Unit 2. VHDL and Simulation

2.1 Introduction to VHDL

- Conventional Hardware Specification
  - Truth tables
  - Boolean equations
  - State diagrams
  - Pseudo-code behavioral algorithms
  - Schematic diagrams
  - Netlist, proprietary cad formats
- Advantages
  - Similar to methods in other engineering areas
  - Familiar, graphical
- Disadvantages
  - Too many different method
  - Specification languages typically are not defined in syntax or semantics
  - Specifications are not typically manipulatable

2.1 Introduction to VHDL (cont'd)

- Motivation of HDLs
  - Obtain benefits of an unambiguous, standard specification language
  - Facilitate use of computer-aided design (CAD) and computer-aided engineering tools
  - Facilitate exploration of rapidly improving logic synthesis technology
  - Increase designer efficiency, permit rapid prototyping, reduce time-to market, etc.
- Some Common HDLs
  - ABEL
  - Verilog-HDL(cadence, IEEE 1364)
  - VHDL (US DoD, now IEEE 1164 and 1076

2.1 Introduction to VHDL cont’d: Synthesis Technology

Evolution of synthesis technology

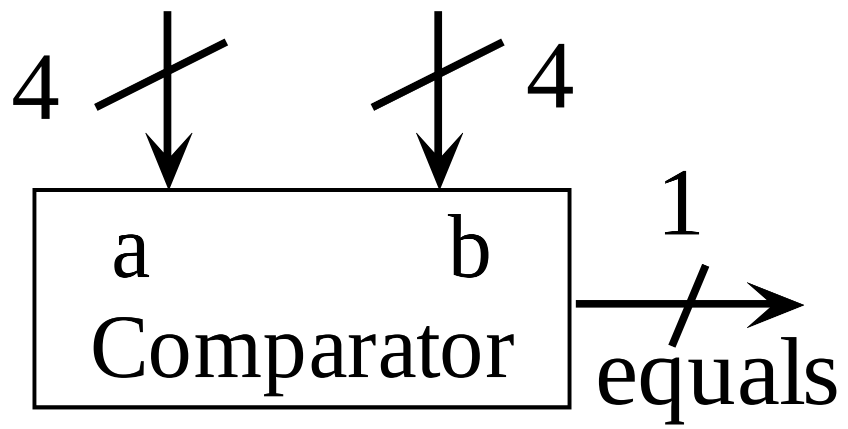
- logic minimization software
- PLA synthesis software
- Multiple-level combination logic synthesis
- Sequential logic synthesis
- Automatic mapping to gate arrays, standard cells, (PLDs), FPGAs
- VHDL and logic synthesis
- VHDL provided a key platform for commercializing logic synthesis technology
- IEEE standard 1076.3 defines a subset of VHDL for use by logic synthesis tools

2.1.1History of VHDL

- Very high speed Integrated Circuit Program
  - The US department of Defense funded the VHSIC program in the 1970’s and 1980’s to promote the improvement of semiconductor technology
  - One product of the VHSIC program was VHDL
- Originals of VHDL
  - To improve documentation of complex hardware designs and thus improve the ability to subcontract the design of military systems
  - To provide a standard modeling and simulation language
- Initial Standards: IEEE 1076 (1987)

2.1.2 Fundamental Concept: A Simple Design

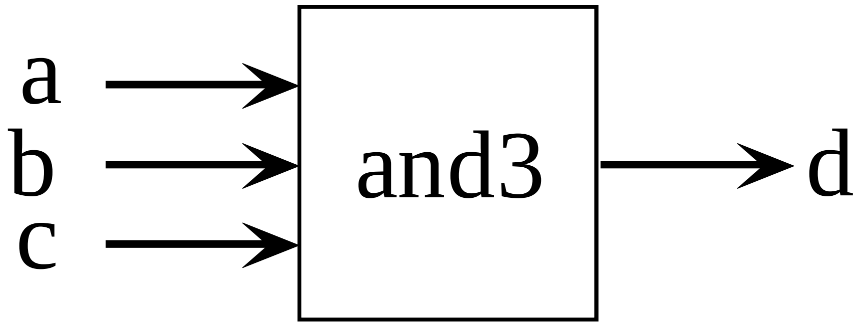
```
- - eqcomp4 is a four bit equality comparator
entity eqcomp4 is
 port ( a, b: in bit_vector(3 downto 0);
        equals: out bit);  -- equals is output
 end entity eqcomp4;

architecture dataflow of eqcomp4 is
  begin
    equals <= ‘1’ when (a=b) else ‘0’;
  end architecture dataflow;
- - end of the program
```



2.1.2 Example 2.1

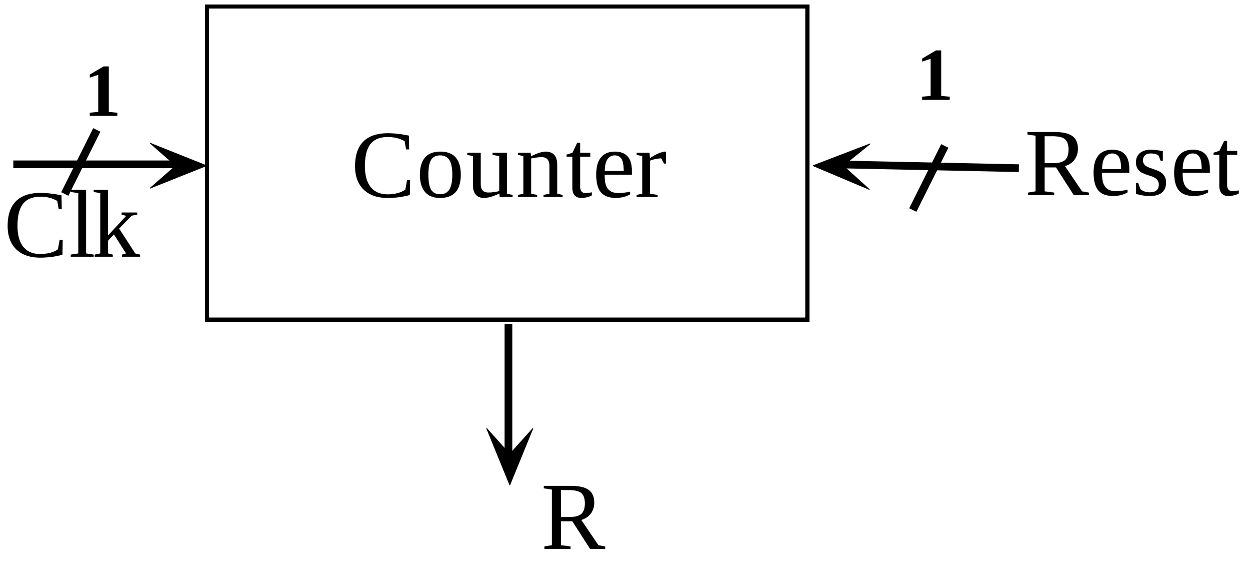
Design an entity of a three input AND gate

```
- - three input and gate
entity and3 is
 port ( a, b, c : in bit;
        d: out bit);  -- d is output
 end entity and3;

architecture structure1 of and3 is
  begin
    d <= a and b and c;
 end architecture structure1;
- - end of the program
```



2.1.2 Example 2.2 : An N-bit Counter

```
Entity Counter is
    Generic (N: Natural);
Port(Clk: in bit;
    reset: in bit;
    R: out natural range 0 to N-1);
End Entity counter;
```

