REQUIREMENTS FOR A SYNCHRONOUS FIFO,
First-In First-Out Buffer

Document #: fifo_req_001
Release Date: __
Revision Number: ____
Revision Date: __
Originator
Name: ____
Phone: ___
email: __

Approved:
Name:
Phone:
email:

--
Revisions History :
Date:
Version:
Author:
Description:
Synchronous FIFO to be used as an IP. FIFO management (e.g., push, pop, error handling) is external to the FIFO.

1. SCOPE

1.1 Scope

This document establishes the requirements for an Intellectual Property (IP) that provides a synchronous First-In First-Out (FIFO) function.

The specification is primarily targeted for component developers, IP integrators, and system OEMs.

1.2 Purpose

These requirements shall apply to a synchronous FIFO with a simple interface for inclusion as a component. This requirement includes SystemVerilog assertions to further clarify the properties of the FIFO.

1.3 Classification

This document defines the requirements for a hardware design.

2. DEFINITIONS

2.1 PUSH

The action of inserting data into the FIFO buffer.

2.2 POP

The action of extracting data from the FIFO buffer

2.3 FULL

The FIFO buffer being at it maximum level.

2.4 EMPTY

The FIFO buffer with no valid data.

2.5 Read and Write Pointers

Pointers represent internal structure of the FIFO to identify where in the buffer data will be stored (write pointer, *wr_ptr*), or be read (read pointer, *rd_ptr*)

3. APPLICABLE DOCUMENTS

3.1 Government Documents

None

3.2 Non-government Documents

None

4. ARCHITECTURAL OVERVIEW

4.1 Introduction

The FIFO component shall represent a design written in SystemVerilog with SystemVerilog assertions. The FIFO shall be synchronous with a single clock that governs both reads and writes. The FIFO typically interfaces to a controller for the synchronous pushing and popping of data. Figure 4.1 represents a high level view of the interfaces.

FIFO Requirements Example (continued)

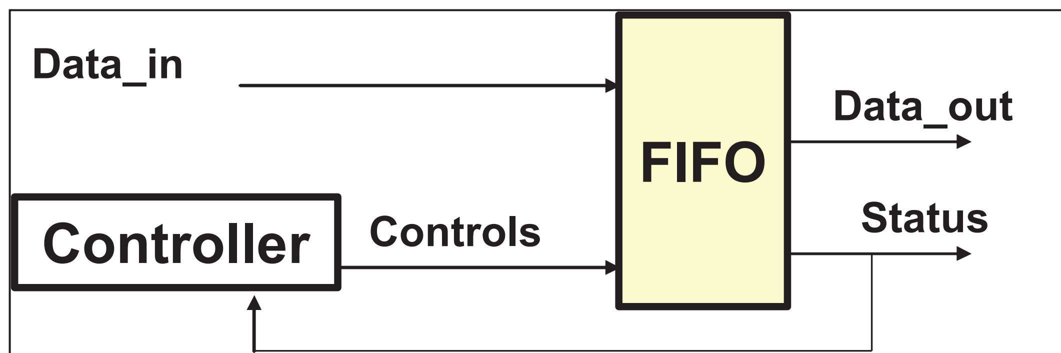

Figure 4.1 High Level View of the FIFO Interfaces

The FIFO shall include the following features:
1. Parameterized storage space for data buffers
2. Parameterized data widths for the data.
3. Flag information for FULL, EMPTY, ALMOST FULL at the ¾ level, ALMOST EMPTY at the ¼ level.
4. A synchronous RESET capability.

4.2 System Application

The FIFO can be applied in a variety of system configurations. Figure 4.2-1 demonstrates one such configuration where the FIFO interfaces on one side to a bus controller, and on the other side to a different controller. All buses use the same system clock. It is the responsibility of the enqueue/dequeue controller to manage the integrity of quantity of data transferred into and extracted out of the FIFO.

