8

The On-chip Network

Because of the relentless improvement in transistor technology predicted by Moore's law, processors from 1965 onwards steadily kept on getting more complex. Their performance kept on increasing, and this was the main driver of efforts in the computer architecture community. However, by 2005 it was clear that single core performance will not increase anymore. The signs of saturation were visible on the horizon. There were two big reasons for this. The first was that power dissipation was becoming a very important factor. It was becoming very hard to limit the on-chip power dissipation and consequent temperature rise. As a result, the cores had to be made simpler. The second reason was that the phenomenon of diminishing returns had set in. We were already exploiting as much of ILP as we could, and the benefits of increasing the issue width or investing in creating better predictors was marginal. As a result, the efforts centered around increasing the on-chip cache size. Caches increased in complexity, and very soon became large multi-banked caches, where the timing was dominated by the time it takes to reach the banks that are the farthest away in terms of distance. This also placed a limit on the performance gains.

Over the next few years cores became simpler and the attention shifted to parallel processing. Multicore processors that arose out of this effort have gradually created the space for manycore processors, where the cores are even simpler and more numerous.

As a result of these trends, a modern processor chip is a mix of 10-20 cores, and an equivalent number of cache banks. Additionally, there are other active elements such as memory controllers and specialized accelerators. The typical layout of a manycore processor die looks like one of the sub-figures in Figure 8.1. The cores and cache banks are either organized as a chess board, or the cores are on the rim and the cache banks are in the center.

The memory controllers are always located on the periphery. Dedicated accelerators can either be in the center if their primary job is to perform computations or can be towards the periphery if they need to communicate a lot with memory.

All of these elements inclusive of cores and cache banks need to send messages between each other while executing a program. For example, if core A wants to read a memory location that is in cache bank B, then A needs to send a read-miss message to B. Subsequently, B needs to send the value stored in the cache line back to A, which is another message. We can have different types of communication between the nodes. To manage all of these senders, receivers, and data packets, we need to implement an on-chip network. The on-chip network or NoC (network on-chip) needs to route messages from senders to receivers. Unlike the internet, **we are not allowed to drop messages** in an on-chip network. As a result, we need to have complex protocols to ensure timely and reliable message delivery.

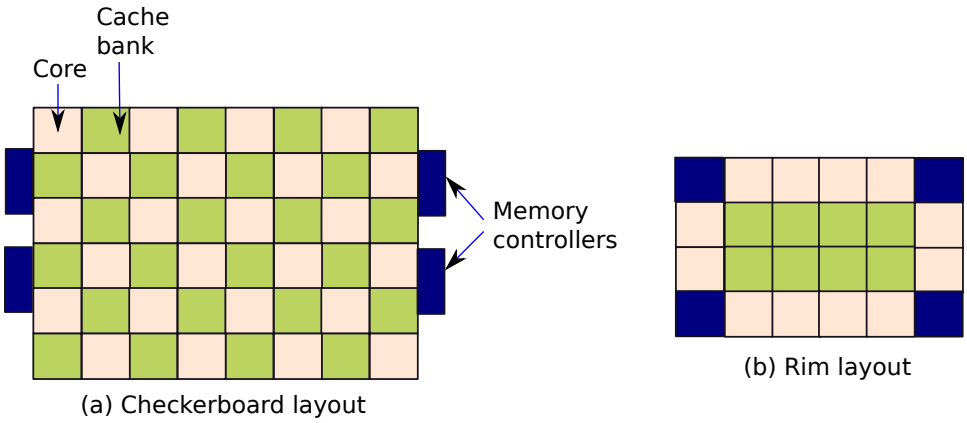


Figure 8.1: Different kinds of multicore layouts

Definition 51
An on-chip network (or network-on-chip or just NoC) is a network that connects cores, cache banks, accelerators, and memory controllers within the chip.

We can think of a processor as a small city, where instead of cars, network packets flow between the cores and cache banks. Akin to a city, we can have traffic jams, queuing, and congestion. The same way that we have a system of lights, and traffic police to manage traffic in cities, we need to have something similar in the form of protocols to manage the on-chip traffic. Additionally, there are other problems that also happen in modern cities. In situations with heavy traffic, it is possible that we can have a gridlock where cars are not able to move because a circular dependence forms between the cars. Car A needs car B to move, car B needs car C to move, and finally car C needs car A to move. Such circular dependences can lead to long jams and gridlocks. We can have a similar situation in an NoC as well. We can have *deadlocks*, where a set of packets simply cannot reach their destination because a circular loop of dependences forms among them. We also can have *starvation*, where a given packet is not able to reach its destination because other packets are causing it to stall. We can also have the case where the packet moves round-and-round in circles and never reaches its final destination – this is known as a *livelock*.

Along with solving the issues of deadlock and starvation, we need to ensure that we are able to maximize the throughput of the network, and minimize the average or worst-case latency. All of these are difficult problems, and to solve these problems a full field of on-chip networks has emerged over the last few years. In this chapter, we shall look at some key results in this field and motivate the student to study further.

8.1 Overview of an NoC

Let us quickly summarize all our learning till now. We have made the case for an NoC by observing that today's multicore processors have a multitude of cores and cache banks. A core or cache bank can communicate with another core or cache bank. To support all possible communication patterns, it is necessary to design an NoC that can support a high throughput and also have a low latency.

8.1.1 Nodes and Links

Let us start with creating an abstract model of the system. Let us model an NoC as a graph (defined in Section 2.3.2) that has a set of nodes and a set of edges. A *node* is a generic and abstract component that can send or receive a message. An *edge* is a communication channel between two nodes. A node can be attached to a core, a cache bank, a memory controller, I/O controller, or any other component that is capable of transmitting or receiving a message. Let us define a message as a set of bytes. We shall look at it in detail in Section 8.2.1. For the time being, it is a set of bytes that needs to be atomically sent from the sender node to the receiver node. An *atomic delivery* implies that either the entire message is delivered or nothing is delivered. The sender and the receiver know how to understand the message.

Given that almost all the major components within a chip need to communicate with each other by sending messages, we need not have separate communication systems for different components. All of them can use a generic communication unit that sends and receives messages – this can be thought of as a node in the NoC. A realization of such a generic communication unit or a node is called a *router*. A router is always attached to a component such as a core or a cache bank. Nowadays, a few adjoining cores and cache banks are typically grouped into a *tile*; there is one router per tile. It transmits and receives messages on its behalf. Additionally, the routers coordinate among themselves to send messages on the network. Henceforth, when we shall refer to a *node*, we will actually be referring to a router in the NoC. These terms might be used interchangeably as well.

Definition 52
A router is a generic communication unit in an NoC. Every component that wishes to communicate using the NoC needs to have access to a router that sends and receives messages on its behalf. In addition, in modern networks a message is sent from a source to a destination by passing it from router to router. The routers cooperate and coordinate among themselves to deliver the message at the final destination.

Unlike connections of yesteryear where all the routers were connected to a single set of copper wires (known as a bus) this method does not scale for modern NoCs. In modern NoCs, the connections are one-to-one, which means that every copper wire is connected to only two nodes. Such connections, known as *links* (or *edges* in the graph), connect a pair of nodes. We can have two types of links: buffered and unbuffered.

Buffered and Unbuffered Links

Assume we have a wire of length l. Its delay is equal to κl^2, where κ is a constant of proportionality (refer to Section 7.3 for the derivation). If the delay is a quadratic function of the length of the wire, then it can become very large. Let us instead split the wire into segments of fixed length, and insert a buffer between consecutive segments. The buffer is a latch or flip-flop that simply reads the bits sent on its incoming link, and then sends them out on its outgoing link. Such buffers are also called *repeaters*. Assume that the delay of a repeater is d.

We claim that by splitting a long wire into segments, and by introducing repeaters, we can reduce the overall delay of the wire. Let us do the math.

Let s be the number of segments, where the length of each segment is l/s. We shall thus require $s-1$ repeaters. The net delay D is given by

$$\begin{aligned} D &= (s-1)d + s \times \kappa \frac{l^2}{s^2} \\ &= (s-1)d + \kappa \frac{l^2}{s} \end{aligned}$$

Let us now find the optimal value of s.

$$\begin{aligned}\frac{\partial D}{\partial s} &= d - \kappa\frac{l^2}{s^2} = 0\\ &\Rightarrow s^2 = \kappa\frac{l^2}{d}\\ &\Rightarrow s = \sqrt{\frac{\kappa}{d}}\cdot l\end{aligned}$$

Thus, the optimal value of the number of segments s is $\sqrt{\frac{\kappa}{d}}\cdot l$. The optimal delay is given by

$$\begin{aligned}D &= (s-1)d + \kappa\frac{l^2}{s}\\ &= \left(\sqrt{\frac{\kappa}{d}}\cdot l - 1\right)d + \kappa\frac{l^2}{\sqrt{\frac{\kappa}{d}}\cdot l}\\ &= \sqrt{\kappa d}\cdot l - d + \sqrt{\kappa d}\cdot l\\ &= 2\sqrt{\kappa d}\cdot l - d\end{aligned} \tag{8.1}$$

The important point to observe in Equation 8.1 is that the delay is now a linear function of the length of the wire. As a result a repeated or a buffered wire is significantly faster as compared to a wire that does not have repeaters. Most long wires, also called *global wires,* in the chip are buffered. Such repeaters sadly do not come for free. They have an associated area and power cost. Hence, we do not use them for interconnects that are over short distances. Such interconnects are called *local wires*.

8.1.2 Network Topology

Let us now look at the network topology: the way the nodes and links are laid out. In most books on computer architecture, the authors spend a lot of time discussing different kinds of network topologies including their mathematical properties. However, in modern processors, the network topology is typically very simple. If the number of cores is limited to four, then we often have a bus (see Figure 8.2), which is at its core a set of parallel copper wires. Unfortunately, buses have severe limitations in terms of scalability and bandwidth. All the routers compete for the same bus and this causes a lot of contention. As a result, buses are typically not considered good candidates for networks in large chips.

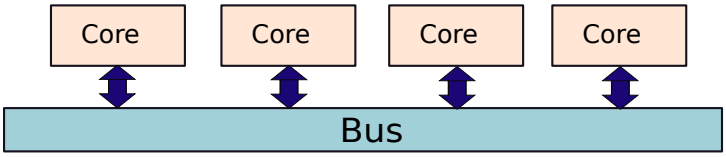


Figure 8.2: A bus connecting four cores

Hence, for large chips with a lot of cores and cache banks, the most common topology is a mesh (2D matrix) or a torus (2D matrix with the ends of each row and column connected). It is possible to have more complex high-radix structures that have more than 4 incoming and outgoing links per router; however, this is rare. High-radix structures have many links per node such as Clos networks and hypercubes [Sarangi, 2015]. Additionally, they have additional properties such as immunity to multiple link failures, which is not of particular concern in on-chip networks. Such topologies are thus more commonly used in large cluster computers.

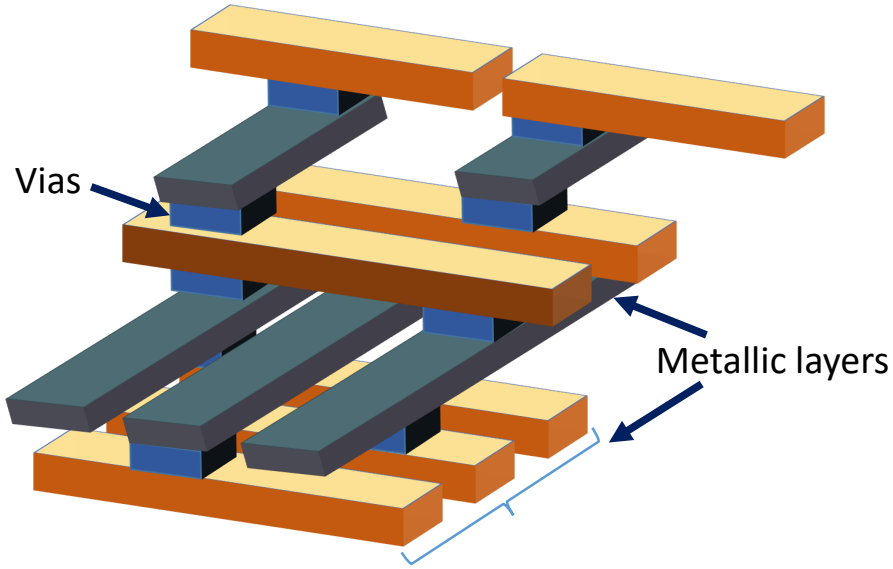


Figure 8.3: Metal layers in a modern chip

Metal Layers in a Modern Chip

In a modern VLSI chip we need to have a complex network of wires between communicating nodes. A simple bus is not enough. If we have an elaborate network, then the wires will intersect each other. However, if two copper wires intersect each other, then they do not remain separate wires anymore. A conducting path forms between all the senders and the receivers. As a result, we need to design a more complicated structure where we have several layers of wires such that all the wires that we want can be placed on the chip.

To realize this goal, a VLSI chip is composed of multiple layers. The lowest layer is made of silicon. On this layer we create all the transistors. However, this layer does not have enough space to connect all the sender-receiver pairs with copper wires. As a result there is a need to create additional layers on top of this layer that exclusively contain wires. These are known as the *metal layers*. Modern chips as of 2020 have 10-15 such metal layers, where consecutive layers are separated by an insulating layer that is made of silicon dioxide (SiO_2). The lower metal layers have local wires and the higher layers have global wires. Each layer is made of a dielectric material with a low dielectric constant such that the capacitance between wires is reduced to a minimum: this reduces crosstalk noise. In such layers, we can fabricate a wire by creating a trench in the layer and depositing copper in it. We can thus think of a wire as something like a filled trench in the layer. Modern VLSI processes allow us to create thousands of wires on a layer in this fashion.

This is shown in Figure 8.3, where we can see small wires in different layers. Let's say that we need to connect a functional unit A with a functional unit B. Then it is necessary to connect them with a wire. It might not be possible to connect them with a wire that fully resides on the first metal layer. We thus need to create small segments of wires in different layers, and connect them with vertical links made of copper. These vertical links (see Figure 8.3) are known as vias, or trans-silicon vias (TSVs). Hence, a long wire between a source and a destination can traverse through multiple metal layers and vias.

Further, notice that in Figure 8.3 the wires in consecutive layers are perpendicular to each other. This is a standard wire routing technique. If we assume that the wires in one layer are oriented along

the x-axis, then in the layer above it, they are oriented along the y-axis. This automatically ensures that all the wires in a layer do not intersect with each other. Secondly, if we have enough space, two layers are enough to connect any sender and receiver. However, this does not happen in practice because we run out of space in layers and thus 10-15 metal layers are required.

For the sake of completeness it is necessary to mention that every chip has three additional layers: power, ground, and clock. The power and ground layers are arranged as a grid. They are connected to the supply and the ground terminals respectively. Another layer that does not have any role in signal routing is the clock layer, where the external clock signal is distributed to all the functional units. The clock distribution network is typically arranged as an H-Tree (see Section 7.3.2). Recall that an H-Tree ensures that the distance from the source (located at the center), and each of the receivers is the same. This ensures that all the users of the clock receive the transitions in the clock signal at almost the same time, which leads to minimal clock skew: difference in the time of arrival of the clock signal across different functional units.

Let us summarize.

Way Point 8

- *A simple bus is not enough for connecting multiple cores and cache banks in a modern chip – there is a lot of contention.*
- *We need to have complex interconnections between the communicating routers.*
- *It is thus necessary to create multiple metal layers, where the layers close to the silicon are for local wires, and the layers farther away contain longer interconnects known as global wires.*
- *A connection from a given source to a destination might have segments in different layers.*
- *Copper wires across different layers are connected by trans-silicon vias (TSVs).*

Let us now use a multi-layer VLSI chip to create more complex interconnections.

Mesh and Torus

Figure 8.4 shows the mesh topology. A mesh is a simple 2D matrix, where we connect two adjacent nodes on the same row or the same column with a link.

The mesh is a very popular structure in on-chip networks because it is very easy to create. It uses rectilinear links that are either horizontal or vertical. Fabricating such structures is very easy in modern VLSI processes. In comparison, it is fairly difficult to fabricate wires that are at oblique angles (neither vertical nor horizontal). Most VLSI fabrication processes do not allow designers to create such wires.

One of the major problems with mesh networks is that the network diameter – maximum delay between two nodes if we are following the shortest path between them – is high. If we are considering an $N \times N$ mesh, then the diameter is $2N - 2$ (distance between two diagonally opposite corners). This is measured in terms of the number of links that we need to traverse. Let us further reduce this. We can use the torus topology in this case (refer to Figure 8.5).

The only addition is the long wires between the ends of reach row and the ends of each column. They effectively reduce the diameter. In this case, the nodes that are the farthest apart are the center and any of the corners. The diameter is $N/2 + N/2$ $(= N)$. Note that the assumption is that N is even. We have thus reduced the effective diameter roughly by a factor of 2. However, we have also increased the number of wires and introduced a few very long wires that span the length of the chip. Such long wires

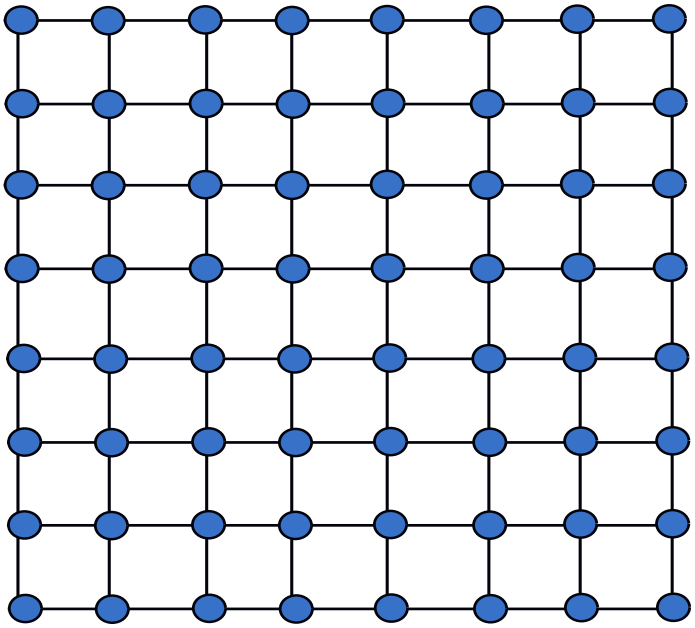


Figure 8.4: The mesh topology

will have very large delays and may not give us any significant advantage. Thus, most designs use the folded torus design shown in Figure 8.6.

Note that the design shown in Figure 8.6 is equivalent to the torus shown in Figure 8.5. However, the connections have been made differently. A node in a row is not connected to the node that is directly adjacent to it (in the next column). Instead, it is connected to the node (on the same row) that is two columns away. We have a similar connection pattern for the nodes in each column. Even though such designs increase the timing delay between adjacent nodes by a factor of 2; however, they eliminate long wires completely.

High-Radix Networks

Each node in a torus or a mesh is connected to 2 to 4 other nodes. These are examples of low-radix networks, where each node is connected to a few other nodes. In comparison, if we increase the number of links per node, we shall have a high-radix network. Such networks have some favorable properties such as a low diameter and higher path diversity. The term "path diversity" refers to the diversity of paths between a given source-destination pair. The advantage of increased path diversity is that we can react better to network congestion. Let us consider some common high-radix networks that are commonly used in cluster computers. They are difficult to fabricate using current VLSI technologies that do not allow oblique wires. However, there is some ongoing research that focuses on using these networks in NoCs by creating versions of the network that can be fabricated with current technology. In such networks we rearrange the nodes on a 2D plane such that it is easy to route wires between them. These are known as *flattened networks*.

Hypercubes

Figure 8.7 shows the hypercube topology. A hypercube refers to a family of network topologies that are constructed recursively. An order 0 hypercube H_0 is a single node. To construct an order 1

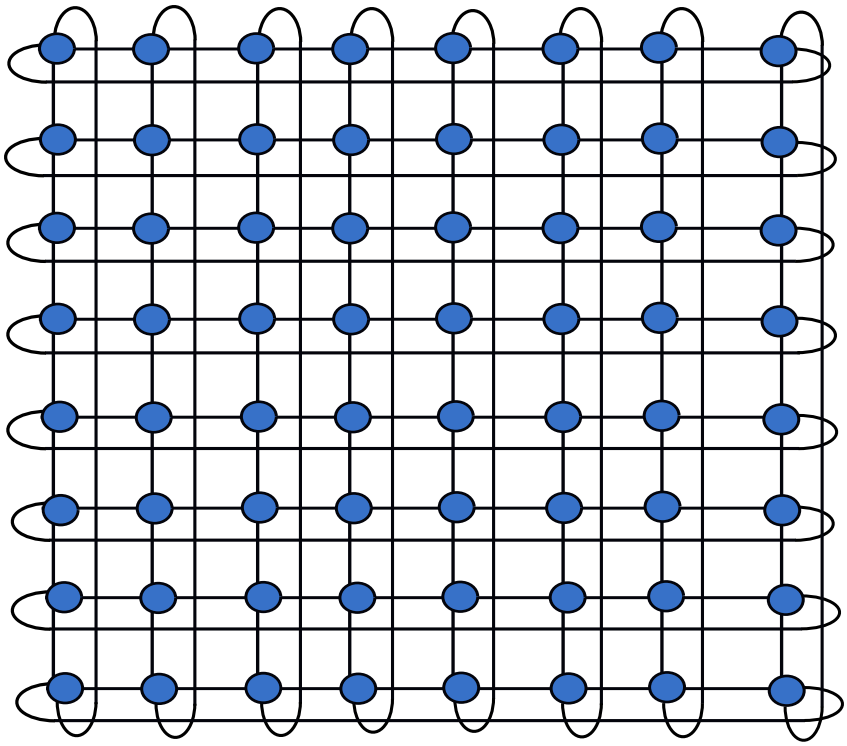


Figure 8.5: The torus topology

hypercube H_1 we take two copies of order 0 hypercubes and connect them together (see Figure 8.7(b)). Similarly, to create H_2 we take two copies of H_1 and connect the corresponding nodes together. Observe that we can number the nodes as binary numbers. For example, the nodes in H_2 can be numbered 00, 01, 11, and 10 respectively.

Now, to construct H_3 we take two copies of H_2, and connect the nodes with the same numbers with each other. For example, we connect the nodes numbered 00 in each hypercube with each other. We then add a prefix the nodes in one copy of H_2 with a 0 and the nodes in the other copy of H_2 with a 1. The numbers for the nodes labeled 00 in the two copies of H_2 become 000 and 100, respectively, in H_3. We follow the same process for the rest of the nodes. On similar lines, we can create H_4, H_5, and so on.

Let us now summarize some properties of this network, which are also easy to derive. In a hypercube with N (power of 2) nodes, each node is connected to $log_2(N)$ other nodes. This is easily visible in Figure 8.7 where we see that in a hypercube with N nodes, each node is labeled with a $log_2(N) - bit$ binary number. When we traverse a link, we flip only one of the binary bits. Given that we can change any one of the $log_2(N)$ bits in the label, it automatically follows that every node has $log_2(N)$ neighbors.

We can extend this result to prove that the diameter of a hypercube is $log_2(N)$. Consider two nodes with labels L and $\bar{L}$ (bitwise complement of L) respectively. To send a message between the nodes we need to traverse a sequence of links. In each traversal we flip a single bit in the label. Given that all the $log_2(N)$ bits differ between the labels of the nodes, we need to flip (complement) all the bits, and this means that we need to traverse $log_2(N)$ links. Note that the diameter is not more than $log_2(N)$ because the Hamming distance (number of corresponding bits that differ) between two labels is limited to the size of the labels, which is $log_2(N)$ bits. Since in every link traversal, we complement a single bit, we will never need to perform more than $log_2(N)$ traversals if we are proceeding on the shortest path.

Clos Network

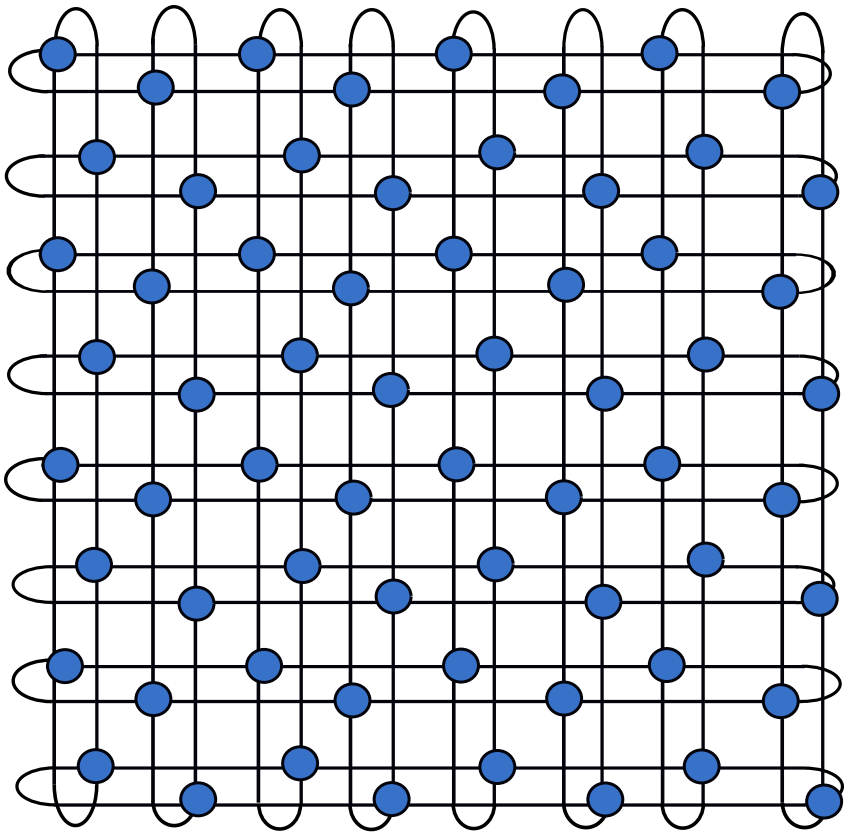


Figure 8.6: The folded torus topology

In Figure 8.8 we show a Clos network. Typical three-layer Clos networks are traditionally described using three parameters: n, m, and r. The first (leftmost) layer or the ingress layer contains r $[n \times m]$ switches. An $n \times m$ switch has n inputs and m outputs. It implements an all-to-all connection where any input can be connected to any output. However, the caveat is that at any given point in time any output of the switch can be connected to only one input, and any input can be connected to only one output.

In Figure 8.8 we show an example with $n = 4$, $m = 4$, and $r = 3$. The ingress layer accepts the input messages. The messages are then sent to switches in the middle layer. The middle layer contains m $[r \times r]$ switches. Note that there is a pattern in the interconnections. Consider a switch in the ingress layer. It has m outputs. Each of these outputs is connected to a different switch in the middle layer.

Finally, consider the third layer or the egress layer. This consists of r $[m \times n]$ switches. Each of the outputs of a switch in the middle layer is connected to a different switch in the egress layer. The output terminals of the egress layer are the outputs of the entire Clos network.

We thus have a total of nr inputs and nr outputs. Note that any message on any input terminal can be routed to any output terminal in the Clos network. Furthermore, it is possible that the input and output terminals might be connected to the same set of routers. This means that we can have nr routers, where if they need to send a message, they drop a message at the corresponding input terminal of the Clos network. Similarly, we can connect each router to an output terminal of the Clos network. If we connect the routers in this fashion, then the Clos network ensures that we can send a message from any router to any other router. Such a network is known as the folded Clos network.

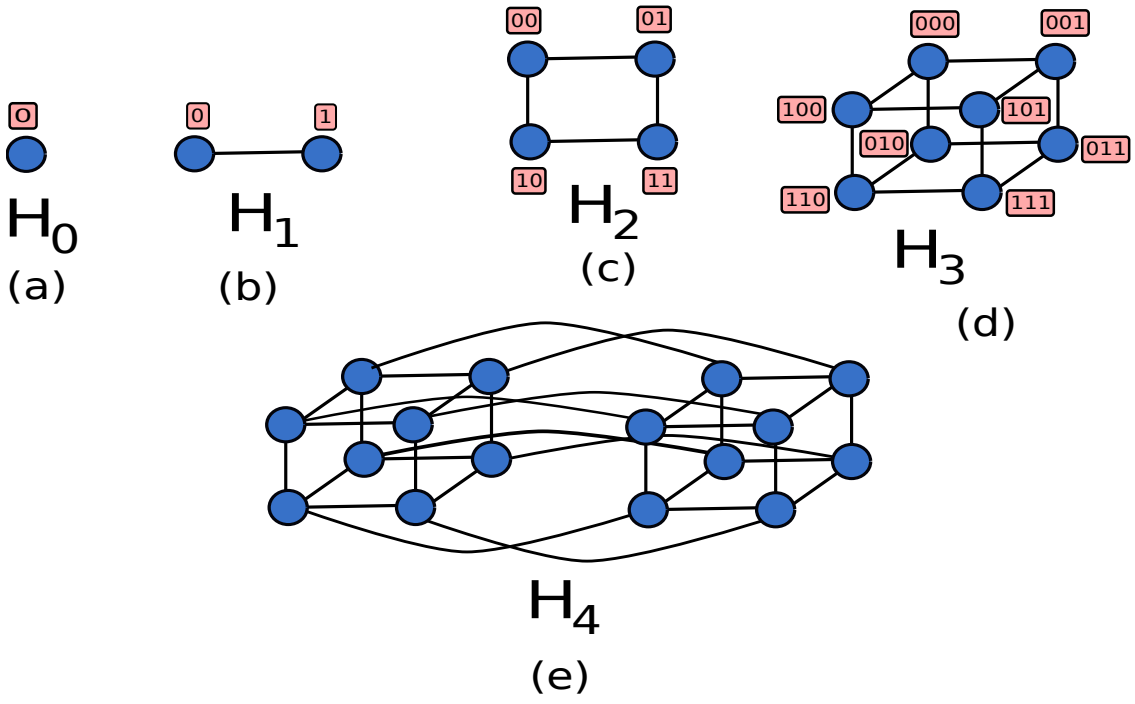


Figure 8.7: The hypercube topology

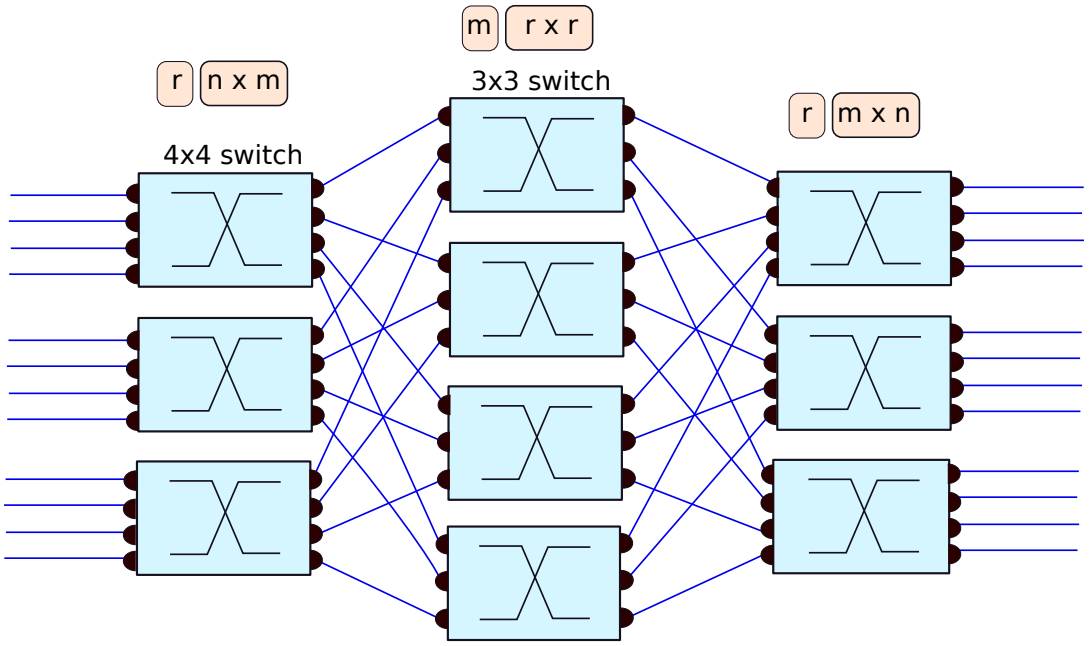


Figure 8.8: The Clos network

Clos networks have some more beautiful properties. Let us quickly list them without going through the proofs. Interested readers can refer to [Clos, 1953].

