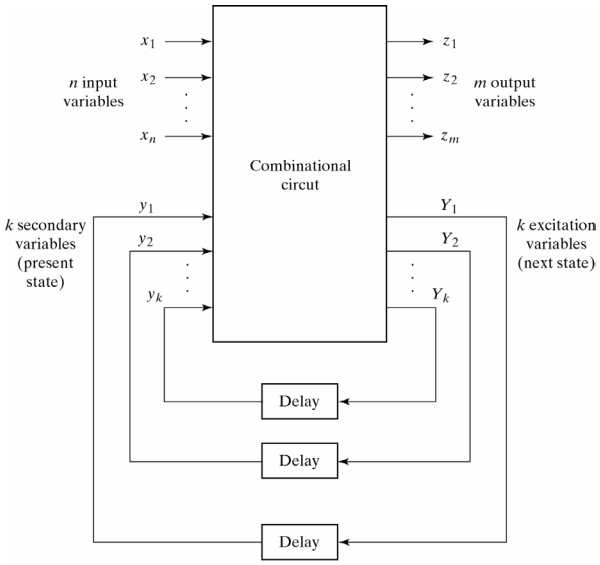
Asynchronous Sequential Circuits

Asynchronous sequential circuits:

- Do not use clock pulses. The change of internal state occurs when there is a change in the input variable.
- Their memory elements are either unclocked flip-flops or time-delay elements.
- They often resemble combinational circuits with feedback.
- Their synthesis is much more difficult than the synthesis of clocked synchronous sequential circuits.
- They are used when speed of operation is important.

The communication of two units, with each unit having its own independent clock, must be done with asynchronous circuits.

The general structure of an asynchronous sequential circuit is as follows:



There are *n* input variables, *m* output variables, and *k* internal states.

The *present* state variables (y_1 to y_k) are called secondary variables. The *next* state variables (Y_1 to Y_k) are called *excitation* variables.

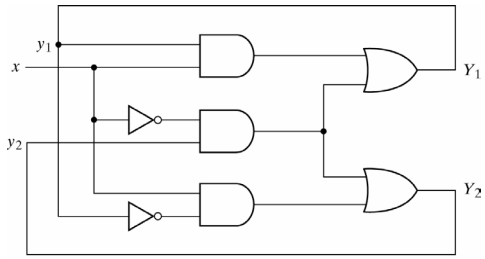
Fundamental-mode operation assumes that the input signals change one at a time and only when the circuit is in a stable condition.

1. Analysis Procedure

The analysis of asynchronous sequential circuits proceeds in much the same way as that of clocked synchronous sequential circuits. From a logic diagram, Boolean expressions are written and then transferred into tabular form.

1.1 Transition Table

An example of an asynchronous sequential circuit is shown below:

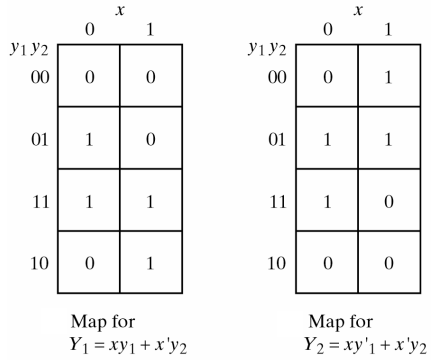


The analysis of the circuit starts by considering the excitation variables (Y_1 and Y_2) as outputs and the secondary variables (y_1 and y_2) as inputs.

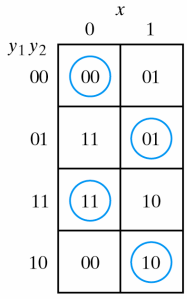
The Boolean expressions are:

$$Y_1 = xy_1 + x'y_2$$
$$Y_2 = xy_1' + x'y_2$$

The next step is to plot the Y_1 and Y_2 functions in a map:



Combining the binary values in corresponding squares the following *transition table* is obtained:



The transition table shows the value of $Y = Y_1Y_2$ inside each square. Those entries where $Y = y$ are circled to indicate a stable condition.

The circuit has four stable *total states* – y_1y_2x = 000, 011, 110, and 101 – and four unstable total states – 001, 010, 111, and 100.

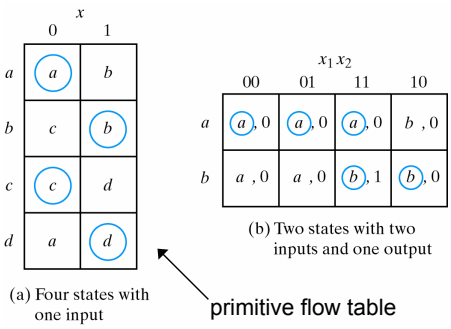
The *state table* of the circuit is shown below:

Present State		Next State $x = 0$		$x = 1$	
0	0	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	1	1	1	0

This table provides the same information as the transition table.

