Test Vector Guidelines

In order to stimulate a device off board, a series of logical vectors must be applied to the device inputs. These vectors are called test vectors and are mostly used to stimulate the design inputs and check the outputs against the expected values. In other words, on a tester, the test vectors replace the HDL testbench used by designers to verify the design functionality. In case of a functional failure analysis, Actel uses test vectors to duplicate the failure of the device under test. Since antifuse FPGAs cannot be reprogrammed, the quality of the test vectors is critical to ensure a complete and timely failure analysis. Due to the tester constraints, the test vectors are not fully capable of resembling the signal conditions on the real board. Hence, the vectors should be generated carefully to duplicate the observed misbehavior of the silicon. This document provides very specific details and guidelines to generate an appropriate set of test vectors that can be used by Actel or customers to duplicate the functional failure on the FPGA while the device is taken off the board and placed on the tester.

Actel requires customers to follow a standard format generating their test vectors. A standardized format helps customers and Actel in the following ways:

- Improves simulation by reducing errors
- Simplifies design verification on automatic test equipment
- Reduces project cycle times
- Reduces debugging costs

This application note provides guidelines for test vector sets. These guidelines are are recommended for functional verification of designs in Actel FPGAs, and they cover several aspects of design verification, including:

- Waveform file formats
- Timing and sampling
- Tester constraints
- Handling bidirectional signals
- Simultaneously-switching outputs (SSOs)

Vector File Formats

The vector file contains binary signals corresponding to all ports of the design programmed in the device. Each line of the file forms a vector, which indicates the state of the ports at a given time. Therefore each line in the file must be associated with a time that is increasing ***uniformly*** with each simulation cycle.

The vector files should be ASCII files in a tabular format. There are two main formats for the test vectors. First, a non-return-to-zero (NRZ) format that implies two lines in the file associated with each cycle. The other option is to use either a return-to-zero (RZ) or a return-to-one (RTO) format. In these cases there is a single line per cycle. The format of a signal cannot be changed within a given vector file. However, different signals of the given vector can follow different formats. For example, in a given vector, the clock signal may be in RZ format while the data inputs follow the NRZ format. It is the customer's responsibility to highlight these details for Actel to avoid any false error during analysis or vector verification.

Individual entries in each line should use the following guidelines:

- Inputs – '0', '1', 'Z', or 'X'
- Outputs – 'L', 'H', 'Z', or 'X'
- Bidirectionals
  - When used as an input – '0', '1', 'Z', or 'X'
  - When used as an output – 'L', 'H', 'Z', or 'X'

Since signal entries in existing customer files may not conform to the above guidelines, Actel can provide routines to automatically translate ASCII vector files. Additionally, conversion of the vector file entries into

the above guidelines can be embedded in the vector generation HDL code. Routines for the data format conversion in the HDL code are presented in "Vector Verification" on page 10. A sample of a vector file in NRZ format is given in Figure 1.

