Assignment on Architectural Design of Digital Integrated Circuits

Number-8

(Each question carries 10 marks each)

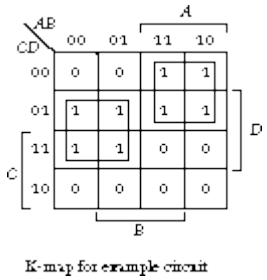
1. **Draw different circuits for implementing (A+B+C+D+E+F+G+H)/8. Analyze the accuracy in each of the case and find out the optimal circuit in respect of accuracy. Describe the advantages and disadvantages of each of the implementation.**

   Solution: Please find at the last page.

2. **What is glitch in the digital circuit? Explain the glitch considering one example (with full waveform). How to reduce the glitch in the circuit.**

   Ans: A glitch is an unwanted pulse at the output of a combinational logic network. A circuit with the potential for a glitch is said to have a hazard. In other words, a hazard is something intrinsic about a circuit; a circuit with a hazard may or may not glitch, depending on the input patterns and the electrical characteristics of the circuit. In this section we will develop a procedure that leads to hazard-free circuits.

   Glitch Detection and Elimination in Two-Level Networks: Consider the four-variable function $F(A,B,C,D) = S\, m(1,3,5,7,8,9,12,13)$. Its K-map is shown in Figure 1



K-map for example circuit

Figure:1

   The minimum sum of products form for the function is $A\bar{C}+\bar{A}D$.

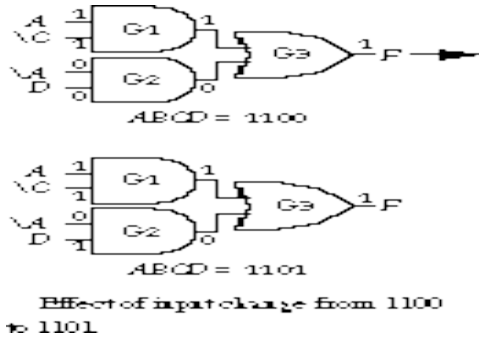


Figure 2

The gate-level implementation of *F* is given in above Figure 2. Let's examine what happens when the inputs change from *ABCD* = 1100 to 1101. When the inputs are 1100, the output of gate G1 is 1 while G2's output is 0. Thus, the output from G3 is 1. When the input changes by a single bit to 1101, the outputs of the gates remain unchanged. G1 implements the prime implicant that covers both of the input configurations we considered; it remains asserted despite the input changes. A glitch cannot happen in this case.

Now consider an input change from 1101 to 0101, another single-bit change in the inputs. When *A* goes low, $\bar{A}$ goes high, but only after a gate delay. For a short time, both *A* and $\bar{A}$ are low. This allows the outputs from G1 and G2 to be low at the same time, and thus *F* goes low. When $\bar{A}$ finally does go high, G2 will go high and *F* will return to 1. A glitch has happened! The step-by-step process is shown in Figure 3 below.

A close examination of the K-map of Figure 1 suggests what caused the problem. When the initial and final inputs are covered by the same prime implicant, no glitch is possible. But when the input change spans prime implicants, a glitch can happen. Of course, if G1 is much slower to change than G2, you might not see the glitch on *F*. The hazard is always there. Whether you actually see the glitch depends on the timings of the individual gates.

A strategy for eliminating the hazard is to add redundant prime implicants to guarantee that all single-bit input changes are covered by one such implicant. Suppose we add the implicant $\bar{C}D$ to the implementation for *F*. This strategy does not change the function's truth table. By adding the term $\bar{C}D$, *F* remains asserted for the inputs 1101 and 0101, independent of the change to input *A*.

This method eliminates the static 1-hazard, but what about static 0-hazards? First, reexpress the function *F* in minimum product of sums form: $F = (\bar{A} + \bar{C})(A + D)$

The K-map in Figure 4 clearly indicates that a static 0-hazard exists when the input changes from 1110 to 0110.

