Routing algorithms

What path does a message travel from its source to its destination? This is determined by the routing algorithm.

Given a current node and a destination node, the routing algorithm chooses the next *port* and *channel* on which to send out the message.

Thus, a routing algorithm is a function $R: N \times N \rightarrow C$.

A switch usually uses one of three mechanisms to determine the output channel from info in the packet header.

- arithmetic,
- source-based port select, and
- a table lookup.

A switch needs to route a packet every few cycles, so it needs to be fast.

In regular topologies, simple arithmetic suffices.

Example: Δx, Δy routing in a grid. (What is Δx, Δy routing?)

Finish this example.

West ($-x$)	$\Delta x < 0$
East ($+x$)	
South ($-y$)	$\Delta x = 0$,
North ($+y$)	
Processor	$\Delta x = 0$, $\Delta y = 0$

To accomplish this routing, the switch needs to test the address in the header and increment or decrement one routing field.

Usually, routing is done in *dimension order*—first across the x-dimension, then the y-dimension, then the z-dimension (if any), etc.

So in a binary cube, the switch determines the most significant bit where the destination node number differs from the current node number.

Sometimes a packet header has a *relative address* embedded in it.

Example: If the source node is 001010 and the destination node is 100101, what would be the relative address embedded at the source node?

In this case, the switch just looks for the first non-zero bit and routes accordingly.

In general, in a mesh or cube, routing is done by moving from lowest (x) dimension to the highest.

This is called routing in *dimension order*. In a hypercube, it is called *e-cube* routing. It was used in nCube hypercubes and the Intel Paragon, among others.

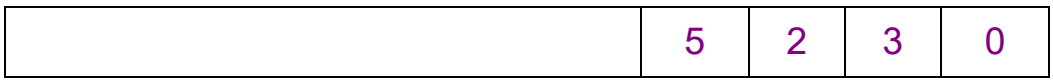
More generally, the source builds a header consisting of the output port # for each switch along the route.

This is called *source-based* routing.

Each switch just strips off the port number from the front of the message and sends it on.

Example: Let's assume we have a hypercube of dimension n, where n links are attached to each node. An output-port value of 0 means move across dimension 0 (i.e., flip the lsb); a value of n–1 means move across dimension n–1 (i.e., flip the msb).

Assume that n = 6, and that the header specifies these output ports:



If the packet starts out at node 36 (= 100100_2), where does it end up?

All of the intelligence is at the source node. Arbitrary routing can be supported without much logic in the switches.

Disadvantage: Header is large, of variable size.

Table-driven routing associates a small routing table at each switch. It allows for a small fixed-size header.

In table-driven routing,

- The packet header contains a routing field i.
- The output port is $R[i]$, where R is the routing table.
- Usually the table also contains the routing field for the next step in the route.

Disadvantage: The switch must contain quite a bit of routing state. Fairly large tables are needed even for simple routing algorithms.

This approach was used by ATM and HPPI switches, but isn't too practical for multiprocessors, because of the large number of routing patterns that they must support.

One important difference between network routers and multiprocessor switches: Time constraints.

Routing may be—

- *Deterministic*, where the route is completely determined by the (source, destination) pair, and not by the intermediate state, or
- *Adaptive*, where the route is influenced by traffic along the way.

A routing algorithm may be *minimal*, meaning that it only selects shortest paths toward the destination (no backtracking), or *nonminimal* (can allow longer paths).

Minimal algorithms need not be deterministic. Can you see why?

Similarly, adaptive algorithms may be minimal.

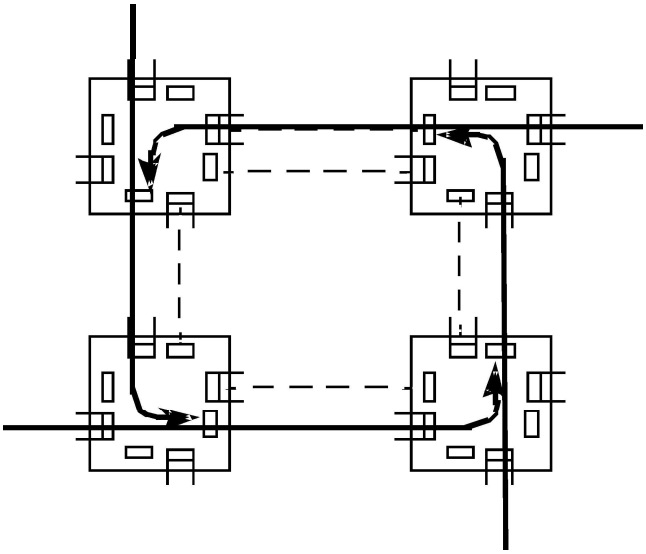
Let's compare the two types.

