# CME 435: Verification of Digital Systems

# Xswitch Verification Project Report

# December 16, 2019

# verification of xswitch

The design under test (DUT) is a type of packet switch called an Xswitch. It consists of four input ports and four output ports.

## Interfaces

### downstream

* data\_in: (8-bit input) Data received by design
* addr\_in: (8-bit input) Address of port to route data to
* valid\_in: (1-bit input) Indicates input is valid
* rcv\_rdy: (1-bit output) Indicates that device is ready to accept data

### upstream

* data\_out: (8-bit output) Data was routed to this port
* addr\_out: (8-bit output) Input port where this data came from
* valid\_out: (1-bit output) Indicates output data is valid
* data\_rd: (1-bit input) Indicates that the current data has been read

## protocols

### downstream

* Inputs must be driven from testbench on the positive edge of the clock
* The DUT will sample inputs and drive outputs on the negative edge of the clock
* When data is driven the valid\_in bit of the corresponding port must be set
* If rcv\_rdy bit of corresponding port is not set then data may not be driven

### upstream

* Input data is driven to the output port specified by addr\_in
* addr\_out is updated to reflect source port of data
* valid\_out indicates whether data is available on the corresponding output port
* data\_rd must be set to indicate that new data can be driven to the corresponding output port

## design features

The DUT supports the following features that need to be verified:

1. Input data can be driven to any output port

* input data is a single byte driven to data\_in for the corresponding port
* addr\_in is used to indicate the desired destination port
* addr\_out reflects the source port
* data\_out is the output containing the driven byte of data

1. Input data must only be driven when valid\_in is set

* if valid\_in is not set then data should not be driven to the specified output port

1. Input data must only be driven when rcv\_rdy is set

* if rcv\_rdy is not set then the data previously driven into the system has not been received on the output

1. When output data is received data\_rd must be set

* data\_rd is used to indicate that data has been received by the recipient

1. valid\_out must be set by the DUT when there is valid data on data\_out for a corresponding port

## test cases

In order to verify all of the identified design features in the previous section the following tests are required:

### 1.1 test sanity check

This is the base test case that exercises all ports by driving data to unique destination ports. In order to properly verify the system a task called sanity\_check() is implemented in the scoreboard to ensure that all input data is routed to the appropriate output ports. The data driven must also match the corresponding input data.

The data driven into the system is randomized by generating random data\_in and addr\_in values for each of the input ports. A key constraint on the randomly generated addr\_in value is that it must be a unique value between 0 and 3 for each of the four input ports on each clock cycle.

The expected output is for each input byte to be driven to the specified output port, while also adhering to the protocols defined in the protocols section of the report.

### 1.2 test reset

This test case involves running 1.1 Test Sanity Check twice. The reason this test is so simple to implement is because each run of the environment starts with a reset. In this test case the full sanity check is run and then the system is reset. This allows the reset functionality to be monitored after the system has been thoroughly exercised.

The expected results for this test are two clean sanity check runs of the system. If the reset works as intended, then the system will properly clear itself and start switching data in the second run just as it did in the first run.

### 1.3 test valid in

This test case involves randomizing valid\_in for each of the transactions described in 1.1 Test Sanity Check. This allows the testbench to attempt driving data to the system while valid\_in is low. In this case one would expect the data to be refused and the output to not be driven.

The expected result is to have the data\_in bytes that are driven with valid\_in set high to properly be outputted by the DUT. If the data\_in byte is driven with valid\_in low then the output should not be updated and valid\_out should be low.

## Bugs

### 2.1 rcv\_rdy low when data\_rd set high

This bug was identified during 1.1 Test Sanity Check. An assertion was set to enforce rcv\_rdy being set when data\_out is received by the monitor. In this case the monitor set data\_rd but rcv\_rdy remained low. This caused the driver to be unable to drive a packet to the DUT.

# \*\* Error: Assertion error.

# Time: 2830 ns Started: 2830 ns Scope: tbench\_top.dut.assert\_dut.genblk5[1].genblk1[1].a\_data\_rd\_rcv\_rdy File: assertions.sv Line: 107

# \*\* Error: Assertion error.

# Time: 2830 ns Started: 2810 ns Scope: tbench\_top.dut.assert\_dut.genblk6[1].genblk1[1].a\_rcv\_rdy\_eq\_data\_rd1 File: assertions.sv Line: 132

### 2.2 reset is bad news

During 1.2 Test Reset assertions were being triggered after the second reset. This indicates that driving data into the system after a reset is not possible. The valid\_in bit is not set, although the test case always attempts to set valid\_in before driving data.

# \*\* Error: Assertion error.

# Time: 265890 ns Started: 265890 ns Scope: tbench\_top.dut.assert\_dut.genblk2[0].genblk1[1].a\_valid0 File: assertions.sv Line: 50

### 2.3 driving without valid\_in sets valid\_out high

This bug was identified when running 1.3 Test Valid In. In this case data was attempted to be driven while valid\_in was not set. It turns out the valid\_out would be set by the DUT in this case, violating the protocol defined above. This is the same assertion as the one in the bug above, however in this case one would expect to see valid\_in not set during some transactions. In these cases valid\_out would also be expected to be set low.

# \*\* Error: Assertion error.

# Time: 2890 ns Started: 2890 ns Scope: tbench\_top.dut.assert\_dut.genblk2[1].genblk1[1].a\_valid0 File: assertions.sv Line: 50

## Functional coverage

To be brief, all of the functional coverage bins identified in the design were hit when running all three of the testcases and merging the individual reports. The reason for this is that randomizing all inputs to drive unique outputs in a system with only four ports is a small task. In addition the inputs were all driven to unique outputs with minimum data bytes and maximum data bytes, as well as a value in between.

The coverage holes that exist are due to toggles not occurring for the rcv\_rdy signal. The testbench always receives the output data on every clock cycle, so rcv\_rdy never goes low. I would like to fix this coverage hole, if given more time. Otherwise, all functional and assertion coverage is met in the regression test.

## Testbench performance

The testbench was designed with maximum reusability in mind. In this case all bugs were able to be identified by running three unique testcases. No changes had to be made to the testbench architecture to run these testcases. One of the key features here is a base sanity check performed by the scoreboard. In addition, the architecture supports class parametrization so that the base transaction class can be extended to override constraints on the input data. This allows the randomized data to be targeted more specifically to individual testcase goals. In addition to the sanity check, the testbench also supports assertions from a file called assertions.sv. This module asserts many different properties on all of the testcases as they are run. This helps identify bugs 2.2 and 2.3.

The testbench also handles particular error injections by having a timeout task in environment.sv that will end the test if it is blocked from finishing due to a particular bug. In addition to the timeout, the driver will also not send the transaction to the scoreboard if addr\_in is invalid or valid\_in is not set. This check prevents the scoreboard from comparing erroneous inputs to valid outputs.

The testbench attempts to verify the same features in each of the testcases, however each of the testcases exposes a unique bug, which prevents all features from being verified in each test case.

NOTE: As the files are currently implemented, the Phase 9 testcases have bugs enabled so that each test can individually expose a bug. The regression test has bugs disabled so that the output of the test is not too long. These bugs can easily be enabled by uncommenting the line in each of the testcase files.