The solution is to add the redundant prime implicant $(\bar{C} + D)$ to the product of sums expression for *F*. The resulting expression is equivalent to the sum of products form that eliminates the static 1-hazard:

$$
\begin{aligned}
F &= (\overline{A}+\overline{C})(A+D)(C+\overline{D}) \\
&= (\overline{C}+\overline{A}D)(A+D) \\
&= A\overline{C}+A\overline{A}D+\overline{C}D+\overline{A}D \\
&= A\overline{C}+\overline{C}D+\overline{A}D
\end{aligned}
$$

Alternatively, we can use a shortcut to analyze the function for 0-hazards. We start with the expression that is free of static 1-hazards and work with its complement. We can then superimpose the analysis on the original K-map, looking at the zeros of the original function. A static 0-hazard exists if the implicants of the complement do not cover all adjacent 0's.

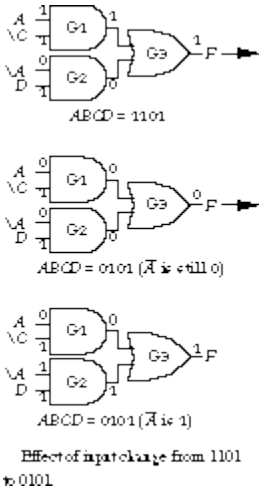


Figure 3

The revised expression for F is $A\overline{C}+\overline{A}D+\overline{C}D$. Working with its complement, we get the following:

$$
\begin{aligned}
\overline{F} &= \overline{A\overline{C}+\overline{A}D+\overline{C}D} \\
&= (\overline{A}+C)(A+\overline{D})(C+D) \\
&= AC+AC\overline{D}+C\overline{D}+\overline{A}C\overline{D}+\overline{A}\overline{D} \\
&= AC+C\overline{D}+\overline{A}\overline{D}
\end{aligned}
$$

This collection of terms does indeed cover all adjacent 0's in the K-map for the revised F. This expression is free of both static 1-hazards and 0-hazards.

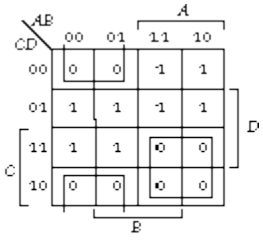


Figure 4

General Strategy for Static Hazard Elimination The preceding example leads to a general strategy for eliminating static hazards in two-level networks. Let's consider static 1-hazards first. Starting with the K-map, we examine it to make sure that all adjacent elements of the on-set are

covered by a prime implicant. If they are not, we add redundant prime implicants until all elements of the on-set are covered.

We follow a similar procedure to eliminate static 0-hazards. Given the sum of products form for the function that eliminates the static 1-hazards, we write it in product of sums form using Boolean algebra. Then we verify that adjacent elements of the off-set are covered by a common prime implicant in the product of sums form. If necessary, we add more prime implicants to cover any uncovered adjacencies.

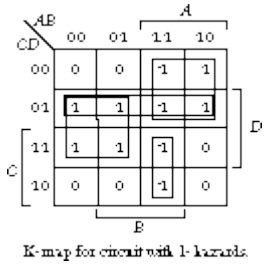
Example: A Multilevel Function Let's consider the following multilevel Boolean function:

$F = ABC + (A + D)(\overline{A} + \overline{C})$

A quick application of the distributive law yields $F_1 = ABC + A\overline{A} + A\overline{C} + A\overline{D} + \overline{C}\overline{D}$

This is the transient output function in sum of products form. Since *A* and its complement are treated as independent variables, the term $A \cdot \overline{A}$ is kept in the transient output function.

Once the function is in two-level form, we follow the procedure described in the previous subsection. First we check for static 1-hazards in the function. Note that the term $A \cdot \overline{A}$ can never cause a 1-hazard (it does indicate that a 0-hazard exists), so it can be eliminated from consideration when analyzing for 1-hazards. The K-map for the remaining terms is shown in Figure 5.



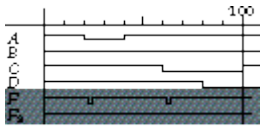
K-map for circuit with 1-hazards.

Figure 5

The function contains static 1-hazards, such as an input transition from *ABCD* = 1111 to 0111 or 1111 to 1101.

The remedy is to add the necessary redundant prime implicants. In the K-map of Figure 5, this is achieved by adding the terms *AB* and *BD* to the sum of products form of *F:* $F_2 = A\overline{C} + \overline{A}D + \overline{C}\overline{D} + AB + BD$

Because *AB* completely covers the term *ABC,* we have eliminated it from *F*2.



Waveform with 1-hazards.

