ALGORITHMIC STATE MACHINES

Overview…

Clocked synchronous state machines

- State machine structures – Mealy and Moore types.

Analysis of state machines.

Algorithmic state machine (ASM) chart notation.

Synthesis of state machines from ASM Chart

- Traditional, Multiplexer, One-hot methods.

Practical Design Considerations (clock skew, input signal sync)

Design examples – a few, diverse examples illustrating the *top-down digital circuit design process*.

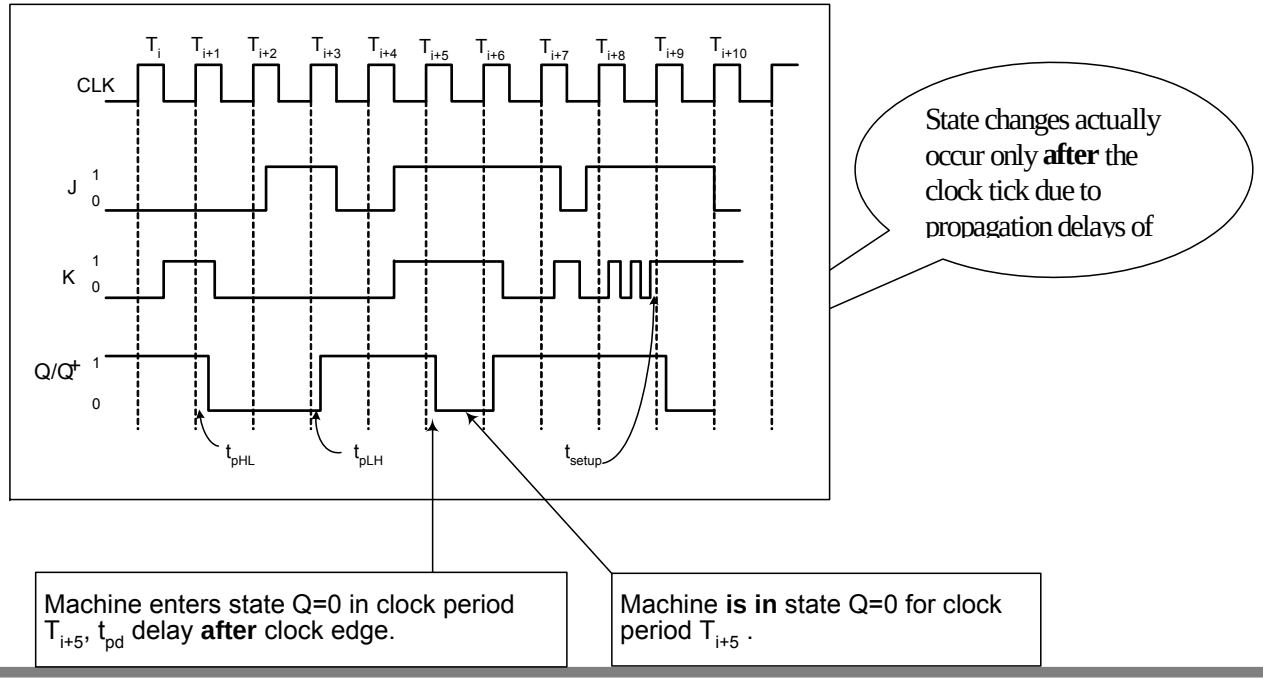
- Modular design of state machines – example.

Clocked synchronous state machines…

Meaning:

- *Clocked synchronous*: clock signal is directly connected to all components with clock inputs.
- *State machine*: a circuit with sequential (& combinational) circuit elements.

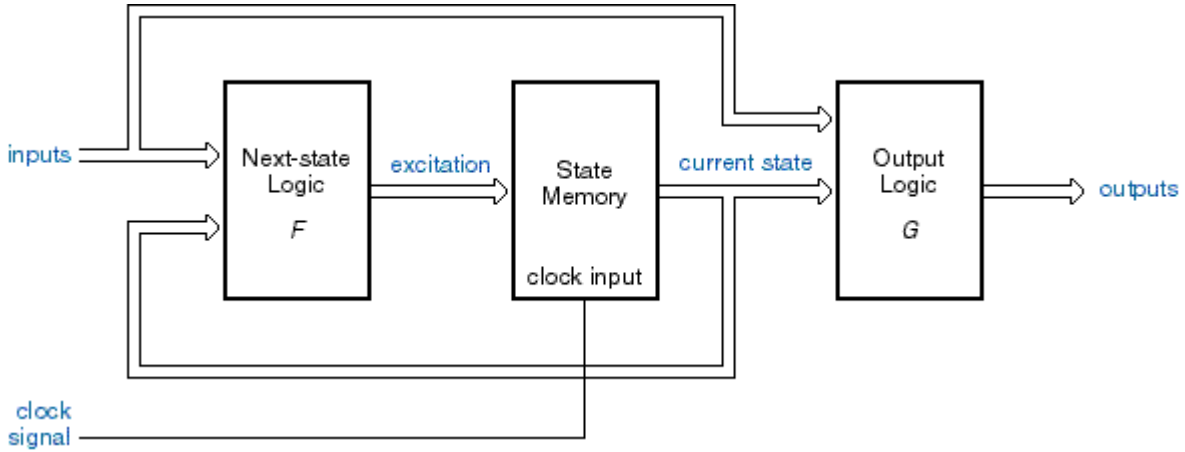
State machines change state only at *active clock transition*.



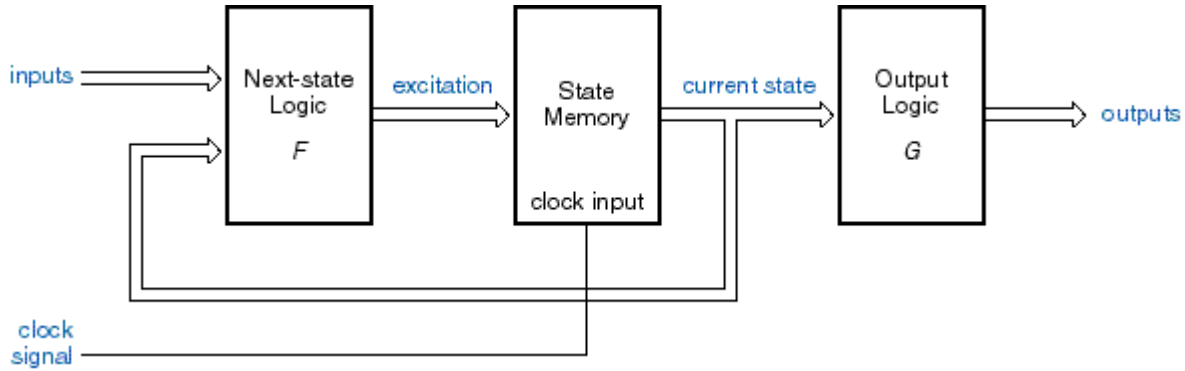
State machine structure…

Two types: Mealy and Moore machines.

- **Mealy machine:** ***output is a function of a present state & inputs.***



- **Moore machine:** ***output is a function of a present state only.***



State machine structure… cont'd

State memory

- set of *n* FFs store current state of machine; up to 2^n states.
- FFs can be J-K or D, but D FFs preferred as they are simpler *(1 input vs. 2 inputs for J-K FFs)*

Next state logic

- a combinational circuit which decides the next state of the machine based on current state and inputs:

$$Next\ state = f(inputs, current\ state)$$

Output logic for Mealy machines

- outputs depend on the current state as well as the inputs.

$$outputs = g(inputs, current\ state)$$

Moore & Mealy machines differ only in the output logic; in Moore machines, outputs are functions of current state only.

$$outputs = g(current\ state)$$

Analysis of state machines… *Goals*..

Characterize as Mealy or Moore machine

Determine next-state as function of inputs & current state

- as in design of synchronous counters..
- set of *n* FFs store current state of machine; up to 2^n states.

Determine output as function of current state (Moore) or as function of current state & current inputs (Mealy).

- decode state bits to identify particular states & generate outputs

Express as machine behavior as state/output table or as state diagram.

Describe machine behavior.

Algorithmic state machine ASM chart notation

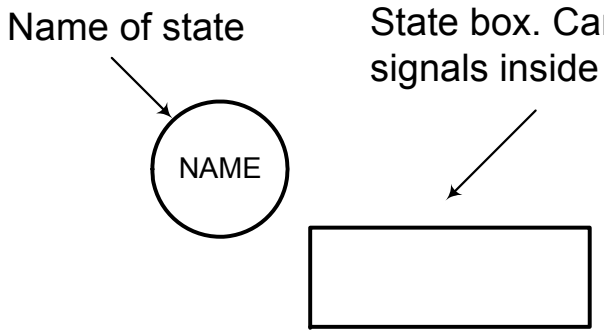
This is notation for describing synchronous state machines.

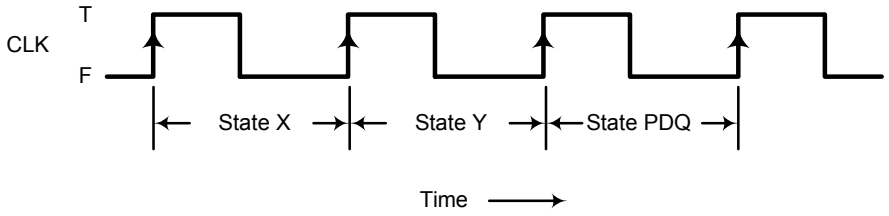
ASM charts bear a superficial resemblance to flow charts:

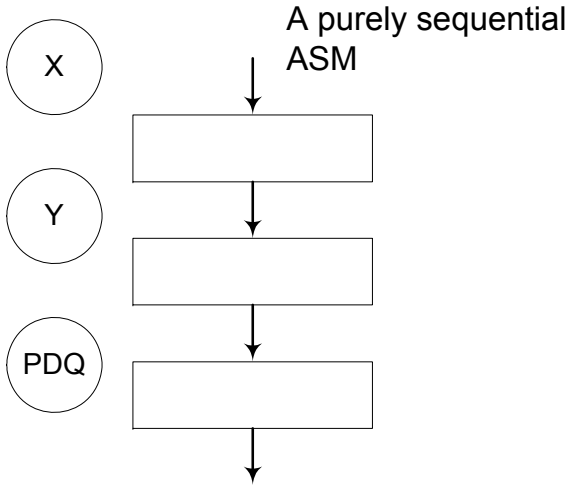
- **difference is that ASM charts have the concept of a sequence of time intervals built in.**
- **flow charts describe only the sequence of events, not their duration.**

In ASMs, machine changes from present state to next state at the *active clock transition* (ACT) & remains there for duration of clock cycle.

ASM.. notations



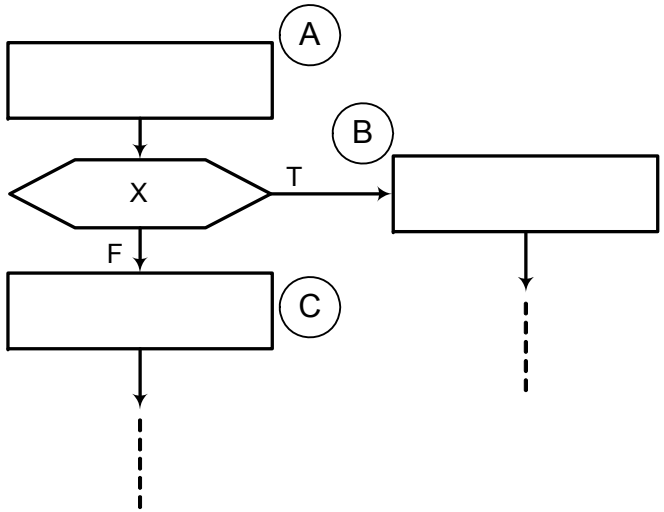


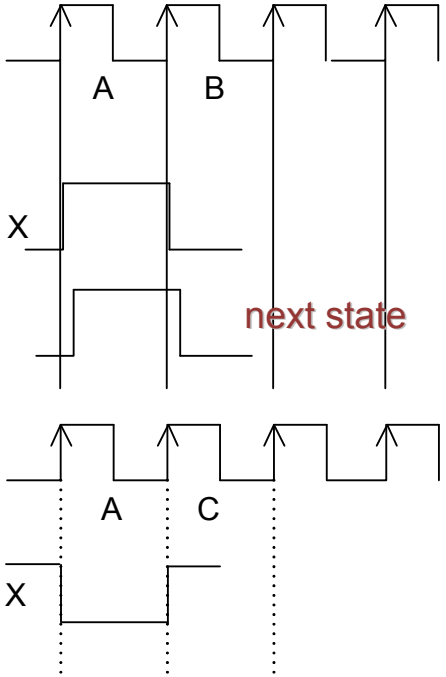


The waveform shows the transition from one state to the next at the clock tick. The ASM notation at right is equivalent to the waveform notation.