- Do nonminimal algorithms have any advantages over minimal algorithms?

- Do minimal algorithms have any advantages over nonminimal algorithms?

Deadlock-free routing

Deadlock occurs when two or more packets are "circularly" waiting for resources that are held by other packets in the group.



The diagram at the right illustrates how this can occur with 2-hop messages. Each packet is waiting for a link occupied by another packet.

What conditions are necessary for deadlock to occur?

- a *shared resource*
- that is *incrementally allocated* and
- non-preemptible.

A channel is a shared resource, and channels are acquired incrementally, as a route is built up.

How can deadlock be avoided? Basically, by constraining how channel resources are allocated. Routing in dimension order is one way. How can we see that in the diagram above?

How do we prove that an algorithm is deadlock free?

- Each channel *is* a resource (or each channel *contains* resources).

- Messages need to use resources in order; this introduces dependences between resources.
- We need to show that there are no cycles in the channel-dependence graph

We can do this by finding a numbering of channel resources such that every legal route follows a monotonic sequence

$\Rightarrow$ no traffic pattern can lead to deadlock

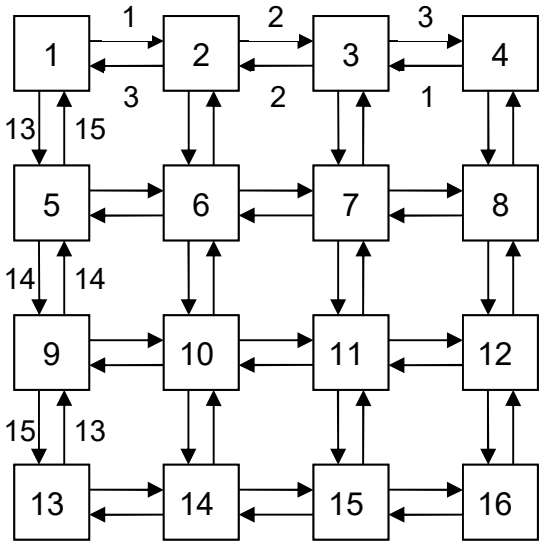
This is trivial for acyclic networks, such as shuffle-exchange and butterfly networks.

It is also trivial for trees and fat trees, as long as the upward and downward channels are independent.

Example: Show Δx, Δy routing on a k-ary 2D array is deadlock free.

We view each bidirectional channel as a pair of unidirectional channels numbered independently.

Channels in the x direction are given lower numbers than channels in the y direction.



Given this numbering, any routing sequence that starts out in the x direction, turns and then goes in the y direction is increasing.

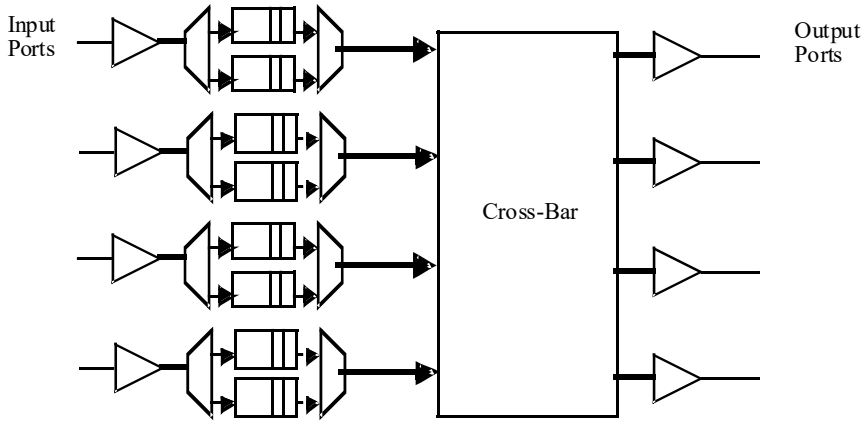
Virtual channels

This proof easily generalizes to hypercubes—but not to toruses because of the wraparound edges.