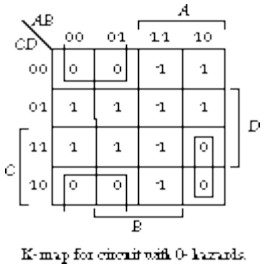
Figure 8 compares the timing behavior of the original function *F* and its revised expression *F*2. Notice that *F* glitches on the input transitions 1111 to 0111 and 1111 to 1101, while *F*2 eliminates the glitches.

We use the shortcut method to verify that the new expression is free of static 0-hazards. For the original function *F*, F is

$$\begin{aligned} F &= \overline{\bar{A}\bar{B}C + (A + D)(\bar{A} + \bar{C})} \\ &= (\bar{A} + \bar{B} + \bar{C})(A\bar{D} + AC) \\ &= \bar{A}\bar{D} + \bar{A}\bar{B}\bar{D} + \bar{A}C\bar{D} + AB\bar{C} \\ &= \bar{A}\bar{D} + AB\bar{C} \end{aligned}$$

Figure 6

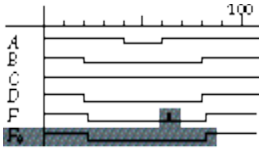
This expression corresponds to the circled 0's in the K-map of Figure 7.



K-map for circuit with 0- hazards

Figure 7

The function has a 0-hazard on the transition from 1010 to 0010, as shown in the timing waveform of



Waveform with 0-hazards

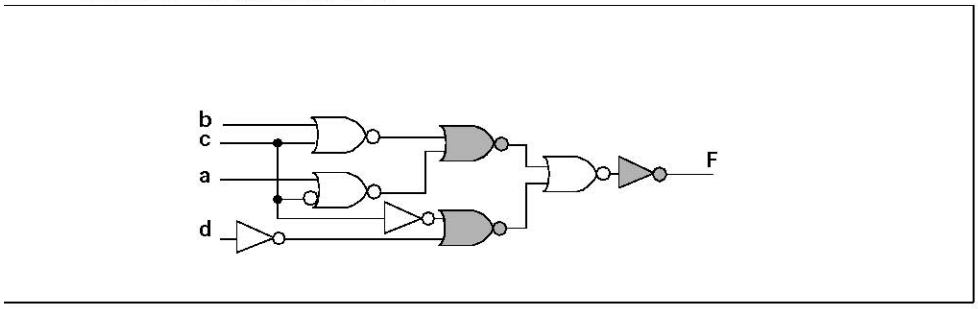
Figure 8

Figure 8. This problem can be fixed by adding the implicant BCD to F. The following is a two-level expression for *F* that is free of static 0-hazards: $F_3 = (A + D)(\bar{A} + B + C)(B + C + \bar{D})$

Expanding *F*3 to place it into sum of products form yields *F*2. Both expressions are simultaneously free of static 0- and 1-hazards.

3. **Find all the hazards in F?**

Find All The Hazards In F.



Ans:

Step 1) Remove confusing extended overbars.

$$\overline{b + c + \overline{a} + c + \overline{c} + d}$$

This is legal because DeMorgan's law does not change hazards

DeMorgan's Laws in Graphical Form (Review)

FIG. 1-20 Equivalent graphical forms for AND, OR, NAND and NOR.