ASM.. with a conditional branch

An ASM with a conditional branch



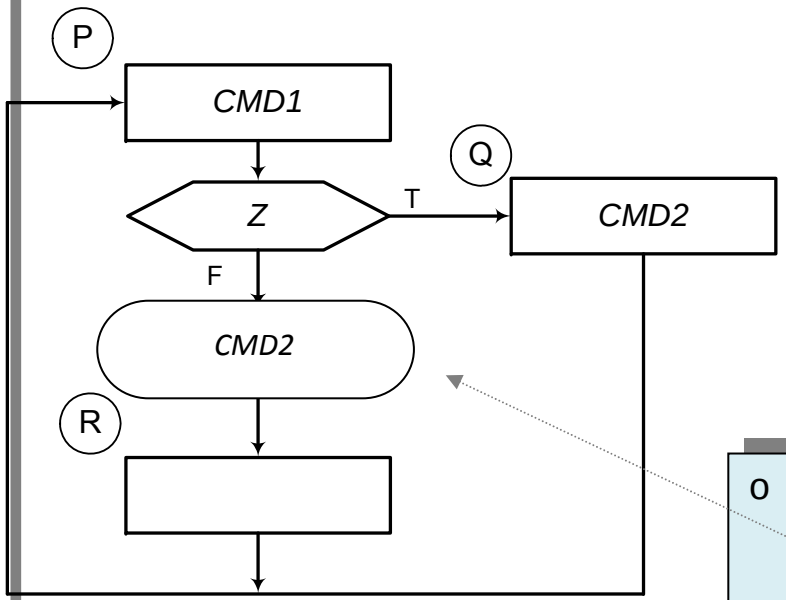


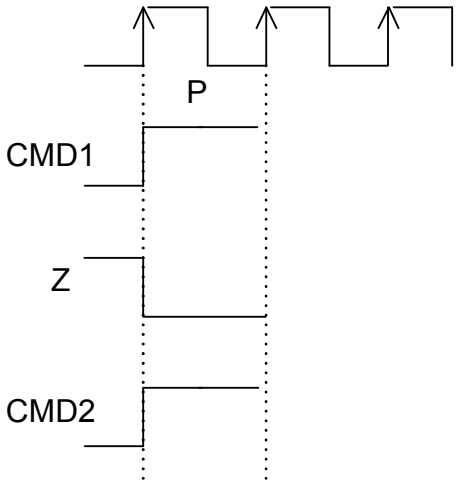
X is an input signal whose value decides what the next state of the machine should be when it is currently in state A.

- Decision to jump to state B or C from state A is made during state A.
- Jump occurs at next **ACT**.
- Test does NOT require a separate clock period – it is done in state A. Hence, decision diamond is part of parent state A.

ASM.. with conditional (& unconditional) outputs

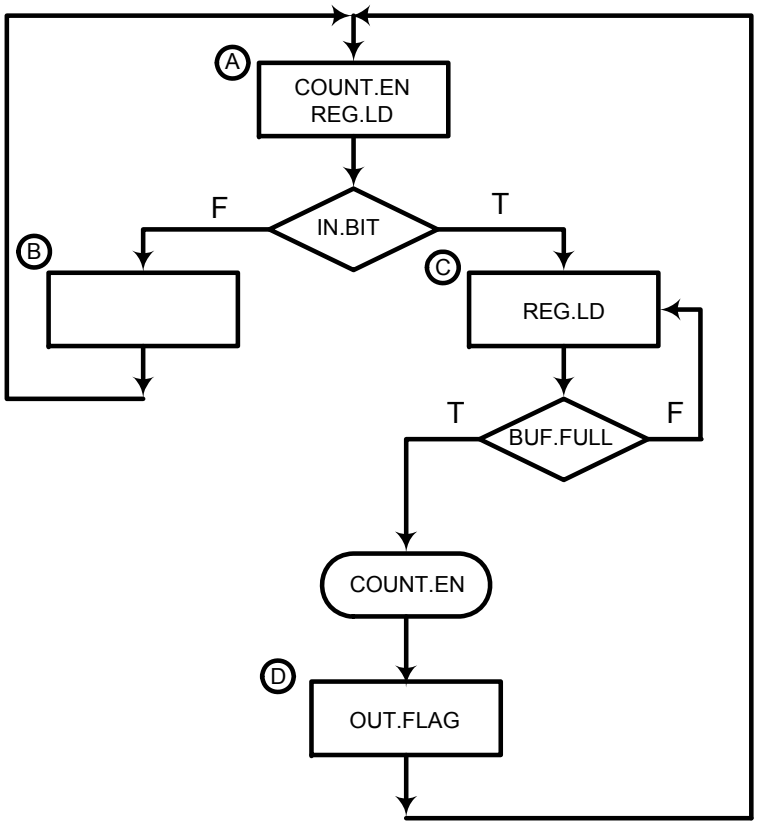
An ASM with a conditional output



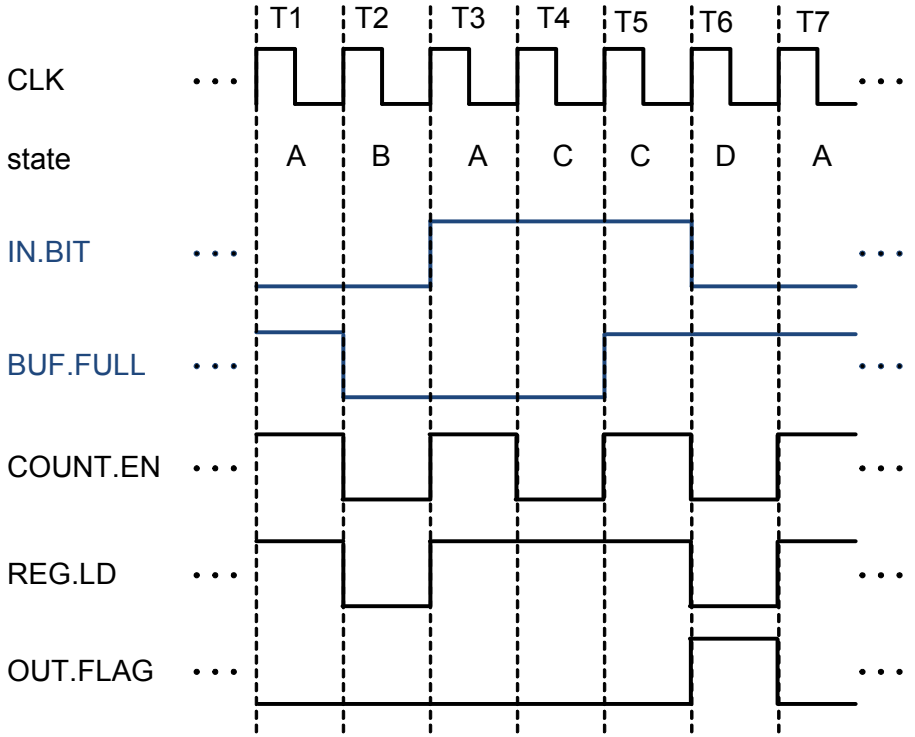


- A signal written within a **state rectangle** is 1 whenever the controller is in that state. These are called ***unconditional outputs***.
- If an output signal also depends on the value of an input signal, it is called a ***conditional output***. This is indicated by writing the signal in an oval box.
- In this example, **CMD2** is **unconditional** in state Q, and conditional in state P.

Analysis of state machines… An example...



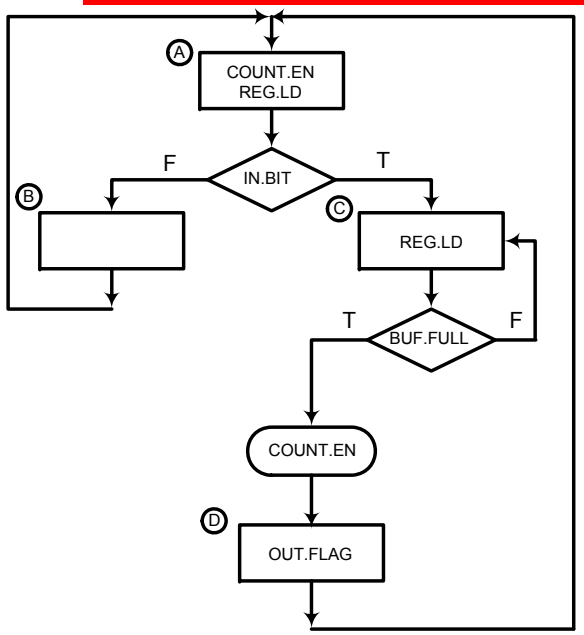
(a)



Above ASM chart shows how the state machine makes state transitions as a function of current states & inputs.

ASM charts, next-state table and timing diagrams are all equivalent descriptions. Each can be inferred from the other.

Analysis of state machines… Next state table...



(a)

Next state table is inferred from ASM chart:

o given a **current state A,B,C or D** and **inputs** IN.BIT & BUF.FULL ► next state

For realization of state machine, **state code bits** C1 C0 used to represent states:

$A=\overline{C1}\,\overline{C0}$ $B=\overline{C1}C0$ $C=C1\overline{C0}$ $D=C1C0$

But next tables below do not completely represent the state machine. *Why?*

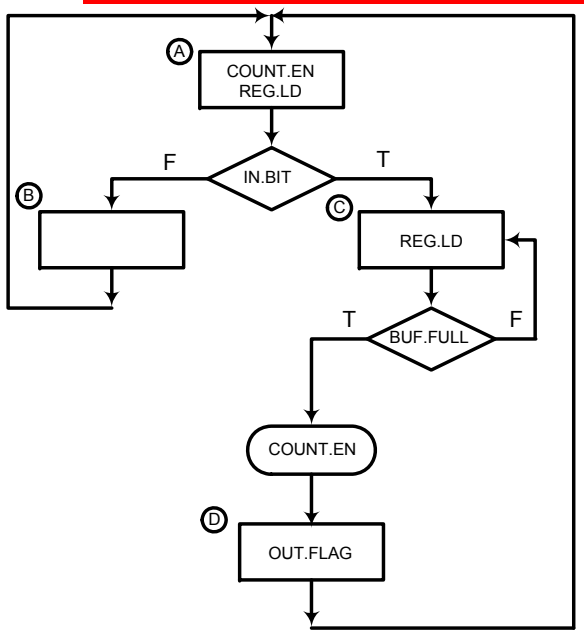
	IN.BIT, BUF.FULL			
S	**0 0**	**0 1**	**1 1**	**1 0**
A	B	B	C	C
B	A	A	A	A
D	A	A	A	A
C	C	D	D	C

Next state table in terms of state names.

	IN.BIT, BUF.FULL			
C1, C0	**0 0**	**0 1**	**1 1**	**1 0**
0 0	0 1	0 1	1 0	1 0
0 1	0 0	0 0	0 0	0 0
1 1	0 0	0 0	0 0	0 0
1 0	1 0	1 1	1 1	1 0

Next state table in terms of state code bits..

