802.11a Transmitter: A Case Study in Microarchitectural Exploration

Nirav Dave, Michael Pellauer, Steve Gerding, & Arvind

Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
Email: {ndave, pellauer, sgerding, arvind}@mit.edu

Abstract

Hand-held devices have rigid constraints regarding power dissipation and energy consumption. Whether a new functionality can be supported often depends upon its power requirements. Concerns about the area (or cost) are generally addressed after a design can meet the performance and power requirements. Different micro-architectures have very different area, timing and power characteristics, and these need RTL-level models to be evaluated. In this paper we discuss the microarchitectural exploration of an 802.11a transmitter via synthesizable and highly-parametrized descriptions written in Bluespec SystemVerilog (BSV). We also briefly discuss why such architectural exploration would be practically infeasible without appropriate linguistic facilities.

No knowledge of 802.11a or BSV is needed to read this paper.

1. Introduction

802.11a is an IEEE standard for wireless communication [4]. The protocol translates raw bits from the MAC into *Orthogonal Frequency-Division Multiplexing (OFDM) symbols* or sets of 64 32-bit fixed-width complex numbers. The protocol is designed to operate at different data rates; at higher rates it consumes more input to produce each symbol. Regardless of the rate, all implementations must be able to generate an OFDM symbol every 4 microseconds.

Because wireless protocols generally operate in portable devices, we would like our designs to be as energy-efficient as possible. This energy requirement makes software-based implementations of the transmitter unreasonable; a good implementation on a current-generation programmable device would need thousands of instructions to generate a single symbol, requiring the processor to run in the hundreds of MHz range to meet the performance rate of 250K symbols/sec. A dedicated hardware block, on the other hand, may be able to meet the performance rate even while operating in the sub-MHz range and consequently consume two orders of magnitude less power.

In general, the power consumption of a design can be lowered by *parallelizing* the design and running it at a lower frequency. This requires duplicating hardware resources. A minimum power design can be achieved by lowering the clock frequency and voltage to just meet the performance requirement. On the other hand we can save area by *folding* the hardware — reusing the same hardware over multiple clock cycles — and running at a higher frequency. Though we are primarily concerned with keeping the power of the entire system to a minimum, it is not obvious *a priori* which particular transmitter microarchitecture is best: the large lowest power one, or a smaller higher-power design which frees up critical area resources for other parts of the design. The entire gamut of designs from highly parallel to very serial must be considered.

In this paper we describe the implementation of several highly-parametrized designs of the 802.11a transmitter in Bluespec SystemVerilog (BSV). We present performance, area, and power results for all of these designs. We discuss the benefits of a language with a strong enough type system and static elaboration capability to express and implement non-trivial parametrization. We conclude by arguing that without these kinds of linguistic facilities, such architectural exploration becomes significantly more difficult, to the point where it may not take place at all. In fact, *the main contribution of this paper is to show, by a real example, that such exploration is possible early in the the design process and yields deep insight into the area-power tradeoff.*

No knowledge of 802.11a is needed to understand the architectural explorations discussed in this paper. We explain BSV syntax as we use it. However, some familiarity with Verilog syntax is assumed.

Organization: We begin with an overview of the 802.11a transmitter and show why it is important to focus on the

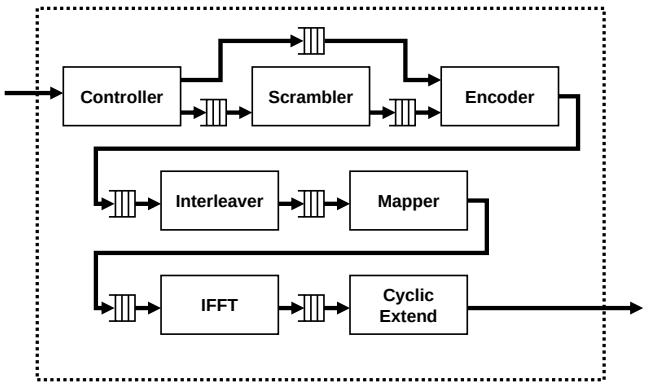

Figure 1. 802.11a Transmitter Design

IFFT block (Section 2). In Section 3 we present a combinational circuit implementation of the IFFT and use it as a reference implementation in the rest of the paper. We also show the power of BSV functions and parametrization in the design of combinational circuits. In Sections 4, 5, and 6 we discuss general microarchitectural explorations and how they are applied to our transmitter pipeline. In Section 7 we discuss the performance, area, and some power characteristics of each design. In Section 8 we discuss related work. Finally we conclude by discussing the role of HDLs in design exploration and reusable IP packaging.

2. 802.11a Transmitter Design

The 802.11a transmitter design can be decomposed into separate well-defined blocks as shown in the Figure 1.

Controller: The Controller receives packets from the MAC layer as a stream of data. The Controller is responsible for creating header packets for each data packet to be sent as well as for making sure that each part of the data stream which comprises a single packet has the correct control annotations (e.g. the encoding rate).

Scrambler: The Scrambler XORs each data packet with a pseudo-random pattern of bits. This pattern is concisely described at 1-bit per cycle using a 7-bit shift register and 2 XOR gates. A natural extension of this design would be to unroll the loops to operate on multiple bits per cycle. The initial value of the shift register is reset for each packet.

