



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	Contribution table	2
2	Instructions for the grader	3
3	Introduction	4
4	Phase 1, Scrambler and Descrambler	4
4.1	Matlab Implementation	4
4.2	HDL Implementation	7
4.2.1	Obsolete Implementation	7
4.2.2	Performance Discussion	10
4.2.3	Optimized Implementation	10
4.2.4	Performance Discussion	11
5	Phase 2, Encoder and Decoder	12
5.1	Matlab Implementation	12
5.2	HDL Implementation	16
5.2.1	Performance Discussion	18
6	Phase 3, Interleaver and De-Interleaver	18
6.1	Matlab Implementation	19
6.2	HDL Implementation	23
6.2.1	Performance Discussion	25
7	Phase 4, Mapper, Demapper and Pilot Inserter, and Remover	26
7.1	Matlab Implementation	26
7.2	HDL Implementation	27
7.2.1	Performance Discussion of integrated TX and RX	28
8	Integration, Matching and Verification	29
8.1	Matlab Implementation	29
8.1.1	Code Section 5.1	29
8.1.2	Code Section 5.2	30
8.1.3	Code Section 5.3	31
8.2	HDL Implementation	31

1 Contribution table

Part	Amin	Amirhossein
Software Implementation of Scrambler/Descrambler	✓	-
HDL Implementation of Scrambler/Descrambler	-	✓
HDL Verification of Scrambler/Descrambler	✓	-
Scrambler/Descrambler Report	✓	-
Software Implementation of Encoder/Viterbi Decoder	✓	-
HDL Implementation of Encoder/Viterbi Decoder	-	✓
HDL Verification of Encoder/Viterbi Decoder	✓	-
Report of SW Implementation of Encoder/Viterbi Decoder	✓	-
Report of HDL Implementation of Encoder/Viterbi Decoder	-	✓
SW Implementation of Efficient Interleaver/De-Interleaver	✓	-
HDL Implementation of Interleaver/De-Interleaver	-	✓
Report of SW Implementation of Interleaver/De-Interleaver	✓	-
Report of HDL Implementation of Interleaver/De-Interleaver	-	✓
Software Implementation of Mapper/Demapper	✓	-
HDL Implementation of Mapper/Demapper	-	✓
Report of SW Implementation of Mapper/Demapper	✓	-
Report of HDL Implementation of Mapper/Demapper	-	✓
Software Implementation of Pilot Inserter/Remover	✓	-
HDL Implementation of Pilot Inserter/Remover	-	✓
Report of SW Implementation of Pilot Inserter/Remover	✓	-
Report of HDL Implementation of Pilot Inserter/Remover	-	✓
Software Integration	✓	-
Software Verification of Standard's example	✓	-
Software Verification of generalized random data	✓	-
HDL Integration	-	✓
HDL Integration Verification	✓	✓
Software Integration Report	✓	-
HDL Integration Report	✓	-

2 Instructions for the grader

We've organized the uploaded zip file is in 5 folders (plus this report, of course!), 4 of which are for phases 1 to 4. Each includes two subfolders containing MATLAB files and Verilog files (modules+testbenches) of that phase. You may also find and test all parts of the transmitter and receiver by going to MATLAB and Verilog folders of the folder "Integrated."

For grading each phase, please go to its dedicated section according to the table of contents. The report for each phase contains more than enough explanation for the algorithms that we have used, RTL and MATLAB code walk-through and synthesis report. For your convenience, parts of the project that we believe may contribute to bonus marks, are indicated with [Bonus](#) accompanied by an explanation of how it goes beyond the expectation of the project.

Also MATLAB codes are quite well-organized, written with many details, and have plenty of comments to make sure of any reader's convenience. I believe it may be eligible for a [Bonus](#)!

In addition, the report is complete and well-organized too. We've put a great deal of time into this report to create a manuscript that can be used by anyone later who may wish to integrate our project as part of his/her work. I also believe that this level of organization may be eligible for a [Bonus](#), too :)

3 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.

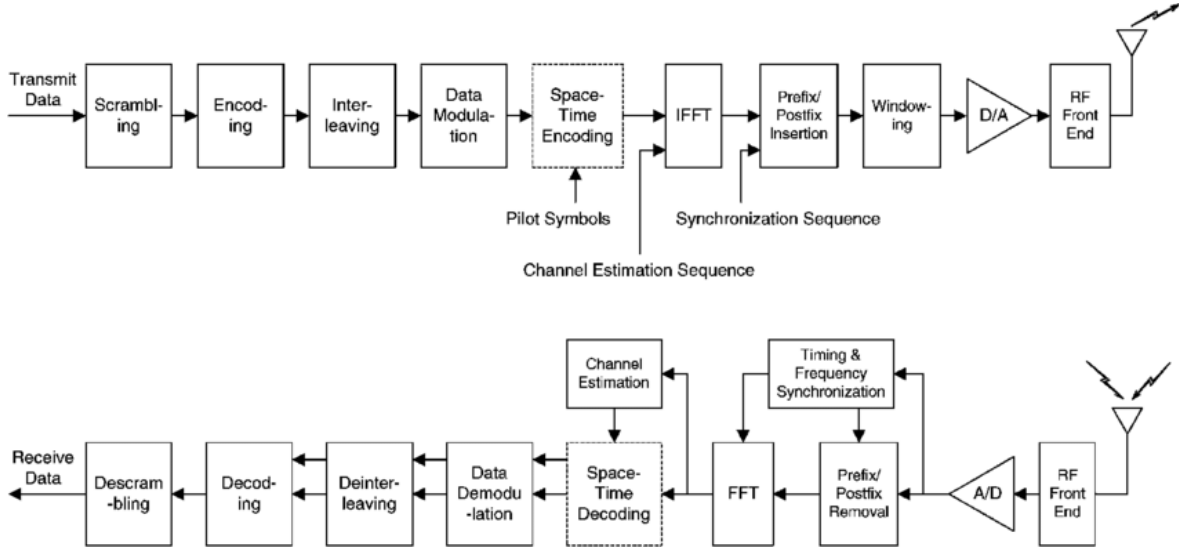


Figure 1: Top level diagram of the design

4 Phase 1, Scrambler and Descrambler

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

4.1 Matlab Implementation

By opening the subfolder "Matlab" in the folder "Phase.1" 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. It 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]. It 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 a 7-bit shift register. Its circuit is shown in Figure 2. At

