Hardware Implementation of an 802.11a Transmitter

Group 1: Elizabeth Basha, Steve Gerding, and Rose Liu
{e_basha|sgerding|rliu}@mit.edu

1 Introduction

In this work, we discuss the design and implementation of an 802.11a Transmitter. 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 is shown in Figure 1.

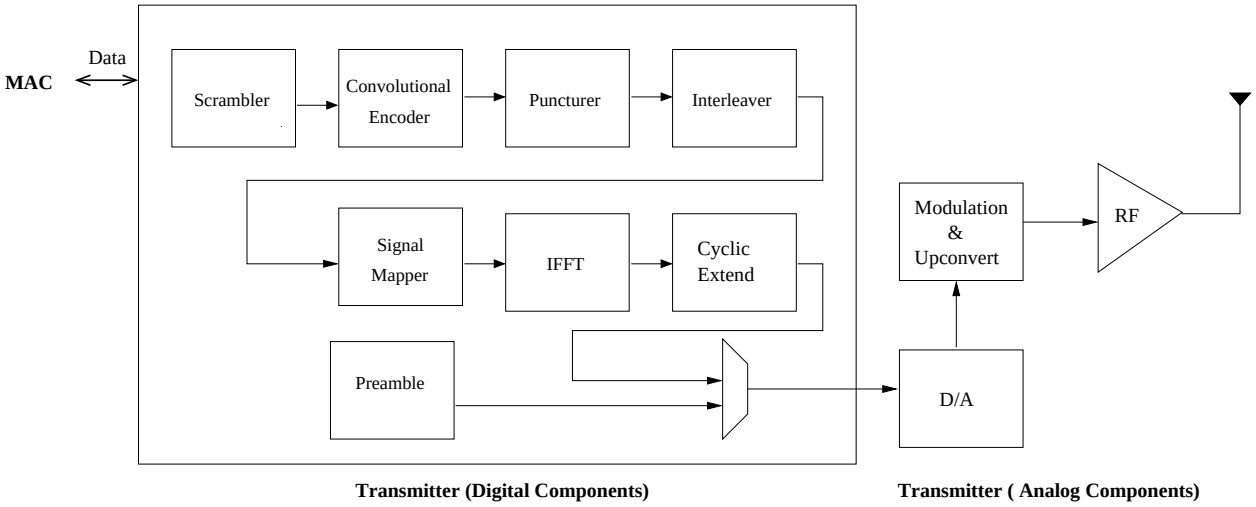


Figure 1: Top Level Diagram of an 802.11a Transmitter.

1.1 Specifications

Our implementation supports the mandatory data rates of 6, 12 and 24 Mbits/s. The modulation parameters corresponding to these data rates are shown in Figure 2.

Data rate	Modulation	Coding Rate	Coded bits/sub-carrier	Coded bits/OFDM Symbol	Data bits/OFDM symbol
(Mbits/s)		(R)	(N_{bpsc})	(N_{cbps})	(N_{dbps})
6	BPSK	1/2	1	48	24
12	QPSK	1/2	2	96	48
24	16-QAM	1/2	4	192	96

Figure 2: Modulation Parameters Corresponding to Data Rates.

Since we implement only the Transmitter side of the 802.11a specification, we support the following subset of services/messages to the Medium Access Control (MAC) layer.

- `PHY_TXSTART.req(LENGTH,DATARATE,SERVICE,TXPWR_LEVEL): MAC transmit request & parameters`
- `PHY_TXSTART.conf: confirmation of transmit request`
- `PHY_DATA.req(DATA): MAC request to send data`
- `PHY_DATA.conf: confirmation that data was received from MAC`

1.2 Design Exploration Overview

We have three main design goals: minimize area, minimize power, and achieve just-in-time performance to meet the 6, 12, and 24Mb/s data rates. After ensuring the functional correctness of our system, we explore techniques such as reducing frequency and V_{DD}. In addition, we minimize the area of our design by reusing and sharing logic units, which also reduces leakage power in our system.

In Section 2 we touch upon some specific design choices at the module level. The techniques we explore are summarized below.

- Vary input data frame size
- Use lookup tables to eliminate the need for multipliers
- Unroll serial algorithms to improve performance
- Determine whether an unrolled IFFT algorithm or reuse of one stage of the design will provide the best power, area, and performance trade-off.

1.3 Implementation Strategy

We implement our design using Bluespec, a high-level hardware description language. Bluespec supports explicit and implicit synchronization primitives, allowing us to describe all actions and transactions atomically. We compile our design into Verilog using the Bluespec compiler. We also use Synopsys VCS for simulation, Design Compiler for synthesis, and Encounter for place and route.

1.4 Test Strategy

Because the 802.11a Transmitter can be decomposed into separate, well defined blocks, we implement and verify these blocks independently. To facilitate the testing of our modules, we use the framework of the Extreme Benchmark Suite (XBS), shown in Figure 3. For each module, an XBS input generator is constructed to generate test input patterns to stress the module. The test patterns are a combination of random input data and directed tests designed to test corner cases of the design. An XBS output checker is constructed for each module. Once a module is implemented in Bluespec, it is outfitted with the XBS Bluespec test harness and subjected to the aforementioned tests.

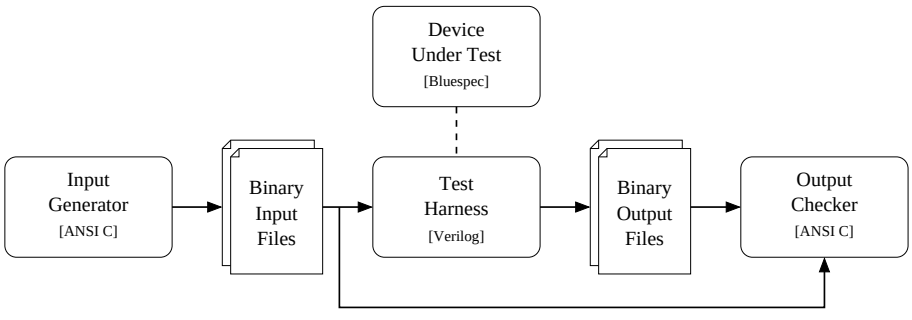


Figure 3: Test Strategy using XBS.

Once we implement and test the individual modules, we verify the entire system by combining our modules and testing them using XBS, as described above.

2 System Overview

Transmission is initiated by the MAC through a PHY_TXSTART.request, which passes the LENGTH, DATARATE, SERVICE, and TXPWR_LEVEL parameters to the Controller module. The LENGTH parameter, ranging from 1 to 4095, defines the number of data octets in the packet. The DATARATE parameter describes the bit rate at which the packet is transmitted. The received SERVICE field from the MAC always consists of 16 zero bits. The TX-PWR_LEVEL parameter ranges from 1 to 8, and indicates which power level should be used for transmission.

Once the Controller has received the PHY_TXSTART request, it generates and transmits the preamble and header sections of the PPDU frame format (format of message). The PPDU transmission format consists of a preamble, header, and data as shown in Figure 4.

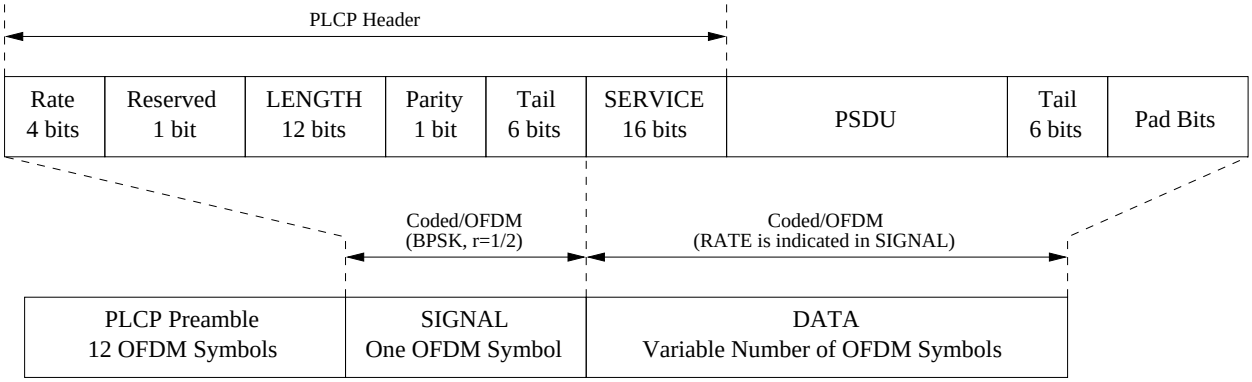


Figure 4: Format of Message: PPDU Frame Format.

2.1 Preamble Generation

The preamble is used by the receiver for timing synchronization purposes. It consists of 10 repetitions of a short training sequence, and 2 repetitions of a long training sequence. The short training sequence is composed of 12 sub-carriers modulated by the elements of the fixed sequence S, given by

S = $\sqrt{\frac{13}{6}}$ * {0, 0, 1+j, 0, 0, 0, -1-j, 0, 0, 0, 1+j, 0 ,0 ,0, -1-j, 0, 0, 0, -1-j, 0, 0, 0, 1+j, 0, 0, 0, 0, 0, 0, 0, -1-j, 0, 0, 0, -1-j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0, 0}

The long training sequence is composed of 53 sub-carriers which are modulated by the fixed sequence L, given by

L = {1, 1, -1, -1, 1, 1, -1, 1, -1, 1, 1, 1, 1, 1, 1, -1, -1, 1, 1, -1, 1, -1, 1, 1, 1, 1, 0, 1, -1, -1, 1, 1, -1, 1, -1, 1, -1, -1, -1, -1, -1, 1, 1, -1, -1, 1, -1, 1, -1, 1, 1, 1, 1}

Since the Preamble is fixed, it can be pre-computed and stored in the Controller to be transmitted whenever a new transmit request arrives.