Convolutional Encoder: The Convolutional Encoder generates 2 bits of output for for every input bit it receives. Similar to the scrambler, the design can be described concisely as 1-bit per cycle with a shift register and a few XOR gates. Again unrolling the loop is an obvious and natural parametrization.

Puncturer: Our design only implements the lowest 3 data rates ($\{6, 12, 24\}$ Mb/s) of the 802.11a specification. At these rates the Puncturer does no operation on the data, so we will omit it from our design and discussion.

Interleaver: The Interleaver operates on the OFDM symbol, in block sizes of 48, 96, or 192 bits depending on which rate is being used. It reorders the bits in a single packet. Assuming each block only operates on 1 packet at a time, this means that at the fastest rate we can expect to output only once every 4 cycles.

Mapper: The mapper also operates on an OFDM symbol level. It takes the interleaved data and translates it directly into the 64 complex numbers representing different frequency "tones."

IFFT: The IFFT performs an 64-point inverse Fast Fourier Transform on the complex frequencies to translate them into the time domain, where they can then be transmitted wirelessly. Our initial implementation (discussed in greater detail in Section 3) was a combinational design based on a 4-point butterfly.

Cyclic Extender: The Cyclic Extender extends the IFFT-ed symbol by appending the beginning and end of the message to the full message body.

2.1. Preliminary Design Synthesis

When we began this project we had a good description of the 802.11a transmitter algorithm available to us. It took approximately three man-days to understand and code up the algorithm in BSV (the authors are BSV experts). This time included coding a library of arithmetic operations for complex numbers. This library is approximately 265 lines of BSV code.

The RTL for our initial design was generated using the Bluespec Compiler (version 3.8.67), and synthesized with Synopsis Design Compiler (version X-2005.09) with TSMC 0.18μm standard cell libraries. In this design, the steady state throughput at the highest-supported rate (24Mb/s) was 1 symbol for every four clock cycles. Therefore, we needed a clock frequency of 1MHz to meet the 4 microsecond requirement. The clock frequency for synthesis was set to 2 MHz to provide sufficient slack for place-and-route. With this setting the initial implementation had an area of 4.69 mm^2 or roughly 500K 2X1-NAND gates equivalents. The breakdown of the lines of code and relative areas for each block is given in Figure 2.

We can see that the number of lines of source code for a block have no correlation with the size of the area the block occupies. The Convolutional Encoder requires the most code to describe, but takes effectively no area. The IFFT, on the other hand, is only slightly shorter, yet represents a substantially larger fraction of the total area. Additionally, the critical path of the IFFT is many times larger than the critical path of any other block in the design.

Given these preliminary statistics, we focused our efforts on the design of the IFFT block. Ultimately we will present seven different designs for the transmitter created by plugging in seven different variations of the IFFT block.

Design Block	Lines of Code	Relative Area
Controller	49	0%
Scrambler	40	0%
Conv. Encoder	113	0%
Interleaver	76	1%
Mapper	112	11%
IFFT	95	85%
Cyc. Extender	23	3%

Figure 2. Initial Design Results

3. Baseline: Combinational IFFT

In this section we describe the combinational IFFT block from our initial transmitter design. This implementation serves both as a reference implementation for verification and as a baseline to compare with our alternative microarchitectures.

At a high level, the n-point IFFT can be partitioned into $log_k(n)$ stages of $\frac{n}{k}$ k-point "butterfly" submodules (bfly$k$ for short). At the end of each stage of butterflies, the output values are permuted before being passed to the next stage. It would be straightforward to create an IFFT description parametrized by n but such parametrization is not needed in this design because the 802.11a specification requires only a 64-point IFFT.

Even limited to a 64-point IFFT, we are free to choose the size of the butterfly submodule. An IFFT using bfly4s uses fewer arithmetic operators than one using bfly2s. By similar reasoning, a bfly8-based design uses fewer operators than bfly4. However, larger butterflies constrain where the computation can be partitioned, and can limit further microarchitectural choices. To simplify the discussion we only consider bfly4 sub-blocks in this work. Figure 3 shows the circuit structure of our combinational IFFT.

3.1. The bfly4 Function

There are many similarities between combinational hardware logic and functions in a software language. Both have well-defined inputs and outputs and are composed of simple sub-functions (in hardware, gates and in software, assembly instructions). Intermediate values have a hardware analog in wires.

In Bluespec a function definition corresponds exactly to a combinational logic definition. A function call is evaluated by the compiler, and the resulting combinational logic inlined. There is no notion of a run-time stack in BSV: even recursive function calls are totally unfolded at compile time.

