University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Science

EECS150, Spring 2010

Homework 2 Solutions: FPGAs, LUTs and Basic Synchronous Digital Logic

1. **Area:** The optimal solution in terms of area is **83**, which corresponds to a 3,3,2[1] configuration as shown in Figure 1: Chart (A). Notice that the 2,2,2,2 solution yields a cost of $15 \times (2^2+2) = 90$. The reason that the 3,3,2 variety saves some on cost is because the number of inputs does not exactly fit into sets of 3-LUTs (shown in light green in the figure). Notice also that the cost of a 4-LUT is $2^4 + 4 = 20$ yet the cost of $3\times$ 2-LUTs (which make a 4-LUT) is only 18. Furthermore, a 5-LUT costs $2^5 + 5 = 37$ yet the equivelent circuit ($2\times$ 2-LUTs and a 3-LUT) costs only 23. These observations tell you to never use the larger in-degree[2] LUTs in the area optimized solution.

   Delay: Since LUTs have delay proportional to their in-degree (delay$= n$), there are many optimal delay solutions, all of which have a delay of **8**. Ideal configurations include (but are not limited to): 2,2,2,2 $-$ 3,3,2 $-$ 4,4 $-$ 2,2,4, and 2,4,2. The table in Figure 1: Chart (B) shows the 2,2,4 case.

   Recall that the 4-LUT and $3\times$ 2-LUT configuration (both of which can implement a 4-LUT) in Part 1 had different costs. This is not the case with this new cost function ($3\times$ 2-LUT has a cost of 4 (as two of the 2-LUTs operate in parallel) and the 4-LUT, itself, has a cost of 4). So, in this part you can interchange different in-degree LUTs and still end up with the same cost.

   In general, optimal solutions will be balanced trees. That is, your aim is to minimize the maximum length path from the root of the tree (the last logic gate) down to the leaves (the entry-point logic gates). The cost of each hop is given by the cost function, which in this case is just the number of children at each node in the tree.

   In terms of digital design principles, minimizing the worst path in a pure combinational logic network refers to parallelizing parts of that logic so that more can happen at once (a central theme to hardware design). In hardware terminology, the maximum path length that you are trying to minimize is referred to as the *critical path*.

   Interestingly enough, one of the optimal delay configurations (3,3,2) is also the optimal area configuration!

[1] Each number refers to the in-degree of the LUTs at that level in the reduction.
[2] The number of inputs into the LUT.

