Chapter 22

Asynchronous Sequential Circuits

Asynchronous sequential circuits have state that is not synchronized with a clock. Like the synchronous sequential circuits we have studied up to this point they are realized by adding state feedback to combinational logic that implements a next-state function. Unlike synchronous circuits, the state variables of an asynchronous sequential circuit may change at any point in time. This asynchronous state update – from next state to current state – complicates the design process. We must be concerned with hazards in the next state function, as a momentary glitch may result in an incorrect final state. We must also be concerned with *races* between state variables on transitions between states whose encodings differ in more than one variable.

In this chapter we look at the fundamentals of asynchronous sequential circuits. We start by showing how to analyze combinational logic with feedback by drawing a flow table. The flow table shows us which states are stable, which are transient, and which are oscillatory. We then show how to synthesize an asynchronous circuit from a specification by first writing a flow table and then reducing the flow table to logic equations. We see that state assignment is quite critical for asynchronous sequential machines as it determines when a potential race may occur. We show that some races can be eliminated by introducing transient states.

After the introduction of this chapter, we continue our discussion of asynchronous circuits in Chapter 23 by looking at latches and flip-flops as examples of asynchronous circuits.

22.1 Flow Table Analysis

Recall from Section 14.1 that an asynchronous sequential circuit is formed when a feedback path is placed around combinational logic as shown in Figure 22.1(a). To analyze such circuits, we break the feedback path as shown in Figure 22.1(b)

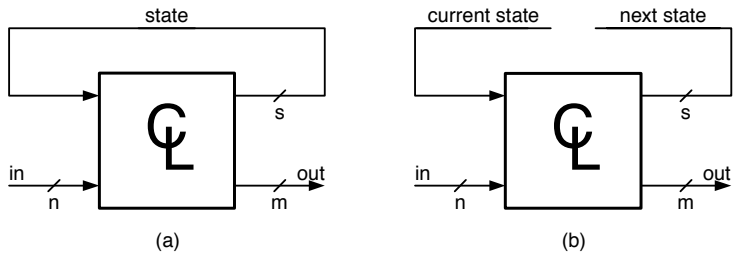


Figure 22.1: Asynchronous sequential circuit. (a) A sequential circuit is formed when a feedback path carrying state information is added to combinational logic. (b) To analyze an asynchronous sequential circuit, we break the feedback path and look at how the next state depends on the current state.

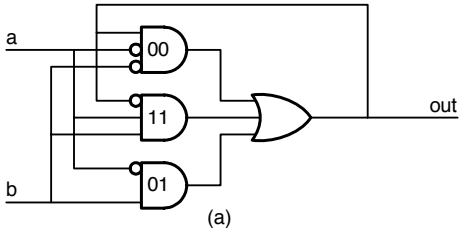
and write the equations for the *next* state variables as a function of the *current* state variables and the inputs. We can then reason about the dynamics of the circuit by exploring what happens when the current state variables are updated, in arbitrary order if multiple bits change, with their new values.

At first this may look just like the synchronous sequential circuits we discussed in Section 14.2. In both cases we compute a next state based on current state and input. What's different is the dynamics of how the current state is updated with the next state. Without a clocked state register, the state of an asynchronous sequential circuit may change at any time (asynchronously). When multiple bits of state are changing at the same time (a condition called a *race*. The bits may change at different rates resulting in different end states. Also, a synchronous circuit will eventually reach a steady state where the next state and outputs will not change until the next clock cycle. An asynchronous circuit on the other hand may never reach a steady state. It is possible for it to oscillate indefinitely in the absence of input changes.

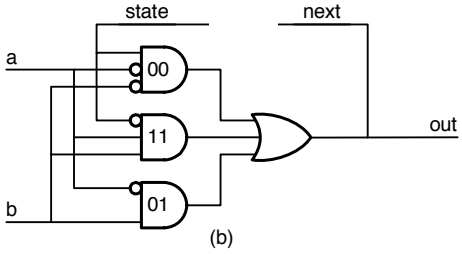
We have already seen one example of analyzing an asynchronous circuit in this manner - the RS flip-flop of Section 14.1. In this section we look at some additional examples and introduce the *flow table* as a tool for the analysis and synthesis of asynchronous circuits.

Consider the circuit shown in Figure 22.2(a). Each of the AND gates in the figure is labeled with the input state ab during which it is enabled. For example, the top gate, labeled 00, is enabled when a is high and b is low. To analyze the circuit we break the feedback loop as shown in Figure 22.2(b). At this point we can write down the next-state function in terms of the inputs, a and b, and the current state. This function is shown in tabular form in the *flow table* of Figure 22.2(c).

Figure 22.2(c) shows the next state for each of the eight combinations of inputs and current state. Input states are shown horizontally in Gray-code order. Current states are shown vertically. If the next state is the same as the current state, this state is *stable* since updating the current state with the next



(a)



(b)

State	Next			
	00	01	11	10
0	(0)	1	1	(0)
1	(1)	(1)	0	0

(c)

Figure 22.2: An example asynchronous sequential circuit. (a) The original circuit. (b) With feedback loop broken. (c) Flow table showing next-state function. Circled entries in the flow table are *stable* states.

state doesn't change anything. If the next state is different than the current state, this state is *transient* since as soon as the current state is updated with the next state, the circuit will change states.

For example, suppose the circuit has inputs ab = 00 and the current state is 0. The next state is also 0, so this is a stable state - as shown by the circled 0 in the leftmost position of the top row of the table. If from this state input b goes high, making the input state ab = 01, we move one square to the right in the table. In this case, the 01 AND gate is enabled and the next-state is 1. This is an unstable or transient situation since the current state and next state are different. After some amount of time (for the change to propagate) the current state will become 1 and we move to the bottom row of the table. At this point we have reached a stable state since the current and next state are now both 1.

If there is a cycle of transient states with no stable states we have an *oscillation*. For example, if the inputs to the circuit of Figure 22.2 are ab = 11, the next state is always the complement of the current state. With this input state, the circuit is never stable, but instead will oscillate indefinitely between the 0 and 1 states. This is almost never a desired behavior. An oscillation in an asynchronous circuit is almost always an error.

So, what does the circuit of Figure 22.2 do? By this point the estute reader will have realized that its an RS flip-flop with an oscillation feature added. Input a is the reset input. When a is high and b is low, the state is made 0 when a is lowered the state remains 0. Similarly b is the set input. Making b high while a is low sets the state to 1 and it remains at 1 when b is lowered. The only difference between this flip-flop and the one of Figure 14.2 is that when both inputs are high the circuit of Figure 22.2 oscillates while the circuit of Figure 14.2 resets.

To simplify our analysis of asynchronous circuits we typically insist that the environment in which the circuits operate obey the *fundamental mode* restriction:

> **Fundamental-Mode:** Only one input bit may be changed at a time and the circuit must reach a stable state before another input bit is changed.

A circuit operated in fundamental-mode need only worry about one input bit changing at a time. Multiple-bit input changes are not allowed. Our setup- and hold-time restrictions on flip-flops are an example of a fundamental-mode restriction. The clock and data inputs of the flip flop are not allowed to change at the same time. After the data input changes, the circuit must be allowed to reach a steady-state (setup time) before the clock input can change. Similarly, after the clock input changes, the circuit must be allowed to reach a steady-state (hold time) before the data input can change. We will look at the relation of setup and hold time to the design of the asynchronous circuits that realize flip-flops in more detail in Chapter 23.

In looking at a flow-table, like the one in Figure 22.2, operating in the fundamental mode means that we need only consider input transitions to adjacent

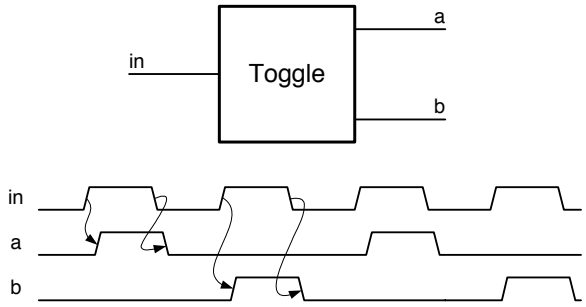


Figure 22.3: A toggle circuit alternates pulses on it input *in* between its two outputs *a* and *b*.

squares (including wrapping from leftmost to rightmost). Thus, we don't have to worry about what happens when the input changes from 11 (oscillating) to 00 (storing). This can't happen. Since only one input can change at a time, we must first visit state 10 (reset) or 01 (set) before getting to 00.

In some real world situations, it is not possible to restrict the inputs to operate in fundamental mode. In these cases we need consider multiple input changes. This topic is beyond the scope of this book and the interested reader is referred to some of the texts listed in Section 22.4.

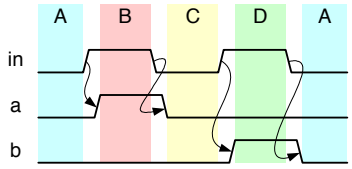
22.2 Flow-Table Synthesis: The Toggle Circuit

We now understand how to use a flow-table to analyze the behavior of an asynchronus circuit. That is, given a schematic, we can draw a flow table and understand the function of the circuit. In this section we will use a flow table in the other direction. We will see how to create a flow table from the specification of a circuit and then use that flow table to synthesize a schematic for a circuit that realizes the specification.

Consider the specification of a toggle circuit - shown graphically in Figure 22.3. The toggle circuit has a single input *in* and two outputs *a* and *b*.[1] Whenever *in* is low, both outputs are low. The first time *in* goes high, output *a* goes high. On the next rising transition of *in*, output *b* goes high. On the third rising input, *a* goes high again. The circuit continues steering pulses on *in* alternately between *a* and *b*.

The first step in synthesizing a toggle circuit is to write down its flow table. We can do this directly from the waveforms of Figure 22.3. Each transition of the input potentially takes us to a new state. Thus, we can partition the waveform into potential states as shown in Figure 22.4. We start in state A. When *in* rises we go to state B where output *a* is high. when *in* falls again we

[1] In practice a reset input *rst* is also required to initialize the state of the circuit.



State	Next (in) 0	Next (in) 1	Out (a,b)
A	(A)	B	00
B	C	(B)	10
C	(C)	D	00
D	A	(D)	01

Figure 22.4: A flow table is created from the specification of the toggle circuit by creating a new state for every input transition until the circuit is obviously back to the same state.

go to state C. Even though C has the same output as A, we know its a different state because the next transition on *in* will cause a different output. The second rising edge on *in* takes us to state D with output *b* high. When *in* falls for the second time we go back to state A. We know that this state is the same as state A since the behavior of the circuit at this point under all possible inputs is indistinguishable from where we started.

Once we have a flow table for the toggle circuit, the next step is to assign binary codes to each of the states. This state assignment is more critical than with synchronous machines. If two states X and Y differ in more than one state bit, a transition from X to Y requires first visiting a *transient* state with one state bit changed before arriving at Y. In some cases, a *race* between the two state bits may result. We discuss races in more detail in Section 22.3. For now, we pick a state assignment (shown in Figure 22.5(a) where each state transition switches only a single bit.

