

# Building Blocks

## Entity Declaration

Description	Example
<pre>entity entity_name is port (   [signal] identifier {, identifier}: [mode] signal_type   {; [signal] identifier {, identifier}: [mode] signal_type}); end [entity] [entity_name];</pre>	<pre>entity register8 is port (   clk, rst, en: in std_logic;   data: in std_logic_vector(7 downto 0);   q: out std_logic_vector(7 downto 0); end register8;</pre>

## Entity Declaration with Generics

Description	Example
<pre>entity entity-name is generic (   [signal] identifier {, identifier}: [mode] signal-type   [:= static_expression]   {; [signal] identifier {, identifier}: [mode] signal_type   [:= static_expression] } ); port (   [signal] identifier {, identifier}: [mode] signal_type   {; [signal] identifier {, identifier}: [mode] signal_type} ); end [entity] [entity_name];</pre>	<pre>entity register_n is generic (   width: integer := 8); port (   clk, rst, en: in std_logic;   data: in std_logic_vector(width-1 downto 0);   q: out std_logic_vector(width-1 downto 0)); end register_n;</pre>

## Architecture Body

Description	Example
<pre>architecture architecture_name of entity is type_declaration   signal_declaration   constant_declaration   component_declaration   alias_declaration   attribute_specification   subprogram_body begin {process_statement   concurrent_signal_assignment_statement   component_instantiation_statement   generate_statement} end [architecture] [architecture_name];</pre>	<pre>architecture archregister8 of register8 is begin process (rst, clk) begin if (rst = '1') then q &lt;= (others =&gt; 0); elseif (clk'event and clk = '1') then if (en = '1') then q &lt;= data; else q &lt;= q; end if; end if; end process; end archregister8;  architecture archfsm of fsm is type state)type is (st0, st1, st2); signal state: state_type; signal y, z: std_logic; begin process begin wait until clk' = '1'; case state is when st0 =&gt; state &lt;= st1; y &lt;= '1'; when st1 =&gt; state &lt;= st2; z &lt;= '1'; when others =&gt; state &lt;= st3; y &lt;= '0'; z &lt;= '0'; end case; end process; end archfsm;</pre>

## Declaring a Component

Description	Example
<pre> <b>component</b> component_name <b>port</b> (   [signal] identifier {, identifier}: [mode] signal_type   {; [signal] identifier {, identifier}: [mode] signal_type} ); <b>end component</b> [component_name]; </pre>	<pre> <b>component</b> register8 <b>port</b> (   clk, rst, en: <b>in</b> std_logic ;   data:      <b>in</b> std_logic_vector(7 <b>downto</b> 0);   q:        <b>out</b> std_logic_vector(7 <b>downto</b> 0)); <b>end component</b>; </pre>

## Declaring a Component with Generics

Description	Example
<pre> <b>component</b> component_name <b>generic</b> (   [signal] identifier {, identifier}: [mode] signal_type   [: =static_expression]   {; [signal] identifier {, identifier}: [mode] signal_type   [: =static_expression] ); <b>port</b> (   [signal] identifier {, identifier}: [mode] signal_type   {; [signal] identifier {, identifier}: [mode] signal_type} ); <b>end component</b> [component_name]; </pre>	<pre> <b>component</b> register8 <b>generic</b> (   width: integer := 8 ); <b>port</b> (   clk, rst, en: <b>in</b> std_logic;   data:      <b>in</b> std_logic_vector(width-1 <b>downto</b> 0);   q:        <b>out</b> std_logic_vector (width-1 <b>downto</b> 0)); <b>end component</b>; </pre>

## Component Instantiation (named association)

Description	Example
<pre> instantiation_label: component_name <b>port map</b> (   port_name =&gt;  signal_name                   expression                   variable_name                   <b>open</b>   {, port_name =&gt; signal_name                   expression                   variable_name                   <b>open}}</b>); </pre>	<pre> <b>architecture</b> archreg8 of reg8 is   <b>signal</b> clock, reset, enable: std_logic;   <b>signal</b> data_in, data_out: std_logic_vector(T <b>downto</b> 0); <b>begin</b>   first_reg8:   register8   <b>port map</b> (     clk =&gt; clock,     rst =&gt; reset,     en  =&gt; enable,     data =&gt; data_in,     q   =&gt; data_out); <b>end archreg8</b>; </pre>

