

ASYNCHRONOUS STATE MACHINE DESIGN THE DIGILOCASYN

As a design example, we shall redesign the Digilock as an asynchronous machine. This has been chosen for several reasons: (1) you are already familiar with what the machine is supposed to do and (2) the clock for the clocked version would probably cost more than the pal!

It may be said, in general that there is no difference in the design of asynchronous machines and synchronous machines except that you use unlocked storage elements for the state variables. While this is true in general, there are several features which, although unimportant in synchronous machines, are very important in asynchronous machines.

According to Sandige's text, the design of a state machine may begin with one of a number of things: (1) a state diagram (best for synchronous machines); (2) a state-transition table, (3) etc.

The State-transition table is the best way to begin with asynchronous machines where it is usually called a FLOW TABLE. Furthermore, we begin with a PRIMITIVE Flow Table which will eventually get reduced to a reduced flow table.

There are two important rules in making these primitive tables: (1) Only one flip-flop may change at a time - another input may not change until the machine has reached a stable state. (2) every input change must lead to a different state. The reason for the first rule is the same as the reason why the D input to an edge-triggered flip-flop may not change during the clock transition. It may lead to unpredictable results. What happens if two do change at the same time? We'll try to answer that later.

Why the second rule which means we must include many more states than we really need? The answer here is why do we take a reduced Boolean function, expand it to its minterm form, and then reduce it again? By taking it apart, we have the opportunity to put it back together in a more efficient form.

So, let's look at the PFT for this lock.

PS	B0= 0 B1= 0	B0= 1 B1= 0	B0= 1 B1= 1	B0= 0 B1= 1	UNL.
INIT	<u>INIT</u>	B0P1	-	ERR1	0
ERR1	INIT	-	ERR2	<u>ERR1</u>	0
ERR2	-	ERR3	<u>ERR2</u>	ERR1	0
ERR3	INIT	<u>ERR3</u>	ERR2	-	0
B1P1	B0R1	<u>B0P1</u>	ERR2	-	0

B0R1	<u>B0R1</u>	ERR3	-	B1P1	0
B1P1	B1R1	-	ERR2	<u>B1P1</u>	0
B1R1	<u>B1R1</u>	ERR2	-	B1P2	0
B1P2	B1R2	-	ERR2	<u>B1P2</u>	0
B1R2	<u>B1R2</u>	B0P2	-	ERR1	0
B0P2	UL1	<u>B0P2</u>	ERR2	-	0
UL1	<u>UL1</u>	UL2	-	UL3	1
UL2	UL1	<u>UL2</u>	UL4	-	1
UL3	UL1	-	UL4	<u>UL3</u>	1
UL4	-	UL2	<u>UL4</u>	UL3	1

This table indicate that the machine has 15 primitive states. It will be left you to analyze this table other than to note that there must be an error state for each column except the 00 column which is the initial state. There are four unlock states, one in each column since we intend the bolt to remain open until a RESET button is pressed. In other words, the machine will hang up in these four states.

Now, we look for ways of reducing the number of states be MERGING them. Two state may be merged if (a) they have the same outputs and (b) they have the same states in each column (whether or not they are stable and don't cares may be used). Note that if one designs a Mealy machine (not generally recommended), the outputs may be different.

The most common way of merging states is with a merger diagram. Consult Sandige's text for such a diagram. A simple analysis of this machine will reveal that the three error states may all be merged together (we'll call the merged state ERR) and the four UL stakes may be merged (we call this ULK). The table will now be as below.

