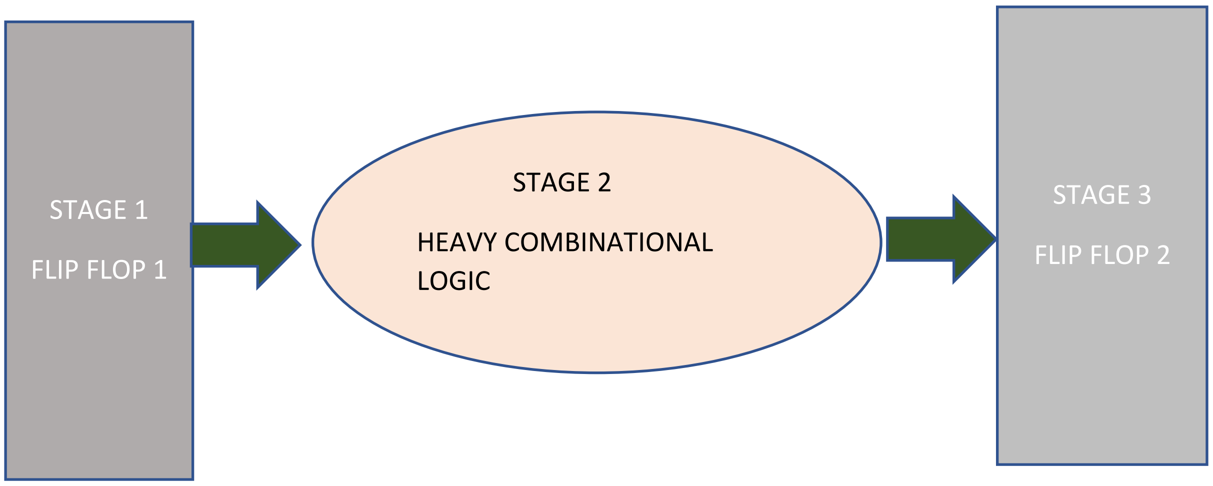
How to fix timing problems

To know how to fix timing problems, we will first need to know what timing problems exist. Problems such as a long path, race (short path), metastability and more. These problems arise when we design our system and put it through simulation and synthesis. Few things are done to take care of these problems and avoid getting them in later stages of design or in production which may result in a damaged chip. One of them is Static Timing Analysis (STA). STA should be performed quite often right from the early stages of the design. STA gives the capacitances driven by each gate, the rising and falling delays as well as relationships for each gate. Using the delays, we can compute the long path and the short path in the design. Other is retiming the paths. Retiming a logic just means that we move a couple of latches from one position to another which breaks the critical path, in such a way that there is no change in the logic that is computed.

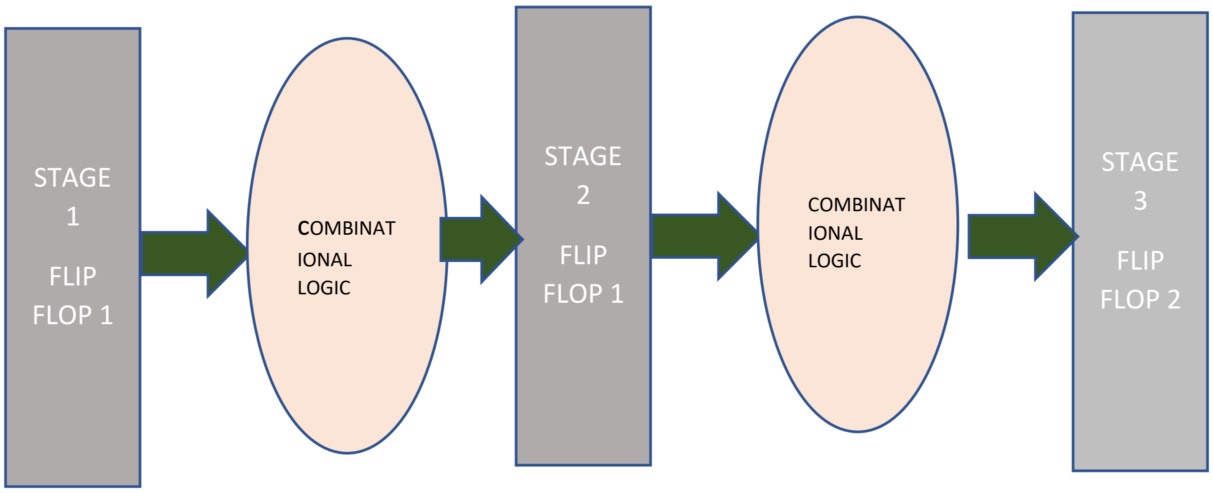
Another way to fix timing problem is using a reduced cycle time. This can be done by implementing pipelining. By pipelining your logic, you break a huge chunk of combinational logic into multiple parts and add flip flop between them. Amongst all the timing problems, we find that Race is given the highest priority over others. This is because unlike Long Path, Race fails at all clock periods. We shall discuss solutions briefly below:

