Abel™ −HDL vs. IEEE−1076 VHDL

Abstract

Currently there exist several popular Hardware Description Languages (HDLs) that allow designers to describe the function of complex logic circuits textually, as opposed to schematically. One of the most widely used of these languages is Data I/O's Abel−HDL. Abel−HDL, as a language, can be used to describe the behavior of logic circuits that can be fitted to a wide variety of PALs, PLDs, PROMs, and FPGAs from a variety of programmable logic IC manufacturers. IEEE−1076 VHDL (VHSIC Hardware Description Language) has recently been gaining widespread support. VHDL is an open, portable language defined and standardized by the IEEE that can be used to describe the behavior of an entire system from the highest levels of functionality all the way down to the logic-gate level. A majority of CAE vendors, programmable IC manufacturers, and third-party software vendors already have or are planning tools that support VHDL logic synthesis, logic modeling, and/or VHDL simulation.

The purpose of this application note is to compare and contrast the complexity and basic features of Abel−HDL with those of IEEE−1076 VHDL. Both of these languages are very robust in their support of different types of constructs that can be used to describe the same functionality at different levels of abstraction. It is beyond the scope of this document to exhaustively describe these possibilities or to present a complete tutorial for writing code in either language because of the great variety of constructs and syntax available with which to describe the functionality of a given circuit. Rather, a simple example design that contains a mixture of synchronous and asynchronous logic circuits will be shown. Sample code is written in both Abel−HDL and VHDL that describes the example's functionality and synthesizes to create functionally identical hardware. The code written here represents a typical level of abstraction that balances readability with compactness. With experience, designers can develop their own preferences for style. For instance, state machines can be described in a number of ways: state tables, IF-THEN-ELSE statements, CASE-WHEN statements, or explicitly using a combination of Register-Transfer-Level (RTL) code (individually describe each gate/register as a component with its inputs and outputs) and/or Boolean equations.

Example Description

The following example is a circuit that creates a 50% duty cycle clock with programmable frequency. *Figure 1* shows the block diagram of this Programmable Clock Generator. The output of the circuit is CLK_OUT, whose period is equal to clock(ns)*incnt*2. To program the device, LD_CNT is used to latch the value present at the INCNT(3:0) inputs into the 4-Bit Input Register. The output of this register is ENDCNT(3:0). The clock input is used to clock the 4-bit Up/Down Counter, whose output is COUNT(3:0). The 4-bit Comparator is an asynchronous comparator that compares the values of COUNT and ENDCNT. Its outputs are endeqcnt (ENDCNT = COUNT), endltcnt (ENDCNT < COUNT), endeq0 (ENDCNT = 0), and cnteq0 (COUNT = 0). Note that it is possible for ENDCNT to be less than COUNT if a new value for ENDCNT is loaded into the input registers that is less than the current value of COUNT. The cnt_state State Machine controls the CLK_OUT signal and is clocked and enabled by the clock and en inputs, respectively.

