From WiFi to WiMAX: Techniques for High-Level IP Reuse across Different OFDM Protocols

Man Cheuk Ng, Muralidaran Vijayaraghavan, Nirav Dave, Arvind
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Email: {mcn02,vmurali,ndave,arvind}@csail.mit.edu

Gopal Raghavan, Jamey Hicks
Nokia Research Center Cambridge
Nokia Corporation
Email: {gopal.raghavan,jamey.hicks}@nokia.com

Abstract

Orthogonal Frequency-Division Multiplexing (OFDM) has become the preferred modulation scheme for both broadband and high bitrate digital wireless protocols because of its spectral efficiency and robustness against multipath interference. Although the components and overall structure of different OFDM protocols are functionally similar, the characteristics of the environment for which a wireless protocol is designed often result in different instantiations of various components. In this paper, we describe how we can instantiate baseband processoring of two different wireless protocols, namely 802.11a and 802.16 in Bluespec from a highly parameterized code for a generic OFDM protocol. Our approach results in highly reusable IP blocks that can dramatically reduce the time-to-market of new OFDM protocols. One advantage of Bluespec over SystemC is that our code is synthesizable into high quality hardware, which we demonstrate via synthesis results. Using a Viterbi decoder we also demonstrate how parameterization can be used to study area-performance tradeoff in the implementation of a module. Furthermore, parameterized modules and modular composition can facilitate implementation-grounded algorithmic exploration in the design of new protocols.

1 Introduction

In this paper, we demonstrate that it is possible to generate efficient hardware for two different wireless protocols, namely 802.11a [8] and 802.16 [9], from the same code base written in Bluespec SystemVerilog (BSV). The following two features are essential for such designs: 1) a polymorphic type systems that permits highly parameterized codes and 2) the ability to compose independently created modules with predictable functionality and performance. The latter capability permits the refinement of individual modules to meet the performance objectives and exploration of area-performance tradeoffs without exacerbating the verification problem.

Although Verilog and VHDL support parameterized designs, the standard practice is to do such parameterization using perl scripts that generate specific RTL versions. It may be possible to do highly parameterized designs in SystemC, though we have not seen it reported in the literature. SystemC, unlike Bluespec, only provides limited capability to synthesize parameterized designs into efficient hardware. Parameterization in Bluespec, on the other hand, can be used freely because it does not cause any extra logic gates to be generated; the compiler removes all the static parameterization during the "static elaboration" phase.

This work has grown out of studying the area-power tradeoffs in the design of 802.11a transmitter, which was reported earlier [4]. After we implemented the 802.11a receiver, it was suggested by Nokia that we should focus on the design problem of multi-radios. Modern cell phones usually contain multiple radios, typically three, but sometimes as many as seven, all of which are implemented as special hardware blocks. It would be beneficial to share not only the design cost of such radios but even the actual hardware blocks among those radios that do not operate concurrently. Based on these discussions we focused on "OFDM based" protocols and built a set of highly reusable IP blocks, which can be instantiated with different parameters for different protocols. Though the development of such modules requires domain expertise, we think the use of such modules in designing and implementing new protocols requires considerably less knowledge. We hope this paper demonstrates how the cost of hardware design, i.e., ASIC blocks, and the time-to-market can be reduced dramatically by reusable parameterized IP.

This paper requires no prior knowledge of wireless protocols – we explain the basic concepts as needed in the next two sections. The level of parameterization in the de-

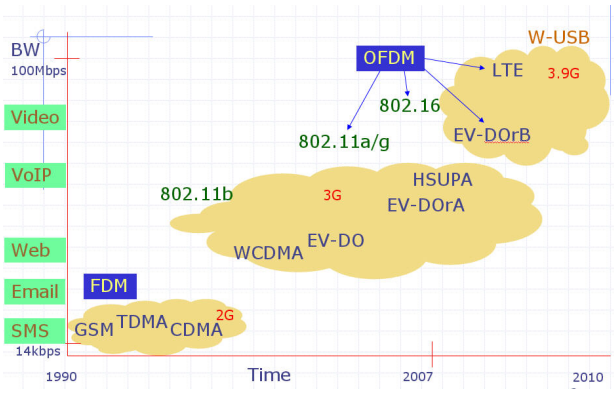

Figure 1. Digital wireless protocol evolution

sign presented involves some sophisticated programming. However, only a rudimentary knowledge of Bluespec is assumed, which an uninitiated reader can get by reading, for example, the 802.11a paper [4]. We will explain advanced programming concepts as needed.

Organization: We begin by describing the importance of OFDM-based protocols in the digital wireless landscape (Section 2), followed by description of a generic OFDM baseband transceiver (Section 3). We then explain how the various blocks of 802.11a and 802.16 protocols are instances of these generic blocks. Next, we show the overall structure of the OFDM transceiver in Bluespec and discuss the issues related to parameterization (Section 4). We further discuss the parameterization of modules, using the example of Viterbi decoder, for architectural exploration to meet area, power and performance goals (Section 5). Finally, we present synthesis results (Section 6), related work (Section 7) and our conclusions (Section 8).

2 Digital Wireless Landscape

Since the early 1990's, there has been a rapid evolution in digital wireless protocols to enable higher data rates, improve bandwidth efficiency and to offer services to more users. There is a dramatic shift from purely voice based services to high bitrate data services to support web browsing, VoIP and high definition video. Another interesting development is the convergence of services offered via broadband wireless access like WiMAX and cellular networks like 3.9G. The underlying technology that enables this high data rate in non-line-of-sight environment is a modulation scheme known as Orthogonal Frequency Division Multiplexing (OFDM) [2]. OFDM has been around for several decades, but now its robustness to multipath interference has been proven in practice by widespread deployment of 802.11a/g and ADSL.

