Sequential Logic III

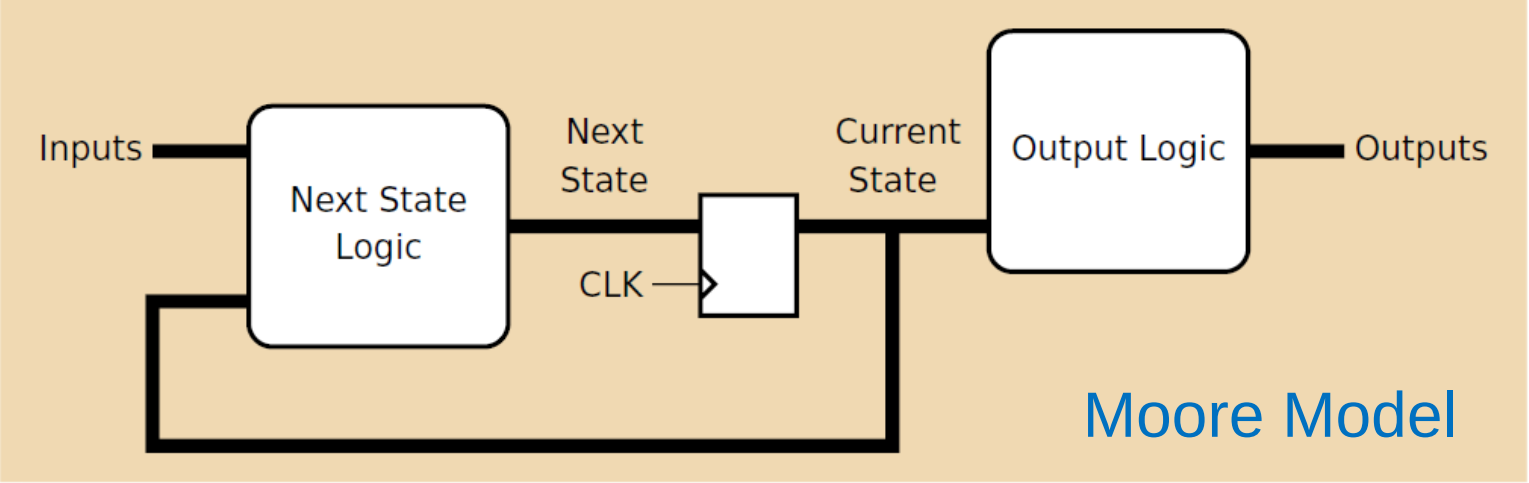
Design of Sequential Circuits

Sequential Circuit Design

- The **analysis** of sequential circuits starts from a circuit diagram and culminates in a state table or state diagram.
- In sequential circuit **design (synthesis)**, we reverse the process: we turn a set of descriptions/specifications into a working circuit.
  - We first make a state table or state diagram to express the computation.
  - Then we can turn that table or diagram into a sequential circuit, also known as *finite state machine* (FSM).

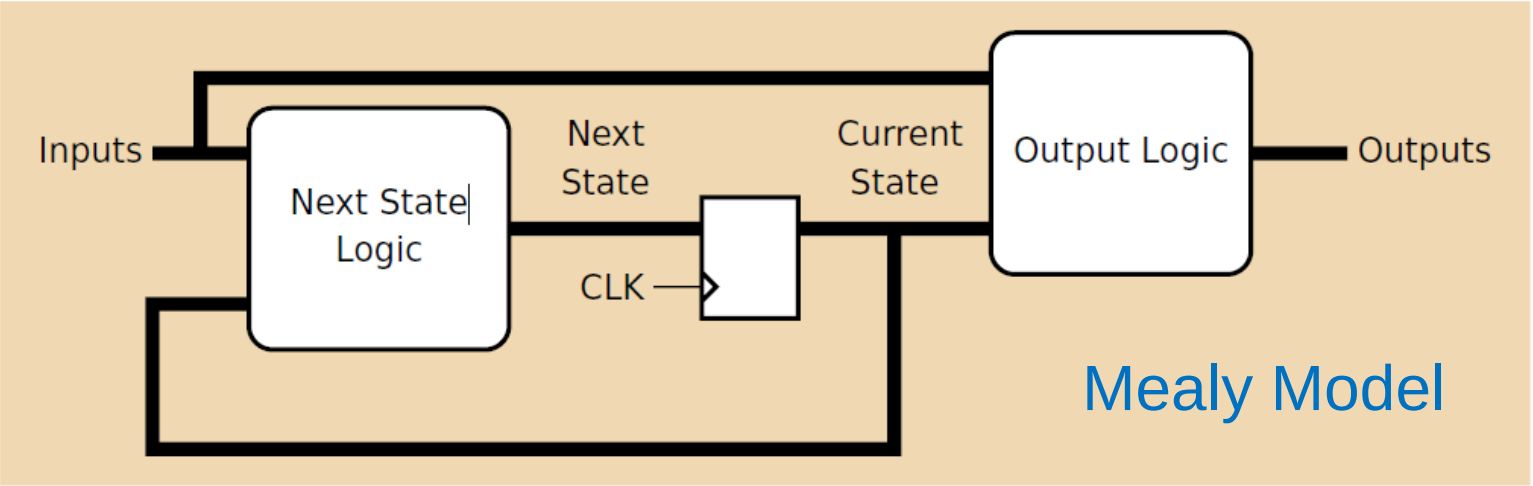
Finite State Machines

- There are two types of state machines: the Moore model, and the Mealy model.