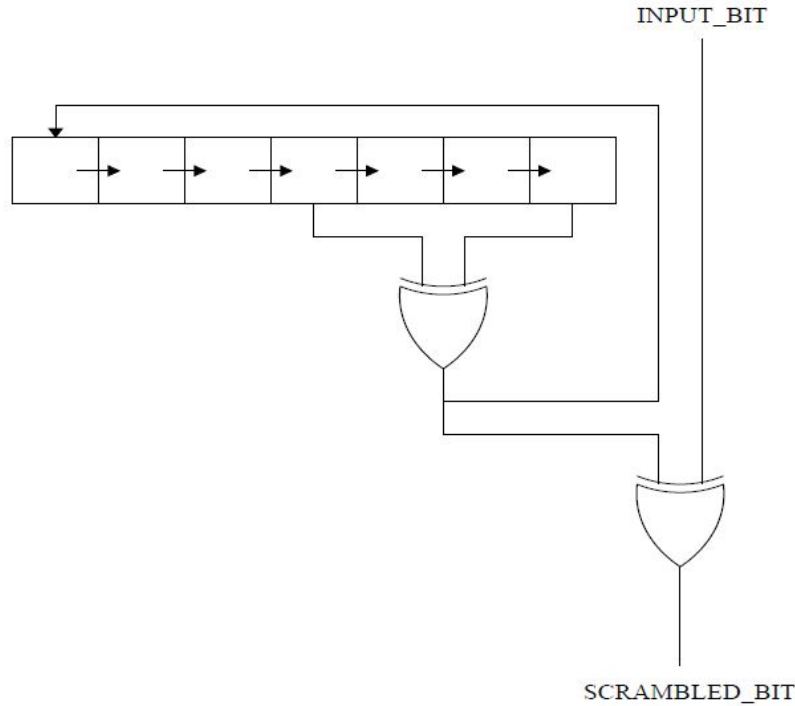


Figure 2: Scrambler hardware implementation

each 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 into 4 phases. For this phase, please refer to the code between Phase 1 and 2.

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 large packet of PSDU size (a random length) between [1,4095] (According to the standard). Suppose you have selected the "numOfSamples" to be 100; when this section finishes, you'll have 100 different PSDU lengths, 100 different rates, and of course, 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.) Note that it's not possible to scramble each packet using scrambler individually because the seed is constantly changing. Suppose we scramble the first packet with our initial seed; if we continue and scramble the next packet with the same initial seed, we'll arrive at wrong scrambled data, and verification of its golden outputs is going to fail in HDL testbench. Hence we need to concatenate all Data fields and scramble them all at once so that the changes of the seed is considered for all of them. This way, the verification in HDL testbench is successful. (In fact, this was one of the major errors that we encountered when trying to get bit matching.) Also, we keep in mind that SIGNAL FIELD doesn't require scrambling, so there's no concern for that.
- section 1.4: Since the process of descrambling is exactly the same as scrambling, we pass our scrambled data from the 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 zero, 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, 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 the 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, all files are available in subfolder "Phase 1 txt files", 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 bitstreams 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 the test bench, you'll need to copy all of them (from "Standard's example data" subfolder) to testbench's directory and edit testbench to read from these for input and verification. (Instructions are available in test bench comments.)

4.2 HDL Implementation

By opening the subfolder "Verilog" in the folder "Phase_1" 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 4.2.4.

4.2.1 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 for 2 clks, for the other one, 3 clks, to distinguish, we use `MODE`.

`x_scrambled` is the serial output of this module. The valid bit is used in the 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 the asynchronous reset is deasserted, valid bit is set to 0, the initial state is loaded from the input, and the counter is set to 0 too. The counter keeps the measure of how many posedges we've been through since the reset has become 1. If the instance is a 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 the 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 scramble 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.) It indicates that the process has finished. The last initial block 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 the operation in some integer, and these serve this purpose.

4.2.2 Performance Discussion

After synthesizing this module on Xilinx on Virtex 6 FPGA board, there were several 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 a warning concerning 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 accommodate 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 4.2.4. Another issue is that here we need to externally determine whether our instance of this module is going to act as a scrambler or descrambler. But noting that the operation is inherently the same, this implementation and requirement to distinguish between instantiations seem 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.

4.2.3 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's 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 automatically start whenever scrambler has its first valid output.

4.2.4 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

By comparing table 2 to table 1 it is evident that we've achieved quite an improvement! In addition to having much fewer logic units consumed, we've also adapted the

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

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. If you believe we've achieved a minor optimization and this effort is eligible, please consider a [Bonus](#) for that.

That's all for phase 1 :)

5 Phase 2, Encoder and Decoder

For phase 2, we will discuss the implementation of convolutional encoder and Viterbi decoder blocks.

5.1 Matlab Implementation

By opening the the subfolder "Matlab" in the folder "Phase_2" you'll find the following .m files:

- dataField.m: {inputs: RATE, LENGTH, PSDU} {outputs: DATA} — Explained in Phase 1.
- RateSelect.m: {inputs: x} {outputs: y} — Explained in Phase 1.
- signal.m: {inputs: RATE, LENGTH} {outputs: SIGNAL} — Explained in Phase 1.
- scramble.m: {inputs: x, scramInit} {outputs: y} — Explained in Phase 1
- convolutionalEncode.m: {inputs: x, initState, puncturePattern} {outputs: convolvedData} — This function gets vector input along with the initial state of the encoder registers along with the puncture pattern that shall be used for rate increment, and returns the convolutionally encoded version of x. x could be of any size, initState should be a binary column vector.

Encoder consists of 6 shift registers plus 2 1-bit adders. Its circuit is shown in Figure 3. At each stage, the incoming bit is XORed with the XOR result of 2nd,

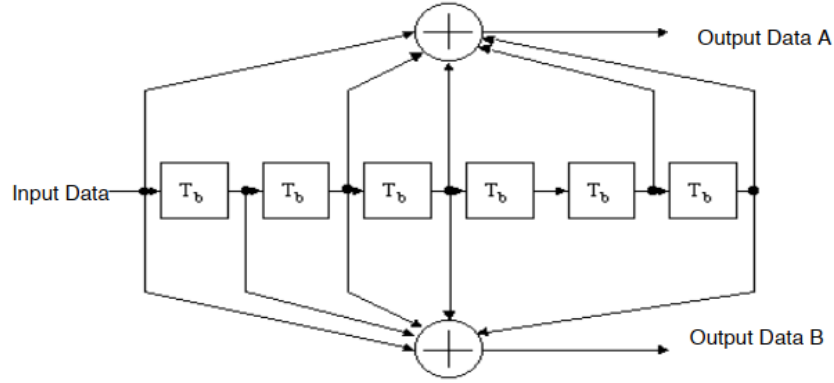


Figure 3: Convolutional encoder hardware implementation

3rd, 5th, and 6th registers to return the output bit A (which should be read first by the block coming after encoder) and is XORed with the XOR result of 1st, 2nd, 3rd, and 6th registers to return the output bit B (which should be read after bit A by the block coming after encoder). Then a shift and replacement is done according to figure 3. This is done in `convolutionalEncode.m` for varying input sizes. The for loop plays the role of a clock. Also note that any puncture pattern could be received as an input by this block and it's repeated as much as required so as to accommodate the vector y which is the unpunctured output of decoder. After that, those entries corresponding to ones in the long puncture vector are preserved in the final output and other entries are discarded. That's how we achieve higher rates.

