

Chapter 5 – Registered Logic and State Machines

Digital Design has two main areas. Combinational logic and clocked or registered logic. We have already discussed the former in depth. In the last chapter we discussed flip-flops and how they work. We are now ready to begin designing with flip-flops. This is the most interesting and fun thing of digital design.

For processing data in a computer, it is necessary to construct a device, a circuit, that will be able to hold an item of data. Perhaps the most common data items are characters. The most common code for representing numbers is ASCII. ASCII is a seven-bit code. In the past, an eight bit was used to add *parity*. (See Glossary). How do we hold a character, or any other data item? The answer is a register. A register is a collection of flip-flops that operate as a unit. In a processor, a register may be 8, 16, 32 or even 64 bits long. Each bit is a flip-flop. The most common registers are eight bit. We will discuss registers later in the chapter. We will begin our discussion of registered logic with *state machines*.

A state machine has two or more defined states. It is customary to characterize a state machine with a pictorial called a *state diagram*. A state diagram is simply an array of circles conveniently placed interconnected by lines showing directionality with arrows. Each circle is identified as a state. Given that the state machine is in a particular state, the diagram shows which states are reachable from that state, and also the condition for transitioning to that state. It really is a lot simpler than it sounds.

Provided that the state diagram is an accurate depiction of the state machine, a *state transition table* can be constructed from the state diagram. The state transition table contains the information needed to design a circuit that accomplishes the state machine. Figure 1-1 shows a state diagram for a state machine whose purpose is unknown for the time being. It appears that the state sequence depends upon the present state, and the value of an input, X. We will return to this state machine later. For now, simply recognize what a state diagram looks like.

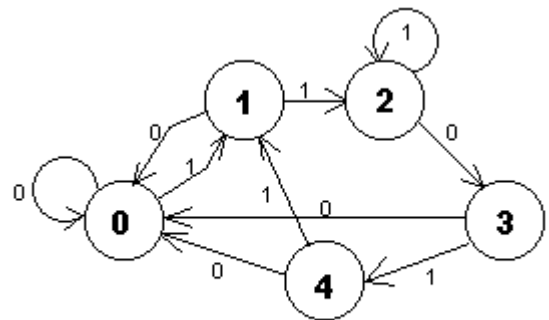


Figure 5-1 State Machine

How do we represent a state? Clearly, we can give the states letters or numbers. Suppose for example, our state machine requires five states, call them 0, 1, 2, 3, 4 and 5. In the last chapter we learned that we can maintain a single bit of memory with a flip-flop. A single flip-flop can have two states, 0 and 1. If we have two flip-flops, we can represent four states: 00, 01, 10 and 11. These are the binary numbers 0 through 3. If we need five states, we'll need to add a flip-flop. Now we can have states 000, 001, 010, 011 and 100 which are binary numbers 0 through 4. We do not need the combinations 101, 110 and 111. The fact that we don't need these states could make the circuitry easier to design. To make life easier and add no unnecessary complication, we can let 000 represent state 0, 001 state 1, etc. This is arbitrary, of course. We could have let any 3-bit combination represent any state but in most cases nothing is gained by this.

In the most general case, the next state, given the present state, might be any other allowable state, so that the state machine does not necessarily progress with the states in numerical order. A special case, however, is a state machine that does in fact advance unconditionally in numerical order. We call such a state machine a counter. This is the simplest kind of state machine and it will be a logical place to start.

Counters

Since our goal is to be able to use a CPLD to do registered logic, we will decline to use J-K flip-flops in our discussion. This is because it is less practical to implement J-K flip-flops in CPLDs. Most PLDs with register capability will at the minimum be capable of implementing D-type flip-flops. Some also can have T flip-flops. We will limit our state machine discussion to D and T flip-flops.

You don't know how much this pains me to leave J-K's out of the discussion, 'cause J-K's are fun! Let me tell you why in one paragraph, then sadly, we won't talk about J-K's anymore in this chapter. In a state transition table (to be discussed soon), if, for example, the next state requires a flip-flop to be set, you might assume that J will be a one and K will be a zero. But suppose the flip-flop that must be one is already a one, then K must be a zero, but it doesn't matter whether J is one or zero. If J is also zero, the flip-flop does not change state and the one that is already there remains. If J is a one, then the flip-flop will be set, and that's OK, too. So in this case, J is a *don't care*. You can prove to yourself that if the flip-flop must be a one and is currently a zero then J must be one, but K is *don't care*. For every table entry, either J or K will be a don't care! Always. So using K-maps, the many don't cares typically make for easy reduction of the equations for for the J and K functions. In CPLD's equation reduction is not so important, but even that doesn't matter because it isn't practical to implement J-K's. Let's get back to counters.

