Digital Electronics – Sequential Logic

Dr. I. J. Wassell

Sequential Logic

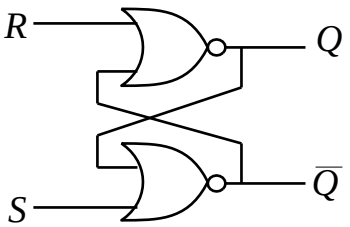
- The logic circuits discussed previously are known as *combinational*, in that the output depends only on the condition of the latest inputs
- However, we will now introduce a type of logic where the output depends not only on the latest inputs, but also on the condition of earlier inputs. These circuits are known as *sequential*, and implicitly they contain *memory* elements

Memory Elements

- A memory stores data – usually one bit per element
- A snapshot of the memory is called the *state*
- A one bit memory is often called a *bistable*, i.e., it has 2 stable internal states
- *Flip-flops* and *latches* are particular implementations of bistables

RS Latch

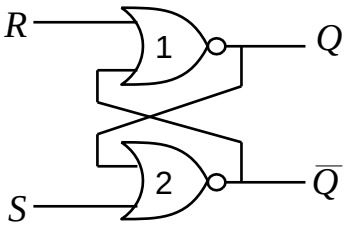
- An RS latch is a memory element with 2 inputs: Reset (R) and Set (S) and 2 outputs: Q and $\overline{Q}$.



S	R	Q'	$\overline{Q}'$	comment
0	0	Q	$\overline{Q}$	hold
0	1	0	1	reset
1	0	1	0	set
1	1	0	0	illegal

Where Q' is the next state and Q is the current state

RS Latch - Operation

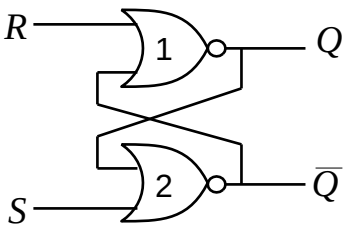


NOR truth table

a	b	y	
0	0	1	b complemented
0	1	0	
1	0	0	always 0
1	1	0	

- R = 1 and S = 0
  - Gate 1 output in 'always 0' condition, $Q = 0$
  - Gate 2 in 'complement' condition, so $\overline{Q} = 1$
- This is the (R)eset condition

RS Latch - Operation

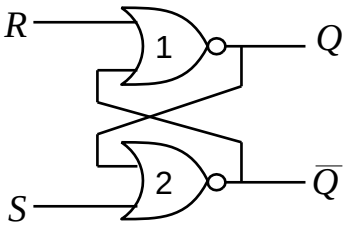


NOR truth table

a	b	y	
0	0	1	b complemented
0	1	0	
1	0	0	always 0
1	1	0	

- S = 0 and R to 0
  - Gate 2 remains in 'complement' condition, $\overline{Q} = 1$
  - Gate 1 into 'complement' condition, $Q = 0$
- This is the hold condition

RS Latch - Operation

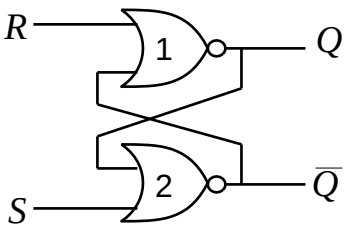


NOR truth table

a	b	y	
0	0	1	b complemented
0	1	0	
1	0	0	always 0
1	1	0	

- S = 1 and R = 0
  - Gate 1 into 'complement' condition, $Q = 1$
  - Gate 2 in 'always 0' condition, $\overline{Q} = 0$
- This is the (S)et condition

RS Latch - Operation



NOR truth table

a	b	y	
0	0	1	b complemented
0	1	0	
1	0	0	always 0
1	1	0	

- S = 1 and R = 1
  - Gate 1 in 'always 0' condition, $Q = 0$
  - Gate 2 in 'always 0' condition, $\overline{Q} = 0$
- This is the illegal condition

RS Latch – State Transition Table

- A *state transition table* is an alternative way of viewing its operation

Q	S	R	Q'	comment
0	0	0	0	hold
0	0	1	0	reset
0	1	0	1	set
0	1	1	0	illegal
1	0	0	1	hold
1	0	1	0	reset
1	1	0	1	set
1	1	1	0	illegal

- A state transition table can also be expressed in the form of a *state diagram*

RS Latch – State Diagram

- A *state diagram* in this case has 2 states, i.e., Q=0 and Q=1
- The state diagram shows the input conditions required to transition between states. In this case we see that there are 4 possible transitions
- We will consider them in turn

RS Latch – State Diagram

Q	S	R	Q'	comment
0	0	0	0	hold
0	0	1	0	reset
0	1	0	1	set
0	1	1	0	illegal
1	0	0	1	hold
1	0	1	0	reset
1	1	0	1	set
1	1	1	0	illegal

$Q = 0 \quad Q' = 0$

From the table we can see:

$\bar{S}.\bar{R} + \bar{S}.R + S.R =$

$\bar{S}.(\bar{R} + R) + S.R = \bar{S} + S.R =$

$(\bar{S} + S).(\bar{S} + R) = \bar{S} + R$

$Q = 1 \quad Q' = 1$

From the table we can see:

$\bar{S}.\bar{R} + S.\bar{R} = \bar{R}.(\bar{S} + S) =$

$\bar{R}$

RS Latch – State Diagram

Q	S	R	Q'	comment
0	0	0	0	hold
0	0	1	0	reset
0	1	0	1	set
0	1	1	0	illegal
1	0	0	1	hold
1	0	1	0	reset
1	1	0	1	set
1	1	1	0	illegal

$Q = 1 \quad Q' = 0$

From the table we can see:

$\bar{S}.R + S.R =$

$R.(\bar{S} + S) = R$

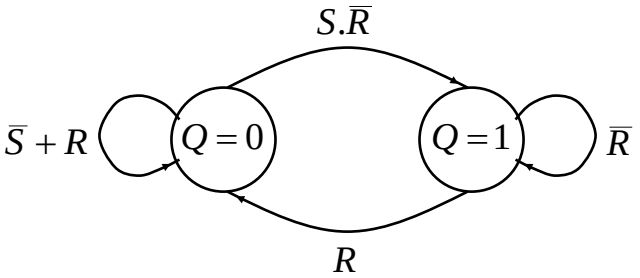
$Q = 0 \quad Q' = 1$

From the table we can see:

$S.\bar{R}$

RS Latch – State Diagram

- Which gives the following state diagram:



- A similar diagram can be constructed for the $\overline{Q}$ output
- We will see later that state diagrams are a useful tool for designing sequential systems

Clocks and Synchronous Circuits

- For the RS latch we have just described, we can see that the output state changes occur directly in response to changes in the inputs. This is called *asynchronous* operation
- However, virtually all sequential circuits currently employ the notion of *synchronous* operation, that is, the output of a sequential circuit is constrained to change only at a time specified by a global *enabling* signal. This signal is generally known as the system *clock*

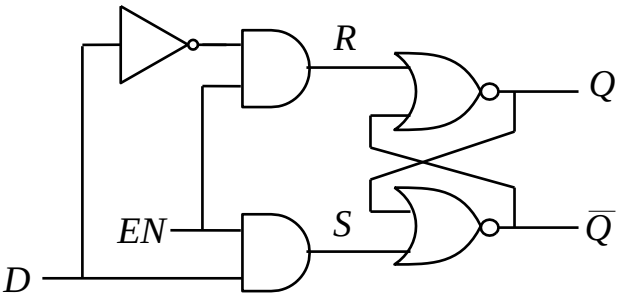
Clocks and Synchronous Circuits

- The Clock: What is it and what is it for?
  - Typically it is a square wave signal at a particular frequency
  - It imposes order on the state changes
  - Allows lots of states to appear to update simultaneously
- How can we modify an asynchronous circuit to act synchronously, i.e., in synchronism with a clock signal?

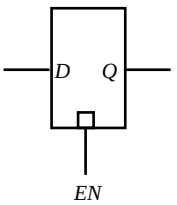
Transparent D Latch

- We now modify the RS Latch such that its output state is only permitted to change when a valid enable signal (which could be the system clock) is present
- This is achieved by introducing a couple of AND gates in cascade with the R and S inputs that are controlled by an additional input known as the *enable* (EN) input.

