

# 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

4/8/05
Eylon Caspi
2

# Modular System Design

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

4/8/05
Eylon Caspi
3

# 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

4/8/05
Eylon Caspi
4

# Example: Bursty Communication

- ◆ Producer envelope: (8 cycles)
- ◆ Consumer envelope: (8 cycles)
- ◆ Together: (11 cycles)
- ◆ Bursty communication
  - P + C each have average throughput 1/2
  - Producer is bursty
  - Consumer is steady

4/8/05
Eylon Caspi
5

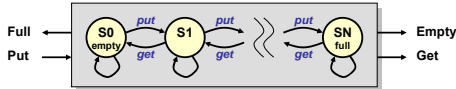
# Bursty Communication + Queue

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

4/8/05
Eylon Caspi
6

## 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



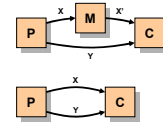
4/8/05

Eylon Caspi

7

## 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)



4/8/05

Eylon Caspi

8

## 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

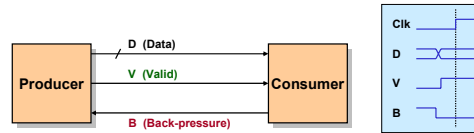
4/8/05

Eylon Caspi

9

## 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



4/8/05

Eylon Caspi

10

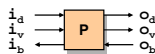
## Process Control (Fire / Stall)

- ◆ **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 (i_v && !o_b) begin
    i_b=0; o_v=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



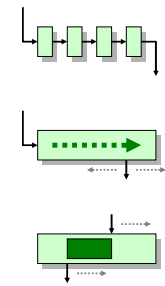
4/8/05

Eylon Caspi

11

## 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



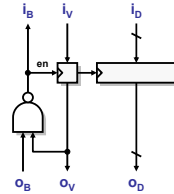
4/8/05

Eylon Caspi

12

## 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 combinational connects producer, consumer, via B



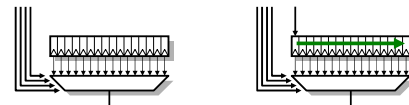
4/8/05

Eylon Caspi

13

## 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

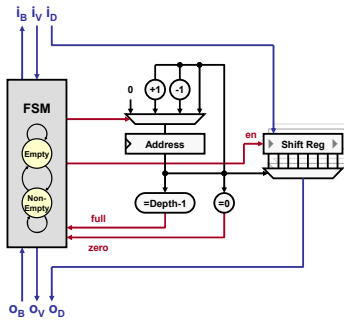
4/8/05

Eylon Caspi

14

## 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**
  - $o_v = (\text{State} == \text{Non-Empty})$
  - $i_b = (\text{Address} == \text{Depth}-1)$
- ◆ **FSM decides**
  - Whether to consume
  - Whether to produce
  - Next Address
  - Next State
- ◆ **Depth ≥ 2 for full rate**



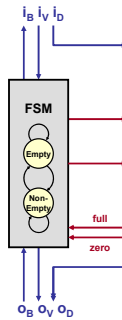
4/8/05

Eylon Caspi

15

## 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*



4/8/05

Eylon Caspi

16

## 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
    
```

4/8/05

Eylon Caspi

17

## 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_0; // - 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
    
```

4/8/05

Eylon Caspi

18

## 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
```

4/8/05

Eylon Caspi

19

## 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)
```

4/8/05

Eylon Caspi

20

## 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
    end
    else begin // - empty & !i_v => idle
      shift_en_ <= 0;
      addr_ <= 0;
      state_ <= state_empty;
    end
  end
end
```

4/8/05

Eylon Caspi

21

## 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
  end
  else begin // - full & !o_b => produce
    shift_en_ <= 0;
    addr_ <= addr-1;
    state_ <= state_nonempty;
  end
end
```

