5 Concurrent Code

5.1 Introduction

Having finished laying out the foundations of VHDL (chapters 1 to 4), we can now concentrate on the design (code) proper.

A combinational logic circuit is one in which the outputs depend solely on the current inputs, therefore exhibiting no memory, as in the feed-forward model of figure 5.1(a). In contrast, a sequential logic circuit is one in which the outputs do depend on previous system state(s), so storage elements are needed, along with a clock signal to control the system evolution and possibly a reset too, as in the model of figure 5.1(b) (the storage elements are usually D-type flip-flops (DFFs)).

VHDL code can be concurrent (parallel) or sequential. Only statements placed inside a PROCESS, FUNCTION, or PROCEDURE (the last two are called subprograms) are executed sequentially. However, because VHDL is inherently concurrent, a PROCESS, as a whole, is also concurrent with respect to any other (external) statements. In other words, a process body or a subprogram call is also a concurrent statement.

Two other concurrent statements are the BLOCK statement and COMPONENT instantiations. These, however, can be viewed as just different ways of organizing the code, without any new internal statements (hence they will be studied in chapter 8, which is in the system-level part of the book).

Apart from the pieces of code mentioned above, there are three purely concurrent statements (they can only be used outside sequential code—that is, outside PROCESS or subprograms), which are WHEN, SELECT, and GENERATE. (Note: See in section 5.10 the new options for WHEN and SELECT specified in VHDL 2008.)

Following the same reasoning, there are four purely sequential statements (they can only be used inside sequential code), which are IF, WAIT, LOOP, and CASE.

While concurrent code is intended only for the design of combinational circuits, sequential code can be used indistinctly to design both sequential and combinational circuits.

Statements for concurrent code (WHEN, SELECT, GENERATE) are studied in this chapter, while statements for sequential code (IF, WAIT, LOOP, CASE) are described in the next.
Remember that in a concurrent code the order of the statements does not matter. For example, if a code uses three concurrent statements, called \( \text{stat1} \), \( \text{stat2} \), and \( \text{stat3} \), then any of the following sequences will render the same physical circuit: \( \{\text{stat1, stat2, stat3}\} = \{\text{stat3, stat2, stat1}\} = \{\text{stat1, stat3, stat2}\} \).

5.2 Using Operators

Operators were discussed in section 4.2 and summarized in figure 4.1. Basically any kind of circuit can be designed using only operators. This approach, however, is viable only for arithmetic circuits or simple logic circuits. In example 5.1, a multiplexer is designed using only logical operators.

Example 5.1: Multiplexer Implemented with Operators

Implement the \( 4 \times 1 \) (four inputs of one bit each) multiplexer of figure 5.2 using only logical operators.

**Solution** This multiplexer’s logical equation (Pedroni 2008) is \( y = sel_1 \cdot sel_0 \cdot x_0 + sel_1' \cdot sel_0 \cdot x_1 + sel_1 \cdot sel_0' \cdot x_2 + sel_1' \cdot sel_0' \cdot x_3 \), which employs only AND, OR, and NOT operators. Its implementation is in lines 13–16 of the code below.

1  --------------------------------------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
ENTITY mux IS
  PORT (x0, x1, x2, x3: IN STD_LOGIC;
    sel: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    y: OUT STD_LOGIC);
END mux;

ARCHITECTURE operators_only OF mux IS
BEGIN
  y <= (NOT sel(1) AND NOT sel(0) AND x0) OR
       (NOT sel(1) AND sel(0) AND x1) OR
       (sel(1) AND NOT sel(0) AND x2) OR
       (sel(1) AND sel(0) AND x3);
END operators_only;

5.3 The WHEN Statement

WHEN is the simplest conditional statement. It is approximately equivalent to the sequential statement IF. A simplified syntax for WHEN is presented below.

```
assignment_expression WHEN conditions ELSE
  assignment_value WHEN conditions ELSE
  ...
```

Examples

```
x <= '0' WHEN rst='0' ELSE
   '1' WHEN a='0' OR b='1' ELSE
   '-'; --don't care
y <= "00" WHEN (a AND b)="01" ELSE
   "11" WHEN (a AND b)="10" ELSE
   "ZZ"; --high impedance
```

Note that multiple conditions (boolean expressions) are accepted in the WHEN statement, which are grouped using AND, OR, and NOT.

The WHEN statement does not require that all input values be specified. However, when implementing combinational circuits (truth tables), it is a good practice to always cover all input options in order to prevent the inference of latches. For such, the keyword OTHERS is usually helpful.
Another sometimes useful keyword for concurrent code is UNAFFECTED, which should be used when no action is to take place. Note, however, that this usually causes the inference of latches, so this keyword should only be used when memorization of the previous system state is indeed wanted.

**Example** Both codes below implement a positive-level D-type latch (Pedroni 2008). The code on the right, however, covers all possible input values, explicitly documenting the fact that a memory is indeed wanted and the inference of a latch was not by accident.

```
q <= '0' WHEN rst='1' ELSE d WHEN clk='1';
q <= '0' WHEN rst='1' ELSE d WHEN clk='1' ELSE UNAFFECTED;
```

In VHDL 2008, WHEN can also be used in sequential code and allows boolean tests. (See details in section 5.10.)

### 5.4 The SELECT Statement

SELECT is another concurrent statement. It is approximately equivalent to the sequential statement CASE. A simplified syntax for SELECT is presented below.

```
WITH identifier SELECT
    assignment_expression WHEN values,
    assignment_value WHEN values,
    ...
```

**Examples**

```
WITH control SELECT
    y <= "000" WHEN 0 | 1,
        "100" WHEN 2 TO 5,
        "Z--" WHEN OTHERS;
WITH (a AND b) SELECT
    y <= "00" WHEN "001",
        "11" WHEN "100",
        UNAFFECTED WHEN OTHERS;
```

As shown in the first example, SELECT allows the use of multiple values (instead of multiple conditions), which can only be grouped with "|" (means "or") or "TO" (for range), as follows:

- `WHEN value1 | value2 | ...` -- `value1` or `value2` or ...
- `WHEN value1 TO value2` -- `range`

The SELECT statement requires that all input values be covered (complete truth table), for which the keyword OTHERS is often helpful.
UNAFFECTED is another sometimes useful keyword, already described in section 5.3. See, however, the observation about the inference of latches made in that section.