1.2 Flow Table

In a *flow table* the states are named by letter symbols. Examples of flow tables are as follows:



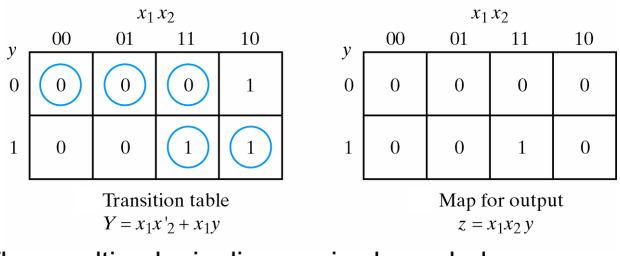
(a) Four states with one input

(b) Two states with two inputs and one output

primitive flow table

In order to obtain the circuit described by a flow table, it is necessary to assign to each state a distinct value.

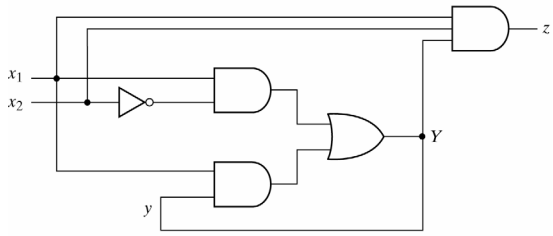
This assignment converts the flow table into a transition table. This is shown below:



Transition table
$Y = x_1x'_2 + x_1y$

Map for output
$z = x_1x_2y$

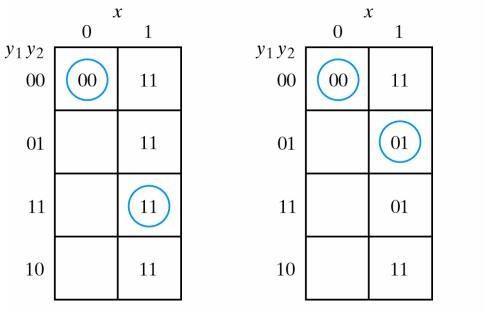
The resulting logic diagram is shown below:



1.3 Race Conditions

A *race* condition exists in an asynchronous circuit when two or more binary state variables change value in response to a change in an input variable. When unequal delays are encountered, a race condition may cause the state variable to change in an unpredictable manner.

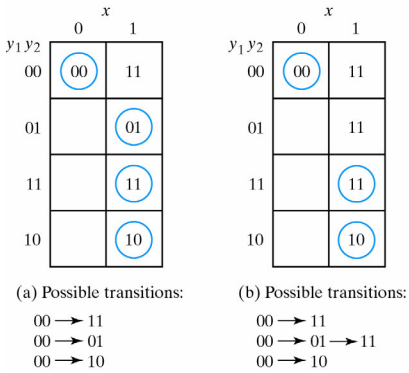
If the final stable state that the circuit reaches does not depend on the order in which the state variables change, the race is called a *noncritical race*. Examples of noncritical races are illustrated in the transition tables below:



(a) Possible transitions:
00 → 11
00 → 01 → 11
00 → 10 → 11

(b) Possible transitions:
00 → 11 → 01
00 → 01
00 → 10 → 11 → 01

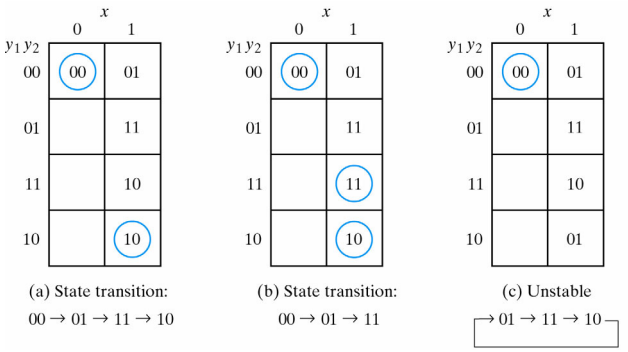
The transition tables below illustrate critical races:



(a) Possible transitions:
00 → 11
00 → 01
00 → 10

(b) Possible transitions:
00 → 11
00 → 01 → 11
00 → 10

Races can be avoided by directing the circuit through a *unique* sequence of intermediate unstable states. When a circuit does that, it is said to have a *cycle*. Examples of cycles are:



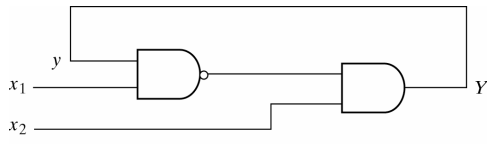
(a) State transition:
00 → 01 → 11 → 10

(b) State transition:
00 → 01 → 11

(c) Unstable

1.4 Stability Considerations

An asynchronous sequential circuit may become unstable and *oscillate* between unstable states because of the presence of feedback. The instability condition can be detected from the transition table. Consider the following circuit:



The excitation function is:

$$Y = (x_1 y)' x_2 = (x_1' + y') x_2 = x_1' x_2 + x_2 y'$$

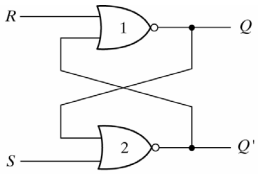
and the transition table for the circuit is:

y \ x_1x_2	00	01	11	10
0	(0)	1	1	(0)
1	0	(1)	0	0

Those values of Y that are equal to y are circled and represent stable states. When the input x_1x_2 is 11, the state variable alternates between 0 and 1 indefinitely.