Some of the challenges in wireless communication are interference from other RF sources, self-interference due to multipath transmissions, and frequency dependent signal loss (fading). In the narrowband environment, simple modulation schemes, such as frequency modulation (FM), amplitude modulation (AM), and phase modulation (PM), protect against interferences and signal loss. However, such simple schemes do not offer high data transmission capacity. With higher channel bandwidth and greater rates of mobility, inter and intra-symbol interference for a single carrier becomes significant. Traditional FDM (Frequency Domain Modulation) techniques have larger guard bands between subcarriers and waste bandwidth. OFDM offers an elegant solution by spreading data across many closely-packed and overlapping narrowband subcarriers. In OFDM, the spacing between sub-carriers is carefully designed such that they are orthogonal to each other, meaning the product of two carriers with different frequencies is zero if sampled at frequency determined by sub-carrier spacing. This results in zero cross-talk between sub-carriers. In short, to get high data rates we need higher bandwidth. To avoid self-symbol interference and inter symbol interference the bandwidth is divided into multiple narrowbands that carry lower data rates and the sub-carriers are placed orthogonal to each other. As a result, OFDM provides high spectral efficiency and is robust against multipath interferences.

3 Generic OFDM Baseband Transceiver

The structure of a generic OFDM baseband transceiver is shown in Figure 2. Our focus is on a parameterized implementation of the baseband processing blocks so that we can instantiate different implementations of the blocks to support multiple radio protocols. We will not discuss the implementations of MAC (Media Access Controller) and analog to digital (A/D) or digital to analog (D/A) converters because they entail entirely different sets of issues. In wireless communications, the fundamental unit of communication is a *symbol*, which encodes one or more data bits. An OFDM symbol in turn is defined as a collection of digital samples, which are usually represented as complex numbers. The size of the symbol is determined by the number of subcarriers used in the system. In general, a fraction of subcarriers, known as data subcarriers, are used for data transmission and the remaining subcarriers are used for pilots and guard bands. Pilots provide information which is used by the receiver to better estimate the frequency fading in transmitted symbol thereby increasing the chance of a successful data reception. Guard bands are normally added to both sides of the frequency spectrum to avoid interference with other carriers.

The data received from the medium access control layer flows through various processing steps, converting it to an OFDM symbol which is transmitted over the air. Similarly, on the receiver side, OFDM symbols are formed from the signals received through the A/D converter and processed through various stages. Fnially, the resulting received data is sent to the MAC. In the rest of this section, we briefly describe each of the blocks in these transceiver pipelines, and discuss parameters for each block that are needed to describe two specific OFDM-based protocols, namely 802.11a and 802.16. In each case, where it is appropriate, we also point out the performance required to meet the standard. The table in Figure 3 summarizes the parameters used by various blocks in both protocols. It should be read in conjunction with the following description of the transceiver pipelines.

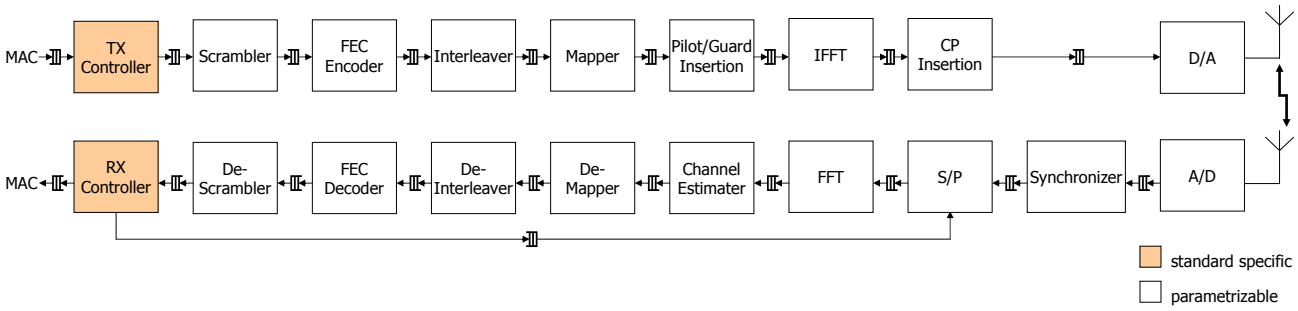

Figure 2. Generic OFDM baseband transceiver blocks

3.1 Transmitter Pipeline

TX Controller: Receives information from the MAC and generates the control and data for all the subsequent blocks.
Scrambler: Randomizes the data bit stream to remove repeated patterns, like long sequences of zeros and ones. This enables better results for Forward Error Correction (FEC). A scrambler is usually implemented with linear feedback shift registers (LFSR). An LFSR has two algorithmic settings: the size of the shift register and the linear function, e.g., $x^7 + x^4 + 1$, for generating the feedback.
FEC Encoder: Encodes data and adds redundancy to the bit stream to enable the receiver to detect and correct errors.

Both protocols use convolutional coding, however, 802.16 also requires Reed-Solomon encoding before the data is passed to the convolutional encoder.