- In the Moore model, the outputs are only a function of the current state. The outputs are synchronized.

Finite State Machines



- In the Mealy model, the outputs are a function of the current state and the external inputs.
- Outputs have immediate reaction to inputs without waiting for next clocking event. This means the outputs may change asynchronously.

Design Procedure

Step 1:

Make a state table based on the problem statement. The table should show the present states, inputs, next states and outputs. (It may be easier to find a state diagram first, and then convert into a table.)

Step 2:

Assign binary codes to the states in the state table. If you have n states, your binary codes will have at least ≥ $\log_2 n$ digits, and your circuit will have at least $\log_2 n$ flip-flops.

Step 3:

For each flip-flop and each row of your state table, find the flip-flop input values that are needed to generate the next state from the present state. You can employ flip-flop excitation tables to do that.

Step 4:

Find simplified equations for the flip-flop inputs and the outputs.

Step 5:

Build the circuit!

Example: Sequence Recognizers

- A sequence recognizer is a special kind of sequential circuit that looks for a special bit pattern in some input.
- The recognizer circuit has only one input, X.
  - One bit of input is supplied on every clock cycle. For example, it would take 20 cycles to scan a 20-bit input.
  - This is an easy way to permit arbitrarily long input sequences.
- There is one output, Z, which is 1 when the desired pattern is found.
- Our example will detect the bit pattern “1001”:

Inputs: 11**1001**10**1001001**10…
Outputs: 00000**1**00000**1**00**1**00…

Here, one input and one output bit appear every clock cycle.

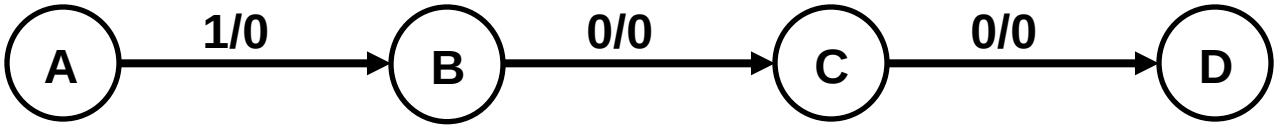
- This requires a sequential circuit because the circuit has to “remember” the inputs from previous clock cycles, in order to determine whether or not a match was found.

Step 1: Make a state table

- The first thing you have to figure out is precisely how the use of state will help you solve the given problem.
  - Make a state table based on the problem statement. The table should show the present states, inputs, next states and outputs.
  - Sometimes it is easier to first find a state diagram and then convert that to a table.
- This is usually the most difficult step. Once you have the state table, the rest of the design procedure is the same for all sequential circuits.
- Sequence recognizers are especially hard! They're the hardest example we'll see in this class, so if you understand this you're in good shape.

A basic state diagram (Mealy)

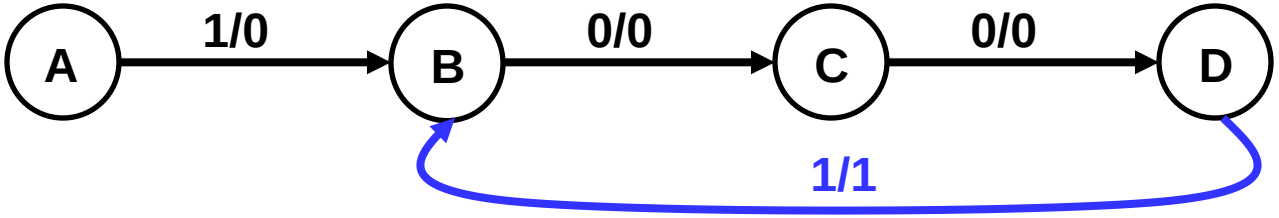
- What state do we need for the sequence recognizer?
  - We have to "remember" inputs from previous clock cycles.
  - For example, if the previous three inputs were 100 and the current input is 1, then the output should be 1.
  - In general, we will have to remember occurrences of parts of the desired pattern—in this case, 1, 10, and 100.
- We'll start with a basic state diagram:



State	Meaning
A	None of the desired pattern (1001) has been input yet.
B	We've already seen the first bit (1) of the desired pattern.
C	We've already seen the first two bits (10) of the desired pattern.
D	We've already seen the first three bits (100) of the desired pattern.