With the state assignment, realizing the logic for the toggle circuit is a simple matter of combinational logic synthesis. We redraw the flow table as a Karnaugh map in Figure 22.5(b). The Karnaugh map shows the symbolic next state function - i.e., each square shows the next state name (A through D) for that input and current state. The arrows show the path through the states followed during operation of the circuit. Understanding this path is important for avoiding races and hazards. We refer to this Karnaugh map showing the state transitions as a *trajectory map since it shows the trajectory of the state variables.*

We redraw the Karnaugh map with state names replaced by their binary codes in Figure 22.5(c), and separate maps for the two state variables s_0 and s_1

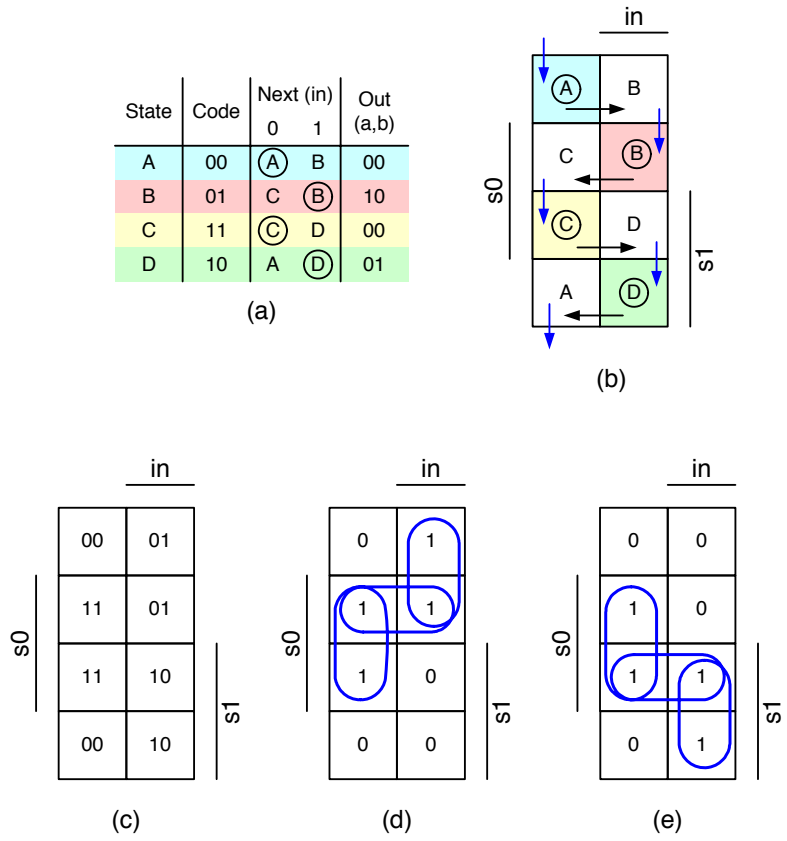


Figure 22.5: Implementing the toggle circuit from its flow table. (a) Flow table with state assignment. (b) Flow table mapped to Karnaugh map. (c) Next state codes mapped to Karnaugh map. (d) Karnaugh map for s0. (e) Karnaugh map for s1.

are shown in Figure 22.5(d) and (e) respectively. From these Karnaugh maps we write down the equations for s_0 and s_1:

$$s_0 = (\overline{s_1} \wedge in) \vee (s_0 \wedge \overline{in}) \vee (s_0 \wedge \overline{s_1}), \quad (22.1)$$

$$s_1 = (s_1 \wedge in) \vee (s_0 \wedge \overline{in}) \vee (s_0 \wedge s_1). \quad (22.2)$$

The last implicant in each expression is required to avoid a hazard that would otherwise occur. Asynchronous circuits must be hazard free along their path through the input/state space. Because the current state is being constantly fed back a glitch during a state transition can result in the circuit switching to a different state - and hence not implementing the desired function. For example, suppose we left the $s_0 \wedge \overline{s_1}$ term out of (22.2). When *in* goes low in state B, s_0 might go momentarily low before s_1 comes high. At this point the middle term of both equations becomes false and s_1 never goes high - the circuit goes to state A rather than C.

All that remains to complete our synthesis is to write the output equations. Output a is true in state 01 and output b is true in state 10. The equations are thus:

$$a = \overline{s_1} \wedge s_0, \quad (22.3)$$

$$b = s_1 \wedge \overline{s_0}. \quad (22.4)$$

22.3 Races and State Assignment

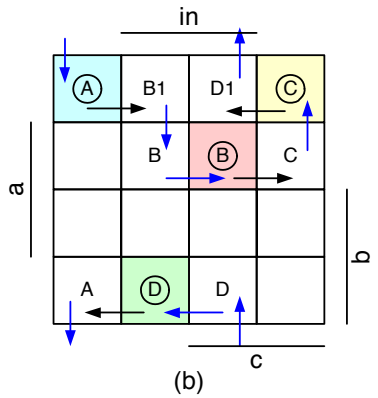
To illustrate the problem of multiple state variables changing simultaneously, consider an alternate state assignment for the toggle circuit shown in Figure 22.6(a). Here we observe that the two outputs, a and b can also serve as state variables, so we can add to the outputs just one additional state variable c to distinguish between states A and C giving the codes shown in the figure.[2]

With this state assignment, the transition from state A ($cab = 000$) to state B (110) changes both c and a. If the logic is designed so that *in* going high in state A makes both c and a go high, they could change in an arbitrary order. Variable a could change first, variable c could change first, or they could change simultaneously. If they change simultaneously, we go from state A directly to state B with no intermediate stops. If a changes first, we go first to state 010 which is not assigned and then, if the logic in state 010 does the right thing, to state 110. If c changes first, the machine will go to state C (100) where the high input will then drive it to state D. Clearly, we cannot allow c to change first. This situation where multiple state variables can change at the same time is called a *race*. The state variables are *racing* to see which one can change first.

[2]Note that the bit ordering of the codes is c,a,b.

State	Code (c,a,b)	Next (in) 0	Next (in) 1	Out (a,b)
A	000	(A)	B	00
B	110	C	(B)	10
C	100	(C)	D	00
D	001	A	(D)	01

(a)



(b)

Figure 22.6: An alternate state assignment for the toggle circuit requires multiple state variables to change on a single transition. (a) The flow table with the revised state assignment. (b) A trajectory map showing the introduction of transient states B1 = 010 and D1 = 101.

When the outcome of the race affects the end state - as in this case - we call the race a *critical race*.

To avoid the critical race that could occur if we allow both a and c to change at the same time, we specify the next-state function so that only a can change in state A. This takes us to a *transient state* 010, which we will call B1. When the machine reaches state B1, the next state logic then enables c to change.

The introduction of this transient state is illustrated in the trajectory map of Figure 22.6(b). When the input goes high in state A, the next state function specifies B1 rather than B. This enables only a single transition, downward as shown by the blue arrow - which corresponds to a rising, to state B1. A change in c is not enabled in this state to avoid a horizontal transition into the square marked D1. Once the machine reaches state B1, the next state function becomes B which enables the change in c, a horizontal transition, to stable state B.

A transient state is also required for the transition from state C 100 to state D 001. Both variables c and b change between these two states. An uncontrolled race in which variable c changes first could wind up in state A 000 which is not correct. To prevent this race, we enable only b to change when *in* rises in state C. This takes us to a transient state D1 (101). Once in state D1, c is allowed to fall, taking us to state D (001).

Figure 22.7 illustrates the process of implementing the revised toggle circuit. Figure 22.7(a) shows a Karnaugh map of the next-state function. Each square of the Karnaugh map shows the code for the *next* state for that present state and input. Note that where the next state equals the present state the state is stable. Transient state B1 (at 010 - the second square along the diagonal) is not stable since it has a next state of 110.

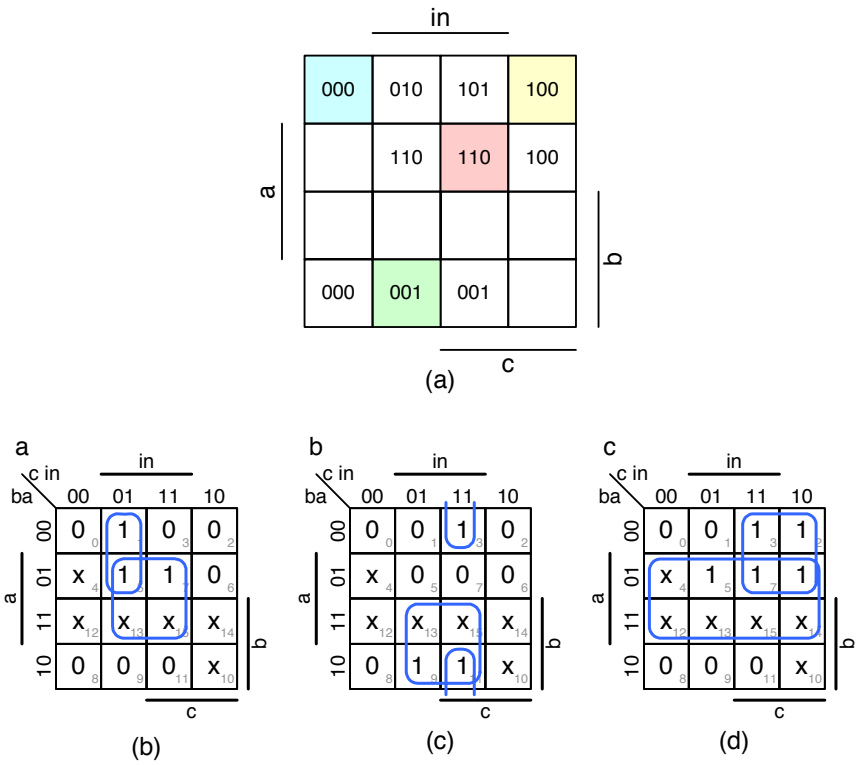


Figure 22.7: Implementation of the toggle circuit with the alternate state assignment of Figure 22.6(a). (a) Karnaugh map showing 3-bit next state function c,a,b. (b) Karnaugh map for a. (c) Karnaugh map for b. (d) Karnaugh map for c.

From the next-state Karnaugh map we can write the individual Karnaugh maps for each state variable. Figure 22.7 (b) through (d) show the Karnaugh maps for the individual variables - a, b, and c respectively. Note that the states that are not visited along the state trajectory (the blank squares in Figure 22.7(a)) are don't cares. The machine will never be in these states, thus we don't care what the next state function is in an unvisited state.

From these Karnaugh maps we write the state variable equations as:

$$a = (in \wedge \overline{b} \wedge \overline{c}) \vee (in \wedge a), \tag{22.5}$$

$$b = (in \wedge \overline{a} \wedge c) \vee (in \wedge b), \tag{22.6}$$

$$c = a \vee (\overline{b} \wedge c). \tag{22.7}$$

Note that we don't require separate equations for output variables since a and b are both state variables and output variables.

22.4 Bibliographic Notes

many early computer designs were completely asynchronous

over time, the simplicity of synchronous design won out.

today async is used mostly for flip-flops and interfaces to inherently asynchronous processes.

there is an active research area in asynchronous design - async conferences.

texts that give more detail on the field include Unger and Kohavi

22.5 Exercises

22–1 Analysis. write a flow table for a schematic. State what the circuit does.

22–2 Synthesis. write a flow table for a proportional phase comparator. Synthesize a gate-level circuit from your flow table.

22–3 *Toggle Synthesis.* Synthesize a version of the toggle circuit where the state assignments are as in Figure 22.6(a) except that state B is encoded as *cab* = 010.

22–4 *Edge Toggle.* The toggle circuit of Section 22.2 is a *pulse toggle* — a circuit in which alternate pulses on the input alternate between the two outputs. For this exercise you are to design an *edge toggle* — a circuit in which edges on the input cause edges that alternate between the two outputs.

22–5 *Three-Way Toggle.* Design a toggle circuit like the one in Section 22.2 except that pulses on the input alternate over three outputs.

22–6 *Three-Way Edge Toggle.*

22–7 State Reduction. Write a flow table for the toggle circuit but this time creating a new state for each of the first eight transitions on *in*. Then determine which states are equivalent to reduce your flow table to a four-state table.

22–8 Races.

Chapter 23

Flip Flops

Flip-flops are among the most critical circuits in a modern digital system. As we have seen in previous chapters, flip flops are central to all synchronous sequential logic. Registers (built from flip-flops) hold the state (both control and data state) of all of our finite-state machines. In addition to this central role in logic design, flip-flops also consume a large fraction of the die area, power, and cycle time of a typical digital system.

Up until now, we have considered a flip-flop as a black box.[1] In this chapter we *look inside* the flip-flop. We derive the logic design of a typical D-flip-flop and show how the timing properties introduced in Chapter 15 follow from this design.

We first develop the flip-flop design informally - following an intuitive argument. We start by developing the latch. The implementation of a latch follows directly from its specification. From the implementation we can then derive the setup, hold, and delay times of the latch. We then see how to build a flip-flop by combining two latches in a master-slave arrangement. The timing properties of the flip-flop can then be derived from its implementation.

Following this informal development, we then derive the design of a latch and flip-flop using flow-table synthesis. This serves both to reinforce the properties of these storage elements and to give a good example of flow-table synthesis. We introduce the concept of *state equivalence* during this derivation. This formal derivation can be skipped by a casual reader.

23.1 Inside a Latch

A schematic symbol for a latch is shown in Figure 23.1(a) and waveforms illustrating its behavior and timing are shown in Figure 23.1(b). A latch has two inputs data d and enable g, and one output q. When the enable input is high,

[1] A *black box* is a system that we understand the external specifications, but not the internal implementation - as if the system were inside an opaque (black) box that keeps us from seeing how it works.

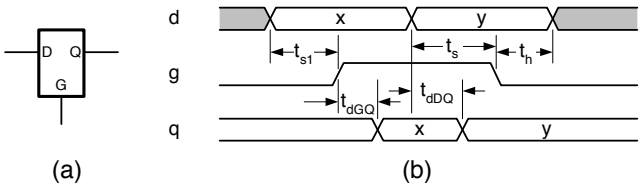


Figure 23.1: A latch. (a) Schematic symbol. (b) Waveforms showing timing properties.

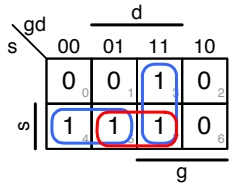


Figure 23.2: Karnaugh map for a latch showing hazard.

the output follows the input. When the enable input is low, the output holds its current state.