Transparent D Latch



Symbol

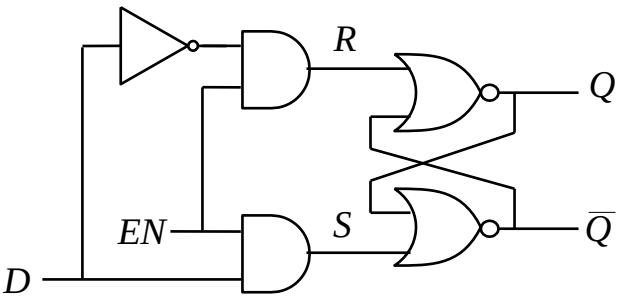


- See from the AND truth table:
  - if one of the inputs, say a is 0, the output is always 0
  - Output follows b input if a is 1
- The complement function ensures that R and S can never be 1 at the same time, i.e., illegal avoided

AND truth table

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

Transparent D Latch



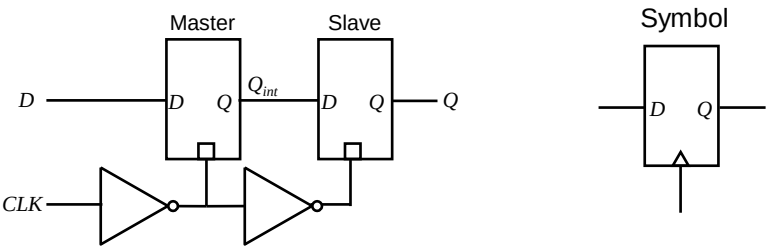
D	EN	Q'	$\overline{Q}'$	comment
X	0	Q	$\overline{Q}$	RS hold
0	1	0	1	RS reset
1	1	1	0	RS set

- See Q follows D input provided EN=1. If EN=0, Q maintains previous state

Master-Slave Flip-Flops

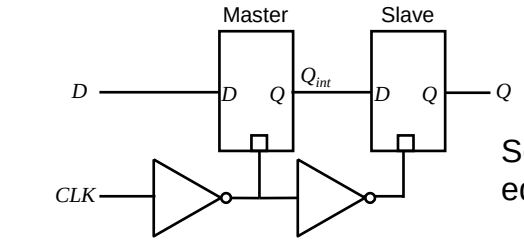
- The transparent D latch is so called '*level*' triggered. We can see it exhibits transparent behaviour if *EN*=1. It is often more simple to design sequential circuits if the outputs change only on the either rising (positive going) or falling (negative going) '*edges*' of the clock (i.e., enable) signal
- We can achieve this kind of operation by combining 2 transparent D latches in a so called *Master-Slave* configuration

Master-Slave D Flip-Flop

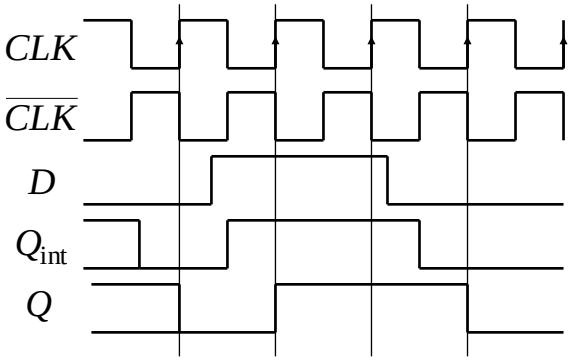


- To see how this works, we will use a timing diagram
- Note that both latch inputs are effectively connected to the clock signal (admittedly one is a complement of the other)

Master-Slave D Flip-Flop



See Q changes on rising edge of CLK



Note propagation delays have been neglected in the timing diagram

D Flip-Flops

- The Master-Slave configuration has now been superseded by new F-F circuits which are easier to implement and have better performance
- When designing synchronous circuits it is best to use truly edge triggered F-F devices
- We will not consider the design of such F-Fs on this course

Other Types of Flip-Flops

- Historically, other types of Flip-Flops have been important, e.g., J-K Flip-Flops and T-Flip-Flops
- However, J-K FFs are a lot more complex to build than D-types and so have fallen out of favour in modern designs, e.g., for field programmable gate arrays (FPGAs) and VLSI chips

Other Types of Flip-Flops

- Consequently we will only consider synchronous circuit design using D-type FFs
- However for completeness we will briefly look at the truth table for J-K and T type FFs

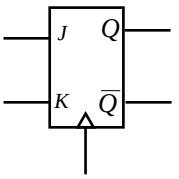
J-K Flip-Flop

- The J-K FF is similar in function to a clocked RS FF, but with the illegal state replaced with a new ‘toggle’ state

J K	Q' $\overline{Q}'$	comment
0 0	Q $\overline{Q}$	hold
0 1	0 1	reset
1 0	1 0	set
1 1	$\overline{Q}$ Q	toggle

Where Q' is the next state
and Q is the current state

Symbol



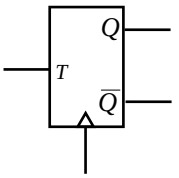
T Flip-Flop

- This is essentially a J-K FF with its J and K inputs connected together and renamed as the T input

T	Q' $\overline{Q}'$	comment
0	Q $\overline{Q}$	hold
1	$\overline{Q}$ Q	toggle

Where Q' is the next state
and Q is the current state

Symbol



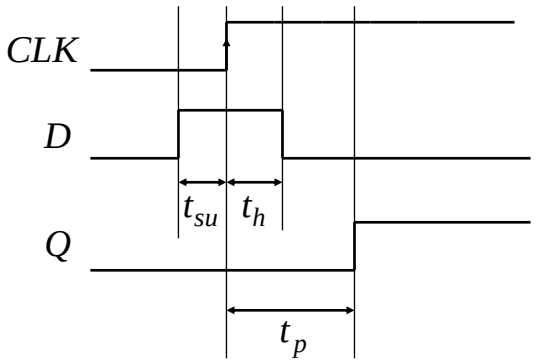
Asynchronous Inputs

- It is common for the FF types we have mentioned to also have additional so called ‘asynchronous’ inputs
- They are called asynchronous since they take effect independently of any clock or enable inputs
- Reset/Clear – force Q to 0
- Preset/Set – force Q to 1
- Often used to force a synchronous circuit into a known state, say at start-up.

Timing

- Various timings must be satisfied if a FF is to operate properly:
  - *Setup time*: Is the minimum duration that the data must be stable at the input before the clock edge
  - *Hold time*: Is the minimum duration that the data must remain stable on the FF input after the clock edge

Timing



t_{su} Set-up time

t_h Hold time

t_p Propagation delay

Applications of Flip-Flops

- Counters
  - A clocked sequential circuit that goes through a predetermined sequence of states
  - A commonly used counter is an n-bit binary counter. This has n FFs and 2^n states which are passed through in the order 0, 1, 2, ….2^n-1, 0, 1, .
  - Uses include:
    - Counting
    - Producing delays of a particular duration
    - Sequencers for control logic in a processor
    - Divide by m counter (a divider), as used in a digital watch

Applications of Flip-Flops

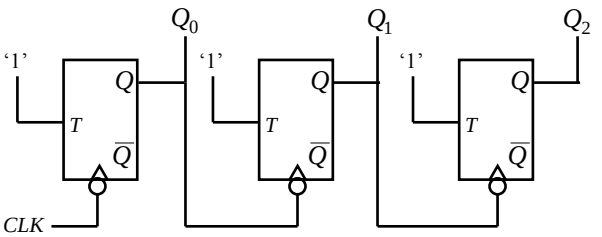
- Memories, e.g.,
  - Shift register
    - Parallel loading shift register : can be used for parallel to serial conversion in serial data communication
    - Serial in, parallel out shift register: can be used for serial to parallel conversion in a serial data communication system.

Counters

- In most books you will see 2 basic types of counters, namely *ripple* counters and *synchronous* counters
- In this course we are concerned with synchronous design principles. Ripple counters do not follow these principles and should generally be avoided if at all possible. We will now look at the problems with ripple counters

Ripple Counters

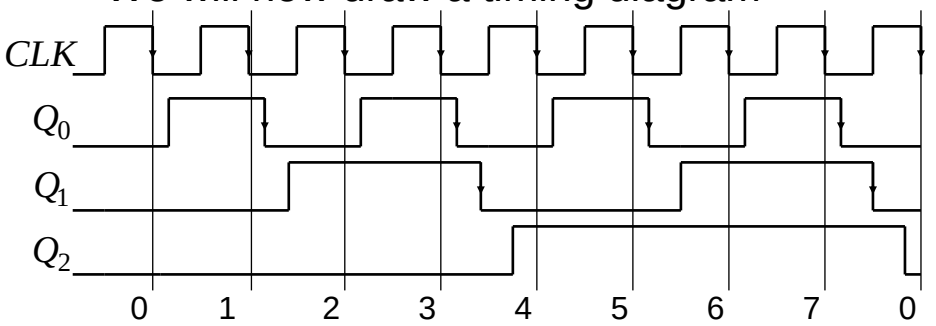
- A ripple counter can be made be cascading together negative edge triggered T-type FFs operating in 'toggle' mode, i.e., T =1



- See that the FFs are not clocked using the same clock, i.e., this is **not** a synchronous design. This gives some problems….

Ripple Counters

- We will now draw a timing diagram



- Problems:

See outputs do not change at the same time, i.e., synchronously. So hard to know when count output is actually valid.

Propagation delay builds up from stage to stage, limiting maximum clock speed before miscounting occurs.