State machine structure… System I



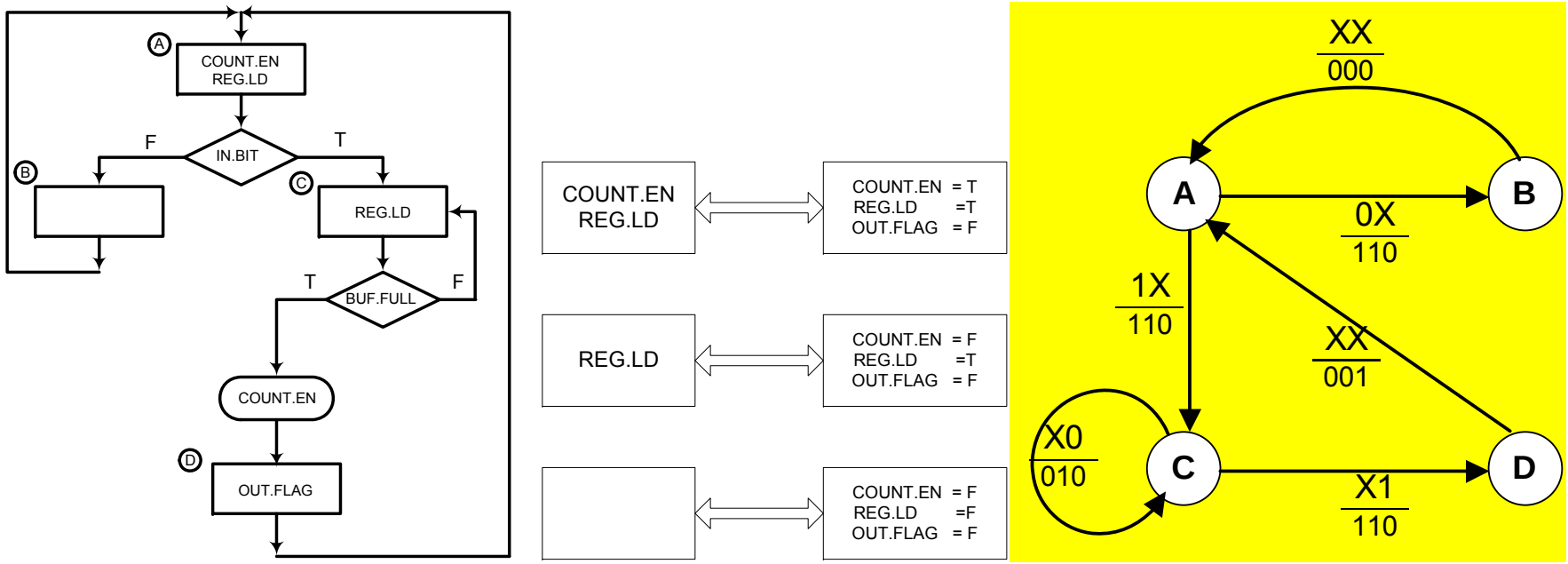
(a)

	IN.BIT, BUF.FULL			
S	**0 0**	**0 1**	**1 1**	**1 0**
A	B (1,0,1)	B (1,0,1)	C (1,0,1)	C (1,0,1)
B	A (0,0,0)	A (0,0,0)	A (0,0,0)	A (0,0,0)
D	A (0,1,0)	A (0,1,0)	A (0,1,0)	A (0,1,0)
C	C (1,0,0)	D (1,0,1)	D (1,0,1)	C (1,0,0)

*Next state table **and** current outputs (REG.LD, OUT.FLAG, COUNT.EN) in terms of state names.*

Next state table above completely describes state machine behavior as in the Algorithmic State Machine (ASM) chart which shows state transitions as well as outputs.

Comparing ASM chart vs. State Transition Graph



(a)

Note: ASM chart uses minimal notation – no clutter.

Inputs : IN.BIT , BUF.FULL .
Outputs: COUNT.EN, REG.LD, OUT.FLAG

Synthesis of state machines from ASM Chart

Previously, we saw how state machine operation could be inferred from circuit diagram.

Now let's consider the reverse: given an ASM chart or next state/output table or state transition graph: *how to realize the state machine circuit* (*synthesis*).

The first task is to represent controller states numerically. This is called state assignment.

Previous ASM chart had states A,B,C,D; each of these states must be assigned a code number to be used in a digital circuit. For example (there are 4 states, so need two bits to represent them):

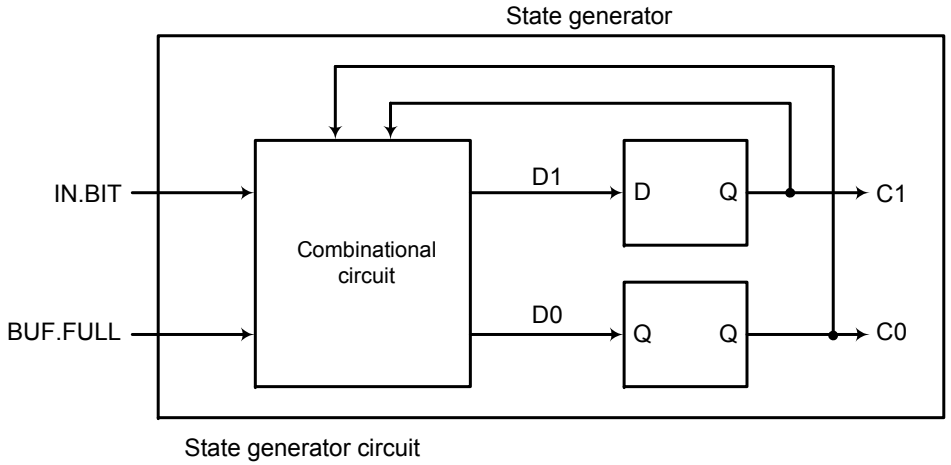
State Name	State number	State Bits C1 C0
A	0	0 0
B	1	0 1
C	2	1 0
D	3	1 1

Synthesis from ASM Chart ... continued

Thus, controller states can be stored in FFs and in this example, 2 FFs (D, J-K, etc.) are needed.

Next, given an ASM chart for which state assignment has been done, we must first realize a state generator circuit which will be responsible for *moving* the machine from one state to another depending on inputs & the current state. ***(Note: this is similar to designing synchronous counters).***

For the example ASM chart, the functional block diagram of the state generator circuit:



Synthesis from ASM Chart

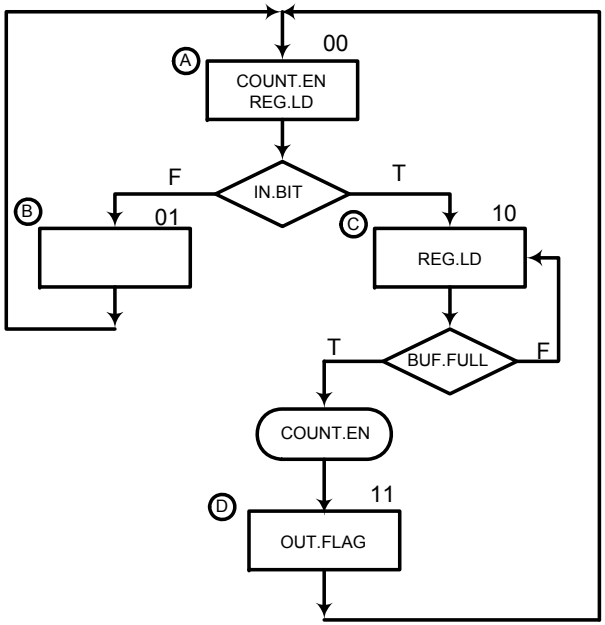
This basically involves the design of the *combinational circuit* which will use the inputs & *current state* of machine to move it into required *next state* (next state logic). *(like designing synchronous counters).*

Following this, the output logic circuit is (easily) realized and there are several methods to realize the next state logic:

- **Traditional method (with SSI chips)**
- **PLA/PAL based method (LSI)**
- **Multiplexer based method (MSI)**
- **One-hot method.**

Depending on problem, one of these methods can be chosen.

ASM Synthesis...: 1. Traditional method



Example ASM chart with state assignment

Equations for outputs:
REG.LD = A + C
OUT.FLAG = D
COUNT.EN = A + C BUF.FUL

	IN.BIT, BUF.FULL			
C1, C0	0 0	0 1	1 1	1 0
0 0	0 1	0 1	1 0	1 0
0 1	0 0	0 0	0 0	0 0
1 1	0 0	0 0	0 0	0 0
1 0	1 0	1 1	1 1	1 0

Next state table for C1+, C0+ in terms of state bits.

Since D-FFs are used, D1= C1+ and D2=C0+.

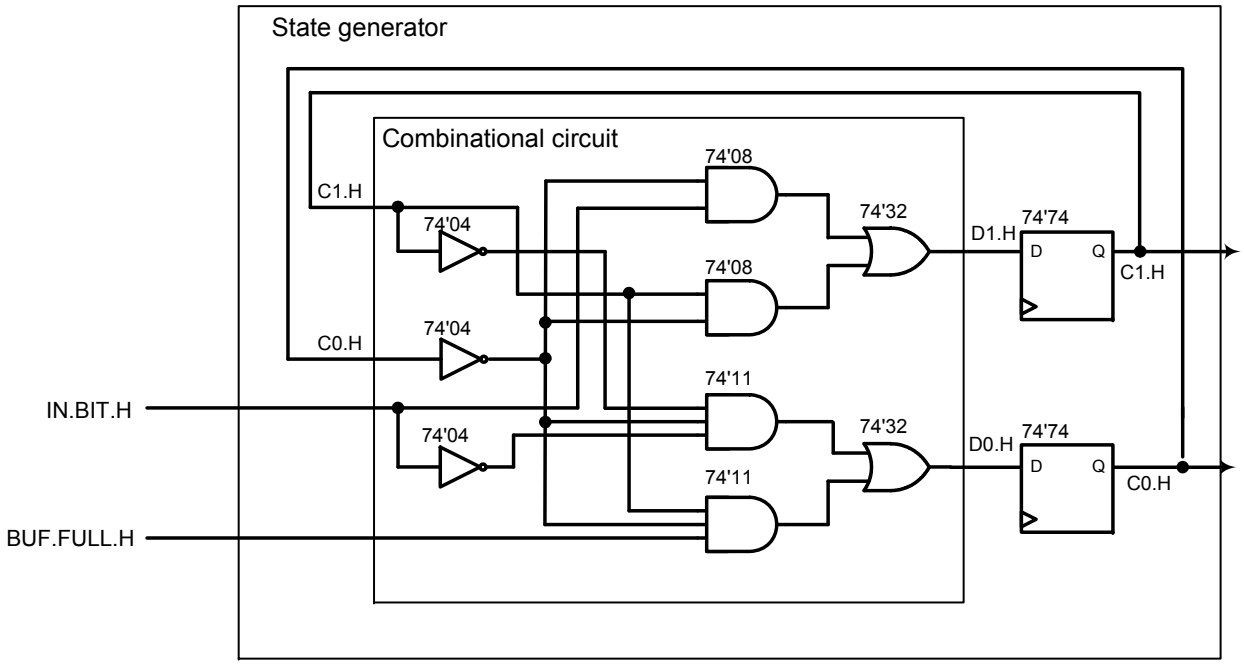
The **Karnaugh maps** for D1 & D2 follow from the above ***next state*** table.

