Faculty of Engineering
Cairo University

Functional Verification of Narrow-band IOT Physical Layer Uplink Transmitter

Faculty of Engineering Cairo university
Elecrtonics and Electrical Communication Department

Under the supervision of
Prof. Hassan Mostafa

Si-Vision mentors

Eng.Hussien Galal Eng.Sameh El-Ashry

Functional Verification of Narrow-band IOT Physical Layer Uplink Transmitter

Faculty of Engineering Cairo university
Elecrtonics and Electrical Communication Department

Under the supervision of
Prof. Hassan Mostafa

Si-Vision mentors

Eng.Hussien Galal **Eng.Sameh El-Ashry**

By:
Abdulrahman Tarek
Ahmed Mohamed Mahmoud Khalil
Ahmed Taha Abdelrahman
Amira Yehia Ahmed
Mahmoud Mohamed El-Araby
Omnia Tayseer Mohamed

August 2020

Abstract

Increased design complexity has resulted in the need for efficient verification. The verification process is crucial for discovering and fixing bugs prior to fabrication and system integration. However, as designs increase in complexity, the use of traditional verification techniques with VHDL and Verilog may fall short to provide a proper toolset.
Narrowband IoT (NBIoT) is a Low Power Wide Area (LPWA) technology that works virtually anywhere. It connects devices more simply and efficiently on already established mobile networks, and handles small amounts of fairly infrequent 2way data, securely and reliably. It has many advantages: very low power consumption,excellent extended range in buildings and underground, easy deployment into existing cellular network architecture, network security amp; reliability, lower component cost.
This thesis explores the use of the universal verification methodology (UVM) to verify the Physical Layer of NB-IOT LTE Uplink digital Transmitter using complete environment for the chain.

Acknowledgement

The success and final outcome of this project required a lot of guidance and assistance from many people and we are extremely privileged to have got this all along the completion of our project. All that we have done is only due to such supervision and assistance and we would not forget to thank them.
First, to our advisor, **Dr. Hassan Mostafa** and our advisors **Eng.Hussien Galal, Si- Vision**, Egypt and **Eng. Sameh El-Ashry, Si-Vision**, Egypt for their patient guidance, enthusiastic encouragement and useful critiques of this project work.
Certainly, we will never find words enough to express the gratitude we owe first to Allah and to our families. Their tender love and support have always been the cementing force for building what we achieved. The all-round support rendered by them provided the much needed stimulant to sail through the phases of stress and strain.
Finally, Thanks are due to previous GP members (the designers). Their thesis was supportive, informative and guided us in many critical issues

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Over the past decade, there has been a tremendous growth in the number of wireless devices. As wireless technology matures, this number is expected to grow at an even higher rate with the goal being able to connect every physical device to the internet. This presents a huge opportunity for wireless system designers as 99.4 percent of the physical devices are still unconnected. Machine to machine (M2M) communication is needed where devices would communicate with each other without any human interaction. It is expected that the revenue from M2M devices will grow from \$200 million in 2011 to \$1.2 trillion in 2022. Some of the services which might need machine type communication (MTC) include consumer electronics, security, public safety, automotive, utilities, remote maintenance, payments, health and smart cities. Until now, most of the wireless technologies were focused on improving the quality and performance of consumer electronic devices intended for human communication. MTC, however, is different from human communication as the throughput/delay/power requirements vary to a great extent based on the application and thus, require different connectivity solutions.

While services involving consumer electronics, building security and maintenance can be served by a local area network (LAN), services such as remote maintenance (sensors, vending machines etc), utilities (power, gas, water, heating etc), smart cities etc need a wide area network (WAN). Since the cellular technology is pretty mature and is already deployed in most parts of the world, cellular based MTC systems can provide a solution for the IoT applications which need a WAN. LTE is one of the latest and widely accepted, high speed wireless communication standard by 3rd generation partnership project (3GPP). The standard is revised regularly, in order to accommodate additional channel conditions and provide better connectivity with higher data rates and efficient resource utilization.

However, LTE was designed for high speed communication and is not optimized for applications that need to support a potentially large number of low-rate, low power and delay tolerant devices. These low cost devices, typically used in applications such as sensors, remote maintenance, tracking, health-care, utilities etc, are expected to have very low complexity, low mobility and low power consumption with a very long battery life. Hence, it is desirable to develop LTE based wireless systems; suitable for low data-rate and low power IoT applications.

Therefore, NB-IOT LTE is the suitable choice for this applications, this devided into two section IOT and NB-LTE.

1.1.1 IOT

In the early evolution, it is known as “Internet of Computers”; then changed to “Internet of People”; and recently, with the rapid development in the ICT, it is recognized as the “Internet of Things”. In the IoT, different devices and smart objects are included to expand the Internet and become accessible and uniquely identified. The connectivity is enhanced from “any-time, any-place” for “any-one” into “anytime, any-place” for “any-thing”. In the ICT innovations and economy developments, a significant focus has shifted to the IoT related technologies where it is widely considered as one of the most important infrastructures of their promotion and one of the future promise strategies. The main aim is to enable interaction and integration of the physical world and the cyber space.

The IoT is a new revolution in communication technology which means that everything, from tires to hairbrush, will be assigned a unique identifier so can be addressed, connected to other things and exchange information. There is no exact or standard definition of the IoT yet. There are different attempts such as, it is defined as “based on the traditional information carriers including the Internet, telecommunication network and so on, Internet of Things (IoT) is a network that interconnects ordinary physical objects with the identifiable addresses so that provides intelligent services". or, defined IoT as "a worldwide network of interconnected objects uniquely addressable, based on standard communication protocols", semantically as its origin expression is composed of two words: "Internet" and "Things". However, the true value of IoT is in its ability to connect a variety of heterogeneous devices including everyday existing objects, embedded intelligent sensors, context-aware computations, traditional computing networks and smart objects that differ in their design, systems, protocols, intelligence, applications, vendors and sizes.

1.1.2 NB-LTE

The earlier LTE-based standards (Cat-0 and Cat-1) that were proposed as solutions for MTC prior to Release 13 failed to secure a significant share of the IoT market. Unlike these, however, NB-IoT aims to meet the stringent targets for both low device modem cost of below $5 and long battery life in excess of 10 years, which are likely to make it much more successful. Characteristics cited by the GSM Association as making NB-IoT particularly attractive to users include:

- Low power consumption: current consumption of the order of 1nA enables devices to operate for up to 10 years on a single charging cycle.
- Low device unit cost and area to integrate into a unified IoT/MTC platform.
- Improved outdoor and indoor penetration coverage compared with existing wide area technologies.
- Secure connectivity and strong authentication.
- Optimized data transfer, supporting small, intermittent blocks of data.
- Network scalability to increase capacity.

1.1.2.1 Background

Realizing the importance of IoT for low-power and low cost applications with extended coverage and very long battery life, 3GPP introduced narrowband IoT (NB-IoT), as a part of their LTE-Release-13. Although NB-IoT is based on LTE, but it's a new radio-access technology as it is not fully backward compatible with the existing LTE devices. However, it can be easily integrated in the existing LTE network by allocating some of the time and frequency resources to NB-IoT.
NB-IoT occupies 180 kHz of spectrum, which is substantially smaller than LTE bandwidths of 1.4–20 MHz

Spectrum, NB-IoT is designed to support three different deployment scenarios:

1. Stand alone: utilizing a 200 kHz band used by global system for mobile communication (GSM) frequencies as shown in see 1.1.

2. Guard band: occupying a 180 kHz wide physical resource block in the guard band of existing LTE carrier band as shown in see 1.2.

3. In band: occupying one physical resource block within the LTE carrier bandwidth shown in see 1.3.

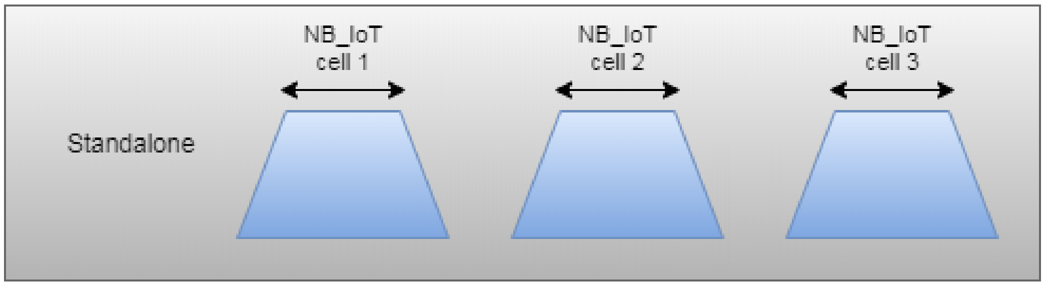


Figure 1.1: Standalone of NBIOT in GSM

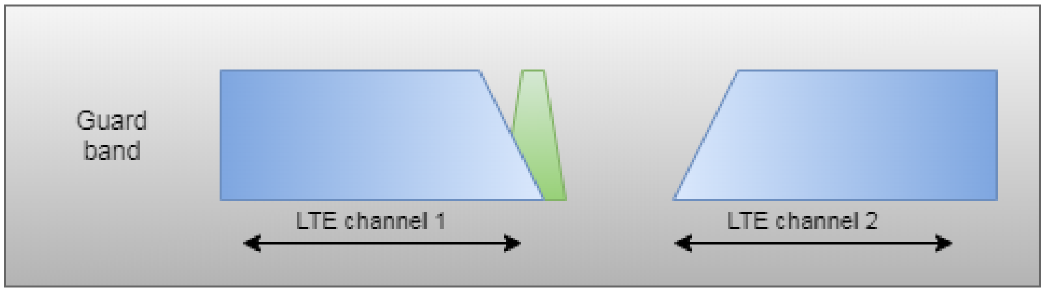


Figure 1.2: Introduction/NBIOT in guard band

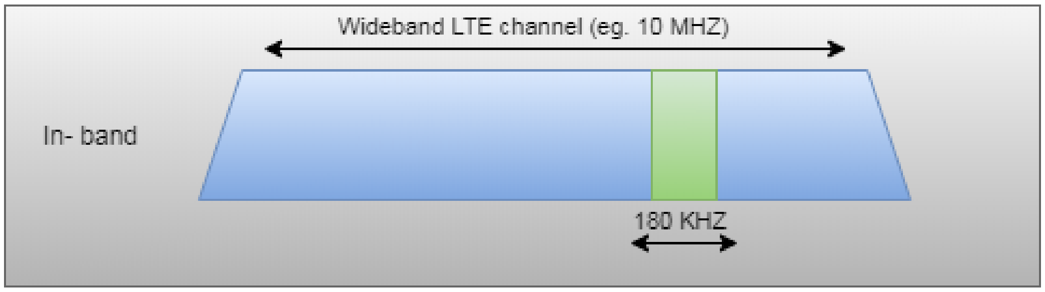


Figure 1.3: Introduction/NBIOT in band

Thus, NB-IoT reuses the existing LTE design with respect to physical layer processing which reduces the time required to develop the NB-IoT devices to a great extent.

1.1.2.2 Purpose

As discussed in the previous section, NB-IoT provides a provision to allocate a single resource block (RB) to each user, instead of at-least six resource blocks in existing LTE. A single RB corresponds to a bandwidth of 180 kHz (and hence the name narrow-band) as compared to the minimum bandwidth of 1.4 MHz in conventional LTE systems. The resource block concept is discussed in later chapters. The purpose of this thesis is to verify a new, low power, wireless communication system targeted towards IoT applications, having very low throughput requirements with relaxed transmission delay requirements. The overall system, similar to NB-IoT, will be based on LTE and will have very low complexity, consume low power and have a long battery life. This system is able to allocate a very narrow bandwidth of 15 kHz to each user while retaining the existing LTE physical layer structure.

1.1.2.3 Problem Formulation

NB-IoT standard allocates one resource block per user irrespective of the throughput requirements of the application. A single resource block comprises of twelve subcarriers, each separated by 15kHz in frequency, giving a bandwidth of 180kHz. This might not be an efficient way of using the available spectrum when the throughput requirement is extremely low. In extreme cases, when the data-rate requirement is very low, we might as well allocate each subcarrier to a different user. This would enable a very efficient use of the available spectrum with a multiplexing gain of twelve. The transmitter in this case will be a NB-IoT transmitter with each subcarrier carrying data for a separate user.

1.1.2.4 Limitations

In this thesis, we study the physical layer properties of the transmitter architecture, with a focus on the uplink path of the system. Problems related to resource allocation and higher layer management is not discussed as a part of the thesis. Further, since the target data-rate and power requirements are low, only lower order modulation schemes, for instance, binary phase-shift keying (BPSK) and quadrature phase-shift keying (QPSK).

1.1.2.5 Frame Structure

Downlink, uplink and sidelink transmissions are organized into radio frames with $T_f = T_s * 307200 = 10ms$ duration where T_s is the number of time units =1 / (1500*2048) seconds.

Three radio frame structures are supported

- Type 1, applicable to FDD only
- Type 2, applicable to TDD only
- Type 3, applicable to LAA secondary cell operation only

The difference between FDD and TDD is that in TDD system (Time division duplex) same frequency band FC is used by both Transmit and receive path at different time instants while FDD system (Frequency division duplex) different frequency bands Fc1 and Fc2 are used by transmit and receive paths at same time instant as shown in see 1.4.

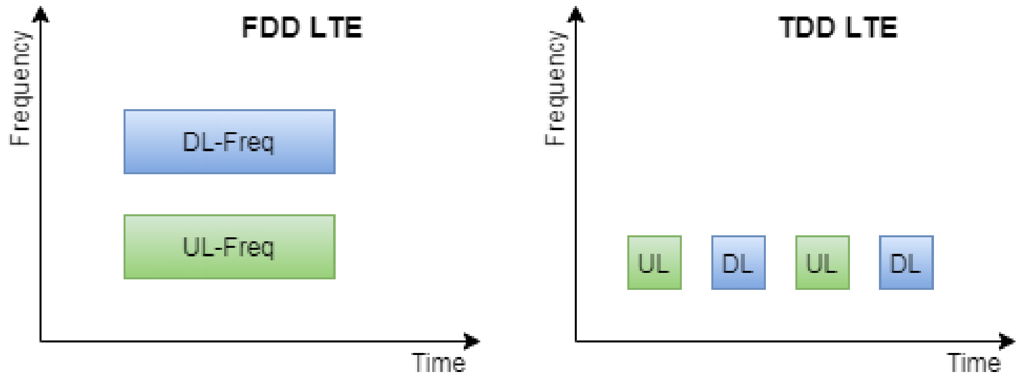


Figure 1.4: Introduction/FDD and TDD differences.png

In NBIOT-LTE we use frame structure type 1 which is applicable to both full duplex and half duplex FDD only ,Each radio frame is $T_f = 307200 * T_s = 10ms$ long and consists of 10 subframes of length $30720 * T_s = 1ms$ numbered from 0 to 9.

For subframes using $\delta f = 15kHz$, subframe i is defined as two slots, 2i and 2i +1, of length $T_{slot} = 15360 * T_s = 0.5ms$ each.
For FDD, 10 subframes are available for downlink transmission and 10 subframes are available for uplink transmissions in each 10 ms interval. Uplink and downlink transmissions are separated in the frequency domain.
In half-duplex FDD operation, the UE cannot transmit and receive at the same time while there are no such restrictions in full-duplex FDD.

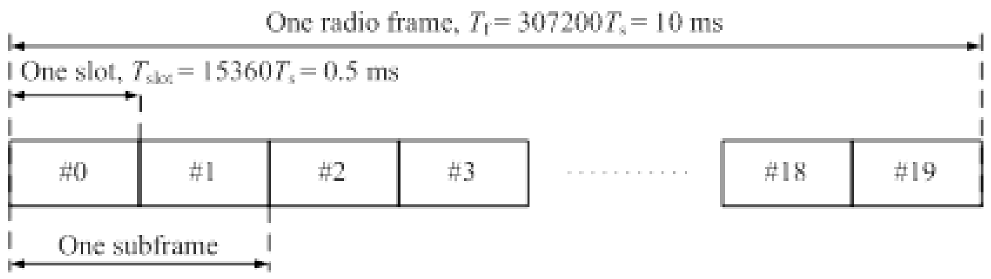

One radio frame, $T_f = 307200T_s = 10$ ms
One slot, $T_{slot} = 15360T_s = 0.5$ ms
#0
#1
#2
#3
#18
#19
One subframe

Figure 1.5: Frame structure type 1

1.1.2.6 Slot Structure

Uplink waveform in LTE-NB is the same as in legacy LTE Uplink. That is, LTE-NB Uplink is also using SC-FDMA. But there are some differences between LTE-NB and legacy LTE in terms of structure of uplink signal. In addition, there is a new unit called RU (Resource Unit) that exists in LTE-NB but not used in legacy LTE.

1.1.2.7 Subcarrier Spacing

There are two different types of sub-carrier spacing in LTE-NB. One is 15 KHz (this is same as in legacy LTE) and the other one is 3.75 KHz

Subcarrier spacing	N_{sc}^{UL}	T_{slot}
$\delta f = 3.75KHZ$	48	61440* T_s
$\delta f = 15KHZ$	12	15360* T_s

Table 1.1: Subcarrier spacing and corresponding number of subcarriers per UL

In this thesis we are using 15 KHz subcarrier spacing.

1.1.2.8 Resource Grid

The transmitted signal in each slot is described by one or several resource grids of $N_{RB\_UL} * N_{SC\_RB}$ subcarriers and N_{symbol_UL} SC-FDMA symbols. The quantity N_{RB_UL}depends on the uplink transmission bandwidth configured in the cell and shall fulfil.

$$N_{RB}^{min,UL} \leq N_{RB}^{UL} \leq N_{RB}^{max,UL}$$

Where $N_{RB}^{min,UL}$ and $N_{RB}^{max,UL}$ are the smallest and largest uplink bandwidths. N_{RB}^{UL} is number of resource blocks in uplink, N_{SC_RB} is Number of subcarriers resource blocks, N_{symbol_UL} is number of symbols per uplink, $N_{RB}^{min,UL}$ is the minimum number of resource blocks in uplink and $N_{RB}^{max,UL}$ is the maximum number of resource blocks in uplink. Resource grid in a frame based on 15 KHz subcarrier spacing, Number of subcarriers is easily calculated to 12 sub-carriers within 180 KHz BW (LTE-NB System Bandwidth), there are 20 slots within a radio frame. Representing this case in a graphical format, it becomes as shown in see 1.6. Basically this is the same as in legacy LTE uplink resource grid.

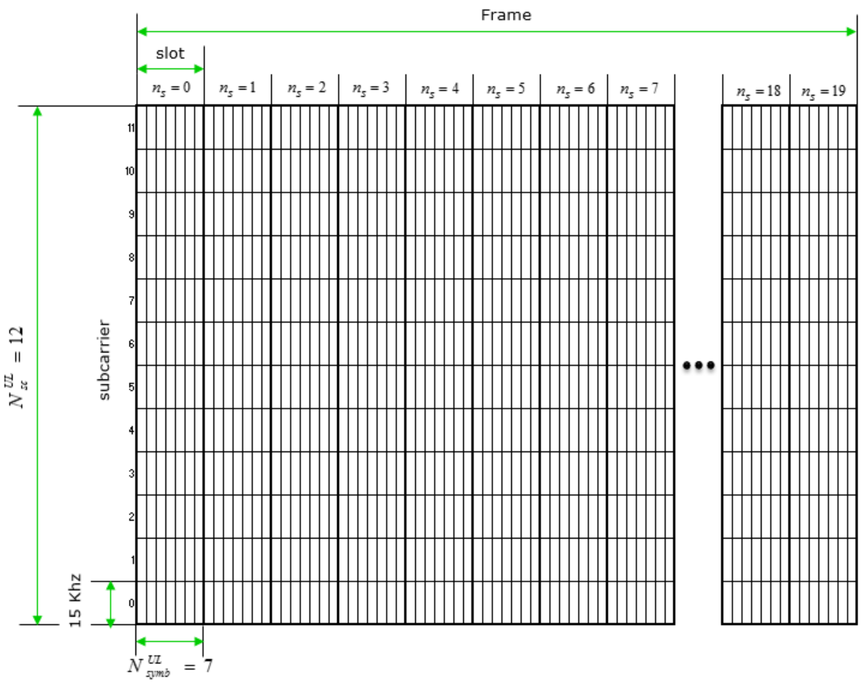


Figure 1.6: Resource grid with 15kHz spacing

If a 3.75 KHz subcarrier spacing resource grid frame is drawn, 48 subcarriers exist in the 180 KHz BW (LTE-NB System Bandwidth), with 5 slots within a radio frame. Translating this case into a graphical format, it would become as the following see 1.7.

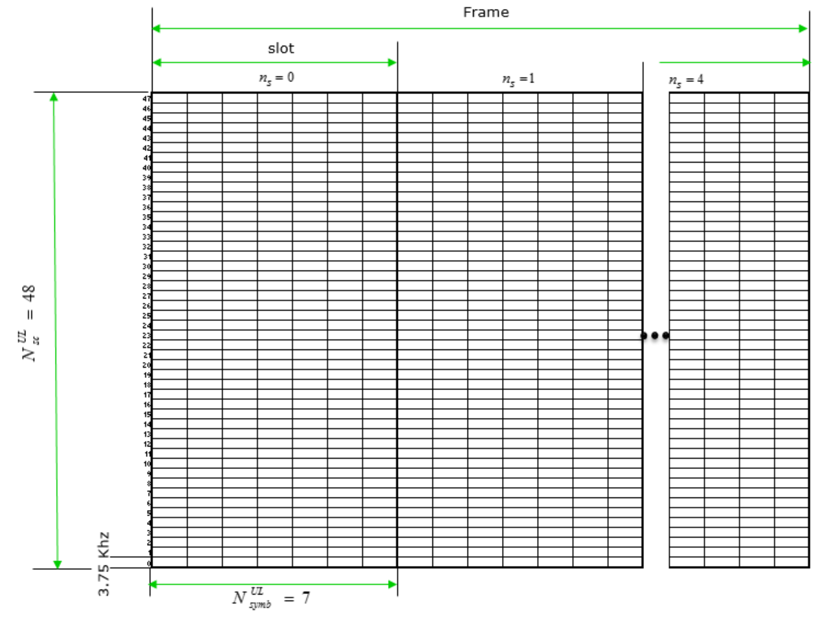


Figure 1.7: Resource grid with 3.75kHz spacing

Comparing 3.75 KHz resource grid and 15 KHz resource grid in the two figures 1.6 and 1.7, some points can be noticed the followings:

- Sub-carrier spacing in 3.75 KHz became narrower by 4 times comparing to 15 KHz resource grid.
- Symbol length in 3.75 KHz resource grid became longer by 4 times comparing to 15 KHz resource grid (this is a basic property for OFDM. The narrower sub-carrier spacing is, the longer symbol length).
- Length of a Radio Frame is defined to be the same (10 ms) in both 3.75 KHz resource grid and 15 KHz resource grid.
- The number of slots within a radio frame in 3.75 KHz resource grid became smaller by 4 times comparing to 15 KHz resource grid.
- The number of OFDM symbols within a slot is the same (=7) in both 3.75 KHz resource grid and 15 KHz Grid.

NPUSCH	δf	N_{sc}^{RU}	N_{UL}^{slots}	N_{UL}^{symbol}
1	15 KHz	1	16	7
		3	8	
		16	4	
		12	2	

Table 1.2: Number of subcarriers per RU in each carrier spacing

1.1.2.9 Resource unit

A kind of new concept in LTE-NB UL resource assignment comparing to legacy LTE, LTE-NB introduce a new resource unit called RU (Resource Unit) as a basic unit for NPUSCH allocation. This unit can take several different types of configurations as defined in the table 1.2, noticing that NPUSCH format 1 and 15 KHz subcarrier spacing is assumed in this thesis.

For NPUSCH format 1, Sub Carrier Spacing = 15 KHz, the number of sub carrier in one RU is 1 and the number of slots within the RU is 16. The graphical representation that I interpreted is as shown below see 1.8 the table the red slots will send to the eNodeB.

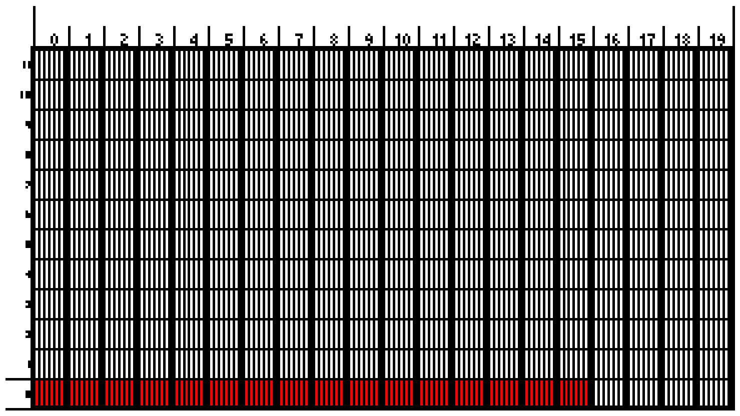


Figure 1.8: NPUSH format1 15KHZ subcarrier spacing

1.1.2.10 Resource Allocation

In standard 213, the narrow band section describes The DCI, which is a Downlink Control Information. The resource allocation information in uplink, DCI format N0 for NPUSCH transmission indicates to a scheduled UE

- A set of contiguously allocated subcarriers of a resource unit determined by the Subcarrier indication field in the corresponding DCI.
- A number of resource units determined by the resource assignment field in the corresponding.
- A repetition number determined by the repetition number field in the corresponding DCI.

Field	num of Bits
Flag for format N0/format N1 differentiation	1
Subcarrier indication	6
Resource Assignment	3
Scheduling delay	2
Modulation and coding scheme	4
Redundancy version	1
Repetition number	3
New data indicator	1
DCI subframe repletion number	2
Total number of Bits	23

Table 1.3: DCI format N0 information

- For the second field "subcarrier indication":

Subcarrier indication field (I_{su})	Set of Allocated subcarriers (n_{sc})
0-11	I_{su}
12-15	3(I_{su})+0,1,2
16-17	6(I_{su})-16 +0, 1, 2, 3 , 4 , 5
18	0, 1, 2, 3, 4 ,5 , 6, 7 ,8 ,9 10 ,11
19-63	Reserved

Table 1.4: Allocated subcarriers

I_{su}	n_{sc}
0	1
1	2
2	3
3	4
4	5
5	6
6	8
7	10

Table 1.5: Number of resource units for NPUSCH

This allows us to know Number of resource units.
- For the "Modulation and coding scheme number" field

MCS Index I_{MCS}	Modulation Order Q_m	TBS Index I_{TBS}
0	1	0
1	1	2
2	2	1
3	2	3
4	2	4
5	2	5
6	2	6
7	2	7
8	2	8
9	2	9
10	2	10

Table 1.6: Modulation order and transport block index

It indicates the modulation order whether it's QPSK or BPSK and from the Transport block index, the transport block size can be obtained from table see 1.6

I_{TBS}	I_{RU}							
	0	1	2	3	4	5	6	7
0	16	32	56	88	120	152	208	256
1	24	56	88	144	176	208	256	344
2	32	72	144	176	208	256	328	424
3	40	104	176	208	256	28	440	568
4	56	120	208	256	328	408	552	680
5	72	144	224	328	424	504	680	872
6	88	176	256	392	504	600	808	1000
7	104	224	328	472	584	712	1000	1224
8	120	256	392	56	680	808	1096	1384
9	136	296	456	616	776	936	1256	1544
10	144	28	504	680	872	1000	1384	1736
11	176	76	584	776	1000	1192	1608	2024
12	208	440	680	1000	1128	1352	1800	2280
13	224	488	744	1032	1256	1544	2024	2536

Table 1.7: Transport block size.

In our chain we made it on maximum transport block size 2536 in release14.

1.2 Digital design

The semiconductor industry is the aggregate collection of companies engaged in the design and fabrication of semiconductors. It formed around 1960, once the fabrication of semiconductor devices became a viable business. The industry's annual semiconductor sales revenue has since grown to over \$481 billion, as of 2018. The semiconductor industry is in turn the driving force behind the wider electronics industry, with annual power electronics sales of £135 billion (\$218 billion) as of 2011, annual consumer electronics sales expected to reach \$2.9 trillion by 2020, tech industry sales expected to reach \$5 trillion in 2019, and e-commerce with over \$29 trillion in 2017.

1.2.1 Digital design flow

see 1.9 presents a conceptual design flow from the specifications to the final product. The flow in the figure shows a top-down approach, the reality of an industrial development is much more complex, involving many iterations through various portions of the flow in the figure, until the final design converges to a form that meets the requirements of functionality, area, timing, power and cost. The design specifications are generally presented as a document describing a set of functionalities that the final solution will have to provide and a set constraint that it must satisfy. In this context, the functional design is the initial process of deriving a potential and realizable solution from these specifications and requirements. This is sometimes referred to as modeling and includes such activities as hardware/software trade off and micro-architecture design.

Cause the large scale of the problem, the development of a functional design is usually carried out using a hierarchical approach, so that a single designer can concentrate on a portion of the model at any given time. Thus, the architectural description provides a partition of the design in distinct modules, each of which contributes a specific functionality to the overall design. These modules have well defined input/output interfaces and protocols for communicating with the other components of the design. Among the results of this design phase is a high-level functional description, often a software program in C or in a similar programming language, that simulates the behavior of the design with the accuracy of one clock cycle and reflects the module partition.
It is used for performance analysis and also, as a reference model to verify the behavior of the more detailed designs developed in the following stages.

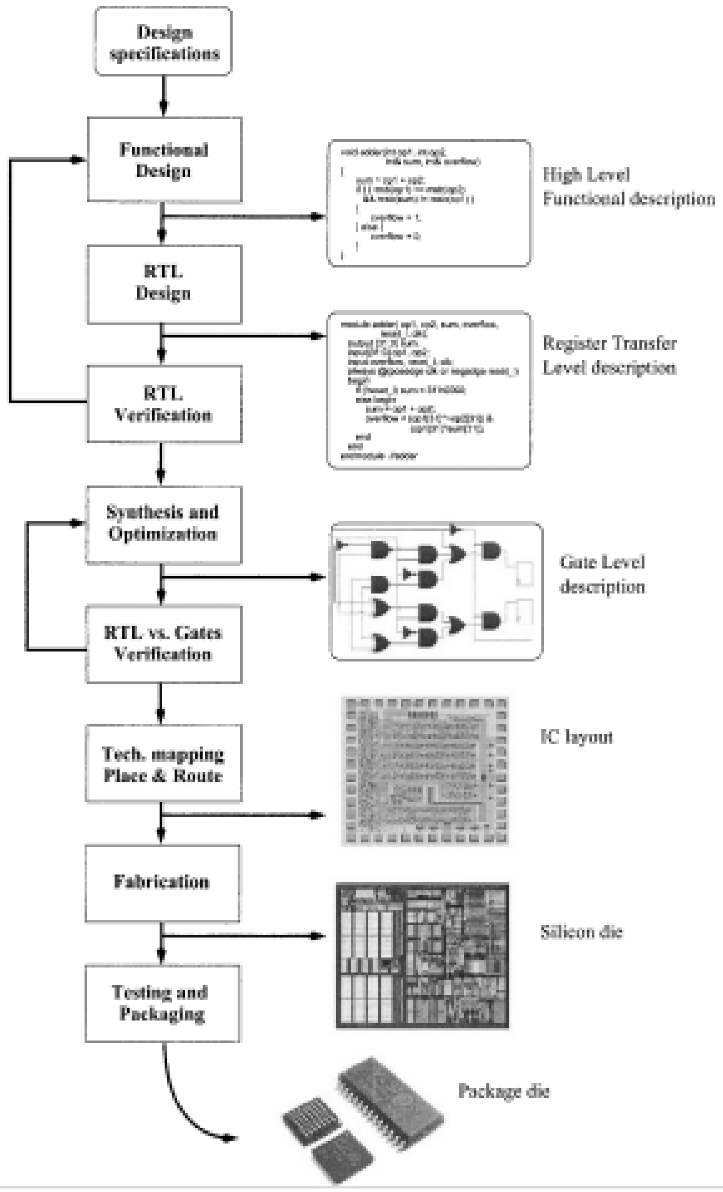


Figure 1.9: Conceptual design flow of a digital system

From the functional design model, the hardware design team proceeds to the Register Transfer Level (RTL) design phase. During this phase, the architectural description is further refined: memory elements and functional components of each model are designed using a Hardware Description Language (HDL). This phase also entails the development of the clocking system of the design and architectural trade-offs such as speed and power.

With the RTL design, the functional design of our digital system ends and its verification begins. **Functional verification** consists of acquiring reasonable confidence that a circuit will function correctly, under the assumption that no manufacturing fault is present. The underlying motivation is to remove all possible design errors before proceeding to the expensive phase of chip manufacturing. Each time functional errors are found, the model needs to be modified to reflect the proper behavior. During Functional verification, the verification team develops various techniques and numerous suites of tests to check that the design behavior corresponds to the initial specifications. When that is not the case, the functional design model needs to be modified to provide the correct behavior specified and the RTL design updated consequently. It is also possible that the Functional verification phase reveals incongruous or overlooked aspects in the original set of specifications and it is found that the specification document is to be updated instead of the RTL description.

In the diagram of see 1.9, Functional verification appears as one isolated phase of the design flow. However, in practical designs, the verification of the RTL model is carried on in parallel with the other design activities and it often lasts until chip layout. An overview of the verification methodologies that are common in today's industrial developments is presented in see chapter 3 .

The next design phase consists of **the Synthesis and Optimization** of the RTL design. The overall result of this phase is to generate a detailed model of a circuit, which is optimized based on the design constraints. For instance, a design could be optimized for power consumption or the size of its final realization (IC area) or for the ease of testability of the final product. The detailed model produced at this point describes the design in terms of its basic logic components, such as AND, OR, NOT or XOR, in addition to memory elements. Optimizing the netlist, or gate-level description, for constraints such as timing and power requirements is an increasingly challenging

aspect of current developments and it usually involves multiple iterations of trial-and-error attempts before reaching a solution that satisfies the requirements.
Such optimizations may, in turn, introduce functional errors that require additional function verification.

All the design phases, up to this point, have minimal support from Computer- Aided Design (CAD) software tools and are almost entirely handcrafted by the design and verification team. Consequently, they absorb a preponderant fraction of the time and cost involved in developing a digital system. Starting with synthesis and optimization, most of the activities are semi-automatic or at least heavily supported by CAD tools.
Automating the RTL verification phase, is the next challenge that the CAD industry is facing in providing full support for digital systems development.

The synthesized model needs to be verified. The objective of RTL versus gates verification, or equivalence checking, is to guarantee that no errors have been introduced during the synthesis phase. It is an automatic activity, requiring minimal human interaction, that compares the pre-synthesis RTL description to the post-synthesis gate-level description in order to guarantee the functional equivalence of the two models.

At this point, it is possible to proceed to technology mapping and placement and routing. The result is a description of the circuit in terms of geometrical layout used for the fabrication process. Finally, the design is fabricated, and the microchips are tested and packaged.