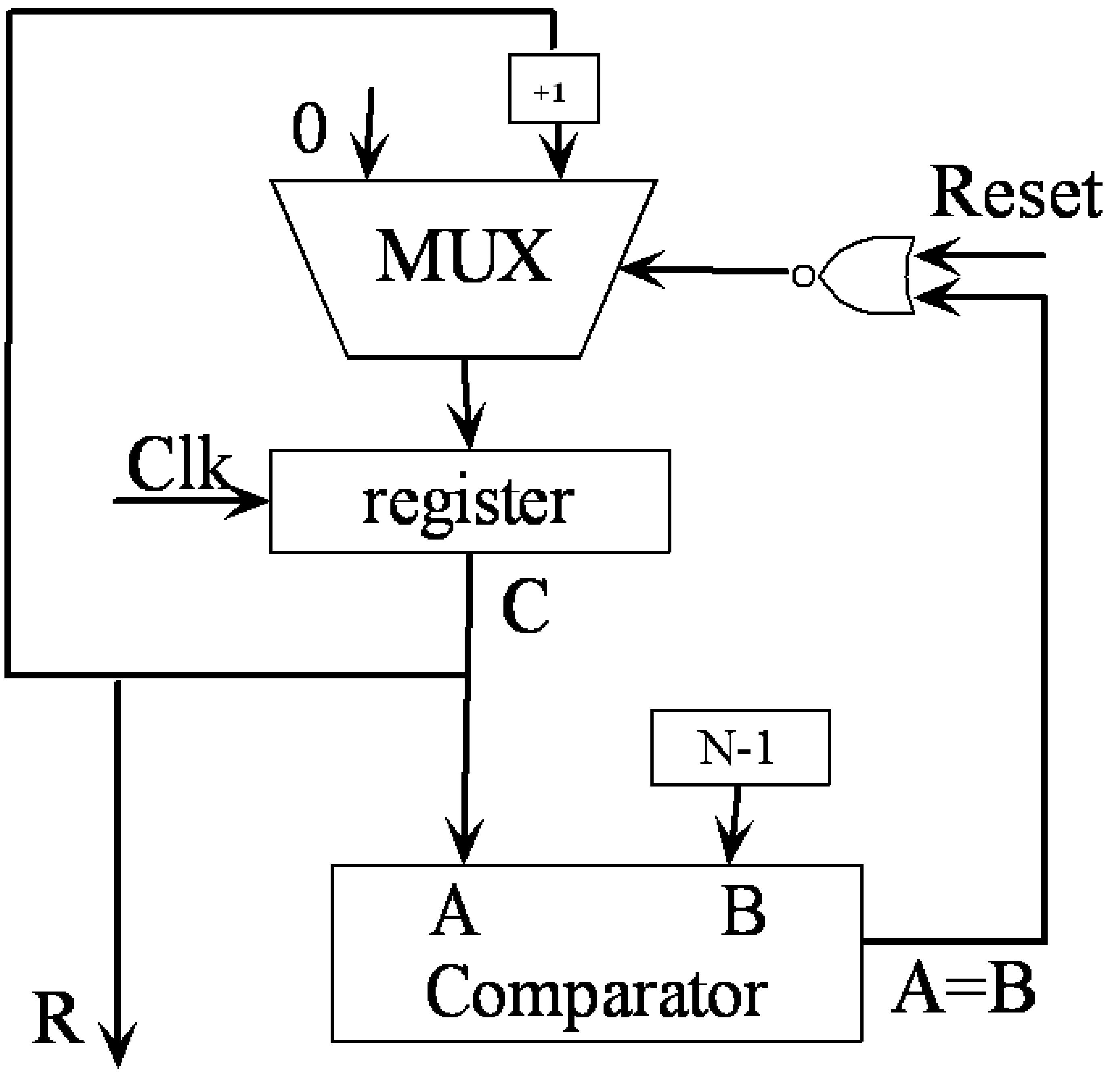


2.1.2 Example 2.2 : An N-bit Counter (cont’d)

Counter with Synch. reset

```
Architecture sync of counter is
    Signal C: Natural range 0 to N-1;
Begin
    R<= C;
    P_count : process (Clk) is
    begin
      If Clk =‘1’ and Clk’event then
        If reset =‘1’ or C=‘N-1’ then
            C<=0; -- clear counter
        Else
            C<=C+1;
        End if;
      End if;
    End process P_count;
End architecture Sync;
```

2.1.2 Example 2.2 : An N-bit Counter (cont'd)



2.1.3 Lexical Elements

- Comments -- comments are important
- Identifiers are a sequence of non-space characters that obey the following rules
  - Every character is either a letter, a digit, or the underscore (_)
  - The first character in the sequence is a letter
  - The sequence contains no adjacent underscores, and the last character is not an underscore

Remarks: VHDL identifiers are case-insensitive

Some examples:

Last@value   5bit_counter   _AO

Clock__pulse   good_one

Extended identifiers: \999\

2.1.3 Lexical Elements (cont’d)

- Reserved words: words or keywords are reserved for special use in VHDL. They can’t be used as identifiers

abs
access
after
alias
all
and
architecture
array
assert
attribute

begin
block
body
buffer
bus

case
component
configuration
constant

disconnect
downto

else
elsif
end
entity
exit

file
for
function

generate
generic
group
guarded

if
impure
in
inertial
inout
is

label
library
linkage
literal
loop

map
mod

nand
new
next
nor
not
null

of
on
open
or
others
out

package
port
postponed
procedure
process
pure

range
record
register
reject
rem
report
return
rol
ror

select
severity
signal
shared
sla
sll
sra
srl
subtype

then
to
transport
type

unaffected
units
until
use

variable

wait
when
while
with

xnor
xor

2.1.3 Lexical Elements (cont'd)

- Special symbols
  - $ ' ( ) * +, - . / : ; < => |
  - => ** := /+ >= <= <>
- Numbers: integer literal and real literal
  - Example 10 0 102 4.13
  - Exponential notation 46E3 1E+12 5e0 3.0e-3
  - Base other than 10
    - Base of 2   2#10000000#
    - Base of 8   8#0.4#, what is this in decimal?
  - Underline as separators:
    - 123_456 3.141_592_6   2#1111_1100_0000#
- Characters
  - 'A' --uppercase letter
  - 'z'   -- lower case letter
  - ','   -- coma
  - ' '   --the separator character space

2.1.3 Lexical Elements (cont’d)

- Strings: a sequence of characters
  - “a string”
  - “we can include and printing characters”
  - “”
  - “string in string ““a string ””. ”
  - “if we can’t write a string in one line”
  - &” then we break it into two lines”
- bit Strings
  - B (base of 2)   B “0101 0011”
  - b“1111_0010”
  - O (base of 8) O“372”
  - X (base of 16) X“FA” what is this ?
  - X “10” what is this one ?

2.1.4 Syntax Description

- Combine lexical elements to form valid VHDL description
- Syntactic category
- Rules of syntax EBNF (Extended Backus-Naur Form)
  - Example of a variable assignment:
  - Variable_assignment <= target:=expression;
  - D0 := 25+6;
- Optional component [ ]
  - Function_call <= name[(association_list)]
- Combine alternatives |
  - Mode <= in|out|inout
  - Example for process statement

```
Process_statement <=
Process is
        {process_item}
        Begin
        {sequential_statement}
End Process;-- will be talked about later on
```

2.2 Scalar Data Types and Operations

2.2.1 Constants, variables, signals

Constant_declarations <=

Constant identifier {,…} : subtype_indication [:= expression];

-- constant have a value that is defined once during initialization, and then remains unchanged.

-- constants in subprograms are recomputed each time the subprogram is called

Examples:

Constant number _of_bytes : integer :=4;

Constant e : real := 2.718;

Constant prop_delay : time := 3 ns;

Variable_declarations <=

variable identifier {,…} : subtype_indication [:= expression];

-- have a value that is updated immediately as a result of an assignment statement.

Examples:

Variable index : integer := 0;

Variable start,finish : time := 0 ns;

2.2.1 Constants, Variables, Signals

Signals:

- used to model hardware signal conductors
- information is communicated between design components only via signals.
- signals can have fixed links to other signals.
- signals have a present value, as well as a sequence of past and projected future values (variables only have a present value).
- signal values are scheduled to be changed by means of assignment statements:
- A<= new_constant_value;

2.2.1 Variable Versus signals?

Variable and signals are easily confused at first.

