Lattice VHDL Training

Part I

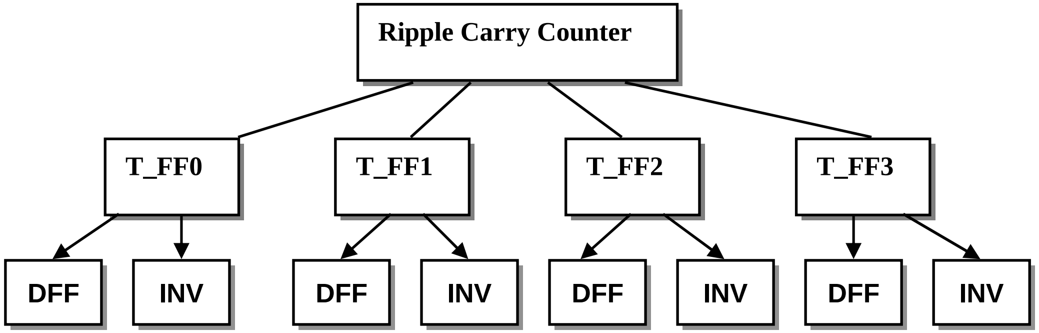
VHDL

Basic Modeling Structure

VHDL Design Description

- VHDL language describes a digital system as a set of modular blocks. Each modular block is described by a pair of **entity** and **architecture**.

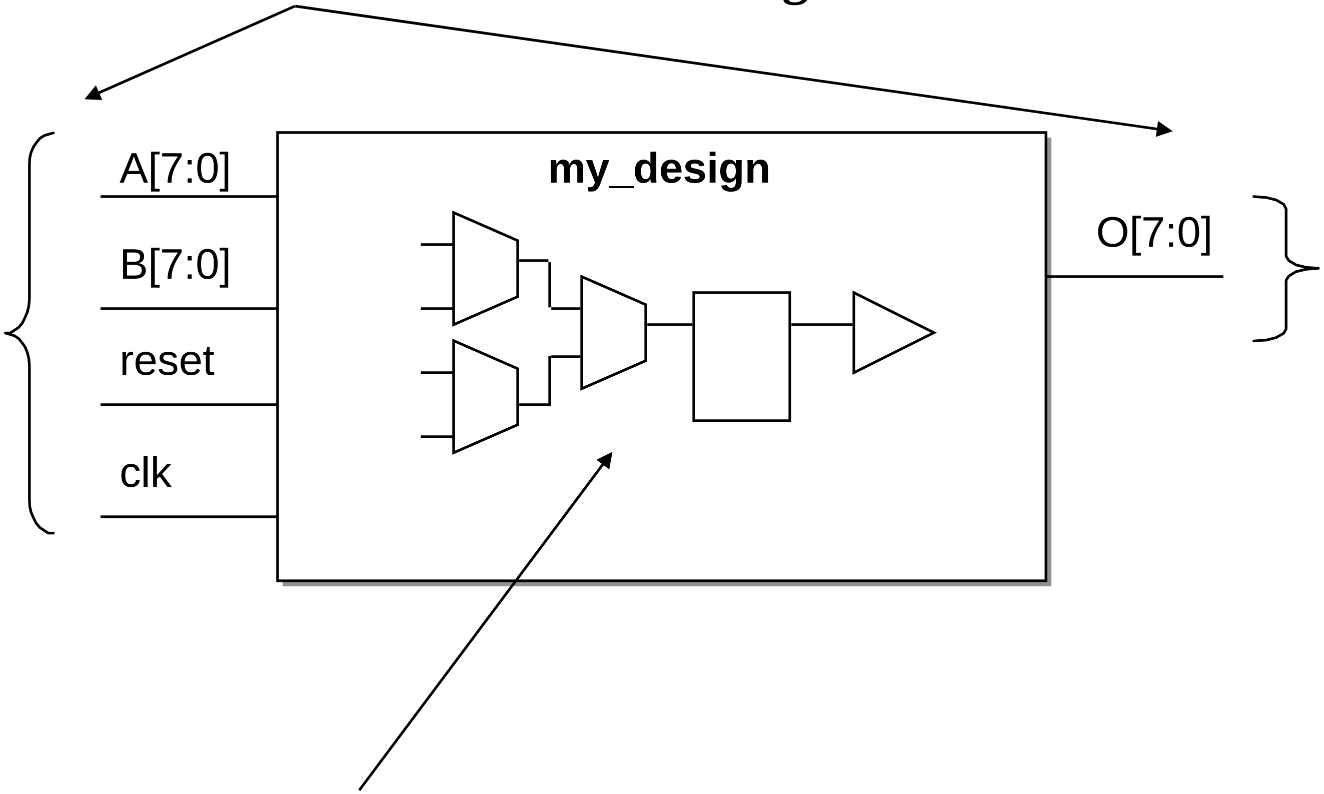
entity counter …. ….. **end** counter;	**architecture** counter_design **of** counter …. …. **end** counter_design;
entity tff …. …. **end** tff;	**architecture** tff_design **of** tff **is** …. …. **end** tff;
entity inv …. ….. **end** inv;	**architecture** inv_design **of** inv …. …. **end** inv_design;
entity dff …. …. **end** dff;	**architecture** dff_design **of** dff **is** …. …. **end** dff;



VHDL Design Descriptions

- VHDL design description consist of an ENTITY and ARCHITECTURE pair

❑ The **ENTITY** describes the **design I/Os**



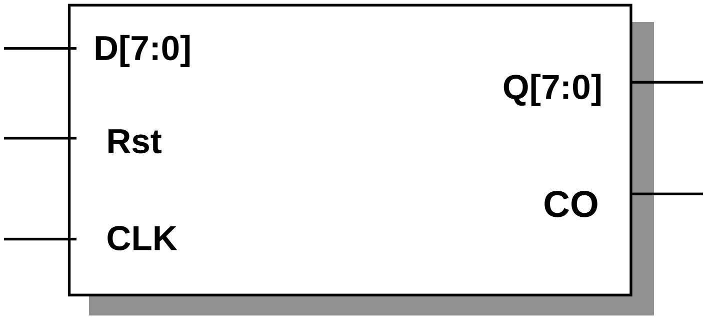
❑ The **ARCHITECTURE** describes the **content of the design**

The Entity - example

- VHDL description of the entity:

```
ENTITY black_box is
     port (  rst :   IN std_logic;
             clk :   IN std_logic;
             d   :   IN std_logic_vector (7 downto 0);
             q   :   BUFFER std_logic_vector (7 downto 0);
             co  :   OUT std_logic  );
END black_box;
```