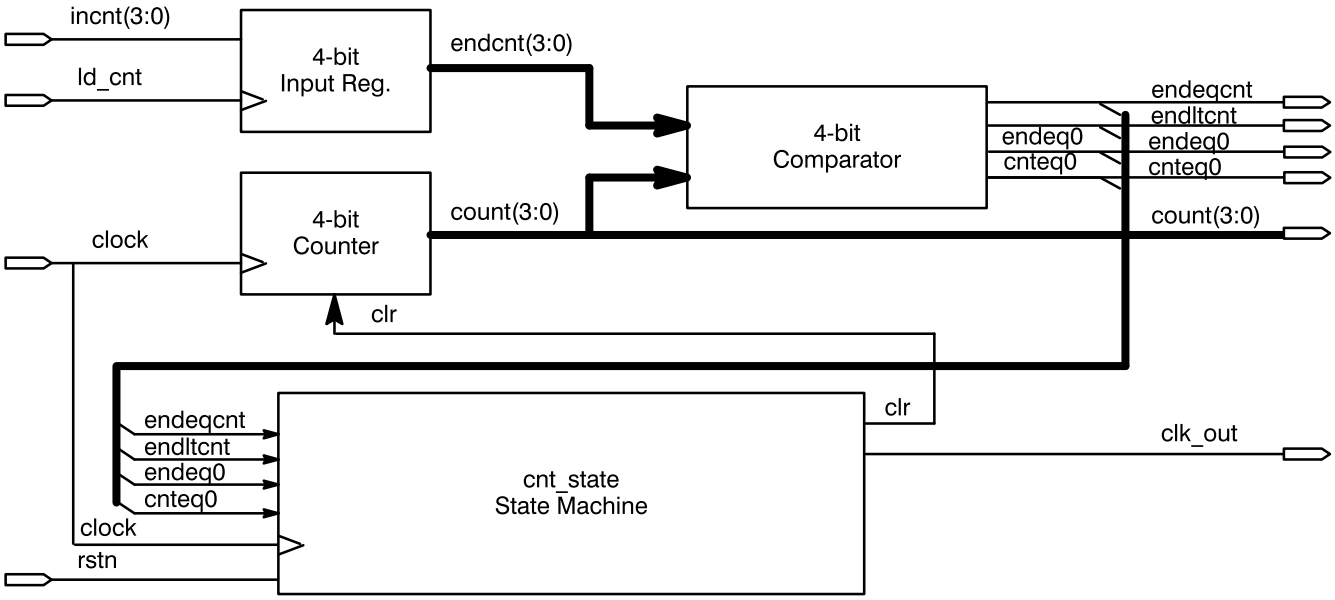

Figure 1. Block Diagram

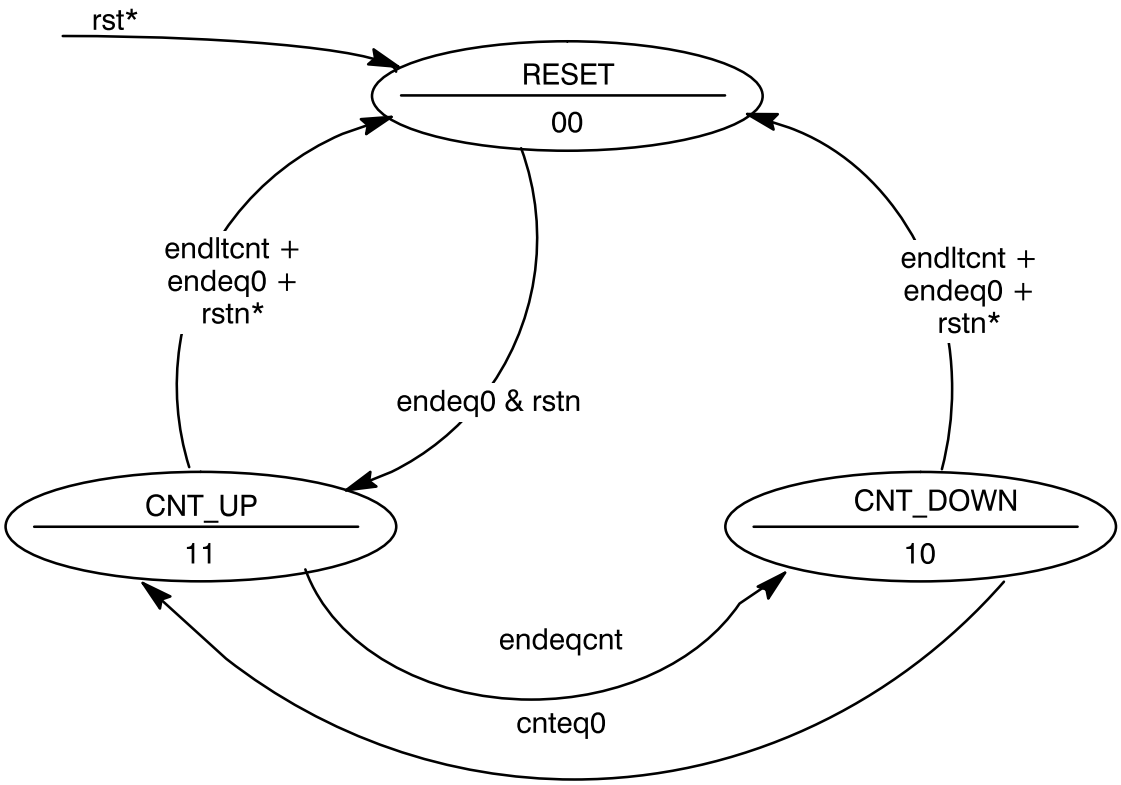

Figure 2. State Diagram

Figure 2 shows the state diagram for cnt_state. The state machine consists of three states: *reset*, *cnt_up*, and *cnt_down*. Two state bits are used to describe this Mealy-type state machine. The state machine powers-up to the reset state. It also synchronously enters the *reset* state from any state if RSTN=0. Once RSTN=1, the state machine will count up until COUNT equals ENDCNT, at which point it will begin to count down until COUNT equals 0 and then repeat the cycle. If at any time ENDCNT = 0, cnt_state will return to *reset*. Also, if ENDCNT is ever less than COUNT, thus signifying an invalid condition, cnt_state will go to *reset*.

The PLD targeted is the CY7C335. It was chosen because it can be used to illustrate a variety of features contained in some PLDs such as input registers, multiple clocks, buried registers, and synchronous reset/preset. However, any PLD with sufficient resources could be targeted. This is one of the main advantages of using a HDL. High-level languages, by design, allow a designer to write generic code that can be targeted to different devices/ architectures with little or no modification. Going one step further, VHDL allows simulation and debugging of the logic from the source code. This can greatly reduce the overall design cycle time by allowing functional verification of the logic prior to targeting a specific physical device. Once the logic has been verified, the designer can then compile and fit the same design into a variety of devices. From here, the designer can decide which implementation best suits his requirements.

Abel−HDL vs. VHDL

In general, the constructs used to describe logic function in both ABEL−HDL an IEEE−1076 VHDL are very similar. Each can accept Boolean equations, truth tables, state descriptions (IF-THEN and CASE-WHEN), and signal assignments that are quite similar in appearance. However, differences do exist in syntax and in the overhead sections. These are the declarative sections which define hierarchical organization, design libraries, etc. As we shall see, some of these statements found in the VHDL code have no direct counterpart in Abel−HDL. This is because Abel−HDL was created with a single purpose in mind, logic synthesis for PALs and PLDs. Therefore, with Abel−HDL, assumptions can be made by the compiler which can simplify the overhead