1.2.2 Functional verification

As we observed in the previous section, the correctness of a digital circuit is a major consideration in the design of digital systems. Given the extremely high and increasing costs of manufacturing microchips are very expensive. At the same time, function verification, which is verifying that RTL description do its function correctly, is still one the most challenging activities in digital system development: as, it is still carried on mostly with ad-hoc tests, scripts and, often, even ad-hoc tools developed by design and verification teams. In the best scenarios, the development of this verification infrastructure can be amortized among a family of designs with similar architecture and functionality. Moreover, verification methodology still lacks any standard or even a commonly accepted plan of attack, with the consequence that each hardware engineering team has its own distinct verification practices, which often change with subsequent designs by the same team, due to the insufficient "correctness confidence-level" that any of the current approaches provide. Given this scenario, it is not only easy to see why many digital IC development teams report that more than 70% of the design time and engineering resources are spent in verification, but it is clear why verification is, thus, the bottleneck in the time-to-market odyssey for integrated circuit development.

1.3 The verification process

Finding bugs not only the verification engineer goal, The goal of hardware design is to create a device that performs a particular task, such as a DVD player, network router, or radar signal processor, based on a design specification, as a verification engineer your job is to make sure the device can accomplish that task successfully that is; the design is an accurate representation of the specification. Bugs are what you get when there is a discrepancy. The behavior of the device when used outside of its original purpose is not your responsibility although you want to know where those boundaries lie.[6]

What types of bugs are lurking in the design? The easiest ones to detect are at the block level, in modules created by a single person. Did the ALU correctly add two numbers? Did every bus transaction successfully complete? Did all the packets make it through a portion of a network switch? It is almost trivial to write directed tests to find these bugs as they are contained entirely within one block of the design.[6]

After the block level, the next place to look for discrepancies is at boundaries between blocks. Interesting problems arise when two or more designers read the same description yet have different interpretations. For a given protocol, what signals change and when? The first designer builds a bus driver with one view of the specification, while a second builds a receiver with a slightly different view. Your job is to find the disputed areas of logic and maybe even help reconcile these two different views.[6]

Once you have verified that the DUT performs its designated functions correctly, you need to see how it operates when there are errors. Can the design handle a partial transaction, or one with corrupted data or control fields? Just trying to enumerate all the possible problems is difficult, not to mention how the design should recover from them. Error injection and handling can be the most challenging part of verification.[6]

1.3.1 Basic Testbench Functionality

The purpose of a testbench is to determine the correctness of the design under test (DUT). This is accomplished by the following steps which also lead to the basics component of the verification environment:

- Generate stimulus (sequence).
- Apply stimulus to the DUT (Driver).
- Capture the response (Monitor).
- Check for correctness (Scoreboard).
- Measure progress against the overall verification goals (Coverage).

Generating stimulus are most important step where this step generate the inputs which expresses a certain feature test, so verification test can be divided into two types based on the stimulus generation where:

- Direct testing.
- Constrained-random stimulus testing.

1.3.2 Direct testing.

Traditionally, when faced with the task of verifying the correctness of a design, you may have used directed tests. Using this approach, you look at the hardware specification and write a verification plan with a list of tests, each of which concentrated on a set of related features. Armed with this plan, you write stimulus vectors that exercise these features in the DUT. You then simulate the DUT with these vectors and manually review the resulting log files and waveforms to make sure the design does what you expect. Once the test works correctly, you check off the test in the verification plan and move to the next.[6]

This incremental approach makes steady progress, always popular with managers who want to see a project making headway. It also produces almost immediate results, since little infrastructure is needed when you are guiding the creation of every stimulus vector. Given enough time and staffing, directed testing is sufficient to verify many designs. Figure see 1.10 shows how directed tests incrementally cover the features in the verification plan. Each test is targeted at a very specific set of design elements. Given enough time, you can write all the tests need for 100% coverage of the entire verification plan.

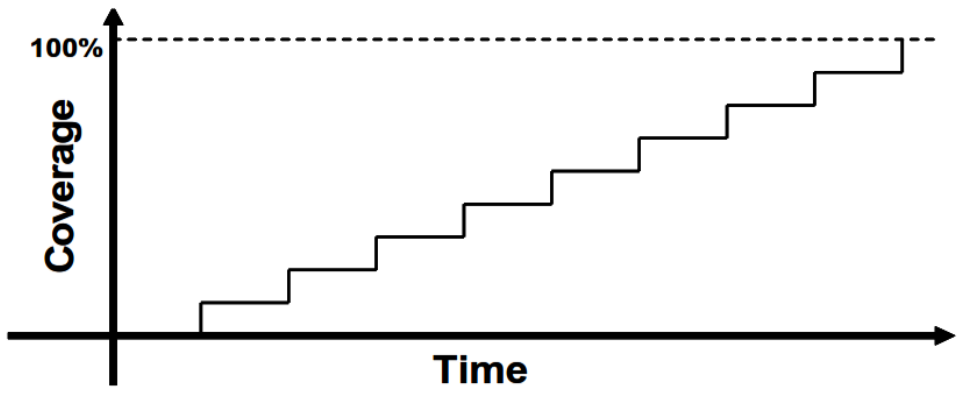


Figure 1.10: directed tests incrementally cover the features in the verification plan

When the design complexity doubles, it takes twice as long to complete or requires twice as many people. Neither of these situations is desirable. You need a methodology that finds bugs faster in order to reach the goal of 100% coverage.

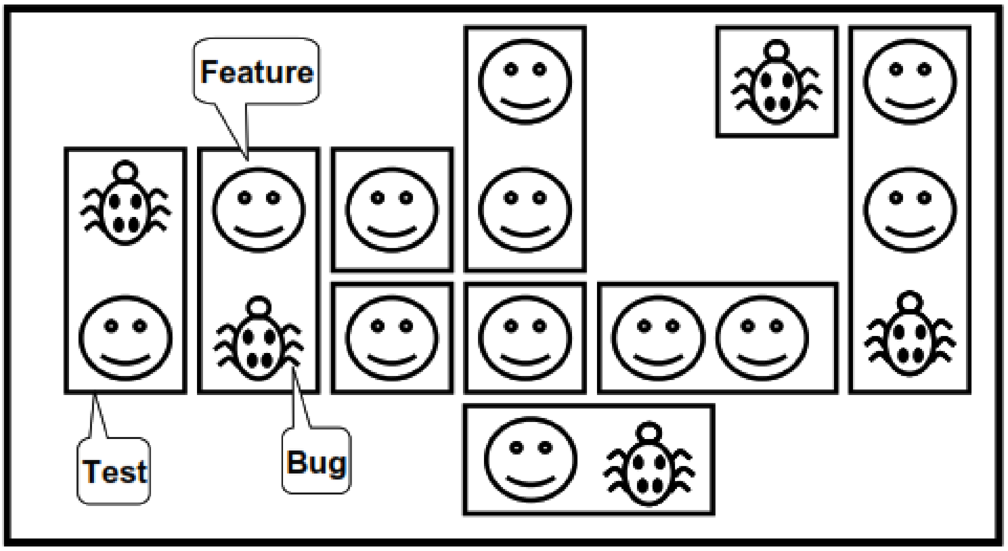


Figure 1.11: total design space and the features that get covered by directed testcases

Figure see 1.11 shows the total design space and the features that get covered by directed testcases. In this space are many features, some of which have bugs. You need to write tests that cover all the features and find the bugs.

1.3.3 Constrained-random stimulus testing

While you want the simulator to generate the stimulus, you don't want totally random values. You use the SystemVerilog language to describe the format of the stimulus ("address is 32-bits, opcode is X, Y, or Z, length ¡ 32 bytes"), and the simulator picks values that meet the constraints. These values are sent into the design, and also into a high-level model that predicts what the result should be. The design's actual output is compared with the predicted output.

When you think of randomizing the stimulus to a design, the first thing that you might think of is the data fields, but there are many field to be randomized such as:

- Device configuration
- Environment configuration
- Input data
- Protocol exceptions
- Delays
- Errors and violation

Random stimulus is crucial for exercising complex designs. A directed test finds the bugs you expect to be in the design, while a random test can find bugs you never anticipated. Once you start using automatically generated stimulus, you need an automated way to predict the results, generally a scoreboard or reference model. Building the testbench infrastructure, including self-prediction, takes a significant amount of work. A layered testbench helps you control the complexity by breaking the problem into manageable pieces. Transactors provide a useful pattern for building these pieces. With appropriate planning, you can build a testbench infrastructure that can be shared by all tests and does not have to be continually modified. You just need to leave "hooks" where the tests can perform certain actions such as shaping the stimulus and injecting disturbances. Conversely, code specific to a single test must be kept separate from the testbench so it does not complicate the infrastructure.

Building this style of testbench takes longer than a traditional directed test- bench, especially the self-checking portions, causing a delay before the first test can be run. This gap can cause a manager to panic, so make this effort part of your schedule. In Figure see 1.12, you can see the initial delay before the first random test runs.

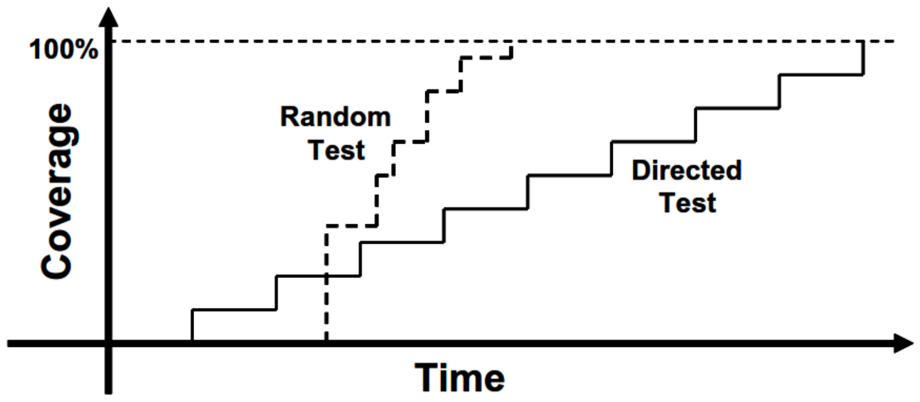


Figure 1.12: How random tests incease coverage rate

While this up-front work may seem daunting, the payback is high. Every test you create shares this common testbench, as opposed to directed tests where each is written from scratch. Each random test contains a few dozen lines of code to constrain the stimulus in a certain direction and cause any desired exceptions, such as creating a protocol violation. The result is that your single constrained-random testbench is now finding bugs faster than the many directed ones.

As the rate of discovery begins to drop off, you can create new random constraints to explore new areas. The last few bugs may only be found with directed tests, but the vast majority of bugs will be found with random tests.

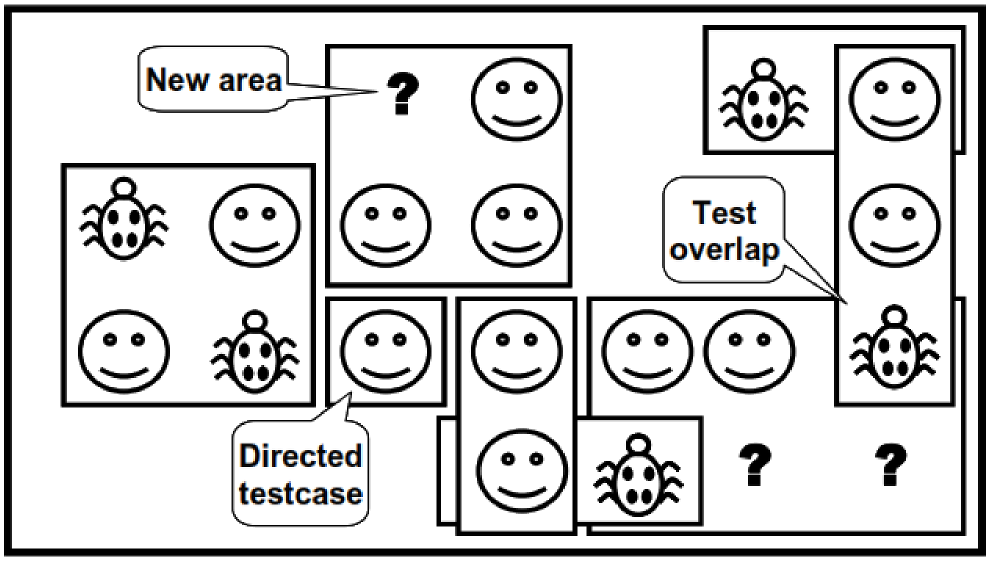


Figure 1.13: constrained-random tests find new bugs

Figure 1.13 shows the coverage for constrained-random tests over the total design space. First, notice that a random test often covers a wider space than a directed one. This extra coverage may overlap other tests, or may explore new areas that you did not anticipate. If these new areas find a bug, you are in luck! If the new area is not legal, you need to write more constraints to keep away. Lastly, you may still have to write a few directed tests to find cases not covered by any other constrained-random tests.

When using random stimulus, you need functional coverage to measure verification progress.

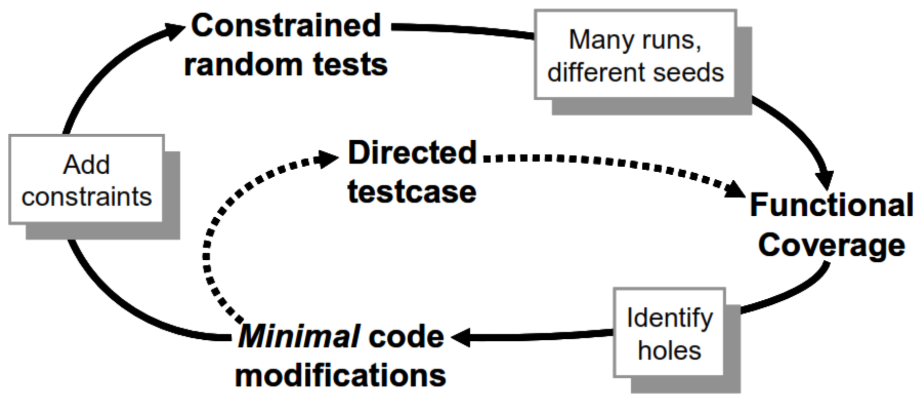


Figure 1.14: the paths to achieve complete coverage

Figure 1.14 shows the paths to achieve complete coverage. Start at the upper left with basic constrained-random tests. Run them with many different seeds. When you look at the functional coverage reports, find the holes, where there are gaps. Now you make minimal code changes, perhaps with new constraints, or injecting errors or delays into the DUT. Spend most of your time in this outer loop, only writing directed tests for the few features that are very unlikely to be reached by random tests.

1.3.4 Coverage.

Coverage associated with the two categories we just described can be combined to form a coverage space, which is often referred to as a coverage model. For instance, an explicit specification coverage space consists of coverage metrics that are manually created by an engineer, derived from a design's requirements document or specification. Another kind of explicit coverage is the instrumentation created by an engineer that is based on the behavior encapsulated by the design implementation, such as the filling or emptying events associated with a particular FIFO in an RTL model. Similarly, an implicit implementation coverage space consists of coverage metrics that are automatically extracted by a tool (such as a simulator), and derived from a design implementation (such as an RTL model). Another part of the implicit specification coverage space consists of coverage metrics that are automatically extracted by a tool, and are derived from the design specification. This part of the coverage space is currently an area of academic research, although there have been a few EDA tools recently emerge that attempt to automatically extract higher-level coverage properties by observing the effects of simulation patterns on an implementation (such as an RTL model). Note that these higher-level functional behaviors cannot be automatically extracted from the implementation alone, which is why they fall into the coverage metrics associated with the implicit specification coverage space.[10]

There are two primary forms of coverage metrics in production use in industry today and these are:

- Code Coverage Metrics (Implicit coverage).
- Functional Coverage/Assertion Coverage Metrics (Explicit coverage).

1.3.4.1 Code Coverage

Introduce various coverage metrics associated with a design model's implicit implementation coverage space. In general, these metrics are referred to as code coverage or structural coverage metrics. There are many types of code coverage such as:

- **Toggle Coverage**
  Toggle coverage is a code coverage metric used to measure the number of times each bit of a register or wire has toggled its value. Although this is a relatively basic metric, many projects have a testing requirement that all ports and registers, at a minimum, must have experienced a zero-to-one and one-to-zero transition. In general, reviewing a toggle coverage analysis report can be overwhelming and of little value if not carefully focused. For example, toggle coverage is often used for basic connectivity checks between IP blocks. In addition, it can be useful to know that many control structures, such as a one-hot select bus, have been fully exercised.[10]

- **Line Coverage**
  Line coverage is a code coverage metric we use to identify which lines of our source code have been executed during simulation. A line coverage metric report will have a count associated with each line of source code indicating the total number of times the line has executed. The line execution count value is not only useful for identifying lines of source code that have never executed, but also useful when the engineer feels that a minimum line execution threshold is required to achieve sufficient testing. Line coverage analysis will often reveal that a rare condition required to activate a line of code has not occurred due to missing input stimulus. Alternatively, line coverage analysis might reveal that the data and control flow of the source code prevented it either due to a bug in the code, or dead code that is not currently needed under certain IP configurations. For unused or dead code, you might choose to exclude or filter this code during the coverage recording and reporting steps, which allows you to focus only on the relevant code.

- **Statement Coverage**
  Statement coverage is a code coverage metric we use to identify which statements within our source code have been executed during simulation. In general, most engineers find that statement coverage analysis is more useful than line coverage since a statement often spans multiple lines of source code-or multiple statements can occur on a single line of source code. A metrics report used for statement coverage analysis will have a count associated with each line of source code indicating the total number of times the statement has executed. This statement execution count value is not only useful for identifying lines of source code that have never executed, but also useful when the engineer feels that a minimum statement execution threshold is required to achieve sufficient testing.

- **Block Coverage**
  Block coverage is a variant on the statement coverage metric which identifies whether a block of code has been executed or not. A block is defined as a set of statements between conditional statements or within a procedural definition, the key point being that if the block is reached,

all the lines within the block will be executed. This metric is used to avoid unscrupulous engineers from achieving a higher statement coverage by simply adding more statements to their code.

- **Branch Coverage**
  Branch coverage (also referred to as decision coverage) is a code coverage metric that reports whether Boolean expressions tested in control structures (such as the if, case, while, repeat, forever, for and loop statements) evaluated to both true and false. The entire Boolean expression is considered one true-or-false predicate regardless of whether it contains logical-and or logical-or operators.

- **Expression Coverage**
  Expression coverage (sometimes referred to as condition coverage) is a code coverage metric used to determine if each condition evaluated both to true and false. A condition is an Boolean operand that does not contain logical operators. Hence, expression coverage measures the Boolean conditions independently of each other

- **Finite-State Machine**
  Coverage Today's code coverage tools are able to identify finite state machines within the RTL source code. Hence, this makes it possible to automatically extract FSM code coverage metrics to measure conditions. For example, the number of times each state of the state machine was entered, the number of times the FSM transitioned from one state to each of its neighboring states, and even sequential arc coverage to identify state visitation transitions.[10]

1.3.5 Functional Coverage

The objective of functional verification is to determine if the design requirements, as defined in our specification, are functioning as intended. But how do you know if all the specified functionality was actually implemented? Furthermore, how do we know if all the specified functionality was really tested? Code coverage metrics will not help us

answer these questions.
In this section, we introduce an explicit coverage metric referred to as functional coverage, which can be associated with either the design's specification or implementation coverage space. The objective of measuring functional coverage is to measure verification progress with respect to the functional requirements of the design. That is, functional coverage helps us answer the question: Have all specified functional requirements been implemented, and then exercised during simulation? The details on how to create a functional coverage model are discussed separately in the Testplan to functional coverage chapter.
The functional behavior of any design, at least as observed from any interface within the verification environment, consists of both data and temporal components. Hence, from a high-level, there are two main types of functional coverage measurement we need to consider: Cover Groups' and Cover Properties.

Cover Group Modeling
With respect to functional coverage, the sampling of state values within a design model or on an interface is probably the easiest to understand. We refer to this form of functional coverage as cover group modeling. It consists of state values observed on buses, grouping of interface control signals, as well as register. The point is that the values that are being measured occur at a single explicitly or implicitly sampled point in time. SystemVerilog covergroups are part of the machinery we typically use to build the functional data coverage models, and the details are discussed in the block level design example and the discussion of the corresponding example covergroup implementations.

Cover Property Modeling
With respect to functional coverage, temporal relationships between sequences of events are probably the hardest to reason about. However, ensuring that these sequences of events are properly tested is important. We use cover property modeling to measure temporal relationships between sequences of events. Probably the most popular example of cover properties involves the handshaking sequence between control signals on a bus protocol. Other examples include power-state transition coverage associated with verifying a low-power design. Assertions and coverage properties are part of the machinery that we use to build

temporal coverage models, and are addressed in the bus protocol monitor example.

1.3.6 Assertion

. Assertions are primarily used to validate the behavior of a design. An assertion is a check embedded in design or bound to a design unit during the simulation. Warnings or errors are generated on the failure of a specific condition or sequence of events. Assertions are used to
- Check the occurrence of a specific condition or sequence of events.
- Provide functional coverage. There are two kinds of assertions:
- Immediate Assertions.
- Concurrent Assertions.

- **Immediate Assertions:**

Immediate assertions check for a condition at the current simulation time. An immediate assertion is the same as an if..else statement with assertion control. Immediate assertions have to be placed in a procedural block definition.

- **Concurrent Assertions:**

Concurrent assertions check the sequence of events spread over multiple clock cycles. The concurrent assertion is evaluated only at the occurrence of a clock tick - The test expression is evaluated at clock edges based on the sampled values of the variables involved - It can be placed in a procedural block, a module, an interface or a program definition

Chapter 2

UVM overview

2.1 What is UVM?

UVM is short for Universal Verification Methodology. The accellera UVM standard was created by EDA vendors like Cadence ,Mentor and Synopsys, and customers and design companies like ARM ,IBM , . . . etc. The UVM implementation is based on system Verilog base classes , UVM is a situation of verification knowledge and experience, this combination provides a powerful flexible methodology to create reusable, scalable and interoperable testbenches that help to simulate and verify the wide variety of IC designs.

2.1.1 System Verilog in UVM

UVM is built with SystemVerilog's Object Oriented Programming constructs based on aggregation or composition and inheritance concepts. Aggregation or composition means that a class has a reference to another class, in other words an object container relationship and inheritance concept describes the relationship between base classes and extended ones you can say it shows the UVM hierarchy.

2.2 UVM Methodology

The verification methodology has many goals, the most important goal is reusability which means a configurable test environments for a variety of tests with reusable testbench components from project to project . Reusability saves a lot of time and effort putting the parts of the test together, connecting different components inside the testbench and creating stimulus. This allows the engineer to create stimulus and components once then reuse that work again in the same project or in other different projects. The second goal is interoperability needed for tools from multiple vendors and with multiple types of tools as well. For example an engineer develops a testbench with one simulator and wants to make sure that your code behaves correctly in other simulators as the system Verilog is very large and most of time many vendors contributes to implement the features. UVM allows each vendor to focus on a common subset of system verilog feature so that a design simulates consistently, UVM also allows verification intellectual property (VIP) models in your testbench that's why In-house verification code of components that make up the design and commercial verification code for of the shelf components can be used.There is no need to write the code from scratch, moreover the UVM separates stimulus generation from delivering it to the DUT, In UVM, classes that describe the transaction are different from the classes that describe how components are connected together that allows several engineers to generate stimulus and develop the testbench in parallel, UVM code also is written in a maintainable manner so that it is easy to read or modify according to your needs in the project.

2.3 UVM Topology

Using UVM we are trying to build a verification environment that can be used over and over for many test, the main idea is to separate the stimulus from the test bench ,So that the stimulus will be responsible for defining what exactly will happen for this particular simulation run , while the testbench will be responsible for defining all components that are needed to interact with DUT. The test class is to build and configure the environment and to generate stimulus we can also determine how many times are we going to run a particular stimulus and what type of transactions are we going to generate. The environment class instantiate the components for driving transactions into the DUT, Monitoring values read fro the DUT and checking the result as well.see2.1 The DUT communicates with the testbench through a systemverilog interface that has different methods which you can call to drive transactions to the design and read them out of it , So this structure rarely changes. All of these of these components are constructed under a base class called uvm_root which constructs your testbench and start the simulation phases.

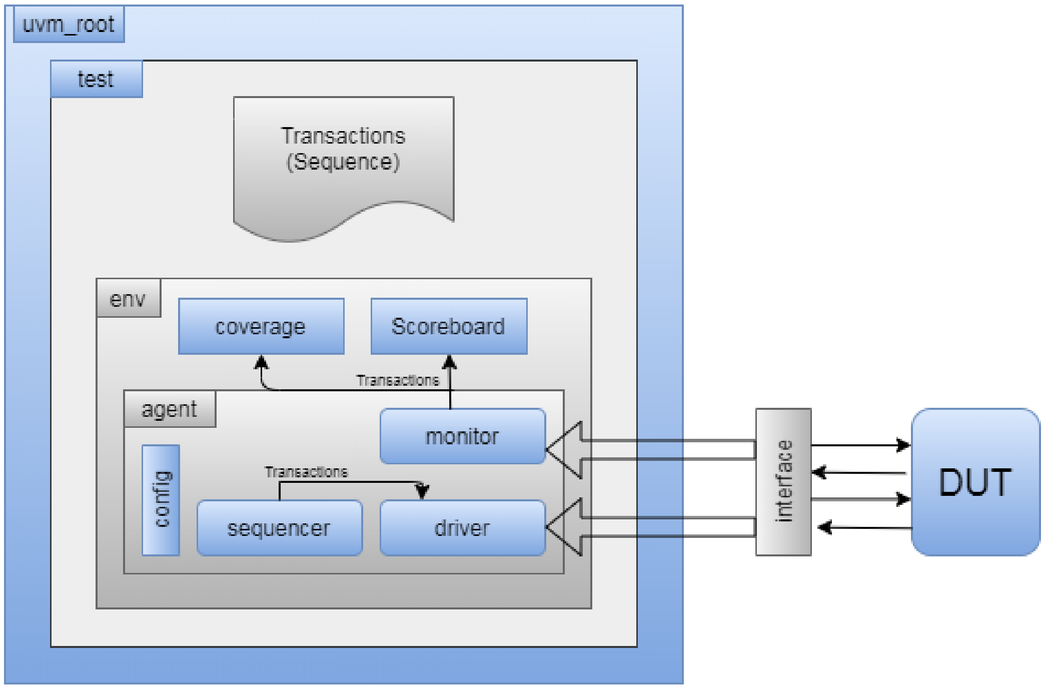


Figure 2.1: UVM classes , connections between testbench and DUT

The DUT communicates with the test environment through a systemverilog interfacre , So every interface inside your design needs an agent that encapsulates everything needed for counicating with this interface , first drive transactions into DUT so the driver send transactions to the interface which then wiggles to DUT pins. A sequencer components connected to the driver sends the transactions to the driver then the driver sends the transactions to the DUT through the interface, To verify the result the monitor watches the pin wiggles through the interface coverts those pin wiggles into transactions and sends them to the scoreboard or coverage collector for checking the values and verify results , This is done an analysis port which is like a sysytemverilog mailbox. A configuration object is a class with configuration values like the virtual interface.

2.3.1 UVM factory

Previously, creating a dynamically different type of objects required modifying source code which contradicts the reusability concept of UVM. UVM factory is a mechanism introduced by the UVM to improve the flexibility and scalability of the testbench by allowing the user to substitute an existing class object by any of its inherited child class objects. Factory is a critical aspect which is introduced in the UVM that builds everything in the UVM environment like dynamically adaptable testbenches, which are tests created and compiled at run time . Therefore, factory requires that all the classes to be registered with the factory with macros like 'uvm_component_utils and 'uvm_object_utils macros.

2.4 Class Hierarchy

The UVM package contains a class library that provides a set of base classes which can be extended by users as required , UVM object is the base class for all UVM data and hierarchical classes . its role is to define a set of methods for common operation such as (create , copy , compare, print). there are two group class that are inherited from uvm_object the first one is uvm_component that is used to build the testbench topology also there classes are dynamically created they exist for the entire simulation , the classes has additional characteristics like being in fixed location inside uvm topology and having methods that are called in a fixed order to build and connect the testbench run the test and report the results , The second droup is transaction classes the stimulus is described into the design by extending uvm base classes a single transaction is a sequence item and multiple items form a sequence . Transactions are transient object they are created and destroyed during the simulation run and has no fixed location in the topology and created in the test ,flow into the driver or are created in the monitor and are sent into the scoreboard The complete diagram expansion in 2.2 may show other predefined component types that are driver from uvm_component class , The class should be used as the base class for any user definend components so to create my_agent class you should etend uvm_agent base class and te same for driver, monitor . . . etc. The uvm class library provides all the building blocks you need to develop and easily constructed.

2.4.1 UVM TLM communication

The communication between components has 2 connections called a TLM that stands for transaction level modeling , The TLM connections has 2 pins the initiator that has an object called port and the target contains an object called export, As in the driver when it pulls transactions from sequencer , A port is a one to one connection to an export a less common type is when producer pushes transactions to a consumer. Another kind of connections called Analysis port export that is used to connect monitor to both scoreboard and coverage collectors so it 's a one to many connection.see 2.2

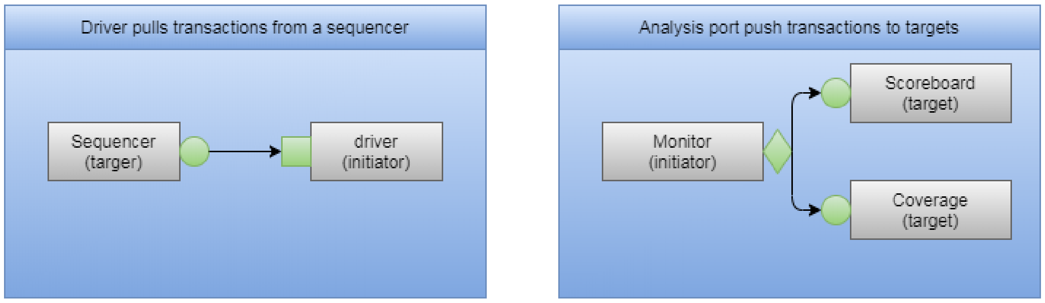


Figure 2.2: Transaction level modeling VS Analysis port export

2.4.2 DUT connections to testbench

The DUT's ports can not be connected directly to the testbench class objects so a different SystemVerilog means of communication, which is virtual interfaces is used.see 2.3 The DUT's ports are connected to an instance of an interface. The Testbench communicates with the DUT through the interface instance. Using a virtual interface as a reference or handle to the interface instance, the testbench can access the tasks, functions, ports, and internal variables of the SystemVerilog interface. As the interface instance is connected to the DUT pins, the testbench can monitor and control the DUT pins indirectly through the interface elements.

2.4.3 How are UVM classes related?

The UVM library defines a set of base classes and utilities that facilitate the design of modular, scalable, reusable verification environments. All UVM

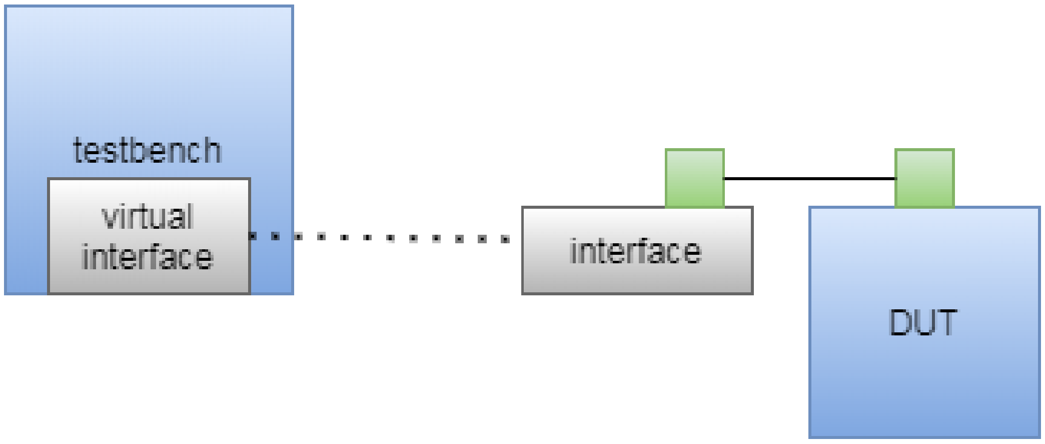


Figure 2.3: DUT connected to test-bench through irtual interface

classes are derived from uvm_object and an individual transaction is a sequence item contains a transactions properties and methods , A series of generated sequence items are known as sequence to obtain a real stimulus. The transactions are sent to the sequencer that routes between multiple sequences, Sequencer and drivers are components which are permanent objects created at the start of simulation and remain for the entire simulation. UVM sequence class is derives from uvm_sequence base class that contains a task body to generate one or more sequences , There are 4 steps should be done to generate transactions first one is to creat a sequence item object ,then wait for a driver to request a transaction through sequencer, The third step is to assign the transaction values where you have to check your randomization and Lastly send the transaction to driver and wait for completion. Sequences can have randomized properties by allocating them as random variables then the sequences can behave differently each time when it is started, Complex sequences may get a feedback fro the dut to choose between branches to complete the sequence.

2.5 UVM phases

It is a concept of how the UVM act from the start of simulation to the end in which all the testbench components goes through these set of phases sequentially, so it is a synchronizing mechanism for the environment in the life cycle of a simulation.

2.5.0.1 Why does UVM need phases?

Because UVM uses system Verilog OOP which enables reusing and editing classes and objects which can be created at different times, so it is possible to create a new object in the middle of the simulation , which could end by calling a component while it hasn't been initialized yet leading to wrong testbench outputs

2.5.0.2 Why Verilog testbenches don't need phases?

Because it consists of static modules which have a set of ports to communicate with other test bench components Static modules means have their instances created at the beginning of the simulation, so there are no worries about any component being called without it being created

2.5.1 Phases hierarchy

UVM -phases can be grouped into three categories see 2.4

1. Build time phases

Phases executed in the start of simulation in which the testbench components are constructed ,configured and connected in zero time simulation since this phase methods are functions executed in zero time simulation .

Build phase Is done from the top to the bottom .

They consist of :

- Build_phase: function used to build test bench components and create their instance
- Connect_phase: function used to connect between different testbench components via TLM ports.
- End_of_elaboration_phase: function used to display UVM topology and other functions required to be done after connection
- Start_of_simulation_phase: function used to set initial run-time configuration and display topology.