1. If $m \geq n$, we can always connect an unused ingress terminal with an unused egress terminal by rearranging the rest of the connections. We will not encounter a case where either the new message transfer or any of the existing message transfers have to be terminated because some switches and terminals along the way are fully busy.

2. If $m \geq 2n - 1$, we can always connect an unused ingress terminal with an unused egress terminal to send traffic without rearranging the rest of the connections between the network's input and output terminals.

Butterfly Network

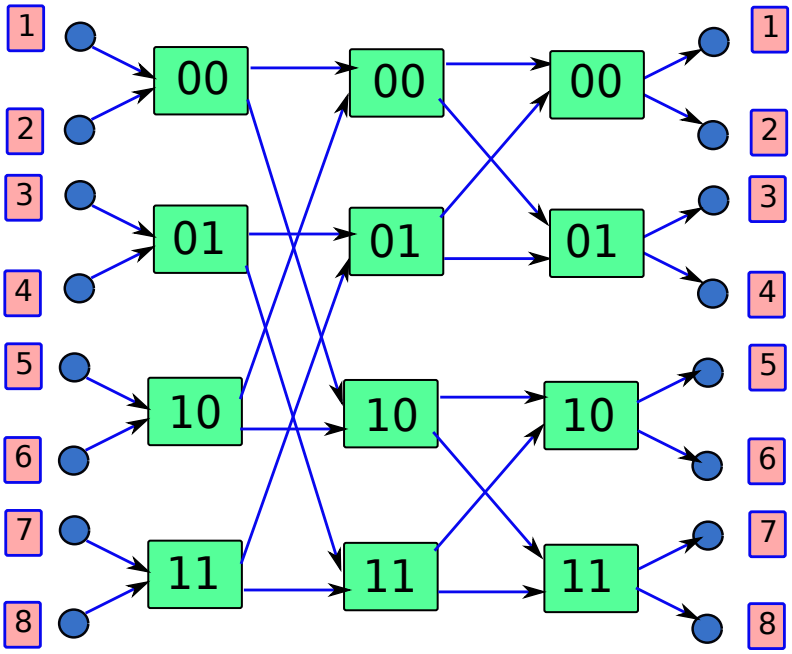


Figure 8.9: The Butterfly network

In the Clos network we have different kinds of switches with different numbers of input and output terminals. Fabricating such heterogeneous structures represents a challenge. Hence, homogeneity is preferred. Variants of the Clos network exist that have multiple intermediate layers with this property. A famous network in this class is the Benes network ($m = n = 2$) that uses only 2×2 switches. Note that in this case we are creating a low-radix network out of a high-radix network.

Let us look at a similar network called a Butterfly network that uses low-radix switches. High performance implementations of Butterfly networks can use high-radix switches; however, they are not covered in this book. Interested readers can refer to [Kim et al., 2007].

The design of a basic Butterfly network is shown in Figure 8.9. The network has N input and N output terminals, and $log_2(N)$ layers of 2×2 switches. Consider the leftmost layer of switches. Each switch has two input terminals, which are connected to two nodes respectively. In our example network, we have 8 nodes. The first layer of switches decide whether the destination lies in the set of first four

nodes (1-4) or the set of last four nodes (5-8). The message is routed accordingly. The next layer does another round of filtering. They look at a set of 4 nodes and divide it into two halves. Then the message is routed to the right set of nodes – first half or second half. Finally, the third (rightmost) layer of switches route the message to one of the two destination nodes.

Note that the input and output nodes can be the same. This is similar to the way we connected them in the folded Clos network. Such a topology is also known as a *folded* Butterfly network. Note that as compared to the Clos network, a Butterfly network lacks path diversity. For a given input-output pair of terminals, there is a single path. If we compare this with a torus, then the latter seems to be more efficient. However, let us compare the number of links. For a torus with N nodes we have $2N$ links. However, for a Butterfly network with N nodes we have $(N + Nlog_2(N))$ links: we have $log_2(N)$ levels with $N/2$ switches each. Given that we have more links for large N, the effects of congestion are reduced. In addition, the diameter is $log_2(N) + 1$, which is significantly better than the diameter in tori[1], which is roughly N.

8.2 Message Transmission

Let us look at methods to send messages in a network from a given source to a given destination. To keep the description simple, we shall assume a mesh based topology in this section. The results are generic and hold for all kinds of topologies.

8.2.1 Basic Concepts

Let us first describe some basic concepts. Assume that the sender can send any number of bytes to the receiver. A sequence of bytes that forms one logical unit is called a *message*. Higher protocol layers operate exclusively at the level of messages.

However, at the level of the NoC, a single message might be too big to handle in entirety. We thus divide it into a set of fixed size *packets*. For the purpose of transmission in an NoC, a packet is a consolidated unit. This means that all the bits in the packet flow along the same path; we do not send different parts of a packet along different routes. Moreover, note that in general in an on-chip network we do not drop packets, as we do in conventional networks. However, there are some rare examples of on-chip networks where packets are occasionally dropped, and then we have two options: either we retransmit the packet or the entire message.

The size of packets also can be fairly large for on-chip networks. It is thus necessary to break a packet into units of information that the routers can treat as a basic unit for the purpose of storage and transmission. We thus divide a packet into several flow control digits, referred to as *flits*. As of 2020, most on-chip networks use 8 or 16-byte flits. A packet can be 64 to 128 bytes long; it thus contains a sequence of flits. The first flit in the packet is known as the *head flit*. Subsequently, we have a sequence of *body flits*, and the last flit in the packet is known as the *tail flit*. The head flit typically contains the id of the destination router, details of the route (if it has been precomputed), and other information of interest that we shall describe in subsequent sections. The routers analyze the contents of the head flit and compute the routing information. The subsequent body and tail flits follow the same route as the head flit. It is never the case that different flits of the same packet are sent along different routes. This will make the design of the entire on-chip network very complex and this level of complexity is not desirable from the point of view of power consumption. Additionally, the area overhead of routers will become prohibitive because we need to keep track of a lot of additional state.

The flits are physically sent on copper wires that connect two routers. To send a 64-bit flit, we would ideally like to have 64 parallel copper wires between the two routers. However, this is often not desirable with long high-speed links. This is because it is hard to ensure that electrical signals are synchronized across the wires. It is possible that all the signals may not arrive at the same time at the receiver. There

[1] plural of torus

might be tiny mismatches in the lengths of the wires, or due to aging, the RC delays of the wires might change over time. We need circuits to compensate for this drift in timing. The typical approach that is used is that the data transmission is synchronized with respect to a clock signal. The maximum possible delay across wires is thus limited to some fraction of the clock cycle period. With wider links it becomes more difficult to design such circuits. Hence, the only option that remains is to reduce the bandwidth and the transmission rate. This is why most long, high-speed links are serial nowadays: send signals using a single wire.

However, these links are typically several centimeters long, whereas we are talking of links that are limited to a few hundred microns long. In this case, we can afford a limited-width parallel link. For example, we can have a link with 16 or 32 wires. Let's say that we have 32 wires, and the flit size is 8 bytes (= 64 bits). In this case, we need to make two successive transfers: 32 bits each. Each such group of 32 bits is known as a *phit* (physical digit). Therefore, in this case a flit consists of 2 phits. Note that it is necessary to transmit all the phits in a flit consecutively on the link. A flit cannot be split across routers. The routers do not recognize phits. They expect full flits to be transmitted and received. They have small circuits that do the job of breaking down flits into phits, and reconstructing them.

Definition 53

Message *A message is a stream of bytes that makes sense at the level of the application.*

Packet *A message is divided into a sequence of packets where the NoC treats a packet as a consolidated unit. All the bytes in the packet follow the same path.*

Flits *Packets are further subdivided into flits of the same size. All the flits in the packet follow the same route. Additionally, each router has buffers, where each entry can store a single flit.*

Phits *Due to limitations in the signaling technology, it is not possible to have very wide links between two neighboring routers. We thus have narrow links and send a flit over multiple clock cycles. We thus divide a flit into multiple phits, where a phit (physical digit) represents the set of bits that are sent in a single clock cycle.*

Let us summarize.

Way Point 9

1. *We have discussed messages, packets, flits, and phits. They need to be sent from a sender to a destination through a sequence of routers.*
2. *Each router needs to read in the message, a flit at a time, temporarily store the flits, and forward them on its outgoing links to a neighboring router.*

Basics of Flow Control

To send a message from a source to a destination, it is necessary to reserve resources along the way. These resources are buffer space in the routers, and the permission to transmit along the links that comprise the path from the source node to the destination node. Note that in this section we shall

use the terms *node* and *router* interchangeably to denote a network entity that can send, receive, and forward flits.

There is thus a need to set up a path between the source and the destination, which basically involves allocating and deallocating buffer space on the way. There are three ways that this can be done. We can either do it at the granularity of the entire message, or at the granularity of a packet, or at the level of flits. This process is known as *flow control*.

Definition 54
Management of the flow of flits between nodes in an NoC is known as flow control. It often deals with reserving buffer space in routers and reserving the right to transmit flits.

8.2.2 Flow Control across a Single Link

Let us first consider a very simple scenario, which is sending a message across a link from router A to router B. We can reserve space in B at several levels: at the level of a full message, at the level of packets, or at the level of flits. These will lead to different flow control schemes in an NoC, which differ significantly. However, for sending messages along a single link, the schemes are not very different. At the abstract level, they use the same set of mechanisms. Let us describe a basic problem in this space. Without loss of generality, let us assume that we are performing flow control at the level of flits. At this point, let us introduce the term *channel*, which is defined as a unidirectional link between two adjoining nodes in an NoC. Typically, the term *link* refers to a physical connection, and the term *channel* refers to a logical connection. The connotation will be clear from the usage.

Consider a situation in which we need to send 20 flits from A to B. It is possible that B has buffer space for only 10 flits. When the buffers fill up, it is necessary for A to stop sending flits. Since we are not allowed to drop flits, we need to ensure that every single flit is buffered in the downstream router: B in this case. This implies that before we send a flit, we need to ensure that we have space for it in the downstream router B. This forms the basis of flow control at the level of a single link. The mechanisms are similar for flow control at the level of flits, packets, and messages.

Let us outline the basic structure of a solution. A needs to have a precise idea of the buffer space in B. It can underestimate, however it cannot overestimate. This means that if B has 4 free buffers, then A can estimate that B has 2 free buffers. In this case, A will only send 2 flits. It can send 2 more flits, yet it will not send because it will think that B will run out of buffer space. This will hurt performance. The reverse case, which is when A assumes that B has 5 free buffers might lead to situations where we need to drop a flit.

In a conventional network such as the internet, packets are dropped, and there are retransmissions. However, we cannot afford this luxury at the level of an on-chip NoC, where power and performance are very critical issues. Our protocols also need to be rather simple.

Credit based Flow Control

The simplest approach is credit based flow control, where A maintains an estimate of the number of free buffers at B. This estimate is known as the *credit*. When B frees a buffer, it sends a message to A, and A increments its credit. Subsequently, A can send more flits to B. The exact mechanism is as follows.

Every router has a module to process credits. It maintains the number of credits for each outgoing link. Initially the number of credits is equal to the number of buffers in the downstream router, B. Subsequently, when A sends a flit, it decrements its credit count. Now if we look at B, it sends a message to increment the credit count to A whenever it frees a buffer for the $A \rightarrow B$ link. If A runs out of credits, then it stops sending messages. Here, the overhead is the number of additional credit messages: one per freed buffer. Figure 8.10 shows an example of flow control using credits.

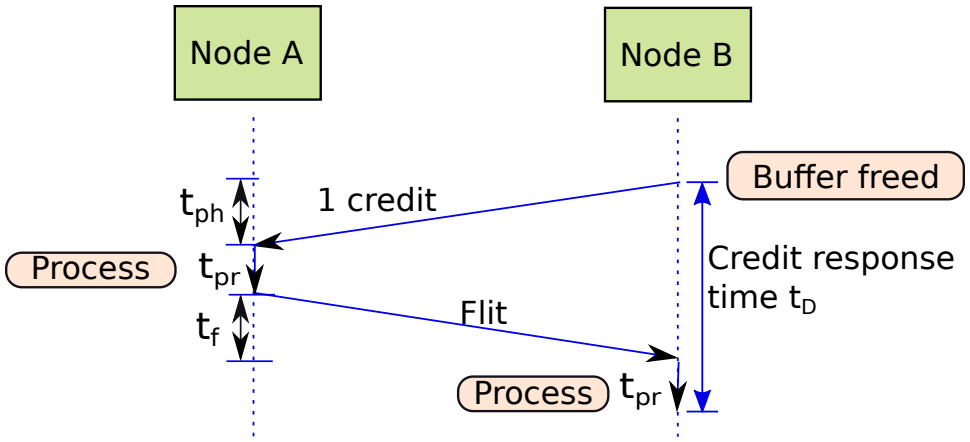


Figure 8.10: Credit based flow control

Let us mathematically analyze this protocol. Let the time it takes a single phit to traverse the link be t_{ph} cycles. This means that to send a credit message (1 phit) it will take t_{ph} cycles. Subsequently, this needs to be processed. Let us assume that the time it takes to process any message (single-phit or multi-flit) be t_{pr} cycles, and the time it takes to send a flit be t_f cycles. Thus, the credit round trip delay, t_D (see Figure 8.10), is given as follows:

$$t_D = (t_{ph} + t_{pr}) + (t_f + t_{pr}) = t_{ph} + t_f + 2t_{pr} \tag{8.2}$$

The unit of t_D is in cycles. Typically, the cores and the router share the same clock. However, if this is not the case, let "cycles" in this case mean router cycles. We would ideally want t_D to be zero, which means that the moment a buffer is free, it is immediately filled up. However, because of signal propagation delays, this will not be the case.

Let us look at t_D from B's point of view. In this period, it could have received a maximum of t_D/t_f flits from A subject to the availability of buffer space. The denominator is t_f because A takes t_f cycles to send a single flit. Assume that when a credit was sent, at least t_D/t_f buffers were free. In this case, A would not have stalled even if it sent a phit every cycle. However, if less than t_D/t_f buffers were free, then there is a possibility of a stall on A's side. It is possible that it would run out of credits and not be able to send phits leading to idle cycles.

We can view this result slightly differently as well. Assume that B is able to send its flits as soon as possible. Whenever it sends a flit, it also sends the corresponding credit message to A. In such a situation, if we want A to never stall, then it means that A should always have enough credits available. After it sends a message, its credit count gets decremented. We never want this credit count to become zero. If its starting count is t_D/t_f, then in this setting its credits will never become zero because of the following reason. If it sends a flit at cycle 0, then it will reach B, t_f cycles later, B will process it and send the flit downstream (as per our assumption), and send a credit back to A. This entire process will take t_D cycles. Given that credits come back to A after t_D cycles, if it has t_D/t_f starting credits then it will never run out of them. This would also mean that B needs to have at least t_D/t_f buffers.

On-Off Flow Control

The main disadvantage of credit based flow control is that we need to send a message when every buffer is freed. This increases the load on the NoC, and it is also not power efficient. To increase the power efficiency we need to send messages sparingly. One trivial approach is to send a message once for every k buffers freed ($k > 1$). This will decrease the number of messages; however, it will make A less responsive.

On these lines, let us propose a protocol called on-off flow control, where we propose a set of rules that A can use to decide whether it can send a flit to B. Let us create two thresholds: N_{ON} (on threshold) and N_{OFF} (off threshold). If the number of free buffers in B becomes equal to N_{OFF}, then it sends a message to A to stop sending. Once A receives this message, it stops sending flits. On the flip side, when the number of free buffers becomes equal to N_{ON}, B sends a message to A to start sending flits. Figure 8.11 shows an example of flit transmission using on-off flow control.

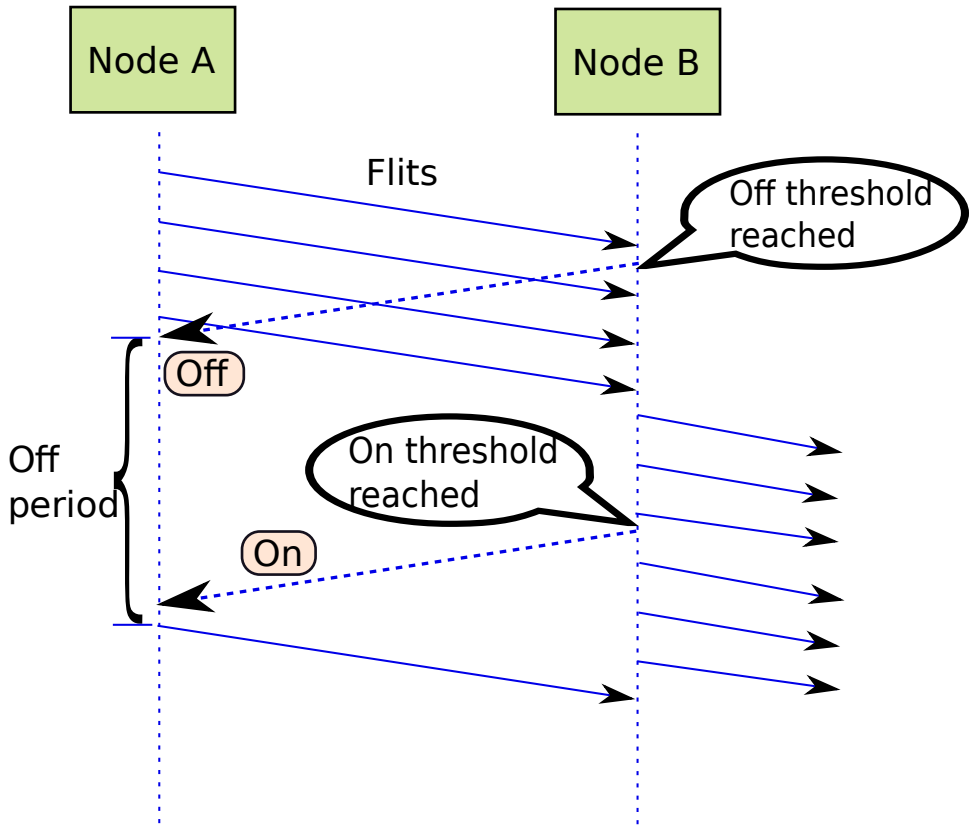


Figure 8.11: On-off flow control

Let us analyze this protocol using the same method and same terminology that we used for credit based flow control. In this case, we need to define two thresholds: N_{OFF} and N_{ON}. Let us first look at N_{OFF}, which is one less than the minimum number of free buffers that B needs to have for A to be allowed to transmit. Note that the key constraint is that we are never allowed to drop a packet.

Assume that B receives a flit from A after local processing at time t. This is when the number of free buffers is decremented by 1. Assume that B finds that the *off* threshold has been reached. It immediately sends an *off* message to A (assume it is a single phit). The fact that the *off* threshold will be reached can be figured out when the flit is being processed at B. Along with decrementing the free buffer count the *off* message can thus be sent in the same cycle. It takes an additional $t_{ph} + t_{pr}$ cycles for the *off* message to reach A and get processed. At that point, A stops sending new flits. However, we need to ensure that we do not run out of buffers at B because of flits in flight – flits sent by A before it was asked to turn itself off. The number of messages in flight can be estimated as follows.

Here are a few points to note. B sends the *off* message at time t. This flit must have been sent by A at time $t - t_f - t_{pr}$.

The time at which the *off* message reaches A and finishes its processing is $t + t_{ph} + t_{pr}$. Let us now

focus on the time interval $[t - t_f - t_{pr} + 1, t + t_{ph} + t_{pr}]$. During this time interval, A can send flits to B. These flits will be received by B after it has sent the *off* message. It needs to have enough buffers to store them. So how many buffers do we need?

If you haven't noticed it yet, the duration of the time interval is $t_f + t_{ph} + 2t_{pr}$. We have seen this expression before. It is the expression for t_D (see Equation 8.2). This means that during this period, when A has not seen the *off* message, it can send t_D/t_f flits. All of them have to be buffered at B because we are not allowed to drop a flit. The N_{OFF} threshold thus needs to be at least t_D/t_f.

$$N_{OFF} \geq \frac{t_D}{t_f} \tag{8.3}$$

Let us now find N_{ON}, which is the threshold at which B can allow A to send messages. The reason that B sometimes does not allow A to send messages is because it may run out of buffers. Once its buffers start clearing up, it can get in more flits. Let us assume that all the flits in the buffers for channel $A \rightarrow B$ need to go from B to C. If the $B \rightarrow C$ channel is blocked, then a back pressure will be exerted on the channel $A \rightarrow B$. Once, B gets an *on* message from C, it can start sending its flits to C. Now, when should it send an *on* message to A?

There is clearly no hard and fast rule. This is an engineering decision and needs to be taken based on simulation results. However, let us look at some general principles. We want to reduce the number of *on* and *off* messages. Hence, we would like to set $N_{ON} > N_{OFF}$ such that if we are sending flits, we keep on sending them for some time. We do not want A to frequently turn off and turn on – this is wasteful in terms of messages and power consumption.

We thus have:

$$N_{ON} > N_{OFF} \tag{8.4}$$

Regarding the value of N_{ON}, let us see what happens if it is equal to $N_{OFF} + 1$. In this case, B sends an *on* message, and when A processes it, it dispatches a flit. If by the time B receives the flit it has freed some more buffers, then there is no problem. Otherwise, if no buffers are freed, then the number of free buffers will become equal to N_{OFF}, and an *off* message has to be sent. All the in-flight flits will take up buffers at B. Subsequently, when the number of free buffers rises beyond N_{ON}, A can start sending flits again.

If we do not want the cycle to repeat too frequently, we need to set a large value of N_{ON} such that if A is transmitting flits, it will continue to do so for a longer duration. This will reduce the number of *on* and *off* messages. The exact number needs to be determined after conducting exhaustive simulations. However, this protocol places a lower bound on the total number of buffers at B. Let us elaborate.

Consider the communication shown in Figure 8.12. Let us assume that N_{ON} is equal to the total number of buffers at B. If B has N buffers, we are assuming that $N = N_{ON}$. Additionally, let the interval between sending an *on* message and receiving a flit because of it, be referred to as *the shadow of the on message*. Since N_{ON} is assumed to be equal to the total number of buffers, which means that we send an *on* message when all the buffers are free, B will be idle in the shadow of the *on* message. It will not have any flits to send to any downstream router. This is a suboptimal situation.

To ensure that B is able to utilize the shadow of the *on* message to transmit flits to other downstream routers, we need to set $N > N_{ON}$. In this case, B will have $N - N_{ON}$ flits with it when it sends the *on* message. Let us see what happens if $N - N_{ON} \geq t_D/t_f$. Note that the shadow of the *on* message is t_D units of time. During this period, t_D/t_f flits can be transmitted by B to downstream routers. Hence, if $N - N_{ON} \geq t_D/t_f$, it means that B can fully utilize the shadow of the *on* messages to send flits to downstream routers.

Now let us collate all our equations.

$$N \geq N_{ON} + \frac{t_D}{t_f} > N_{OFF} + \frac{t_D}{t_f} \geq \frac{t_D}{t_f} + \frac{t_D}{t_f} = 2\frac{t_D}{t_f} \tag{8.5}$$

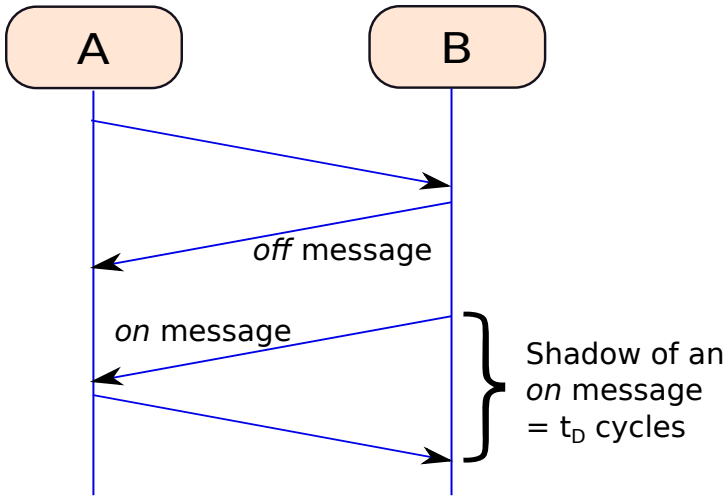


Figure 8.12: The shadow of an *on* message

We thus have

$$N > 2\frac{t_D}{t_f} \tag{8.6}$$

Thus, the total number of buffers, N, needs to be at least $2t_D/t_f + 1$. This will ensure that in the shadow of an *on* message, B is not idle.

8.2.3 Message Based Flow Control

The classic approach to do this is to use a method called circuit switching that has been around since a long time. Recall the early days of telephony[2]. In those days, to place a call from one number to another it was necessary to first reserve the entire route. The person trying to make the call used to first call the operator. Then the operator used to manually compute the route, call other operators and reserve the lines for the entire route. Once the route was reserved, the operator would call the person trying to make the call, and then the call would be connected. This entire procedure took minutes to hours. All of us should thank modern technology that has made life so much better!

We can do something very similar in NoCs. We can reserve the entire path from the source to the destination. In this case, the router at the source can compute the entire path, and reserve buffer space as well as the right to transmit along all the routers on the way. Once the circuit (path from the source to the destination) has been reserved, the entire message can be sent. This method is known as *circuit switching* because we are reserving the full path (the full circuit).

Let us elaborate. After computing the route we send a packet, known as a *probe packet* along the circuit. We always reserve some amount of bandwidth for probe packets. The probe packet sets up the path (or the route). It tells all the routers along the way that they should expect a given message transmission for which they need to have enough free buffers available as well as provide access to send the flits along the desired outgoing links. Once the probe packet reaches the destination, the process of setting up the path is complete. Then the destination sends an acknowledgement to the source indicating that the path has been set up. After the source receives the acknowledgement, it can start sending the message. After it has sent the message, it needs to tear down the path. An easy way of doing this is

[2]Those who were too young those days can always look up the internet to find out how the good old days used to be!

to set an additional bit in the last flit of the message indicating that it is the last flit in the message. Routers on the way can see this bit and dismantle the path that has been set up.

This is by itself a fairly simple mechanism. However, it is associated with large overheads in terms of the path setup delay. Let us derive the time it takes for transmitting the entire message, assuming best case conditions. We shall assume in the subsequent discussion and even in later protocols that one message consists of just one packet. The reason is that for long messages, the total time is simply the ratio of the length of the message and the bandwidth of the link regardless of the protocol (subject to reasonable assumptions). The differences between the protocols arise when we consider the time associated with sending the first packet. Furthermore, given that most message transfers in practical NoCs are single packets that contain a few flits, this is a valid assumption.

Let the length of a packet be L flits, and the bandwidth of a link be B flits per cycle. Furthermore, let the destination be K hops away and let it take 1 cycle to traverse each hop. The total time thus required for the probe packet to reach the destination, and for the acknowledgement to come back is $2K$ cycles. In this case, we are assuming that the probe packet is 1 flit (1 cycle per hop), and the acknowledgement is sent instantaneously. Once the acknowledgement reaches the source, the data transmission starts. To put the last byte on the first link, it takes L/B (size of the packet divided by the bandwidth) cycles. Assume for the sake of simplicity that B divides L and $B \geq 1$.

Subsequently, we require $K - 1$ cycles for the last byte to reach the destination node. We subtract 1 from K because we are already accounting for its transmission time in the expression L/B. The total time thus required is $3K + L/B - 1$ cycles. Let us visualize the process of transmitting the message in Figure 8.13. This diagram is known as a space-time diagram. The columns represent cycles, and the rows represent hops. Such diagrams are used to visualize the actions of a given flow control or message transmission protocol. We shall find them to be very useful while describing different flow control mechanisms.

Before proceeding further, let us differentiate between the terms: *throughput* and *bandwidth*. They are often confused; however, they do not mean the same. Refer to Definition 55.

Definition 55

- *The* bandwidth *is defined as the largest possible rate (measured as bytes per unit time) at which we can send data through a channel or between two points in a network.*
- *On the other hand, the* throughput *is defined as the data rate that we practically observe in a given setting across a channel, or between two nodes in a network. The throughput is always less than or equal to the bandwidth. It is limited by congestion, nuances of the transmission protocol, errors, and any other phenomenon that retards the flow of flits.*

Now, let us consider the average case. We might require much more time. This is because the probe packet might get stuck at any point. In this case, we need to wait. The same can happen to the acknowledgement as well. Moreover, at any point of time while propagating the probe packet, we might run out of buffer space in routers along the way. We need to wait till buffer space is created.

While transmitting the message, we need to pretty much reserve buffer space in routers for at least an entire packet along the entire path. This is unnecessarily conservative. We are essentially reserving the routers for more time than is actually required. The duration of time from the point of reserving buffer space in a router till the point at which the resources in the router are released is not characterized by continuous data transmission. We need to wait for the acknowledgement to reach the source, data transmission to begin, and reach the routers on the way. During this time, it would have been very much possible to send other messages. However, in this protocol, we refrained from doing so. This reduced the net throughput of the system.

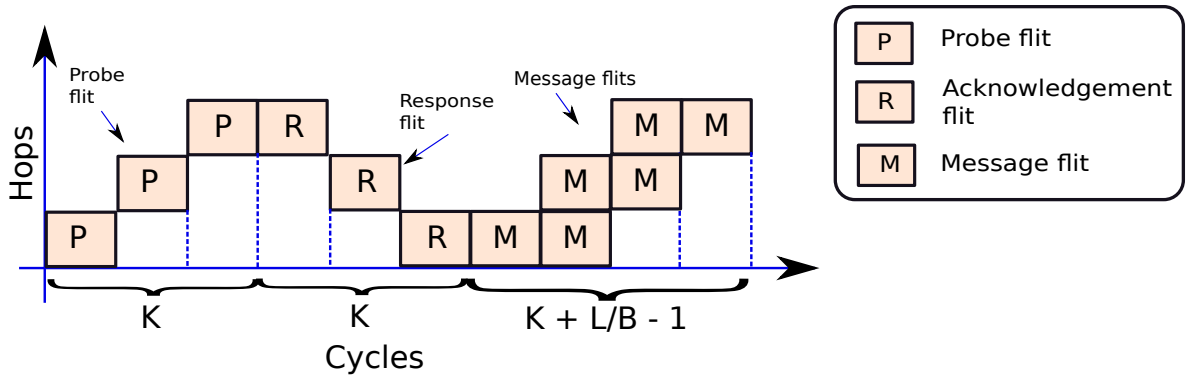


Figure 8.13: Space-time diagram for circuit switching. $K = 3$ cycles, $L = 2$ flits, $B = 1$ flit/cycle. Assume each packet contains a single flit. The y-axis is the hop count (measured from the source).

In addition, freezing a route in advance is not always a good idea. It is possible that there might be many more messages that might want to use parts of the route. They will not be able to traverse the route till it is released.

Now that we have discussed the negative aspects of the protocol, let us briefly enumerate its advantages. Any kind of circuit switching protocol is always very simple, and works well in a scenario with less contention. They can also be implemented and verified easily. To understand the performance advantages, let us consider the latency once again. We had computed it to be $3K + L/B - 1$ cycles. If L/B is significantly greater than K, we can assume that the net latency is equal to L/B cycles. This means that if we need to send a very long message, the additional timing overhead of the probe packet, and the acknowledgement appear to be negligible. Since we have reserved the full path, the entire message can be sent without any subsequent delays. One more advantage of this system is that we can use routers with very few buffers that are just needed to hold in-transit regular flits and probe/acknowledgement flits. In fact, such schemes can also be used with bufferless router designs. Once a path is set up, we do not need any long term packet storage structures in the routers. The routers just need to read data from the input ports, and write the data out at the relevant output ports. They need not buffer the data beyond the flits that are either not fully read or fully written. This will make our routers smaller, and more power efficient.

To conclude, circuit switching is simple, straight forward, yet is not efficient. It is suited for scenarios that have less contention in the network and very long message lengths.

8.2.4 Packet Based Flow Control: Store and Forward (SAF)

Instead of doing flow control at the level of messages, let us do it at the level of packets. As we shall see, this will give us far more flexibility in scheduling the reservation of resources, and will increase the net message transfer bandwidth among nodes in a chip.

In this case, we do not reserve a full path for a message. Instead, we transmit individual packets separately. There are different mechanisms for choosing the path (route) that each packet will take. This can be done either statically or dynamically. We shall study routing mechanisms in detail in Section 8.3. However, for all of these mechanisms, we need to ensure that we have enough resources in terms of buffer space along the way. We do not ever want to lose a flit because we were not able to transmit it to the neighboring node. We assume that each router has several flit buffers. To store an entire packet, we need multiple flit buffers – one per each constituent flit.