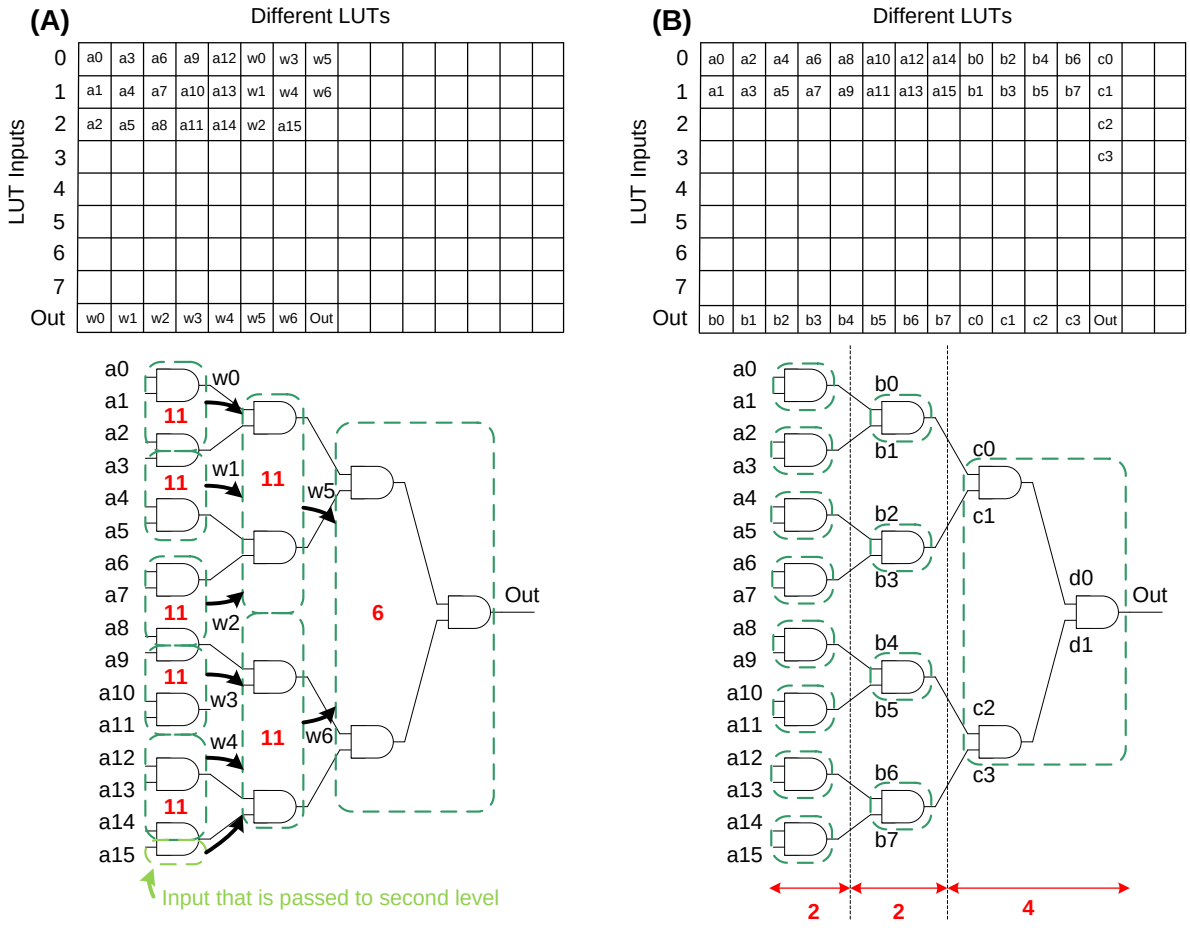
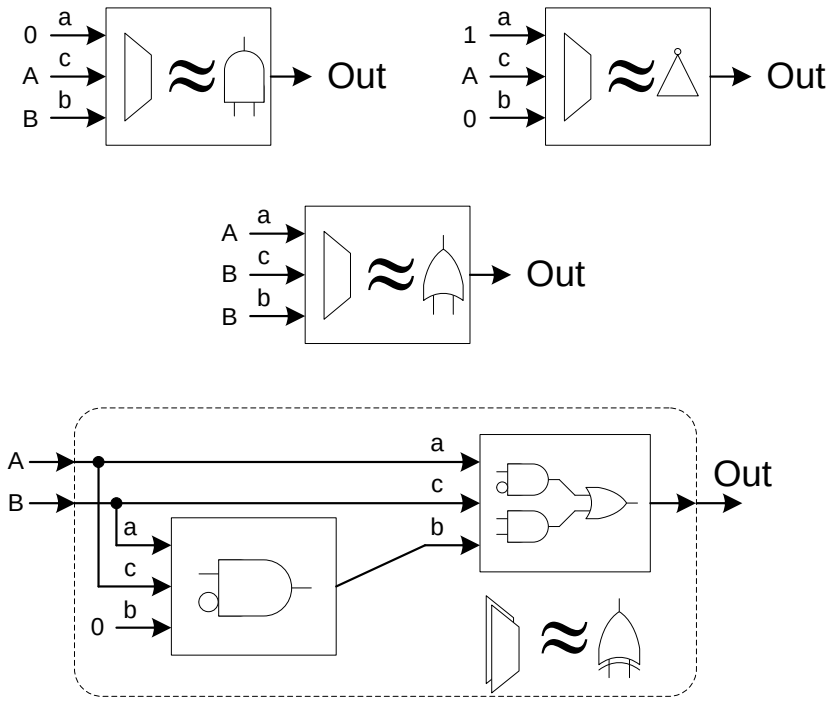


Figure 1: Problem 1. Chart (A) corresponds to area analysis and Chart (B) corresponds to delay analysis. In both, cost is shown in red.

