ISSN 2319 – 1953
International Journal of Scientific Research in Computer Science Applications and Management Studies

Design and Simulation of NOC Router Using VERILOG

Talluri Deepak[1], T. Pattalu Naidu[2]
[1]Department of ECE, [2]Assistant Professor,Department of ECE, VLSI,
[1,2]Avanthi Institute of Engineering and Technology, Visakapatnam dist, A.P, India.
deepak.talluri@yahoo.com[1], naidu.tamarana1@gmail.com[2]

Abstract: Router is a packet based protocol. Router drives the incoming packet which comes from the input port to output ports based on the address contained in the packet. The router is a "Network Router" has a one input port from which the packet enters. It has three output ports where the packet is driven out. Packet contains 3 parts. They are Header, data and frame check sequence. Packet width is 8 bits and the length of the packet can be between 1 bytes to 63 bytes. Packet header contains three fields DA and length. Destination address (DA) of the packet is of 8 bits. The switch drives the packet to respective ports based on this destination address of the packets. Each output port has 8-bit unique port address. If the destination address of the packet matches the port address, then switch drives the packet to the output port, Length of the data is of 8 bits and from 0 to 63. Length is measured in terms of bytes. Data should be in terms of bytes and can take anything. Frame check sequence contains the security check of the packet. It is calculated over the header and data. The Verification Plan is based on System Verilog Hardware Verification Language. The methodology used for Verification is Constraint random coverage driven verification.

I. INTRODUCTION

Based upon mechanism of communication digital circuit use for data exchange between different modules or circuits, they have been classified into two different categories [2] which are as;

- Synchronous circuits
- Asynchronous circuits

Synchronous Circuits:

In synchronous circuits, all the circuit elements are synchronized on a single clock frequency. Changes in circuit signals are based on the change in logic level of clock signal. How fast a circuit is? It can be estimated through the frequency of clock signal over which circuit is designed to operate. Various parameters need to be taken care of while designing the circuit. A circuit cannot operate at any frequency. In ideal case, output should come at same instant when an input is applied. However there are some components involved from input to output, these components required some time for their operation, so it takes some time to change the output when an input signal is changed. Normally, an analysis is performed from input to output paths, the longest path in term of time consumption is termed as critical path. This path is further optimized to improve circuit timing.

As mentioned earlier, synchronous circuit performs tasks according to the clock signal. So each circuit element is linked to global cock signal. To guarantee the correct operation, clock signal should reach at same time to each element. H tree [3] topology is usually used as for uniform clock distribution without any delay. However, sometime clock signal gets delay and this delay is termed as clock skew. Ideally clock skew should be zero.

The challenge of the verifying a large design is growing exponentially. There is a need to define new methods that makes functional verification easy. Several strategies in the recent years have been proposed to achieve good functional verification with less effort. Recent advancement towards this goal is methodologies. The methodology defines a skeleton over which one can add flesh and skin to their requirements to achieve functional verification. OVM (open verification methodology) is one such efficient methodology and best thing about it is, it is free. This ovm is built on system Verilog and used effectively to achieve maintainability, reusability, speed of verification etc. This project is aimed at building a reusable test bench for verifying usb-i2c Protocol Bridge by using system Verilog and ovm

In this document the use of vmm and system Verilog to verify a design and to develop a reusable test bench is explained in step by step as defined by verification principles and methodology. The test bench contains different components and each component is again composed of subcomponents, these components and subcomponents can be reused for the future projects as long as the interface is same.

The report is organized as two major portions; first part is brief introduction and history of the functional verification which tells about different technologies, strategies and methodologies used today for verification. Literature survey will contain an organized collection of data from different sources and significant changes that took place in the verification and design. Ovm methodology basics illustrate some of the methodology concepts necessary for understanding of the project which assumes a prior knowledge of the system Verilog language.

Second part is verification plan specifying the verification requirements and approaches to attack the problem, architecture of the test bench gives complete description about the components and sub components used to achieve the verification goals and also explains about improvements made in the design of the usb-i2c bridge, test plan identifies all the test case required to meet the goals and finally results of the project

Asynchronous Circuits:

Single clock distribution to whole complex of circuit can be very challenging. So, instead of using clock signal to synchronize tasks, circuits can also be operated based on handshake signals. Circuits that perform handshaking before data exchange are known as asynchronous circuits. There are two handshake mechanisms on which circuits can be designed [4].

A. 4-Phase Handshake:

In 4-phase handshake sender puts the request (Req) signal high and in response receiver puts ACK (acknowledgement) signal high. Sender sends the data, receiver after getting this data put ACK signal low. In last sender also puts request

signal low and one data transaction is 18 completed. This operation can be illustrated with the help of Fig. 1. This handshaking is also known as return to zero (RTZ).

B. 2-Phase Handshake:

In 2-phase handshake mechanism instead of wasting time and energy in returning to logic level zero, events are defined on request and acknowledgement signals. Although this type of handshaking is efficient as compared to other type but it is complex in implementation.

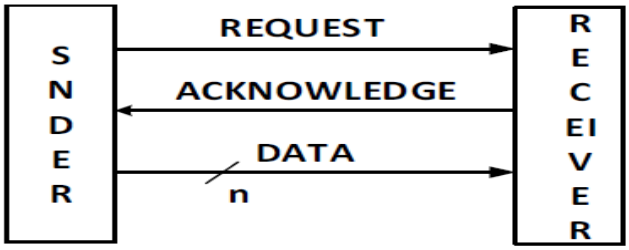

Figure 1: Asynchronous handshaking mechanisms for data transfer

II. LITERATURE SURVEY

Verification:

Verification is not a testbench, nor is it a series of testbenches. Verification is a process used to demonstrate that the intent of a design is preserved in its implementation. In this chapter, I introduce the basic concepts of verification, from its importance and cost, to making sure you are verifying that you are implementing what you want. The differences are presented between various verification approaches as well as the difference between testing and verification. It is also showed that how verification is key to design reuse, and challenges of verification reuse.