2. Run time phases

Actual simulation that consumes time happens in this UVM phase and runs

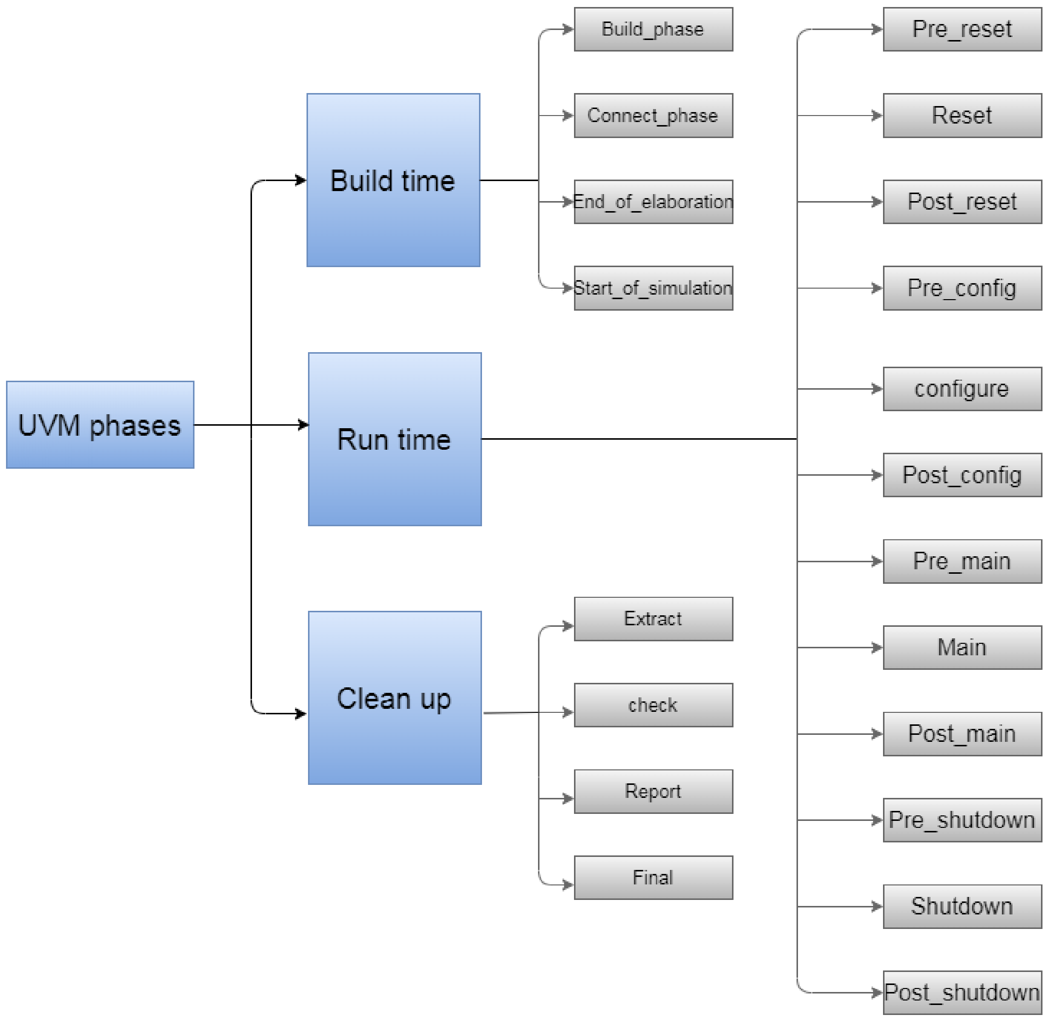


Figure 2.4: DUT connected to test-bench through virtual interface

parallel to other UVM run-time phases. Consists of :

- Pre-reset : the pre_reset phase starts at the same time as the run phase. Its purpose is to take care of any activity that should occur before the reset, such as waiting for a power-good signal to go active.
- Reset: responsible for DUT reset.
- Post_reset: responsible for any required actions after reset.
- Pre_configure: This phase is intended for anything that is required to prepare for the DUT configuration process after the DUT is out of reset.
- Configure: configure phase is used to put the DUT into a known state before the stimulus could be applied to the DUT.
- Post_configure: This phase is intended to wait for the effect of the configuration to propagate through the DUT.
- Pre_main : pre_main is used to ensure that all the components needed to generate the stimulus are ready to do so.
- Main: main phase is where the stimulus specified by the Test case is generated and applied to the DUT.
- Post_main: Used for any final act after the main phase.
- Pre_shutdown: This phase is acts like a buffer to apply any stimulus before the shutdown phase starts.
- Shutdown: The shutdown phase is to ensure that the effects of stimulus generated during the main phase has propagated through the DUT and that the resultant data has drained away.
- Post_shutdown: post_shutdown is intended for any final activity before exiting the run phase. After it UVM Testbench starts the cleanup phase.

3. Clean-Up phases

It is the phase where the results of the testcase are collected and reported. Consists of:

beginitemize

- Extract : Used to retrieve and process information from scoreboards and functional coverage monitors.

- Check: Used to check that the DUT behaved correctly and to identify any errors that may have occurred during the execution of the test bench.

- Report: Used to display the results of the simulation or to write the results to file.

- Final: Used to complete any other outstanding actions that the test bench has not already completed.

2.6 UVM environment approaches

2.6.0.1 UVM Transactions approach

We've modularized our work by providing methods in each class that do the work of that class and by being careful to avoid situations where one class needs to know the internal workings of another . to maintain adaptability and reusability for passing data between classes (tester , driver , coverage and scoreboard) we exploit the advantages of classes , methods and OOP as the class and the objects we instantiate from that class have two useful points : classes have methods that interact with the data and hide details from users. We work with objects through handles, and we can pass these handles around our test- bench. Therefore several object scan easily share a piece of data. this is implemented these advantages using UVM class library transactions. Encapsulating all this data in the transaction makes the rest of the test-bench much simpler. For example, tester won't need to figure out legal values to drive the test-bench. It will simply let the transaction randomize itself.

transactions classes definition We define transactions by extending the uvm_transaction base class and writing the methods (convert2string , do_copy , do_compare) .Transactions encapsulate both data and all the operations we can do to that data. Data fields can be randomized using System Verilog's built-in randomize method. The uvm_transaction class extends uvm_object, not uvm_component, there is a result_transaction class to hold results and another transaction class that extends command_transaction to generate different stimulus without changing the tester object this class will use same way as the command_transaction but under dedicated constrains. The result_transaction class is just like the command_transaction class , The scoreboard will use the do_compare() method to compare predicted results to actual results. Transaction-level simulation makes it easier to compare predicted and actual results Both the result monitor and the predictor create result_transaction objects. The result_monitor passes us an actual result Then we get the corresponding command from the command_monitor and use the predict_result() method to create a predicted result_transaction. We use compare() to see if we got the right result. The scoreboard is now much simpler. we override the command_transaction with the class that generates stimulus , The override causes the tester to create a constrained transaction from the class mentioned rather than a command_transaction, without modifying the tester.

This transaction approach focuses on data. Classes and objects are created to easily create, compare, and transport data. Although in this approach the data classes are separated from the structure classes, the data stimulus is not separated from structure.

Good test benches separate the order of the transactions (the test stimulus) from the test bench structure. The structure should remain unchanged regardless of the order of transactions So another approach called sequence approach is used in 3 blocks (rate-matcher , FFT , IFFT) environments and also used in the Top level environment.see 2.5

2.6.0.2 UVM Sequences approach

UVM sequences separate stimulus from the test-bench structure. They allow us to create one test-bench structure and then run different data through it , thus completing our journey through the UVM. A sequence item is created to carry out the data and the uvm_sequencer can replace the tester in the transctions class. The uvm_sequence_item carries data from uvm_sequences through the uvm_sequencer to a uvm_driver. The sequence_item class is exactly the same as the command_transaction class except that the class extends uvm_sequence_item instead of uvm_transaction. The sequencer class takes sequence_items from a sequence and passes them on to a driver. The UVM provides us with a uvm_sequencer base class. The driver class extend uvm_driver and parameterize it to work with the sequence_item and inherit the seq_item_port object and all its functionality. The run phase calls the get_next_item() method on the seq_item_port object, This method blocks until the sequencer puts data into the port and then gives us a sequence_item object. calling the item_done() method on the seq_item_port object to tell the sequencer that it can send us another sequence item. In this approach environment the sequencer doesn't need a FIFO to connect to the driver. the driver expects to connect to a sequencer and can do it directly.

The uvm_sequencer comes equipped with an object called a seq_item_export see 2.6 connect the driver to the sequencer by calling the connect() method on the driver' s seq_item_port. The uvm_sequence class sits outside the UVM hierarchy (it has no parent argument in its constructor) but can feed data into the UVM hierarchy through the uvm_sequencer.

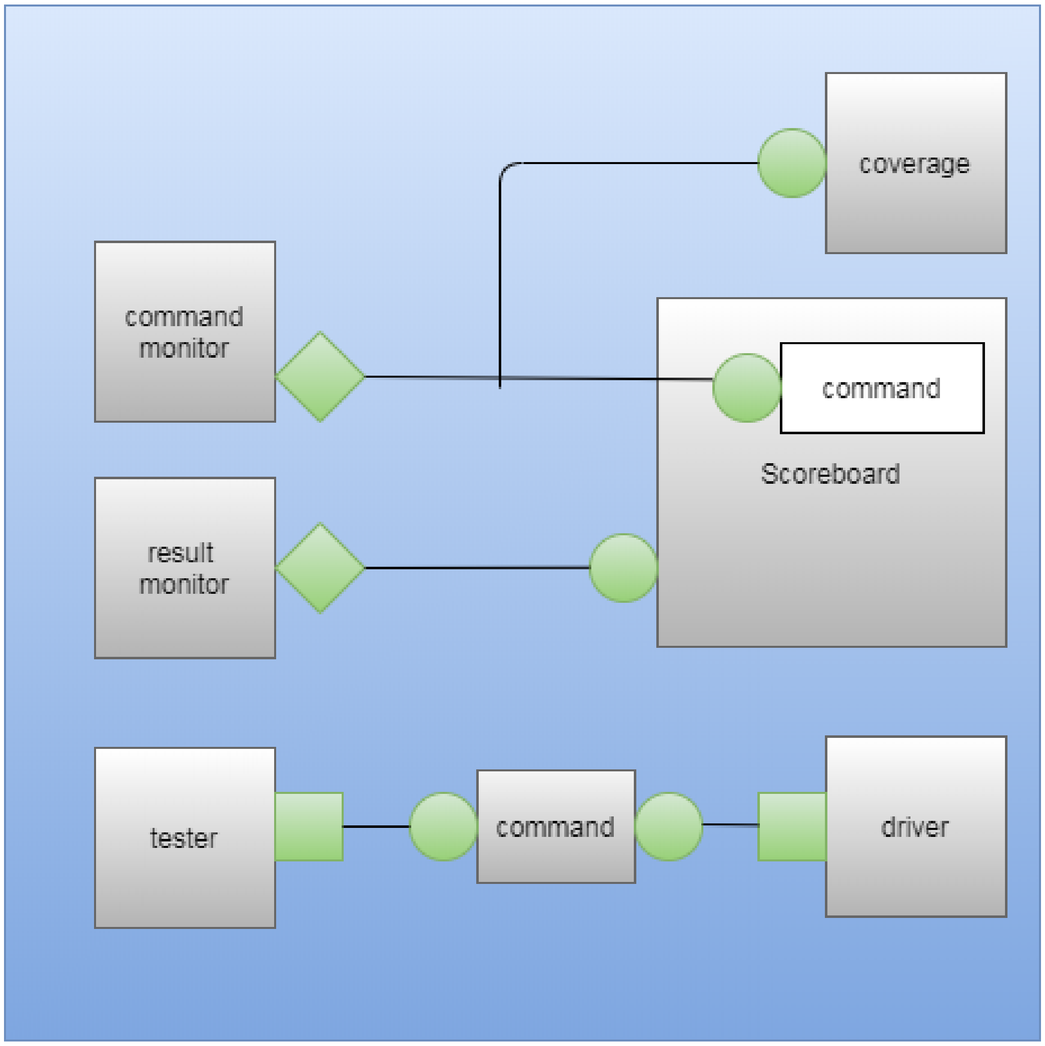


Figure 2.5: UVM Tansactions approach envirnoment connectionss

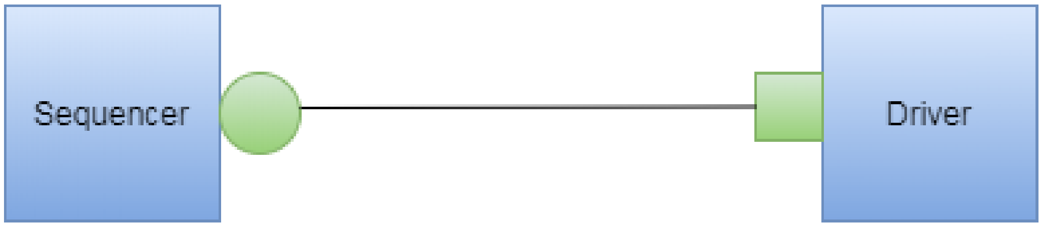


Figure 2.6: Sequencer connected directly to driver through seq_item_export.

Chapter 3

Standard specifications and functionality

We have subjected to 3GPP Narrowband IOT LTE Standard release 14. In this Chapter, the standard description for each block and its algorithm in the chain will be stated.[1]

3.1 Physical uplink shared channel

The baseband signal representing the physical uplink shared channel is defined in terms of the following steps: see 3.1

- scrambling.
- Modulation of scrambled bits to generate complex-valued symbols.
- Mapping of the complex-valued modulation symbols onto one or several transmission layers.
- Transform precoding to generate complex-valued symbols.
- Precoding of the complex-valued symbols.
- Mapping of precoded complex-valued symbols to resource elements.
- Generation of complex-valued time-domain SC-FDMA signal for each antenna port.

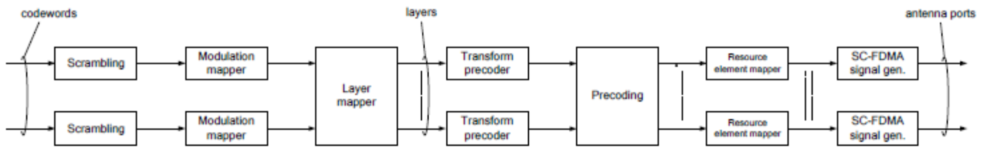


Figure 3.1: Overview of uplink physical channel processing [1]

3.2 Channel coding, multiplexing and interleaving

3.2.1 Cyclic Redundancy Check (CRC)

Denote the input bits to the CRC computation by a_0,a_1,.....,a_{A-1} , and the parity bits by p_0,p_1,.....,p_{l-1} is equal to TBS and L is the number of parity bits at which L=24 bits for NB-IOT. The parity bits for NB-IOT are generated the following cyclic generator polynomial:
$g_{CRC24A} = [D^{24} + D^{23} + D^{18} + D^{17} + D^{14} + D^{11} + D^10 + D^7 + D^6 + D^5 + D^4 + D^3 + D + 1]$
The bits after CRC attachment are denoted by b_0,b_1,.....,$b_{A+l}-_1$ where B = A+ L. The relation between a_k and b_k is :

$$b_k = a_k \quad \text{For k = 0, 1, 2,. . . , A-1}$$

$$b_k = a_{k-A} \quad \text{For k = A, A+1, A+2,. . . , A+l-1}$$

There is no segmentation as for NB-IOT maximum TBS=2536 bits it's less than Z=6144 bits so there is only one code block C=1.[3]

3.2.2 Turbo Encoder

According to 3GPP narrowband IOT LTE Standard release 14, Turbo Encoder consists of two recursive Convolutional Encoder and one Turbo code internal interleaver as shown in see 3.2

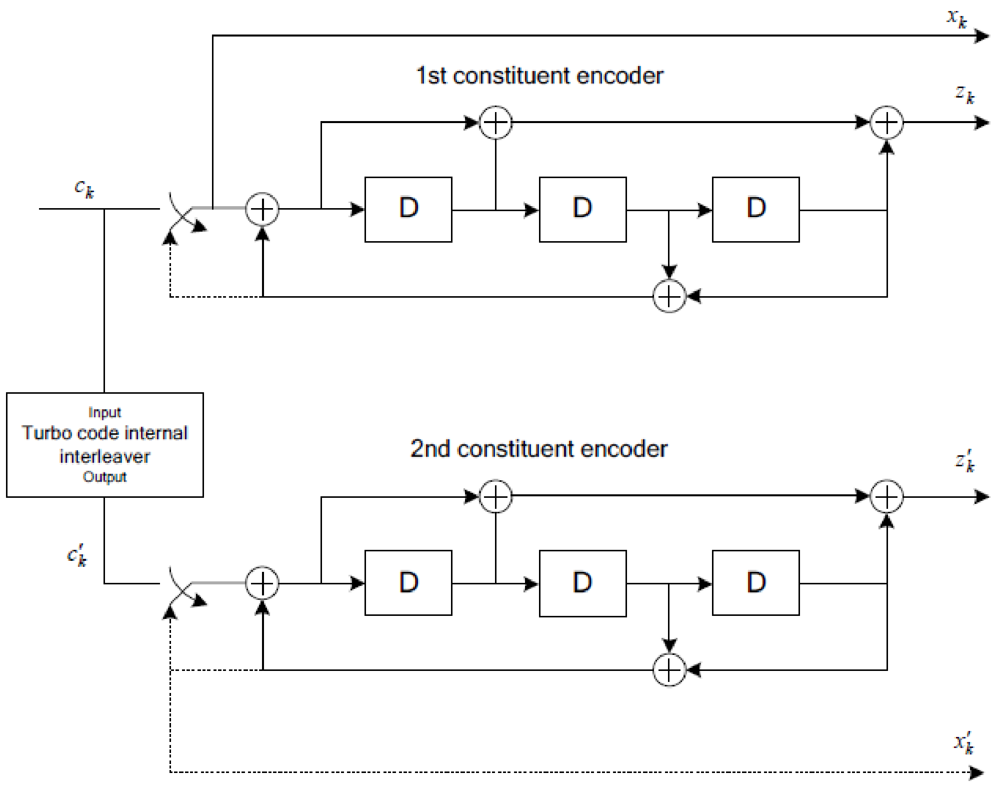


Figure 3.2: Internal Structure of NB ILTE Turbo Encoder of rate 1/3

3.2.2.1 Recursive Convolutional Encoder

The scheme of recursive Convolutional encoder is a Parallel Concatenated Convolutional Code (PCCC) with two 8-state constituent encoders see 2.3, the coding rate of turbo encoder is 1/3, the recursive convolutional encoders are differing from ordinary convolutional encoder as it has both feedback and feedforward convolutional encoders.

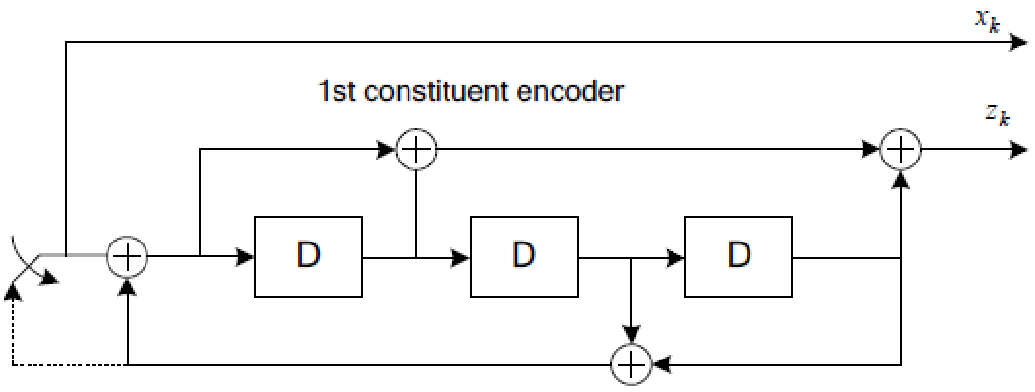


Figure 3.3: Recursive Convolutional Encoder

The transfer function of the 8-state constituent code for the PCCC is:

$$G(D) = [1, \frac{g1(D)}{g0(D)}]$$
$$\text{where } g0(D) = 1 + D^2 + D^3$$
$$g0(D) = 1 + D + D^3$$

The initial value of the shift registers of the 8-state constituent encoders shall be all zeros when starting to encode the input bits.

The output from the turbo encoder is

$$d_k^{(0)} = X_k$$
$$,d_k^{(1)} = z_k$$
$$,d_k^{(2)} = z_k^{'}$$
For k = 0,1,2,...,K-1.

The bits input to the turbo encoder are denoted by c_0,c_1,c_2,c_{k-1},and the bits output from the first and second 8-state constituent encoders are denoted by z_0,cz_1,z_2,z_{k-1}, and $z_0^{'}$,$z_1^{'}$,....,$z_{k-1}^{'}$,respectively. The bits output from the turbo code internal interleaver are denoted by $c_0^{'}$,$c_1^{'}$,$c_2^{'}$,$c_{k-1}^{'}$, and these bits are to be the input to the second 8-state constituent encoder.[3]

3.2.2.2 Trellis termination for turbo encoder

Trellis termination is performed by taking the tail bits from the shift register feedback after all information bits are encoded. Tail bits are padded after the encoding of information bits. The first three tail bits shall be used to terminate the first constituent encoder (upper switch of figure in lower position) while the second constituent encoder is disabled. The last three tail bits shall be used to terminate the second constituent encoder see3.3 while the first constituent encoder is disabled. The transmitted bits for trellis termination shall then be:

$d_k^{(0)} = x_k$, $d_{k+1}^{(0)} = z_{k+1}$, $d_{k+2}^{(0)} = x_k^{'}$,$d_{k+3}^{(0)} = z_{k+1}^{'}$

$d_k^{(1)} = z_k$, $d_{k+1}^{(1)} = x_{k+2}$, $d_{k+2}^{(1)} = z_k^{'}$,$d_{k+3}^{(1)} = x_{k+2}^{'}$

$d_k^{(2)} = X_{k+1}$,$d_{k+1}^{(2)} = z_{k+2}$,$d_{k+2}^{(2)} = x_{k+1}^{'}$,$d_{k+3}^{(2)} = z_{k+2}^{'}$

3.2.2.3 Internal Interleaver

The bits input to the turbo code internal interleaver are denoted by c_0,c_1,c_2,.....,c_{k-1}, where K is the number of input bits = TBS + 24 CRC bits. The bits output from the turbo code internal interleaver are denoted by $c_0^{'}$,$c_1^{'}$,$c_2^{'}$,.....,$c_{k-1}^{'}$, The relationship between the input and output bits is as follows:

$$c_i^{'}{=}c_{\pi(i)} \text{ ,Where i=0,1,2,... (k-1)}$$

Where the relationship between the output index i (ordered bits index) and the input index $\pi(i)$-interleaved index- satisfies the following quadratic form: $\pi(i) = (f_1 * i + f_2 * i^2)\%K$ The parameters f_1 and f_2 depend on the block size K and are summarized in Table in the standard.[3]

M	K	f_1	f_2	M	K	f_1	f_2	M	K	f_1	f_2	M	K	f_1	f_2
1	40	3	10	34	304	37	76	67	624	41	234	100	1280	199	240
2	48	7	12	35	312	19	78	68	640	39	80	101	1312	21	82
3	56	19	42	36	320	21	120	69	656	185	82	102	1344	211	252
4	64	7	16	37	328	21	82	70	672	43	252	103	1763	21	86
5	72	7	18	38	336	115	84	71	688	21	86	104	1408	43	88
6	80	11	20	39	344	193	86	72	704	155	44	105	1440	149	60
7	88	5	22	40	352	21	44	73	720	79	120	106	1472	45	92
8	96	11	24	41	360	133	90	74	736	139	92	107	1501	49	846
9	104	7	26	42	368	81	46	75	752	23	94	108	1536	71	48
10	112	41	84	43	376	45	94	76	768	217	48	109	1568	13	28
11	120	103	90	44	384	23	48	77	784	25	98	110	1600	17	80
12	128	15	32	45	392	243	98	78	800	17	80	111	1632	25	102
13	136	9	34	46	400	151	40	79	816	127	102	112	1664	183	104
14	144	17	108	47	408	155	120	80	832	25	52	113	1696	55	954
15	152	9	38	48	416	25	52	81	848	239	106	114	1728	127	96
16	160	21	120	49	424	51	106	82	864	17	48	115	1760	27	110
17	168	101	84	50	432	47	72	83	880	137	110	116	1792	29	112
18	176	21	44	51	440	91	110	84	896	215	112	117	1824	29	114
19	184	57	46	52	448	29	168	85	912	29	114	118	1856	57	116
20	192	23	48	53	456	29	114	86	928	15	58	119	1888	45	354
21	200	13	50	54	464	247	58	87	944	147	118	120	1920	31	120
22	208	27	52	55	472	29	118	88	960	29	60	121	1952	59	610
23	216	11	36	56	480	89	180	89	976	59	122	122	1984	185	124
24	224	27	56	57	488	91	122	90	992	65	124	123	2016	113	420
25	232	85	58	58	496	157	62	91	1008	55	84	124	2048	31	64
26	240	29	60	59	504	55	84	92	1024	31	64	125	2112	17	66
27	248	33	62	60	512	31	64	93	1056	17	66	126	2176	171	136
28	256	15	32	61	528	17	66	94	1088	171	204	127	2240	209	420
29	264	17	198	62	544	35	68	95	1120	67	140	128	2304	253	216
30	272	33	68	63	560	227	420	96	1152	35	72	129	2368	367	444
31	280	103	210	64	576	65	96	97	1152	35	72	130	2432	265	456
32	288	19	36	65	592	19	74	98	1216	39	76	131	2496	181	468
33	296	19	74	66	608	37	76	99	1248	19	78	132	2560	39	80

Table 3.1: Turbo code internal interleaver parameters

3.2.3 Rate Matching

The basic function of rate matching module is to match the number of bits in transport block to the number of bits that can be transmitted in the given allocation, it also controls the rate as the turbo encoder gives 1/3 rate, we can increase or decrease rate by using this block according to DCI and channel quality information. By means of rate matching, any arbitrary code rate can be achieved from a fixed-rate mother code.[3] Rate matching mainly consists of three main parts:

- First part is sub-block interleavers which are used to interleave the three information bit streams $d_k^{(0)}$, $d_k^{(1)}$ and $d_k^{(2)}$ coming from turbo-encoder as shown see 3.4.
- Second part is bit collection block which concatenates the three output streams of the three sub-block interleavers $v_k^{(0)}$, $v_k^{(1)}$,$v_k^{(2)}$ which represent the systematic bit stream, parity bit stream and interleaved parity stream respectively as in see 3.4.
- Last sub-block is bit-selection block which select the start point inside buffer to get output according to upper layer parameters described in following sections.

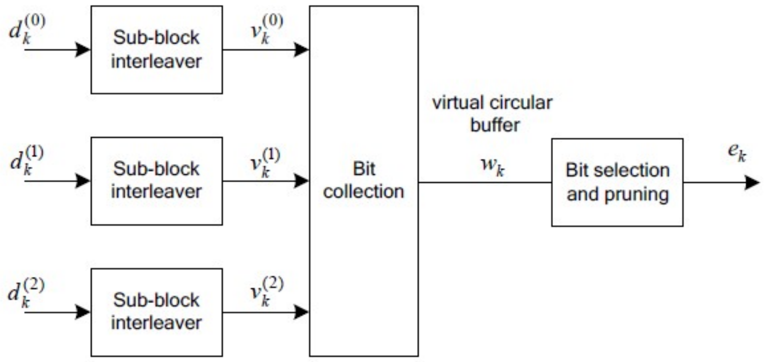


Figure 3.4: Standard block diagram for rate matching

3.2.3.1 Sub-block interleavers

The input bits to the sub-block interleaver are denoted by $d_0^{(i)}$, $d_1^{(i)}$,$d_2^{(i)}$,.....,$d_{D-1}^{(i)}$, where D is the number of bits D=TBS+24+4 where the 24 bits are CRC bits and the other 4 bits are the trellis termination bits added by the encoder and i=0,1,2. The interleaving is done after putting the data in matrix form according to the following steps:

- Assign $C_{subblock}^{TC} = 32$ to be the number of columns of the matrix. The columns of the matrix are numbered 0, 1, 2,. . . , $C_{subblock}^{TC} - 1$ from left to right.

- Determine the number of rows of the matrix $R_{subblock}^{TC}$ at which the rows of rectangular matrix are numbered 0, 1, 2,. . . , $R_{subblock}^{TC} - 1$ from top to bottom. Number of rows can be found by calculating minimum integer $R_{subblock}^{TC}$ such that:
$D \leq (R_{subblock}^{TC} C_{subblock}^{TC})$

- The next step is to determine the number of dummy bits to complete the rectangular matrix according to next scenario :

-If $(R_{subblock}^{TC} C_{subblock}^{TC})$¿D, then N_D= ($R_{subblock}^{TC} C_{subblock}^{TC}$ -D) where N_D is the number of dummy bits.
-Dummy bits are padded at the beginning of the matrix such that Y_k = ¡NULL¿ for k = 0, 1,. . . , N_D - 1.

Then, the input data bits entered and placed after dummy bits as follow $Y_{N_D+k} = d_k^{(i)}$, k = 0, 1,. . . , D-1, and the bit sequence Y_k is written into the $(R_{subblock}^{TC} R_{subblock}^{TC})$ matrix row by row starting with bit y0 in column 0 of row 0 as in figure see 3.4

$$\begin{bmatrix} Y_0 & Y_1 & Y_2 & \cdots & Y_{c_{subblock}^{TC}-1} \\ Y_{c_{subblock}^{TC}} & Y_{c_{subblock}^{TC}+1} & Y_{c_{subblock}^{TC}+2} & \cdots & Y_{2c_{subblock}^{TC}-1} \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ Y_{(R_{subblock}^{TC}-1)*C_{subblock}^{TC}} & Y_{(R_{subblock}^{TC}-1)*C_{subblock}^{TC}+1} & Y_{(R_{subblock}^{TC}-1)*C_{subblock}^{TC}+2} & \cdots & Y_{(R_{subblock}^{TC}-1)*C_{subblock}^{TC}-1} \end{bmatrix}$$

After the write state ends reading state starts from the matrix according to permutation table see 3.2 which is the same table for $d_k^{(0)}$ and $d_k^{(1)}$ but for $d_k^{(2)}$it's different.

For $d_k^{(0)}$ and $d_k^{(1)}$:

Perform the inter-column permutation for the matrix based on the pattern ¡P (j)¿ j 0,1,. . . , $C_{subblock}^{TC} - 1$ that is shown in table see3.2, where P (j) is the original column position of the j-th permuted column. After permutation of the columns, the inter-column permuted
$(R_{subblock}^{TC} * C_{subblock}^{TC})$ matrix is as in figure, while the output of the sub-block interleaver is the bit sequence read out column by column from the inter-column permuted $(R_{subblock}^{TC} * C_{subblock}^{TC})$ matrix. The bits after sub-block interleaving are denoted by $v_0^{(i)}$,$v_1^{(i)}$, $v_2^{(i)}$,. . . ..,$v_{k-1^{(i)}}$, where $v_0^{(i)}$ corresponds to $Y_{P(0)}$, $v_1^{(i)}$ to $Y_{P(0)} + C_{subblock}^{TC}$ ..and $K\pi = (R_{subblock}^{TC} * C_{subblock}^{TC})$.

$$\begin{bmatrix} [h] & & & & \\ Y_{p(0)} & Y_{p(0)} & Y_{p(0)} & \cdots & Y_{p(c_{subblock}^{TC}-1)} \\ Y_{p(0)+c_{subblock}^{TC}} & Y_{p(1)+c_{subblock}^{TC}} & Y_{p(2)+c_{subblock}^{TC}} & \cdots & Y_{p(c_{subblock}^{TC})+c_{subblock}^{TC}} \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ Y_{p(0)+(R_{subblock}^{TC}-1)*C_{subblock}^{TC}} & Y_{p(1)+(R_{subblock}^{TC}-1)*C_{subblock}^{TC}} & Y_{p(2)+(R_{subblock}^{TC}-1)*C_{subblock}^{TC}+2} & \cdots & Y_{p(C_{subblock}^{TC})(R_{subblock}^{TC}-1)*2C_{subblock}^{TC}-1} \end{bmatrix}$$

Number of columns $C_{subblock}^{TC}$	Inter-column permutation pattern p(0),p(1) , ... P($C_{subblock}^{TC} - 1$)
32	0, 16, 8, 24, 4, 20, 12, 28, 2, 18 , 10, 26, 6, 22, 14, 30, 1, 17, 9, 25, 5, 21, 13, 29, 3, 19, 11, 27, 7, 23, 15, 31

Table 3.2: Inter-column permutation pattern

For $d_k^{(2)}$:
The output of the sub-block interleaver is denoted by $v_0^{(2)}$,$v_1^{(i)}$, $v_2^{(i)}$,. . . ..,$v_{(k_{-1)}^{(2)}}$,
where $v_k^{(i)}$ =$Y_{(k)}$
and where

$$\pi(k) = (p(K/R_{subblock}^{TC}) + C_{subblock}^{TC} * (k \mod R_{subblock}^{TC}) + 1) \mod k_\pi$$

One of the simplifications reached after tracing the equation of inter-column permutation for the third stream columns that on substituting by k for each $R_{subblock}^{TC}$ the result that can be reached is that the table can be used with adding one to each column index to be as in table see6.18.