- `viterbiDecode.m`: {inputs: `dataChunk`, `trellis`, `predecessors`, `initCurrState`} {outputs: `decodedData`, `finalCurrState`} — For the purpose of decoding, it was highly recommended to use Viterbi algorithm. Hence, we'll briefly discuss it and then go through our code. This algorithm is based on the critical assumption that we're facing minimum noise in the communication channel. Therefore tries to find the most likely input pattern that has resulted in what we've received. We need three definitions to go on:
 - Trellis structure: Trellis provides a good framework for understanding decoding. Suppose we have the entire trellis in front of us for a code, and now receive a sequence of bits. If there are no errors (i.e., the noise is low), then there will be some path through the states of the trellis that would exactly match up with the received sequence. Each column of trellis corresponds to a time step. Each entry of this matrix-like structure, corresponds to a state of those 6 shift registers in convolutional encoder and a specific time step. For each of them, if the input is 0, we are connected to some next state and if the input is 1, we are connected to some

next other state. For each input there are also two outputs associated with it. (bit A,B in convolutional encoder) in the last column of figure 4 2 output bits are black colored.

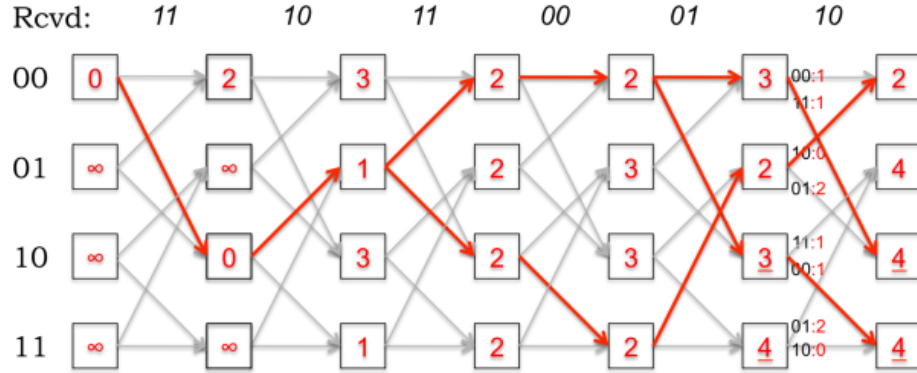


Figure 4: Trellis Structure used for decoder's trace back

– Branch Metric:

The branch metric is a measure of the “distance” between what was transmitted and what was received, and is defined for each arc in the trellis. It’s basically the Hamming distance between the expected parity bits and the received ones. An example is shown in figure 5, where the received bits are 00. For each state transition, the number on the arc shows the branch metric for that transition. Two of the branch metrics are 0, corresponding to the only states and transitions where the corresponding Hamming distance is 0. The other non-zero branch metrics correspond to cases when there are bit errors.

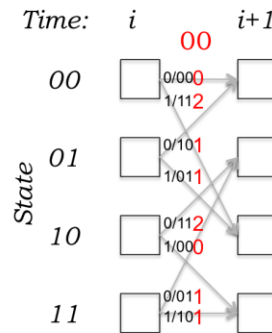


Figure 5: Branch Metric Example

– Path Metric:

The path metric is a value associated with a state in the trellis (i.e., a value associated with each node). It corresponds to the Hamming distance

over the most likely path from the initial state to the current state in the trellis. By “most likely”, we mean the path with smallest Hamming distance between the initial state and the current state, measured over all possible paths between the two states. The path with the smallest Hamming distance minimizes the total number of bit errors, and is most likely when the BER is low.

Now that we have all the ingredients, we may proceed. Encoder starts with initial state of all zeros for each packet. So the decoder assumes the same, starts from all zeros state. **IMPORTANT:** Since our packet lengths can greatly vary in size (i.e. from 1 to 4096 bytes) it’s evident that keeping track of this amount of path and path metrics is going to require a huge amount of memory and is absolutely inefficient. Therefore, according to several research and recommendations, we split each encoded package into chunks of bits and decode them separately. If the size of data chunk is about encoder length (6) times 5, then it’s safe to assume that the very first bits of a chunk won’t have significant effect on the decoding of the bits of the following chunk, given that noise is low. But it’s important to note we need to get the most likely state when the decoding process finishes for a chunk, and then give it as the initial state of the next chunk. Other chunks shall not start at state of all zeros. That’s why we’ve included an initial state as an input of decoder. Because we’re going to decode the data of a packet in chunks. So we decode the data in 24-bit chunks because it’s the **greatest common divisor** of N_CBPS that is consistent with all data rates.

Now that we are aware of all the details, the rest of the code is easy to understand. Initial value for PathMetric is infinity for all states but the initial one for which the metric is assigned zero. We need to keep the traversed path along the minimum path metric so we defined a variable traversedPath for that reason. In the for loop, each loop corresponds to a clock cycle. At each cycle we get two bits from the encoding stream, calculate all branch metrics for inputs 0 and 1 for all states, then for each state the minimum of its path metric along its predecessors is achieved. Predecessor is a cell and keeps the predecessor states of each state along with the input bit that brings us to that state from its predecessors. When the for loop terminates, all path metrics have been calculated all the way from the first time step to the end. Finding the most likely last state is done by taking the minimum value of the last time step. It shall be used for the next chunk so it is returned as the finalCurrState. After that by doing a simple traceback from the last state and using the predecessor data structure we may decode the data and return it decodedData.

IMPORTANT: For rates higher than $1/2$, we need to insert **Dummy zero bits** at the place of punctured bits. Hence, for each data rate, we need to use a puncture pattern to insert those bits before passing the data to the Viterbi decoder. More importantly, these dummy bits should not be included in branch metric calculation so we need to use the same puncture pattern as an erase marker to

ignore these 0 bits when calculating branch metrics.

Since our encoder and viterbi decoder work for **All Rates**, we believe it is eligible for a [Bonus](#) because it goes beyond the expectation of the project. (Just a single rate was asked.)

- `puncPattern.m`: {inputs: Rate} {outputs: pattern, coeff} — For the purpose of puncturing, we require a data rate, then according to the standard's patterns on page 18, we return a puncture pattern and the coeff is used to transform the lengths of future variables, i.e. a $3/4$ rated data has $2/3$ length of its input, so we need to multiply some lengths by $3/2$. The use of this coefficient becomes clear later.

