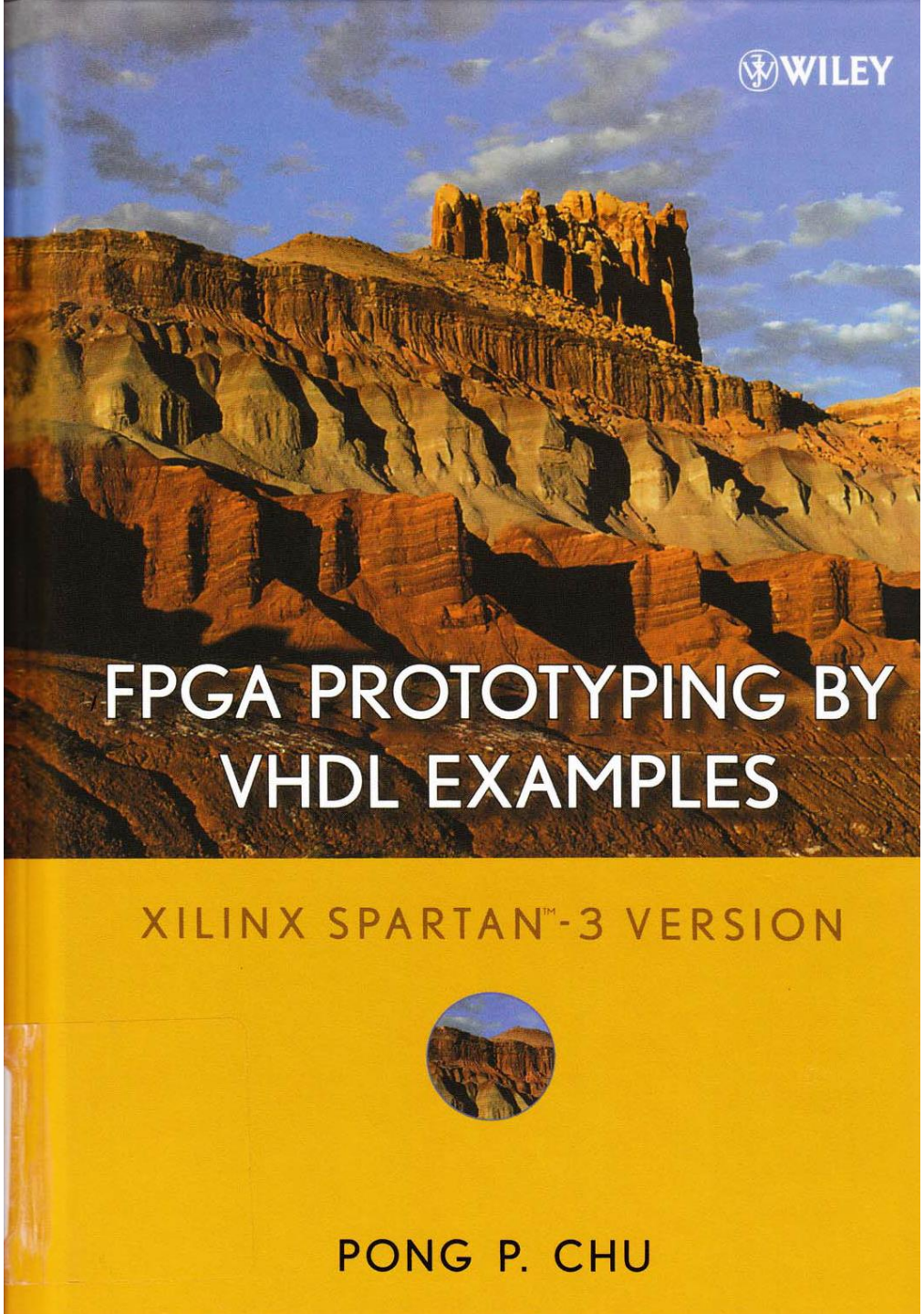
VHDL

Very High Speed Integrated Circuits

Hardware Description Language

The vast majority of this powerpoint comes from the VHDL tutorial in the Georgia Tech lab book by Furman (the VHDL tutorial posted on ASU learn), and the "FPGA Prototyping by VHDL Examples" and "RTL Hardware Design Using VHDL" books by Chu. There is also some material from the Digital Systems book by Tocci. The site www.NandLand.com has tutorials on FPGAs and VHDL. Help on VHDL commands is on the websites: http://vhdl.renerta.com/ and https://www.ics.uci.edu/~jmoorkan/vhdlref/



This text was written for Xilinx (the main competitor of Altera), but most VHDL commands work on both FPGAs.

VHDL LANGUAGE ELEMENTS

Simulation only

Reserved words

ABS	FOR	PACKAGE
ACCESS	FUNCTION	PORT
AFTER	GENERATE	PROCEDURE
ALIAS	GENERIC	PROCESS
ALL	GUARDED	RANGE
AND	IF	RECORD
ARCHITECTURE	IN	REGISTER
ARRAY	INOUT	REM
ASSERT	IS	REPORT
ATTRIBUTE	LABEL	RETURN
BEGIN	LIBRARY	SELECT
BLOCK	LINKAGE	SEVERITY
BODY	LOOP	SIGNAL
BUFFER	MAP	SUBTYPE
BUS	MOD	THEN
CASE	NAND	TO
COMPONENT	NEW	TRANSPORT
CONFIGURATION	NEXT	TYPE
CONSTANT	NOR	UNITS
DISCONNECT	NOT	UNTIL
DOWNTO	NULL	USE
ELSE	OF	VARIABLE
ELSIF	ON	WAIT
END	OPEN	WHEN
ENTITY	OR	WHILE
EXIT	OTHERS	WITH
FILE	OUT	XOR

VHDL LANGUAGE ELEMENTS

Identifiers

- An identifier is the name of an object
- Objects are named entities that can be assigned a value and have a specific data type
- Objects include signals, variables, and constants
- Must start with an alphabet character and end with an alphabet or a numeric character
- Can contain numeric characters or underscore and cannot contain spaces
- Are not case sensitive
- May be up to 32 characters long
- Cannot contain any reserved words

Symbols

--	begins comment (to end of line)
( )	encloses port names in entity declaration, encloses highest priority operations in Boolean and arithmetic expressions
' '	encloses scalar values
" "	encloses array values
;	ends VHDL statements and declarations
,	separates objects
:	separates object identifier names from mode and data type in declarations
<=	assigns values in signal assignment statements
:=	assigns values in variable assignment statements or to constants
=>	separates signal assignment statements from WHEN clause in CASE statements
+	addition operator
-	subtraction operator
=	equality operator
/=	inequality operator
>	greater than comparator
>=	greater than or equal to comparator
<	less than comparator
<=	less than or equal to comparator
&	concatenation operator

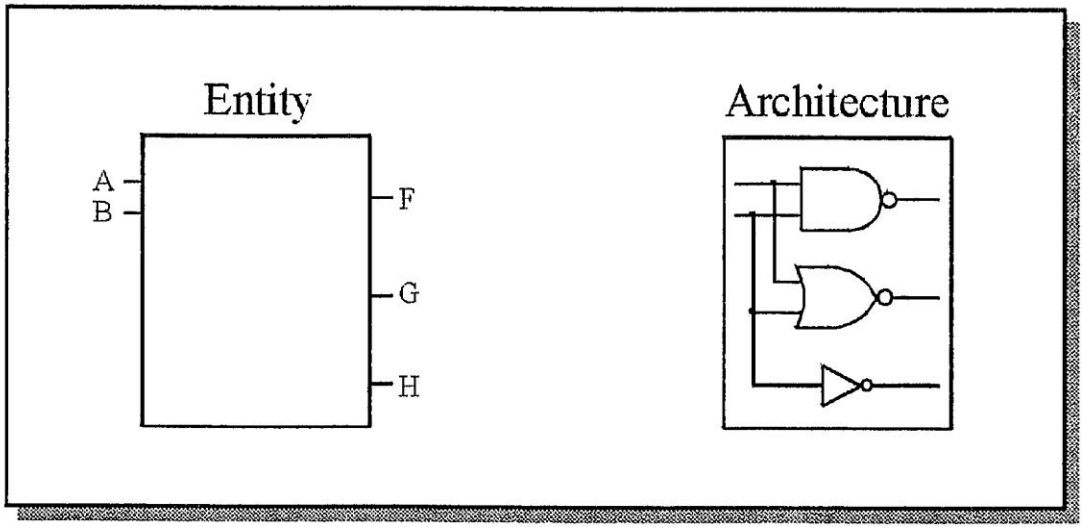
Synthesis Data Types

BIT	object can only have single-bit values of '0' or '1'
STD_LOGIC	object with multi-value logic including '0', '1', 'X', 'Z'
INTEGER	objects with whole number (decimal) values, e.g., 54, –21
BIT_VECTOR	objects with arrays of bits such as "10010110"
STD_LOGIC_VECTOR	objects with arrays of multi-value logic, e.g., "01101XX"

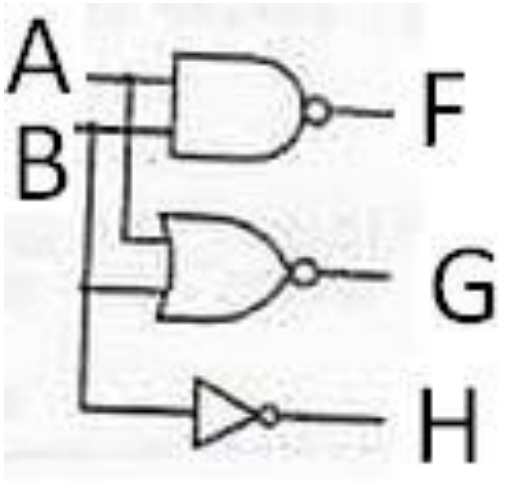
Fig. 6-2b Summary of VHDL language elements

24. Summarizing Entity and Architecture Viewpoints

An earlier topic illustrated our device with a transparent view of the entire system. The two illustrations separate the external and internal views.



```
ARCHITECTURE arch OF basis_gates IS
BEGIN
        F <= A nand B;
        G <=  A nor B;
        H <= not B;
END arch;
```



The 3 circuits run concurrently (simultaneously) in parallel. However, usually output H changes first because an INVERTER is a faster gate than a NAND or a NOR.

1.2.4 Data type and operators

VHDL is a *strongly typed language*, which means that an object must have a data type and only the defined values and operations can be applied to the object. Although VHDL is rich in data types, our discussion is limited to a small set of predefined types that are suitable for synthesis, mainly the `std_logic` type and its variants.

std_logic type The `std_logic` type is defined in the `std_logic_1164` package and consists of nine values. Three of the values, `'0'`, `'1'`, and `'Z'`, which stand for logical 0, logical 1, and high impedance, can be synthesized. Two values, `'U'` and `'X'`, which stand for "uninitialized" and "unknown" (e.g., when signals with `'0'` and `'1'` values are tied together), may be encountered in simulation. The other four values, `'-'`, `'H'`, `'L'`, and `'W'`, are not used in this book.