## Component Instantiation with Generics (named association)

Description	Example
<pre> Instantiation_label: Component_name <b>generic map</b>(   generic_name =&gt;  signal_name                     expression                     variable_name                     <b>open</b>   {, generic_name =&gt; signal_name                     expression                     variable_name                     <b>open}}</b> <b>port map</b> (   port_name =&gt;    signal_name                   expression                   variable_name                   <b>open</b>   {, port_name =&gt; signal_name                   expression                   variable_name                   <b>open}}</b>); </pre>	<pre> <b>architecture</b> archreg5 of reg5 is   <b>signal</b> clock, reset, enable: std_logic;   <b>signal</b> data_in, data_out: std_logic_vector(7 <b>downto</b> 0); <b>begin</b>   first_reg5:   register_n   <b>generic map</b> (width =&gt; 5) --no semicolon here   <b>port map</b> (     clk =&gt; clock ,     rst =&gt; reset,     en  =&gt; enable,     data =&gt; data_in,     q   =&gt; data_out); <b>end archreg5</b>; </pre>

## Component Instantiation (positional association)

Description	Example
<pre> instantiation_label: component_name <b>port map</b> (signal_name   expression             variable_name   open   {, signal_name   expression     variable_name   open}); </pre>	<pre> <b>architecture</b> archreg8 of reg8 is   <b>signal</b> clock, reset, enable: std_logic;   <b>signal</b> data_in, data_out: std_logic_vector(7 <b>downto</b> 0); <b>begin</b>   first_reg8:   register8   <b>port map</b> (clock, reset, enable, data_in, data_out); <b>end archreg8</b>; </pre>

## Component Instantiation with Generics (positional association)

Description	Example
instantiation_label: component_name generic map ( signal_name   expression   variable_name   open {, signal_name   expression   variable_name   open}) port map ( signal_name   expression   variable_name   open {, signal_name   expression   variable_name   open});	architecture archreg5 of reg5 is signal clock, reset, enable: std_logic; signal data_in, data_out: std_logic_vector(7 downto 0); begin first_reg5: register_n generic map (5) port map (clock, reset, enable, data_in, data_out); end archreg5;

## Concurrent Statements

### Boolean Equations

Description	Example
relation { and relation }   relation { or relation }   relation { xor relation }   relation { nand relation }   relation { nor relation }	v<= (a and b and c) or d; --parenthesis req'd w/ 2-level logic w <= a or b or c; x <= a xor b xor c; y <= a nand b nand c; z <= a nor b nor c;

### When-else Conditional Signal Assignment

Description	Example
{expression when condition else} expression;	x<= '1' when b = c else '0'; x<= j when state = idle else k when state = first_state else l when state = second_state else m when others;

### With-select-when Select Signal Assignment

Description	Example
with selection_expression select {identifiers <= expression when identifier   expression   discrete_range   others,} identifier <= expression when identifier   expression   discrete_range   others;	architecture archfsm of fsm is type state_type is (st0, st1, st2, st3, st4, st5, st6, st7, st8); signal state: state_type; signal y, z: std_logic_vector(3 downto 0); begin with state select x<= "0000" when st0   st1; -- st0 "or" st1 "0010: when st2   st3; z when st4; z when others; end archfsm;

### Generate Scheme for Component Instantiation or Equations

Description	Example
generate_label: (for identifier in discrete_range)   (if condition) generate {concurrent_statement} end generate [generate_label];	g1: for i in 0 to 7 generate reg1: register8 port map (clock, reset, enable, data_in(i), data_out(i); g2: for j in 0 to 2 generate a(j) <= b(j) xor c(j); end generate g2;

# Sequential Statements

## Process Statement

Description	Example
<pre>[process_label:] process (sensitivity_list)   type_declaration     constant_declaration     variable_declaration     alias_declaration } begin { wait_statement     signal_assignment_statement     variable_assignment_statement     if_statement     case_statement     loop_statement } end process [process_label];</pre>	<pre>my_process process (rst, clk)   constant zilch : std_logic_vector(7 downto 0) :=     "0000_0000"; begin   wait until clk = '1';   if (rst = '1') then     q &lt;= zilch;   elsif (en = '1') then     q &lt;= data;   else     q &lt;= q;   end if end my_process;</pre>

## if-then-else Statement

Description	Example
<pre>if condition then sequence_of_statements { elsif condition then sequence_of_statements } [else sequence_of_statements] end if;</pre>	<pre>if (count = "00") then   a &lt;= b; elsif (count = "10") then   a &lt;= c; else   a &lt;= d; end if;</pre>