2. The mystery circuit was a $2:1$ MUX and the various logic gate implementations are shown here:



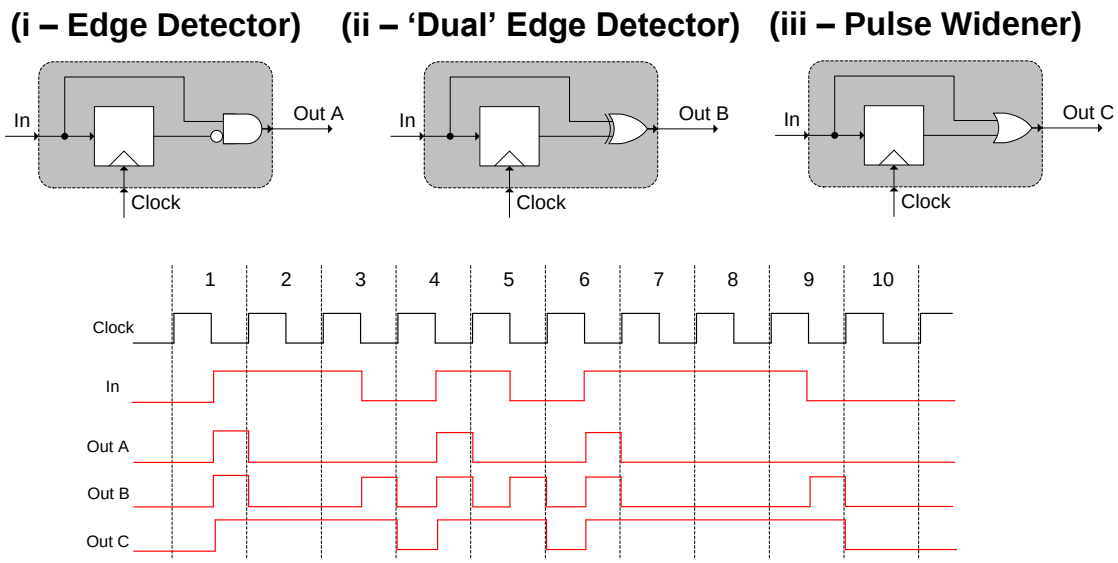
Note that the and, or and not gates each require a single MUX while the xor requires two MUXs. For the and, or and not gates, one way to think about this problem is that you have two and gates, an inverter, and an or gate per MUX. Your objective is to extract the and, or or not function from this composition of gates. In order to do this, keep the following in mind:

$$A \bullet 1 = A$$
$$A \bullet 0 = 0$$
$$A \mid 1 = 1$$
$$A \mid 0 = A$$

These properties effectively allow you to eliminate gates in the MUX so that you can isolate the gate that you need.

A key insight with the *xor* gate is that you use one input to select between the second input and the second input's complement (draw out the truth table to see this). In this case, input A is the signal you are sending to the output and input B is making the decision between A and its complement. Think about the function of the *and* gate with an inverted input! This logic is used as a gated inverter for the *xor*'s purpose.

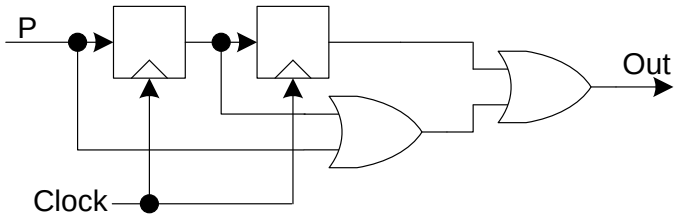
3. Shown below are the three "mystery circuits:"



The first circuit (i) is called an **edge detector**. This circuit generates a pulse (a signal that is high for at most one clock cycle) at its output when the input changes from a level 0 to a level 1. This particular edge detector is called a "positive edge detector" because it only detects the rising edge of the input signal. This need not be the case: if you switch which input into the *and* gate is inverted, you will get a "negative edge detector."

The extension to this family of circuits is the second mystery circuit (ii), called the **dual edge detector**. The dual edge detector pulses whenever the input's rising *or* falling edge is seen at the input.

The final circuit in the set (iii) is known as a **pulse widener**. The pulse widener in this problem will extend any signal it sees at its input by one cycle. The pulse widener can be extended to widen input signals to arbitrary degrees. Can you think of how? (*Hint:* the design that you have right now will serve as a "tile" that you can use over and over). An example of a pulse widener that stretches the input for three cycles is shown here:



All three of the original circuits are implemented in SLICE-level logic in Figure 2. Since the only difference between the circuits is the gate at the output (which can be partitioned into a LUT in all cases), all three look the same from a Virtex-5 CLB perspective.

