CHAPTER 6

PROGRAMMING FPGA APPLICATIONS IN VHDL

Nachiket Kapre
Department of Computer Science
California Institute of Technology
André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania

Modern field-programmable gate arrays (FPGAs) contain hundreds of thousands of lookup tables (LUTs), hundreds of embedded memories, and hundreds of multipliers connected through a programmable interconnect fabric. Obviously it is intractable to program the FPGA at the granularity of these individual elements. However, with modern synthesis and layout tools, it is possible to describe a design simply by writing logical expressions, a level higher than gates, and letting the tools do the rest. Register transfer level (RTL) design is a popular discipline for describing these logical expressions. It allows the designer to express the design by describing the logic between each pair of register stages. This allows her to carefully control register-to-register logic depth while freeing her from selecting the actual gates and their mapping to the FPGA. Very High-Speed Integrated Circuit Hardware Description Language (VHDL) is one popular programming language that supports RTL hardware descriptions.

VHDL enjoys widespread popularity among designers in the industry, along with its close cousin, Verilog. Indeed, almost all modern CAD tools that perform simulation, synthesis, and layout support both. Verilog differs from VHDL primarily in the syntax it uses (VHDL is derived from Ada; Verilog, from C), but both languages are IEEE standards and are periodically reviewed to reflect changing industry realities and expectations.

VHDL is a strongly typed, Ada-based programming language that includes special constructs and semantics for describing concurrency at the hardware level. These concurrency constructs are new for most programmers and can be a source of confusion for beginners. In the following sections, we provide a tutorial overview of how to express and compose synchronous designs in VHDL. Through examples, we highlight the control one can exercise in VHDL to direct proper synthesis of hardware. We first look at how VHDL can be used to describe a design structurally as a composition of subcircuits. We then show how to express hardware in RTL form. Next we illustrate how hardware can be generated parametrically in a programmable

manner. Finally we outline the basic tool and workflow for developing VHDL designs.

This chapter is by no means a complete discussion of all VHDL language features. For a more comprehensive treatment of language syntax and coding style the reader is referred to the work of Ashenden [1, 2] and the appropriate vendor manuals (e.g., Xilinx, Inc. [3]).

6.1 VHDL PROGRAMMING

Programming in VHDL is quite different from programming in C because of its concurrent semantics. However, it does have several similarities with object-oriented languages like C++ and Java (e.g., encapsulation and interfaces). These common principles should help beginners understand the basic structure of the language and help them relate to hardware-specific VHDL constructs. In this section, we describe a few simple design elements in VHDL to outline key language features and illustrate important programming concepts.

We first show how to program a 2-input multiplexer using a structural abstraction. We then program a 4-input multiplexer using RTL semantics. Next we illustrate the use of parametric hardware generation by creating a 16-bit wide, 4-input multiplexer using a 1-bit, 4-input multiplexer from the previous example. Then we combine structural and RTL styles in a finite-state machine (FSM) datapath example to show how to use them in the same design. This final example introduces the programming of FSMs in VHDL.

6.1.1 Structural Description

To describe a multiplexer structurally, we first decompose it into primitive gates derived from its Boolean equations. Each gate is *instantiated* individually and then connected to others. We can think of a structural decomposition as a textual representation of a schematic or as subroutines in a conventional programming language such as C. As with schematic capture, a structural decomposition permits code for a recurring design element to be shared. This means that we can design an element once and instantiate it as many times as required. Unlike schematic capture, a textual structural description can be modified and updated easily with a text editor. Moreover, a hierarchical decomposition allows the designer to manage the complexity of a large hardware design by breaking it up into individual, manageable pieces. Listing 6.1 and Figure 6.1 illustrate the following important concepts.

Listing 6.1 ■ A structural 2-input multiplexer.

```
1 library ieee;
2 use ieee.std_logic_1164. all;
3
4 — this is the entity declaration for the 2-input mux
5 — it is a list of ports into the module.
```

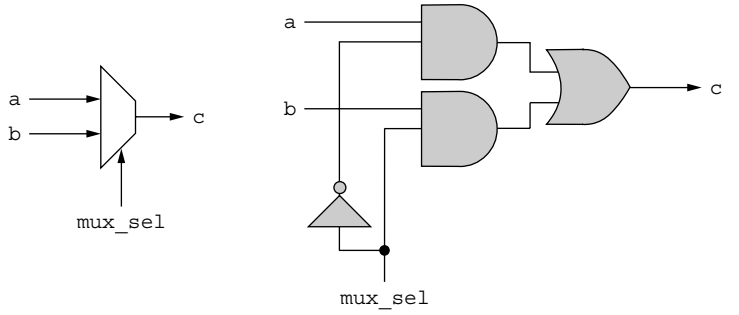

FIGURE 6.1 ■ A structural 2-input multiplexer.

```
 6 entity mux2 is
 7 port (
 8    a : in std_logic;
 9    b : in std_logic;
10    mux_sel : in std_logic;
11    c : out std_logic
12 );
13 end;
14
15 — this is where the structure of the multiplexer is defined
16 architecture struct of mux2 is
17
18 — all components that will be used in the structure
19 — need to be declared before use.
20 component notgate is
21 port (
22    a : in std_logic;
23    b : out std_logic
24 );
25 end component;
26
27 component andgate is
28 port (
29    a : in std_logic;
30    b : in std_logic;
31    c : out std_logic
32 );
33 end component;
34
35 component orgate is
36 port (
37    a : in std_logic;
38    b : in std_logic;
39    c : out std_logic
40 );
41 end component;
42
43 — internal signals/wires used to connect the components
44 — also need to be declared here.
```

```
45 signal muxsel_inverted_sig : std_logic;
46
47 signal sela_sig : std_logic;
48 signal selb_sig : std_logic;
49
50 — this signifies the start of the structural code
51 begin
52
53 — instantiation of the inverter
54 inverter_inst_0 : notgate
55 port map (
56    a => mux_sel,
57    b => muxsel_inverted_sig
58 );
59
60 — instantiation of the and gate
61 and_inst_a : andgate
62 port map (
63    a => a,
64    b => muxsel_inverted_sig,
65    c => sela_sig
66 );
67
68 — another instantiation of the and gate
69 and_inst_b : andgate
70 port map (
71    a => b,
72    b => mux_sel,
73    c => selb_sig
74 );
75
76 or_inst : orgate
77 port map (
78    a => sela_sig,
79    b => selb_sig,
80    c => c
81 );
82
83 end;
```

1. VHDL files typically start by including the IEEE library and certain important packages like `std_logic_1164` (Listing 6.1, lines 1–2) that permit the use of type `std_logic` and Boolean operations on it. Additional packages such as `std_logic_arith` and `std_logic_unsigned` are often included for supporting arithmetic operations.