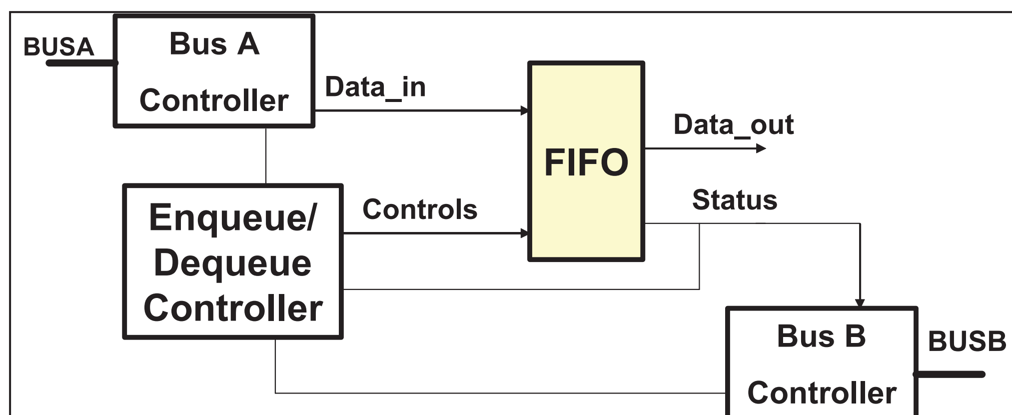

Figure 4.2-2 Hardwired Application of a FIFO

FIFO Requirements Example (continued)

5. PHYSICAL LAYER

The physical hardware interfaces shall be as shown in Figure 5.0

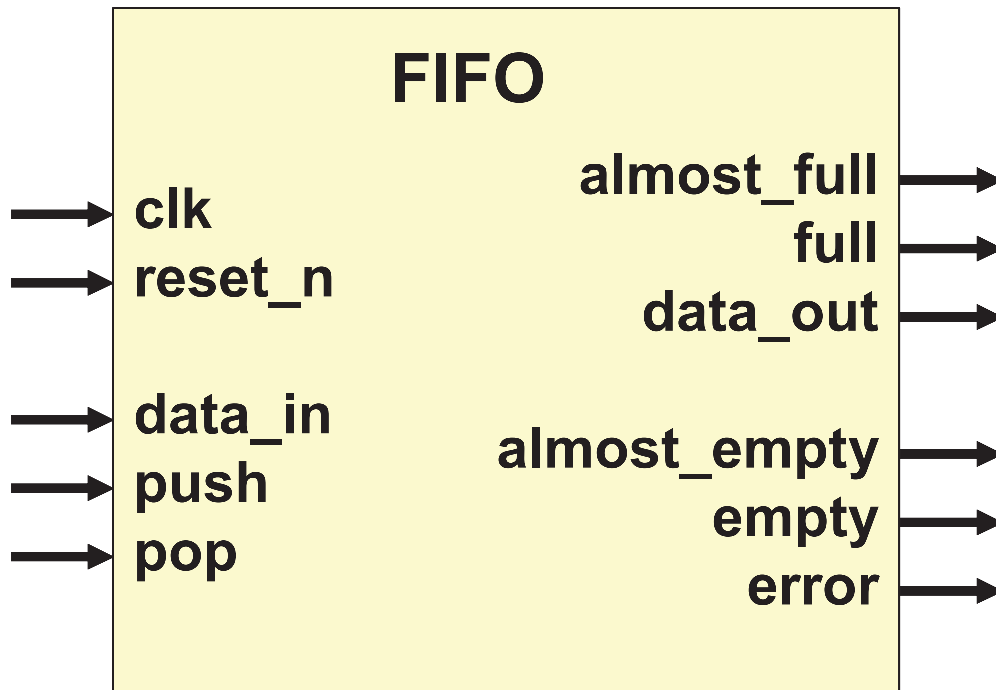

Figure 5.0 Interfaces of the FIFO

A SystemVerilog description of the interface is shown in Figure 5.1.

```
// PACKAGE for type and parameter definitions
// ch4/fifo_queue/fifo_pkg.sv
package fifo_pkg;
  timeunit 1ns;
  timeprecision 100ps;
  localparam BIT_DEPTH = 4; // 2**BIT_DEPTH = depth of fifo
  localparam FULL = int'(2** BIT_DEPTH -1);
  localparam ALMOST_FULL = int'(3*FULL / 4);
  localparam ALMOST_EMPTY = int'(FULL/4);
  localparam WIDTH = 32;
  typedef logic [WIDTH-1 : 0] word_t;
  typedef word_t [0 : 2**BIT_DEPTH-1] buffer_t;
  // Other types for testbench support can be added here
endpackage :  fifo_pkg
//17
```

FIFO Requirements Example (continued)