Number of columns $C^{TC}_{subblock}$	Inter-column permutation pattern p(0)+1,p(1)+1 , ... P($C^{TC}_{subblock} - 1) + 1$
32	1, 17, 9, 25, 5, 21, 13, 29, 3, 19 , 11, 27, 7, 23, 15, 31, 2, 18, 26, 6, 22, 14, 30, 4, 20, 12, 28, 8, 24, 16, 0

Table 3.3: Inter-column permutation pattern for third input

3.2.3.2 Bit collection

This block concatenate the three streams in circular buffer at first it adds the original interleaved stream which are known as systematic bits then interlacing between two other parity streams, the circular buffer of length $K_w = 3K$ for the coded block is generated as following :

$w_k = v_k^{(0)}$ for k = 0,. . . , K_{-1}

$w_{k_\pi+2k} = v_k^{(1)}$ for k = 0,. . . , K_{-1}

$w_{k\pi+2k+1} = v_k^{(2)}$ for k = 0,. . . , K_{-1}

3.2.3.3 Bit-selection

Denote the soft buffer size for code block (only one code block for NB-IOT) by N_cb bits, where the size N_cb is obtained as follows:

$$N_{cb} = K_w$$
$$\text{Where } K_w = 3K$$

Denoting by E the rate matching output sequence length for the coded block and the redundancy version number which used to determine the starting transmission point in the buffer for this transmission,
where $rv_{idx} = 0, rv_{idx} = 2$ for NB-IOT and the rate matching output bit sequence indexes are K=0,1,. . . .,E-1. To determine the output length, define G which represents the total number of bits available for the transmission of one transport block, then set $G' = G/N_L.Q_m$ where modulation index Q_m is 1 for BPSK and 2 for QPSK. While $N_L$is equal to the number of layers a transport block is mapped onto which is equal to 1 layer in NB-IOT, therefore $G' = G/Q_m$ then the output length can be determined as $E = Qm * \lceil G' \rceil$ The last stage is to choose point to start from inside the buffer so define K_0 at which

$$K_0 = R^{TC}_{subblock}.(2.\lceil N_{cb}/(8R^{TC}_{subblock}\rceil).rv_{idx} + 2)$$

where

$R^{TC}_{subblock}$ is the number of rows defined in previous section. However, this equation can be simplified in NB-IOT to be

$$K_0 = R^{TC}_{subblock}.(24.rv_{idx} + 2), \text{ as}$$
$$N_{cb} = K_w = 3K = 3(R^{TC}_{subblock}C^{TC}_{subblock}) = 3(R^{TC}_{subblock}32).$$

Then output can be determined according the following pseudo code where k is bit index counter while j is loop counter to avoid reading NULL dummy bits and the modulus here just to represent the idea of circular buffer when $K_0 + j = N_cb$ it returns to index zero inside the buffer.[3] Set k = 0 and j = 0

```
While (k < E )
If ($w_{k_0+j} mod N_{cb} \neq NULL$)
$e_k = w_{k_0+j} mod N_{cb}$
k = k +1
End if
j = j +1
End while
```

3.2.4 Data Multiplexing and Channel interleaver

After rate matching, interleaving is applied per resource unit without any control information multiplexing in order to apply a time-first rather than frequency-first mapping, where the input sequence is the portion of e for a resource unit (output of the Rate matching) and where maximum number of columns C_{max} is

$$C_{max} = (N^{UL}_{symbol} - 1) * N^{UL}_{slot}$$

Where N^{UL}_{symbol} is the number of SC-FDMA symbols for NPUSCH in a UL resource unit, $N^{UL}_{slot}$is number of subcarriers in the frequency domain for NB-IoT and NscRU is number of consecutive subcarriers in an UL resource unit for NB-IoT as given in Table 3.3 [3]

3.2.4.1 Data and Control Multiplexing

The control and data multiplexing is performed such that HARQ-ACK information is present on both slots and is mapped to resources around the demodulation reference signals. In addition, the multiplexing ensures that control and data information are mapped to different modulation symbols. The inputs to the data and control multiplexing are the coded bits of the control information denoted by $q_0, q_1, q_2, \ldots, q_{NL*Q_CQI-1}$ and the coded bits of the UL-SCH denoted by $f_0, f_1, f_2, \ldots, f_{G-1}$The output of the data and control multiplexing operation is denoted by $g_0, g_1, g_2, \ldots, g_{H'-1}$.
Where $H = G + N_L * Q_{CQI}$ And $H' = H/N_L Q_m$ H is the total number of coded bits allocated for UL-SCH data and CQI/PMI information across the NL tansmission layers of the transport block.[3] In Narrowband IoT-LTE (Assumptions):

$$Q_m = 1 \text{ for BPSK and 2 for QPSK}$$
$$N_L = 1$$
$$Q_{CQI} = 0, \text{ As no control bits is multiplexed}$$

f0, f1, f2, .. ,
Code bits
Multiplexer
g0, g1, g2, ...
Multiplexed output

Figure 3.5: Multiplexer block diagram

```
i = k =0
While (i < G) then place the data
$g_k = [f_i, .....f_{i+Qm*N_L-1}]^T$
i=i+Qm*N$_L$
k=k+1
```

Then for BPSK :

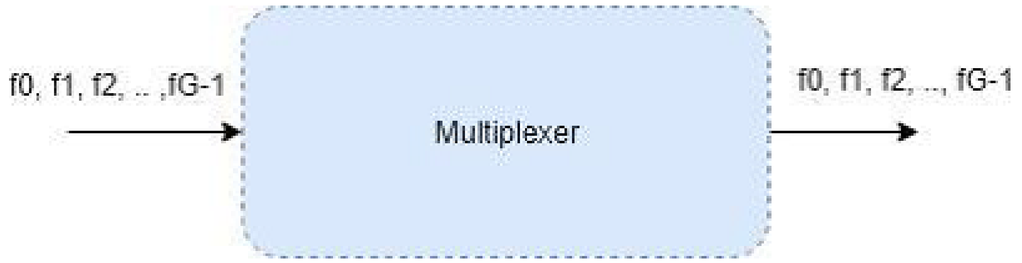


Figure 3.6: Multiplexing BPSK stream

For Qpsk

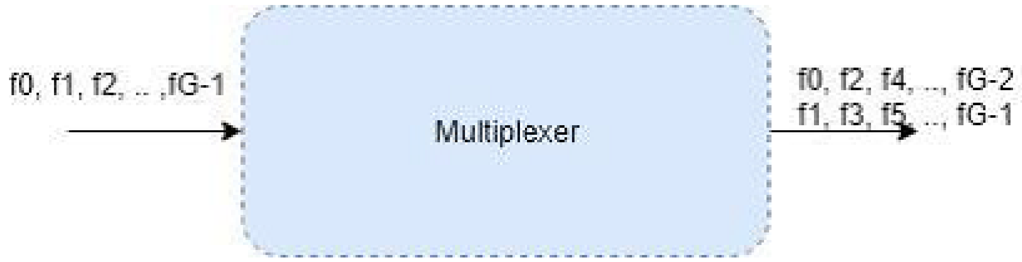


Figure 3.7: Multiplexing QPSK stream

3.2.4.2 Channel Interleaver

The channel interleaver described in this section in conjunction with the resource element mapping implements a time-first mapping of modulation symbols onto the transmit waveform while ensuring that the HARQ-ACK and RI information are present on both slots in the subframe. But the DUT did not consider any control information, so code bits $g_0, g_1, g_2, \ldots, g_{H'-1}$ are only interleaved in channel interleaver. The output bit sequence from the channel interleaver is derived as follows:

1- Assign C_{max} by equation to be the number of columns of the matrix. The columns of the matrix are numbered 0, 1, 2, . . . , C_{max1} from left to right.

2- The number of rows of the matrix is $R'_{max} = (H_{total}{}'.Q_m.N_L))C_{max}$ and we define $R_{max}{}' = R_{max}(Q_m.N_L))$. The rows of the rectangular matrix are numbered 0, 1, 2, . . . , R_max1 from top to bottom.

3-Write the input vector sequence, for k = 0, 1,. . . ,H'^{-1} into the $R_{max}C_{max}$ matrix by sets of $Q_m.N_L$ rows starting with the vector Y_0 in column 0 and rows 0 to $Q_m.N_L$-1. We filled $R_{max}C_{max}H'$ with zeros in the last $Q_m.N_{Lrows}$. (Assumption)

Where $Y_0 = f_0$ for BPSK that occupies ($Q_m.N_L = 1$) rows and $Y0 = \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}$

$$\begin{bmatrix} Y_0 & Y_1 & Y_2 & \dots & Y_{c_{max}-1} \\ Y_{c_{max}} & Y_{c_{max}+1} & Y_{c_{max}+2} & \dots & Y_{2c_{max}-1} \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ Y_{R'_{max}-1*C_{max}} & Y_{(R'_{max}-1)*C_{max}+1} & Y_{(R'_{max}-1)*C_{max}+2} & \dots & Y_{(R'_{max}-1)*C_{max}-1} \end{bmatrix}$$

The output of the block interleaver is the bit sequence read out column by column from the $R_{max}C_{max}$ matrix. The bits after channel interleaving are denoted by $h_0, h_1, h_2 \dots h_{R_{max}xC_{max-1}}$.

3.3 Physical Channel and modulation

3.3.1 Scrambler

Scrambler mainly consists of two linear feedback shift registers which simply generating a L=31- Golden Sequence C(n) by 2 paths of flip flops which initialized by two different values as shown in figure . for input codeword, the block of bits $b^{(q)}(0)$,. . . ,$b^{(q)}(M_{bit}^{(q)} - 1)$, where $M_{bit}^{(q)}$ is the number of bits transmitted in codeword on the physical uplink shared channel in one subframe, shall be scrambled with a UE-specific scrambling sequence prior to modulation, resulting in a block of scrambled bits $\check{b}^{(q)}(0)$,. . . ,$\check{b}^{(q)}(M_{bit}^{(q)} - 1)$ In order to get the required output, the first m-sequence shall be initialized with $x_1(0) = 1, x_1(n) = 0, n = 1, 2, \dots, 30$. On the other hand the initialization of the second m-sequence is denoted by $C_{init} = \sum_{i=0}^{30} x_2(i).2^i$ and c_{init} for NB-IOT is given by

$$c_{init} = n_{RNTI}.2^{14} + n_f mod2.2^{13} + \lfloor n_s 2 \rfloor.2^9 + N_{ID}^{N_c el} l.$$

This equation calculates the initial decimal value for x_2 which can be converted to its binary value to initialize the register. To work properly, initial 1600 shift cycles should be taken in consideration to induce extra randomness for the initial state, so $N_c = 1600$ and the output sequence is given by:

$C(n) = (x_1(n_{Nc}) + x_2(n + N_c)) \mod 2$
$x_1(n + 31) = (x_1(n + 3) + x_1(n)) \mod 2$
$x_2(n + 31) = (x_2(n + 3) + x_2(n + 2)) \mod 2$
, so 1600 shift cycles should be performed directly after initialization and before enabling scrambler to be able to receive any input.[3]

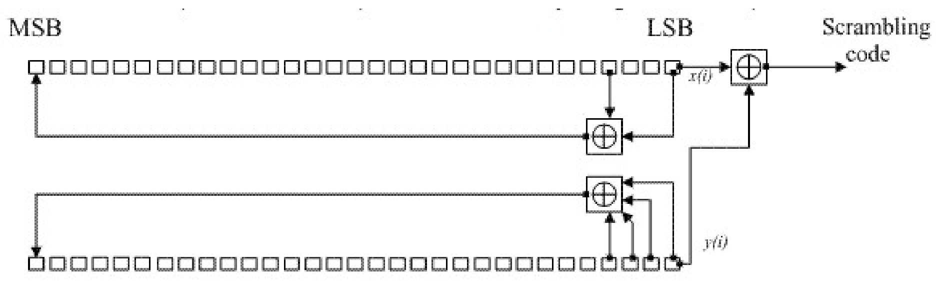


Figure 3.8: Scrambler standard internal structure

3.3.2 Modulation Mapper

Modulation is a process in which the incoming data stream is modulated onto a carrier, the modulation process involves switching the amplitude, frequency or phase of sinusoidal carrier in some fashion in accordance with the incoming data; there are three basic modulation schemes known as ASK, FSK and PSK. According to 3GPP TS 36.211 version 14.4.0 Release 14, Table see3.4 specifies the modulation mappings applicable for the narrowband physical uplink shared channel. [1]

BPSK: In case of BPSK modulation, a single bit, b(i), is mapped to a complex-valued modulation symbol x=I+jQ according to Table see 3.5

NPUSCH format	N_{sc}^{RU}	Modulation scheme
1	1	BPSK , QPSK
	1	QPSK
2	1	BPSK

Table 3.4: Modulation Mapping

b(i)	I	Q
0	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$
0	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$

Table 3.5: BPSK Modulation Mapping

QPSK

In case of QPSK modulation, a pair of bits, b(i),b(i+1), is mapped to a complex-valued modulation symbol x=I+jQ according to Table see 3.6.

b(i),b(i+1)	I	Q
00	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$
01	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$
10	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$
11	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$

Table 3.6: QPSK Modulation Mapping

3.3.3 FFT

The block of complex values x (0), x (1) ...., x($M_{symbol}^{Layer}-1$) , generated from the modulation mapper, is divided into a set of $M_{symbol}^{Layer}/M_{sc}^{Npusch}$ sets, each corresponding to one SC-FDMA symbol. Transform precoding shall be applied according to

$$y(l.M_{sc}^{Npusch}+k) = \frac{1/\sqrt{M_{sc}^{Npusch}}}{)} \sum_{(k=0)}^{M_s c^N pusch-1} x(l.M_{sc}^{pusch}+i) e^{-j\frac{2ik/(M_{sc}^{Npusch}}{)}}$$
$$k = 0, 1, \ldots\ldots M_{sc}^{Npusch}-1$$
$$l = 0, 1, \ldots\ldots.(M_{symbol}^{Layer})/(M_{sc}^{Npusch})-1$$

Resulting in a block of complex-valued symbols y (0), y (1)...., $y(M_{symbol}^{Layer}-1)$. Where$M_{sc}^{Npusch} = N_{sc}^{RU}.N_{sc}^{RU}$ is the number of allocated subcarriers and it could be determined for NB-IoT from table. Resource units are used to describe the mapping of the NPUSCH to resource elements. A resource unit is defined as $N_{slots}^{UL} * N_{symbols}^{UL}$ consecutive SC-FDMA symbols in the time domain and N_{sc}^{RU}, consecutive subcarriers in the frequency domain, where $N_{slots}^{UL}, N_{symbols}^{UL} and N_{sc}^{RU}$, are given by table. In our design we assumed working with NPUSCH format 1 with 15 KHz spacing.[1]

3.3.4 Resource element Mapper

3.3.4.1 Resource grid

A transmitted physical channel or signal in a slot is described by one or several resource grids of N_{sc}^{UL} subcarriers and N_{symbol}^{UL} SC-FDMA symbols. The resource grid is illustrated in figure. The slot number within a radio frame is denoted n_s where$n_s \epsilon$ (0 , 1 , .. , 19) for = 15 KHZ and $n_s \epsilon$ (0 , 1 , .. , 4) for = 3.75 KHZ

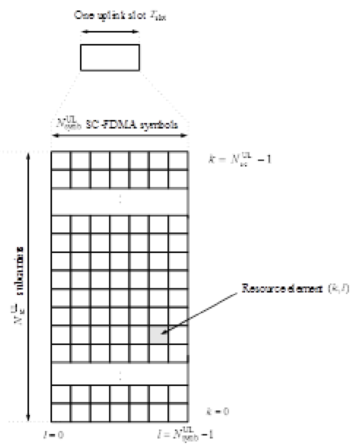

One uplink slot T_{slot}
N_{symb}^{UL} SC-FDMA symbols
$k = N_{sc}^{UL} - 1$
N_{sc}^{UL} subcarriers
Resource element (k, l)
$k = 0$
$l = 0$
$l = N_{symb}^{UL} - 1$

Figure 3.9: Uplink resource grid for NB-IoT

The uplink bandwidth in terms of subcarriers N_{sc}^{UL}, and the slot duration $T_{slot}$are given in table 3.4.

3.3.4.2 Resource element

Each element in the resource grid is called a resource element and is uniquely defined by the index pair (k,l) in a slot where k=0,. . . , $N_{sc}^{UL}-1$ and l=0,. . . ,$N_{symbol}^{UL}-1$ are the indices in the frequency and time domains, respectively. Resource element (k,l) corresponds to the complex value $a_{k,l}$. Quantities $a_{k,l}$ corresponding to resource elements not used for transmission of a physical channel or a physical signal in a slot shall be set to zero.[1]

3.3.4.3 Resource unit

Resource units are used to describe the mapping of the NPUSCH to resource elements. A resource unit is defined as $N_{symbol}^{UL}N_{slots}^{UL}$ consecutive SC-FDMA symbols in the time domain and N_{sc}^{RU} consecutive subcarriers in the frequency domain, where N_{sc}^{RU} and N_{symbol}^{UL} are given by table 1.2

3.3.4.4 Mapping to physical resources

NPUSCH can be mapped to one or more than one resource units, N_{RU} each of which shall be transmitted M_{rep}^{NPUSCH} times. The block of complex-valued symbols $z(0), \ldots, z(M_{symbol}^{ap}-1)$which are the output of the FFT block are mapped in sequence starting with z(0) to subcarriers assigned for transmission of NPUSCH. The mapping to resource elements (k,l) corresponding to the subcarriers assigned for transmission and not used for transmission of reference signals shall be in increasing order of first the index k, then the index l, starting with the first slot in the assigned resource unit. After mapping to N_{slots} slots, the N_{slots} slots shall be repeated $M_{identical}^{NPUSCH}-1$ additional times, before continuing the mapping of z(.) to the following slot, where

$$M_{identical}^{NPUSCH} = \begin{cases} min(\lfloor M_{rep}^{NPUSCH}/2\rfloor, 4) & \text{for} N_{sc}^{RU} > 1 \\ 1 & \text{for} N_{sc}^{RU} = 1 \end{cases}$$

$$N_{slots} = \begin{cases} 1 & \text{for} \delta f = 3.75KHZ \\ 2 & \text{for} \delta f = 15KHZ \end{cases}$$

DCI information of resource allocation: DCI information of resource allocation: The resource allocation information in uplink DCI format N0 for NPUSCH transmission indicates to a scheduled UE. A set of contiguously allocated subcarriers n_{sc} of a resource unit determined by the Subcarrier indication field in the corresponding DCI , A number of resource units n_{RU}

determined by the resource assignment field in the corresponding DCI according to table see 3.7. A repetition number n_{Rep} determined by the repetition number field in the corresponding DCI according to table see 3.8. The subcarrier spacing f of NPUSCH transmission is assumed 15 kHz for our operation. For NPUSCH transmission with subcarrier spacing f=15 kHz, the subcarrier indication field I_{sc} in the DCI determines the set of contiguously allocated subcarriers n_{sc} according to Table.see 3.9

Subcarrier indication field (I_{su})	Set of Allocated subcarriers (n_{sc})
0-11	I_{su}
12-15	3(I_{su})+0,1,2
16-17	6(I_{su})-16 +0, 1, 2, 3 , 4 , 5
18	0, 1, 2, 3, 4 ,5 , 6, 7 ,8 ,9 10 ,11
19-63	Reserved

Table 3.7: Allocated subcarriers

I_{su}	n_{sc}
0	1
1	2
2	3
3	4
4	5
5	6
6	8
7	10

Table 3.8: Number of resource units for NPUSCH

I_{su}	n_{sc}
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

Table 3.9: Number of resource units for NPUSCH

3.3.5 IFFT and CP

IFFT is the block responsible for transforming frequency sub channels into time domain samples to be transmitted through the RF blocks. LTE uplink uses SC-FDMA which is a modified form of the OFDM with similar throughput performance and complexity, SC-FDMA is viewed as DFT-coded OFDM where time-domain symbols are transformed to frequency domain symbols and then go through the standard OFDM modulation, SC-FDMA has all the advantages of OFDM like robustness against multi-path signal propagation, the block diagram for the SC-FDMA is shown in figure. see 3.10

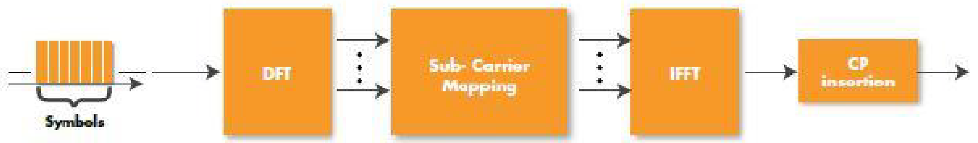


Figure 3.10: SCFDMA Block diagram

The main advantage of SC-FDMA is the low Peak Average Power Ratio (PAPR) of the transmit signal, PAPR is a big concern for user equipment, as PAPR relates to the power amplifier efficiency as low PAPR allows the power amplifier to operate close to the saturation region resulting in high efficiency that is why SC-FDMA is the preferred technology for user terminals. The NB-IoT supported bandwidth is 180 KHz with number of subcarriers depending on the spacing between sub channels, 2 sub channels are defined in the standard 15 KHz and 3.75 KHz each with its own symbol duration. For more clear understanding of the symbol durations and frequency allocation of symbols we need to illustrate this on the resource grid, the resource grid represents time on the x-axis and frequency on the y-axis. In its time domain structure, each LTE frame is 10ms long and comprises of 10 sub-frames, each of 1ms duration. The sub-frames are further divided into two time slots of 0.5ms, each comprising 7 SC-FDMA symbols in normal cyclic prefix operation and 6 SC-FDMA symbols in extended cyclic prefix mode. For the sake of simplicity, we shall restrict our discussion only to normal cyclic prefix mode. A fixed subcarrier spacing of 15 KHz gives a SC-FDMA symbol duration of 66.67 us. A cyclic prefix of duration 5.2 us is used for the first SC-FDMA symbol and 4.68 us for the rest of the symbols in each time-slot. This gives a total symbol duration of 71.87 us for the first symbol and 71.35 us for the rest of the symbols in a time-slot.

The IFFT subcarriers are grouped into sets of 12 subcarriers with spacing of 15 KHz between each adjacent sub channels, each group is called a resource block NB-IoT has one resource block only per UE transmitter. To implement IFFT block which supports 12 subcarriers as the max number of allocated sub carriers we may use 16 point IFFT as a minimum size. The less the IFFT size, the less the output sample rate is as this number of samples in addition to cyclic prefix samples are distributed on the same defined SC-FDMA symbol duration. In our implementation we use 16 point IFFT to decrease latency and power dissipation of the block and output sample rate could be compensated by an up sample filter to upgrade the rate to a convenient rate for the digital to analog converter (ADC) block. These settings are equivalent to cyclic prefix samples per Sc-fdma symbol which is 16 samples =2 cyclic prefix samples, peak data rate = 336 Kbit/s and output sample rate =36 Ksample/sec.

Chapter 4

Block level testing

The main mission of top level testing is to prove that all the internal blocks are connected and can work together as required. The top level DUT is considered as a black box That it can only be accessed through its ports and registers. This assumes that the design is complicated and it can benefit from block level testing.

Block level testing can provide great benefit if there are complicated algorithmic functions that are applied to a data stream. There may be several operations along a modem coding chain that are extremely difficult to prove completely. For instance, a FFT or IFFT translation block. In a block level environment, it is easy to stimulate the block with full random data, but in the full design, the inputs may be limited (by design). With a reference model of the algorithmic function to compare against, randomness will ensure achieving full coverage. Moreover, the advantage of block level testing is that, the DUT being smaller, simulations will run faster. Also, because the functionality of a block is only part of the top level DUT. When the full DUT is being simulated, simulation speed will get slower.
In the other hand, one disadvantage of block level testing is that special BFM's and/or models may have to be created. Also, that a separate environment has to be created for each block. Once all blocks have been proven, the top level testing should not have to concern itself with the low level functionality of these algorithms, but more on the control and overall input/output.

As discussed in chapter three , two different approaches were clarified In the first one (i.e. transaction approach), we have separated data from structure, we have not separated data stimulus from structure. This tester class is the problem. The tester creates new transactions and feeds them into the testbench. This means the tester is doing two things: choosing the order of the transactions and feeding them to the Driver. This makes reuse a problem. The tester is not the perfect solution to the problem of creating new transactions, it cannot be reused because it has the side effect of determining stimulus. The transaction type can be overridden to control data randomization, but the entire tester class also must be changed to change the number of transactions and the way they are sent. If more tests and more combinations were added, there would be more of tester classes. This would be complex hard to maintain. It's better to separate the order of the transactions (i.e. generating the data stimulus) from the Environment structure. The structure should remain unchanged regardless of the order of transactions. In the second approach (i.e. transaction approach), UVM sequences separate stimulus from the Environment structure. They allow us to create one testbench structure and then run different data through it. This is a complete and an adaptable testbench.

4.1 Transaction Approach

It was used to verify 3 block of the transmitter chain.

4.1.1 Turbo Encoder

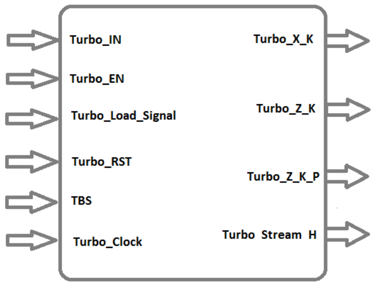


Figure 4.1: Turbo Encoder input and output ports

Port name	Direction	Width	Description
Turbo_IN	input	1 bit	Input stream
Turbo_clk	input	1 bit	system clock
Turbo_enable	input	1 bit	Block enable
Turbo_Load_sinal	input	1 bit	Input load signal
Turbo$_RST$	input	1 bit	Block reset
TBS	input	1 bit	Transport bolck size
Turbo_X_k	output	1 bit	Output stream 1
Turbo_Z_k	output	1 bit	Output stream 2
Turbo_Z_k_P	output	1 bit	Output stream 3
Turbo_Stream_H	output	1 bit	Flag signal to indicate the valid output
Turbo_done	output	1 bit	Flag to indicate the end of the valid output

Table 4.1: Turbo Encoder input and output ports

4.1.1.1 TEST PLAN

Unknown (X, Z) Signals

For the Turbo Encoder the following signals must be known under certain conditions:

Signal	Condition
Turbo_IN	Turbo_EN, Turbo_RST and Turbo_Load_Signal are all at logic ‘1’
Turbo_EN	Always in a known state
Turbo_Load_signal	Turbo_EN and Turbo_RST are both at logic ‘1’
Turbo_RST	Always in a known state
TBS	Turbo_EN, Turbo_RST and Turbo_Load_Signal are all at logic ‘1’
Turbo_X_k	Turbo_Stream_H is at logic ‘1’
Turbo_Z_k	Turbo_Stream_H is at logic ‘1’
Turbo_Z_k_P	Turbo_Stream_H is at logic ‘1’
Turbo_Stream_H	always be in a known state
Turbo_done	always be in a known state

Table 4.2: Turbo Encoder Unknown (X, Z) Signals

Timing Relationships

For the Turbo Encoder, the following temporal relationships can be defined:

Sequence	What to check
Output length	Turbo_Stream_H = 1 / For a number of cycles according to TBS
Trellis Termination	Initially : Output length sequence
	After 2 cycles: Turbo_Stream_H = 1 / For 4 cycles
Transaction done	Initially : Trellis Termination sequence
	After 1 cycle: Turbo_done = 1

Table 4.3: Turbo Encoder temporal timing relationships

Functional Coverage

The fields that are relevant to Turbo Encoder functional coverage are:

Specifications	cover
TBS	Cover all values in table 3.1

Table 4.4: Turbo Encoder Functional coverage

4.1.1.2 Environment

Transaction This class encapsulates all Turbo Encoder driving variables. A set constraints was defined on these variables.

Constraints

1-**Transport block size Constraint:**

- Since we have only specific values to TBS signal, we need to limit the randomization of TBS to be as follow TBS∈(Table 3.1)

2-**Control Constraint**:

- Gives distribution to the constraint solver for **Turbo_EN,** 90% logic '1' and the other 10% is logic '0'
- Gives distribution to the constraint solver for **Turbo_RST,** 90% logic '1' and the other 10% is logic '0'
- Gives distribution to the constraint solver for **Turbo_EN,** 90% logic '1' and the other 10% is logic '0'

Driver

The driver gets the transaction from the tester. While Turbo_EN and **Turbo_RST** are both at logic 1, the driver asserts **Turbo_Load_Signal** and starts to send the input bits for (TBS + 24) consecutive cycles then **Turbo_Load_Signal** shall go to a logic 0 in the next cycle. The designed block is not pipelined. therefore, the driver goes to a sleep state waiting for **Turbo_done** signal to be sampled at logic 1 in order to get the next transaction.

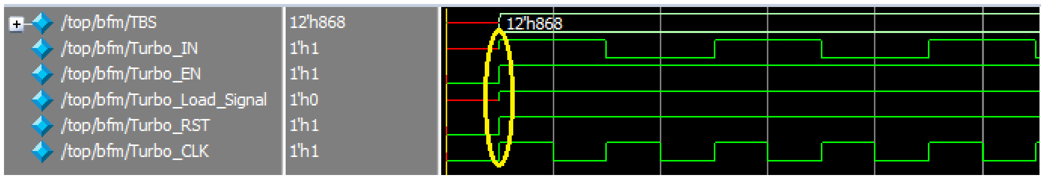


Figure 4.2: Turbo Encoder Input driving

Input Monitor

Once **Turbo_EN**, **Turbo_RST** and **Turbo_Load_Signal** are all at logic 1, the monitor enters a loop of sampling **Turbo_IN** for (TBS+24) consecutive clock cycles. The sampling takes place at the positive edge. Meanwhile, in each cycle the monitor checks that the control signals and TBS are stable. If not, it breaks the sampling loops.

Once the sampling loop is completed successfully, the monitor broadcast the transaction to the coverage and scoreboard components via the TLM Analysis port. The monitor waits for **Turbo_Load_Signal** to go to logic 1 again which indicates a new transaction

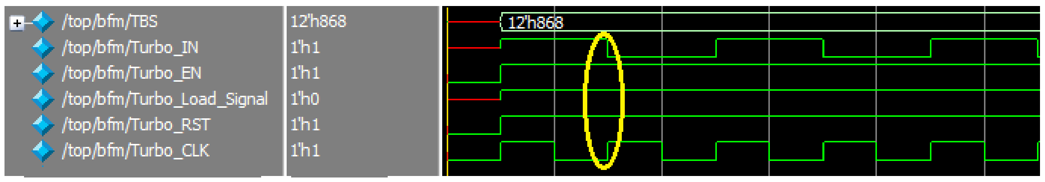


Figure 4.3: Turbo Encoder input sampling

Output Monitor

As a result of the output bits being generated in inconsecutive clock cycles, the output monitor enters a waiting state until Turbo_Stream_H is sampled at logic 1. Once this event is triggered, **Turbo_X_k**, **Turbo_Z_k** and **Turbo_Z_k_P** will be sampled at the same positive edge.

The monitor continues in the previous operation until Turbo_done is to be sampled at logic 1 which means the transaction is completed.

If the number of sampled output bits were equal to (TBS+28), the output monitor sends the output transaction to the scoreboard component via the TLM Analysis port.

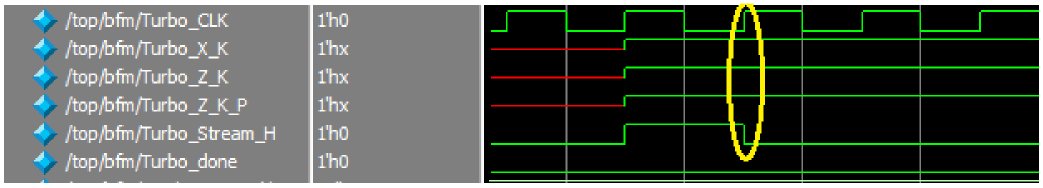


Figure 4.4: Turbo Encoder output sampling

Scoreboard

The scoreboard is implemented based on the two constituent encoders as shown in 2.3 Predict result function is responsible for predicting the output, which will be compared with the DUT output.

- The received input transaction will be interleaved based on TBS see 3.1
- The output bits will be calculated through some Boolean expressions using for loop.
- Trellis termination bits is calculated separately as X_K' is part of the padded bits
- The encoded information is padded with the generated trellis termination Then the comparing process starts

Coverage

We've defined our functional coverage model in terms of stimulus in (test plan chapter). The Turbo Encoder is fully tested if we've run a complete set of these terms:

- **TBS**

Transport block size is nothing but the payload for physical layer. All of the values shown in table see 3.1 should be check to fully test the internal interleaver.
A covergroup is used to capture functional coverage for TBS bus. There is a defined coverage bin for each value of TBS in the coverage model. When the write function is invoked. The Coverage should be triggered to sample the coverage values

Assertions

One way to verify the Rate matcher is to implement a monitor as a System Verilog interface that uses a mixture of SVA concurrent assertions to observe and check the bus traffic.
It's a passive verification component which monitors the block signals, it can be attached to external signals in the top level of the UVM Environment. In this module we check the value of the interface signals, if one of them is in an undefined state, then it will cause a problem under certain conditions. There are two type of signals to be checked:

1. Signal must be known all time to ensure proper operation of the whole chain; these signals are:

- Turbo_RST
- Turbo_EN
- Turbo_Stream_H
- Turbo_done

2. Signal must be known only under certain condition; these signals are:

- Turbo_IN
- TBS
- Turbo_X_k
- Turbo_Z_k
- Turbo_Z_k_P

This module is also used to verify the signals sequences for all types of transfers to ensure successful communication between the Turbo Encoder and the other blocks in the Transmitter chain.

The sequences checked by this Assertions module are:

1. **Output sequence length .**

The Output length of encoded information is variable according to the Transport block size. the coding rate of turbo encoder is 1/3 with (TBS+24) output bits generated on each bus. Therefore, **Turbo_Stream_H** shall be at logic 1 for a number of (TBS+24) inconsecutive cycles.

2. **Trellis Termination sequence.**

Trellis termination is performed by taking the tail bits from the shift register feedback after all information bits are encoded. Tail bits are padded after the encoding of information bits. **Turbo_Stream_H** shall be sampled at logic 1 for 4 consecutive cycles after the output length sequence is finished.

3. **Transaction done sequence.**

Turbo_done indicates that the block finished the current operation and is ready for new input. Turbo_done must be sampled at logic one after **Turbo_Stream_H** is sampled at logic ‘1’ for (TBS+28) cycles.

These sequences are used to verify the timing between commands.

4.1.2 Modulator

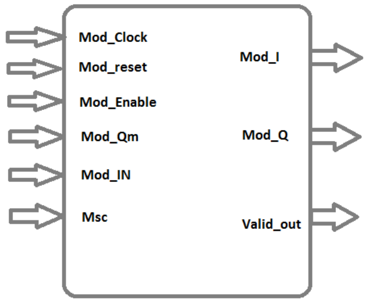


Figure 4.5: Modulator input and output ports

Port name	Direction	Width	Description
Mod_IN	input	1 bit	Serial input stream from scrambler
Mod_clk	input	1 bit	system clock
Mod_EN	input	1 bit	Block enable
Mod_IN_EN	input	1 bit	Shows whether the value on Mod_IN bus is valid or not
Mod_RST	input	1 bit	Active low reset signal, puts the block into its initial state
modQ$_m$	input	1 bit	Modulation index (BPSK or QPSK), upper layer parameter
Msc	input	2 bits	Number of subcarriers, upper layer parameter
Mod_O_valid	output	1 bit	Signal to show the current values of Mod$_I andMod_Q arevalid$
mod_I	output	16 bit	Real part of output symbol, represented in fixed point (integer part: 6 bits, fraction part: 10 bits)
mod_Q	output	16 bit	Imaginary part of output symbol, represented in fixed point (integer part: 6 bits, fraction part: 10 bits)

Table 4.5: Turbo Encoder input and output ports

4.1.2.1 TEST PLAN

Unknown (X, Z) Signals

For the Modulator the following signals must be known under certain conditions:

Signal	Condition
Mod_IN	Mod_RST, Mod_EN and Mod_IN_EN are all at logic '1'
Mod_EN	Always in a known state
Mod_IN_Enable	Mod_RST and Mod_EN are both at logic '1'
Mod_Q_m	Mod_RST and Mod_EN are both at logic '1'
Msc	Mod_RST and Mod_EN are both at logic '1'
Mo_I	Mod_O_Valid is at logic '1'
Mo_Q	Mod_O_Valid is at logic '1'
Mo_o_valid	Always in a known state

Table 4.6: Modulator Unknown (X, Z) Signals

Timing Relationships

For the Modulator, the following temporal relationships can be defined:

Sequence	What to check

Reset Effect	Mod_O_Valid = 0
	Mod_Q = 0
	Mod_I = 0

Output Timing	Initially : Mod_IN_EN == 1
	After a number of cycles depending on modulation index: Mod_O_Valid = 1 / For 1 cycle

Table 4.7: Modulator temporal timing relationships

Functional Coverage

The fields that are relevant to the Modulator functional coverage are:

Specifications	cover
Mod_IN	Cover all possible input symbols
Mod_Q_m	Cover all modulation schemes
Msc	Cover all possible subcarrier numbers
Mod_IN, Msc,Mod_Qm	Cover all combination of all inputs

Table 4.8: Modulator Functional coverage

4.1.2.2 Environment

Transaction

This class encapsulates all Modulator driving variables. A set constraints was defined on these variables.

