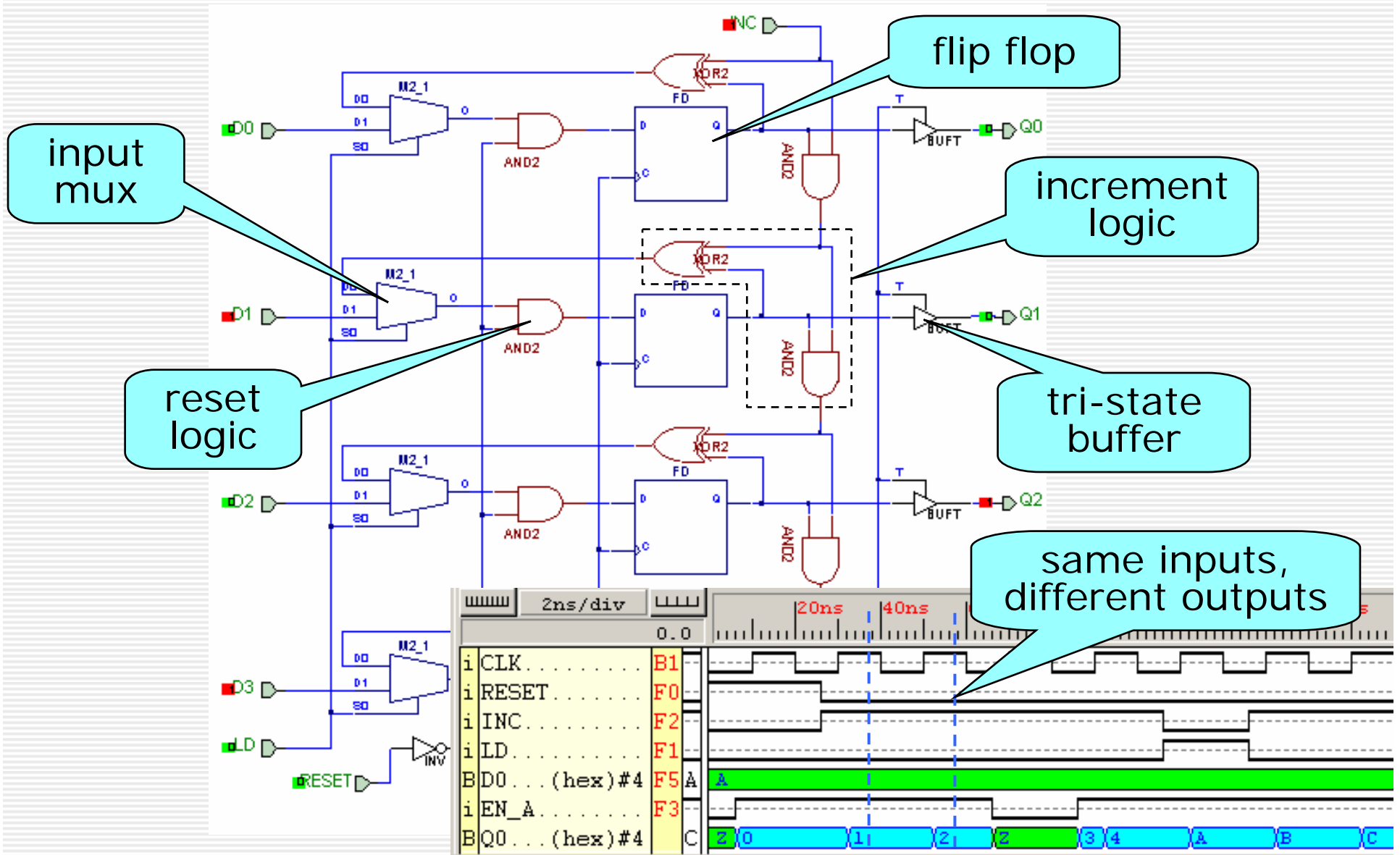


# Introduction to Sequential Circuits

- Basic Sequential Circuit Design (W 521-522)
- Latches and Flip Flops (W 523-541)
- Sequential Circuit Analysis (W 542-553)
- General Circuit Design Method (W 553-576)
- Designing Sequential Circuits with VHDL  
(online tutorial and W 625-645)



# Program Counter Schematic (4 bit)



# Program Counter in VHDL

```
entity program_counter is
  port (
    clk, en_A, ld, inc, reset: in STD_LOGIC;
    aBus: out STD_LOGIC_VECTOR(15 downto 0);
    dBus: in STD_LOGIC_VECTOR(15 downto 0)
  );
end program_counter;
architecture pcArch of program_counter is
  signal pcReg: STD_LOGIC_VECTOR(15 downto 0);
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then pcReg <= x"0000";
      elsif ld = '1' then pcReg <= dBus;
      elsif inc = '1' then
        pcReg <= pcReg + x"0001";
      end if;
    end if;
  end process;
  aBus <= pcReg when en_A = '1' else "ZZZZZZZZZZZZZZZZ";
end pcArch;
```

