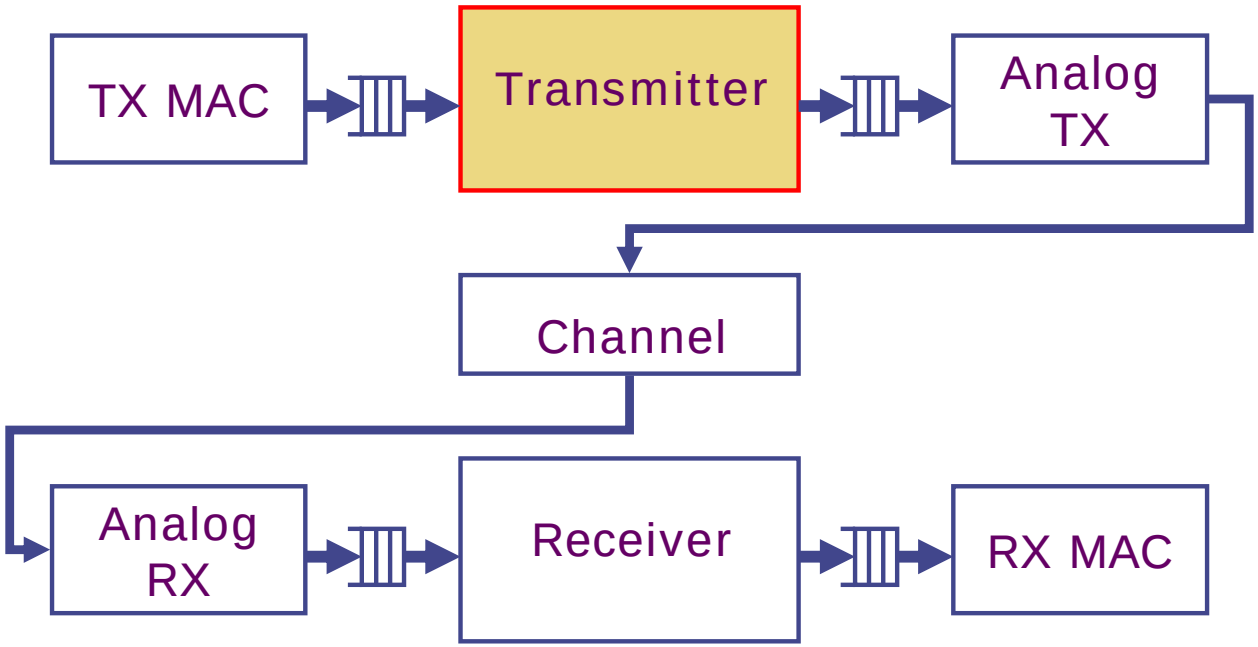
Bluespec-3: Architecture exploration using static elaboration

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

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)



Nomenclature

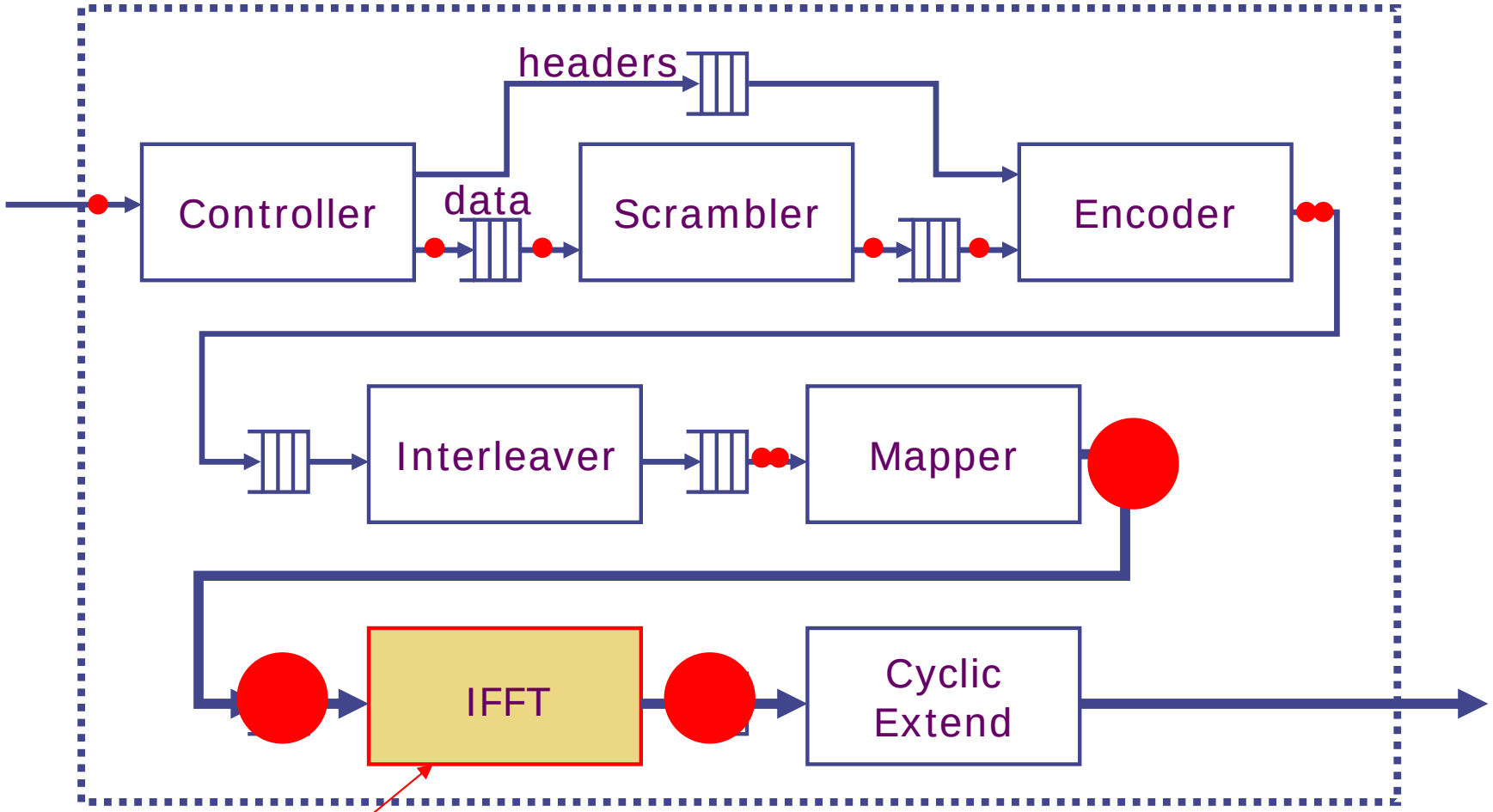
- Base data unit of the system: 24 uncoded bits
- Sample – One complex baseband value
- Symbol – One OFDM symbol that will be transmitted
  - In time domain: 64 Samples long
  - In frequency domain: 64 Tones (48 data, 4 pilot, 12 unused)
  - Represented in fixed point (16 bit real, 16 bit imag)
- Frame - A unit of data, corresponds to:
  - 1 Symbol at 6 Mbps (i.e. 1 frame represents one symbol)
  - ½ Symbol at 12 Mbps (i.e. 2 frames represent one symbol)
  - ¼ Symbol at 24 Mbps (i.e. 4 frames represent one symbol)
- Message – A sequence of data Symbols preceded by a header Symbol (SIGNAL)

