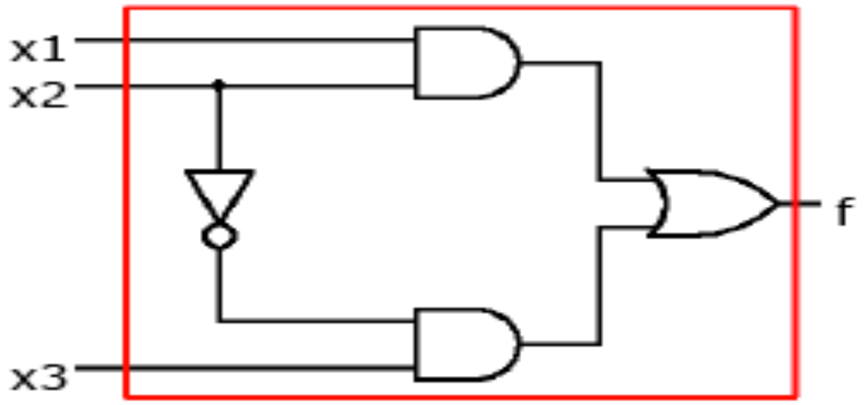
CLEMSON
UNIVERSITY

M.C. Smith

Adapted from:
"VHDL Quick Start" by Peter J. Ashenden

VHDL INTRO (PART II)

VHDL Code Example



```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY example1 IS
   PORT (x1,x2,x3 : IN std_logic;
         f        : OUT std_logic);
END example1;
ARCHITECTURE LogicFunc OF example1 IS
BEGIN
   f <= (x1 AND x2) OR (NOT x2 AND x3);
END LogicFunc;
```

Boolean Operators In VHDL

- VHDL has built-in support for the following operators
  - AND logical AND
  - OR logical OR
  - NOT logical NOT
  - NAND, NOR, XOR, XNOR
- Assignment operator: <=
  - A variable (usually an output, mode **OUT**) should be assigned the result of the logic expression on the right hand side of the operator
- VHDL does not assume any precedence of logic operators… Use parentheses in expressions to determine precedence
  - Exception to this is NOT logical operator
- In VHDL, a logic expression is called a ***simple assignment statement***. There are other types that will be introduced that are useful for more complex circuits

Native Operators

- Logical - defined for type bit, bit_vector, boolean*
  - AND, NAND
  - OR, NOR
  - XOR, XNOR
  - NOT

•The NOT operator has highest precedence
•The rest are (and, nand, xor, etc.) are of equal precedence so you just used parentheses where necessary

- Relational - defined for types bit, bit_vector, integer*
  - = (equal to)
  - /= (not equal to)
  - < (less than)
  - <= (less than or equal to)
  - > (greater than)
  - >= (greater than or equal to)

•Relational operators return a boolean value
•When comparing vectors of different lengths, the comparison is done from left to right. Therefore "111" is greater than "10111"

** overloaded for std_logic, std_logic_vector; will explain this shortly…*

Native Operators (contd.)

- Unary Arithmetic - defined for type integer*
  - - (arithmetic negate)
- Arithmetic - defined for type integer*
  - +(addition), * (multiplication)
  - - (subtraction)
  - /, mod, rem, abs, **
- Concatenation - defined for strings
  - &

VERY IMPORTANT!!!!!

- Note, a STRING is any sequence of characters, therefore a std_logic_vector is an example of a STRING

** overloaded for std_logic, std_logic_vector; will explain this shortly…*

Overloaded Operators

- In VHDL, the scope of all of the previous operators can be extended (or overloaded) to accept any type supported by the language, e.g.,

```
--      assume a declaration of a 16-bit vector as
  SIGNAL pc IS std_logic_vector(15 DOWNTO 0);
--      then a valid signal assignment is
  pc <= pc + 3;
--      assuming the '+' operator has been overloaded to
---     accept std_logic_vector and integer operands
```

- The std_logic_1164 package defines overloaded logical operators (AND, OR, NOT, etc.,) for the std_logic and std_logic_vector types

Legal VHDL Identifiers

- Letters, digits, and underscores only (first character must be a letter)
- The last character cannot be an underscore
- Two underscores in succession are not allowed
- Using reserved words is not allowed (the VHDL editor will highlight reserved words for this reason)
- Examples
  - Legal
    - tx_clk, Three_State_Enable, sel7D, HIT_1124
  - Not Legal
    - _tx_clk, 8B10B, large#num, case, clk_