As a last remark, any signal assignment (like those in this chapter, for example) can be preceded by a label, which was omitted in the simplified syntaxes above because that is a rarely used practice.

In VHDL 2008, SELECT can also be used in sequential code and the matching SELECT? statement was introduced, which allows the use of don’t care inputs. (See details in section 5.10.)

**Example 5.2: Multiplexer Implemented with WHEN and SELECT**

Implement the same multiplexer of example 5.1, but now with $N$-bit inputs instead of single bit, as shown in figure 5.3. Specify $N$ using GENERIC (section 2.6). Present two solutions: with WHEN and with SELECT. Show also simulation results.

**Solution**  A VHDL code (with two architectures) for this circuit is presented below, under the title `mux` (line 5). $N$ is entered as a generic parameter (line 6), which is used in lines 7 and 9 to establish the size of the input-output buses. Only STD_LOGIC_VECTOR ports (industry standard) are employed in the code. In the first architecture (called `with_WHEN`), the WHEN statement is employed, while in the second architecture (called `with_SELECT`), the SELECT statement is used instead. Note that in both cases all possible input values are covered.

```
1  -------------------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------------------
5  ENTITY mux IS
6    GENERIC (N: INTEGER := 8);
7    PORT (x0, x1, x2, x3: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
8        sel: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
9        y: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
```
Simulation results (for $N = 8$), confirming the correct circuit operation, are included in figure 5.3. Recall that the apparent glitches in the waveform for $y$ are expected because multiple signals are considered at once, whose actual values neither change instantaneously nor change all exactly at the same time.

Only one entity-architecture pair can be synthesized at a time. Therefore, if we want to write more than one architecture for the same entity (as in the example above), we need either to comment all but one out (with "--") or include a CONFIGURATION declaration in the code in order to direct the compiler to the desired units. In the example above, both architectures can be included in the same code if followed by a declaration like that below (placed outside any entities or architectures).

```
BEGIN
  ARCHITECTURE with_SELECT OF mux IS
  BEGIN
    WITH sel SELECT
    y <= x0 WHEN "00",
      x1 WHEN "01",
      x2 WHEN "10",
      x3 WHEN OTHERS;
  END ARCHITECTURE;
END ARCHITECTURE;
```

This declaration causes the compiler to choose the pair `mux-with_WHEN`. After compiling and simulating this code, just change the name of the architecture in the configuration declaration above to test the other architecture. One (minor) disadvantage of this approach is that it does not automatically prevent the compiler from checking the syntax in the unselected unit. (Details about CONFIGURATION will be seen in chapter 8.)
Example 5.3: ALU

An ALU (Arithmetic Logic Unit) is shown in figure 5.4(a), having \( a, b, \text{cin} \) (carry in), and opcode (operation code) as inputs, and \( y \) as output. The desired functionality is expressed in the truth table of figure 5.4(b), where each function is selected by a different value of opcode. Note that the upper eight instructions are logical, while the lower eight are arithmetic. Design this circuit using the concurrent statement SELECT, satisfying the following conditions:

1) The arithmetic operations must be signed.
2) The number of bits for inputs \( a \) and \( b \) must be generic.
3) All ports must be of type STD_LOGIC(_VECTOR) (industry standard).
4) Simulation results must also be included in the solution.

**Solution** Figure 5.4(c) shows a possible ALU implementation (among several other options). The circuit contains two main sections, called logic and arithmetic units, each controlled by the same three LSBs of opcode. The MSB of opcode is employed to control a multiplexer, letting the logic result out when low or the arithmetic result out if high.

A VHDL code for this circuit is presented below, under the title alu (line 6). The number of bits in \( a \) and \( b \) is a generic parameter (line 7), and all ports (lines 8–11) are of type STD_LOGIC(_VECTOR). Because the arithmetic operations were asked to be signed, the package numeric_std (line 4) was included in the library/package declarations. The code proper is divided according to figure 5.4(c)—that is, a logic unit (lines 22–30), an arithmetic unit (lines 32–43), and a multiplexer (lines 45–47).
The implementation of the logic unit is straightforward. However, because the arithmetic unit must be *signed*, the same procedure used in the recommended solution of example 3.9 (see also recommendations in section 5.7) is adopted here; that is, the inputs are explicitly converted from `STD_LOGIC_VECTOR` to `SIGNED` (by type casting, in lines 32–33), they are then processed, and finally the result is converted back to `STD_LOGIC_VECTOR` (at the mux input, line 47, again by type casting).

Note also that because *cin* is `STD_LOGIC`, not `SIGNED`, `NATURAL`, or `INTEGER`, it could not participate directly in the sum of line 43 (observe in the package `numeric_std`, in appendix J, that the overloaded operator "+" does not contain the `SIGNED + STD_LOGIC` option), so a small integer (lines 19 and 34) was created to allow the sum.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 USE ieee.numeric_std.all;
4 -----------------------------------------------------
5 ENTITY alu IS
6   GENERIC (N: INTEGER := 8); --word bits
7   PORT (a, b: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
8       cin: IN STD_LOGIC;
9       opcode: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
10      y: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
11 END ENTITY;
12 -----------------------------------------------------
13 ARCHITECTURE alu OF alu IS
14   SIGNAL a_sig, b_sig: SIGNED(N-1 DOWNTO 0);
15   SIGNAL y_sig: SIGNED(N-1 DOWNTO 0);
16   SIGNAL y_unsig: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
17   SIGNAL small_int: INTEGER RANGE 0 TO 1;
18   BEGIN
19     ------Logic unit:------------
20     WITH opcode(2 DOWNTO 0) SELECT
21        y_unsig <= NOT a WHEN "000",
22        NOT b WHEN "001",
23        a AND b WHEN "010",
24        a OR b WHEN "011",
25        a NAND b WHEN "100",
26        a NOR b WHEN "101",
27        a XOR b WHEN "110",
28        a XNOR b WHEN OTHERS;
29     ------Arithmetic unit:--------
30     a_sig <= SIGNED(a);
31     b_sig <= SIGNED(b);
32     small_int <= 1 WHEN cin='1' ELSE 0;
33   END ARCHITECTURE alu;
34 ```
WITH opcode(2 DOWNTO 0) SELECT
  y_s <= a_s WHEN "000",
  b_s WHEN "001",
  a_s + 1 WHEN "010",
  b_s + 1 WHEN "011",
  a_s - 1 WHEN "100",
  b_s - 1 WHEN "101",
  a_s + b_s WHEN "110",
  a_s + b_s + small_int WHEN OTHERS;
-------Mux:---------------------
WITH opcode(3) SELECT
  y <= y_uns WHEN '0',
  STD_LOGIC_VECTOR(y_s) WHEN OTHERS;
END ARCHITECTURE;
-----------------------------------------------------

Simulation results are depicted in figure 5.5. The upper graph is for logic instructions, while the lower graph exhibits results from arithmetic operations. The reader is invited to examine both to check the correct circuit operation.