So, how can we do deadlock-free routing on toruses and other higher-order networks?

The basic approach is to provide *virtual channels*.

To do this we need more than one packet buffer per channel.



Example: Consider a torus with unidirectional links.

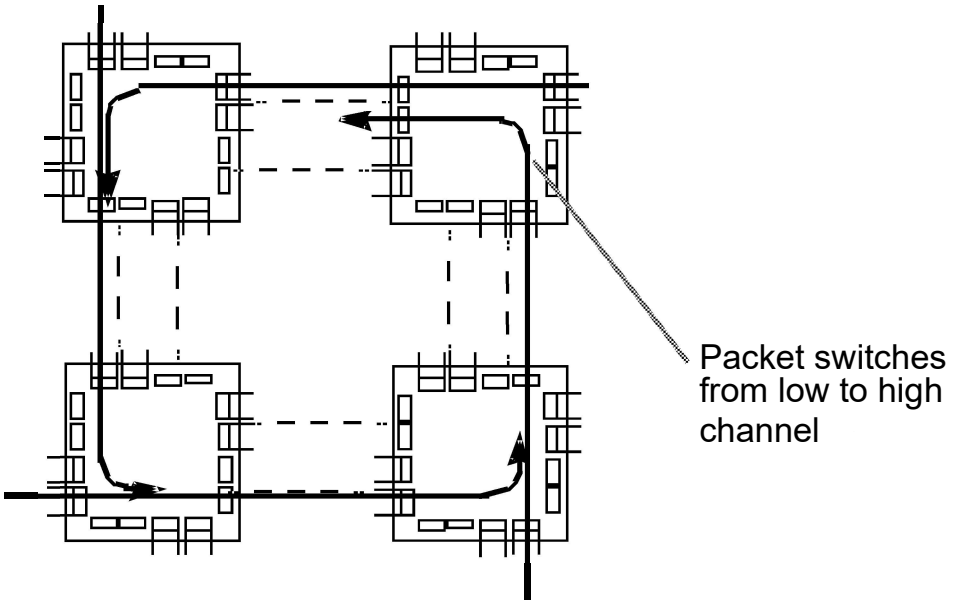
Reserve one packet buffer on each channel for messages destined for nodes with a higher number than their source, i.e., messages that don't use wraparound channels.

Now such messages will always be able to make progress.

Wraparound messages may be delayed, but the network will not deadlock.

Notice that this requires no more links or switches, just more buffers.

How can we break the four-message deadlock?



Let's provide two virtual channels per physical channel, as shown above.

- Messages at a node numbered higher than their destination are routed on the "high" channels.
- Messages at a node numbered lower than their destination are routed on the "low" channels.

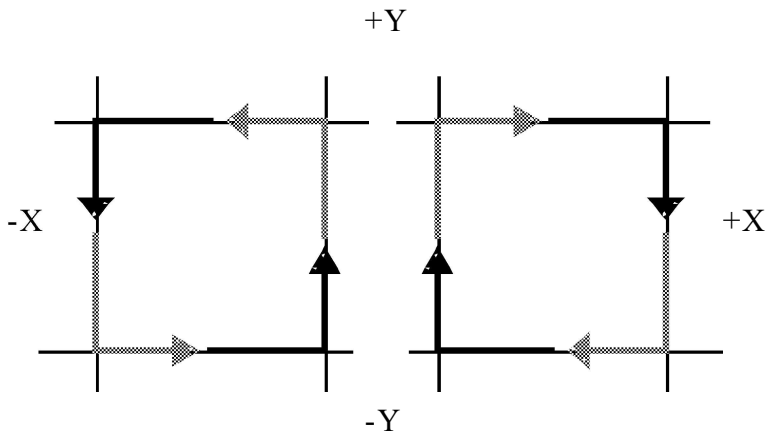
This breaks the dependence cycle.

Example: In the *k*-ary 2D array above, let's assume we have messages $1 \rightarrow 4$, $4 \rightarrow 12$, $12 \rightarrow 10$, and $10 \rightarrow 1$. Show that all messages can make progress.