Operators Summary

- Logical
  - and, or, nand, nor, xor, xnor, not
  - NOTE: nand and nor not associative
- Relational
  - =, /=, <, <=, >, >=
- Shift
  - sll, srl, sla, sra, rol, ror
  - Shift logical or arithmetic
  - Rotate left/right
- Adding
  - +, −, &
- Multiplying
  - *, /, mod, rem
  - a rem b = a − (a/b) * b
  - a mod b = a − b * n (for some n)
- Other
  - abs, **

VHDL Assignment Stmts

- VHDL provides several types of statements that can be used to assign logic values to signals
  - Simple assignment statements
    - Used previously, for logic or arithmetic expressions
  - Selected signal assignments
  - Conditional signal assignments
  - Generate statements
  - If then else statements
  - If-then-– Case statements

Architecture Body May Contain:

- Concurrent Signal Assignment statements (CSAs)
  - a <= b + c;
  - Activated when any of the associated signals change their values
  - Independent from other statements in the given architecture
- Component Instantiations
  - Pairs the entity with the architecture
  - Defines a sub-component of the design
  - Associates signals or values with the ports
- Processes with behavioral descriptions
  - An independent sequential process representing the behavior of some portion of the design

CSAs Example

- Example below specifies a half adder
- Whenever A or B changes in value, both signal assignments will be activated concurrently → Hardware parallelism!

```
architecture dataflow of HalfAdder is
begin
      Sum <= A xor B;
      Carry <= A and B;
end architecture;
```

Component Instantiation

- Introduces a relationship to a component declaration
- Port map may be either *named* or *positional*
- Either way, it must match!

Preferred!

```
reg5 : fill_reg
port map ( clk => clk,
           rst => rst,
           write => write_comp,
           read => readout,
           data_in => dataout_comp,
           FFout => FFout,
           data_out => datareg5_out);

reg5 : fill_reg
port map (clk, rst, write_comp,
           readout, dataout_comp,
           FFout, datareg5_out);
```

Other Assignment Statements

- For ***concurrent assignment statements (or concurrent signal assignments - CSAs)*** the order in which they appear in VHDL code does not affect the meaning of the code
- VHDL provides a second category of statements, ***sequential assignment statements***, for which the ordering of the statements **may** affect the meaning of the code
  - **If then else** and **case** statements are examples of sequential stmts
- VHDL requires that sequential assignment statements be placed inside another statement, the ***process*** statement
- Therefore, all behavioral designs require caution!

What is a VHDL “Process” ?

- Processes are either awake or asleep (active or inactive)
- A process normally has a sensitivity list
  - When a signal in that sensitivity list changes value, the process wakes up and all of the sequential statements are “executed”
  - For example, a process with a clock signal in its sensitivity list will become active on changes of the clock signal
- At the end of the process, all outputs are assigned and the process goes back to sleep until the next time a signal changes in the sensitivity list

```
<label> process ( <sensitivity list> )
      < process declarations >
begin
      < statements >
end process <label>;
```

Process Statement

- Process executes or is evaluated whenever an event occurs on a signal in the sensitivity list
  - Remember: No sensitivity list is required, BUT if not, wait statements must be used in the process
- Multiple processes can execute concurrently
- The statements describing the behavior are executed **sequentially**
  - This is true from a simulation standpoint
  - From a synthesized hardware point-of-view, multiple assignments to a single signal (variable) generally implies multiplexing of the assignments to produce a single output
- Assignments made inside the process are not visible outside the process until all statements in the process have been evaluated
  - If there are multiple assignments to the same signal inside a process, only the last one has any visible effect
- The <label> is optional

Process Example (from a simulation)

```
process
begin
    wait for 15 ns;
    clk <= not(clk);
end process;

process (clk, rst)
begin
if rst = '1' then
    readout <= '0';
elsif (clk'event and clk= '1') then
    if (fout = '1') then
            readout <= '1';
    else
            readout <='0';
    end if;
end if;
end process;
```

```
<label> process ( <sensitivity list> )
        < process declarations >
begin
        < statements >
end process <label>;
```

2 processes operating in parallel

1st process generates the clock

2nd process has the clock in the sensitivity list

Behavioral Statements

- Wait Statement – causes suspension of a process or procedure
  - WAIT ON <sensitivity> UNTIL <expression> FOR <time>
- Examples:
  - WAIT ON A, B, C;
  - WAIT UNTIL A > 10;
  - WAIT FOR 30ns;
  - WAIT ON A FOR 100ns;
  - WAIT UNTIL C < 4 FOR 50ms;
  - WAIT ON B UNTIL A > 4;
  - WAIT ON C UNTIL B > 5 FOR 30us;
  - WAIT; -- suspends forever

Concurrency

- “Concurrent” means **nonprocedural (i.e. not in a Process statement!)**
  - The order in which the CSA stmts appear textually has nothing to do with the order in which they execute
  - They execute at the same time!!!!!!!!
- Note: `INT1` and `INT2` are internal signals, used only within the **architecture** (keep scope in mind)

