Architecture exploration in Bluespec

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Guest Lecture 6.973 (lecture 7)

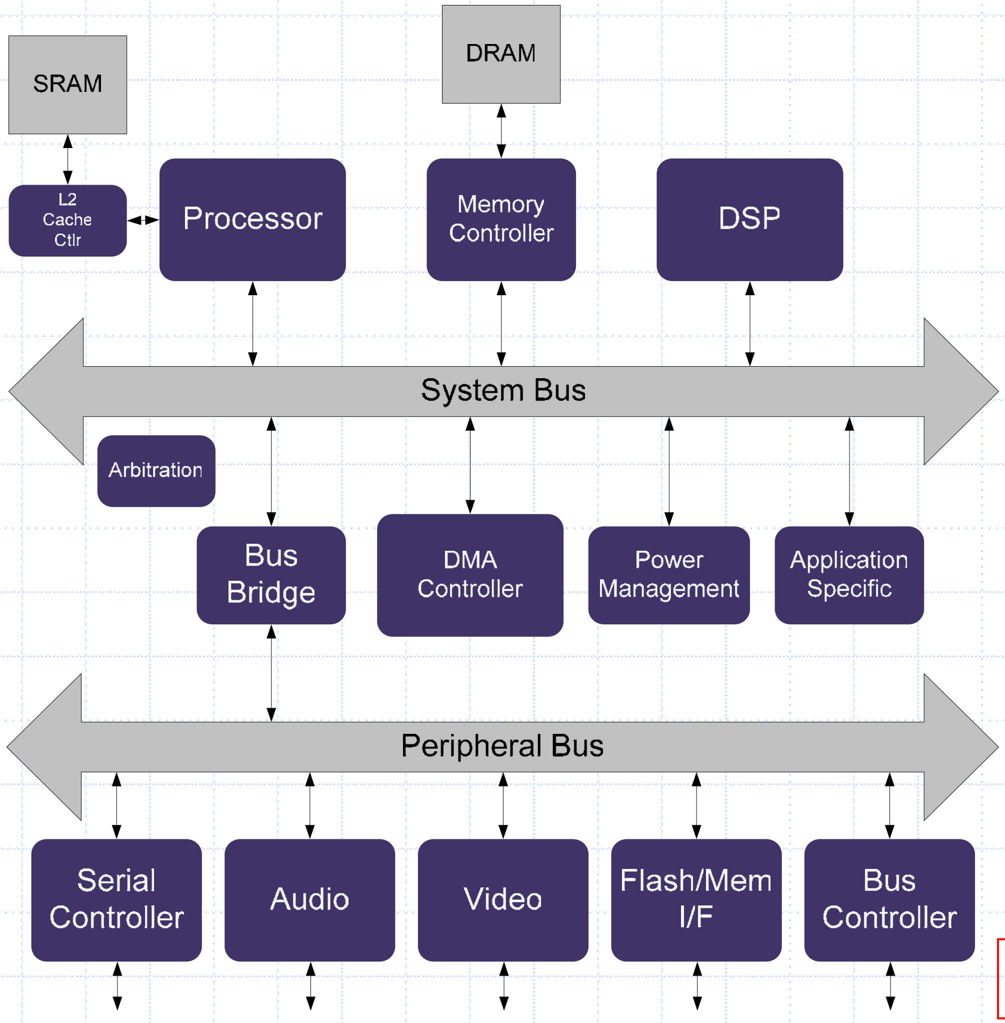
Chip design has become too risky a business

- Ever increasing size and complexity
  - Microprocessors: 100M gates ⇒ 1000M gates
  - ASICs: 5M to 10M gates ⇒ 50M to 100M gates
- Ever increasing costs and design team sizes
  - > $10M for a 10M gate ASIC
  - > $1M per re-spin in case of an error (does not include the redesign costs, which can be substantial)
- 18 months to design but o*nly an eight-mon* *selling opportunity in the market* *th*

⇒
- Fewer new chip-starts every year
- Looking for alternatives, e.g., FPGA's

Typical SOC Architecture

For example: Cell phone

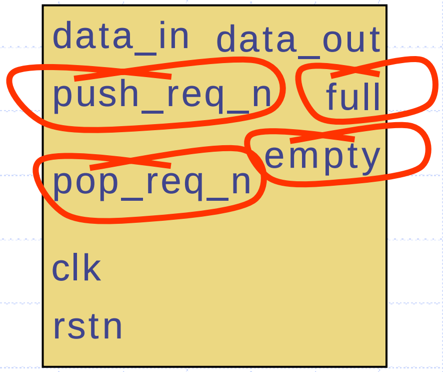


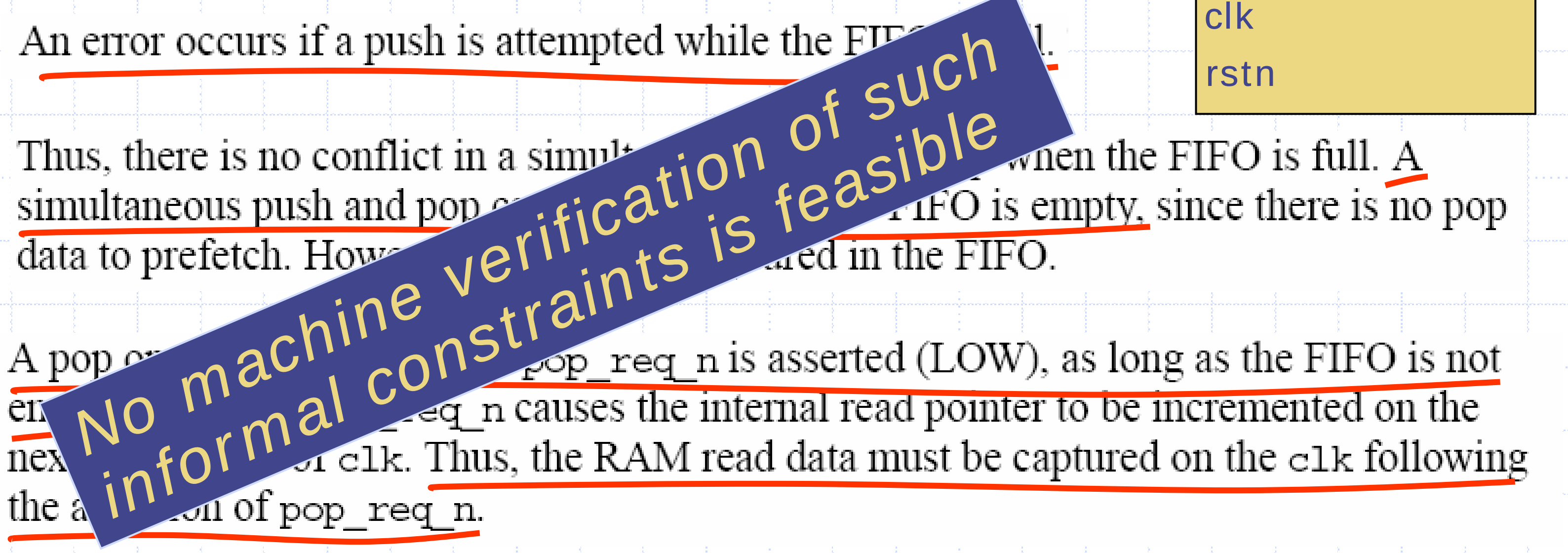
- Hardware/software development needs to be tightly coupled in order to meet performance/power/cost goals
- System validation for functionality and performance is very difficult
- Stable platform for software development
- *IP block reuse is essential to mitigate development costs*

IP = Intellectual Property

IP re-use sounds great until you start to use it...

Example: Commercially available FIFO IP block




An error occurs if a push is attempted while the FIF[illegible]l.

Thus, there is no conflict in a simult[illegible] when the FIFO is full. A simultaneous push and pop c[illegible] FIFO is empty, since there is no pop data to prefetch. How[illegible]red in the FIFO.

A pop o[illegible] pop_req_n is asserted (LOW), as long as the FIFO is not em[illegible]eq_n causes the internal read pointer to be incremented on the nex[illegible] of clk. Thus, the RAM read data must be captured on the clk following the a[illegible]ion of pop_req_n.

