Sharif University of Technology

Department of Electrical Engineering

ASIC/FPGA Final Project Report

Software and Hardware Implementation of IEEE 802.11a Standard WLAN PHY

Mohammad Amin Mansouri — 94105174

Amirhossein Tavassol — 96104868

Spring 2020

Contents

1 Introduction **2**

2 Matlab Implementation **2**

3 HDL Implementation **5**

3.1 Obsolete Implementation 5

3.1.1 Performance Discussion 7

3.2 Optimized Implementation 8

3.2.1 Performance Discussion 9

1 Introduction

In this project we are going to implement IEEE 802.11a standard both using MATLAB and Verilog. 802.11a [1] is an IEEE standard for wireless communication that operates in the 5GHz band, using Orthogonal Frequency Division Multiplexing (OFDM). OFDM is an efficient multi-carrier modulation technique where the baseband signal is the composite of multiple data encoded sub-carriers. A top level diagram of the Transmitter and the Receiver is shown in Figure 1.

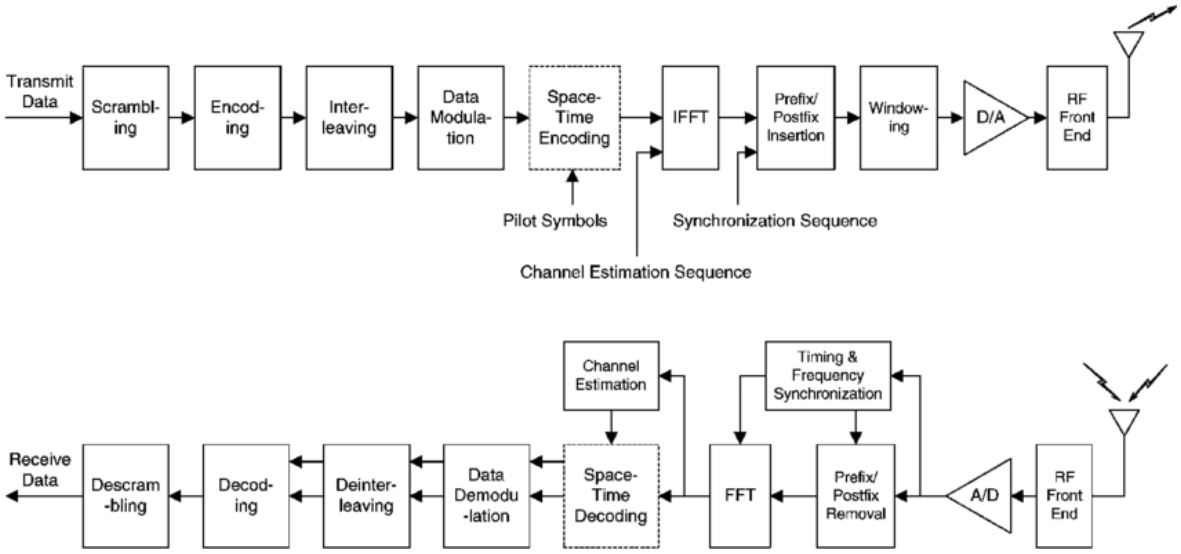

Figure 1: *Top level diagram of the design*

For phase 1, we will discuss the implementation of scrambler and descrambler blocks.

2 Matlab Implementation

By opening the the folder "Matlab" you'll find the following .m files:

- dataField.m: {inputs: RATE, LENGTH, PSDU} {outputs: DATA} — Gets the rate at which the packet should be transmitted along with the length of the PSDU in bytes and the PSDU itself as a long column of bits. Returns the Data field for this packet as a long column. SERVICE and TAIL bits are appended to the vector and the number of pad bits is calculated according to the relations provided by the standard (equations 11 to 13). More details of interfacing could be found by referring to the comments at the beginning of this file.
- RateSelect.m: {inputs: x} {outputs: y} — Gets an integer in the closed range [1,8]. Returns an integer representing the bit rate in the set {6, 9, 12, 18, 24, 36,

48, 54}. This function is a simple mapping of random numbers to bit rates and is used by the main.m.

- signal.m: {inputs: RATE, LENGTH} {outputs: SIGNAL} — Gets the rate at which the packet should be transmitted along with the length of the PSDU in bytes. Returns the SIGNAL field of the packet that should later be used by the Mapper block.