Constraints

1- Modulation index constraint:

- There are 2 values for the modulation index BPSK and QPSK, the modulation index is randomized with equal distribution for these values

2- Msc Constraint:

- There are 3 values for the number of subcarriers, 3, 6 and 12 subcarriers. It's randomized with equal distribution for these values

3-Control Constraint:

- Gives distribution to the constraint solver for **Mod_EN**, 90% logic '1' and the other 10% is logic '0'

- Gives distribution to the constraint solver for **Mod_RST**, 90% logic '1' and the other 10% is logic '0'

- Gives distribution to the constraint solver for **Mod_IN_EN,** 90% logic '1' and the other 10% is logic '0'

Driver

The driver receives the transactions from the tester which include the values for input signals, it converts a transaction to its corresponding bit level signals and applies them to the DUT interface. The driver checks the modulation index, if it's 0 i.e. BPSK modulation, it drives a single bit to the interface while applying the given value of **Mod_IN_EN**, if it's 1 i.e. QPSK modulation, it drives a pair of bits to the interface

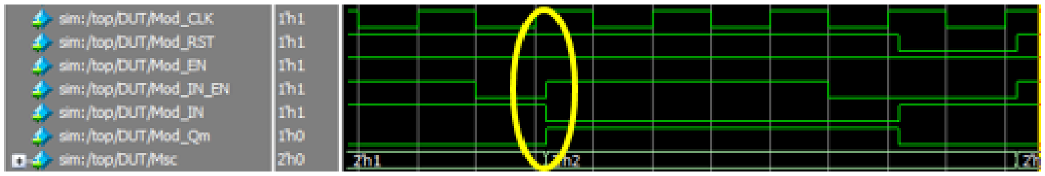


Figure 4.6: modulator driving

Input Monitor

The monitor checks at every positive edge of the clock for the Mod_IN_EN signal, if it's sampled at logic '1', it samples the data from the DUT's interface. If the modulation index is at logic '0' (i.e. BPSK), the monitor groups the data in a transaction and writes it in the analysis port for the subscribers to read. If it's at logic '1' it waits another clock cycle to capture the second bit of the transaction (i.e. QPSK).

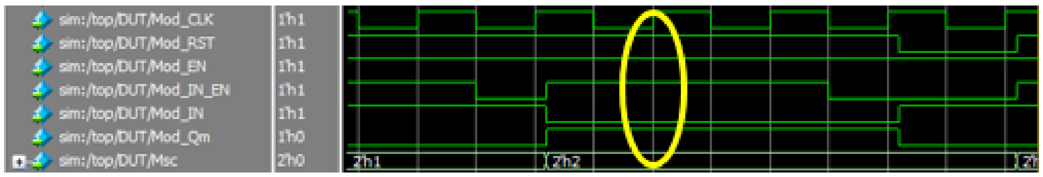


Figure 4.7: modulator sampling

Output Monitor

The output monitor checks at every positive edge of the clock if the Mod_O_Valid signal is sampled at logic '1', then it samples the real and imaginary outputs of the modulation symbol and writes them into its analysis port.

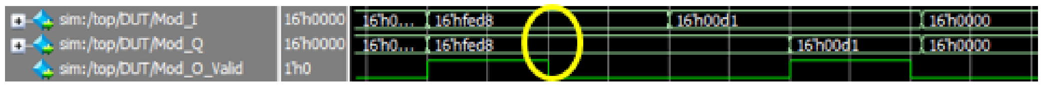


Figure 4.8: Modulator output sampling

Scoreboard

The prediction in the case of the modulator is simple; a case statement that checks the value of the input bits, the modulation index and the number of subcarriers, it produces the corresponding real and imaginary part for that symbol. The following table shows the fixed point values for each corresponding value of the constellation points

Real value	Fixed point value
$\frac{1}{\sqrt{2}} * \frac{1}{\sqrt{3}}$	4'h01A2
$\frac{1}{\sqrt{2}} * \frac{1}{\sqrt{6}}$	4'h0128
$\frac{1}{\sqrt{2}} * \frac{1}{\sqrt{12}}$	4'h00D1
$-\frac{1}{\sqrt{2}} * \frac{1}{\sqrt{3}}$	4'hFESE
$-\frac{1}{\sqrt{2}} * \frac{1}{\sqrt{6}}$	4'hFED8
$-\frac{1}{\sqrt{2}} * \frac{1}{\sqrt{12}}$	4'hFF2F

Table 4.9: Modulator Fixed Point Representation mapping

Coverage

We've defined our functional coverage model in terms of stimulus in (test plan chapter). The coverage block has 3 covergroups, Scheme, BPSK and QPSK

- **Scheme**

This covergroup is used to cover both modulation schemes, BPSK and QPSK, through the input Qm which is the modulation index. One coverpoint is enough to accommodate for it. Two more covergroups are needed to cover the data and subcarrier cases.

- **BPSK**

This covergroup has 2 coverpoints, one for the input symbol (Mod_IN), and another for the number of subcarrier (**Msc**). It also has cross coverage between these two coverpoints

- **QPSK**

IThis Covergroup has the same coverpoints as BPSK, the difference is that in case of BPSK, the input symbol is 1 bit, while in case of QPSK the input symbol is two bits, so this covergoup samples two consecutive bits driven to the input (**Mod_IN**) and covers all the cases for the input symbol. The coverpoint for Msc and cross coverage are the same as BPSK. The combination of these 3 covergroups insures the Modulator has been tested thoroughly

- **Cross coverage**

All the previous covergroups are crossed together to make sure that the modulator's exercised all possible combinations and has been tested thoroughly

Assertions

One way to verify the Modulator is to implement a monitor as a System Verilog interface that uses a mixture of SVA concurrent assertions to observe and check the bus traffic. It's a passive verification component which monitors the block signals, it can be attached to external signals in the top level of the UVM Environment. In this module we check the value of the interface signals, if one of them is in an undefined state, then it will cause a problem under certain conditions. There are two type of signals to be checked:

1. Signal must be known all time to ensure proper operation of the whole chain; these signals are:

- Mod_RST
- Mod_EN
- Mod_O_Valid

2. Signal must be known only under certain condition; these signals are:

- Mod_IN_EN
- Mod_IN
- Mod_Qm
- Msc

This module is also used to verify the signals sequences for all types of transfers to ensure successful communication between the Modulator and the other blocks in the Transmitter chain. The sequences checked by this Assertions module are:

1. **Reset effect sequence:**

When reset is at logic '0', all the outputs of the modulator must be sampled at logic '0'

2. **Output timing sequence:**

There are two cases depending on the modulation index i.e. (Mod_Qm):

- BPSK

If Mod_IN_EN is sampled at logic 1 for 1 clock cycle and Mod_Qm=0 (i.e. BPSK) then on the next positive edge of the clock, Mod_O_Valid should be sampled at logic 1 for one cycle indicating an output symbol is valid

- QPSK

If **Mod_IN_EN** is sampled at logic 1 for 2 clock cycles and Mod_QM=1 (i.e. QPSK) then on the next positive edge of the clock, Mod_O_Valid should be sampled at logic 1 for one cycle indicating an output symbol is valid. These sequences are used to verify the timing between commands.

4.1.3 Resource Element Mapper

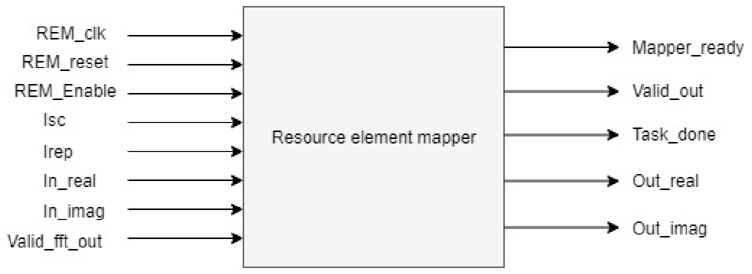


Figure 4.9: REM input and output ports

Port name	Direction	Width	Description
REM_clk	input	1 bit	system clock
REM_enable	input	1 bit	Block enable
REM_res	input	1 bit	Block reset
Isc	input	5 bits	Upper layer signal to tell the number of subcarriers and their start index
Irep	input	3 bits	Upper layer signal to tell the number of repetitions
In_real	input	16 bits	Real part of the input symbol
In_imag	input	16 bits	Imaginary part of the input symbol
Mapper_ready	output	1 bit	Flag to tell the previous block that the rem is ready to receive input
Valid_FFT_out	input	1 bit	Fag to tell that the output of the previous block is valid
Valid_out	output	1 bit	Flag to indicate that Output of REM is valid

Table 4.10: Resource Element Mapper input and output ports

4.1.3.1 TEST PLAN

Unknown (X, Z) Signals

For the Resouce Element Mapper the following signals must be known under certain conditions:

Signal	Condition
REM_res	always be in a known state
Isc	REM_enable and REM_res are both at logic ‘1’
REM_enable	always in a known state
Irep	REM_enable and REM$_r esarebothatlogic$‘$1'$
In_imag	REM_enable, REM$_r esandValid\_FFT\_outareallatlogic$‘$1'$
In_real	REM_enable, REM$_r esandValid\_FFT\_outareallatlogic$‘$1'$
Task_done	always be in a known state
Valid_out	always be in a known state
Out_imag	REM_enable,REM_res and valid_out are at logic ‘1’
Out_real	REM_enable,REM_res and valid_out are at logic ‘1’

Table 4.11: Resource element mapper Unknown (X, Z) Signals

Timing Relationships

For the Resource Element Mapper, the following temporal relationships can be defined:

Sequence	What to check
Output latency	After a number of cycles according to Isc and Irep: Valid_out = 1
Output length	Valid_out = 1 / For 12 cycles
Transaction done	Initially : Output length
	After 1 cycles: MTask_done = 1 / For 4 cycles
Operation holding	Initially: Irep = 3,4,5,6,7 , Output length / 13 times
	Maper_ready = 0

Table 4.12: Resource element mapper temporal Timing relationship

Functional Coverage

The fields that are relevant to REM functional coverage are:

Specifications	cover
Isc	Cover all values from 0 to 18
Irep	Cover all values from 0 to 7
Ise & Irep	Cover all combinations of the two inputs

Table 4.13: Resource element mapper functional coverage

4.1.3.2 Environment

Transaction This class encapsulates all Turbo Encoder driving variables. A set of constraints was defined on these variables.

Constraints

1- Number of subcarriers Constraint

- Since we have only specific values to Isc signal, we need to limit the randomization of Isc to be as follow $I_{sc} \in$ [0:18] as specified in (3.7)

2-Control Constraint:

- Gives distribution to the constraint solver for **REM_enable**, 90% logic '1' and the other 10% is logic '0'
- Gives distribution to the constraint solver for **REM_res**, 90% logic '1' and the other 10% is logic '0'
- Gives distribution to the constraint solver for **Valid_FFT_out,** 90% logic '1' and the other 10% is logic '0'
- Gives distribution to the constraint solver for rem_out_en, 90% logic '1' and the other 10% is logic '0'

Driver

While **REM_enable** and **REM_res** are both at logic 1, the driver asserts **Valid_FFT_out** and starts to send the input according to number of subcarrier in consecutive cycles then **Valid_FFT_out** shall go to logic 0 in the next cycle.

The designed block is not pipelined. Therefore, the driver goes to a sleep state waiting for **Task_done** signal to be sampled at logic '1' in order to get the next transaction.

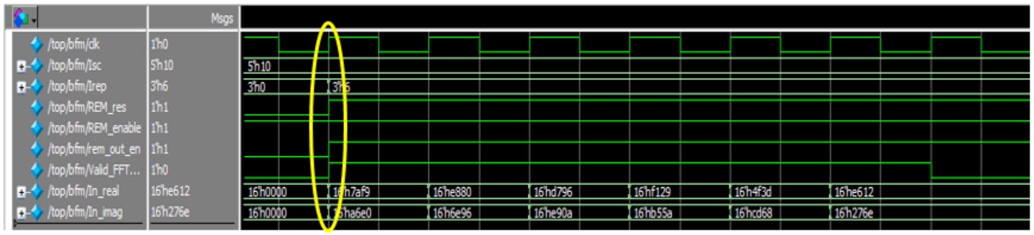


Figure 4.10: Resource Element Mapper input driving

Input Monitor

Once **REM_enable** and **REM_res** and **Valid_FFT_out** are all at logic '1', the monitor enters a loop of sampling **Input_real**, **Input_imag**, **Isc** and **Irep** for number of clock cycles depending on the number of subcarrier allocated and the number of repetition. The sampling takes place at the positive edge. Meanwhile, in each cycle the monitor checks that the control signals are stable. If not, it breaks the sampling loops.

Once the sampling loop is completed successfully, the monitor broadcast the transaction to the coverage and scoreboard components via the TLM Analysis port. The monitor waits for the repetition of Task_done to be sampled at logic 1 for 14 times which indicates a new transaction.

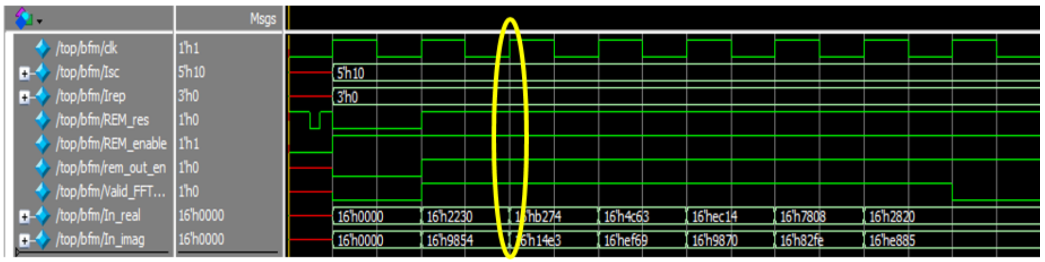

Msgs
/top/bfm/clk 1'h1
/top/bfm/Isc 5'h10
/top/bfm/Irep 3'h0
/top/bfm/REM_res 1'h0
/top/bfm/REM_enable 1'h1
/top/bfm/rem_out_en 1'h0
/top/bfm/Valid_FFT... 1'h0
/top/bfm/In_real 16'h0000
/top/bfm/In_imag 16'h0000
5'h10
3'h0
16'h0000 16'h2230 16'hb274 16'h4c63 16'hec14 16'h7808 16'h2820
16'h0000 16'h9854 16'h14e3 16'hef69 16'h9870 16'h82fe 16'he885

Figure 4.11: Resource Element Mapper input sampling

Output Monitor

As a result of the output being generated in inconsecutive clock cycles, the output monitor enters a waiting state until Valid_out is sampled at logic 1. Once this event is triggered, Out_real and Out_imag will be sampled at the same positive edge.

The monitor continues in the previous operation until Task_done has been sampled be at logic 1 for 14 times which means the transaction is completed. Then the output monitor sends the output transaction to the scoreboard component via the TLM Analysis port.

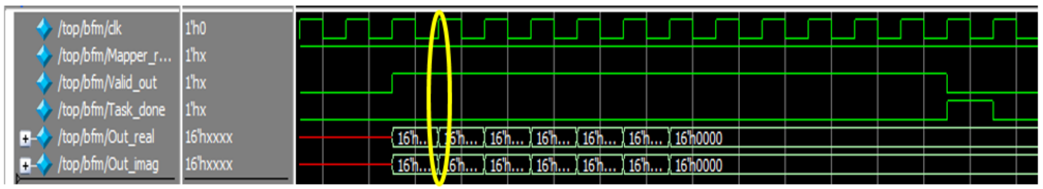

/top/bfm/clk 1'h0
/top/bfm/Mapper_r... 1'hx
/top/bfm/Valid_out 1'hx
/top/bfm/Task_done 1'hx
/top/bfm/Out_real 16'hxxxx
/top/bfm/Out_imag 16'hxxxx
16'h... 16'h... 16'h... 16'h... 16'h... 16'h... 16'h0000
16'h... 16'h... 16'h... 16'h... 16'h... 16'h... 16'h0000

Figure 4.12: Resource Element Mapper output sampling

Scoreboard

We need to model a memory which represent resource grid as shown in 1.6

- the repetition is done after one resource unit N_{RU} which equal to 14 blocks of input data every block is equal to number of subcarriers. In order to test the repetition feature the input transaction must be at least 14 N_{RU} , so we model this using three dimensions array, first dimension indicates the input data width which is equal to 16 bit, second dimension indicates the number of subcarriers allocated which has a maximum value of 12 and the third dimension indicates slots which has a maximum value of 14 slots to test the repetition feature.

- Consequently, the output of DUT is equal to (12 $\times$ 14 $\times$ number of repetitions) so we model this with four dimensions' array, first dimension indicates the width of output data (16 bits), second dimension indicates the outputs of every slot which is equal to 12, the third dimension indicates number of slots which is equal to 14 slots and the fourth dimension indicate the number of repetitions.
- Using the received input transaction, we assign in the four-dimensions array using for nested loops first loop on number of subcarrier (frequency) and second loops on number of slots (time) and the third loops on number of repetition

Coverage

We've defined our functional coverage model in terms of stimulus in (test plan chapter). REM is fully tested if we've run a complete set of these terms:

- Isc

  Upper layer parameter which determines the number of subcarrier and the starting index of subcarriers. All of the values vary from 0 to 18 should be check to fully test all possible register access which indicates all possible subcarriers. A covergroup is used to capture functional coverage for Isc bus. There is a defined coverage bin for each value of Isc in the coverage model.

- Irep

  Upper layer parameter which determines the number of repeated blocks of data. All of the values vary from 0 to 7 should be check to fully test all possible register access which indicates all possible repetition. A covergroup is used to capture functional coverage for Irep bus. There is a defined coverage bin for each value of Irep in the coverage model.

- Cross coverage

  All the previous Coverpoint are crossed together to make sure that the REM's exercised all possible combinations and has been tested thoroughly. When the write function is invoked. The Coverage should be triggered to sample the coverage values.

Assertions

way to verify the REM is to implement a monitor as a System Verilog interface that uses a mixture of SVA concurrent assertions to observe and check the bus traffic.
It's a passive verification component which monitors the protocol signals, it can be attached to external signals in the top level of the UVM Environment. In this module we check the value of the interface signals, if one of them is in an undefined state, then it will cause a problem under certain conditions.

There are two type of signals to be checked:
1. Signal must be known all time to ensure proper operation of the whole chain; these signals are:

- REM_res
- REM_enable
- Task_done
- Valid_out

Signal must be known only under certain condition clarified in table (); these signals are:

- Isc
- Irep
- In_imag
- In_real
- Out_imag
- Out_real

This module is also used to verify the signals sequences for all types of transfers to ensure successful communication between REM block and the other blocks in the Transmitter chain.

These sequences are used to verify the timing between commands:

1. **Output latency sequence.**

This sequence is to check the Latency of the block; we are concerned with the timing of first valid output. The time of the rising edge of Valid_out indicates the latency of the block.

2. **Output length sequence.**

The Output length of REM block is 12 symbols for every transaction and it's generated in consecutive cycles. Valid_out indicates that the generated output is valid and ready to be sampled. It must be held at logic '1' for 12 consecutives cycles.

3. **Transaction done sequence.**

Output length of block is constant and equal to 12 and the output is generated in consecutive cycles. Task_done indicates that the block finished the current operation and is ready for new input. the rising edge of Task_done must happen after 12 consequences Valid_out.

4. **Operation holding.**

FFT takes 54 cycles to produce new data to the resource element mapper, however, if repetition occurs, the REM will be sending old data stored in its memory, so the data from the FFT will be lost. Therefore, the REM sends a signal (i.e. Mapper_ready) to the control unit to tell if it can receive new input or not. If it's sampled to be at logic 1, the control unit disables the FFT and all the previous blocks, if I's not, the chain runs in the normal operation.
These sequences are used to verify the timing between commands.

4.2 Sequence Approach

It was used to verify 3 block of the transmitter chain.

4.2.1 Rate Matcher

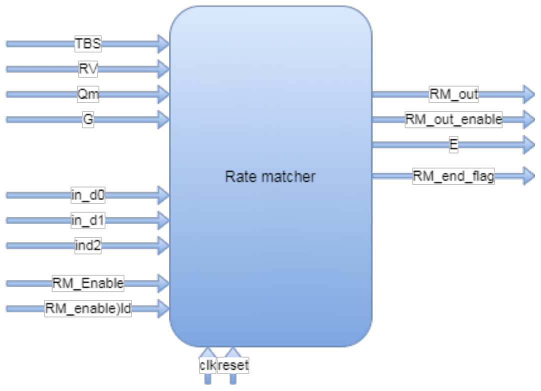


Figure 4.13: Rate Matcher input and output ports

Port name	Direction	Width	Description
RM_clk	input	1 bit	system clock
RM_Enable	input	1 bit	Block enable
RM_enable_ld	input	1 bit	Input load signal
RM$_reset$	input	1 bit	Block reset
TBS	input	12 bits	Transport bolck size
G	input	12 bits	determine output sequence length
R_v	input	1 bit	Redundancy version
Q_m	input	12 bits	Modulation index
E	output	12 bits	Output sequence length
RM_out	output	1 bit	Output bit stream
RM_out_enable	output	1 bit	Valid output enable

Table 4.14: Rate matcher input and output ports

4.2.1.1 TEST PLAN

Unknown (X, Z) Signals

For the Rate Matcher the following signals must be known under certain conditions:

Signal	Condition
In_d0	RM_enable_ld, RM$_{E}nable, RM\_resetareallatlogic‘1'$
In_d1	RM_enable_ld, RM$_{E}nable, RM\_resetareallatlogic‘1'$
In_d1	RM_enable_ld, RM$_{E}nable, RM\_resetareallatlogic‘1'$
RM_Enable	Always in a known state
RM_enable_ld	RM_Enable and RM_reset are both at logic 1
RM$_{r}eset$	Always in a known state
TBS	RM_Enable, RM_reset and RM_enable_ld are all at logic ‘1’
G	RM_Enable, RM_reset and RM_enable_ld are all at logic ‘1’
Rv	RM_Enable, RM_reset and RM_enable_ld are all at logic ‘1’
Qm	RM_Enable, RM_reset and RM_enable_ld are all at logic ‘1’
RM_out	RM_out_enable is at logic ‘1’
RM_out_enable	Always in a known state
RM_end_flag	Always in a known state

Table 4.15: Rate matcher unknown (X, Z) Signals

Timing Relationships

For the Rate Matcher, the following temporal relationships can be defined:

Sequence	What to check
Output length	RM_out_enable = 1 / For a number of cycles according to G
Transaction done	Initially : Output length
	After 1 cycle: RM_end_flag = 1 / For at least 1 clock cycle and the output is not a dummy bit.

Table 4.16: Rate matcher temporal timing relationshipsp

Functional Coverage

The fields that are relevant to Rate Matcher functional coverage are:

Specifications	cover
TBS	Cover all values in (lookup table number)
G	Cover all values from 0: 2880
Rv	cover all vaues
Qm	cover all vaues

Table 4.17: Rate Matcher functional coverage

4.2.1.2 Environment

Sequence item

This class encapsulates all Rate Matcher driving variables. A set constraints was defined on these variables.

Constraints

1-Transport block size Constraint:

- Since we have only specific values to TBS signal, we need to limit the randomization of TBS to be as follow TBS(Table elvalues)

2-G value constraint:

- The maximum value for G is 2880 therefor the size is [11:0] 12 bits.

3- Redundancy version constraint

- Gives distribution to the constraint solver for Rv, 90% logic ‘0’ and the other 10% is logic ‘1’ for retransmissions cases

4- Modulation index constraint:

- Gives distribution to the constraint solver for Qm, 50% logic ‘1’ and the other 50% is logic ‘0’

5-Control Constraint:

- Gives distribution to the constraint solver for RM_Enable, 95% logic ‘1’ and the other 5% is logic ‘0’
- Gives distribution to the constraint solver for RM_reset, 95% logic ‘1’ and the other 5% is logic ‘0’
- Gives distribution to the constraint solver for RM_enable_ld, 95% logic ‘1’ and the other 5% is logic ‘0’

Driver

While RM_Enable, RM_reset are both at logic 1, the driver asserts and RM_enable_ld and starts to feed the DUT through in_d0, in_d1 and in_d2 for (TBS + 28) clock cycles then RM_enable_ld shall go to a logic 0 in the next cycle.

The designed block is not pipelined. therefore, the driver goes to a sleep state waiting for RM_end_flag signal to be sampled at logic 1 in order to get the next transaction.

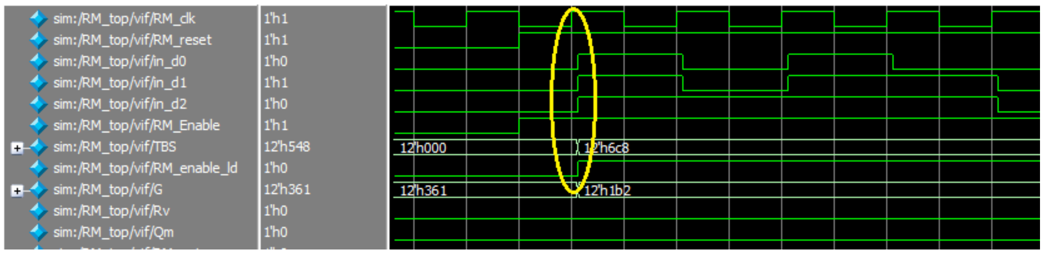


Figure 4.14: Rate Matcher input driving

Input Monitor

Once **RM_Enable, RM_reset** and **RM_enable_ld** are all at logic '1', the control signals **TBS, G, Rv** and **Qm** are sampled in the same positive edge clock cycle of RM_enable_ld then the monitor enters a loop of sampling in_d0, in_d1 and in_d2 for (TBS+28) clock cycles. The sampling takes place at the positive edge. Meanwhile, in each cycle the monitor checks that the control signals, TBS and G are stable. If not, it breaks the sampling loops.

Once the sampling loop is completed successfully, the monitor broadcast the transaction to the coverage and scoreboard components via the TLM Analysis port.

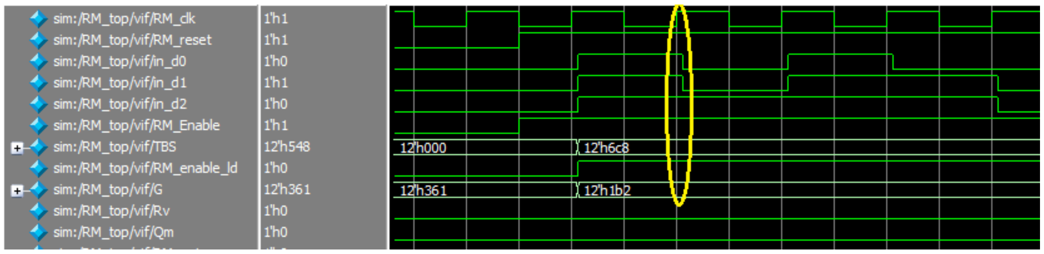


Figure 4.15: Rate Matcher input sampling

Output Monitor

There are two main cases that the output monitor is dealing with. First in the normal scenario, the output monitor enters a waiting state until **RM_out_enable** is sampled at logic 1. Once this event is triggered, **RM_out** and E will be sampled at the same positive edge. The monitor continues in the previous operation until **RM_end_flag** is to be sampled at logic 1 which means the transaction is completed.

If the number of sampled output bits will be equal to E, the output monitor sends the output transaction to the scoreboard component via the TLM Analysis port.

In the second scenario, the DUT has no output for the sent input transaction. Therefore, the result monitor will break the sampling loop after 10000 clock cycles and send zero output to scoreboard. the output monitor sends the output transaction to the scoreboard component via the TLM Analysis port.

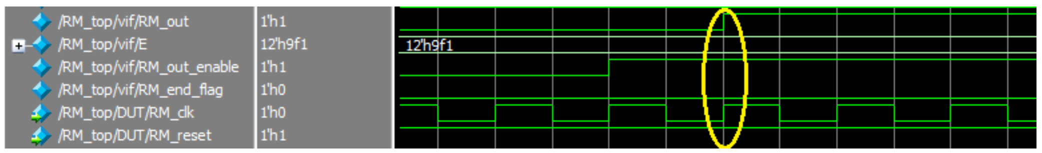


Figure 4.16: Rate Matcher output sampling

Scoreboard

The implemented scoreboard for the rate-matcher can be described as follows: 2.5 Three matrices for each input vector are allocated to store the input vector from the turbo encoder (i.e. data, data parity, data parity interleaved) in the three sub-blocks interleave as mentioned in the rate-matcher standard see 3.2.3 .

- The received input transaction will be interleaved based on TBS see 3.1
- The size of these matrices is Row32 where $Rows = \lceil \frac{TBS}{32} \rceil$, 32 is the fixed number of columns. the input bits number is not necessary equal to Row32 , So to fill the matrix elements we add dummy bits in the first row of these matrices the dummy bits are modeled as high impedance 'Z' in the model.
- After filling the matrix with dummy bits and input data the interleaving phase comes which is an inter-column interleaving according to fixed tables that are mentioned in see 3.2.
- The bit collection phase, the 3 matrices are concatenated in a new matrix, to do the bit selection or to determine where to start to generate the output bits we should calculate the K0 as follows $Rows*((24*RV)+2)$. The output sequence length E should be calculated to determine how many bits the model will generate $E = (Q_m + 1) * (\frac{G}{Q_m+1}$. Then the comparing process starts.

Coverage

We've defined our functional coverage model in terms of stimulus in (test plan chapter). A covergroup is used to capture functional coverage for control signals. The Rate Matcher is fully tested if we've run a complete set of these terms:

- TBS

Transport block size is nothing but the payload for physical layer. All of the values shown in Table (lookup) should be check to fully test the internal interleaver. There is a defined coverage bin for each value of TBS in the coverage model. When the write function is invoked. The Coverage should be triggered to sample the coverage values

- G

an upper layer parameter that is used to calculate the required output sequence. this parameter can take any values from 0 to 2880. Two bins are used to check the design response in both cases minimum and maximum values,

- R

Redundancy version is an upper layer parameter that is used in the bit selection phase to determine the location from where we start to generate the output bits. two bins are used to check the system response in the two cases Rv= 0 and Rv=1, this parameter is in control coverage group.

- Qm

Modulation index That determines the modulation order whether the transaction is a BPSK or QPSK . Two bins are used to observe the system response in the two case Qm=0 , Qm =1.

- Cross coverage

All the previous covergroups are crossed together to make sure that the modulator's exercised all possible combinations and has been tested thoroughly When the write function is invoked. The Coverage should be triggered to sample the coverage values

Assertions

One way to verify the Rate matcher is to implement a monitor as a System Verilog interface that uses a mixture of SVA concurrent assertions to observe and check the bus traffic.It's a passive verification component which monitors the block signals, it can be attached to external signals in the top level of the UVM Environment.In this module we check the value of the interface signals, if one of them is in an undefined state, then it will cause a problem under certain conditions.

There are two type of signals to be checked:

1. Signal must be known all time to ensure proper operation of the whole chain; these signals are:

- RM_reset
- RM_Enable
- RM_out_enable
- RM_end_flag

2. Signal must be known only under certain condition; these signals are:

- RM_enable_ld
- In_d0
- In_d1
- In_d2
- G
- Rv
- Qm
- RM_out

This module is also used to verify the signals sequences for all types of transfers to ensure successful communication between the Rate matcher and the other blocks in the Transmitter chain.

The sequences checked by this Assertions module are:

1. **Output length sequence.**

The Output length of the collected bits is variable according to G parameter see quation number. Therefore, RM_out_enable shall be sampled at logic 1 for a number of consecutive cycles defining a valid output.

2. **Transaction done sequence.**

end_flag indicates that the block finished the current operation and is ready for new input. end_flag must be sampled at logic one after RM_out_enable completes output length sequence successfully.

These sequences are used to verify the timing between commands.

4.2.2 FFT

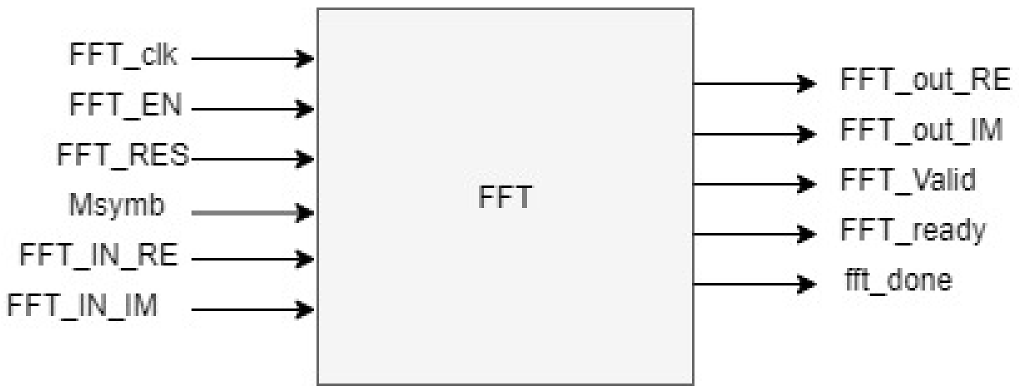


Figure 4.17: FFT input and output ports

Port name	Direction	Width	Description
Turbo_IN	input	1 bit	Input stream
FFT_clk	input	1 bit	system clock
FFT_EN	input	1 bit	Block enable
FFT$_RST$	input	1 bit	Block reset
M_{symb}	input	1 bit	To choose between 3 and 6 and 12 points FFT
FFT_IN_RE	input	16 bits	Real part of the input symbol
FFT_IN_IM	input	16 bits	Imaginary part of the input symbol
FFT_out_RE	input	16 bits	Real part of the output symbol
FFT_out_IM	input	16 bits	Imaginary part of the output symbol
FFT_Vaid	output	1 bit	the output of the FFT is valid
FFT_ready	output	1 bits	FFT is ready to receive new input

Table 4.18: FFT input and output ports

4.2.2.1 TEST PLAN

Unknown (X, Z) Signals

For FFT, the following signals must be known under certain conditions:

Signal	Condition
FFT_IN_RE	FFT_ready, FFT_RES and FFT_EN are all at logic ‘1’ in the previous cycle
FFT_IN_Im	FFT_ready, FFT_RES and FFT_EN are all at logic ‘1’ in the previous cycle
FFT_EN	Always be in a known state
FFT_RES	Always in a known state
M_{symb}	FFT_ready, FFT_RES and FFT_EN are all at logic ‘1’
FFT_ready	always be in a known state
FFT_out_RE	FFT_Valid is at logic ‘1’
FFT_out_Im	FFT_Valid is at logic ‘1’
FFT_Valid	always be in a known state
fft_done	always be in a known state

Table 4.19: FFT unkown signals

Timing Relationships

For FFT, the following temporal relationships can be defined:

Sequence	What to check
FFT output length	**FFT_Valid** = 1 / For a number of cycles according to M_{symbol}
FFT Transaction done	Initially :FFT Output length
	After 1 cycle: FFT_done = 1 / For one clock cycle
FFT new input	Initially :FFT Output length
	FFT_ready = 1 / For a number of cycles according to M_{symbol}

Table 4.20: FFT timing relationship

Functional Coverage

The fields that are relevant to FFT functional coverage are:

Specifications	cover
FFT_IN_RE	Cover the maximum and minimum values
FFT_IN_IM	Cover the maximum and minimum values
FFT_IN_RE & FFT_IN_IM	Cover all combination between their maximum and minimum values
M_{symbol}	Cover all possible numbers of subcarriers
	Cover all sequenced combination between them

Table 4.21: FFT functional coverage

4.2.2.2 Environment

Sequence item

This class encapsulates all FFT driving variables. A set constraints was defined on these variables.