'1'	Logic 1 (just like BIT type)
'0'	Logic 0 (just like BIT type)
'z'	High impedance*
'-'	don't care (just like you used in your K maps)
'U'	Uninitialized
'X'	Unknown
'W'	Weak unknown
'L'	Weak '0'
'H'	Weak '1'

*We will study tristate logic in Chapter 8.

TABLE 4-9 Common VHDL data types.

Data Type	Sample Declaration	Possible Values	Use
BIT	y :OUT BIT;	'0' '1'	y <= '0';
STD_LOGIC	driver :STD_LOGIC	'0' '1' 'z' 'x' '-'	driver <= 'z';
BIT_VECTOR	bcd_data :BIT_VECTOR (3 DOWNTO 0);	"0101" "1001" "0000"	digit <= bcd_data;
STD_LOGIC_VECTOR	dbus :STD_LOGIC_VECTOR (3 DOWNTO 0);	"0Z1X"	IF rd = '0' THEN dbus <= "zzzz";
INTEGER	SIGNAL z:INTEGER RANGE –32 TO 31;	–32.. –2, –1,0,1,2 . . . 31	IF z > 5 THEN . . .

Boolean equation – but don't use bit vector, use std_logic_vector.

```
-- Programmer: J.S. Clements  September 24, 2003     VHDL combinatorial lab

-- Program: Prime Number Detector using Boolean Equations
-- VHDL equation syntax from Tocci 9th ed. Lab Book, Unit6V, Example 6-1, p.97


ENTITY primenumberequ IS          -- entity name must match file name? yes
      PORT
            (
                  a : IN BIT_VECTOR(3 DOWNTO 0);  -- four-bit input A3,A2,A1,A0

                  y : OUT BIT  -- output single bit
            );
END primenumberequ;

ARCHITECTURE primenumberbool OF primenumberequ IS
BEGIN
            -- Boolean equations for output of prime number detector

            -- y is true if 4-bit input (0-15) is prime(1,2,3,5,7,11,13)

            -- need () because AND doe not have priority over OR

                y  <= (NOT a(3) AND a(0))
                   OR (a(2) AND NOT a(1) AND a(0))
                   OR (NOT a(2) AND a(1) AND a(0))
                   OR (NOT a(3) AND NOT a(2) AND a(1));
END primenumberbool;
```

Tocci – but don't use bit vector, use std_logic_vector because the Altera library uses it and so does the rest of the world.

```
ENTITY bitwise_and IS
PORT(d, g           :IN BIT_VECTOR (3 DOWNTO 0);
     y              :OUT BIT_VECTOR (3 DOWNTO 0));
END bitwise_and;
ARCHITECTURE a OF bitwise_and IS
BEGIN
     y <= d AND g;
END a;
```

What circuit does this VHDL make?

3.4 MODELING WITH A PROCESS

3.4.1 Process

To facilitate system modeling, VHDL contains a number of *sequential statements*, which are executed in sequence. Since their behavior is different from that of a normal concurrent circuit model, these statements are encapsulated inside a *process*. A process itself is a concurrent statement. It can be thought of as a black box whose behavior is described by sequential statements.

Sequential statements include a rich variety of constructs, but many of them don't have clear hardware counterparts. A poorly coded process frequently leads to unnecessarily complex implementation or cannot be synthesized at all. Detailed discussion of sequential statements and processes is beyond the scope of this book. For synthesis, we restrict the use of the process to two purposes:

- Describe routing structures with *if* and *case* statements.
- Construct templates for memory elements (discussed in Chapter 4).

The simplified syntax of a process with a sensitivity list is

```
process(sensitivity_list)
begin
   sequential statement;
   sequential statement;
   . . .
end process;
```

The sensitivity_list is a list of signals to which the process responds (i.e., is "sensitive to"). For a combinational circuit, all the input signals should be included in this list. The body of a process is composed of any number of sequential statements.

3.4.2 Sequential signal assignment statement

The simplest sequential statement is a *sequential* signal assignment statement. The simplified syntax is

```
sig <= value_expression;
```

The statement must be encapsulated inside a process.

Although its syntax is similar to that of a simple *concurrent* signal assignment statement, the semantics are different. When a signal is assigned multiple times inside a process, only the last assignment takes effect. For example, the code segment

```
process(a,b)
begin
```

```
    c <= a and b;
    c <= a or b;
end process;
```

is the same as

```
process(a,b)
begin
    c <= a or b;
end process;
```

On the other hand, if they are concurrent signal assignment statements, as in

```
-- not within a process
c <= a and b;
c <= a or b;
```

the code infers an and cell and an or cell, whose outputs are tied together. It is not allowed in most device technology and thus is a design error.

The semantics of assigning a signal multiple times inside a process is subtle and can sometimes be error-prone. Detailed explanations can be found in the references cited in the Bibliographic section. We use multiple assignments only to avoid unintended memory, as discussed in Section 3.5.4.

31. THE IF...ELSE CONSTRUCT

Statements within a **PROCESS** are evaluated *sequentially*. (Read the last sentence again!) Therefore, a **PROCESS** must be used to define the behavior of a circuit that is not a simple combinatorial equation.

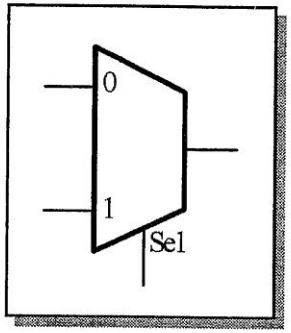
The **IF...ELSE** combination, for instance, is a sequential construct that can only be utilized inside a **PROCESS** statement.

Q: What logical TTL device does this VHDL code describe?

```
ENTITY unknown IS

   PORT( sel      : IN  STD_LOGIC;
         input0   : IN  STD_LOGIC;
         input1   : IN  STD_LOGIC;
         output   : OUT STD_LOGIC );

END unknown;

ARCHITECTURE arch OF unknown IS
BEGIN

   PROCESS( sel, input0, input1 )
   BEGIN

      IF ( sel = '0' ) THEN
         output <= input0;
      ELSE
         output <= input1;
      END IF;

   END PROCESS;

END arch;
```

Ans: A 2:1 Multiplexer.

Notice that the process describes how *output* is set to either *input0* or *input1*. As with a physical multiplexer, *output* is not the result of a sequential process even though the description is!



Tocci – Internal Signal, but don't use bit, don't use integers for input pins

```
ENTITY fig4_59 IS
PORT(digital_value:IN INTEGER RANGE 0 TO 15;     -- declare 4-bit input
      too_cold, just_right, too_hot :OUT BIT);
END fig4_59 ;

ARCHITECTURE howhot OF fig4_59 IS
SIGNAL status    :BIT_VECTOR (2 downto 0);
BEGIN
   PROCESS (digital_value)
      BEGIN
         IF (digital_value <= 8) THEN status <= "100";
         ELSIF (digital_value > 8 AND digital_value < 11) THEN
               status <= "010";
         ELSE  status <= "001";
         END IF;
      END PROCESS ;
   too_cold    <= status(2);      -- assign status bits to output
   just_right  <= status(1);
   too_hot     <= status(0);
END howhot;
```

FIGURE 4-59 Temperature range example in VHDL using ELSIF.

35. THE CASE STATEMENT

The **CASE** statement tests for equalities and can only be used inside a **PROCESS** due to its sequential nature. It is used to replace **IF…ELSIF…ELSIF…ELSE** - type constructs where equality is the only test. (i.e., sel = '0' or sel = '1', etc.)

(Should you need to test for greater than (>), less than (<), or anything other than equality (=), then you need to use the **IF…ELSIF…ELSIF…ELSE**-type construct.

```
PROCESS( sel, input0, input1 )
BEGIN

    CASE sel IS
       WHEN '0' =>
          output <= input0;
       WHEN '1' =>
          output <= input1;
       WHEN OTHERS =>
          output <= input1;
    END CASE;

END PROCESS;
```

The **WHEN OTHERS** is required as the last statement of the **CASE** construct.

Q: In the example, since it seems that the *sel* signal is always either a '1' or a '0', is it still necessary to have the **WHEN OTHERS** statement? Why?

U, X are for simulation

Ans: Yes. Particularly when a system is being simulated, initial values are sometimes unknown for several cycles. In addition, but without going into the nuances here, there are several other valid signal conditions defined for the **STD_LOGIC** signal type as illustrated to the right. The **WHEN OTHERS** statement takes care of all other possibilities.

```
'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
```

43. Using a VECTOR TO THE CASE STATEMENT

By indicating the name *data* in the sensitivity list, the **PROCESS** will be sensitive to any bit change in the *data* vector. Additionally, the assignment statements after each **WHEN**... with regards to *data* also need to be changed.

Q: How would the changes look in the **CASE** statement?

Ans: The sensitivity list would no longer need to list all variables, and the assignments with regards to *data* would have to reference each designated bit in *data* with a subscript.

```
PROCESS( sel, data )
  BEGIN

    CASE sel IS
      WHEN "000" =>
         output <= data( 0 );
      WHEN "001" =>
         output <= data( 1 );
      WHEN "010" =>
         output <= data( 2 );
      WHEN "011" =>
         output <= data( 3 );
      WHEN "100" =>
         output <= data( 4 );
      WHEN "101" =>
         output <= data( 5 );
      WHEN "110" =>
         output <= data( 6 );
      WHEN "111" =>
         output <= data( 7 );
      WHEN OTHERS =>
         output <= '0';
    END CASE;

END PROCESS;
```

3.5.4 Unintended memory

Although a process is flexible, a subtle error in code may infer incorrect implementation. One common problem is the inclusion of unintended memory in a combinational circuit. The VHDL standard specifies that a signal will *keep its previous value* if it is not assigned in a process. During synthesis, this infers an internal state (via a closed feedback loop) or a memory element (such as a latch).

To prevent unintended memory, we should observe the following rules while developing code for a combinational circuit:

- Include all input signals in the sensitivity list.
- Include the else branch in an if statement.
- Assign a value to every signal in every branch.

For example, the following code segment tries to generate a greater-than (i.e., `gt`) and an equal-to (i.e., `eq`) output signal: Fix this code.

```
process(a)                  -- b missing from sensitivity list
begin
   if (a > b) then          -- eq not assigned in this branch
      gt <= '1';
   elsif (a = b) then       -- gt not assigned in this branch
      eq <= '1';
   end if;                  -- else branch is omitted
end process;
```

Although the syntax is correct, it violates all three rules. For example, `gt` will keep its previous value when the `a>b` expression is `false` and a latch will be inferred accordingly.