Both protocols also use a technique known as *puncturing* to reduce the transmitted number of bits. For higher transmission rates in low-noise channels, the encoded data is *punctured* by deleting bits before transmission and replacing them with fixed values on reception. This reduces the number of bits to be carried over the channel and depends on the decoder to correctly reconstruct the data.
Interleaver: Rearranges blocks of data bits by mapping adjacent coded bits into non-adjacent subcarriers to protect against burst errors. The block size is the same as the number of bits that are coded in a single OFDM symbol. The symbol size itself is determined by the number of data subcarriers and the modulation scheme employed.
Mapper: Passes interleaved data through a serial to parallel converter, mapping groups of bits to separate carriers, and encoding each bit group by frequency, amplitude, and phase. The output of the Mapper contains only the values of data subcarriers for an OFDM symbol.
Pilot/Guard Insertion: Adds the values for pilot and guard subcarriers. The subcarrier indices are protocol-specific. Both protocols use scramblers to generate values for the pilots and use null values for the guard subcarriers.
IFFT: Converts symbols from the frequency domain to the time domain. The size of the IFFT is determined by the number of subcarriers used by the given OFDM protocol.
CP Insertion: Copies some samples from the end of the symbol to the front to add some redundancy to the symbols. These duplicated samples are known as a cyclic prefix (CP). The purpose of the cyclic prefix is to avoid Inter-Symbol Interference (ISI) caused by multipath propagation.

This block also adds a preamble before the first transmitted symbol. A preamble is a collection of predefined complex numbers known by the receiver so that it can detect the start of new transmission. The preambles for the two protocols have similar structure.

After CP insertion, the symbol are converted into analog signals by D/A converter and transmitted through the air.

3.2 Receiver Pipeline

The functionality of the blocks in the receiver is roughly the reverse of the functionality of their corresponding blocks in the transmitter. However, since the receiver has to recover data from a degraded signal, some receiver blocks have to do more processing and consequently require more implementation effort. When the antenna detects the signal, it amplifies the signal and passes it to the A/D converter to generate baseband digital samples.
Synchronizer: Detects the starting position of an incoming packet based on preambles. It is extremely important for the synchronizer to correctly estimate the OFDM symbol boundaries so that subsequent blocks process appropriate collection of samples together. In many implementations, the synchronizer also detects and corrects carrier frequency offset that is caused by the difference in the oscillator frequencies at transmitter and receiver or due to the Doppler Effect. The synchronizer uses the preamble to perform timing and frequency synchronization. There are many different implementations of the synchronizer, most of which involve auto-correlation and cross-correlation. For the synchronizer to support different protocols, it needs to know the preamble structure, the symbol size and the CP size of the protocol.
Serial to Parallel (S/P): Removes the cyclic prefix (CP) and then aggregates samples into symbols before passing them to the FFT. It also propagates the control information from the RX Controller to subsequent blocks.
FFT: Converts OFDM symbols from the time domain back into the frequency domain.
Channel Estimator: Uses the information from pilots to estimate and compensate for frequency-dependent signal

	802.11a	802.16
Scrambler		
shift register size	7 bits	15 bits
linear function	$x^7 + x^4 + 1$	$x^{15} + x^{14} + 1$
throughput	54Mbps	26.2Mbps
FEC Encoder (Reed-Solomon)		
encoder profile (N,K,T)	NA	(255,239,8)
supported profiles (N,K,T)	NA	(12,12,0), (32,24,4), (40,36,2), (64,48,8), (80,72,4), (108,96,6), (120,108,6)
throughput	NA	29.1Mbps
FEC Encoder (Conv)		
constraint length	7	7
supported rates	$1/2$, $2/3$, $3/4$	$1/2$, $2/3$, $3/4$, $5/6$
generator polynomials	133_{OCT} & 171_{OCT}	171_{OCT} & 133_{OCT}
throughput	72Mbps	35Mbps
Interleaver		
block size (bits) (blockSize)	48, 96, 192, 288	192, 384, 768, 1152
throughput	1 block per 4μs	1 block per 33μs
Mapper		
modulations	BPSK, QPSK, 16-QAM, 64-QAM	BPSK, QPSK, 16-QAM, 64-QAM
throughput	48 samples per 4μs	192 samples per 33μs
Pilot/Guard Insertion		
pilot indices	-21, -7, 7, 21	-88, -63, -38, -13, 13, 38, 63, 88
guard indices	-32 to -27, 0, 27 to 31	-128 to -101, 0, 101 to 127
throughput	64 samples per 4μs	256 samples per 33μs
IFFT		
size	64	256
throughput	64 samples per 4μs	256 samples per 33μs
CP. Insertion		
CP size	16 (32 for preamble)	8, 16, 32, 64
short preamble	8 16-sample symbols	4 64-sample symbols
long preamble	2 64-sample symbols	2 128-sample symbols
throughput	80 samples per 4μs	264 samples per 33μs
Synchronizer		
preamble settings	same as CP. Insertion	same as CP. Insertion
throughput	20M samples per sec	8M samples per sec
S/P		
symbol / CP sizes	64 / 16	256 / 8, 16, 32, 64
throughput	64 samples per 4μs	256 samples per 33μs
FFT		
size	64	256
throughput	64 samples per 4μs	256 samples per 33μs
Channel Estimator		
preamble settings	same as CP. Insertion	same as CP. Insertion
pilot/guard settings	same as Pilot/Guard Insertion	same as Pilot/Guard Insertion
throughput	48 samples per 4μs	192 samples per 33μs
Demapper		
demodulations	BPSK, QPSK, 16-QAM, 64-QAM	BPSK, QPSK, 16-QAM, 64-QAM
throughput	288 decisions per 4μs	1152 decisions per 33μs
Deinterleaver		
block size (decisions)	48, 96, 192, 288	192, 384, 768, 1152
throughput	1 block per 4μs	1 block per 33μs
FEC Decoder (Viterbi)		
conv. code settings	same as Conv. Encoder	same as Conv. Encoder
throughput	54Mbps	29.1Mbps
FEC Decoder (Reed-Solomon)		
Reed-Solomon settings	NA	same as Reed-Solomon Encoder
throughput	NA	26.2Mbps
Descrambler		
LFSR settings	same as Scrambler	same as Scrambler
throughput	54Mbps	26.2Mbps

Figure 3. Algorithmic settings of 802.11a and 802.16 transceivers

degradation. The channel estimator estimates and corrects the errors caused by multipath interference. Similar to the synchronizer, there are many different algorithms for channel estimation. Many of them use either the preambles or the pilots to estimate the effect of the interference on each data subcarrier. We parameterize the channel estimator by protocol-specific preamble and pilot values.

