Digital Design and Implementation of DCSK Modem

CND 111 Final Project.
Under Supervision of Prof. Khaled Mohamed.

Omar T. Amer[1], **Yahia A. Hatem**[2],
Mostafa M. Darwish[3] **Mennatullah Mohamed**[4]

[1]V23009946
[2]V23009902
[3]V23009976
[4]V23010369

Center of Nanoelectronics & Devices
American University in Cairo
Cairo, Egypt
12/2/2023

Preface

Symbols

Symbol	Meaning
T_s	Symbol duration.
e_k	Transmitter output.
β	Number of chaos chips in a DCSK frame, 2β =Spreading factor.
k	Chip index.
x	Chaos chip.

Abbreviations

Abbreviation	Meaning
AWGN	Additive White Gaussian Noise
ALM	Adaptive Logic Module
CSI	Channel State Information.
E2E	End-to-End.
FF	Flip-Flop.
FSM	Finite State Machine.
MAC	Multiply and Accumulate.
MCMM	Multi-corner Multi-mode
MODEM	Modulator/Demodulator.
PISO	Parallel-In Serial-Out Shift Register.
RX	Receiver.
SIPO	Serial-In Parallel-Out Shift Register.
SF	Spreading Factor.
SNR	Signal to Noise Ratio.
TX	Transmitter.
UUT	Unit Under Test.

Text Styles

Style	Meaning
SMALL CAPS	HDL Module Reference.
`Monospace_bold`	Net Reference.
Bold_Italics	State of an FSM.

Key Design Features

- Synthesizable, fully independent target-agnostic IP Core for FPGA, SoC, and ASIC.
- 32-bit input messages.
- Per-message spreading factor selection.
    1. SF4
    2. SF8
    3. SF16
    4. SF32
- System Frequency of 50 MHz.
- Data rate of 1.5 Mbit/s with SF32.
- Synchronous design with single clock.
- Provided with self-checking testbench code.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter discusses the following:

- Introduction to Differential Chaos Shift-Keying.
- Logistic Map Overview.

1.1 Introduction to DCSK

Differential chaos shift-keying (**DCSK**) is a modulation scheme based on using *chaos* to modulate a signal instead of a sinusoidal carrier. The reference signal (chaos) is sent over the channel, then after half a T_s the reference signal is modulated with the information signal and sent to the RX side. The composition of a DCSK frame can be seen in Equation. 1.1.

$$e_k = \begin{cases} x_k & \text{for } 1 < k < \beta \\ s_i x_{k-\beta} & \text{for } \beta < k \leq 2\beta \end{cases} \tag{1.1}$$

Where 2β is the spreading factor and k is the chip index.

e_k is sent over an AWGN channel, the signal at the RX side is given by:

$$r_{sig} = e_k + n_k \tag{1.2}$$

The received signal is correlated with a delayed version of itself (delayed by half a symbol's duration, which is a delay by β clock cycles) and then the sign of the result is the demodulated information bit.

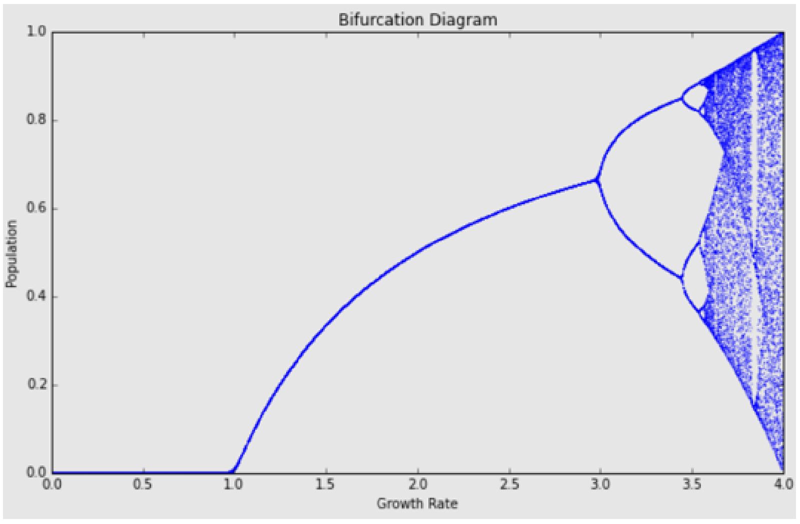
1.2 An Overview of the Logistic Map

Many methods exists to generate a chaotic signal. The Logistic Map is the most simple to understand and is the least complex to implement. The Logistic Map is given by the following equation:

$$n_{i+1} = r \times n_i(1 - n_i) \tag{1.3}$$

Most values of r beyond ≈ 3.56995 exhibit chaotic behavior. We can see in Figure 1.1 The effect of different values of r on n_{i+1}.