[17] The complete SystemVerilog interface with assertions is in file ch4/fifo_if.sv

```
// INTERFACE of FIFO
interface fifo_if(input clk, reset_n);
  import fifo_pkg:: *; // access to package
  logic push; // push data into the fifo
  logic pop;  // pop data from the fifo
  logic almost_full;  // fifo is at 3/4 maximum level
  logic almost_empty;  // fifo is at 1/4 maximum level
  logic full;  // fifo is at maximum level
  logic empty; // fifo is at the zero level (no data)
  logic error; // fifo push or pop error
  word_t data_in;
  word_t data_out;

// FIFO DUV, FIFO Slave interface
  modport fslave_if (
                 output empty,
                 output almost_empty,
                 output almost_full,
                 output full,
                 output data_out,
                 input data_in,
                 input push,
                 input pop);

  // FIFO driver, FIFO Driver interface
   modport fdrvr_if (
                 output data_in,
                 output push,
                 output pop,
                 input empty,
                 input almost_empty,
                 input almost_full,
                 input full,
                 input data_out);
 // tasks / sequences / properties / assertions shall be added here (see Section 5.1)
endinterface : fifo_if
```


Figure 5.1 SystemVerilog FIFO Interface

5.1 Interface Port Description

The individual port elements in the interface in figure 5.1 are described in this section with requirements on them captured as assertions. Since some of the ports describe data intensive portion of the system (such as the data being popped from the FIFO), some of the SystemVerilog testbench features such as queues and tasks are used to capture their requirements. Since these tasks and queues are meant solely for the purpose of specification and verification, and do not have a direct correlation to the hardware implementation of the FIFO, they are

FIFO Requirements Example (continued)

declared in the interface itself:

```
// Data queue for verification.
// Queue, maximum size is 2**BIT_DEPTH
  word_t dataQ [$:2**BIT_DEPTH-1];
// Data read from queue
  word_t data_fromQ;

// Push and Pop tasks
task pop_task;
   begin
     data_in <= 'X; // unsized Xs
     pop  <= 1'b1;
     data_fromQ <= dataQ.pop_front();
     @ (posedge clk);
    end
endtask : pop_task

task push_task (word_t data);
     begin
       $display ("%0t %m Push data %0h ", $time, data);
       data_in <= data; //data to be written
        push <= 1'b1;
        dataQ.push_back(data); // push to dataQ
        @ (posedge clk);
      end
 endtask : push_task

task idle_task(int num_idle_cyles);
  begin
    push <= 1'b0;
    pop <= 1'b0;
    data_in <= 'X;
    assert (num_idle_cycles < 10000) else
      $warning ("%0t %0m idle_task is invoked with LARGE number of idle
cycles %0d ", num_idle_cycles);
    repeat (num_idle_cycles) @ (posedge clk);
  end
endtask : idle_task
```

Data Queue

Input data stored into FIFO buffer 1 cycle following the push

Use immediate assertion for simple, local checks.

FIFO Requirements Example (continued)

5.1.1 Data Input/Output
Figure 5.1.1 provides a timing diagram of the interface.

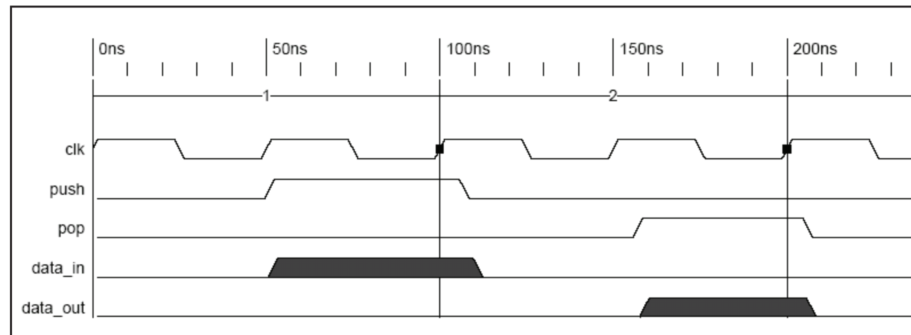

Figure 5.1.1 FIFO Interface Timing Diagram

5.1.1.1 Data_In
Direction: Input, Peripheral -> FIFO;
Size: Determined by WIDTH parameter; Active level: High
Data sent from a peripheral device to the FIFO under the control of the *push* control.
5.1.1.2 Data_Out
Direction: Output, FIFO -> Peripheral;
Size: Determined by WIDTH parameter; Active level: *High*
FIFO data sent to a peripheral device under the control of *pop* signal.