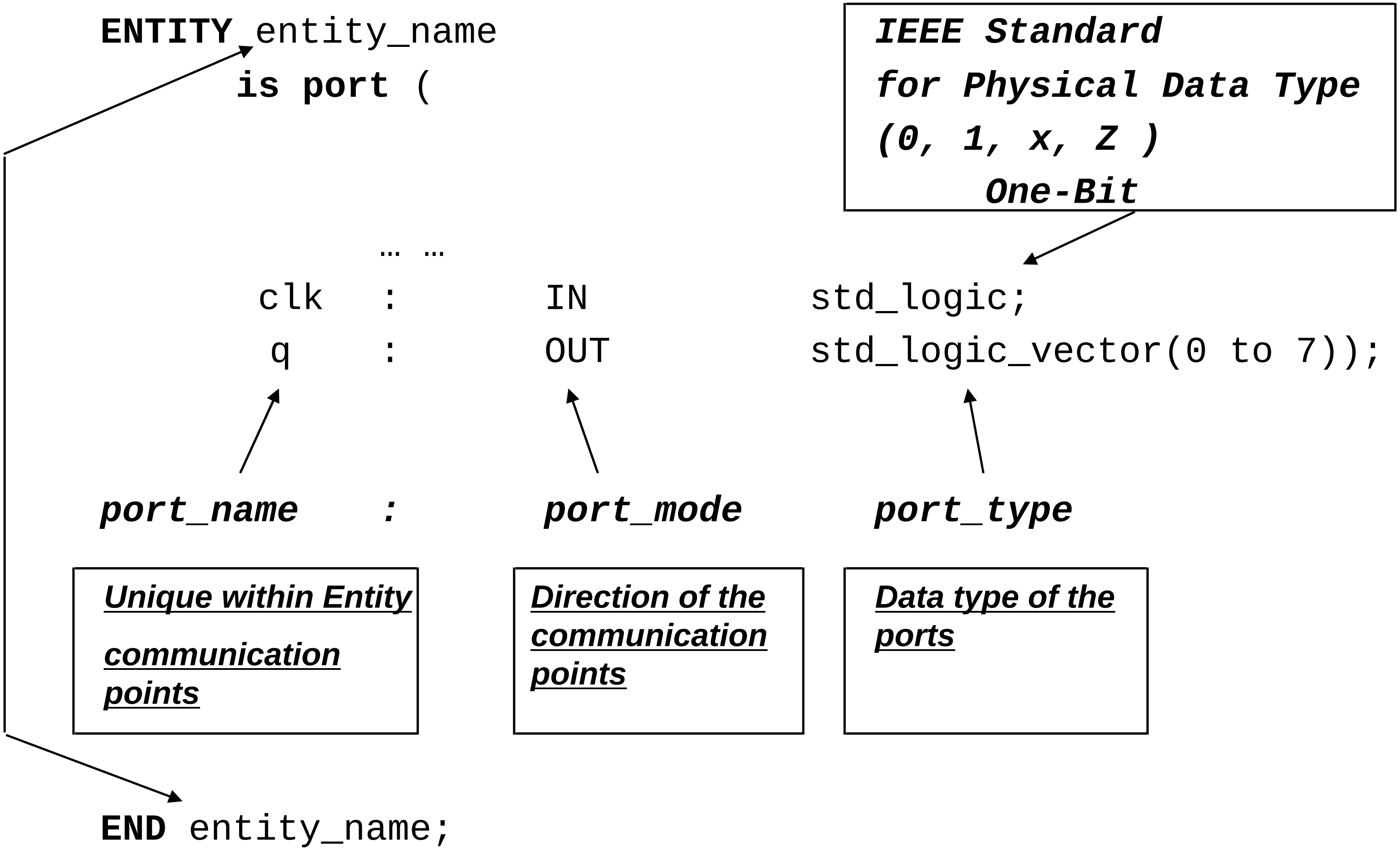
Black Box



- Note: VHDL is **NOT** case sensitive!

The Entity - General Format for Port Declaration

- Entity **only** describes the circuit **interface** but not function



```
ENTITY entity_name
       is port (

              … …
         clk    :       IN          std_logic;
          q     :       OUT         std_logic_vector(0 to 7));

port_name       :       port_mode       port_type

END entity_name;
```

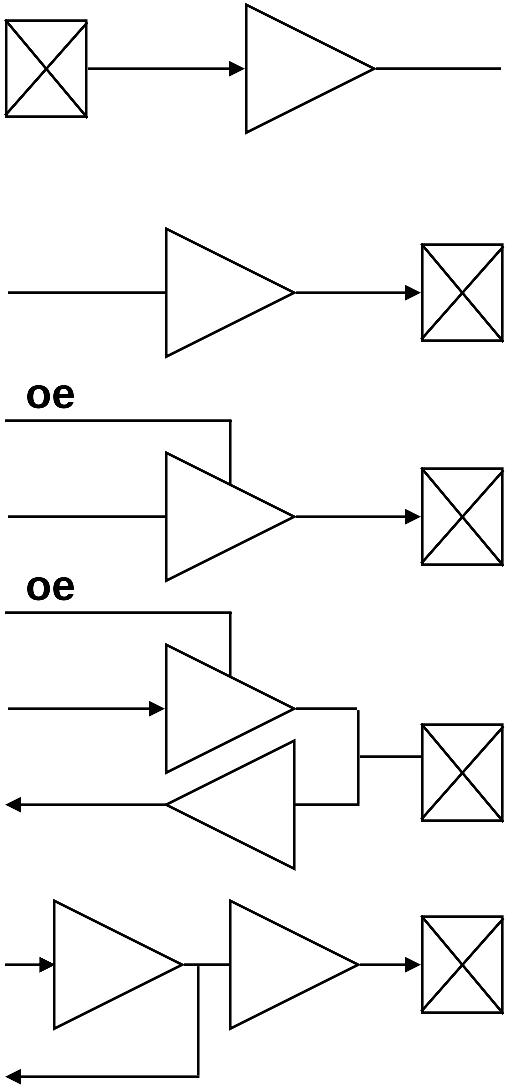
IEEE Standard
for Physical Data Type
(0, 1, x, Z)
One-Bit

Unique within Entity communication points

Direction of the communication points

Data type of the ports

PORT Modes



- IN
- OUT
- OUT (Tri-State)
- INOUT
- Buffer

PORT Types

- **integer** Useful as index holders for loops and constants. When used for I/O signals, usually reference counters
- **Boolean** Can take values of ‘TRUE’ and ‘FALSE’
- **std_logic** Standard industry logic type, has values of ‘0’, ‘1’, ‘X’, and Z’ defined by IEEE std 1164.
- **std_logic_vector** A grouping of std_logic, standard industry logic type

PORT Types

- VHDL is a strongly typed language. You cannot assign one signal type to another signal type.

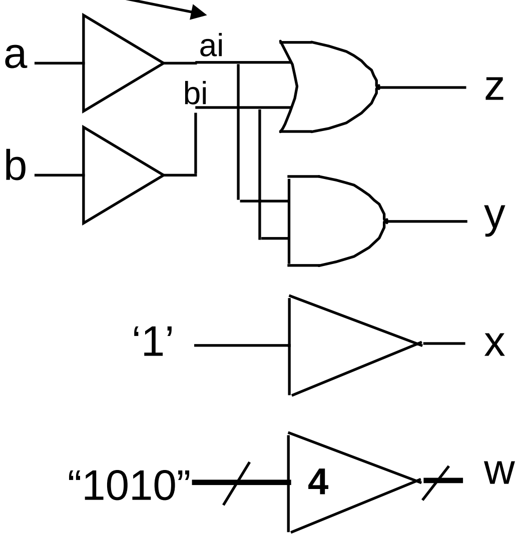
```
ENTITY example is
   port (Q1 : OUT Integer; …
          Q2 : OUT std_logic_vector( 3 downto 0));
END example;

Architecture behavior of example is
begin

         -- Q2 is declared in 4 bit std_logic_vector type

  ✓      Q1 <= 7;                 -- 7 : decimal integer;
         Q2 <= “1001”;            -- “1001” : 4 bit std_logic vector
  ✗      ……
         Q2 <= 9;                 -- 9 : decimal integer
         …  …
end behavior;
```