Demapper: Demodulates data and converts samples to encoded bits, which are used by the FEC decoder. The number of encoded bits generated per sample is determined by the specific modulation scheme. The parameters of this block are modulation schemes supported and the functions for converting samples to decisions.

Deinterleaver: Reverses the interleaving performed by transmitter and restores the original arrangement of bits.

FEC Decoder: Uses the redundant information that was introduced at the transmitter to detect and correct any errors that may have occurred during transmission. Both 802.11a and 802.16 use the Viterbi algorithm [13] to decode convolutionally encoded data. To support multiple protocols, the decoder uses the same parameter settings as the convolutional encoder at the transmitter side. Since 802.16 also uses Reed-Solomon encoding, corresponding Reed-Solomon decoder that supports appropriate profiles is used in the receiver side.

Descrambler: Reverses the scrambling performed by the transmitter.

RX Controller: Based on the decoded data received from Descrambler, the RX Controller generates the control feedback to S/P block.

Figure 3 summarizes the parameters used by various blocks in 802.11a and 802.16.

4 General Considerations for Parametric Implementations

We employ several techniques to enable significant module reuse and customization across different protocols, architectures, and design points. The high-level structure in which modules are interconnected follows a transaction-level modeling style. Furthermore we restrict communication between the modules to pass messages containing control and data values. The control part is not modified by any module as the message flows through the pipeline and is stripped off before the message leaves the baseband processing section. The control part varies in type and value for different protocols; a challenge in coding reusable modules is to relate different (dynamic) control information to the different (static) instantiations of a module. In this section, we illustrate this point using the parameterized coding of the Scrambler block.

4.1 Transaction-Level Modeling Style Interfaces

In order to decouple modules, the interface of each module is implemented with the ready/enable handshaking approach, which is embodied in the `Put` and `Get` interfaces [3]. The Bluespec compiler automatically enforces ready/enable handshaking between modules connected in this manner such that an upstream module will block if the downstream module is not ready. This interface style is compatible with transaction-level modeling (TLM) approach [5]. Thus, the interface of each module is declared as follows:

```
interface Block#(type in_mesg, type out_mesg);
  interface Put#(in_mesg)  in;
  interface Get#(out_mesg) out;
endinterface
```

This is a highly polymorphic definition in that the types of the messages going in and out of the module `Block` are themselves passed in as static parameters. Note that parameterized types are indicated with a hash mark (e.g. `Mesg#`).

The code above defines that the interface has an input method called `in` and an output method called `out` and these methods have `Put` and `Get` interfaces, respectively. `Put` and `Get` interfaces are part of the Bluespec library. By making use of the `mkConnection` function one can easily connect the `Get` method of a module to the `Put` method of another module provided the declared types match. The BSV compiler automatically generates the logic needed to transfer the data from the `Get` to the `Put` whenever both methods are ready.

The generic OFDM transmitter pipeline can be described as follows using the `mkConnection` function:

```
mkConnection(tx_controller.out, scrambler.in);
mkConnection(scrambler.out, encoder.in);
mkConnection(encoder.out, interleaver.in);
mkConnection(interleaver.out, mapper.in);
mkConnection(mapper.out, pilotInsert.in);
mkConnection(pilotInsert.out, ifft.in);
mkConnection(ifft.out, cpInsert.in);
```

The modules of the receiver pipeline are connected in a similar fashion. Since the structure of the OFDM transceiver is the same across protocols, this portion of the code will remain the same regardless of the protocol we are implementing. The changes will appear when we instantiate the modules from the module definitions. BSV uses the symbol $<$- for module instantiation. In the following code, we show how one can instantiate the modules for the transceiver using module definitions (all the functions whose names start with `mk`). All module definitions are generic except those that instantiate the controllers, the encoder and the decoder:

Transmitter Modules Instantiations:

```
tx_controller <- mkProtocol_Controller();
// protocol-specific
scrambler     <- mkScrambler(scramblerCtrl,
                             lfsrSz,
                             lFunc);
encoder       <- mkEncoder();
// protocol-specific
interleaver   <- mkInterleaver(intrlvrCtrl,
                               intrlvrGetIdx,
                               blockSize);
mapper        <- mkMapper(mapperCtrl,
                          invertInput);
pilot_insert  <- mkPilot_Insert(guardPos,
                                pilotPos,
                                pilotFuncs);
ifft          <- mkFFT_IFFT(ifftCtrl,
                            ifftSize);
```

```
  cp_insert     <- mkCP_Insert(cpCtrl,
                               symbolSize);
```

Different OFDM protocols use different collection of FEC schemes. For instance, as shown in Figure 3, 802.11a uses convolutional encoder with puncture while and 802.16 uses Reed-Solomon encoder followed by convolutional encoder with puncture. The encoders are sufficiently different that sharing a parameterized module definition would be awkward, as a result, we made separate definitions, as shown below:

802.16 encoder:

```
module mkEncoder (Encoder80216);
  // state elements
  encoder_rs    <- mkRS_Encoder(rs_encoderCtrl);
  encoder_conv  <- mkConv_Encoder(poly_g_0,
                                  poly_g_1);
  encoder_punc  <- mkPuncturer(puncCtrl,
                               puncFuncs);

  // connections
  mkConnection(encoder_rs.out,
               encoder_conv.in);
  mkConnection(encoder_conv.out,
               encoder_punc.in);

  // Get, Put methods
  interface in = encoder_rs.in;
  interface out = encoder_punc.out;
endmodule
```

802.11a encoder:

```
module mkEncoder (Encoder80211);
  // state elements
  encoder_conv  <- mkConv_Encoder(poly_g_0,
                                  poly_g_1);
  encoder_punc  <- mkPuncturer(puncCtrl,
                               puncFuncs);

  // connections
  mkConnection(encoder_conv.out,
               encoder_punc.in);

  // Get, Put methods
  interface in = encoder_conv.in;
  interface out = encoder_punc.out;
endmodule
```

OFDM Messages: In our OFDM library, blocks communicate with each other by passing OFDM messages, which consist of two fields: `control` and `data`. The type and format of the data is independent of the protocol being implemented, but the `control` encoding is protocol-specific and generated by the TX Controller or RX Controller. Consequently, we define the following generic type for OFDM messages:

```
typedef struct{
    ctrl_t                control;
    Vector#(sz, data_t)   data;
} Mesg#(type ctrl_t, numeric type sz,
        type data_t);
```

The type of `control` is defined by `ctrl_t` and the type of `data` by a vector of `data_t` of size `sz`. `ctrl_t`, `data_t` and `sz` are static parameters which are evaluated at compiled time. Thus, the `in_mesg` and `out_msg` types are all instances of `Mesg` type. For example, the type of the message for the 802.11a Scrambler (`ScramMesg`) may be defined as follows, where types `ctrl_t` and `data_t` are instantiated to `Ctrl80211` and `Bit#(1)`, respectively:

```
typedef Mesg#(Ctrl80211, sz, Bit#(1))
        ScramMesg#(numeric type sz);
```

Note that the `ScramMesg` type still has `sz` as a type parameter.

4.2 Parameterization of the Scrambler

In order to support various protocols, it is important that our implementations be as flexible as possible. We achieve this by parameterizing the implementation for both data types and widths. We illustrate this by describing the implementation of the scrambler module.

The scrambler randomizes the input bit stream by XORing each bit with a pseudo-random binary sequence (PRBS) generated by a linear feedback shift register (LFSR). For example, for the linear function $x^7 + x^4 + 1$, we first compute the `feedback` bit by XORing the 4th and 7th bit of the random number in `lfsr`. Then, we generate the output bit by XORing `inData` with the `feedback` bit. Finally, we compute the new random number by shifting the `feedback` bit into the current random number. For higher performance, we can process up to `steps` bits of `inData` at a time.

Function `scramble` given below implements the LFSR. In BSV, functions are compiled to combinatorial circuits. This function takes as input the coefficients of the LFSR polynomial as integer vector `lFunc`, the initial value of the shift register `lfsr`, and the input bit vector `inData`. It returns a 2-tuple of values: the new value of the shift register and the scrambled data.

```
function Tuple2#(Bit#(lfsrSz),
                 Vector#(steps, Bit#(1)))
         scramble(
           Vector#(fsz, Integer)   lFunc,
           Bit#(lfsrSz)            lfsr,
           Vector#(steps, Bit#(1)) inData);

   Vector#(steps, Bit#(1)) outData;
   Bit#(lfsrSz) nextLfsr = lfsr;
   Bit#(1) feedback;
   for(i = 0; i < steps; i = i + 1)
   begin
      feedback = 0;
      // loop to generate the feedback bit
      for(j = 0; j < fsz; j = j + 1)
          feedback = nextLfsr[lFunc[j]] ^
                     feedback; // XOR
      // XOR feedback and inData for outData
      outData[i] = inData[i] ^ feedback;
      // shift feedback into LSB
      nextLfsr = {nextLfsr[n-2:0],
                  feedback};
   end
   return tuple2(nextLfsr, outData);
endfunction
```

One important note about this code is that it represents a combinational circuit. All the `for` loops are completely unrolled at compile time to generate a DAG during the static elaboration phase of the BSV compiler. Consequently, the

above definition can be compiled only if lFunc (the linear function), lfsrSz (the shift register size) and steps (the number of bits to be processed) are known at compile time, and they must be passed as parameters to the mkScrambler when it is used to instantiate a scrambler module.

With our definition, the scramble function can be used by both the 802.11a and the 802.16 protocols. When we use it in 802.11a, the lFunc will be a vector containing values 4 and 7, and the fsz will be 7. On the other hand, lFunc will contain 14 and 15 and fsz will be 15 when it is used in 802.16. The number of bits to be processed (steps) is not specified by the protocol and can be set to meet the performance goals. It turns out that the scrambler is not the slowest module and thus, even the steps value of one is sufficient to meet the performance requirements.

Dynamic Parameters: The scrambler can have three operational modes:

1. Normal: Input is randomized using the current LFSR state
2. Bypass: Input is forwarded without processing
3. NewSeed: The LFSR state is reset with a given value and then the input is randomized.