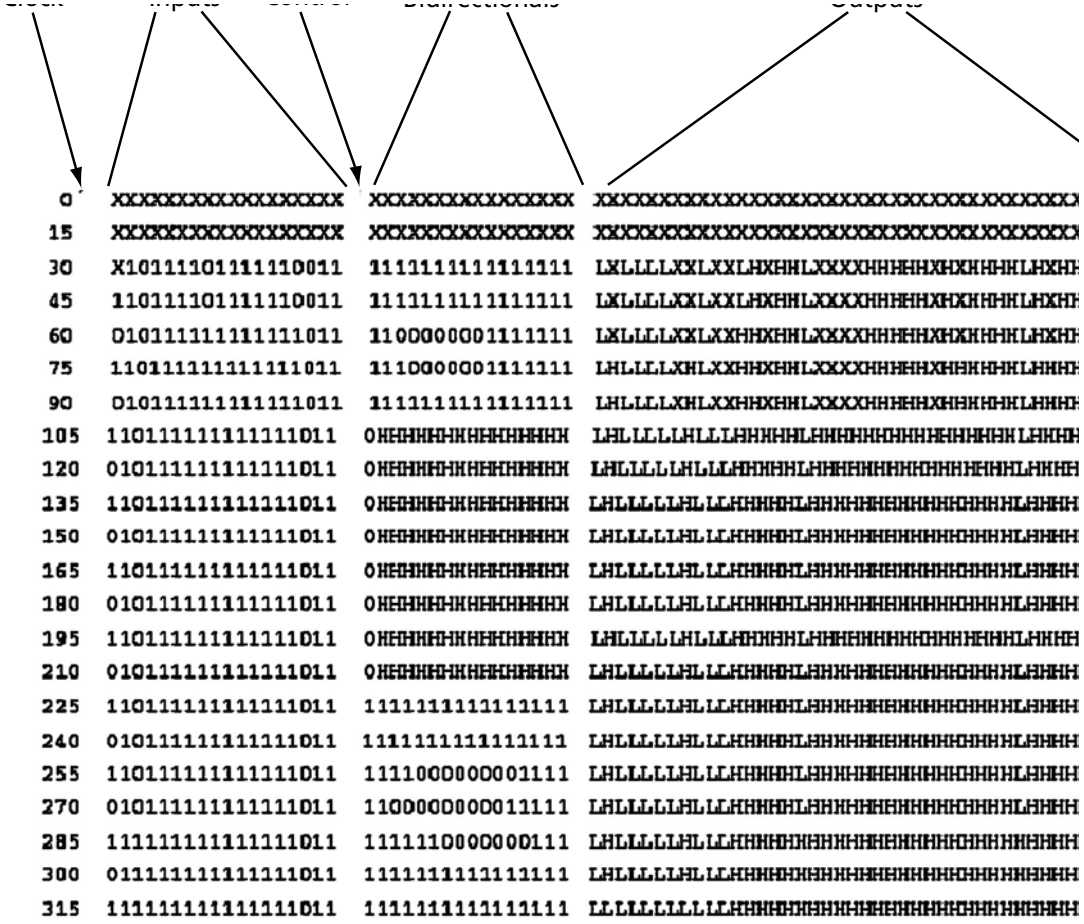

Figure 1 • **An Example Vector File in NRZ Format**

Tester Constraints

The vector files are applied to a FPGA using one of several different types of testers at Actel. The testers limitations along with general hardware considerations imply constraints which need to be taken into account during the test vector generation.

Reference Clock and Asynchronous Inputs

The tester activities such as applying input signals and reading back the outputs of the design are synchronous to a reference clock which is extracted from the vector file. Therefore *synchronous* vector files are *always* required. Actel recommends customers to put the reference clock signal as the first signal in the vector file.

Since the vector files must be always synchronous, care should be taken in generating vectors for the asynchronous signals. The asynchronous inputs are applied to the device synchronous to the reference clock edge and asynchronous outputs are sampled and checked synchronously.

As with asynchronous inputs, users should take care if multiple clock inputs to the design are asynchronous with respect to each other. Due to the synchronous nature of the vector files, the asynchronous clocks may be affected by missing edges with respect to the reference clock. As a result, clocks must be synchronized or errors will occur.

In case of multiple clocks input to the design, the best candidate for the reference clock varies from design to design and also depends on the module or signal that needs to be analyzed during the test. Considering the possibility of missing edges of other clocks, customers should select one of the clocks in the design as the reference clock that serves the best in stimulating and analyzing the target signal. For the purpose of vectors (duplicating silicon functional failure), frequencies are often not important. In such cases, the customers may consider changing the asynchronous clocks frequencies to avoid missing clock edges in the vectors with respect to the reference clock. If users choose to do so, the reference clock must be the fastest clock.

Timing and Sampling

The key functions of vector generation are the stimulation and sampling of signals in the vector file. Figure 2 illustrates a simple logic network with a clock, an input signal, and an output signal. The output signal changes as a result of a change in the input signal. This change is reflected after the next (positive) edge of the clock in the circuit shown. The assertion of IN must be far enough in advance of the rising edge of the CLK signal to allow the signal to propagate through the input combinatorial logic and also meet the data input's setup requirements relative to the clock edge. In addition, there is a tester-specific requirement. The test equipment has pin-to-pin skew in the order of a few nanoseconds, and accounting for this skew is difficult in simulations. After the flip-flop, as shown in Figure 2, there may also be combinatorial logic. The sampling of an output (for comparison with its expected value in a vector file) must be done after the value has had time to settle.

