# Sequential Statements

Prof. K. J. Hintz

Department of Electrical
and
Computer Engineering
George Mason University

# Sequential Statements

- If Statements
- Case Statements
- Null Statements
- Loop Statements
- Assertion & Report Statements

# If Statements

- Value of a *Boolean* Expression Determines Which Statements Are Executed
  - Expression must evaluate to TRUE or FALSE

# If Statements Syntax

```
[ if_label : ] if Boolean_expression
  then sequential_statement
  { elsif Boolean_expression
    then sequential_statement }
[ else sequential_statement ]
end if [ if_label ] ;
```

# If Statement Entity, *e.g.*,

```
entity NiCadCharger is
 port (    Voltage , Current    : in real ;
           AC                   : in bit ;
           Charged , Recharge   : out bit
       ) ;
end entity NiCadCharger ;
```

# If Statement Architecture, *e.g.*,

```
architecture ChargerArch1 of NiCadCharger is
 begin
  Charger_A: process ( Voltage ,
                       Current , AC ) is
  begin
  if Voltage  >= 9.6 then
     Charged  <=  '1' ;
     Recharge <= '0' ;
```

# If Statement, *e.g*

```
  elseif   ( AC = '1' and Current <   0.5 )
   then
    Charged  <= '0' ;
    Recharge <= '1' ;
   else
    Charged  <= '0' ;
    Recharge <= '0' ;
      end process Charger_A ;
end architecture ChargerArch1 ;
```

# Case Statement

- *Particular* Value of an Expression Determines Which Statements Are Executed

# Case Statement Syntax

```
[ case_label : ] case expression is
  ( when choices =>
    { sequential_statement } )
  { ... }
end case   [ case_label ] ;

choices <=
    ( simple_expression | discrete_range
        | element_simple_name | others )
    {   | ... }
```

# Choices in Case Statements

- Locally Static, Determined During Analysis Phase
- Exactly One Choice for Each Possible Value of Selector Expression
- More Than One Choice Can Be Listed for Each “When”

# Choices in Case Statements

- Case Specification Alternatives
  - Enumerate specific value(s)
  - Discrete Range
  - Subtype

- others
  - Keyword Which Precedes the Alternative to Be Used If All Other Case Alternatives Fail

# Case Statement, *e.g.*,

```
entity Multiplexer is
 port (
  MuxSelect : in subtype MuxType is
                positive range 0 to 3 ;
  In_0 , In_1 , In_2 , In_3 : in  bit ;
  MuxOut                    : out bit ) ;
end entity Multiplexer ;
```

# Case Statement, *e.g.*,

```
4_to_1_MUX :
 case MuxSelect is
  when 0 =>
   MuxOut <= In_0 ;
  when 1 =>
   MuxOut <= In_1 ;
```

# Case Statement, *e.g.*,

```
 when 2 =>
  MuxOut <= In_2 ;
 when 3 =>
  MuxOut <= In_3 ;
end case 4_to_1_MUX ;
```

# Null Statement Syntax

- Need Method of Specifying When No Action Is to Be Performed, *e.g.*, In Case Statement

```
[ null_label : ] null ;
```

# Null Statement, *e.g.*,

- Use As "Stub" for Code to Be Written

```
FlirFocus : process ( range , aperture )
 begin
  null ;
 end process FlirFocus ;
```

# Loop Statements

- Used for Repeated Execution of Sequential Statements

- Alternatives
  - Infinite
    - » Single or multi-phase clock
    - » Whole system turned on

# More Loop Statements

- Exit on condition
- Inner & Outer Loops
- Next
- While
- For

# Loop Statement Syntax

```
[ loop_label : ] loop
  { sequential_statement }
end loop [ loop_label ] ;
```

# Infinite Loop Entity, *e.g.*,

```
entity 2_Phase_Clock is
 port ( Clk                    : in bit ;
        Phase_1 , Phase_2      : out bit ) ;
end entity 2_Phase_Clock ;
```

# Infinite Loop Architecture, *e.g.*,

```
architecture 2PC of 2_Phase_Clock
 begin
  variable P1 : bit ;
  loop
   wait until Clk = '1'
   if P1 = '0' then
    Phase_1    <=   '0' ;
    Phase_2    <=   '1' ;
    P1         :=   '1' ;
```

