Sequential Logic Design

Lecture 6

Outline

- Sequential Logic
- Virtual Input/Output (VIO) Core
- Latch Generation

The outputs of sequential logic depend on both current and prior input values

Sequential logic has memory

Certain previous inputs are explicitly remembered or distilled into the state of the system, a set of bits called state variables that contain all the information about the past necessary to explain the future behavior of the circuit.

Simple Sequential Circuits, i.e. Latches and Flip-Flops

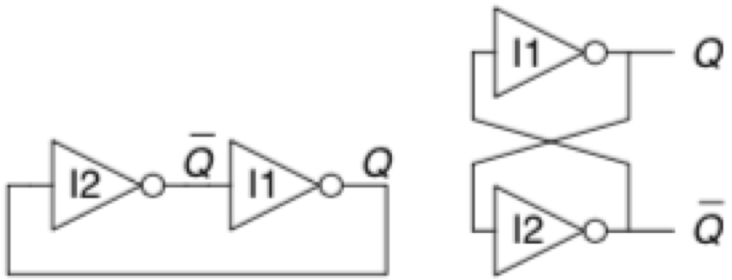
Synchronous Sequential Circuits

Finite State Machines

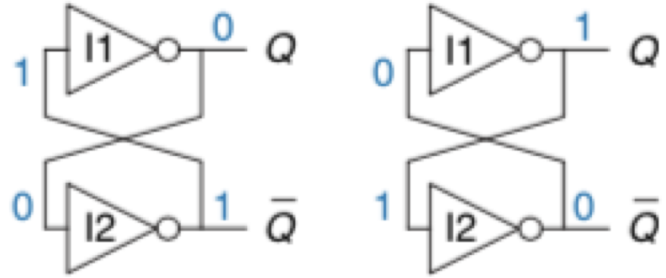
Analyze the Speed and Discuss Parallelism

Fundamental building block of memory is a bistable element, an element with two stable states

i.e. Cross-coupled inverter pair, it is cyclic: Q depends on $\bar{Q}$, vice versa



Two stable states Q = 0 and Q = 1

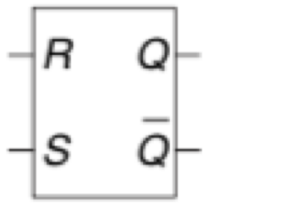


An element with N stable states conveys $\log_2 N$ bits of information, so a bistable element stores one bit

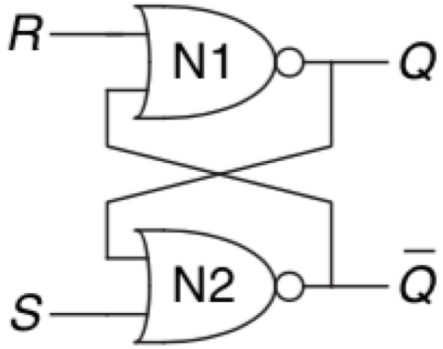
Other bistable elements, such as latches and flip-flops, provide inputs to control the value of the state variable

SR Latch

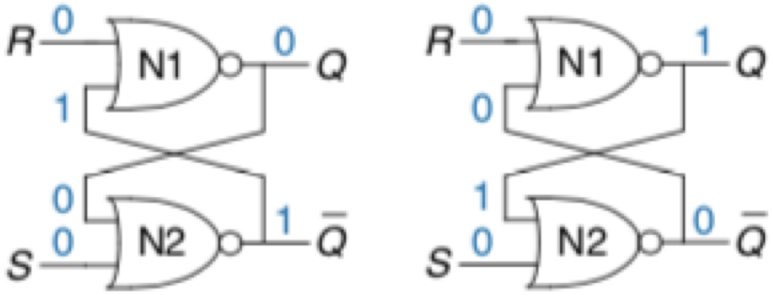
Its state can be controlled through the S and R inputs, which set and reset the output Q.



SR latch symbol



SR latch schematic



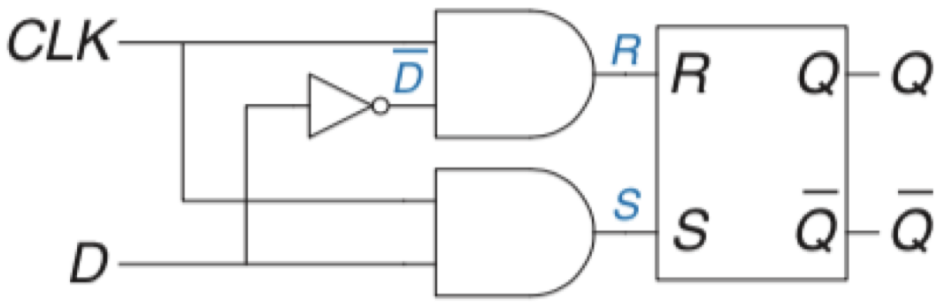
Case	S	R	Q	$\bar{Q}$
IV	0	0	Q_{prev}	$\bar{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

SR latch truth table

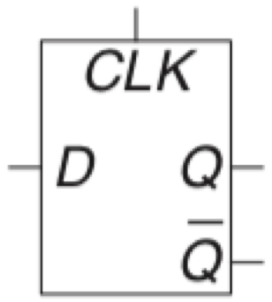
Combines what and when

Produces $Q = \bar{Q} = 0$

D Latch



CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X	$\overline{X}$	0	0	Q_{prev}	$\overline{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0



CLK = 1, the latch is transparent

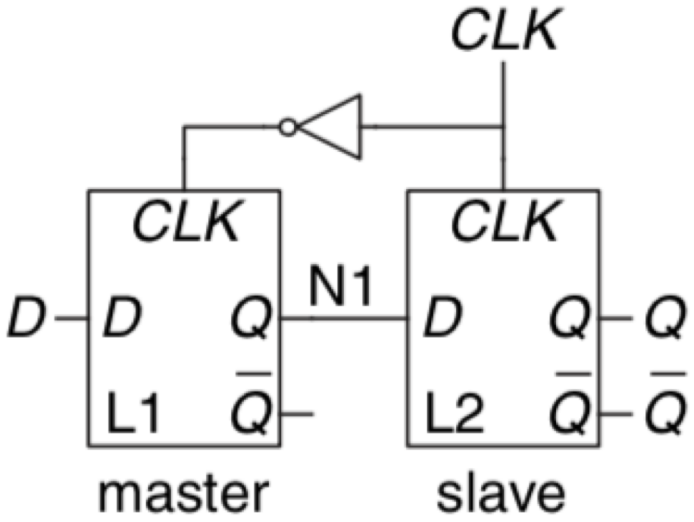
CLK=0, the latch is opaque

- Data input, D, controls what the next state should be.
- The clock input, CLK, controls when the state should change.
- In all cases, $\overline{Q}$ is the complement of Q.
- The D latch avoids the case of simultaneously asserted R and S inputs.

The D latch updates its state continuously while CLK = 1.

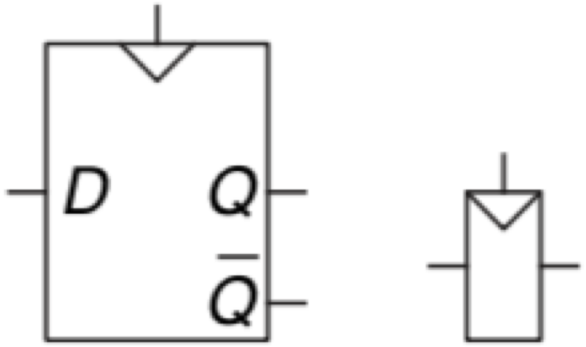
D Flip-Flop

Two back-to-back D latches controlled by complementary clocks

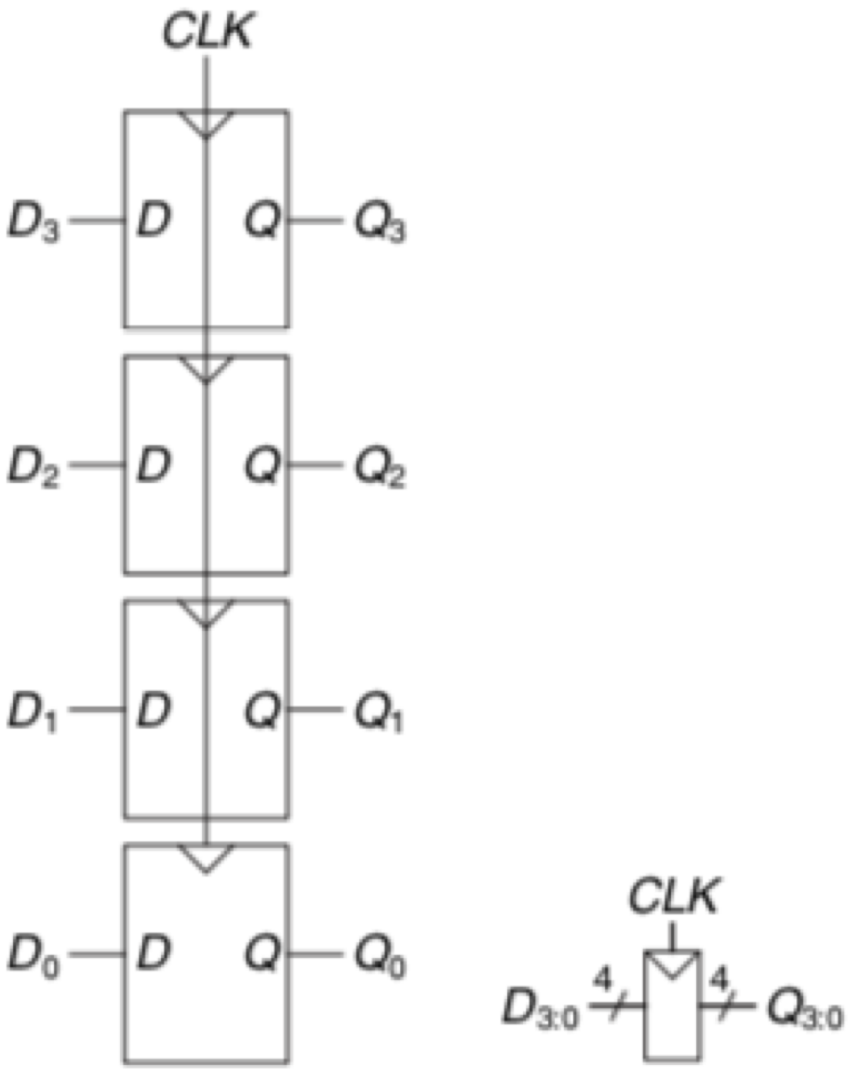


- CLK = 0, the master latch is transparent and the slave is opaque.
  value at D propagates through to N1.
- CLK = 1, the master goes opaque and the slave becomes transparent.
  value at N1 propagates through to Q, but N1 is cut off from D.

D flip-flop copies D to Q on the rising edge of the clock, and remembers its state at all other times.



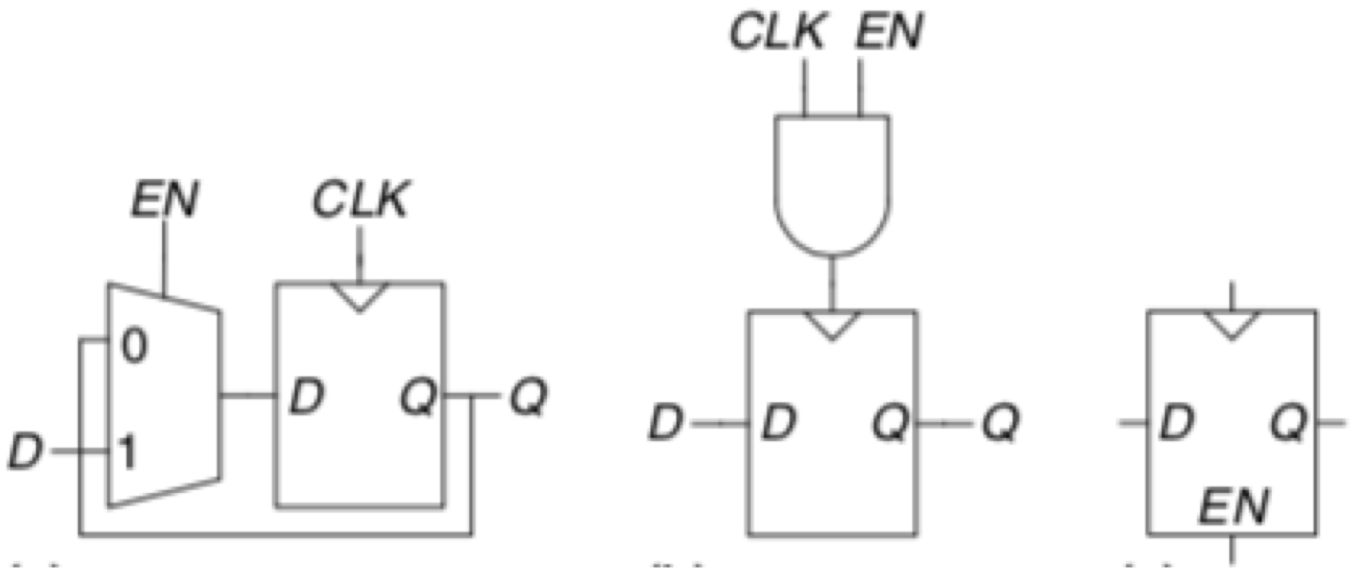
Register



An N-bit register is a bank of N flip-flops that share a common CLK input, so that all bits of the register are updated at the same time.

