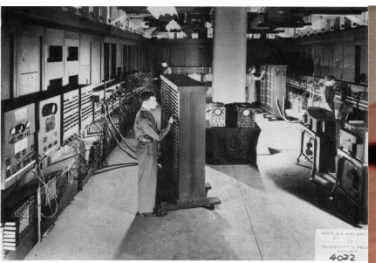
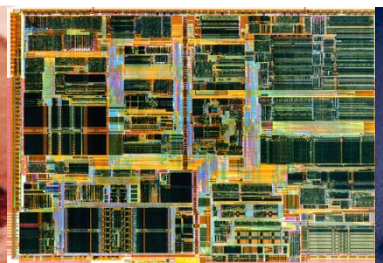
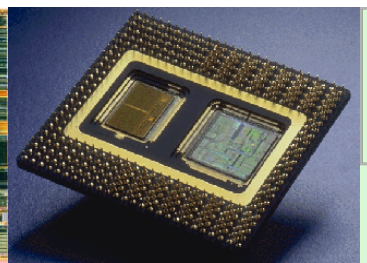
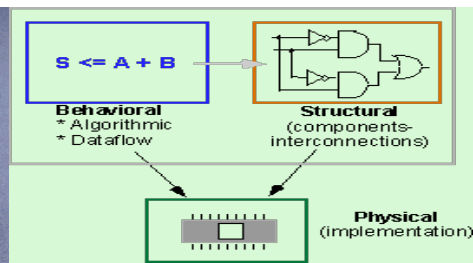
CPE 690: Introduction to VLSI Design

Lecture 4
VHDL – part III

Bryan Ackland
Department of Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ 07030

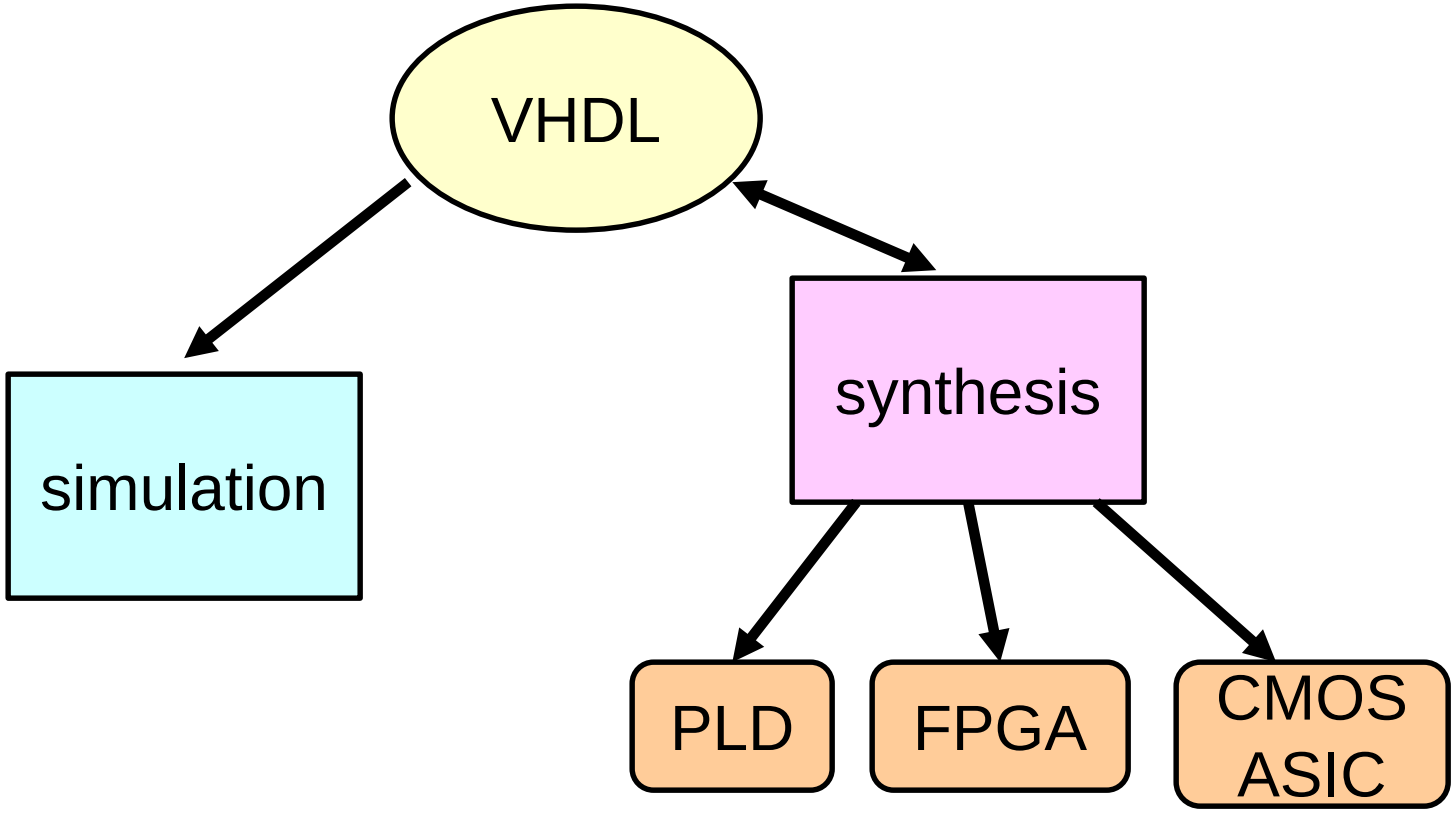


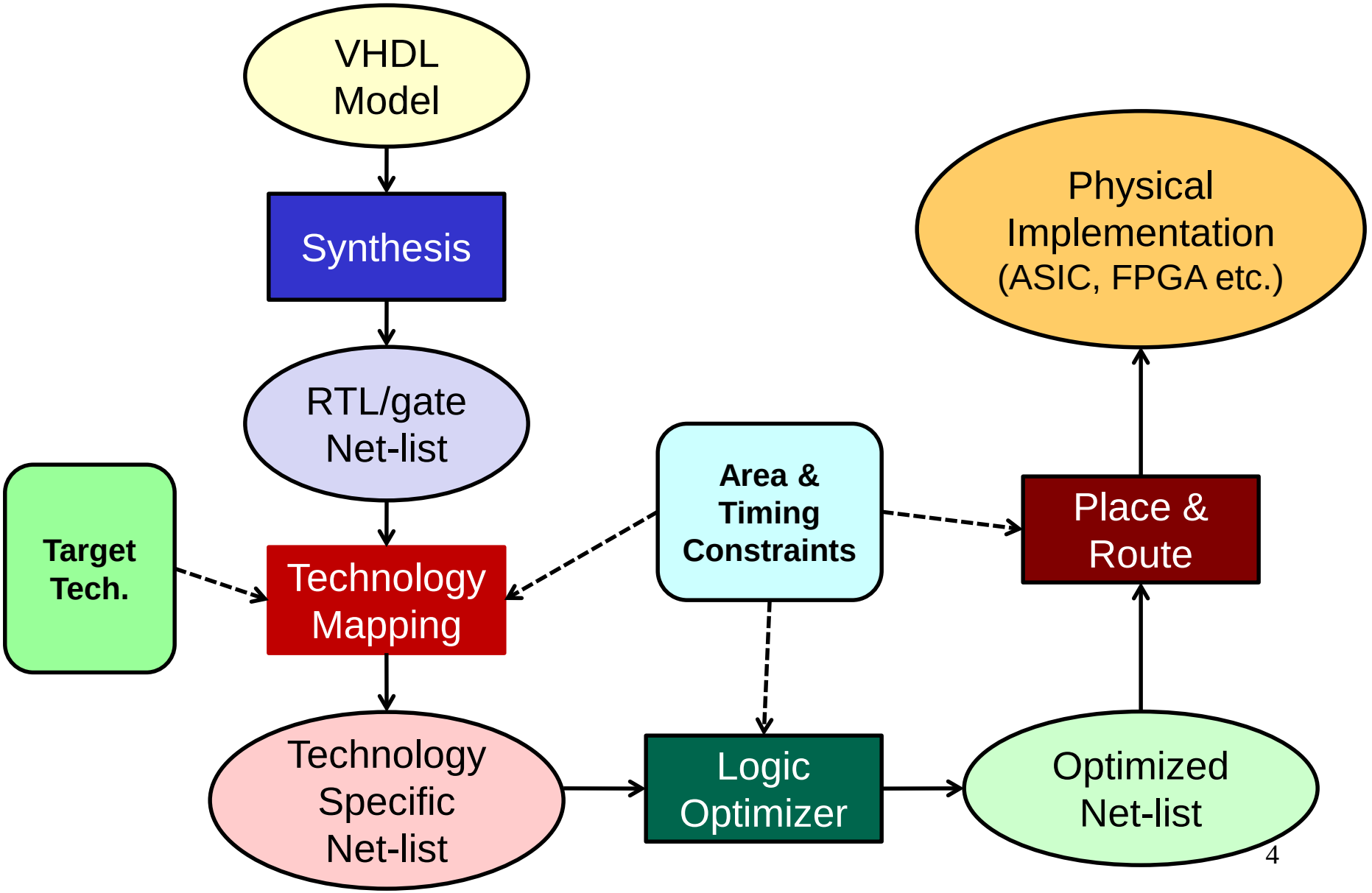
VHDL Synthesis

What is Synthesis?

- Process of creating a RTL or gate level net-list from a VHDL model
- Net-list can be used to create a physical (hardware) implementation



Synthesis Process



Limitations of Synthesis

- VHDL can model a hardware system at different levels of abstraction from gate level up algorithmic level
- Synthesis tools cannot, in general, map arbitrary high level behavioral models into hardware.
- Many VHDL constructs have no hardware equivalent:

```
wait for 10 ns;
signal a1: real;
y<= x'last_value;
assert a=b report "mismatch" severity error;
```

- Even if VHDL code is synthesizable, it may lead to very inefficient (in terms of area, speed) implementation
- In moving from VHDL high level behavioral description to synthesizable VHDL hardware description, designer needs to know:
  - What subset of VHDL is synthesizable
  - What hardware in inferred by various VHDL constructs

Inference

- Synthesis is the process of hardware inference followed by optimization
- The synthesis compiler infers hardware structures from your VHDL code
- Those hardware structures are subsequently optimized to meet your area and/or speed constraints.

- Part of being a good synthesis designer is being able to put yourself in the place of the compiler and understand what hardware constructs are likely to be inferred from your code.

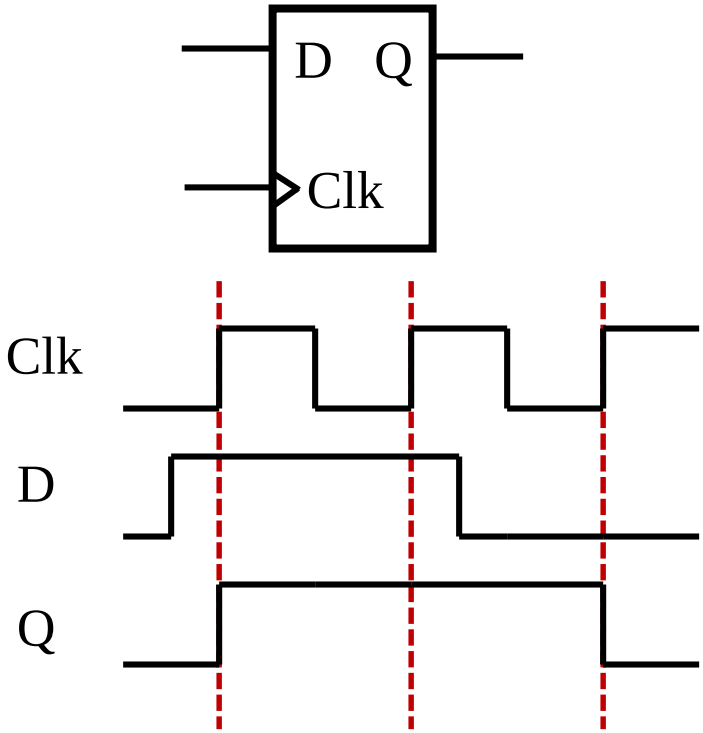
Inferring Signals

- In VHDL, signals are used to represent timed information flow between subsystems.
- When synthesizing from VHDL, the basic hardware implementations of signals are:
  - Wires
  - Latches (level sensitive)
  - Flip-flops (edge triggered)
- Signals generated by combinational circuits (where output depends only on the current value of the inputs) can be implemented as wires
- Signals generated by sequential circuits (where output depends on current and previous value of inputs) are implemented as latches or flip-flops

Flip-flop vs. Latch

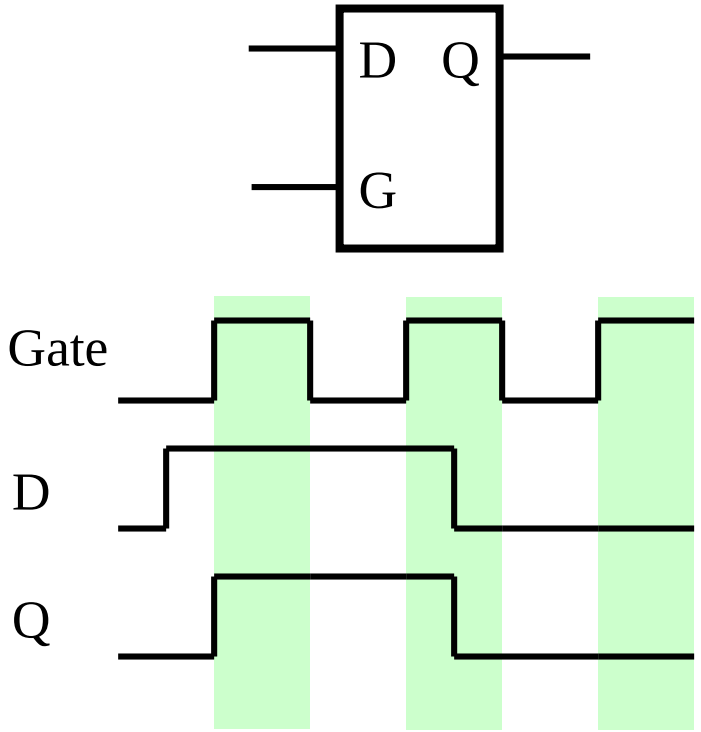
Flip-flop

stores data when clock rises



Latch

passes data when G is high
stores data when G is low



Inference from Declarations

- When a signal is declared, there needs to be sufficient information to determine the correct number of bits

```
signal result: std_logic_vector (12 downto 0);
signal count: integer;
signal index: integer range 0 to 18:= 0;

type state_type is (state0, state1, state2, state3);
signal next state: state_type;
```

- How many bits are required to represent *count* ?
  - Compilers are conservative: they will only perform transformations that are guaranteed not to produce incorrect answers
- Initializations are often ignored
  - Need to include run-time initialization in hardware (e.g. reset)
- How many bits are required to represent *next_state* ?

Inference from Simple CSA's

target_signal <= *expression*;

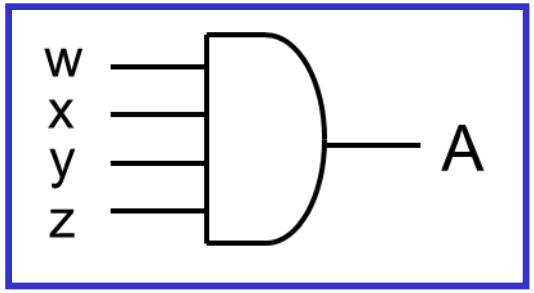
e.g. A <= (B **nand** C) **or** Y **after** 12 **ns**;

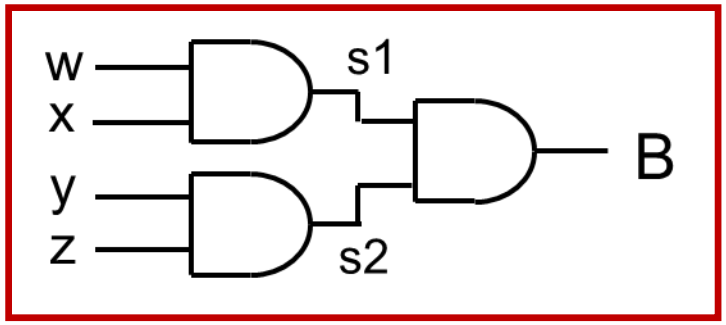
- A simple CSA assigns a value to a target signal that is a function of the present value of the signals in the RHS expression.
  - Whenever an event occurs on any signal in *expression*, *target_signal* is re-evaluated.
  - no memory of previous values
- The synthesis compiler will infer a combinational circuit
- Delay information (e.g. **after** 12 **ns**) will be ignored.
- You can control inferred structure by appropriately grouping logic operations into assignment statements

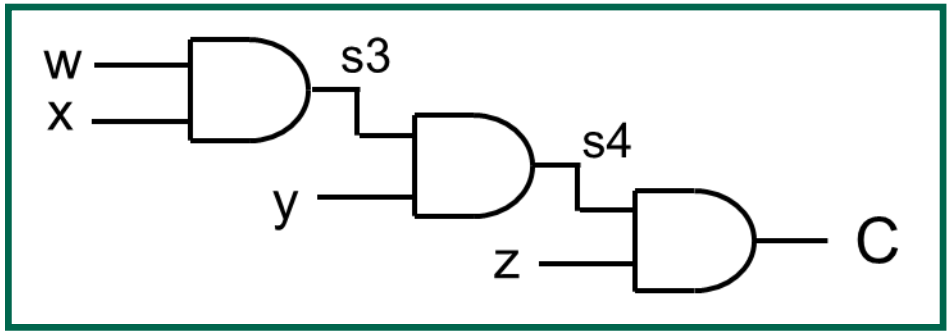
Logical Grouping

- Use of intermediate signals can be used to infer different logical (RTL) implementations:

```
entity abc is
port(w, x, y, z: in std_logic;
     A, B, C: out std_logic);
end entity abc;

architecture dataflow of abc is
signal s1, s2: std_logic;
begin
A <= w and x and y and z;
s1 <= w and x;
s2 <= y and z;
B <= s1 and s2;
s3 <= w and x;
s4 <= s3 and y;
C <= s4 and z;
end architecture;
```

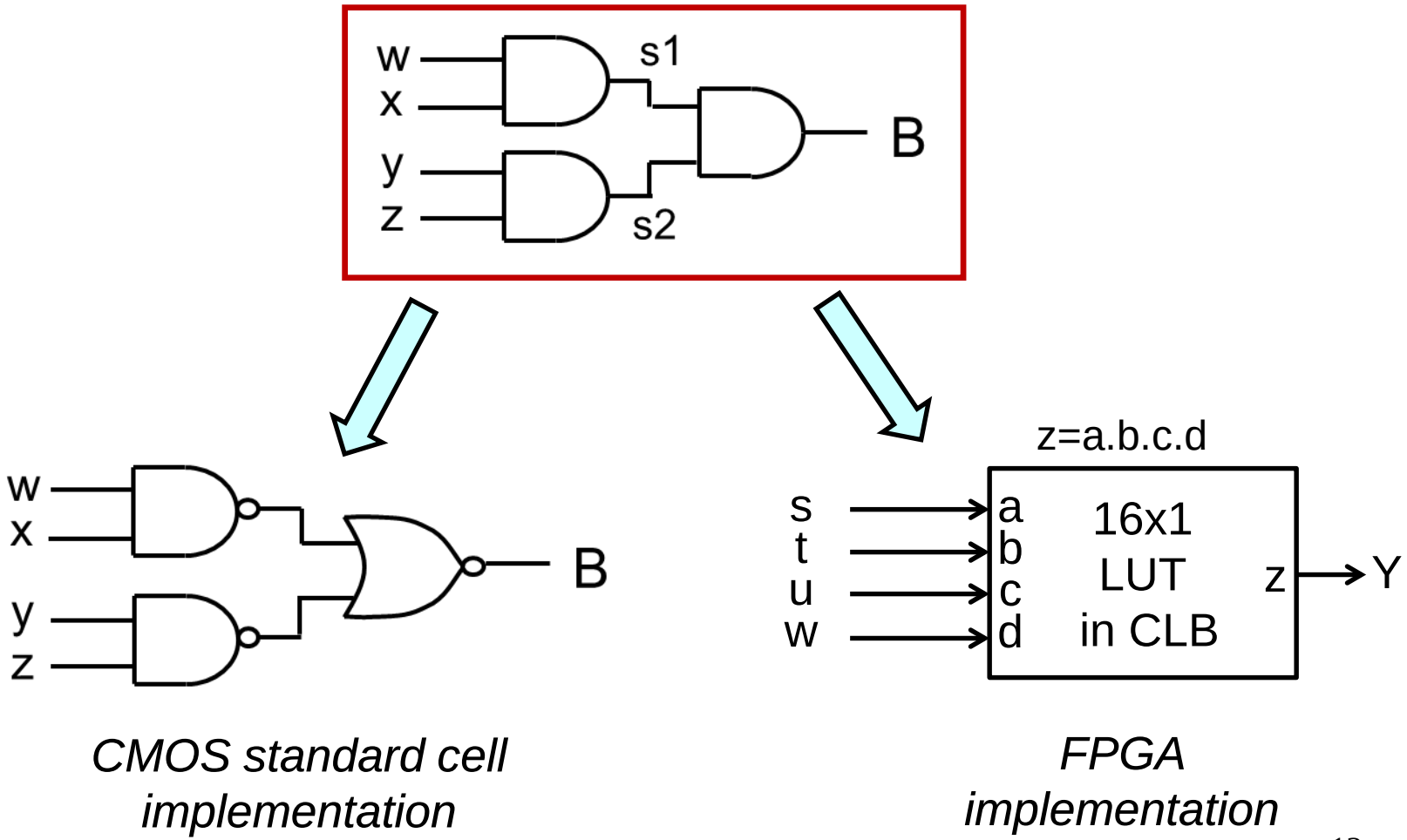






Logical Grouping Modified by Mapping

- RTL structure may be considerably modified by technology mapping and optimization:

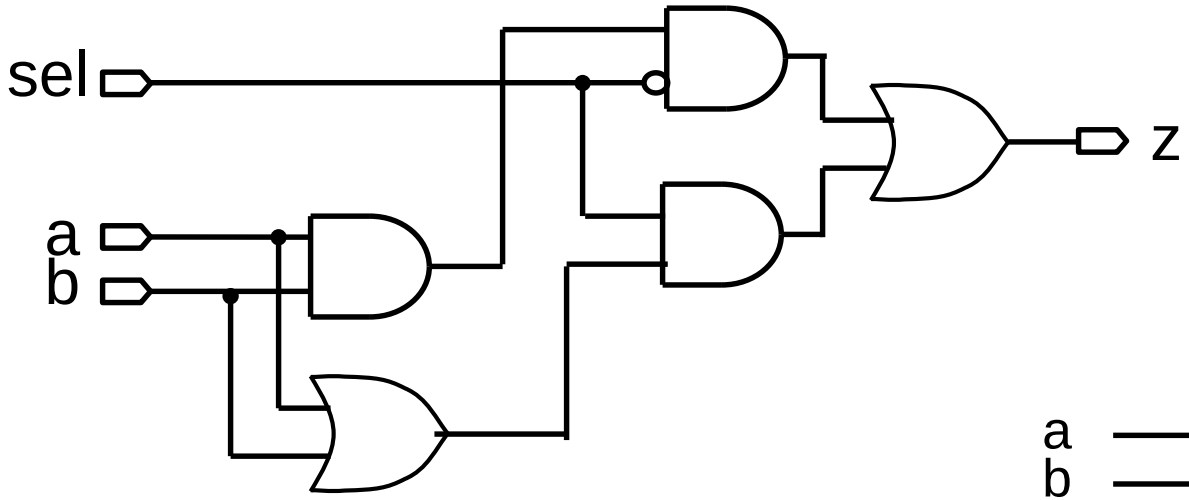


CMOS standard cell implementation

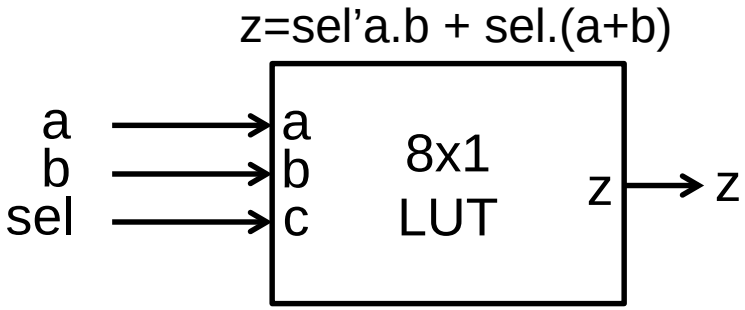
FPGA implementation

Inference from Conditional Signal Assignment

```
architecture cond_sa of mux is
begin
    z <=  a and b when sel='0' else
            a or b;
end cond_sa;
```



gate-level implementation

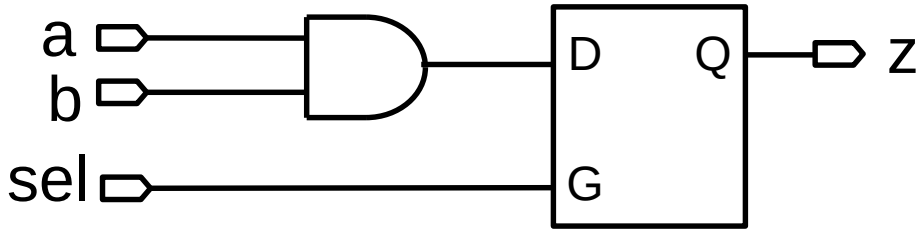


FPGA implementation

Incomplete Assignment Implies Latch

- What happens if there are combinations of inputs that do not result in an assignment?

```
architecture cond_sa of mux is
begin
    z <=  a and b when sel='1';
end cond_sa;
```

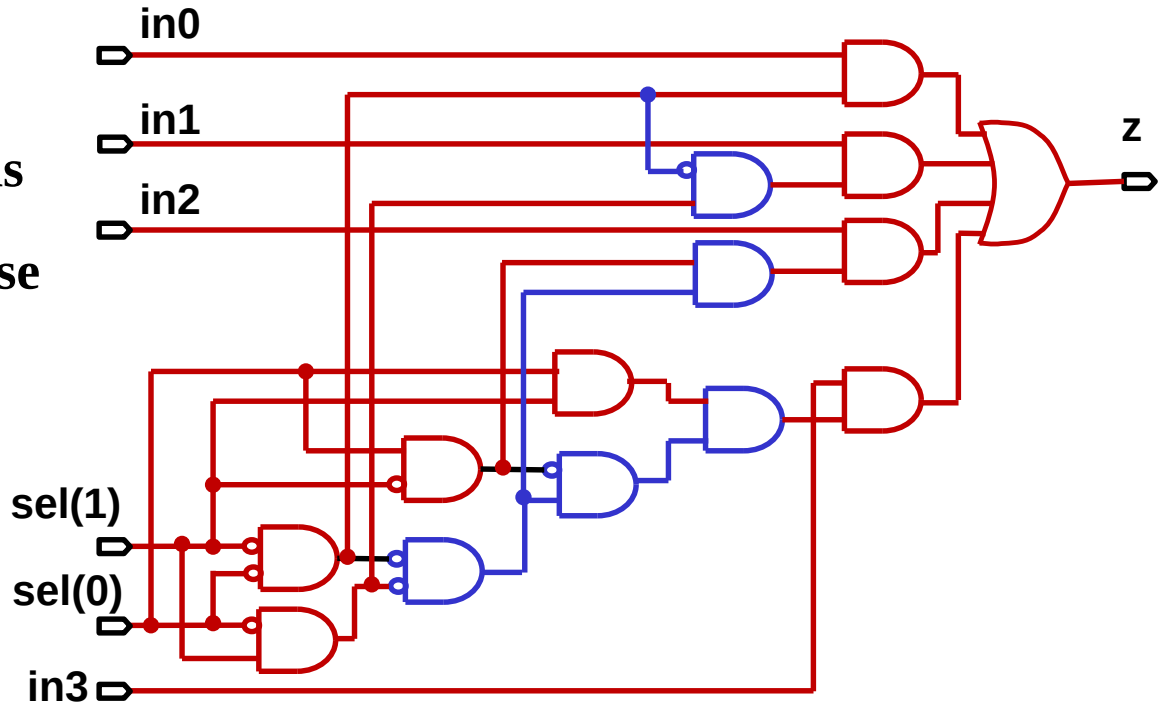


- A latch is inferred to cover the case *sel='0'*
  - This is now a sequential circuit – is that what we intended?
- We may not care what the result is when *sel='0'*
- Result is unnecessary, hazardous hardware
  - If we really wanted sequential operation, better to use flip-flop
- If combinational circuit is intention, make sure output is always assigned a value by using a final *else* clause

Conditional Maintains Priority Order

- This is great for priority encoder, but what about multiplexer where all conditions are mutually exclusive?

```
architecture behave of mux4 is
begin
    z <=  in0 when sel="00" else
          in1 when sel="01" else
          in2 when sel="10" else
          in3 when sel="11" else
          '0';
end behave;
```

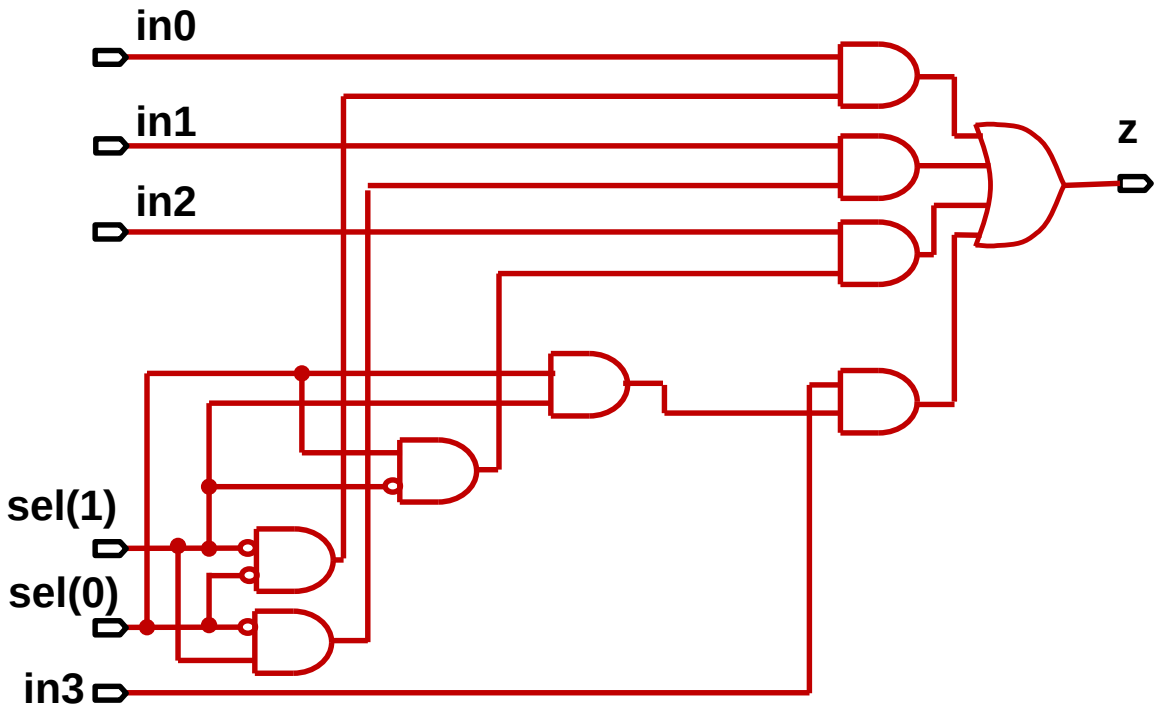


- Only red gates are needed to implement multiplexer
- Blue gates are inferred to maintain priority coding
- Not needed because the clauses are mutually exclusive

Selected Signal Assignment

- SSA is better construct for building a muliplexer
  - No longer implied priority order
  - Clauses are required to be mutually exclusive
  - No redundant gates

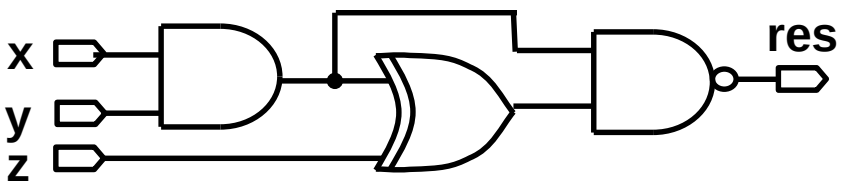
```
architecture SSA of mux4 is
begin
    with sel select
    z <=  in0 when “00” ,
          in1 when “01”,
          in2 when “10” ,
          in3 when “11”
          ‘X’ when others;
end SSA;
```



Inferring Logic from Processes and Variables

- Simple variable assignment statements generate combinational logic
- Sensitivity list is often ignored by synthesis compiler

```
entity syn_y is
port ( x,y,z: in std_logic;
     res: out std_logic);
end entity syn_v;

architecture behav of syn_v is
begin
pr1: process (x,y)
     variable v1,v2: std_logic;
     begin
          v1 := x and y;
          v2 := v1 xor z;
          res <= v1 nand v2;
     end process;
end behav;
```



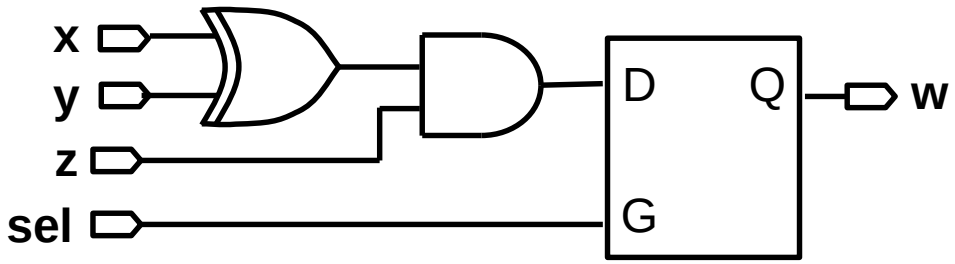
If-then-else Statements

- Like conditional assignment statements, if-then-else statements will generate latches *unless every "output signal" of the statement is assigned a value each time the process executes*

  (a) One branch of the if-then-else clause must always be taken AND

  (b) All output signals are assigned values in each branch

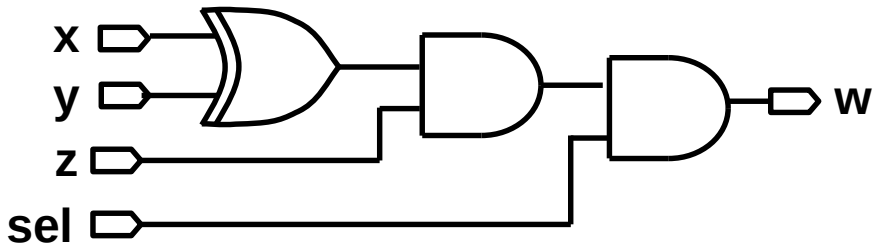
```
architecture behav of syn_if is
begin
pr1: process (x,y,z,sel)
     variable v1: std_logic;
     begin
          if sel='1' then
          v1 := x xor y;
          w <= v1 and  z;
          end if;
     end process;
end behav;
```



Avoiding latches by “initialization”

- Can ensure that a signal is always assigned in an if-then-else clause by including a final else clause that assigns a default value to all “output signals” of the statement
- Another alternative is to set a signal to a default value before the if-then-else statement:

```
architecture behav of syn_if is
begin
pr1: process (x,y,z,sel)
    variable v1: std_logic;
    begin
        w <= ‘0’;
        if sel=‘1’ then
        v1 := x xor y;
        w <= v1 and  z;
        end if;
    end process;
end behav;
```



Case Statement

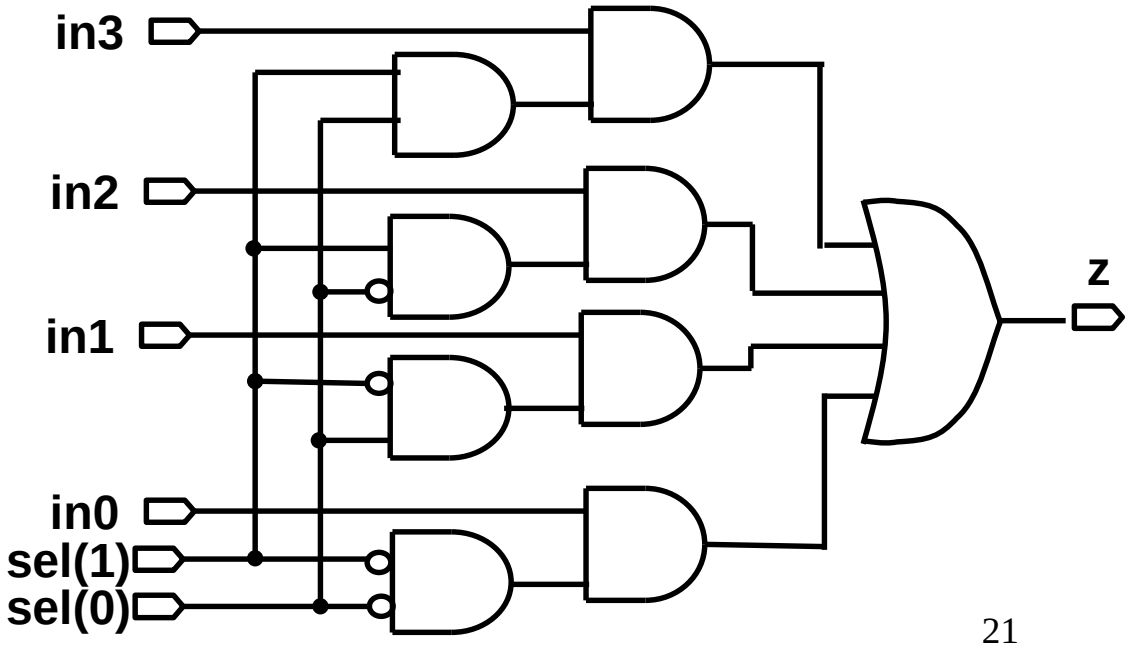
- Case statement is ideally suited for implementing multiplexers
  - all clauses must be mutually exclusive
  - no implied priority between clauses (unlike if-then-else)
  - no redundant logic
- Need to obey same rules to avoid latches

  (a) One branch of the case statement must always be true AND

  (b) If a signal is assigned a value in one branch of a case statement, it must be assigned a value in all branches

4 input multiplexer

```
architecture behav of syn_mux is
begin
pr1: process (in0,in1,in2,in3,sel)
     begin
          case sel is
          when “00” => z <= in0;
          when “01” => z <= in1;
          when “10” => z <= in2;
          when “11” => z <= in3;
          end case;
     end process;
end behav2;
```

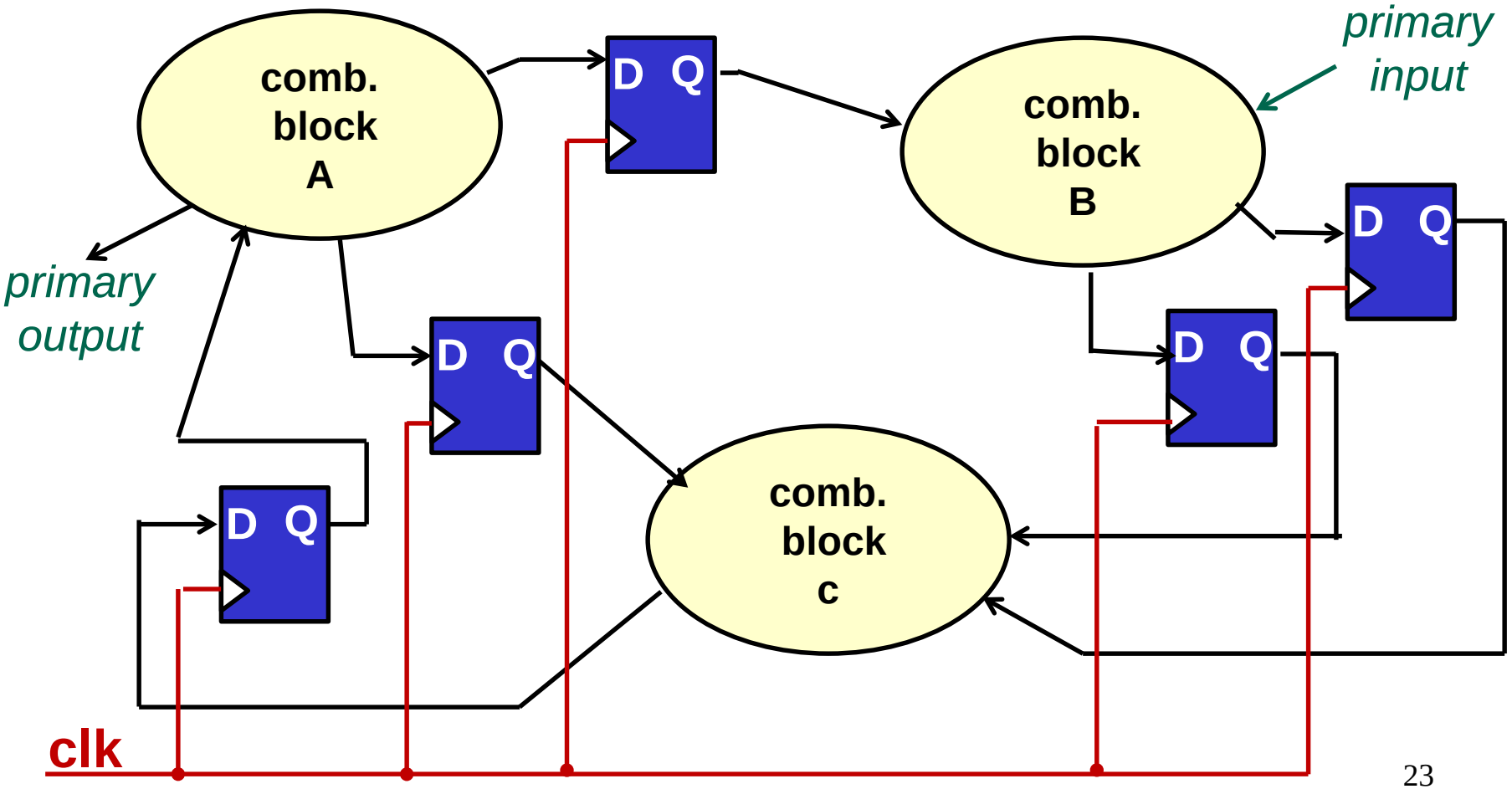


Sequential Circuits

- Any moderately complex digital system requires the use of sequential circuits, e.g. to
  - identify and modify current state of system
  - store and hold data
  - identify sequences of inputs
  - generate sequences of outputs
- If and case statements and conditional signal assignments can be used to infer latches
- Latches are not preferred means of generating sequential circuits.
  - A latch is transparent when LE is high – can lead to unintended asynchronous sequential behavior when a result is fed back to an input
  - Latches are prone to generate race conditions
  - Circuits with latches are difficult to verify timing
  - Circuits with latches are difficult to test