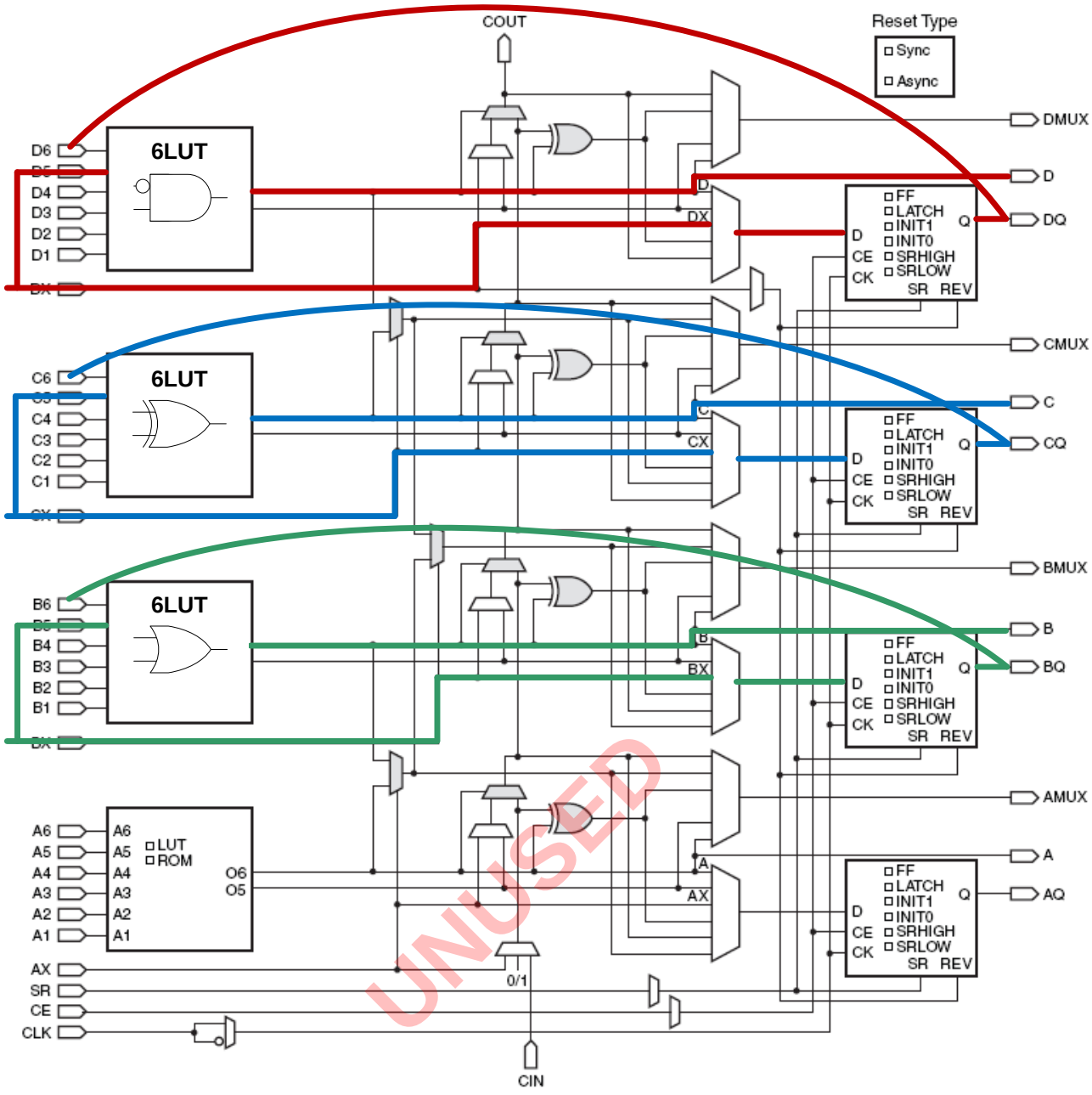
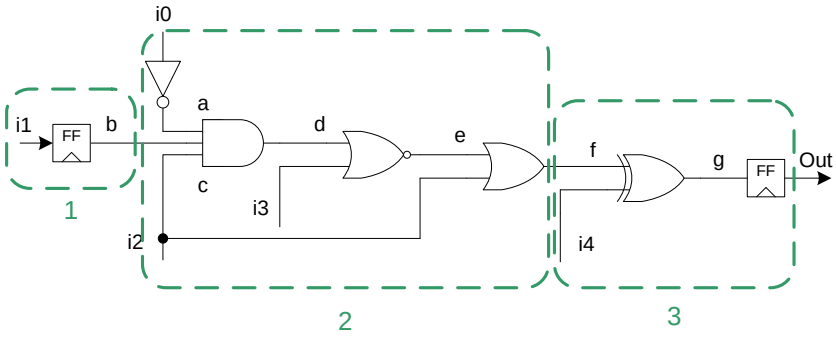