A majority of counters are strictly binary, mostly up-counters, but occasionally down. For our first example, we will implement a modulo-8 counter with a straight binary count sequence. This is a 3-bit counter whose sequence repeats every eight counts: 000, 001, . . . 111, 000, . . . It divides the input clock by eight. We will make a counter first with D-flip-flops, then with T-flip-flops.

A counter is a special case of a state machine. The eight states of our counter can be defined by three flip-flops. Each can represent the binary digit of a 3-bit number that can go from 000 to 111 or 0 to 7. Let the flip-flop representing the LSB (least significant bit) be called A, B the next bit, and C the MSB. We could draw the state diagram, but that would be trivial. It would consist of eight circles with the numbers 0 through 7 enclosed. Circle 0 would show an arrow going to circle 1. Circle 1 would have an arrow pointing to circle 2, etc.

In many state machines there are multiple choices in transitioning to other states. For example, from state 0, one may be able to go to state 1 or state 5 or stay in state zero, depending up conditions which occur. A counter is an example of a state machine with *no variation* in the state progression possible. That is, from state 0 the only choice for the next state is state 1, and for state 1 state 2, etc. Such a state machine is called a sequencer. The conditions for progressing to the next state may be different, but the identity of the next state is never in question in a sequencer. A counter usually has no conditions. It simply advances to the next state in order when the clock occurs. A simple counter could have an enable input that says that the counter will count only if the enable is true. For now, we'll say our counter progresses one state every time a clock occurs. Keep in mind that our purpose here is not to design a super counter, but simply to show the technique for implementing state machines.

D-flip-flop Counter

Let us implement this state machine the traditional way using D-type flip-flops. This is a three-step process.

- 1) Make a state diagram that accurately describes the state transitions.
- 2) Make a state transition table that enumerates all possible state transitions.
- 3) From the table, fill in Karnaugh maps to derive (and reduce) the flip-flop equations.

We'll skip the first step since it is trivial. We already noted that there are no conditions for going to different states. Each state progresses unconditionally to the next state in order. Figure 5-2 shows a state transition table for this problem. There are two parts to the table. The left side is labeled PRESENT for the present state. The right side is labeled NEXT for the next state. The left side usually includes the variables of the state counter and any inputs that affect state transitions. In this simple example, there are no inputs. There are only the counter variables. It will take three flip-flops to specify eight unique states. We will call them A, B and C with A being the LSB. Traditionally when letters are used to designate binary weighting, A is the LSB. This applies to counters, multiplexors, decoders, etc. The right side usually has the state variables and the logic functions necessary to implement the state machine. We see that there appear to be no columns for logic functions. We'll see why momentarily.

The left hand side of the state transition table has at least one state value for each state in the state diagram. If the state diagram shows paths from that state to two other states, there will be two entries for that state. Thus, the total number of entries in the state transition table is the same as the total number of arrowed lines leaving all of the states in the state diagram. If for example, if in state two there are two possible destination states, then state two must be represented on the “present state” side of the table twice. They necessarily differ in the condition(s) that determine the next state. For our counter there are eight states. And each state goes to just one other state. Therefore the total number of transitions and entries in the table is eight.

PRESENT			NEXT		
C	B	A	C	B	A
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

$$D_C = /C^*B^*A + C^*/A + C^*/B$$

$$D_B = /B^*A + B^*/A$$

$$D_A = /A$$

		BA			
		00	01	11	10
C	0	0	0	1	0
	1	1	1	0	1

D_C

		BA			
		00	01	11	10
C	0	0	1	0	1
	1	0	1	0	1

D_B

		BA			
		00	01	11	10
C	0	1	0	0	1
	1	1	0	0	1

D_A

Figure 5-2 State Transition Table and K-Maps for D Flip-flops

The right hand side of the table shows the next state in terms of the flip-flop variables, in this case C, B and A, but also the logic functions required to make the corresponding flip-flops go to that state. For D flip-flops these logic functions might be called D_A , D_B and D_C . And for a T flip-flop T_A , etc. For D flip-flops, however, the D will always be identical to the value of the flip-flop, that is, if flip-flop C becomes a one in the next state, then D_C will have to be a one. To list the D's would be redundant and they are left off. It is different for a T flip-flop. If the present state is a one and the flip-flop is to become a zero, then the T must be a one so that the flip-flop will toggle. Thus, the T's need to be calculated for each transition and may or may not be the same as the next state flip-flop value.

Let us now use the state transition table to fill in the Karnaugh maps. Let us do the A flip-flop first. What we are going to generate is the function for D_A . When the present state is 000, the next state is 001 and the A flip-flop will become a one. So in the K map we put a 1 in the cell CBA = 000. For present state 001, the next state is 010 and the A flip-flop will become zero. So the D input wants to be 0. So for cell 001 we enter a zero. We see that for the even states 000, 010, 100 and 110 the A flip-flop will become a one. Figure 5-2 shows the completed K map. The two columns of one are adjacent and form a group of four whose identity is A -bar. But we could have figured that out without the K map. In other words, the A flip-flop becomes a one when it presently a zero and becomes a zero when it is presently a one. So A always becomes A -bar!