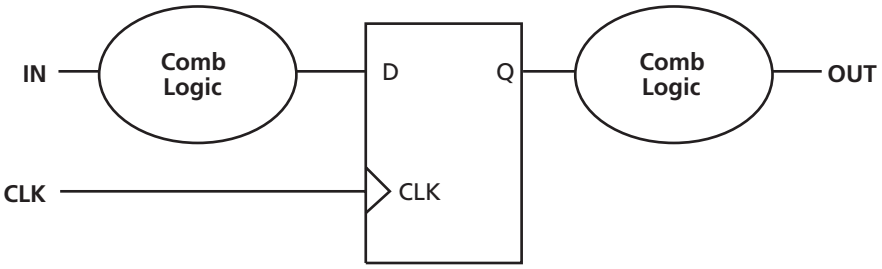

Figure 2 • **Sample Logic Circuit**

Test vectors of the sample logic circuit in Figure 2 should be generated considering the timing of the circuit. Figure 3 on page 4 shows the relative timing of the CLK, IN, and OUT signals. The time t_{setup} corresponds to the minimum recommended assertion time of IN before the rising edge of CLK. Similarly, t_{sample} is the minimum time after the rising edge of CLK when the OUT signal can be accurately sampled. The value of t_{setup} should be large enough to avoid any setup time violation. The external setup and hold time requirement of the design defines value of t_{setup}. The external set and hold time requirements for each input can be obtained by exporting a timing report from Designer software with the setup and hold option checked. The value of t_{sample} should be greater than the largest clock-to-out delay of the design. Furthermore, customers should avoid problems with tester pin-to-pin skew (an output sampled when an input for the NEXT cycle is asserted), the assertion and sampling points should be 5 ns or more apart to account for the tester pin-to-pin skew.

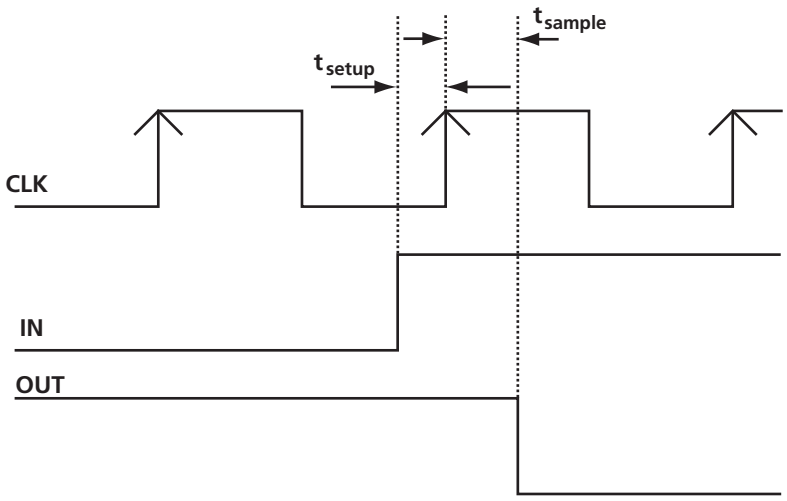

Figure 3 • **Timing for the Sample Logic Circuit**

Bidirectional I/Os and Contention

An additional timing constraint is imposed by bidirectional signals. Contention occurs when the tester switches to drive a pin as an input *while the same pin is already acting as a driver*. Avoiding contention is important. Since the tester is unable to both drive and not drive on a given cycle, contention can cause physical damage to the device under test. The bidirectional control signals should be adjusted so that I/O drive direction changes occur just before (or at) the cycle boundary under worst-case speed conditions. Furthermore, considering the propagation delay of control signals of bidirectional buffers, users should make sure that the bidirectional ports are defined as inputs before starting to drive them by tester to avoid any type of contention on the line.

Bidirectional I/O control signals should be used in vector generation for proper format and timing of the bidirectional signals considering that during each cycle of the reference clock, each I/O is either input or output. Handling bidirectional I/Os during vector generation is discussed in details in "Vector Verification" on page 10. However, if the bidirectional I/Os of the design are not related to the purpose of the test vectors, it is recommended to configure them as either inputs or outputs throughout the vector file if possible.

Tester Hardware Constraint

If possible, at-speed simulation is recommended. Current Actel test equipment supports clock frequency range from 100 kHz (minimum) to about 75 MHz (maximum). The resolution in setting the tester period is currently 1ns. This applies to all design verification of Actel FPGAs on Actel test equipment. In addition, there are other hardware constraints associated with particular vector formats as shown in Table 1.

