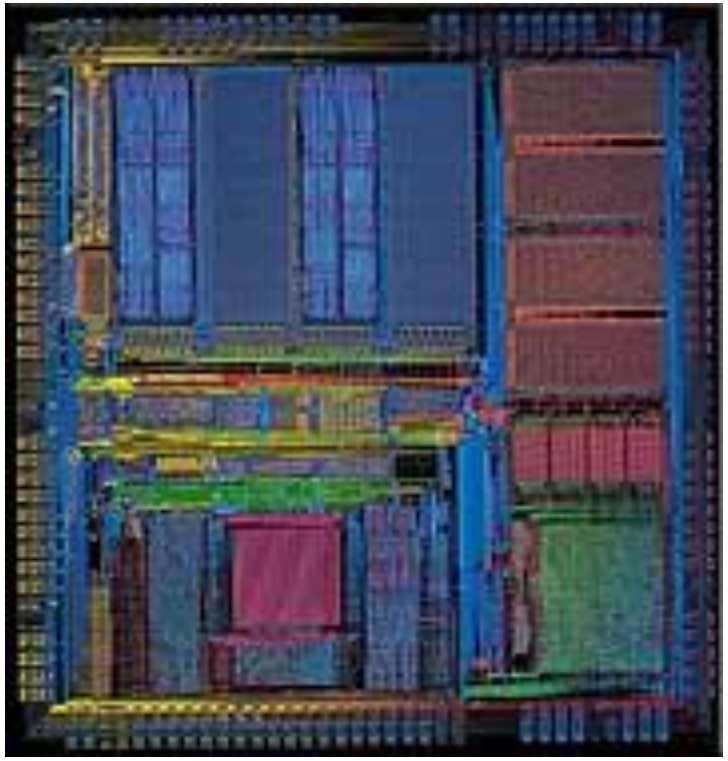
VHDL

Parviz Keshavarzi



Quick Review

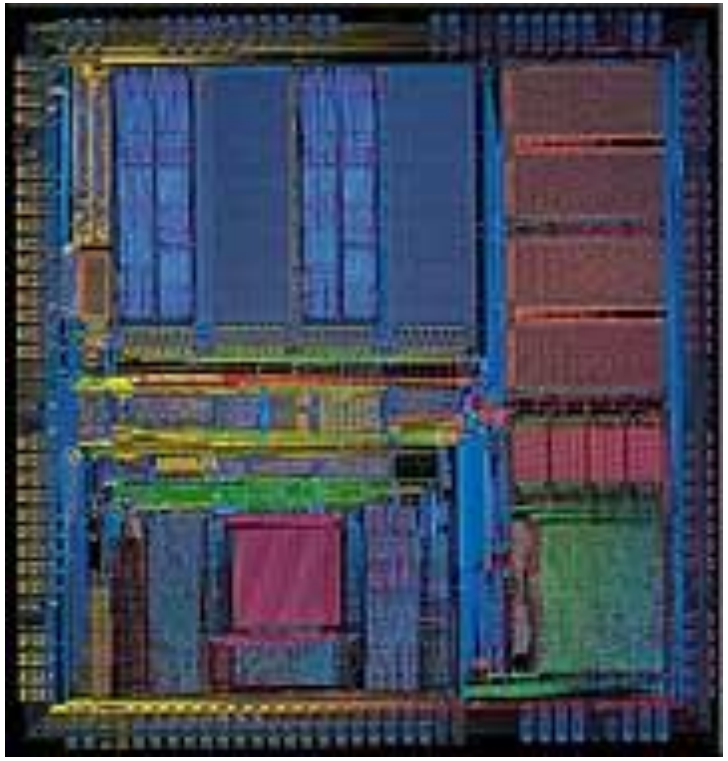
Feb, 2022

Acknowledgement

These slides used or are derived from the following source:

- Dr. Karam Chatha's VHDL course taught at Arizona State University.
- Melnik
- Jason D. Bakos "VHDL and HDL Designer Primer" university of South Carolina
- Tuft Slides
- Nitin Yogi, Digital Logic Circuits course (yoginit@auburn.edu)
- ECE 448 George Mason University

Part 1:

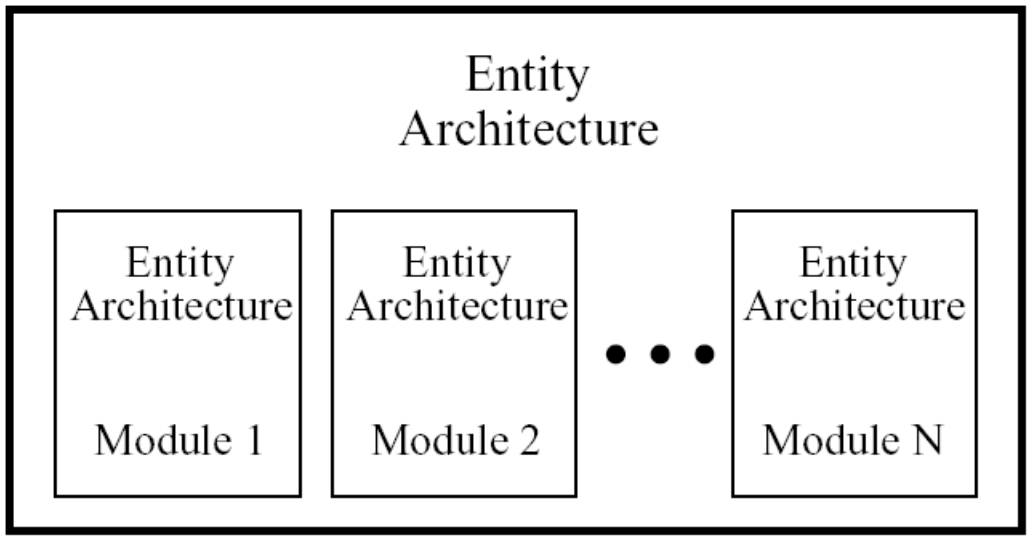


VHDL In a Glance

VHDL Program Structure

- Every VHDL program consists of two main parts:
  - Entity
  - Architecture
- Entity describes Inputs and outputs of a design
- Architecture describes functionality of a design

VHDL Program Structure




```
entity entity-name is
   [port(interface-signal-declaration);]
end [entity] [entity-name];

architecture architecture-name of entity-name is
   [declarations]
begin
   architecture body
end [architecture] [architecture-name];
```

VHDL Program Structure Example

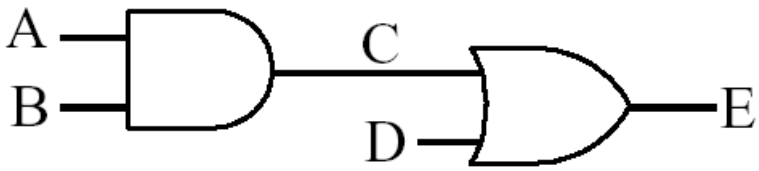
- *Concurrent signal assignment*

<=

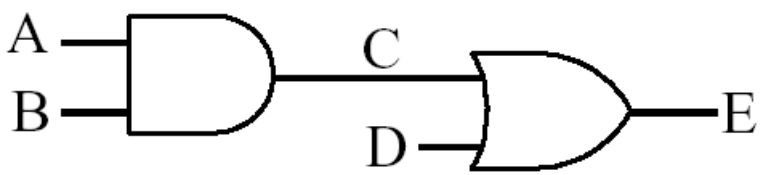
```
target_signal <= expression;
```

- ***Data-flow VHDL Example:***

```
ENTITY AndOrGates IS
        PORT (A, B, D     : IN  BIT;
              E           : OUT BIT);
END AndOrGates;

  ARCHITECTURE dataflow OF AndOrGates IS
  SIGNAL  C:  BIT;
  BEGIN
          C  <= A  AND  B;
          E  <= C  OR  D;
  END dataflow ;
```



VHDL Description of Combinational Networks



Concurrent Statements

```
C <= A and B  after 5 ns;
E <= C or D   after 5 ns;
```

If delay is not specified, “delta” delay is assumed

```
C <= A and B;
E <= C or D;
```

Order of concurrent statements is not important

```
E <= C or D;
C <= A and B;
```

This statement executes repeatedly

```
CLK <= not CLK after 10 ns;
```

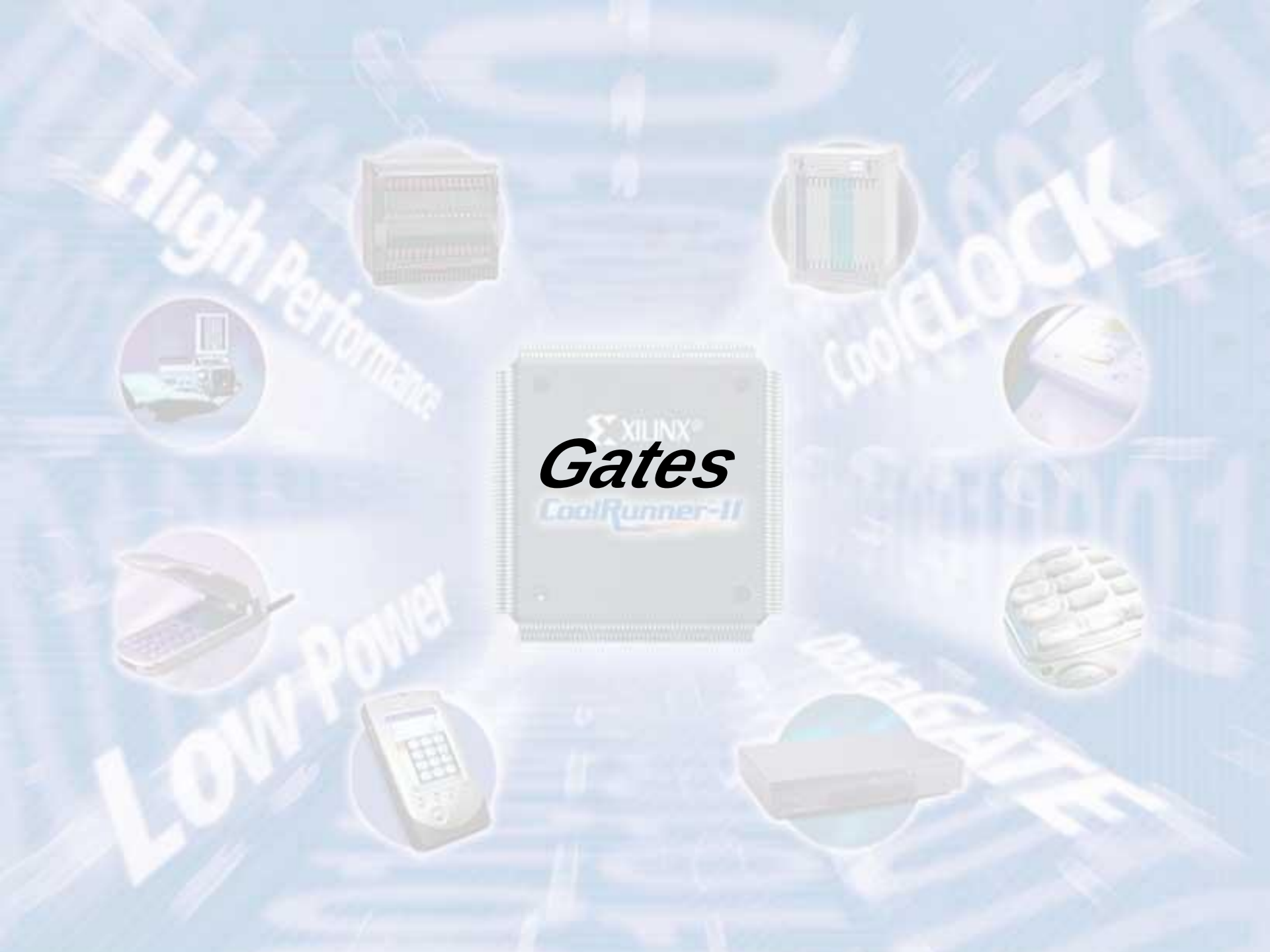
This statement causes a simulation error

```
CLK <= not CLK;
```

Data-Flow VHDL

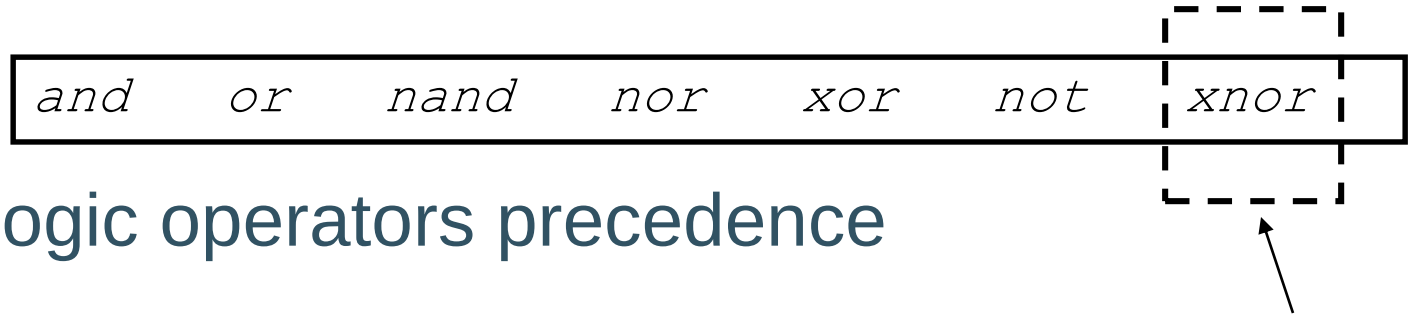
Concurrent Statements

- ***simple*** ***concurrent signal assignment***
  (⇐)
- ***conditional*** ***concurrent signal assignment***
  (when-else)
- ***selected*** ***concurrent signal assignment***
  (with-select-when)