5.5 The GENERATE Statement

GENERATE is another concurrent statement. In its most popular form (unconditional GENERATE), it is equivalent to the sequential statement LOOP (chapter 6) in the sense that it too is employed to have a section of code repeated a number of times. GENERATE also allows the inclusion of an IF condition (conditional GENERATE), hence with some
similarity to the combination of the *sequential* statements LOOP and IF. Both forms can be nested inside one another.

*Unconditional GENERATE* (also called FOR-GENERATE) is used to create multiple instances of a section of code. A simplified syntax for it is shown below. Notice that a label is required, and that the word BEGIN is only needed when declarations are made.

```
label: FOR identifier IN range GENERATE
    [declarative_part
    BEGIN]
    concurrent_statements_part
END GENERATE [label];
```

**Example** Below, three signals are declared, then three equivalent sections of code utilizing the GENERATE statement are presented. In all three the label is `gen`, the identifier is `i`, and the range is 0-to-7 or 7-downto-0.

```
SIGNAL a, b, x: BIT_VECTOR(7 DOWNTO 0);
---------------------------------------
gen: FOR i IN 0 TO 7 GENERATE
    x(i) <= a(i) XOR b(7-i);
END GENERATE;
---------------------------------------
gen: FOR i IN a'RANGE GENERATE
    x(i) <= a(i) XOR b(7-i);
END GENERATE;
---------------------------------------
gen: FOR i IN a'REVERSE_RANGE GENERATE
    x(i) <= a(i) XOR b(7-i);
END GENERATE;
```

*Conditional GENERATE* (also called IF-GENERATE) includes an IF statement in the GENERATE loop. A simplified syntax is shown below.

```
label: IF condition GENERATE
    [declarative_part
    BEGIN]
    concurrent_statements_part
END GENERATE [label];
```
This version of GENERATE is of limited interest. To make it more interesting, additional features were specified in VHDL 2008, allowing the use of ELSIF/ELSE, plus the use of alternative labels and the END keyword before END GENERATE. Another option, called CASE-GENERATE, was also introduced in VHDL 2008 (see details in section 5.10).

Another important remark about GENERATE (and the same is true for LOOP, which will be studied in the next chapter) is that both range limits are normally required to be static. For example, say that in the section of code below $x$ is an input (therefore, a non-static parameter). Then this code might not be synthesizable.

```
-------------------------------
NotOK: FOR i IN 0 TO x GENERATE
    ...
END GENERATE;
-------------------------------
```

It is also important to be aware of multiply-driven signals. As will be seen in chapters 6 and 7, multiple assignments to the same VARIABLE are fine because its value is updated immediately, but that is not allowed for SIGNAL. This problem is illustrated in the example below. In section 7.7, a way of circumventing such a limitation will be introduced.

**Example** The first of the three sections of code below is fine because values are assigned to each bit of $x$ (a signal) only once. However, the second is not correct, because a value is assigned to $y$ several (up to four) times. The same applies to the third section of code, where $z$ might also receive up to four assignments.

```
-----------------------------------------------
SIGNAL a, b, x, y: BIT_VECTOR(3 DOWNTO 0);
SIGNAL z: INTEGER RANGE 0 TO 7;
-----------------------------------------------
OK: FOR i IN x'RANGE GENERATE
    x(i)<'1' WHEN (a(i) AND b(i))='1' ELSE '0';
END GENERATE;
-----------------------------------------------
NotOK: FOR i IN y'LOW TO y'HIGH GENERATE
    y <="1111" WHEN (a(i) AND b(i))='1' ELSE
    "0000";
END GENERATE;
-----------------------------------------------
NotOK: For i IN 0 TO 3 GENERATE
    z <= z + 1 WHEN a(i)='1';
END GENERATE;
-----------------------------------------------
```
We conclude this section by presenting two complete design examples using GENERATE.

**Example 5.4: Generic Address Decoder with GENERATE**

Redesign the generic address decoder of example 2.4, this time using only STD_LOGIC-based ports (industry standard).

**Solution** A VHDL code for this problem is presented below. All ports are STD_LOGIC-based (lines 8–10). The GENERATE statement is employed in lines 17–19, containing just one assignment (using WHEN, line 18). Note that `address` was converted into an integer in line 16 using a functions available in the package `std_logic_unsigned` (see figure 3.10). Lines 14 and 16 can obviously be suppressed if we choose to write "...WHEN i=conv_integer(address) ..." in line 18. The size of the code is fixed, regardless of the number of input bits. Simulation results are similar to those in figure 2.7.

```vhdl
1  ENTITY address_decoder IS
2    GENERIC (N: NATURAL := 3); -- number of address bits
3    PORT (address: IN STD_LOGIC_VECTOR (N-1 DOWNTO 0);
4       ena: IN STD_LOGIC;
5       word_line: OUT STD_LOGIC_VECTOR(2**N-1 DOWNTO 0));
6    END ENTITY;
7  ARCHITECTURE decoder OF address_decoder IS
8    SIGNAL addr: NATURAL RANGE 0 TO 2**N-1;
9    BEGIN
10       addr <= conv_integer(address);
11       gen: FOR i IN word_line'RANGE GENERATE
12          word_line(i)<'0' WHEN i=addr AND ena='1' ELSE '1';
13    END GENERATE;
14  END ARCHITECTURE;
```

A very useful application for GENERATE is in the instantiation of components to build larger, structural circuits. Even though COMPONENT will be studied in chapter 8, a preliminary example is presented below in order to illustrate the usage of GENERATE in this kind of design.
Example 5.5: COMPONENT Instantiation with GENERATE

Figure 5.6 illustrates the construction of a larger multiplexer using multiple instances of a basic unit. In (a), a $2 \times 1$ (two inputs of one bit each) mux is shown, which is instantiated three times in (b), resulting the $2 \times 3$ mux of (c). Design this circuit using a structural approach (that is, with COMPONENT instantiations), with GENERATE employed to make the instantiations.