5.1.2 Push / Pop
5.1.2.1 push
Direction: Input, Peripheral -> FIFO; Size: 1 bit, Active level: high
When *push* is active, *data_in* shall be stored into the FIFO buffer at the next clock cycle. It is an error if a push with no pop control occurs on a full FIFO. The following property characterizes these requirements:

```
// never a push and full and no pop
  sequence q_push_error;
    !(push && full && !pop);
  endsequence : q_push_error
  ap_push_error : assert property (@(posedge clk) q_push_error);
```

5.1.2.2 pop
Direction: Input, Peripheral -> FIFO; Size: 1 bit, Active level: high
When *pop* is active, *data_out* shall carry the data that was first stored into the FIFO, but was not yet popped. The *data_out* shall be asserted in the same cycle of *pop* control. It is an error if a pop control occurs on an empty FIFO. The following properties and task characterize these requirements:

// Data out timing and data integrity

FIFO Requirements Example (continued)

```
property p_pop_data;
   @ (posedge clk)
        pop |-> data_out == data_fromQ;
        // from 5.1  pop_task
endproperty : p_pop_data
ap_pop_data : assert property (p_pop_data);

// never a pop on empty
 sequence q_pop_error;
     ! (pop && empty && !push);
endsequence : q_pop_error
ap_pop_error : assert property (@ (posedge clk) q_pop_error);
```

5.1.2.3 Push-Pop Data Sequencing

Data entered into the FIFO buffer shall be outputted in the same order that it is entered. The *push_task* and *pop_task* tasks, and the properties characterized in sections 5.1.2.1 and 5.1.2.2 define the ordering sequence. Specifically, data pushed in the back of the FIFO buffer is extracted from the front of the buffer in a first-in, first-out manner.

5.1.3 Status Flags
5.1.3.1 Full
Direction: Output, FIFO -> Peripheral ; Size: 1 bit, Active level: high
When the FIFO reaches the maximum depth of the buffer, as defined by the parameter BIT_DEPTH, then the *full* flag shall be active. The following sequence and property characterize this requirement:

```
sequence qFull;
 @ (posedge clk)
   dataQsize == BIT_DEPTH;
endsequence : qFull

property p_fifo_full;
   @ (posedge clk)   qFull |-> full;
endproperty : p_fifo_full
ap_fifo_full : assert property (p_fifo_full);
```

```
int dataQsize; // queue size
assign dataQsize=dataQ.size;
// See section 8.2.7 for
// guidelines on using dynamic
// data types inside properties.
```

5.1.3.2 Almost Full
Direction: Output, FIFO -> Peripheral ; Size: 1 bit, Active level: high
When the number of entries in the FIFO reaches or is greater than the predefined value of ¾ of the maximum depth of the buffer, as defined by the parameter ALMOST_FULL, then the *almost_full* flag shall be active. The following sequence and property characterizes this requirement:

```
sequence qAlmost_full;
 @ (posedge clk)
   dataQsize >= ALMOST_FULL;
endsequence : qAlmost_full
```

FIFO Requirements Example (continued)

```
property p_fifo_almost_full;
   @ (posedge clk)   qAlmost_full |-> almost_full;
endproperty : p_fifo_almost_full
ap_fifo_almost_full : assert property (p_fifo_almost_full);
```

5.1.3.3 Empty

Direction: Output, FIFO -> Peripheral ; Size: 1 bit, Active level: high

When all the enqueued data has been dequeued, then the *empty* flag shall be active. A reset shall cause the empty flag to be active. The following sequence and properties characterize these requirements:

```
// sequence definition, use in cover for empty
 sequence qEmpty;
  @ (posedge clk)
     dataQsize==0;
 endsequence : qEmpty

 property p_fifo_empty;
   @ (posedge clk)   qEmpty |-> empty;
 endproperty : p_fifo_empty
 ap_fifo_empty : assert property (p_fifo_empty);
```

The property for the flags at reset time is defined in section 5.1.4.

5.1.3.4 Almost Empty

Direction: Output, FIFO -> Peripheral ; Size: 1 bit, Active level: high

When the number of entries in the FIFO reaches or is less the predefined value of ¼ of the maximum depth of the buffer, as defined by the parameter ALMOST_EMPTY, then the *almost_empty* flag shall be active. The following sequence and property characterize this requirement:

```
 sequence qAlmost_empty;
   @ (posedge clk)    dataQsize <= ALMOST_EMPTY;
 endsequence : qAlmost_empty

 property p_fifo_almost_empty;
   @ (posedge clk)  qAlmost_empty |-> almost_empty;
 endproperty : p_fifo_almost_empty
 ap_fifo_almost_empty : assert property (p_fifo_almost_empty);
```