2. The VHDL description of a hardware module requires an `entity` declaration (Listing 6.1, lines 6–13) that specifies the interface of the module with the outside world. It is an enumeration of the interface ports. The declaration also provides additional information about the ports such as their direction (in/out), data type, bit width, and endianness. An entity declaration

in VHDL is analogous to an interface definition in Java or a function header declaration in C.

3. Almost all VHDL signals and ports use the data type `std_logic` and `std_logic_vector`. These data types define how VHDL models electrical behavior of signals, which we discuss in the Multivalued logic subsection of Section 6.1.5. The vector `std_logic_vector` allows declaration of buses that are bundled together. We will see its use in a subsequent example.

4. While an entity specifies the interface of a hardware module, its internal structure and function are enclosed within the `architecture` definition (Listing 6.1, lines 16–83).

5. In a structural description of a module, the constituent submodules are declared, instantiated, and connected to each other. Each submodule needs to be first declared in the `component` declaration (Listing 6.1, lines 20–25). This is merely a copy of the entity declaration where only the submodule's interface is specified. Once the components are declared, they can then be instantiated (Listing 6.1, lines 54–58). Each instance of the component is unique, and a component can have multiple instances (Listing 6.1, lines 61 and 69). The instantiated components are connected to each other via internal signals by a process called *port mapping* (Listing 6.1, lines 55–58). Port mapping is performed on a signal-by-signal basis using the => symbol. It is analogous to assembling a set of integrated circuits (ICs) on a breadboard and wiring up the connections between the IC pins using jumper wires. Observe the similarity between the schematic representation of the multiplexer and the structural VHDL in the example.

6. Notice in the example that the component for the AND gate is reused for each AND gate in the design (Listing 6.1, lines 61–66 and 69–74). This is one of the benefits of a structural representation—it permits reuse of existing code for recurring design elements and helps reduce total code size.

7. The submodules used in Listing 6.1 are primitives supported in the vendor library. In a larger design that is a collection of several multiplexers, the different multiplexers can be declared, instantiated, and connected to each other as required. A design can have several such levels of structural hierarchy. Hierarchy is a fairly common technique for design composition.

6.1.2 RTL Description

The multiplexer's RTL description can be specified much more succinctly than its corresponding structural representation. In RTL, logic is organized as transformations on data bits between register stages. By selecting the number of pipeline stages wisely, the designer can create a high-performance, high-speed hardware implementation, and by carefully deciding the degree of resource sharing, the size of the mapped design can be controlled as well. RTL provides the designer with sufficient low-level control to allow her to create an implementation that meets her specifications.

For the VHDL description, we still need the logical equations that define the multiplexer, but these can now be represented directly as equations, from

which a synthesis tool *infers* the actual gates. The tool tries to choose the gates on the basis of user-specified design criteria such as high speed or small area.

Listing 6.2 shows how to write a 4-input multiplexer with registered outputs (Listing 6.1 simply showed a 2-input multiplexer without a register).

1. As before, we start with the package and entity declarations (Listing 6.2, lines 6–18).

2. The RTL description of the VHDL entity is enclosed in the **`architecture`** block (Listing 6.2, lines 20–52). The logic equations and registers that are part of the RTL description are written here. Earlier, we used the **`architecture`** block to write the structural port-mapping statements.

Listing 6.2 ■ RTL for a 4-input multiplexer.

```
1  — library and package includes
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  — entity declaration for the 4-input multiplexer
6  entity mux4 is
7  port (
8     clk : in std_logic;
9     reset : in std_logic;
10    a : in std_logic;
11    b : in std_logic;
12    c : in std_logic;
13    d : in std_logic;
14    — notice the use of the type vector.
15    mux_sel : in std_logic_vector (1 downto 0);
16    e : out std_logic
17 );
18 end;
19
20 — RTL description of the multiplexer is defined here
21 architecture rtl of mux4 is
22
23 — internal signals used in the multiplexer are
24 — declared here before use
25 signal e_c : std_logic;
26
27 — indicates start of the actual RTL code
28 begin
29
30 — concurrent signal assignment
31 — the multiplexer functionality is described
32 — at a level above gates
33 e_c <= a when mux_sel="00" else
34    b when mux_sel="01" else
35    c when mux_sel="10" else
36    d;
37
38 — sequential signal assignment
```

```
39 process (clk, reset)
40 begin
41
42 — action under reset
43 if (reset = '1') then
44    e <= '0';
45 — action under rising clock edge
46 elsif (clk' EVENT and clk='1') then
47    e <= e_c;
48 end if;
49
50 end process;
51
52 end;
```

3. In the structural example, we saw how signals were used as wires for connecting component ports. In VHDL, signals are also used for representing logic. A signal can be defined as a function of one or more signals. The assignment operation is represented by the symbol <=, which is analogous to the = operation in C; however, the manner in which signals are assigned values is quite different from C.

4. As before, a signal needs to be declared before the **begin** statement (Listing 6.2, line 25). Each signal is defined using a signal assignment statement that describes the logic that drives it. A signal assignment statement can be either concurrent or sequential.

5. A concurrent signal assignment is used to describe the logic equation for the multiplexer (Listing 6.2, lines 33–36). Concurrent statements are written inside the **begin**–**end** statements of the **architecture** block but outside any **process** blocks (Listing 6.2, lines 39–50). For simulation purposes, a concurrent statement can be thought of as being evaluated in parallel with other concurrent statements.

6. In the listing, a sequential assignment describes a register (Listing 6.2, lines 39–50). The behavior of the register under reset and a rising edge of the clock is defined between the **begin**–**end** statements of the **process** block, which is itself enclosed within the **begin**–**end** statements of the **architecture** block (Listing 6.2, lines 21–52). A **process** block is executed only when any signal on its *sensitivity list* (e.g., `clk` and `reset` signals in Listing 6.2, line 39) changes value.