Overlapping occurrences of the pattern

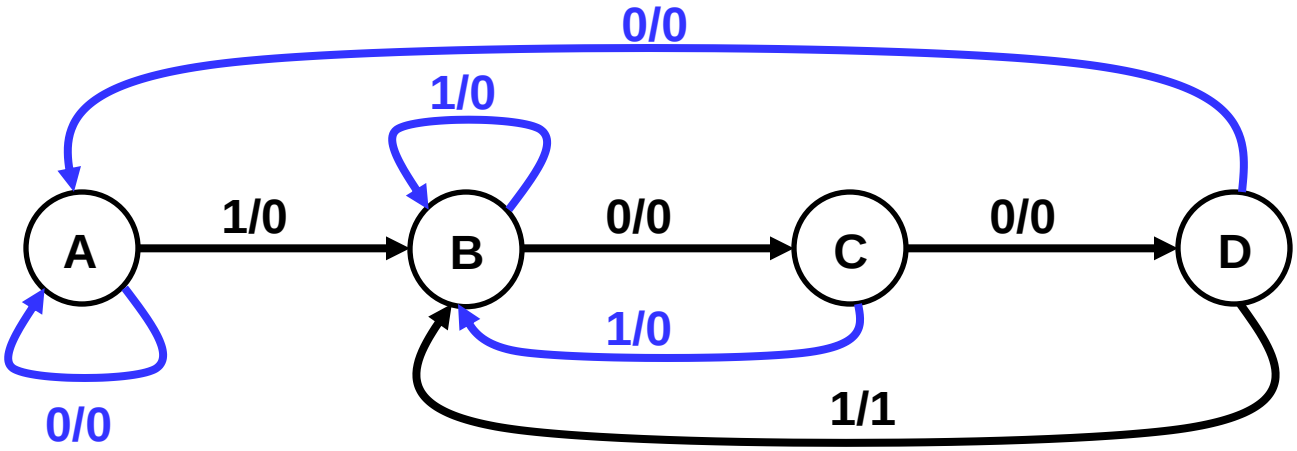
- What happens if we're in state D (the last three inputs were 100), and the current input is 1?
  - The output should be a 1, because we've found the desired pattern.
  - But this last 1 could also be the start of another occurrence of the pattern! For example, 1001001 contains *two* occurrences of 1001.
  - To detect overlapping occurrences of the pattern, the next state should be B.



State	Meaning
A	None of the desired pattern (1001) has been input yet.
B	We've already seen the first bit (1) of the desired pattern.
C	We've already seen the first two bits (10) of the desired pattern.
D	We've already seen the first three bits (100) of the desired pattern.

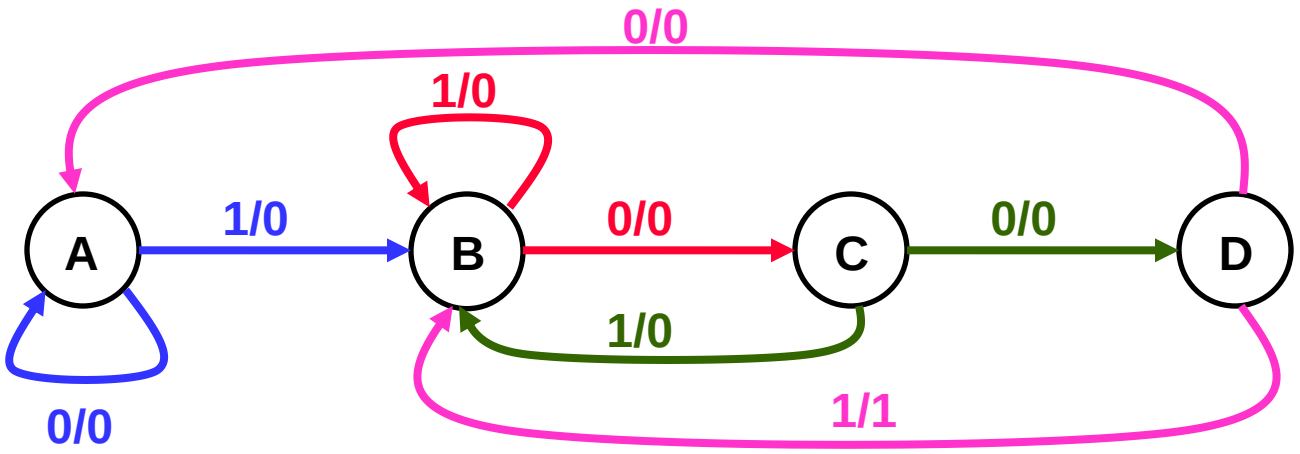
Filling in the other arrows

- Remember that we need *two* outgoing arrows for each node, to account for the possibilities of X=0 and X=1.
- The remaining arrows we need are shown in blue. They also allow for the correct detection of overlapping occurrences of 1001.

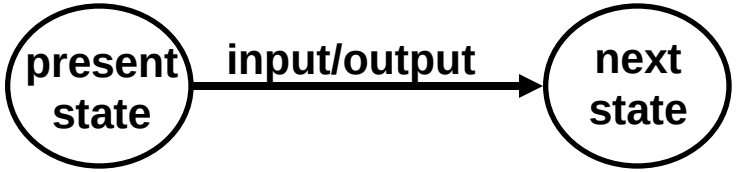


State	Meaning
A	None of the desired pattern (1001) has been input yet.
B	We've already seen the first bit (1) of the desired pattern.
C	We've already seen the first two bits (10) of the desired pattern.
D	We've already seen the first three bits (100) of the desired pattern.