Solution  

A VHDL code for this circuit is shown below, consisting of two parts. The first part builds the basic unit (`mux2x1`), which is then instantiated in the second part (main code) using the GENERATE statement (lines 18–20). (Details on COMPONENT construction and usage will be seen in chapter 8.)

```vhdl
1 -----The component (mux2x1):------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------
5 ENTITY mux2x1 IS
6 PORT (a, b, sel: IN STD_LOGIC;
7 x: OUT STD_LOGIC);
8 END ENTITY;
9 ---------------------------------
10 ARCHITECTURE mux2x1 OF mux2x1 IS
11 BEGIN
12 x<=a WHEN sel='0' ELSE b;
13 END ARCHITECTURE;
14 ---------------------------------
15 -------Main code:---------------------
16 LIBRARY ieee;
17 ```

Concurrent Code

Figure 5.6

(a) A $2 \times 1$ mux that (b) instantiated three times (c) creates a $2 \times 3$ mux.
3 USE ieee.std_logic_1164.all;
4 -------------------------------------------------------
5 ENTITY mux2x3 IS
6 PORT (a, b: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
7     sel: IN STD_LOGIC;
8     x: OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
9 END ENTITY;
10 -------------------------------------------------------
11 ARCHITECTURE mux2x3 OF mux2x3 IS
12 ---Component declaration:-----
13 COMPONENT mux2x1 IS
14 PORT (a, b, sel: IN STD_LOGIC; x: OUT STD_LOGIC);
15 END COMPONENT;
16 BEGIN
17 ---Component instantiation:---
18 generate_mux2x3: FOR i IN 0 TO 2 GENERATE
19     comp: mux2x1 PORT MAP (a(i), b(i), sel, x(i));
20 END GENERATE generate_mux2x3;
21 END ARCHITECTURE;
22 -------------------------------------------------------

5.6 Implementing Sequential Circuits with Concurrent Code

In principle, only combinational circuits should be implemented with concurrent code. We know, however, that using only NAND or NOR gates any digital circuit can be constructed. Since sequential logic circuits are digital circuits, and since NAND or NOR gates can be easily constructed with concurrent code, then sequential circuits can obviously also be constructed with pure concurrent code. However, this approach is only viable for simple circuits, because in general the code would be much longer, more complex to write and debug, and unnatural to follow. In conclusion, the use of concurrent code to design sequential circuits is in general not recommended.

Example 5.6: DFF Implemented with Concurrent Code

Figure 5.7 shows, on the left, a pair of multiplexers (combinational circuits), whose connections emulate a DFF (a sequential circuit), shown on the right. Write a VHDL code for this circuit and examine the expressions inferred by the compiler.

Solution A corresponding VHDL code is presented below. The intermediate signal p is specified in line 8, and the multiplexers are implemented in lines 10–11 using the WHEN statement. Looking at the compilation report (fitter equations), the expected result is the inference of two serially connected latches, because a multiplexer with a feedback loop (as in figure 5.7) emulates a latch. In conclusion, the resulting circuit should present the cor-
rect functionality, but because of its construction, it is slower than an actual, prefabricated DFF (the latter is optimized to operate specifically as a DFF).

```vhdl
-- --------------------------------------------
1 ENTITY concurrent_dff IS
2 PORT (d, clk: IN BIT;
3 q: BUFFER BIT);
4 END ENTITY;
5 -- --------------------------------------------
6 ARCHITECTURE concurrent OF concurrent_dff IS
7 SIGNAL p: BIT;
8 BEGIN
9 p <= d WHEN clk='0' ELSE p; --1st mux
10 q <= p WHEN clk='1' ELSE q; --2nd mux
11 END ARCHITECTURE;
12 -- --------------------------------------------
```

5.7 Implementing Arithmetic Circuits with Operators

Arithmetic circuits are normally constructed using only arithmetic operators (\(+\), \(-\), \(*\), \(/\), **, ABS, REM, and MOD, described in section 4.2). The first four in this list are by far the most frequently used, so special attention is dedicated to them in this section.

The main `interface` (PORT) types for arithmetic circuits are INTEGER (should be avoided) and STD_LOGIC_VECTOR (industry standard). On the other hand, internally, the preferred types are UNSIGNED and SIGNED (as seen in chapter 3, these types are defined in the packages numeric_std [preferred] and std_logic_arith).

To use such operators, it is necessary to know the size (number of bits) required for the result as a function of the operands’ sizes. For example, below are definitions for four functions (\(+\), \(-\), \(*\), \(/\)), copied from the package numeric_std, for the case of signed inputs and signed output (\(L\) and \(R\) represent the left and right operands).