Ripple Counters

- If you observe the frequency of the counter output signals you will note that each has half the frequency, i.e., double the repetition period of the previous one. This is why counters are often known as dividers
- Often we wish to have a count which is not a power of 2, e.g., for a BCD counter (0 to 9).To do this:
  - use FFs having a Reset/Clear input
  - Use an AND gate to detect the count of 10 and use its output to Reset the FFs

Synchronous Counters

- Owing to the problems identified with ripple counters, they should not usually be used to implement counter functions
- It is recommended that *synchronous* counter designs be used
- In a synchronous design
  - all the FF clock inputs are directly connected to the clock signal and so all FF outputs change at the same time, i.e., *synchronously*
  - more complex combinational logic is now needed to generate the appropriate FF input signals (which will be different depending upon the type of FF chosen)

Synchronous Counters

- We will now investigate the design of synchronous counters
- We will consider the use of D-type FFs only, although the technique can be extended to cover other FF types.
- As an example, we will consider a 0 to 7 up-counter

Synchronous Counters

- To assist in the design of the counter we will make use of a modified *state transition table.* This table has additional columns that define the required FF inputs (or e*xcitation* as it is known)
  - Note we have used a state transition table previously when determining the state diagram for an RS latch
- We will also make use of the so called '*excitation table*' for a D-type FF
- First however, we will investigate the so called *characteristic table* and *characteristic equation* for a D-type FF

Characteristic Table

- In general, a characteristic table for a FF gives the next state of the output, i.e., Q' in terms of its current state Q and current inputs

Q	D	Q'
0	0	0
0	1	1
1	0	0
1	1	1

Which gives the characteristic equation,

$$Q' = D$$

i.e., the next output state is equal to the current input value

Since Q' is independent of Q the characteristic table can be rewritten as

D	Q'
0	0
1	1

Excitation Table

- The characteristic table can be modified to give the excitation table. This table tells us the required FF input value required to achieve a particular next state from a given current state

Q	Q'	D
0	0	0
0	1	1
1	0	0
1	1	1

As with the characteristic table it can be seen that Q', does not depend upon, Q , however this is not generally true for other FF types, in which case, the excitation table is more useful. Clearly for a D-FF, $D = Q'$

Characteristic and Excitation Tables

- Characteristic and excitation tables can be determined for other FF types.
- These should be used in the design process if D-type FFs are not used
- For example, for a J-K FF the following tables are appropriate:

Characteristic and Excitation Tables

J	K	Q'
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

Truth table

Q	Q'	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Excitation table

- We will now determine the modified state transition table for the example 0 to 7 up-counter

Modified State Transition Table

- In addition to columns representing the current and desired next states (as in a conventional state transition table), the modified table has additional columns representing the required FF inputs to achieve the next desired FF states

Modified State Transition Table

- For a 0 to 7 counter, 3 D-type FFs are needed

Current state	Next state	FF inputs
$Q_2 Q_1 Q_0$	$Q_2' Q_1' Q_0'$	$D_2 D_1 D_0$
0 0 0	0 0 1	0 0 1
0 0 1	0 1 0	0 1 0
0 1 0	0 1 1	0 1 1
0 1 1	1 0 0	1 0 0
1 0 0	1 0 1	1 0 1
1 0 1	1 1 0	1 1 0
1 1 0	1 1 1	1 1 1
1 1 1	0 0 0	0 0 0

The procedure is to:

- Write down the desired count sequence in the current state columns
- Write down the required next states in the next state columns
- Fill in the FF inputs required to give the defined next state

Note: Since $Q' = D$ (or $D = Q'$) for a D-FF, the required FF inputs are identical to the Next state

Synchronous Counter Example

- If using J-K FFs for example, we need J and K input columns for each FF
- Also note that if we are using D-type FFs, it is not necessary to explicitly write out the FF input columns, since we know they are identical to those for the next state
- To complete the design we now have to determine appropriate combinational logic circuits which will generate the required FF inputs from the current states
- We can do this from inspection, using Boolean algebra or using K-maps.

Synchronous Counter Example

Current state	Next state	FF inputs
$Q_2 Q_1 Q_0$	$Q_2' Q_1' Q_0'$	$D_2 D_1 D_0$
0 0 0	0 0 1	0 0 1
0 0 1	0 1 0	0 1 0
0 1 0	0 1 1	0 1 1
0 1 1	1 0 0	1 0 0
1 0 0	1 0 1	1 0 1
1 0 1	1 1 0	1 1 0
1 1 0	1 1 1	1 1 1
1 1 1	0 0 0	0 0 0

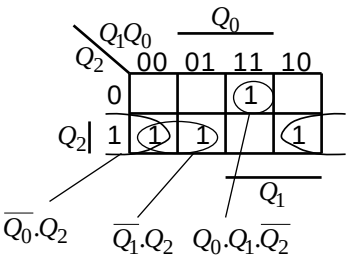
By inspection,

$D_0 = \overline{Q_0}$

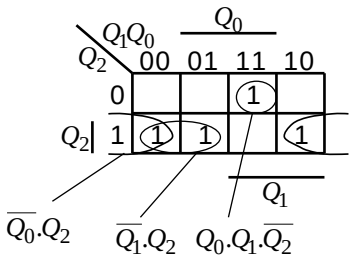
Note: FF_0 is toggling

Also, $D_1 = Q_0 \oplus Q_1$

Use a K-map for D_2,



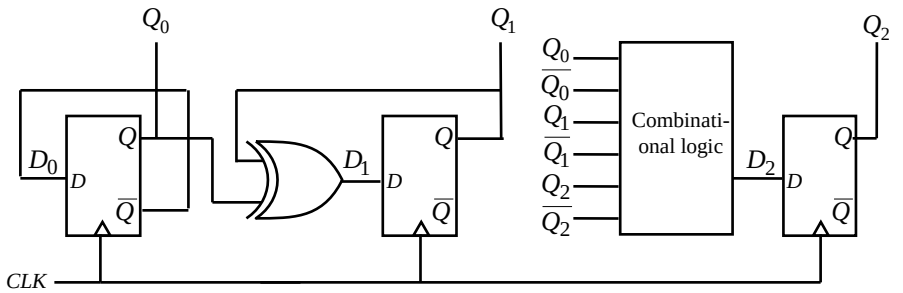
Synchronous Counter Example



So,

$$D_2 = \overline{Q_0}.Q_2 + \overline{Q_1}.Q_2 + Q_0.Q_1.\overline{Q_2}$$

$$D_2 = Q_2.(\overline{Q_0}. + \overline{Q_1}) + Q_0.Q_1.\overline{Q_2}$$

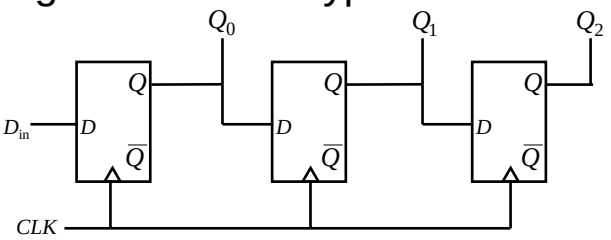


Synchronous Counter

- A similar procedure can be used to design counters having an arbitrary count sequence
  - Write down the state transition table
  - Determine the FF excitation (easy for D-types)
  - Determine the combinational logic necessary to generate the required FF excitation from the current states – **Note:** remember to take into account any unused counts since these can be used as don't care states when determining the combinational logic circuits

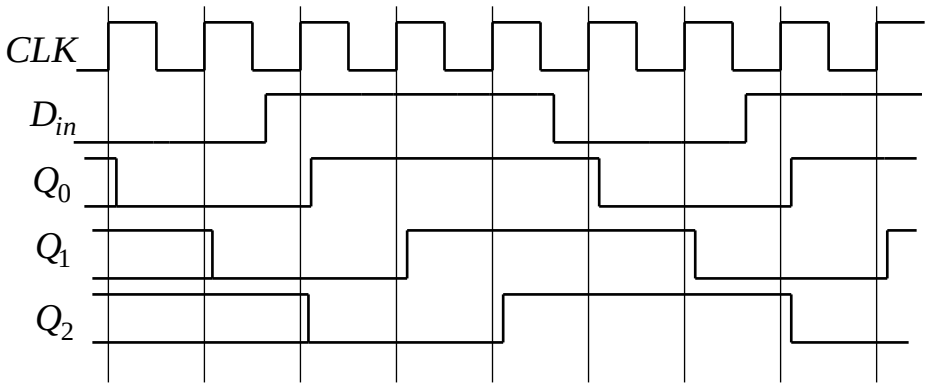
Shift Register

- A shift register can be implemented using a chain of D-type FFs



- Has a serial input, D_{in} and parallel output Q_0, Q_1 and Q_2.

Shift Register

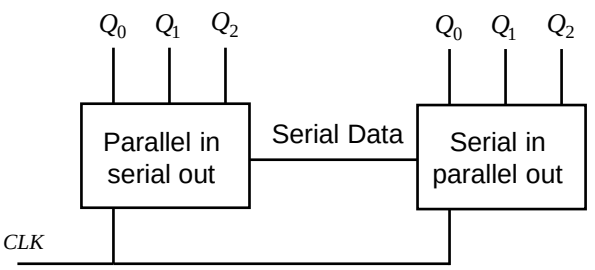


- See data moves one position to the right on application of each clock edge

Shift Register

- Preset and Clear inputs on the FFs can be utilised to provide a parallel data input feature
- Data can then be clocked out through Q_2 in a serial fashion, i.e., we now have a parallel in, serial out arrangement
- This along with the previous serial in, parallel out shift register arrangement can be used as the basis for a serial data link

Serial Data Link



- One data bit at a time is sent across the serial data link
- See less wires are required than for a parallel data link

Synchronous State Machines

Synchronous State Machines

- We have seen how we can use FFs (D-types in particular) to design synchronous counters
- We will now investigate how these principles can be extended to the design of synchronous state machines (of which counters are a subset)
- We will begin with some definitions and then introduce two popular types of machines