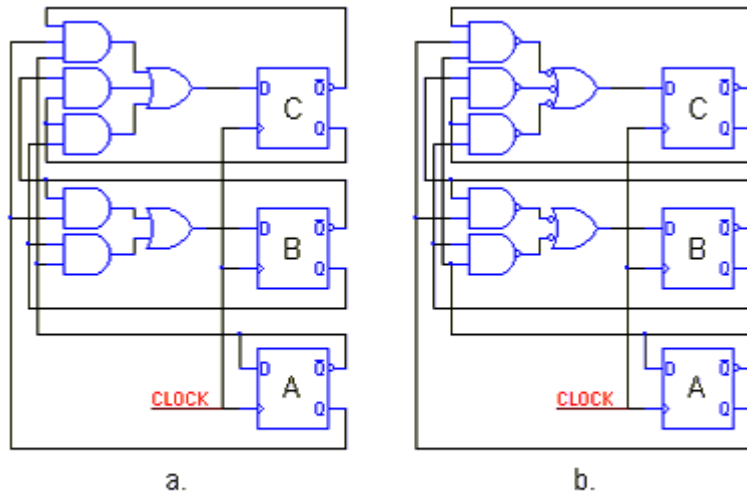


Figure 5-3 Modulo 8 Counter with D Flip-flops

Using the same process of filling in the K map from the transition table we get the maps (and equations) for D_B and D_C . Figure 5-3a. shows the circuit to implement the counter using AND and OR gates to get the usual sum of products solution. Note, however, that an equivalent solution can be obtained by using duality and employing NANDs for the AND function and NANDs for the OR function. This equivalent solution is found in Figure 5-3b. Using conventional family logic, the entire counter will take a pair of dual D flip-flops, a quad 2-input NAND gate, and a triple 3-input NAND gate, or four chips total. As simple as our circuit is, it still takes four chips.

The amazing thing is that we have designed the circuit (and no one else could do it any better!) Consider that you need a modulo 8 counter whose sequence is not linear, for example, 000, 001, 011, 010, 110, 111, 101, 100, 000. If you look carefully you'll notice that the code for this counter is special. In going from one state to the next one and only one bit changes state. In the linear counter, one, two or even three bits may change at one time. In the general case, as states transition, since the flip-flops do not change state instantaneously and in perfect sync, there may be a very brief instant when the code in transition is neither the first code nor the destination code. For example, in changing from 111 to 000, if the middle bit gets there first, the code is momentarily 101. Outputs which decode the state may have narrow glitches in them. The second sequence, called a *Gray* code, does not have this problem. There are no intermediate codes. If the counter changes from 001 to 011 only the initial and final decode states are possible. Gray codes are particularly popular for encoders, for example a binary encoder that outputs changing codes as a shaft turns.

Since the algorithm that determines the flip-flop equations does not depend in any way upon the sequence order or the ordering of the present/next entries in the table, it is just as easy to create the equations for a Gray code counter as it is a linear counter. The count sequence can be perfectly arbitrary to serve any special need. That is, there is nothing special about a linear counter as far as designing it goes. A counter with an arbitrary sequence is just as easily to implement.

T-Flip-flop Counters

We are now going to recode our original 3-bit binary linear counter, but this time using T-type flip-flops. Figure 5-4 shows the state transition table and K-maps for a 3-bit counter. Unlike the D flip-flop where the D logic value is identical to the next state value of the flip-flop, the T logic is different from the next flip-flop value, and needs to be tabulated.

Let's review a T flip-flop again. When the T input is zero, the state of the flip-flop does not change when the next clock occurs. If the T input is one, the state of the flip-flop changes or 'toggles' when the next clock occurs. Let's now go through a few rows of the state transition table and see how the T values are determined. Look at just the A flip-flop first. In row one, the A will go from a 0 in the present state to a 1 in the next state. Therefore it must be toggled and $T = 1$. In the second row, A will go from a 1 to 0, and again must be toggled. In fact, since the A flip-flop is continually alternating or toggling, its T input will simply be one! Let's look at B next. In the first row, B is zero and remains zero. It does not change state, and T is zero. In the second row, B goes from 0 to 1. It must toggle and $T = 1$. In the third row, B is 1 and remains 1 so it must not toggle and $T = 0$. It can be seen that the T_B column alternates between toggling and not toggling. Finally, look at C. C is zero and remains zero for the first three rows, so $T = 0$ for the first three rows. For the fourth row, C goes from 0 to 1 and $T = 1$. C now remains one for the next three rows and $T = 0$. In the last row, C goes from 1 to 0 and must toggle or $T = 1$.