2.2 Header Generation

As shown in Figure 4, the main components of the header are the RATE and LENGTH fields, which are given in the transmit request from the MAC. The reserved and tail bits are always 0. The header is always encoded with BPSK using a coding rate R = 1/2 at 6 Mbits/s. Once encoded, the header is sent through the rest of the units (Puncturer, Interleaver, Mapper, IFFT, and Cyclic Extend) with the appropriate control signal to specify the 6Mbits/s data rate. Note that the header is not scrambled.

After forming the header and sending it to the Encoder, the Controller calculates and sets the rate dependent modulation parameters (listed in Figure 2) for data transmission. Modules which are data rate dependent will receive a control message from the Controller to be configured accordingly.

The Controller then forms the data section of the PPDU frame format by appending the main message data to the SERVICE field and adding trailing 0s. The resulting packet is then sent to the Scrambler.

2.3 Scrambler

A simple implementation of the Scrambler consists of 7 shift registers and 2 XORs as shown in Figure 5. The Scrambler is of length-127, meaning it repeatedly generates a 127-bit sequence for a given pseudo-random initial state. Each incoming data bit is XORed with the current bit in the 127-bit sequence.

The first 7 bits to be sent into the Scrambler are the beginning of the SERVICE parameter. These 7 bits are re-written with the initial state of the Scrambler so descrambling can be done in the receiver.

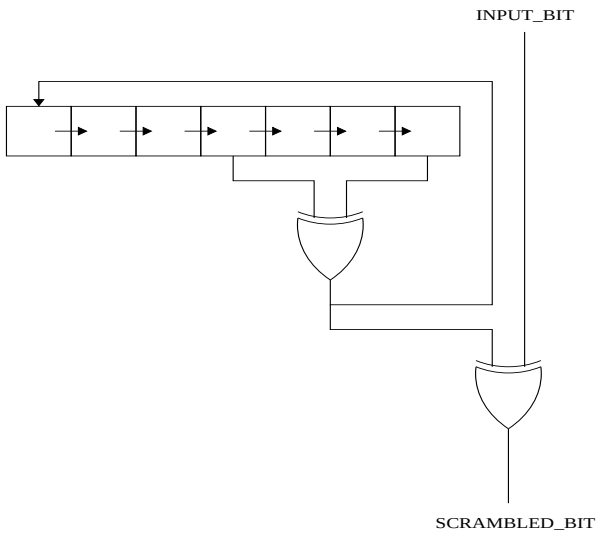


Figure 5: Simple Implementation of Scrambler.

After reaching the end of the message data, the Scrambler replaces the trailing scrambled zero bits with non-scrambled zero bits to reset the Convolutional Encoder for the next message.

2.4 Convolutional Encoder and Puncturer

As shown in Figure 6, a simple version of a Convolutional Encoder consists of 7 shift registers and 3 XORs. Each input bit into the Encoder produces 2 output bits (even and odd). The even bit should be read by the Puncturer before the odd bit.

The Puncturer is not needed for the data rates we are implementing. Therefore we decided to omit the Puncturer in our design.

2.5 Interleaver

The Interleaver operates at an OFDM symbol level with a block size of N_{cbps} (48, 96, or 192) bits. Within each block, the bits are reordered in two steps. In the first step, adjacent coded bits are reordered to map to nonadjacent sub-carriers. In the second step, adjacent coded bits are mapped alternately into less and more significant bits of the sub-carrier constellation.

Let k denote the index of the coded bit before interleaving, i denote the index after the first step, and j denote the index after the second step. The value s = max(N_{bpsc}/2,1).

$i = (\frac{N\_cbps}{16}) * (kmod16) + floor(k/16), k = 0...N\_cbps - 1$

$j = s * floor(\frac{i}{s}) + (i + N\_cbps - floor(16 * \frac{i}{N\_cbps}))mods, i = 0...N\_cbps - 1$

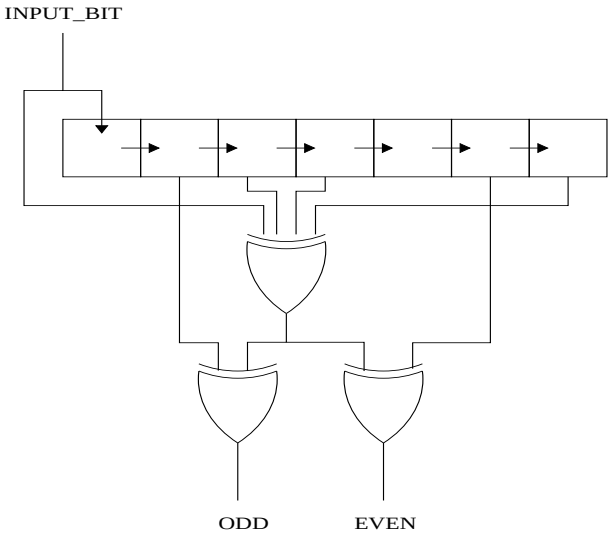


Figure 6: Simple Implementation of Convolutional Encoder.

Since these indices can be calculated beforehand, the Interleaver basically reorders the input bits in a set pattern based on the data rate.

2.6 Mapper

The Mapper also operates at the OFDM symbol level with a block size of N_{cpsc} (48, 96, or 192) bits. Each block is divided into sub-blocks at the OFDM sub-carrier level. Sub-blocks are of size N_{bpsc} (1, 2, or 4) bits. The Mapper (stage-1) first converts each sub-block into a complex number representing BPSK, QPSK, or 16-QAM constellation points. Note that the modulation type may be different for the header and data parts of the message. The resulting 48 complex pairs are then normalized by K_{MOD}. The I, Q, and K_{MOD} values for each modulation type are shown in Figures 7 to 10.

Modulation	K_{mod}
BPSK	1
QPSK	$\frac{1}{\sqrt{2}}$
16-QAM	$\frac{1}{\sqrt{10}}$

Figure 7: K_{MOD} Table.

in_0	I-out	Q-out
0	-1	0
1	1	0

Figure 8: BPSK encoding for I and Q values.

After converting each of the sub-blocks into complex pairs, the Mapper (stage-2) then collects the 48 complex number outputs (one OFDM symbol) and maps each complex number to one of 48 sub-carriers represented as frequency offset indices. Pilots (also complex pairs) are inserted in the four other sub-carriers to ensure robustness against frequency offsets and phase noise. The 52 total sub-carriers per OFDM symbol are indexed -26 to 26, with the 0 sub-carrier omitted and filled with a zero. These 52 complex pairs are then padded to create 64 complex pairs. The final

in_0	I-out	in_1	Q-out
0	-1	0	-1
1	1	1	1

Figure 9: QPSK encoding for I and Q values.

(in_0,in_1)	I-out	(in_2, in_3)	Q-out
00	-3	00	-3
01	-1	01	-1
11	1	11	1
10	3	10	3

Figure 10: 16-QAM encoding for I and Q values.

output of the Mapper consists of 64 complex pairs each ordered to a frequency offset index in the OFDM symbol.

2.7 IFFT and Cyclic Extension

The Inverse Fast Fourier Transform (IFFT) module converts the complex frequency values into complex time values for transmission. There is no specification on the size or representation of the data nor the algorithm required.

After the data transformation, cyclic extension is performed to form a guard interval.

2.8 Data Rates

The relative data input-output bit ratios for each module are shown in Figure 11. BITWIDTH_IQ is the bit width we use to represent the I and Q values of the complex numbers. For the Interleaver and Mapper, the ratios change depending on the data rate.

	Bits In	Bits Out
Scrambler	1	1
Conv Encoder	1	2
Puncturer	2	2
Interleaver	48, 96, or 192	48, 96, or 192
Mapper(stage-1)	48, 96, or 192	48*(2*BITWIDTH_IQ)
Mapper(stage-2)	48*(2*BITWIDTH_IQ)	64*(2*BITWIDTH_IQ)
IFFT	64*(2*BITWIDTH_IQ)	64*(2*BITWIDTH_IQ)
Cyclic Extend	64*(2*BITWIDTH_IQ)	81*(2*BITWIDTH_IQ)

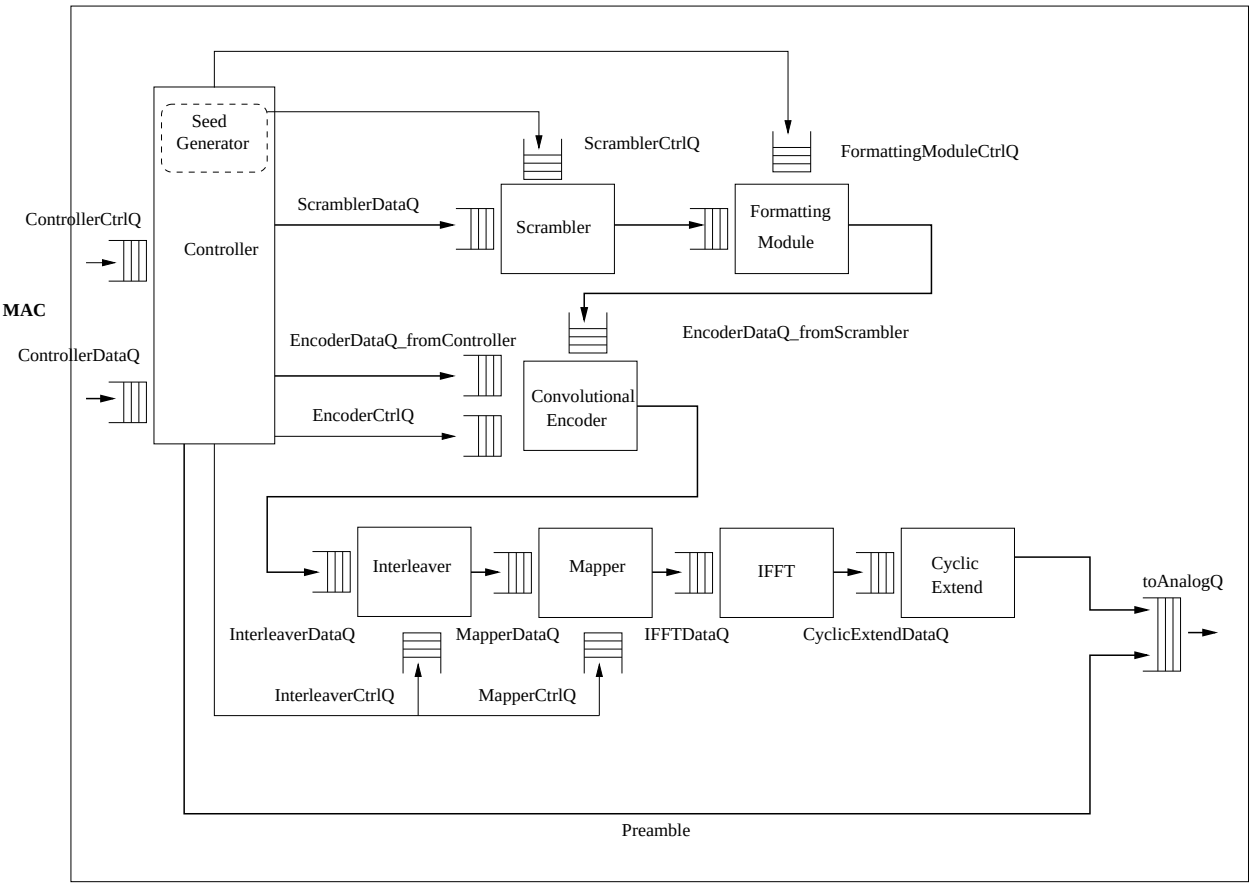
Figure 11: Input-output bitwidths of each module.

