12-2019

UVM Verification of a Floating Point Multiplier

Nicholas J. Marsaw
njm3706@rit.edu

UVM verification of a Floating Point Multiplier

by

Nicholas J. Marsaw

Graduate Paper

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Electrical Engineering

Approved by:

____

Mr. Mark A. Indovina, Senior Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

____

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York

December, 2019

I dedicate this work to my elementary school teacher Darrel Dupra, who passed away in 2010. He took time to encourage me to think critically and to enjoy the journey as I progressed throughout my academics, and played a crucial role in my pursuit of Electrical Engineering.

Declaration

I hereby declare that except where specific reference is made to the work of others, that all contents of this Graduate Paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This Graduate Project is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Nicholas J. Marsaw

December, 2019

Acknowledgements

I want to thank Mark A. Indovina for his support, advice, and guidance throughout my graduate research and education. Your passion for the engineering field and dedication to your students is truly valuable. I would also like thank my family for their encouragement as I've worked through my education. You have been extremely patient and loving.

Lastly, I would like to thank Anna for her love and support over the past few years as I have been finishing up my academics. You're very special to me, and I couldn't have accomplished this without you.

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. Especially when performing verification on designs involving audio signal processing, untested corner cases and bugs may result in significant and sometimes undiscovered processing errors. This paper explores the use of SystemVerilog and the universal verification methodology (UVM) class library to verify a pipelined floating-point multiplier (FMULT) within the adaptive differential pulse code modulation (ADPCM) specification.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When an intellectual property (IP) chip is taped out, bugs and design flaws are found in the hardware and require re-spin. In order to mitigate time and cost spent on reworking chip designs, verification is used to catch these issues prior to tape out. Verification has become increasingly necessary as gate sizing has decreased, allowing for increased design complexity in smaller chips. In the past few decades, the hardware description languages (HDL) most commonly used did not present sufficient verification constructs, and as a result many engineers made use of other languages such as OpenVera in order to attain the level of functionality their testbenches required. Other engineers and companies designed their own verification languages and libraries as well. In 2005, SystemVerilog (SV), an object-oriented programming language, was adopted as an IEEE standard with the goal of unifying verification and design, and providing a language for verification that has readability, reusability and efficiency.

Following the adoption of SV, the open verification methodology (OVM), a class library written in SV, was created. OVM provides automation and transaction level modeling for SystemVerilog testbench designs. The testbench structure provided by OVM allows for reusability in other verification environments and makes use of tools provided in SystemVerilog such as code coverage, assertions, and DPI. OVM would later evolve into the universal verification methodology (UVM), which combines various verification practices to make up the first standardized verification methodology. This paper explores the use of SV and UVM for verifying the floating point multiplier (FMULT) used in the G.726 Adaptive Differential Pulse-Code Modulation (ADPCM) design specification [1], which consists of multiplying an 11-bit floating point binary number with a 16-bit floating point binary number, resulting in a 16-bit product. The FMULT was designed in Verilog with a pipelined architecture using one adder for the necessary additions.

1.1 Research Goals

The goal of this paper is to research and develop a testbench using SystemVerilog and UVM, verifying the floating point multiplier (FMULT). The testbench is a multi-layered, self-checking design. For success, the following goals are considered:

- Understanding ADPCM operation and how the FMULT relates to the overall specification
- Designing a test environment in UVM with self-checking using a reference model and random stimulus
- Running simulations for RTL and gate-level designs
- Collecting coverage results and test results

1.2 Contributions

The major contributions for the paper are as follows:

- A floating point multiplier (FMULT) designed in Verilog

- Verification of the FMULT using a multi-layered testbench written in SystemVerilog and UVM
- Reusable UVM components to conduct verification on other parts of the ADPCM

1.3 Organization

The organization of the paper is as follows:

- Chapter 2: This chapter provides context to the UVM through research
- Chapter 3: This chapter discusses adaptive differential pulse code modulation and where the FMULT is used in the design
- Chapter 4: This chapter provides an overview to UVM and the main components used in a multi-layered testbench
- Chapter 5: This chapter discusses the architecture of the testbench and the design integration
- Chapter 6: The results of the tests are provided and discussed
- Chapter 7: The paper concludes here and possible future work is discussed

Chapter 2

Bibliographical Research

Prior to the introduction of verification methodologies, engineers used traditional verification techniques to verify intellectual property (IP) before tape out. These traditional techniques had their limitations; the testbench design affected code reuse and reapplication in future designs [2]. Another drawback with the use of traditional verification was its inability to test complex systems due to the lack of a strong tool set. This time consuming process would take up over 70% of the time spent on the designs, and the introduction of verification methodologies in the following decades would serve to help lower the time and effort put into chip verification [3]. These methodologies aimed at providing a verification language, library, and/or tool set with reusability. One way these methodologies accomplished this was through the use of object oriented programming (OOP), which was found in the Advanced Verification Methodology (AVM) [4], Universal Reuse Methodology (URM), *e* Reuse Methodology (eRM), Open Verification Methodology (OVM) and the universal verification methodology (UVM). Using OOP allowed the testbench to be broken up into smaller components, providing increased flexibility, simplicity, and reusability lacking in traditional verification techniques [5]. Of the various methodologies created and adopted, UVM is gaining ground and becoming popular among verification engineers. UVM is also the first methodology to be standardized.

One of the stepping stones to the development of UVM was SystemVerilog (SV). SV sought to address some of the issues in the verification process across the industry, some of which being the lack of unified design, specification, and verification [6]. The verification language was designed to fully support backwards compatibility with Verilog as well as Verilog constructs. In essence, SV was an expansion to the Verilog HDL, providing more robustness in verification. As a language capable of both design and verification, or a hardware description and verification language (HDVL), SV was adopted by IEEE as a standard in 2005 [7]. SV also included several tools beneficial for thorough verification of complex designs: assertions, coverage, DPI, and supported data types not present in Verilog. Assertions and coverage are two components of UVM inherited from SV, and are critical tools used for verification.

Assertions are used to indicate an error if a particular event occurs during simulation run time. The event typically involves output comparison or the behavior of the design under test (DUT) during verification (i.e. enable is not active when it should be, etc). There are 2 types of assertions in SV: concurrent and immediate [8]. Concurrent assertions involve conditions that must be satisfied by the design at all times. Immediate assertions however are checked periodically, typically after an event. SV provides the assertion tool set through System Verilog Assertions (SVA), which can be added and synthesized within the design for debugging and verification. [9] explores synthesizing assertions in a design, stating that the assertions are not treated as the code, but as properties that must hold up in the design. The proposed design was run in parallel with assertion checking from Synopsys OpenVera Assertions (OVA) checker, producing the same results. The simulation for the proposed design ran faster than that of the OVA checker. While the floating-point multiplier (FMULT) proposed in this paper did not include synthesized assertions, this is an area that could be beneficial to explore in future work for both debugging and run time purposes. SVA has also been used for assertion-based verification (ABV), which has been proven to increase efficiency and lower effort in catching corner cases when verifying the design [10].

Coverage is a measurement used in verification for determining the quality of the testing done on the DUT [11]. Quantified as a percentage, the higher coverage is, the more of the design was tested. This includes corner cases, functional coverage, toggling, and user-defined coverage groups. In SV these are known as cover groups. The goal is to reach 100% percent coverage if possible, as untested code could result in defects and extraneous costs after tape out [12].

The UVM is a powerful verification methodology written in SV, providing functionality found in AVM, OVM, URM and eRM [13]. UVM maintains transaction-level modeling (TLM) found in SV and includes a separate component for handling testbench stimulus known as a sequence, which is separated from the testbench structure [14]. There is value to this, as it allows for flexibility for stimulus generation within a testbench design. A class library is used to provide the building blocks for the methodology [15]. Typically used as a multi-layered design, the UVM provides reusability, but tends to be too complicated for simple designs requiring verification. Its flexible framework however proves valuable for complicated designs with mixed-signal verification capabilities [16].

Chapter 3

Adaptive Differential Pulse Code Modulation

Adaptive differential pulse code modulation (ADPCM) is a process of encoding and decoding audio signals from analog to digital and vice versa. It expands on both pulse-code modulation (PCM) and differential pulse-code modulation (DPCM). Converting these audio signals to digital has several benefits: lower costs per data line, ease of maintenance, and high quality signal regeneration at repeaters [17]. ADPCM was first introduced in 1973 by Bell Labs, supporting encoding and decoding for bit rates including 24 kb/s and 32 kb/s. In 1980 Bell Labs published a paper expanding on the ADPCM described in [17], discussing the algorithmic nature and architecture of the ADPCM in depth [18]. ADPCM was released as a specification in 1984, and is commonly used today as the G.726 specification [1].

PCM is the bare-bones modulation approach. Figure 3.1 illustrates the process of encoding a signal using PCM.The process starts with sampling the signal at a frequency typically set to twice the maximum frequency of the analog signal. If the sampling frequency is higher, oversampling can occur which might require signal reconstruction. If the sampling frequency is lower, then the signal will be under-sampled and the data can be misinterpreted. Following sampling, the data is then quantized, placing it in a digital-friendly format. The data sampled is quantized as an approximation of the analog signal, representing the magnitude of the analog signal in binary.

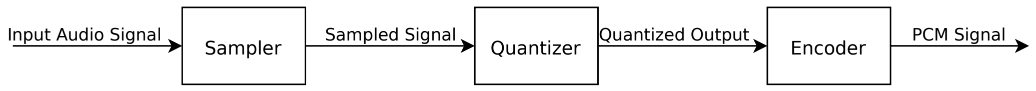


Figure 3.1: PCM Encoding Process

The quantization is determined by the minimum and maximum frequencies, in addition to the sampling frequency. While there are an infinite number of amplitudes that can occur within the minimum and maximum frequency range, the amplitudes are broken up into known values, distributed into L number of evenly-spaced regions. This allows for a constrained range of values that can be used for the approximation of the sampled waveform. The result produces a staircase waveform parallel to the analog waveform from the input function. Following quantization, the data is then encoded in accordance with the G.711 specification [19], which relies on the encoding law used for the data received. There are 2 laws covered within the specification; μ-law and A-law. One distinction between the two is that μ-law uses 13 bits, whereas A-law only uses 12 bits for quantization, and as a result requires a different encoding and decoding process.