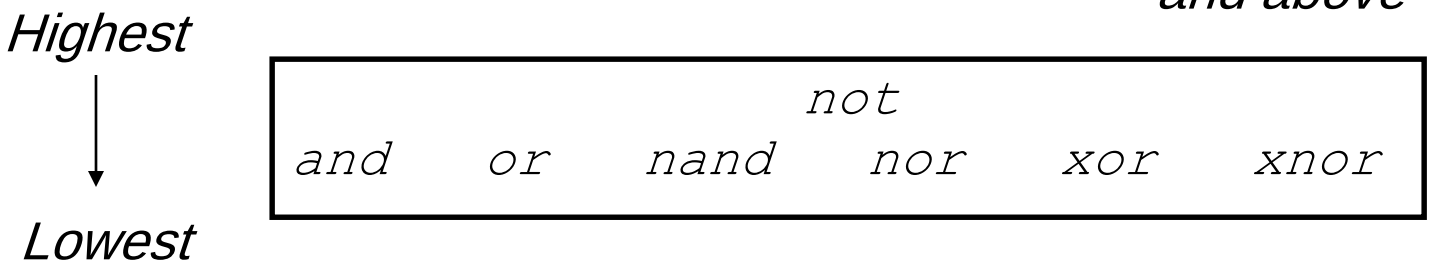

High Performance
Low Power
XILINX
CoolRunner-II
Gates

Logic Operators

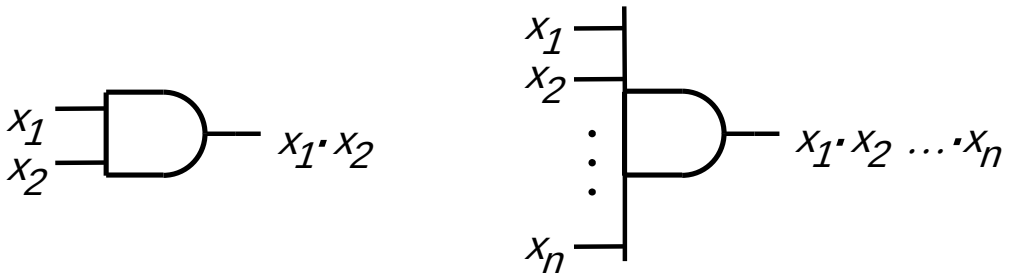
- Logic operators



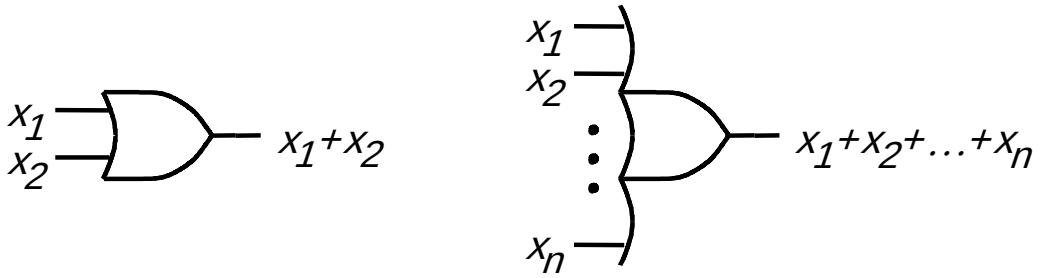
- Logic operators precedence



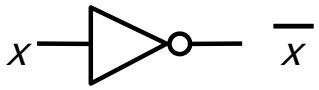
Basic Gates – AND, OR, NOT



(a) AND gates

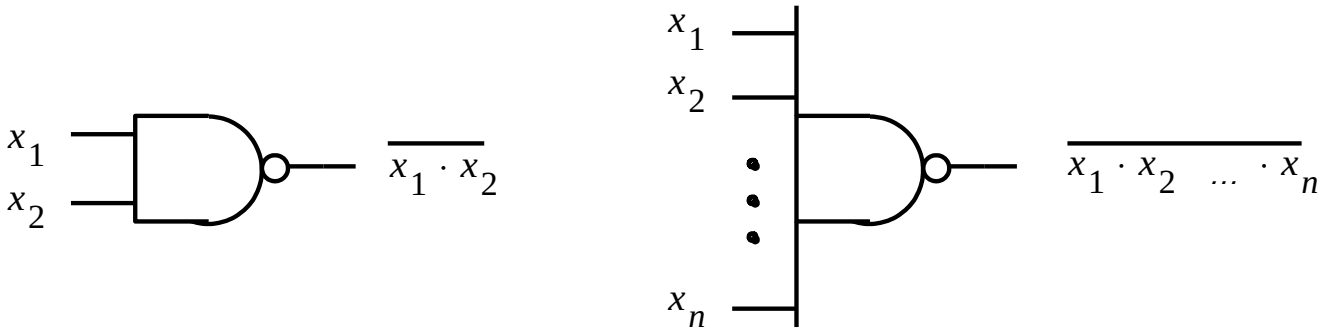


(b) OR gates

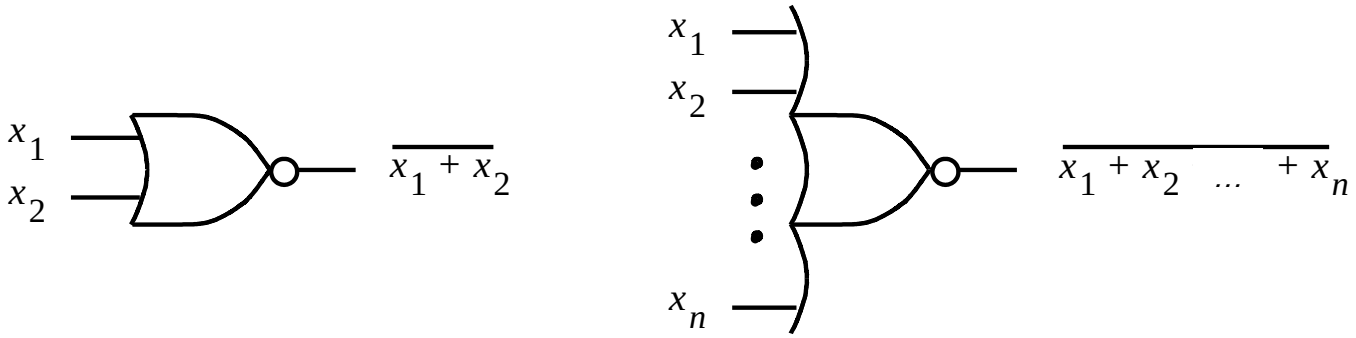


(c) NOT gate

Basic Gates – NAND, NOR

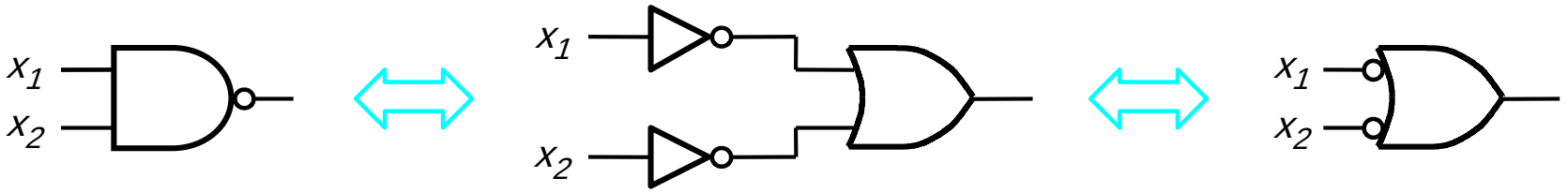


(a) NAND gates

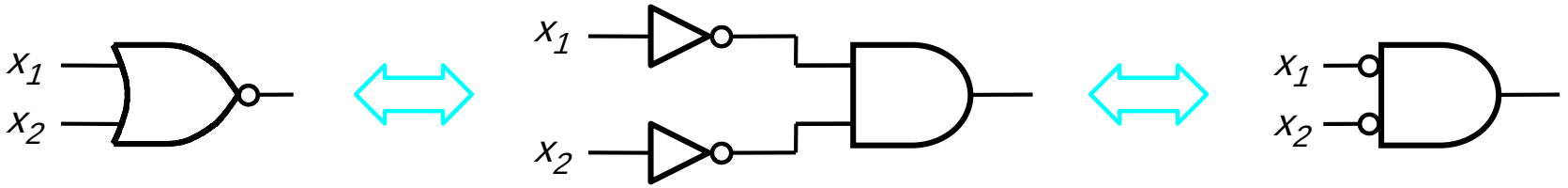


(b) NOR gates

DeMorgan's Theorem and other symbols for NAND, NOR



(a) $\overline{x_1 x_2} = \overline{x}_1 + \overline{x}_2$

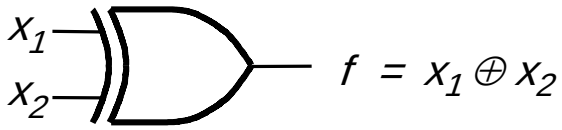


(b) $\overline{x_1 + x_2} = \overline{x}_1 \overline{x}_2$

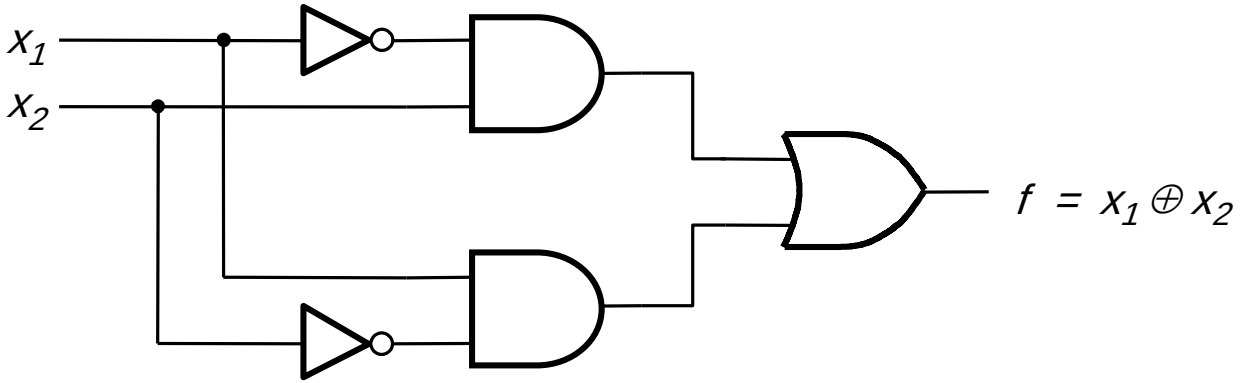
Basic Gates – XOR

x_1	x_2	$f = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

(a) Truth table



(b) Graphical symbol

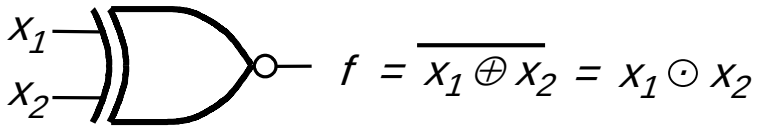


(c) Sum-of-products implementation

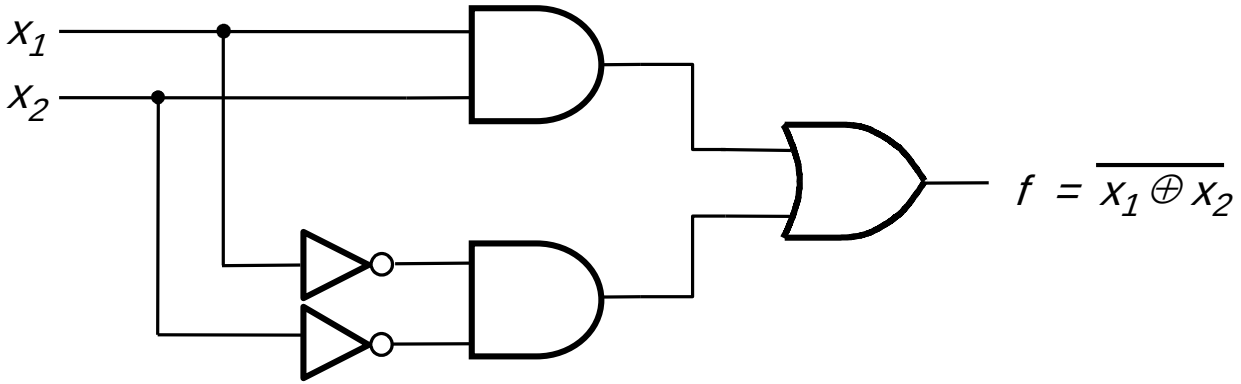
Basic Gates – XNOR

x_1	x_2	$f = \overline{x_1 \oplus x_2}$
0	0	1
0	1	0
1	0	0
1	1	1

(a) Truth table

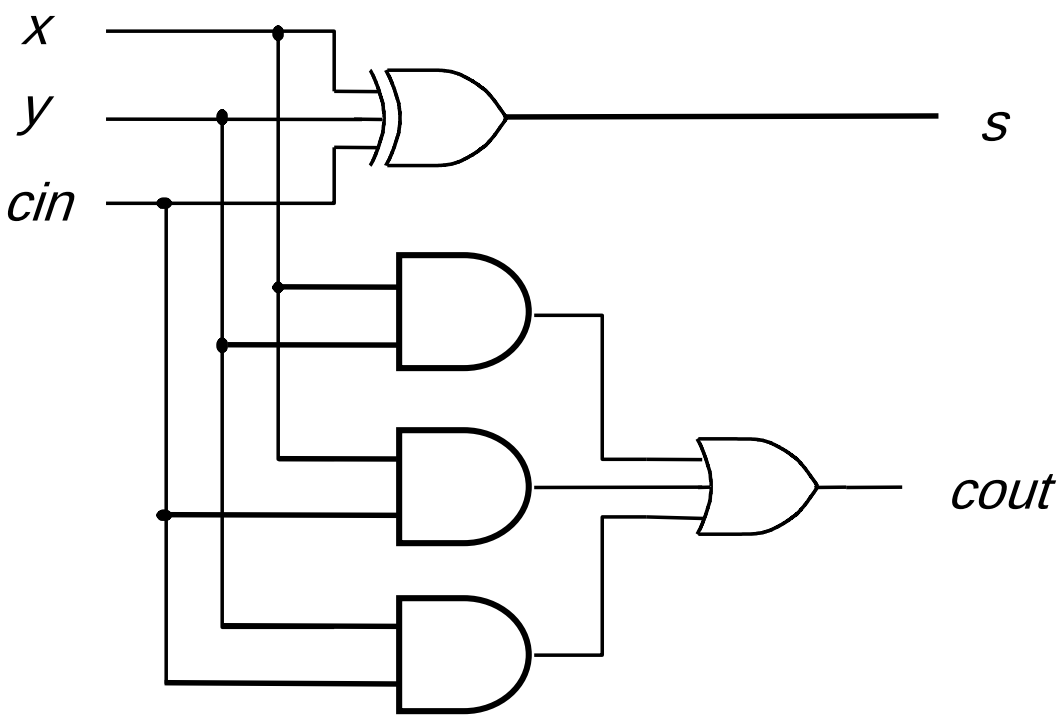


(b) Graphical symbol



(c) Sum-of-products implementation

Data-flow VHDL Example: Full Adder



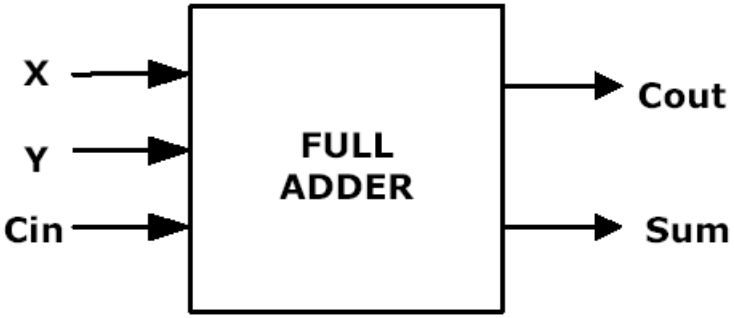
Data-flow VHDL Example: Full Adder

```
ENTITY fulladder IS
        PORT ( x      : IN     BIT;
               y      : IN     BIT;
               cin    : IN     BIT;
               s      : OUT    BIT;
               cout   : OUT    BIT) ;
END fulladder ;
```

```
ARCHITECTURE dataflow OF fulladder IS
BEGIN
        s     <=  x XOR y XOR cin ;
        cout  <= (x AND y) OR (cin AND x) OR (cin AND y) ;
END dataflow ;
```

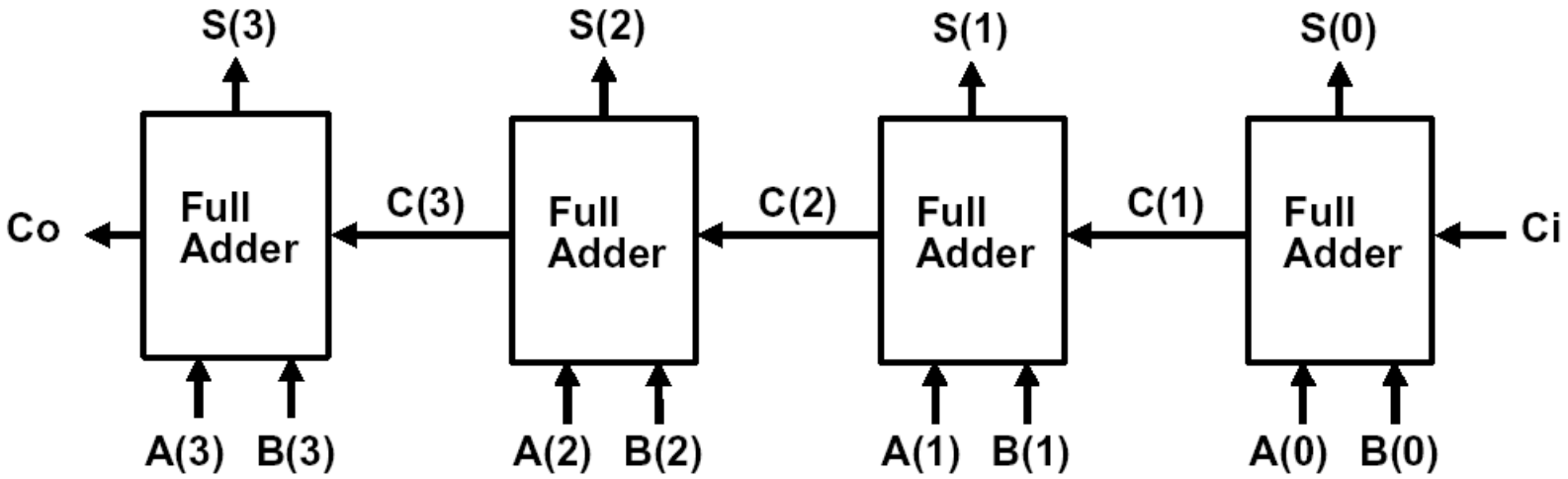
Entity-Architecture Pair

Full Adder Example




```
entity FullAdder is
  port (X, Y, Cin: in bit;     -- Inputs
    Cout, Sum: out bit);       -- Outputs
end FullAdder;

architecture Equations of FullAdder is
begin                          -- Concurrent Assignments
  Sum  <= X xor Y xor Cin after 10 ns;
  Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
end Equations;
```

