Module 10: Structural Design of Sequential Circuits

Revision: September 10, 2007

Produced in cooperation with DIGILENT®
www.digilentinc.com

Overview

This lab introduces the founding concepts used in the design of sequential circuits. Sequential circuits use memory to store information about past inputs, and they use that information to effect future output changes. Although combinational logic circuits form the backbone of digital circuits, sequential circuits are used in the vast majority of useful devices – there are more than 100 billion in existence.

Background

Sequential circuit characteristics

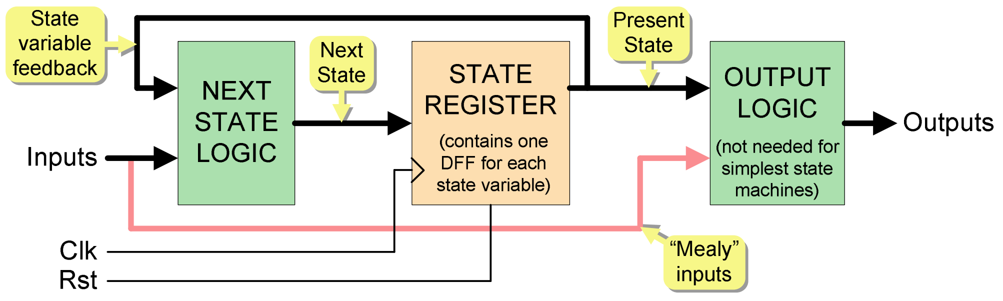
Many problems require the detection or generation of a sequence of events. As examples, an electronic combination-lock door controller must detect when a particular sequence of numbered buttons has been pressed, and an elevator controller must create a sequence of signals to shut the doors, move the cabin, and then reopen doors. In such situations, a circuit can only advance to the next action (or next state) if the current action is known. For example, an elevator should not move until the doors are shut, and pressing a “2” at some point on a combination lock may or may not contribute to the unlock sequence. A circuit that operates according to a specific sequence of events is called a “state machine” or “sequential circuit”. A state machine requires memory to store information about past actions, and it uses that memory to help determine what action to take next. Outputs from sequential circuits are functions of the current inputs and memorized past inputs – this is in contrast to a combinational circuit, where the outputs are strictly a function of the current inputs.

Most memory-containing circuits provide data storage for computing devices. Examples include RAM arrays for computers, registers and register files for microprocessors, cache memories, accumulators, status indicators, etc. These memory circuits may use flip-flops, latches, or RAM cells (depending on the particular application), and they are only used to store data elements in a processor environment. Memory devices used in sequential circuits do not store data, but rather the operating state of the circuit. The state of a sequential circuit is defined by the collective contents of all of its memory devices. The value stored in each memory device in a state machine is referred to as a state variable. Since a state variable can only take one of two values (‘0’ or ‘1’), a circuit with N state variables must be in one of 2^N states, and each state is defined by a unique N-bit binary number. The memory devices in a given state machine are collectively referred to as the state register.

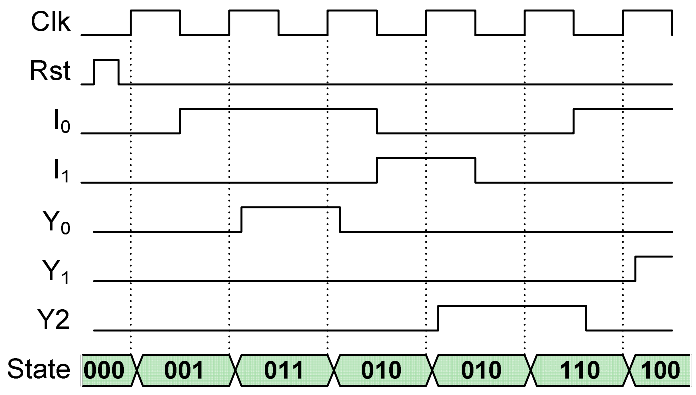
The previous lab presented basic memory devices, including the basic cell, the D-latch, and the DFF. While any of these (or other) memory devices could be used to implement a state register, the sequential machine design process is greatly simplified if memory devices with certain characteristics are used. Those characteristics are: the ability to be driven to a stable operating state (‘0’ or ‘1’); a timing signal that generates the smallest possible sampling window to dictate exactly when new data can be written; a single data input that directly programs the memory device; and a single reset signal that can drive the output to ‘0’ regardless of the data or clock input signals. All of these characteristics are contained in a DFF, and DFF’s are used in practically all sequential circuits. In fact, DFFs can be used to construct any sequential circuit, and their use will always yield the smallest, simplest sequential circuits.

A sequential circuit follows the general model shown below. The state register is controlled directly by an external clock and reset signal. Data inputs to the state register arise from a “next state” logic block that combines circuit inputs with state register outputs - this feedback of the state variables is the reason a sequential circuit can implement a given sequence of events. Without this feedback, future state register changes could not be based on past events, and so ordered sequences could not be implemented. The output from the state register is called the "present state", and the input to the state register is called the "next state". At each edge of the clock, the next state is written into the state register and so becomes the current state.

Like the next-state logic circuit, the output logic circuit contains only combinational devices. In the figure below, the most general state machine model is shown, with circuit inputs fed forward to the output logic block where they can be combined with state variables to determine overall circuit outputs. This most general model is called the "Mealy" model; in the simpler "Moore" model, only the state variables drive the output logic block, so the feed-forward signal would not be shown (i.e., the red line would be absent). In simpler state machines like counters and other basic sequence generators, the output logic block may not be present at all. In such cases, the state register outputs are used as the overall circuit outputs.