Need Fixed Point Arithmetic

- Floating point is too inefficient to use
- We need to represent fractional values between -1 and 1 in our system
- Fixed Point: use a 16 bit integer to represent each value
  - Store the value multiplied by 2^{15} (32,768)
  - Use 2's compliment arithmetic on fixed point values, but watch for overflow
  - MSB indicates sign of number (1 for negative)
- Examples:

  -1.0 => 0x8000 (-32768)
  1/√2 => 0x5a82 (23170)
  -3/√10 => 0x8692 (-31086)

Transmitter Overview



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

compute intensive

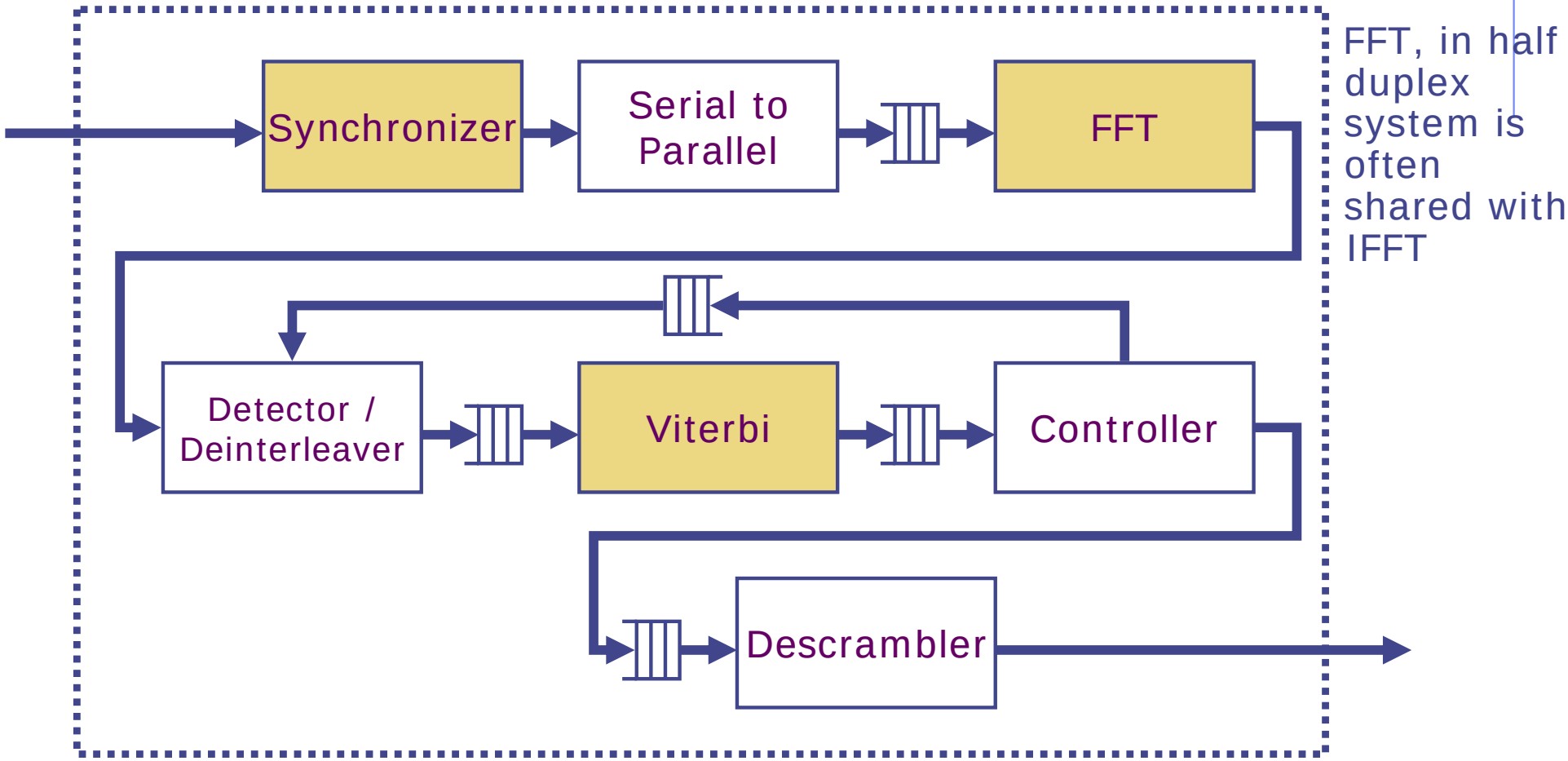
Mapper

- Maps incoming data to tones based on rate
- Outputs 1 OFDM symbol to the IFFT
- Depending on the rate, 48, 96, or 192 bits of input may be required to fill one symbol.

Input: [rate (2), data (48)]

Output: [data (64 complex numbers)]

Receiver Overview



Synchronizer

- Performs two important tasks:
  - Timing estimation and synchronization
    - Decides when a new message is present
    - Tells rest of receiver at which sample the incoming symbol starts
  - Frequency offset estimation and correction
    - Estimates the offset of the transmitter and receiver clocks
    - Rotates input data to correct for this offset

Extremely complicated !

Viterbi Decoder

- Uses the Viterbi algorithm to decode convolutionally encoded symbols
- Requires three 48-bit inputs to perform sufficient traceback
  - Will only output a frame after it receives the two subsequent frames
  - Detector flushes the Viterbi module with zeros after header and end of message

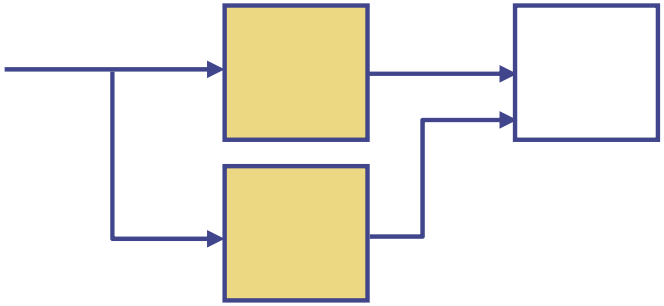
IFFT Requirements

- 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?