- `main.m`:

We've used this file in phase 1. We'll continue from Phase 2 section which could be easily observed in the file.

- section 2.1: First we construct the trellis struct used in the following steps. This trellis' generator polynomials are defined according to the standard's convolutional encoder. Since the states are numbered from 0 to 63 and it created difficulty when working with matlab, we modify the struct by changing the numbering. Finally we construct the predecessor data structure which we will utilize in the procedure of tracing back and calculating the path metrics.
- section 2.2: In order to test the encoder and the decoder, we generate random binary inputs (you may vary its size), convolutionally encode it with initial state of zero, puncture pattern is automatically obtained according to the `puncPattern` function which receives randomly generated rates for our samples. `initCurrState` is the first state. It should be obvious by now, since when we start, we want our initial state to be all zeros and that corresponds to state 1. We then decode it and display the Bit Error Rate (BER) for verification.
- section 2.3: In this section we generate test vector `encDec_test_vectors.txt` used by HDL test bench `encoderDecoder_tb.v` as input. Also golden outputs are written to txt files for comparison in HDL. All of these files will be written in subfolder "Encoder Vectors".
IMPORTANT: Our Viterbi decoder only works for $1/2$ data rate so please set the rate in section 2.2 to one of these: 6, 12, 24 Mb/s. Then copy the files to where you've put the `encoderDecoder_tb.v` test bench and you'll observe that both encoder and decoder are verified.

5.2 HDL Implementation

By opening the subfolder "Verilog" in folder "Phase_2" you'll observe these Verilog files: `encoder.v`, `counter0_23.v`, `decoder.v`, `error_adder.v`, `error_adder1.v`, `error_adder2.v`, `er-`

ror_adder3.v, min_error.v, min8.v, min8_index.v. There's also a test bench encoderDecoder_tb.v.

The contents of the test bench are similar to that of scrambler and hence are pretty obvious and we won't go further into its details. Due to the severe shortage of time, we can't go into the details of auxiliary Verilog files. Here's a description of the major module, decoder.v (encoder.v is clear):

This module takes 48 bit length data block. First computes errors in 24 clocks (refer to figure 6 for the structure) and writes the wasted bit of state in each step in 'path_mem' to be able to return the path. When error computing finishes, it finds the minimum error and returns that path and keeps input bits to give it to the output. Writing in 'path_mem' will be done once from left to right and once from right to left for optimal memory usage. Because of 'min8's large area we only enable its inputs when we need to compute minimum error for power saving (refer to figure 7 for the structure). We start computing block errors from the last state of the previous block. This module can continuously take input and decode it. When input stops (run = 0) before the block is completed, it waits until the input comes. Also note that the largest delays is due to the path from err_22 to mpn_0. We have traded it off with being able to continuously take input, o.w., our frequency could be much higher by giving up this feature.

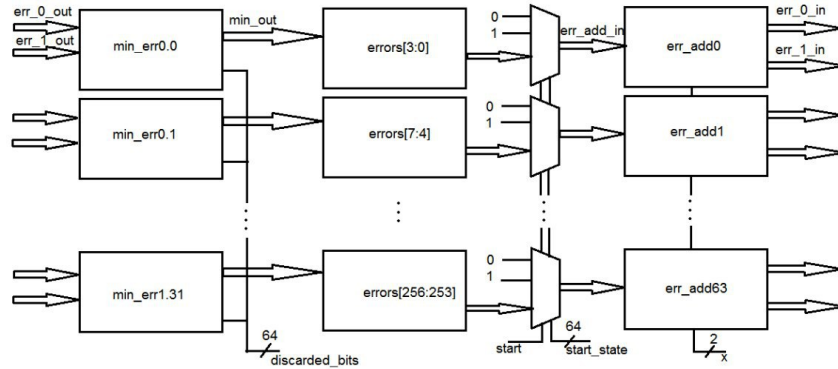


Figure 6: Error Calculation

Bonus: Though the detailed explanation of this module wasn't possible, we've put a great deal of effort into this, and it can **continuously** take data and decode after a latency of 33. This took a lot of effort. Also, keep in mind that we decode in blocks to achieve zero BER, otherwise a very huge memory is required to keep track of the traversed path. We believe this decoder may be eligible for a **Bonus**.

IMPORTANT: While the rest of all design is compatible with all data rates, Our encoder and decoder are only compatible with 1/2 coding rate. Please keep this in mind while generating test vectors for the verification of this phase and integration phase. (Though we've taken care of this issue while generating test and have mentioned it in MATLAB when test vectors are being generated.)

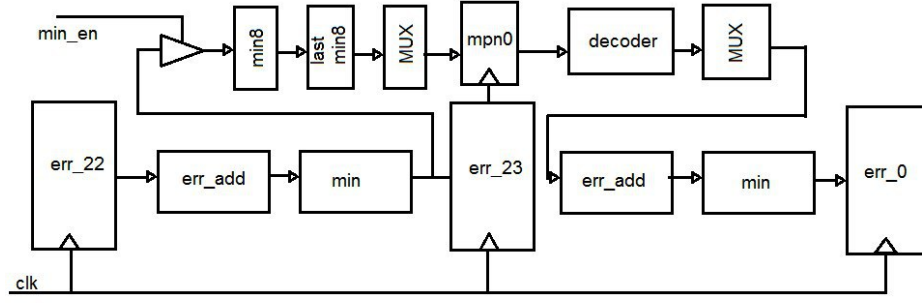


Figure 7: Error Calculation Power Saving by Pipelining

5.2.1 Performance Discussion

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	6	301440	0%
Number of Slice LUTs	3	150720	0%
Number of fully used LUT-FF pairs	0	9	0%
Number of bonded IOBs	7	600	1%
Number of BUFG/BUFGCTRLs	1	32	3%

Table 3: Encoder utilization summary on Virtex 6 FPGA

Minimum period	0.799ns (1251.56 MHz)
Minimum input arrival time before clock	1.354ns
Maximum output required time after clock	1.626ns
Maximum combinational path delay	0.904ns

Table 4: Encoder timing analysis on Virtex 6 FPGA

It is easily observable how the decoder slows down the circuit and it's a very hardware intensive circuit.

6 Phase 3, Interleaver and De-Interleaver

For phase 3, we will discuss the implementation of interleaver and de-interleaver blocks.