- Both signals and variables can be assigned values (also, a signal can be assigned the value of a variable, and vice versa)

Differences between the variables & signals

- Signal correspond to physical signals associated with conductors or busses.
- Variables are a convenience for more easily describing algorithms that might be used in process and subprograms. There is not necessarily any hardware associated with a variable.
- A variable's value can be changed immediately as a result of an assignment statement (which must use the := symbol).
- A signal's value can be changed no sooner than the beginning of the next simulation cycle. The <= symbol must be used.

2.2.2 Scalar types

Type declarations

Type_declaration <= **type** identifiers **is** type_definition;

Example:

Type apples **is range** 0 **to** 100;

Example 2.2 :

Package int_types **is**

  type small_int **is range** 0 **to** 255;

End package int_types;

Use work.int_types.all;

Entity small_adder **is**

  port(a,b: **in** small_int; s: **out** small_int);

End entity small_adder;

2.2.2 Scalar types (cont'd)

Integer Types

Integer _Type_declaration <=

type identifiers **is range** expression (**to** | **downto)** expression;

Example:

Type day_of_month **is range** 0 **to** 31;

Declare variable of this type:

Variable today: day_of_month := 9;

Floating Types …

Arithmetic operations:

- + - * /
- Mod rem abs **

2.2.2 Scalar types (cont'd)

Physical Types

Physical_type_definition <=

type identifiers **is range** expression (**to** | **downto)** expression

Units

Identifier;

{identifier=physical_literal;}

End units [identifier]

Example:

Type resistance **is range** 0 **to** 1E9

Units

Ohm;

kohm = 1000 ohm;

Mohm = 1000 kohm;

End units resistance;

Declare variable of this type:

Variable R1: resistance := 900 ohm;

2.2.2 Scalar types (cont’d)

Enumeration Types:

Type water level **is** (too_low, low, high);

Characters …

Boolean types

Type boolean **is** (false, true);

Bits

Type bit **is** (‘0’,’1’);

Standard Logic

```
Type std_ulogic is (‘U’,  --uninitilized
                    ‘X’, --forcing unkown
                    ‘0’, --zero
                    ‘1’, --one
                    ‘Z’, --high impedance
                    ‘W’, -- weak unkown
                    ‘L’, -- weak zero
                    ‘H’, -- weak one
                    ‘-’); --don’t care
```

Sub types

Subtype small_int **is** integer **range** –128 **to** 127

2.2.2 Scalar types (cont'd)

Type qualification

Type Logic_level **is** (unkown, low, undriven, high);

Type system_state **is** (unkown, ready, busy);

To distinguish between common unknown: Use logic_level'(unkown) and

system_state'(unkown)

Type conversion:

real(123)   integer(12.4)

Attributes of Scalar types:

- T'left – first(leftmost) value in T
- T'right last(rightmost) value in T
- T'low
- T'high
- T'ascending True if T is an ascending
- T'image(x)
- T'value(s)

2.2.2 Scalar Types (cont'd)

- VHDL is a strongly typed language
- every object has a unique type.
- objects of different types cannot be mixed together in expressions.
- object typing can be determined statically
- the type of every object must be clear from the VHDL program before any simulation has taken place.
- the types of object must be declared explicitly in all program scopes.

2.3 Sequential Statement

2.3.1 if statement

2.3.2 case statement

2.3.3 Null statement

2.3.4 loop statement

2.3.5 assertion and report statements

2.3.1 If Statement

Syntax rule

If_statement <=

[if_label:]

If boolean_expression **then**

{sequential_statement}

{**elsif** boolean_expression **then**

{sequential_statement}}

[**else**

{sequential_statement}]

End if [if_label];

Example for If_statement <=

If (count =“00”) **then**

a<=b;

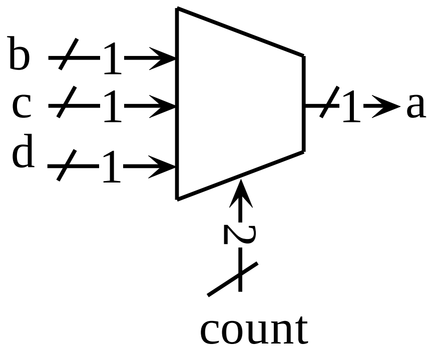
Elsif (count =“10”) **then**

a<=c;

Else

a<=d;

End if ;



2.3.2 Case Statement

Syntax rule

case_statement <=

[case_label:]

case expression **is**

(**when** choices => {sequential_statement})

{…}

End case [case_label];

Example for case_statement <=

case count **is**

When “00” =>

a<=b ;

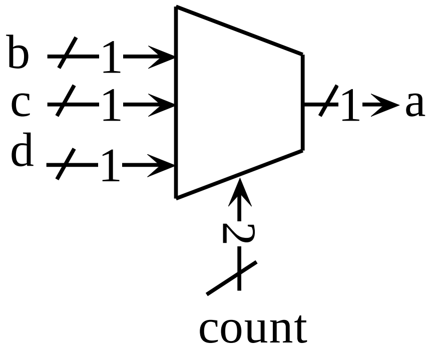
When “10” =>

a<=c ;

When others =>

a<=d ;

End case ;

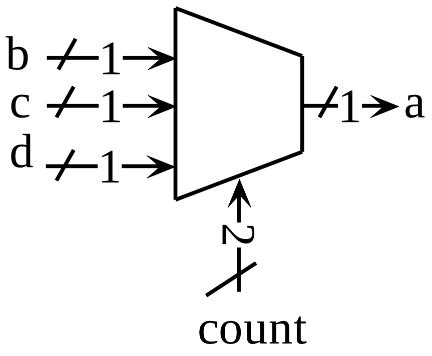


2.3.3 Null Statement

Null_statement <= [label:] null;

Example

```
case count is
    When "00" =>
          a<=b ;
    When "10" =>
          a<=c ;
    When "01" =>
          a<=d ;
    When "11" =>
          null ;
    End case ;
```



2.3.4 Loop Statement

Infinite loop:

```
Loop_statement <=
        [loop_label:]
            **loop**
          {sequential_statement}
            End loop [loop_label] ;
```

Example: **Loop**

```
            wait until clk ='1';
            count <= count_value;
        end loop;
```

While loop:

```
Loop_statement <=
        [loop_label:]
     while condition loop
          {sequential_statement}
     End loop [loop_label] ;
```

2.3.4 Loop Statement (cont'd)

Example for While loop:

```
n := 1;
Sum := 0;
 while n <100 loop
        n := n+1;
        Sum := sum +n;
  End loop ;
```

For loop:

```
Loop_statement <=
      [loop_label:]
   For identifiers in discrete range loop
        {sequential_statement}
    End loop [loop_label] ;
```

Example for the for loop

```
    For n in 1 to 100 loop
        Sum := sum +n;
     End loop ;
```

2.3.4 Loop Statement (cont’d)

Exit statement <=

[label:] exit [loop_label] [when boolean_expression];

```
Loop
        wait until clk ='1' or reset ='1';
        Exit when reset = '1';
        count <= count_value;
end loop;
```

NEXT statement

```
Loop                                Loop
  statement 1;                        statement 1;
    Next when condition               If not condition then
  Statement 2;                              Statement 2;
End loop;                              End if;
                                    End loop;
```

2.3.5 Assertion and Report Statement

Assertion_statement <=

[label:] **assert** boolean_expression [**report** expression] [**severity** expression];

assert initial_value <= max_value

report “ initial value too large”

2.4 Composite Data Types and Operations

Array types

Type BIT **is range** 0 to 1;

Type word **is array** (31 **downto** 0) of bit;

Example:

Signal MEM_BUS: WORD;**--** will be defined later

MEM_BUS(0) <= 0 ;

MEM_BUS(1) <=0 ;

MEM_BUS(2) <=1 ;

Records

Type time_stamp **is record**

    seconds: integer **range 0** to 59;

    minutes: integer **range 0** to 59;

    hours : integer **range 0** to 23;

End record time_stamp;

Variable sample_time, current_time: time_stamp;