The Architecture body - General Format

```
Entity entity_name is port ( … … );
End    entity_name;

ARCHITECTURE architecture_name of entity_name is

 signal ai, bi : std_logic;

BEGIN
   ai <= a;
   bi <= b;
   y <= (ai AND bi);
   z <= (ai OR bi);
   x <= ‘1’;
   w <= “1010”;

END architecture_name;
```

Architecture must be associated with entity

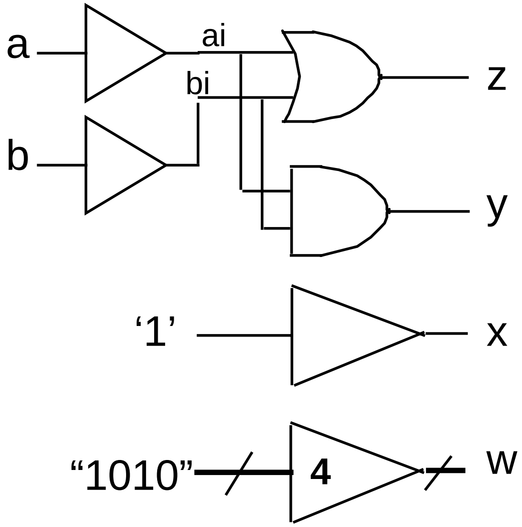
Architecture declaration section: declare internal signals (nets)



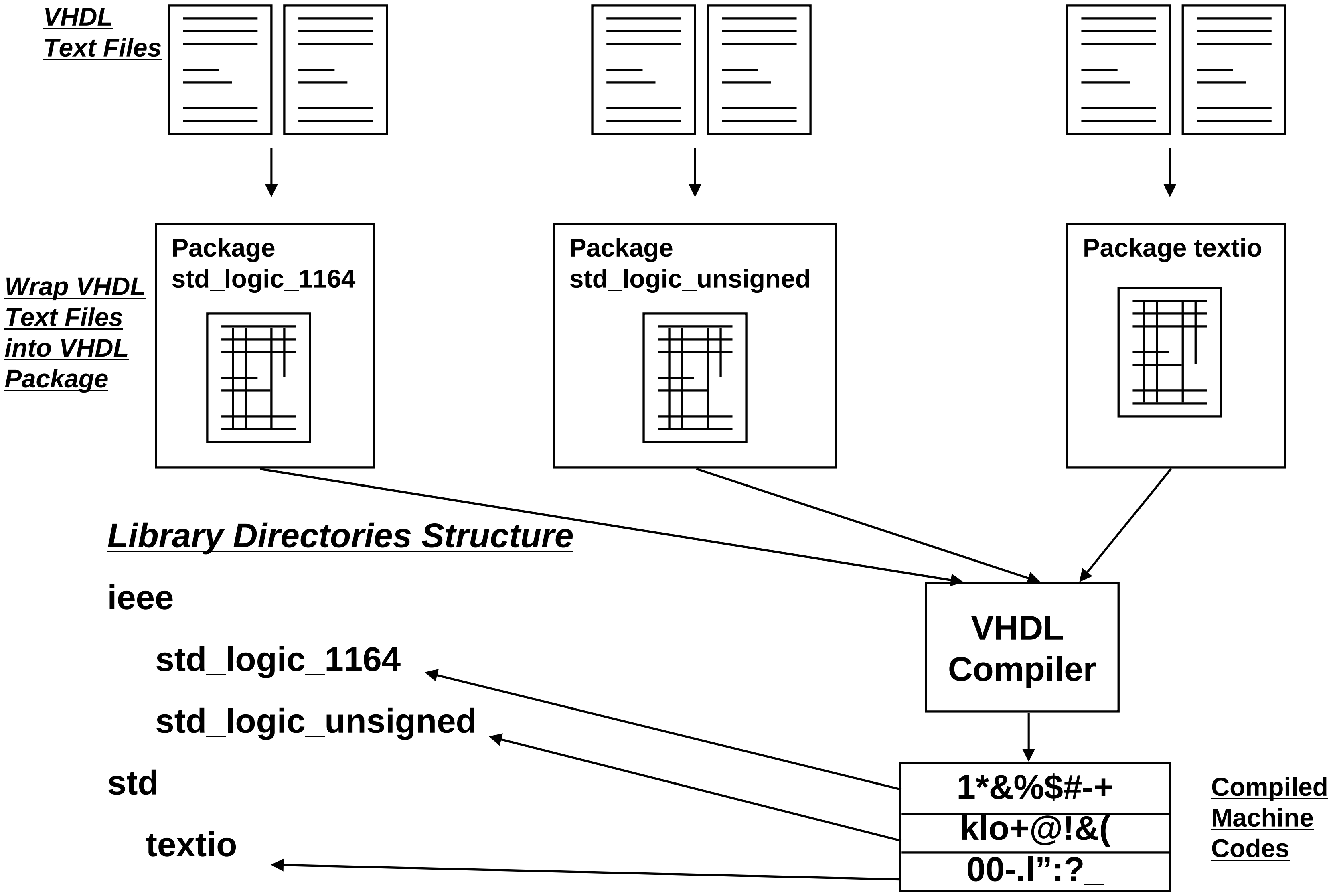
Entity / Architecture / Libraries - example

- Every design has an ENTITY/ARCHITECTURE pair

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY logic is
      PORT (  a,b :    IN std_logic;
              z, x, y: OUT std_logic;
              w:       OUT std_logic_vector (3 downto 0));
END logic;

ARCHITECTURE behavior of logic is
      signal ai, bi : std_logic;
BEGIN
      ai <= a;
      bi <= b;
      y <= (ai AND bi);
      z <= (ai OR bi);
      x <= ‘1’;
      w <= “1010”;

END behavior;
```



Libraries



VHDL Text Files

Wrap VHDL Text Files into VHDL Package

Package std_logic_1164

Package std_logic_unsigned

Package textio

Library Directories Structure

ieee

std_logic_1164

std_logic_unsigned

std

textio

VHDL Compiler

1*&%$#-+

klo+@!&(

00-.l":?_

Compiled Machine Codes

Libraries

- Library is a place to keep *precompiled packages* so that they can be used in other VHDL designs

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
```

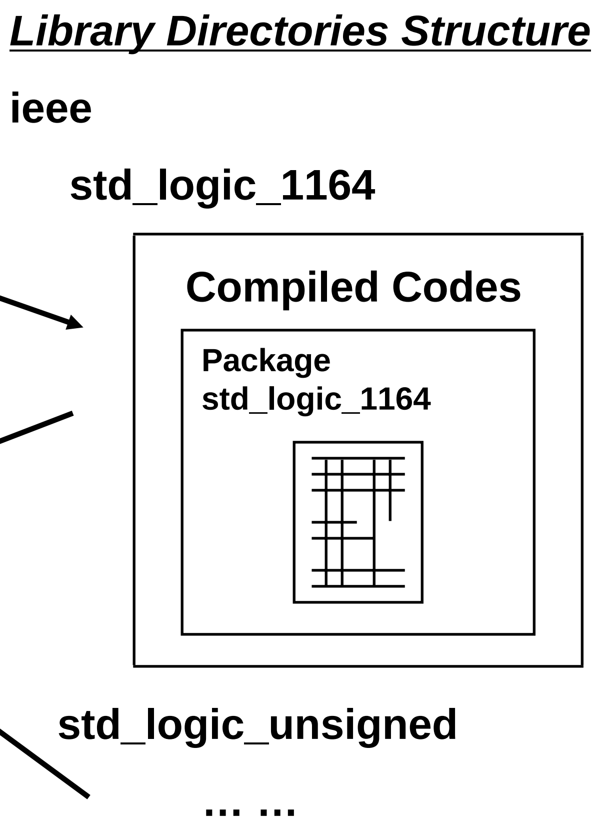
ieee - symbolic name for IEEE standard library.
Contains packages std_logic_1164, std_logic_unsigned, etc.

std_logic_1164 - name of the VHDL package
Package std_logic_1164 contains declaration of data type for std_logic and std_logic_vector.

std_logic_unsigned - name of the VHDL package.
Package std_logic_unsigned contains the declaration of operators, functions for std_logic and std_logic_vector arithmetic operations.

