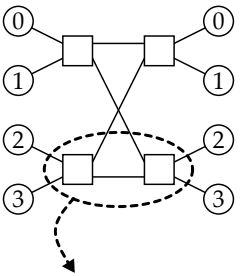
ECE 4750 Computer Architecture, Fall 2015
T07 Fundamental Network Microarchitecture

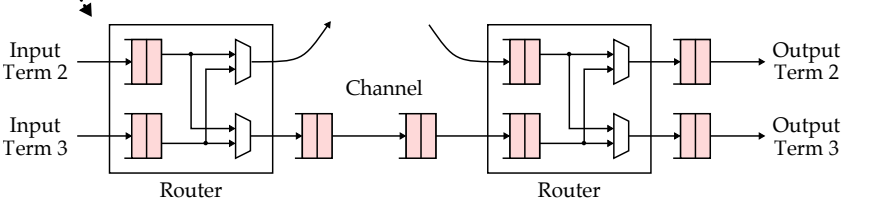
School of Electrical and Computer Engineering
Cornell University

revision: 2015-10-26-13-36

1. Buffer Microarchitecture



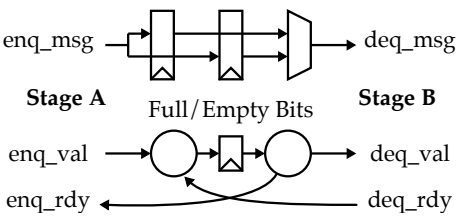
Buffer microarchitecture focuses on how we implement the many queues that are used in the network terminals, channels, and routers. We will use "network buffer" and "network queue" interchangeably.



- Network queues are usually one read, one write port
- Network queues implemented with either register files or SRAMs
- Total buffering can be a critical technology constraint, especially in on-chip networks where wires are cheap but buffers are expensive
- We will study three kinds of buffers:
  - Normal Queues : no combinational paths
  - Pipe Queues : combinational path from deq ready to enq rdy
  - Bypass Queues : combinational path from enq val to deq val

1.1. Normal Queues

Normal queues have no combinational connections between the val/rdy signals. This means we cannot enqueue a new message if the queue is full, even if we are dequeuing a message on the same cycle.

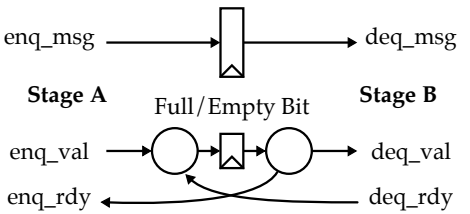


	0	1	2	3	4	5	6	7	8	9	10	11
pkt0												
pkt1												
pkt2												
pkt3												
pkt4												
pkt5												
pkt6												

cyc	A	(	|	)	B
0		(	|	)	
1		(	|	)	
2		(	|	)	
3		(	|	)	
4		(	|	)	
5		(	|	)	
6		(	|	)	
7		(	|	)	
8		(	|	)	
9		(	|	)	
10		(	|	)	
11		(	|	)	

Assume the dequeue interface is not ready on cycles 4–6

A single-element normal queue cannot sustain full throughput. The cycle after we enqueue a message, the queue is full preventing us from enqueing a new message *even* if we are dequeuing a message on that same cycle.



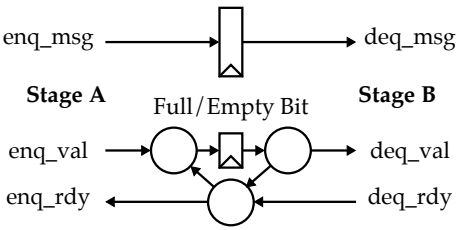
	0	1	2	3	4	5	6	7	8	9	10	11
`pkt0`												
`pkt1`												
`pkt2`												
`pkt3`												
`pkt4`												
`pkt5`												
`pkt6`												

cyc	A	( )	B
0		( )	
1		( )	
2		( )	
3		( )	
4		( )	
5		( )	
6		( )	
7		( )	
8		( )	
9		( )	
10		( )	
11		( )	

Assume the dequeue interface is not ready on cycles 4–6

1.2. Pipe Queues

Pipe queues have a combinational connection from the deq_rdy to enq_rdy. This means we can now enqueue a new message even if the queue is full, as long as we are dequeuing a message on the same cycle.



