1.3 Timing

Timing refers to the clock speed of a design. The maximum delay between any two sequential elements in a design will determine the max clock speed.

The maximum speed, or maximum frequency, can be defined according to the straightforward and well-known maximum-frequency equation (ignoring clock-to-clock jitter):

$$F_{max} = \frac{1}{T_{clk-q} + T_{logic} + T_{routing} + T_{setup} - T_{skew}}$$

where F_{max} is maximum allowable frequency for clock; $T_{clk\text{-}q}$ is time from clock arrival until data arrives at Q; T_{logic} is propagation delay through logic between flip-flops; $T_{routing}$ is routing delay between flip-flops; T_{setup} is minimum time data must arrive at D before the next rising edge of clock (setup time); and T_{skew} is propagation delay of clock between the launch flip-flop and the capture flip-flop.

The next sections describes various methods and trade-offs required to improve timing performance.

1.3.1 Add register layers

Intermediate layers of registers can be added to a critical path to improve timing if additional clock cycle latency does not violate the design specifications.

The design of a FIR (Finite Impulse Response) implementation in *fir.sv* and synthesized in Figure 1.3.1a does not meet timing as all the multiply and add operations occur in one clock cycle. In other words, the critical path of one multiplier and one adder is greater than the minimum clock period requirement.

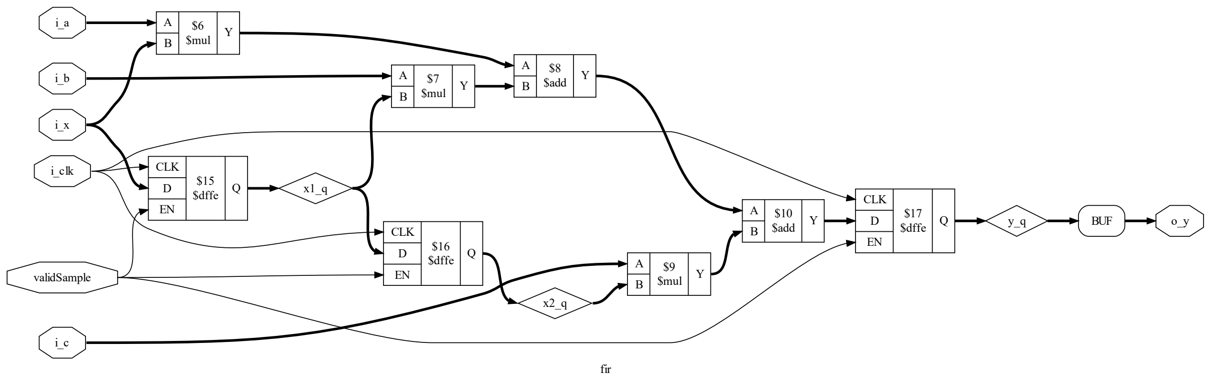

Figure 1.3.1a: Synthesized design of a FIR that would not meet timing criteria.

If the latency requirement is not fixed at 1 clock cycle, a pipeline layer can be added between the multipliers and the adder to improve timing. This is implemented in *pipelined_fir.sv* and synthesized in Figure 1.3.1b.

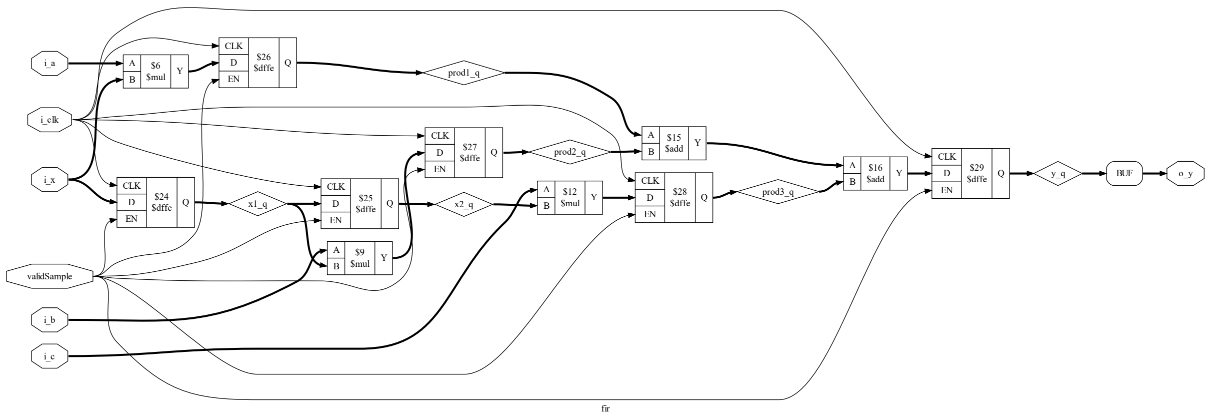
Figure 1.3.1b: Synthesized design of a pipelined FIR.

