Simplification of State Machines

This semester has been spent developing the techniques needed to design digital circuits. Now we can compile a list of just what it takes to complete a successful digital design project.

Specify the problem

Derive the state table and/or the state diagram

Reduce the # of states to the minimum possible

Choose an appropriate state assignment

Construct an appropriate transition table

Develop the design equations

Implement the circuit

The only topic in that list that is left to be discussed is that of minimizing the number of states. Discussion of State Reduction is normally divided into two different areas:

A. State Reduction in Completely Specified Circuits

B. State Reduction in Incompletely Specified Circuits

We will limit our discussion to the first area. This topic is expanded on in EET 420 and usually encompasses the second discussion area.

The removal of redundant states is important for a number of reasons.

(1) Cost: The number of memory elements is directly related to the number of states.

(2) Complexity: The more states the circuit contains, the more complex the design and its associated implementation becomes.

(3) Aids failure analysis: Diagnostic routines are usually based on the assumption that there are no redundant states.

There are 3 major State reduction Methods: Reduction by

(a) Inspection (b) Partitioning (c) Implication table.

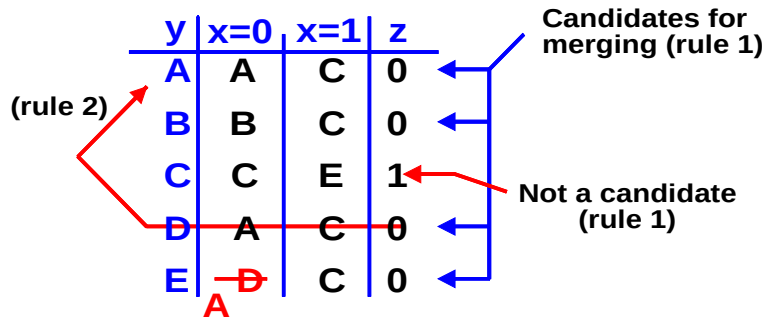
We will study the 1st two methods and leave the Implication table method for EET 420.

Reduction By Inspection

This is the easiest and most obvious technique. We can merge compatible states if we follow some simple rules.

1. Two states that have different outputs are not compatible and can't be merged.
2. Two states are compatible and can be merged **if**, **under all possible input conditions**, the **next states and outputs are the same**.
3. Two states are compatible and can be merged if, under all possible input conditions, the merging of the two states will **make the next states compatible and the outputs the same**.

The following is an example from Professor Hackworth's EET420 text.



y	x=0	x=1	z
A	A	C	0
B	B	C	0
C	C	~~E~~ A	1
~~E~~	~~A~~	~~C~~	~~0~~

(rule 2)

Note the substitution of all E's with A's along with the merger.

y	x=0	x=1	z
A	A	C	0
~~B~~ A	~~B~~	C	0
C	C	A	1

This application of rule 3 is quite often hard to notice. That's why we need other methods of state reduction.

(rule 3) if B were merged with A then we could replace B with A and then the Next states and the outputs would match.

y	x=0	x=1	z
A	A	C	0
C	C	A	1

Note that with this final circuit we have made quite a hardware savings. We went from 5 states (3 ff) to 2 states (1 ff) along with a significant reduction in control circuitry.

Let's look at another small example. In doing so, we will also look at a couple of new ways to write state tables which will help us in our reductions.

X	y	Y	Z		
0	A	B	0	☜	
0	B	C	0	☜	
0	C	D	1		←
0	D	C	0	☜	
1	A	C	1		←
1	B	A	1		←
1	C	B	0	☜	
1	D	A	1		←

Two groups of possible candidates for merging.

If we merge A and B, would the results be the same?

X = 0 {
AB ⇒ ~~BC~~
AD ⇒ ~~BC~~
BD ⇒ CC
}

BC's outputs (z) are different therefore the two states are not compatible.

X = 1 {
AB ⇒ ~~AC~~
AD ⇒ ~~AC~~
BD ⇒ AA
}

AC's outputs (z) are different therefore the two states are not compatible.

