Chapter 3
Baseline Switching Modules and Routers

Having described the flow of data on a point-to-point link (1-to-1 connection) and the implications of each design choice, in this chapter, we move one step forward and describe the operation of modules that allow many to one and many to many connections. The operation of such modules involves, besides flow control, additional operations such as allocation and multiplexing that require the addition of extra control state per input and per output.

In many cases, it is advantageous to allow two or more peers to share the same link for transmitting data to one receiver. This is a common example in modern systems like when many on-chip processors are trying to access the same off-chip memory controller. An example of sharing a link by many peers is shown in Fig. 3.1. In this example, each input (IP core) generates one packet that is heading towards the memory controller (receiver). The link cannot accommodate the flits of many packets simultaneously. Therefore, we need to develop a structure that would allow both packets to share the wires of the link.

Sharing the wires of the link requires the addition of a multiplexer in front of the link (at the output of the multiple-input sender). The multiplexer select signals are driven the arbiter that determines which input will connect to the memory controller. We have two design options on how to drive the select signals. The arbiter can select in each cycle a different input or it can keep the selection fixed for many cycles until one input is able to transmit a complete packet. VCT and WH switching policies requires that each packet is sent on the link un-interrupted. In other words, once the head of the packet passes the output multiplexer the connection is fixed until the tail of the same packet passes from the output port of the sender.

Alternatively, we could change the value of the multiplexer's select signal on each cycle, thus allowing flits of different packets to be interleaved on the link on consecutive cycles. This operation is prohibited for WH and VCT and can be applied only when more state is kept for the packets stored at the receiver. This extra state makes the input buffer of the receiver look like a parallel set of independent queues, called virtual channels, and is the subject of the following book chapters.

G. Dimitrakopoulos et al., *Microarchitecture of Network-on-Chip Routers: A Designer's Perspective*, DOI 10.1007/978-1-4614-4301-8_3

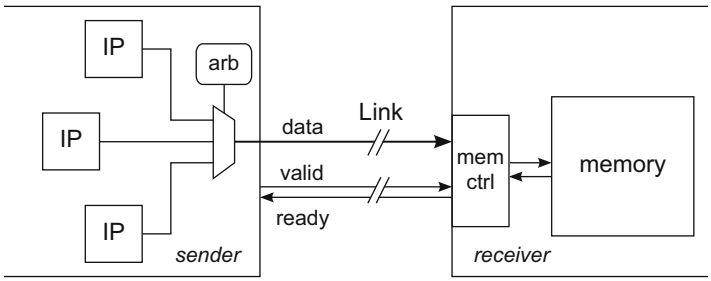

Fig. 3.1 Sharing a memory controller to multiple cores of the chip

The arbiter that drives the select signals of the multiplexer is a sequential circuit that receives the requests from the inputs and decides which input to grant based on its internal priority state. The priority state keeps track of the relative priorities of the inputs using one or more bits depending on the complexity of the priority selection policy. For example, a single priority bit per input suffices for round-robin policy, while for more complex weight-based policies, such as firstcome- first-served (FCFS) or age-based allocation, multi-bit priority state is required. Round-robin arbitration logic, which is the most widely applied policy and the easiest to implement, scans the input requests in a cyclic manner beginning from the position that has the highest priority and grants the first active request. On the next arbitration cycle, the position that was granted receives the lower priority. The design details involved in arbiter design can be found in Chap. 4.

In this chapter, we begin our discussion on switching with the simple example of many inputs sending data to a shared output via a common link and next we will describe how this simple design can evolve gradually to support multiple outputs, thus actually deriving a fully fledged NoC router.

3.1 Multiple Inputs Connecting to One Output

The design of a multiple-input to one output connection besides arbitration should take also into account the output flow control mechanism for guaranteeing that the flits leaving from each input will find the necessary buffer space in the shared output.

Without loss of generality we assume that each input is attached to the switching module (arbiter and multiplexer) via a buffer that respects the ready/valid handshake protocol. The same holds for the output. The output buffer accepts the valid and the associated data from the output multiplexer and returns to all inputs one ready signal that declares the availability of buffer space at the output. The buffers at the inputs and outputs can be either simple 1-slot EBs, or 2-slot ones or even larger FIFOs that can host many flits. An abstract organization of the multiple-input-one-output connection is shown in the left upper side of Fig. 3.2.

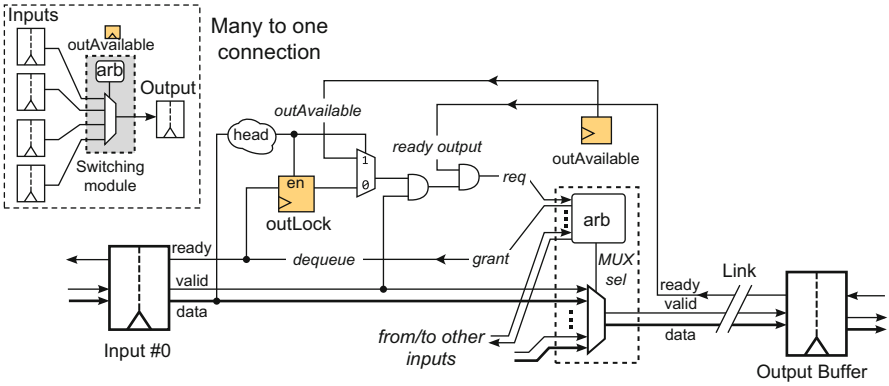

Fig. 3.2 The organization of a many-to-one connection including the necessary output and input state variables as well as the request generation and grant handling logic that merges switching operations with link-level flow control

Each input wants to send a packet that contains one head flit, a number of body flits and a tail flit that declares the end of the packet. Since each flit should travel on the shared link as an atomic entity the link should be allocated to the packet as a whole: The head flit will arbitrate with the head flits of the other inputs and once it wins it will lock the access to the output. This lock will be released only by the tail flit of the packet. Therefore, an output state variable is needed, called *outAvailable*, that declares the output's availability to connect to a new input. The same state bit exists also at each input and called *outLock*. When $outLock[i] = 1$ means that the ith input has been connected to the output. Following Fig. 3.2, the *outAvailable* flag is placed at the output of the switching module (arbiter and multiplexer). This is the most reasonable placement since the *outAvailable* state bit is not a characteristic variable of the output buffer (or the receiver in general) but a variable needed to guide the arbitration decisions taken locally by the switching module at the sender's side.