Table 1 • **Additional Tester Hardware Constraints**

Parameter	Value	NRZ	RZ, RTO
Delay - Min	0ns	Y	Y
Delay – Max	Cycle	Y	Y
Pulse Width – Min	5ns	N/A*	Y
Output Strobe Width – Min	5ns	Y	Y
Output Strobe Width	Cycle	Y	Y

**Not applicable since the pulse width cannot be less than half cycle.*

Simultaneously Switching Outputs

Noise voltages in test fixtures can sometimes be increased with large numbers of Simultaneously Switching Outputs (SSOs). To avoid triggering unintended device state changes, the number of SSOs in vector files should be minimized. Customers can reduce the number of SSO during the test by avoiding unnecessary output switching. This can be done controlling the enable signals to disable the unnecessary sections of the design. Note that the test vectors are usually generated to highlight a functional failure by stimulating a specific internal node or output of the design.

Vector Generation

In test vector generation, the input vectors must be designed to stimulate the logic module or the I/O that needs to be monitored. In simple designs in which it is easy to toggle the desired module by stimulating few inputs, users may write a simple test bench to do so. Furthermore, the simplest designs can be tested on a bench board. In such cases, instead of test vectors, only stimulus instructions are required. For example the stimulus instructions can be as simple as the following:

Pull up input1 of the design, pull the rest of the inputs low and toggle the clock.

However, in a complicated design, where it is difficult to relate the inputs to the destination internal node or I/O directly, the easiest way to generate test vectors is to generate them from the test bench. Note that the verification test bench should naturally stimulate the desired logic, so it should create accurate vectors easily. The vector generation consists of repeating few fundamental steps: A text file should be opened; the inputs are sampled at the beginning of each cycle; the outputs are sampled after being settled; the collection of the sampled signals is written in the file.

A test bench usually contains the following blocks under the top level module:

- Stimulus module
- Instantiation of the design top module (referred to as “DUT” throughout this document)
- Special Routines

The following should be added to the test bench for vector generation flow:

- Top-level routines for:
  - Check clock generation
  - Generating time stamp (optional)
  - Character conversion to tester format
- Processes for:
  - Formatting each line of the Vector file
  - Converting each bit of each signal to tester format
  - Writing completed vector line into the file

Customers can apply the testbench to the RTL-level design or gate-level netlist to create the test vectors, assuming that the back-annotated simulation results are consistent with RTL simulation results. The following sections provide specific instructions for generating test vectors from a test bench.

Opening the Vector File

The vector file should also be opened in the testbench during the vector generation to save the test vectors. The file can be opened using the following commands:

VHDL: `file TestFile: text open write_mode is “test_vector.txt”;`

Verilog: `TestFile = $fopen(“test_vector.txt”);`

Note that in VHDL language, the user should refer to the following library to be able to use the above instruction.

```
library std;
use std.textio.all;
```

Creating Input and Output Vectors

A vector should be defined in the testbench to accommodate all the inputs of the design. This vector is used as a temporary storage of the input vector line before writing them into the file. Therefore the width of this vector should be equal to the sum of the input pins of the device. All the inputs should be concatenated into this vector. The following examples of the vector assignment in VHDL and Verilog:

VHDL: `IN_VECTOR <= sysclk & clk_1 & rst_n & vs_ext & ram_addr & ram_data;`

Verilog: `IN_VECTOR <= {sysclk, clk_1, rst_n, vs_ext, ram_addr, ram_data};`

In the above example, if sysclk, clk_1, rst_n and vs_ext are one-bit and ram_addr and ram_data are 8-bit wide each, the IN_VECTOR register should be defined with the width of 20.

Similarly, the outputs of the device should be stored in a temporary register. The temporary register should be as wide as the output pins of the design. All the outputs are concatenated into this vector similar to outputs. The bidirectional signals will be discussed in “Handling Bidirectional Ports” on page 9.