There was no need to map T_A . It is one for all possible states. Mapping T_B , we see that T is one for the group of four that is A. In the map for T_C , the two entries that are one are a group of two that is B^*A . Figure 5-5 shows the circuit of our counter. Aside from the flip-flops, it takes a single two-input AND gate.

While we used a state transition table and K-maps to determine the equations for the T inputs, we could have done it by an analysis of the count sequence. In a linear counter, the LSB always toggles. To make a T flip-flop toggle

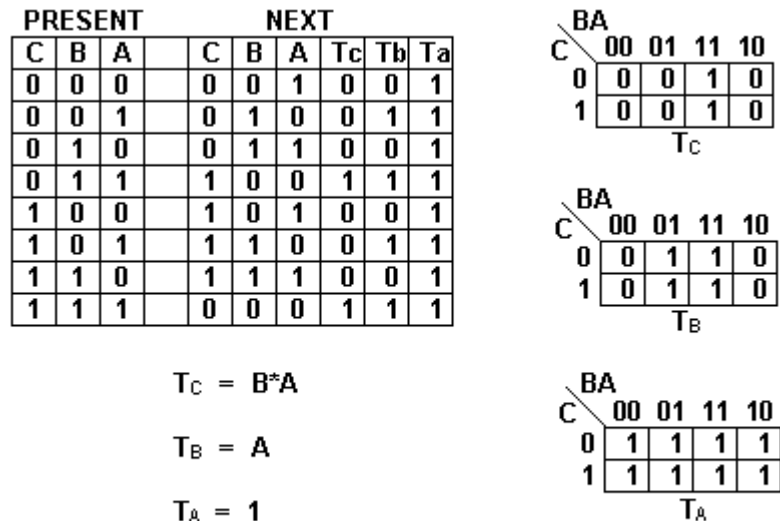


Figure 5-4 State Transition Table and K-Maps for T Flip-flops

always, just connect its T input to one. Consider the sequence 00, 01, 10, 11, 00 . . . The second or B flip-flop whenever the A flip-flop is one. So T_A is A. Finally, consider the sequence 000, 001, 010, 011, 100, 101, 110, 111, 000 . . . Note that C toggles when both A and B are one. Do you C the pattern? If we had a fourth flip-flop, D, it would toggle when A and B and C were all one, that is after the state 0111 and also after 1111. The pattern should be clear. For an up counter, the T input is a one (and will toggle), when all bits on the downstream side (of lesser significance) are all ones. This is amazingly simple. If we had an eight-bit counter, the T input for the most significant bit would be a single AND term. Indeed, all T functions for linear counters are a single AND term. This is great for implementing counters in CPLD's, because the number of OR inputs are limited, but the width of the AND or product term can have as many variables as necessary in it.

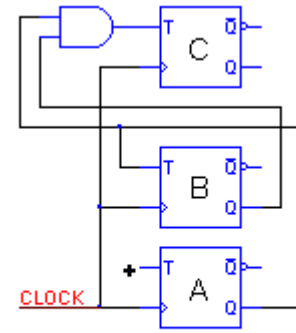


Figure 5-5 Mod 8 Counter with T Flip-flops

Consider a counter with the following sequence: 000, 111, 110, 101, 100, 011, 010, 001, 000 . . . This is a linear down counter. The A bit, as before toggles all the time. When does the B bit toggle? Let's see, it will toggle when the present state is 000, or 110, or 100, or 010 etc. That is, B toggles when A is zero. So the equation for T_B is:

$T_B = \bar{A}$. Similarly, T_C toggles when the present state is 000 and 100. These two states have \bar{A} and \bar{B} both true. Do you see the pattern? For a down counter, the T input for any counter bit is the single AND function of the primed state of all flip-flops of lower order! Wow! What could be simpler?

We are on a roll now, and shouldn't stop. Let's make an up/down counter with an input called U for up. When U is a one we should count up, and when U is a zero we should count down. All we need is the OR of the product for the up counter when U is true and the product for the down toggle when U is false. Thus for T_D :

$$T_D = U * C * B * A + \bar{U} * \bar{C} * \bar{B} * \bar{A}$$

Ta-dah! The high bit of the up/down counter is shown in Figure 5-5. It should be noted that while no logic families ever enumerated T-type flip-flops per say, they all had J-K flip-flops. And a J-K flip-flop with the J and K tied together is a T-type flip-flop since when both J and K are zero the flip-flop remains unchanged, but toggles when both are one.