Each input needs to declare its availability to connect to the output. This is done via the valid bits of the input buffers. For the head flits, as shown in Fig. 3.2, the valid bits are first qualified with the *outAvailable* state bit. If the output is not available, all valid bits will be nullified or kept alive in the opposite case. Before transferring the qualified valid bits to the arbiter we need also to guarantee that there is at least one free buffer slot at the sink. Therefore, the qualified valid bits are masked with the ready signal of the output that declares buffer availability. After those two masking steps the valid signals from each input act as requests to the arbiter that will grant only one of them. Once the arbiter finishes its operation it returns a set of grant wires that play a triple role.

- They drive the select signals of the output multiplexer that will switch to the output the flit of the selected input.

- They set the *outLock* bit of the winning input. In the next cycles, the body and the tail flits do not need to qualify their requests again with *outAvailable* but they are driven directly from the *outLock* bit provided that they have valid data to sent.
- They drive the ready_in signals of the inputs. The assertion of the appropriate ready_in signal will cause a dequeue operation to the corresponding input buffer since both its valid_out and its ready_in signal will be asserted in the same cycle. The inputs that did not win will see a ready_in = 0 and thus they will keep their data in their buffer.

When the tail flit leaves the source it de-allocates the per-input and per-output state bits *outLock*[i] and *outAvailable*, respectively, by driving them to their free state. Once *outAvailable* is asserted, the inputs with valid head flits can try to win arbitration and lock the output for them, provided that there is buffer space available at the output.

Using this simple configuration, arbitration is actually performed in each cycle for all flits. However, once *outAvailable* $= 0$, meaning that the output has been allocated to a specific input, and *outLock*[i] $= 1$, meaning that the selected input is the ith one, then only the requests of that input will reach the arbiter. The requests of the rest inputs will be nullified expecting the output to be released. In the meantime, the arbiter always grants input i and updates its priority to position $i + 1$ (next in round-robin order so that input i has the least priority in the next cycle). During a packet's duration from a specific input, the priority of the arbiter will always return to the same position since only one (and the same) request will be active every cycle. Once the output is released by the tail of the packet the priority will move to a different input depending on which input was finally granted.

In many real cases, it is necessary to isolate the timing path of the link from that of the arbitration and multiplexing. The obvious choice is to add an EB, preferably with 2 slots, that isolates the timing paths and provides additional buffering space, i.e., outgoing data can stop independently at the output of the multiplexer. In this configuration, shown in Fig. 3.3, the ready signals of the output that were used as qualifiers in the example of Fig. 3.2 are replaced by the ready signals of the intermediate EB. The rest request generation logic remains the same and the $outAvailable$ flag is updated when a head/tail flit passes the output of the multiplexer and moves to the intermediate EB.

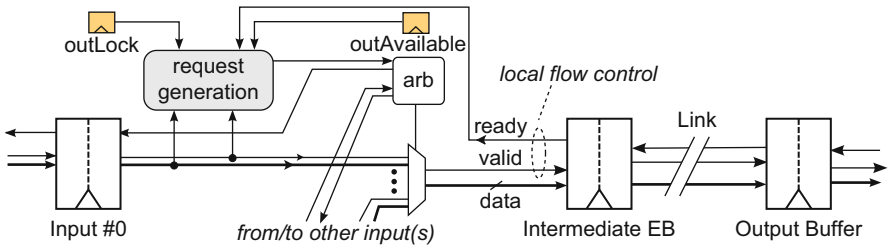

Fig. 3.3 The addition of a local output EB isolates the operation of the switching module from link traversal

3.1.1 Credit-Based Flow Control at the Output Link

The baseline architecture involves a buffer at the output module as well as an optional intermediate one. Assuming that the intermediate buffer is not present we can re-draw the microarchitecture of the primitive switching element following the abstract flow control model developed in Chap. 2. In this case, illustrated in Fig. 3.4a, the output buffer consists of a data buffer and a free slots counter that is updated by the output buffer for increasing its value and by the incoming valid signals from the output of the multiplexer for reducing its value.