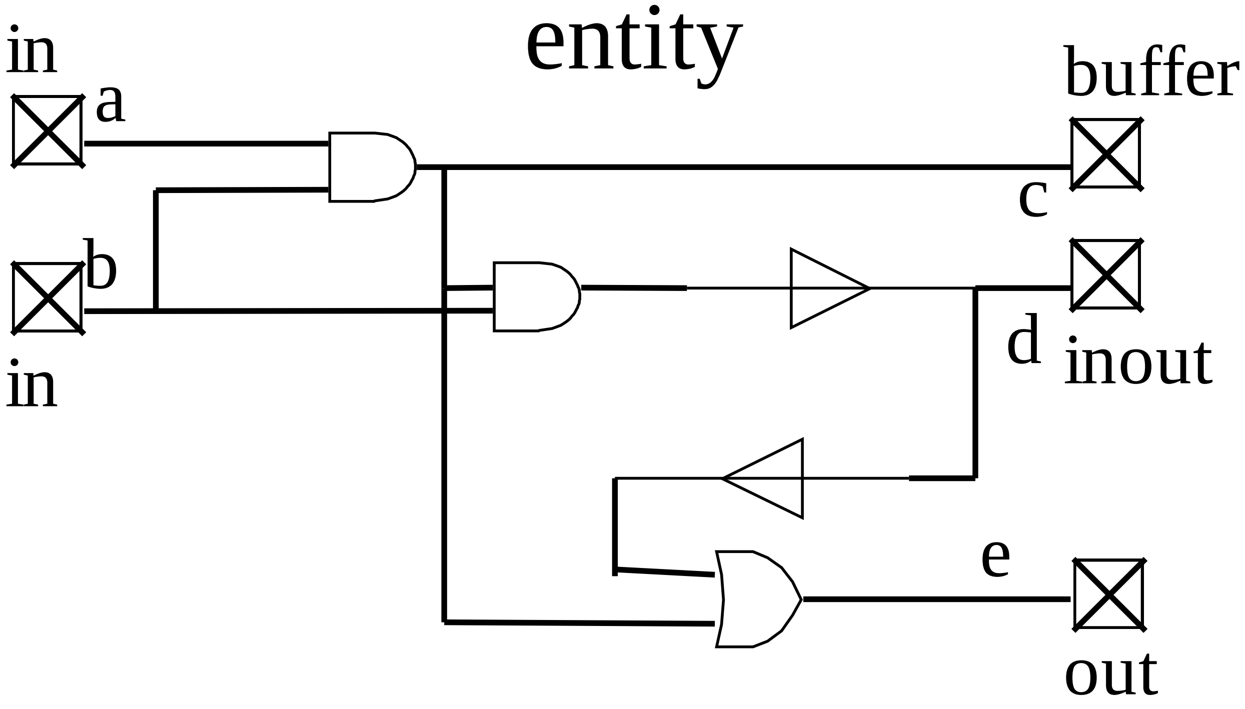
Current_time.seconds := 30;

Current_time.hours := 13;

2.5 Modeling Constructs

- VHDL inherited many modularity ideas form the DoD software language ADA
- Hardware specifications are composed of five kinds of design units:
  - **Entities**
  - **Architectures**
  - **Configurations**
  - **Packages**
  - **Package bodies**
- Design units are provided to the VHDL simulation and/or synthesis environment in source files;
- Design units can also be included from libraries of pre-designed data types, signal types, signal type conversions, components etc.

2.5.1 Modeling Constructs: entity



Entity block **is**

Port (a, b: **in** bit;

c: **buffe**r bit;

d: **inout** bit;

e: **out** bit);

End entity block;

buffer can be used for all output signals

2.5.2 Modeling Constructs: Architecture Bodies

Architecture_body <=

Architecture identifier **of** entity_name **is**

{block_declaration}

Begin

{ concurrent_statement}

End [architecture][identifier];

Example:

```
Entity adder is
Port (a: in word;
      b: in word;
      sum: out word);
End entity adder;
```

```
Architecture ad1 of adder is
Begin
Add_a_b: process(a,b) is
Begin
   sum <= a+b;
End process add_a_b;
End architecture ad1;
```

Signal declarations

Signal_declaration <=

Signal identifier {…} : subtype_indication [:= expression]

2.5.3 Two Main Levels of VHDL Specification

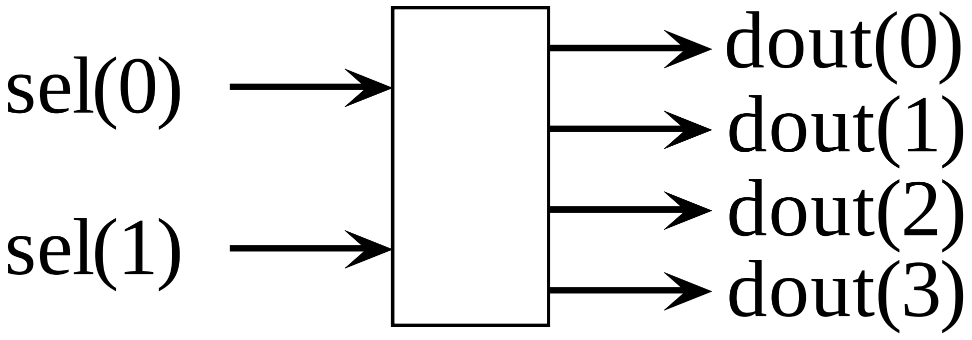
1) Behavior level:

- What is the system supposed to do?
- Components described using algorithms that do not necessarily reflect the actual hardware structure of likely implementations.
- Signal don’t necessary need to be binary values. Data types can be chosen to facilitate high-level description

2) Structure level:

- What is the structure of an implementation?
- Design specified using realizable components
- Binary representation of data types and signals are used.

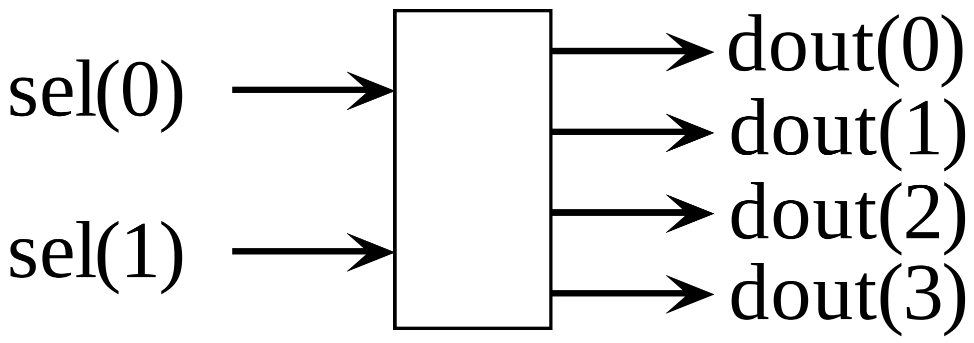
Example 2-to 4 Decoder



VHDL entity for the decoder

```
Entity decoder is
 port ( sel : in bit_vector (1 downto 0);
        dout : out bit_vector (3 downto 0));
 constant delay : time := 5 ns;
end entity decoder;
```

