802.11a Transmitter: A Case Study in Microarchitectural Exploration

Nirav Dave, Michael Pellauer, Steve Gerding, & Arvind

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.

```
  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

```
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)`

```
        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*

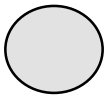

Figure 8. Function *f* with explicit sharing

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

It turns out that the `stage`

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();
```

Transmitter Design (IFFT Block)	Area (mm^2)	Symbol Latency (cycles)	Throughput (cycle/symbol)	Min. Freq to Achieve Req. Rate	