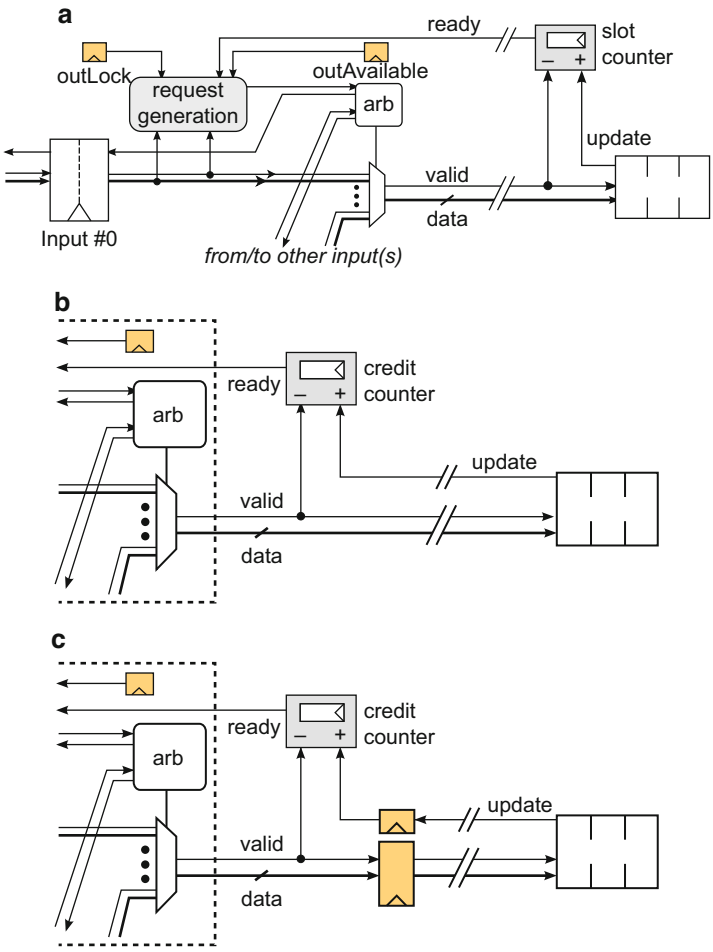

Fig. 3.4 The changes required in order for the switching module to connect to a credit-based flow controlled link. The output of the switching module may include an additional pipeline register for isolating the internal timing paths from the link

Equivalently the slot counter can be moved at the output of the switching module (at the other side of the link) and act as a local output credit counter as shown in Fig. 3.4b. The output credit counter mirrors the available buffer slots of the output. It sends a ready signal to all inputs when the number of available buffer slots at the output buffer is greater than zero. The inputs qualify their valid signals exactly the same way as in the case of the ready/valid handshake. Therefore, when a certain input is connected to the output (the output was available and the arbiter granted the particular input), it knows exactly about the availability of new credits at the output via the output credit counter.

It should be noted that the ready signal that is asserted when $creditCounter > 0$, is only driven by the current state of the credit counter. The credit decrement and increment signals update only the value of the credit counter and the new value will be seen by the ready signal in the next clock cycle. Therefore, the dependency cycle formed by credit decrement → ready → request generation → arbiter's grant → credit decrement is broken after the ready signal, which also helps in isolating the timing paths starting request generation logic. Equivalently, each input buffer, independent from the rest, sends also its own credit update in the backward direction once it dequeues a new flit.

Using the output credit counter simplifies also the addition of pipeline stages on the link. For example in Fig. 3.4c the output of the multiplexer is isolated by a simple pipeline register, i.e., outgoing data cannot stop at this point, and the readiness of the output buffer is handled via the output credit counter. As described also in the previous chapter referring to a single point-to-point link, even if additional pipeline stages are added between inputs and the output once the ready signal is consumed by the input without any further delay the credit protocol guarantees maximum throughput will the least buffering requirements. In this case, the receiver needs to provide 3 buffer slots to absorb the in-flight traffic due to the increased forward and backward latency $L_f = 2$, $L_b = 2$.

3.1.2 Granularity of Buffer Allocation

Under WH switching principle, each flit of a packet can move to the output assuming that at least one credit is available. On the contrary VCT requires flow control to extend at the packet level by allocating any buffering resources at packet granularity. In both cases the flits of the packets are not interleaved at the output. Interleaving is enabled by virtual channels that will be presented in the following chapters.

In a packet-based flow control, which is commonly used in off-chip networks, both the channels and the buffers are allocated in units of packets, while flit-based flow control allocates both resources in units of flits. On-chip networks have often utilized the flit-based flow control. The main difference between packet and flit-level flow control is in how the buffer resource is allocated. With packet-based flow control before any packet moves to an output, the buffer for the entire packet needs to be allocated; thus, for a packet of L flits, an input needs to obtain L credits before

the packet can be sent. Once the buffer for the entire packet has been allocated, the channel resource can be allocated on flit granularity. A multi-flit packet can be interrupted during transmission from input to output; the packet will not necessarily be sent continuously. However, when the head flit arrives at an input, it reserves the next L slots in the output buffer such that the whole packet to be kept at the output in the case of downstream stall.

Even if using flit-level flow control, buffers can be allocated at the packet level by employing atomic buffer allocation. In this case, the head flit of a packet is not buffered behind the tail flit of another packet in the same buffer. In effect, buffers are implicitly allocated on packet granularity, even if flit-based flow control is used. This operation can be achieved by not releasing the *outAvailable* flag when the tail flit arrives at the output buffer but when it leaves the output buffer. In this way, when the next head flit arrives, it will find the output buffer empty. In every case that the buffers are allocated at the packet level the amount of buffers required is equal to the size of the longest packet, which inevitably leads to low buffer utilization for short packets.

In the following we adopt the non-atomic buffer allocation principles. However, in any case that atomic buffer allocation is needed the aforementioned rules can be applied to enforce it.

3.1.3 Hierarchical Switching

Arbitration and multiplexing for reaching the output link can be performed hierarchically by merging at each step a group of inputs and allowing one flit from them to progress to the output. An example of a hierarchical 1-output switch organization is depicted in Fig. 3.5a. The main difference of hierarchical switching relative to single-step switching is that at each step a 2-input arbiter and a 2-to-1 multiplexer is enough to switch the flits between two inputs, while in the single-step case the arbiter and the multiplexer employed should have as many inputs as the inputs of the whole switching module.