While T flip-flops are probably the best choice for linear counters, there is one other condition needed to achieve this simplicity. A linear counter made of T flip-flops must count through all 2^N possible states. Any sequence that is truncated, or doesn't start at zero will have extra ones in the K-map that will spoil the perfect binary pattern. For a linear counter that doesn't go full count, it may as efficient to use D flip-flops. With programmable logic, however, it probably doesn't matter too much.

Asynchronous (Ripple) Counters

The counters we have been designing are synchronous. That is, all flip-flops are clocked in unison by a single clock. When an application doesn't require all bits of the counter to change simultaneously, an asynchronous or ripple counter is sufficient. In this type of counter the LSB is clocked from an external source. When that bit falls from a one to a zero, the flip-flop output itself can be connected to the clock of the next flip-flop which is wired to toggle. A D flip-flop toggles when the Q-bar output is connected to the D input. A D flip-flop, however, takes a positive going edge, so that it is the Q-bar output that is connected to the clock from the previous input. The same would be true of a T flip-flop that is clocked on a high-going edge. As mentioned above, a J-K flip-flop with its J and K tied together is a T flip-flop, but J-K flip-flops are traditionally clocked with low-going edges. So a ripple counter can be made with J-K

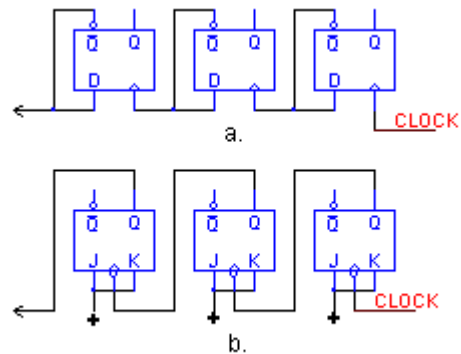


Figure 5-6 Ripple Counters, D and JK Flip-flops

flip-flops by connecting the Q output of the previous flip-flop to the clock and connecting J and K to a one so that it always toggles when clocked. So in a ripple counter, the count ripples down the counter like a wave. Figure 5-6 shows the connections of ripple counters made from D and J-K flip-flops.

The Hold/Enable Function

Returning to synchronous counters, synchronous counters and other state machines are typically clocked with a constant clock or heartbeat. If conditions are not met for a state change, then the counter or state machine must hold the present state when the next clock occurs. Synchronous counters generally have one or more *enables* that must be true for the count to progress. If the enable is false, then the counter should hold its current state. That is, if the flip-flop is currently a one, a one must be applied to the D input, and if a zero, a zero should be applied. Depending on the application the input that determines whether the state machine changes state could be called an Enable to enable a state transition or Hold to hold the current state. Figure 5-7 shows the logic to add an enable function to a D flip-flop. The variable, L, is the logic that will determine the next state. E is an enable that will permit the state to change when the next clock occurs. Note, that some PLDs have an optional clock enable function

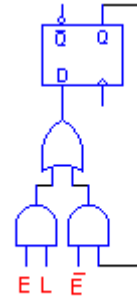


Figure 5-7 D with Enable

Now, consider the hold/enable function for a T flip-flop. The T flip-flop by definition holds whenever T is zero. So the T function itself is inherently a hold function. It either complements or holds. All that is necessary to get a hold with a T flip-flop then is to AND the enable or hold-not variable with the regular function for T.

On the other hand, to specifically set or reset a T flip-flop is not so easy. The only function available is toggle. If you want to set the flip-flop and it is currently zero then T must be true. On the other hand if T is already set then T must be false. Figure 5-8 shows the required circuit to load a T flip-flop with data D. In other words, this is what it takes to make a D flip-flop out of a T flip-flop, whereas Figure 5-7 shows how to make a T flip-flop out of a D flip-flop. The NFL rule applies. There is No Free Lunch. D flip-flops are good for entering data. T flip-flops are good for counting and holding. The application may suggest the better approach. The Xilinx CPLD family implements either D or T flip-flops, so it doesn't matter which turns out to be better.

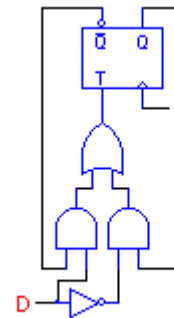


Figure 5-8 T with Data Entry

Non Trivial Counters

The counters we have just designed were admittedly very simple. Practical counters usually have other features. For starters, counters are usually made in 4-bit groups. If a second 4-bit counter (clocked by the same clock) is to follow a first, it will count only once every sixteen clocks. This requires that it use an enable as just described. It also requires an output from the lower counter that tells when the lower counter is at full count. This output is applied to the enable input of the counter above.

Consider a four-bit counter with the following requirements:

1. Two enable inputs
2. A full-count output to enable an upstream counter
3. Four inputs to provide a start count together with a low-true Load input. (The load is synchronous with the clock.
4. A low-true clear input that causes the counter to be reset to 0000 when true, also synchronous with the clock.

