A

Examples

This appendix presents examples that demonstrate basic concepts of Synopsys FPGA Compiler II / FPGA *Express*:

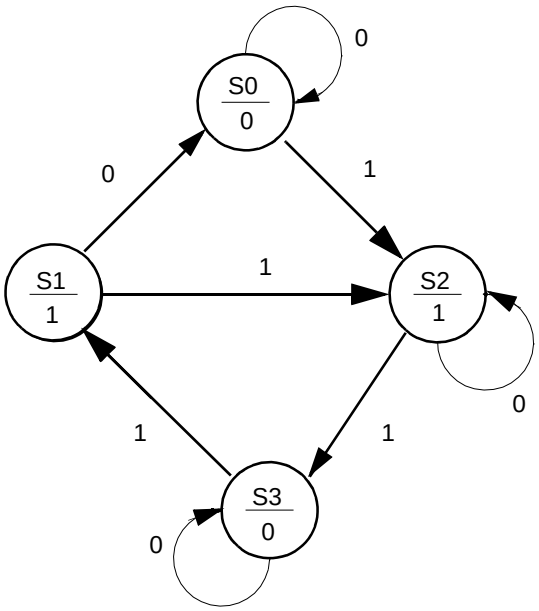
- Moore Machine
- Mealy Machine
- Read-Only Memory
- Waveform Generator
- Smart Waveform Generator
- Definable-Width Adder-Subtracter
- Count Zeros—Combinational Version
- Count Zeros—Sequential Version
- Soft Drink Machine—State Machine Version

- Soft Drink Machine—Count Nickels Version
- Carry-Lookahead Adder
- Serial-to-Parallel Converter—Counting Bits
- Serial-to-Parallel Converter—Shifting Bits
- Programmable Logic Arrays

Moore Machine

Figure A-1 is a diagram of a simple Moore finite state machine. It has one input (X), four internal states (S0 to S3), and one output (Z).

Figure A-1 Moore Machine Specification



Present state	Next state		Output (Z)
	X=0	X=1	X=0
S0	S0	S2	0
S1	S0	S2	1
S2	S2	S3	1
S3	S3	S1	0

The VHDL code implementing this finite state machine is shown in Example A-1, which includes a schematic of the synthesized circuit.

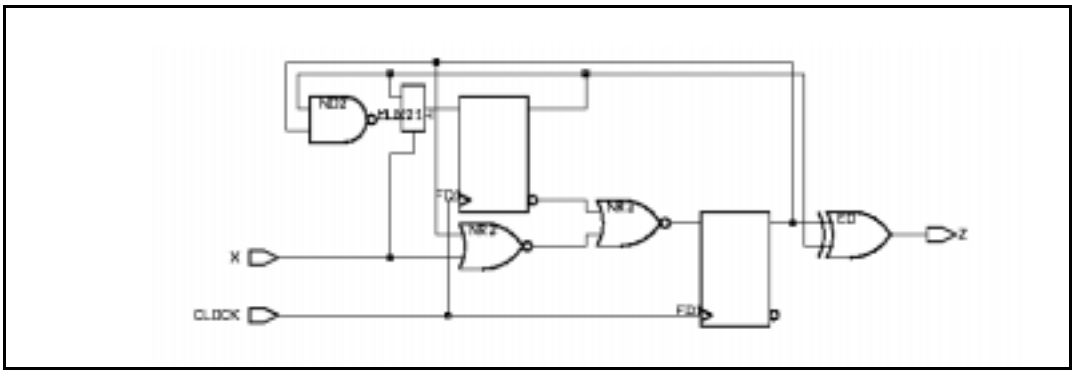
The machine description includes two processes. One process defines the synchronous elements of the design (state registers); the other process defines the combinational part of the design (state assignment case statement). For more details on using the two processes, see “Combinational Versus Sequential Processes” on page 5-55.

Example A-1 Implementation of a Moore Machine

```
entity MOORE is                -- Moore machine
  port(X, CLOCK: in BIT;
       Z: out BIT);
end MOORE;

architecture BEHAVIOR of MOORE is
  type STATE_TYPE is (S0, S1, S2, S3);
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
begin

  -- Process to hold combinational logic
  COMBIN: process(CURRENT_STATE, X)
  begin
    case CURRENT_STATE is
      when S0 =>
        Z <= '0';
        if X = '0' then
          NEXT_STATE <= S0;
        else
          NEXT_STATE <= S2;
        end if;
      when S1 =>
        Z <= '1';
        if X = '0' then
          NEXT_STATE <= S0;
        else
          NEXT_STATE <= S2;
        end if;
      when S2 =>
        Z <= '1';
        if X = '0' then
```

```
            NEXT_STATE <= S2;
          else
            NEXT_STATE <= S3;
          end if;
        when S3 =>
          Z <= '0';
          if X = '0' then
            NEXT_STATE <= S3;
          else
            NEXT_STATE <= S1;
          end if;
      end case;
    end process COMBIN;

    -- Process to hold synchronous elements (flip-flops)
    SYNCH: process
    begin
      wait until CLOCK'event and CLOCK = '1';
      CURRENT_STATE <= NEXT_STATE;
    end process SYNCH;
  end BEHAVIOR;
```

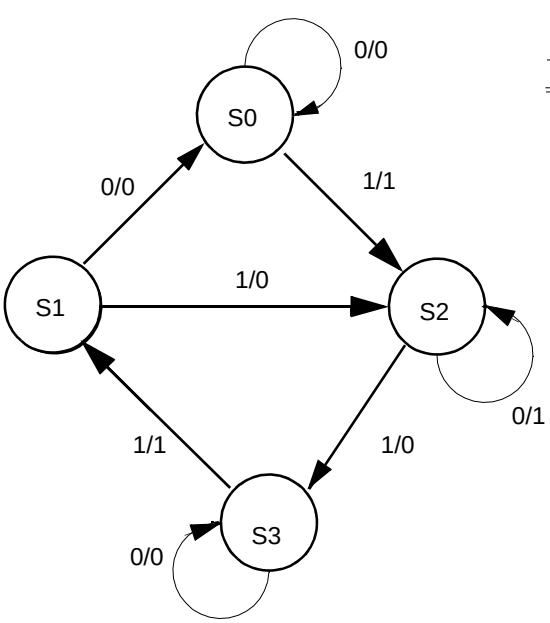
Figure A-2 Moore Machine Schematic



Mealy Machine

Figure A-3 is a diagram of a simple Mealy finite state machine. The VHDL code for implementing this finite state machine is shown in Example A-2. The machine description includes two processes, as in the previous Moore machine example.

Figure A-3 Mealy Machine Specification



Present state	Next state		Output (Z)	
	X=0	X=1	X=0	X=1
S0	S0	S2	0	1
S1	S0	S2	0	0
S2	S2	S3	1	0
S3	S3	S1	0	1

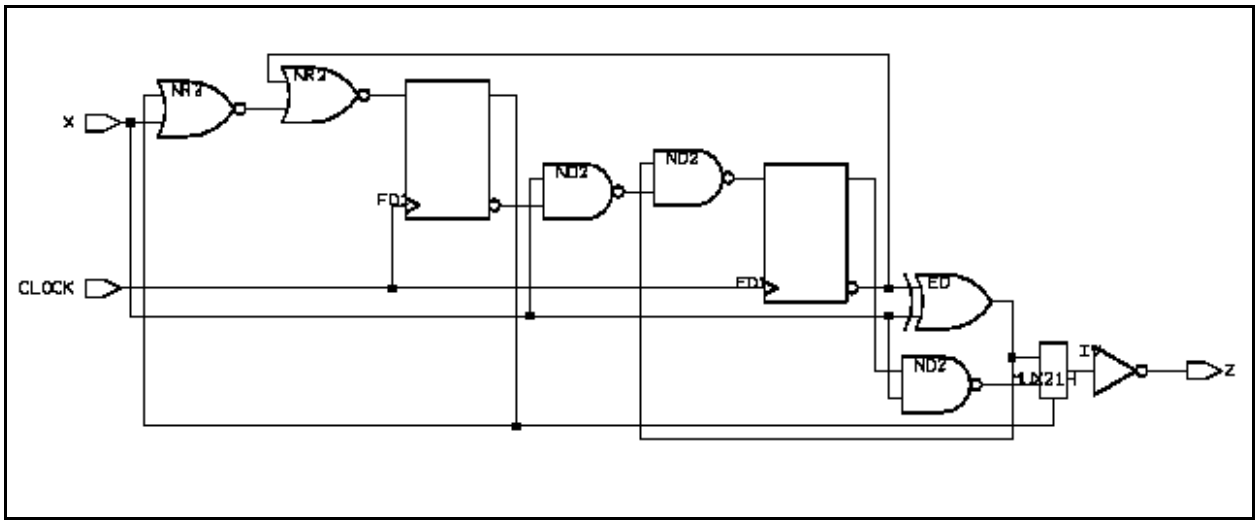
Example A-2 Implementation of a Mealy Machine

```
entity MEALY is              -- Mealy machine
  port(X, CLOCK: in BIT;
       Z: out BIT);
end MEALY;

architecture BEHAVIOR of MEALY is
  type STATE_TYPE is (S0, S1, S2, S3);
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
```

```
begin

  -- Process to hold combinational logic.
  COMBIN: process(CURRENT_STATE, X)
  begin
    case CURRENT_STATE is
      when S0 =>
        if X = '0' then
          Z <= '0';
          NEXT_STATE <= S0;
        else
          Z <= '1';
          NEXT_STATE <= S2;
        end if;
      when S1 =>
        if X = '0' then
          Z <= '0';
          NEXT_STATE <= S0;
        else
          Z <= '0';
          NEXT_STATE <= S2;
        end if;
      when S2 =>
        if X = '0' then
          Z <= '1';
          NEXT_STATE <= S2;
        else
          Z <= '0';
          NEXT_STATE <= S3;
        end if;
      when S3 =>
        if X = '0' then
          Z <= '0';
          NEXT_STATE <= S3;
        else
          Z <= '1';
          NEXT_STATE <= S1;
        end if;
    end case;
  end process COMBIN;
  -- Process to hold synchronous elements (flip-flops)
  SYNCH: process
```

```
  begin
    wait until CLOCK'event and CLOCK = '1';
    CURRENT_STATE <= NEXT_STATE;
  end process SYNCH;
end BEHAVIOR;
```

Figure A-4 Mealy Machine Schematic



Read-Only Memory

Example A-3 shows how you can define a read-only memory in VHDL. The ROM is defined as an array constant, ROM. Each line of the constant array specification defines the contents of one ROM address. To read from the ROM, index into the array.

The number of ROM storage locations and bit-width is easy to change. The subtype ROM_RANGE specifies that the ROM contains storage locations 0 to 7. The constant ROM_WIDTH specifies that the ROM is 5 bits wide.

After you define a ROM constant, you can index into that constant many times to read many values from the ROM. If the ROM address is computable (see “Computable Operands” on page 4-16), no logic is built and the appropriate data value is inserted. If the ROM address is not computable, logic is built for each index into the value. In Example A-3, ADDR is not computable, so logic is synthesized to compute the value.

FPGA Compiler II / FPGA *Express* does not actually instantiate a typical array-logic ROM, such as those available from ASIC vendors. Instead, it creates the ROM from random logic gates (AND, OR, NOT, and so on). This type of implementation is preferable for small ROMs and for ROMs that are regular. For very large ROMs, consider using an array-logic implementation supplied by your ASIC vendor.

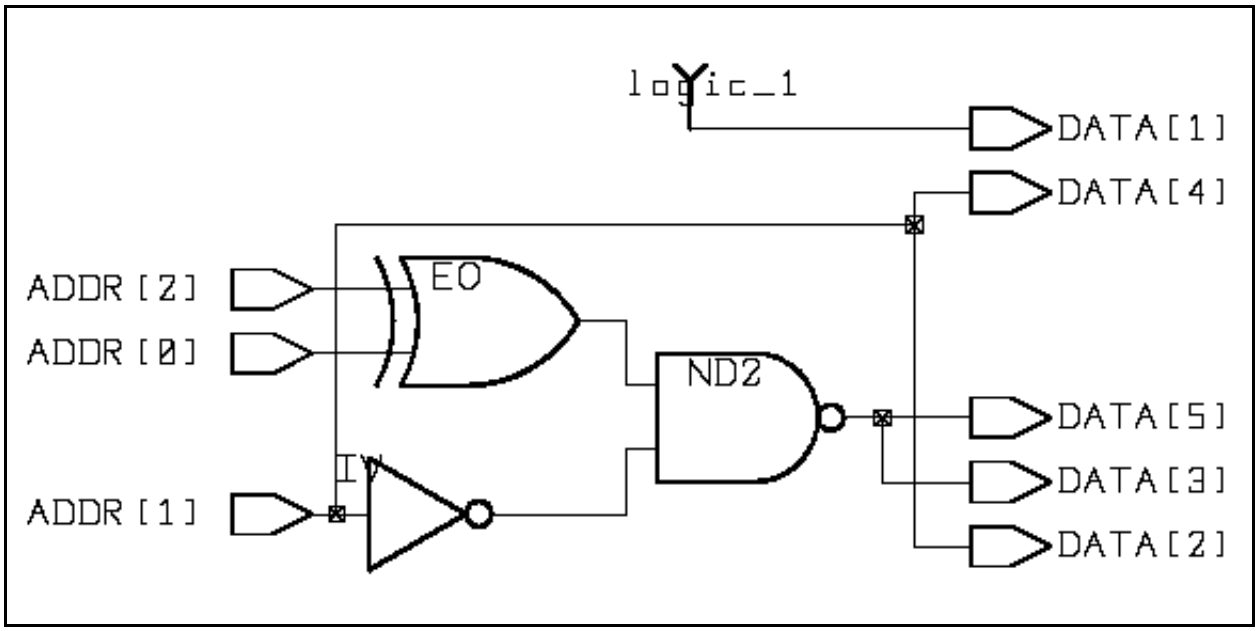
Example A-3 shows the VHDL source code and the synthesized circuit schematic.

Example A-3 Implementation of a ROM in Random Logic

```
package ROMS is
  -- declare a 5x8 ROM called ROM
  constant ROM_WIDTH: INTEGER := 5;
  subtype ROM_WORD is BIT_VECTOR (1 to ROM_WIDTH);
  subtype ROM_RANGE is INTEGER range 0 to 7;
  type ROM_TABLE is array (0 to 7) of ROM_WORD;
  constant ROM: ROM_TABLE := ROM_TABLE'(
      ROM_WORD'("10101"),              -- ROM contents
      ROM_WORD'("10000"),
      ROM_WORD'("11111"),
      ROM_WORD'("11111"),
      ROM_WORD'("10000"),
      ROM_WORD'("10101"),
      ROM_WORD'("11111"),
      ROM_WORD'("11111"));
end ROMS;
use work.ROMS.all;   -- Entity that uses ROM
entity ROM_5x8 is
```

```
  port(ADDR: in ROM_RANGE;
       DATA: out ROM_WORD);
end ROM_5x8;
architecture BEHAVIOR of ROM_5x8 is
begin
  DATA <= ROM(ADDR);      -- Read from the ROM
end BEHAVIOR;
```

Figure A-5 ROM Schematic



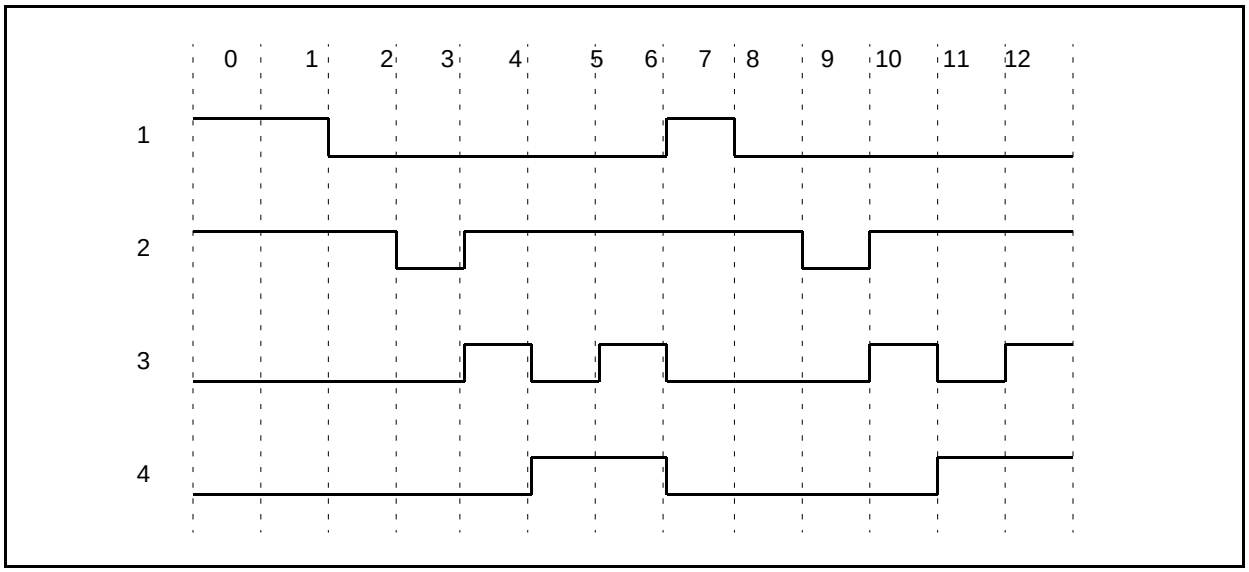
Waveform Generator

The waveform generator example shows how to use the previous ROM example to implement a waveform generator.

Assume that you want to produce the waveform output shown in Figure A-6.

1. First, declare a ROM wide enough to hold the output signals (4 bits) and deep enough to hold all time steps (0 to 12, for a total of 13).
2. Next, define the ROM so that each time step is represented by an entry in the ROM.
3. Finally, create a counter that cycles through the time steps (ROM addresses), generating the waveform at each time step.

Figure A-6 Waveform Example



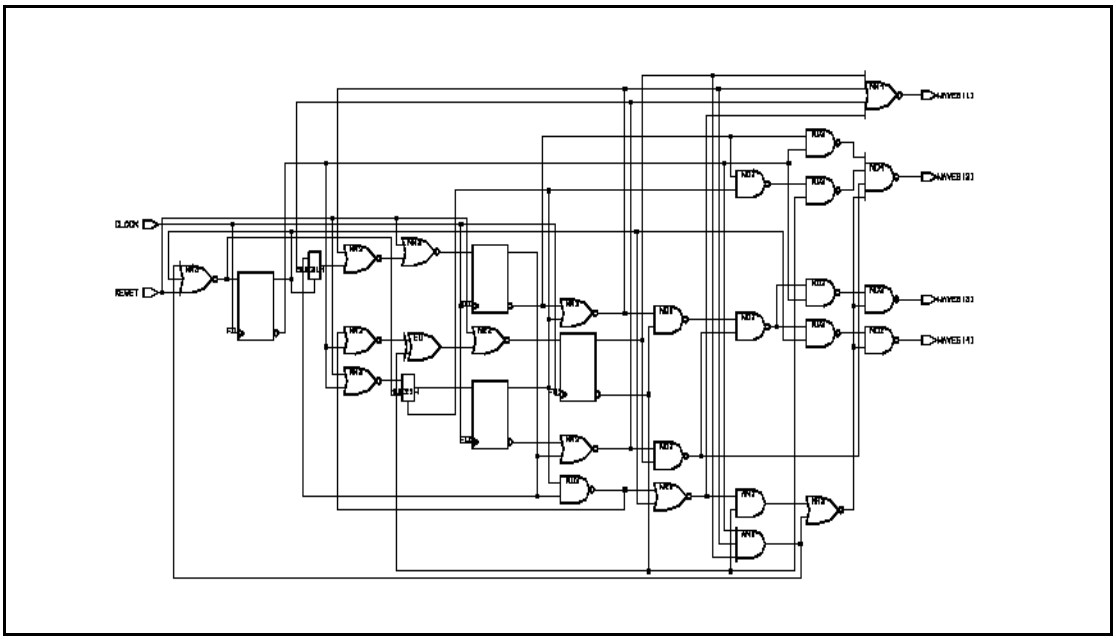
Example A-4 shows an implementation for the waveform generator. It consists of a ROM, a counter, and some simple reset logic.

Example A-4 Implementation of a Waveform Generator

```
package ROMS is
  -- a 4x13 ROM called ROM that contains the waveform
  constant ROM_WIDTH: INTEGER := 4;
  subtype ROM_WORD is BIT_VECTOR (1 to ROM_WIDTH);
  subtype ROM_RANGE is INTEGER range 0 to 12;
  type ROM_TABLE is array (0 to 12) of ROM_WORD;
  constant ROM: ROM_TABLE := ROM_TABLE'(
      "1100",   -- time step 0
      "1100",   -- time step 1
      "0100",   -- time step 2
      "0000",   -- time step 3
      "0110",   -- time step 4
      "0101",   -- time step 5
      "0111",   -- time step 6
      "1100",   -- time step 7
      "0100",   -- time step 8
      "0000",   -- time step 9
      "0110",   -- time step 10
      "0101",   -- time step 11
      "0111");  -- time step 12
end ROMS;

use work.ROMS.all;
entity WAVEFORM is            -- Waveform generator
  port(CLOCK: in BIT;
       RESET: in BOOLEAN;
       WAVES: out ROM_WORD);
end WAVEFORM;
```

```
architecture BEHAVIOR of WAVEFORM is
  signal STEP: ROM_RANGE;
begin

  TIMESTEP_COUNTER: process   -- Time stepping process
  begin
    wait until CLOCK'event and CLOCK = '1';
    if RESET then             -- Detect reset
      STEP <= ROM_RANGE'low;  -- Restart
    elsif STEP = ROM_RANGE'high then  -- Finished?
      STEP <= ROM_RANGE'high;  -- Hold at last value
   -- STEP <= ROM_RANGE'low;   -- Continuous wave
    else
      STEP <= STEP + 1;        -- Continue stepping
    end if;
  end process TIMESTEP_COUNTER;

  WAVES <= ROM(STEP);
end BEHAVIOR;
```

Figure A-7 Waveform Generator Schematic



When the counter STEP reaches the end of the ROM, STEP stops, generates the last value, then waits until a reset. To make the sequence automatically repeat, remove the following statement:

```
STEP <= ROM_RANGE'high;  -- Hold at last value
```

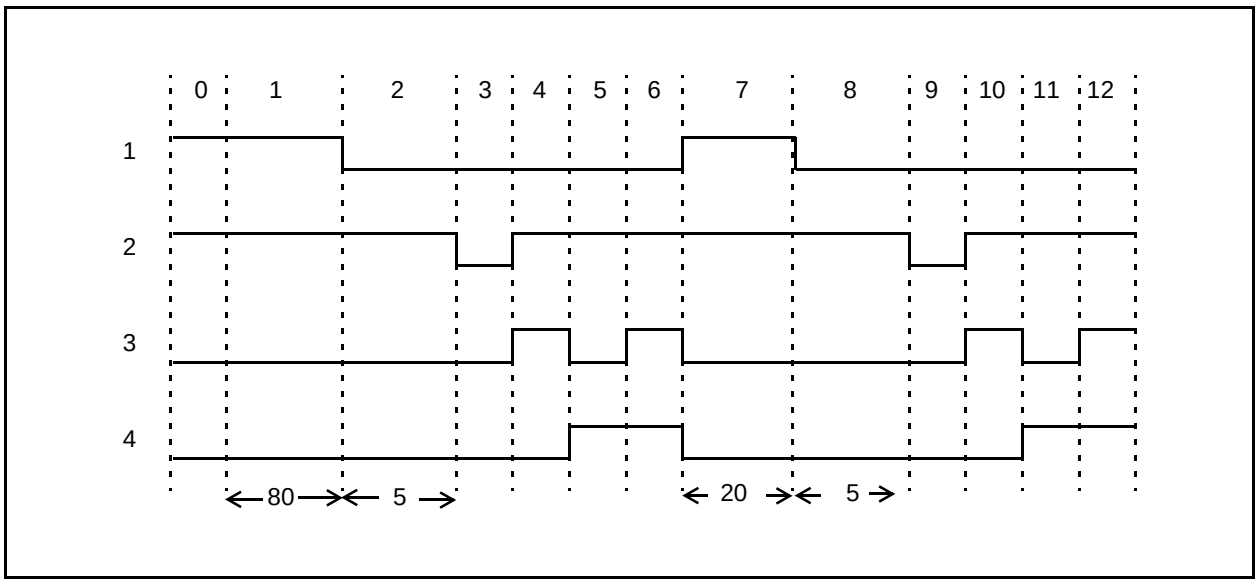
Use the following statement instead (commented out in Example A-4):

```
STEP <= ROM_RANGE'low;   -- Continuous wave
```

Smart Waveform Generator

The smart waveform generator in Figure A-8 is an extension of the waveform generator in Figure A-6 on page A-10. But this smart waveform generator is capable of holding the waveform at any time step for several clock cycles.