This is shown in Figure 8.14, where we show two nodes A and B, and a channel from A to B. Every channel has an associated set of flit buffers at its destination. The $A \rightarrow B$ channel has 4 dedicated

flit buffers in node B. Any flit that is transmitted on the channel is first buffered in the channel's flit buffers in B, and then B forwards them to their destination. We can see in the figure that out of the four flit buffers, two are empty, and two are occupied. A channel can be thought of as the combination of a link and a set of flit buffers at the destination node. A point to note is that each set of flit buffers is actually a first-in-first-out queue (FIFO queue). We shall use the term "flit buffers" and "flit queue" interchangeably.

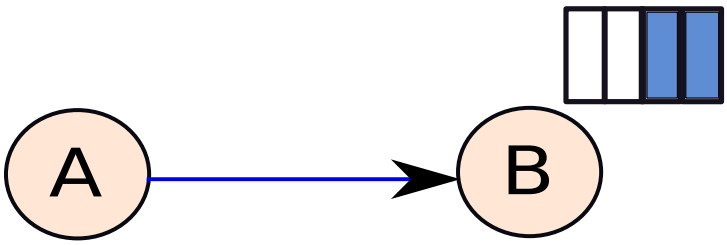


Figure 8.14: A channel and flit buffers

The simplest flow control algorithm for packets is known as the *store and forward* (or SAF) protocol. In this case, we forward the entire packet from one node to the next node on the path. In the next node, we wait for the entire packet to arrive before we transmit the first flit of the packet to the subsequent node on the path. The corresponding space-time diagram is shown in Figure 8.15.

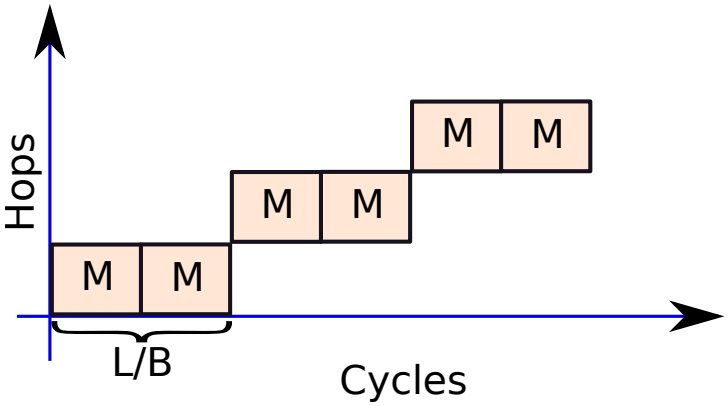


Figure 8.15: Space-time diagram for the store-and-forward approach. $K = 3$ cycles or hops, $L = 2$ flits, $B = 1$ flit/cycle. We are assuming that a packet has 2 flits. The space-time diagram is shown for a single packet.

From Figure 8.15, let us compute the time that is required for a packet transmission. Let us use the same assumptions and the same parameters as we had used for the computation for circuit switched networks. Recall that the three parameters were L (length of the packet) B (bandwidth of a link), and the number of links or hops (K) between the source and the destination. We assume that it takes one cycle to traverse a single hop. As we can observe in Figure 8.15, the total time that is required is $L/B \times K$ cycles. This is because it will take L/B cycles to traverse each hop, and there are K such hops.

Let us compare this formula with what we had derived for a circuit switched network. In that case, the total time taken for a single-packet message transfer was $3K + L/B - 1$ cycles. We can quickly observe that the total time required is much more for our current approach, SAF flow control: the time required is $L/B \times K$ cycles. This is because we wait at every node for the entire packet to arrive, and

only then we transmit the packet to the neighboring node. This is clearly inefficient and negates the benefits accrued out of a packet switching scheme.

8.2.5 Packet Based Flow Control: Virtual Cut Through (VCT)

It is a much better idea to pipeline the packets such that the head flit does not have to wait for the tail flit to arrive at the router. If the head flit can make progress and move to the next node, it should be allowed to do so. This will increase performance because in this case unless there is any congestion, the flits in the packet will never wait. This method of flow control is known as the *virtual cut through* (VCT) approach.

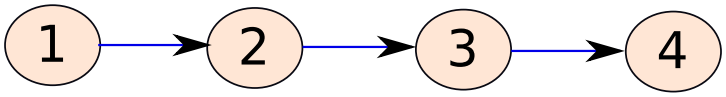


Figure 8.16: Simple network with 4 nodes

Let us show an example of such a transmission in Figure 8.16. Let's say that we need to send a message from node 1 to node 4. At a given instant of time, the flits of the message can be in the nodes 2, 3, and 4. This would not have been possible with the SAF method. However, in this case, it is possible to get more performance because we are transmitting the flits of a packet as soon as possible. Note that there is a caveat: the flow control is still packet based. This means that whenever node A sends the head flit to node B, we need to ensure that in node B, we have enough buffer space to store the entire packet. This is because if the head flit gets blocked at B, the rest of the flits in the packet will continue to arrive, and we need adequate buffer space to store them.

The advantage of this scheme is that we retain the simplicity associated with packet based flow control, and in addition, we do not unnecessarily need to block the head flit if a few of the body flits have still not arrived. Let us draw a sample space-time diagram for message transmission using this kind of flow control. It is shown in Figure 8.17.

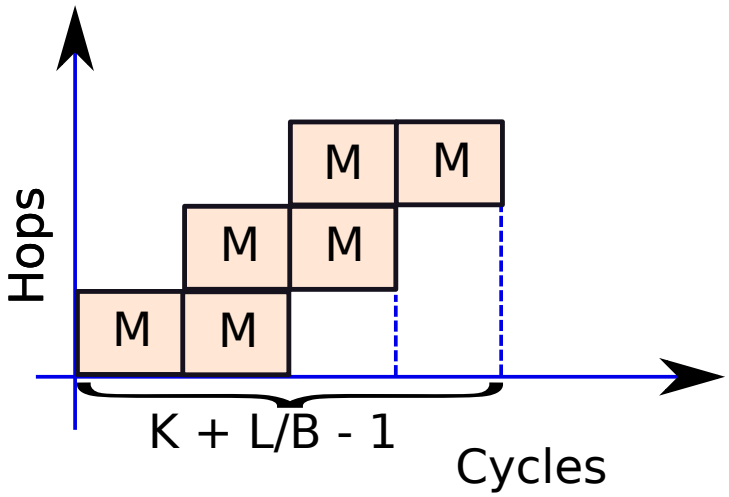


Figure 8.17: Space-time diagram for the VCT approach. $K = 3$ hops or cycles, $L = 2$ flits, $B = 1$ flit/cycle. A packet has 2 flits.

The total time required to send a packet that is L flits long can be derived from Figure 8.17. It is

$L/B + K - 1$ cycles. This is because it will take L/B cycles for the source to transmit the last byte. Subsequently, the last flit needs to travel through $K - 1$ hops to reach the destination. Hence, the total time is equal to $L/B + K - 1$ cycles.

Let us now comment on the relative advantages and disadvantages of this scheme. The advantage is that it is the fastest scheme that we have seen up till now (refer to Table 8.1). We observe that VCT is clearly the fastest. For large messages, the time it takes to transmit a message with VCT is similar to the time it takes with circuit switching because $L/B \gg K$. However, circuit switching has other problems, notably the difficulty in reserving resources along a path, and also the fact that it has the potential to keep large parts of the network idle.

Scheme	Time (cycles)
Circuit switching	$3K + L/B - 1$
Store and forward (SAF)	$K \times L/B$
Virtual cut through (VCT)	$K + L/B - 1$

Table 8.1: Single-packet message transmission times using different schemes (assuming no congestion)

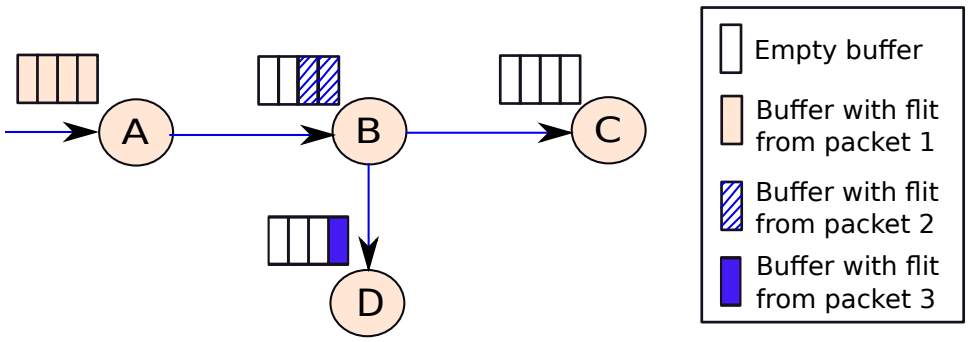


Figure 8.18: Stalls in the VCT scheme

The disadvantages of flow control at the level of packets are several. The biggest disadvantage is that we need to reserve space at the granularity of packets. For example, consider the case where a packet consists of 4 flits. If we have buffer space in the next router for only 3 flits, then we will not be able to transmit any flit in the packet. This means that we will have to wait till one more flit buffer in the next router on the path is free. This is a waste of time because we could have transmitted flits to the next router, and it is possible that in the time being another flit buffer in the next router would have been freed.

Let us illustrate this situation with an example shown in Figure 8.18. Here, we want to transmit a 4-flit packet (packet 1) from node A to C. However, in node B, we do not have enough buffer space available for the entire packet. This is because a packet (packet 2) in B needs to be sent to D. It occupies 2 buffers, and 2 buffers are free. Since packet 1 contains 4 flits, and we reserve space for entire packets, the transmission from A to B cannot proceed. We thus have to wait for 2 cycles for both the flits in packet 2 to leave B and get buffered in D. Then only, we can transmit flits from packet 1.

8.2.6 Flit based Flow Control: Wormhole Flow Control

The only way to solve this issue is to go for flit based flow control, where we decide whether to transmit or not at the level of flits. In other words, it is not necessary for subsequent routers on the path to reserve buffer space at the granularity of packets. This definitely will improve performance at the cost of added complexity.

By removing the restriction of reserving space at the level of packets, we can get a higher throughput, and reduce the latency of message transmission. Let us reconsider the example in Figure 8.18, and see what happens if we reserve space at the granularity of flits. This is shown in Figure 8.19. In this figure, we avoid the problems that we had with VCT based flow control. We always transmit a flit to the next router on the path, if it is possible to do so. Recall that with VCT based flow control, we were not able to transmit any flits of packet 1 till all of B's 4 flit buffers were empty. However, in this case, we need not be constrained by this. Figure 8.19 shows the situation, 1 cycle later. We sent one flit of packet 2 from B to D. At the same time, we sent one flit of packet 1 from A to B. As a result, in the flit buffers of B, we have flits from both packets 1 and 2. Henceforth, for the next 3 cycles we can keep sending a flit from A to B, and thus there are no stalls as far as the traffic from A to B is concerned. This assumes that we send the remaining flit of packet 2 from B to D within this period.

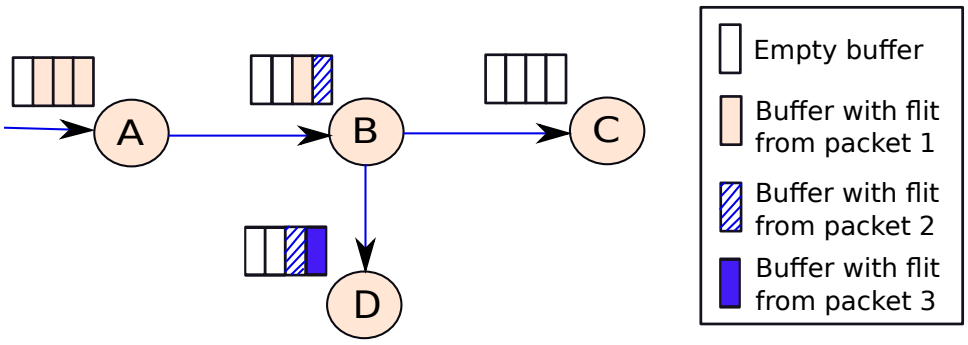


Figure 8.19: Wormhole flow control

The best case time for transmitting a single packet still remains the same: $L/B + K - 1$ cycles. However, in this case, we can deal with congestion much better. If there is congestion, and we do not have enough buffer space to store an entire packet, we can still forward some flits of the packet and wait till more space is created. This mechanism is known as *wormhole flow control*. A wormhole is a hole that a worm or an insect makes by burrowing through wood or mud. The way that flits in a packet flow through the network is similar. We can visualize this as a worm moving through its burrow.

Let us quickly go through the advantages of wormhole flow control.

1. Routers can be smaller. They do not need to have space to buffer multiple, large packets. They can have less storage space, and fewer flit buffers.

2. Furthermore, as compared to the SAF and VCT techniques, it propagates flits sooner because of the reduced waiting times.

Wormhole routing is far from perfect. The problems can be illustrated in Figure 8.20. Consider the following scenario. Packet 1 needs to traverse nodes A, B, and C. Simultaneously, packet 2 needs to traverse nodes A, B, and D. Assume that there is congestion at node D, and this stalls packet 2. In wormhole switching we do not allow later flits to overtake earlier flits because they are all in the same queue. Hence, as we see in Figure 8.20, flits from packet 2 block the queue at node B. This blocks the flits of packet 1, even though they can make progress and can be sent to C, whose buffers are empty. For the flit queue at node B, we have head-of-line blocking, also known as HOL blocking, which means that the flit at the head of the queue is blocked, and as a result the rest of the flits in the queue are also blocked, even though some flits in the queue could make progress.

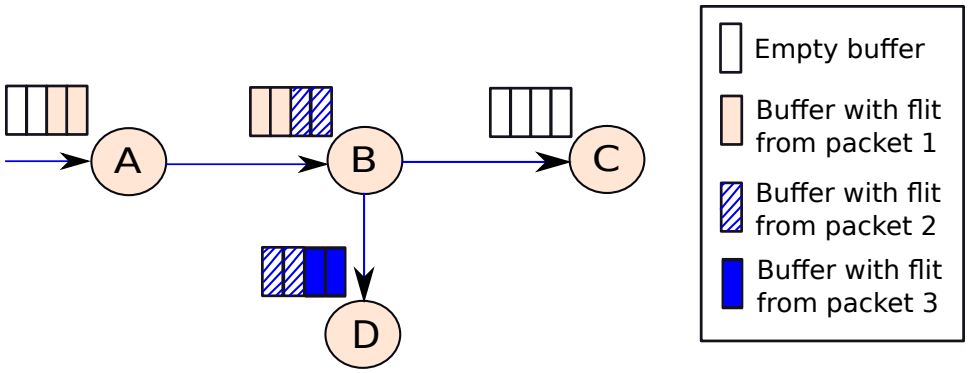


Figure 8.20: Problems with wormhole flow control

Definition 56
Head-of-line (HOL) blocking is a phenomenon in on-chip networks where a sequence of flits is stuck because the flit at the head of the queue cannot move to another node, even though other flits in the body of the queue can traverse their routes.

8.2.7 Flit based Flow Control: Virtual Channel Based

The main problem with wormhole flow control is HOL blocking. Reconsider the fact that the flits of packet 1 could have made progress in the example shown in Figure 8.20, but they got blocked by flits of packet 2. Let us try to fix this problem by proposing some simple solutions.

What if we had two channels from node A to node B? We could allocate one channel to packet 1 and the other to packet 2. This would solve our problem. Since each channel has its dedicated set of flit buffers, there would be no HOL blocking. Regardless of packet 2, packet 1 could make progress and reach its destination node C. We can do slightly better in terms of reducing the overheads. We can still have two sets of flit buffers, but we can use a single physical link; we can multiplex the transmission of packets 1 and 2 through the link. From a conceptual standpoint, the entire system works as if there are two channels between nodes A and B.

To multiplex a physical channel across different packet transmissions, it is necessary to keep track of the flits that belong to each packet. For example, if we are multiplexing between packets 1 and 2, then we need to have a method of marking the flits that belong to each of these packets. It should never be the case that we are not exactly aware of which packet a given flit belongs to. This means that if N packets are waiting to be transmitted on a channel, we assign a $\log(N)$-bit id to all the flits in each packet. This will help us correctly group the flits into packets. However, just numbering the flits and packets isn't all that is there to physical channel multiplexing. Choosing the flits that need to be transmitted next is equally difficult. Previously, we always sent flits belonging to the same packet, hence, in terms of choice there was nothing to choose. However, now since we have multiple packet flows, we need to make choices. This logic introduces some complexity and increases the overheads.

Virtual Channels

Figure 8.21 summarizes our discussion. We started with the picture on the top, where we proposed multiple physical channels: one per packet. The overheads were prohibitive; we then proposed a single

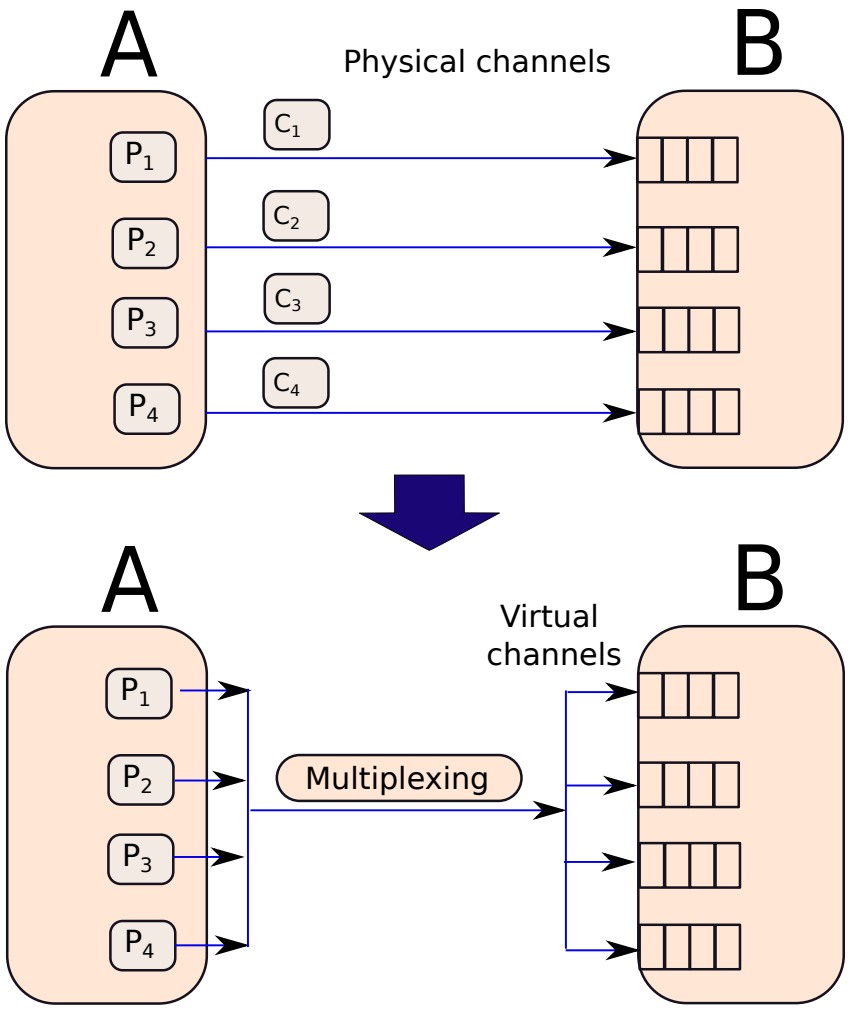


Figure 8.21: Replacing a single physical channel with multiple virtual channels

channel that is multiplexed between packets. Conceptually, each packet is assigned to a *virtual channel* that has its own set of flit buffers (or a flit queue). *Virtual channels*, abbreviated as VCs, form the core of the routers of modern NoCs.

A virtual channel is like virtual memory in some sense. It provides an abstraction or illusion of a physical channel. As we can see in Figure 8.21, we have multiple flit queues at the input ports of every router. This means that if there is a link from A to B, we have multiple flit queues at B's input ports for the channel $A \rightarrow B$. Each flit queue represents a different virtual channel (VC). When A transmits a flit to B, it indicates the id of the virtual channel that it is using. Assume that we have 4 flit queues at B. Then, when a flit is sent from A to B, we need to mark the id of the VC on the flit. If A says that a given flit belongs to VC 2, then B reads that information and queues the flit in the second flit queue for the $A \rightarrow B$ channel. Similarly, if A were to stamp a given flit with the VC id 3, then B needs to buffer that flit in the queue corresponding to the third VC of the $A \rightarrow B$ physical channel. Let us look at the other side. Every cycle, A needs to pick a flit from the set of packets that need to be sent to B; it might have multiple choices. It might have four different packets in its buffers ready to be sent to B. In all the previous schemes that we have seen, the only option that A had was to pick a packet,

send all of its flits, and then switch to another packet.

However, in this case, it can choose flits from different packets, and send them across different virtual channels to B. It can use different heuristics to decide which flit needs to be sent along the multiplexed physical channel. Again, the best case latency to route a full packet is $L/B + K - 1$ cycles, which is similar to the wormhole and VCT techniques.

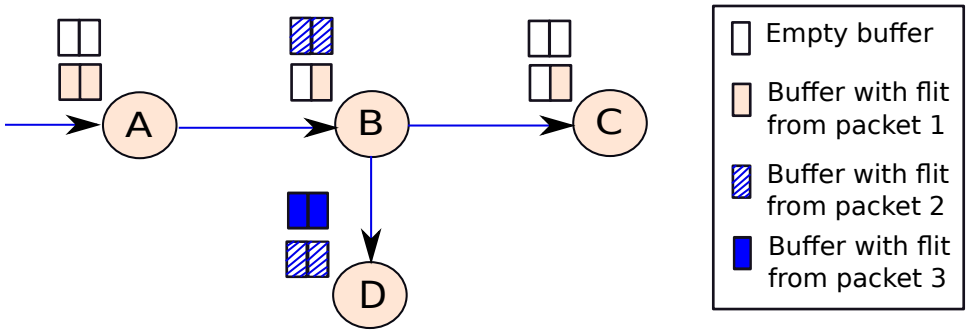


Figure 8.22: Virtual channel based flow control

However, this method does avoid the HOL blocking problems of wormhole flow control. Let us consider the same example as we had shown in Figure 8.20. Let us show its operation with VCs (virtual channels). We assume that we have two VCs per physical channel. The operation of the protocol is shown in Figure 8.22. We observe that in this case, node B is not a bottleneck anymore. It processes two separate flows simultaneously: $A \rightarrow B \rightarrow C$, and $A \rightarrow B \rightarrow D$. In this case, packet 2 does not block packet 1 because packet 1 is on a different VC. Furthermore, node A multiplexes the $A \rightarrow B$ channel between both the packets, and thus flits for both the packets can be sent in the same time window. As a result, flits in packet 1 make progress, even though the flits in packet 2 are blocked at nodes B and D.

This is precisely the greatness of the VC based approach, which is that we do not allow packets taking one route to block packets taking another route. We allot them to separate VCs, and this allows us to ensure that we can move as many flits in the network as possible. This reduces the effects of congestion, decreases the end-to-end latency, and improves the overall throughput. There are a few more advantages of virtual channels such as deadlock avoidance. We shall take a look at such issues in Section 8.3.

8.3 Routing

Let us quickly recapitulate some basic concepts.

A network can be visualized as a graph with a set of nodes and links. A *node* is defined as an entity that can transmit and receive messages over the NoC. A node is connected to other nodes as defined by the network topology. For example, in a mesh, a node in the center of the network is connected to four other nodes via links, where a link is a physical channel via which we send a message. The links are a set of parallel copper wires in conventional NoCs. In a typical scenario, given two nodes in the network, we need to send a message between them. We can have many paths between these nodes. The process of choosing a path between two nodes is known as *routing*. The route (path) between two nodes can either be decided in advance (static), or it can be computed as the message travels from the source to the destination (dynamic routing).

Definition 57
The process of choosing a path between two nodes in an on-chip network is known as routing. *Routing*

can primarily be of two types: static and dynamic. When the path between two nodes is known in advance, we refer to this method as static routing. In contrast, when the path is not fixed, and is decided as the message is traveling from the source to the destination node, we refer to this method is known as dynamic routing.

Given a network topology, a source, and a destination, let us understand what are the properties of a good route. Once we know what a good route looks like, we can design a routing algorithm to compute it.

Let us draw an analogy with real life. If we are going from point A to point B in a city, then what are our priorities? We would always like to reach the destination as quickly as possible – in the shortest time. This typically means that we would like to traverse the minimum number of links while going from the source to the destination. The implicit assumption here is that the time it takes to traverse a link is always the same, and we do not spend time doing anything else. This method is known as shortest-path routing, where we would always like to traverse the shortest path with the minimum number of links.

If links have variable delays, then also we can use shortest-path based routing. In this case, we choose the route that requires the least amount of time to traverse. Shortest path based routing is typically a good choice when we are performing static routing. We can use the Djikstra's shortest path algorithm [Cormen et al., 2009] for computing the shortest path between a pair of nodes.

Even though such algorithms seem very simple, straightforward, and optimal; however, in practice they are not very effective. Again let us come back to the analogy of a crowded city. If we always take the shortest route between two points, it might not always take the shortest time. This is because we might enter a crowded intersection, where we might get stranded for a long time. Sometimes it is necessary to take some diversions such that we will reach our destination sooner even though the route might be longer. We thus learn our first lesson: whenever there is congestion in a network, the shortest route in terms of the distance or expected traversal time need not be the shortest in terms of the actual traversal time. In fact, if there is congestion, it is possible that a message might wait at one of the intermediate nodes for a long time until the congestion reduces.

Hence, to reduce the time that it takes to go from point A to point B, the shortest route is not always the best route. We need to take appropriate diversions and go via alternative paths, the same way an experienced cab driver navigates his way through a busy city.

Let us now look at some of the problems that can happen in automated routing algorithms. It should not be the case that we keep going round and round in circles. In this case, we are not waiting at one particular point; however, we are also not making any real progress and reaching the destination. Such a scenario is known as a *livelock*. We need to ensure that livelocks never happen in practice. There is nothing wrong if a message goes round and round in cycles a few times, nevertheless, it should ultimately reach its destination. If we have the possibility of livelocks, then a message can be stuck in the network for an indefinite period. In addition, it is also possible that a given message is not able to make progress because we continuously give preference to other messages. As a result, we are either in a position where we cannot inject it into the network, or the message is stuck at some intermediate node because it is giving way to other messages. Such a scenario is known as *starvation*.

Definition 58 *A* livelock *is defined as a general condition where the state of the system changes continuously; however, there is no long term progress. The classic example of a livelock is two people approaching each other in a corridor. Both of them try to cross each other; however, the moment one person moves to his left, the person on the other end moves to his right (and vice versa). Hence, they continue to face each other, and even though they are moving, they fail to make progress and reach their destinations!*

Starvation *is defined as a situation where a message is not able to make progress because routers chose to transmit other messages in its place for an indefinite period. Either it fails to get injected into the network or it gets stuck at an intermediate node because other messages are transmitted in its place.*

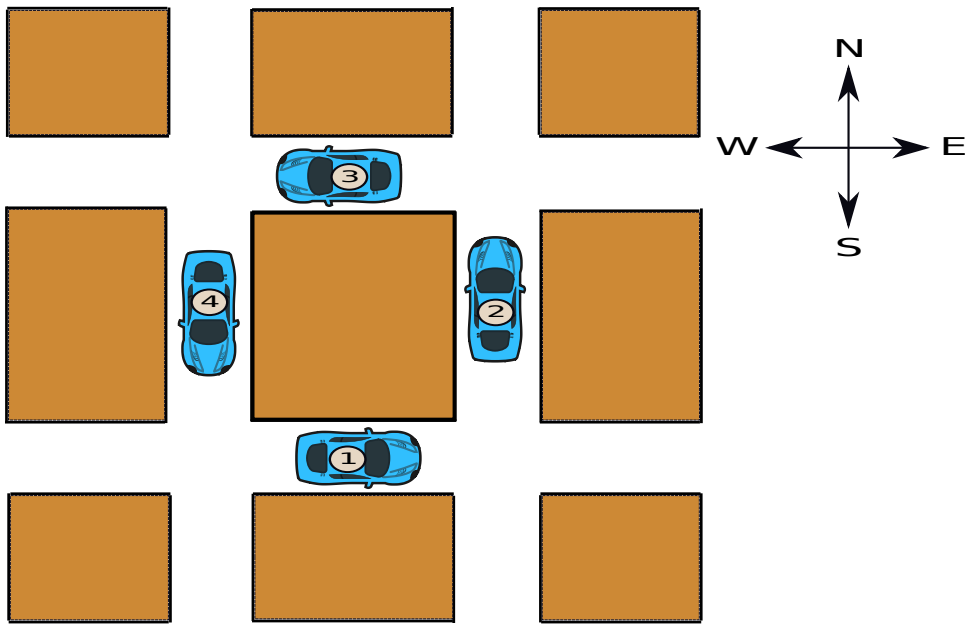


Figure 8.23: Gridlock in a roundabout

What else can happen in a city? Let us look at Figure 8.23. It shows an image of typical city traffic where there is a gridlock. If we look closely we can conclude that no car is able to move. Car 1 is trying to go north, it is blocked by car 2 that is trying to go west, which in turn is blocked by car 3 that is trying to go south, which is blocked by car 4 that is trying to go east, and finally this car is blocked by car 1. There is a circular wait where no car is able to make progress. Such a situation in computer science is known as a *deadlock*. It can lead to an infinite wait, and the entire system can stall. In such a situation we have a circular wait, where no car driver is willing to give up. If we have a system where we can fly a helicopter and pick a waiting car and drop it at its destination, then we will never have a deadlock! However, in the normal case, a deadlock is possible, unless one of the cars is willing to back out and try a different route.

Definition 59
A deadlock is defined as a situation where multiple nodes try to send messages, yet none of them are successful because there is a circular wait. Assume that the nodes are numbered $V_1 \dots V_n$*, and node* V_i *waits on node* V_{i+1} *for it to free some space such that it can send a message to it* $(V_i \rightarrow V_{i+1})$*. We also try to send a message from node* V_n *to* V_1*. We thus have a circular wait of the form:* $V_1 \rightarrow V_2 \dots \rightarrow V_n \rightarrow V_1$*. It is possible to resolve such a deadlock only if we can remove some messages from the nodes and send them along different paths.*

We can have a very similar situation while sending messages as shown in Figure 8.24. Note that each node has a finite message storage capacity, and it cannot accept newer messages. Assume that a node can only store one message at a time (for the ease of explanation), and a message is not sent until there is free space available in the destination node. Further, assume that we have four adjoining nodes that have one message each, and they want to move in the directions as shown in the figure. It is clear that there is a deadlock situation because the message at node A cannot move to node B because node B does not have the space to store it. For the same reason, the message at node B cannot move to node C. We have a circular wait, and since no message can be dropped, we have a deadlock. Such kind of deadlocks have to be avoided at all costs.

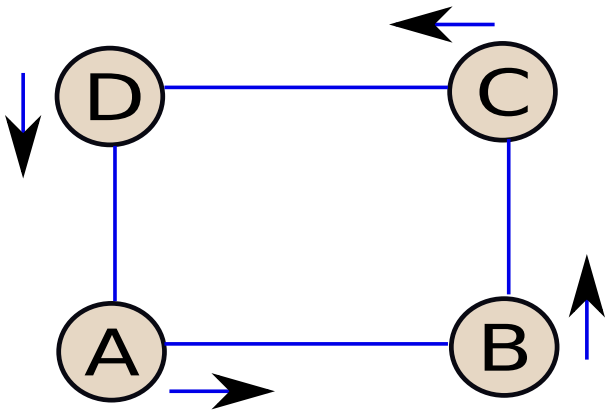


Figure 8.24: A deadlock in a 4-node system

Way Point 10

- *Our main aim while trying to route a message between two points of the NoC is that we want the message to reach in the shortest possible time. Often we would like to maximize the throughput of the network, which means that we should be able to transport the maximum number of bytes per unit time between all pairs of transmitting sources and destinations.*
- *If there is no congestion in the network, then the shortest path between two nodes is also the path that takes the least amount of time. However, if we have congestion at the nodes, then the problem becomes complicated. We need to take the network congestion into account, and this creates the need to often take longer paths in terms of the number of links traversed.*
- *In all cases, we would like to avoid starvation, livelocks and deadlocks. If routers do not transmit a flit yet continue to transmit other flits in its place for an indefinite period, then this condition is known as* starvation. *In contrast, a* livelock *is a situation where flits move through the network; however, they move around in circles and do not reach the destination in a finite amount of time. Whereas, a deadlock refers to a situation where in a set of nodes, we have flits that are stuck and cannot make any progress. This typically happens when we do not have enough storage (buffer) space available at the nodes, and we have a circular wait.*

8.3.1 Handling Starvation and Livelocks

Let us look at avoiding and recovering from starvation and livelocks. This is far simpler than dealing with deadlocks.