Libraries

```
LIBRARY ieee;
-- Makes Library ieee visible
-- to the following VHDL design.
USE ieee.std_logic_1164.ALL;
-- Indicates design can use all
-- declarations of Package
-- std_loigc_1164 in library ieee.
USE ieee.std_logic_unsigned.ALL;
```



Library Directories Structure

ieee

std_logic_1164

Compiled Codes

Package
std_logic_1164

std_logic_unsigned

… …

```
ENTITY design is Port
    ( q : OUT std_logic; … )
End design;
architecture context of design is
begin
        q <= a + b; … …
End context;
```

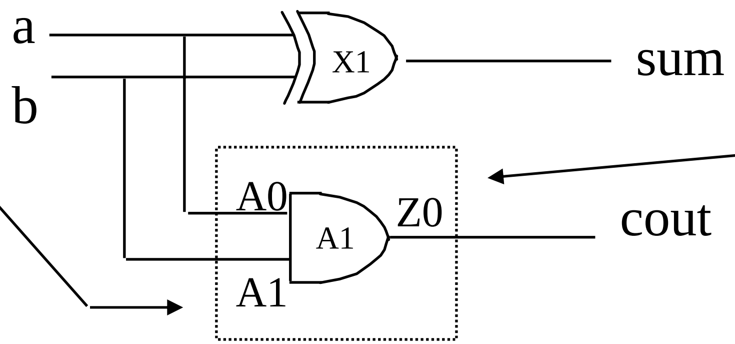
VHDL Language Syntax &

VHDL Design Methods

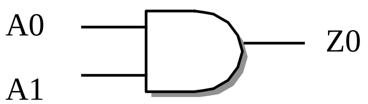
Concurrent Statement

Instantiation Statement → *Structural Modeling*

```
Architecture structural_modeling of one_bit_half_adder is
  begin
        X1: XOR2 Port Map ( Z0 => sum,  A0 => a, A1 => b );
        A1: AND2 Port Map ( Z0 => cout, A0 => a, A1 => b );

        -- A1: AND2 Port Map (    cout,         a,         b );

end structural_modeling;
```



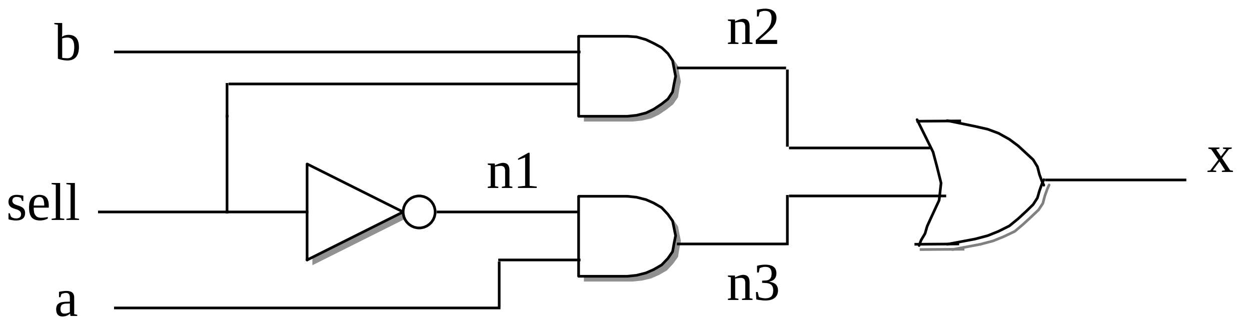
Library Symbol: AND2



Entity AND2 **is Port** (
Z0 : **out** std_logic;
A0 : **in** std_logic;
A1 : **in** std_logic);
END AND2;

Concurrent Statement

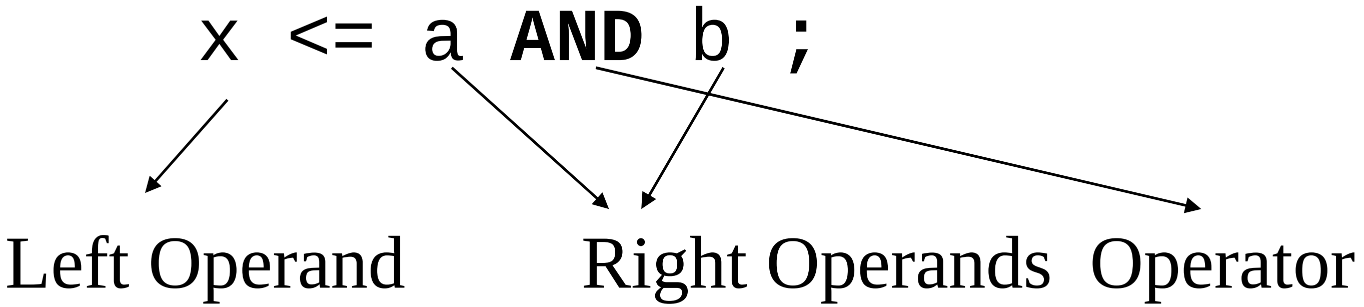
- ***Dataflow Modeling by***
  - ❑ boolean equations assignments
  - ❑ conditional assignments (When-Else, With-Select-When)



```
Architecture dataflow_modeling of example is
     signal n1, n2, n3 : std_logic; -- Declare Internal Nets
begin
     x <= n2 OR n3;              n2 <= b AND sell;
     n3 <= n1 AND a;             n1 <= NOT sell;

     --   x <= (a AND (NOT sel1)) OR (b AND sel1);
end example;
```

Concurrent Statement - Boolean Equation Assignment

- **Boolean Equation Assignment Format**

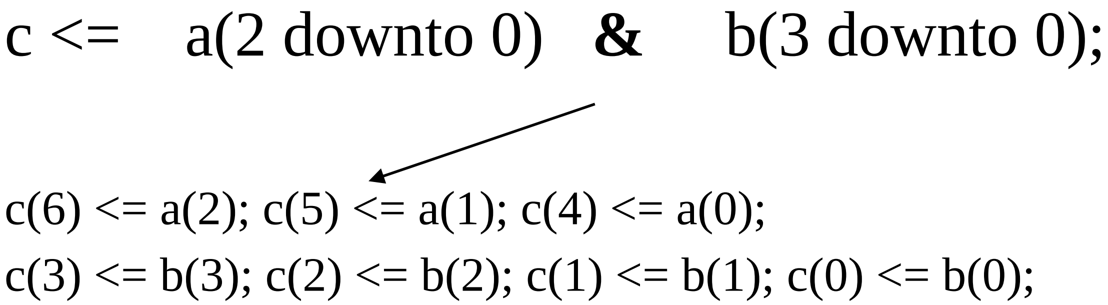


x <= a **AND** b ;