The information regarding how the input data is to be processed is specified in the control part of the message. This information is extracted by the scramblerCtrl function in the following scrambler module rule:

```
rule scrambling(True); //scrambling rule
  let mesg = inQ.first();
  let gCtrl = mesg.ctrl;
  let sCtrl = scramblerCtrl(gCtrl);
  let data = mesg.data;
  case (sCtrl) matches
     tagged Bypass:
        outQ.enq(Mesg{ctrl: gCtrl,
                      data: data});
     tagged Normal:
     begin
        match {.oBits, .oSeed} =
            scramble(data, lfsr);
        lfsr <= oSeed;
        outQ.enq(Mesg{ctrl: gCtrl,
                      data: oBits);
     end
     tagged NewSeed .nSeed:
     begin
        match {.oBits, .oSeed} =
            scramble(data, nSeed);
        lfsr <= oSeed;
        outQ.enq(Mesg{ctrl: gCtrl,
                       data: oBits);
     end
    endcase
endrule
```

A challenge arises because different protocols use the scrambler differently. For example, the 802.11a protocol requires the scrambler not to scramble the header, while the 802.16 protocol requires the header to be scrambled too. This is why, as we pointed out earlier, the control part encoding is protocol-specific.

In our implementation, the control encoding of both 802.11a and 802.16 contain a field called region. The possible values for the region field for 802.11a are Header, FirstData, Data and Tail; The possible values for the region field for 802.16 are FirstData, Data and Tail. In addition to the region field, the 802.16 control encoding also contains a field seed which specifies the value for the scrambler seed. The following codes show the definitions of the scramblerCtrl for the two protocols:

802.11a scramblerCtrl:

```
function ScramblerCtrl
    scramblerCtrl80211(Ctrl80211 ctrl);
   return case (ctrl.region)
              Header:    Bypass;
              FirstData: NewSeed 7b101101;
              Data: Normal;
              Tail: Bypass;
          endcase;
endfunction
```

802.16 scramblerCtrl:

```
function ScramblerCtrl
    scramblerCtrl80216(Ctrl80216 ctrl);
   return case (ctrl.region)
              FirstData: NewSeed ctrl.seed;
              Data: Normal;
              Tail: Bypass;
          endcase;
endfunction
```

We solve this problem by passing the scrambleCtrl function as a parameter to the scrambler module. The follows show the definition of the mkScrambler with all the required parameters we discussed earlier:

```
module mkScrambler#(
              function ScramblerCtrl
                scramblerCtrl(ctrl_t ctrl),
              Integer lfsrSz,
              Vector#(fsz, Integer) lFunc)
       (Scrambler#(ctrl_t, steps));

   // state elements
   Reg#(Bit#(lfsrSz)) lfsr <- mkRegU;
   ... fifos (inQ, outQ) ...

   // Scrambling rule
   ...

   // Get, Put methods
   ...

endmodule
```

5 Performance Tuning through Architectural Exploration

We explored various design alternatives in order to ensure that our design meets the performance goals. To facilitate such architectural exploration, we parameterize the model so that different designs can be instantiated from the same code base during the static elaboration phase. A parameterized FFT which can be instantiated with different microarchitecture at synthesis time is shown in [4]. In this section, we illustrate a way of parameterizing components of the Viterbi decoder to enable architectural explorations.

A Viterbi decoder uses the Viterbi algorithm [13] to decode bitstreams encoded using a convolutional forward er-

ror correction code. The algorithm determines the most likely input bitstream given the received noisy encoded stream.

To understand the Viterbi algorithm, it helps to understand the convolutional encoder. A k-bit convolutional encoder is a state machine consisting of 2^k states, with transitions between states conditional on input bits and emitting a fixed number of output bits. In these two protocols, 2 bits are emitted per input bit. The current state of the encoder is named by the last k input bits. Because transitions are conditional on a single bit, each state of the encoder can be reached only from two previous states.

The Viterbi algorithm uses dynamic programming to find the most likely state transition sequence followed by the encoder given a received bit sequence. The algorithm retraces this sequence of states to reconstruct the original bit stream.

Too much time and memory would be required for the decoder to wait until it has received the entire data sequence before producing a result. However, we can achieve almost the same level of accuracy by recording only the last n transitions, and emitting one bit per timestep. In practice, a value of $n = 5(k + 1)$ yields satisfactory results. For 802.11a and 802.16, $k = 6$, so $n = 35$.

In our implementation, the Viterbi decoder consists of two modules: the path metric unit and the Traceback unit. The path metric unit contains a 2^k word memory, where each entry is essentially the probability that a sequence of input bits ended in that state. In practice, cumulative error between the hypothesized bit stream and the received bit stream for that state is used as the *path metric* for that state. The traceback unit records the most likely n state sequence leading to each state, encoded as one bit per state transition, logically organized as an n entry shift register where each entry is 2^k bits.

The path metric unit updates all 2^k entries for every 2 observations received from the demapper module. Once all the new path metrics are computed, the old path metrics can be discarded. As it computes each new path metric, it records the previous state pointer in the traceback unit.

After the path metric unit updates the values for all the states, the traceback unit follows the recorded previous state pointers and emits the bit corresponding to the oldest transition in the sequence.

Path Metric Unit

The path metric unit, which is shown in Figure 4, contains one or more Add-Compare-Select (ACS) units for calculating the path metrics for each state. The ACS computes two path metrics at a time, as shown in Figure 5. The number of ACS units, which is a parameter to the path metric unit, controls number of path metrics updated per cycle.