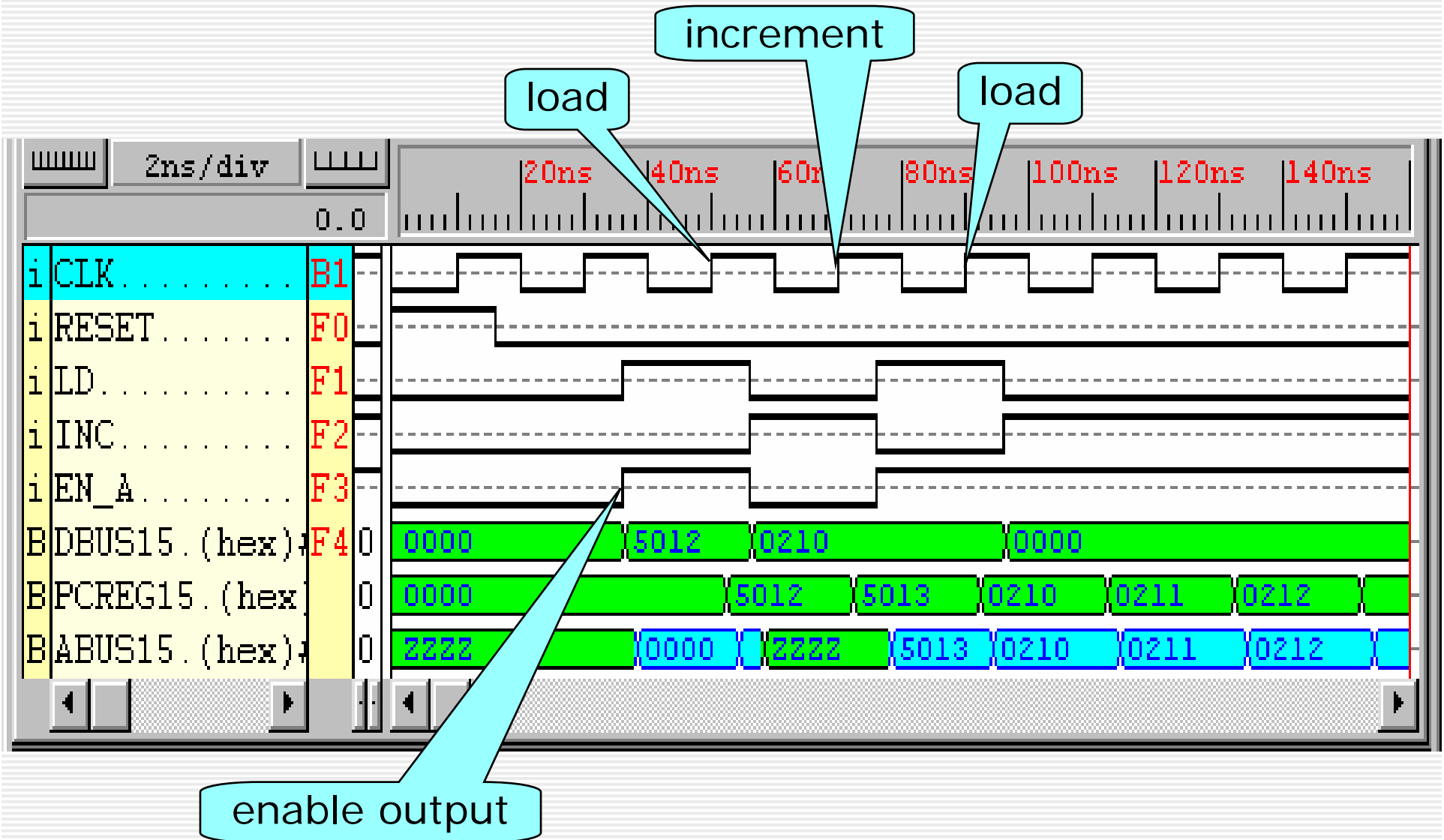
PC  
register

reset  
logic

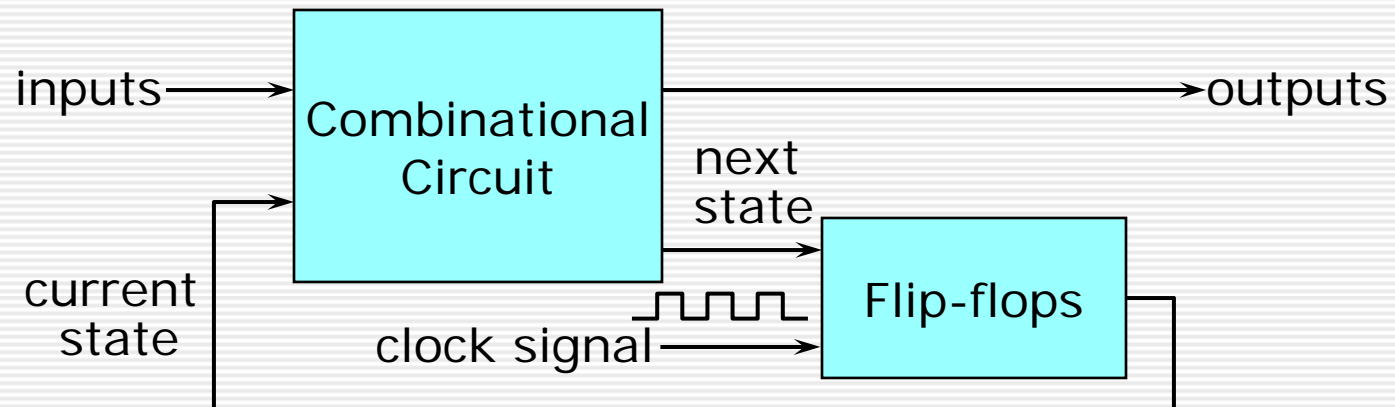
output  
to aBus

increment  
logic

# VHDL PC Simulation



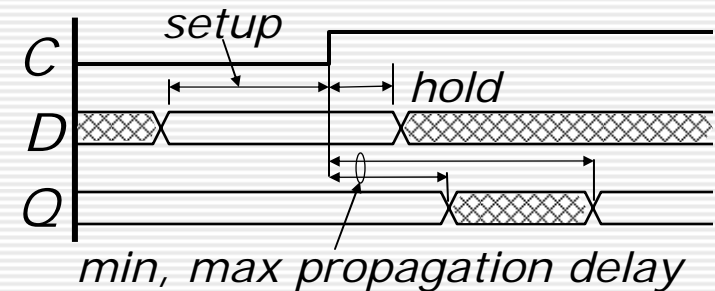
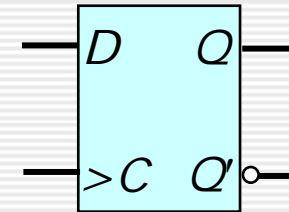
# Clocked Sequential Circuits



- In *sequential* circuits, output values may depend on both current and past input values.
  - » consists of combinational circuit plus *storage elements*
  - » each storage element stores one bit of information
  - » the *state* of a sequential circuit is the set of stored values
- In *clocked sequential circuits*, state changes are driven by *clock signals*.
  - » information stored using *flip-flops*.

# Edge-Triggered D Flip Flop

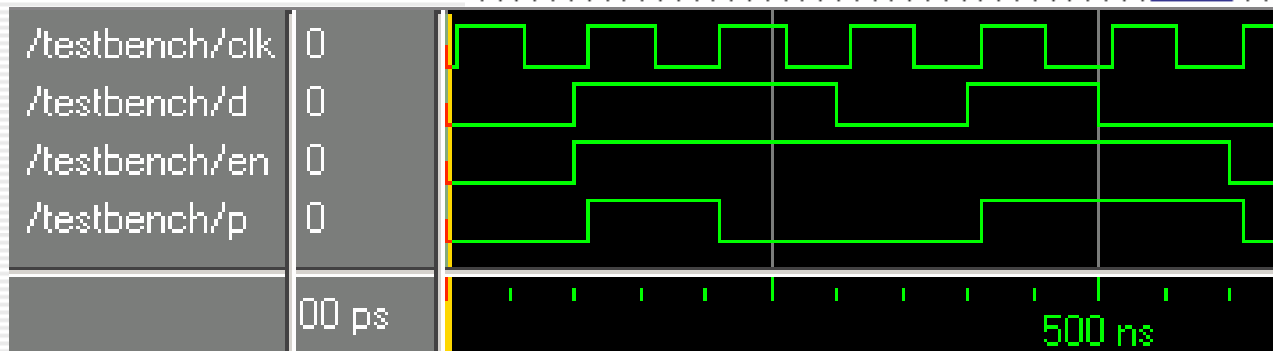
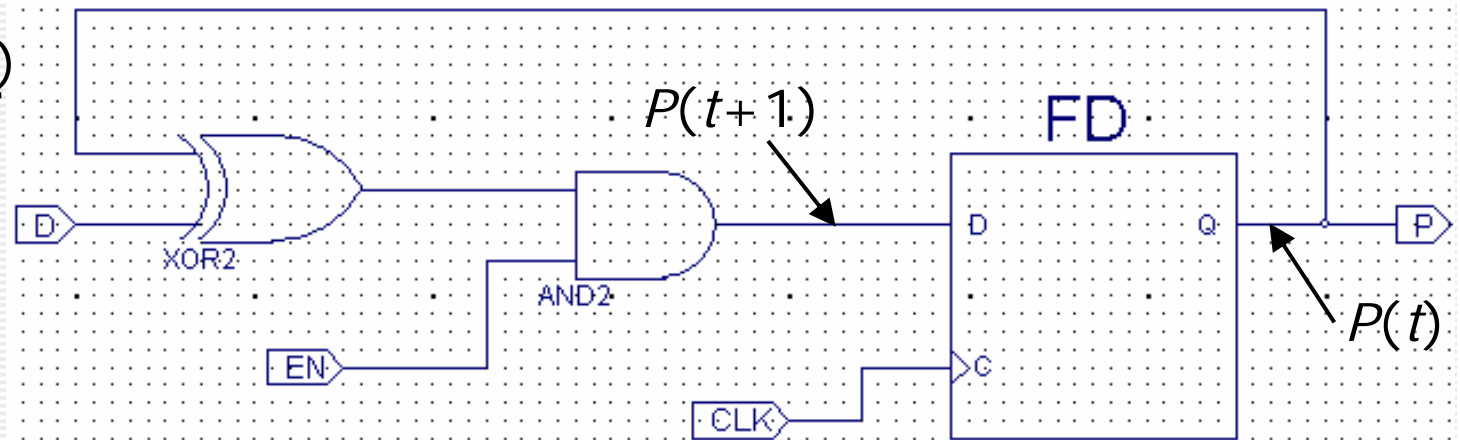
- $D$  flip flop stores value at  $D$  input when clock rises.
- Most widely used storage element for sequential circuits.
- *Propagation time* is time from rising clock to output change.
- If input changes when clock rises, new value is uncertain.
  - » output may oscillate or may remain at intermediate voltage (*metastability*)
- Timing rules to avoid metastability
  - »  $D$  input must be stable for *setup time* before rising clock edge
  - » must remain stable for *hold time* following rising clock edge



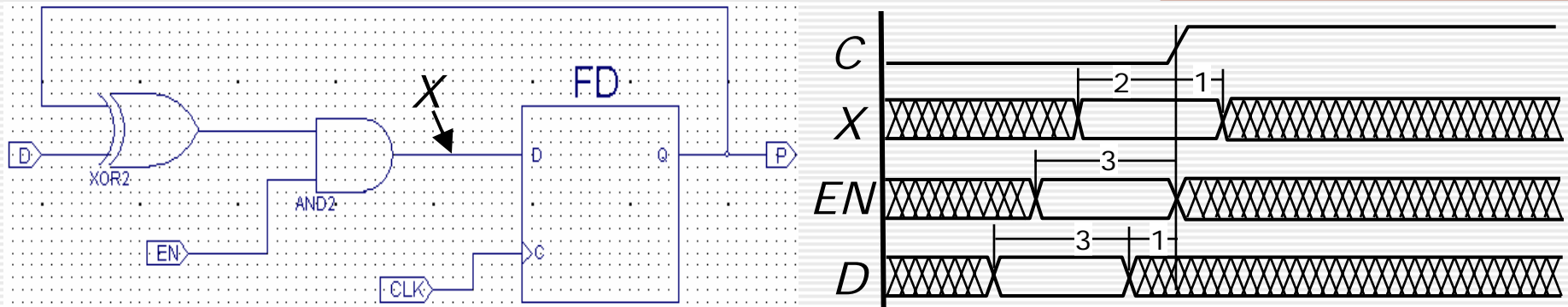
# Serial Parity Generator

- Circuit with data input, enable input & parity output.
  - » when enable is low, output is low; when enable is high, output is high if number of "1"s seen since enable went high is odd
- Next state table gives next state and output, as function of current state and input.

$P(t)$	$D$	$EN$	$P(t+1)$
x	x	0	0
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0



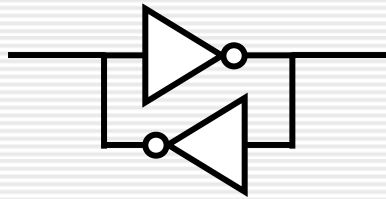
# Input Timing for Parity Circuit



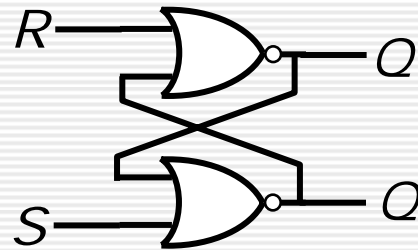
- To meet setup & hold time requirements of flip flop, inputs to circuit must be stable during certain times.
  - » let setup time=2 ns, hold time=1 ns and gate delay=1 ns
  - » then  $D$  must be stable from 4 ns before clock edge until 1 ns after clock edge; similarly for  $EN$
  - » these input conditions are summarized in timing diagram
  - » if gate delay can vary between .4 and 1.5 ns, then stable period for  $D$  is from 5 ns before clock edge to .2 ns after.