To achieve maximum flexibility and increase the throughput of the system by allowing multiple packets to move in parallel closer to the output, we should modify also the request generation logic of the baseline design. In the baseline case, every input before issuing a request to the arbiter qualified its valid signal with the *outAvailable* flag of the output and then masked the result with the ready signal of the output buffer (see Fig. 3.2). In the hierarchical implementation this is not possible since there is no global arbiter to check the requests of all inputs. Instead, we assume that each merging point can be considered as a partial output and has its own *outAvailable* flag. In this way, at each merging point, we can use unchanged the allocation and multiplexing logic designed for the baseline case (Fig. 3.2) including also the *outLock* variable at the input of each merging step.

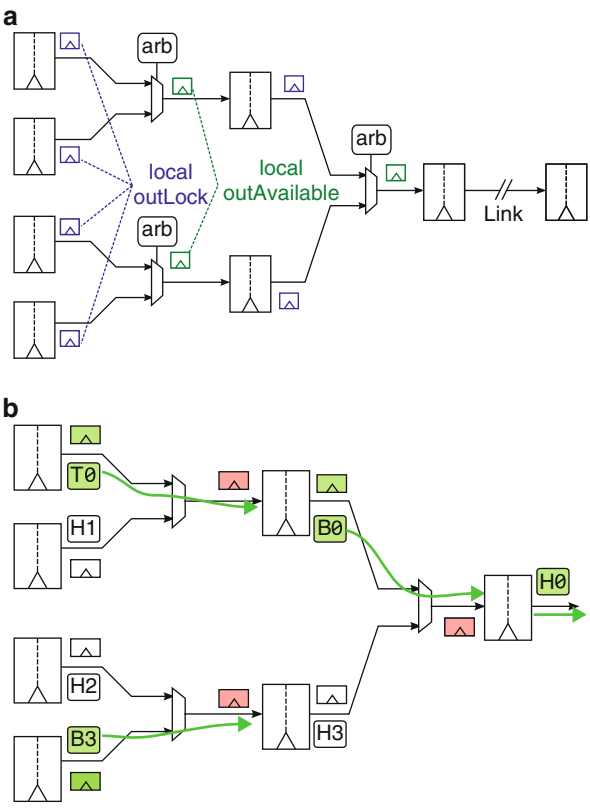

Fig. 3.5 (**a**) The organization of a hierarchical multiple-inputs to one output switching module and (**b**) the interleaving of flits from different inputs at the branches of the hierarchical switching module

In this hierarchical implementation of the switch, every two inputs either at the first stage of arbitration or inside the multiplexing tree can gain access only to the local output that they see in front of them (the output of a merging unit). In this way, the flits of a packet can move atomically up to the point that they see the corresponding *outLock* bits set. If another branch of the merging tree has won access to the next intermediate node, then the flits of the packet stall and wait the next required resource to be released. This partial blocking of packets inside the merging tree is shown in Fig. 3.5b. In this way, even if some branches of the tree remain idle due to downstream blocking, the allocation of different branches of the tree to different packets increases the overall throughput of the switching module. Switching single-flit packets (acting both as head and tails) in this configuration allows for maximum utilization since every local output or the global one can be given to another branch on a per-cycle basis.

3.2 The Reverse Connection: Splitting One Source to Many Receivers

The opposite connection of one-to-many (splitting), i.e., distributing a result from the output to the appropriate input is simpler than the many to one connection (merging) described in the previous paragraphs. Once a new flit arrives at the output it should know the input to which it should be distributed. Then transferring the incoming flit is just a matter of flow control; to guarantee that the receiving input buffer has at least one position available. From all ready signals of the input receiving buffers only one is selected based on the destination id of the incoming flit. Once the selected signal is asserted a transfer occurs between the transmitting buffer at the output and the receiving buffer at the corresponding input. The organization of this split connection is shown in Fig. 3.6.

Please note that the receiving buffers at the input shown in Fig. 3.6 and the transmitting buffer at the output are different from the buffers shown in Fig. 3.2, which play the opposite role, e.g., the input buffers transmit new flits while the output buffer receives new flits.

In this splitting connection there is no obligation to send complete packets uninterrupted and flits from different packets can be interleaved at the output of the transmitting buffer, provided that they return to a different input. Additionally, when an input is not ready to receive new returning flits, there is no need for the rest inputs to remain idle. Allowing the output to distribute flits to the available inputs requires splitting the transmitting buffers to multiple ones; one per destination and adding the appropriate arbitration and multiplexing logic. In this case, the flits that move to different inputs cannot block each other, thus allowing maximum freedom in terms of distributing incoming flits to their destined inputs.

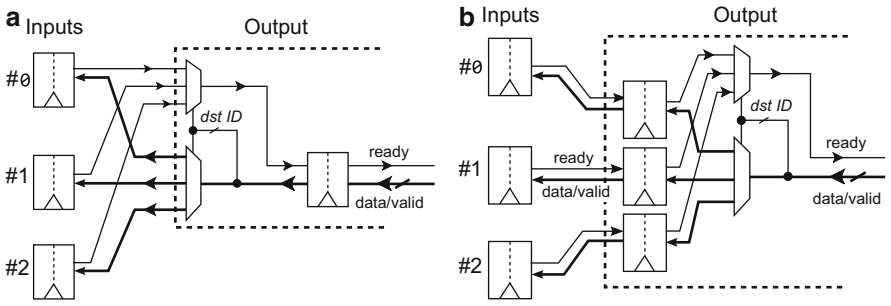

Fig. 3.6 (**a**) Splitting flits to multiple receivers requires only checking the buffer availability at the receiver's side. (**b**) Per-destination buffers remove any flow-control dependency across different receivers

3.3 Multiple Inputs Connecting to Multiple Outputs Using a Reduced Switching Datapath

The generalization of link sharing involves multiple outputs at the other side of the link as shown in the upper left corner of Fig. 3.7. In this case, each packet should know in its head flit to which destination it is heading to. The link can host flits from different sources provided that they move to a different output. Each output independently from the rest should see the flits of a packet arriving atomically one after the other from the same input. This limitation can be removed by adopting virtual channels as it will be shown in later chapters.