IN.BIT BUF.FULL \ C1 C0	00	01	11	10
00	0	0	0	1
01	0	0	0	1
11	1	0	0	1
10	1	0	0	1

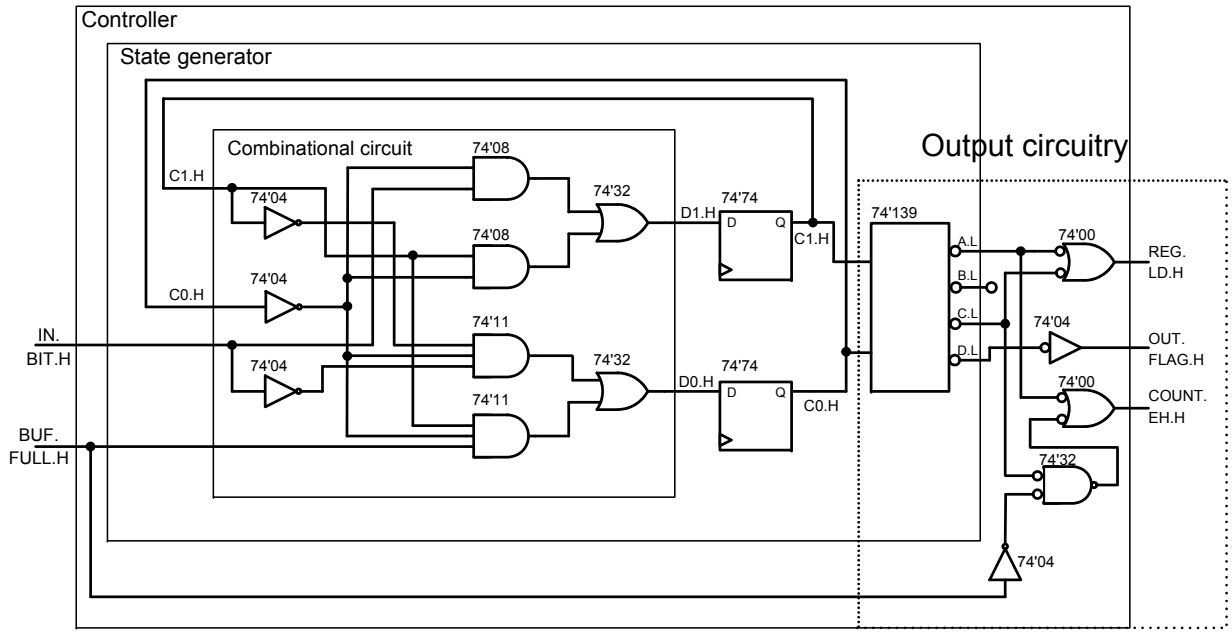
$D1 = \overline{C0} \cdot IN.BIT + C1 \cdot \overline{C0}$

IN.BIT BUF.FULL \ C1 C0	00	01	11	10
00	1	0	0	0
01	1	0	0	1
11	0	0	0	1
10	0	0	0	0

$D0 = \overline{C1} \cdot \overline{C0} \cdot \overline{IN.BIT} + C1 \cdot \overline{C0} \cdot BUF.FULL$



D flip-flop state generator realization



Controller realization with D flip-flops for the ASM chart

ASM Synthesis...: 1. Traditional method : *Some points to note!*

- In traditional method, either **J-K** or **D flip-flops** can be used.
  - **J-K FFs** lead to somewhat more compact designs because J-K FFs are more flexible than D's but more design work needed to find the logic for 2 inputs.
- No *obvious correspondence* between circuit and ASM chart.
- Minor changes in algorithm require a fresh design of next-state combinational circuit.
- Lacks clarity of design.

ASM Synthesis...: 2. PAL/PLA based method

	IN.BIT, BUF.FULL			
C1, C0	0 0	0 1	1 1	1 0
0 0	0 1	0 1	1 0	1 0
0 1	0 0	0 0	0 0	0 0
1 1	0 0	0 0	0 0	0 0
1 0	1 0	1 1	1 1	1 0

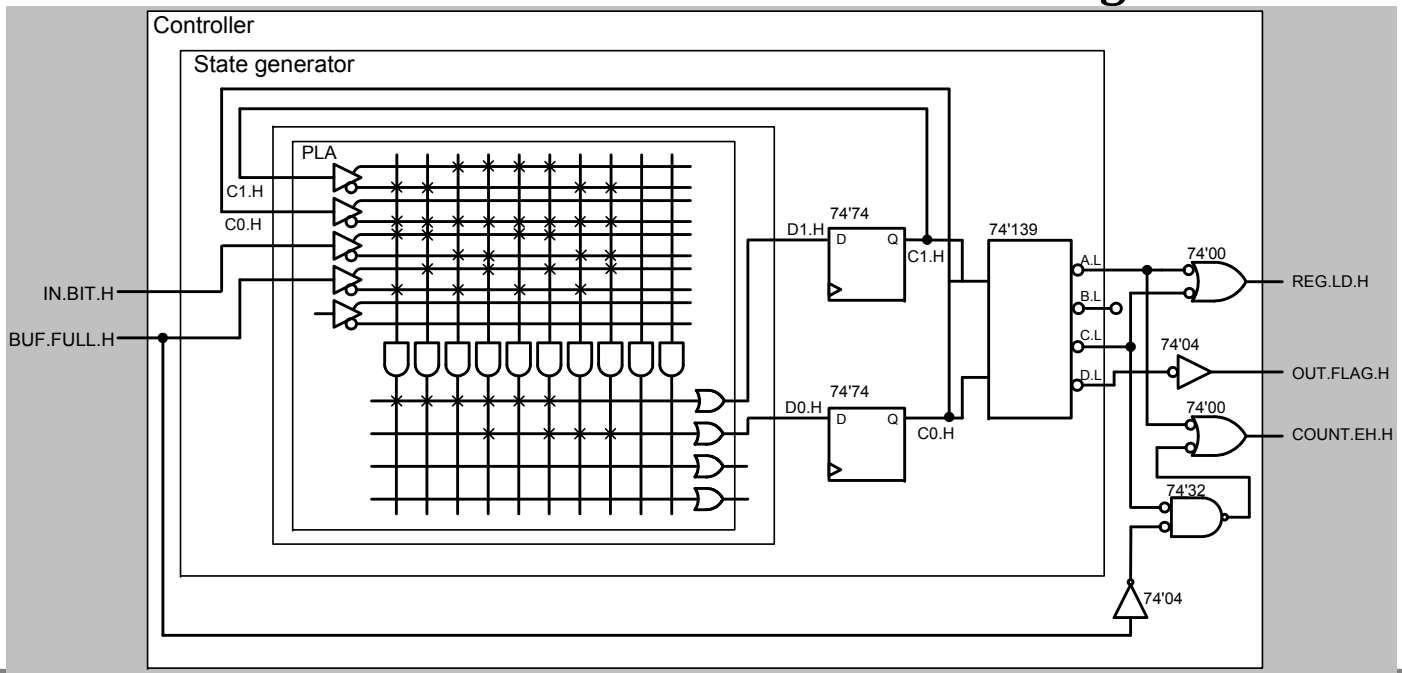
Derive next-state table for state generator (as before).

Determine input equations of FFs composed of minterms from truth table.

$$D1 = \overline{C1}\cdot\overline{C0}\cdot IN.BIT\cdot\overline{BUF.FULL} + \overline{C1}\cdot\overline{C0}\cdot IN.BIT\cdot BUF.FULL + C1\cdot\overline{C0}\cdot IN.BIT\cdot\overline{BUF.FULL} + C1\cdot\overline{C0}\cdot\overline{IN.BIT}\cdot BUF.FULL + C1\cdot\overline{C0}\cdot IN.BIT\cdot BUF.FULL$$

$$D0 = \overline{C1}\cdot\overline{C0}\cdot\overline{IN.BIT}\cdot\overline{BUF.FULL} + \overline{C1}\cdot\overline{C0}\cdot\overline{IN.BIT}\cdot BUF.FULL + C1\cdot\overline{C0}\cdot IN.BIT\cdot\overline{BUF.FULL} + C1\cdot\overline{C0}\cdot IN.BIT\cdot BUF.FULL$$

This leads to 2-level AND/OR circuit realization using PLA/PAL.



Controller with PLA state generator realization for the ASM chart

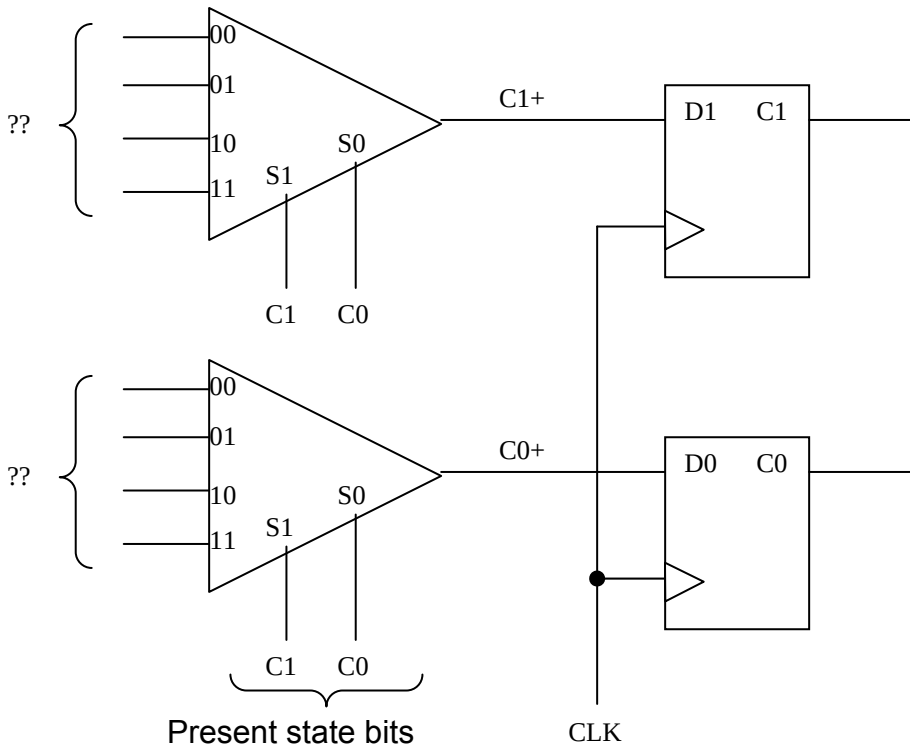
ASM Synthesis...: 2. PAL/PLA based method

Some points to note

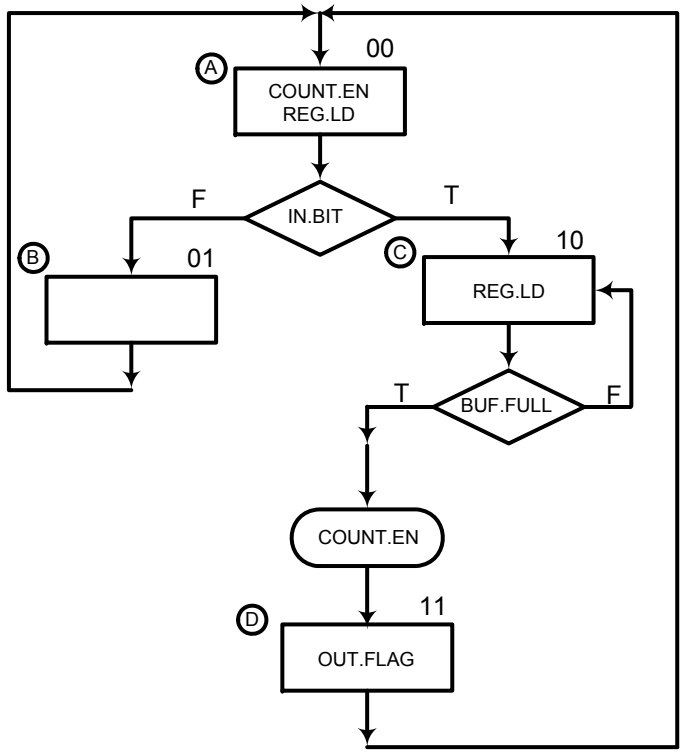
- Obvious advantage of this method over traditional method is **replacement of several IC's with one**.
- Dramatic **reduction in package count** for complex controllers.
- Method becomes even more attractive when **PLD contains on-board FFs** – possible to realize entire controller with one chip.
- With PLD devices, it makes sense to use D FFs instead of J-K FFs – need to provide logic for only one D input rather 2 for J-K FFs.

ASM Synthesis...: 3. Multiplexer based method

Here the combinational circuit required in the state generator is realized using multiplexers (to "look up" the next state).



- One **MUX** is used to compute the next state for each flip-flop.
- Each **MUX** must be wide enough to have an input for each ASM state.
- The present states of the FFs provide the selection control inputs to the **MUX**.
- So now the problem can be looked at as follows: *given the present state, what must be the signal at the corresponding **MUX** input to drive the machine to the correct state at the next ACT*.
- This can be done by inspection!
- Consider the inputs of one **MUX** at a time, e.g. first the **MUX** who's output is C1+, and then the **MUX** with output C0+.