PS	B0= 0 B1= 0	B0= 1 B1= 0	B0= 1 B1= 1	B0= 0 B1= 1	UNL.
INIT	<u>INIT</u>	B0P1	-	ERR	0
ERR	INIT	<u>ERR</u>	<u>ERR</u>	<u>ERR</u>	0
B1P1	B0R1	<u>B0P1</u>	ERR	-	0
B0R1	<u>B0R1</u>	ERR	-	B1P1	0
B1P1	B1R1	-	ERR	<u>B1P1</u>	0

B1R1	<u>B1R1</u>	ERR	-	B1P2	0
B1P2	B1R2	-	ERR	<u>B1P2</u>	0
B1R2	<u>B1R2</u>	B0P2	-	ERR1	0
B0P2	ULK	<u>B0P2</u>	ERR	-	0
ULK	<u>ULK</u>	<u>ULK</u>	<u>ULK-</u>	<u>ULK</u>	1

We now have ten states (one more than the synchronous version would require). The next thing which must be done is STATE ASSIGNMENT. While just about any state assignment will work efficiently in a synchronous machine, in asynchronous machines it is necessary to assign states so as to avoid race conditions. It is not always possible to eliminate all races so, those that can not be eliminated must be "fixed." See Sandiges text for a discussion of this.

Race conditions exist when two (or more) state variables must change as a result of an input change. Therefore, we attempt to make state assignments so that only one state variable changes per transition. This is not always possible. One possible state assignment is shown below.

PS	B0= 0 B1= 0	B0= 1 B1= 0	B0= 1 B1= 1	B0= 0 B1= 1	UNL.
INIT	<u>0000</u>	0001	-	????	0
ERR	0000	<u>????</u>	<u>????</u>	<u>????</u>	0
B1P1	0011	<u>0001</u>	????	-	0
B0R1	<u>0011</u>	????	-	0010	0
B1P1	0110	-	????	<u>0010</u>	0
B1R1	<u>0110</u>	????	-	0111	0
B1P2	0101	-	????	<u>0111</u>	0
B1R2	<u>0101</u>	0100	-	????	0
B0P2	1100	<u>0100</u>	????	-	0
ULK	<u>1100</u>	<u>1100</u>	<u>1100</u>	<u>1100</u>	1

The ERROR state was not assigned above. The above assignment meets the requirements that only one state variable changes per input change. But, what should ERR be? Let's try 1000 which satisfies the requirement for the transition from the INIT state.

PS	B0= 0 B1= 0	B0= 1 B1= 0	B0= 1 B1= 1	B0= 0 B1= 1	UNL.
INIT	0000	0001	-	1000	0
ERR	0000	1000	1000	1000	0
B1P1	0011	0001	1000	-	0
B0R1	0011	1000	-	0010	0
B1P1	0110	-	1000	0010	0
B1R1	0110	1000	-	0111	0
B1P2	0101	-	1000	0111	0
B1R2	0101	0100	-	1000	0
B0P2	1100	0100	1000	-	0
ULK	1100	1100	1100	1100	1