Behavior-level architecture in VHDL



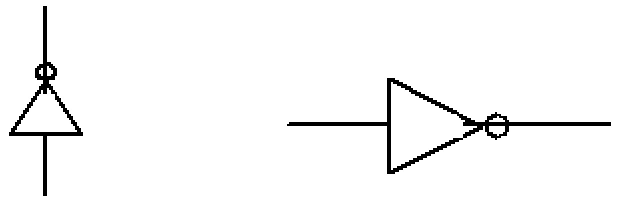
Behavior-level architecture in VHDL

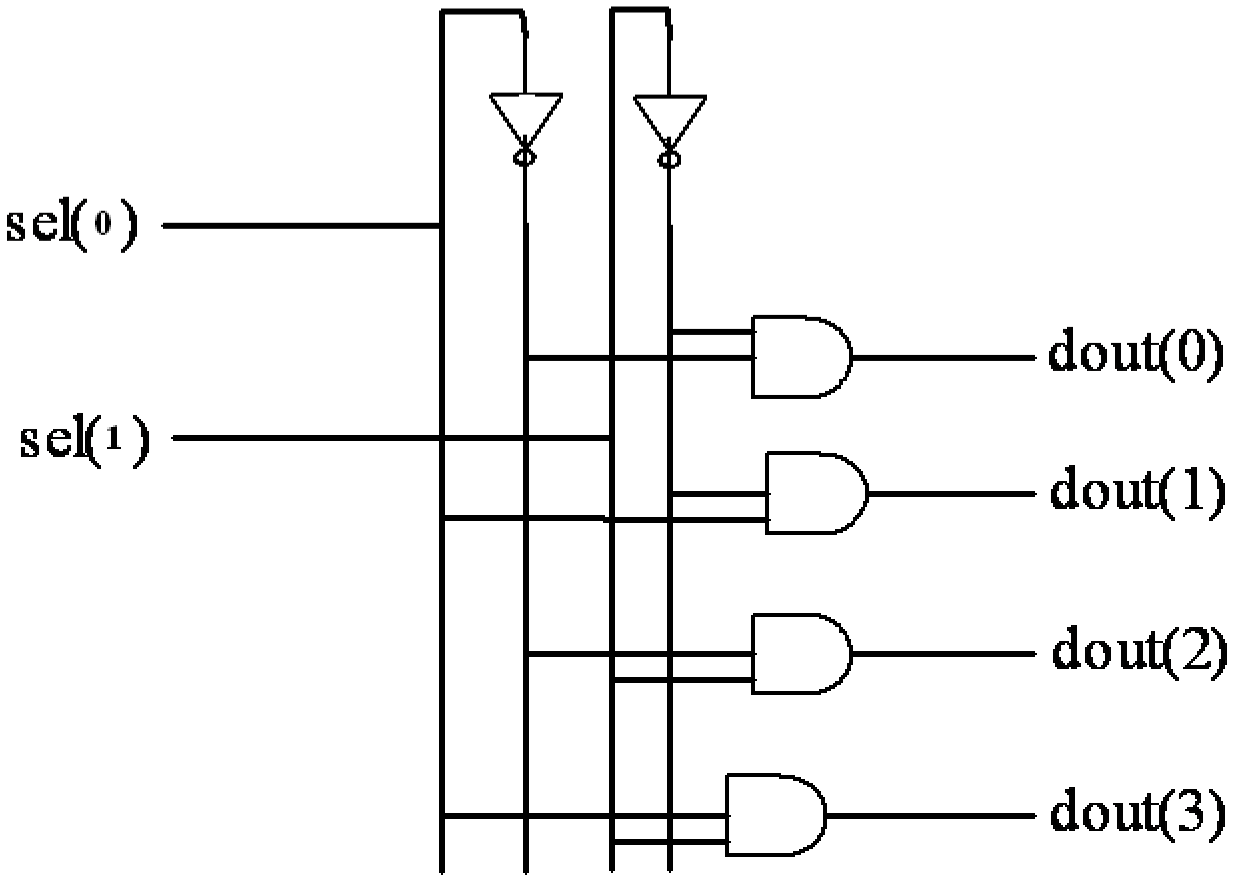
```
Architecture behavior1 of decoder is
 begin
with sel  select
dout <=
   "0001" after delay when "00",
   "0010" after delay when "01",
   "0100" after delay when "10",
   "1000" after delay when "11",
 end behavior1 ;
```

Structure-level architecture

```
Architecture structure1 of decoder is
   component and2 –pre-defined part type
       Port ( I1, I2 : in bit; O1 out bit);
   End component;
 component inverter –pre-defined part type
       Port ( I1 : in bit; O1 out bit);
   End component;
Signal sel_bar: bit_vector (1 downto 0);
Begin
    inv_0: inverter port map (I1=>sel(0),
    O1=>sel_bar(0));
    inv_1: inverter port map (I1=>sel(1),
    O1=>sel_bar(1));
   and_0:and2
    port map (I1=>sel_bar(0), I2=>sel_bar(1),
    O1=>dout(0));
   and_1:and2
     port map (I1=>sel(0), I2=>sel_bar(1), O1=>dout(1));
   and_2:and2
     port map (I1=>sel_bar(0), I2=>sel(1), O1=>dout(2));
   and_3:and2
    port map (I1=>sel(0), I2=>sel(1), O1=>dout(3));
End structure1 ;
```

Structure-level schematic





2.5.4 Modeling Constructs

Signal assignment

Signal_assignment_statement <=

[label:] name <= [delay] waveform;

Waveform <= (value_expression [**after** time_expression]) {…}

y <= a or b after 5 ns;

Wait statement <=

[label:] **wait** [**on** signal name {…}]
[**until** boolean_expression]
[**for** time_expression];

2.5.4 Modeling Constructs (cont’d): signal attributes

S’delayed(T)

- if T>0, then a signal is returned that is identical to S delayed by time T. If T=0 (or is absent), then S is returned delayed by time delta.

S’stable(T)

- if T>0, then a signal is returned that has value TRUE if S has not changed for the past time T; at other times the signal has value FALSE. If T=0 (or is absent), then the signal will be FALSE during a simulation cycle when S changes values; otherwise the signal is TRUE.

S’quiet(T)

- if T>0, then a signal is returned that has value TRUE if S has not been updated for the past time T; at other times the signal has value FALSE. If T=0 (or is absent), then the signal will be FALSE during a simulation cycle when S is updated; otherwise the signal is TRUE.

2.5.4 Modeling Constructs (cont’d): signal attributes

S’active(T)

- Boolean that is true if signal S has been updated during the current simulation cycle

S’event

- Boolean that is true if signal S has changed value during the current simulation cycle

S’LAST_EVENT

- The amount of time elapsed since signal S last changed value.

S’LAST_ACTIVE

- The amount of time elapsed since signal S was last updated.

S’LAST_VALUE

- The value of signal S before the last time that signal S changed values.

2.5.4 Modeling Constructs (cont’d)

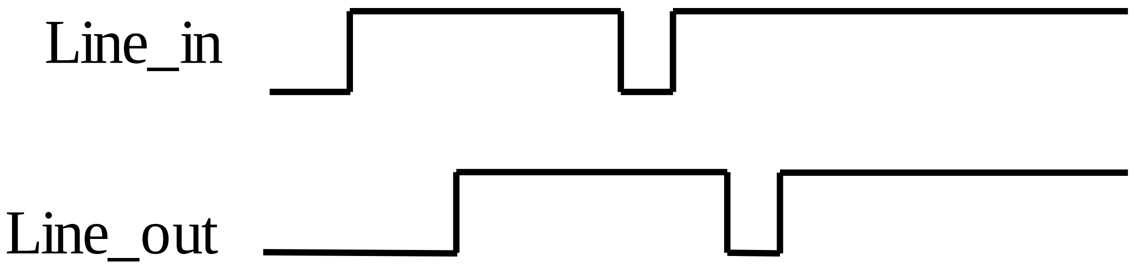
Delay_mechanism<=

transport | [**reject** time_expression]
inertial

Example for transport:

Line_out <= transport line_in after 3 ns;

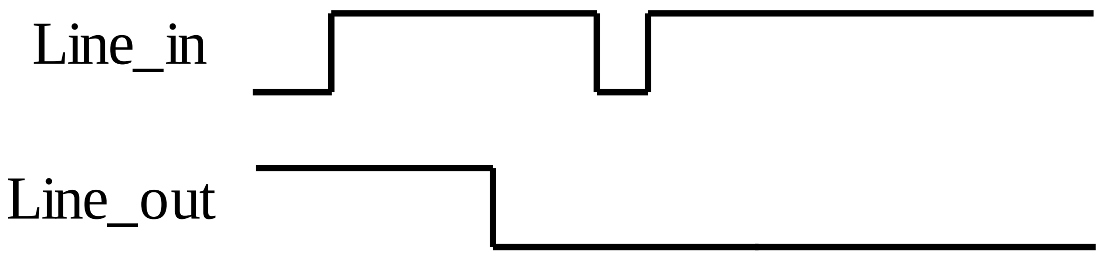
Remarks: the output is shift by the time delay



Example for inertial delay:

Line_out <= inertial not line_in after 3 ns;

- Remarks: if a signal would produce an output pulse shorter than the propagation delay, the the output pulse does not happen



2.5.4 Modeling Constructs (cont’d)

Example for both inertial and reject

Line_out <= reject 2 ns inertial not line_in after 3 ns;

Remarks: if a signal would produce an output pulse shorter than the reject limit delay, the the output pulse does not happen

Process statements <=

[process_label:]

Process [(signal_name{…})] [**is**]

{ process_item}

Begin

{sequential_statement}

End process [process_label]

2.5.5 Modeling Concurrency

- In real digital hardware, components all operate at the same time and signals are updated in parallel.
- How to model concurrency/parallel in VHDL
- Components models are decomposed into **processes** that execute in parallel
- Different signals have values that change in parallel over time
- VHDL provides the ability to specify times in the future when signals will be updated.
- VHDL provides the ability to specify synchronization points, when the values of a group of signals are examined and/or updated for the same time instant.