Figure A-8 Waveform for Smart Waveform Generator Example



The implementation of the smart waveform generator is shown in Example A-5. It is similar to the waveform generator in Example A-4 on page A-11, but has two additions. A new ROM, D_ROM, has been added to hold the length of each time step. A value of 1 specifies that the corresponding time step should be one clock cycle long; a value of 80 specifies that the time step should be 80 clock cycles long. The second addition to the previous waveform generator is a delay counter that counts the clock cycles between time steps.

In the architecture of this example, a selected signal assignment determines the value of the NEXT_STEP counter.

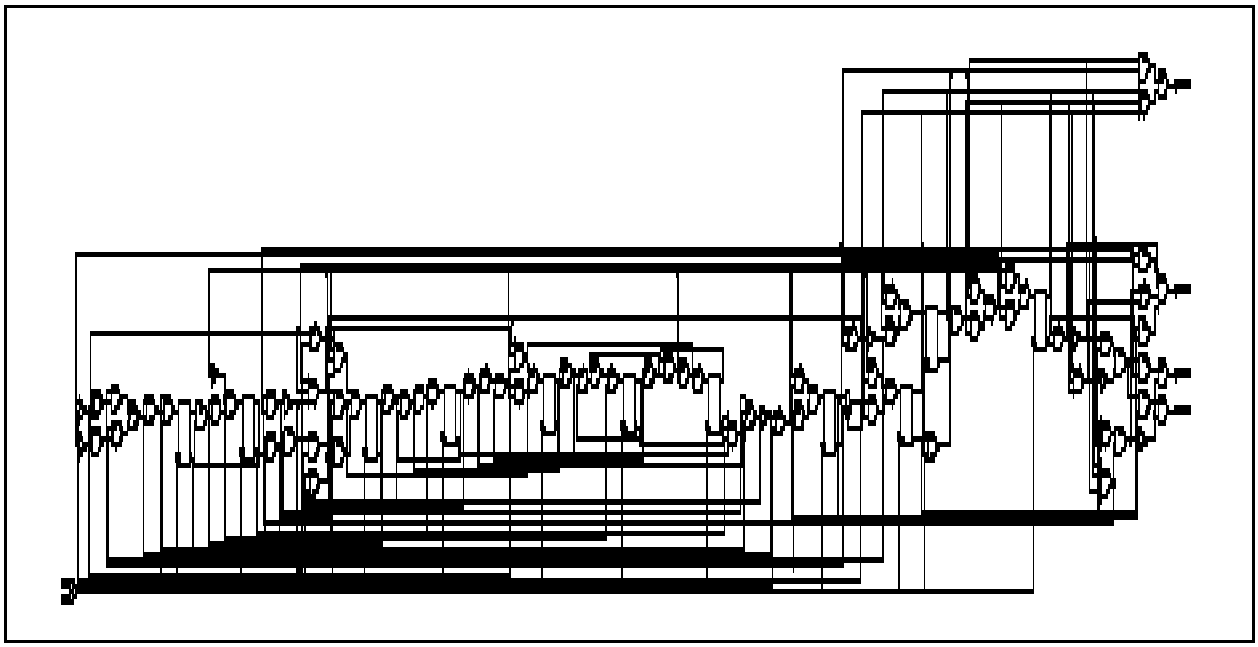
Example A-5 Implementation of a Smart Waveform Generator

```
package ROMS is

  -- a 4x13 ROM called W_ROM containing the waveform
  constant W_ROM_WIDTH: INTEGER := 4;
  subtype W_ROM_WORD is BIT_VECTOR (1 to W_ROM_WIDTH);
  subtype W_ROM_RANGE is INTEGER range 0 to 12;
  type W_ROM_TABLE is array (0 to 12) of W_ROM_WORD;
  constant W_ROM: W_ROM_TABLE := W_ROM_TABLE'(
    "1100",   -- time step 0
    "1100",   -- time step 1
    "0100",   -- time step 2
    "0000",   -- time step 3
    "0110",   -- time step 4
    "0101",   -- time step 5
    "0111",   -- time step 6
    "1100",   -- time step 7
    "0100",   -- time step 8
    "0000",   -- time step 9
    "0110",   -- time step 10
    "0101",   -- time step 11
    "0111");  -- time step 12

  -- a 7x13 ROM called D_ROM containing the delays
  subtype D_ROM_WORD is INTEGER range 0 to 100;
  subtype D_ROM_RANGE is INTEGER range 0 to 12;
```

```
  type D_ROM_TABLE is array (0 to 12) of D_ROM_WORD;
  constant D_ROM: D_ROM_TABLE := D_ROM_TABLE'(
      1,80,5,1,1,1,1,20,5,1,1,1,1);
end ROMS;

use work.ROMS.all;
entity WAVEFORM is        -- Smart Waveform Generator
  port(CLOCK: in BIT;
       RESET: in BOOLEAN;
       WAVES: out W_ROM_WORD);
end WAVEFORM;

architecture BEHAVIOR of WAVEFORM is
  signal STEP, NEXT_STEP: W_ROM_RANGE;
  signal DELAY: D_ROM_WORD;
begin

  -- Determine the value of the next time step
  NEXT_STEP <= W_ROM_RANGE'high when
                    STEP = W_ROM_RANGE'high
               else
                 STEP + 1;
  -- Keep track of which time step we are in
  TIMESTEP_COUNTER: process
  begin
    wait until CLOCK'event and CLOCK = '1';
    if RESET then             -- Detect reset
      STEP <= 0;              -- Restart waveform
    elsif DELAY = 1 then
      STEP <= NEXT_STEP;      -- Continue stepping
    else
      null;          -- Wait for DELAY to count down;
    end if;          -- do nothing here
  end process TIMESTEP_COUNTER;

   -- Count the delay between time steps
  DELAY_COUNTER: process
  begin
    wait until CLOCK'event and CLOCK = '1';
    if RESET then             -- Detect reset
      DELAY <= D_ROM(0);      -- Restart
    elsif DELAY = 1 then      -- Have we counted down?
```

```
        DELAY <= D_ROM(NEXT_STEP);  -- Next delay value
      else
        DELAY <= DELAY - 1;   -- decrement DELAY counter
      end if;
    end process DELAY_COUNTER;

    WAVES <= W_ROM(STEP);     -- Output waveform value
  end BEHAVIOR;
```

Figure A-9 Smart Waveform Generator Schematic



Definable-Width Adder-Subtracter

VHDL lets you create functions for use with array operands of any size. This example shows an adder-subtracter circuit that, when called, is adjusted to fit the size of its operands.

Example A-6 shows an adder-subtracter defined for two unconstrained arrays of bits (type BIT_VECTOR) in a package named MATH. When an unconstrained array type is used for an argument to a subprogram, the actual constraints of the array are taken from the actual parameter values in a subprogram call.

Example A-6 MATH Package for Example A-7

```
package MATH is
  function ADD_SUB(L, R: BIT_VECTOR; ADD: BOOLEAN)
      return BIT_VECTOR;
    -- Add or subtract two BIT_VECTORs of equal length
end MATH;

package body MATH is
    function ADD_SUB(L, R: BIT_VECTOR; ADD: BOOLEAN)
        return BIT_VECTOR is
      variable CARRY: BIT;
      variable A, B, SUM:
          BIT_VECTOR(L'length-1 downto 0);
    begin
      if ADD then
          -- Prepare for an "add" operation
          A := L;
          B := R;
          CARRY := '0';
      else

          -- Prepare for a "subtract" operation
          A := L;
          B := not R;
          CARRY := '1';
      end if;

      -- Create a ripple carry chain; sum up bits
      for i in 0 to A'left loop
        SUM(i) := A(i) xor B(i) xor CARRY;
        CARRY := (A(i) and B(i)) or
                 (A(i) and CARRY) or
                 (CARRY and B(i));
      end loop;
```

```
      return SUM;        -- Result
    end;
end MATH;
```

Within the function ADD_SUB, two temporary variables, A and B, are declared. These variables are declared to be the same length as L (and necessarily, R) but have their index constraints normalized to L'length-1 downto 0. After the arguments are normalized, you can create a ripple carry adder by using a for loop.

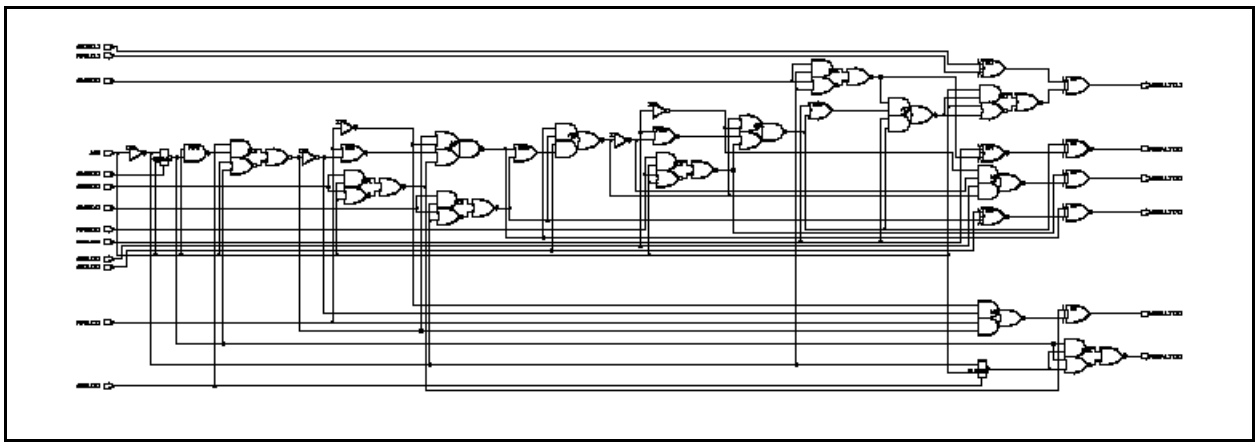
No explicit references to a fixed array length are in the function ADD_SUB. Instead, the VHDL array attributes 'left and 'length are used. These attributes allow the function to work on arrays of any length.

Example A-7 shows how to use the adder-subtracter defined in the MATH package. In this example, the vector arguments to functions ARG1 and ARG2 are declared as BIT_VECTOR(1 to 6). This declaration causes ADD_SUB to work with 6-bit arrays. A schematic of the synthesized circuit follows Example A-7.

Example A-7 Implementation of a 6-Bit Adder-Subtracter

```
use work.MATH.all;

entity EXAMPLE is
    port(ARG1, ARG2: in BIT_VECTOR(1 to 6);
         ADD: in BOOLEAN;
         RESULT : out BIT_VECTOR(1 to 6));
end EXAMPLE;

architecture BEHAVIOR of EXAMPLE is
begin
  RESULT <= ADD_SUB(ARG1, ARG2, ADD);
end BEHAVIOR;
```

Figure A-10 6-Bit Adder-Subtracter Schematic



Count Zeros—Combinational Version

The count zeros—combinational example illustrates a design problem in which an 8-bit-wide value is given and the circuit determines two things:

- That no more than one sequence of zeros is in the value.
- The number of zeros in that sequence (if any). This computation must be completed in a single clock cycle.

The circuit produces two outputs: the number of zeros found and an error indication.

A valid input value can have at most one consecutive series of zeros. A value consisting entirely of ones is defined as a valid value. If a value is invalid, the zero counter resets to 0. For example, the value 00000000 is valid and has eight zeros; value 11000111 is valid and has three zeros; value 00111100 is invalid.

Example A-8 shows the VHDL description for the circuit. It consists of a single process with a for loop that iterates across each bit in the given value. At each iteration, a temporary INTEGER variable (TEMP_COUNT) counts the number of zeros encountered. Two temporary Boolean variables (SEEN_ZERO and SEEN_TRAILING), initially false, are set to true when the beginning and end of the first sequence of zeros is detected.

If a zero is detected after the end of the first sequence of zeros (after SEEN_TRAILING is true), the zero count is reset (to 0), ERROR is set to true, and the for loop is exited.

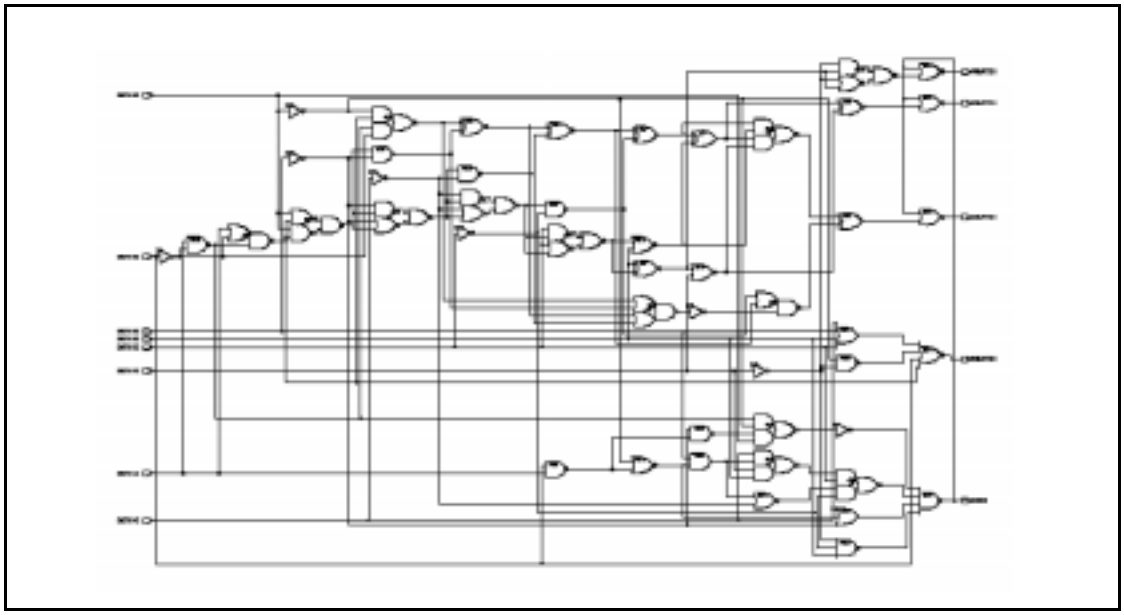
Example A-8 shows a combinational (parallel) approach to counting the zeros. The next example shows a sequential (serial) approach.

Example A-8 Count Zeros—Combinational

```
entity COUNT_COMB_VHDL is
  port(DATA:  in  BIT_VECTOR(7 downto 0);
       COUNT: out INTEGER range 0 to 8;
       ERROR: out BOOLEAN);
end COUNT_COMB_VHDL;

architecture BEHAVIOR of COUNT_COMB_VHDL is
begin
  process(DATA)
    variable TEMP_COUNT : INTEGER range 0 to 8;
    variable SEEN_ZERO, SEEN_TRAILING : BOOLEAN;
  begin
    ERROR <= FALSE;
    SEEN_ZERO <= FALSE;
    SEEN_TRAILING <= FALSE;
    TEMP_COUNT <= 0;
    for I in 0 to 7 loop
      if (SEEN_TRAILING and DATA(I) = '0') then
        TEMP_COUNT <= 0;
        ERROR <= TRUE;
        exit;
      elsif (SEEN_ZERO and DATA(I) = '1') then
```

```
        SEEN_TRAILING <= TRUE;
      elsif (DATA(I) = '0') then
        SEEN_ZERO <= TRUE;
        TEMP_COUNT <= TEMP_COUNT + 1;
      end if;
    end loop;

   COUNT <= TEMP_COUNT;
  end process;

end BEHAVIOR;
```

Figure A-11 Count Zeros—Combinational Schematic



Count Zeros—Sequential Version

The count zeros—sequential example shows a sequential (clocked) variant of the preceding design (Count Zeros—Combinational Version).

The circuit now accepts the 8-bit data value serially, 1 bit per clock cycle, by using the DATA and CLK inputs. The other two inputs are

- RESET, which resets the circuit
- READ, which causes the circuit to begin accepting data bits

The circuit's three outputs are

- IS_LEGAL, which is true if the data was a valid value
- COUNT_READY, which is true at the first invalid bit or when all 8 bits have been processed
- COUNT, the number of zeros (if IS_LEGAL is true)

Note:

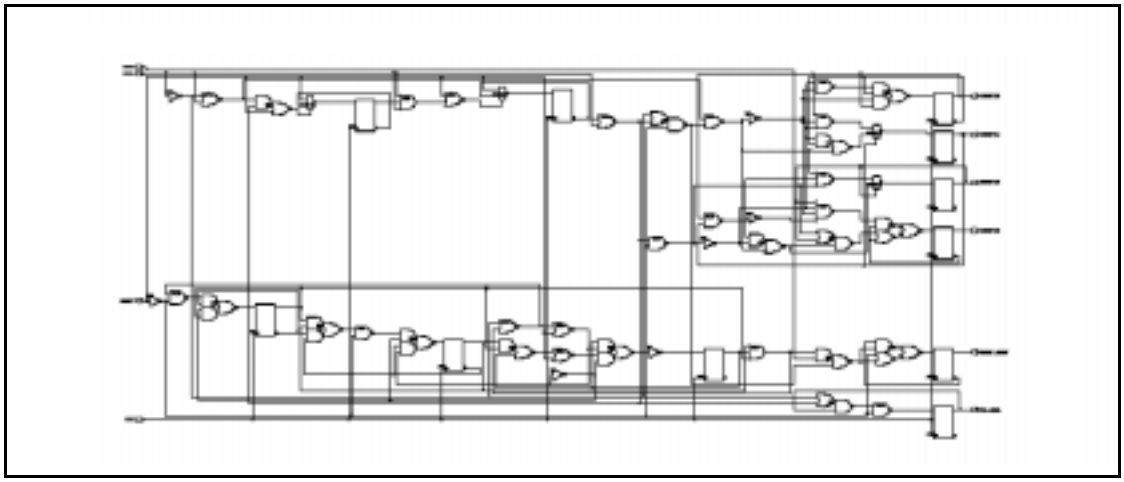
The output port COUNT is declared with mode BUFFER so that it can be read inside the process. OUT ports can only be written to, not read in.

Example A-9 Count Zeros—Sequential

```
entity COUNT_SEQ_VHDL is
  port(DATA, CLK: in BIT;
       RESET, READ: in BOOLEAN;
       COUNT: buffer INTEGER range 0 to 8;
       IS_LEGAL: out BOOLEAN;
       COUNT_READY: out BOOLEAN);
end COUNT_SEQ_VHDL;
architecture BEHAVIOR of COUNT_SEQ_VHDL is
```

```
begin
  process
    variable SEEN_ZERO, SEEN_TRAILING: BOOLEAN;
    variable BITS_SEEN: INTEGER range 0 to 7;
  begin
    wait until CLK'event and CLK = '1';

    if(RESET) then
      COUNT_READY <= FALSE;
      IS_LEGAL    <= TRUE;      -- signal assignment
      SEEN_ZERO   <= FALSE;     -- variable assignment
      SEEN_TRAILING <= FALSE;
      COUNT       <= 0;
      BITS_SEEN   <= 0;
    else
      if (READ) then
        if (SEEN_TRAILING and DATA = '0') then
          IS_LEGAL <= FALSE;
          COUNT <= 0;
          COUNT_READY <= TRUE;
        elsif (SEEN_ZERO and DATA = '1') then
          SEEN_TRAILING := TRUE;
        elsif (DATA = '0') then
          SEEN_ZERO <= TRUE;
          COUNT <= COUNT + 1;
        end if;

        if (BITS_SEEN = 7) then
          COUNT_READY <= TRUE;
        else
          BITS_SEEN <=  BITS_SEEN + 1;
        end if;

      end if;    -- if (READ)
    end if;      -- if (RESET)
  end process;
end BEHAVIOR;
```

Figure A-12 Count Zeros—Sequential Schematic



Soft Drink Machine—State Machine Version

The soft drink machine—state machine example is a control unit for a soft drink vending machine.

The circuit reads signals from a coin input unit and sends outputs to a change dispensing unit and a drink dispensing unit.

Here are the design parameters for Example A-10 and Example A-11:

- This example assumes that only one kind of soft drink is dispensed.
- This is a clocked design with CLK and RESET input signals.
- The price of the drink is 35 cents.
- The input signals from the coin input unit are NICKEL_IN (nickel deposited), DIME_IN (dime deposited), and QUARTER_IN (quarter deposited).

- The output signals to the change dispensing unit are NICKEL_OUT and DIME_OUT.
- The output signal to the drink dispensing unit is DISPENSE (dispense drink).
- The first VHDL description for this design uses a state machine description style. The second VHDL description is in Example A-11.

