CHAPTER 4

CONCURRENT SIGNAL ASSIGNMENT STATEMENTS OF VHDL

Concurrent signal assignment statements are simple, yet powerful VHDL statements. Since there is a clear mapping between the language constructs of an assignment statement and hardware components, we can easily visualize the conceptual diagram of the VHDL description. This helps us to develop a more efficient design. According to the VHDL definition, concurrent signal assignment statement has two basic forms: the *conditional signal assignment statement* and the *selected signal assignment statement*. For discussion purposes, we add an additional one, the *simple signal assignment statement*, which is a conditional assignment statement without any condition expression.

4.1 COMBINATIONAL VERSUS SEQUENTIAL CIRCUITS

A digital circuit can be broadly classified as combinational or sequential. A *combinational circuit* has no internal memory or state and its output is *a function of inputs only*. Thus, the same input values will always produce an identical output value. In a real circuit, the output may experience a short transient period after an input signal changes. However, the identical output value will be obtained when the signal is stabilized. In term of implementation, a combinational circuit is a circuit without memory elements (latches or flip-flops) or a closed feedback loop. A *sequential circuit*, on the other hand, has an internal state, and its output is *a function of inputs as well as the internal state*.

Although concurrent signal assignment statements can be used to describe sequential circuits, this is not the preferred method. We limit the discussion to combinational circuits in this chapter. We use the VHDL *process* to specify sequential circuits and study them in Chapter 8.

4.2 SIMPLE SIGNAL ASSIGNMENT STATEMENT

4.2.1 Syntax and examples

A simple signal assignment statement is a conditional signal assignment statement without the condition expression and thus is a special case of a conditional signal assignment statement. In VHDL definition, the simplified syntax of the simple signal assignment statement can be written as

```
signal_name <= projected_waveform;
```

The `projected_waveform` clause consists of two kinds of specifications: the expression of a new value for the signal and the time when the new value takes place. For example, consider the statement

```
y <= a + b + 1 after 10 ns;
```

which indicates that whenever the `a` or `b` signal changes, the expression `a+b+1` will be evaluated, and its result will be assigned to the y signal after 10 ns.

The time aspect of `projected_waveform` normally corresponds to the internal propagation delay to complete the computation of the expression. However, since the propagation delay depends on the components, device technology, routing, fabrication process and operation environment, it is impossible to synthesize a circuit with an exact amount of delay. Therefore, for synthesis, explicit timing information is not specified in VHDL code. The default δ-delay is used in the projected waveform. The syntax becomes

```
signal_name <= value_expression;
```

The `value_expression` clause can be a constant value, logical operation, arithmetic operation and so on. Following are a few examples:

```
status <= '1';
even <= (p1 and p2) or (p3 and p4);
arith_out <= a + b + c - 1;
```

Note that the timing aspect is not dropped. It is just specified implicitly as a δ-delay. The previous statements implicitly imply

```
status <= '1' after δ;
even <= (p1 and p2) or (p3 and p4) after δ;
arith_out <= <= a + b + c - 1 after δ;
```

4.2.2 Conceptual implementation

Deriving the conceptual hardware block diagram for a simple signal assignment statement is straightforward. The entire statement can be thought of as a circuit block. The output of the circuit is the signal in the left-hand side of the statement, and the inputs are all the signals that appear in the right-hand-side value expression. We then map each operator of the value expression into a smaller circuit block and connect their inputs and outputs accordingly. The conceptual diagrams of three previous statements are shown in Figure 4.1.

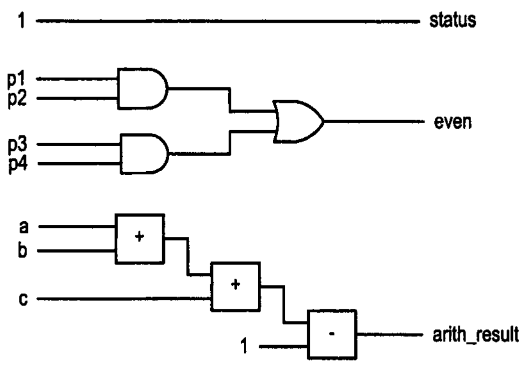


Figure 4.1 Conceptual diagrams of three simple signal assignment statements.

Note that these diagrams are only conceptual sketches. They will be transformed and simplified during synthesis. The circuit sizes of different VHDL operators vary significantly, and some of them, like the division operator, cannot be synthesized automatically. We examine this issue in detail in Chapter 6.

4.2.3 Signal assignment statement with a closed feedback loop

According to VHDL definition, it is syntactically correct for a signal to appear on both sides of a concurrent signal assignment statement. When an output signal is used as an input in the value expression, a closed feedback loop is formed. This may lead to the creation of an internal state or even oscillation. Consider the following VHDL statement:

```
q <= (q and (not en)) or (d and en);
```

In this example, the q signal is the output but also appears in the right-hand-side expression. The q output takes the value of the d signal if the en signal is '1' and it keeps its previous value if the en signal is '0'. Note that the output (i.e., q) now depends on input (i.e., en and d) as well as internal state (the previous value of q), and thus the circuit is no longer a combinational circuit. If we modify the previous statement by inverting q:

```
q <= ((not q) and (not en)) or (d and en);
```

the q output oscillates between '0' and '1' when the en signal is '0'.

When a signal assignment statement contains a closed feedback loop, it becomes sensitive to internal propagation delay and may exhibit race or oscillation. This kind of circuit confuses synthesis software and complicates verification and testing processes. It is a really bad coding practice and should be avoided completely in VHDL synthesis. The shortfall of delay-sensitive design and the disciplined derivation of sequential circuits are discussed in detail in Chapters 8 and 9.

Table 4.1 Function table of a 4-to-1 multiplexer

Input s	**Output** x
0 0	a
0 1	b
1 0	c
1 1	d

4.3 CONDITIONAL SIGNAL ASSIGNMENT STATEMENT

4.3.1 Syntax and examples

The simplified syntax of conditional signal assignment statement is shown below. As in Section 4.2.2, we assume that a timing specification is embedded implicitly in δ-delay and use `value_expression` to substitute the `projected_waveform` clause:

```
signal_name <= value_expr_1 when boolean_expr_1 else
               value_expr_2 when boolean_expr_2 else
               value_expr_3 when boolean_expr_3 else
                  . . .
               value_expr_n;
```

The `boolean_expr_i` (`i=` 1, 2, 3, . . ., n) term is a Boolean expression that returns `true` or `false`. These Boolean expressions are evaluated successively in turn until one is found to be `true`, and the corresponding value expression is assigned to the output signal. In other words, the first Boolean expression, `boolean_expr_1`, is checked first. If it is `true`, the first value expression, `value_expr_1`, will be assigned to the output signal. If it is `false`, the second Boolean expression, `boolean_expr_2`, will be checked next. This process continues until all Boolean expressions are checked. The last value expression, `value_expr_n`, will be assigned to the signal if none of the Boolean expressions is `true`.

In the remaining subsection, we use several simple examples to illustrate the use of conditional signal assignment statements. The circuits include a multiplexer, a decoder, a priority encoder and a simple arithmetic logic unit (ALU).

Multiplexer A multiplexer is essentially a virtual switch that routes a selected input signal to the output. The function table of an 8-bit 4-to-1 multiplexer is show in Table 4.1. In this circuit, the a, b, c and d signals can be considered as input data, and the s signal is a 2-bit selection signal that specifies which input data will be routed to the output. The VHDL code for this circuit is shown in Listing 4.1.