4-bit Adder




```
entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); Ci: in bit;      -- Inputs
        S: out bit_vector(3 downto 0); Co: out bit);      -- Outputs
end Adder4;
```

4-bit Adder (cont'd)

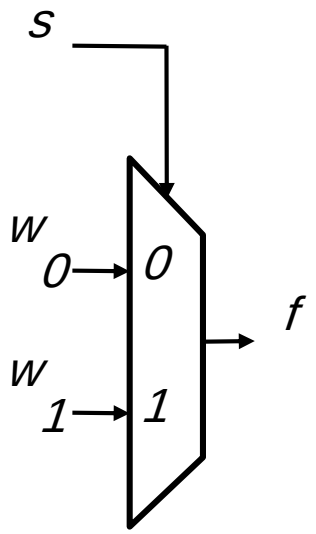
```
entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); Ci: in bit;      -- Inputs
        S: out bit_vector(3 downto 0); Co: out bit);      -- Outputs
end Adder4;

architecture Structure of Adder4 is
component FullAdder
    port (X, Y, Cin: in bit;            -- Inputs
          Cout, Sum: out bit);          -- Outputs
end component;
signal C: bit_vector(3 downto 1);
begin     --instantiate four copies of the FullAdder
    FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
    FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
    FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
    FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
end Structure;
```

Conditional concurrent signal assignment

When - Else

```
target_signal <= value1 when condition1 else
                 value2 when condition2 else
                   . . .
                 valueN-1 when conditionN-1 else
                 valueN;
```

2-to-1 Multiplexer



(a) Graphical symbol

s	f
0	w_0
1	w_1

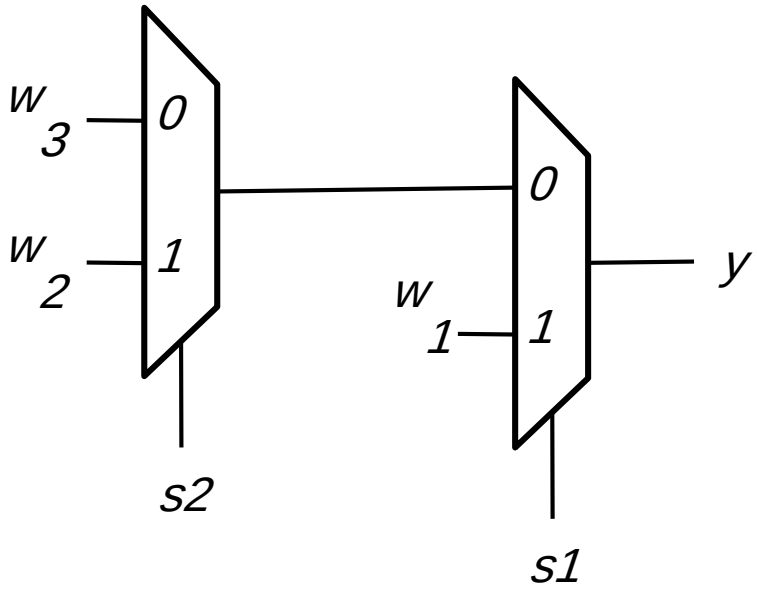
(b) Truth table

VHDL: `f <= w1 WHEN s = '1' ELSE w0 ;`

or

`f <= w0 WHEN s = '0' ELSE w1 ;`

Cascade of two multiplexers



VHDL:

```
y <= w1  WHEN s1 =  '1' ELSE
     w2  WHEN s2 =  '1' ELSE
     w3 ;
```

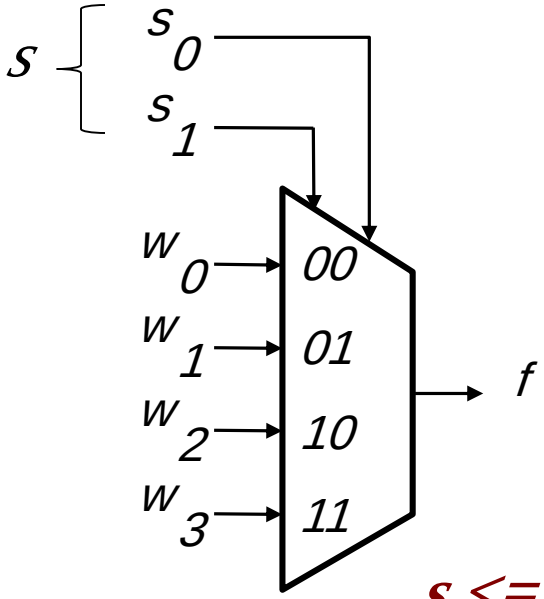
Selected concurrent signal assignment

With –Select-When

```
with choice_expression select
   target_signal <= expression1 when choices_1,
                    expression2 when choices_2,
                        . . .
                    expressionN when choices_N;
```

4-to-1 Multiplexer

(a) Graphic symbol



(b) Truth table

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

```
s <= s1 &  s0;
WITH s SELECT
        f <= w3WHEN “11",
             w2 WHEN "10",
             w1 WHEN "01",
             w0 WHEN OTHERS ;
```

Process

Process

- Contain sequential statements that define algorithms
- Executed when one of the signals in the sensitivity list has an event

```
proc1: process (a,b,c)
    begin
        x<=a and b and c;
end process proc1;
```

```
proc2: process
    begin
        x<=a and b and c;
        wait on a,b,c;
end process proc2;
```

Modeling Flip-Flops Using VHDL Processes

General form of process

```
process(sensitivity-list)
   begin
      sequential-statements
   end process;
```

- Whenever one of the signals in the sensitivity list changes, the sequential statements are executed in sequence one time

Sequential Style Syntax

```
[ process_label : ] PROCESS
[( sensitivity_list )]

  process_declarations

BEGIN

  process_statements

END PROCESS [ process_label ] ;
```

NO SIGNAL DECLARATIONS!

- Assignments are executed sequentially inside processes.

Sequential Statements

- {Signal, Variable} assignments
- Flow control
  - if <condition> then <statments>
    [elsif <condition> then <statments>]
    else <statements>
    end if;
  - for <range> loop <statments> end loop;
  - while <condition> loop <statments> end loop;
  - case <condition> is
    when <value> => <statements>;
    when <value> => <statements>;
    when others => <statements>;
- Wait on <signal> until <expression> for <time>;

Wait Statements

- ***Wait on*** an alternative to a sensitivity list
  - Note: a process cannot have both wait statement(s)
    and a sensitivity list
- Generic form of a process with wait statement(s)

```
process
begin
    sequential-statements
    wait statement
    sequential-statements
    wait-statement
    ...
end process;
```

How wait statements work?

- Execute seq. statement until a wait statement is encountered.
- Wait until the specified condition is satisfied.
- Then execute the next set of sequential statements until the next wait statement is encountered.
- ...
- When the end of the process is reached start over again at the beginning.

Forms of Wait Statements

```
wait on sensitivity-list;
wait for time-expression;
wait until boolean-expression;
```

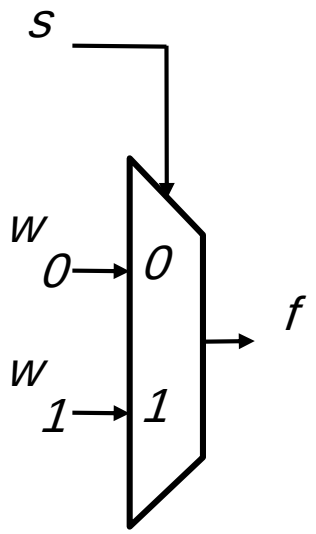
- Wait on
  - until one of the signals in the sensitivity list changes
- Wait for
  - waits until the time specified by the time expression has elapsed
  - What is this:
    `wait for 0 ns;`
- Wait until
  - the boolean expression is evaluated whenever one of the signals in the expression changes, and the process continues execution when the expression evaluates to TRUE

If statement: examples

```
if sel = '0' then
    result <= input_0;          -- executed if sel = 0
else
    result <= input_1;          -- executed if sel /= 0
end if;
```

```
if sel = '0' then
    result <= input_0; -- executed if sel = 0
elsif sel = 1 then
    result <= input_1; -- executed if sel = 1
else
    result <= input_2; -- executed if sel /= 0, 1
end if;
```

2-to-1 Multiplexer



(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table

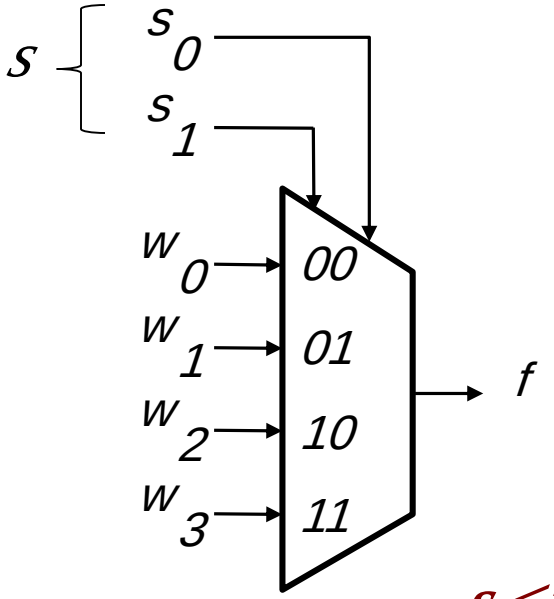
```
if sel = '1' then
        f <= w1;
else
        f <= w0;
end if;
```

Case statement: examples

```
type opcodes is (nop, add, sub);
case opcode is
    when add => acc <= acc + op;
    when sub => acc <= acc – op;
    when nop => null;
end case;
```

4-to-1 Multiplexer

(a) Graphic symbol



(b) Truth table

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

```
s <= s1 &  s0;
CASE s IS
        WHEN "11" => f <= w3;
        WHEN "10" => f <= w2;
        WHEN "01" => f <= w1;
        WHEN OTHERS => f <= w0;
```

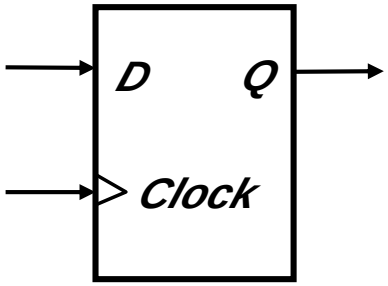
Case statement: rules

- all possible values of the selector expression must be covered,
- each possible value must be covered by one and only one choice,
- the choice values must be locally static, that is known at analysis stage, and
- if the **others** choice is used, it must be the last alternative and the only choice in the alternative.

Registers

Edge-Trigger Register

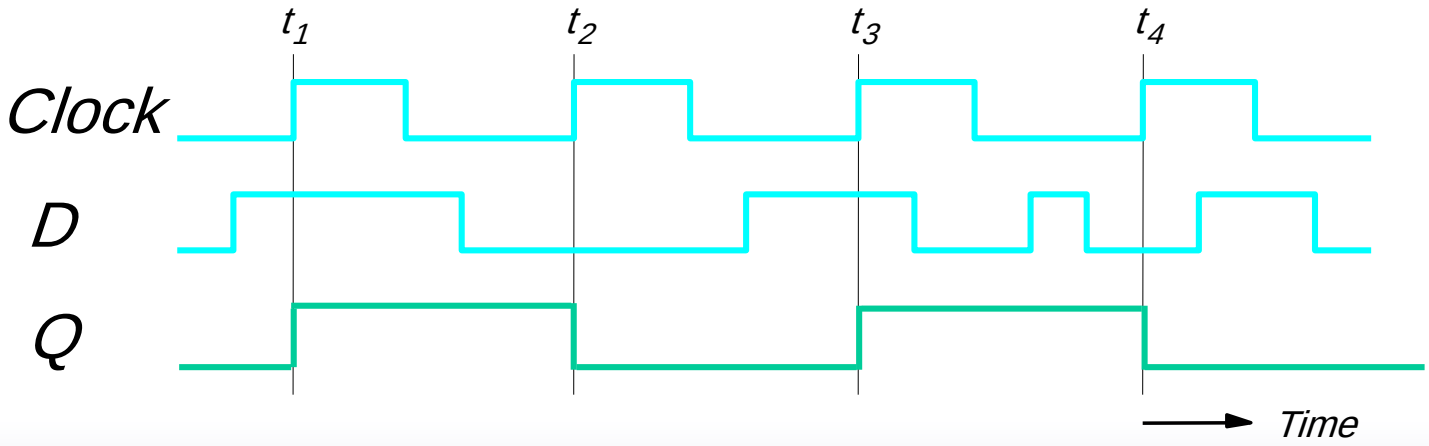
Graphical symbol



Truth table

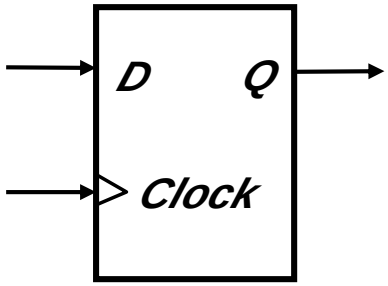
Clk	D	Q(t+1)
↑	0	0
↑	1	1
0	–	Q(t)
1	–	Q(t)

Timing diagram



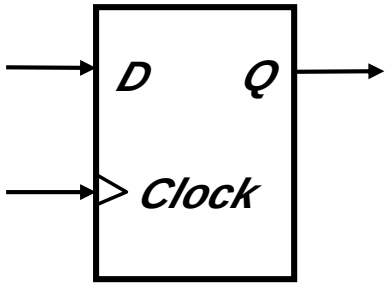
Edge-Trigger Register: Another Edge definition

```
ENTITY EdgeReg IS
      PORT (   D, Clock  : IN      BIT ;
               Q         : OUT     BIT) ;
END EdgeReg ;

ARCHITECTURE behavioral OF EdgeReg IS
BEGIN
      PROCESS ( Clock )
      BEGIN
          IF Clock'EVENT AND Clock = '1' THEN
               Q <= D ;
          END IF ;
      END PROCESS ;
END behavioral ;
```



Edge-Trigger Register: VHDL Code

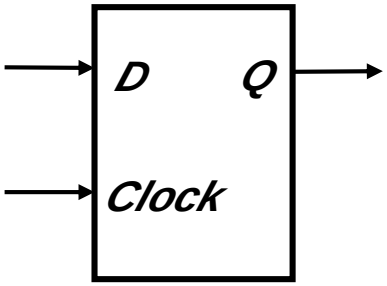
```
ENTITY EdgeReg IS
      PORT (   D, Clock  : IN      BIT ;
               Q         : OUT   BIT) ;
END EdgeReg ;

ARCHITECTURE behavioral2 OF EdgeReg IS
BEGIN
      PROCESS ( Clock )
      BEGIN
          IF rising_edge(Clock) THEN
               Q <= D ;
          END IF ;
      END PROCESS ;
END behavioral2;
```