The overall structure of the path metric unit is similar to that of a single FFT stage [4], with the FFT butterflies replaced by the ACS units. With the generalized pipelining technique presented in [4], we can easily parameterize the design of the path metric unit with the number of ACS units. This parameterization represents a tradeoff between

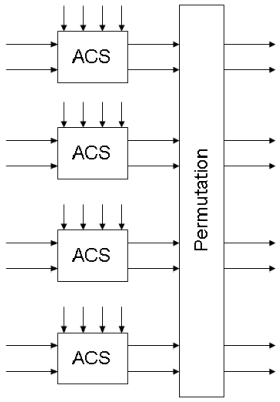

Figure 4. Path metric unit for k = 3

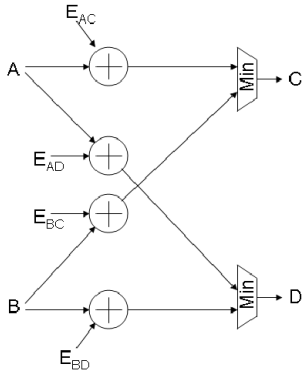

Figure 5. Add-Compare-Select unit

area and power: the area increases as we increase the number of ACS units, while the power decreases.

Traceback Unit

The traceback unit contains a $n \times 2^k$ bit shift register and a decoder which reconstructs one bit at a time by traversing the most likely state transition sequence. Traversing $n = 35$ transitions in one cycle leads to long cycle times. To reduce the critical path, we pipelined the traceback unit, with a parameterized pipeline depth.

In the pipelined implementation of the traceback unit, a single pipeline stage in the decoder traverses s pointers. There are t such stages, such that $s \times t = n$ and $n = 35$. Each pipeline register needs to store one traceback memory column of 2^k bits and a current state index.

We varied the number of pipeline stages (1, 5, 7 and 35) to analyze various design alternatives. Figure 6 shows the area and power measurements for different pipeline depths. The results reflect the minimum frequency required to support the 54 Mbps bitrate for 802.11a. The figure also shows the number of bits written to the traceback memory per cycle and the complexity of the address decode logic for reading out the bits from the traceback memory, which is the input size of the multiplexing logic to read the data from each element of the shift register. The results show that the 5-stage decode consumes the least area and least power. The

Pipeline depth	Bits written per cycle	Read complexity	Area (mm^2)	Frequency (MHz)	Power (mW)
35	35x64	2:1 mux	0.746	108	67.945
7	7x64	6:1 mux	0.713	108	52.310
5	5x64	8:1 mux	0.652	108	48.434
1	1x64	36:1 mux	Does not meet timing		

Figure 6. Viterbi synthesis results using TSMC 180nm library

1-stage version fails to meet the timing requirements.

6 Results

We wrote the designs of all the transceiver components in Bluespec. We verified each component using testbenches written in Bluespec. Simulations were carried out using Bluesim, which is a cycle-accurate simulator for Bluespec designs.

We generated a variety of RTL transceiver components for both the 802.11a and 802.16. All the components were generated from our OFDM library by passing appropriate parameter values.

The RTL was generated using the Bluespec Compiler (version 3.8.69), and then synthesized with Synopsys Design Compiler (version Y-2006.06) with the TSMC 180nm standard cell libraries. Figure 7 shows the post-synthesis area and power estimates as well as the clock frequency for the design to meet its respective standard. The power estimates were statically generated by Design Compiler.

The lines of code for each parameterized module and general libraries that we implemented are given in the table. The entire code is less than 8000 lines.

The 802.11a transceiver code takes 6544 lines of code, while the 802.16 transceiver code takes 6648. Of this, more than 85% of the code is shared. This gives us evidence of how much code we can leverage from the common library when implementing a different protocol.

7 Related Work

Dave *et al.* [4] discuss the microarchitectural exploration of an 802.11a transmitter via synthesizable highly-parameterized descriptions in Bluespec. They explore various microarchitectures of the IFFT, which is the most resource-intensive module of the 802.11a transmitter. Our paper is in the same genre but shows a much more elaborate use of parameters, primarily motivated by IP reuse.

Nordin *et al.* [11] present a parameterized generator for Discrete Fourier Transform (DFT). The generator accepts parameters like the input size of the Fourier transform as well as microarchitectural parameters that control the concurrency in the generated DFT cores. This approach is similar to our approach in the sense that it enables parameterization of both algorithmic and microarchitectural exploration. The approaches differ in that they rely on scripts and other techniques to generate Verilog programs, whereas we rely on the parameterization capability of the hardware-description language itself. The other difference is the scope of the parameterization — we have parameterized the whole OFDM transceiver, whereas [11] does so only for DFT. Similar remarks apply to Zhang *et al.* [14], who present a framework to enable algorithmic and architectural co-design for interference suppression in wireless receivers.

Salefski *et al.* [12] show how reconfigurable processing can meet the needs for wireless base station design while providing the programmability to allow field upgrades as standards evolve. This is an orthogonal concern to the sharing of code across multiple protocols.

Brier *et al.* [1] show how C/C++ models can be used for architectural exploration and verification of DSP modules. It proposes guidelines for building C/C++ models that aid in the verification process.

Hourani *et al.* describe domain specific tools for the signal processing [6, 7]. These tools automatically generate different architectural variations for signal processing algorithms, enabling algorithm experts who are not skilled hardware designers to make area/performance/power tradeoffs.

Krstic *et al.* [10] present a VHDL implementation of low power 802.11a baseband processor. They show the post-layout area and power estimations of their implementation which is synthesized with their 250nm standard cell library. Our high-level parameterized Bluespec implementation is comparable to their dedicated VHDL implementation in terms of area and power.

8 Conclusions

Power and cost constraints dictate a need for specialized circuits in the burgeoning market of handheld devices and sensors. Yet the ever increasing chip design costs and time-to-market of ASICs creates a major hurdle in the exploitation of this opportunity. We think that parameterized reusable components are the most immediate solution to this problem. In this paper, we have shown that various components for OFDM-based wireless protocols can be created in a manner so that they can be instantiated with appropriate parameter values to be part of different protocols. A powerful library of OFDM-based components can dramatically reduce the cost of implementing OFDM-based protocols and can also facilitate algorithmic exploration of new protocols. We already have a set of components that are rich enough to implement both 802.11a and 802.16 transceivers; we would like to evaluate if our library is rich enough to implement other OFDM protocols.