4/8/05

Eylon Caspi

22

## 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
```

4/8/05

Eylon Caspi

23

## Shift Register Queue – Verilog 8

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

4/8/05

Eylon Caspi

24

## Characterization on FPGA

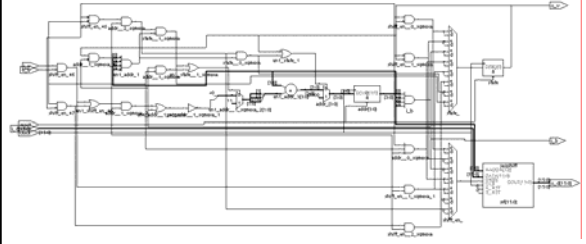
- ◆ Synplify Pro 8.0 compiler
  - Options: 200MHz, 0.5ns outputs, FSM Explorer, FSM Compiler, Resource Sharing, Retiming, Pipelining
- ◆ Target: Xilinx Spartan 3 1000 FPGA, speed -5
  - XC3S1000 = 17,280 4-LUT cells
- ◆ Script to compile with different depths, widths
- ◆ Graph in Excel using Pivot Charts

4/8/05

Eylon Caspi

25

## SRL Queue: RTL (depth 16, width 16)

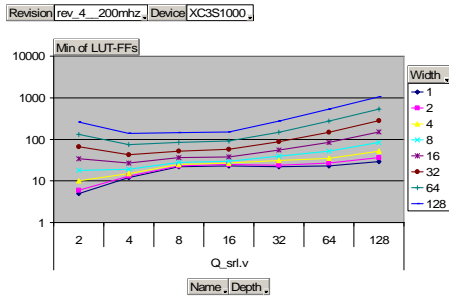


4/8/05

Eylon Caspi

26

## SRL Queue: Area



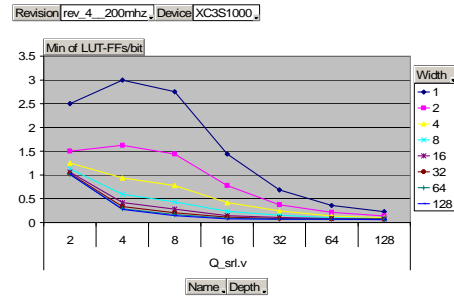
- ◆ Depth 2: Shift register infers FFs, not SRL16

4/8/05

Eylon Caspi

27

## SRL Queue: Area Per Bit



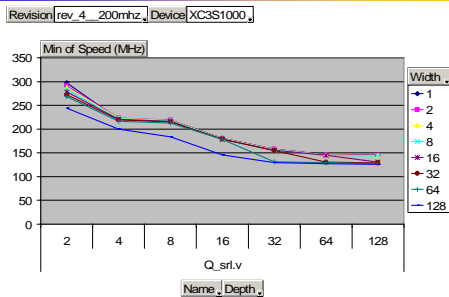
- ◆ Quickly beats enabled register queue (1 LUT-FF per bit)

4/8/05

Eylon Caspi

28

## SRL Queue: Speed

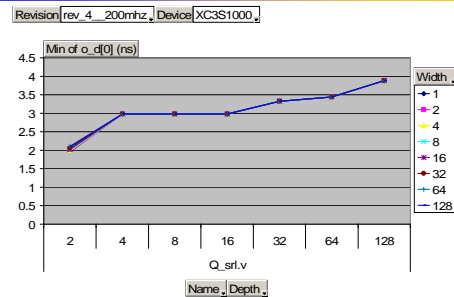


4/8/05

Eylon Caspi

29

## SRL Queue: Data Delay



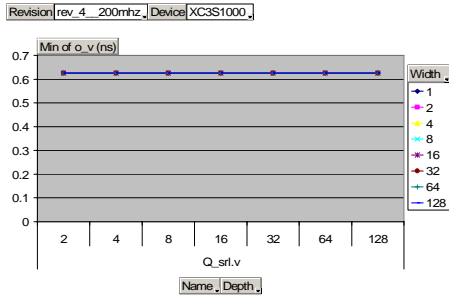
- ◆ Slow CLK-to-D due to dynamic addressing of shift register

4/8/05

Eylon Caspi

30

## SRL Queue: Valid Delay



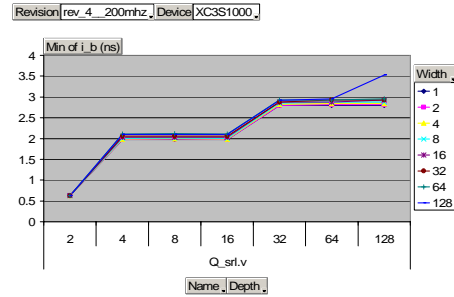
◆ Clk-to-V = Clk-to-Q of state register

4/8/05