Listing 4.1 4-to-1 multiplexer based on a conditional signal assignment statement

```
library ieee;
use ieee.std_logic_1164.all;
entity mux4 is
   port(
      a,b,c,d: in std_logic_vector(7 downto 0);
      s: in std_logic_vector(1 downto 0);
      x: out std_logic_vector(7 downto 0)
   );
```

```
end mux4;

architecture cond_arch of mux4 is
begin
   x <= a when (s="00") else
        b when (s="01") else
        c when (s="10") else
        d;
end cond_arch;
```

The first two lines are used to invoke the IEEE std_logic_1164 package so that the std_logic data type can be used. The next part is the entity declaration, which specifies the input and output ports of this circuit. The input ports include a, b, c and d, which are four 8-bit input data, and s, which is the 2-bit control signal. The output port is the 8-bit x signal. The architecture part uses one conditional signal assignment statement. The Boolean condition s="00" is evaluated first. If it is true, the first value expression, a, is assigned to x. If it is false, the next Boolean condition, s="01", will be evaluated. If it is true, b is assigned to x or the next Boolean expression, s="10", will be evaluated. If all three Boolean expressions are false, the last value expression, d, is assigned to x.

There is an issue about the use of the std_logic data type. At first glance, it seems that s is implied to be "11" when the first three Boolean expressions are false, and thus d is assigned to x. However, there are nine possible values in std_logic data type and, for the 2-bit s signal, there are 81 (i.e., 9*9) possible combinations, including the expected "00", "01", "10" and "11" as well as the metavalue combinations, such as "0Z", "UX", "0-" and so on. Therefore, d is assigned to x for the "11" condition, as well as other 77 metavalue combinations. However, these 77 combinations can exist only in simulation. In a real circuit, comparison of metavalues, as in s="0Z", cannot be implemented, and sometimes is meaningless, as in s="UX". In general, except for the limited use of 'Z', the metavalues of the std_logic data type will be ignored by synthesis software, and thus the final circuit will be synthesized as we originally expected. Some synthesis software also accepts VHDL code using 'X' for the unused metavalue combinations:

```
x <= a when (s="00") else
     b when (s="01") else
     c when (s="10") else
     d when (s="11") else
     'X';
```

The code leads to the same physical implementation.

Binary decoder A binary decoder is an n-to-2^n decoder, which has an n-bit input and a 2^n-bit output. Each bit of the output represents an input combination. Based on the value of the input, the circuit activates the corresponding output bit. The function table of a simple 2-to-2^2 decoder is shown in Table 4.2. The VHDL code for this circuit is shown in Listing 4.2.

Listing 4.2 2-to-2^2 binary decoder based on a conditional signal assignment statement

```
library ieee;
use ieee.std_logic_1164.all;
entity decoder4 is
   port(
      s: in  std_logic_vector(1 downto 0);
```

Table 4.2 Function table of a 2-to-2^2 binary decoder

Input s	Output x
0 0	0001
0 1	0010
1 0	0100
1 1	1000

Table 4.3 Function table of a 4-to-2 priority encoder

Input	Output	
r	code	active
1 – – –	11	1
0 1 – –	10	1
0 0 1 –	01	1
0 0 0 1	00	1
0 0 0 0	00	0

```
      x: out std_logic_vector(3 downto 0)
   );
end decoder4;

architecture cond_arch of decoder4 is
begin
   x <= "0001" when (s="00") else
        "0010" when (s="01") else
        "0100" when (s="10") else
        "1000";
end cond_arch;
```

Again, the first two lines are used to invoke the IEEE std_logic_1164 package. The entity declaration shows the circuit with a 2-bit input, a, and a 4-bit output, x. The architecture body uses one conditional signal assignment statement, which evaluates the Boolean conditions s="00", s="01" and s="10" one after another. The value expressions are constants that reflect the desired output patterns.

Priority encoder A priority encoder checks the input requests and generates the code of the request with highest priority. The function table of a 4-to-2 priority encoder is shown in Table 4.3. There are four input requests, r(3), r(2), r(1) and r(0). The outputs include a 2-bit signal, code, which is the binary code of the highest-priority request, and a 1-bit signal, active, which indicates whether there is an active request. The r(3) request has the highest priority. When it is asserted, the other three requests are ignored and the code signal becomes "11". If r(3) is not asserted, the second highest request, r(2), is examined. If it is asserted, the code signal becomes "10". The process repeats until all the requests are checked. The code signal returns "00" when only r(0) is asserted or no request is asserted. The active signal can be used to distinguish the two conditions. The VHDL code for this circuit is shown in Listing 4.3. The requests are grouped together and

Table 4.4 Function table of a simple ALU

Input ctrl	**Output** result
0 - -	src0 + 1
1 0 0	src0 + src1
1 0 1	src0 - src1
1 1 0	src0 **and** src1
1 1 1	src0 **or** src1

represented by a 4-bit signal, `r`. Individual bits of the `r` signal are checked in descending order, starting with `r(3)`. Since operation of the priority encoder is similar to the definition of the conditional signal assignment statement, it is a good way to code this type of circuit (note the simple Boolean expressions in the code). A separate simple signal assignment statement is used to describe the `active` output.

Listing 4.3 4-to-2 priority encoder based on a conditional signal assignment statement

```
library ieee;
use ieee.std_logic_1164.all;
entity prio_encoder42 is
   port(
      r: in std_logic_vector(3 downto 0);
      code: out std_logic_vector(1 downto 0);
      active: out std_logic
   );
end prio_encoder42;

architecture cond_arch of prio_encoder42 is
begin
   code <= "11" when (r(3)='1') else
           "10" when (r(2)='1') else
           "01" when (r(1)='1') else
           "00";
   active <= r(3) or r(2) or r(1) or r(0);
end cond_arch;
```

Simple ALU An ALU performs a set of arithmetic and logical operations. The function table of a simple ALU is shown in Table 4.4. The inputs include two 8-bit data sources, `scr0` and `src1`, and a control signal, `ctrl`, which specifies the function to be performed. The output is the 8-bit `result` signal, which is the computed result. There are five functions, including three arithmetic operations, which are incrementing, addition and subtraction, and two logical operations, which are bitwise and and or operations. Furthermore, we assume that the input and output are interpreted as signed integers when an arithmetic function is selected.

For this circuit, the input data are interpreted as a collection of bits for the logical operation and as a signed number for the arithmetic operation. To achieve better portability, we normally use the `std_logic_vector` data type in the port declaration and then convert it to the desired data type in architecture body. The VHDL code is shown in Listing 4.4. The

IEEE numeric_std package and its signed data type are used to facilitate the arithmetic operation. When an addition, subtraction or incrementing operation is specified, we first convert the input to the signed data type, perform the operation and then convert the result back to the std_logic_vector data type. To make the code clear, we introduce three separate simple signal assignment statements and the sum, diff, and inc signals for the intermediate results of arithmetic operations.

Listing 4.4 Simple ALU based on a conditional signal assignment statement

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity simple_alu is
   port(
      ctrl: in  std_logic_vector(2 downto 0);
      src0, src1: in std_logic_vector(7 downto 0);
      result: out std_logic_vector(7 downto 0)
   );
end simple_alu;

architecture cond_arch of simple_alu is
   signal sum, diff, inc: std_logic_vector(7 downto 0);
begin
   inc <= std_logic_vector(signed(src0)+1);
   sum <= std_logic_vector(signed(src0)+signed(src1));
   diff <= std_logic_vector(signed(src0)-signed(src1));
   result <= inc  when ctrl(2)='0' else
             sum  when ctrl(1 downto 0)="00" else
             diff when ctrl(1 downto 0)="01" else
             src0 and src1 when ctrl(1 downto 0)="10" else
             src0 or src1;
end cond_arch;
```

4.3.2 Conceptual implementation

Recall that the syntax of the simplified conditional signal assignment statement is