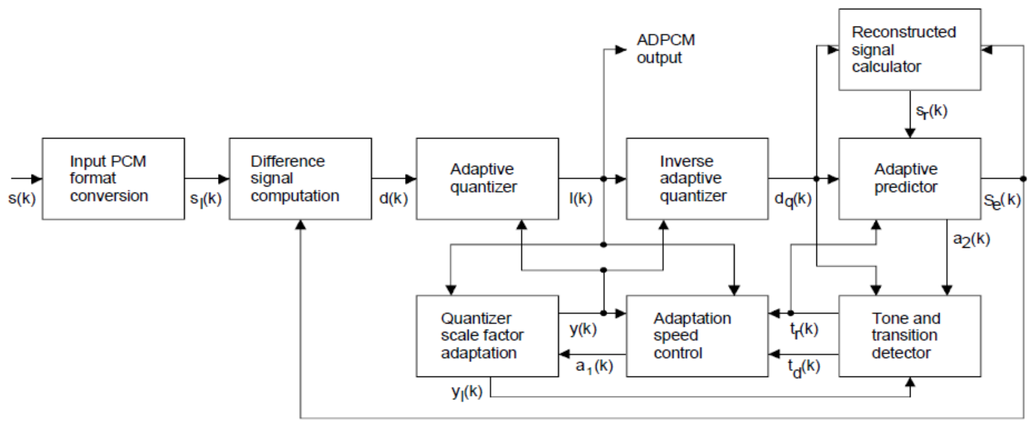
The sampling and quantization processes both have potential for error in PCM. Data sampled and quantized can result in an inaccurate approximate, either undershooting or overshooting the sample point on the original analog frequency. DPCM worked to mitigate this error. Instead of simply quantizing and encoding the analog signal, DPCM takes the difference between the current sample and a predicted sample. This predicted sample originates from calculations performed on the previous sample, utilizing the assumption that the change between 2 samples will be small. The result is no longer a sampled value, but rather a difference between 2 sampled values [20]. This difference mapped alongside the analog waveform will form a staircase as well,

Table 3.1: ADPCM Data Rates

Data Rate	Quantizer Bit Width
16 kb/s	2-bit
24 kb/s	3-bit
32 kb/s	4-bit
40 kb/s	5-bit

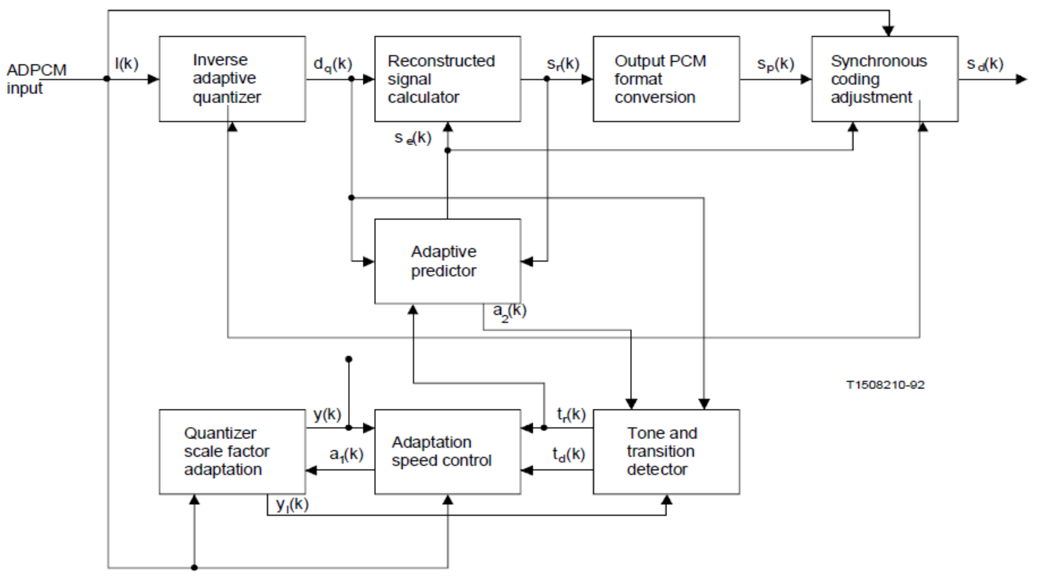
but with smaller values, which allows for more adaptation [21]. One of the benefits DPCM has over PCM in addition to mitigating sample error is the requirement of smaller register sizes used in quantization.

ADPCM is similar to DPCM and is outlined in the G.726 and G.722 specifications [1, 22]. Instead of a defined step size for sampling like in PCM and DPCM, ADPCM is designed with variability to accomodate both large and small changes in the sampled signal. Also, in accordance with the G.726 specification, the ADPCM can be used for handling multiple data rates. The bit width of the quantizer output is scaled based on the data rate. Table 3.1 illustrates the data rates present, and the resulting output of the quantizer relative to the data rate. Figures 3.2 and 3.3 show the diagram for the encoder and decoder in the ADPCM, respectively. The quantization process is enhanced in order to provide this functionality. In addition to the quantizer, the ADPCM has a quantizer scale factor adaptation (QSFA), which is used to compute the quantizer's scaling factor. This scale factor is determined by 2 things: the previous quantizer output and the output of the adaptation speed control. In order to compute the scale factor, the QSFA calculates both a slow *($y_l(k)$)* and a fast ($y_u(k)$) scale factor. Equations 3.1 and 3.2 illustrate the fast and slow scale factor equations, respectively. $W[I(k)]$ makes use of a lookup table, $y(k)$ is the scaling factor, and $a_l(k)$ is the adaptation speed control..



ADPCM output
Reconstructed signal calculator
$s_r(k)$
Input PCM format conversion
Difference signal computation
Adaptive quantizer
Inverse adaptive quantizer
Adaptive predictor
s(k)
$s_l(k)$
d(k)
I(k)
$d_q(k)$
$s_e(k)$
$a_2(k)$
Quantizer scale factor adaptation
Adaptation speed control
Tone and transition detector
y(k)
$t_r(k)$
$a_1(k)$
$t_d(k)$
$y_l(k)$

Figure 3.2: ADPCM Encoder Block Diagram [1]



ADPCM input
I(k)
Inverse adaptive quantizer
$d_q(k)$
Reconstructed signal calculator
$s_r(k)$
Output PCM format conversion
$s_p(k)$
Synchronous coding adjustment
$s_d(k)$
$s_e(k)$
Adaptive predictor
$a_2(k)$
T1508210-92
Quantizer scale factor adaptation
y(k)
$a_1(k)$
Adaptation speed control
$t_r(k)$
$t_d(k)$
Tone and transition detector
$y_l(k)$

Figure 3.3: ADPCM Decoder Block Diagram [1]

$$y_u(k) = (1-2^{-5})y(k) + 2^{-5}W[I(k)] \tag{3.1}$$

$$y_l(k) = (1-2^{-6})y_l(k-1) + 2^{-6}y_u(k) \tag{3.2}$$

$$y(k) = a_l(k)y_u(k-1) + [1-a_l(k)]y_l(k-1) \tag{3.3}$$

As noted in equation 3.3, the scale factor sent to the quantizer uses the slow and fast factors calculated from the previous sampled value, making use of previously collected data to predict the output and sample size necessary to encode the input signal properly. The adaptation speed control operation is documented in [1]. Due to the dynamic stepping of the quantizer in the ADPCM, it proves to be both an economic and efficient digital coding solution for speech compression [23].

In addition to the QSFA, the adaptive predictor and reconstructed signal calculator (APRSC) blocks are utilized to generate the predicted signal which is compared to the current PCM signal. The APRSC is a multi-step, algorithmic design that contains both a sixth order predictor used for modeling zeros, and a second order predictor used for modeling poles of the predicted input signal [1]. Within the APRSC, each order of the predictors requires the use of a floating-point multiplier (FMULT), which produces each of the outputs required for constructing the predicted signal. The FMULT design implemented in this paper is discussed in section 5.1 . The FMULT has a 16-bit input and an 11-bit input, and produces a 16-bit output. Both inputs are converted from two's compliment to floating point format and multiplied. The result is then converted back to two's compliment and sent to the accumulator. For the sixth-order predictor, the FMULT multiplies the predictor coefficient *Bn* with the quantized difference signal *DQn*. For the second-order predictor, the FMULT multiplies the predictor coefficient *An* with the reconstructed signal

SRn. In total, the FMULT block is used 8 times in the APRSC.

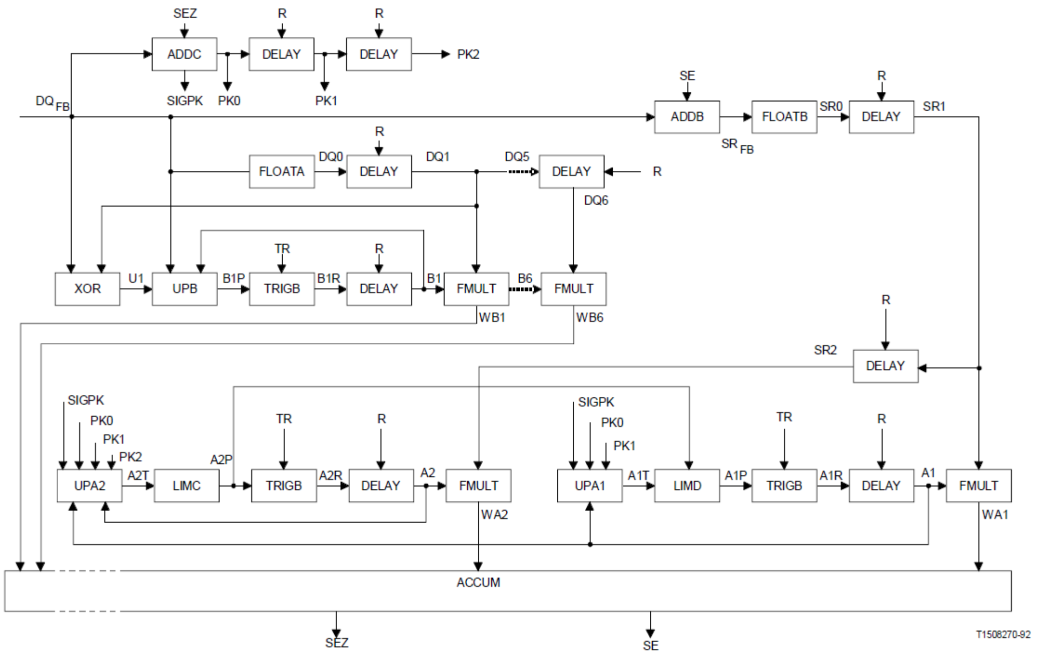


Figure 3.4: APRSC Block Diagram [1]

Chapter 4

UVM Overview

The basic UVM testbench hierarchy is discussed in this chapter. UVM provides a multi-layered testbench architecture where components of each layer communicate through transactions, inheriting concepts and functionality from OVM, URM, eRM, and VMM.