Example A-10 Soft Drink Machine—State Machine

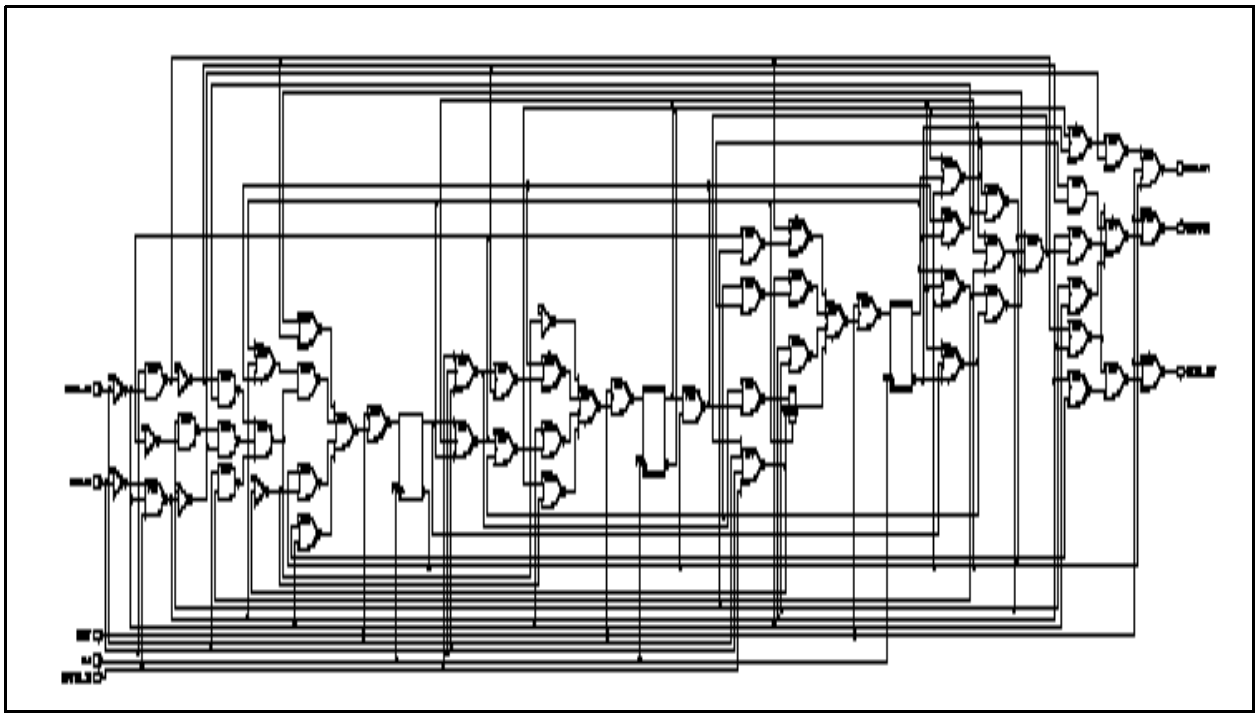
```
library synopsys; use synopsys.attributes.all;

entity DRINK_STATE_VHDL is
  port(NICKEL_IN, DIME_IN, QUARTER_IN, RESET: BOOLEAN;
       CLK: BIT;
       NICKEL_OUT, DIME_OUT, DISPENSE: out BOOLEAN);
end DRINK_STATE_VHDL;

architecture BEHAVIOR of DRINK_STATE_VHDL is
  type STATE_TYPE is (IDLE, FIVE, TEN, FIFTEEN,
                  TWENTY, TWENTY_FIVE, THIRTY, OWE_DIME);
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
  attribute STATE_VECTOR : STRING;
  attribute STATE_VECTOR of BEHAVIOR : architecture is
                       ”CURRENT_STATE”;

attribute sync_set_reset of reset : signal is ”true”;
begin

  process(NICKEL_IN, DIME_IN, QUARTER_IN,
          CURRENT_STATE, RESET, CLK)
  begin
    -- Default assignments
    NEXT_STATE <= CURRENT_STATE;
    NICKEL_OUT <= FALSE;
    DIME_OUT <= FALSE;
    DISPENSE <= FALSE;

    -- Synchronous reset
```

```
if(RESET) then
  NEXT_STATE <= IDLE;
else

  -- State transitions and output logic
  case CURRENT_STATE is
    when IDLE =>
      if(NICKEL_IN) then
        NEXT_STATE <= FIVE;
      elsif(DIME_IN) then
        NEXT_STATE <= TEN;
      elsif(QUARTER_IN) then
        NEXT_STATE <= TWENTY_FIVE;
      end if;

    when FIVE =>
      if(NICKEL_IN) then
        NEXT_STATE <= TEN;
      elsif(DIME_IN) then
        NEXT_STATE <= FIFTEEN;
      elsif(QUARTER_IN) then
        NEXT_STATE <= THIRTY;
      end if;
    when TEN =>
      if(NICKEL_IN) then
        NEXT_STATE <= FIFTEEN;
      elsif(DIME_IN) then
        NEXT_STATE <= TWENTY;
      elsif(QUARTER_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
      end if;
    when FIFTEEN =>
      if(NICKEL_IN) then
        NEXT_STATE <= TWENTY;
      elsif(DIME_IN) then
        NEXT_STATE <= TWENTY_FIVE;
      elsif(QUARTER_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
        NICKEL_OUT <= TRUE;
      end if;
```

```
when TWENTY =>
  if(NICKEL_IN) then
    NEXT_STATE <= TWENTY_FIVE;
  elsif(DIME_IN) then
    NEXT_STATE <= THIRTY;
  elsif(QUARTER_IN) then
    NEXT_STATE <= IDLE;
    DISPENSE <= TRUE;
    DIME_OUT <= TRUE;
  end if;

when TWENTY_FIVE =>
  if(NICKEL_IN) then
    NEXT_STATE <= THIRTY;
  elsif(DIME_IN) then
    NEXT_STATE <= IDLE;
    DISPENSE <= TRUE;
  elsif(QUARTER_IN) then
    NEXT_STATE <= IDLE;
    DISPENSE <= TRUE;
    DIME_OUT <= TRUE;
    NICKEL_OUT <= TRUE;
  end if;

when THIRTY =>
  if(NICKEL_IN) then
    NEXT_STATE <= IDLE;
    DISPENSE <= TRUE;
  elsif(DIME_IN) then
    NEXT_STATE <= IDLE;
    DISPENSE <= TRUE;
    NICKEL_OUT <= TRUE;
  elsif(QUARTER_IN) then
    NEXT_STATE <= OWE_DIME;
    DISPENSE <= TRUE;
    DIME_OUT <= TRUE;
  end if;

when OWE_DIME =>
  NEXT_STATE <= IDLE;
  DIME_OUT <= TRUE;
```

```
        end case;
      end if;
    end process;

    -- Synchronize state value with clock
    -- This causes it to be stored in flip-flops
    process
    begin
      wait until CLK'event and CLK = '1';
      CURRENT_STATE <= NEXT_STATE;
    end process;
  end BEHAVIOR;
```

Figure A-13 Soft Drink Machine—State Machine Schematic



Soft Drink Machine—Count Nickels Version

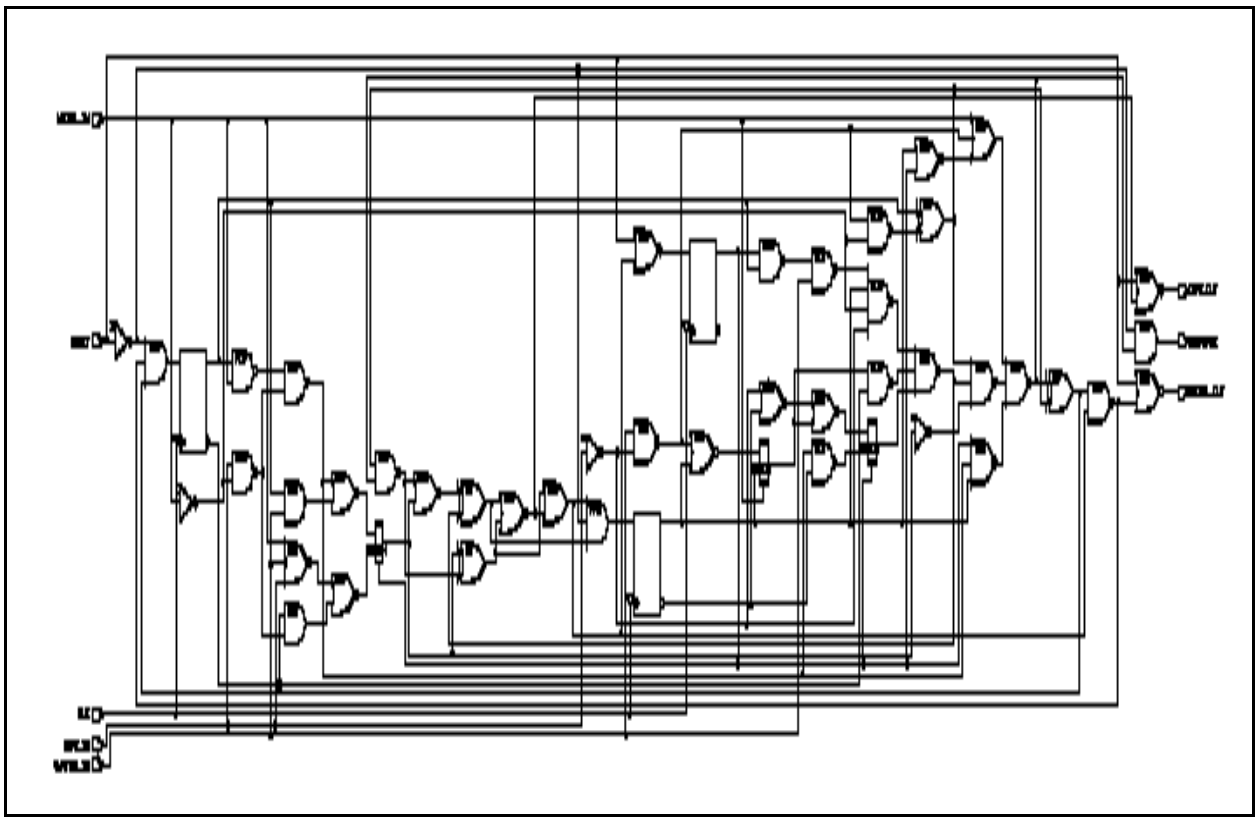
The soft drink machine—count nickels example uses the same design parameters as the preceding Example A-10 (Soft Drink Machine—State Machine), with the same input and output signals. In this version, a counter counts the number of nickels deposited. This counter is incremented by 1 if the deposit is a nickel, by 2 if it is a dime, and by 5 if it is a quarter.

Example A-11 Soft Drink Machine—Count Nickels

```
entity DRINK_COUNT_VHDL is
  port(NICKEL_IN, DIME_IN, QUARTER_IN, RESET: BOOLEAN;
       CLK: BIT;
       NICKEL_OUT, DIME_OUT, DISPENSE: out BOOLEAN);
end DRINK_COUNT_VHDL;

architecture BEHAVIOR of DRINK_COUNT_VHDL is
  signal CURRENT_NICKEL_COUNT,
         NEXT_NICKEL_COUNT: INTEGER range 0 to 7;
  signal CURRENT_RETURN_CHANGE, NEXT_RETURN_CHANGE : BOOLEAN;
begin

  process(NICKEL_IN, DIME_IN, QUARTER_IN, RESET, CLK,
          CURRENT_NICKEL_COUNT, CURRENT_RETURN_CHANGE)
    variable TEMP_NICKEL_COUNT: INTEGER range 0 to 12;
  begin
    -- Default assignments
    NICKEL_OUT <= FALSE;
    DIME_OUT <= FALSE;
    DISPENSE <= FALSE;
    NEXT_NICKEL_COUNT <= 0;
    NEXT_RETURN_CHANGE <= FALSE;

    -- Synchronous reset
    if (not RESET) then
      TEMP_NICKEL_COUNT <= CURRENT_NICKEL_COUNT;

      -- Check whether money has come in
      if (NICKEL_IN) then
        -- NOTE:  This design will be flattened, so
        --   these multiple adders will be optimized
        TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT + 1;
```

```
      elsif(DIME_IN) then
        TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT + 2;
      elsif(QUARTER_IN) then
        TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT + 5;
      end if;

      -- Enough deposited so far?
      if(TEMP_NICKEL_COUNT >= 7) then
        TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT - 7;
        DISPENSE <= TRUE;
      end if;

      -- Return change
      if(TEMP_NICKEL_COUNT >= 1 or
         CURRENT_RETURN_CHANGE) then
        if(TEMP_NICKEL_COUNT >= 2) then
          DIME_OUT <= TRUE;
          TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT - 2;
          NEXT_RETURN_CHANGE <= TRUE;
        end if;
        if(TEMP_NICKEL_COUNT = 1) then
          NICKEL_OUT <= TRUE;
          TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT - 1;
        end if;
      end if;

      NEXT_NICKEL_COUNT <= TEMP_NICKEL_COUNT;
    end if;
  end process;

  -- Remember the return-change flag and
  -- the nickel count for the next cycle
  process
  begin
    wait until CLK'event and CLK = '1';
    CURRENT_RETURN_CHANGE <= NEXT_RETURN_CHANGE;
    CURRENT_NICKEL_COUNT <= NEXT_NICKEL_COUNT;
  end process;

end BEHAVIOR;
```

Figure A-14 Soft Drink Machine—Count Nickels Version Schematic



Carry-Lookahead Adder

This example uses concurrent procedure calls to build a 32-bit carry-lookahead adder. The adder is built by partitioning of the 32-bit input into eight slices of 4 bits each. Each of the eight slices computes propagate and generate values by using the PG procedure.

Propagate (output P from PG) is ’1’ for a bit position if that position propagates a carry from the next-lower position to the next-higher position. Generate (output G) is ’1’ for a bit position if that position generates a carry to the next-higher position, regardless of the carry-in from the next lower position. The carry-lookahead logic reads the carry-in, propagate, and generate information computed from the inputs. The logic computes the carry value for each bit position and makes the addition operation an XOR of the inputs and the carry values.

Carry Value Computations

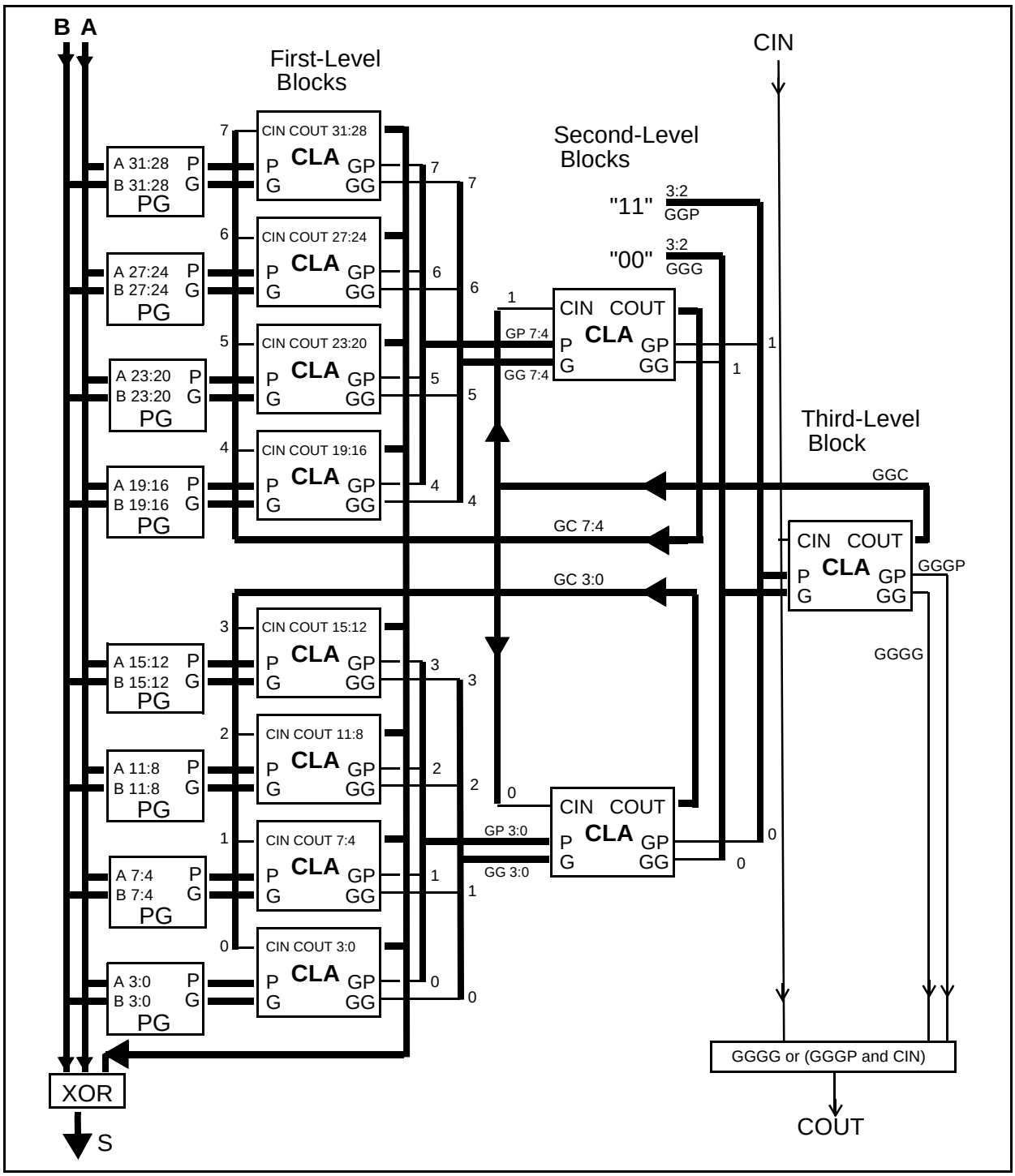
The carry values are computed by a three-level tree of 4-bit carry-lookahead blocks.

- The first level of the tree computes the 32 carry values and the eight group-propagate and generate values. Each of the first-level group-propagate and generate values tells if that 4-bit slice propagates and generates carry values from the next-lower group to the next-higher group. The first-level lookahead blocks read the group carry computed at the second level.

- The second-level lookahead blocks read the group-propagate and generate information from the four first-level blocks and then compute their own group-propagate and generate information. The second-level lookahead blocks also read group carry information computed at the third level to compute the carries for each of the third-level blocks.
- The third-level block reads the propagate and generate information of the second level to compute a propagate and generate value for the entire adder. It also reads the external carry to compute each second-level carry. The carry-out for the adder is ’1’ if the third-level generate is ’1’ or if the third-level propagate is ’1’ and the external carry is ’1’.

  The third-level carry-lookahead block is capable of processing four second-level blocks. But because there are only two second-level blocks, the high-order 2 bits of the computed carry are ignored; the high-order two bits of the generate input to the third-level are set to zero, ”00”; and the propagate high-order bits are set to ”11”. These settings cause the unused portion to propagate carries but not to generate them. Figure A-15 shows the overall structure for the carry-lookahead adder.

Figure A-15 Carry-Lookahead Adder Block Diagram



The VHDL implementation of the design in Figure A-15 is accomplished with four procedures:

CLA
: Names a 4-bit carry-lookahead block.

PG
: Computes first-level propagate and generate information.

SUM
: Computes the sum by adding the XOR values to the inputs with the carry values computed by CLA.

BITSLICE
: Collects the first-level CLA blocks, the PG computations, and the SUM. This procedure performs all the work for a 4-bit value except for the second- and third-level lookaheads.

Example A-12 shows a VHDL description of the adder.

Example A-12 Carry-Lookahead Adder

```
package LOCAL is
  constant N:   INTEGER := 4;

  procedure BITSLICE(
      A, B: in BIT_VECTOR(3 downto 0);
      CIN: in BIT;
      signal S: out BIT_VECTOR(3 downto 0);
      signal GP, GG: out BIT);
  procedure PG(
      A, B: in BIT_VECTOR(3 downto 0);
      P, G: out BIT_VECTOR(3 downto 0));
  function SUM(A, B, C: BIT_VECTOR(3 downto 0))
      return BIT_VECTOR;
  procedure CLA(
      P, G: in BIT_VECTOR(3 downto 0);
      CIN: in BIT;
      C: out BIT_VECTOR(3 downto 0);
      signal GP, GG: out BIT);
end LOCAL;
```

```
package body LOCAL is
  ------------------------------------------------
  -- Compute sum and group outputs from a, b, cin
  ------------------------------------------------

  procedure BITSLICE(
      A, B: in BIT_VECTOR(3 downto 0);
      CIN: in BIT;
      signal S: out BIT_VECTOR(3 downto 0);
      signal GP, GG: out BIT) is

    variable P, G, C: BIT_VECTOR(3 downto 0);
  begin
    PG(A, B, P, G);
    CLA(P, G, CIN, C, GP, GG);
    S <= SUM(A, B, C);
  end;

  ------------------------------------------------
  -- Compute propagate and generate from input bits
  ------------------------------------------------

  procedure PG(A, B: in BIT_VECTOR(3 downto 0);
               P, G: out BIT_VECTOR(3 downto 0)) is

  begin
    P <= A or B;
    G <= A and B;
  end;

  ------------------------------------------------
  -- Compute sum from the input bits and the carries
  ------------------------------------------------

  function SUM(A, B, C: BIT_VECTOR(3 downto 0))
      return BIT_VECTOR is

  begin
    return(A xor B xor C);
  end;

  ------------------------------
  -- 4-bit carry-lookahead block
  ------------------------------

  procedure CLA(
      P, G: in BIT_VECTOR(3 downto 0);
      CIN: in BIT;
      C: out BIT_VECTOR(3 downto 0);
```

```
      signal GP, GG: out BIT) is
    variable TEMP_GP, TEMP_GG, LAST_C: BIT;
  begin
    TEMP_GP <= P(0);
    TEMP_GG <= G(0);
    LAST_C <= CIN;
    C(0) <= CIN;

    for I in 1 to N-1 loop
      TEMP_GP <= TEMP_GP and P(I);
      TEMP_GG <= (TEMP_GG and P(I)) or G(I);
      LAST_C <= (LAST_C and P(I-1)) or G(I-1);
      C(I) <= LAST_C;
    end loop;

    GP <= TEMP_GP;
    GG <= TEMP_GG;
  end;
end LOCAL;

use WORK.LOCAL.ALL;

---------------------------------
-- A 32-bit carry-lookahead adder
---------------------------------

entity ADDER is
  port(A, B: in BIT_VECTOR(31 downto 0);
       CIN: in BIT;
       S: out BIT_VECTOR(31 downto 0);
       COUT: out BIT);
end ADDER;
architecture BEHAVIOR of ADDER is

  signal GG,GP,GC: BIT_VECTOR(7 downto 0);
    -- First-level generate, propagate, carry
  signal GGG, GGP, GGC: BIT_VECTOR(3 downto 0);
    -- Second-level gen, prop, carry
  signal GGGG, GGGP: BIT;
    -- Third-level gen, prop

begin
  -- Compute Sum and 1st-level Generate and Propagate
  -- Use input data and the 1st-level Carries computed
  -- later.
  BITSLICE(A( 3 downto  0),B( 3 downto  0),GC(0),
           S( 3 downto  0),GP(0), GG(0));
  BITSLICE(A( 7 downto  4),B( 7 downto  4),GC(1),
           S( 7 downto  4),GP(1), GG(1));
```

```
BITSLICE(A(11 downto  8),B(11 downto  8),GC(2),
         S(11 downto  8),GP(2), GG(2));
BITSLICE(A(15 downto 12),B(15 downto 12),GC(3),
         S(15 downto 12),GP(3), GG(3));
BITSLICE(A(19 downto 16),B(19 downto 16),GC(4),
         S(19 downto 16),GP(4), GG(4));
BITSLICE(A(23 downto 20),B(23 downto 20),GC(5),
         S(23 downto 20),GP(5), GG(5));
BITSLICE(A(27 downto 24),B(27 downto 24),GC(6),
         S(27 downto 24),GP(6), GG(6));
BITSLICE(A(31 downto 28),B(31 downto 28),GC(7),
         S(31 downto 28),GP(7), GG(7));

-- Compute first-level Carries and second-level
-- generate and propagate.
-- Use first-level Generate, Propagate, and
-- second-level carry.
process(GP, GG, GGC)
  variable TEMP: BIT_VECTOR(3 downto 0);
begin
  CLA(GP(3 downto 0), GG(3 downto 0), GGC(0), TEMP,
      GGP(0), GGG(0));
  GC(3 downto 0) <= TEMP;
end process;

process(GP, GG, GGC)
  variable TEMP: BIT_VECTOR(3 downto 0);
begin
  CLA(GP(7 downto 4), GG(7 downto 4), GGC(1), TEMP,
      GGP(1), GGG(1));
  GC(7 downto 4) <= TEMP;
end process;

-- Compute second-level Carry and third-level
--   Generate and Propagate
-- Use second-level Generate, Propagate and Carry-in
--   (CIN)
process(GGP, GGG, CIN)
  variable TEMP: BIT_VECTOR(3 downto 0);
begin
  CLA(GGP, GGG, CIN, TEMP, GGGP, GGGG);
  GGC <= TEMP;
end process;

-- Assign unused bits of second-level Generate and
--   Propagate
GGP(3 downto 2) <= ”11”;
GGG(3 downto 2) <= ”00”;
```

```
  -- Compute Carry-out (COUT)
  -- Use third-level Generate and Propagate and
  --   Carry-in (CIN).
  COUT <= GGGG or (GGGP and CIN);
end BEHAVIOR;
```

Implementation

In the carry-lookahead adder implementation, procedures perform the computation of the design. The procedures can also be in the form of separate entities and used by component instantiation, producing a hierarchical design. FPGA Compiler II / FPGA *Express* does not collapse a hierarchy of entities, but it does collapse the procedure call hierarchy into one design.