1. **Pipelining**:
   Each logic block has some delay associated with it. This delay is called the Logic Delay associated with it. Logic Delay is defined as the time required for the output to change after the input changes. It is also called the propagation time to go from input to output. Circuits that are complex can have many levels of logic with Logic Delays.
   As stated above, pipelining adds flip flops in between logic by breaking them into smaller parts. The path from one flip flop to another takes one entire clock cycle to execute. Earlier we sliced a big chunk of logic into multiple parts, this left us with less logic between each flop path. That is, we now have less logic to execute per clock cycle, and can run the design at higher frequency than before.

Before pipelining:

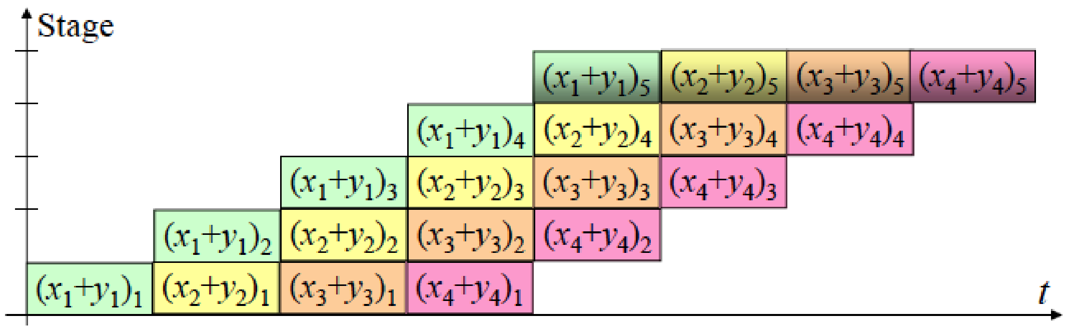


After pipelining:



The procedure for pipelining can be summed in following:

- The path that is causing timing issues like Long Path is identified after a round of simulation and synthesis.
- The combinational logic in this path is divided into multiple parts.
- Flip flops are added in between after breaking the combinational logic.
- We must make sure if the outputs of every flip flop are reset when every reset condition occurs.
- If interfaces are included in your design, we must make sure they are pipelined to same amount of level of pipelining as the logic blocks.


Stage
$(x_1+y_1)_5$ $(x_2+y_2)_5$ $(x_3+y_3)_5$ $(x_4+y_4)_5$
$(x_1+y_1)_4$ $(x_2+y_2)_4$ $(x_3+y_3)_4$ $(x_4+y_4)_4$
$(x_1+y_1)_3$ $(x_2+y_2)_3$ $(x_3+y_3)_3$ $(x_4+y_4)_3$
$(x_1+y_1)_2$ $(x_2+y_2)_2$ $(x_3+y_3)_2$ $(x_4+y_4)_2$
$(x_1+y_1)_1$ $(x_2+y_2)_1$ $(x_3+y_3)_1$ $(x_4+y_4)_1$
t

The figure above is an example of 5 stage pipelined adder. Here, as we see that first stage adds two variables. The second stage adds the other variables synchronously with the previous stage variables and the result is stored. Thus this continues until the last variable is added and stored. All stages occur in a simultaneous manner.

The Static Timing Analysis method is used to obtain long paths and short paths. For this, we first calculate the logic delays through every gate. This logic delay depends upon the load capacitance as well as the input or wire capacitance. There are few conditions we need to take into consideration to avoid timing issues like Long Path and Short Path. In order to avoid Long Path following condition must be met:

$$C.T. - C \rightarrow Q - (2 \times Skew) - I.S. >= L_d$$

Where C.T. is the clock time, L_d is the logic delay, C → Q (Clock to Q) is the time taken for the output to appear after the clock comes and I.S. is the input setup time.

If the above conditions are not met, pipelining is required in the design. The above equation changes as follows:

$$C.T. - C \rightarrow Q - (2 \times Skew) - I.S. >= L_d / N$$

N being the number of pipeline stages.