No machine verification of such informal constraints is feasible

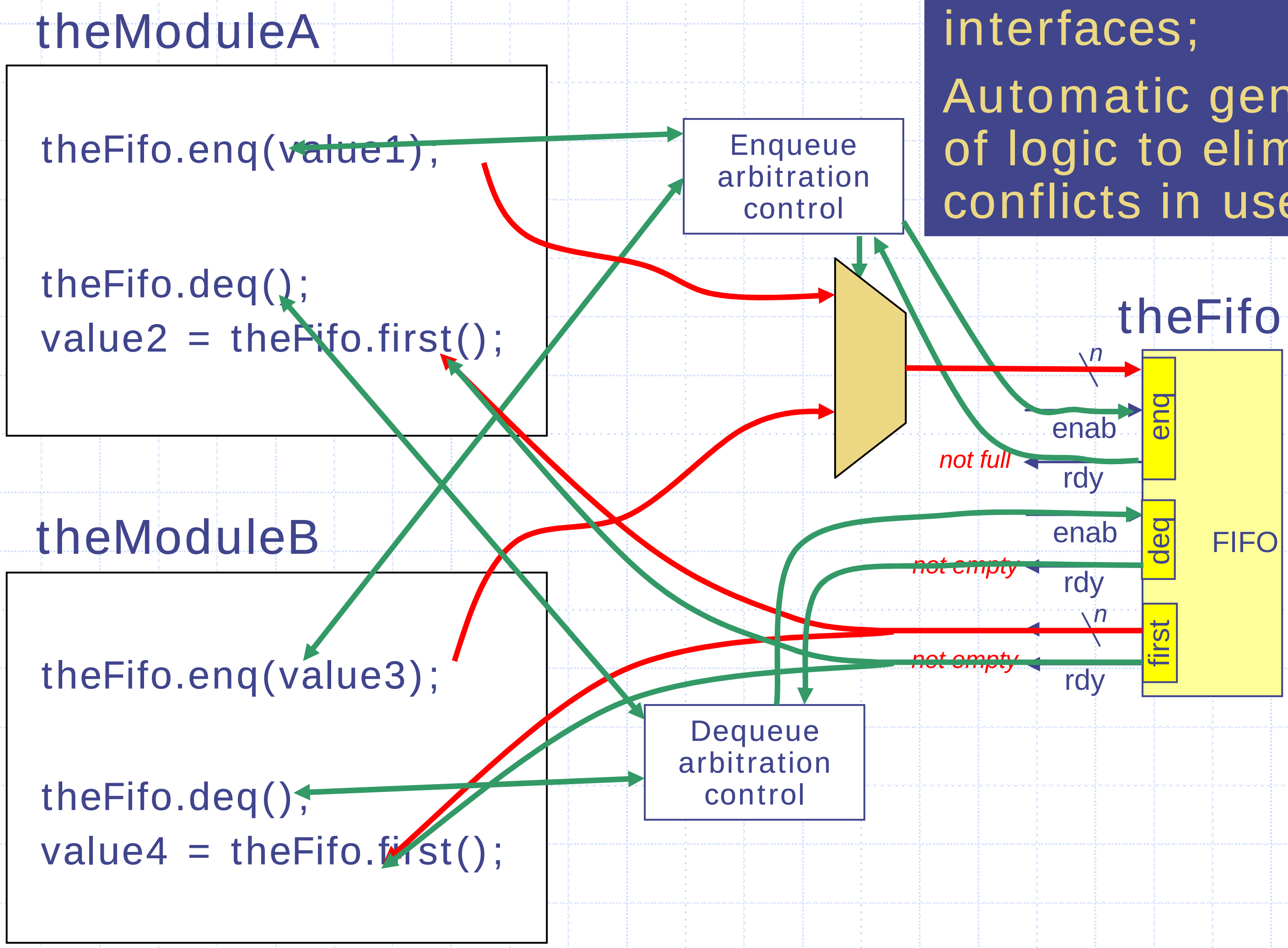
These constraints are spread over many pages of the documentation...

New semantics for expressing behavior to reduce design complexity

- Decentralize complexity: *Rule-based specifications (Guarded Atomic Actions)*
  - Let us think about one *rule* at a time
- Formalize composition: *Modules with guarded interfaces*
  - Automatically manage and ensure the correctness of connectivity, i.e., correct-by-construction methodology
  - Retain resilience to changes in design or layout, e.g. compute latency Δ's
  - Promote regularity of layout at macro level

Bluespec

Bluespec promotes composition through guarded interfaces



Self-documenting interfaces;

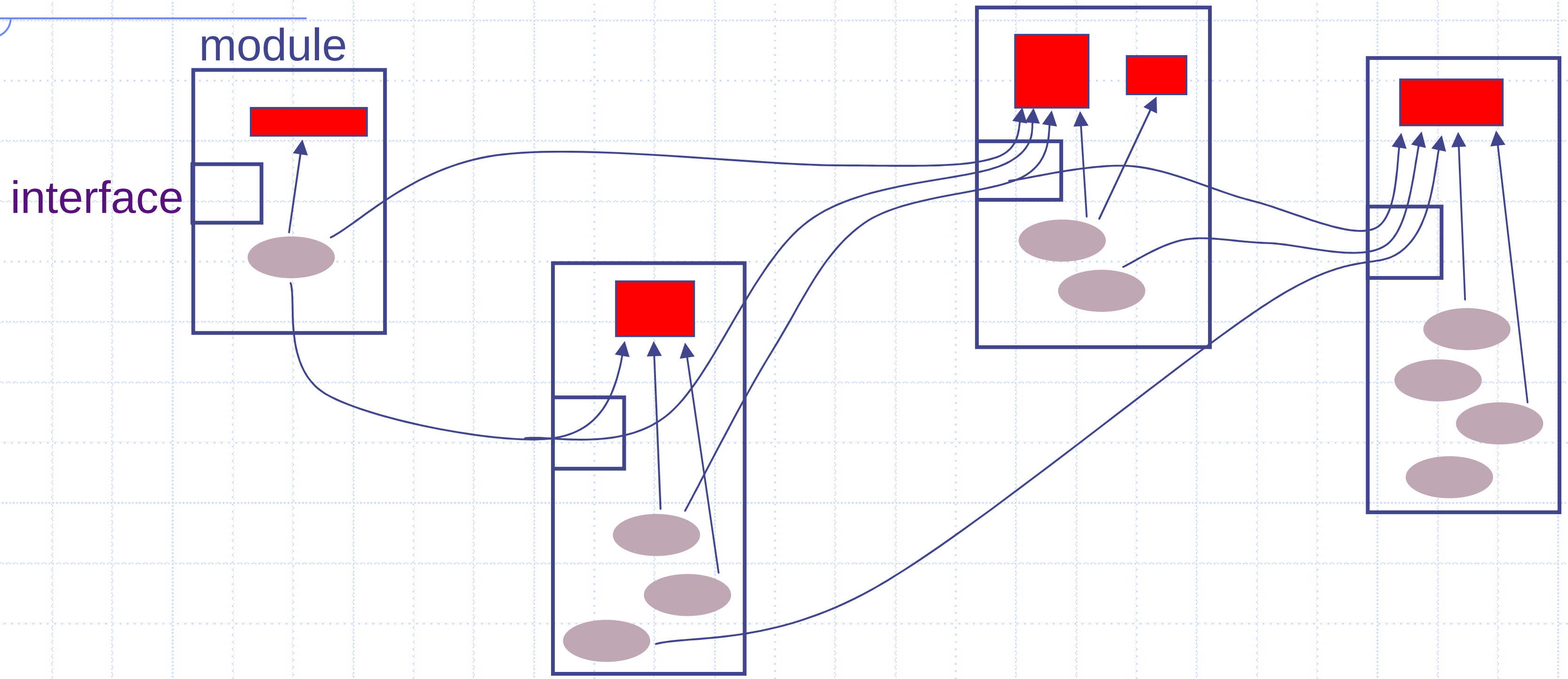
Automatic generation of logic to eliminate conflicts in use.

In Bluespec SystemVerilog (BSV) …

- Power to express complex static structures and constraints
  - Checked by the compiler
- "Micro-protocols" are managed by the compiler
  - The compiler generates the necessary hardware (muxing and control)
  - Micro-protocols need less or no verification
- Easier to make changes while preserving correctness

➔ *Smaller, simpler, clearer, more correct code*

Bluespec: State and Rules organized into *modules*



All *state* (e.g., Registers, FIFOs, RAMs, ...) is explicit.

Behavior is expressed in terms of atomic actions on the state:

Rule: condition ➔ action

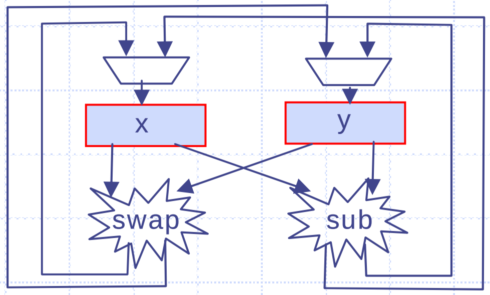
Rules can manipulate state in other modules only *via* their interfaces.