Constraints

1-Msymb constraint:

- Since we have only specific values to TBS signal, we need to limit the randomization of TBS to be as follow Msymb [02]; 2'00 for 3-point FFT, 2'01 for 6-point FFT and 2'10 for 12-point FFT with equal distribution

2-Control Constraint:

- Gives distribution to the constraint solver for **FFT_EN**, 90% logic '1' and the other 10% is logic '0'
- Gives distribution to the constraint solver for **FFT_RES**, 90% logic '1' and the other 10% is logic '0'

Driver

While **FFT_EN** and **FFT_RES** are both at logic 1, the driver drives the inputs **FFT_IN_RE** and **FFT_IN_IM** to the interface and change it every clock cycle according to the radix of FFT required (i.e. Msymb). Since the block is not pipelined. therefore, the driver goes to idle state waiting for **FFT_ready** signal to be sampled at logic 1 in order to get the next input.

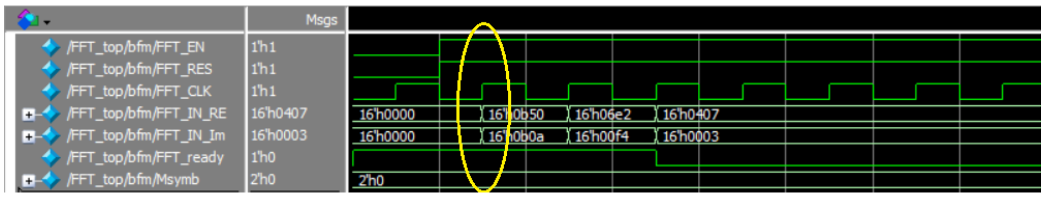

Msgs
/FFT_top/bfm/FFT_EN 1'h1
/FFT_top/bfm/FFT_RES 1'h1
/FFT_top/bfm/FFT_CLK 1'h1
/FFT_top/bfm/FFT_IN_RE 16'h0407
/FFT_top/bfm/FFT_IN_Im 16'h0003
/FFT_top/bfm/FFT_ready 1'h0
/FFT_top/bfm/Msymb 2'h0
16'h0000 16'h0b50 16'h06e2 16'h0407
16'h0000 16'h0b0a 16'h00f4 16'h0003
2'h0

Figure 4.18: FFT driving

Input Monitor

Once **FFT_EN** and **FFT_RES** are both at logic 1, the monitor samples Msymb value and enters a loop of sampling **FFT_IN_RE** and **FFT_IN_IM** for number of clock cycles according to Msymb. The sampling takes place at the positive clock edge. Once the sampling loop is completed successfully, the monitor sends the sequence item to the coverage and scoreboard components via the TLM Analysis port. The monitor enters an ideal state waiting for **FFT_ready** to go to logic 1 again which indicates a new transaction.

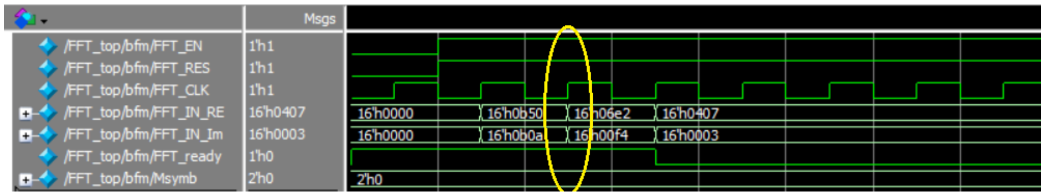

Msgs
/FFT_top/bfm/FFT_EN 1'h1
/FFT_top/bfm/FFT_RES 1'h1
/FFT_top/bfm/FFT_CLK 1'h1
/FFT_top/bfm/FFT_IN_RE 16'h0407
/FFT_top/bfm/FFT_IN_Im 16'h0003
/FFT_top/bfm/FFT_ready 1'h0
/FFT_top/bfm/Msymb 2'h0
16'h0000 16'h0b50 16'h06e2 16'h0407
16'h0000 16'h0b0a 16'h00f4 16'h0003
2'h0

Figure 4.19: FFT sampling

Output Monitor

The output monitor enters a waiting state until **FFT_Valid** is sampled at logic 1. Once this event happened, Monitor samples Msymb of the interface and enter a loop sampling the output of the FFT (i.e. **FFT_out_RE** and **FFT_out_Im**) for 3, 6 or 12 clock cycles depending on Msymb. The monitor continues in the previous operation until **FFT_Valid** is sampled at logic 0 which mean the FFT output has ended. The monitor sends the output transaction to the scoreboard component via the TLM Analysis port

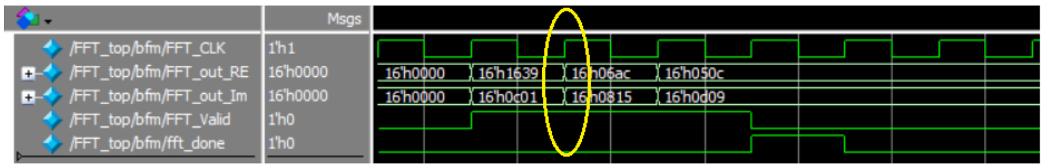

Msgs
/FFT_top/bfm/FFT_CLK 1'h1
/FFT_top/bfm/FFT_out_RE 16'h0000
/FFT_top/bfm/FFT_out_Im 16'h0000
/FFT_top/bfm/FFT_Valid 1'h0
/FFT_top/bfm/fft_done 1'h0
16'h0000 16'h1639 16'h06ac 16'h050c
16'h0000 16'h0c01 16'h0815 16'h0d09

Figure 4.20: FFT output

Scoreboard

The implementation of the golden model for the FFT is done by converting the input stream into real data type then implementing the butterfly algorithm for 3,6,12 FFT that consists of multiplication and additions performed on the real representation. Afterwards, the DUT output is converted into real representation and the error of transformation is calculated between them which should satisfy a required accuracy. The scoreboard consists of:

- Fix2dec function which covert the randomized input from the stimulus to real data value type.
- structure to encapsulate the input to the FFT functions and another structure to encapsulate the output.
- FFT-3 function which implement the butter of radix-3 FFT as shown in the figure see4.21
- FFT-6 function which implement the butter of radix-6 FFT as shown in the figure see 4.22
- FFT-12 function which implement the butter of radix-12 FFT as shown in the figure see 4.23
- The predict function of the scoreboard call the required FFT function depending on M_{symbol} and the output of the DUT is convert to real data type using the transformation function that was discussed earlier.
- The difference between the DUT real output and the FFT function output is checked to be compatible with the desired accuracy

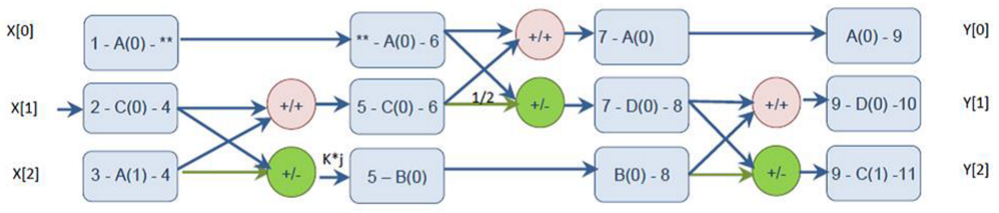


Figure 4.21: FFT 3-point SFG

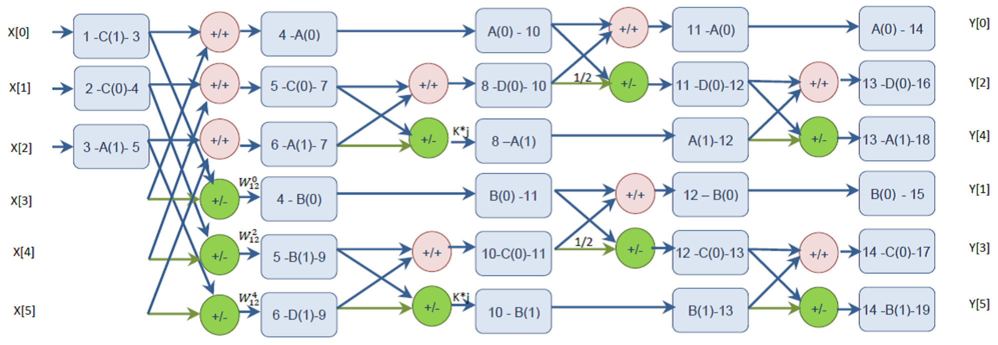


Figure 4.22: FFT 6-point SFG

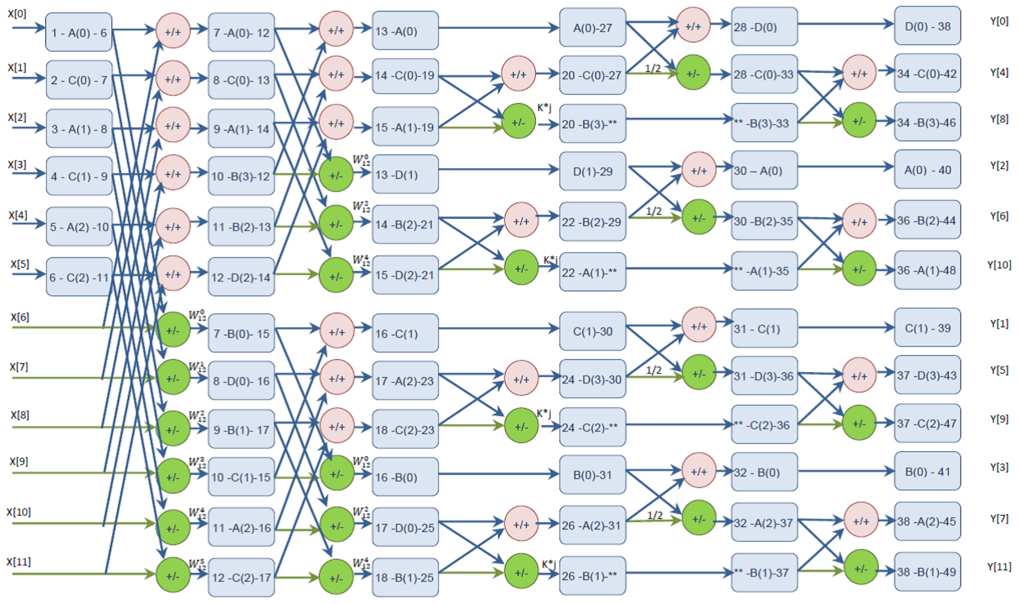


Figure 4.23: FFT 12-point SFG

- The predict function of the scoreboard call the required FFT function depending on Msymb and the output of the DUT is convert to real data type using the transformation function that was discussed earlier.
- The difference between the DUT real output and the FFT function output is checked to be compatible with the desired accuracy

Coverage

We've defined our functional coverage model in terms of stimulus in (test plan chapter). FFT is fully tested if we've run a complete set of these terms:

- FFT_IN_RE & FFT_IN_IM

  They are the real and imaginary terms of the Input signal to the FFT block which the operation of butterfly should be applied on, the 16-bit of each is divided in the design into 6-bit decimal and 10-bit fraction. Since the maximum and the minimum value of the real and the imaginary input are considered as a corner values, they should be tested to ensure that the butterfly of FFT work properly.

  A covergroup is used to capture functional coverage for Input bus. Two coverpoints are used for the two input signals where two coverage bins are defined for each value (maximum and minimum) in each coverpoint.

- Msymb

  It defines the number of subcarriers to be allocated in the frequency domain which Consequently defines the number of the FFT point to be performed or the butterfly to be used whether 3 ,6 or 12 point FFT.

  Therefore, all the FFT points shall be tested separately to ensure the properly operation of each butterfly.

  A covergroup is used to capture functional coverage for Msymb bus. A coverpoint is used where three coverage bins are defined for each value (0 for three subcarriers, 1 for six subcarriers and 2 for 12 subcarriers).

Since the block doesn't require a reset between each operation, a sequenced combination between each two, different FFT points, butterfly operations has to be verified.

A Coverpoint transition bins in the same covergroup is used to check all the possible sequence of the Msymb values.

Assertions

One way to verify the FFT is to implement a monitor as a System Verilog interface that uses a mixture of SVA concurrent assertions to observe and check the bus traffic. It's a passive verification component which monitors the block signals, it can be attached to external signals in the top level of the UVM Environment. In this module we check the value of the interface signals, if one of them is in an undefined state, then it will cause a problem under certain conditions. There are two type of signals to be checked:

1. Signal must be known all time to ensure proper operation of the whole chain; these signals are:

- FFT_EN
- FFT_RES
- FFT_RES
- FFT_ready
- FFT_Valid
- fft_done

2. Signal must be known only under certain condition; these signals are:

- FFT_IN_RE
- FFT_IN_Im
- Msymb
- FFT_out_RE

- FFT_out_Im

This module is also used to verify the signals sequences for all types of transfers to ensure successful communication between FFT block and the other blocks in the Transmitter chain.

The sequences checked by this Assertions module are:

1. **FFT output length sequence.**

The output length of block is variable according to the number of subcarriers that would be allocated. It can be either 3, 6 or 12 and the output is generated in consecutive cycles. **FFT_Vaild** indicates that the generated output is valid and ready to be sampled. It must be held at logic '1' for a number of consecutives cycles depending on the value of **Msymb**

2. **FFT transaction done sequence**.

FFT_done indicates that the block finished the current operation to avoid any unwanted data and it is ready for the new one. Once the last output is sampled, **FFT_done** must be sampled at logic '1' for one cycle at the next positive edge

3. **FFT new input sequence.**

FFT_ready indicates that the FFT block is ready for operation and ready to take input. Since the FFT take input serially as one input per cycle, the ready signal has to be stable at logic '1' as long as the input is being feed to the FFT block. when **FFT_ready** is sampled at logic '1', the previous block puts the data on the input bus in order to be sample at the next positive edge. Once the last output is sampled, **FFT_ready** must be sampled at logic '1' at the next positive edge. Basically, **FFT_ready** has to be stable for either 3 ,6 or 12 clock cycles depending on the number of subcarrier that would be allocated (i.e. M_{symb}).

These sequences are used to verify the timing between commands.

4.2.3 IFFT

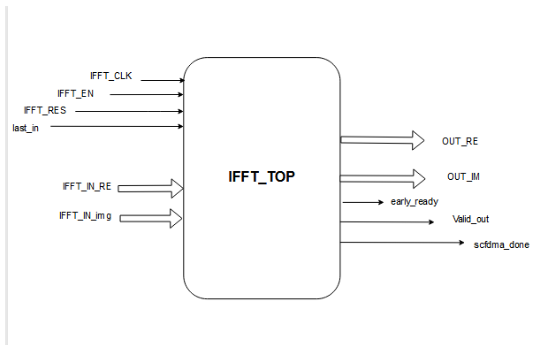


Figure 4.24: IFFT input and output ports

Port name	Direction	Width	Description
IFFT_clk	input	1 bit	system clock
IFFT_EN	input	1 bit	Block enable
IFFT_RST	input	1 bit	Block reset
Last_in	input	16 bits	Igeneral control unit raised to 1 when the last SC-FDMA symbol is transferred from resource mapper to the IFFT block.
IFFT_IN_RE	input	16 bits	Real part of the output symbol
IFFT_IN_IM	input	16 bits	imaginary part of the output symbol
out_RE	output	16 bits	Real part of the output symbol
out_IM	output	16 bits	imaginary part of the output symbol
Vaid_out	output	1 bit	general control unit raised to 1 from cycles 54 to 11 at each IFFT iteration to enable the output of resource element mapper to enter the IFFT block from cycles 1 to 12.
early_ready	output	1 bits	general control unit raised to 1 from cycles 54 to 11 at each IFFT iteration to enable the output of resource element mapper to enter the IFFT block from cycles 1 to 12.
Sc_fdma done	output	1 bit	general control unit raised

Table 4.22: IFFT input and output ports

4.2.3.1 TEST PLAN

Unknown (X, Z) Signals

For IFFT, the following signals must be known under certain conditions:

Signal	Condition
IFFT_IN_RE	early_ready, IFFT_RES and IFFT_EN are all at logic '1' in the previous cycle
IFFT_IN_IM	early_ready, IFFT_RES and IFFT_EN are all at logic '1' in the previous cycle
IFFT_EN	Always be in a known state
IFFT_RES	Always in a known state
Last_in	always be in a known state
early_ready	always be in a known state
Valid_out	always be in a known state
Sc_fdma_done	always be in a known state
OUT_RE	Valid_out is at logic '1'
OUT_IM	Valid_out is at logic '1'

Table 4.23: IFFT unkown signals

Timing Relationships

For IFFT, the following temporal relationships can be defined:

Sequence	What to check
Output length	Initially: early_ready = 1
	After 54 cycles: Valid_out = 1 / For 54 cycles
Data Transaction done	Initially: last_in = 1
	After 108 cycle: scfdma_done = 1

Table 4.24: Rate Matcher timing temporal relationship

4.2.3.2 Environment

Sequence item

This class encapsulates all IFFT driving variables. A set constraints was defined on these variables.

Constraints

1-Control Constraint:

- Gives distribution to the constraint solver for **IFFT_EN**, 90% logic ‘1’ and the other 10% is logic ‘0’
- Gives distribution to the constraint solver for **IFFT_RES**, 90% logic ‘1’ and the other 10% is logic ‘0’
- Gives distribution to the constraint solver for **Last_in**, 90% logic ‘1’ and the other 10% is logic ‘0’

Driver

The driver gets the sequence item from the sequencer. While **IFFT_RES, IFFT_EN** and **early_ready** are all at logic ‘1’, the driver starts to send the input in 12 consecutive cycles. The designed block is not pipelined. Therefore, the driver goes to a sleep state waiting for early_ready signal to be sampled at logic ‘1’ in order to get the next sequence item.

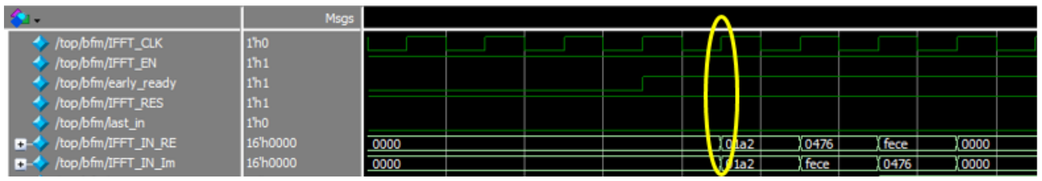


Figure 4.25: IFFT driving

Input Monitor

Once **IFFT_RES** and **IFFT_EN** are both at logic ‘1’, the monitor enters a loop of sampling **IFFT_IN_RE** and **IFFT_IN_IM** (i.e. Frequency domain symbols) for 12 clock cycles. The sampling takes place at the positive clock edge. Once the sampling loop is completed successfully, the monitor broadcast the transaction to the coverage and scoreboard components via the TLM Analysis port. The monitor enters an ideal state waiting for **early_ready** to go to logic '1' again which indicates a new transaction.

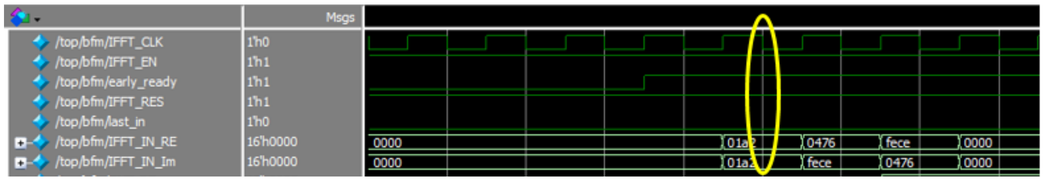


Figure 4.26: IFFT sampling

Output Monitor

The output monitor enters a waiting state until **Valid_out** is high. Once this event is triggered, **OUT_RE** and **OUT_IM** will be sampled at the same positive edge. each symbol is fixed for 3 clock cycles so the output sampling occurs every 3 clock cycles. the sampling lasts for 54 clock cycles i.e. 18 symbols (16 output symbols and 2 cyclic prefix). The output monitor sends the output transaction to the scoreboard component via the TLM Analysis port

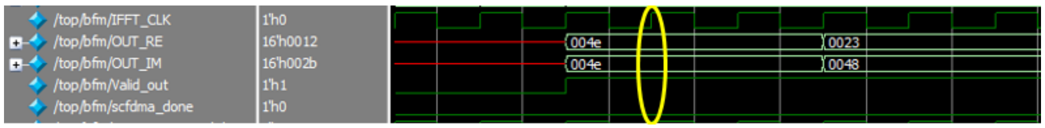


Figure 4.27: IFFT output

Scoreboard

16 point IFFT used to cover 12 subcarriers with spacing of 15 KHz between each adjacent sub channels. The scoreboard is implemented based on butter fly unit as shown in figure see 3.10

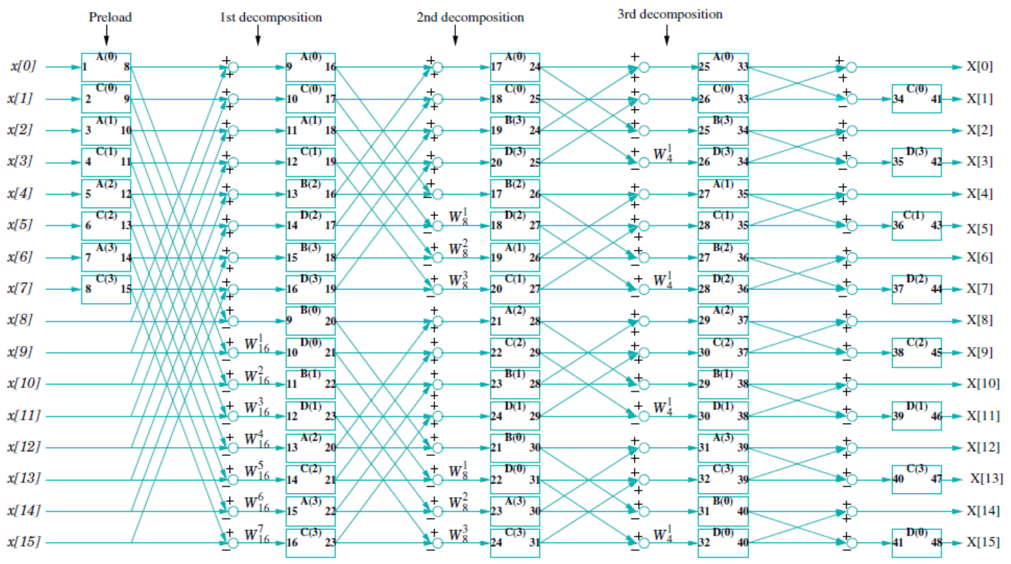


Figure 4.28: IFFT 16-point SFG

Predict result function is responsible for predicting the output, which will be compared with the DUT output.

- Twiddle values has generated using MATLAB then used as a look up table in the scoreboard.
- As the prediction model operates at real representation, first we convert the binary input to real type using fix2dec function.
- The last four bits is padded with zeros as we have 12 subcarriers and the IFFT is 16 points
- The model consists of four stages according to figure each stage is implemented using for loop.
- The output is converted to binary type using dec2fix function and assigned in **OUT_RE** and **OUT_IM.**
- Finally, cyclic prefix is inserted. transformation function that was discussed earlier.

Assertions

One way to verify the IFFT is to implement a monitor as a System Verilog interface that uses a mixture of SVA concurrent assertions to observe and check the bus traffic. It's a passive verification component which monitors the protocol signals, it can be attached to external signals in the top level of the UVM Environment. In this module we check the value of the interface signals, if one of them is in an undefined state, then it will cause a problem under certain conditions.

There are two type of signals to be checked:
1. Signal must be known all time to ensure proper operation of the whole chain; these signals are:

- IFFT_EN
- IFFT_RES
- Last_in
- early_ready
- Valid_out
- Sc_fdma_done

2. Signal must be known only under certain condition; these signals are:

- IFFT_IN_RE
- IFFT_IN_IM
- Msymb
- OUT_RE
- OUT_IM

This module is also used to verify the signals sequences for all types of transfers to ensure successful communication between the IFFT and the other blocks in the Transmitter chain.

The sequences checked by this Assertions module are:

1. Output length sequence.

Valid_out goes to logic '1' after the first iteration of IFFT which takes 54 clock cycle after early_ready being sampled at logic '1'. It should be stable for 54 clock cycles (18 symbols each symbol is fixed 3 clock cycles).

2. Data transmission done sequence

sc_fdma_done goes to logic '1' when the final SC-FDMA symbol is completely transmitted which means after asserting Last_in by 108 clock cycles (54 clock cycle latency and 18 symbols each symbol is fixed 3 clock cycles).

These sequences are used to verify the timing between commands.

Chapter 5

Top level testing

5.1 Specifications extraction

Port name	Direction	Width	Description
System_rst	input	1 bit	Reset the whole block
System_clk	input	1 bit	Operating clock
System_input	input	1 bit	Serial input bitstream from the upper layer
TBS_Start	input	1 bit	Control signal from the upper layer to indicate the beginning of the transmission
TBS	input	12 bits	Transport Block size
G	input	12 bits	indicate the output sequence length for the rate matcher
Rv	input	1 bit	Reduncdancy version
Qm	input	1 bit	Modulation index
Isc	input	5 bits	indicate number of subcarriers and their start index for REM
Irep	input	3 bits	indicate number of repetitions to REM
N_slots	input	1 bit	Number of time slots
Msc	output	1 bit	Number of subcarriers
Scramb_nf	input	1 bit	LSB of system frame number
Scramb_RNTI	input	16 bit	Radio Network Temporary identifier
Scramb_Ncell_ID	input	9 bits	Index of cell identity
Scramb_ns	input	4 bits	First slot of transmission codeword
Out_real	output	16 bits	Real part of output symbol, 6 bits for integer part and 10 bits for fraction part
Out_imag	output	16 bits	imaginary part of output symbol, 6 bits for integer part and 10 bits for fraction part
Valid_out	output	1 bit	output symbol is valid for transmission or not
busy_signal	output	1 bit	the design is processing a packet to prevent receiving of another packett
init_shift_done	output	1 bit	to indicate that the scrambler finished its initial shifting operation and ready to operate.

Table 5.1: Chain input and output ports

5.2 Test plan

5.2.1 Unknown (X, Z) Signals

For the Chain the following signals must be known under certain conditions:

Signal	Condition
System_rst	Always in a known state
TBS_Start	System_RST at logic '1' indicate the beginning of the transmission
TBS	During a single transmission
G	During a single transmission
Rv	During a single transmission
Qm	During a single transmission
Isc	During a single transmission
Irep	During a single transmission
N$_{s}lots$	During a single transmission
Scramb_nf	During a single transmission
Scramb$_{R}NTI$	During a single transmission
Scramb$_{R}NTI$	During a single transmission
Scramb_Ncell_ID	During a single transmission
Scramb_ns	During a single transmission
Out_real	Valid_out at logic '1'
Out_imag	Valid$_{o}utatlogic'1'$
Valid_out	Always in a known state

Table 5.2: Chain input and output ports

5.2.2 Timing Relationships

The timing relationships between the signals in the protocol can be described using sequences and properties. For the top level chain, the following temporal relationships can be defined:

Sequence	What to check
	Valid_out = 0

Reset Effect	Out_real = 0
	Out_imag= 0

Signal stability	Initially : TBS$_start = 1$ for one clock cycle Upper layer parameters are stable until busy signal becomes low

Table 5.3: Chain temporal timing relationships

5.3 Environment

5.3.1 Sequence item

This class encapsulates all top level driving variables. A set constraints was defined on these variables.

Constraints

1-**Transport block size Constraint:**

- Since we have only specific values to TBS signal, we need to limit the randomization of TBS to be as follow TBS 3.1

2- **G value constraint**

- The maximum value for G is 2880 therefor the size is [11:0] 12 bits.

3- **Redundancy version constraint**

- Gives distribution to the constraint solver for Rv, 90% logic '0' and the other 10% is logic '1' for retransmissions cases

4- **Modulation index constraint**:

- There are 2 values for the modulation index BPSK and QPSK, the modulation index is randomized with equal distribution for these values Qm, 50% logic '1' and the other 50% is logic '0'

5- **Msc Constraint**:

- There are 3 values for the number of subcarriers, 3, 6 and 12 subcarriers. It's randomized with equal distribution for these values signal randomized with equal distribution for these values Msc, '00', '01'and '10';

6-**Number of subcarriers Constraint**:

- Since we have only specific values to Isc signal, we need to limit the randomization of Isc to be as follow I_sc $\epsilon[0:18]$ as specified in 3.7

7- **Number of slots Constraint**:

- Since we have only specific values to N_slots signal , we need to limit the randomization of N_slots to be as follow $N\_slots\epsilon[1:16]$

7- **Number of slot Constraint**:

- Since we have only specific values to $Scramb\_ns$ signal, we need to limit the randomization of $Scramb\_ns$ to be as follow $Scramb\_ns\epsilon[0:9]$

5.3.2 Driver

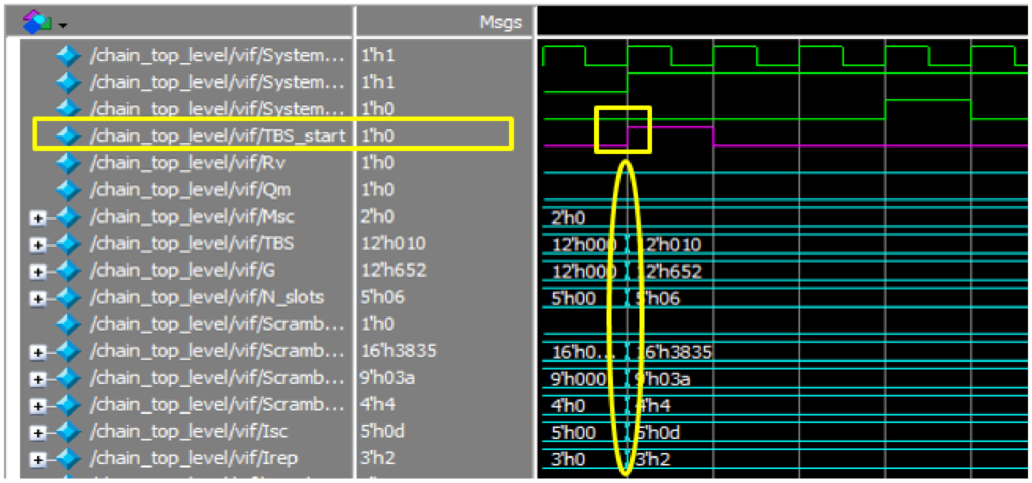
Driving the input to the chain is supposed to be according to the busy_signal but this signal is not working properly , then the input driving in the environment needs to be accurately calculated according to the output length for each transaction to keep driving after the previous transaction is completed. Since The output of the block interleaver is the bit sequence read out column by column from the

$$(R_{max} * C_{max}) \text{ where:}$$
$$G' = \frac{G}{Q_m}$$
$$E = Qm * \lceil G' \rceil$$
$$H' = \lceil E * Q_m \rceil$$
$$C_{max} = (N_{symbol}^{UL} - 1) * N_{slot}^{UL}$$
$$R_{max} = \left\lceil \frac{H'}{C_{max}} \right\rceil$$

Data path through modulator and fft without changing in its size and when enter the REM and IFFT the size is changed according to :

$$N_{slot}(R_{max} * C_{max})/N_{sc}$$
Number of N_{Ru}=$\frac{N_slots}{14}$; N_slots % 14 =0
output data length Number of N_{Ru} $*N\_slots * 14 * 18$
Number of N_{Ru} $=\frac{N\_slots}{14}$; $N\_slots\%14 \neq 0$
output data length Number of $N_{Ru} * N\_slots * 14 * 18 + (N_slots\%14 = 0) * 18$

Then the TBS_start signal is set to '0' , then the control signals and the input stream is derived to the interface followed by 24 bit '0's for the CRC ,once the valid_out signal has a posedge a counter is incremented and compared to the output calculated length so as not to drive new transaction until the operating one is completed

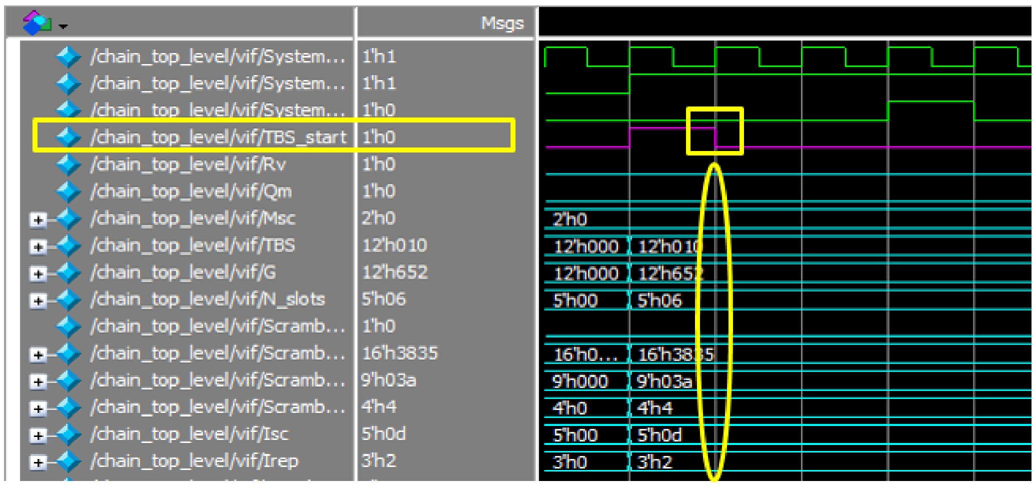


	Msgs
/chain_top_level/vif/System...	1'h1
/chain_top_level/vif/System...	1'h1
/chain_top_level/vif/System...	1'h0
/chain_top_level/vif/TBS_start	1'h0
/chain_top_level/vif/Rv	1'h0
/chain_top_level/vif/Qm	1'h0
/chain_top_level/vif/Msc	2'h0
/chain_top_level/vif/TBS	12'h010
/chain_top_level/vif/G	12'h652
/chain_top_level/vif/N_slots	5'h06
/chain_top_level/vif/Scramb...	1'h0
/chain_top_level/vif/Scramb...	16'h3835
/chain_top_level/vif/Scramb...	9'h03a
/chain_top_level/vif/Scramb...	4'h4
/chain_top_level/vif/Isc	5'h0d
/chain_top_level/vif/Irep	3'h2

Figure 5.1: Chain Input driving

5.3.3 Input monitor

Once TBS_start is sampled at logic 1, the monitor starts with sampling the control signals then it enters a loop of sampling System_input for (TBS) consecutive clock cycles. The sampling takes place at the positive edge. Once the sampling loop is completed successfully, the monitor broadcast the transaction to the coverage and scoreboard components via the TLM Analysis port. The monitor waits for TBS_start to be sampled at logic 1 which indicates a new transaction.



```
                                      Msgs
/chain_top_level/vif/System...        1'h1
/chain_top_level/vif/System...        1'h1
/chain_top_level/vif/System...        1'h0
/chain_top_level/vif/TBS_start        1'h0
/chain_top_level/vif/Rv               1'h0
/chain_top_level/vif/Qm               1'h0
/chain_top_level/vif/Msc              2'h0
/chain_top_level/vif/TBS              12'h010
/chain_top_level/vif/G                12'h652
/chain_top_level/vif/N_slots          5'h06
/chain_top_level/vif/Scramb...        1'h0
/chain_top_level/vif/Scramb...        16'h3835
/chain_top_level/vif/Scramb...        9'h03a
/chain_top_level/vif/Scramb...        4'h4
/chain_top_level/vif/Isc              5'h0d
/chain_top_level/vif/Irep             3'h2
```

Figure 5.2: Chain Input Sampling

5.3.4 Output monitor

As a result of the output being generated in consecutive clock cycles, the output monitor enters a waiting state until the latency of chain then Valid_out is sampled at logic 1. Once this event is triggered, Out_real and Out_imag will be sampled at the same positive edge.
The monitor continues in the previous operation until the length of the output according to equation 1 which means the transaction is completed. Then the output monitor sends the output transaction to the scoreboard component via the TLM Analysis port.