5.1.4 Reset

Direction: Input, Peripheral -> FIFO ; Size: 1 bit, Active level: low

The *reset_n* is an active low reset control that clears the pointers and the status flags. The *reset_n* is asynchronous to the system clock *clk*. See properties defined in section 5.1.3.3 for the behavior of the empty flag when *reset_n* is asserted in the FIFO.

```
 property p_fifo_ptrs_flags_at_reset;
   @ (posedge clk)
     !reset_n |-> ##[0:1] !almost_empty && ! full && !almost_full && empty;
 endproperty  : p_fifo_ptrs_flags_at_reset
 ap_fifo_ptrs_flags_at_reset : assert property (p_fifo_ptrs_flags_at_reset);
```

FIFO Requirements Example (continued)

5.15 Clock
Direction: Input, Peripheral -> FIFO ; Size: 1 bit, Active edge: rising edge
The *clk* clock is the synchronous system clock for the FIFO for both the read and write transactions, active on the positive edge of the clock. The clock shall be at 50% duty cycle.

5.16. Error
Direction: Output, FIFO -> Peripheral; Size: 1 bit, Active level: high
When either an overflow (push on full) or underflow (pop on empty) error has occurred, the error flag shall be asserted. The following properties characterize the *error* output.

```
// Reusing the q_push_error and q_pop_error definitions,
property p_error_flag;
  @ (posedge clk)
        q_push_error or q_pop_error |=> error;
endproperty : p_error_flag
ap_error_flag : assert property (p_error_flag);
```

6. PROTOCOL LAYER
The FIFO operates on single word writes (*push*) or single word reads (*pop*).

7. ROBUSTNESS
7.1 Error Detection
The FIFO shall lump all overflow (push on full) or underflow (pop on empty) errors as a single error output. See section 5.16 for details.

8. HARDWARE AND SOFTWARE
8.1 Fixed Parameterization
The FIFO shall provide the following parameters used for the definition of the implemented hardware during hardware build:

BIT_DEPTH where 2**BIT_DEPTH represents the depth of FIFO.
WIDTH represents the data width.
ALMOST_FULL (0.75 * (2 ** BIT_DEPTH))
ALMOST_EMPTY (0.25 * (2 ** BIT_DEPTH))

8.2 Software Interfaces
The FIFO shall enter input data (*data_in*) into the FIFO buffer when the *push* control is active. It shall provide data from the buffer upon an activation of the *pop* control. See section 5.1.2 Push / Pop for definition of the properties that characterize these controls. The FIFO contains no internal registers that can be configured.

This section typically contains the internal registers that the software can access and configure.

9. PERFORMANCE
9.1 Frequency
The FIFO shall support a maximum rate of 25 MHz.

FIFO Requirements Example (continued)

9.2 Power Dissipation
The power shall be less than 0.01 watt at 25 MHz.

9.3 Environmental
Does not apply.

9.4 Technology
The design shall be adaptable to any technology because the design shall be portable and defined in SystemVerilog RTL.

10. TESTABILITY
None required.

11. MECHANICAL
Does not apply.

12. Backup Information
A copy of the FIFO interface model and supporting package is included in the download files.

FIFO Requirements Example (continued)

6.3.2 Verification Plan

The following demonstrates the application of assertions in a verification plan to clarify the verification goals and milestones.

Header page	**VERIFICATION PLAN FOR SYNCHRONOUS FIFO, First-In First-Out Buffer**
Pertinent logistics data about the requirements. *Conform to company policies and style*	**Document #: fifo_ver_plan_001** **Release Date: __/__/__** **Revision Number: ____** **Revision Date: __/__/__** **Originator** **Name:** **Phone:** **email:** **Approved:** **Name:** **Phone:** **email:**
	--- **Revisions History:** **Date:** **Version:** **Author:** **Description:** Describes verification approaches for the FIFO buffer. **---** …

FIFO Verification Plan Example

1. SCOPE

1.1 Scope

This document establishes the verification plan for the FIFO design specified in the requirements specification. It identifies the features to be tested, the test cases, the expected responses, and the methods of test case application and verification. SystemVerilog Assertions properties specify characterizations that the design must meet, and test sequences that must be covered.

The verification plan is primarily targeted for component developers, IP integrators, and system OEMs.

1.2 Purpose

The verification plan provides a definition of the testbench, verification properties, test environment, coverage sequences, application of test cases, and verification approaches for the FIFO design as specified in the requirement specification number *fifo_req_001*, and in the implementation document number *fifo_dsgn_001*.[18]

The goals of this plan is not only to provide an outline on how the component will be tested, but also to provide a strawman document that can be scrutinized by other design and system engineers to refine the verification approach.