syntax needed. Because VHDL is a one-language-fits-all standard which applies to synthesis, modeling, and system definition at all levels, some syntax overhead is necessary to fully describe a design in the proper context. For example, the use of standard and user-definable packages and libraries allows many designs to share commonly used definitions, components, macros, functions, etc. Fortunately for the logic designer, these statements are used in most designs so that familiarity with them comes quickly and easily. This commonality also allows ample “cutting-and-pasting” from design to design.

The following sections compare and contrast the source code files for Abel−HDL and VHDL on a logical section-by-section basis. Both of these files, when compiled, create functionally identical logic. Copies of the source code files in their entirety can be found in the Appendices.

Design and I/O Declarations

The basic structure of both Abel−HDL and VHDL source files allow for one or more design units to be defined within it. Each design unit is a complete logic description. Multiple design units may be combined hierarchically in a single top-level source file which binds them together. In the example, we have used a single design unit for simplicity. The first section of code chosen for comparison contains the design declaration and device I/O declarations. In this example, the pin numbers have been fixed. This optional in either language. The declaration of the target device is also optional in the source code itself. The targeted device need not be declared until it’s time to compile and fit the logic.

Figure 3 contains the Abel−HDL code that declares the module name (line 01), the device (line 03), and the input and output pin numbers and types (lines 04−09). Line 45 is the end statement that completes the design module.

The corresponding VHDL code is shown in *Figure 4*. VHDL requires a slightly different structure. A design unit consists of an Entity section and an Ar-

```
01: module clk_gena;

02: declarations
03: device 'p335';

    "Inputs
04:  clock, ld_cnt, rstn                pin 3,1,7;
05:  incnt3, incnt2, incnt1, incnt0     pin 6,5,4,2;

    "Outputs
06:  endeqcnt, endltcnt                 pin 25,23 istype 'com';
07:  endeq0, cnteq0                     pin 24,27 istype 'com';
08:  clk_out                            pin 17 istype 'reg_d';
09:  count3, count2, count1, count0     pin 19,15,28,26;
.
.
.
45: end clk_gena;
```

