VC Formal Lab
Symbolic Variable

Learning Objectives

In this VC Formal lab, you will use a FIFO example to learn to do the following:

- Use symbolic variable
- Find a solution to converge the properties

Lab Duration:
40 minutes

Familiarity with the SystemVerilog Assertion (SVA) language and knowledge of basic formal verification concepts are required for this lab.

Files Location

All files for this VC Formal lab are in directory:
$VC_STATIC_HOME/doc/vcst/examples/FPV/Abstraction/Symbolic_Variable

Directory Structure	
FPV/Abstraction/Symbolic_Variable	Lab main directory
README_VCFormal_Symbolic_Variable.pdf	Lab instructions
design/	Verilog RTL code of the Device Under Test (DUT)
sva/	SVA properties to check functionality of the DUT
run/	Run directory
solution/	Solution directory

Resources

The following resources are available for in-depth guidance regarding VC Formal usage, commands, and variables.

VC Formal User Guide:
$VC_STATIC_HOME/doc/vcst/VC_Formal_Docs/VC_Formal_UG.pdf

VC Formal Apps Quick References Guides:
$VC_STATIC_HOME/doc/vcst/VC_Formal_Docs/Quick_Reference_Guides/

VC Formal Apps Tcl Templates:
$VC_STATIC_HOME/doc/vcst/VC_Formal_Docs/Quick_Reference_Guides/vcf_tcl_templates/

Prepare your Environment

1. Set environment variable pointing to your VC Formal installation directory:

```
    %setenv VC_STATIC_HOME /tools/synopsys/vcstatic
```

2. Add path $VC_STATIC_HOME/bin to the PATH environment variable.

3. Change your working directory to FPV/Abstraction/Symbolic_Variable/run:

```
    %cd FPV/Abstraction/Symbolic_Variable/run
```

Now you are ready to begin the lab.

Create a run.tcl Setup File

VC Formal has a Tcl-based command interface. It is common to start with a Tcl file to set up and compile a design. In this step, you will create a VC Formal Tcl file for the DUT, a FIFO, used in this lab.

4. Open file run.tcl (any arbitrary name is ok to use) using any text editor:

```
    %vi run.tcl
```

5. Add command to enable FPV App mode (default when starting VC Formal):

```
    set_fml_appmode FPV
```

6. Specify Formal TB top level module name as Tcl variable:

```
    set design fifo
```

7. Add command to compile DUT and SVA properties:

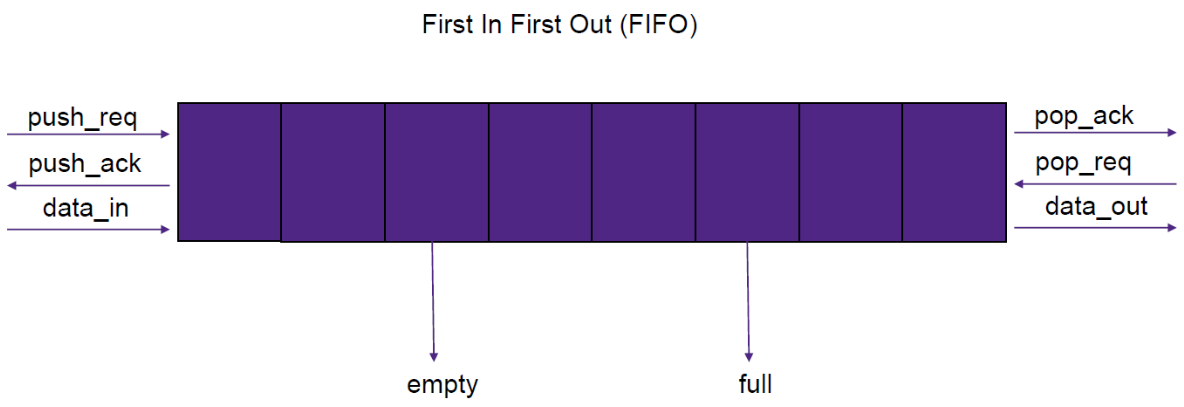
   The DUT files and filelist are located under directory FPV/Abstraction/Symbolic_Variable/design. The assertion and bind files are located under directory FPV/Abstraction/Symbolic_Variable/sva.

```
    read_file -top $design -format sverilog -sva \
         -vcs {-f ./filelist.flist}
```

8. Save run.tcl file and exit editor.

FIFO Design Implementation

- Design writes and read data in First In First Out Order.
- “full” flag raised when design has no free space.
- “empty” flag raised when the design has no data inside.
- Data written when “push_req” is asserted and “full” flag is low. “push_ack” gets asserted,
- Data read when “pop_req” is asserted and “empty” flag is low. “pop_ack” gets asserted.