The keyword signal is included before some of the interface parameter declarations. This keyword is required for the out formal parameters when the actual parameters must be signals.

The output parameter C from the CLA procedure is not declared as a signal; thus, it is not allowed in a concurrent procedure call. Only signals can be used in such calls. To overcome this problem, subprocesses are used, declaring a temporary variable TEMP. TEMP receives the value of the C parameter and assigns it to the appropriate signal (a generally useful technique).

Serial-to-Parallel Converter—Counting Bits

This example shows the design of a serial-to-parallel converter that reads a serial, bit-stream input and produces an 8-bit output.

The design reads the following inputs:

SERIAL_IN
: The serial input data.

RESET
: The input that, when it is ’1’, causes the converter to reset. All outputs are set to 0, and the converter is prepared to read the next serial word.

CLOCK
: The value of RESET and SERIAL_IN, which is read on the positive transition of this clock. Outputs of the converter are also valid only on positive transitions.

The design produces the following outputs:

PARALLEL_OUT
: The 8-bit value read from the SERIAL_IN port.

READ_ENABLE
: The output that, when it is ’1’ on the positive transition of CLOCK, causes the data on PARALLEL_OUT to be read.

PARITY_ERROR
: The output that, when it is ’1’ on the positive transition of CLOCK, indicates that a parity error has been detected on the SERIAL_IN port. When a parity error is detected, the converter halts until restarted by the RESET port.

Input Format

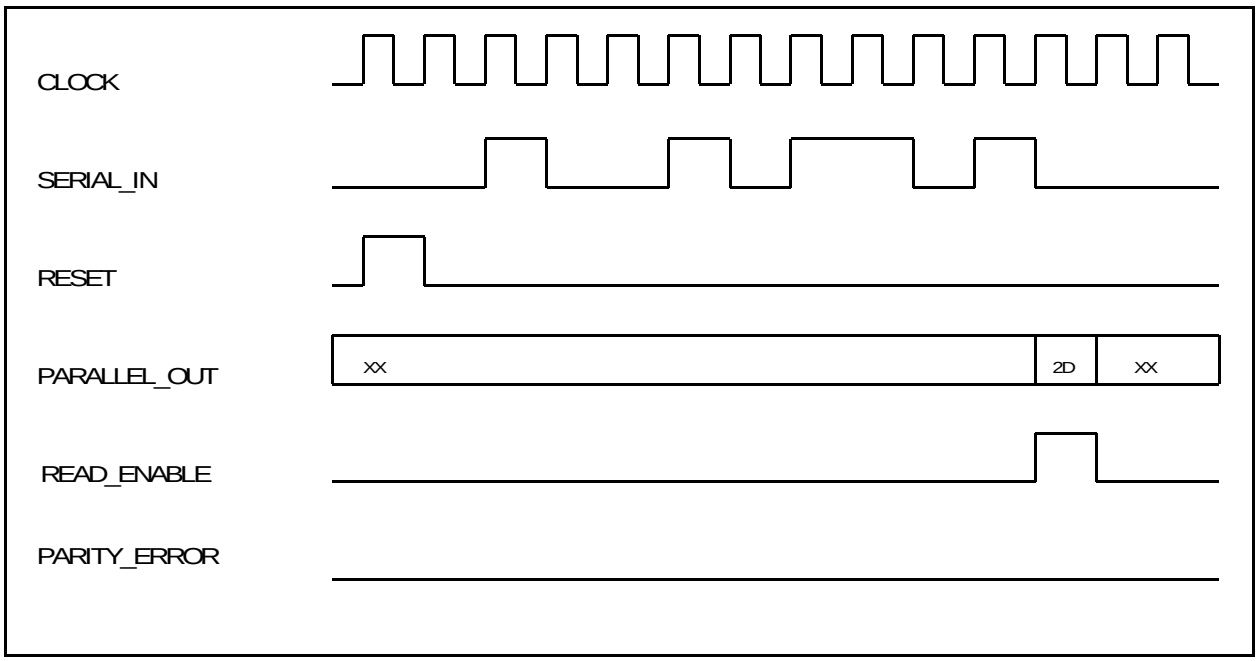
When no data is being transmitted to the serial port, keep it at a value of ’0’. Each 8-bit value requires ten clock cycles to read it. On the eleventh clock cycle, the parallel output value can be read.

In the first cycle, a ’1’ is placed on the serial input. This assignment indicates that an 8-bit value follows. The next eight cycles transmit each bit of the value. The most significant bit is transmitted first. The tenth cycle transmits the parity of the 8-bit value. It must be ’0’ if an even number of ’1’ values are in the 8-bit data, and ’1’ otherwise. If the converter detects a parity error, it sets the PARITY_ERROR output to ’1’ and waits until the value is reset.

On the eleventh cycle, the READ_ENABLE output is set to ’1’ and the 8-bit value can be read from the PARALLEL_OUT port. If the SERIAL_IN port has a ’1’ on the eleventh cycle, another 8-bit value is read immediately; otherwise, the converter waits until SERIAL_IN goes to ’1’.

Figure A-16 shows the timing of this design.

Figure A-16 Sample Waveform Through the Converter



Implementation Details

The implementation of the converter is as a four-state finite-state machine with synchronous reset. When a reset is detected, the converter enters a WAIT_FOR_START state. The description of each state follows

WAIT_FOR_START

Stay in this state until a ’1’ is detected on the serial input. When a ’1’ is detected, clear the PARALLEL_OUT registers and go to the READ_BITS state.

READ_BITS

If the value of the current_bit_position counter is 8, all 8 bits have been read. Check the computed parity with the transmitted parity. If it is correct, go to the ALLOW_READ state; otherwise, go to the PARITY_ERROR state.

If all 8 bits have not yet been read, set the appropriate bit in the PARALLEL_OUT buffer to the SERIAL_IN value, compute the parity of the bits read so far, and increment the current_bit_position.

ALLOW_READ
: This is the state where the outside world reads the PARALLEL_OUT value. When that value is read, the design returns to the WAIT_FOR_START state.

PARITY_ERROR_DETECTED
: In this state, the PARITY_ERROR output is set to ’1’ and nothing else is done.

This design has four values stored in registers:

CURRENT_STATE
: Remembers the state as of the last clock edge.

CURRENT_BIT_POSITION
: Remembers how many bits have been read so far.

CURRENT_PARITY
: Keeps a running XOR of the bits read.

CURRENT_PARALLEL_OUT
: Stores each parallel bit as it is found.

The design has two processes: the combinational NEXT_ST containing the combinational logic and the sequential SYNCH that is clocked.

NEXT_ST performs all the computations and state assignments. The NEXT_ST process starts by assigning default values to all the signals it drives. This assignment guarantees that all signals are driven under all conditions. Next, the RESET input is processed. If RESET is not active, a case statement determines the current state and its computations. State transitions are performed by assigning the next state's value you want to the NEXT_STATE signal.

The serial-to-parallel conversion itself is performed by these two statements in the NEXT_ST process:

```
NEXT_PARALLEL_OUT(CURRENT_BIT_POSITION) <= SERIAL_IN;
NEXT_BIT_POSITION <= CURRENT_BIT_POSITION + 1;
```

The first statement assigns the current serial input bit to a particular bit of the parallel output. The second statement increments the next bit position to be assigned.

SYNCH registers and updates the stored values previously described. Each registered signal has two parts, NEXT_... and CURRENT_... :

NEXT_...
: Signals hold values computed by the NEXT_ST process.

CURRENT_...
: Signals hold the values driven by the SYNCH process. The CURRENT_... signals hold the values of the NEXT_... signals as of the last clock edge.

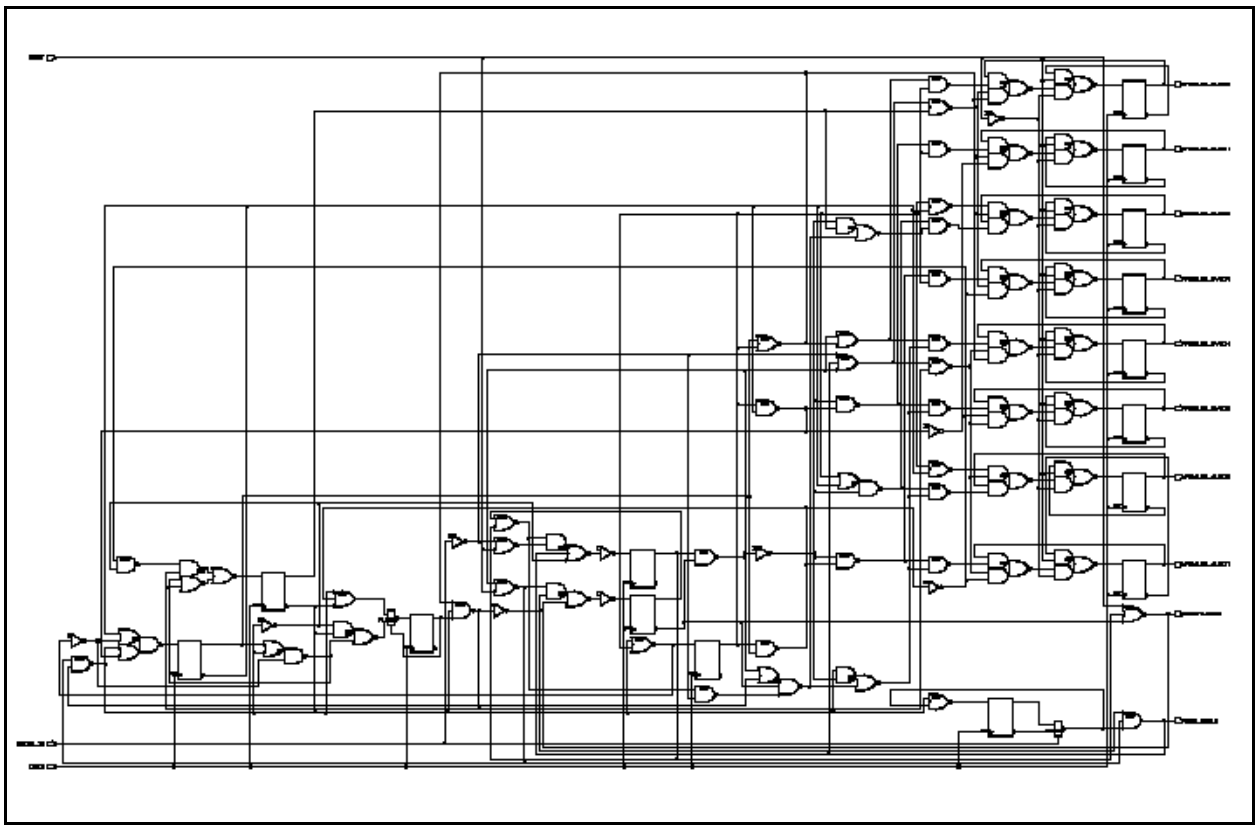
Example A-13 shows a VHDL description of the converter.

Example A-13 Serial-to-Parallel Converter—Counting Bits

```
-- Serial-to-Parallel Converter, counting bits

package TYPES is
  -- Declares types used in the rest of the design
  type STATE_TYPE is (WAIT_FOR_START,
                      READ_BITS,
                      PARITY_ERROR_DETECTED,
                      ALLOW_READ);
  constant PARALLEL_BIT_COUNT: INTEGER := 8;
  subtype PARALLEL_RANGE is INTEGER
      range 0 to (PARALLEL_BIT_COUNT-1);
  subtype PARALLEL_TYPE is BIT_VECTOR(PARALLEL_RANGE);
end TYPES;

use WORK.TYPES.ALL;     -- Use the TYPES package

entity SER_PAR is       -- Declare the interface
  port(SERIAL_IN, CLOCK, RESET: in BIT;
       PARALLEL_OUT: out PARALLEL_TYPE;
       PARITY_ERROR, READ_ENABLE: out BIT);
end SER_PAR;

architecture BEHAVIOR of SER_PAR is
  -- Signals for stored values
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
  signal CURRENT_PARITY, NEXT_PARITY: BIT;
  signal CURRENT_BIT_POSITION, NEXT_BIT_POSITION:
      INTEGER range PARALLEL_BIT_COUNT downto 0;
  signal CURRENT_PARALLEL_OUT, NEXT_PARALLEL_OUT:
      PARALLEL_TYPE;
begin
NEXT_ST: process(SERIAL_IN, CURRENT_STATE, RESET,
                 CURRENT_BIT_POSITION, CURRENT_PARITY,
                 CURRENT_PARALLEL_OUT)
  -- This process computes all outputs, the next
  --   state, and the next value of all stored values
  begin
    PARITY_ERROR <= '0'; -- Default values for all
    READ_ENABLE <= '0';  --  outputs and stored values
    NEXT_STATE <= CURRENT_STATE;
    NEXT_BIT_POSITION <= 0;
    NEXT_PARITY <= '0';
    NEXT_PARALLEL_OUT <= CURRENT_PARALLEL_OUT;

    if (RESET = '1') then      -- Synchronous reset
      NEXT_STATE <= WAIT_FOR_START;
    else
```

```
      case CURRENT_STATE is   -- State processing
        when WAIT_FOR_START =>
          if (SERIAL_IN = '1') then
            NEXT_STATE <= READ_BITS;
            NEXT_PARALLEL_OUT <=
                PARALLEL_TYPE'(others=>'0');
          end if;
        when READ_BITS =>
          if (CURRENT_BIT_POSITION =
              PARALLEL_BIT_COUNT) then
            if (CURRENT_PARITY = SERIAL_IN) then
              NEXT_STATE <= ALLOW_READ;
              READ_ENABLE <= '1';
            else
              NEXT_STATE <= PARITY_ERROR_DETECTED;
            end if;
          else
            NEXT_PARALLEL_OUT(CURRENT_BIT_POSITION) <=
                SERIAL_IN;
            NEXT_BIT_POSITION <=
                CURRENT_BIT_POSITION + 1;
            NEXT_PARITY <= CURRENT_PARITY xor
                           SERIAL_IN;
          end if;
        when PARITY_ERROR_DETECTED =>
          PARITY_ERROR <= '1';
        when ALLOW_READ =>
          NEXT_STATE <= WAIT_FOR_START;
      end case;
    end if;
  end process NEXT_ST;

  SYNCH: process
    -- This process remembers the stored values
    --    across clock cycles
  begin
    wait until CLOCK'event and CLOCK = '1';
    CURRENT_STATE <= NEXT_STATE;
    CURRENT_BIT_POSITION <= NEXT_BIT_POSITION;
    CURRENT_PARITY <= NEXT_PARITY;
    CURRENT_PARALLEL_OUT <= NEXT_PARALLEL_OUT;
  end process SYNCH;

  PARALLEL_OUT <= CURRENT_PARALLEL_OUT;

end BEHAVIOR;
```

Figure A-17 Serial-to-Parallel Converter—Counting Bits Schematic



Serial-to-Parallel Converter—Shifting Bits

This example describes another implementation of the serial-to-parallel converter in the last example. This design performs the same function as the previous one but uses a different algorithm to do the conversion.

The previous implementation used a counter to indicate the bit of the output that was set when a new serial bit was read. In this implementation, the serial bits are shifted into place. Before the conversion occurs, a ’1’ is placed in the least-significant bit position. When that ’1’ is shifted out of the most significant position (position

0), the signal NEXT_HIGH_BIT is set to ’1’ and the conversion is complete.

Example A-14 shows the listing of the second implementation. The differences are highlighted in bold. The differences relate to the removal of the ..._BIT_POSITION signals, the addition of ..._HIGH_BIT signals, and the change in the way NEXT_PARALLEL_OUT is computed.

Example A-14 Serial-to-Parallel Converter—Shifting Bits

```
package TYPES is
  -- Declares types used in the rest of the design
  type STATE_TYPE is (WAIT_FOR_START,
                      READ_BITS,
                      PARITY_ERROR_DETECTED,
                      ALLOW_READ);
  constant PARALLEL_BIT_COUNT: INTEGER := 8;
  subtype PARALLEL_RANGE is INTEGER
      range 0 to (PARALLEL_BIT_COUNT-1);
  subtype PARALLEL_TYPE is BIT_VECTOR(PARALLEL_RANGE);
end TYPES;

use WORK.TYPES.ALL;      -- Use the TYPES package

entity SER_PAR is        -- Declare the interface
  port(SERIAL_IN, CLOCK, RESET: in BIT;
       PARALLEL_OUT: out PARALLEL_TYPE;
       PARITY_ERROR, READ_ENABLE: out BIT);
end SER_PAR;

architecture BEHAVIOR of SER_PAR is
  -- Signals for stored values
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;

  signal CURRENT_PARITY, NEXT_PARITY: BIT;
  signal CURRENT_HIGH_BIT, NEXT_HIGH_BIT: BIT;
  signal CURRENT_PARALLEL_OUT, NEXT_PARALLEL_OUT:
     PARALLEL_TYPE;
begin

NEXT_ST: process(SERIAL_IN, CURRENT_STATE, RESET,
                 CURRENT_HIGH_BIT, CURRENT_PARITY,
                 CURRENT_PARALLEL_OUT)
  -- This process computes all outputs, the next
```

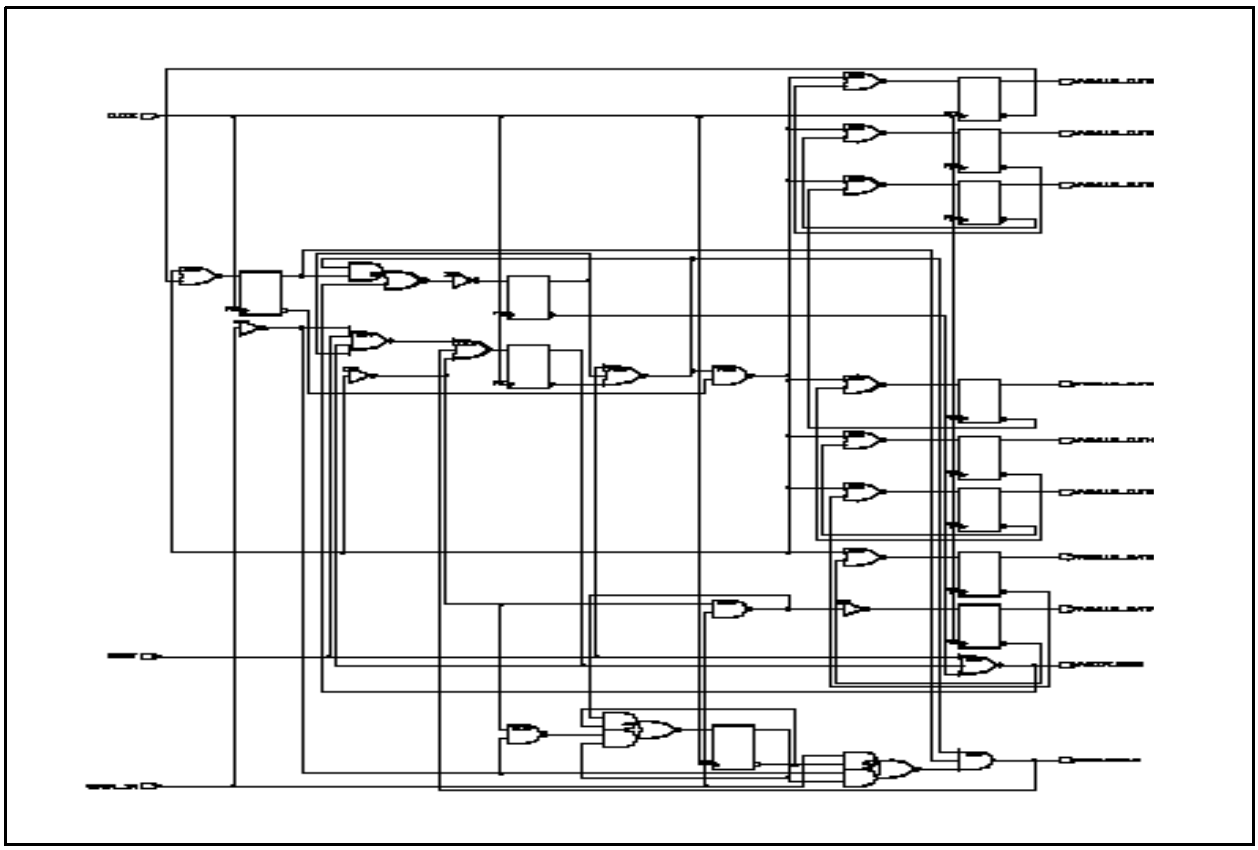
```
--   state, and the next value of all stored values
begin
  PARITY_ERROR <= '0'; -- Default values for all
  READ_ENABLE <= '0';  --  outputs and stored values
  NEXT_STATE <= CURRENT_STATE;
  NEXT_HIGH_BIT <= '0';
  NEXT_PARITY <= '0';
  NEXT_PARALLEL_OUT <= PARALLEL_TYPE'(others=>'0');
  if(RESET = '1') then      -- Synchronous reset
    NEXT_STATE <= WAIT_FOR_START;
  else
    case CURRENT_STATE is   -- State processing
      when WAIT_FOR_START =>
        if (SERIAL_IN = '1') then
          NEXT_STATE <= READ_BITS;
          NEXT_PARALLEL_OUT <=
              PARALLEL_TYPE'(others=>'0');
        end if;
      when READ_BITS =>
        if (CURRENT_HIGH_BIT = '1') then
          if (CURRENT_PARITY = SERIAL_IN) then
            NEXT_STATE <= ALLOW_READ;
            READ_ENABLE <= '1';
          else
            NEXT_STATE <= PARITY_ERROR_DETECTED;
          end if;
        else
          NEXT_HIGH_BIT <= CURRENT_PARALLEL_OUT(0);
          NEXT_PARALLEL_OUT <=
              CURRENT_PARALLEL_OUT(
                  1 to PARALLEL_BIT_COUNT-1) &
              SERIAL_IN;
          NEXT_PARITY <= CURRENT_PARITY xor
                         SERIAL_IN;
        end if;
      when PARITY_ERROR_DETECTED =>
        PARITY_ERROR <= '1';
      when ALLOW_READ =>
        NEXT_STATE <= WAIT_FOR_START;
    end case;
  end if;
end process NEXT_ST;

SYNCH: process
  -- This process remembers the stored values
  --    across clock cycles
begin
  wait until CLOCK'event and CLOCK = '1';
```

```
    CURRENT_STATE <= NEXT_STATE;
    CURRENT_HIGH_BIT <= NEXT_HIGH_BIT;
    CURRENT_PARITY <= NEXT_PARITY;
    CURRENT_PARALLEL_OUT <= NEXT_PARALLEL_OUT;
  end process SYNCH;

  PARALLEL_OUT <= CURRENT_PARALLEL_OUT;

end BEHAVIOR;
```

Note:

The synthesized schematic for the shifter implementation is much simpler than that of the previous count implementation in Example A-13. It is simpler because the shifter algorithm is inherently easier to implement.