## case-when Statement

Description	Example
<pre>case expression is { when identifier   expression   discrete_range   others =&gt;   sequence_of_statements } end case;</pre>	<pre>case count is   when "00" =&gt;     a &lt;= b;   when "10" =&gt;     a &lt;= c;   when others =&gt;     a &lt;= d; end case;</pre>

## for-loop Statement

Description	Example
<pre>[loop_label:] for identifier in discrete_range loop   { sequence_of_statements } end loop [loop_label];</pre>	<pre>my_for_loop for i in 3 downto 0 loop   if reset(i) = '1' then     data_out(i) := '0';   end if; end loop my_for_loop;</pre>

## while-loop Statement

Description	Example
<pre>[loop_label:] while condition loop   { sequence_of_statements } end loop [loop_label];</pre>	<pre>count := 16; my_while_loop: while (count &gt; 0) loop   count := count - 1;   Result &lt;= result + data_in; end loop my_while_loop;</pre>

## Describing Synchronous Logic Using Processes

### No Reset (Assume clock is of type std\_logic)

Description	Example
<pre>[process_label:] process (clock) begin   if clock'event and clock = '1' then --or rising_edge     synchronous_signal_assignment_statement;   end if; end process [process_label]; -----or----- [process_label:] process begin   wait until clock = '1';   synchronous_signal_assignment_statement; end process [process_label];</pre>	<pre>reg8_no_reset: process (clk) begin   if clk'event and clk = '1' then     q &lt;= data;   end if; end process reg8_no_reset; -----or----- reg8_no-reest: process begin   wait until clock = '1';   q &lt;= data; end process reg8_no_reset;</pre>

### Synchronous Reset

Description	Example
<pre>[process_label:] process (clock) begin   if clock'event and clock = '1' then     if synch_reset_signal = '1' then       synchronous_signal_assignment_statement;     else       synchronous_signal_assignment_statement;     end if;   end if; end process [process_label];</pre>	<pre>reg8_sync_reset: process (clk) begin   if clk'event and clk = '1' then     if sync_reset = '1' then       q &lt;= "0000_0000";     else       q &lt;= data;     end if;   end if; end process;</pre>

### Asynchronous Reset or Preset

Description	Example
<pre>[process_label:] process (reset, clock) begin   if reset = '1' then     asynchronous_signal_assignment_statement;   elsif clock'event and clock = '1' then     synchronous_signal_assignment_statement;   end if; end process [process_label];</pre>	<pre>reg8_async_reset: process (asyn_reset, clk) begin   if asyn_reset = '1' then     q &lt;= (others =&gt; '0');   elsif clk'event and clk = '1' then     q &lt;= data;   end if; end process reg8_async_reset;</pre>

### Asynchronous Reset and Preset

Description	Example
<pre>[process_label:] process (reset, preset, clock) begin   if reset = '1' then     asynchronous_signal_assignment_statement;   elsif preset = '1' then     asynchronous_signal_assignment_statement;   elsif clock'event and clock = '1' then     synchronous_signal_assignment_statement;   end if; end process [process_label];</pre>	<pre>reg8_async: process (asyn_reset, asyn_preset, clk) begin   if asyn_reset = '1' then     q &lt;= (others =&gt; '0');   elsif asyn_preset = '1' then     q &lt;= (others =&gt; '1');   elsif clk'event and clk = '1' then     q &lt;= data;   end if; end process reg8_async;</pre>

### Conditional Synchronous Assignment (enables)

Description	Example
<pre>[process_label:] process (reset, clock) begin   if reset = '1' then     asynchronous_signal_assignment_statement;   elsif clock'event and clock = '1' then     if enable = '1' then       synchronous_signal_assignment_statement;     else       synchronous_signal_assignment_statement;     end if;   end if; end process [process_label];</pre>	<pre>reg8_sync_assign: process (rst, clk) begin   if rst = '1' then     q &lt;= (others =&gt; '0');   elsif clk'event and clk = '1' then     if enable = '1' then       q &lt;= data;     else       q &lt;= q;     end if;   end if; end process reg8_sync_assign;</pre>