The correct code should be

```
process(a,b)
begin
    if (a > b) then
        gt <= '1';
        eq <= '0';
    elsif (a = b) then
        gt <= '0';
        eq <= '1';
    else
        gt <= '0';
        eq <= '0';
    end if;
end process;
```

Since multiple sequential signal assignment statements are allowed inside a process, we can correct the problem by assigning a default value in the beginning:

```
process(a,b)
begin
    gt <= '0';              -- assign default value
    eq <= '0';
    if (a > b) then
        gt <= '1';
    elsif (a = b) then
        eq <= '1';
    end if;
end process;
```

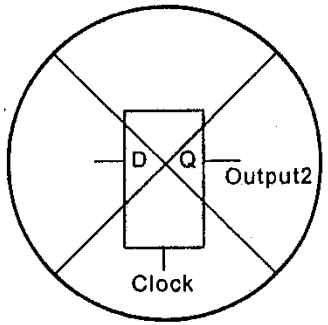
Don't need else when using default values.

The gt and eq signals assume '0' if they are not assigned a value later. As discussed earlier, assigning a signal multiple times inside a process can be error-prone. For synthesis, this should not be used in other context and should be considered as shorthand to satisfy the "assigning all signals in all branches" rule.

6.9 Accidental Synthesis of Inferred Latches

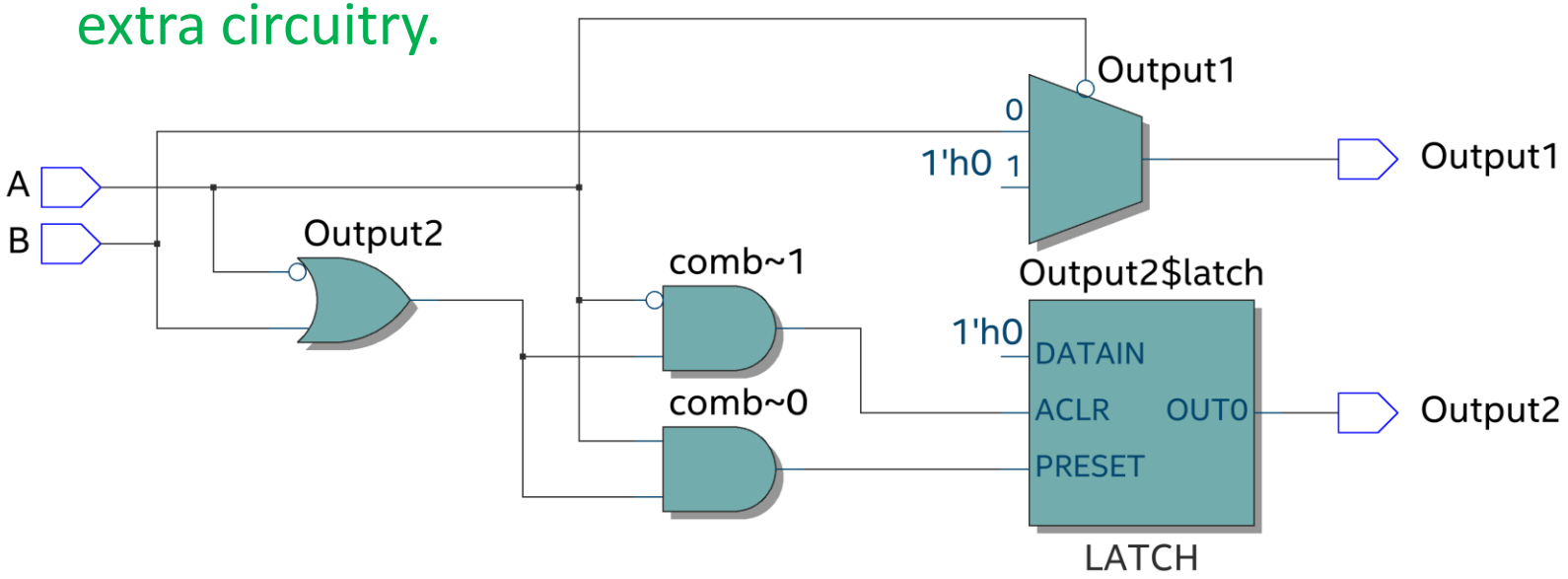
Here is a very common problem to be aware of when coding VHDL for synthesis. If a non-clocked process has any path that does not assign a value to an output, VHDL assumes you want to use the previous value. A level triggered latch is automatically generated or inferred by the synthesis tool to save the previous value. In many cases, this can cause serious errors in the design. Edge-triggered flip-flops should not be mixed with level-triggered latches in a design or serious timing problems will result. Typically this can happen in CASE statements or nested IF statements. In the following example, the signal OUTPUT2 infers a latch when synthesized. Assigning a value to OUTPUT2 in the last ELSE clause will eliminate the latch.

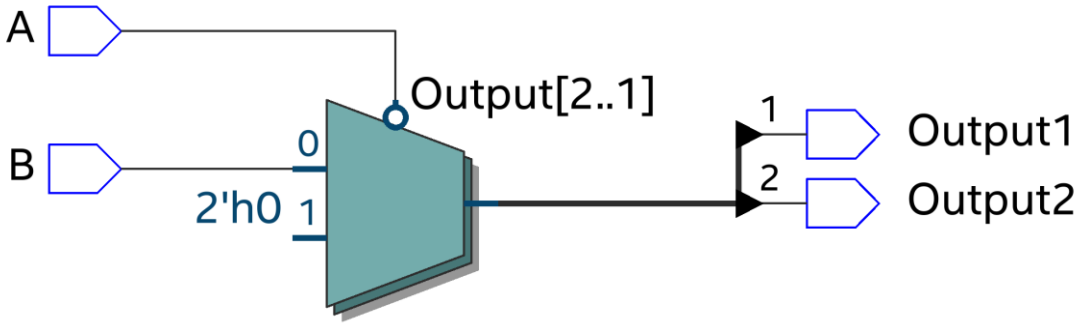
```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
ENTITY ilatch IS
        PORT( A, B                  : IN     STD_LOGIC;
              Output1, Output2      : OUT    STD_LOGIC  );
END ilatch;

ARCHITECTURE behavior OF ilatch IS
BEGIN
        PROCESS ( A, B )
        BEGIN
                IF A = '0' THEN
                   Output1 <= '0';
                   Output2 <= '0';
                ELSE
                   IF B = '1' THEN
                      Output1 <= '1';
                      Output2 <= '1';
                   ELSE                    -- Latch inferred since no value is assigned
                      Output1 <= '0';      --  to output2 in the else clause!
                   END IF;
                END IF;
        END PROCESS;
END behavior;
```



In std_logic combinatorial circuits, not using ELSE and WHEN OTHERS also generates inferred latches.

Using Quartus RTL viewer (Tools/Netlist Viewers/RTL viewer).
Output2 not specified, get inferred latch instead of MUX and extra circuitry.

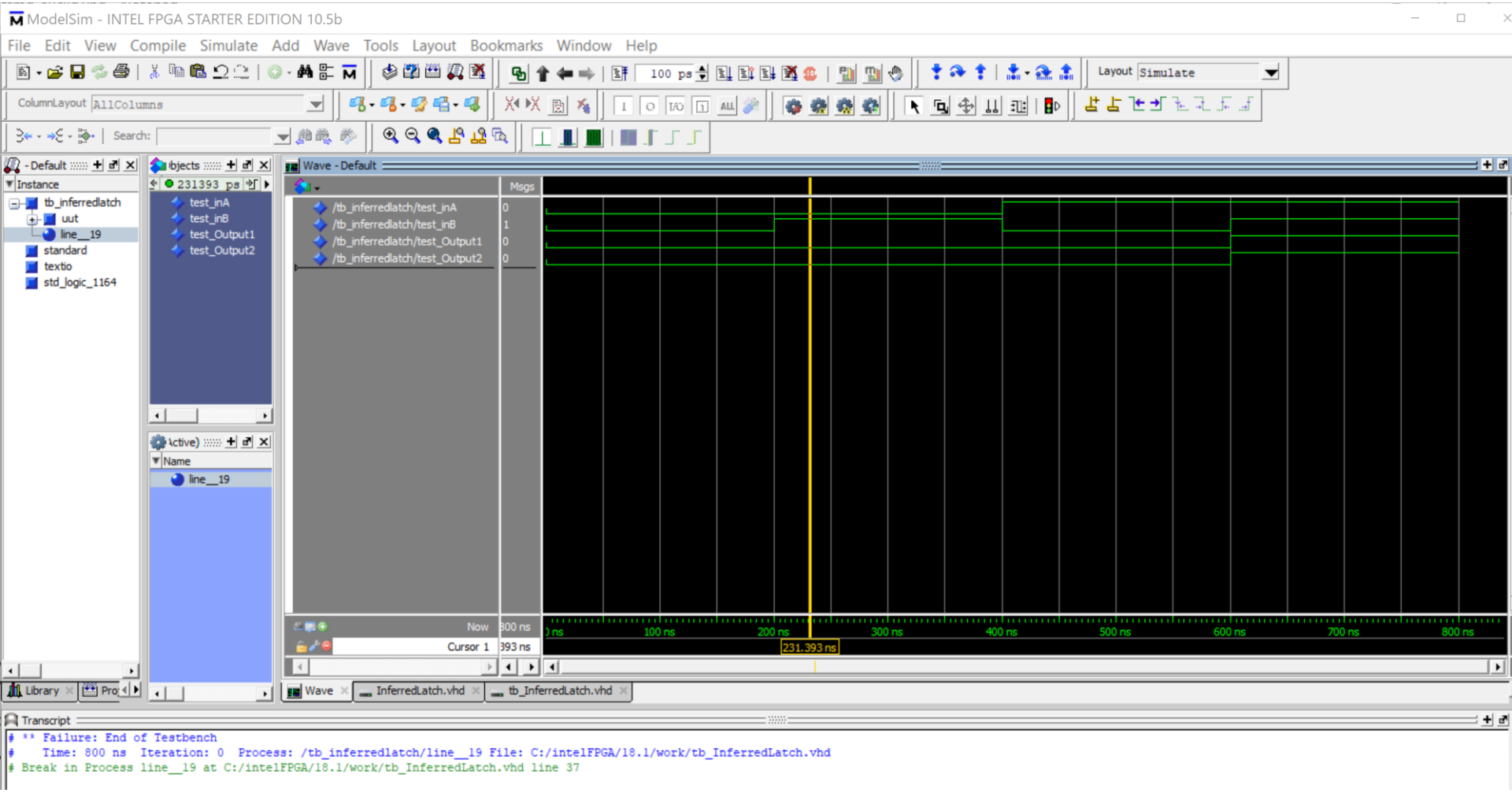




With Output2 specified, no inferred latch, just **two** stacked 2:1 MUXes.