Left Operand Right Operands Operator

The ‘<=‘ operation is also used to signify ‘taking on the value of’

- **Concatenate Assignment**



c <= a(2 downto 0) **&** b(3 downto 0);

c(6) <= a(2); c(5) <= a(1); c(4) <= a(0);

c(3) <= b(3); c(2) <= b(2); c(1) <= b(1); c(0) <= b(0);

Standard VHDL Operators

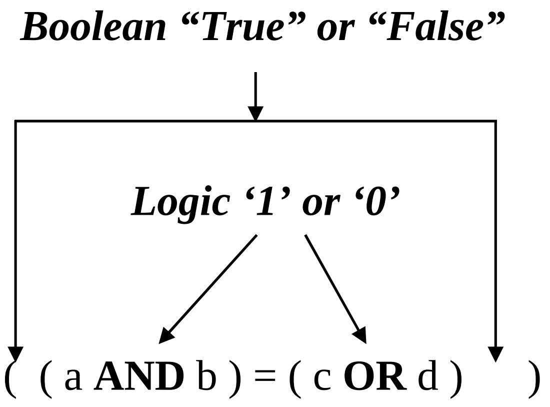
- **Logical Operators**

  AND

  OR

  XOR

  NOT


Boolean "True" or "False"

Logic '1' or '0'

((a **AND** b) = (c **OR** d))

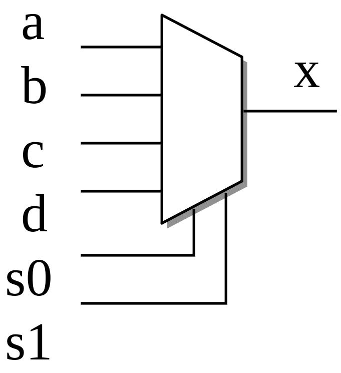
a, b, c, d : std_logic

- **Relational Operators**

=	Equal to
/=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Concurrent Conditional Assignment

- With-Select-When
- Example

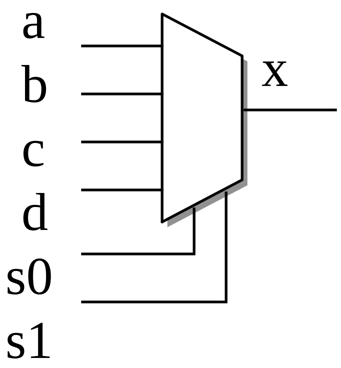


```
entity mux is port (
     a, b, c, d:in std_logic;
     s:      in std_logic_vector(1 downto 0);
     x:      out std_logic );
end mux;

architecture archmux of mux is
begin
  with s select
     x <= a when “00”,      -- x is assigned based on s
          b when “01”,
          c when “10”,
          d when “11”;
end archmux;
```

Concurrent conditional Assignment

- When-Else
- Same example of 4-to-1 mux

```
architecture archmux of mux is

begin
     x <= a when ( s = “00” ) else
          b when ( s = “01” ) else
          c when ( s = “10” ) else
          d;
end archmux;
```



- WITH-SELECT-WHEN must specify all mutually exclusive conditions
- WHEN-ELSE does not have to

With-Select-When vs. WHEN-ELSE

```
with s select
     x <= a when   “00”,
          b when   “01”,
          c when   “10”,
          d when   “11”;
```

Specify chose values based on data type of “S”

- Specify all mutual exclusive conditions

```
x <= a when ( s = “00” ) else
     b when ( s = “01” ) else
     c when ( s = “10” ) else
     d;
```

Specify Boolean conditions

- Don’t have to specify all mutually exclusive conditions, the last one can be defaulted after “ELSE” keyword

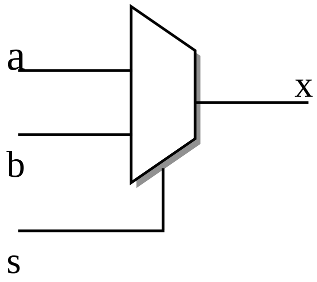
Sequential Statements

- **Process Statement** → ***Behavioral Modeling***
- A PROCESS is used to describe sequential events and is included in the ARCHITECTURE of the design.
- An ARCHITECTURE can contain several PROCESS statements.
- PROCESS statements have 3 parts:
  - ❑ Sensitivity list :
    - includes signals used in the PROCESS
    - process is active when a signal in sensitivity list changes value
  - ❑ PROCESS :
    - the description of behavior
  - ❑ BEGIN - END PROCESS statement:
    - describes the beginning & ending of the PROCESS

Process - Sequential Statement

- Simple example of PROCESS

```
mux: PROCESS (a, b, s) -- the sensitivity list
BEGIN
     if ( s = '0' ) then
            x <= a;
     else                                  -- define the process section
            x <= b;
     end if;
END PROCESS mux;
```

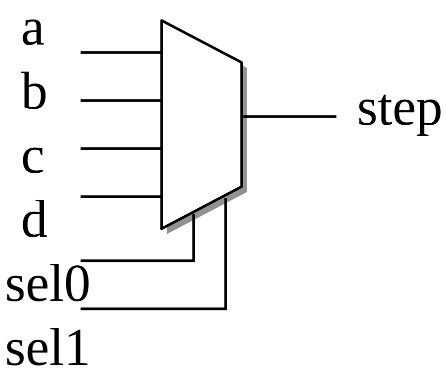


- Here the process ‘mux’ is sensitive to signals ‘a’,‘b’ and ‘s’. Whenever signal ‘a’ or ‘b’ or ‘s’ changes value, the statements inside the process will be evaluated

IF - THEN - ELSE

- It is a sequential statement and can only be used in PROCESS
- To select a specific execution path based on Boolean evaluation of a condition or set of conditions
- Cascaded If-Then-Else

```
PROCESS (select, a, b, c, d)
BEGIN
     if (select = “00”) then
             step <= a;
     elsif (select = “01”) then
             step <= b;
     elsif (select = “10”) then
             step <= c;
     else
             step <= d;
     end if;
END PROCESS;
```

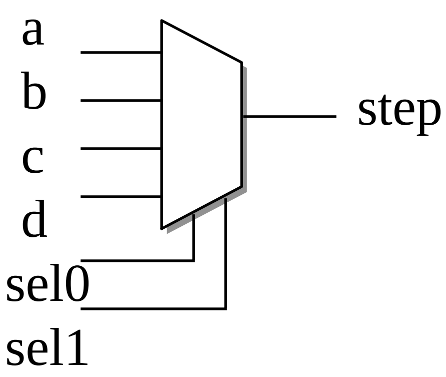


- ELSIF allows multiple conditions in one statement
- Must have an “END IF” statement for every “IF” statement

IF - THEN - ELSE

- Nested If-Then-Else

```
PROCESS (select, a, b, c, d)
BEGIN
     if (select = “00”) then
             step <= a;
     else
        if (select = “01”) then
             step <= b;
        else
             if (select = “10”) then
                    step <= c;
             else
                    step <= d;
             end if;
        end if;
     end if;

END PROCESS;
```