Since the switching module serves multiple outputs it holds a different *outAvailable* state bit for each output. When $outAvailable[j] = 1$ it means that the jth output is free and has not been allocated by any packet. Also, the switching module receives multiple ready bits; one from each output declaring buffer availability of the corresponding output. As in the baseline case, we assume that there is one arbiter and one multiplexer that should switch in a time-multiplexed manner multiple inputs to multiple outputs.

Each source receives the *outAvailable* flags from each output and selects the one that corresponds to the destination stored at the head flit of the packet. The selected *outAvailable* flag is masked as in the baseline case of a single output with the valid signal (not empty) of the buffer of the source. This is only done for the head flits in order to check if the destined output is available. In parallel each input receives the ready signals from all outputs and selects the one that corresponds to the selected output port. Masking the qualified valid signal with the selected ready bit guarantees

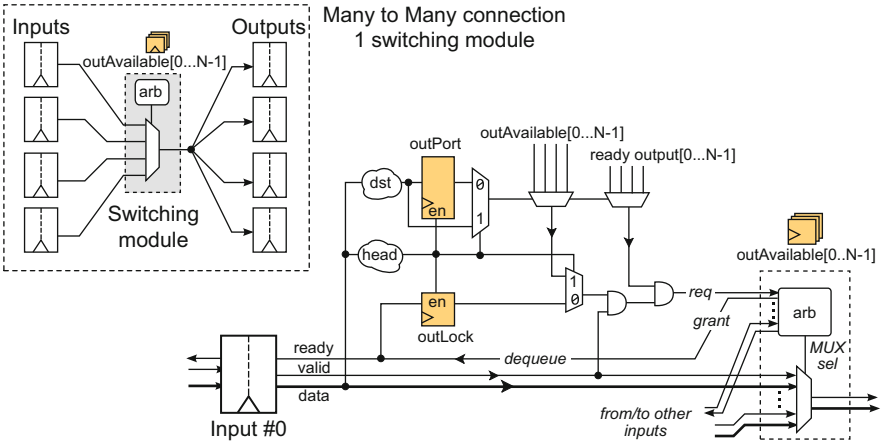

Fig. 3.7 The organization of a many-to-many connection using only one switching module including the request generation, the output and input state variables and the distribution of the necessary flow control signals for supporting N different output ports

that if the head flit wins arbitration it will find an empty buffer slot at the output. This request generation procedure is depicted in Fig. 3.7.

The arbiter receives the requests from all inputs and grants only one. The grant signals are distributed to all inputs. When the head flit of the ith input wins a grant, three parallel actions are triggered:

- The $outLock[i]$ variable is asserted.
- A new state variable that is added per input, called $outPort[i]$, stores the destined output port indexed by the head flit. This new variable is needed per input since after the head flit is gone, the body and the tail flits should know which output to ask for.
- The head flit is dequeued from the input buffer by asserting the corresponding ready_in signal.

The body and the tail flits drive their arbiter requests via their local $outPort[i]$ and $outLock[i]$ variables. Although the *outAvailable* flags are checked only by the head flits, the ready signals are checked every cycle by all flits of the packet. After the two masking operations – one for availability (only for the head flits) and one for readiness (for all flits) – are complete a new request is generated for the arbiter. The arbiter in each cycle can select a different input and move the corresponding flits to the appropriate output. In the next cycles, the packets from other inputs that will try to get access to an un-available output port will delete their requests at the request generation stage and thus only the locked input will be available for that output.

3.3.1 Credit-Based Flow Control at the Output Link

Under credit-based flow control, the inputs before sending any flits to their selected output should guarantee that there are available credits at that output. If this is true when a flit leaves the input buffer and moves to the output it consumes one credit from the appropriate credit counter. The implementation of credit-based flow control requires the addition of one credit counter for each output placed at the output of the switching module, as shown in Fig. 3.8. The credit counter reduces the available credits every time a new valid flit reaches the output and increases the available credits once an update signal arrives from the corresponding output buffer. Multiple credit updates can arrive in each cycle, each one referring to a different output buffer. Ready signals (one for each output) that declare credit availability, i.e., $creditCounter[i] > 0$, are generated by the counters at the output of the switching module and distributed to all inputs.

Equivalently the input buffers when they dequeue a new flit they are obliged to send backwards a credit update according to the credit-based flow-control policy.

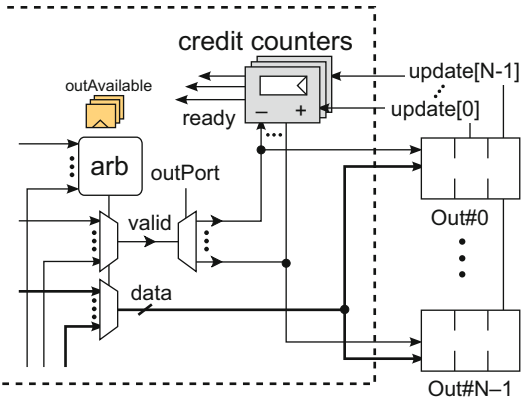

Fig. 3.8 The output includes one credit counter for each output that gets updated by a separate update signal. A credit is consumed when a flit passes the output multiplexer using the valid signal that is de-multiplexed to the selected output port