Importance of verification

Today, in the era of multi-million gate ASICs and FPGAs, reusable intellectual property (IP), and system-on-a-chip (SoC) designs, verification consumes about 70% of the design effort. Design teams, properly staffed to address the verification challenge, include engineers dedicated to verification. The number of verification engineers can be up to twice the number of RTL designers.

Given the amount of effort demanded by verification, the shortage of qualified hardware design and verification engineers, and the quantity of code that must be produced, it is no surprise that, in all projects, verification rests squarely in the critical path. The fact that verification is often considered after the design has been completed, when the schedule has already been ruined, compounds the problem. It is also the reason verification is the target of the most recent tools and methodologies. These tools and methodologies attempt to reduce the overall verification time by enabling parallelism of effort, higher abstraction levels and automation.

If efforts can be parallelized, each other as well as in parallel with the implementation of the design. Additional resources can be applied effectively to reduce the total verification time. For example, digging a hole in the ground can be parallelized by providing more workers armed with shovels. To parallelize the verification effort, it is necessary to be able to write—and debug—testbenches in parallel with

Formal verification, property checking,functional verification, and rule checkers verify different things because they have different origin and reconvergence points.

Equevalence checking

In its most common use, equivalence checking compares two netlists to ensure that some netlist post-processing, such as scanchain insertion, clock-tree synthesis or manual modification1, did not change the functionality of the circuit.

Another popular use of equivalence checking is to verify that the netlist correctly implements the original RTL code. If one trusted the synthesis tool completely, this verification would not be necessary. However, synthesis tools are large software systems that depend on the correctness of algorithms and library information. History has shown that such systems are prone to error. Equivalence checking is used to keep the synthesis tool honest. In some rare instances, this form of equivalence checking is used to verify that manually written RTL code faithfully represents a legacy gatelevel design.

Property checking

Property checking is a more recent application of formal verification technology. In it, assertions or characteristics of a design are formally proven or disproved. For example, all state machines in a design could be checked for unreachable or isolated states. A more powerful property checker may be able to determine if deadlock conditions can occur.

Functional verification

The main purpose of functional verification is to ensure that a design implements intended functionality. As shown by the reconvergence model in Figure 1-8, functional verification reconciles a design with its specification. Without functional verification, one must trust that the transformation of a specification document into RTL code was performed correctly, without misinterpretation of the specification's intent.It is important to note that, unless a specification is written in a formal language with precise semantics,1 it is impossible to prove that a design meets the intent of its specification. Specification documents are written using natural languages by individuals with varying degrees of ability in communicating their intentions. Any document is open to interpretation. One can easily prove that the design does not implement the intended function by identifying a single discrepancy. The converse, sadly, is not true: No one can prove that there are no discrepancies. Functional verification, as a process, can show that a design meets the intent of its specification. But it cannot prove it.

Functional verification approaches

Functional verification can be accomplished using three complementary approaches: black-box, white-box and grey-box.

Black box verification

With a black-box approach, functional verification is performed without any knowledge of the actual implementation of a design. All verification is accomplished through the available interfaces, without direct access to the internal state of the design, without knowledge of its structure and implementation. This method suffers from an obvious lack of visibility and controllability. It is often difficult to set up an interesting state combination or to isolate some functionality. It is equally difficult to observe the response from the input and locate the source of the problem. This difficulty arises from the frequent long delays between the occurrence of a problem and the appearance of its symptom on the design's outputs.

The advantage of black-box verification is that it does not

depend on any specific implementation. Whether the design is implemented in a single ASIC, RTL code, transaction-level model, gates, multiple FPGAs, a circuit board or entirely in software, is irrelevant. A black-box functional verification approach forms a true conformance verification that can be used to show that a particular design implements the intent of a specification, regardless of its implementation. A set of black-box testbenches can be developed on a transaction-level model of the design and run, unmodified, on the RTL model of the design to demonstrate that they are equivalent. Black-box testbenches can be used as a set of golden testbenches.

White box verification

As the name suggests, a white-box approach has full visibility and controllability of the internal structure and implementation of the design being verified. This method has the advantage of being able to set up an interesting combination of states and inputs quickly, or to isolate a particular function. It can then easily observe the results as the verification progresses and immediately report any discrepancies from the expected behavior.

However, this approach is tightly integrated with a particular implementation. Changes in the design may require changes in the testbench. Furthermore, those testbenches cannot be used in gate-level simulations, on alternative implementations or future redesigns. It also requires detailed knowledge of the design implementation to know which significant conditions to create and which results to observe.

Gray box verification

Grey-box verification is a compromise between the aloofness of black-box verification and the dependence on the implementation of white-box verification. The former may not fully exercise all parts of a design, while the latter is not portable.

As in black-box verification, a grey-box approach controls and observes a design entirely through its top-level interfaces. However,the particular verification being accomplished is intended to exercise significant features specific to the implementation. The same verification of a different implementation would be successful, but the verification may not be particularly more interesting than any other black-box verification. A typical grey-box test case is one written to increase coverage metrics. The input stimulus is designed to execute specific lines of code or create a specific set of conditions in the design. Should the structure (but not the function)of the design change, this test case, while still correct, may no longer contribute toward better coverage.

Testing versus verification

Testing is often confused with verification. The purpose of the former is to verify that the design was manufactured correctly. The purpose of the latter is to ensure that a design meets its functional intent.During testing; the finished silicon is reconciled with the netlist that was submitted for manufacturing.

Testing is accomplished through test vectors. The objective of these test vectors is not to exercise functions. It is to exercise physical locations in the design to ensure that they can go from 0 to 1 and from 1 to 0 and that the change can be observed. The ratio of physical locations tested to the total number of such locations is called test coverage. The test vectors are usually automatically generated to maximize coverage while minimizing vectors through a process called automatic test pattern generation (ATPG).