4.1 UVM Hierarchy

Figure 4.1 illustrates the basic UVM testbench hierarchy. These components are crucial for testbench operation, and UVM optimizes operation in each related to their function.

4.1.1 Sequencing

This is a functionality that differs between UVM and SV. There are 3 parts to sequencing: sequence item, sequence, and sequencer.

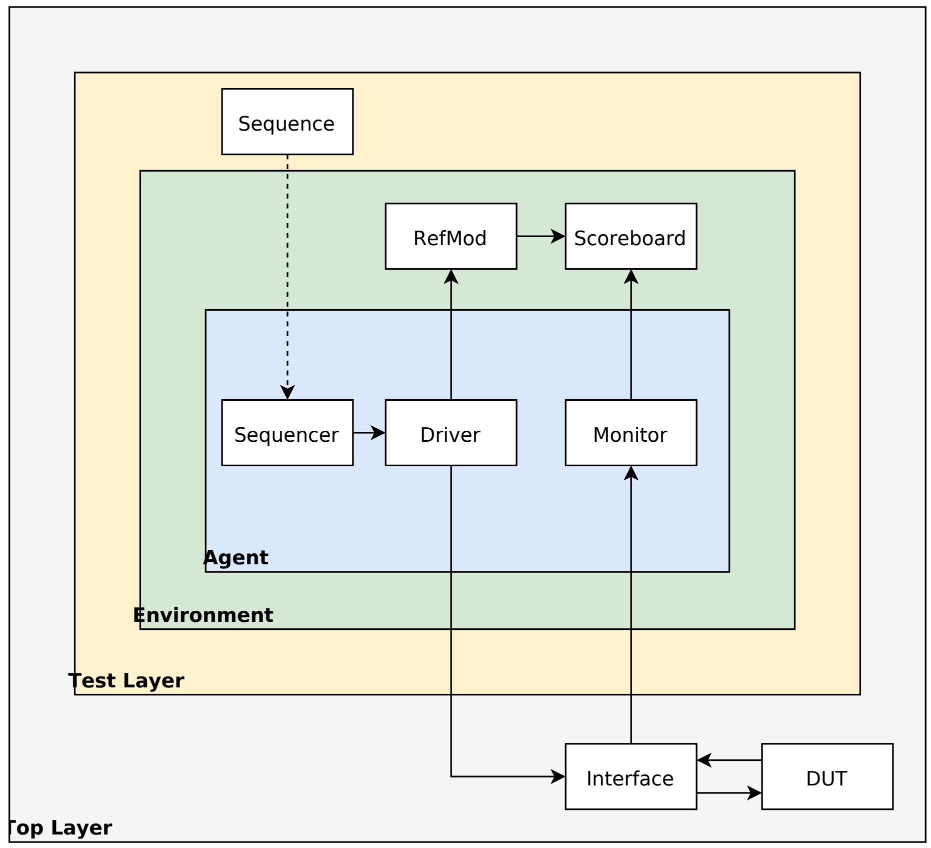


Figure 4.1: Basic UVM Testbench Hierarchy

4.1.1.1 Sequence Item

The sequence item is the component used for transactions between the sequencer and driver. The sequence item is a customizable transaction packet, and is a key component for the sequence and sequencer. The sequence item extends from class *uvm_sequence_item*.

4.1.1.2 Sequence

The sequence is a UVM class used for the generation of stimulus for the testbench. This is typically found at the test-level. The sequence will generate random stimulus and will interact with the driver through the sequencer, sending the data in the form of a sequence item. The sequence extends from *uvm_sequence*.

4.1.1.3 Sequencer

The sequencer is a different UVM class than the sequence, and is instantiated within the agent. A sequence will use the sequencer as the medium to handle transactions within the testbench, specifically the driver. The sequencer extends from class *uvm_sequence*.

4.1.2 Interface

The interface is a UVM component used to connect a DUT or other component to the testbench. Typically, a clock is passed to the interface from the top level instead of using the driver to manage it. Virtual interfaces are commonly used to provide one peripheral for all UVM components to either drive or collect data from the DUT.

4.1.3 Driver

The driver serves the purpose of driving the DUT and, if present, reference models through transactions. The data used by the driver for driving the DUT and models comes from the sequencer in the form of a sequence item. The driver will get the data from the sequencer, and will then send the data to the DUT via a virtual interface tied to the DUT. Reference models can be driven using *uvm_put_ports.* The driver extends from class *uvm_driver*.

4.1.4 Monitor

The monitor is used for managing output transactions and coverage. It will send the collected data to the scoreboard, comparator (if present), or other components for verification. The monitor can also serve the purpose of asserting output conditions as well as verifying the design. The monitor extends class *uvm_monitor*.

4.1.5 Agent

An agent is used to handle transactions through an interface to a design, and a testbench can have multiple agents. Typically, the agent will have the driver, monitor, and sequencer instantiated within it. The agent is also used to connect the driver to the sequencer as well as any reference models, if present. The agent extends class *uvm_agent.*

4.1.6 Environment

The environment contains any agents, the scoreboard, and reference models (if present). Similar to the agent, the environment is also used to handle connections between various components, typically the sequencer to the driver, and if a reference model is present, connecting it to the driver as well as the monitor through a FIFO; UVM's mailbox. The environment extends class *uvm_env*.

4.1.7 Scoreboard

The scoreboard receives data from the monitor and will typically run comparisons on the data received from the monitor, acting as a comparator. The scoreboard also will keep track of the results, which can be accessed during the report phase of the UVM testbench.

4.1.8 Test

This layer instantiates the test environment and the sequence. This layer encapsulates all lower level components in each layer. The test layer is a component extending the UVM class *uvm_test*.

4.1.9 Top

The top level of the UVM testbench is a SV module that instantiates the DUT, interfaces and the test to be performed. Operations such as resets and clock frequencies can be set at this level. The UVM test to perform is also selected at this level.

4.2 Testbench Operation

A UVM testbench consists of 3 main phases: build phase, run-time phase, and clean up phase. These phases are inherited from the class *uvm_component* and provide an organizational structure to the testbench.

4.2.1 Build Phase

The build phase is executed at the start of the simulation. There are 4 functions within the build phase, of which the *build_phase* and *connect_phase* are most used. During *build_phase*, components are created locally or connected to virtual components. During *connect_phase*, FIFOs, get ports, and put ports are connected to higher or lower level components. 2 other functions exist in the build phase: *start_of_simulation_phase* and *end_of_elaboration_phase*. These are used for setting the initial run time and making final adjustments to the testbench prior to simulation, respectively. The build phase executes prior to the actual simulation, and takes up 0 simulation time.

4.2.2 Run-time Phase

The run-time phase is executed during the simulation. Operations such as driving, monitoring, and checking occur during the run-time phase, and are called in the task *run_phase*. The run-time phase also has several functions used for handling DUT resets, configurations, and shutdown.

4.2.3 Clean Up Phase

The clean up phase occurs last before the simulation ends. The purpose of this phase is to check the data collected by the testbench (via the scoreboard) at the end of simulation, and determine whether the test has either passed or reached sufficient coverage. 2 functions used in the clean up phase are the *report_phase* and the *final_phase*. The *report_phase* is useful for printing out any results from the test, and the *final_phase* will complete any tasks not already completed by earlier phases. One factor to be mindful of, however, is that the clean up phase operates bottom-up, so report phases of lower level components will execute before higher level components. A way to avoid clutter for the report phase is to utilize the phase from one of the higher level components.

Chapter 5

Design and Test Methodology

This chapter discusses the design used for the FMULT as well as the testbench architecture used to verify the FMULT.

5.1 FMULT Design

The final step in the APRSC involves accumulating the values calculated by each of the 8 FMULTs used in the hierarchical design (See Figure 3.4). As an option to help lower the resources required for this step, the FMULT was designed with a pipelined architecture and a single resource adder written in Verilog. The design had 2 data inputs: *An* and *SRn*, which were 16-bits and 11-bits, respectively. In order to incorporate the single-resourced adder design, the FMULT required the use of a state machine to manage the 2 additions required per the G.726 design specification [1]. In order to properly pipeline this design, the inputs to the FMULT must have 1 clock cycle between each new set of input stimulus, otherwise the pipeline will lag and the additions will fall out of sync. Figure 5.1 illustrates the timing diagram of the proposed FMULT design as well as the values driven to the adder within the FMULT, resulting in a 6-stage

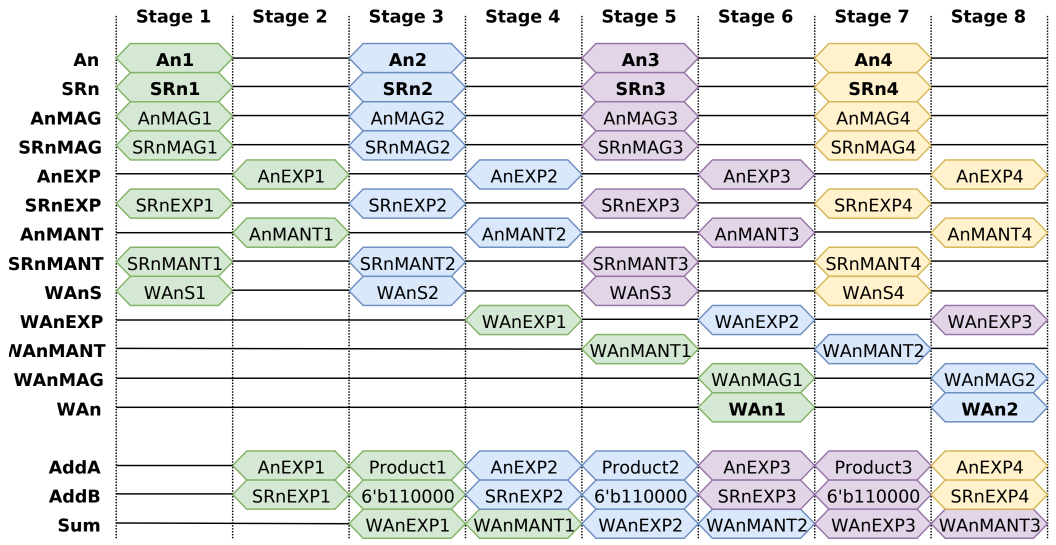


Figure 5.1: Pipelined FMULT Timing Diagram

pipeline. The design also incorporated several flip flops to maintain data values through pipeline stages (not pictured in Figure 5.1).

