Hardware Queues

Eylon Caspi
University of California, Berkeley
eylon@cs.berkeley.edu

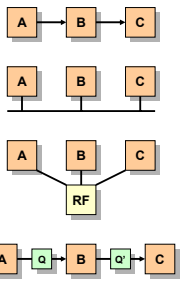
4/8/05

Outline

- Why Queues?
- Queue Connected Systems
  - "Streaming Systems"
- Queue Implementations
- Stream Enabled Pipelining

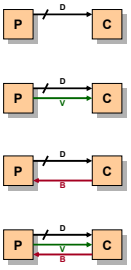
Modular System Design

- Decompose into modules to manage complexity, performance
- How to connect modules?
  - Wires
  - Shared bus
  - Register File / Memory
  - Queues

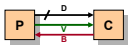


Flow Control

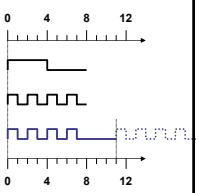
- Synchronous operation
  - Data every cycle
- Producer may stall
  - Data valid signal
- Consumer may stall
  - Back-pressure signal
- Either may stall
  - Valid + Back-pressure



Example: Bursty Communication

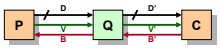


- Producer envelope: (8 cycles)
- Consumer envelope: (8 cycles)
- Together: (11 cycles)

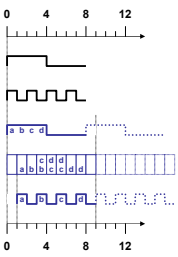


- Busrty communication
  - P + C each have average throughput ½
  - Producer is bursty
  - Consumer is steady

Bursty Communication + Queue

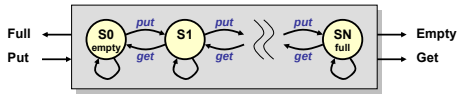


- Producer envelope: (8 cycles)
- Consumer envelope: (8 cycles)
- Producer → Queue: (8 cycles)
- Queue contents:
- Queue → Consumer: (8 cycles)



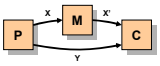
Abstract Queue

- **Container that maintains order**
  - FIFO = First In, First Out
  - Maintains system correctness regardless of communication delay
- **4 interfaces (methods)**
  - Full, Empty, Put, Get

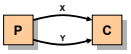


Queues Reschedule Data

- **For performance**
  - Smooth bursty communication
  - Smooth dynamic rate communication
- **For correctness – prevent deadlock**
  - Align data tuples that arrive with different delays, like pipeline registers



  - Reorder: P:{x,y} C:{y,x}
  - Store: P:{x*,y} C:{x,y}



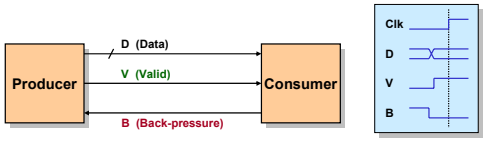
- **For convenience**
  - Buffer up / packetize to communicate with microprocessor
  - Off chip
  - Time multiplexed computation (SCORE)

Streaming Systems

- **Suppose queues were the *only* form of IPC**
  - *Stream* = FIFO channel with buffering (queue)
  - Every compute module (*process*) must stall waiting for
    - Input data
    - Output buffer space
- **System is robust to delay, easy to pipeline**
- **Hardware design decisions:**
  - Stream / flow control protocol
  - Process control (fire, stall)
  - Queue implementation
  - Stream pipelining
  - Queue depths

Wire Protocol for Streams

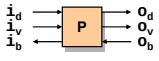
- **D = Data, V = Valid, B = Back-pressure**
- **Synchronous (rendezvous) transaction protocol**
  - Producer asserts **V** when **D** ready, Consumer deasserts **B** when ready
  - Transaction commits if ($\neg$**B** $\wedge$ **V**) at clock edge



Process Control (Fire / Stall)