CASE - WHEN

- It is a sequential statement and can only be used in PROCESS

```
ARCHITECTURE archdesign OF design IS
BEGIN
     decode: PROCESS (a, b, c, option)
     BEGIN
            CASE option IS
                   WHEN "00"   => output <= a;
                   WHEN "01"   => output <= b;
                   WHEN "10"   => output <= c;
                   WHEN OTHERS => output <= '0';
            END CASE;
     END PROCESS decode;
END archdesign;
```

- OTHERS is all other possible value for signals of type std_logic

Data Objects -Signal, Constant, Variable

- SIGNAL
  - ❑ used to declare internal signals; not external signals
  - ❑ interconnects components
  - ❑ may be assigned to an external signal

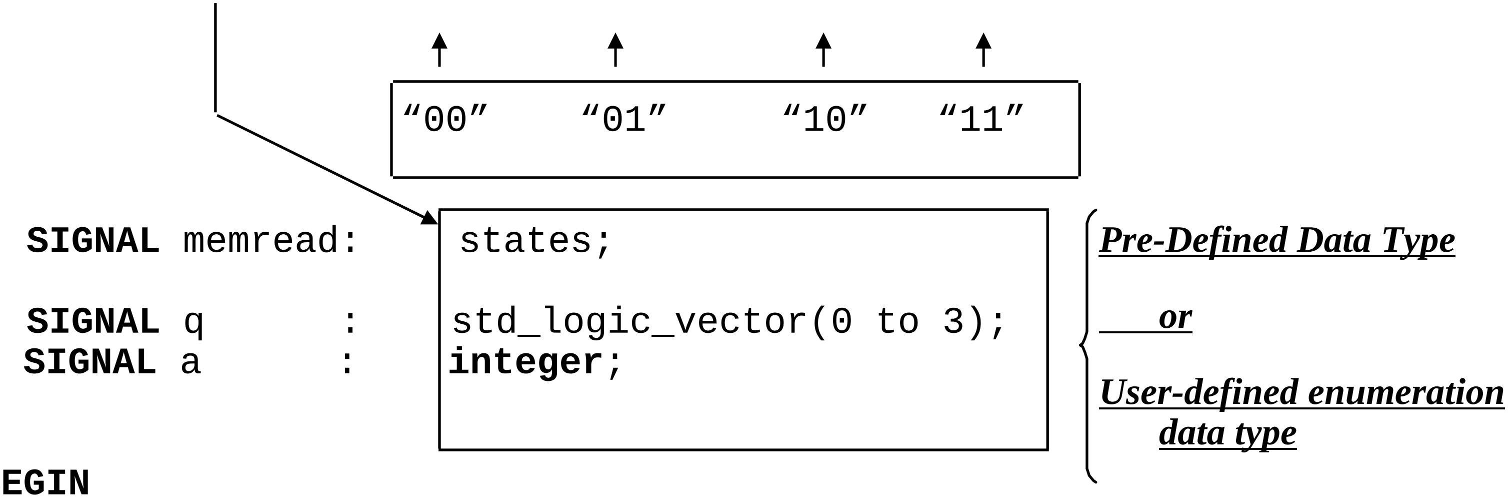
```
ARCHITECTURE behavior of example is
  SIGNAL count:       std_logic_vector (3 downto 0);
  SIGNAL flag:        integer;
  SIGNAL mtag:        integer range 0 to 15;
  SIGNAL stag:        integer range 100 downto 0;
BEGIN

--mtag is a 4-bit array; MSB is mtag(0); LSB is mtag(3)
--stag is a 7-bit array; MSB is stag(6); LSB is stag(0)
--always declared in ARCHITECTURE section
```

Data Objects

- SIGNAL in User-defined enumeration type

  ❑ represents state elements in a state-machine

```
ARCHITECTURE behavior of example is

  TYPE states is (state0, state1, state2, state3);
```



“00”   “01”   “10”   “11”

```
  SIGNAL memread:     states;

  SIGNAL q      :    std_logic_vector(0 to 3);
  SIGNAL a      :    integer;

BEGIN
```

Pre-Defined Data Type

or

User-defined enumeration data type

```
--each state (state0, state1, etc) represents a distinct state.
```

Data Objects

- CONSTANT
  - ❑ holds a specific value of a type that cannot be changed within the design description

```
ARCHITECTURE behavior of example is
  CONSTANT width: integer := 8;
BEGIN
-- “width” is a constant with integer type and has a value of “8”
```

- VARIABLE
  - ❑ used to declare local values only within a given PROCESS.

```
PROCESS (s)
  VARIABLE result: integer := 12;
BEGIN

-- “result” is a VARIABLE with an intial value of “12”.
-- value of “result” may be modified within a PROCESS.
```

- variable assignment use :=
- port/signal assignment use <=

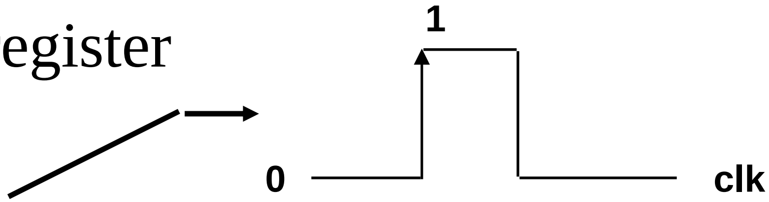
Understand VHDL Synthesis

and

VHDL Design Application

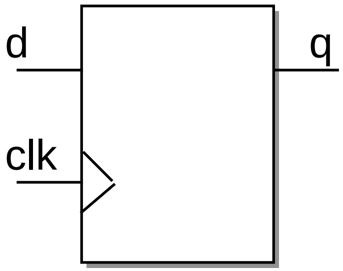
REGISTERS in Behavioral VHDL

- 3 ways to describe a register



```
❑ PROCESS (clk)
  BEGIN
    IF (clk’event and clk=‘1’) THEN -- rising edge of clk
               q <= d;
    END IF;
  END PROCESS;
-- falling edge of clk => (clk’event and clk = ‘0’)
```

```
❑ PROCESS (clk)
  BEGIN
    IF RISING_EDGE (clk) THEN
               q <= d;
    END IF;
  END PROCESS;
```



```
❑ PROCESS   -- no sensitivity list
  BEGIN
    WAIT UNTIL clk’event AND clk = '1';
               q <= d;
  END PROCESS;
```

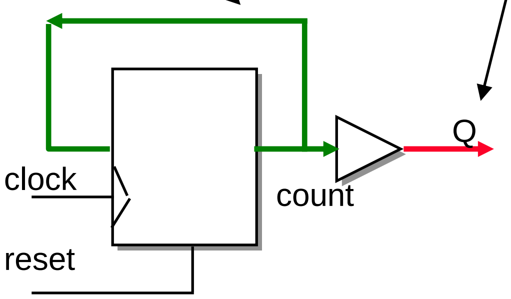
Synchronous/Asynchronous Reset Register

```
ARCHITECTURE behavior of synchronous_reset_register is
BEGIN
  PROCESS (clk)
  BEGIN    IF (clk’EVENT and clk = ‘1’) then
               IF (rst = ‘1’) then
                      q <= ‘0’;
               ELSE
                      q <= d;
               END IF;
           END IF;
  END PROCESS;
END behavior;

ARCHITECTURE behavior of Asynchronous_reset_register is
BEGIN
  PROCESS (rst, clk)
  BEGIN   IF (rst = ‘1’) then
               q <= ‘0’;
          ELSIF (clk’EVENT and clk = ‘1’) then
               q <= d;
          END IF;
  END PROCESS;
END behavior;
```