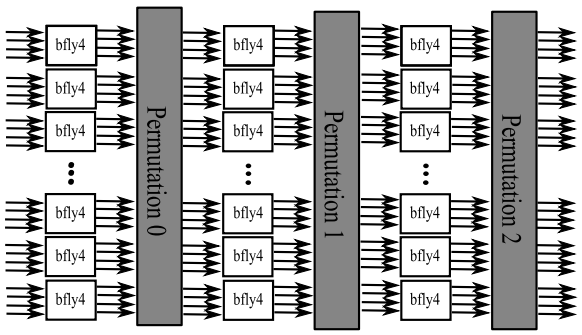

Figure 3. Combinational IFFT Module

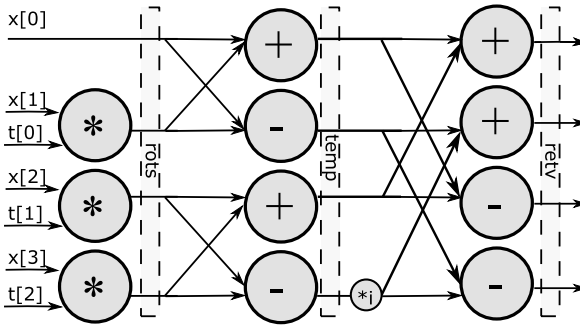

Figure 4. The bfly4 Circuit: twiddle values t[n] are statically-known parameters

The bfly4 circuit is shown in Figure 4. It takes as input 4 complex numbers to be transformed, and three "twiddle" factors which represent an initial rotation. The output of the circuit is four new complex numbers. This circuit can be described in Bluespec as the following function:

```
function Vector#(4,Complex#(n))
         bfly4(Vector#(3,Complex#(n)) twids,
               Vector#(4,Complex#(n)) xs);

  Vector#(4, Complex#(n)) retval = newVector(),
                          tao    = newVector(),
                          alpha  = newVector();

  Complex#(n) rots[0] = xs[0];
  Complex#(n) rots[1] = twids[0] * xs[1];
  Complex#(n) rots[2] = twids[1] * xs[2];
  Complex#(n) rots[3] = twids[2] * xs[3];

  Complex#(n) temp[0] = rots[0] + rots[2];
  Complex#(n) temp[1] = rots[0] - rots[2];
  Complex#(n) temp[2] = rots[1] + rots[3];
  Complex#(n) temp[3] = rots[1] - rots[3];

  // rotate temp_3 by 90 degrees
```

```
  temp[3] = mult_by_i(temp[3]);

  retv[0] = temp[0] + temp[2];
  retv[1] = temp[1] - temp[3];
  retv[2] = temp[0] - temp[2];
  retv[3] = temp[1] + temp[3];

  return retv;
endfunction
```

Note that the `Complex` type has been written in such a way that it represents complex numbers of any bit-precision n. Type parameters are indicated by the # sign. For example, `Vector#(4, Complex#(n))` is a vector of 4 n-bit precision complex numbers. The `newVector` function creates an uninitialized vector. The `Complex` type is a structure consisting of real and imaginary parts (i and q respectively):

```
typedef struct {
    SaturatingBit#(n) i;
    SaturatingBit#(n) q;
} Complex(type n);
```

All arithmetic on complex numbers is defined in terms of saturating fixed-point arithmetic. For lack of space we omit the description of these complex operators — all of them have been implemented in BSV using ordinary integer arithmetic and bitwise operations. The parametrization of the bfly4 is realized partially through the overloading of arithmetic operators; the compiler is able to select statically the correct operator based on its type.

A polymorphic BSV function such as the above bfly4 description can be thought of as a *combinational logic generator*. The Bluespec compiler instantiates the bfly4 function for a specific bit-width during a compiler phase known as *static elaboration*. During this phase the compiler:

- Instantiates functions and submodules with specific parameter values
- Unrolls loops and recursive function calls
- Performs aggressive constant propagation including the propagation of don't-care values

We can leverage static elaboration to create a more concise, more generalized description [1]. For example, a vector in the above function is simply a convenient way of grouping wires. The vector addresses are statically known and do not survive the elaboration phase. In our design we often use vectors of variables, and vectors of submodules such as registers.

In hardware compilation *constant propagation* sometimes achieves results which are surprising from a software point of view. For example, for each bfly4 in our combinational design the twiddle factors are statically known. Each twiddle factor is effectively a random number which rules out any word level simplification (e.g., multiplication by 1 or 0). But the bit-level representation of each constant allows the gate-level constant propagation to dramatically simplify the area-intensive multiply circuit. *A general bfly4 circuit which takes all values as inputs is 2.5 times larger than a specialized circuit with statically known twiddle factors (208μm^2 vs. 83μm^2)!*

Later, we will explore other variants of the IFFT where the multipliers cannot be statically optimized, but sharing of hardware occurs at a higher level.