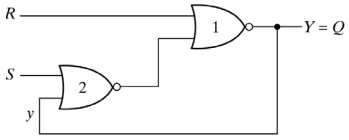
2. Circuits with *SR* Latches

The *SR* latch is used as a time-delay element in asynchronous sequential circuits. The NOR gate *SR* latch and its truth table are:



S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(After $SR = 10$)
0	1	0	1	
0	0	0	1	(After $SR = 01$)
1	1	0	0	

The feedback is more visible when the circuit is redrawn as:



The Boolean function of the output is:

$$Y = [(S + y)' + R]' = (S + y)R' = SR' + R'y$$

and the transition table for the circuit is:

y \ *SR*	00	01	11	10
0	(0)	(0)	(0)	1
1	(1)	0	0	(1)

$Y = SR' + R'y$
$Y = S + R'y$ when $SR = 0$

The behaviour of the *SR* latch can be investigated from the transition table.

The condition to be avoided is that both S and R inputs must not be 1 simultaneously. This condition is avoided when *SR* = 0 (i.e., ANDing of *S* and *R* must always result in 0).

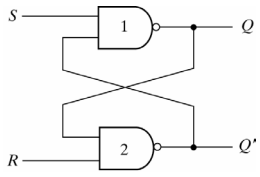
When *SR* = 0 holds at all times, the excitation function derived previously:

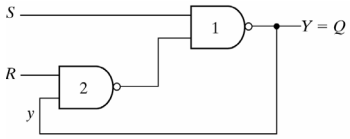
$$Y = SR' + R'y$$

can be expressed as:

$$\boxed{Y = S + R'y}$$

The NAND gate *SR* latch and its truth table are:





S	R	Q	Q'	
1	0	0	1	
1	1	0	1	(After $SR = 10$)
0	1	1	0	
1	1	1	0	(After $SR = 01$)
0	0	1	1	

The transition table for the circuit is:

y \ *SR*	00	01	11	10
0	1	1	(0)	(0)
1	(1)	(1)	(1)	0

$Y = S' + Ry$ when $S'R' = 0$

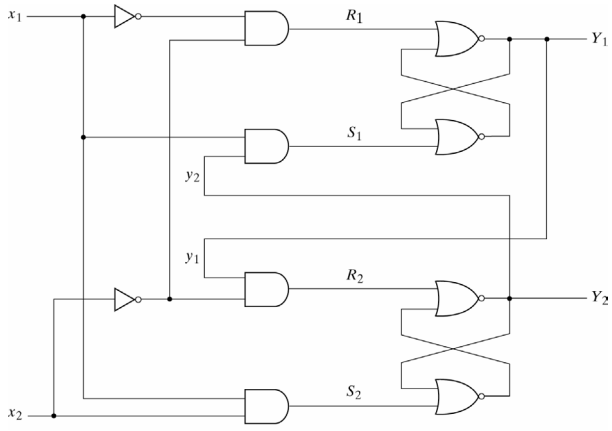
The condition to be avoided here is that both *S* and *R* not be 0 simultaneously which is satisfied when *S*′*R*′ = 0.

The excitation function for the circuit is:

$$Y = [S(Ry)']' = S' + Ry$$

2.1 Analysis Example

Consider the following circuit:



The first step is to obtain the Boolean functions for the *S* and *R* inputs in each latch:

$$S_1 = x_1 y_2 \qquad S_2 = x_1 x_2$$
$$R_1 = x_1' x_2' \qquad R_2 = x_2' y_1$$

The next step is to check if *SR* = 0 is satisfied:

$$S_1 R_1 = x_1 y_2 x_1' x_2' = 0$$
$$S_2 R_2 = x_1 x_2 x_2' y_1 = 0$$

The result is 0 because $x_1 x'_1 = x_2 x'_2 = 0$

The next step is to derive the transition table of the circuit. The excitation functions are derived from the relation $Y = S + R'y$ as:

$$Y_1 = S_1 + R_1' y_1$$
$$= x_1 y_2 + (x_1 + x_2) y_1 = x_1 y_2 + x_1 y_1 + x_2 y_1$$

$$Y_2 = S_2 + R_2' y_2$$
$$= x_1 x_2 + (x_2 + y_1') y_2 = x_1 x_2 + x_2 y_2 + y_1' y_2$$

Next a composite map for $Y = Y_1 Y_2$ is developed:

$y_1 y_2$ \ $x_1 x_2$	00	01	11	10
00	(00)	(00)	01	(00)
01	(01)	(01)	11	11
11	00	(11)	(11)	10
10	00	(10)	11	(10)

Investigation of the transition table reveals that the circuit is stable.