- ***rising_edge()*** *is only in VHDL-93 and above*

D latch

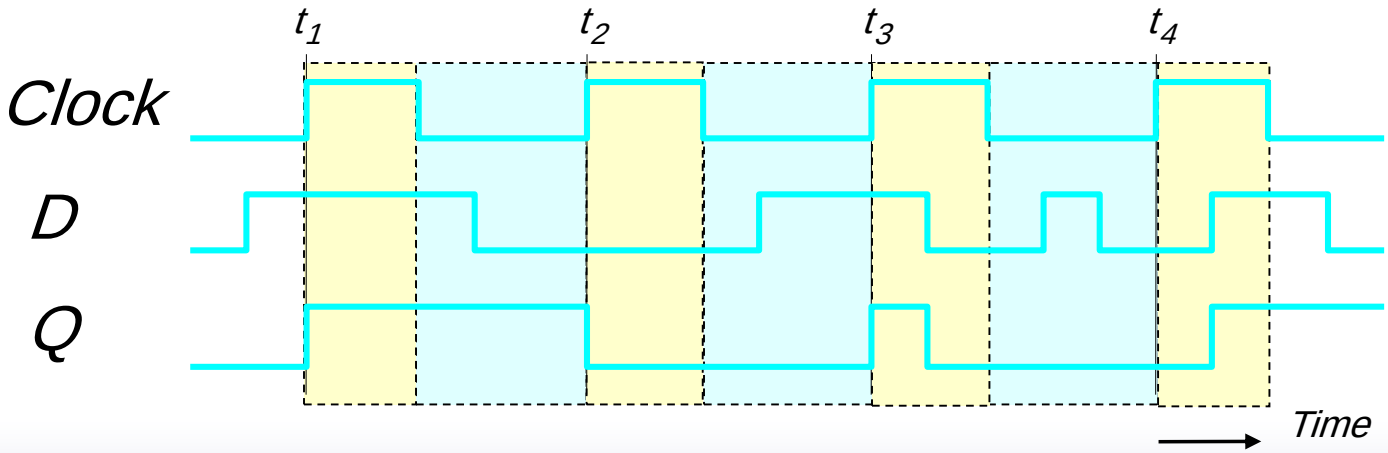
Graphical symbol



Truth table

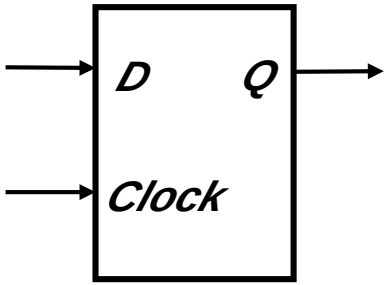
Clock	D	Q(t+1)
0	–	Q(t)
1	0	0
1	1	1

Timing diagram



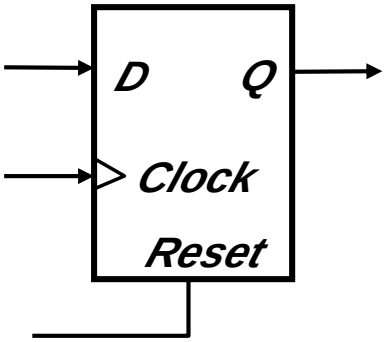
D latch

```
ENTITY latch IS
      PORT (   D, Clock  : IN        BIT;
               Q         : OUT       BIT) ;
END latch ;

ARCHITECTURE behavioral OF latch IS
BEGIN
      PROCESS ( D, Clock )
      BEGIN
            IF Clock = '1' THEN
                  Q <= D ;
            END IF ;
      END PROCESS ;
END behavioral;
```



Edge-Trigger Reg. with asynchronous reset

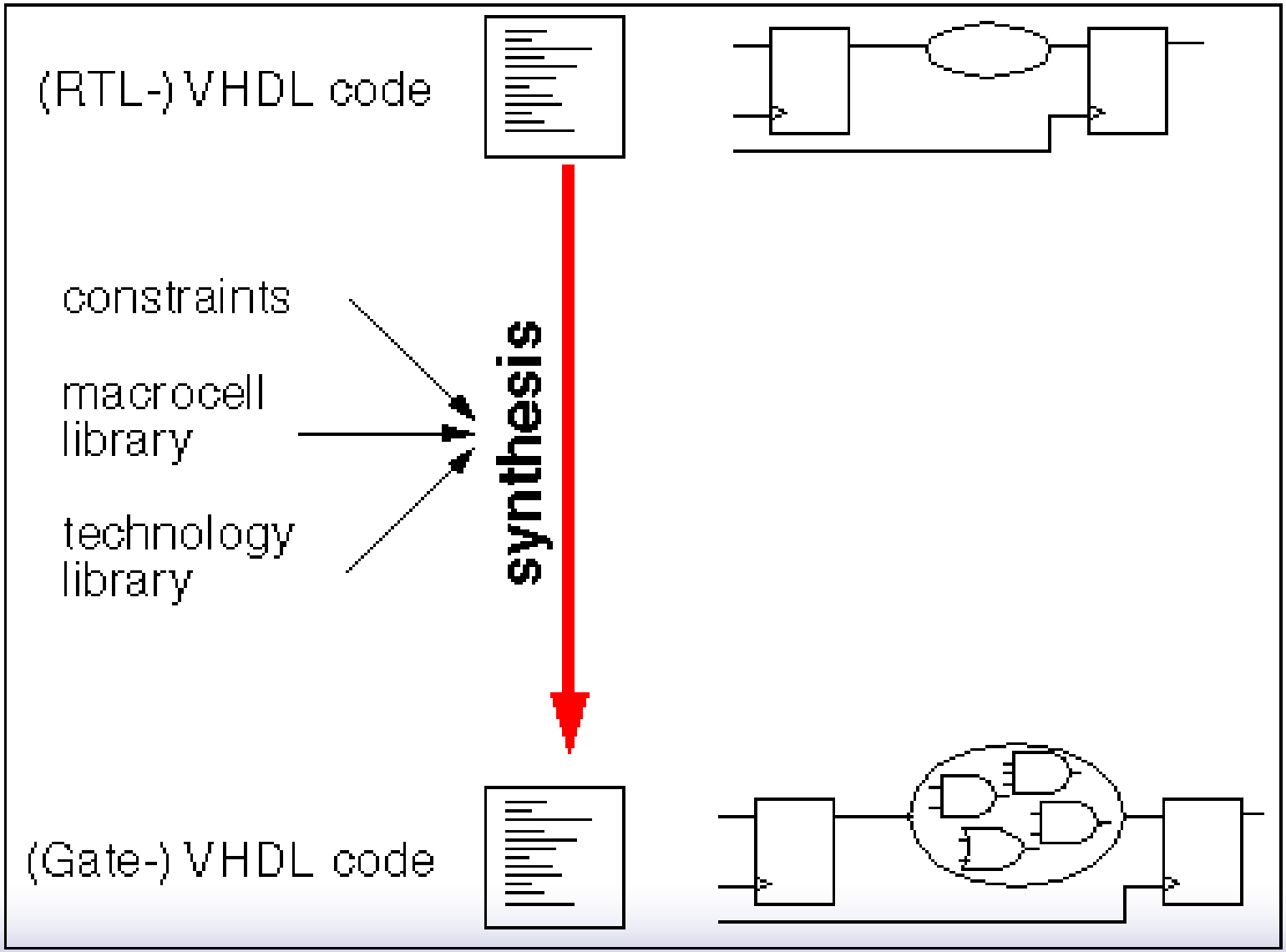
```
ENTITY Reg_ar IS
      PORT (   D, Reset, Clock      : IN        ; BIT
               Q                    : OUT       BIT) ;
END Reg_ar ;

ARCHITECTURE behavioral OF Reg_ar IS
BEGIN
      PROCESS ( Reset, Clock )
      BEGIN
            IF Reset = '1' THEN
                  Q <= '0' ;
            ELSIF rising_edge(Clock) THEN
                  Q <= D ;
            END IF ;
      END PROCESS ;
END behavioral ;
```



Synthesis

Synthesis

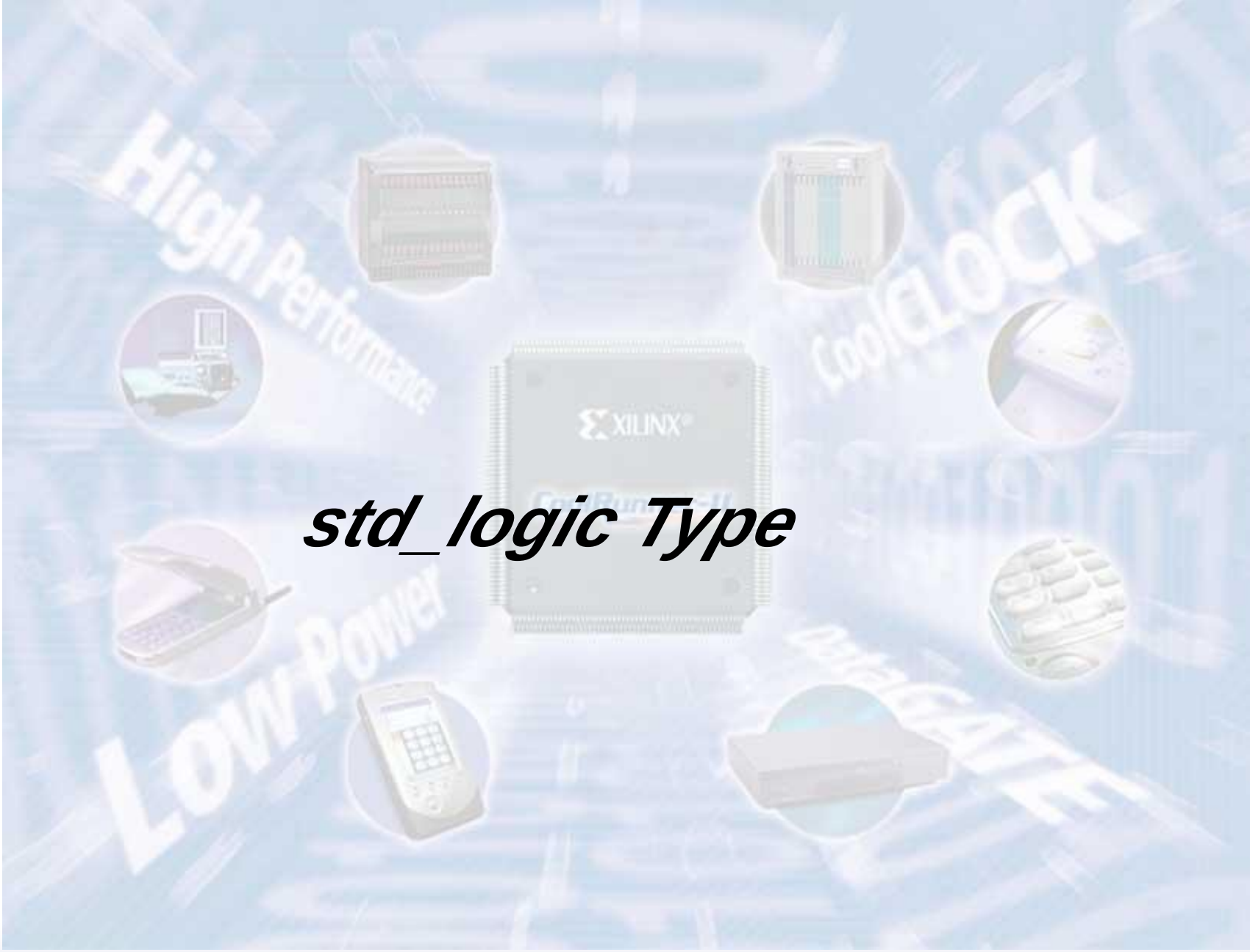
- Synthesis converts a high level VHDL description code into the gate level netlist (contains gates and their interconnections), it is usually with the help of synthesis tools.
- For VHDL,
  - only a subset of the language is synthesizable,
  - and different tools support different subsets.
  - RTL style code is encouraged because it is synthesizable.
- In RTL, it is possible to split the code into two blocks that contain:
  - either purely combinational logic
  - or sequential block (e.g. process) for register, FSM, ‘’’


(RTL-) VHDL code
constraints
macrocell library
technology library
synthesis
(Gate-) VHDL code

Synthesis Inpus and Outputs

- Synthesis Inputs:
  - VHDL Code Design
  - Constraints
  - Technology Library
- Synthesis Output:
  - Gates and their interconnections

- For simulation, we can use the unsynthesizable VHDL or Verilog code in the test bench to generate the stimulus.



std_logic Type

Example VHDL Code

- 3 sections to a piece of VHDL code
- File extension for a VHDL file is .vhd
- Name of the file should be the same as the entity name (nand_gate.vhd) [OpenCores Coding Guidelines]

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY nand_gate IS
     PORT(
          a    : IN STD_LOGIC;
          b    : IN STD_LOGIC;
          z    : OUT STD_LOGIC);
END nand_gate;

ARCHITECTURE model OF nand_gate IS
BEGIN
     z <= a NAND b;
END model;
```

LIBRARY DECLARATION

ENTITY DECLARATION

ARCHITECTURE BODY

IEEE 1164 Standard Logic

- ***std_logic*** type: a 9-valued logic system
  - 'U' – Uninitialized
  - 'X' – Forcing Unknown
  - ***'0' – Forcing 0***
  - ***'1' – Forcing 1***
  - ***'Z' – High impedance***
  - 'W' – Weak unknown
  - 'L' – Weak 0
  - 'H' – Weak 1
  - '-' – Don't care

If forcing and weak signal are tied together, the forcing signal dominates.

Useful in modeling the internal operation of certain types of ICs.

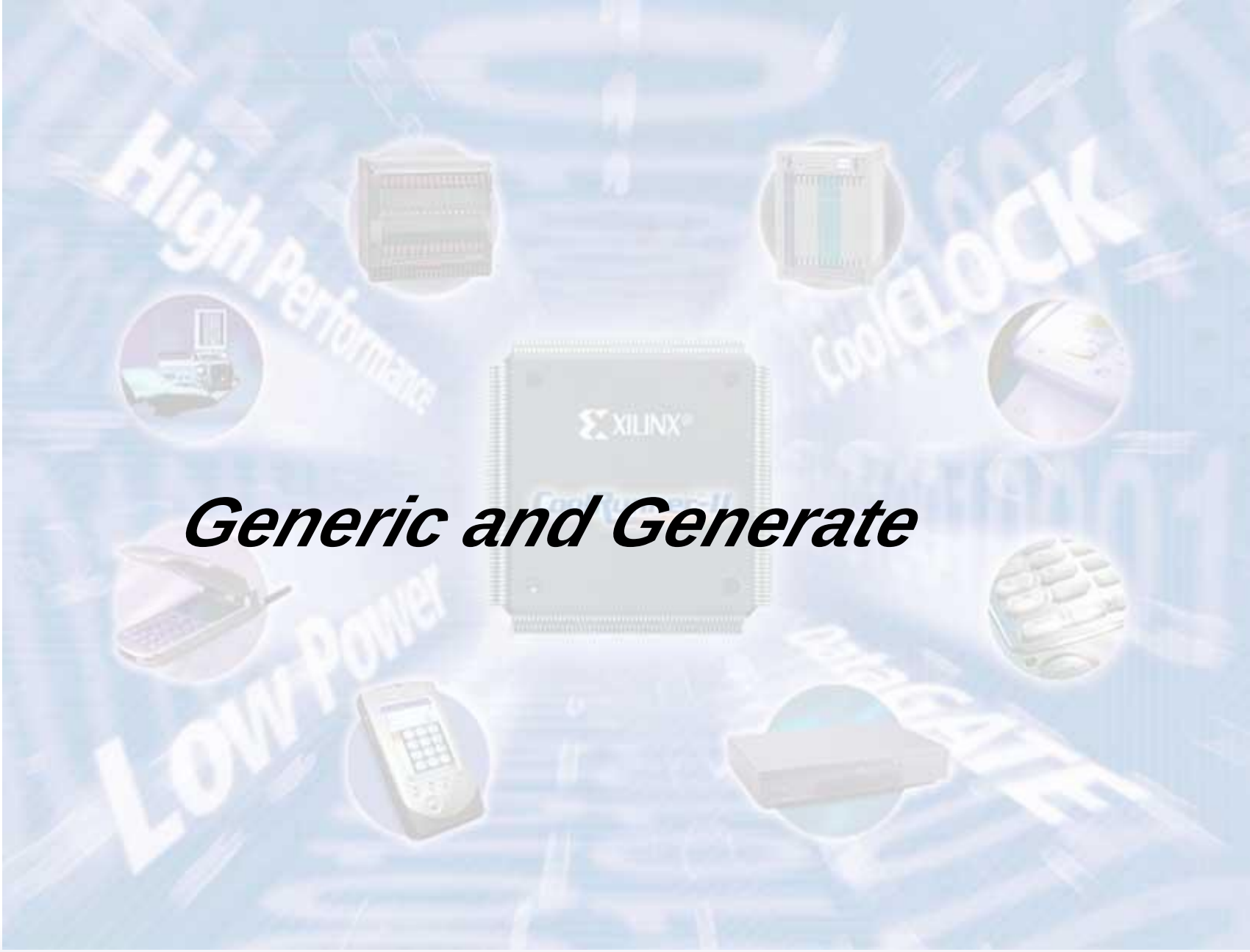
In this course we use a subset of the IEEE values: X10Z

BIT versus STD_LOGIC

- BIT type can only have a value of '0' or '1'
- STD_LOGIC can have nine values
  - '0', '1', 'Z', 'U', 'X', 'L', 'H', 'W', '-'

    Useful mainly for simulation
  - '0', '1', and 'Z' are synthesizable

    (***your codes should contain only these three values***)



Generic and Generate

Generic Statement

- Generics allow the component to be customized upon instantiation.
- Generics pass information from the entity to the architecture.
- Common uses of generics
  - Customize timing
  - Alter range of subtypes
  - Change size of arrays