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	338	301440	0%
Number of Slice LUTs	6266	150720	4%
Number of fully used LUT-FF pairs	133	6471	2%
Number of bonded IOBs	7	600	1%
Number of BUFG/BUFGCTRLs	1	32	3%

Table 5: Decoder utilization summary on Virtex 6 FPGA

Minimum period	11.594ns (86.248 MHz)
Minimum input arrival time before clock	11.958ns
Maximum output required time after clock	0.930ns

Table 6: Decoder timing analysis on Virtex 6 FPGA

6.1 Matlab Implementation

By opening the the subfolder "Matlab" in the folder "Phase_3" you'll find the following .m files:

- rateDependents.m: {inputs: dataRate} {outputs: Modulation, N_BPSC, N_CBPS, N_DBPS} — This function gets dataRate ans by a switch cases determines all rate dependent parameters that will be used for interleaving, de-interleaving and mapping/demapping. It's just a function to increase reusability.
- RateSelect.m: {inputs: x} {outputs: y} — Explained in Phase 1.
- signal.m: {inputs: RATE, LENGTH} {outputs: SIGNAL} — Explained in Phase 1.
- main.m:
We've used this file in phases 1,2. We'll continue from Phase 3 section which could be easily found in the file.
 - section 3.1: It's pretty simple. The whole main code for this phase is in section 3.1. You may choose a rate, then modulation and N_CBPS is retrieved from the function "rateDependents". Since interleaver gets **blocks** of data and does the interleaving operation, we also divide it into N_CBPS blocks and perform the interleaving. For each block we also do the de-interleaving and give the final bit error rate (BER). If you run this section and obtain zero, it means the job is done. The primary optimization happens in the following function.

- `interleaveDeinterleave.m`: {inputs: `x`, `mode`, `N_CBPS`, `ncol`, `nrow`, `mapmode`} {outputs: `y`} — This function gets data block (of size `N_CBPS`), mode of function (0 for interleaving and 1 for de-interleaving), and `N_CBPS`. `ncol` is fixed according to the standard and is equal to 16. `nrow` is `N_CBPS/16`. if the interleaver/deinterleaver is not efficiently implemented, it occupies silicon area as many as some significant blocks (i.e., Fast Fourier Transform (FFT), Viterbi Decoder, and Phase Tracking.) do. Therefore, designing a high-speed low-complexity interleaver/deinterleaver for the 802.11n WLAN is very important.[2] The interleaver used in IEEE 802.11a/g/n WLAN is block interleaver with block size corresponding to the number of bits in a single OFDM symbol. Let k be the index of the coded bits before the first permutation, let i be the index after the first permutation, j be the index after the second permutation. The first permutation which ensures adjacent coded bits mapped onto nonadjacent subcarriers is defined as:


$$i = N_{row}(k \bmod N_{col}) + \text{floor}(k/N_{col}) \quad (1)$$

The second permutation ensures that adjacent coded bits are mapped alternately onto less and more significant bits of the constellation and, thereby, long runs of low reliability bits are avoided. It is defined as:

$$j = s(i_{ss}) \times \text{floor}(i/s(i_{ss})) + (i + N_{cbpss}(i_{ss}) - \text{floor}(N_{col} \times i/N_{cbpss}(i_{ss}))) \times \text{mod}(s(i_{ss})) \quad (2)$$

Due to complex mathematical computation in (1)-(2), the direct arithmetic way to yield the permuted sequence is hardware inefficient. In order to avoid the hardware intensive blocks, for example, dividers, multipliers, and so forth, those complex equations are further replaced with optimized hardware structures. In (1)-(2), the parameter $s(i_{ss})$ is 1 for both BPSK and QPSK, 2 for 16QAM, and 3 for 64QAM. Next we consider the three different cases separately. For the BPSK and QPSK case, (2) can be rewritten as $j = i$. For the 16QAM case, the parameter $N_{cbpss}(i_{ss})$ is integer times of 2. For the 64QAM case, the parameter $N_{cbpss}(i_{ss})$ is integer times of 3. By analyzing the resulting equations, we'll arrive at the following figures:

→ permuted_sequence


(0)	(N_{row})	$(2N_{\text{row}})$	\dots
(1)	$(1 + N_{\text{row}})$	$(1 + 2N_{\text{row}})$	\dots
(2)	$(2 + N_{\text{row}})$	$(2 + 2N_{\text{row}})$	\dots
\vdots	\vdots	\vdots	

N_{col}

N_{row}

Figure 8: Permuted sequence from (1)-(2) for BPSK and QPSK

→ permuted_sequence


(0)	(N_{row}) +1	$(2N_{\text{row}})$	\dots
(1)	$(1 + N_{\text{row}})$ -1	$(1 + 2N_{\text{row}})$	\dots
(2)	$(2 + N_{\text{row}})$ +1	$(2 + 2N_{\text{row}})$	\dots
\vdots	\vdots	\vdots	

N_{col}

N_{row}

Figure 9: Permuted sequence from (1)-(2) for 16QAM

→ permuted_sequence

(0)	(N_{row}) +2	$(2N_{\text{row}})$ +1	\dots
(1)	$(1 + N_{\text{row}})$ -1	$(1 + 2N_{\text{row}})$ +1	\dots
(2)	$(2 + N_{\text{row}})$ -1	$(2 + 2N_{\text{row}})$ -2	\dots
\vdots	\vdots	\vdots	

N_{col}

N_{row}

Figure 10: Permuted sequence from (1)-(2) for 64QAM

c_f \ r_f	0	1
0	0	1
1	0	-1

(a)

c_f \ r_f	001	010	100
001	0	2	1
010	0	-1	1
100	0	-1	-2

(b)

Figure 11: Submatrixes for (a) 16QAM and (b) 64QAM

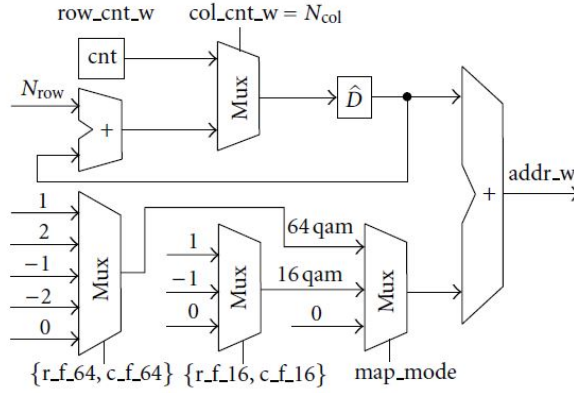


Figure 12: Proposed hardware structure for (1)-(2)

It can be seen that the sequences inside brackets are identical for all the three cases. Hence, the permuted sequence from (1)-(2) can be represented as the summation of identical sequence with offset sequence. For the BPSK and QPSK case, the offset sequence is all-zero sequence. For the 16QAM case, only 0, +1, and -1 are consisted in the offset sequence. And for the 64QAM case, the offset sequence is formed with 0, +1, -1, +2, and -2. To efficiently generate the offset sequence in hardware, a novel solution that divides the interleaving matrix into several small submatrixes is proposed. Figure 11 shows the submatrixes for 16QAM and 64QAM, where the row flag (r.f) and column flag (c.f) are used to determine the location of each index, thereby, the offset sequence can be generated using a few multiplexers and cyclic shift registers. Eventually, (1)-(2) can be implemented using an optimized hardware structure as shown in Figure 12. The upside circuits are used to generate the identical sequence, whereas the three underside multiplexers are used to generate the offset sequence.