There is a critical race condition when the circuit is initially in total state $y_1 y_2 x_1 x_2 = 1101$ and x_2 changes from 1 to 0. If Y_1 changes to 0 before Y_2, the circuit goes to total state 0100 instead of 0000.

2.2 *SR* Latch Excitation Table

Lists the required inputs *S* and *R* for each of the possible transitions from the secondary variable *y* to the excitation variable *Y*.

y	*Y*	*S*	*R*
0	0	0	*X*
0	1	1	0
1	0	0	1
1	1	*X*	1

Useful for obtaining the Boolean functions for *S* and *R* and the circuit's logic diagram from a given transition table.

2.3 Implementation Example

Consider the following transition table:

y \ $x_1 x_2$	00	01	11	10
0	(0)	(0)	(0)	1
1	0	0	(1)	(1)

$$Y = x_1 x_2' + x_1 y$$

From the information given in the transition table and the *SR* latch excitation table, we can obtain maps for the *S* and *R* inputs of the latch:

y \ $x_1 x_2$	00	01	11	10
0	0	0	0	1
1	0	0	*X*	*X*

Map for $S = x_1 x'_2$

y \ $x_1 x_2$	00	01	11	10
0	*X*	*X*	*X*	0
1	1	1	0	0

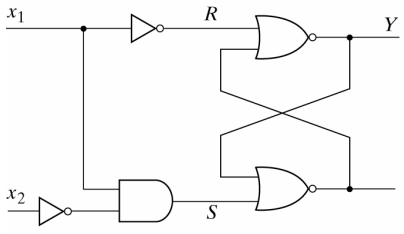
Map for $R = x'_1$

X represents a *don't care* condition.

The maps are then used to derive the simplified Boolean functions:

$$S = x_1 x_2' \qquad R = x_1'$$

The logic diagram consists of an *SR* latch and gates required to implement the *S* and *R* Boolean functions. The circuit when a NOR *SR* latch is used is as shown below:



With a NAND *SR* latch the complemented values for *S* and *R* must be used.

3. Design Procedure

There are a number of steps that must be carried out in order to minimize the circuit complexity and to produce a stable circuit without critical races. Briefly, the design steps are as follows:

1. Obtain a primitive flow table from the given specification.
2. Reduce the flow table by merging rows in the primitive flow table.
3. Assign binary states variables to each row of the reduced flow table to obtain the transition table.
4. Assign output values to the dashes associated with the unstable states to obtain the output maps.
5. Simplify the Boolean functions of the excitation and output variables and draw the logic diagram.

The design process will be demonstrated by going through a specific example:

3.1 Design Example – Specification

Design a gated latch circuit with two inputs, *G* (gate) and *D* (data), and one output *Q*. The gated latch is a memory element that accepts the value of *D* when *G* = 1 and retains this value after *G* goes to 0. Once *G* = 0, a change in *D* does not change the value of the output *Q*.

Step 1: Primitive Flow Table

A *primitive flow table* is a flow table with only one stable total state in each row. The total state consists of the internal state combined with the input.

To derive the primitive flow table, first a table with all possible total states in the system is needed:

	Inputs		Output	
State	*D*	*G*	*Q*	**Comments**
a	0	1	0	$D = Q$ because $G = 1$
b	1	1	1	$D = Q$ because $G = 1$
c	0	0	0	After state *a* or *d*
d	1	0	0	After state *c*
e	1	0	1	After state *b* or *f*
f	0	0	1	After state *e*

Each row in the above table specifies a total state.

The resulting primitive table for the gated latch is shown below:

	DG: 00	01	11	10
a	*c*, –	(*a*), 0	*b*, –	–, –
b	–, –	*a*, –	(*b*), 1	*e*, –
c	(*c*), 0	*a*, –	–, –	*d*, –
d	*c*, –	–, –	*b*, –	(*d*), 0
e	*f*, –	–, –	*b*, –	(*e*), 1
f	(*f*), 1	*a*, –	–, –	*e*, –

First, we fill in one square in each row belonging to the stable state in that row.

Next recalling that both inputs are not allowed to change at the same time, we enter dash marks in each row that differs in two or more variables from the input variables associated with the stable state.

Next we find values for two more squares in each row. The comments listed in the previous table may help in deriving the necessary information.

A dash indicates don't care conditions.

Step 2: Reduction of the Primitive Flow Table

The primitive flow table can be reduced to a smaller number of rows if two or more stable states are placed in the same row of the flow table. The simplified merging rules are as follows:

1. Two or more rows in the primitive flow table can be merged into one if there are non-conflicting states and outputs in each of the columns.
2. Whenever, one state symbol and *don't care* entries are encountered in the same column, the state is listed in the merged row.
3. If the state is circled in one of the rows, it is also circled in the merged row.
4. The output state is included with each stable state in the merged row.

Now apply these rules to the primitive flow table shown previously.