# The SR Latch

Basic Storage Element



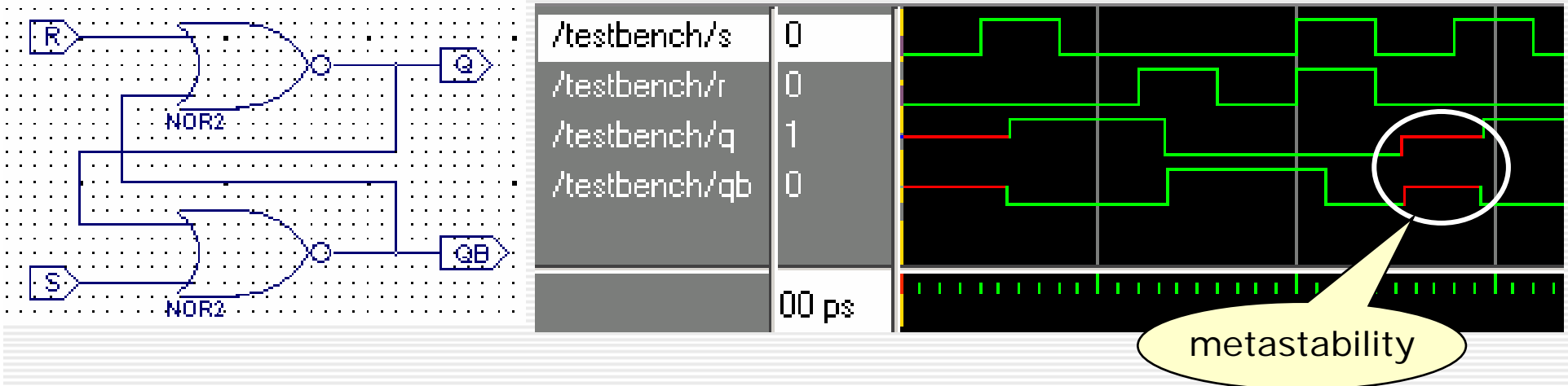
SR Latch



S	R	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	??

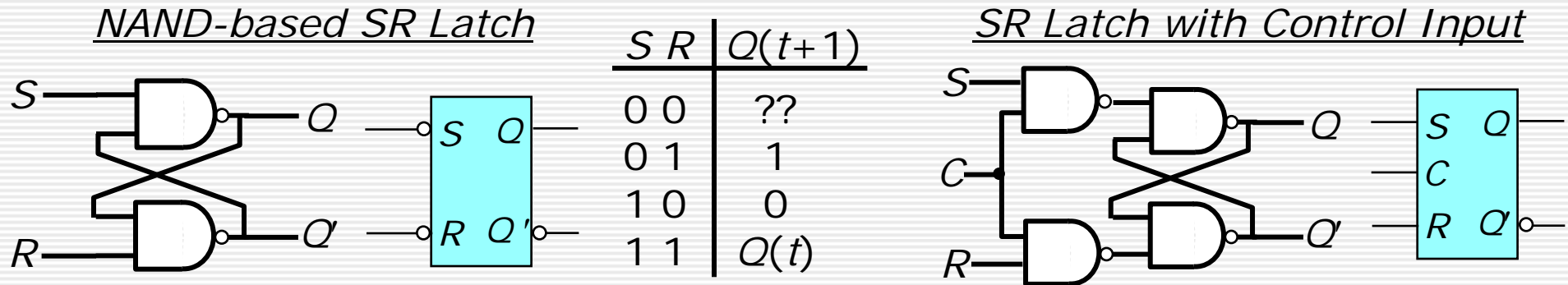
- Pair of inverters provides stable storage.
- To enable stored value to be changed, use cross-coupled NOR gates.
  - » equivalent to inverter pair when both inputs are low
- SR latch is key building block for flip flops.
  - » when  $S=1, R=0$  latch is *set*
  - » when  $S=0, R=1$  latch is *reset*
  - » when  $S=0, R=0$  latch retains value
  - » when  $S=1, R=1$  latch state is undefined

# S-R Latch Behavior



- Note that when  $S=R=1$ , both outputs are low.
  - » outputs are not complements of each other in this case
- When  $S, R$  drop together, latch output is undefined.
  - » may remain at intermediate voltage
  - » or, may oscillate between low and high values
- Latch *metastability* can cause unpredictable circuit behavior.
- For these reasons, avoid  $S=R=1$  condition.

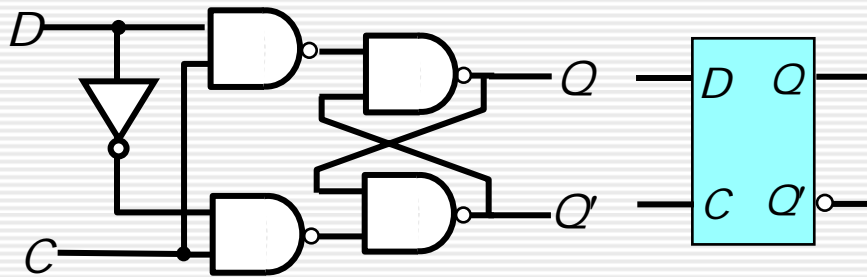
# More on SR Latches



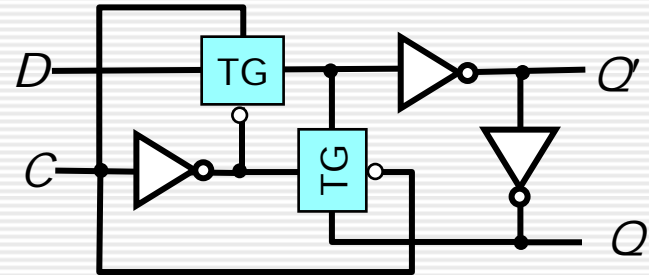
- SR latch most often implemented with NAND gates.
  - » inputs are active low (negative logic inputs)
  - » when both inputs are low, both outputs high
  - » when inputs rise together, outputs can become metastable
- SR latch with control input changes state only when control input is high.
  - » inputs are active high
  - » forbidden input condition is  $C=S=R=1$
  - » change  $S, R$  inputs when  $C=0$

# D Latch

*D Latch*

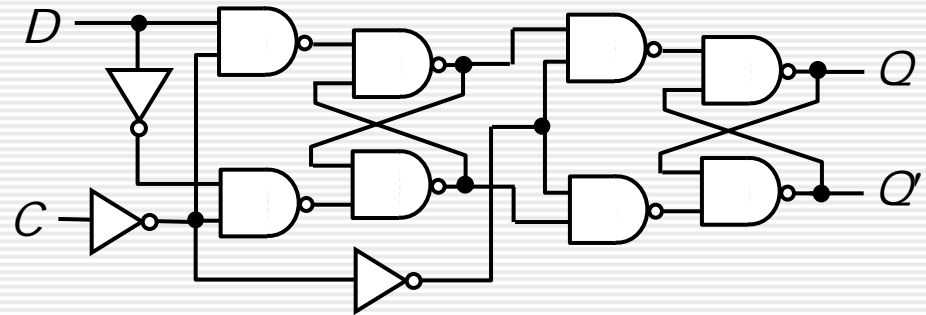
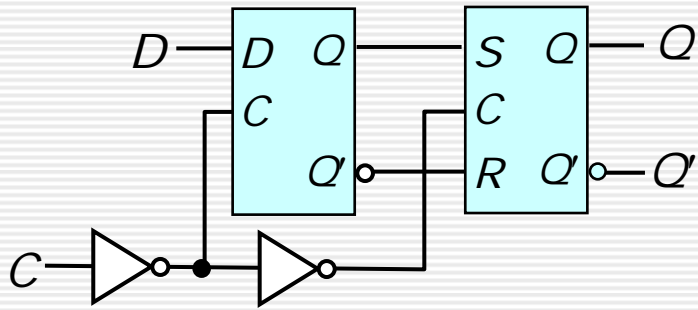


*Alternative Implementation*



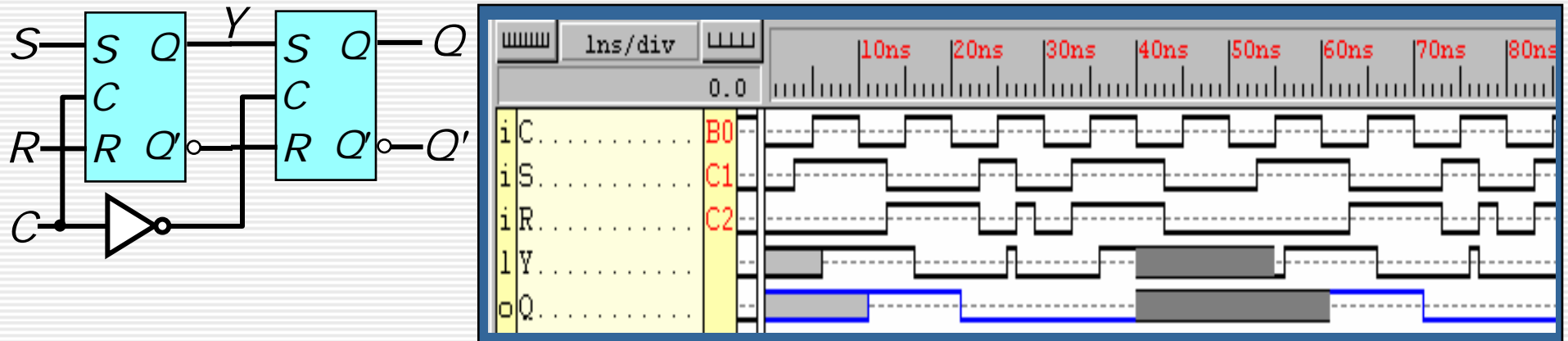
- The *D* latch stores the value on the *D* input when the control input is asserted.
  - » no forbidden input combinations
  - » but input should be stable when the control input drops
  - » if not, outputs may become metastable
- Alternative implementation uses transmission gates.
  - » TGs enable either input or feedback path
  - » in CMOS, this uses 10 transistors vs. 18

# Implementing D Flip Flops



- When clock rises, value in  $D$ -latch propagates to  $SR$ -latch and outputs.
- New value determined by  $D$  input at time clock rises.
- Flip flop setup and hold time conditions designed to prevent metastability in latches.
- Propagation delay determined primarily by  $SR$ -latch delay.