Notice the inverter on the select line.
2'h0 is Verilog notation. It means 2-bit wide hexadecimal literal with value 0, i.e., "00".
Does the circuit operation match the vhdl code?

Using ModelSim testbench FUNCTIONAL simulation (not timing). Output2 delay due to inferred latch does not show up in a FUNCTIONAL simulation.



7. THE WITH-SELECT CONSTRUCT

The WITH...SELECT construct is another shorthand method for assigning a signal without using a PROCESS statement. As illustrated here, this construct has been used to define another 2:1 multiplexer.

```
WITH sel SELECT
      output <=input0 WHEN '0',
               input1 WHEN '1',
               input1 WHEN OTHERS;
```

WITH-SELECT can **never** be used inside a process. You might be tempted to use a WITH-SELECT as a concurrent statement nested inside an IF statement, but this is not allowed because the IF statement has to be in a PROCESS, and the WITH-SELECT can not.

```
ENTITY fig4_51 IS
PORT(
      a,b,c :IN BIT;              --a is most significant
      y     :OUT BIT);
END fig4_51;

ARCHITECTURE truth OF fig4_51 IS
      SIGNAL in_bits :BIT_VECTOR(2 DOWNTO 0);
      BEGIN
      in_bits <= a & b & c;     --concatenate input bits into bit_vector
            WITH in_bits SELECT
            y    <=      '0'  WHEN "000",      --Truth Table
                         '0'  WHEN "001",
                         '0'  WHEN "010",
                         '1'  WHEN "011",
                         '0'  WHEN "100",
                         '1'  WHEN "101",
                         '1'  WHEN "110",
                         '1'  WHEN "111";
END truth;
```

Don't use bit, use std_logic.
Then must add WHEN OTHERS

FIGURE 4-51 VHDL design file for Figure 4-7.

With-Select - illustrates putting multiple input combinations into one line of code

Examples We use two simple examples to demonstrate use of the conditional signal assignment statement. The first example is a priority encoder. The priority encoder has four requests, `r(4)`, `r(3)`, `r(2)`, and `r(1)`, which are grouped as a single 4-bit `r` input, and `r(4)` has the highest priority. The output is the binary code of the highest-order request.

Listing 3.3 Priority encoder using a selected signal assignment statement

```
architecture sel_arch of prio_encoder is
begin
    with r select
        pcode <= "100" when "1000"|"1001"|"1010"|"1011"|
                            "1100"|"1101"|"1110"|"1111",
                 "011" when "0100"|"0101"|"0110"|"0111",
                 "010" when "0010"|"0011",
                 "001" when "0001",
                 "000" when others;    -- r="0000"
end sel_arch;
```

The code exhaustively lists all possible combinations of the `r` signal and the corresponding output values. Note that the | symbol is used if the choice is more than one value.

"ORing" choices

WHEN - ELSE

32. SHORTHAND FOR A MULTIPLEXER

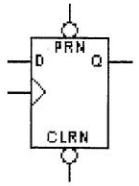
```
ARCHITECTURE behavior OF multiplexer1 IS
BEGIN

    output <= input0 WHEN (sel = '0') ELSE input1;

END behavior;
```

As stated before, concurrent signal assignments are shorthand methods for **PROCESS** statements. The **WHEN...ELSE** construct above performs the same function as previously illustrated with the **IF...ELSE** inside the **PROCESS**. Note that in the previous question using the **IF...ELSE**, the sensitivity list of the **PROCESS** included all signals affecting the assignment of *output*.

62. PORT MAPS

A **PORT MAP** is simply a way to designate which pins of a device are connected where. Assume for a moment that you are building a D flip-flop with the corresponding **ENTITY** statement. It turns out that this flip-flop is actually part of

```
ENTITY dff1 IS

   PORT( d          : IN  STD_LOGIC;
         clk        : IN  STD_LOGIC;
         clrn       : IN  STD_LOGIC;
         prn        : IN  STD_LOGIC;
         q          : OUT STD_LOGIC );

END dff1;
```



the Altera library of components. In the graphics editor, you might take the output of a NAND and connect it to the D input by dragging a line between the two. In VHDL, you use a **PORT MAP** to form the connections between components. As illustrated below, the **PORT MAP** looks very much like the **ENTITY** declaration of the componenent, and in fact, it connects signals in the local device to the input and output signals in the component's **ENTITY** statement. It simply needs to know what each line is connected to in the **ARCHITECTURE** body.

```
<label>:  dff PORT MAP( d     => <connected to>
                        clk   => <connected to>
                        clrn  => <connected to>
                        prn   => <connected to>
                        q     => <connected to>  );
```

Q: If X is to be connected to the d input of the flip-flop, how would this be designated in the **PORT MAP**.

Ans: `DFF1: dff PORT MAP( d => X, ...`

points to the right

VHDL Heirarchy – Creating a multicomponent top-level design without schematic entry

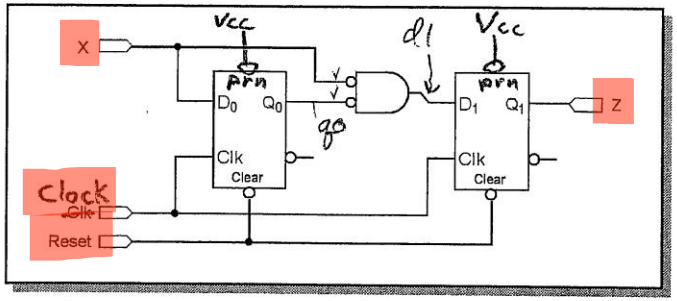
Advantage: Does not need proprietary software that only works with Altera, etc.

Disadvantage: Sometimes hard to visualize the circuit. It is good to document it with a schematic drawing with the same signal and pin names.

The first example in the next two slides implements the circuit diagram for a state machine. It also introduces using Altera library components (a D flip flop). The Altera component library statement must be added. In addition, you have to match Altera's component name and pin names exactly. See the Altera help file for the component you are interested in. Create a signal for each internal wire between components. A signal is not required for wires that go to pins. It is usually easier to create **simple logic parts** in the top-level design (the NOR gate below) instead of using their Altera component. Also note the creation of a high signal VCC.

Port Map Example –Using Altera Library FFs

The circuit diagram illustrates a state machine made up of two D flip-flops and a NOR gate. Given the ability to create D flip-flops, this state machine can be described in many different VHDL forms.



Instead of building each flip-flop, we will access the built-in component library, designate the **PORT MAP**, and build the state machine by designating the connections.

First, note that an additional **LIBRARY/USE** statement is needed to access the flip-flop component library.

Q: The **PORT MAP** for DFF1 shows that the last argument, "q," is connected to the output signal, _____. Does this match the graphical representation?

Ans: Z. Yes, the output signal Z is driven by the "q" of the second D flip-flop in the graphical circuit illustration.

```
LIBRARY altera;
USE altera.maxplus2.dff;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY state_mach1 IS

   PORT( X, clock, reset: IN  STD_LOGIC;
         Z              : OUT STD_LOGIC);

END state_mach1;

ARCHITECTURE behavior OF state_mach1 IS
   SIGNAL q0, d1, vcc : STD_LOGIC;
BEGIN
   vcc <= '1';
   d1  <= not(q0) and not(X);

DFF0: dff PORT MAP(  d    => X,
                     clk  => clock,
                     clrn => reset,
                     prn  => vcc,
                     q    => q0     );

DFF1: dff PORT MAP(  d    => d1,
                     clk  => clock,
                     clrn => reset,
                     prn  => vcc,
                     q    => Z      );

END behavior;
```

Connections in an Altera **PORT MAP** statement can only be made with explicitly defined signals. The result of the NOR equation, therefore, has to be assigned to *d1* and then *d1* can be connected to *d* via the **PORT MAP**.

Signals to the left of the connection operator (=>) in the **PORT MAP** cannot be referenced directly in the rest of the **ARCHITECTURE** body, since these are the **ENTITY** signal names of the component. Also, a signal simply tied to *d* (instead of *d1*) would be ambiguous since there are two flip-flops! The signals to the right of the connection operator (=>) are local to the rest of the **ARCHITECTURE** body.

```
LIBRARY altera;
USE altera.maxplus2.dff;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY state_mach1 IS

   PORT( X, clock, reset: IN  STD_LOGIC;
         Z              : OUT STD_LOGIC);

END state_mach1;

ARCHITECTURE behavior OF state_mach1 IS
   SIGNAL q0, d1, vcc : STD_LOGIC;
BEGIN
   vcc <= '1';
   d1  <= not(q0) and not(X);

DFF0: dff PORT MAP( d    => X,
                    clk  => clock,
                    clrn => reset,
                    prn  => vcc,
                    q    => q0      );

DFF1: dff PORT MAP( d    => d1,
                    clk  => clock,
                    clrn => reset,
                    prn  => vcc,
                    q    => Z       );
END behavior;
```