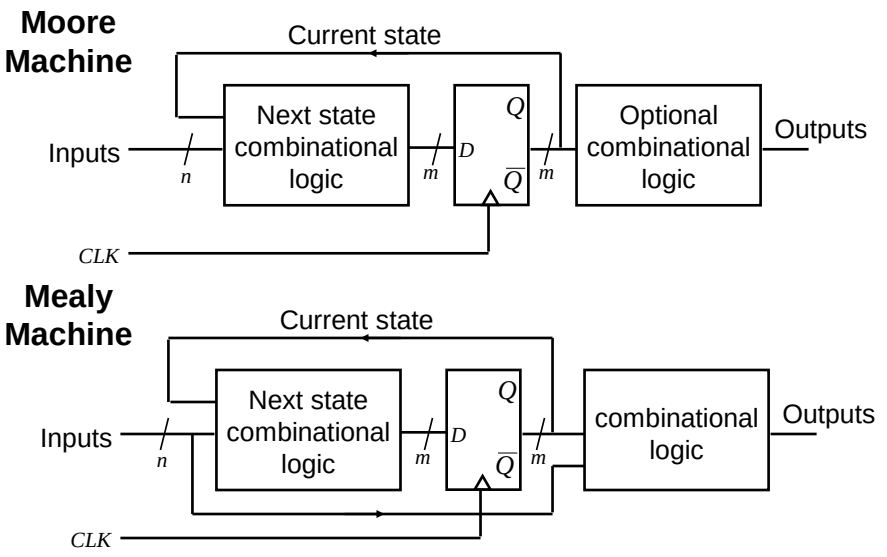
Definitions

- **Finite State Machine (FSM)** – a deterministic machine (circuit) that produces outputs which depend on its internal state and external inputs
- **States** – the set of internal memorised values, shown as circles on the state diagram
- **Inputs –** External stimuli, labelled as arcs on the state diagram
- **Outputs –** Results from the FSM

Types of State Machines

- Two types of state machines are in general use, namely *Moore* machines and *Mealy* machines
- We will see that the state diagrams (and associated state tables) corresponding with the 2 types of machine are slightly different

Machine Schematics

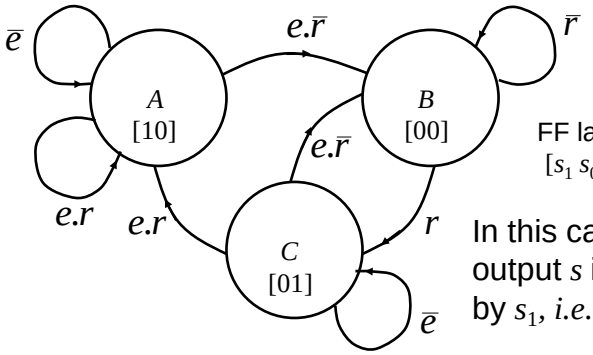


Moore vs. Mealy Machines

- Outputs from Mealy Machines depend upon the timing of the inputs
- Outputs from Moore machines come directly from clocked FFs so:
  - They have guaranteed timing characteristics
  - They are glitch free
- Any Mealy machine can be converted to a Moore machine and vice versa, though their timing properties will be different

Moore Machine State Diagram

- Example FSM has 3 states (A, B and C), inputs e and r, and output s

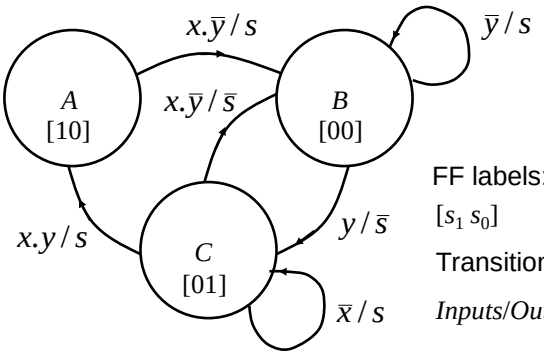


In this case the output s is given by s_1, *i.e.*, $s=s_1$

- See **inputs only** appear on transitions between states, i.e., next state is given by current state and current inputs
- Outputs determined from current state via combinational logic (if required)

Mealy Machine State Diagram

- Example FSM has 3 states (A, B and C), inputs x and y, and output s

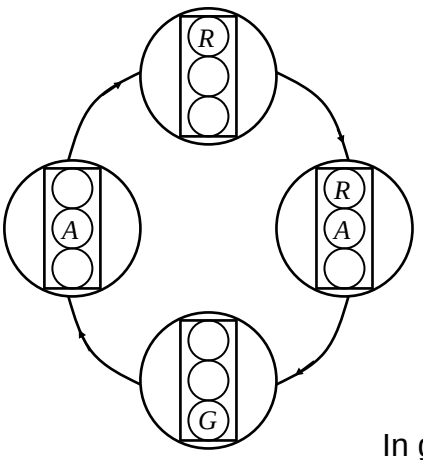


- Inputs **and outputs** appear on transitions between states, i.e., next state is given by current state and current inputs
- Output determined from current state and inputs via combinational logic

Moore Machine - Example

- We will design a Moore Machine to implement a traffic light controller
- In order to visualise the problem it is often helpful to draw the state transition diagram
- This is used to generate the state transition table
- The state transition table is used to generate
  - The next state combinational logic
  - The output combinational logic (if required)

Example – Traffic Light Controller



See we have 4 states

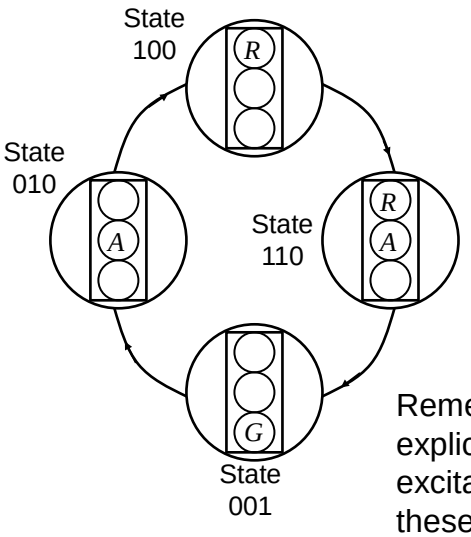
So in theory we could use a minimum of 2 FFs

However, by using 3 FFs we will see that we do not need to use any output combinational logic

So, we will only use 4 of the 8 possible states

In general, state assignment is a difficult problem and the optimum choice is not always obvious

Example – Traffic Light Controller



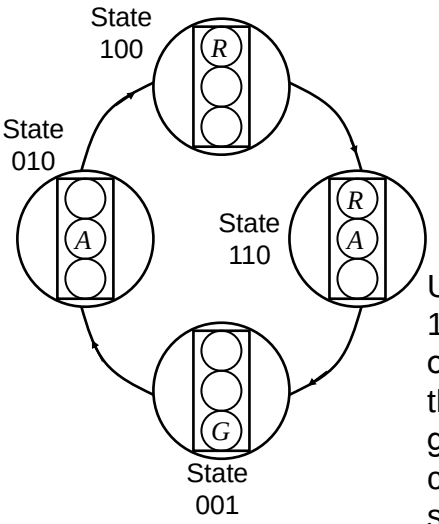
By using 3 FFs (we will use D-types), we can assign one to each of the required outputs (R, A, G), eliminating the need for output logic

We now need to write down the state transition table

We will label the FF outputs R, A and G

Remember we do not need to explicitly include columns for FF excitation since if we use D-types these are identical to the next state

Example – Traffic Light Controller



Current state			Next state		
R	A	G	R'	A'	G'
1	0	0	1	1	0
1	1	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0

Unused states, 000, 011, 101 and 111. Since these states will never occur, we don't care what output the next state combinational logic gives for these inputs. These don't care conditions can be used to simplify the required next state combinational logic

Example – Traffic Light Controller

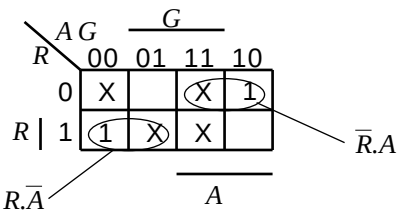
Current state			Next state		
R	A	G	R'	A'	G'
1	0	0	1	1	0
1	1	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0

Unused states, 000, 011, 101 and 111.

We now need to determine the next state combinational logic

For the R FF, we need to determine D_R

To do this we will use a K-map



$$D_R = R.\overline{A} + \overline{R}.A = R \oplus A$$

Example – Traffic Light Controller

Current state			Next state		
R	A	G	R'	A'	G'
1	0	0	1	1	0
1	1	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0

Unused states, 000, 011, 101 and 111.