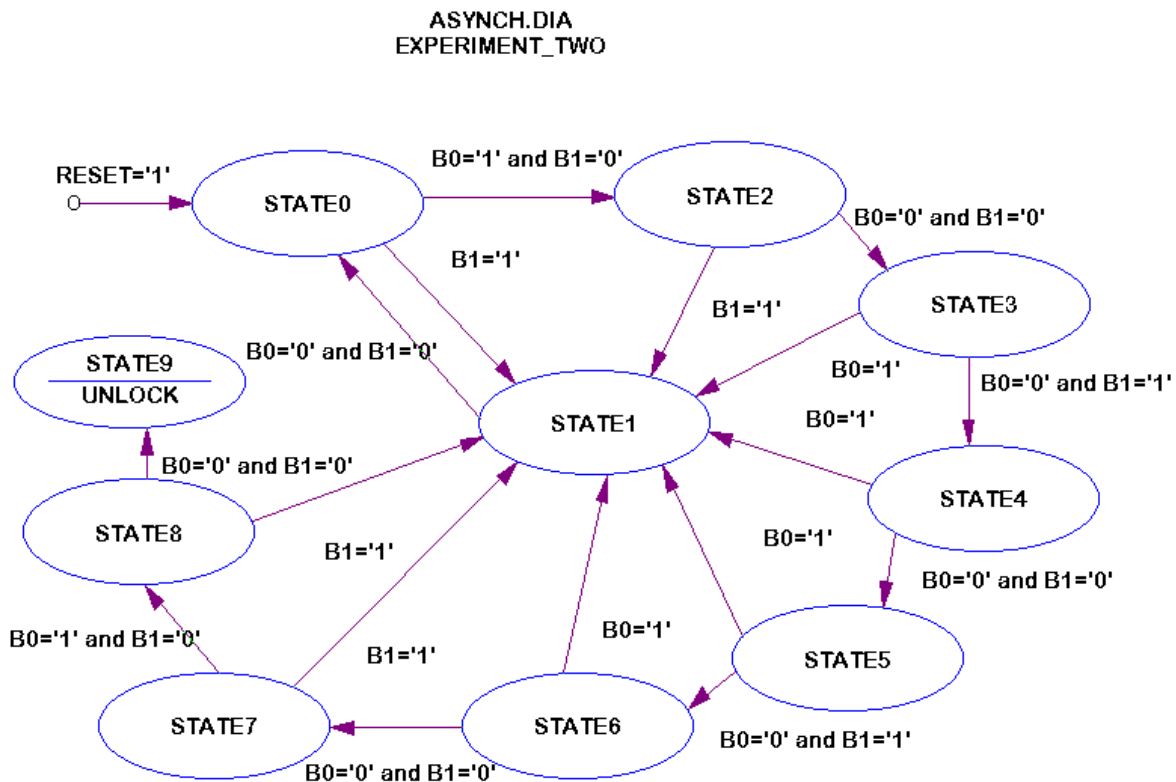
Consider that B1P! with both buttons pressed. Two variables must change to go from 0001 to 1000. Three transitions are possible: 0001 to 1000 (not likely), 0001 to 0000 to 1000, and 0001 to 1001 to 1000. Going to state 0000 is ok since that state is directed to 1000 also. There is no state 1001 (listed). We need to list state 1001 and place a 1000 there to ensure proper transitions. For this machine, the following state assignment should work.

PS	B0= 0 B1= 0	B0= 1 B1= 0	B0= 1 B1= 1	B0= 0 B1= 1	UNL.
INIT	0000	0001	1000	1000	0
ERR	0000	1000	1000	1000	0
B1P1	0011	0001	1000	1000	0
B0R1	0011	1000	1000	0010	0
B1P1	0110	1000	1000	0010	0
B1R1	0110	1000	1000	0111	0
B1P2	0101	1000	1000	0111	0
B1R2	0101	0100	1000	1000	0
B0P2	1100	0100	1000	1000	0
ULK	1100	1100	1100	1100	1

1001	0000	1000	1000	1000	0
1010	0000	1000	1000	1000	0
1011	0000	1000	1000	1000	0
1101	0000	1000	1000	1000	0
1110	0000	1000	1000	1000	0
1111	0000	1000	1000	1000	0

Note that the six unused states have been added, are never stable, and always go to state 1000 (the ERR state) except if the first columns where they go to the INIT state.

We could now attempt to translate this to VHDL, but let's not!



Above is a StateCAD description of this machine (unfortunately it was drawn before the state names were reassigned, so bear with this). It looks the same as the synchronous machine would except for the ERR state (which is STATE1 in the diagram. The unused states do not appear, we will take care of them later.

If we compile this diagram, we will get the VHDL code for a synchronous machine. Not to worry, we can fix that very easily. We also don't know about the state assignment, but that is not big deal either. Below is some of the VHDL generated.