Multipliers are good candidates for pipelining because the calculations can be easily be broken up into stages. Additional pipelining is possible by breaking the multipliers and adders up into stages that can be individually registered.

In general, adding register layers improves timing by dividing the critical path into two paths of smaller delay.

1.3.2 Parallel Structures

The second strategy for architectural timing improvements is to reorganize the critical path such that logic structures are implemented in parallel. This technique should be used whenever a function that currently evaluates through a serial string of logic can be broken up and evaluated in parallel.

Consider the multiplication of an 8-bit binary number with itself. The number can be split into two 4-bit binary numbers i.e X ={A,B} and the multiply operation can be reorganized as: X*X = {A, B}*{A, B} = {(A*A), (2*A*B), (B*B)}; reducing it to a series of 4-bit multiplications that can recombined.

This has been implemented in *parallel_pipelined_power3.sv* and the synthesized design is shown in Figure 1.3.2.

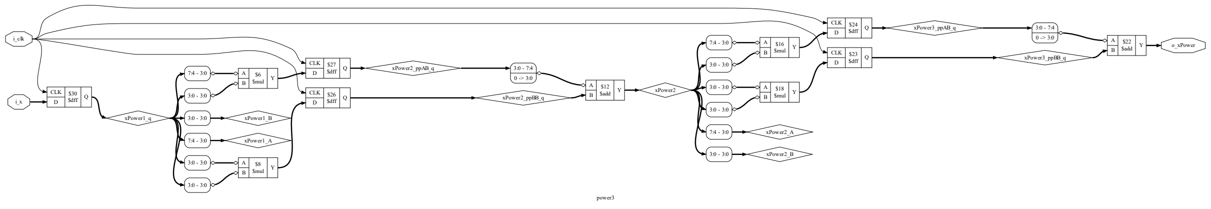
Figure 2: Synthesized design of a parallelized implementation that calculates the cube of a number.

Points of interest of the synthesized design: TODO

Separating a logic function into a number of smaller functions that can be evaluated in parallel reduces the path delay to the longest of the substructures.

1.3.3 Flatten Logic Structures

By removing priority encodings where they are not needed, the logic structure is flattened and the path delay is reduced.

1.3.4 Register Balancing

This strategy involves redistributing logic evenly between registers to minimize the worst-case between any two registers. This technique should be used whenever logic is highly imbalanced between the critical path and an adjacent path. Because the clock speed is limited by only the worst-case path, it may only take one small change to successfully rebalance the critical logic without changing its functionality.

Consider an adder that adds three 8-bit inputs. It's implemented in *adder.sv* and the synthesized design is shown in Figure 1.3.4a. The first register stage consists of rA, rB, and rC, and the second stage consists of Sum. If the critical path is defined through the adder, some of the logic in the critical path can be moved back a stage, thereby balancing the logic load between the two register stages.

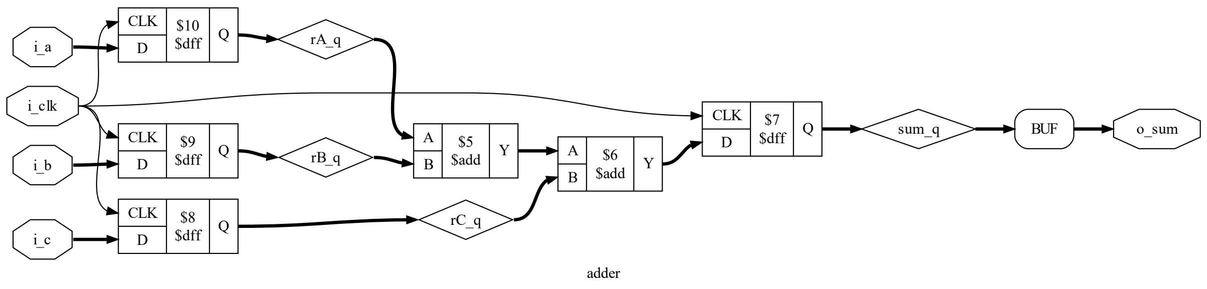

Figure 1.3.4a: Synthesized design of an adder that adds three 8-bit inputs.

This has been implemented in *balanced_adder.sv* and the synthesized design is shown in Figure 1.3.4b. Thus, register balancing improves timing by moving combinatorial logic from the critical path to an adjacent path.