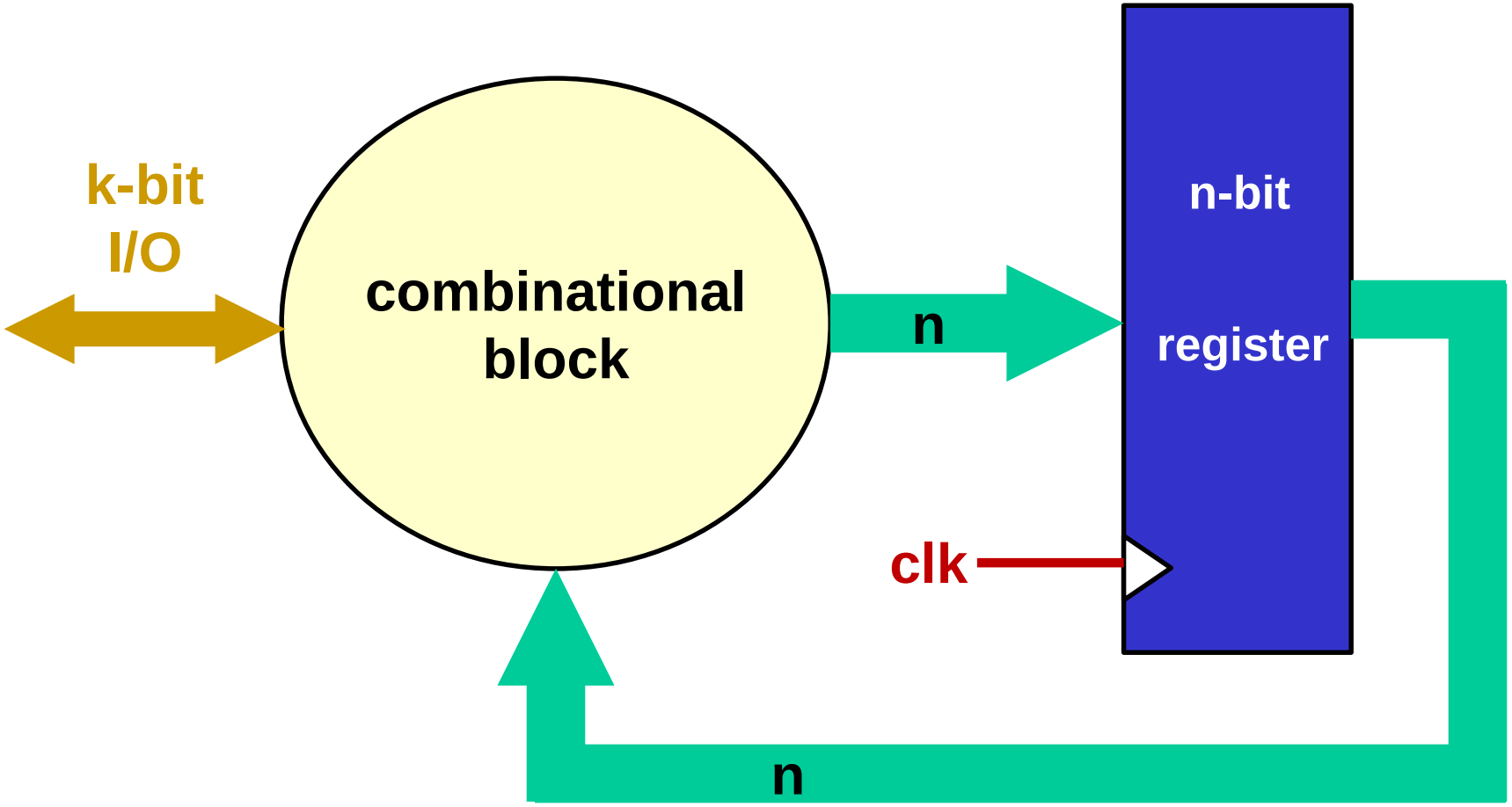
Synchronous (Single Clock) Digital Design

- Preferred design style is combinational circuit modules connected via positive (negative) edge-triggered flip-flops that all use a common clock.



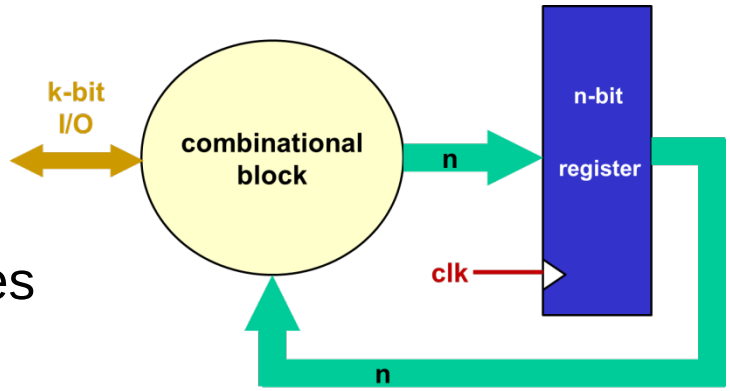
Finite State Machine

- Single clock synchronous system can be modeled as a single combinational block and a multi-bit register



Advantages of single clock synchronous

- Edge triggered D flip-flops are never transparent
  - no unintended asynchronous sequential circuits
- Timing can be analyzed by:
  - determining all combinational delays
  - just add them up
  - checking flip-flop setup and hold times



- No race conditions
  - only time signal values matter is on clock edge
- Easy to test
- Most real systems cannot be designed using a single clock
  - different timing constraints on various I/O interfaces
  - clock delays across chip
  - goal is to make the single clock modules as large as possible

D Flip-flop

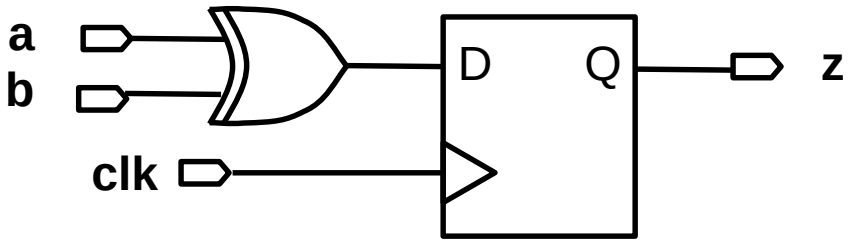
- Edge triggered D flip-flops are preferred sequential component
- Within a process, positive edge triggered operation is inferred using:

  if rising_edge (clk) **then**

  if clk**'event and** clk='1' **then**

- For example:

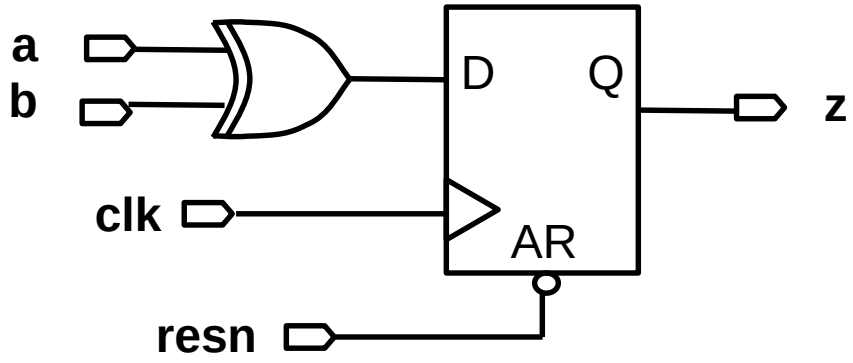
```
architecture RTL of FFS is
begin
p0: process (clk)
    begin
        if rising_edge (clk) then
            z <= a xor b;
        end if;
    end process;
end RTL;
```



Inferring D Flip-flop with Asynchronous Reset

- Asynchronous reset takes precedence over clock
- Flip-flop can also include asynchronous set

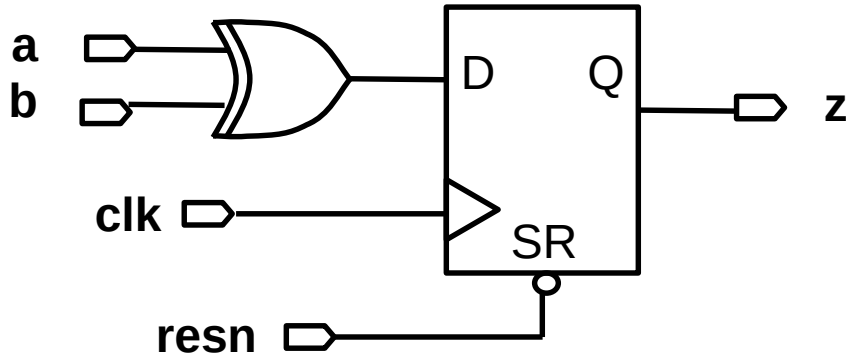
```
architecture RTL of ARFF is
begin
p0: process (resn, clk)
    begin
        if resn='0'  then
            z <= '0';
        elsif rising_edge (clk) then
            z <= a xor b;
        end if;
    end process;
end RTL;
```



Inferring D Flip-flop with Synchronous Reset

- Synchronous reset waits for clock
- Flip-flop can also include synchronous set

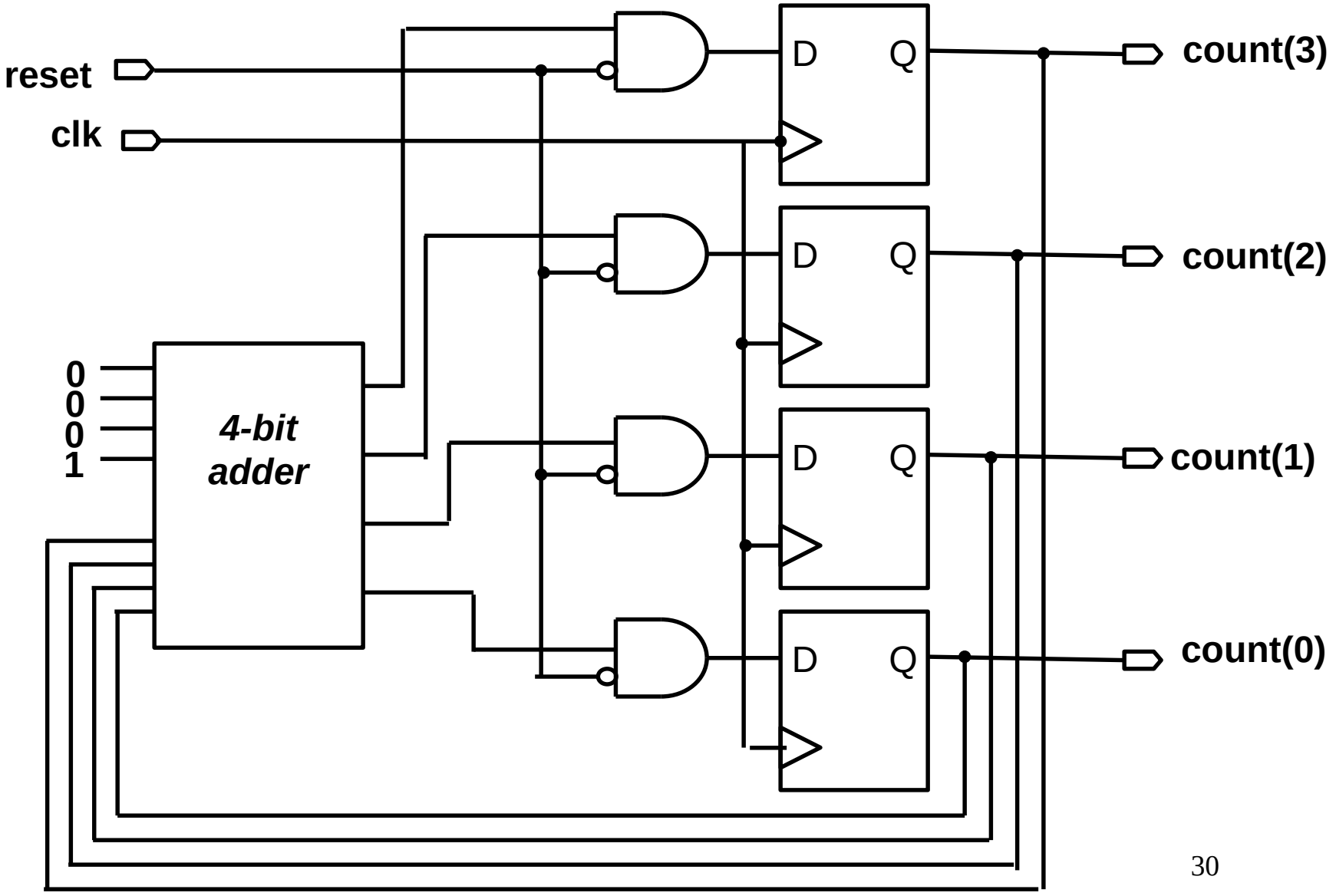
```
architecture RTL of ARFF is
begin
p0: process (clk)
    begin
        if rising_edge (clk) then
            if resn='0'  then
                z <= '0';
            else
                z <= a xor b;
            end if;
        end if;
    end process;
end RTL;
```



Example: 4-bit synchronous counter

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

entity count4 is
port( clk, reset: in std_logic;
    count: out std_logic_vector (3 downto 0));
end entity count4;

architecture RTL of count4 is
begin
p0: process (clk, reset)
    variable vcount: std_logic_vector (3 downto 0);
    begin
        if rising_edge (clk) then
            if reset='1' then
                vcount := "0000";
            else
                vcount := vcount+1;
            end if;
        end if;
        count <= vcount;
    end process;
end RTL;
```

Example: 4-bit synchronous counter



Wait Statement

- Only one wait statement permitted per process
- Must be the first statement in the process
- Must be of the **wait until** form
  - cannot use **wait for** or **wait on** constructs

- Both of these will trigger execution on rising clock edge:

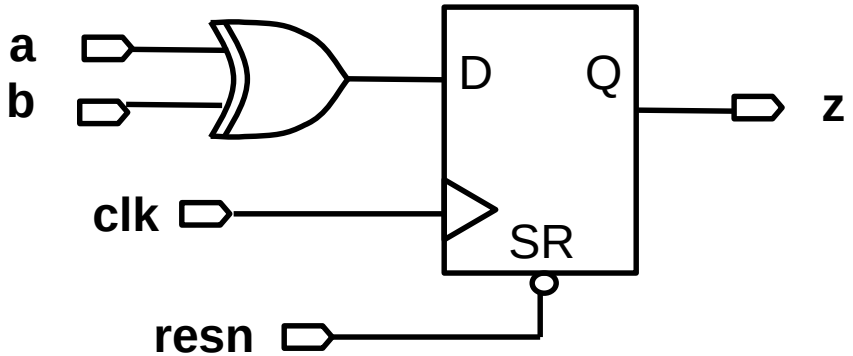
  wait until clk'event **and** clk='1';

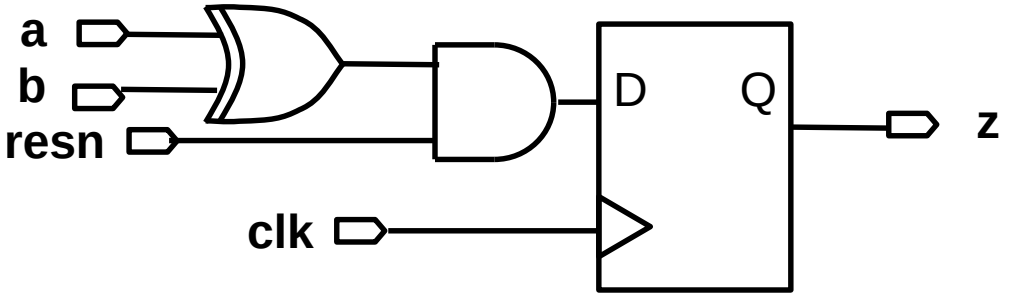
  wait until rising_edge(clk); -- only with std_logic type

- A D flip-flop will be inferred for all signals assigned in the process
  - all signals are synchronously assigned in process

Wait Statement - Example

```
architecture RTL of ARFF is
begin
p0: process
    begin
        wait until clock;
        if resn='0' then
            z <= '0';
        else
            z <= a xor b;
        end if;
    end process;
end RTL;
```





Loop Statements

- For loop is most commonly supported by synthesis compilers
- Iteration range should be known at compile time
- While statements are usually not supported because iteration range depends on a variable that may be data dependent.
- Compiler will unroll the loop, e.g.:

```
for k in 1 to 3 loop
    shift_reg(n) <= shift_reg(n-1);
end loop;
```

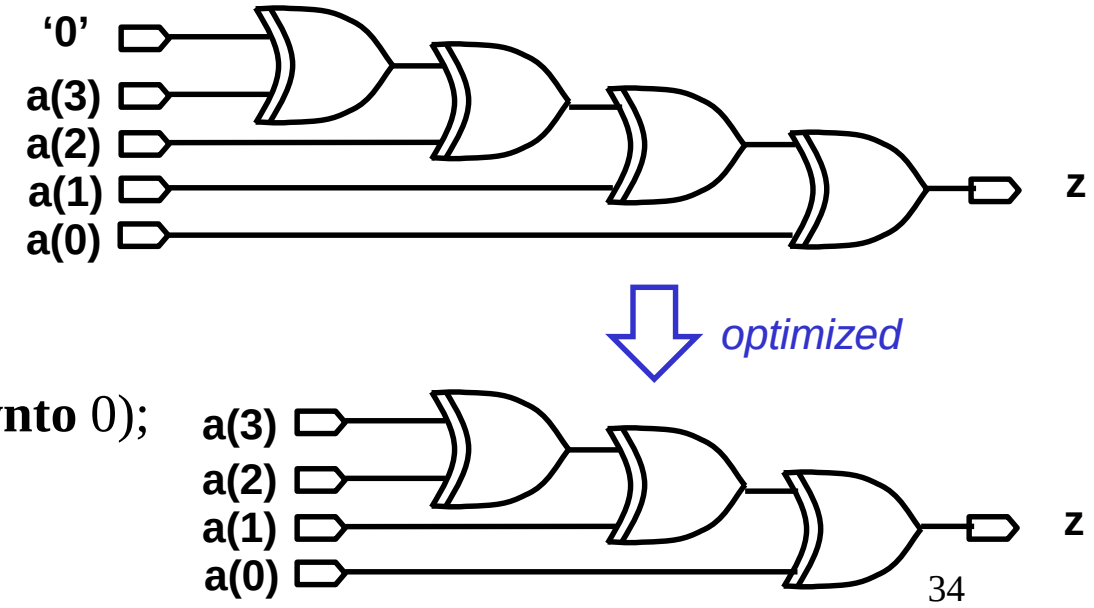
will be replaced by:

```
shift_reg(1) <= shift_reg(0);
shift_reg(2) <= shift_reg(1);
shift_reg(3) <= shift_reg(2);
```

Functions

- Since:
  - functions are executed in zero time and
  - functions do not remember local variable values between calls
- Functions will typically infer combinational logic
- Function call is essentially replaced by in-line code