Figure 3. Abel−HDL Design and I/O Declarations

chitecture section. By themselves, each of these is considered a separate design component. The Entity section defines the component name (line 01). The port statement (lines 02−06) declares the I/O of the entity. For each signal, a mode (in, out, buffer) defines the direction of the pin (buffer signifies output with feedback). The signal type is also defined here. The type of a signal defines size and possible values which that signal can take on. Type bit defines a one-bit signal that can have the values of "0" and "1". Type bit_vector (0 to 3) declares that the signal is a 4-bit vector, each bit of which can take on the values of "0" or "1". Line 07 declares the target device and is optional. Lines 08−12 declares the fixed pin assignments and is also optional. Note that this line could be written as a single line terminated with a ";". The "&" in this context signifies a continuation from the previous line. Line 13 is the end statement that terminates the *clk_genv* entity.

Lines 14 and 15 are statements that call out other libraries and packages making them visible to this design. These libraries and packages may be predefined in the VHDL language or may be user defined. They may contain components, functions, procedures, declarations, etc., which may then be used by the current design. For instance, *work.int_math.all* enables all functions contained in the package *int_math* (integer math), which is found in the work library. These functions describe the operation of the "+" and "−" operators used in the up and down counter logic descriptions. The package *rtlpkg* contains the definition of the Global Synchronous Set (gss) statement used on line 27.

The second part of a complete design unit is the Architecture section. In this section is where we find the description of the behavior of the black box defined in the Entity section. Associated with the Architecture statement are begin and end statements. Line 16 declares the Architecture name, behave, for the following statements which describe the functionality of the Entity *clk_genv*. The Entity and Architecture sections are separated because VHDL allows multiple architectures to be defined for a given entity. Only one architecture can be associated with an entity in a given design. This feature allows multiple versions of an architecture to be saved in a library. The Configuration statement is used to select a specific architecture (see Reference 3).

```
01: entity clk_genv is
02: port(clock, ld_cnt, rstn     :in bit;
03:     incnt                    :in bit_vector(3 downto 0);
04:     count                    :buffer bit_vector(3 downto 0);
05:     endeqcnt, endltcnt, clk_out :buffer bit;
06:     endeq0, cnteq0           :buffer bit);

07: attribute part_name of clk_genv : entity is ”c335”;
08: attribute pin_numbers of clk_genv : entity is
09:  ”clock:3 ld_cnt:1 rstn:7 clk_out:17 ”
10:& ”incnt(3):6 incnt(2):5 incnt(1):4 incnt(0):2 ”
11:& ”count(3):19 count(2):15 count(1):28 count(0):26 ”
12:& ”endeqcnt:25 endltcnt:23 endeq0:24 cnteq0:27”;
13: end clk_genv;

14: use work.int_math.all;
15: use work.rtlpkg.all;

16: architecture behave of clk_genv is
.
.
.
22: begin
.
.
.
60: end behave;
```

Figure 4. VHDL Desing and I/O Declarations

Internal Signal Declarations

In both Abel−HDL and VHDL, internal signals (nodes) may be defined. These signals do not connect directly to an input or an output pin and may result from buried logic or may simply represent a wire to transfer data. Both languages are similar in that the signal name and type are declared.

As can be seen in *Figure 5*, lines 10 and 11 of the Abel−HDL code declare the signals RST_CTR and ENDCNT3...0. Each is defined as type node, as opposed to type pin, which would mean that the signal

```
10:  rst_ctr                              node istype ’reg_d’;
11:  endcnt3,endcnt2,endcnt1,endcnt0      node;
12:  incnt    = [incnt3,incnt2,incnt1,incnt0];
13:  count   = [count3,count2,count1,count0];
14:  endcnt  = [endcnt3,endcnt2,endcnt1,endcnt0];
15:  outputs = [count,endeqcnt,endltcnt,endeq0,cnteq0,clk_out];
16:  cnt_state = [rst_ctr,clk_out];
```

Figure 5. Abel−HDL Signal Declarations

```
17: signal endcnt : bit_vector(3 downto 0);
18: signal cnt_state : bit_vector(0 to 1);
```

Figure 6. VHDL Signal Declarations

would be connected to an I/O pin. An optional signal attribute may be used with the keyword istype to define the signal's characteristics more explicitly. Lines 12 − 16 show the groupings of signals into sets. Defining sets allows a group of signals to be referenced by one name. Any operation performed on the set name will be performed on each member of the set.