FIFO Test Plan

Create assertions to test:

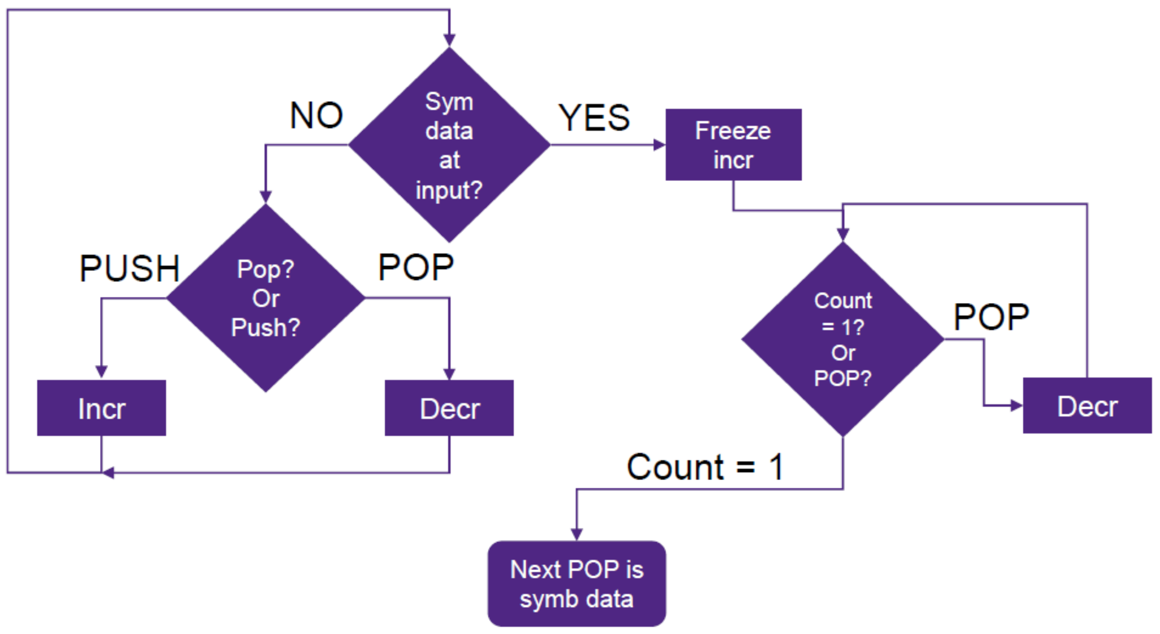
- Design should write data and read data in a First In First Out order
- When the design has no more free space the full flag should be raised
- When the design has no data inside the empty flag should be raised
- When the full flag is high, push_ack must be low
- When the empty flag is high, pop_ack must be low
- When push_req is high and push_ack is low, push_req must be kept high and data_in stable
- When pop_req is high and pop_ack low, pop_req must be kept high

Add design constraints:
- Push request and data_inheld till push ack received
- Pop request held till pop ack received

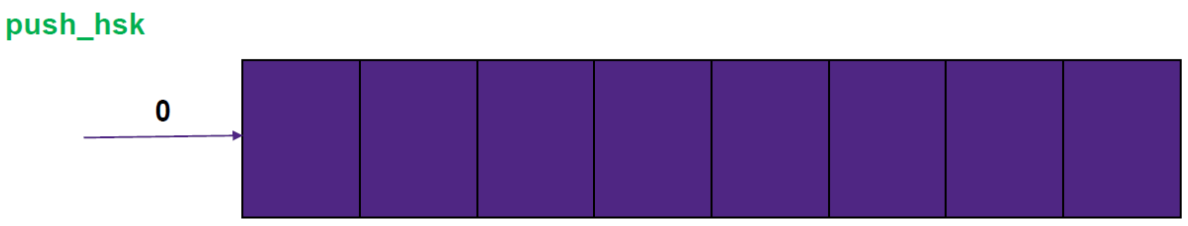
Lab Solution with Symbolic Variable

- In formal testbench we are going to take advantage of this to track a single item, representing all possible items, of data through the design.
- Declare a symbolic item of watched data that we will track through the design.
- Declare registers which will flag the events that we are interested in:
    - Symb data entering DUT
    - Symb data exiting DUT
- Use a counter to track the symbolic data through the design:
    - If symb data in flag is not set:
        - Push = increment
        - Pop = decrement
- When the symb data inside flag has been set freeze the increment condition. The counter now holds the number of elements ahead of the symb data. Continue to decrement the counter on every pop, and when it has been reduced to 1, on the next pop the symbolic data should appear on the output port.