1.3 Classification

This document defines the test methods for a hardware design.

2 DEFINITIONS

2.1 BFM

A Bus Functional Model is a model that emulates the operation of an interface (i.e., the bus), but not necessarily the internal operation of the interface.

2.2 Transaction

Tasks that need to be executed to verify the device under test. An example of a transaction would be a push with specified DATA along with a simultaneous pop.

3. APPLICABLE DOCUMENTS

3.1 Government Documents

None.

3.2 Non-government Documents

Document #: *fifo_req_001*, Requirement Specification for a Synchronous FIFO.

FIFO Verification Plan Example (continued)

[18] The implementation document is not supplied because it is not within the scope of this book, which focuses on SystemVerilog Assertions rather than RTL design.

3.3 Executable specifications
Interface verification properties written in SystemVerilog, file *ch4/fifo_if.sv*.

. 3.4 Reference Sources
SystemVerilog 3.1a LRM[19].

4. COMPLIANCE PLAN
SystemVerilog with assertions along with simulation will be used as the verification language because it is an open language that provides good constructs and verification features. This plan consists of the following:

- Feature extraction and test strategy
- Test application approach for the FIFO
- Test verification approach

4.1 Feature Extractions and Test Strategy
The design features are extracted from the requirements specification. For each feature of the design, a test strategy is recognized. The strategy consists of directed and pseudo-random tests. A verification criterion for each of the design feature is documented. This feature definition, test strategy, test sequence, and verification criteria forms the basis of the functional verification plan. Table 4.1 summarizes the feature extraction and verification criteria for the functional requirements.

For corner testing, pseudo-random **push** and **pop** transactions will be simulated to mimic a FIFO in a system environment. The environment will perform the following transactions at pseudo-random intervals:

1. Create push requests
2. Create pop requests
3. Force resets

The properties specified in section 5 of the specification document will be used. Properties are also used to clarify the test sequences.

FIFO Verification Plan Example (continued)

[19] http://www.eda.org/sv/SystemVerilog_3.1a.pdf

Table 4.1 Feature Extraction and Verification Criteria

Tst #	FEATURE & DIRECTED TEST STRATEGY	SPEC #	Priority	TEST SEQUENCE / STRATEGY	VERIFICATION CRITERIA Interface verification SVA properties (*fifo_props.sv*) + the following
1	**Fixed Parameterization** - Bit_depth (FIFO buffer size) - data width	8.1, 5.1	1	**Configuration Setup** Buffer depth == 2**4, 2**8 Width==16, 32 Pseudo-random push and pop transactions. Unique data patterns using random values.	Design compiles and elaborates correctly. Simulation + monitoring of properties and coverage. Verification of output sequence (i.e., first-in is first-out)

2	**RESET**
	. Reset applied when fill state of FIFO is at different levels.
		5.1.4	1	**property** p_t1_full; @ (**posedge** clk)
				full |=> !reset_n;
				endproperty : p_t1_full
				
				property p_t2_afull; @ (**posedge** clk)
				almost_full |=> !reset_n;
				endproperty : p_t2_afull
				
				property p_t3_empty; @ (**posedge** clk)
				empty |=> !reset_n;
				endproperty : p_t3_empty
				
				property p_t4_a empty; @ (**posedge** clk)
				almost _empty |=> !reset_n;
				endproperty : p_t4_a empty
					Simulation + monitoring of properties and coverage.

Tst #	FEATURE & DIRECTED TEST STRATEGY	SPEC #	Priority	TEST SEQUENCE	VERIFICATION CRITERIA Interface verification SVA properties (*fifo_props.sv*) + the following
3	**COVERAGE** Using the properties and sequences defined in 4.1		1	**cover property** (p_push_pop_sequencing); **cover property** (qFull); **cover property** (qEmpty); **cover property** (qAlmost_empty); **cover property** (qAlmost_full); **cover property** (qOffFull); **cover property** (qOffEmpty); **cover property** (qOffAlmost_empty); **cover property** (qOffAlmost_full);	Sequences must all have a coverage count of at least one.

Page intentionally left blank.

4.2 Testbench Architecture
Several architectural elements must be considered in the definition of the testbench environment, including the following:

- Reusability / ease of use / portability / verification language
- Number of BFMs to emulate the separate busses
- Synchronization methods between BFMs
- Transactions definition and sequencing methods
- Transactions driving methods
- Verification strategies for design and its subblocks

Figure 4.2.1-1 represents the testbench architecture. The testbench makes use of the FIFO interface definition, FIFO package, and the FIFO property module. The testbench includes a *transactor* block to generate transactions such as **reset, push, pop**, and **idle** cycles. A set of server tasks provides the low level protocols to execute the transactions.