VHDL allows signal names that represent, among others, bit vectors such as the one shown in line 17 in *Figure 6*. Here ENDCNT is equivalent to [ENDCNT(3), ENDCNT(2), ENDCNT(1), ENDCNT(0)]. As seen earlier in the entity declaration, INCNT has been defined as a port (I/O pins) and is a 4-bit vector similar to ENDCNT. Individual signals cannot be declared and grouped into sets as with Abel−HDL. Rather, groups are declared initially as bit vectors. The individual members of the set can then be operated upon separately or as a group (line 58 of the VHDL source code shows cnt_state(1) being assigned to the output CLK_OUT).

State-Machine State Definitions

The section where state register assignments are declared is very similar for Abel−HDL and VHDL. Both languages require assignment of a constant value to a name which gets compared to the current value of the state bits (cnt_state) in the actual state machine implementation (IF-THEN-ELSE, CASE-WHEN). Shown in this document is one method of designing a state machine. Both Abel−HDL and VHDL allow a variety of ways in which to create a

```
17:  reset     = [0,0];
18:  cnt_up    = [1,1];
19:  cnt_down  = [1,0];
```

Figure 7. Abel−HDL State Machine Definition

state machine. For large state machines, a more compact implementation might be with a Truth-Table in which inputs and outputs are described in a tabular form. This method is more compact but some may find it less "readable" than other methods. Both languages also support Mealy, Moore, and one-hot (one register per state) state machine implementations. *Figure 7* shows the Abel−HDL state assignment code.

Figure 8 shows the VHDL code needed to make state assignments. Note here the increased verbosity relative to Abel−HDL. This, again, is due to the fact that VHDL is a more general-purpose language and that statements must be more explicit. Here, a constant is defined to be a certain type (bit_vector) and then is assigned a initial value using the ":=" operator. Note that VHDL's usage of the ":=" operator is different than its meaning as a registered signal assignment operator in Abel−HDL.

Combinatorial Logic Equations

Both Abel−HDL and VHDL allow combinatorial Boolean) logic equations. As *Figures 9* and *10* show, the syntax is quite similar. Combinatorial statements in Abel−HDL are signified by a "=" operator. Use of the *istype reg* attribute in the signal declaration section and/or use of the appropriate explicit

```
19: constant reset : bit_vector(0 to 1) := "00";
20: constant cnt_up : bit_vector(0 to 1) := "11";
21: constant cnt_down : bit_vector(0 to 1) := "10";
```

Figure 8. VHDL State Machine Definition

```
20: equations
21:    endeqcnt = ((endcnt.fb - 1) == count.fb);
22:    endltcnt = (endcnt.fb < count.fb);
23:    endeq0 = (endcnt.fb == 0);
24:  cnteq0 = (count.fb == 1);
25:    outputs.sp = !rstn;
26:    count.clk = clock;
```

Figure 9. Abel−HDL Combinatorial Logic Equations

```
23:  endeqcnt <= '1' when (count = (endcnt-1)) else '0';
24:  endltcnt <= '1' when (endcnt < count) else '0';
25:  endeq0   <= '1' when (endcnt = "0000") else '0';
26:  cnteq0   <= '1' when (count = "0001") else '0';
27:  gss <= NOT(rstn);
```

Figure 10. VHDL Combinatorial Logic Equations

signal extensions (.c, .q, .d, etc...) and the ":=" operator signifies registered logic. In VHDL, the syntax for both a combinatorial and registered signal assignment is the same, "<=". The difference being determined by where in the code the statement appears. It is treated as a registered signal only if the signal assignment statement occurs inside of a clocked process. (See References 1 and 3 for full explanation of processes.)