## Translating a State Flow Diagram to a Two-Process FSM Description

Description	Example																	
<div style="margin-top: 10px;"> <table border="1" style="display: inline-table; border-collapse: collapse;"> <thead> <tr> <th rowspan="2">state</th> <th colspan="2">outputs</th> </tr> <tr> <th>oe</th> <th>we</th> </tr> </thead> <tbody> <tr> <td>idle</td> <td>0</td> <td>0</td> </tr> <tr> <td>decision</td> <td>0</td> <td>0</td> </tr> <tr> <td>write</td> <td>0</td> <td>1</td> </tr> <tr> <td>read</td> <td>1</td> <td>0</td> </tr> </tbody> </table> </div>	state	outputs		oe	we	idle	0	0	decision	0	0	write	0	1	read	1	0	<pre> architecture state_machine of example is   type StateType is (idle, decision, read, write);   signal present_state, next_state : StateType; begin state_comb:process(present_state, read_write, ready) begin   case present_state is     when idle =&gt;   oe &lt;= '0' ; we &lt;= '0';     if ready = '1' then       next_state &lt;= decision;     else       next_state &lt;= idle;     end if;     when decision =&gt;  oe &lt;= '0' ; we &lt;= '0';     if (read_write = '1') then       next_state &lt;= read;     else       -- read_write='0'       next_state &lt;= write;     end if;     when read =&gt;      oe &lt;= '1' ; we &lt;= '0';     if (read = '1') then       next_state &lt;= idle;     else       next_state &lt;= read;     end if;     when write =&gt;     oe &lt;= '0' ; we &lt;= '1';     if (ready = '1') then       next_state &lt;= idle;     else       next_state &lt;= write;     end if;   end case; end process state_comb;  state_clocked:process(clk) begin   if (clk'event and clk = '1') then     present_state &lt;= next_state;   end if; end process state_clocked; end state_machine; </pre>
state		outputs																
	oe	we																
idle	0	0																
decision	0	0																
write	0	1																
read	1	0																

## Data Objects

### Signals

Description	Example
<ul style="list-style-type: none"> <li>• Signals are the most commonly used data object in synthesis designs.</li> <li>• Nearly all basic designs, and many large designs as well, can be fully described using signals as the only kind of data object.</li> <li>• Signals have projected output waveforms.</li> <li>• Signal assignments are scheduled, not immediate; they update projected output waveforms.</li> </ul>	<pre> architecture archinternal_counter of internal_counter is   signal count, data:std_logic_vector(7 downto 0); begin process(clk)   begin     if (clk'event and clk = '1') then       if en = '1' then         count &lt;= data;       else         count &lt;= count + 1;       end if;     end if;   end process; end archinternal_counter; </pre>

### Constants

Description	Example
<p>Constants are used to hold a static value; they are typically used to improve the readability and maintenance of code.</p>	<pre> my_process: process (rst, clk)   constant zilch : std_logic_vector(7 downto 0) := "0000_0000"; begin   wait until clk = '1';   if (rst = '1') then     q &lt;= zilch;   elsif (en = '1') then     q &lt;= data;   else     q &lt;= q;   end if; end my_process; </pre>

## bit and bit\_vector

Description	Example
<ul style="list-style-type: none"><li>• Bit values are: '0' and '1'.</li><li>• Bit_vector is an array of bits.</li><li>• Pre-defined by the IEEE 1076 standard.</li><li>• This type was used extensively prior to the introduction and synthesis-tool vendor support of std_logic_1164.</li><li>• Useful when metalogic values not required.</li></ul>	<pre>signal x: bit; ... if x = '1' then   state &lt;= idle; else   state &lt;= start; end if;</pre>

## Boolean

Description	Example
<ul style="list-style-type: none"><li>• Values are TRUE and FALSE.</li><li>• Often used as return value of function</li></ul>	<pre>signal a: boolean; ... if x = '1' then   state &lt;= idle; else   state &lt;= start; end if;</pre>

## Integer

Description	Example
<ul style="list-style-type: none"><li>• Values are the set of integers.</li><li>• Data objects of this type are often used for defining widths of signals or as an operand in an addition or subtraction.</li><li>• The types std_logic_vector and bit_vector work better than integer for components such as counters because the use of integers may cause "out of range" run-time simulation errors when the counter reaches its maximum value.</li></ul>	<pre>entity counter_n is   generic (     width: integer := 8);   port (     clk, rst, in: std_logic;     count: out std_logic_vector(width-1 downto 0)); end counter_n; ... process(clk) begin   if (rst = '1') then     count &lt;= 0;   elsif (clk'event and clk = '1') then     count &lt;= count + 1;   end if; end process;</pre>