For avoiding starvation, we need to have fair routers that do not hold a set of flits for an indefinite duration while transmitting other flits. They need to have a priority scheme, where they transmit flits fairly. We shall discuss such schemes when we discuss the design of routers in Section 8.4.

Handling livelocks is also easy. We can take inspiration from traditional computer networks. We add a *hop count* field with the head flit of each packet. As the head flit traverses each router, we increment the hop count. We can define a threshold for the hop count, which can be a function of the shortest distance between the source and the destination. For example, if the length of the shortest path between the source and the destination is 6 hops, we can set the threshold to be 12 hops. After we have crossed the threshold, we can instruct the routers along the way to only route the message along the shortest path to the destination. They are not allowed to *send* the packet along other paths. Another variation of this approach is to give more priority to packets with a higher hop count in routers. This will ensure that the probability that they will take the shortest path to the destination increases over time.

8.3.2 Deadlocks in Routing Algorithms

Before we look at routing algorithms in depth, it is important to understand the notion of deadlocks in more detail. This is because avoiding deadlocks will be one of our primary objectives while designing routing algorithms. We shall see that taking care of livelocks and starvation is relatively easier. However, designing protocols that are provably deadlock-free is difficult.

A *deadlock* is defined as a situation where we have a set of flits that are not able to make progress because of a circular wait. This means that we are trying to send a flit on a given channel, that channel is not free, the flit blocking that channel is not able to make forward progress, and so on.

Dependence Graphs

Let us explain the notion of deadlocks by developing a set of theoretical tools. Consider a system of four nodes as shown in Figure 8.25. Assume we have a single virtual channel (VC) per physical channel. Here, packet P_1 at node A is trying to use channels 1 and 2, P_2 at node B is trying to use channels 2 and 3, P_3 at C is trying to use channels 3 and 4, and P_4 at D is trying to use channels 4 and 1. Assume that at a given point of time P_1 holds channel 1, P_2 holds channel 2, P_3 holds channel 3, and P_4 holds channel 4. If a packet occupies at least one flit buffer of a channel, it is said to *hold* it. There is a circular wait here if we assume that we do not have enough buffer space for even transmitting a single flit. As a result, none of the packets will be able to make forward progress. This situation represents a deadlock, and needs to be avoided in all cases.

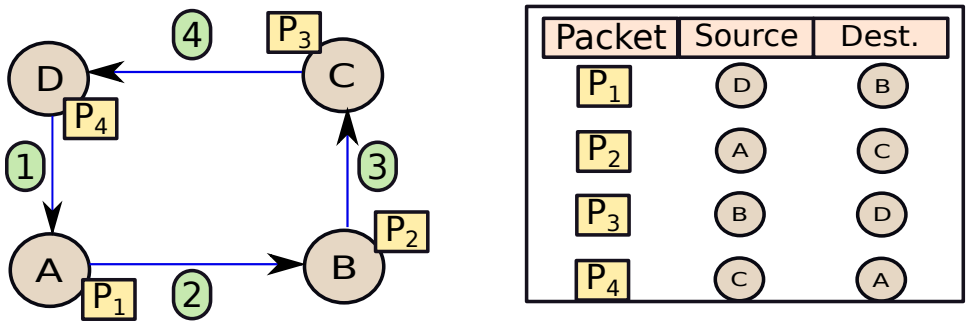


Figure 8.25: Deadlock in an NoC (single VC per channel)

To model this situation and even more complex situations, we need to introduce a new theoretical tool called the *Resource Dependence Graph (RDG)* shown in Figure 8.26 (a graph is defined in Section 2.3.2). Let us have two kinds of nodes: agents and resources. In this case an agent is a packet and the resource is a channel. We add an edge (hold edge) from resource R to agent A, if A holds R. Similarly, we add an edge from A to R (wait edge) if A is waiting for R to become free. Note that the *hold* and *wait* edges are in opposite directions. A resource dependence graph (RDG) for the scenario shown in Figure 8.25 is shown in Figure 8.26. Consider the case of packet P_1. It holds channel 1, hence there is an arrow from channel 1 to P_1. P_1 waits for channel 2 to become free. Hence, there is an arrow from P_1 to channel 2. The rest of the edges are added similarly.

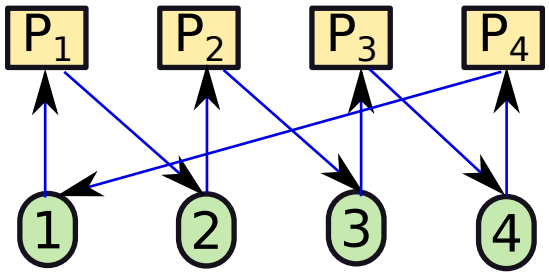


Figure 8.26: Resource dependence graph for Figure 8.25

This graph can be further simplified into a *channel dependence graph* or CDG. In this case, we remove the agents (packets), and just have the channels (physical or virtual) as the nodes. There is an edge from channel C to channel C' if an agent that holds channel C waits for channel C' to become free. The equivalent CDG for Figure 8.25 (and Figure 8.26) is shown in Figure 8.27. Note that we are assuming that channels are *unidirectional*, and the buffer space associated with the channel is at the receiver node. This means that if there is a connection between nodes A and B then there are two channels for message transfer: $A \rightarrow B$ and $B \rightarrow A$. The channel $A \rightarrow B$ has buffer space at node B.

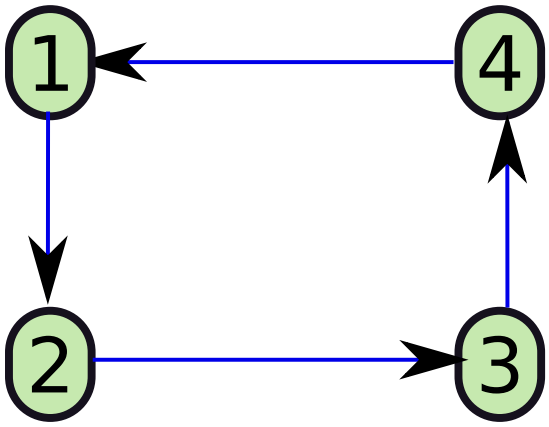


Figure 8.27: Channel dependence graph

Let us quickly observe that there is a cycle in the graph. This hold-and-wait cycle indicates the same situation as Figure 8.24, where we have a deadlock. We claim that whenever we have a cycle in the CDG, we have a deadlock, and vice versa. This is easy to prove. Consider the fact that every edge in the graph indicates that the agent holding a source node is waiting for the agent holding the destination node. This means that in a cycle all the agents (packets in this case) form a cyclic dependence (circular

wait), which represents a deadlock. The converse is also true; if there is a deadlock then the circular wait between the channels will be visible in the CDG as a cycle.

Having multiple virtual channels does not fix the issue. Let us consider the same example with two virtual channels per physical channel. In this case, let us number the virtual channels corresponding to each physical channel with the subscripts 0 and 1 respectively. Even if a packet is allowed a choice between the virtual channels, then also we can have a deadlock (see Figure 8.28). We simply need to consider two sets of packets flowing along the same route, and with the same set of dependences as we have seen before. In this case, both the sets of packets will not be able to progress due to the lack of virtual channels. We shall thus have a deadlock regardless of the number of virtual channels.

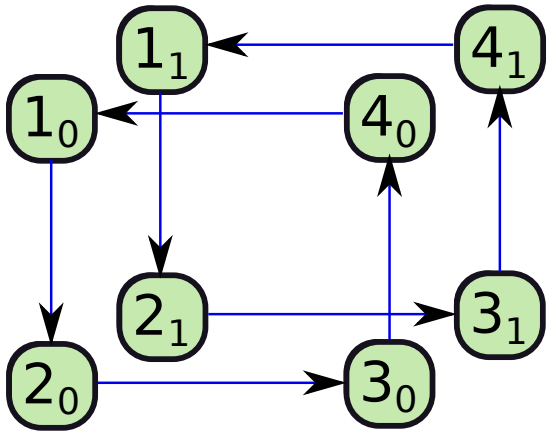


Figure 8.28: Deadlocks in an NoC with 2 virtual channels

Turn Graph

The channel dependence graph (CDG) is the classical tool that is used to detect deadlocks. We just need to check for cycles. However, it is not very useful beyond this. This is because the information about the orientation (or direction) of the channel is lost. Let us thus create another theoretical tool – a turn graph abbreviated as TG. We create a TG from the original topology of the NoC and a subset of the CDG.

In the CDG, let us consider a path $\mathcal{C}$ that may contain cycles as well. $\mathcal{C}$ is an ordered set of channels $(C_1, \ldots C_n)$, where channel C_i is dependent on channel C_{i+1}: the end node of channel C_i has to be the starting node of channel C_{i+1}. The TG allows us to visualize the cycles in $\mathcal{C}$ better in terms of the orientation of the channels that make the cycle.

To define the TG, let us first define $\mathcal{G}$, which is a graph that represents the topology of the network. This graph has nodes (representing the routers) and directed edges (channels). The orientation of the edges is the same as that in the actual NoC. For example, the graph for a 2D mesh is a rectangular grid of nodes connected via edges. If an edge goes from north to south in the actual NoC, then its orientation in $\mathcal{G}$ is the same.

The TG for $\mathcal{C}$ is a sub-graph of $\mathcal{G}$ that contains all the channels in $\mathcal{C}$ and no other channel. Additionally, we insert a new node called the *channel node* in the middle of each edge or channel. This means that if there is an edge from node A to node B in the original graph $\mathcal{G}$, then in the TG we have an edge from A to the channel node C_{AB} and then an edge from C_{AB} to B.

Before looking at an example, let us appreciate the key insights that were used to construct the TG. The first insight is that the turn graph captures the orientation of a channel in the actual NoC. The second insight is that for channels C_i and C_{i+1} in $\mathcal{C}$, where C_i depends on channel C_{i+1}, there is a path

from the channel nodes corresponding to C_i and C_{i+1} in the TG. This can be generalized as follows. For $C_i \in \mathcal{C}$ and $C_j \in \mathcal{C}$, where $i < j$, there is a path from the channel nodes corresponding to C_i and C_j in the corresponding turn graph.

Let us consider an example in Figure 8.29. Figure 8.29(a) shows the network topology ($\mathcal{G}$) of a 2D mesh. Figure 8.29(b) shows a CDG with circular (cyclic) waiting for 4 channels: 1, 2, 3, and 4. Then we create a turn graph for these channels, and orient the channels in exactly the same directions as they are oriented in $\mathcal{G}$. We then add the four channel nodes. This creates the turn graph for this set of channels as shown in Figure 8.29(c) (note the positions of the channel nodes).

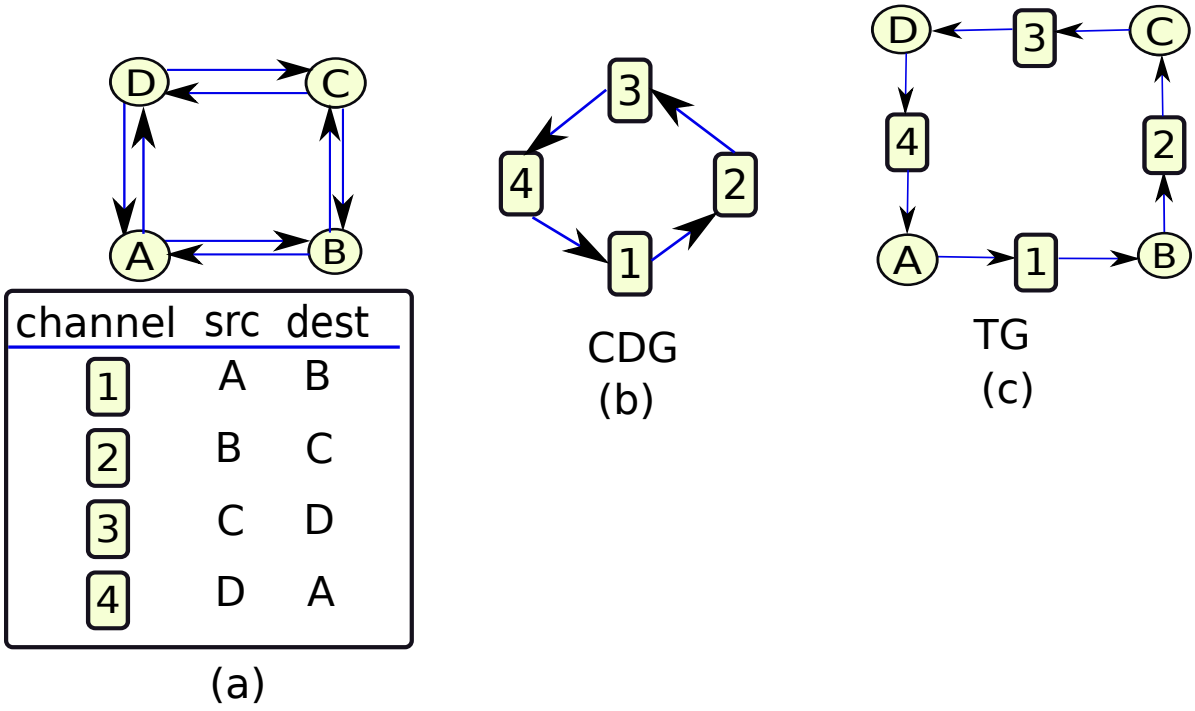


Figure 8.29: (a) Graph $\mathcal{G}$, (b) CDG with a cycle, and (c) its equivalent TG.

Let us quickly understand the benefit of having a turn graph. Consider an edge between two channels in the CDG such as the edge between channels 1 and 2. This translates to a *turn* in the TG between nodes A, B, and C. In fact, we can make some general observations here. Consider an edge between any two channels C_i and C_{i+1} in the CDG. Let C_i be between nodes N_i and N_{i+1} and C_{i+1} be between nodes N_{i+1} and N_{i+2}. Note that the channels have to share a node in common if there is a dependence. This translates to a sequence of three nodes (N_i, N_{i+1}, and N_{i+2}) connected via edges in the equivalent turn graph with channel nodes in the middle. Either all three nodes are collinear, or we make a *turn* while going from N_i to N_{i+2}. For us the **turns are of interest**, and we shall see that a study of such turns underpins the development of deadlock-free routing algorithms.

Let us now look at the most important property of a TG. Consider a CDG with a cycle. We can always consider a path $\mathcal{C}$ in the CDG that contains the cycle. We can then construct a TG for this path. Given that each channel in $\mathcal{C}$ has an associated channel node in the TG, and for any two channels $C_i, C_j \in \mathcal{C}$ $(i < j)$, there is a path from the channel node corresponding to C_i to the channel node corresponding to C_j, we can say that we shall have a cycle in the TG as well. This path will comprise the same set of channels that have the cyclic dependence in the CDG. We can also say that if for a given routing protocol, we cannot construct a TG with a cycle, then we cannot have a path in the CDG that has a cycle – this means that the routing protocol is deadlock-free.

Important Point 15
If for a given routing protocol ,we can never construct a TG with a cycle, then we cannot have a path in the CDG that has a cycle – this means that the routing protocol is deadlock-free.

Avoiding and Recovering from Deadlocks

Let us look at methods to handle the issue of deadlocks. There are two important concepts in this field.

Deadlock Avoidance or Prevention The first is that we avoid or prevent deadlocks by design[3]. This means that we design the routing protocol in such a way that deadlocks do not happen. Such approaches are also known as pessimistic approaches because we deliberately constrain the routing protocol to avoid deadlocks. This means that we somehow ensure that in all possible turn graphs that we can create for executions with a given routing protocol, there are no cycles. We sacrifice some performance in this process.

Deadlock Recovery The other approach is an optimistic approach. Here we choose the most efficient method of routing and allow deadlocks to happen. If we detect a deadlock, then we initiate a process of recovery. The process of recovery involves either deallocating resources by aborting a packet transmission or allocating some additional temporary storage to deadlocked flits such that they can progress.

Let us initially focus on deadlock avoidance mechanisms using specialized routing protocols. Subsequently, we shall look at deadlock recovery mechanisms.

8.3.3 Dimension-Ordered Routing

Let us consider the simplest class of deadlock-free routing protocols called dimension-ordered routing, the simplest of which is X-Y routing. Consider a mesh of nodes as shown in Figure 8.30. Each node has an x coordinate and a y coordinate. Let us describe the algorithm for routing a message from node A (x_1, y_1) to node B (x_2, y_2).

We first traverse in the x direction from (x_1, y_1) to (x_2, y_1). Then we traverse in the y direction from (x_2, y_1) to (x_2, y_2). In other words, we always give preference to the x direction over the y direction.

Let us evaluate this routing protocol using the three metrics that we have learned: number of links traversed, livelocks, and deadlocks.

This algorithm clearly yields the shortest path. We traverse the minimum number of links in both the axes. There is also no potential for a livelock because we never go round and round in circles. We move along the y direction only after we have completed all our moves in the x direction. In each axis, if we take the shortest route to the same row or same column as the destination, we are guaranteed to traverse through the shortest path.

Now, let us prove that such a routing strategy is also deadlock-free. Let us prove by contradiction. Assume that there is a deadlock. This means that there must be a cycle in the channel dependence graph (CDG). Let us consider the smallest cycle in the CDG. Since packets are not allowed to make a U-turn, we need to have some channels along the x-axis in the cycle. Let the channels (physical or virtual) be $C_1 \dots C_n$, where the cycle is $C_1 \rightarrow C_2 \rightarrow \dots C_n \rightarrow C_1$. Without loss of generality, let us assume that C_1 is along the x-axis. Let us visualize the channels as a turn graph (see Figure 8.31). We have annotated each channel node with the number of the channel.

[3]Some texts separately define the terms: deadlock avoidance and deadlock prevention. However, we shall combine the concepts and use a single term – deadlock avoidance.

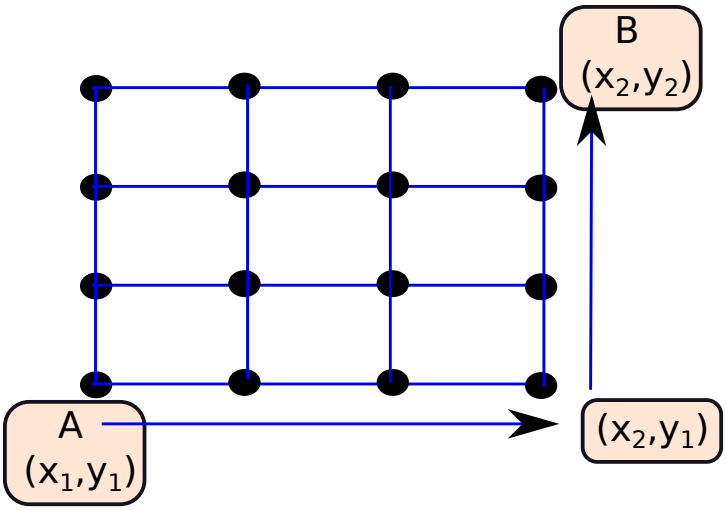


Figure 8.30: X-Y routing in a mesh

First, assume that C_1 is oriented towards the east and the cycle is anti-clockwise. Now, for the cycle to complete in Figure 8.31, we need to have the following turns: $E \rightarrow N$ (east to north), $N \rightarrow W$ (north to west), $W \rightarrow S$ (west to south), and $S \rightarrow E$ (south to east). The turn that we are interested in is between channels C_k and C_{k+1} (the $N \rightarrow W$ turn). This is not allowed in X-Y routing because it means that a packet first traverses along the y-axis, then moves along the x-axis. Such turns (packet movements) are strictly disallowed in X-Y routing. We thus have a contradiction, and this means that we cannot have a cycle in any TG, and by implication in any CDG with such cycles.

If C_1 is oriented towards the west, we can prove a similar result, and we can do the same for clockwise cycles. This means that the equivalent TG and CDG are always cycle-free.

Hence, deadlocks are not possible and the X-Y routing protocol is deadlock-free. This protocol is ordered by the dimension: first x and then y. If we have more dimensions such as in a 3D mesh network (in a 3D chip), then we can also use this approach by first ordering the dimensions.

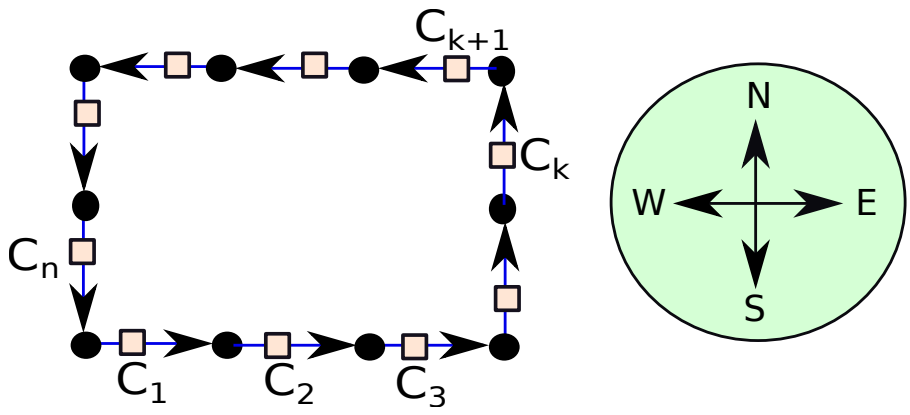


Figure 8.31: A cycle in the turn graph. The nodes are shown with filled, dark circles and the channel nodes are shown with squares.

8.3.4 Oblivious Routing

The main problem with X-Y routing is that the path between every source-destination pair is fixed. This is a desirable feature if we want simplicity; however, it does not allow us any flexibility and if there is congestion in the network, there is no way to route around it. As a result, such algorithms do not perform well when we have congestion in networks. Their lack of flexibility can prove to be a major detriment in networks with some degree of congestion in nodes.

There is thus a need to create an algorithm that provides far more flexibility. This is only possible when we do not have a fixed path between each source-destination pair. Let us thus describe the Valiant's algorithm. The algorithm is very simple; the steps are as follows.

1. Assume we want to route a message from point A to point B. Choose a random point P in the mesh. Let us call it the *pivot point*.
2. First route the message from A to P using a provably deadlock-free algorithm such as X-Y routing.
3. Then route the message from P to B using a similar algorithm as used in the previous step.

Let us understand the pros and cons of this algorithm. The obvious advantage is that we are randomizing the route. By choosing an intermediate point we are ensuring that we do not have a deterministic route between the source and the destination. As a result, it is much easier to not get stuck in hotspots. In addition, because of the randomization, the traffic will uniformly spread throughout the chip. It will be hard for traffic hotspots to even form in the first place.

On the flip side, the main shortcoming of this algorithm is that it lengthens the path from the source to the destination. This is because the pivot point might be far away from both the source and the destination. While sending a message from A to B, we need to incur the additional latency involved in sending the message from A to P and then from P to B. We thus observe a trade-off between the probability of avoiding network congestion and the end-to-end latency. Such kind of routing is known as *oblivious* routing because the source and the destination nodes are effectively unaware of each other. The source just needs to be aware of the pivot point, and the pivot point simply needs to be aware of the destination. This algorithm is otherwise free of deadlocks because in each step we use a provably deadlock-free algorithm such as X-Y routing.

Minimally Oblivious Routing

Let us try to address the main shortcoming of oblivious routing by introducing a more efficient version known as *minimally oblivious routing*. Here, we place restrictions on the pivot point. We do not allow it to be randomly placed at any point in the network. Instead, we demarcate a region around the source or destination and constrain the pivot point to only be within that region. This will ensure that the additional detour involved in going from the source to the destination via the pivot point gets reduced as much as possible.

Let us consider the case where the pivot point is placed in the vicinity of the destination. This situation is shown in Figure 8.32. In this case, we first route the message from A (source) to the pivot point P. Now, given the fact that P is in the vicinity of B (destination), we do not incur a very significant penalty (in terms of latency) by using this method.

This method represents an intermediate point between fully oblivious routing, and deterministic shortest path routing. There is a trade-off between the degree of congestion experienced by a packet, and the length of the path that it takes. Note that it is not necessary that the routing protocol be a variant of X-Y routing. We can use any routing algorithm that has deadlock freedom as one of its properties.

At this point, let us briefly glance back. We first looked at X-Y routing, which is provably deadlock-free. It is however very restrictive in its choice of paths. There is no *path diversity*, which is defined as the set of paths that can be taken from a source node to reach a given destination node. As a result, even

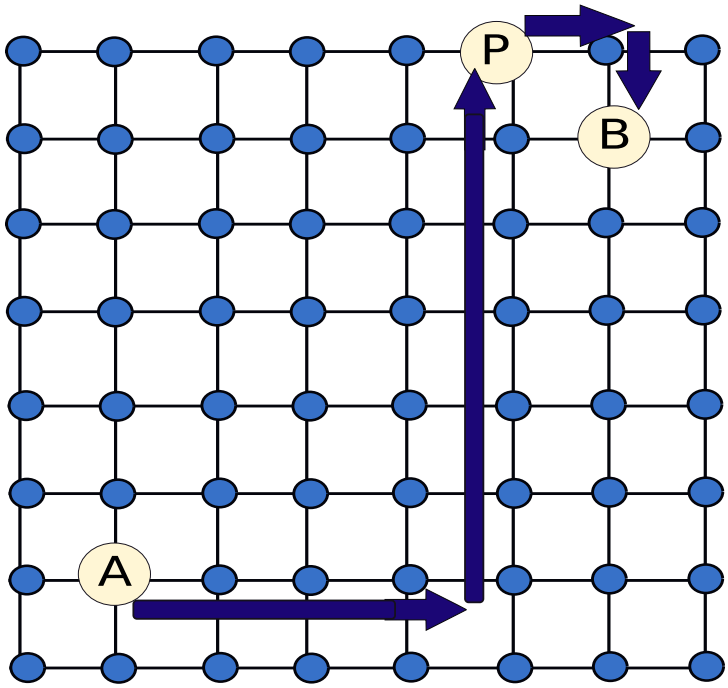


Figure 8.32: Minimally oblivious routing

if there is congestion along the way, nothing can be done. To a certain extent, this problem is solved with oblivious routing; however, we need to limit the path diversity such that we can avoid extremely long and circuitous routes between the source and destination. Hence, we proposed minimally oblivious routing that achieves a trade-off between these conflicting requirements.

Definition 60
Path diversity *is defined as the number of paths from a source node to a given destination node.*

8.3.5 Adaptive Routing

Whenever we make a routing decision at a router, we run the risk of creating an edge in the CDG between two VCs (virtual channels): the input VC (VC_1), and the output VC (VC_2). An edge will be created in the CDG if a flit coming in via VC_1 is not able to leave the router via VC_2 because the latter does not have enough space in its buffers.

There are two ways in which VC_1 and VC_2 can be related. Either they are in the same direction, or they are in perpendicular directions (turn). For the time being, let us not consider U-turns. Moreover, since all the flits in a packet flow along the same path, if the head flit takes a turn, the rest of the flits do the same.

In this section, we shall argue that if we do not restrict turns in our routing algorithm, then there is a possibility of deadlocks. Let us thus look at the notion of turns in some detail.

Notion of Turns

Traditionally, the directions in a mesh based network are represented as *north*, *south*, *east*, and *west*. They are abbreviated as N, S, E, and W respectively (also see Figure 8.33).

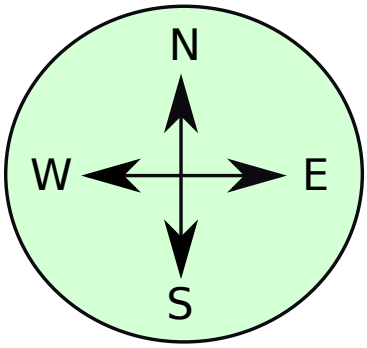


Figure 8.33: Directions used in routing

Every router in a mesh or torus has five input ports and five output ports. The five input ports (ingress ports) are N, S, E, W, and local. Recall that we divide a chip into a set of *tiles*, where each tile consists of a few adjoining cores and cache banks. Each tile has a router associated with it. When any core or cache within a tile desires to send a packet on the network, it sends a message to its local router. The local router accepts the message via the *local* port. It then sends the message via its output (or egress) ports. On similar lines, it has 5 output ports: N, S, E, W, and local. To deliver a message to the attached tile, the *local* output port is used.

A flit can either continue straight through a router, or take a turn. There are four possible ways of going straight: continue north, south, east, or west. Going straight by itself is not a problem and does not lead to deadlocks. We have deadlock situations when flits take turns because only then a situation with a circular wait can form. Let us thus look at the space of turns.

A flit going north can take two possible turns: go west, or go east. Let us designate these turns as $N \rightarrow W$ and $N \rightarrow E$ respectively. Similarly, for all the other directions, there are two possible ways in which we can take turns. There are thus 8 possible turns that a message can take. Let us quickly look at the number of turns that are allowed in X-Y routing (see Section 8.3.3). Recall that when a flit is traveling along the y-axis, it cannot take a turn in the x direction. This means that if we are going north or south, we cannot take a turn. This automatically precludes 4 turns: $N \rightarrow E$, $N \rightarrow W$, $S \rightarrow E$, and $S \rightarrow W$. The only 4 turns that are allowed are $E \rightarrow N$, $E \rightarrow S$, $W \rightarrow N$, and $W \rightarrow S$. This is precisely why we have maintained that the X-Y routing algorithm is very restrictive in terms of paths: it allows only 4 out of 8 turns. Let us propose algorithms that allow more turns and also guarantee deadlock freedom.

Cycles

We can always take a complex cycle in the TG and simplify it by fusing a chain of dependences to form a single dependence till it becomes equivalent to one of the cycles shown in Figure 8.34. The cycle is either clockwise or it is anti-clockwise. The rest of the edges do not matter. We are the most interested in the specific turns that create these cycles.

The key learning here is that in a clockwise cycle, we shall definitely have 4 turns: $E \rightarrow S$, $S \rightarrow W$, $W \rightarrow N$, and $N \rightarrow E$. Similarly, in an anti-clockwise cycle, we shall also have these 4 turns: $W \rightarrow S$, $S \rightarrow E$, $E \rightarrow N$, and $N \rightarrow W$. From each cycle, if we can eliminate at least one turn, then we are sure that cycles will not form: clockwise or anti-clockwise. If cycles do not form in the TG, then as we have discussed earlier, there is no possibility of a deadlock.

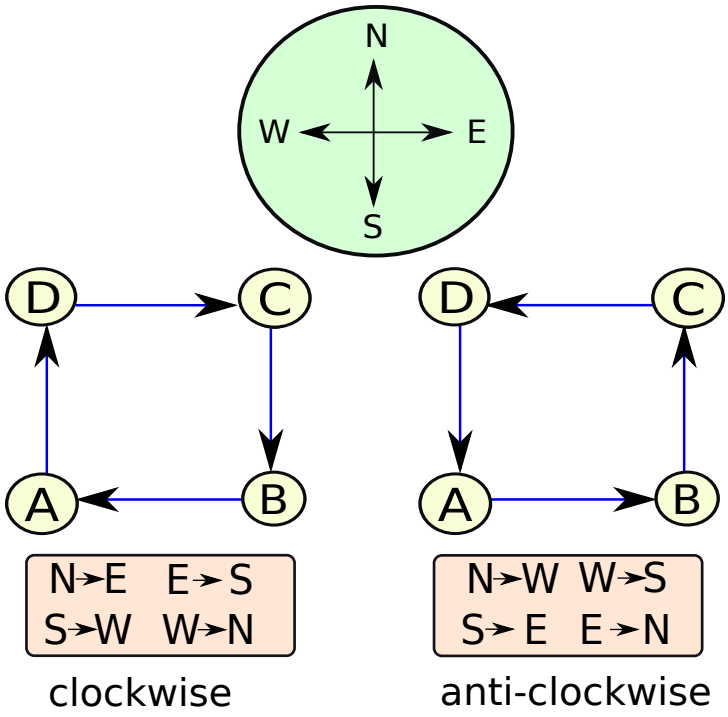


Figure 8.34: Basic turns in a turn graph

If we consider the X-Y routing protocol again, we observe that it prohibits 4 turns. Two of these turns, $N \rightarrow E$ and $S \rightarrow W$, are present in a clockwise cycle. Since we are not allowed to take these turns, we shall never have a clockwise cycle. Similarly, two other turns that are prohibited and are part of an anti-clockwise cycle are $S \rightarrow E$, and $N \rightarrow W$. Hence, we shall also never have an anti-clockwise cycle. As a result, the TG with an X-Y routing protocol is acyclic, and the protocol is thus deadlock-free.