Symbolic Data Tracker Flowchart

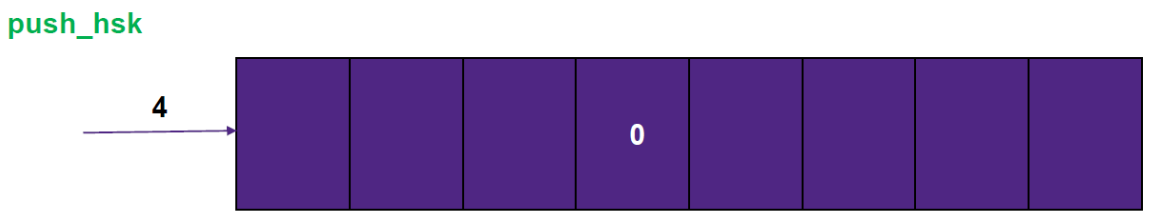


Symbolic Data Tracker Example



Scoreboard:

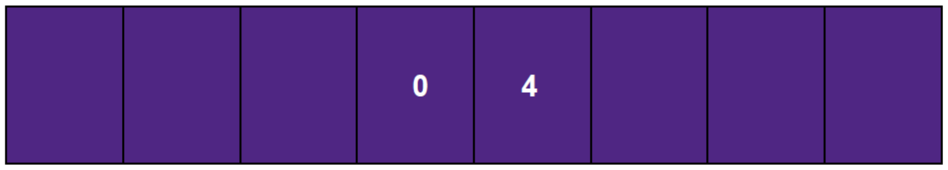
symb_data_in = 0
symb_data_out = 0
data_ahead_counter = 0



Scoreboard:

symb_data_in = 0
symb_data_out = 0
data_ahead_counter = 1

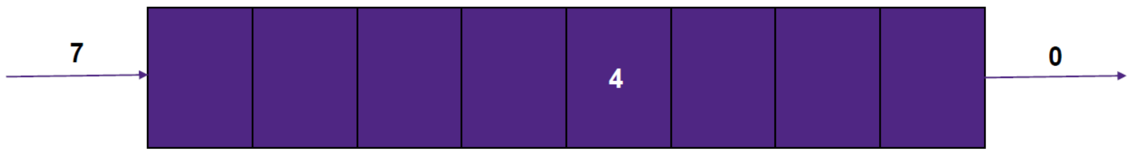
pop_hsk



Scoreboard:

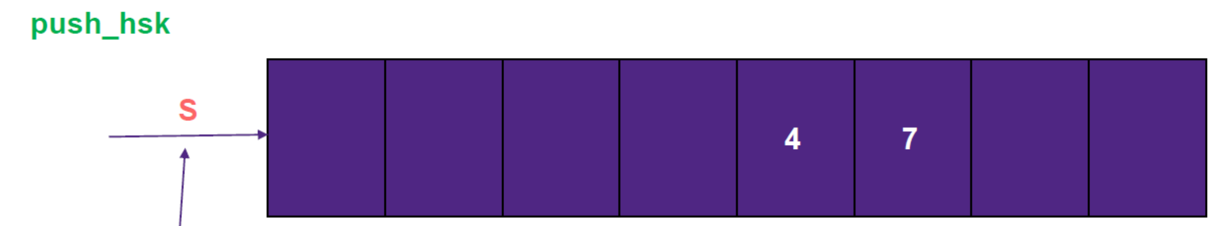
symb_data_in = 0
symb_data_out = 0
data_ahead_counter = 2

push_hsk



Scoreboard:

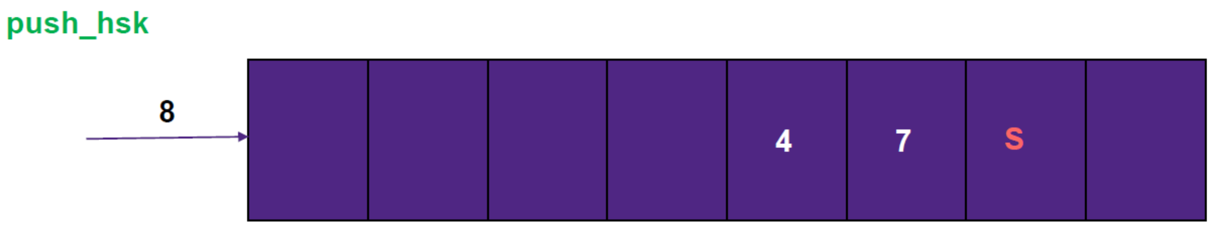
symb_data_in = 0
symb_data_out = 0
data_ahead_counter = 1



