Operators

- Logical operators
- Relational operators
- Shift Operators
- Adding operators
- Multiplying operators
- Miscellaneous operators

Logical Operators

- 6 Logical operators
  - And
  - Or
  - Nand
  - Nor
  - Xor
  - Not(highest precedence)

Relational Operators

- 6 Relational operators

  - =(equal)
  - /=(not equal)
  - <
  - >
  - <=
  - >=

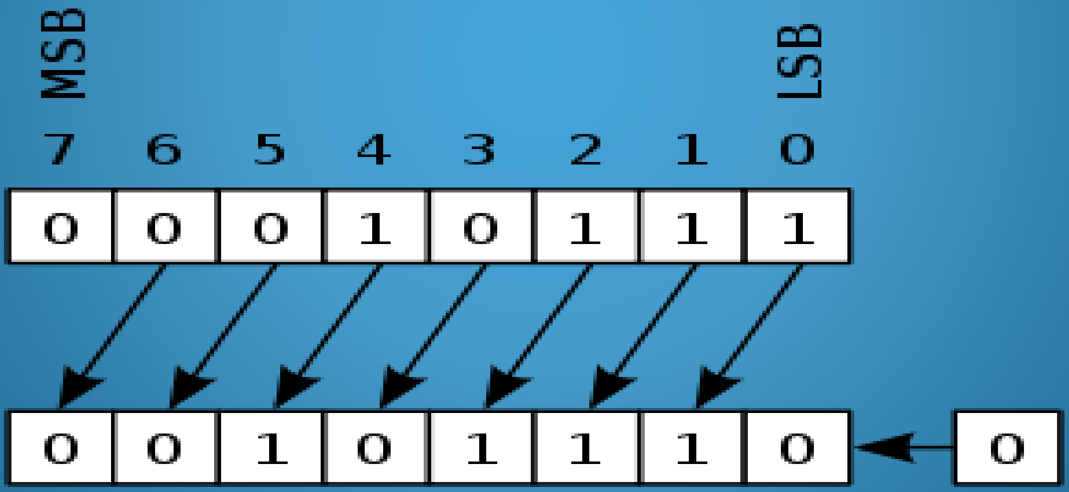
Note: Used for Comparision.

Shift Operators

- 6 shift operators
  - sll- shift left logical
  - srl- shift right logical
  - sla – shift left arithmetic
  - sra – shift right arithmetic.
  - rol- rotate left
  - ror – rotate right