As shown in Figure 23.1(b) a latch, like a flip-flop has a setup time t_s and a hold time t_h. An input must be setup t_s before the enable *falls* and held for t_h after the enable falls for the input value to be correctly stored. Latch delay is characterized by both delay from the enable rising to the output changing, t_{dGQ}, and delay from the data input changing to the output changing, t_{dDQ}. For the enable to dominate the delay, the input must be setup at least t_{s1} before the enable rises. Usually, this is just a question of which signal (d or g) is on the critical path and $t_{s1} = t_{dDQ} - t_{dGQ}$. As we shall see below, these times can be derived from the logic design of the latch and the need to meet the fundamental-mode restriction (Section 22.1) during latch operation.

From the description of a latch, we can write down its logic equation:

$$q = (g \wedge d) \vee (\overline{g} \wedge q). \tag{23.1}$$

That is, when g is true, $q = d$, and when g is false, q holds its state ($q = q$). This is almost correct. As can be seen from the Karnaugh map of Figure 23.2, there is a hazard that may occur when g changes state and both d and q are high. To cover this hazard, we must add an additional implicant to the equation.

$$q = (g \wedge d) \vee (\overline{g} \wedge q) \vee (d \wedge q). \tag{23.2}$$

From Equation (23.2) we can draw a gate-level schematic for a latch as shown in Figure 23.3(a). This implementation of a latch is often called an *Earle* latch

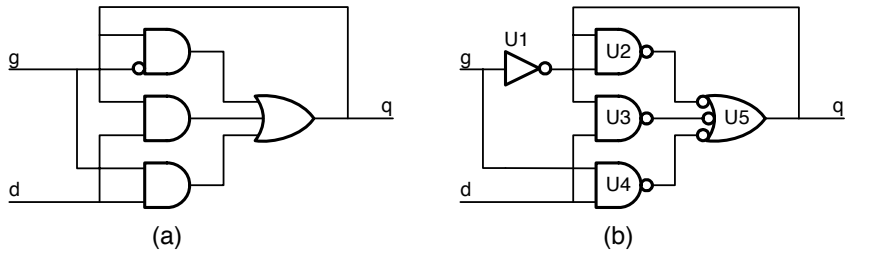


Figure 23.3: Gate-level schematic of an Earle latch. (a) Using abstract AND and OR gates. (b) CMOS implementation using only inverting gates.

after its original developer. For a CMOS implementation, we can redraw the Earle latch using only inverting gates (an inverter and four NAND gates) as shown in Figure 23.3(b).

From the schematic of Figure 23.3 we can now derive the timing properties of the latch. Let the delay of gate Ui in the figure be t_i - in practice we would calculate the delay of these gates as described in Section 5.4 — and the delays may be different for rising and falling edges and for different states. First consider the setup time, t_s. To meet the fundamental-mode restriction, after changing input d, the circuit must be allowed to reach a stable state before input g falls. For the circuit to reach a stable state the change in d (rising or falling) must propagate through gates U4, U5, and U3 in that order and both state variable q and the output of gate U3 must reach a stable state before g falls. This delay, and hence the setup time, is calculated as:

$$t_s = t_4 + t_5 + t_3. \tag{23.3}$$

This setup calculation illustrates some of the subtlety of asynchronous circuit analysis. One might assume that since d is connected directly to U3, the path from d to the output of U3 would be direct — d to U3. However, this is not the case when d is low. If d is low and the circuit is stable, then q is low. Thus, when d rises, it does not switch U3, because q, the other input of U3, is low. The change on d has to propagate through U4 and U5, making q high, before U3 switches. Hence the path that must stabilize is d to U4 to U5 to U3.

Now consider hold time. After g falls, the circuit must again come to a steady state before d can change again. In particular, if d is high, the change on g must propagate through U1 and U2 to enable the loop of U2 and U5 before d is allowed to fall. Hence the hold time is just:

$$t_h = t_1 + t_2. \tag{23.4}$$

To complete our analysis we see that each propagation delay is just the delay from an input g or d to the output q. For t_{dDQ} this path includes gates U3, U4, and U5, and for t_{dGQ} the path is through U4 and U5 (recall we are only

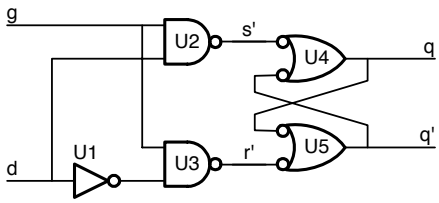


Figure 23.4: Latch built from an RS flip-flop.

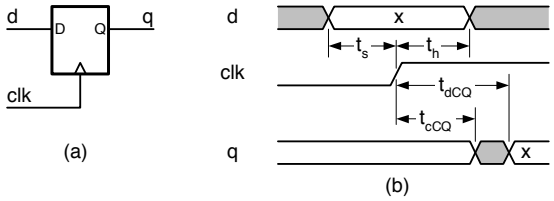


Figure 23.5: Edge-triggered D-flip flop. (a) Schematic symbol. (b) Timing diagram showing behavior.

worried about g rising here). Thus we have:

$$t_{dDQ} = \max(t_3, t_4) + t_5, \tag{23.5}$$

$$t_{dGQ} = t_4 + t_5. \tag{23.6}$$

An alternate gate implementation of a latch is shown in Figure 23.4. Here we construct a latch by appending a gating circuit to an RS flip-flop (Section 14.1). The RS flip-flop is formed by NAND gates U4 and U5. When the upper input of U4, $\overline{s}$, is asserted (low), the flip-flop is set ($q = 1$,$\overline{q} = 0$). When the lower input of U5, Br, is asserted (low), the flip-flop is reset ($q = 0$,$\overline{q} = 1$).

The gating circuit, formed by U1, U2, and U3, sets the flip-flop when $g = 1$ and $d = 1$, and resets the flip-flop when $g = 1$ and $d = 0$. Thus, when g is high, the output q follows input d. When g is low, the flip-flop is neither set nor reset and holds its previous state.

While the latch of Figure 23.4 has identical logical behavior to the Earle latch of Figure 23.3, it has very different timing properties. We leave derivation of these timing properties as Exercise 23–1.

23.2 Inside a Flip-Flop

An edge-triggered D-type flip-flop updates its output with the current state of its input on the rising edge of the clock. At all other times the output holds its current state. The schematic symbol and timing diagram of a D-flip-flop are

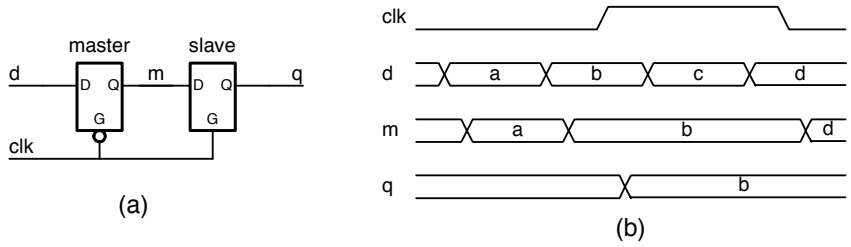


Figure 23.6: A master-slave D-flip-flop is constructed from two latches.

shown in Figure 23.5 (repeated from Figure 15.4). In Section 14.2 we saw how D-flip-flops are used for the state registers of synchronous sequential circuits and in Chapter 15 we looked at the detailed timing properties of the D-flip-flop. In this section, we derive a logic design for the D-flip-flop and see how this logic design gives rise to the timing properties of the flip-flop.

A latch with a negated enable does half of what we need to make a flip-flop. It samples its input when the enable rises and holds the output steady while the negated enable is high. The problem is that its output follows its input when the negated enable is low. We can use a second latch, in series with the first, to correct this behavior. This latch, with a normal enable, prevents the output from changing when the enable is low.

A D-flip-flop implemented with two latches in series with complemented enables is shown in Figure 23.6(a). Waveforms illustrating the operation of the flip flop are shown in Figure 23.6(b). The first latch, called the *master* samples the input on the rising edge of the clock onto intermediate signal m. In the waveforms, value b is sampled and held steady on m while the clock is high. Signal m, however follows the input when the clock is low. The second latch, called the *slave* (because it follows the master), enables the data captured on signal m onto the output when the clock is high. When the clock falls, it samples this value - still held steady by the master - and holds this value on the output when the clock is low. The net result of the two latches is a device that acts as a D-flip-flop. It samples the data on the rising edge of the clock and holds it steady until the next rising edge. The master holds the value when the clock is high - while the slave is transparent, and the slave holds the value when the clock is low - while the master is transparent.

For correct operation of the master-slave flip-flop, it is critical that the output of the master not change until t_h after the slave clock falls. That is, the hold time constraint of the slave must be met. If the output of the master latch were to change too quickly after the clock falls, the new value on intermediate signal m (d in the figure) could race through to the output before the slave latch blocks its flow. In practice this is rarely a problem unless there is a large amount of clock skew (Section 15.4) between the master and slave latches.

From the schematic of Figure 23.6(a) and the timing properties of the latch,

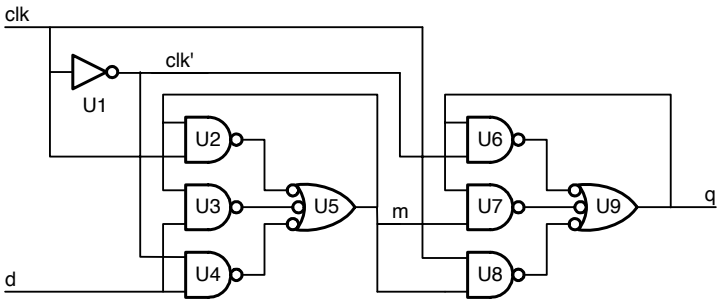


Figure 23.7: Gate-level schematic of a master-slave D-flip-flop.

we can derive the timing properties of the flip-flop. The setup and hold times of the flip-flop are just the setup and hold times of the master latch. This is the device doing the sampling. The delay of the flip-flop, t_{dCQ} is the delay of the slave latch t_{dGQ}.[2] The output q changes to the new value t_{dGQ} after the clock rises.

To illustrate the timing parameters of the D-flip-flop in a more concrete way, consider the gate-level schematic of Figure 23.7. This figure shows the master-slave flip-flop of Figure 23.6 expanded out to the gate level. The inverter on the enable input of each latch has been factored out to a single inverter U1 that generates $\overline{clk}$. The polarity of the clock connection to the master and slave are opposite.

The setup time t_s of the flip-flop shown in Figure 23.7 is the amount of time required for the circuit to reach a steady-state after d changes when clk is low.

Strictly speaking, the path here is through U4, U5, and U3 or U7 (if q is high). However, we don't really care if the output of U7 reaches steady-state or not since the clock is about to go high enabling U8. So we ignore U7 and focus on the path that ends at the output of U3 giving:

$$t_s = t_4 + t_5 + t_3 = t_{sm}. \tag{23.7}$$

where t_{sm} is the setup time of the master latch.

The hold time of this flip-flop is the time for the master part of the circuit to settle after the clock rises. This is just the maximum of delay of gates U1 and U2. If the data changes before U1 has brought $\overline{clk}$ low, state m might be affected. Similarly, the data cannot change until the output of U2 has stabilized or we may loose the value stored in the master latch. Thus we have:

$$t_h = \max(t_1, t_2) = t_{hm}. \tag{23.8}$$

where t_{hm} is the hold time of the master latch. Compared to (23.4) we don't

[2]This assumes that the setup time is large enough so that signal m is stable t_{s1} before the clock rises. See Exercise **??** for an example where this is not the case.

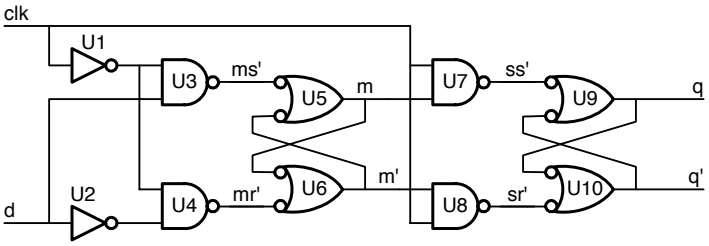


Figure 23.8: Alternate gate-level schematic of a master-slave D-flip-flop.

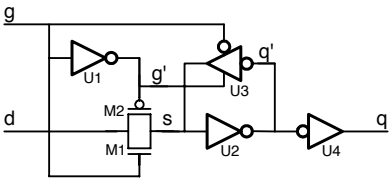


Figure 23.9: CMOS latch circuit using a transmission gate and a tri-state inverter.

add t_1 to t_2 here because with the low-true enable of the master latch, U1 is not in the path from *clk* to the output of U2.

For hold-time calculation we are only concerned with the master latch reaching steady state. When the clock rises, it may take longer for the slave to reach steady state. However, we will capture the data reliably as long as the master reaches steady state.

Finally, t_{dCQ} is the delay from the clock rising to the output of the flip-flop. The path here is through U8 and U9 giving:

$$t_{dCQ} = t_8 + t_9 = t_{dGQs}. \tag{23.9}$$

where t_{dGQs} is t_{dGQ} for the slave latch.