Eylon Caspi

31

## SRL Queue: Back-Pressure Delay



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

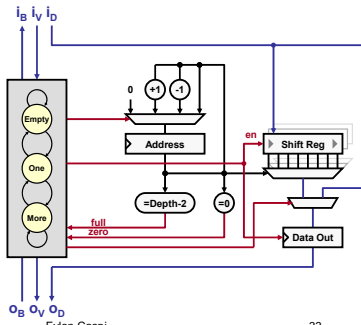
4/8/05

Eylon Caspi

32

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

- ◆ Registered data out
  - $o_v$  (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
  - $o_v = \text{!(State==Empty)}$
  - $i_b = (\text{Address} == \text{Depth}-2)$



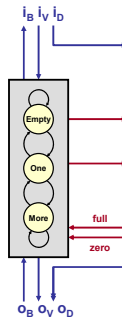
4/8/05

Eylon Caspi

33

## 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 ( $i_v \wedge o_v$ ) then consume
  - If ( $!i_v \wedge o_v$ ) then consume + produce
  - If ( $!i_v \wedge !o_v$ ) then idle
  - If ( $i_v \wedge !o_v$ ) then produce
- ◆ State More (consume into shift reg.)
  - If (full) then
    - If ( $!o_v$ ) then idle
    - If ( $!o_v$ ) then produce
  - Else (neither full nor empty)
    - If ( $i_v \wedge o_v$ ) then consume
    - If ( $!i_v \wedge !o_v$ ) then consume + produce
    - If ( $!i_v \wedge o_v$ ) then idle
    - If ( $i_v \wedge !o_v$ ) then produce

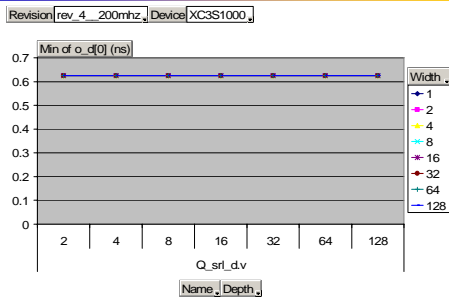


4/8/05

Eylon Caspi

34

## SRL+D: Data Delay



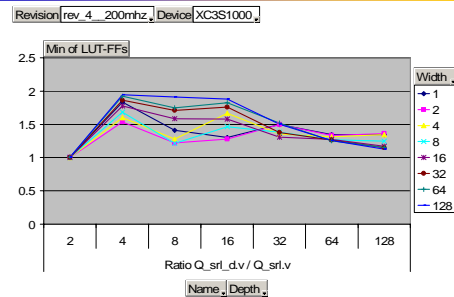
◆ Clk-to-D = Clk-to-Q of data output register

4/8/05

Eylon Caspi

35

## SRL+D Queue: Area Change



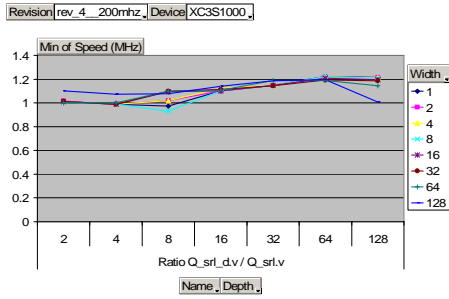
◆ Larger area from data out reg. – cannot pack with shift reg.

4/8/05

Eylon Caspi