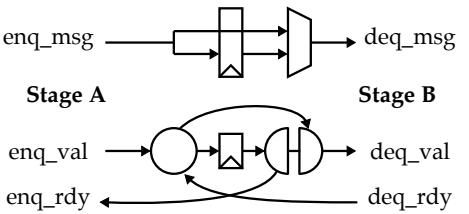
	0	1	2	3	4	5	6	7	8	9	10	11
pkt0												
pkt1												
pkt2												
pkt3												
pkt4												
pkt5												
pkt6												

cyc	A ( ) B
0	( )
1	( )
2	( )
3	( )
4	( )
5	( )
6	( )
7	( )
8	( )
9	( )
10	( )
11	( )

Assume the dequeue interface is not ready on cycles 4–6

1.3. Bypass Queues

Bypass queues have a combinational connection from the enq_val/enq_msg to deq_val/deq_msg. This means if the queue is empty, the message will "bypass" the queue and be sent combinationally from the enqueue interface to the dequeue interface.

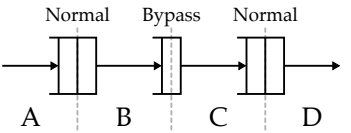


	0	1	2	3	4	5	6	7	8	9	10	11
pkt0												
pkt1												
pkt2												
pkt3												
pkt4												
pkt5												
pkt6												

cyc	A ( ) B
0	( )
1	( )
2	( )
3	( )
4	( )
5	( )
6	( )
7	( )
8	( )
9	( )
10	( )
11	( )

Assume the dequeue interface is not ready on cycles 4–6

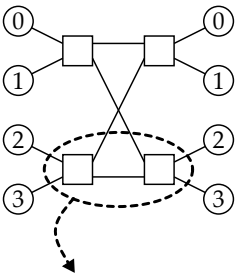
1.4. Composing Queues



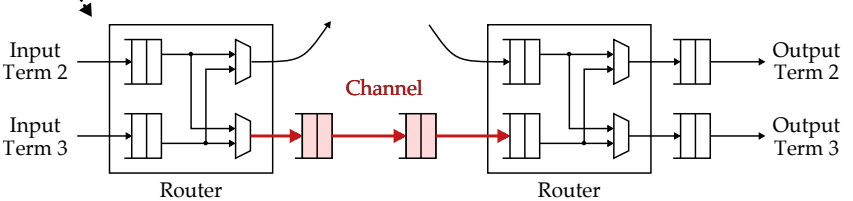
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
pkt0																
pkt1																
pkt2																
pkt3																
pkt4																
pkt5																
pkt6																
pkt7																

cyc	A	(	|	)	B	(		)	C	(	|	)	D
0		(	|	)		(		)		(	|	)	
1		(	|	)		(		)		(	|	)	
2		(	|	)		(		)		(	|	)	
3		(	|	)		(		)		(	|	)	
4		(	|	)		(		)		(	|	)	
5		(	|	)		(		)		(	|	)	
6		(	|	)		(		)		(	|	)	
7		(	|	)		(		)		(	|	)	
8		(	|	)		(		)		(	|	)	
9		(	|	)		(		)		(	|	)	
10		(	|	)		(		)		(	|	)	
11		(	|	)		(		)		(	|	)	

2. Channel Microarchitecture



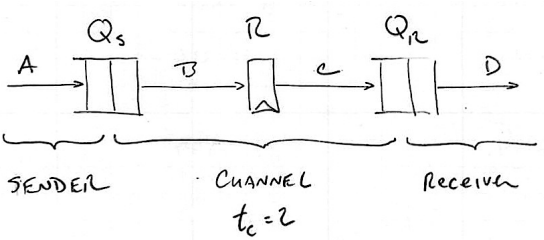
Channel microarchitecture focuses on how we pipeline the channel and thus how the sender and receiver coordinate to manage the buffer at the receiver. Often called flow control.