Verification technologies

Presents various verification technologies separately from each other. Each technology is used in multiple EDA tools. A specific EDA tool may include more than one technology. For example, "super linting" tools leverage linting and formal technologies. Hybrid- or semi-formal tools use a combination of simulation and formal technologies.

Linting

The term "lint" comes from the name of a UNIX utility that parses a C program and reports questionable uses and potential problems. When the C programming language was created by Dennis Ritchie, it did not include many of the safeguards that have evolved in later versions of the language, like ANSI-C or C++, or other stronglytyped languages such as Pascal or ADA. lint evolved as a tool to identify common mistakes programmers made, letting them find the mistakes quickly and efficiently, instead of waiting to find them through a dreaded segmentation fault during execution of the program.

Limitation of linting technology is that it is often too paranoid in reporting problems it identifies. To avoid letting an error go unreported, linting errs on the side of caution and reports potential problems where none exist. This results into a lot of false errors. Designers can become frustrated while looking for non-existent problems and may abandon using linting altogether.

Code Reviews

Although not technically linting, the objective of code reviews is essentially the same: Identify functional and coding style errors before functional verification and simulation. Linting can only identify questionable language uses. It cannot check if the intended behaviour has been coded. In code reviews, the source code produced by a designer is reviewed by one or more peers. The goal is not to publicly ridicule the author, but to identify problems with the original code that could not be found by an automated tool. Reviews can identify discrepancies between the design intent and the implementation. They also provide an opportunity for suggesting coding improvements, such as better comments, better structure or the use of assertions.

A code review is an excellent venue for evaluating the maintainability of a source file, and the relevance of its comments and assertions. Other qualitative coding style issues can also be identified. If the code is well understood, it is often possible to identify functional errors or omissions.

Simulation

Simulation is the most common and familiar verification technology. It is called "simulation" because it is limited to approximating reality. A simulation is never the final goal of a project. The goal of all hardware design projects is to create real physical designs that can be sold and generate profits. Simulation attempts to create an artificial universe that mimics the future real design. This type of verification technology lets the designers interact with the design before it is manufactured and correct flaws and problems earlier.

Within that simplified universe, the only thing a simulator does is execute a description of the design. The description is limited to a well-defined language with precise semantics. If that description does not accurately reflect the reality it is trying to model, there is no way for you to know that you are

simulating something that is different from the design that will be ultimately manufactured. Functional correctness and accuracy of models is a big problem as errors cannot be proven not to exist.

Code coverage

Code coverage is a technology that can identify what code has been (and more importantly not been) executed in the design under verification. It is a technology that has been in use in software engineering for quite some time. The problem with a design containing an unknown bug is that it looks just like a perfectly good design. It is impossible to know, with one hundred percent certainty, that the design being verified is indeed functionally correct. All of your testbenches simulate successfully, but are there sections of the RTL code that you did not exercise and therefore not triggered a functional error? That is the question that code coverage can help answer.

Only the code for the design under verification needs to be covered and thus instrumented. The objective of code coverage is to determine if you have forgotten to exercise some code in the design. The code for the test benches need not be traced to confirm that it has executed. If a significant section of a test bench was not executed, it should be reflected in some portion of the design not being exercised. Furthermore, a significant portion of test bench code is executed only if errors are detected. Code coverage metrics on test bench code are therefore of little interest.

Functional coverage

Functional coverage is another technology to help ensure that a bad design is not hiding behind passing test benches. Although this technology has been in use at some companies for quite some time, it is a recent addition to the arsenal of general-purpose verification tools. Functional coverage records relevant metrics (e.g., packet length, instruction op-code, buffer occupancy level) to ensure that the verification process has exercised the design through all of the interesting values. Whereas code coverage measures how much of the implementation has been exercised, functional coverage measures how much of the original design specification has been exercised.

High functional coverage does not necessarily correlate with high code coverage. Whereas code coverage is concerned with recording the mechanics of code execution, functional coverage is concerned with the intent or purpose of the implemented function. For example, the decoding of a CPU instruction may involve separate case statements for each field in the opcode. Each case statement may be 100 percent code-covered due to combinations of field values from previously decoded opcodes. However, the particular combination involved in decoding a specific CPU instruction may not have been exercised.

Like code coverage, functional coverage metrics are collected at runtime, during a simulation run. The values from individual runs are collected into a database or separate files. The functional coverage metrics from these separate runs are then merged for offline analysis. The marginal coverage of individual runs can then be graded to identify which runs contributed the most toward the overall functional coverage goal. These runs are then given preference in the regression suite, while pruning runs that did not significantly contribute to the objective.

SystemVerilog provides a set of predefined methods that let a testbench dynamically query a particular functional coverage metric. The testbench can then use the information to modify its current behavior. For example, it could increase the probability of generating values that have not been covered yet. It could decide to abort the simulation should the functional coverage not have significantly increased since the last query.

III. ROUTER DESIGN SPECIFICATION

Router is a packet based protocol. Router drives the incoming packet which comes from the input port to output ports based on the address contained in the packet

The router has a one input port from which the packet enters. It has three output ports where the packet is driven out. The router has an active low synchronous input resetn which resets the router.