Figure 2: Problem 3 (SLICEL placement and routing). Red shows the edge detector. Blue shows the dual-edge detector. Green shows the pulse widener.

4. Here is one possible CLB partitioning:



Different CLBs

CLB inputs/outputs/config bits	1	2	3
a		i0	f
b		b	i4
c		i2	
d		i3	
e	i1		1'bx
S	1'b0		1'b1
y	b		Out
z		f	

In the third CLB, $e = 1'bx$ indicates a "don't care" (either a logic 1 or 0). We explicitely set this input to *some* logic value because we are using the MUX in this CLB to pass its value to the flip-flop (if left disconnected, a MUX input can cause failure at the output depending on how the MUX is implemented).

Other valid partitions that yield a three CLB system exist for this problem. The second CLB can terminate at the nor function and the third CLB can extend to the or, for example. If the CLB was given a delay model like in Problem 1, both solutions would still have the same delay. The primary constraints in partitioning the CLBs are as follows:

(a) The $i_1 \rightarrow b$ flip-flop must be its own CLB because there is no CLB function that supports combinational logic after the flip-flop.

(b) There must be two CLBs implementing the combinational logic after the first flip-flop and before the second flip-flop. This is because there are five inputs driving this combinational logic at different stages.

5. For those who are interested, the datasheet that covers the Spartan-3 architecture is ug331.pdf .

   (a) Despite the Spartan and Virtex being different families, the Spartan (like the Virtex) uses dedicated MUXs to build larger LUTs. Figure 3 shows the path that connects the two 4-LUTs together. In this case, the F5MUX is responsible for joining the outputs of the two 4-LUTs together.

   (b) The primitive distributed RAM cell in the Virtex-5 is a 32×1 cell (the notation being $Depth \times Width$). This comes directly from the fact that the Virtex-5 LUT primitive is the 5-LUT. Five inputs gives us $2^5 = 32$ addresses and the 5-LUT has a single-bit output only. From this, given that the Spartan-3 has 4-LUT primitives, we can infer that the distributed RAM primitive on the Spartan-3 is a 16×1 memory. (Although it isn't really fair to guarentee this given that the Spartan-3 is an arbitrary architecture, it is indeed the case). In both cases (Spartan-3 and Virtex-5), distributed RAM primitives can be built up into larger memories using the F. . . MUXs.

   (c) This question could have been interpretted in several ways and was meant to be open-ended.

   First, the Virtex-5 has 4 6-LUTs per SLICEM and the Spartan-3 has 2 4-LUTs. Thus, since there are $8\times$ Virtex-5 SLICEMs and $16\times$ Spartan-3 SLICEMs, there are technically the same number of LUTs.

   The problem, however, doesn't specify the average logic in-degree in the design. That is, how many inputs are required to drive a single bit of output in some block of combinational logic. If the in-degree was greater than 4, then perhaps the 6-LUT architecture would save LUTs.

   The problem also didn't specify whether different combinational outputs were logically driven by overlapping sets of inputs. If this was the case, and the overlap was confined to 5 inputs, then the effective LUT count on the Virtex-5 would have increased because each 6-LUT is made up of two 5-LUTs. The Spartan-3 does not support this by design (the MC15 output is used to implement SRLs and cannot be used to implement $2\times$ 3-LUTs per 4-LUT).

   The problem does say that the design uses a lot of $2:1$ MUXs relative to other logic functions. If we have $2\times$ the LUTs on the Virtex-5 because of 5-LUTs, then the Virtex-5 might be more ammenable to this design. The Spartan-3 has trump card, however, and implements $2:1$ MUXs directly in the SLICE (see **FiMUX** towards the top of the SLICE. The designers of the Spartan chips probably recognized that MUXs were awkward in 4-LUTs (a 3-LUT would implement a $2:1$ MUX and a 6-LUT will implement a $4:1$ MUX). As a result, coupled with the fact that these circuits are often needed in practice, they built support directly into the fabric. Note that there is only one $2:1$ MUX per SLICE. This might not seem like a lot. Think about the implications of there being more than one. If that were the case, then the assumption would be that the average design would be able to support just as many $2:1$ MUXs as LUTs!