To see how this is done the primitive flow table is separated into two parts of three rows each:

	DG 00	01	11	10
a	c , –	(a), 0	b , –	– , –
c	(c), 0	a , –	– , –	d , –
d	c , –	– , –	b , –	(d), 0

	DG 00	01	11	10
b	– , –	a , –	(b), 1	e , –
e	f , –	– , –	b , –	(e), 1
f	(f), 1	a , –	– , –	e , –

States that are candidates for merging

Each part shows three stable states that can be merged because there no conflicting entries in each of the four columns.

Since a dash represents a *don't care* condition it can be associated with any state or output.

The first column of can be merged into a stable state c with output 0, the second into a stable state a with output 0, etc.

The resulting reduced flow table is as follows:

	DG 00	01	11	10
a, c, d	(c), 0	(a), 0	b , –	(d), 0
b, e, f	(f), 1	a , –	(b), 1	(e), 1

	DG 00	01	11	10
a	(a), 0	(a), 0	b , –	(a), 0
b	(b), 1	a , –	(b), 1	(b), 1

Reduced table (two alternatives)

3.2 Transition Table and Logic Diagram

To obtain the circuit described by the reduced flow table, a binary value must be assigned to each state. This converts the flow table to a transition table.

In assigning binary states, care must be taken to ensure that the circuit will be free of critical races. No critical races can occur in a two-row flow table.

Assigning 0 to state a and 1 to state b in the reduced flow table, the following transition table is obtained:

	DG 00	01	11	10
a	(a), 0	(a), 0	b , –	(a), 0
b	(b), 1	a , –	(b), 1	(b), 1

→

y	DG 00	01	11	10
0	0	0	1	0
1	1	0	1	1

The transition table is, in effect, a map for the excitation variable Y. The simplified Boolean function for Y as obtained from the map is:

$$Y = DG + G'y$$

There are two *don't care* outputs in the final reduced flow table. By assigning values to the output as shown below:

	DG 00	01	11	10
a	(a), 0	(a), 0	b , –	(a), 0
b	(b), 1	a , –	(b), 1	(b), 1

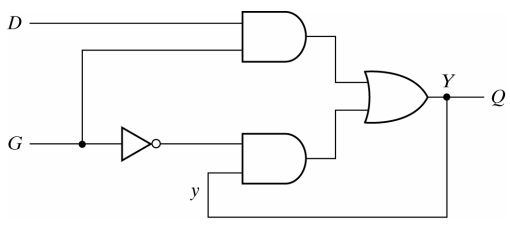
→

y	DG 00	01	11	10
0	0	0	1	0
1	1	0	1	1

it is possible to make output Q equal to Y.

If the other possible values are assigned to the don't care outputs, output Q is made equal to y.

In either case, the logic diagram of the gated latch is as follows:



The diagram can be also implemented by means of an *SR* latch.

Using the procedure outlined previously (i.e., from a given transition table), we first obtain the Boolean functions for *S* and *R* as shown below:

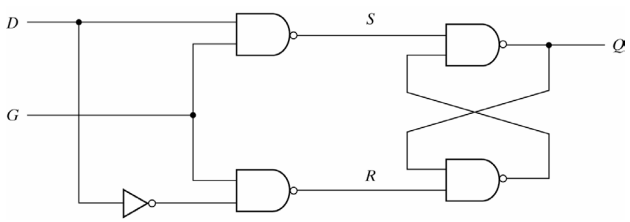
y	DG 00	01	11	10
0	0	0	1	0
1	X	0	X	X

$S = DG$

y	DG 00	01	11	10
0	X	X	0	X
1	0	1	0	0

$R = D'G$

When a NAND *SR* latch is used the logic diagram is as shown below:



The gated latch is a level-sensitive *D*-latch.

3.3 Assigning Outputs to Unstable States

The stable states in a flow table have specific output values associated with them. The unstable states have unspecified output values denoted by a dash. Consider the following flow table (a):

a	(*a*), 0	*b* , –
b	*c* , –	(*b*), 0
c	(*c*), 1	*d* , –
d	*a* , –	(*d*), 1

(a) Flow table

0	0
X	0
1	1
X	1

(b) Output assignm

Now consider the transition between two stable states via an unstable state.

Case 1: Both stable states have a 0 or a 1 output value.

Case 2: The stable states have different output values (0 and 1 or 1 and 0).

The correct output values that must be assigned to each state are listed in table (b) above.

4. Reduction of State and Flow Tables

The procedure for reducing the number of internal states in an asynchronous sequential circuit resembles the procedure that is used for synchronous circuits.

4.1 Implication Table

The state-reduction procedure for completely specified state tables is based on the algorithm that two states in a state table can be combined into one if they can be shown to be equivalent.

There are occasions when a pair of states do not have the same next states, but, nonetheless, go to equivalent next states. Consider the following state table:

Present State	Next State x = 0	Next State x = 1	Output x = 0	Output x = 1
a	*c*	*b*	0	1
b	*d*	*a*	0	1
c	*a*	*d*	1	0
d	*b*	*d*	1	0

(*a*, *b*) *imply* (*c*, *d*) and (*c*, *d*) *imply* (*a*, *b*). Both pairs of states are equivalent; i.e., *a* and *b* are equivalent as well as *c* and *d*.