```
signal_name <= value_expr_1 when boolean_expr_1 else
               value_expr_2 when boolean_expr_2 else
               value_expr_3 when boolean_expr_3 else
                . . .
               value_expr_n;
```

Its semantics specifies that the Boolean expressions are evaluated in descending order until a condition is true, and then the corresponding value expression is assigned to the output signal. The key to implementing this construct is to achieve the desired descending order of evaluations. In a traditional programming language, descending order is implicitly observed because of the sequential execution of a single, shared CPU. In synthesis, we must use hardware to achieve this task.

The structure of conditional signal assignment statement implies a priority routing network since the Boolean expressions are evaluated in an orderly manner and the one evaluated earlier assumes a higher priority. Once the evaluation of a Boolean expression is true, the

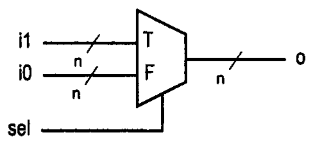


Figure 4.2 Conceptual diagram of an abstract multiplexer.

result of the corresponding value expression is routed to output. Unlike the *temporal* execution of the traditional programming language, the priority routing network is done on a *spatial* basis. Furthermore, since we cannot create hardware dynamically, dedicated hardware is needed for each Boolean expression and each value expression.

In summary, constructing the conditional signal assignment statement requires three groups of hardware:

- Value expression circuits
- Boolean expression circuits
- Priority routing network

Value expression circuits realize the value expressions, `value_expr_1`, · · ·, `value_expr_n`, and one of the results is routed to the output. *Boolean expression circuits* realize the Boolean expressions, `boolean_expr_1`, · · ·, `boolean_expr_n`, and their values are used to control the priority routing network. The *priority routing network* is the structure that routes and controls the desired value to the output signal.

A priority network can be implemented by a sequence of 2-to-1 multiplexers. To better illustrate the conceptual implementation, we utilize an "abstract multiplexer." Recall that a multiplexer is like a switch and uses a selection signal to select an input port and connect it to the output port. Any signal appearing in that input port will be routed to the output port. In an abstract multiplexer, the selection and input port designation are specified around the data type of the selection signal. Each input port is designated to a value of the data type of the selection signal, and one input port is selected according to the current value of the selection signal. For example, if the selection signal has a data type of `boolean`, there will be two input ports, designated as `T` (for `true`) and `F` (for `false`) respectively. If the selection signal has a value of `true`, the data from the `T` port will be routed to output. On the other hand, if the selection signal has a value of `false`, the data from the `F` port will be selected. The block diagram of this multiplexer is shown in Figure 4.2. The number of bits of the inputs and output may vary, and the symbol, `n`, is used to designate the width of the buses. During synthesis, the symbolic values can easily be mapped into binary representations of a physical multiplexer.

With the 2-to-1 abstract multiplexer, we can start to construct a priority network. Let us first consider a simple conditional signal assignment statement that has only one when clause:

```
sig <= value_expr_1 when boolean_expr_1 else
       value_expr_2;
```

The conceptual realization of this statement is shown in Figure 4.3. The three "clouds" represent the implementations of `value_expr_1`, `value_expr_2` and `boolean_expr_1` respectively. The result of `boolean_expr_1` is connected to the selection signal of the multiplexer. If it is `true`, the result from `value_expr_1` will be routed to the output port of the multiplexer. Otherwise, the result from `value_expr_2` will be routed to the output port.

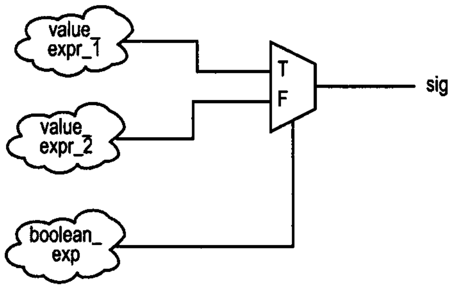


Figure 4.3 Conceptual diagram of a simple conditional signal assignment statement.

When there are more when clauses, we can perform the previous process repetitively and build the routing network in stages. Consider a statement with three when clauses:

```
sig <= value_expr_1 when boolean_expr_1 else
       value_expr_2 when boolean_expr_2 else
       value_expr_3 when boolean_expr_3 else
       value_expr_4;
```

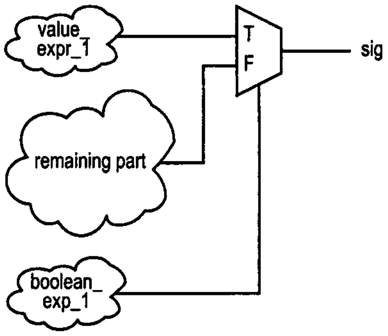
The construction sequence is shown in Figure 4.4. First we construct the first when clause, which corresponds to the highest-priority condition. If the result of boolean_expr_1 is true, the result of the corresponding value expression, value_expr_1, is routed to output, as shown in Figure 4.4(a). On the other hand, if the result of boolean_expr_1 is false, the result from the remaining part of the statement, which is shown as a single cloud, will be used. This cloud can be constructed using a multiplexer similar to the first when clause, with its output connected to the F port of the rightmost multiplexer, as shown in Figure 4.4(b). After repeating this process one more time, we construct the third when clause and complete the conceptual implementation, as shown in Figure 4.4(c).

The construction process can be applied repeatedly to any number of when clauses. Since each clause will introduce one extra stage of multiplexer network, the depth of the network grows as the number of clauses increases. Although the conceptual construction is straightforward, it is difficult for synthesis software to transform an extremely deep multiplexer network to an efficient implementation. Thus, we should be aware of the impact on the number of when clauses. Discussion in Chapter 6 provides more insight on this issue.

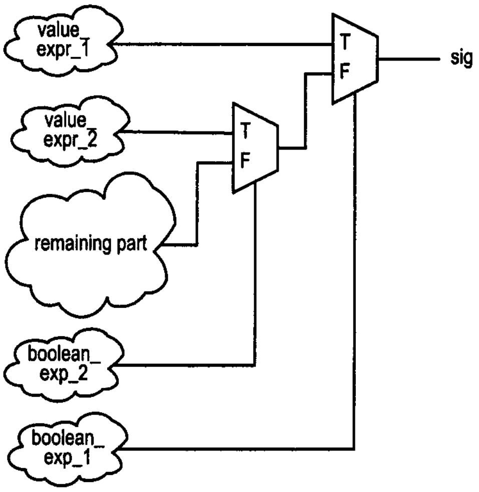
4.3.3 Detailed implementation examples

Obtaining the conceptual diagram is only the first step in synthesis. We must derive the more detailed implementation for the multiplexers and "clouds" and eventually construct everything by using cells of the given technology library. Many of these tasks can be done in synthesis software, which is discussed in Chapter 6. In this section, we manually derive some simple circuits from VHDL segments to illustrate the basic synthesis process.

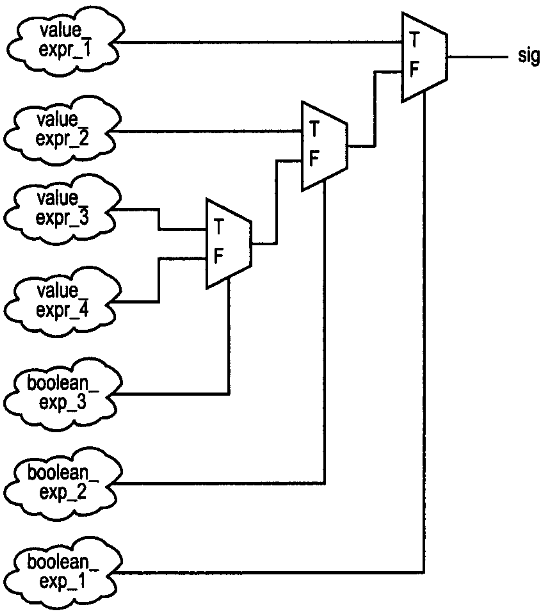
Implementation of a 2-to-1 multiplexer An abstract 2-to-1 multiplexer has two symbolic ports, T and F. We can map it directly to a regular 2-to-1 multiplexer. The schematic of a 1-bit 2-to-1 multiplexer is shown in Figure 4.5(a). The two abstract ports, T and F, are mapped to the *i*1 and *i*0 ports respectively. In this circuit, the and cells can be interpreted as "passing gates," controlled by separate enable signals. When the enable signal is '1',



(a) First when clause

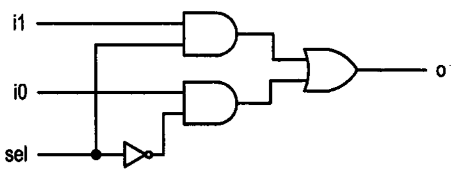


(b) Second when clause

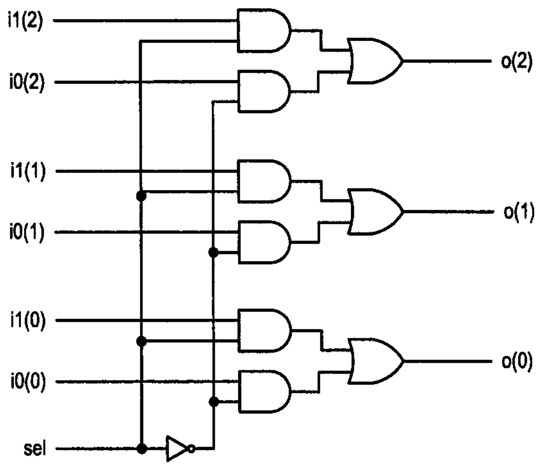


(c) Third when clause

Figure 4.4 Construction of a multi-condition conditional signal assignment statement.



(a) 1-bit 2-to-1 multiplexer



(b) 3-bit 2-to-1 multiplexer

Figure 4.5 Gate-level implementation of a multiplexer.

the gate is open and the input signal is passed to output. When it is '0', the gate is closed and the output is set to '0'. The two enable signals are *sel* and *sel'* respectively, and thus one of the inputs will be passed to output. In terms of a logic expression, the output can be expressed as

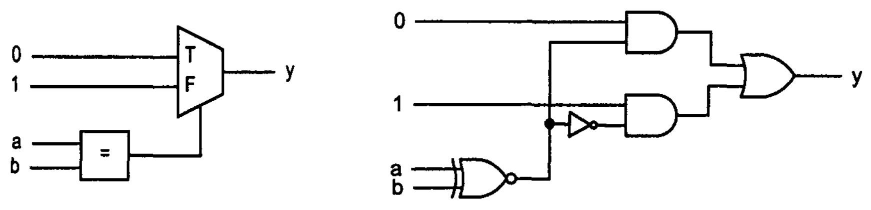
$$o = sel' \cdot i0 + sel \cdot i1$$

For an n-bit 2-to-1 multiplexer, the control signals remain the same, but the gating structure will be duplicated n times. The schematic of a 3-bit 2-to-1 multiplexer is shown in Figure 4.5(b).

Example 1 Consider the following VHDL segment:

```
. . .
signal a,b,y: std_logic;
. . .
y <= '0' when a=b else
     '1';
. . .
```

This is a simple conditional signal assignment statement that contains one when clause. The conceptual diagram is shown in Figure 4.6(a). Let us consider the implementation of `a=b`, which is a 1-bit comparison circuit. According to VHDL definition, the input data type is `std_logic`, which has nine values, and the output data type is `boolean`, whose value can be `true` or `false`. During synthesis, we only consider the `'0'` and `'1'` of the `std_logic`



(a) Conceptual diagram (b) Gate-level diagram

Figure 4.6 Synthesis of example 1.

Table 4.5 Truth table of a 1-bit comparator.

input	**output**
a b	a=b
0 0	1
0 1	0
1 0	0
1 1	1

data type since the other seven values are meaningless for a physical circuit. We also map the `true` and `false` to logic 1 and logic 0 of the physical circuit. Now the operation a=b can be represented in a traditional truth table, as shown in Table 4.5. The function can be expressed as $a' \cdot b' + a \cdot b$, or simply $(a \oplus b)'$, which is an xnor gate. We can now refine the conceptual diagram into the gate-level implementation, and the new diagram is shown in Figure 4.6(b). We can derive the logic expression of this circuit. Based on the expression of the multiplexer, the output can be expressed as

$$y = sel' \cdot i0 + sel \cdot i1$$

The sel, $i0$ and $i1$ are connected to $(a \oplus b)'$, '1' and '0' respectively, and thus the expression becomes

$$y = sel' \cdot i0 + sel \cdot i1 = (a \oplus b)'' \cdot 1 + (a \oplus b)' \cdot 0$$

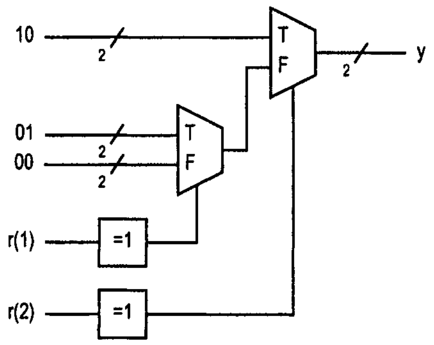
which can be simplified to

$$y = a \oplus b$$

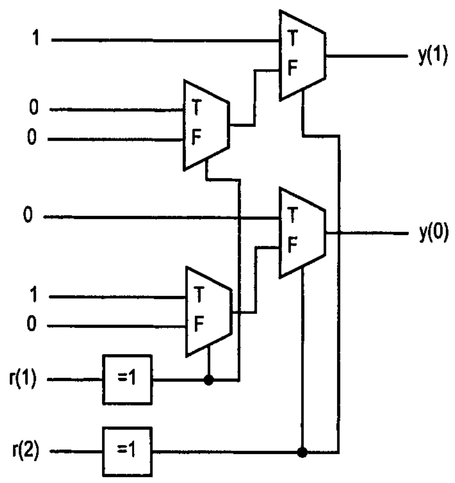
Thus, the final simplified circuit is a single xor gate.

Example 2 Consider the following VHDL segment:

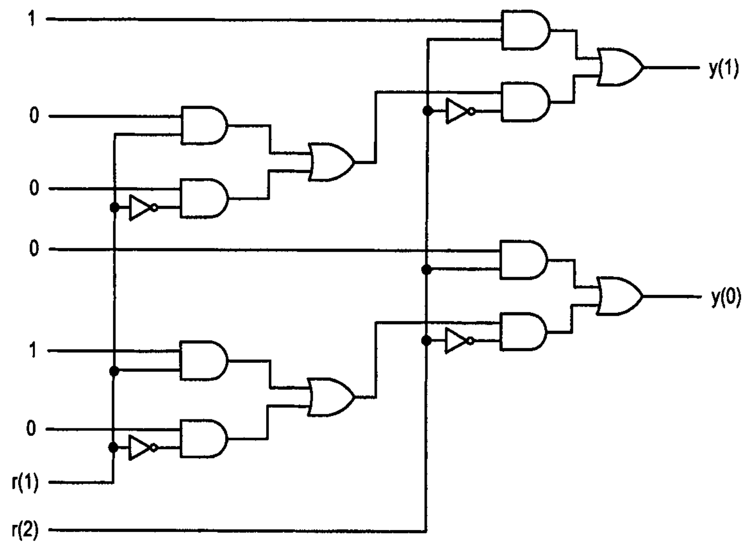
```
. . .
signal r: std_logic_vector(2 downto 1);
signal y: std_logic_vector(1 downto 0);
. . .
y <= "10" when r(2)='1' else
     "01" when r(1)='1' else
     "00";
. . .
```



(a) "Bus" conceptual diagram



(b) Bit-by-bit conceptual diagram



(c) Gate-level diagram

Figure 4.7 Synthesis of example 2.

The conceptual diagram of this segment is shown in Figure 4.7(a). The next step is to derive gate-level implementation. Since the output has two bits, we have to split the conceptual diagram into two single-bit diagrams, as in Figure 4.7(b). Note that the implementation of the Boolean expressions, `r(2)='1'` and `r(1)='1'`, consists simply of the `r(2)` and `r(1)` signals themselves, and no additional logic is needed. After we substitute the multiplexer with its gate-level implementation, the resulting circuits are shown in Figure 4.7(c). We can derive the logic expressions for `y(0)` and `y(1)` using a procedure similar to that in example 1. After simplification, these logic expressions become

$$y(1) = r(2)$$

$$y(0) = r(2)' \cdot r(1)$$

Example 3 Consider the following VHDL segment:

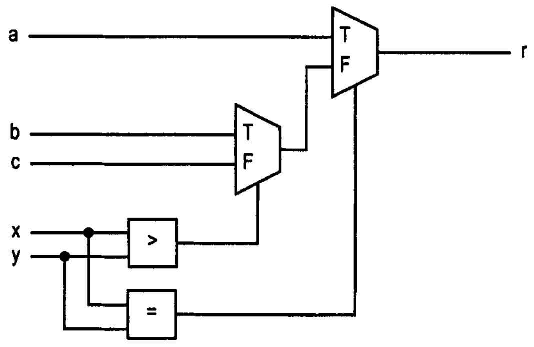
```
. . .
signal a,b,c,x,y,r: std_logic;
. . .
r <= a when x=y else
     b when x>y else
     c;
. . .
```

The conceptual diagram of this segment is shown in Figure 4.8(a). By using the procedure to realize the a=b expression of example 1, we can derive the implementation of the x>y expression, which is $x \cdot y'$. The corresponding gate level circuit is shown in Figure 4.8(b). We can also derive the logic expression for the output and perform simplification to reduce the circuit size. The logic expression for this circuit is more involved and manually simplifying this circuit becomes a tedious task. This task is better left for software, which is good for a mechanical and repetitive procedure.

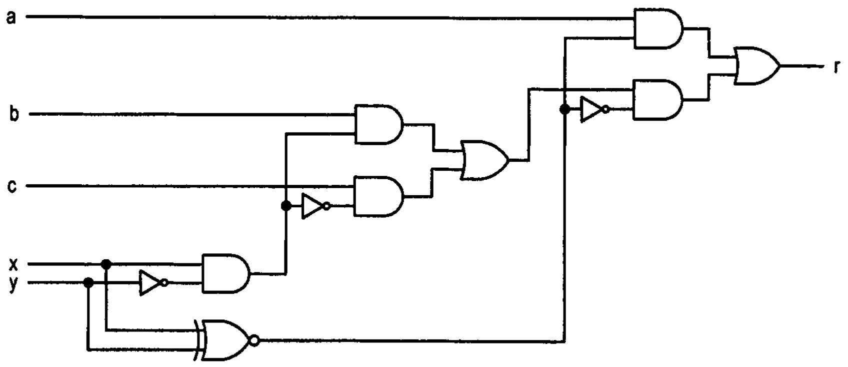
Example 4 Consider the following VHDL segment:

```
. . .
signal a,b,r: unsigned(7 downto 0);
signal x,y: unsigned(3 downto 0);
. . .
r <= a+b when x+y>1 else
     a-b-1 when x>y and y!=0 else
     a+1;
. . .
```

The initial block diagram of this segment is shown in Figure 4.9(a). While the initial block diagram is similar to the previous examples, the value expressions and Boolean expressions are more involved. More complex components, such as an adder and comparator, are needed for implementation. After we implement the clouds, the block diagram is shown in Figure 4.9(b). We can continue to refine the circuit by replacing these components with their gate-level implementations and eventually derive the logic expressions. With these components, performing gate-level simplification becomes much more difficult and good coding practice at the RT level can improve the circuit efficiency significantly. These issues are discussed in Chapter 7.

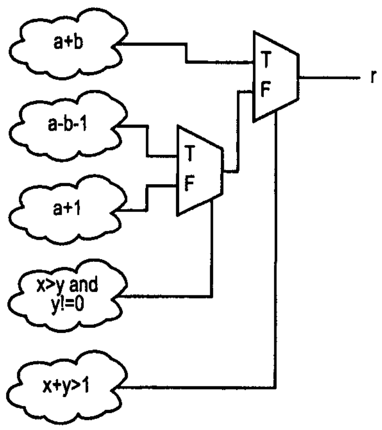


(a) Conceptual diagram

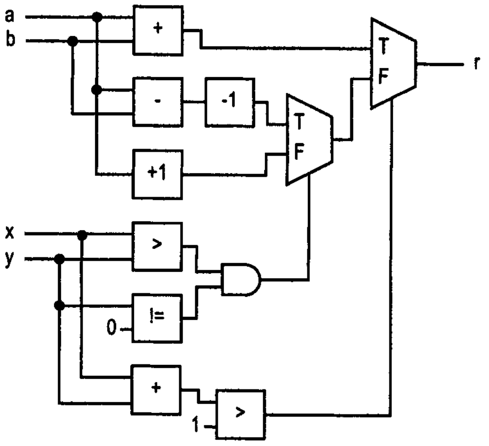


(b) Gate-level diagram

Figure 4.8 Synthesis of example 3.



(a) Initial diagram



(b) Detailed conceptual diagram

Figure 4.9 Refinement of example 4.

4.4 SELECTED SIGNAL ASSIGNMENT STATEMENT

4.4.1 Syntax and examples

The simplified syntax of the selected signal assignment statement is shown below. As in conditional signal assignment statement, we assume that the timing specification is embedded in δ-delay and substitute `value_expression` for the `projected_waveform` clause.

```
with select_expression select
   signal_name <= value_expr_1 when choice_1,
                  value_expr_2 when choice_2,
                  value_expr_3 when choice_3,
                     . . .
                  value_expr_n when choice_n;
```

The selected signal assignment statement assigns an expression to a signal according to the value of `select_expression`. It is somewhat like a case statement in a traditional programming language. The `select_expression` term is used as the key for selection and it must result in a value of a discrete type or one-dimensional array. In other words, the evaluated result of `select_expression` can have only a finite number of possibilities. For example, a signal of the `bit_vector(1` **downto** `0)` data type can be used as `select_expression` since it contains only four possible values: `"00"`, `"01"`, `"10"` or `"11"`. A choice (i.e., `choice_i`) must be a valid value or a set of valid values of `select_expression`. The values of choices have to be *mutually exclusive* (i.e., no value can be used more than once) and *all inclusive* (i.e., all values have to be used). In other words, all possible values of `select_expression` must be covered by one and only one choice. The reserved word, **others**, can be used in the last choice (i.e., `choice_n`) to represent all the previously unused values.

We use the same multiplexer, binary decoder, priority encoder and ALU circuit of Section 4.3.1 to illustrate use of the selected signal assignment statement. Since this statement is a natural match to implement a truth table, an additional example is included for this purpose.

Multiplexer Let us consider the 8-bit 4-to-1 multiplexer of Section 4.3.1. The VHDL code for this circuit is shown in Listing 4.5. The entity declaration is identical and thus is omitted.

Listing 4.5 4-to-1 multiplexer based on a selected signal assignment statement

```
architecture sel_arch of mux4 is
begin
   with s select
     x <= a when "00",
          b when "01",
          c when "10",
          d when others;
end sel_arch;
```

We need to be cautious about the metavalues of the `std_logic` and `std_logic_vector` data types. There is an issue about the use of these data types for `select_expression`. Recall that there are nine possible values in `std_logic` data type and there are 81 (i.e., 9*9) possible combinations for the 2-bit `s` signal, including the expected `"00"`, `"01"`, `"10"` and `"11"` as well as 77 other metavalue combinations, such as `"ZZ"`, `"UX"` and `"0-"`, which are not meaningful in synthesis and will be ignored accordingly. In the code, the **when others** clause covers the "11" choice as well as the metavalue combinations. We cannot simply list the last choice as "11":