3 UTL Model of Transmitter

In the following subsections, we give a general Unit- Transaction Level (UTL) description of each module. Figure 12 is a top level UTL model of the Transmitter.

3.1 System Assumptions

In order to focus on more detailed design exploration issues, we made the following system level assumptions to simplify the overall Transmitter design.



Transmitter Top Level Module

Figure 12: Top Level UTL Model of Transmitter.

- Preamble: Since the preamble is static for every message, we assume another module after the Transmitter will prepend the preamble.
- Data Padding: We assume all data padding and tail bits are added in the MAC layer.

3.2 Controller

input:
- ControllerCtrlQ:{Ctrl_type, rate, header length, message length, data length}
- ControllerDataQ:{FRAMESIZE}

output:
- ScramblerCtrlQ{data length},
- ScramblerDataQ: {FRAMESIZE},
- FormattingModuleCtrlQ {message length, data length},
- EncoderDataQ_fromController:{FRAMESIZE},
- EncoderCtrlQ:{header_length, data_length},
- InterleaverCtrlQ:{rate, length},
- MapperCtrlQ:{rate, length, reset},

architectural state: rate, header length, message length, data length, txpwr_level, frames_left_to_read

Transactions:

1. TransmitHeader(header length, data length, rate)

```
(if ControllerCtrlQ.first.Ctrl_type == transmit)
and (frames_left_to_read == 0))

a. Set frames_left_to_read
b. TransmitHeader(rate, header length)
   - Form header packet: Hdr
   - EncoderCtrlQ.enq(header_length, data_length);
   - EncoderDataQ_fromController.enq(Hdr)
   - InterleaverCtrlQ.enqueue(6Mbits/s, header length)
   - MapperCtrlQ.enqueue(6Mbits/s, header length, reset=1)

c. Initialize Scrambler:
   - Generate random 7 bit sequence
   - ScramblerCtrlQ.enqueue(7-bit seq, length)

d. Initialize Formatting Module
   - FormattingModuleCtrlQ.enq(message length, data length)
```

2. TransmitData(data length, rate, data)

```
(if ControllerDataQ not empty and (frames_left_to_read > 0))

   a. Set rate for data transmission if have not yet done so
      - InterleaverCtrlQ.enqueue(Rate, data length)
      - MapperCtrlQ.enqueue(Rate, data length, reset=0)

   b. ScramblerDataQ.enq(data)
```

Scheduler: Priority in descending order: TransmitData, TransmitHeader

3.3 Scrambler

input:
- ScramblerCtrlQ:
  - {7 seed bits}
  - {data length}
- ScramblerDataQ:{Data}

output: EncoderDataQ_fromScrambler:{Scrambled Data}

architectural state: 7, 1-bit registers, octets_to_scramble

Transactions:

1. Initialize(7-bit seed, data length)

```
(if ScramblerCtrlQ not empty & octets_to_scramble == 0)
 // Only re-initialize when scrambling of all data using
 // old sequence is complete

a. (seed, data length) =  ScramblerCtrlQ.pop()
b. InitializeRegisters(seed)
c. octets_to_scramble = data length
```

2. Scramble()

```
(if (ScramblerDataQ is not empty) and (octets_to_scramble > 0))
 // octets_to_scramble records how many more octets to scramble
 // using a particular initialization sequence.

a. data_in = ScramblerDataQ.pop();
b. Scramble data_in
c. FormattingModuleDataQ.enq(ScrambledData)
d. Decrement octets_to_scramble
```

Scheduler: Priority in descending order: Initialize, Scramble

3.4 Formatting Module

input:
- FormattingModuleCtrlQ:{msg_length, data_length}
- FormattingModuleDataQ:{data}

output: EncoderDataQ_fromScrambler:{data (even, odd) pairs}

architectural state: msg_octets_remaining, data_octets_remaining

Transactions:

1. init()

```
(if FormattingModuleCtrlQ not empty and (msg_octets_remaining == 0))

 a. (msg_length, data_length) = FormattingModuleCtrlQ.pop()
 b. msg_octets_remaining = msg_length
 b. data_octets_remaining = data_length + 2 (service length)
```

2. format()

```
(if FormattingModuleDataQ not empty and (msg_octets_remaining > 0))

 a. data_in = FormattingModuleDataQ.pop()
 b. If data_octets_remaining == 0, 1 or 2:
data_in <= substitute 0s in for the 6 tail bits at correct byte.
 c. EncoderDataQ.enq(data_in)
 d. Decrement msg_octets_remaining and data_octets_remaining
```

Scheduler: Priority in descending order: init, format

3.5 Convolutional Encoder

input:
- EncoderCtrlQ:{hdr_length, data_length}
- EncoderDataQ_fromController:{data}
- EncoderDataQ_fromScrambler:{data}

output: InterleaverDataQ:{data (even, odd) pairs}

architectural state: 7 1-bit registers, hdr_octets_remaining, data_octets_remaining

Transactions:

1. init()

```
(if EncoderCtrlQ not empty and (hdr_octets_remaining == 0)
and (data_octets_remaining == 0))

 a. (hdr_length, data_length) = EncoderCtrlQ.pop()
 b. hdr_octets_remaining = hdr_length
 b. data_octets_remaining = data_length
```

2. Encode_Data_from_Controller()

```
(if EncoderDataQ_fromController not empty and (hdr_octets_remaining > 0))

 a. data_in = EncoderDataQ_fromController.pop()
 b. For each bit in data_in, compute output pairs.
 c. InterleaverDataQ.enq(data (even,odd) pairs)
 d. Decrement hdr_octets_remaining
```

3. Encode_Data_from_Scrambler()

```
 (if (EncoderDataQ_fromScrambler not empty) and
 (hdr_octets_remaining == 0) and  (data_octets_remaining > 0))

 a. data_in = EncoderDataQ_fromController.pop()
 b. For each bit in data_in, compute output pairs.
 c. InterleaverDataQ.enq(data (even,odd) pairs)
 d. Decrement data_octets_remaining
```

Scheduler: Priority in descending order: init, Encode_Data_from_Controller, Encode_Data_from_Scrambler

3.6 Interleaver

input:
- InterleaverDataQ:{48 bits}. Depending on data rate, total data for symbol may span multiple 48-bit frames.
- InterleaverCtrlQ:rate,length

output: MapperDataQ:{48 bits}. Since output may span multiple 48-bit frames, may take multiple cycles to write out output.

architectural state: datarate, datarate_frames_remaining, symbol_frames_remaining

Transactions:

1. init(dataRate, length)

```
(if InterleaverCtrlQ not empty and datarate_frames_remaining == 0)

a. {rate,length} = InterleaverCtrlQ.deq()
b. datarate = rate
c. datarate_frames_remaining = length/octetsPerFrame
d. symbol_frames_remaining = octetsPerSymbol(datarate)/octetsPerFrame
```

2. Interleave(data)

```
(if InterleaverDataQ is not empty,
  and using the currently set datarate (datarate_frames_remaining > 0))

a. Interleave data
b. If symbol is complete (symbol_frames_remaining == 0):
    then: MapperDataQ.enq(InterleavedData), reset symbol_frames_remaining
   Else: decrement symbol_frames_remaining

c. Decrement datarate_frames_remaining
```

Scheduler: Priority in descending order: init, Interleave

3.7 Mapper

input:
- MapperDataQ:{48 bits}. Depending on data rate may take multiple cycles to read in total data for symbol
- MapperCtrlQ:rate,length, reset

output: IFFTDataQ:{64 complex pairs} (each I/Q is 16 bits)

architectural state: pilot polarity vector index, datarate, data rate_frames_remaining, symbol_frames_remaining

Transaction:

1. init(dataRate, length)

```
(if MapperCtrlQ not empty and datarate_frames_remaining == 0)

a. {rate,length,reset} = MapperCtrlQ.deq()
          - If (reset == 1) reset pilot polarity vector index.
b. datarate = rate
c. datarate_frames_remaining = length/octetsPerFrame
d. symbol_frames_remaining = octetsPerSymbol(datarate)/octetsPerFrame
```

2. Map(data)

```
(if MapperDataQ is not empty, and datarate_frames_remaining > 0)

a. Map data:
   - compute complex pairs for the coded bits of each sub-carrier
   - map complex pairs to sub-carrier indices
If completed mapping entire symbol (symbol_frames_remaining == 0):
   - reset symbol_frames_remaining
   - add pilots using pilot polarity vector, and index
   - increment pilot polarity vector index
   - pad to 64 complex pairs
   - IFFTDataQ.enq(MappedData)
Else: decrement symbol_frames_remaining

b. Decrement datarate_frames_remaining
```

Scheduler: Priority in descending order: init, Map

3.8 IFFT

input: IFFTDataQ:{64 complex pairs} represents 1 symbol in frequency domain

output: CyclicExtendDataQ:{64 complex pairs} in time domain

architectural state: none

Transactions:

1. IFFT(Data)

```
(if IFFTDataQ is not empty)

a. Perform IFFT on Data
b. CyclicExtendDataQ.enq(IFFTDataOut)
```

Scheduler: Priority in descending order: IFFT

3.9 Cyclic Extend

input: CyclicExtendDataQ:{64 complex pairs}

output: toAnalogQ:{64 complex pairs}

architectural state: none

Transactions:

1. CyclicExtend(Data)

```
(if CyclicExtendDataQ is not empty)

a. Perform CyclicExtension on Data
b. toAnalogQ.enq(CyclicExtendedData)
```

Scheduler: Priority in descending order: CyclicExtend

4 Testing

XBS is used to test individual modules and the system as a whole.