- Start by assuming single phit packets, study two low-level flow-control schemes:
  - On-Off Flow Control
  - Elastic-Buffer Flow Control
- Then assume multi-phit packets, study two higher-level flow-control schemes:
  - Store-and-Forward Flow Control
  - Virtual-Cut-Through Flow Control
- Note that all of these flow-control schemes are non-dropping, but dropping flow-control schemes are also possible
  - Reduces buffering requirements
  - Requires nacks or timeouts
  - Can be expensive under high-load due to retries

2.1. On-Off Flow-Control

- Use a single on-off signal to indicate whether or not the receiver queue is full: on means still space, off means queue is full
- On-off signal is essentially the same as a stall signal
- May need to send this signal ahead of time to ensure that by the time we can actually stall the channel we don't have to drop packets
- We will use the following example to explore four different ways of implementing on-off flow control:
  - Combinational stall signal
  - Partial combinational stall signal
  - Pipelined stall signal
  - Partial pipelined stall signal

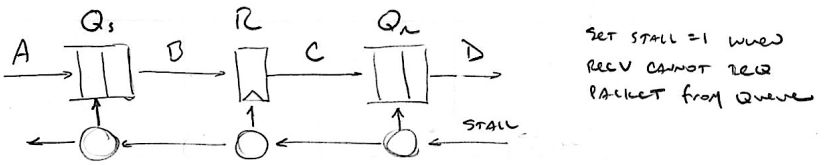


* ASSUME SINGLE FLIT PACKETS

HOW DOES SENDER KNOW WHEN IT CAN SEND A PACKET TO RECEIVER? HOW DOES IT KNOW THERE IS ROOM IN THE RECEIVER'S QUEUE?

On/off flow-control with combinational stall signal

Assume we can combinationally stall all pipeline registers in the channel as well as the sender queue.

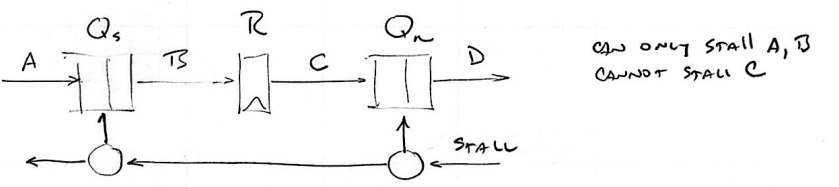


pkt0													
pkt1													
pkt2													
pkt3													
pkt4													
pkt5													
pkt6													
pkt7													

- How deep does the receiver queue need to avoid dropping packets?
- What if $t_c = 3$?

On/off flow-control with partial combinational stall signal

Assume we can combinationally stall the sender queue, but we cannot stall the pipeline registers in the channel. This might be because we have multiple bits in flight on a cable or wire at the same time, or the overhead for stalling all pipeline registers is too high.

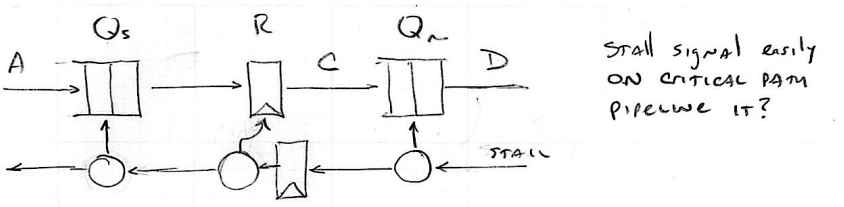


pkt0													
pkt1													
pkt2													
pkt3													
pkt4													
pkt5													
pkt6													
pkt7													

- How deep does the receiver queue need to avoid dropping packets?
- Extra buffering in receiver queue is called “skid buffering”
- What if $t_c = 3$?

On/off flow-control with pipelined stall signal

Assume that while we can stall the sender queue and pipeline registers, we cannot do so combinationally. We must pipeline the stall signal. This might be because the stall signal is on the critical path.

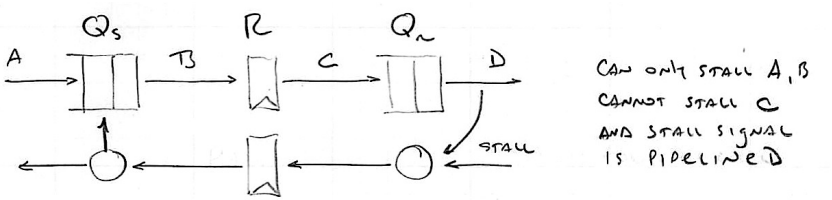


pkt0													
pkt1													
pkt2													
pkt3													
pkt4													
pkt5													
pkt6													
pkt7													

- How deep does the receiver queue need to avoid dropping packets?
- What if $t_c = 3$?