Enabled Flip-Flop

When EN is FALSE, the enabled flip-flop ignores the clock and retains its state

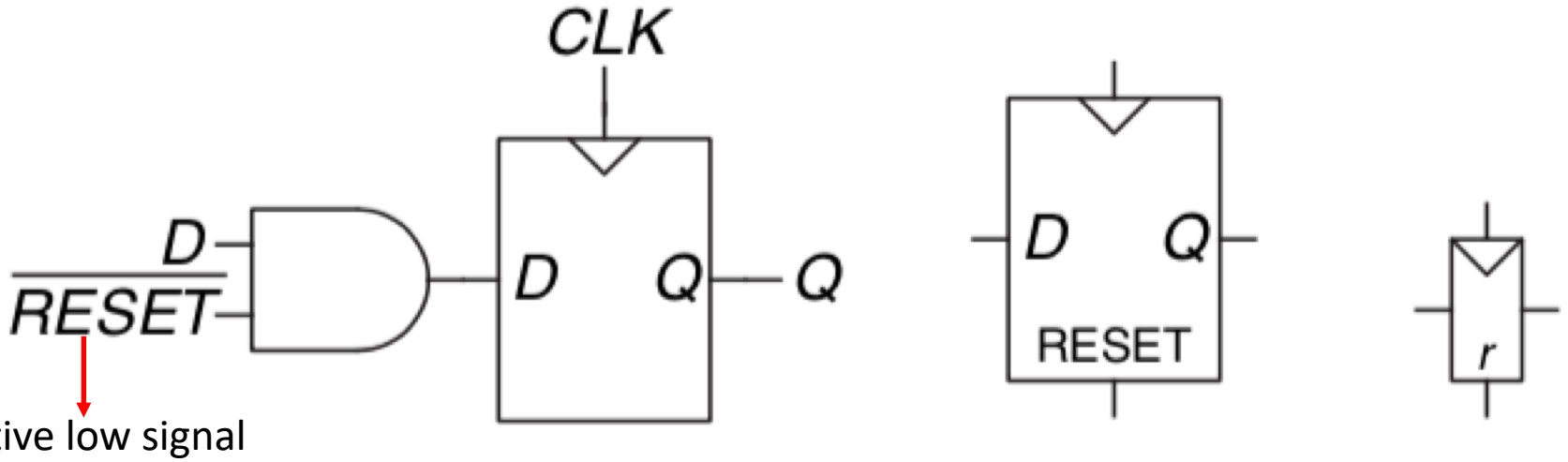


Resettable Flip-Flop

- When RESET is FALSE: the flip-flop behaves like an ordinary D flip-flop.
- When RESET is TRUE: the flip-flop ignores D and resets the output to 0
- May be synchronously or asynchronously resettable

Reset only on the rising edge of CLK

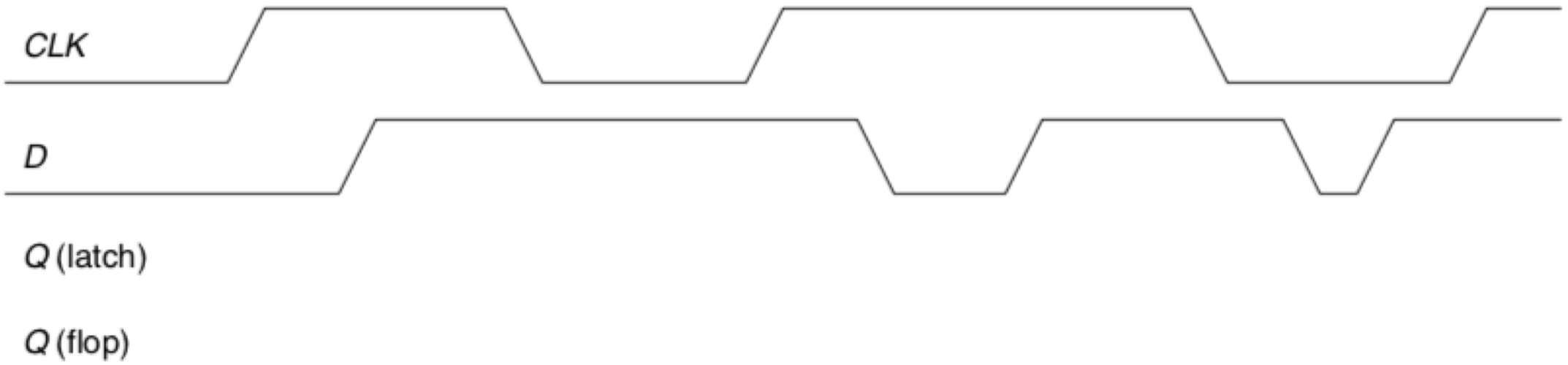
Reset as soon as RESET becomes TRUE, independent of CLK



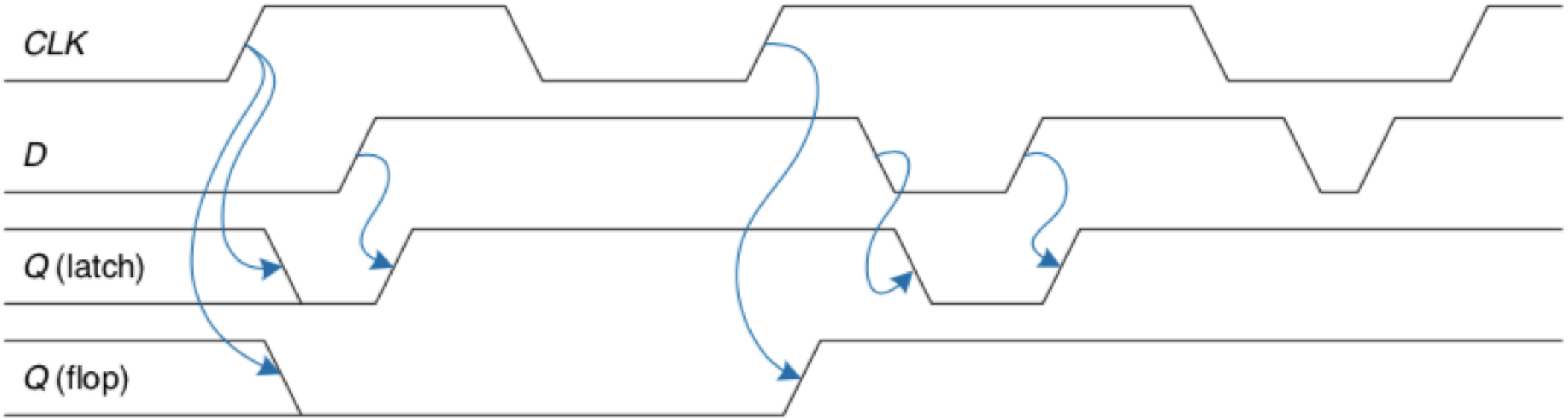
active low signal

Example

The D and CLK inputs shown are applied to a D latch and a D flip-flop. Determine the output, Q, of each device.

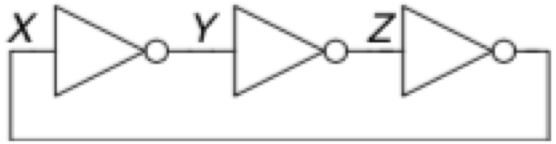


Solution



Example

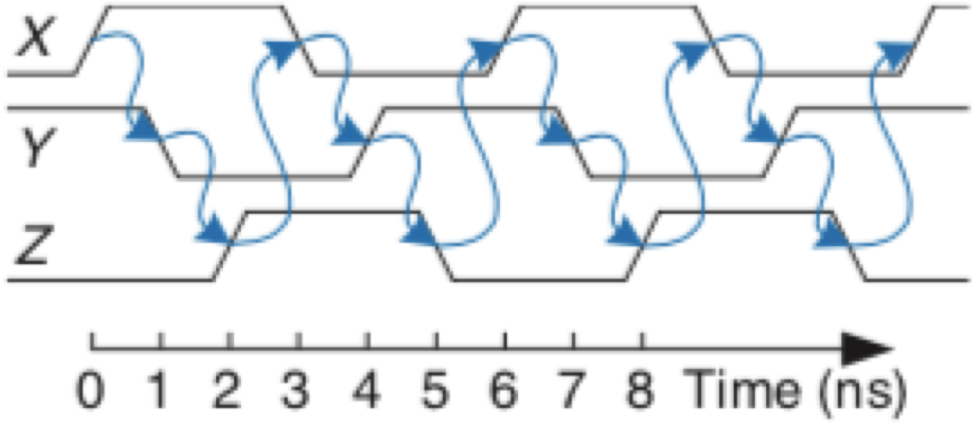
Consider the following 3 inverter loop, Each inverter has a propagation delay of 1 ns. Determine what the circuit does.



Solution

- The circuit has no stable states and is said to be unstable or astable. Each node oscillates between 0 and 1 with a period of 6 ns

Ring Oscillator Circuit

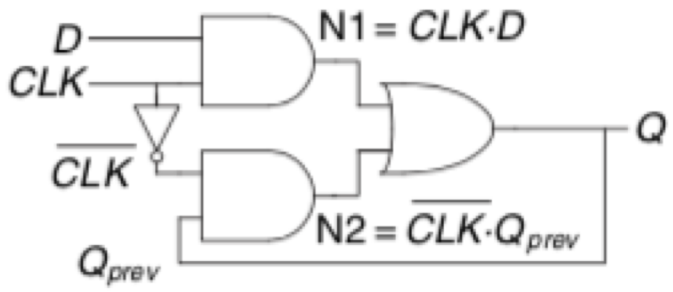


Example- Issues with Asynchronous Design

Consider the asynchronous D latch design. Suppose the delay through the inverter from *CLK* to $\overline{CLK}$ is 2 ns and the delays of the AND and OR gates are 1 ns. Assume initially CLK = D = 1. Obtain the waveforms when CLK falls

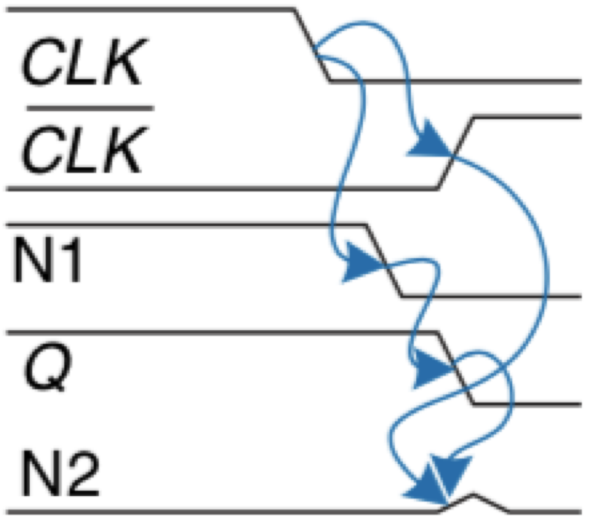
CLK	*D*	Q_{prev}	*Q*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$$



Solution

The latch should remember its old value, keeping Q = 1.
Nodes N1 and Q may both fall before CLK rises. In such a case, N2 will never rise, and Q becomes stuck at 0.



Delay depends on how the inverter was manufactured, the power supply voltage, and temperatures.

the circuit has a race condition that causes it to fail

Synchronous Logic Design

Sequential circuits with cyclic paths can have undesirable races or unstable behavior.

To avoid these problems:

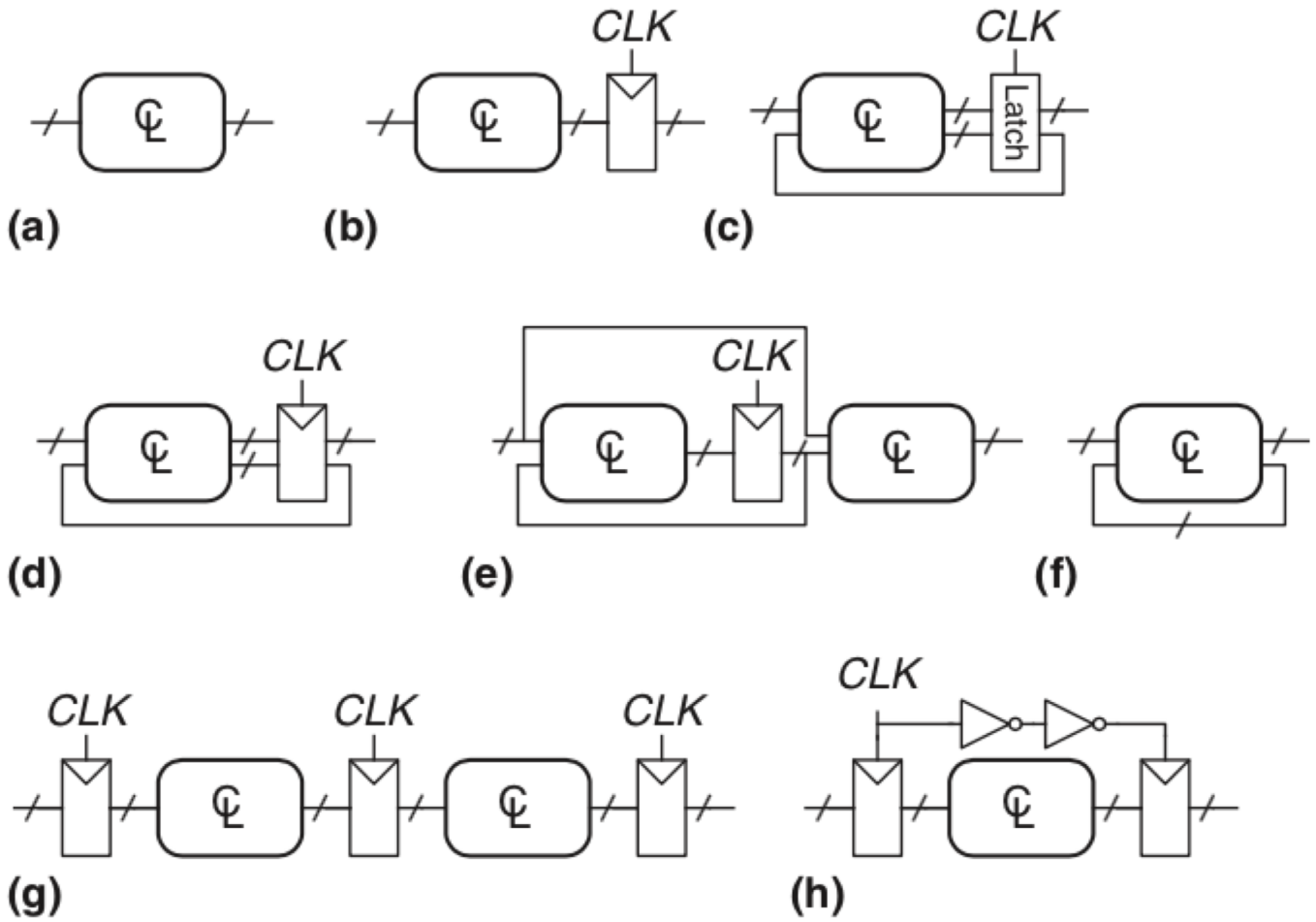
- Break the cyclic paths by inserting registers in the feedback path which contain the state of the system that are synchronized to the clock

Synchronous sequential circuits consists of interconnected circuit elements such that:

- Every circuit element is either a register or a combinational circuit
- At least one circuit element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one register.