- **In state X, *fire* if**
  - Inputs desired by X are ready (Valid)
  - Outputs emitted by X are ready (Back-pressure)
- **Firing guard / control flow**

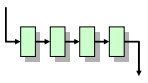
```
if (iv && !ob) begin
    ib=0; ov=1;
    ...
end
```



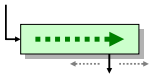
- **Subtlety: master, slave**
  - Process is slave
    - To synchronize streams, (1) wait for flow control in, (2) fire / emit out
  - Connecting two slaves would deadlock
  - Need master (queue) between every pair of modules

Queue Implementations

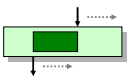
- **Systolic**
  - Cascade of depth-1 stages (or depth-N)



- **Shift register**
  - Put: shift all entries
  - Get: tail pointer

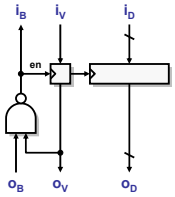


- **Circular buffer**
  - Memory with head / tail pointers



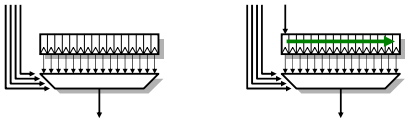
Enabled Register Queue

- **Systolic, depth-1 stage**
- **1 state bit (empty/full) = V**
- **Shift data in unless**
  - Full and downstream not ready to consume queued element
- **Area → 1 FF per data bit**
- **Area on FPGA**
  - Area → 1 LUT-FF cell per data bit
  - But depth-1 (1 stage) is nearly free, since data registers pack with logic
- **Speed: as fast as FF**
  - But combinationally connects producer, consumer, via B



Xilinx SRL16

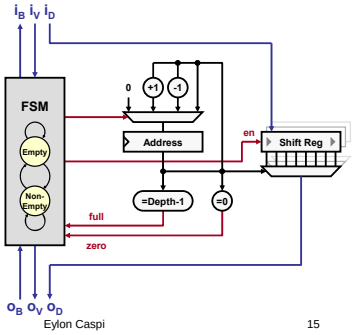
- **SRL16 = Shift register of depth 16 in one 4-LUT cell**
  - Shift register of arbitrary width: *parallel* SRL16, arbitrary depth: *cascade* SRL16
- **Improve queue density by 16x**



4-LUT Mode **SRL16 Mode**

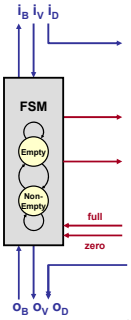
Shift Register Queue

- **State: empty bit + capacity counter**
- **Data stored in shift reg**
  - In at position 0
  - Out at position *Address*
- ***Address* = number of stored elements minus 1**
- **Flow control**
  - o_v = (*State*==Non-Empty)
  - i_b = (*Address*==*Depth*-1)
- **FSM decides**
  - Whether to consume
  - Whether to produce
  - Next Address
  - Next State
- **Depth ≥ 2 for full rate**



Shift Register Queue – Control

- **State Empty**
  - If (i_v) then *consume*
  - If (! i_v) then *idle*
- **State Non-empty**
  - If (*full*) then
    - If (o_b) then *idle*
    - If (! o_b) then *produce*
  - Else (*neither full nor empty*)
    - If ($i_v \wedge o_b$) then *consume*
    - If ($i_v \wedge$! o_b) then *consume* + *produce*
    - If (! $i_v \wedge o_b$) then *idle*
    - If (! $i_v \wedge$! o_b) then *produce*



Shift Register Queue – Verilog 1

```
module Q_srl (clock, reset, i_d, i_v, i_b, o_d, o_v, o_b);

   parameter depth = 16;     // - greatest #items in queue  (2 <= depth <= 256)
   parameter width = 16;     // - width of data (i_d, o_d)

   input     clock;
   input     reset;

   input  [width-1:0] i_d;      // - input  stream data (concat data + eos)
   input              i_v;      // - input  stream valid
   output             i_b;      // - input  stream back-pressure

   output [width-1:0] o_d;      // - output stream data
   output             o_v;      // - output stream valid
   input              o_b;      // - output stream back-pressure
```