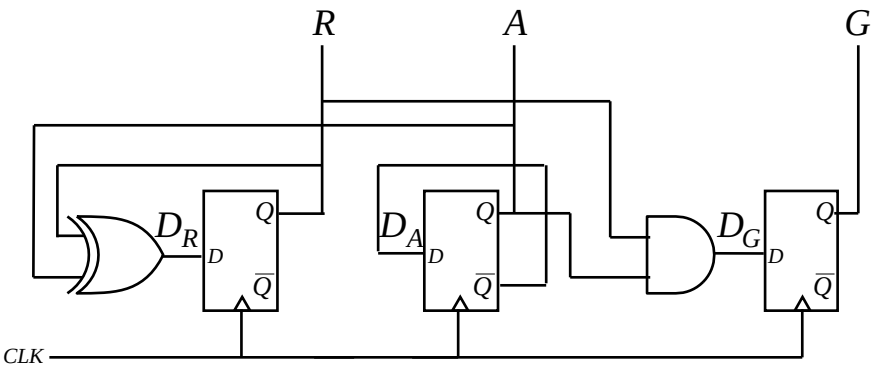
By inspection we can also see:

$$D_A = \overline{A}$$

and,

$$D_G = R.A$$

Example – Traffic Light Controller



FSM Problems

- Consider what could happen on power-up
- The state of the FFs could by chance be in one of the unused states
  - This could potentially cause the machine to become stuck in some unanticipated sequence of states which never goes back to a used state

FSM Problems

- What can be done?
  - Check to see if the FSM can eventually enter a known state from any of the unused states
  - If not, add additional logic to do this, i.e., include unused states in the state transition table along with a valid next state
  - Alternatively use asynchronous Clear and Preset FF inputs to set a known (used) state at power up

Example – Traffic Light Controller

- Does the example FSM self-start?
- Check what the next state logic outputs if we begin in any of the unused states
- Turns out:

Start state	Next state logic output
000	010
011	100
101	110
111	001

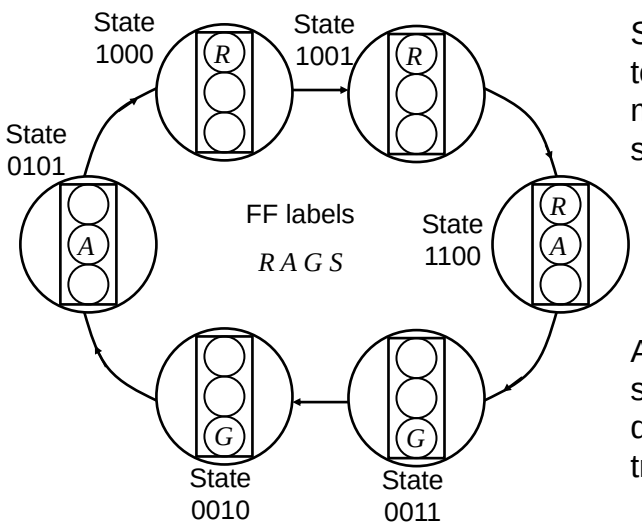
Which are all valid states

So it does self start

Example 2

- We extend Example 1 so that the traffic signals spend extra time for the R and G lights
- Essentially, we need 2 additional states, i.e., 6 in total.
- In theory, the 3 FF machine gives us the potential for sufficient states
- However, to make the machine combinational logic easier, it is more convenient to add another FF (labelled S), making 4 in total

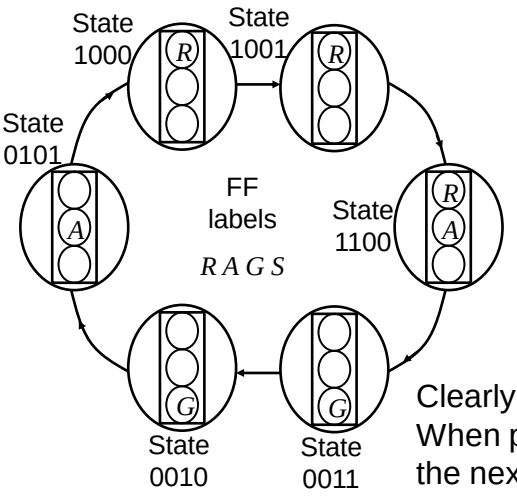
Example 2



See that new FF toggles which makes the next state logic easier

As before, the first step is to write down the state transition table

Example 2



Current state				Next state			
R	A	G	S	R'	A'	G'	S'
1	0	0	0	1	0	0	1
1	0	0	1	1	1	0	0
1	1	0	0	0	0	1	1
0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	1
0	1	0	1	1	0	0	0

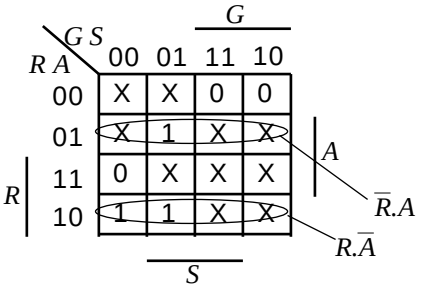
Clearly a lot of unused states. When plotting k-maps to determine the next state logic it is probably easier to plot 0s and 1s in the map and then mark the unused states

Example 2

Current state				Next state			
R	A	G	S	R'	A'	G'	S'
1	0	0	0	1	0	0	1
1	0	0	1	1	1	0	0
1	1	0	0	0	0	1	1
0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	1
0	1	0	1	1	0	0	0

We will now use k-maps to determine the next state combinational logic

For the R FF, we need to determine D_R



$$D_R = R.\overline{A} + \overline{R}.A = R \oplus A$$

Example 2

Current state				Next state			
R	A	G	S	R'	A'	G'	S'
1	0	0	0	1	0	0	1
1	0	0	1	1	1	0	0
1	1	0	0	0	0	1	1
0	0	1	1	0	0	1	0
0	0	1	0	0	1	0	1
0	1	0	1	1	0	0	0

We can plot k-maps for D_A and D_G to give:

$$D_A = R.S + G.\overline{S} \quad \text{or}$$

$$D_A = R.S + \overline{R}.\overline{S} = \overline{R \oplus S}$$

$$D_G = R.A + G.S \quad \text{or}$$

$$D_G = G.S + A.\overline{S}$$

By inspection we can also see:

$$D_S = \overline{S}$$

State Assignment

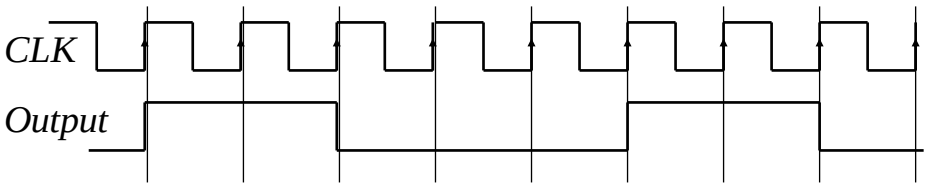
- As we have mentioned previously, state assignment is not necessarily obvious or straightforward
  - Depends what we are trying to optimise, e.g.,
    - Complexity (which also depends on the implementation technology, e.g., FPGA, 74 series logic chips).
      - FF implementation may take less chip area than you may think given their gate level representation
      - Wiring complexity can be as big an issue as gate complexity
    - Speed
  - Algorithms do exist for selecting the 'optimising' state assignment, but are not suitable for manual execution

State Assignment

- If we have m states, we need at least $\log_2 m$ FFs (or more informally, bits) to encode the states, e.g., for 8 states we need a min of 3 FFs
- We will now present an example giving various potential state assignments, some using more FFs than the minimum

Example Problem

- We wish to investigate some state assignment options to implement a divide by 5 counter which gives a 1 output for 2 clock edges and is 0 for 3 clock edges



Sequential State Assignment

- Here we simply assign the states in an increasing natural binary count
- As usual we need to write down the state transition table. In this case we need 5 states, i.e., a minimum of 3 FFs (or state bits). We will designate the 3 FF outputs as c, b, and a
- We can then determine the necessary next state logic and any output logic.

Sequential State Assignment

Current state			Next state		
c	b	a	c'	b'	a'
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	0	0	0

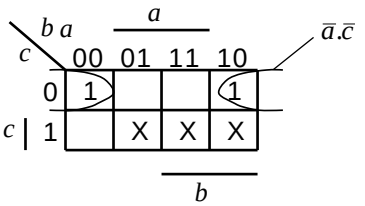
Unused states, 101, 110 and 111.

By inspection we can see:

The required output is from FF b

Plot k-maps to determine the next state logic:

For FF a:



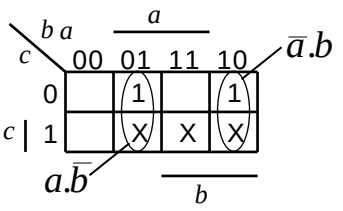
$$D_a = \bar{a}.\bar{c}$$

Sequential State Assignment

Current state			Next state		
c	b	a	c'	b'	a'
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	0	0	0

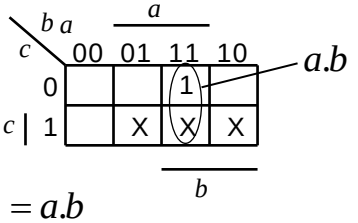
Unused states, 101, 110 and 111.

For FF b:



$$D_b = \bar{a}.b + a.\bar{b} = a \oplus b$$

For FF c:



$$D_c = a.b$$

Sliding State Assignment

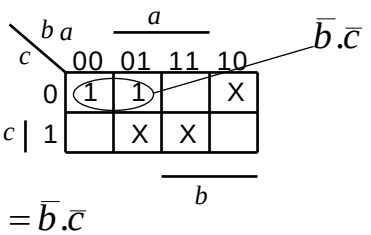
Current state			Next state		
c	b	a	c'	b'	a'
0	0	0	0	0	1
0	0	1	0	1	1
0	1	1	1	1	0
1	1	0	1	0	0
1	0	0	0	0	0

Unused states, 010, 101, and 111.

By inspection we can see that we can use any of the FF outputs as the wanted output

Plot k-maps to determine the next state logic:

For FF a:



$$D_a = \bar{b}.\bar{c}$$

Sliding State Assignment

Current state			Next state		
c	b	a	c'	b'	a'
0	0	0	0	0	1
0	0	1	0	1	1
0	1	1	1	1	0
1	1	0	1	0	0
1	0	0	0	0	0

Unused states, 010, 101, and 111.