36

## SRL+D Queue: Speedup



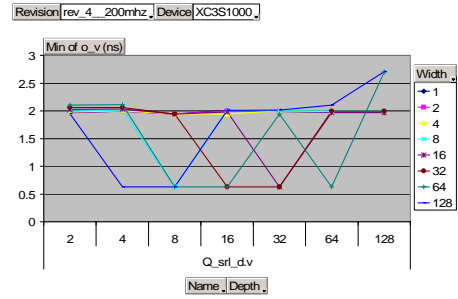
◆ Slight speedup, up to ~20%

4/8/05

Eylon Caspi

37

## SRL+D Queue: Valid Delay



◆  $o_v = !(State == Empty)$ , state is encoded

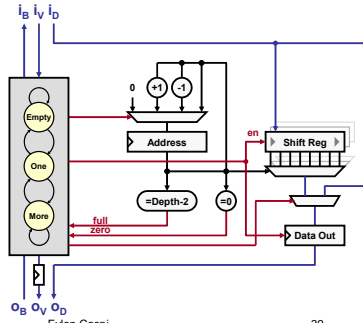
4/8/05

Eylon Caspi

38

## Pre-Computed Valid (SRL+DV)

- ◆ Registered valid out
  - $o_v$  (clock-to-Q delay)
  - Non-retimable
- ◆ Flow control
  - $o_v_{next} = !(State_{next} == Empty)$
  - $i_b = (Address == Depth-2)$

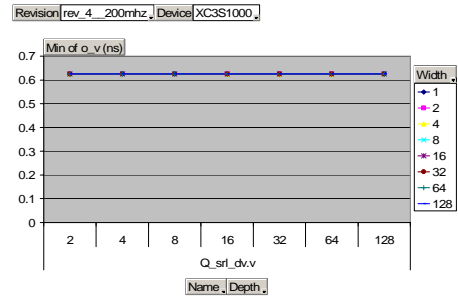


4/8/05

Eylon Caspi

39

## SRL+DV: Valid Delay



◆ Click-to-V = Click-to-Q of V output register

4/8/05

Eylon Caspi

40

## SRL+DV: Back-Pressure Delay



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

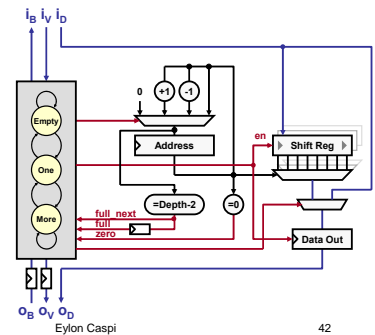
4/8/05

Eylon Caspi

41

## 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
  - $o_v_{next} = !(State_{next} == Empty)$
  - $i_b_{next} = (Address_{next} == Depth-2)$

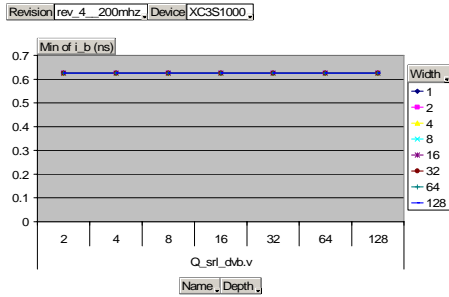


4/8/05

Eylon Caspi

42

## SRL+DVB: Back-Pressure Delay



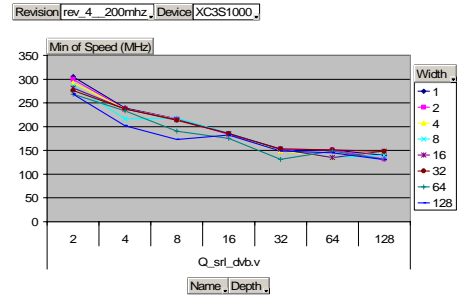
◆ Clk-to-B = Clk-to-Q of B output register

4/8/05

Eylon Caspi

43

## SRL+DVB: Speed



◆ Can we improve speed?

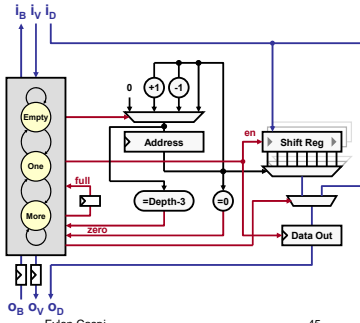
4/8/05

Eylon Caspi

44

## 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<sub>next</sub> = !(State<sub>next</sub> == Empty)
  - i<sub>b\_next</sub> = full<sub>next</sub>