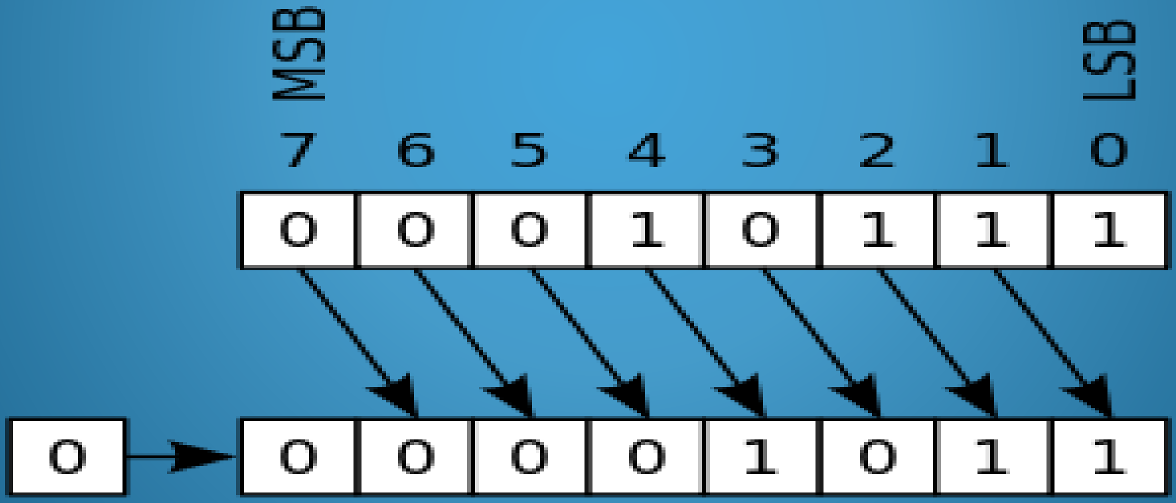
Shift Operators

- Sll- shift left logical



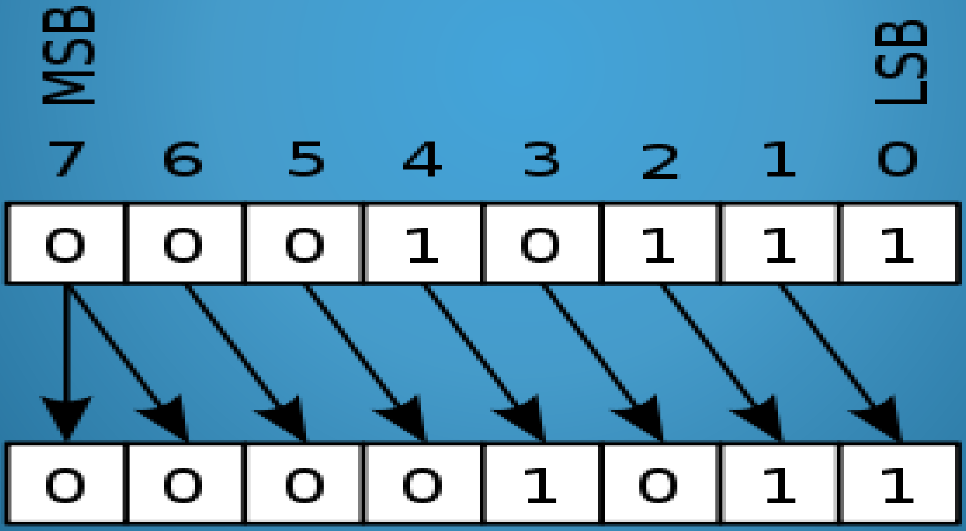
Shift Operators

- Srl- shift right logical



Shift Operators

- Sla is similar to that of sll.
- Sra – shift right Arithmetic



Shift Operators

- ROL – rotate left
  - 10010011 rol 1 = 00100111
- ROR – rotate right
  - 10010011 ror 1 = 11001001

Adding Operators

- Adding operators
  - +
  - -
  - & (‘0’ & ‘1’ -> “01”)

Multiply Operators

- Multiply operators
  - *
  - /
  - Mod
  - Rem

Miscellaneous Operators

- Miscellaneous operators
  - abs
  - ** (exponentiation)

IF-THEN-ELSE Statement

General syntax is:

```
    if (condition_1) then
        begin … statements_1 …
        end;
    elsif (condition_2) then
        begin … statements_2 …
        end;
    else
        begin …statements _i …end;
    end if;
```

Process Statement

- All statements in a process occur sequentially.
- If statements are defined in a process statement.
- Processes have sensitivity list.

```
[process-label]:Process (sensitivity list)
                [process-item declarations]
begin
              sequential statements.
end process;
```

Variable Assignment Statement

```
Variable-object := expression;

Signal A,Z: integer;

pr: process(A)
    variable v1,v2: integer
begin
            v1:=A-v2;
            z<=-v1;
end process pr;
```

Signal Assignment Statement

```
Signal-object <= expression;

Signal A,Z: integer;

pr: process(A)
     variable v1,v2: integer
begin
                v1:=A-v2;
                z<=-v1;
end process pr;
```

Case Statement

Case expression is

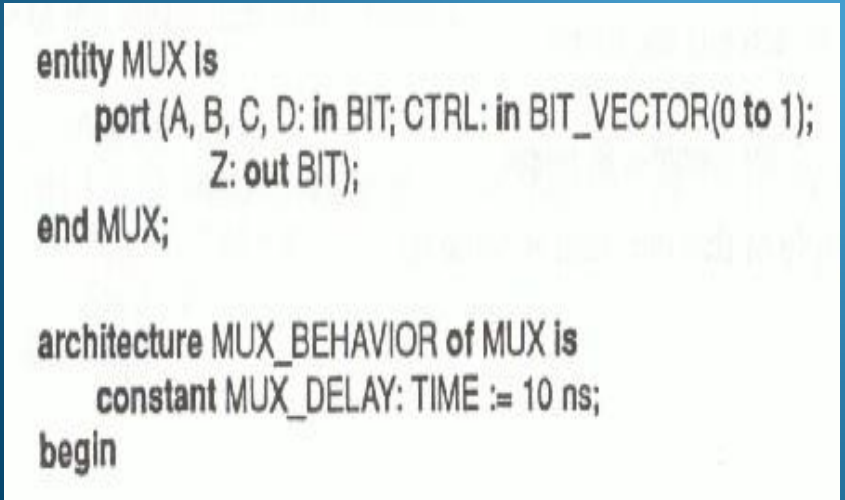
when choices => sequential statements

when choices => sequential statements

[when others => sequential statements]