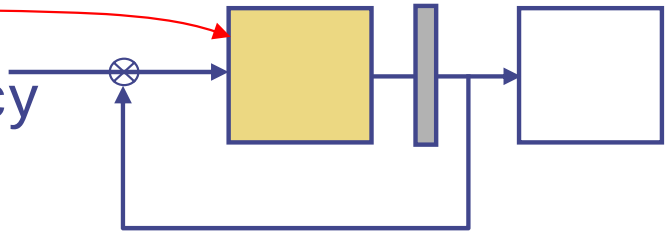
Area-Frequency Tradeoff

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

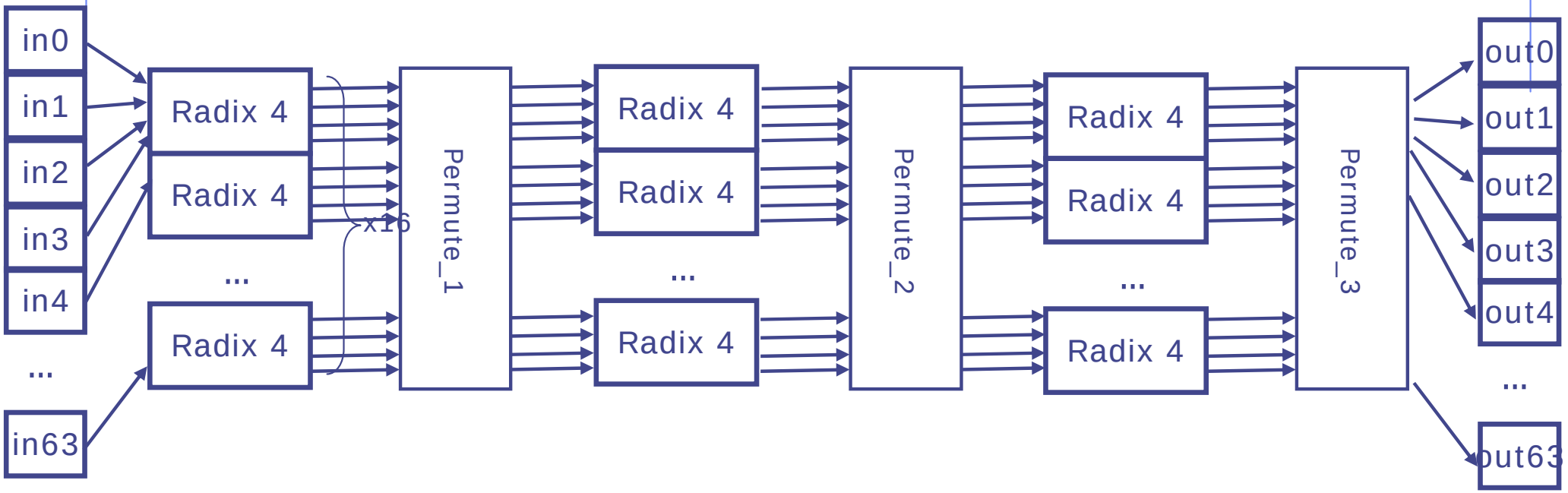


Reuse

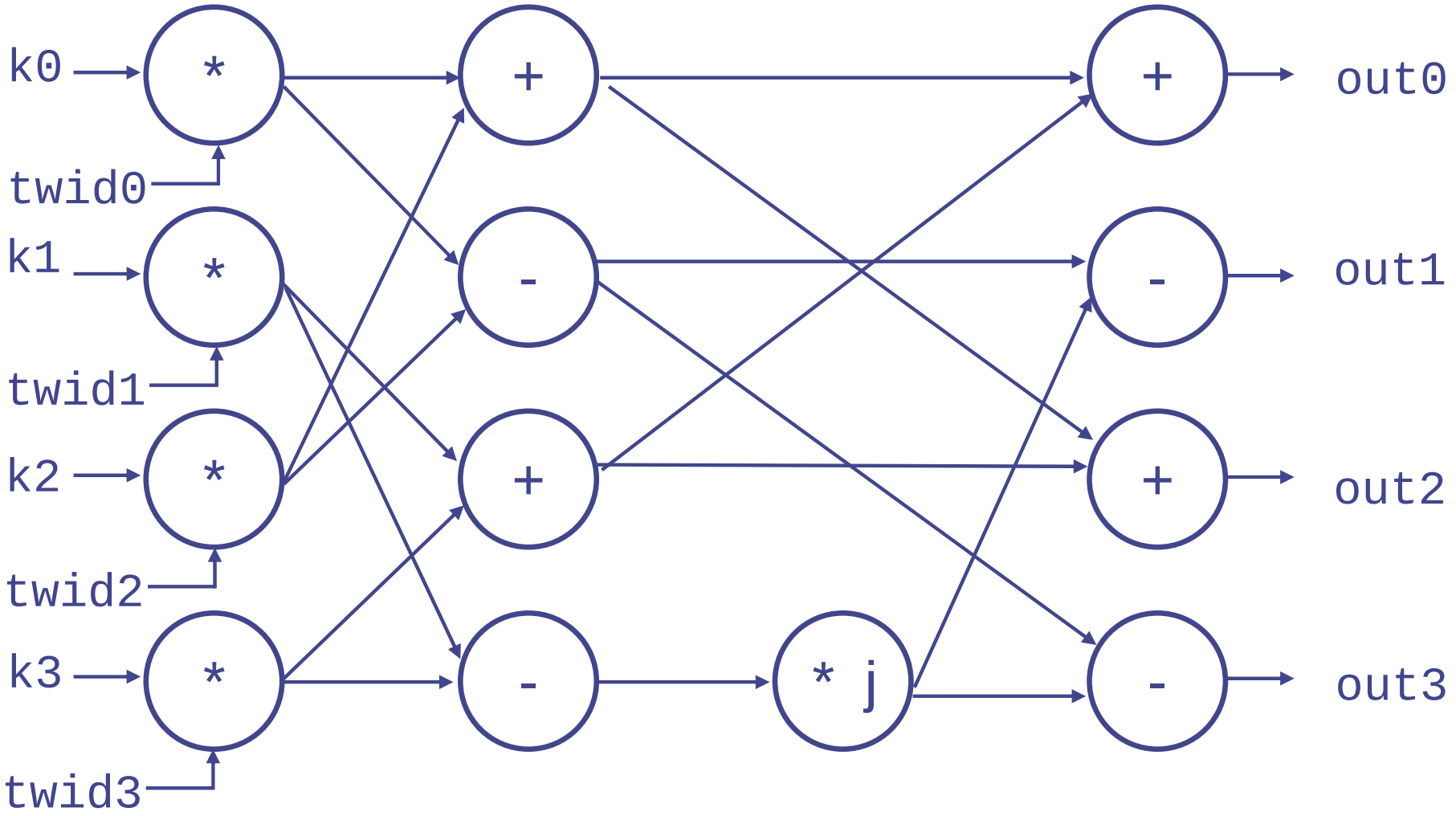