This strategy can be generalized to *d* dimensions.

Turn-model routing

Note that *x-y* routing in dimension order allows only four of the eight possible turns a packet might make on its way to its destination.

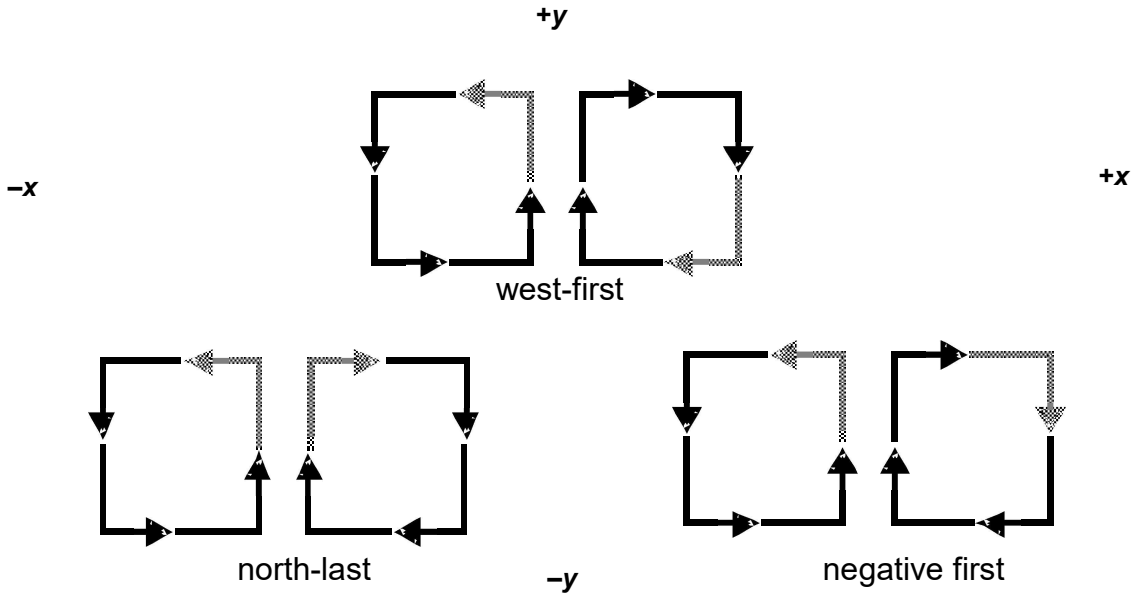


When a packet is traveling in the $\pm x$ direction, it is legal to turn into the $\pm y$ direction, but once it is traveling in the $\pm y$ direction, it can make no further turns.

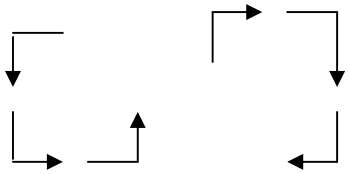
Intuitively, we could prevent deadlock by disallowing only one turn in each cycle.

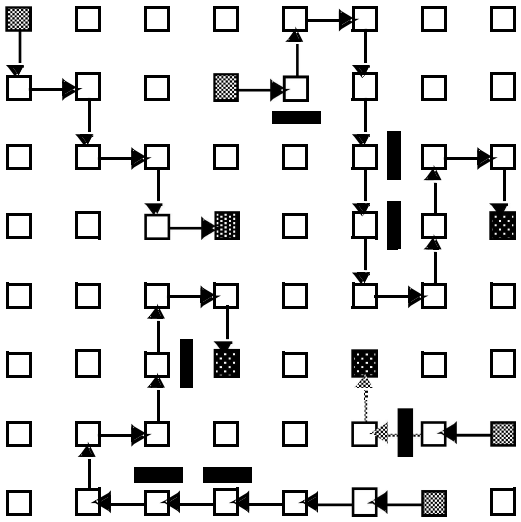
It turns out that of the 16 possible ways to prohibit two turns in a 2D array, 12 prevent deadlock.

These consist of the three algorithms below, and rotations of them.



Can you see that this “algorithm” does not prevent deadlock?





Each of these algorithms allows nonminimal routes to be followed. On the right are some legal west-first routes.

Exercise (on paper): Give one other example of a legal turn-model routing algorithm, and one that is not deadlock free.