# SR Master-Slave Flip Flop

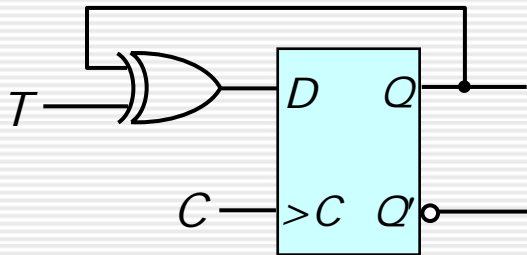


- The SR master-slave flip flop uses two SR latches with complementary enables.
- First stage follows *all changes while clock is high*, but second stage only "sees" value after the clock drops.
  - » *not the same as a negative edge-triggered flip flop*
- Forbidden input combination causes metastability.
- Recommended usage: change  $S$ ,  $R$  only when  $C=0$ .

# Other Types of Flip Flops

## ■ Toggle flip flop.

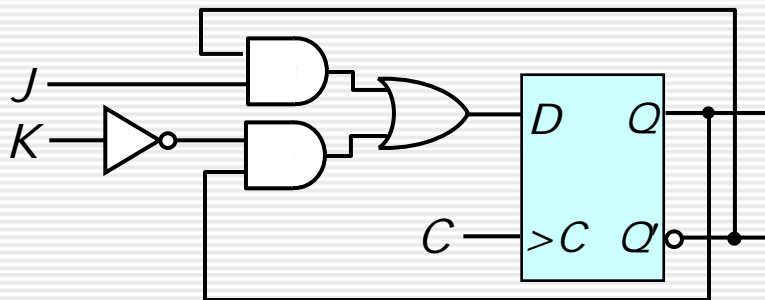
» change state when  $T$  input high



$T$	$Q(t+1)$
0	$Q(t)$
1	$Q'(t)$

## ■ J-K flip flop.

» set when  $J$  high, reset when  $K$  high, toggle when both high

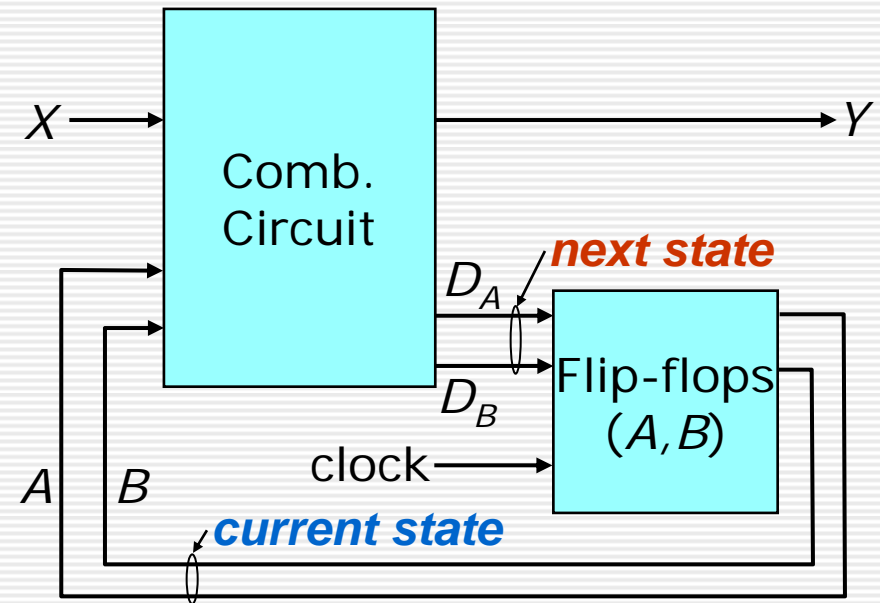


$J$	$K$	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$Q'(t)$

# State Tables

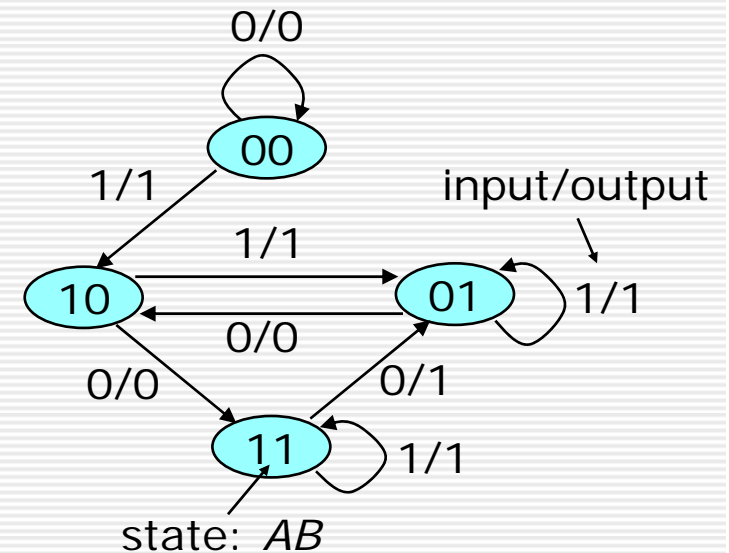
- The behavior of a sequential circuit can be defined by a *state table*, which specifies
  - » the outputs produced by circuit under different conditions
  - » and how inputs cause state transitions
- The following state table describes a sequential circuit with two flip flops, one input & one output.

<i>Current State</i>		<i>Input</i>	<i>Output</i>		<i>Next State</i>	
<i>A</i>	<i>B</i>	<i>X</i>	<i>Y</i>		<i>D<sub>A</sub></i>	<i>D<sub>B</sub></i>
0	0	0	0	0	0	0
0	0	1	1	1	1	0
0	1	0	0	0	1	0
0	1	1	1	1	0	1
1	0	0	0	0	1	1
1	0	1	1	1	0	1
1	1	0	1	1	0	1
1	1	1	1	1	1	1



# State Diagrams

- *State diagrams* are a more intuitive way to represent the information in a state table.
- State diagrams are often used as a high level specification for a sequential circuit.
- Note that output value on an arc is determined by *current state* and the input value.
- The state diagram contains *exactly the same information* as the state table

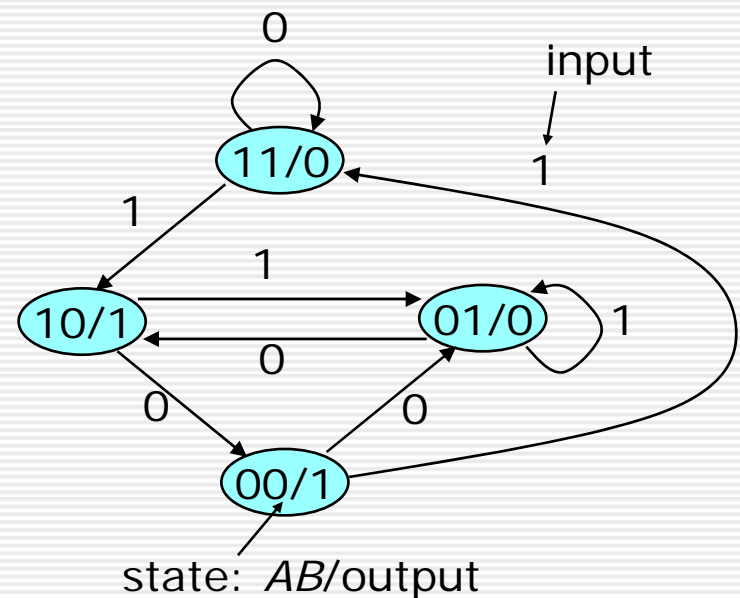


Current State	Input	Output	Next State
A B	X	Y	$D_A D_B$
0 0	0	0	0 0
0 0	1	1	1 0
0 1	0	0	1 0
0 1	1	1	0 1
1 0	0	0	1 1
1 0	1	1	0 1
1 1	0	1	0 1
1 1	1	1	1 1

# Synchronous and Asynchronous Outputs

- A sequential circuit output is called *synchronous* if it is a function of the current state only.
  - » such outputs change only following a clock transition
  - » synchronous outputs make it easier to ensure that setup and hold time conditions are satisfied
- Other outputs are *asynchronous*.
- Circuits that have all synchronous outputs are called *Moore model* circuits.
  - » simplified state diagram used for Moore model circuits
- Circuits with one or more asynchronous output are called *Mealy model* circuits.

Moore Model State Diagram

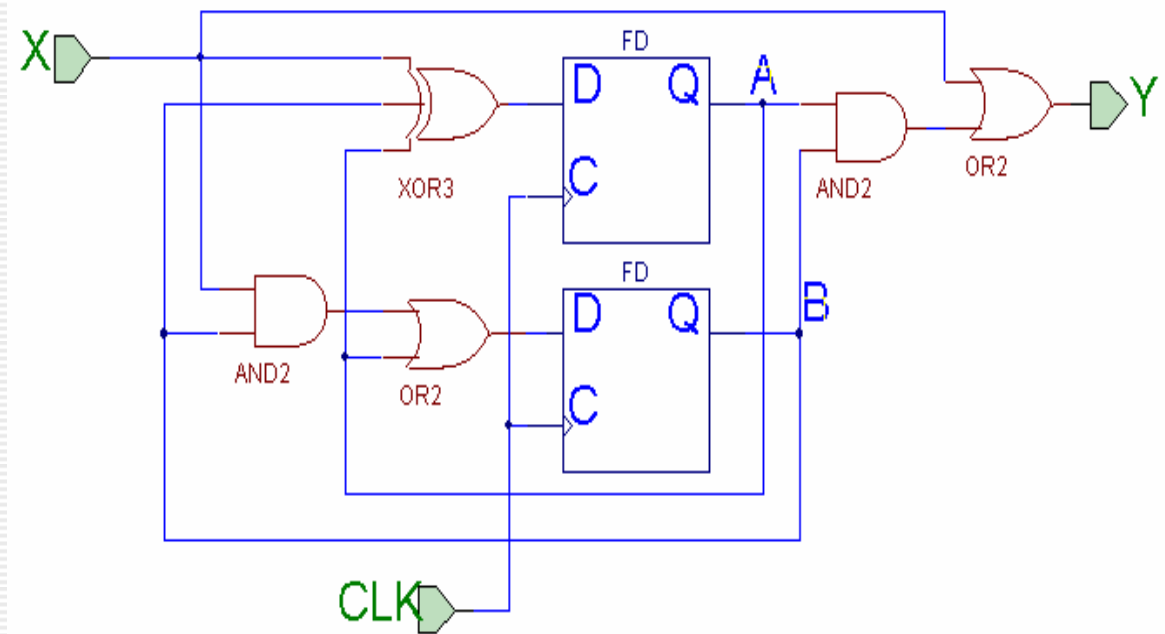


# Analyzing Sequential Circuits

- Analysis involves finding the specification (e.g. state diagram) for a given sequential circuit.