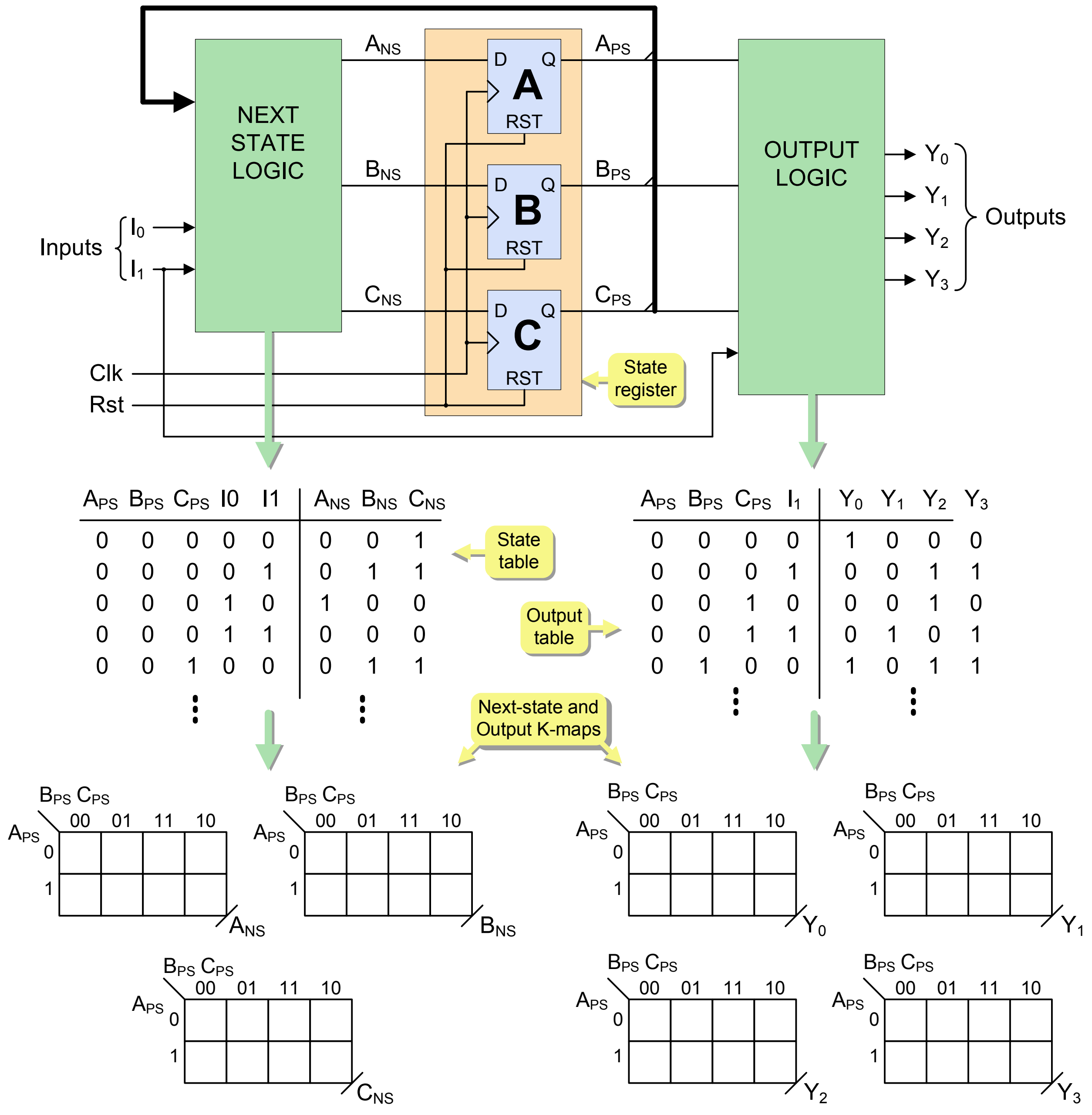
The example timing diagram on the right shows the behavior of a hypothetical state machine (what the state machine does is not important here – just examine the timing diagram). Note that every rising clock edge causes a state transition, where the "next state" is clocked into the state register flip-flops to become the "present state". Each state is uniquely identified by the contents of the state register, called the "state code". This example shows three state variables, so eight distinct states are possible. The state machine progresses from state 0 to states 1, 3, 2, 2, 6, and 4 based on the inputs I_0 and I_1 and the current state code. Note also that the outputs Y_0, Y_1, and Y_2 change just after the clock – this is generally the case, because the state codes change just after the clock edge, and the state codes are inputs to the output logic block.



Designing Sequential Circuits

The most difficult task in designing sequential circuits occurs at the very start of the design, in determining what characteristics of a given problem require sequential operations, and more particularly, what behaviors must be represented by a unique state. A poor choice of states coupled with a poor understanding of the problem can make a design lengthy, difficult and error prone. With better understanding and a better choice of states, the same problem might well be trivial. Whereas it is relatively straight-forward to describe sequential circuit structure and define applicable engineering design methods, it is relatively challenging to find analytical methods capable of matching design problem requirements to eventual machine states. Restated, we can effectively present *how* to design, but we will present *what* to design through examples and guided design problems. And so this initial and most important design task, identifying behaviors in the solution-space to a problem that require unique states, will be presented over time through examples, and you must learn this skill through experience (some general guidelines will also be presented later). In general, the first step in designing a new state machine is to identify all behaviors that might need states, and all branching dependencies between states. Then, as an understanding of the problem and solution evolve, original choices can be rethought, challenged, and improved.