Adaptive routing

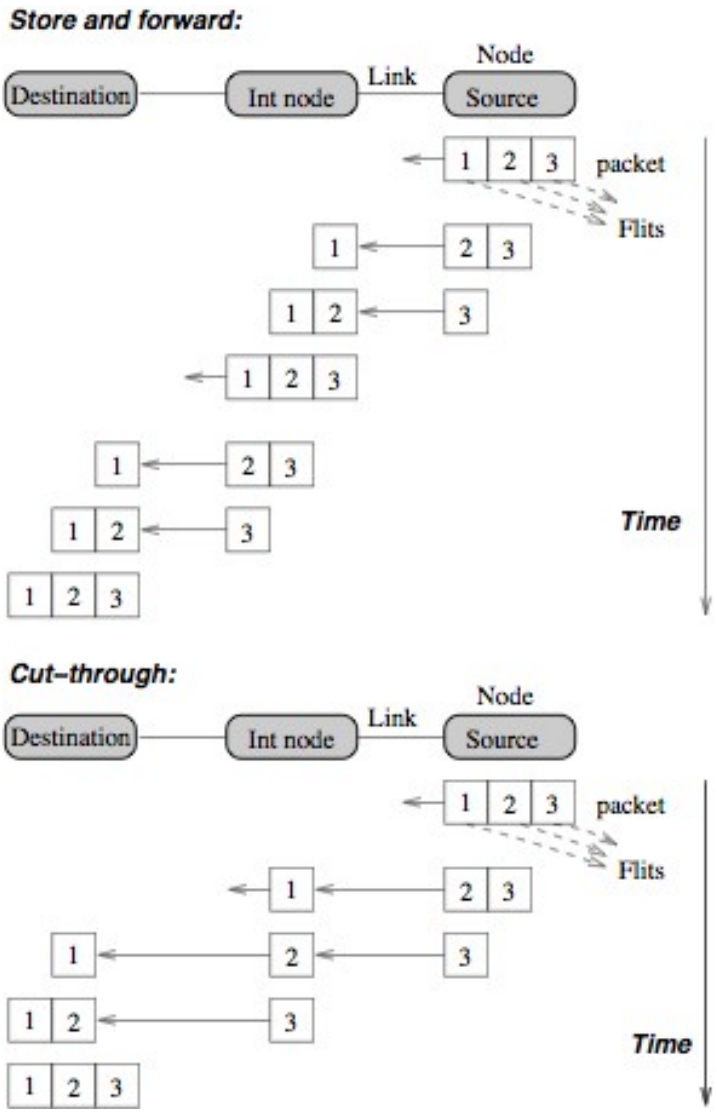
Adaptive routing has several advantages.

- If there is only one route from source to destination, failure of a single link can leave the network disconnected.
- It can allow messages to avoid regions where there are long queues and a lot of contention.

One interesting adaptive algorithm is “hot-potato” routing.

A switch never buffers packets. If > 1 packet is destined for the same output channel, the switch “misroutes” all but one.

Store and Forward vs. Cut-Through



Early networks used store-and-forward routing:

A packet was sent from one node to an intermediate node and buffered there.

But this took a lot of time:

$T = h \times (T_{switch} + T_{xmit})$

For the last 25 years or so, cut-through routing has taken over.