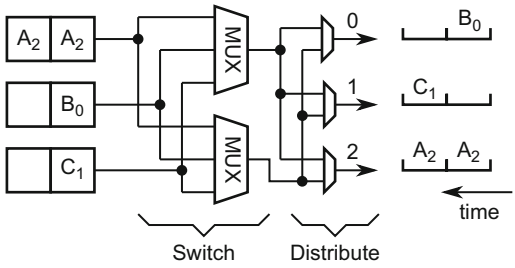

Fig. 3.9 The two multiplexers can service more inputs in parallel but the distribution of the flits to their destined output requires an additional distribution network of multiplexers

3.3.2 Adding More Switching Elements

Using one arbiter and one multiplexer for switching packets to many outputs limits the throughput seen at each output since at most one flit per cycle is dequeued from all inputs. We can increase the throughput of the whole switching module by adding more datapath logic, as shown in Fig. 3.9. In this case, the router is able to deliver two independent flits to any two available outputs. The internal datapath of the router now consists of two arbiters and multiplexers that prepare two output results. The flits that appear at the output of the multiplexers may belong to any output. Therefore, we need to add additional multiplexers that distributed the intermediate results to their correct output. In order for this circuit to operate correctly we need to guarantee beforehand that the intermediate results are heading to a different output. This means that the arbiters of the two multiplexers need to communicate and grant only the requests that refer to different outputs. This inter-arbiter communication serializes the allocation operation and limits the effectiveness of the switching element. This problem is solved if we fully unroll the datapath and provide a separate multiplexer per output that can connect directly to all inputs. The operation of the unrolled-datapath architecture is described in detail in the following section.

3.4 Multiple Inputs Connecting to Multiple Outputs Using an Unrolled Switching Datapath

The design of switching elements has evolved so far from simple point-to-point links (1-to-1 connections) that were useful in understanding the operation of flow control, to many-to-one connections as well as many-to-many connections using only one arbiter and one multiplexer. In this section, we focus on the many-to-many connection but try to increase the throughput seen by the switch as whole. Our main goal is to move from the 1 flit per cycle traversing the switch as shown in Fig. 3.7, to many flits travelling to different outputs per cycle. To achieve this we need to fully unroll the datapath presented in the previous section by adding a separate multiplexer and arbiter pair at each output following the connection of Fig. 3.10a. In this way, each output independently from the rest can accept and forward to the output link a new flit as shown in Fig. 3.10b. The set of per-output multiplexers constitute the crossbar of the switch that enables the implementation of an input-output permutation, provided that each input selects a different output.

Besides the unrolling of the datapath, the input and output operations involved remain more or less the same to the ones described for the reduced datapath. Each input and each output has its own buffer space that employs a ready/valid protocol and can be from a simple 1-slot EB to a fully fledged FIFO. The most simple choice can be a 2-slot EB that provides lossless and full throughput operation without allowing any backpressure combinational paths to propagate inside the switch and increase inevitably the clock cycle. While a 2-slot EB is enough for most cases, bursty traffic and high congestion in the network may call for more buffers that will absorb the extra traffic.

As in all previous cases, each input holds 2 state variables. The *outPort*[*i*] that holds the destined output port of the packet stored at the *i* th input and the *outLock*[*i*] bit that declares whether the packet of the particular source has gained an exclusive

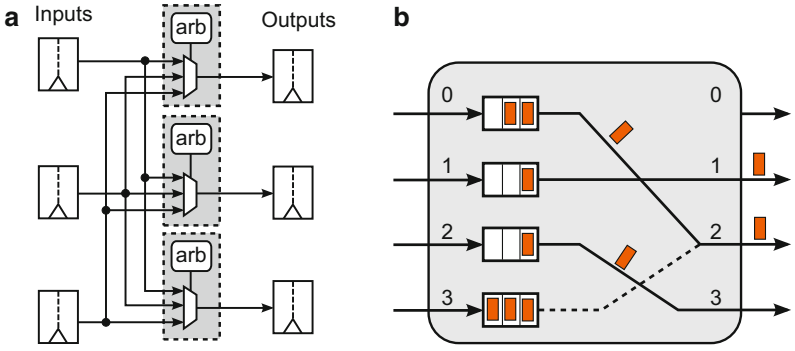

Fig. 3.10 (**a**) The fully unrolled organization of the switching datapath that includes a separate per output arbiter and multiplexer and (**b**) its parallel switching properties that allows different input-output connections to occur concurrently

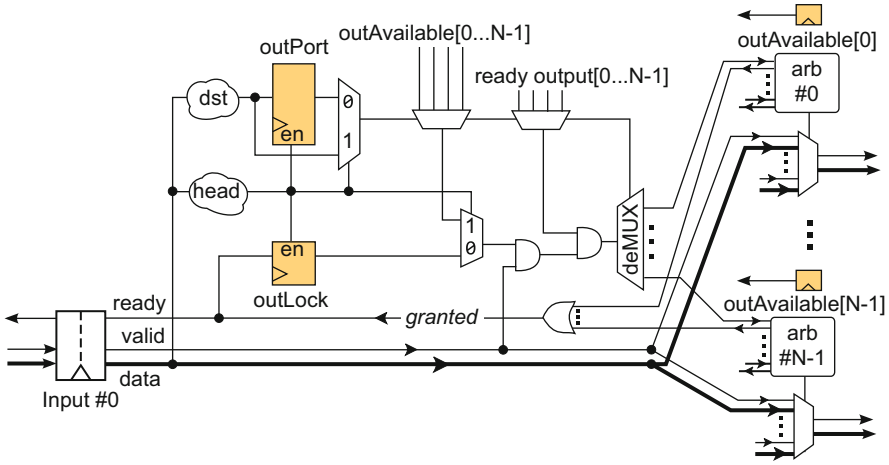

Fig. 3.11 The organization of the request generation and grant handling logic per input port that incorporates also flow control handshake and the necessary input and output state variables

access to *outPort*[*i*]. Recall that *outPort*[*i*] is a N-bit vector following the one hot code. If the jth bit of *outPort*[*i*] is asserted, it means that the packet from the ith input should connect to the jth output port of the switch. Also, each output holds one state variable called *outAvailable*[*j*] (corresponds to the jth output) that denotes if it is free or if it has been allocated to a selected input port. In all cases, the per-input and per-output variables are set by the head flits and released by the tail flits of a packet.