```vhdl
-- --------------------------------------------
FUNCTION "+" (L, R: SIGNED) RETURN SIGNED;
--Result SUBTYPE: SIGNED(MAX(L'LENGTH, R'LENGTH)-1 DOWNTO 0)
```

Figure 5.7
DFF implemented with multiplexers (example 5.7).
The conclusions from the definitions above are:

1) For "+" and "-": The size of the result must be equal to the size of the largest operand.
2) For "*": The size of the result must be equal to the sum of the operands’ sizes.
3) For "/": The size of the result must be equal to the size of the numerator.

As already mentioned, a fundamental aspect of arithmetic circuits is their *nature*, which can be *unsigned* or *signed*. As seen in section 1.8, the range covered with $N$ bits in the former is from 0 to $2^N - 1$, while in the latter it goes from $-2^{N-1}$ to $2^{N-1} - 1$. For example, with 4 bits, the range is from 0 to 15 when unsigned or from $-8$ to $+7$ if signed.

Negative numbers are represented in two’s complement form. Consequently, addition and subtraction are essentially the same function because the latter is just the former preceded by a two’s complement operation. Some examples are shown below for a 4-bit *signed* system.

\begin{align*}
5 + 2 &= "0101" + "0010" = "0111" = 7 \\
5 - 2 &= 5 + (-2) = "0101" + "1110" = "0011" = 3 \\
5 - (-2) &= 5 + 2 = "0111" = 7 \\
-5 + 2 &= "1011" + "0010" = "1101" = -3
\end{align*}

Multiplication and division with signed numbers involve again two’s complement operations. A negative number must be two’s complemented to attain its absolute value. Then the multiplication or division is performed, with the result two’s complemented again if the result is to be negative (that is, if the signs of the operands are different). Some examples are shown below, again for a *signed* system with 4-bit inputs.

\begin{align*}
5*3 &= "0101" * "0011" = "00001111" = 15 \\
-5*3 &= -(5*3) = -("0101" * "0011") = -("00001111") = "11100011" = -15 \\
5/3 &= "0101" / "0011" = "0001" = 1 \\
-5/3 &= -(5/3) = -("0101" / "0011") = -("0001") = "1111" = -1
\end{align*}
Because multiplication and division are not subject to overflow (due to the number of bits adopted in the respective functions), the corresponding VHDL code is straightforward. However, that is not the case for addition and subtraction, in which there are also carry-in/out bits to be taken care of.

Figure 5.8 shows three diagrams for an adder/subtracter. In (a), the operands are \( a \) and \( b \) (eight bits each), the carry-in bit is \( c_{in} \), the sum and its carry-out bit are \( \text{sum} \) and \( c_{out_{sum}} \), and the subtraction and its carry-out are \( \text{sub} \) and \( c_{out_{sub}} \). An equivalent representation appears in (b), in which the carry-out bits are combined with \( \text{sum} \) and \( \text{sub} \) (note that now they have 9 bits; the MSB is \( c_{out} \)). Finally, in (c), there are no carry bits, so this circuit is subject to overflow.

Assume that it is an unsigned system. Then the sum in figure 5.8(a) can be computed as follows:

```
SIGNAL a_uns, b_uns, sum: UNSIGNED(7 DOWNTO 0);
SIGNAL sum_uns: UNSIGNED(9 DOWNTO 0);
SIGNAL cin, cout_sum: STD_LOGIC;
sum_uns <= ('0' & a_uns & cin) + ('0' & b_uns & '1');
sum <= sum_uns(8 DOWNTO 1);
cout_sum <= sum_uns(9);
```

Note in the code above that the operands for \( \text{sum}_\text{uns} \) have 10 bits, attained by appending \( c_{in} \) or '1' on the right and '0' on the left. The LSB of \( \text{sum}_\text{uns} \) is discarded and its MSB is \( c_{out_{sum}} \).

Assuming now that the system is signed, the following can be done:

```
SIGNAL a_sig, b_sig, sum: SIGNED(7 DOWNTO 0);
SIGNAL sum_sig: SIGNED(9 DOWNTO 0);
SIGNAL cin, cout_sum: STD_LOGIC;
sum_sig <= (a_sig(7) & a_sig & cin) + (b_sig(7) & b_sig & '1');
sum <= sum_sig(8 DOWNTO 1);
cout_sum <= sum_sig(9);
```

Figure 5.8
Adder/subtracter with the carry-out bits (a) separated from \( \text{sum} \) and \( \text{sub} \), (b) grouped with \( \text{sum} \) and \( \text{sub} \), and (c) inexistent. The first two are equivalent and not subject to overflow, whereas in the last one overflow can occur.
Note that the only difference in this code with respect to the previous code is in the 
sign-extension bit, which must now be a copy of the leftmost bit (see binary arithmetic algo-
rithms in Pedroni (2008)).

The problem with the above approaches is that the expressions do not work for subtrac-
tion, which is often necessary, especially in signed systems. Consequently, a more general 
approach is needed. A solution is shown below. Note in the expression for \( sub \_sig \) that all 
three operands (including \( cin \)) were sign-extended (a '0' is used for \( cin \) because it is a non-
negative number), so both additions and subtractions can now be performed. A complete 
design example will be shown shortly.

```vhdl
SIGNAL a\_sig, b\_sig, sum, sub: SIGNED(7 DOWNTO 0);
SIGNAL sum\_sig, sub\_sig: SIGNED(8 DOWNTO 0);
SIGNAL cin, cout\_sum, cout\_sub: STD\_LOGIC;
sum\_sig <= (a\_sig(7) & a\_sig) + (b\_sig(7) & b\_sig) + ('0' & cin);
sum <= sum\_sig(7 DOWNTO 0);
cout\_sum <= sum\_sig(8);
sub\_sig <= (a\_sig(7) & a\_sig) - (b\_sig(7) & b\_sig) + ('0' & cin);
sub <= sub\_sig(7 DOWNTO 0);
cout\_sub <= sub\_sig(8);
```

Contrary to the \(+\) and \(-\) operators for integer arithmetic, fixed-point and floating-point 
arithmetic are guaranteed to be overflow free because the size of the output vector in both 
\(+\) and \(-\) operations is defined to be one unit larger than the largest input (see comments 
on arithmetic operators in section 3.8). The case of integer arithmetic is further covered in 
the recommendations and design example below.

Based on the above discussion, plus the note at the end of example 3.4 and the contents 
of example 3.9, the following is recommended:

1) For the interfaces (PORT specifications), use only STD\_LOGIC(_VECTOR) (industry 
standard).
2) Internally, use only (UN)SIGNED.
3) For the type above, use the package numeric\_std (standardized by IEEE).
4) Before performing any computation, explicitly convert the data from 
STD\_LOGIC(_VECTOR) to (UN)SIGNED. This can be done with type casting (section 
3.18).
5) Make the computations.
6) Finally, return the result to STD\_LOGIC(_VECTOR). Type casting can again be used.
Example 5.7: Recommended Adder/Subtracter Implementation

Write a VHDL code that implements the adder/subtracter of figure 5.8(a) or 5.8(b) (physically, they are equal). In the design, follow the recommendations just presented. Assume that it is part of a signed system.

Solution A VHDL code for this circuit is presented below, under the title signed_add_sub (line 6). Because the circuit is signed, the package numeric_std was included in line 4. The number of bits in the operands is a generic parameter (line 7). The type of all ports is STD_LOGIC(_VECTOR) (lines 8–12). The inputs are $a$, $b$, and cin, while the outputs are sum and sub.

The code proper (lines 20–33) is organized in four parts. In the first part (lines 21–22), the operands are explicitly converted to SIGNED. In the second part (lines 24–25), they are added and subtracted, with carry-in included. In the third part (lines 27–28), the signals are converted back to STD_LOGIC_VECTOR, with the carry-out bits grouped with sum and sub, as in figure 5.8(b). The forth part (lines 30–33) is equivalent to the third, just with the carry-out bits separated from sum and sub, as in figure 5.8(a). To have the code resemble figure 5.8(a) instead of 5.8(b), just comment out lines 10, 27, and 28 and uncomment lines 11–12 and 30–33. Simulation results are shown in figure 5.9.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
-----------------------------------------------------------------------
ENTITY signed_add_sub IS
  GENERIC (N: INTEGER := 4); --number of input bits
  PORT (a, b: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
    cin: IN STD_LOGIC;
    sum, sub: OUT STD_LOGIC_VECTOR(N DOWNTO 0));
  --sum, sub: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
  --cout_sum, cout_sub: OUT STD_LOGIC);
-----------------------------------------------------------------------
```