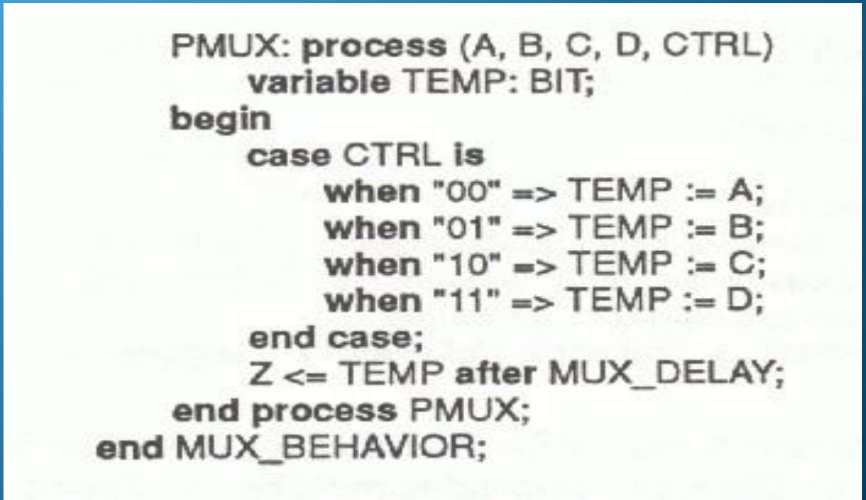
End case;

Example:



```
entity MUX is
    port (A, B, C, D: in BIT; CTRL: in BIT_VECTOR(0 to 1);
            Z: out BIT);
end MUX;

architecture MUX_BEHAVIOR of MUX is
    constant MUX_DELAY: TIME := 10 ns;
begin
```



```
    PMUX: process (A, B, C, D, CTRL)
        variable TEMP: BIT;
    begin
        case CTRL is
            when "00" => TEMP := A;
            when "01" => TEMP := B;
            when "10" => TEMP := C;
            when "11" => TEMP := D;
        end case;
        Z <= TEMP after MUX_DELAY;
    end process PMUX;
end MUX_BEHAVIOR;
```

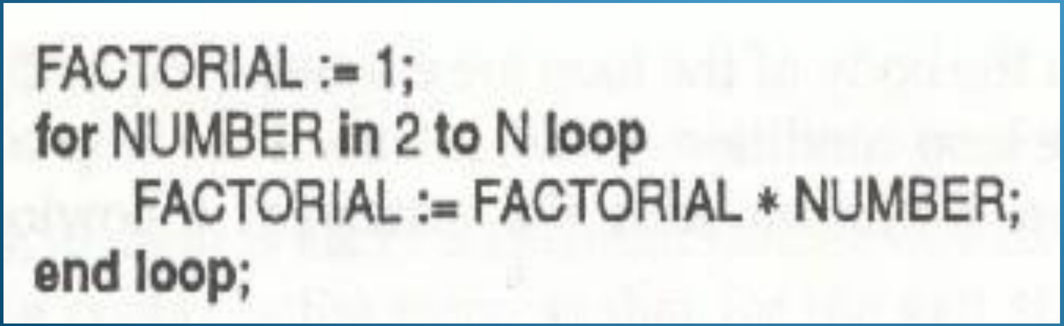
Loop Statement

```
[Loop-Label]: iteration-scheme loop
                        sequential statements
end loop[loop-label];
```

Three types of iteration-scheme:

1st type:

for identifier in range



```
FACTORIAL := 1;
for NUMBER in 2 to N loop
    FACTORIAL := FACTORIAL * NUMBER;
end loop;
```

Loop Statement

[Loop-Label]: iteration-scheme loop

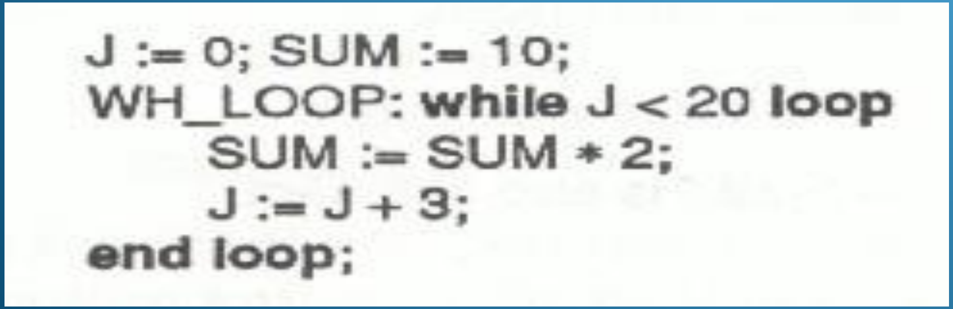
sequential statements

end loop[loop-label];