The state machine used in the FMULT has only 2 states, one for each of the additions. The first state adds *AnEXP* and *SRnEXP*, and the second state adds *AnMANT* * *SRnMANT* and 48. The FMULT performs the first addition during the second stage of the pipeline, and the second addition during the third stage, and will continue to go back and forth between these states during operation.

5.2 Testbench Design

The testbench follows the basic UVM testbench architecture with adjustments to the monitor, operating as the scoreboard in addition to monitoring the outputs and coverage. Also, a reference model written in C is incorporated to provide a baseline for the DUT's operation.

This section discusses each of the components used in the testbench and their functionality.

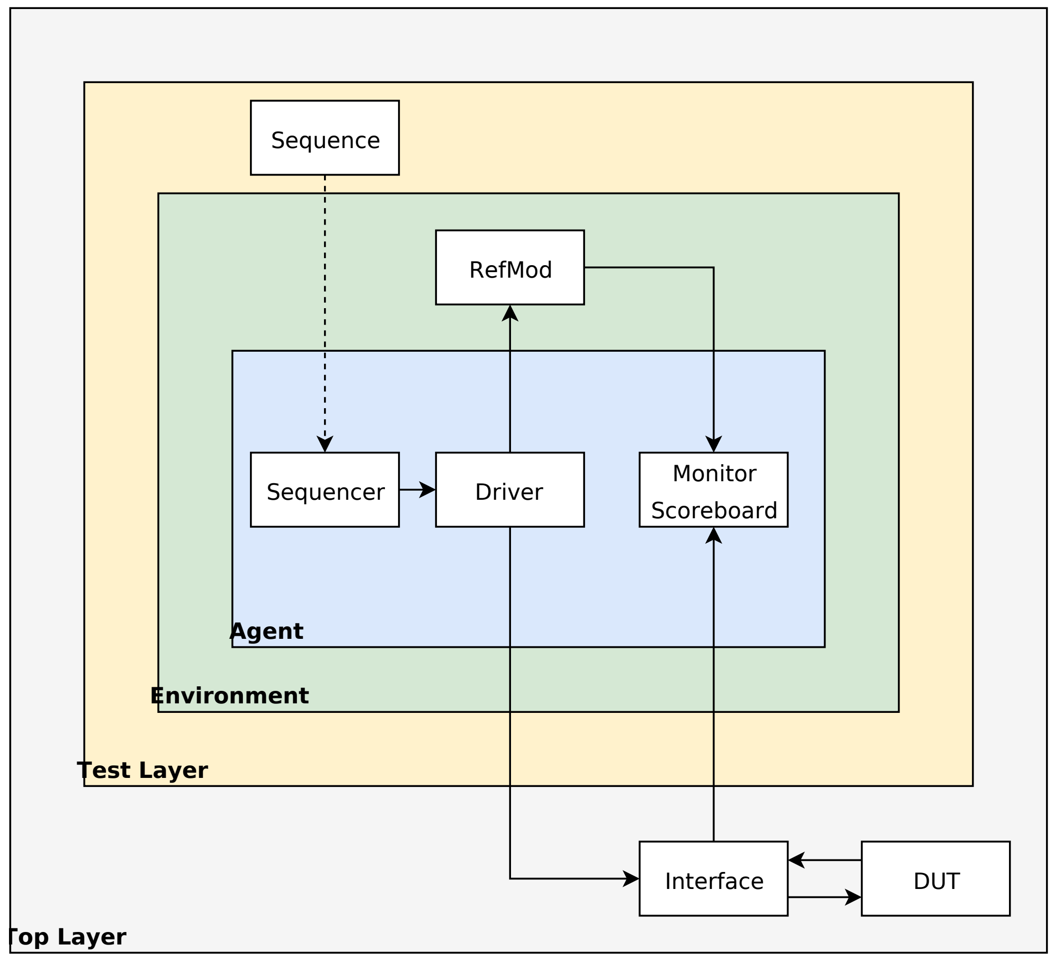


Figure 5.2: FMULT Testbench Design

5.2.1 Sequence Items

5.2.1.1 *in_sqr_item*

This sequence item is used within the *uvm_sequence* to generate random stimulus necessary to drive both the DUT and the reference model. There are 2 pieces of data sent in this transaction packet: *An* and *SRn*.

5.2.1.2 *out_sqr_item*

This sequence item is used by the reference model to send data to the monitor via transactions. There is one piece of data sent in this transaction packet: *WAn*.

5.2.2 Sequence

The sequence generates random stimulus for *SRn* and *An*. The stimulus generated is used by both the DUT and the reference model initially sent to the driver.

5.2.3 Interface

The interface contains a clock, reset, scan insertion cells and the FMULT inputs and output *An*, *SRn* and *WAn*. The clock is passed through the interface from a clock generated at the top level and is used to synchronize the testbench with the DUT.

5.2.4 Driver

The driver performs 2 primary tasks: get the transaction from the sequencer, and use the received stimulus to drive the DUT and the reference model. A *uvm_put_port* is used to send the data from the driver to the reference model, which is a layer up from the driver. In order for the driver to interact with the DUT, a virtual interface is used.

5.2.5 Monitor

The monitor, in addition to monitoring coverage and receiving the output from the DUT and reference model, also handles the comparison of data and simulation duration. The scoreboard is also included in the monitor; data is sent to the monitor from the DUT through the interface, and from the reference model via a *uvm_put_port*. The monitor does not require the use of *try_put*, and reads the data every time the FIFO is filled. However, in order to take into account the 6-stage pipeline, the monitor delays comparing values for 6 clock cycles.

The monitor also includes a *report_phase*, providing simulation information including run time, coverage, tests run and the pass rate.

The monitor serves as both the monitor and scoreboard due to the simplicity of the testbench design.

5.2.6 Agent

The agent instantiates the driver, monitor, and sequencer. The agent also handles the connection between the sequencer and the driver.

5.2.7 Environment

The environment contains an instantiation of the agent and reference model. In addition, a *connect_phase* is used to connect the driver to the reference model, and reference model to the monitor. A report phase is also used in the environment to display the number of passes and number of fails, which included the functionality of the scoreboard in addition to monitoring the outputs and coverage.

5.2.8 Test

The environment and sequence are instantiated here. The sequence used involves random stimulus generation using the variables within the *in_sqr_item*.

5.2.9 Top

The top level entity instantiates the interface used to connect to the DUT, resets it, and also drives the clock. Within the top level, the test to run is also chosen.

5.2.10 DPI Functions

DPI functions, written in C, are implemented in the monitor to keep track of wall time for the simulation, generate a text report, and email the results. This proved to be a valuable tool to keep track of test results.

5.2.11 Watermark

A configuration file is used to determine how many random stimulus will be generated and checked by the sequencer, and the monitor keeps track of this. Once the watermark is reached, the monitor drops the objection and the simulation ends.

Chapter 6

Results and Discussion

The results of the FMULT are in this chapter. The design was simulated using both RTL and gate-level simulations, and passed for all stimulus.

6.1 RTL and Gate Level Simulation Results

The DUT was simulated using Cadence electronic design automation (EDA) tools [24]. The simulation ran until a watermark of random stimulus was met. Table 6.1 displays simulation results and timing. Tables 6.2 and 6.3 show the coverage results for RTL and gate-level, respectively.

The ultimate goal is to achieve 100% functional coverage, and when using random stimulus this is typically seen with higher test runs. Because *An* is a 16-bit number, there are 65,356 possible combinations for the randomly generated input. Therefore, at least 65,356 test runs would be required, assuming the random stimulus hit each combination once. 100,000 cases was not sufficient to reach full coverage, but using a watermark of 1,000,000 or higher attained 100% functional coverage. Figure 6.1 illustrates the relationship between watermark. and coverage results for RTL, and Figure 6.2 for gate.

Table 6.1: Simulation Results

	RTL		Gate-Level	
Test Runs	Passing Rate	Wall Time (s)	Passing Rate	Wall Time (s)
10	100%	0.0069	100%	0.058
100	100%	0.069	100%	0.146
1000	100%	0.454	100%	0.6140
10000	100%	1.313	100%	2.009
100000	100%	11.108	100%	16.117
1000000	100%	93.140	100%	150.133
10000000	100%	918.027	100%	1489.162

Table 6.2: RTL Simulation Coverage Results

Test Runs	Code Coverage	Functional Coverage	An Coverage	SRn Coverage
10	90.27%	50.15%	0.02%	0.59%
100	92.35%	51.28%	0.16%	4.98%
1000	95.22%	59.99%	1.52%	38.13%
10000	96.15%	78.29%	13.90%	99.27%
100000	96.15%	94.29%	77.14%	100%
1000000	96.15%	100%	100%	100%
10000000	96.15%	100%	100%	100%

Table 6.3: Gate-Level Simulation Coverage Results

Test Runs	Code Coverage	Functional Coverage	An Coverage	SRn Coverage
10	95.92%	50.15%	0.02%	0.59%
100	96.94%	51.28%	0.16%	4.98%
1000	96.94%	60.12%	1.52%	38.96%
10000	96.94%	78.29%	13.90%	99.27%
100000	96.94%	94.31%	77.25%	100%
1000000	96.94%	100%	100%	100%
10000000	96.94%	100%	100%	100%

Another important factor in simulation is timing. The simulation ran fairly quickly, but higher watermarks required more time to be allotted for the conclusion of the simulation. Figure 6.3 shows the relationship between watermark and time, and Table 6.1 includes the run time for each watermark.

While the testbench is able to verify the behavior of the design, a c model with the desired operation was required to verify the correctness of the DUT. Each random test stimulus was processed by a C-model and the DUT, and each test passed for every test set, which did not require significant processing time.