Now with all these descriptions, the code becomes clear. We strongly believe

that this optimization is eligible for a [Bonus](#). It greatly optimizes the hardware rather than just mapping through hardware inefficient multipliers and dividers and floors.

6.2 HDL Implementation

By opening the subfolder "Verilog" in folder "Phase_3" you'll observe two folders "interleaver" and "de-interleaver". In the "interleaver" you'll find these verilog files: counter8.v, interleave2_odd.v, interleave2_even.v, regulator.v, and interleaver.v.

interleaver.v: First we are in 's0' state. 'odds' and 'evens' are two shift registers that two input wires are connected to them. After taking 'count_end' (count_end is a number) valid inputs, the signal 'gozero' will be set to one and we go to the state 's1'. 'count_end' depends on N_cbps('mod') that is specified in the 'counter8' module. After going to state 's1' by any valid input, we go to 's2', 's3', ..., 's8' state. In each of these states the content of the 'inter1_evens' and 'inter1_odds' is valid. These wires are connected to some points of shift registers in such a way that output of them be the output of data after first interleave. when 'mod' is 00 the output is on 'inter1_X[2:0]', when 'mod' is 01 the output is on 'inter1_X[5:0]' and so on. Of course in any of the 'mod's we give output in 8 clocks but with different throughput. 'interleave2_odd' and 'interleave2_even' modules perform the second interleave only by changing 'inter_mod_X' in 's1' to 's8' state. In 'pre_x_X' signals output arrangement is last to first and different 'mod's are incompatible so by using the 'regulator' module we sort them from first to last so the output of this module is easier to process for other modules.

In the "de-interleaver" you'll find these Verilog files: counter8.v, interleave2_odd.v, interleave2_even.v, and deinterleaver.v. In this module we almost do the inverse process of 'interleaver' module with similar structure in such a way that we take input in first 8 clocks but its output is valid in the whole time.

cr_matcher: This is used in integration phase (refer to figures [13](#), [14](#)).

This module is created to make the input of 'interleaver' compatible to it in different 'cr's (coding rates) so that the 'interleaver' sees the same pattern of input in every 'cr'. The difference is that only in some of the clocks the 'valid' signal may be zero. To do this it uses a simple FSM and flexible data plane.

This module enables the correct performance of encoder and transmitter for **all data rates**. Hence we believe it is eligible for a [Bonus](#) because only two rates were asked.

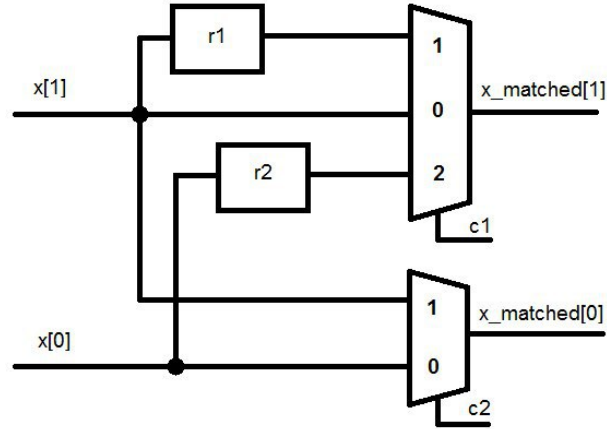


Figure 13: cr_matcher data plane

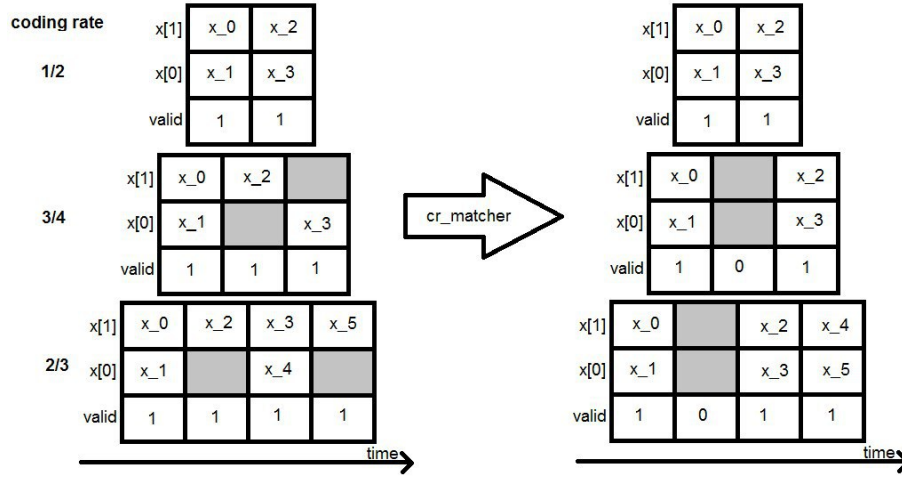


Figure 14: cr_matcher operation

These are symbols used in Verilog modules:

mod	N_cbps	N_bpssc	modulation
00	48	1	BPSK
01	96	2	QPSK
10	192	4	16-QAM
11	288	6	64-QAM

Table 7: symbols used in verilog codes

inter_mod	interleave_type
00	without interleave
01	displace odds and evens
10	in 3 bits first goes last
11	in 3 bits last goes first

Table 8: symbols used in verilog codes

cr	coding_rate
00	1/2
01	2/3
10	3/4
11	3/4

Table 9: symbols used in verilog codes

6.2.1 Performance Discussion

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	87	301440	0%
Number of Slice LUTs	168	150720	0%
Number of fully used LUT-FF pairs	83	172	48%
Number of bonded IOBs	44	600	7%
Number of BUFG/BUFGCTRLs	1	32	3%

Table 10: Interleaver utilization summary on Virtex 6 FPGA

Minimum period	2.583ns (387.222 MHz)
Minimum input arrival time before clock	2.491ns
Maximum output required time after clock	3.357ns
Maximum combinational path delay	3.090ns

Table 11: Interleaver timing analysis on Virtex 6 FPGA

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	121	301440	0%
Number of Slice LUTs	166	150720	4%
Number of fully used LUT-FF pairs	116	171	67%
Number of bonded IOBs	44	600	7%
Number of BUFG/BUFGCTRLs	1	32	3%

Table 12: De-Interleaver utilization summary on Virtex 6 FPGA

Minimum period	2.493ns (401.204 MHz)
Minimum input arrival time before clock	2.280ns
Maximum output required time after clock	2.997ns
Maximum combinational path delay	2.765ns

Table 13: De-Interleaver timing analysis on Virtex 6 FPGA

7 Phase 4, Mapper, Demapper and Pilot Inserter, and Remover

For phase 4, we will discuss the implementation of Mapper/Demapper and Pilot Inserter/Remover blocks.

