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.

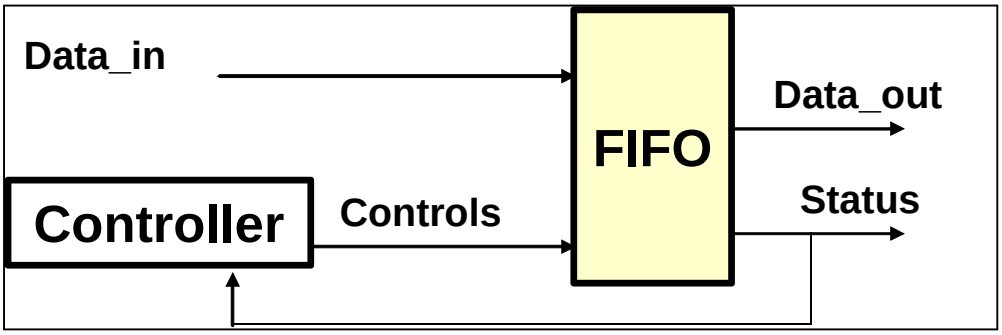

Figure 4.1 High level view of the FIFO interfaces

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.

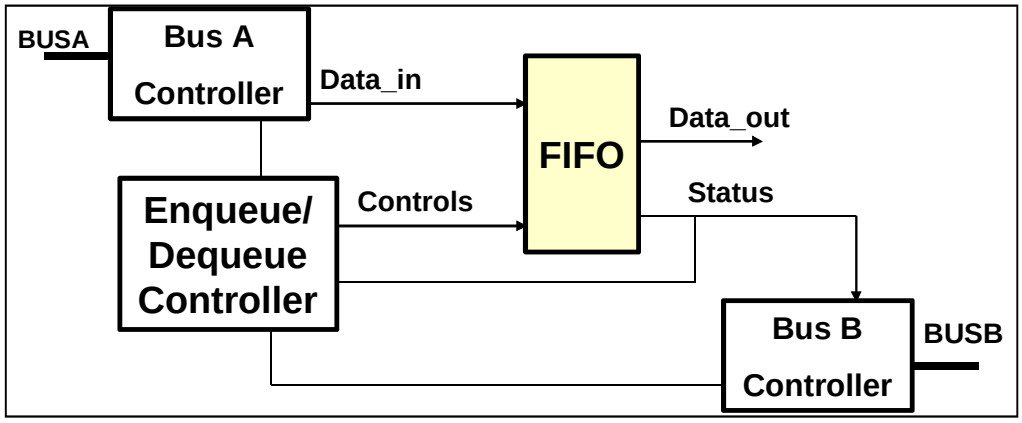

Figure 4.2-2 Hardwired Application of a FIFO

5. PHYSICAL LAYER

The physical hardware interfaces shall be as shown in Figure 5.0

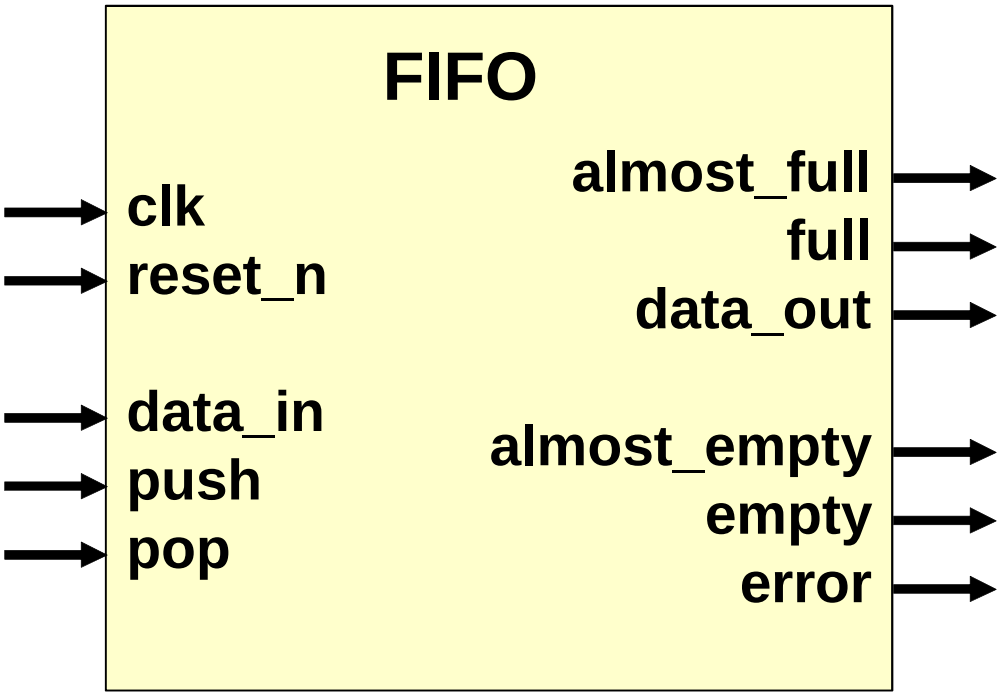

Figure 5.0 Interfaces of the FIFO

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

```
// PACKAGE for type and parameter definitions
/ //ch6/6.9/fifo_vmm
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
//58
```

58 The complete SystemVerilog interface with assertions is in file *ch6/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 SIave 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.
endinterface : fifo_if
```

Interface description with modports clarifies the use of the ports. Maintain ordering convention: outputs first, inputs last.

Used by application interfaced to the FIFO

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 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 control.

Use immediate assertion for simple, local checks.

5.1.1 Data input/output

Figure 5.1.1 provides a timing diagram of the interface.

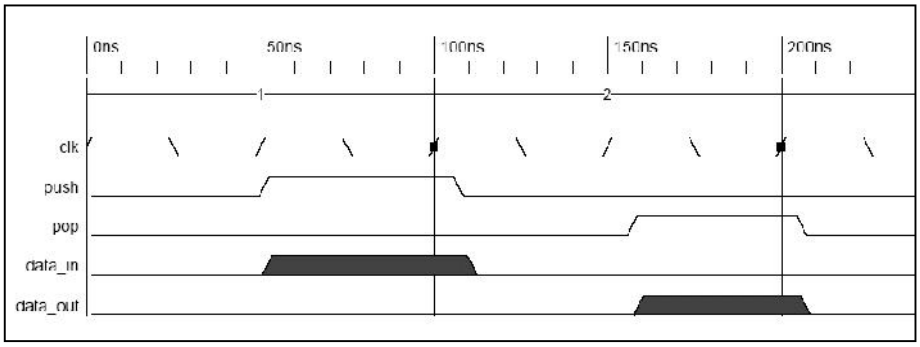

Figure 5.1.1 FIFO Interface Timing Diagram

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
  cq_push_error : cover sequence  (@(posedge clk) q_push_error);
  ap_push_no_pop: assert property(@ (posedge clk) push |-> !full || pop);
```

Could also use the **let** construct:
```
let q_push_error =  !(push && full && !pop);
```

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

// never a pop on empty
sequence q_pop_error;
    ! (pop && empty && !push);
endsequence : q_pop_error
cq_pop_error : cover sequence  (@ (posedge clk) q_pop_error);
ap_pop_nopush: assert property(@ (posedge clk) pop |-> !empty || push);
```