6.2 RTL and Gate Level Synthesis Results

The FMULT was synthesized and simulated for gate sizes of 32 nm, 65 nm, 90 nm and 180 nm using Synopsys design compiler [25]. The synthesis results are recorded in Table 6.4. Figure 6.4 displays the area per gate size, and Figure 6.5 shows the number of gates as well.

Table 6.4: Synthesis Results

		Gate Size			
Category	Component	32 nm	65 nm	90 nm	180 nm
Area (μm^2)	Combinational Area	1084.432	1111.320	4382.208	8325.979
	Buff/Inv Area	80.310	65.520	387.072	578.794
	Non-Comb Area	724.31	867.24	2984.141	6566.314
	Total Cell Area	1808.743	1978.560	7366.349	14892.293
	Gate Count	1186	1374	1332	1492
Power	Internal Power (μW)	34.239	53.800	33.417	95.900
	Switching Power (μW)	3.272	4.980	16.126	409.000
	Leakage Power (nW)	$1.650 * 10^{11}$	82.018	$3.190 * 10^{10}$	80.464
	Total Power (μW)	202.086	58.900	81.430	505.000
Coverage	Test Coverage	99.98%	99.97%	99.98%	99.98%
Timing	Worst Path Delay (ns)	19.805	19.730	19.840	19.496

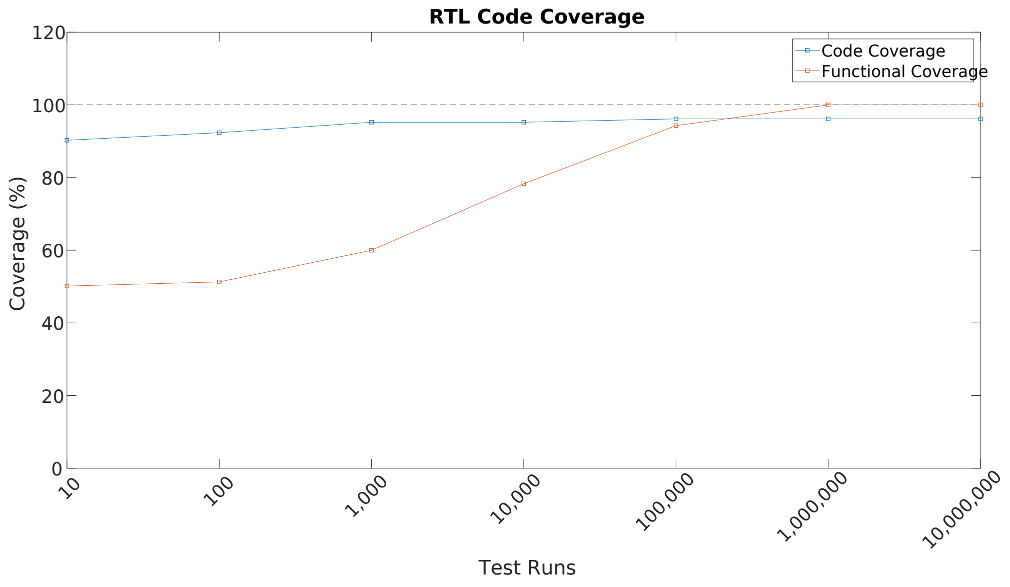


Figure 6.1: RTL Code Coverage

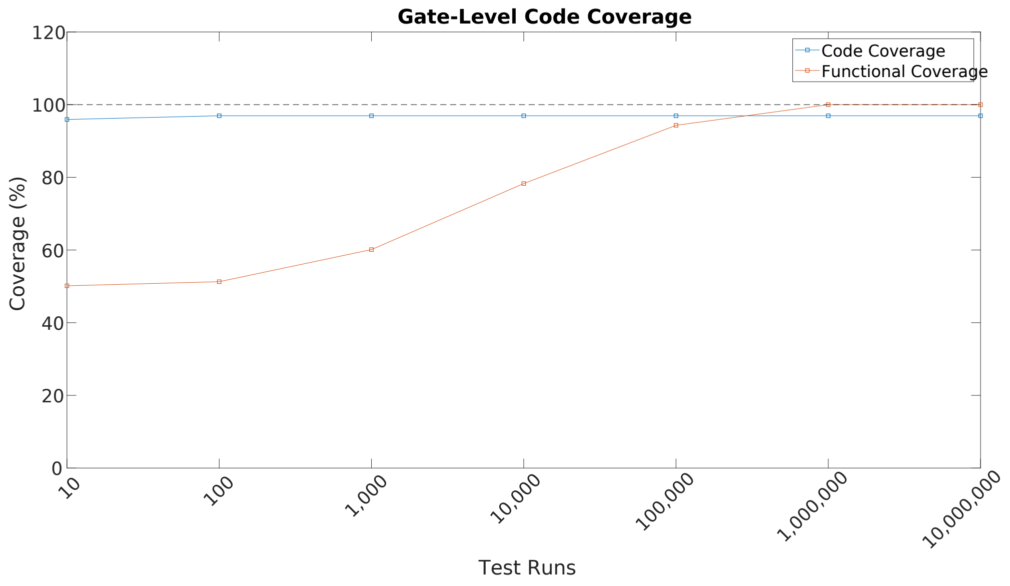


Figure 6.2: Gate-Level Code Coverage

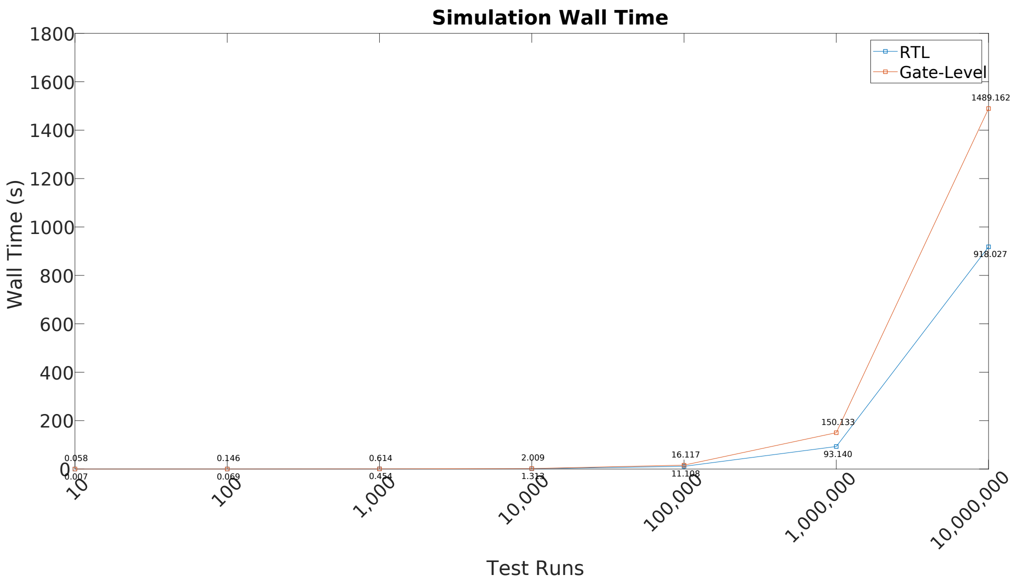


Figure 6.3: Wall Time vs. Watermark

6.3 Discussion

This testbench design using UVM was able to verify the functionality of the FMULT. Several points can be observed following verification:

1. The DUT did not fail for any test set of random stimulus
2. Higher watermarks/test runs require exponentially more time to complete
3. As the watermark increased, the functional coverage also increased
4. A watermark of around 1,000,000 is needed for the design to reach 100% functional coverage
5. As the gate size decreased, the area of the device also decreased as expected

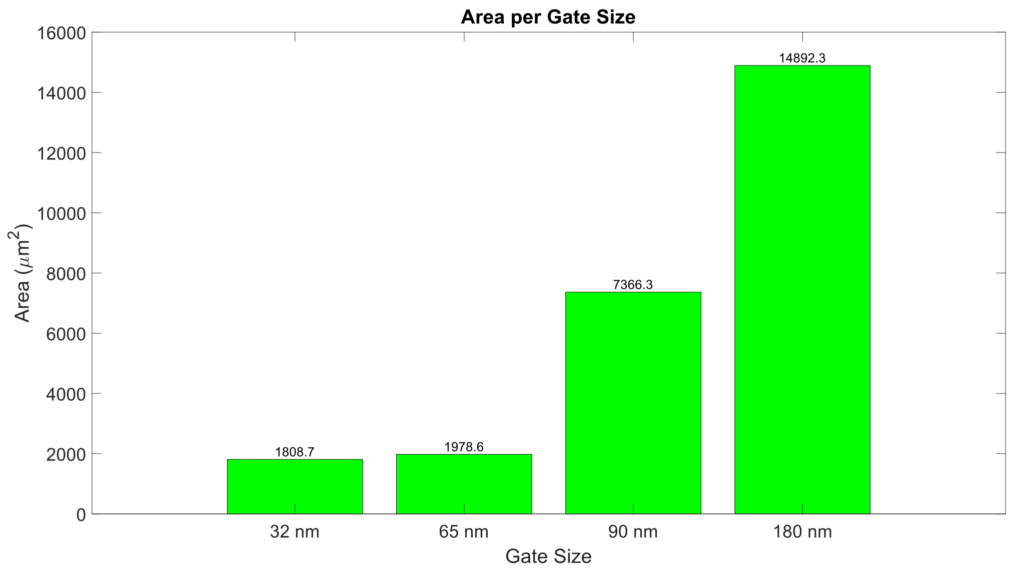


Figure 6.4: Area Per Gate Size

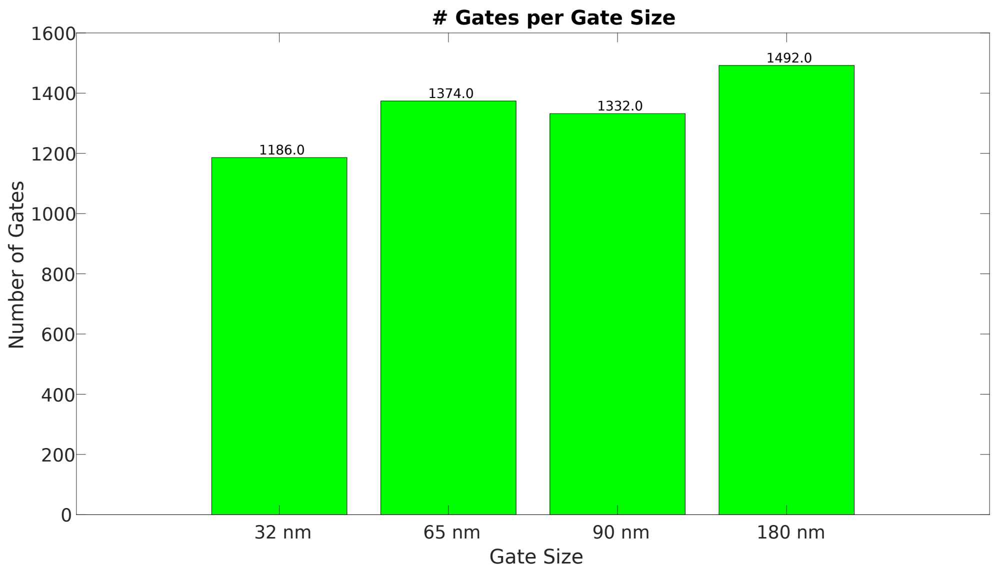


Figure 6.5: Number of Gates Per Gate Size

Chapter 7

Conclusion

The FMULT was successfully verified using UVM and a multi-layered testbench approach. The testbench was able to achieve 100% functional coverage at watermarks exceeding 1,000,000, thoroughly verifying the design. The approach and results are documented in the previous chapters. The FMULT was tested using random input stimulus and the outputs were compared with a reference model written in C, in which the FMULT passed for every test set.