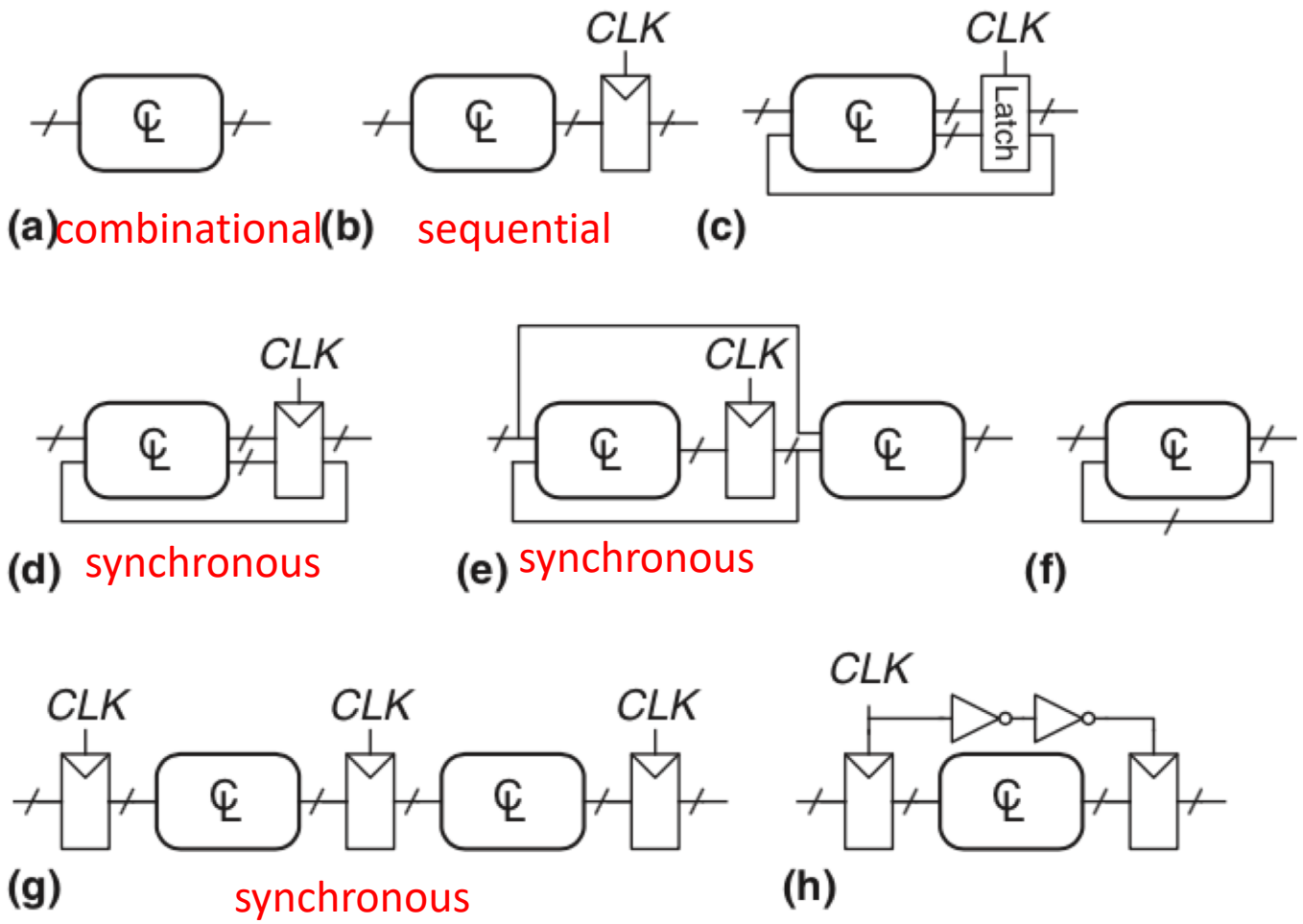
Example

Which of the circuits are synchronous sequential circuits?



Solution

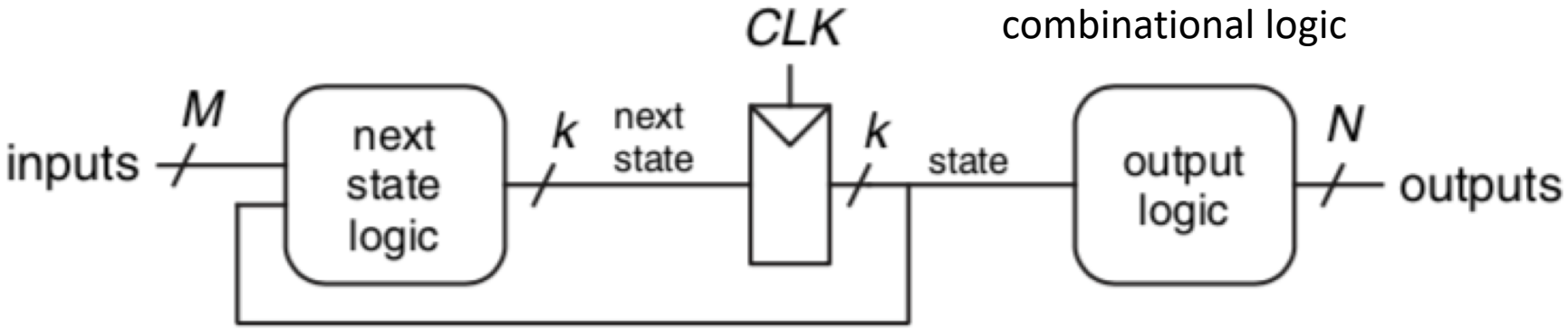
Which of the circuits are synchronous sequential circuits?



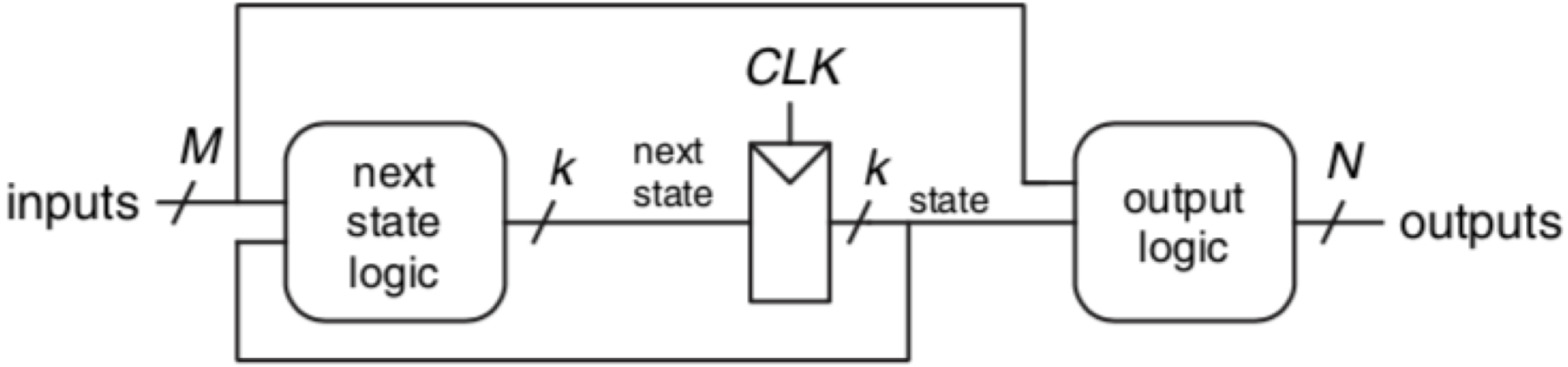
Finite State Machines

provide a systematic way to design synchronous sequential circuits given a functional specification

A circuit with k registers can be in one of a finite number (2^K) of unique states



(a) Moore machine the outputs depend only on the current state of the machine

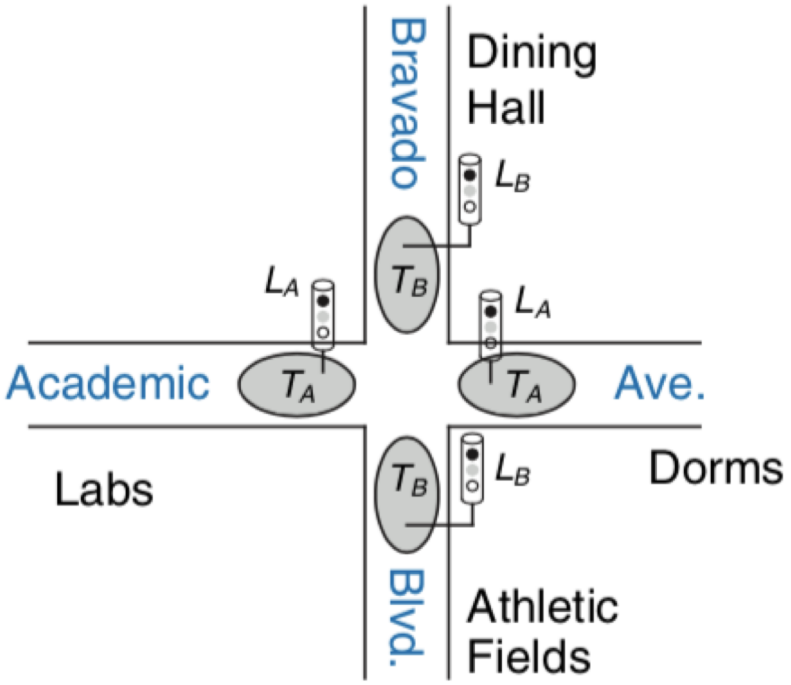


(b) Mealy machine the outputs depend on both the current state and the current inputs

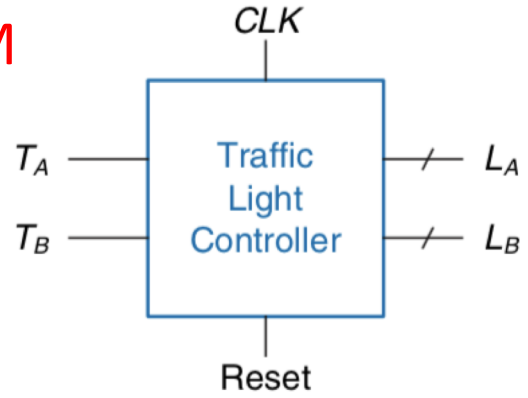
State Transition Diagram

- Circles represent states and
- Arcs represent transitions between states.
- The transitions take place on the rising edge of the clock.

Design Example



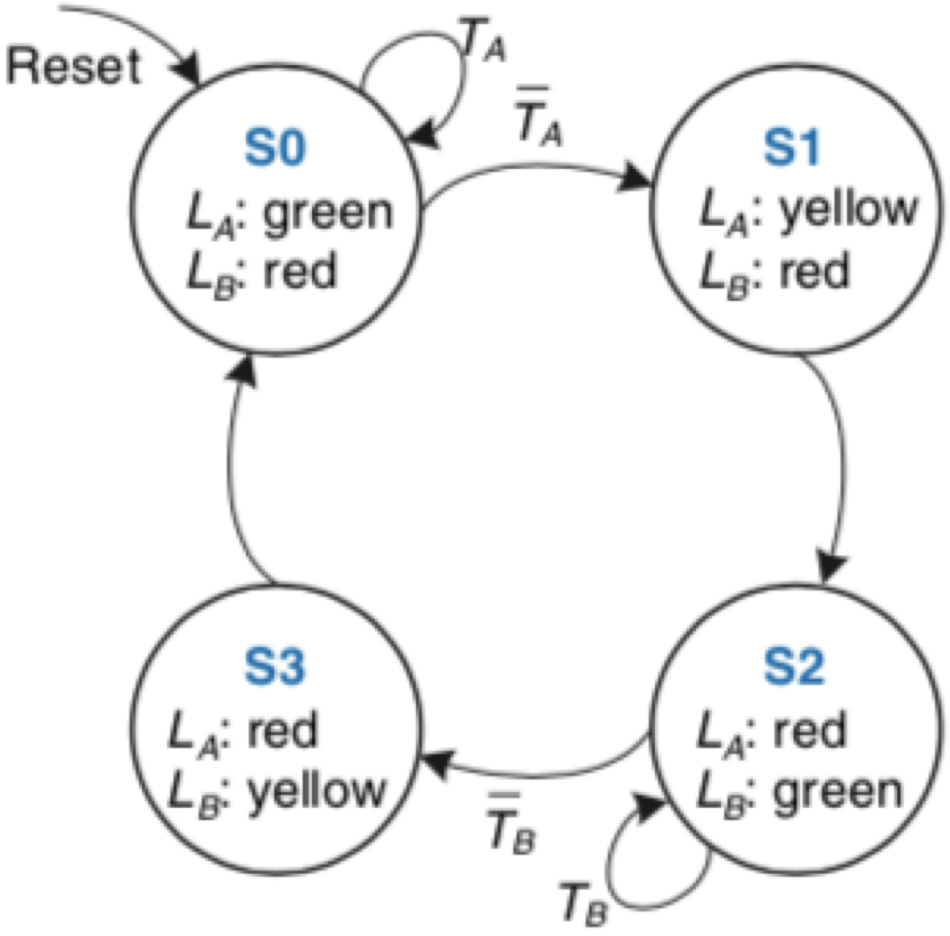
FSM



- Install two traffic sensors, TA and TB, on Academic Ave. and Bravado Blvd., respectively. Each sensor indicates TRUE if students are present and FALSE if the street is empty.
- Install two traffic lights, LA and LB, to control traffic. Each light receives digital inputs specifying whether it should be green, yellow, or red.
- Hence, the FSM has two inputs, TA and TB, and two outputs, LA and LB.
- Provide a clock with a 5-second period. On each clock tick (rising edge), the lights may change based on the traffic sensors.
- Provide a reset button so that the controller can be put in a known initial state when it first turns on.