```
ENTITY adder IS
GENERIC(n: natural :=2);
PORT(
      A: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
      B: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
      C: OUT STD_LOGIC;
      SUM: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0)
);
END adder;
```

Generics

- Generics allow the component to be customized upon instantiation.
- Generics pass information from the entity to the architecture.
- Common uses of generics
  - Customize timing
  - Alter range of subtypes
  - Change size of arrays

```
ENTITY adder IS
GENERIC(n: natural :=2);
PORT(
      A: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
      B: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
      C: OUT STD_LOGIC;
      SUM: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0)
);
END adder;
```

A word on generics

- Generics are typically ***integer*** values
  - In this class, the entity inputs and outputs should be std_logic or std_logic_vector
  - But the generics can be ***integer***
- Generics are given a default value
  - ***GENERIC (N : INTEGER := 16)*** ;
  - This value can be overwritten when entity is instantiated as a component
- Generics are very useful when instantiating an often-used component
  - Need a 32-bit register in one place, and 16-bit register in another
  - Can use the same generic code, just configure them differently

Use of OTHERS

OTHERS stand for any index value that has not been previously mentioned.

Q <= "00000001" can be written as Q <= (0 => '1', OTHERS => '0')

Q <= "10000001" can be written as Q <= (7 => '1', 0 => '1', OTHERS => '0')
or Q <= (7 | 0 => '1', OTHERS => '0')

Q <= "00011110" can be written as Q <= (4 downto 1=> '1', OTHERS => '0')

Component Instantiation in VHDL-87

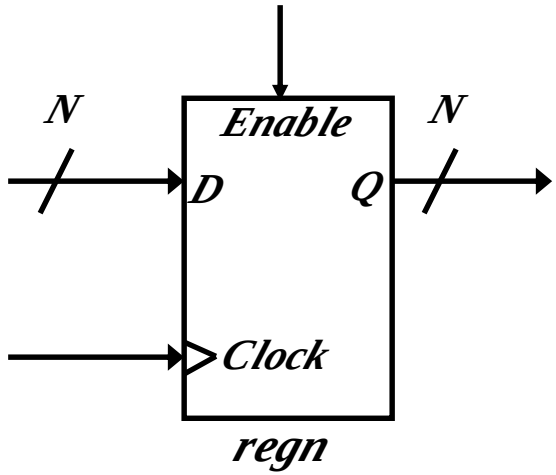
```
U1: regn        GENERIC MAP (N => 4)
                PORT MAP (D => z ,
                          Reset => reset ,
                          Clock => clk,
                          Q => t );
```

Component Instantiation in VHDL-93

```
U1:  ENTITY work.regn(behavioral)
        GENERIC MAP (N => 4)
        PORT MAP (D => z ,
                  Reset => reset ,
                  Clock => clk,
                  Q => t );
```

N-bit register with enable

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regne IS
        GENERIC ( N : INTEGER := 8 ) ;
        PORT (    D                 : IN      STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
                  Enable, Clock     : IN      STD_LOGIC ;
                  Q                 : OUT     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END regne ;

ARCHITECTURE behavioral OF regne IS
BEGIN
        PROCESS (Clock)
        BEGIN
                IF rising_edge(Clock) THEN
                        IF Enable = '1' THEN
                                Q <= D ;
                        END IF ;
                END IF;
        END PROCESS ;
END behavioral ;
```



Technology Modeling

- One use of generics is to alter the timing of a certain component.
- It is possible to indicate a generic timing delay and then specify the exact delay at instantiation.

```
COMPONENT inv IS
    PORT ( in1 : IN BIT;
           out1 : OUT BIT);
    GENERIC (tplh, tphl : TIME);
END COMPONENT;
```

- *The example above declares the interface to a component named inv.*
- *The propagation time for high-to-low and low-to-high transitions can be specified later.*

Structural Statements

- The GENERIC MAP is similar to the PORT MAP in that it maps specific values to generics declared in the component.

```
PACKAGE my_stuff IS
    COMPONENT and_gate
      GENERIC ( tplh, tphl : time);
      PORT ( in1, in2 : IN BIT; out1 : OUT BIT);
    END COMPONENT;
END my_stuff;

USE Work.my_stuff.ALL;
ARCHITECTURE test OF test_entity
      SIGNAL S1, S2, S3 : BIT;
BEGIN
    Gate1 : my_stuff.and_gate
      GENERIC MAP (2 ns, 3 ns)
      PORT MAP (S1, S2, S3);
END test;
```

Generate Statement

- Structural for-loops: The GENERATE statement
  - Some structures in digital hardware are repetitive in nature. (RAM, ROM, registers, adders, multipliers, …)
  - VHDL provides the GENERATE statement to automatically create regular hardware.
  - Any VHDL concurrent statement may be included in a GENERATE statement, including another GENERATE statement.

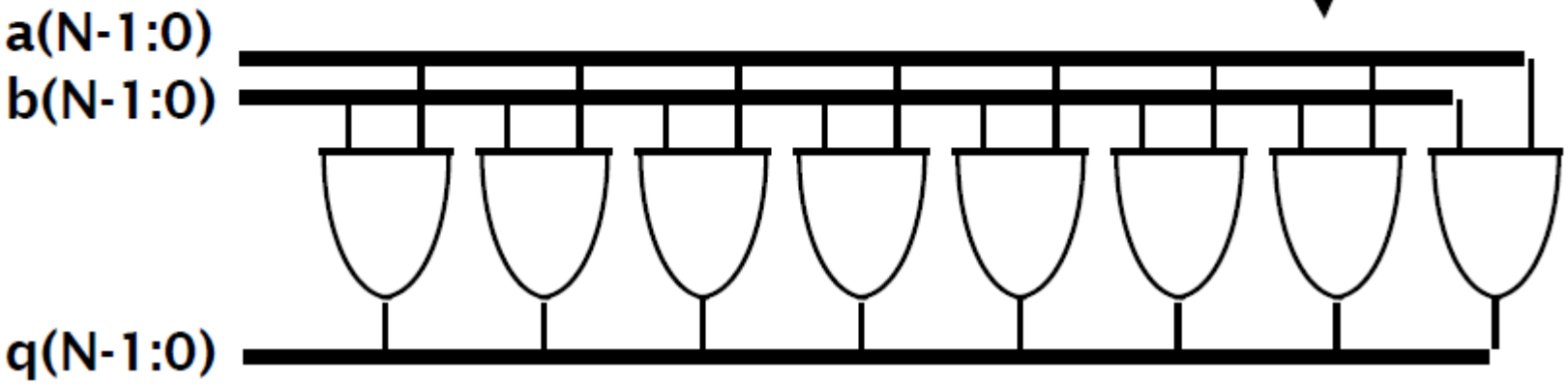
Generate Statement Syntax

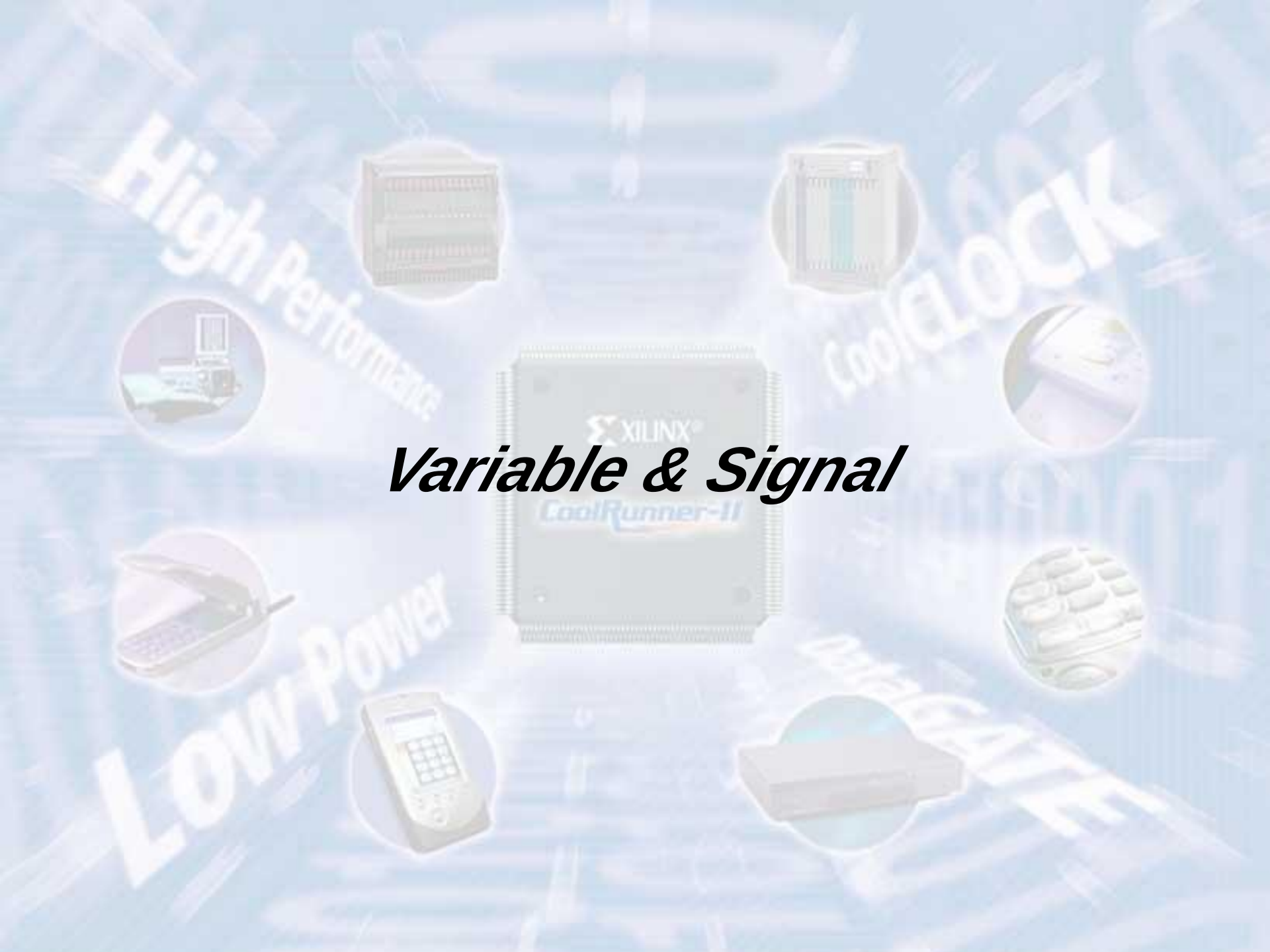
- All objects created are similar.
- The GENERATE parameter must be discrete and is undefined outside the GENERATE statement.

```
name : FOR N IN 1 TO 8 GENERATE
    concurrent-statements
END GENERATE name;
```

Example: Array of AND-gates

```
USE work.my_gates.all;
ARCHITECTURE structural OF and_bit_vector IS
BEGIN
    G1 : FOR i IN N-1 DOWNTO 0 GENERATE
      and_array : and_gate
        GENERIC MAP (2 ns, 3 ns)
        PORT MAP (i1=>a(i),i2=>b(i),q=>q(i));
    END GENERATE G1;
END structural;
```





Variable & Signal

Variables

- What are they for:
  Local storage in processes, procedures, and functions
- Declaring variables

```
variable list_of_variable_names : type_name
[ := initial value ];
```

- Variables must be declared within the process in which they are used and are local to the process
  - Note: exception to this is SHARED variables

Signals

- Signals must be declared outside a process
- Declaration form

```
signal list_of_signal_names : type_name
[ := initial value ];
```

- Declared in an architecture can be used anywhere within that architecture

Constants

- Declaration form

```
constant constant_name : type_name := constant_value;

constant delay1 : time := 5 ns;
```

- Constants declared at the start of an architecture can be used anywhere within that architecture
- Constants declared within a process are local to that process

Variables vs. Signals

- Variable assignment statement

```
variable_name := expression;
```

  - expression is evaluated and the variable is instantaneously updated (no delay, not even delta delay)

- Signal assignment statement

```
signal_name <= expression [after delay];
```

  - expression is evaluated and the signal is scheduled to change after delay; if no delay is specified the signal is scheduled to be updated after a delta delay

Variables vs. Signals (cont'd)

Process Using Variables

```
entity dummy is
end dummy;

architecture var of dummy is
      signal trigger, sum: integer:=0;
begin
      process
      variable var1: integer:=1;
      variable var2: integer:=2;
      variable var3: integer:=3;
      begin
            wait on trigger;
            var1 := var2 + var3;
            var2 := var1;
            var3 := var2;
            sum <= var1 + var2 + var3;
      end process;
end var;
```

Sum = ?

Process Using Signals

```
entity dummy is
end dummy;

architecture sig of dummy is
      signal trigger, sum: integer:=0;
      signal sig1: integer:=1;
      signal sig2: integer:=2;
      signal sig3: integer:=3;
begin
      process
      begin
         wait on trigger;
         sig1 <= sig2 + sig3;
         sig2 <= sig1;
         sig3 <= sig2;
         sum <= sig1 + sig2 + sig3;
      end process;
end sig;
```

Sum = ?

Predefined VHDL Types

- Variables, signals, and constants can have any one of the predefined VHDL types or they can have a user-defined type
- Predefined Types
  - bit – {‘0’, ‘1’}
  - boolean – {TRUE, FALSE}
  - integer – [-2^{31} - 1.. 2^{31} – 1}
  - real – floating point number in range –1.0E38 to +1.0E38
  - character – legal VHDL characters including lower-uppercase letters, digits, special characters, ...
  - time – an integer with units fs, ps, ns, us, ms, sec, min, or hr

User Defined Type

- Common user-defined type is *enumerated*

```
type state_type is (S0, S1, S2, S3, S4, S5);
signal state : state_type := S1;
```

  - If no initialization, the default initialization is the leftmost element in the enumeration list (S0 in this example)

- VHDL is strongly typed language => signals and variables of different types cannot be mixed in the same assignment statement, and no automatic type conversion is performed

Arrays

- Example

```
type SHORT_WORD is array (15 downto 0) of bit;

signal DATA_WORD : SHORT_WORD;
variable ALT_WORD : SHORT_WORD := "0101010101010101";
constant ONE_WORD : SHORT_WORD := (others => '1');
```

  - ALT_WORD(0) – rightmost bit
  - ALT_WORD(5 downto 0) – low order 6 bits

- General form

```
type arrayTypeName is array index_range of element_type;
signal arrayName : arrayTypeName [:=InitialValues];
```

Arrays (cont'd)

- Multidimensional arrays

```
type matrix4x3 is array (1 to 4, 1 to 3) of integer;
variable matrixA: matrix4x3 :=
((1,2,3), (4,5,6), (7,8,9), (10,11,12));
```

  - matrixA(3, 2) = ?