```
function parity (data: std_logic_vector)
     return std_logic is
variable par: std_logic:= '0';
begin
    for i in data'range loop
        par := par xor data(i);
    end loop;
    return (par);
end function parity;

signal a: std_logic_vector (3 downto 0);
begin
    z <= parity (a);
```

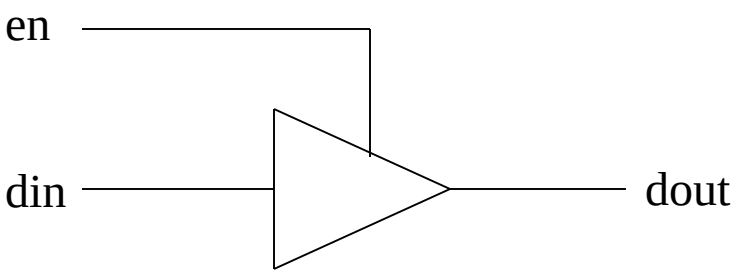


Procedures

- Like functions, procedures do not remember local variable values between calls
- Procedures, however, do not necessarily execute in zero time
- And procedures may assign new events to signals in parameter list
- Like function, think of procedure calls as substituting the procedure as inline code in the calling process or architecture
- Can include a wait statement, but then cannot be called from a process
  - Generally avoid wait statements in a procedure
  - Synchronous behavior can be inferred using if-then-else
- If procedure is called from a process with wait statement, flip-flops will be inferred for all signals assigned in procedure

Tri-state Gates

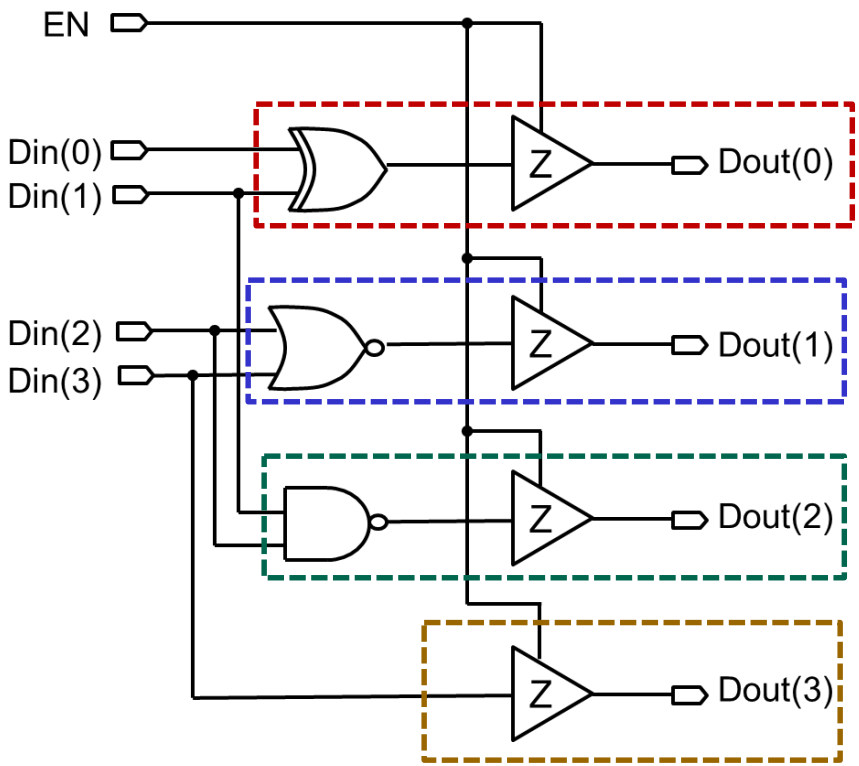
- A tri-state gate is inferred if the output value is conditionally set to high impedance Z



din	en	dout
0	1	0
1	1	1
X	0	Z

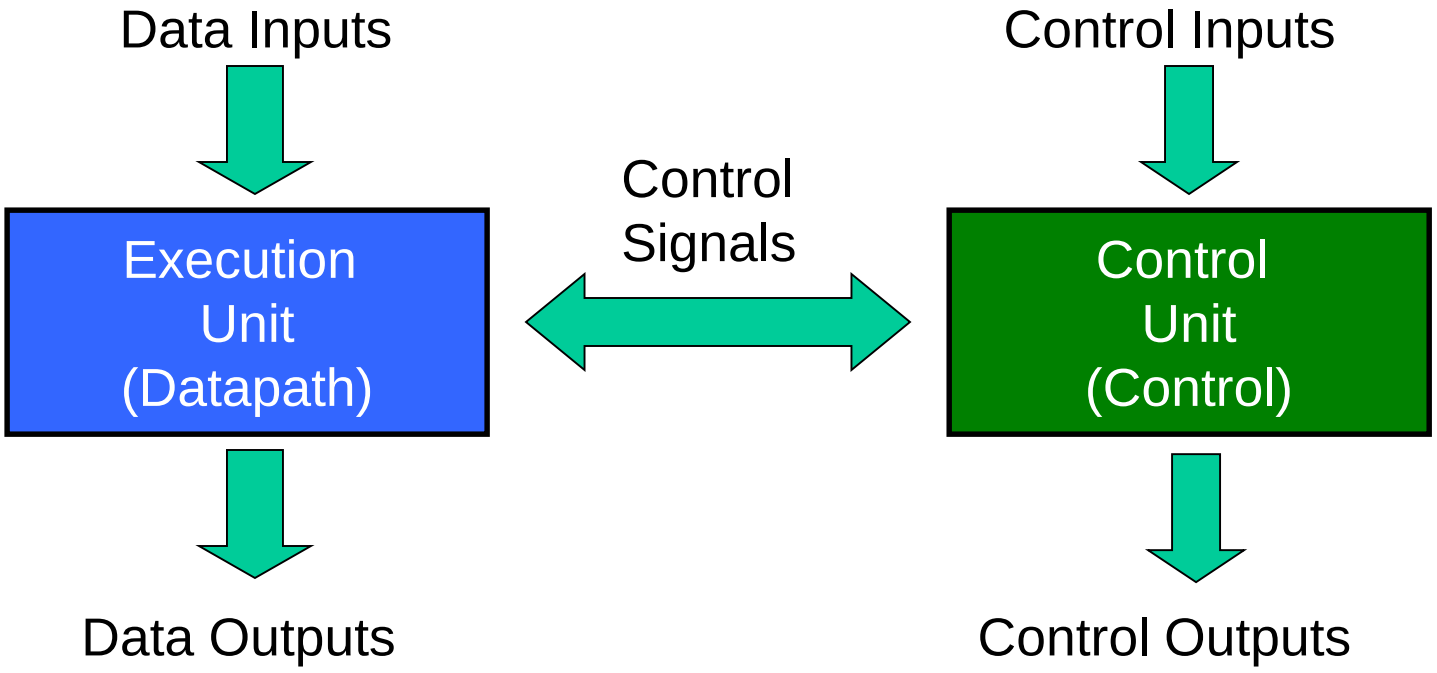
Examples of Tri-state Gates

```
architecture RTL of STATE3 is
begin
  seq0 : process (Din, EN)
  begin
    if (EN = '1') then
      Dout(0) <= Din(0) xor Din(1);
    else DOUT(0) <= 'Z';
    end if;
  end process;
  seq1 : process (EN, Din)
  begin
    case EN is
      when '1' => Dout(1) <= Din(2) nor Din(3);
      when others => Dout(1) <= 'Z';
    end case;
  end process;
  Dout(2) <= Din(1) nand Din(2) when EN = '1' else 'Z';
  with EN select
      Dout(3) <= Din (3) when '1',
      'Z'    when others;
end RTL;
```



Finite State Machines

Structure of Typical Digital System



Provides necessary resources & interconnect to perform specified task e.g.:

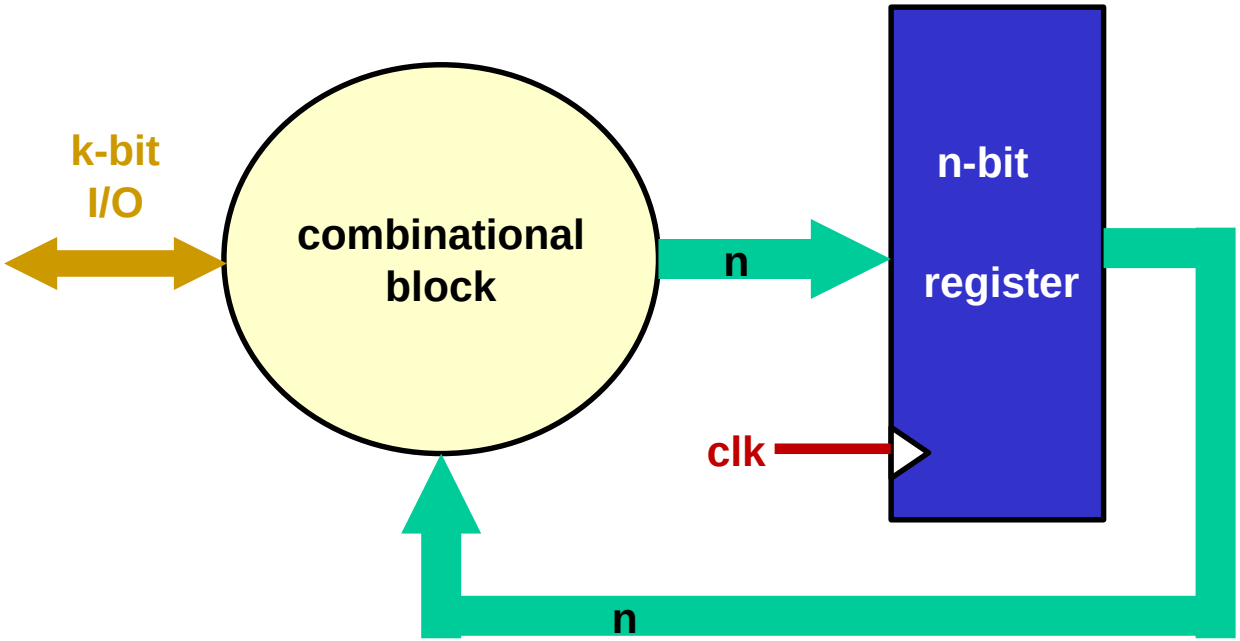
Adders, Multipliers, Shifters, Registers, Memories, etc.

Controls data movement and operation of execution unit. Usually follows some “program” or “sequence”.

Often implemented as one or more Finite State Machine(s)

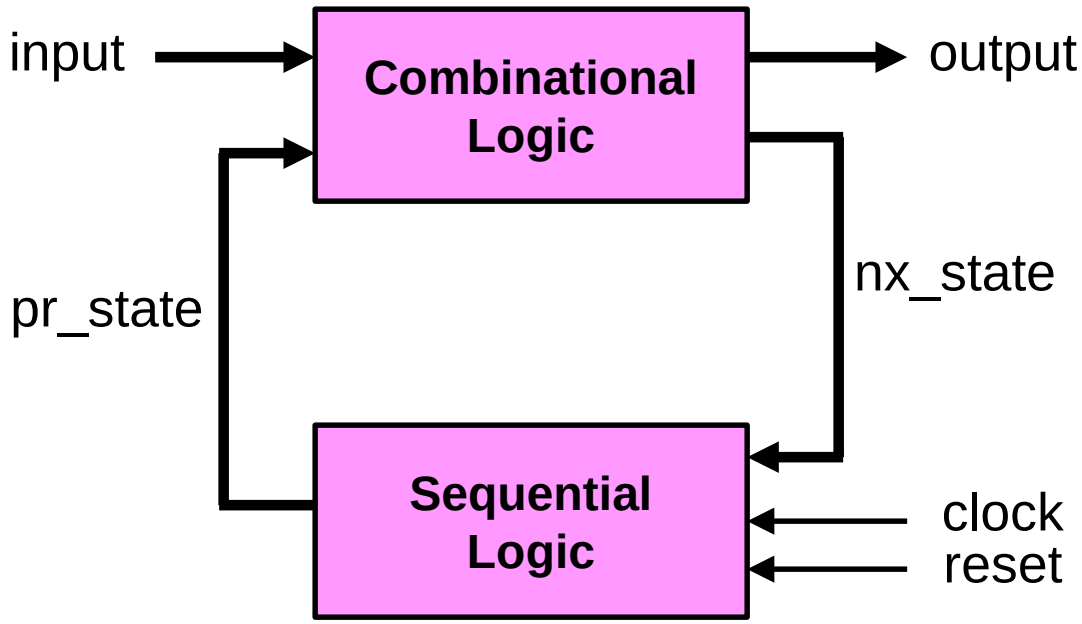
Finite State Machine

- Single clock synchronous system can be modeled as a single combinational block and a multi-bit register



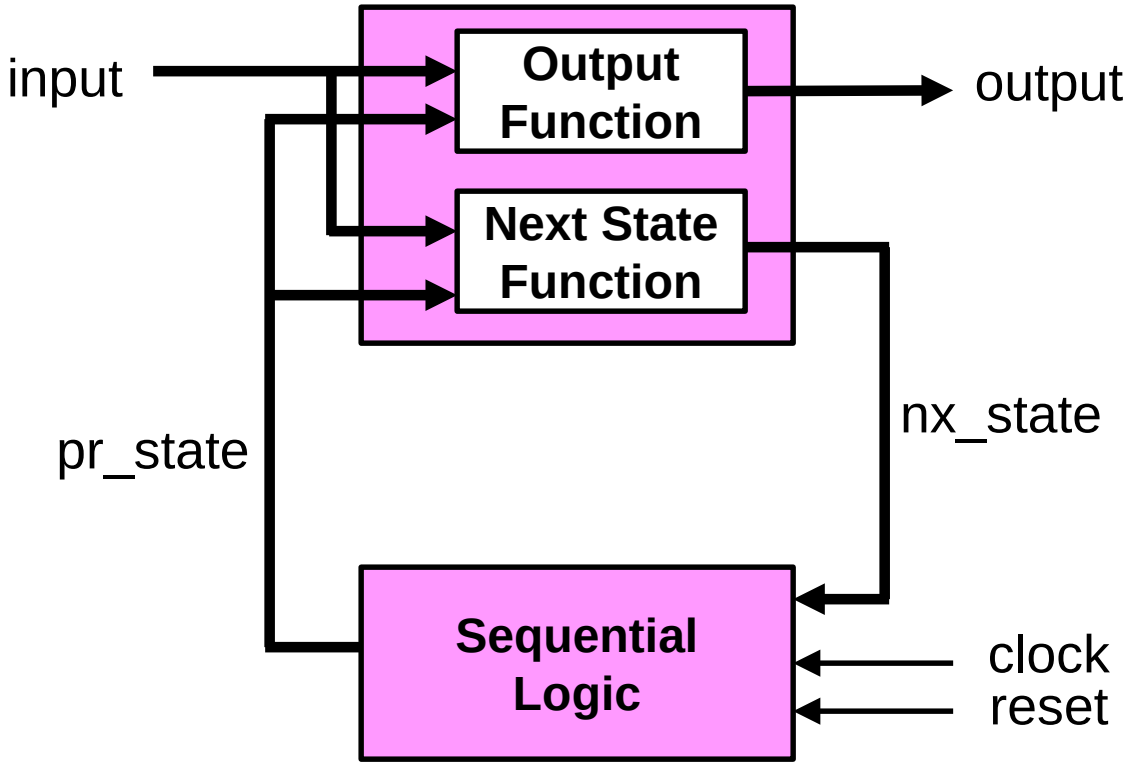
- Values stored in registers are state of the system
- Number of states is finite ($\leq 2^n$)
- State may change as a function of inputs
- Outputs are function of state and inputs

General Architecture of FSM



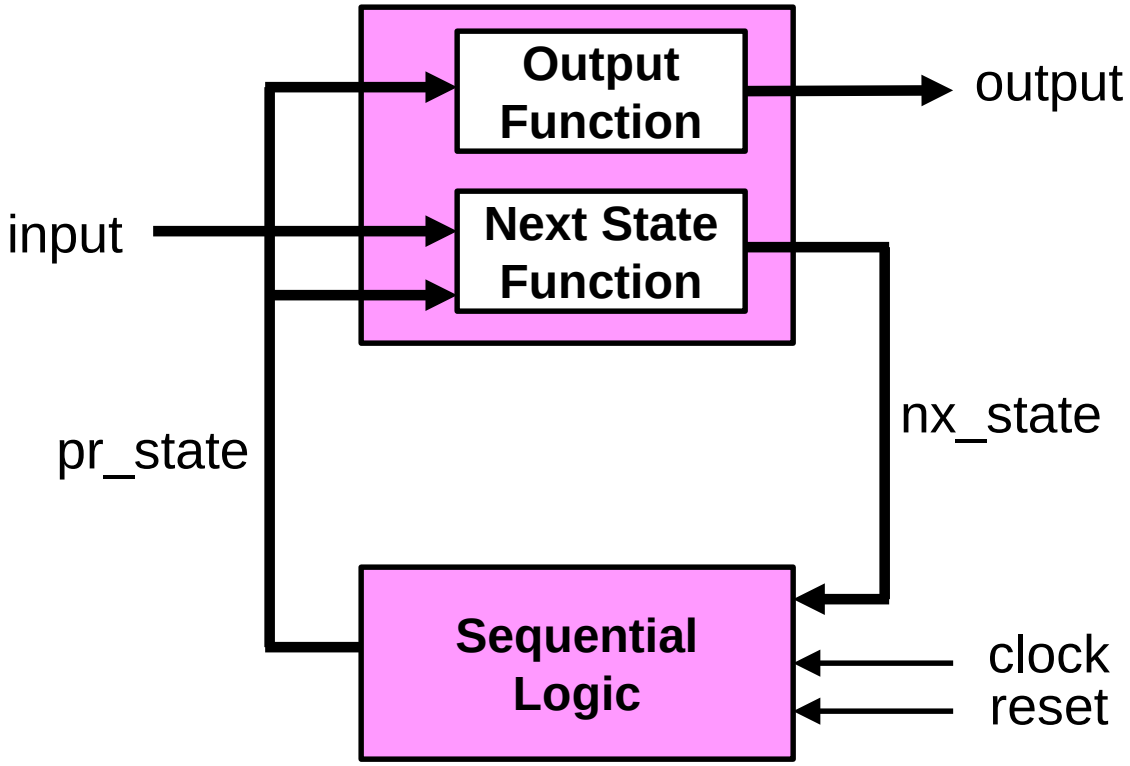
- Next state (nx_state) is a function of present state (pr_state) and inputs
- Output is a function of pr_state. May also be a function of inputs
- Reset allows system to be set to a known state

Mealy Machine



- Output is a function of pr_state *and* inputs
- Fast response (input -> output) – no FF's in way
- Leads to a fewer number of states
- Propagates asynchronous behavior (e.g. glitches) from input to output

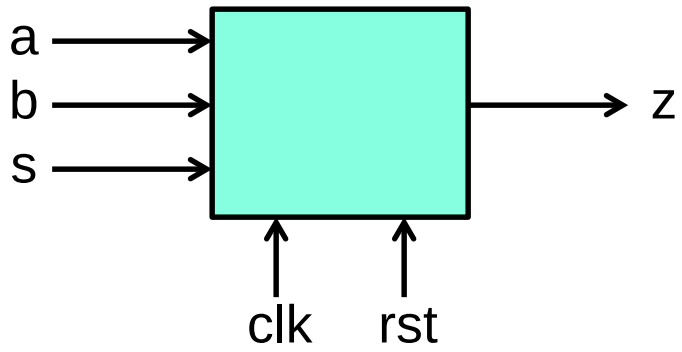
Moore Machine

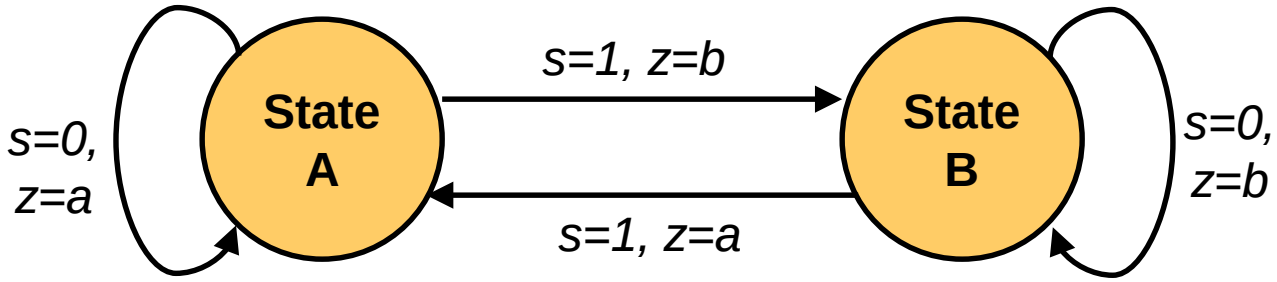


- Output is a function of pr_state *only*
- Slower response: (input -> output) requires clock transition
- Usually requires more states
- Operation is fully synchronous

Example: Toggle Multiplexer

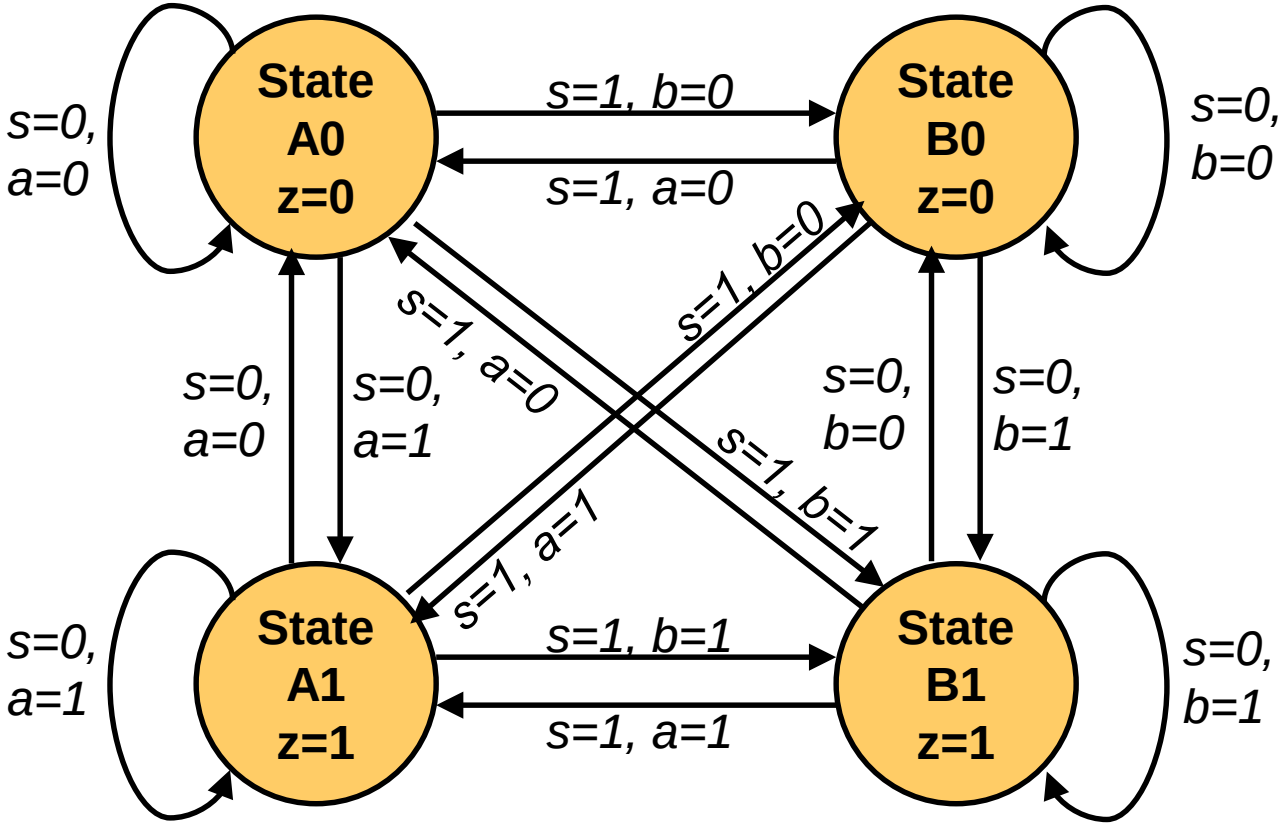
- Design a synchronous multiplexer that selects between two 1-bit inputs a and b. The multiplexer switches from one input to the other whenever a third input s is set to ‘1’





- This is a Mealy machine

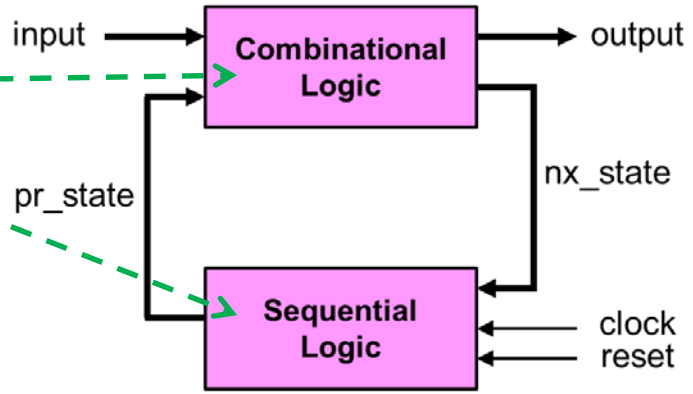
Example: As a Moore Machine



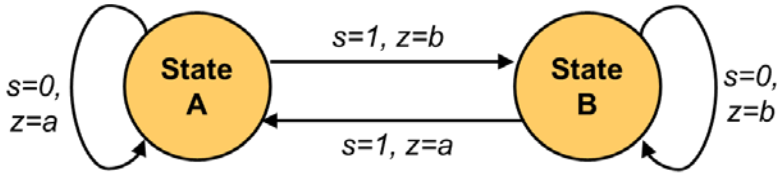
- This is a Moore machine

Toggle Multiplexer as Mealy Machine

- General approach is to model FSM as two communicating concurrent processes
  - Combinational process
  - Edge triggered clock process

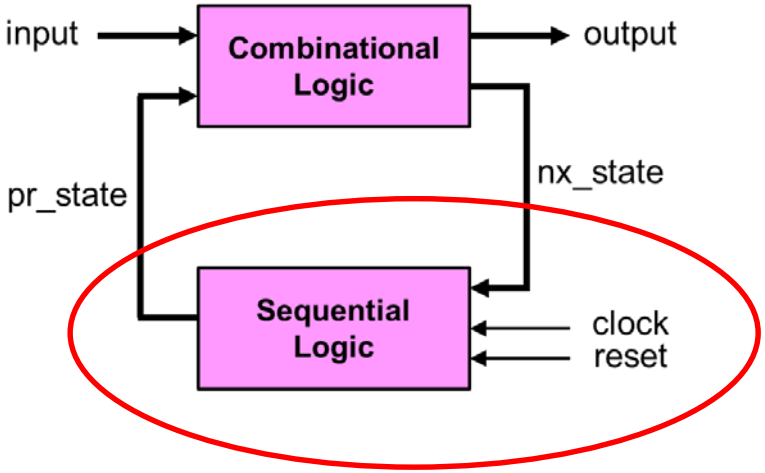


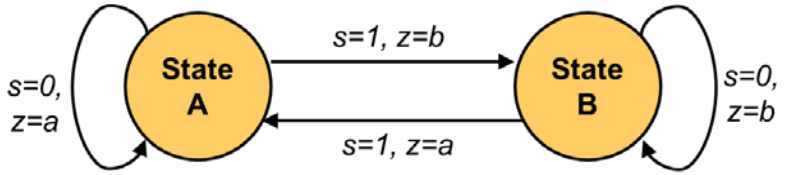