7.1 Future Work

The testbench structure provided proved to be a useful and efficient form of verification for the FMULT. However, this approach is not limited to only the FMULT. Suggestions to continue the work presented in this paper are below:

- The FMULT is one of several components within the APRSC. This verification approach can be used to test the remaining low-level components, as well as the APRSC
- The RTL can be redesigned without a single-resource adder, allowing the design to be pipelined with new stimulus every clock cycle

- An implementation using UVM and the G.726 test sequences specification [26] would be valuable for verifying the ADPCM

References

[1] 40, 32, 24, 16 kbit/s ADAPTIVE DIFFERENTIAL PULSE CODE MODULATION (ADPCM). Recommendation G.726. *International Telecommunications Union (ITU)*, 1990.

[2] W. Ramirez, H. Gomez, and E. Roa. On UVM Reliability in Mixed-Signal Verification. In *2019 IEEE 10th Latin American Symposium on Circuits Systems (LASCAS)*, pages 233–236, Feb 2019. doi:10.1109/LASCAS.2019.8667543.

[3] E. M. Hamed, K. Salah, A. H. Madian, and A. G. Radwan. An Automated Lightweight UVM Tool. In *2018 30th International Conference on Microelectronics (ICM)*, pages 136–139, Dec 2018. doi:10.1109/ICM.2018.8704037.

[4] G. Renuka, V. Ushashree, and P. C. Reddy. Verification of communication based SoC using advanced verification methodology. In *2016 International Conference on Communication and Signal Processing (ICCSP)*, pages 0010–0013, April 2016. doi:10.1109/ICCSP.2016.7754162.

[5] Y. Oh and Gi-Yong Song. Simple hardware verification platform using SystemVerilog. In *TENCON 2011 - 2011 IEEE Region 10 Conference*, pages 1414–1417, Nov 2011. doi:10.1109/TENCON.2011.6129042.

[6] IEEE Approved Draft Standard for System Verilog–Unified Hardware Design, Specification, and Verification Language. *IEEE P1800/D6, August 2012*, pages 1–1312, Dec 2012.

[7] G. Tumbush and Chris Spear. *SystemVerilog for Verification, Third Edition: A Guide to Learn- ing the Testbench Language Features*. Springer, 2012.

[8] H. El-Kharashy, M. Khami, A. Sala, and M. Korany. A novel assertions-based code coverage automatic CAD tool. In *IEEE EUROCON 2017 -17th International Conference on Smart Technologies*, pages 277–281, July 2017. doi:10.1109/EUROCON.2017.8011119.

[9] S. Das, R. Mohanty, P. Dasgupta, and P. P. Chakrabarti. Synthesis of System Verilog Assertions. In *Proceedings of the Design Automation Test in Europe Conference*, volume 2, pages 1–6, March 2006. doi:10.1109/DATE.2006.243776.

[10] R. Sebastian, S. R. Mary, M. Gayathri, and A. Thomas. Assertion based verification of SGMII IP core incorporating AXI Transaction Verification Model. In *2015 International Conference on Control Communication Computing India (ICCC)*, pages 585–588, Nov 2015. doi:10.1109/ICCC.2015.7432965.

[11] J. Sanguinetti and E. Zhang. The relationship of code coverage metrics on high-level and RTL code. In *2010 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 138–141, June 2010. doi:10.1109/HLDVT.2010.5496649.

[12] A. Tran, M. Smith, and J. Miller. A Hardware-Assisted Tool for Fast, Full Code Coverage Analysis. In *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, pages 321–322, Nov 2008. doi:10.1109/ISSRE.2008.22.

[13] IEEE Standard for Universal Verification Methodology Language Reference Manual. *IEEE Std 1800.2-2017*, pages 1–472, May 2017. doi:10.1109/IEEESTD.2017.7932212.

[14] Ray Salemi. *The UVM Primer: An Introduction to the Universal Verification Methodology*. Number 4. Boston Light Press, 2013.

[15] Universal Verification Methodology (UVM) 1.2 Class Reference. pages 1–938, June 2014.

[16] K. Salah. A UVM-based smart functional verification platform: Concepts, pros, cons, and opportunities. In *2014 9th International Design and Test Symposium (IDT)*, pages 94–99, Dec 2014. doi:10.1109/IDT.2014.7038594.

[17] P. Cummiskey, N. S. Jayant, and J. L. Flanagan. Adaptive quantization in differential PCM coding of speech. *The Bell System Technical Journal*, 52(7):1105–1118, Sep. 1973. doi:10.1002/j.1538-7305.1973.tb02007.x.

[18] J. R. Boddie, J. D. Johnston, C. A. McGonegal, J. W. Upton, D. A. Berkley, R. E. Crochiere, and J. L. Flanagan. Digital signal processor: Adaptive differential pulse-code-modulation coding. *The Bell System Technical Journal*, 60(7):1547–1561, Sep. 1981. doi:10.1002/j.1538-7305.1981.tb00283.x.

[19] Pulse Code Modulation (PCM) of Voice Frequencies ITU-T Recommendation G.711. *International Telecommunication Union (ITU)*, 1988.

[20] Takenori Yoshimura, Kei Hashimoto, Keiichiro Oura, Yoshihiko Nankaku, and Keiichi Tokuda. Speaker-dependent Wavenet-based Delay-free ADPCM Speech Coding. *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2019. doi:10.1109/icassp.2019.8682264.

[21] N. S. Jayant. Digital coding of speech waveforms: PCM, DPCM, and DM quantizers. *Proceedings of the IEEE*, 62(5):611–632, 1974. doi:10.1109/proc.1974.9484.

[22] ITU-T Recommendation G.722, 7 kHz Audio-Coding Within 64 kbit/s. *ITU-T*, Nov 1988.

[23] P. Cummiskey, N. S. Jayant, and J. L. Flanagan. Adaptive Quantization in Differential PCM Coding of Speech. *The Bell System Technical Journal*, 1973.

[24] Synopsys Design Compiler, 2019. URL: https://www.cadence.com/.

[25] Cadence EDA Tools, 2019. URL: https://www.synopsys.com/.

[26] Description of the Digital Test Sequences for the Verification of the G.726 40, 32, 24 and 16 kbit/s ADPCM Algorithm. 1990.

Appendix I

Source Code

I.1 FMULT Design

```
module FMULT (
                reset ,
                clk ,
                scan_in0 ,
                scan_en ,
                test_mode ,
                scan_out0 ,
       An,
       SRn,
                WAn
          );

input
```

```
    reset,                      // system reset
    clk;                        // system clock
input wire [15:0]
    An;
input wire [10:0]          // ALSO Bn; Memory Input
    SRn;       // Reconstructed Signal Input
input
    scan_in0,                   // test scan mode data input
    scan_en,                    // test scan mode enable
    test_mode;                  // test mode select

output
    scan_out0;                  // test scan mode data output
output reg [15:0]
    WAn;       // Partial Product/Signal Estimate Output

wire [13:0] AnMAG;
reg  [3:0]  AnEXP;
reg  [5:0]  AnMANT;

wire [3:0] SRnEXP;
wire [5:0] SRnMANT;

reg [11:0] SRnAnMult;

```

```
wire WAnS;
reg  WAnS1,WAnS2,WAnS3,WAnS4;
reg  [4:0] WAnEXP,WAnEXP1,WAnEXP2,WAnEXP3;
reg  [7:0] WAnMANT;
reg [15:0] WAnMAG;

reg  [11:0] A;
reg   [5:0] B;
reg  state;
wire [11:0] SUM;

//parameter STATE1 = 0;
//parameter STATE2 = 1;

// Adder
assign SUM     = A + B;

// SRnEXP and SRnMANT Calc
assign SRnEXP  = SRn[9:6];
assign SRnMANT = SRn[5:0];

// WAnS Calc
assign WAnS = SRn[10] ^ An[15];

// AnMAG Calc
```

```
assign AnMAG = An[15] ? (16'd16384 - (An[15:2])) & 14'd8191 :
    An[15:2];

// Pipeline Stage 1 - ASynchronous Calculations
always@(posedge clk or posedge reset)
begin
      if(reset) begin
   AnEXP     = 4'b0000;
   AnMANT   <= 6'b000000;
      end

      else begin

   // AnEXP and AnMANT Calculations
   casez(AnMAG)
   13'b0000000000000: begin
         AnEXP   = 4'b0000;
         AnMANT <= 6'b100000;
         end
   13'b0000000000001: begin
         AnEXP   = 4'b0001;
         AnMANT <= ({AnMAG[13:0],6'b000000})>>AnEXP;
         end
   13'b000000000001?: begin
         AnEXP   = 4'b0010;
```

```
          AnMANT <= ({AnMAG[13:0],6'b000000})>>AnEXP;
          end
     13'b00000000001??: begin
          AnEXP  = 4'b0011;
          AnMANT <= ({AnMAG[13:0],6'b000000})>>AnEXP;
          end
     13'b0000000001???: begin
          AnEXP  = 4'b0100;
          AnMANT <= ({AnMAG[13:0],6'b000000})>>AnEXP;
          end
     13'b000000001????: begin
          AnEXP  = 4'b0101;
          AnMANT <= ({AnMAG[13:0],6'b000000})>>AnEXP;
          end
     13'b00000001?????: begin
          AnEXP  = 4'b0110;
          AnMANT <= ({AnMAG[13:0],6'b000000})>>AnEXP;
          end
     13'b0000001??????: begin
          AnEXP  = 4'b0111;
          AnMANT <= ({AnMAG[13:0],6'b000000})>>AnEXP;
          end
     13'b000001???????: begin
          AnEXP  = 4'b1000;
          AnMANT <= ({AnMAG[13:0],6'b000000})>>AnEXP;
```

```
        end
    13'b00001????????: begin
        AnEXP  = 4'b1001;
        AnMANT <= ({AnMAG[13:0],6'b000000})>>AnEXP;
        end
    13'b0001?????????: begin
        AnEXP  = 4'b1010;
        AnMANT <= ({AnMAG[13:0],6'b000000})>>AnEXP;
        end
    13'b001??????????: begin
        AnEXP  = 4'b1011;
        AnMANT <= ({AnMAG[13:0],6'b000000})>>AnEXP;
        end
    13'b01???????????: begin
        AnEXP  = 4'b1100;
        AnMANT <= ({AnMAG[13:0],6'b000000})>>AnEXP;
        end
    13'b1????????????: begin
        AnEXP  = 4'b1101;
        AnMANT <= ({AnMAG[13:0],6'b000000})>>AnEXP;
        end
    endcase
      end
end