Figure 1.1: Bifurcation Diagram of the logistic map.



Chapter 2

Architecture

This chapter discusses the proposed architecture of following modules:

- Transmitter.
  - Chaos Generator.
  - Modulator.
  - Message Buffer.
  - Chip/Bit Counter.
  - Transmitter Finite State Machine.
- Receiver.

In this chapter, we discuss the architecture of the modem, as well as the design choices made.

2.1 Transmitter

The transmitter is split into three major parts:

1. Chaos generator.
2. Modulator.
3. FSM Control Unit.

2.1.1 Chaos Generator

This module generates the chaotic bit sequence required for modulation. It has three sub-modules:

1. 16-bit Chaotic Sequence Generator.
2. The Chaos Expander.
3. Chaos PISO.

Chaotic Sequence Generator

The chaotic sequence generator generates 16 bits of chaos using an 8 bit value and an 8 bit seed. The sequence generator we are using is the Logistic Map (See equation 1.3). A straight forward implementation of the logistic map requires two multiplications, and a subtraction. However, clever choice of r can optimize away one multiplication. We choose $r = 4$ to replace multiplication by r with two logical shift lefts. This saves area, power, and improves performance.

The multiplication is performed using radix-4 booth algorithm. The recoding of the multiplier is done in parallel to increase performance (see Figure 2.2).

The Chaos Expander

We propose an throughput-friendly algorithm to expand a 16-bit sequence into a 256-bit sequence while maintaining chaotic properties. Already-existing algorithms that can be used for expansion were considered (such as SHA256, or running two instances of MD5 in parallel) are computationally expensive. The pseudocode for our algorithm can be found below.

Using an expansion algorithm means that the chaotic sequence generator doesn't have to be always on. This saves power. This also saves area because we can generate a chaos sequence of length 16 instead of having to generate one of length 256. This also saves performance. See Algorithm 1.