Example ASM chart with state assignment

PRESENT STATE: C_1 C_0 = 0 0:

NEXT STATE (C1+ C0+)	CONDITION	MUX INPUTS
0 1	if IN.BIT=0	MUX_{C1}= IN.BIT
1 0	if IN.BIT=1	MUX_{C0}= IN.BIT

PRESENT STATE: C_1 C_0 = 0 1:

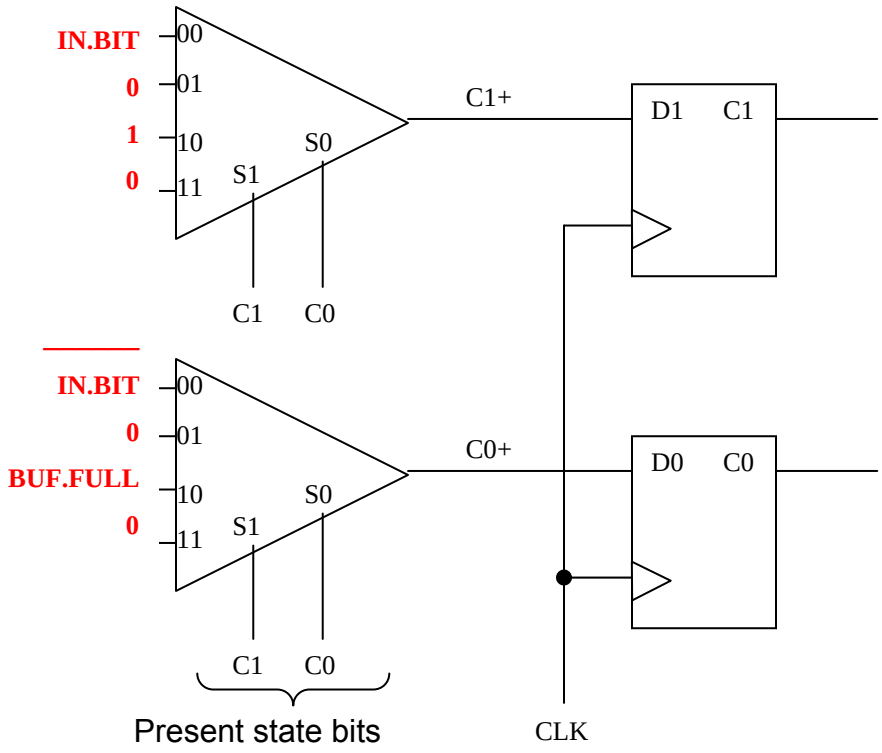
NEXT STATE (C1+ C0+)	CONDITION	MUX INPUTS
0 0	ALWAYS	MUX_{C1}= 0
	ALWAYS	MUX_{C0}= 0

PRESENT STATE: C_1 C_0 = 1 0:

NEXT STATE (C1+ C0+)	CONDITION	MUX INPUTS
1 0	if BUF.FULL=0	MUX_{C1}= 1
1 1	if BUF.FULL=1	MUX_{C0}= BUF.FULL

PRESENT STATE: C_1 C_0 = 1 1:

NEXT STATE (C1+ C0+)	CONDITION	MUX INPUTS
0 0	ALWAYS	MUX_{C1}= 0
	ALWAYS	MUX_{C0}= 0



PRESENT STATE: C_1 C_0 = 0 0:

NEXT STATE (C1+ C0+)	CONDITION	MUX INPUTS
0 1	if IN.BIT=0	MUX_{C1}= IN.BIT
1 0	if IN.BIT=1	MUX_{C0}= IN.BIT

PRESENT STATE: C_1 C_0 = 0 1:

NEXT STATE (C1+ C0+)	CONDITION	MUX INPUTS
0 0	ALWAYS	MUX_{C1}= 0
	ALWAYS	MUX_{C0}= 0

PRESENT STATE: C_1 C_0 = 1 0:

NEXT STATE (C1+ C0+)	CONDITION	MUX INPUTS
1 0	if BUF.FULL=0	MUX_{C1}= 1
1 1	if BUF.FULL=1	MUX_{C0}= BUF.FULL

PRESENT STATE: C_1 C_0 = 1 1:

NEXT STATE (C1+ C0+)	CONDITION	MUX INPUTS
0 0	ALWAYS	MUX_{C1}= 0
	ALWAYS	MUX_{C0}= 0

ASM Synthesis...: 3.MUX method : Some points to note

- ASM controller outputs formed as before.
- Since **MUX** inputs obtained directly from ASM charts, greater clarity.
- Size of **MUX** increases linearly with number of states.

No. of states	No. of MUXs	MUX input size
5-8	3	8
9-16	4	16
17-32	5	32

- Method attractive up to about 16 states, unwieldy after that. For greater than 16 states, the one-hot method may be better.

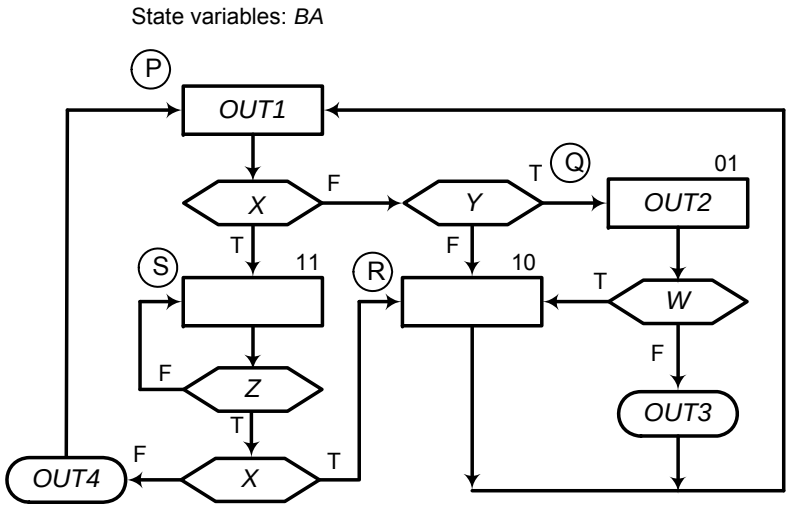
Dealing with unused states: ***minimum cost*** *vs minimum risk*.

- Minimum cost assumes that a machine will never enter an unused state.
  - Then the next state in the next state table is marked as don't care. This can simplify excitation logic and hence, cost.
  - But what happens if machine does get into an unused state, e.g. during power up or due to noise? Behaviour could be weird.
- Minimum risk designs explicitly guard against this:
  - If machine ever goes into an unused state, must provide a way to get back to the main algorithm, e.g. choose to go to a particular state (*arbitrarily can be chosen*).

ASM Synthesis...: 4. One Hot method

One FF used per state → no state assignments are necessary.

The design is such that only one state FF is 1 during each clock period – hence the name "one hot".



STATE TRANSITION DATA FOR A ONE-HOT IMPLEMENTATION OF THE ASM

Next state	Present state	Condition for transition
P	Q	$\overline{W}$
	R	T
	S	$Z \cdot \overline{X}$
Q	P	$\overline{X} \cdot Y$
R	P	$\overline{X} \cdot \overline{Y}$
	Q	W
	S	$X \cdot Z$
S	P	X
	S	$\overline{Z}$

Equations for FF inputs from the table:

next state P: P(D) = Q$\overline{W}$ + R + SZX

next state Q: Q(D) = P$\overline{XY}$

next state R: R(D) = PXY + Q$\overline{W}$ + SXZ

next state S: S(D) = PX + SZ

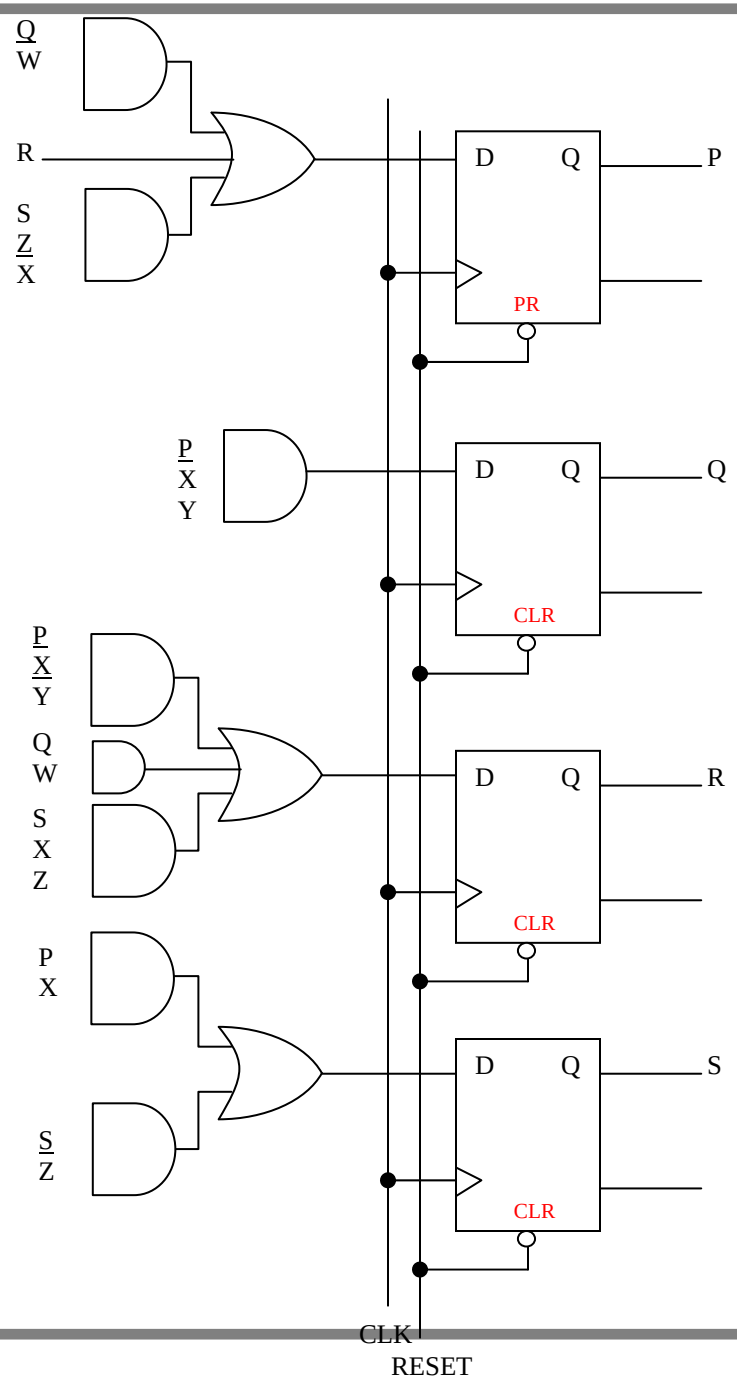
Controller Outputs:

OUT1 = P

OUT2 = Q

OUT3 = $\overline{QW}$

OUT4 = SZ$\overline{X}$



- Generation of controller outputs, not shown here (but which must be done to complete the design), is simple.
- The **one-hot** controller must be initialized properly when powered up, i.e., exactly one FF representing the state is 1, and all others are 0.
- Once properly initialized, controller will sequence through all states correctly.
- Initialization can be done by using a master reset (**asynchronous**) signal on power up.
- Use of this idea is shown in previous diagram, where it is assumed that P is the starting state. (Note clever use of mixed logic notation).

4. One-Hot: Some points to note

- Design is easy, and circuits easily relate to ASM chart.
- Size of state generator circuit grows linearly with number of states (not bad).
- Since many FFs are available on one chip, package count compares favorably with other methods.

ASM Synthesis. Practical Design Considerations

Assumptions made about the systems:

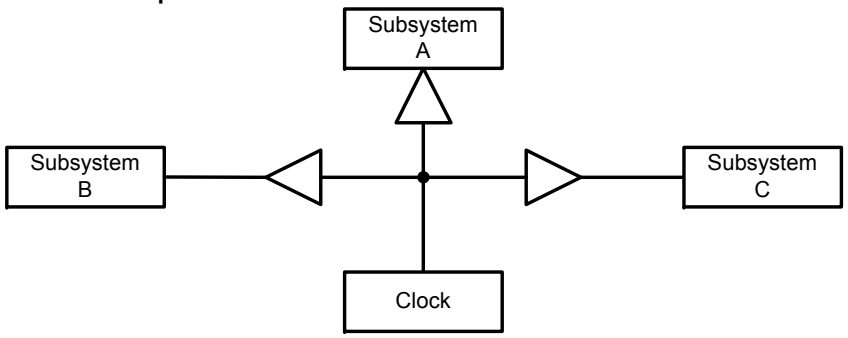
(a) State machines are **synchronous**, actions **governed by a master clock**.