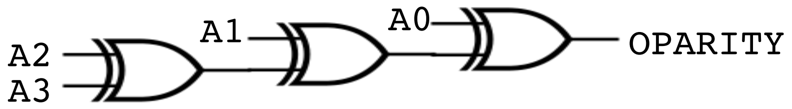
```
entity XOR2_OP is
port (A, B : in BIT;
      Z : out BIT);
end XOR2_OP;
architecture AND_OR_CONCURRENT
  of XOR2_OP is
  signal INT1, INT2 : BIT;
begin
  INT1 <= A and not B;
  INT2 <= not A and B;
  Z <= INT1 or INT2;
end AND_OR_CONCURRENT;
```

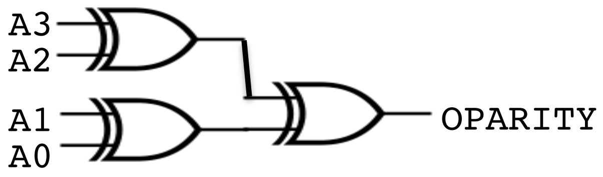
Concurrency (cont.)

- Concurrency is fundamental to hardware and VHDL!
- Think in terms of *parallel signal transformations*
- Design tips:
  - Large procedural descriptions are ***hard to synthesize***
  - Concurrency – must think about how it is ***realized*** – "visualize the possible hardware"
  - Think about design ***decomposition*** (modular and hierarchical)

- Next example shows two designs for an odd parity detector
- Think about which one will have better performance (i.e. yield a faster design)?

VHDL Stmts & Hardware Concurrency





```
entity OPARITY_DETECT is
port (A3,A2,A1,A0: in std_logic;
     OPARITY : out std_logic);
end OPARITY_DETECT;
architecture design1 of
  OPARITY_DETECT is
  signal INT1, INT2 : BIT;
begin
  INT1 <= A3 xor A2;
  INT2 <= INT1 xor A1;
  OPARITY <= INT2 xor A0;
end design1;
```

```
entity OPARITY_DETECT is
port (A3,A2,A1,A0: in std_logic;
     OPARITY : out std_logic);
end OPARITY_DETECT;
architecture design2 of
  OPARITY_DETECT is
  signal INT1, INT2 : BIT;
begin
  INT1 <= A3 xor A2;
  INT2 <= A0 xor A1;
  OPARITY <= INT2 xor INT1;
end design2;
```

Combinational Logic

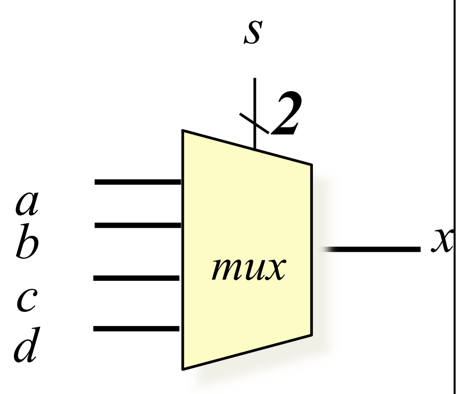
- Can be described with concurrent statements
  - boolean equations
  - when-else
  - with-select-when
  - component instantiations
- Can be described with sequential statements
  - if-then-else
  - case-when

Lets look at examples of each…

Combinational Logic w/ Boolean Equations

- Boolean Equations can be used in both concurrent and sequential signal assignment statements…
- A 4-1 multiplexer is shown below

```
x <=  (a AND NOT(s(1)) AND NOT(s(0))) OR
     (b AND NOT(s(1)) AND s(0)) OR
     (c AND s(1) AND NOT(s(0))) OR
     (d AND s(1) AND s(0)) ;
```



concurrent

Selective Signal Assignment: with-select-when

- Assignment based on a selection signal
- WHEN clauses must be mutually exclusive
- Use a WHEN OTHERS when all conditions are not specified
- Only one reference to the signal, only one assignment operator (<=)

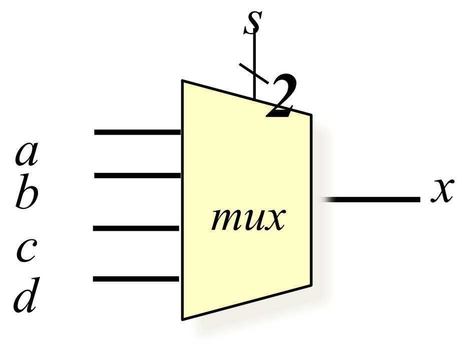
```
WITH selection_signal SELECT
signal_name <= value_1 WHEN value_1 of selection_signal,
               value_2 WHEN value_2 of selection_signal,
               ...
               value_n WHEN value_n of selection_signal,
               value_x  WHEN OTHERS;
```

concurrent

Combinational Logic w/ Selective Signal Assignment

- The same 4-1 multiplexer is shown below

```
with s select
   x <= a when "00" ,
          b when "01" ,
          c when "10" ,
          d when others ;
```