Figure A-18 Serial-to-Parallel Converter—Shifting Bits Schematic



With the count algorithm, each of the flip-flops holding the PARALLEL_OUT bits needed logic that decoded the value stored in the BIT_POSITION flip-flops to see when to route in the value of SERIAL_IN. Also, the BIT_POSITION flip-flops needed an incrementer to compute their next value.

In contrast, the shifter algorithm requires neither an incrementer nor flip-flops to hold BIT_POSITION. Additionally, the logic in front of most PARALLEL_OUT bits needs to read only the value of the previous flip-flop or '0'. The value depends on whether bits are currently being read. In the shifter algorithm, the SERIAL_IN port needs to be connected only to the least significant bit (number 7) of the PARALLEL_OUT flip-flops.

These two implementations illustrate the importance of designing efficient algorithms. Both work properly, but the shifter algorithm produces a faster, more area-efficient design.

Programmable Logic Arrays

This example shows a way to build programmable logic arrays (PLAs) in VHDL. The PLA function uses an input lookup vector as an index into a constant PLA table and then returns the output vector specified by the PLA.

The PLA table is an array of PLA rows, where each row is an array of PLA elements. Each element is either a one, a zero, a minus, or a space ('1', '0', '–', or ' '). The table is split between an input plane and an output plane. The input plane is specified by zeros, ones, and minuses. The output plane is specified by zeros and ones. The two planes' values are separated by a space.

In the PLA function, the output vector is first initialized to be all zeros. When the input vector matches an input plane in a row of the PLA table, the ones in the output plane are assigned to the corresponding bits in the output vector. A match is determined as follows:

- If a zero or one is in the input plane, the input vector must have the same value in the same position.
- If a minus is in the input plane, it matches any input vector value at that position.

The generic PLA table types and the PLA function are defined in a package named LOCAL. An entity PLA_VHDL that uses LOCAL needs only to specify its PLA table as a constant, then call the PLA function.

The PLA function does not explicitly depend on the size of the PLA. To change the size of the PLA, change the initialization of the TABLE constant and the initialization of the constants INPUT_COUNT, OUTPUT_COUNT, and ROW_COUNT. In Example A-15, these constants are initialized to a PLA equivalent to the ROM shown previously (Example A-3). Accordingly, the synthesized schematic is the same as that of the ROM, with one difference: in Example A-3, the DATA output port range is 1 to 5; in Example A-15, the OUT_VECTOR output port range is 4 down to 0.

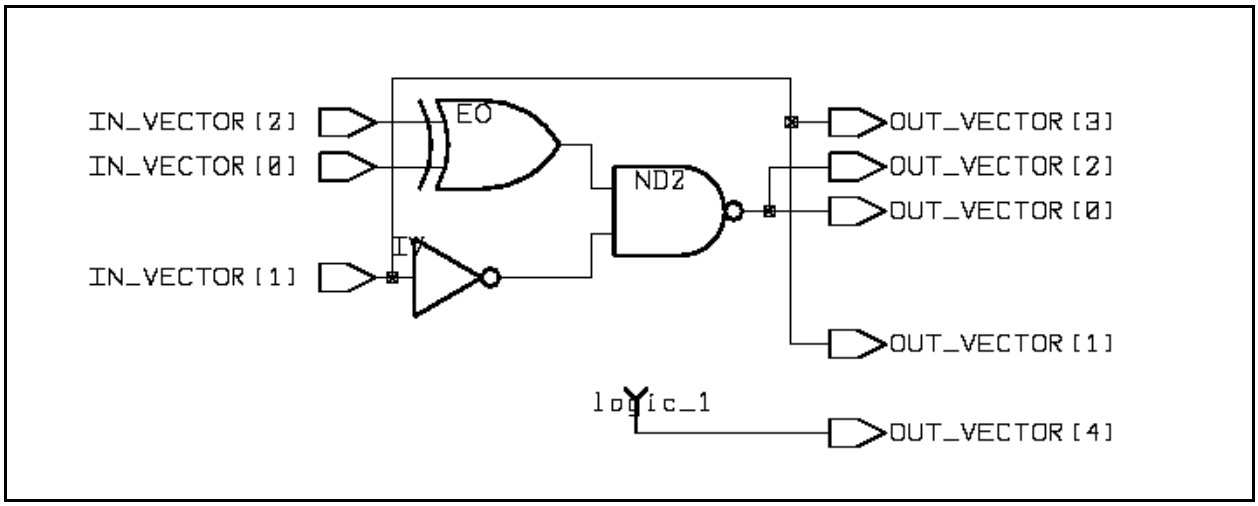
Example A-15 Programmable Logic Array

```
package LOCAL is
  constant INPUT_COUNT: INTEGER := 3;
  constant OUTPUT_COUNT: INTEGER := 5;
  constant ROW_COUNT: INTEGER := 6;
  constant ROW_SIZE: INTEGER := INPUT_COUNT +
                                OUTPUT_COUNT + 1;
  type PLA_ELEMENT is ('1', '0', '-', ' ');
  type PLA_VECTOR is
      array (INTEGER range <>) of PLA_ELEMENT;
  subtype PLA_ROW is
      PLA_VECTOR(ROW_SIZE - 1 downto 0);
  subtype PLA_OUTPUT is
      PLA_VECTOR(OUTPUT_COUNT - 1 downto 0);
  type PLA_TABLE is
      array(ROW_COUNT - 1 downto 0) of PLA_ROW;

  function PLA(IN_VECTOR: BIT_VECTOR;
               TABLE: PLA_TABLE)
      return BIT_VECTOR;
end LOCAL;

package body LOCAL is

  function PLA(IN_VECTOR: BIT_VECTOR;
               TABLE: PLA_TABLE)
      return BIT_VECTOR is
    subtype RESULT_TYPE is
        BIT_VECTOR(OUTPUT_COUNT - 1 downto 0);
    variable RESULT: RESULT_TYPE;
    variable ROW: PLA_ROW;
    variable MATCH: BOOLEAN;
    variable IN_POS: INTEGER;

  begin
    RESULT <= RESULT_TYPE'(others => BIT'( '0' ));
    for I in TABLE'range loop
      ROW <= TABLE(I);
      MATCH <= TRUE;
      IN_POS <= IN_VECTOR'left;
```

```
        -- Check for match in input plane
        for J in ROW_SIZE - 1 downto OUTPUT_COUNT loop
          if(ROW(J) = PLA_ELEMENT'( '1' )) then
            MATCH <= MATCH and
                     (IN_VECTOR(IN_POS) = BIT'( '1' ));
          elsif(ROW(J) = PLA_ELEMENT'( '0' )) then
            MATCH <= MATCH and
                     (IN_VECTOR(IN_POS) = BIT'( '0' ));
          else
            null;     -- Must be minus ("don't care")
          end if;
          IN_POS <= IN_POS - 1;
        end loop;

        -- Set output plane
        if(MATCH) then
          for J in RESULT'range loop
            if(ROW(J) = PLA_ELEMENT'( '1' )) then
              RESULT(J) <= BIT'( '1' );
            end if;
          end loop;
        end if;
      end loop;
      return(RESULT);
    end;
  end LOCAL;

  use WORK.LOCAL.all;
  entity PLA_VHDL is
    port(IN_VECTOR: BIT_VECTOR(2 downto 0);
         OUT_VECTOR: out BIT_VECTOR(4 downto 0));
  end PLA_VHDL;

  architecture BEHAVIOR of PLA_VHDL is
    constant TABLE: PLA_TABLE := PLA_TABLE'(
         PLA_ROW'("--- 10000"),
         PLA_ROW'("-1- 01000"),
         PLA_ROW'("0-0 00101"),
         PLA_ROW'("-1- 00101"),
         PLA_ROW'("1-1 00101"),
         PLA_ROW'("-1- 00010"));
```

```
begin
  OUT_VECTOR <= PLA(IN_VECTOR, TABLE);
end BEHAVIOR;
```

Figure A-19 Programmable Logic Array Schematic



B

Synopsys Packages

The following Synopsys packages are included with this release:

- std_logic_1164 Package

  Defines a standard for designers to use in describing the interconnection data types used in VHDL modeling.

- std_logic_arith Package

  Provides a set of arithmetic, conversion, and comparison functions for SIGNED, UNSIGNED, INTEGER, STD_ULOGIC, STD_LOGIC, and STD_LOGIC_VECTOR types.

- numeric_std Package

  The numeric_std package is an alternative to the std_logic_arith package. It is the IEEE standard 1076.3-1997, and documentation about it is available from IEEE. For more information, see “numeric_std Package” on page B-20.

- std_logic_misc Package

  Defines supplemental types, subtypes, constants, and functions for the std_logic_1164 package.

- ATTRIBUTES Package

  Declares synthesis attributes and the resource sharing subtype and its attributes.

std_logic_1164 Package

The std_logic_1164 package defines the IEEE standard for designers to use in describing the interconnection data types used in VHDL modeling. The logic system defined in this package might be insufficient for modeling switched transistors, because such a requirement is out of the scope of this package. Furthermore, mathematics, primitives, and timing standards are considered orthogonal issues as they relate to this package and are, therefore, beyond its scope.

The std_logic_1164 package file has been updated with Synopsys synthesis directives.

To use this package in a VHDL source file, include the following lines at the beginning of the source file:

```
library IEEE;
use IEEE.std_logic_1164.all;
```

When you analyze your VHDL source, FPGA Compiler II / FPGA *Express* automatically finds the IEEE library and the std_logic_1164 package. However, you must analyze those use packages that are not in the IEEE and Synopsys libraries before processing a source file that uses them.

std_logic_arith Package

Functions defined in the std_logic_arith package provide conversion to and from the predefined VHDL data type INTEGER, arithmetic, comparison, and BOOLEAN operations. This package lets you perform arithmetic operations and numeric comparisons on array data types. The package defines some arithmetic operators (+, -, *, ABS) and the relational operators (<, >, <=, >=, =, /=). (IEEE VHDL does not define arithmetic operators for arrays and defines the comparison operators in a manner inconsistent with an arithmetic interpretation of array values.)

The package also defines two major data types of its own: UNSIGNED and SIGNED (see “Data Types” on page B-6 for details). The std_logic_arith package is legal VHDL; you can use it for both synthesis and simulation.

You can configure the std_logic_arith package to work on any array of single-bit types. You encode single-bit types in 1 bit with the ENUM_ENCODING attribute.

You can make the vector type (for example, std_logic_vector) synonymous with either SIGNED or UNSIGNED. This way, if you plan to use mostly UNSIGNED numbers, you do not need to convert your vector type to call UNSIGNED functions. The disadvantage of making your vector type synonymous with either UNSIGNED or SIGNED is that it causes redefinition of the standard VHDL comparison functions (=, /=, <, >, <=, >=).

Table B-1 shows that the standard comparison functions for BIT_VECTOR do not match the SIGNED and UNSIGNED functions.

Table B-1 UNSIGNED, SIGNED, and BIT_VECTOR Comparison Functions

ARG1	op	ARG2	UNSIGNED	SIGNED	BIT_VECTOR
”000”	=	”000”	true	true	true
”00”	=	”000”	true	true	false
”100”	=	”0100”	true	false	false
”000”	<	”000”	false	false	false
”00”	<	”000”	false	false	true
”100”	<	”0100”	false	true	false

Using the Package

To use the std_logic_arith package in a VHDL source file, include the following lines at the beginning of the source file:

```
library IEEE;
use IEEE.std_logic_arith.all;
```

Modifying the Package

The std_logic_arith package is written in standard VHDL. You can modify or add to it. The appropriate hardware is then synthesized.

For example, to convert a vector of multivalued logic to an INTEGER, you can write the function shown in Example B-1. This MVL_TO_INTEGER function returns the integer value corresponding to the vector when the vector is interpreted as an unsigned (natural) number. If unknown values are in the vector, the return value is –1.

Example B-1 New Function Based on a std_logic_arith Package Function

```
library IEEE;
use IEEE.std_logic_1164.all;

function MVL_TO_INTEGER(ARG : MVL_VECTOR)
  return INTEGER is
  -- pragma built_in SYN_FEED_THRU
  variable uns: UNSIGNED (ARG'range);
begin
    for i in ARG'range loop
        case ARG(i) is
            when '0' | 'L' => uns(i) := '0';
            when '1' | 'H' => uns(i) := '1';
            when others    => return -1;
        end case;
    end loop;
    return CONV_INTEGER(uns);
end MLV_TO_INTEGER;
```

Note the use of the CONV_INTEGER function in Example B-1.

FPGA Compiler II / FPGA *Express* performs almost all synthesis directly from the VHDL descriptions. However, several functions are hard-wired for efficiency. They can be identified by the following comment in their declarations:

```
-- pragma built_in
```

This statement marks functions as special, causing the body of the function to be ignored. Modifying the body does not change the synthesized logic unless you remove the built_in comment. If you want new functionality, write it by using the built_in functions; this is more efficient than removing the built_in function and modifying the body of the function.

Data Types

The std_logic_arith package defines two data types: UNSIGNED and SIGNED.

```
type UNSIGNED is array (natural range <>) of std_logic;
type SIGNED is array (natural range <>) of std_logic;
```

These data types are similar to the predefined VHDL type BIT_VECTOR, but the std_logic_arith package defines the interpretation of variables and signals of these types as numeric values.

UNSIGNED

The UNSIGNED data type represents an unsigned numeric value. FPGA Compiler II / FPGA *Express* interprets the number as a binary representation, with the farthest-left bit being most significant. For example, the decimal number 8 can be represented as

```
UNSIGNED'("1000")
```

When you declare variables or signals of type UNSIGNED, a larger vector holds a larger number. A 4-bit variable holds values up to decimal 15, an 8-bit variable holds values up to 255, and so on. By definition, negative numbers cannot be represented in an UNSIGNED variable. Zero is the smallest value that can be represented.

Example B-2 illustrates some UNSIGNED declarations. The most significant bit is the farthest-left array bound, rather than the high or low range value.

Example B-2 UNSIGNED Declarations

```
variable VAR: UNSIGNED (1 to 10);
  -- 11-bit number
  -- VAR(VAR'left) = VAR(1) is the most significant bit

signal SIG: UNSIGNED (5 downto 0);
  -- 6-bit number
  -- SIG(SIG'left) = SIG(5) is the most significant bit
```

SIGNED

The SIGNED data type represents a signed numeric value. FPGA Compiler II / FPGA *Express* interprets the number as a 2's-complement binary representation, with the farthest-left bit as the sign bit. For example, you can represent decimal 5 and –5 as

```
SIGNED'("0101")  -- represents +5
SIGNED'("1011")  -- represents -5
```

When you declare SIGNED variables or signals, a larger vector holds a larger number. A 4-bit variable holds values from –8 to 7; an 8-bit variable holds values from –128 to 127. A SIGNED value cannot hold as large a value as an UNSIGNED value with the same bit-width.

Example B-3 shows some SIGNED declarations. The sign bit is the farthest-left bit, rather than the highest or lowest.

Example B-3 SIGNED Declarations

```
variable S_VAR: SIGNED (1 to 10);
  -- 11-bit number
  -- S_VAR(S_VAR'left) = S_VAR(1) is the sign bit

signal S_SIG: SIGNED (5 downto 0);
  -- 6-bit number
  -- S_SIG(S_SIG'left) = S_SIG(5) is the sign bit
```

Conversion Functions

The std_logic_arith package provides three sets of functions to convert values between its UNSIGNED and SIGNED types and the predefined type INTEGER. This package also provides the std_logic_vector. Example B-4 shows the declarations of these conversion functions, with BIT and BIT_VECTOR types.

Example B-4 Conversion Functions

```
subtype SMALL_INT is INTEGER range 0 to 1;
function CONV_INTEGER(ARG: INTEGER)  return INTEGER;
function CONV_INTEGER(ARG: UNSIGNED) return INTEGER;
function CONV_INTEGER(ARG: SIGNED)   return INTEGER;
function CONV_INTEGER(ARG: STD_ULOGIC) return SMALL_INT;

function CONV_UNSIGNED(ARG: INTEGER;
                       SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: UNSIGNED;
                       SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: SIGNED;
                       SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: STD_ULOGIC;
                       SIZE: INTEGER) return UNSIGNED;
```

```
function CONV_SIGNED(ARG: INTEGER;
                     SIZE: INTEGER)   return SIGNED;
function CONV_SIGNED(ARG: UNSIGNED;
                     SIZE: INTEGER)   return SIGNED;
function CONV_SIGNED(ARG: SIGNED;
                     SIZE: INTEGER)   return SIGNED;
function CONV_SIGNED(ARG: STD_ULOGIC;
                     SIZE: INTEGER)   return SIGNED;

function CONV_STD_LOGIC_VECTOR(ARG: INTEGER;
                   SIZE: INTEGER)   return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: UNSIGNED;
                   SIZE: INTEGER)   return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: SIGNED;
                   SIZE: INTEGER)   return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: STD_ULOGIC;
                   SIZE: INTEGER)   return STD_LOGIC_VECTOR;
```

There are four versions of each conversion function. The VHDL operator overloading mechanism determines the correct version from the function call's argument types.

The CONV_INTEGER functions convert an argument of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an INTEGER return value. The CONV_UNSIGNED and CONV_SIGNED functions convert an argument of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an UNSIGNED or SIGNED return value whose bit width is SIZE.

The CONV_INTEGER functions have a limitation on the size of operands. VHDL defines INTEGER values as being between –2147483647 and 2147483647. This range corresponds to a 31-bit UNSIGNED value or a 32-bit SIGNED value. You cannot convert an argument outside this range to an INTEGER.

The CONV_UNSIGNED and CONV_SIGNED functions each require two operands. The first operand is the value converted. The second operand is an INTEGER that specifies the expected size of the converted result. For example, the following function call returns a 10-bit UNSIGNED value representing the value in sig.

```
ten_unsigned_bits := CONV_UNSIGNED(sig, 10);
```

If the value passed to CONV_UNSIGNED or CONV_SIGNED is smaller than the expected bit-width (such as representing the value 2 in a 24-bit number), the value is bit-extended appropriately. FPGA Compiler II / FPGA *Express* places zeros in the more significant (left) bits for an UNSIGNED return value, and it uses sign extension for a SIGNED return value.

You can use the conversion functions to extend a number's bit-width even if conversion is not required. For example,

```
CONV_SIGNED(SIGNED'("110"), 8) ⇒ "11111110"
```

An UNSIGNED or SIGNED return value is truncated when its bit-width is too small to hold the ARG value. For example,

```
CONV_SIGNED(UNSIGNED'("1101010"), 3) ⇒ "010"
```

Arithmetic Functions

The std_logic_arith package provides arithmetic functions for use with combinations of the Synopsys UNSIGNED and SIGNED data types and the predefined types STD_ULOGIC and INTEGER. These functions produce adders and subtracters.

There are two sets of arithmetic functions: binary functions having two arguments, such as A+B or A*B, and unary functions having one argument, such as –A. Example B-5 and Example B-6 show the declarations for these functions.

Example B-5 Binary Arithmetic Functions

```
function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED;   R: SIGNED)   return SIGNED;
function "+"(L: UNSIGNED; R: SIGNED)   return SIGNED;
function "+"(L: SIGNED;   R: UNSIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: INTEGER)  return UNSIGNED;
function "+"(L: INTEGER;  R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED;   R: INTEGER)  return SIGNED;
function "+"(L: INTEGER;  R: SIGNED)   return SIGNED;
function "+"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
function "+"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED;   R: STD_ULOGIC) return SIGNED;
function "+"(L: STD_ULOGIC; R: SIGNED)   return SIGNED;

function "+"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "+"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "+"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED;   R: SIGNED)   return SIGNED;
function "-"(L: UNSIGNED; R: SIGNED)   return SIGNED;
function "-"(L: SIGNED;   R: UNSIGNED) return SIGNED;
function "-"(L: UNSIGNED; R: INTEGER)  return UNSIGNED;
function "-"(L: INTEGER;  R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED;   R: INTEGER)  return SIGNED;
function "-"(L: INTEGER;  R: SIGNED)   return SIGNED;
function "-"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
function "-"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED;   R: STD_ULOGIC) return SIGNED;
function "-"(L: STD_ULOGIC; R: SIGNED)   return SIGNED;

function "-"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
```

```
function "-"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "-"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "-"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "-"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "-"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;

function "*"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "*"(L: SIGNED;   R: SIGNED)   return SIGNED;
function "*"(L: SIGNED;   R: UNSIGNED) return SIGNED;
function "*"(L: UNSIGNED; R: SIGNED)   return SIGNED;
```

Example B-6 Unary Arithmetic Functions

```
function "+"(L: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED)   return SIGNED;
function "-"(L: SIGNED)   return SIGNED;
function "ABS"(L: SIGNED) return SIGNED;
```

The unary arithmetic functions in Example B-5 and Example B-6 determine the width of their return values, as follows:

1. When only one UNSIGNED or SIGNED argument is present, the width of the return value is the same as that argument's.
2. When both arguments are either UNSIGNED or SIGNED, the width of the return value is the larger of the two argument widths. An exception is that when an UNSIGNED number is added to or subtracted from a SIGNED number that is the same size or smaller, the return value is a SIGNED number 1 bit wider than the UNSIGNED argument. This size guarantees that the return value is large enough to hold any (positive) value of the UNSIGNED argument.

The number of bits returned by + and – is illustrated in Table B-2.

```
signal U4: UNSIGNED (3 downto 0);
signal U8: UNSIGNED (7 downto 0);
signal S4: SIGNED (3 downto 0);
signal S8: SIGNED (7 downto 0);
```

Table B-2 Number of Bits Returned by + and –

+ or -	U4	U8	S4	S8
U4	4	8	5	8
U8	8	8	9	9
S4	5	9	4	8
S8	8	9	8	8

In some circumstances, you might need to obtain a carry-out bit from the + or – operation. To do this, extend the larger operand by 1 bit. The high bit of the return value is the carry, as shown in Example B-7.