- Unconstrained array type

```
type intvec is array (natural range<>) of integer;

type matrix is array (natural range<>,natural range<>)
of integer;
```

  - range must be specified when the array object is declared

```
signal intvec5 : intvec(1 to 5) := (3,2,6,8,1);
```

Predefined Unconstrained Array Types

- Bit_vector, string

```
type bit_vector is array (natural range <>) of bit;
type string is array (positive range <>) of character;
constant string1: string(1 to 29) := "This string is 29 characters."
```

```
constant A : bit_vector(0 to 5) := "10101";
-- ('1', '0', '1', '0', '1');
```

- Subtypes
  - include a subset of the values specified by the type

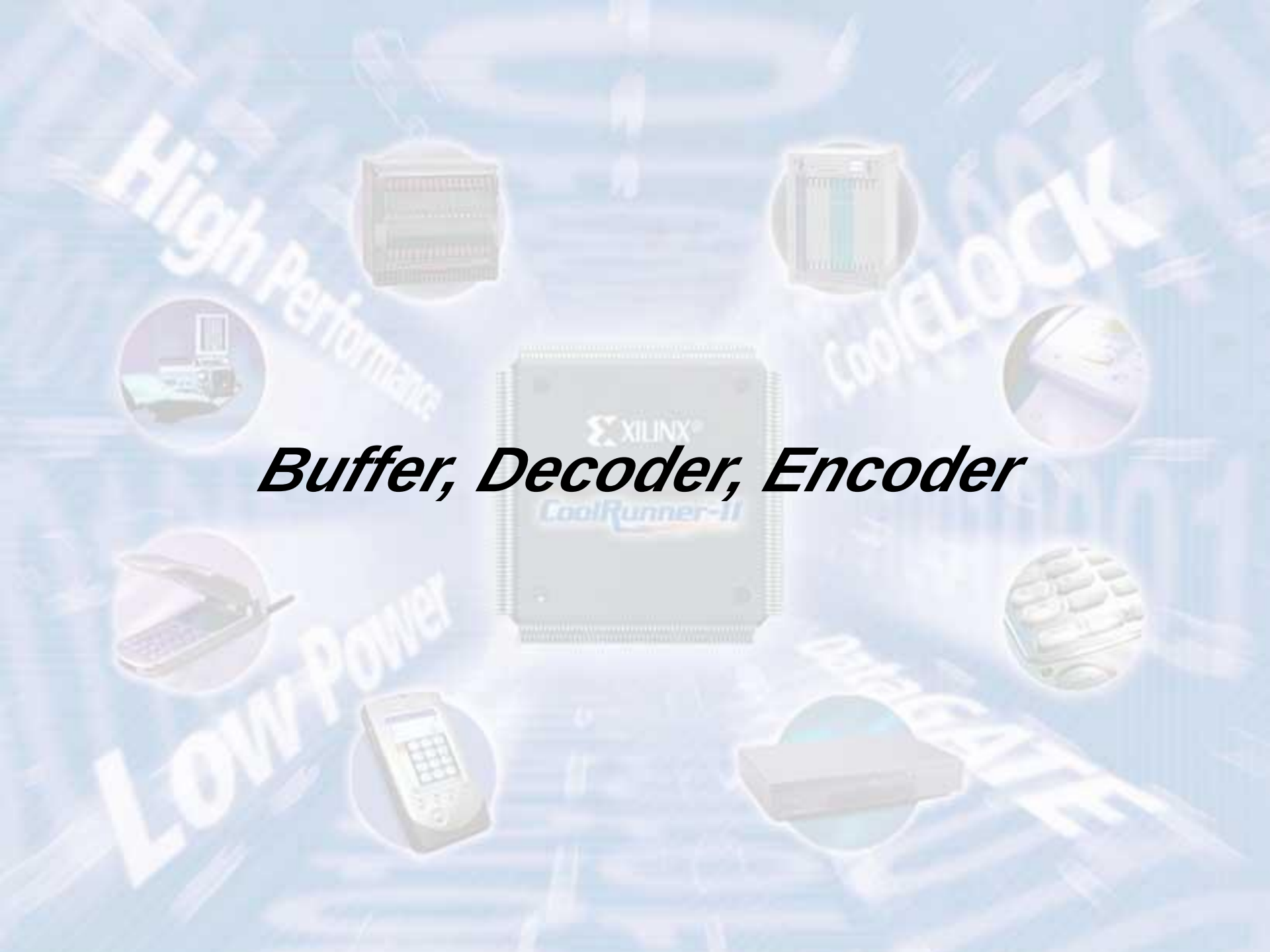
```
subtype SHORT_WORD is : bit_vector(15 to 0);
```

- POSITIVE, NATURAL –
  predefined subtypes of type integer

VHDL Operators

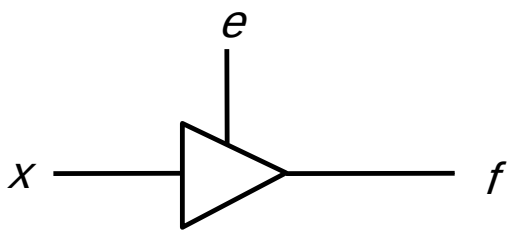
1. Binary logical operators: ***and or nand nor xor xnor***
2. Relational: = /= < <= > >=
3. Shift: ***sll srl sla sra rol ror***
4. Adding: + - & (concatenation)
5. Unary sign: + -
6. Multiplying: * / ***mod rem***
7. Miscellaneous: ***not abs*** **

- Class 7 has the highest precedence (applied first), followed by class 6, then class 5, etc



Buffer, Decoder, Encoder

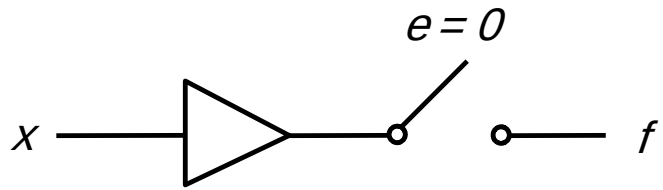
Tri-state Buffer

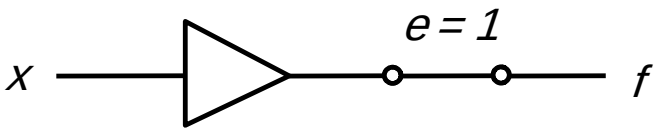


(a) A tri-state buffer

e	x	f
0	0	Z
0	1	Z
1	0	0
1	1	1

(c) Truth table



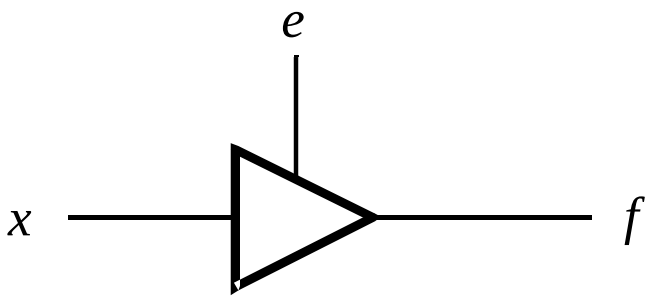


(b) Equivalent circuit

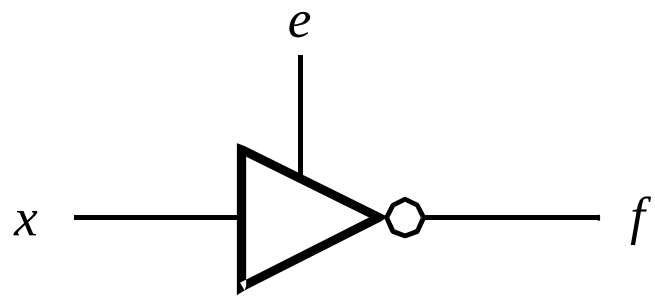
Tri-state Buffer entity

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY tri_state IS
   PORT ( e:    IN STD_LOGIC;
                x:  IN STD_LOGIC;
               f: OUT STD_LOGIC
              );
END tri_state;

ARCHITECTURE dataflow OF tri_state IS
BEGIN
    f <= x WHEN (e = '1') ELSE   'Z';
END dataflow;
```

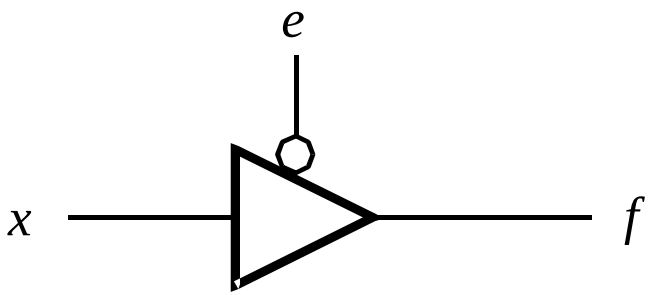
Four types of Tri-state Buffers



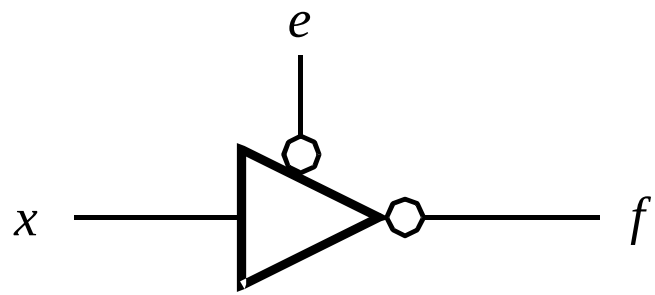

```
f <= x WHEN (e = '1') ELSE   'Z';
```




```
f <= not x WHEN (e = '1') ELSE   'Z';
```




```
f <= x WHEN (e = '0') ELSE   'Z';
```



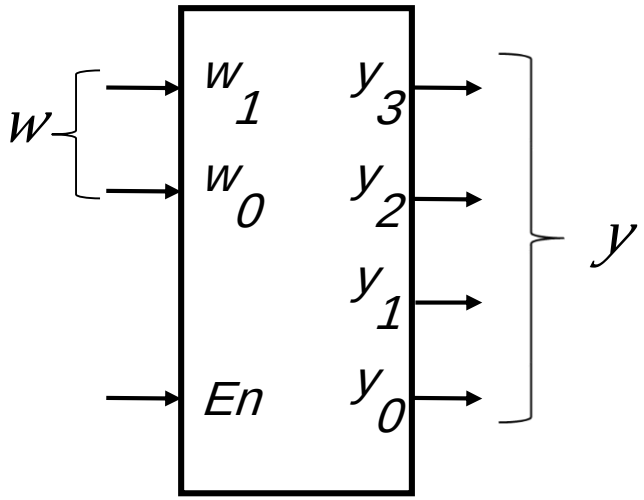

```
f <= not x WHEN (e = '0') ELSE   'Z';
```

2-to-4 Decoder

(a) Truth table

En	w_1	w_0	y_3	y_2	y_1	y_0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0
0	x	x	0	0	0	0

(b) Graphical symbol

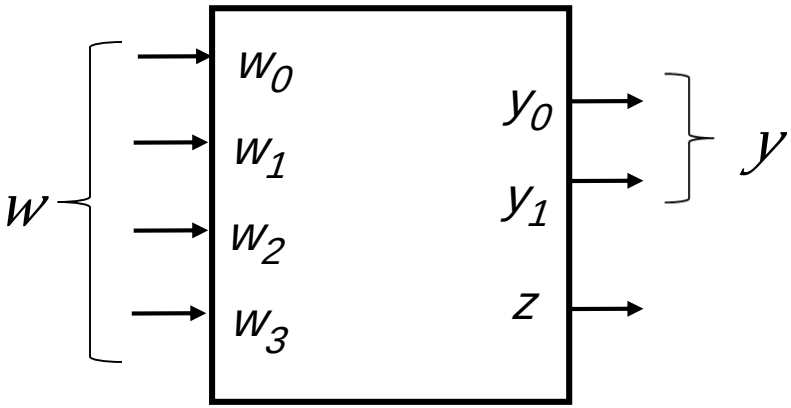



```
Enw <= En & w ;
WITH Enw SELECT
    y <= "0001" WHEN "100",
         "0010" WHEN "101",
         "0100" WHEN "110",
         "1000" WHEN "111",
         "0000" WHEN OTHERS ;
```

VHDL code for a 2-to-4 Decoder entity

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
      PORT (   w    : IN      STD_LOGIC_VECTOR(1 DOWNTO 0) ;
               En   : IN      STD_LOGIC ;
               y    : OUT     STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END dec2to4 ;

ARCHITECTURE dataflow OF dec2to4 IS
      SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
      Enw <= En & w ;
      WITH Enw SELECT
            y <= "0001" WHEN "100",
                 "0010" WHEN "101",
                 "0100" WHEN "110",
                 "1000" WHEN "111",
                 "0000" WHEN OTHERS ;
END dataflow ;
```

Priority Encoder



w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	-	-	0
0	0	0	1	0	0	1
0	0	1	-	0	1	1
0	1	-	-	1	0	1
1	-	-	-	1	1	1

```
y <= "11" WHEN w(3) = '1' ELSE
     "10" WHEN w(2) = '1' ELSE
     "01" WHEN w(1) = '1' ELSE
     "00";

z <= '0' WHEN w = "0000" ELSE '1';
```

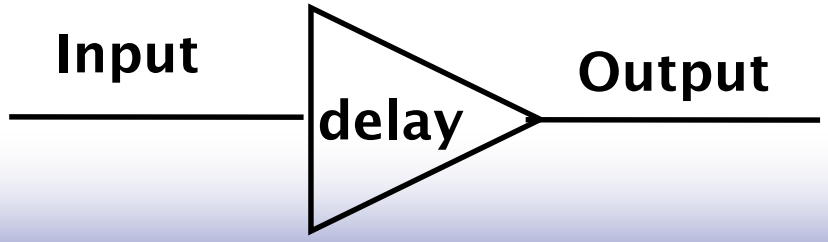
VHDL code for a Priority Encoder entity

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y   : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z   : OUT   STD_LOGIC ) ;
END priority ;

ARCHITECTURE dataflow OF priority IS
BEGIN
    y <=   "11" WHEN w(3) = '1' ELSE
           "10" WHEN w(2) = '1' ELSE
           "01" WHEN w(1) = '1' ELSE
           "00" ;
    z <= '0' WHEN w = "0000" ELSE '1';
END dataflow ;
```

Delay Types

Delay Types

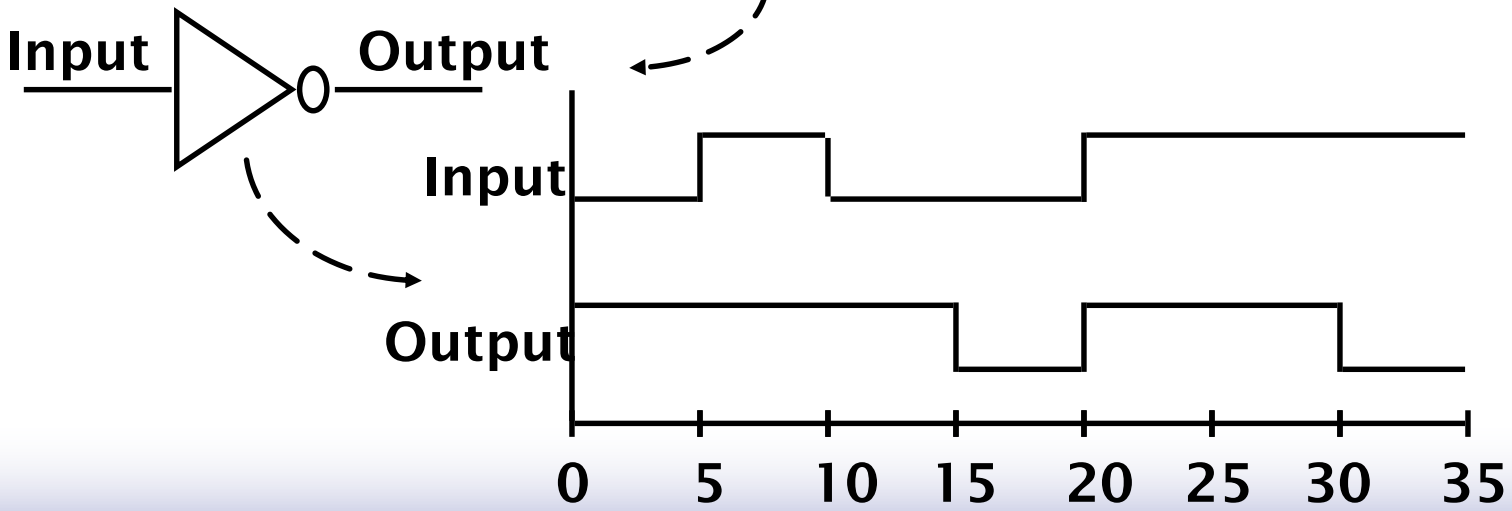
- All VHDL signal assignment statements prescribe an amount of time that must transpire before the signal assumes its new value
- This prescribed delay can be in one of three forms:
  - Transport -- prescribes propagation delay only
  - Inertial -- prescribes propagation delay and minimum input pulse width
  - Delta -- the default if no delay time is explicitly specified



Transport Delay

- Transport delay must be explicitly specified
  - I.e. keyword “TRANSPORT” must be used
- Signal will assume its new value after specified delay