(b) At time of state change, all inputs to ASM are stable, i.e. inputs change sync. with system clock.

(c) Clock edge reaches each circuit element **simultaneously**.

Violation of (a) or (b) will lead to serious problems; they are the basis of synchronous design. (c) Can be enforced by good construction practices.

- Do **NOT** gate the clock!
- Use either all +ve edge triggered FFs or all –ve edge triggered FFS.
- Beware different lengths of clock paths.Use:

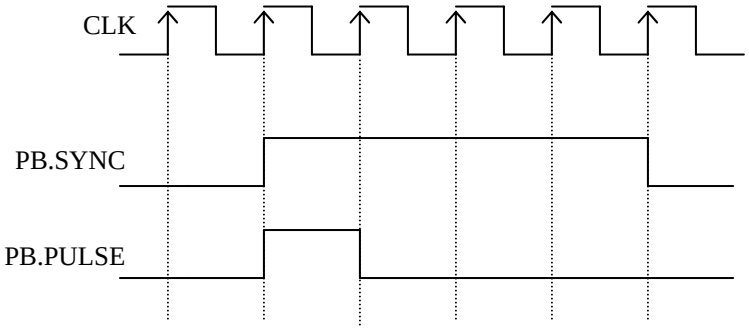


A buffered radial distribution system
for the master clock

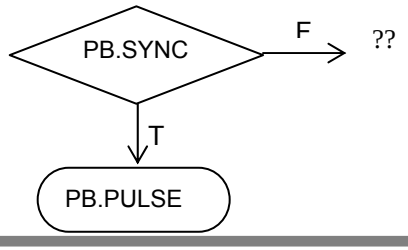
- **Asynchronous inputs** cause problems, and must be avoided. A common example of an asynchronous input is operator controlled push button switches – these are **not** synchronized to the clock.
- Two kinds of problems can occur with asynchronous signals, ***transition races***, *and* ***output races***.

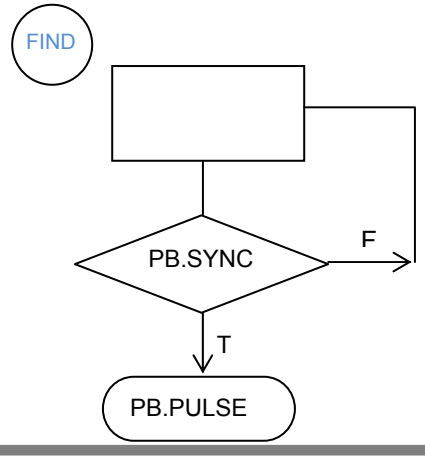
Design example1 : Single pulser circuit.

Design a circuit using a debounced push button which produces a clock synchronized signal PB.SYNC (i.e., satisfies set-up & hold times) and operates as follows:
(a) Asserts an output PB.PULSE for one clock period when debounced push button switch is pressed.
(b) Disallows additional assertions of output until PB is released.



- Translate understanding of problem into an ASM chart, next state/output table, etc.



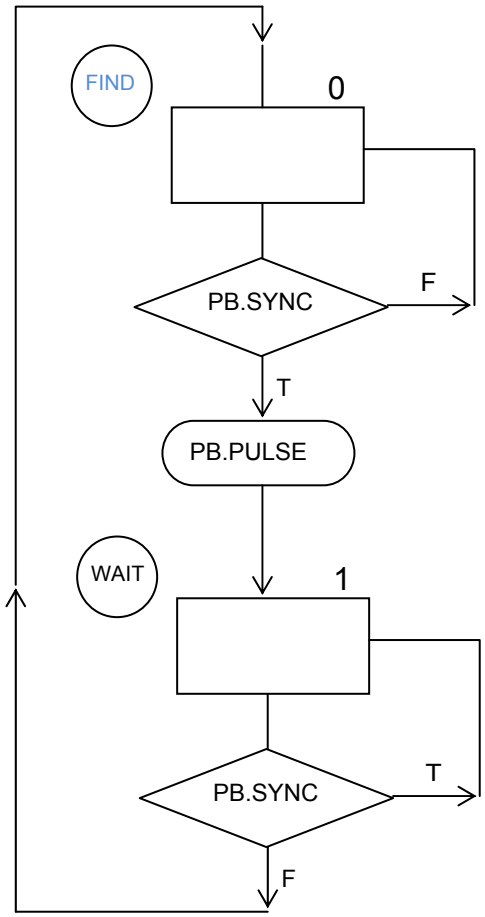


Design example1: Single pulser circuit.

Final ASM chart with state assignments shown.

State variable: A

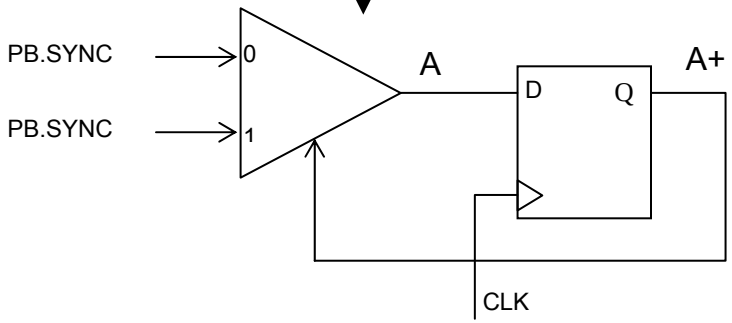
A=1: WAIT
A=0: FIND



Output:
PB.PULSE = FIND . PB.SYNC
= $\overline{A}$. PB.SYNC

Realization:

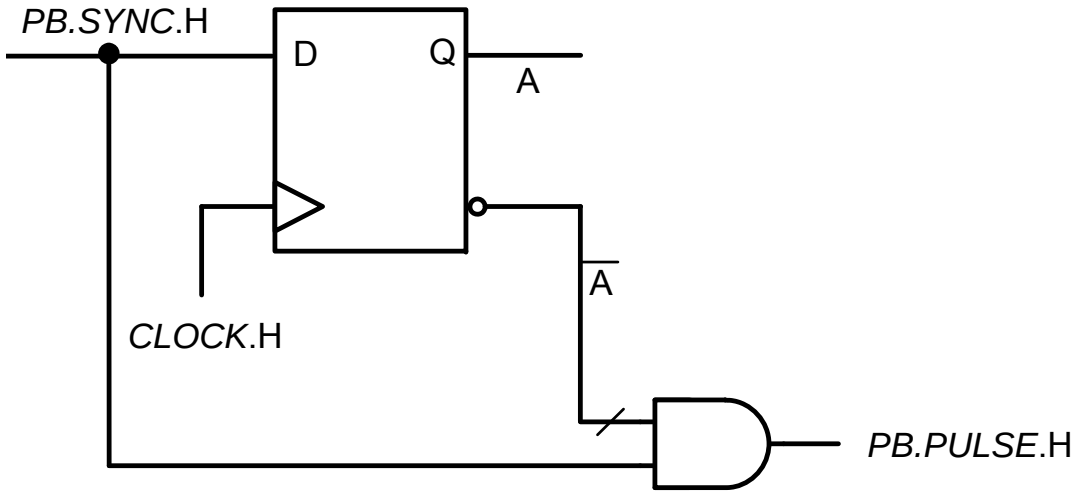
MUX method seems like a reasonable way – need one D FF and one 2-input MUX.



→ No need for MUX!

Design example1 : Single pulser circuit.

Complete single pulser circuit with synchronizing D FF :

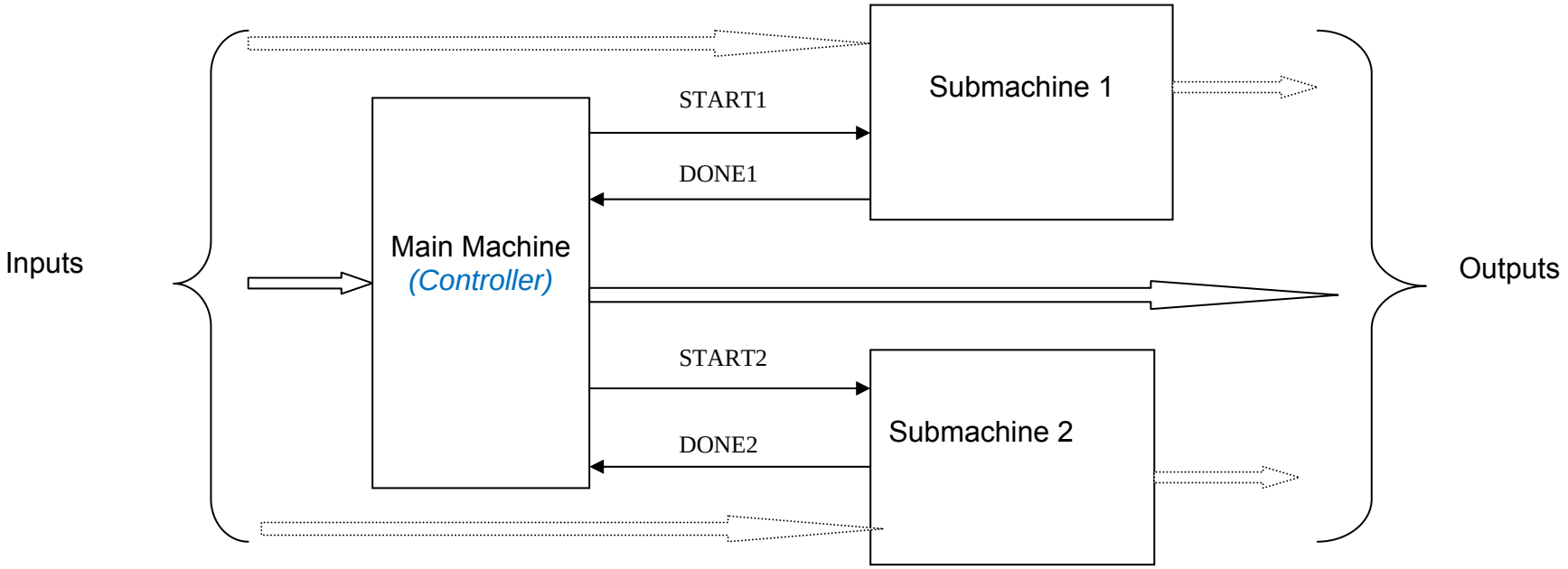


A single-pulser circuit

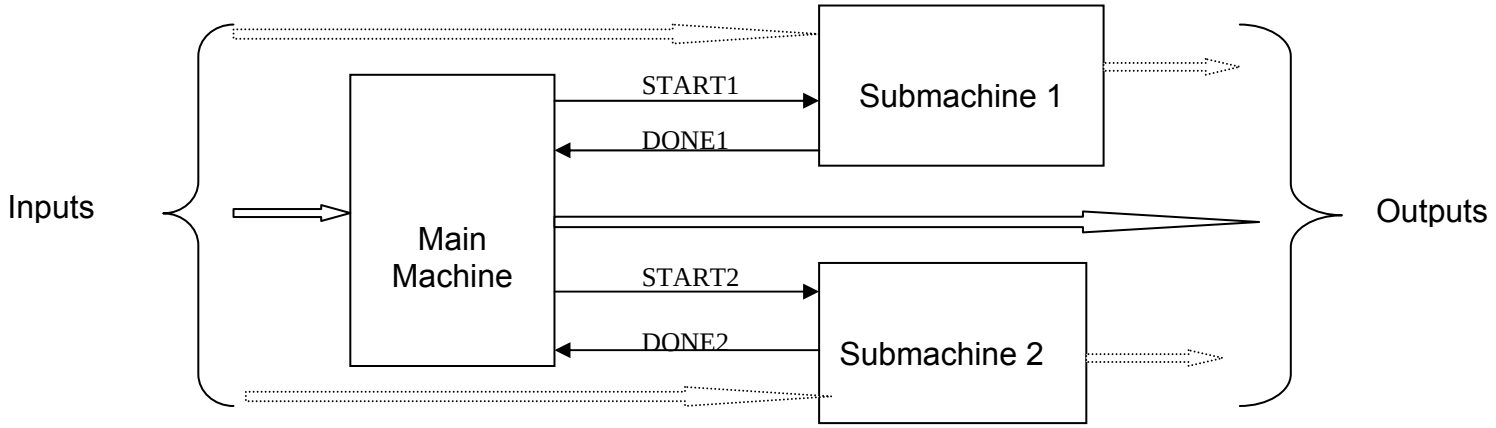
ASM Synthesis: *Modular design of state machines*

Modular design of state machines

- Large state machines are **difficult** to *conceptualize*, *design* and *debug*.
- Hence, useful to solve problem with a collection of **smaller state machines**.