An alternate gate-level schematic for a master-slave D-flip-flop is shown in Figure 23.8. This circuit uses the RS-flip-flop based latch of Figure 23.4 for the master and slave latches of the flip flop. We leave the derivation of t_s, t_h, and t_{dCQ} for this flip-flop as Exercise 23–2

23.3 CMOS Latches and Flip-Flops

The latch circuits of either Figure 23.3 or Figure 23.4 using static CMOS gates. CMOS technology, however, also permits us to construct a latch with a transmission gate and a tri-state inverter as shown in Figure 23.9. Most CMOS

latches use transmission gates in this style because it results in a latch that is both smaller and faster than the alternative gate circuits.

When enable g is high (and $\overline{g}$ is low), the transmission gate formed by NFET M1 and PFET M2 is *on* allowing the value on input d to pass to storage node s. If $d = 1$, PFET M2 passes the 1 from d to s, and if $d = 0$, NFET M1 passes the 0 from d to s. Output q follows storage node s buffered by inverters U2 and U4. Thus, when g is high, the output q follows input d.

When enable g goes low the transmission gate formed by M1 and M2 turns off isolating storage node s from the input. At this time the input is sampled onto the storage node. At the same time, tri-state inverter U3 turns on, closing a storage loop from s back to s through two inverters. This feedback loop reinforces the stored value, allowing it to be retained indefinitely. The tri-state inverter is equivalent to an inverter followed by a transmission gate.

We can calculate the setup, hold, and delay times of the CMOS latch in the same manner as we did for the gate-based latches. When the input changes while g is high, the effect of the change on the storage loop must settle out before g is allowed to fall. The input change must propagate through the transmission gate, and inverter U2. We need not wait for U4 to drive output q since that doesn't affect the storage loop. Thus, the setup time is:

$$t_s = t_g + t_2, \tag{23.10}$$

where t_g is the delay of the transmission gate.

After g goes low, we need to hold the value on input d until the transmission gate is completely shut off. This is just the delay of inverter U1:

$$t_h = t_1. \tag{23.11}$$

We do not need to wait for feedback gate U3 to turn on. Its output, storage node s, is already in the correct state and there is sufficient capacitance on node s to hold its value steady until U3 turns on.

The delay times are calculated by tracing the path from input to output:

$$t_{dGQ} = t_1 + t_g + t_2 + t_4. \tag{23.12}$$

$$t_{dDQ} = t_g + t_2 + t_4. \tag{23.13}$$

A CMOS flip-flop can be constructed from two CMOS latches as shown in Figure 23.10. The master latch, formed by NFET M1, PFET M2, inverter U2, and tri-state inverter U3, connects input d to storage node m when enable g is low and holds m when g is high The slave latch, formed by NFET M3, PFET M4, inverter U4, and tri-state inverter U5 connects master latch state $\overline{m}$ to storage node $\overline{s}$ when enable g is high. An additional inverter, U6 generates output q. Output q is logically identical to node s at the output of U4. Isolating the output from the storage loop in this manner, however, is critical for the synchronization properties of the flip-flop as discussed in Chapter **??**. We leave the analysis of this flip-flop as Exercise 23–3.

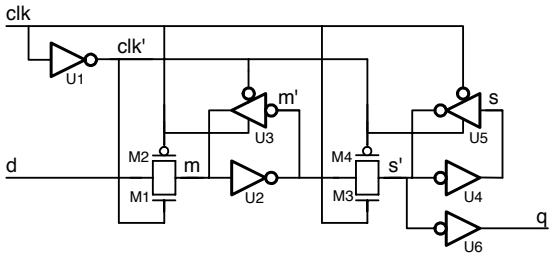


Figure 23.10: CMOS flip-flop circuit is constructed from two CMOS latches.

23.4 Flow-Table Derivation of The Latch*

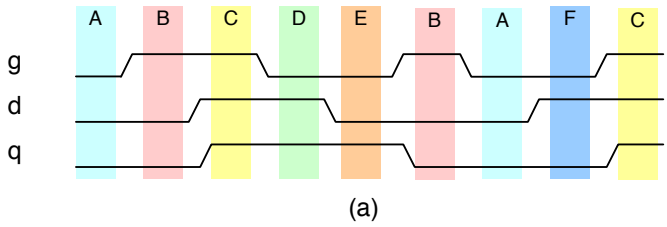
The latch and flip-flop are themselves asynchronous circuits and can be synthesized using the flow-table technique we developed in Chapter 22. Given an English-language description of a latch, we can write a flow-table for the latch as shown in Figure 23.11. To enumerate the states, we start with one state - all inputs and outputs low (state A) - and from this state toggle each input. We then repeat this process from each new state until all possible states have been explored.

In state A, toggling g high takes us to state B and toggling d high takes us to state F. This gives us the first line of the flow table of Figure 23.11(b). In state B, toggling g low takes us back to state A and toggling d high takes us to state C with q high. This gives us the second line of the flow table. We continue this way constructing the flow table line by line. until all states have been explored. The end result is the table of Figure 23.11(b).

In constructing this flow table we have created a new state for each input combination that didn't obviously match a previous state. This gives us six states, A-F, which would take three state variables to represent. However, many of these states are *equivalent* and can be combined. Two states are equivalent if they are indistinguishable from the inputs and outputs of the circuit.

We define equivalence recursively. Two states are *0-equivalent* if they have the same output for all input combinations. Here states A, B, and F are 0-equivalent as are states C, D, and E. A two states are *k-equivalent* they have the same output for all input combinations and their next-states for each input combination are $k-1$-equivalent. Here we see that the next states for A, B, and F aren't just equivalent, they are the same, so A, B, and F are also 1-equivalent. Similarly for C, D, and E.

In practice we find equivalent states by forming sets of states that are 0-equivalent (e.g., {A,B,F} and {C,D,E}) and then sets of states that are 1-equivalent, and so on. As soon as the sets don't change. That is, when we find a set of states that is both k-equivalent and $k+1$-equivalent we're done - these sets are equivalent.



(a)

State	Next (g d) 00	01	11	10	Out (q)
A	(A)	F		B	0
B	A		C	(B)	0
C		D	(C)	B	1
D	E	(D)	C		1
E	(E)	D		B	1
F	A	(F)	C		0

(b)

State	Next (clk d) 00	01	11	10	Out (q)
A	(A)	(A)	C	(A)	0
C	(C)	(C)	(C)	A	1

(c)

Figure 23.11: Flow table synthesis of a latch. (a) Waveforms, (b) Flow table, (c) Reduced flow table, (d) Karnaugh map, (e) Schematic.

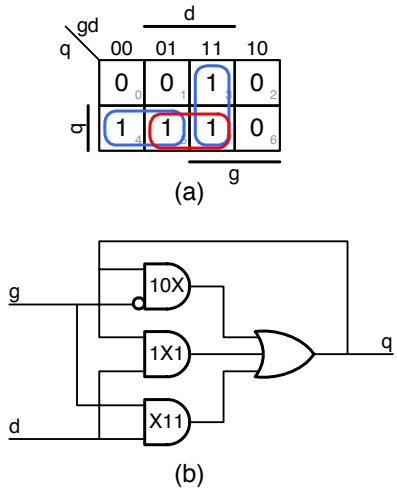


Figure 23.12: Logic design of a latch. (a) Karnaugh map. (b) Schematic diagram. The 1X1 implicant is needed to prevent a hazard.

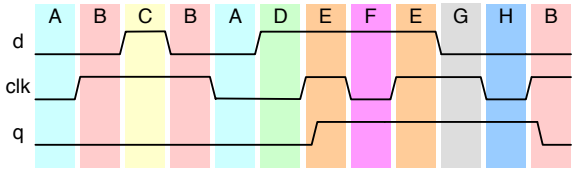
In this case, states A, B, and F are equivalent and C, D, and E are equivalent. Rewriting the flow table with just two states A and C (one for each equivalence class) gives us the reduced flow table of Figure 23.11(c).

If we use the output as the state variable, assigning state A an encoding of 0 and state C an encoding of 1, we can rewrite the reduced flow table of Figure 23.11(c) as the Karnaugh map of Figure 23.12(a). There are three implicants shown on the Karnaugh map. While we can cover the function with just two implicants (qgd = X11 $\vee$ 10X), we need the third, 1X1, to avoid a hazard (see Section 6.10). If the output were to momentarily dip low when the enable g input falls (going from qgd = 111 to 101), the end state could be 001 - with the latch loosing the stored 1. The added implicant (1X1) avoids this hazard. A schematic diagram for the latch circuit is shown in Figure 23.12(b).

23.5 Flow-Table Synthesis of a D-Flip-Flop*

Figure 23.13 shows the derivation of a flow table for a D-type flip-flop. We start by showing a waveform that visits all eight states shown in the flow table. As with the latch, we construct the flow table by toggling each input (clk and d) in each state. We create a new state for each input combination unless it is obviously equivalent to a state that we have already visited. The resulting flow table has eight states and is shown in Figure 23.13(b).

The next step is to find state equivalence classes. We start by observing that the 0-equivalent sets are $\{A, B, C, D\}$ and $\{E, F, G, H\}$. For 1-equivalence



(a)

State	Next (clk d) 00	01	11	10	Out (q)
A	(A)	D		B	0
B	A		C	(B)	0
C		D	(C)	B	0
D	A	(D)	E		0
E		F	(E)	G	1
F	H	(F)	E		1
G	H		E	(G)	1
H	(H)	F		B	1

(b)

State	Code	Next (clk d) 00	01	11	10	Out (q)
ABC	00	(A)	D	(C)	(B)	0
D	01	A	(D)	E		0
EFG	11	H	(F)	(E)	(G)	1
H	10	(H)	F		B	1

(c)

Figure 23.13: Flow-table synthesis of an edge-triggered D-flip-flop. (a) Waveforms. (b) Flow table. (c) Reduced flow table.

we observe that the next-state of D with an input of 11 is in a different class than that of the rest of its set. Similarly for H with an input of 10. Thus, the 1-equivalent sets become $\{A, B, C\}, \{D\}, \{E, F, G\}$, and $\{H\}$.

A reduced flow table with just these four states is shown in Figure 23.13(c). Our output q will be one of our state variables. We assign a second state variable as shown in the *Code* column of the table. Here q is the MSB of the two-bit state code and our new variable is the LSB.

Figure 23.14 shows how the flow-table of Figure 23.13(c) is reduced to logic. We start by redrawing the flow-table as a Karnaugh map with symbolic entries in Figure 23.14(a). Variable q is the output, and variable x is our additional state variable. The next step is to replace the symbolic next states with their state codes qx as shown in Figure 23.14(b). We next write down the Karnaugh maps for the two state variables in Figure 23.14(d) - q is on the left, x is on the right.

The two next-state variable functions can each be covered with two implicants. As with the latch, we must add a third to each to avoid a hazard. For variable q we have implicants $qxcd$ = X11X, 1X0X, and, to elimnate the hazard, 11XX. For x we have implicants X11X, XX01, and, for the hazard, X1X1. Drawing out these implicants as the logic diagram of Figure 23.14(d), we see that we have synthsized the master-slave D-flip-flop using Earle latches of Figure 23.7. The astute reader will notice that the top AND gate of the master latch and the bottom AND gate of the slave latch have identical inputs and can be replaced by a single shared AND gate.

23.6 Bibliographic Notes

Earle latch

Survey of FF circuits

23.7 Exercises

23–1 *Latch Timing Properties.* Compute t_s, t_h, t_{dDQ}, and t_{dGQ} for the latch of Figure 23.4 in terms of the delays of the individual gates. Assume that the delay of gate U_i is t_i.

23–2 *Flip-Flop Timing Properties.* Compute t_s, t_h, and t_{dCQ} for the D-flip-flop of Figure 23.8 in terms of the delays of the individual gates. Assume that the delay of gate U_i is t_i.

23–3 *CMOS Flip-Flop Timing Properties.* Compute t_s, t_h, and t_{dCQ} for the D-flip-flop of Figure 23.10 in terms of the delays of the individual gates (and transmission gates). Assume that the delay of gate U_i is t_i and the delay of each transmission gate is t_g.