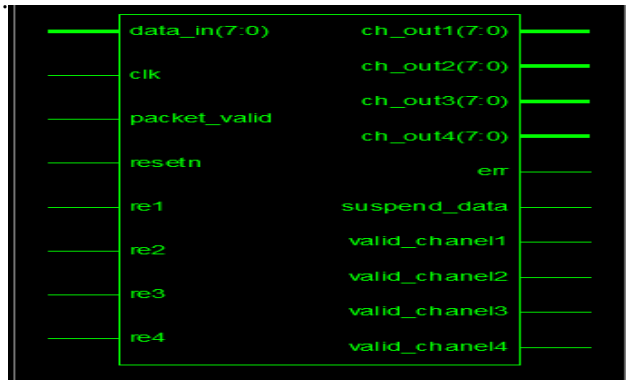

Figure 2- Block Diagram Of Five Port Router

Data packet moves in to the input channel of one port of router by which it is forwarded to the output channel of other port. Each input channel and output channel has its own decoding logic which increases the performance of the router. Buffers are present at all ports to store the data temporarily.

The buffering method used here is store and forward. Control logic is present to make arbitration decisions. Thus communication is established between input and output ports.. According to the destination path of data packet, control bit lines of FSM are set. The movement of data from source to destination is called switching mechanism The packet switching mechanism is used here, in which the flit size is 8 bits .Thus the packet size varies from 0 bits to 8 bits. A detailed explanation of Design is as follow

Packet Format

Packet contains 3 parts. They are Header, payload and parity.

- Packet width is 8 bits and the length of the packet can be between 1 bytes to 63 bytes.

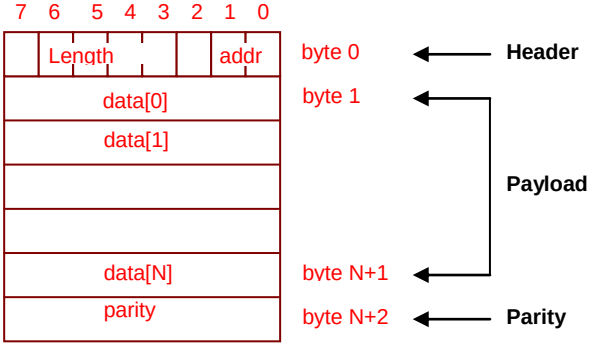

Figure 3- Data Packet Format

Packet Header

Packet header contains two fields DA and length.

DA:

- Destination address of the packet is of 2 bits. The router drives the packet to respective ports based on this destination address of the packets.
- Each output port has 2-bit unique port address. If the destination address of the packet matches the port address, then router drives the packet to the output port. The address"3"is in valid.
- Length: Length of the data is of 6 bits and from 1 to 63. It specifies the number of data bytes.
- A packet can have a minimum data size of 1 byte and a maximum size of 63 bytes.
  - If Length = 1, it means data length is 1 bytes
  - If Length = 2, it means data length is 2 bytes
  - If Length = 63, it means data length is 63 bytes

Packet – Payload

Data: Data should be in terms of bytes and can take anything.

Parity: This field contains the security check of the packet. It should be a byte of even, bitwise parity, calculated over the header and data bytes of the packet.

Router Input Protocol

The characteristics of the DUV input protocol are as follows:

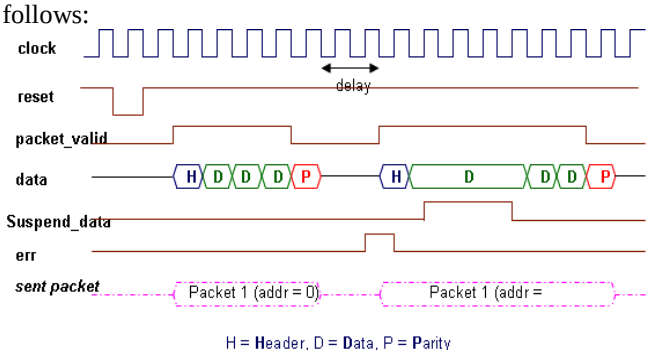

Figure 4- Router Input Protocol

- All input signals are active high and are synchronized to the falling edge of the clock. This is because the DUV router is sensitive to the rising edge of clock. Therefore, driving input signals on the falling edge ensures adequate setup and hold time, but the signals can also be driven on the rising edge of the clock.
- The packet_valid signal has to be asserted on the same clock as when the first byte of a packet (the header byte), is driven onto the data bus.
- Since the header byte contains the address, this tells the router to which output channel the packet should be routed (data_out_0, data_out_1, or data_out_2).
- Each subsequent byte of data should be driven on the data bus with each new rising/falling clock.
- After the last payload byte has been driven, on the next rising/falling clock, the packet_valid signal must be deasserted , and the packet parity byte should be driven. This signals packet completion.
- The input data bus value cannot change while the suspend_data signal is active (indicating a FIFO overflow). The packet driver should not send any more bytes and should hold the value on the data bus. The width of suspend_data signal assertion should not exceed 100 cycles.
- The err signal asserts when a packet with bad parity is detected in the router, within 1 to 10 cycles of packet completion

Router Output Protocol

The characteristics of the output protocol are as follows:

- All output signals are active high and can be synchronized to the rising/falling edge of the clock. Thus, the packet receiver will drive sample data at the rising/falling edge of the clock. The router will drive and sample data at the rising edge of clock.
- Each output port data_out_X (data_out_0, data_out_1, data_out_2) is internally buffered by a FIFO of 1 byte width and 16 location depth.
- The router asserts the vld_out_X (vlld_out_0, vld_out_1 or vld_out_2) signal when valid data appears on the vld_out_X (data_out_0, data_out_1 or data_out_2) output bus. This is a signal to the packet receiver that valid data is available on a particular router.
- The packet receiver will then wait until it has enough space to hold the bytes of the packet and then respond with the assertion of the read_enb_X (read_enb_0, read_enb1 or read_enb_2) signal that is an input to the router.
- The read_enb_X (read_enb0, read_enb_1 or read_enb_2) input signal can be asserted on the rising/falling clock edge in which data are read from the data_out_X (data_out_0, data_out_1 or data_out_2) bus.
- As long as the read_enb_X (read_enb_0, read_enb_1 or read_enb_2) signal remains active, the data_out_X (data_out_0, data_out_1 or data_out_2) bus drives a valid packet byte on each rising clock edge.
- The packet receiver cannot request the router to suspend data transmission in the middle of the packet. Therefore, the packet receiver must assert the read_enb_X (read_enb_0, read_enb_1 or read_enb_2) signal only after it ensures that there is adequate space to hold the entire packet.
- The read_enb_X (read_enb_0, read_enb_1 or read_enb_2) must be asserted within 30 clock cycles of the vld_out_X (vld_out_0, vld_out_1 or vld_out_2) being asserted. Otherwise, there is too much congestion in the packet receiver.
- The DUV data_out_X (data_out_0, data_out_1 or data_out_2) bus must not be tri-stated (high Z) when the DUV signal vld_out_X (vld_out_0, vld_out_1or vld_out_2) is asserted (high) and the input signal read_enb_X (read_enb_0, read_enb_1 or read_enb_2) is also asserted high.