- ◆ **zero pre-computation is less critical**

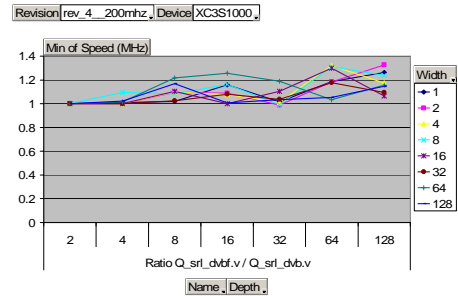


4/8/05

Eylon Caspi

45

## SRL+DVBF: Speedup



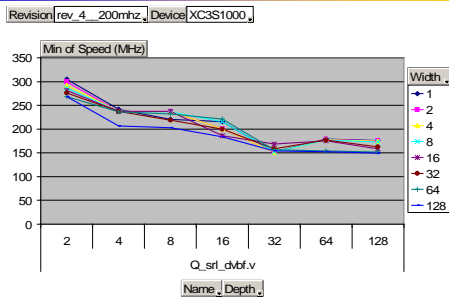
◆ Speedup from specialization of full\_next, up to ~30%

4/8/05

Eylon Caspi

46

## SRL+DVBF: Speed

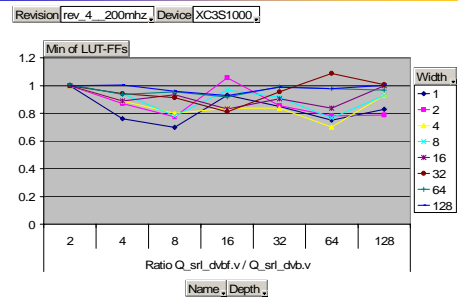


4/8/05

Eylon Caspi

47

## SRL+DVBF: Area Change



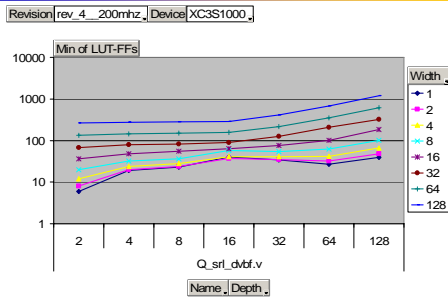
◆ Area savings from specialization of full\_next, up to ~30%

4/8/05

Eylon Caspi

48

## SRL+DVBF: Area



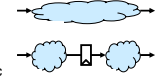
4/8/05

Eylon Caspi

49

## 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

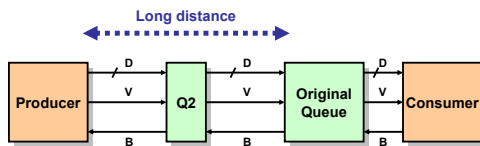
4/8/05

Eylon Caspi

50

## Interconnect Relaying

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



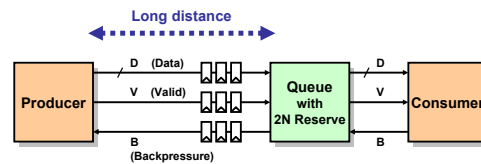
4/8/05

Eylon Caspi

51

## Interconnect Pipelining

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



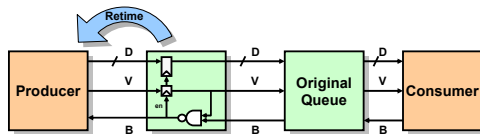
4/8/05

Eylon Caspi

52

## 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!**



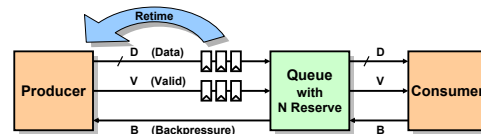
4/8/05

Eylon Caspi

53

## 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  $\leq N$**
- ◆ **No manual modification of processes!**



4/8/05

Eylon Caspi

54

## 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

4/8/05

Eylon Caspi

55