We can clearly do better than X-Y routing. We need not prohibit that many turns. We just need to prohibit one turn each in the two cycles – clockwise and anti-clockwise. This means that by allowing 6 out of the 8 turns, we can create deadlock-free routing algorithms.

Deadlock-free Routing Algorithms with 6 Turns

For each cycle, we need to prohibit one turn. We have four possible choices for each cycle (clockwise and anti-clockwise), and thus there are 16 possible choices overall. We can thus design 16 possible routing algorithms that allow 6 out of 8 turns. Some of these routing algorithms have a name. Let us review a few of them in Table 8.2.

In the West-first algorithm we always go west first. The second direction is never west. This is why we disallow the turns $N \rightarrow W$, and $S \rightarrow W$. On similar lines, we have the North-last protocol, where we always go north at the end. In the Negative-first protocol we go in the negative directions – west or south – at first. We can create many more such algorithms and assign names to them. The key idea is that with such protocols, we have armed ourselves with more turns such that we have a choice of more routes, and we consequently have more path diversity. This will also allow us to deal with congestion better.

The entire family of such protocols is deadlock-free. We just need to avoid two turns, and it is guaranteed that we shall not have any deadlocks. The next question that arises is, "How do we ensure that we always choose the best possible route in the face of congestion?" This question will be answered

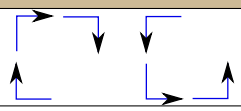
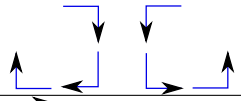
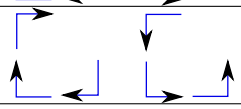
Name	Turns disallowed	Allowed turns
West-first	$N \rightarrow W$, $S \rightarrow W$	
North-last	$N \rightarrow W$, $N \rightarrow E$	
Negative-first	$N \rightarrow W$, $E \rightarrow S$	

Table 8.2: Routing algorithms with 6 allowed turns

in Section 8.4 when we discuss the design of the router.

8.3.6 Preventing Deadlocks by using Virtual Channels

We have discussed a host of routing algorithms where we prevent deadlocks by disallowing a certain set of turns. This ensures that we do not form cycles: clockwise or anti-clockwise. However, there are other methods to ensure that we do not have deadlocks, and some of these mechanisms are based on innovative uses of virtual channels. A historical note is due here. Virtual channels were originally proposed to prevent deadlocks, and they later on got adapted for a host of other things as we have been seeing in the last few sections.

Let us try to understand why VCs are well suited to avoid deadlocks. VCs are channels in their own right because they have their own set of buffers and all physical channels are multiplexed between their VCs. Hence, in the channel dependence graph, we have VCs and not physical channels. Restricting turns is one of the ways in which we can solve a cycle forming among the VCs; however, we can also prevent cycles by placing other kinds of restrictions on the way we select and use VCs.

Routing in a 4-Node System

Let us consider a very simple example in Figure 8.35. In this case, we have 4 nodes (A, B, C, and D) and 4 channels with 2 VCs per channel. As we have seen in Figure 8.28, we can have a deadlock in such an NoC.

Let us number the two VCs corresponding to each physical channel as 0 and 1. The two VCs from node A to B are named AB_0 and AB_1 respectively. Other VCs are named in a similar manner. Now, between nodes A and B let us draw an imaginary line called the *date line*[4]. The VC assignment algorithm is as follows. Whenever we inject a packet into a router, we always inject it into VC 0. It travels only via the VCs numbered 0 till it either reaches its destination or till it crosses the date line. Once it crosses the date line, it transitions to the VCs numbered 1. For example, when it traverses through the VC AB_0, the next VC that it needs to be allocated is BC_1. Henceforth, the message will remain in the VCs numbered 1 till it reaches its destination. A flit will never move from a VC numbered 1 to a VC numbered 0. Let us prove that this algorithm is free of deadlocks.

Theorem 8.3.6.1 *The routing algorithm using a date line in a 4-node system with 2 VCs per channel is deadlock-free.*

[4]This is conceptually inspired from the international date line on the globe.

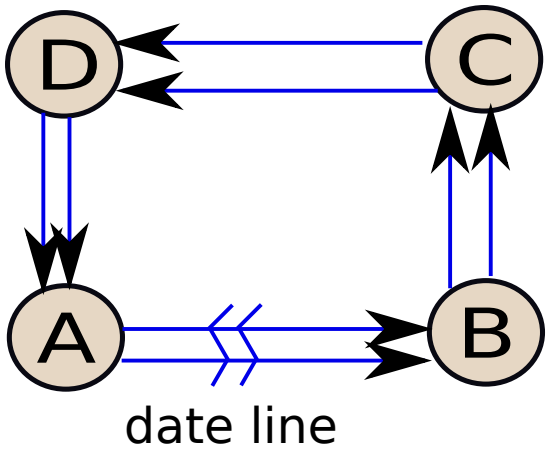


Figure 8.35: Network with 4 nodes and 2 VCs per channel

Proof: Assume that this protocol leads to a deadlock. Let us show prove by contradiction that this is not possible. There are three cases: we have a cycle with only channels numbered 0, we have a cycle with channels numbered 1, or we have a cycle with channels numbered both 0 and 1.

Case I: We have a cyclic dependence with VCs numbered 0. Since U-turns are not allowed, the cycle will consist of 4 edges: AB_0, BC_0, CD_0, and DA_0. Now there cannot be an edge between AB_0 and BC_0 in the CDG because by the definition of the date line, there can only be an edge between AB_0 and BC_1 in the CDG. Hence, such a cycle is not possible.

Case II: In this case, all the channels in the cycle are numbered 1. Consider the case of the edge between channels AB_1 and BC_1. This means that a flit has acquired the VC AB_1 and wishes to acquire the VC BC_1, which is busy. This is not possible. To acquire a channel numbered 1, it must have already traversed the channel AB_0 before. This means it must have moved through only channels numbered 1, and visited the rest of the nodes before traversing the physical channel between A and B once again. If a flit has visited all the nodes in an NoC (as in this case), it must have visited its destination also. Therefore, the flit should not have been in circulation anymore, and thus there is no way in which it is possible to have a dependence between the channels AB_1 and BC_1 in the CDG. This case is thus not possible.

Case III: Consider a cycle with channels that are numbered both 0 and 1. There has to be an edge in the CDG between a channel numbered 1 and a channel numbered 0. This is not allowed in our routing protocol. Hence, this case will never happen.

Thus, we prove by contradiction that it is not possible to have a cycle in the CDG. ∎

Even though this approach looks easy, it has two important drawbacks.

1. Creating a date line in a simple 4-node network is easy. However, a similar mechanism in a larger network with a complex topology is difficult.
2. The VCs numbered 1 are relatively less utilized as compared to the VCs numbered 0. This design choice unequally utilizes the system, and is thus wasteful in terms of resources.

The key learning in this scheme is as follows.

To avoid creating deadlocks, impose an order in which resources are acquired.

Routing in Rings

Let us now extend this result to an N-node ring, where naive protocols can have deadlocks (see Example 7). Moreover, X-Y routing cannot be used in rings because a y-axis does not exist.

We can use a similar date line based approach by using two virtual channels. Let us arbitrarily define a date line between nodes 1 and 2. In this case, it is easy to extend the proof of Theorem 8.3.6.1 to show that a similar algorithm using two virtual channels is free of deadlocks.

Example 7 *Show that it is possible to have deadlocks in a ring where the routing protocol always constrains a flit to move in the clockwise direction. Assume that we have a single VC per physical channel in the clockwise direction and each VC has k buffers.*

Answer: *Consider the following communication pattern. Node i tries to send k flits to node $(i+2)\%N$ (numbers increase in a clockwise direction). Let us introduce the operator $+_N$, which is defined as follows: $a +_N b = (a+b)\%N$. Here '%' is the remainder or modulo operator.*

Now consider the following sequence of events. Each node i sends k flits from node i to $i +_N 1$. All the nodes do this simultaneously. Subsequently, any given node i will not be able to make progress because the channel between $i +_N 1$ and $i +_N 2$ is occupied by the flits being sent from node $i +_N 1$. Thus, the flit gets blocked. In the equivalent CDG there is an arrow between the channel $\langle i, i +_N 1\rangle$, and $\langle i +_N 1, i +_N 2\rangle$. Note that we have such dependences for all i, and this leads to a cycle in the CDG. No flit will be able to move to its second channel in the route because that channel is blocked. This is a deadlock.

Routing in Tori

Let us now consider a torus (see Section 8.1.2). If we consider the traffic pattern along a single row or column, which is arranged as a ring of nodes, deadlocks are possible as we saw in Example 7.

To avoid such deadlocks, let us define two date lines: one along the x-axis, and one along the y-axis. The date lines intersect every row and every column. These ensure that while traversing a row or column in the torus, deadlocks are not possible. We can then use X-Y routing as the overall scheme albeit with the additional constraint that when we cross a dateline we transition to the VC numbered 1. With both of these protocols, we can ensure that routing in tori is free of deadlocks.

Escape VCs for Deadlock Recovery

Let us now look at a radically different idea. Consider a simple example with two VCs per physical channel. Let the message be ordinarily constrained to VC 0. Let us use a fast shortest path based routing protocol for the set of edges that use VC 0. This sub-network gives us efficiency, yet it can lead to a deadlock.

Let us propose a method for deadlock recovery. This can be easily achieved by associating a counter with every flit in a router. When it enters the router, we initialize the counter. Every cycle we increment the counter. Once the flit leaves the router we reset the counter. However, if the counter reaches an upper threshold, then it means that the flit is stuck at the router for a long period of time. This can possibly indicate a deadlock. For practical purposes, let us assume that this does represent a deadlock situation and try to recover from it.

Deadlock recovery in conventional networks involves dropping packets. However, in the case of on-chip networks, we are not allowed to drop packets. Hence, we need to find another method of recovering from deadlocks. Let us have another set of VCs (numbered 1). Consider the sub-network comprising VCs numbered 1. In this sub-network, let us use a different routing protocol that is provably deadlock-free such as X-Y routing. Any flits that gets possibly stuck in a deadlock moves to the sub-network

with VCs numbered 1, and travels to the destination in this sub-network. Since the routing process is deadlock-free in this sub-network, it is guaranteed that the flit will reach its destination.

Now, if we consider the entire system, it is also free of deadlocks. The sub-network with VCs numbered 0 can suffer from a deadlock. However, flits will not wait for an indefinite period. After a certain amount of time has elapsed, they will transition to the other sub-network (VCs with number 1). In the latter sub-network, the flits are guaranteed to reach their destination because we use an algorithm that will not have deadlocks.

The advantage of this network is good performance. We use a high performing sub-network (VCs numbered 0) to route flits. However, occasionally we might have a deadlock. In such cases, we move the flits to a much slower sub-network (VCs numbered 1) that guarantees the delivery of flits to the destination without the possibility of deadlocks. The only disadvantage of this design is that the VCs with number 1 are relatively less frequently utilized as compared to the VCs with number 0. This represents a wastage of resources.

We can further extend this idea to a network that has 4 or 8 VCs per physical channel. We can divide the VCs into different classes, and have different routing algorithms for each class of VCs. We can use the same idea. If we discover a deadlock in the network, we move the flit to a VC in another class that uses a provably deadlock-free routing algorithm.

8.4 Design of a Router

Let us now look at the design of a router. A router is arguably the most important component in the NoC. It includes all the logic and state to store, forward, and route flits. It also includes the logic to allocate virtual channels, detect deadlocks, and perform flow control. In a certain sense, we will combine everything that we have learned in the past few sections in this section.

A router in a modern NoC is a highly complex pipelined structure. The reason we need to pipeline it is that the latency of a router is fairly high, and thus to sustain a high throughput, pipelining is required. In this section, we shall first describe a quintessential 5-stage router and then propose optimizations to reduce the number of stages. We shall show in later subsections that it is possible to reduce the number of stages from 5 to 3 by making sophisticated optimizations.

Let us explain the design of a router that can be used in a 2D mesh or torus. Akin to normal in-order pipelines, it also has 5 stages.

8.4.1 Input Buffering

Consider the start of the packet transmission process. A core or cache bank decides to send a message over the network. It creates a packet and then sends the sequence of flits to its nearest router. Each router has a dedicated input port for local traffic. This input port can be used by all the senders and receivers in its tile (a set of cores and cache banks). If we are using a packet or circuit switched network, then we need a single buffer queue. Otherwise, the input port will have multiple buffer queues, where each such queue is a VC. The requests can be fairly allotted to VCs such that we do not have starvation.

Now, consider an intermediate node in the network. If we have a packet or circuit switched network, there will be a single set of buffers associated with each physical channel (refer to Figure 8.36(a)). We simply start writing the packets to the buffer queue associated with the physical channel at the destination router. However, in the case of flit based switching with VCs, the situation is slightly more complicated. In this case, we shall have multiple queues associated with the same physical channel. Each queue corresponds to a VC. Hence, every flit that enters the router needs to know the id of the VC that it is using. Assume a system with 2 VCs per physical channel as shown in Figure 8.36(b), which are numbered 0 and 1 respectively. Every flit entering router B from router A needs to know the id of the VC that it belongs to. If it is using VC 0, it gets queued in the queue for VC 0 in router B, and likewise if it is using VC 1, it gets queued in the buffer queue corresponding to VC 1.

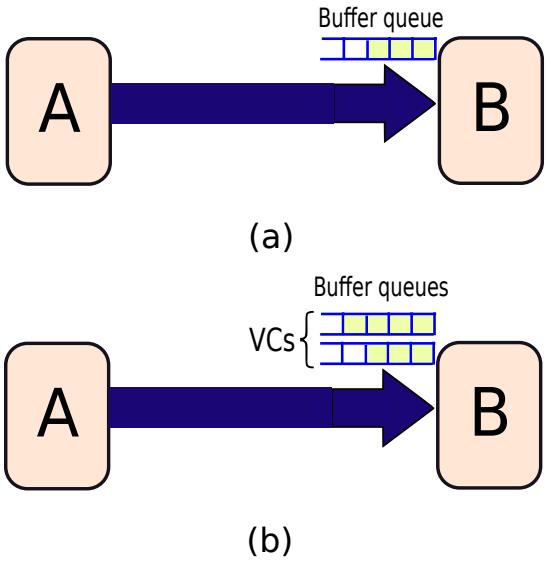


Figure 8.36: Input buffering: (a) without virtual channels, (b) with virtual channels

The process of writing flits to buffer queues, is known as *input buffering*. This ensures that we are able to store flits in the downstream router (B in this case). There are two possible ways of implementing input buffering. The first is to use separate queues (SRAM arrays in hardware) for each VC. This is a very easy and simple solution. However, it suffers from several drawbacks. The first is that it limits the storage capacity at the routers. Even if the other queues (for other VCs) are free, we will not be able to use their space.

Hence, if storage space is our primary concern, we can use a shared buffer based approach as shown in Figure 8.37. In this case, we have a single, large array of flits. Different VCs are assigned to different non-overlapping regions of this array. Conventionally, the two ends of a queue are referred to as the *head* and *tail* respectively. The only state that we need to maintain in this case for each VC is the id of its corresponding *head* and *tail* pointers. Whenever we insert or remove a flit from a queue, we need to update these pointers. The advantage of this design is that the storage structure is more flexible. If a given VC needs a lot of space at any given point in time, and the rest of the VCs are unutilized, then it is possible to accommodate this requirement using this design.

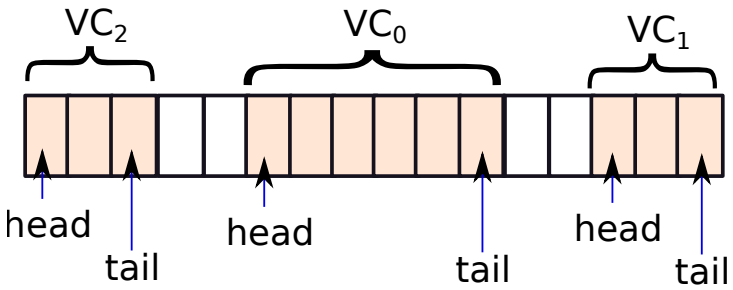


Figure 8.37: All the VCs using a single shared buffer

8.4.2 Route Computation

The next stage of packet processing is to compute the route that the packet will take. This needs to be done once, only for the head flit. The rest of the flits in the packet will follow the same route. The head flit needs to have the id of the destination node. In addition, as discussed in Section 8.3.1, it can have more information such as the number of hops it has already traversed to avoid livelocks.

A dedicated route computation (RC) engine needs to compute the id of the next router. There are 4 possible directions in which the packet can be sent: north, south, east, and west. There are several issues that we need to keep in mind. In general, we should always try to send a packet on the shortest path to its destination. However, there can be issues related to congestion, and thus sometimes taking longer paths can save time, and avoid network hotspots. Let us look at some common methods for computing routes.

Source Routing

In this case, the entire route is computed a priori. The route is then embedded into the head flit. At every node, the router checks the contents of the head flit. The route is stored as a queue of directions in the head flit. It removes the head of the queue, and sends the packet along that direction. Ultimately when we arrive at the destination, the queue of directions in the head flit becomes empty.

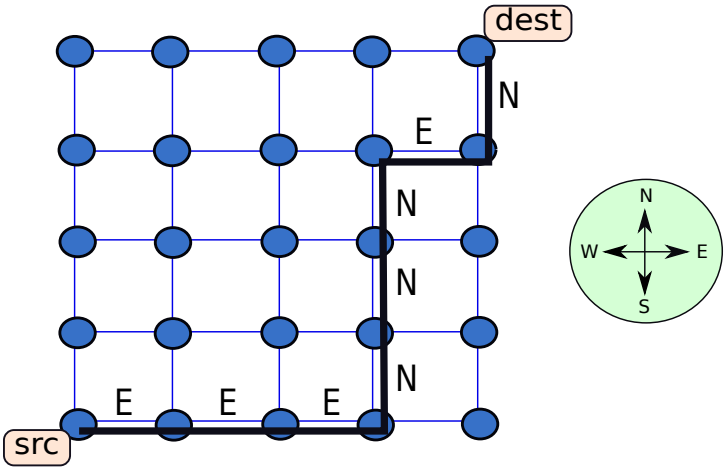


Figure 8.38: Example of source routing in a mesh network

Figure 8.38 shows an example of source routing. In this case, the original route from the source to the destination is computed to be $EEENNNEN$. Each letter in this route represents the direction the flits should take. In every node we remove the head of the queue, which is a direction D, and send the packet along direction D. The disadvantages of this scheme are obvious. We always compute static routes, and we have no means to deal with congestion in the network or take dynamic decisions. Sometimes it is possible that the temperature of a given zone within the chip rises to unacceptable levels. It is necessary to turn off all activity in the area, and also route NoC packets around it. Such strategies are not possible with source routing. In addition, there are overheads to store all the routing information in the head flit, and modify the head flit at every router. The head flit needs to be modified because we need to remove the entry at the top of the queue that stores the list of all the directions in which the packet needs to be sent.

However, the scheme is not completely devoid of advantages. It is simple, and we do not need to compute the routes dynamically. This saves power. To create path diversity we can store multiple routes for the same destination, and randomly or on the basis of expected congestion, choose between them.

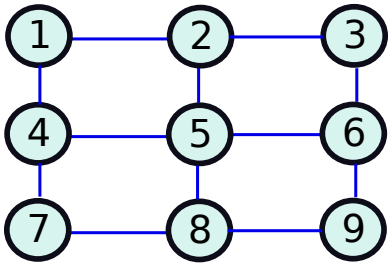
Routing based on Node Tables

Let us explore a different kind of approach. Instead of doing the entire routing at the source, let us instead do the job of routing at the routers. This means that every time a packet reaches a router, we simply compute which outgoing link the packet is going to take. In the case of a 2D torus or mesh, we have four choices: N, S, E, and W.

We keep a table at each router, known as the *node table*. For each destination, we store the directions that the packet needs to take at the egress (exit point) of the router (refer to Example 8). Note that in a routing algorithm based on turns, it is also necessary to have multiple rows in this table: each row corresponds to a direction from which the packet has arrived. For each pair of direction and destination, we have an entry in the table. This contains the direction that the packet needs to take as it exits the router. Note that in some cases, it might be possible to send the packet along multiple directions.

Example 8

For the given network, compute the node table at node 5 for the North-last routing algorithm. Instead of showing all the columns, show only the column for a flit coming from the local tile. Note that we always prefer the shortest path.



Answer: *Since we are using the North-last routing algorithm, the $N \rightarrow W$ and $N \rightarrow E$ turns are disallowed. The node table without these turns is as follows. Here, the operator '|' stands for 'OR' (any of the specified routes can be taken).*

Destination	*Direction*
1	W
2	N
3	E
4	W
6	E
7	$S \mid W$
8	S
9	$S \mid E$

Note that for nodes 1 and 3, we could have proceeded north as well. However, since the turns $N \rightarrow W$ and $N \rightarrow E$ are disallowed, we discarded these routes from the node table.

When we have a choice of multiple directions we need to base our routing decision on several considerations.

1. The first is livelocks. If a packet has already traversed a given number of hops, it should be sent preferably along the shortest possible path to its destination. Otherwise, we will have a scenario where the packet will be moving around in circles and not reaching its destination.

2. If livelocks are not an issue, then we should try to minimize the time it will take to reach the destination. When there is no congestion, the least-time path is the shortest path. However, in the presence of congestion, alternative routes become more favorable. We can use the flow-control circuitry to get an idea of the average buffer usage/occupancy at the neighboring routers. Once we have this information, we can choose the next hop based on a combination of the following information: distance to the destination, number of free virtual channels in the next hop, and the rate of buffer usage for each virtual channel. We can also use a weighted sum of these quantities (appropriately normalized). The individual weights need to be determined by conducting exhaustive simulation studies. We can also use multi-hop information if we have it with us such as the congestion in local or remote neighborhoods.

Note that it is not always necessary to maintain tables. It takes space to store the tables, and it also requires energy to access a given row and column in the table. In some cases, where simplicity is required, we can use a simple combinational circuit to compute the route that a packet needs to take. This is very easy to do with schemes such as X-Y routing or some simple turn based schemes.

8.4.3 Virtual Channel Allocation

After computing the route, we need to start the process of sending the packet across the computed outgoing link. A given link or physical channel will have many VCs associated with it. There is a need to allocate a virtual channel to the packet. This process is known as VC allocation. Note that the problem of virtual channel allocation is not simple. It is possible that different packets, which have arrived via different physical channels could be vying for the same set of virtual channels.

Thus, the prime goals of the VC allocator are as follows:

- The VC allocation process should be fair. No request should be made to wait for an indefinite period. This will lead to starvation.
- It should be done quickly in less than one cycle.

The VC allocator can be modeled as a black box. In every cycle it takes a set of requests, and returns the number of the VC that was allotted to each request. We shall look at the general problem of resource allocation in Section 8.4.6. Hence, we are not discussing VC allocation separately.

The design of the router as described till this point is shown in Figure 8.39. We are assuming two VCs per physical channel. In the first stage, we store the flit in the input buffers corresponding to the VC. Then in the next stage we send the data to the route computation unit via a pipeline register. After we have computed the route, we allocate a VC in the next cycle. We do not have a pipeline register at the end of this stage because we assume that the allotted VC is written to the head flit of the packet.

8.4.4 Switch Allocation

Once we have allotted a virtual channel, the packet is ready to be transmitted via an outgoing link (physical channel). However, this is a complex process. Consider a representative example. Assume we have two VCs per physical channel. For the N, S, E, and W incoming channels we have a total of 8 VCs. In addition, we shall have 2 more buffer queues (same as VCs from our point of view) for local traffic. We thus have 10 input VCs. It is possible that all of these VCs have flits in them, which are waiting to be transmitted via outgoing links. In this case, we have 5 outgoing links: local, N, S, E, and W. Thus, the problem of resource allocation is as follows. We have 10 input channels and 5 output channels. To effect a packet transmission, we need to map an input to an output. Such a hardware structure is known as a *switch*. The 10×5 switch for our router is shown in Figure 8.40.

Realize that it is possible that all 10 packets at the input VCs might want to pass through the same outgoing channel. However, at any one time, only one flit can pass through an outgoing channel. Hence,

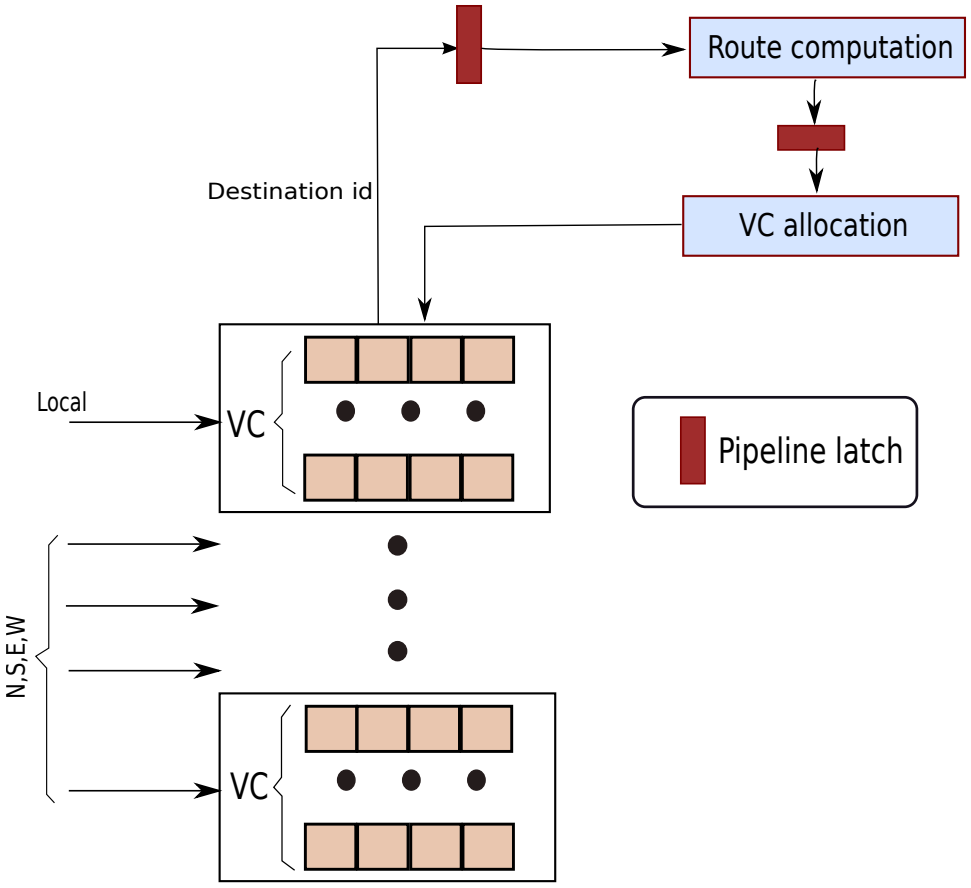


Figure 8.39: Router with the first three stages: input buffering, route computation, and VC allocation

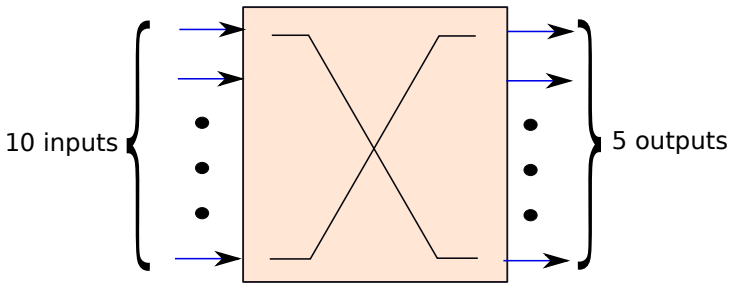


Figure 8.40: 10×5 switch

there is a need to choose one out of the several flits that are possibly interested to traverse an outgoing channel. At this stage, we can multiplex a physical channel among several virtual channels. This is where we need to be *fair* and ensure that every flit gets transmitted within a possibly bounded amount of time. If the switch allocator is fair, then starvation is not possible.

In addition, it is possible to define several conditions of optimality in this case. Consider a system where we have a priority associated with each packet. If we wish to ensure that all the high priority packets are routed first, then our switch allocator needs to give preference to flits in such packets. Thus,

the objective function here is to minimize the end-to-end latency of high priority packets. If we want to increase the system throughput by freeing VCs, then the strategy should be to quickly send all the body and tail flits, after a head flit has been sent. This will ensure that the VC and the associated buffers clear up as soon as possible.

Alternatively, if we wish to minimize the average end-to-end latency, then the best strategy is to often give preference to the smallest packets (least number of flits). In many cases it can be proven that this is indeed the most optimal strategy, and a better strategy does not exist. There are many more kinds of optimization strategies for different classes of networks and objective functions. Refer to the survey paper by Gabis et al. [Gabis and Koudil, 2016].

8.4.5 Switch Traversal

Once a port in the switch has been allocated an outgoing channel, the only thing that remains for a flit is to traverse the switch, and reach the outgoing link. This process is known as switch traversal. There are several ways to design a switch. Let us look at one of the simplest possible ways (refer to Figure 8.41).

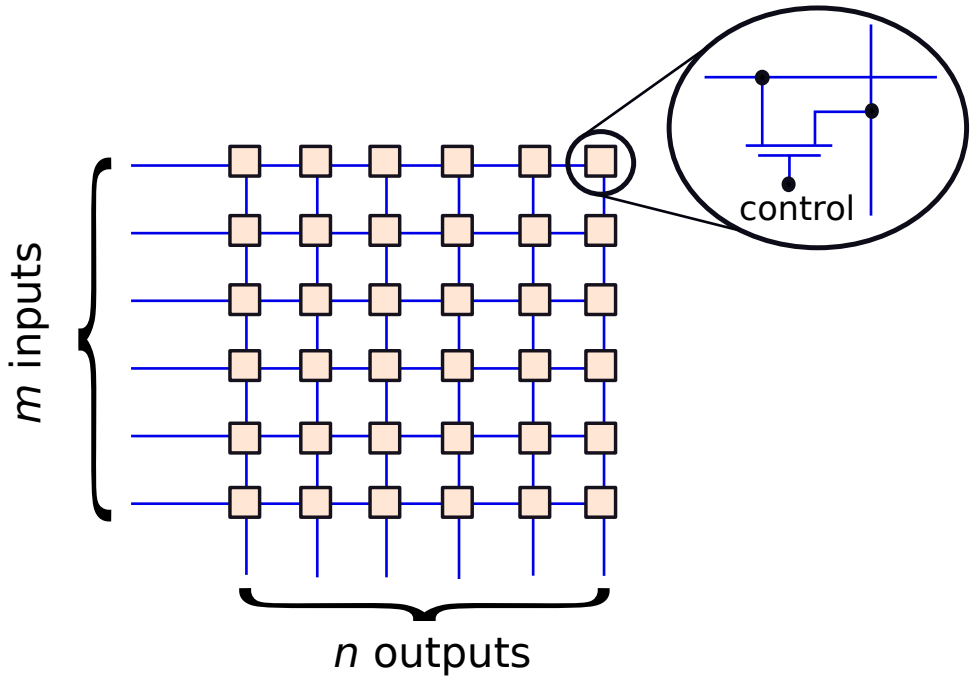


Figure 8.41: An $m \times n$ switch with pass transistors (a connection is made if $control = 1$)

We normally design a switch as an array of pass transistors, where the number of rows is equal to the number of input ports, and the number of columns is equal to the number of output ports. Here, each input port corresponds to an input VC, and each output port corresponds to an outgoing link (physical channel). We connect an input to an output by enabling the transistors between the horizontal and vertical wires. This creates a connection and data can thus be transferred from the input to the corresponding output. Note that any outgoing link can transfer a flit for only a single input VC at any given point of time. Each flit that is being transferred on a channel also contains the virtual channel id that it belongs to. Recall that the VC is allocated earlier in the VC allocation stage, and furthermore all the flits in the same packet are allocated the same outgoing link and the same VC.

The process of switch traversal is fairly simple. A connection is made between the horizontal and vertical connections by programming the transistors. The flit travels seamlessly towards the output port.

From the output port, the flit proceeds on the outgoing link. In almost all routers as of 2020, output buffering is not used. This means that at the outputs we do not have any buffers. As soon as a flit exits the switch, it is directly placed on one of the outgoing links. Such a switch that looks like a 2D matrix is known as a *crossbar*.