Let's do the same thing except this time, let's put the table into another form.

	y	x=0 Y/z	x=1 Y/z
	A	B/0	C/1
	B	C/0	A/1
	C	D/1	B/0
	~~D~~	~~C/0~~	~~A/1~~

⇨

y	x=0 Y/z	x=1 Y/z
A	B/0	C/1
B	C/0	A/1
C	B/1	B/0

Note that since D was redundant, **we replaced every D with a B.**

What did we save with this State Merger? Saved on gates, not on FF's.

Let's rewrite the table another way ONE MORE TIME.

	y	y x=0	y x=1	z x=0	z x=1
	A	B	C	0	1
	B	C	A	0	1
	C	D	B	1	0
	~~D~~	~~C~~	~~A~~	~~0~~	~~1~~

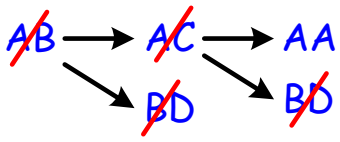
⇨

y	y x=0	y x=1	z x=0	z x=1
A	B	C	0	1
B	C	A	0	1
C	B	B	1	0

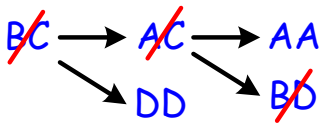
Let's try a second example. We will use the "By Inspection" method to reduce the following state table:

y	Y x=0	Y x=1	z x=0	z x=1
A	A	B	0	0
B	C	D	0	0
C	A	D	0	0
D	E	F	0	1
E	A	F	0	1
F	G	F	0	1
G	A	F	0	1

Let's look at the instances where **Z is equal to 00:**



State AB had a NEXT STATE of AC and BD.
Following the AC thread we get:
State AC has NEXT STATES of AA and BD.
By definition, AA will be fine, but BD is not compatible because B has an output of 00 and D has an output of 01. If one item in any thread of a state inspection is incompatible, the whole thread (and the state) is bad (fruit of the poison tree) as well!!
We don't even have to look at the other thread coming off of AB but if you like, note that the thread starts with BD which we have already determined to be incompatible.



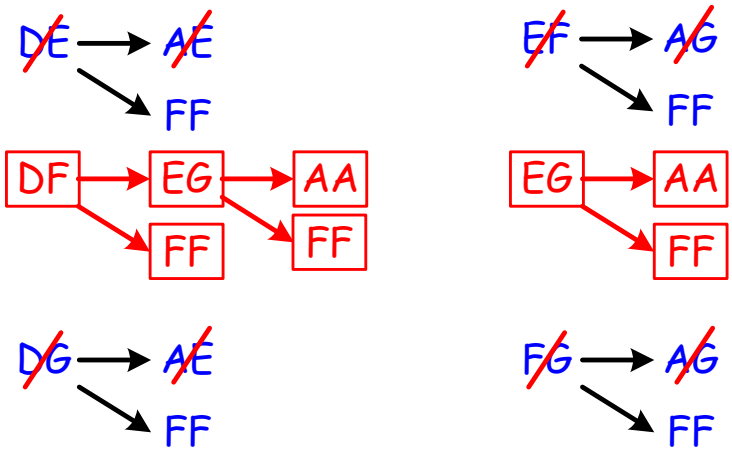
BC is the other possibility for Z = 00. Note that if we work thru the top thread, we get to BC which we have shown to be incompatible, therefore **the entire tree is poisoned.**

Continued on the next page:

Continuing the example, let's now look at the output condition of **Z = 01**

y	Y x=0	Y x=1	z x=0	z x=1
A	A	B	0	0
B	C	D	0	0
C	A	D	0	0
D	E	F	0	1
E	A	F	0	1
F	G	F	0	1
G	A	F	0	1

This time, we will use less explanation for the failures. Try to follow why each tree failed:



y	Y x=0	Y x=1	Z x=0	Z x=1
A	A	B	0	0
B	C	D	0	0
C	A	D	0	0
D	E	~~F~~ D	0	1
E	A	~~F~~ D	0	1
~~F~~	~~G~~	~~F~~	~~0~~	~~1~~
~~G~~	~~A~~	~~F~~	~~0~~	~~1~~

y	Y x=0	Y x=1	Z x=0	Z x=1
A	A	B	0	0
B	C	D	0	0
C	A	D	0	0
D	E	D	0	1
E	A	D	0	1

Note that we replaced the single G with an E and all the occurrences of F with D. We reduced the circuit from 7 states (3 FF) to 5 states (3FF). We still didn't save on flip-flops but we most likely saved on control circuitry.