5.3.5 Scoreboard

The top level scoreboard is required to predict the output of the whole chain, so it was decided to reuse the existing models in the block level testing environments. Since an environment was developed for each of the six blocks discussed in block level testing, modeling the three remaining blocks is needed.

- CRC model:

  The CRC is an error detection technique used to calculate parity bits that are added to the transmitted data. It is implemented using LFSR, the polynomial of CRC block has been modeled using loop of input size (TBS) iterations, the input concatenated by 24 zeros then xoring and shifting is done for 24 bits then the output transmitted to Turbo Encoder block with (24+TBS) output length.

- Data Multiplexing and Channel interleaver model:

  Data multiplexing ensures control and data information are mapped to different modulation symbols, channel interleaver interleave the data to avoid burst error. -Multiplexing the data is done depending on the Q_M signal as shown in the following figure see 3.6 3.7.

  Interleaving of the data is done by reading the multiplexed data column by column.

- Scrambler model:

  The scrambler is used to randomize input bit stream using two LFSRs in parallel, it is modeled using a for loop of input size which mimics the functionality of the LFSRs. Modifications to the original models were needed so they operate correctly when handing data to each other.

- Modulator modification:

  The original model of the modulator operates on a single input symbol at a time, the modified model receives the whole input stream and then performs its operation repeatedly on each symbol.

- FFT modification:

  Same as Modulator, instead of operating on a specific number of input symbols according to the number of subcarriers. The model receives the whole symbols, divides them into groups according to the number of subcarriers and performs its operation repeatedly.

- REM modification:

  In REM model the input is received in three dimensional arrays using three nested loops of input size (FFT output size) iterations to operate on single RU, number of slots iterations and number of subcarriers iterations.

  The REM outputs are formed as two-dimensional arrays to deliver the proper IFFT input using three nested loops of number of repetitions iterations, number of slots iterations and twelve subcarriers iterations.

- IFFT modification:

  Same as FFT, In the IFFT model the input block is divided in segments of twelve using two nested loops.

  The whole model was implemented by creating a class for each block's model, each model contains a function that predicts the output and parameters necessary for this function. For these classes to communicate with each other, another class was used to encapsulate the outputs for each model.

  Scoreboard prediction function receives the sequence item of the chain then it instantiates object of each class of the chain model, instantiates objects of the encapsulated output of each block and then calls the predict function of each object created which performs the prediction of golden model of each block.

5.3.6 Coverage

We've defined our functional coverage model in terms of stimulus in (test plan chapter). A covergroup is used to capture functional coverage for control signals. The top chain is fully tested if we've run a complete set of these terms:

- TBS

Transport block size is nothing but the payload for physical layer. All of the values shown in Table (lookup) should be check to fully test the internal interleaver. There is a defined coverage bin for each value of TBS in the coverage model. When the write function is invoked. The Coverage should be triggered to sample the coverage values.

- G

an upper layer parameter that is used to calculate the required output sequence. this parameter can take any values from 0 to 2880. Two bins are used to check the design response in both cases minimum and maximum values,

- R

Redundancy version is an upper layer parameter that is used in the bit selection phase to determine the location from where we start to generate the output bits. two bins are used to check the system response in the two cases Rv= 0 and Rv=1, this parameter is in control coverage group.

- Qm

Modulation index That determines the modulation order whether the transaction is a BPSK or QPSK . Two bins are used to observe the system response in the two case Qm=0 , Qm =1.

- Scheme

This covergroup is used to cover both modulation schemes, BPSK and QPSK, through the input

- BPSK

This covergroup has 2 coverpoints, one for the input symbol (Mod_IN), and another for the number of subcarrier (Msc). It also has cross coverage between these two coverpoints

- QPSK

IThis Covergroup has the same coverpoints as BPSK, the difference is that in case of BPSK, the input symbol is 1 bit, while in case of QPSK the input symbol is two bits, so this covergoup samples two consecutive bits driven to the input (Mod_IN) and covers all the cases for the input symbol. The coverpoint for Msc and cross coverage are the same as BPSK. The combination of these 3 covergroups insures the Modulator has been tested thoroughly.

- Isc

Upper layer parameter which determines the number of subcarrier and the starting index of subcarriers. All of the values vary from 0 to 18 should be check to fully test all possible register access which indicates all possible subcarriers. A covergroup is used to capture functional coverage for Isc bus. There is a defined coverage bin for each value of Isc in the coverage model.

- Irep

Upper layer parameter which determines the number of repeated blocks of data. All of the values vary from 0 to 7 should be check to fully test all possible register access which indicates all possible repetition. A covergroup is used to capture functional coverage for Irep bus. There is a defined coverage bin for each value of Irep in the coverage model.

- FFT_IN_RE FFT_IN_IM

They are the real and imaginary terms of the Input signal to the FFT block which the operation of butterfly should be applied on, the 16-bit of each is divided in the design into 6-bit decimal and 10-bit fraction. Since the maximum and the minimum value of the real and the imaginary input are considered as a corner values, they should be tested to ensure that the butterfly of FFT work properly. A covergroup is used to capture functional coverage for Input bus. Two coverpoints are used for the two input signals where two coverage bins are defined for each value (maximum and minimum) in each coverpoint.

- Msymb

It defines the number of subcarriers to be allocated in the frequency domain which Consequently defines the number of the FFT point to be performed or the butterfly to be used whether 3 ,6 or 12 point FFT. Therefore, all the FFT points shall be tested separately to ensure the properly operation of each butterfly. A covergroup is used to capture functional coverage for Msymb bus. A coverpoint is used where three coverage bins are defined for each value (0 for three subcarriers, 1 for six subcarriers and 2 for 12 subcarriers) Since the block doesn't require a reset between each operation, a sequenced combination between each two, different FFT points, butterfly operations has to be verified A Coverpoint transition bins in the same covergroup is used to check all the possible sequence of the Msymb values.

5.3.7 Assertions

In this module we check the value of the interface signals, if one of them is in an undefined state, then it will cause a problem under certain conditions. There are two type of signals to be checked:

1. Signal must be known all time to ensure proper operation of the whole chain; these signals are:

- System_RST
- Valid_Out

2. Signal must be known only under certain condition; these signals are:

- TBS
- Qm
- Rv
- Msc
- G
- N_slots
- Scramb_nf
- Scramb_ns
- Scramb_RNTI
- Scramb_Ncell_ID
- Isc
- Irep

This module is also used to verify the signals sequences for all types of transfers The sequences checked by this Assertions module are:

1. Reset effect sequence:

When reset is at logic ‘0’, all the outputs of the top level chain must be sampled at logic ‘0’

2. Signal stability

When TBS_start is raised to logic ‘1’ for one clock cycle and until the end of the transmission, which is indicated by the busy$_s ignalbecominglogic‘0'$

Chapter 6

Results and conclusion

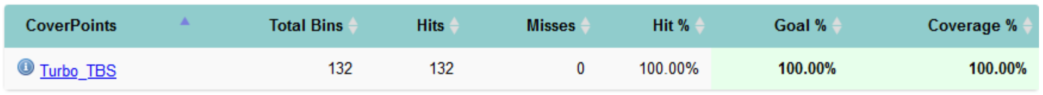
6.1 Block level testing

6.1.1 Turbo Encoder results

These results are obtained by running all tests and merging the coverage results

6.1.1.1 Functional coverage

Cover groups results are shown see 6.18 6.20 6.19

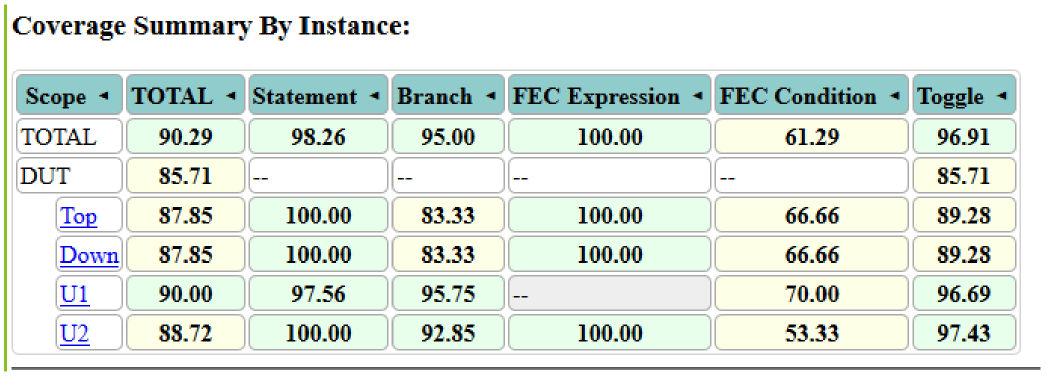


CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
Turbo_TBS	132	132	0	100.00%	100.00%	100.00%

Figure 6.1: Turbo Encoder coverage group control

6.1.1.2 Code Coverage

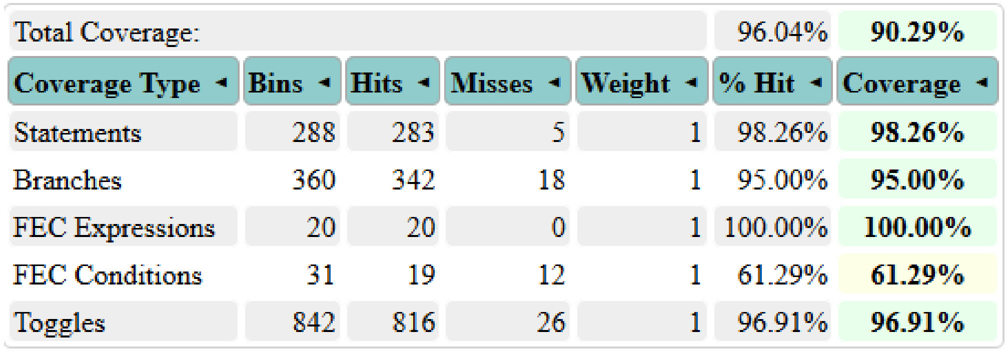
code coverage results are as shown. see 6.20



Coverage Summary By Instance:

Scope ◂	TOTAL ◂	Statement ◂	Branch ◂	FEC Expression ◂	FEC Condition ◂	Toggle ◂
TOTAL	90.29	98.26	95.00	100.00	61.29	96.91
DUT	85.71	--	--	--	--	85.71
Top	87.85	100.00	83.33	100.00	66.66	89.28
Down	87.85	100.00	83.33	100.00	66.66	89.28
U1	90.00	97.56	95.75	--	70.00	96.69
U2	88.72	100.00	92.85	100.00	53.33	97.43

Figure 6.2: Turbo Encoder results By Instance

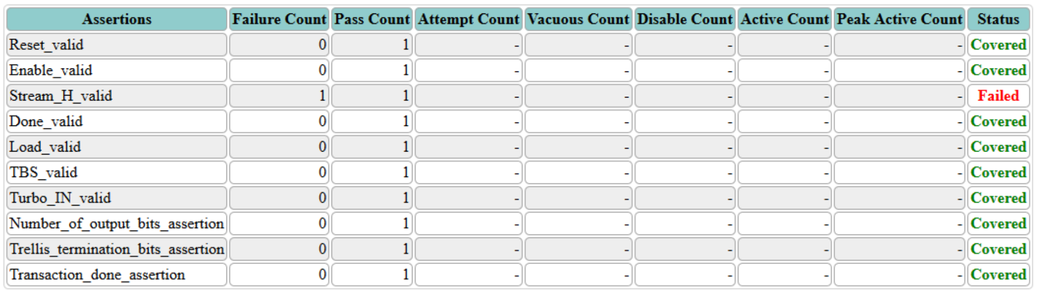


Total Coverage:					96.04%	90.29%
Coverage Type ◂	**Bins ◂**	**Hits ◂**	**Misses ◂**	**Weight ◂**	**% Hit ◂**	**Coverage ◂**
Statements	288	283	5	1	98.26%	98.26%
Branches	360	342	18	1	95.00%	95.00%
FEC Expressions	20	20	0	1	100.00%	100.00%
FEC Conditions	31	19	12	1	61.29%	61.29%
Toggles	842	816	26	1	96.91%	96.91%

Figure 6.3: Turbo Encoder results Recursive Hierarchical Coverage Details

6.1.1.3 Assertions

All the assertions passed except for Stream_H_valid as shown see 6.23



Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
Reset_valid	0	1	-	-	-	-	-	Covered
Enable_valid	0	1	-	-	-	-	-	Covered
Stream_H_valid	1	1	-	-	-	-	-	Failed
Done_valid	0	1	-	-	-	-	-	Covered
Load_valid	0	1	-	-	-	-	-	Covered
TBS_valid	0	1	-	-	-	-	-	Covered
Turbo_IN_valid	0	1	-	-	-	-	-	Covered
Number_of_output_bits_assertion	0	1	-	-	-	-	-	Covered
Trellis_termination_bits_assertion	0	1	-	-	-	-	-	Covered
Transaction_done_assertion	0	1	-	-	-	-	-	Covered

Figure 6.4: Turbo Encoder Assertion Coverage Report

6.1.1.4 Turbo Encoder Bugs

Output is incorrect in some TBS values This bug was found by the environment, the internal interleaver fails at multiple values of the transport block size i.e. TBS The its output is incorrect. These values are: 504, 520, 1064, 968, 2088, 2152, 2280, 2216, 2408, 2472, 2344, 2536. Consequently, the Turbo generates incorrect bits for Turbo_Z_K_P and incorrect trellis termination bits according to the equations

$d_{k+2}{}^{(0)} = x'_k, d_{k+3}{}^{(0)} = z'_{k+1}$

$d_{k+2}{}^{(1)} = z'_k, d_{k+3}{}^{(1)} = x'_{k+2}$

$d_{k+2}{}^{(2)} = x'_{k+1}, d_{k+3}{}^{(2)} = z'_{k+2}$

Turbo_Stream_H flag goes in undefined state This bug was found by Assertions module, Turbo_Stream_H flag goes in undefined state for one cycle in the beginning of the simulation in the beginning of the simulation. Turbo_Stream_H indicates that this output bit is valid in order for the Next block to sample it. As discussed in chapter elplan, this flag cannot be unknown.

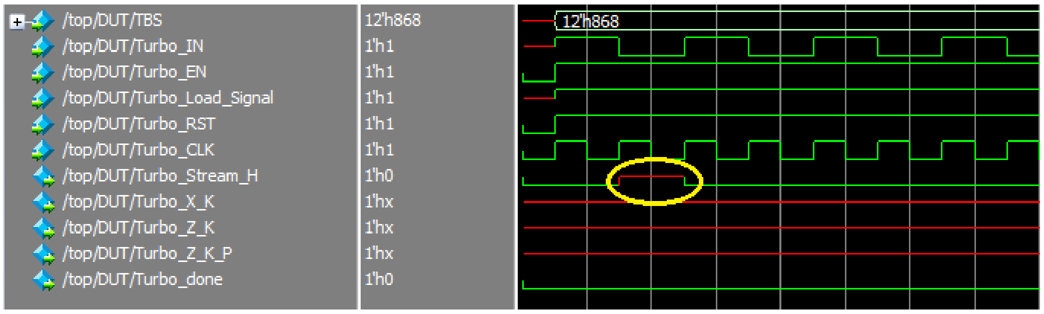


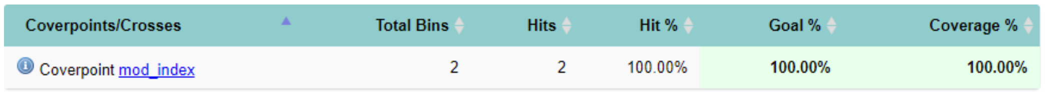
Figure 6.5: Incorrect state for Turbo_Stream_H flag

6.1.2 Modulator results

These results are obtained by running all tests and merging the coverage results

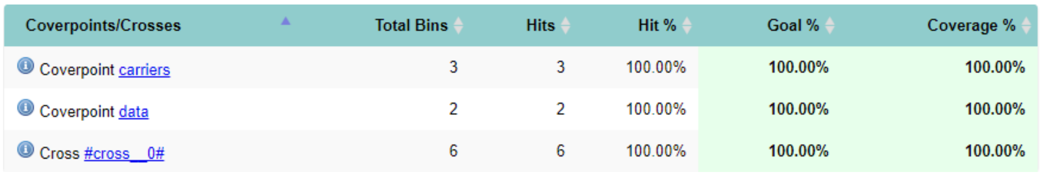
6.1.2.1 Functional coverage

Cover groups results are shown see **??** 6.7 6.8



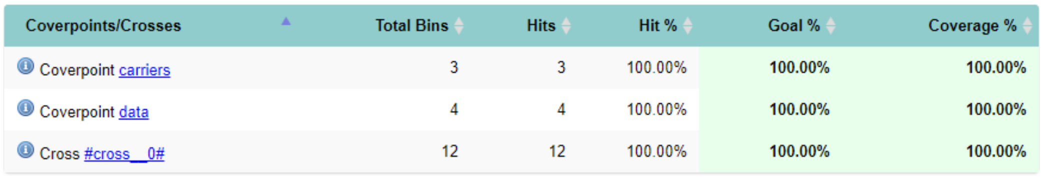
Coverpoints/Crosses	Total Bins	Hits	Hit %	Goal %	Coverage %
Coverpoint mod_index	2	2	100.00%	100.00%	100.00%

Figure 6.6: Modultaor Scheme covergroup



Coverpoints/Crosses	Total Bins	Hits	Hit %	Goal %	Coverage %
Coverpoint carriers	3	3	100.00%	100.00%	100.00%
Coverpoint data	2	2	100.00%	100.00%	100.00%
Cross #cross__0#	6	6	100.00%	100.00%	100.00%

Figure 6.7: Modulator BPSK covergroup

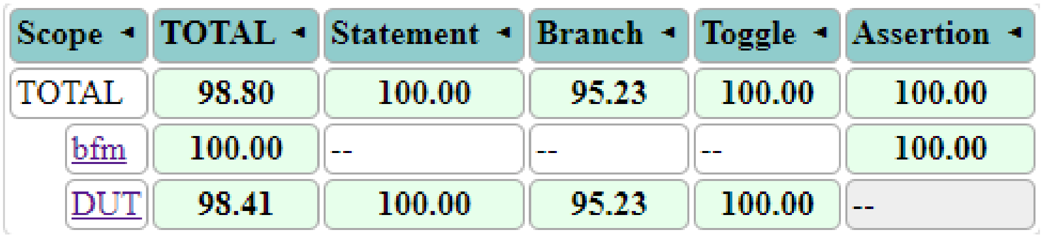


Coverpoints/Crosses	Total Bins	Hits	Hit %	Goal %	Coverage %
Coverpoint carriers	3	3	100.00%	100.00%	100.00%
Coverpoint data	4	4	100.00%	100.00%	100.00%
Cross #cross__0#	12	12	100.00%	100.00%	100.00%

Figure 6.8: Modulator QPSK covergroup

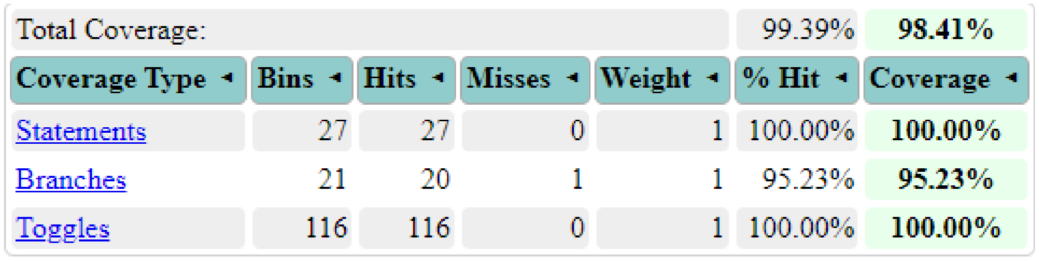
6.1.2.2 Code Coverage

code coverage results are as shown. see 6.9



Scope	TOTAL	Statement	Branch	Toggle	Assertion
TOTAL	98.80	100.00	95.23	100.00	100.00
bfm	100.00	--	--	--	100.00
DUT	98.41	100.00	95.23	100.00	--

Figure 6.9: Modulator coverage summary by instance

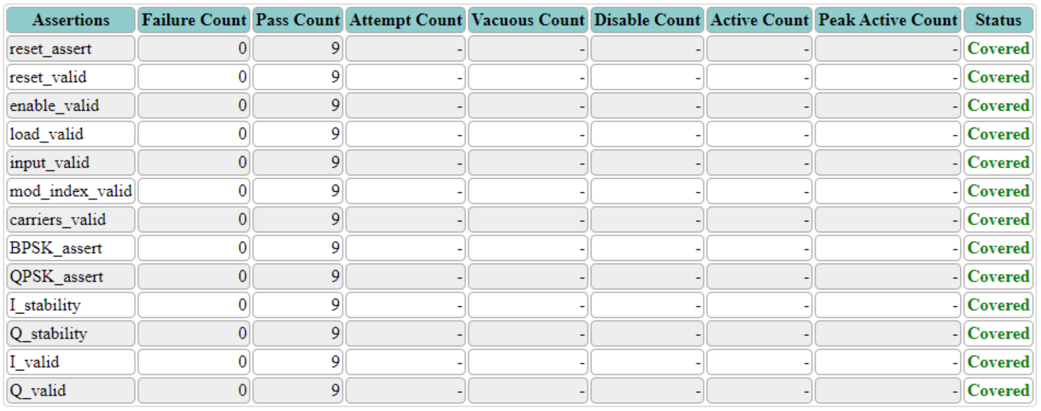


Total Coverage:					99.39%	98.41%
Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	27	27	0	1	100.00%	100.00%
Branches	21	20	1	1	95.23%	95.23%
Toggles	116	116	0	1	100.00%	100.00%

Figure 6.10: Modulator Recursive Hierarchical Coverage Details

6.1.2.3 Assertions

All the assertions passed as shown see 6.11



Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
reset_assert	0	9	-	-	-	-	-	Covered
reset_valid	0	9	-	-	-	-	-	Covered
enable_valid	0	9	-	-	-	-	-	Covered
load_valid	0	9	-	-	-	-	-	Covered
input_valid	0	9	-	-	-	-	-	Covered
mod_index_valid	0	9	-	-	-	-	-	Covered
carriers_valid	0	9	-	-	-	-	-	Covered
BPSK_assert	0	9	-	-	-	-	-	Covered
QPSK_assert	0	9	-	-	-	-	-	Covered
I_stability	0	9	-	-	-	-	-	Covered
Q_stability	0	9	-	-	-	-	-	Covered
I_valid	0	9	-	-	-	-	-	Covered
Q_valid	0	9	-	-	-	-	-	Covered

Figure 6.11: Modulator assertion coverage

6.1.2.4 Modulator Bugs

No bugs were found in the modulator

6.1.2.5 Modulator negative test results

When applying an illegal value to input Msc, the DUT raises the signal Mod_O_Valid indicating that the output is valid while both Mod_I and Mod_Q are unknown.

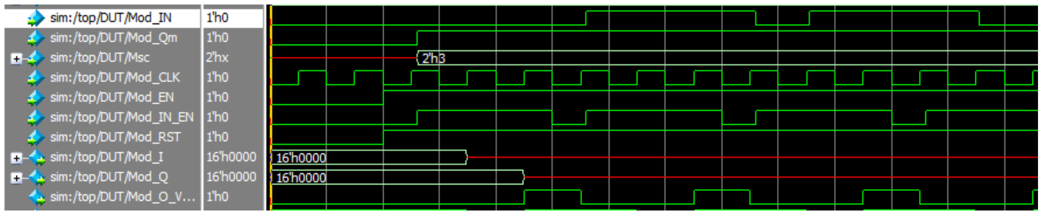

sim:/top/DUT/Mod_IN 1'h0
sim:/top/DUT/Mod_Qm 1'h0
sim:/top/DUT/Msc 2'hx
sim:/top/DUT/Mod_CLK 1'h0
sim:/top/DUT/Mod_EN 1'h0
sim:/top/DUT/Mod_IN_EN 1'h0
sim:/top/DUT/Mod_RST 1'h0
sim:/top/DUT/Mod_I 16'h0000
sim:/top/DUT/Mod_Q 16'h0000
sim:/top/DUT/Mod_O_V... 1'h0
2'h3
16'h0000
16'h0000

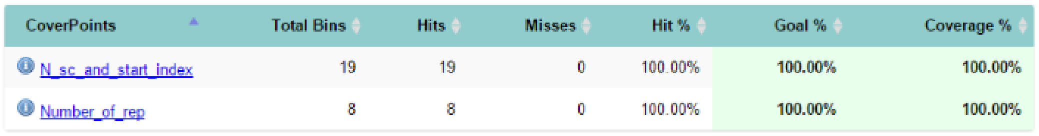
Figure 6.12: Modulator negative test

6.1.3 Resource Element Mapper results

These results are obtained by running all tests and merging the coverage results

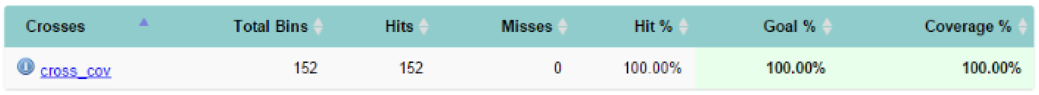
6.1.3.1 Functional coverage

Cover groups results are shown see 6.13 6.14



CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
N_sc_and_start_index	19	19	0	100.00%	100.00%	100.00%
Number_of_rep	8	8	0	100.00%	100.00%	100.00%

Figure 6.13: Resource Element Mapper covergroup input

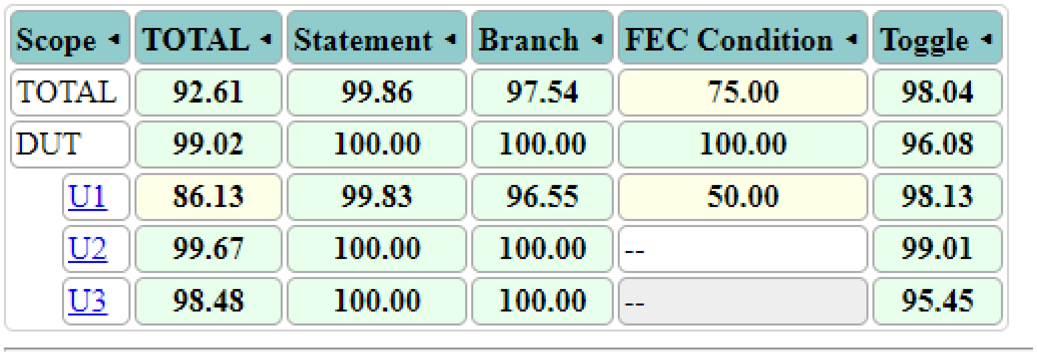


Crosses	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
cross_cov	152	152	0	100.00%	100.00%	100.00%

Figure 6.14: Resource Element Mapper cross coverage

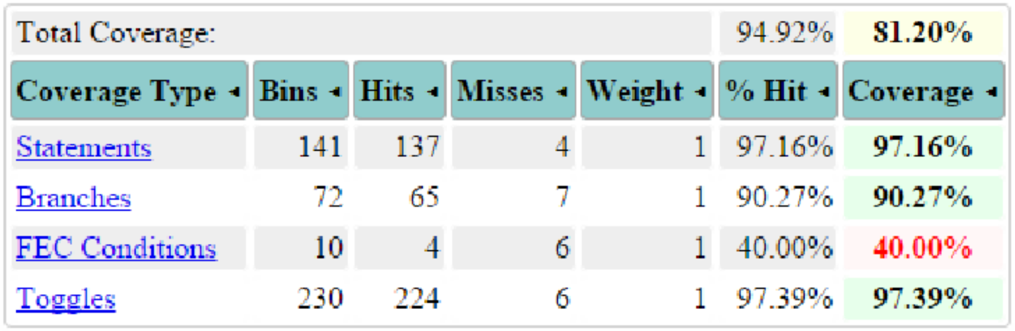
6.1.3.2 Code Coverage

code coverage results are as shown. see 6.15



Scope	TOTAL	Statement	Branch	FEC Condition	Toggle
TOTAL	92.61	99.86	97.54	75.00	98.04
DUT	99.02	100.00	100.00	100.00	96.08
U1	86.13	99.83	96.55	50.00	98.13
U2	99.67	100.00	100.00	--	99.01
U3	98.48	100.00	100.00	--	95.45

Figure 6.15: Resource Element Mapper coverage summary by instance

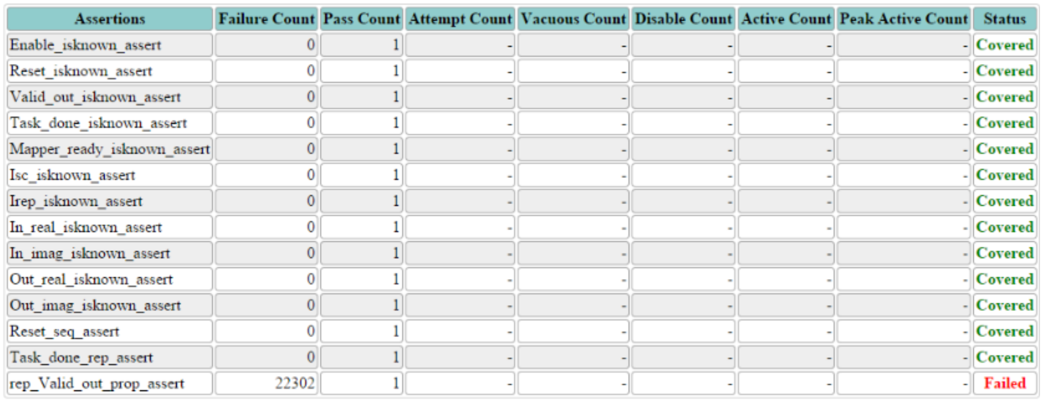


Total Coverage:					94.92%	81.20%
Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	141	137	4	1	97.16%	97.16%
Branches	72	65	7	1	90.27%	90.27%
FEC Conditions	10	4	6	1	40.00%	40.00%
Toggles	230	224	6	1	97.39%	97.39%

Figure 6.16: Resource Element Mapper Recursive Hierarchical Coverage Details

6.1.3.3 Assertions

All the assertions passed as shown see 6.17



Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
Enable_isknown_assert	0	1	-	-	-	-	-	Covered
Reset_isknown_assert	0	1	-	-	-	-	-	Covered
Valid_out_isknown_assert	0	1	-	-	-	-	-	Covered
Task_done_isknown_assert	0	1	-	-	-	-	-	Covered
Mapper_ready_isknown_assert	0	1	-	-	-	-	-	Covered
Isc_isknown_assert	0	1	-	-	-	-	-	Covered
Irep_isknown_assert	0	1	-	-	-	-	-	Covered
In_real_isknown_assert	0	1	-	-	-	-	-	Covered
In_imag_isknown_assert	0	1	-	-	-	-	-	Covered
Out_real_isknown_assert	0	1	-	-	-	-	-	Covered
Out_imag_isknown_assert	0	1	-	-	-	-	-	Covered
Reset_seq_assert	0	1	-	-	-	-	-	Covered
Task_done_rep_assert	0	1	-	-	-	-	-	Covered
rep_Valid_out_prop_assert	22302	1	-	-	-	-	-	Failed

Figure 6.17: Resource Element Mapper assertion coverage

6.1.3.4 Resource Element Mapper Bugs

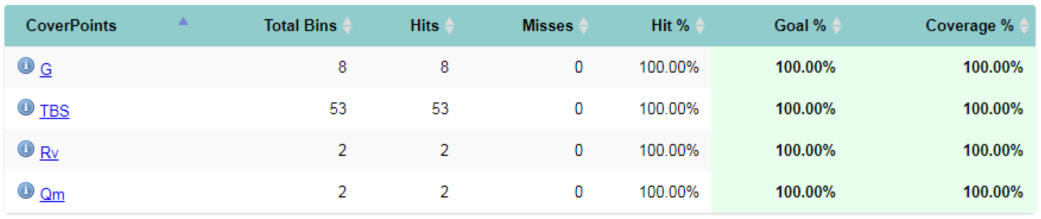
No bugs were found in the Resource Element Mapper.