Optimized Designs of Crossbar Switches

Sadly, large crossbars are not considered to be optimal designs. The chief reasons are the usual suspects namely high power consumption, high area, and increased latency due to larger circuits. However, the key advantage is simplicity and support for high throughput. Given that the overheads often outweigh the advantages, typically a monolithic design is not preferred.

This means that if there are N input VCs and M output channels, an $N \times M$ switch is not preferred. It is considered to be expensive in terms of power, area, and latency. Let us create an approximate metric for area and latency. The area is proportional to the product $N \times M$, and the latency is proportional to the distance the flits travel, which is $M + N$. Let us define these metrics as the area cost and the latency cost respectively.

Thus, for a 10×5 router, the area cost is 50, and the latency cost is 15. It has full flexibility: any input can be connected to any output.

There are several strategies that could be adopted for creating a more efficient design. The first is that we use a hierarchical approach. Out of the 10 VCs, we can first choose 5: one for each incoming link. Then we can have a 5×5 switch. This hierarchical strategy is shown in Figure 8.42. We first have a set of five 2×1 switches, and then we have a single large 5×5 switch. The area cost is the sum of 5 times 2×1, and 25 (5×5), which comes to 35, and the latency cost is $2 + 1 + 5 + 5$, which is 13. The main problem with this design is that it does not allow both the VCs of a single physical channel to send flits simultaneously when they are destined for different outgoing channels.

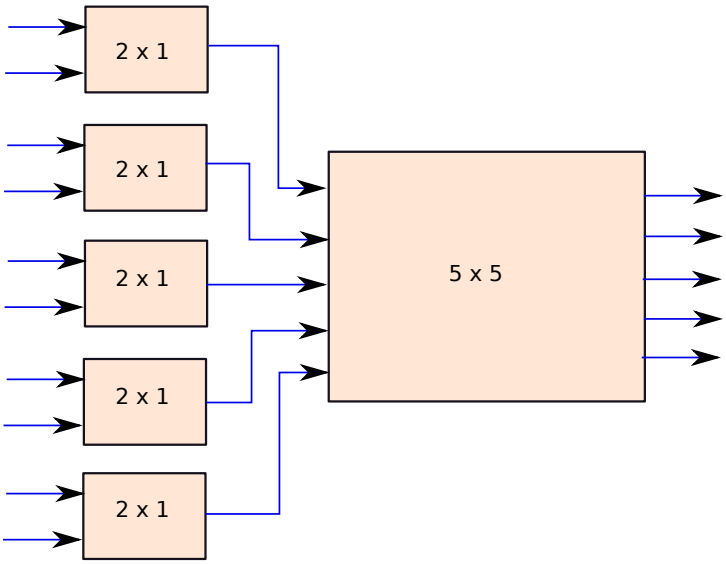


Figure 8.42: Hierarchical switch with 10 inputs and 5 outputs

Let us try to solve the problem with the hierarchical switch by creating a different kind of grouping. Let us again create a 2-level hierarchy, where the first level has two 5×2 switches. The insight here is that we do not always have enough traffic to keep all the 5 links busy. We can safely assume that for an overwhelming majority of time, a maximum of 4 packets are simultaneously flowing through the

outgoing links. The second level has one 4×5 switch. The area cost is 40 and the latency cost is 16. This represents a point between a fully hierarchical design and a fully flexible design in terms of the overall area (see Figure 8.43). Many more such designs are possible. Depending upon how we want to optimize area, power, and latency, we choose the appropriate design.

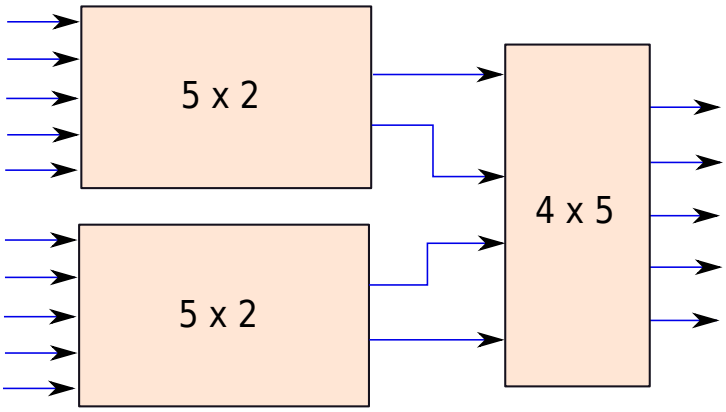


Figure 8.43: Hierarchical switch with wider internal switches

The three switch designs are summarized in Table 8.3. Let us now look at a different way of creating switches.

Design		Area cost	Latency cost
Level 1	**Level 2**		
10×5	-	50	15
Five 2×1	5×5	35	13
Two 5×2	One 4×5	40	16

Table 8.3: Area and latency costs of different switch designs

Dimension Slicing

This approach is suitable for protocols that use X-Y routing or other forms of dimension-ordered routing. Let us start out by noting that there is an asymmetry in the directions that a packet can take with this routing protocol. Packets traveling in the x direction can either go straight or take a turn. However, packets traveling in the y direction can only travel straight. Let us use this fact to design a more efficient switch.

First consider a switch that has three inputs: E, W, and *local*. Packets can be sent along any of its outgoing links. This switch has three outgoing links: E, W, and a link to another switch. It is thus a 6×3 switch as shown in Figure 8.44. Note that we are assuming two VCs for each physical channel. Now, all the packets that are traveling along the x axis can go straight through this switch. However, if they are destined for other outgoing links, then they need to go to the second switch as shown in Figure 8.44. The second switch has five inputs: one from the first switch, two VCs each corresponding to the N and S directions. There are three outputs: N, S, and *local*. This is a 5×3 switch. Quickly note that the only directions that the packet can take are north, south, and local. This is a direct consequence of the X-Y routing protocol. A packet from the north or south cannot take a turn towards the east or west. This helps us reduce the number of outgoing links.

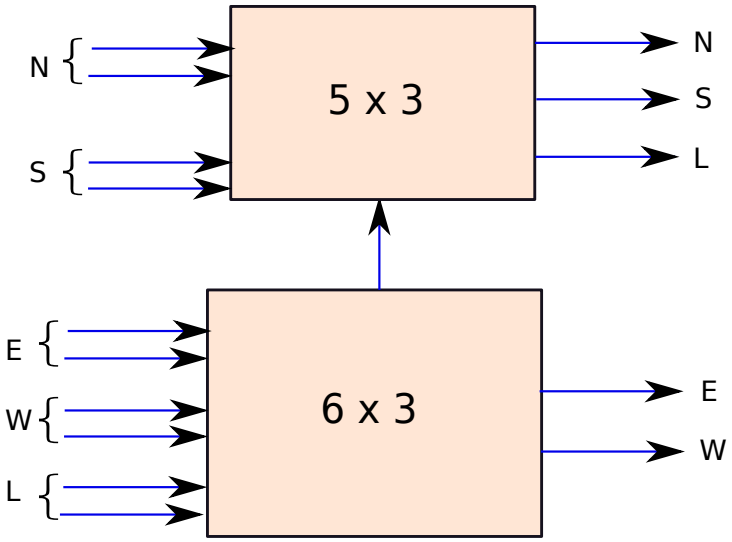


Figure 8.44: Dimension-sliced switch

The costs of this pair of switches are as follows: area cost = 33 and latency cost = 17. This is clearly the most area efficient solution that we have seen so far (refer to Table 8.3). We were able to design an efficient scheme because our routing protocol constrains the routes and thus some connections could be avoided.

Through this small example, we would like to highlight the fact that it is possible to co-design the routing protocol and the design of the switch.

8.4.6 Allocators and Arbiters

Given that we have discussed the need for VC allocation and switch allocation, let us look at the process of allocation in some detail. First, let us define some terms rigorously. An allocator creates a one-to-one match between N requests and M resources. For example, a virtual channel allocator has M virtual channels at its disposal for a given physical channel. In any given cycle, for a particular physical channel, it might get N requests, and it needs to allocate them M resources (VCs). If $N \leq M$, then there is no problem. We can do a simple one-to-one mapping for the N requests. However, the problem arises when $N > M$. In this case, we need to first choose a subset of requests that we are going to allocate, and then we can map them to the M resources. The process of choosing a subset of requests needs to be fair, in the sense that it should never be the case that a request is kept waiting for an indefinite period. This will lead to starvation.

The other important concept is *arbitration*. In this case, an arbiter chooses one out of N requests.

Definition 61
An allocator *creates a one-to-one mapping between a subset of N requests and M resources, whereas an* arbiter *is far more specific: it chooses one out of N requests for resource allocation.*

Theoretical Fundamentals

Let us generalize the problem of allocation. If we have N requests and M resources, then there are many possible ways of mapping the requests to the resources. We have been assuming that any request can be mapped to any resource. However, this need not be the case all the time. It is possible that a given request can only be mapped to a specific subset of resources. This is the general problem of allocation, where we need to find a *matching* between resources and requests.

In a system with N requests and M resources, let us define an additional function, f, that indicates if a given request can be mapped to a resource or not. Let the requests be $R_1 \dots R_n$, and the resources be $S_1 \dots S_m$. The function $f(R_i, S_j)$ is true if R_i can be mapped to S_j, otherwise it is false. Let us now define the conditions that we shall use to map requests to resources.

Condition 1 Every request is mapped to at most a single resource.

Condition 2 Every resource is mapped to at most a single request.

Condition 3 Request R_i is mapped to resource S_j if $f(R_i, S_j)$ is true.

Let us look at a simple example in the figure below, where we have requests and resources. There is an edge (with a dotted line) between a request and a resource if the request can be mapped to the resource. An edge with a solid line means that the request is mapped to the resource.

Note that the final mapping needs to follow the three conditions that we have enumerated. In this case, we have two requests and two resources. R_1 can be mapped to S_1 only, whereas R_2 can be mapped to S_1 or S_2. If we map R_2 to S_1, then we cannot map R_1 to any other resource. The only resource that R_1 can be mapped to is S_1 and that has already been mapped to R_2. Hence, we shall have a single mapping (see Figure 8.45(b)).

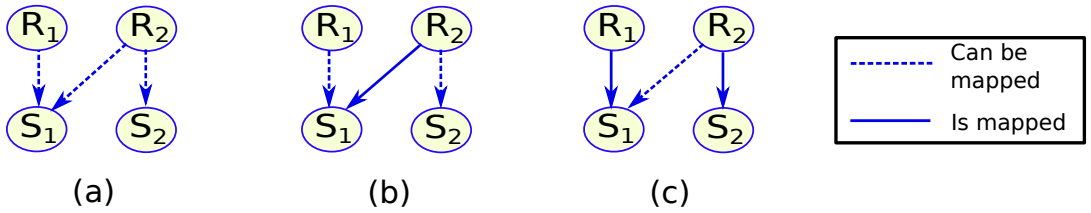


Figure 8.45: Mappings between requests and resources

Now, if we map R_1 to S_1, then S_2 is free. At this stage, the mapping is not *maximal*, which means that it is possible to create more mappings. We thus create another mapping between R_2 and S_2 (see Figure 8.45(c)). In this case, we were able to create two mappings as opposed to the previous case, where we created a single mapping. This is thus a more optimal solution. There is a theoretical name to this problem. It is called the *maximum matching* problem in bipartite graphs. The graph that we have drawn in Figure 8.45(a) is a bipartite graph because we have two classes of nodes – requests and resources – that have no edges between them. We only have edges between a request and a resource.

The aim is to match (or map) the maximum number of requests to resources. This problem is solvable and there are excellent algorithms for finding maximum matches; however, the solutions are slow from the point of view of hardware and are difficult to realize using a simple circuit. Hence, we most of the time try to compute a *maximal matching or mapping*, where it is not possible to create any additional mappings. The solutions shown in Figure 8.45(b) and 8.45(c) are both maximal. However, the solution in Figure 8.45(c) is better because we map more requests to resources.

Let us now discuss a few simple arbiters and allocators.

Round Robin Arbiter

The simplest method of performing arbitration is round robin arbitration. In this case, the resource is distributed among the requesting agents in a round robin fashion. This means that at the outset the resource is given to requesting agent 1, then to agent 2, till we reach agent N. The assumption is that the total number of agents in the system is equal to N. After agent N finishes using the resource, we again come back to agent 1, and continue in the same fashion.

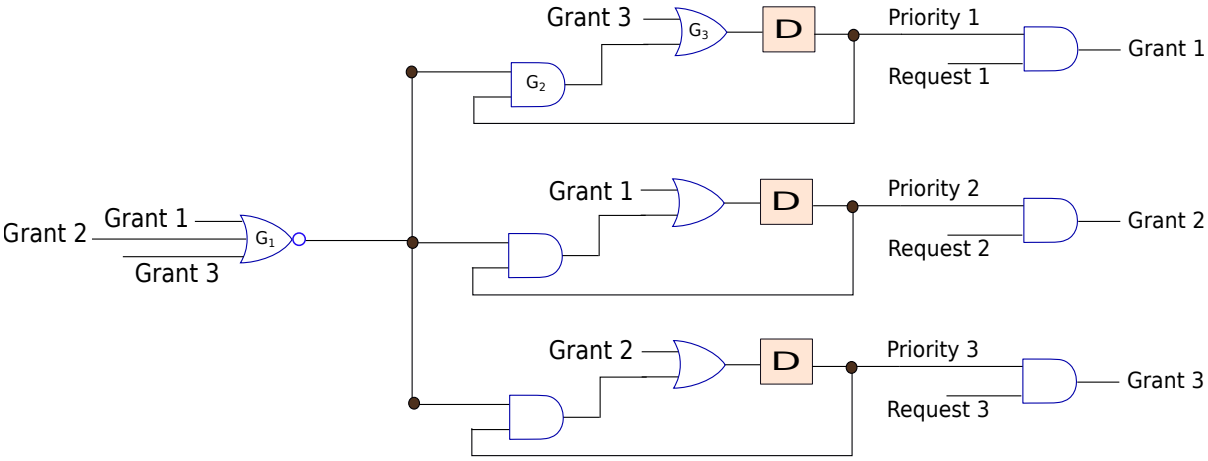


Figure 8.46: A simple round robin arbiter

Figure 8.46 shows a simple round robin arbiter. We assume that there are N request lines and N grant lines. Each request line is connected to a requesting agent. If the agent needs to place a request, it raises the voltage of its request line to a logical 1. On similar lines, there are N grant lines. Whenever a request is granted to the i^{th} agent, the i^{th} grant line is set to 1. The rest of the grant lines are set to 0.

In Figure 8.46, we have three requesting agents. They assert the lines *Request 1*, *Request 2*, and *Request 3* respectively. Each request line is connected to an AND gate, whose other input is a priority line. For example, we have an AND gate that has two inputs: request line 1, and a priority for the first input (*Priority 1*). The output is the grant signal for the first input, *Grant 1*. The grant signal is only asserted when the corresponding request and priority lines are both high. We have similar AND gates for the other two request and priority lines.

Let us now come to the main circuit. Gate G_1 is a NOR gate that computes a logical NOR of all the grant lines. If any of the grant lines is equal to 1, then the output of G_1 is 0. This means that the output of G_2 is also 0, because it is an AND gate. This further implies that the output of G_3 is *Grant 3*. If the value of the grant line for the third input (*Grant 3*) is 1, then this value gets recorded in the D flip-flop. At the beginning of the next cycle, the value of the *Priority 1* line is set to 1. This means that the request for the first agent can be granted. This is according to the round robin policy, which says that after agent 3 gets a chance, it is time for agent 1 to get a chance. Hence, its priority line is set to high. At this stage, the reader needs to convince herself that the round robin strategy is indeed being followed. After agent 1 gets a chance, its grant line is set to 1, then agent 2 gets a chance and so on.

Now, let us look at the uncommon cases. Assume that *Grant 3* is equal to 0. If any one of *Grant 1* or *Grant 2* would have been set, then the output of G_2 would have been 0, and the output of G_3 would also have been 0. This means that *Priority 1* would also be 0. This is the correct behavior. The *Priority 1* line should only be 1 after *Grant 3* has been asserted (as per our round robin policy).

Finally, consider another case where all the grant lines are deasserted (set to 0). In this case, the

output of G_1 is 1. The output of G_2 is the value of *Priority 1*, which is also the output of G_3 (because *Grant 3* is 0). This means that *Priority 1* maintains its previous value. The same holds for the rest of the priority lines: *Priority 2* and *Priority 3*. In other words, if no grant lines are asserted, then the priorities maintain their prior values. However, if there are sufficient requests, then the grant lines are asserted using a round robin policy.

The main drawback of this approach is that we are constrained to using a round robin policy. Even if an agent is not interested in acquiring a resource, it still needs to be a part of the algorithm. As long as it does not assert its grant line, the next agent cannot win the arbitration (be granted a request). In a practical scenario, it is possible that a given agent might not have requests even though its priority line may be set to high. To stop indefinite waits, we can add extra logic that asserts the grant line of an agent whose priority is high, if it does not receive any requests for a given period of time.

Matrix Arbiter

The round robin arbiter works for small systems; however it is not very flexible. If some agents are not interested in the resource, then there is no way of removing them from the protocol. Let us look at the matrix arbiter, which is more flexible in this regard.

In this case, we create an $N \times N$ Boolean matrix, $\mathcal{W}$, where N is the number of requesting agents. The properties of the matrix $\mathcal{W}$ are as follows.

1. $\forall i, \mathcal{W}[i,i] = \phi$. All the diagonal entries have null values.
2. $\forall i, j, \mathcal{W}[i,j] \neq \mathcal{W}[j,i]$ if $i \neq j$.

This means that the diagonal entries have null values. Secondly, $\mathcal{W}[i,j]$ and $\mathcal{W}[j,i]$ have dissimilar values when $i \neq j$. In this matrix, $\mathcal{W}[i,j] = 1$ means that agent i has a priority over agent j. This implies the following:

> If all the entries in the i^{th} row are 1 (other than the diagonal element), then the i^{th} agent has the highest priority. It can be granted the resource.

The basic steps in the arbitration algorithm are as follows.

1. If a given agent is not interested then it sets all the entries in its row to 0, and all the entries in its column to 1 (other than diagonal elements).
2. In every cycle, the request is granted to the agent, which has ones in all the entries in its row other than the diagonal element.
3. Once the i^{th} agent is done servicing its request, it sets all the entries in its row to 0, and sets all the entries in its column to 1 (other than the diagonal elements). This means that it relegates itself to the lowest priority.

Figure 8.47 shows an example. In this case, agent 1 is assigned the resource first because all the non-diagonal entries in its row are 1. Subsequently, it resets all the entries in row 1, and sets all the entries in column 1. Since agent 2 is not interested in acquiring the resource, the next agent that should be granted the resource is agent 3 (all the entries in its row are 1). After agent 3 is done, it follows the same procedure. The next agent that should be granted the resource is agent 4. After agent 4 is done, it sets all the entries in column 4 to 1.

Even though agent 2 was not interested, we end up with a situation where the rest of the agents are done with processing their requests, and all the entries in the second row are equal to 1. At this point, only agent 2 can satisfy its request, and none of the other agents can. We thus see that we are constrained to provide every agent a single chance in a cycle of N requests. This algorithm is more

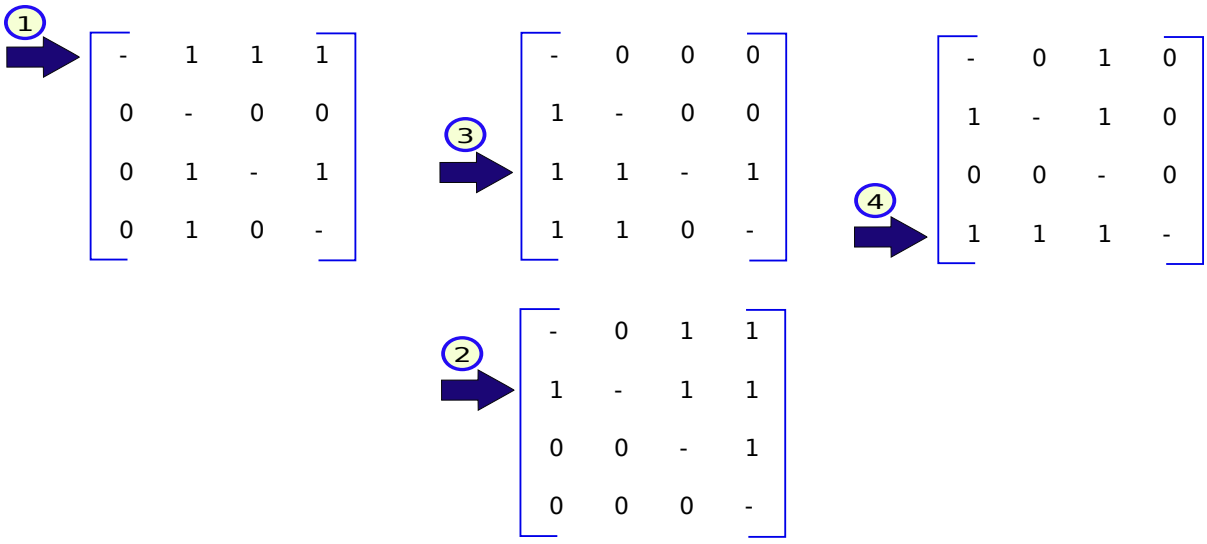


Figure 8.47: Matrix arbiter. The '-' symbol represents a null value.

flexible than round robin allocation in the sense that we do not have to stick to a pre-defined order. However, we still need to take care of the situation that has been created with agent 2.

An intuitive way of solving this problem is as follows. When agent 2 can acquire the resource, yet it does not have a valid request, it simply follows the protocol of relegating itself to the lowest priority (set the row to zeros, and column to ones).

Separable Allocator

Both the round robin and matrix arbiters have limited flexibility. Furthermore, they cannot be used to allocate N agents to M resources. For such scenarios, we need a more general allocator. Let us first look at a naive design, where we have 3 agents, and we have 3 resources. Each agent produces three inputs, which are the request lines for the three resources. We set R_{ij} equal to 1 if the i^{th} agent wishes to acquire the j^{th} resource. We can then create a separable allocator as shown in Figure 8.48.

The separable allocator has two columns of arbiters. An arbiter in the first column chooses one request from each agent. Recall that we cannot allocate two requests issued by the same agent at the same time. We need to choose one of the agent's requests. This is done by the 3×3 arbiters in the first column. Since we have three agents, we have three arbiters. Consider the first arbiter in the first column. It has three inputs: R_{11}, R_{12}, and R_{13}. There are three grant lines as outputs – one corresponding to each request. Let us number them G_{11}, G_{12}, and G_{13} respectively. At most one of them can be asserted (set to 1). If none of the request lines are set, then all the grant lines (G_{11}, G_{12}, and G_{13}) need to be set to 0.

The outputs of the first three arbiters are inputs to the arbiters in the second column. Consider the first arbiter in the second column. Its inputs are the grant signals generated by arbiters in the first column: G_{11}, G_{21}, and G_{31}. They correspond to all the requests for the first resource. Only one of the requests can be chosen (or granted). We follow the same logic here as arbiters in the first column, and choose one of the requests. The outputs of the arbiters in the second column are the final grant lines, which are routed to the agents.

The separable allocator can lead to sub-optimal allocations. Assume that we have a situation where each agent is interested in all the resources. This means that in any maximal matching, we can assign each agent to a distinct resource. However, in a separable allocator it is possible that in the first column,

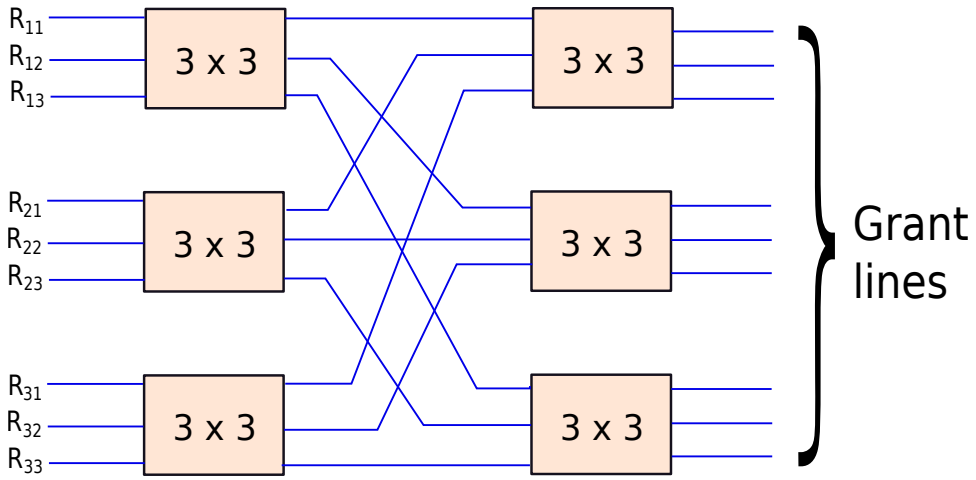


Figure 8.48: A separable allocator

we only choose requests for let's say resource 1. In this case, we can only map one agent to one resource (resource 1). This is clearly sub-optimal. Instead of three mappings, we are creating just one. We need an arbiter with a more global view.

Wavefront Allocator

Let us develop an algorithm where we can simultaneously allocate resources to the requesting agents. This will not create situations where we have different rounds with sub-optimal choices being made.

Consider a matrix, $\mathcal{W}$, where the rows represent the agents and the columns represent the resources. Assume for the sake of simplicity that the number of agents N is equal to the number of resources M ($M = N$). Let us give two tokens to each diagonal element $\mathcal{W}[i, i]$. One of them is called a *row token*, and the other is called a *column token*. The row token propagates along a row, and the column token propagates along a column.

In the matrix $\mathcal{W}$ (shown pictorially in Figure 8.49), we start out with giving one row token and one column token to each of the diagonal elements. If agent i is interested in acquiring resource j then we mark the cell $\mathcal{W}[i, j]$. It is shown with a shaded color in Figure 8.49.

The matrix is connected like a torus, where neighboring cells in a row and column are connected, and there is a connection between the first cell of a row and the last cell of the row (likewise in a column).

The algorithm consists of multiple rounds. In each round, each cell executes the following steps.

1. If a given cell in the matrix has a row token and a column token, and is interested in acquiring a resource, then it is allotted that resource. Cell $\mathcal{W}[i, j]$ is interested in acquiring a resource if agent i is interested in getting access to resource j. This cell *consumes* both the tokens, and removes them from the system.

2. If a cell is not interested in a resource, and it has a token, then it sends the token to a neighboring cell such that it can be processed in the next round. If it has a row token it sends it to the cell on the left, and if it has a column token then it sends it to the cell that is just below it.

3. If a token reaches an edge of the matrix, it traverses the long edges to wrap around the matrix and start from the other end of the matrix (same row or same column) in the next round.

Consider the same matrix $\mathcal{W}$ as shown in Figure 8.49, where all the cells are not interested in acquiring any resource. Let us understand the movement of tokens across rounds (refer to Figure 8.50).

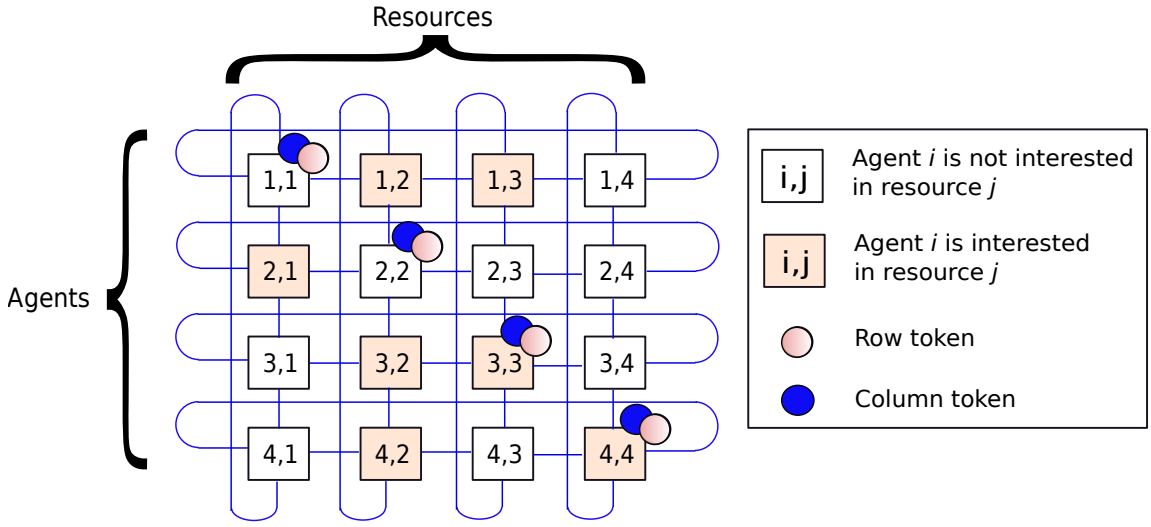


Figure 8.49: The matrix $\mathcal{W}$ with agents, resources, and connections between neighboring cells

The initial state is round 1, where all the tokens are given to the diagonal elements. Before the next round, all the row tokens move one step to the left. If a token is at the edge of the matrix, it wraps around the matrix. For example, the row token on cell (1,1) goes to cell (1,4). The row tokens in the other cells move one step to the left. Similarly, the column tokens move one step down, and the token in the bottom-most row wraps around. For example, the column token in cell (2,2) moves to (3,2), and the column token in cell (4,4) moves to (1,4) (wraps around).

Observe that in the second round, we have two kinds of cells: cells with two tokens (one row and one column) or no tokens. In the subsequent rounds (round 3 and round 4), the same property holds. Even after the tokens wrap around, we never have the case that a cell has a single token. We would like to advise the reader to manually verify that this property holds across the four rounds, and also if we consider a bigger matrix.

Let us now consider all the cells in a round that have both the tokens (shown using dotted lines in Figure 8.50). It is like a wavefront that is propagating towards the bottom-left. Parts of the wavefront wrap around the edges. The wavefront propagates one step diagonally in each round. Let us look at some features of this wavefront. In the example shown in Figure 8.50, the size of the wavefront is exactly N (in an $N \times N$ matrix). It never has two cells in the same row or in the same column. Any cell on the wavefront, which has two tokens, can consume both the tokens if it is interested in the resource. Henceforth, no other cell in that row or in that column can acquire any resource in the future. In other words, if an agent is allocated a resource, it cannot be allocated any other resource, and likewise if a resource is allocated to an agent, then it cannot be allocated to any other agent. This is precisely the property of allocation that we needed to ensure.

Let us define a cell to be *free* if no other cell in the same row or column has been allocated a request. Our observation is that all *free* cells either have two tokens or do not have a token.

Let us now see if this observation holds true for a case where a few of the agents are interested in acquiring resources. This situation is shown in Figure 8.51, where cells (2,2) and (3,1) are interested in acquiring a resource. In the first round, cell (2,2) consumes both the tokens, thus all the cells in the second column and second row are not free after the first round. For example, the cells (2,1) and (3,2) are not free in the second round. Also observe that they have a single token each.

The remaining tokens propagate as per the rules that we have defined. Again in round 3, the cell (3,1) consumes the two tokens that it gets. Finally, we have 4 tokens left in round 4.