The details of the request generation and grant handling logic attached to each input (or else called the input controller) is shown in Fig. 3.11. Each input receives N *outAvailable* bits (one per output) and N ready signals that declare buffer availability in the specific clock cycle. The flits of the packet select the *outAvailable* and ready signals that correspond to their destined output port. In the case of head flits this information comes directly from the bits of the packet (dst field) while in the case of body and tail flits comes from the stored *outPort*[*i*] variable. For the head flits, the valid bit of each source is masked with the selected availability flag. If the output is available the valid bit will remain active. If the output is taken it will be nullified. This qualified valid bit then should check for buffer availability at the selected output port. Therefore, it is again masked with the selected ready signal to produce the request sent to the output arbiters. If the selected output buffer is available, the corresponding head flit can try to gain access to the selected output port. The masked input requests should be distributed to the appropriate output arbiter. As shown in Fig. 3.11, this is done by an input demultiplexer that transfers the qualified valid bits to the appropriate output. From each input, N request lines connect to the outputs where only one of them is active.

The grants produced by the output arbiters are reshuffled and gathered per input. The OR gate, depicted in Fig. 3.11, merges the grant bits to one grant bit that is

asserted when there is one grant bit equal to one. An asserted grant bit means that the corresponding input has won in arbitration and can move to the selected output. Concurrently the *outLock* bit is set to one and the *outPort* variable is set to the output port pointed by the head flit. The input buffer receives a ready_in signal that causes the head flit to be dequeued and transferred to all output ports. However, only one output multiplexer, driven by its associated arbiter, will select this flit. When the head flit arrives at the output it de-asserts the *outAvailable* flag, showing to the rest inputs that this output port has been allocated and cannot be used by another packet.

As shown in Fig. 3.11, the rest flits of the packet will check first the *outLock*[*i*] bits. If it is set, they will generate a new request using the stored *outPort*[*i*] variable. For them winning arbitration will be easy since they will be the only flits that will ask for the output indexed by *outPort*[*i*]. Of course, their requests (valid signal of the input buffer) are also masked with the selected ready signal of the corresponding output port to guarantee that, when they leave the input, there will be available buffer space to host them at the output. Once the tail flit reaches the output of the switch it re-asserts the local *outAvailable* flag allowing the requesting inputs to participate in arbitration in the next cycles.

Credit-based flow control does not include any more details than the ones presented in Sect. 3.3.1. According to Fig. 3.12, one credit counter is added per output that receives the credit updates from the corresponding output buffer and informs all inputs about the availability of free buffer slots using the ready signal. Also, each input once it dequeues a new flit it sends backward a credit update.

In many cases, the outputs contain a simple pipeline register, instead of an output buffer, that just isolates the intra and inter router timing paths. Under this configuration the design of the switch remains the same. The only difference is that the credit counter reflects the empty slots available at the buffer at the other side of the link. As expected, this configuration increases the round-trip time of the communication between two flow-controlled buffers since the data and the credit

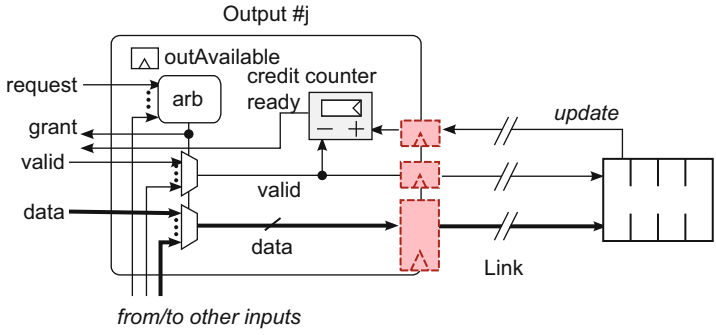

Fig. 3.12 The addition of a credit controller per output that may optionally include additional pipeline registers, allows the connection of the unrolled switching module to multiple independent credit-based flow-controlled links

updates in the forward direction spend one more cycle before reaching an output buffer. Without any further change, by just increasing the buffer size at the output, full throughput communication is guaranteed.

3.5 Head-of-Line Blocking

Assume for example the case of a 3-input and 3-output switch shown in Fig. 3.13. If two inputs request the same output, then contention arises and the arbiter will grant only one of the two competing inputs. The one that won arbitration will pass to the requested output and leave the router in the next clock cycle, provided that a buffer is available downstream. The packet that lost arbitration will be blocked in the input buffer until the tail of the winning packet leaves the router too. In our example, input 2 participated in the arbitration for output 2 and lost. However, besides the two flits heading to output 2, input 2 holds also flits that want to leave from output 1 that is currently idle. Unfortunately, those flits are behind the frontmost position of the buffer at input 2 and are needlessly blocked, as depicted in Fig. 3.13. This phenomenon is called head-of-line blocking and is a major performance limiter for switches. It can be alleviated only by allowing more flexibility at the input buffers that should allow multiple flits to compete in parallel during arbitration even if they don't hold the frontmost position.