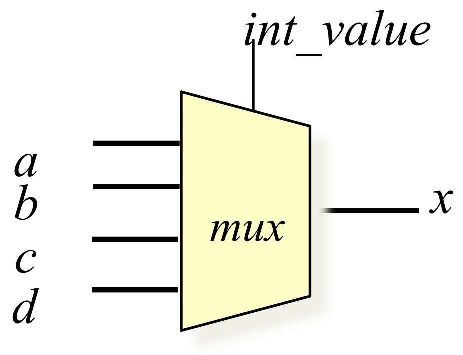


concurrent

More on with-select-when

- You can use a range of values

```
with int_value select
  x <= a when 0 to 3,
       b when 4 | 6 | 8 ,
       c when 10 ,
       d when others ;
```



concurrent

Conditional Signal Assignment: when-else

- Assign a value based on conditions
- Any simple expression can be a condition
- Priority goes in order of appearance
- Only one reference to the signal, only one assignment operator (<=)
- Use a final ELSE to avoid latches

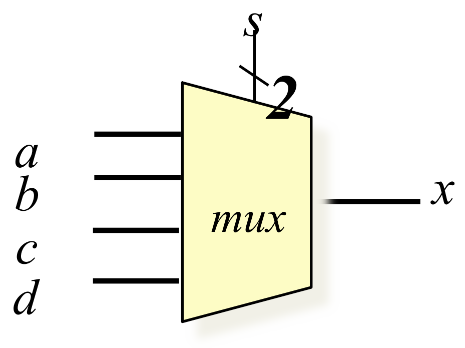
```
signal_name <= value_1 WHEN condition1 ELSE
         value_2 WHEN condition2 ELSE
         ...
         value_n WHEN condition N ELSE
         value_x ;
```

concurrent

Combinational Logic w/ Conditional Signal Assignment

- The same 4-1 multiplexer is shown below

```
x <= a when (s = “00”) else
     b when (s = “01”) else
     c when (s = “10”) else
     d ;
```



concurrent

Combinational Logic w/ Conditional Signal Assignment

- The when conditions do not have to be mutually exclusive (as in with-select-when)
- A priority encoder is shown below
  - Order of the conditions matters!!!!!!!

```
j <= w when (a = ‘1’) else
     x when (b = ‘1’) else
     y when (c = ‘1’) else
     z when (d = ‘1’) else
     “000” ;
```

concurrent

Combinatorial Logic w/ Sequential Statements

- Grouped inside **Process Statements**
- Processes are concurrent with *one another* and with *concurrent signal assignment statements*
- Order of sequential statements ***<u>does</u>*** make a difference in synthesis

sequential

Sequential Statements: if-then-else

- Used to select a set of statements to be executed
- Selection based on a boolean evaluation of a condition or set of conditions
- Absence of ELSE results in implicit memory (FF or latch)

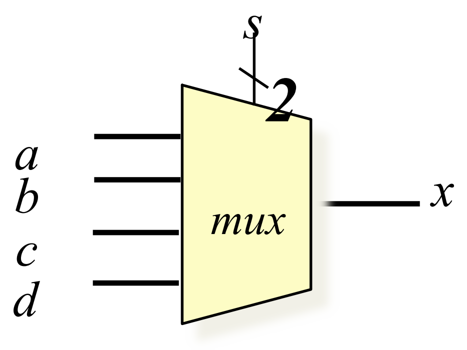
```
IF condition(s) THEN
   do something;
ELSIF condition_2 THEN     -- optional
   do something different;
ELSE                       -- optional
   do something completely different;
END IF ;
```

sequential

if-then-else

- 4-1 mux shown below

```
mux4_1: process (a, b, c, d, s)
    begin
        if  s = “00” then x <= a ;
           elsif  s = “01” then x <= b ;
           elsif  s = “10” then x <= c ;
           else   x <= d ;
        end if;
    end process mux4_1 ;
```



sequential

Sequential Statements: Case-When

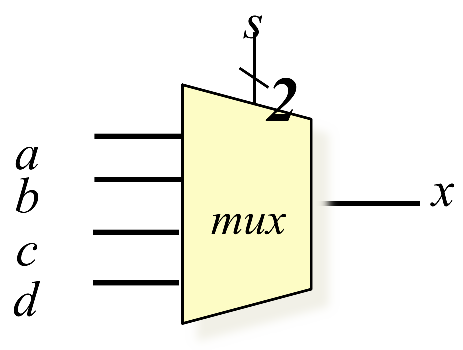
```
CASE selection_signal IS
  WHEN value_1_of_selection_signal  =>
   (do something) -- set of statements 1
  WHEN value_2_of_selection_signal  =>
   (do something) -- set of statements 2
...
  WHEN value_N_of_selection_signal  =>
   (do something) -- set of statements N
  WHEN OTHERS  =>
   (do something) -- default action
END CASE ;
```