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY ASYNCH IS
    PORT (CLK, B0, B1, RESET: IN std_logic;
          UNLOCK : OUT std_logic);
END;

```

This is the ENTITY part. What is CLK doing there, this is an asynchronous machine? OK, get rid of it with your editor!

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY ASYNCH IS
    PORT (B0, B1, RESET: IN std_logic;
          UNLOCK : OUT std_logic);
END;

```

Now, looking at the first process, we have,

```

PROCESS ( RESET, next_sreg)
BEGIN
    IF ( RESET='1' ) THEN
        sreg <= STATE0;
    ELSIF CLK'EVENT and CLK='1' then sreg <= next_sreg;
    END IF;
END PROCESS;

```

What's CLK doing here? Get rid of it!

```

PROCESS ( RESET, next_sreg)
BEGIN
    IF ( RESET='1' ) THEN
        sreg <= STATE0;
    ELSE sreg <= next_sreg;
    END IF;
END PROCESS;

```

There is no further mention of CLK in the VHDL code, so the machine is now asynchronous. All that remains is to use the state assignment we decided upon instead of what StateCAD used. We modify the following code as shown:

```

CONSTANT STATE0 : std_logic_vector (3 DOWNTO 0) := "0000";
CONSTANT STATE1 : std_logic_vector (3 DOWNTO 0) := "1000";
CONSTANT STATE2 : std_logic_vector (3 DOWNTO 0) := "0001";
CONSTANT STATE3 : std_logic_vector (3 DOWNTO 0) := "0011";
CONSTANT STATE4 : std_logic_vector (3 DOWNTO 0) := "0010";
CONSTANT STATE5 : std_logic_vector (3 DOWNTO 0) := "0110";
CONSTANT STATE6 : std_logic_vector (3 DOWNTO 0) := "0111";
CONSTANT STATE7 : std_logic_vector (3 DOWNTO 0) := "0101";
CONSTANT STATE8 : std_logic_vector (3 DOWNTO 0) := "0100";
CONSTANT STATE9 : std_logic_vector (3 DOWNTO 0) := "1100";

```

This module may now be compiled, after one more important change.

WHEN OTHERS

Must be changed to:

```
WHEN OTHERS => UNLOCK<='0'; next_sreg<=STATE1;
```

To take care of the six unused states.

When compiled, we get the following from the REPORT file.

```
unlock =
  sreg_3 * sreg_2 * /sreg_1 * /sreg_0

sreg_0 =
  /sreg_3 * /sreg_2 * /sreg_1 * b0 * /b1 * /reset
+ /sreg_3 * sreg_2 * sreg_1 * /b0 * b1 * /reset
+ /sreg_3 * sreg_0 * /b0 * /b1 * /reset

sreg_1 =
  /sreg_3 * /sreg_2 * sreg_0 * /b0 * /b1 * /reset
+ /sreg_3 * sreg_1 * /b0 * b1 * /reset
+ /sreg_3 * sreg_1 * /sreg_0 * /b0 * /reset

sreg_2 =
  /sreg_3 * sreg_1 * /sreg_0 * /b0 * /b1 * /reset
+ sreg_3 * sreg_2 * /sreg_1 * /sreg_0 * /reset
+ /sreg_3 * sreg_2 * /sreg_1 * /b1 * /reset
+ /sreg_3 * sreg_2 * sreg_1 * /b0 * /reset

/sreg_3 =
  /sreg_2 * /sreg_1 * /sreg_0 * /b0 * /b1
+ /sreg_3 * sreg_0 * /b0 * /b1
+ /sreg_3 * /sreg_1 * b0 * /b1
+ /sreg_3 * sreg_1 * /b0
+ reset
```

Note the absence of any clocked flip-flops. Not, in particular, the absence of any (overt) flip-flops or latches. In this type of design, the propagation delay in the PAL is used as a storage device (it “stores” the state of the machine for the several nS it takes for the outputs to change).

Does it work? Here’s proof (below). Don’t get too excited yet! A simulator will not show the affect of critical races (if any) since all flip-flops in simulation change state at exactly the same time. The only way to really test it is in the actual hardware.