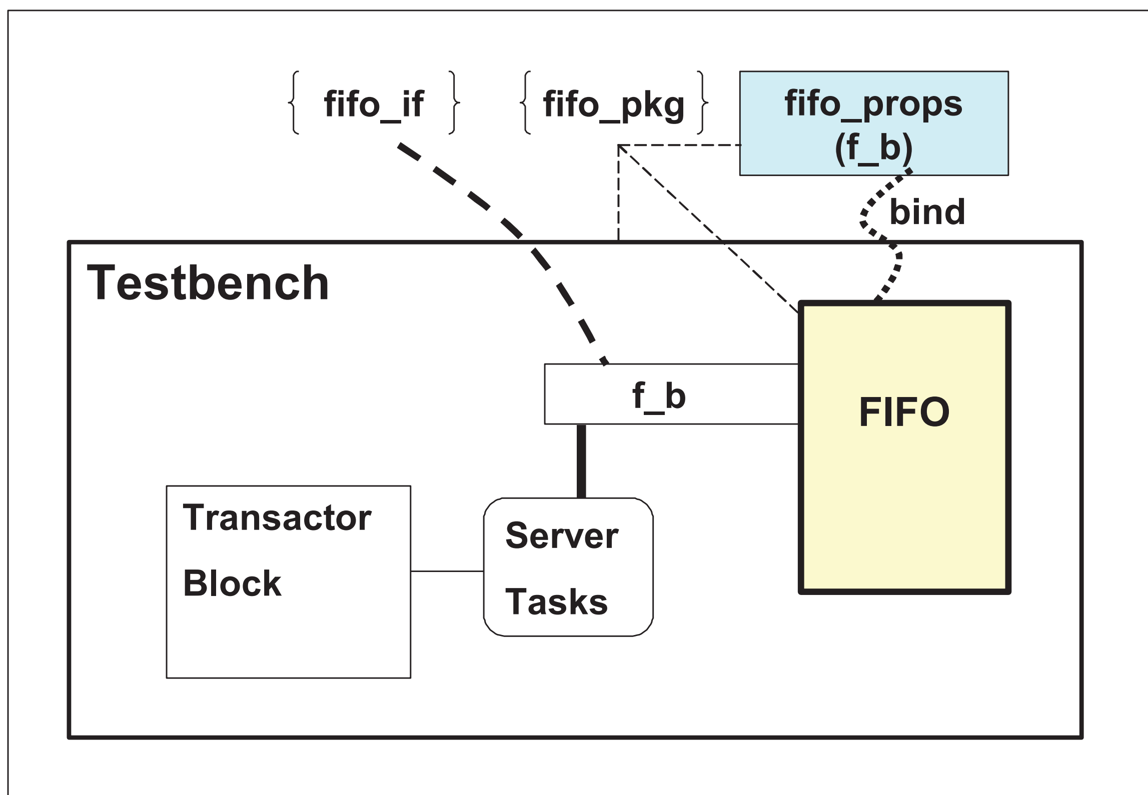

Figure 4.2.1-1 FIFO Testbench Architecture

SystemVerilog will be used for this design because it is a standard language, and is portable across tools. A reusable design style will be applied. A SystemVerilog package captures the common parameters for this design and is shown in Figure 4.2.1-2. A property module file is defined in Figure 4.2.1-2, for binding to the FIFO from within the testbench. An outline of the FIFO testbench that demonstrates the module instantiations and binding is shown in Figure 4.2.1-3

FIFO Verification Plan Example (continued)

```
// PACKAGE for type and parameter definitions
package fifo_pkg;
  timeunit 1ns;
  timeprecision 100ps;
  localparameter BIT_DEPTH = 4; // 2**BIT_DEPTH = depth of fifo
  localparameter FULL = int'(2** BIT_DEPTH -1);
  localparameter ALMOST_FULL = int'(3*FULL / 4);
  localparameter ALMOST_EMPTY = int'(FULL/4);
  localparameter WIDTH = 32;
  typedef logic [WIDTH-1 : 0] word_t;
  typedef word_t [0 : 2**BIT_DEPTH-1] buffer_t;

  // types supporting the testbench
    endpackage : fifo_pkg
```