(sort of the sequential counterpart to the with-select-when)

sequential

The CASE Statement: 4-1 Mux

```
ARCHITECTURE archdesign OF design IS
 SIGNAL s: std_logic_vector(0 TO 1);
BEGIN
 mux4_1: PROCESS (a,b,c,d,s)
 BEGIN
    CASE s IS
       WHEN "00"   => x <= a;
       WHEN "01"   => x <= b;
       WHEN "10"   => x <= c;
       WHEN OTHERS => x <= d;
    END CASE;
 END PROCESS mux4_1;
END archdesign;
```

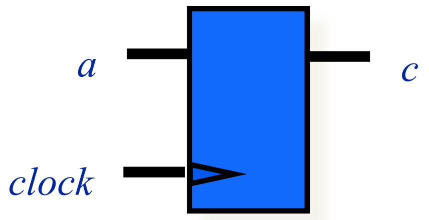


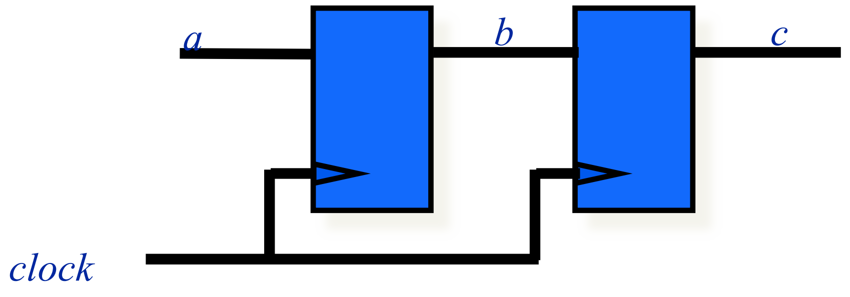
sequential

Signal Assignment in Processes

Which Circuit do you think is Correct?

```
ARCHITECTURE arch_reg OF reg IS
SIGNAL b: std_logic
reg2: PROCESS
BEGIN
    WAIT UNTIL clock = '1';
       b <= a;  -- after the rising clock edge, a goes to b
       c <= b;  -- after the rising clock edge, b goes to c
END PROCESS reg2;
END arch_reg
```





Signal Assignment in Processes

- Inside processes, signals are not updated immediately
  → Instead, they are scheduled to be updated
- Signals are updated at the END PROCESS
- Therefore, on the previous slide, two registers will be synthesized (c <= b will be the old b)
- In some cases, the use of a concurrent statement outside the process will fix the problem, but this is not always possible
- So how else can we fix this problem ?

Variables

- When a concurrent signal assignment outside the process cannot be used, the previous problem can be avoided using a variable
  - Variables are like signals, BUT they can only be used inside a given PROCESS (signals are inside the architecture)
  - They cannot be used to communicate information between processes
  - Variables can be of any valid VHDL data type
  - The value assigned to a variable is available immediately
  - Assignment of variables is done using a colon (:), like this:

    c := a AND b;

Using Variables vs. Signals

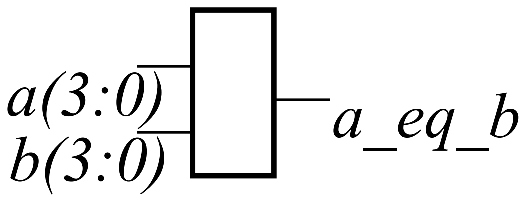
- Solution using a variable within a process:

```
-- assume a and c are signals defined elsewhere

ARCHITECTURE arch_reg OF reg IS
PROCESS
VARIABLE b: std_logic ;
BEGIN
   WAIT UNTIL clock = '1' ;
      b := a ;   -- this is immediate
      c <= b ;   -- this is scheduled
END PROCESS ;
END arch_reg;
```

Exercise #2: Architecture Declaration of a Comparator

- The entity declaration is as follows:



```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY compare IS PORT (
    a, b  : IN std_logic_vector(3 DOWNTO 0);
    a_eq_b: OUT std_logic);
END compare;
```

- Write an architecture that causes **a_eq_b** to be asserted when **a** is equal to **b**
- Multiple solutions exist