One method of capturing the behavioral requirements of a state machine is through the creation of a state table. A state table is nothing more than a truth table that specifies the requirements for the next-state logic, with inputs coming from the state register and from outside the circuit. The state table lists all required states, and all possible next states that might follow a given present state.



NEXT STATE LOGIC
OUTPUT LOGIC
A_{NS}
B_{NS}
C_{NS}
A_{PS}
B_{PS}
C_{PS}
D Q A RST
D Q B RST
D Q C RST
Inputs
I_0
I_1
Clk
Rst
Y_0
Y_1
Y_2
Y_3
Outputs
State register
State table
Output table
Next-state and Output K-maps
$B_{PS}C_{PS}$ 00 01 11 10 A_{PS} 0 1 A_{NS}
$B_{PS}C_{PS}$ 00 01 11 10 A_{PS} 0 1 B_{NS}
$B_{PS}C_{PS}$ 00 01 11 10 A_{PS} 0 1 C_{NS}
$B_{PS}C_{PS}$ 00 01 11 10 A_{PS} 0 1 Y_0
$B_{PS}C_{PS}$ 00 01 11 10 A_{PS} 0 1 Y_1
$B_{PS}C_{PS}$ 00 01 11 10 A_{PS} 0 1 Y_2
$B_{PS}C_{PS}$ 00 01 11 10 A_{PS} 0 1 Y_3

A_{PS}	B_{PS}	C_{PS}	I0	I1	A_{NS}	B_{NS}	C_{NS}
0	0	0	0	0	0	0	1
0	0	0	0	1	0	1	1
0	0	0	1	0	1	0	0
0	0	0	1	1	0	0	0
0	0	1	0	0	0	1	1
		⋮				⋮	

A_{PS}	B_{PS}	C_{PS}	I_1	Y_0	Y_1	Y_2	Y_3
0	0	0	0	1	0	0	0
0	0	0	1	0	0	1	1
0	0	1	0	0	0	1	0
0	0	1	1	0	1	0	1
0	1	0	0	1	0	1	1
			⋮			⋮	

State-to-state transitions can be directed by input signals, so the table must list any input signals required to cause a given transition. The figure above shows an expanded model of a state machine, and illustrates how the state/truth table can be used to find the next-state logic. In the state table, the first four rows all show “000” for the state variables. This is because there are two inputs, and a next state must be specified for all possible combinations of inputs. From the state table, you can deduce that if the machine is in state “000” and the inputs are both ‘0’, then the next state will be “001”; if the machine is in state “000” and the inputs are ‘0’ and ‘1’, then the next state will be “011”; and so on.

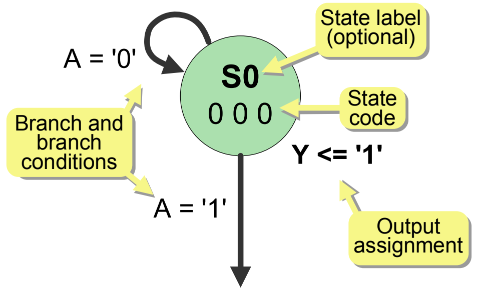
The output truth table shows how the state variables and any Mealy inputs are combined to form outputs. In this example, only one of the two inputs (I_1) is used by the output logic circuit. The state and output tables can be combined into a single truth table (also called a state table) to specify all combinational logic requirements (i.e., both next-state and output requirements) in a single table.

The next-state truth table requires at maximum N input columns for each of N state variables, and M input columns for each of M circuit inputs. It is not required that all possible states nor all possible combinations of inputs be used; hence, the next-state truth table need not have all $2^{(N+M)}$ rows present. Just which rows are required in the truth table depends on which of the 2^N possible states are used in a given sequential circuit, as well as which inputs are used in each state (again, choosing states and branching conditions is the more difficult engineering challenge, and several examples in this and future lab exercises will help illustrate the process). For each row of the truth table, the next-state output values are assigned according to the desired next state. The use of DFFs in the state register is assumed, so a '1' in an output column will cause the corresponding DFF to transition to a '1' on the next clock edge.

Although truth tables (or state tables) can always be used to specify next-state and output logic, they suffer from a significant drawback: it is difficult to visualize the sequential nature of a circuit’s behavior. A more useful method exists for specifying next-state and output logic that has a powerful advantage – it lets us not only specify logic requirements, but also clearly visualize the sequential and/or algorithmic behavior of a circuit.

Designing sequential circuits using state diagrams

A state diagram represents states with circles, and transitions between states by arrows exiting one circle and arriving at another. A binary number called the “state code” can be written in the state-circle to indicate the value stored in the state register when the state machine is in that state. Directed arrows leaving one state and arriving at another show permissible state transitions. Input variable requirements for transitions are shown immediately next to each transition; the indicated transition will only take place if the input conditions shown are present. Transitions (also called branches) occur at every clock edge; thus, at every edge, the present state is exited, and the next-state entered. Often, it is required that for some input conditions, the machine hold in a given state – this holding condition is shown as a directed arrow leaving and re-entering the same state. In the partial state diagram shown, the state register contains three flip-flops: if the state register is storing “000”, then it will remain in that state if A is ‘0’ at the next clock edge; otherwise it will transition out of the state if A is ‘1’. The figure on the right uses VHDL syntax for checking and assigning logic values. Many texts use conventional logic equation symbols instead.