Such a counter would require a significant amount of logic and design. It might easily take ten or more chips when implemented with family logic. Circuits of such complexity have long been available in logic families. The chip describe could be the 74X163 binary counter with synchronous load and clear. Figure 5-9 shows the schematic of a

74LS163 counter as taken from the Texas Instruments data sheet. It looks a little complicated, doesn't it. But we know enough about counters to analyze it. Notice that it is made with J-K flip-flops. The counter should count if 1) Load is false (high) 2) CLR is false (high) and the two enables ENP and ENT are true. As a counter, the flip-flops are used as T types. Recall that for a linear up counter with T, the LSB flip-flop, A, toggles every time. The B flip-flop toggles when A is true. C toggles when A AND B are true and D toggles when A AND B AND C are true. The AND gates to the left of center before these ANDs. The ANDs also include the AND of the two EN signals. Thus, based on the AND gates, a one is applied to both the J and K inputs via the OR gate. When at least one of the enables is false, and neither LOAD nor CLR are true, then the OR gate feeding both the J and K has two low inputs. The output of the ORs is false, both J and K are false and the flip-flop remains unchanged.

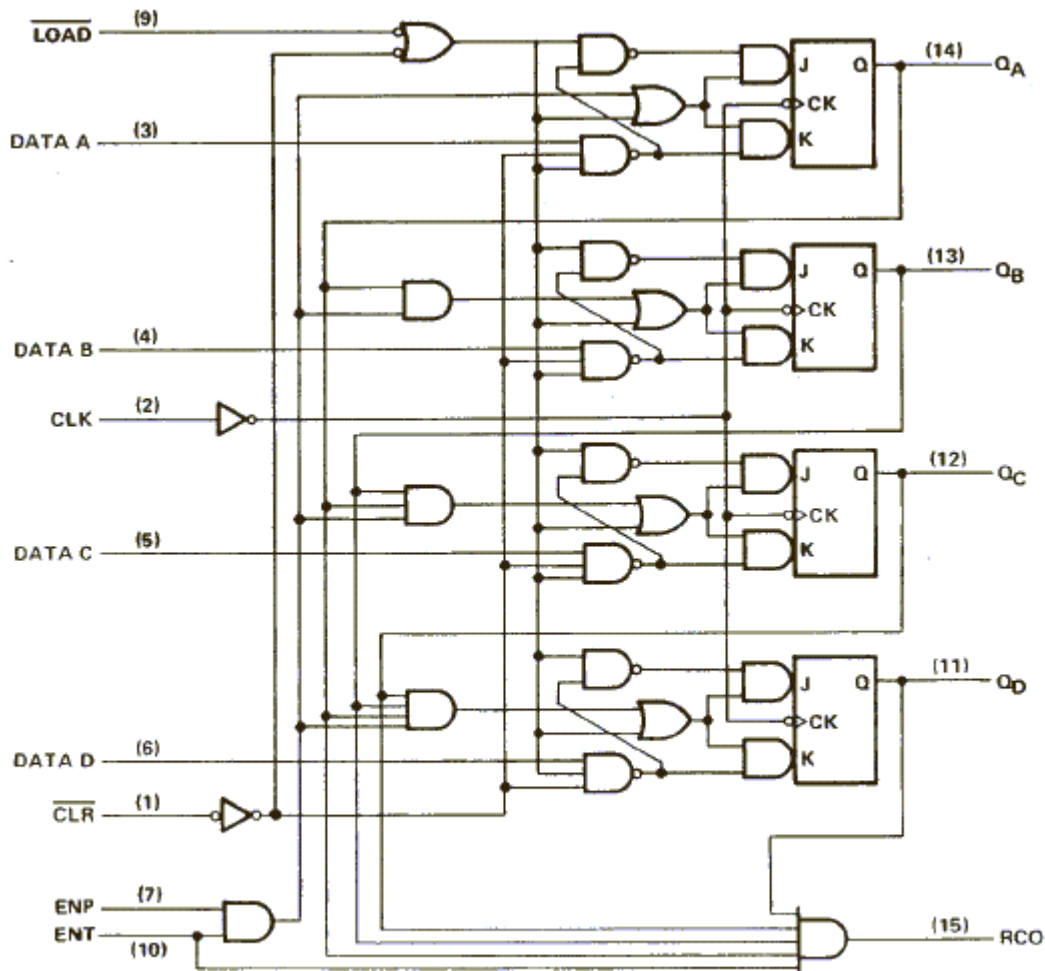


Figure 5-9 74LS163 4-Bit Binary Counter with Synchronous Load and Clear

When CLR is true (low), a zero is applied to the NAND gates feeding the K input of the flip-flop, and the output of the NAND is forced high, so K is high. This high is inverted by the NAND gate feeding the J and puts a zero on the J. So the flip-flop is cleared.