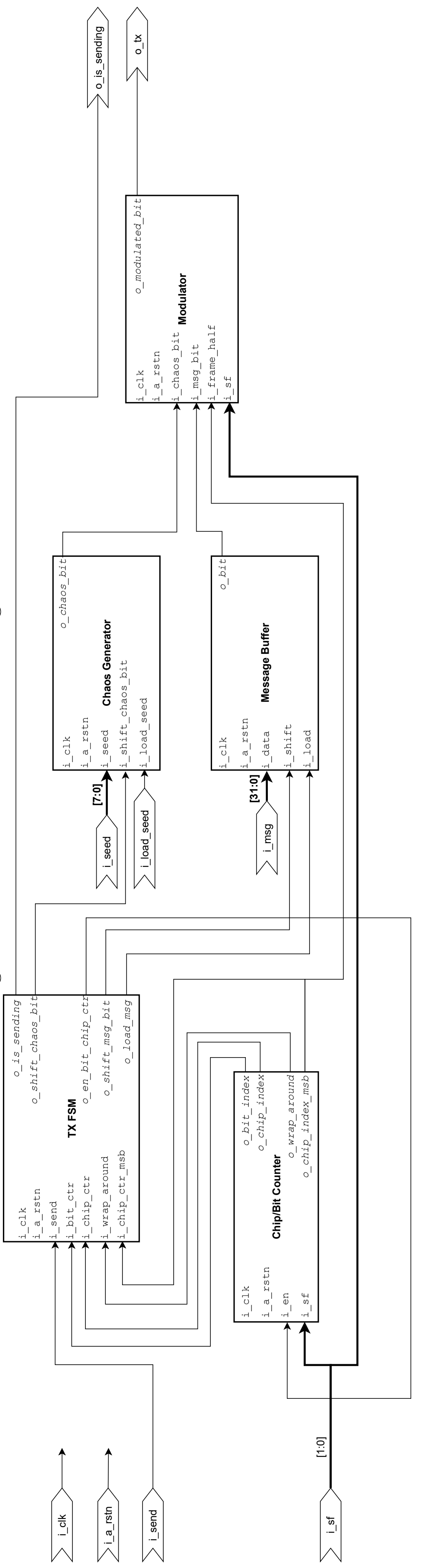
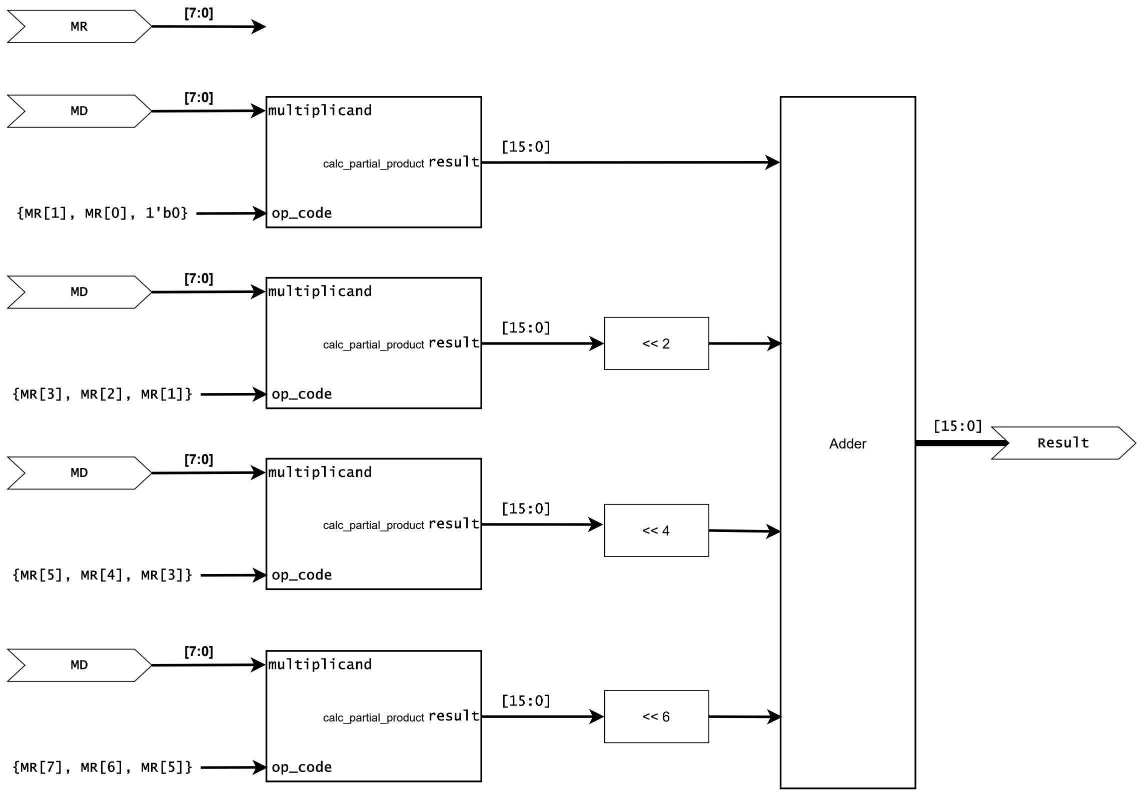


Figure 2.1: Transmitter Block Diagram

Figure 2.2: Parallel Partial Product Radix-4 Booth Multiplier



Algorithm 1 Chaos Expansion Algorithm

```
Require: $xpanded$ is a $16 \times 16$ packed array.
Require: $input$ is a 16-bit vector.
  for $i = 0;\ i < 15;\ i = i + 1$ do
    $xpanded[i] = input[i]\ ?\ input\ :\ \sim input$
  end for
  return $xpanded$
```

If the i^{th} input bit is "1", copy the input into the i^{th} pack of the expanded output. If the bit is "0", then copy the complement of the input instead. This algorithm expands the input but leaves some statistical correlation between the bits of the expanded output. This is solved by shuffling the bits before returning the final output. Each bit location is mapped to a random number which is the new location of that bit (See Appendix).

Chaos PISO

The output of the CHAOS EXPANDER is 256 bits, but for each sent bit we only need a single bit of chaos. The CHAOS PISO stores all the expanded 256 bits and outputs a single bit when requested by the FSM. This buffer has an `o_empty` signal that is asserted when all 256 bits of chaos are depleted. This signal is fed back to the module to refill the chaos bits. The chaos-refill action is independent of the FSM and is internally managed by the CHAOS GENERATOR.