On/off flow-control with partial pipelined stall signal

Assume that we cannot stall the pipeline registers *and* we must pipeline the stall signal for the sender queue. This might because we have multiple bits in flight on a cable or wire at the same time, and it takes some number of cycles to send the stall signal back to the sender.

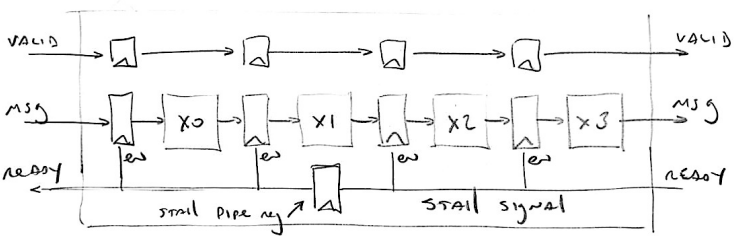


pkt0													
pkt1													
pkt2													
pkt3													
pkt4													
pkt5													
pkt6													
pkt7													

- How deep does the receiver queue need to sustain full throughput?
- What if $t_c = 3$?
- In general, need $2 \times t_c$ elements in Q_R to avoid dropping packets.
  - Few extra elements depending on how receiver turn-around
  - Buffers are poorly utilized
  - Stall as *soon* as D stalls *even* if sender has no packets to send!
  - Credit-based flow-control has better buffer utilization

Activity: Flow control in a pipelined multiplier

Consider the following 4-stage pipelined multiplier with a val/rdy input/output interface



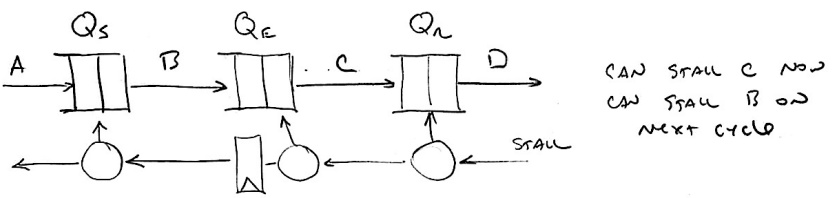
Assume the stall signal is on the critical path and so we insert a stall pipeline register in the X1 stage

Draw a pipeline diagram illustrating how this multiplier executes a stream of multiply transactions. What modifications do we need to avoid dropping transactions?

mul A													
mul B													
mul C													
mul D													
mul E													
mul F													
mul G													

2.2. Elastic Buffer Flow-Control

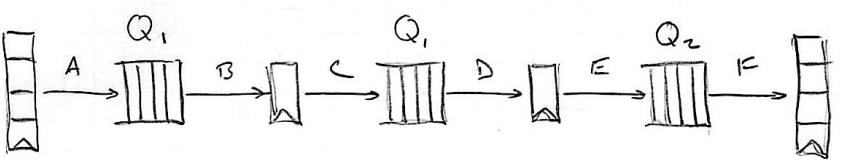
Instead of centralizing the buffering required to avoid dropping packets in the receiver, we can also distribute that buffering along the channel. In elastic-buffer flow-control, each pipeline register turns into a small two-element queue. Head of the queue is effectively the pipeline register, while the second element is skid-buffering.



pkt0													
pkt1													
pkt2													
pkt3													
pkt4													
pkt5													
pkt6													
pkt7													

2.3. Store-and-Forward Flow-Control

So far we have assumed single-phit packets. How should we handle multi-phit packets? Assume we always allocate buffers in units of a complete packet (there are other schemes that do not require this). In store-and-forward flow-control, once all phits in a packet have been completely received in a queue, we can then forward the phits to the next queue.