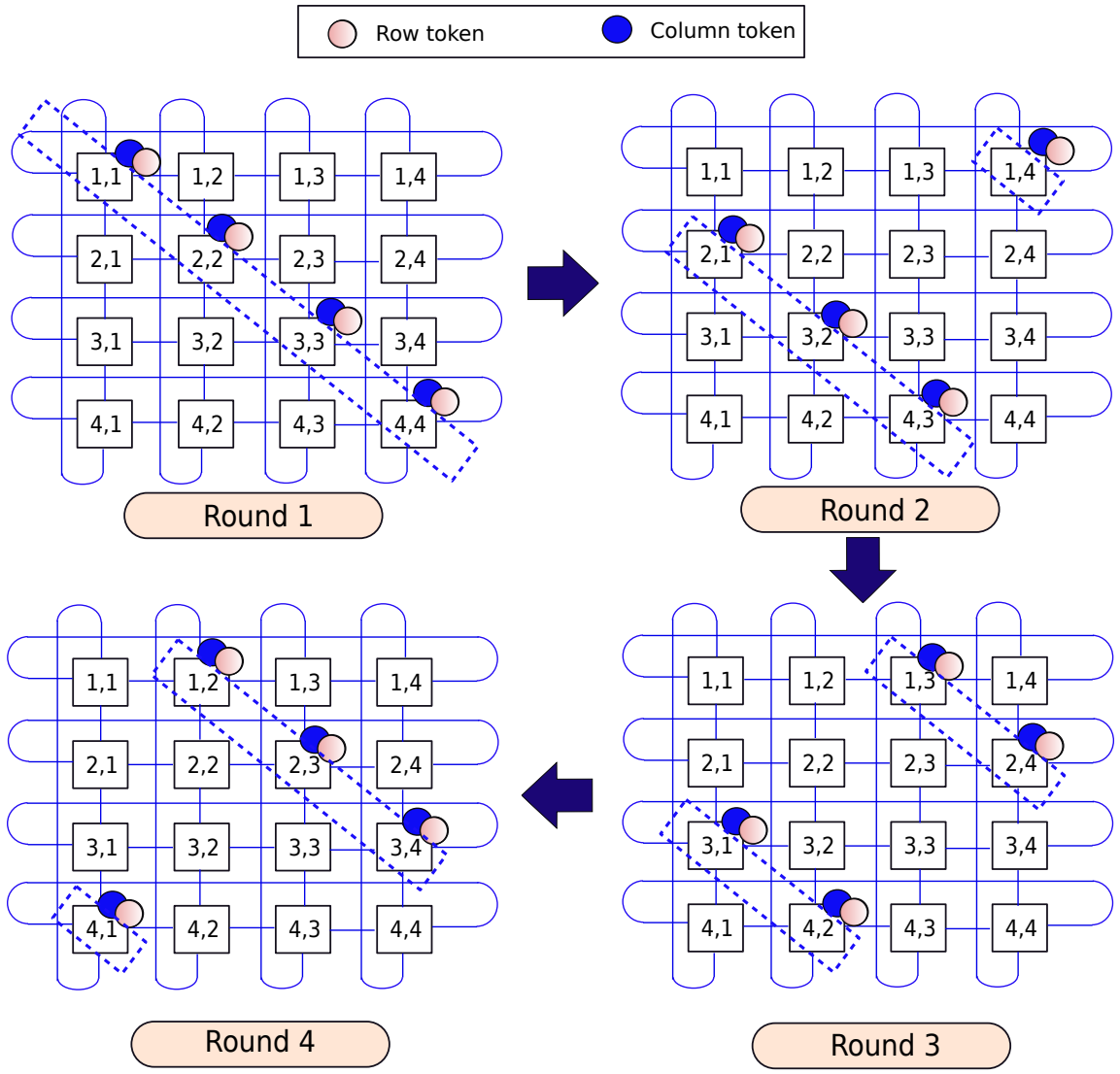


Figure 8.50: The movement of tokens in different rounds

Let us look at the cells that are free in each round. We observe that our earlier observation still holds. All the free cells either have two tokens or do not have any token. We never have a situation where a free cell has a single token. This is not possible because for a cell to have a single token, one of the tokens in its row or column must have been consumed. This means that the cell is not free anymore.

From both these examples (Figures 8.50 and 8.51), we can conclude that the mapping that is produced is correct – no agent is allocated more than one resource, and no resource is allocated to more than one agent. Is it maximal?

If the mapping is not maximal, then it means that there is an agent-resource pair that can be still mapped after all the rounds are over. If it can be mapped, it must be the case that the corresponding cell was always free. At some point, the wavefront must have crossed it, and at that point the cell should have consumed both the tokens. Since this has not happened, there is a contradiction, and it is not possible to have a free cell after all the rounds are over. Thus, we have proven that a wavefront allocator

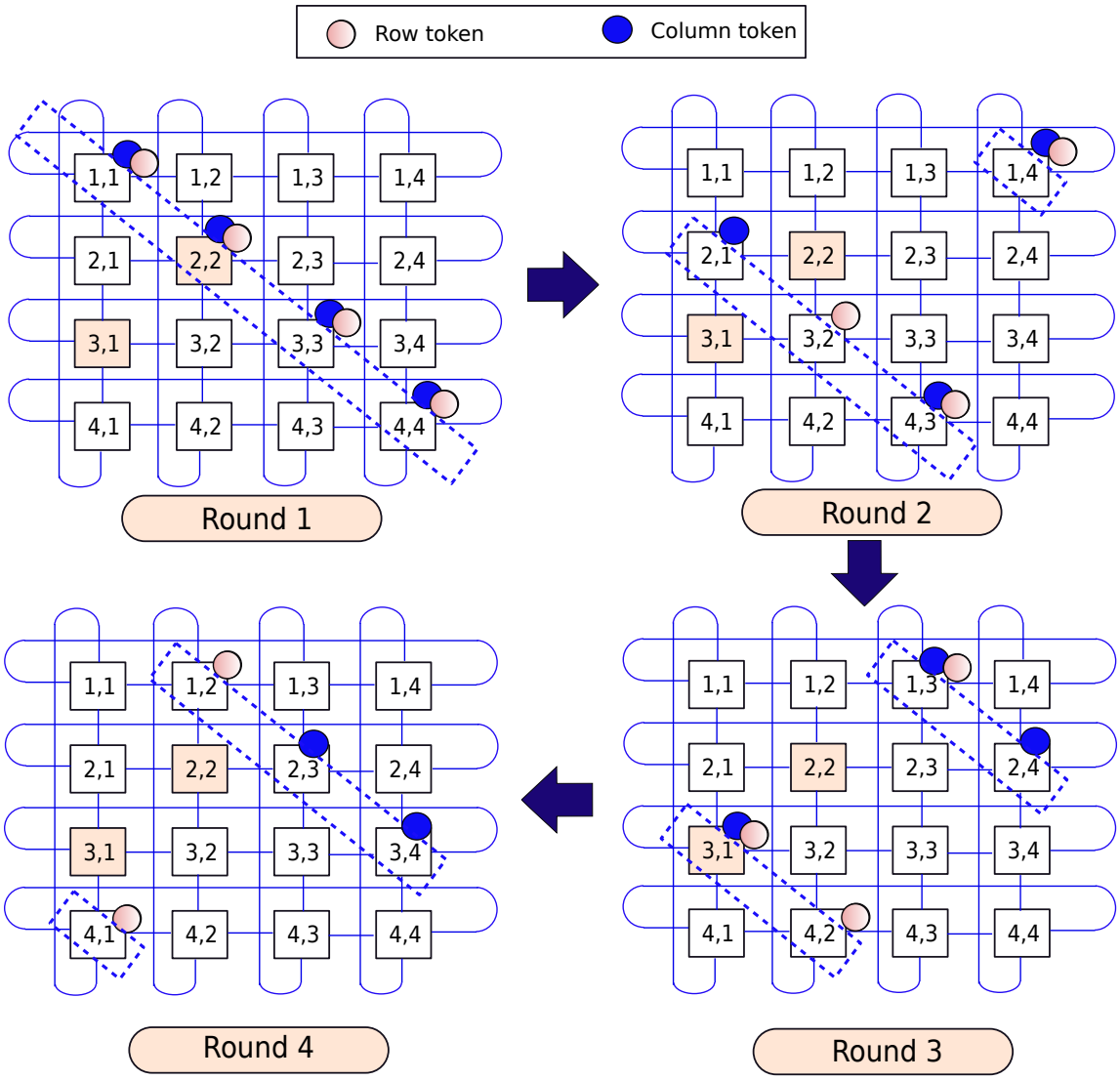


Figure 8.51: The movement of tokens in different rounds when two cells are interested in acquiring a resource

produces a maximal mapping. It is not necessarily optimal though.

Summary

Figure 8.52 shows the full design of the router. Note the credit computation unit that processes the credits received from neighboring routers and forwards credits when buffers get freed.

8.4.7 The Router's Pipeline

We have described the five stages of processing a packet at a router: BW (buffer write), RC (route computation), VA (VC allocation), SA (switch allocation), and ST (switch traversal) (see Table 8.4). Note that the head flit needs to go through these 5 stages. However, for the rest of the flits (body and

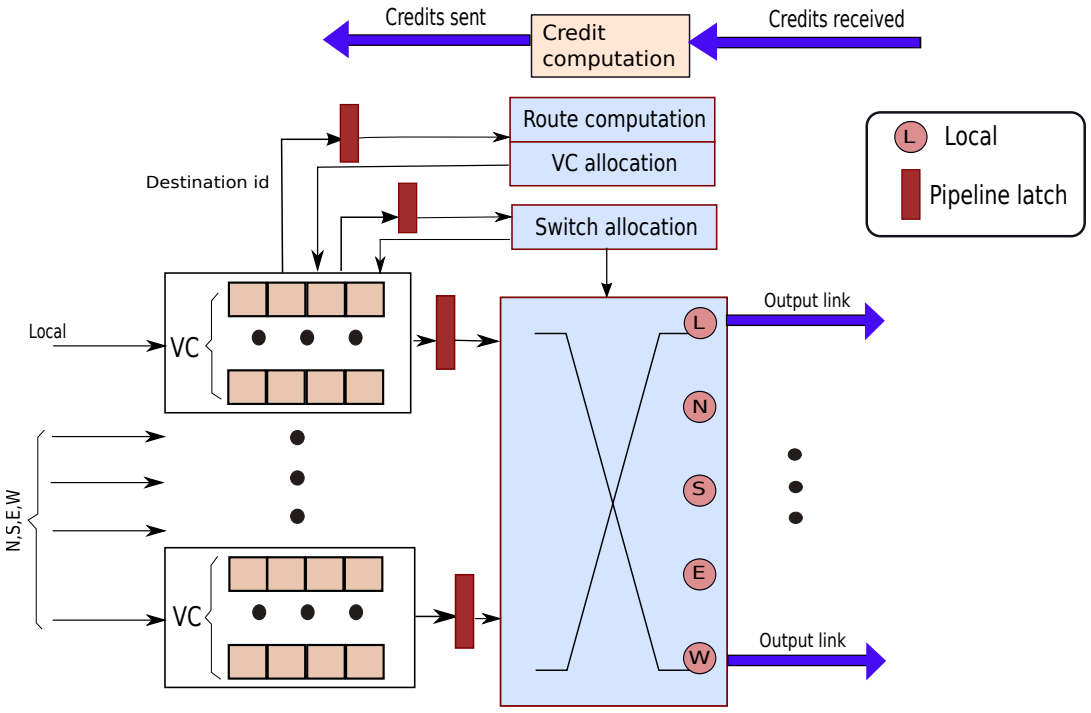


Figure 8.52: Design of the router

tail), they need not go through all of these stages because they will follow the same route as the head flit and also use the same virtual channel. Hence, for these flits we can skip the RC and VA stages. For them the router's pipeline has only three stages: BW, SA, and ST. However, the critical path is still determined by the head flit because the body and tail flits are not allowed to overtake the head flit.

Name	Abbreviation
Buffer Write	BW
Route Computation	RC
VC Allocation	VA
Switch Allocation	SA
Switch Traversal	ST

Table 8.4: Stages in a router's pipeline

Similar to processor pipelines, we can make a simplistic assumption that they take roughly the same amount of time. This can also be ensured by sizing the buffers and structures appropriately. As we assume in a standard in-order pipeline, each stage takes 1 cycle to traverse. Thus, to traverse the entire pipeline in a router, it will take 5 cycles.

This pipeline can be thought of as a simple 5-stage in-order pipeline where flits enter, and leave via the outgoing channels in the same order in which they arrived. It is possible that a flit F in packet P may stall because of the lack of availability of resources such as VCs or switch ports. In this case, there will be a stall in the router's pipeline, and we will not be able to transmit the flit, or any subsequent flit in packet P.

This situation is depicted in the space-time diagram in Figure 8.53. Unlike a processor, where one

stall stops all the instructions in the earlier stages of the pipeline from making progress, in this case flits in other VCs can proceed. They are not blocked. The only flits that are blocked are flits in the current packet. Here also, there is an important difference. In a processor, two instructions can never be in the same pipeline stage; however in this case, two flits of the same packet can be in the same router pipeline stage. Let us elaborate. It is possible for flit $F3$ to be written into a buffer when flit $F2$ is in the buffer write stage waiting for flit $F1$ to make progress. This means that two flits of the same packet can be in the buffer write stage (either being written into a buffer or waiting to progress to the next stage). We have however not shown this in our space-time diagram in Figure 8.53 for the sake of readability and simplicity. In the figure, we assume that flit $F3$ enters the BW stage after $F2$ leaves it.

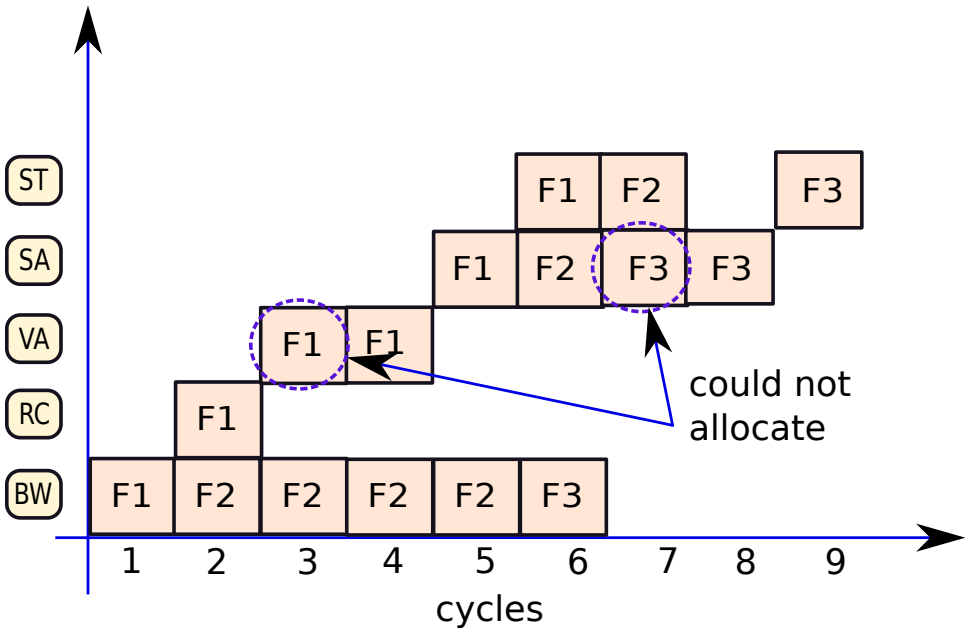


Figure 8.53: Space-time diagram of a 3-flit packet being sent through a router's 5-stage pipeline

Let us now point out an irony. It takes roughly 1 cycle to traverse a link, and it takes 5 cycles to traverse a router's pipeline! This means that the delay in the routers is 5 times the time it takes to propagate through all the links. This further means that the latency of routers is the primary determinant of the on-chip transmission delay. Let us consider some representative numbers. Consider a system with 32 cores and 32 cache banks. We can arrange them as a 8×8 chessboard. The worst case delay between two points in a 2D mesh can be calculated as follows. Assume that we always take the shortest path. In this case, the worst case delay is incurred when we send a message from one corner to the diagonally opposite corner. We need to traverse through 14 hops and 15 routers. The total time required is $14 \times 1 + 15 \times 5$ (=89) cycles. Now, if we want a reply from the cache bank at the diagonally opposite corner, the total transmission delay for the request and the reply is equal to two times 89 cycles, which is 178 cycles. This is significant, and is of the order of the main memory access time, which is typically between 200 and 300 cycles.

Hence, there is an urgent need to reduce the time of propagation within a router such that the total latency can also commensurately decrease. It is necessary to speed up the NoC. We cannot reduce the speed of the links themselves because we are constrained by basic physics. The only option that is available to us is to make the routers faster. We need to somehow compress the 5-stage router pipeline to a shorter pipeline. Let us look at some pipeline optimizations to speed up the NoC by reducing the number of stages.

Lookahead Routing

The first stage, buffer write, is required. Without this stage the packet will get lost. Let us thus look at the next stage, Route Computation (RC). We have assumed that this stage takes 1 cycle. Can we move this stage out of the critical path? The answer is No, because the subsequent stage, VC allocation, needs this information. Unless we know the route (the outgoing physical channel), we will not be able to allocate a VC. It does indeed look like that we need to have the RC stage as the second stage, and this needs to be on the critical path. However, we can do something ingenious.

Let us compute the route one hop in advance. Consider Figure 8.54, where we show the route from nodes 1 to 6 (nodes along the route are numbered 1, 2, 3, 4, 5, and 6). Before sending the head flit to the starting node (node 1), we compute the next hop. Thus, the portion of the route, $1 \rightarrow 2$, is known. Then the head flit enters the pipeline of the router. Since the route is known, the process of route computation is off the critical path. We can assign virtual channels, and switch ports in the router's pipeline, and in parallel compute the route that should be taken after exiting node 2. We compute this to be node 3. After leaving node 1, when the head flit reaches node 2, the subsequent route is known in advance. We know that the next hop is $2 \rightarrow 3$; hence, we can start assigning virtual channels. Similar to the process followed in node 1, we can compute the route to be followed after leaving node 4, which is $3 \rightarrow 4$.

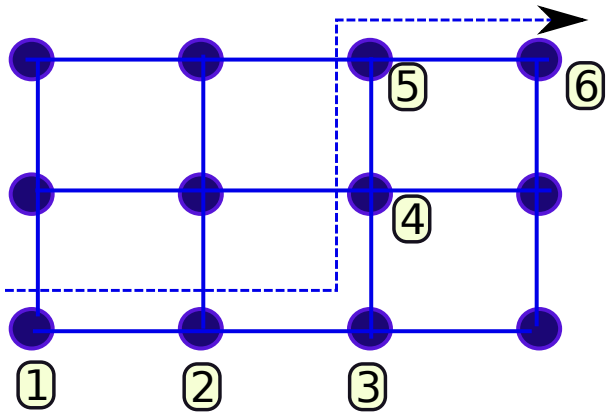


Figure 8.54: Route from nodes 1 to 6

Our 5-stage pipeline thus becomes a 4-stage pipeline because we have removed the RC stage off the critical path. It executes in parallel, and it simply needs to compute its result before the head flit leaves the router. The resultant pipeline is shown in Figure 8.55 for both head flits and body/tail flits.

Bypassing

Assume a router is very lightly loaded, which means that very little traffic flows through the router. In this case, we need not increase the overhead of the process of routing by first writing the flit to a buffer, and then looking for free ports in the switch. If we make an optimistic assumption that the switch is free, and we can get access to its ports without contention, then we can further shorten the length of the router pipeline. This is called *bypassing* [5]. The pipeline is shown in Figure 8.56. We skip the buffer write (BW), and switch allocation (SA) stages.

Whenever we decide to adopt this technique, there are several things we must take into consideration. First, it is possible that two flits are trying to enter the same ingress (input) port of the switch (assuming that we do not have one port for each VC). In this situation one of the flits needs to wait. We can write

[5]Note that bypassing is a router optimization. It is not the same as forwarding and bypassing in pipelines. These are different terms.

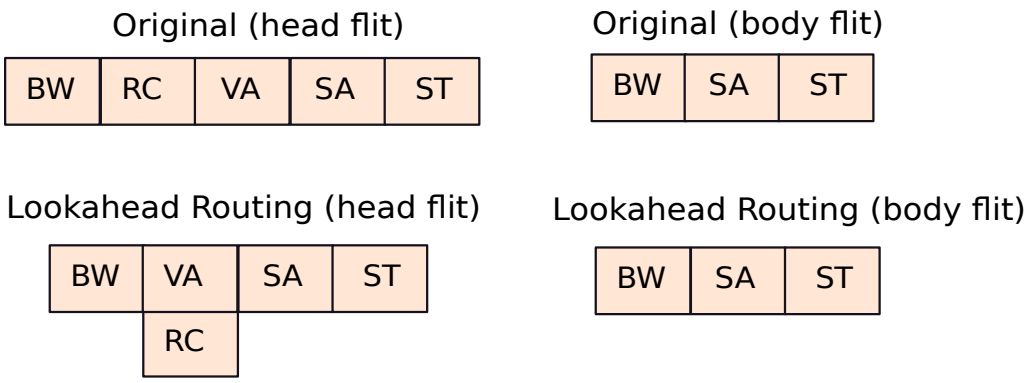


Figure 8.55: Pipeline stages in lookahead routing

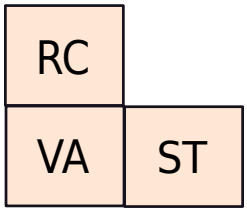


Figure 8.56: 2-stage pipeline in the case of bypassing (for the head flit)

that flit to a buffer, and make it traverse through the regular pipeline. It is also possible that we might have contention at the output links. For example, two flits might be trying to leave the router via the westward outgoing link. In this case, we have a conflict, and one of the flits needs to wait, and get buffered.

As we can observe from Figure 8.56, in the best case, the router pipeline gets compressed to just two stages, which is the best that can be done with our current model. We need at least one cycle for computing the route even with lookahead routing; we allocate a VC in parallel. Subsequently, we need one more cycle for traversing the switch.

Speculative VC Allocation

Even though bypassing can effectively reduce a 5-stage pipeline to a 2-stage pipeline, it cannot be done always. Bypassing can only be done when the amount of traffic in the network is low. This is non-deterministic in nature, and its applicability is limited in scenarios with high NoC traffic. Such schemes are called speculative schemes, because we are making a guess – speculating – that a switch port, and the outgoing link is free.

Let us instead speculatively perform VC allocation (VA stage) by assuming that we shall find one free VC. This will help us remove this stage from the critical path. VC allocation can be done simultaneously with switch allocation and route computation as shown in Figure 8.57 (combined with lookahead routing). By the time we have allocated a VC, the head flit will be in the switch traversal stage. Just before it is sent on the outgoing link, we can add the VC information and send it on the link. This represents the best case scenario where we are able to allocate a VC. However, this need not be the case always, particularly if the load on the network is high. In this case, it is possible that we might run out of VCs. As a result, speculative VC allocation will not succeed. Similar to the case of bypassing, we need to make the head flit follow the rest of the stages of the regular pipeline without resorting to any form of

speculation.

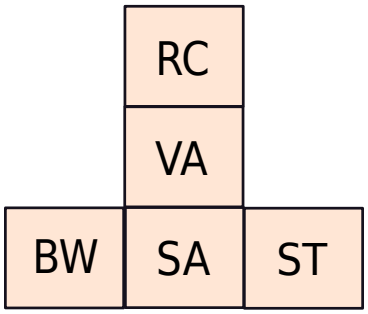


Figure 8.57: Speculative VC allocation

The advantage of speculative VC allocation is that it reduces a 5-stage pipeline to a 3-stage pipeline as shown in Figure 8.57. This will not be possible all the time; however, the likelihood of its success is in general much more than the success with bypassing. This is because in this case, we just need a free VC, whereas in the case of bypassing we need a free input port in the switch and a free outgoing link as well. Of course, the final answer depends on the number of VCs per physical channel, the size of the switch, and the nature of the traffic. This question is best answered with architectural simulation studies.

Late VC Selection

Let us introduce another approach, where we remove the VA stage altogether. Once switch ports are allocated, let us send the head flit through the switch. In the time being, let us try to allocate a VC using a very simple allocator. Let us maintain a queue of free VCs for each outgoing link. Whenever we need to allocate a VC for the head flit, we simply select the head of the queue. The VC information is appended to the head flit after it traverses the switch; the head flit along with the VC is then sent to the next router on the path. Subsequent flits in the packet use the same VC. The reason that we can afford a simple allocator such as a queue is because only one flit traverses a link at any time, and thus there is no need for arbitration across multiple requests.

Let us now look at the finer points. If a head flit is allocated a switch port in the SA stage, then we are making the assumption that it will find a free VC. If it does not find a free VC, then we cannot send the flit. Note that in this case we are speculating, which means that we send the head flit through the switch in the hope of finding a free VC for the outgoing link similar to speculative VC allocation. If the speculation fails, then we have to cancel the process and make the head flit go through the regular pipeline.

We can opt for a slightly more conservative option. We check if the queue has entries before entering the switch allocation stage. If there are no free entries, then the head flit does not enter the SA stage in the first place, we wait to be allocated a VC.

The next question that we need to answer is when do we return a VC to the queue. This can be done when the VC is deallocated – when the tail flit of the packet leaves the router.

8.5 Non-Uniform Cache Architectures

After the complexity of single cores reached saturation, the additional transistors were used to increase the number of cores and the sizes of the caches. The sad part with increasing the sizes of the caches is that they become very large to manage. Hence, as we have seen in Section 7.3, to keep the delays in check, we divide a large cache into banks, and we further divide banks into subarrays. There is a

trade-off between the time that a circuit takes to locate and route the signal to the right subarray and the access time of the subarray.

Such conventional designs worked very well when the size of the on-chip cache was of the order of hundreds of kilobytes. However, such designs fail to scale when the cache size exceeds a few megabytes. The main reason for such caches failing to scale is the delay incurred in routing the request to the right subarray. As we have argued, it is thus necessary to create an on-chip network (NoC).

An NoC allows a bank based design, which allows parallel accesses to different banks. Even though this method increases the throughput, let us understand the effect on latency. Consider a chessboard based design with 32 cores and 32 cache banks (8×8) with a mesh based NoC. To move a flit from one corner to the diagonally opposite corner we need to traverse through 15 routers and 14 links. Traversing the links is fast (typically less than a cycle); however the routers are slow (3-5 cycles) in spite of the numerous optimizations that are used to speed up router pipelines.

Moreover, the delays of different banks are not the same. Proximate (nearby) banks take a lesser amount of time to access as compared to banks that are far away. NoC delays can typically vary from 5 to 50 cycles to access different banks. Whenever we have variance to such an extent, we are presented with a huge opportunity for proposing optimizations to cache access protocols. We should strive to somehow place data closer to the requesting cores.

Consequently, let us look at a set of proposals called non-uniform cache architectures [Kim et al., 2003]; they are also referred to as *NUCA* architectures. These schemes propose to manage large L2/L3 caches such that the mean access time is reduced in caches with elaborate NoCs and variable bank access times.

We shall introduce two main types of architectures in this space.

S-NUCA Cache blocks are allotted to banks statically.

D-NUCA Cache blocks are allotted to banks dynamically. Blocks can migrate between banks at run time.

8.5.1 Static NUCA(S-NUCA)

The basic idea is very simple. If we have K cache banks, we use $log_2(K)$ bits from a block's address to address the bank. Let us consider a representative example. Assume a 32-bit addressing scheme, a 64-byte block size, and 256 sets per cache bank. We will thus have to dedicate 6 bits to address a byte within the block. The block address is thus 26 bits. In addition, we need to use 8 bits to address the set. The remaining 18 bits are the tag. Next, assume that we have 32 banks. This means that we need 5 bits to decide the bank. Which 5 bits should these be?

Let us answer this question by considering multiple designs. Consider the two alternatives shown in Figure 8.58. We show two schemes: S_{LSB} and S_{no-LSB}. In the S_{LSB} scheme, we choose the 5 LSB bits (bits 1 to 5) of the block address to uniquely address each bank. Bits 6-13 are used to address the set. In the S_{no-LSB} scheme, we use the 8 LSB bits (bits 1 to 8) of the block to address the set, and then we use bits 9-13 to address the bank. These are the two most popular schemes.

To understand which design is better, let us appreciate a basic fact. We expect the randomness in the least significant bits to be higher than the more significant bits in any address. This is because of spatial locality. Most of the time we will be accessing different cache blocks in the same region of memory. Thus, the lower order bits will vary more. Since the memory region remains the same, the higher order bits that determine it will remain the same. For example, if most of the accesses are within the same page then all the bits that determine the page id shall remain the same.

Let us now use this insight to answer our question. In the S_{LSB} scheme, we shall have more randomness (more uniformity) in choosing the bank; however, the access pattern for sets within a bank will not be that uniform. In the S_{no-LSB} scheme, we shall have the reverse effect. The accesses to sets within a bank will be more uniform, but bank accesses will be less uniform. What do we want to optimize?

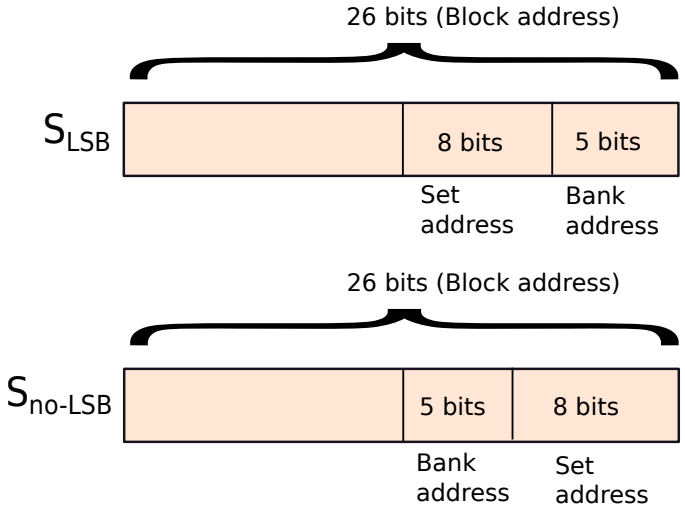


Figure 8.58: Two methods to address a bank and a set in the S-NUCA protocol

Essentially, there is a trade-off between bank conflicts and set conflicts. The answer is dependent on the traffic pattern and requires simulation studies.

8.5.2 Dynamic NUCA(D-NUCA)

The main disadvantage of an S-NUCA protocol is that we are not taking the usage of blocks into account. We should ideally place a frequently used block close to the requesting core. This is not happening. Hence, let us propose the D-NUCA protocol that places blocks close to the requesting cores and migrates blocks based on the access pattern.

The key idea is to divide the set of banks into columns as shown in Figure 8.59. We call each such column a *bank set*. A block can be present within any bank of the bank set. There is no replication though.

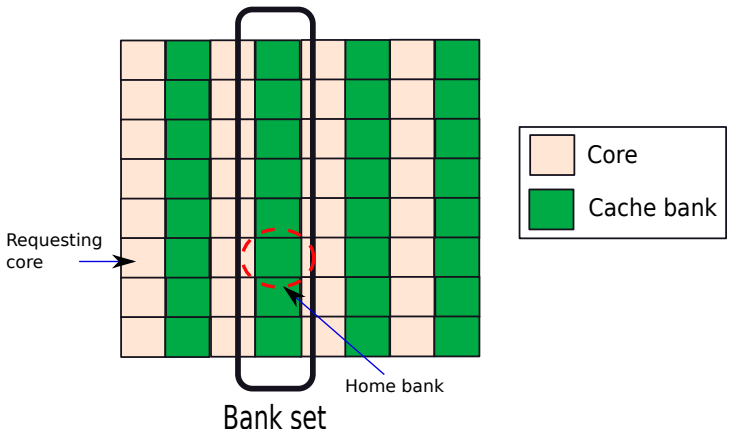


Figure 8.59: Bank sets in D-NUCA. The cores and cache banks are organized in columns.

In Figure 8.59 we have 4 bank sets. We can use 2 bits from the block address to determine the bank set. This is where we need to do something to ensure that we are able to achieve our objective of

placing data close to the requesting cores. Let us designate one of the banks within the bank set as the *home bank*. There is no hard and fast rule regarding how we designate the home bank. However, most research proposals in this area designate the bank in the bank set that is the closest to the requesting core (shortest routing path) as the home bank [Arora et al., 2015]. This is shown in Figure 8.59. The requesting core sends the memory request to the home bank first; it has a separate home bank in each bank set.

Search Policies

Once the request reaches the home bank, we need to search for the block. We first search within the home bank. If there is a hit, then we send the value back to the requesting core. If there is a miss, then we need to search the rest of the banks within the bank set. In this case each bank set is a column – a linear sequence of banks. We need to search the rest of the banks according to a given search policy. As described by Arora et al. [Arora et al., 2015] there are three types of commonly used search policies as shown in Figure 8.60. Note that in this scheme, we never replicate a block across banks; every block has a single location. The search policies are as follows.