```
-- TRANSPORT delay example
Output <= TRANSPORT NOT Input AFTER 10 ns;
```



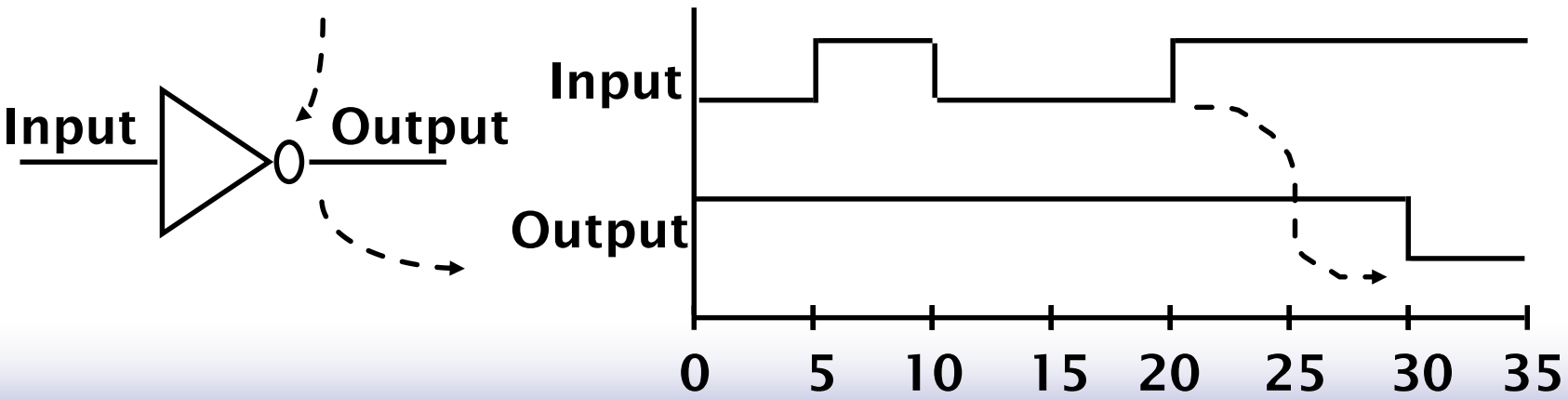
Inertial Delay

- Provides for specification propagation delay and input pulse width, i.e. 'inertia' of output:

```
target <= [REJECT time_expression] INERTIAL waveform;
```

- Inertial delay is default and REJECT is optional:

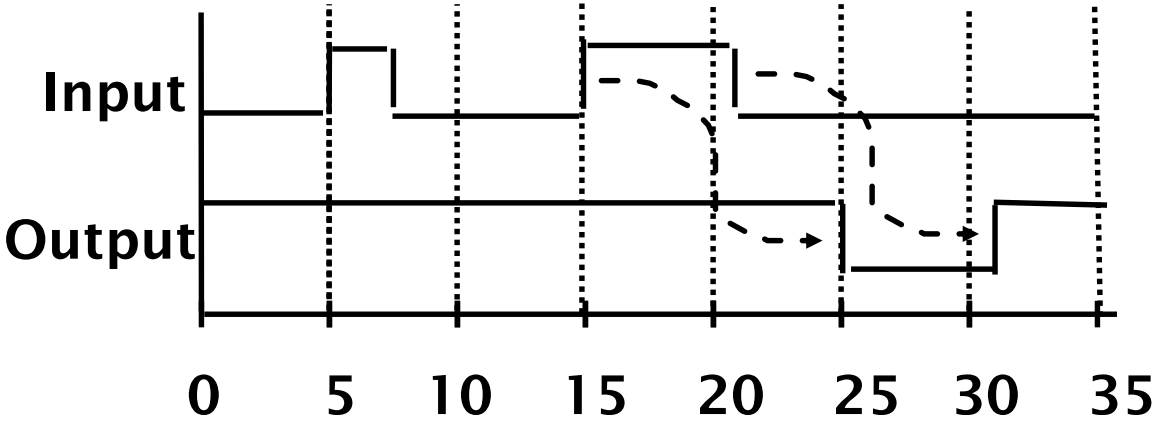
```
Output <= NOT Input AFTER 10 ns;
-- Propagation delay and minimum pulse width are 10ns
```



Inertial Delay (cont.)

- Example of gate with 'inertia' smaller than propagation delay
  - e.g. Inverter with propagation delay of 10ns which suppresses pulses shorter than 5ns

```
Output <=  REJECT 5ns INERTIAL NOT Input AFTER 10ns;
```



- Note: the REJECT feature is new to VHDL 1076-1993

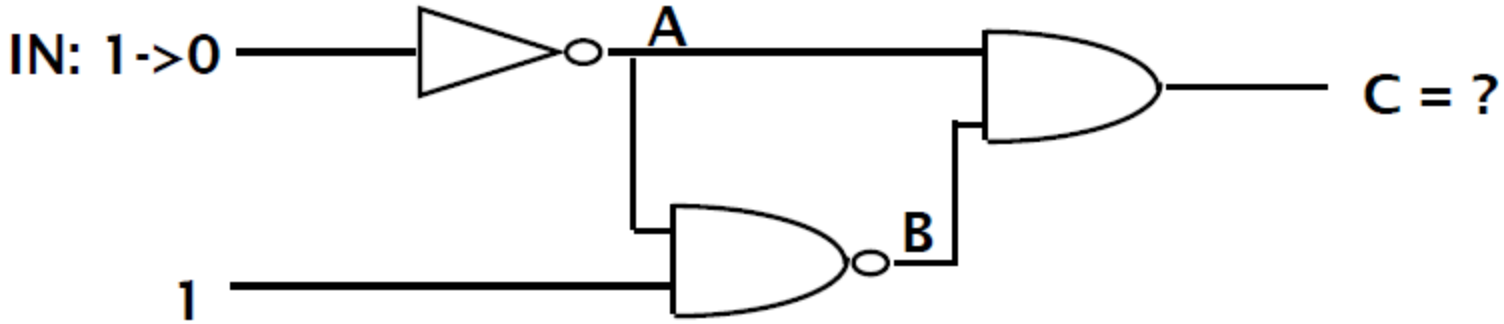
Delta Delay

- Default signal assignment propagation delay if no delay is explicitly prescribed
  - VHDL signal assignments do not take place immediately
  - Delta is an infinitesimal VHDL time unit so that all signal assignments can result in signals assuming their values at a future time
  - E.g.

```
Output <= NOT Input;
-- Output assumes new value in one delta cycle
```

- Supports a model of concurrent VHDL process execution
  - Order in which processes are executed by simulator does not affect simulation output

Example – Delta Delay

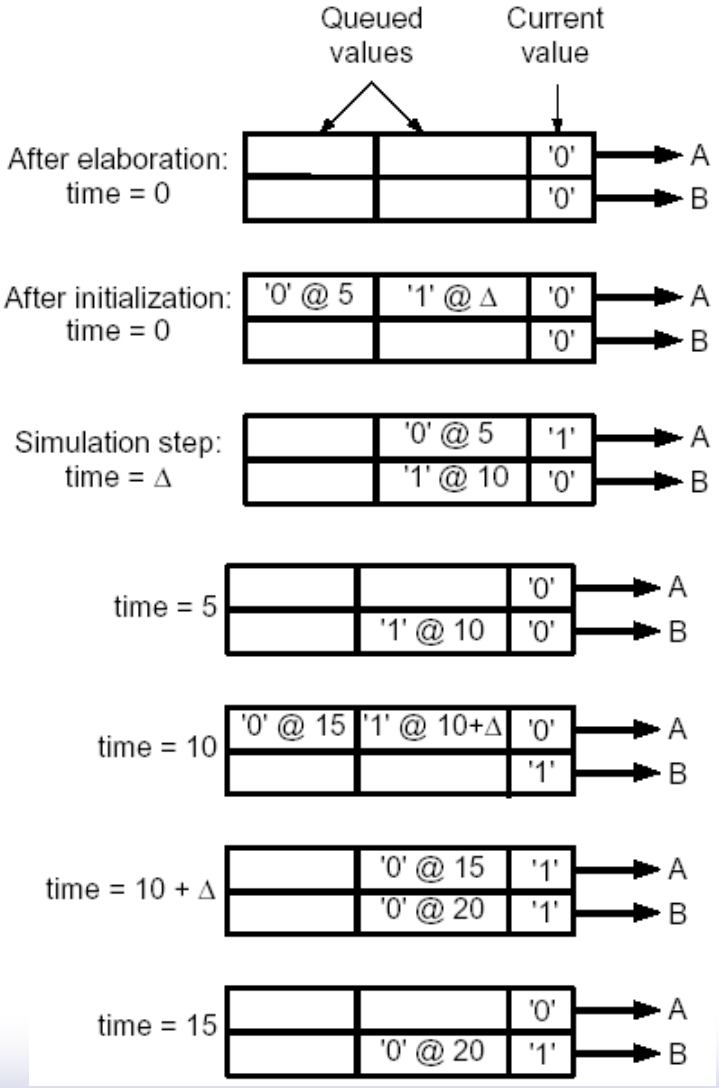


Using delta delay scheduling

Time	Delta	Event
0 ns	1	IN: 1->0 eval inverter
	2	A: 0->1 eval NAND, AND
	3	B: 1->0 C: 0->1 eval AND
	4	C: 1->0
1 ns		

Simulation Example

```
entity simulation_example is
end simulation_example;

architecture test1 of simulation_example is
   signal A,B: bit;
begin
   P1: process(B)
   begin
      A <= '1';
      A <= transport '0' after 5 ns;
   end process P1;

   P2: process(A)
   begin
      if A = '1' then B <= not B after 10 ns; end if;
   end process P2;
end test1;
```



Problem #1

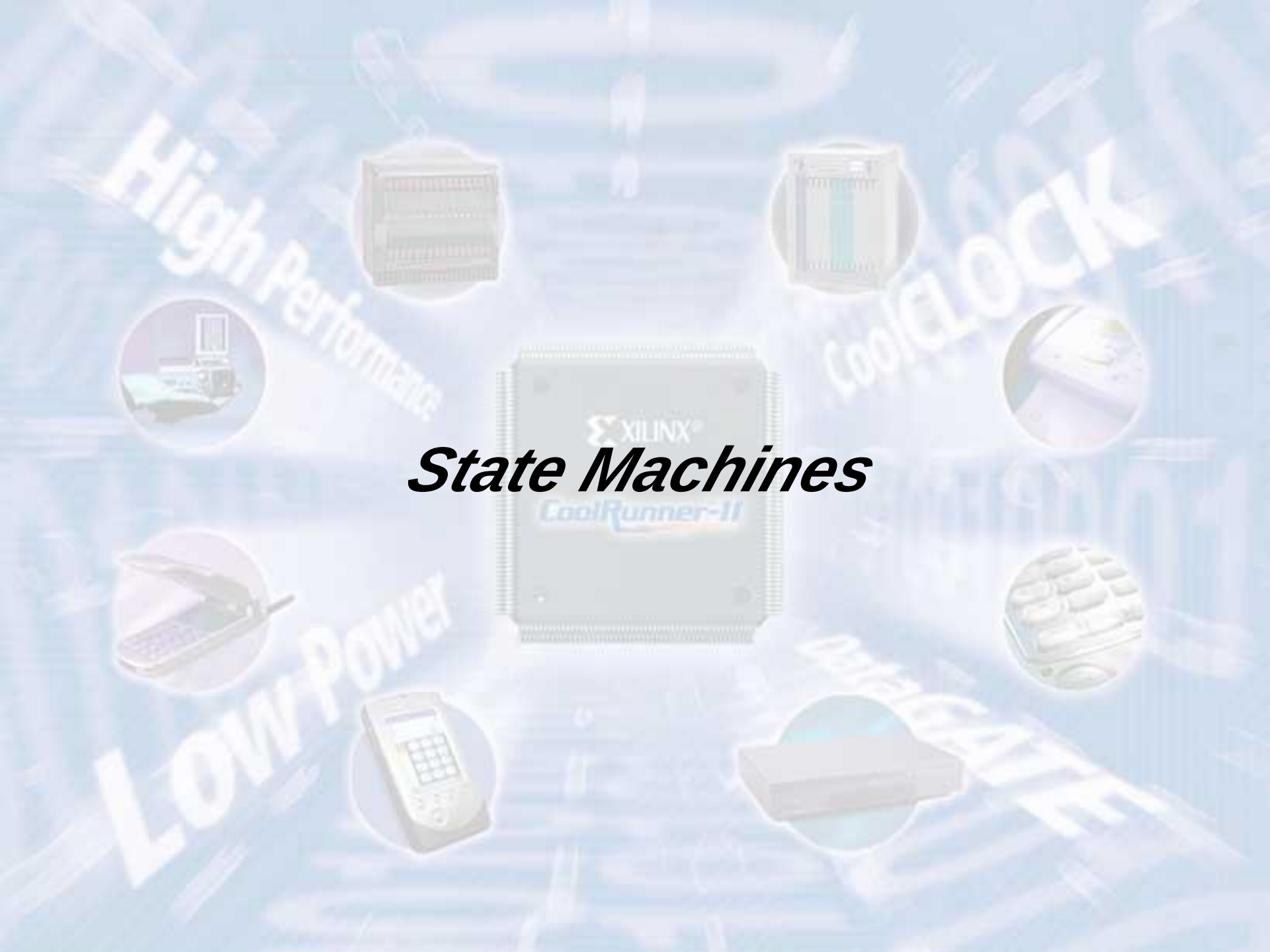
- Using the labels, list the order in which the following signal assignments are evaluated if in2 changes from a '0' to a '1'. Assume in1 has been a '1' and in2 has been a '0' for a long time, and then at time *t* in2 changes from a '0' to a '1'.

```
entity not_another_prob is
        port (in1, in2: in bit;
        a: out bit);
end not_another_prob;


architecture oh_behave of not_another_prob is
        signal b, c, d, e, f: bit;
begin
        L1:  d <= not(in1);
        L2:  c<= not(in2);
        L3:  f <= (d and in2) ;
        L4:  e <= (c and in1) ;
        L5:  a <= not b;
        L6:  b <= e or f;
end oh_behave;
```

Problem #2

- Under what conditions do the two assignments below result in the same behavior? Different behavior? Draw waveforms to support your answers.

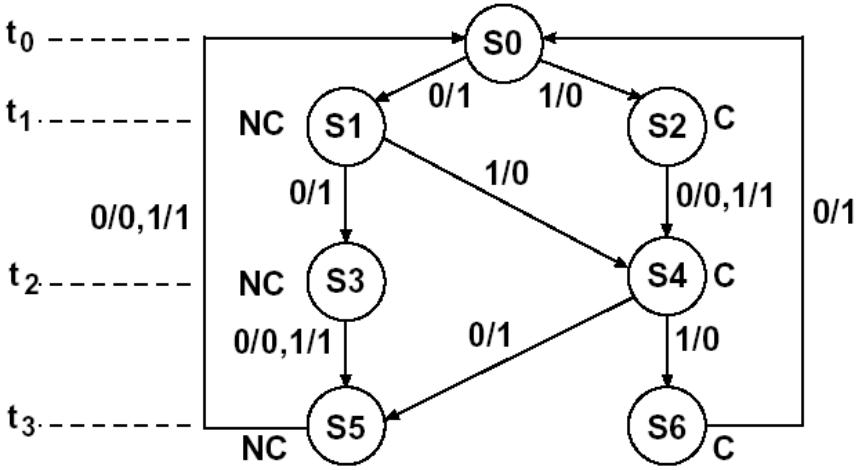
```
out <= reject 5 ns inertial (not a) after 20 ns;
out <= transport (not a) after 20 ns;
```



State Machines

Modeling a Sequential Machine

Mealy Machine for
8421 BCD to 8421 BCD + 3 bit serial converter



	NS		Z	
PS	X = 0	X = 1	X = 0	X = 1
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	–	1	–

How to model this in VHDL?

Behavioral VHDL Model