Programming with rules: A simple example

Euclid's algorithm for computing the Greatest Common Divisor (GCD):

15		6	
9		6	*subtract*
3		6	*subtract*
6		3	*swap*
3		3	*subtract*
0	*answer:*	3	*subtract*

GCD in BSV



```
module mkGCD (I_GCD);
   Reg#(int) x <- mkRegU;
   Reg#(int) y <- mkReg(0);

   rule swap ((x > y) &&  (y != 0));
       x <= y;  y <= x;
   endrule
   rule subtract ((x <= y) && (y != 0));
       y <= y – x;
   endrule

   method Action start(int a, int b) if (y==0);
      x <= a;  y <= b;
   endmethod
   method int result() if (y==0);
       return x;
   endmethod
endmodule
```

State

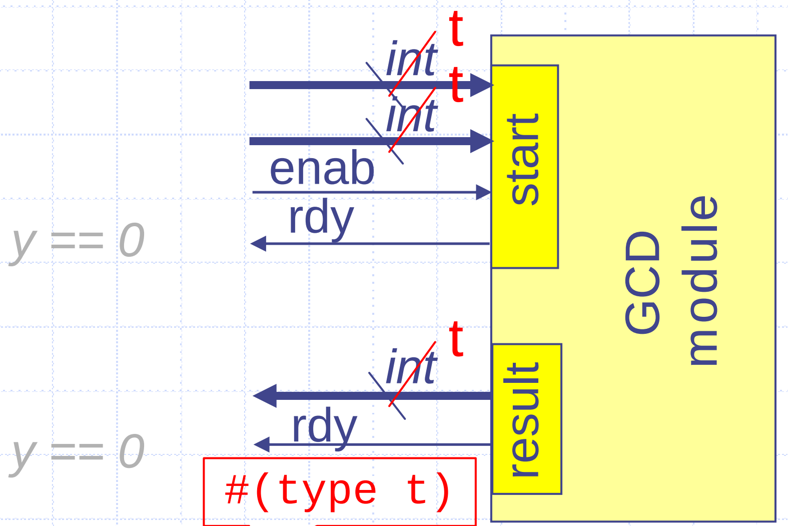
```
typedef int Int#(32)
```

Internal behavior

External interface

Assumes x /= 0 and y /= 0

GCD Hardware Module

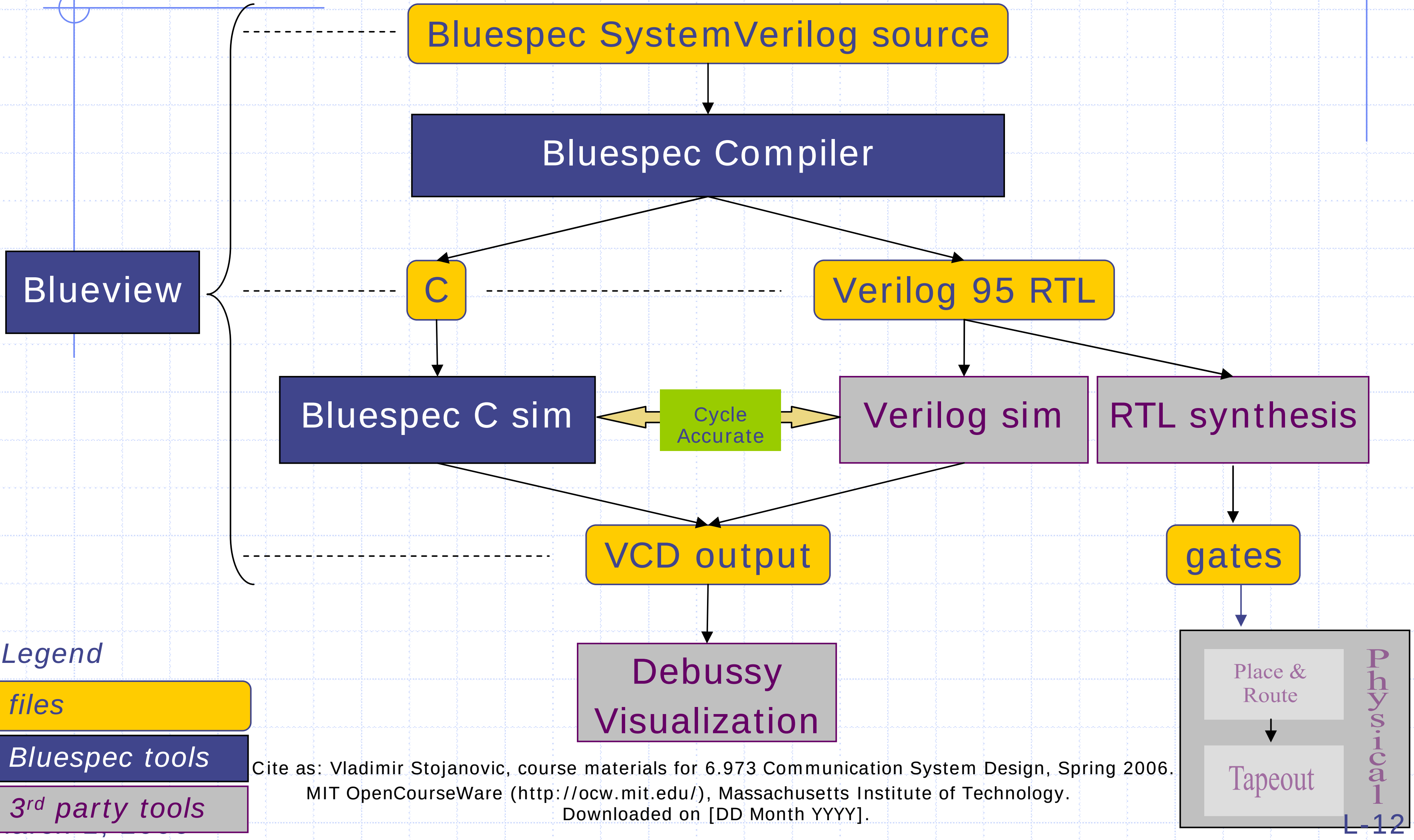


In a GCD call t could be
```
Int#(32),
UInt#(16),
Int#(13), ...
```

```
#(type t)
interface I_GCD;
    method Action start (int a, int b);
    method int result();
endinterface
```
(with `int` replaced by `t` in each method)

- The module can easily be made polymorphic
- Many different implementations can provide the same interface: `module mkGCD (I_GCD)`