This type of component library development depends on a language like Bluespec SystemVerilog, which has the necessary type system and static elaboration facilities to make this level of parameterization feasible. One also needs a language with proper modular composition. Furthermore, without the ability to synthesize the designs, meaningful evaluations of these designs would be impossible.

Module	Lines of code		Area (mm^2)		Frequency (MHz)		Power (mW)	
	802.11a	802.16	802.11a	802.16	802.11a	802.16	802.11a	802.16
TX Controller	267	195	0.029	0.013	5	5	0.104	0.071
Scrambler	61		0.008	0.008	5	5	0.042	0.049
RS Encoder	–	105	–	0.027	–	5	–	0.114
Conv. Encoder	42		0.008	0.007	5	5	0.034	0.028
Puncturer	144		0.053	0.040	5	5	0.103	0.092
Interleaver	161		0.073	0.563	5	5	0.307	1.139
Mapper	89		0.420	1.719	5	5	1.572	6.346
Pilot	64		0.443	1.768	5	5	1.928	7.667
IFFT	318		4.736	11.651	5	5	11.961	38.319
CP Inserter	134		0.194	0.747	5	5	0.787	3.054
Synchronizer	1027		1.048	1.577	20	8	15.297	11.389
S/P	98		0.563	2.248	20	8	9.856	15.532
FFT	shared (IFFT)		4.526	10.733	5	5	11.431	35.300
Channel Est.	133		0.371	1.480	5	5	1.615	6.423
Demapper	202		0.303	0.828	5	5	0.627	2.328
Deinterleaver	shared (Interleaver)		0.212	0.779	5	5	0.900	3.375
Depuncturer	153		0.174	0.148	5	5	0.261	0.265
Viterbi Decoder	863		0.818	0.797	60	30	43.010	21.062
RS Decoder	–	45	–	0.007	–	5	–	0.027
Descrambler	shared (Scrambler)		0.008	0.008	5	5	0.042	0.049
RX Controller	153	86	0.321	1.263	5	5	2.424	9.638
Libraries	2163		–	–	–	–	–	–
Parameters	472	565	–	–	–	–	–	–
Total	6544	6648	14.308	36.411	–	–	102.301	162.267

Figure 7. Synthesis results for modules in 802.11a and 802.16 transceivers using the TSMC 180nm library. Area and power estimations are generated by Synopsys Design Compiler. The areas for the 802.11a and 802.16 designs are equivalent to 1.4M and 3.6M two-input NAND gates respectively.

Acknowledgments: This research is funded by Nokia Inc. (2006 Grant to CSAIL-MIT) and the National Science Foundation (Grant #CCF-0541164). In this project, the implementations of most modules were written by Man Cheuk Ng. The Reed-Solomon coders and the traceback unit of the Viterbi were written by Muralidaran Vijayaraghavan. The design of the FFT are based on the design of Nirav Dave's 64 points FFT with the extension to support different FFT sizes. This project is supervised by Gopal Raghavan and Jamey Hicks at NRC and Professor Arvind at MIT.

References

[1] D. Brier and R. Mitra. Use of C/C++ models for architecture exploration and verification of DSPs. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*. ACM Press, 2006.

[2] R. W. Chang. Synthesis of band-limited orthogonal signals for multichannel data transmission. *Bell System Technical Journal*, 45:1775–1796, Dec. 1966.

[3] N. Dave. Designing a Processor in Bluespec . Master's thesis, MIT, Cambridge, MA, Jan. 2005.

[4] N. Dave, M. Pellauer, S. Gerding, and Arvind. 802.11a Transmitter: A Case Study in Microarchitectural Exploration. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*. ACM-IEEE, 2006.

[5] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*, chapter 8. Springer, 2002.

[6] R. Hourani, R. Jenkal, R. Davis, and W. Alexander. Automated Architectural Exploration for Signal Processing Algorithms. In *Proc. IEEE 40th ASILOMAR Conf. on Signals, Systems and Computers*. IEEE, 2006.

[7] R. Hourani, R. Jenkal, R. Davis, and W. Alexander. Tool integration for Signal Processing Architectural Exploration. In *Presentation in IEEE Electronic Design Process Symposium (EDPS'06)*. IEEE, 2006.

[8] IEEE. *IEEE standard 802.11a supplement. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1999.

[9] IEEE. *IEEE standard 802.16. Air Interface for Fixed Broadband Wireless Access Systems*, 2004.

[10] M. Krstic, K. Maharatna, A. Troya, E. Grass, and U. Jagdhold. Baseband Processor for IEEE 802.11a Standard with Embedded BIST. *Facta Universitatis, Series: Electronics and Energetics*, 17:231–239, Aug. 2004.

[11] G. Nordin, P. Milder, J. Hoe, and M. Puschel. Automatic Generation of Customized Discrete Fourier Transform IPs. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*. ACM Press, 2005.

[12] B. Salefski and L. Caglar. Re-configurable computing in wireless. In *DAC '01: Proceedings of the 38th annual conference on Design automation*. ACM Press, 2001.

[13] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. In *IEEE Transactions on Information Theory*, 1967.

[14] N. Zhang, B. Haller, and R. Broderson. Systematic architectural exploration for implementing interference suspension techniques in wireless receivers. In *2000 IEEE Workshop on Signal Processing Systems*. IEEE Signal Processing Society, 2000.