Shift Register Queue – Verilog 2

```
parameter addrwidth =
              (  (((depth))     ==0) ? 0      // - depth==0   LOG2=0
               : (((depth-1)>>0)==0) ? 0      // - depth<=1   LOG2=0
               : (((depth-1)>>1)==0) ? 1      // - depth<=2   LOG2=1
               : (((depth-1)>>2)==0) ? 2      // - depth<=4   LOG2=2
               : (((depth-1)>>3)==0) ? 3      // - depth<=8   LOG2=3
               : (((depth-1)>>4)==0) ? 4      // - depth<=16  LOG2=4
               : (((depth-1)>>5)==0) ? 5      // - depth<=32  LOG2=5
               : (((depth-1)>>6)==0) ? 6      // - depth<=64  LOG2=6
               : (((depth-1)>>7)==0) ? 7      // - depth<=128 LOG2=7
               :                       8)     // - depth<=256 LOG2=8
               ;

  reg    [addrwidth-1:0] addr, addr_, a_;      // - SRL16 address
                                               //     for data output
  reg                    shift_en_;            // - SRL16 shift enable
  reg    [width-1:0]     srl [depth-1:0];      // - SRL16 memory

  parameter state_empty    = 1'b0;   // - state empty   : o_v=0 o_d=UNDEFINED
  parameter state_nonempty = 1'b1;   // - state nonempty: o_v=1 o_d=srl[addr]
                                     //     #items in srl = addr+1

  reg      state, state_;            // - state register
```

Shift Register Queue – Verilog 3

```
wire      addr_full_;             // - true iff addr==depth-1
wire      addr_zero_;             // - true iff addr==0

assign addr_full_ = (addr==depth-1);         // - queue full
assign addr_zero_ = (addr==0);               // - queue contains 1

assign o_d = srl[addr];                      // - output data from queue
assign o_v = (state==state_empty) ? 0 : 1;   // - output valid if non-empty
assign i_b = addr_full_;                     // - input bp if full
```


Shift Register Queue – Verilog 4

```
always @(posedge clock or negedge reset) begin       // - seq always: FFs
   if (!reset) begin
      state <= state_empty;
      addr  <= 0;
   end
   else begin
      state <= state_;
      addr  <= addr_;
   end
end // always @ (posedge clock or negedge reset)

always @(posedge clock) begin                        // - seq always: SRL16
   // - infer enabled SRL16 from shifting srl array
   // - no reset capability;  srl[] contents undefined on reset
   if (shift_en_) begin
      // synthesis loop_limit 256
      for (a_=depth-1; a_>0; a_=a_-1) begin
         srl[a_] <= srl[a_-1];
      end
      srl[0] <= i_d;
   end
end // always @ (posedge clock or negedge reset)
```


Shift Register Queue – Verilog 5

```
always @* begin                                  // - combi always
   case (state)

     state_empty: begin             // - (empty, will not produce)
        if (i_v) begin              // - empty & i_v => consume
           shift_en_ <= 1;
           addr_     <= 0;
           state_    <= state_nonempty;
        end
        else begin                  // - empty & !i_v => idle
           shift_en_ <= 0;
           addr_     <= 0;
           state_    <= state_empty;
        end
     end
```


Shift Register Queue – Verilog 6

```
state_nonempty: begin
          if (addr_full_) begin             // - (full, will not consume)
             if (o_b) begin                 // - full & o_b => idle
                shift_en_ <= 0;
                addr_     <= addr;
                state_    <= state_nonempty;
             end
             else begin                     // - full & !o_b => produce
                shift_en_ <= 0;
                addr_     <= addr-1;
                state_    <= state_nonempty;
             end
          end
```