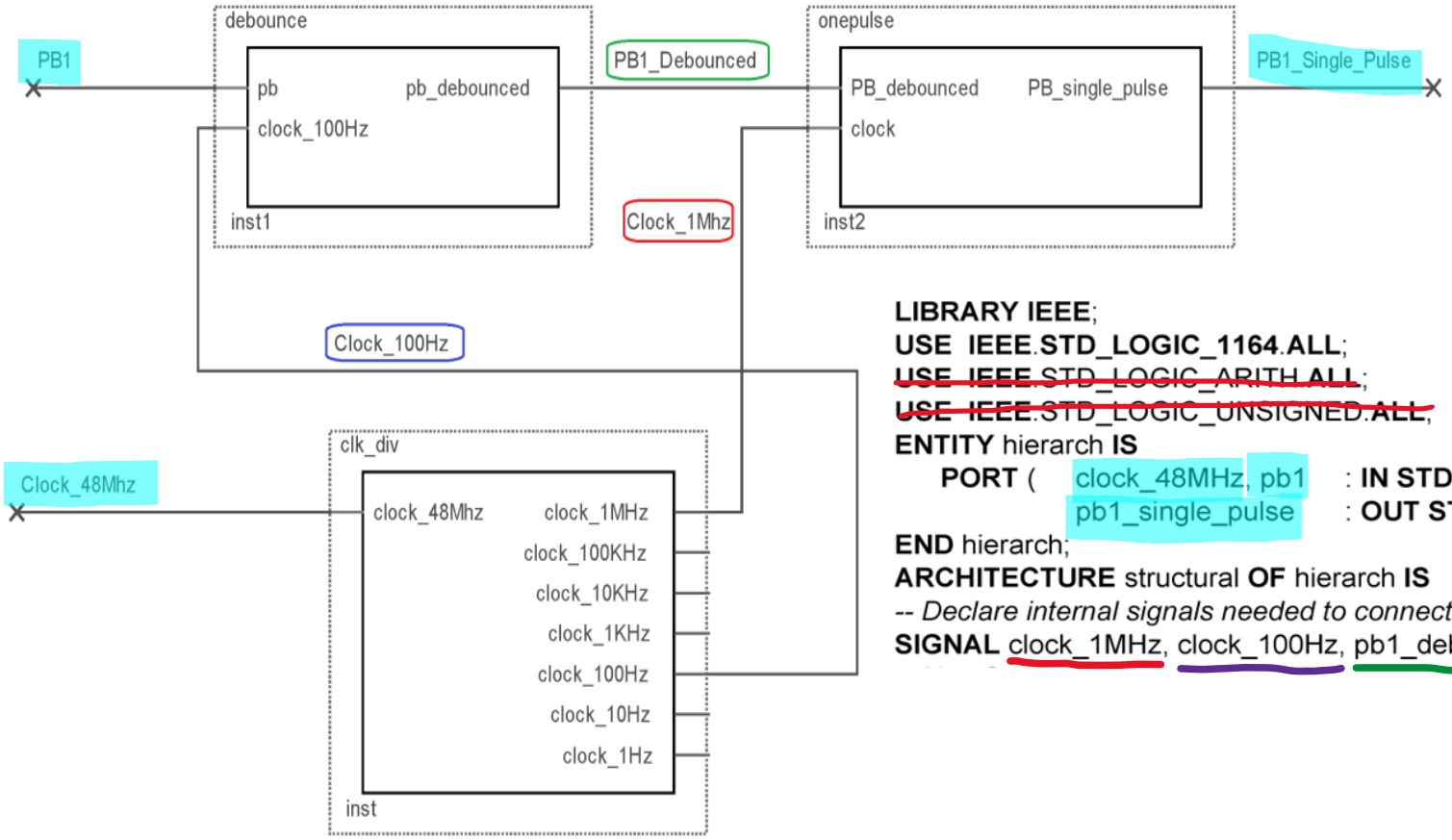
Q: Could you have declared your own D flip-flop instead of using the library component?

The second example in the next two slides connects three components created by Georgia Tech and included with their Rapid Prototyping SOPC edition lab book . It shows how to add the component definitions using the old way (VHDL-87 standard) which is required when the component is not in the Altera library. Note that the definitions inside each component are the same as the port map for the component entity. Some designs use more than one "instance" of the component. Therefore you must make up a unique name for each instance. Notice how the use of informative signal names is important (I find their using the same name as the component pin confusing) when writing the port statements at the end of the file. But there also is usually a need for documentation with a schematic.

Using old VHDL-87 component declarations.
How to convert schematic to VHDL code

1) Port Statement for external PINS

2) Make each INTERNAL wire a signal.




```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
USE  IEEE.STD_LOGIC_ARITH.ALL;
USE  IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY hierarch IS
    PORT (  clock_48MHz, pb1     : IN STD_LOGIC;
            pb1_single_pulse     : OUT STD_LOGIC);
END hierarch;
ARCHITECTURE structural OF hierarch IS
-- Declare internal signals needed to connect submodules
SIGNAL clock_1MHz, clock_100Hz, pb1_debounced : STD_LOGIC;
```

Figure 6.2 Schematic of Hierarchical Design Example

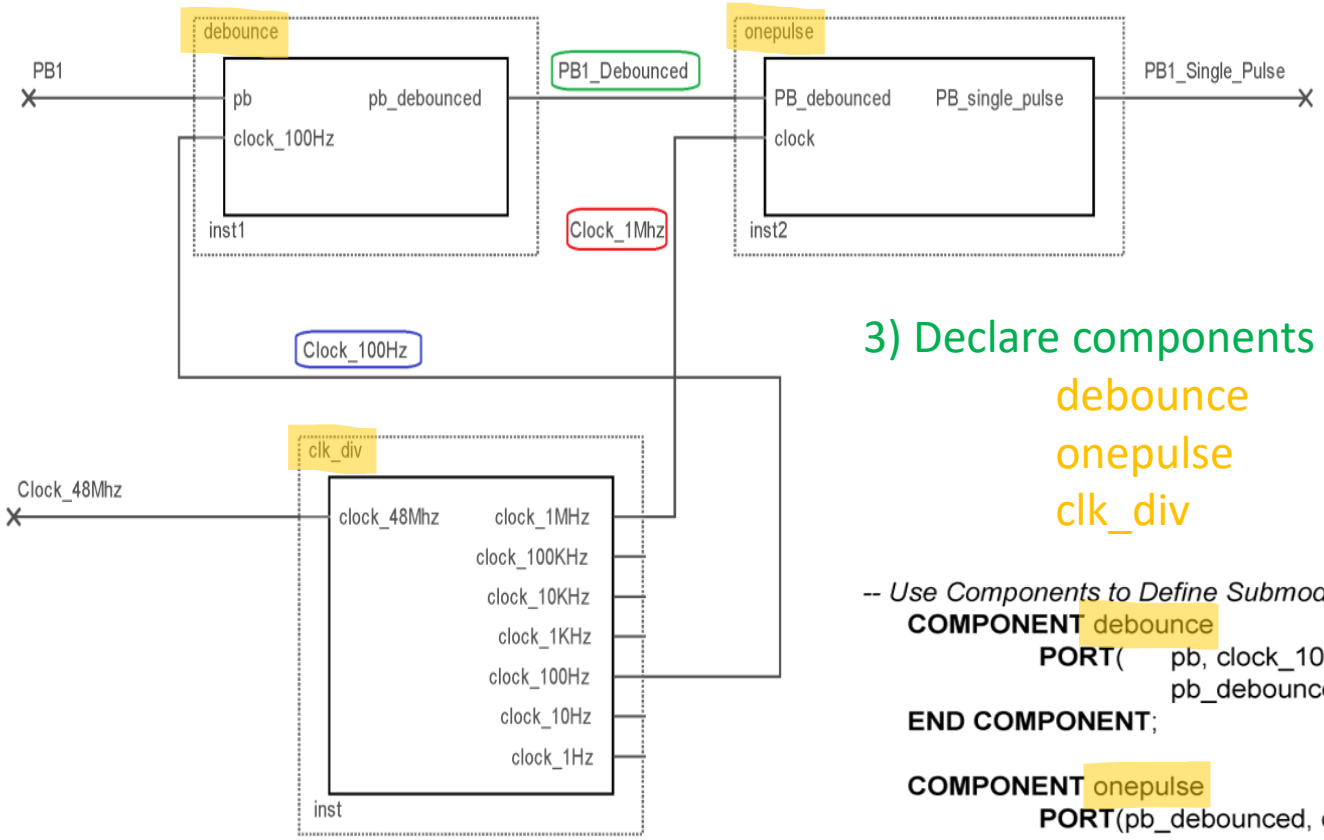


Figure 6.2 Schematic of Hierarchical Design Example

3) Declare components (copy port statements)

debounce

onepulse

clk_div

```
-- Use Components to Define Submodules and Parameters
    COMPONENT debounce
            PORT(   pb, clock_100Hz     : IN    STD_LOGIC;
                    pb_debounced        : OUT   STD_LOGIC);
    END COMPONENT;

    COMPONENT onepulse
            PORT(pb_debounced, clock    : IN    STD_LOGIC;
             pb_single_pulse            : OUT   STD_LOGIC);
    END COMPONENT;

    COMPONENT clk_div
            PORT(   clock_48MHz         : IN    STD_LOGIC;
                    clock_1MHz          : OUT   STD_LOGIC;
                    clock_100kHz        : OUT   STD_LOGIC;
                    clock_10kHz         : OUT   STD_LOGIC;
                    clock_1kHz          : OUT   STD_LOGIC;
                    clock_100Hz         : OUT   STD_LOGIC;
                    clock_10Hz          : OUT   STD_LOGIC;
                    clock_1Hz           : OUT   STD_LOGIC);
    END COMPONENT;
BEGIN
```

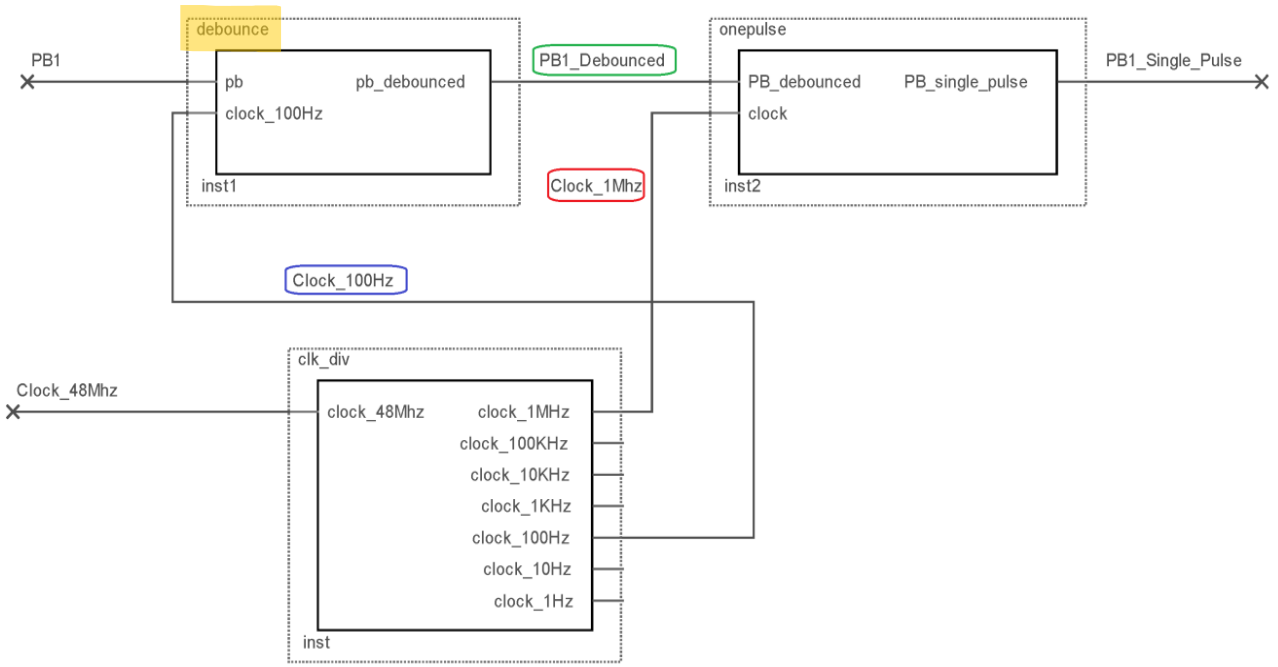


Figure 6.2 Schematic of Hierarchical Design Example

Note: You must make up a unique name for each instance. For example: debounce1

4) Connect components

```
-- Use Port Map to connect signals between components in the hierarchy
debounce1 : debounce PORT MAP  (pb => pb1, clock_100Hz = >clock_100Hz,
                                pb_debounced = >pb1_debounced);

prescalar : clk_div  PORT MAP (clock_48MHz = >clock_48MHz,
                               clock_1MHz =>clock_1MHz,
                          clock_100hz = >clock_100hz);

single_pulse : onepulse PORT MAP (pb_debounced = >pb1_debounced,
                                  clock => clock_1MHz,
                                   pb_single_pulse => pb1_single_pulse);
END structural;
```

The third example in the next three slides connects two components (1-bit comparators) to make a 2-bit comparator. It shows how to add the component definitions using the new way (VHDL-93 standard) which is required when the component is not in the Altera library. Some designs use more than one "instance" of the component. Therefore you must make up a unique name for each instance, the same as for the old way. However, instead of using the COMPONENT keyword and copying the port map, you tell the VHDL compiler where the component is stored, by using the ENTITY keyword followed by which library holds the component. The components are usually in the default library, which is named WORK.