Moore Reduction
(also known as "Partitioning)

(page 581)

	Y		z	
y	X=0	X=1	X=0	X=1
A	D	B	0	0
B	E	A	0	0
C	G	F	0	1
D	A	D	1	0
E	A	F	1	0
F	C	B	0	0
G	A	E	1	0

1. The 1st step is to assume that all states are compatible.

   P_0=(ABCDEFG)

2. Then we start grouping the states based on their outputs.

   P_1=(ABF)(C)(DEG)

3. Then we group these groups by Next State.

For ABF, x=0,

The next state is DEC. Note that in P_1, DE is in the same group (DEG) so AB can stay in the same group. However, C is not in the same group as DE was so F will have to be separated out into its own group. We now have **partition 2** (P_2):

P_2=(AB)(C)(DEG)(F)

For ABF, x=1,

The next state is ABB which becomes AB. AB were in the same group in the last partition, P_2 so they can still share a group in P_3.

P_3=(AB)(C)(DEG)(F)

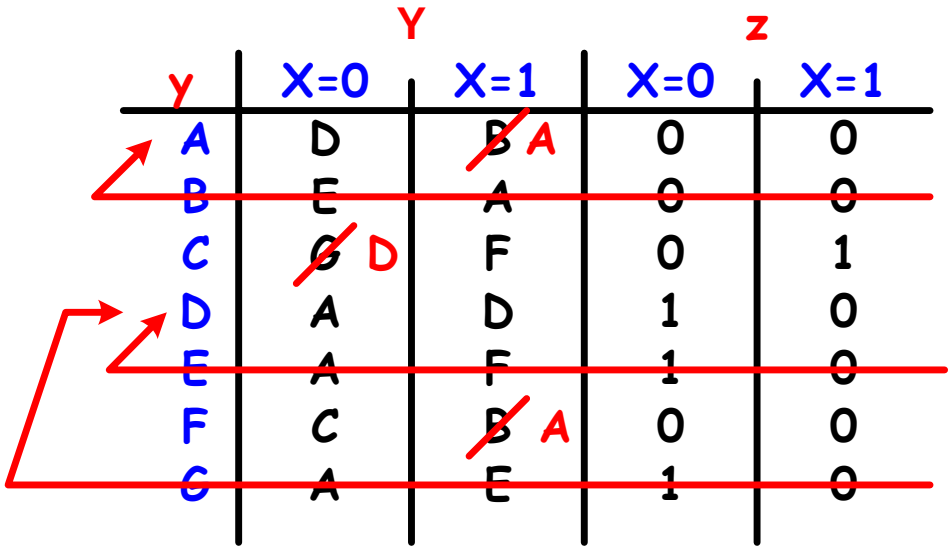
For DEG, X=0,

Next state is AAA. Obviously they are in same group in P_3 so DEG can remain grouped. P_4=(AB)(C)(DEG)(F)

For DEG, X=1,

Next state is DDE which are all in the same group in P_4, so we can keep DEG together. P_5=(AB)(C)(DEG)(F)

Once a partition repeats itself, you are finished. So, for this group, we get:



	Y		z	
y	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

Reduced from

7 states (3 ff)
to
4 states (2 ff) + control circuitry.

Let's redo our example on page 7 by using Moore reduction.