Sketch the state transition diagram

State Transition Diagram



It is abstract: uses states labeled {S0, S1, S2, S3} and outputs labeled {red, yellow, green}

when in state S0, the system will remain in that state if TA is TRUE and move to S1 if TA is FALSE

when in state S1, the system will always move to S2

while in state S2, LA is red and LB is green

Create a State Transition Table describing the next state as a function of the inputs and current states. Obtain the Boolean expressions.

State Transition Table

Indicates, for each current state and input, what the next state, S', should be.

- To build a circuit the states and outputs must be assigned binary encodings

  i.e. S1:0, LA1:0, and LB1:0

Table 3.1 State transition table

Current State S	Inputs T_A	Inputs T_B	Next State S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Table 3.2 State encoding

State	Encoding $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Table 3.3 Output encoding

Output	Encoding $L_{1:0}$
green	00
yellow	01
red	10

Table 3.4 State transition table with binary encodings

Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S_1'	S_0'
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Boolean equations sum-of-products form:

$$S_1' = \overline{S}_1 S_0 + S_1 \overline{S}_0 \overline{T}_B + S_1 \overline{S}_0 T_B$$

$$S_0' = \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B$$

Simplify

$$S_1' = S_1 \oplus S_0$$

$$S_0' = \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B$$

Output Table

Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Boolean expressions

$$L_{A1} = S_1$$

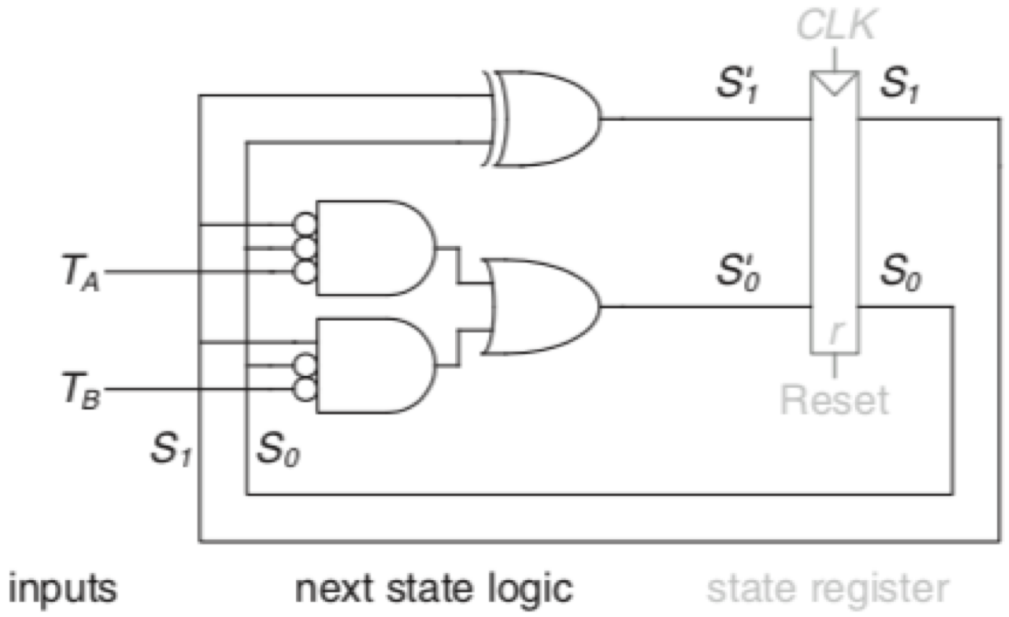
$$L_{A0} = \overline{S}_1 S_0$$

$$L_{B1} = \overline{S}_1$$

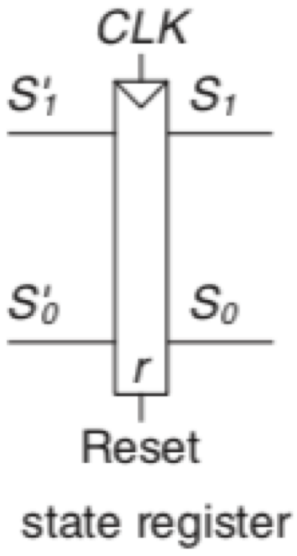
$$L_{B0} = S_1 S_0$$

Sketch the Moore FSM

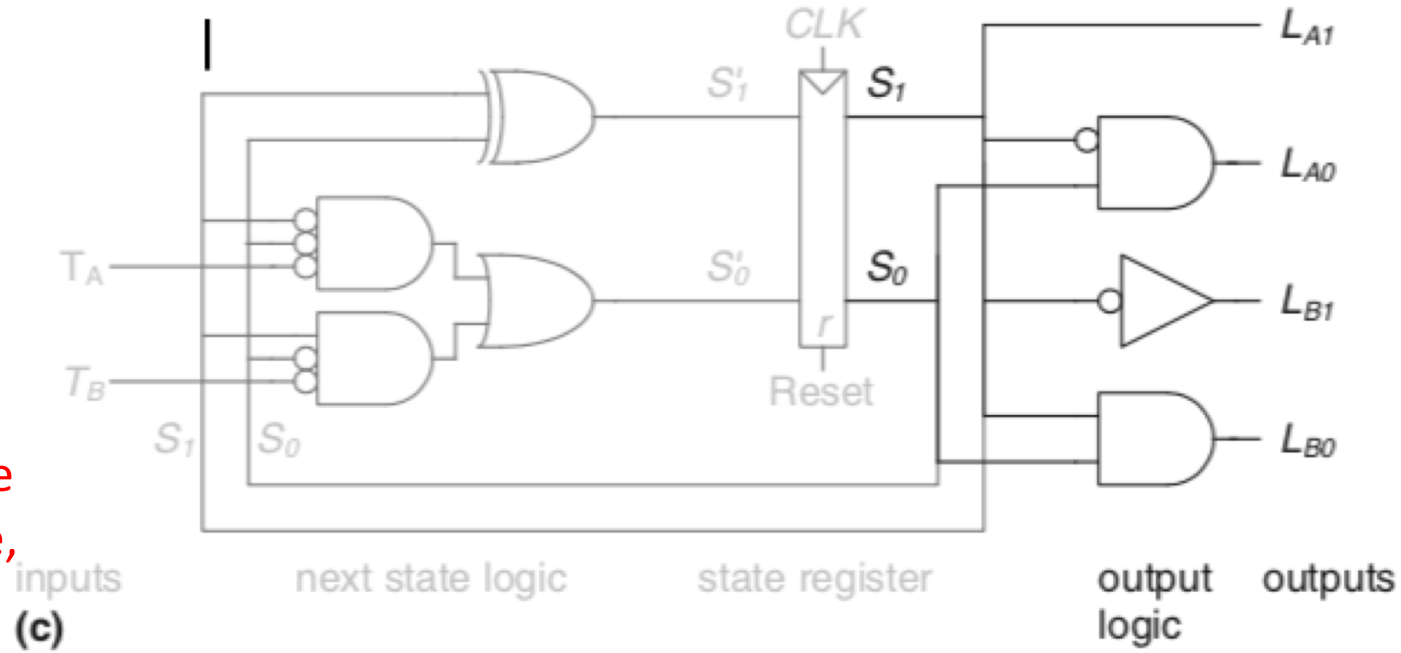
Moore FSM



Computes the next state from the current state and inputs

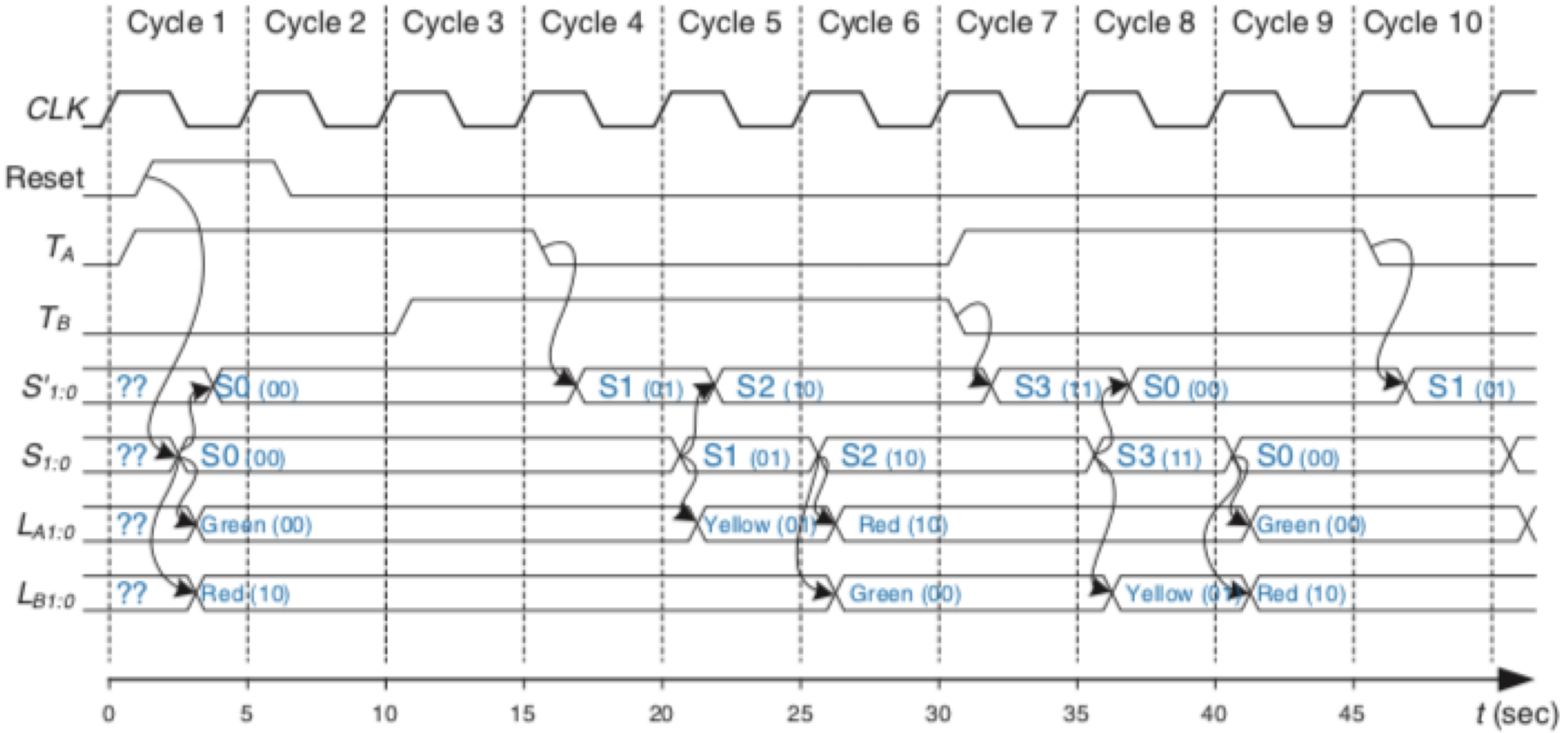


On each clock edge, the state register copies the next state, S1':0, to become the state S1:0



Computes the outputs from the current state

Timing Diagram



Arrows indicate causality; for example, changing the state causes the outputs to change, and changing the inputs causes the next state to change.

Dashed lines indicate the rising edges of CLK when the state changes.

Unfactored and Factored State Machines

Study Example 3.8

State Encoding

How to determine the encoding that produces the circuit with the fewest logic gates or the shortest propagation delay?

by inspection!

Choice between binary encoding and one-hot encoding (one bit is “hot” or TRUE at any time)

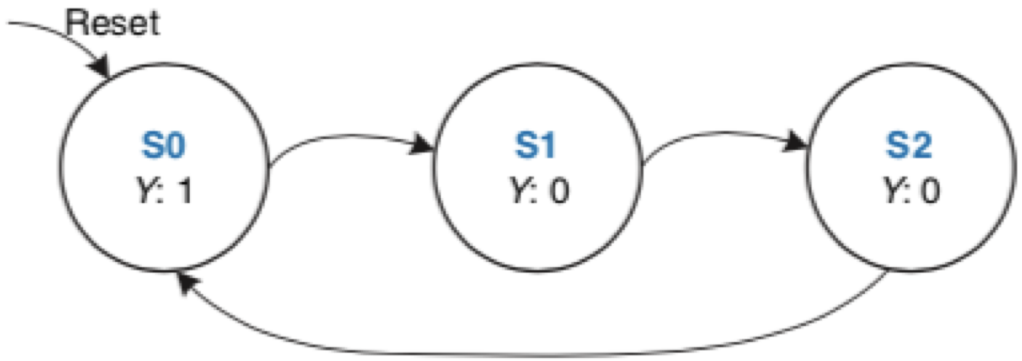
i.e. one-hot encoded FSM with three states would have state encodings of 001, 010, and 100

Each bit of state is stored in a flip-flop, so one- hot encoding requires more flip-flops than binary encoding.
However, with one-hot encoding, the next-state and output logic is often simpler, so fewer gates are required.

Example

A divide-by-N counter has one output and no inputs. The output Y is HIGH for one clock cycle out of every N. In other words, the output divides the frequency of the clock by N. Sketch circuit designs for a divide-by-3 counter using binary and one-hot state encodings.