Bluespec Tool flow



Legend

files

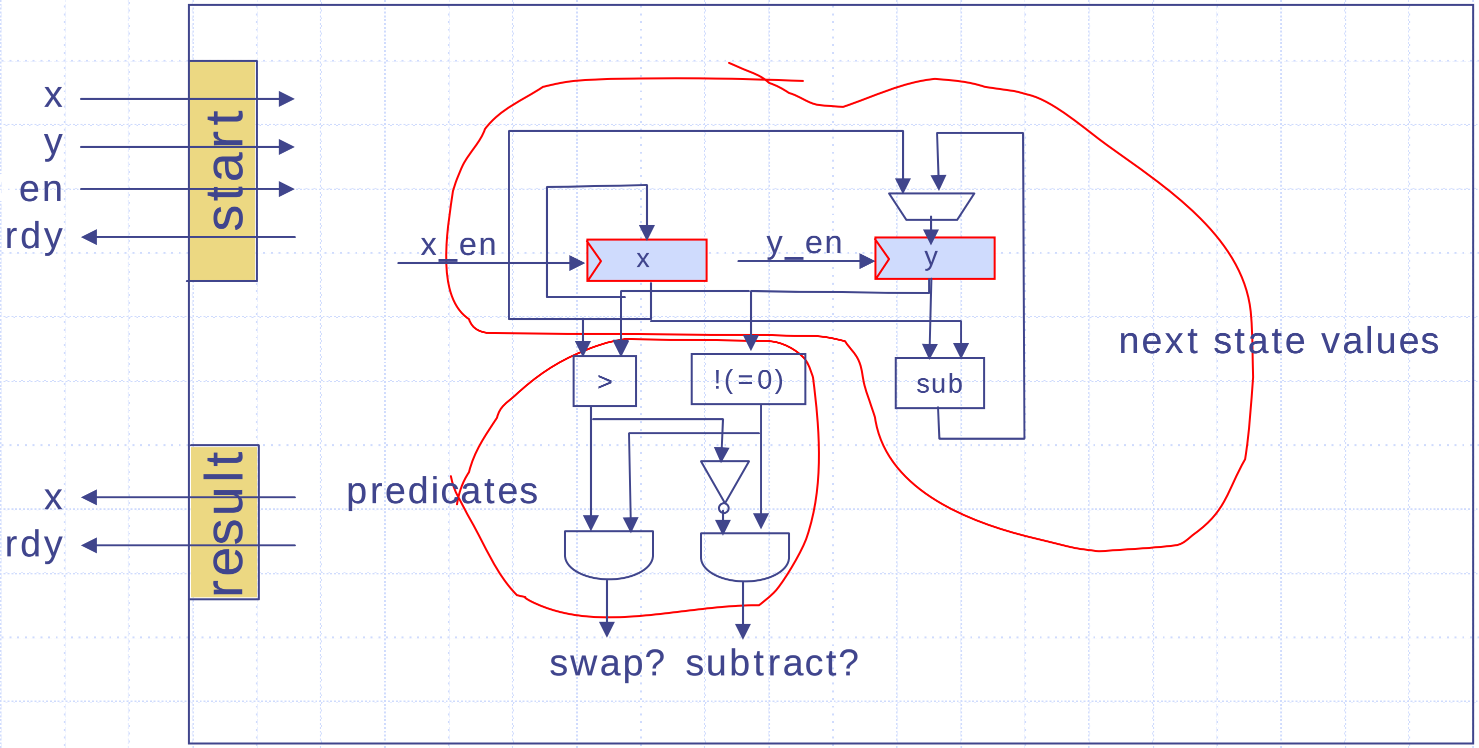
Bluespec tools

3rd party tools

Generated Verilog RTL: GCD

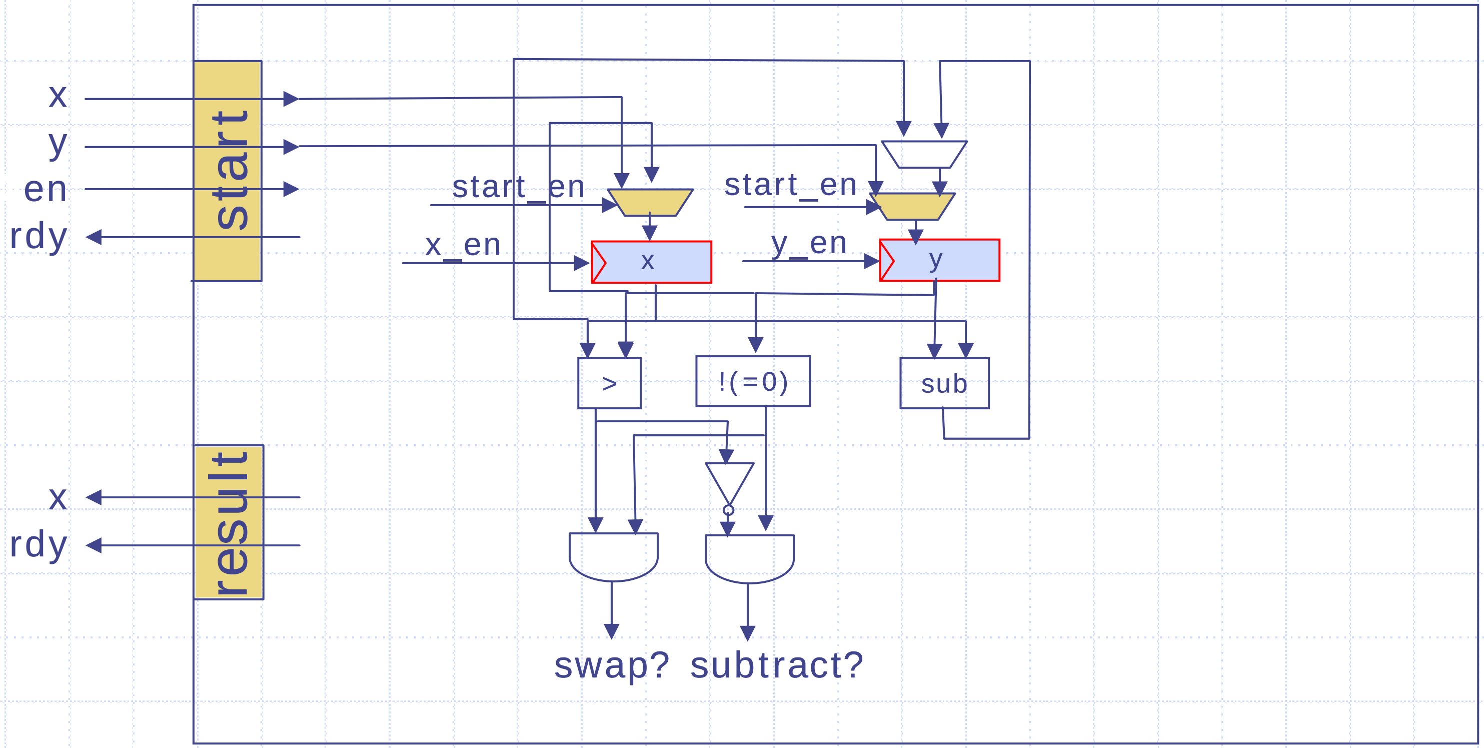
```
module mkGCD(CLK,RST_N,start_a,start_b,EN_start,RDY_start,
             result,RDY_result);
  input  CLK; input  RST_N;
// action method start
  input [31 : 0] start_a; input [31 : 0] start_b; input EN_start;
  output RDY_start;
// value method result
  output [31 : 0] result; output RDY_result;
// register x and y
  reg [31 : 0] x;
  wire [31 : 0] x$D_IN; wire x$EN;
  reg [31 : 0] y;
  wire [31 : 0] y$D_IN; wire y$EN;
...
// rule RL_subtract
  assign WILL_FIRE_RL_subtract = x_SLE_y___d3 && !y_EQ_0___d10 ;
// rule RL_swap
  assign WILL_FIRE_RL_swap = !x_SLE_y___d3 && !y_EQ_0___d10 ;
...
```

Generated Hardware



x_en = swap?
y_en = swap? OR subtract?

Generated Hardware Module



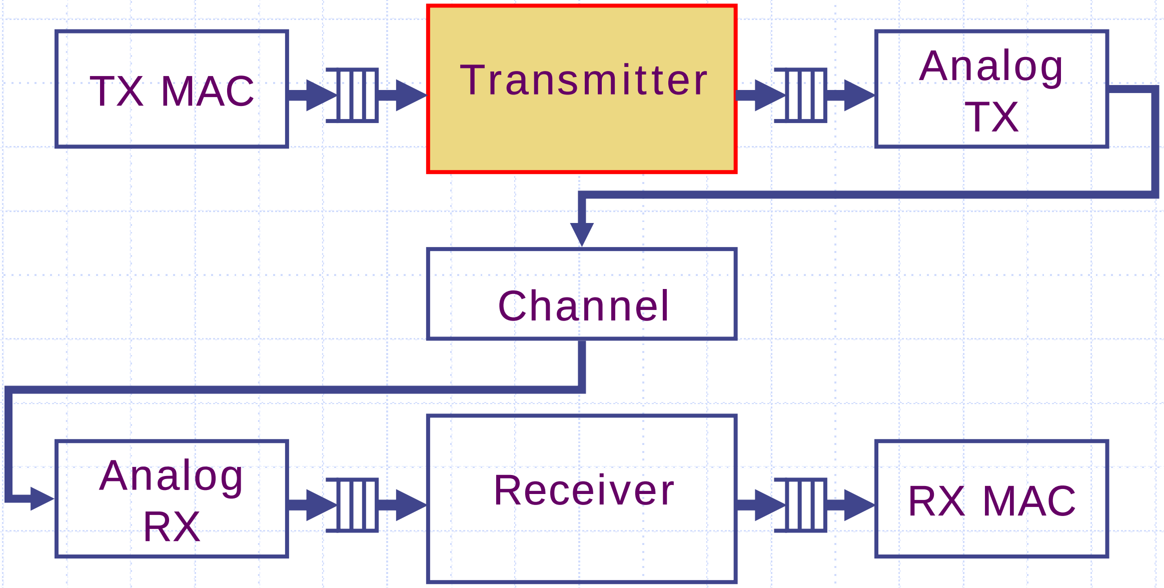
x_en = swap? OR start_en

y_en = swap? OR subtract? OR start_en

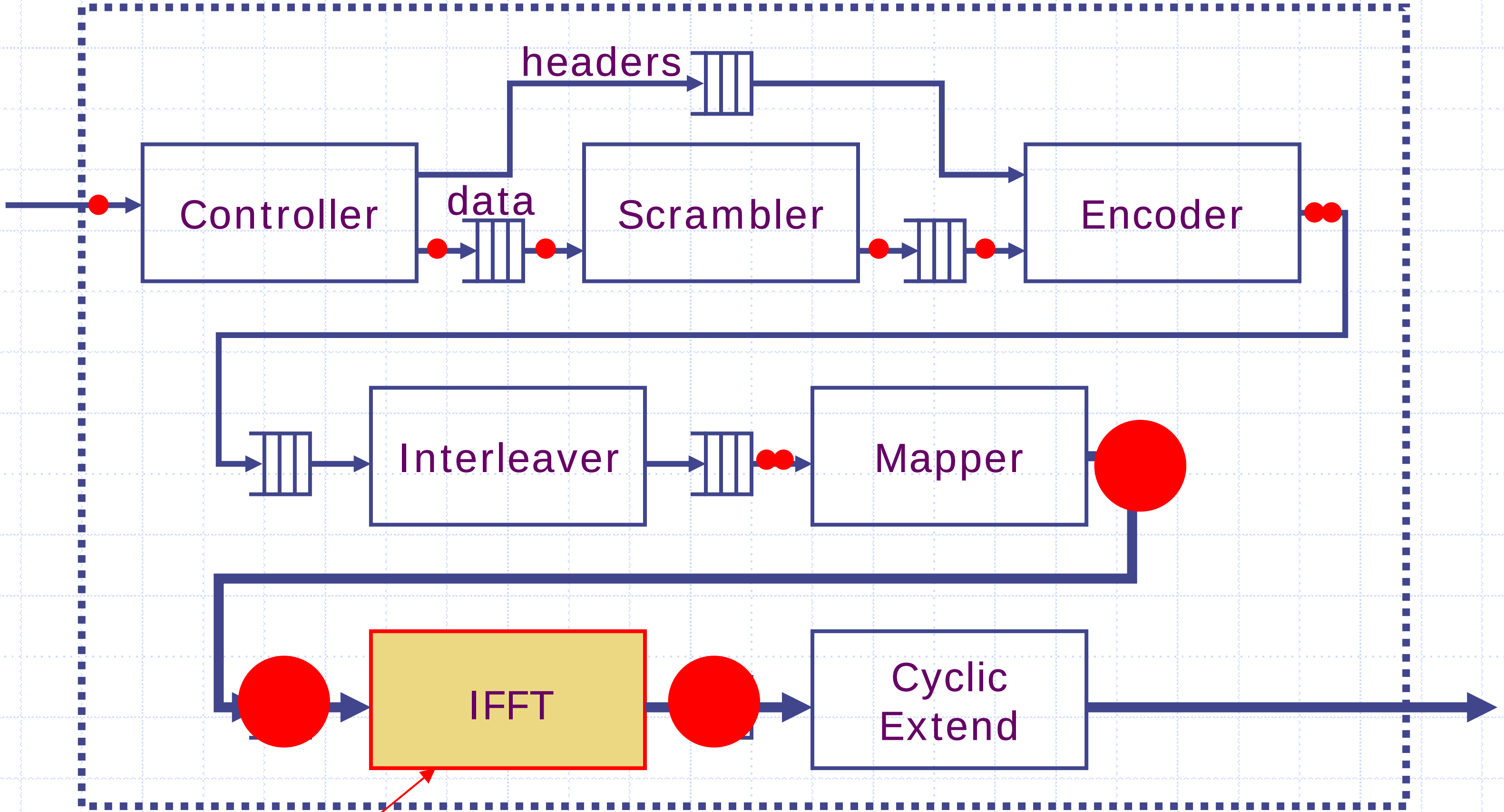
rdy = (y==0)