Shift Register Queue – Verilog 7

```
          else begin                        // - (mid: neither empty nor full)
             if (i_v && o_b) begin          // - mid & i_v & o_b => consume
                shift_en_ <= 1;
                addr_     <= addr+1;
                state_    <= state_nonempty;
             end
             else if (i_v && !o_b) begin    // - mid & i_v & !o_b => cons+prod
                shift_en_ <= 1;
                addr_     <= addr;
                state_    <= state_nonempty;
             end
             else if (!i_v && o_b) begin    // - mid & !i_v & o_b => idle
                shift_en_ <= 0;
                addr_     <= addr;
                state_    <= state_nonempty;
             end
             else if (!i_v && !o_b) begin   // - mid & !i_v & !o_b => produce
                shift_en_ <= 0;
                addr_     <= addr_zero_ ? 0           : addr-1;
                state_    <= addr_zero_ ? state_empty : state_nonempty;
             end
          end  // else: !if(addr_full_)
        end  // case: state_nonempty
```


Shift Register Queue – Verilog 8

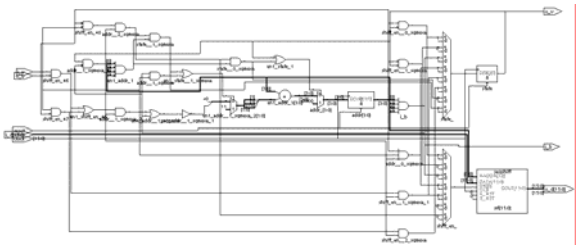
```
     endcase  // case(state)
   end  // always @ *

endmodule  // Q_srl
```

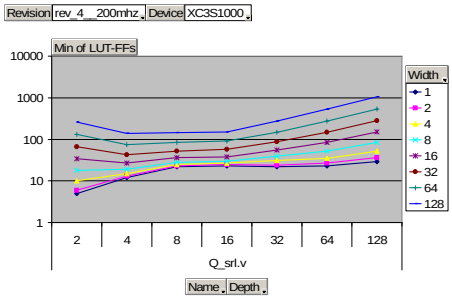
Characterization on FPGA

- **Synplify Pro 8.0 compiler**
  - Options: 200MHz, 0.5ns outputs, FSM Explorer, FSM Compiler, Resource Sharing, Retiming, Pipelining
- **Target: Xilinx Spartan 3 *1000* FPGA, speed -5**
  - XC3S1000 = 17,280 4-LUT cells
- **Script to compile with different depths, widths**
- **Graph in Excel using Pivot Charts**

SRL Queue: RTL (depth 16, width 16)

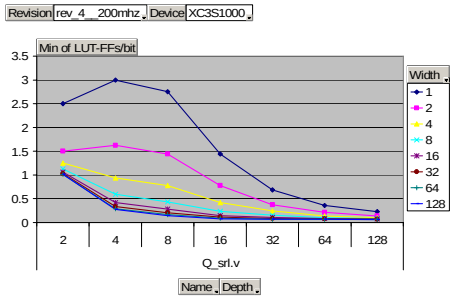


SRL Queue: Area



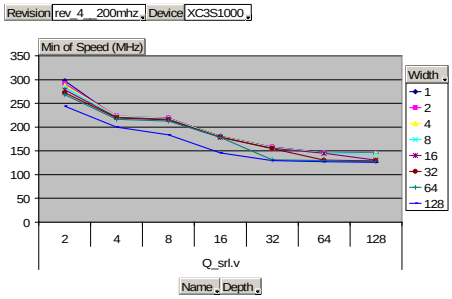
- **Depth 2: Shift register infers FFs, not SRL16**