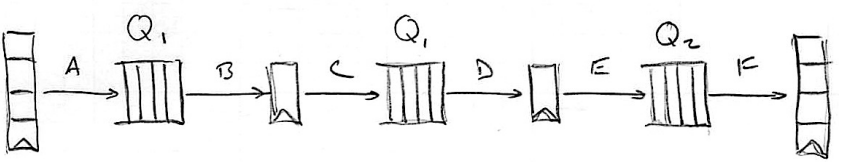


Assume four phits/packet, so each packet has one head phit (H), two body phits (B), and one tail phit (T).

pkt0 H																		
pkt0 B																		
pkt0 B																		
pkt0 T																		

2.4. Virtual-Cut-Through Flow-Control

Store-and-forward is common in large-scale data-center or multi-socket networks, but the overhead of serializing/deserializing packets can be significant in on-chip networks. Again, assume we always allocate buffers in units of a complete packet. In virtual-cut-through flow-control, we can start forwarding phits to the next queue right-away.

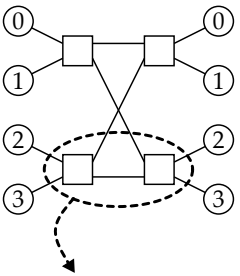


Assume four phits/packet, so each packet has one head phit (H), two body phits (B), and one tail phit (T).

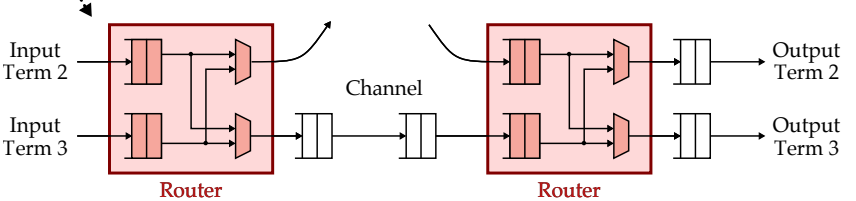
pkt0 H																		
pkt0 B																		
pkt0 B																		
pkt0 T																		

In this course, always assume virtual-cut-through flow-control.

3. Router Microarchitecture

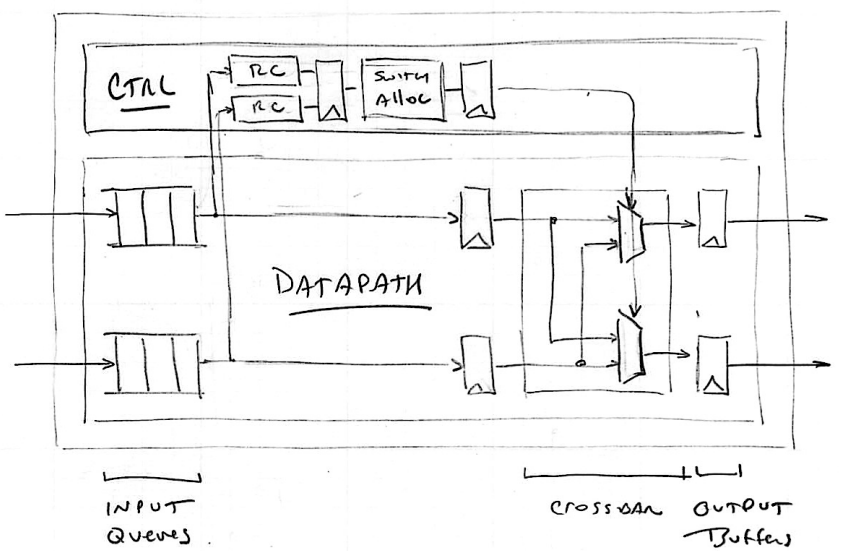


Router microarchitecture focuses on how we do the routing and arbitration within each router of the network. Although an FSM microarchitecture is possible, on-chip networks almost always use single-cycle or pipelined microarchitectures.