Similarly, to avoid Race in the design, following condition must be met:

$$I.H. + (2 \times Skew) - C \rightarrow Q <= L_d$$

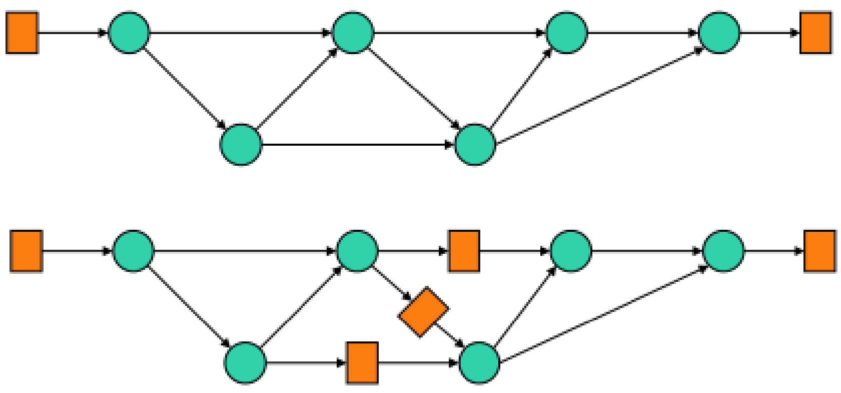
Where I.H. is the input hold time. Rest of the parameters have the same meaning as above.

After pipelining the above condition changes as follows:

$$I.H. + (2 \times Skew) - C \rightarrow Q <= L_d / N$$

Now that we have divided the logic into N stages, the logic delay L_d is reduced compared to its previous value. This tells us that the stages are executed in a concurrent manner with a buffer added to each of the pipelined stages.

2. **Retiming:**
   The concept of retiming explains that the timing of a given logic design to be changed or retimed. How can the logic be retimed? We do this by either reduction of clock period or by rearranging registers such that we have less combinational logic between two registers.



   The above figure explains how the logic between two registers is minimized by adding three more registers in between the combinational logic blocks. This method is also called fast optimal algorithm.
   Another way of retiming is pipelining our design which is described above.

3. **The Use of parenthesis**:

By correct use of parenthesis, we can eliminate the problem of Race and Long path. Whatever we put inside the parenthesis gets done first. For example, we have below logic:

A+B+ECD+FYZ+GLK+H

If input E is racing, we put E as deep nested as possible inside parenthesis.

A+B+(((E)C)D)+FYZ+GLK+H

This will result in input E moving ahead of the logic tree and a significant increase in the overall delay.
If input E is creating a long path, we exclude E from parenthesis and put rest of the inputs inside.

A+B+E(CD)+FYZ+GLK+H

Here, E gets worked at last and is moved to the end of the logic tree. Hence, this is one of the quick fixes that can be used to fix timing problems.

4. **The Use of Multiplexers**:
Another quick fix method is the use of Multiplexers. A multiplexer is a device which is simply a state machine, which has 2^n inputs and n select lines. 2:1, 4:1, 8:1 and 16:1 are few examples of mux's. It selects the input or combination of inputs to be outputted based on the select lines.
There are two ways you can implement a mux in Verilog. First is by writing 'if else' statements and the other is by using ternary operator. The latter is a preferred because using 'if else' statement can be expensive in terms of area, gates and power as well as time; While the latter one is optimized for both resources as well as speed. A multiplexer has a delay of 'a gate and a half' which makes it speed optimized. Below is an example for mux implementation:

Example 1:
```
if (reset)
a <= 0;
b <= 0;
end else begin
a <= #1 a_d;
b <= #1 b_d;
end
```

Example 2:
```
a <= reset ? 0 : a_d;
b <= reset ? 0 : b_d;
```

5. **Use of DesignWare**:
There exists a library in the Synopsys environment where a number of logic blocks are defined. These blocks are reusable, that means you can include them in your top-level file and easily use them in your design.
The advantage of these DesignWare libraries is that they are coded in such a way that the synthesizer understands them without a lot of effort. They are much faster than the handmade low level designs and they occupy less area. There are options where you can

choose designs which are pipelined internally in multiple stages. For example, if you have the following multiplication operation to be performed:

$$Y = A * B$$

Assume that this multiplication has introduced a Long Path problem in your design. Immediate instinct would be to pipeline this multiplier into stages to eliminate the Long Path. If you manually pipeline the multiplier in your design, there are chances that this might become the Long Path. Instead, you have an option of using a DW02_mult_(the number of stages) DesignWare multiplier from the Synopsys library. You could do the same for adders as well.