Finally, making the state table



Remember how the state diagram arrows correspond to rows of the state table:

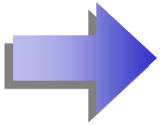


Present State	Input	Next State	Output
A	0	A	0
A	1	B	0
B	0	C	0
B	1	B	0
C	0	D	0
C	1	B	0
D	0	A	0
D	1	B	1

Step 2: Assign binary codes to states

- We have four states ABCD, so we need at least two flip-flops Q_1Q_0.
- The easiest thing to do is represent state A with Q_1Q_0 = 00, B with 01, C with 10, and D with 11.
- The state assignment can have a big impact on circuit complexity, but we won't worry about that too much in this class.

Present State	Input	Next State	Output
A	0	A	0
A	1	B	0
B	0	C	0
B	1	B	0
C	0	D	0
C	1	B	0
D	0	A	0
D	1	B	1



Present State Q_1	Present State Q_0	Input X	Next State Q_1	Next State Q_0	Output Z
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	1	1

Step 3: Find flip-flop input values

- Next we have to figure out how to actually make the flip-flops change from their present state into the desired next state.
- This depends on what kind of flip-flops you use!
- We'll use two JKs. For each flip-flip Q_i, look at its present and next states, and determine what the inputs J_i and K_i should be in order to make that state change.

Present State		Input	Next State		Flip flop inputs				Output
Q_1	Q_0	X	Q_1	Q_0	J_1	K_1	J_0	K_0	Z
0	0	0	0	0					0
0	0	1	0	1					0
0	1	0	1	0					0
0	1	1	0	1					0
1	0	0	1	1					0
1	0	1	0	1					0
1	1	0	0	0					0
1	1	1	0	1					1

Finding JK flip-flop input values

- For JK flip-flops, this is a little tricky. Recall the characteristic table:

J	K	Q(t+1)	Operation
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q'(t)	Complement

- If the present state of a JK flip-flop is 0 and we want the next state to be 1, then we have *two* choices for the JK inputs:
  - We can use JK=10, to explicitly set the flip-flop's next state to 1.
  - We can also use JK=11, to complement the current state 0.
- So to change from 0 to 1, we must set J=1, but K could be *either* 0 or 1.
- Similarly, the other possible state transitions can all be done in two different ways as well.

JK excitation table

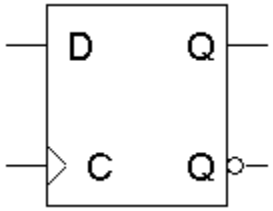
- An excitation table shows what flip-flop inputs are required in order to make a desired state change.

Q(t)	Q(t+1)	J	K	Operation
0	0	0	x	No change/reset
0	1	1	x	Set/complement
1	0	x	1	Reset/complement
1	1	x	0	No change/set

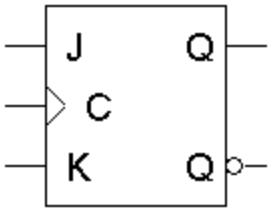
- This is the same information that's given in the characteristic table, but presented "backwards."

J	K	Q(t+1)	Operation
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q'(t)	Complement

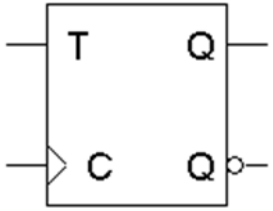
Excitation tables for all flip-flops



Q(t)	Q(t+1)	D	Operation
0	0	0	Reset
0	1	1	Set
1	0	0	Reset
1	1	1	Set



Q(t)	Q(t+1)	J	K	Operation
0	0	0	x	No change/reset
0	1	1	x	Set/complement
1	0	x	1	Reset/complement
1	1	x	0	No change/set



Q(t)	Q(t+1)	T	Operation
0	0	0	No change
0	1	1	Complement
1	0	1	Complement
1	1	0	No change

Back to the example

- We can now use the JK excitation table on the right to find the correct values for *each* flip-flop's inputs, based on its present and next states.

Q(t)	Q(t+1)	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Present State Q_1	Present State Q_0	Input X	Next State Q_1	Next State Q_0	Flip flop inputs J_1	Flip flop inputs K_1	Flip flop inputs J_0	Flip flop inputs K_0	Output Z
0	0	0	0	0					0
0	0	1	0	1					0
0	1	0	1	0					0
0	1	1	0	1					0
1	0	0	1	1					0
1	0	1	0	1					0
1	1	0	0	0					0
1	1	1	0	1					1

Back to the example

- We can now use the JK excitation table on the right to find the correct values for *each* flip-flop's inputs, based on its present and next states.

Q(t)	Q(t+1)	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Present State		Input	Next State		Flip flop inputs				Output
Q_1	Q_0	X	Q_1	Q_0	J_1	K_1	J_0	K_0	Z
0	0	0	0	0	0	x	0	x	0
0	0	1	0	1	0	x	1	x	0
0	1	0	1	0	1	x	x	1	0
0	1	1	0	1	0	x	x	0	0
1	0	0	1	1	x	0	1	x	0
1	0	1	0	1	x	1	1	x	0
1	1	0	0	0	x	1	x	1	0
1	1	1	0	1	x	1	x	0	1

Step 4: Find equations for inputs and outputs

- Now you can make Karnaugh-maps and find equations for each of the four flip-flop inputs, as well as for the output Z.
- These equations are in terms of the present state and the inputs.
- The advantage of using JK flip-flops is that there are many don't care conditions, which can result in simpler MSP equations.