6.1.4 Rate matcher results

These results are obtained by running all tests and merging the coverage results

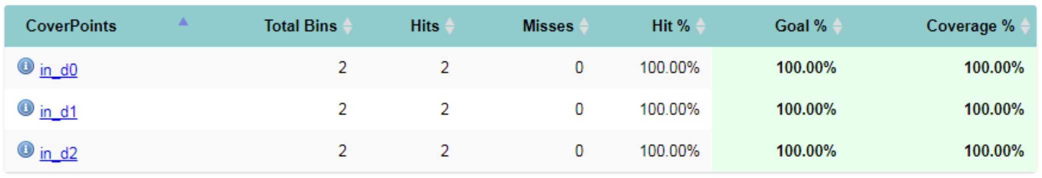
6.1.4.1 Functional coverage

Cover groups results are shown see 6.18 6.20 6.19



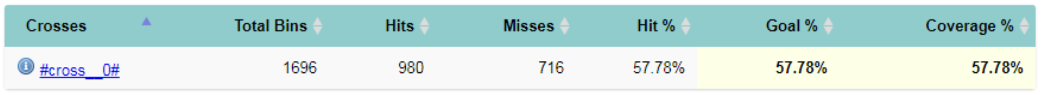
CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
G	8	8	0	100.00%	100.00%	100.00%
TBS	53	53	0	100.00%	100.00%	100.00%
Rv	2	2	0	100.00%	100.00%	100.00%
Qm	2	2	0	100.00%	100.00%	100.00%

Figure 6.18: Rate matcher coverage group control



CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
in_d0	2	2	0	100.00%	100.00%	100.00%
in_d1	2	2	0	100.00%	100.00%	100.00%
in_d2	2	2	0	100.00%	100.00%	100.00%

Figure 6.19: Rate matcher coverage group data

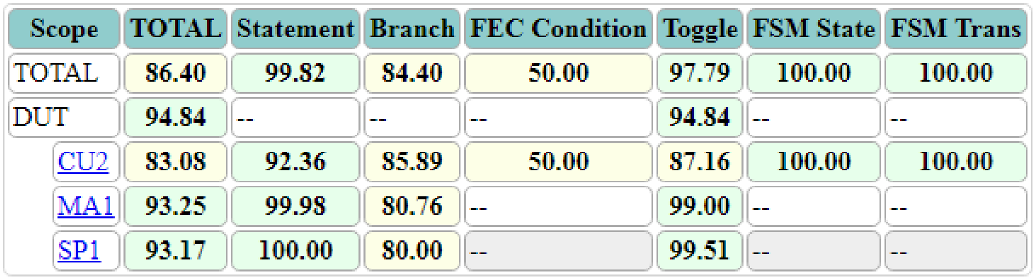


Crosses	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
#cross__0#	1696	980	716	57.78%	57.78%	57.78%

Figure 6.20: Rate matcher cross coverage group control

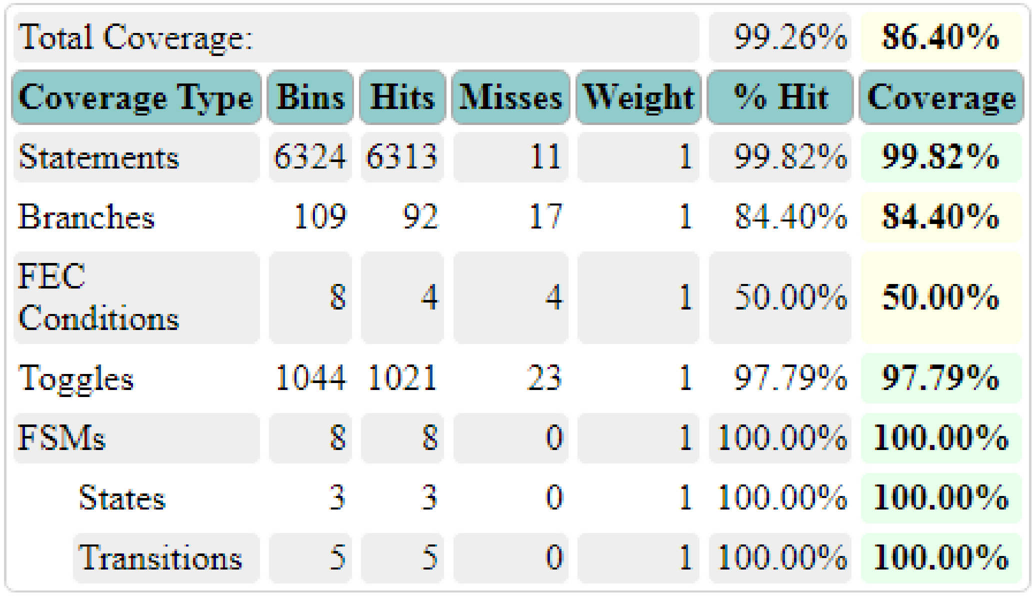
6.1.4.2 Code Coverage

code coverage results are as shown. see 6.21



Scope	TOTAL	Statement	Branch	FEC Condition	Toggle	FSM State	FSM Trans
TOTAL	86.40	99.82	84.40	50.00	97.79	100.00	100.00
DUT	94.84	--	--	--	94.84	--	--
CU2	83.08	92.36	85.89	50.00	87.16	100.00	100.00
MA1	93.25	99.98	80.76	--	99.00	--	--
SP1	93.17	100.00	80.00	--	99.51	--	--

Figure 6.21: Rate matcher Coverage results By Instance

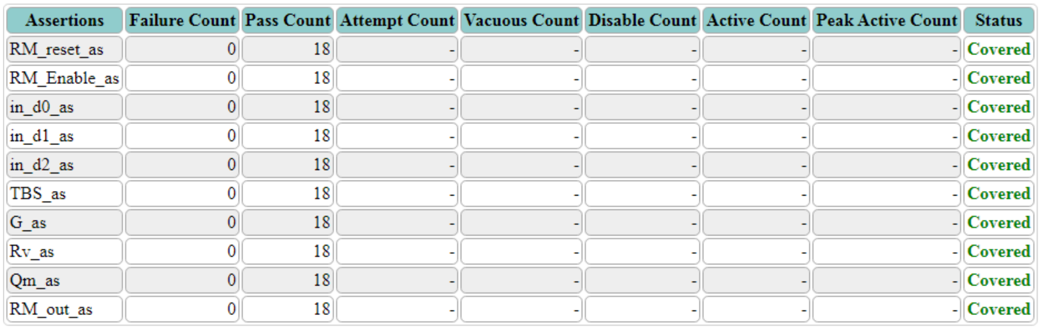


Total Coverage:					99.26%	86.40%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	6324	6313	11	1	99.82%	99.82%
Branches	109	92	17	1	84.40%	84.40%
FEC Conditions	8	4	4	1	50.00%	50.00%
Toggles	1044	1021	23	1	97.79%	97.79%
FSMs	8	8	0	1	100.00%	100.00%
States	3	3	0	1	100.00%	100.00%
Transitions	5	5	0	1	100.00%	100.00%

Figure 6.22: Rate matcher Recursive Hierarchical Coverage Details

6.1.4.3 Assertions

All the assertions passed as shown see 6.23



Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
RM_reset_as	0	18	-	-	-	-	-	Covered
RM_Enable_as	0	18	-	-	-	-	-	Covered
in_d0_as	0	18	-	-	-	-	-	Covered
in_d1_as	0	18	-	-	-	-	-	Covered
in_d2_as	0	18	-	-	-	-	-	Covered
TBS_as	0	18	-	-	-	-	-	Covered
G_as	0	18	-	-	-	-	-	Covered
Rv_as	0	18	-	-	-	-	-	Covered
Qm_as	0	18	-	-	-	-	-	Covered
RM_out_as	0	18	-	-	-	-	-	Covered

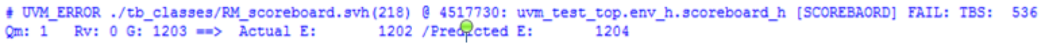
Figure 6.23: Rate matcher Assertion Coverage Report

6.1.4.4 Rate matcher Bugs

- **Incorrect value for the output sequence length value E**

A bug was found by the environment , the bug is The DUT fails to calculate output sequence length value in certain case: when the upper layer parameters values are as follow modulation index(Qm) = 1 and G = an odd number as the output length is determined $E = Qm * \lceil G' \rceil$, then it's observed that the DUT fails to do ceiling function

for example Qm=1 , G 1203 E resulted from DUT =1202 which is wrong. E resulted form scoreboard model = 1204 which is right.



```
# UVM_ERROR ./tb_classes/RM_scoreboard.svh(218) @ 4517730: uvm_test_top.env_h.scoreboard_h [SCOREBAORD] FAIL: TBS:   536
Qm: 1   Rv: 0 G: 1203 ==>  Actual E:        1202 /Predicted E:        1204
```

Figure 6.24: Incorrect value for the output sequence length value E

- **Error in stream bits**

it 's found that the last matrix of the 3 matrices do the interpolation in a wrong way as according to 3 GPP NB-LTE standard the interpolation is done by interchanging the columns of the matrix according to certain table

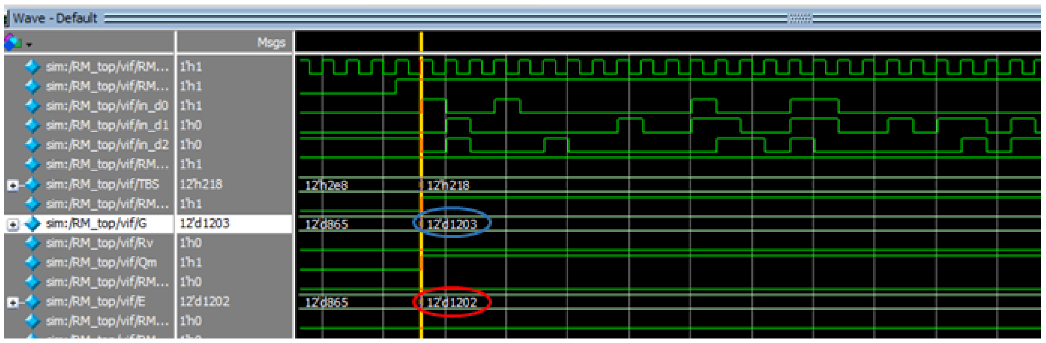

Wave - Default
Msgs
sim:/RM_top/vif/RM... 1'h1
sim:/RM_top/vif/RM... 1'h1
sim:/RM_top/vif/in_d0 1'h1
sim:/RM_top/vif/in_d1 1'h0
sim:/RM_top/vif/in_d2 1'h0
sim:/RM_top/vif/RM... 1'h1
sim:/RM_top/vif/TBS 12'h218
sim:/RM_top/vif/RM... 1'h1
sim:/RM_top/vif/G 12'd1203
sim:/RM_top/vif/Rv 1'h0
sim:/RM_top/vif/Qm 1'h1
sim:/RM_top/vif/RM... 1'h0
sim:/RM_top/vif/E 12'd1202
sim:/RM_top/vif/RM... 1'h0
12'h2e8
12'h218
12'd865
12'd1203
12'd865
12'd1202

Figure 6.25: Incorrect value for the output sequence length value E wave form

, however when interpolation is done on third matrix in DUT there is an up/down flipping done to the last column which is wrong. This problem cannot affect the output bits negatively except when the required output length is nearly greater than 3 times the input bits.

- **No resulted output in high TBS values**

It's found that the DUT is not responding or resulting any output in high transport block size (TBS) values 2024, 2280, 2536. the environment can overcomes this case and do a new random test automatically.

- **Combinational implementation output sequence length**

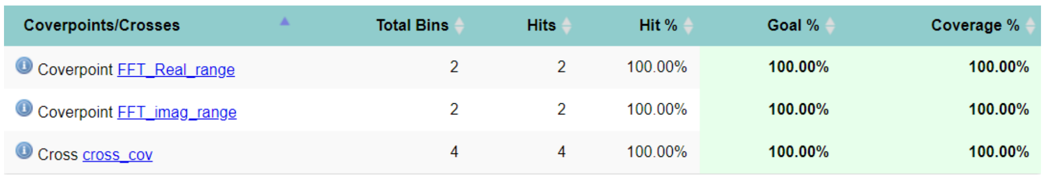
It's observed that in the RTL the output sequence length E is implemented combinational not sequential which means the value of output sequence length can be changed within the transaction and this may affects the design badly in case of glitches or all metastability cases.

6.1.5 FFT results

These results are obtained by running all tests and merging the coverage results

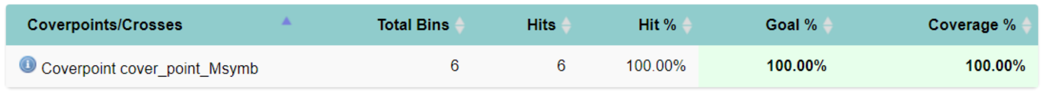
6.1.5.1 Functional coverage

Cover groups results are shown see **??** 6.27



Coverpoints/Crosses	Total Bins	Hits	Hit %	Goal %	Coverage %
Coverpoint FFT_Real_range	2	2	100.00%	100.00%	100.00%
Coverpoint FFT_imag_range	2	2	100.00%	100.00%	100.00%
Cross cross_cov	4	4	100.00%	100.00%	100.00%

Figure 6.26: fft covergroup input

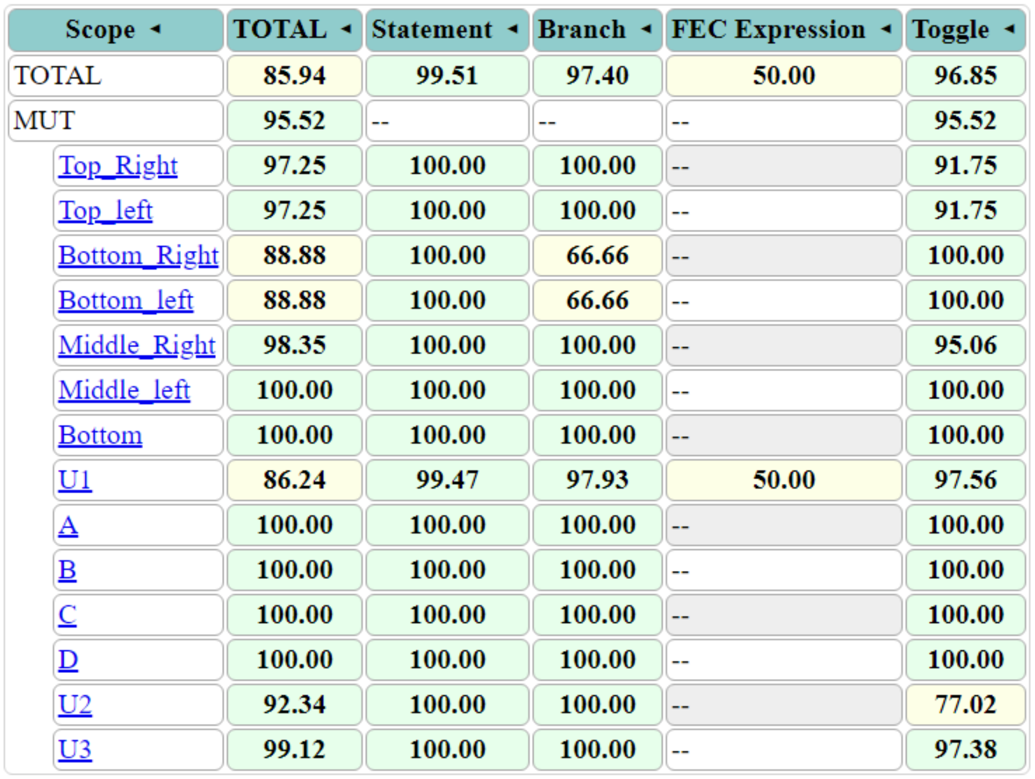


Coverpoints/Crosses	Total Bins	Hits	Hit %	Goal %	Coverage %
Coverpoint cover_point_Msymb	6	6	100.00%	100.00%	100.00%

Figure 6.27: FFT Covergroup Msymb

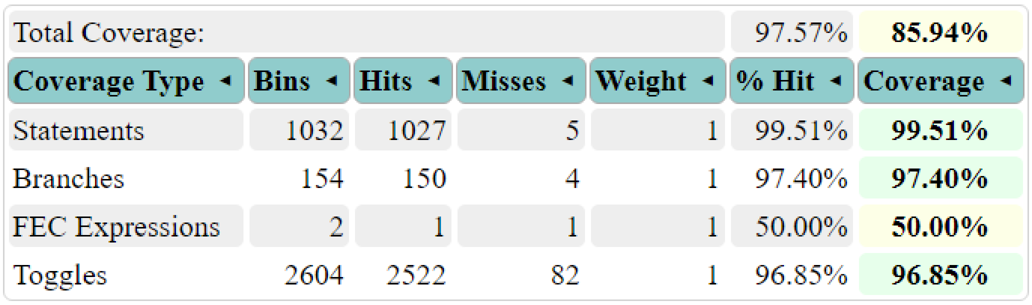
6.1.5.2 Code Coverage

code coverage results are as shown. see 6.28



Scope	TOTAL	Statement	Branch	FEC Expression	Toggle
TOTAL	85.94	99.51	97.40	50.00	96.85
MUT	95.52	--	--	--	95.52
Top_Right	97.25	100.00	100.00	--	91.75
Top_left	97.25	100.00	100.00	--	91.75
Bottom_Right	88.88	100.00	66.66	--	100.00
Bottom_left	88.88	100.00	66.66	--	100.00
Middle_Right	98.35	100.00	100.00	--	95.06
Middle_left	100.00	100.00	100.00	--	100.00
Bottom	100.00	100.00	100.00	--	100.00
U1	86.24	99.47	97.93	50.00	97.56
A	100.00	100.00	100.00	--	100.00
B	100.00	100.00	100.00	--	100.00
C	100.00	100.00	100.00	--	100.00
D	100.00	100.00	100.00	--	100.00
U2	92.34	100.00	100.00	--	77.02
U3	99.12	100.00	100.00	--	97.38

Figure 6.28: FFT coverage summary by instance

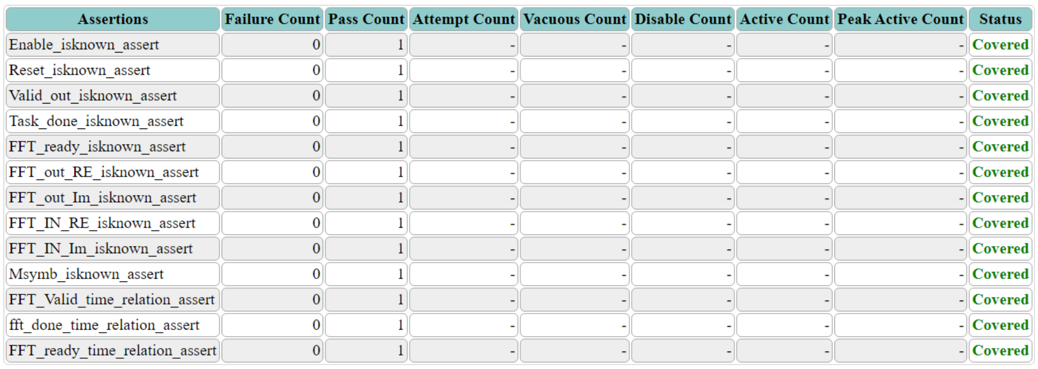


Total Coverage:					97.57%	85.94%
Coverage Type ◂	**Bins** ◂	**Hits** ◂	**Misses** ◂	**Weight** ◂	**% Hit** ◂	**Coverage** ◂
Statements	1032	1027	5	1	99.51%	**99.51%**
Branches	154	150	4	1	97.40%	**97.40%**
FEC Expressions	2	1	1	1	50.00%	**50.00%**
Toggles	2604	2522	82	1	96.85%	**96.85%**

Figure 6.29: FFT Recursive Hierarchical Coverage Details

6.1.5.3 Assertions

All the assertions passed as shown see 6.30



Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
Enable_isknown_assert	0	1	-	-	-	-	-	Covered
Reset_isknown_assert	0	1	-	-	-	-	-	Covered
Valid_out_isknown_assert	0	1	-	-	-	-	-	Covered
Task_done_isknown_assert	0	1	-	-	-	-	-	Covered
FFT_ready_isknown_assert	0	1	-	-	-	-	-	Covered
FFT_out_RE_isknown_assert	0	1	-	-	-	-	-	Covered
FFT_out_Im_isknown_assert	0	1	-	-	-	-	-	Covered
FFT_IN_RE_isknown_assert	0	1	-	-	-	-	-	Covered
FFT_IN_Im_isknown_assert	0	1	-	-	-	-	-	Covered
Msymb_isknown_assert	0	1	-	-	-	-	-	Covered
FFT_Valid_time_relation_assert	0	1	-	-	-	-	-	Covered
fft_done_time_relation_assert	0	1	-	-	-	-	-	Covered
FFT_ready_time_relation_assert	0	1	-	-	-	-	-	Covered

Figure 6.30: FFT assertion coverage

6.1.5.4 FFT Bugs

No bugs were found in the FFT.

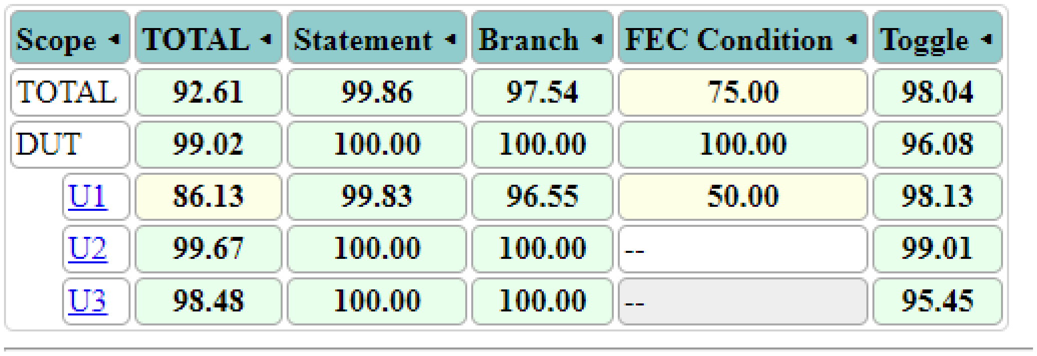
6.1.6 IFFT results

6.1.6.1 Functional coverage

Functional coverage for the IFFT is irrelevant as the block's functionality is implemented incorrectly

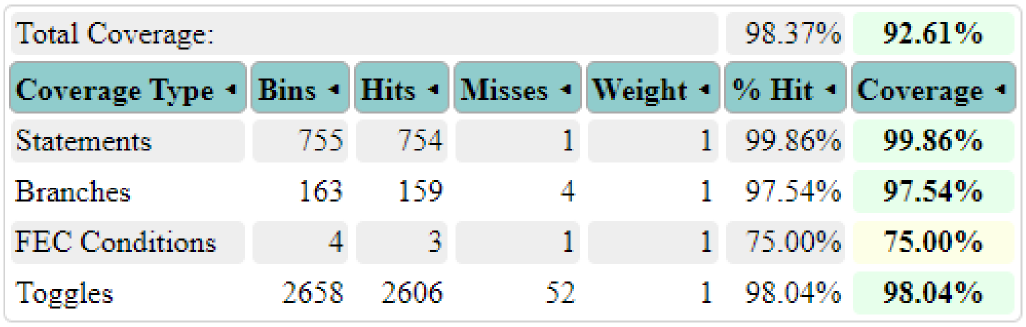
6.1.6.2 Code Coverage

code coverage results are as shown. see 6.31



Scope	TOTAL	Statement	Branch	FEC Condition	Toggle
TOTAL	92.61	99.86	97.54	75.00	98.04
DUT	99.02	100.00	100.00	100.00	96.08
U1	86.13	99.83	96.55	50.00	98.13
U2	99.67	100.00	100.00	--	99.01
U3	98.48	100.00	100.00	--	95.45

Figure 6.31: IFFT coverage summary by instance

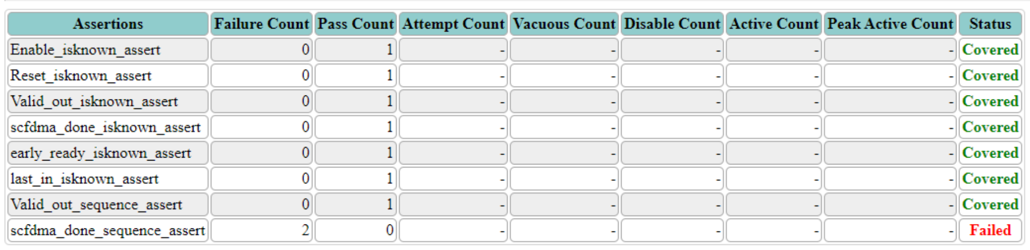


Total Coverage:					98.37%	92.61%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	755	754	1	1	99.86%	99.86%
Branches	163	159	4	1	97.54%	97.54%
FEC Conditions	4	3	1	1	75.00%	75.00%
Toggles	2658	2606	52	1	98.04%	98.04%

Figure 6.32: IFFT Recursive Hierarchical Coverage Details

6.1.6.3 Assertions

For scfdma_done failed as the design specifications defined it to raise to 1 when the final SC-FDMA symbol is completely transmitted but this does not happen (it raises after 53 cycle after the first output symbol). as shown see 6.33



Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
Enable_isknown_assert	0	1	-	-	-	-	-	Covered
Reset_isknown_assert	0	1	-	-	-	-	-	Covered
Valid_out_isknown_assert	0	1	-	-	-	-	-	Covered
scfdma_done_isknown_assert	0	1	-	-	-	-	-	Covered
early_ready_isknown_assert	0	1	-	-	-	-	-	Covered
last_in_isknown_assert	0	1	-	-	-	-	-	Covered
Valid_out_sequence_assert	0	1	-	-	-	-	-	Covered
scfdma_done_sequence_assert	2	0	-	-	-	-	-	Failed

Figure 6.33: IFFT assertion coverage

6.1.6.4 IFFT Bugs

The IFFT has incorrect function implementation, the DUT output is not the same as the estimated from the scoreboard (which is verified using matlab built in function).

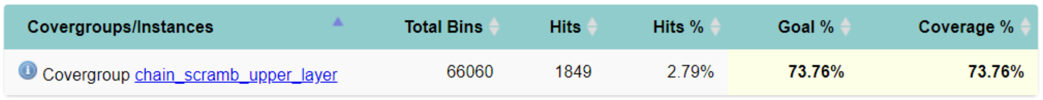
6.2 Top level testing

6.2.1 Top level results

These results are obtained by running all tests and merging the coverage results

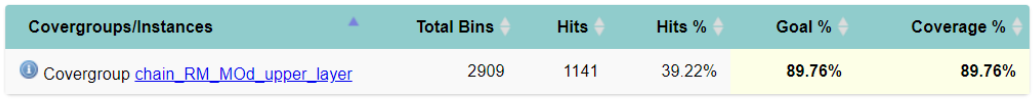
6.2.1.1 Functional coverage

Cover groups results are shown see 6.34 6.35 6.36



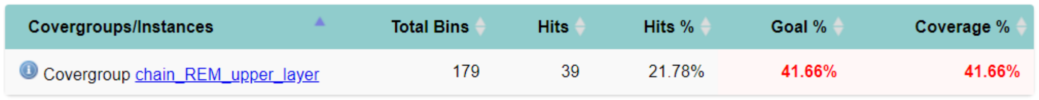
Covergroups/Instances	Total Bins	Hits	Hits %	Goal %	Coverage %
Covergroup chain_scramb_upper_layer	66060	1849	2.79%	73.76%	73.76%

Figure 6.34: Top scramb covergroup input



Covergroups/Instances	Total Bins	Hits	Hits %	Goal %	Coverage %
Covergroup chain_RM_MOd_upper_layer	2909	1141	39.22%	89.76%	89.76%

Figure 6.35: Top RM-mod Covergroup upper layer

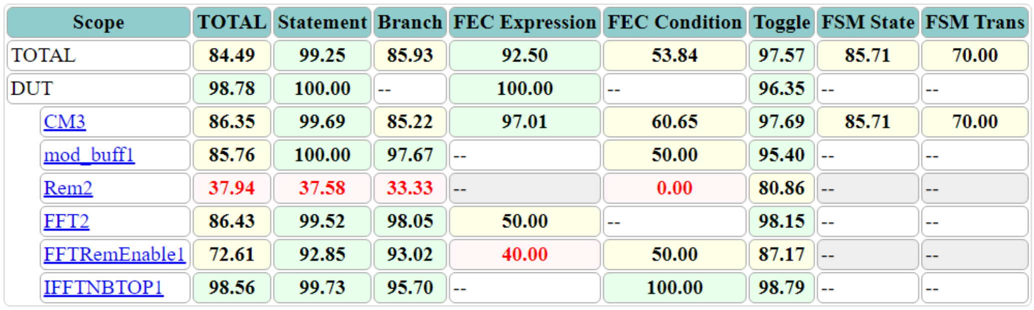


Covergroups/Instances	Total Bins	Hits	Hits %	Goal %	Coverage %
Covergroup chain_REM_upper_layer	179	39	21.78%	41.66%	41.66%

Figure 6.36: Top REM Covergroup upper layer

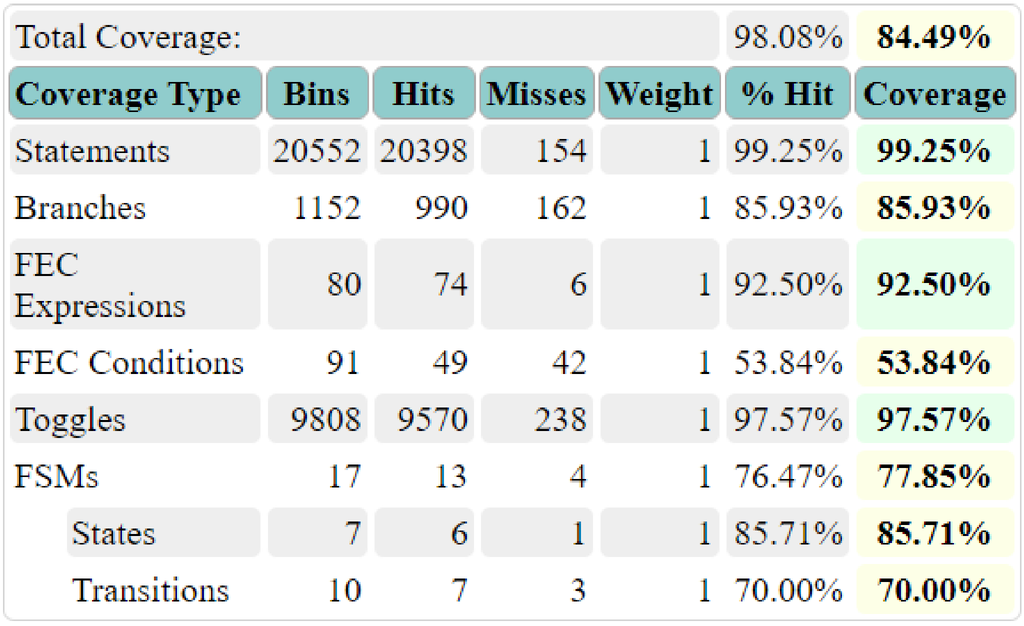
6.2.1.2 Code Coverage

Code Coverage didn't reach 100% as there is no control on the internal signals TBS has only even value then some signals don't toggle , Also TBS valid values according to standard don't cover all range of values.
code coverage results are as shown. see 6.37 6.38



Scope	TOTAL	Statement	Branch	FEC Expression	FEC Condition	Toggle	FSM State	FSM Trans
TOTAL	84.49	99.25	85.93	92.50	53.84	97.57	85.71	70.00
DUT	98.78	100.00	--	100.00	--	96.35	--	--
CM3	86.35	99.69	85.22	97.01	60.65	97.69	85.71	70.00
mod_buff1	85.76	100.00	97.67	--	50.00	95.40	--	--
Rem2	37.94	37.58	33.33	--	0.00	80.86	--	--
FFT2	86.43	99.52	98.05	50.00	--	98.15	--	--
FFTRemEnable1	72.61	92.85	93.02	40.00	50.00	87.17	--	--
IFFTNBTOP1	98.56	99.73	95.70	--	100.00	98.79	--	--

Figure 6.37: Top coverage summary by instance

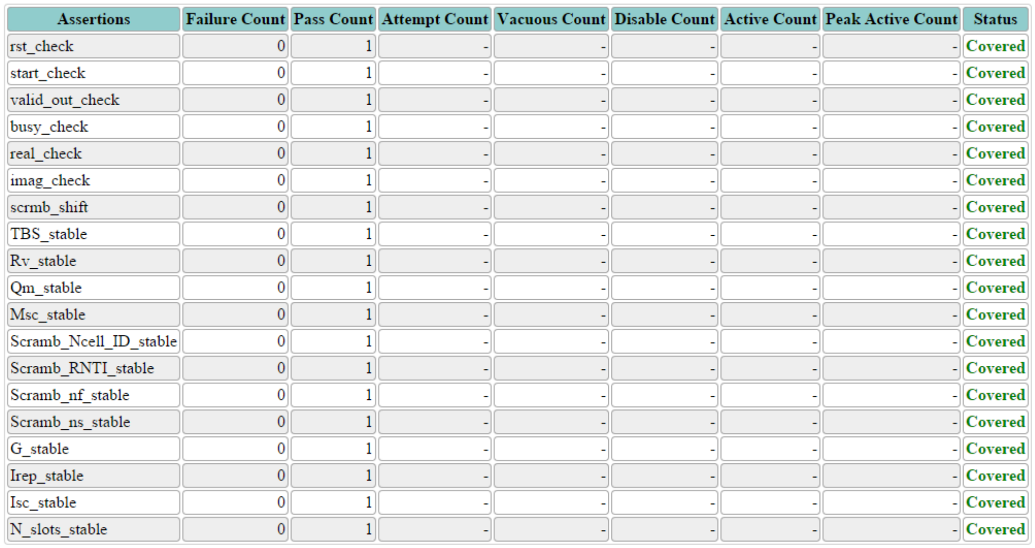


Total Coverage:					98.08%	**84.49%**
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	20552	20398	154	1	99.25%	**99.25%**
Branches	1152	990	162	1	85.93%	**85.93%**
FEC Expressions	80	74	6	1	92.50%	**92.50%**
FEC Conditions	91	49	42	1	53.84%	**53.84%**
Toggles	9808	9570	238	1	97.57%	**97.57%**
FSMs	17	13	4	1	76.47%	**77.85%**
States	7	6	1	1	85.71%	**85.71%**
Transitions	10	7	3	1	70.00%	**70.00%**

Figure 6.38: Top Recursive Hierarchical Coverage Details

6.2.1.3 Assertions

The assertions are shown see 6.39



Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
rst_check	0	1	-	-	-	-	-	Covered
start_check	0	1	-	-	-	-	-	Covered
valid_out_check	0	1	-	-	-	-	-	Covered
busy_check	0	1	-	-	-	-	-	Covered
real_check	0	1	-	-	-	-	-	Covered
imag_check	0	1	-	-	-	-	-	Covered
scrmb_shift	0	1	-	-	-	-	-	Covered
TBS_stable	0	1	-	-	-	-	-	Covered
Rv_stable	0	1	-	-	-	-	-	Covered
Qm_stable	0	1	-	-	-	-	-	Covered
Msc_stable	0	1	-	-	-	-	-	Covered
Scramb_Ncell_ID_stable	0	1	-	-	-	-	-	Covered
Scramb_RNTI_stable	0	1	-	-	-	-	-	Covered
Scramb_nf_stable	0	1	-	-	-	-	-	Covered
Scramb_ns_stable	0	1	-	-	-	-	-	Covered
G_stable	0	1	-	-	-	-	-	Covered
Irep_stable	0	1	-	-	-	-	-	Covered
Isc_stable	0	1	-	-	-	-	-	Covered
N_slots_stable	0	1	-	-	-	-	-	Covered

Figure 6.39: Top assertion coverage

6.3 Conclusion

Function verification is a very essential phase to verify any RTL to test its function correctly, is still one the most challenging activities in digital system development , The purpose of a testbench is to determine the correctness of the design under test (DUT).
Generating stimulus are most important step where this step generate the inputs which expresses a certain feature test, Random stimulus is crucial for exercising complex designs. A directed test finds the bugs you expect to be in the design, while a random test can find bugs you never anticipated.

Chapter 7

References

- [1] 3GPP TS 36.211 V14.4.0 (2017-09)
- [2] 3GPP TS 36.201 V14.1.0 (2017-03)
- [3] 3GPP TS 36.212 V14.4.0 (2017-09)
- [4] 3GPP TS 36.213 V14.4.0 (2017-09)
- [5]Low_Power_Design_of_the_Baseband_Physical_Layer_of_NarrowBand_IoT_LTE _Uplink_Digital_Transmitter
- [6]Design_and_Verification_of _Digital_Systems
- [7] systemverilog for verification . chris spear . 2012
- [8] Ray-Salemi-The-UVM-Primer-An-Introduction-to-the-Universal-Verification-Methodology-Boston-Light-Press-2013
- [9] Universal Verification Methodology (UVM) 1.1 Class Reference
- [10] Cookbook by Mentor Graphics-2012