7.1 Matlab Implementation

By opening the the subfolder "Matlab" in the folder "Phase_4" you'll find the following .m files:

- rateDependents.m: {inputs: dataRate} {outputs: Modulation, N_BPSC, N_CBPS, N_DBPS} — Explained in Phase 3.
- main.m:
We've used this file in phases 1,2, and 3. We'll continue from Phase 4 section which could be easily found in the file.
 - section 4.1: You may choose a rate, then modulation and N_CBPS is retrieved from the function "rateDependents" just like before. Random data is generated, mapped, demapped, and the BER of the operation is obtained and displayed. If you run this section and obtain zero, it means the job is done.

- section 4.2: We know that the polarity of pilots comes from a specific pseudo random binary sequence. First this sequence is generated, then the mapped data from previous section is divided into blocks of size 48 (48 subcarriers), then 4 pilots are added to it (along with the zero in the middle) and pilot insertion completes. After that, these blocks are passed to `pilotRemove` to omit those 4 pilots and the zero and return the pilot removed data, which is itself identical to the mapped data. The BER of this operation is also displayed in the end.
- `mapper.m`: {inputs: data, mapMode, Kmod} {outputs: y} — This function takes data, modulation scheme, and a boolean variable Kmod. If the latter is true, then a normalizer is multiplied with the output of the mapper according to the standard[1] (on page 19). Since essentially the D/A and A/D need to take care of these analog signals, if this argument is set to false, then no normalizer is used and the values are digital-friendly. (This is the way we’ve coded the mapper, we’ve ignored the Kmod) So this argument is set to **FALSE** when passing test vectors to HDL test bench o.w. the BER won’t be zero.
- `demapper.m`: {inputs: mappedData, mapMode, Kmod} {outputs: y} — This function takes mappedData, modulation scheme, and the previously mentioned boolean variable Kmod. If the latter is true, then the normalizer is compensated at the output of the demapper according to the standard[1] (on page 19). Again, since this is the way we’ve coded the demapper, it is **CRITICAL** to set this argument to **FALSE** when passing test vectors to HDL test bench o.w. the BER won’t be zero. If software verification is done, Rounding which is evident in the comments is **EXTREMELY CRITICAL**! It was the root cause of severe mismatch when I was testing it! Because mapped data isn’t precise after Kmod multiplication, for arriving at demapped symbols, we **MUST** do the rounding.
- `pilotInsert.m`: {inputs: symbolBlock, blockID, pseudoRandomSeq} {outputs: y} — This function takes symbolBlock, blockID, and the pseudoRandomSeq. According to the standard[1] (on page 19), first element of the pseudoRandomSeq determines the polarity of pilots for SIGNAL FIELD symbol. That’s why there’s a block ID, other symbols also use this variable to obtain their pilots. After that, a simple series of assignments is enough to insert the pilots.
- `pilotRemove.m`: {inputs: pilotInsertedSymbols} {outputs: y} — This function takes pilotInsertedSymbols and after a simple series of assignments returns the pilot removed symbols.

7.2 HDL Implementation

By opening the subfolder "Verilog" in folder "Phase_4" you’ll observe two folders: `mapper` and `demapper`. In folder "mapper" you’ll see these verilog files: `mapper.v`,

mapper_pi:

The 'mapper' module only does a combinational mapping from inputs to complex numbers represented in 8 bit, 4 bit real and 4 bit imaginary, and each, in sign-magnitude format. The mapping depends on the modulation type. the 'mapper_pi' consist of six 'mapper' modules that make six subcarriers content in each clock, also it has a 'scrambler' module to make pilot content and give it to output in this 8 clock with four lines. (hence the names is mapper + pilot)

In folder "mapper" you'll see these verilog files: demapper.v, demapper_pr:

This module only performs inverse mapping of the 'mapper_pi' module and don't use pilots to remove them! (Ignore the warnings because they refer to pilots not being used which is natural since the implementation doesn't go further to ifft and fft blocks.)

7.2.1 Performance Discussion of integrated TX and RX

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	169	301440	0%
Number of Slice LUTs	272	150720	0%
Number of fully used LUT-FF pairs	143	298	47%
Number of bonded IOBs	67	600	11%
Number of BUFG/BUFGCTRLs	1	32	3%

Table 14: TX utilization summary on Virtex 6 FPGA

Minimum period	3.389ns (295.072MHz)
Minimum input arrival time before clock	1.496ns
Maximum output required time after clock	1.530ns

Table 15: TX timing analysis on Virtex 6 FPGA

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	464	301440	0%
Number of Slice LUTs	6641	150720	4%
Number of fully used LUT-FF pairs	201	6904	2%
Number of bonded IOBs	55	600	9%
Number of BUFG/BUFGCTRLs	1	32	3%

Table 16: RX utilization summary on Virtex 6 FPGA

Minimum period	12.213ns (81.879 MHz)
Minimum input arrival time before clock	13.620ns
Maximum output required time after clock	0.777ns

Table 17: RX timing analysis on Virtex 6 FPGA

8 Integration, Matching and Verification

8.1 Matlab Implementation

This section is organized in two code sections:

8.1.1 Code Section 5.1

In section 5.1, I've implemented the whole system and tested standard's example to show you the walk through of transmitter and receiver procedures. **Please run section 1.7 before running section 5.1.** Please consider giving a [Bonus](#) if you find this section useful and clean. For you convenience, I've even constructed tables of standard [1] in annex G so that you can quickly verify the correctness of these operations. Since there's a huge amount of comments there, I'll briefly describe what goes on there.

1. DATA field for the standard's sample message (SM) is created and scrambled. SIGNAL field does NOT requier scrambling.
2. 6 bits of TAIL should be set to zero to initialize the encoder.
3. Convolutional encoding on DATA field is performed with respect to the data rate. SIGNAL field is encoded with fixed 1/2 encoding rate.
4. Interleaving for DATA and SIGNAL fields is performed.
5. Symbol mapping for DATA and SIGNAL fields is performed.
6. Pilot insertion for DATA and SIGNAL fields is performed.
7. Pilot removal for DATA and SIGNAL fields is performed.
8. Symbol demapping for DATA and SIGNAL fields is performed.
9. De-interleaving for DATA and SIGNAL fields is performed.
10. Viterbi Decoding for DATA and SIGNAL fields is performed.
11. Descrambling for DATA field is performed.