Example B-7 Using the Carry-Out Bit

```
process
    variable a, b, sum: UNSIGNED (7 downto 0);
    variable temp: UNSIGNED (8 downto 0);
    variable carry: BIT;
begin
    temp  <= CONV_UNSIGNED(a,9) + b;
    sum   <= temp(7 downto 0);
    carrY <= temp(8);
end process;
```

Comparison Functions

The std_logic_arith package provides functions for comparing UNSIGNED and SIGNED data types with each other and with the predefined type INTEGER. FPGA Compiler II / FPGA *Express* compares the numeric values of the arguments, returning a BOOLEAN value. For example, the following evaluates true:

```
UNSIGNED'("001") > SIGNED'("111")
```

The std_logic_arith comparison functions are similar to the built-in VHDL comparison functions. The only difference is that the std_logic_arith functions accommodate signed numbers and varying bit-widths. The predefined VHDL comparison functions perform bitwise comparisons and do not have the correct semantics for comparing numeric values (see "Relational Operators" on page 4-5).

These functions produce comparators. The function declarations are listed in two groups: ordering functions ("<", "<=", ">", ">="), shown in Example B-8, and equality functions ("=", "/="), shown in Example B-9.

Example B-8 Ordering Functions

```
function "<"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN; ="(L:
INTEGER;  R: SIGNED)   return BOOLEAN;
```

Example B-9 Equality Functions

```
function "="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED;   R: SIGNED)   return BOOLEAN;
function "="(L: UNSIGNED; R: SIGNED)   return BOOLEAN;
function "="(L: SIGNED;   R: UNSIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: INTEGER)  return BOOLEAN;
function "="(L: INTEGER;  R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED;   R: INTEGER)  return BOOLEAN;
function "="(L: INTEGER;  R: SIGNED)   return BOOLEAN;

function "/="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED;   R: SIGNED)   return BOOLEAN;
function "/="(L: UNSIGNED; R: SIGNED)   return BOOLEAN;
function "/="(L: SIGNED;   R: UNSIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: INTEGER)  return BOOLEAN;
function "/="(L: INTEGER;  R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED;   R: INTEGER)  return BOOLEAN;
function "/="(L: INTEGER;  R: SIGNED)   return BOOLEAN;
```

Shift Functions

The std_logic_arith package provides functions for shifting the bits in SIGNED and UNSIGNED numbers. These functions produce shifters. Example B-10 shows the shift function declarations. For a list of shift and rotate operators, see “Operators” on page C-9.

Example B-10 Shift Functions

```
function SHL(ARG: UNSIGNED;
             COUNT: UNSIGNED)  return UNSIGNED;
function SHL(ARG: SIGNED;
             COUNT: UNSIGNED)  return SIGNED;

function SHR(ARG: UNSIGNED;
             COUNT: UNSIGNED)  return UNSIGNED;
function SHR(ARG: SIGNED;
             COUNT: UNSIGNED)  return SIGNED;
```

The SHL function shifts the bits of its argument ARG left by COUNT bits. SHR shifts the bits of its argument ARG right by COUNT bits.

The SHL functions work the same for both UNSIGNED and SIGNED values of ARG, shifting in zero bits as necessary. The SHR functions treat UNSIGNED and SIGNED values differently. If ARG is an UNSIGNED number, vacated bits are filled with zeros; if ARG is a SIGNED number, the vacated bits are copied from the ARG sign bit.

Example B-11 shows some shift function calls and their return values.

Example B-11 Shift Operations

```
variable U1, U2: UNSIGNED (7 downto 0);
variable S1, S2: SIGNED   (7 downto 0);
variable COUNT:  UNSIGNED (1 downto 0);
. . .
U1 <= ”01101011”;
U2 <= ”11101011”;

S1 <= ”01101011”;
S2 <= ”11101011”;

COUNT <= CONV_UNSIGNED(ARG => 3, SIZE => 2);
. . .
SHL(U1, COUNT) = ”01011000”
SHL(S1, COUNT) = ”01011000”
SHL(U2, COUNT) = ”01011000”
SHL(S2, COUNT) = ”01011000”

SHR(U1, COUNT) = ”00001101”
SHR(S1, COUNT) = ”00001101”
SHR(U2, COUNT) = ”00011101”
SHR(S2, COUNT) = ”11111101”
```

Multiplication Using Shifts

You can use shift operations for simple multiplication and division of UNSIGNED numbers if you are multiplying or dividing by a power of 2.

For example, to divide the following UNSIGNED variable U by 4, use this syntax:

```
variable U: UNSIGNED (7 downto 0) := ”11010101”;
variable quarter_U: UNSIGNED (5 downto 0);

quarter_U <= SHR(U, ”01”);
```

ENUM_ENCODING Attribute

Place the synthesis attribute ENUM_ENCODING on your primary logic type (see “Enumeration Encoding” on page 3-4). This attribute allows FPGA Compiler II / FPGA *Express* to interpret your logic correctly.

pragma built_in

Label your primary logic functions with built_in pragmas. Pragmas allow FPGA Compiler II / FPGA *Express* to interpret your logic functions easily. When you use a built_in pragma, FPGA Compiler II / FPGA *Express* parses but ignores the body of the function. Instead, FPGA Compiler II / FPGA *Express* directly substitutes the appropriate logic for the function. You need not use built_in pragmas, but they can result in runtimes that are 10 times as fast.

Use a built_in pragma by placing a comment in the declaration part of a function. FPGA Compiler II / FPGA *Express* interprets a comment as a directive if the first word of the comment is pragma. Example B-12 shows the use of a built_in pragma.

Example B-12 Using a built_in pragma

```
function ”XOR” (L, R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
  -- pragma built_in SYN_XOR
    begin
        if (L = ’1’) xor (R = ’1’) then
            return ’1’;
        else
            return ’0’;
        end if;
end ”XOR”;
```

Two-Argument Logic Functions

Synopsys provides six built-in functions for performing two-argument logic functions:

- SYN_AND
- SYN_OR
- SYN_NAND
- SYN_NOR
- SYN_XOR
- SYN_XNOR

You can use these functions on single-bit arguments or equal-length arrays of single bits. Example B-13 shows a function that takes the logical AND of two equal-size arrays.

Example B-13 Built-In AND for Arrays

```
function ”AND” (L, R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
  -- pragma built_in SYN_AND
    variable MY_L: STD_LOGIC_VECTOR (L’length-1 downto 0);
    variable MY_R: STD_LOGIC_VECTOR (L’length-1 downto 0);
    variable RESULT: STD_LOGIC_VECTOR (L’length-1 downto 0);
begin
    assert L’length = R’length;
    MY_L <= L;
    MY_R <= R;
    for i in RESULT’range loop
        if (MY_L(i) = ’1’) and (MY_R(i) = ’1’) then
            RESULT(i) <= ’1’;
        else
            RESULT(i) <= ’0’;
        end if;
    end loop;
    return RESULT;
end ”AND”;
```

One-Argument Logic Functions

Synopsys provides two built-in functions to perform one-argument logic functions:

- SYN_NOT
- SYN_BUF

You can use these functions on single-bit arguments or equal-length arrays of single bits. Example B-14 shows a function that takes the logical NOT of an array.

Example B-14 Built-In NOT for Arrays

```
function ”NOT” (L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
  -- pragma built_in SYN_NOT
     variable MY_L: STD_LOGIC_VECTOR (L’length-1 downto 0);
     variable RESULT: STD_LOGIC_VECTOR (L’length-1 downto 0);
begin
    MY_L <= L;
    for i in result’range loop
        if (MY_L(i) = ’0’ or MY_L(i) = ’L’) then
            RESULT(i) <= ’1’;
        elsif (MY_L(i) = ’1’ or MY_L(i) = ’H’) then
            RESULT(i) <= ’0’;
        else
            RESULT(i) <= ’X’;
        end if;
    end loop;
    return RESULT;
end ”NOT”;
```

Type Conversion

The built-in function SYN_FEED_THRU performs fast type conversion between unrelated types. The synthesized logic from SYN_FEED_THRU wires the single input of a function to the return value. This connection can save CPU time required to process a complicated conversion function, as shown in Example B-15.

Example B-15 Use of SYN_FEED_THRU

```
type COLOR is (RED, GREEN, BLUE);
attribute ENUM_ENCODING : STRING;
attribute ENUM_ENCODING of COLOR : type is ”01 10 11”;
...

function COLOR_TO_BV (L: COLOR) return BIT_VECTOR is
  -- pragma built_in SYN_FEED_THRU
begin
    case L is
       when RED   => return ”01”;
       when GREEN => return ”10”;
       when BLUE  => return ”11”;
    end case;
end COLOR_TO_BV;
```

numeric_std Package

FPGA Compiler II / FPGA *Express* supports nearly all of numeric_std, the IEEE Standard VHDL Synthesis Package, which defines numeric types and arithmetic functions.

Caution!

The numeric_std package and the std_logic_arith package have overlapping operations. Use of these two packages simultaneously during analysis could cause type mismatches.

Understanding the Limitations of numeric_std package

The 1999.05 version of FPGA Compiler II / FPGA *Express* does not support the following numeric_std package components:

- divide, rem, or mod operators

  If your design contains these operators, use the std_logic_arith package.

- TO_01 function as a simulation construct

Using the Package

Access numeric_std package with the following statement in your VHDL code:

```
library IEEE;
use IEEE.numeric_std.all;
```

Synopsys packages are pre-analyzed and do not require further analyzing. To list the packages currently in memory, use the following command:

```
report_design_lib
```

Data Types

The numeric_std package defines the following two data types in the same way that the std_logic_arith package does:

- USIGNED

  ```
  type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
  ```

See “UNSIGNED” on page B-6 for more information.

- SIGNED

  ```
  type SIGNED is array (NATURAL range <>) of STD_LOGIC;
  ```

  See “SIGNED” on page B-7 for more information.

Conversion Functions

The numeric_std package provides functions to convert values between its USIGNED and SIGNED types. Example B-16 shows the declarations of these conversion functions.

Example B-16 numeric_std Conversion Functions

```
function TO_INTEGER (ARG: UNSIGNED) return NATURAL;
function TO_INTEGER (ARG: SIGNED) return INTEGER;
function TO_UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;
function TO_SIGNED (ARG: INTEGER; SIZE: NATURAL) return SIGNED;
```

TO_INTEGER, TO_SIGNED, and TO_UNSIGNED are similar to CONV_INTEGER, CONV_SIGNED, and CONV_UNSIGNED in std_logic_arith (see “Conversion Functions” on page B-8).

Resize Function

The resize function numeric_std supports is shown in the declarations in Example B-17.

Example B-17 numeric_std Resize Function

```
function RESIZE (ARG: SIGNED; NEW_SIZE: NATURAL) return SIGNED;
function RESIZE (ARG: UNSIGNED; NEW_SIZE: NATURAL) return UNSIGNED;
```

Arithmetic Functions

The numeric_std package provides arithmetic functions for use with combinations of Synopsys UNSIGNED and SIGNED data types and the predefined types STD_ULOGIC and INTEGER. These functions produce adders and subtracters.

There are two sets of arithmetic functions, which the numeric_std package defines in the same way that the std_logic_arith package does (see “Arithmetic Functions” on page B-10 for more information):

- Binary functions having two arguments, such as

  A+B

  A*B

  Example B-18 shows the declarations for these functions.

- Unary functions having one argument, such as

  –A

  abs A

  Example B-19 on page B-24 shows the declarations for these functions.

Example B-18 numeric_std Binary Arithmetic Functions

```
function "+" (L, R: UNSIGNED) return UNSIGNED;
function "+" (L, R: SIGNED) return SIGNED;
function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
function "+" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
function "+" (L: INTEGER; R: SIGNED) return SIGNED;
function "+" (L: SIGNED; R: INTEGER) return SIGNED;
```

```
function "-" (L, R: UNSIGNED) return UNSIGNED;
function "-" (L, R: SIGNED) return SIGNED;
function "-" (L: UNSIGNED;R: NATURAL) return UNSIGNED;
function "-" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
function "-" (L: SIGNED; R: INTEGER) return SIGNED;
function "-" (L: INTEGER; R: SIGNED) return SIGNED;

function "*" (L, R: UNSIGNED) return UNSIGNED;
function "*" (L, R: SIGNED) return SIGNED;
function "*" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
function "*" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
function "*" (L: SIGNED; R: INTEGER) return SIGNED;
function "*" (L: INTEGER; R: SIGNED) return SIGNED;
```

Example B-19 numeric_std Unary Arithmetic Functions

```
function "abs" (ARG: SIGNED) return SIGNED;
function "-" (ARG: SIGNED) return SIGNED;
```

Comparison Functions

The numeric_std package provides functions to compare UNSIGNED and SIGNED data types to each other and to the predefined type INTEGER. FPGA Compiler II / FPGA *Express* compares the numeric values of the arguments and returns a BOOLEAN value.

These functions produce comparators. The function declarations are listed in two groups:

- Ordering functions ("<", "<=", ">", ">="), shown in Example B-20
- Equality functions ("=", "/="), shown in Example B-21 on page B-25

Example B-20 numeric_std Ordering Functions

```
function ">" (L, R: UNSIGNED) return BOOLEAN;
function ">" (L, R: SIGNED) return BOOLEAN;
function ">" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function ">" (L: INTEGER; R: SIGNED) return BOOLEAN;
function ">" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function ">" (L: SIGNED; R: INTEGER) return BOOLEAN;

function "<" (L, R: UNSIGNED) return BOOLEAN;
function "<" (L, R: SIGNED) return BOOLEAN;
function "<" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function "<" (L: INTEGER; R: SIGNED) return BOOLEAN;
function "<" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function "<" (L: SIGNED; R: INTEGER) return BOOLEAN;

function "<=" (L, R: UNSIGNED) return BOOLEAN;
function "<=" (L, R: SIGNED) return BOOLEAN;
function "<=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function "<=" (L: INTEGER; R: SIGNED) return BOOLEAN;
function "<=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function "<=" (L: SIGNED; R: INTEGER) return BOOLEAN;

function ">=" (L, R: UNSIGNED) return BOOLEAN;
function ">=" (L, R: SIGNED) return BOOLEAN;
function ">=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function ">=" (L: INTEGER; R: SIGNED) return BOOLEAN;
function ">=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function ">=" (L: SIGNED; R: INTEGER) return BOOLEAN;
```

Example B-21 numeric_std Equality Functions

```
function "=" (L, R: UNSIGNED) return BOOLEAN;
function "=" (L, R: SIGNED) return BOOLEAN;
function "=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function "=" (L: INTEGER; R: SIGNED) return BOOLEAN;
function "=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function "=" (L: SIGNED; R: INTEGER) return BOOLEAN;

function "/=" (L, R: UNSIGNED) return BOOLEAN;
function "/=" (L, R: SIGNED) return BOOLEAN;
function "/=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
```

```
function "/=" (L: INTEGER; R: SIGNED) return BOOLEAN;
function "/=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function "/=" (L: SIGNED; R: INTEGER) return BOOLEAN;
```

Defining Logical Operators Functions

The numeric_std package provides functions that define all of the logical operators: NOT, AND, OR, NAND, NOR, XOR, and XNOR. These functions work just like similar functions in std_logic_1164, except that they operate on SIGNED and UNSIGNED values rather than on STD_LOGIC_VECTOR values. Example B-22 shows these function declarations.

Example B-22 numeric_std Logical Operators Functions

```
function "not" (L: UNSIGNED) return UNSIGNED;
function "and" (L, R: UNSIGNED) return UNSIGNED;
function "or" (L, R: UNSIGNED) return UNSIGNED;
function "nand" (L, R: UNSIGNED) return UNSIGNED;
function "nor" (L, R: UNSIGNED) return UNSIGNED;
function "xor" (L, R: UNSIGNED) return UNSIGNED;
function "xnor" (L, R: UNSIGNED) return UNSIGNED;

function "not" (L: SIGNED) return SIGNED;
function "and" (L, R: SIGNED) return SIGNED;
function "or" (L, R: SIGNED) return SIGNED;
function "nand" (L, R: SIGNED) return SIGNED;
function "nor" (L, R: SIGNED) return SIGNED;
function "xor" (L, R: SIGNED) return SIGNED;
function "xnor" (L, R: SIGNED) return SIGNED;
```

Shift Functions

The numeric_std package provides functions for shifting the bits in UNSIGNED and SIGNED numbers. These functions produce shifters. Example B-23 shows the shift function declarations.

Example B-23 numeric_std Shift Functions

```
function SHIFT_LEFT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
function SHIFT_RIGHT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
function SHIFT_LEFT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
function SHIFT_RIGHT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;

function ROTATE_LEFT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
function ROTATE_RIGHT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
function ROTATE_LEFT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
function ROTATE_RIGHT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
```

The SHIFT_LEFT function shifts the bits of its argument ARG left by COUNT bits. SHIFT_RIGHT shifts the bits of its argument ARG right by COUNT bits.

The SHIFT_LEFT functions work the same for both UNSIGNED and SIGNED values of ARG, shifting in zero bits as necessary. The SHIFT_RIGHT functions treat UNSIGNED and SIGNED values differently:

- If ARG is an UNSIGNED number, vacated bits are filled with zeros
- If ARG is a SIGNED number, the vacated bits are copied from the ARG sign bit

Example B-26 on page B-29 shows some shift functions calls and their return values.

Rotate Functions

ROTATE_LEFT and ROTATE_RIGHT are similar to the shift functions.

Example B-24 shows rotate function declarations.

Example B-24 numeric_std Rotate Functions

```
ROTATE_LEFT (U1, COUNT) = "01011011"
ROTATE_LEFT (S1, COUNT) = "01011011"
ROTATE_LEFT (U2, COUNT) = "01011111"
ROTATE_LEFT (S2, COUNT) = "01011111"

ROTATE_RIGHT (U1, COUNT) = "01101101"
ROTATE_RIGHT (S1, COUNT) = "01101101"
ROTATE_RIGHT (U2, COUNT) = "01111101"
ROTATE_RIGHT (S2, COUNT) = "01111101"
```

Shift and Rotate Operators

The numeric_std package provides shift operators and rotate operators, which work in the same way that shift functions and rotate functions do. The shift operators are: sll, srl, sla, and sra. Example B-25 shows some shift and rotate operator declarations. Example B-26 on page B-29 includes some shift and rotate operators.

Example B-25 numeric_std Shift Operators

```
function "sll" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
function "sll" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
function "srl" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
function "srl" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
function "rol" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
function "rol" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
function "ror" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
function "ror" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
```

Example B-26 Some numeric_std Shift Functions and Shift Operators

```
Variable U1, U2: UNSIGNED (7 downto 0);
Variable S1, S2: SIGNED (7 downto 0);
Variable COUNT: NATURAL;
...
U1 <= "01101011";
U2 <= "11101011";
S1 <= "01101011";
S2 <= "11101011";
COUNT <= 3;
...
SHIFT_LEFT (U1, COUNT) = "01011000"
SHIFT_LEFT (S1, COUNT) = "01011000"
SHIFT_LEFT (U2, COUNT) = "01011000"
SHIFT_LEFT (S2, COUNT) = "01011000"

SHIFT_RIGHT (U1, COUNT) = "00001101"
SHIFT_RIGHT (S1, COUNT) = "00001101"
SHIFT_RIGHT (U2, COUNT) = "00011101"
SHIFT_RIGHT (S2, COUNT) = "11111101"

U1 sll COUNT = "01011000"
S1 sll COUNT = "01011000"
U2 sll COUNT = "01011000"
S2 sll COUNT = "01011000"

U1 srl COUNT = "00001101"
S1 srl COUNT = "00001101"
U2 srl COUNT = "00011101"
S2 srl COUNT = "11111101"

U1 rol COUNT = "01011011"
S1 rol COUNT = "01011011"
U2 rol COUNT = "01011111"
S2 rol COUNT = "01011111"

U1 ror COUNT = "01101101"
S1 ror COUNT = "01101101"
U2 ror COUNT = "01111101"
S2 ror COUNT = "01111101"
```

std_logic_misc Package

The std_logic_misc package resides in the lib/packages/IEEE/src/std_logic_misc.vhd subdirectory of the FPGA Compiler II / FPGA *Express* directory. It declares the primary data types the Synopsys VSS tools support.

Boolean reduction functions take one argument (an array of bits) and return a single bit. For example, the AND reduction of ”101” is ”0”, the logical AND of all three bits.

Several functions in the std_logic_misc package provide Boolean reduction operations for the predefined type STD_LOGIC_VECTOR. Example B-27 shows the declarations of these functions.

Example B-27 Boolean Reduction Functions

```
function AND_REDUCE  (ARG: STD_LOGIC_VECTOR) return UX01;
function NAND_REDUCE (ARG: STD_LOGIC_VECTOR) return UX01;
function OR_REDUCE   (ARG: STD_LOGIC_VECTOR) return UX01;
function NOR_REDUCE  (ARG: STD_LOGIC_VECTOR) return UX01;
function XOR_REDUCE  (ARG: STD_LOGIC_VECTOR) return UX01;
function XNOR_REDUCE (ARG: STD_LOGIC_VECTOR) return UX01;
function AND_REDUCE  (ARG: STD_ULOGIC_VECTOR) return UX01;
function NAND_REDUCE (ARG: STD_ULOGIC_VECTOR) return UX01;
function OR_REDUCE   (ARG: STD_ULOGIC_VECTOR) return UX01;
function NOR_REDUCE  (ARG: STD_ULOGIC_VECTOR) return UX01;
function XOR_REDUCE  (ARG: STD_ULOGIC_VECTOR) return UX01;
function XNOR_REDUCE (ARG: STD_ULOGIC_VECTOR) return UX01;
```

These functions combine the bits of the STD_LOGIC_VECTOR, as the name of the function indicates. For example, XOR_REDUCE returns the XOR of all bits in ARG. Example B-28 shows some reduction function calls and their return values.

Example B-28 Boolean Reduction Operations

```
AND_REDUCE("111") = '1'
AND_REDUCE("011") = '0'

OR_REDUCE("000")  = '0'
OR_REDUCE("001")  = '1'

XOR_REDUCE("100") = '1'
XOR_REDUCE("101") = '0'

NAND_REDUCE("111")= '0'
NAND_REDUCE("011")= '1'

NOR_REDUCE("000") = '1'
NOR_REDUCE("001") = '0'

XNOR_REDUCE("100")= '0'
XNOR_REDUCE("101")= '1'
```

ATTRIBUTES Package

The ATTRIBUTES package declares all the supported synthesis (and simulation) attributes. These include:

- FPGA Compiler II / FPGA *Express* constraints and attributes
- State vector attributes
- Resource sharing attributes
- General attributes for interpreting VHDL (described in Chapter 3, "Data Types")
- Attributes for use with the Synopsys VSS tools

Reference this package when you use synthesis attributes:

```
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;
```

C

VHDL Constructs

Many VHDL language constructs, although useful for simulation and other stages in the design process, are not relevant to synthesis. Because these constructs cannot be synthesized, FPGA Compiler II / FPGA *Express* does not support them.

This appendix provides a list of all VHDL language constructs, with the level of support for each, followed by a list of VHDL reserved words.

This appendix describes

- VHDL Construct Support
- VHDL Reserved Words

VHDL Construct Support

A construct can be fully supported, ignored, or unsupported. Ignored and unsupported constructs are defined as follows:

- Ignored means that the construct is allowed in the VHDL source but is ignored by FPGA Compiler II / FPGA *Express*.
- Unsupported means that the construct is not allowed in the VHDL source and that FPGA Compiler II / FPGA *Express* flags it as an error. If errors are in a VHDL description, the description is not translated (synthesized).