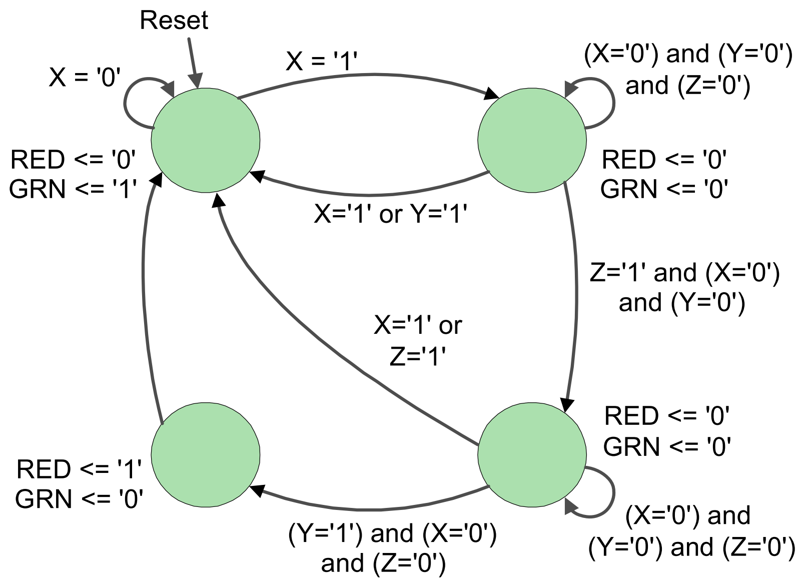


When a state diagram is used as a conceptual tool to help arrive at a given problem solution, it is typically sketched and modified in an iterative fashion. Circles are drawn representing possible states, interconnected according to problem requirements, and redrawn and reconnected as the problem and solution become clearer in the designer's mind. Once a state diagram has been created that captures the design specifications, a fairly automatic procedure can be applied to create a circuit from the diagram.

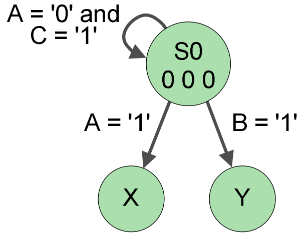
State-to-state transitions occur when the state register is loaded with new next-state values. Since the state register can only be written on a CLK edge, state-to-state transitions can only occur on the CLK edge. Thus, the presence of the CLK signal is implied in a state machine, and the CLK signal is not shown in the state diagram. Likewise, RST or PRE signals are not shown in a state diagram; rather, an arrow is shown pointing to an initial state that the machine should assume whenever a “reset” signal is asserted. A ‘0’ bit in the reset state requires the RST input of the corresponding state register DFF to be connected to the reset signal, and a ‘1’ in the reset state requires the PRE input to be connected to the reset signal. Thus, RST and PRE signals are not shown in the state diagram – their presence is implied when an initial state is identified. Only signals that are needed by the next-state or output logic circuits are shown in the state diagram.

An example of a simple state diagram is shown below. This machine receives input from three buttons labeled X, Y, and Z, and asserts two signals called "RED" and “GRN”: RED if and only if the proper three-button-press sequence X-Z-Y is detected; GRN when a new sequence starts. This “early stage” state diagram does not show state variables or state names. The diagram has evolved by the iterative process mentioned above – states and branching conditions were added and modified as the needs of the problem became clearer, until a complete solution was found.

Note that for each state, the branching conditions take into account all possible input combinations, and no ambiguous branching conditions are present. If some input combinations are not accounted for, or if branching conditions indicate more than one next state, unpredictable operation can occur. The partial state diagrams below illustrate these points – in the diagram on the left, if both A and B are ‘1’, or if C = ‘0’, it is not clear which branch to take. The need to unambiguously show possible next states is important enough that many texts name two rules: the “sum rule” states that all inputs leaving a given state must OR to a logic ‘1’; and the “mutual exclusion rule” states that any combination of inputs can indicate only one next state.

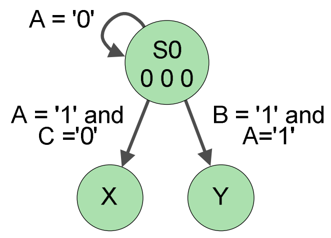


In the figure below, the logic graphs illustrate a simple method for ensuring that both the sum rule and exclusion rule have been obeyed (these graphs resemble, but are not, K-maps). One graph is needed to analyze branching conditions from each state, and the number of input variables determines the graph’s size (input variables are used as the axis variables for the logic graph). Each cell in the graph represents the unique combination of inputs indicated by the axis variables, and cell entries show the



A \ B C	00	10	11	10
0		S0+Y	S0+Y	
1	X	X+Y	X+Y	X

"Sum" errors: cells 0 and 2
"Exclusion" errors: cells 1,3,5,7



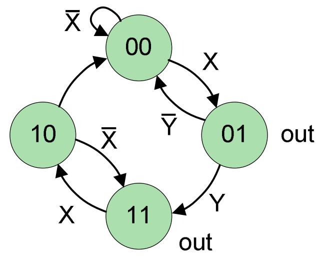
A \ B C	00	10	11	10
0	S0	S0	S0	S0
1	X	X	Y	Y

No errors - each cell has one and only one entry.

next state for the branch conditions indicated by the axis variables. Information can be transferred to the logic graph to document the next-states for all branches from a given state. Each cell should have one and only one entry – an empty cell indicates the sum rule has been violated, and more than one entry indicates the exclusion rule has been violated. The state diagram on the left of the figure above shows both sum rule and exclusion rule violations, and so the state diagram must be modified before further design activities are attempted. In the example shown, one possible solution that removes all "unknowns" and redundancies is shown. Note that removing ambiguities changes the branching conditions – it is up to the designer to choose new branches that are consistent with the problem description. In general, after a state diagram has been sketched, and before any further circuit design activities are undertaken, it is good design practice to ensure that neither the sum rule nor the exclusion rule are violated.