Present State		Input	Next State		Flip flop inputs				Output
Q_1	Q_0	X	Q_1	Q_0	J_1	K_1	J_0	K_0	Z
0	0	0	0	0	0	x	0	x	0
0	0	1	0	1	0	x	1	x	0
0	1	0	1	0	1	x	x	1	0
0	1	1	0	1	0	x	x	0	0
1	0	0	1	1	x	0	1	x	0
1	0	1	0	1	x	1	1	x	0
1	1	0	0	0	x	1	x	1	0
1	1	1	0	1	x	1	x	0	1

$J_1 = X' Q_0$

$K_1 = X + Q_0$

$J_0 = X + Q_1$

$K_0 = X'$

$Z = Q_1 Q_0 X$

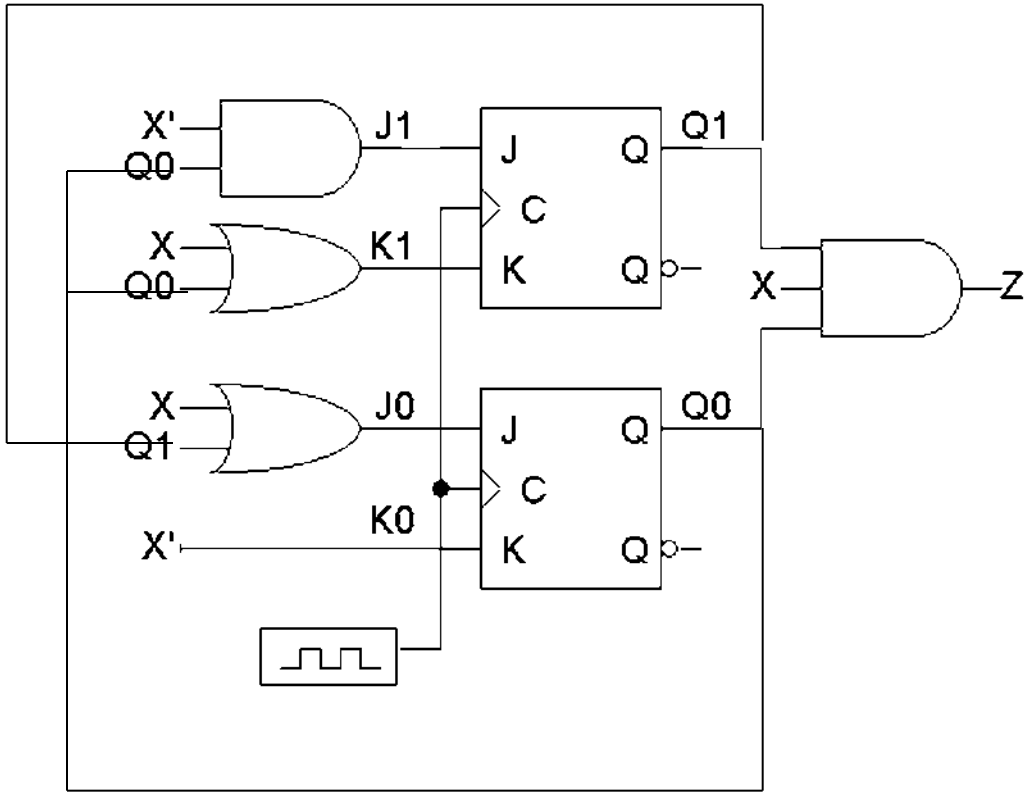
Step 5: Build the circuit

- Lastly, we use these simplified equations to build the completed circuit.

$J_1 = X' Q_0$
$K_1 = X + Q_0$

$J_0 = X + Q_1$
$K_0 = X'$

$Z = Q_1 Q_0 X$



Building the same circuit with D flip-flops

- What if you want to build the circuit using D flip-flops instead?
- We already have the state table and state assignments, so we can just start from Step 3, finding the flip-flop input values.
- D flip-flops have only one input, so our table only needs two columns for D_1 and D_0.

Present State		Input	Next State		Flip-flop inputs		Output
Q_1	Q_0	X	Q_1	Q_0	D_1	D_0	Z
0	0	0	0	0			0
0	0	1	0	1			0
0	1	0	1	0			0
0	1	1	0	1			0
1	0	0	1	1			0
1	0	1	0	1			0
1	1	0	0	0			0
1	1	1	0	1			1

D flip-flop input values (Step 3)

- The D excitation table is pretty boring; set the D input to whatever the next state should be.
- You don't even need to show separate columns for D_1 and D_0; you can just use the Next State columns.