Since the CY7C335 used in this example, like many other devices, contains special features, such as global or individual resets, presets, or OEs, there needs to be a means to expressly access them. Lines 25 and 27 of the Abel−HDL and VHDL, respectively, show how to access the available global synchronous preset of the device. In the Abel−HDL code we have defined a set to be all of the output signals (see Line 14). By simply using the .sp extension, !rstn is assigned to the global synchronous preset signal. Note that it is not necessary to define a set. Since in this device the preset is global, by assigning !rstn to the preset of one output register, all are automatically connected. Line 26 assigns the signal clock to the counter registers.

In VHDL, since extensions are not allowed in signal assignments, special functions are created and placed in a standard package which access these specific device features. In this case, the gss (global synchronous set) function is found in the package *rtlpkg*. When using a function (or other statement) found in a package, a use statement must be added (Line 15) so that the contents of the package are visible to this design. This requirement may seem cumbersome on the surface, it in fact represents one of the most powerful advantages of using VHDL. It gives the ability to save and reuse commonly used statements in standard or user defined packages which can then be accessed by any design. These packages can in turn be placed in libraries which can be organized by function, project, etc.

Input Register Logic Definition

The CY7C335 was chosen for this example because it can be used to illustrate a variety of features that may be found in other programmable logic devices. In this section, we are making use of the registered inputs. The inputs are defined as INCNT(3:0). Once registered, these signals become ENDCNT(3:0) and are assigned to internal nodes in the device.

In the Abel_HDL code shown in *Figure 11*, Line 27 assigns the signal LD_CNT to be the clock for the ENDCNT registers. Line 28 uses the registered assignment operator, ":=", to create ENDCNT from INCNT.

```
27:  endcnt.clk = ld_cnt;
28:  endcnt := incnt;
```

Figure 11. Abel−HDL Input Register Definition

In VHDL, to create registered logic, a process must be used. This highlights a key concept in VHDL, the notion of concurrent vs. sequential statements. All concurrent statements are continuously and simultaneously evaluated, creating combinatorial logic. Sequential statements, as the name implies, are evaluated in order. The IF-THEN-ELSE construct is a classic example of a sequential statement. In VHDL, only statements found within a process are sequential. The processes themselves are concurrent and are continuously evaluated at the same time as all other statements between the begin and end of an Architecture section. A process is awakened (i.e., evaluated) when a change occurs in the value of a signal that is sensitive to that process. Sensitive signals are defined by the use of a sensitivity list or a wait statement at the beginning of the process. In our example we have used the wait statement to awaken a process when the clock signal for the associated logic sees a rising edge (Lines 28−31 of *Figure 12*). Here, the *in_reg* process is evaluated when a rising edge occurs on LD_CNT. Inside the process statements are evaluated sequentially. In this process there is a single statement which, when evaluated, causes the value of INCNT to be transferred to ENDCNT. This effectively creates a register clocked by LD_CNT.

State-Machine Description

The description of the simple three-state state machine (see *Figure 2*) is shown next for both Abel−HDL and VHDL in *Figures 13* and *14*, respectively. In general, both languages allow a variety of state machine definition methods. Included are truth tables, IF-THEN-ELSE statements, and CASE-WHEN statements. Both implementations require a state machine name declaration, clock declaration, and state descriptions.

```
28: in_reg : process begin
29:     wait until (ld_cnt = '1');
30:      endcnt <= incnt;
31: end process;
```

Figure 12. VHDL Input Register Definition

```
29:  cnt_state.clk = clock;
30:  state_diagram cnt_state
31:       state reset:
32:                 count := 0000;
33:            if (endeq0) then reset;
34:            else cnt_up;

35:       state cnt_up:
36:                 count := (count.fb + 1);
37:            if (endltcnt # endeq0) then reset;
38:            else if (endeqcnt) then cnt_down;
39:            else cnt_up;

40:       state cnt_down:
41:                 count := (count.fb - 1);
42:            if (endltcnt # endeq0) then reset;
43:            else if (cnteq0) then cnt_up;
44:            else cnt_down;
```

