

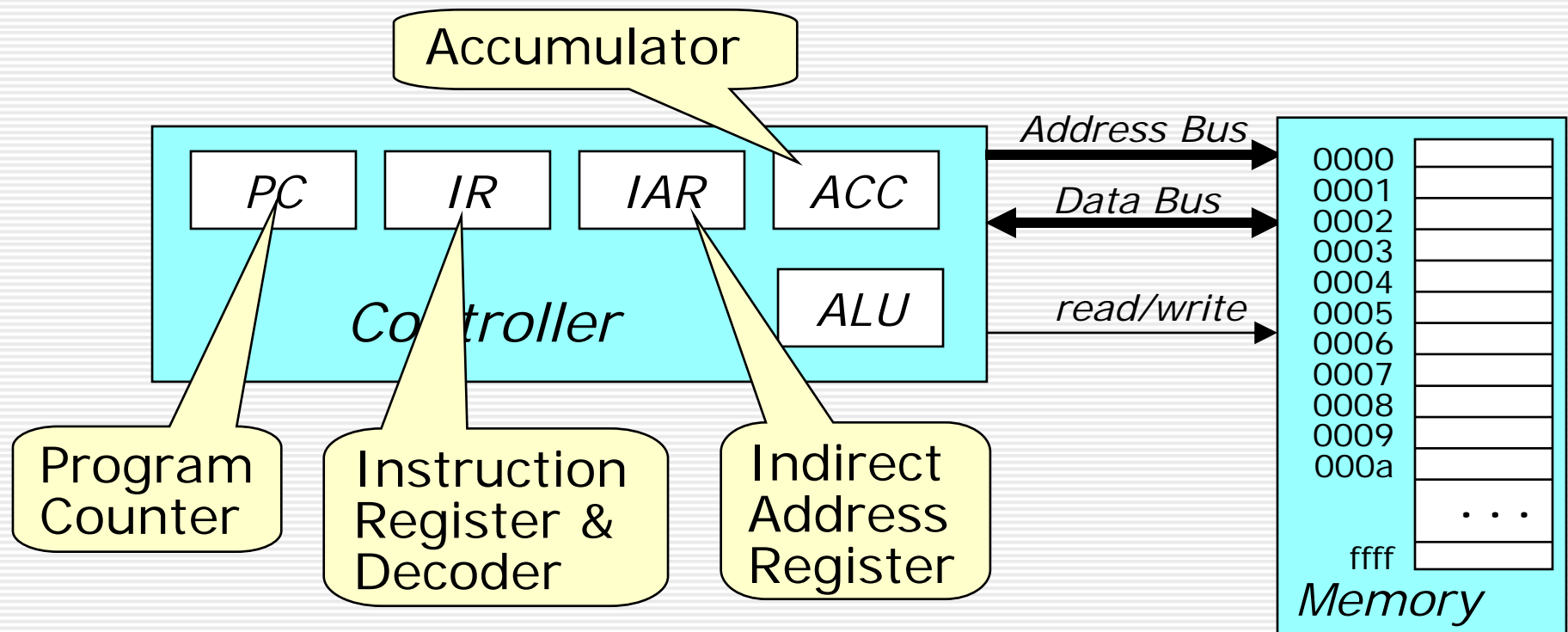
# Sequential Circuit Design

- Registers and Counters  
(W 691-694, 697-701, 710-758)
- Complex Sequential Circuits  
(W 758-787)

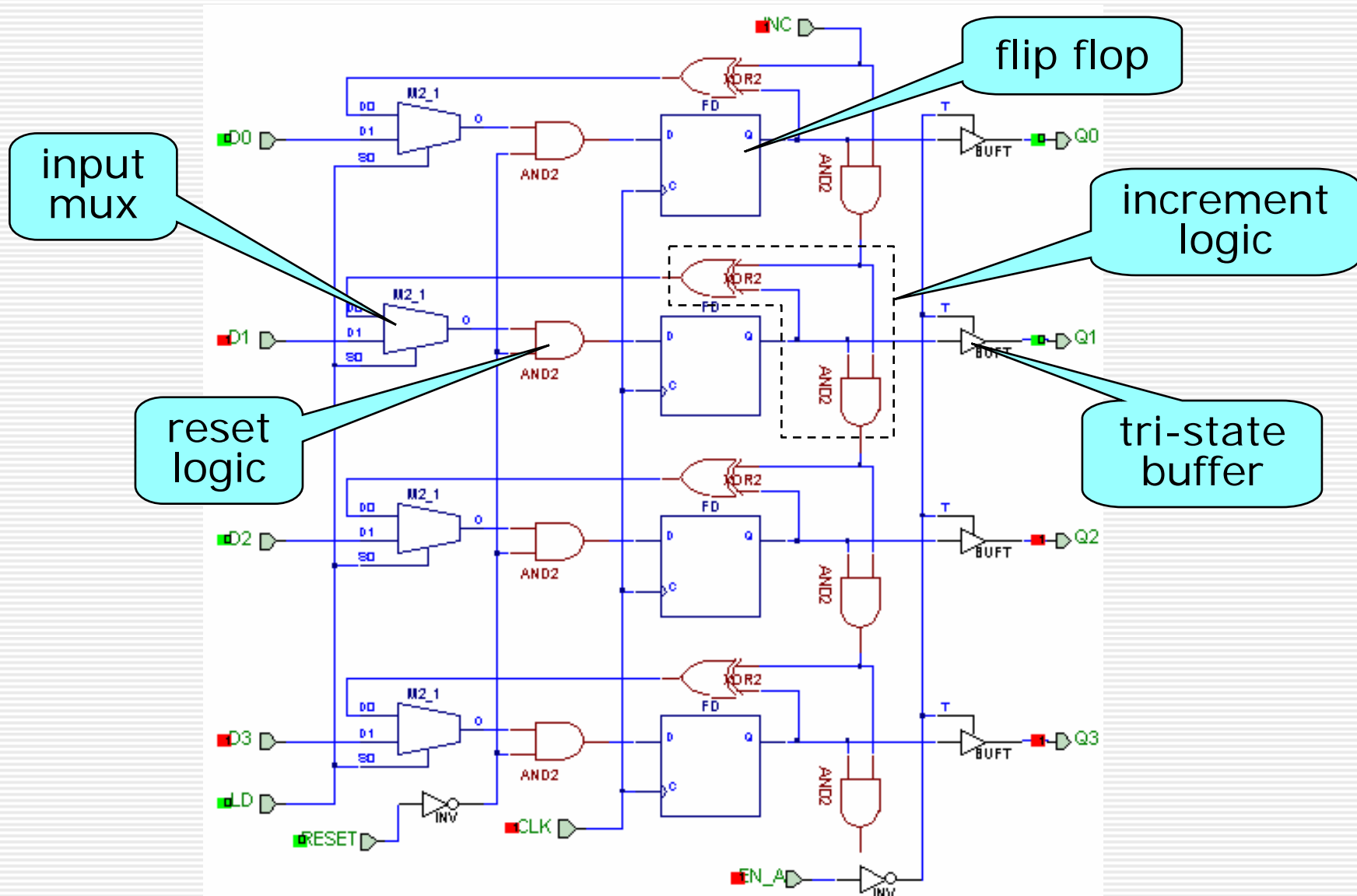


# Registers in the Basic Computer

- *Registers* are basic building blocks in digital systems.
  - » store information
  - » auxiliary circuits may modify stored information or “steer it” to and from register



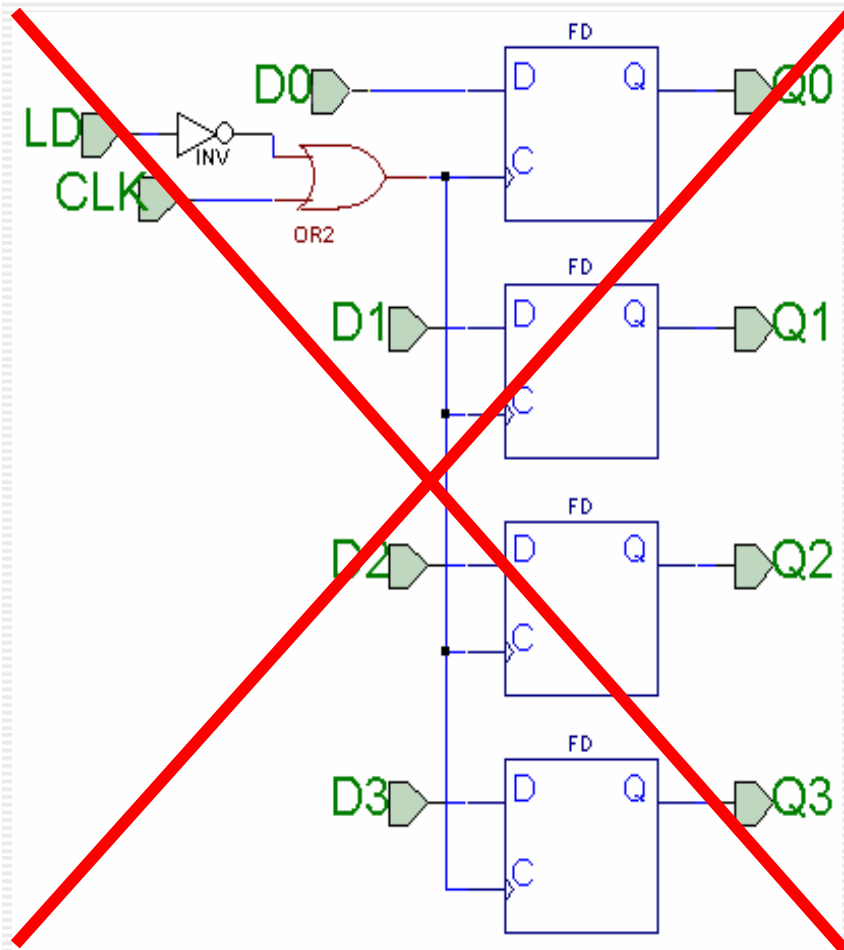
# Program Counter Schematic (4 bit)