Twice the frequency but half the area



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
```

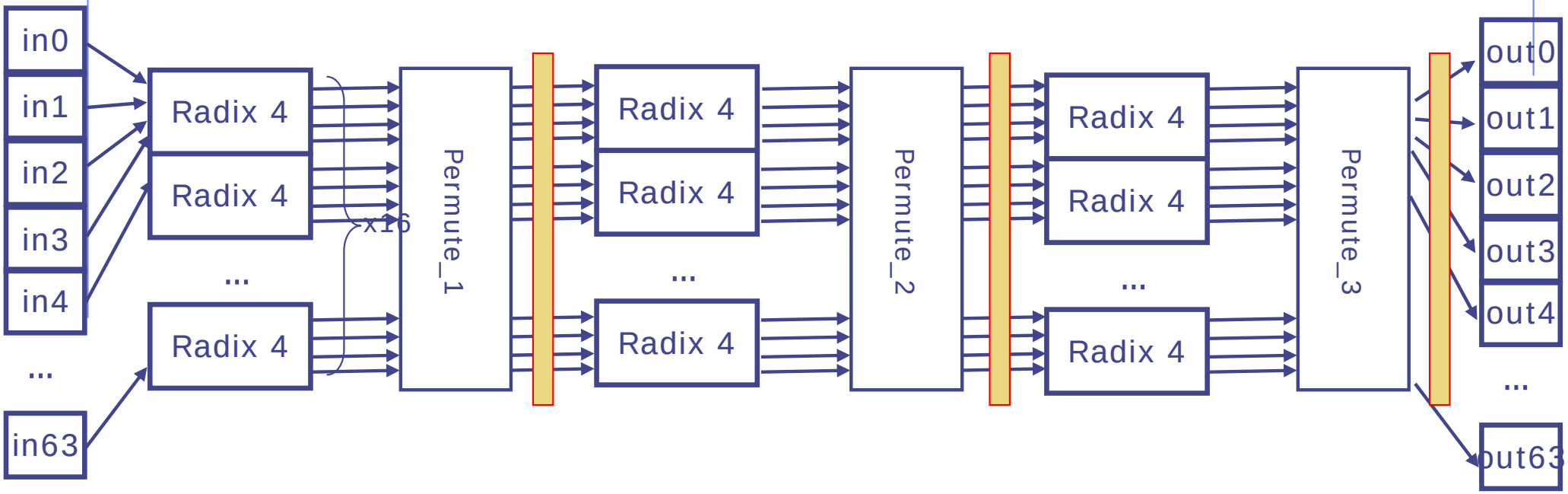
Bluespec code for pure Combinational Circuit