	Y		z	
y	x=0	x=1	x=0	x=1
A	A	B	0	0
B	C	D	0	0
C	A	D	0	0
D	E	F	0	1
E	A	F	0	1
F	G	F	0	1
G	A	F	0	1

1. Assume the all are compatible.

   P_0=(ABCDEFG)

2. Group by output.

   P_1=(ABC)(DEFG)

3. Group by next state.

   P_2 = (A)(BC)(DF)(EG)

   P_3 = (A)(B)(C)(DF)(EG)

   P_4 = (A)(B)(C)(DF)(EG)

   P_3 = P_4 thus reduced

	Y		z	
y	x=0	x=1	x=0	x=1
A	A	B	0	0
B	C	D	0	0
C	A	D	0	0
D	E	~~F~~ D	0	1
E	A	~~F~~ D	0	1
F	G	F	0	1
G	A	F	0	1

	Y		z	
y	x=0	x=1	x=0	x=1
A	A	B	0	0
B	C	D	0	0
C	A	D	0	0
D	E	D	0	1
E	A	D	0	1

Same answer as before.

(Example 9.3 in Nelson, et. al.) Reduce the following stable by applying Moore's Reduction.

y	Y x=0	Y x=1	z x=0	z x=1
A	E	D	0	0
B	A	F	1	0
C	C	A	0	1
D	B	A	0	0
E	D	C	1	0
F	C	D	0	1
G	H	G	1	1
H	C	B	1	1

P_0=(ABCDEFGH)

P_1=(AD)(BE)(CF)(GH)

P_2=(AD)(BE)(CF)(G)(H)

P_3=(AD)(BE)(CF)(G)(H)

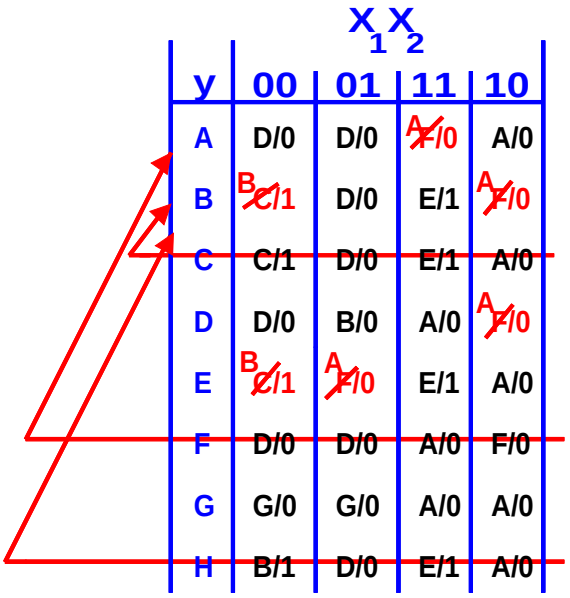
$P_3 = P_2$, Thus reduced

y	Y x=0	Y x=1	z x=0	z x=1
A	~~E~~ B	~~D~~ A	0	0
B	A	~~F~~ C	1	0
C	C	A	0	1
~~D~~	~~B~~	~~A~~	~~0~~	~~0~~
~~E~~	~~D~~	~~C~~	~~1~~	~~0~~
~~F~~	~~C~~	~~D~~	~~0~~	~~1~~
G	H	G	1	1
H	C	B	1	1

y	Y x=0	Y x=1	z x=0	z x=1
A	B	A	0	0
B	A	C	1	0
C	C	A	0	1
G	H	G	1	1
H	C	B	1	1

Example 9.4 (in Nelson, et. al.)

y	X_1X_2 00	01	11	10	
A	D/0	D/0	F/0	A/0	P_0=(ABCDEFGH)
B	C/1	D/0	E/1	F/0	
C	C/1	D/0	E/1	A/0	P_1=(ADFG)(BCEH)
D	D/0	B/0	A/0	F/0	P_2=(AFG)(D)(BCEH)
E	C/1	F/0	E/1	A/0	
F	D/0	D/0	A/0	F/0	P_3=(AF)(D)(G)(E)(BCH)
G	G/0	G/0	A/0	A/0	P_4=(AF)(D)(G)(E)(BCH)
H	B/1	D/0	E/1	A/0	P_3=P_4, Thus reduced