Q(t)	Q(t+1)	D	Operation
0	0	0	Reset
0	1	1	Set
1	0	0	Reset
1	1	1	Set

Present State		Input	Next State		Flip flop inputs		Output
Q_1	Q_0	X	Q_1	Q_0	D_1	D_0	Z
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	1	0	1	0	0
0	1	1	0	1	0	1	0
1	0	0	1	1	1	1	0
1	0	1	0	1	0	1	0
1	1	0	0	0	0	0	0
1	1	1	0	1	0	1	1

Finding Equations (Step 4)

- You can do Karnaugh-maps again, to find:

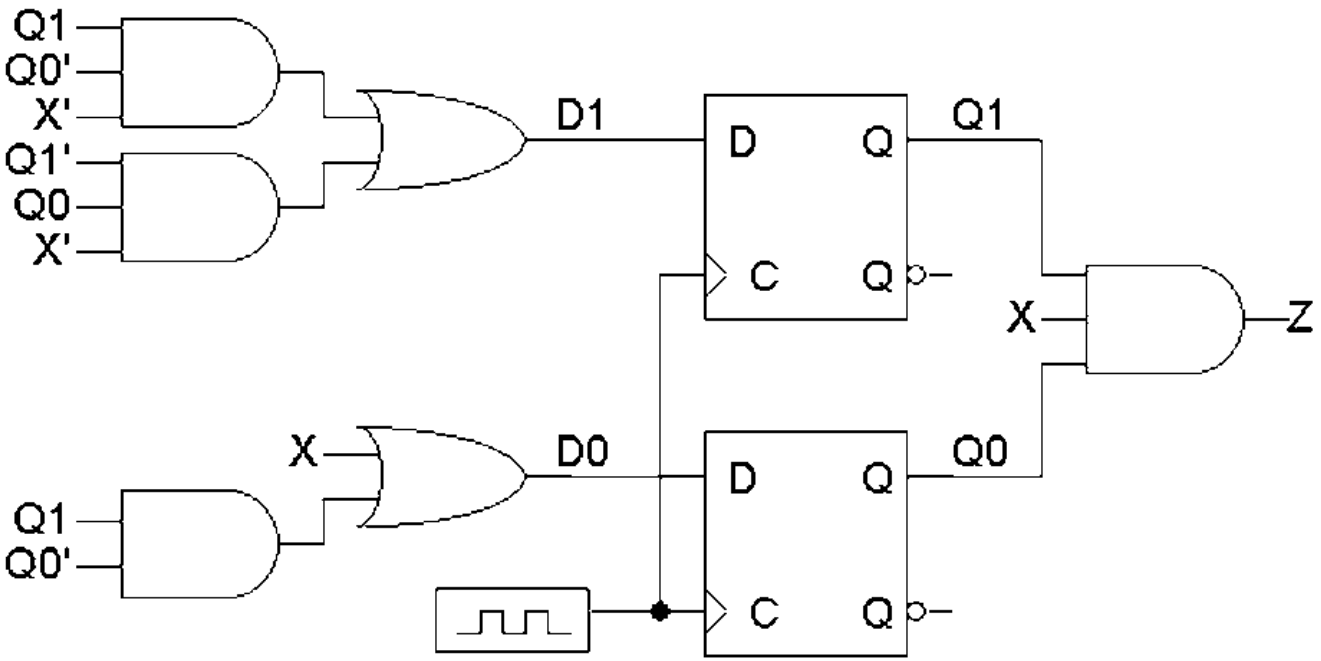
$$D_1 = Q_1\, Q_0'\, X' + Q_1'\, Q_0\, X'$$