```
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

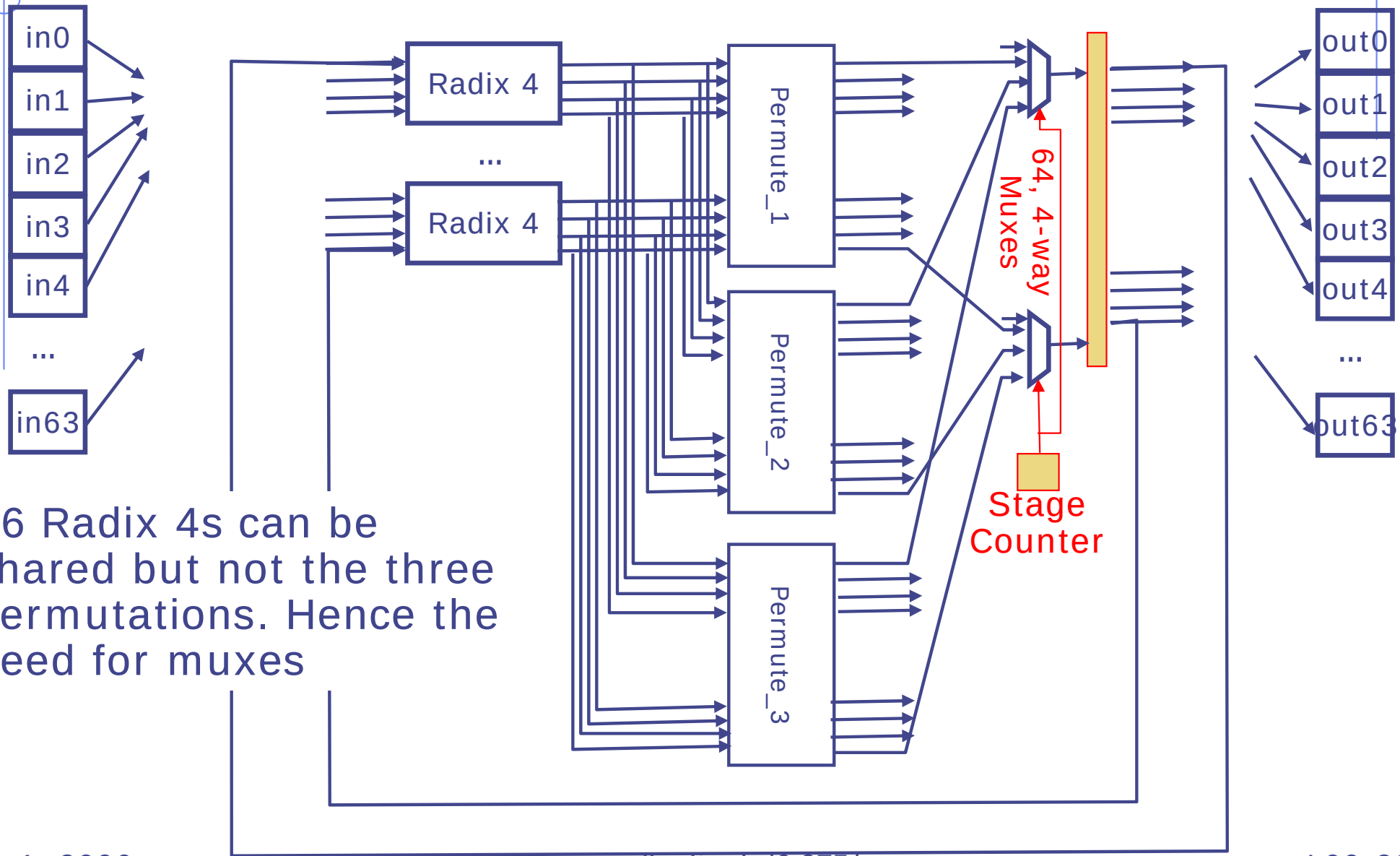
Bluespec code for Pipeline Stage