Table 2.1: Chaos Generator Pin-Out

Name	Width	Direction	Description
`i_clk`	1	Input	Positive edge clock.
`i_arst_n`	1	Input	Active-low asynchronous reset.
`i_seed`	8	Input	Seed
`i_load_seed`	1	Input	Load the seed into the module.
`i_shift_chaos_bit`	1	Input	Output one bit from the CHAOS PISO
`o_chaos_bit`	1	Output	The serial output of the CHAOS PISO.

2.1.2 Modulator

The purpose of the MODULATOR is to modulate the chaos bits with the information bits. The message is first loaded into the MESSAGE PISO and is then serially output. The output bit is inverted and xor-ed with a delayed version of the chaos bit from CHAOS GENERATOR (delayed by β). The maximum delay is required when sending a message with SF32 and is equal to 16 bits. Therefore, a 16-bit shift-register is implemented and tapped at the following locations: 2, 4, 8, 16. The taps are multiplexed and the required delay value can be selected according to the spreading factor.

Table 2.2: Spreading Factor Encoding.

Spreading Factor	Encoding
SF4	00
SF8	01
SF16	10
SF32	11

Table 2.3: Modulator Pin-Out

Name	Width	Direction	Description
`i_clk`	1	Input	Positive edge clock.
`i_arst_n`	1	Input	Active-low asynchronous reset.
`i_msg_bit`	1	Input	The input message bit.
`i_chaos_bit`	1	Input	The bit of chaos to be modulated.
`i_frame_half`	1	Input	Asserted if (and only if) the index of the chip being sent is less than β.
`i_sf`	2	Input	The encoded spreading factor.
`o_modulated_bit`	1	Output	The serial output of the MODULATOR.

2.1.3 Message Buffer

The Message Buffer is a parallel-in serial-out shift register that stores the message to be sent.

Table 2.4: Message Buffer Pin-Out

Name	Width	Direction	Description
`i_clk`	1	Input	Positive edge clock.
`i_arst_n`	1	Input	Active-low asynchronous reset.
`i_data`	32	Input	Message to be sent
`i_load`	1	Input	Load the message from `i_data` into the buffer.
`i_shift`	1	Input	Shift a bit from the message into `o_bit`.
`o_bit`	1	Output	Serially-output bit.

2.1.4 Chip/Bit Counter

This counter keeps track of the index of the bit (and subsequently, the chip) that are begin sent. Since the system allows changing the SF during runtime, the maximum value of the counter is 1024 for SF32. The counter wraps around to zero based on the selected spreading factor according to Table 2.5. The chip index can be easily inferred from the bit counter with minimal logic. The maximum number of chips is 16 chips per bit. The chip index increments with each bit sent, but wraps around after 2β. We can take the least significant bits of the bit counter as the chip counter. (see Table 2.5)

Table 2.5: Chip/Bit Counter spreading factor-dependant values.

SF	Wrap-around value	Chip Index
SF4	$2 \times 32 \times 2$	`o_bit_index[1:0]`
SF8	$2 \times 32 \times 4$	`o_bit_index[2:0]`
SF16	$2 \times 32 \times 8$	`o_bit_index[3:0]`
SF32	$2 \times 32 \times 16$	`o_bit_index[4:0]`

Table 2.6: Chip/Bit Counter Pin-Out

Name	Width	Direction	Description
`i_clk`	1	Input	Positive edge clock.
`i_arst_n`	1	Input	Active-low asynchronous reset.
`i_en`	1	Input	Enable the counter.
`i_sf`	2	Input	The encoded spreading factor.
`o_bit_index`	10	Input	Index of the current bit being sent.
`o_chip_index`	5	Output	Index of the current chip being sent.
`o_chip_index_msb`	1	Output	MSB of `o_chip_index`. Adjusted for chip counter wrap-around.
`o_wrap_around`	10	Output	Index at which the bit counter wraps around.

2.1.5 Finite State Machine

The TX FSM is a two-state (single FF) FSM that controls the outputs of the modules stated above. The two states are ***IDLE*** and ***SEND***.

IDLE

The transmitter is ***IDLE*** if it is not sending anything in a given clock cycle. The transmitter can load a message into the MESSAGE BUFFER and go into the ***SEND*** state if the `i_send` input is asserted.