SRL Queue: Area Per Bit

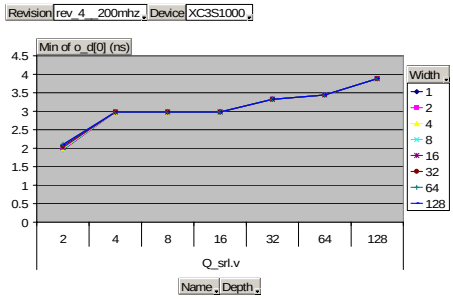


- **Quickly beats enabled register queue (1 LUT-FF per bit)**

SRL Queue: Speed

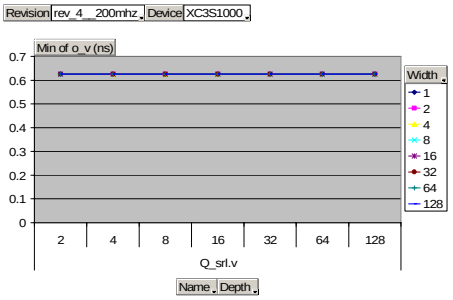


SRL Queue: Data Delay



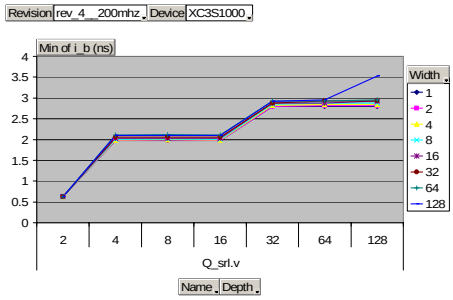
- **Slow Clk-to-D due to dynamic addressing of shift register**

SRL Queue: Valid Delay



- Clk-to-V = Clk-to-Q of state register

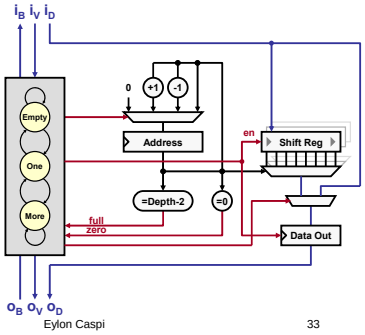
SRL Queue: Back-Pressure Delay



- Clk-to-B slows due to (1) wider addr cmp, (2) higher addr fanout

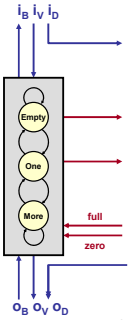
SRL Queue with Data Output Reg. (SRL+D)

- **Registered data out**
  - o_d (clock-to-Q delay)
  - Non-retimable
- **Data output register *extends* shift register**
- **Bypass shift register when queue empty**
- **3 States**
- ***Address* = number of stored elements minus 2**
- **Flow control**
  - o_v = !(*State*==Empty)
  - i_b = (*Address* ==*Depth*-2)

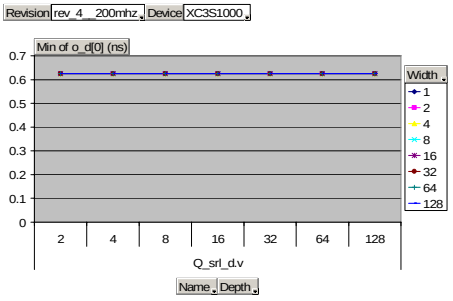


SRL+D Queue – Control

- **State Empty** (consume into D out reg.)
  - If (i_v) then *consume*
  - If (! i_v) then *idle*
- **State One** (consume into shift reg.)
  - If ($i_v \wedge o_b$) then *consume*
  - If ($i_v \wedge$! o_b) then *consume* + *produce*
  - If (! $i_v \wedge o_b$) then *idle*
  - If (! $i_v \wedge$! o_b) then *produce*
- **State More** (consume into shift reg.)
  - If (*full*) then
    - If (o_b) then *idle*
    - If (! o_b) then *produce*
  - Else (*neither full nor empty*)
    - If ($i_v \wedge o_b$) then *consume*
    - If ($i_v \wedge$! o_b) then *consume* + *produce*
    - If (! $i_v \wedge o_b$) then *idle*
    - If (! $i_v \wedge$! o_b) then *produce*