(a) Waveform



(b) State transition diagram

Solution

State transition table

Current State	Next State
S0	S1
S1	S2
S2	S0

Output table

Current State	Output
S0	1
S1	0
S2	0

$$S_1' = \overline{S}_1 S_0$$
$$S_0' = \overline{S}_1 \overline{S}_0$$
$$Y = \overline{S}_1 \overline{S}_0$$

State transition table with binary encoding

Current State S_1	Current State S_0	Next State S_1'	Next State S_0'
0	0	0	1
0	1	1	0
1	0	0	0

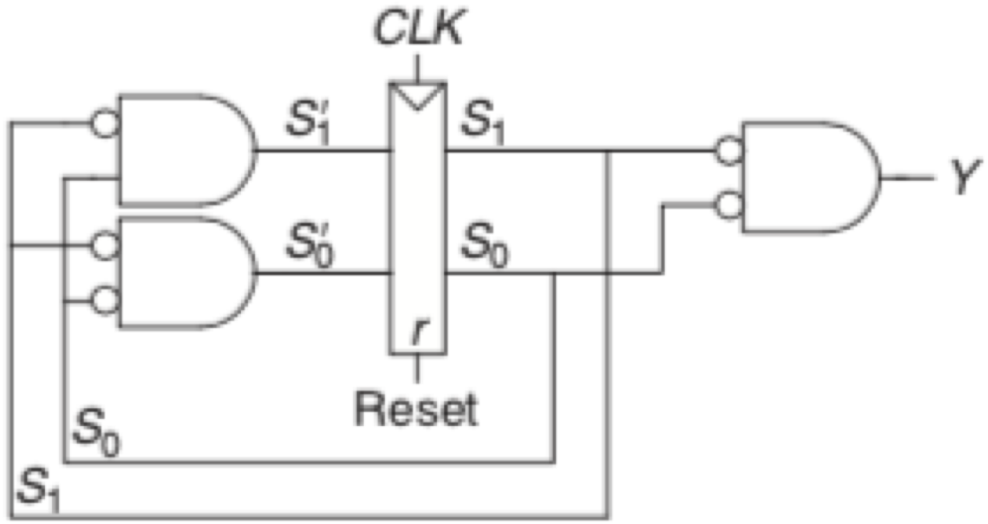
One-hot and binary encodings

State	One-Hot Encoding S_2	One-Hot Encoding S_1	One-Hot Encoding S_0	Binary Encoding S_1	Binary Encoding S_0
S0	0	0	1	0	0
S1	0	1	0	0	1
S2	1	0	0	1	0

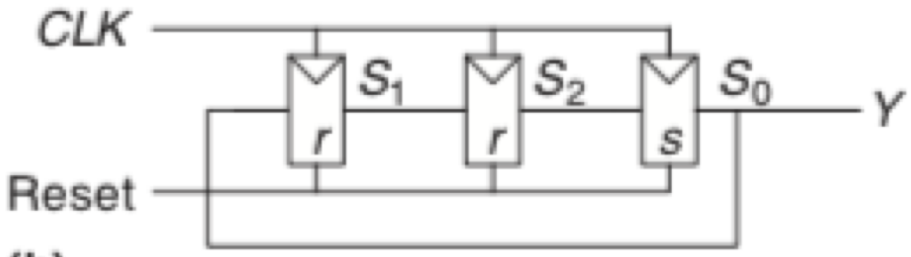
$$S_2' = S_1$$
$$S_1' = S_0$$
$$S_0' = S_2$$
$$Y = S_0$$

State transition table with one-hot encoding

Current State S_2	Current State S_1	Current State S_0	Next State S_2'	Next State S_1'	Next State S_0'
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1



(a) Binary



(b) One-hot encodings

Note: hardware for the binary encoded design could be optimized to share the same gate for Y and S0'

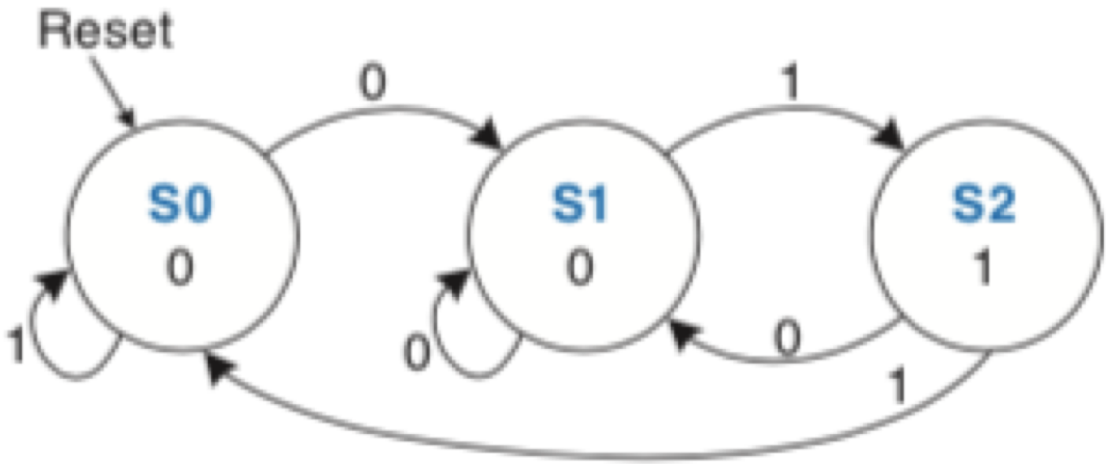
one-hot encoding requires both settable (s) and resettable (r) flip-flops to initialize the machine to S0 on reset

Example-Moore and Mealy Machines

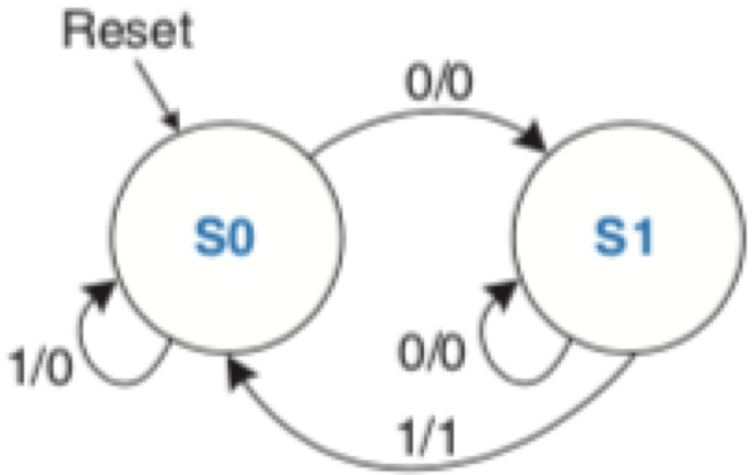
Alyssa P. Hacker owns a pet robotic snail with an FSM brain. The snail crawls from left to right along a paper tape containing a sequence of 1's and 0's. On each clock cycle, the snail crawls to the next bit. The snail smiles when the last two bits that it has crawled over are 01.

- Design the FSM to compute when the snail should smile. The input A is the bit underneath the snail's antennae. The output Y is TRUE when the snail smiles.
- Compare Moore and Mealy state machine designs.
- Sketch a timing diagram for each machine showing the input, states, and output as Alyssa's snail crawls along the sequence 0100110111

State Transition Diagrams



Moore machine



Mealy machine

Each arc is labeled as A/Y

Input Output

State Transition and Output Table

Current State S	Input A	Next State S'
S0	0	S1
S0	1	S0
S1	0	S1
S1	1	S2
S2	0	S1
S2	1	S0

Moore

Current State S	Output Y
S0	0
S1	0
S2	1

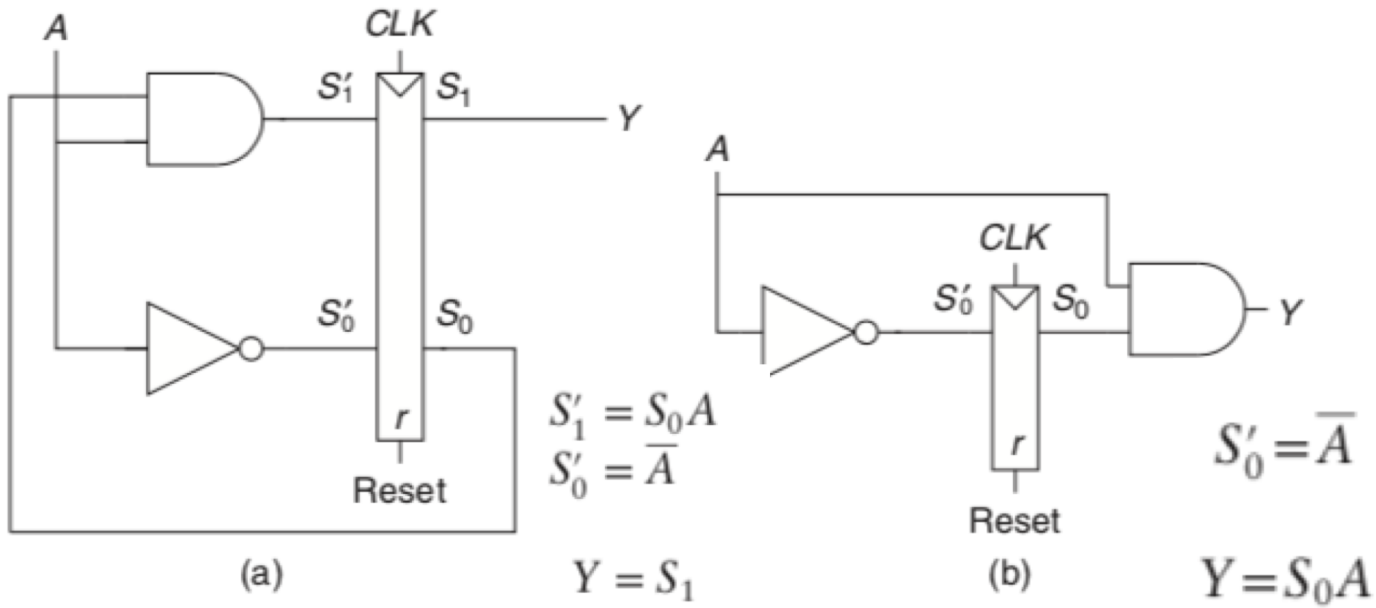
Mealy

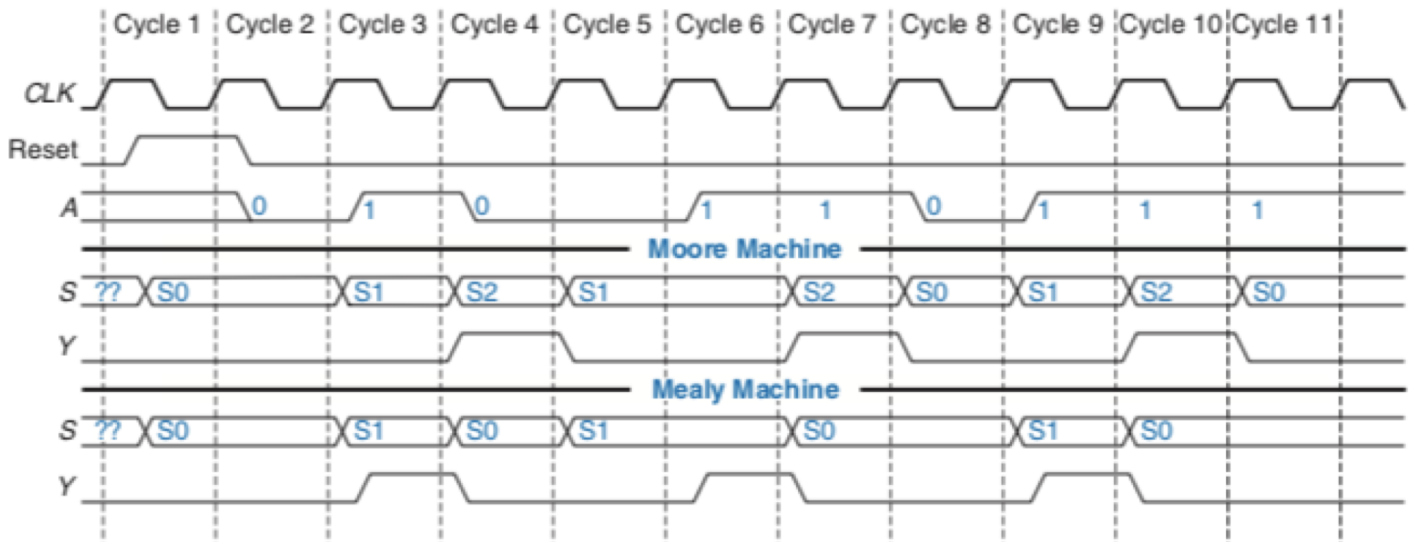
Current State S	Input A	Next State S'	Output Y
S0	0	S1	0
S0	1	S0	0
S1	0	S1	0
S1	1	S0	1

Current State S_1	Current State S_0	Input A	Next State S_1'	Next State S_0'
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

Current State S_1	Current State S_0	Output Y
0	0	0
0	1	0
1	0	1

Current State S_0	Input A	Next State S_0'	Output Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1





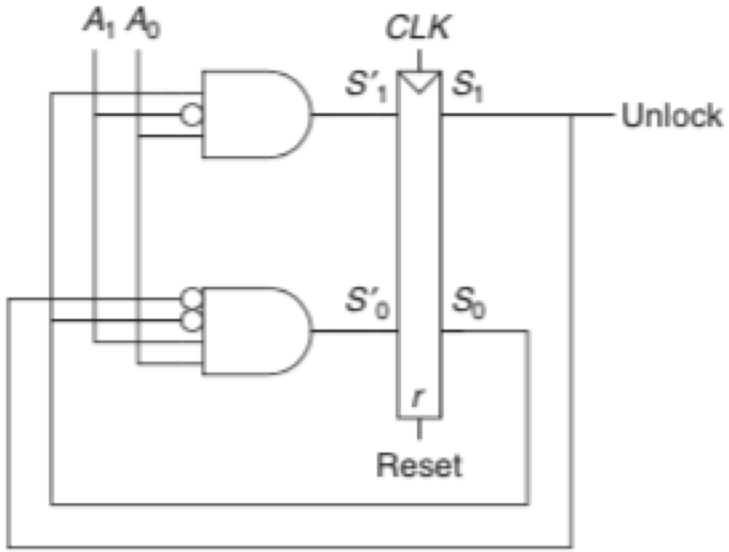
The Mealy machine's output rises a cycle sooner because it responds to the input rather than waiting for the state change

Deriving an FSM from a Schematic

- Examine circuit, stating inputs, outputs, and state bits.
- Write next state and output equations.
- Create next state and output tables.
- Reduce the next state table to eliminate unreachable states.
- Assign each valid state bit combination a name.
- Rewrite next state and output tables with state names.
- Draw state transition diagram.
- State in words what the FSM does.