$$D_0 = X + Q_1\, Q_0'$$

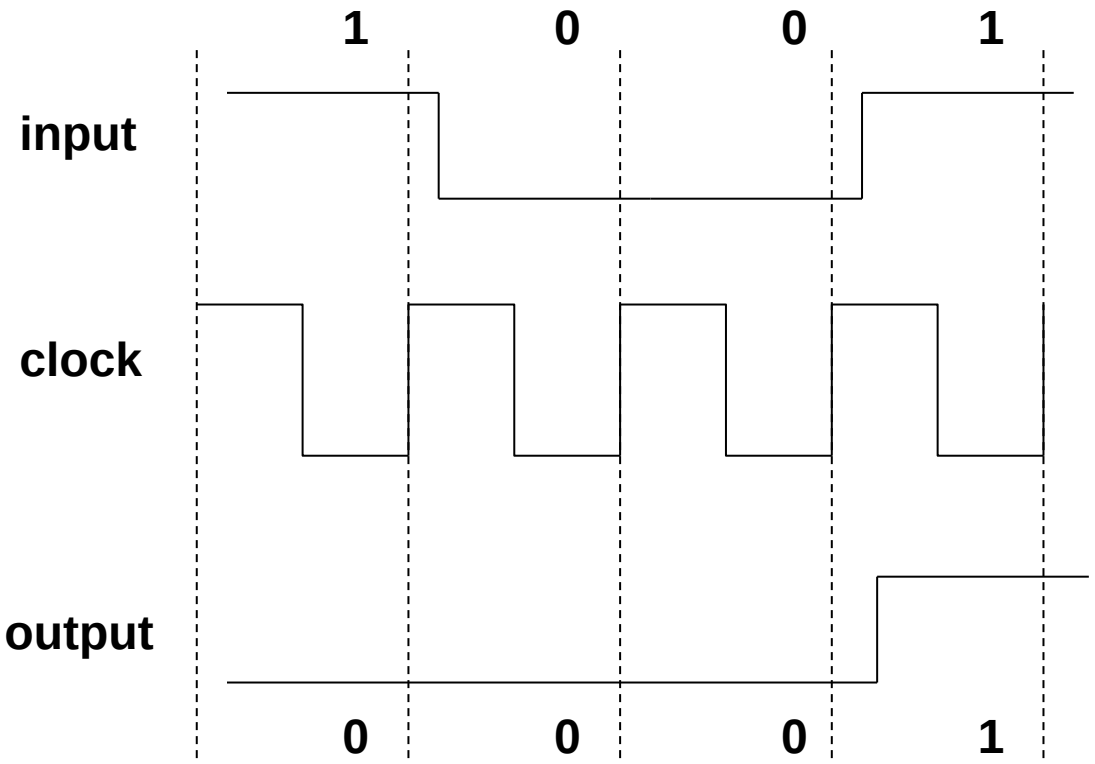
$$Z = Q_1\, Q_0\, X$$

Present State Q_1	Present State Q_0	Input X	Next State Q_1	Next State Q_0	Flip flop inputs D_1	Flip flop inputs D_0	Output Z
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	1	0	1	0	0
0	1	1	0	1	0	1	0
1	0	0	1	1	1	1	0
1	0	1	0	1	0	1	0
1	1	0	0	0	0	0	0
1	1	1	0	1	0	1	1

Building the Circuit (Step 5)

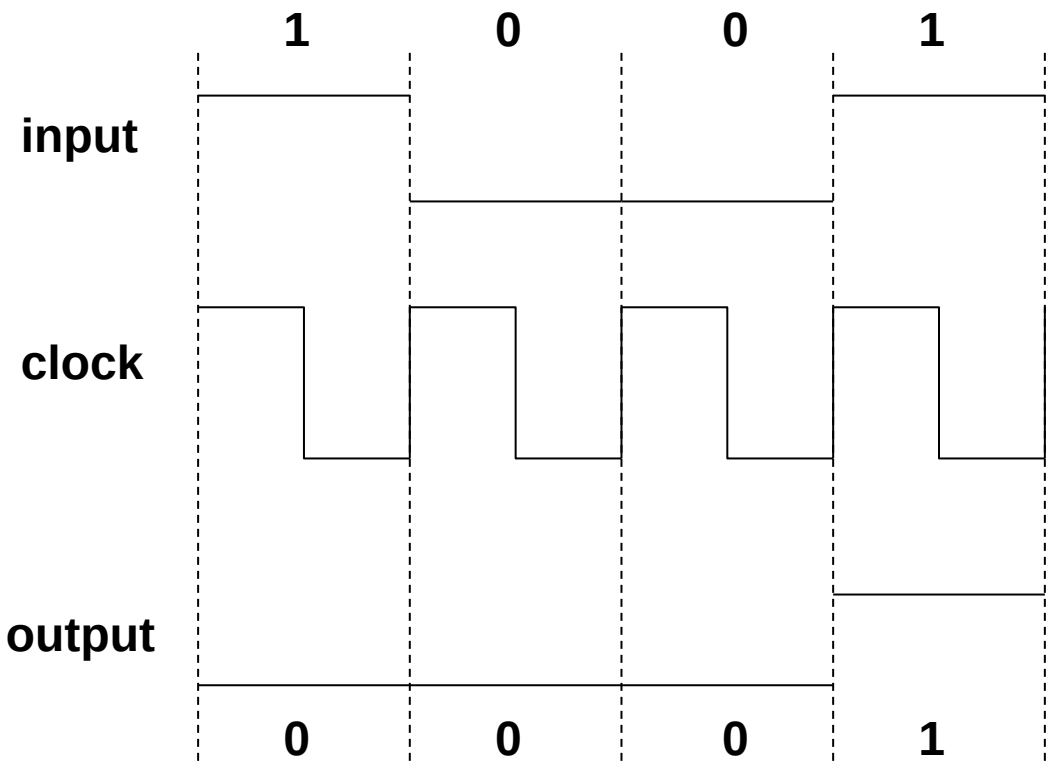


Waveform Analysis

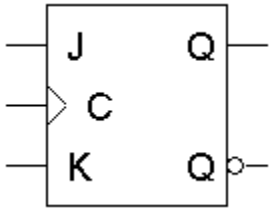


- Asynchronous outputs possible for asynchronous inputs…

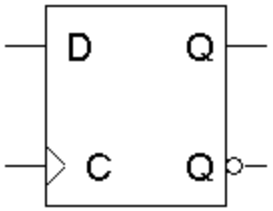
Waveform Analysis



Flip-flop Comparison



JK flip-flops are good because there are many don't care values in the flip-flop inputs, which can lead to a simpler circuit.



D flip-flops have the advantage that you don't have to set up flip-flop inputs at all, since Q(t+1) = D. However, the D input equations are usually more complex than JK input equations

In practice, D flip-flops are used more often.

