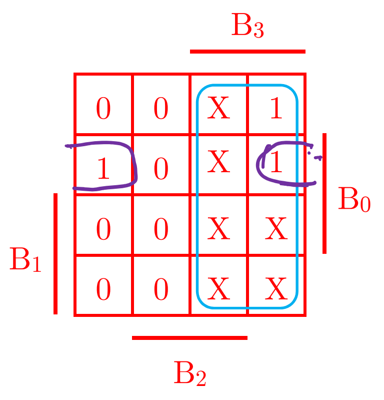
radial LEDs

(A) Complete the partial truth table below. [4 pt]

B_3	B_2	B_1	B_0	S_5	S_4	S_2
0	0	0	0	**0**	**0**	**1**
0	0	0	1	**1**	**1**	**0**
0	0	1	0	**0**	**1**	**0**
0	0	1	1	**0**	**1**	**0**
0	1	0	0	**0**	**1**	**0**
0	1	0	1	**0**	**1**	**0**
0	1	1	0	**0**	**1**	**1**
0	1	1	1	**0**	**1**	**0**
1	0	0	0	**1**	**0**	**0**
1	0	0	1	**1**	**0**	**0**
1	0	1	0	**X**	**X**	**X**
1	0	1	1	**X**	**X**	**X**
1	1	0	0	**X**	**X**	**X**
1	1	0	1	**X**	**X**	**X**
1	1	1	0	**X**	**X**	**X**
1	1	1	1	**X**	**X**	**X**

Remember that 0 goes in the first/top row!

(B) In the space below, solve for the minimal logical expression (using 2-input gates) for $\mathbf{S_5}$. [4 pt]



$V = B_3 + \overline{B}_2\overline{B}_1B_0$

(C) *Briefly* describe how you would theoretically (*i.e.* before actually running it in hardware) check if the inputs 10-15 produce valid semaphores or not. [2 pt]

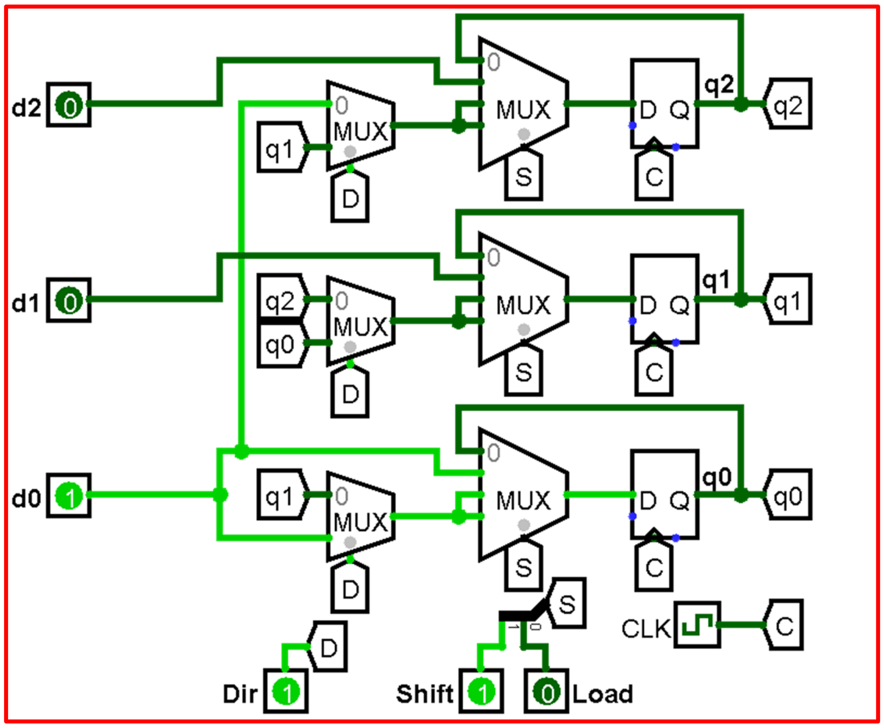
Find the minimal logic for signals S_0 to S_7 (*e.g.* using K-maps). For each row B = 0b1010 to 0b1111, see how many of these signals are high. For a valid semaphore, *exactly two* signals S_0 to S_7 must be high/1 and preferably exclusive (*i.e.* not the same as for "0" to "9").