4.1 Individual Module Tests

Testing of each individual module requires the following components:

- XBS input generator
- XBS output checker
- XBS reference implementation used by output checker
- Verilog test harness that calls the Bluespec implementation

For individual module testing we rely on random input data as well as directed data to test corner cases. The following provides a general overview of the corner cases that are tested for some main modules.

Controller:
- Test for correct control messages when transmitting successive messages with different rates

Scrambler:
- Asymmetric 7-bit initial sequence
- Scramble multiple messages sequentially with varying initial sequences and lengths

Convolutional Encoder:
- Encode multiple messages with varying lengths

Interleaver & Mapper:
- Interleave and Map successive messages with different data rates and lengths

IFFT:
- Radix 4 Module: Pattern combinations of maximum and minimum numbers, and positive and negative numbers to double-check proper sign determination and overflow capabilities
- 64-point IFFT: Worst case positive/negative and minimum/maximum first and second stage input patterns

4.2 System Level Testing

System testing requires the following components:

- System level XBS input generator
- System level XBS output checker
- XBS reference implementation used by output checker
- Test harness for entire system

At the system level, we test that the transmitted packets contain all components of the message in the right order. We also make sure that data from different messages does not get mixed together, when we process parts of different messages in parallel. In addition, we ensure that the Transmitter meets the data rates of 6, 12, and 24 Mbits/s, as well as corner cases such as sending messages with the maximum and minimum lengths.

5 Initial Microarchitectural Design and Implementation

5.1 Design

The Transmitter consists of 1 top level wrapper module and 8 individual modules, as seen in the UTL diagram in Figure 12. What is different from the UTL diagram is that each module contains its own set of input and output queues. The top level module has Bluespec rules that pop data from the output queues of each module and push data into the input queues of the next module. This design does create a dead cycle in some cases, but this is not an issue because we meet the specified data rates. We decided to use this queue layout for easy integration with XBS and individual module testing.

In this section, we describe the microarchitectural design of each of the modules.

5.2 Controller

```
Input: Ctrl{Ctrl_type, rate, header length, message length, data length}
       Data - Framesize is 24 bits
Output: ScramblerCtrl{seed, data length},
        FormattingModuleCtrl{message length, data length},
EncoderCtrl{header_length, data_length},
        InterleaverCtrl{rate, length},
        MapperCtrl{rate, length, reset}
        ScramblerData{data:FRAMESIZE},
        EncoderData{hdr:FRAMESIZE}
```

The Controller is divided into three stages. In the first stage, it reads in the input request and starts sending out control messages to the individual modules. For modules that require two control messages per message (Interleaver and Mapper), the control signal sends another control message out in the second stage. In the third stage, the Controller reads in the data and passes it to the relevant module. The Controller cannot process a new request until it has finished processing the current request through all three stages. This makes keeping track of the state for each message simpler.

Scrambler Seed Generator The Scrambler seed generator is implemented within the Controller. At the beginning of each message, it provides a 7 bit initialization value for the Scrambler. For initial system level testing, we currently hard code 1 7-bit value to be output by the Seed Generator.

5.3 Scrambler

```
Input: Ctrl{7-bit seed, 13-bit length}
       Data - Framesize is 24 bits
Output: Scrambled Data - Framesize is 24 bits
```

Instead of scrambling each input bit one at a time, as shown in Figure 5, the Scrambler is designed so that it can process 24 bits in one pass. At the beginning of each message, the Scrambler first generates the entire 127-bit scramble sequence based on the seed. Each bit of the 127-bit sequence is generated one at a time, in the manner shown in Figure 5. Therefore the initialization stage takes 127 cycles. This overhead is acceptable because most messages will be greater than 127 bits. Thereafter the savings of being able to scramble 24 bits in one pass for the length of the message will more than make up for the initialization overhead. After the initialization stage, data is read in 24 bits per cycle and is XORed with the corresponding bits of the 127-bit sequence. The 127-bit sequence is stored in a circular shift register. After each cycle, the circular shift register is shifted left by 24 bits.

5.4 Formatting Module

```
Input: Ctrl{13-bit total msg length, 12-bit data length
       (not including 2 service octets)}
       Scrambled Data in 24 bit frames

Output: Scrambled Data - Framesize is 24 bits with the 6 tail bits replaced
        with zeros.
```

The Formatting Module replaces the scrambled 6 tail bits (shown in Figure 4) with unscrambled zeros. To accomplish this, the Formatting Module performs 2 tasks. One task is to pass the data received from the Scrambler to the Convolutional Encoder. Another task is identify the 6 tail bits in the data stream and replace these 6 bits with zeros.

At the beginning of each message the Formatting Module stores the message and data lengths (in octets) into two registers. The message length is the length of the entire message (not including preamble or header). The message length tells the Formatting Module how many octets it needs to pass from the Scrambler to the Convolutional Encoder for a particular message (task one). The data length is the length of the service and data portions of the message. The data length is used to identify the location of the 6 tail bits (task two). The Formatting Module decrements these

registers each time it reads a frame of data from the Scrambler and passes it to the Convolutional Encoder. When the data length register reaches zero, the Formatting Module replaces the next 6 bits (the tail bits) obtained from the Scrambler with zeros.

Since the data length (in octets) may not be evenly divisible by the frame size (24 bits), the Formatting Module also checks to see when the data length register reaches 1 and 2. Depending on the case, the Formatting Module will replace the correct 6 bits in the current frame with zeros.

5.5 Convolutional Encoder

```
Input: Ctrl{hdr_length, data_length}
       Data from Scrambler and Controller - Framesize is 24 bits
Output: Encoded Data from Scrambler (data) and Controller (header) -
  Framesize is 24 bits.
```

The Convolutional Encoder is split into two stages, as shown in Figure 13. In the first stage, the Encoder orders the data coming in from its two input queues. The Encoder must encode the header data coming from the Controller before encoding the data from the Scrambler. In the second stage, the Encoder encodes the ordered data.

Instead of only encoding one bit per cycle, as show in Figure 6, we designed the Convolutional Encoder to encode one frame of data every cycle, by unrolling the simple Encoder algorithm. Figure 14 shows the unrolled Encoder design. Every cycle, the Encoder first sets up the history bit vectors based on the inputs and the values in the history buffer. One history bit vector is required per input bit. The history buffer acts as the bit vector for the first input bit. The Encoder XOR's the data with the corresponding values in the history bit vectors to produce the outputs. The Encoder also updates the history buffer each cycle so that the history bit vectors can be computed in the next cycle.

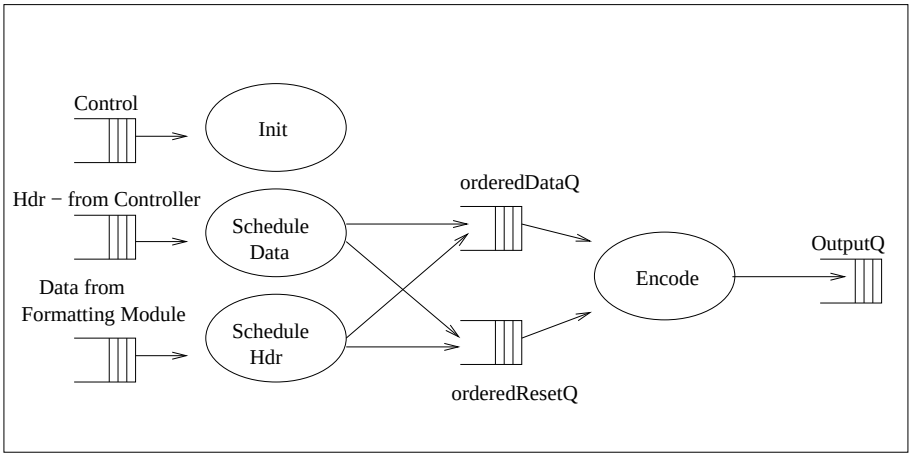


Figure 13: Microarchitecture Design of Convolutional Encoder.