Constructs are listed in the following order:

- Design units
- Data types
- Declarations
- Specifications
- Names
- Operators
- Operands and expressions
- Sequential statements
- Concurrent statements
- Predefined language environment

Design Units

entity

The entity statement part is ignored. Generics are supported, but only of type INTEGER. Default values for ports are ignored.

architecture

Multiple architectures are allowed. Global signal interaction between architectures is unsupported.

configuration

Configuration declarations and block configurations are supported, but only to specify the top-level architecture for a top-level entity.

The use clauses, attribute specifications, component configurations, and nested block configurations are unsupported.

package

Packages are fully supported.

library

Libraries and separate compilation are supported.

subprogram

Default values for parameters are unsupported. Assigning to indexes and slices of unconstrained out parameters is unsupported, unless the actual parameter is an identifier.

Subprogram recursion is unsupported if the recursion is not bounded by a static value.

Resolution functions are supported for wired-logic and three-state functions only.

Subprograms can be declared only in packages and in the declaration part of an architecture.

Data Types

enumeration
: Enumeration is fully supported.

integer
: Infinite-precision arithmetic is unsupported.

  Integer types are automatically converted to bit vectors whose width is as small as possible to accommodate all possible values of the type's range. The type's range can be either in unsigned binary for nonnegative ranges or in 2's-complement form for ranges that include negative numbers.

physical
: Physical type declarations are ignored. The use of physical types is ignored in delay specifications.

floating
: Floating-point type declarations are ignored. The use of floating-point types is unsupported except for floating-point constants used with Synopsys-defined attributes.

array
: Array ranges and indexes other than integers are unsupported.

  Multidimensional arrays are unsupported, but arrays of arrays are supported.

record
: Record data types are fully supported.

access
: Access type declarations are ignored, and the use of access types is unsupported.

file
: File type declarations are ignored, and the use of file types is unsupported.

incomplete type declarations
: Incomplete type declarations are unsupported.

Declarations

constant
: Constant declarations are supported except for deferred constant declarations.

signal
: Register and bus declarations are unsupported. Resolution functions are supported for wired and three-state functions only. Declarations other than from a globally static type are unsupported. Initial values are unsupported.

variable
: Declarations other than from a globally static type are unsupported. Initial values are unsupported.

shared variable
: Variable shared by different processes. Shared variables are fully supported.

file
: File declarations are unsupported.

interface
: Buffer and linkage are translated to out and inout, respectively.

alias

Alias declarations are supported, with the following exceptions:

- An alias declaration that lacks a subtype indication
- A nonobject alias—such as an alias that refers to a type.

component

Component declarations that list a name other than a valid entity name are unsupported.

attribute

Attribute declarations are fully supported, but the use of user-defined attributes is unsupported.

Specifications

attribute

Others and all are unsupported in attribute specifications. User-defined attributes can be specified, but the use of user-defined attributes is unsupported.

configuration

Configuration specifications are unsupported.

disconnection

Disconnection specifications are unsupported. Attribute declarations are fully supported, but the use of user-defined attributes is unsupported.

Names

simple
: Simple names are fully supported.

selected
: Selected (qualified) names outside a use clause are unsupported. Overriding the scopes of identifiers is unsupported.

operator symbol
: Operator symbols are fully supported.

indexed
: Indexed names are fully supported, with one exception: Indexing an unconstrained out parameter in a procedure is unsupported.

slice
: Slice names are fully supported, with one exception: Using a slice of an unconstrained out parameter in a procedure is unsupported unless the actual parameter is an identifier.

attribute
: Only the following predefined attributes are supported: base, left, right, high, low, range, reverse_range, and length. The event and stable attributes are supported only as described with the wait and if statements (see “wait Statements” on page 5-50). User-defined attribute names are unsupported. The use of attributes with selected names (name.name’attribute) is unsupported.

Identifiers and Extended Identifiers

An identifier in VHDL is a user-defined name for any of these: constant, variable, function, signal, entity, port, subprogram, parameter, and instance.

Specifics of Identifiers

The characteristics of identifiers are:

- They can be composed of letters, digits, and the underscore character (_).
- Their first character cannot be a number, unless it is an extended identifier (see Example C-1).
- They can be of any length.
- They are case-insensitive.
- All of their characters are significant.

Specifics of Extended Identifiers

The characteristics of extended identifiers are:

- Any of the following can be defined as one:
  - Identifiers that contain special characters
  - Identifiers that begin with numbers
  - Identifiers that have the same name as a keyword

- They start with a backslash character (\), followed by a sequence of characters, followed by another backslash (\).
- They are case-sensitive.

Example C-1 shows some extended identifiers.

Example C-1 Sample Extended Identifiers

```
\a+b\                    \3state\
\type\                   \(a&b)|c\
```

For more information about identifiers and extended identifiers, see “Identifiers” on page 4-23.

Operators

logical
: Logical operators are fully supported.

relational
: Relational operators are fully supported.

addition
: Concatenation and arithmetic operators are fully supported.

signing
: Signing operators are fully supported.

multiplying
: The * (multiply) operator is fully supported. The / (division), mod, and rem operators are supported only when both operands are constant or when the right operand is a constant power of 2.

miscellaneous
: The ** operator is supported only when both operands are constant or when the left operand is 2. The abs operator is fully supported.

operator overloading
: Operator overloading is fully supported.

short-circuit operation
: The short-circuit behavior of operators is not supported.

Shift and Rotate Operators

You can define shift and rotate operators for any one-dimensional array type whose element type is either of the predefined types, BIT or Boolean. The right operand is always of type integer. The type of the result of a shift operator is the same as the type of the left operand. The shift and rotate operators are included in the list of VHDL reserved words in Table C-1 on page C-17. There is more information about the shift and rotate operators that numeric_std supports in “Shift and Rotate Operators” on page B-28. The shift operators are:

sll
: Shift left logical

srl
: Shift right logical

sla
: Shift left arithmetic

sra
: Shift right arithmetic

The rotate operators are

rol
: Rotate left logical

ror
: Rotate right logical

Example C-2 illustrates the use of shift and rotate operators.

Example C-2 Sample Showing Use of Shift and Rotate Operators

```
architecture arch of shft_op is
begin
    a <= "01101";
    q1 <= a sll 1;         -- q1 = "11010"
    q2 <= a srl 3;         -- q2 = "00001"
    q3 <= a rol 2;         -- q3 = "10101"
    q4 <= a ror 1;         -- q4 = "10110"
    q5 <= a sla 2;         -- q5 = "10100"
    q6 <= a sra 1;         -- q6 = "00110"
end;
```

xnor Operator

You can define the binary logical operator xnor for predefined types BIT and Boolean, as well as for any one-dimensional array type whose element type is BIT or Boolean. The operands must be the same type and length. The result also has the same type and length. The xnor operator is included in the list of VHDL reserved words in Table C-1 on page C-17.

Example C-3 Sample Showing Use of xnor Operator

```
a <= "10101";
b <= "11100";
c <= a xnor b;             -- c = "10110"
```

Operands and Expressions

based literal
: Based literals are fully supported.

null literal
: Null slices, null ranges, and null arrays are unsupported.

physical literal
: Physical literals are ignored.

string
: Strings are fully supported.

aggregate
: The use of types as aggregate choices is supported. Record aggregates are supported.

function call
: Function calls are supported, with one exception: Function conversions on input ports are not supported, because type conversions on formal ports in a connection specification (port map) are not supported.

qualified expression
: Qualified expressions are fully supported.

type conversion
: Type conversion is fully supported.

allocator
: Allocators are unsupported.

static expression
: Static expressions are fully supported.

universal expression

Floating-point expressions are unsupported, except in a Synopsys-recognized attribute definition. Infinite-precision expressions are not supported. Precision is limited to 32 bits; all intermediate results are converted to integer.

Sequential Statements

wait

The wait statement is unsupported unless it is in one of the following forms:

```
wait until                        clock = VALUE;
wait until     clock'event  and clock = VALUE;
wait until not clock'stable and clock = VALUE;
```

VALUE is '0', '1', or an enumeration literal whose encoding is 0 or 1. A wait statement in this form is interpreted to mean "wait until the falling (VALUE is '0') or rising (VALUE is '1') edge of the signal named clock." You cannot use wait statements in subprograms.

assert

Assert statements are ignored.

report

Report statements are ignored.

statement label

Statement labels are ignored.

signal

Guarded signal assignment is unsupported. The transport and after signals are ignored. Multiple waveform elements in signal assignment statements are unsupported.

variable

Variable statements are fully supported.

procedure call

Type conversion on formal parameters is unsupported. Assignment to single bits of vectored ports is unsupported.

if

The if statements are fully supported.

case

The case statements are fully supported.

loop

The for loops are supported, with two constraints: The loop index range must be globally static, and the loop body must not contain a wait statement. The while loops are supported, but the loop body must contain at least one wait statement. The loop statements with no iteration scheme (infinite loops) are supported, but the loop body must contain at least one wait statement.

next

Next statements are fully supported.

exit

Exit statements are fully supported.

return

Return statements are fully supported.

null

Null statements are fully supported.

Concurrent Statements

block
: Guards on block statements are supported. Ports and generics in block statements are unsupported.

process
: Sensitivity lists in process statements are ignored.

concurrent procedure call
: Concurrent procedure call statements are fully supported.

concurrent assertion
: Concurrent assertion statements are ignored.

concurrent signal assignment
: The guarded keyword is supported. The transport keyword is ignored. Multiple waveforms are unsupported.

component instantiation
: Type conversion on the formal port of a connection specification is unsupported.

generate
: The generate statements are fully supported.

Predefined Language Environment

severity_level type
: The severity_level type is unsupported.

time type
: The time type is ignored if time variables and constants are used only in after clauses. In the following two code fragments, both the after clause and TD are ignored:

```
constant TD: time := 1.4 ns;
X <= Y after TD;

X <= Y after 1.4 ns;
```

now function
: The now function is unsupported.

TEXTIO package
: The TEXTIO package is unsupported.

predefined attributes
: These predefined attributes are supported: base, left, right, high, low, range, reverse_range, ascending, and length. The event and stable attributes are supported only in the if and wait statements, as described in “wait Statements” on page 5-50.

VHDL Reserved Words

Table C-1 lists the words that are reserved for the VHDL language and cannot be used as identifiers:

Table C-1 VHDL Reserved Words

abs	exit	new	select
access		next	severity
after	file	nor	shared
alias	for	not	signal
all	function	null	sla
and			sll
architecture		of	sra
array	generate	on	srl
assert	generic	open	subtype
attribute	group	or	
	guarded	others	then
begin		out	to
block	if		transport
body	impure	package	type
buffer	in	port	
bus	inertial	postponed	unaffected
	inout	procedure	units
case	is	process	until
component		pure	use
configuration	label		
constant	library	range	variable
	linkage	record	
disconnect	literal	register	wait
downto	loop	reject	when
		rem	while
else	map	report	with
elsif	mod	return	
end		rol	xnor
entity	nand	ror	xor

Glossary

anonymous type
A predefined or underlying type with no name, such as universal integers.

ASIC
Application-specific integrated circuit.

behavioral view
The set of Verilog statements that describe the behavior of a design by using sequential statements. These statements are similar in expressive capability to those found in many other programming languages. See also the *data flow view*, *sequential statement*, and *structural view* definitions.

bit-width
The width of a variable, signal, or expression in bits. For example, the bit-width of the constant 5 is 3 bits.

character literal
Any value of type CHARACTER, in single quotation marks.

computable
Any expression whose (constant) value FPGA Compiler II / FPGA *Express* can determine during translation.

constraints
The designer's specification of design performance goals. FPGA Compiler II / FPGA *Express* uses constraints to direct the optimization of a design to meet area and timing goals.

convert
To change one type to another. Only integer types and subtypes are convertible, along with same-sized arrays of convertible element types.

data flow view
The set of VHDL/Verilog statements that describe the behavior of a design by using concurrent statements. These descriptions are usually at the level of Boolean equations combined with other operators and function calls. See also the *behavioral view* and *structural view* definitions.

design constraints
See *constraints*.

flip-flop
An edge-sensitive memory device.

HDL
Hardware Description Language.

identifier
A sequence of letters, underscores, and numbers. An identifier cannot be a VHDL/Verilog reserved word, such as type or loop. An identifier must begin with a letter or an underscore.

latch
A level-sensitive memory device.

netlist
A network of connected components that together define a design.

optimization
The modification of a design in an attempt to improve some performance aspect. FPGA Compiler II / FPGA *Express* optimizes designs and tries to meet specified design constraints for area and speed.

port
A signal declared in the interface list of an entity.

reduction operator
An operator that takes an array of bits and produces a single-bit result, namely the result of the operator applied to each successive pair of array elements.

register
A memory device containing one or more flip-flops or latches used to hold a value.

resource sharing
The assignment of a similar VHDL/Verilog operation (for example, +) to a common netlist cell. Netlist cells are the resources—they are equivalent to built hardware.

RTL
Register transfer level, a set of structural and data flow statements.

sequential statement
A set of VHDL/Verilog statements that execute in sequence.

signed value
A value that can be positive, zero, or negative.

structural view
The set of VHDL/Verilog statements used to instantiate primitive and hierarchical components in a design. A VHDL/Verilog design at the structural level is also called a netlist. See also the *behavioral view* and *data flow view* definitions.

subtype
A type declared as a constrained version of another type.

synthesis
The creation of optimized circuits from a high-level description. When VHDL/Verilog is used, synthesis is a two-step process: translation from VHDL/Verilog to gates and optimization of those gates for a specific FPGA library.

technology library
A library of cells available to FPGA Compiler II / FPGA *Express* during the synthesis process. A technology library can contain area, timing, and functional information on each cell.

translation
The mapping of high-level language constructs onto a lower-level form. FPGA Compiler II / FPGA *Express* translates RTL VHDL/ Verilog descriptions to gates.

type
In VHDL/Verilog, the mechanism by which objects are restricted in the values they are assigned and the operations that can be applied to them.

unsigned
A value that can be only positive or zero.

variable
An electrical quantity that can be used to transmit information. A signal is declared with a type and receives its value from one or more drivers. Signals are created in Verilog through either wire or reg declarations.

VHDL
VHSIC hardware description language.

VHSIC
Very high speed integrated circuit, a high-technology program of the United States Department of Defense.

Index

Symbols

” B-25, B-25
”&” (concatenation operator) 4-8
”**” (exponentiation operator) 4-12
”*” (multiplying operator) 4-11
”*” function B-12, B-24
”+” (adding operator) 4-8
”+” (unary operator) 4-10
”+” function B-11, B-24
”/=” (relational operator) 4-6
”/=” function B-14
”/” (multiplying operator) 4-11
”<=” function B-14
”<” function B-14
”=” (relational operator) 4-6
”=” function B-14, B-26
”>=” (relational operator) 4-6
”>=” function B-14, B-25
”>” (relational operator) 4-6
”>” function B-14, B-25
”-” (adding operator) 4-8
”-” (unary operator) 4-10
”-” function B-24
”–” function B-11

A

abs (absolute value operator) 4-12
actual parameters
  subprograms 2-26
adder-subtracter (example) A-17
adding operators 4-8
aggregate target syntax 5-9
aggregates C-12
aggregates (array literals) 4-18
aggregates, record 3-14
algorithms
  processes 2-19
alias declarations
  supported C-6
and (logical operator) 4-3
architecture 2-5
  dataflow
    two-input NAND gate 2-34
  defined 2-5
  overriding entity port names 2-8
  RTL
    two-input NAND gate 2-34
  statement, entity 2-5
  structural
    two-input NAND gate 2-33
arithmetic functions
  numeric_std

binary B-23
unary B-23
arithmetic operators
adding 4-8
multiplying 4-11
negating 4-10
arithmetic optimization
considering overflow from carry bits 8-10
introduction 8-6
array attributes
RANGE
example 5-28
array data type
attributes
high 3-12
index 3-12
left 3-12
length 3-12
low 3-12
predefined 3-12
range 3-12
reverse_range 3-12
right 3-12
using 3-12
concatenating 4-9
constrained
array_type_name 3-10
defining 3-10
illustration 3-10
index 3-10
syntax 3-10
definition of 3-9
index
constrained 3-9
ordering 4-6
unconstrained
advantages 3-11
array_type_name 3-11
defining 3-11
element_type_name 3-11
range_type_name 3-11
syntax 3-11
array literals
as aggregates 4-18
as bit strings 4-28
array_type_name 3-10, 3-11
arrival time 8-8
assert statement C-13
assignment statement
aggregate target 5-9
field target 5-8
indexed name target 5-4
signal
syntax 5-12
simple name target 5-3
slice target 5-7
syntax 5-2
variable
syntax 5-11
async_set_reset attribute 7-5
async_set_reset_local attribute 7-5
async_set_reset_local_all attribute 7-5
asynchronous designs
optimization 8-37
using 8-23
asynchronous processes 6-3
attribute declarations C-6
attributes
array 3-12
as operands 4-20
ENUM_ENCODING B-17
synthesis_off 7-8
synthesis_on 7-8
VHDL
ENUM_ENCODING 3-5
ENUM_ENCODING values 3-7
ATTRIBUTES package B-2, B-31

B

binary arithmetic functions
example B-11
numeric_std B-23

binary bit string 4-28
bit string literals 4-28
BIT type 3-17
bit vectors
  as bit strings 4-28
bit width (of operands) 4-15
BIT_VECTOR type 3-17, B-4
block 2-17
block statement
  block_declarative_item 6-10
  edge-sensitive latch 6-13
  guarded 6-10
  guarded blocks 6-12
  level-sensitive latch 6-12
  nested blocks 6-11
block statements
  guards C-15
block_declarative_item
  entity architecture 2-6
  in block statement 6-10
body
  subprogram 2-23
Boolean reduction functions B-30
BOOLEAN type 3-17
buffer
  port mode 2-4, 2-24, 2-25
built_in directive
  logic functions B-18
  type conversion B-19
  using B-17
built_in pragma
  example of using B-17
bus resolution function 6-9
bused clock
  syntax 7-22

C

carry-lookahead adder (example) A-32
carry-out bit
  example of using B-13
case statement
  invalid usages 5-21
  syntax 5-17
character literals 4-26
character string literals 4-28
CHARACTER type 3-17
clock, bused 7-22
combinational feedback
  paths 8-35
combinational logic 8-2
combinational processes 5-55, 6-5
common subexpressions
  sharing 8-12
comparison functions
  numeric_std B-24
compiler directives 5-45
component
  declaration
    generic parameter 2-11
    N-bit adder 2-11
    port name and order 6-23
    two-input AND gate, example 2-11
  implication
    directives 5-46
    example 5-47
    latches and registers 5-55
    three-state driver 7-59
  instantiation
    defined 2-17
    direct 6-25
    port map 6-23
    search order 2-13
    statement 2-13, 6-22
  mapping subprogram to 5-45
component declarations C-6
component implication
  registers 7-1
computable operands 4-16
concatenation operator 4-9
concurrent procedure call
  eqivalent process 6-14

syntax 6-14
concurrent signal assignment 6-17
conditional signal assignment 6-18
selected signal assignment 6-20
concurrent statement
block 2-17
component instantiation 2-17
procedure call 2-17
signal assignment 2-18
supported C-15
conditional signal assignment
equivalent process 6-19
syntax 6-18
conditionally assigned variable 7-19
conditionally specified signal 8-36
constant declaration
defined 2-18
supported C-5
value 2-18
constant propagation 8-21
constants
record aggregates 3-14
constrained data array 3-10
constructs, VHDL
architecture 2-5
constant declaration 2-18
subtype declaration 2-32
type declaration 2-32
variable declaration 2-20, 2-31
block
constant declaration 2-18
subtype declaration 2-32
type declaration 2-32
component instantiation 2-13
declaration
constant 2-18
signal 2-21
variable 2-20, 2-31
entity
constant declaration 2-18
defined 2-2
subtype declaration 2-32
type declaration 2-32
operator
overloading 2-30
package
constant declaration 2-18
subtype declaration 2-32
type declaration 2-32
process
constant declaration 2-18
defined 2-19
subtype declaration 2-32
type declaration 2-32
signal
bus resolution function 6-9
resolution function 2-40
subprogram
constant declaration 2-18
function 2-22, 2-24
overloading 2-29
procedure 2-22, 2-23
subtype declaration 2-32
type declaration 2-32
subtype
declaration 2-32
defined 2-32
variable
declaration 2-20, 2-31
control unit (example)
counting A-29
state machine A-24
CONV_INTEGER functions B-8
CONV_SIGNED functions B-9
CONV_UNSIGNED functions B-8
conversion functions
arithmetic
binary B-11
for adders and subtracters B-10
unary B-12
numeric_std
TO_INTEGER B-22
TO_SIGNED B-22

TO_UNSIGNED B-22
std_logic_arith package B-8
count zeros (example)
combinational A-19
sequential A-22
COUNTER3
description
structural design 2-15
critical path 8-8

D

data type
abstract
BOOLEAN 3-1
advantages 3-2
array
constrained 3-10
syntax 3-10
array attributes
high 3-12
index 3-12
left 3-12
length 3-12
low 3-12
range 3-12
reverse_range 3-12
right 3-12
BIT 3-18
BIT_VECTOR 3-19
BOOLEAN 3-18
CHARACTER 3-18
described 3-1
enumeration syntax 3-3
hardware-related BIT 3-1
integer
defined 3-19
syntax 3-8
new type defined
BYTE, example 3-2
predefined
STANDARD package 3-1
record 3-13
subtype
defined 3-3
syntax 2-32
supported C-4
SYNOPSYS
std_logic_signed 3-9
std_logic_unsigned 3-9
data types
numeric_std
SIGNED B-22
UNSIGNED B-22
dataflow architecture
NAND2 entity 2-34
declaration
constant 2-18
example 2-18
incorrect use of port name example 2-9
signal
example 2-21
incorrect use of port name example 2-9
logical 4-4
subprogram
function syntax 2-24
procedure syntax 2-23
subtype 2-32
supported C-5
variable
defined 2-20, 2-31
example 2-20, 2-31
definitions
register inference 7-1
design architecture
concurrent statement 2-17
block 2-17
block_declarative_item 2-6
component instantiation 2-13, 2-17
procedure call 2-17
process 2-17, 2-19
signal assignment 2-18
declaration section
component 2-10

constant 2-10
signal 2-10
subprogram 2-10
type 2-10
design units
package 2-35, 2-36
subprogram 2-22
organization, illustrated 2-5
Design Compiler
asynchronous designs 8-23
design style
data type
enumeration 3-3
integer 3-8
data types 3-2
design unit
package 2-35
supported C-3
designs
efficiency 8-22
structure 8-3
direct component instantiation 6-25
directives
built_in
identifying B-6
using B-17
component implication 5-46
map_to_entity 5-45, 6-14
resolution_method 2-41
return_port_name 5-45
synthetic 9-2
translate_off, warning 9-3
translate_on, warning 9-3
using 2-42
dont care inference
example 8-29
simulation versus synthesis 8-33
using 8-32