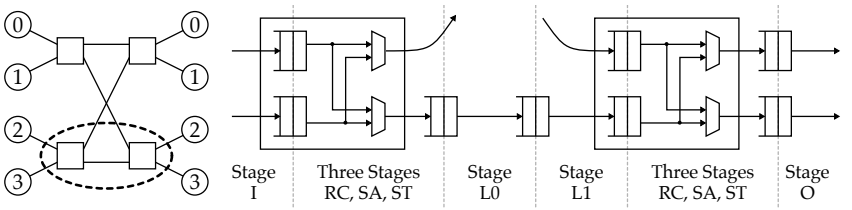
3.1. Pipelined Router

Three-stage router pipeline suitable for simple 2-ary butterfly topology



- Router Computation (RC)
  - Simple combinational logic for oblivious routing algorithm
  - Duplicate per input port to avoid structural hazard
- Switch Allocation (SA)
  - Two 2-input arbiters, one per output port
  - Grant and hold, hold after head phit until tail phit
- Switch Traversal (ST)
  - Cross the crossbar and write output buffer

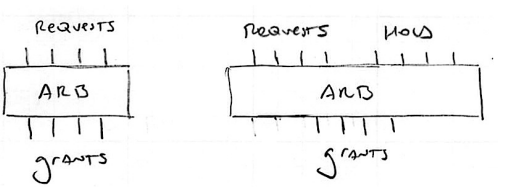
Let's use a pipeline diagram to illustrate a four-phit packet traversing from input terminal 3 to output terminal 3.



pkt0 H													
pkt0 B													
pkt0 B													
pkt0 T													

- Only header phit does route computation
- Body/tail phits cannot bypass header phit, must wait in input queue

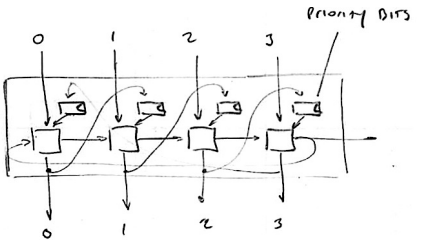
3.2. Arbitration



- Requesters set request signal high if need shared resource
- Arbiter sets a single grant signal high for winning requester
- Grant and hold arbiter allows requester to "hold on" to shared resource until finished

Round-Robin Arbiter

In fixed-priority abritration, the same requester always has the highest priority. In round-robin arbitration, the priority changes: winner on one cycle has lowest priority on next cycle.

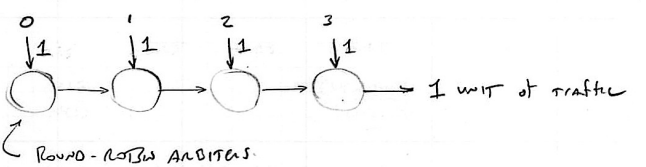


Reqs				Priority				Grants			
0	1	2	3	0	1	2	3	0	1	2	3
0	0	0	0	1	0	0	0				
0	0	0	0								
0	0	0	0								
0	0	0	0								
0	0	0	0								

Arbiter Fairness

- WEAK FAIRNESS : every REQUEST EVENTUALLY SERVED
- STRONG FAIRNESS : REQUESTS SERVED EQUALLY OFTEN

LOCAL VS. GLOBAL FAIRNESS



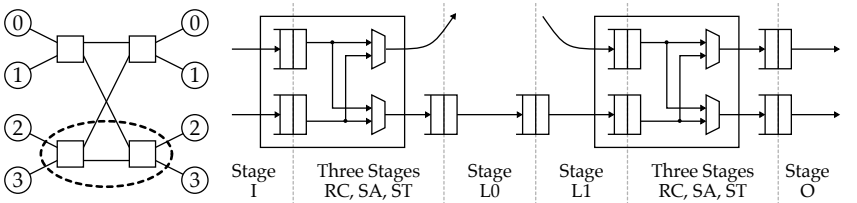
WHAT PERCENTAGE OF THE OUTPUT BANDWIDTH COMES FROM EACH INPUT TERMINAL?

0: 0.125
1: 0.125
2: 0.25
3: 0.5

} NO global STRONG FAIRNESS even though each ROUND ROBIN ARBITER HAS LOCAL STRONG FAIRNESS

Pipeline diagram with arbitration

Let's use a pipeline diagram to illustrate a two four-phit packets traversing through the network. Packet 0 is going from input terminal 2 to output terminal 2. Packet 1 is going from input terminal 3 to output terminal 3. Both packets arrive at the first router at the same time. Assume packet 0 wins arbitration.



pkt0 H																			
pkt0 B																			
pkt0 B																			
pkt0 T																			

pkt1 H																			
pkt1 B																			
pkt1 B																			
pkt1 T																			