Example

Consider the following circuit:



Derive the state transition diagram

Solution

This is a Moore machine because the output depends only on the state bits

The next state and output equations:

$$S_1' = S_0\overline{A_1}A_0$$

$$S_0' = \overline{S_1}\,\overline{S_0}A_1A_0$$

$$Unlock = S_1$$

Next state

Current State S_1	Current State S_0	Input A_1	Input A_0	Next State S_1'	Next State S_0'
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	0

Output table

Current State S_1	Current State S_0	Output *Unlock*
0	0	0
0	1	0
1	0	1
1	1	1

Reduce Tables

Remove unused states and combining rows using don't cares

- The S1:0 = 11 state is never listed as a possible next state so rows with this current state are removed.
- For current state S1:0 = 10, the next state is always S1:0 = 00, independent of the inputs, so don't cares are inserted for the inputs.

Current State		Input		Next State	
S_1	S_0	A_1	A_0	S_1'	S_0'
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	X	X	0	0

Current State		Output
S_1	S_0	*Unlock*
0	0	0
0	1	0
1	0	1

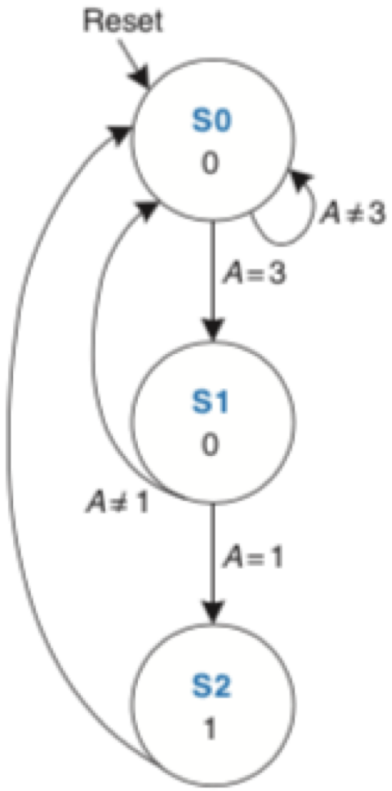
Assign names

S0 is $S_{1:0} = 00$, S1 is $S_{1:0} = 01$, and S2 is $S_{1:0} = 10$.

Current State S	Input A	Next State S'
S0	0	S0
S0	1	S0
S0	2	S0
S0	3	S1
S1	0	S0
S1	1	S2
S1	2	S0
S1	3	S0
S2	X	S0

Current State S	Output *Unlock*
S0	0
S1	0
S2	1

State transition diagram



The finite state machine unlocks the door only after detecting an input value, $A_{1:0}$, of three followed by an input value of one. The door is then locked again.

Timing

- A sequential element has an aperture time around the clock edge, defined by a setup time and a hold time, before and after the clock edge, respectively. During which the input must be stable for the flip-flop to produce a well-defined output.
- Recall: Static discipline limited us to using logic levels outside the forbidden zone, the dynamic discipline limits us to using signals that change outside the aperture time.
- The clock period has to be long enough for all signals to settle. The skew, due to variation in arrival of the clock does to all flip-flops, further increases the clock period.
- Using synchronizers when interfacing with asynchronous inputs

Dynamic Discipline

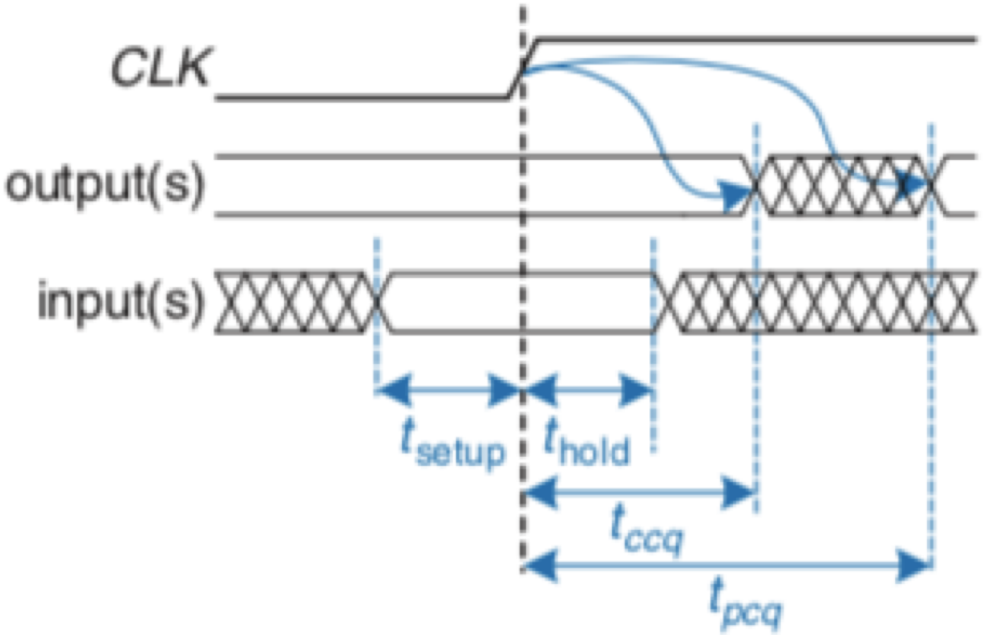
A synchronous sequential circuit, i.e. a flip-flop or FSM, has timing specifications:

After CLK rise, the outputs:

- Start to change after t_{ccq}
- Must settle within t_{pcq}

For correct input sampling, the inputs:

- Must have stabilized at least t_{setup}, before CLK rising
- Must remain stable for at least t_{hold}, after CLK rising



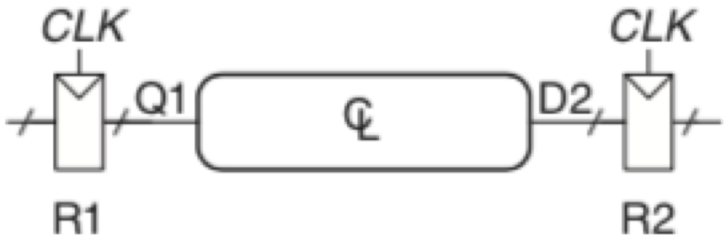
t_{ccq} - clock-to-Q contamination delay
t_{pcq} - clock-to-Q propagation delay
t_{setup} - setup time
t_{hold} - hold time

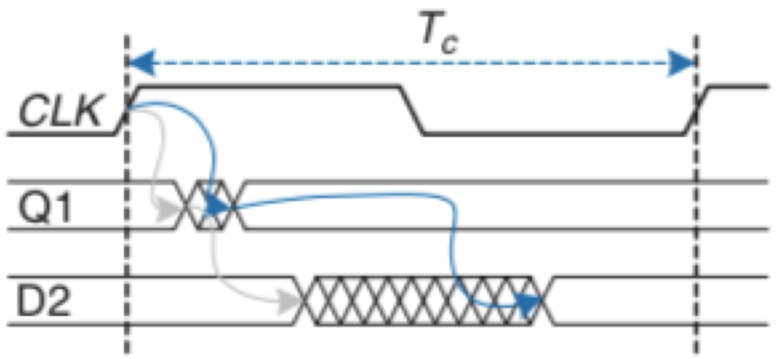
Aperture Time = $t_{setup} + t_{hold}$

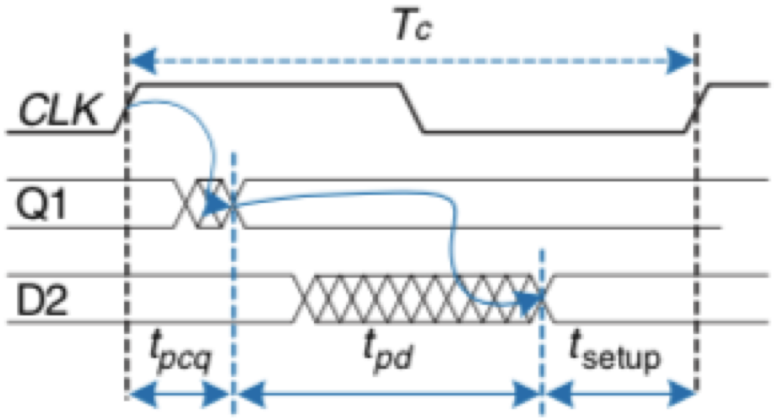
total time for which the input must remain stable.

Dynamic Discipline that the inputs of a synchronous sequential circuit must be stable during the setup and hold aperture time around the clock edge.

Setup Time Constraint







Minimum clock period:

$$T_c \geq t_{pcq} + t_{pd} + t_{setup}$$

Under the control of the designer

specified by the manufacturer

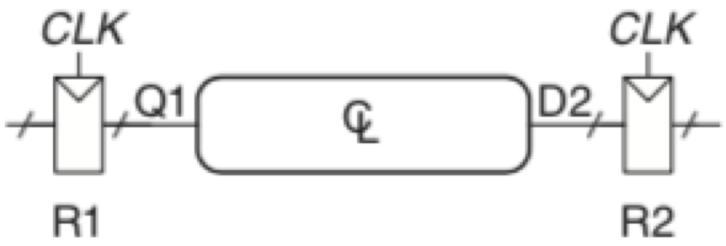
Setup time constraint or max-delay constraint :

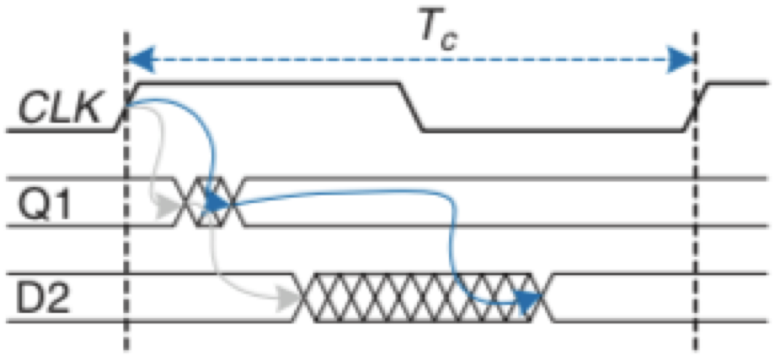
$$t_{pd} \leq T_c - (t_{pcq} + t_{setup})$$

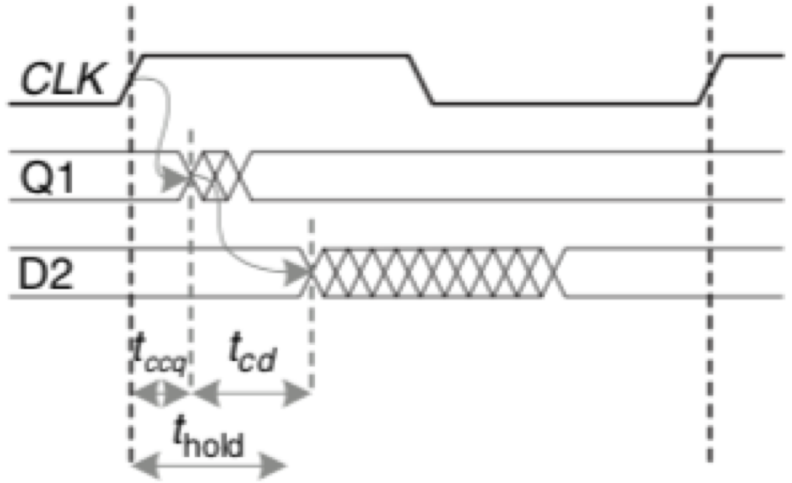
sequencing overhead

Maximum propagation delay through the combinational logic

Hold Time Constraint







R_2 input, D_2, must not change until some time, t_{hold}, after the rising edge of the clock.

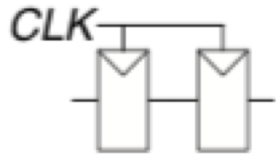
D_2 might change after $t_{ccq} + t_{cd} \geq t_{\text{hold}}$

Hold time constraint or min-delay constraint

$t_{cd} \geq t_{hold} - t_{ccq}$

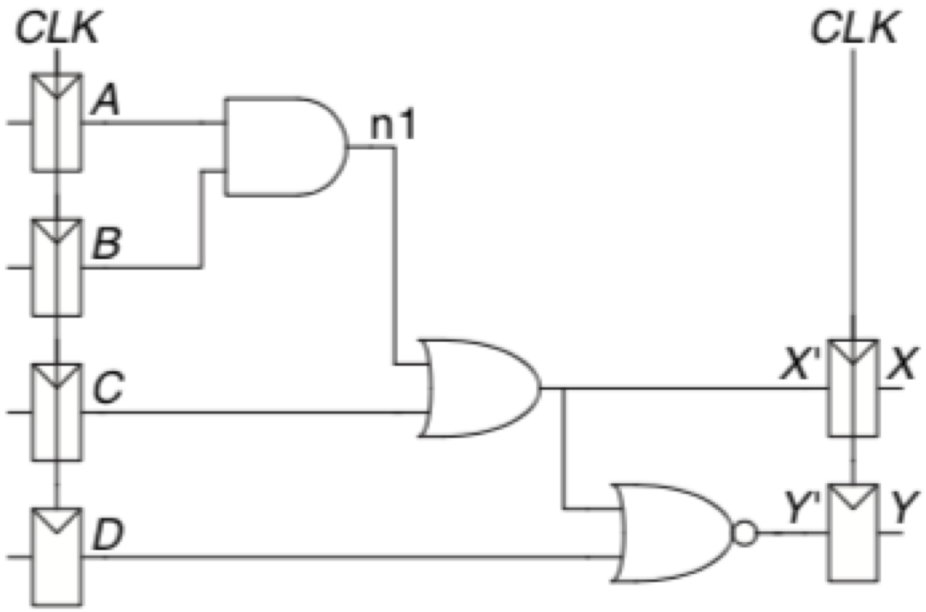
Minimum contamination delay through the combinational logic.