```

```

// Pipeline Stage 2 - Adder (1st Iteration)
always@(posedge clk or posedge reset)
begin
    if (reset) begin
  SRnAnMult <= 11'b00000000000;
  WAnS1       <= 16'b0000000000000000;
          //state       <= STATE1;
          //save_state  <= STATE1;
    end
    else begin
          //save_state  <= ~save_state;
          //state <= save_state;
  SRnAnMult <= SRnMANT * AnMANT;
  WAnS1       <= WAnS;
    end
end


// Pipeline Stage 3,4 - Adder (2nd Iteration)
always@(posedge clk or posedge reset)
begin
    if (reset) begin
  WAnS2     <= 16'b0000000000000000;
  WAnS3     <= 16'b0000000000000000;
  WAnS4     <= 16'b0000000000000000;
  WAnEXP1  <= 5'b00000;
  WAnEXP2  <= 5'b00000;
  WAnEXP3  <= 5'b00000;
    end
    else begin
  WAnEXP1  <= WAnEXP;
  WAnEXP2  <= WAnEXP1;
  WAnEXP3  <= WAnEXP2;
  WAnS2     <= WAnS1;
  WAnS3      <= WAnS2;
  WAnS4     <= WAnS3;
    end
end


// Pipeline Stage 5 - Output
always@(posedge clk or posedge reset)
begin
    if (reset) begin
  WAnMAG = 16'b0000000000000000;
  WAn      <= 16'b0000000000000000;
         state <= 1'b1;
    end
    else begin
```

```
            state <= ~state;
  if (WAnEXP1 <= 26) begin
        WAnMAG = ({WAnMANT[7:0],7'b0000000})>>(6'd26-WAnEXP1);
  end
  else begin
        WAnMAG = (({WAnMANT[7:0],7'b0000000})<<(WAnEXP1 - 6'd26))
              & 16'b0111111111111111;
  end
  WAn <= WAnS4 ? (17'd65536 - WAnMAG) : WAnMAG;
     end
end


// Pipeline -> Adder Block
always@(posedge clk or posedge reset)
begin
     if (reset) begin
  A            <= 11'b00000000000;
  B            <= 6'b000000;
  WAnMANT      <= 8'b00000000;
  WAnEXP       <= 5'b00000;
     end
     else begin
           casez(state)
        1'b0: begin
        WAnEXP  <=  SUM;
        A           <=  SRnAnMult;
        B           <=  6'b110000;
           end
           1'b1:  begin
        WAnMANT  <=  SUM[11:4];
           A           <=  SRnEXP;
           B           <=  AnEXP;
           end
     endcase
        end
end

endmodule
```

I.2 Interface

```
interface intf(input clk);
  logic reset;
  logic scan_in0;
  logic scan_en;
  logic test_mode;
  logic scan_out0;
  logic [15:0] An;
  logic [10:0] SRn;
  logic [15:0] WAn;

endinterface: intf
```

I.3 Input Sequence Item

```
class in_sqr_item extends uvm_sequence_item;
  rand logic [15:0] An;
  rand logic [10:0] SRn;

  `uvm_object_utils_begin(in_sqr_item)
    `uvm_field_int(An, UVM_ALL_ON | UVM_HEX)
    `uvm_field_int(SRn, UVM_ALL_ON | UVM_HEX)
  `uvm_object_utils_end

  function new(string name = "in_sqr_item");
    super.new(name);
  endfunction

endclass: in_sqr_item
```

I.4 Output Sequence Item

```
class out_sqr_item extends uvm_sequence_item;

  logic [15:0] WAn;

  `uvm_object_utils_begin(out_sqr_item)
    `uvm_field_int(WAn, UVM_ALL_ON | UVM_HEX)
  `uvm_object_utils_end

  function new(string name = "out_sqr_item");
    super.new(name);
  endfunction

endclass: out_sqr_item
```

I.5 Reference Model

```c
#include <stdio.h>
#include <math.h>

extern int
fmult(x, y1)
int x, y1;
{
  //printf("REFMOD Inputs %x, %x\n",x,y1);
  int xs, xmag, xexp, xmant;
  int ys, yexp, ymant;
  int wxs, wx, wxexp, wxmant, wxmag, i;

  xs = (x >> 15) & 1;
  xmag = xs ? (16384 - (x >> 2)) & 8191 : x >> 2;
  xexp = 0;
  for (i = 0; i <= 13; i += 1) {
    if (!(xmag >> i)) {
      xexp = i;
      break;
    }
    if (i == 13)
      printf("mag didn't get set in fmult\n");
  }
```

```
    xmant = xmag ? (xmag << 6) >> xexp : 1 << 5;

    ys = (y1 >> 10) & 1;
    yexp = (y1 >> 6) & 15;
    ymant = y1 & 63;

    wxs = ys ^ xs;
    wxexp = yexp + xexp;
    wxmant = ((ymant * xmant) + 48) >> 4;
    wxmag = wxexp > 26 ? ((wxmant << 7) << (wxexp - 26)) & 32767
           : (wxmant << 7 ) >> ( 26 - wxexp);
    wx = wxs ? (65536 - wxmag) & 65535 : wxmag & 65535;

    return(wx);
}
```

I.6 Sequencer

```
class seq_in extends uvm_sequence #(in_sqr_item);
    `uvm_object_utils(seq_in)
    int An,SRn;
    function new(string name="seq_in");
        super.new(name);
    endfunction: new

    task body;
        in_sqr_item tx;

        forever begin
            tx = in_sqr_item::type_id::create("tx");
            start_item(tx);
            assert(tx.randomize());
            finish_item(tx);
        end
    endtask: body
endclass: seq_in
```

I.7 Driver

```
typedef virtual intf intf_vif;

class driver extends uvm_driver #(in_sqr_item);

  `uvm_component_utils(driver)

  uvm_put_port #(in_sqr_item) icp;
  intf_vif vif;

  event begin_record,end_record;

  covergroup cov_in;
    dut_An   : coverpoint vif.An
            {bins An[] = {[0:65535]};}
    dut_SRn : coverpoint vif.SRn
            {bins SRn[] = {[0:2047]};}
  endgroup

  function new(string name = "driver",uvm_component parent =
      null);
    super.new(name,parent);
    icp    = new("icp",this);
    cov_in = new();
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    void'(uvm_resource_db#(intf_vif)::read_by_name(.scope("ifs"
        ),.name("intf_vif"),.val(vif)));
  endfunction

  virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    fork
      reset_signals();
      drive(phase);
    join
  endtask

  virtual protected task reset_signals();
    forever begin
      vif.reset     = 1;
      vif.scan_in0  = 0;
      vif.scan_en   = 0;
      vif.test_mode = 0;
     @(negedge vif.clk);
     @(negedge vif.clk);
     @(negedge vif.clk);
```

```
      @(negedge vif.clk);
       vif.reset = 0;
      @(posedge vif.reset);
    end
  endtask

  virtual protected task drive(uvm_phase phase);
    $display("Waiting");
    wait(vif.reset === 1);
    $display("Reset Asserted");
   @(negedge vif.reset);
    $display("Reset De-asserted");
   @(posedge vif.clk);
    forever begin
      seq_item_port.get(req);
      -> begin_record;
      write_it(req);
    end
  endtask

  virtual protected task write_it(in_sqr_item tr);
    vif.An  <= tr.An;
    vif.SRn <= tr.SRn;
    //$display("An  -> %d\nSRn -> %d\n",tr.An,tr.SRn);
    icp.put(tr);
    @(posedge vif.clk)
    @(posedge vif.clk)
    cov_in.sample();
    -> end_record;
  endtask

  virtual task record_tr();
    forever begin
      @(begin_record);
      begin_tr(req,"driver");
      @(end_record);
      end_tr(req);
    end
  endtask

endclass
```