Design a 802.11a Transmitter

- 802.11a is an IEEE Standard for wireless communication
- Frequency of Operation: 5Ghz band
- Modulation: Orthogonal Frequency Division Multiplexing (OFDM)



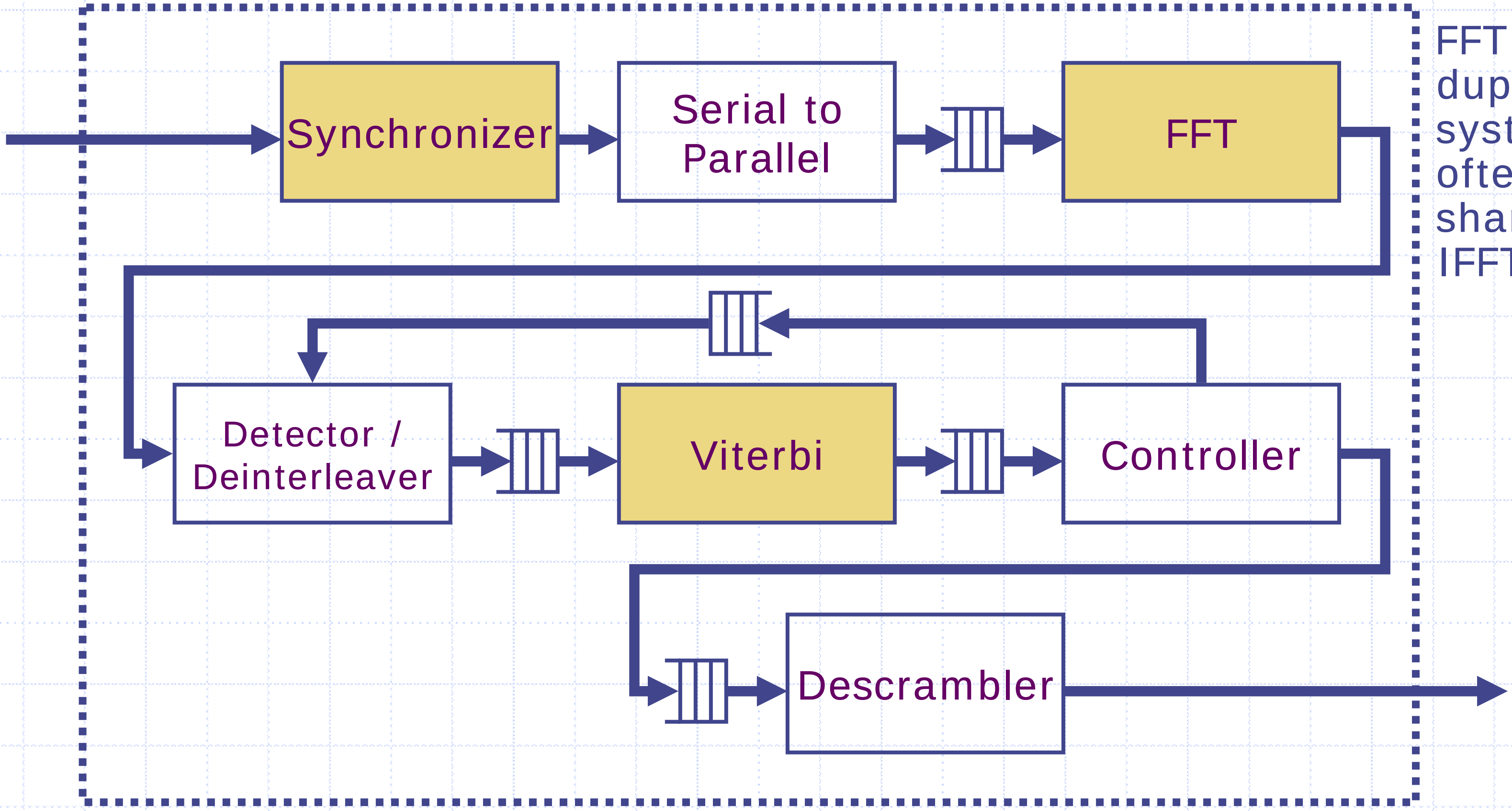
Transmitter Overview



IFFT Transforms 64 (frequency domain) complex numbers into 64 (time domain) complex numbers

🟨 compute intensive

Receiver Overview



FFT, in half duplex system is often shared with IFFT

🟨 compute intensive

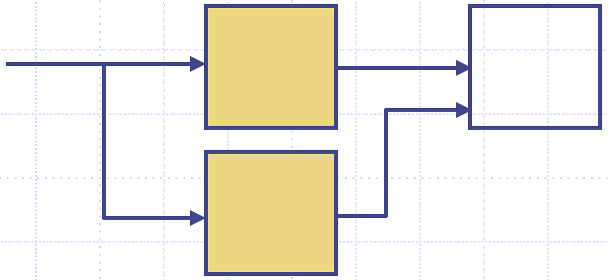
IFFT Requirements

- 802.11a needs to process a symbol in 4 μsec (250KHz)
  - IFFT must output a symbol every 4 μsec
    - i.e. perform an Inverse FFT of 64 complex numbers
  - Each module before IFFT must process every 4 μsec
    - 1 frame for 6Mbps rate
    - 2 frames for 12Mbps rate
    - 4 frames for 24Mbps rate
  - Even in the worst case (24Mbps) the clock frequency can be as low as 1Mhz.

But what about the area & power?

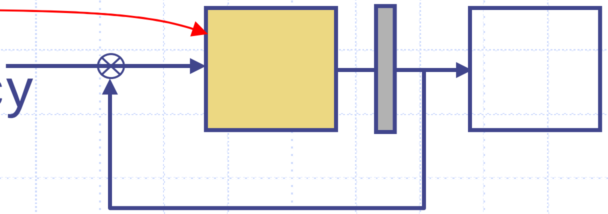
Area-Frequency Tradeoff

We can decrease the area by multiplexing some circuits and running the system at a higher frequency