Consider Back-to-back flip-flops



$t_{\text{hold}} \leq t_{ccq}$

Example - Timing Analysis



Flip-flops have the following specifications:

t_{ccq} = 30 ps.

t_{pcq} = 80 ps.

t_{setup} = 50 ps.

t_{hold} = 60 ps.

Each logic gate has:

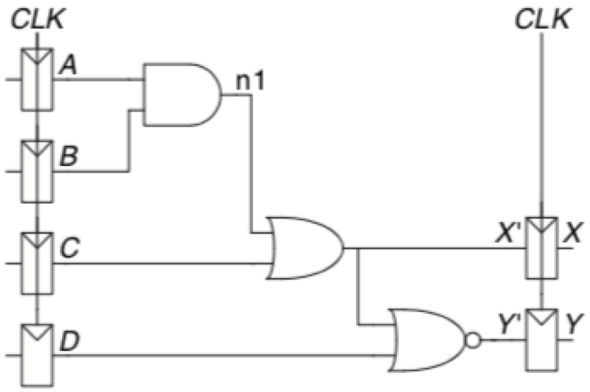
t_{pd} = 40 ps.

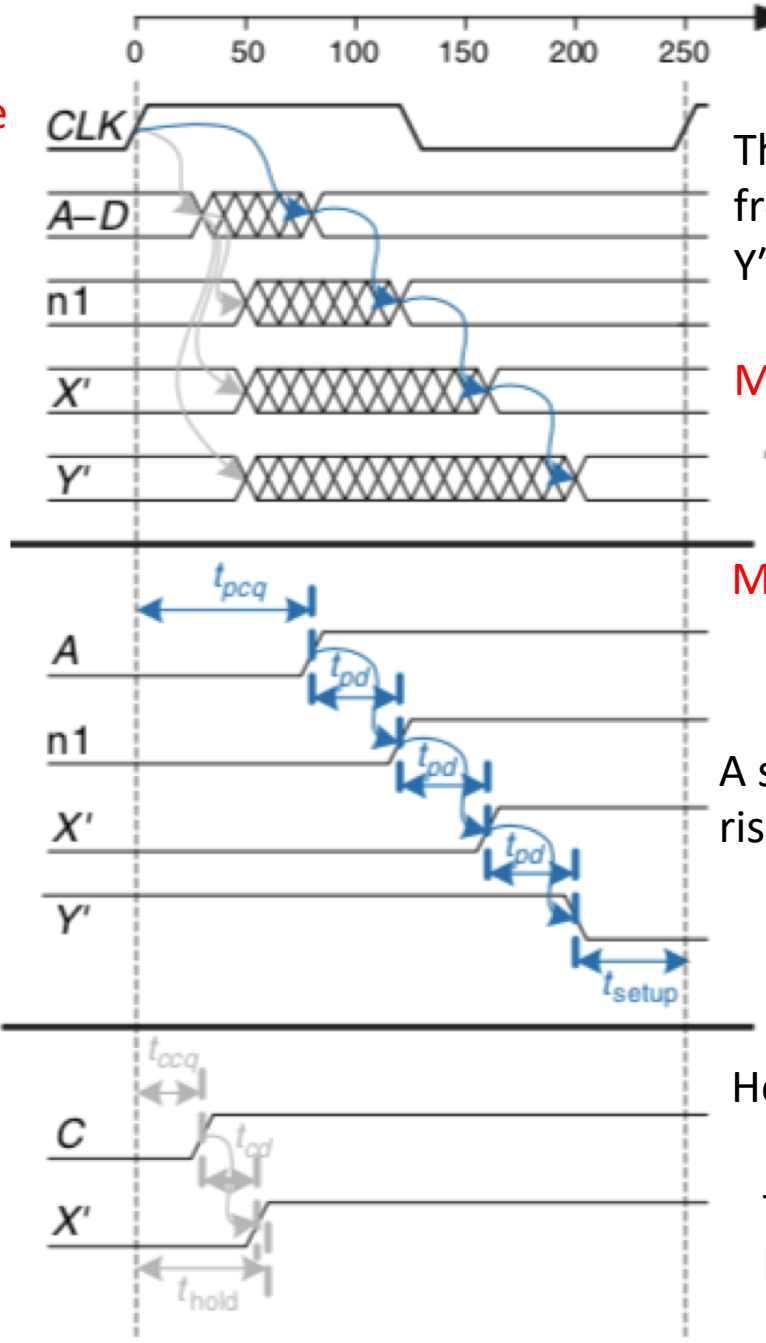
CLK t_{cd} = 25 ps.

Determine the maximum clock frequency and whether any hold time violations could occur.

Solution

General Case





Critical Path

Short Path

The critical path occurs when B=1, C=0, D=0, and A rises from 0 to 1.

Y' must setup before the next rising edge of the CLK.

Minimum cycle time:

$$T_c \geq t_{pcq} + 3\,t_{pd} + t_{setup} = 80 + 3 \times 40 + 50 = 250\text{ps}$$

Maximum clock frequency $f_c = 1/T_c = 4$ GHz.

A short path occurs when A = 0 and C rises, causing X' to rise each gate switches after only a contamination delay

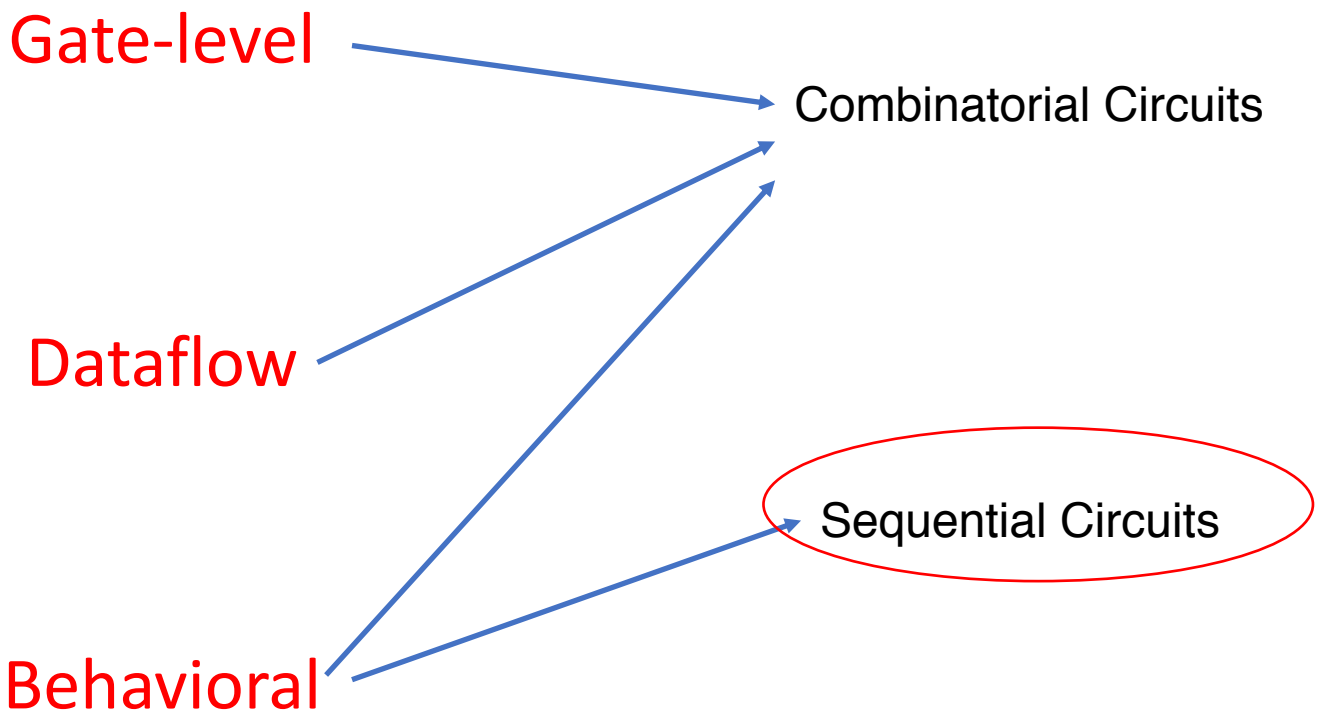
$$t_{ccq} + t_{cd} = 30 + 25 = 55 \text{ ps}$$

However, the flip-flop has a hold time of 60 ps

The circuit has a hold time violation and may behave erratically at any clock frequency

Verilog HDL

Three kinds of modeling styles

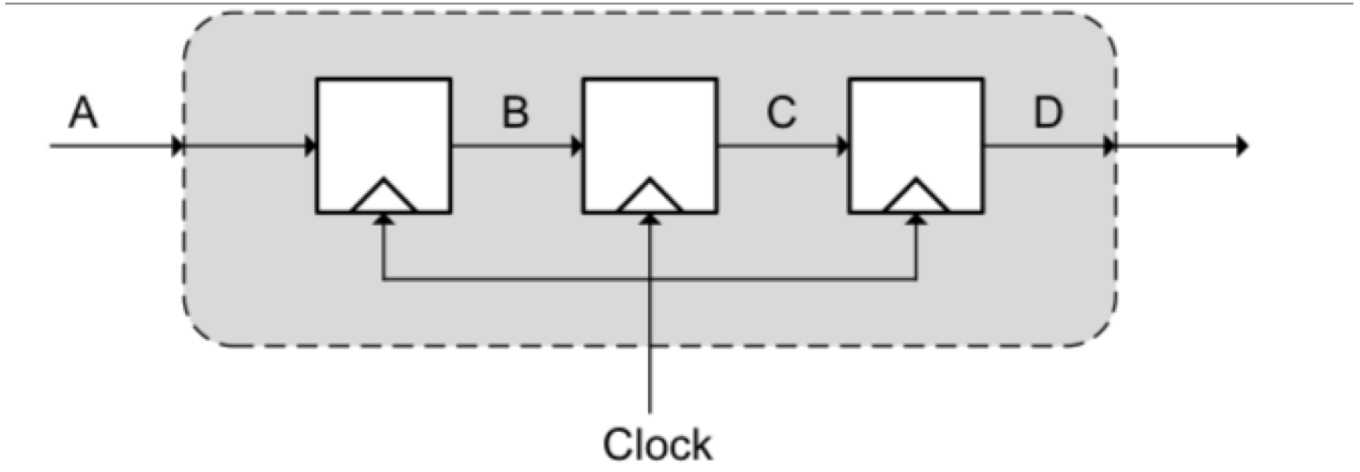


always@(posedge Clock) Blocks

always@(posedge Clock) => always at the positive edge of the clock

- used to describe Sequential Logic
- Non-blocking assignments (<=) are used

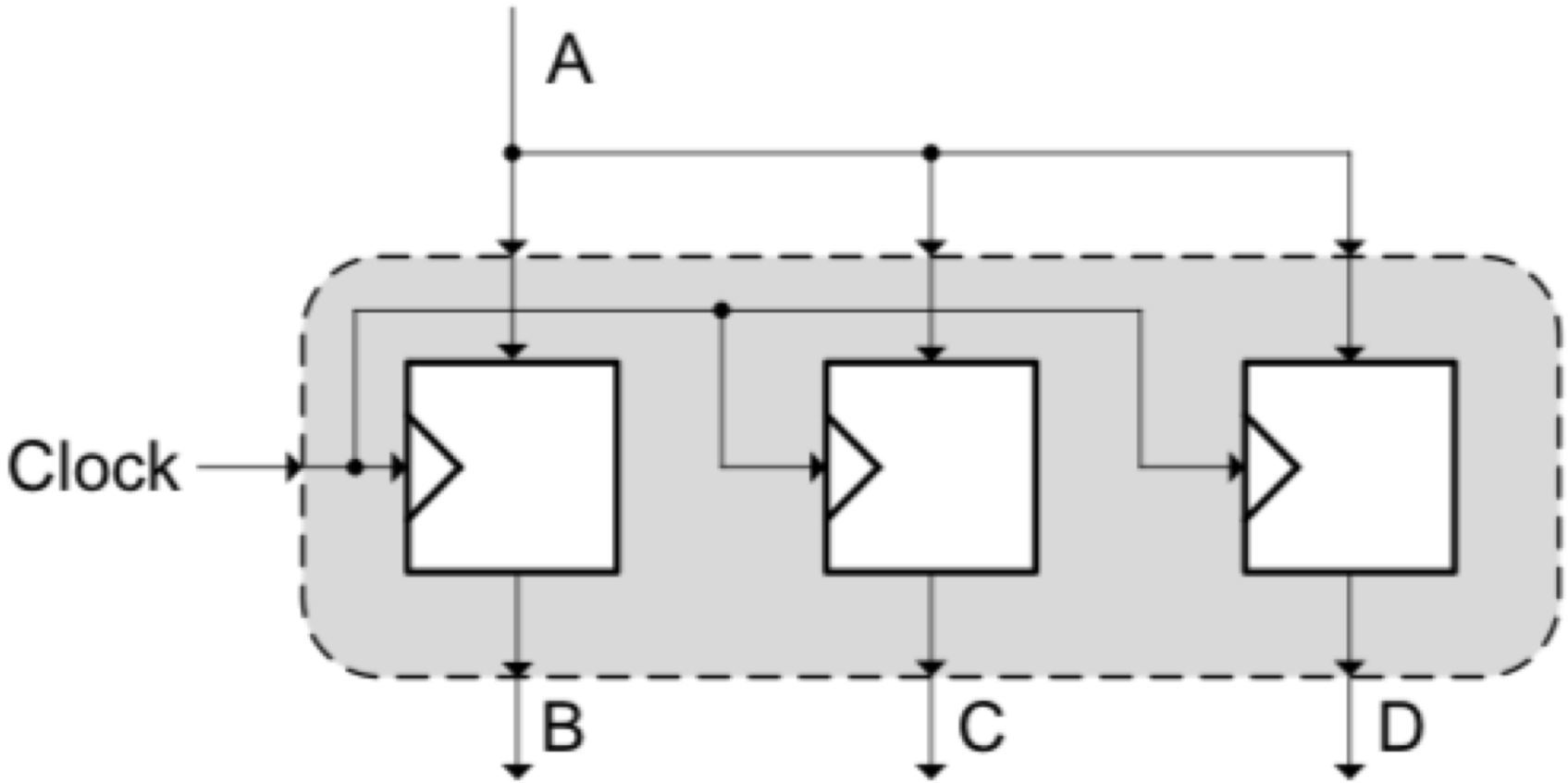
Example – Shift Register




```
always @(posedge Clock)
begin
        B <= A;
        C <= B;
        D <= C;
end
```

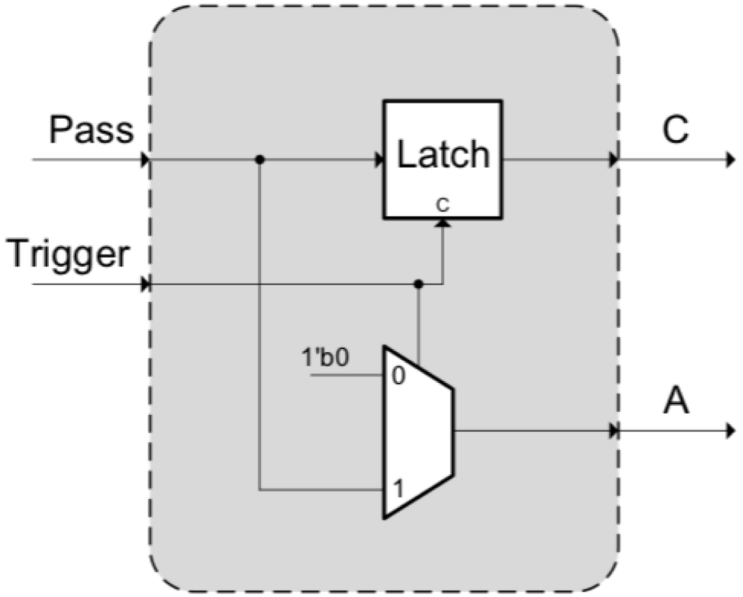
What if you use blocking assignments?