5.6 Interleaver

```
Input: Ctrl{rate,length}
       Data - Framesize is 48 bits
Output: Interleaved Data - Framesize is 48 bits
```

The Interleaver rearranges the bits of the input data in some fixed pattern based on the rate. It is split into two stages, as shown in Figure 15. In the first stage, the Interleaver reads in each frame of data, rearranges them, and writes each bit into the corresponding index of a 192 bit buffer. Depending on the rate, the Interleaver either needs to operate on blocks of 1, 2, or 4 frames of data in this first stage, which changes the latency of the Interleaver. After

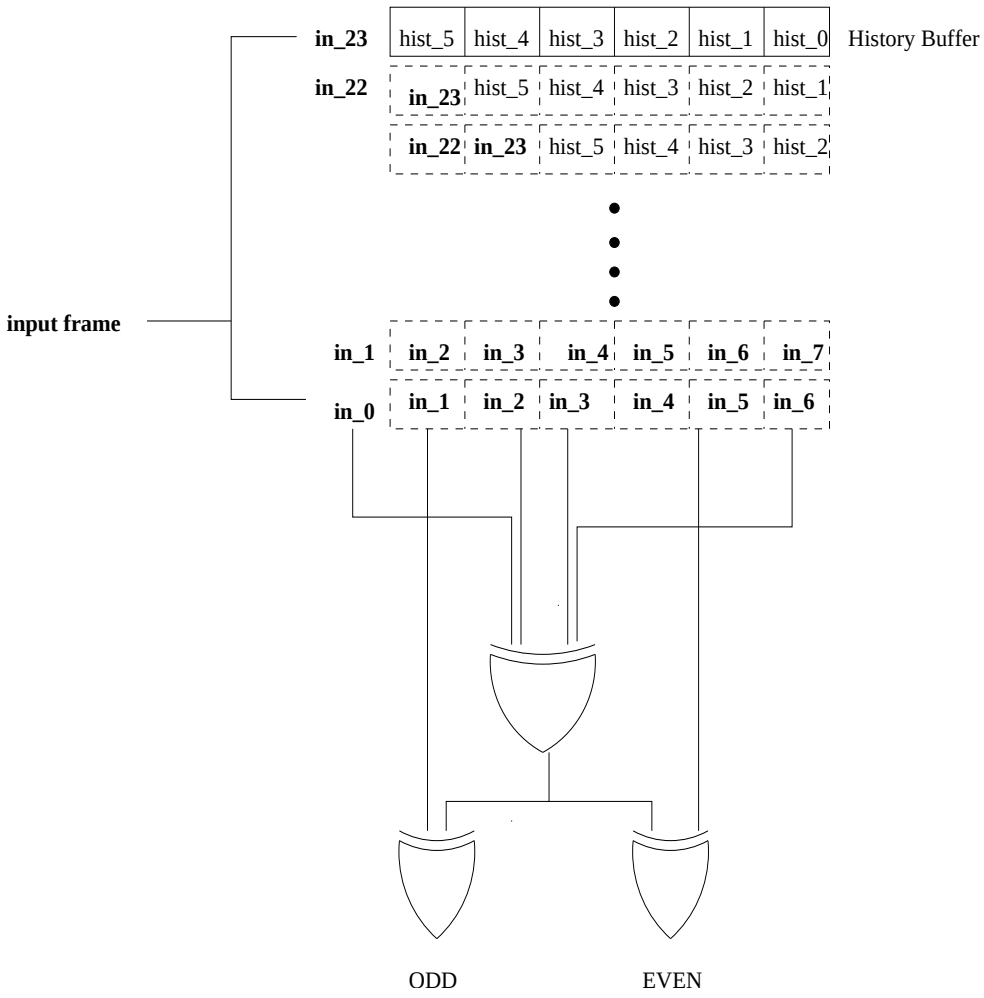


Figure 14: Unrolled Convolutional Encoder Design.

the required entries in the 192 bit buffer have been filled, the second stage of the Interleaver reads the data out of the buffer one frame at a time, and pushes them to the output queue. Currently the two stages cannot be overlapped.

5.7 Mapper

```
Input: Ctrl{rate,length,reset}
       Data - Framesize is 48 bits
Output: Frequency Domain OFDM Symbol - 64 complex pairs
```

Calculation of the complex pairs requires a multiplication of the K_{MOD} and the mapped complex pair value. In order to eliminate the need for multipliers in the Mapper, we decided to pre-compute and store all 16 possible products, eliminating the need for 2 multipliers per parallel complex pair multiplication. In our design, we aimed to compute as many as 48 complex pairs in parallel (for the 6MB/s rate) and, therefore, by storing the products, not only did we save in area, but we also decreased the critical path in the Mapper.

At the beginning of each message, the pilot polarity vector index needs to be reset, which is indicated by the reset bit.

As shown in Figure 16, the Mapper is split into two stages similar to the Interleaver. The first stage may iterate multiple times (depending on the data rate) before all the complex pairs have been mapped and pilots inserted. A 64

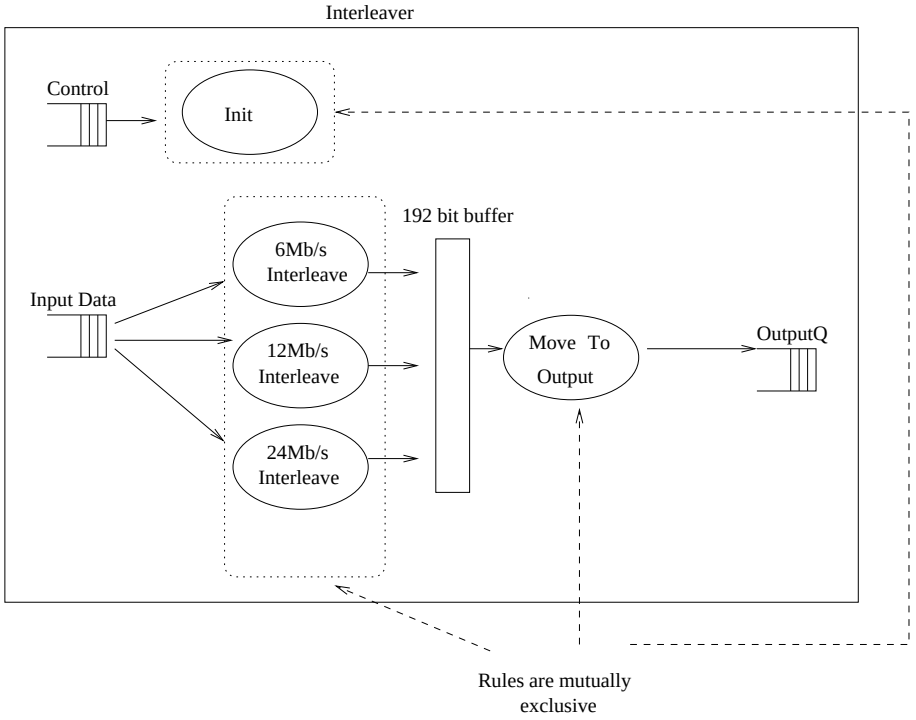


Figure 15: Microarchitectural Design of Interleaver.

complex pair register file is used to temporarily store the computed complex pairs until mapping is complete. Then, in the second stage, the complex pairs are read out of the register file to the output queue.

5.8 IFFT

```
Input:  Data - 64 complex pairs
Output: OFDM Symbol - 64 complex pairs
```

For initial design purposes, the IFFT is completely unrolled. It accepts 64 complex pairs of data, which it then sends through three stages of data. Each stage consists of 16 radix-4 nodes (see Figure 17). The radix-4 node contains all of the computational complexity and has three stages: a twiddle multiplication and two combinational stages (see Figure 18). Each node requires 16 multiplication units and 24 adds. Because signed multiplication is not supported, significant logic is required to convert signed values and support a fixed-point number representation. After completing the IFFT calculation, a reorder stage aligns the data correctly before writing it to the output queue.

Due to anomalies in Bluespec during implementation, the format of the data changes within the IFFT block. While the Mapper passes a vector of complex pairs, the IFFT implementation converts this to a structure for the calculation and then reconverts to a vector for the Cyclic Extender.

5.9 Cyclic Extender

```
Input:  Data - 64 complex pairs
Output: Guarded OFDM Symbol - 81 complex pairs
```

Cyclic Extension involves prepending the data with the last 16 complex pairs of itself. The first complex pair of the data is appended at the end. This is shown in Figure 19.

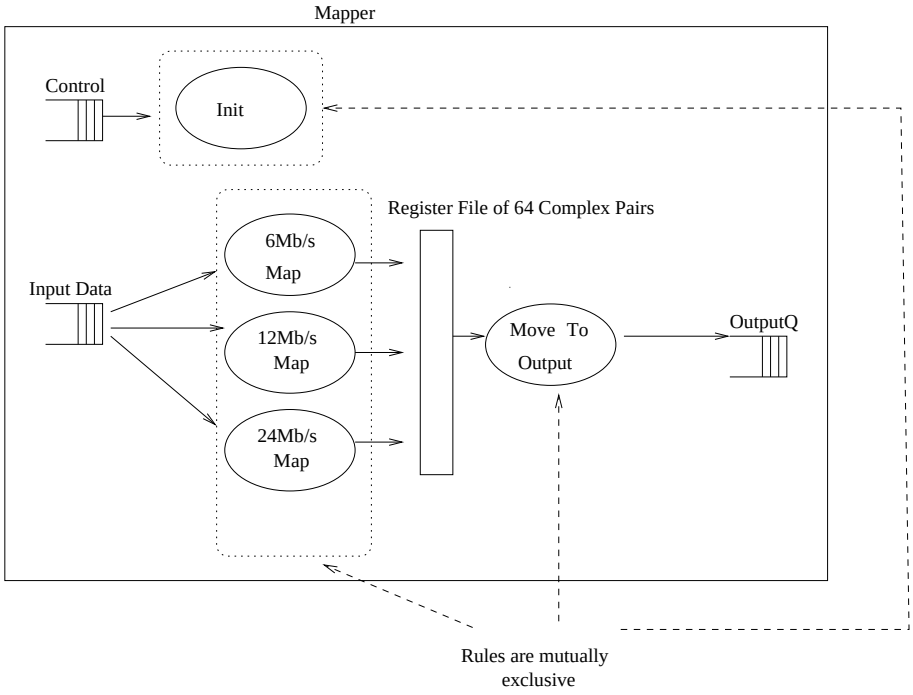


Figure 16: Microarchitectural Design of Mapper.