Format Conversion

As mentioned in “Vector File Formats” on page 1, Actel requires specific format for test vector files as follows:

- Inputs – '0', '1', 'Z', or 'X'
- Outputs – 'L', 'H', 'Z', or 'X'

The vectors, created in the previous section, can be converted in this format before being written in the file. The conversion is performed by the following functions in the code:

Input vector format conversion (VHDL):

```
function IN_VAL(v:std_logic_vector) return string is
--Variables to hold the converted format value
variable temp_string: string(1 to v'length);
-- Variable to hold the length of the input vector
variable vl : natural := v'length;
begin
-- Loop to span the input vector and convert to the input vector format
   for i in v'range loop
      if (v(i) ='1') or (v(i)='H') then
         temp_string(vl-i) := '1';
      elsif (v(i)='0') or (v(i)='L') then
         temp_string(vl-i) := '0';
      elsif (v(i)='Z') then
         temp_string(vl-i) := 'Z';
      else
         temp_string(vl-i) := 'X';
      end if;
   end loop;
```

```
   return(temp_string);
end function IN_VAL;
```

Passing IN_VECTOR, created in the previous section, through the IN_VAL function will convert the IN_VECTOR contents into the Actel input vector format.

Output vector format conversion (VHDL):

```
function OUT_VAL(v:std_logic_vector) return string is
--Variables to hold the converted format value
variable temp_string: string(1 to v'length);
-- Variable to hold the length of the input vector
variable vl : natural := v'length;
begin
-- Loop to span the input vector and convert to the input vector format
   for i in v'range loop
      if (v(i) ='1') or (v(i)='H') then
         temp_string(vl-i) := 'H';
      elsif (v(i)='0') or (v(i)='L') then
         temp_string(vl-i) := 'L';
      elsif (v(i)='Z') then
         temp_string(vl-i) := 'Z';
      else
         temp_string(vl-i) :=  'X';
      end if;
   end loop;
   return(temp_string);
end function OUT_VAL;
```

The above functions can be included in the vector generation code (user testbench) to perform the format conversion on the fly. However, users can generate the vector file without using the format conversion functions and change the format of the file later using a text editor. This approach is preferred in the Verilog flow in which the format conversion is somewhat more complicated.

Input and Output Sampling Time

The inputs of the design, residing in the input vector defined in the test bench (IN_VECTOR in the previous sections), can be sampled and be written at each edge (or active edge) of the reference clock. However, the input sampling time may vary from design to design and it is up to the customer to choose the best time to sample the inputs. The best sampling time for inputs can be determined by examining the input signal activities near the edge of the reference clock and looking into the design external setup and hold time. The sampled inputs should contain the values that determine the state of the outputs before the next active edge of the clock. Note that on the tester, the inputs will be applied to the device synchronous to the edge of the reference clock.

Before sampling, enough time should be given to the outputs of the design to settle to their final value after sampling the inputs (or after the active edge of the reference clock). This time interval should be equal or greater than the largest clock to out delay of the design. The output settlement time interval can be defined by creating a virtual clock in the testbench. This virtual clock should be have the same frequency as the reference clock and delayed by the output settlement time. The outputs can be sampled at the active edge of the virtual clock which should occur before the next active edge of the reference clock. Figure 4 on page 8 illustrates the virtual clock and sampling times in a vector generation code where the virtual clock and reference clock are labeled as CHK_CLK and REF_CLK respectively.

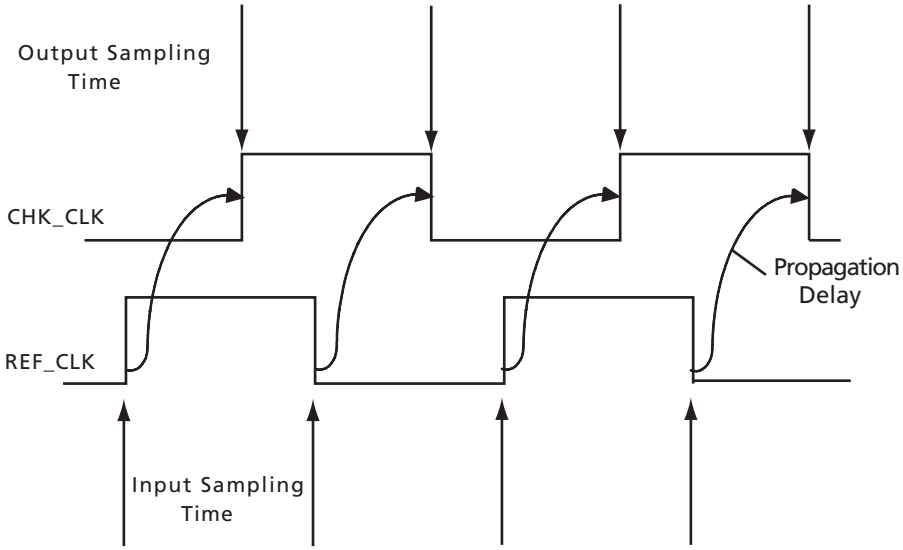

Figure 4 • **Input and Output Sampling Time With Respect to the Reference Clock and Virtual Clock**

The propagation delay value is design dependent and customers must sample the outputs when they have reached their final value after the active edge of the reference clock. Note that if the inputs do not change till the time that outputs are sampled, they can be sampled at the same time as the outputs (active edge of the virtual clock).