```
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

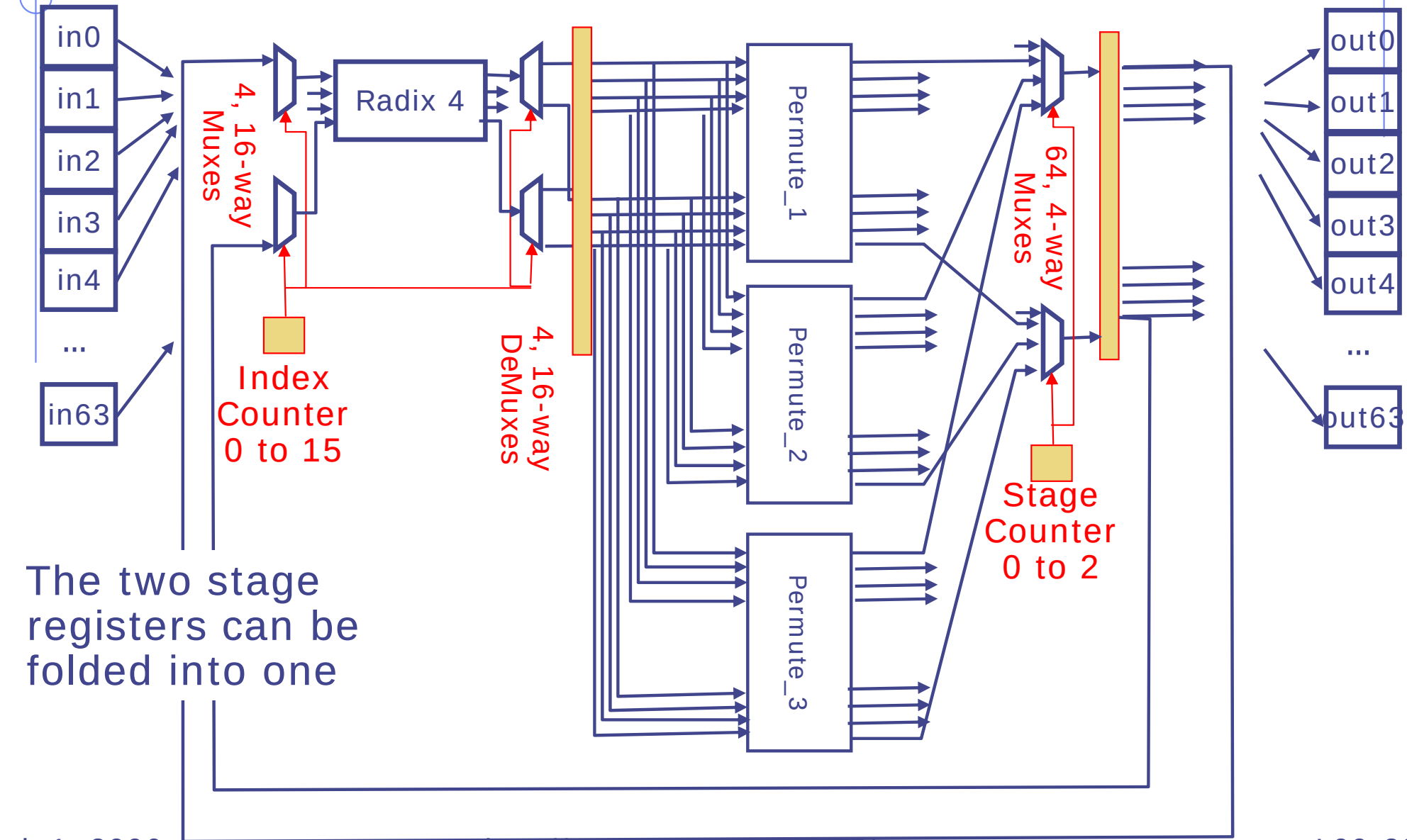
Bluespec Code for Circular Pipeline