When LOAD is true and CLR is not true, the DATA inputs are applied to the NANDs feeding the K inputs. Again, these NANDs are inverted by the NANDs feeding the J inputs. If a DATA bit is high, then its K is low and its J is high and the flip-flop will be set. If it's low, then the reverse is true and the flip-flop is reset.

Finally, note that there is an output (called RCO) which is true for a full count plus the ENT enable.

An interesting exercise would be to create this counter using T flip-flops and a sum of products feeding the T which accounts for the enables, the load, the clear and the applied data.

Non Sequential State Machines

We started the chapter talking about state machines and used as an example a counter or sequencer, that is, a state machine whose sequence is fixed and unchanging. Now we will consider the more general case of a state machine whose progression of states depends not only on the current state, but on one or more inputs. Let us propose a problem. We are looking at a clocked stream of data, and we want to know when a particular pattern has occurred. Namely, let us look for the four-bit pattern 1101. We really need to draw the state diagram for this problem and we have. Look back at Figure 5-1. There you will see a state diagram for this problem.

The first question we might ask is how many states will this take? Assume that the initial state, zero, defines that none of the pattern has appeared. Given that we are in zero, we should remain in that state until a one appears. That one could be the first one of our pattern and we acknowledge that by going to state one. In state one, if a zero occurs, we've just queered the sequence and have to start over, so we go back to state zero, but if we get a one, we advance to state two. In state two, what will another one do? This will be three ones in a row. You might think this should send us back to zero, but you'd be wrong. We still have two ones in a row and that should keep us in state two. A zero, of course, takes us to state three. In state three, a zero queers the sequence completely again and we go back to zero. Sorry, Charley. A one is a winner and takes us to state four, which indicates success.

Now, here we have to make a decision. Is ANY pattern 1101 valid? If so, the last one of the 1101 could be the first one of a new sequence. We will assume, however, that once a complete sequence occurs, we have to start from the beginning. So if we are in state four, what happens if a one occurs. Be careful. We don't go back to zero. Actually, state four is similar to zero, since from four a new sequence can be launched. A one will take us to state one. A zero, however, will take us back to state zero.

Now that we have a state diagram, the next step in the process is to make a state transition table. If our state diagram is accurate, this shouldn't be difficult. The state transition table shows the transition from each state to every other state. Before we do that, however, we need to determine how many flip-flops are needed to represent all of the states. Our state machine has five states. It takes three bits to represent five different states, so our *state counter* will have three flip-flops. There are situations, however, where fewer aren't necessarily better and using more than the minimal number of flip-flops to represent the states may actually take less hardware overall.

Getting back to our state machine, we'll use three flip-flops to represent the five required states. It is customary to simply identify states by their binary weighting, so that 000 is state zero, 001 is state one, . . . 100 is state four. We'll give the flip-flops the names A, B and C, where A is the LSB and C the MSB. Note, that a three-bit counter has more states than we need. Namely, states 101, 110 and 111 will not be used and won't occur. We can take advantage of this to simplify the design. These states will create *don't cares* in our Karnaugh maps.

To make the state transition table we simply enumerate all of the states in a table and for each state make a new line in the table for each state that that state can transition to. In this example, the input variable X is the only factor that determines the next state (besides the state itself). So each state will have two possible transitions: one for when X = 0 and another for when X = 1. Looking at the state diagram and starting with state zero, we see that when X = 0 we remain in state zero, but when X = 1 we go on to state one. When in state one, X=0 takes us back to zero, while X=1 advances us to two. In two, it is X=0 that advances us, but X=1 does not take us back to zero but just keeps us in two. State two says that the last two inputs (at least) were ones. The last two rows in the table denote present states that will not occur.

We are now ready to fill in the right hand side, the *next* side, of the state transition table. But first we have to decide what kind of flip-flops we will use, D or T? Let's do both and see which is better. As shown in the counter example earlier, the D inputs are the same as the next state for the counter bits, so we need only to add three columns for the T inputs. The K-maps of Figure 5-10 show the results. There is very little difference between using T's or D's. Traditionally, the measure of a good design is a minimum number of literals. For example, if the function is the OR of three AND functions of two inputs each, that is six literals. Today, however, with PLDs, any OR term is a product of any number of literals. It doesn't matter whether there are two terms in the AND term or twenty. One

product term is one OR term, and it is the number of OR inputs or product terms that matters, not the number of a literals in a product term. So in comparing the results for T and D flip-flops we need to look at the number of product terms alone. The T solution has two product terms for the C and B flip-flops and three for the A, or a total of seven product terms. The D solution has one for the C and two each for the B and A or five total. At first glance the T solution looks good, but the D is better. In neither case is the CPLD even mildly stressed since most CPLD's will have at least five product terms (OR inputs) per macrocell.