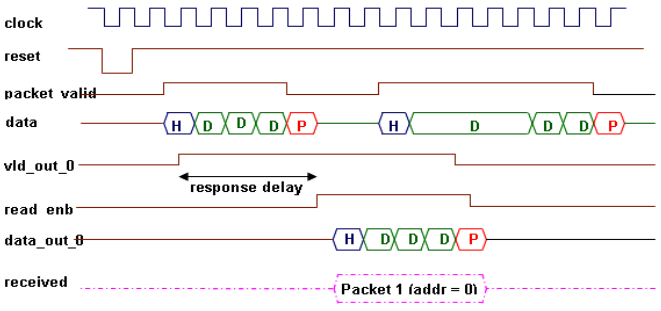

Figure 5- Router output Protocol

IV. FIVE PORT ROUTER ARCHITECTURE

The Five Router Design is done by using of the three blocks .the blocks are 8-Bit Register, Router controller and output block. the router controller is design by using FSM design and the output block consists of three fifo's combined together the fifo's are store packet of data and when u want to data that time the data read from the FIFO's. In this router design has four outputs that is 8-Bit size and one 8_bit data port it using to drive the data into router we are using the global clock and reset signals, and the err signal and suspended data signals are output's of the router .the FSM controller gives the err and suspended_data_in signals .this functions are discussed clearly in below FSM description

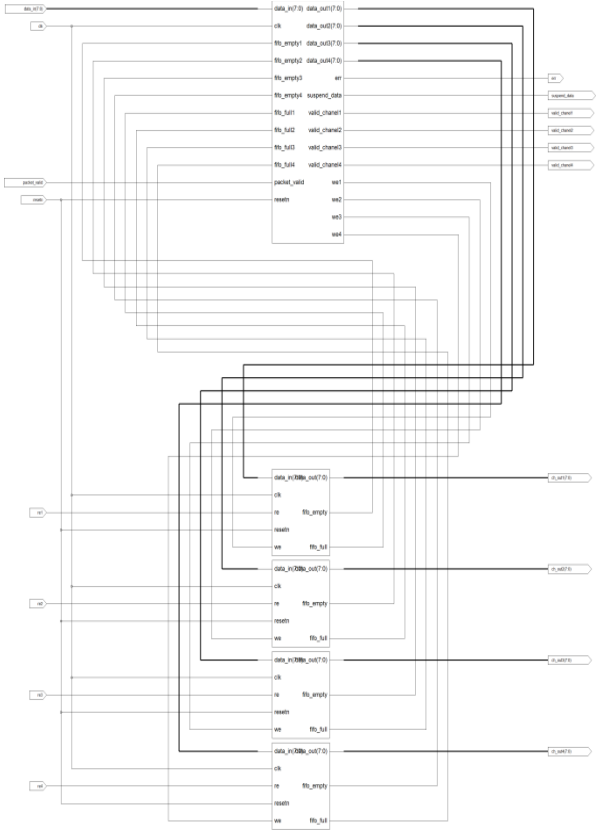

Figure 6- Five Port Router Architecture

The router_reg module contains the status, data and parity registers for the Network router_1x3.

These registers are latched to new status or input data through the control signals provided by the fsm_router.

There are 3 fifo for each output port, which stores the data coming from input port based on the control signals provided by fsm_router module.

The fsm_router block provides the control signals to the fifo, and router_reg module. The Router blocks Diagram shown below fig…

Router blocks are

- Register
- Router controller(FSM)
- FIFO Output Block

Register Block

This module contains status, data and parity registers required by router. All the registers in this module are latched on rising edge of the clock.

Data registers latches the data from data input based on state and status control signals, and this latched data is sent to the fifo for storage. Apart from it, data is also latched into the parity registers for parity calculation and it is compared with the parity byte of the packet. An error signal is generated if packet parity is not equal to the calculated parity.

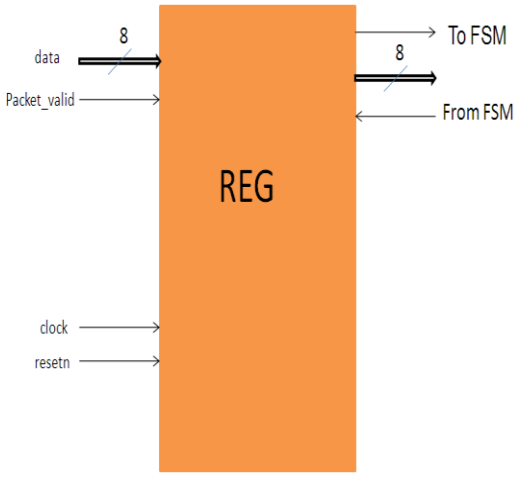

Figure 7- Four Port Router Register

If resetn is low then output (dout, err, parity_done and low_packet_valid) are low.