# Registers and Counters

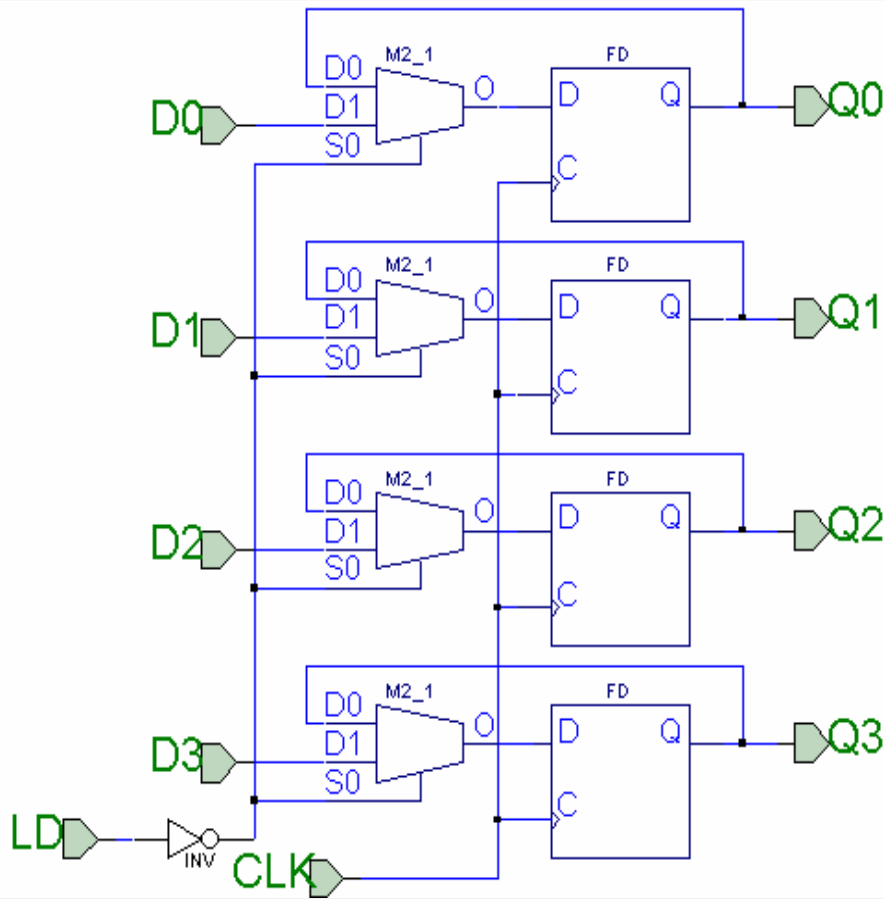
- A *register* is a set of flip flops, often supplemented by additional circuits to control input and output.
  - » can have parallel I/O or serial I/O or combination
- Usually, used to store a set of related bits.
  - » bits that collectively represent an integer value
  - » bits of an ASCII character code
  - » status bits for a device in a computer system (disk controller)
- *Counters* are registers that store numeric values along with circuits to increment/decrement the stored value.
  - » up-counters, down-counters, up-down counters
  - » generalized counters
    - BCD counters, gray-code counters, ...

# Simple Parallel Load Register



- Four bit register.
  - » if LD is high when clock rises, new values are stored
  - » LD should drop only while CLK is high
- Registers using *gated clocks* can lead to timing problems.
  - » increases *clock skew*
  - » may lead to violations of flip flop setup, hold time specs
  - » extra care needed to ensure correct operation
  - » safer to avoid clock gating whenever possible

# Preferred Parallel Load Register



- Multiplexor for each register bit.
  - » new value loaded when LD is high
  - » otherwise, old value stored
- No gated clock, minimizing *clock skew*.
  - » simplifies checking of setup and hold time specs.
  - » can focus on delays between connected flip flops
- Increases gate count by about 30%.

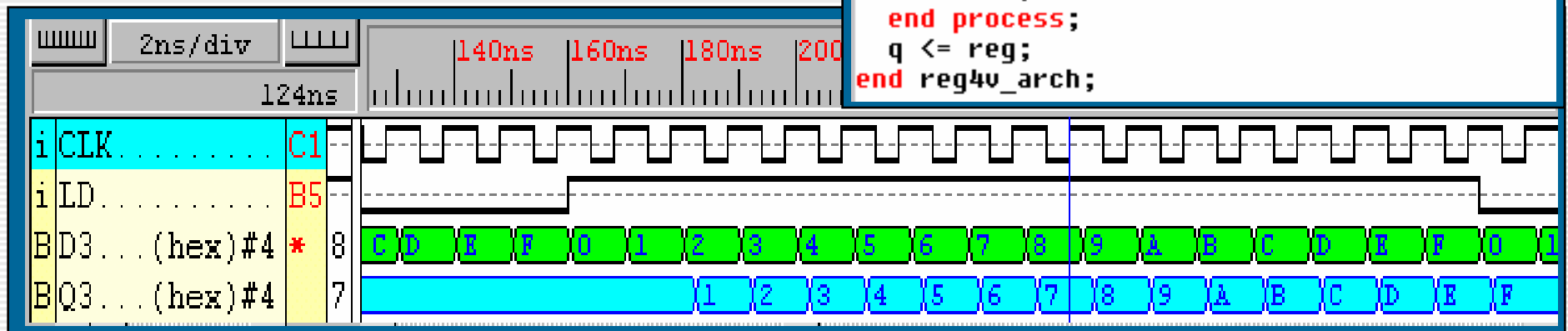
# VHDL Specification for Register

- Register stores new input value when **ld** is high.
- Otherwise, retains old value.

```
library IEEE;
use IEEE.std_logic_1164.all;

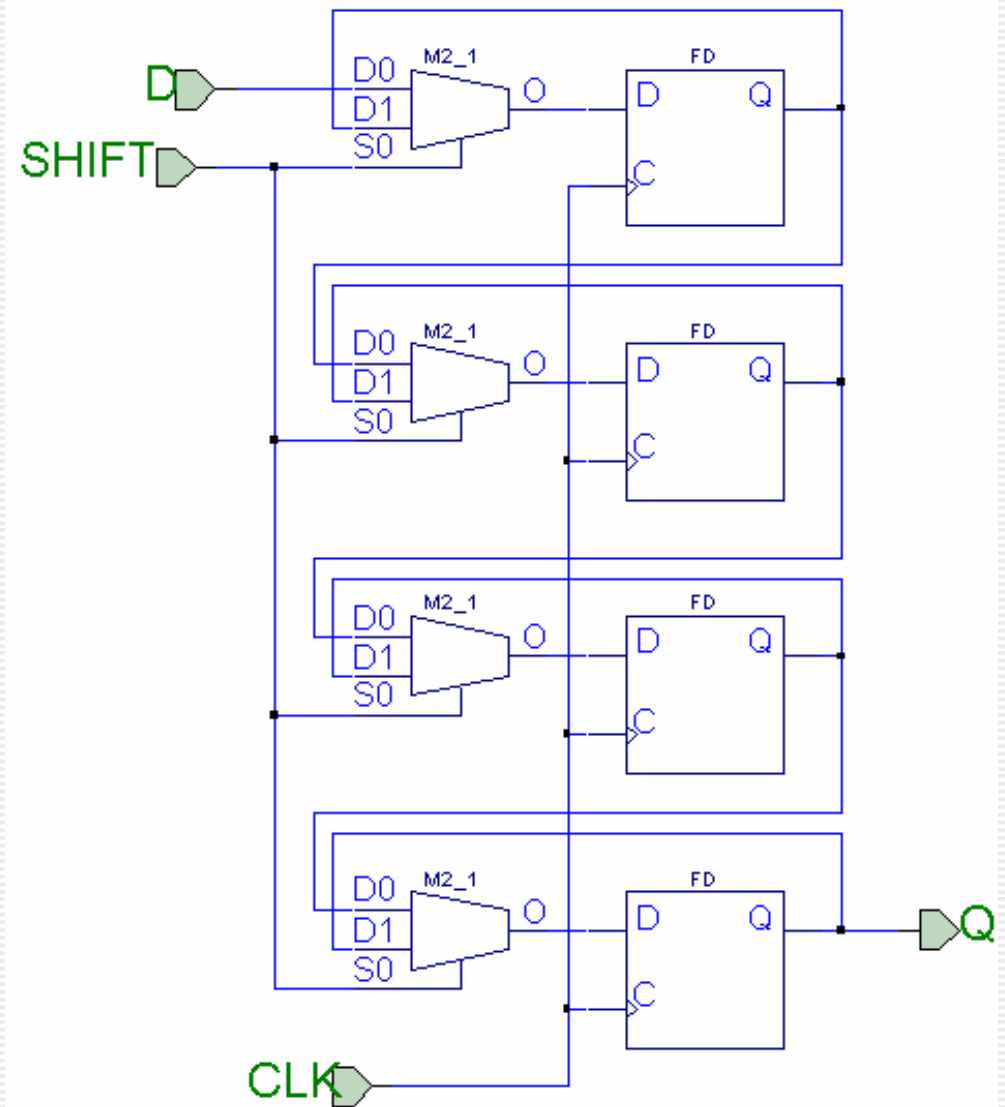
entity reg4v is
  port (
    d: in STD_LOGIC_VECTOR(3 downto 0);
    ld, clk: in STD_LOGIC;
    q: out STD_LOGIC_VECTOR(3 downto 0)
  );
end reg4v;

architecture reg4v_arch of reg4v is
  signal reg: STD_LOGIC_VECTOR(3 downto 0);
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if ld = '1' then
        reg <= d;
      end if;
    end if;
  end process;
  q <= reg;
end reg4v_arch;
```