ASM Synthesis: *Modular design of state machines*



- **Main machine** executes a main algorithm to **control** the **submachines** & get the job done. Sends command signals to **submachines** and gets feedback signals from **submachines**.
- **Submachines** respond to external inputs as well as commands from **main machine**. Can give outputs as well as feedback to the **main machine**.
- Common examples of **submachines** are *counters*, *shift registers*, etc.
- Sometimes the **main machine** is called the ***controller*** and the **submachines** are called ***controlled circuit elements*** or ***architectural elements***.
- The trick here is to do the **modularization** appropriately, and pick natural components for the submachines. This can greatly simplify design problems.

Modeling State Machines Using VHDL

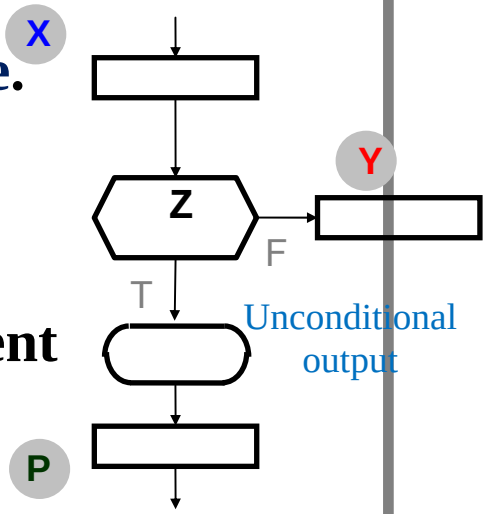
State machines modeling using VHDL

- **State Machine triggered at *rising* or *falling* edge by:**

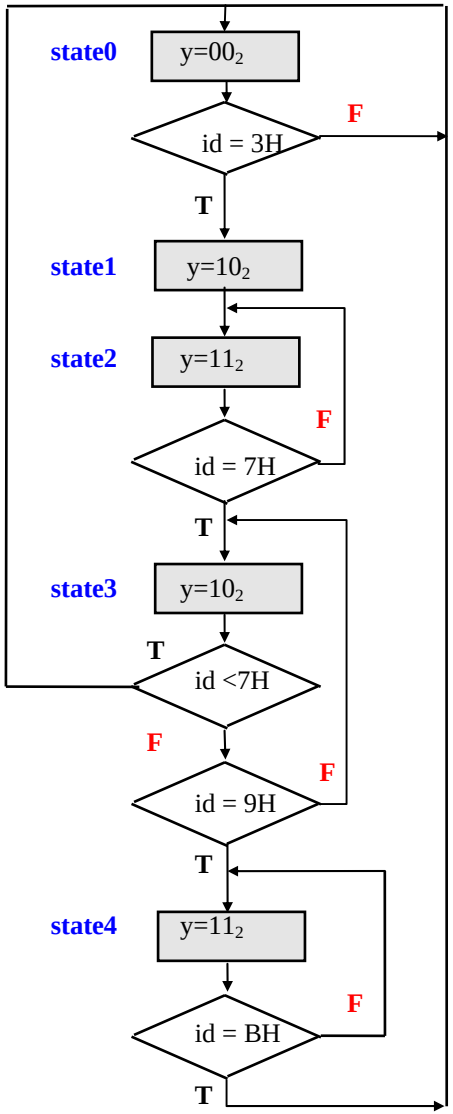
```
clk'event and clk='1'   -- rising edge triggered
clk'event and clk='0'   -- falling edge triggered
                  OR
rising_edge(clk)        -- function
```

Behavioral Modeling

- **Modeled using `case` statement in a `process`.**
- **Branches of `case` contain behavior for each state.**
- **State information is stored in a signal.**
- **Conditional check implemented using `if` statement**

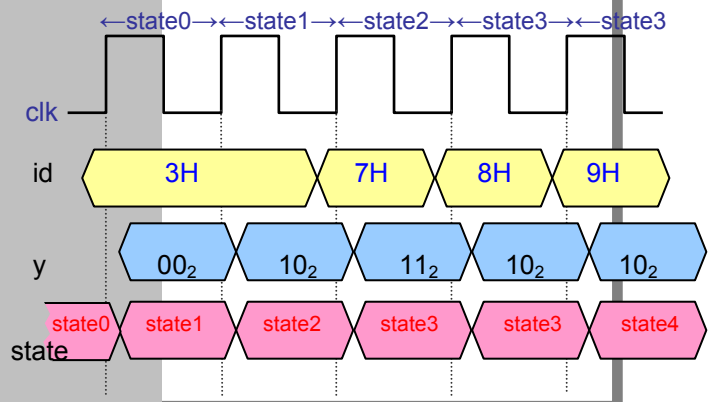


State Machine m_1 based on ASM chart

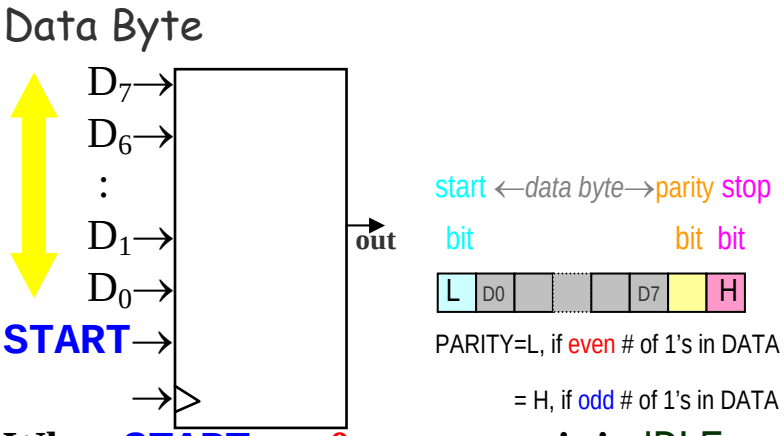



```
entity state_m_1 is
        port(clk, rst:in std_logic; id: in std_logic_vector(3 downto 0);
             y: out std_logic _vector(1 downto O));
end state_m_1;
architecture archstate_m_1 of state_m_1 is
        type states is (state0, state1, state2, state3, state4);
        signal state: states;
begin
        process (clk, rst)
          begin
          if rst='1' then state <= state0;   -- asynchronous reset
                        y <= "00";
          elsif (clk'event and clk='l') then
                case state is
                 when state0 =>
                        y <= "00";
                        if id = x"3" then state <= state1;
                        end if;
                 when state1 =>
                        y <= "10";
                        state <= state2;
                 when state2 =>
                        y <= "11";
                        if id = x"7" then state <= state3;
                        end if;
                 when state3 =>
                        y <= "10";
                        if id < x"7" then state <= state0;
                        elsif id = x"9" then
                                state <= state4;
                        end if;
                 when state4 =>
                        y <= "11";
                        if id = x"b" then state <= state0;
                        end if;
                end case;
            end if;
           end process;
end archstate_m_1;
```

- **states is an enumerated type**
- **state machine is described in a `process`**
- **outputs are specified for *every* state and *next state* determined in *parallel***
- **`CASE` contains all of state transitions**
- ***example* demonstrates *algorithmic* & *intuitive* fashion that VHDL permits in the description of state machines.**
- **A *timing diagram* is shown below:**



State Machine *Example*: Transmission of serial data



When START → 0, converter is in IDLE state & gives a HIGH output.

When START → 1, DATA is loaded into converter & converter must serially output a bit stream as above.

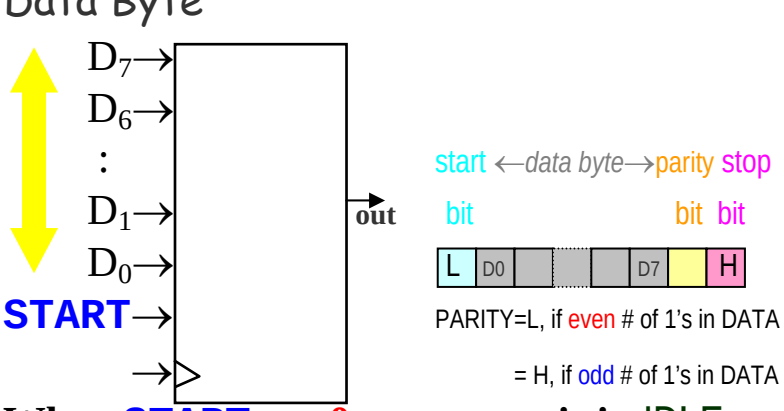
In *VHDL modeling of state machines*, model the *entire system* in *VHDL* which integrates use of "architectural elements"/

STEP#1→ Draw flowchart (or ASM Chart) that precisely describes the operation of the state machine.

STEP#2→ Translate to *VHDL* program

STEP#3→ Check simulation. Revise flowchart & *VHDL* program until simulation is correct!

State Machine *Example*: Transmission of serial data



When START → 0, converter is in IDLE state & gives a HIGH output.

When START → 1, DATA is loaded into converter & converter must serially output a bit stream as above starting with START-BIT.

In *VHDL modeling of state machines*, model the *entire system* in *VHDL* which integrates use of "architectural elements"

STEP#1→ Draw flowchart (or *ASM Chart*) that precisely describes the operation of the state machine.

STEP#2→ Translate to *VHDL* program

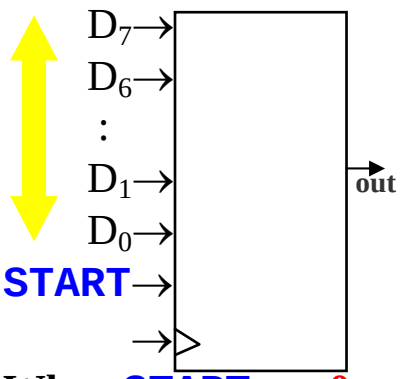
STEP#3→ Check simulation. Revise flowchart & *VHDL* program until simulation is correct!

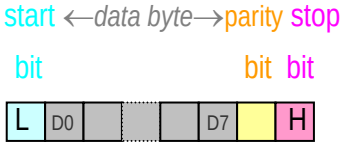
Architectural elements:

1. **Shift Register** to load & shift the input data bits
2. **Counter** to keep track of number of bits being *shifted*.
3. **Generate Parity**.

State Machine *Example*: Transmission of serial data

DATA Byte





PARITY=L, if even # of 1's in DATA

= H, if odd # of 1's in DATA

When START → 0, converter is in IDLE state & gives a HIGH output.