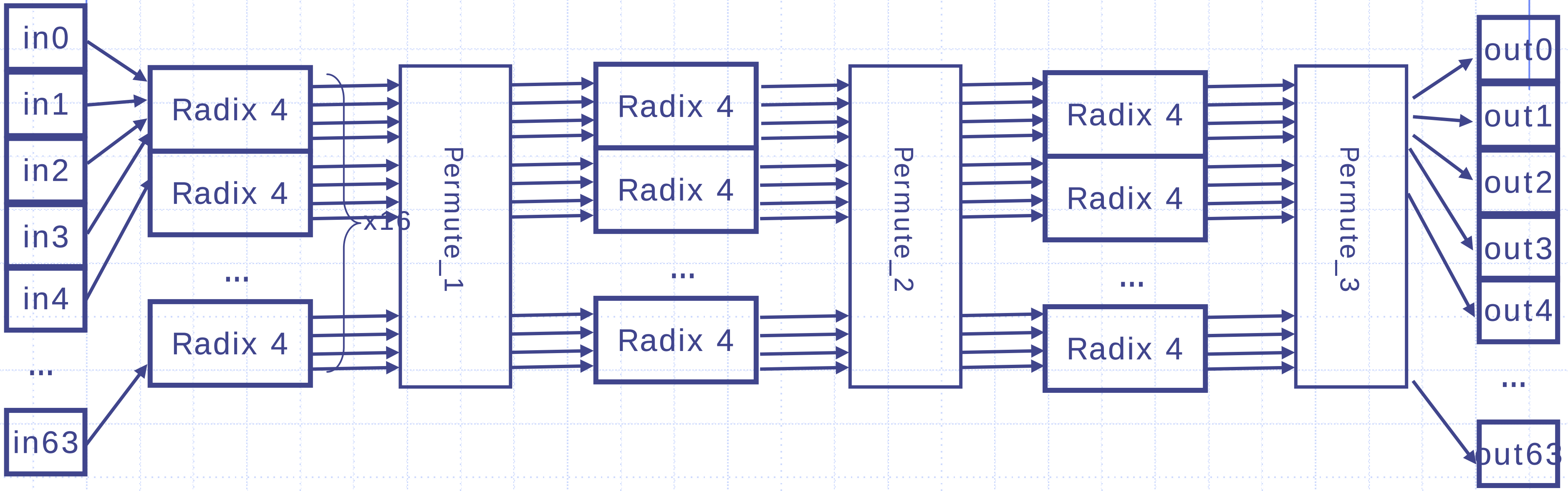


Reuse

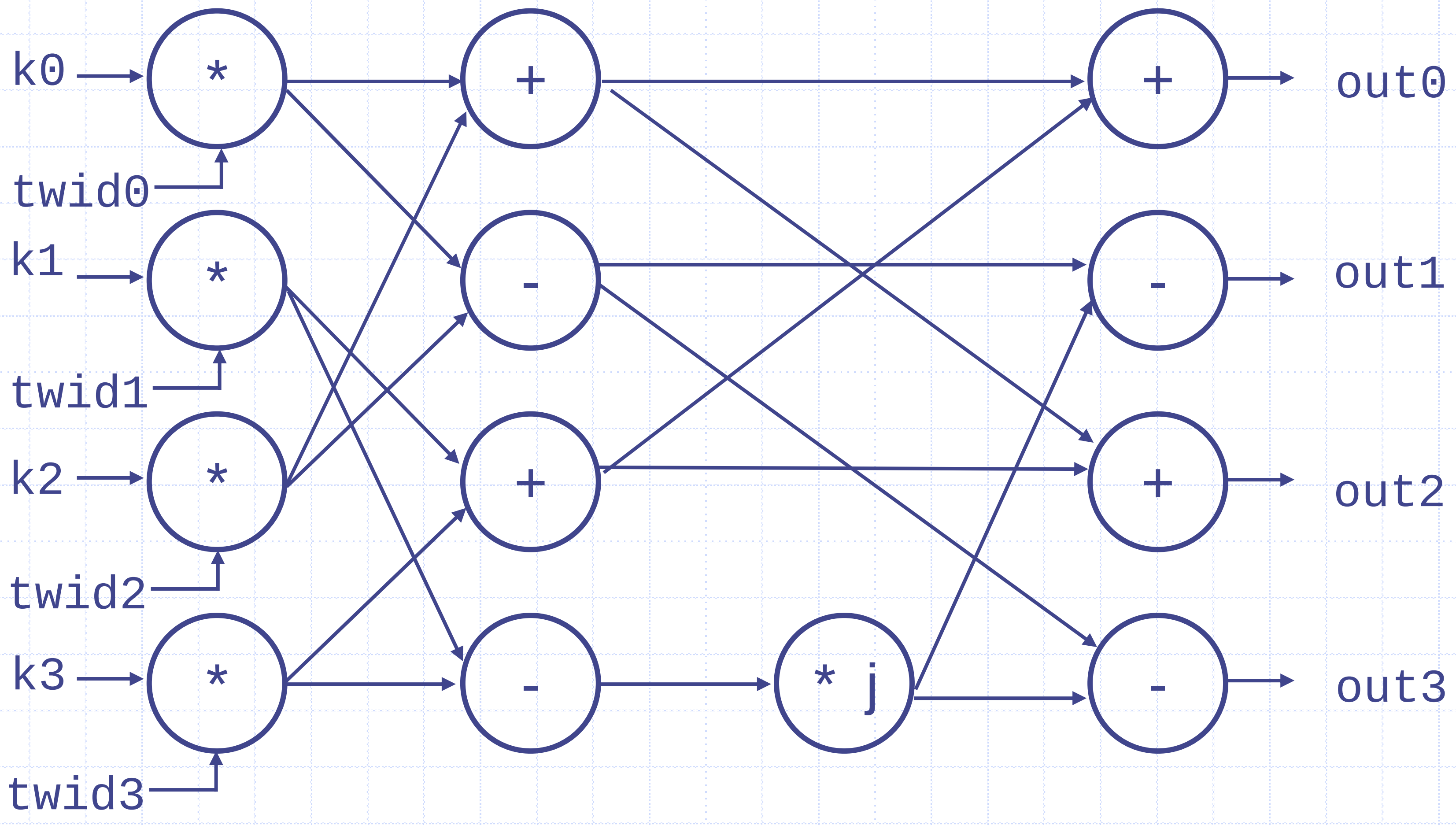
Twice the frequency but half the area



Combinational IFFT



Radix-4 Node



Bluespec code: Radix-4 Node

```
function Tuple4#(Complex, Complex, Complex, Complex)
           radix4(Tuple4#(Complex, Complex, Complex, Complex) twids,
                  Complex k0, Complex k1, Complex k2, Complex k3);

  match {.t0, .t1, .t2, .t3} = twids;
  Complex m0 = k0 * t0; Complex m1 = k1 * t1;
  Complex m2 = k2 * t2; Complex m3 = k3 * t3;

  Complex y0 = m0 + m2; Complex y1 = m0 - m2;
  Complex y2 = m1 + m3; Complex y3 = m1 - m3;

  Complex y3_j = Complex {i: negate(y3.q), q: y3.i};

  Complex z0 = y0 + y2; Complex z1 = y1 - y3_j;
  Complex z2 = y0 - y2; Complex z3 = y1 - y3_j;

  return tuple4(z0, z1, z2, z3);

endfunction
```

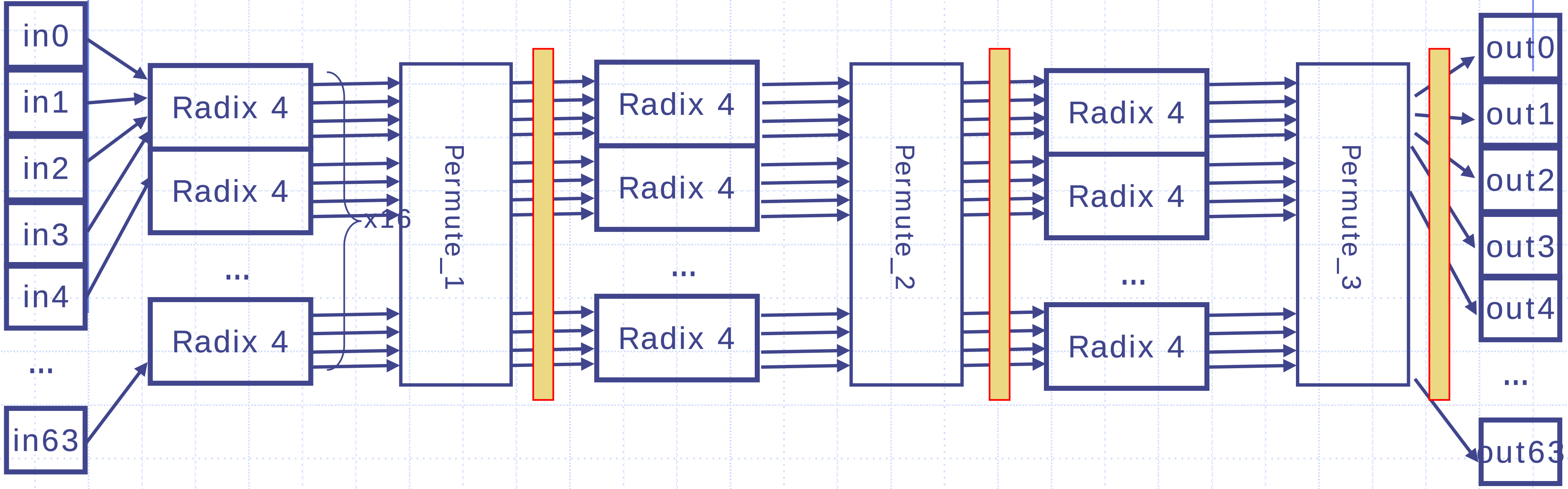
Bluespec code for pure Combinational Circuit

```
function SVector#(64, Complex) ifft (SVector#(64, Complex) in_data);
//Declare vectors
    SVector#(64, Complex) stage12_data = newSVector();
    SVector#(64, Complex) stage12_permuted = newSVector();
    SVector#(64, Complex) stage12_out = newSVector();
    SVector#(64, Complex) stage23_data = newSVector();
    …
//Radix 4 stage 1 (unpermuted)
    for (Integer i = 0; i < 16; i = i + 1)
    begin
      Integer idx = i * 4;
      let twid0 = getTwiddle(0, fromInteger(i));
      match {.y0, .y1, .y2, .y3} = radix4(twid0,
                                 in_data[idx], in_data[idx + 1],
                                 in_data[idx + 2], in_data[idx + 3]);
      stage12_data[idx] = y0;     stage12_data[idx + 1] = y1;
      stage12_data[idx + 2] = y2; stage12_data[idx + 3] = y3;
    end
//Stage 1 permutation
    for (Integer i = 0; i < 64; i = i + 1)
      stage12_permuted[i] = stage12_data[permute_1to2[i]];
//Continued on next slide…
```

Bluespec code for pure Combinational Circuit *continued*

```
// (* continued from previous *)
    stage12_out = stage12_permuted; //Later implementations will change this
//Radix 4 stage 2 (unpermuted)
    for (Integer i = 0; i < 16; i = i + 1)
    begin
      Integer idx = i * 4;
      let twid1 = getTwiddle(1, fromInteger(i));
      match {.y0, .y1, .y2, .y3} = radix4(twid1,
                                        stage12_out[idx], stage12_out[idx + 1],
                                        stage12_out[idx + 2], stage12_out[idx + 3]);
      stage23_data[idx] = y0;      stage23_data[idx + 1] = y1;
      stage23_data[idx + 2] = y2; stage23_data[idx + 3] = y3;
    end
//Stage 2 permutation
    for (Integer i = 0; i < 64; i = i + 1)
      stage23_permuted[i] = stage23_data[permute64_2to3[i]];
…
//Repeat for Stage 3
…
    return stage3out_permuted;
endfunction
```

Pipelined IFFT



Put a register to hold 64 complex numbers at the output of each stage.