# Shift Registers

- Shift when **SHIFT** is low.
- Shift registers support serial input and output.
  - » useful for communication over serial channels
- With parallel outputs, can be used for serial-to-parallel conversion.
- With parallel inputs can be used for parallel-to-serial conversion.
  - » requires 3 input muxes and second control input



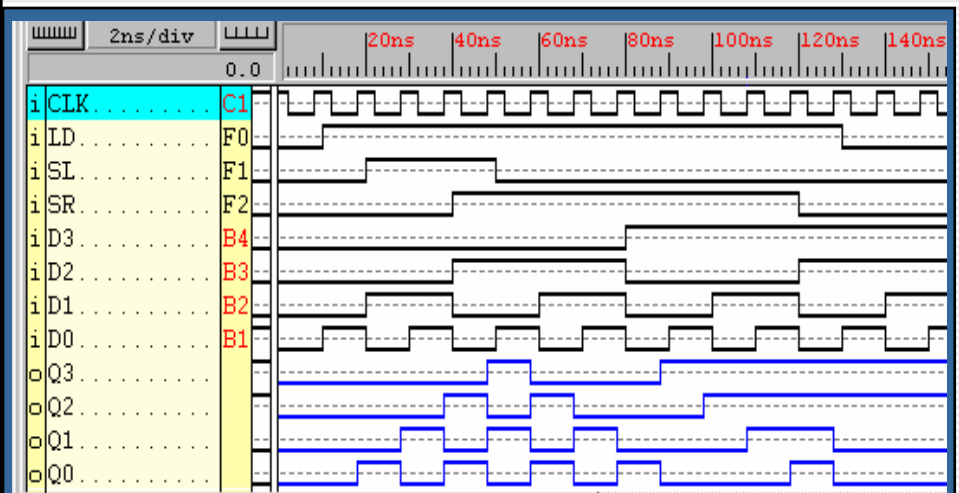
# VHDL for Bidirectional Shift Register

```
library IEEE;
use IEEE.std_logic_1164.all;

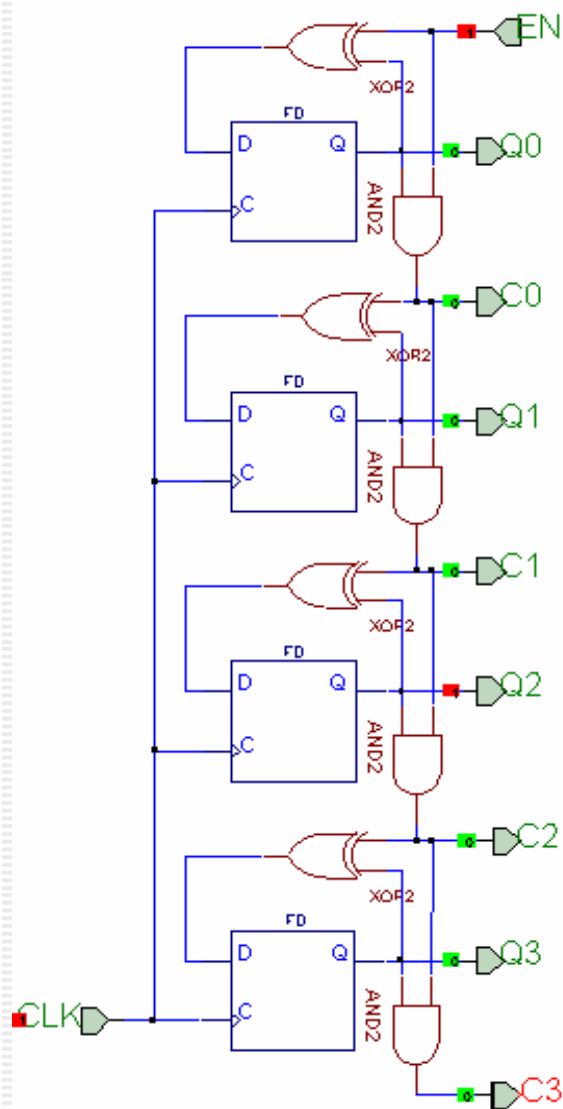
entity sreg is
  port (
    d: in STD_LOGIC_VECTOR(3 downto 0);
    ld, sl, sr: in STD_LOGIC;
    clk: in STD_LOGIC;
    q: out STD_LOGIC_VECTOR(3 downto 0)
  );
end sreg;

architecture sreg_arch of sreg is
  signal reg: STD_LOGIC_VECTOR(3 downto 0);
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if ld = '1' then
        if sl = '1' then
          reg <= reg(2 downto 0) & d(0);
        elsif sr = '1' then
          reg <= d(3) & reg(3 downto 1);
        else
          reg <= d;
        end if;
      end if;
    end if;
  end process;
  q <= reg;
end sreg_arch;
```

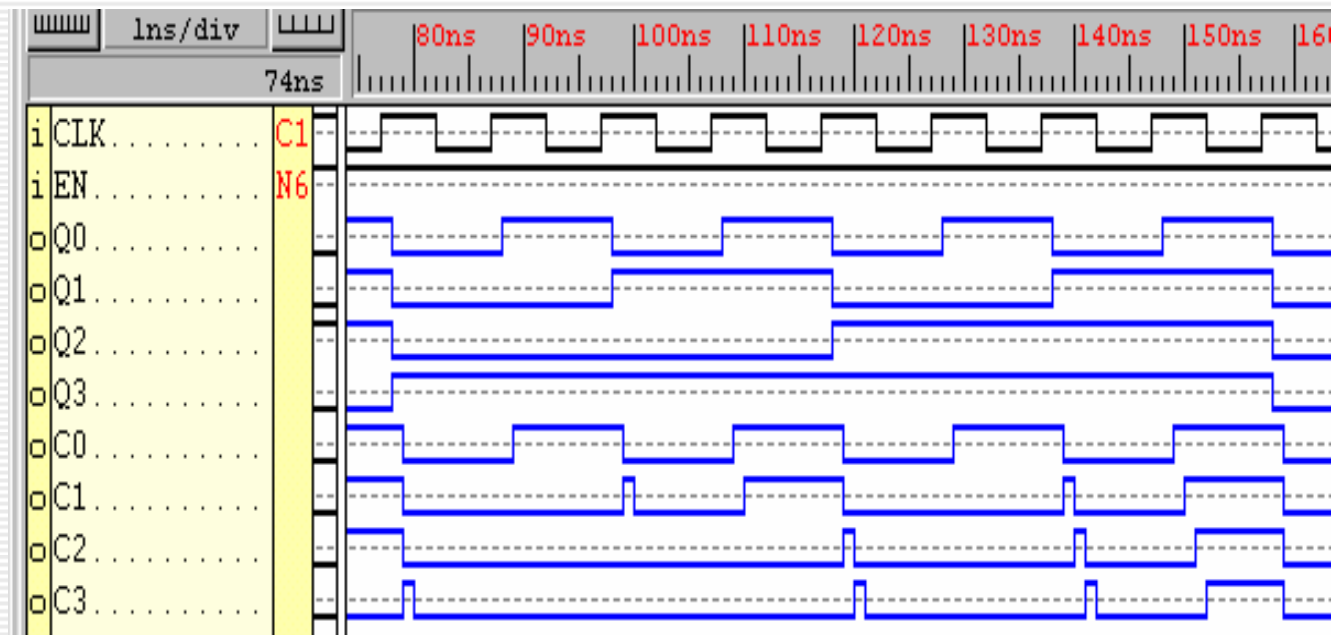
- **ld** enables loading.
  - » if **sl** asserted then left shift
  - » else if **sr** asserted then right
  - » else parallel load