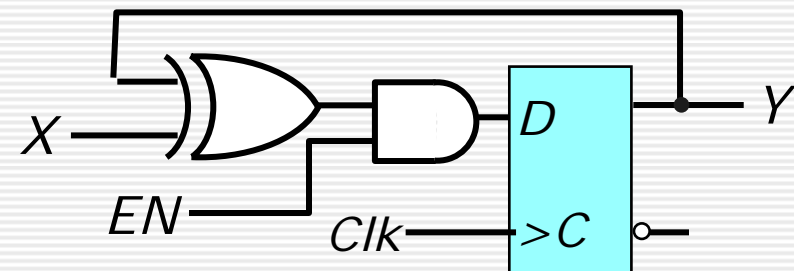
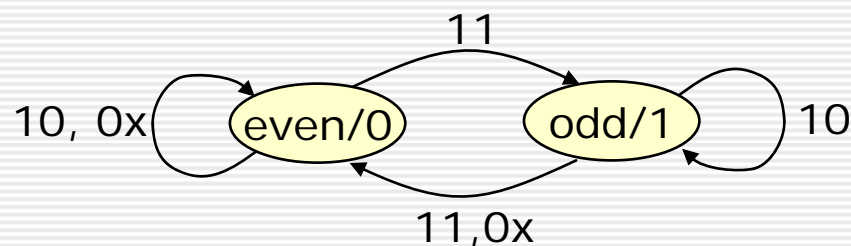
- Procedure

1. Name inputs, outputs and flip flops.
2. Write output equations
  - $Y = AB + X$
3. Write next-state equations.
  - $D_A = A \oplus B \oplus X,$
  - $D_B = A + BX$
4. Format and fill in state table.
5. Draw state diagram.



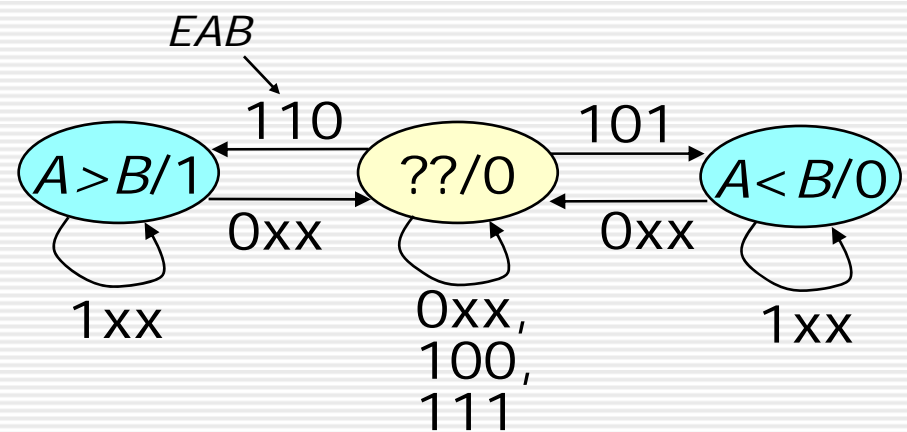
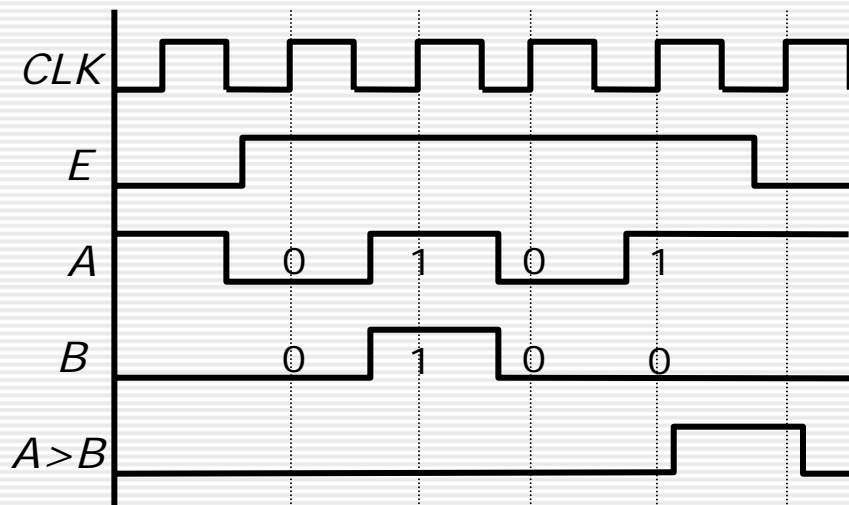
# Sequential Circuit Design Procedure

- State machine specification often given in English.
  - » e.g. design a circuit with inputs  $X$ ,  $EN$  and output  $Y$ ;  $Y=0$  when  $EN=0$ ; during a period when  $EN=1$ ,  $Y=1$  if  $X$  has been 1 during an odd number of clock ticks, else  $Y=0$
- Procedure.
  1. Determine what things the circuit must “remember.”
  2. Define states and draw state diagram.
  3. Determine number of flip flops and choose state encoding.
  4. Construct state table.
  5. Determine logic equations for each output signal.
  6. Determine logic equation for each flip flop input.

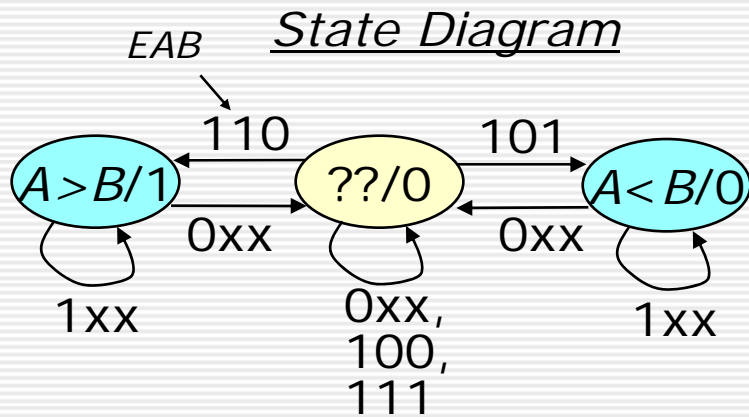


# Sequential Comparator

- A *sequential comparator* has two data inputs ( $A, B$ ), an enable input ( $E$ ) and a single output ( $A > B$ ).
  - » if enable is low at clock edge, then output becomes low after clock edge and stays low so long as enable is low
  - » when enable is high, the circuit compares  $A$  and  $B$  numerically (assuming the values are presented with the most-significant bit, first) and outputs 1 if  $A > B$ .
- Example:



# Sequential Comparator Design



Three states implies at least 2 flip flops. One encoding is

- 00 for ??
- 10 for  $A > B$ ,
- 01 for  $A < B$

Present State $s_1 s_0$	Inputs $EAB$	Output $A > B$	Next State $D_{s_1} D_{s_0}$
00	0xx	0	00
	100	0	00
	110	0	10
	101	0	01
10	1xx	1	10
	0xx	1	00
01	1xx	0	01
	0xx	0	00

## Output equation:

$$A > B = s_1 s_0'$$

(simplifies to  $s_1$ )

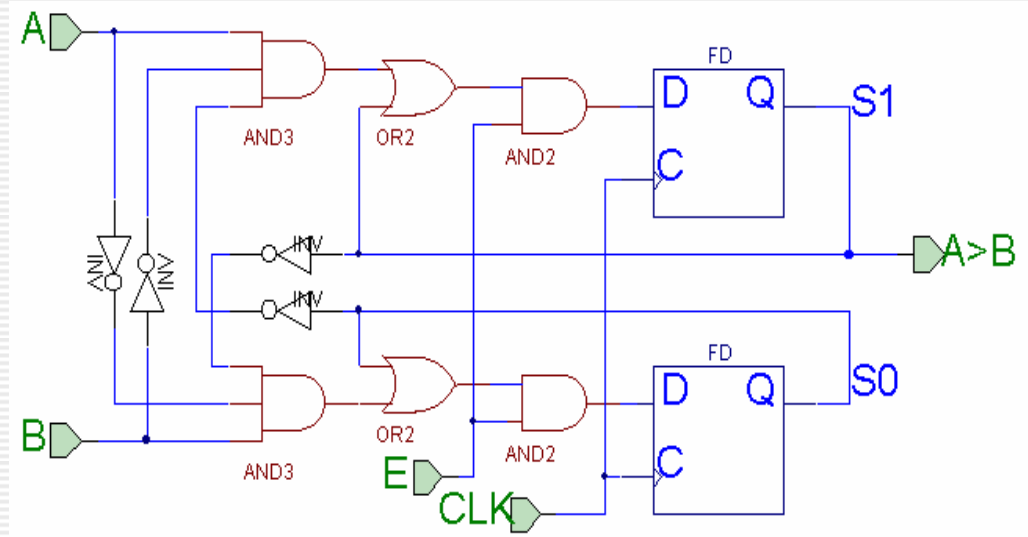
## Next state equations:

$$D_{s_1} = (s_1 + s_1' s_0' AB') E$$

$$= (s_1 + s_0' AB') E$$

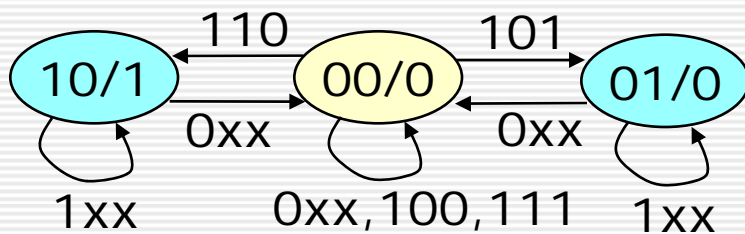
$$D_{s_0} = (s_0 + s_1' s_0' A'B) E$$

$$= (s_0 + s_1' A'B) E$$

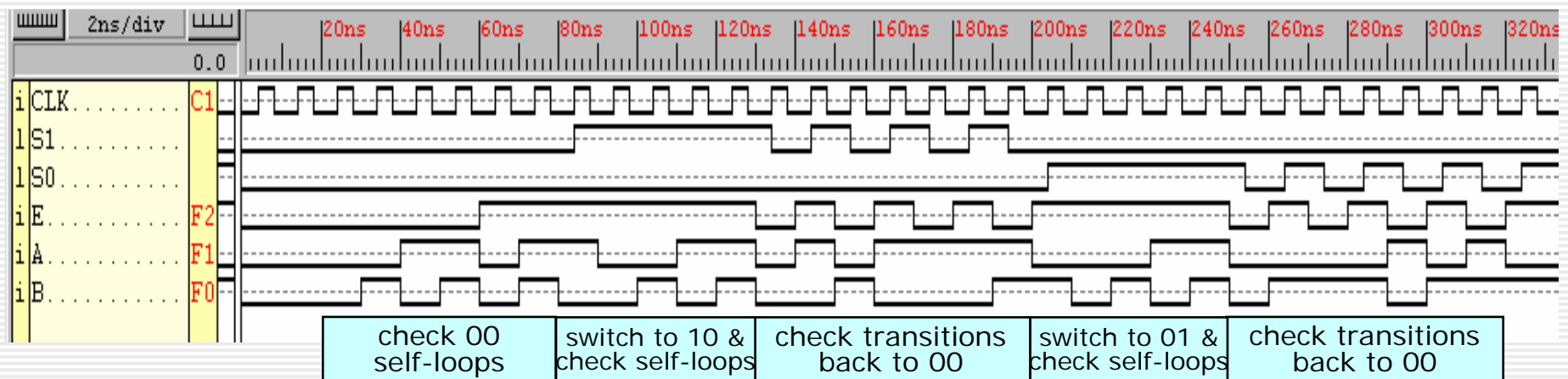


# Verifying Sequential Circuits

- To fully verify a sequential circuit, must check *all* state transitions (including “non-transitions”).
  - » use state diagram to plan input sequence
  - » for transitions with don't cares, check all possibilities



1. check all self-loops in 00.
2. switch to 10 and check self-loops
3. check transitions back to 00
4. switch to 01 and check self-loops
5. check transitions back to 00



# Timing Analysis of Sequential Circuits

---

- Determine if circuit subject to *internal* hold time violations; if so, eliminate by adding delay.
- Ignoring input signals, find smallest clock period for which setup time conditions are always met.
- Determine time periods (relative to clock edge) during which inputs must be stable.
- Determine time periods (relative to clock edge) when outputs may be changing (synchronous outputs).
- Input and output conditions used to ensure that connected sequential circuits interoperate correctly.
  - » if circuit *A* connects to circuit *B*, verify that output of *A* is not changing when *B* requires that its input be stable
  - » simplifies timing analysis of larger circuits

# Timing Analysis Procedure

## ■ Internal hold time violations.

» for every ff-to-ff path, check

$$(\textit{minimum} \text{ ff prop. delay}) + (\textit{minimum} \text{ comb. circuit delay}) \\ > (\text{hold time}) + (\text{clock skew})$$

omit skew for paths from output to input of same ff.

## ■ Minimum clock period

» find ff-to-ff path with largest value of

$$(\textit{maximum} \text{ ff prop. delay}) + (\textit{maximum} \text{ comb. circuit delay}) \\ + (\text{setup time}) + (\text{clock skew})$$

omit skew for paths from output to input of same ff.

## ■ Input timing analysis

» each input must be stable from

$$(\text{clock\_edge}) - ((\textit{maximum} \text{ input-to-ff delay}) + (\text{setup time})) \\ \text{to } ((\text{clock\_edge}) + (\text{hold time})) - (\textit{minimum} \text{ input-to-ff delay})$$

## ■ Output timing analysis

» outputs can change from

$$(\text{clock\_edge}) + (\textit{minimum} \text{ clock-to-output delay}) \\ \text{to } (\text{clock\_edge}) + (\textit{maximum} \text{ clock-to-output delay})$$

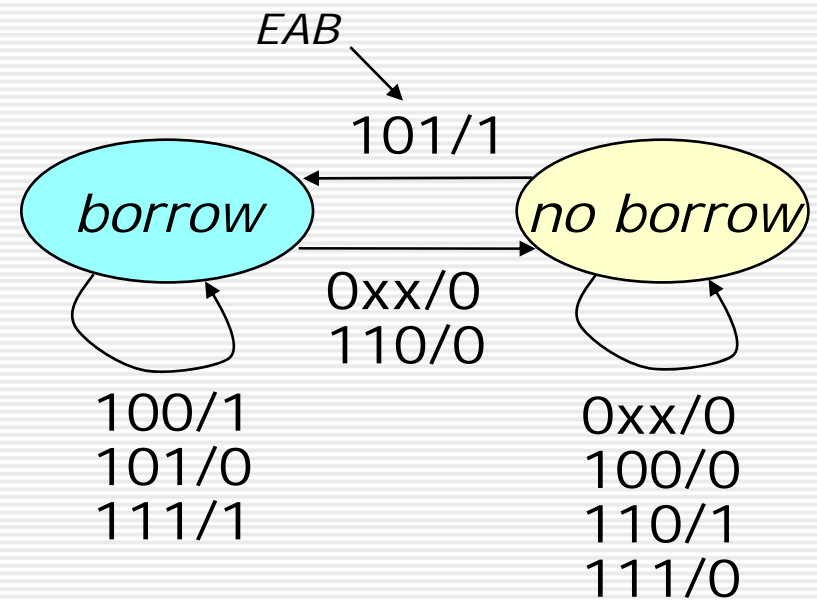
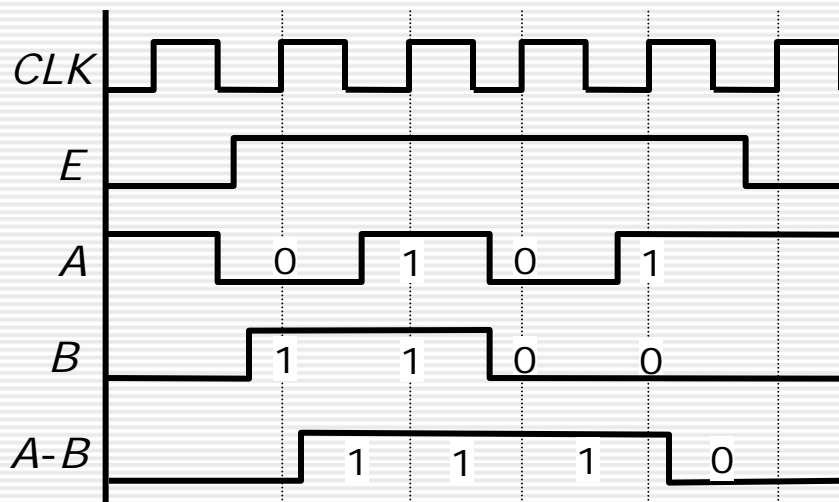
## ■ When combining circuits, check for possible timing violations.