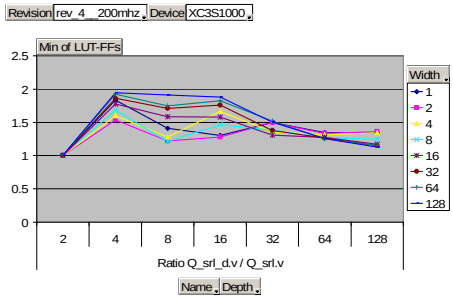


SRL+D: Data Delay



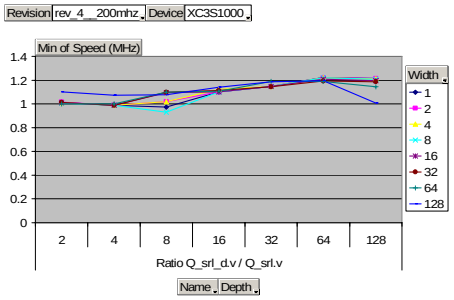
- Clk-to-D = Clk-to-Q of data output register

SRL+D Queue: Area Change



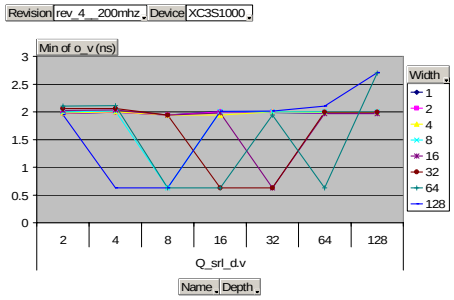
- Larger area from data out reg. – cannot pack with shift reg.

SRL+D Queue: Speedup



- **Slight speedup, up to ~20%**

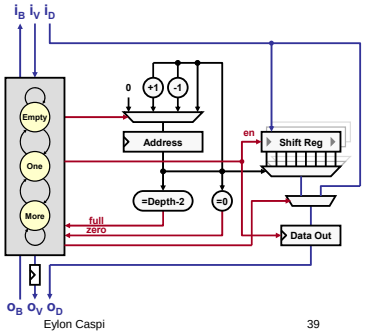
SRL+D Queue: Valid Delay



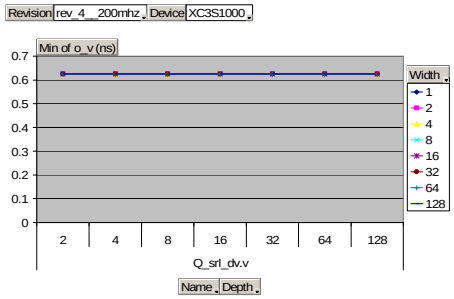
- **o_v = !(*State*==Empty), state is encoded**

Pre-Computed Valid (SRL+DV)

- **Registered valid out**
  - o_v (clock-to-Q delay)
  - Non-retimable
- **Flow control**
  - o_{v_next} = !(*State_next* ==Empty)
  - i_b = (*Address* ==*Depth*-2)