By inspection we can see that:

For FF b:

$$D_b = a$$

For FF c:

$$D_c = b$$

Shift Register Assignment

- As the name implies, the FFs are connected together to form a shift register. In addition, the output from the final shift register in the chain is connected to the input of the first FF:
  - Consequently the data continuously cycles through the register

Shift Register Assignment

Current state					Next state				
e	d	c	b	a	e'	d'	c'	b'	a'
0	0	0	1	1	0	0	1	1	0
0	0	1	1	0	0	1	1	0	0
0	1	1	0	0	1	1	0	0	0
1	1	0	0	0	1	0	0	0	1
1	0	0	0	1	0	0	0	1	1

Unused states. Lots!

Because of the shift register configuration and also from the state table we can see that:

$$D_a = e$$
$$D_b = a$$
$$D_c = b$$
$$D_d = c$$
$$D_e = d$$

By inspection we can see that we can use any of the FF outputs as the wanted output

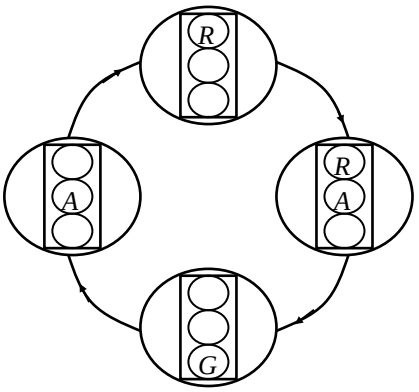
See needs 2 more FFs, but no logic and simple wiring

One Hot State Encoding

- This is a shift register design style where only one FF at a time holds a 1
- Consequently we have 1 FF per state, compared with $\log_2 m$ for sequential assignment
- However, can result in simple fast state machines
- Outputs are generated by ORing together appropriate FF outputs

One Hot - Example

- We will return to the traffic signal example, which recall has 4 states

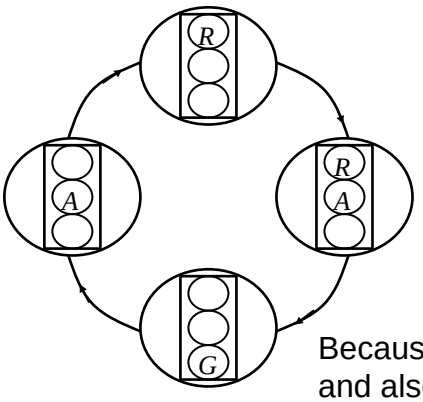


For 1 hot, we need 1 FF for each state, i.e., 4 in this case

The FFs are connected to form a shift register as in the previous shift register example, however in 1 hot, only 1 FF holds a 1 at any time

We can write down the state transition table as follows

One Hot - Example



Current state				Next state			
r	ra	g	a	r'	ra'	g'	a'
1	0	0	0	0	1	0	0
0	1	0	0	0	0	1	0
0	0	1	0	0	0	0	1
0	0	0	1	1	0	0	0

Unused states. Lots!

Because of the shift register configuration and also from the state table we can see that: $D_a = g \quad D_g = ra \quad D_{ra} = r \quad D_r = a$

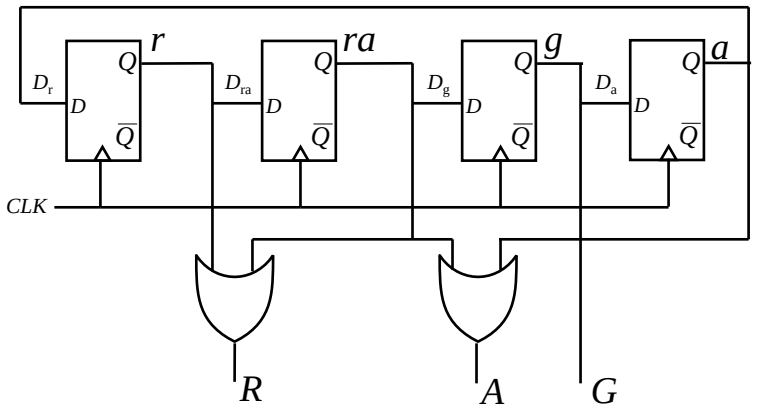
To generate the R, A and G outputs we do the following ORing:

$$R = r + ra \qquad A = ra + a \qquad G = g$$

One Hot - Example

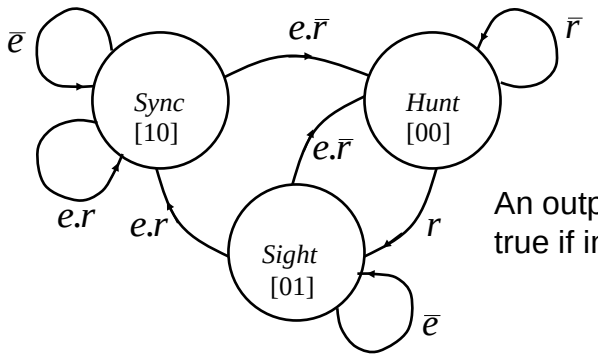
$D_a = g \quad D_g = ra \quad D_{ra} = r \quad D_r = a$

$R = r + ra \qquad A = ra + a \qquad G = g$



Tripos Example

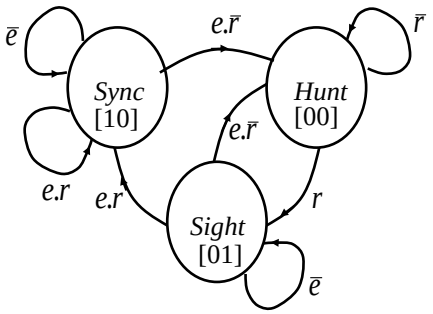
- The state diagram for a synchroniser is shown. It has 3 states and 2 inputs, namely e and r. The states are mapped using sequential assignment as shown.



FF labels
$[s_1\ s_0]$

An output, s should be true if in *Sync* state

Tripos Example



Unused state 11

From inspection, $s = s_1$

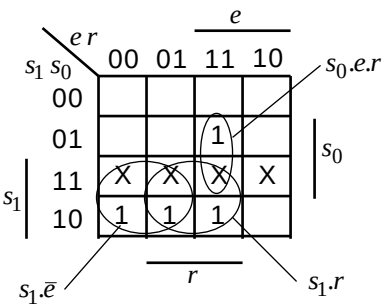
Current state		Input		Next state	
s_1	s_0	e	r	s_1'	s_0'
0	0	X	0	0	0
0	0	X	1	0	1
0	1	0	X	0	1
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	X	1	0
1	0	1	0	0	0
1	0	1	1	1	0
1	1	X	X	X	X

Tripos Example

Current state		Input		Next state	
s_1	s_0	e	r	s_1'	s_0'
0	0	X	0	0	0
0	0	X	1	0	1
0	1	0	X	0	1
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	X	1	0
1	0	1	0	0	0
1	0	1	1	1	0
1	1	X	X	X	X

Plot k-maps to determine the next state logic

For FF 1:



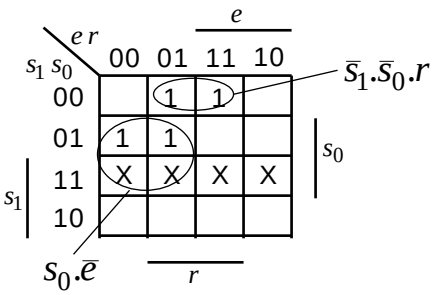
$$D_1 = s_1.\bar{e} + s_1.r + s_0.e.r$$

Tripos Example

Current state		Input		Next state	
s_1	s_0	e	r	s_1'	s_0'
0	0	X	0	0	0
0	0	X	1	0	1
0	1	0	X	0	1
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	X	1	0
1	0	1	0	0	0
1	0	1	1	1	0
1	1	X	X	X	X

Plot k-maps to determine the next state logic

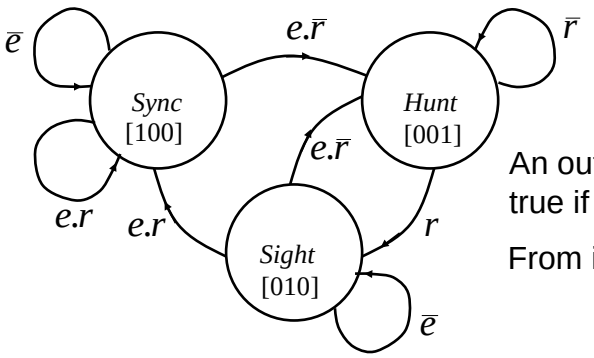
For FF 0:



$$D_0 = s_0.\bar{e} + \bar{s}_1.\bar{s}_0.r$$

Tripos Example

- We will now re-implement the synchroniser using a 1 hot approach
- In this case we will need 3 FFs