5.10 Design Analysis

After running some initial system tests, we decided that the 127-cycle initialization of the Scrambler, at the beginning of each message, needed to be redesigned. For a series of medium to long messages, the 127 cycle overhead has little effect on the data rate. However, for the exceptional case of a series of minimum length messages, the 127 cycle initialization stage severely impacts the data rate. Although we think that a series of minimum length packets will not occur very often, we still want our Transmitter design to meet the data rate for all possible inputs.

5.11 Synthesis Results

After pushing our design through synthesis using the Synopsys Design Compiler, we obtained the following estimation of our design's area:

Module	**Area (μm^2)**
Input Queues	7,428
Controller	29,621
Scrambler	25,970
Formatting Module	12,403
Convolutional Encoder	38,017
Interleaver	51,270
Mapper	404,401
IFFT	29,023,736
Cyclic Extend	598,689
Output Queue	410,469
Miscellaneous Logic	409
Total	30,602,413

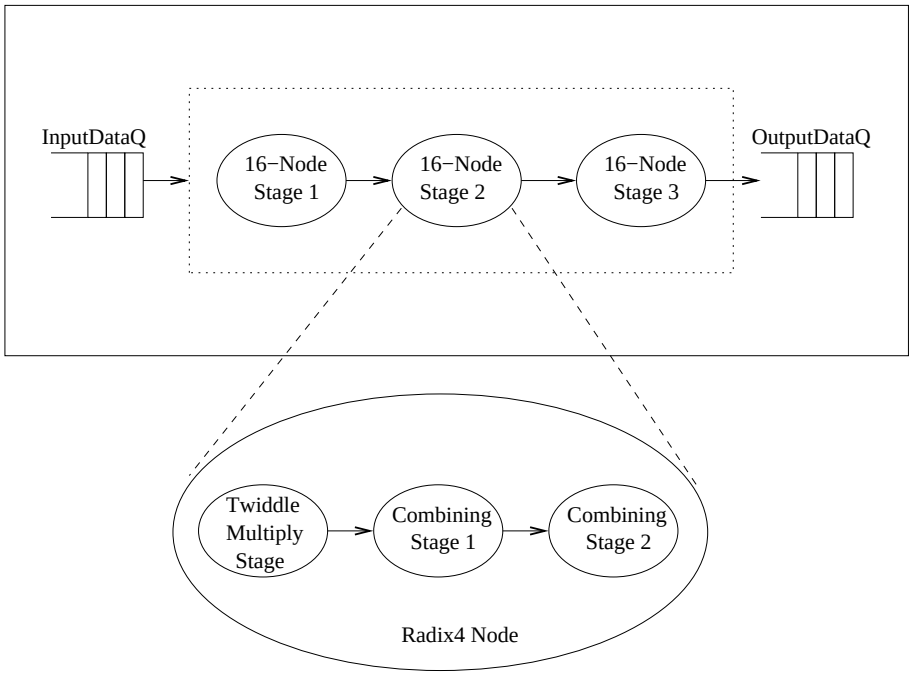


Figure 17: Microarchitectural Design of IFFT.

The critical path delay in our design was determined to be 64.36 nanoseconds through the IFFT unit. This means that our design would be able to run at 15.54 MHz. Our initial synthesis results clearly indicate that the IFFT, which accounts for 94.8% of our total area and sets the critical path of our circuit, must be a main focus of design exploration.

6 First-Pass Design Exploration

6.1 Design Modifications

In the first pass of our design exploration, we decreased the initialization stage of the Scrambler from 127 cycles to 1 cycle. This optimization was made at the cost of generality, but since the 802.11a specification did not define the range of supported Scrambler seeds, we decided that this loss of seed generality would not limit the Transmitter design.

Initially, we designed the system so that it could operate on a large range of 7 bit Scrambler seeds. But in order to support the use of this large range of seeds, we needed the Scrambler to compute, in real time, the 127 bit initialization sequence. This computation took 127 cycles.

In our revised design of the Scrambler, we decided to support 16 fixed seeds. We pre-computed the 127 bit sequences for each of these seeds and stored them in a lookup table within the Scrambler. Now the seed generator pseudo-randomly provides an index (from 0 to 15) which the Scrambler uses to select the corresponding 127 bit sequence. The use of the lookup table reduces the Scrambler initialization stage to 1 cycle.

The Scrambler seed generator uses the input length field to generate an index into the Scrambler lookup table, as shown below.

```
scrambler_sequence_index = {length[4], length[2], length[10], length[0]}
```

Since the length generally changes from message to message, the Scrambler will be initialized differently each time.

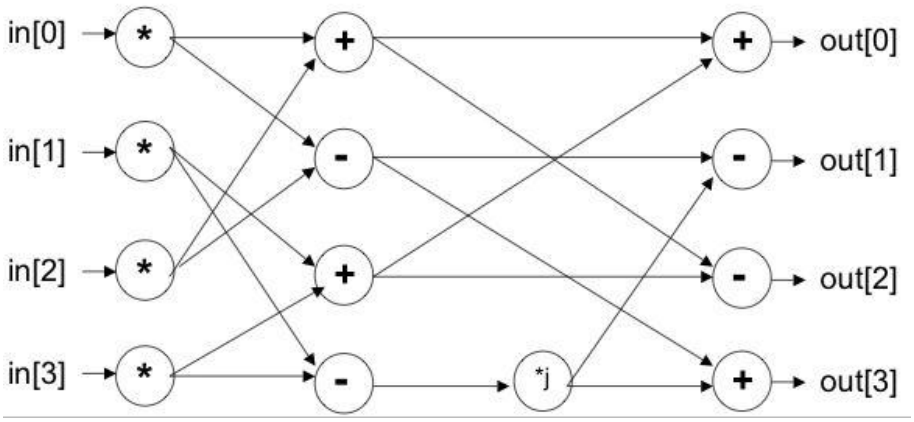


Figure 18: Radix 4 Module.

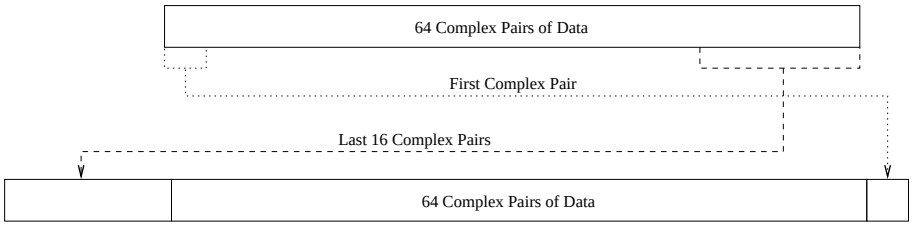


Figure 19: Cyclic Extension.

6.2 Output Analysis

In this section we analyze the throughput of the Transmitter, at different data rates, to identify the main bottleneck of the system. For the bottleneck modules, we propose some design modifications to achieve full throughput.

System-Level Analysis Using the lowest data rate of 6Mb/s and starting from an empty pipeline, it takes 24 cycles to get out 1 header and 1 data OFDM symbol. Subsequent data symbols come out once every other cycle, as seen in Figure 20. Therefore we expect the highest throughput we can achieve with the current design is 0.5. The simulation output confirms this analysis: one long message sent at 6Mb/s, gives an output rate of 0.49.

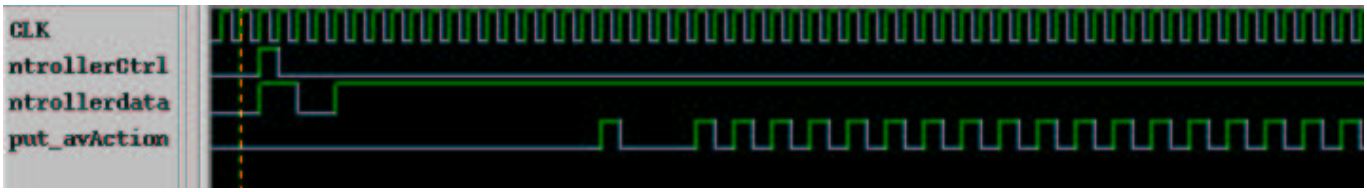


Figure 20: Waveform Output of 1 long message sent at 6Mb/s.

Using the 12 Mb/s data rate, the output rate is decreased by a factor of 2. Using the highest data rate of 24Mb/s, the output rate is decreased by a factor of 4, resulting in a throughput of 0.12.

Unit-Level Analysis Further unit-level analysis identified the bottlenecks to be the Interleaver and Mapper. The throughput of each module, measured in terms of output frames per cycle, is shown in Figure 21.

Module	6Mb/s	12Mb/s	24Mb/s
Scrambler	0.99	0.99	0.99
Formatter	0.99	0.99	0.99
Conv. Encoder	0.99	0.99	0.99
Interleaver	0.49	0.49	0.49
Mapper	0.49	0.33	0.199
IFFT	0.99	0.99	0.99
Cyclic Extender	0.99	0.99	0.99

Figure 21: Unit-Level throughput (output frames/cycle) of key modules at different data rates.

Multiple Messages When a series of short messages are sent, the startup overhead before the header of each message is 2 cycles. The time between the header and the data portion of the message varies depending on the data rate.

FIFO Queue Sizing We experimented with larger queue sizes in between each module, but observed no improvement in throughput. Therefore the bottleneck is not in the queues but in the Interleaver.

6.2.1 Design Modifications for Bottleneck Modules

In order to achieve full throughput, of 1 output frame per cycle, for the Interleaver and Mapper, the rules for different stages in the modules need to fire simultaneously. Basically, these modules need to become fully pipelined.

As described in Section 5.6, the Interleaver does not overlap computation with the outputting of data. Because it takes multiple cycles to output all the data for the 12Mb/s and 24Mb/s rates, overlapping of the computation and output stages can be tricky. Data that has not yet been output must not be overwritten. As shown in Figure 15 the interleaving and output rules share the 192 bit buffer. Mutual exclusion of the stages was necessary to synchronize the the rules over this shared resource. In order to overlap the interleaving and output stages, we need to add an additional 192 bit buffer. This way, while one stage is reading/writing one buffer, the other stage can be reading/writing the other buffer. By alternating between the buffers, full throughput can be achieved.