Output signal names are shown near every state during which they must be asserted. If an output must to be asserted in consecutive states, the output should be shown on the state diagram in consecutive states. One method of preparing a state diagram is to show output names only near the states in which they are asserted. A better method is to show each output driven to '1' or '0' in every state – this avoids any confusion.

Once the sequential behavior of a problem has been captured with a state diagram, state codes can be assigned to each state. The state codes show the actual contents of the state register when the state machine is in that state. For a state diagram with N states, at least $\log_2 N$ state variables are required so that each state can be assigned a unique number. In the example above, the state diagram has 4 states, so $\log_2 4=2$ state variables are required. More than the required number of state variables can be used, but in general, the fewest number of state variables needed are used, since adding more state variables creates a larger and more complex circuit. Any state code can be assigned to any state, but in practice certain rules can be used to guide the assignment of state codes.

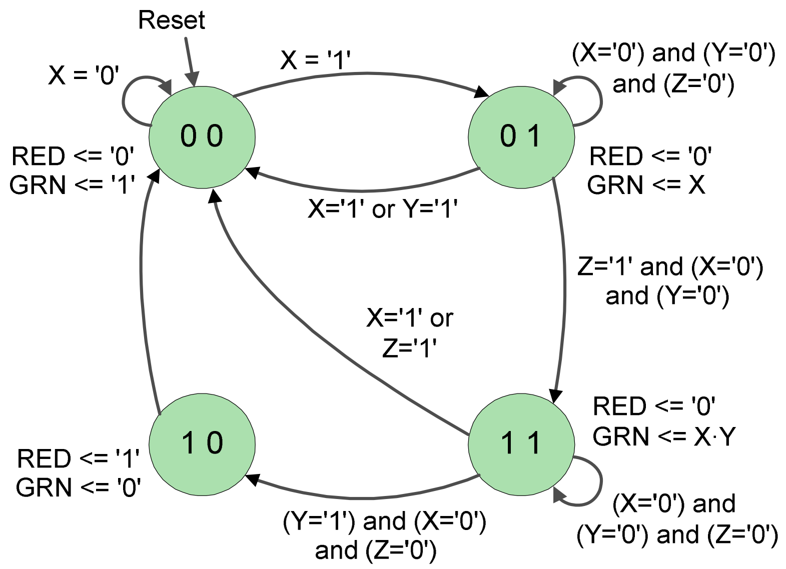


In general, state codes are chosen to minimize the required logic in the next state and/or output logic circuits, or to eliminate timing problems in sequential circuit outputs. One rule of thumb is to minimize the number of flip-flops that change state during any state transition. Ideally, only one flip-flop would change state for any transition in the diagram (a state-to-state transition where only one state variable changes is known as "unit-distance coding"). It is usually not possible to create a situation wherein all transitions are unit-distance coded, but it is generally possible to choose state codes that yield the highest number of unit-distant coded states. A second rule of thumb is to match state register bits to output requirements wherever possible. For example, in a four-state machine with an output that must be asserted in two of the states, it may be possible to assign state codes such that the output is asserted only when one of the flip-flops is a '1', thereby eliminating the output logic altogether. The figure on the right above shows both unit-distant coding, and matching an output to state codes (i.e., the output is ‘1’ whenever flip-flip #2 is a ‘1’, meaning no output logic is required).

Structural design of sequential circuits

A state diagram with state codes and complete branching conditions contains all information required for the design of optimal next-state and output logic circuits. In fact, a state diagram contains exactly the same information as the state table (or next-state truth table), with the added benefit of showing sequential flow. By following a few simple rules, the information in a state diagram can be transferred directly to K-maps so that a minimal next state logic circuit can be found.

The process is illustrated in the three figures below using a state diagram similar to the one presented earlier (but in this state diagram, the GRN output is now a Mealy output that combines the X and Y inputs with state codes – see states “01” and “11”). In the first step, all branch conditions are checked to ensure that neither the sum rule nor the exclusion rule is violated (branch condition checking uses the logic graphs as shown). State codes are assigned so that a minimum number of bits change across the set of all state transitions. In this example, it is not possible to use unit-distant coding for all state transitions, nor is it possible to match outputs to state codes. The state codes shown result in the greatest number of transitions having unit-distant codes. The second step is to transfer information from the state diagram to K-maps so that logic circuits can be defined.