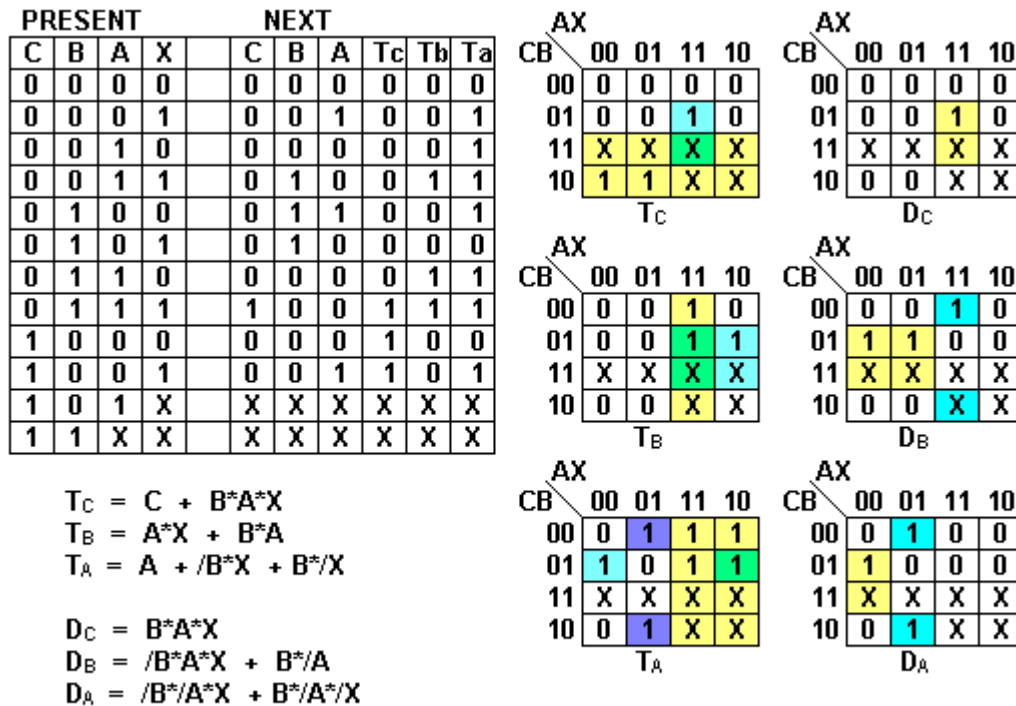


Figure 5-10 State Transition Table and K-Maps for State Machine

There are some interesting state machine problems that are worth discussing, but we'll save them for later.

Other Register Functions

A register is a collection of flip-flops that works together in some fashion to hold a multi-bit binary number. Sometimes they are simply number holders, or if the outputs are available to the world, output devices for a computer system.

Another kind of register is a shift register. A shift register is a collection of flip-flops in which the output of each flip-flop is connected to the D input of its neighbor to the right (or left). When the register is clocked, the data moves one bit position to the right (or left). The data from the rightmost bit is either lost, or it could be clocked into another shift register. (In a computer, the end bit that might be otherwise lost is typically shifted into the carry flag.)

Shifting a binary number one bit causes the number to either be multiplied or divided by two. For example, the number 00001010 (which has a binary weight of ten) when shifted right becomes 00000101 which has a weight of five. Shifting it left becomes 00010100 which has a weight of twenty.. It is not the purpose here to study the arithmetic implications of shifting, but simply to mention that shifting may have arithmetic implications.

Another very important function of shift registers is to perform serial-to-parallel or parallel-to-serial conversions.

These conversion techniques are particularly necessary for interfacing to computers. For example, the popular COM channels on a PC are serial interfaces. The computer takes a byte of data, writes it to a register in an I/O structure and the data is automatically shifted out at some known rate. The advantage of serial data transmission is that it takes far fewer wires. Minimally, a bi-directional serial link can have three wires, one for send, one for receive and one for the signal common or ground. Two transmit a byte of data in a parallel fashion would take eight wires for the data, one or more wires for signal common and a clock wire. If the data is bi-directional then additional wires are needed. The cost of cables with many wires is expensive. The advantage of parallel transmission of data is that it is theoretically faster. In recent times, however, techniques for transmitting data serially have improved so much that it is still preferable to send data serially than in parallel. Very high-speed communication among computers is achieved with networking which uses very high-speed serial data transmission. Now high-speed serial communication has come to peripheral devices using a new protocol called USB for Universal Serial Bus. While printers have traditionally used a parallel interface, newer printers also provide a USB interface which takes a much thinner cable and transmits data faster.

<http://www-s.ti.com/sc/ds/sn74ls164.pdf>