23–4 *Flip-Flop Contamination Delay.* Consider a flip-flop constructed by placing a delay line with delay t_d between the master and slave latches. When

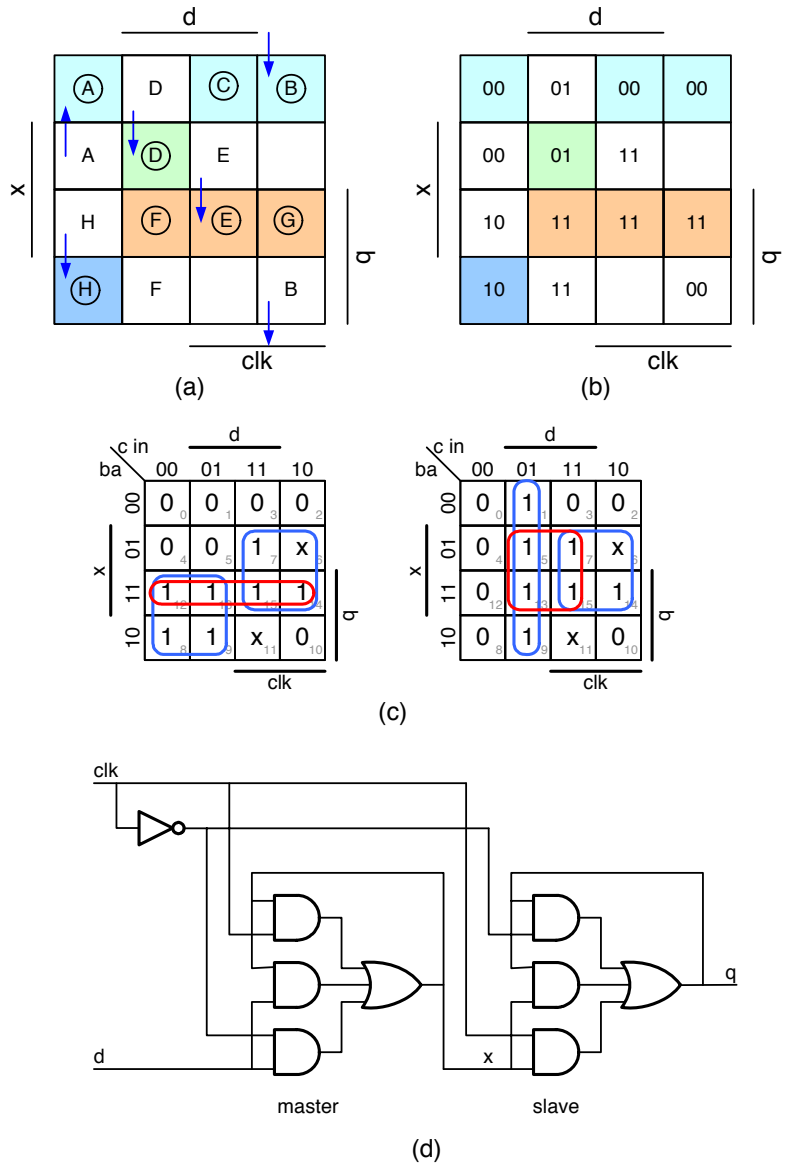


Figure 23.14: Deriving the logic design of a D-flip-flop from the flow-table of Figure 23.13(c). (a) Flow-table drawn on a Karnaugh map. (b) Karnaugh map showing next-state function. (c) Karnaugh maps for each bit of next-state function. (d) Logic diagram derived from Karnaugh maps.

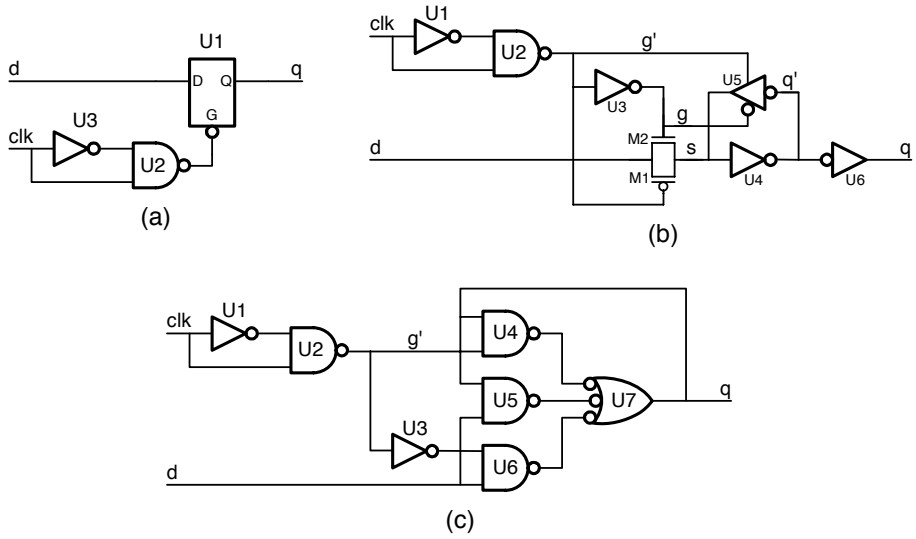


Figure 23.15: A pulsed-latch D-type flip-flop. (a) Schematic using latch symbol. (b) Schematic showing internals of CMOS latch. (c) Schematic showing internals of an Earle latch. When *clk* rises, NAND gate U2 generates a narrow low-going pulse on $\overline{g}$ that enables the latch to sample d.

the flip-flop input d changes exactly t_s before the clock rises, signal m at the input of the delay line becomes valid exactly at the clock edge, and signal $m1$ at the output of the delay line and input to the slave latch becomes valid t_d after the rising edge of the clock. What is the contamination delay t_{cCQ} and propagation delay t_{dCQ} of this modified flip-flop.

23–5 *Pulsed-Latch Flip-Flop.* One of my favorite D-flip-flop designs is shown in Figure 23.15. This flip flop consists of a single latch that is gated by a pulse generator. When *clk* rises, a narrow pulse is generated on enable $\overline{g}$ to sample d. The latch then holds the sampled value until the next rising edge of the clock. Answer the following questions about the circuit of Figure 23.15(c). Assume that the delay of gate U_i is t_i.
(a) What is the minimum pulse width generated by U2 for which this circuit will work properly?
(b) What is the maximum pulse width generated by U2 (if any) for which this circuit will work properly?
(c) Assuming that the pulse width out of U2 is the value you computed in part (a), compute t_s, t_h, and t_{dCQ} for this flip-flop and compare these values to the timing parameters of the master-slave flip-flop analyzed in the text. In particular compare the *overhead*, t_s+t_{dCQ} of the two flip-flops.

23–6 *CMOS Pulsed-Latch Flip-Flop.* Repeat Exercise 23–5 but for the CMOS pulsed latch of Figure 23.15(b).

23–7 *Combining Logic and Storage.*

23–8 *Two-Phase Clocking.*

23–9 *State equivalence.* Non-trivial state equivalence problem.

23–10 **SPICE analysis of setup and hold times.*

Chapter 24

Metastability and Synchronization Failure

What happens when we violate the setup- and hold-time constraints of a flip-flop? Up until now, we have considered only the normal behavior of a flip-flop when these constraints are satisfied. In this chapter we investigate the abnormal behavior that occurs when we violate these constraints. We will see that violating setup and hold times may result in the flip-flop entering a *metastable* state in which its state variable is neither a one or a zero. It may stay in this metastable state for an indefinite amount of time before arriving at one of the two stable states (zero or one). This *synchronization failure* can lead to serious problems in digital systems.

To stretch an analogy, flip-flops are a lot like people. If you treat them well, they will behave well. If you mistreat them, they behave poorly. In the case of flip-flops, you treat them well by observing their setup and hold constraints. As long as they are well treated, flip-flops will function properly never missing a bit. If, however you mistreat your flip-flop by violating the setup and hold constraints, it may react by misbehaving — staying indefinitely in a metastable state. This chapter explores what happens when these good flip-flops go bad.

24.1 Synchronization Failure

When we violate the setup or hold time constraint of a D-flip-flop we can put the internal state of the flip-flop into an *illegal* state. That is, the internal nodes of the flip-flop can be left at a voltage that is neither a 0 nor a 1. If the output of the flip-flop is sampled while it is in this state the result is indeterminate and possibly inconsistent. Some gates may see the flip-flop output as a 0 while others may see it as a 1 and still others may propagate the indeterminate state.

Consider the following experiment with a D-flip-flop: Initially both d and clk are low. During our experiment they both rise. If signal d rises t_s before clk the output q will be 1 at the end of the experiment. If signal clk rises t_h

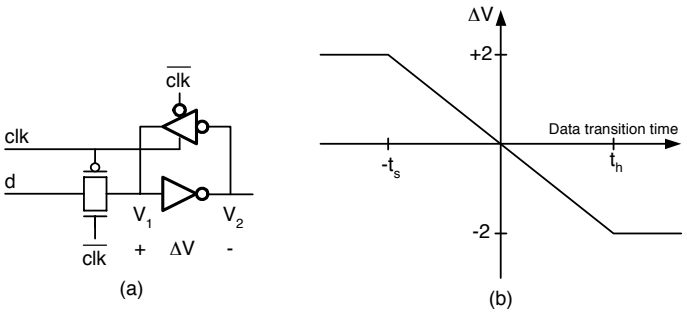


Figure 24.1: Abnormal operation of the master latch of a flip flop. (a) Schematic of the latch. (b) State voltage vs. data transition time. As the data transition time sweeps from t_s before the clock to t_h after the clock, the state voltage – i.e., the voltage across the storage inverters ($\Delta V = V_1 - V_2$) – changes from +2 to -2.

before d, the output q will be 0 at the end of the experiment. Now consider what happens as we sweep the rise time of d relative to clk from t_s before clk to t_h after clock. Somewhere during this interval, the output q at the end of our experiment changes from 1 to 0.

To see what happens when we change the input during this *forbidden interval*, consider the master latch of a CMOS D-flip-flop as shown in Figure 24.1(a). Here we assume that the supply voltage is 1V, so a logic 1 is 1V and a logic 0 is 0V. Figure 24.1(b) shows the initial state voltage of the flip-flop (the voltage across the inverter $\Delta V = V_1 - V_2$ the instant after the clock rises) as a function of data transition time. If d rises at least t_s before clk, the node lableled V_1 is fully charged to 1V and the node labeled V_2 is fully discharged to 0V before the clock falls. Thus the state voltage $\Delta V = V_1 - V_2$ is 2V. As d changes later - the state voltage is reduced as shown in Figure 24.1(b). At first, V_1 is still fully charged, but V_2 doesn't have time to fully discharge. As d rises still later, V_1 doesn't have time to fully charge. Finally, when d rises t_h after the clock V_1 doesn't have time to charge at all and we have $V_1 = 0$ and $V_2 = 1$ so $\Delta V = -2$.

The initial state voltage ΔV is a continuous function of the data transition time as it sweeps from $-t_s$ to t_h. It may not be an exact linear function as shown in the figure, but it is continuous, and it does cross zero at some point during this interval. Over this entire interval, the flip-flop has an initial state voltage that is not +2 or -2, its output is not a fully restored digital signal and may be misinterpreted by following stages of logic.

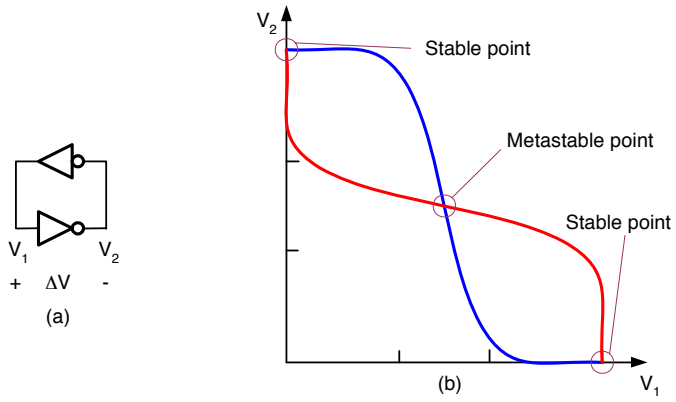


Figure 24.2: (a) The input latch of a flip-flop with the clock high acts as two back-to-back inverters. (b) DC transfer characteristics of the back-to-back inverters. The blue curve shows $V_2 = f(V_1)$ and the red curve shows $V_1 = f(V_2)$. The system has two stable points and one metastable point.

24.2 Metastability

The good news about synchronization failure is that most of the illegal states that our flip-flop can be left in after violating timing constraints decay quickly to a legal 0 or 1 state. Unfortunately it is possible for the circuit to be left in an illegal *metastable* state where it may remain for an arbitrary amount of time before decaying to a legal state.

After the clock rises, the latch of Figure 24.1(a) becomes a regenerative feedback loop (Figure 24.2(a)). The input transmission gate of the latch is off and the feedback tri-state inverter is enabled so that the equivalent circuit is that of two back-to-back inverters.

The DC transfer curve of these back-to-back inverters is shown in Figure 24.2(b). This figure shows the transfer curve of the forward inverter, V_2 as a function of V_1 ($V_2 = f(V_1)$ blue line), and the transfer curve of the feedback tri-state inverter, V_1 as a function of V_2 ($V_1 = f(V_2)$ red line). There are three points on the figure where the two lines cross. These points are *stable* in the sense that, in the absence of perturbations, the voltages V_1 and V_2 need not change. Since $V_1 = f(V_2) = f(f(V_1))$ the circuit can sit at any of these three points indefinitely.

At any point other than these three stable points, the circuit quickly converges to one of the outer two stable points. For example, suppose that V_2 is just slightly below the center point as shown in Figure 24.3. This drives V_1 to a point found by going horizontally from this point to the red line. This in turn drives V_2 to a point found by going vertically to the blue line, and so on. The circuit quickly converges to $V_1 = 1$, $V_2 = 0$.