X \ YZ	00	01	11	10
0	00	00	00	00
1	01	01	01	01

Leaving State 00

X \ YZ	00	01	11	10
0	01	11	00	00
1	00	00	00	00

Leaving State 01

X \ YZ	00	01	11	10
0	00	00	00	00
1	00	00	00	00

Leaving State 10

X \ YZ	00	01	11	10
0	11	00	00	10
1	00	00	00	00

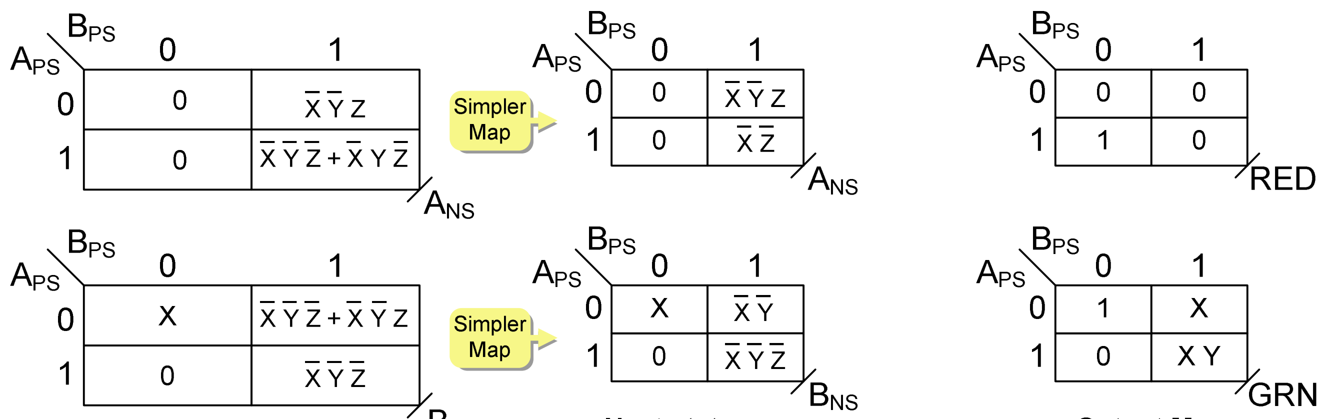
Leaving State 11

Logic graphs to check branch conditions

In this example, two state variables and two outputs require four K-maps, one for each of the next-state circuit, and one for each output. The next-state circuits will drive the D inputs of the state-variable flip-flops, and the output logic circuits will produce outputs based on the state variables and inputs. The state variables are used as the K-map index variables for all four maps. In the next-state maps, branch condition inputs are shown as entered variables. Thus, loops in the next-state maps will be in terms of the state codes (axis variables) and inputs (entered variables). For output maps, a ‘1’ or ‘0’ is placed in a cell to indicate whether an output is asserted in that state; for Mealy outputs, the input variables that drive the output are placed in the maps as entered variables. The “Rules” below describe to process of populating K-maps in detail.

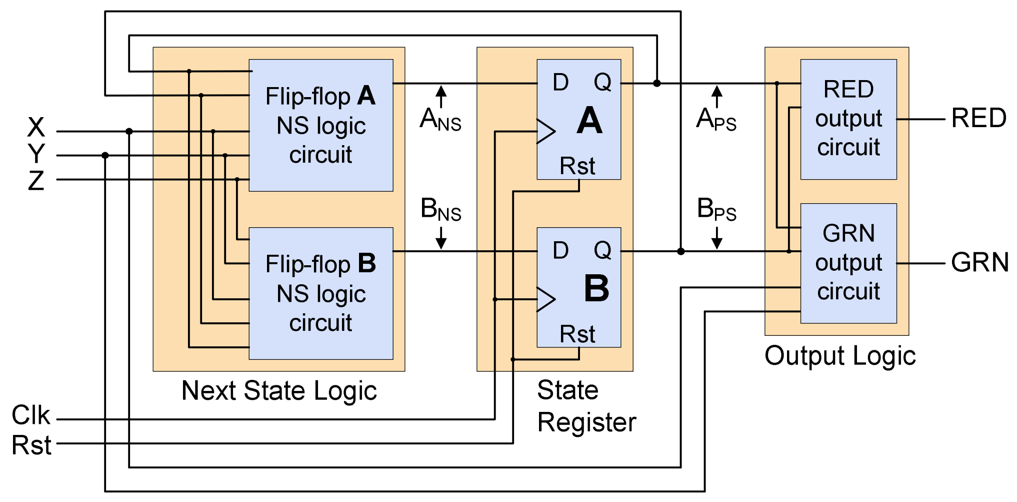
1. Sketch one K-map for each state variable and each output. The state variables are the K-map index variables (and so K-map size is determined by the number of state variables). Since state variables are used on the K-map indexes, each cell K-map cell corresponds to a present state.
2. For next state K-maps, enter branch conditions from each present state into the corresponding K-map cell if and only if the branch leads to a next state where the state DFF being mapped is '1'.
3. For Moore model K-maps, enter a '1' in each K-map cell where the output must be asserted; for Mealy model K-maps, enter a '1' for unconditional outputs, or the variable (or expression) for conditional outputs in each K-map cell where the output must be asserted.

The process is applied to the state diagram above, resulting in the K-maps shown below.


B_{PS}
A_{PS}
0
1
0
$\overline{X}\,\overline{Y}Z$
0
$\overline{X}\,\overline{Y}\,\overline{Z}+\overline{X}Y\overline{Z}$
A_{NS}
Simpler Map
0
$\overline{X}\,\overline{Y}Z$
0
$\overline{X}\,\overline{Z}$
A_{NS}
0
0
1
0
RED
X
$\overline{X}\,\overline{Y}\,\overline{Z}+\overline{X}\,\overline{Y}Z$
0
$\overline{X}\,\overline{Y}\,\overline{Z}$
B_{NS}
Simpler Map
X
$\overline{X}\,\overline{Y}$
0
$\overline{X}\,\overline{Y}\,\overline{Z}$
B_{NS}
1
X
0
X Y
GRN