As their name suggests, sequential assignment statements enclosed within a **process** block are executed sequentially. A process is suspended when it finishes evaluating all of the statements it can inside the block, and signals are assigned values only at that time. Additionally, during evaluation of a **process** block, a signal retains the same logical value it had when the process began execution. This can be a potential source of confusion for new programmers. In Listing 6.6, we show how to write combinational logic using sequential statements.

7. Notice the compactness with which the multiplexer was described in Listing 6.2 (52 lines of RTL code versus 83 for structural). This is one of the key benefits of RTL over purely structural descriptions.

6.1.3 Parametric Hardware Generation

VHDL allows the designer to generate hardware as a function of some changeable parameter. This is a useful technique for code reuse when we need several variants of an element in the same design (e.g., an 8-bit and 16-bit adder in the same design). Certain design parameters are often not known until late in the design cycle, and some can change as the design specification evolves to meet customer requirements. It might also be necessary to perform a parametric design space exploration based on certain variables before deciding on the final architecture. These issues can be resolved with VHDL **generic**s.

The generics are specified at the start of the entity declaration. In the simplest form, VHDL allows the designer to write signals as vectors of parametric width. More advanced uses of parametric hardware generation employ **generate** statements, and **generate** loops can be used to create multiple copies of a repeating logic block.

In Listing 6.3 and Figure 6.2, we illustrate the use of parametric hardware generation using a multibit 4-input multiplexer. The width of the multiplexer is defined by a generic DATA_WIDTH (Listing 6.3, lines 8–13), which sets the range of the vectors in the interface and is later used as the termination value in the **generate** loop (Listing 6.3, lines 47–61). DATA_WIDTH copies of the 4-input multiplexer described in Listing 6.2 are instantiated and connected to the interface ports appropriately (Listing 6.3, lines 54–59).

Listing 6.3 ■ Parametric generation of a multibit 4-input multiplexer.

```
1  — library and package includes
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  — entity declaration of the multiplexer array
6  entity mux4_array is
7  — definition of the generic for this entity
8  generic (
9     — here 16 is the default value
10    — it can be redefined during
11    — instantiation, or during synthesis
12    DATA_WIDTH : integer := 16
13 );
14 port (
15    clk : in std_logic;
16    reset : in std_logic;
17    — notice the use of generic for constraining the vector length
18    a : in std_logic_vector(DATA_WIDTH-1 downto 0);
19    b : in std_logic_vector(DATA_WIDTH-1 downto 0);
20    c : in std_logic_vector(DATA_WIDTH-1 downto 0);
21    d : in std_logic_vector(DATA_WIDTH-1 downto 0);
```

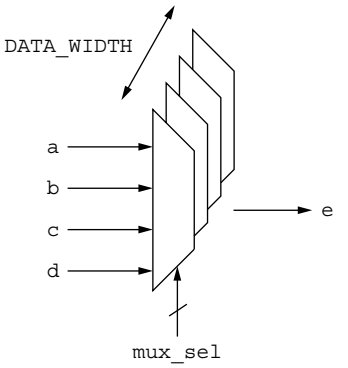

FIGURE 6.2 ■ Parametric generation of a multibit 4-input multiplexer.

```
22    mux_sel : in std_logic_vector(1 downto 0);
23    e : out std_logic_vector(DATA_WIDTH-1 downto 0)
24 );
25 end;
26
27 — the parametric code is enclosed within the architecture block
28 architecture parametric of mux4_array is
29
30 — like structural VHDL, the component being used needs to be declared here
31 component mux4 is
32 port (
33    clk : in std_logic;
34    reset : in std_logic;
35    a : in std_logic;
36    b : in std_logic;
37    c : in std_logic;
38    d : in std_logic;
39    mux_sel : in std_logic_vector(1 downto 0);
40    e : out std_logic
41 );
42 end component;
43
44 begin
45
46 — loop for generating a programmable number of mux4 instances
47 bitslices_gen : for i in 0 to DATA_WIDTH-1 generate
48    inst_mux : mux4
49    port map (
50       clk => clk,
51       reset => reset,
52       — notice the use of loop variable i for indexing
53       — into the array
54       a => a(i),
55       b => b(i),
56       c => c(i),
57       d => d(i),
58       mux_sel => mux_sel,
```

```
59        e => e(i)
60    );
61 end generate bitslices_gen;
62
63 end;
```

6.1.4 Finite-state Machine Datapath Example

In Listing 6.4, we design a time-shared datapath that computes Ax^2+Bx+C using only one multiplier and one adder. The design is naturally separated into state machine controller and datapath components. The controller and the datapath are designed using RTL and composed together structurally. The multiplier, the adder, and the associated multiplexers and registers are part of the datapath, whereas the control signals for the datapath multiplexers (Listing 6.4, lines 80–82) and registers (Listing 6.4, lines 84–86) are generated by the controller. Figure 6.3 shows the structural decomposition and the associated VHDL code. We can see that the control signals are connected from the controller to the datapath in the structural VHDL representation.

Listing 6.4 ■ A structural representation of the FSM datapath design.

```
1  — library and package includes
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.std_logic_unsigned.all;
5
6  — entity declaration of the state-machine controller
7  entity fsm_datapath is
8  port (
9     — system signals
10    clk : in std_logic;
11    reset : in std_logic;
12
13    — input interface
14    start : in std_logic;
15    A : in std_logic_vector(3 downto 0);
16    B : in std_logic_vector(3 downto 0);
17    C : in std_logic_vector(3 downto 0);
18    x : in std_logic_vector(3 downto 0);
19
20    — output interface
21    output_valid : out std_logic;
22    result : out std_logic_vector(12 downto 0)
23 );
24 end;
25
26 architecture struct of fsm_datapath is
27
28 component fsm is
29  port (
30    — system signals
```