- scramble.m: {inputs: x, scramInit} {outputs: y} — This function gets vector input along with the initial state of the scrambler registers and returns the scrambled version of x. x could be of any size, scramInit could be either an integer in [1,127] or a column vector representing the same number in binary. Both situations will be handled.
  Scrambler consists of 7-bit shift register. Its circuit is shown in Figure 2. At each

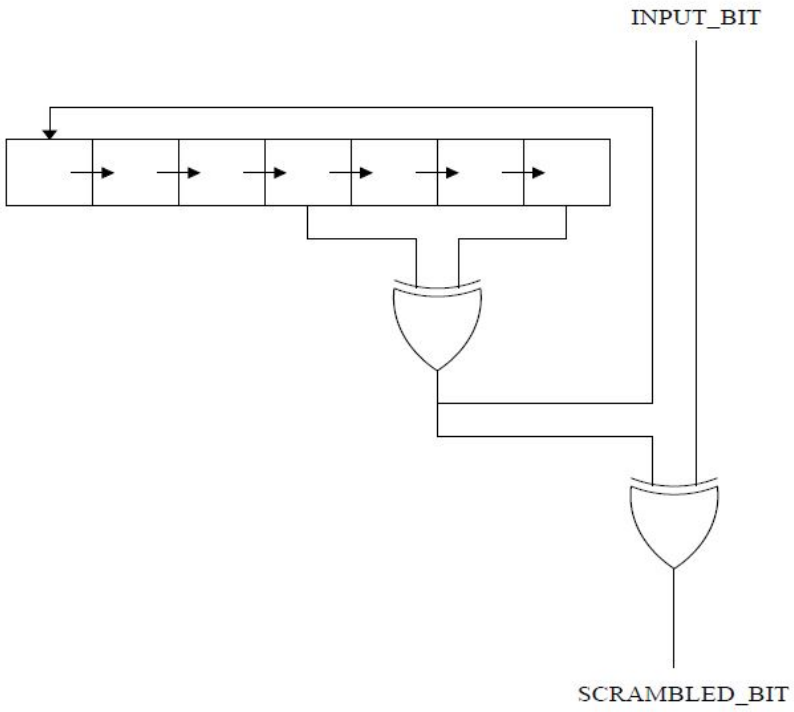

Figure 2: *Scrambler hardware implementation*

stage, the incoming bit is XORed with the XOR result of 4th and 7th registers and is returned as the output scrambled bit, then a shift and replacement is done according to figure 2. This is done in scramble.m for varying input sizes. The for loop plays the role of a clock.

- main.m:
  This file is divided into sections for easier access and is also divided in 4 phases that shall be completed later.
  ATTENTION: For your convenience, we've written the code in a way that by following these sections, you may generate random test vectors along with their golden outputs. You may also verify the correctness of the software implementation by going through these steps.

  - section 1.1: To change the number of packets you wish to be transmitted and tested using HDL, change "numOfSamples" to whatever number you want. (As long as you're confident that your system doesn't run out of memory!) Then some random numbers between [1,8] are generated and by the help of RateSelect.m will give us different bitRates, one for each packet. Also a random length will be generated for each packet's PSDU. Suppose you have selected the "numOfSamples" to be 100, When this section finishes, you'll have 100 different PSDU lengths, 100 different rates, and indeed 100 PSDUs!
  - section 1.2: By utilizing "signal" and "dataField" functions (explained earlier) we arrive at the corresponding SIGNAL and DATA_FIELDS of our samples.
  - section 1.3: The initial state of scrambler is randomly generated, and the scrambled versions of samples' DATA_Fields are generated. (By utilizing our scramble function.)
  - section 1.4: Since the process of descrambling is exactly the same as scrambling, we pass our scrambled data from previous section along with the same seed, once more to the scrambler function to achieve samples' descrambled DATA_FIELDS.
  - section 1.5: It's quite clear what happens there. It's just a double check that everything is fine and descrambler outputs are the very same inputs of the scrambler. If you see that the number displayed in the command window is the same as numOfSamples, then the scrambler and descrambler have done their job correctly! Software implementation finished.
  - section 1.6: This section has the responsibility of writing various variables and data to text files that will be read by the HDL testbench. It will write the PSDUs, BitRates, scrambled Data fields to their corresponding files (that we will not use in this phase), and also writes DATA_FIELDS of our samples and their descrambled version (which are basically the same, but in case you don't trust us, we've written them.) to test_vector.txt and golden_outputs_m.txt respectively, just as described in project description. Note that every single bit is written in a new line. These bits will be read in a serial manner by the HDL testbench. When this section finished running,

please copy test_vector.txt and golden_outputs_m.txt to the same directory that you've put scrambler_tb.v (explained later) for simulation.

– sections 1.7 and 1.8: By looking at annex G.2 of IEEE's 802.11a standard (page 55) you'll find an example message (Ode to joy ...). The standard has provided the resulting bit streams of each block supposing that the message is this example. We've generated this message in section 1.7 and have stored it in sampleMessage.txt. In section 1.8, its data field is generated first, then this data field is scrambled and descrambled and the results are stored in sm_golden_scrambler_output_m.txt and sm_golden_outputs_m.txt respectively. You may open these text files to verify the correctness of software implementation. If you want to use them with test bench, you'll need to copy all of them to testbench's directory and edit testbench to read from these for input and verification. (Instructions are available in comments.)