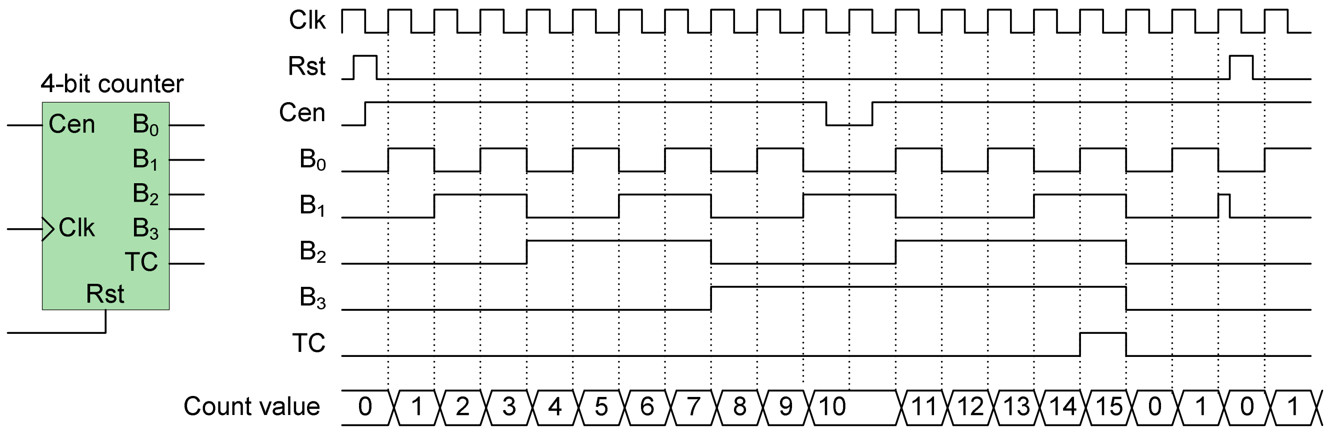
Next-state maps **Output Maps**

The third and final step is to create a circuit from the equations obtained from looping the K-maps. A block diagram of the circuit is shown below – you should recognize the Mealy model schematic. Following the methods described and with sufficient practice, a wide variety of state machines can be designed.



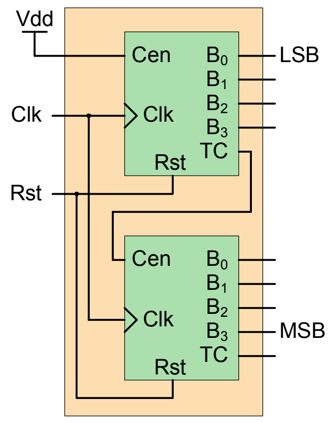
Binary Counters

A binary counter is a simple state machine whose outputs are a repeating sequence of n-bit binary numbers in the range 0 to 2^n -1 (see figure below). At each edge of the clock, the output pattern changes from a binary number X to binary number X + 1; at the end of the count range (at binary number 2^n -1), the counter rolls over, and the next clock will start the count range over at binary number 0. Practical binary counters come in 4-bit, 8-bit, and 16-bit sizes, with count ranges from 0 to 16, 256, and 64K respectively. Counter output bits toggle at rate equal to $1/2^n$ of the input clock input frequency, where n is the bit position (beginning with "1" for the LSB). Counters find many uses in the design of digital systems. As examples, they are often used to generate sequential addresses into a memory array, to create unique states for use in a state machine, or to implement a specific delay or clock-divide ratio.

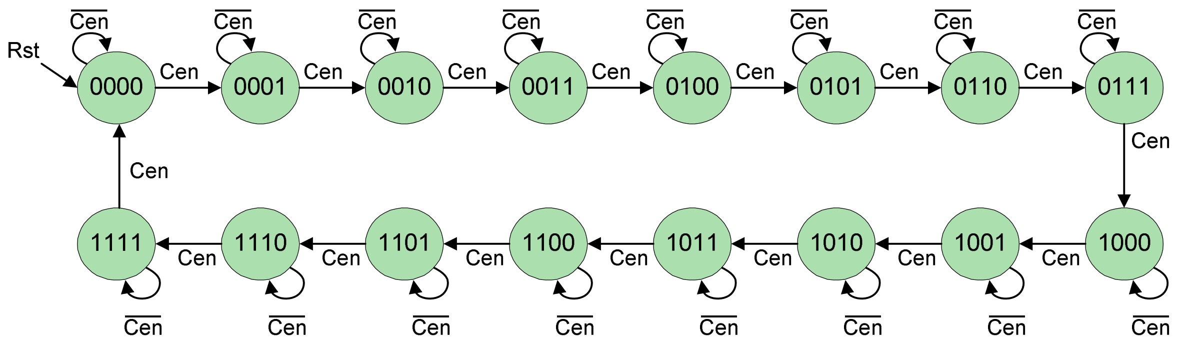


Counters are often designed with a counter enable input (CEN) so that counting can be suspended under certain conditions. When CEN is asserted, the counter will increment with each successive clock edge, and when CEN is not asserted, the counter will simply maintain its current output. Counters are also often designed with a "terminal count" (TC) output that is asserted as the AND of all output bits –that is, TC is asserted only when all counter bits are '1'. Note that when all bits are '1', the counter's next state will be all '0's. Hence the signal name terminal count – when it is asserted, the counter has reached the end of its range. Both CEN and TC are shown in the timing diagram above.

Smaller counters can be chained together to form larger counters by using the TC output and CEN input. When the first, or least significant, or fastest running counter reaches the end of its count range, it will assert TC. If TC is connected to the CEN of the next counter, then the next counter will increment by one each time the first counter reaches the end of its range.



A counter is somewhat unique among state machines in that: the state variables themselves are the circuit outputs; every state code is used; and every next-state state code is simply the present-state state code + 1. A state diagram for a 4-bit binary counter is shown below. The CEN input must be asserted for a state transition to occur. If DFFs with clock enable inputs are used, then the CEN input can connect to all DFFs clock enable inputs. In this case, CEN would not appear in the state diagram since, like the CLK and RST signals, CEN would not be a part of the next state logic (rather, it would connect directly to the flip-flops instead).