```
library  ieee;
use ieee.std_logic_1164.all;

entity tmpx is
port(a,b,s,clk,rst: in std_logic;
     z: out std_logic);
end entity tmpx;

architecture mealy of tmpx is
type state is (stateA, stateB);
signal pr_state, nx_state : state;
begin
```



Clock Process

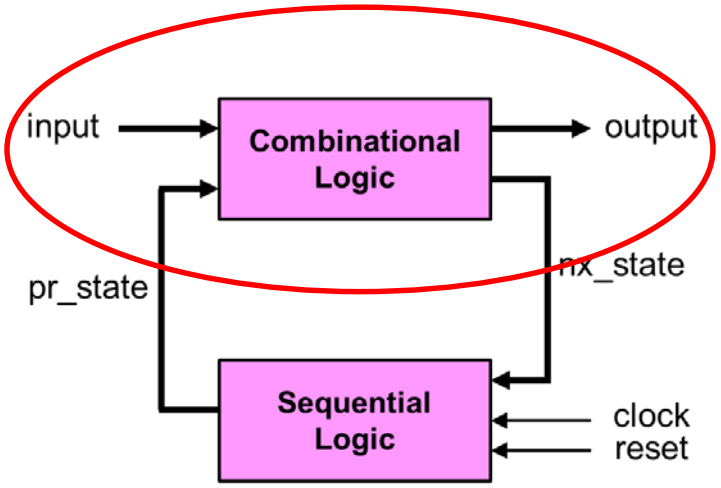
```
p_clk: process (rst, clk)
    begin
        if (rst = '1') then
            pr_state <= stateA;
        elsif (clk'event and clk = '1') then
            pr_state <= nx_state;
        end if;
    end process;
```

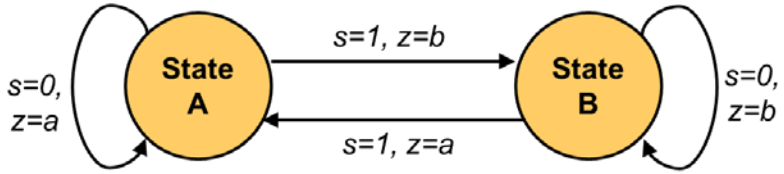




Combinational Process uses Case Statement

```
p_comb: process (a, b, s, pr_state)
    begin
        case pr_state is
        when stateA =>
            z <= a;
            if (s = '1') then nx_state <= stateB;
                else nx_state <= stateA;
            end if;
        when stateB =>
            z <= b;
            if (s = '1') then nx_state <= stateA;
                else nx_state <= stateB;
            end if;
        end case;
    end process;
end architecture mealy;
```

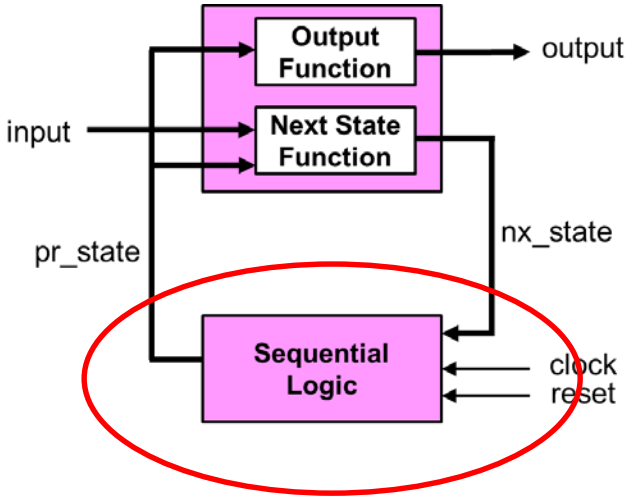


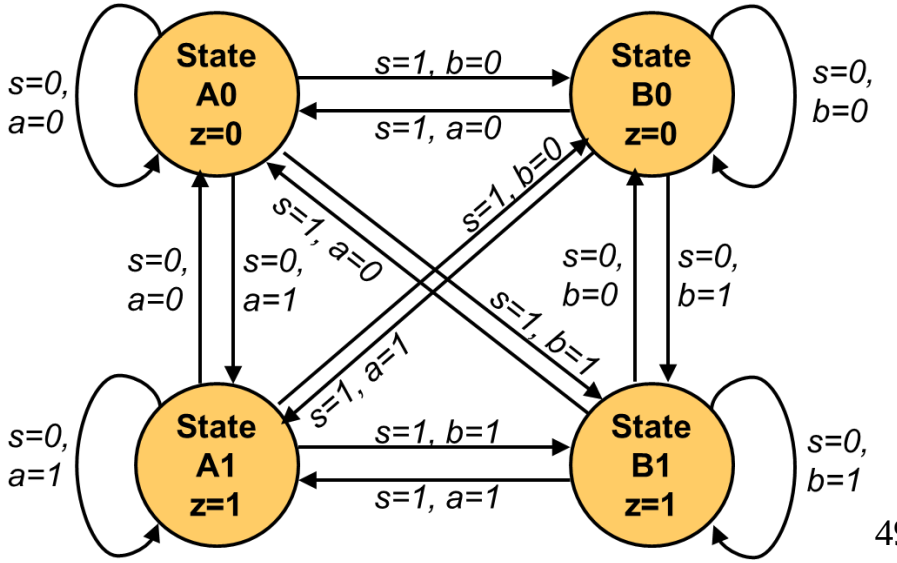


- Make sure all signals are assigned in all branches of case statement to avoid inferring latches

Moore Machine Implementation

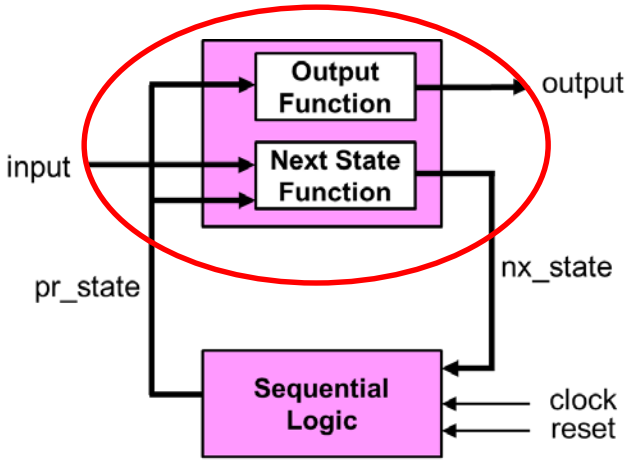
```
architecture moore of tmpx is
type state is (A0, A1, B0, B1);
signal pr_state, nx_state: state;

begin
p0: process (rst, clk)
        if (rst = '1') then
            pr_state := A0;
        elsif (clk'event and clk = '1') then
            pr_state <= nx_state;
        end if;
    end process;
```

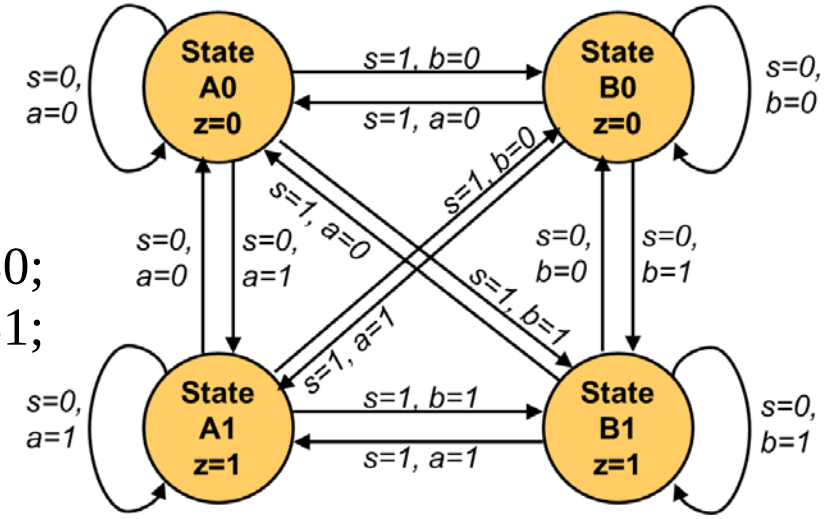




Moore Machine Implementation (2)

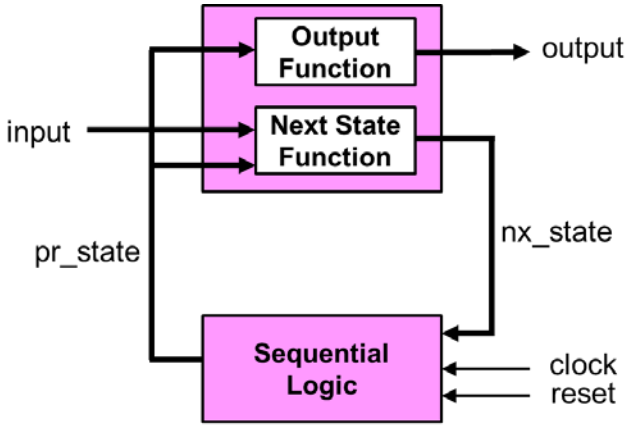
```
p_comb: process (a, b, s, pr_state)
    begin
        case pr_state is
        when A0 =>
            z<='0';
            if s='0' and a='1' then nx_state<=A1;
            elsif s='1' and b='0' then nx_state<= B0;
            elsif s='1' and b='1' then nx_state<= B1;
            else nx_state<=A0;
            end if;
        when A1 =>
            z<='1';
            if s='0' and a='0' then nx_state<=A0;
            elsif s='1' and b='0' then nx_state<= B0;
            elsif s='1' and b='1' then nx_state<= B1;
            else nx_state<=A1;
            end if;
```

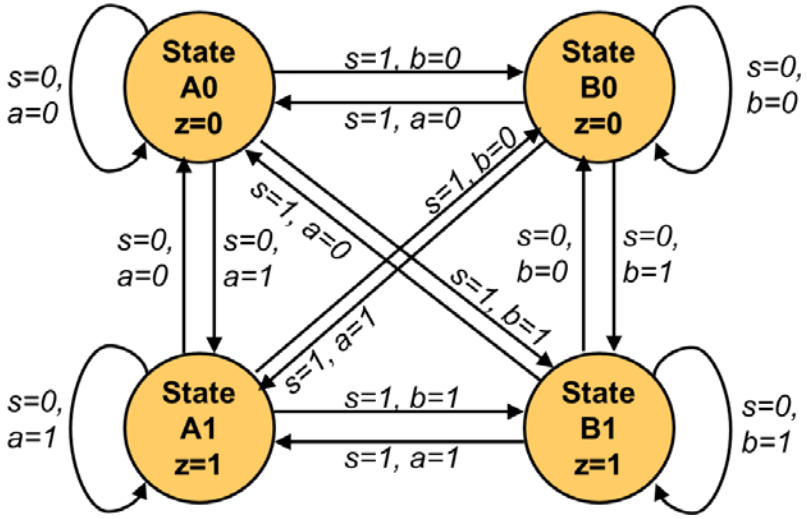




Moore Machine Implementation (3)

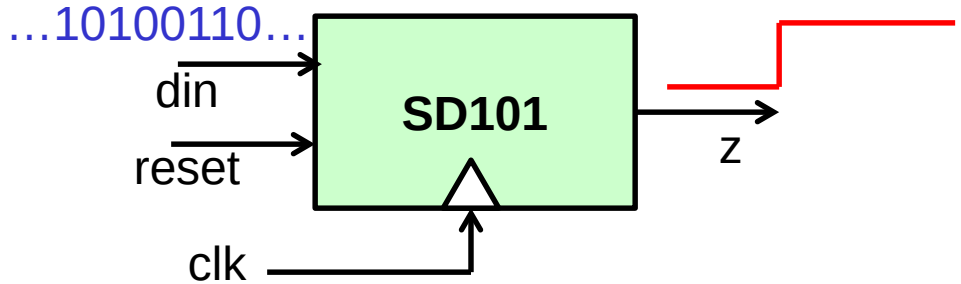
```
    when B0 =>
        z<='0';
        if s='0' and b='1' then nx_state<=B1;
        elsif s='1' and a='0' then nx_state<= A0;
        elsif s='1' and a='1' then nx_state<= A1;
        else nx_state<=B0;
        end if;
    when B1 =>
        z<='1';
        if s='0' and b='0' then nx_state<=B0;
        elsif s='1' and a='0' then nx_state<= A0;
        elsif s='1' and a='1' then nx_state<= A1;
        else nx_state<=B1;
        end if;
    end case;
end process;
end moore;
```

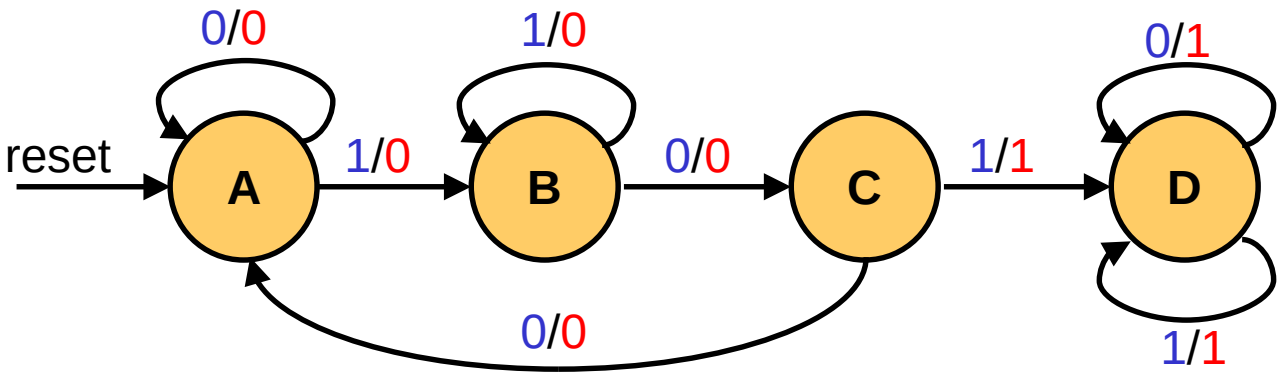




Example: Sequence Detection

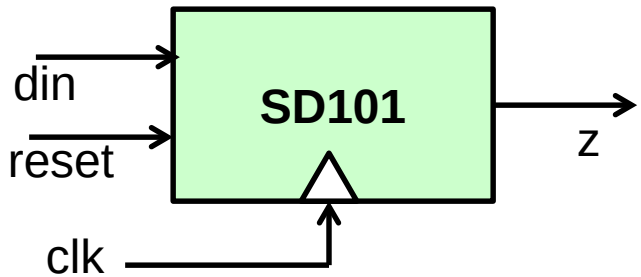
Build a Mealy FSM that has looks for sequence “101” in a serial input stream. When it detects the sequence, it outputs a 1 and holds that value until the machine is reset.





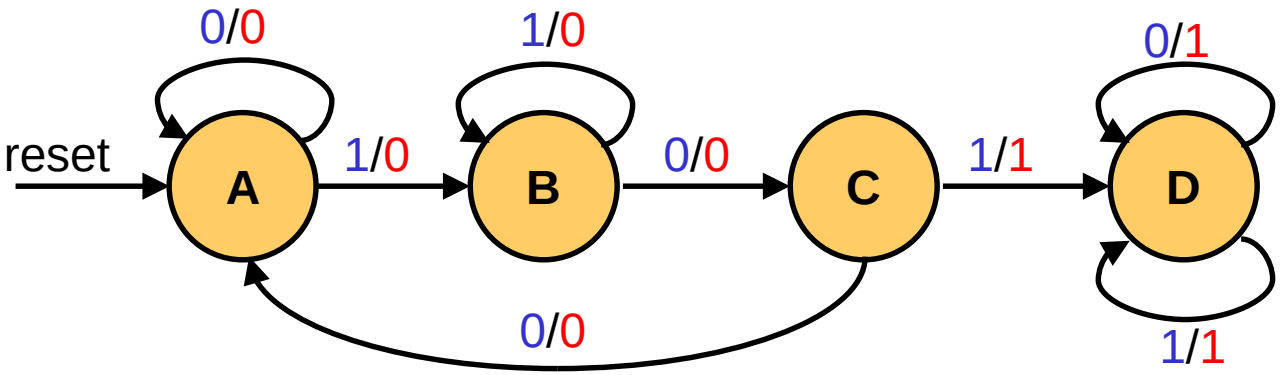
Sequence Detection: Clock Process