Even more hardware but clock can go faster – less combinational circuitry between two stages

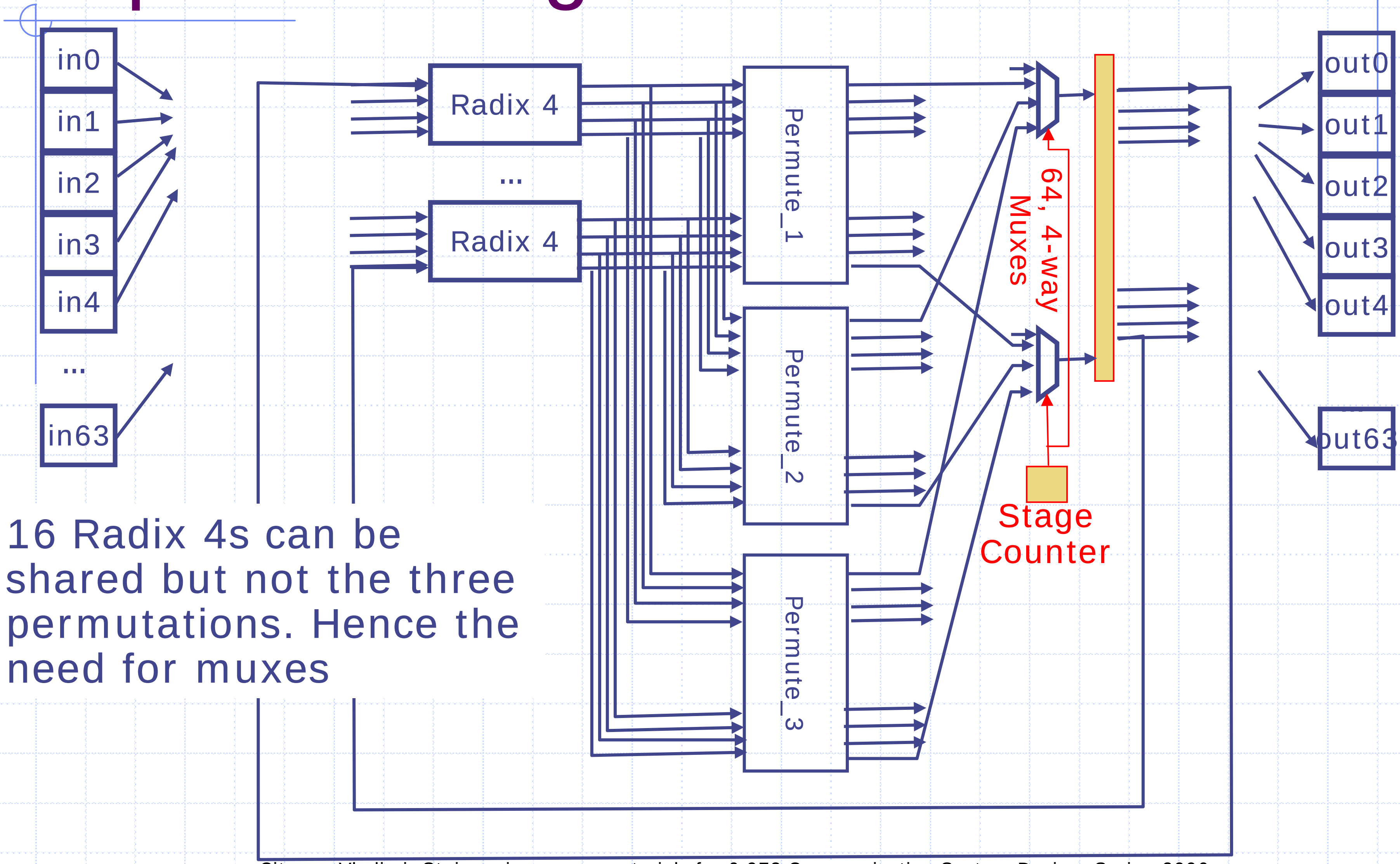
Bluespec code for Pipeline Stage

```
module mkIFFT_Pipelined() (I_IFFT);
//Declare vectors
    SVector#(64, Complex) in_data;
    SVector#(64, Complex) stage12_data = newSVector();
    …
//Declare FIFOs
    FIFO#(SVector#(64, Complex)) in_fifo <- mkFIFO();
//Declare pipeline registers
    Reg#(SVector#(64, Complex)) stage12_reg <- mkReg(newSVector());
    Reg#(SVector#(64, Complex)) stage23_reg <- mkReg(newSVector());
//Read input
  in_data = in_fifo.first();
//Radix 4 stage 1 (unpermuted)
    for (Integer i = 0; i < 16; i = i + 1)
    begin
      Integer idx = i * 4;
      let twid0 = getTwiddle(0, fromInteger(i));
     match {.y0, .y1, .y2, .y3} = radix4(twid0,
                                  in_data[idx], in_data[idx + 1],
//Continue as before…
```

Bluespec code for Pipeline Stage

```
…
//Read from pipe register for stage 2
  stage12_out = stage12_reg;

//Radix 4 stage 2 (unpermuted)
  for (Integer i = 0; i < 16; i = i + 1)
…

//Read from pipe register for stage 3
  stage23_out = stage23_reg;

  rule writeRegs (True);
      stage12_reg <= stage12_permuted;
      stage23_reg <= stage23_permuted;
      in_fifo.deq(); out_fifo.enq(stage3out_permuted);
  endrule

  method Action inp (Vector#(64, Complex) data);
    in_fifo.enq(data);
  endmethod
…
endmodule
```

Circular pipeline: Reusing the Pipeline Stage



16 Radix 4s can be shared but not the three permutations. Hence the need for muxes

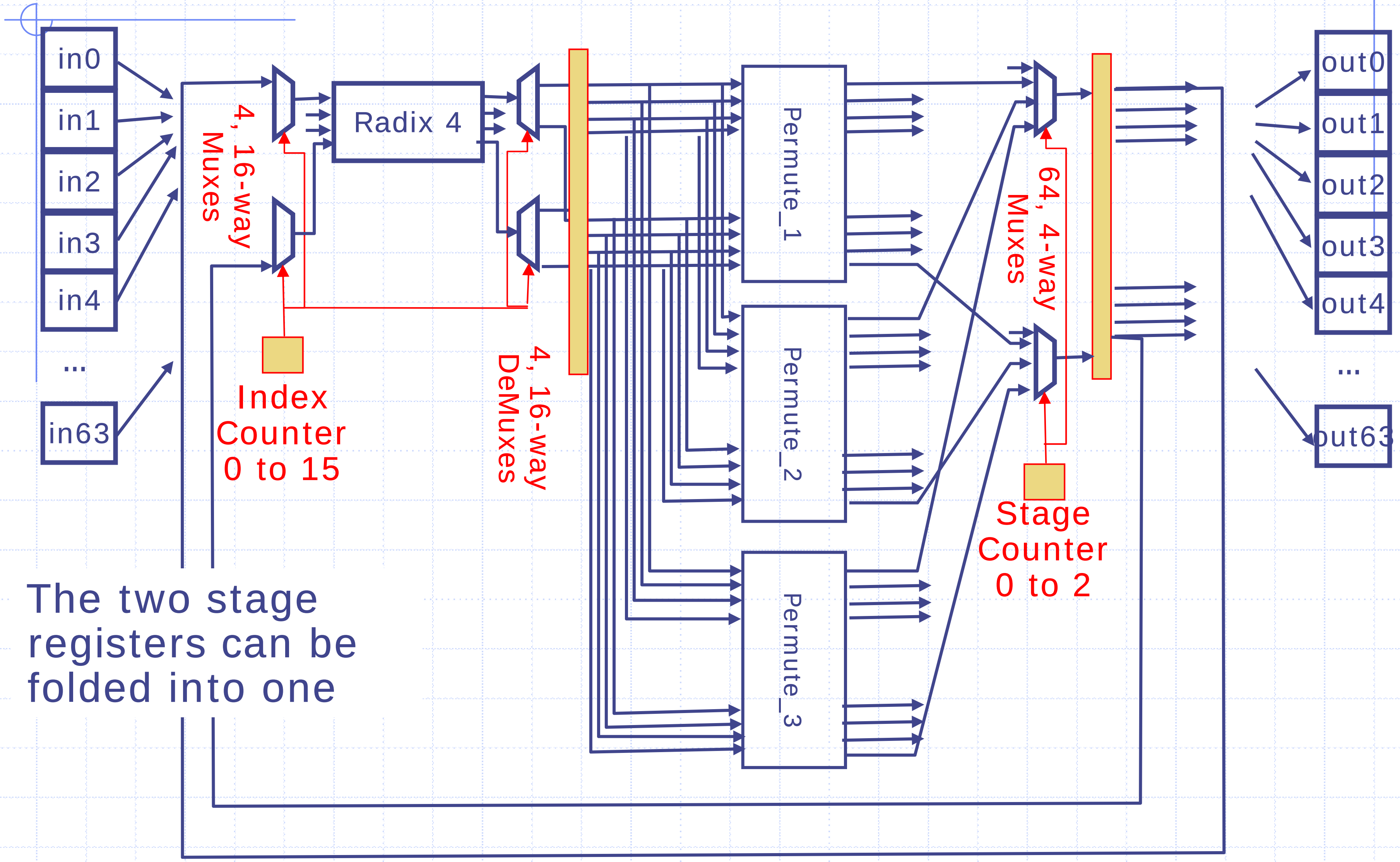
Bluespec Code for Circular Pipeline

```
module mkIFFT_Circular (I_IFFT);
    SVector#(64, Complex) in_data = newSVector();
    SVector#(64, Complex) stage_data = newSVector();
    SVector#(64, Complex) stage_permuted = newSVector();
//State elements
    Reg#(SVector#(64, Complex)) data_reg <- mkReg(newSVector());
    Reg#(Bit#(2)) stage_counter <- mkReg(0);
    FIFO#(SVector#(64, Complex)) in_fifo <- mkFIFO();
//Read input
    in_data = data_reg;
//Perform a single Radix 4 stage (unpermuted)
    for (Integer i = 0; i < 16; i = i + 1)
    begin
      Integer idx = i * 4;
      let twid = getTwiddle(stage_counter, fromInteger(i));
      match {.y0, .y1, .y2, .y3} = radix4(twid,
                                         in_data[idx], in_data[idx + 1],
                                         in_data[idx + 2], in_data[idx + 3]);
      stage_data[idx] = y0;     stage_data[idx + 1] = y1;
      stage_data[idx + 2] = y2; stage_data[idx + 3] = y3;
    end
//Continued…
```