2nd type of iteration-scheme:

while boolean-expression

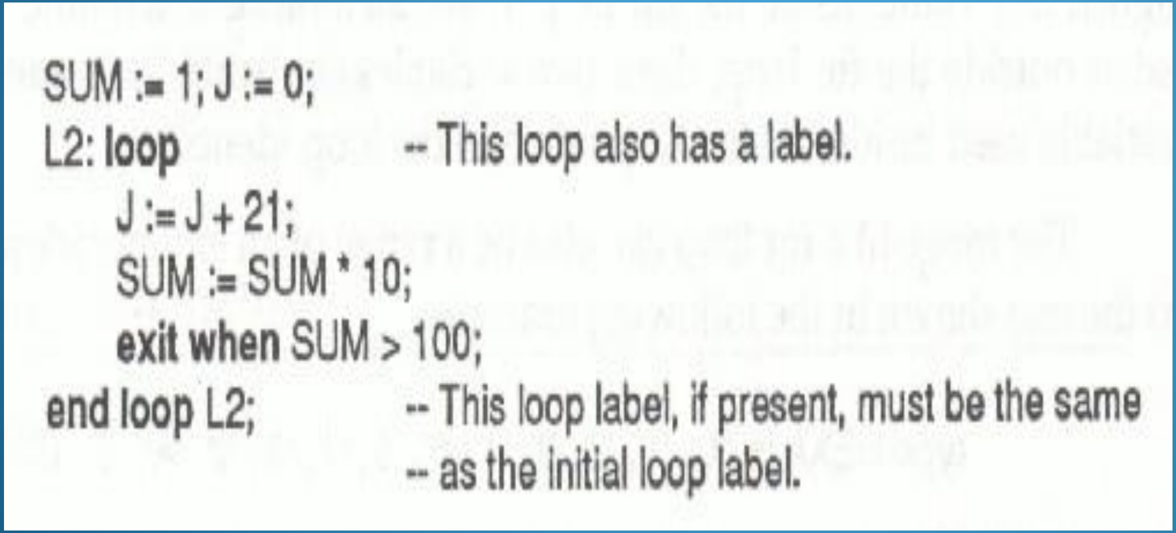
Example:



```
J := 0; SUM := 10;
WH_LOOP: while J < 20 loop
      SUM := SUM * 2;
      J := J + 3;
end loop;
```

Loop Statement

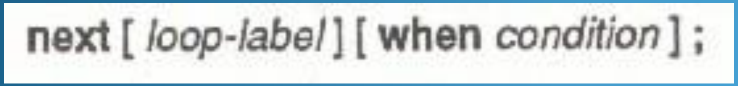
3rd type of iteration-scheme:



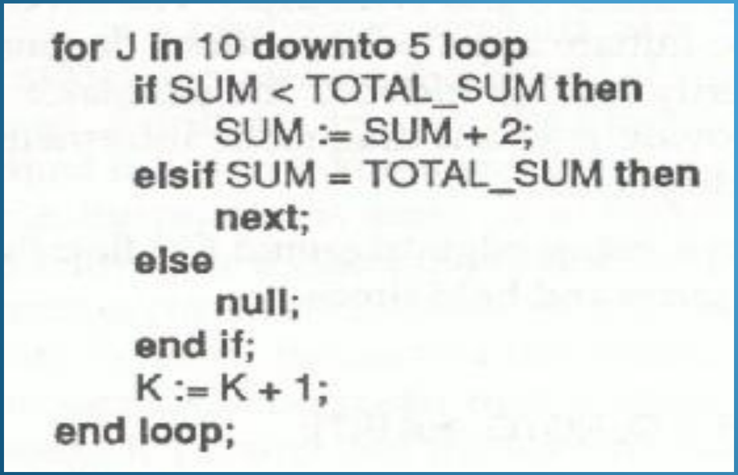
```
SUM := 1; J := 0;
L2: loop            -- This loop also has a label.
      J := J + 21;
      SUM := SUM * 10;
      exit when SUM > 100;
end loop L2;        -- This loop label, if present, must be the same
                    -- as the initial loop label.
```

Next Statement

- Can be used only inside a loop.
- Begins execution back at the beginning of the loop.



