Data Flow Modeling in VHDL

ECE-331, Digital Design
Prof. Hintz
Electrical and Computer Engineering
Modeling Styles

- **Behavioral Modeling**
  - Explicit definition of mathematical relationship between the input and output
  - No implementation information

- **Structural Modeling**
  - Implicit definition of I/O relationship through particular structure
  - Interconnection of components
Behavioral Modeling

- All VHDL processes execute concurrently
- Non-procedural
  - Data-flow
  - Concurrent execution
- Procedural
  - Algorithmic
  - Sequential execution of statements
  - Equivalent to a single concurrent statement
Data Flow Model

- Concurrent Statements
  - Execute in arbitrary order
  - Execute only when any of input variables changes

Local_Sig_1 <= In_1 AND In_2 ;

Local_Sig_2 <= In_1 OR Local_Sig_1;
Signal Assignment Statements

- Two Types
  - **Conditional** concurrent signal assignment statement
  - **Selected** concurrent signal assignment statement
- Each of These Has a Sequential Process Equivalent
- Either Form Can Be Used and Are Equivalent
Other Statements

- Null Statements
- Loop Statements
- Assertion & Report Statements
Conditional Statements

- The Value of an Expression Is Assigned to a Signal When A Condition Is Evaluated As True
- Condition Must Evaluate to a BOOLEAN
BIT or BOOLEAN?

- Logical Types Are Not Equal
  - **BIT** for signals
    - ‘0’ or ‘1’
    - Character type
  - **BOOLEAN** for conditions
    - TRUE or FALSE
Relational Operators

Relational Operators Have No Relative Precedence

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>/=</td>
<td>Not equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater Than or equal to</td>
</tr>
</tbody>
</table>
Conditional Concurrent Syntax

\[
signal\_identifier <= options
\]
\[
conditional\_waveforms ;
\]

\[
options <= \\
[ guarded ] [ delay\_mechanisms ]
\]

\[
conditional\_waveforms <= \\
{ waveform when condition else } \\
waveform [ when condition ]
\]
Waveform Syntax

waveform  <=

( value_expression  [  after  time_expression  ]  )
{ ,  . . .  }
Sequential Equivalent, If

[ if_label : ] if Boolean_expression
  then
    sequential_statement
  { elsif Boolean_expression then
    sequential_statement }
[ else
  sequential_statement ]
end if [ if_label ] ;
If Statement, e.g.,

```vhdl
entity NiCadCharger is
  port ( Voltage, Current : in real;
         AC : in bit;
         Charged, Recharge : out bit);
end entity NiCadCharger;
```
If Statement, \textit{e.g.},

\begin{verbatim}
architecture ChargerArch1 of NiCadCharger is
  begin
    Charger_A: process (Voltage, Current, AC) is
      begin
        if Voltage >= 9.6 then
          Charged <= '1';
          Recharge <= '0';
      end
  end
\end{verbatim}
If Statement, *e.g.*,  

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

- The Particular Value of an Expression Determines Which Statements Are Executed
- The Sequential Equivalent To the Select Concurrent Conditional Assignment Statement Is The Case Statement
Select Concurrent Syntax

```
with expression select
  signal_identifier <= options

  selected_waveforms ;

selected_waveforms <=

  { waveform when choices , }

waveform when choices
```
Case Statement Syntax

\[
\text{[ case\_label : ] case expression is}
\]

\[
( \text{when choices =>}
\]

\[
\{ \text{sequential\_statement} \}
\]

\[
\{ \ldots \}
\]

\[
\text{end case [ case\_label ] ;}
\]
Alternatives for “choices”

```plaintext
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**
- **Others**: Precedes the Alternative to Be Used If All Other Case Alternatives Fail
Case Statement, \( e.g. \),

```vhdl
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:

```vhdl
case MuxSelect is
  when 0 =>
    MuxOut <= In_0;
  when 1 =>
    MuxOut <= In_1;
```