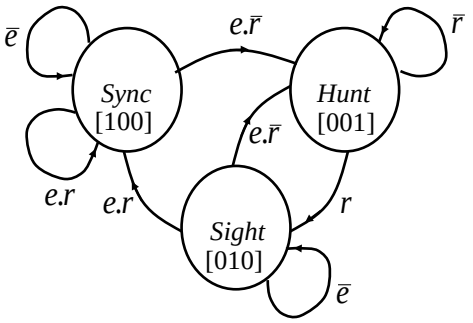


FF labels
$[s_2\ s_1\ s_0]$

An output, s should be true if in *Sync* state

From inspection, $s = s_2$

Tripos Example



Current state			Input		Next state		
s_2	s_1	s_0	e	r	s_2'	s_1'	s_0'
0	0	1	X	0	0	0	1
0	0	1	X	1	0	1	0
0	1	0	0	X	0	1	0
0	1	0	1	0	0	0	1
0	1	0	1	1	1	0	0
1	0	0	0	X	1	0	0
1	0	0	1	0	0	0	1
1	0	0	1	1	1	0	0

Remember when interpreting this table, because of the 1-hot shift structure, only 1 FF is 1 at a time, consequently it is straightforward to write down the next state equations

Tripos Example

Current state			Input		Next state		
s_2	s_1	s_0	e	r	s_2'	s_1'	s_0'
0	0	1	X	0	0	0	1
0	0	1	X	1	0	1	0
0	1	0	0	X	0	1	0
0	1	0	1	0	0	0	1
0	1	0	1	1	1	0	0
1	0	0	0	X	1	0	0
1	0	0	1	0	0	0	1
1	0	0	1	1	1	0	0

For FF 2:

$$D_2 = s_1.e.r + s_2.\bar{e} + s_2.e.r$$

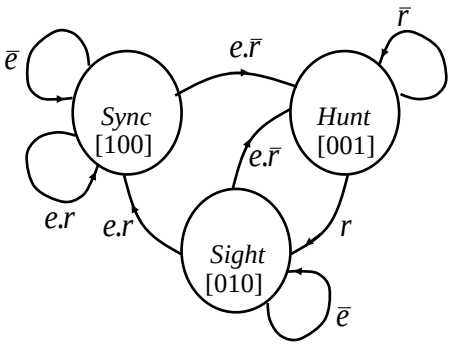
For FF 1:

$$D_1 = s_0.r + s_1.\bar{e}$$

For FF 0:

$$D_0 = s_0.\bar{r} + s_1.e.\bar{r} + s_2.e.\bar{r}$$

Tripos Example



Note that it is not strictly necessary to write down the state table, since the next state equations can be obtained from the state diagram

It can be seen that for each state variable, the required equation is given by terms representing the incoming arcs on the graph

For example, for FF 2: $D_2 = s_1.e.r + s_2.\bar{e} + s_2.e.r$

Also note some simplification is possible by noting that:

$s_2 + s_1 + s_0 = 1$ (which is equivalent to e.g., $s_2 = \overline{s_1 + s_0}$)

Tripos Example

- So in this example, the 1 hot is easier to design, but it results in more hardware compared with the sequential state assignment design

Elimination of Redundant States

- Sometimes, when designing state machines it is possible that unnecessary states may be introduced
- In general, reducing the number of states may reduce the number of FFs required and may also reduce the complexity of the next state logic owing to the presence of more unused states (don't cares)

Elimination of Redundant States - Example

- Consider the following State Table that corresponds with a Mealy Machine implementation
- This is so, since the inputs and outputs from the machine are on the transitions (arcs) between states
- The following state table is drawn in a compact form by incorporating the 2 possible input values as parallel columns within both the next state and output columns of the table

Example

Current State	Next State X=0	Next State X=1	Output (Z) X=0	Output (Z) X=1
A	B	C	0	0
B	D	E	0	0
C	F	G	0	0
D	H	I	0	0
E	J	K	0	0
F	L	M	0	0
G	N	P	0	0
H	A	A	0	0
I	A	A	0	0
J	A	A	0	1
K	A	A	0	0
L	A	A	0	1
M	A	A	0	0
N	A	A	0	0
P	A	A	0	0

- From the table, we see that there is no way of telling states H and I apart, so we can replace I with H when it appears in the Next State portion of the table

Example

Current State	Next State X=0	Next State X=1	Output (Z) X=0	Output (Z) X=1
A	B	C	0	0
B	D	E	0	0
C	F	G	0	0
D	H	H	0	0
E	J	K	0	0
F	L	M	0	0
G	N	P	0	0
H	A	A	0	0
J	A	A	0	1
K	A	A	0	0
L	A	A	0	1
M	A	A	0	0
N	A	A	0	0
P	A	A	0	0

- We also see that there is now no way to get to state I so we can remove row I from the table
- Similarly, rows K, M, N and P have the same next state and output as H and can be replaced by H

Example

Current State	Next State X=0	Next State X=1	Output (Z) X=0	Output (Z) X=1
A	B	C	0	0
B	D	E	0	0
C	F	G	0	0
D	H	H	0	0
E	J	H	0	0
F	L	H	0	0
G	H	H	0	0
H	A	A	0	0
J	A	A	0	1
L	A	A	0	1

- Similarly, there is now no way to get to states K, M, N and P and so we can remove these rows from the table
- Also, the next state and outputs are identical for rows J and L, thus L can be replaced by J and row L eliminated from the table

Example

Current State	Next State X=0	Next State X=1	Output (Z) X=0	Output (Z) X=1
A	B	C	0	0
B	D	E	0	0
C	F	G	0	0
D	H	H	0	0
E	J	H	0	0
F	J	H	0	0
G	H	H	0	0
H	A	A	0	0
J	A	A	0	1

- Now rows D and G are identical, as are rows E and F.
- Consequently, G can be replaced by D, and row G eliminated. Also, F can be replaced by E and row F eliminated from the table

Example

Current State	Next State X=0	Next State X=1	Output (Z) X=0	Output (Z) X=1
A	B	C	0	0
B	D	E	0	0
C	E	D	0	0
D	H	H	0	0
E	J	H	0	0
H	A	A	0	0
J	A	A	0	1

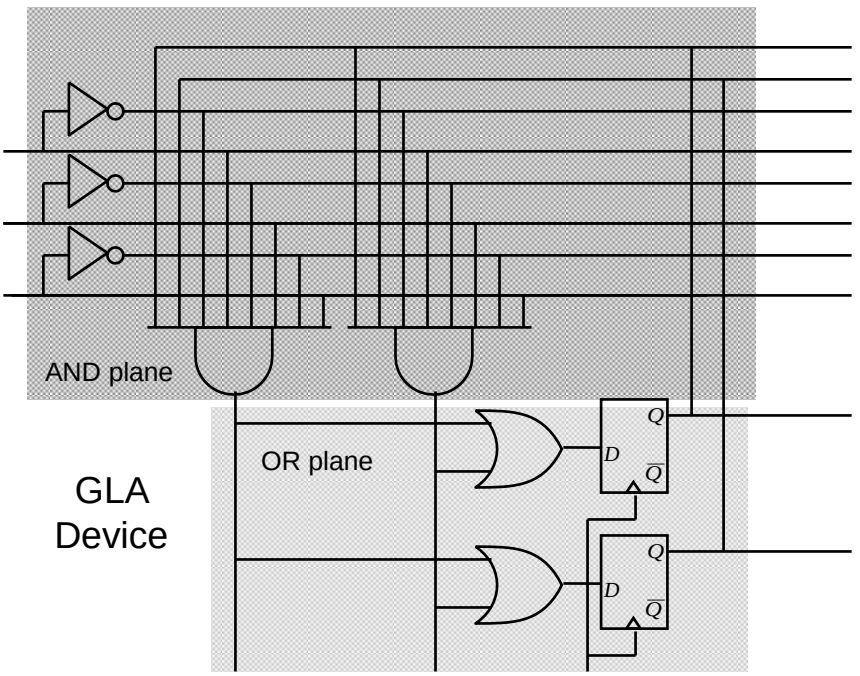
- The procedure employed to find equivalent states in this example is known as *row matching*.
- However, we note row matching is not sufficient to find all the equivalent states except for certain special cases

Implementation of FSMs

- We saw previously that programmable logic can be used to implement combinational logic circuits, i.e., using PLA devices
- PAL style devices have been modified to include D-type FFs to permit FSMs to be implemented using programmable logic
- One particular style is known as Generic Logic Array (GLA)

GLA Devices

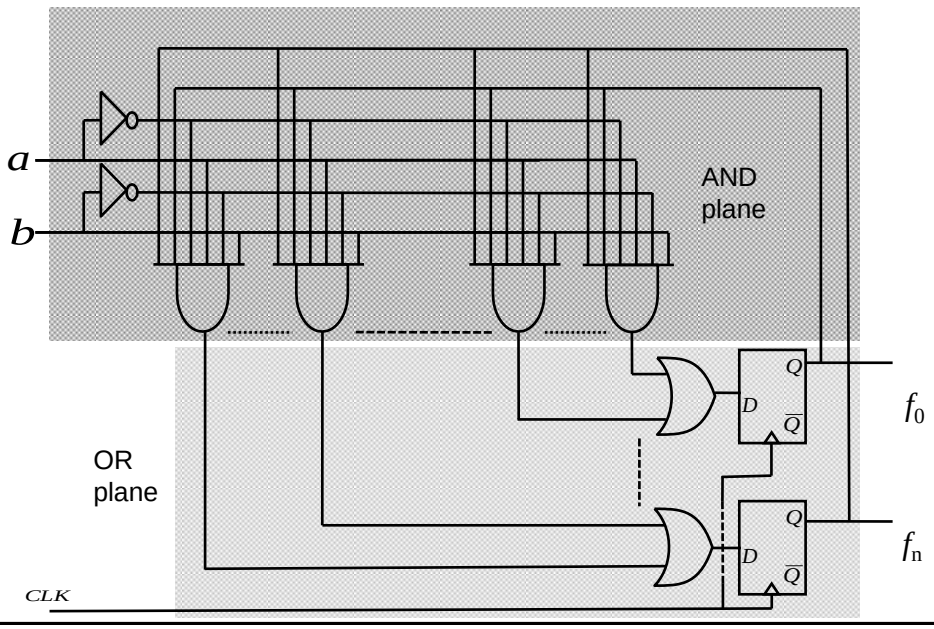
- They are similar in concept to PLAs, but have the option to make use of a D-type flip-flops in the OR plane (one following each OR gate). In addition, the outputs from the D-types are also made available to the AND plane (in addition to the usual inputs)
  - Consequently it becomes possible to build programmable sequential logic circuits



GLA Devices

- A modified form of a GLA known as a Generic Array Logic (GAL) is used in the Hardware Laboratory classes to implement various FSMs.