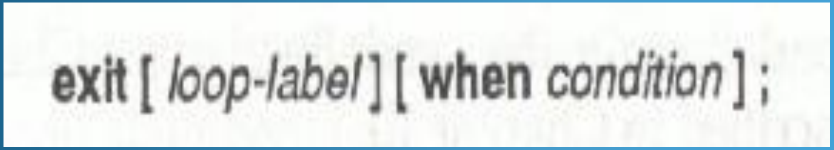
```
next [ loop-label ] [ when condition ] ;
```



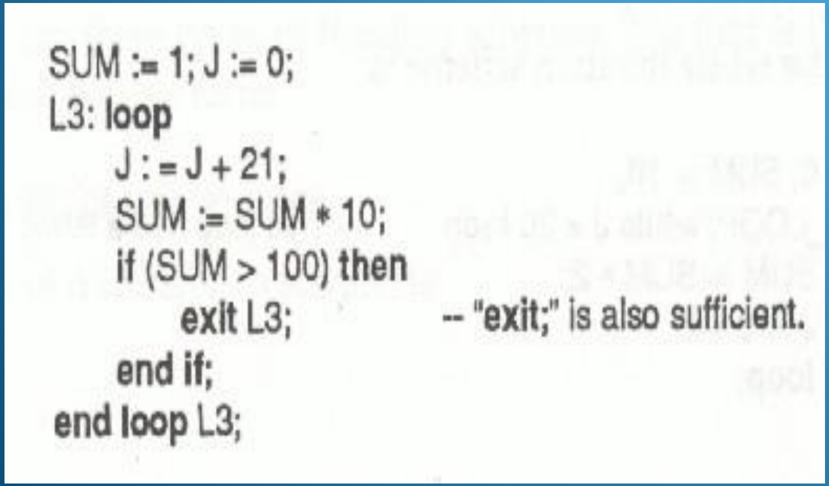
```
for J in 10 downto 5 loop
    if SUM < TOTAL_SUM then
        SUM := SUM + 2;
    elsif SUM = TOTAL_SUM then
        next;
    else
        null;
    end if;
    K := K + 1;
end loop;
```

Exit Statement

Used to jump out of the a loop, cannot be used except inside a loop.



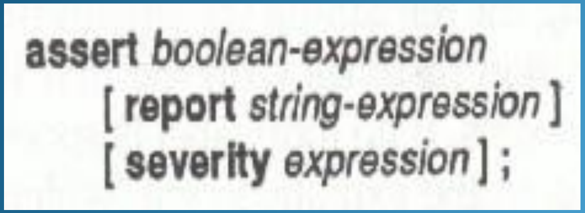
```
exit [ loop-label ] [ when condition ] ;
```



```
SUM := 1; J := 0;
L3: loop
      J : = J + 21;
      SUM := SUM * 10;
      if (SUM > 100) then
            exit L3;              -- "exit;" is also sufficient.
      end if;
end loop L3;
```

Assertion Statement

- Used for modeling constraints of an entity.



```
assert boolean-expression
      [ report string-expression ]
      [ severity expression ] ;
```

- Report statement can be used to generate a message.

  E.g: report – “This circuit error”.
- Type severity_level is(note,warning,error, failure).
- It can be included anywhere in a process body.

With..Select Statement

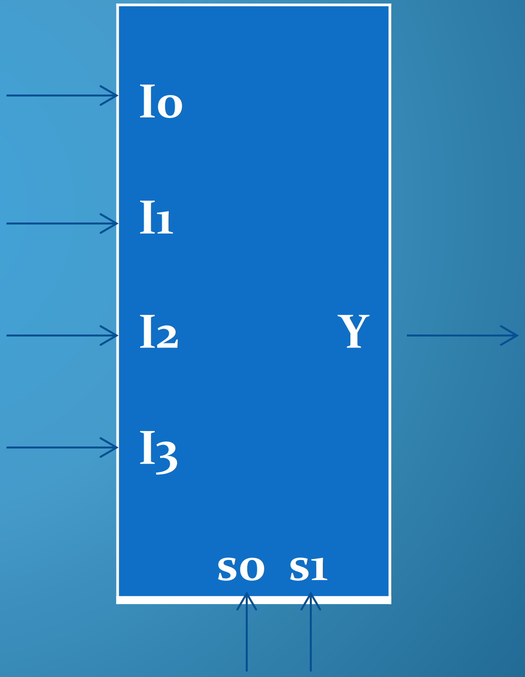
- Based on the select expression, different values are selected for a target signal.
- Syntax:

```
with expression select
        target-signal <= waveform-elements when choices,
                         waveform-elements when choices,
                         …
                         waveform-elements when choices;
```