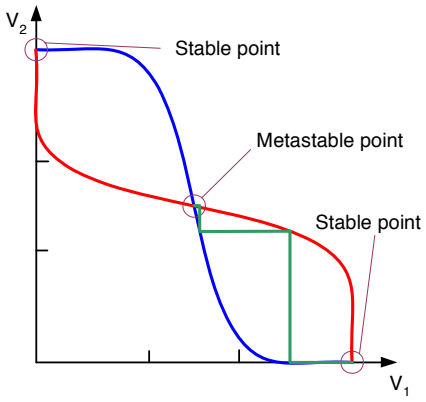


Figure 24.3: The dynamics of the back-to-back inverter circuit can be approximated by repeatedly applying the DC transfer characteristics of the two inverters as shown by the line on this figure.

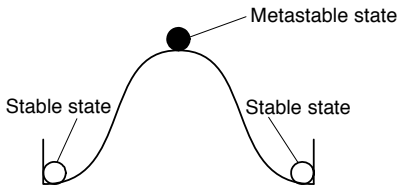


Figure 24.4: A ball at the top of a hill is in a *metastable* state. At rest, there is no force acting to push the ball left or right. However, a slight purturbation to the left or right will cause the ball to leave this state and fall to the left or right stable states at the bottom of the hill.

This iteration of the DC transfer curves is an oversimplification, but it gets the main point across. The circuit is stable at any of the three stable points. However, if we purturb the state slightly from either of the two end points, the state returns to that end point. If we disturb the state slightly from the midpoint, the state will quickly converge to the nearest end-point. A state, like the central stable state, where a small purturbation causes a system to leave that state is called *metastable*.

There are many physical examples of metastable states. Consider a ball at the top of a curved hill as shown in Figure 24.4. This ball is in a stable state. That is, it will stay in this state since, if its exactly centered, there is no force acting to pull it left or right. However, if we give the ball a slight push left or right (a perturbation), it will leave this state and wind up in one of the two stable states at the bottom of the hill. In these states, a small push will result

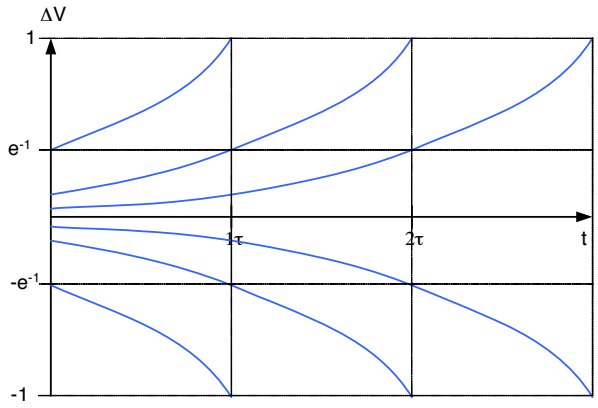


Figure 24.5: Plot of $\Delta V(t)$ for different values of $\Delta V(0)$. The magnitude of ΔV increases by e each time constant.

in the ball returning to the same state.

The ball on the hill is exactly analogous to our flip-flop. The flip flop in the metastable state (center point of Figure 24.2) is exactly like the ball on top of the hill. All it takes is one little push (or not being exactly centered to begin with) and the circuit *rolls* down hill to one of the two stable states.

The dynamics of the back-to-back inverter circuit are in fact governed by the differential equation:

$$\frac{d\Delta V}{dt} = \frac{\Delta V}{\tau_s}, \tag{24.1}$$

where τ_s is the time constant of the back-to-back inverters. In simple terms, the rate of change of the flip-flop state ΔV is directly proportional to its magnitude. The further it is from zero, the faster it moves away — until it is limited by the power supply.

The solution of this differential equation is:

$$\Delta V(t) = \Delta V(0) \exp\left(\frac{t}{\tau_s}\right). \tag{24.2}$$

This solution is plotted in Figure 24.6 for several initial values of ΔV. With the exponential regeneration of the back-to-back inverters, the magnitude of ΔV increases an e-fold each τ_s. Thus if the circuit starts with $\Delta V(0) = e^{-1}$ it takes time τ_s for the circuit to converge to $\Delta V(t) = 1$. Similarly if the circuit starts at $\Delta V(0) = e^{-2}$ it takes $2\tau_s$ to converge, and so on. Generalizing, we see that the amount of time that it takes to converge to $\Delta V = +1$ or -1 is:

$$t_s = -\tau_s \log\left(\Delta V(0)\right). \tag{24.3}$$

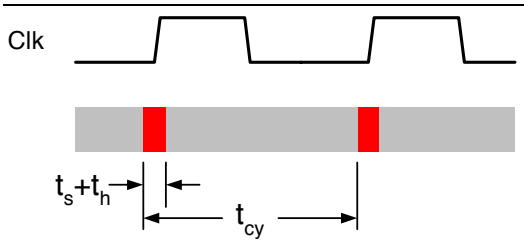


Figure 24.6: Sampling an asynchronous signal with a clock. If a transition of the asynchronous signal happens during the $t_s + t_h$ interval around the clock edge, our flip-flop may enter an illegal state.

24.3 Probability of Entering and Leaving an Illegal State

If we sample an asynchronous signal (one that can change at any instant), with a clock, what is the probability that our flip-flop will enter a metastable state? Suppose our clock period is t_{cy} and our flip-flop has setup time t_s and hold time t_h. Each transition of the asynchronous signal is equally likely to occur at any point during the cycle. Thus, there is a probability of:

$$P_E = \frac{t_s + t_h}{t_{cy}} = f_{cy}(t_s + t_h) \tag{24.4}$$

that a given transition will violate the setup or hold time of the flip-flop and hence cause it to enter an illegal state.

If the asynchronous signal has transitions with frequency $f_a\ s^{-1}$, then P_E of the asynchronous edges fall during the *forbidden* setup and hold region of the cycle. Thus, the frequency of errors (violating setup or hold time) is:

$$f_E = f_a P_E = f_a f_{cy}(t_s + t_h). \tag{24.5}$$

For example, suppose we have a flip-flop with $t_s = t_h = 100$ps and our cycle time is $t_cy = 2$ns. Then the probability of error is

$$P_E = \frac{t_s + t_h}{t_{cy}} = \frac{200ps}{2ns} = 0.1. \tag{24.6}$$

Now consider the case where the asynchronous signal has a transition frequency of 1MHz. Then the frequency of errors is

$$f_E = f_a P_E = 1\text{MHz}(0.1) = 100\text{KHz}. \tag{24.7}$$

We see that for realistic numbers, sampling an asynchronous signal can lead to an unacceptably high error rate.

As we shall see in Chapter 25, a partial solution to the problem of frequently entering illegal states is to *hide* the condition for a period of time, t_w, to allow the illegal state to decay to one of the two stable states. We can calculate the probability of a flip-flop still being in an illegal state after waiting time t_w by considering the probability of certain initial states and the time required for them to decay.

Given that a flip-flop enters an illegal state, it lands at a particular state voltage ΔV with some probability. We can (conservatively) estimate this probability distribution to be uniform. Then, from (24.3) the probability of taking longer than t_w to exit an illegal state is the same as the probability that

$$|\Delta V(0)| < \exp\left(\frac{-t_w}{\tau_s}\right).$$

With $|\Delta V(0)|$ uniformly distributed between zero and one, the probability of taking longer than t_w to reach a stable state is just:

$$P_S = \exp\left(\frac{-t_w}{\tau_s}\right). \tag{24.8}$$

For example, suppose we have a flip-flop with $\tau_s = 100$ps and we wait $t_w = 2ns = 20\tau_s$ for an illegal state to decay. To still be in an illegal state after $20\tau_s$, the flip-flop must have started with $|\Delta V(0)|$ smaller than $\exp(-20)$. With $|\Delta V(0)|$ uniformly distributed over $[0, 1]$, the probability of this is $P_S = \exp(-20)$.

24.4 A Demonstration of Metastability

To many students metastability is just and abstract concept until they see it for themselves. At that point, something clicks and they realize that metastabilty is real and it can happen to their flip-flops. This is best done via a laboratory exercise or a classroom demonstration. Hopefully you will have a chance to experience a live demonstration of metastabilty. If not, then this section will at least show pictures of what it really looks like.

figure of test setup

describe test setup

scope photos of metastable decay

24.5 Bibliographic Notes

24.6 Exercises

24–1 *Probability of Error.*

24–2 *Frequency of Error.*

24–3 *Time to Decay.*

24–4 *Probabiity of Exiting and Illegal State.*

24–5 *False synchronizer.* Metastable state in 1 region. Why doesn't this work.

Chapter 25

Synchronizer Design

In a synchronous system, we can avoid putting our flip-flops in illegal or metastable states by always obeying the setup- and hold-time constraints. When sampling asynchronous signals or crossing between different clock domains, however, we cannot guarantee that these constraints will be met. In these cases, we design a *synchronizer* that through a combination of waiting, for metastable states to decay, and isolation reduces the probablity of synchronization failure.

A brute-force synchronizer consisting of two back-to-back flip-flops is commonly used to synchronize single-bit signals. The first-flip-flop samples the asynchronous signal and the second flip-flop isolates the possibly bad output of the first flip-flop until any illegal states are likely to have decayed. Such a brute-force synchronizer cannot be used on multi-bit signals unless they are encoded with a Gray code. If multiple-bits are in transition when sampled by the synchronizer they are independently resolved, possibly resulting in incorrect codes with some bits sampled before the transition and some after the transition. We can safely synchronize multi-bit signals with a FIFO (first-in first-out) synchronizer. A FIFO serves both to synchronize the signals, and to provide flow control, ensuring that each datum produced by a transmitter in one clock domain is sampled exactly once by a receiver in another clock domain — even when the clocks have different frequencies.

25.1 Where are Synchronizers Used?

Synchronizers are used in two distinct applications. First, when signals are coming from a truly asynchronous source, they must be synchronized before being input to a synchronous digital system. For example, a push-button switch pressed by a human produces an asynchronous signal. This signal can transition at any time, and so must be synchronized before it can be input to a synchronous circuit. Numerous physical detectors also generate truly asynchronous inputs. Photodetectors, temperature sensors, pressure sensors, all produce outputs with transitions that are gated by physical processes, not a clock.

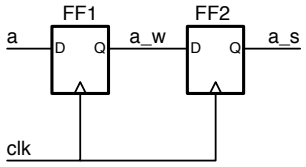


Figure 25.1: A brute-force synchronizer consists of two back-to-back flip-flops. The first flip flow samples asynchronous signal a producing signal a_w. The second flip flop waits one (or more) clock cycles for any metastable states of the first flip-flop to decay before resampling a_w to produce synchronized outptu a_s.

The other use of synchronizers is to move a synchronous signal from one *clock domain* to another. A clock domain is simply a set of signals that are all synchronous with respect to a single clock. For example, in a computer system it is not unusual to have the processor operate from one clock *pclk*, and the memory system operate from a different clock *mclk*. These two clocks may have very different frequencies. For example a *pclk* may be 2GHz while *mclk* is 800MHz. Signals that are synchronous to *pclk* — i.e., in the *pclk* clock domain — cannot be directly used in the memory system. It must first be synchronized with *mclk*. Similarly signals in the memory system, must first be synchronized with *pclk* before they can be used in the processor.

In moving signals between clock domains, there are two distinct synchronization tasks. If we wish to move a sequence of data where each datum in the sequence must be preserved, we use a *sequence synchronizer*. For example, to send eight words in sequence (one word at a time over a data bus) from the processor to the memory system we need a sequence synchronizer. On the other hand, if we wish to monitor the state of a signal we need a *state synchronizer*. A state synchronizer outputs a recent sample of the signal in question synchronized to its output clock domain. To allow the processor to monitor the depth of a queue in the memory system we use a state synchronizer - we don't need every sample of queue depth (one each clock), just one recent sample. On the other hand, we can't use a state synchronizer to pass data between the two subsystems, it may drop some elements and repeat others.

25.2 A Brute-Force Synchronizer

Synchronization of single-bit signals is often done with a *brute-force synchronizer* as shown in Figure 25.1. The flip-flop FF1 samples an asynchronous signal a producing output a_w. Signal a_w is unsafe because of the high frequency with which FF1 will enter an illegal state. To guard the rest of the system from this unsafe signal, we wait one (or more) clock periods for any illegal states of FF1 to decay before resampling it with FF2 to produce output a_s.