Figure 13. Abel−HDL State Machine Equations

```
32: counter : process begin
33:  wait until (clock = '1');
34:       case cnt_state is
35:       when reset =>
36:         count <= "0000";
37:       if (endeq0 = '0') then cnt_state <= cnt_up;
38:       else cnt_state <= reset;
39:       end if;

40:       when cnt_up =>
41:      count <= count +1;
42:       if (endltcnt='1' OR endeq0='1') then
43:      cnt_state <= reset;
44:       elsif (endeqcnt = '1') then cnt_state <= cnt_down;
45:       else cnt_state <= cnt_up;
46:       end if;

47:       when cnt_down =>
48:      count <= count - 1;
49:       if (endltcnt='1' OR endeq0='1') then
50:      cnt_state <= reset;
51:       elsif (cnteq0 = '1') then cnt_state <= cnt_up;
52:       else cnt_state <= cnt_down;
53:       end if;

54:       when others =>
55:      count <= "0000";
56:      cnt_state <= reset;

57:       end case;
58: end process;
59:  clk_out <= cnt_state(1);
```

Figure 14. VHDL State Machine Equations

In the Abel−HDL code of *Figure 13*, line 29 declares the signal clock to be the clock source for the state registers. Line 30 declares the following state descriptions to be for the state machine cnt_state. Lines 31−44 are the descriptions for each of the three states. Within each state description can be found signal assignments and IF-THEN-ELSE statements defining the conditional next-state assignments.

Similar to Abel−HDL, the VHDL code of *Figure 14* contains a clock declaration on line 33 (the wait until statement implies clock is the register clock in this process), and a state machine declaration on line 34 (the case statement defines cnt_state as the state machine under evaluation). Lines 35−57 contain the individual state descriptions. Lines 32 and 58 declare the beginning and end of the process called counter. This explicit declaration of the beginning and end of processes is necessary because of the VHDL distinction between concurrent statements and sequential (within a process) statements. Note the addition of the "when others" statement. This is added to insure that the state machine can recover from an invalid (undefined) state. Lastly, line 59 as-

signs the value of the state bit cnt_state(1) to the output signal CLK_OUT. Had this statement been placed inside the process it would have been treated as a sequential statement and, therefore, would be registered. This would have caused a registered, or pipelined, delay to be added to CLK_OUT.

Summary

In summary, as design languages, Abel−HDL and IEEE−VHDL are really quite similar in complexity. Many experienced Abel−HDL users may perceive VHDL to be unnecessarily complicated. This may be true if one is limited to the smaller playing field covered by Abel−HDL. VHDL, on the other hand, covers a broader set of applications, such as full system-level description and simulation. The extra verbosity is minimal when compared to the extra functionality provided. For instance, VHDL allows true source code simulation, an easy migration path to ASICs (standard, portable language), and design with different types such as integers, enumerated types, records, etc. It also is truly device independent. For instance, falling edge clocks and XORs can be described behaviorally in VHDL whereas in Abel−HDL, the target device must be declared and specific fuses programmed to make use of these special features.

References

1. Cypress Semiconductor, Warp2 *User's Manual*, 1993.
2. Data I/O, *Abel Design Software User Manual*, 1990.
3. S. Mazor and P. Langstraat, *A Guide To VHDL*, Kluwer Academic Publishers, Norwell, MA, 1992.
4. Douglas L. Perry, VHDL Second Edition, McGraw-Hill Series, Computer Engineering, 1994.

Appendix A. VHDL Design File for Prog. Clock Generator

```
01: entity clk_genv is
02: port(clock, ld_cnt, rstn  :in bit;
03:      incnt             :in bit_vector(3 downto 0);
04:      count             :buffer bit_vector(3 downto 0);
05:      endeqcnt, endltcnt, clk_out  :buffer bit;
06:      endeq0, cnteq0    :buffer bit);
07: attribute part_name of clk_genv : entity is ″c335″;
08: attribute pin_numbers of clk_genv : entity is
09:    ″clock:3 ld_cnt:1 rstn:7 clk_out:17 ″
10: & ″incnt(3):6 incnt(2):5 incnt(1):4 incnt(0):2 ″
11: & ″count(3):19 count(2):15 count(1):28 count(0):26 ″
12: & ″endeqcnt:25 endltcnt:23 endeq0:24 cnteq0:27″;
13: end clk_genv;

14: use work.int_math.all;
15: use work.rtlpkg.all;

16: architecture behave of clk_genv is

17: signal endcnt : bit_vector(3 downto 0);
18: signal cnt_state : bit_vector(0 to 1);
19: constant reset : bit_vector(0 to 1) := ″00″;
20: constant cnt_up : bit_vector(0 to 1) := ″11″;
21: constant cnt_down : bit_vector(0 to 1) := ″10″;

22: begin

23: endeqcnt <= ’1’ when (count = (endcnt-1)) else ’0’;
24: endltcnt <= ’1’ when (endcnt < count) else ’0’;
25: endeq0   <= ’1’ when (endcnt = ″0000″) else ’0’;
26: cnteq0   <= ’1’ when (count = ″0001″) else ’0’;
27: gss <= NOT(rstn);

28: in_reg : process begin
29:     wait until (ld_cnt = ’1’);
30:       endcnt <= incnt;
31: end process;

32: counter : process begin
33:    wait until (clock = ’1’);
34:    case cnt_state is
35:    when reset =>
36:        count <= ″0000″;
37:      if (endeq0 = ’0’) then cnt_state <= cnt_up;
38:      else cnt_state <= reset;
39:      end if;
```