The checking of each pair of states for possible equivalence in a table with a large number of states can be done systematically by means of an *implication table*. This a chart that consists of squares, one for every possible pair of states, that provide spaces for listing any possible implied states. Consider the following state table:

Present State	Next State x = 0	Next State x = 1	Output x = 0	Output x = 1
a	*d*	6	0	0
b	*e*	*a*	0	0
c	*g*	*f*	0	1
d	*a*	*d*	1	0
e	*a*	*d*	1	0
f	*c*	*b*	0	0
g	*a*	*e*	1	0

The implication table is:

	a	*b*	*c*	*d*	*e*	*f*
b	*d* , *e* ✓					
c	×	×				
d	×	×	×			
e	×	×	×	✓		
f	*c* , *d* ×	*c* , *e* × *a* , *b*	×	×	×	
g	×	×	×	*d* , *e* ✓	*d* , *e* ✓	×

On the left side along the vertical are listed all the states defined in the state table except the last, and across the bottom horizontally are listed all the states except the last.

The states that are not equivalent are marked with a 'x' in the corresponding square, whereas their equivalence is recorded with a '√'.

Some of the squares have entries of implied states that must be further investigated to determine whether they are equivalent or not.

The *step-by-step* procedure of filling in the squares is as follows:

1. Place a cross in any square corresponding to a pair of states whose outputs are not equal for every input.
2. Enter in the remaining squares the pairs of states that are implied by the pair of states representing the squares. We do that by starting from the top square in the left column and going down and then proceeding with the next column to the right.
3. Make successive passes through the table to determine whether any additional squares should be marked with a 'x'. A square in the

table is crossed out if it contains at least one implied pair that is not equivalent.

4. Finally, all the squares that have no crosses are recorded with check marks. The equivalent states are: (*a*, *b*), (*d*, *e*), (*d*, *g*), (*e*, *g*).

We now combine pairs of states into larger groups of equivalent states. The last three pairs can be combined into a set of three equivalent states (*d*, *e*, *g*) because each one of the states in the group is equivalent to the other two. The final partition of these states consists of the equivalent states found from the implication table, together with all the remaining states in the state table that are not equivalent to any other state:

(*a*, *b*) (*c*) (*d*, *e*, *g*) (*f*)

The reduced state table is:

Present State	Next State		Output	
	x = 0	x = 1	x = 0	x = 1
a	*d*	*a*	0	0
c	*d*	*f*	0	1
d	*a*	*d*	1	0
f	*c*	*a*	0	0

4.2 Merging of the Flow Table

There are occasions when the state table for a sequential circuit is incompletely specified.

Incompletely specified states can be combined to reduce the number of states in the flow table. Such states cannot be called equivalent, but, instead they are said to be *compatible*.

The process that must be applied in order to find a suitable group of compatibles for the purpose of merging a flow table is divided into three steps:

1. Determine all compatible pairs by using the implication table.
2. Find the maximal compatibles using a merger diagram.
3. Find a minimal collection of compatibles that covers all the states and is closed.

We will now proceed to show and explain the three procedural steps using the following primitive flow table:

	DG			
	00	01	11	10
a	*c*, –	(*a*), 0	*b*, –	–, –
b	–, –	*a*, –	(*b*), 1	*e*, –
c	(*c*), 0	*a*, –	–, –	*d*, –
d	*c*, –	–, –	*b*, –	(*d*), 0
e	*f*, –	–, –	*b*, –	(*e*), 1
f	(*f*), 1	*a*, –	–, –	*e*, –

4.3 Compatible Pairs

Two states are compatible if in every column of the corresponding rows in the flow table, they are identical or compatible states and if there is no conflict in the output values.

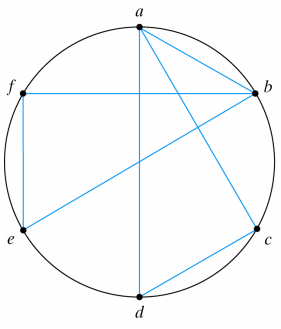
The compatible pairs (√) are:

(*a*, *b*) (*a*, *c*) (*a*, *d*) (*b*, *e*) (*b*, *f*) (*c*, *d*) (*e*, *f*)

b	√				
c	√	*d*, *e* ×			
d	√	*d*, *e* ×	√		
e	*c*, *f* ×	√	*d*, *e* × *c*, *f* ×	×	
f	*c*, *f* ×	√	×	*d*, *e* × *c*, *f* ×	√
	a	*b*	*c*	*d*	*e*

4.4 Maximal Compatibles

The *maximal compatible* is a group of compatibles that contains all the possible combinations of compatible states. The maximal compatible can be obtained from a merger diagram:



The above merger diagram is obtained from the list of compatible pairs derived from the previous implication table. A line represents a compatible pair. A triangle constitutes a compatible with three states. The maximal compatibles are:

(*a*, *b*) (*a*, *c*, *d*) (*b*, *e*, *f*)

In the case where a state is not compatible to any other state, an isolated **dot** represents this state.