How well does the synchronizer of Figure 25.1 work? In other words, what is

the probability of a_s being in an illegal state after a transition on a? This will happen only if (1) FF1 enters an illegal state and (2) this state has not decayed before a_w is resampled by FF2. FF1 enters an illegal state with probability P_E (24.4) and it will remain in this state after a waiting time t_w with probability P_S (24.8). Thus, the probability of FF2 entering an illegal state is:

$$P_{ES} = P_E P_S = \left(\frac{t_s + t_h}{t_{cy}}\right) \exp\left(\frac{-t_w}{\tau_s}\right). \tag{25.1}$$

The waiting time t_w here is not a full clock cycle, but rather a clock cycle less the required overhead:

$$t_w = t_{cy} - t_s - t_{dCQ}. \tag{25.2}$$

For example, if we have $t_s = t_h = t_{dCQ} = \tau_s = 100$ps and $t_{cy} = 2$ns, then the probability of FF2 entering an illegal state is:

$$\begin{aligned} P_{ES} &= \left(\frac{t_s + t_h}{t_{cy}}\right) \exp\left(\frac{-t_w}{\tau_s}\right) \\ &= \left(\frac{100ps + 100ps}{2ns}\right) \exp\left(\frac{-1.8ns}{100ps}\right) \\ &= 0.1 \exp(-18) = 1.5 \times 10^{-9}. \end{aligned}$$

If signal a has a transition frequency of 100MHz, then the frequency of failure is:

$$f_{ES} = f_a P_{ES} = (100MHz)(1.5 \times 10^{-9}) = 0.15Hz. \tag{25.3}$$

If this synchonizer failure probability isn't low enough, we can make it lower by waiting longer. This is best accomplished by adding clock enables to the two flip flops and enabling them once every N clock cycles. This lengthens the wait time to

$$t_w = N t_{cy} - t_s - t_{dCQ}. \tag{25.4}$$

In our example above, waiting two clock cycles reduces our failure probability and frequency to:

$$P_{ES} = 0.1 \exp -38 = 3.1 \times 10^{-17}, \tag{25.5}$$

$$f_{ES} = (100MHz)(3.1 \times 10^{-17}) = 3.1 \times 10^{-9} Hz. \tag{25.6}$$

Using a clock enable to wait longer is more efficient than using multiple flip-flops in series because with the clock enables we only pay the flip-flop overhead $t_s + t_{dCQ}$ once, rather than once per flip-flop. Each clock cycle after the first adds a full t_{cy} to our waiting time. With flip-flops in series, each additional flip-flop adds $t_{cy} - t_s - t_{dCQ}$ to our waiting time.

How low a failure probability is low enough? This depends on your system and what it is used for. Generally we want to make the probability of synchronization failure significantly smaller than some other system failure mode. For

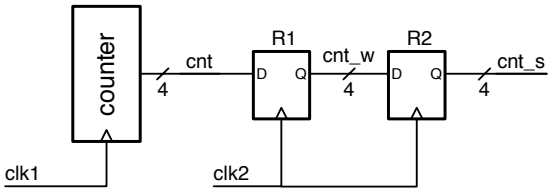


Figure 25.2: Incorrect method of synchronizing a multi-bit signal. Counter clocked by $clk1$ is sampled by synchronizer clocked by $clk2$. On a transition where multiple bits of signal cnt change, the synchronized output cnt_s may see some of the bits change, but not others giving an incorrect result.

example, in a telecommunication system where the bit-error rate of a line is 10^{-20} it would suffice to make a synchronizer with a failure probability P_{ES} of 10^{-30}. For some systems used for life-critical functions, the mean-time to failure (MTTF $= 1/f_{ES}$) must be made long compared to the lifetime of the system times the number of systems produced. So if the system is expected to last 10 years (3.1×10^8s), and we build 10^5 systems, we would like to make our MTTF much larger than 3.1×10^{13} (i.e., f_{ES} should be much less than 3×10^{-14}. Here we might set a goal of $f_{ES} = 10^{-20}$ (less than one failure every 10^{11} years - per system).

25.3 The Problem with Multi-bit Signals

While a brute-force synchronizer does a wonderful job synchronizing a single-bit signal, it **cannot** be used to synchronize a multi-bit signal unless that signal is Gray coded (i.e., unless only one bit of the signal changes at a time). Consider, for example, the situation shown in Figure 25.2. The output cnt of a four-bit counter clocked by one clock, $clk1$, is synchronized with a second clock, $clk2$, operating at a different frequency. Suppose on the count from 7 (0111) to 8 (1000) all of the bits of cnt are changing when $clk2$ rises - violating setup and hold times on register R1. The four flip-flops of R1 all enter illegal states. During the next cycle of $clk2$ these states all decay to 0 or 1 with high probability, so that when $clk2$ rises again, a legal four-bit digital signal is sampled, and output on cnt_s.

The problem is that with high probability the output on cnt_s is wrong. For each of the changing bits of cnt the synchronizer can settle to either a 0 or a 1 state. In this case, when all four bits are changing, the output of the synchronizer could be any number between 0 and 15.

The only time a brute-force synchronizer can be used with multi-bit signals is when these signals are guaranteed to change at most one bit between synchronizer clock transitions. For example, if we wish to synchronize a counter in this manner, we must use a Gray-code counter - that changes exactly one bit

```
module GrayCount4(clk, rst, out) ;
  input clk, rst ;
  output [3:0] out ;
  wire [3:0] out, next ;

  DFF #(4) count(clk, next, out) ;

  assign next[0] = !rst & !(out[1]^out[2]^out[3]) ;
  assign next[1] = !rst & (out[0] ? !(out[2]^out[3]) : out[1]) ;
  assign next[2] = !rst & ((out[1] & !out[0]) ? !out[3] : out[2]) ;
  assign next[3] = !rst & (!(|out[1:0]) ? out[2] : out[3]) ;
endmodule
```

Figure 25.3: Verilog description of a four-bit Gray-code counter.

on each count. A four-bit Gray-code counter uses the sequence 0, 1, 3, 2, 6, 7, 5, 4, 12, 13, 15, 14, 10, 11, 9, 8, ... Only one bit changes between adjacent elements of this sequence. For example, the eigth transition is from 4 (0100) to 12 (1100). Only the MSB changes during this transition. If we replace the counter of Figure 25.2 with a four-bit gray code counter, and the 4 to 12 transition takes place during a rising edge of $clk2$, then only the MSB of R1 will enter an illegal state. The low three bits will remain steady at 100. Either way the MSB settles, 0 or 1, the output is a legitimate value, 4 or 12.

Verilog code for a 4-bit Gray-code counter that generates this sequence is shown in Figure 25.3.

25.4 A FIFO Synchronizer

If we can't use a brute-force synchronizer on arbitrary multi-bit signals and our signal is not amenable to Gray coding, then how can we move it from one clock domain to another? There are several synchronizers that accomplish this task. The key concept behind all of them is removing synchronization from the multi-bit data path. The synchronization is moved to a control path that is either a single bit or a Gray-coded signal.

Perhaps the most common multi-bit synchronizer is the FIFO synchronizer. The datapath of a FIFO synchronizer is shown in Figure 25.4. The FIFO synchronizer works by using a set of registers R0 to RN. Data are stored into the registers under control of the input clock and read from the registers under control of the output clock. A *tail* pointer selects which register is to be written next, and a *head* pointer selects which register is to be read next. Data are added at the tail of the queue, and removed from the head of the queue. The head and tail pointers are Gray-coded counters that are decoded to one-hot to drive the register enables and multiplexer select lines. Using Gray-code encoding for

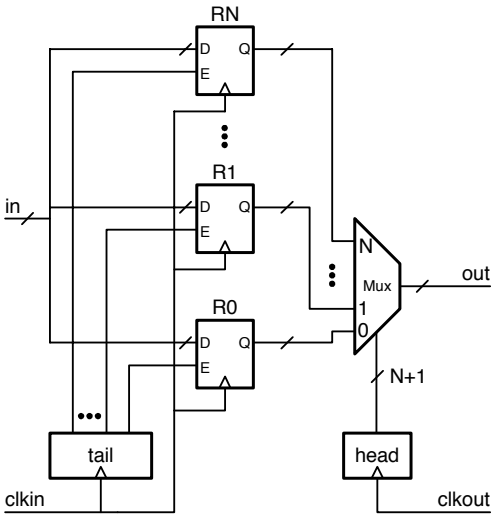


Figure 25.4: Data path of a FIFO synchronizer. Input data are placed in registers $R0, \ldots, RN$ by the input clock *clkin*. Output data are selected from among the registers by an output clock *clkout*. A control path, not shown, ensures that (1) data is placed in a register before it is selected for output and (2) that data is not overwritten before it is read.

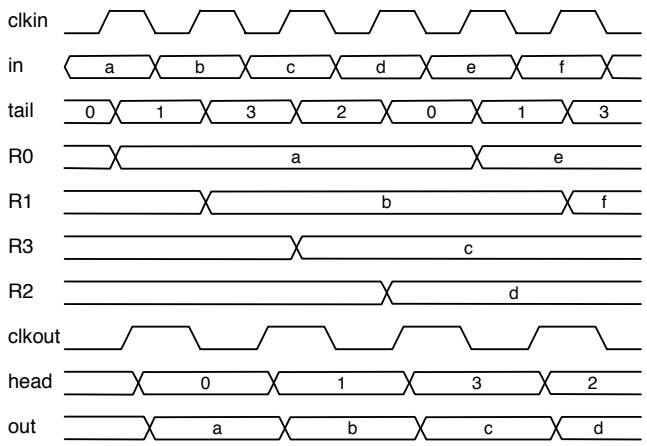


Figure 25.5: Timing diagram showing operation of a FIFO synchronizer with four registers.

the counters enables them to be synchronized using brute-force synchronizers in the control path.

A timing diagram showing operation of a FIFO synchronizer with four registers (R0 to R3) is shown in Figure 25.5. The input clock *clkin* in this example is faster than the output clock *clkout*. On each rising edge of the input clock, a new datum on line *in* is written to one of the registers. The register written is selected by the *tail* pointer which increments with a Gray-code pattern (0,1,3,2,0...). The first datum a is written to register R0, the second datum b is written to R1, c is written to R3, and so on. With four registers, the output of each register is valid for four input clocks.

On the output side, each rising edge of clock *clkout* advances the *head* pointer selecting each register in turn. The first rising edge of *clkout* sets head to 0 selecting the contents of register R0 a to be driven onto the output. The four input-clock valid period of R0 more than overlaps the one output-clock period during which it is selected, so no input-clock driven transitions are visible on the output. The only output transitions are derived from (and hence synchronous with) *clkout*. The second rising edge of *clkout* advances *head* to 1 selecting b from R1 to appear on the output, the third edge selects c from R3, and so on.

By extending the valid period of the input data using multiple registers, the FIFO synchronizer enables this data to be selected on the output without having *clkout* sample any signal with transitions synchronized with *clkin*. Thus, there is no probability of violating setup and hold times in this datapath.

Of course, we haven't eliminated the need to synchronize (and the probability of synchronization failure), we've just moved it to the control path. You will observe that with the input clock running faster than the output clock, our

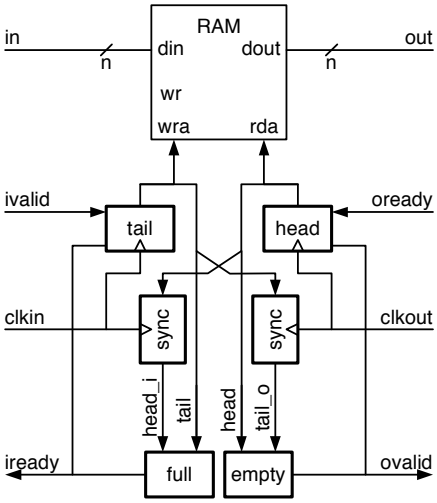


Figure 25.6: FIFO synchronizer showing control path.

FIFO synchronizer will quickly overflow unless we apply some *flow control*. The input needs to be stopped from inserting any more data into the FIFO when all four registers are full. Similarly, if *clkout* were faster than the input clock, we would need to stop the output from removing data from the FIFO when it is empty.

We add flow-control to our FIFO in the control path. A full block diagram of the FIFO including the control path is shown in Figure 25.6 (the registers have been grouped together into a RAM array) and the control path by itself is shown in Figure 25.7. In the control path we add two flow-control signals to each of the two interfaces. On both the input and output interfaces, the *valid* signal is true if the transmitter has valid data on the data line and the *ready* signal is true if the receiver is ready to accept new data. A data transfer takes place only if *valid* and *ready* are both true. On the input side, *ivalid* is an input signal, and *iready* is asserted if the FIFO is not full. On the output side, *oready* is an input signal, and *ovalid* is asserted if the FIFO is not empty.