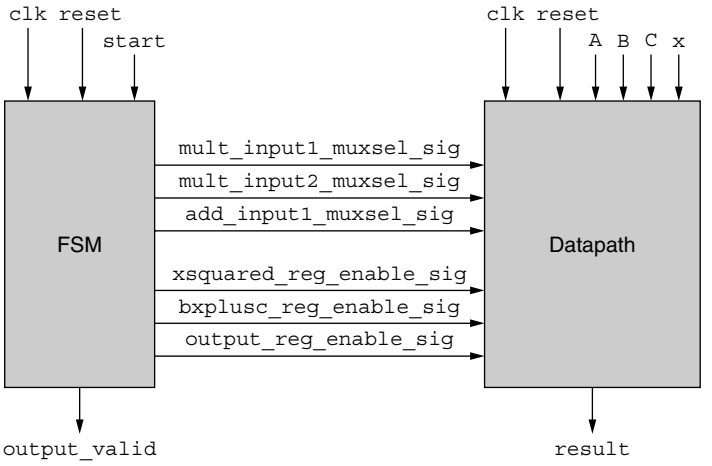

FIGURE 6.3 ■ A structural representation of the FSM datapath design.

```
31     clk : in std_logic;
32     reset : in std_logic;
33
34     — start the computation
35     start : in std_logic;
36
37     — datapath multiplexer select
38     mult_input1_muxsel : out std_logic_vector(1 downto 0);
39     mult_input2_muxsel : out std_logic;
40     add_input1_muxsel : out std_logic;
41
42     — register enables
43     xsquared_reg_enable : out std_logic;
44     bxplusc_reg_enable : out std_logic;
45     output_reg_enable : out std_logic;
46
47     — indicate output is valid
48     output_valid : out std_logic
49 );
50 end component;
51
52 component datapath is
53 port (
54     — system signals
55     clk : in std_logic;
56     reset : in std_logic;
57
58     — input operands
59     A : in std_logic_vector(3 downto 0);
60     B : in std_logic_vector(3 downto 0);
61     C : in std_logic_vector(3 downto 0);
62     x : in std_logic_vector(3 downto 0);
63
```

```
64     — datapath multiplexer select
65     mult_input1_muxsel : in std_logic_vector(1 downto 0);
66     mult_input2_muxsel : in std_logic;
67     add_input1_muxsel : in std_logic;
68
69     — register enables
70     xsquared_reg_enable : in std_logic;
71     bxplusc_reg_enable : in std_logic;
72     output_reg_enable : in std_logic;
73
74     — output data
75     result : out std_logic_vector(12 downto 0)
76  );
77  end component;
78
79  — internal wires for connecting the components
80  signal mult_input1_muxsel_sig : std_logic_vector(1 downto 0);
81  signal mult_input2_muxsel_sig : std_logic;
82  signal add_input1_muxsel_sig : std_logic;
83
84  signal xsquared_reg_enable_sig : std_logic;
85  signal bxplusc_reg_enable_sig : std_logic;
86  signal output_reg_enable_sig : std_logic;
87
88  — start component instantion and wiring
89  begin
90
91  datapath_inst : datapath
92  port map (
93
94     — system signals
95     clk => clk,
96     reset => reset,
97
98     — input operands
99     A => A,
100    B => B,
101    C => C,
102    x => x,
103
104    — datapath multiplexer select
105    mult_input1_muxsel => mult_input1_muxsel_sig,
106    mult_input2_muxsel => mult_input2_muxsel_sig,
107    add_input1_muxsel => add_input1_muxsel_sig,
108
109    — register enables
110    xsquared_reg_enable => xsquared_reg_enable_sig,
111    bxplusc_reg_enable => bxplusc_reg_enable_sig,
112    output_reg_enable => output_reg_enable_sig,
113
114    — output data
115    result => result
116 );
117
```

```
118 fsm_inst : fsm
119 port map (
120    — system signals
121    clk => clk,
122    reset => reset,
123
124    — start the computation
125    start => start,
126
127    — datapath multiplexer select
128    mult_input1_muxsel => mult_input1_muxsel_sig,
129    mult_input2_muxsel => mult_input2_muxsel_sig,
130    add_input1_muxsel => add_input1_muxsel_sig,
131
132    — register enables
133    xsquared_reg_enable => xsquared_reg_enable_sig,
134    bxplusc_reg_enable => bxplusc_reg_enable_sig,
135    output_reg_enable => output_reg_enable_sig,
136
137    — indicate output is valid
138    output_valid => output_valid
139 );
140
141 end;
```

We use the RTL form to describe the datapath, and we use a combination of concurrent and sequential statements for this purpose. The structure of the datapath is shown in Listing 6.5 and Figure 6.4.

Listing 6.5 ■ A time-shared datapath for computing $Ax^2 + Bx + C$.

```
1  — include the unsigned package to support arithmetic operations.
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.std_logic_unsigned.all;
5
6  — describes the interface to the datapath,
7  — with its operands and control signals listed.
8  entity datapath is
9  port (
10    — system signals
11    clk : in std_logic;
12    reset : in std_logic;
13
14    — input operands
15    A : in std_logic_vector(3 downto 0);
16    B : in std_logic_vector(3 downto 0);
17    C : in std_logic_vector(3 downto 0);
18    x : in std_logic_vector(3 downto 0);
19
```