For example, assume that the reference clock of a design (labeled as REF_CLK) runs at 40MHz in the testbench. A similar code to the following should exist in the test bench:

```
begin
   while (TRUE) loop
      REF_CLK <= '1';
      wait for (25 / 2) * ns;
      REF_CLK <= '0';
      wait for (25 / 2) * ns;
   end loop ;
   wait;
end process;
```

If in this example design, the outputs take 15ns to settle to their final values after the active edge of the REF_CLK, the following process can be used to create the virtual clock for sampling the outputs:

```
process
     begin
   wait for 15 ns;
   while (TRUE) loop
      CHECK_CLK <= '1';
         wait for (25 / 2) * ns;
      CHECK_CLK <= '0';
      wait for (25 / 2) * ns;
   end loop ;
   wait;
end process;
```

Or

```
CHECK_CLK <= REF_CLK after 15 ns;
```

Writing Vectors into The File

The input and output vectors in the test bench contain the line information of the vector file. The sampling time, discussed in the previous section, can be the same time that the vectors are being written into the file. The input vector can be sampled at the active edge of the reference clock and written into the file along with the output vector at the active edge of the virtual clock. The following is an example of writing the input and output vectors with the converted format into the file at the rising edge of the check clock.

```
process (check_clk)
variable BufLine: line;

begin
if (rising_edge(check_clk)) then
   write(Bufline,IN_VAL(in_vector)); --IN_VAL function in “Format Conversion” section
   write(Bufline,string'("    ")); -- Space between the input and output vectors
   write(Bufline,OUT_VAL(out_vector)); --OUT_VAL function in “Format Conversion”
   section
   writeline(TestFile,Bufline);
end if;
end process;
```

Handling Bidirectional Ports

The drive direction of the bidirectional ports is defined by their bidirectional I/O buffer control signal. As mentioned in “Tester Hardware Constraint” on page 4, the tester is unable to both drive and not drive on a given cycle. Therefore control signals should be monitored during the test vector generation whether they are internal signals or external inputs to the design. When none of the external ports can indicate the status of the bidirectional I/O control signals at each time, the control signals should be brought out to the test bench level via creating virtual output ports in the top level of the design and connect the control signals to the ports. By doing this, the control signal can be read and monitored in the test bench level for test vector generation.

The bidirectional signals in the testbench can be stored in a vector, defined in the testbench. Then the control signals should be monitored in each cycle of the reference clock and if they indicate the direction as output the vector should be formatted with the output vector format. Otherwise, the bidirectional I/Os are configured as inputs in that cycle and the vector is formatted with the input vector format.

As an example consider a design which has an 8-bit bidirectional bus. This bus is stored in an 8-bit wide vector, named bidi1_vector in the testbench. The direction of the bidirectional bus is controlled by a signal called control. The following process defines the format of the bidi1_vector and writes it into the vector file according to the direction of the I/Os.

```
process (check_clk)
variable BufLine: line;
begin
if (rising_edge(check_clk)) then
     if (control = '1') then
        write(Bufline,OUT_VAL(bidi1_vector));
     else
        write(Bufline,IN_VAL(bidi1_vector));
     end if;
     write(Bufline,string'("    "));
     writeline(TestFile,Bufline);
```

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

Supplementary Material

In addition to the test vectors the customers should provide the following along with the test vectors:

- Test Setup
  - Power supply values
  - Input levels Vih and Vil
  - Output levels Voh and Vol
  - Clock cycle time and the format (NRZ/RZ/R1/DNRZ etc.) and timing of rising edge and falling edge for RZ and R1 and delay for DNRZ format.
  - Regular input data format (Normally NRZ or DNRZ). No edge time is required for NRZ format.
- Pin List: The pin list contains the Port names, Ports pin numbers and their type (in/out/bidirectional and I/O standards) if possible. It is absolutely required to identify the type of the bidirectional ports in the pin list. The following is an example of a pin list:

```
CMNCLK          21
CPBTESTL        41
CMNBLREQL       107
CMNRWLY         29          bidir
CMNRWLX         36
CMNREQLY        3
CMNREQLX        1
CMNRESETLY      19
```

In the vector files, column 1 MUST represent the vector for the first port in the pin list, column 2 must represent for second line of the pin list and so forth.