Figure 5.9
Simulation results from the adder (or subtracter) of example 5.7.
ARCHITECTURE signed_add_sub OF signed_add_sub IS
  SIGNAL a_sig, b_sig: SIGNED(N-1 DOWNTO 0);
  SIGNAL sum_sig, sub_sig: SIGNED(N DOWNTO 0);
BEGIN
  -----convert to signed:--------------
  a_sig <= signed(a);
  b_sig <= signed(b);
  -----add and subtract:---------------
  sum_sig <= (a_sig(N-1) & a_sig) + (b_sig(N-1) & b_sig) + ('0' & cin);
  sub_sig <= (a_sig(N-1) & a_sig) - (b_sig(N-1) & b_sig) + ('0' & cin);
  -----output option #1:---------------
  sum <= std_logic_vector(sum_sig);
  sub <= std_logic_vector(sub_sig);
  -----output option #2:---------------
  --sum <= std_logic_vector(sum_sig(N-1 DOWNTO 0));
  --cout_sum <= std_logic(sum_sig(N));
  --sub <= std_logic_vector(sub_sig(N-1 DOWNTO 0));
  --cout_sub <= std_logic(sum_sig(N));
END ARCHITECTURE;

5.8 Preventing Combinational-Logic Simplification

A series of synthesis attributes were described in section 4.6; namely:

- `enum_encoding` attribute
- `chip_pin` attribute
- `keep` attribute
- `preserve` attribute
- `noprune` attribute.

As shown there, the `keep` attribute can be used to tell the compiler not to simplify (suppress) specific nodes that would otherwise be removed during the optimization process. Its application was illustrated in example 4.4, in which a delay line was constructed. Another example is included (example 5.8), followed by another (less objective) approach that does not make use of the `keep` attribute.

**Example 5.8: Short-Pulse Generator with the `keep` Attribute**

Figure 5.10(a) shows a typical circuit used for pulse-shortening in pulse-based flip-flop implementations (Pedroni 2008). Note that the delay line contains three inverters, which
would be reduced to just one by the compiler if no measure were taken to prevent it. Design this circuit using the `keep` attribute to preserve nodes \( a \), \( b \), and \( c \). Include simulation results in your solution. Before starting, draw in figure 5.10(b) the expected waveforms at nodes \( c \) and \( \text{short clk} \). Assume that the propagation delay through each inverter and through the AND gate is 1 ns, which is the distance between the vertical lines in the figure.

**Solution** It is left to the reader to fill figure 5.10(b). A VHDL code for this circuit is presented below. Note the use of `keep` in lines 9–10, telling the compiler to preserve nodes \( a \), \( b \), and \( c \). Observe also that the code proper is not altered by the presence of this attribute. The RTL view produced by the compiler is shown in figure 5.11, along with simulation results. Does the overall shape of \( \text{short clk} \) in figure 5.11 coincide with your sketch in figure 5.10(b)?

```vhdl
1 ENTITY short_pulse_gen IS
2   PORT (clk: IN BIT;
3       short_clk: OUT BIT);
4 END ENTITY;
5
6 ARCHITECTURE short_pulse OF short_pulse_gen IS
7   SIGNAL a, b, c: BIT;
8   ATTRIBUTE keep: BOOLEAN;
9   ATTRIBUTE keep OF a, b, c: SIGNAL IS TRUE;
```

Figure 5.10
Short-pulse generator of example 5.8.

Figure 5.11
RTL view and simulation results from the short-pulse generator of example 5.8.
Another (but less generic, shown here for Quartus II) solution for the problem above can be devised with the LCELL primitive, which consists of a buffer that can be inserted into the signal path. This primitive is instantiated as a COMPONENT (chapter 8), with the following specifications:

```
COMPONENT LCELL
  PORT (a_in: IN STD_LOGIC;
       a_out: OUT STD_LOGIC);
END COMPONENT;
```

**Example 5.9: Short-Pulse Generator with the LCELL Primitive**

Redesign the circuit of example 5.8, this time employing the LCELL primitive instead of the keep attribute.

**Solution**  A VHDL code for this problem is shown below. The COMPONENT declaration for LCELL (lines 12–15) is in the declarative part of the architecture (this megafunction is available in the `altera_mf_components.vhd` file). Six auxiliary signals are declared in line 11 instead of three because the wires must be broken in order to insert the buffers. In the code proper, three COMPONENT instances of LCELL are created (lines 18, 20, 22). The inferred circuit is similar to that in the previous example.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY short_pulse_gen IS
  PORT (clk: IN STD_LOGIC;
        short_clk: OUT STD_LOGIC);
END ENTITY;