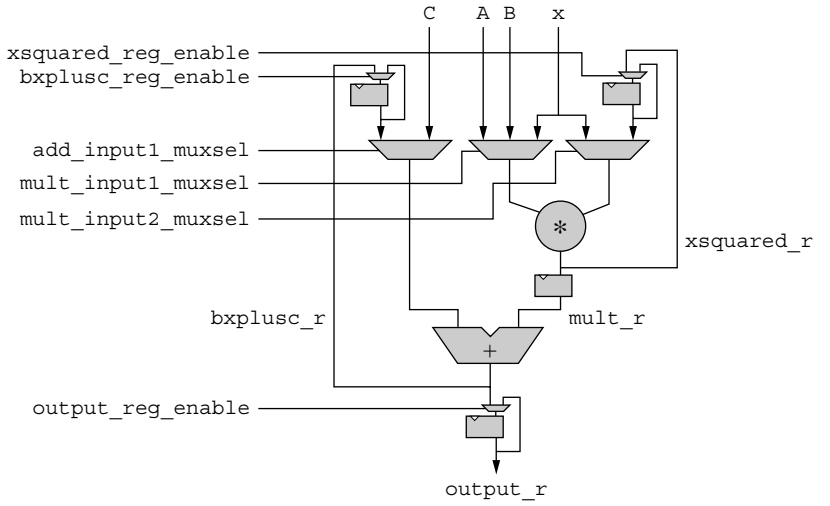

FIGURE 6.4 ■ A time-shared datapath for computing $Ax^2 + Bx + C$.

```
20     — datapath multiplexer select
21     mult_input1_muxsel : in std_logic_vector(1 downto 0);
22     mult_input2_muxsel : in std_logic;
23     add_input1_muxsel : in std_logic;
24
25     — register enables
26     xsquared_reg_enable : in std_logic;
27     bxplusc_reg_enable : in std_logic;
28     output_reg_enable : in std_logic;
29
30     — output data
31     result : out std_logic_vector(12 downto 0)
32 );
33 end;
34
35  architecture rtl of datapath is
36
37  — notice the different bitwidths on each signal
38  — these precisions have been carefully selected
39  — based on the multiply/add operations and input
40  — bitwidths
41  signal mux_0_c : std_logic_vector(3 downto 0);
42  signal mux_1_c : std_logic_vector(7 downto 0);
43  — mux_1_c needs 8 bits of precision due to
44  — x–squared at the input
45  signal mux_2_c : std_logic_vector(8 downto 0);
46  — mux_2_c needs 9 bits of precision due to
47  — precision of Bx+C
48
49  signal mult_c : std_logic_vector(11 downto 0);
50  — product of 8-bit and 4-bit inputs is 12-bit
51
```

```
52 signal add_c : std_logic_vector(12 downto 0);
53 — sum of 12-bit and 9-bit inputs is 13-bits with overflow
54
55 signal mult_r : std_logic_vector(11 downto 0);
56 signal output_r : std_logic_vector(12 downto 0);
57 signal bxplusc_r : std_logic_vector(8 downto 0);
58 signal xsquared_r : std_logic_vector(7 downto 0);
59
60
61 begin
62
63 — concurrent statements to describe the multiplexers
64 mux_0_c <= A when mult_input1_muxsel = "00" else
65            B when mult_input1_muxsel = "01" else
66            x;
67
68 mux_1_c <= "0000"&x when mult_input2_muxsel = '0' else
69            xsquared_r;
70
71 mux_2_c <= "00000"&C when add_input1_muxsel = '0' else
72            bxplusc_r;
73
74 — multiplier
75 mult_c <= mux_0_c * mux_1_c;
76
77 — adder
78 — the extra 0s at the MSB of the inputs are
79 — to capture overflow bit in the result
80 add_c <= ("0000"&mux_2_c) + ('0'&mult_r);
81
82 — define all registers
83 all_registers : process(clk, reset)
84 begin
85
86 if (reset= '1') then
87
88    mult_r <= (others=>'0');
89    xsquared_r <= (others=>'0');
90    bxplusc_r <= (others=>'0');
91    output_r <= (others=>'0');
92
93 elsif (clk' EVENT and clk='1') then
94
95   — infer simple register
96   mult_r <= mult_c;
97
98   — notice that we are not specifying
99   — the else condition. the synthesis tool will
100  — infer a latch for this case. if enable is
101  — low, previous value will be retained.
102  if (xsquared_reg_enable='1') then
103     xsquared_r <= mult_c(7 downto 0);
104  end if;
105
```

```
106    if (bxplusc_reg_enable='1') then
107       bxplusc_r <= add_c(8 downto 0);
108    end if;
109
110    if (output_reg_enable='1') then
111       output_r <= add_c;
112    end if;
113
114 end if;
115 end process;
116
117 — drive the output with a simple wire from the register
118 result <= output_r;
119
120 end;
```

Included in this datapath design is the special package `std_logic_unsigned` (Listing 6.5, line 4), which allows us to express arithmetic operations using high-level symbols (`+` and `*`) on signals of type `std_logic_vector`. These functions are defined in the package. The package also helps us infer the right kind of arithmetic units (e.g., signed or unsigned). VHDL supports the signed data type for arithmetic operations.

Notice that we must carefully specify the precision required for all internal signals (Listing 6.5, lines 37–58). We must also pad extra 0s when the input signal precision is smaller than that of the operator (Listing 6.5, lines 68–69). The concatenation operator `&` in VHDL further allows us to combine the right mix of signals to enter the datapath as required by the design. This low-level control makes VHDL suitable for designers seeking to customize their designs to the problem.

We represent the multiplexers, multipliers, and adders using concurrent statements (Listing 6.5, lines 63–80), which are evaluated in parallel and inferred as combinational logic blocks. Note that all three multiplexers evaluate their inputs simultaneously. Concurrent statements allow the designer to capture this hardware-level concurrency in VHDL. Also note, however, that there is a dataflow dependency between the multiplexers and the multiplier (as well as the multiplexer and the adder). These dependencies are converted into wires that connect the appropriate logic blocks together, but each logic block continues to evaluate its inputs in parallel. The dataflow dependency only means that signal changes are propagated to the downstream multiplier input after a suitable delay for the multiplexer evaluation (see Delta delay subsection of Section 6.1.5 for more information on this delay).

We express the registers in the design using sequential statements inside the **`process`** block (Listing 6.5, lines 83–115). Most registers have a conditional signal assignment (Listing 6.5, lines 102–104). Notice the absence of an **`else`** statement or a default value on the rising clock edge. This implies that the signal retains its previous value if the condition for assignment is not

satisfied. VHDL automatically infers feedback from the output to the multiplexer at the register input. If the **else** is present or if a default value is specified, no feedback will be inferred. This can be seen in Listing 6.6 (signals in the next-state decoder process have default values, avoiding inference of feedback paths).

To design the state machine controller, we first create a time sequence of operations that must be performed to obtain the final result. This gives us a cycle-by-cycle schedule for how the datapath elements are shared between the different operations. Each of these cycles is represented by a state, which is then decoded into multiplexer select and register enable signals for the datapath. The VHDL for this state machine is written in an RTL form specialized for state machines. It is shown in Listing 6.6 and illustrated in Figure 6.5.