```
library  ieee;
use ieee.std_logic_1164.all;

entity SD101 is
    port(din, reset,clk: in std_logic;
        z: out std_logic);
end entity SD101;

architecture mealy of SD101 is
type state is (stateA, stateB, stateC, stateD);
signal pr_state, nx_state : state;
begin
p_clk: process (rst, clk)
    begin
        if (reset = '1') then
                pr_state <= stateA;
        elsif (clk'event and clk = '1') then
            pr_state <= nx_state;
        end if;
    end process;
```



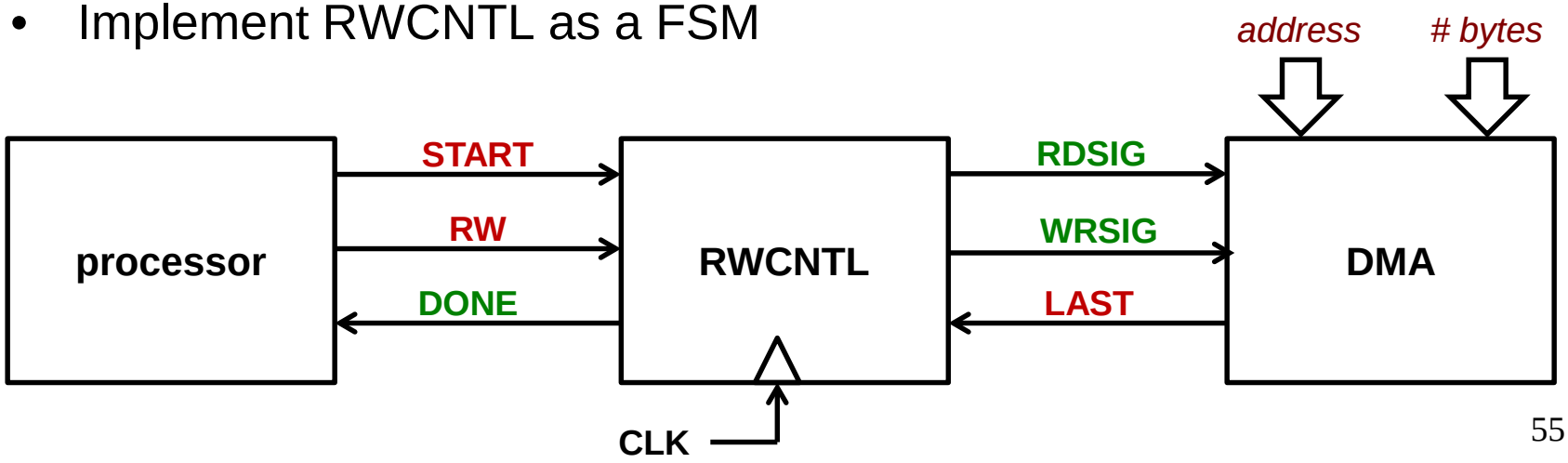
Sequence Detection: Combinational Process

```
p_comb: process (din,  pr_state)
    begin
        case pr_state is
        when stateA =>
            z <= '0';
            if (din = '1') then nx_state <= stateB;
            else nx_state <= stateA;
            end if;
        when stateB =>
            z <= '0';
            if (din = '1') then nx_state <= stateB;
            else nx_state <= stateC;
            end if;
        when stateC =>
            if (din = '1') then z <= '1';
                nx_state <= stateD;
            else z <= '0';
                nx_state <= stateA;
            end if;
        when stateD =>
            z <= '1';
            nx_state <= stateD;
        end case;
    end process;
```



Example: Read-Write Controller

- Suppose we have a read-write controller RWCNTL that acts as an interface between a processor and a DMA unit.
  - The DMA unit is used to transfer blocks of data between the processors memory and an external storage device like a hard disk.
  - Assume the processor has already communicated the memory address and byte count information to the DMA unit (this is part of the “data path”)
- Operation:
  - Processor raises start signal with RW=1 for reading, RW=0 for writing
  - RWCNTL sets (and holds) RDSIG or WRSIG to enable data transfer
  - When transfer is complete, DMA raises signal LAST
  - RWCNTL then resets RDSIG/WRSIG and raises DONE flag
- Implement RWCNTL as a FSM

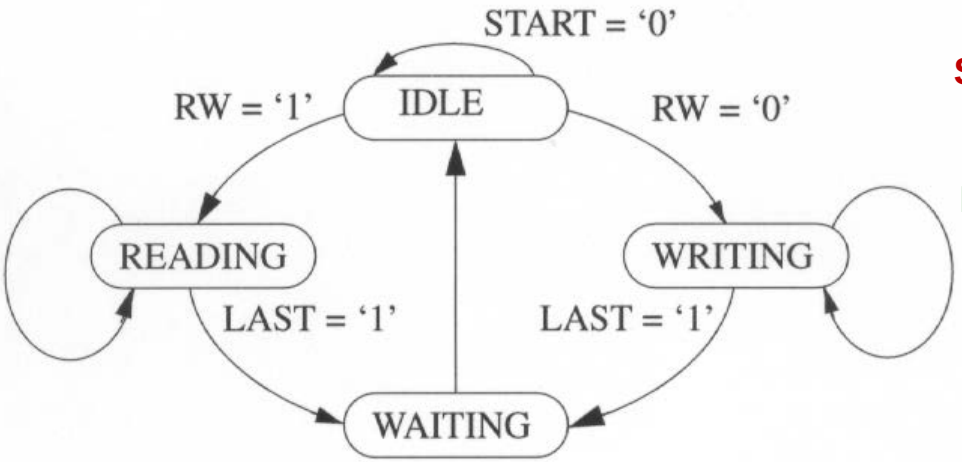


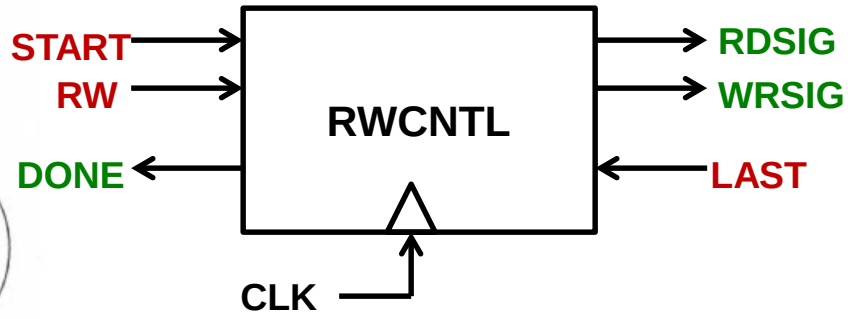
Example: Read-Write Controller

Specifications: RWCNTL starts at state IDLE, waiting for input signal START to go to '1' and then changes to either state READING or state WRITING depending on the value of RW input.

States READING and WRITING persist until input signal LAST goes to '1' which changes state to WAITING. After one clock cycle, state WAITING always goes to state IDLE.

The FSM has three outputs RDSIG, WRSIG, and DONE. They are '1' when they are in state READING, WRITING, and WAITING, respectively, otherwise they are '0'.

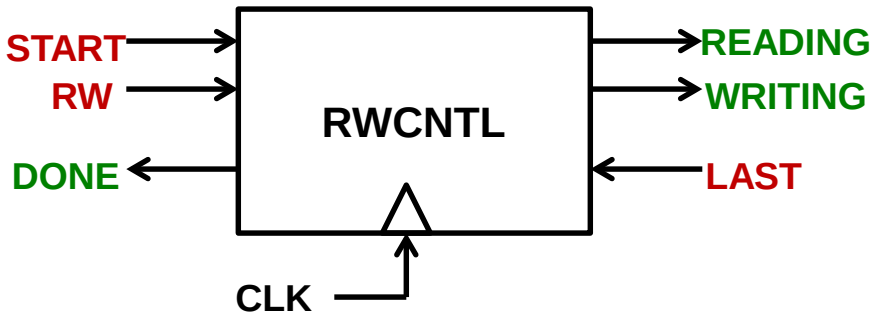




RWCNTL: Clock Process

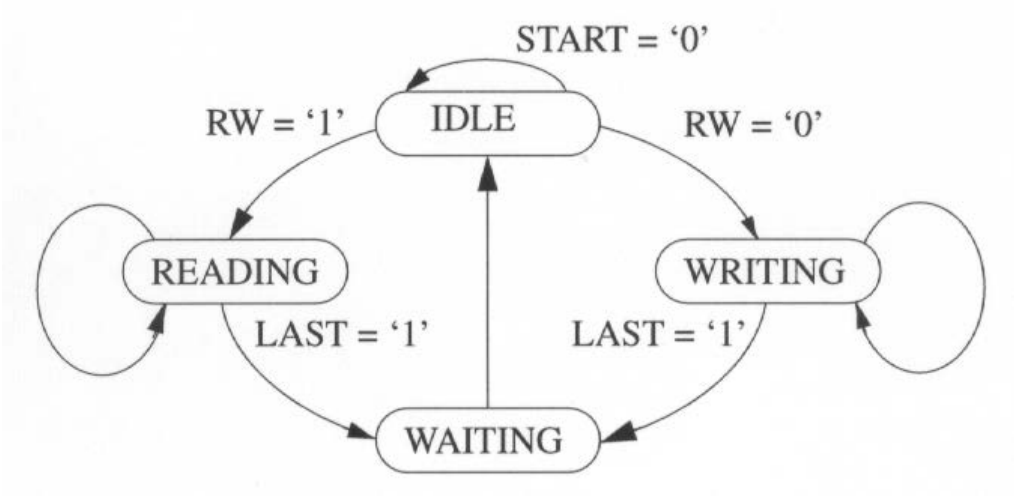
```
entity RWCNTL is
  port(
    CLK : in  std_logic;
    START : in  std_logic;
    RW    : in  std_logic;
    LAST  : in  std_logic;
    RDSIG : out std_logic;
    WRSIG : out std_logic;
    DONE  : out std_logic);
end entity RWCNTL;

architecture RTL of RWCNTL is
  type STATE is (IDLE, READING,
    WRITING, WAITING);
  signal PR_STATE, NX_STATE: STATE;
begin
```

```
seq : process
  begin
    wait until CLK'event and CLOCK = '1';
    PR_STATE <= NX_STATE;
  end process;
end RTL;
```

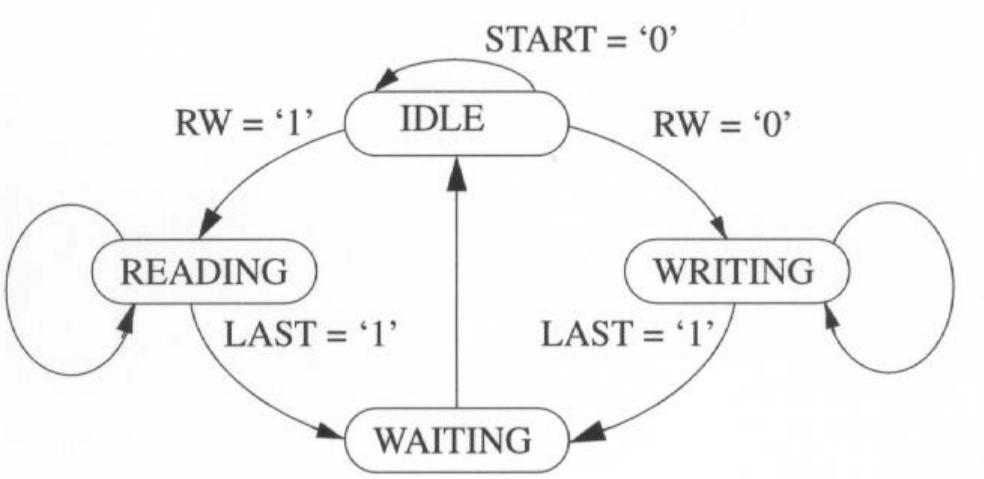


RWCNTL: Combinational Process

```
comb : process (PR_STATE, START, RW, LAST)
  begin
    DONE  <= '0'; RDSIG <= '0'; WRSIG <= '0';
    case PR_STATE is
      when IDLE =>
        if START = '0' then
          NX_STATE <= IDLE;
        elsif RW = '1' then
          NX_STATE <= READING;
        else
          NX_STATE <= WRITING;
        end if;
      when READING =>
        RDSIG <= '1';
        if LAST = '0' then
          NX_STATE <= READING;
        else
          NX_STATE <= WAITING;
        end if;
```

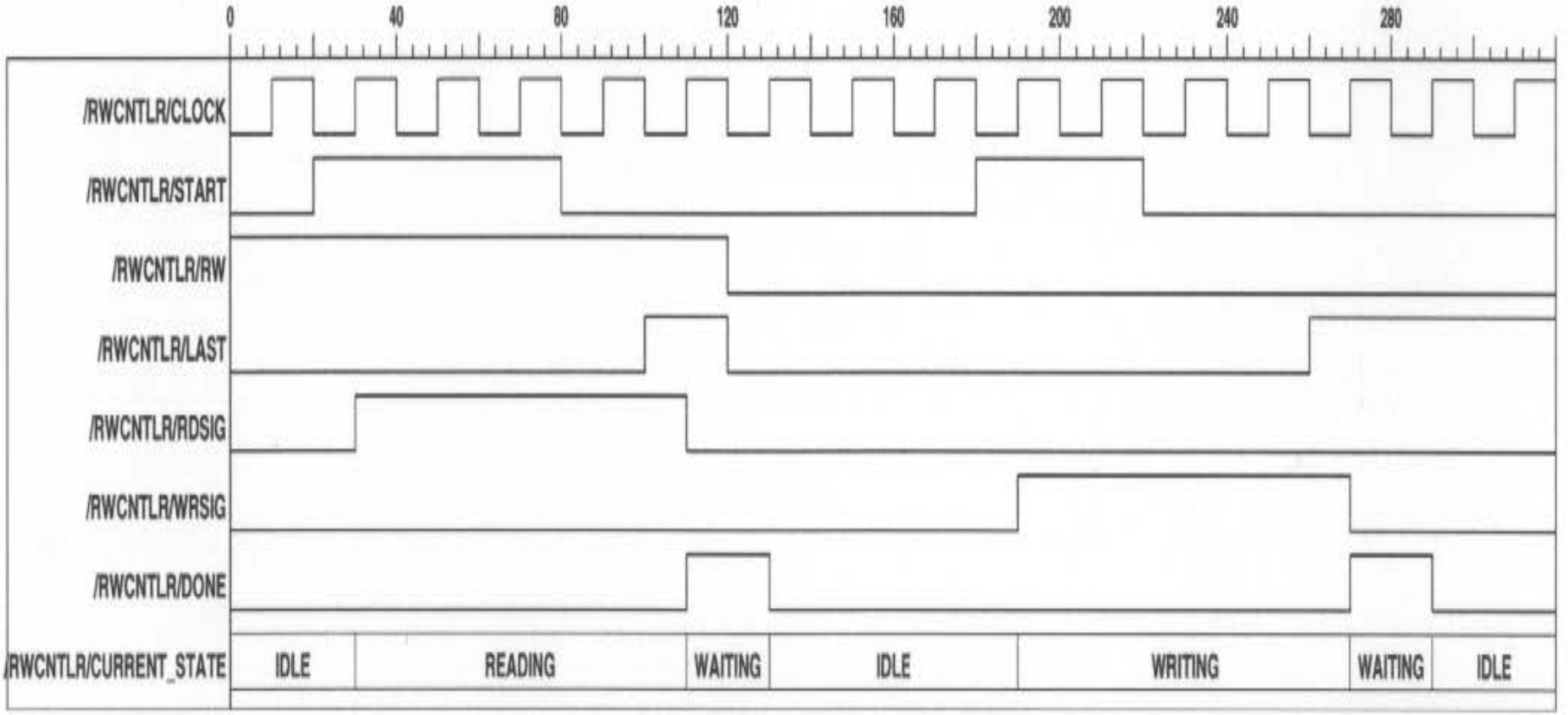


RWCNTL: Combinational Process (2)

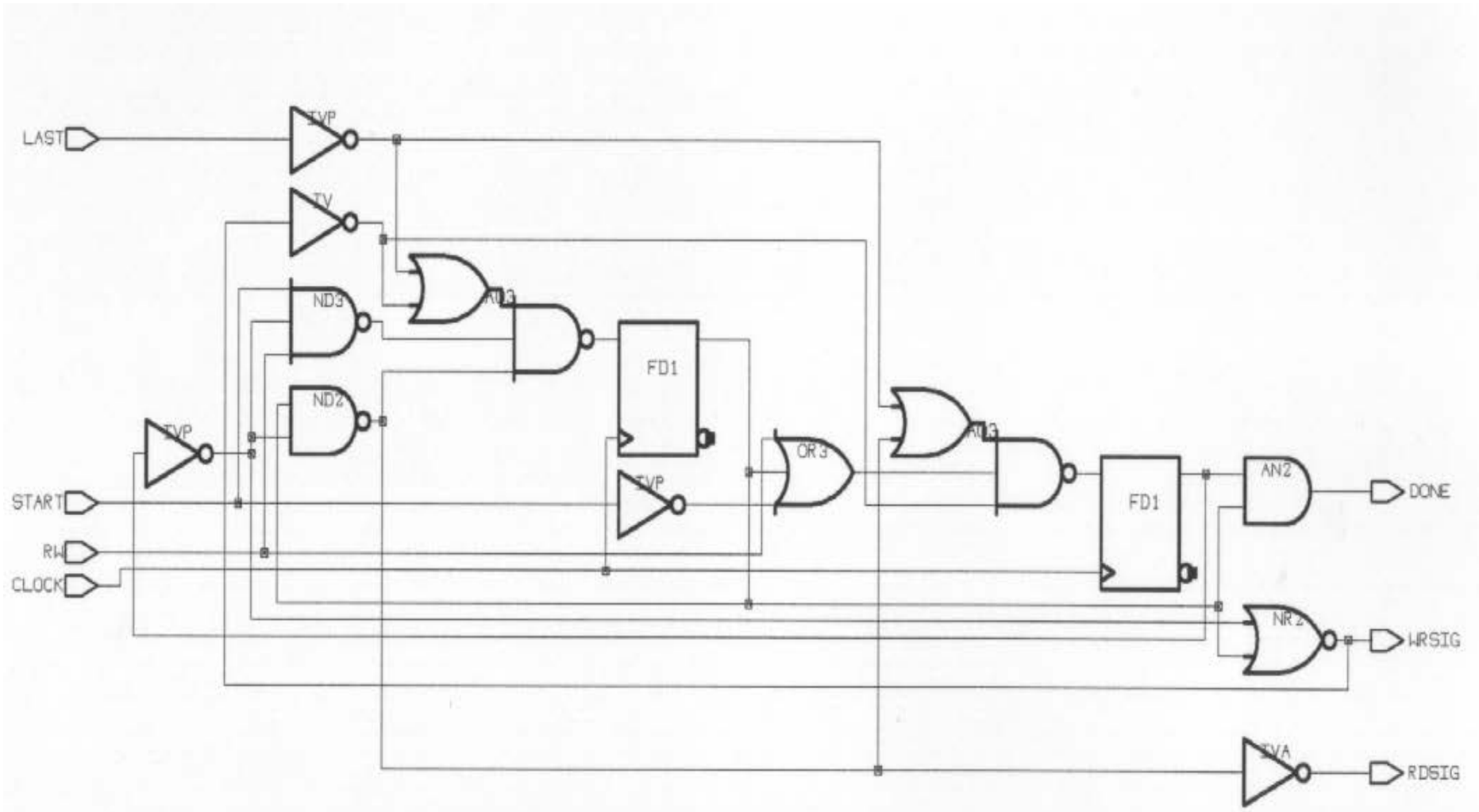