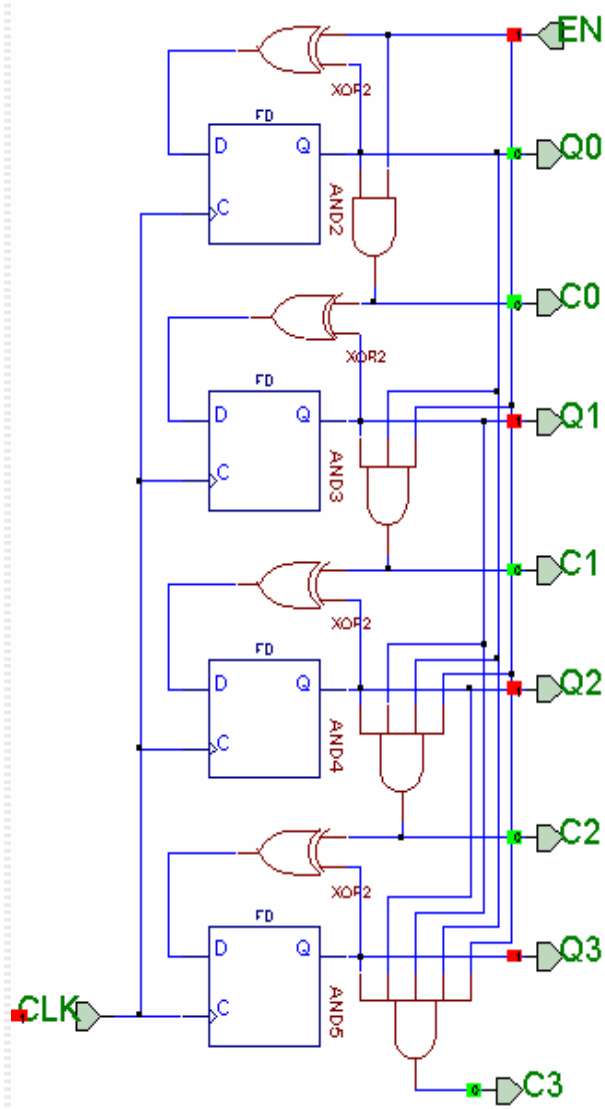
# Synchronous Ripple Carry Counter



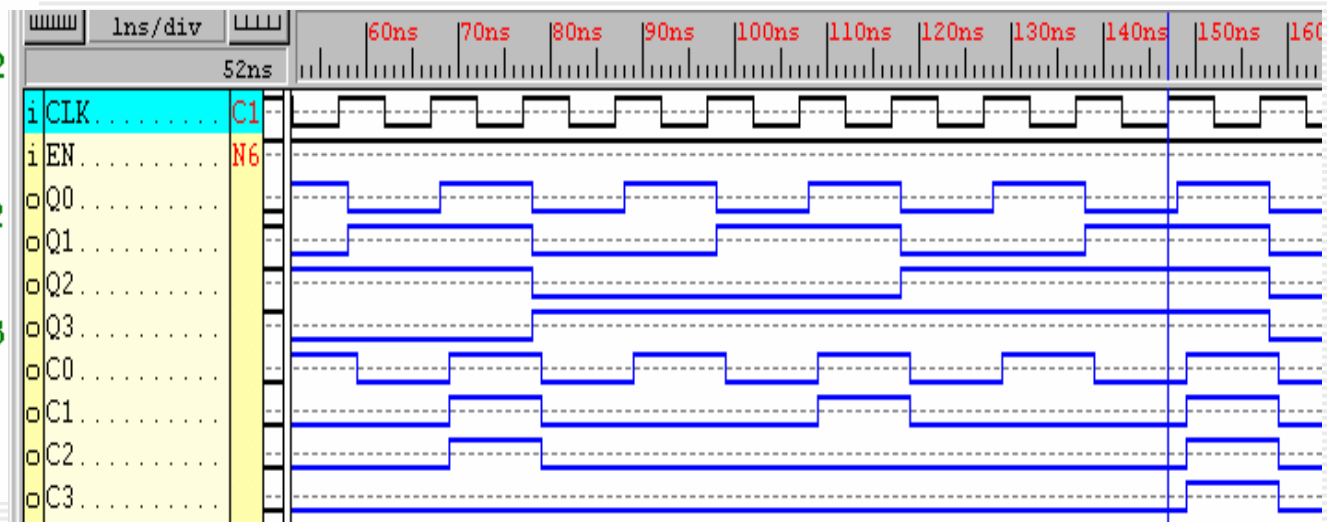
- Change in low order bit can affect carry in all higher bits.
- No problem, so long as carry stable by next rising clock edge.
- Can be too slow for counters with many bits.



# Counter with Carry Look-ahead



- Carries sent forward to eliminate carry propagation delay.
- In large counters, carry logic becomes major part of counter complexity.
- Large fanout of carry signals limit performance gains.
- Scalable carry-lookahead incrementer better choice for large  $n$ .



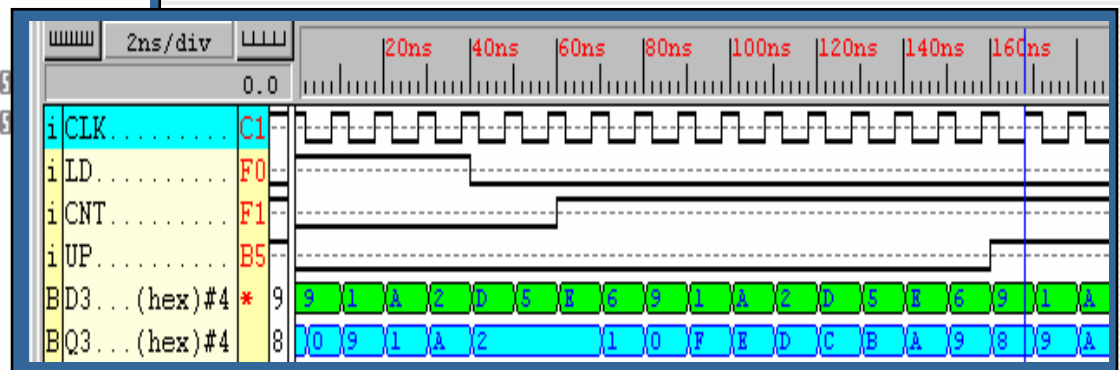
# Up-down Counter with Parallel Load

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ud_cntr is
  port (
    d: in STD_LOGIC_VECTOR(3 downto 0);
    ld, cnt, up, clk: in STD_LOGIC;
    q: out STD_LOGIC_VECTOR(3 downto 0)
  );
end ud_cntr;

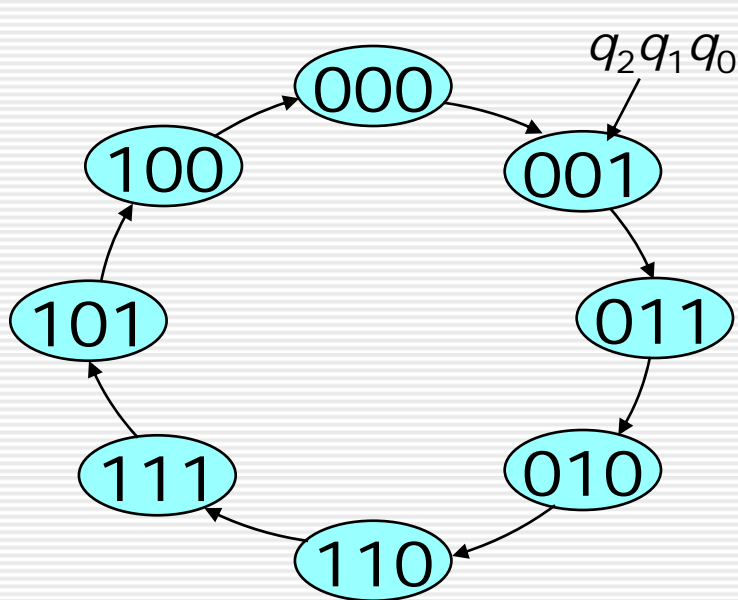
architecture ud_cntr_arch of ud_cntr is
  signal reg: STD_LOGIC_VECTOR(3 downto 0);
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if ld = '1' then
        reg <= d;
      elsif cnt = '1' then
        if up = '1' then
          reg <= reg + "0001";
        else
          reg <= reg - "0001";
        end if;
      end if;
    end if;
  end process;
  q <= reg;
end ud_cntr_arch;
```

- Operations
  - » load (when ld high)
  - » count up (if ld low, cnt, up high)
  - » count down (if ld low, cnt high, up low)
- Synthesizer generates carry logic.
  - » can optimize for either size or speed



# Non-Standard Counters

- Counters are sometimes defined that count in an order other than standard numerical order.
- The state machine below is for a *gray code* counter in which one bit changes at a time.



		$q_1 q_0$			
		00	01	11	10
$q_2$	0	1	1	0	0
	1	0	0	1	1

		$q_1 q_0$			
		00	01	11	10
$q_2$	0	0	1	1	1
	1	0	0	0	1

		$q_1 q_0$			
		00	01	11	10
$q_2$	0	0	0	0	1
	1	0	1	1	1

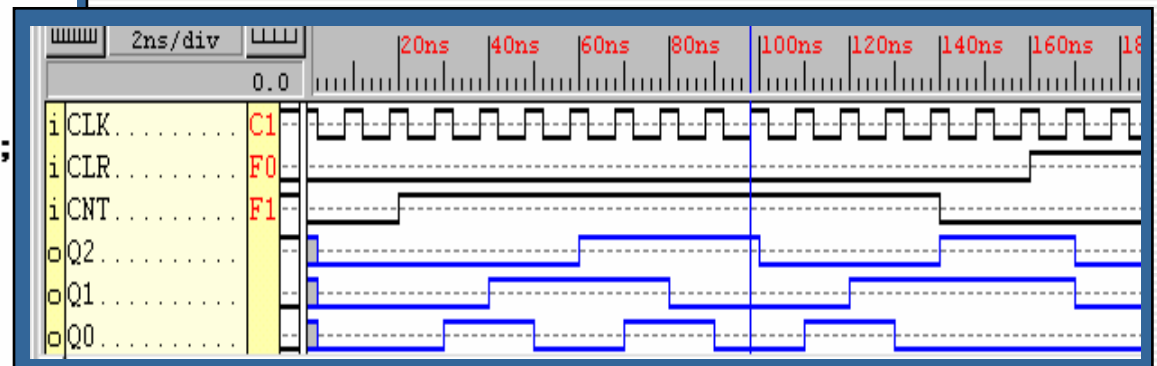
# VHDL for Gray Code Counter