Listing 6.6 ■ A state machine for generating control signals for the time-shared datapath.

```
1  — library and package includes
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  — entity declaration of the state-machine controller
6  entity fsm is
7  port (
8     — system signals
9     clk : in std_logic;
10    reset : in std_logic;
11
12    — start the computation
13    start : in std_logic;
14
15    — datapath multiplexer select
16    mult_input1_muxsel : out std_logic_vector(1 downto 0);
17    mult_input2_muxsel : out std_logic;
18    add_input1_muxsel : out std_logic;
19
20    — register enables
21    xsquared_reg_enable : out std_logic;
22    bxplusc_reg_enable : out std_logic;
23    output_reg_enable : out std_logic;
24
25    — indicate output is valid
26    output_valid : out std_logic
27 );
28 end;
29
30 — state-machine code is enclosed is defined inside this architecture block
31 architecture behav of fsm is
32
33 — define an enumerated type for state
34 type state_type is (IDLE, COMPUTE_BX, COMPUTE_BXPLUSC_AND_XSQR,
35                     COMPUTE_AXSQR, COMPUTE_ASQRPLUSBXPLUSC, ASSERT_OUTPUT);
```

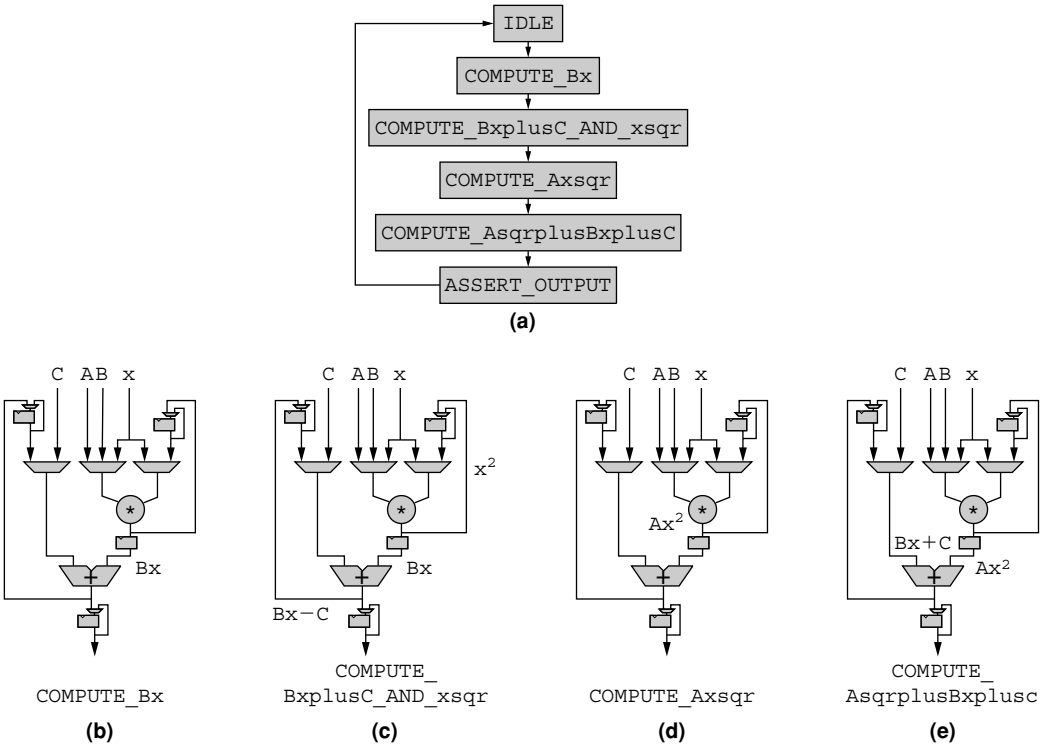

FIGURE 6.5 ■ A state machine for generating control signals for the time-shared datapath. Labels on wires show dataflow steps in calculation.

```
36  signal state_c : state_type;
37  signal state_r : state_type;
38
39  — internal signals
40  signal mult_input1_muxsel_c : std_logic_vector(1 downto 0);
41  signal mult_input2_muxsel_c : std_logic;
42  signal add_input1_muxsel_c : std_logic;
43  signal xsquared_reg_enable_c : std_logic;
44  signal bxplusc_reg_enable_c : std_logic;
45  signal output_reg_enable_c : std_logic;
46  signal output_valid_c : std_logic;
47
48  — start the signal assignments
49  begin
50
51  — logic to compute the next state of the state machine
52  — also generate the control signals [only combinational, right now]
53  next_state_decoder : process(state_r, start)
54  begin
55
56  — given initial values for all signals
57  mult_input1_muxsel_c <= "00";
58  mult_input2_muxsel_c <= '0';
```

```
59  add_input1_muxsel_c <= '0';
60  xsquared_reg_enable_c <= '0';
61  bxplusc_reg_enable_c <= '0';
62  output_reg_enable_c <= '0';
63  output_valid_c <= '0';
64  state_c <= IDLE;
65
66  — specify state transistions
67  — update state variable
68  — update the control signals
69  case state_r is
70    when IDLE =>
71
72       — conditional state transition
73       if (start='1') then
74
75          state_c <= COMPUTE_BX;
76
77          mult_input1_muxsel_c <= "01"; — select B
78          mult_input2_muxsel_c <= '0'; — select x
79
80    end if;
81
82    when COMPUTE_BX =>
83
84    — unconditional state transition
85    state_c <= COMPUTE_BXPLUSC_AND_XSQR;
86
87    mult_input1_muxsel_c <= "10"; — select x
88    mult_input2_muxsel_c <= '0'; — select x
89    xsquared_reg_enable_c <= '1'; — save x*x
90    bxplusc_reg_enable_c <= '1'; — save Bx+C
91    add_input1_muxsel_c <= '1'; — select C
92
93  when COMPUTE_BXPLUSC_AND_XSQR =>
94
95    state_c <= COMPUTE_AXSQR;
96
97    mult_input1_muxsel_c <= "00"; — select A
98    mult_input2_muxsel_c <= '1'; — select xsqr
99
100 when COMPUTE_AXSQR =>
101
102   state_c <= COMPUTE_ASQRPLUSBXPLUSC;
103
104   add_input1_muxsel_c <= '1'; — select Bx+C
105   output_reg_enable_c <= '1';
106
107 when COMPUTE_ASQRPLUSBXPLUSC =>
108
109   state_c  <= ASSERT_OUTPUT;
110   output_valid_c <= '1';
111
112 when ASSERT_OUTPUT =>
113
```

```
114    state_c <= IDLE;
115
116  end case;
117
118  end process;
119
120  — describe the registers that hold the state bits
121  — the actual bits will be inferred by the
122  — synthesis tool from the symbolic states
123  state_register : process(clk, reset)
124  begin
125
126  if (reset = '1') then
127    state_r <= IDLE;
128  elsif (clk' EVENT and clk='1') then
129    state_r <= state_c;
130  end if;
131
132  end process;
133
134  — register the control signals generated during state transitions
135  output_logic : process(clk, reset)
136  begin
137
138  if (reset = '1') then
139
140     mult_input1_muxsel <= "00";
141     mult_input2_muxsel <= '0';
142     add_input1_muxsel <= '0';
143     xsquared_reg_enable <= '0';
144     bxplusc_reg_enable <= '0';
145     output_reg_enable <= '0';
146     output_valid <= '0';
147
148  elsif (clk EVENT and clk='1') then
149
150    mult_input1_muxsel <= mult_input1_muxsel_c;
151    mult_input2_muxsel <= mult_input2_muxsel_c;
152    add_input1_muxsel <= add_input1_muxsel_c;
153    xsquared_reg_enable <= xsquared_reg_enable_c;
154    bxplusc_reg_enable <= bxplusc_reg_enable_c;
155    output_reg_enable <= output_reg_enable_c;
156    output_valid <= output_valid_c;
157
158  end if;
159
160  end process;
161
162  end;
```