3.2. IFFT Function

A straightforward way to represent the IFFT of Figure 3 is to write each bfly4 block explicitly. We use vectors to represent the intermediate values (wires) between the different stages of bfly4 blocks. We also need to define the specific twiddle values used for each bfly4 block. Similarly we will define a vector to represent the permutations between stages. Expressed as a function we get:

```
function ifftA(Vector#(64,Complex#(16)) x);
  prebfly0 = x;
  twid_0_0 = ... ;
  twid_0_1 = ... ;
// Stage 1
  postbfly0[3:0]   = bfly4(twid_0_0,
                           prebfly0[3:0]);
  ...
  postbfly0[63:60] = bfly4(twid_0_15,
                           prebfly0[63:60]);
  //Permute 1
  prebfly1[0] = postbfly0[0];
  prebfly1[1] = postbfly0[4];
  ...
// Stage 2
  postbfly1[3:0]   = bfly4(twid_1_0,
                           prebfly1[3:0]);
  ...
  postbfly1[63:60] = bfly4(twid_1_15,
                           prebfly1[63:60]);
  //Permute 2
  prebfly2[0] = postbfly1[0];
  prebfly2[1] = postbfly1[4];
  ...
// Stage 3
  postbfly2[3:0] = bfly4(twid_2_0,
                           prebfly2[3:0]);
  postbfly2[7:4] = bfly4(twid_2_1,
                           prebfly2[7:4]);
  ...
  final[0] = postbfly2[0];
  final[1] = postbfly2[4];
  return(final[63:0]);
endfunction
```

In fact this is how many Verilog programmers would write this code, probably using their favorite generation scripts.

Improved Representations in BSV: Looking at this description we can see a lot of replication. Each bfly4 in the stage has a very regular pattern in its input parameters. If

we organize all of the twiddles and permutations as vectors, then we can rewrite each stage using loops:

```
function ifftB(Vector#(64,Complex#(16)) x);
 //compute following constants at compile time
 twid0[0] = ... ; ... twid0[47] = ... ;
 twid1[0] = ... ; ... twid1[47] = ... ;
 twid2[0] = ... ; ... twid2[47] = ... ;

 permute[2:0][63:0]    = ... ;

 prebfly0 = x;

 //Stage 1
 for(Integer i = 0; i < 16; i = i + 1)
   postbfly0[4*i+3 : 4*i] =
       bfly4(    twid0[3*i+2 : 3*i],
                 prebfly0[4*i+3 : 4*i]);
 for(Integer i = 0; i < 64; i = i + 1)
    prebfly1[i] = postbfly0[permute[0][i]];

 //Stage 2
 for(Integer i = 0; i < 16; i = i + 1)
   postbfly1[4*i+3 : 4*i] =
       bfly4(   twid1[3*i+2 : 3*i],
                prebfly1[4*i+3 : 4*i]);
 for(Integer i = 0; i < 64; i = i + 1)
    prebfly2[i] = postbfly1[permute[1][i]];

 //Stage 3
 for(Integer i = 0; i < 16; i = i + 1)
   postbfly2[4*i+3 : 4*i] =
       bfly4(   twid2[4*i+3 : 4*i],
                prebfly2[4*i+3 : 4*i]);
 for(Integer i = 0; i < 64; i = i + 1)
        final[i] = postbfly2[permute[2][i]];

 return(final[63:0]);
endfunction
```

This new organization makes no change to the represented hardware. After the compiler unrolls the for-loops and does constant propagation the result is the exact same gate structure as `ifftA`.

Here we can see another level of regularity. This time it lies across all of the stages. We can rewrite the rule as:

```
function ifftC(Vector#(64,Complex#(16)) x);

 //compute following constants at compile time

 twid[2:0][47:0] = ... ;
 permute[2:0][63:0]    = ... ;

 for(Integer stage=0; stage<3; stage=stage+1)
   begin
      if (stage == 0)
          prebfly[stage][63:0] = x;
       else
          prebfly[stage][63:0] = out[stage-1];

    for(Integer i = 0; i < 16; i = i + 1)
      postbfly[stage][4*i+3 : 4*i] =
          bfly4(   twid[stage][3*i+2 : 3*i],
                   prebfly[stage][4*i+3 : 4*i]);

    for(Integer i = 0; i < 64; i = i + 1)
        out[i] = postbfly[stage][permute[stage][i]];
    end

 return(out[2][63:0]);
endfunction
```

Now we have a concise description of the hardware, exactly what an experienced BSV designer would have written in the first place. We have not shown the code for generating the twiddles and the permutation. Fortunately, like the bfly4 organization, these constants have a mathematical definition which can be represented as a function using sines, cosines, modulo, multiply, etc. A good question to ask is if such a description still represents good combinational logic. Again, just like the bfly4 parametrization, the inputs to each call to these functions are statically known. Consequently the compiler can aggressively optimize away the combinational logic to produce circuits exactly the same as `ifftA`.

As noted in the last section this IFFT design occupies roughly 85% of the total area and has a critical path several times larger than the critical path of any other block. Next we explore how to reuse parts of this hardware to reduce the area. All such designs involve introducing registers to hold intermediate values. As a stepping store to the designs that reuse hardware we first describe a simple pipelining of the combinational IFFT in the next section. This will also have the effect of reducing the critical path of the IFFT design by a factor of three.

4. The Pipelined IFFT

At a high level, pipelining is simply partitioning a task into a sequence of smaller sub-tasks which can be done in parallel. We can start processing the next chunk of data before we finish processing the previous chunk. Generally, the stages of a pipeline operate in lockstep (see Figure 5).