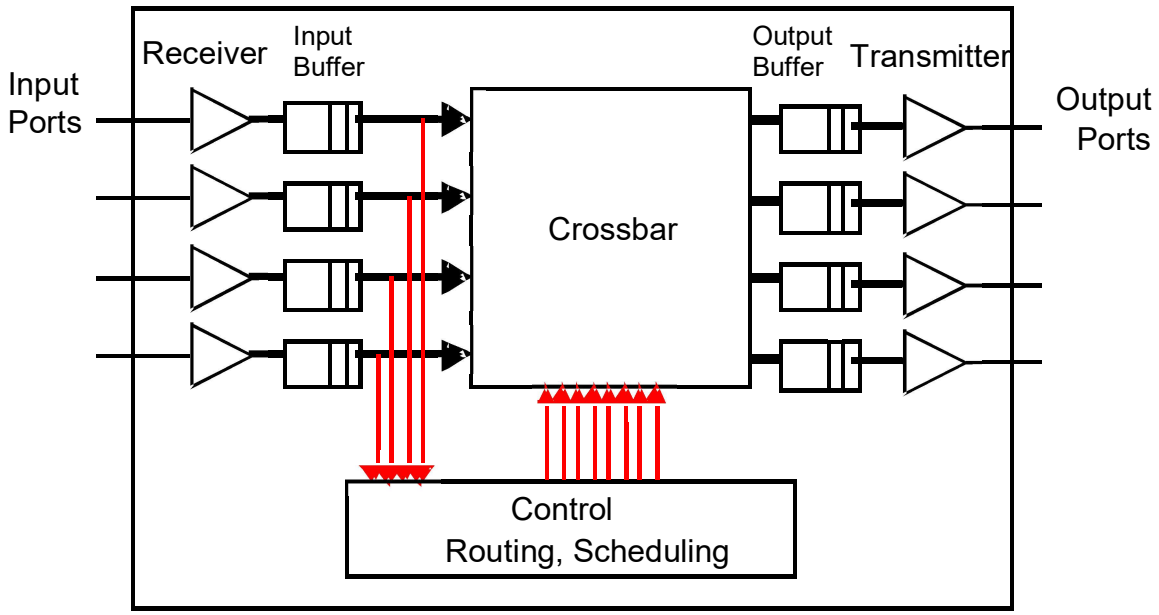
Effectively, a pipeline is set up, and flits are sent through, one after another, and are switched without being buffered.

$T = h \times$

Switch Design

[§12.4] In earlier lectures, we have seen that switches in an interconnection network connect inputs to outputs, usually with some kind of buffering.

Here is a basic diagram of a switch.



Usually, the number of input ports is the same as the number of output ports. This is called the *degree* of the switch.

Each input port includes a receiver. A received message goes into an input buffer.

Each output port includes a transmitter to drive the link. There may also be an output buffer.

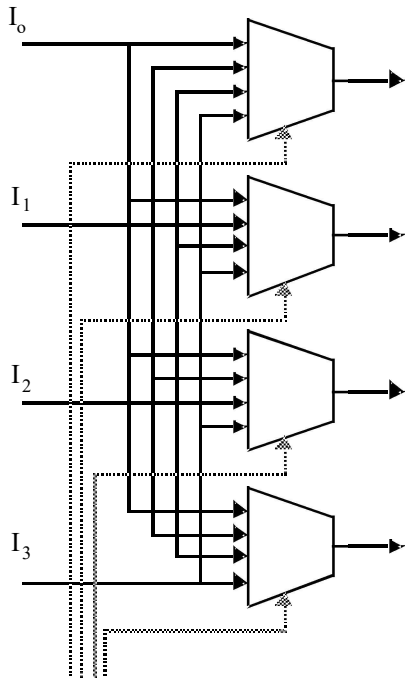
The control logic implements the routing algorithm. It must include a way to arbitrate among competing requests for the same output port.

A major constraint on switch size is the number of pins. How would this be determined?

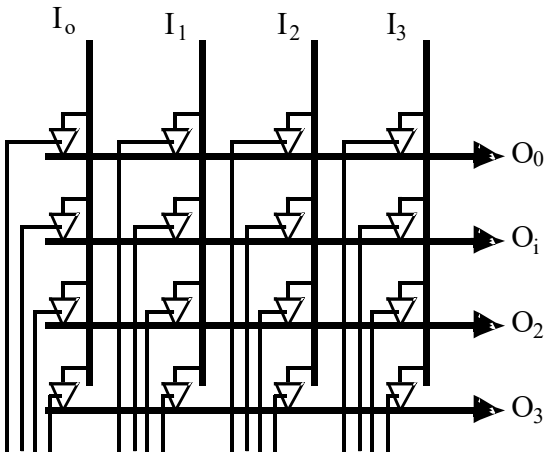
This tends to favor fast serial links. They use the least pins and eliminate problems with skew across bit lines in a channel.

With parallel links, one of the wires is essentially a clock for the data on the others.

Logically, the crossbar in an $n \times n$ switch is just an n-way multiplexer associated with each dimension (at right).



In VLSI, it is typically realized as a bus with a set of n tristate drivers (below).



An increasingly common technique is to use memory as a crossbar, by writing for each input port and reading for each output port. Explain.