Exercise #2: Three possible solutions

- *Concurrent* statement solution using a *conditional* assignment:

```
ARCHITECTURE df1_compare OF compare IS
BEGIN
    a_eq_b <= '1' WHEN a = b ELSE '0';
END df1_compare;
```

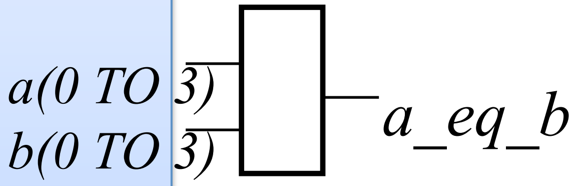
- *Concurrent* statement solution using boolean equations:

```
ARCHITECTURE df2_compare OF compare IS
BEGIN
    a_eq_b <= NOT(
        (a(0) XOR b(0)) OR
        (a(1) XOR b(1)) OR
        (a(2) XOR b(2)) OR
        (a(3) XOR b(3)));
END df2_compare;
```

Exercise #2: Three possible solutions

- Solution using a process with sequential statements:

```
ARCHITECTURE behv_compare OF compare IS
BEGIN
comp: PROCESS (a, b)
    BEGIN
        IF a = b THEN
            a_eq_b <= '1';
        ELSE
            a_eq_b <= '0';
        END IF;
    END PROCESS comp;
END behv_compare;
```



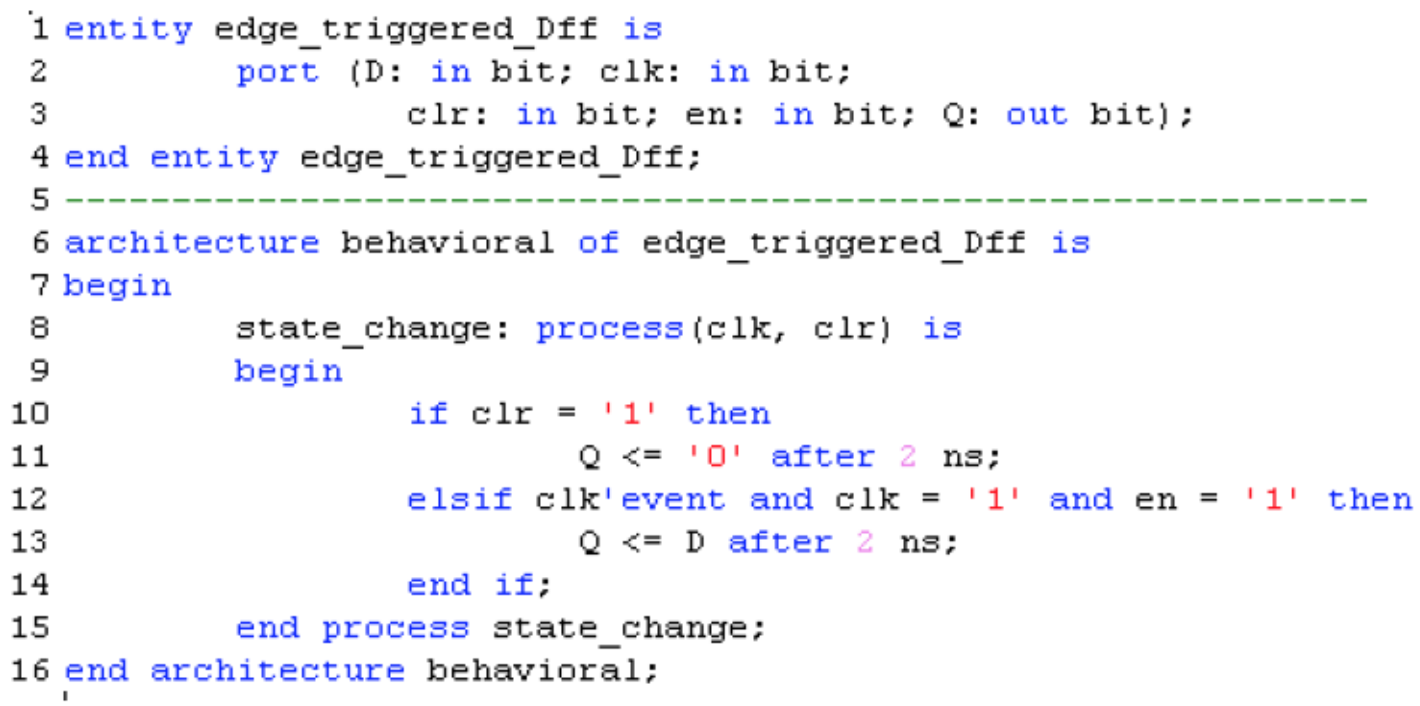
Generate Statement

- Whenever we write structural VHDL code and instantiate components, we often create ***instances*** of a particular ***component***
  - A multi-stage ripple carry adder made from a number of single-bit full adders might be an example
- If we need to create a large number of instances of a component, a more compact form is desired
- VHDL provides a feature called the **FOR GENERATE** statement
  - This statement provides a loop structure for describing regularly structured hierarchical code
  - NOTE: The loop cannot be terminated early

```
name : FOR N IN 1 TO 8 GENERATE
        concurrent-statements
END GENERATE name;
```