2.5.5 Modeling Concurrency(cont'd)

VHDL processes can be used for concurrent statement

Example :

```
Proc1: process(A,B) [is]
Begin
    C<= A or B after 5 ns;
End process;
```

Another example:

```
Proc2: process(A,B)
Begin
    C<= A or B;
    Wait on A, B;
End process;
```

2.5.5 Modeling Concurrency(cont’d)

- A VHDL process can be thought of as a sub-program that is called once at the beginning of the simulation
- All VHDL processes execute in parallel
- When the simulation starts, each process begins executing statements following the **begin** statement
- Execution is suspended when the next wait statement is encountered
- **Wait;** - suspends process forever
- **Wait on** signal_list;
- **Wait until** condition;
- **Wait for** time_value;
- Once the end process statement is encountered, execution returns to the statement following the begin statement.

2.5.5 Modeling Concurrency(cont’d): Concurrent Statements

Sequence of Boolean equations:

```
F <= a nor b nor c;
D <= a and b and c;
E  <= a nor b or c;
```

When-else conditional signal assignment:

```
Architecture example of fsm is
…
With state select
X<= “0000” when s0|s1
“0010” when s2|s3;
Y when s4;
Z when others;
End example;
```

2.5.5 Modeling Concurrency(cont’d): Concurrent Statements

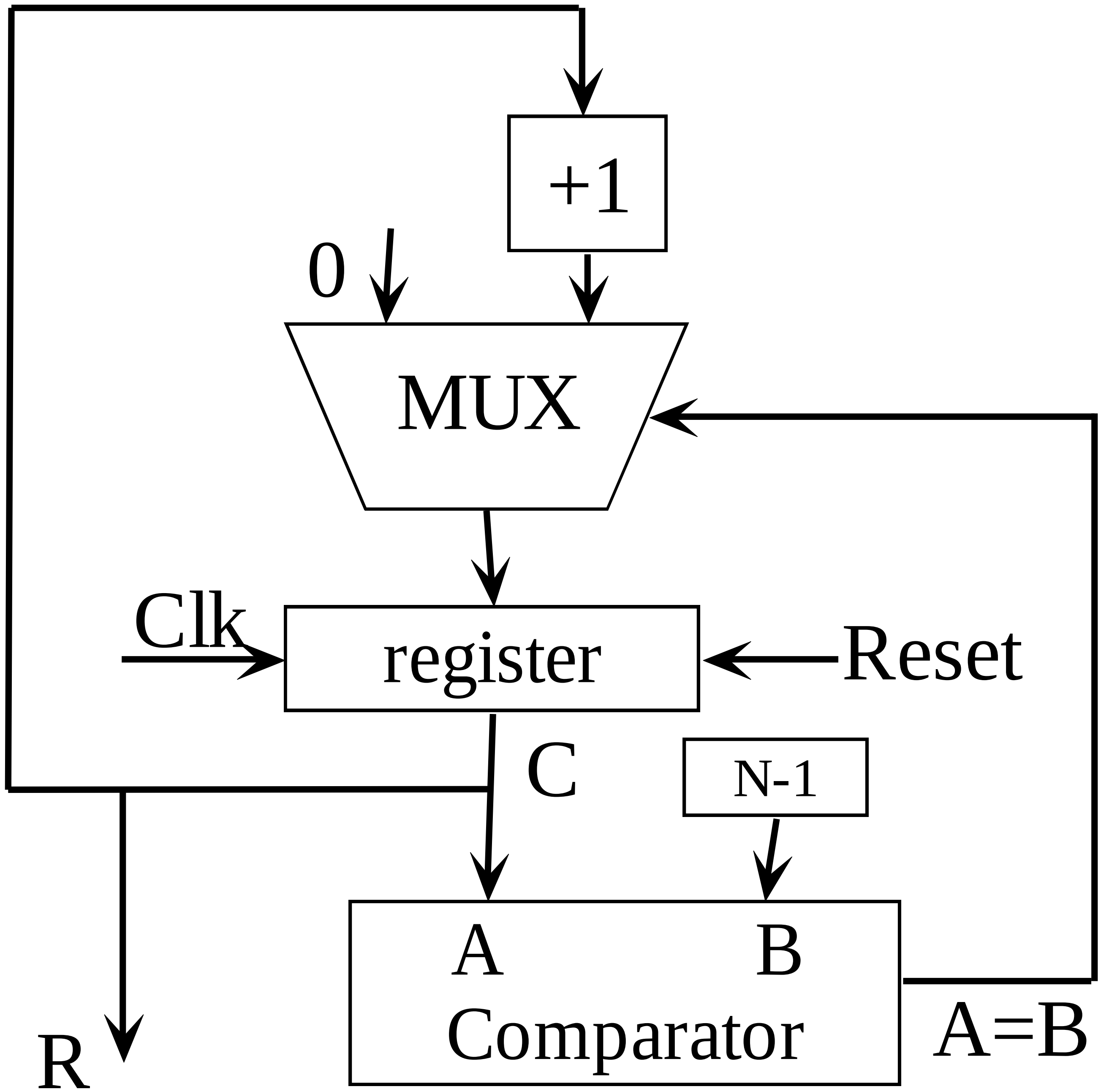
Multiple assignment using Generate:

```
g1: for j in 0 to 2 generate
    a(j) <= b(j) or c(j);
    End generate g1;

g2: c(1) <=c(0) and a(1);
For k in 2 to 20 generate
    c(k) <= c(k-1) and a(k);
    End generate g2;

g3: For l in 0 to 8 generate
    Reg1: register9 port map (clk, reset, enable, d_in(l),
    d_out(l));
    End generate g3;
```

2.5.6 Example : Counter with asyn. reset

```
Entity Counter is
    Generic (N: Natural);
Port(Clk: in bit;
        reset: in bit;
    R: out natural range 0 to N-1);
End counter;
Architecture Async of counter is
Signal C: Natural range 0 to N-1;
Begin
R<= C;
P_count : process (Clk, reset)
begin
 If reset =‘1’ then
    C <=0; -- clear the counter
    elsIf clk =‘1’ and clk`event then
        If C = N-1 then
        C<=0; -- clear counter
    Else
        C<=C+1;
    End if;
End if;
End process P_count;
End Async;
```

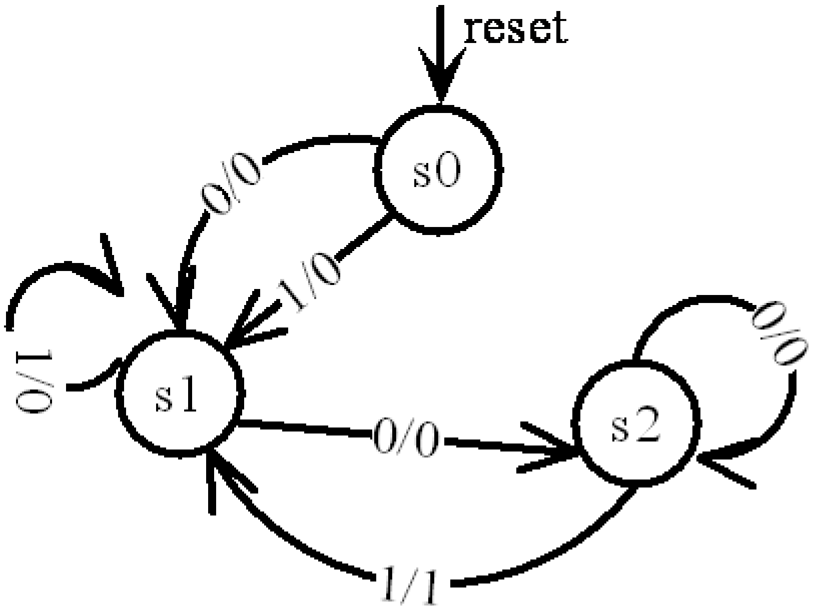
2.5.6 Example counter with asynch. Reset (cont’d)



2.5.6 Modeling Finite State Machine

- VHDL is easy to implement finite state machines
- When combined with logic synthesis, a hardware designer no longer needs to be concerned with the problems of state assignments, logic minimization, etc.
- Instead the designer can concentrate on high level behavior.