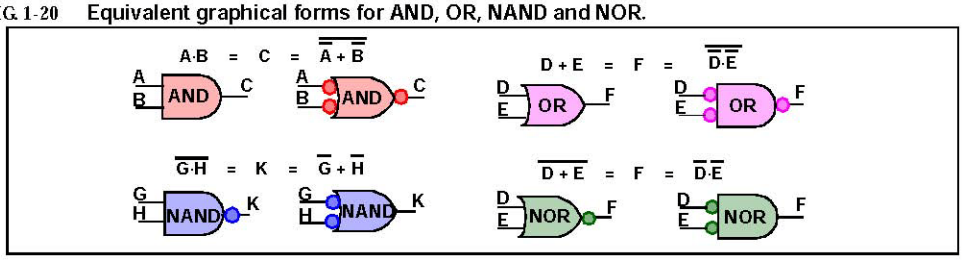
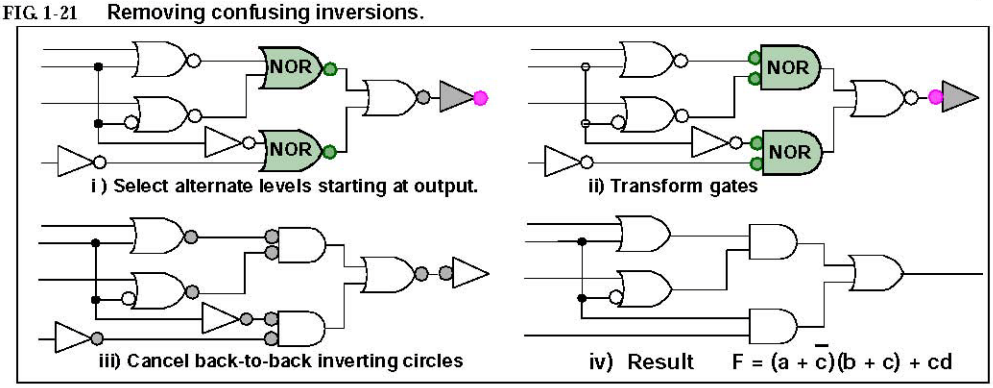
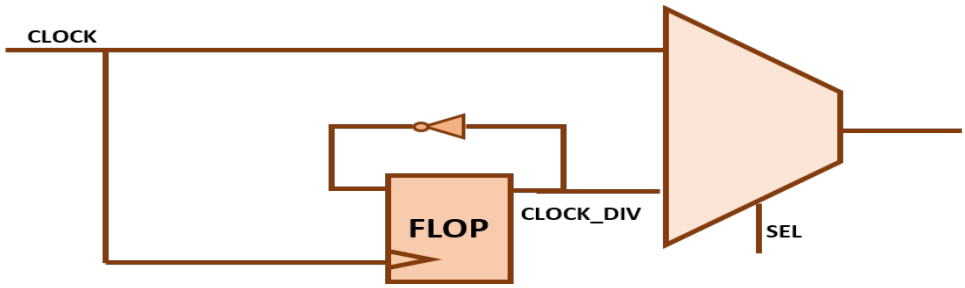


FIG. 1-21 Removing confusing inversions.



4. **In the following figure, it is desired to toggle the select of the mux from CLOCK_DIV to CLOCK and both the clocks are running. What are the architectural and STA considerations for the same?**



SOLUTION:
This is a very good example to understand how clock gating checks work, although you may/may not find any practical application for the same. We have to toggle the select of the multiplexer such that there is no glitch at the output. Let us consider architectural considerations first:

Architectural considerations:

Launching flip-flop of 'select' signal: In the post clock gating checks at a multiplexer, we discussed that if there is a mux getting clock at its inputs and select as data, then, there are two possible scenarios:

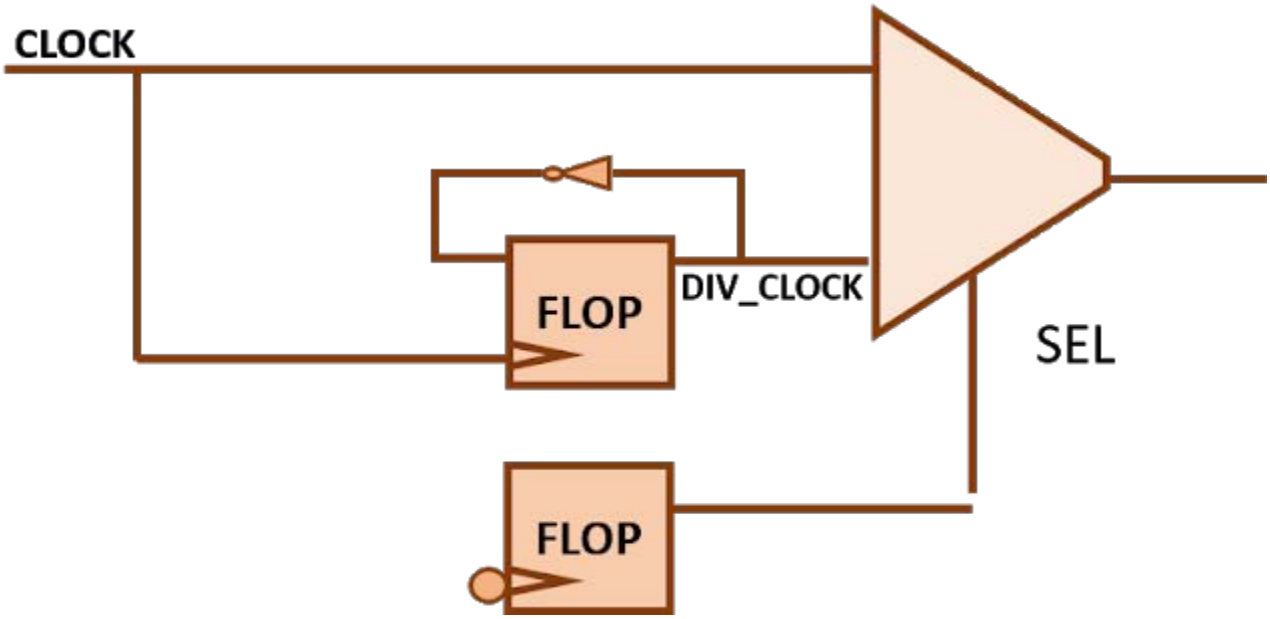
- If the other clock is at state "0", then AND type check is formed and select has to launch from negative edge-triggered flip-flop
- If the other clock is at state "1", then OR type check is formed and select has to launch from positive edge-triggered flip-flop