E

edge expression 7-58
element_type_name 3-11
encoding
values
ENUM_ENCODING attribute 3-7
vectors
ENUM_ENCODING attribute 3-6
entity
architecture
defined 2-2
syntax 2-5
three-bit counter 2-7
two-input NAND gate 2-34
composition 2-2
consistency
component instantiation 2-14
defined 2-2
generic specification
example 2-5
syntax 2-3
port specification
overriding port names 2-8
port modes 2-4, 2-24, 2-25
syntax 2-4
specification
NAND2 gate 2-3, 2-34
three-bit counter 2-7
ENUM_ENCODING attribute 3-5, B-17
values 3-7
vectors 3-7
enumerated types
ordering 4-6
enumeration data type
encoding
ENUM_ENCODING attribute 3-5
ENUM_ENCODING value 3-7
example 3-5
literal value 3-4
example
COLOR 3-4
encoding 3-5
MY_LOGIC 3-4
literal, overloaded 3-4

syntax 3-3
enumeration literals 4-27
equality functions
example B-14
equality operators 4-6
escaped identifier.*See* extended identifier
examples
adder-subtracter A-17
asynchronous design
incorrect 8-27
carry-lookahead adder A-32
case statement
enumerated type 5-18
combinational process 6-5, 6-6
component implication 5-47
control unit
counting A-29
state machine A-24
count zeros
combinational A-19
sequential A-22
dont care usage 8-29
enumeration encoding
dont care 8-29
for ... generate 6-28
function call 5-42
if statement 5-15
integer data type
definitions 3-8
Mealy finite state machine A-5
Moore finite state machine A-2
PLA A-51
ROM A-7
sequential processes 6-6
serial-to-parallel converter
counting bits A-40
shifting bits A-47
simulation driver 9-3
subprograms
component implication 5-47
declarations 5-36
function call 5-42
synchronous design 8-23
three-state component
registered input 7-67
two-phase clocked design 7-20
wait statement 5-56
in a loop 5-52
multiple waits 5-52
waveform generator
complex A-13
simple A-10
exit statement 5-33
exponentiation operator 4-12
expression tree 8-7
subexpressions in 8-9
expressions
described 4-1
relational
true 4-7
supported C-12
use 4-1
using parentheses in 8-9
expressions, VHDL, tick (') 4-29
extended identifier C-8

F

falling_edge 7-22, 7-28
feedback paths 8-35
field target syntax 5-8
file declarations C-5
flip-flop
definition 7-1
inference 7-22
for ... generate statement
example 6-28
syntax 6-26
for ... loop statement
and exit statement 5-33
arrays 5-28
label as identifier in 5-25, 6-27
syntax 5-23, 5-25

formal parameters
  subprograms 2-26
fully specified
  signal 8-35
  variable 8-35
function
  call 4-22, 5-41
  declaration
    syntax 2-24
  resolution
    allowed 2-41
    bus 6-9
    creating 2-40
    directives, using 2-41
    example 2-42
    marking 2-41
    signal 2-40
    syntax, declaration 2-40
    syntax, subtype 2-41
    syntax, type 2-40
  value 2-22
functions
  description 5-38
  implementations
    mapped to component 5-47
    mapped to gates 5-49
  return statement 5-43

G

generate statements
  for ... generate 6-26
  if ... generate 6-26
generic
  map
    component instantiation 2-13
  parameter
    component declaration 2-11
    two-input AND gate 2-11
  specification
    entity 2-3
    entity syntax 2-3
  values
    mapping 2-14
guard
  on block statement C-15
guarded blocks
  in block statement 6-12
guarded keyword C-15

H

hdlin_pragma_keyword variable 9-2
hexadecimal bit string 4-28
high attribute 3-12
high impedance state 7-59

I

identifier C-8
  extended C-9
identifiers
  defined 4-23
  enumeration literals 4-27
if ... generate statement
  syntax 6-31
if statement
  creating registers 7-23
implying registers 7-1
incompletely specified 8-35
indexed name target 5-4
indexed names
  computability 4-25
  using 4-24
inequality operators 4-6
inference report
  example 7-3
inferred registers
  limitations 7-57
instantiation
  component
    direct 6-25
integer data type

bits, accessing
std_logic_signed package 3-9
std_logic_unsigned package 3-9
defining 3-8
definitions
example 3-8
encoding 3-8
INTEGER type 3-17
subrange 3-8

K

keywords C-17

L

language constructs, VHDL
concurrent statements
assertion C-15
block 2-17, C-15
component instantiation 2-13, 2-17, C-15
function 2-22, 2-24
generate C-15
procedure 2-22, 2-23
procedure call 2-17, C-15
process 2-17, 2-19, C-15
signal assignment 2-18, C-15
data types
access C-4
array 3-10, C-4
enumeration 3-3, C-4
file C-5
floating C-4
incomplete type declarations C-5
integer 3-8, C-4
physical C-4
record C-4
subtype 2-32
dataflow
entity, NAND2 2-34
declaration
constant 2-18
signal 2-21
variable 2-20, 2-31
declarations
alias C-6
attribute C-6
component C-6
constant C-5
file C-5
interface C-5
shared variable C-5
signal C-5
variable C-5
design units
architecture 2-5, C-3
configuration 2-34, C-3
entity C-3
entity, NAND2 2-3
library C-3
package 2-35, 2-36, C-3
subprogram C-3
subprogram, overloading 2-29
expressions
aggregate C-12
allocator C-12
based literal C-12
function call C-12
null literal C-12
physical literal C-12
static expression C-12
string C-12
type conversion C-12
universal expression C-13
names
attribute C-7
indexed C-7
operator symbol C-7
selected C-7
simple C-7
slice C-7
operands
aggregate C-12
allocator C-12

based literal C-12
function call C-12
null literal C-12
physical literal C-12
static expression C-12
string C-12
type conversion C-12
universal expression C-13
operators
addition C-9
logical C-9
miscellaneous C-10
multiplying C-9
overloading 2-30, C-10
relational C-9
short-circuit operation C-10
signing C-9
predefined language environment
now function C-16
predefined attributes C-16
severity_level type C-16
TEXTIO package C-16
time type C-16
reserved words C-17
sequential statements
assertion C-13
case C-14
exit C-14
if C-14
loop C-14
next C-14
null C-14
procedure call C-14
report C-13
return C-14
signal C-13
statement labels C-13
variable C-13
wait C-13
specifications
attribute C-6
configuration C-6
disconnection C-6
latch
definition 7-1
latch inference
local variables 7-11
latches
edge-sensitive
not in guarded block statement 6-13
level-sensitive
guarded block statement 6-12
left attribute 3-12
length attribute 3-12
literal
enumeration
character, defined 3-3
identifier, defined 3-3
literals
as operands 4-26
bit strings 4-28
character 4-26
character string 4-28
enumeration 4-27
numeric 4-26
string 4-28
logic
combinational 8-2
logical operators 4-3
loop statement 5-22
syntax 5-23
low attribute 3-12

M

map_to_entity directive 5-45, 6-14
mapping
generic values
example 2-14
instantiation 2-14
port connections
example 2-15
expressions 2-15

Mealy finite state machine (example) A-5
mod (multiplying operator) 4-11
Moore finite state machine (example) A-2
multiple driven signals 6-8
multiplication using shifts B-16
multiplying operators 4-11

N

names C-7
  attributes 4-20
  field names 4-30
  qualified 4-30
  record names 4-30
  slice names 4-32
nand (logical operator) 4-3
NAND2 entity
  syntax
    dataflow architecture 2-34
    RTL architecture 2-34
    specification 2-3
    structural architecture 2-33
NATURAL subtype 3-17
N-bit adder
  declaration
    example 2-11
nested blocks
  in block statement 6-11
netlist
  defined 2-13
next statement
  in named loops 5-32
noncomputable operands 4-16
nor (logical operator) 4-3
not (logical operator) 4-3
null range 4-33
null slice 4-33
null statement 5-58
numeric literals 4-26
numeric_std package
” B-25, B-25
”*” function B-24
”+” function B-24
”/=” equality function B-26
”=” equality function B-26
”>=” ordering function B-25
”>” ordering function B-25
”-” function B-24
accessing B-21
arithmetic functions
  binary B-23
  binary example B-24
  unary B-23
  unary example B-24
comparison functions
  equality B-26
  ordering B-25
conversion functions
  TO_INTEGER B-22
  TO_UNSIGNED B-22
  UNSIGNED B-22
data types
  SIGNED B-22
  UNSIGNED B-22
IEEE documentation B-1
location B-21
logical operators
  AND B-26
  NAND B-26
  NOR B-26
  NOT B-26
  OR B-26
  XNOR B-26
  XOR B-26
report_design_lib command B-21
resize function B-23
rotate functions B-28
rotate operators B-28
shift functions
  ROTATE_LEFT B-27
  ROTATE_RIGHT B-27
  SHIFT_LEFT B-27

SHIFT_RIGHT B-27
shift operators B-28
unsupported components B-21
use with std_logic_arith package B-20

O

octal bit string 4-28
one_cold attribute 7-7
one_hot attribute 7-7
operands
aggregates 4-18
attributes 4-20
bit width 4-15
computable 4-16
defined 4-14
field 4-30
function call 4-22, 5-41
identifiers 4-23
in expressions
defined 4-1
grouping 4-5
integer
predefined operators 4-8
literal 4-26
character 4-26
enumeration 4-27
numeric 4-26
string 4-28
noncomputable 4-16
qualified expressions 4-29
record 4-30
slice names 4-32
supported C-12
type conversions 4-34
operators
absolute value 4-12
adding 4-8
arithmetic
adding 4-8
multiplying 4-11
negation 4-10
array
catenation 4-9
relational 4-6
catenation 4-9
described 4-2
equality 4-6
exponentiation 4-12
in expressions 4-1
logical 4-3
multiplying
predefined 4-11
restrictions on use 4-11
ordering 4-6
and array types 4-6
and enumerated types 4-6
overloading 2-30
defined 2-30
examples 2-30
precedence 4-3
predefined 4-2
relational
described 4-5
std_logic_arith package 4-7
rotate C-11
numeric_std B-28
shift C-10
numeric_std B-28
sign 4-10
supported C-9
unary 4-10
xnor C-11
optimization
arithmetic expressions 8-6
NAND2 gate 2-33
or (logical operator) 4-3
ordering
operators 4-6
ordering functions
example B-14
others (in aggregates) 4-20
others (in case statement) 5-17
overflow characteristics

arithmetic optimization 8-10
overloading
enumeration
literal 3-4
enumeration literals 4-27
operators 2-30
defined 2-30
resolving by qualification 4-30
subprograms 2-29
defined 2-29

P

package
body syntax 2-39
component declaration in 2-35
constant declaration in 2-35
declaration
example 2-38
syntax 2-37
defined 2-35
numeric_std
IEEE documentation B-1
package_body_declarative_item 2-39
STANDARD 3-17
std_logic_arith 4-7
std_logic_signed 3-9
std_logic_unsigned 3-9
structure
body 2-36
declaration 2-36
subprogram in 2-35
TEXTIO 3-16
type declaration in 2-35
use statement syntax 2-36, 2-38
package_body_declarative_item 2-39
package_declarative_item 2-37
package_name 2-37
packages
Synopsys-supplied B-1
parameters, subprogram
actual 2-26
formal 2-26
PLA (example) A-51
port
as signal 2-21
connections, mapping example 2-15
map 2-13
mode
buffer 2-4, 2-24
entity port specification 2-4, 2-24, 2-25
in 2-4, 2-24, 2-25
inout 2-4, 2-24
out 2-4, 2-24
name
consistency among entities 2-9, 2-12
incorrect use 2-9
type
consistency among components 2-12
POSITIVE subtype 3-17
pragma keyword comment
hdlin_pragma_keyword variable 9-2
pragmas.*See* directives
predefined attributes
array 3-12
supported C-7
predefined attributes, supported C-16
predefined language environment C-16
predefined VHDL operators 4-3
procedure
call (defined) 2-17
call syntax 5-39
subprogram declaration syntax 2-23
subprogram description 5-38
process
as algorithm 2-19
declaration 2-19
defined 2-19
description 2-19
sequential statements in 2-19
process statement 6-2, 6-10
processes
asynchronous 6-3

combinational
example 6-5
combinational logic 5-55
sensitivity lists 6-3
sequential
example 6-6
sequential logic 5-55
synchronous 6-3
wait statement 5-50

Q

qualified expressions 4-29

R

range attribute 3-12
range_type_name 3-11
record aggregates 3-14
record data type 3-13
record operands 4-30
records
as aggregates C-12
register
definition of 7-1
inference 7-1
register inference
attribute
async_set_reset 7-5
async_set_reset_local 7-5
async_set_reset_local_all 7-5
one_cold 7-7
one_hot 7-7
sync_set_reset 7-6
sync_set_reset_local 7-6
sync_set_reset_local_all 7-6
D latch 7-10
definition 7-1
edge expressions 7-22
if statement 7-23
if versus wait 7-23
signal edge 7-22
SR latch 7-8
templates 7-3
wait statement 7-22
wait versus if 7-23
relational operators 4-5
rem (multiplying operator) 4-11
report statement C-13
reserved words C-17
resize function
numeric_std B-23
resolution function
allowed 2-41
creating 2-40
directive, using 2-41
directives
resolution_method three_state 2-41
resolution_method wired_and 2-41
resolution_method wired_or 2-41
example 2-42
marking 2-41
signal 2-40
syntax
declaration 2-40
subtype 2-41
type 2-40
resolution functions
bus 6-9
resolution_method
three_state directive 2-41
wired_and directive 2-41
wired_or directive 2-41
resolved signal
creating 2-42
example 2-42
subtype declaration 2-40
syntax 2-41
using 2-42
return statement 5-43
return_port_name directive 5-45
reverse_range attribute 3-12

right attribute 3-12
rising_edge 7-22, 7-26
ROM (example) A-7
rotate functions
  numeric_std B-28
rotate operators C-11
  numeric_std B-28
RTL Analyzer
  architecture
    NAND2 entity 2-34

S

selected signal assignment
  equivalent process 6-22
  syntax 6-20
sensitivity lists 6-3
sequential processes 5-55, 6-6
sequential statement
  if, syntax 5-15
sequential statements
  supported C-13
serial-to-parallel converter (example)
  counting bits A-40
  shifting bits A-47
shared variable C-5
sharing
  common subexpressions
    automatically determined 8-12
shift functions
  example B-15
  numeric_std B-27
shift operations
  example B-16
shift operators C-10
  numeric_std B-28
signal
  as port 2-21
  assignment 2-18
    examples 5-11, 5-13
    syntax 5-12
  declaration 2-21
    example 3-4
    logical 4-4
  in package 2-37
  multiple drivers
    bus 6-9
  resolution function 2-40
  resolved 2-40
signals
  concurrent signal assignment 6-17
  conditional signal assignment 6-18
  drivers 6-8
  edge detection 7-22
  registering 7-54
  selected signal assignment 6-20
  supported C-5
  three-state 6-8
SIGNED data type
  defined B-6, B-7
  std_logic_arith package B-4
SIGNED data types
  numeric_std package B-22
simple name target 5-3
simulation
  dont care values 8-33
  driver example 9-3
slice names
  limitations 4-33
  syntax 4-32
slice target syntax 5-7
specifications C-6
STANDARD package 3-17
state machine (example)
  controller A-24
  Mealy A-5
  Moore A-2
statement
  assignment
    aggregate target, syntax 5-9
    field target, syntax 5-8
    indexed name target, syntax 5-4

slice target, syntax 5-7
case
enumerated type 5-18
invalid usages 5-21
syntax 5-17
concurrent
block 2-17
component instantiation 2-17
procedure call 2-17
process 2-17, 2-19
signal assignment 2-18
for ... loop
syntax 5-25
loop
syntax 5-23
loop syntax 5-22
sequential
assignment, syntax 5-2
if 5-15
while ... loop syntax 5-24
statement labels C-13
std_logic_1164 package B-2
std_logic_arith package B-2, B-3
_REDUCE functions B-30
”*” function B-12
”+” function B-11
”/=” function B-14
”<=” function B-14
”<” function B-14
”=” function B-14
”>=” function B-14
”>” function B-14
”–” function B-11
arithmetic functions B-10
Boolean reduction functions B-30
built_in functions B-6
comparison functions B-13
CONV_INTEGER functions B-8
CONV_SIGNED functions B-9
CONV_UNSIGNED functions B-8
conversion functions B-10
data types B-6
modifying the package B-5
ordering functions B-14
shift function B-15
using the package B-4
std_logic_misc package B-30
std_logic_signed package 3-9
std_logic_unsigned package 3-9
string literals 4-28
bit 4-28
character 4-28
STRING type 3-17
structural architecture
NAND2 entity 2-33
structural design
component
instantiation statement 2-13
description
COUNTER3 2-15
subexpressions in expression tree 8-9
subprogram
body
calls, examples 2-26
examples 2-29
function syntax 2-28
procedure syntax 2-26
declaration
examples 2-25
function syntax 2-24
overloading 2-29
procedure, syntax 2-23
syntax 2-28
overloading
defined 2-29
examples 2-29
parameter 2-26
profile 2-29
sequential statement 2-22
subprograms
calling 5-37
defined 5-35
defining 5-36
mapping to components

example 5-47
matching entity 5-45
procedure versus function 5-38
subrange
integer data type 3-8
subtype data type
declaration 2-32
defining 3-21
SYN_FEED_THRU
example of using B-20
sync_set_reset attribute 7-6
sync_set_reset_local attribute 7-6
sync_set_reset_local_all attribute 7-6
synchronous
designs 8-23
example 8-23
processes 6-3
synopsys keyword comment
hdlin_pragma_keyword variable 9-2
Synopsys packages B-1
std_logic_misc package B-30
Synopsys-defined package
std_logic_arith 3-21
std_logic_signed
integers 3-9
overload for arithmatic 3-9
std_logic_unsigned
integers 3-9
overload for arithmatic 3-9
syntax
array data type
constrained 3-10
unconstrained 3-11
assignment statement
aggregate target 5-9
field target 5-8
indexed name target 5-4
signal 5-2, 5-12
simple name target 5-3
slice target 5-7
variable 5-2, 5-11
bused clock 7-22
case statement 5-17
clock, bused 7-22
component
declaration statement 2-10
instantiation statement 2-13
constant declaration 2-18
enumeration data type 3-3
for ... loop statement 5-25
generic_declaration 2-3
if statement 5-15
integer data type 3-8
loop statement 5-22, 5-23
NAND2
dataflow architecture 2-34
RTL architecture 2-34
specification 2-3
structural architecture 2-33
operator
overloading 2-30
package body 2-39
resolution function
declaration 2-40
subtype 2-41
type 2-40
signal declaration 2-21
subprogram
overloading 2-29
subprogram declaration
body, examples 2-29
body, function syntax 2-28
function 2-24
procedure 2-23
procedure body 2-27
subtype 2-32
type
declaration 2-30
use statement, package 2-36
variable declaration 2-20, 2-31
while ... loop statement 5-24
synthetic comments
hdlin_pragma_keyword variable 9-2

synthetic comments.*See* directives

T

target
  signal assignment syntax 5-3
  variable assignment syntax 5-2
TEXTIO package 3-16
three-bit counter
  circuit description
    entity architecture 2-7
    entity specification 2-7
three-state
  gate 7-65
    registered enable 7-67
    without registered enable 7-68
  inference 7-59
  registered drivers 7-65, 7-67
  registered input 7-67
  signals 6-8
tick (') in VHDL expressions 4-29
time type C-16
TO_INTEGER function
  conversion
    numeric_std B-22
TO_SIGNED function
  conversion
    numeric_std B-22
TO_UNSIGNED function
  conversion
    numeric_std B-22
translate_off directive, warning 9-3
translate_on directive, warning 9-3
transport keyword C-15
two-input AND gate
  component declaration example 2-11
two-input NAND gate
  dataflow architecture syntax 2-34
  RTL architecture syntax 2-34
  specification syntax 2-3
  structural architecture 2-33
two-input N-bit comparator
  example 2-5
two-phase design 7-20
type
  conversion
    syntax 4-34
types
  as aggregates C-12

U

unary arithmetic functions
  example B-12
  numeric_std B-23
unary operators
  sign 4-10
unconstrained arrays
  example using A-17
unconstrained data array 3-9, 3-10
UNSIGNED data type
  defined B-6
  std_logic_arith package B-4
UNSIGNED data types
  numeric_std package B-22
use statement 2-36

V

variable
  assignment
    examples 5-11, 5-13
    syntax 5-2, 5-11
  declaration 2-20, 2-31
    defined 2-20, 2-31
    example 3-4
  initializing 2-20, 2-31
variables
  conditionally-assigned 7-19
  hdlin_pragma_keyword 9-2
vectors
  encoding
    ENUM_ENCODING attribute 3-6

VHDL
aggregates 4-18
architectures 2-5, 6-1
array data type 3-9
BIT data type 3-18
BIT_VECTOR data type 3-19
BOOLEAN data type 3-18
case statement 5-17
CHARACTER data type 3-18
component
implication 5-46
instantiation 2-13
concurrent procedure call 6-14
concurrent statement
block 6-1
process 6-1, 6-2, 6-10
concurrent statements
supported C-15
configuration 2-34
data type supported
enumeration C-4
data type unsupported
integer C-4
data type, supported
enumeration 3-3
integer 3-8
data type, unsupported
access (pointer) types 3-20
file (disk file) types 3-20
floating-point 3-20
physical 3-20
declarations C-5
design units C-3
directives 9-2
enumeration data type 3-3
errors in descriptions 8-38
exit statement 5-33
expressions, supported C-12
for ... loop statement 5-23, 5-25, 6-27
functions 2-22
generate statement 6-26
identifiers 4-23
integer data type 3-8, 3-19
keywords C-17
literals 4-26
names C-7
NATURAL subtype 3-19
null statement 5-58
operands
categories 4-14
supported C-12
operators
precedence 4-3
predefined 4-2
supported C-9
package
composition 2-35
use statement syntax 2-36
port modes 2-4, 2-24, 2-25
POSITIVE subtype 3-19
predefined attributes, supported C-7
predefined data types 3-16
predefined language environment C-16
procedures 2-22
process statement 6-2, 6-10
qualified expressions 4-29
record data type 3-13
reserved words C-17
return statement 5-43
sensitivity lists 6-3
sequential statements, supported C-13
shorthand expressions 8-22
specifications C-6
STANDARD package 3-17
STRING type 3-19
subprograms 2-22, 5-35
subtype data type 3-3, 3-21
TEXTIO package 3-16
three-state components 7-59
type conversion 4-34
wait statement 5-50
VHDL Analyzer
in synthesis process 8-38
VHDL assertions 7-8

VHDL Compiler
  asynchronous designs 8-23
  attributes
    supported C-7
    Synopsys C-7
  ATTRIBUTES package 3-5
  component
    consistency 2-12
    implication 5-46
    instantiation, entities 2-14
  design efficiency 8-22
  design structure 8-3
  directives 9-2
  dont care information 8-29
  entities
    consistency 2-12
  enumeration encoding 3-5
  integer encoding 3-8
  operators
    supported C-9
  port names
    consistency 2-12
  sensitivity lists 6-3
  source directives 9-1, 9-2
  syntax checking 8-38
  wait statement
    limitations 5-54
    usages 5-50

W

wait statement 5-50
  creating registers 7-22
  example 5-56
    multiple waits 5-52
    while loop 5-52
waveform generator (example)
  complex A-13
  simple A-10
while ... loop statement syntax 5-24

X

xnor (logical operator) 4-3
xnor operator C-11
xor (logical operator) 4-3