I.8 Monitor

```
class monitor #(type T = out_sqr_item) extends uvm_monitor;
  typedef monitor #(T) this_type;
  `uvm_component_param_utils(this_type)

  const static string type_name = "monitor #(T)";

  uvm_put_imp #(T, this_type) from_rm;
  intf_vif vif;
  in_sqr_item tr;
  out_sqr_item exp;

  int start_time,run_time;
  int hrs, min, sec;
  int watermark,wmfile,wmstring;

  logic free;
  int count,num_matches,num_mismatches;
  event begin_delay,end_delay,endsimulation,compared;

  covergroup cov_out;
    DUT_0  : coverpoint vif.WAn[0];
    DUT_1  : coverpoint vif.WAn[1];
    DUT_2  : coverpoint vif.WAn[2];
    DUT_3  : coverpoint vif.WAn[3];
    DUT_4  : coverpoint vif.WAn[4];
    DUT_5  : coverpoint vif.WAn[5];
    DUT_6  : coverpoint vif.WAn[6];
    DUT_7  : coverpoint vif.WAn[7];
    DUT_8  : coverpoint vif.WAn[8];
    DUT_9  : coverpoint vif.WAn[9];
    DUT_10 : coverpoint vif.WAn[10];
    DUT_11 : coverpoint vif.WAn[11];
    DUT_12 : coverpoint vif.WAn[12];
    DUT_13 : coverpoint vif.WAn[13];
    DUT_14 : coverpoint vif.WAn[14];
    DUT_15 : coverpoint vif.WAn[15];

    REF_0  : coverpoint exp.WAn[0];
    REF_1  : coverpoint exp.WAn[1];
    REF_2  : coverpoint exp.WAn[2];
    REF_3  : coverpoint exp.WAn[3];
    REF_4  : coverpoint exp.WAn[4];
    REF_5  : coverpoint exp.WAn[5];
    REF_6  : coverpoint exp.WAn[6];
    REF_7  : coverpoint exp.WAn[7];
    REF_8  : coverpoint exp.WAn[8];
    REF_9  : coverpoint exp.WAn[9];
    REF_10 : coverpoint exp.WAn[10];
    REF_11 : coverpoint exp.WAn[11];
    REF_12 : coverpoint exp.WAn[12];
    REF_13 : coverpoint exp.WAn[13];
    REF_14 : coverpoint exp.WAn[14];
    REF_15 : coverpoint exp.WAn[15];

    CC_0   : cross DUT_0, REF_0
    {bins pass = binsof(DUT_0) && binsof(REF_0);}
    CC_1   : cross DUT_1, REF_1
    {bins pass = binsof(DUT_1) && binsof(REF_1);}
    CC_2   : cross DUT_2, REF_2
    {bins pass = binsof(DUT_2) && binsof(REF_2);}
    CC_3   : cross DUT_3, REF_3
    {bins pass = binsof(DUT_3) && binsof(REF_3);}
    CC_4   : cross DUT_4, REF_4
    {bins pass = binsof(DUT_4) && binsof(REF_4);}
    CC_5   : cross DUT_5, REF_5
    {bins pass = binsof(DUT_5) && binsof(REF_5);}
    CC_6   : cross DUT_6, REF_6
    {bins pass = binsof(DUT_6) && binsof(REF_6);}
    CC_7   : cross DUT_7, REF_7
    {bins pass = binsof(DUT_7) && binsof(REF_7);}
    CC_8   : cross DUT_8, REF_8
    {bins pass = binsof(DUT_8) && binsof(REF_8);}
    CC_9   : cross DUT_9, REF_9
    {bins pass = binsof(DUT_9)  && binsof(REF_9);}
    CC_10  : cross DUT_10, REF_10
    {bins pass = binsof(DUT_10)  && binsof(REF_10);}
    CC_11  : cross DUT_11, REF_11
    {bins pass = binsof(DUT_11)  && binsof(REF_11);}
    CC_12  : cross DUT_12, REF_12
    {bins pass = binsof(DUT_12)  && binsof(REF_12);}
    CC_13  : cross DUT_13, REF_13
    {bins pass = binsof(DUT_13)  && binsof(REF_13);}
    CC_14  : cross DUT_14, REF_14
    {bins pass = binsof(DUT_14)  && binsof(REF_14);}
    CC_15  : cross DUT_15, REF_15
    {bins pass = binsof(DUT_15)  && binsof(REF_15);}
  endgroup

  function new(string name, uvm_component parent);
    super.new(name,parent);
    from_rm  = new("from_rm",this);
    exp      = new("exp");
    cov_out  = new();
    count          = 0;
    num_mismatches = 0;
    num_matches    = 0;
    free           = 0;
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    void'(uvm_resource_db#(intf_vif)::read_by_name(.scope("ifs"
        ),
          .name("intf_vif"), .val(vif)));
    tr = in_sqr_item::type_id::create("tr",this);
  endfunction

  virtual function string get_type_name();
    return type_name;
  endfunction

  virtual function int get_watermark();
    wmfile = $fopen("src/watermark.param","r");
    wmstring = $fscanf(wmfile,"%d",watermark);
    if (watermark == "") return 0;
    else $display("Running to Watermark of: %d",watermark);
    return 1;
  endfunction

  virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    super.run_phase(phase);
    fork
```

```
      compare_transactions(phase);
      end_sim(phase);
    join
  endtask

  virtual task end_sim(uvm_phase phase);
    @(endsimulation)
    phase.drop_objection(this);
  endtask

  virtual task put (out_sqr_item t);
    exp.copy(t);
  endtask

  virtual function bit try_put(out_sqr_item t);
    exp.copy(t);
    return 1;
  endfunction

  virtual function bit can_put();
    return free;
  endfunction

  virtual task compare_transactions(uvm_phase phase);
    a1 : assert (get_watermark() == 1);
    start_time = get_time();
    wait(vif.reset === 1);
   @(negedge vif.reset);

   -> begin_delay;
   do begin
     @(negedge vif.clk);
      count++;
   end while (count != 6);
   -> end_delay;

    forever begin
     @(negedge vif.clk);
      if (exp.WAn !== vif.WAn) begin
        num_mismatches++;
        uvm_report_warning(get_type_name(),$sformatf("Output
           Mismatch RM: %h DUT: %h (%f mismatches)",exp.WAn, vif.
           WAn, num_mismatches),UVM_NONE);
      end
      else begin
        //uvm_report_info(get_type_name(),$sformatf("Output
           match RM: %h DUT: %h (%f mismatches)",exp.WAn, vif.WAn
           , num_mismatches),UVM_NONE);
        num_matches++;
      end
```

```
        cov_out.sample();
        @(negedge vif.clk);
        -> compared;

// Uncomment for Coverage-Based functionality instead of
   Watermark
/*
        if (((num_matches + num_mismatches) % 10000) == 0) begin
  $display("%d Runs",num_matches + num_mismatches);
  if($get_coverage() >= 100) begin
            -> endsimulation;
          end
        end
*/
// Uncomment for Watermark-Based Functionality instead of
   Coverage
///*
        if((num_matches + num_mismatches) == watermark) begin
          -> endsimulation;
        end
//*/

      end
    endtask

    function void report_phase(uvm_phase phase);
      $display("Simulation Ended. Number of Tests: %d",
          num_matches + num_mismatches);
      $display("Total Coverage: %.2f",$get_coverage());
      $display("Num Passes: %d\nNum Fails: %d",num_matches,
          num_mismatches);
      run_time = get_time() - start_time;
      hrs = run_time / 3600;
      min = (run_time - (hrs*3600)) / 60;
      sec = (run_time - (hrs*3600) - (min*60));
      $display("Run Time: %d Hrs %d Min %d Sec",hrs,min,sec);
      generate_report(num_matches,num_mismatches,watermark,0,0,0,
          run_time,$get_coverage());
      email_report();
    endfunction

endclass
```

I.9 Agent

```
class agent extends uvm_agent;
  sequencer sqr;
  driver    dvr;
  monitor   #(out_sqr_item) mtr;

  uvm_put_port #(in_sqr_item) icp;

  `uvm_component_utils(agent)

  function new(string name = "agent", uvm_component parent =
     null);
    super.new(name,parent);
    icp = new("icp",this);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    sqr = sequencer::type_id::create("sqr",this);
    dvr = driver::type_id::create("dvr",this);
    mtr = monitor#(out_sqr_item)::type_id::create("mtr",this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
```

```
      super.connect_phase(phase);
      dvr.icp.connect(icp);
      dvr.seq_item_port.connect(sqr.seq_item_export);
   endfunction

endclass
```

I.10 Environment

```
class env extends uvm_env;

  agent      mstr;
  refmod     rm;
  uvm_tlm_analysis_fifo #(in_sqr_item) to_rm;

  `uvm_component_utils(env)

  function new(string name, uvm_component parent = null);
    super.new(name,parent);
    to_rm = new("to_rm",this);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    mstr =     agent::type_id::create("mstr",this);
    rm   =    refmod::type_id::create("rm",  this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    // Sequencer to Ref Mod FIFO
    mstr.icp.connect(to_rm.put_export);

    // Ref Mod FIFO to Ref Mod
    rm.in.connect(to_rm.get_export);

    // Ref Mod to Monitor
    rm.out.connect(mstr.mtr.from_rm);
  endfunction

  virtual function void end_of_elaboration_phase(uvm_phase
     phase);
    super.end_of_elaboration_phase(phase);
  endfunction

  virtual function void report_phase(uvm_phase phase);
    super.report_phase(phase);
    `uvm_info(get_type_name(), $sformatf("Reporting Matched %0d
       ",mstr.mtr.num_matches),UVM_NONE)
    if(mstr.mtr.num_mismatches) begin
      `uvm_error(get_type_name(),$sformatf("Saw %0d mismatched
         samples", mstr.mtr.num_mismatches))
    end
  endfunction

endclass
```

I.11 Test

```
class FMULT_test extends uvm_test;
  env env_h;
  seq_in sqr_h;
  `uvm_component_utils(FMULT_test)

  function new(string name,uvm_component parent = null);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env_h = env::type_id::create("env_h",this);
    sqr_h = seq_in::type_id::create("sqr_h",this);
  endfunction

  task run_phase(uvm_phase phase);
    sqr_h.start(env_h.mstr.sqr);
  endtask

endclass
```

I.12 Top

```

`include "uvm_macros.svh"
`include "intf.sv"

`include "FMULT_pkg.sv"


module test;

  import   uvm_pkg::*;
  import FMULT_pkg::*;

  logic clk;

  intf vif(clk);

  initial begin
    clk       = 0;
    vif.reset = 0;
  end

  always #5 clk = ~clk;

```

```
   initial begin
     $timeformat(-9,2,"ns",16);
     `ifdef SDFSCAN
       $sdf_annotate("sdf/FMULT_tsmc18_scan.sdf", test.top);
     `endif
     uvm_resource_db#(intf_vif)::set(.scope("ifs"),.name("
        intf_vif"),.val(vif));
     $set_coverage_db_name("FMULT");
     run_test("FMULT_test");
   end

  FMULT top(vif.reset,
            clk,
            vif.scan_in0,
            vif.scan_en,
            vif.test_mode,
            vif.scan_out0,
            vif.An,
            vif.SRn,
            vif.WAn);

endmodule
```