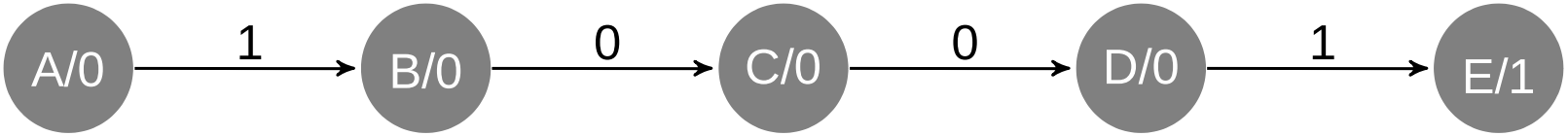
- There is only one input for each flip-flop, not two.
- There are no excitation tables to worry about.
- D flip-flops can be implemented with slightly less hardware than JK flip-flops.

Moore machine

- Here we show how the same sequence detector could be implemented as a Moore machine
- In this case the output can only depend on the state, *not on the inputs*
- Therefore we show the outputs inside the state circles on the state diagram, not on the edges joining the states…
- This leads to one extra state required for the Moore machine compared to the Mealy in this case.
- Moore machines often require extra states, but they have the advantage of always generating a synchronous output.

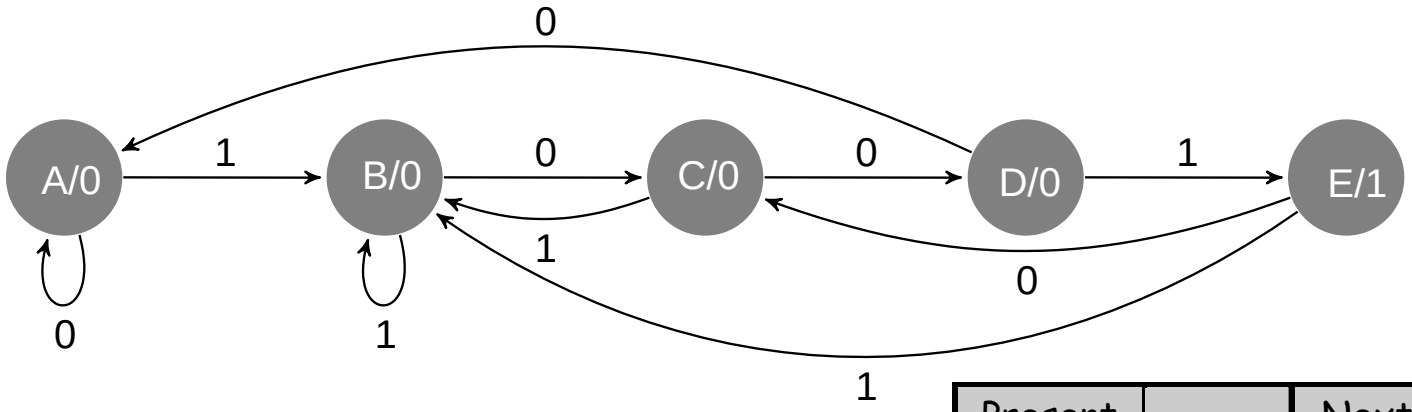
A basic state diagram (Moore)

- Repeating the state diagram for the sequence detector, this time using the Moore machine approach, we require one extra state
- This is because we need a state that corresponds to the situation where the full sequence has been detected (1001)
- In the Mealy approach, when we had detected 100, we then returned to an earlier state, with the *output depending on the input* – outputting a ‘0’ if we got a ‘0’, and a ‘1’ if we got a one ‘1’



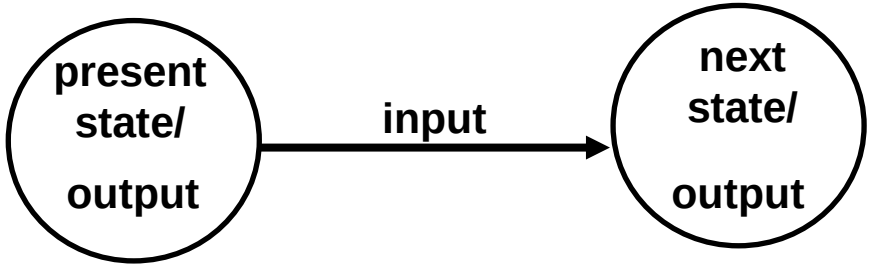
State	Meaning
A	None of the desired pattern (1001) has been input yet.
B	We’ve seen the first bit (1) of the desired pattern.
C	We’ve seen the first two bits (10) of the desired pattern.
D	We’ve seen the first three bits (100) of the desired pattern.
E	We’ve seen all four bits (1001) of the desired pattern.

Finally, making the state table



- The final Moore state diagram is shown above
- We need two arrows from each state
- The rest of the process then proceeds as per the Mealy machine

Remember how the state diagram arrows correspond to rows of the state table for Moore machine:



Present State	Input	Next State	Output
A	0	A	0
A	1	B	
B	0	C	0
B	1	B	
C	0	D	0
C	1	B	
D	0	A	0
D	1	E	
E	0	C	1
E	1	B	