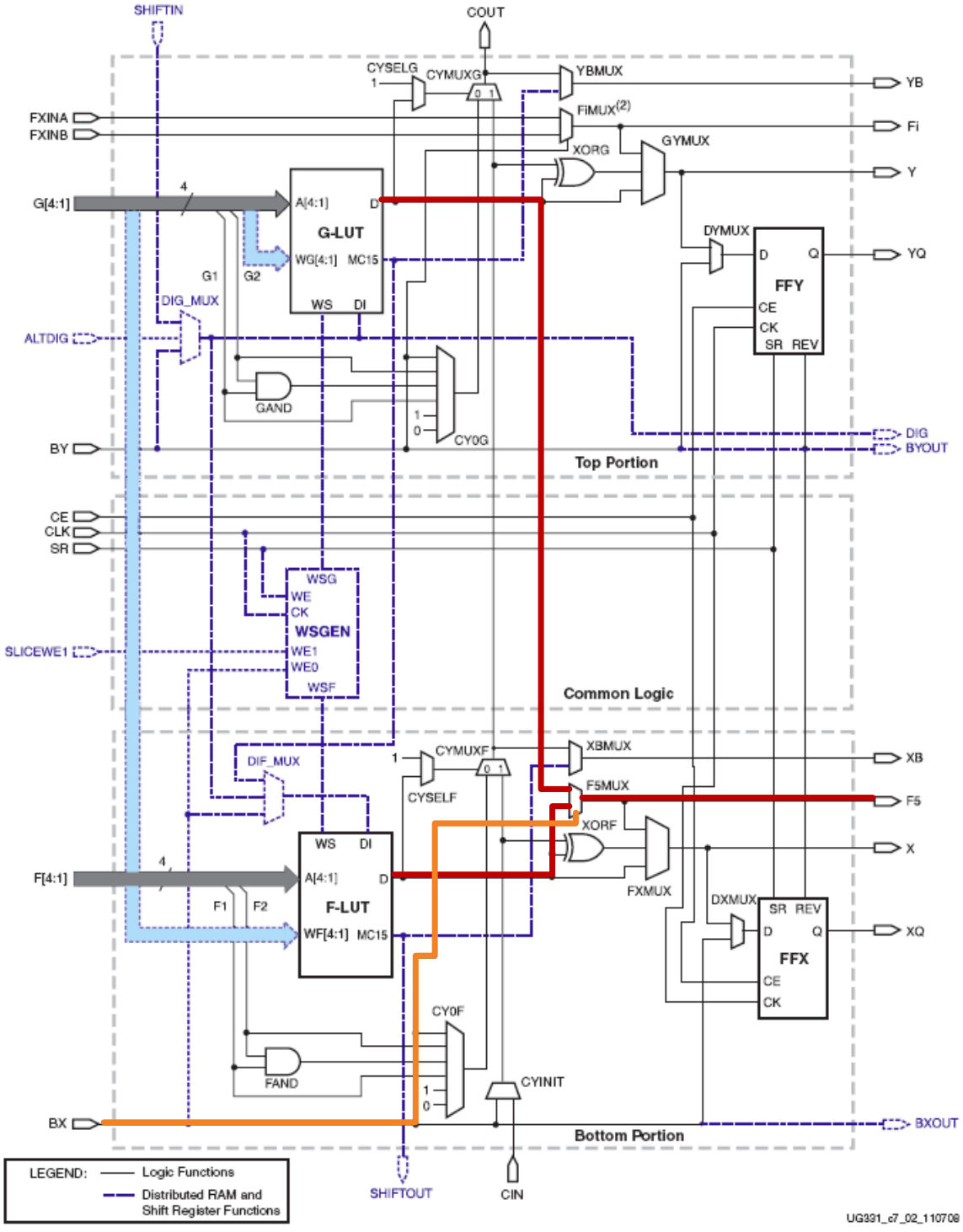


Figure 3: Problem 5.