ARCHITECTURE short_pulse OF short_pulse_gen IS
  SIGNAL a1, a2, b1, b2, c1, c2: STD_LOGIC;
  COMPONENT LCELL IS
    PORT (a_in: IN STD_LOGIC;
          a_out: OUT STD_LOGIC);
  END COMPONENT;
  BEGIN
    a <= NOT clk;
    b <= NOT a;
    c <= NOT b;
    short_clk <= clk AND c;
  END ARCHITECTURE;
```
BEGIN
a1 <= NOT clk;
buffer_a: COMPONENT LCELL PORT MAP (a1, a2);
b1 <= NOT a2;
buffer_b: COMPONENT LCELL PORT MAP (b1, b2);
c1 <= NOT b2;
buffer_c: COMPONENT LCELL PORT MAP (c1, c2);
short_clk <= clk AND c2;
END ARCHITECTURE;
-----------------------------------------------

5.9 Allowing Multiple Signal Assignments

As will be seen in section 7.4, only one assignment can be made to a SIGNAL, while a VARIABLE allows multiple assignments. One way of circumventing such a limitation will be introduced in section 7.7. To solve some of the exercises in section 5.11, a preview of section 7.7 might be helpful.

5.10 VHDL 2008

With respect to the material covered in this chapter, the main additions specified in VHDL 2008 are those listed below.

1) The concurrent WHEN and SELECT statements can be used also in sequential code. For example, WHEN can replace IF or can be used inside IF, while SELECT can replace CASE.

2) WHEN allows boolean tests (the only consequence of this is sometimes a slightly shorter code, at the cost of reduced code clarity). For example, the two codes below are equivalent (the traditional format is on the left).

\[
\begin{align*}
x & \leq '0' \text{ WHEN } rst='0' \text{ ELSE } '1' \text{ WHEN } a='0' \text{ OR } b='1' \text{ ELSE } '-' \\
x & \leq '0' \text{ WHEN NOT } rst \text{ ELSE } '1' \text{ WHEN NOT } a \text{ OR } b \text{ ELSE } '-' 
\end{align*}
\]

3) The matching "SELECT?" statement was introduced, which allows the use of don’t care inputs. An example is shown below.

WITH interrupt SELECT?
priority <= 4 WHEN "1---",
3 WHEN "01--",
2 WHEN "001-",
1 WHEN "0001",
0 WHEN OTHERS;
4) In the conditional IF-GENERATE statement, the use of ELSIF/ELSE is allowed. A simplified syntax is shown below (on the left).