5. Race-Free State Assignment

The main objective in choosing a proper binary state assignment is the prevention of critical races.

Critical races are avoided when states between which transitions occur in a flow table are given adjacent assignments. (e.g., 010 and 111 are adjacent).

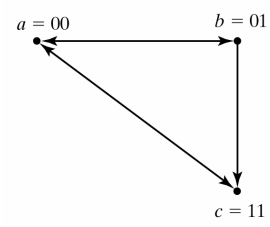
No critical races can occur in a two-row flow table.

5.1 Three-Row Flow Table Example

Consider the following reduced flow-table. For simplicity the outputs have been omitted:

	x_1x_2 00	01	11	10
a	(a)	b	c	(a)
b	a	(b)	(b)	c
c	a	(c)	(c)	(c)

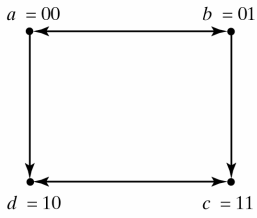
In row a there is a transition from state a to state c and from state a to state c. This information is transferred into a *transition diagram*:



The binary state assignment in the transition table will cause a critical race during the transition from a to c because there are two changes in the binary state variables.

A race-free assignment can be obtained by adding an extra row to the flow table:

	x_1x_2 00	01	11	10
a	(a)	b	d	(a)
b	a	(b)	(b)	c
c	d	(c)	(c)	(c)
d	a	–	c	–



The use of a fourth row does not increase the number of binary state variables, but allows the formation of *cycles* between two stable states.

The resulting transition table is shown below:

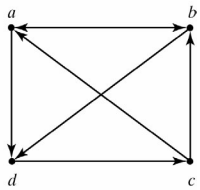
	x_1x_2 00	01	11	10
$a = 00$	(00)	01	10	(00)
$b = 01$	00	(01)	(01)	11
$c = 11$	10	(11)	(11)	(11)
$d = 10$	00	–	11	–

The two dashes represent unspecified states and can be considered *don't care* conditions. However, 10 must not be assigned to these squares to avoid an unwanted stable state in the fourth row.

5.2 Four-Row Flow Table Example

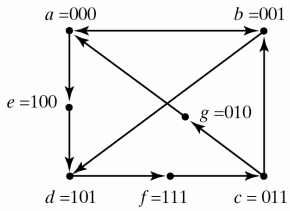
A flow table with four rows requires a minimum of two state variables. Consider the following flow table and its corresponding transition diagram:

	00	01	11	10
a	b	(a)	d	(a)
b	(b)	d	(b)	a
c	(c)	a	b	(c)
d	c	(d)	(d)	c



A state assignment map that is suitable for any four-row flow table is shown below:

y_1y_2 y_3	00	01	11	10
0	a	b	c	g
1	e	d	f	



States a, b, c, and d are the original states, and e, f, and g are extra states. The assignment ensures that a *cycle* is produced so that only one binary variable changes at a time.

By using the assignment given by the map, the four-row table can be expanded to a seven-row table that is *free* of critical races:

	00	01	11	10
000 = *a*	*b*	(*a*)	*e*	(*a*)
001 = *b*	(*b*)	*d*	(*b*)	*a*
011 = *c*	(*c*)	*g*	*b*	(*c*)
010 = *g*	–	*a*	–	–
110 –	–	–	–	–
111 = *f*	*c*	–	–	*c*
101 = *d*	*f*	(*d*)	(*d*)	*f*
100 = *e*	–	–	*d*	–

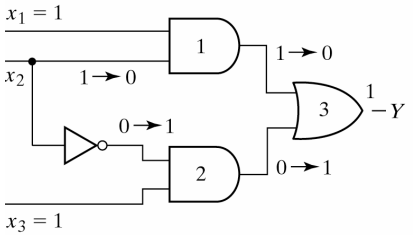
6. Hazards

Hazards are unwanted switching transients that may appear at the output of a circuit because different paths exhibit different propagation delays.

Hazards occur in combinational circuits, where they may cause a temporary false-output value. When this condition occurs in asynchronous sequential circuits, it may result in a transition to a wrong stable state.

6.1 Hazards in Combinational Circuits

The following circuit demonstrates the occurrence of a hazard:



Assume that all three inputs are initially equal to 1.

Then consider a change of x_2 from 1 to 0. The output momentarily may go to 0 if the propagation through the inverter is taken into account.