y	X_1X_2 00	01	11	10
A	D/0	D/0	A/0	A/0
B	B/1	D/0	E/1	A/0
D	D/0	B/0	A/0	A/0
E	B/1	A/0	E/1	A/0
G	G/0	G/0	A/0	A/0

State Assignments

State assignment procedures are concerned with methods for assigning binary values to states in such a way as to reduce the cost of the combinational circuit which drives the flip-flops (memory elements). These methods are mainly useful when a circuit is viewed as a black box. Such a circuit may follow a sequence of internal states, but the binary values of the individual states may be of no consequence outside of the box as long as the circuit produces the required sequence of output bits for any sequence of input bits. By this definition, this assignment procedure obviously doesn't apply to circuits whose external outputs are taken directly from flip-flops with a particular binary sequence specified.

There are many methods of optimizing the state assignments made to a given circuit. Most are extremely involved and are beyond the scope of this course. However, one particular method of state assignment optimization is essentially simple and easy to follow.

In the following rules, there term "adjacent assignments" means that the state assignments made to a pair of states differ by only 1 bit, i.e. "Gray Codes".

Rule 1: States that have the same next states for a given input should be given logically adjacent assignments.

Rule 2: States that are the next states of a given present state, under logically adjacent inputs, should be given logically adjacent assignments.

Rule 3: If there is a conflict between rule 1 and 2, rule 1 takes precedence.

Let's apply these rules to an example:

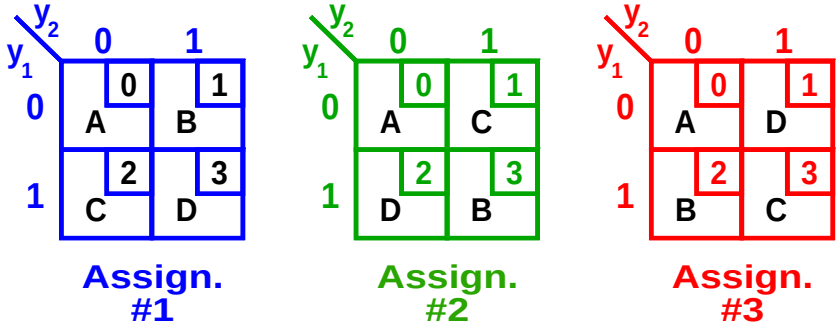
Assign. #1	Assign. #2	Assign. #3	y	Y x=0	Y x=1	z x=0	z x=1
00	00	00	A	A	B	0	0
01	11	10	B	A	C	0	0
10	01	11	C	D	C	0	0
11	10	01	D	A	B	0	1

Using Rule # 1, the state pair **(A,B)** should be given adjacent assignments because **both** of them **go to state A for x = 0**. Similarly, state pairs **(B,D)**, **(A,D)**, and **(B,C)** all should be given adjacent assignments; the pair **(A,D) appears twice**.

The application of Rule # 2 shows that the state pair **(A,B)** should be given adjacent assignments because **they are next states of the present state A.** For similar reasons, state pairs **(A,C) and (C,D)** should be adjacent with **(A,B) appearing twice**.

Rule 2:
States that are the next states of a given present state, under logically adjacent inputs, should be given logically adjacent assignments.

Rule 1:
States that have the same next states for a given input should be given logically adjacent



The figure above demonstrates the three different choices for state assignments which attempt to meet as many of the adjacency requirements as possible. It can hopefully be seen that **Assignment 3** satisfies most of the adjacencies and hence produces a better result then the other assignments. It might be suggested that **Assignment 2**

produces the same number of adjacencies as **#3**, but it doesn't fulfill the adjacency requirement for state pair **(A,B)** as determined by Rule 1. (Note that we don't bother here with an equation for z since it is not affected by a state assignment.)

Let's take a look at the logic equations produced by each assignment. The actual work to produce these equations follows but will not be covered in class. It is left up to the student to verify the equations for him or herself.

As can be easily seen below, our choice of Assign. #3 is clearly the winner in the logic component simplicity contest.