The output parity_done is high

when the input ld_state is high and (fifo-full and packet_valid) is low or when the input laf_state and output low_packet_valid both are high and the previous value of parity_done is low. It is reseted to low value by reset_int_reg signal.

The output low_packet_valid is high

- when the input ld_state is high and packet_valid is low.
- It is reseted to low by reset_int_reg signal.

First data byte i.e., header is latched inside the internal register first_byte when detect_add and packet_valid signals are high, So that it can be latched to output dout when lfd_state signal goes high.

Then the input data i.e., payload is latched to output dout if ld_state signal is high and fifo_full is low.

Then the input data i.e., parity is latched to output dout if ld_state signal is high and fifo_full is low.

The input data is latched to internal register full_state_byte when ld_state and fifo_full are high; this full_state_byte data is latched inside the output dout when laf_state goes high.

Internal parity register stores the parity calculated for packet data, when packet is transmitted fully, the internal calculated parity is compared with parity byte of the packet. An error signal is generated if packet parity is not equal to the calculated parity.

The register of "Network Router" is show ni above fig. in this register block diagram inputs are taken form outside the world and drives the FSM controller based an this register outputs controller is working.

Router Controller (Fsm)

This module generates all the control signals when new packet is sent to router. These control signals are used by other modules to send data at output, writing data into the fifo.

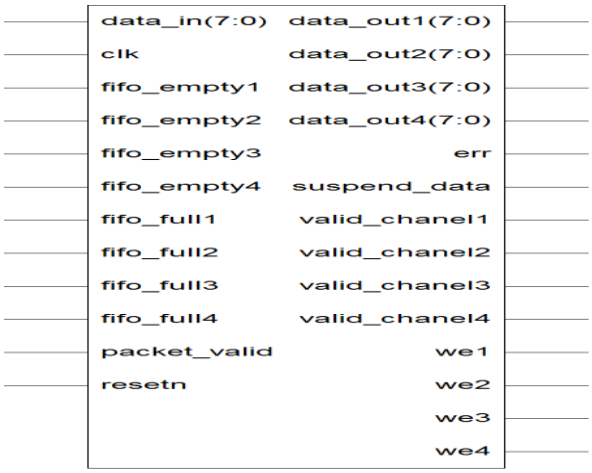

Figure 8- Five Router Controller Block

The 'fsm_router' module is the controller circuit for the router.

V. SIMULATION RESULTS

The below figures shows the simulation results of test cases applied to the DUT . Figure 9.1shows the response of the device for the control test case at the usb interface. Figure 6.2 shows the master transmitter sending random data to the external slave device.

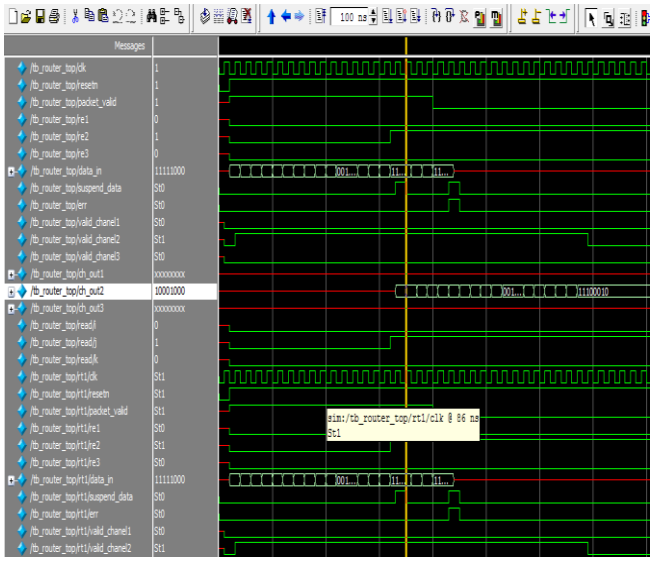

Figure 9 Simulation results of Four Port Router for Noc.

Data_in[1:7] = input =deferent data packets
Data_in[0] = 11111001
Output : Ch_out1 = 8'bXXXXXXXX
Ch_out2 = data_in,
Ch_out3 = 8'bXXXXXXXX.
After 80ns again applying inputs immediately output obtained, because of no delay elements.
Data_in[1:7] = input =deferent data packets
Data_in[0] = 11111000
Output : Ch_out2 = 8'bXXXXXXXX,
Ch_out1 = data_in,
Ch_out3 = 8'bXXXXXXXX.

Code Coverage

The tests are written according the testplan and the tests are made into a single regression which contains all the tests and a perl script is written to run all the tests. This script can also run with the coverage option to see the coverage results. It creates a html page which contains the results of the tests that are run in that regression. When this script is run with the coverage option, it creates a directory with name "report" and dumps all the coverage html reports generated by the Questa tool. The extent of verification cannot be judged manually. The Questa tool offers coverage aspects on the written RTL code. Code coverage measures the amount of HDL code that has been exercised by all the tests. It not only checks the coverage in terms of lines covered, but the states covered in state machines, the values of the signals that are toggled etc. By running the tests in the testplan the coverage that is attained is as follows

Scope	TOTAL	Cvg	Cover	Statement	Branch	Expression	Condition	Toggle	FSM State	FSM Trans
TOTAL	81.8%	--	--	79.9%	77.9%	82.2%	69.5%	60.1%	100.0%	100.0%
Router_top	83.3%	--	--	78.3%	72.5%	81.4%	70.3%	80.4%	100.0%	100.0%
Router_reg	77.0%	--	--	86.1%	91.0%	96.2%	67.7%	40.2%	--	--
router_controller	76.6%	--	--	79.4%	79.6%	86.2%	71.1%	48.4%	100.0%	100.0%
router_ouput_drive	76.8%	--	--	81.8%	84.6%	--	66.7%	75.9%	--	--
inst_rx_fifo	64.1%	--	--	77.3%	80.8%	--	66.7%	20.7%	--	--
slave_tx_fifo	70.6%	--	--	81.8%	84.6%	--	66.7%	44.8%	--	--
slave_rx_fifo	62.4%	--	--	77.3%	80.8%	--	66.7%	12.1%	--	--