Could also use the **let** construct:
let q_pop_error = ! (pop && empty && !push);

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;
   dataQ_size == BIT_DEPTH;
endsequence : qFull

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

Could also use the **let** construct:
let qFull = dataQ_size == BIT_DEPTH;

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;
   dataQ_size >= ALMOST_FULL;
endsequence : qAlmost_full
```

Could also use the **let** construct:
let qAlmost_full = dataQ_size >= ALMOST_FULL;

```
property p_fifo_almost_full;
    qAlmost_full |-> almost_full;
endproperty : p_fifo_almost_full
ap_fifo_almost_full : assert property (@ (posedge clk)  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;
    dataQ_size==0;
endsequence : qEmpty
```

Could also use the **let** construct:
let qEmpty = dataQ_size==0;

```
property p_fifo_empty;
    qEmpty |-> empty;
endproperty : p_fifo_empty
ap_fifo_empty : assert property (@ (posedge clk) 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;
    dataQ_size <= ALMOST_EMPTY;
endsequence : qAlmost_empty
```

Could also use the **let** construct:
let qAlmost_empty = dataQ_size <= ALMOST_EMPTY;

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

5.15 Clock

Direction: Input, Peripheral -> FIFO ; Size: 1 bit, Active edge: rising edge

The *clk* clock is the synchronous system clock for both the read and write transactions; it is 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;
   q_push_error or q_pop_error |=> error;
endproperty : p_error_flag
ap_error_flag : assert property (   @ (posedge clk) 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 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 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.

FIFO Requirements Example

9. PERFORMANCE

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

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.

6.9.2 Test Plan

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