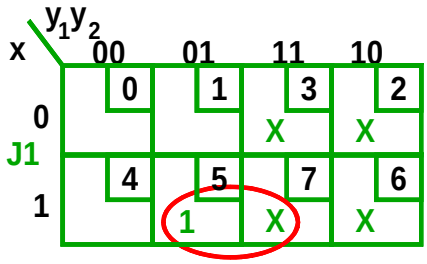
	J_1	K_1	J_2	K_2	*#gates*
#1	xy_2	y_2	$x\bar{y}_1 + \bar{x}y_1$ *or* $x \oplus y_1$	$\bar{x} + \bar{y}_1$	1 AND gate 1 OR gate 1 wire and a choice between: 1 XOR gate or 1 OR gate 2 additional AND gates
#2	$\bar{x}y_2 + x\bar{y}_2$ *or* $x \oplus y_2$	$\bar{x} + y_2$	x	$\bar{x}$	1 OR gate 2 wire and a choice between: 1 XOR gate or 1 additional OR gate 2 additional AND gates
#3	x	$\bar{x}$	xy_1	$\bar{y}_1$	1 AND gate 3 wires

State Assignment #1

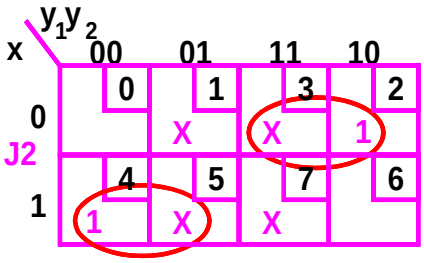
Assign #1	y
00	A
01	B
10	C
11	D

x	y_1	y_2		Y_1	Y_2		J_1	K_1	J_2	K_2
0	0	A	0	0	A	0	0	X	0	X
0	0	B	1	0	A	0	0	X	X	1
0	1	C	0	1	D	1	X	0	1	X
0	1	D	1	0	A	0	X	1	X	1
1	0	A	0	0	B	1	0	X	1	X
1	0	B	1	1	C	0	1	X	X	1
1	1	C	0	1	C	0	X	0	0	X
1	1	D	1	0	B	1	X	1	X	0

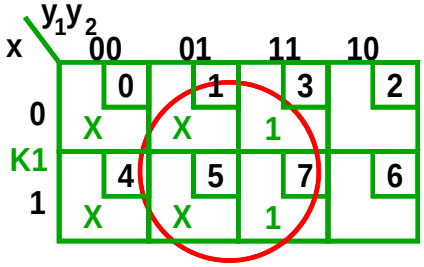
PS	NS	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0



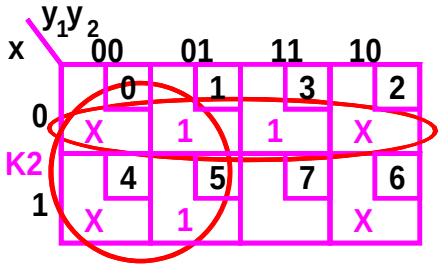
$J_1 = x + y_2$



$J_2 = x\overline{y_1} + \overline{x}y_1 = x \oplus y_1$



$K_1 = y_2$



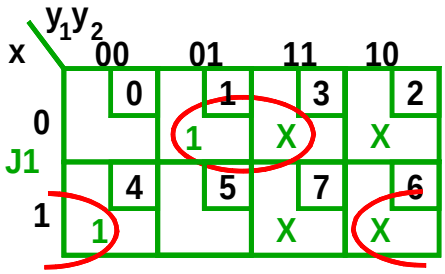
$K_1 = \overline{x} + \overline{y_1} = \overline{xy_1}$