By encoding the state of the controller with an enumerated data type (Listing 6.6, lines 34–36), we can defer the actual encoding of the state bits until the synthesis stage. The synthesis tool then assigns a bit encoding to optimize logic. It is easier to verify the operation of the state machine using

symbolic states. It is also easier to update and modify symbolic state machine code.

In the next-state decoder (Listing 6.6, lines 53–118), we enumerate all possible states of the state machine and define state transitions from each of them. These transitions are expressed as conditions under which the state changes.

We use the **process** block for describing purely combinational logic in the next-state decoder of the state machine (Listing 6.6, lines 53–118). Previously, we used **process** for describing only registers (Listing 6.2, lines 39–50). This shows how we can write combinational logic here as well. In this listing, notice that the same signal is assigned values multiple times in the **process** block (signal `mult_input1_muxsel_c` in Listing 6.6, lines 57, 77, 87, and 97). As the statements are evaluated sequentially, the last signal assignment statement to be evaluated is considered valid, superseding all previous assignments. During execution of sequential statements in a process, for purposes of determining new signal values all signals are considered to have the same value they had at the start of the process. Signals that are assigned values inside the process will acquire those values only when process execution is complete—that is, **process** suspends. It is in this aspect that the VHDL sequential semantics are different from those of a conventional programming language (e.g., C). Figure 6.6 shows similar code written in C and VHDL to illustrate how the different execution semantics lead to different answers.

In Listing 6.6, all signals are assigned a value at the beginning of the process. By design, only one **when** subblock of the **case** statement will be evaluated, which means that only those signals that have assignments inside the valid **when** subblock will get new values (Listing 6.6, line 77, 87, or 97 will execute; line 57 will execute in all cases). According to the VHDL sequential signal assignment rule, these new assignments will hold when the process suspends. Other signals will simply carry the default values they were assigned at the start. This avoids the inference of feedback that we saw earlier (refer to Listing 6.2).

```
1 process(clk)
2 begin
3
4       if(clk 'EVENT'and clk='1') then
5               counter <= counter + 1;
6               if(counter=10)
7                       counter <= 0;
8               end if;
9       end if;
10
11 end process;
12
13 — if counter=9 at start of process,
14 — when process suspends, counter=10.
```

(a)

```
1 int updatecounter(int counter){
2        counter++;
3
4       if(counter==10)
5          counter = 0;
6
7        return counter;
8 }
9
10 // updatecounter(9) returns 0
```

(b)

FIGURE 6.6 ■ Comparison of sequential VHDL (a) and C (b) assignment semantics.

6.1.5 Advanced Topics

Delta delay

VHDL uses an event-driven simulation model. A signal is evaluated only when an event—that is, a signal transition associated with the input signals—has occurred. Once a statement is evaluated, its associated signal needs to be assigned the newly generated value. However, this is not done right away so as to keep the evaluations of other statements from using this new value immediately, potentially leading to inconsistent results.

Remember that in VHDL all concurrent statements are evaluated in parallel. Hence, to keep the simulation consistent VHDL uses delta delay, in which the newly generated value is scheduled as an event at the following delta. (A delta is simply a logical delay used in the simulator and not a physical delay of the circuit.) The simulator will generate as many deltas as required depending on the logical depth of the circuit and its input transitions. Once all events for a given delta are exhausted, the simulator proceeds to the earliest delta at which the next event exists. Physical time in the simulator is advanced only when no more events are left to be processed at the last delta at the current physical time. Sometimes the simulator is unable to advance its physical time because of asynchronous, combinational feedback loops that continue generating new events at incremental deltas. Such loops should be avoided when programming VHDL, and modern synchronous simulation and synthesis tools usually warn the designer if such a loop is detected.

Multivalued logic

Another electrical behavior is modeled in VHDL using the multivalued logic type `std_logic`. It allows a signal to have different kinds of electrical states, apart from a Boolean 0 or 1, which are required for modeling tristate drivers, multiple simultaneous drivers (usually a design error), uninitialized signals, and weak drivers.

6.2 HARDWARE COMPILATION FLOW

To fully understand how VHDL fits into the design process, we expand the FPGA compilation process shown in Figure I.2. Our flow is shown in Figure 6.7.

1. The hardware designer begins the design-engineering process with a problem specification—that is, a functional description of the problem along with additional performance and area constraints that the implementation must meet.

2. Based on this specification and the inherent problem structure, the designer identifies an appropriate system architecture to use for the implementation. We saw different kinds of system architectures in Chapter 5.

3. The designer writes VHDL code to describe this design using structural and RTL styles that we saw earlier in this chapter.