The circuit implements the Boolean function in *sum-of-products*:

$$Y = x_1x_2 + x_2'x_3$$

This type of implementation may cause the output to go to 0 when it should remain a 1. This is known as a *static 1-hazard*:

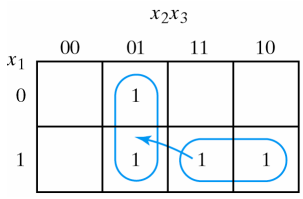
If the circuit was implemented in *product-of-sums*, namely:

$$Y = (x_1 + x_2')(x_2 + x_3)$$

Then the output may momentarily go to 1 when it should remain 0. This is referred to as a *static 0-hazard*:

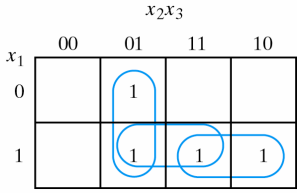
A third type of hazard, known as *dynamic hazard* causes the output to change 2 or 3 time when it should be change from 1 to 0 or 0 to 1:

The occurrence of the hazard can be detected by inspecting the map of the particular circuit:



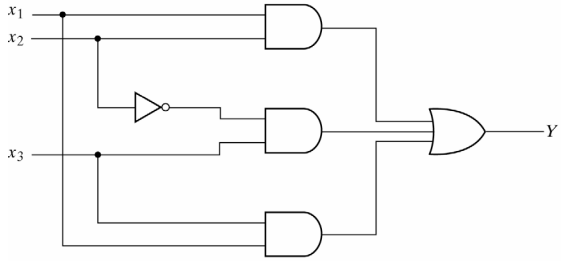
$$Y = x_1x_2 + x_2'x_3$$

The remedy for eliminating a hazard is to enclose the two minterms in question with another product term that overlaps both groupings:



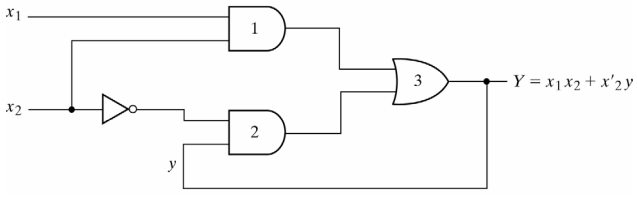
$$Y = x_1x_2 + x_2'x_3 + x_1x_3$$

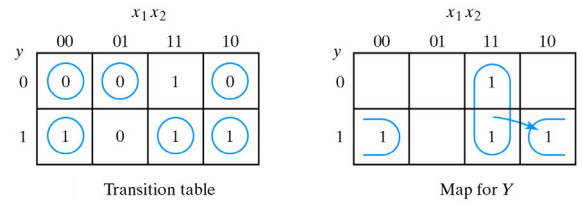
The hazard-free circuit is:



6.2 Hazards in Sequential Circuits

Consider the following asynchronous sequential circuit:





If the circuit is in total state yx_1x_2 = 111 and input x_2 changes from 1 to 0, the next total state should be 110. However, because of the *hazard*, output *Y* may go 0 momentarily.

If this false signal feeds back into gate 2 before the output of the inverter goes to 1, the output of gate 2 will remain at 0 and the circuit will switch to the incorrect total state 010.

This can be eliminated by adding an extra gate.

6.3 Implementation with *SR* Latches

An alternative way to avoid static hazards is to realize the asynchronous sequential circuit with *SR* latches.

A momentary 0 signal applied to the *S* or *R* inputs of a NOR latch will have no effect on the state of the latch.

A momentary 1 signal applied to the *S* or *R* inputs of a NAND latch will have no effect on the state of the latch.

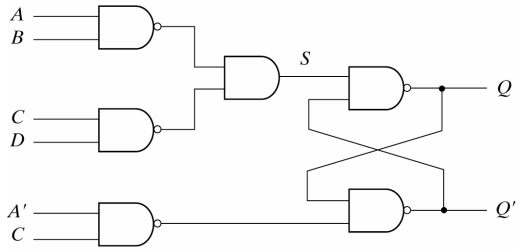
Consider a NAND *SR* latch with the following Boolean functions for *S* and *R*:

$$S = AB + CD$$
$$R = A'C$$

Since this is a NAND latch we must apply the complemented values to the inputs:

$$S = (AB + CD)' = (AB)'(CD)'$$
$$R = (A'C)'$$

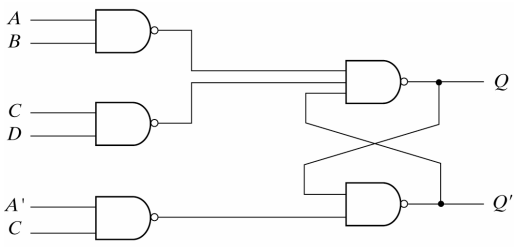
This results in the following implementation:



The Boolean function for output *Q* is:

$$Q = (Q'S)' = [Q'(AB)'(CD)']$$

The above function may also be generated with two levels of NAND gates:



If output *Q* is equal to 1, then *Q'* is equal to 0. If two of the three inputs go momentarily to 1, the NAND gate associated with output *Q* will remain at 1 because *Q'* is maintained at 0.

6.4 Essential Hazards

An *essential hazard* is the result of the effects of a single input variable change reaching one feedback path before another feedback path.

Essential hazards cannot be corrected by adding redundant gates as in static hazards.

They can always be eliminated in a realization by the insertion of sufficient delays in the feedback paths. Facility in doing this comes only with experience.