Generate Statement Example

- In this example, we will build a 32-bit wide shift register using a DFF with clear



```
entity edge_triggered_Dff is
        port (D: in bit; clk: in bit;
                clr: in bit; en: in bit; Q: out bit);
end entity edge_triggered_Dff;
-------------------------------------------------------------
architecture behavioral of edge_triggered_Dff is
begin
        state_change: process(clk, clr) is
        begin
                if clr = '1' then
                        Q <= '0' after 2 ns;
                elsif clk'event and clk = '1' and en = '1' then
                        Q <= D after 2 ns;
                end if;
        end process state_change;
end architecture behavioral;
```

Example Cont.

- Using this DFF, we want to place 32 of them in a row and wire the *i*th input to the *(i-1)*st output
- The first input should be wired to *shift-in* and the last output should be wired to *shift-out*
- We could do this with 32 component instantiation statements of the form:

```
dff1: component edge_triggered_Dff
              port map (Sin, clk, clr, en, parallel_data(1) );
dff2: component edge_triggered_Dff
              port map (parallel_data(1), clk, clr, en, parallel_data(2) );
dff3: …
```

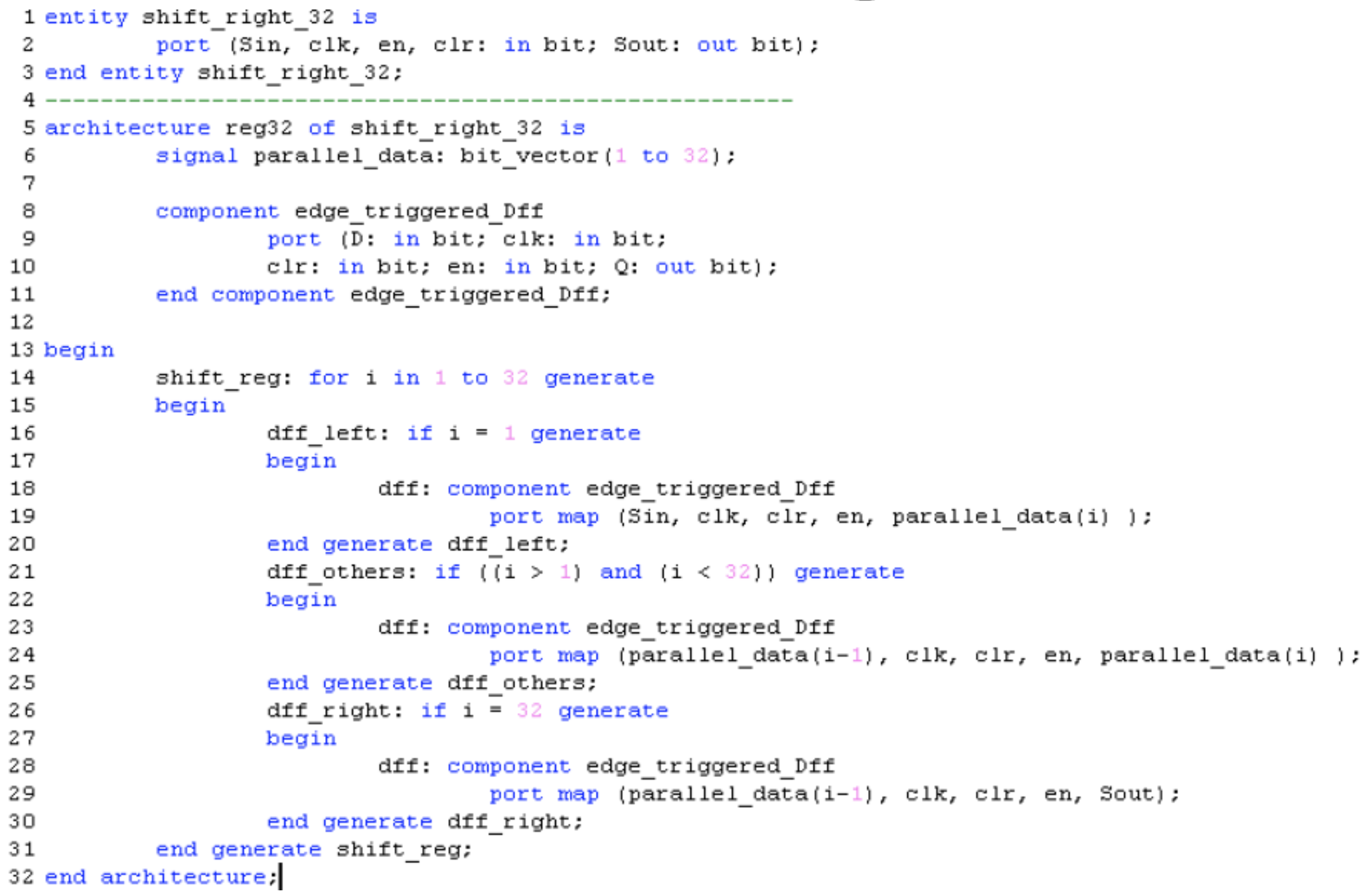
Example Cont.