The 4th slide shows how to implement the 1-bit comparator component the old way (VHDL-87).

1.3 STRUCTURAL DESCRIPTION

A digital system is frequently composed of several smaller subsystems. This allows us to build a large system from simpler or predesigned components. VHDL provides a mechanism, known as *component instantiation*, to perform this task. This type of code is called *structural description*.

An alternative to the design of the 2-bit comparator of Section 1.2.6 is to utilize the previously constructed 1-bit comparators as the building blocks. The diagram is shown in Figure 1.2, in which two 1-bit comparators are used to check the two individual bits and their results are fed to an and cell. The aeqb signal is asserted only when the two bits are equal.

1-bit equality comparator eq1 (sum of products from truth table)

```
library ieee;
use ieee.std_logic_1164.all;
entity eq1 is
   port(
      i0, i1: in std_logic;
      eq: out std_logic
   );
end eq1;

architecture sop_arch of eq1 is
   signal p0, p1: std_logic;
begin
   -- sum of two product terms
   eq <= p0 or p1;
   -- product terms
   p0 <= (not i0) and (not i1);
   p1 <= i0 and i1;
end sop_arch;
```

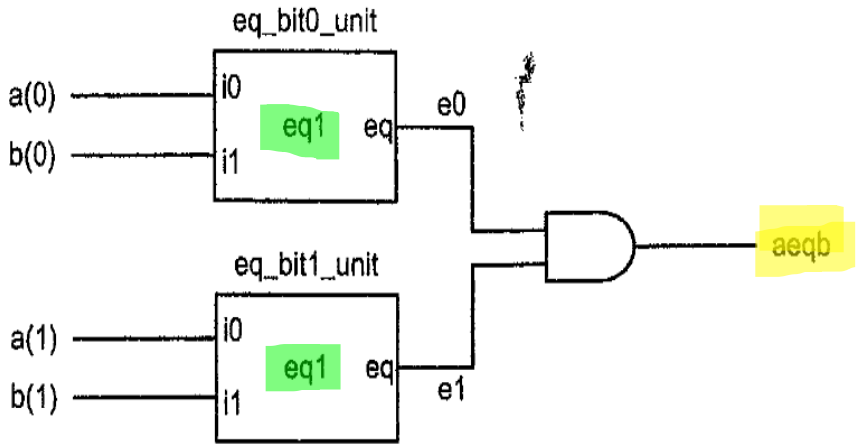


Figure 1.2 Construction of a 2-bit comparator from 1-bit comparators.

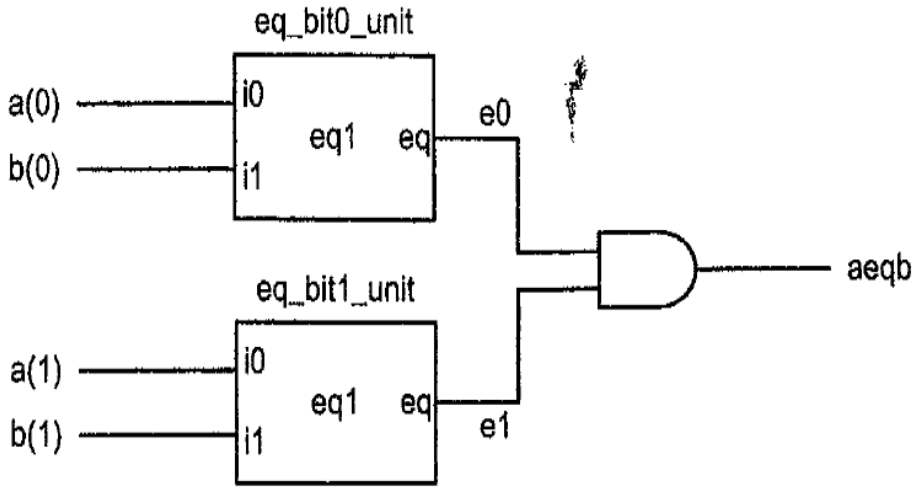


Figure 1.2 Construction of a 2-bit comparator from 1-bit comparators.

The corresponding code is shown in Listing 1.3. Note that the entity declaration is the same and thus is not included.

This is the new way to add components

Listing 1.3 Structural description of a 2-bit comparator

```
architecture struc_arch of eq2 is
   signal e0, e1: std_logic;
begin
   -- instantiate two 1-bit comparators
   eq_bit0_unit: entity work.eq1(sop_arch)
      port map(i0=>a(0), i1=>b(0), eq=>e0);
   eq_bit1_unit: entity work.eq1(sop_arch)
      port map(i0=>a(1), i1=>b(1), eq=>e1);
   -- a and b are equal if individual bits are equal
   aeqb <= e0 and e1;
end struc_arch;
```

The code includes two component instantiation statements, whose syntax is:

```
unit_label: entity lib_name.entity_name(arch_name)
    port map(
        formal_signal=>actual_signal,
        formal_signal=>actual_signal,
        . . .
    );
```

The first portion of the statement specifies which component is used. The unit_label term gives a unique id for an instance, the lib_name term indicates where (i.e., which library) the component resides, and the entity_name and arch_name terms indicate the names of the entity and architecture. The arch_name term is optional. If it is omitted, the last compiled architecture body will be used. The second portion is port mapping, which indicates the connection between *formal signals*, which are I/O ports declared in a component's entity declaration, and *actual signals*, which are the signals used in the architecture body.

The first component instantiation statement is

```
eq_bit0_unit: entity work.eq1(sop_arch)
    port map(i0=>a(0), i1=>b(0), eq=>e0);
```

The work library is the default library in which the compiled entity and architecture units are stored, and eq1 and sop_arch are the names of the entity and architecture defined in Listing 1.1. The port mapping reflects the connections shown in Figure 1.2. The component instantiation statement is also a concurrent statement and represents a circuit that is encompassed in a "black box" whose function is defined in another module.

This example demonstrates the close relationship between a block diagram and code. The code is essentially a textual description of a schematic. Although it is a clumsy way for humans to comprehend a diagram, it puts all representations into a single HDL framework. The Xilinx ISE package includes a simple schematic editor utility that can perform schematic capture in graphic format and then convert the diagram into an HDL structural description.

The component instantiation statement is added in VHDL 93. Older codes may use the mechanism in VHDL 87, in which a component must first be declared (i.e., made known) and then used. The code in this format is shown in Listing 1.4.

Note that the original clause,

```
eq_bit0_unit: entity work.eq1(sop_arch)
```

is replaced by a clause with the declared component name

```
eq_bit0_unit: eq1
```

However, the **port map** statements are the same in both ways.

Listing 1.4 Structural description with VHDL-87

```
architecture vhd_87_arch of eq2 is
    -- component declaration using the old VHDL-87 way.
    component eq1
        port(
            i0, i1: in std_logic;
            eq: out std_logic
        );
    end component;
    signal e0, e1: std_logic;
begin
    -- instantiate two 1-bit comparators
    eq_bit0_unit: eq1    -- use the declared name, eq1
        port map(i0=>a(0), i1=>b(0), eq=>e0);
    eq_bit1_unit: eq1    -- use the declared name, eq1
        port map(i0=>a(1), i1=>b(1), eq=>e1);
    -- a and b are equal if individual bits are equal
    aeqb <= e0 and e1;
end vhd_87_arch;
```

Handling Unused Signals with Port Maps

Reference www.NandLand.com "Dealing with unused signals in VHDL"

1. A signal that you created that is not used in your design:
   Quartus will generate a warning and not implement it.

2. Instantiating a component with an input signal that is not used:
   for STD_LOGIC, connect it to '0'
   for STD_LOGIC_VECTOR, connect them to (OTHERS => '0').
   Note that OTHERS => '0' is shorthand for "00000000…",
   and you don't have to know ahead of time how many of bits there are.

3. Instantiating a component with an output signal that is not used:
   Connect it to OPEN.

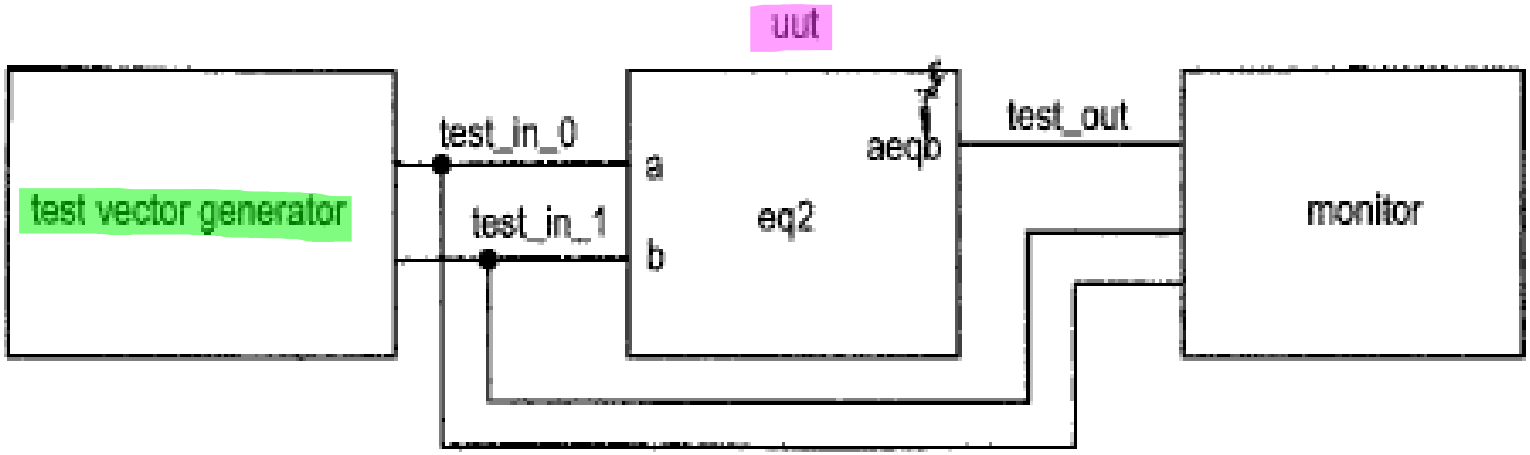


Figure 1.3 Testbench for a 2-bit comparator.