8.1.2 Code Section 5.2

In section 5.2, I've created modules that both enable complete and automatic verification of the results, as well as enabling the user to send **as many messages** as he/she wants, i.e. the code is written in a way that does not limit the user to generate one message and verify the system. You can determine what number of messages you wish to be transmitted and received, then all the PSDUs for these messages are created along with their random length, data rates, SIGNAL and DATA fields. Also you may choose the maximum length of PSDUs that are going to be generated. Your last degree of freedom is the stage. This variable determines up until which block in transmitter and receiver are going to be verified and are in play. If it's set to "scrambler", only scrambler and descrambler are present. If it's set to "mapper", all modules are present except for the pilot inserter and remover. Finally the BER for each of these configurations is presented. The automatic process is that after creation of DATA and SIGNAL fields, SIGNALs are transmitted via transmitterSignal and receiver decodes them via receiverSignal. Now receiver has all the parameters required for decoding each message so recvSIGNAL is the input for receiver. Please consider giving a [Bonus](#) if you find this section useful and clean as well as section 5.1. This amount of automation and modular coding took a lot of time. Software integration works for all rates and all number of samples with any configuration you wish.

In the following I'll briefly explain .m files that were NOT present in previous phases:

- codedRate2dataRate.m: {inputs: x} {outputs: dataRate} — When we read first 4 bits of SIGNAL in the receiver, we need to determine the data rate. This function provides this mapping.
- correctnessCheck.m: {inputs: PSDUs, txStream, rxStream, stage, isDATA} {outputs: NONE} — This function gives the BER of the automated process in section 5.2 for various configurations. isDATA determines whether this function is used to check the correctness of received SIGNAL fields or DATA fields because the process for them differs a little bit. If it's comparing SIGNAL fields, it simply sums the xor of txStream and rxStream and displays the result as the BER. If it's the DATA field and the stage is not scrambler, because of TAIL bits becoming zero in tx side, it's nonsense to compare descrambled data, then we need to compare the resulting PSDUs of receiver with the transmitter. If the stage is scrambler, since no TAIL bit is set to 0, it is ok to compare descrambled DATA fields with the originals and give the resulting BER.
- receiver.m: {inputs: channelDATA, recvSIGNAL, stage, scramInit, trellis, predecessors} {outputs: output} — Takes DATA fields from channel and uses scramInit and recvSIGNAL along with trellis structure and predecessors to decode DATA fields of messages coming in from the channel. It's output depends on the stage.

- receiverSignal.m: {inputs: channelSIGNAL, stage, trellis, predecessors} {outputs: dataRate} — Utilized for receiving SIGNAL fields of the messages. It's output depends on the stage.
- transmitter.m: {inputs: DATA_FIELDS, dataRates, Lengths, stage} {outputs: channelDATA, scramInit} — Takes DATA fields from higher network layers and uses scramInit and data rates along with lengths and stage to encode DATA fields of messages and puts them on the PHY channel. It's output depends on the stage. scramInit output of this module is used in receiver for descrambling.
- trasmitterSignal.m: {inputs: SIGNALS, stage} {outputs: channelSIGNAL} — Utilized for sending SIGNAL fields of the messages.
- viterbiSetup.m: {inputs: NONE} {outputs: trellis, predecessors} — Creates trellis structure and predecessors for the use of Viterbi decoder according to standard [1].
- testGenerator.m: {inputs: numOfSamples, MaximumLength} {outputs: DATA_FIELDS, SIGNALS, PSDUs, Lenghts, bitRates} — Takes as input number of messages and their maximum PSDU length and returns their PSDUs and DATA fields, etc. This function summarizes sections 1.1 to 1.3 of the code.

IMPORTANT: In all of the following files, Kmod boolean in mappers and demappers is set to false:

transmitter.m, trasmitterSignal.m, receiver.m, receiverSignal.m

The reason was mentioned earlier. Since these introduce inaccuracies and hardware cannot handle them itself (it requires D/A and A/D), we do not include the Kmod normalizer. Although you may set it to true in all of these 4 files and observe that our software implementation is ok with that. (Because of rounding in the demapper) But for HDL verification, we have set them to false.

8.1.3 Code Section 5.3

In this section we generate txt files for test bench. NOTE that since our hardware decoder and encoder only work for 1/2 rate, we should generate messages with data rate of either 6, 12, or 24. You may change that in the code. CRITICAL: If you changed this, also change the Rate reg in txrx_tb to be consistent. The comments there will help you for this. Also, NOTE that the input for our hardware is DATA field, not the PSDU. That's why our test vector is DATA field. Finally, test and golden vectors are written in folder "Integration" (relative to where the main.m file is).

8.2 HDL Implementation

By opening the subfolder "Verilog" in folder "Integrated" you'll observe these folders: EncoderDecoder, Rx, Tx. You'll also find PHY.v, RX.v, TX.v, txrx_tb.v Verilog files.

EncoderDecoder just contains all files from Phase 2 (except for the test bench). Tx contains all files from interleaver, mapper, encoder, cr_matcher, and scrambler. Rx contains all files from de-interleaver, demapper, dencoder, and descrambler. RX.v is the module for receiver and TX.v the module for transmitter. PHY.v is the major module of WLAN PHY which we instantiate in our test bench txrx_tb. TX supports all rates (which is eligible for a [Bonus](#) because not all rates for transmitter were asked.) but RX only supports rate = 1/2. For verification, copy phy_in.txt and phy_golden_outputs_m.txt from "Integration" folder (where main.m is) to wherever the txrx_tb test bench is. Also put all Verilog files in one place, synthesize PHY.v and all should be synthesized. Run the simulation and have fun!

Thanks for putting your time into reading this kindly and patiently report. Please don't forget to consider [Bonuses](#)! We're glad that we learned a lot from this project. That's all! Bye!

References

- [1] Ieee standard for telecommunications and information exchange between systems - lan/man specific requirements - part 11: Wireless medium access control (mac) and physical layer (phy) specifications: High speed physical layer in the 5 ghz band. *IEEE Std 802.11a-1999*, pages 1–102, 1999.
- [2] Zhen-dong Zhang, Bin Wu, Yumei Zhou, and Xin Zhang. Low-complexity hardware interleaver/deinterleaver for IEEE 802.11a/g/n WLAN. *VLSI Design*, 2012: 948957:1–948957:7, 2012. doi: 10.1155/2012/948957. URL <https://doi.org/10.1155/2012/948957>.