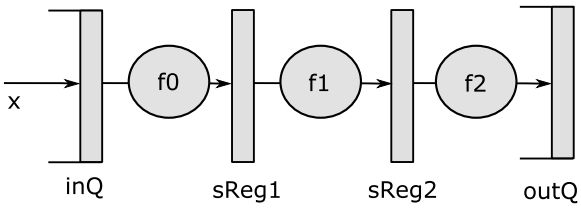

Figure 5. Synchronous Pipeline

In BSV, we can represent such a 3-stage pipeline using the following Guarded Atomic Action, or rule:

```
rule sync-pipeline (True);
  let sx0 = inQ.first();
  inQ.deq();
  sReg1 <= f0(sx0);
  let sx1 = sReg1;
  sReg2 <= f1(sx1);
  let sx2 = sReg2;
  outQ.enq(f2(sx2));
endrule
```

A rule consists of a set of *actions* that alter the state and a predicate (*guard*) which signifies when it is valid for these state changes to occur. The state altered by the above rule consists of two fifos (`inQ`, `outQ`) and two registers (`sReg1`, `sReg2`); the actions are to set the value of registers (e.g., `sReg2 <= f2(sx1)`), or to enqueue or dequeue from a fifo. The actions are combined into one atomic action which are inherently parallel. This means that all the reads happen before all writes as in non-blocking assignments in Verilog.

This rule shows that all stages of the pipeline read a value from the previous stage and store the value for the next stage. According to BSV semantics, this rule won't fire if either `inQ` is empty or `outQ` is full. These conditions are implicit and incorporated into the rule predicate by the compiler. For details of Bluespec synthesis and scheduling see [3] and [6].

Though straightforward to write, this rule has some problems. First, some registers may not hold valid values, for instance, when the pipeline has just started to fill. We must be careful not to enqueue junk values into the output queue. Second, since we do not move data when we cannot take a value from `inQ`, the last few values will be left in the pipeline. BSV provides elegant solutions to both these problems. We can prevent junk values from entering by providing a valid bit for each value and explicitly predicating the enqueue operation on `outQ` for valid data only. In BSV this is done using the `Maybe` datatype, which is a tagged union:

```
typedef union tagged {
    void   Invalid;
    data_T Valid;
} Maybe#(type data_T);
```

This is equivalent to adding a valid bit to any datatype.

We declare the registers to hold a `Maybe` type. Then we replace the `inQ` with a FIFO which returns `Invalid` when empty and `outQ` with a FIFO which accepts Maybe values (by simply ignoring enqueued `Invalid`).

Once this is done, we can easily prevent values from becoming stuck by checking to see if the input queue is empty and using the `Invalid` value in place of attempting to remove an input.

Note that the rule was already parametrized by functions $f0$, $f1$ and $f2$. This level of parametrization is sufficient for the pipelined IFFT; we only need to replace these function calls representing the 3 bfly stages. In fact we can generalize further by writing a single function `stage_f` given below. Most of the code for the `stage_f` is taken from the outer loop body of `ifftC`. The first parameter selects one of the 3 possible stage operations.

```
function stage_f(Bit#(2) stage,
             Vector#(64, Complex#(n)) prebfly);

 Vector#(64, Complex#(n)) out = newVector();

 for(Integer i = 0; i < 16; i = i + 1)
    postbfly[stage][4*i+3 : 4*i] =
             bfly4(  twids[stage][3*i+2 : 3*i],
                    prebfly[stage][4*i+3 : 4*i]);

 for(Integer i = 0; i < 64; i = i + 1)
    out = postbfly[stage][permute[stage][i]];
 return(out[63:0]);
endfunction
```

Now $f0$ (and similarly $f1$ and $f2$) can be defined as follows:

```
function Vector#(64, Complex#(n)) f0(
                  Vector#(64, Complex#(n)) x);
  return(stage_f(0,x));
endfunction
```

We note in passing that we can also write the synchronous pipeline rule so that it is parametrized by the number of stages. Assume that we have a combinational function f, similar to our `stage_f` function, which takes two parameters: the current stage i and the input value x and returns the result of fi on x. The following now models an n stage synchronous pipeline:

```
Vector#(TSub#(n,1), Reg#(Maybe#(data_T)))
          sRegs = newVector();
//Instantiate n registers
for (Integer i = 0; i < n - 1 ; i = i + 1)
  sRegs[i] <- mkReg(Invalid);

rule sync-pipeline (True);
  Maybe#(data_T) sx;
  for (Integer i = 1; i < n; i = i + 1)
   begin
    //Get stage input
    if (i != 0)
      sx = sRegs[i-1];
    else
     if (inQ.notEmpty)
       begin
         sx = inQ.first();
         inQ.deq();
       end
     else
       sx = Invalid;

    //Calculate value
    Maybe#(data_T) ox;
    case(sx) matches
```

```
      tagged Valid .x:
           ox = f(fromInteger(i),x);
      tagged  Invalid:
           ox = Invalid;
    endcase

    //Write Outputs
    if(i == n-1)
      outQ.enq(ox);
    else
      sRegs[i] <= ox;
endrule
```

It is important to keep in mind that because the stage parameter is known at compile time, the compiler can optimize each call of f to be specific to each stage.