## Enumeration Types

Description	Example
<ul style="list-style-type: none"><li>• Values are user-defined</li><li>• Commonly used to define states for a state machine.</li></ul>	<pre>architecture archfsm of fsm is   type state_type is (st0, st1, st2);   signal state: state_type;   signal y, z: std_logic; begin   process   begin     wait until clk'event = '1';     case state is       when st0 =&gt;         state &lt;= st2;         y &lt;= '1'; z &lt;= '0';       when st1 =&gt;         state &lt;= st3;         y &lt;= '1'; z &lt;= '1';       when others =&gt;         state &lt;= st0;         y &lt;= '0'; z &lt;= '0';       end case;     end process;   end archfsm;</pre>

## Variables

Description	Example
<ul style="list-style-type: none"> <li>Variables can be used in processes and subprograms -- that is, in sequential areas only.</li> <li>The scope of a variable is the process or subprogram</li> <li>A variable in a subprogram does not retain its value between calls.</li> <li>Variables are most commonly used as the indices of loops or for the calculation of intermediate values, or immediate assignment.</li> <li>To use the value of a variable outside of the process or subprogram in which it was declared the value of the variable must be assigned to a signal</li> <li>Variable assignment is immediate, not scheduled</li> </ul>	<pre>architecture archloopstuff of loopstuff is   signal data: std_logic_vector(3 downto 0);   signal result: std_logic; begin   process (data)     variable tmp: std_logic;   begin     tmp := '1';     for i in a'range downto 0 loop       tmp := tmp and data(i);     end loop;     result &lt;= tmp;   end process; end archloopstuff;</pre>

## Data Types and Subtypes

### std\_logic

Description	Example
<ul style="list-style-type: none"> <li>Values are:           <ul style="list-style-type: none"> <li>'U', -- Uninitialized</li> <li>'X', -- Forcing unknown</li> <li>'0', -- Forcing 0</li> <li>'1', -- Forcing 1</li> <li>'Z', -- High impedance</li> <li>'W', -- Weak unknown</li> <li>'L', -- Weak 0</li> <li>'H', -- Weak 1</li> <li>'-', -- Don't care</li> </ul> </li> <li>The standard multivalued logic system for VHDL model interoperability.</li> <li>A resolved type (i.e., a resolution function is used to determine the value of a signal with more than one driver).</li> <li>To use must include the following two lines:           <pre>library ieee; use ieee.std_logic_1164.all;</pre> </li> </ul>	<pre>Signal x, data, enable: std_logic; ... x &lt;= data when enable = '1' else 'Z';</pre>

### std\_ulogic

Description	Example
<ul style="list-style-type: none"> <li>Values are:           <ul style="list-style-type: none"> <li>'U', -- Uninitialized</li> <li>'X', -- Forcing unknown</li> <li>'0', -- Forcing 0</li> <li>'1', -- Forcing 1</li> <li>'Z', -- High impedance</li> <li>'W', -- Weak unknown</li> <li>'L', -- Weak 0</li> <li>'H', -- Weak 1</li> <li>'-', -- Don't care</li> </ul> </li> <li>An unresolved type (i.e., a signal of this type may have only one driver).</li> <li>Along with its subtypes, std_ulogic should be used over user-defined ability of VHDL models among synthesis and simulation tools.</li> <li>To use must include the following two lines:           <pre>library ieee; use ieee.std_logic_1164.all;</pre> </li> </ul>	<pre>Signal x, data, enable: std_ulogic; ... x &lt;= data when enable = '1' else 'Z';</pre>

### std\_logic\_vector and std\_ulogic\_vector

Description	Example
<ul style="list-style-type: none"> <li>Are arrays of types std_logic and std_ulogic.</li> <li>Along with its subtypes, std_logic_vector should be used over user-defined types to ensure interoperability of VHDL models among synthesis and simulation tools.</li> <li>To use must include the following two lines:           <pre>library ieee; use ieee.std_logic_1164.all;</pre> </li> </ul>	<pre>signal mux: std_logic_vector (7 downto 0) ... if state = address or state = ras then   mux &lt;= dram_a; else   mux &lt;= (others =&gt; 'Z'); end if;</pre>