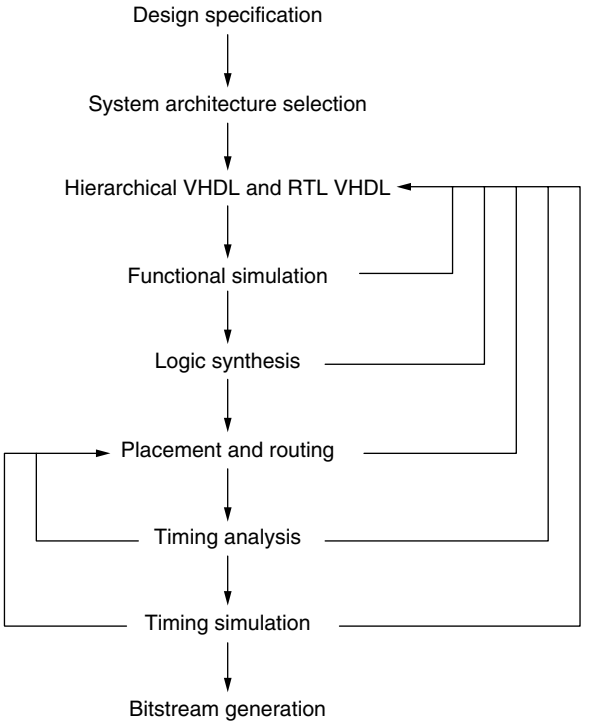

FIGURE 6.7 ■ FPGA compilation flow.

4. Once the VHDL is written, the designer needs to first check if her VHDL meets functional specifications, using a suitable testbench that can be written in VHDL itself. The testbench and the design are run in a logic-level simulator. The testbench generates appropriate test vectors for the design and verifies the result. This is typically an iterative process, and the designer continues to refine the VHDL design until the functional specification is met.

5. After verifying correctness, the designer then proceeds to the FPGA back-end phase, a multistage (and iterative) process. It starts with synthesis, where the synthesis tool converts the VHDL description of the design (excluding the testbench) into a logic-level FPGA netlist. This netlist is generated by first inferring hardware from VHDL code and then optimizing it through several state-of-the-art algorithms—for example, logic minimization, retiming (Chapter 18), covering (Chapter 13), and sharing to meet timing and area constraints. Constraints can be specified as a separate input to the tool by the user.

6. The designer uses backend tools to perform placement (Chapter 14) and routing (Chapter 17) on the synthesized logic elements to map them to an actual physical device (logic elements are assigned physical LUTs while the

wires between them are mapped to the interconnect fabric). This is typically the most time-consuming step of the backend process. The designer can help direct these tools using additional constraints (see Section 6.2.1) either to improve the quality of the final mapped design or to reduce the compilation time needed.

7. Once the design is placed and routed, the designer can perform static timing analysis to ensure that the timing constraints are met. FPGA tools can also write out a post place and route timing annotated VHDL netlist for a timing simulation that models logic and interconnect delays accurately. Specific timing requirements not covered in the simple static timing analysis can then be simulated and checked.

8. If the designer is satisfied with the performance of her implemented hardware, the tools generate a programming file for the FPGA device (Chapter 19).

6.2.1 Constraints

Constraints are an indispensable tool directive that a designer can use to help her designs meet required specifications. They can be used to direct the synthesis tools in optimizing the design for either high-speed operation or low-area implementation (these are usually conflicting goals). For example, the designer can specify a frequency target that Synplify Pro (a synthesis tool) must meet using the following timing constraint.

```
set option –frequency 300.000
```

This sets the target frequency for the compilation to be 300 MHz. Similarly, designers can provide timing constraints for the placement and routing phases as well.

```
TIMESPEC "clock signal name"=3.3ns;
```

More important, a designer can give physical floorplanning constraints to direct the placement and routing algorithms to use a specified region on the chip.

```
INST "*" AREA GROUP = "dummy name";
AREA GROUP "dummy_name" RANGE = SLICE X0Y0:SLICE X100Y100;
```

Here we create a group `dummy_name` containing all hardware elements in the design using wildcards (`*`). Then we specify a rectangular box from 0,0 to 100,100 on the FPGA. The units are measured in `SLICE`s; a `SLICE` is a cluster of a few, usually four, Xilinx FPGA LUTs. The proper selection of these constraint values is typically based on intuition and can be refined with designer experience. Placement constraints, such as the one in the previous code snippet, are vendor and device specific, but each vendor typically has analogous constraints for each device.

6.3 LIMITATIONS OF VHDL

Although VHDL currently enjoys a healthy market share, there are several limitations and drawbacks in the language:

- VHDL syntax is verbose, extremely cumbersome, and requires several lines of code to describe even simple logic elements (e.g., a register typically requires four to ten lines of code).
- Hardware needs to be described at a very low level of abstraction (i.e., RTL). The programmer is responsible for specifying the logic that goes between each register stage, which can become a significant programming challenge for large irregular designs with thousands of registers and unique logic between register stages.
- As technology and FPGA architectures evolve, the optimal amount of pipelining required to meet the desired cycle time changes. Because RTL is written for a specific number of registers in the logic path, it needs to be rewritten when the number of register stages changes. In other words, the amount of logic between register stages must be modified accordingly.
- Low-level descriptions also make it hard for synthesis tools to optimize and schedule logic. Programmer bias disallows optimizations that might have otherwise been possible in a more flexible description.
- Hardware described in VHDL suffers from the additional drawback of significantly long verification times. It is known that equivalent simulation-specific, cycle-accurate models written in C, C++, Java, or other higher-level language can be simulated 10 to 100 times faster than in VHDL. Verification is a significant portion of the design cycle, and there is demand to contain the time spent on it.

In subsequent chapters, we will see other high-level languages that address many of these limitations (e.g., Chapters 7, 9, and 10). In many cases, however, these languages use VHDL as an intermediate target in their mapping flow.

References

[1] P. Ashenden. *The Designer's Guide to VHDL*, 2nd ed., Morgan Kaufmann, 2002.
[2] P. Ashenden. *The Student's Guide to VHDL*, Morgan Kaufmann, 1998.
[3] *Development System Reference Guide*, Xilinx, Inc.