3 HDL Implementation

By opening the the folder "Verilog Modules" you'll observe two subfolders: 1.Obsolete, 2.Optimized. First we'll discuss the obsolete version and then we'll elaborate on optimization adjustments and problems that should be addressed in that implementation, and finally present the final design for the scrambler module in section 3.2.1.

3.1 Obsolete Implementation

If you go the subfolder "obsolete" you'll find the following .v files:

- signalField.v: {inputs: 8-bit RATE, 12-bit LENGTH} {outputs: 24-bit SIGNAL} — Gets the rate at which the packet should be transmitted along with the length of the PSDU in bytes. Returns the SIGNAL field of the packet that should later be used by the Mapper block. This module has no use in the current phase.

- scrambler.v: {inputs: 1-bit x, 7-bit initialState, 1-bit MODE, 1-bit clk, 1-bit reset} {outputs: 1-bit x_scrambled, 1-bit valid } —
  This function gets one serial bit along with the initial state of the scrambler registers and returns the scrambled version of x. Obviously the output of this function shall be used.
  x is the serial input of scrambler, initialState is the initial state of the scrambler/descrambler explained earlier, MODE determines whether the instance of this module is acting as a scrambler or descrambler. For verification, when the modules start, the output of scrambler is not valid for two clocks and the output of descrambler is not valid for 3 clocks because at first posedge after reset, only input is valid, the next posedge, scrambler output is meaningful, the next

posedge, descrambler is also in the game. That's why we need MODE. For one instance of scrambler we need to wait 2 clks, for the other one, 3 clks, to distinguish, we use MODE.
x_scrambled is the serial output of this module. Valid bit is used in testbench. As was mentioned above, when we're writing results to text files, we should note not to write unknowns (X) in the file. This valid bit helps us to compare the results and golden vectors in a proper manner.
As is obvious by going through the code, when asynchronous reset is deasserted, valid bit is set to 0, initial state is loaded from the input and counter is set to 0 too. Counter keeps the measure of how many posedges we've been through since the reset has become 1. If the instance is an scrambler (MODE = 1), after two posedges, the circuit is permitted to work and it's important to note that prior to this event, no shift should occur in initial state registers. That's why the operations are encapsulated in the if-clause. The same goes for descrambler instance (MODE = 0) with the slight difference that the starting condition is that counter be greater or equal than 2.

- scrambler_tb.v: {inputs: NONE} {outputs: NONE} — Please don't bother yourself to check commented registers and wires, they were meant for test purposes only. Wires and registers used in the testbench are as follows:

  - reg inputSerialBit: Serial bits read from test_vectors are stored in this reg.
  - reg goldenOutputBit: Serial bits read from golden_outputs_m are stored in this reg.
  - reg goldenScramblerOutputBit: Serial bits read from golden_scrambler_outputs_m.txt are stored in this reg. This will be used for the comparison of Matlab scrambler outputs with HDL scrambler outputs.
  - reg [6:0] initState: This is the initial state of scrambler and is initialized inside an initial block.
    IMPORTANT NOTE: Set it to whatever scarmInit that Matlab has generated in the workspace while you were generating test vectors. If you don't do so, obviously the default value of HDL module which is "0000_000" will be used for scrambling. Though this does not affect the bit matching of descrambler (because it will both scrambler and descramble with that same default seed), it does affect the bit matching of scrambler because Matlab has used some other seed to scramble the data.
  - reg clk
  - reg reset
  - wire scrambler_output: This wire is connected to the scrambler instance's output and then is connected to the descrambler's serial input.

- wire descrambler_output: This wire is connected to the descrambler instance's output and then is compared to goldenOutputBit read from the golden output file generated by Matlab.
- wire valid_scrambler: Valid bits were explained in the description of scramble.v. This bit serves that purpose.
- wire valid_descrambler: Valid bits were explained in the description of scramble.v. This bit serves that purpose.
- reg [31:0] failuresCounter: Counts the number of events that goldenOutputBit is NOT the same as descrambled_output.
- reg [31:0] successCounter: Counts the number of events that goldenOutputBit IS the same as descrambled_output.
- reg [31:0] scramblerfailuresCounter: Counts the number of events that goldenScramblerOutputBit is NOT the same as scrambled_output.
- reg [31:0] scramblersuccessCounter: Counts the number of events that goldenScramblerOutputBit IS the same as scrambled_output.
- reg finished: When $fscanf arrives at the end of test_vectors.txt (eof is TRUE), this flag becomes one. (It has been deasserted at reset.) Indicates that the process has finished. The last initial blocks waits during the whole procedure for this flag to be asserted. #100 After this event, the number of failures and successes are printed to the transcript and then the testbench is stopped.
- integer f_scrambler_out: Serial outputs of scrambler are written to a file for your reference, which this integer is the pointer to.
- integer f_scrambler_golden_output: Serial golden output bits of scrambler that are used for comparison with the scrambler output, are read from golden_scrambler_outputs_m.txt and this integer is the pointer to that file.
- integer f_descrambler_out: Serial outputs of descrambler are written to a file for your reference, which this integer is the pointer to.
- integer f_test_vectors: Serial inputs of scrambler are read from test_vectors.txt and this integer is the pointer to that file.
- integer f_golden_outputs_m: Serial golden output bits that are used for comparison with the descrambler output are read from golden_outputs_m.txt and this integer is the pointer to that file.
- integer i,j,k: When we use $fscanf, we need to store the result of operation in some integer and these serve this purpose.