SEND

The transmitter is now sending a bit. The `o_is_sending` output is asserted.

State Outputs

When in the ***IDLE*** state, the TX is not doing anything, hence, its functionality is divided into two parts:

1. Transition into the ***SEND*** state when the input `i_send` is asserted.
2. if `i_send` is asserted, load the MESSAGE BUFFER with the data available on its `i_data` port.

When the TX is in the ***SEND*** it does the following:

1. Enable the bit counter (hence the chip counter).
2. Assert `o_is_sending` to signal that valid data is being sent.
3. In the first half of the frame (detected by the MSB of the bit counter) serially output the chaos bits stored in the CHAOS GENERATOR.
4. in the second half of the frame, stop putting out chaos bits.

5. in the second half of the frame, serially output the message bits if the bit counter is not zero.
6. Load a new message if the bit counter will wrap around in the next cycle and the `i_send` signal is asserted.

Table 2.7: State Transitions

Current State	Next State	Condition
IDLE	***SEND***	iff `i_send` is asserted.
SEND	***IDLE***	iff the bit counter will wrap around in the next cycle and `i_send` is not asserted

Table 2.8: State Outputs

Current State	Outputs
IDLE	All outputs are deasserted except for `o_load_msg = i_send`
SEND	`o_is_sending` and `o_en_bit_chip_ctr` are always enabled. `o_shift_chaos_bit` is asserted in the first half of the frame (`!i_chip_ctr_msb`) `o_shift_msg_bit` is asserted only after a falling edge has been detected on `i_chip_ctr_msb` and `i_bit_ctr != 0` `o_load_msg` and The bit counter will wrap around in the next cycle `i_send` is asserted.

Table 2.9: Demodulator Pin-Out

Name	Width	Direction	Description
`Clk`	1	Input	Positive edge clock.
`N_Rst`	1	Input	Active-low asynchronous reset.
`Valid`	1	Input	The TX is currently sending bits.
`Spread_Factor_Sel`	2	Input	The encoded spreading factor.
`Out_Data`	32	Output	The demodulated word.
`Valid_Data`	1	Output	A message has been demodulated successfully.

2.2 Receiver

The objective of the demodulator is to reconstruct the transmitted data bit from the received bitstream. Table 2.2 shows the pin-out of the demodulator.

2.2.1 RX FSM

The RX finite state machine has four states:

- ***IDLE***
- ***STORE_CHAOS_SEQ***
- ***CORRELATION***
- ***STORE_DEMOD_BIT***

IDLE

The demodulator is ***IDLE*** if it there is no valid traffic on the channel.

STORE_CHAOS_SEQ

The demodulator is now receiving data. Receive all chaos bits.

CORRELATION

Keep receiving. Correlate the bits being received with the chaos bits obtained earlier. Correlation is done using a simple XNOR gate.

STORE_DEMOD_BIT

Store the result of the demodulation.

Chapter 3

Testing

This chapter discusses the following topics:

- Testing Environment Architecture.
- Test Scenario Generation.

3.1 Testing Environment Architecture

The test we are running is an E2E test with randomized scenario generation. In this section we discuss the test environment in full detail.

The environment consists of a *monitor* and a *driver*, as well as out UUTs: the TX and the RX modules. The main workings of the environment can be explained as follows:

1. Random data is generated to be sent as a message.
2. A random spreading factor is selected from a the list of all valid spreading factors.
3. The message is *driven* to the TX and thus sent. The message is also pushed in the *tx queue*.
4. Iterate over the last three steps for a random amount of iterations.
5. Wait for a random amount of clock cycles. This simulates communication in bursts.
6. When the RX finishes receiving a message, its `Valid_Data` output is asserted.
7. When the *monitor* detects a rising edge on said output, the *tx_queue* is popped and the popped message is compared with the `Out_Data` output of the RX. A match means that the system is operational.