» include *timing margin* that is at least equal to the clock skew

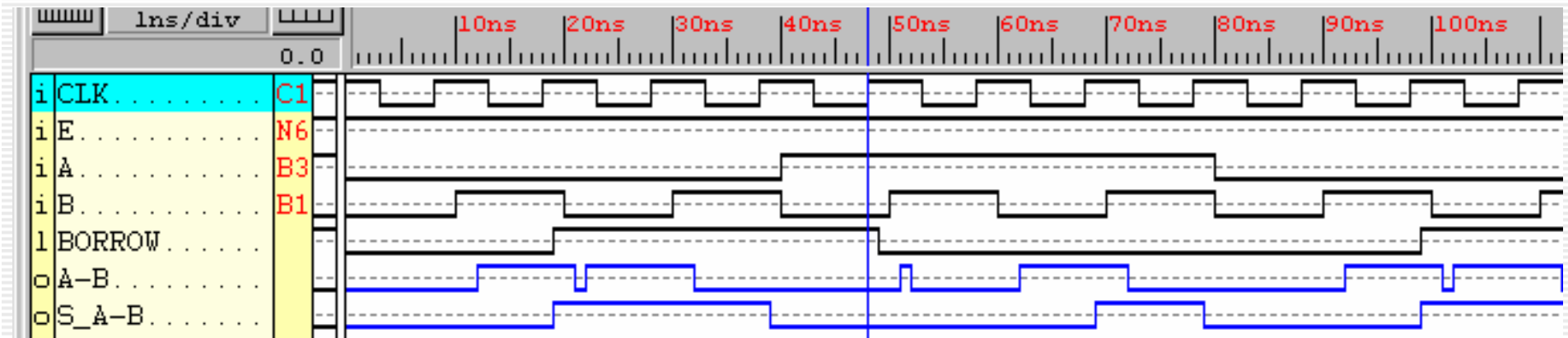
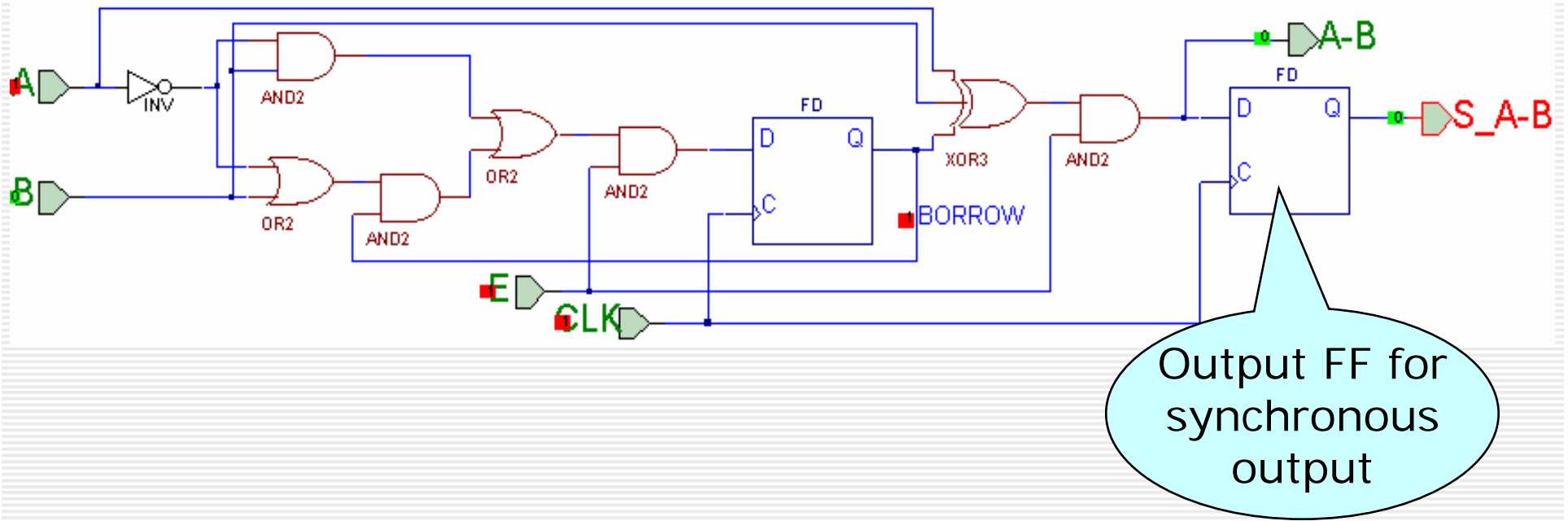


# Serial Subtraction Circuit

- A *serial subtraction circuit* has two data inputs ( $A, B$ ), an enable input ( $E$ ) and a single output ( $A-B$ ).
  - » when enable is low, the output is zero
  - » when enable is high, the circuit subtracts  $B$  from  $A$  numerically (assuming the values are presented with the least-significant bit, first) and outputs the difference, serially.
- Example:



# Serial Subtractor Design



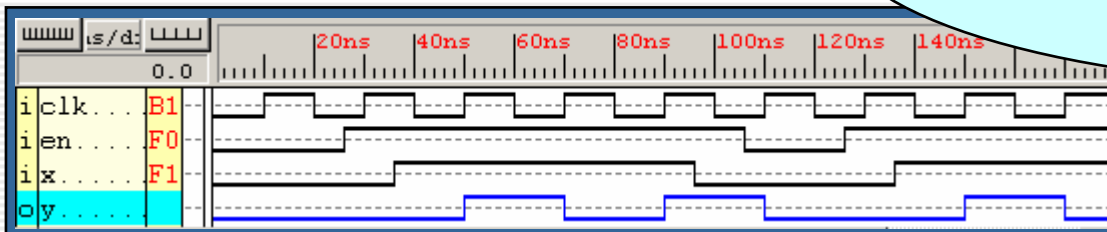
# Sequential Circuits in VHDL

```
entity sparity is port (  
    clk, en, x: in STD_LOGIC;  
    y: out STD_LOGIC );  
end sparity;  
architecture arch of sparityv is  
    signal s: std_logic;  
begin  
    process (clk) begin  
        if clk'event and clk = '1' then  
            if en = '0' then  
                s <= '0';  
            else  
                s <= x xor s;  
            end if;  
        end if;  
    end process;  
    y <= s;  
end arch;
```

Sensitivity list specifies signals that trigger changes.

Test for rising clock edge synchronizes actions.

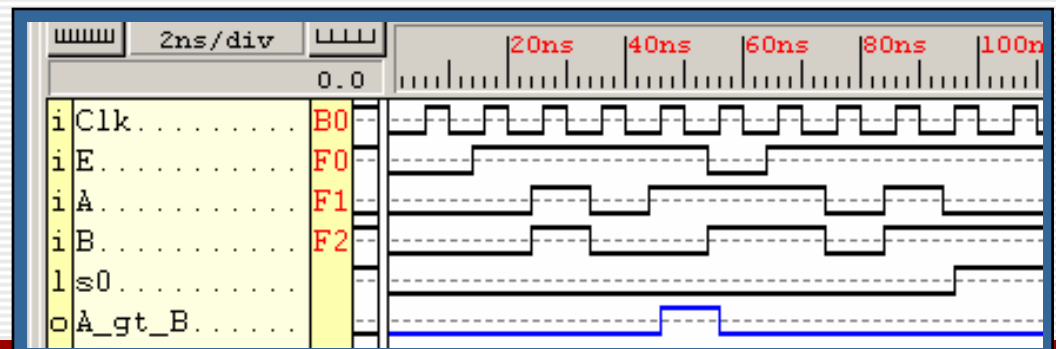
Edge-triggered flip flop implied by synchronous assignment



# Serial Comparator in VHDL

```
entity scompare is port (  
    A, B, E, Clk: in STD_LOGIC;  
    A_gt_B: out STD_LOGIC);  
end scompare;  
architecture arch of scompare is  
    signal s1, s0: STD_LOGIC;  
begin  
    process(clk) begin  
        if clk'event and clk = '1' then  
            if E = '0' then  
                s1 <= '0'; s0 <= '0';  
            elsif s1 = '0' and s0 = '0'  
                and A = '1' and B = '0' then  
                s1 <= '1'; s0 <= '0';  
            elsif s1 = '0' and s0 = '0'  
                and A = '0' and B = '1' then  
                s1 <= '0'; s0 <= '1';  
            end if;  
        end if;  
    end process;  
    A_gt_B <= s1;  
end arch;
```

- Same basic structure as serial parity circuit.
  - » signals for flip flops
  - » **if** defines next state logic
    - no change to s1, s0 when none of specified conditions holds
    - so, no code needed for self-loops in state diagram
  - » separate signal assignment for output



# Simpler Form of Seq. Comparator

```
entity seqcmpv is port (  
    A, B, E, Clk: in STD_LOGIC;  
    A_gt_B: out STD_LOGIC);  
end seqcmpv;  
architecture arch of seqcmpv is  
type state_type is (unknown, Abigger, Bbigger);  
signal state: state_type;  
begin  
    process(clk) begin  
        if clk'event and clk = '1' then  
            if E = '0' then  
                state <= unknown;  
            elsif state = unknown then  
                if A = '1' and B = '0' then  
                    state <= Abigger;  
                elsif A = '0' and B = '1' then  
                    state <= Bbigger;  
                end if;  
            end if;  
        end if;  
    end process;  
    A_gt_B <= '1' when state = Abigger else '0';  
end arch;
```

State type with  
named values.

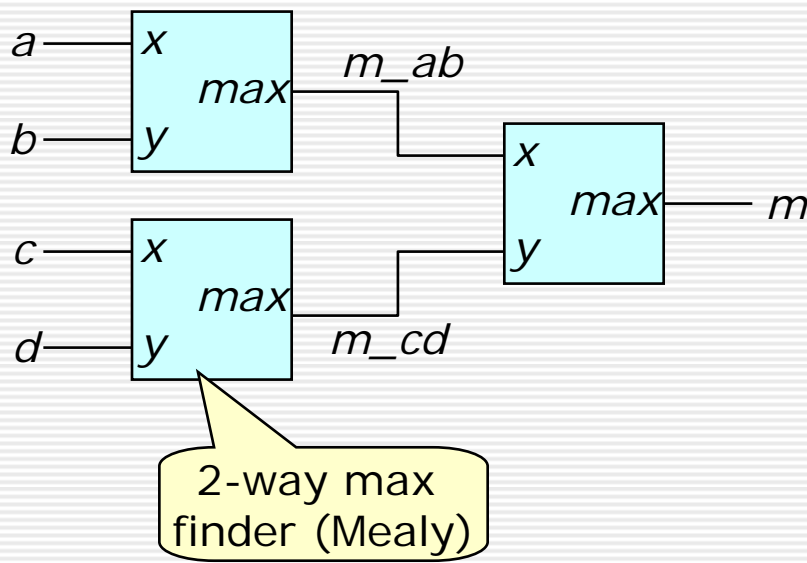
Use of state names  
makes code easier  
to understand.  
Synthesizer can  
optimize state  
assignment