With..Select Statement

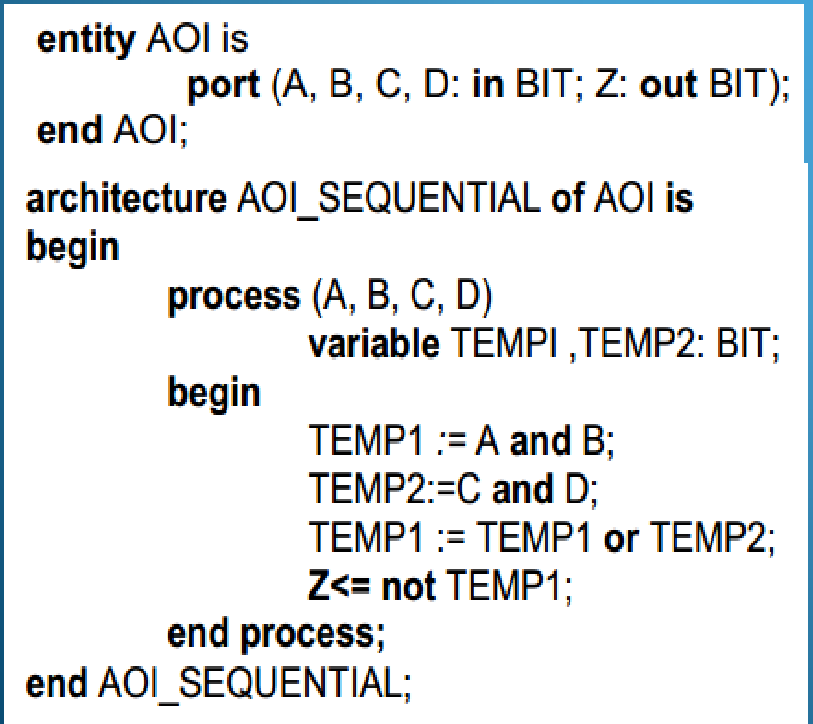
Example:
Write a VHDL code for 4:1 mux.

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
        port(i0,i1,i2,i3: in std_logic;
             sel: in std_logic_vector(1 downto 0);
                y: out std_logic);
end mux;
architecture mux4 of mux is
begin
        with sel select
                y<=i0 when “00”;
                   i1 when “01”;
                   i2 when “10”;
                   i3 when “11”;
                   ‘x’ when others;
end mux;
```

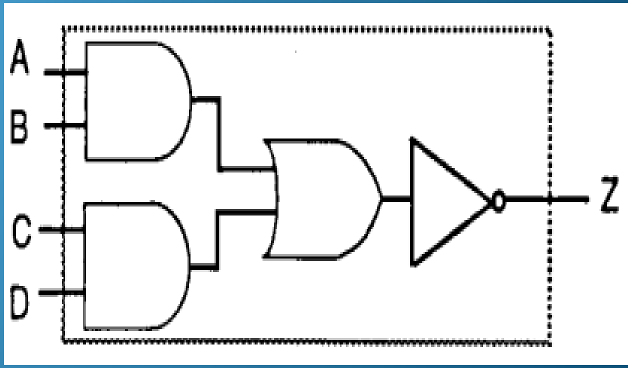


Behavioral Modeling Style

- specifies the behavior of an entity as a set of statements that are executed sequentially in the specified order.

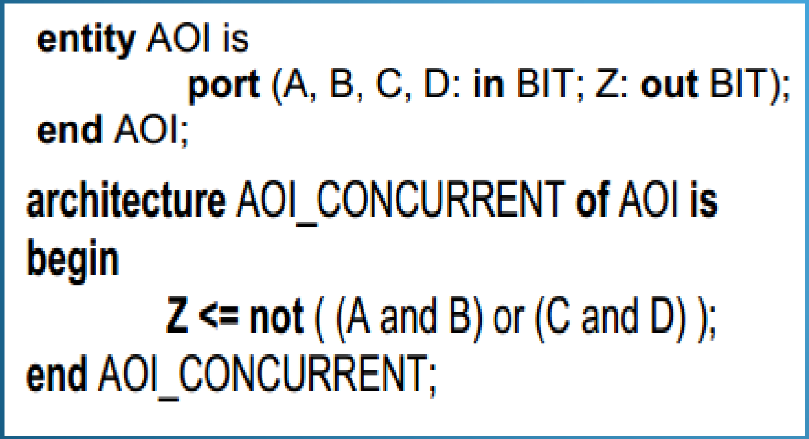