State Assignment #2

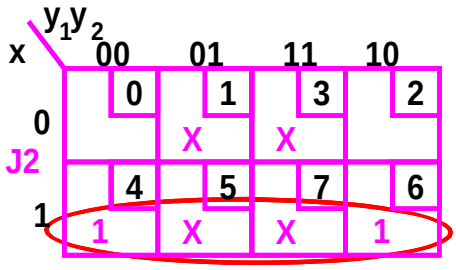
Assign #2	y
00	A
11	B
01	C
10	D

x	y_1	y_2		Y_1	Y_2		J_1	K_1	J_2	K_2
0	0	A	0	0	A	0	0	X	0	X
0	0	C	1	1	D	0	1	X	X	1
0	1	D	0	0	A	0	X	1	0	X
0	1	B	1	0	A	0	X	1	X	1
1	0	A	0	1	B	1	1	X	1	X
1	0	C	1	0	C	1	0	X	X	0
1	1	D	0	1	B	1	X	0	1	X
1	1	B	1	0	C	1	X	1	X	0

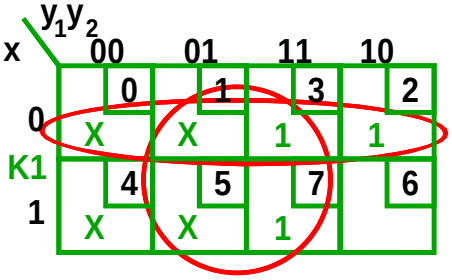
PS	NS	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0



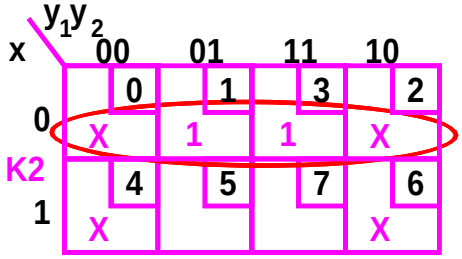
$J_1 = x\overline{y_2} + \overline{x}y_2 = x \oplus y_2$



$J_2 = x$



$K_1 = \overline{x} + y_2$



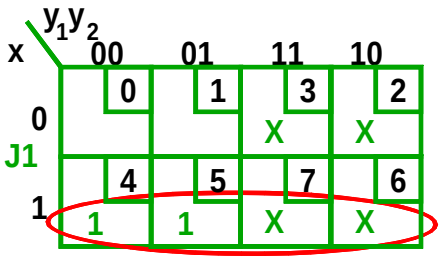
$K_2 = \overline{x}$

State Assignment #3

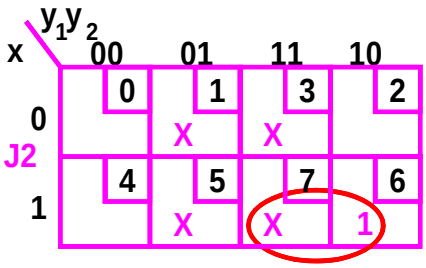
Assign #3	y
00	A
10	B
11	C
01	D

x	y_1	y_2		Y_1	Y_2		J_1	K_1	J_2	K_2
0	0	A	0	0	A	0	0	X	0	X
0	0	D	1	0	A	0	0	X	X	1
0	1	B	0	0	A	0	X	1	0	X
0	1	C	1	0	D	1	X	1	X	0
1	0	A	0	1	B	0	1	X	0	X
1	0	D	1	1	B	0	1	X	X	1
1	1	B	0	1	C	1	X	0	1	X
1	1	C	1	1	C	1	X	0	X	0

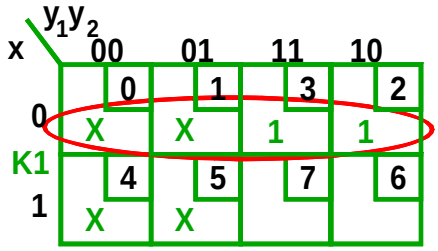
PS	NS	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0



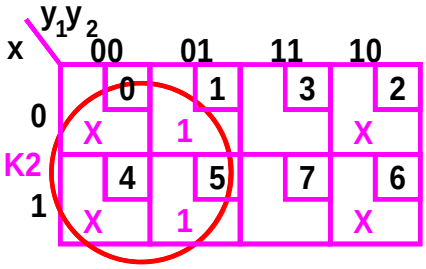
$J_1 = X$



$J_2 = xy_1$



$K_1 = \overline{x}$



$K_2 = \overline{y_1}$