# Recommended Practice for State Machines

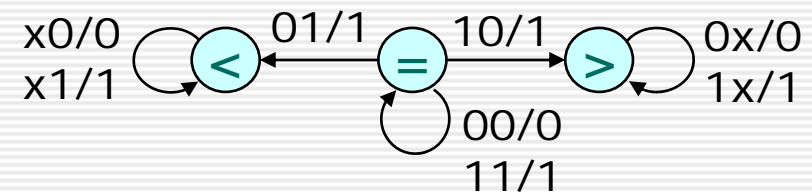
1. Determine the inputs and outputs of your circuit.
2. Determine and name the states in your circuit.
3. Create entity with all of inputs, outputs, reset and clock.
4. Create an enumerated state\_type with all your state names;  
example: type state\_type is (start, add, shift, wait)
5. Write process to update the state on each clock edge.  
process(clk) begin  
    if clk'event and clk = '1' then  
        if reset = '1' then -- or if EN = '0' then ...  
            state <= initial state;  
        elsif then  
            - - *add next state logic here*  
        end if;  
    end if;  
end process;
6. Outside process, write assignments for each output signal.  
    » for complex logic, use (separate) process for output signals

# 4-Way Max Finder Design

- Design a circuit with four serial inputs and a serial output equal to the largest input value (values come in msb first). Include reset and make output synchronous.
- Break down into three 2-way maximum circuits.
  - » note that 2-way max circuit works like comparator, but propagates largest value rather than simply determining which is largest



*Simplified state diagram for 2-way max finder*



# 4-Way Max Finder VHDL

```
entity max4 is port (  
    clk, reset : in std_logic;  
    a,b,c,d : in std_logic;  
    biggest : out std_logic);  
end max4;
```

```
architecture arch of max4 is
```

```
type stateType is (eq, lt, gt); -- states for 2-way comparator
```

```
function nextState(state:stateType; x,y:std_logic) return stateType is  
begin
```

```
    if state = eq and x > y then return gt;  
    elsif state = eq and x < y then return lt;  
    else return state;  
    end if;
```

```
end function nextState;
```

maxBit function used to  
define outputs for each  
of the 2-way max finders

nextState function used to  
define next states for each  
of the 2-way max finders

```
function maxBit(state: stateType; x,y: std_logic) return std_logic is  
begin
```

```
    if state = gt or (state = eq and x > y) then return x;  
    else return y;  
    end if;
```

```
end function maxBit;
```

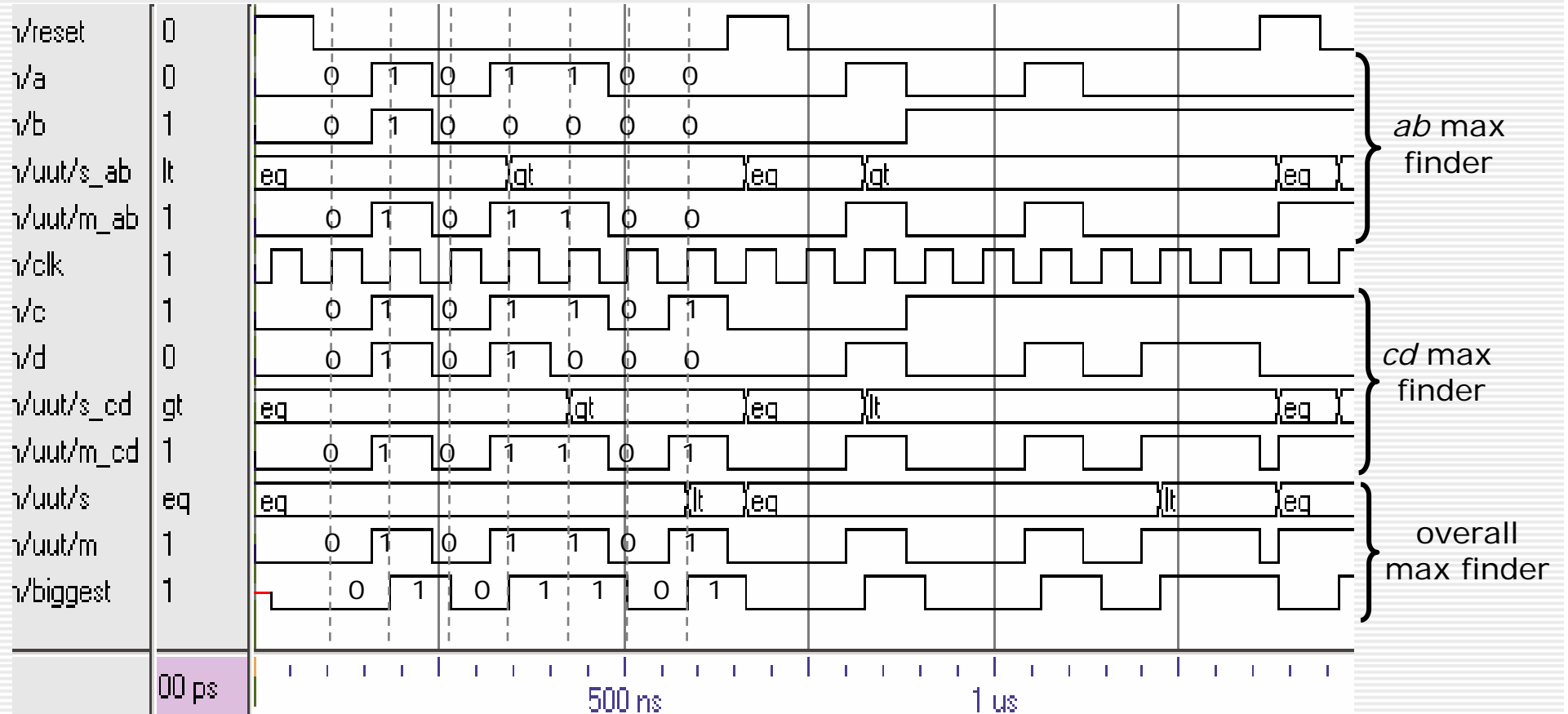
```
signal s_ab, s_cd, s: stateType;
signal m_ab, m_cd, m: std_logic;
begin
  m_ab <= maxBit(s_ab,a,b);
  m_cd <= maxBit(s_cd,c,d);
  m <= maxBit(s,m_ab,m_cd);
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then
        s_ab <= eq; s_cd <= eq; s <= eq;
        biggest <= '0';
      else
        s_ab <= nextState(s_ab,a,b);
        s_cd <= nextState(s_cd,c,d);
        s <= nextState(s,m_ab,m_cd);
        biggest <= m;
      end if;
    end if;
  end process;
end arch;
```

2-way max finder outputs are combinational function of current state and input values

synchronous updating of state signals

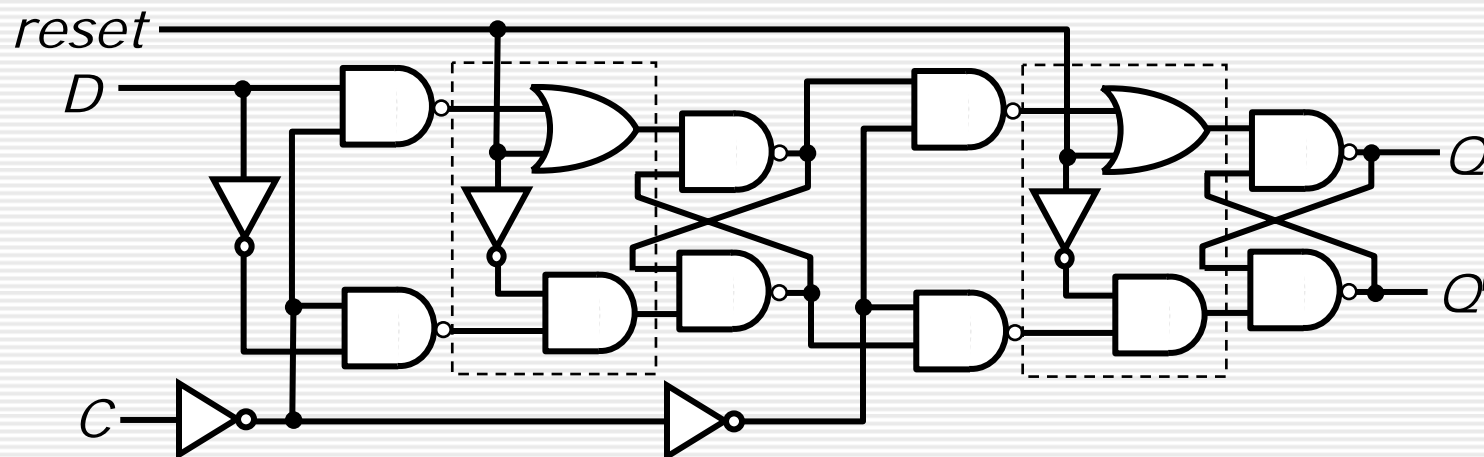
overall output assigned synchronously

# Simulation Results



# Flip Flops with Asynchronous Resets

- To simplify initialization, flip flops are often equipped with *asynchronous* resets.
  - » asynchronous resets clear flip flop independent of clock
- D flip flop with asynchronous reset.



- FPGAs often have asynchronous resets built-in
  - » to use built-in reset, VHDL must be written differently
  - » caveat: using async. reset makes design less "portable"

# Asynchronous Resets in VHDL

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity sparityv is
5      port (
6          clk, reset, x: in STD_LOGIC;
7          y: out STD_LOGIC
8      );
9  end sparityv;
10
11 architecture sparityv_arch of sparityv is
12     signal s: std_logic;
13     begin
14         process(clk,reset) begin
15             if reset = '1' then
16                 s <= '0';
17             elsif clk'event and clk = '1' then
18                 s <= x xor s;
19             end if;
20         end process;
21         y <= s;
22     end sparityv_arch;
```

process responds to changes in clk and reset

initialization does not depend on clk

normal state changes only allowed when reset=0