Now, since both the clocks are running simultaneously, both with act as "other clock" for each other. Let us choose to keep both the clocks at state "0" when select toggles. The same discussion holds true for the other scenario as well, just that appropriate values will hold. Thus,

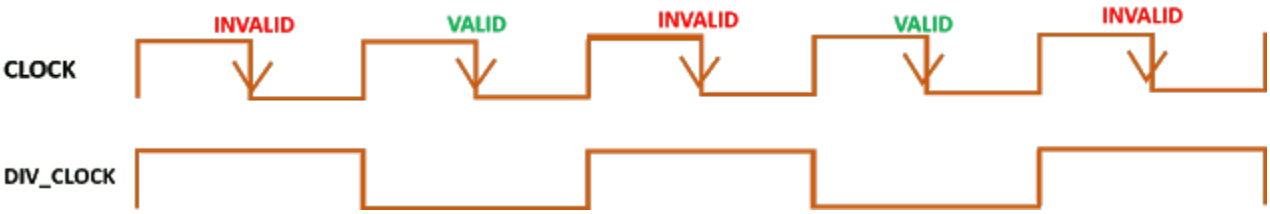
(i) Both clocks required to be at state '0' when clock toggles

(ii) There is AND-type clock gating check formed between 'select' and both clocks

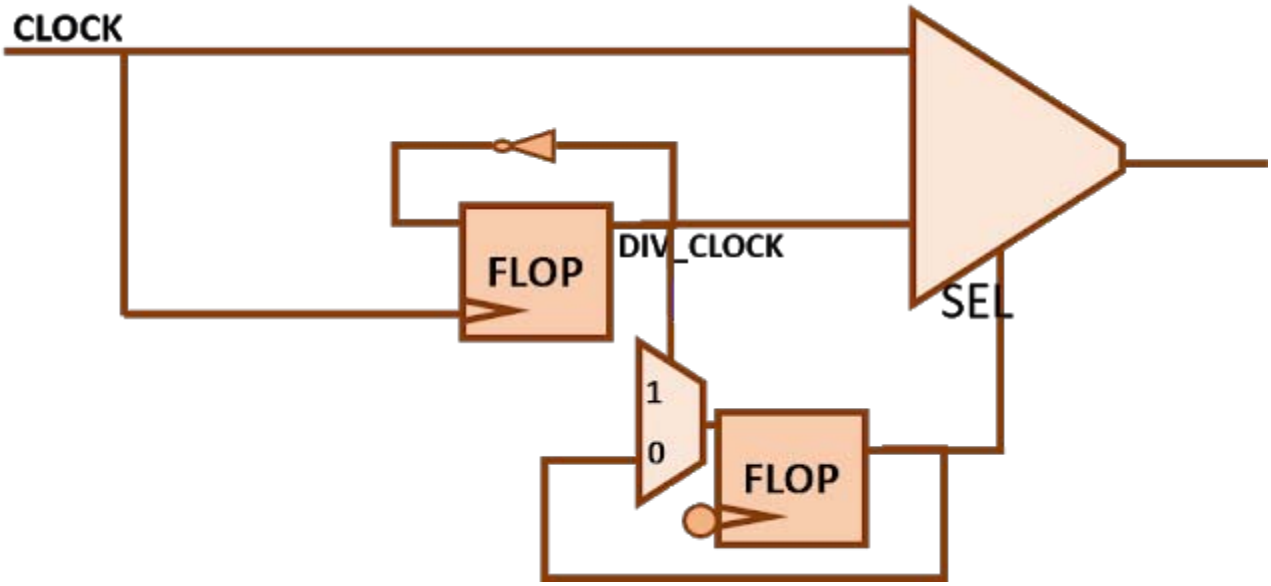
(iii) 'select' launches from negative edge-triggered flip-flop.