```
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)) new_post_reg = 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);//Stage Counter =0 => no value
  Reg#(Bit#(5)) idx_counter <- mkReg(16); //Idx_Counter =16 => permute
  FIFO#(SVector#(64, Complex)) in_fifo <- mkFIFO();

  let twid = getTwiddle(stage_counter, idx_counter);
  match {.y0, .y1, .y2, .y3} =
     radix4(twid, select(in_data, {idx_counter,2'b00}),
                  select(in_data, {idx_counter,2'b01}),
                  select(in_data, {idx_counter,2'b01}),
                  select(in_data, {idx_counter,2'b10}));
//Permutation takes post_reg's values back to data_reg
  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;
```

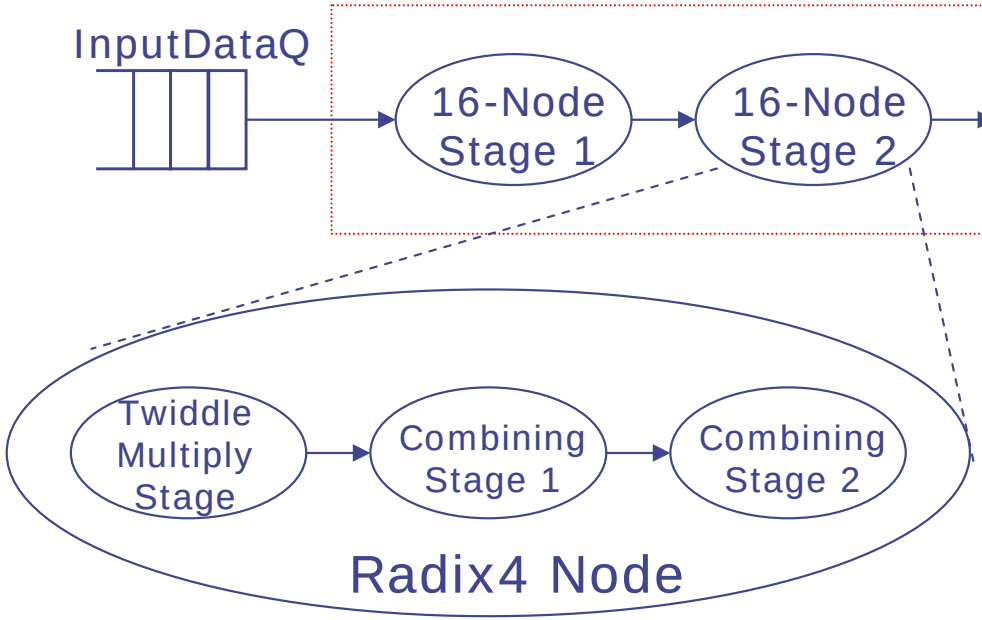
Bluespec Code for Extreme Reuse-2

```
rule doRadix(stage_counter != 0);
   if (idx_counter < 16) //We need to calc new radix values
    begin
         //generates new_post_reg value: 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);
         new_post_reg    = update(stage_data3,idx + 3, y3);
         post_reg <= new_post_reg;
    end
   else //(idx_counter == 16) We need to permute
      begin
        data_reg <= premutedV;
      end

   //We always increment counters
   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

- Did not have time to synthesize these various designs
- But we have results from a term project from last year
  - Steve Gerding, Elizabeth Basha & Rose Liu