Symbolic data has appeared on the input port. In the next clock cycle the symb_data_in flag should be set and increment frozen

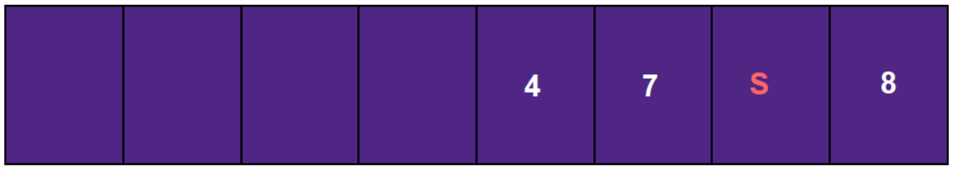
Scoreboard:

symb_data_in = 0
symb_data_out = 0
data_ahead_counter = 2



Scoreboard:

symb_data_in = 1
symb_data_out = 0
data_ahead_counter = 3

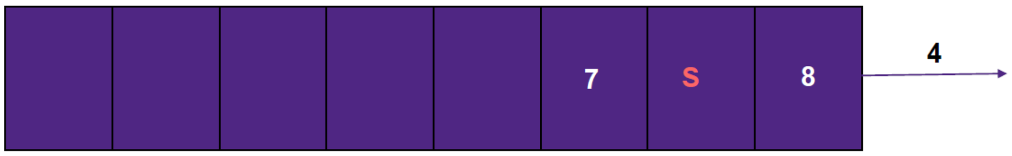


Scoreboard:

symb_data_in = 1
symb_data_out = 0
data_ahead_counter = 3

Another push has happened, but because of symb_data_in being set, the increment has been frozen

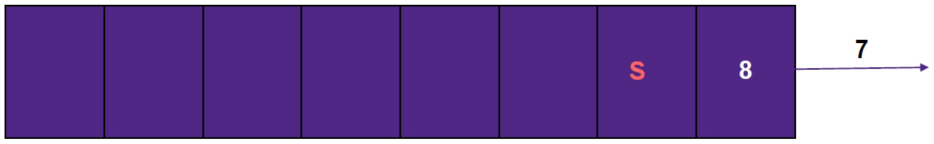
pop_hsk



Scoreboard:

symb_data_in = 1
symb_data_out = 0
data_ahead_counter = 2

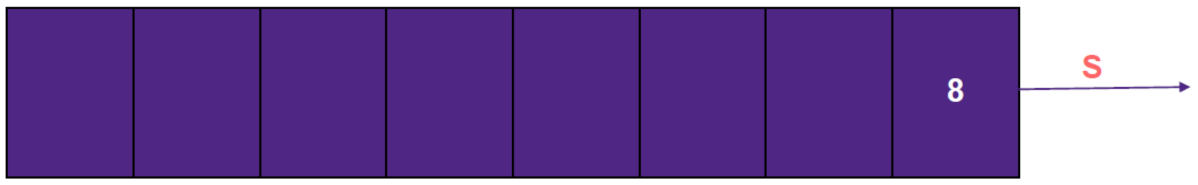
pop_hsk



Scoreboard:

symb_data_in = 1
symb_data_out = 0
data_ahead_counter = 1

In this clock cycle we have seen the symb data in and the counter has reduced to 1. On the next pop the data on the output port should match the symb_data



Scoreboard:

symb_data_in = 1
symb_data_out = 0
data_ahead_counter = 0

- Code constraints on push_req and pop_req to meet the requirements:

```
am_push_req_and_data_in_stable_when_no_ack: assume property
      (`clk_rst  push_req && !push_ack |=> $stable({push_req,
           data_in}));

am_pop_req_stable_when_no_ack: assume property
      (`clk_rst pop_req&& !pop_ack |=> pop_req);
```

- Code assertions to check design behavior:

```
as_full_flag_correct: assert property
      (`clk_rst (fullness_counter== DEPTH) |-> full);

as_empty_flag_correct: assert property
      (`clk_rst (fullness_counter== 0) |-> empty);

as_no_push_ack_on_full: assert property
      (`clk_rst full |-> !push_ack);

as_no_pop_ack_on_empty: assert property
      (`clk_rst empty |-> !pop_ack);
```

- Code constraint to make the symbolic data stable:

```
am_data_symb_stable: assume property
      (`clk_rst $stable(symb_data));
```

- Code data integrity check:

```
as_data_integrity: assert property
      (`clk_rst symb_data_in && !symb_data_out &&
           (data_ahead_counter== 1)
           && pop_hsk |=> (data_out== symb_data));
```