As seen in Figure 21, the Mapper's throughput decreases as data rate is increased. This is expected because as the data rate increases, the ratio of input frames to one output frame increases. At 6Mb/s, one input frame results in one output frame. At 12Mb/s, 2 input frames are needed to create one output frame. At 24Mb/s, 4 input frames are needed per output frame. Based on this analysis, the ideal throughputs are 1, 0.5, and 0.25 for the 6Mb/s, 12Mb/s and 24Mb/s data rates respectively. The reason why the Mapper's throughput is lower than that of the ideal is because the Bluespec rules for computation and outputting are mutually exclusive. Therefore, when the Mapper is outputting data (1 cycle), it is not computing. This added cycle for outputting data decreased the Mapper's throughput from the ideal.

In order to improve the Mapper's throughput, the rule for outputting data and the rule for computation need to be able to fire simultaneously. Therefore pipelining is the solution. Pipelining the Mapper would not be difficult. It just requires some extra registers to keep the state of each pipeline stage.

If we include the proposed modifications for the Interleaver and Mapper into our system, we will be able to achieve the system throughput for each data rate, as shown in Figure 22.

Module	6Mb/s	12Mb/s	24Mb/s
Transmitter	0.99	0.49	0.24

Figure 22: Projected system throughput (output frames/cycle) after proposed modifications.

At the full system throughput, our Transmitter will be able to process one 24 bit input frame per cycle. Therefore, when transmitting at 6Mb/s, 12Mb/s, and 24Mb/s, the minimum clock frequencies required are 250 KHz (4 us clock period), 500 KHz (2 us clock period), and 1MHz (1 us clock period) respectively. Therefore, we can reduce our clock frequency from 15.54 MHz to 1MHz and lower V_{DD} accordingly to achieve a lower power Transmitter design.

6.2.2 Current Design Point Analysis

Without incorporating the optimizations to the bottleneck modules, our Transmitter is able to process one 24 bit input frame every 2 cycles. Therefore the minimum clock frequencies required are 500 KHz, 1 MHz, and 2MHz for the 6, 12, and 24 Mb/s data rates respectively. Since our current clock frequency of 15.54 MHz is larger than the minimum clock frequency required, we clearly meet our target data rates. Therefore with the unmodified design, we can reduce our clock frequency from 15.54 MHz to 2MHz and lower V_{DD} to reduce power.

6.2.3 Design Decisions

Our two main design goals are minimizing area and power. We feel that the current Transmitter size of 30mm^2 is much too large. Since the fully unrolled IFFT dominates the Transmitter area, our first priority is to minimize the area of the IFFT. We decided to hold off on implementing the proposed changes because we believe that the IFFT will become the main performance bottleneck of the system when we reduce its area. When the IFFT becomes the main bottleneck, any optimizations to the other modules will not benefit performance.

6.3 Synthesis Results

Since the IFFT dominates the area and clock period in the Transmitter, the addition of a lookup table in the Scrambler did not change the synthesis results much.

7 Second-Pass Design Exploration

7.1 Design Modifications

The initial results for the IFFT clearly demonstrate that a significant design change is required. The area for the IFFT is 29.1mm^2, 94.8% of the entire system, and the clock cycle is dominated by the IFFT due to the lack of pipelining. In order to reduce both area and the critical path, we fold the calculation to reduce the number of computational stages from three down to one, consisting of 16 nodes (see Figure 23).

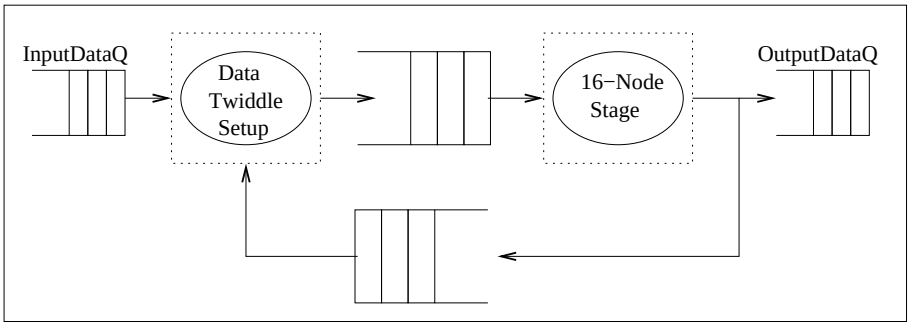


Figure 23: Microarchitectural State of IFFT Folded to 1 Stage.

Additional logic prior to the stage reorders the data and computes the twiddle factors. We expect the new IFFT design to reduce the system area by a factor of three, although the additional logic will increase the area by some amount. The cycle time and the throughput of the system should also decrease, since the single stage will be time multiplexed (over multiple cycles) to perform the entire calculation.

7.2 Performance Analysis

The optimized IFFT takes 7 cycles in steady state to compute one output (3 to setup the data and twiddles, 3 to compute the different IFFT stages, and 1 to output the data). Therefore the highest throughput attainable by the IFFT alone is 0.143.

Figure 24 shows the throughput of the system at each data rate. For the 6Mb/s and 12Mb/s data rates, the IFFT is the bottleneck with one output packet produced every 7 cycles. For the 24Mb/s data rate, one output is produced every 8 cycles because the Interleaver processes one packet every 2 cycles, requiring 8 cycles to produce the needed 4 input packets to calculate one output at 24Mb/s. Therefore, the minimum clock frequency required to meet the 6Mb/s and 12Mb/s data rates is 1.75 MHz, and the minimum frequency required to meet the 24Mb/s data rate is 2MHz. Place and route results show that our system runs at a frequency of 33.33 MHz. Therefore our Transmitter clearly meets all the required data rates. For a lower power design, we could reduce the system clock frequency to 2MHz and lower V_{DD}.

Module	6Mb/s	12Mb/s	24Mb/s
Transmitter	0.142	0.142	0.123

Figure 24: System Throughput (output frames/cycle) after Design Exploration 2.

We debated whether we should implement the modifications for the Interleaver and Mapper proposed in Section 6.2.1. Implementing these changes would enable us to decrease the minimum required clock frequency from 2MHz to 1.75MHz, and we can get further power savings. However, implementing these changes will increase the area of the Transmitter. Increased area results in increased capacitance, which would reduce the amount of power savings obtained by decreasing the frequency to 1.75MHz. Further analysis using power-delay curves is needed before we can decide if any changes to our system are worthwhile.

7.3 Synthesis Results

Implementing this change decreases the area of the IFFT by 49% and decreases the cycle time by 15% as can be seen in the table. The size of one calculation stage increases due to the separation of the fixed twiddle numbers from the computation, reducing the optimizations possible.

Module	Area (μm^2)
AN2D2	25
AO21D1	10
BUFFD12	144
BUFFD16	244
CKND2	721
CKND16	2,658
CKNXD0	23
CKNXD16	246
FIFO2_00000800_0	400,586
FIFO2_00000800_1	264,230
FIFO2_00001004_0	511,243
FIFO2_00001004_1	285,250
INVD0	12,516
INVD1	1,089
INVD2	10,194
INVD16	28
MUX2D1	556
MUX2D2	168
MUX2ND0	23,716
ND3D2	11
NR2D2	8
OR4D1	11
XOR2D1	13
module_ifft_data_setup	53,661
module_ifft_stage	12,594,934
module_ifft_twiddle_setup	2,549
Total	14,164,846
Cycle Time	9.88 ns

7.4 Tool Flow Modifications

To reduce area even further, we next decided to exploit the tool capabilities. We modified our scripts to run at high area effort, changing:

`compile -map_effort medium -area_effort` **none** `-boundary_optimization`

to:

`compile -map_effort medium -area_effort` **high** `-boundary_optimization`

This surprisingly resulted in significant area reduction of 47% as well as a slight cycle time reduction, shown in the following table.

Module	Area (μm^2)
AN2D1	20
AO21D0	8
BUFFD0	96
CKND0	3
CKND16	98
CKNXD0	44
FIFO2_00000800_0	327,867
FIFO2_00000800_1	175,364
FIFO2_00001004_0	360,445
FIFO2_00001004_1	200,296
INVD0	11,888
INVD1	6,154
INVD2	5
INVD8	122
MUX2D1	3,761
MUX2ND0	21,042
ND3D1	6
NR2D1	5
OR4D1	11
XOR2D1	13
module_ifft_data_setup	30,923
module_ifft_stage	5,473,274
module_ifft_twiddle_setup	351
Total	6,611,707
Cycle Time	9.52 ns

7.5 Place and Route

After place and route, the current area of the Transmitter is 6.34mm^2, and the cycle time is 30.5ns.

8 Final Modifications

Resizing FIFO Queue We noticed that further area savings can be obtained if we decrease the sizes of the input and output queues for the Mapper, IFFT, and Cyclic Extender. Although each queue only had 2 elements, the data width is very large resulting in a large area per element. We decided to resize the output FIFO of the Mapper, input and output FIFOs of the IFFT, input and output FIFOs of the Cyclic Extender, and the output FIFO of the top level Transmitter module to 1 element. This did not reduce the throughput of the system, so our minimum clock frequency remains at 2 MHz.

Optimized IFFT In performing the design exploration of the IFFT, two rules controlled the two different steps required, one to setup the data and the other to calculate the IFFT stage. This resulted in 6 stages to perform the calculation as well as extra queues for passing data between the rules. The hope was that by separating these stages the design could be pipelined, but that would require additional hardware. Given that the design exceeds the data rate necessary, this unnecessarily increases the area. These two rules were then combined (see Figure 25), resulting in the removal of one queue and a latency decrease to 4 cycles. The synthesis results are shown in the table below.