Appendix A. VHDL Design File for Prog. Clock Generator (continued)

```
40:     when cnt_up =>
41:      count <= count +1;
42:       if (endltcnt='1' OR endeq0='1') then
43:      cnt_state <= reset;
44:       elsif (endeqcnt = '1') then cnt_state <= cnt_down;
45:       else cnt_state <= cnt_up;
46:       end if;

47:  when cnt_down =>
48:      count <= count – 1;
49:       if (endltcnt='1' OR endeq0='1') then
50:      cnt_state <= reset;
51:       elsif (cnteq0 = '1') then cnt_state <= cnt_up;
52:       else cnt_state <= cnt_down;
53:       end if;

54:    when others =>
55:      count <= "0000";
56:      cnt_state <= reset;

57:     end case;
58: end process;
59:  clk_out <= cnt_state(1);
60: end behave;
```

Appendix B. Abel−HDL Design File for Prog. Clock Generator

```
01: module clk_gena;

02: declarations
03: device 'p335';

    "Inputs
04:  clock, ld_cnt, rstn       pin 3,1,7;
05:  incnt3, incnt2, incnt1, incnt     pin 6,5,4,2;

    "Outputs
06:  endeqcnt, endltcnt      pin 25,23 istype 'com';
07:  endeq0, cnteq0        pin 24,27 istype 'com';
08:  clk_out           pin 17 istype 'reg_d';
09:  count3, count2, count1, count0    pin 19,15,28,26;

10:  rst_ctr           node istype 'reg_d';
11:  endcnt3,endcnt2,endcnt1,endcnt    node;

12:  incnt   = [incnt3,incnt2,incnt1,incnt0];
13:  count   = [count3,count2,count1,count0];
14:  endcnt  = [endcnt3,endcnt2,endcnt1,endcnt0];
15:  outputs = [count,endeqcnt,endltcnt,endeq0,cnteq0,clk_out];

16:  cnt_state = [rst_ctr,clk_out];
17:  reset     = [0,0];
18:  cnt_up    = [1,1];
19:  cnt_down  = [1,0];

20: equations

21:  endeqcnt = ((endcnt.fb – 1) == count.fb);
22:  endltcnt = (endcnt.fb < count.fb);
23:  endeq0 = (endcnt.fb == 0);
24:  cnteq0 = (count.fb == 1);
25:  outputs.sp = !rstn;
26:  count.clk = clock;
27:  endcnt.clk = ld_cnt;
28:  endcnt := incnt;

29:  cnt_state.clk = clock;
30:  state_diagram cnt_state
31:       state reset:
32:                 count := 0000;
33:            if (endeq0) then reset;
34:            else cnt_up;
```

Appendix B. Abel−HDL Design File for Prog. Clock Generator (continued)

```
35:       state cnt_up:
36:                 count := (count.fb + 1);
37:            if (endltcnt # endeq0) then reset;
38:            else if (endeqcnt) then cnt_down;
39:            else cnt_up;

40:       state cnt_down:
41:                 count := (count.fb – 1);
42:            if (endltcnt # endeq0) then reset;
43:            else if (cnteq0) then cnt_up;
44:            else cnt_down;
45: end clk_gena;
```

Abel is a trademark of Data I/O Corporation.