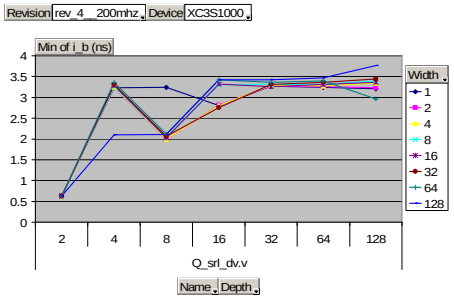


SRL+DV: Valid Delay



- **Clk-to-V = Clk-to-Q of V output register**

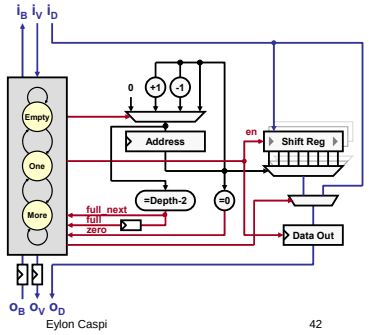
SRL+DV: Back-Pressure Delay



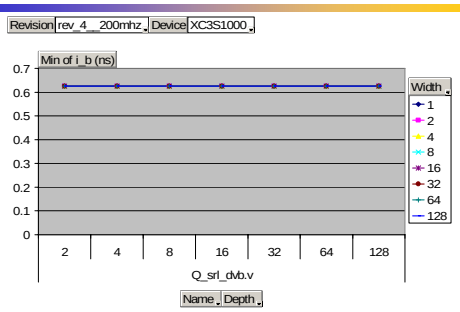
- **Clk-to-B slows w/depth, (1) wider addr cmp, (2) higher addr fanout**

Pre-Computed Back-Pressure (SRL+DVB)

- **Registered back-pressure out**
  - o_b (clock-to-Q delay)
  - Non-retimable
- **Based on pre-computed fullness**
  - *full_next* = (*Address_next* ==*Depth*-2)
- **Flow control**
  - o_{v_next} = !(*State_next* ==Empty)
  - i_{b_next} = (*Address_next* ==*Depth*-2)

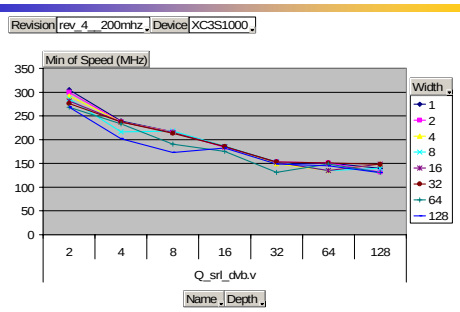


SRL+DVB: Back-Pressure Delay



- **Clk-to-B = Clk-to-Q of B output register**

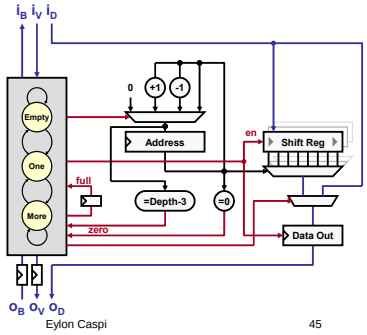
SRL+DVB: Speed



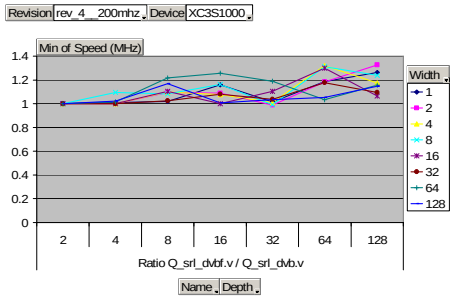
- **Can we improve speed?**

Specialized, Pre-Computed Fullness (SRL+DVBF)

- **Critical loop:**
  - Address compare,
  - FSM,
  - Address update
- **Speed up *full* pre-computation by special-casing *full_next* for each state**
- **Flow control**
  - o_v_next = !(State_next ==Empty)
  - i_b_next = full_next
- ***zero* pre-computation is less critical**