2.5.6 Modeling Finite State Machine : Example



Present	Next state		Output	
State	X=0	X=1	X=0	X=1
S0	S1	S1	0	0
S1	S2	S1	0	0
S2	S2	S1	0	1

Architecture of State Machine

```
Architecture state_machine of example is
Type stateType is (s0,s1,s2);
Signal present_state,next_state:stateType;
Begin
Comb logic: process(present_state,x)
Begin
Case present_state is
When s0 => output <=‘0’;
Next_state <=s1;
When s1 => output <=‘0’;
If (x=‘1’) then Next_state <=s1;
Else Next_state <=s2;
End if;
When s2 =>
If (x=‘1’) then Next_state <=s1; Output <=‘1’;
Else Next_state <=s2; output <=‘0’;
End if;
End case;
End process comb_logic;
```

2.6 Subprograms & Packages & use clause (cont'd)

Procedure encapsulates a collection of sequential statements that are executed for their effect

Subprogram_body <=

Procedure identifier [(parameter_list)] **is**

Begin

{sequential_statement}

End [procedure] [identifier];

Function encapsulates a collection of statement that compute a result

Subprogram_body <=

[**pure** | **impure**]

Function identifier [(parameter_list)] **return** type_mark **is**

{subprogram_declarative_item}

Begin

{sequential_statement}

End [function] [identifier];

Return_statement <= [label:] return expression;

2.6 Subprograms & Packages & use clause (cont'd)

- Package provide an important way of organizing the data and subprogram declared in a model
- Package_declaration <=
- **package** identifier **is**
- {package_declarative_item}
- **End** [package] [identifier];

- Use clause allows us to make any name form a library or package directly visible

  Use_clause <= Use selected_name {…};

  Selected_name <=

  Name.(identifier|character_literal|operator_symbol|all)

2.6.1 Procedures

Example:

Procedure average_sample **is**

Variable total:real := 0.0;

Begin

Assert samples' length >0 **severity** failure;

For index **in** samples' range **loop**

Total :=total+sample(index);

End loop;

Average := total/real(samples' length);

End procedure average_samples;

The action of a procedure are invoked by a procedure call statement

Procedure_call_statement <= [label:] procedure_name;

Example:

Average_samples;

2.6.1 Procedures (cont'd)

Return statement in a procedure

To handle exceptional conditions, the procedure may return in the middle of the procedure.

Return_statement <= [label:] **return**;

Procedure parameters

Interface_list <= ([**constant** | **variable** | **signal**] identifier {…}:[mode] subtype_indication [:=static_expression]) {;…}

mode <= **in** | **out** | **inout**

Example :

```
Type func_code is (add, substract);
Procedure do_arith_op (op: in func_code) is
   variable result: integer;
Begin
   case op is
     when add =>
                 result := op1+op2;
     when subtract =>
                 result :=op1-op2;
    end case;
End procedure do_arith_op;
```

2.6.2 Functions

Example:

```
Function limit(value, min, max :integer)
return integer is
Begin
    If value > max then
        Return max;
    Elsif  value < min then
        Return min;
    Else
        Return value;
    End if;
End function limit;
```

Pure and impure functions:

Pure function: same parameter values for same results

Impure function: same parameter values for possible different results.

Overloading

2.6.2 Functions (cont'd): Visibility of Declarations

```
Architecture arch of ent is
        Type t is…;
        Signal s:t;
        Procedure p1(…) is  - - p1 t s are visible global
                Variable v1:t;      -- v1 is visible only in procedure1
        Begin
                V1:=s;
        End Procedure p1;
Begin – arch
        Proc1: process is
                Variable v2:t;                  -- v2 is visible in proc1
                Procedure p2(…) is              --p2 is visible in proc1
                        Variable v3:t;          --v3 is only visible in procedure2
                Begin
                        P1(v2, v3…);
                End procedure p2;
        Begin –proc1
                P2(V2,…);
        End process proc1;
        Proc2: process is
        …
        Begin –proc2
                P1(…);
        End process proc2;
End architecture arch;
```

2.6.3 Packages

Example :

```
Package cpu_type is
    Constant word_size:positive := 16;
    Constant address_size :positive :=24;
    Subtype address is bit_vector(address_size-1
    downto 0);
End package cpu_type;
```

The cpu_type package has been analyzed and placed into the work library.

```
Entity address_decoder is
    Port (addr : in work.cpu_types.address;
        ……..);
    End entity address_decoder;
```

Remarks:

Each package declaration that includes subprogram declarations or deferred constant declarations must have corresponding package body to fill in the missing details. However, if a package only include other kinds of declarations, such as types, signals, constant. No package body is necessary.

2.6.3 Packages (cont'd) : Package bodies

Example :

```
Package some_arithmetic is
    Function limit(value, min, max :integer) return
    integer;
    constant word_size:positive := 16;
    Constant address_size :positive :=24;
    ……..
End package some_arithmetic;

Package body some_arithmetic is
    Function limit(value, min, max :integer) return
    integer is
    Begin
        If value > max then
            Return max;
        Elsif  value < min then
            Return min;
        Else
        Return value;
        End if;
    End function limit;
    …..
End package body some_arithmetic;
```

2.6.3 Use clause

Variable Next_address: work.cpu_types.address;

……..

Changes to

Use work.cpu_types;

    Variable Next_address: cpu_types.address;

    …..

Example:

```
Library ieee;
use ieee.std_logic_1164.std_logic;
Entity logic_block is
    Port (a, b: in std_logic;
        Y,z: out std_logic);
End entity logic_Block;
```

2.7 Resolved Signals & Generic Constants

Problem: Multiple output ports connecting one signal.

Type tri_state_logic **is** (‘0’, ‘1’, ‘z’);

Type tri_state_logic_array **is** array (integer range<>) of tri_state_logic;

```
    Function resolve_tri_state_logic(value : in
    tri_state_logic_array) return tri_state_logic is
        Variable result : tri_state_logic :=‘Z’;
    Begin
        For index in values' range loop
            If values(index) /= ‘z’ then
                Result :=values(index);
            End if;
        End loop;
        Return result;
    End function resolve_tri_state_logic;
```

Signal s1: resolve_tri_state_logic tri_state_logic**;**

Subtype resolved_logic **is** resolve_tri_state_logic tri_state_logic;

Signal S2,S3: resolved_logic;

2.7.1 Resolved Signals (cont’d)

IEEE std_logic_1164 resolved subtypes

Type std_ulogic **is** (‘U’,’X’,’0’,’1’,’Z’,’W’,’L’,’H’,’-’);

Type std_ulogic_vector **is** array (natural range<>) of std_ulogic;

Function resolved(s:std_ulogic_vector) **return** std_ulogic;

Subtype std_logic **is** resolved std_ulogic;

Type std_logic_vector **is** array (natural range <>) of std_logic;

2.7.2 Generic Constants

Generic: writing parameterized models

Entity_declaration <=

```
Entity identifier is
    [generic (generic_interface_list);]
    [port (port_interface_list);]
    {entity_declarative_item};
[begin
    Concurrent_assertion_statement |
    passive_concurrent_procedure_call_statement |
    passiv_process_statement}]
End [entity] [identifier];
```

A simple example

```
Entity and2 is
    Generic (Tpd : time);
    Port (a,b :  in bit; y :out bit);
End entity and2;

Architecture simple of and2 is
Begin
    And2_function:
    Y<= a and b after Tpd;
End architecture simple;
```

2.7.2 Generic Constants (cont'd)

A generic constant is given an actual value when the entity is used in a component instantiation statement.