Bluespec Code for Circular Pipeline

```
//Stage permutation
    for (Integer i = 0; i < 64; i = i + 1)
      stage_permuted[i] = case (stage_counter)
                            0: return in_wire._read[i];
                            1: return stage_data[permute64_1to2[i]];
                            2: return stage_data[permute64_2to3[i]];
                            3: return stage_data[permute64_3toOut[i]];
                          endcase;

  rule writeRegs (True);
      data_reg <= stage_permuted;
      stage_counter <= stage_counter + 1;
  endrule

  method Action inp(SVector#(64, Complex) data) if (stage_counter == 0);
    in_fifo.enq(data);
    stage_counter <= 1;
  endmethod
…
endmodule
```

Just one Radix-4 node!



The two stage registers can be folded into one

Bluespec Code for Extreme reuse

```
module mkIFFT_SuperCircular (I_IFFT);
  SVector#(64, Complex) in_data = newSVector();
  SVector#(64, Complex) stage_data = newSVector();
  SVector#(64, Complex) permutedV  = newSVector();
//State
  Reg#(SVector#(64, Complex)) data_reg <- mkReg(newSVector());
  Reg#(SVector#(64, Complex)) post_reg <- mkReg(newSVector());
  Reg#(Bit#(2)) stage_counter <- mkReg(0);
  Reg#(Bit#(5)) idx_counter <- mkReg(16);
  FIFO#(SVector#(64, Complex)) in_fifo <- mkFIFO();
//Read input
  in_data = data_reg;
//Do one-sixteenth of a Radix 4 stage (unpermuted)
  Bit#(6) idx = {idx_counter, 2'b00};    //idx = idx_counter * 4
//Use DYNAMIC select and update of the Vector
  let twid = getTwiddle(stage_counter, idx_counter);
  match {.y0, .y1, .y2, .y3} = radix4(twid,
                                    select(in_data, idx), select(in_data, idx + 1),
                                    select(in_data, idx + 2), select(in_data, idx + 3));
//generates post_reg after writing in the 4 new values
  let stage_data0 = post_reg;
  let stage_data1 = update(stage_data, idx, y0);
  let stage_data2 = update(stage_data1,idx + 1, y1);
  let stage_data3 = update(stage_data2,idx + 2, y2);
  stage_data      = update(stage_data3,idx + 3, y3);
//Continued…
```

Bluespec Code for Extreme reuse-2

```
//Permutation is based on the current stage
    for (Integer i = 0; i < 64; i = i + 1)
      permutedV[i] = case (stage_counter)
                       1: return post_reg[permute64_1to2[i]];
                       2: return post_reg[permute64_2to3[i]]
                       3: return
   post_reg[permute64_3toOut[i]];
                       default: return in_fifo.first()[i];
                     endcase;

  rule writeRegs (stage_counter != 0);
      post_reg <= stage_data;
      if (idx == 16)
        data_reg <= permutedV;
      idx_counter <= (idx_counter == 16) ? 0: idx_counter + 1;
      if (idx_counter == 16)
        stage_counter <= stage_counter + 1;
  endrule
//Everything else as before…
```

Synthesis results

Nirav Dave & Mike Pellauer

Design	Area (mm^2)	CLK Period	Throughput (1 symbol)	Latency
Comb.	1.03	15 ns	15ns	15 ns
Pipelined	1.46	7 ns	7 ns	21 ns
Circular	0.83	8 ns	24 ns	24 ns
1 Radix	0.23	8 ns	408 ns	408 ns

TSMC .13 micron; numbers reported are before place and route.

Single radix-4 node design is ¼ the size of combination design but still meets the throughput requirement easily; clock can reduced to 15 to 20 Mhz

Synthesis results

Steve Gerding, Elizabeth Basha & Rose Liu

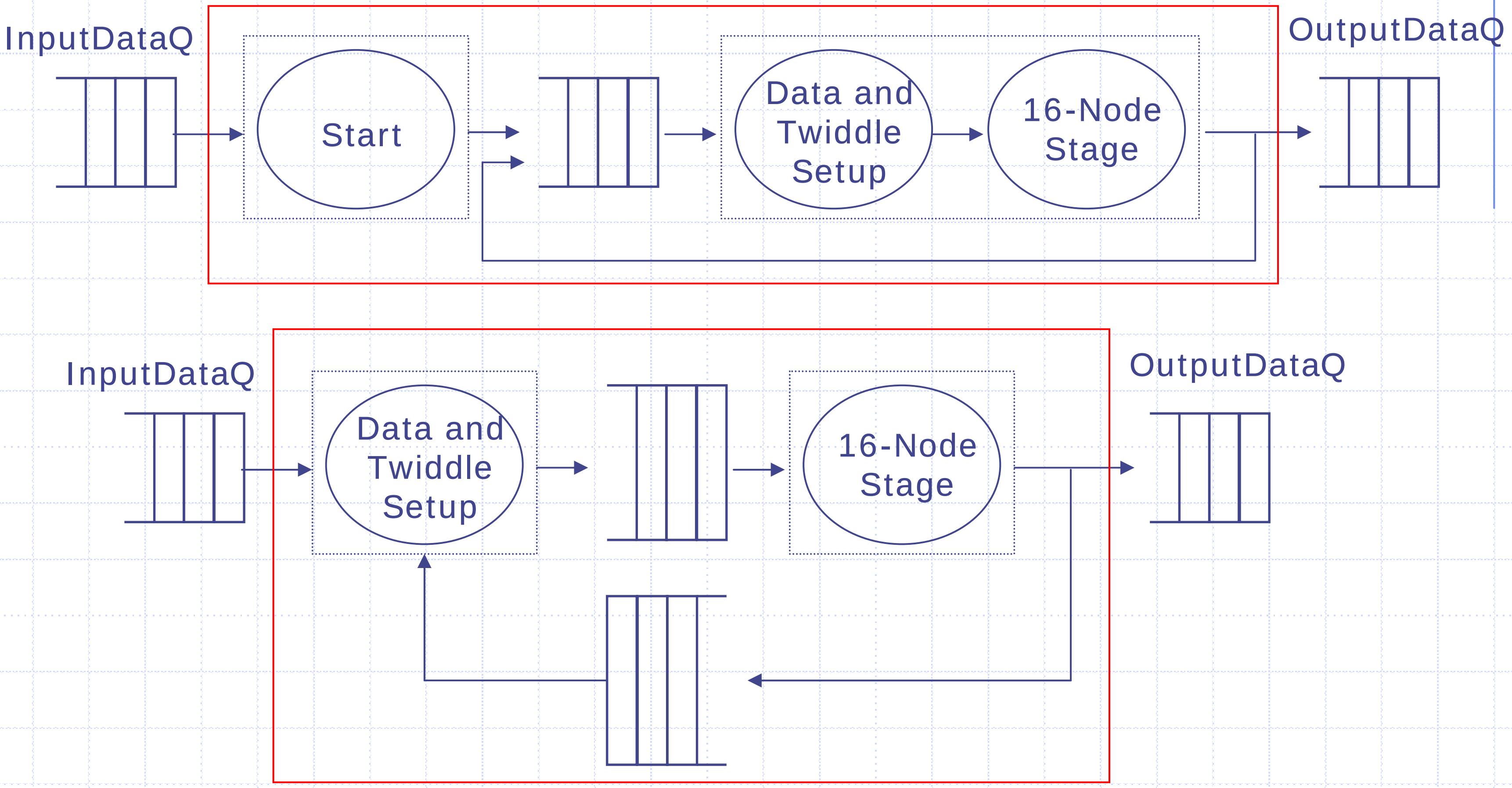
Design	Area (mm^2)	CLK Period	Throughput (1 symbol)	Latency
Comb.	29.12	63 ns	63 ns	63 ns
Circular-2stages	5.19	30 ns	90 ns	180 ns
Circular	4.57	33 ns	99 ns	99 ns

TSMC .13 micron; numbers reported are after place and route

Two stage circular pipeline design is not good.

Circular pipeline design can meet the throughput requirement at 750KHz!

Two circular pipelines



Steve Gerding, Elizabeth Basha & Rose Liu