# Infinite Loop Architecture, *e.g.*,

```
    else
     Phase_1 <= '1' ;
     Phase_2 <= '0' ;
     P1 := '0' ;
   end if ;
  end loop 2PC ;
end architecture 2PC ;
```

# Exit on Condition, *e.g.*,

```
variable String_Length : positive :=   0 ;
constant String_Max    : positive := 80 ;
StringFill : loop
 wait until Char_In ;
  String_Length := String_Length + 1 ;
 exit when String_Length = String_Max ;
end loop StringFill ;
```

# Inner & Outer Loops

```
for Row_Index in 1 to Row_Max
 Outer_Loop:  loop
  Inner_Loop:  loop
  exit Outer_Loop when Pixel_In = EOF ;
  New_Image ( Row_index, Col_Index ) :=
   Pixel_In ;
  end loop Inner_Loop ;
 end loop OuterLoop ;
```

# Next Loops

- The `Next` Statement Terminates Execution of the Current Iteration and Starts the Subsequent Iteration
- If There Is a Loop Label the Statement Applies to That Loop
- If There Is No Loop Label, the Statement Applies to the Inner-Most Enclosing Loop

# Next Loop, *e.g.*,

```
loop_1: loop
 loop_2: loop
  something ;
  next loop_1 when String_Length = 0 ;
  more_something ;
 end loop loop_2 ;
end loop loop_1 ;
```

# While Loop, *e.g.*,

- The Loop Only Executes, and Continues to Execute, If the Boolean Expression Evaluates to True, and Continues to Be Evaluated As True.

```
While String_Length <= String_Max
   String1:  loop
     String_Length := String_Length + 1 ;
end loop String1 ;
```

# For Loops

- The Loop Variable Is of Type Constant and Hence It Cannot Be Modified Within the Loop
- The Loop Variable Is a Strictly Local Constant

# For Loop, *e.g.*,

```
for String_Index in 1 to String_Max
 String_Reverse :  loop
  My_String ( String_Index ) :=
   Buffer ( String_Max - String_Index +1 );
end loop String_Reverse ;
```

# Assertion Statements

- Assertion Statements Check Expected Conditions at Their Location in the Program.

- Assertion Statements Are Not “If” Statements Since They Test for the Correct, Expected Results Rather Than an Error.

# Assertion Statement Syntax

[ *assertion_label* : ] **assert**

*Boolean_expression*

[ *report expression* ]

[ *severity expression* ] ;

# Assertion Statements

- Expression Must Evaluate to String
- If Other Than the Expected Condition, the Report and Severity Expressions Are Executed

# Uses of Assertion Statements

- Simulation
  - notify user when statement is executed
  - optionally print report expression
  - optionally print severity *e.g., (note, warning, error, failure)*
  - determine whether to continue

# Uses of Assertion Statements

- Synthesis
  - Value in assertion statement is assumed and circuit optimized on that value
- Verification
  - Determine that the assertation statement is true for all possible values based on all possible routes to the statement

# Report Statement

- A Note Is Printed Whenever the Expression Occurs
- Report Always Produces a Message
- Useful for Tracing Values or Paths During Execution
- Expression Must Evaluate to String

# Report Statement Syntax

[ *report_label* : ] **report** *expression*

[ *severity_expression* ] ;

# HW 2-11

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

PACKAGE Clock_2_11_pkg IS
  COMPONENT Clock_2_11
   --GENERIC ( ) ;
   PORT ( ClockOut  : out  bit  := '0' );
  END COMPONENT ;
END Clock_2_11_pkg ;
```

# HW 2-11

```
ENTITY Clock_2_11 IS
  -- GENERIC ( );
 PORT (  ClockOut  : out  bit  := '0'  );
END Clock_2_11 ;
```

# HW 2-11

```
ARCHITECTURE KJH_Clock OF Clock_2_11 IS
BEGIN
clock_gen : PROCESS
 BEGIN
  ClockOut <= '1';  WAIT FOR 10 ns ;
  Clockout <= '0';   WAIT FOR 10 ns ;
 END PROCESS clock_gen ;
END KJH_Clock ;
```

# End of Lecture