Binary counters in VHDL

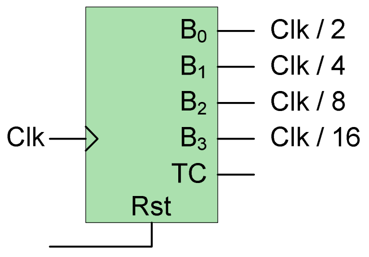
A counter circuit can be implemented using structural or behavioral VHDL. A structural counter design would instantiate the required number of flip-flops as components, and then define next-state logic circuits to drive each flip-flop D input. This design process is rather tedious when compared to a behavioral VHDL design, but in return a much better simulation model could be developed. The structural design of various counters will be covered in depth in a later module.

A behavioral counter can take advantage of the IEEE STD_LOGIC_UNSIGNED library available in any standard VHDL environment. The SLU library allows the use of standard arithmetic operators with STD_LOGIC types (see the fourth line in the example below), making a counter design trivial. Note that the counter output is a vector named B that is defined as an “inout” type so that it can be used on either side of an assignment operator.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity counter is
    Port ( clk : in  STD_LOGIC;
           rst : in  STD_LOGIC;
           B : inout  STD_LOGIC_VECTOR (3 downto 0));
end counter;

architecture Behavioral of counter is

begin

process (clk, rst)
  begin
  if rst = '1' then B <= "0000";
    elsif (clk'event and clk='1') then
      B <= B + 1;
    end if;
  end process;
end Behavioral;
```

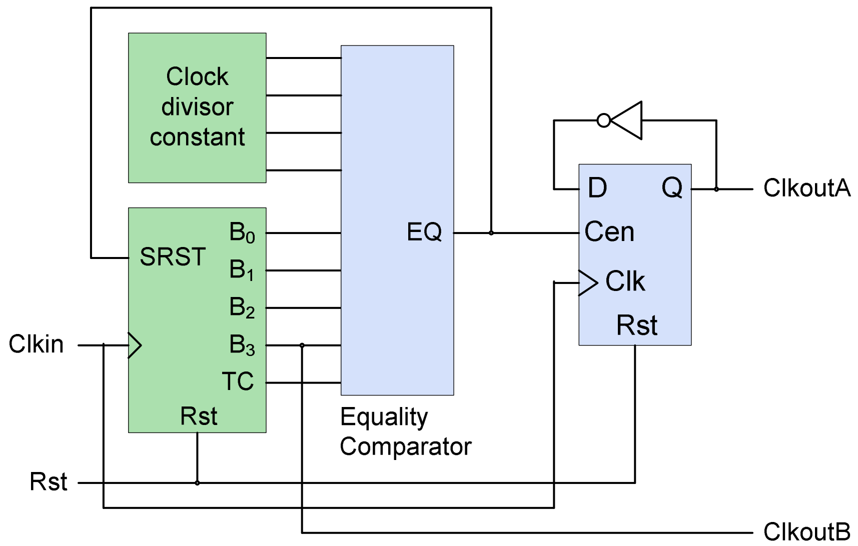
Behavioral VHDL for a 4-bit binary counter

A “clock divider” is one of the more common applications for counters. In this application, a higher frequency clock signal drives a counter’s clock input, and the counter outputs provide lower frequency signals at $1/2^n$ of the input frequency, where n is the counter output bit number (assuming bit #1 is the LSB). Thus, the LSB of the counter provides a frequency of ½ the input frequency, bit #2 provides $1/4^{th}$ the input frequency, bit #3 $1/8^{th}$ the frequency, and so on. In most technologies, the output of one flip-flop (such as a counter output bit) can directly drive the clock inputs of other flip-flops.



Simple clock divider

A simple divider works well for generating frequencies that are power-of-two divisors of the input frequency. To create divider frequencies that are any integer divisor of the input frequency, an equality comparator can be used to compare the count value to a divisor. If a clock with frequency 1/N is required, then a divisor of N/2 can drive one side of the comparator (with the counter driving the other side). The output of the comparator can be used as a synchronous reset to restart the counter from ‘0’ (at twice the desired frequency), and also as a clock-enable for flip-flop that has its output tied to its input through an inverter (CkloutA in the figure). The output of this flip-flop will produce the desired frequency with a 50% duty cycle (duty cycle is the fraction of time a signal spends at ‘1’; a 50% duty cycle means the signal is ‘1’ half the time and ‘0’ half the time) . Note that a simpler circuit can produce a clock frequency of 1/N if a 50% duty cycle is not required (and in most applications, duty cycle is not important). This simpler circuit resets the counter when it reaches N (instead of N/2 as above), and then uses the MSB of the counter as the output clock. This signal will have the desired frequency, but it will not have a 50% duty cycle.



Clock divider for any integer divisor

Behavioral VHDL code for a clock divider that divides a 50MHz clock to a 1Hz clock is shown below. In the example code, note that a constant has been used to define the divider ratio; this constant can be changed to set any desired divider ratio. Note also the MSB of the counter is used as clkout, resulting in a clock signal with the correct frequency that does not have a 50% duty cycle.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity clkdiv is
    Port ( clk : in  STD_LOGIC;
           rst : in  STD_LOGIC;
           clkout : out  STD_LOGIC);
end clkdiv;

architecture Behavioral of clkdiv is

constant cntendval : STD_LOGIC_VECTOR(25 downto 0) := "10111110101111000010000000";
signal cntval : STD_LOGIC_VECTOR (25 downto 0);

begin

process (clk, rst)
  begin
  if rst = '1' then cntval <= "00000000000000000000000000";
    elsif (clk'event and clk='1') then
      if (cntval = cntendval) then cntval <="00000000000000000000000000";
        else cntval <= cntval + 1;
        end if;
    end if;
  end process;

  clkout <= cntval(25);

end Behavioral;
```