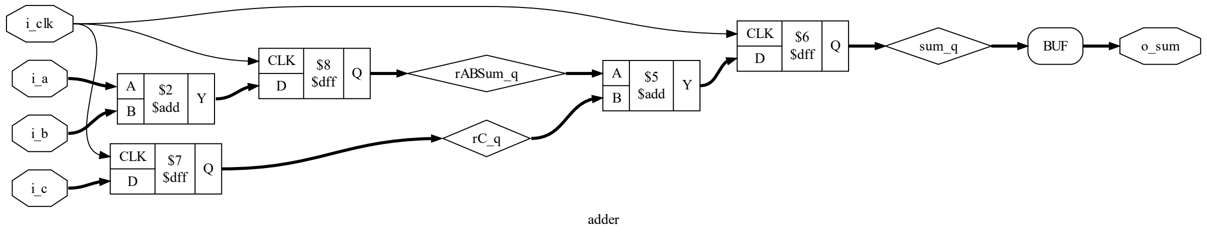

Figure 1.3.4b: Synthesized design of a balanced adder that adds three 8-bit inputs. One of the add operations has been moved back one stage between the input and the first register stage.

1.3.5 Reorder Paths

The fifth strategy is to reorder the paths in the data flow to minimize the critical path. This technique should be used whenever multiple paths combine with the critical path, and the combined path can be reordered such that the critical path can be moved closer to the destination register. With this strategy, we will only be concerned with the logic paths between any given set of registers.

Consider the module implemented in *randomLogic.sv* and its synthesized design in Figure 1.3.5a. The critical path is between C and Out and consists of a comparator in series with two gates before reaching the final decision mux.

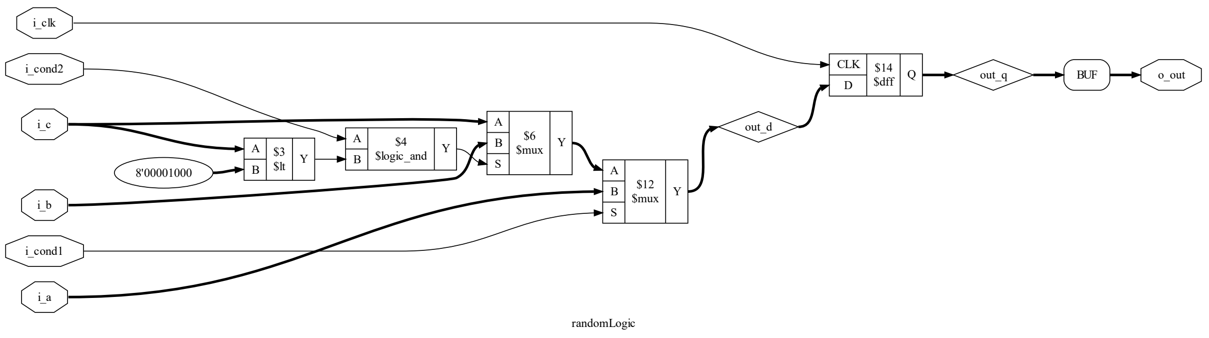

Figure 1.3.5a: Synthesized design of random logic with long critical path.

If the conditions are not mutually exclusive, the code can be modified to reorder the long delay of the comparator as implemented in *reorganized_randomLogic.sv.* The synthesized design is shown in Figure 1.3.5b. Thus, timing can be improved by reordering paths that are combined with the critical path in such a way that some of the critical path logic is placed closer to the destination register.

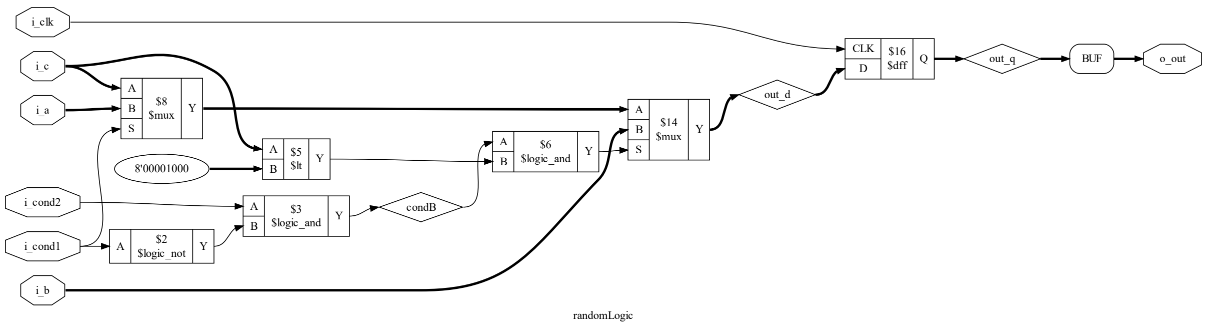

Figure 1.3.5b: Synthesized design of random logic that has been reordered to reduce the critical path.