The *iready* (not full) and *ovalid* (not empty) signals are generated by comparing the head and tail pointers. This comparison is complicated by the fact that head and tail are in different clock domains. Signal *head* is synchronous with *clkout* while *tail* is synchronous with *clkin*. We solve this problem by using a pair of multi-bit brute-force synchronizers to produce a version of *head* in the input clock domain *head_i* and a version of *tail* in the output clock domain *tail_o*. This synchronization is only allowed because head and tail are Gray-coded and thus will have only one bit in transition at any given point in time. Also, note that the synchronization delays the signals, so that *head_i* and *tail_o* are up to

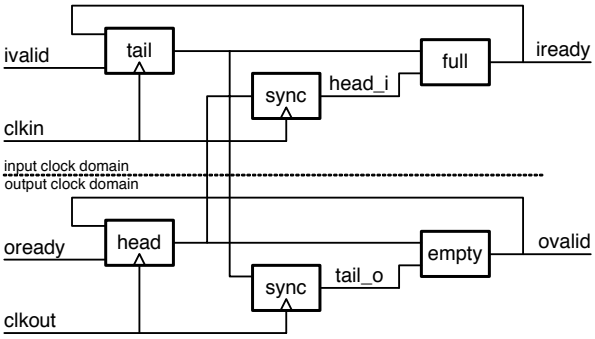


Figure 25.7: Control path of FIFO synchronizer showing clock domains.

two clock cycles behind *head* and *tail*.

Once we have versions of head and tail in the same clock domain, we compare them to determine the full and empty conditions. When the FIFO is empty, head and tail are the same, so we can write:

```
assign empty = (head == tail_o) ;
assign ovalid = !empty ;
```

When the FIFO is full, head and tail are also the same. Thus, if we were to allow all registers to be used, we would need to add additional state to discriminate between these two conditions. Rather than add this complexity, we simply declare the FIFO to be full when it has just one location empty and write:

```
assign full = (head_i == inc_tail) ;
assign iready = !full ;
```

where `inc_tail` is the result of incrementing the tail pointer along the Gray-code sequence. This approach always leaves one register empty. For example, with four registers, only three would be allowed to contain valid data at any time. Despite this disadvantage, this approach is usually preferred because of the high complexity of maintaining and synchronizing the extra state needed to discriminate between full and empty when `head == tail`. We leave to the reader, as Exercise 25–7, the design of a FIFO where all locations can be filled.

The Verilog code for the FIFO synchronizer is shown in Figure 25.9. The width and depth of the FIFO are parameterized — defaulting to 8-bits wide with 8 registers. A synchronous RAM module, not shown, contains the 8 registers. A register clocked by `clkout` holds `head` and one clocked by `clkin` holds `tail`. A pair of brute-force synchronizers creates signals `head_i` and `tail_o`. A pair of 3-bit Gray-code incrementers (verilog code in Figure 25.10) increments head and tail — along the Gray-code sequence — producing `inc_head` and `inc_tail`. Note that if this code were to be used for a FIFO deeper than 8 registers wider Gray-code incrementers would need to be used here — `GrayInc3` is not

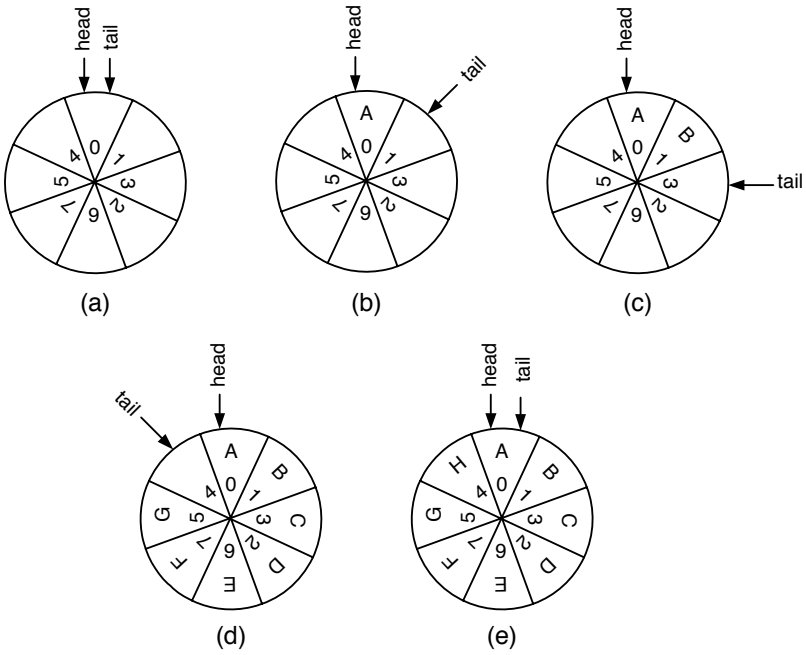


Figure 25.8: FIFO states. (a) FIFO is empty, `head == tail`, (b) After inserting one datum, (c) After inserting two data, (d) Almost full `head == inc(tail)`, (e) Completely full `head == tail`.

```
module AsyncFIFO(clkin, rstin, in, ivalid, iready,
                 clkout, rstout, out, ovalid, oready) ;

  parameter n = 8 ; // width of FIFO
  parameter m = 8 ; // depth of FIFO
  parameter lgm = 3 ; // width of pointer field

  input clkin, clkout, rstin, rstout, ivalid, oready ;
  output iready, ovalid ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  wire [n-1:0] out ;

  // words are inserted at tail and removed at head
  // head_i/tail_o is head/tail synchronized to other clock domain
  // inc_x is head/tail incremented by Gray code
  wire [lgm-1:0] head, tail, next_head, next_tail, head_i, tail_o ;
  wire [lgm-1:0] inc_head, inc_tail ;

  // Dual-Port RAM to hold data
  DP_RAM #(n,m,lgm) mem(.clk(clkin), .in(in), .inaddr(tail[lgm-1:0]),
                        .wr(iready&ivalid) ,
                        .out(out), .outaddr(head[lgm-1:0])) ;

  // head clocked by output, tail by input
  DFF #(lgm) hp(clkout, next_head, head) ;
  DFF #(lgm) tp(clkin, next_tail, tail) ;

  // synchronizers
  BFSync #(lgm) hs(clkin, head, head_i) ; // head in tail domain
  BFSync #(lgm) ts(clkout, tail, tail_o) ; // tail in head domain

  // Gray code incrementers
  GrayInc3 hg(head, inc_head) ;
  GrayInc3 tg(tail, inc_tail) ;

  // iready if not full, oready if not empty
  // input clock for full
  // full when head points one beyond tail
  assign iready = !(head_i == inc_tail) ;
  // output clock for empty
  assign ovalid = !(head == tail_o) ; // output clk

  // tail increments on successful insert
  assign next_tail = rstin ? 0 : (ivalid & iready) ? inc_tail : tail ;

  // head increments on successful remove
  assign next_head = rstout ? 0 : (ovalid & oready) ? inc_head : head ;
endmodule
```

Figure 25.9: Verilog description of a FIFO synchronizer.

```
module GrayInc3(in, out) ;
  input [2:0] in ;
  output [2:0] out ;
  assign out[0] = !(in[1]^in[2]) ;
  assign out[1] = in[0] ? !in[2] : in[1] ;
  assign out[2] = !in[0] ? in[1] : in[2] ;
endmodule
```

Figure 25.10: Verilog description of a 3-bit Gray-code incrementer.

parameterized. Next in the code, flow control signals `iready` and `ovalid` are computed as described above. Finally, the next state for `head` and `tail` are computed. The head and tail are only incremented if both the valid and ready signals for their respective interfaces are true.

Operation of the FIFO synchronizer is shown in the waveforms of Figure 25.11. On the input side, after reset, `iready` becomes true signaling that the FIFO is not full and hence able to accept data. Two cycles later, `ivalid` is asserted and seven data elements are inserted into the FIFO. At this point, the FIFO is full and `iready` goes low. On each data word inserted, `tail` is incremented in a 3-bit Gray-code sequence. Notice that `o_tail` is delayed by two output clocks from `tail`. Later, as words are removed from the FIFO on the output side, `iready` goes high again and additional words are inserted. Signal `ivalid` goes low for three cycles after 9 is inserted. During this period data is not input even though `iready` is true. Just before the end of the simulation, the input (which is running on a faster clock) gets 7 words ahead of the output and `iready` goes low.

On the output side, `ovalid` does not go high, signaling that the FIFO is not empty, until two output cycles after the first datum is inserted on the input side. This is due to the synchronizer delay of signal `tail_o` from which `ovalid` is derived. In the simulation we wait until the FIFO is full and then remove five words, one every other cycle. Note that from removing the first word (01), it takes two input cycles for `iready` to go high. This is due to the synchronizer delay of signal `head_i` from which `iready` is derived. After removing word 05, `iready` remains high for the remainder of the simulation and one word per cycle is removed.

25.5 Bibliographic Notes

There are many more types of synchronizers than we have had time to discuss here. See Dally and Poulton [] for a survey.

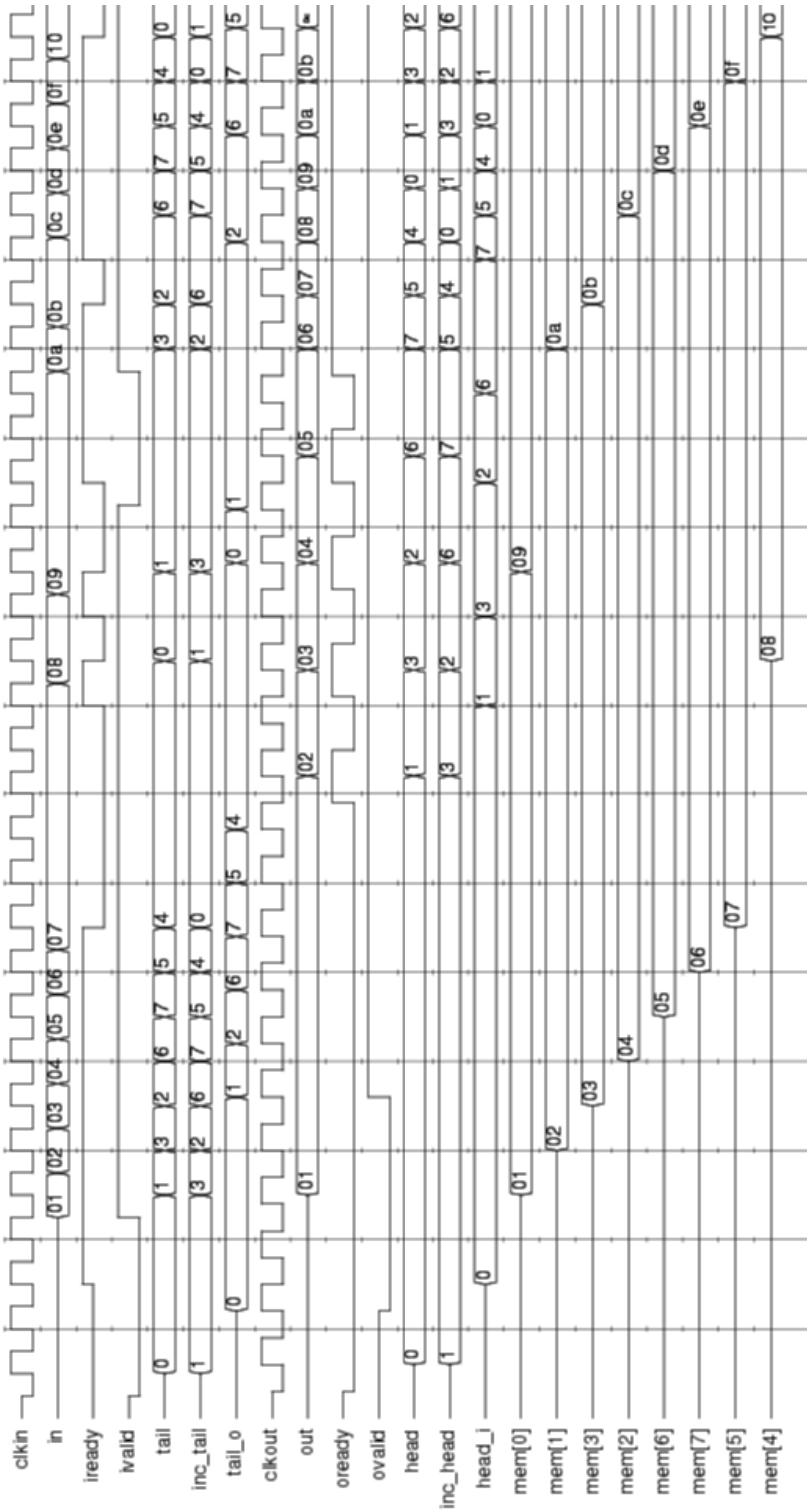


Figure 25.11: Waveforms from simulating the FIFO synchronizer of Figure 25.9

25.6 Exercises

25–1 *Brute force synchronizer.* Calculate failure probability.

25–2 *Brute force synchronizer.* Calculate failure frequency

25–3 *Brute force synchronizer.* Calculate cycles to wait for a given probability.

25–4 *Once-and-only-once synchronizer.* Design a once-and-only-once synchronizer. This circuit accepts and asynchronous input a and a clock clk and outputs a signal that goes high for exactly one clock cycle in response to each rising edge on the input a.

25–5 *Multi-bit synchronization.*

25–6 *FIFO synchronizer.*

25–7 *FIFO synchronizer.* Modify the logic of the FIFO synchronizer to allow the last location to be filled.

25–8 *Clock stopper.*