```
entity SM1_2 is
   port(X, CLK: in bit;  Z: out bit);
end SM1_2;

architecture Table of SM1_2 is
   signal State, Nextstate: integer := 0;
begin
   process(State,X)                    --Combinational Network
   begin
   case State is
      when 0 =>
         if X='0' then Z<='1'; Nextstate<=1; end if;
         if X='1' then Z<='0'; Nextstate<=2; end if;
      when 1 =>
         if X='0' then Z<='1'; Nextstate<=3; end if;
         if X='1' then Z<='0'; Nextstate<=4; end if;
      when 2 =>
         if X='0' then Z<='0'; Nextstate<=4; end if;
         if X='1' then Z<='1'; Nextstate<=4; end if;
      when 3 =>
         if X='0' then Z<='0'; Nextstate<=5; end if;
         if X='1' then Z<='1'; Nextstate<=5; end if;
      when 4 =>
         if X='0' then Z<='1'; Nextstate<=5; end if;
         if X='1' then Z<='0'; Nextstate<=6; end if;
       when 5 =>
          if X='0' then Z<='0'; Nextstate<=0; end if;
          if X='1' then Z<='1'; Nextstate<=0; end if;
       when 6 =>
          if X='0' then Z<='1'; Nextstate<=0; end if;
       when others => null;              -- should not occur
    end case;
 end process;

    process(CLK)                         -- State Register
       begin
          if CLK='1' then                -- rising edge of clock
             State <= Nextstate;
          end if;
       end process;
 end Table;
```

	NS		Z	
PS	X = 0	X = 1	X = 0	X = 1
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	–	1	–

Two processes:

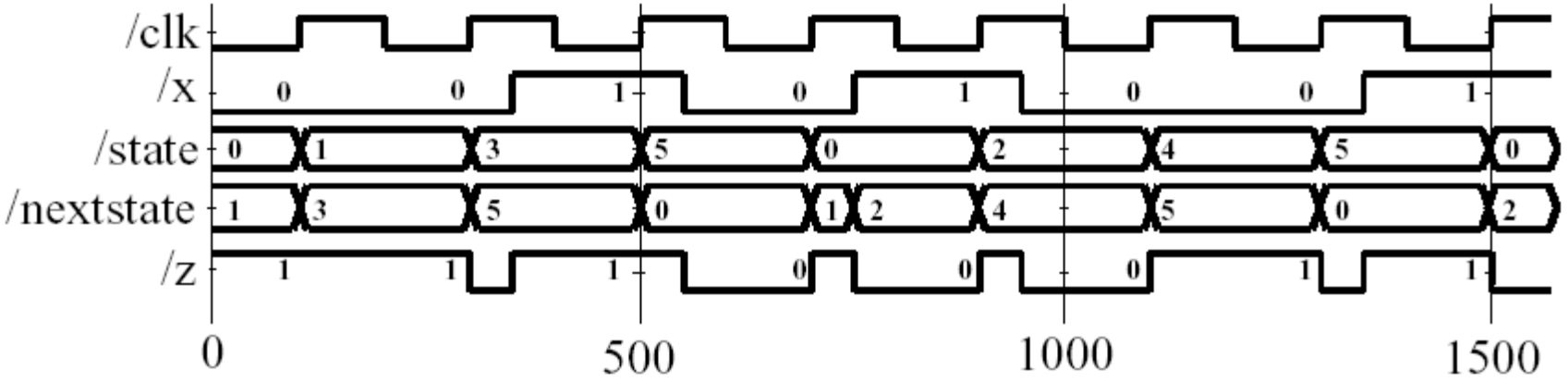
- the first represents the combinational network;
- the second represents the state register

Simulation of the VHDL Model

Simulation command file:

```
wave CLK X State NextState Z
force CLK 0 0, 1 100 -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600
```

Waveforms:



Sequential Machine Model Using State Table

```
entity SM1_2 is
   port (X, CLK: in bit;
      Z: out bit);
end SM1_2;

architecture Table of SM1_2 is
   type StateTable is array (integer range <>, bit range <>) of integer;
   type OutTable is array (integer range <>, bit range <>) of bit;
   signal State, NextState: integer := 0;
   constant ST: StateTable (0 to 6, '0' to '1') :=
      ((1,2), (3,4), (4,4), (5,5), (5,6), (0,0), (0,0));
   constant OT: OutTable (0 to 6, '0' to '1') :=
      (('1','0'), ('1','0'), ('0','1'), ('0','1'), ('1','0'), ('0','1'), ('1','0'));
begin                                  -- concurrent statements
   NextState <= ST(State,X);           -- read next state from state table
   Z <= OT(State, X);                  -- read output from output table
   process(CLK)
   begin
      if CLK = '1' then                -- rising edge of CLK
         State <= NextState;
      end if;
   end process;
end Table;
```

	NS		Z	
PS	X = 0	X = 1	X = 0	X = 1
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	–	1	–

Dataflow VHDL Model

```
-- The following is a description of the sequential machine of
-- Figure 1-17 in terms of its next state equations.
-- The following state assignment was used:
-- S0-->0; S1-->4; S2-->5; S3-->7; S4-->6; S5-->3; S6-->2

entity SM1_2 is
   port(X,CLK: in bit;
      Z: out bit);
end SM1_2;

architecture Equations1_4 of SM1_2 is
   signal Q1,Q2,Q3: bit;
begin
   process(CLK)
   begin
      if CLK='1' then            -- rising edge of clock
         Q1<=not Q2 after 10 ns;
         Q2<=Q1 after 10 ns;
         Q3<=(Q1 and Q2 and Q3) or (not X and Q1 and not Q3) or
            (X and not Q1 and not Q2) after 10 ns;
      end if;
   end process;
   Z<=(not X and not Q3) or (X and Q3) after 20 ns;
end Equations1_4;
```

$$Q_1(t^+) = Q_2$$

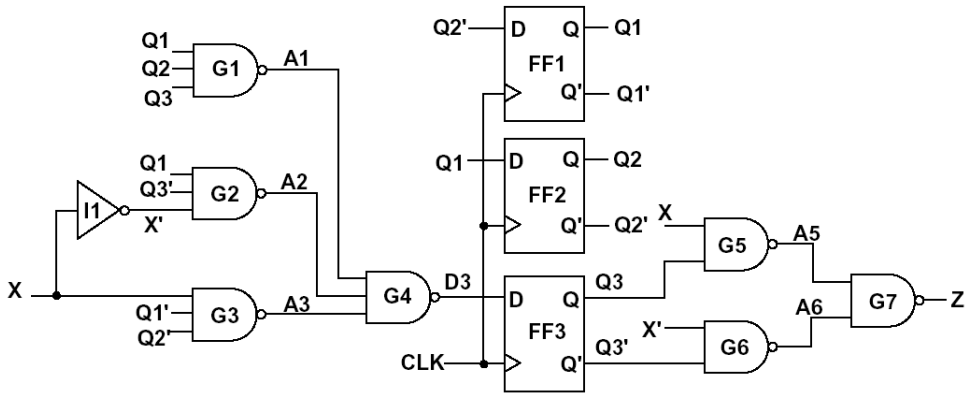
$$Q_2(t^+) = Q_1$$

$$Q_3(t^+) = Q_1Q_2Q_3 + X'Q_1Q'_3 + X'Q'_1Q'_2$$

$$Z = X'Q'_3 + XQ_3$$

Structural Model

```
library BITLIB;
use BITLIB.bit_pack.all;

entity SM1_2 is
   port(X,CLK: in bit;
        Z: out bit);
end SM1_2;

architecture Structure of SM1_2 is
   signal A1,A2,A3,A5,A6,D3: bit:='0';
   signal Q1,Q2,Q3: bit:='0';
   signal Q1N,Q2N,Q3N, XN: bit:='1';
begin
   I1:  Inverter port map (X,XN);
   G1:  Nand3 port map (Q1,Q2,Q3,A1);
   G2:  Nand3 port map (Q1,Q3N,XN,A2);
   G3:  Nand3 port map (X,Q1N,Q2N,A3);
   G4:  Nand3 port map (A1,A2,A3,D3);
   FF1: DFF port map (Q2N,CLK,Q1,Q1N);
   FF2: DFF port map (Q1,CLK,Q2,Q2N);
   FF3: DFF port map (D3,CLK,Q3,Q3N);
   G5:  Nand2 port map (X,Q3,A5);
   G6:  Nand2 port map (XN,Q3N,A6);
   G7:  Nand2 port map (A5,A6,Z);
end Structure
```



Package bit_pack is a part of library BITLIB – includes gates, flip-flops, counters

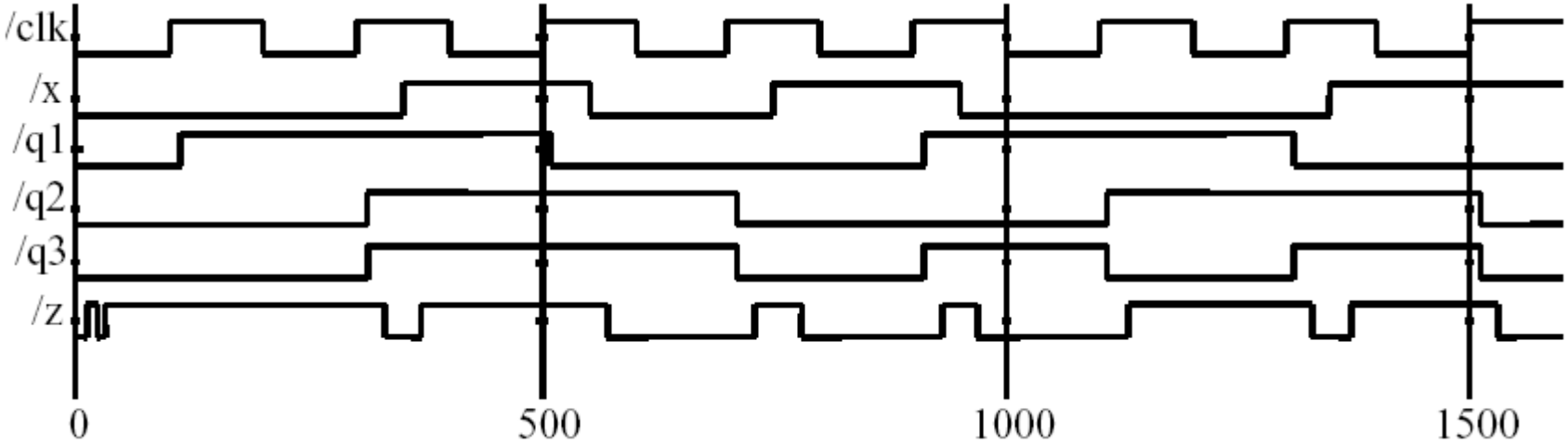
(See Appendix B for details)

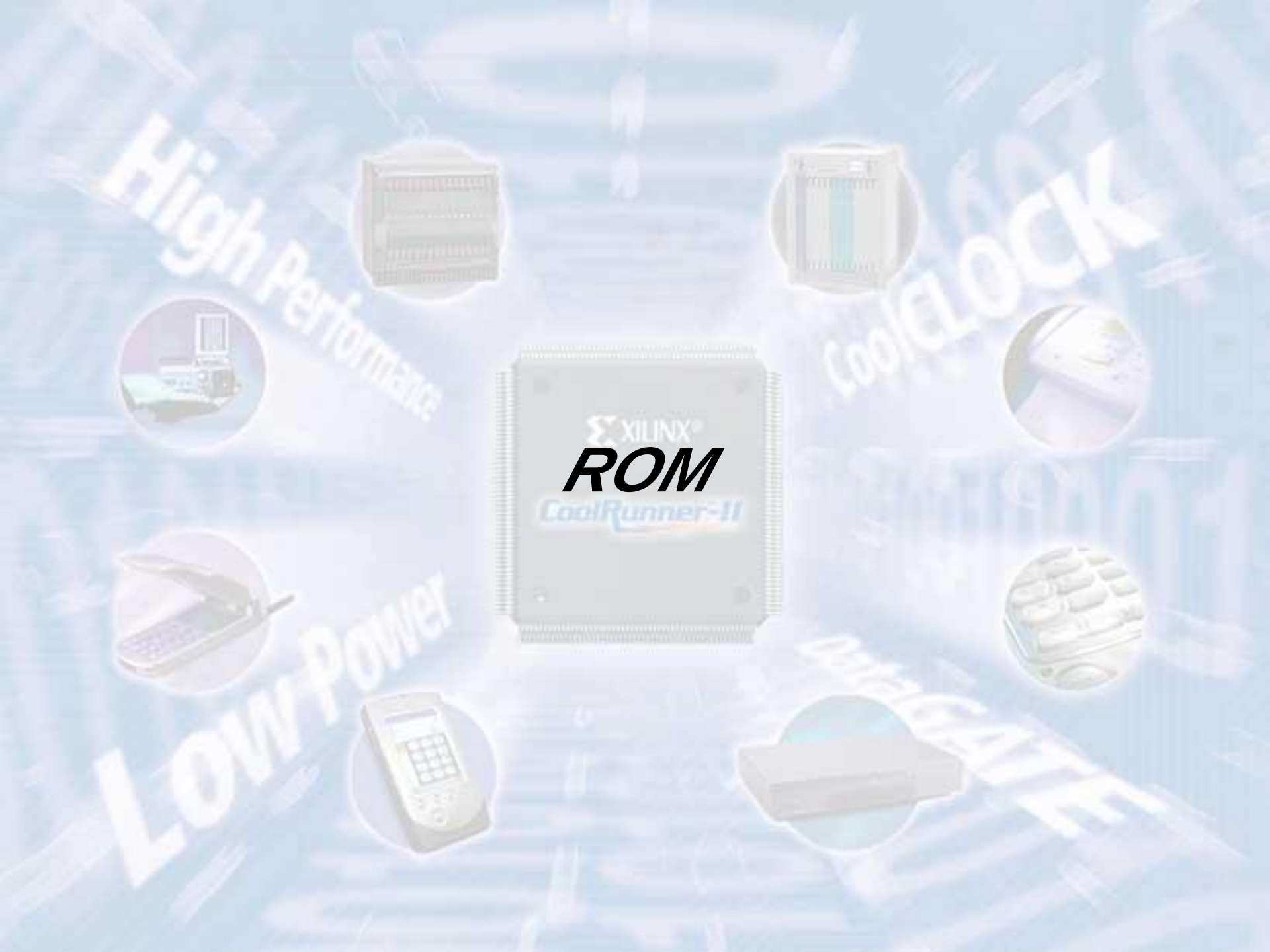
Simulation of the Structural Model

Simulation command file:

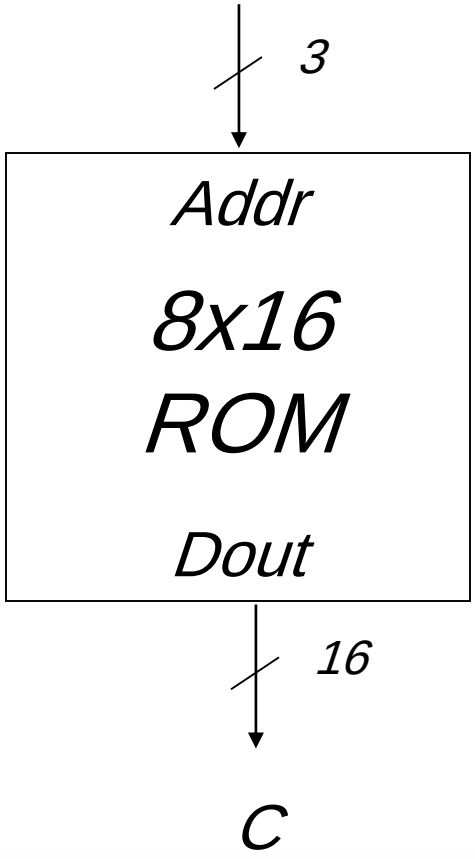
```
wave CLK X Q1 Q2 Q3 Z
force CLK 0 0, 1 100 -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600
```

Waveforms:




ROM
XILINX
CoolRunner-II
High Performance
Low Power
Cool Clock

ROM 8x16 example (1)



ROM 8x16 example (2)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY rom IS

        PORT (
                Addr : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
                Dout : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
            );

END rom;
```

ROM 8x16 example (3)

```
ARCHITECTURE dataflow OF rom IS
SIGNAL temp: INTEGER RANGE 0 TO 7;
TYPE vector_array IS ARRAY (0 to 7) OF STD_LOGIC_VECTOR(15 DOWNTO 0);
CONSTANT memory : vector_array :=
                                    (       X”800A",
                                            X"D459",
                                            X"A870",
                                            X"7853",
                                            X"650D",
                                            X"642F",
                                            X"F742",
                                            X"F548");
BEGIN

    temp <= to_integer(unsigned(Addr));
    Dout <= memory(temp);

END dataflow;
```

ROM 8x16 example (4)

```
ARCHITECTURE dataflow OF rom IS
TYPE vector_array IS ARRAY (0 to 7) OF STD_LOGIC_VECTOR(15 DOWNTO 0);
CONSTANT memory : vector_array :=
                                    (       X"800A",
                                            X"D459",
                                            X"A870",
                                            X"7853",
                                            X"650D",
                                            X"642F",
                                            X"F742",
                                            X"F548");
BEGIN

    Dout <= memory(to_integer(unsigned(Addr)));

END dataflow;
```