3.1.1 Performance Discussion

After synthesizing this module on Xilinx on Virtex 6 FPGA board, there were severall key points that need to be considered. Let's first take a look at Device utilization

summary:

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	55	301440	0%
Number of Slice LUTs	91	150720	0%
Number of fully used LUT-FF pairs	41	105	39%
Number of bonded IOBs	13	600	2%
Number of BUFG/BUFGCTRLs	1	32	3%

Table 1: *obsolete version utilization summary on Virtex 6 FPGA*

From table 1 it seems that quite a huge logic is used for the implementation. Also the synthesizer gives us warning cocerning our counter. Also setting initialState to input at reset is creating some undesired behavior and logic. Furthermore, we need to obey the fact that scrambler is immediately followed by the descrambler, o.w. the design will not work because our code is currently based on the fact that after 2 posedges, scrambler will have valid outputs and after 3 posedges, descrambler will have valid outputs that can be used for comparison. But in later phases where lots of other blocks will get in the path of scrambler/descrambler, these assumptions are certainly violated. To accomodate that we need to design a module that relies on no further assumptions about the intermediate path of these two modules. That's presented in 3.2.1. Another issue is that here we need to externally determine whether our instance of this module is going to act as an scrambler or descrambler. But noting that the operation is inherently the same, this implementation and requirement to distinguish between instantiations seems suboptimal. This problem is also addressed in our final implementation. Now that we know what needs to be handled, let's go to the final implementation.

3.2 Optimized Implementation

If you go the subfolder "optimized" you'll find the following .v files:

- signalField.v: {inputs: 8-bit RATE, 12-bit LENGTH} {outputs: 24-bit SIGNAL} — Same as before.

- scrambler.v: {inputs: 1-bit x, 7-bit initialState, 1-bit run, 1-bit clk, 1-bit reset} {outputs: 1-bit x_scrambled, 1-bit valid, 1-bit rdy } —
  In this module, we've utilized an FSM for our purpose. There are two states: seed_init is when we've passed a posedge after reset. At this time initialState should be loaded to scrambler initial registers. Also next state will be ready from now on, i.e. the module can accept input serial bits. For the next posedges, if the input run is set, the module may start scrambling but whenever this input

is set to zero, then no shift will happen in scrambler's registers and no output in the testbench is read from this module nor is written to any text file. Also to use one less XOR gate, we've taken advantage of the internal_xor wire to reuse it where needed and not synthesize any more XOR gates. valid_run is initially zero and whenever that both 1. seed is loaded and 2.the input is ready (indicated by run == 1) lets module to operate.

- scrambler_tb.v: {inputs: NONE} {outputs: NONE} — Everything is the same except for the connection of scrambler and descrambler. As can be seen in the code, a wire rdy_scrambler is connected from scrambler's rdy output to its "run" input. This way, after the first posedge passes, the module will automatically start working, the next clock, hence there would be no need for external starting of that module. Also the valid output of scrambler ("scrambler_valid") is connected to descrambler's "run" input. This connection enables the descrambler to start automatically whenever scrambler has its first valid output.

3.2.1 Performance Discussion

Let's take a look at the newly generated utilization summary:

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	10	301440	0%
Number of Slice LUTs	10	150720	0%
Number of fully used LUT-FF pairs	9	105	81%
Number of bonded IOBs	14	600	2%
Number of BUFG/BUFGCTRLs	1	32	3%

Table 2: *optimized version utilization summary on Virtex 6 FPGA*

Minimum period	1.098ns
Minimum input arrival time before clock	1.387
Maximum output required time after clock	0.783ns

By comparing table 2 to table 1 it evident that we've achieved quite an improvement! In addition to having much fewer logic units consumed, we've also adapted the module in a way that is far more useful in later phases when other blocks are integrated. There's no analysis required to see when each one should wake up and operate, hence there's no need to manually design control signals for them, thanks to "run" and "rdy" inputs and the FSM, all of that is achieved automatically. Also note that we

do not need to distinguish between scrambler and descrambler when they're initialized.

That's all for phase 1 :)