```
library IEEE;
use IEEE.std_logic_1164.all;

entity gray_cnt is
  port (
    clr, cnt, clk: in STD_LOGIC;
    q: out STD_LOGIC_VECTOR(2 downto 0)
  );
end gray_cnt;

architecture gray_cnt_arch of gray_cnt is
  signal reg: STD_LOGIC_VECTOR(2 downto 0);
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if clr = '1' then reg <= "000";
      elsif cnt = '1' then
        case reg is
          when "000" => reg <= "001";
          when "001" => reg <= "011";
          when "011" => reg <= "010";
          when "010" => reg <= "110";
          when "110" => reg <= "111";
          when "111" => reg <= "101";
          when "101" => reg <= "100";
          when "100" => reg <= "000";
          when others => reg <= "000";
        end case;
      end if;
    end if;
  end process;
  q <= reg;
end gray_cnt_arch;
```

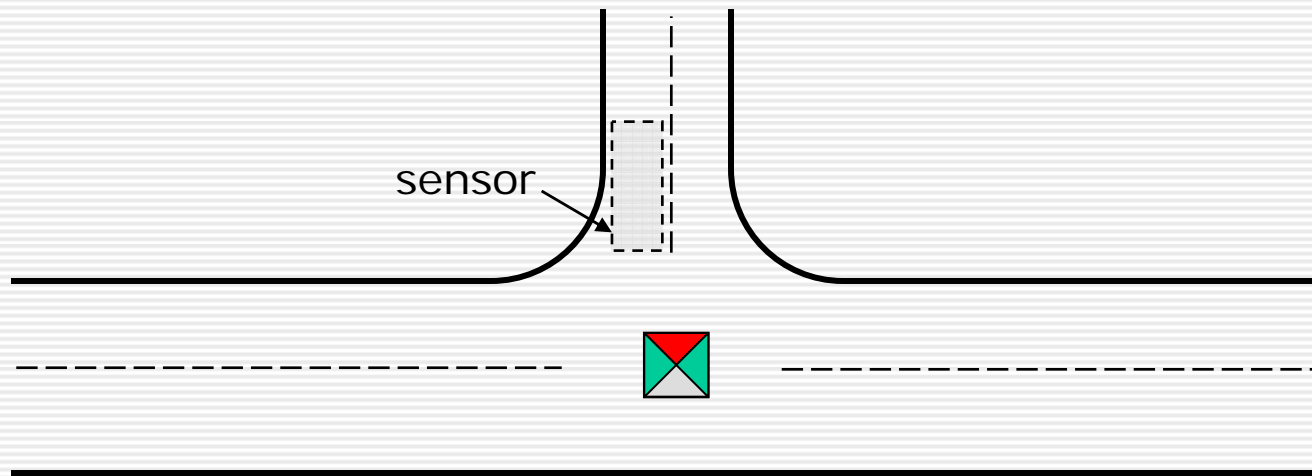
- Assignments in case directly reflect the state diagram.
- By modifying assignments, can produce any non-standard counting order.
- Implementation uses mux controlled by current value.



# Designing Complex Circuits

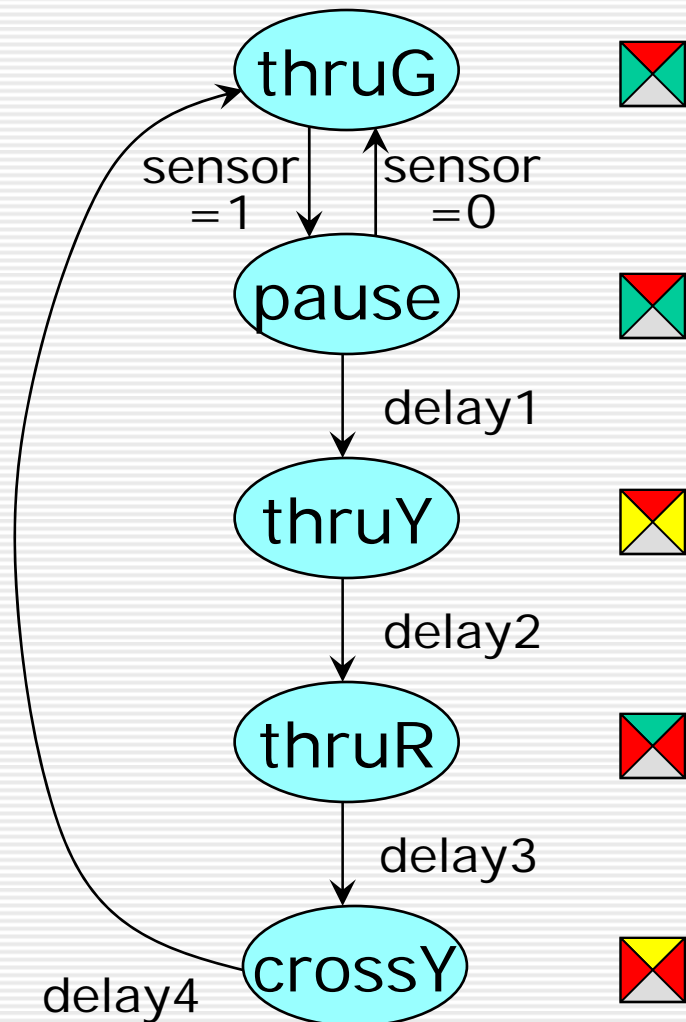
- Determine what the circuit must “remember.”
  - » may include data state
    - stored in data registers
  - » may include control state
    - defines steps in process implemented by the circuit
    - stored in state register
- Define high level state diagram.
  - » state transitions defined among different control states
  - » conditions defined on data in registers may determine control state transitions
    - e.g. if count = limit and increment = 1 goto overflow\_state
  - » state transitions may trigger actions affecting stored data
    - e.g. clear count and set overflow error bit
- In VHDL can often develop code directly from state diagram.
- For schematic design, treat control state machine as separate sequential circuit and define inputs and outputs for required conditions and actions.

# Traffic Light Controller



- T intersection.