Synchronous Reset Counter

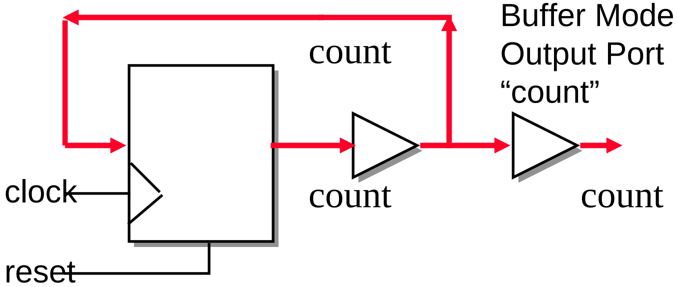
Signal “count” represent hardware NET

Output Port “Q”



```
Entity counter is
   port (clock : IN std_logic;
         reset : IN std_logic;
         Q: OUT std_logic_vector(3 downto 0));
End counter;

Architecture behavior of counter is
   signal count : std_logic_vector(3 downto 0);
begin    upcount: PROCESS (clock)
         BEGIN       IF (clock’EVENT AND clock = '1') THEN
                          IF reset = '1' THEN
                                  count <= “0000”; -- synchronous
                          ELSE
                                  count <= count + “0001”;
                          END IF;
                     END IF;
         END PROCESS upcount;
         Q <= count;
End behavior;
```

Asynchronous Reset Counter

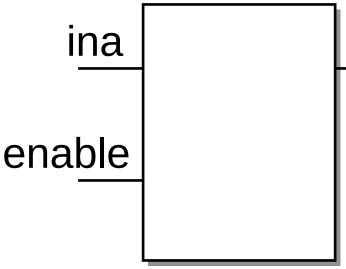


Buffer Mode Output Port “count”

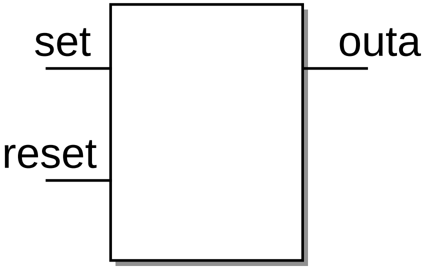
```
Entity counter is
   port (clock, reset : IN std_logic;
         count: buffer std_logic_vector(3 downto 0));
End counter;

Architecture behavior of counter is
   begin upcount: PROCESS (clock, reset)
         BEGIN   IF (reset = '1') THEN
                          -- reset has a higher priority
                          count <= “0000”; -- asynchronous
                 ELSIF (clock’EVENT AND clock = '1') THEN
                          count <= count + “0001”;
                 END IF;
         END PROCESS upcount;
End behavior;
```

LATCHES - in Behavioral VHDL

```
ARCHITECTURE behavior of d_latch is
BEGIN
   PROCESS (enable)
   BEGIN
         IF (enable = '1') then
                outa <= ina;
         END IF;
   END PROCESS;
END behavior;
```



```
ARCHITECTURE behavior of set_reset_latch is
BEGIN
   PROCESS (set, reset)
   BEGIN
         IF (set = '1' AND reset = '0') then
                outa <= '1';
         ELSIF (set = '0' AND reset = '1') then
                outa <= '0';
         END IF;
   END PROCESS;
END behavior;
```

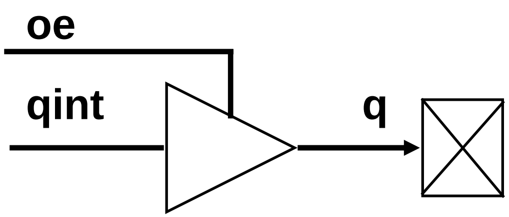


Synchronous/Asynchronous Reset Latch

```
ARCHITECTURE behavior of synchronous_reset_d_latch is
BEGIN
  PROCESS (enable)
  BEGIN   IF (enable = ‘1’) then
               IF ( rst = ‘1’) then
                      outa <= ‘0’;
               ELSE
                      outa <= ina;
               END IF;
          END IF;
  END PROCESS;
END behavior;

ARCHITECTURE behavior of asynchronous_reset_d_latch is
BEGIN
  PROCESS (enable, rst)
  BEGIN   IF (rst = ‘1’) then
               outa <= '0’;
          ELSIF (enable = ‘1’) then
               outa <= ina;
          END IF;
  END PROCESS;
END behavior;
```

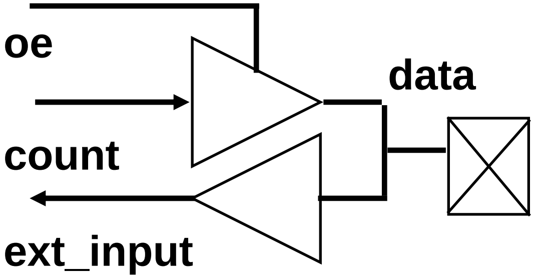
Output Enables

```
Entity design is port ( q : out std_logic_vector(0 to 6); …);
End design;    -- Tri-State Port declared as OUT

ARCHITECTURE behavioral of design is
   BEGIN
   q <= qint when (oe=’1') else "ZZZZZZZ" ;
END behavioral;

ARCHITECTURE behavioral of design is
   BEGIN
   PROCESS (oe)
   BEGIN  if ( oe = ‘1’ ) then
                q <= qint;
          else
                q <= “ZZZZZZZ”;
          end if;
   END PROCESS;
END behavioral;
```



Bi-Directional Port

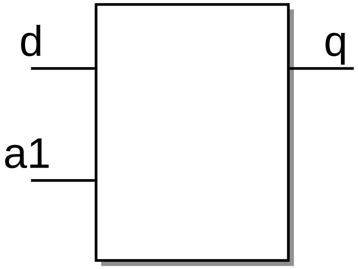
```
Entity design is port(-- Bi-Directional Port declared as INOUT
   data:inout std_logic_vector(0 to 7);...);
End design;

ARCHITECTURE behavior OF design IS
   SIGNAL ext_input : std_logic;
BEGIN
   data <= count WHEN oe = ’1' ELSE “ZZZZZZZZ” ;
   ext_input <= data;
END behavior;

ARCHITECTURE behavior OF design IS
   SIGNAL ext_input : std_logic;
BEGIN
   PROCESS (oe)
   BEGIN
         IF (oe = ‘1’ ) THEN
                  data <= count;
         ELSE
                  data <= “ZZZZZZZZ”;
         END IF;
   END PROCESS;
   ext_input <= data;
END behavior;
```



CPLD Optimization - Caution using “If-Then-Else”

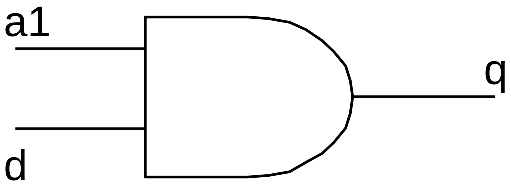
- The following VHDL file does not specify the value of “q” when “a1” is equal to “0”, thus creating a “Latch”.

```
PROCESS (a1, d)
BEGIN
      IF (a1 = ‘1’) then
               q <= d;
      END if;
END PROCESS;
```