```
label: IF condition GENERATE
    [declarative_part
    BEGIN
    concurrent_statements_part
    [ELSIF condition GENERATE
    [declarative_part
    BEGIN
    concurrent_statements_part]
    [ELSE GENERATE
    [declarative_part
    BEGIN
    concurrent_statements_part]
    END GENERATE [label];
```

5) A new form of conditional GENERATE, called CASE-GENERATE, was also introduced. A simplified syntax is shown above (on the right).

6) The use of alternative labels for IF-GENERATE and the use of END before END GENERATE are both allowed.

### 5.11 Exercises

The exercises proposed in this section are to be solved using only truly concurrent code (that is, with WHEN, SELECT, and GENERATE, plus operators, of course). See also the comment in section 5.9.

*Note:* For exercise solutions, please consult the book website.

**Exercise 5.1: Circuit with ‘Don’t Care’ Outputs**

Figure 5.12 shows a diagram for a combinational circuit that must compute the function described in the accompanying truth table. Note that some outputs are marked as ‘don’t care’.

a) Using Karnaugh maps, derive the optimal boolean expressions for both output bits ($y_1$ and $y_0$).

b) Which data type should be used in the VHDL code in order to take advantage of the ‘don’t care’ states?
c) Design this circuit using WHEN or SELECT. After compiling and testing the code, compare the equations implemented by the fitter against those obtained above.

Exercise 5.2: Dual-Edge Flip-Flop

Using concurrent code, design the dual-edge flip-flop of example 7.6 (figure 7.4).

Exercise 5.3: Generic AND and NAND Gates


Exercise 5.4: Generic Parity Generator

The circuit in figure 5.13 has an $N$-bit input $x$, from which an $(N + 1)$-bit output $y$ must be produced. The circuit must detect the parity of $x$, then add an extra bit to it (on the left) such that the final parity (number of '1's) is odd. Design this circuit using concurrent code. Enter $N$ as a generic parameter, so the code can be easily adjusted to any input size. (Suggestion: see section 7.7.)

Exercise 5.5: Parity Generator with Automated Pin Allocation

Assume the case of $N = 4$ in the exercise above (so the circuit has a total of nine ports). Using the chip_pin attribute (section 4.6), make proper pin assignments in the code such that all nine ports are automatically routed to the desired pins (check the target device’s pin list to select pins that are available to the user).

Exercise 5.6: Generic Binary-to-Gray Converter

The regular binary code, which consists of code words ordered according to their increasing unsigned decimal values, constitutes the most commonly used digital code. In some
applications, however, gray code (Pedroni 2008), which is an UDC (Unit Distance Code) because any two adjacent code words differ by just one bit, might be preferred (in electromechanical applications, for example). This exercise deals with the design of a circuit capable of converting regular binary code into gray code.

a) Just to practice with gray code, make a table with two columns, placing the 16 four-bit binary entries in the first column, and the corresponding gray values in the second column.

b) In order to implement a generic converter, find a closed-form expression for binary-to-gray conversion. Use it to check your entries in the second column for part (a).

c) Write a VDHL code from which this converter can be inferred. Enter \( N \) (number of bits) as a GENERIC parameter, and use the closed-form expression obtained above to do the computations. Include simulation results in your solution.

**Exercise 5.7: Hamming Weight with GENERATE**

The **Hamming weight** of a vector is the number of '1's in it. Design a circuit that computes that number for a generic-length vector using only concurrent code. (Suggestion: see section 7.7.)

**Exercise 5.8: Binary Sorter with GENERATE**

Using concurrent code, design a circuit capable of ordering the bits of a bit vector. The ordering should be from left to right, with all '1's coming first (for example, "00011001" would become "11100000"). (Suggestions: solve exercise 5.7 first; see section 7.7)

**Exercise 5.9: ALU with WHEN**

Redesign the ALU of example 5.3 using the WHEN statement instead of SELECT.

**Exercise 5.10: Arithmetic Circuit with INTEGER**

Figure 5.14(a) shows an arithmetic circuit that must be designed to produce the computations specified in the truth table of figure 5.14(b) (it is a “mini-ALU”, with just the arithmetic unit). Write a VHDL code for this circuit under the following constraints:
1) The code must be truly concurrent;
2) All ports must be specified as INTEGER. Note that in this case the output can have more bits than the inputs, so no separate carry-out computation is needed. Note also that some operations are signed.

Exercise 5.11: Arithmetic Circuit with STD_LOGIC

Figure 5.14(c) shows an arithmetic circuit similar to that in the previous exercise, the only difference being that now the output has the same number of bits as the inputs, so a separate wire is needed for the carry-out bit. Design this circuit such that it performs the same operations listed in figure 5.14(b), under the following constraints:

1) Again, the code must be truly concurrent;
2) All ports must be specified as STD_LOGIC(_VECTOR) (industry standard).

Exercise 5.12: Barrel Shifter with INTEGER and BIT_VECTOR

A barrel shifter (Pedroni 2008) is a circuit capable of shifting an input word to the right or to the left by a certain number of bit positions. A top level diagram is shown in figure 5.15, with inputs $x$ (word to be shifted) and $shift$ (number of positions to be shifted), and output $y$ (shifted word). Assuming that our shifter is unregistered (that is, does not contain memory, so it is not dependent on a clock signal) and that the empty positions must be filled with zeros, write a VHDL code from which this circuit can be inferred. Represent the number of bits by $N$ in $x$ and $y$ and $M$ in $shift$ (where $2^M = N$). These parameters must be generic, and the total amount of shift should be allowed to be as large as $N - 1$. Solve this exercise with $shift$ declared as INTEGER and $x$ and $y$ as BIT_VECTOR. Are the shift operators SLL and SRL helpful here?

Exercise 5.13: Barrel Shifter with STD_LOGIC_VECTOR

Repeat the exercise above, this time with all ports specified as STD_LOGIC_VECTOR.

Exercise 5.14: Recommended Unsigned Adder/Subtractor Implementation

Write a VHDL code that implements the adder/subtractor of figure 5.8(a) or 5.8(b) for an unsigned system. In the design, follow the recommendations presented in section 5.7.
Exercise 5.15: Recommended Unsigned Multiplier Implementation

Figure 5.16(a) shows a multiplier. Write a VHDL code that implements this circuit for an unsigned system. In the design, follow the recommendations presented in section 5.7.

Exercise 5.16: Recommended Signed Multiplier Implementation

Figure 5.16(a) shows a multiplier. Write a VHDL code that implements this circuit for a signed system. In the design, follow the recommendations presented in section 5.7.

Exercise 5.17: Recommended Unsigned Divider Implementation

Figure 5.16(b) shows a divider. Write a VHDL code that implements this circuit for an unsigned system. In the design, follow the recommendations presented in section 5.7.

Exercise 5.18: Recommended Signed Divider Implementation

Figure 5.16(b) shows a divider. Write a VHDL code that implements this circuit for a signed system. In the design, follow the recommendations presented in section 5.7.

Exercise 5.19: Frequency Multiplier with the keep Attribute

Figure 5.17 shows two short-pulse generators similar to that in figure 5.10, connected in parallel and operating with complementary versions of the clock.

a) Examine this circuit then draw the expected waveforms for $c_1$, $y_1$, $c_2$, $y_2$, and $clk\_out$.

b) Design this circuit with concurrent code, using keep to preserve internal nodes. Does the resulting waveform for $clk\_out$ match yours?
**Exercise 5.20: Generic Multiplexer**

Consider the multiplexer implemented in example 5.2 (figure 5.3), which has four inputs with an arbitrary number of bits \( N \) per input. To make the design truly generic, consider the case where the number of inputs is also arbitrary \( M \). Redesign that circuit with \( M \) and \( N \) declared as GENERIC constants, in the following two situations:

a) Using a 1D \( \times \) 1D data array for the input (recall that the only predefined data types with dimension 1D \( \times \) 1D are STRING and INTEGER_VECTOR, which are of no interest in the present example).

b) Using a predefined data type.

**Exercise 5.21: INOUT bus**

Solve exercise 13.1.

**Exercise 5.22: INOUT versus BUFFER**

Solve exercise 13.2.

**Exercise 5.23: Floating-Point Adder**

This exercise concerns the “backward” construction of a floating-point (FP) adder. (Suggestion: see first the example in section 3.8.)

a) Figure 5.18 shows simulation results from a FP adder that computes \( x = a + b \), where all signals are expressed with 4 bits for the exponent and also 4 bits for the fraction, that is, \((S)(EEEE)(FFFF)\). Examine each result in the figure and check its correctness. The non-exact values obtained in the simulation are wrong or is this something inherent of FP numbers?

b) Design such an adder, then simulate it and check whether similar results are obtained.

---

**Figure 5.18**

<table>
<thead>
<tr>
<th>Name</th>
<th>0 ps</th>
<th>80.0 ns</th>
<th>160.0 ns</th>
<th>240.0 ns</th>
<th>320.0 ns</th>
<th>400.0 ns</th>
<th>480.0 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>001101000</td>
<td>01001010</td>
<td>011100001</td>
<td>011110110</td>
<td>111100000</td>
<td>111101110</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>010011011</td>
<td>01100010</td>
<td>011010010</td>
<td>011010110</td>
<td>111100000</td>
<td>111101111</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>010011010</td>
<td>01100010</td>
<td>011010010</td>
<td>011010110</td>
<td>111100000</td>
<td>111101111</td>
<td></td>
</tr>
</tbody>
</table>

Expected: 5.5+0.75 = 6.25
Expected: 5.5+5.5 = 11
Obtained: 6.25 (exact)
Obtained: 11 (exact)

Expected: 5.5+135 = 141.5
Expected: 5.5+240 = 245.5
Obtained: 144 (not exact)
Obtained: 248 (not exact)

Expected: 5.5--135 = \(-130.5\)
Expected: 5.5--240 = \(-234.5\)
Obtained: \(-128\) (not exact)
Obtained: \(-232\) (not exact)
Exercise 5.24: Floating-Point Adder, Subtractor, Multiplier, Divider

This exercise concerns the design of a single-precision (32-bit option of IEEE 754) adder, subtracter, multiplier, and divider circuit. (Suggestion: see first the example in section 3.8.)

a) Among the four implementations, that is, \( a + b \), \( a - b \), \( a \times b \), and \( a/b \), which ones do you expect to require the smallest and the largest amount of hardware?

b) Design a single-precision FP adder that computes \( x = a + b \). Compile the circuit in a low-cost (say, Cyclone II or Spartan 3A) and also in a high-end (say, Stratix III or Virtex 5) device. In each case, write down the number of logic cells or LUTs or slices needed by the circuit.

c) Simply modify the output equation to \( x = a - b \) and repeat the compilations, writing down the new amounts of hardware.

d) Repeat the procedure for \( x = a \times b \).

e) Finally, do it for \( x = a/b \).

f) Make a table with all the values obtained above. Compare the results against your predictions in part (a).