- Default to green on main road.
- Sensor enables green for cross street.
- Delay switching for right-turn-on-red from cross street.
- Programmable delays.

# High Level State Machine



- Stay in *thruG* state until sensor is activated.



- Wait in *pause* state to see if sensor deactivates. (right-turn-on-red)



- Then proceed through sequence, waiting in each state for specified time delay.

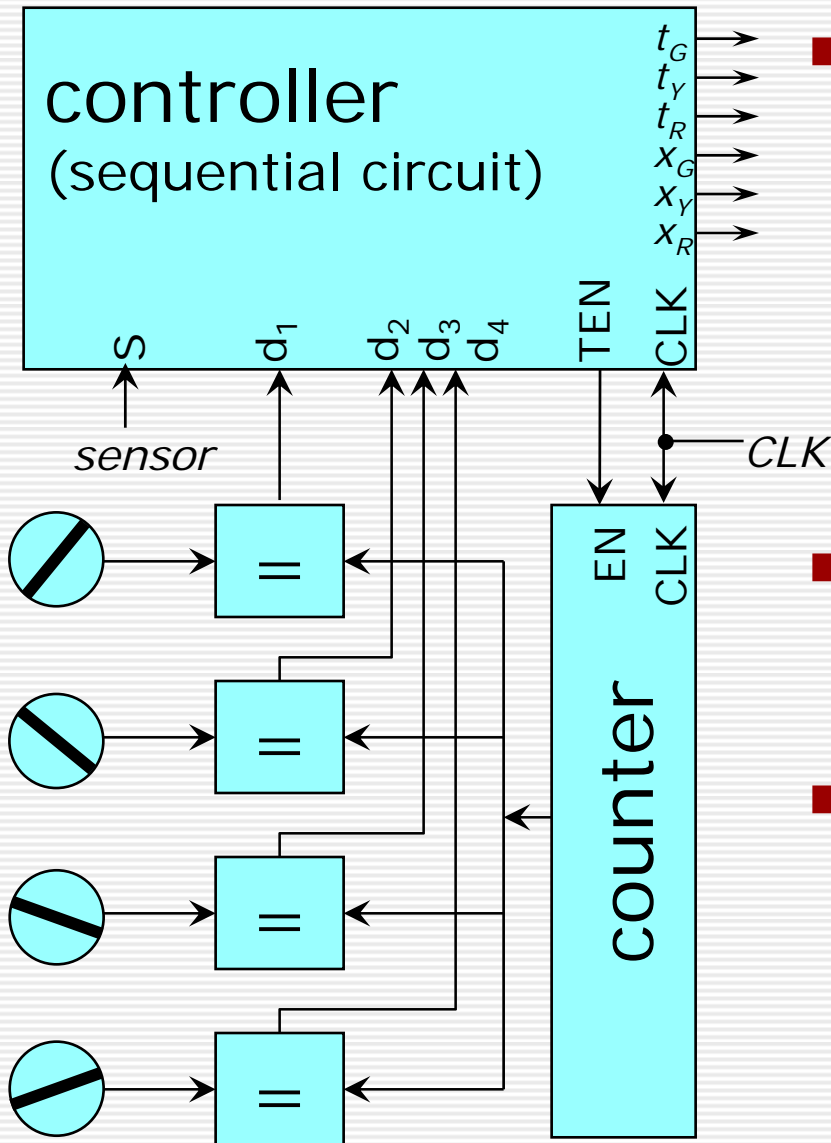


- In each state, provide appropriate control signals for lights.



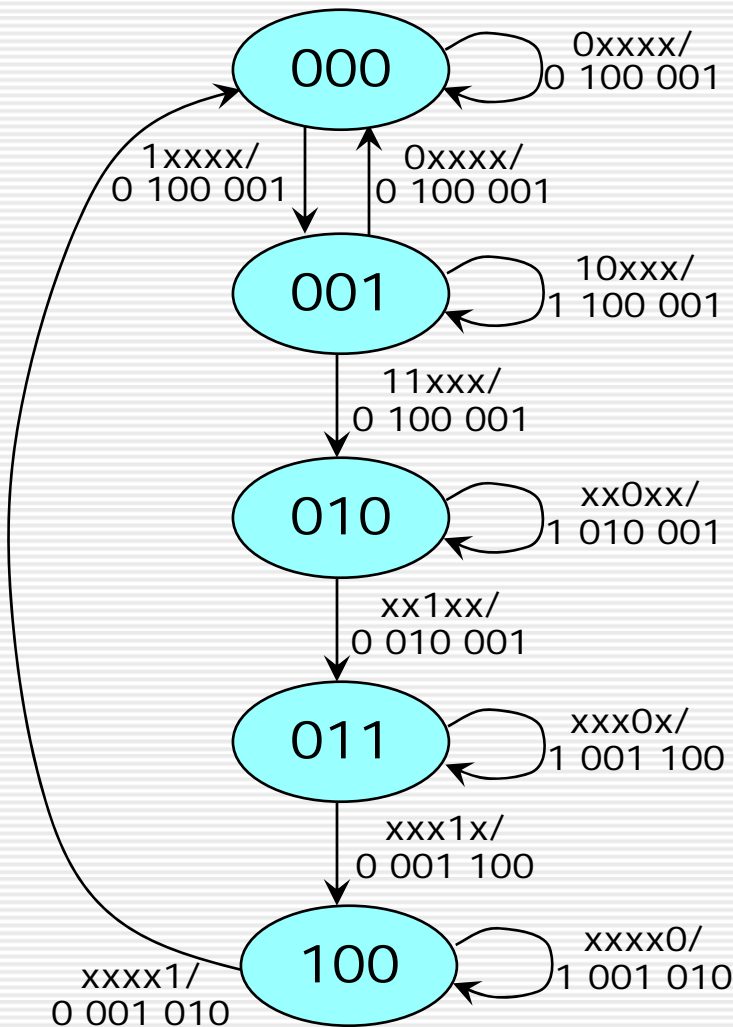
- 5 states, so at least 3 flip flops.

# Block Diagram



- Controller keeps track of state and turns lights on/off.
  - »  $t_G=1$  means thruGreen on, etc.
  - » state assignment
    - 000 = thruG, 001 = pause,
    - 010 = thruY, 011 = thruR,
    - 100 = crossY
- Counter used to regulate delays.
  - » set to zero when not enabled
- Dials at left specify delays.
  - » note how clock frequency affects delay values and counter size

# Detailed State Diagram and State Table



current state	inputs	outputs	next state
$s_2s_1s_0$	$Sd_1d_2d_3d_4$	$TEN t_Gt_Yt_R X_GX_YX_R$	$ns_2ns_1ns_0$
000	0 xxxx	0 100 001	000
000	1 xxxx	0 100 001	001
001	0 xxxx	0 100 001	000
001	1 0xxx	1 100 001	001
001	1 1xxx	0 100 001	010
010	x x0xx	1 010 001	010
010	x x1xx	0 010 001	011
011	x xx0x	1 001 100	011
011	x xx1x	0 001 100	100
100	x xxx0	1 001 010	100
100	x xxx1	0 001 010	000

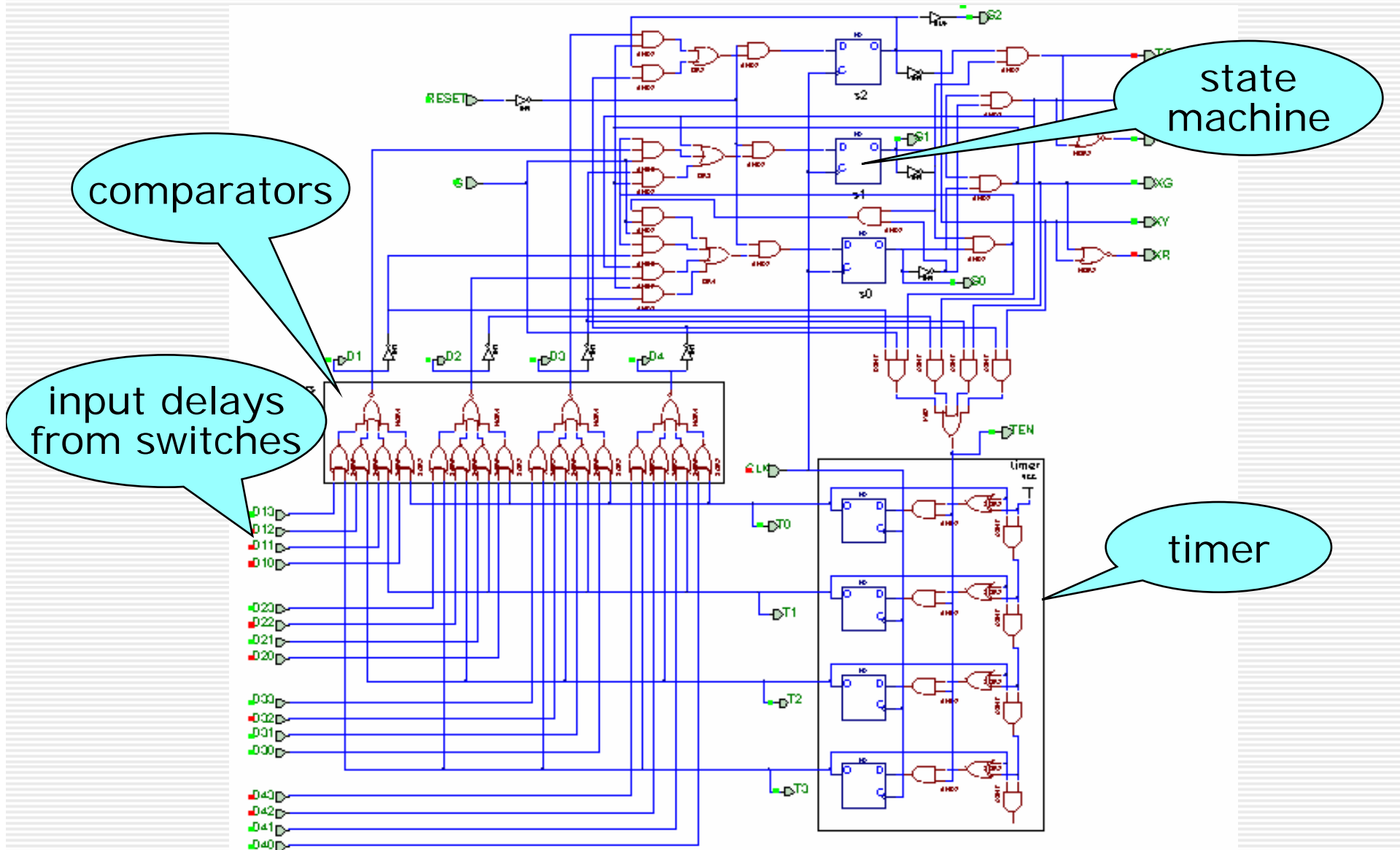
## Output equations

$$\begin{aligned}
 t_G &= s_2's_1' & t_Y &= s_1s_0' & t_R &= (t_G + t_Y)' \\
 X_G &= s_1s_0 & X_Y &= s_2 & X_R &= (X_G + X_Y)' \\
 TEN &= s_1's_0Sd_1' + s_1s_0'd_2' + s_1s_0d_3' + s_2d_4'
 \end{aligned}$$

## Next state equations

$$\begin{aligned}
 ns_2 &= s_1s_0d_3 + s_2d_4' \\
 ns_1 &= s_1's_0Sd_1 + s_1s_0' + s_1s_0d_3' \\
 ns_0 &= s_2's_1's_0'S + s_1's_0Sd_1' + s_1s_0'd_2 + s_1s_0d_3'
 \end{aligned}$$

# Schematic for Traffic Light Controller





# VHDL for Traffic Light Controller