Module	Area (μm^2)
AN2D1	7
BUFFD0	1339
CKND2D0	10
CKND2D1	10
CKND16	49
CKNXD0	57
FIFO2_00000800_0	184,716
FIFO2_00000800_1	74,303
FIFO2_00000802	181,622
INVD0	61
INVD1	68
MUX2D1	27552
MUX2ND0	214
ND2D1	31
ND2D2	8
ND3D1	7
NR2D0	5
module_ifft_data_setup	44,233
module_ifft_stage	3,967,284
module_ifft_twiddle_setup	1388
Total	4,482,965
Cycle Time	27.15 ns

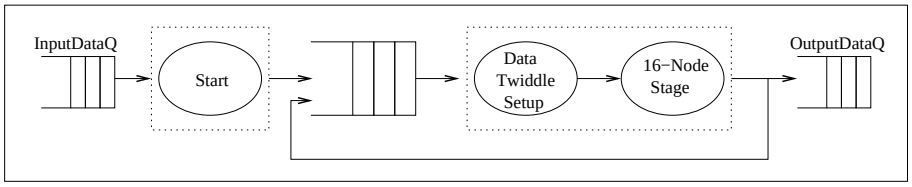


Figure 25: Microarchitectural State of IFFT with Setup and Calculation Combined.

8.1 Performance Evaluation

Before final modifications, the IFFT was the bottleneck for the 6Mb/s and 12 Mb/s data rates. After optimizations, the IFFT decreased its latency from 7 cycles to 4 cycles. Therefore the system throughput increased for these two data rates, as shown in Figure 26. The Interleaver and Mapper remain the bottleneck for the 24Mb/s data rate. Therefore, the throughput at that data rate, as well as the minimum clock frequency of the system, did not change.

Module	6Mb/s	12Mb/s	24Mb/s
Transmitter	0.248	0.248	0.123

Figure 26: System Throughput (output frames/cycle) after Final Modifications.

8.2 Place And Route

Our final Transmitter design is 5.27mm^2, with a cycle time of 32.9ns. This results in a clock freq of 30.4MHz. The physical layout of the different modules on our chip is shown in Figure 27.

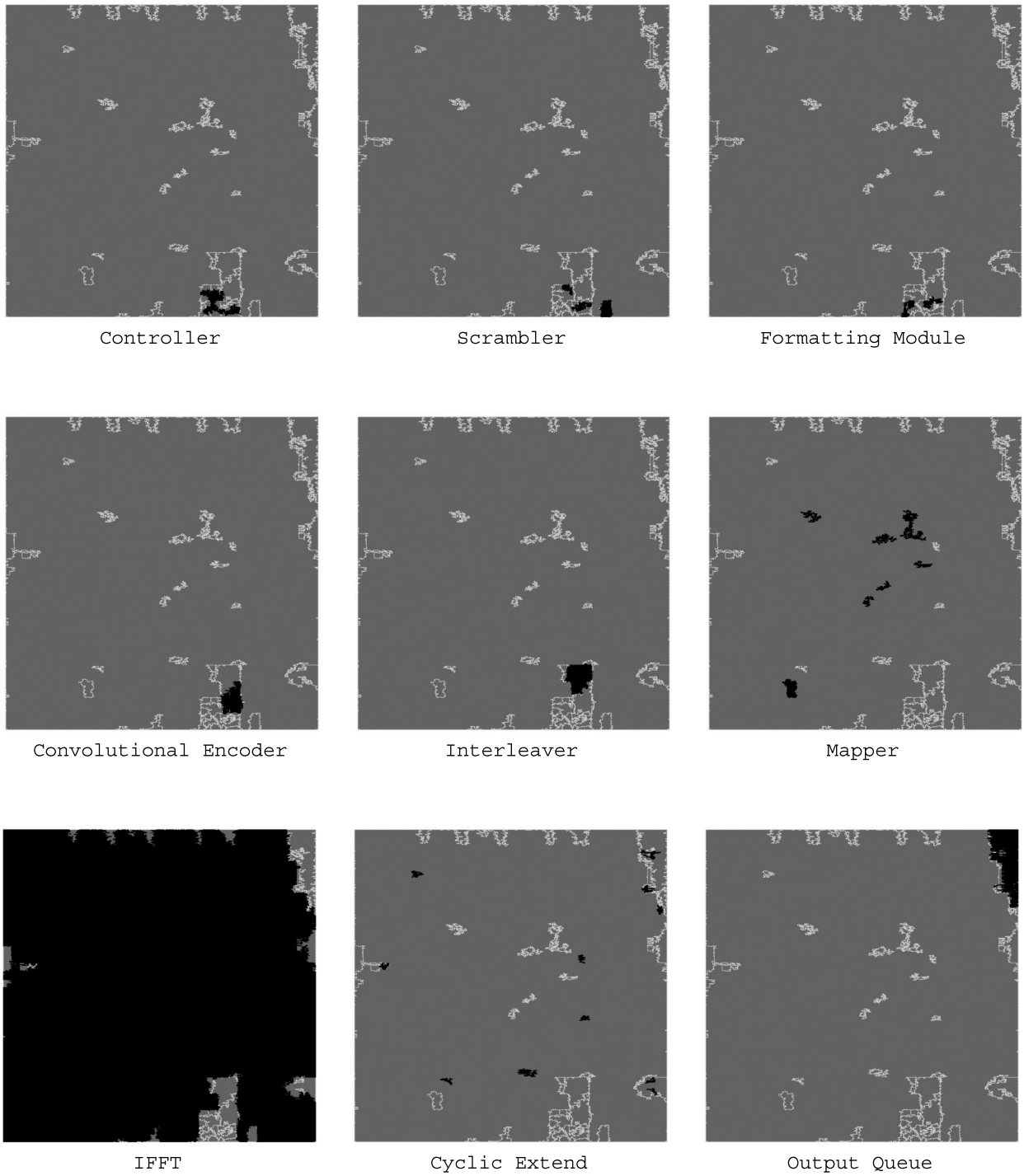


Figure 27: Physical module layout. The black in each diagram represents the area occupied by that module.

8.3 Future Work

Scrambler The Scrambler can be further optimized to support any 7-bit seed and decrease area, while maintaining 1 cycle initialization. This can be achieved by unrolling the Scrambler initialization algorithm, as shown in Figure 28. During the 1 cycle initialization stage, the Scrambler stores the seed in a register. During each compute stage, the Scrambler XORs the corresponding bits of the seed to generate the next 24 bits of the Scrambler sequence. These 24 bits are then XORed with the corresponding input bits to produce the output. At the end of each compute stage, the Scrambler sets up the state of the seed register for the next cycle of computation.

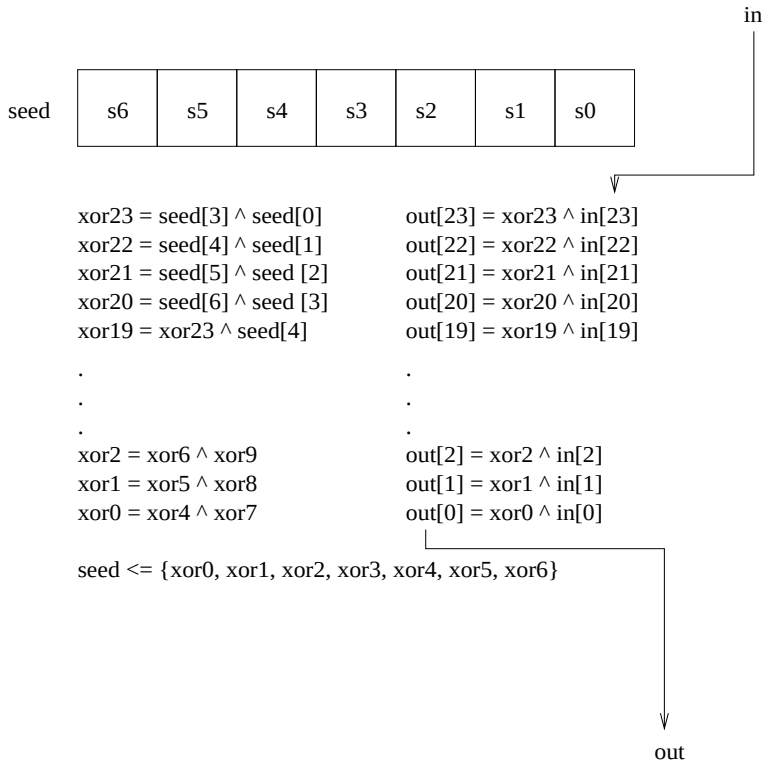


Figure 28: Optimized Scrambler.

Input and Output FIFO Queue Placement We place the input and output queues within each module in order to decouple and simplify the interface between the modules. This also enables the modules to be easily integrated into the XBS test structure for unit testing. A consequence of this design, however, is that data is double buffered, which increases area. Therefore to achieve an even lower area, we can eliminate the output queues within each module allowing data to be buffered only once, in the input queues of each module. For modules with wide FIFOs, such as the Mapper, IFFT, and Cyclic Extend, this would result in significant area savings. However, this new design would change the simple interfaces of each module. The Bluespec methods for each module would need explicit predicate conditions, whereas in the old design, the Bluespec methods relied on implicit FIFO full/empty predicates.

IFFT We could achieve addtional area savings by folding the IFFT calculation even further. Instead of just folding the stages, we could fold the nodes within each stage, reducing the calculation to one radix4 node. This would increase the latency of the IFFT to 49 cycles in steady state, but we have enough excess throughput within the system to account for the extra delay.

9 Conclusion

Our implementation of the 802.11a Physical Layer successfully meets the required data rates of 6Mb/s, 12Mb/s and 24Mb/s. Although our initial area was large, through design exploration we reduced it by 83% to 5.27mm^2. Our final design runs at 30.4MHz, which exceeds the 2MHz frequency required to meet the data rates. This allows us to reduce V_{DD} and operate at a much slower frequency to reduce power.

References

[1] IEEE. *IEEE standard 802.11a supplement. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. High-speed Physical Layer in the 5 GHz Band*, 1999.