6. **Asking the synthesizer to try harder**:
   When we put our design through simulations and synthesis, the synthesizer tries to improve and optimize the design. It tries to eliminate the timing problems from the design by substituting cells for better speed, area and power. When it is done with the optimization or if it gives up, it throws an error and displays the Slack by which it failed in the terminal window. What this means is, we want the synthesizer to put more efforts on improving and optimizing the design by trying more and different types of cells, faster cells. We can do following things for this:
   - We change the setting for compile effort to high from the current setting (medium or low).
   - Increase run times. It will try more tricks and cell substitutions.
   - Set the cycle time 20% lesser than as much as wanted. The synthesizer will now try harder on Long Paths as they appear bigger than they are. Even if the tool fails to optimize, it will be within the range of the desired cycle time.

INTERFACES:

Interfaces are an addition to existing logic design. Interfaces are used in logic for many reasons, mainly to single out presence of data and to enable block reuse. In the entire logic design, it is hard to tell when the output is really valid as there might be values that are not desired by us; so we add interfaces to them.

There are mainly four different types of interfaces, namely, 'Push,' 'Pull', 'Push and Stop', 'Pull and Stop'. These control signals get added to every logic block in our design pipeline. In the 'Push' interface, the control signal is originated from the data source and goes through flip flops all the way till the destination. A Push control signal assumes that the next stage in queue is always be able to accept the data. It tends to empty the pipeline that is filled with data. The data flow is from Source to Destination. The disadvantage of this interface is that the next stage will not be able to accept the data always, still it will try to move the data ahead in the pipeline. This may cause data loss which is why a Push and Stop interface was designed.

The Push control signal goes through series of flip flops to the destination. A Stop control signal is originated from the destination and goes to the source, it ripples back to the source. The data flow for stop signal is from destination to source. When the circuit cannot output the data, it sends a Stop signal to every logic block in the design. A Stop is the logical AND of the Stop signal of the previous block and the valid bit from the Push circuit for the current block.

Stop B = Stop A & Valid B

Assuming that B is the current block and A is the previous block of Stop circuit. Valid B is the signal from the Push interface. Valid signal tells the next block that it has received data from the previous block and it is ready to push it to the next block. A logic block is valid with data when there is a Push from previous block and there is no Stop from the block ahead.

Valid B = Push A & ! Stop A

Assuming that A is the previous block. In this interface, the only assumption here is that the Stop signal stops when data cannot be outputted.

Similarly, we have ‘Pull’ interface and ‘Pull and Stop’ interface. A control signal that originates from the destination and goes to the source is called a Pull signal. In this case, the Pull signal ripples back from the destination to source. It tends to fill in the pipeline with data. It tells the source to place more data on the data bus. It assumes that the data is always available on the data bus.

The disadvantage of a ‘Pull’ interface is that it expects the data to be available at all times, which might not be the case on reset and several other conditions. So, when data is not available to pull, and the Pull signal is expecting data, it might output a value which is not desirable or a junk value. This explains why this interface is not used much. The widely used interface is the ‘Pull and Stop’ interface. The pull signal here works just the way it works in the ‘Pull’ interface. The Stop signal here originates from the source and goes till the destination. Whenever there is no data available to put on the data bus, the source will send out the stop signal indicating the unavailability of the data. This will result in the Pull signal to stop expecting data until the Stop signal is on.

The use of FIFOs:

FIFO stands for First In First Out. It is a memory which has two ports, one for writing data and one for reading the stored data. It has two counters, namely, a Write counter and a Read counter. The Write counter increments when data is put in the FIFO and the Read counter increments when data is outputted from it. These counters are reset on every reset condition that appears.

The most commonly used interface is the ‘Push and Stop’ interface. One of the major disadvantage of this interface is that while it allows the designer to pipeline their circuit blocks, the Stop signal which ripples back from destination to the source can itself become a Long Path if the pipeline become large. To break this long path problem caused by the Stop signal, FIFOs are used on the output of the design. FIFO are also used for reasons such as Rate Smoothing.

SUMMARY:

When logic is designed and implemented, there always are some minor issues and some major ones. It is okay to not care about minor issues because some “quick fixes” that we saw can fix them easily, namely, use of multiplexers, use of parenthesis, lying to the synthesizer so that it tries harder to optimize and more. The issues we should be worried about are the major ones related to timing which are tricky to get around and make them work. Some can be fixed by pipelining the logic design and adding interfaces to them. But others are just very stubborn, and they refuse to work in every other condition. It is critical to solve these errors as they might lead to a great loss to the organization in terms of money and to the engineer in terms of his or her career. This will result into the engineer going back to the design, looking at where he messed up the logic and redesigning that bit of code to make it work.