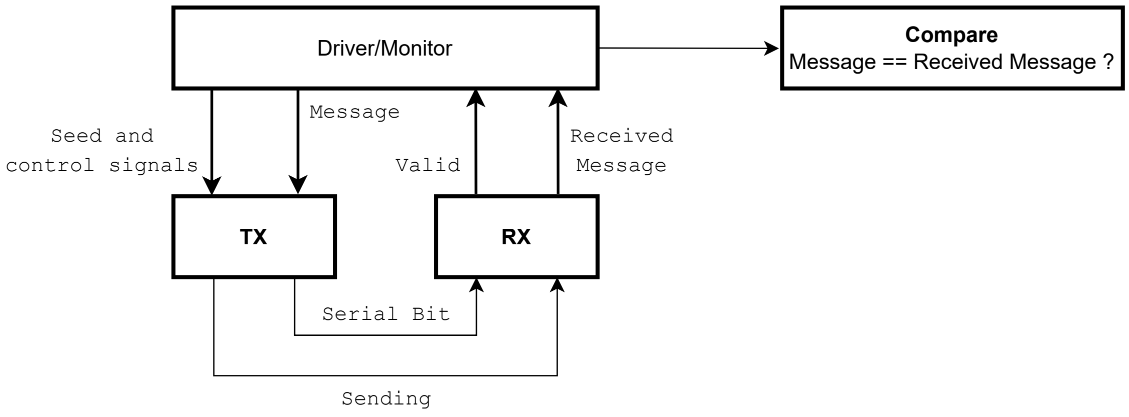
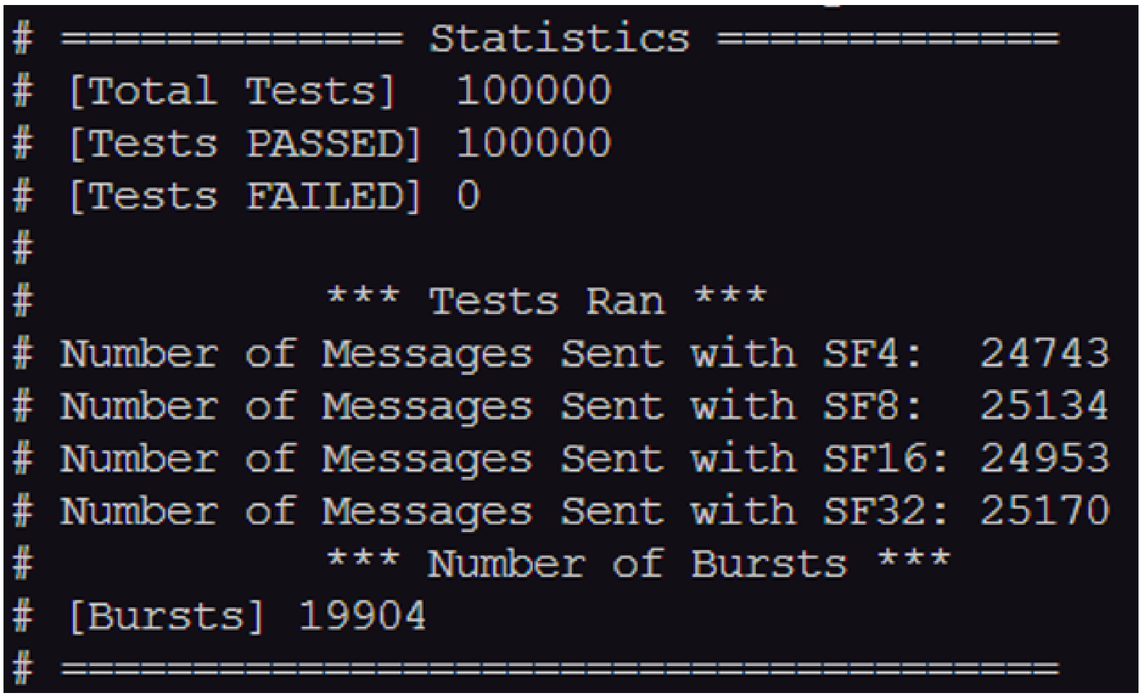


Figure 3.1: Test Environment Architecture

3.2 Test Scenario Generation

A SystemVerilog class was made to provide utility functions during testing. The class *"modem"* has sending and demodulation functions. The inputs to the constructor of the class is a virtual interface to the TX module. The class has two functions:



```
# ============= Statistics =============
# [Total Tests]  100000
# [Tests PASSED] 100000
# [Tests FAILED] 0
#
#           *** Tests Ran ***
# Number of Messages Sent with SF4:  24743
# Number of Messages Sent with SF8:  25134
# Number of Messages Sent with SF16: 24953
# Number of Messages Sent with SF32: 25170
#           *** Number of Bursts ***
# [Bursts] 19904
# ======================================
```

Figure 3.2: Simulation Test Cases Report

Sending Function

The sending function takes as inputs a 32-bit word to send, and a selection of a spreading factor. The function drives the inputs of the TX accordingly.