- Or… we could use *generate* to do the work for us

```
shift_reg: for i in 1 to 32 generate
begin
    dffx: generate
    begin
        dff: component edge_triggered_Dff
            port map (parallel_data(i-1), clk, clr, en, parallel_data(i) );
    end generate dffx;
end generate shift_reg;
```

- Here, we are placing 32 copies of our DFF and wiring them up. There is a problem, however… There is no 0 element *(i*-1 when *i*= 1)
- Another problem exists at the 32nd element where we need to wire the shift out

32-bit Shift Register



```vhdl
entity shift_right_32 is
        port (Sin, clk, en, clr: in bit; Sout: out bit);
end entity shift_right_32;
------------------------------------------------------
architecture reg32 of shift_right_32 is
        signal parallel_data: bit_vector(1 to 32);

        component edge_triggered_Dff
                port (D: in bit; clk: in bit;
                clr: in bit; en: in bit; Q: out bit);
        end component edge_triggered_Dff;

begin
        shift_reg: for i in 1 to 32 generate
        begin
                dff_left: if i = 1 generate
                begin
                        dff: component edge_triggered_Dff
                                port map (Sin, clk, clr, en, parallel_data(i) );
                end generate dff_left;
                dff_others: if ((i > 1) and (i < 32)) generate
                begin
                        dff: component edge_triggered_Dff
                                port map (parallel_data(i-1), clk, clr, en, parallel_data(i) );
                end generate dff_others;
                dff_right: if i = 32 generate
                begin
                        dff: component edge_triggered_Dff
                                port map (parallel_data(i-1), clk, clr, en, Sout);
                end generate dff_right;
        end generate shift_reg;
end architecture;
```

Architecture Example

```
Library IEEE;
    use IEEE.STD_LOGIC_1164.all;
    use IEEE.STD_LOGIC_ARITH.all;
    use IEEE.STD_LOGIC_UNSIGNED.all;
entity count_example is
port(
    clk : in std_logic;
    rst : in std_logic;
    enable : in std_logic;
    load : in std_logic;
    data_in : in std_logic_vector(4 downto 0);
    data_out: out std_logic_vector(4 downto 0));
end count_example;
```

Libraries

Architecture Example (cont.)

```
architecture version1 of count_example is
  component adder
    generic ( WIDTH : INTEGER := 8)
    port (
          in1 :in std_logic_vector(WIDTH−1 downto 0);
          in2 :in std_logic_vector(WIDTH−1 downto 0);
          out :out std_logic_vector(WIDTH−1 downto 0)
    );
   signal CNT :std_logic_vector(4 downto 0);
   signal result :std_logic_vector(4 downto 0);
   CONSTANT ONE :std_logic_vector(4 downto 0) := "00001";
```

Architecture Example (cont.)

```
   begin
   data_out <= CNT;
               adder1 : adder
                  generic map ( WIDTH => 5)
                  port map ( in1 => CNT,    in2 => ONE,       out => result  );
   process (clk, rst)
   begin
   if rst = ’1’  then
               CNT <= "00000";
   elsif (clk’event and clk = ’1’) then
               if (load = ’1’) then
                   CNT <= data_in;
               elsif (enable = ’1’) then
                   CNT <= result;
               else
                   CNT <= CNT;
               end if;
   end if;
   end process;
end version1;
```

Will this code synthesize?

```
entity my_code is
   port(A,B,CLK: in std_logic;
            D: out std_logic);
end my_code;
architecture my_arch of my_code is
   signal C: std_logic;
begin
   process(CLK)
   begin
      if (CLK=‘1’ and CLK’event) then
         C <= A and B;
      end if;
   end process;
end my_arch;
```