- Component_instantiation_statement <=

  Instantiation_label:

  Entity entity_name [(architecture_identifier)]

  [**generic map** (generic_association_list)]

  [**port map**(port_association_list)];

  Example to use and2 for component instantiation:

  Gate1: **entity** work.and2(simple)

  Generic map(Tpd => 2 ns)

  Port map (a=>sig1,b=>sig2,y=>sig_out);

- For number of generic constants:

  Entity control_unit **is**

  Generic (Tpd_clk_out, tpw_clk : delay_length; debug: boolean:=false);

  Port (clk : in bit; ready : in bit; control : out bit);

  End entity control_unit;

  Three ways to write a generic map:

  Generic map(200ps, 1500 ps, false)

  Generic map(tpd_clk_out=>200ps, tpw_clk=> 1500 ps)

  Generic map(200ps, 1500 ps, debug => **open**) - - open means using the default value

2.7.2 Generic Constants (cont'd)

Second use of generic constants is to parameterize their structure.

```
Entity reg is
Generic (width : positive);
Port(d: in bit_vector(0 to width –1);
        q: out bit_vector(0 to width –1);
        …);
End entity reg;

Signal in_data, out_data:bit_vector(0 to bus_size-1);
…
Ok_reg:entity work.reg
Generic map(width=>bus_size)
Port map(d=>in_data, q=> out_data,…);
```

2.8 Components and Configurations

Component_declaration <=

Component identifier [**is**]

[**generic** (generic_interface_list);]

[**port**(port_interface_list);]

End component [identifeir];

Example:

component and2 –pre-defined part type

Port (I1, I2 : **in** bit; O1 **out** bit);

End component;

Component_instantiation_statement <=

Instantiation_label:

[**component**] component_name

[**generic map** (generic_association_list)]

[**port map**(port_association_list)];

2.8 Components and Configurations (cont’d)

Packaging components:

Library ieee; **use** ieee.std_logic_1164.all;

Package serial_interface_defs **is**

Subtype …

Constant …

Component serial_interface is

  Port(…);

End component serial_interface;

End package serial_interface_defs;

Entity declaration:

Library ieee; **use** ieee.std_logic_1164.all;

Use work.serial_interface_defs.**all**;

Entity serial_interface is

  Port(…);

End entity serial_interface;

An architecture body:

Library ieee; **use** ieee.std_logic_1164.all;

Architecture structure1 of micro controller is

  Use work.serial_interface_defs.serial_interface;

Begin

  serial_a : **component** serial_interface

  Port map(…);

…

2.9 Synthesis and Simulation

- **Simulation**
- model testing
- model debugging
- Find design errors,
- Find timing problems,

- **Synthesis**
- Reduction of a design description to a lower-level circuit representation.
- shorter design cycle
- Lower design cost
- Fewer design errors.
- Easier to determine available design trade-offs.

2.10 Predefined Environment

The package STANDARD is always available

Package STANDARD **is**

- **Type** Boolean **is** (FALSE, TRUE);
- **Type** BIT **is** (‘0’,’1’);
- **Type** character is (ASCII characters);
- **Type** severity_level **is** (note, warning, error, failure);
- **Type** time **is** range implementation_defined
- **Units** fs; ps=1000 fs; ns=1000 ps; us=1000ns;ms=1000us;sec=1000ms;min=60sec;hr=60 min;

  End units

Predefined numeric types

- **Type** integer **is range** implementation_defined;
- **Type** real **is range** implementation_defined;

2.10 Predefined Environment (cont’d): standard package

Function Now **return** Time – function that returns current simulation time

Subtype Natural **is** integer **range** 0 **to** integer' high;--numeric subtypes

Subtype positive **is** integer **range** 1 **to** integer' high;

Type string **is array** (**positive range**<>**) of** character;

Type bit_vector **is array**(natural **range <>) of** bit;

End STANDARD;

2.10 Predefined Environment (cont’d)

Package TEXTIO is also always available

Package TEXTIO **is**

> **Type** Line **is** access string;
>
> **Type** text **is** file of string;
>
> **Type** side **is** (right,left);
>
> **Subtype** width **is** natural;
>
> **File** Input :text **is** in “STD_INPUT”;
>
> **File** output : text **is** out “STD_OUTPUT);
>
> **Procedure** readline (F: **in** TEXT; L : **out** Line);
>
> **Procedure** read (L: **inout** line; V : **out** Bit);
>
> **Procedure** read (L: **inout** line; V : **out** Bit_vector);
>
> **Procedure** read (L: **inout** line; V : **out** Boolean);

2.10 Predefined Environment(cont'd): TEXTIO package

Procedure read (L: **inout** line; V : **out** character);

Procedure read (L: **inout** line; V : **out** integer);

Procedure read (L: **inout** line; V : **out** real);

Procedure read (L: **inout** line; V : **out** string);

Procedure read (L: **inout** line; V : **out** time);

Procedure writeline (F: **out** text; L :**in** line);

Procedure write (L:**inout** line; V : **in** bit; justified : **in** side := right; field : **in** width :=0);

Procedure write (L:**inout** line; V : **in** bit_vector; justified : **in** side := right; field : **in** width :=0);

Procedure write (L:**inout** line; V : **in** boolean; justified : **in** side := right; field : **in** width :=0);

Procedure write (L:**inout** line; V : **in** character; justified : **in** side := right; field : **in** width :=0);

Procedure write (L:**inout** line; V : **in** integer; justified : **in** side := right; field : **in** width :=0);

Procedure write (L:**inout** line; V : **in** real; justified : **in** side := right; field : **in** width :=0);

Procedure write (L:**inout** line; V : **in** string; justified : **in** side := right; field : **in** width :=0);

Procedure write (L:**inout** line; V : **in** time; justified : **in** side := right; field : **in** width :=0);

End textio;

2.10 Predefined Environment: Standard IEEE Library

Package STD_logic _1164 is not part of the VHDL standard, but it is so widely used. To access the package, a VHDL program must include the following two lines at the beginning:

Library ieee;

Use ieee.std_logic_1164.all;

Signals in this library have nine values

Type std_logic **is** (

- ‘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

2.10 Predefined Environment: Standard IEEE Library (cont’d)

- The type STD_logic is provided with a resolution function that determines the final obtained when two or more buffers drive different values onto a signal
- The type STD_ULOGIC has the same nine signal values as STD_LOGIC, but without the resolution function.

Type std_logic_vector **is**

Array (**natural** range <>) **of** STD_logic;

Type std_ulogic_vector **is**

Array (**natural** range <>) **of** STD_ulogic;

2.10 Predefined Environment: Standard IEEE Library (cont’d)

The standard IEEE library (cont’d)

- **Function** To_bit (S: std_ulogic; Xmap : Bit := ‘0’) **return** Bit;
- **Function** To_bitvector (S: std_logic_vector; Xmap : Bit := ‘0’) **return** Bit_vector;
- **Function** To_bitvector (S: std_ulogic_vector; Xmap : Bit := ‘0’) **return** Bit_vector;
- **Function** To_stdulogic (B: bit) **return** std_ulogic;
- **Function** To_stdlogicvector (B: bit_vector) **return** std_logic_vector;
- **Function** To_stdulogic (B: std_ulogic_vector) **return** std_logic_vector;
- **Function** To_stdulogic (B: bit_vector) **return** std_ulogic_vector;