A total of 81.8% overall coverage was attained which says that 81.8% of the DUT was verified. The rest of the coverage percentage is due to some part of the code which is kept for some invalid signalling. This percentage of the missed coverage is due to the conditions that has different possibilities like a condition ((x==1) or (y==1)). Tool creates the possible combinations of the condition as 00,01,10,11. Attaining all the conditions is cumbersome to achieve. Another reason is that the invalid FSM transitions that are picked up by the tool. The tool extracts the state machine from the HDL code. In that process some invalid transitions are created that are not present in the state machine which is designed. We can also eliminate the transitions from the coverage and get the coverage 100%.

Functional Coverage

Functional coverage is the determination of how much functionality of the design has been exercised by the verification environment

Code Coverage: This will give information about how many lines are executed, how many times expressions, branches executed. This coverage is collected by the simulation tools.

Both of them have equal importance in the verification. 100% functional coverage does not mean that the DUT is completely exercised and vice-versa.

Verification engineers will consider both coverages to measure the verification progress. we would like to explain this difference with a simple example. Let's say, the specification talks about 3 features, A, B, and C. And let's say that the RTL designed coded only

feature A and B.If the test exercises only feature A and B, then you can 100% code coverage. Thus, even if you have 100% code coverage, you have a big hole (feature C) in the design.So, the verification engineer, has to write functional coverage code for A, B and C and 100% functional coverage means, there are tests for all the features, which the verification engineer has thought of.

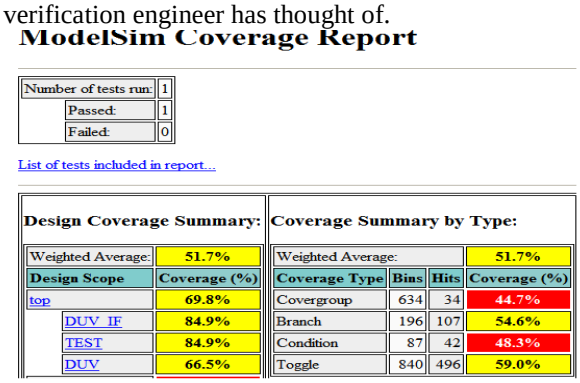

Figure 10:Coverage report using one test case.

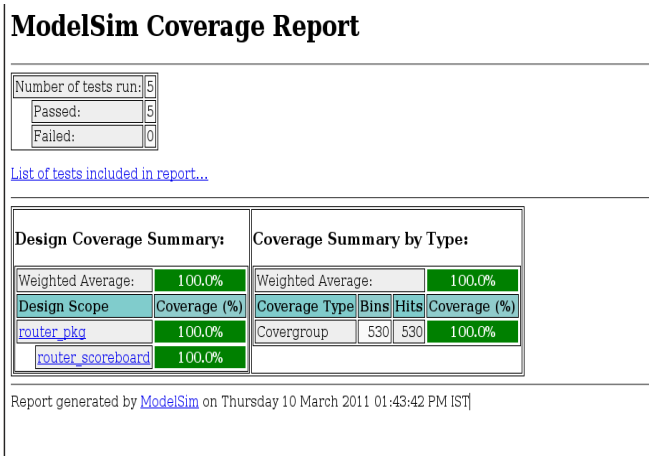

Figure 11 Coverage report results of Four Port Router for Noc.

In figure 9 only one test case has been passed so the coverage is 51.7% and out of 634 bins only 34 bins has been hitted.Inorder to increase the coverage , increase the testcases for which the bins hasnot been covered this was shown in figure 10 .Atlast in figure 11 , 5 testcases has been passed and all the bins(530) has been covered so,finally 100% coverage has been reached.

Synthesis Results

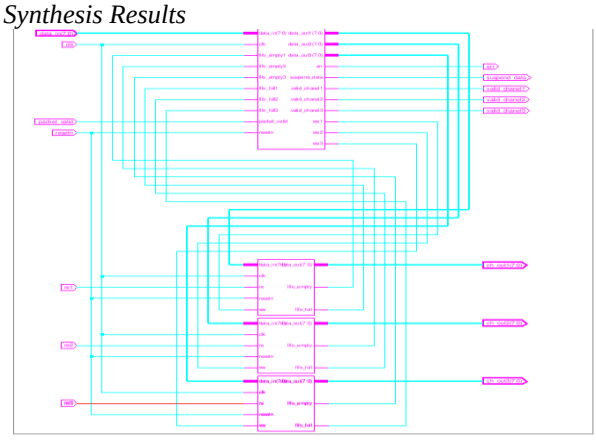

Figure 12 Synthesis results of Four Port Router for Noc.

Design Summary

Number of errors: 0
Number of warnings: 2

Logic Utilization:

- Total Number Slice Registers: 100 out of 7,168 1%
- Number used as Flip Flops: 52
- Number used as Latches: 48
- Number of 4 input LUTs: 188 out of 7,168 2%

Logic Distribution:

- Number of occupied Slices: 129 out of 3,584 3%
- Number of Slices containing only related logic: 129 out of 129 100%
- Number of Slices containing unrelated logic: 0 out of 129 0%

*See NOTES below for an explanation of the effects of unrelated logic

- Total Number of 4 input LUTs: 236 out of 7,168 3%
- Number used as logic: 188
- Number used for Dual Port RAMs: 48
- (Two LUTs used per Dual Port RAM)
- Number of bonded IOBs: 43 out of 141 30%
- IOB Flip Flops: 24
- Number of GCLKs: 3 out of 8 37%
- Total 9.269ns (4.077ns logic, 5.192ns route)
  (44.0% logic, 56.0% route)

Total memory usage is 121748 kilobytes

VI. CONCLUSION

As the functional verification decides the quality of the silicon, we spend 60% of the design cycle time only for the verification/simulation. In order to avoid the delay and meet the TTM, we use the latest verification methodologies and technologies and accelerate the verification process. This project helps one to understand the complete functional verification process of complex ASICs an SoC's and it gives opportunity to try the latest verification methodologies, programming concepts like Object Oriented Programming of Hardware Verification Languages and sophisticated EDA tools, for the high quality verification.