1.4 TESTBENCH

for combinatorial circuits
(see the sequential vhdl ppt for that testbench)

After code is developed, it can be *simulated* in a host compute to verify the correctness of the circuit operation and can be *synthesized* to a physical device. Simulation is usually performed within the same HDL framework. We create a special program, known as a *testbench*, to mimic a physical lab bench. The sketch of a 2-bit comparator testbench program is shown in Figure 1.3. The `uut` block is the unit under test, the `test vector generator` block generates testing input patterns, and the `monitor` block examines the output responses.

Listing 1.5 Testbench for a 2-bit comparator

```
library ieee;
use ieee.std_logic_1164.all;
entity eq2_testbench is
end eq2_testbench;

architecture tb_arch of eq2_testbench is
    signal test_in0, test_in1: std_logic_vector(1 downto 0);
    signal test_out: std_logic;
begin
    -- instantiate the circuit under test
    uut: entity work.eq2(struc_arch)
        port map(a=>test_in0, b=>test_in1, aeqb=>test_out);
    -- test vector generator
    process
    begin
        -- test vector 1
        test_in0 <= "00";
        test_in1 <= "00";
        wait for 200 ns;
        -- test vector 2
        test_in0 <= "01";
        test_in1 <= "00";
        wait for 200 ns;
```

and connect test signals

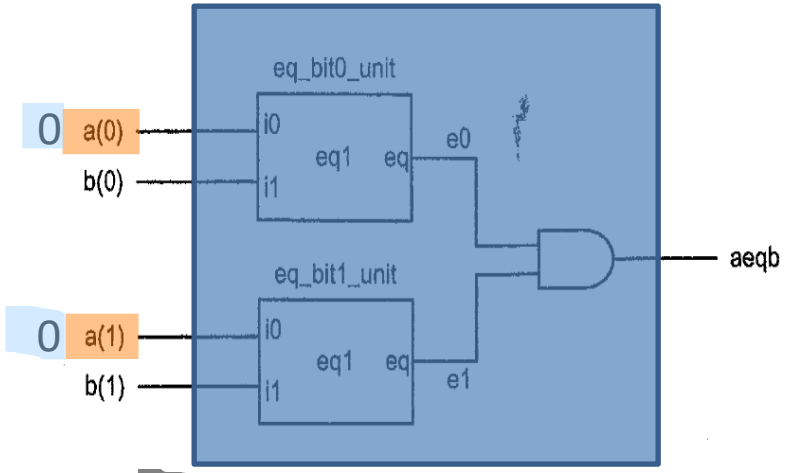


Figure 1.2 Construction of a 2-bit comparator from 1-bit comparators.

```
        -- test vector 3
        test_in0 <= "01";
        test_in1 <= "11";
        wait for 200 ns;
        -- test vector 4
        test_in0 <= "10";
        test_in1 <= "10";
        wait for 200 ns;
        -- test vector 5
        test_in0 <= "10";
        test_in1 <= "00";
        wait for 200 ns;
        -- test vector 6
        test_in0 <= "11";
        test_in1 <= "11";
        wait for 200 ns;
        -- test vector 7
        test_in0 <= "11";
        test_in1 <= "01";
        wait for 200 ns;
    end process;
end tb_arch;
```

All combinations of inputs should be tested whenever possible (16 in this case). However, in real-world applications, it is often not feasible. Problems often show up after the circuit is in actual use.

assert false
report "Simulation Completed“
severity FAILURE;

The code consists of a component instantiation statement, which creates an instance of a 2-bit comparator, and a process statement, which generates a sequence of test patterns.

The process statement is a special VHDL construct in which the operations are performed sequentially. Each test pattern is generated by three statements. For example,

```
-- test vector 2
test_in0 <= "01";
test_in1 <= "00";
wait for 200 ns;
```

The first two statements specify the values for the `test_in0` and `test_in1` signals, and the third indicates that the two values will last for 200 ns.

The code has no monitor. We can observe the input and output waveforms on a simulator's display, which can be treated as a "virtual logic analyzer." The simulated timing diagram of this testbench is shown in Figure 2.16.

Writing code for a comprehensive test vector generator and a monitor requires detailed knowledge of VHDL and is beyond the scope of this book. This listing can serve as a testbench template for other combinational circuits. We can substitute the `uut` instance and modify the test patterns according to the new circuit.

49. ASSIGNMENTS

Assignments (<=) of signals in the **ARCHITECTURE** are intimately tied to the size of the target signal. You can't make a 3-bit **STD_LOGIC_VECTOR** and assign it to a 2-bit **STD_LOGIC_VECTOR**, nor vice versa. You can store part of the 3-bit **STD_LOGIC_VECTOR** into the 2-bit **STD_LOGIC_VECTOR**, but your assignment must be explicit!

Suppose that the following signals have been declared:

```
SIGNAL A, B : STD_LOGIC_VECTOR( 3 DOWNTO 0 );
SIGNAL C    : STD_LOGIC_VECTOR( 2 DOWNTO 0 );
SIGNAL D    : STD_LOGIC;
```

Q: Can you assign A <= C?

52. CONCATENATION OPERATOR (&)

The concatenation operator joins two signals or vectors into another vector. If you specified `D & A`, you would be designating D – A(3) – A(2) – A(1) – A(0) as a single vector.

Q: How would you set *new_vector* equal to A(2), C(1), C(0), D using the concatenate symbol?

Ans: There are a couple of ways:

```
new_vector <= A(2) & C(1) & C(0) & D;

new_vector <= A(2) & C( 1 DOWNTO 0 ) & D;
```

Concatenating input bits into a vector

```
ENTITY fig4_61 IS
PORT( p, q, r      :IN bit;                    --declare 3 bits input
      s            :OUT BIT);
END fig4_61;

ARCHITECTURE copy OF fig4_61 IS
SIGNAL status      :BIT_VECTOR (2 downto 0);
BEGIN
   status <= p & q & r;                        --link bits in order
   PROCESS (status)
      BEGIN
         CASE status IS
            WHEN "100" =>  s <= '0';
            WHEN "101" =>  s <= '0';
            WHEN "110" =>  s <= '0';
            WHEN OTHERS => s <= '1';
         END CASE;
      END PROCESS;
END copy;
```

Don't use bit, use std_logic.

Shift register using concatenation of vectors

Chu Ch. 3 p. 39 FPGA Prototyping

Implementation of the concatenation operator involves reconnection of the input and output signals and only requires "wiring."

One major application of the & operator is to perform shifting operations. Although both VHDL standard and numeric_std package define shift functions, they sometimes cannot be synthesized automatically. The & operator can be used for shifting a signal for a fixed amount, as shown in the following example:

```
signal a: std_logic_vector(7 downto 0);
signal rot, shl, sha: std_logic_vector(7 downto 0);
. . .
-- rotate a to right 3 bits
rot <= a(2 downto 0) & a(8 downto 3);
-- shift a to right 3 bits and insert 0 (logic shift)
shl <= "000" & a(8 downto 3);
-- shift a to right 3 bits and insert MSB
-- (arithmetic shift) 2's complement stays correct even when negative
sha <= a(8) & a(8) & a(8) & a(8 downto 3);
```

An additional routing circuit is needed if the amount of shifting is not fixed. The design of a barrel shifter is discussed in Section 3.7.3.

Table 3.1 Operators and data types of VHDL-93 and IEEE `std_logic_1164` package

Operator	Description	Data type of operands	Data type of result
`a ** b`	exponentiation	`integer`	`integer`
`a * b`	multiplication		
`a / b`	division	*integer type for constants and array boundaries, not synthesis*	
`a + b`	addition		
`a - b`	subtraction		
`a & b`	concatenation	1-D array, element	1-D array
`a = b`	equal to	any	`boolean`
`a /= b`	not equal to		
`a < b`	less than	scalar or 1-D array	`boolean`
`a <= b`	less than or equal to		
`a > b`	greater than		
`a >= b`	greater than or equal to		
not `a`	negation	`boolean`, `std_logic`, `std_logic_vector`	same as operand
`a` **and** `b`	and		
`a` **or** `b`	or		
`a` **xor** `b`	xor		

IEEE numeric_std package The IEEE `numeric_std package` adds two new data types, `unsigned and signed`, and defines the relational and arithmetic operators over the new data types (known as *operator overloading*). The `unsigned and signed` data types are defined as an array with elements of the `std_logic` data type. The array is interpreted as the binary representation of unsigned or signed integers. We have to add an additional use statement to invoke the package:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numric_std.all;    -- invoke numeric_std package
```

The synthesizable overloaded operators are summarized in Table 3.2.

Multiplication is a complicated operation, and synthesis of the multiplication operator * depends on synthesis software and target device technology. Xilinx Spartan-3 FPGA family contains prefabricated combinational multiplier blocks. The Xilinx XST software can infer these blocks during synthesis, and thus the multiplication operator can be used in HDL code. The XCS200 device of the S3 board consists of twelve 18-by-18 multiplier blocks. While the synthesis of the multiplication operator is supported, we need to be aware of the limitation on the number and input width of these blocks and use them with care.

Xilinx specific

Integer data type - positive and negative integers (and zero)
Natural data type - positive integers and zero.
Signed uses two's – complement. Use with integers.
Use unsigned with natural.

Table 3.2 Overloaded operators and data types in the IEEE numeric_std package

Overloaded operator	Description	Data type of operands	Data type of result
a * b	arithmetic	unsigned, natural	unsigned
a + b	operation	signed, integer	signed
a - b			
a = b			
a /= b			
a < b	relational	unsigned, natural	boolean
a <= b	operation	signed, integer	boolean
a > b			
a >= b			

3.2.1 Relational operators

Six relational operators are defined in the VHDL standard: = (equal to), /= (not equal to), < (less than), <= (less than or equal to), > (greater than), and >= (greater than or equal to). These operators compare operands of the same data type and return a value of the `boolean` data type. In this book, we don't use the `boolean` data type directly, but embed it in routing constructs. This is discussed in Sections 3.3 and 3.5. During synthesis, comparators are inferred for these operators.

3.2.2 Arithmetic operators

In the VHDL standard, arithmetic operations are defined for the `integer` data type and for the `natural` data type, which is a subtype of `integer` containing zero and positive integers. We usually prefer to have more control in synthesis and define the exact number of bits and format (i.e., signed or unsigned). The IEEE `numeric_std` package is developed for this purpose. In this book, we use the `integer` and `natural` data types for constants and array boundaries but not for synthesis.

Type conversion Because VHDL is a strongly typed language, `std_logic_vector`, `unsigned`, and `signed` are treated as different data types even when all of them are defined as an array with elements of the `std_logic` data type. A *conversion function* or *type casting* is needed to convert signals of different data types. The conversion is summarized in Table 3.3. Note that the `std_logic_vector` data type is not interpreted as a number and thus cannot be converted directly to an integer, and vice versa.

Table 3.3 Type conversions between `std_logic_vector` and numeric data types

Data type of `a`	To data type	Conversion function/type casting
`unsigned, signed`	`std_logic_vector`	`std_logic_vector(a)`
`signed, std_logic_vector`	`unsigned`	`unsigned(a)`
`unsigned, std_logic_vector`	`signed`	`signed(a)`
`unsigned, signed`	`integer`	`to_integer(a)`
`natural`	`unsigned`	`to_unsigned(a, size)`
`integer`	`signed`	`to_signed(a, size)`

The following examples illustrate the common mistakes and remedies for type conversion. Assume that some signals are declared as follows:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
. . .
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u5, u6, u7: unsigned(3 downto 0);
```