Demodulation Function

This function takes a queue of bits as input. The spreading factor is inferred from the size of the queue. Then the function returns the demodulated 32-bit word.

3.2.1 Scenarios

All test scenarios are automatically generated during runtime. The data of the message can be randomized, as well as the spreading factor. This means that the sender can control the spreading factor during runtime. This simulates the adaptive selection of SF based on the a priori CSI available at the sender to trade-off SNR for data rate. To simulate a priori CSI in the RX side, we send the SF to the RX via a control signal when sending a message.

Realistically, communication happens in bursts. This means that a number of messages is sent, followed by a number of clock cycles at which the sending entity is idle. To simulate this, a 20% chance of a burst ending was introduced after the sending of each message. This inserts random periods of silence between each group of messages where the length of a group of messages is itself random. This scheme covers all possible scenarios. See Figure 3.2 for the simulation of 100,000 test cases.

Chapter 4

Synthesis

This chapter discusses the synthesis results of the TX *and* RX *modules.*

The design was synthesized on the Cyclone V (5CSEBA6U1PI7) FPGA-based Chameleon96 board with a target frequency of 50 MHz using Quartus Prime Lite Edition. In this chapter we show the synthesis results.

4.1 Timing Analysis

MCMM analysis was run on the following corners:

1. Slow 1100mV 100C
2. Slow 1100mV -40C
3. Fast 1100mV 100C
4. Fast 1100mV -40C

Table 4.1: TX Timing Report.

Setup Slack	1.26 ns
Hold Slack	0.171 ns
F_{max}	53.73 MHz

Table 4.2: RX Timing Report.

Setup Slack	2.025 ns
Hold Slack	0.165 ns
F_{max}	55.63 MHz

4.2 Resource Usage

The design does not use any DSP blocks, PLLs, DLLs, memory blocks, or any other FPGA-specific hardware. The design does not depend on any external softcore IPs. This means that the design is guaranteed to be platform-agnostic. Shown below are the number of registers and ALMs used by the modem.

Table 4.3: TX Resource Usage Report.

Registers	242
ALMs	244

Table 4.4: RX Resource Usage Report.

Registers	69
ALMs	89

4.3 Power Dissipation

The TX consumes more power than the RX mainly due to the more complex actions it needs to take (multiplication to generate chaos, and modulation).

Table 4.5: Total Thermal Power of the Modem.

RX	421mW
TX	533mW

Chapter 5

Appendix

Shuffling Matrix

A total of 256 random numbers that range from 0 to 255 were generated using the *randperm* MATLAB function. These numbers are used to shuffle the expanded chaos to remove the statistical correlation between the output bits and themselves and the output bits and the input.

Table 5.1: Shuffling Matrix

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[0:15]	74	87	135	193	178	215	186	54	183	119	220	201	51	96	128	213
[16:31]	31	182	116	21	196	206	170	185	197	24	107	98	205	251	38	109
[32:47]	72	212	189	35	243	45	167	16	53	12	160	244	94	139	110	146
[48:63]	123	34	211	191	164	255	88	153	187	10	106	230	4	68	130	3
[64:79]	103	80	172	28	250	138	207	48	117	114	112	239	126	140	158	169
[80:95]	122	159	203	82	125	113	145	40	171	79	216	90	41	247	67	105
[96:111]	64	222	101	202	229	253	184	252	76	199	144	6	157	195	93	173
[112:127]	156	242	46	177	26	129	69	20	231	142	240	85	92	180	104	102
[128:143]	66	61	13	148	198	165	188	236	5	100	133	218	49	232	50	118
[144:159	]174	73	52	17	97	25	36	59	209	78	108	19	235	137	2	168
[160:175]	86	39	60	81	9	166	134	63	223	29	43	57	245	55	83	192
[176:191]	237	8	127	33	22	0	70	30	7	175	241	32	136	42	77	238
[192:207]	27	11	44	84	162	132	200	91	99	161	151	47	224	208	154	111
[208:223]	234	1	37	225	155	115	210	179	71	150	249	95	194	181	124	23
[224:239]	226	89	246	221	227	56	149	233	58	18	254	204	190	141	217	143
[240:255]	131	176	147	14	248	219	163	121	228	75	62	214	120	15	152	65