Question 2: Shift Registers [13 pts]

We are creating a 3-bit bidirectional shifter (can shift in both directions) with parallel load that takes input bits **Shift**, **Dir**, **Load**, **d0**, **d1**, and **d2**. When shifting, we shift the current bits either to the right (*i.e.* into the less significant bit) when **Dir** = 0 or to the left when **Dir** = 1 and shift in **d0**. When loading, the state bits become d0, d1, and d2. *Shifting takes priority.*

(A) Draw the circuit diagram below using *logic gates* and *routing elements* discussed in class. Assume the clock inputs are connected properly for you. You may use multiple copies of a signal name, which are assumed connected to the same net/wire. [8 pt]

- Make sure you label the corresponding selector bits for ports of routing elements.



(B) In the Verilog testbench below, fill in the blanks to indicate how the state of our bidirectional shifter updates. Assume `assign D = {d2, d1, d0};` [5 pt]

```
initial begin                                          // q pos: 210
   Shift <= 0; Dir <= 0; Load <= 0; D <= 3'b101;     // state: 010
   @(posedge clk);  Shift <= 1;  // state: 010 (no change)
   @(posedge clk);  Dir   <= 1;  // state: 101 (right shift in d0)
   @(posedge clk);  Load  <= 1;  // state: 011 (left shift in d0)
   @(posedge clk);  Shift <= 0;  // state: 111 (left shift in d0)
   @(posedge clk);  $stop();     // state: 101 (load in D)
end
```

Question 3: Parity Bit [9 pts]

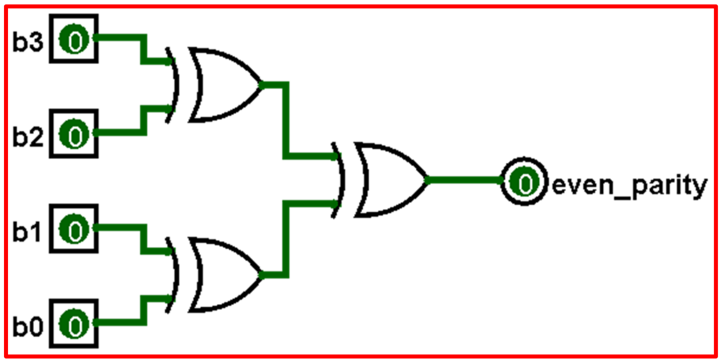
In computing, an *even parity bit* is a bit added to the end of a string of bits to ensure that the parity of the overall group (*i.e.* the string of bits plus the parity bit) is even. Parity bits can be used in various forms of *error checking*.

Examples: For `0b111`, the even parity bit would be **1** so that `0b111`**`1`** has four 1's.
For `0b1010`, the even parity bit would be **0** so that `0b1010`**`0`** has two 1's.

(A) (Circle one) Which type of gate below will be the most helpful in computing parity? [1 pt]

AND NAND NOR OR XNOR [✓ XOR]

(B) Below, implement a 4-bit even parity bit **generator** (*i.e.* compute the parity bit for a 4-bit input string). You may only use a *single type of 2-input logic gate*. [3 pt]



(C) Assume $t_{\text{NOT}} = 10$ ns, $t_{\text{AND}} = t_{\text{OR}} = 25$ ns, and $t_{\text{XOR}} = 40$ ns. We want to implement a parity **checker** circuit that indicates whether or not there (likely) was a transmission error. This circuit should return **0** if the parity of the group (string + parity bit) is even and return **1** if the parity of the group is odd. How much slower, if at all, would this checker circuit be than the parity bit generator from Part B? [3 pt]

__**40**__ ns

Add an extra XOR to add the `even_parity` bit to our computation.

(D) Describe a limitation of this type of error checking (*e.g.* what's an error situation that this parity checker wouldn't be able to detect?). [2 pt]

It can't detect an even number of errors (> 0). So, for example, two bit errors during transmission wouldn't be detected.