Valid negative edges when 'select' can toggle: Now, as mentioned above both the clocks should be zero when select toggles. Figure below shows the valid and invalid edges where 'select' can toggle. As it turns out, select can toggle only on edges labelled "VALID" as both "CLOCK" and "DIV_CLOCK" will be zero then.

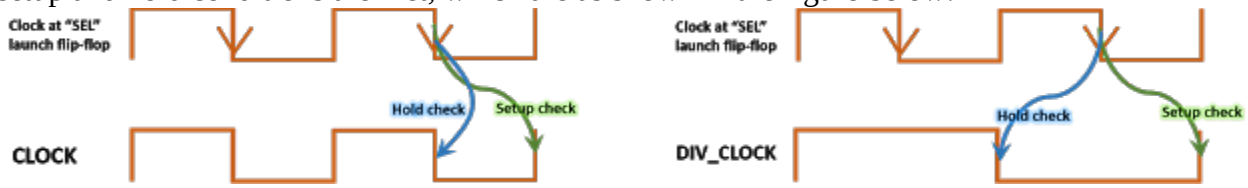


So, to ensure that "SEL" toggles only when DIV_CLOCK is "0", we can add logic to the input of the flip-flop launching "SEL" such that it allows to propagate "SEL" only when DIV_CLOCK is "0".



In the above diagram, flip-flop launching "SEL" will hold its value when DIV_CLOCK = 0. We have to keep in mind that this implementation is just a representation of what needs to be done. the actual implementation may be more complex than this depending upon the requirements.

Timing considerations: Now coming to the timing considerations, we need to ensure that the setup and hold conditions are met, which are as shown in the figure below:



5. **a) In Back-end Design Which Violation Has More Priority? Why?**

   Solution: In back-end design, Hold violation has more priority than Setup Violation. Because hold violation is related to data path and not depends on clock. Setup violation can be eliminated by slowing down the clock (Increasing time period of the clock).

   b) What Is Negative Slack? What Is Slack?

   Solution: The difference between required arrival time and actual arrival time is Negative, then it is called as Negative slack. If there is negative slack, the design is not meeting the timing requirements and the paths. which have negative slack called as violating paths. We have to fix these violations to make the design meeting timing.

The difference between required arrival time and actual arrival time is positive, then is called as positive slack. If there is positive slack, The design is meeting the timing requirements and still it can be improved.

c) How Can You Avoid Hold Time Violations? How Can You Avoid Setup Time Violations?

Solution: These are the technique which we use for hold time violation.

i**.** By adding delays using buffers

ii. By adding lockup-latches

Solution: These are the technique which we use for set-up time violation

i. Play with clock (Useful) skew.
ii. Redesign the flip flops to get lesser setup time
iii. The combo logic between flip flops should be optimized to get minimum delay
iv. Tweak launch flip-flop to have better slew at the clock pin, this will make launch flip-flop to be fast there by helping fixing setup violations.