```
    when WRITING =>
      WRSIG <= '1';
        if LAST = '0' then
          NX_STATE <= WRITING;
      else
          NX_STATE <= WAITING;
      end if;
    when WAITING =>
      DONE <= '1';
      NX_STATE <= IDLE;
  end case;
end process;
```

Simulation Waveforms

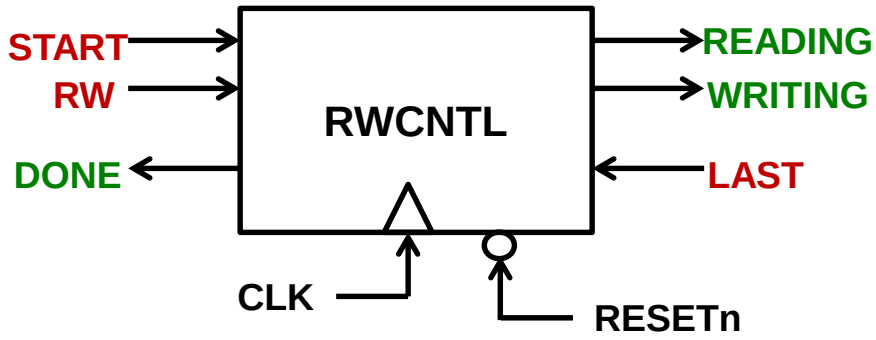


Synthesized RWCNTL

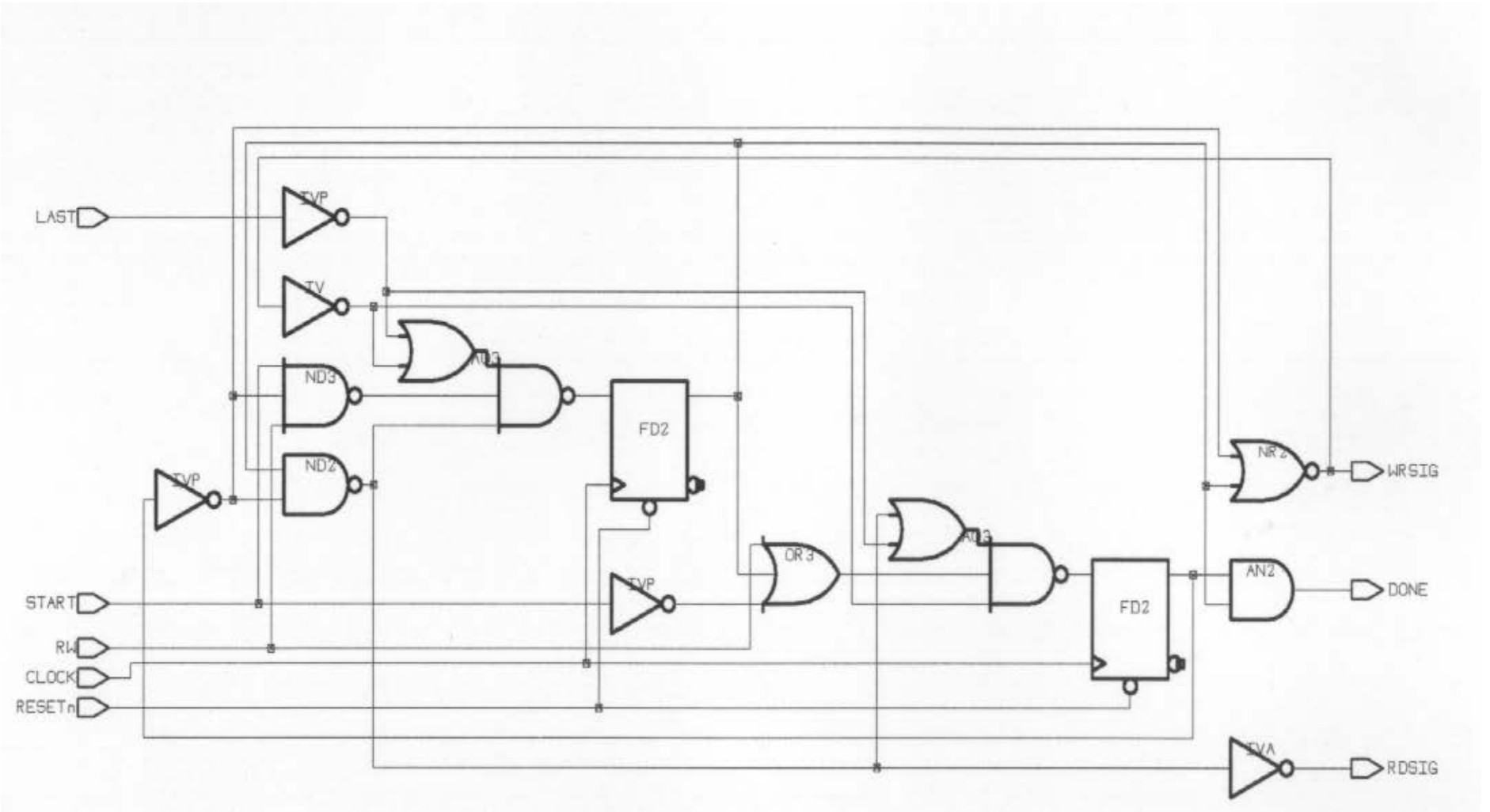


RWCNTL: Add Asynchronous Reset

```
 seq : process
   begin
     if (RESETn = '0') then
        PR_STATE <= IDLE;
      elsif (CLOCK'event and CLOCK = '1') then
        PR_STATE <= NX_STATE;
      end if;
  end process;
end RTL;
```



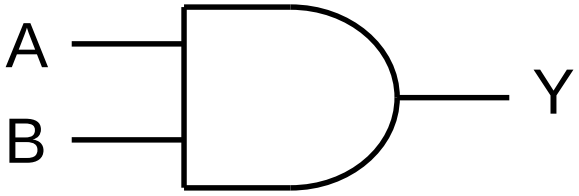
Synthesized RWCNTL with Asynch. Reset

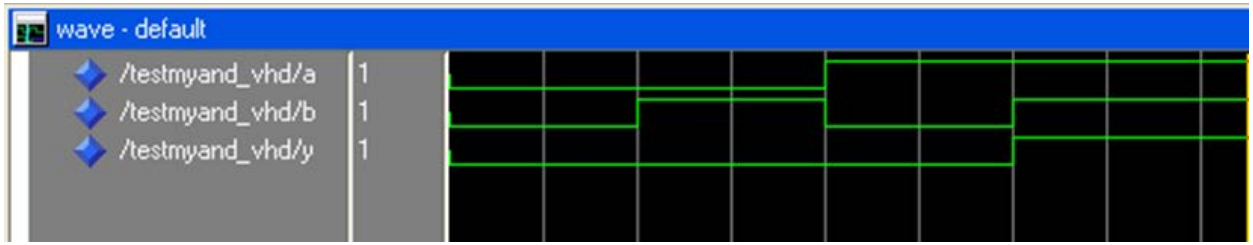


Test Bench Design

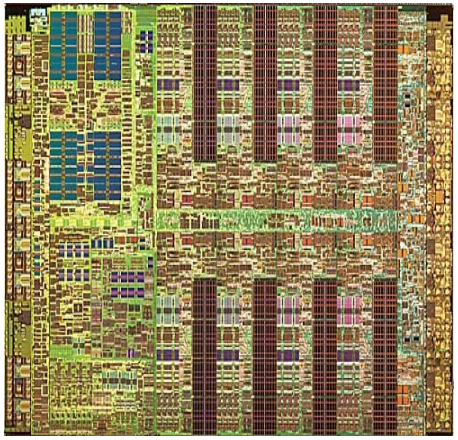
Testing Digital Circuits

- In our class examples, circuits were limited to a few input & outputs
  - Easy to set up test bench with processes to explicitly drive the inputs
  - Look at output waveforms to check correct operation





But how do we test this one?



Processor for SONY PlayStation 3

Test Vectors

- For large complex designs, designers will set up a suite of test vectors
  - One or more files that contain a sequence of inputs and expected outputs

inputs	*expected outputs*
0010011100000110	**11XX010110**
1101001101100101	**11X0111010**
0100011010011010	**1101101110**
1110011111010100	**0110001110**

- Input vectors are typically applied to circuit under test at some regular clock rate. Outputs are read at same rate and compared to expected outputs
- It can take many millions of vectors to adequately test a complex chip e.g., a microprocessor

Design Verification vs. Product Testing

- During the design process, we develop functional test vectors
  - Apply vectors similar to what might be expected in normal operation of chip
  - Look for correct functionality across broad range of expected inputs
  - Purpose of these functional vectors is to debug the design
  - Each time design is refined, we reapply the functional vectors to check that no mistake has been made in moving from behavioral modeling to implementation
- Once design has been synthesized into gates, and checked once more using the functional vectors, we develop a new set of manufacturing test vectors
  - Now checking for manufacturing defects

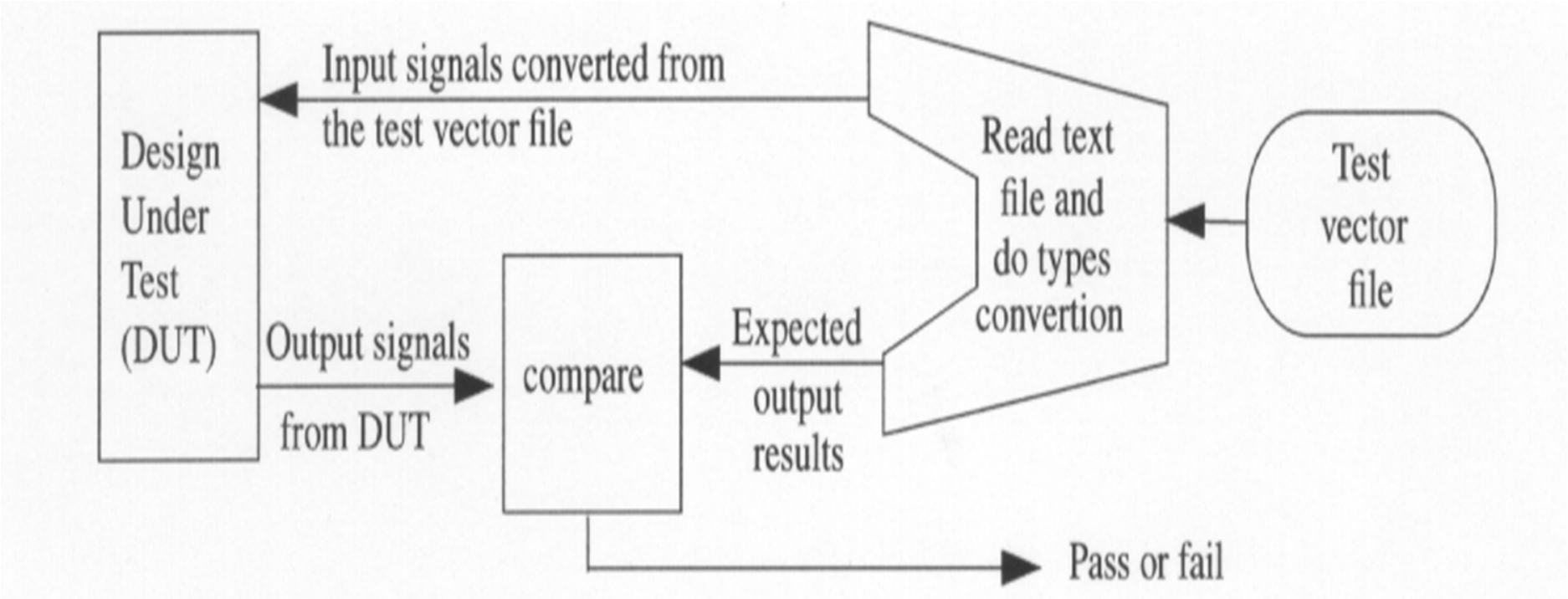
Testing for Manufacturing Defects

- Popular approach is to assume that all manufacturing faults can be modeled as a single node being stuck-at either a ‘0’ or a ‘1’
- Ideally, develop a set of test vectors that would show an error if any node is stuck-at ‘0’ or ‘1’
  - Not practical as it would take too many vectors to exhaustively test for all stuck at faults – tradeoff between test time and cost of sending bad chips into the field
  - Ratio of tested faults to all possible stuck-at faults is known as ***fault coverage*** – 90% is considered good for a large, complex chip

Production chip tester:

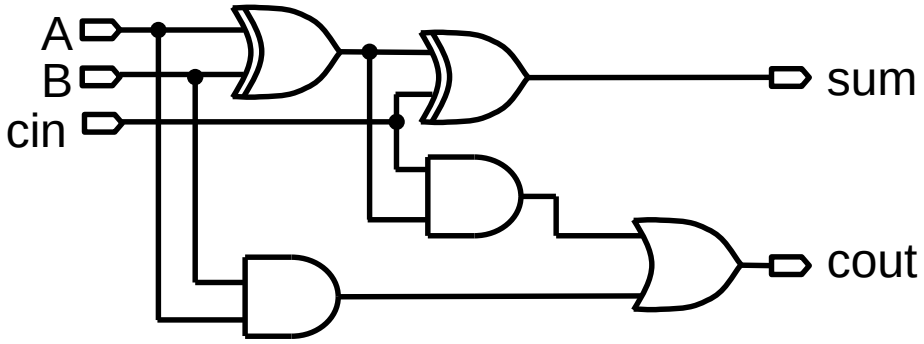
Developing a Test Bench

- Develop a set of test benches using simple example: full adder
- Move towards fully automated model:



Device Under Test: Full-Adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity fadder is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           cin : in  STD_LOGIC;
           sum : out  STD_LOGIC;
           cout : out  STD_LOGIC);
end fadder;

architecture gates of fadder is
signal S1,S2,S3: std_logic;
begin
   S1 <= A xor B after 5 ns;
   S2 <= cin and S1 after 3 ns;
   S3 <= A and B after 3 ns;
   sum <= S1 xor cin after 5ns;
   cout <= S2 or S3 after 3 ns;

end gates;
```

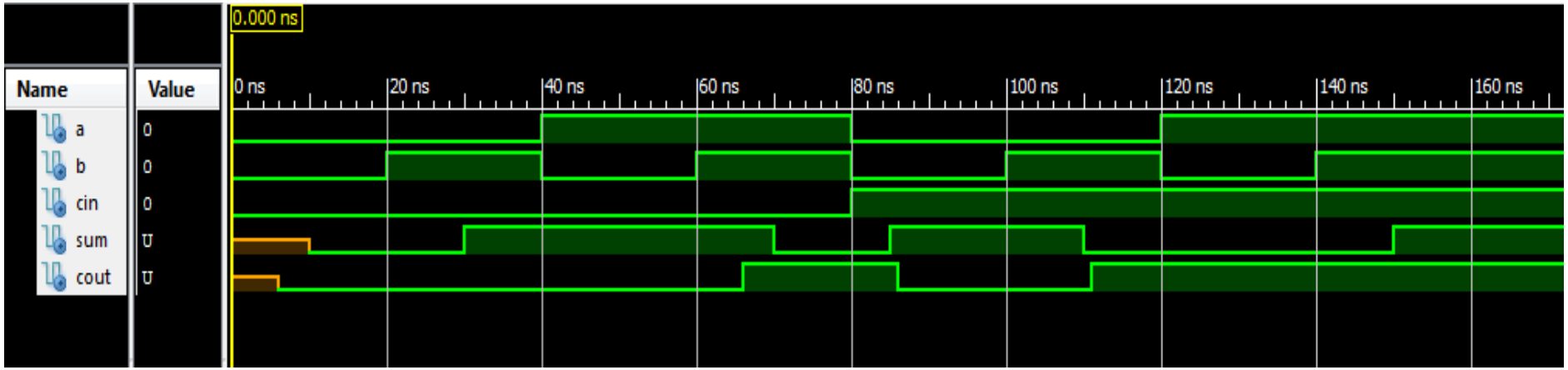


TB1: A Simple Test Bench

```
ENTITY fadd_tb1 IS
END fadd_tb1;

ARCHITECTURE behavior OF fadd_tb1 IS
     -- Component Declaration for UUT
     COMPONENT fadder
    PORT(
         A : IN  std_logic;
         B : IN  std_logic;
         cin : IN  std_logic;
         sum : OUT  std_logic;
         cout : OUT  std_logic
        );
    END COMPONENT;

    --Inputs
   signal A : std_logic := '0';
   signal B : std_logic := '0';
   signal cin : std_logic := '0';

   --Outputs
   signal sum : std_logic;
   signal cout : std_logic;

BEGIN
   -- Instantiate the UUT
   uut: fadder PORT MAP (
          A => A,
          B => B,
          cin => cin,
          sum => sum,
          cout => cout
        );
```

```
TB: process
   constant PERIOD: time:= 20ns;
   BEGIN

      A<='0'; B<='0'; cin<='0';
      wait for PERIOD;

      A<='0'; B<='1'; cin<='0';
      wait for PERIOD;

      A<='1'; B<='0'; cin<='0';
      wait for PERIOD;

      A<='1'; B<='1'; cin<='0';
      wait for PERIOD;

      A<='0'; B<='0'; cin<='1';
      wait for PERIOD;

      A<='0'; B<='1'; cin<='1';
      wait for PERIOD;

      A<='1'; B<='0'; cin<='1';
      wait for PERIOD;

      A<='1'; B<='1'; cin<='1';
      wait for PERIOD;

      wait; -- will wait forever
   END process;
END;
```

TB1: Visually Check Simulation Result




```
This is a Lite version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
ISim>
```

Assert Statement (revisited)

assert *boolean-expression*

[**report** *string-expression*]

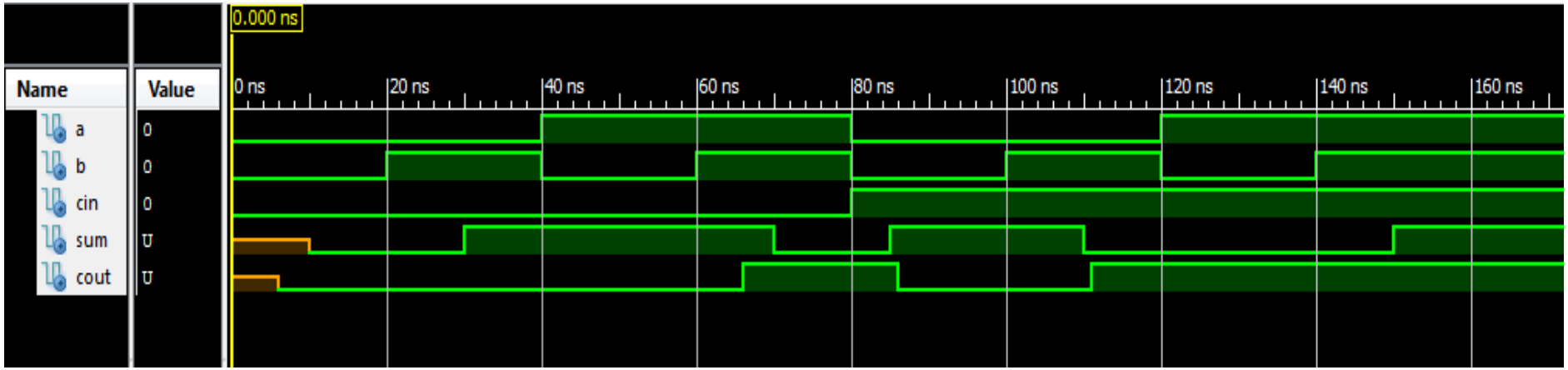
[**severity** *expression*];

- The *boolean-expression* is evaluated and if the expression is false, the *string-expression* specified in report statement is displayed in the simulator window
- The severity statement then indicates to the simulator what action should be taken in response to the assertion failure.
- Severity: NOTE, WARNING, ERROR, FAILURE.