- The following VHDL file specifies the value of “q” when “a1” is equal to “0”, thus creating an AND gate

```
PROCESS (a1, d)
BEGIN
      IF (a1 = ‘1’) then
               q <= d;
      ELSE
               q <= ‘0’;
      END if;
END PROCESS;
```



CPLD Optimization - Caution using “Case-When”

- This VHDL file generates unwanted latches because not all states are defined.

```
signal sel:std_logic_vector(0 to 1);


PROCESS (sel,a,b)
BEGIN
  CASE sel IS
    WHEN “00”=>q<=a;
    WHEN “11”=>q<=b;
  END CASE;
END PROCESS;
```

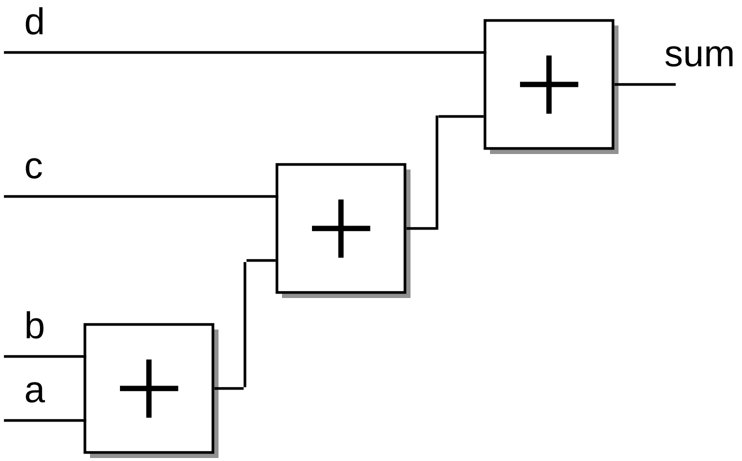
- This VHDL file generates a multiplexer correctly because states “10” and “01” are defined.

```
-- sel can choose either of
   “00”,”01”,”10”, “11”

PROCESS (sel,a,b)
BEGIN
  CASE sel IS
    WHEN “00”=>q<=a;
    WHEN “11”=>q<=b;
    WHEN OTHERS=>q<=‘0’;
  END CASE;
END PROCESS;
```

CPLD Optimization - Use of Parentheses

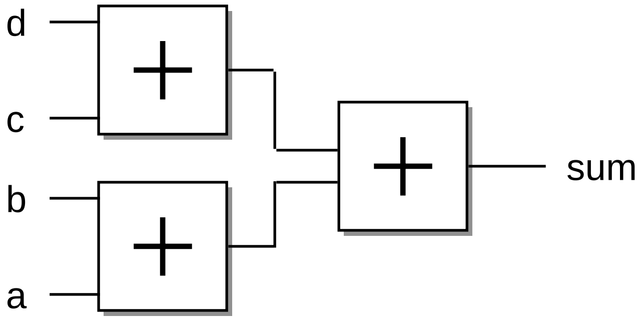
- Example listing below generates a design that uses three cascaded adders and three logic levels.

```
PROCESS (a,b,c,d)
BEGIN
        sum<=a+b+c+d;
END PROCESS;
```



- Example listing below, by inclusion of two sets of parentheses, generates two parallel adders whose outputs the circuit then adds. This design results in two logic levels.

```
PROCESS (a,b,c,d)
BEGIN
  sum<=(a+b)+(c+d);
END PROCESS;
```

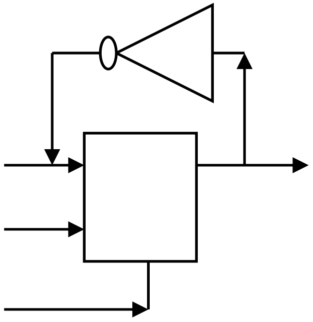


VHDL Hierarchical Design

Concept of Hierarchical Design

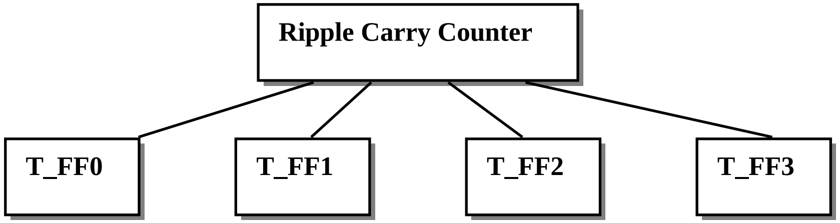
Low-level Design

```
ENTITY tff is port
   (d, clk, rst : in std_logic; q : out std_logic);
end tff;
ARCHITECTURE context of tff is begin
   D_FF:         FD21 port map(d, q, clk, rst);
   NOT_Gate:     INV  port map(d, q);
end context;
```



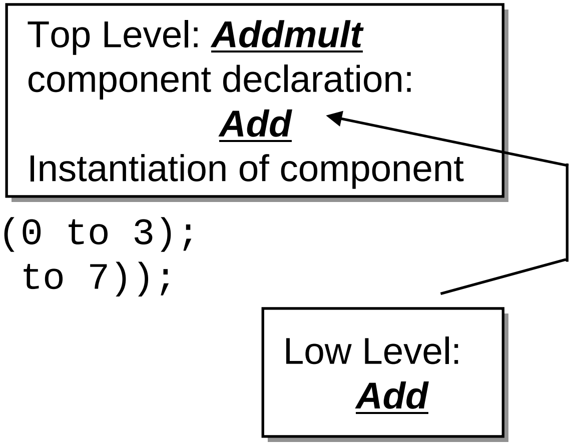
High-Level Design

```
ENTITY ripple_counter is port
   (cnt_en, clk, rst : in std_logic; q : out std_logic  );
end ripple_counter;

ARCHITECTURE structure of
    ripple_counter is
BEGIN
        tff0 : tff port map (…);
        tff1 : tff port map (…);
        tff2 : tff port map (…);
        tff3 : tff port map (…);
end structure;
```



COMPONENT

- Top level example



Top Level: ***Addmult***
component declaration:
Add
Instantiation of component

Low Level:
Add

```
ENTITY addmult is
      port(    sig1,sig2: in  std_logic_vector(0 to 3);
               result:  out std_logic_vector(0 to 7));
end addmult;


ARCHITECTURE structure of addmult is
      SIGNAL s_add: std_logic_vector(0 to 7);

      COMPONENT add   -- component declaration
               port(op1,op2: in std_logic_vector(0 to 3);
                    ret: out std_logic_vector(0 to 7));
      end component;
                                --component instantiation
                                --name of lower level component
BEGIN                            --key word connecting two levels
      add1: add port map(op1=>sig1, op2=>sig2, ret=>s_add);

      result <= s_add - “00000001”;
end structure;
```

COMPONENT (cont.)

- Top level design contains component declaration

```
COMPONENT add
      port(op1,op2: in std_logic_vector(0 to 3);
           ret: out std_logic_vector(0 to 7));
end component;
```

Component name identical to entity name in low-level design.

- Lower level design of previous example

```
ENTITY add is
      PORT(op1,op2: in std_logic_vector(0 to 3);
          ret:  out std_logic_vector(0 to 7));
end add;
ARCHITECTURE dataflow of add is
BEGIN
      ret <= op1 + op2;
end dataflow;
```

port declaration in component and low-level entity are identical.

- Lower level design can be in a separate file or in the same file as the top-level design