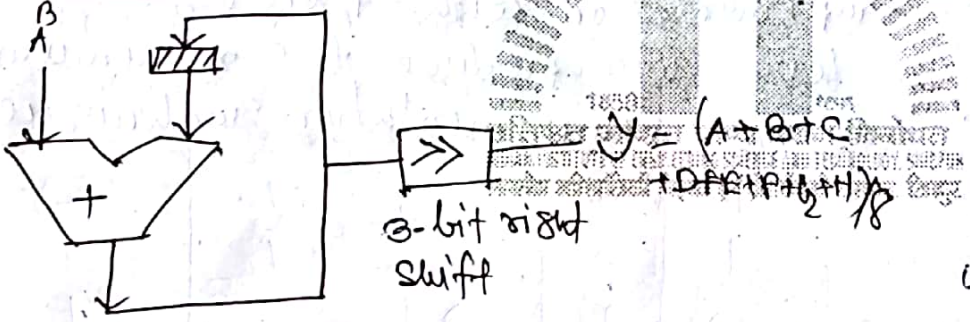
Assignment Solution - 8

Q.1 Draw different circuits for implementing $(A+B+C+D+E+F+G+H)/8$. Analyze the accuracy in each of the case & find out the optimal circuit in respect of accuracy. Describe the advantage & disadvantage of each of the implementation.

Sol: For implementing the desired function two different approaches can be considered 1) Serial 2) Parallel.

Serial Architecture

CASE 1



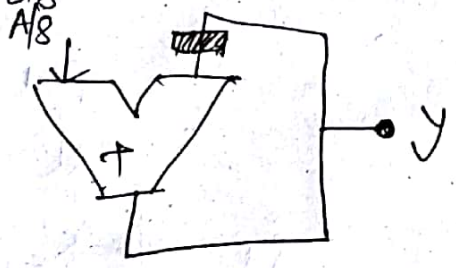
For the 1-bit shifting we can accumulate $(1-\frac{1}{2}) = .5$ error.
So. For 3-bit Shif the maximum error can occur is $7/8 = .875$ in this circuit

CASE-2

Instead of writing the equation $(A+B+C+D+E+F+G+H)/8$ we can do

$\left(\frac{A}{8}+\frac{B}{8}+\frac{C}{8}+\frac{D}{8}+\frac{E}{8}+\frac{F}{8}+\frac{G}{8}+\frac{H}{8}\right)$

:This circuit will be

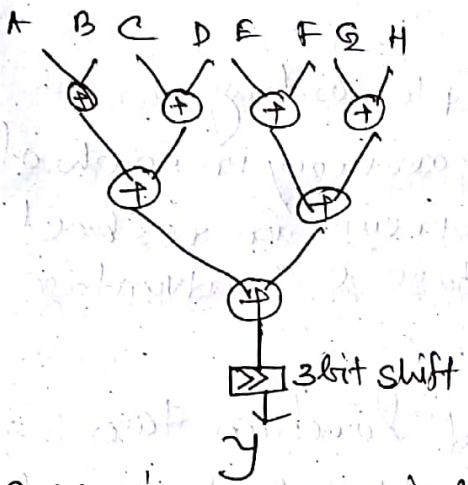


For this implementation the accured error will be (maximum)

$.875 \times 8 = 7$

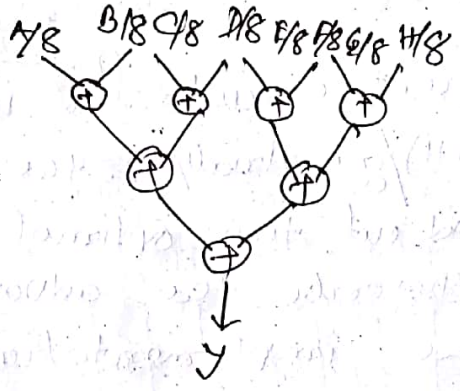
Parallel Implementation

case-1



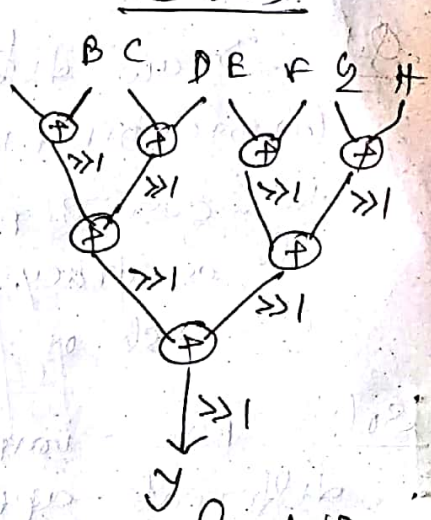
The maximum error will be
$7/8 = .875$

case-2



max error = 7

case-3



max error for this architecture will be
.50

So the parallel Implementation with 1 bit right shift distributed at each level will produce the minimum error amongst all the implementation Techniques.

□