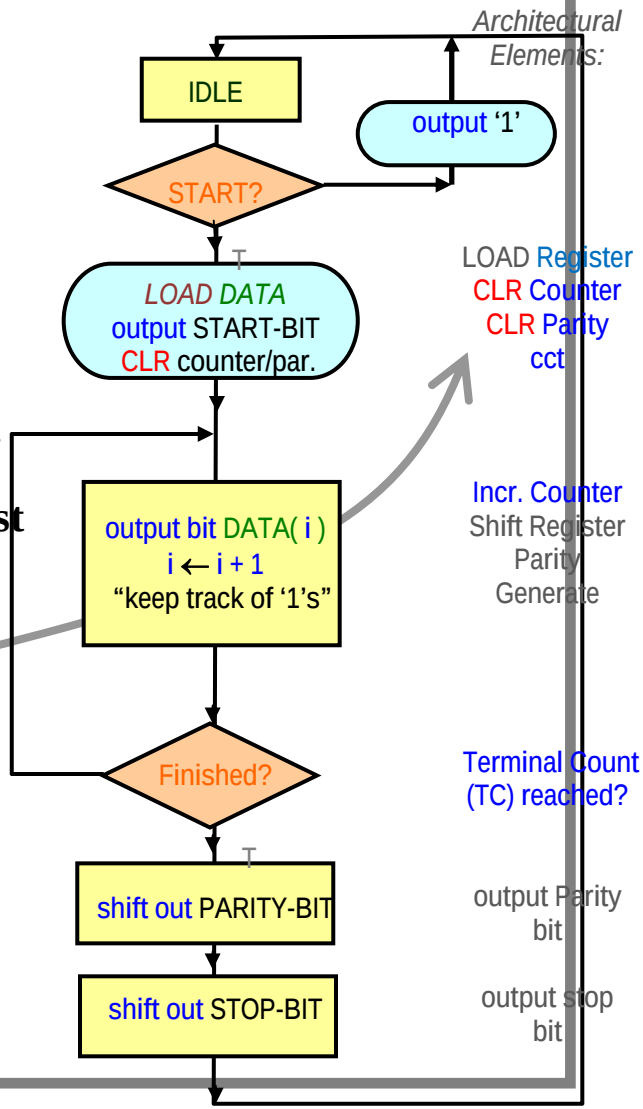
When START → 1, DATA is loaded into converter & converter must serially output a bit stream as above starting with START-BIT.

In *VHDL modeling of state machines*, model the *entire system* in *VHDL* which integrates use of "architectural elements".

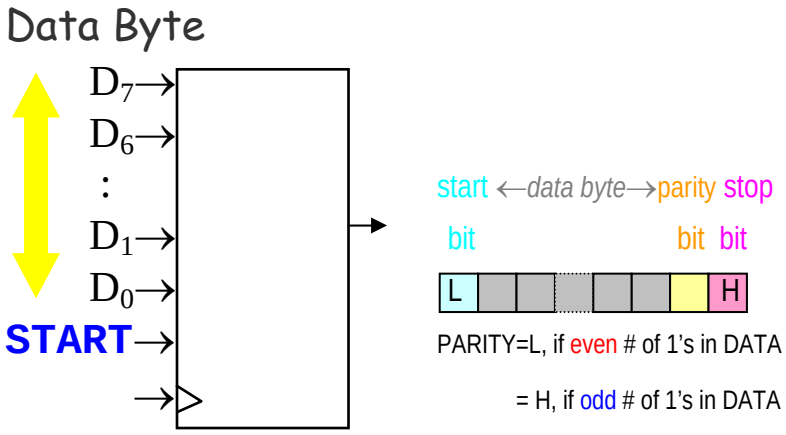
STEP#1→ Draw flowchart (or *ASM Chart*) that precisely describes the operation of the state machine.

STEP#2→ Translate to *VHDL* program

STEP#3→ Check simulation. Revise flowchart & *VHDL* program until simulation is correct!



State Machine *Example*: Transmission of serial data



When START → 0, converter is in an IDLE state & gives a HIGH output

When START → 1, DATA is loaded into converter & converter must serially output a bit stream as above.

LSB MSB

databyte: 1→1→1→1→0→0→0→0
par 0→1→0→1→0→0→0→0→0

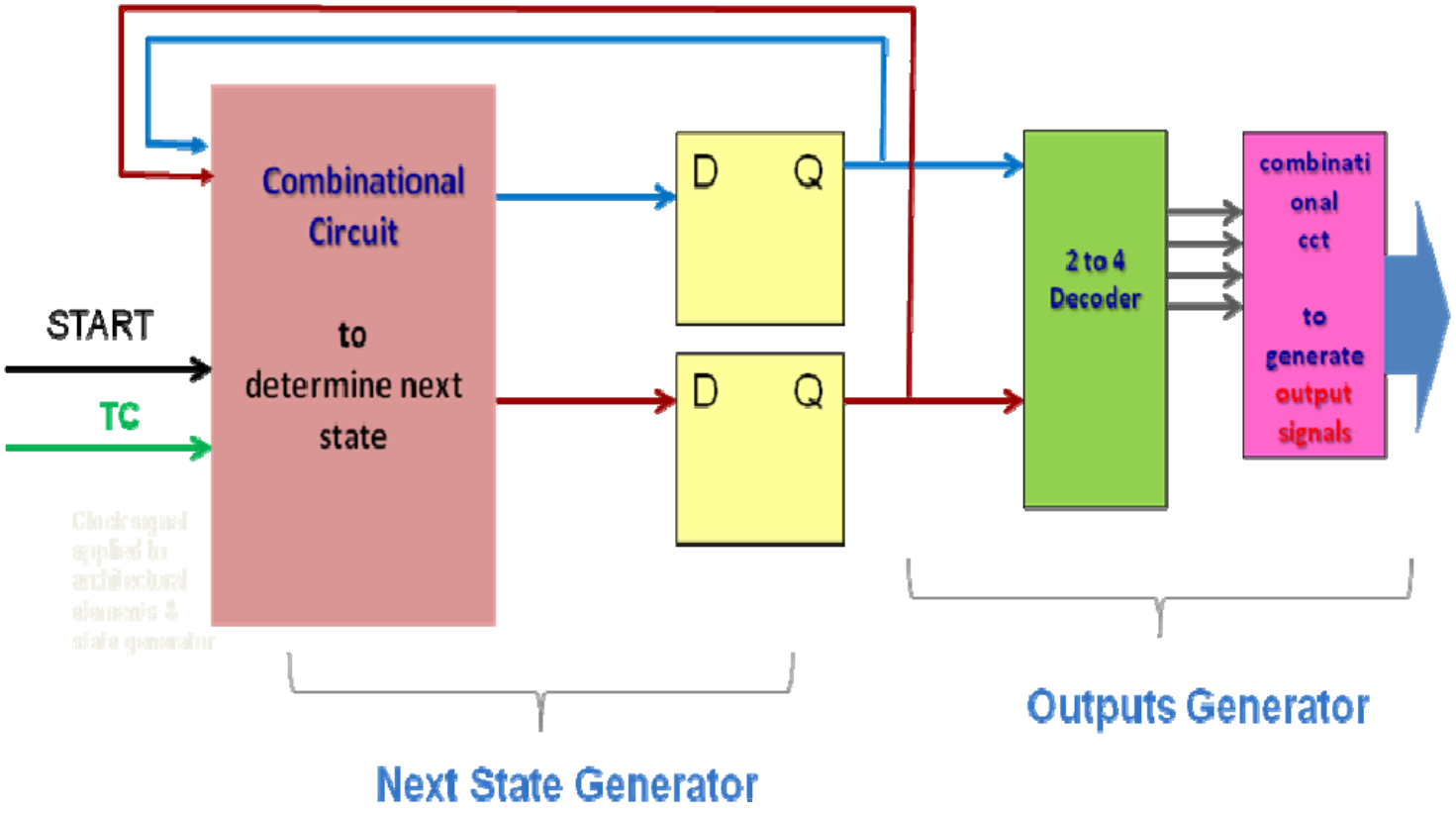
databyte: 0→1→1→0→0→0→0→1→0
par 0→0→1→0→0→0→0→1→1

```
use ieee.std_logic_1164.ALL;
use ieee.std_logic_arith.ALL;
entity serial_sft is
   port( clk, rst, start :in std_logic; data: in std_logic_vector(7 downto 0);
         shift_out: out std_logic);
end serial_sft;

architecture state_m of serial_sft is
          type states is ( idle, shift,  parity, stop );
          signal state: states;
begin
  process (clk, rst)
  variable tmp: std_logic_vector(7 downto 0);
  variable par: std_logic;
  variable i : std_logic_vector (2 downto 0);
  begin
   if rst='1' then state <= idle;  shift_out <= '1';-- asynchronous reset
   elsif (clk'event and clk='l') then
      case state is
           when idle =>                              -- STATE0
             if start= '1' then                        -- if START = '1'
               tmp := data; shift_out<='0';          -- load DATA, out start bit
               state <= shift;  par :='0'; i:="000";  -- clear parity bit & cntr
             else shift_out<='1'; end if;             -- if START = '0'
           when shift=>                              -- STATE1
                    shift_out   <= tmp( i );         -- output DATA( I )
                    par := par xor tmp( i );         -- parity bit update
                    i := i + 1;                      -- increment counter
                    if i = "000" then state <= parity;
                    end if;
           when parity => shift_out<=par; state<=stop;   -- STATE2
           when stop   => shift_out<='1'; state<=idle;   -- STATE3
           end case;
   end if;
 end process;
end state_m;
```

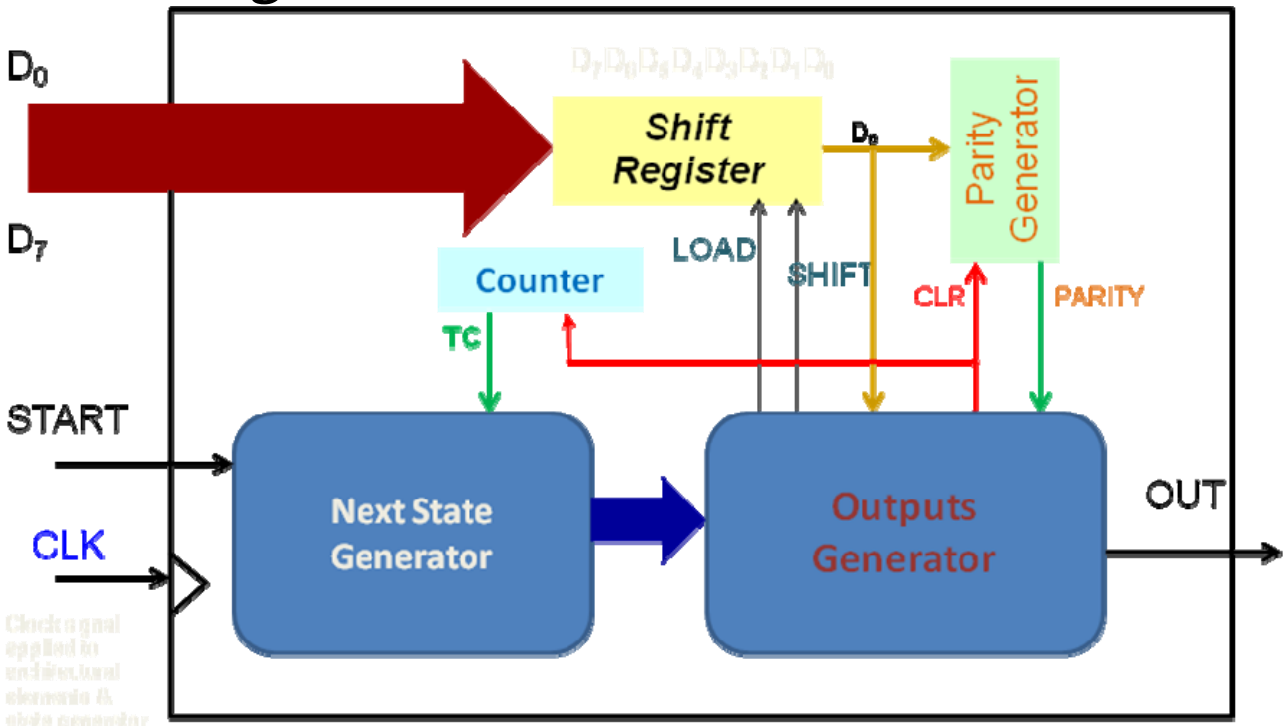
State Machine *Example*: Transmission of serial data

Modular design of this state machine:



State Machine *Example*: Transmission of serial data

Modular design of this state machine: *(showing the arch. elements)*



Shift Register
Loads DATA when LOAD is true
Shifts Right when SHIFT is true

COUNTER (3-bit)
Cleared when CLR is true
When Terminal Count : TC is true

PARITY GEN.
TOGGLE Flip-flop
output cleared when CLR is true

For state machines that require *architectural elements*, lets consider *VHDL* realizations next.