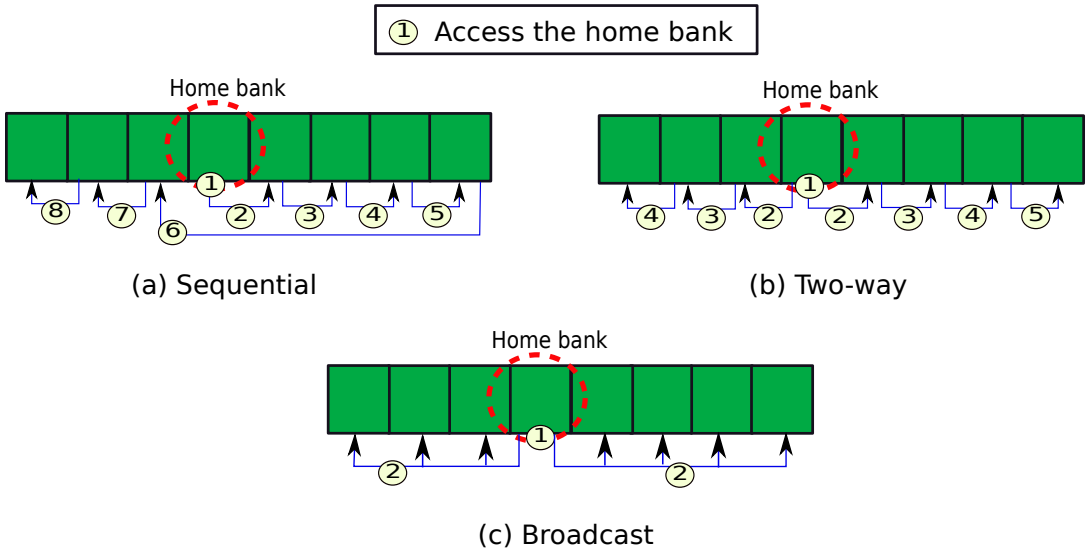


Figure 8.60: Different search policies within a bank set

Sequential We search all the banks in one direction and go till the end. If we do not find the block, then we start searching in the other direction till we reach the end.

Two-Way In this scheme, we send two parallel messages in both the directions. If the block is present in the bank set, then it will be found in only one bank. As compared to the *Sequential* scheme, we are expected to get a response sooner.

Broadcast In this scheme, we broadcast a message to all the banks. On an average, we have a lot of additional bank accesses. However, this is also the fastest scheme if we do not take the contention within banks and the NoC into account. We do not have to waste time searching banks that do not contain the block.

As we can see in Figure 8.60, there is a trade-off between the number of messages we send, the number of banks we search, and the overall latency. Note that *Broadcast* is not necessarily a better scheme. It

increases the rate of bank utilization significantly, and this increases the amount of contention. In a highly loaded system this can decrease the overall performance as well. It is thus necessary to choose the search policy wisely.

Finally, after we locate the block, we send it back to the requesting core. If we don't find it, we signal a miss and a request is sent to the lower level of the memory hierarchy.

Migration

The real magic of a dynamic NUCA cache lies in migration. Upon a hit, we try to bring the block closer to the requesting core. Note that the home bank of the block (from the point of view of the requesting core) is the closest bank in the bank set. If we have a hit in the home bank, then nothing needs to be done because the block resides in the closest possible bank.

Otherwise, we migrate the block towards the home bank. Figure 8.61 shows an example. Initially, we have a hit in bank A, which is not the home bank. We move the block one hop towards the home bank. This means that the next time the same core requests for the same block, the request has to travel one hop less. Ultimately, the block will migrate to the home bank, and remain there assuming that there are no conflicting requests from other cores with different home banks.

By migrating blocks towards the home bank, we are in effect reducing the cache access latency for that block. We are also reducing the number of bank accesses in the *Sequential* and *Two-way* search policies.

When there is an eviction from a bank, we have two choices. Either we write the block to the lower level, or we write it to a bank that is further away from the home bank. In the latter scheme, a block is finally evicted from the cache and written to the lower level when it is evicted from the last bank (rim of the chip) in the bank set. The negative aspect of this scheme is that if a block is not going to be used anymore, it lingers on in the cache for a much longer time. However, on the flip side, the positive aspect is that if the block will be used again, it can be found in a bank that is farther away in the bank set. Ultimately, the choice of the scheme depends on the nature of the workload.

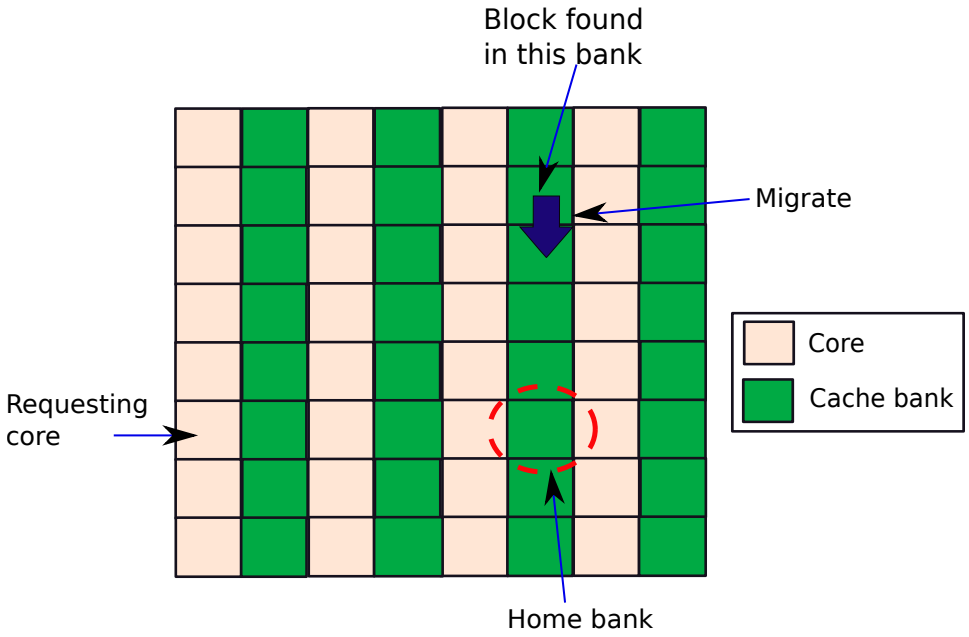


Figure 8.61: Block migration in a D-NUCA cache

This feature helps us reduce the effects of NoC delay in a big way. Even if the chip has a very large NoC, we always try to bring the frequently accessed cache blocks to banks that are the closest to the respective requesting cores subject to the constraint that they remain within their bank sets. This can decrease the latency of cache accesses significantly, and thus is the method of choice in most designs that use NUCA schemes.

Afterthoughts

We can think of a bank set as one large virtual cache bank that all the cores can use. One advantage of having such banks set as opposed to the S-NUCA design is that such bank sets can absorb non-uniformities in bank accesses very effectively. In S-NUCA if there is high contention in one bank, then this will lead to large memory latencies. However, in the case of D-NUCA, blocks will just get distributed in the bank set as per our migration and eviction policies. This will ensure that as a whole, D-NUCA performs better than S-NUCA.

The other advantage of a bank set is that it allows migration of blocks towards proximate banks. This helps us nullify the effect of NoC delays significantly because we reduce the access time of blocks that are the most frequently used.

8.5.3 Advanced Schemes

Till now, we have been making the cardinal assumption that blocks are not replicated across banks in the bank set. Otherwise, there will be a need to keep all of these replicas synchronized. This means that whenever we write to one replica of the block, we need to write to the other replicas as well. This increases the complexity, and in addition, we need a lot of additional state to keep track of all the replicas.

However, this is not a very bad idea as long as we can confine this technique to cache blocks that are read-only. Such cache blocks are of two types: instruction blocks and read-only data blocks. In most programs we never modify the cache blocks that contain instructions while the program is running. Hence, as far as we are concerned, they are read-only blocks. In addition, cache blocks containing data can also be read-only, particularly when they contain constants. All of these read-only blocks can be replicated across banks in the bank set, and thus we can further reduce the access latency. However, detecting read-only blocks is difficult in hardware. Even if we can establish that a given block has not been written to in the last N cycles (N being a very large number), it does not mean that it will not be modified in the future. The only way to do this is by using compiler analyses at the time of program compilation. The compiler can place all the constants in a virtual memory page, and mark it as read-only. We can have an additional bit in the TLB and the page table that marks the page as read-only. Once the hardware is aware of this, it can replicate blocks in the read-only page across cache banks.

For blocks that may be modified, it is not possible to do this unless we invest in a lot of hardware to synchronize the different replicas. We shall devote Chapter 9 to precisely study this problem. We shall conclude that it is possible to do this for smaller caches such as the L1 cache. Maintaining synchronization across replicas is very difficult at the L2 and L3 levels.

Optimizations for Private Data

Assume that a given core is running a thread. Every thread has some private data, which is not accessed by any other thread. This includes the contents of the stack, and the contents of memory areas predefined as thread local storage (TLS) areas. The compiler is aware of the memory regions that are allocated to such areas; hence, it can add instructions to define these memory regions as *private*. In addition, in many architectures, the hardware is also aware of the addresses of these regions. It, too, can set the *private* bit in the corresponding entries of the TLBs and page tables.

Once we access the TLB, we instantly become aware if the block is in a *private* region or not. If it is, then some more optimizations are possible. Since this data will not be required by any other core,

we should not allow the blocks to migrate to a cache bank that is far away. There are several possible strategies.

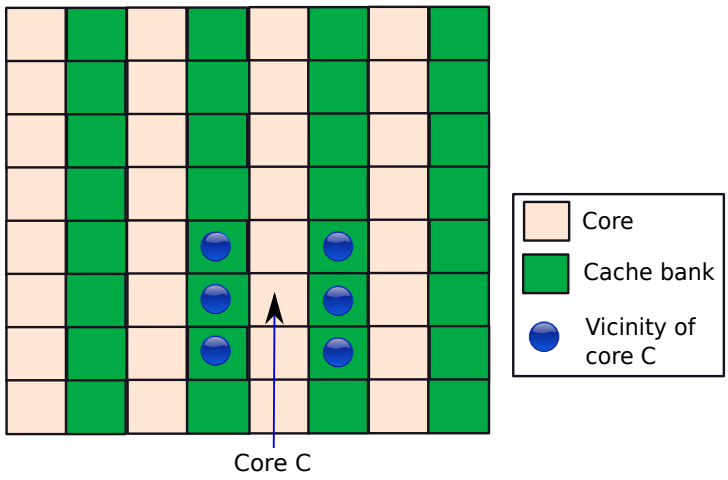


Figure 8.62: Placing private data in the vicinity of core C

- We initially load all the blocks in the private region of a thread to the home bank of its core, and thereafter we discourage their eviction by increasing their priority.
- We define a small window of banks in each bank set on both sides of the corresponding home bank. We put private cache blocks in these banks while initializing a thread. To further discourage their eviction, we always try to evict a block that is not private.
- We do not use the concept of bank sets for private data. Instead, we store all the blocks in banks around the requesting core C (see Figure 8.62). This means that we consider the banks that are closest to the requesting core (irrespective of their bank set), and store all the private cache blocks in them. Even though this approach is correct because these blocks are never shared, there will be a problem if the thread is migrated to another core. The other core needs to know about the location of the private blocks. We can have severe correctness issues unless we migrate all the private data to the vicinity of the new core. This is very expensive.

8.6 Performance Aspects

Let us now look at some ways in which we measure the performance of an NoC. We need to start with a disclaimer that the nature of evaluation depends on the type of optimizations that are proposed and the final use case. For example, if we are designing the NoC for a low power processor, then we should be interested in measuring power consumption. If we are designing the NoC for a high performance processor, then we should be interested in latency and throughput. To assess the performance of an NoC, it is necessary to simulate network traffic and observe the behavior of the NoC. Let us first describe the metrics of interest, and then we shall discuss popular simulation techniques.

8.6.1 Evaluation Metrics

Here is the list of metrics that are used to evaluate an NoC.

Latency One of the simplest metrics is the latency (measured in terms of clock cycles), which is the time that a packet takes to go from a source to a destination. This can be affected by the choice of the route and the degree of congestion in the network. Now, if we consider multiple source-destination pairs, then there are several ways in which we can aggregate this information. We can either consider the mean latency, the variance of the latency divided by the mean, or the mean latency per hop. The mean latency per hop is independent of the distance from the source to the destination – it just indicates the average number of cycles the packet took to traverse each router and the outgoing link. This gives an indication of the degree of congestion in the network.

Throughput The *throughput* of a network is simply defined as the number of bytes that are transferred per unit time throughout the network. A simple way to measure it is to simulate the network for a large number of cycles and compute the throughput over time. Finally, we can report the mean value. Note that the throughput is not the same as the bandwidth. The latter is a theoretical maximum; however, the throughput is a value that is practically observed. The throughput points to the overall efficiency of data transfer in a network.

Energy/Power In modern NoCs, energy and power consumption are important issues. We would like to minimize the energy that the NoC consumes. This can be done by reducing the size of the routers, powering down parts of a router when they are not in use, and minimizing the length of routes.

Area and Routing Complexity An NoC requires on-chip resources for all the links (wires) and the routers. We need to ensure that we have enough space to place all the wires, and also ensure that existing connections between circuit elements do not get lengthened. This requires very sophisticated NoC wire placement algorithms. Additionally, we need to minimize the area that the routers take such that we have enough space left for cores and caches.

8.6.2 Simulation Methodologies

The standard way of simulating the performance of an NoC is to use an NoC simulator. In such a simulator, we completely simulate the behavior of all aspects of the NoC including the routers, the links, and the logic to perform flow control. An NoC simulator can either be standalone, or can be coupled with an architectural simulator that also simulates the processor and the memory system. A standalone NoC simulator needs to be provided inputs regarding the traffic that needs to be simulated. The inputs can be of two types: statistical or trace based.

Standalone Simulators

Statistical inputs specify the probability of injecting a packet in a given node in a cycle. This is known as the *injection rate*. For example, if the injection rate is 0.1 (per cycle) at a node, then it means that there is a 10% probability that the node will start the transmission of a packet to any destination in a cycle. In most network simulations, we typically vary the injection rate and study the behavior of the network. In addition, it is possible to change the uniform probability distribution of packet injections to either a normal distribution, or a Weibull distribution. Now, for a given a packet source, let us look at the different methods to determine the destination. These are also known as different types of traffic.

Types of Traffic

It is necessary to choose a destination node for a given source node while synthetically generating traffic with statistical injection rates. Let us assume a network that is either a torus or a mesh. Each node has a x-y coordinate (each n bits). Then the location of each node can be specified with a $2n$ bit number: $x_{n-1} \ldots x_0, y_{n-1} \ldots y_0$. Let us now discuss different traffic patterns. Refer to Figure 8.63, where for the sake of readability we only show the traffic from the shaded cells. In the following descriptions, we assume

that we send a message from the source, S, to the destination, D. Both S and D are $2n$-bit vectors, where the upper (more significant) n bits represent the x coordinate, and the lower (least significant) n bits represent the y coordinate. We shall represent the x and y coordinates of S using the terms S_x and S_y respectively. Likewise, we define the terms D_x and D_y for the destination. Let $S[i]$ indicate the i^{th} bit in S, where we start counting from 0 (similar definition for $D[i]$). The LSB is always bit 0.

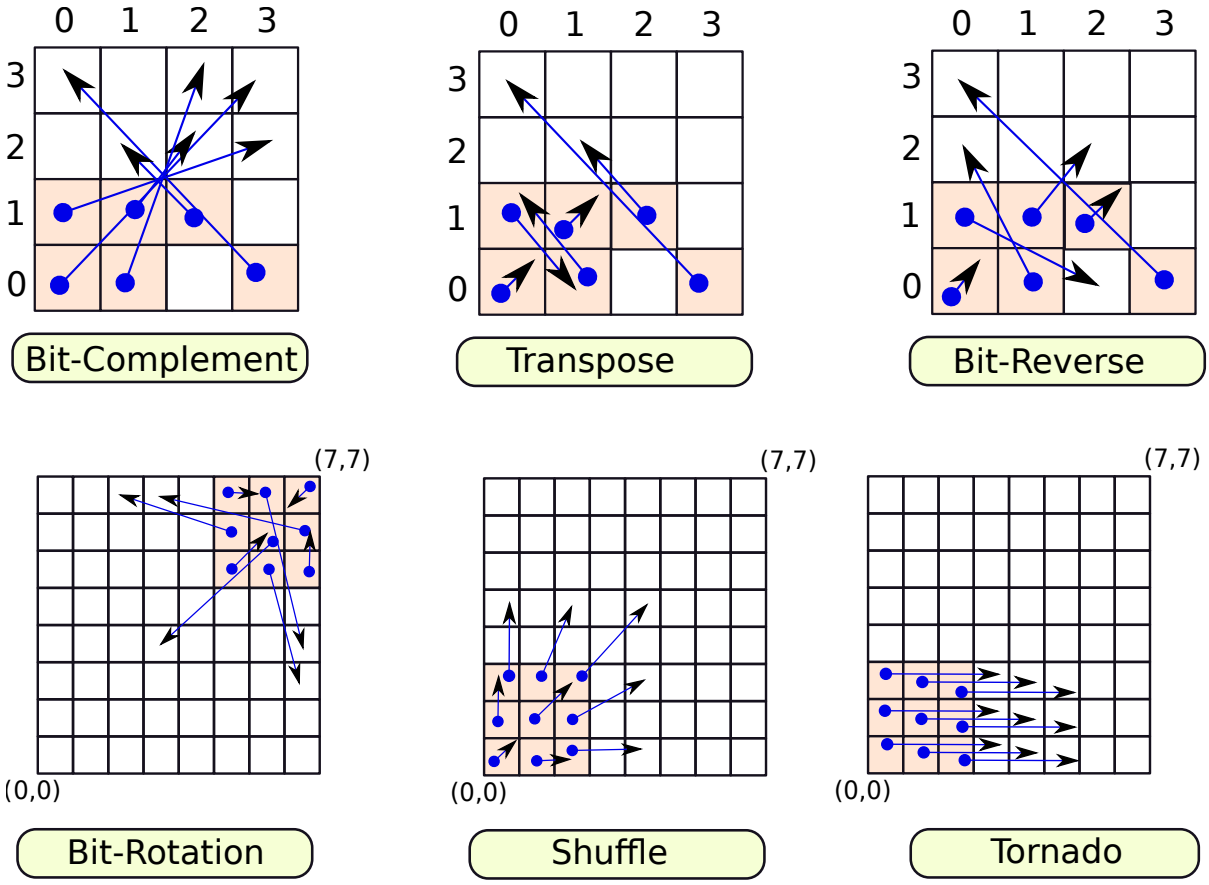


Figure 8.63: Types of traffic

Random In this case, we randomly choose one of the rest of the nodes with a uniform probability distribution. This kind of simulation method is used when we expect to run workloads without any known communication pattern.

Bit-Complement We consider the 1's complement of each of the coordinate values. The destination is equal to $(\bar{S_x}, \bar{S_y})$. For example, if the source is (0,1), the destination is (3,2) in a (4×4) 16-node network. As we can see from Figure 8.63, in this communication pattern, messages try to move towards the diagonally opposite corner. This is a good pattern to test the overall throughput of the network because messages typically tend to traverse long distances.

Transpose The destination is (S_y, S_x) (x and y coordinates of the source interchanged). Such a communication pattern is typically found in linear algebra applications that compute functions on transposed matrices. Here the source and the destination are the same for all the elements on the diagonal ($(0,0)$ to $(3,3)$). This is shown in Figure 8.63, where the source and the destination are

in the same cell for the diagonal elements. The rest of the messages try to cross the diagonal and reach a point that is as far from the diagonal as the source. As compared to Bit-Complement, most of the movement of messages happens in a direction that is perpendicular to the diagonal.

Bit-Reverse In this case, we just reverse the binary bits of the source to get the destination. Formally, we have $D[i] = S[2n - 1 - i]$. Such a communication pattern is found in some implementations of the FFT (Fast Fourier Transform) algorithm. Even though the pattern looks similar to Bit-Complement and Transpose; however, it is far more irregular in nature, with both short distances, and very long distances.

Bit-Rotation This pattern is based on shifting the bits of the source to get the destination. We right shift the $2n$ bits of the source to get the destination. The LSB that is shifted out becomes the MSB of the destination (see Figure 8.64) and the following equations.

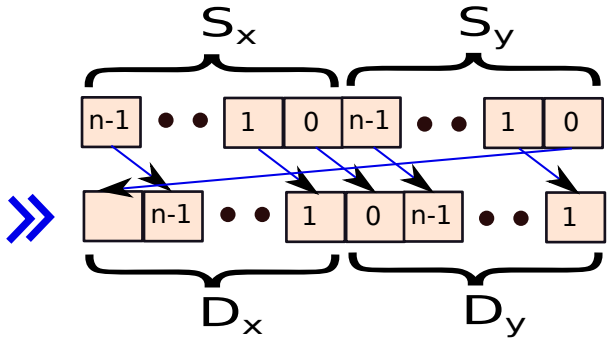


Figure 8.64: Bit-Rotation Pattern

$$\begin{aligned} D_x &= [S_y[0], S_x[n-1], \ldots S_x[1]] \\ D_y &= [S_x[0], S_y[n-1], \ldots S_y[1]] \end{aligned} \tag{8.7}$$

The logic is as follows. After we right shift S_x, we are left with $n - 1$ bits: $S_x[n-1] \ldots S_x[1]$. These are the lowest $n - 1$ bits of D_x. The MSB is set to the LSB of S_y, which is $S_y[0]$. We follow a similar procedure for generating D_y. In simple terms, D_x (or D_y) is generated by shifting S_x (or S_y) one position to the right. The MSB is equal to the LSB of S_y (or S_x). As we see in Figure 8.63, a right shift by 1 position reduces the value of the x and y coordinates in most cases and brings the point closer to the origin (left bottom). However, in some cases, the results are not quite as expected, the reason is that the MSBs that we set can be equal to 1, and thus the net effect is that the points might move further away from the origin. For example, if S_x is odd, then we will shift in a 1 into the MSB of D_y, and thus it may not decrease; however, if S_x is even, then the value of D_y will roughly halve (right shift by 1).

Shuffle This is the reverse of Bit-Rotation. Instead of shifting to the right, we shift to the left. This pattern of communication is frequently seen while computing FFTs (refer to Figure 8.63 for a visualization).

$$\begin{aligned} D_x &= [S_x[n-2], \ldots S_x[0], S_y[n-1]] \\ D_y &= [S_y[n-2], \ldots S_y[0], S_x[n-1]] \end{aligned} \tag{8.8}$$

The value of the MSB that is shifted out of S_x is made the LSB of D_y and vice versa. The intuition here is that we are roughly doubling the coordinates for the destination, when we are moving away

from the origin. This allows us to quickly broadcast messages in a direction away from the origin. Many algorithms use this pattern to quickly divide the work and map it to the nodes.

Tornado In an $N \times N$ matrix, the destination is $((S_x + \lceil N/2 \rceil - 1)\%N, S_y)$. This basically means that we send a message that is roughly $N/2$ nodes away on the same row. If we reach the end of the mesh, then we wrap around. Such kind of patterns are typically seen while solving a system of differential equations, where a core computes the result of some computation and passes it to a subset of its neighbors.

Trace Based Simulators

A trace based NoC simulator does not use any statistical models. We can create traces using processor emulators or synthetic traffic generators. The traces contain details of the packets that each node needs to inject and their destinations. The NoC simulator can then read in these traces and simulate the NoC. The advantage of such kind of simulation is that we can simulate the NoC for specific applications. Generic statistical models do not model specific applications very well.

Architectural Simulation

Most generic architectural simulators such as the Tejas Simulator (see Appendix B) also contain NoC simulators. Such simulations are very accurate because we are additionally modeling the cores and the memory system as well. The exact timing gets simulated more accurately, and the NoC gets the right inputs at the right time. The NoC simulation module by itself is very similar to standalone NoC simulators. Architectural simulators are however much slower than NoC simulators because they need to simulate the rest of the components such as the cores and the memory system as well.

8.7 Summary and Further Reading

8.7.1 Summary

Summary 7

1. *In modern multicore processors, we have tens of cores and cache banks. It is necessary to connect them with an on-chip network (NoC).*
2. *We typically divide a chip into tiles, where each tile contains a set of co-located cores and cache banks. We assign one router to each tile.*
3. *A* router *is the most basic element in an NoC. Its job is to route messages between neighboring routers or accept/initiate messages on behalf of its local tile. It mainly does routing (finding the path to the destination) and flow control (managing the flow of bits between neighboring routers without any information loss).*
4. *A router is also called a* node *in a network, and two nodes are connected by a link – a set of parallel copper wires to carry multiple bits simultaneously.*
5. *The sender node sends a message to a receiver node. A message is typically broken down into packets, where an entire packet is routed as one atomic unit. Each packet is further subdivided into flits (flow control units) and a flit is divided into phits (physical bits). All the bits in a flit are stored together in routers, and all the bits in a phit are transmitted together on the physical copper wires.*

6. *The two flow-control schemes for transmission across a single link are credit based flow control and on-off flow control. They ensure that a message is sent to the destination only when it has the space for it.*

7. *For the entire NoC, we can do flow control at the level of messages, packets, or flits. A common message based flow control method is circuit switching, where we reserve an entire path from the source to the destination. Even though the path setup overhead is high, we can amortize the costs if the messages are long and the contention is low.*

8. *The two most common methods for flow control at the level of packets are store and forward (SAF) and virtual cut through (VCT). In the SAF approach, we first store an entire packet in a router, and then send its head flit to the next router. In the VCT based approach, we don't have to wait for the entire packet to be buffered at a router before we send its head flit to the next router along the way. Note that in both cases, we need to have enough space to buffer the entire packet in each router.*

9. *In flit based flow control, buffering at routers is done at the level of flits. We need not buffer the entire packet; however, all the flits in a packet are constrained to travel along the same route. The two most common algorithms for flit based flow control are wormhole flow control and virtual channel based flow control.*

10. *Virtual channel based flow control is more efficient than wormhole flow control because it multiplexes a physical channel between different packet transmissions. It conceptually breaks a physical channel into different virtual channels (VCs), where the different VCs have their dedicated set of buffers.*

11. *The three main concerns in routing flits across the NoC are deadlocks, livelocks, and starvation.*

12. *Livelocks and starvation can be avoided by adding the notion of a* timeout *to a message. If we augment each message with a hop count, and then send it along the shortest path to the destination after the hop count crosses a threshold, we will avoid the possibility of livelocks. Similarly, we can avoid starvation if we can ensure that a packet or a flit has the highest priority for being allocated a resource if it has waited for more than a given number of cycles.*

13. *Avoiding deadlocks is far more involved. We use two theoretical tools to reason about protocols that are engineered to avoid deadlocks: channel dependence graph (CDG) and turn graph (TG). A channel dependence graph shows the dependence between physical or virtual channels. There is an edge from channel C_1 to C_2 if the flit that holds C_1 is waiting for C_2 to become free. A turn graph is a subset of the graph representing the network. We typically build a turn graph with a small subset of nodes, and links (channels), when we want to take a deep look at the nature of dependences in a given region of the network. Since a dependence in the channel graph is a turn in the turn graph, we can characterize situations such as deadlocks in terms of the turns that the flits want to take.*

14. *X-Y routing is a simple routing algorithm (provably deadlock-free), where we traverse along the x direction first and then traverse along the y direction.*

15. *To increase the path diversity in X-Y routing, we can use oblivious routing, where we first route a message to an intermediate node, and then route it from the intermediate node to the destination. This helps us deal with congestion better.*

16. *Adaptive routing is a more efficient solution where we avoid 2 out of 8 possible turns: one from a clockwise cycle and one from an anti-clockwise cycle.*

17. *We can use the notion of virtual channels to design deadlock free protocols by either ensuring that we get VCs in a certain order (using date lines) or by using escape VCs.*

18. *A router has 5 stages: buffer write (BW), route computation (RC), VC allocation (VA), switch allocation (SA), and switch traversal (ST).*

19. *An* allocator *matches N requests with M resources, whereas an* arbiter *is far more specific; it matches N requests with just 1 resource. We discussed the round robin arbiter, the matrix arbiter, the separable allocator, and the wavefront allocator.*

20. *Several strategies can be used to reduce the latency of a router's pipeline.*

    (a) *We can remove the RC stage from the critical path in lookahead routing by computing the route taken by the packet from the next router on the path towards the destination.*

    (b) *If the contention is low, we can directly try to access the switch and send the packet on the outgoing link. This method is known as bypassing.*

    (c) *If the VCs are normally free, we can try to allocate a VC speculatively. In this case, the VA stage can be moved off the critical path.*

21. *NoC delays can vary between 5 and 50 cycles. This makes the latency of accesses to the last level cache (LLC) quite nondeterministic. To reduce performance losses associated with such variable delays, researchers have proposed NUCA (non-uniform cache architectures) schemes for large caches.*

22. *In an S-NUCA (static NUCA) cache, we distribute all the blocks of the LLC among a multitude of cache banks spread throughout the chip. This scheme does not counteract the effect of large NoC latencies.*

23. *In a D-NUCA (dynamic NUCA) cache, we divide banks into non-overlapping bank sets. We search for a block in the home bank: the bank that is closest to the requesting core in the bank set. If we do not find the block, then we search the rest of the banks in the bank set. If there is a hit, we migrate the block towards the home bank, otherwise we access the lower level of the memory hierarchy.*

24. *The method of choice for assessing the performance and throughput of an NoC is simulation.*

    (a) *We can generate traffic synthetically. These are statistical models that have been derived from many real-world applications.*

    (b) *We can collect traces and feed them to NoC simulators.*

    (c) *Most architectural simulators also contain NoC simulators that take inputs from the memory system and realistically simulate the NoC traffic.*

8.7.2 Further Reading

For an alternative perspective, readers can consult the textbook by Dally and Towles [Dally and Towles, 2004]. The e-book by Jerger [Jerger et al., 2017] describes many contemporary developments in the field. Other sources of information are survey papers on on-chip optical networks such as [Bjerregaard and Mahadevan, 2006, Salminen et al., 2008].

Some major contributions and novel ideas in the design of on-chip networks are as follows: bufferless routing [Moscibroda and Mutlu, 2009], network for an 80-tile chip [Vangal et al., 2007], Garnet network simulator [Agarwal et al., 2009], and express virtual channels that bypass routers [Krishna et al., 2008].

Finally, in the future it is expected that non-conventional interconnects using optical or wireless technology might replace traditional copper based electrical interconnects. A general introduction to the area is given by Karkar et al. [Karkar et al., 2016]. For optical NoCs, readers are referred to the survey paper by Bashir et al. [Bashir et al., 2019] and for wireless NoCs, readers can refer to the report by Li [Li, 2012].

Exercises

Ex. 1 — How does the number of stages in the butterfly topology affect the packet drop rate?

Ex. 2 — Design a modified butterfly topology for 12 nodes using 3 switch stages. Note that in this design, no switch port should be left unused and the radix of all the switches in the same stage should be the same.

Ex. 3 — What is the advantage of flow control at the level of flits?

Ex. 4 — How does a virtual channel increase the throughput of an NoC?

Ex. 5 — Is it possible for a routing protocol with 7 allowed turns to be deadlock-free? Justify your answer.

Ex. 6 — Consider the following routing protocol in a 2D mesh, where the columns are numbered from 1 to N.

- A packet in an even column is not allowed to make the following two turns: east to north and east to south.
- A packet in an odd column is not allowed to make the following two turns: north to west and south to west.
- A packet cannot make a U-Turn.

Prove that this protocol is free of deadlocks.

Ex. 7 — Consider a routing scheme for a 2D mesh where all routes are restricted to at most three right turns, and left turns are not allowed. Is this scheme free of deadlocks? Justify your answer.

Ex. 8 — Among the various routing algorithms described in this chapter, which algorithm is bested suited for the following?

1. Minimum message latency.
2. Maximum throughput.

Ex. 9 — List the typical optimizations that are done in a router's pipeline. Explain their benefits.

* **Ex. 10** — Propose a scheme where we can vary the number of VCs per physical channel based on the traffic pattern.

Ex. 11 — Assume we are using the credit-based flow control mechanism. Propose a scheme to plan the allocation of VCs and switch ports for a few cycles in the future such that these stages can be moved off the critical path.

** **Ex. 12** — Let us say that based on historical data some pairs of nodes and cache banks tend to communicate a lot. Can we leverage this fact to design an NoC that can quickly deliver messages between

these pairs of nodes? How can we ensure that messages can quickly bypass the router's pipeline and traverse from the sender to the receiver (for some source-destination pairs only).

Ex. 13 — Do we need a NUCA cache if we have an ultra-fast interconnect such as an on-chip optical or wireless network?

* **Ex. 14** — Consider the following situation in a NUCA cache. We send a request and there is a miss. However, the block is there in the cache. The *search* message did not find the block because the block was at that moment in transit between two cache banks. How do we detect and prevent such race conditions?

** **Ex. 15** — Instead of designing NUCA caches where each bank set is arranged as a column, can we create other arrangements? Suggest a few and comment on their pros and cons.

Design Problems

Ex. 16 — Design the circuit of a wavefront allocator using a hardware description language (HDL) such as VHDL or Verilog.

Ex. 17 — Design a pipelined router with all the optimizations using an HDL. The final circuit should give higher priority to flits that have been in flight for a longer time.

Ex. 18 — Implement the odd-even routing protocol in the Tejas architectural simulator.