```
entity trafficv is
  port (
    reset, sensor, CLK: in STD_LOGIC;
    d1, d2, d3, d4: in unsigned(wordSize-1 downto 0);
    tG, tY, tR, xG, xY, xR: out STD_LOGIC
  );
end trafficv;
```

unsigned type for  
numeric values – subtype  
of std\_logic\_vector

```
architecture trafficv_arch of trafficv is
  type state_type is (thruG, pause, thruY, thruR, crossY);
  signal state: state_type;
  signal timer: unsigned(wordSize-1 downto 0);
begin
  next_state_process:
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then
        state <= thruG; timer <= (timer'range => '0');
      else
        case state is
          when thruG =>
            if sensor = '1' then state <= pause; end if;
```

use of range attribute  
makes assignment  
independent of length

# VHDL for Traffic Light Controller

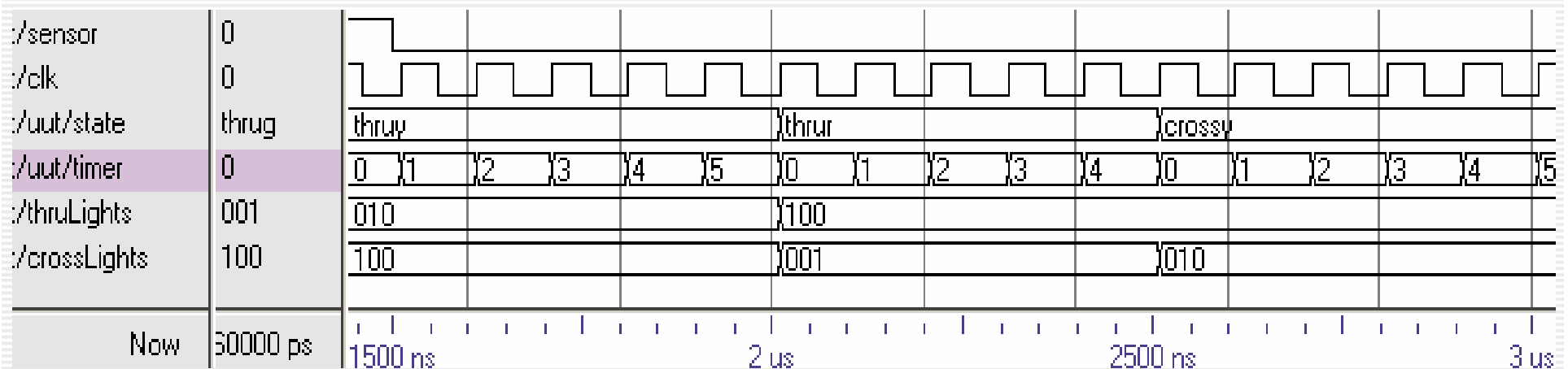
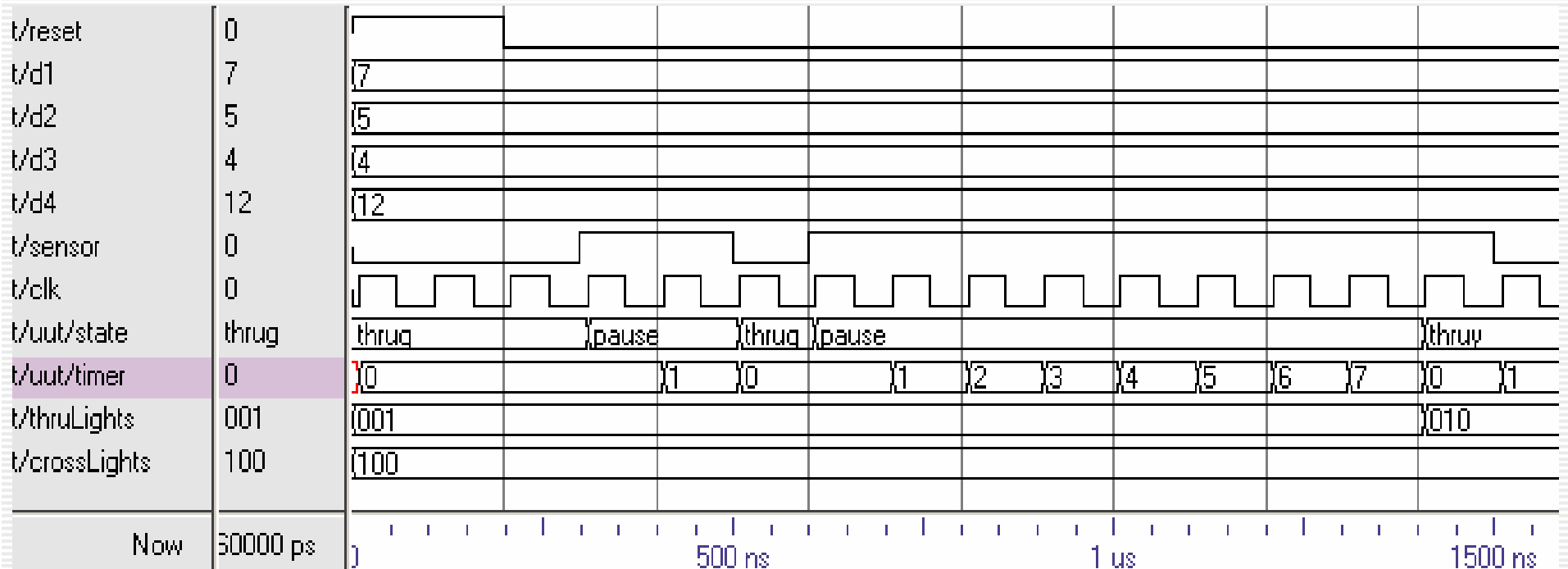
```
when pause =>
    if sensor = '0' then
        state <= thruG; timer <= (timer'range => '0');
    else
        if timer /= d1 then timer <= timer + "1";
        else state <=thruY; timer<=(timer'range=>'0'); end if;
    end if;
when thruY =>
    if timer /= d2 then timer <= timer + "1";
    else state <=thruR; timer<= (timer'range => '0'); end if;
when thruR =>
    if timer /= d3 then timer <= timer + "1";
    else state <=crossY; timer <=(timer'range =>'0'); end if;
when crossY =>
    if timer /= d4 then timer <= timer + "1";
    else state <=thruG; timer <=(timer'range => '0'); end if;
end case;
end if;
end if;
end process;
```

# VHDL for Traffic Light Controller

```
output_process:
  process (state) begin
    tG <= '0'; tY <= '0'; tR <= '0';
    xG <= '0'; xY <= '0'; xR <= '0';
    case state is
      when thruG => tG <= '1'; xR <= '1';
      when pause => tG <= '1'; xR <= '1';
      when thruY => tY <= '1'; xR <= '1';
      when thruR => tR <= '1'; xG <= '1';
      when crossY => tR <= '1'; xY <= '1';
      when others => tG <= '1'; xR <= '1';
    end case;
  end process;

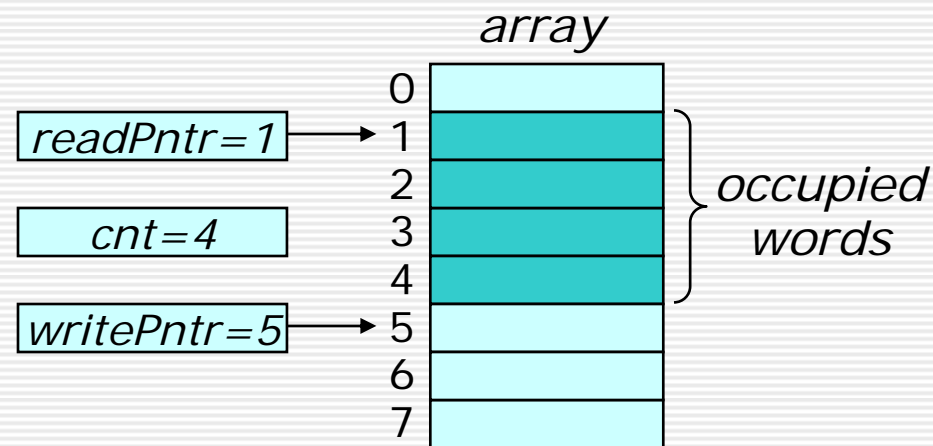
end trafficv_arch;
```

# Simulation of VHDL Version



# Data Queue

- A queue is a data structure that stores a set of values so they can be retrieved in the same order they were stored.
  - » operations are *enqueue* and *dequeue*
  - » separate *dataIn*, *dataOut* ports allow simultaneous *enqueue* & *dequeue*
  - » status signals: *empty* and *full*
  - » implement using an array of registers, pair of pointers and counter



# VHDL Design

```
entity queue is Port (  
    clk, reset: in std_logic;  
    enq, deq : in std_logic;  
    dataIn : in std_logic_vector(wordSize-1 downto 0);  
    dataOut : out std_logic_vector(wordSize-1 downto 0);  
    empty, full : out std_logic);  
end queue;  
architecture Behavioral of queue is  
    constant qSize: integer := 16;  
    constant lgQueueSize: integer := 4;  
    type qStoreTyp is array(0 to qSize-1)  
        of std_logic_vector(wordSize-1 downto 0);  
    signal qStore: qStoreTyp;  
    signal readPntr, writePntr: std_logic_vector(lgQueueSize-1 downto 0);  
    signal count: std_logic_vector(lgQueueSize downto 0);  
    function int(d: std_logic_vector) return integer is  
        -- Convert logic vector to integer. Handy for array indexing.  
    begin return conv_integer(unsigned(d)); end function int;  
begin  
    process (clk) begin  
        if clk'event and clk = '1' then
```

array type  
declaration for  
storing data

pointers and  
count register

type conversion  
function