Case Statement, *e.g.*, 

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

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

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

- 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
  - Infinite
  - Exit on condition
  - Inner & Outer Loops
  - Next
  - While
  - For
Loop Statement Syntax

[ loop_label : ]

loop { sequential_statement }

end loop [ loop_label ] ;
entity 2_Phase_Clock is
  port ( Clk : in bit ;
         Phase_1, Phase_2 : out bit ) ;
end entity 2_Phase_Clock ;
Assertion & Report 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 & Report Statements

- If Other Than the Expected Condition, the Report and Severity Expressions Are Executed

```plaintext
[ assertion_label : ] assert Boolean_expression
  [ report  expression ]
  [ severity  expression ];
```
 Assertion Statements

- Expression Must Evaluate to String
- Uses in simulation
  - Notify user when statement is executed
  - Optionally print report expression
  - Optionally print severity (e.g., note, warning, error, failure)
  - Determine whether to continue
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_label : ] report expression
    [ severity expression ] ;
```
Operator Precedence

- **Highest to Lowest**
  - Unary operator: NOT
  - Relational operators: =, /=, <, <=, >, >=
  - Boolean (bitwise): AND, OR, NAND, NOR, XOR, XNOR

- **Parentheses Can Be Used to**
  - Force particular order of evaluation
  - Improve readability of expressions
Type Declaration/Definition

```plaintext
type identifier is type_definition;

type_definition <=
  scalar_type_definition | composite_type_definition | access_type_definition | file_type_definition
```
Scalar Type

\[
\text{scalar\_type\_definition} \leq \\
\text{enumeration\_type\_definition} \mid \\
\text{integer\_type\_definition} \mid \\
\text{floating\_type\_definition} \mid \\
\text{physical\_type\_definition}
\]
Predefined Enumerated Types

- **type** severity_level is (note, warning, error, failure);

- **type** Boolean is (false, true);
  - Used to model abstract conditions

- **type** bit is ('0', '1');
  - Used to model hardware logic levels
Bit-Vector Type

- Useful Composite Type Since It Groups Bits Together Which Can Represent Register Contents or Binary Numbers.

```vhdl
signal Out_Port_Adx: Bit_Vector
    ( 15 downto 0 );
```
Specifying Values with String Literal

Out_Port_Adx <= B "0110_1001";

Out_Port_Adx <= X "69" ;

Out_Port_Adx <= O "151" ;
Subtype (Slice)

Subtype: Values which may be Taken on by an Object are a Subset of some Base Type and may Include All Values.

```
subtype identifier is subtype_indication;
```

```
subtype_indication <=
  name [ range simple_expression
    ( to | downto ) simple_expression ]
```
Other Subtypes

- A Subtype may Constrain Values from a Scalar Type to be Within a Specified Range

```plaintext
subtype pin_count is integer range 0 to 400;

subtype octal_digits is character range '0' to '7';
```
Subtype Bounds

- A Subtype May Constrain an Otherwise Unconstrained Array Type by Specifying Bounds for the Indices

```pascal
subtype id is string (1 to 20);
subtype MyBus is bit_vector (8 downto 0);
```
Attributes

- Predefined Attributes Associated with Each Type

_Type_Name_ ` Attribute_Name_
Scalar Type Attributes

- **T’left**: leftmost value in T
- **T’right**: rightmost value in T
- **T’low**: least value in T
- **T’high**: greatest value in T
- **T’ascending**: True if ascending range, else false
- **T’image(x)**: a string representing x
- **T’value(s)**: the value in T that is represented by s
Discrete and Physical Attributes

- $T'\text{pos}(x)$: position number of $x$ in $T$
- $T'\text{val}(n)$: value in $T$ at position $n$
- $T'\text{succ}(x)$: value in $T$ at position one greater than that of $x$
- $T'\text{pred}(x)$: value in $T$ at position one less than that of $x$
- $T'\text{leftof}(x)$: value in $T$ at position one to the left of $x$
- $T'\text{rightof}(x)$: value in $T$ at position one to the right of $x$
End of Lecture

- Concurrent
- Sequential
- Conditional
- Types
- Attributes