Figure 4.2.1-2. Supporting Package (/ch4/fifo_queue/fifo_pkg.sv)

```
// --------------------------------------------------
// PROPERTY MODULE for FIFO
// This module is used for verification of the FIFO, and is
// intended to be bound (with the SystemVerilog "bind") to the DUV
module fifo_props (input clk, input reset_n, fifo_if fifo_if);
  import fifo_pkg::*;

  // Coverage points based on value of fifo fullness
// As specified in the Verification Plan, Table 4.1
  property p_t1_full;  @ (posedge clk)
    fifo_if.full |=> reset_n==0;
  endproperty : p_t1_full
  cp_t1_full_1: cover property (p_t1_full);

  property p_t2_afull;  @ (posedge clk)
    fifo_if.almost_full |=> reset_n==0;
  endproperty : p_t2_afull
  cp_t2_afull_1: cover property (p_t2_afull);

  property p_t3_empty;  @ (posedge clk)
     fifo_if.empty |=> reset_n==0;
  endproperty : p_t3_empty
  cp_t3_empty_1: cover property (p_t3_empty);

  property p_t4_aempty;  @ (posedge clk)
    fifo_if.almost_empty |=>  reset_n==0;
  endproperty : p_t4_aempty
  cp_t4_aempty_1 : cover property (p_t4_aempty);

  property p_push_pop_sequencing; @ (posedge clk)
     fifo_if.push => ##[0:$] fifo_if.pop;
  endproperty : p_push_pop_sequencing
```

FIFO Verification Plan Example (continued)

```
// coverage of sequences
// As specified in the Verification Plan, Table 4.1

  cp_push_pop_sequencing  : cover property  (p_push_pop_sequencing);
  c_qFull :               cover property  ( @ (posedge clk) fifo_if.qFull);
  c_qEmpty  :             cover property  (@ (posedge clk)  fifo_if.qEmpty);
  c_qAlmost_empty :       cover property  (@ (posedge clk)  fifo_if.qAlmost_empty);
  c_qAlmost_full :        cover property  (@ (posedge clk)  fifo_if.qAlmost_full);
endmodule : fifo_props
```

Figure 4.2.1-3 Property File for Inclusion with the Bind Construct (/ch4/fifo_queue/fifo_props.sv)

```
// FIFO  testbench Outline
module fifo_tb;
  timeunit 1ns;
  timeprecision 100ps;
   logic clk = 1'b0;   // system clock
   logic reset_n = 1'b0;

   import fifo_pkg::*; // Access to package information

   fifo_if b_if(.*);        // instantiation of fifo interface
   fifo_rtl fifo_rtl_1(.*);   // instantiation of fifo DUV

   // bind the fifo_rtl model to an implicit instantiation (fifo_props_1)
   // of property module fifo_props
   bind fifo fifo_props fifo_props_1(clk, reset_n, b_if);
   task reset_task(int num_rst_cycles);
   begin
    $display("%0t Resetting DUT for %0d cycles ", $time, num_rst_cycles);
     reset_n = 1'b0;
     repeat (num_rst_cycles) begin
        {b_if.push, b_if.pop} = $random % 2;
         b_if.data_in = $random;
        @ (posedge clk);
     end // repeat
     reset_n = 1'b1;
     b_if.push = 1'b0;
     b_if.pop = 1'b0;
  end
  endtask : reset_task

   // testbench code
   initial forever #50 clk = ~clk;
```

```
initial
    begin : client
    // directed tests
     reset_task(5);
    // 3 pushes
    for (int i=0; i<= 3; i++) begin
      // push_task($random % WIDTH);
     b_if.push_task($random );
     b_if.idle_task($random % 5);
     b_if.push_task(11 );
    end

    // 3 pop
    for (int i=0; i<= 3; i++) begin
      b_if.pop_task;
      b_if.idle_task($random % 5);
    end

   // push/pop random
    for (int i=0; i<= 5; i++) begin
      if ($random %2) begin
        b_if.push_task($random % WIDTH);
        b_if.idle_task($random % 3);
      end
      else begin
        b_if.pop_task;
        b_if.idle_task($random % 4);
      end
    end
    $stop;
    end // block: client
  endmodule : fifo_tb
```

FIFO Verification Plan Example (continued)

Figure 4.2.1-4 FIFO Testbench Outline (fifo_tb.sv)

5.0 Design Tools

This is beyond the scope of this book. This section typically defines the names of the linting, simulation, debugging, formal verification, and any other tool used in the verification process.

FIFO Verification Plan Example (continued)