```

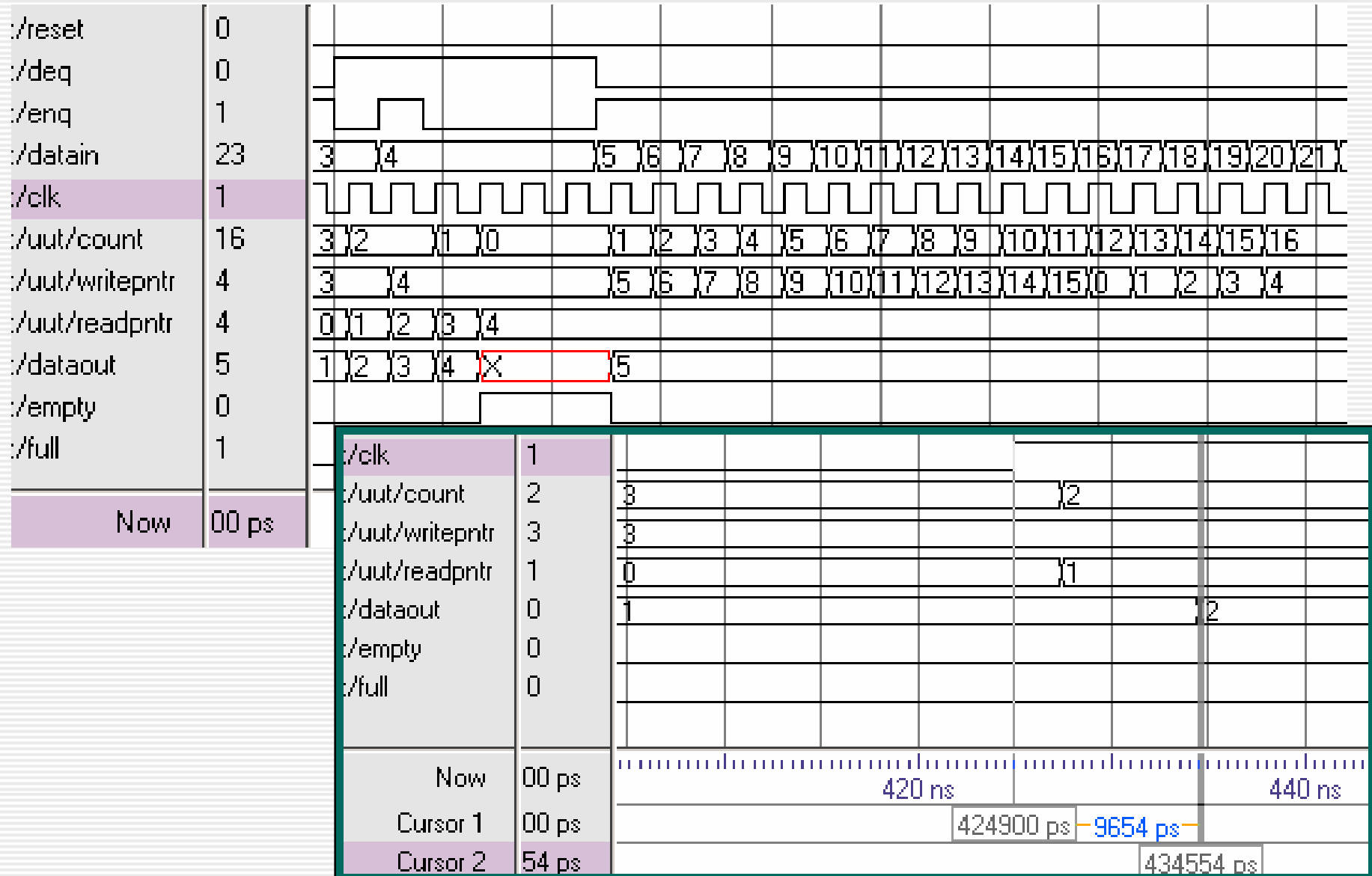
if reset = '1' then
    readPntr <= (readPntr'range => '0');
    writePntr <= (writePntr'range => '0');
    count <= (count'range => '0');
else
    if enq = '1' and deq = '1' then
        if count = 0 then
            qStore(int(writePntr)) <= dataIn;
            writePntr <= writePntr + '1'; count <= count + '1';
        else
            qStore(int(writePntr)) <= dataIn;
            readPntr <= readPntr + '1'; writePntr <= writePntr + '1';
        endif;
    elsif enq = '1' and count < qSize then
        qStore(int(writePntr)) <= dataIn;
        writePntr <= writePntr + '1'; count <= count + '1';
    elsif deq = '1' and count > 0 then
        readPntr <= readPntr + '1'; count <= count - '1';
    end if;
end if;
end if;
end process;
dataOut <= qStore(int(readPntr));
empty <= '1' when count = 0 else '0';
full <= '1' when count = qSize else '0';
end Behavioral;

```

simultaneous  
enq, deq

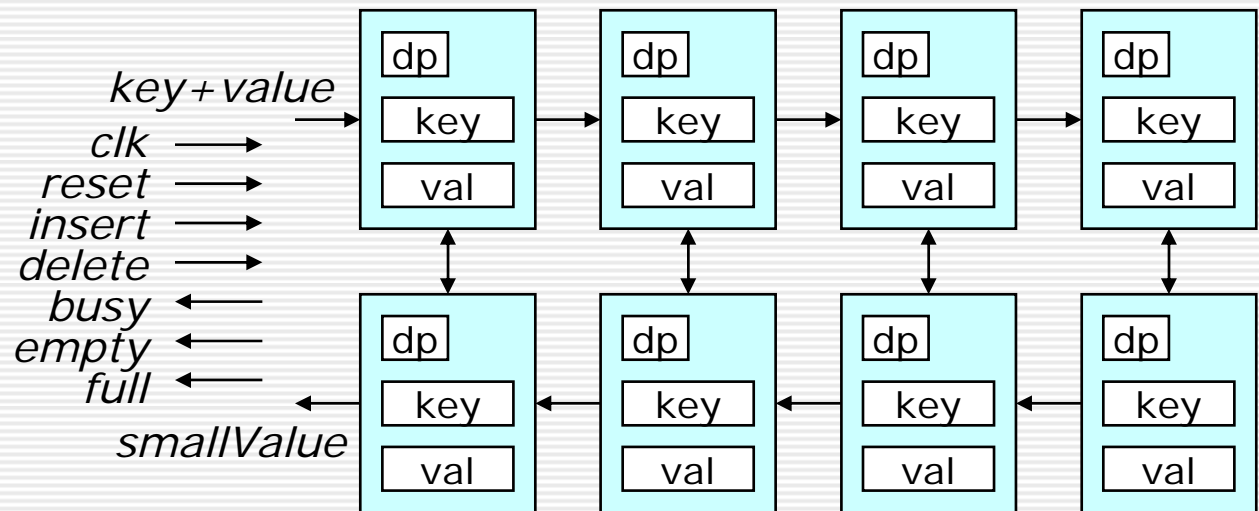
defining  
output signals

# Simulation Results



# Priority Queue

- Priority queue stores (key, value) pairs and always makes value with smallest key available.
  - » operations – reset, insert new pair, delete pair with smallest key
  - » inputs - clk, reset, insert, delete, key, value
  - » outputs – smallValue, empty, full, busy (when high, new inputs ignored)
- Implement as two rows of cells.
  - » each row has data present bit (dp) plus (key, value) registers
  - » keys in bottom row are sorted, keys in columns are sorted
  - » occupied cells to left, occupied top cell must have occupied cell below it
  - » to insert,
    - shift top to right
    - top-bottom swap
  - » to delete
    - shift bottom left
    - top-bottom swap



# VHDL for Priority Queue

```
package commonConstants is
    constant wordSize: integer := 4;
end package commonConstants;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.commonConstants.all;
entity priQueue is
    Port ( clk, reset : in std_logic;
          insert, delete : in std_logic;
          key, value : in std_logic_vector(wordSize-1 downto 0);
          smallValue : out std_logic_vector(wordSize-1 downto 0);
          busy, empty, full : out std_logic
    );
end priQueue;
architecture arch1 of priQueue is
    constant rowSize: integer := 4;
    type pqElement is record
        dp: std_logic;
        key: std_logic_vector(wordSize-1 downto 0);
        value: std_logic_vector(wordSize-1 downto 0);
    end record pqElement;
```

record used to  
group related  
data items

```
type rowTyp is array(0 to rowSize-1) of pqElement;
signal top, bot: rowTyp;
type state_type is (ready, inserting, deleting);
signal state: state_type;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then
        for i in 0 to rowSize-1 loop
          top(i).dp <= '0'; bot(i).dp <= '0';
        end loop;
        state <= ready;
      elsif state = ready and insert = '1' then
        if top(rowSize-1).dp /= '1' then
          for i in 1 to rowSize-1 loop
            top(i) <= top(i-1);
          end loop;
          top(0) <= ('1',key,value);
          state <= inserting;
        end if;
      end if;
    end process;
```

arrays of records  
implement two rows

make all slots  
empty initially

shift top  
row right

```

elsif state = ready and delete = '1' then
  if bot(0).dp /= '0' then
    for i in 0 to rowSize-2 loop
      bot(i) <= bot(i+1);
    end loop;
    bot(rowSize-1).dp <= '0'; state <= deleting;
  end if;
elsif state = inserting or state = deleting then
  for i in 0 to rowSize-1 loop
    if top(i).dp = '1' and
      (top(i).key < bot(i).key or bot(i).dp = '0') then
      bot(i) <= top(i); top(i) <= bot(i);
    end if;
  end loop;
  state <= ready;
end if;
end if;
end process;
smallValue <= bot(0).value when bot(0).dp = '1' else
  (smallValue'range => '0');
empty <= not bot(0).dp;
full <= top(rowSize-1).dp;
busy <= '1' when state /= ready else '0';
end arch1;

```

shift bottom  
row left

compare and  
swap columns

output signal  
definitions  
(all synchronous)

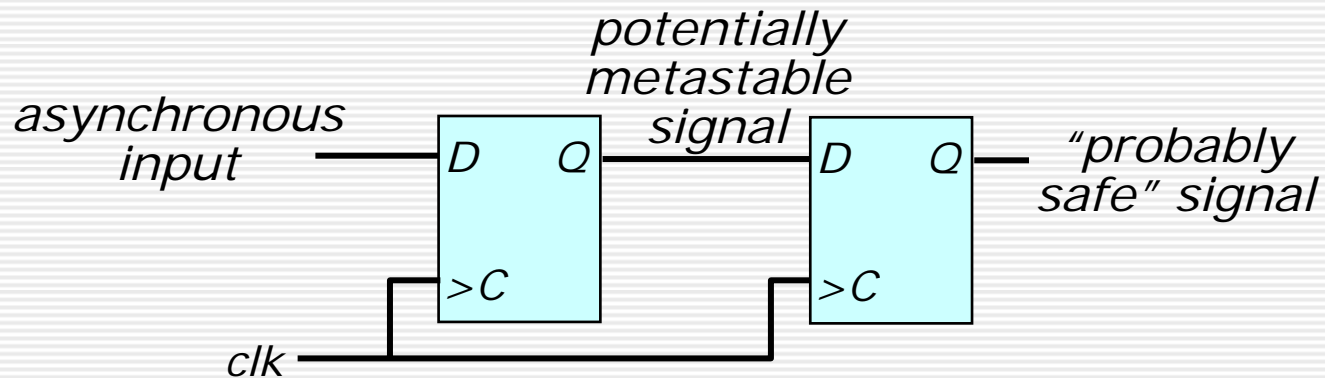


# Metastability

- Most synchronous systems have asynchronous inputs.
  - » keyboard input on a computer,
  - » sensor on a traffic light controller,
  - » card insertion on an ATM, etc.
- Asynchronous inputs change at unpredictable times.
  - » so, can change during clock transition, causing *metastability*
- Output of a metastable flip flop can oscillate or remain at intermediate value causing unpredictable behavior in other flip flops.
  - » metastability usually ends quickly, but *no definite time limit*
  - » so, circuit failures due to metastability *are unavoidable*
  - » however, systems can be designed to make failures rare

# Synchronizers

- *Synchronizers* are used to isolate metastable signals until they are “probably safe.”



- If the clock period is long enough, failure probability is small and expected time between failures is large.

MTBF = Mean Time Between Failures  $\approx (\alpha T/T_0) e^{T/\tau}$   
where  $T$  is the clock period,  $\alpha$  is the average time between asynchronous input changes,  $\tau$  and  $T_0$  are parameters of the flip flop being used.

- If  $T = 50$  ns,  $\alpha = 1$  ms,  $\tau = 1$  ns,  $T_0 = 1$  ns, MTBF  $\approx 8$  trillion years, if  $T = 10$  ns, MTBF becomes 220 seconds!