5. Folded or Circularly-Pipelined IFFT

Our synchronous IFFT pipeline can produce a result every clock cycle. This is overkill as we can't produce that rate of input from the transmitter. If we can meet the specifications by producing a data element every three cycles then it may be possible to fold all three stages from the pipeline in Figure 5 into one, saving area. An example of such a pipeline structure is the folded pipeline shown in Figure 6, which assumes that all stages do identical computation represented by function f. In this structure a data element enters the pipeline, goes around three times, and then is ejected.

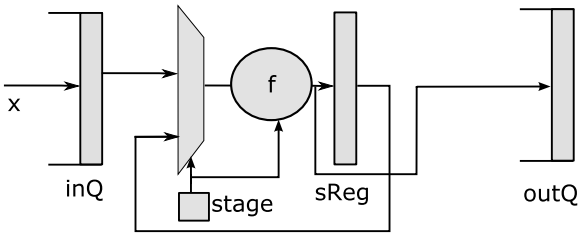

Figure 6. Folded or Circular Pipeline Design

In a folded pipeline, since the same hardware is used for conceptually different stages, we often need some extra state elements and muxes to choose the appropriate combinational logic. For example it is common to have a stage counter and associated control logic to remember where the data is in the pipeline. The code for an n-way folded pipeline such as shown in Figure 6 may be written as follows:

```
rule folded-pipeline (True);
 if (stage==0) in.deq();
 sxIn  = (stage==0) ? inQ.first() : sReg;
 sxOut = f(stage, sxIn);
 if (stage==n)
    outQ.enq(sxOut);
 else
    sReg <= sxOut;
 stage <= (stage == n)? 0 : stage + 1;
endrule
```

The stage function for a folded pipeline with three stages may be written as follows:

```
function f(stage, sx);
  case(stage)
    0: return f0(sx);
    1: return f1(sx);
    2: return f2(sx);
  endcase
endfunction
```

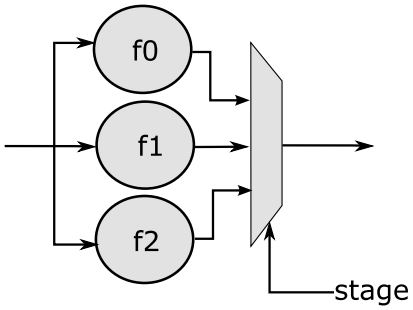

Figure 7. Function f

Considering the pipelines in Figures 5 and 6 and this definition of function f (shown in Figure 7) it is difficult to see how any hardware would be saved. The folded pipeline uses one pipeline register instead of two but it also introduces potentially two large muxes one at the input and one at the output of function f. If $f1$, $f2$ and $f3$ represent large combinational circuits then the real gain can come only by sharing the common parts of these circuits because these functions will never operate together at the same time. As an example, suppose each *fi* is composed of two parts: a sharable part *fs* and an unsharable part *fui* and *fi(x) = fs(fui(x))*. Given this information we could have written function f as follows:

```
function f (stage,sx);
 let sxt = case (stage)
                  0: return fu0(sx);
                  1: return fu1(sx);
                  2: return fu2(sx);
               endcase;
 return fs(sxt);
endfunction
```

where *fs(sx)* represents the shared logic among the three stages (Figure 8). A compiler may be able to do this level of

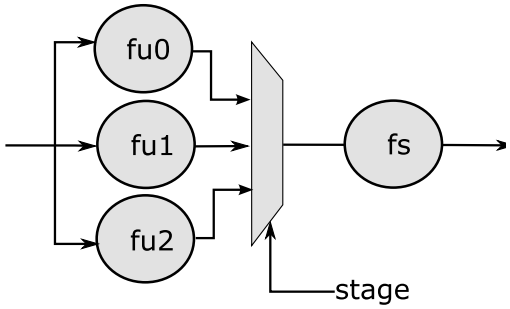

Figure 8. Function f with explicit sharing

common subexpression elimination automatically, but this form is guaranteed to generate the expected hardware.

It turns out that the `stage_f` function given earlier, effectively captures the sharing of the 16 bfly4 blocks in the design. To our dismay, In spite of this sharing, we found that the folded pipeline area ($5.89mm^2$) turned out to be larger than the area of the simple pipeline ($5.14mm^2$)!

Sharing of the bfly4s comes at a cost. Since each bfly4 in our design uses different twiddle constants in different stages it is no longer possible to take advantage of the constant twiddle factors in our optimizations. Recall from our earlier discussion in Section 3 that this results in an increase in area of a factor of 2.5; close to the three-fold reduction we expect from folding! In addition to this, additional area overhead is introduced by the muxes required to pass different twiddles into the bfly4s.

On further analysis, we discovered a contributing factor to the lack of sharing was the use of different permutations for different stages implying one more set of muxes. Once we realized this, we made an algorithmic adjustment and recalculated the constants so that the permutations were the same for each stage in the design, removing these muxes. As can be seen in Figure 9, the folded transmitter design using the new algorithm took 75% of the area of the simple pipeline design (3.97 vs 5.25).

It's not clear how much of this was due to the improved sharing, as we had to change the twiddle constants, which may have changed the possible optimizations. A possible explanation for the increased area for the combinational and simply pipelined IFFTs is the lack of twiddle-related optimizations.