```
with s select
  x <= a when "00",
       b when "01",
       c when "10",
       d when "11";
```

This causes a syntax error since only 4 of 81 values are covered, and thus the choices are not all-inclusive. Some synthesis software may accept the following form:

```
with s select
  x <= a when "00",
       b when "01",
       c when "10",
       d when "11",
      'X'when others; -- may also use '-'
```

The last line will be ignored during synthesis and the same physical circuit will be derived.

Binary decoder The VHDL code for the 2-to-2^2 binary decoder of Section 4.3.1 is shown in Listing 4.6. Again, it is necessary to use **others** as the last choice to cover all metavalue combinations.

Listing 4.6 2-to-2^2 binary decoder based on a selected signal assignment statement

```
architecture sel_arch of decoder4 is
begin
   with s select
     x <= "0001" when "00",
          "0010" when "01",
          "0100" when "10",
          "1000" when others;
end sel_arch;
```

Priority encoder The VHDL code for the 4-to-2 priority encoder is shown in Listing 4.7. Recall that "11" will be assigned to code if r(3) is '1'. This consists of eight possible input combinations of the r signal, which are "1000", "1001", "1010", . . . , "1111". All of them are listed in the first choice. Note that the symbol | is used for specifying multiple values.

Listing 4.7 4-to-2 priority encoder based on a selected signal assignment statement

```
architecture sel_arch of prio_encoder42 is
begin
   with r select
      code <= "11" when "1000"|"1001"|"1010"|"1011"|
                        "1100"|"1101"|"1110"|"1111",
              "10" when "0100"|"0101"|"0110"|"0111",
              "01" when "0010"|"0011",
              "00" when others;
   active <= r(3) or r(2) or r(1) or r(0);
end sel_arch;
```

Intuitively, we may wish to use the '-' (don't-care) value of the `std_logic` data type to make the code compact:

```
    with r select
       code <= "11" when "1---",
               "10" when "01--",
               "01" when "001-",
               "00" when others;
```

While this is syntactically correct, the code does not describe the intended circuit. In VHDL, the '-' value is treated just as an ordinary value of `std_logic`. Since the '-' value will never occur in the physical circuit, the "1---", "01--" and "001-" choices will never be met and the code is the same as

```
code <= "00";
```

This, of course, is not the intended priority encoding circuit. We discuss this issue in more detail in Chapter 6.

A simple ALU The VHDL code of the simple ALU specified in Table 4.4 is shown in Listing 4.8. Note that all four possible combinations of the `ctrl` signal, "000", "001", "010" and "011", are listed in the first choice.

Listing 4.8 Simple ALU based on a selected signal assignment statement

```
architecture sel_arch of simple_alu is
   signal sum, diff, inc: std_logic_vector(7 downto 0);
begin
   inc <= std_logic_vector(signed(src0)+1);
   sum <= std_logic_vector(signed(src0)+signed(src1));
   diff <= std_logic_vector(signed(src0)-signed(src1));
   with ctrl select
      result <= inc           when "000"|"001"|"010"|"011",
                sum           when "100",
                diff          when "101",
                src0 and src1 when "110",
                src0 or src1  when others; -- "111"
end sel_arch;
```

Truth Table Implementation A truth table can be used to specify any combinational function. It is a simple and useful way to describe a small, random combinational circuit. Because the choices list all the possible combinations, the selected signal assignment statement is a natural match for the truth table description. A simple two-input truth table is shown in Table 4.6.

The corresponding VHDL code is shown in Listing 4.9. The a and b signals are concatenated as tmp, which is then used as the select expression. Each row of the truth table now becomes a choice in the selected signal assignment statement and the truth table is implemented accordingly.

Listing 4.9 Truth table based on selected signal assignment statement

```
library ieee;
use ieee.std_logic_1164.all;
entity truth_table is
```

Table 4.6 Truth table of a two-input function

Input a b	Output y
0 0	0
0 1	1
1 0	1
1 1	1

```
   port(
      a,b: in   std_logic;
      y: out    std_logic
   );
end truth_table;

architecture a of truth_table is
   signal tmp: std_logic_vector(1 downto 0);
begin
   tmp <= a & b;
   with tmp select
      y <= '0' when "00",
           '1' when "01",
           '1' when "10",
           '1' when others; -- "11"
end a;
```

4.4.2 Conceptual implementation

Recall that the syntax of the selected signal assignment is

```
with select_expression select
   signal_name <= value_expr_1 when choice_1,
                  value_expr_2 when choice_2,
                  value_expr_3 when choice_3,
                     . . .
                  value_expr_n when choice_n;
```

Conceptually, the selected signal assignment statement can be thought as an abstract multiplexing circuit that utilizes a selection signal to route the result of the designated expression to output. In this multiplexing circuit, each possible value of select_expression has a designated input port in the multiplexer, and select_expression works as the selection signal of this multiplexer. Once its value is determined, the result of the designated value expression is passed to the output port of the multiplexer. In Section 4.3.2, we utilized an abstract 2-to-1 multiplexer with a selection signal of the boolean data type. The multiplexer can be generalized for other kinds of selection signals. For example, consider a selection signal with $k + 1$ different possible values, c0, c1, ..., ck. The abstract multiplexer has $k + 1$ ports, each corresponding to a value, as shown in Figure 4.10.

It is possible that the input and output have multiple bits and the symbol n is used to designate the width of the buses. The conceptual implementation of the selected signal

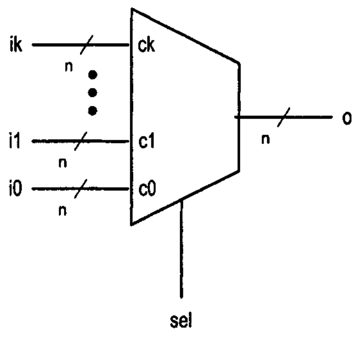


Figure 4.10 Abstract ($k + 1$)-to-1 multiplexer.

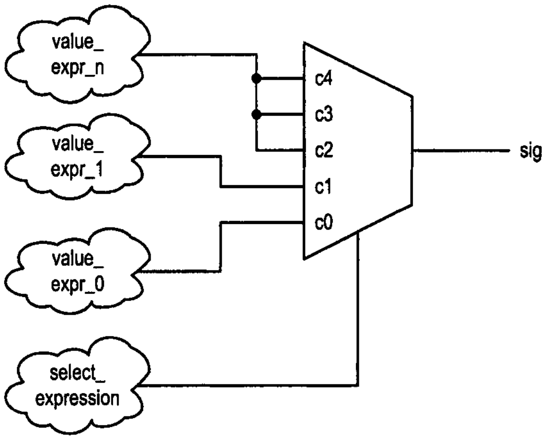


Figure 4.11 Conceptual diagram of a selected signal assignment statement.

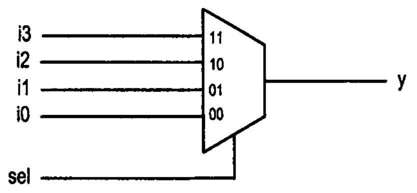
assignment statement involves a single abstract multiplexer and is straightforward. Consider the following statement:

```
with select_expression select
   sig <= value_expr_0 when c0,
          value_expr_1 when c1,
          value_expr_n when others;
```

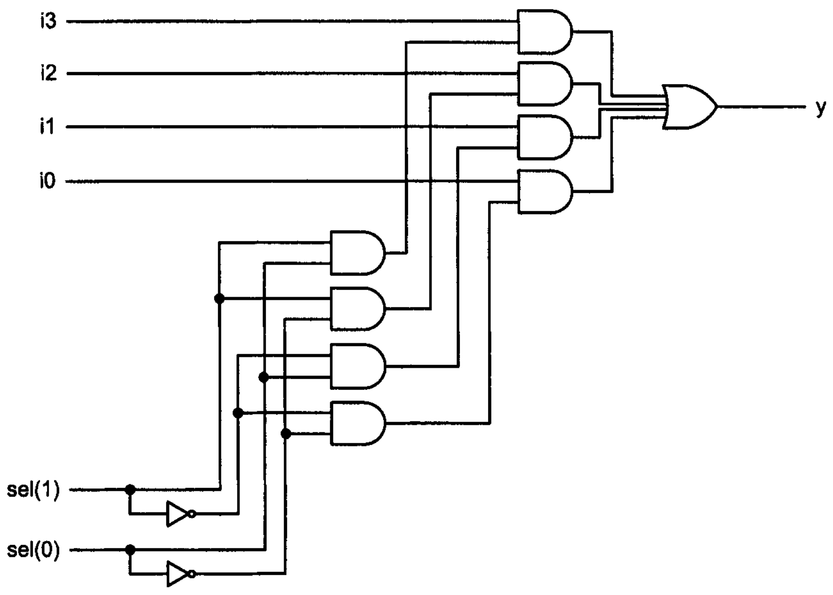
We assume that select_expression may result in one of five possible values: c0, c1, c2, c3 and c4. Note that the last choice, **when others**, of this statement implicitly covers c2, c3 and c4. The conceptual realization of this statement is shown in Figure 4.11.

The clouds represent the implementation of the three value expressions, value_expr_0, value_expr_1 and value_expr_n, and select_expression respectively. The evaluated results of the value expressions are fed into the designated input ports of the multiplexer. The result of select_expression is connected to the selection port of the multiplexer and its value determines which data will be routed to the output port.

All selected signal assignment statements have a similar conceptual diagram. The main difference is in the number of values that select_expression can assume, which in turn determines the size of the multiplexer. Despite the simple conceptual construction, certain



(a) Circuit symbol



(b) Gate-level diagram

Figure 4.12 Circuit symbol and gate-level diagram of a 4-to-1 multiplexer.

device technologies may have difficulty supporting an extremely wide multiplexing circuit. Thus, we should be aware of the number of values in a selection expression.

4.4.3 Detailed implementation examples

As in the implementation of a conditional signal assignment statement, we continue the refining process and realize the conceptual diagram using gate-level components. Following examples illustrate the derivation.

k-to-1 multiplexer An abstract multiplexer with k symbolic ports can easily be mapped to a physical k-to-1 multiplexer with a $\log_2 k$-bit selection signal. The symbol and gate-level diagram of a 1-bit 4-to-1 multiplexer are shown in Figure 4.12. We use the binary representations, "00", "01", "10" and "11", as the names of the ports. The upper and cells can be thought of as "passing gates," each controlled by an enable signal. The corresponding input will be passed to output when the enable signal is '1'. The bottom part is a 2-to-4 binary decoder that generates the enable signal, in which only one bit is activated. In term of a logic expression, the output can be expressed as

$$y = (sel(1)' \cdot sel(0)') \cdot i0 + (sel(1)' \cdot sel(0)) \cdot i1 + (sel(1) \cdot sel(0)') \cdot i2 + (sel(1) \cdot sel(0)) \cdot i3$$

For a multiple-bit 4-to-1 multiplexer, the enable signals remain the same, but the gating structure will be duplicated multiple times.

In VHDL code, the selection signal frequently has a data type of `std_logic_vector`, which includes many meaningless combinations. During synthesis, only `'0'` and `'1'` of nine values will be used, as we discussed in Section 4.3.1.

Example 1 Consider the following VHDL segments:

```
. . .
signal s: std_logic_vector(1 downto 0);
   . . .
   with s select
      x <= (a and b) when "11",
           (a or b)  when "01"|"10",
           '0'       when others;
. . .
```

This is a simple selected signal assignment statement. The selection expression has a data type of `std_logic_vector(1 downto 0)`. Again, although there are 81 possible values, only `"00"`, `"01"`, `"10"` and `"11"` are meaningful for synthesis. Thus, only a 4-to-1 multiplexer is needed. The conceptual diagram and the refined gate-level diagram are shown in Figure 4.13. The logic expression for this circuit is

$$x = (s(1)' \cdot s(0)') \cdot 0 + (s(1)' \cdot s(0)) \cdot (a+b) + (s(1) \cdot s(0)') \cdot (a+b) + (s(1) \cdot s(0)) \cdot (a \cdot b)$$

Example 2 Consider the truth table in Table 4.6 and the corresponding VHDL segment:

```
tmp <= a & b;
with tmp select
   y <= '0' when "00",
        '1' when "01",
        '1' when "10",
        '1' when others;
```

The conceptual diagram is shown in Figure 4.14. The logic expression for this circuit is

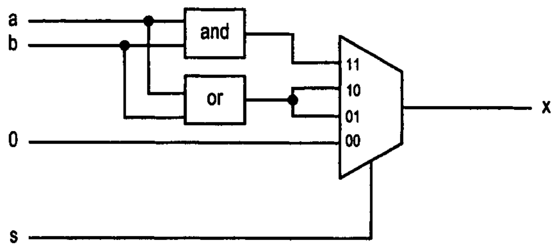
$$y = (a' \cdot b') \cdot 0 + (a' \cdot b) \cdot 1 + (a \cdot b') \cdot 1 + (a \cdot b) \cdot 1$$

The expression can be simplified to $a + b$, which is the or function specified in the truth table.

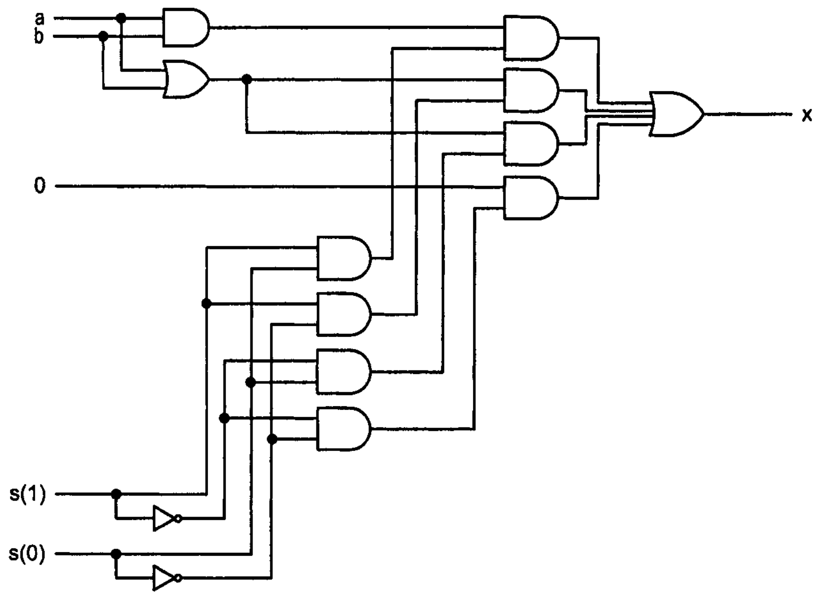
Example 3 Consider the following VHDL segment:

```
. . .
signal a,b,r: unsigned(7 downto 0);
signal s: std_logic_vector(1 downto 0);
   . . .
   with s select
      r <= a+1   when "11",
           a-b-1 when "10",
           a+b   when others;
. . .
```

This segment contains more sophisticated expressions. After we realized the value expression clouds, the block diagram is shown in Figure 4.15.



(a) Conceptual diagram



(b) Gate-level diagram

Figure 4.13 Synthesis of example 1.

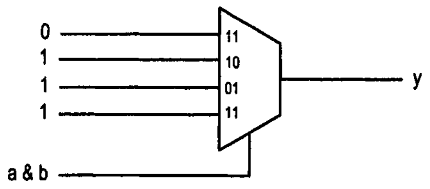


Figure 4.14 Conceptual diagram of truth table–based description.

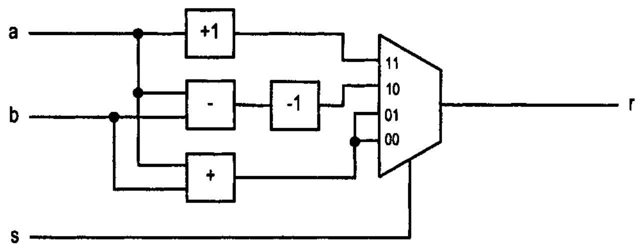


Figure 4.15 Block diagram of example 3.

4.5 CONDITIONAL SIGNAL ASSIGNMENT STATEMENT VERSUS SELECTED SIGNAL ASSIGNMENT STATEMENT

4.5.1 Conversion between conditional signal assignment and selected signal assignment statements

From the synthesis point of view, the conditional signal assignment statement and the selected signal assignment statement imply two different routing structures. The examples presented in the previous sections show that we can describe the same circuit using either a conditional or a selected signal assignment statement. Actually, the conversion between the two forms of assignment statements is always possible.

Converting a selected signal assignment statement to a conditional signal assignment statement is straightforward. Consider a general selected signal assignment statement in which there are eight possible choices: c7, c6, . . ., c1, c0.

```
with sel select
   sig <= value_expr_0 when c0,
          value_expr_1 when c1|c3|c5,
          value_expr_2 when c2|c4,
          value_expr_n when others;
```

We can describe the choices of a when clause as a Boolean expression. For example, **when** c2|c4 can be expressed as (sel=c2) **or** (sel=c4). We can then use these Boolean expressions and convert the selected signal assignment statement to a new format:

```
sig <=
   value_expr_0 when (sel=c0) else
   value_expr_1 when (sel=c1) or (sel=c3) or (sel=c5) else
   value_expr_2 when (sel=c2) or (sel=c4) else
   value_expr_n;
```

Converting a conditional signal assignment statement to a selected statement needs more manipulation. Let us consider a general conditional signal assignment statement with three Boolean expressions:

```
sig <= value_expr_0 when bool_exp_0 else
       value_expr_1 when bool_exp_1 else
       value_expr_2 when bool_exp_2 else
       value_expr_n;
```

We need a 3-bit auxiliary selection signal, `sel`, in which each bit represents a Boolean expression. By specifying proper choices, we can preserve the desired priority. The converted code is

```
sel(2) <= '1' when bool_exp_0 else '0';
sel(1) <= '1' when bool_exp_1 else '0';
sel(0) <= '1' when bool_exp_2 else '0';
with sel select
   sig <= value_expr_0 when "100"|"101"|"110"|"111",
          value_expr_1 when "010"|"011",
          value_expr_2 when "001",
          value_expr_n when others;
```

Note that the pattern of the selected signal assignment statement is very similar to a priority encoder except that the request signals are replaced by the auxiliary selection signals generated from the Boolean expressions.

4.5.2 Comparison between conditional signal assignment and selected signal assignment statements

In the selected signal assignment statement, each choice can be considered as a row in a table. Thus, this statement is a good match for a circuit described by a truth table or a truth table–like function table, such as the decoder, truth table and multiplexer examples discussed in Section 4.4. On the other hand, it is less effective when certain input conditions are given preferential treatment. For example, if we examine the priority encoder example of Section 4.4, eight of the 16 ports of the multiplexer are connected to an identical expression.

The conditional signal assignment statement implicitly enforces the order of the operation and is a natural match for a circuit that needs to give preferential treatment for certain conditions or to prioritize the operations. The priority encoder is a good example of this kind of circuit. The conditional signal assignment statement can also handle complicated conditions. For example, we can write

```
pc_next <=
   pc_reg + offset when (state=jump and a=b) else
   pc_reg + 1 when (state=skip and flag='1') else
. . .
```

A conditional signal assignment statement is less effective to describe a truth table since it may "overspecify" the circuit and thus add unnecessary constraints. For example, consider the multiplexer of Section 4.3.1. The original VHDL segment is

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

The code can also be written as

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

or

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

or many other possible variations. These codes give priority to the condition in the first when clause, although it is not part of the original specification. While this type of code is not wrong, the extra constraint may introduce additional circuitry and make synthesis and optimization more difficult.

Ideally, the synthesis software should automatically determine the optimal structure and derive identical gate-level implementation, regardless of the language constructs used in VHDL descriptions. In reality, this is possible only for small, trivial designs. For a general design, we have to be aware of the effect of the statements on the routing and the "layout" of the final implementation. These aspects are illustrated by examples in Chapter 7.

4.6 SYNTHESIS GUIDELINES

- Avoid a closed feedback loop in a concurrent signal assignment statement.
- Think of the conditional signal assignment and selected signal assignment statements as routing structures rather than sequential control constructs.
- The conditional signal assignment statement infers a priority routing structure, and a larger number of when clauses leads to a long cascading chain.
- The selected signal assignment statement infers a multiplexing structure, and a large number of choices leads to a wide multiplexer.

4.7 BIBLIOGRAPHIC NOTES

Since the focus of the book is on synthesis, only synthesis-related aspects of the concurrent signal assignment statement are discussed. The complete discussion on these constructs can be found in *The Designer's Guide to VHDL, 2nd edition*, by P. J. Ashenden.

The discussion in this chapter illustrates the general schemes to realize concurrent signal assignment statements in various routing structures. Individual synthesis software may map certain language constructs to specific hardware architectures. The software vendors sometimes include a "style guide" in their documentation. It shows the mapping between hardware architecture and the VHDL language constructs.

Problems

4.1 Add an enable signal, `en`, to a 2-to-4 decoder. When `en` is `'1'`, the decoder functions as usual. When `en` is `'0'`, the decoder is disabled and output becomes `"0000"`. Use the conditional signal assignment statement to derive this circuit. Draw the conceptual diagram.

4.2 Repeat Problem 4.1, but use the selected signal assignment statement instead.

4.3 Consider a 2-by-2 switch. It has two input data ports, `x(0)` and `x(1)`, and a 2-bit control signal, `ctrl`. The input data are routed to output ports `y(0)` and `y(1)` according to the `ctrl` signal. The function table is specified below.

Input	**Output**		**Function**
`ctrl`	`y1`	`y0`	
00	x1	x0	pass
01	x0	x1	cross
10	x0	x0	broadcast x0
11	x1	x1	broadcast x1

(a) Use concurrent signal assignment statements to derive the circuit.

(b) Draw the conceptual diagram.

(c) Expand it into gate-level circuit and derive the simplified logic expression in sum-of-products format.

4.4 Consider a comparator with two 8-bit inputs, `a` and `b`. The `a` and `b` are with the `std_logic_vector` data type and are interpreted as unsigned integers. The comparator has an output, `agtb`, which is asserted when `a` is greater than `b`. Assume that only a single-bit comparator is supported by synthesis software. Derive the circuit with concurrent signal assignment statement(s).

4.5 Repeat Problem 4.4, but assume that `a` and `b` are interpreted as signed integers.

4.6 We wish to design a shift-left circuit manually. The inputs include `a`, which is an 8-bit signal to be shifted, and `ctrl`, which is a 3-bit signal specifying the amount to be shifted. Both are with the `std_logic_vector` data type. The output `y` is an 8-bit signal with the `std_logic_vector` data type. Use concurrent signal assignment statements to derive the circuit and draw the conceptual diagram.