In this Four Port Router paper we Design and verified the functionality of Router with the latest Verification methodology i.e.,System Verilog and observed the code coverage and functional coverage of Router by using coverpoints ,cross and different test cases like constrained, weighted and directed testcases.By using these testcases I improved the functional coverage of Router. In this I used one master and eight slaves to monitor the Router.Thus the functional coverage of Router was improved.

The results shows that System Verilog methodology can be used to make reusable test benches successfully. Large part of the test bench is made reusable over multiple projects.even though this reusablity is limited to the interfaces. A large class of devices that are build on these inerfaces can be

verified successfully. Once these components are made the amount of time required to build test benches for other projects can be reduced a lot.

VII. FUTURE WORK

This paper used System verilog i.e., the technology used is direct testcases,randomized testcases ,ovm for verification eventhough the coverage is 100% there may be some errors which cannot be shown so inoder to overcome this the new technology of System verilog i.e.,OVM and UVM.In the coming future the Router can be done by using OVM and UVM.

ACKNOWLEDGEMENT

TALLURI DEEPAK would like to thank *Mr. T. PATTALU NAIDU*, Assistant professor, Department of ECE, *Avanthi Institute of Engineering and Technology*, who had been guiding throughout the project and supporting me in giving technical ideas about the paper and motivating me to complete the work efficiently and successfully.

REFERENCES

[1] L. Benini and G. D. Micheli, "Networks on chips: a new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70 – 78, 2002.
[2] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar, "An 80-tile sub-100-W Tera FLOPS processor in 65-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 29 –41, Jan. 2008.
[3] P. Golani, G. Dimou, M. Prakash, and P. Beerel, "Design of a high speed asynchronous turbo decoder," in *Proc. 13th IEEE Int. Symp. on Asynchronous Circuits and Systems*, pp. 49 - 59, Mar. 2007.
[4] N. Onizawa, V. Gaudet, and T. Hanyu, "Low-energy asynchronous inter leaver for clockless fully parallel LDPC decoding," *IEEE Trans. on Circuits and Systems I: Regular Papers*, vol. 58, no. 8, pp. 1933 –1943, Aug. 2011.
[5] A. Lines, "Asynchronous interconnect for synchronous SoC design," *IEEE Micro*, vol. 24, no. 1, pp. 32 – 41, Jan.-Feb. 2004.
[6] M. Horak, S. Nowick, M. Carlberg, and U. Vishkin, "A low overhead asynchronous interconnection network for gals chip multiprocessors," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 494 –507, April 2011.
[7] D. Rostislav, V. Vishnyakov, E. Friedman, and R. Ginosar, "An asynchronous router for multiple service levels networks on chip," in *Proc. 11th IEEE Int. Symp. on Asynchronous Circuits and Systems*, pp. 44 - 53, Mar. 2005.
[8] T. Bjerregaard and J. Sparsø, "Implementation of guaranteed services in the MANGO clockless network-on-chip," *IEE Proc. Computers and Digital Techniques*, vol. 153, no. 4, pp. 217 – 229, July 2006.
[9] J. Bainbridge and S. Furber, "CHAIN: a delay-insensitive chip area interconnect," *IEEE Micro*, vol. 22, no. 5, pp. 16 – 23, Sep.-Oct. 2002.
[10] D. Lattard, E. Beigne, C. Bernard, C. Bour, F. Clermidy, Y. Durand, J. Durupt, D. Varreau, P. Vivet, P. Penard, A. Bouttier, and F. Berens, "A telecom baseband circuit based on an asynchronous network-on-chip," in *Digest of Technical Papers, IEEE 2007 Int. Solid-State Circuits Conference*, pp. 258 -601, Feb. 2007.
[11] A. Alhussien, C. Wang, and N. Bagherzadeh, "A scalable delay insensitive asynchronous NoC with adaptive routing," in *Proc.IEEE 17th Int. Conf. on Telecommunications*, pp. 995 - 1002, Apr.2010.
[12] Y. Thonnart, P. Vivet, and F. Clermidy, "A fully-asynchronous low power framework for GALS NoC integration," in *Proc. Design, Automation Test in Europe Conference Exhibition 2010*, pp. 33 - 38, Mar. 2010.
[13] W. Song and D. Edwards, "A low latency wormhole router for asynchronous on-chip networks," in *Proc. 15th Asia and South Pacific Design Automation Conference*, pp. 437 - 443, Jan. 2010.
[14] D. Gebhardt, J. You, and K. S. Stevens, "Link pipelining strategies for an application-specific asynchronous NoC," in *Proc. Fifth IEEE/ACM Int. Symp. on Networks on Chip*, pp. 185 - 192, May 2011.
[15] K. Christensen, P. Jensen, P. Korger, and J. Sparsø, "The design of an asynchronous TinyRISC TR4101 microprocessor core," in *Proc. Fourth Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pp. 108 - 119, Mar.-Apr. 1998.
[16] J. Sparsø and S. Furber, "Principles of asynchronous circuit design: a systems perspective," *books.google.com*, Jan. 2001.
[17] M. Dean, T. Williams, and D. Dill, "Efficient self-timed level encoded 2-phase dual-rail (LEDR)," *Advanced Research in VLSI*, pp. 55 – 70, 1991.