TB2: Using Assert Statement

```
TB: process
   constant PERIOD: time:= 20ns;
   BEGIN

      A<='0'; B<='0'; cin<='0';
      wait for PERIOD;
      assert(sum='0' and cout='0')
         report "Test FAILED" severity error;

      A<='0'; B<='1'; cin<='0';
      wait for PERIOD;
      assert(sum='1' and cout='0')
         report "Test FAILED" severity error;

      A<='1'; B<='0'; cin<='0';
      wait for PERIOD;
      assert(sum='1' and cout='0')
         report "Test FAILED" severity error;

      A<='1'; B<='1'; cin<='0';
      wait for PERIOD;
      assert(sum='0' and cout='1')
         report "Test FAILED" severity error;

      A<='1'; B<='1'; cin<='0';
      wait for PERIOD;
      assert(sum='0' and cout='1')
         report "Test FAILED" severity error;

      A<='0'; B<='0'; cin<='1';
      wait for PERIOD;
      assert(sum='1' and cout='0')
         report "Test FAILED" severity error;

      A<='0'; B<='1'; cin<='1';
      wait for PERIOD;
      assert(sum='0' and cout='1')
         report "Test FAILED" severity error;

      A<='1'; B<='0'; cin<='1';
      wait for PERIOD;
      assert(sum='0' and cout='1')
         report "Test FAILED" severity error;

      A<='1'; B<='1'; cin<='1';
      wait for PERIOD;
      assert(sum='1' and cout='1')
         report "Test FAILED" severity error;

      wait; -- will wait forever
   END process;
END;
```

TB2: Using Assert – no errors

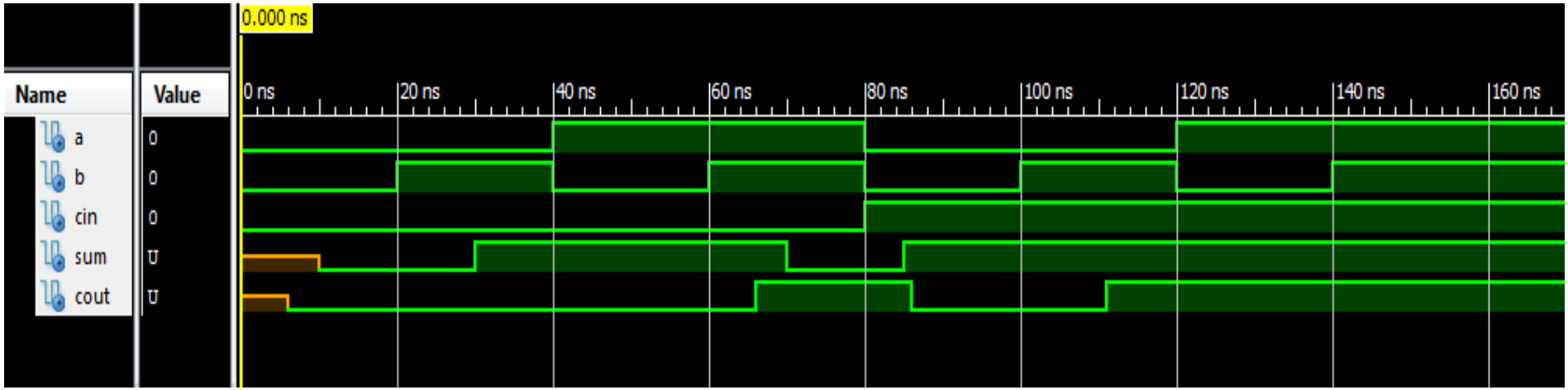



```
This is a Lite version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
ISim>
```

TB2: Using Assert – Introduce error

```
architecture gates of fadder is
signal S1,S2,S3: std_logic;
begin
   S1 <= A xor B after 5 ns;
   S2 <= cin and S1 after 3 ns;
   S3 <= A and B after 3 ns;
   sum <= S1 or cin after 5ns;
   cout <= S2 or S3 after 3 ns;

end gates;
```

Use or instead of xor



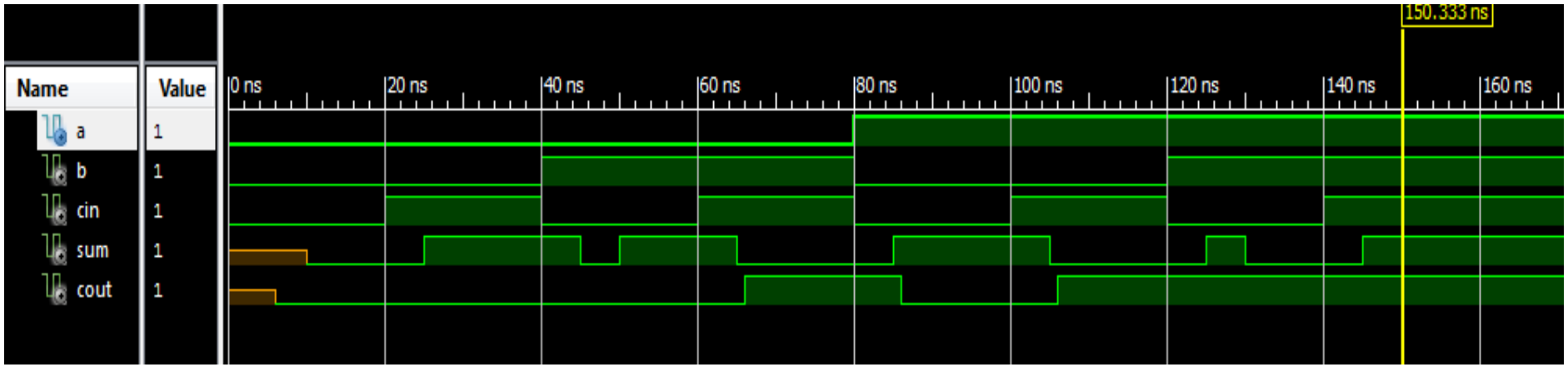

```
This is a Lite version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
at 120 ns: Error: Test FAILED
at 140 ns: Error: Test FAILED
ISim>
```

TB3: Using Test Vector Array

```
ARCHITECTURE behavior OF fadd_tb3 IS
     -- Component Declaration for UUT
    COMPONENT fadder
    PORT(
         A,B,cin : IN  std_logic;
         sum,cout : OUT  std_logic
        );
    END COMPONENT;
   --Inputs
   signal A : std_logic := '0';
   signal B : std_logic := '0';
   signal cin : std_logic := '0';
   --Outputs
   signal sum : std_logic;
   signal cout : std_logic;

   type test_array is array(integer range<>)
      of std_logic_vector(0 to 4);
   constant test_vector: test_array(0 to 7):=(
   ('0', '0', '0', '0', '0'),
   ('0', '0', '1', '1', '0'),
   ('0', '1', '0', '1', '0'),
   ('0', '1', '1', '0', '1'),
   ('1', '0', '0', '1', '0'),
   ('1', '0', '1', '0', '1'),
   ('1', '1', '0', '0', '1'),
   ('1', '1', '1', '1', '1')
   );
```

```
BEGIN
   -- Instantiate the UUT
   uut: fadder PORT MAP (
          A => A,
          B => B,
          cin => cin,
          sum => sum,
          cout => cout
        );
TB: process
   constant PERIOD: time:= 20ns;
   variable testv: std_logic_vector(0 to 4);
   BEGIN

      for i in test_vector'range loop
         testv := test_vector(i);
         A <= testv(0);
         B <= testv(1);
         cin <= testv(2);
         wait for PERIOD;
         assert (sum=testv(3) and cout=testv(4))
            report "Test FAILED" severity error;
         end loop;

      wait; -- will wait forever
   END process;
END;
```

TB3: Using Test Vector Array – no errors




```
# run 1000 ns
Simulator is doing circuit initialization process.
Finished circuit initialization process.
ISim>
```

Reading and Writing Files – TEXTIO Package

- A file is a class of object in VHDL (like signal, variable & constant). Each file has a type.
- The standard VHDL library contains the TEXTIO package which provides a set of file types, data types and I/O procedures for simple text I/O to and from files
- To make the package visible:

  use std.textio.**all**;

- TEXTIO read and write procedures are available for predefined types **bit, bit_vector, character** and **string**
- The ieee.std_logic_textio package overloads these procedures to support std_logic and std_logic_vector

Declaring and Opening Files

- New Types:

  text -- a file of character strings
  line – a string (to or from a text file)

- Example Declarations

  file testfile: text; -- testfile is "file handle"

  variable L: line; -- L is a single line buffer

- Procedure to open a file:

  file_open(*file_handle*, *filename*, *open_kind*);

  – where *open_kind* is one of (read_mode, write_mode or append_mode), for example:

  file_open (testfile, "my_file.txt", read_mode);

Reading Files

- readline (*file_handle*, *line_buffer*);
  - Read one line from "text" file *file_handl*e into *line_buffer*
- read(*line_buffer*, *value*);
  - Read one item from *line_buffer* into variable *value*.
  - Variable *value* can be bit, bit_vector, character or string
  - Variable can also be std_logic or std_logic vector if IEEE std_logic_textio package is used.
- endfile(*file_handle*); --returns boolean (TRUE if at EOF)
- For example:

```
variable buf_in: line;
variable bit0: std_logic;
begin
    readline(my_file, buf_in);
    read(buf_in, bit0);
```

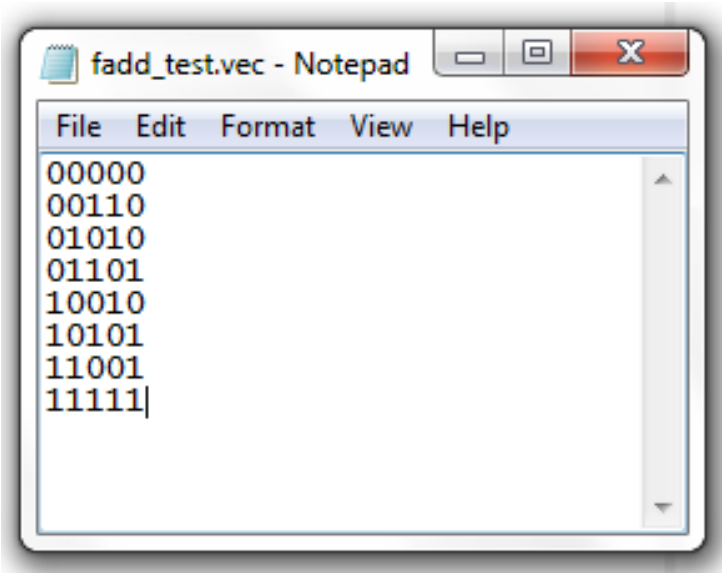
Writing Files

- writeline (*file_handle*, *line_buffer*);
  - Write one line to "text" file *file_handl*e from *line_buffer*
- write(*line_buffer*, *value*);
  - Write variable value into *line_buffer*
  - Variable *value* can be bit, bit_vector, character or string
  - Variable can also be std_logic or std_logic vector if IEEE std_logic_textio package is used.
- For example:

```
variable buf_out: line;
variable abc : bit_vector (3 downto 0);
begin
    write(buf_out, “abc is”);
    write(buf_out, abc);
    writeline(my_file, buf_in);
```

TB4: Reading Vectors from File

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE std.textio.all;
USE ieee.std_logic_textio.ALL;
```




```
TB: process
      constant PERIOD: time:= 20ns;
      file vec_file: text;
      variable buf_in: line;
      variable testv: std_logic_vector(0 to 4);
      BEGIN

        file_open(vec_file,"fadd_test.vec", read_mode);
        while not endfile(vec_file) loop
           readline(vec_file, buf_in);
           read(buf_in, testv);
           --apply the stimulus from the vector
           A <= testv(0);
           B <= testv(1);
           cin <= testv(2);
           wait for PERIOD;
           assert (sum=testv(3) and cout=testv(4))
              report "Test FAILED" severity error;
        end loop;

      wait; -- will wait forever
   END process;
END;
```

TB5: Writing Results to File

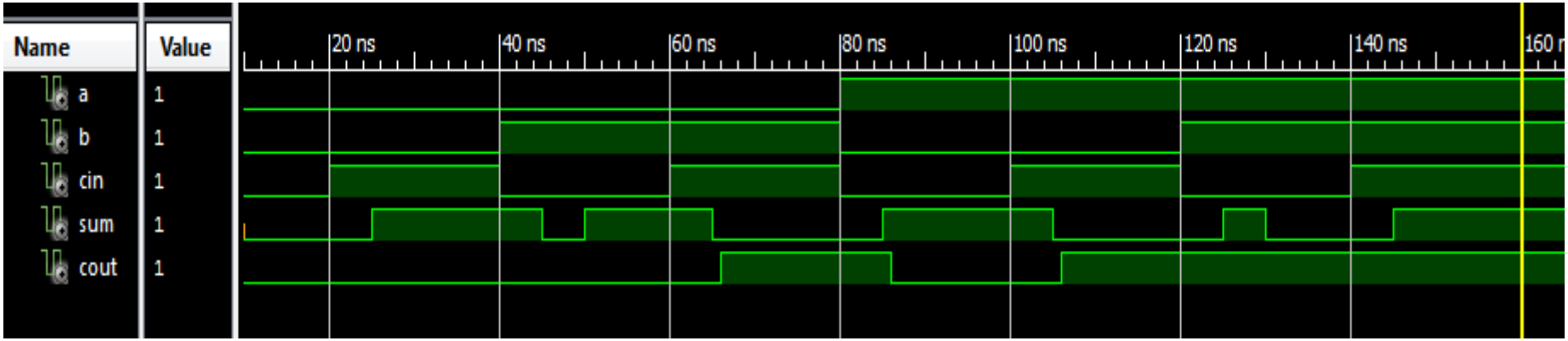
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE std.textio.all;
USE ieee.std_logic_textio.ALL;

ENTITY fadd_tb5 IS
END fadd_tb5;

ARCHITECTURE behavior OF fadd_tb5 IS
     -- Component Declaration for UUT
    COMPONENT fadder
    PORT(
         A,B,cin : IN  std_logic;
         sum,cout : OUT  std_logic
        );
    END COMPONENT;
```

```
   --Inputs
   signal A : std_logic := '0';
   signal B : std_logic := '0';
   signal cin : std_logic := '0';
   --Outputs
   signal sum : std_logic;
   signal cout : std_logic;

BEGIN
   -- Instantiate the UUT
   uut: fadder PORT MAP (
          A => A,
          B => B,
          cin => cin,
          sum => sum,
          cout => cout
        );
```

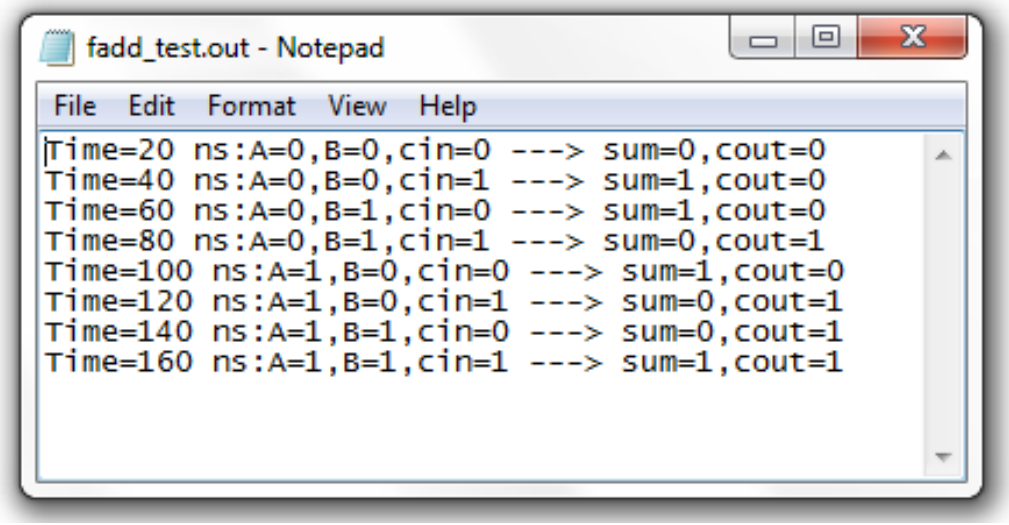
TB5: Writing Results to File (2)

```
TB: process
      constant PERIOD: time:= 20ns;
      file vec_file, result_file: text;
      variable buf_in, buf_out: line;
      variable testv: std_logic_vector(0 to 4);
      BEGIN
         file_open(vec_file,"fadd_test.vec", read_mode);
         file_open(result_file,"fadd_test.out", write_mode);
         while not endfile(vec_file) loop
            readline(vec_file, buf_in);
            read(buf_in, testv);
            --apply the stimulus from the vector
            A <= testv(0);
            B <= testv(1);
            cin <= testv(2);
            wait for PERIOD;
            assert (sum=testv(3) and cout=testv(4))
               report "Test FAILED" severity error;
            write(buf_out, "Time="); write(buf_out, now);
            write(buf_out,":A="); write(buf_out,testv(0));
            write(buf_out,",B="); write(buf_out,testv(1));
            write(buf_out,",cin="); write(buf_out,testv(2));
            write(buf_out," ---> sum="); write(buf_out,sum);
            write(buf_out,",cout="); write(buf_out,cout);
            writeline(result_file,buf_out);
         end loop;
      wait; -- will wait forever
   END process;
END;
```

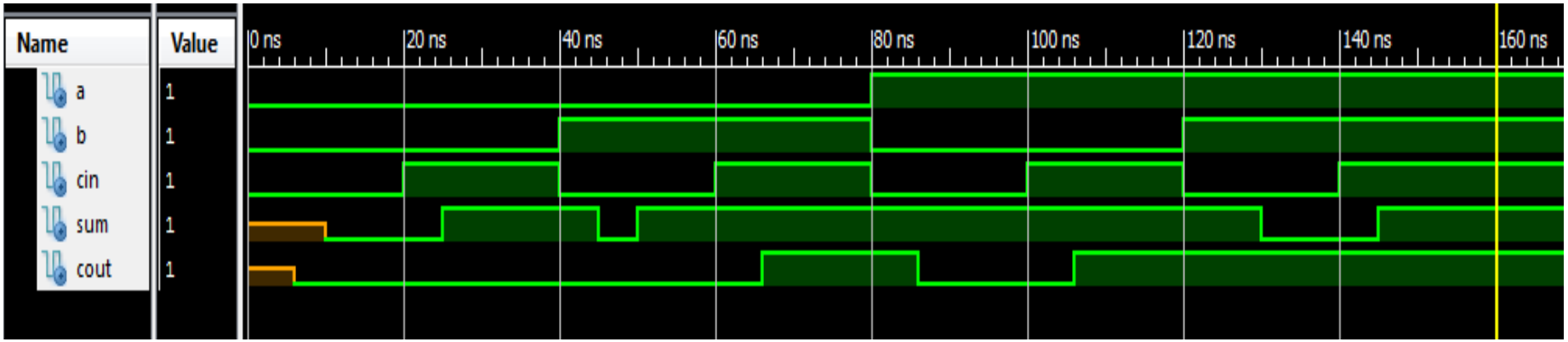
TB5: Writing Results to File (2)



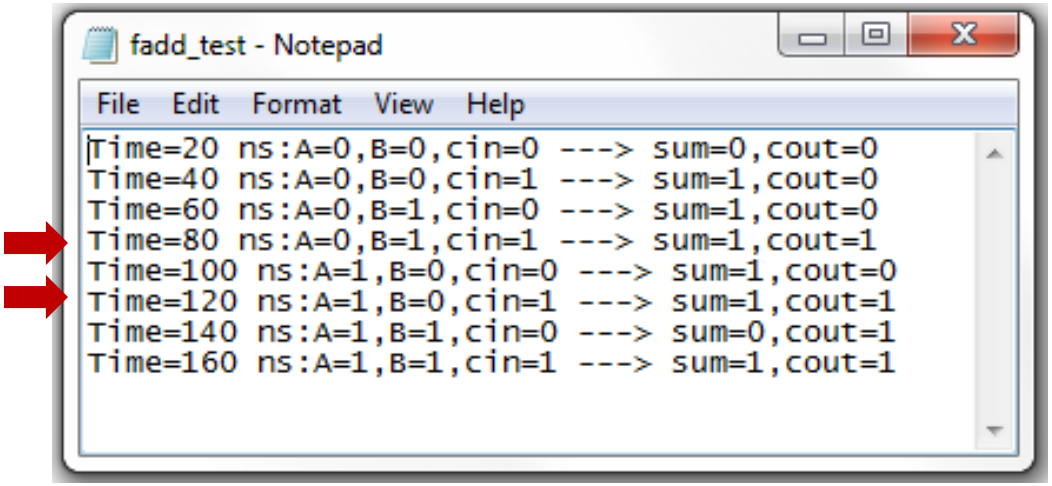

```
This is a Lite version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
ISim>
```



TB5: Writing Results with OR/XOR error




```
architecture gates of fadder is
signal S1,S2,S3: std_logic;
begin
    S1 <= A xor B after 5 ns;
    S2 <= cin and S1 after 3 ns;
    S3 <= A and B after 3 ns;
    sum <= S1 or cin after 5ns;
    cout <= S2 or S3 after 3 ns;

end gates;
```




```
This is a Lite version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
at 80 ns: Error: Test FAILED
at 120 ns: Error: Test FAILED
ISim>
```