## In, Out, buffer, inout

Description	Example
<ul style="list-style-type: none"> <li>In: Used for signals (ports) that are inputs-only to an entity.</li> <li>Out: Used for signals that are outputs – only and for which the values are not required internal to the entity.</li> <li>Buffer: Used for signals that are outputs but for which the values are required internal to the given entity. Caveat with usage: If the local port of the instantiated component is of mode buffer, then if the actual is also a port it must of mode buffer as well. For this reason some designers standardize on mode buffer.</li> <li>Inout: Used for signals that are truly bidirectional. May also be used for signals for that are input-only or output-only, at the expense of code readability.</li> </ul>	<pre> Entity counter_4 is port (   clk, rst, ld: in std_logic;   term_cnt: buffer std_logic;   count: inout std_logic_vector (3 down to 0)); end counter_4; architecture archcounter_4 of counter_4 is   signal int_rst: std_logic;   signal int_count: std_logic_vector(3 downto 0)); begin   process(int_rst, clk)   begin     if(int_rst = '1') then       int_count &lt;= "0000";     elsif (clk'event and clk='1') then       if(ld = '1') then         int_count &lt;=count;       else         int_count &lt;=int_count + 1;       end if;     end if;   end process;   term_cnt &lt;= count(2) and count(0);   --term_cnt is 3   int_rst &lt;=term_cnt or rst;   -- resets at term_cnt   count &lt;=int_count when ld='0' else "ZZZZ";   --count is bidirectional end architecture_4; </pre>

## Operators

All operators of the same class have the same level of precedence. The classes of operators are listed here in the order of decreasing precedence. Many of the operators are overloaded in the `std_logic_1164`, `numeric_bit`, and `numeric_std` packages.

Miscellaneous Operators...page 164

Description	Example
<ul style="list-style-type: none"> <li>Operators: <code>**</code>, <code>abs</code>, <code>not</code></li> <li>The <code>not</code> operator is used frequently, the other two are rarely used for designs to be synthesized.</li> <li>Predefined for any integer type (<code>**</code>), any numeric type (<code>abs</code>), and either bit and Boolean (<code>not</code>).</li> </ul>	<pre> signal a, b, c:bit; ... a &lt;= not (b and c); </pre>

## Multiplying Operators

Description	Example
<ul style="list-style-type: none"> <li>Operators: <code>*</code>, <code>/</code>, <code>mod</code>, <code>rem</code>.</li> <li>The <code>*</code> operator is occasionally used for multipliers; the other three are rarely used in synthesis.</li> <li>Predefined for any integer type (<code>*</code>, <code>/</code>, <code>mod</code>, <code>rem</code>), and any floating point type (<code>*</code>, <code>/</code>).</li> </ul>	<pre> variable a, b:integer range 0 to 255; ... a &lt;= b * 2; </pre>

## Sign

Description	Example
<ul style="list-style-type: none"><li>Operators: +,-.</li><li>Rarely used for synthesis.</li><li>Predefined for any numeric type (floating – point or integer).</li></ul>	<pre>variable a, b, c: integer range 0 to 255; ... a &lt;= - (b+2);</pre>

## Adding Operators

Description	Example
<ul style="list-style-type: none"><li>Operators: +,-</li><li>Used frequently to describe incrementers, decrementers, adders and subtractors.</li><li>Predefined for any numeric type.</li></ul>	<pre>signal count: integer range 0 to 255; ... count &lt;= count+1;</pre>

## Shift Operators

Description	Example
<ul style="list-style-type: none"><li>Operators: sll, srl, sla, sra, rol, ror.</li><li>Used occasionally.</li><li>Predefined for any one-dimensional array with elements of type bit or Boolean. Overloaded for std_logic arrays.</li></ul>	<pre>signal a, b: bit_vector(4 downto 0); signal c: integer range 0 to 4; ... a &lt;= b sll c;</pre>

## Relational Operators

Description	Example
<ul style="list-style-type: none"><li>Operators: =, /=, &lt;, &lt;=, &gt;, &gt;=.</li><li>Used frequently for comparisons.</li><li>Predefined for any type (both operands must be of same type)</li></ul>	<pre>signal a, b: integer range 0 to 255; signal agtb: std_logic; ... if a &gt;= b then   agtb &lt;= '1'; else   agtb &lt;= '0';</pre>

## Logical Operators

Description	Example
<ul style="list-style-type: none"><li>Operators: and, or, nand, nor, xor, xnor.</li><li>Used frequently to generate Boolean equations.</li><li>Predefined for types bit and Boolean. Std_logic_1164 overloads these operators for std_u logic and its subtypes.</li></ul>	<pre>signal a, b, c: std_logic; ... a &lt;= b and c;</pre>