- Design Information:
  - Probe files exported from Designer using the design ADB file
  - Timing report from Designer containing the external setup and hold time requirements

Vector Verification

The generated vectors should be functionally verified before being applied to the real silicon on the tester. Due to the real time delays on the silicon, the generated output vectors may not match the outputs of the silicon. This is similar to simulating the design functionality without back-annotating the delays after place and route. The mismatch may happen if the back-annotated delays (SDF file) are not incorporated in the vector generation. Therefore, the test vectors should be verified using the back-annotated delays (SDF file) to simulate the real time delays on silicon. In vector verification flow, the vectors are read from the vector file. The input vectors are applied to the design as force vectors and the output of the design is compared with the output vectors.

Since the test vectors will be applied to the real silicon with the real time delays, it is recommended that the back-annotated netlist and SDF files be used in vector verification instead of the RTL level design. Similar to the vector generation the timing of applying inputs and sampling outputs is important. In simulation for vector verification, there should be enough time intervals between applying the inputs and checking the outputs.

The following routines should be included in the top-level of the verification code:

- Reference (and check) clock generation
- Format conversion from the tester format

The necessary processes in the code are:

- Reading vector file

- Converting the format from the tester format by calling the appropriate routines.
- Creating "force" vectors to be applied to the design and "observe" vectors for comparison.

Reading Vector Files

The existing vector file can be read using the following commands:

VHDL: file `Tasteful: text open ready-made is "test_vector.txt";`

Verilog: `$readmemb("test_vector.txt",vector_i);`

Note that in the VHDL coding, the file is being written line by line in opposite to the Verilog format where all the contents of the file is being read once. As a result, in Verilog flow, an array of the vector file dimensions should be created in the code to store all the information of the vector file. This is discussed more in "Practical Example of Test Vector Generation Flow" on page 11 where a Verilog code flow example is presented.

Practical Example of Test Vector Generation Flow

All the routines and techniques, presented in the previous sections of this application note, are demonstrated in this section. A top level of a general design is presented in this section along with a testbench used to verify this design. The number of ports of the design is limited to few ports in order to avoid confusion and unnecessary complication, however all three types (output, input and bidirectional) are included in the example.

VHDL Flow Example

The VHDL code of the example design is located in Appendix A of this document. Figure 4 on page 8 summarizes the design top-level ports and their direction.

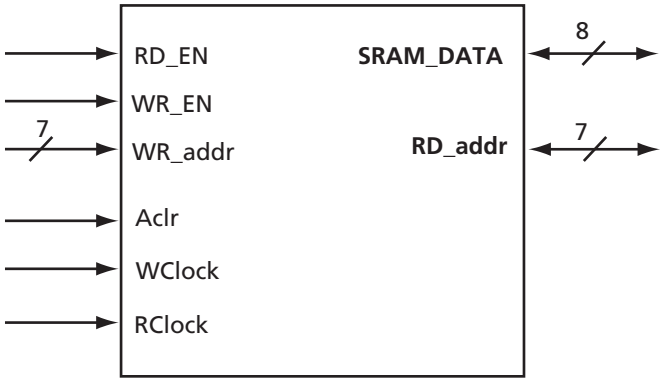

Figure 5 • **Example Design I/O Ports**

Appendix B contains the test vector generation code. Applying the vector generation code to the source code in Appendix A will result in a vector file in NRZ format as the following:

```
011100000000    00000001        LLLLLLL
101100000000    00000001        LLLLLLL
001110000000    00000001        LLLLLLL
110110000000    00000001        LLLLLLL
010110000000    00000001        LLLLLLL
100110000000    00000001        LLLLLLL
............
000110010001    ZZZZZZZZ        LLLLLLL
111110010001    ZZZZZZZZ        LLLLLLL
```

```
011110010001    ZZZZZZZZ        LLLLLLL
101110010001    ZZZZZZZZ        LLLLLLL
001110010001    ZZZZZZZZ        LLLLLLL
111010010001    XXXXXXXX        LLLLLLH
011010010001    XXXXXXXX        LLLLLLH
............
```