You create Parallel registers instead!



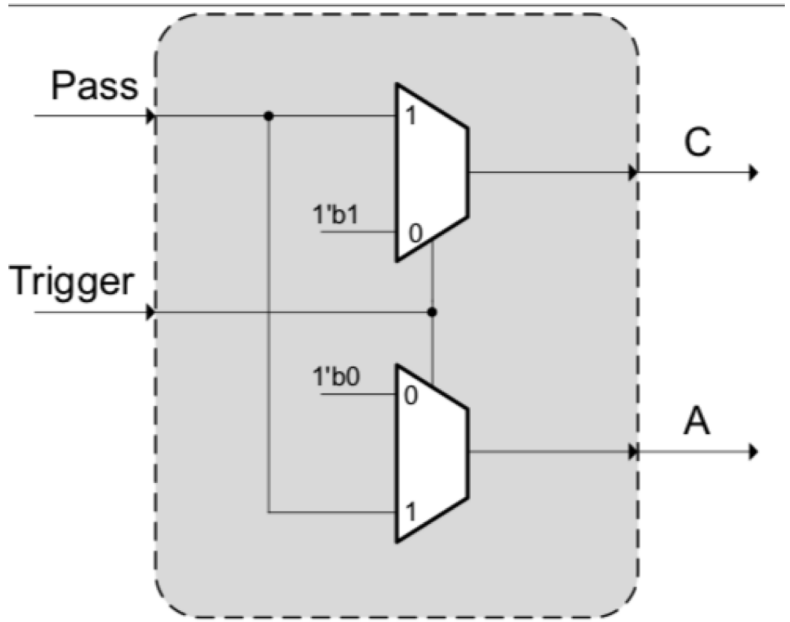
Latch Generation

```
wire Trigger , Pass;
reg A, C;

always @( * )
begin
A = 1'b0;
      if (Trigger)
      begin
            A = Pass;
            C = Pass;
      end
end
```




```
wire Trigger , Pass;
reg A, C;

always @( * )
begin
A = 1'b0;
C = 1'b1;
      if (Trigger)
      begin
            A = Pass;
            C = Pass;
      end
end
```

In every 'execution' of an always@(*) block, each value that is assigned in at least one place must be assigned to a non-trivial value during every execution of the always@(*) block



Read-Only Memories (ROM)

- Consists of interconnected arrays to store an array of binary information. m address input pins and n information output pins to store 2m words information
- Once the binary information is stored it can be read any time but cannot be altered
- It can be used to implement combinatorial circuits
- It uses a decoder
- An address is provided at the input and content of the corresponding word is read at the output pins.

In Verilog, memories can be defined as a two dimensional array using **reg** data type:

i.e. reg [3:0] MY_ROM [15:0];

16x4 memory with 16 locations each location being 4-bit wide

In order for the memory to be read only:

1. memory should only be read and not written into
2. memory should somehow be initialized with the desired content.

$readmemb can be used

Example 4x2 ROM

$readmemb (”data_bin_file", memory_identifier, begin_address, end_address) ;
$readmemh (”data_hex_file", memory_identifier, begin_address, end_address) ;

```
module ROM_4x2 (ROM_data, ROM_addr);
        output [1:0] ROM_data;
        input [1:0] ROM_addr;

        reg [1:0] ROM [3:0]; // defining 4x2 ROM

        assign ROM_data = ROM[ROM_addr]; // reading ROM content at the address ROM_addr

        initial $readmemb (“ROM_data.txt”, ROM, 0, 3); // load ROM content from ROM_data.txt file

endmodule
```

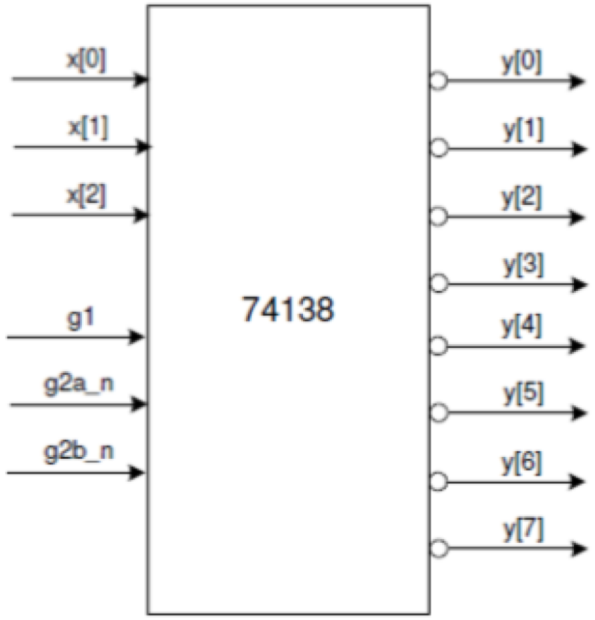
The ROM_data.txt file may have 4 or less lines such as:
11
00
1X
01
Note: unspecified locations will be initialized with 0s, should be present in the same directory if no directory path is provided

Lab 3

1. **Design a 3-to-8 line decoder. Assign the input to SW2-SW0 and output to LED7-LED0. Use dataflow modeling constructs.**
   a. Add the provided testbench and Simulate the design for 50 ns to verify that it works.
   b. Create the XDC file reflecting the LEDs and Switches of your choice for implementation.
   c. Synthesize, implement the design, generate the bitstream and download it into the ZedBoard to verify the functionality.

2. **Design and implement IC 74138 functionality, depicted below, using dataflow modeling and the decoder used in part 1.**

a. Add the provided testbench and Simulate the design for 200 ns to verify that it works.
b. Create the XDC file reflecting the LEDs and Switches of your choice for implementation.
c. Synthesize, implement the design, generate the bitstream and download it into the ZedBoard to verify the functionality.



g_1 g_{2a_n} g_{2b_n}	x_0 x_1 x_2	y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7
0 x x	x x x	1 1 1 1 1 1 1 1
x 1 x	x x x	1 1 1 1 1 1 1 1
x x 1	x x x	1 1 1 1 1 1 1 1
1 0 0	0 0 0	0 1 1 1 1 1 1 1
1 0 0	0 0 1	1 0 1 1 1 1 1 1
1 0 0	0 1 0	1 1 0 1 1 1 1 1
1 0 0	0 1 1	1 1 1 0 1 1 1 1
1 0 0	1 0 0	1 1 1 1 0 1 1 1
1 0 0	1 0 1	1 1 1 1 1 0 1 1
1 0 0	1 1 0	1 1 1 1 1 1 0 1
1 0 0	1 1 1	1 1 1 1 1 1 1 0

3. **Implement a 2-bit by 2-bit multiplier using a ROM and $readmemb system task Use LEDs to display the output in binary.**
    a. Create a text file describing the design output.
    b. Create the XDC file reflecting the LEDs and Switches of your choice for implementation.
    c. Synthesize, implement the design, generate the bitstream and download it into the ZedBoard to verify the functionality.

Decoder 3-to-8 Test Bench

Use the following testbench for Lab 3 Part 1.a:

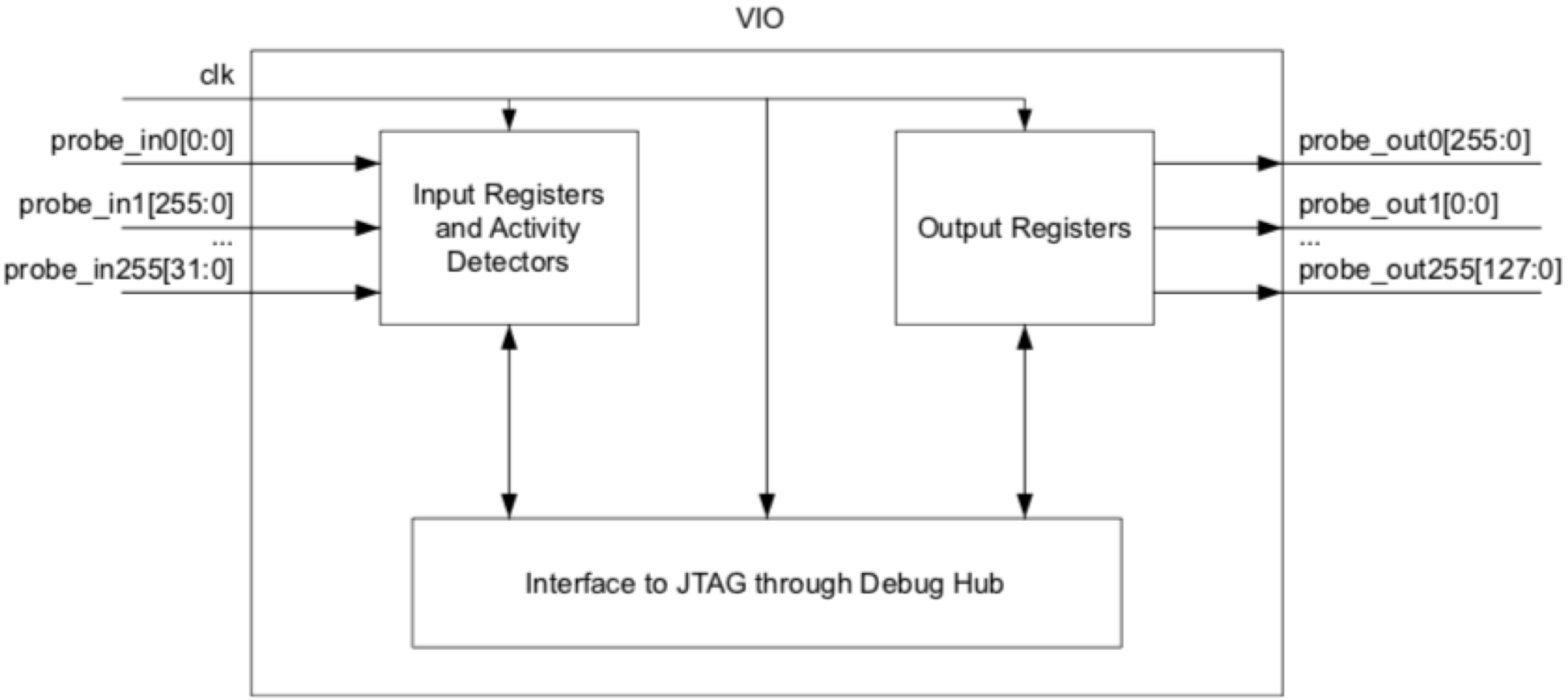
```
`timescale 1ns / 1ps

module decoder_3to8_dataflow_tb(
    );
    reg [2:0] x;
    wire [7:0] y;
    integer k;

    decoder_3to8_dataflow DUT (.x(x), .y(y));

    initial
    begin
     x = 0;
            for (k=0; k < 8; k=k+1)
                #5 x=k;
            #10;
    end
endmodule
```

Use the following testbench for Lab 3 Part 2:

```
`timescale 1ns / 1ps
module decoder_74138_dataflow_tb(  );
   reg [2:0] x;
   reg g1, g2a_n, g2b_n;
   wire [7:0] y;
   integer k;
   decoder_74138_dataflow DUT (.g1(g1), .g2a_n(g2a_n), .g2b_n(g2b_n), .x(x), .y(y));
   initial
   begin
    x = 0; g1 = 0; g2a_n = 1; g2b_n = 1;
              for (k=0; k < 8; k=k+1)
                        #5 x=k;
              #10;
    x = 0; g1 = 1; g2a_n = 0; g2b_n = 1;
              for (k=0; k < 8; k=k+1)
                        #5 x=k;
              #10;
    x = 0; g1 = 0; g2a_n = 1; g2b_n = 0;
              for (k=0; k < 8; k=k+1)
                        #5 x=k;
              #10;
    x = 0; g1 = 1; g2a_n = 0; g2b_n = 0;
              for (k=0; k < 8; k=k+1)
                        #5 x=k;
              #10;
   end
endmodule
```

Virtual Input/output (VIO) Core

The LogiCORETM IP Virtual Input/Output (VIO) core is a customizable core that can both monitor and drive internal FPGA signals in real time.



https://www.xilinx.com/support/documentation/ip_documentation/vio/v3_0/pg159-vio.pdf