IFFT Initial Design

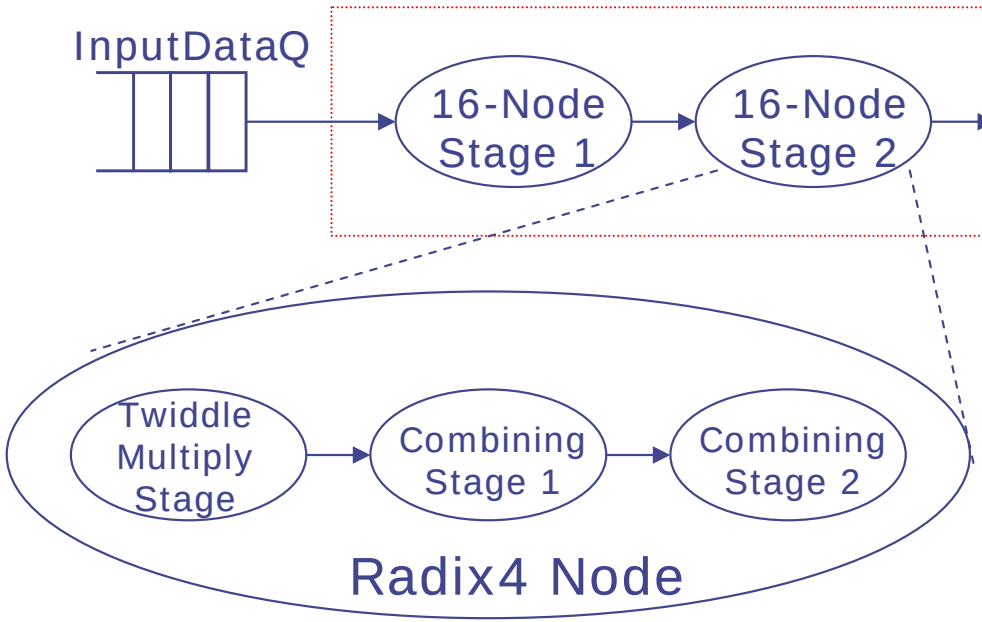


Radix4 Nodes	*	+
1	16	24
48	768	1152

- Area = 29.12μm^2
- Cycle Time = 63.18ns
- Throughput = 1 Symbol / 63.18ns

Steve Gerding, Elizabeth Basha & Rose Liu

IFFT Initial Design

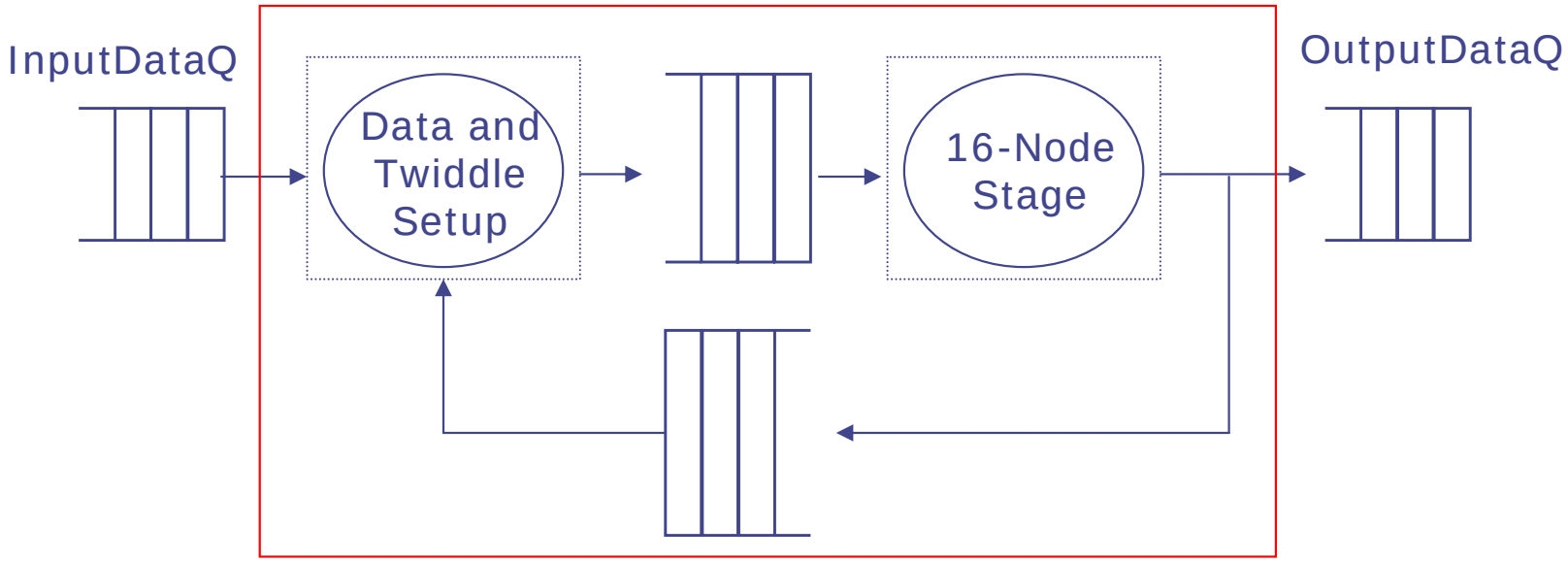


Radix4 Nodes	*	+
1	16	24
48	768	1152

- Area = 29.12μm^2
- Cycle Time = 63.18ns
- Throughput = 1 Symbol / 63.18ns

Steve Gerding, Elizabeth Basha & Rose Liu

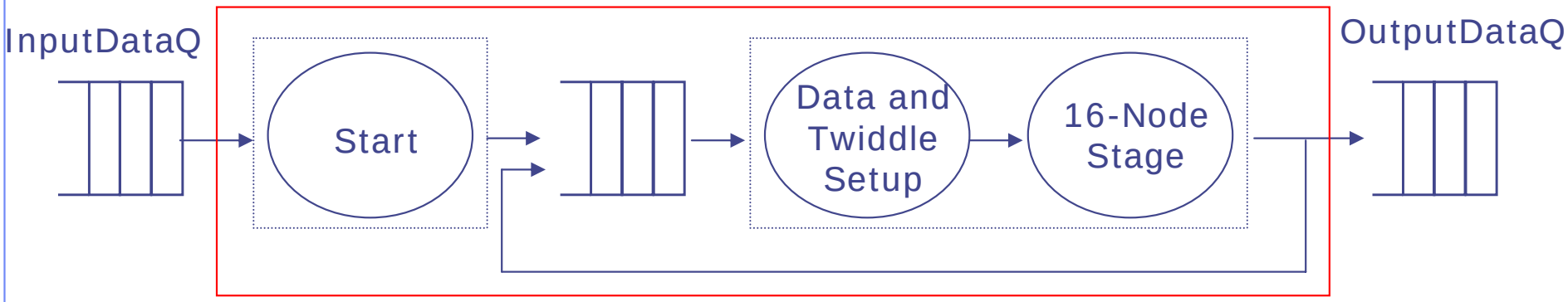
IFFT Design Exploration 1



- Area = 5.19μm^2
- Cycle Time = 30.50ns
- Throughput = 1 Symbol / 3 x 30.50ns
  = 1 Symbol / 91.50ns

Steve Gerding, Elizabeth Basha & Rose Liu

IFFT Design Exploration 2



- Area = 4.57mm^2
- Cycle Time = 32.89ns
- Throughput = 1 symbol / 3x 32.89ns
  = 1 symbol / 98.67ns