Design	Old Area(mm^2)	New Area(mm^2)
Combinational	4.69	4.91
Simple Pipe	5.14	5.25
Folded Pipe	5.89	3.97

Figure 9. IFFT Area Results: New vs. Old Algorithm

6. Further Hardware Reuse: Super-Folded Pipeline IFFT

We wanted to determine if we could further reduce area by using less then 16 bfly4s (i.e. by folding the stage function in the folded pipeline). The stage function given eariler, can be modified to support this. We can "chunk" the i loop in the `stage_f` function to make use of m (< 16) bfly4s as follows:

```
for(Integer i1 = 0; i1 < 16; i1 = i1 + m)
   for(Integer i2 = 0; i2 < m; i2 = i2 + 1)
     begin
       let i = i1 + i2;
       postbfly[stage][4*i+3 : 4*i] =
             bfly4(   twids[stage][3*i+2 : 3*i],
                    prebfly[stage][4*i+3 : 4*i]);
```

This change by itself has no effect on the generated hardware as both loops would be unrolled fully. What we would like to is to build a new superfolded stage function using the inner $i2$ loop.

Consider the case of $m = 2$. What we would like is to pick up the data (64 complex numbers) from the input queue, go around the m bfly4s $\frac{48}{m}(= 24)$ times, and then enqueue the result (64 complex numbers) in the output queue. In each iteration, we will take the entire set of 64 complex numbers, but only manipulate $4 * m(= 8)$ numbers (using the m bfly4s) and leave the rest unchanged. We can deal with permutations by applying the permutation every $\frac{16}{m}(= 8)$ cycles (i.e. when all new data has been calculated).

Taking this approach, the new stage function with m bfly4 blocks is:

```
function Vector#(64,Complex#(n)) stage_f_m_bfly4
               (Bit#(6) stage,
                Vector#(64,Complex#(n)) s_in);

  Vector#(64,Complex#(n)) s_mid = s_in;

  Bit#(6) st16 = stage % 16;
  for(Bit#(6) i2=st16; i2 < st16+n; i2=i2+1)
    s_mid[4*i2+3:4*i2]
        = bfly4(twid(stage[5:4],i2),
                s_in[4*i2+3:4*i2]);

  // permute
  Vector#(64,Complex#(n)) s_out  = newVector();
  for(Integer i = 0; i < 64; i = i + 1)
    s_out[i] = s_mid[permute[i]];

  //return permuted value is appropriate
  return ((st16+m == 16) ? s_out[63:0]:
                           s_mid[63:0]);
endfunction
```

Our new stage function has the exact same form as our original folded design, differing only in the number of iterations

required to complete the computation. Consequently, the rule is also almost the same:

```
rule super-folded-pipeline(True);
  sx = (stage == 0) ? inQ.first(): sReg;
  if(stage == 0)
      inQ.deq();
  Vector#(64,Complex#(n)) sout =
              stage_f_m_bfly4(stage, sx);

  //constant integer divide (optimized away)
  if(stage == (3*16 div m) - 1)
    begin
      outQ.enq(sout);
      stage <= 0;
    end
  else
    begin
      sReg <= sout;
      stage <= stage + 1;
    end
  endrule
```

7. Results

In our explorations we described many different versions of the 802.11a transmitter, varying only the IFFT block implementation. We used 7 specific IFFT designs: the combinational version, a synchronous pipeline version, and 5 super-folded pipeline versions with 16, 8, 4, 2, and 1 bfly4 nodes respectively.

We were able to perform this exploration very quickly. Including the initial 3 man-days to describe the initial design, generating the variants took only an additional 2 man-days. This includes the changing of the algorithm to use constant permutations.

We took all 7 transmitter designs using our new constant permutation IFFT algorithm and ran them through gate-level synthesis (as described in Section 2) to get area estimates. Each of these synthesis runs took approximately 8 hours. We then fed these gate-level descriptions into Sequence Design PowerTheatre to get power estimates. All these results are presented in Figure 10.

At their maximum frequencies, all the designs easily met the performance criteria of producing a symbol every 4 microseconds. As expected, the pipelined design was largest, followed by the combinational, and the folded designs in descending order of the number of bfly4 blocks used.

For power, we see that as we fold our pipeline we take more power. This makes intuitive sense; folding forces the remaining bfly4 blocks to be general, and causes each individual block to be switched proportionally faster, which means more power usage from the IFFT (plus the additional power from a global increase in clock frequency across the entire clock domain).

After our exploration we found that the 5 folded designs, and the combinational design were all Pareto optimal and therefore candidates for the final 802.11a transmitter block in our full system. While the pipelined IFFT design itself is Pareto optimal for IFFT designs, the inability to gain further power reduction from voltage scaling causes it to be worse overall than the combinational design.

In our analysis it became clear that some parts of the 802.11a pipeline (e.g. the IFFT) were constantly generating data at full throughput, while other parts cannot be constantly busy. Thus, these idling blocks could operate at a much lower frequency without affecting performance. This seems to indicate that a multiple-clock domain [2] design should be explored as it could further improve the power usage results.