In the above vectors, the first set are the inputs, the middle set is bidirectional bus (SRAM_DATA) and the third set contains the design outputs (RD_addr). The following is the pin list of the design:

```
WClock              N5
   RClock           M8
   WR_EN            M1
   RD_EN            M5
   Aclr             N4
   WR_addr(6)       H3
   WR_addr(5)       K1
   WR_addr(4)       G4
   WR_addr(3)       J2
   WR_addr(2)       N1
   WR_addr(1)       K2
   WR_addr(0)       J3
   SRAM_DATA(7)     K3 bidir
   SRAM_DATA(6)     P2 bidir
   SRAM_DATA(5)     M3 bidir
   SRAM_DATA(4)     M2 bidir
   SRAM_DATA(3)     N2 bidir
   SRAM_DATA(2)     N3 bidir
   SRAM_DATA(1)     L2 bidir
   SRAM_DATA(0)     L1 bidir
   RD_addr(6)       P5
   RD_addr(5)       N9
   RD_addr(4)       M7
   RD_addr(3)       P4
   RD_addr(2)       M4
   RD_addr(1)       P3
   RD_addr(0)       L3
```

The code in Appendix C can be used to verify the generated vectors.

Verilog Flow Example

In the Verilog flow example, a Verilog equivalent of the Design in Figure 5 on page 11 is used as the design under test. Appendix D includes the Verilog format of the design under test. In the Verilog flow example, three files are generated: input vector (bvec.in), output vector (bvec.out) and bidirectional vector (bvec.bi). The format conversion needs to be done outside the Verilog code. In the bidirectional vector file an additional string is added to each line as indication to the direction of the vector: “in” or “out”. This string can be used during the format conversion but should be removed from the file afterward. The code in Appendix E is used to generate the three vector files. After executing the vector generation code on the design files, three set of files will be generated. The following are the examples of the vectors in each file:

bvec.in:

```
111100000000
011100000000
101110000000
000110000000
............
```

bvec.out:

```
0000000
0000000
0000000
0000000
```

bvec.bi:

```
zzzzzzzz    in
zzzzzzzz    in
00000010    in
00000010    in
........
00001001    out
00001001    out
00001001    out
xxxxxxxx    out
xxxxxxxx    out
xxxxxxxx    out
xxxxxxxx    out
```

After generating the files, bvec.out and bvec.bi should be run through a format conversion script. However, for the bvec.bi file, only the vectors, indicated by "out" should be converted. After finishing the format conversion, the files can be glued together in format similar to the VHDL flow vector file. A similar pin list should be generated too to accompany the vector file.

The sanity of the generated vectors can be verified by the code in Appendix F. In this code, three generated files are being read. The inputs are applied to the design inputs and outputs of the design are registered to be compared against the expected values in the vectors. The bidirectional vectors may act as inputs or outputs according to the state of the control signals.

The following Appendices provide examples of testbenches which generate test vector files and simulate test vectors against a design.

Appendix A

Top level of the example design:
Appendix_A_1.vhd
The definition of the memory block instantiation in top level:
Appendix_A_2.vhd

Appendix B

VHDL testbench with code to generate test vectors:
Appendix_B.vhd

Appendix C

VHDL testbench to read in test vectors and apply them to the design:
Appendix_C.vhd

Appendix D

Verilog flavor of the example design top level:
Appendix_D_1.v
The definition of the memory block instantiation in top level:
Appendix_D_2.v

Appendix E

Verilog testbench with code to generate test vectors:
Appendix_E.v

Appendix F

Verilog testbench to read in test vectors and apply them to the design:
Appendix_F.v

http://www.actel.com

Actel Corporation
2061 Stierlin Court
Mountain View, CA
94043-4655 USA
Tel: (650) 318-4200
Fax: (650) 318-4600

Actel Europe Ltd.
Dunlop House, Riverside Way
Camberley, Surrey GU15 3YL
United Kingdom
Tel: +44 (0)1276 401450
Fax: +44 (0)1276 401490

Actel Japan
EXOS Ebisu Bldg. 4F
1-24-14 Ebisu Shibuya-ku
Tokyo 150 Japan
Tel: +81 03-3445-7671
Fax: +81 03-3445-7668

Actel Hong Kong
39th Floor
One Pacific Place
88 Queensway
Admiralty, Hong Kong
Tel: 852-22735712

5192684-1/09.03