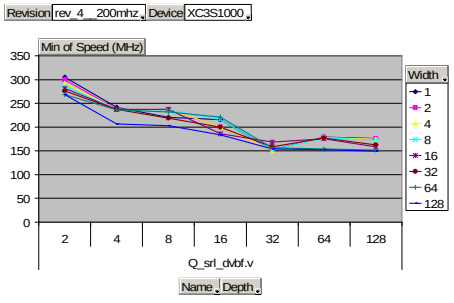


SRL+DVBF: Speedup

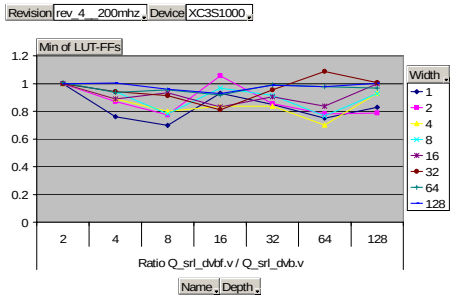


- **Speedup from specialization of *full_next*, up to ~30%**

SRL+DVBF: Speed

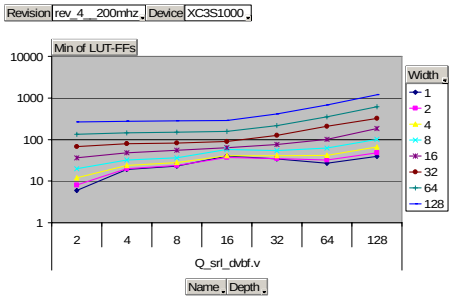


SRL+DVBF: Area Change

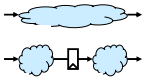


- **Area savings from specialization of *full_next*, up to ~30%**

SRL+DVBF: Area



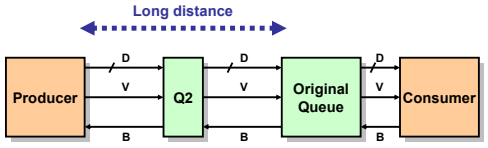
Stream Enabled Pipelining



- **Pipelining = inserting registers to break-up long combinational delay, improve MHz**
  - Logic pipelining: break-up deep logic
  - Interconnect pipelining: break-up long wires
- **Pipelining adds clock cycles of latency**
  - Signals out of sync, stale
- **Requires architectural modification – *difficult***
  - *E.g.* microprocessor pipelined function unit
- **Stream pipelining requires only *stream* modification – *easy***
  - Stream abstraction (queue) admits arbitrary delay
  - Stream implementation admits stylized pipelining

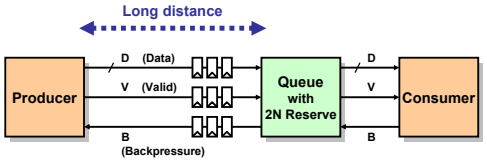
Interconnect Relaying

- **Break-up long distance streams**
- **Relay through depth-2 shift-register queue(s)**
  - Need depth-2 for full throughput (depth-1 is ½ throughput)
  - Can cascade multiple relay stages for longer distance



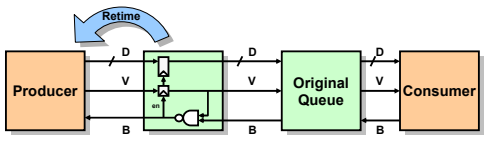
Interconnect Pipelining

- **Add *N* pipeline registers to D, V, B**
  - Mobile registers for placer
- **Stale flow control may overflow queue (by 2*N*)**
  - Staleness = total delay on B-V feedback loop = 2*N*
- **Modify downstream queue to emit back-pressure when empty slots ≤ 2*N***



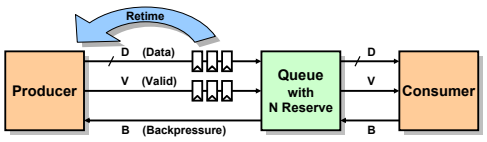
Logic Relaying + Retiming

- **Break-up deep logic in a process**
- **Relay through enabled register queue(s)**
- **Retime registers into adjacent process**
  - This pipelines feed-forward parts of process's datapath
  - Can retime into producer or consumer
- **No manual modification of processes!**



Logic Pipelining

- **Add *N* pipeline registers to D, V**
- **Retime backwards**
  - This pipelines feed-forward parts of producer's data-path
- **Stale flow control may overflow queue (by *N*)**
- **Modify queue to emit back-pressure when empty slots ≤ *N***
- **No manual modification of processes!**



Summary

- **Queues reschedule data**
  - For performance, correctness, convenience
- **Queue Connected (Streaming) Systems**
  - Robust to delay, easy to pipeline
- **Queue Implementations**
  - Systolic – enabled register queue
  - Shift register queue + optimizations
- **Stream Enabled Pipelining**
  - Of interconnect, logic – without modifying process