GAL Devices



FPGA

- Field Programmable Gate Arrays (FPGAs) are the latest type of programmable logic
- Are an array of configurable logic blocks (CLBs) surrounded by Input Output Blocks (IOBs):
  - programmable routing channels permit CLBs to be connected to other CLBs and to IOBs
  - CLBs contain look up tables (LUTs), multiplexers (MUXs) and D-type FFs
  - The FPGA is configured by specifying the contents of the LUTs and select signals for the MUXs

FPGA – Xilinx Spartan

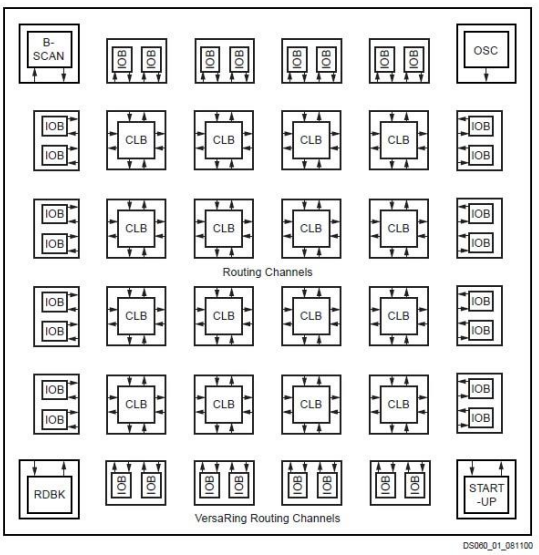
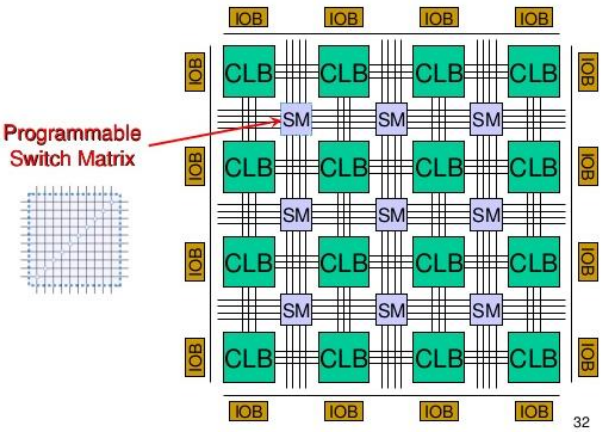


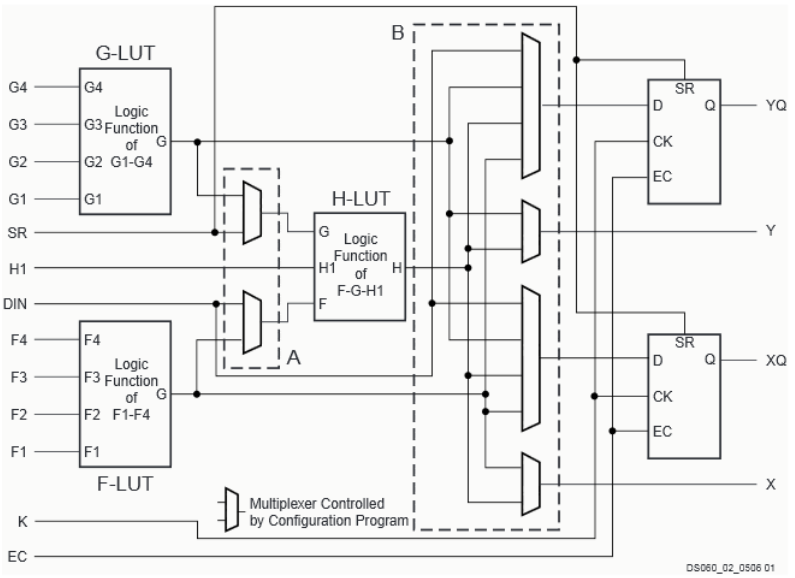
Figure 1: **Basic FPGA Block Diagram**

FPGA – Xilinx Spartan

- Simplified schematic showing CLBs and programmable routing channels, i.e., wires plus programmable switch matrices (SMs)



FPGA - Spartan CLB



FPGA - Spartan CLB

- Has 2, 4-input LUTs (F and G) and 1, 3 input LUT (H)
- Has to ‘combinational’ outputs (Y and X) and 2 ‘registered’ outputs (i.e., from D-FFs) YQ and XQ
- Depending on MUX configuration Y is given by output of either G or H LUTs and X from either F or H LUTs.
- D-FF inputs come from DIN, or from F, G, or H LUTs

FPGA - Spartan CLB

- Thus each CLB can perform up to 2 combinational and/or 2 registered functions
- All functions can involve at least 4 input variables (e.g., G1 to G4, and F1 to F4), but can be up to 9 (owing to the possibility of implementing 2-level combinational logic functions), i.e., G1 to G4, F1 to F4, DIN.
- Created using either a schematic (block) diagram or more likely a Hardware Description Language (HDL) of the design

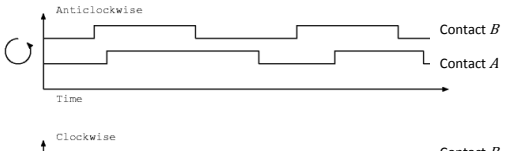
FPGA - Spartan CLB

- The synthesis tool determines how the LUTs, MUXs and routing channels are configured
- This configuration information is then downloaded to the FPGA
- Xilinx devices store their configuration information in static RAM (SRAM) so can be easily reprogrammed
- The SRAM contents can be downloaded either from a computer or from an EEPROM device when the system is powered-up

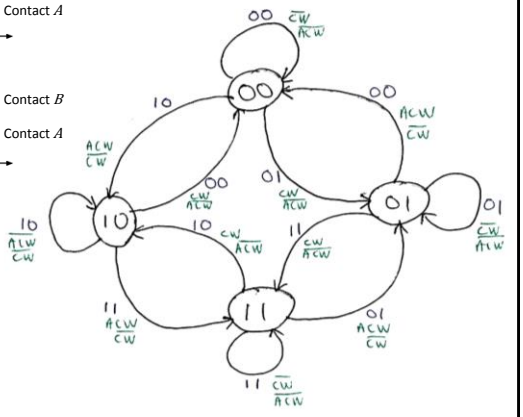
FPGA

- Other FPGA manufacturers are available, e.g., Altera.
- Particular manufacturers have many different product lines
- Main differences will be the no. of CLBs, the structure of the CLBs, internal or external ROM, additional features such as specialised arithmetic blocks, user RAM etc.

Appendix – Workshop 2



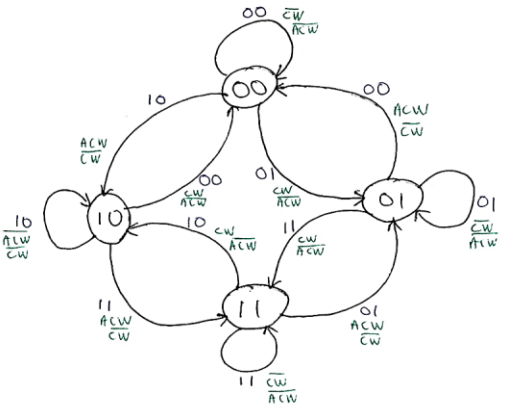
Workshop 2 - State Diagram



Q_B Q_A State
B A Inputs
CW ACW Outputs

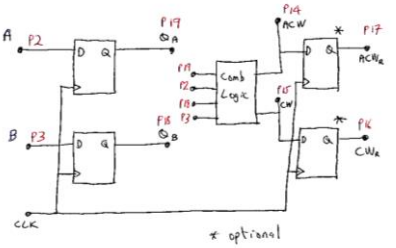
Appendix – Workshop 2

Workshop 2 - State Diagram



Q_B Q_A State
B A Inputs
CW ACW Outputs

Workshop 2 MEALY SOLUTION - REGISTERED



Q_B Q_A State register outputs
B, A Switch contacts
P2, P3 FRC Inputs

Current State Q_B P18	Q_A P19	Next State = Input B P3	A P2	Output CW P15	ACW P14
0	0	0	0	0	0
0	0	0	1	1	0
0	0	1	0	0	1
⋮		⋮		⋮	

Plus dont cares,

0	0	1	1	x	x
0	1	1	0	x	x
1	0	0	1	x	x
1	1	0	0	x	x