Lastly, since Place and Route takes a long time (additional tens of hours), our area and timing numbers were generated at the synthesized gate level. Methodologically this is acceptable because one is primarily interested in the relative merits of each design. To ensure that each design could meet timing after place and route, we placed the required clock period for synthesis as half of that needed to meet the performance requirement. We plan to run our designs through Place and Route to generate more accurate area and power numbers to validate our estimates.

8. Related Work

Efficient implementations of 802.11a is an active field of research. Here we limit consideration to works we consider to be representative of the broader approaches in the field.

Maharatna et al. [5] demonstrate that the IFFT can be effectively implemented as an ASIC using a similar design flow. While their specific microarchitecture achieves results competitive with commercial systems, they do not present the results of any architectural exploration. With a generalized description we could explore whether a variant of their system would achieve even better results.

Zhang and Broderson [8] conducted a case study of several possible implementations of both the IFFT of the 802.11a transmitter and the Viterbi block of a receiver. Rather than exploring standard tool-flow ASICs they used Function-Specific Reconfigurable Hardware (FSRH), standard cell ASICs which retains some dynamic configuration capabilities. Their work shows that the the FSRH implementation significantly outperforms DSP and FPGA implementations in terms of energy efficiency and computation density. We expect that non-reconfigurable ASICs such as those presented here to be at least as efficient.

An alternative strategy to implementing an efficient FFT is to use high-latency RAM banks to store the complex numbers and a small processing element to load addresses from the RAM, process them, and write them back. Son et al. [7] compared such an implementation to a pipelined FFT as presented in Section 4. They conclude that although

Transmitter Design (IFFT Block)	Area (mm^2)	Symbol Latency (cycles)	Throughput (cycle/symbol)	Min. Freq to Achieve Req. Rate	Avg. Power (mW)
Combinational	4.91	10	04	1.0 MHz	3.99
Pipelined	5.25	12	04	1.0 MHz	4.92
Folded (16 Bfly4s)	3.97	12	04	1.0 MHz	7.27
Folded (8 Bfly4s)	3.69	15	06	1.5 MHz	10.9
Folded (4 Bfly4s)	2.45	21	12	3.0 MHz	14.4
Folded (2 Bfly4s)	1.84	33	24	6.0 MHz	21.1
Folded (1 Bfly4)	1.52	57	48	12.0 MHz	34.6

Figure 10. Performance of 802.11a Transmitters for Various Implementations of the IFFT Block

a RAM-based implementation uses less area, it requires a significantly higher clock-speed, and thus is less power-efficient overall.

9. Conclusions

In this paper we explored various microarchitectures of an IFFT block, the critical resource-intensive module in an 802.11a transmitter. We demonstrated how languages with powerful static elaboration capabilities can result in hardware descriptions which are both more concise and more general. We used such generalized descriptions to explore the physical properties of a wide variety of microarchitectures early in the design process. We argue that such high-level language capabilities are essential if future architectural decisions are to be based on empirical evidence rather than designer intuition.

All the folded designs were generated from the same source description, only varying the input parameters. Even the other versions share a lot of common structure, such as the bfly4 definition and the representation of complex numbers and operators. This has a big implication for verification, because instead of verifying seven designs, we had to verify only three, and even these three leveraged submodules which had been unit-tested independently.

The six Pareto optimal designs we generated during exploration provide some good intuition into the area-power tradeoff possible in our design. To reduce our the area of our initial (combinational) design by 20% (the folded design), we increase our power usage by 75%. This tradeoff becomes less costly as we further reduce the design; if we wish to reduce the size by 70%, we increase the power usage by 760%.

In the future we wish to apply this methodology to more complex designs, such as the H.264 video decoder blocks. Such designs would have many more critical design blocks, and better emphasize the benefits of out methodology.

Acknowledgments: The authors would like to thank Nokia Inc. for funding this research. Also, the authors would like to thank Hadar Agam of Bluespec Inc., for her invaluable help in getting power estimates using Sequence Design PowerTheatre.

References

[1] Arvind, R. S. Nikhil, D. L. Rosenband, and N. Dave. High-level Synthesis: An Essential Ingredient for Designing Complex ASICs. In *Proceedings of ICCAD'04*, San Jose, CA, 2004.

[2] E. Czeck, R. Nanavati, and J. Stoy. Reliable design with multiple clock domains. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, 2006.

[3] J. C. Hoe and Arvind. Synthesis of Operation-Centric Hardware Descriptions. In *Proceedings of ICCAD'00*, pages 511–518, San Jose, CA, 2000.

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

[5] K. Maharatna, E. Grass, and U. Jagdhold. A 64-Point Fourier Transform Chip for High-Speed Wireless LAN Application Using OFDM. *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, 39(3), March 2004.

[6] D. L. Rosenband and Arvind. Modular Scheduling of Guarded Atomic Actions. In *Proceedings of DAC'04*, San Diego, CA, 2004.

[7] B. S. Son, B. G. Jo, M. H. Sunwoo, and Y. S. Kim. A High-Speed FFT Processor for OFDM Systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 281–284, 2002.

[8] N. Zhang and R. W. Brodersen. Architectural evaluation of flexible digital signal processing for wireless receivers. In *Proceedings of the Asilomar Conference on Signals, Systems and Computers*, pages 78–83, 2000.