```
entity AOI is
        port (A, B, C, D: in BIT; Z: out BIT);
end AOI;
architecture AOI_SEQUENTIAL of AOI is
begin
        process (A, B, C, D)
                variable TEMPI ,TEMP2: BIT;
        begin
                TEMP1 := A and B;
                TEMP2:=C and D;
                TEMP1 := TEMP1 or TEMP2;
                Z<= not TEMP1;
        end process;
end AOI_SEQUENTIAL;
```

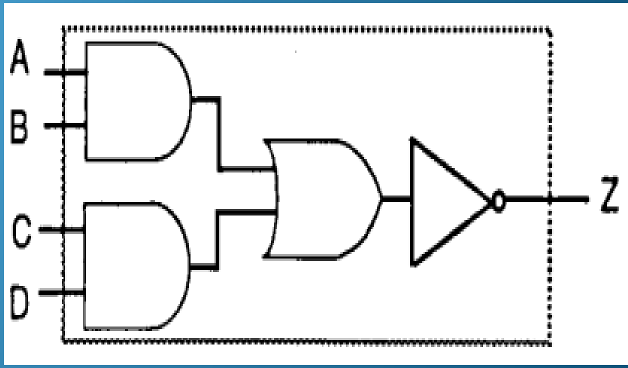


Data Flow Modeling Style

- Flow of data is expressed primarily using concurrent signal assignment statements.



```
entity AOI is
        port (A, B, C, D: in BIT; Z: out BIT);
end AOI;
architecture AOI_CONCURRENT of AOI is
begin
        Z <= not ( (A and B) or (C and D) );
end AOI_CONCURRENT;
```



Structural Statements

Structural VHDL:

- describes the arrangement and interconnection of components.

- Shows interconnects at any level of abstraction.
  - Component instantiation process requires component declarations and component instantiation statements.
  - Component instantiation is one of the building blocks of structural descriptions.
  - Component instantiation declares the interface of the components used in the architecture.
  - Only the interface is visible and the internal components are hidden.
  - The component declaration declares the interface of the component to the architecture.

Structural Statements Contd..

Component Declaration Syntax:

```
component component-name
            port(list-of-interface-ports);
end component;
```

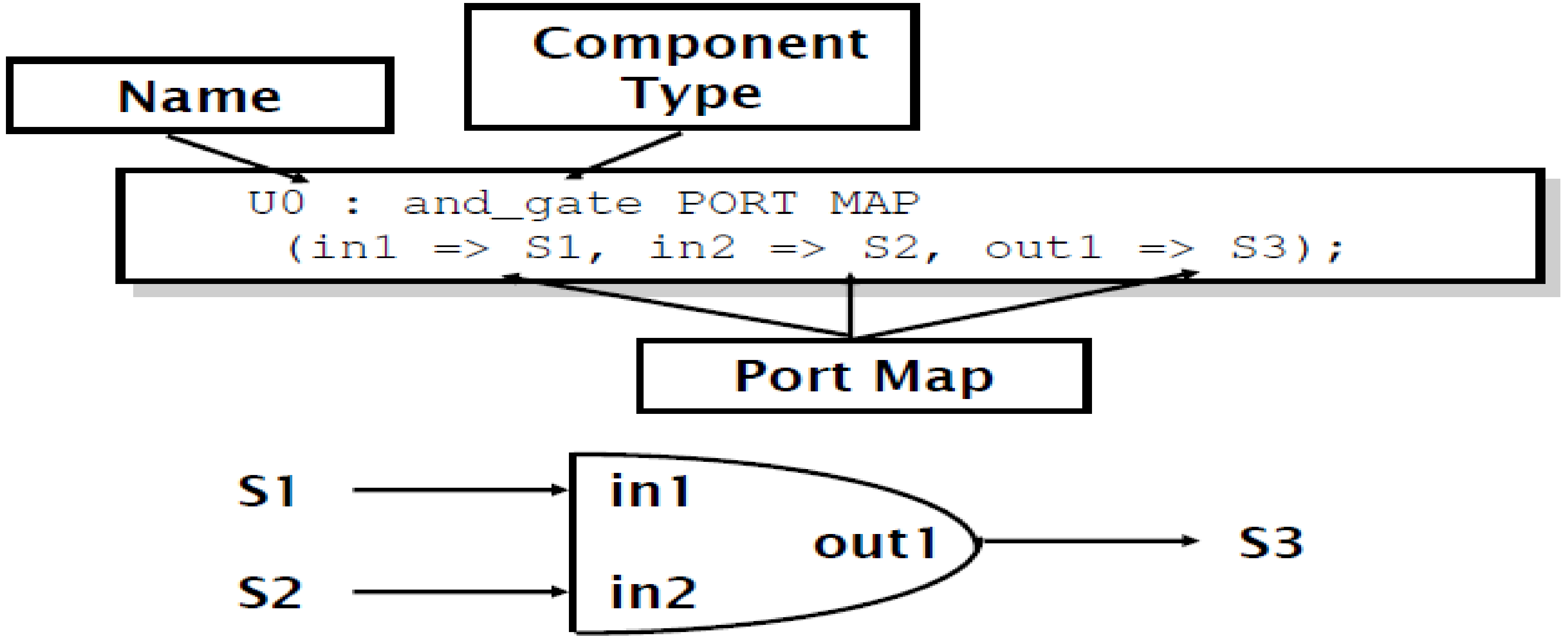
Component Instantiation Syntax:

```
component-label: component-name port map (association-list)
```

Structural Statements

The Instantiation has 3 key parts

- Name
- Component type
- port map

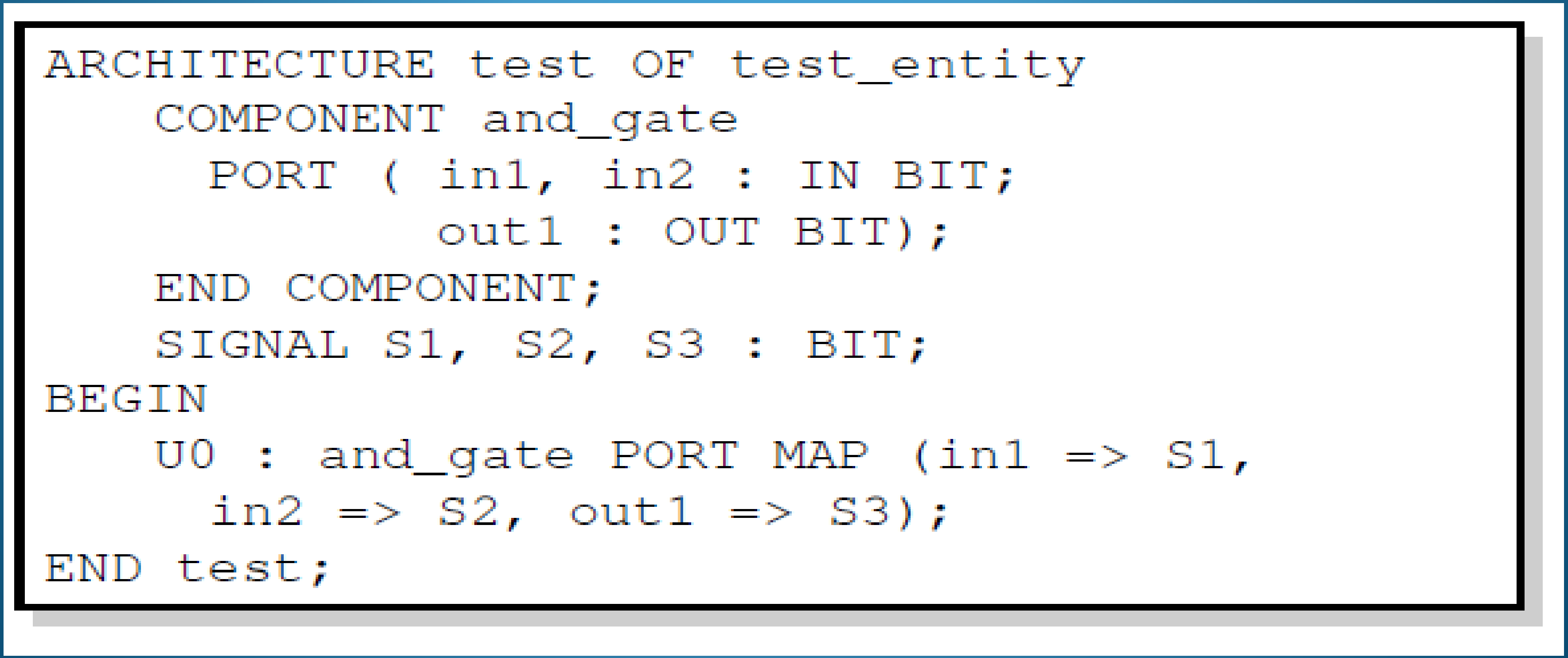


```
U0 : and_gate PORT MAP
  (in1 => S1, in2 => S2, out1 => S3);
```

Structural Statements

Structural VHDL:

The instantiation statement maps the interface of the component to other objects in the architecture.



```
ARCHITECTURE test OF test_entity
   COMPONENT and_gate
     PORT ( in1, in2 : IN BIT;
                 out1 : OUT BIT);
   END COMPONENT;
   SIGNAL S1, S2, S3 : BIT;
BEGIN
   U0 : and_gate PORT MAP (in1 => S1,
     in2 => S2, out1 => S3);
END test;
```