Let us first consider the following assignment statements:

```
u1 <= s1;   -- not ok, type mismatch
u2 <= 5;    -- not ok, type mismatch
s2 <= u3;   -- not ok, type mismatch
s3 <= 5;    -- not ok, type mismatch
```

They are all invalid because of type mismatch. The right-hand-side expression must be converted to the data type of the left-hand-side signal:

```
u1 <= unsigned(s1);          -- ok, type casting
u2 <= to_unsigned(5,4);      -- ok, conversion function
s2 <= std_logic_vector(u3);  -- ok, type casting
s3 <= std_logic_vector(to_unsigned(5,4)); -- ok
```

Use 4 bits

Use 4 bits

Note that two type conversions are needed for the last statement.

Let us consider statements that involve arithmetic operations. The following statements are valid since the + operator is defined with the unsigned and natural types in the IEEE numeric_std package.

```
u4 <= u2 + u1;   -- ok, both operands unsigned
u5 <= u2 + 1;    -- ok, operands unsigned and natural
```

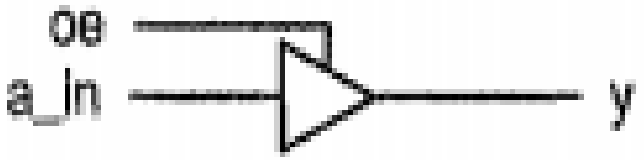
On the other hand, the following statements are invalid since no overloaded arithmetic operation is defined for the std_logic_vector data type:

```
s5 <= s2 + s1; -- not ok, + undefined over the types
s6 <= s2 + 1;  -- not ok, + undefined over the types
```

To fix the problem, we must convert the operands to the unsigned (or signed) data type, perform addition, and then convert the result back to the std_logic_vector data type. The revised code becomes

```
s5 <= std_logic_vector(unsigned(s2) + unsigned(s1)); -- ok
s6 <= std_logic_vector(unsigned(s2) + 1);            -- ok
```

***Nonstandard* arithmetic packages** There are several non-IEEE arithmetic packages, which are `std_logic_arith`, `std_logic_unsigned`, and `std_logic_signed`. The `std_logic_arith` package is similar to the `numeric_std` package. The other two packages do not introduce any new data type but define overloaded arithmetic operators over the `std_logic_vector` data type. This approach eliminates the need for data conversion. Although using these packages seems to be less cumbersome initially, it is not good practice since these packages are not a part of IEEE standards and may introduce a compatibility problem in the long run. We do not use these packages in this book.



oe	y
0	Z
1	a_in

Figure 3.1 Symbol and functional table of a tri-state buffer.

'Z' value of std_logic The `std_logic` data type has a value of `'Z'`, which implies *high impedance* or an open circuit. It is not a normal logic value and can only be synthesized by a *tri-state buffer*. The symbol and function table of a tri-state buffer are shown in Figure 3.1. Operation of the buffer is controlled by an enable signal, `oe` ("output enable"). When it is `'1'`, the input is passed to output. On the other hand, when it is `'0'`, the y output appears to be an open circuit. The code of the tri-state buffer is

```
y <= a_in when oe='1' else 'Z';
```

3.2.4 Summary

Because of the nature of a strongly typed language, the data type frequently confuses a new VHDL user. Since this book is focused on synthesis, only a small set of data types and operators are needed. Their uses can be summarized as follows:

- Use the `std_logic` and `std_logic_vector` data types in entity port declaration and for the internal signals that involve no arithmetic operations.
- Use the 'Z' value only to infer a tri-state buffer.
- Use the IEEE `numeric_std` package and its `unsigned` or `signed` data types for the internal signals that involve arithmetic operation.
- Use the data type casting or conversion functions in Table 3.3 to convert signals and expressions among the `std_logic_vector` and various numerical data types.
- Use VHDL's built-in `integer` data type and arithmetic operators for constant and array boundary expressions, but not for synthesis (i.e., not used as a data type for a signal).

- Embed the result of a relational operation, which is in the `boolean` data type, in routing constructs (discussed in Section 3.3).
- Use a user-defined two-dimensional data type for two-dimensional storage array (discussed in Section 4.2.3).
- Use a user-defined *enumerate data type* for the symbolic states of a finite state machine (discussed in Chapter 5).

3.6.1 Constants

HDL code frequently uses constant values in expressions and array boundaries. One good design practice is to replace the "hard literals" with symbolic constants. It makes code clear and helps future maintenance and revision. The constant declaration can be included in the architecture's declaration section, and it syntax is

```
constant const_name: data_type := value_expression;
```

For example, we can declare two constants as

```
constant DATA_BIT: integer := 8;
constant DATA_RANGE: integer := 2**DATA_BIT - 1;
```

The constant expression is evaluated during preprocessing and thus requires no physical circuit. In this book, we use capital letters for constants.

The use of a constant can best be explained by an example. Assume that we want to design an adder with the carry-out bit. One way to do it is to extend the input by 1 bit and then perform regular addition. The MSB of the summation becomes the carry-out bit. The code is shown in Listing 3.9.

Listing 3.9 Adder using a hard literal

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity add_w_carry is
    port(
        a, b: in std_logic_vector(3 downto 0);
        cout: out std_logic;
        sum: out std_logic_vector(3 downto 0)
    );
end add_w_carry;

architecture hard_arch of add_w_carry is
    signal a_ext, b_ext, sum_ext: unsigned(4 downto 0);
begin
    a_ext <= unsigned('0' & a);      -- extend vector by 1 bit
    b_ext <= unsigned('0' & b);
    sum_ext <= a_ext + b_ext;
    sum <= std_logic_vector(sum_ext(3 downto 0));
    cout <= sum_ext(4);
end hard_arch;
```

The code is for a 4-bit adder. Hard literals, such as 3 and 4, are used for the ranges, as in `unsigned(4 downto 0)` and `sum_ext(3 downto 0)`, and the MSB, as in `sum_ext(4)`. If we want to revise the code for an 8-bit adder, these literals have to be modified manually. This will be a tedious and error-prone process if the code is complex and the literals are referred to in many places.

To improve the readability, we can use a symbolic constant, N, to represent the number of bits of the adder. The revised architecture body is shown in Listing 3.10.

Listing 3.10 Adder using a constant

```
architecture const_arch of add_w_carry is
   constant N: integer := 4;
   signal a_ext, b_ext, sum_ext: unsigned(N downto 0);
begin
   a_ext <= unsigned('0' & a);
   b_ext <= unsigned('0' & b);
   sum_ext <= a_ext + b_ext;
   sum <= std_logic_vector(sum_ext(N-1 downto 0));
   cout <= sum_ext(N);
end const_arch;
```

The constant makes the code easier to understand and maintain.

3.6.2 Generics

VHDL provides a construct, known as a *generic*, to pass information into an entity and component. Since a generic cannot be modified inside the architecture, it functions somewhat like a constant. A generic is declared inside an entity declaration, just before the port declaration:

generic allows changing number of pins and also signal lines in the current design as well as when the design is used as a component (generic mapping)

```
entity entity_name is
    generic(
        generic_name: data_type := default_values;
        generic_name: data_type := default_values;
        . . .
        generic_name: data_type := default_values
    )
    port(
        port_name: mode data_type;
        . . .
    );
end entity_name;
```

For example, the previous adder code can be modified to use the adder width as a generic, as shown in Listing 3.11.

Listing 3.11 Adder using a generic

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity gen_add_w_carry is
    generic(N: integer:=4);
    port(
        a, b: in std_logic_vector(N-1 downto 0);
        cout: out std_logic;
        sum: out std_logic_vector(N-1 downto 0)
    );
end gen_add_w_carry;

architecture arch of gen_add_w_carry is
    signal a_ext, b_ext, sum_ext: unsigned(N downto 0);
begin
    a_ext <= unsigned('0' & a);
    b_ext <= unsigned('0' & b);
    sum_ext <= a_ext + b_ext;
    sum <= std_logic_vector(sum_ext(N-1 downto 0));
    cout <= sum_ext(N);
end arch;
```

generic allows changing number of pins

generic also allows changing number of signal lines

generic also allows changing number of signal lines

The N generic is declared in line 5 with a default value of 4. After N is declared, it can be used in the port declaration and architecture body, just like a constant.

If the adder is later used as a component in other code, we can assign the desired value to the generic in component instantiation. This is known as *generic mapping*. The default value will be used if generic mapping is omitted. Use of the generic in component instantiation is shown below.

```
signal a4, b4, sum4: unsigned(3 downto 0);
signal a8, b8, sum8: unsigned(7 downto 0);
signal a16, b16, sum16: unsigned(15 downto 0);
signal c4, c8, c16: std_logic;
. . .
-- instantiate 8-bit adder
adder_8_unit: work.gen_add_w_carry(arch)
   generic map(N=>8)
   port map(a=>a8, b=>b8, cout=>c8, sum=>sum8));
-- instantiate 16-bit adder
adder_16_unit: work.gen_add_w_carry(arch)
   generic map(N=>16)
   port map(a=>a16, b=>b16, cout=>c16, sum=>sum16));
-- instantiate 4-bit adder
-- (generic mapping omitted, default value 4 used)
adder_4_unit: work.gen_add_w_carry(arch)
   port map(a=>a4, b=>b4, cout=>c4, sum=>sum4));
```

A generic provides a mechanism to create *scalable code*, in which the "width" of a circuit can be adjusted to meet a specific need. This makes code more portable and encourages design reuse.