A good way to understand HOL blocking is use the example presented in Medhi and Ramasamy (2007): Think of yourself in a car traveling on a single-lane road. You arrive at an intersection where you need to turn right. However, there is a car ahead of you that is not turning and is waiting for the traffic signal to turn green. Even though you are allowed to turn right at the light, you are blocked behind the first car since you cannot pass on a single lane.

The throughput expected per output can be easily estimated if the traffic distribution is known beforehand. Without loss of generality, assume that each input wants to transmit a new flit per cycle, and that each flit is destined to each output with equal probability $P = 1/N$. When more than one inputs are heading for the same output only one will get through and the rest will be blocked. An output j will be idle only when none of the inputs have a flit for this output. The probability that input i chooses output j is $P_{ij} = 1/N$. Thus, the probability of not selecting the corresponding output is $P_{ij} = 1 - 1/N$. An input sends or not to output j

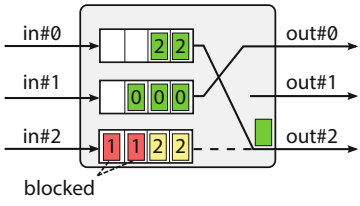

Fig. 3.13 Demonstration of a Head-of-Line Blocking scenario

independently from the rest. Thus, the probability that all N inputs are not sending to output j, thus rendering output j idle, is $\prod P_{ij} = (1-1/N)^N$. Therefore, output j is accepting a new flit with probability $1-(1-1/N)^N$. This value starts from 0.75 for 2×2 switch, moves to 0.703 for a 3×3 switch and converges to 0.63 for large values of N. As proven in Karol et al. (1987), if we take into account that current scheduling decisions are not independent from the previous ones then the maximum throughput per output is lower and saturates around 58 % for large values of N.

3.6 Routers in the Network: Routing Computation

The need to connect many sources to many destinations in a regular manner and without using many wires has led to the design of network topologies. A router is placed at the crossroads of such network topologies as shown in Fig. 3.14 and should forward to the correct output all traffic that arrives at its inputs. Each input/output port of the router that is connected to the network's links should be independently flow controlled providing lossless operation and high communication throughput.

The router should support in parallel all input-output permutations. When only one input requests a specific output, the router should connect the corresponding input with the designated output. When two or more inputs compete for gaining access to the same output in the same cycle the router is responsible for resolving the contention. This means that only one input will gain access to the output port. The flits of the input that lost stay it the input buffer of the current router and retry in the next cycle. Alternatively, the flits of the lost input can be misrouted to the first available output and move to another node of the network, hoping that

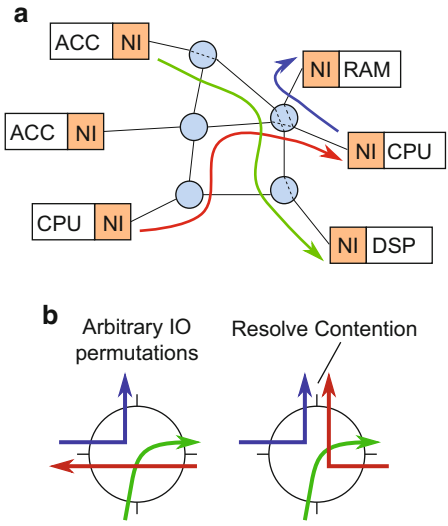

Fig. 3.14 Routers are responsible for keeping the network connected and resolving contention for the same resource while allowing multiple packets to flow in the network concurrently

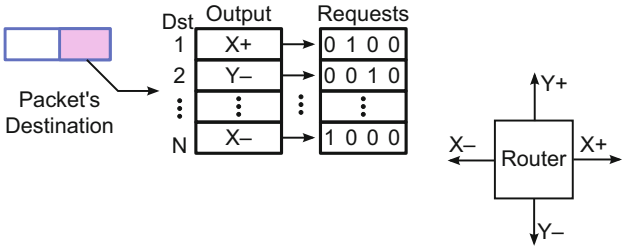

Fig. 3.15 Routing computation at each router is just a translation of the packet's final destination to a local output port request

they will reach from there their destination. Misrouting actually does not resolve contention, but spreads it in space (in the network), while the baseline approach spreads contention in time by allocating one output to one input in each clock cycle (Moscibroda and Mutlu 2009).

Up to now we assumed that the packets arriving at the input of a router knew beforehand their selected output port. In the case of a larger network, each packet will pass through many routers before reaching its final destination. Therefore, a mechanism is required that will inform the packet which output to follow at each intermediate router, in order to get closer to its destination. This mechanism is called routing computation. Routing computation implements the routing algorithm that is a network wide operation, and manages the paths that the packets should follow when travelling in the network. Consequently, each router should respect the properties of the routing algorithm and forward the incoming packets to the appropriate output following the path decided by the routing algorithm (Duato et al. 1997).

For routing computation to work, each packet that travels in the network should provide to the router some form of addressing information. In the case of source routing, the packet knows the exact path to its destination beforehand. The head flit of each incoming packet contains the id of the output port that it is destined to and the router just performs the connection. On the opposite case, when distributed routing is employed, the packet carries at its head flit only the address of the destination node. Selecting the appropriate output is a responsibility of the router that should translate the packet's destination address to a local output port request, as shown in Fig. 3.15. The selection of the appropriate output port is a matter of the routing algorithm that governs the flow of information in the network as a whole but it is implemented in a distributed manner by the routers of the network.

Integrating routing computation logic in the routers is simple. In the simplest case of source routing each packet knows the exact path to its destination and has already stored the turns (output ports) that should follow at each router of the network. In this scenario, the head flit of each packet already holds the request vector needed at each router. Once the head flit reaches the frontmost position of the buffer the corresponding bits of the header are matched directly with the *outPort* wires of

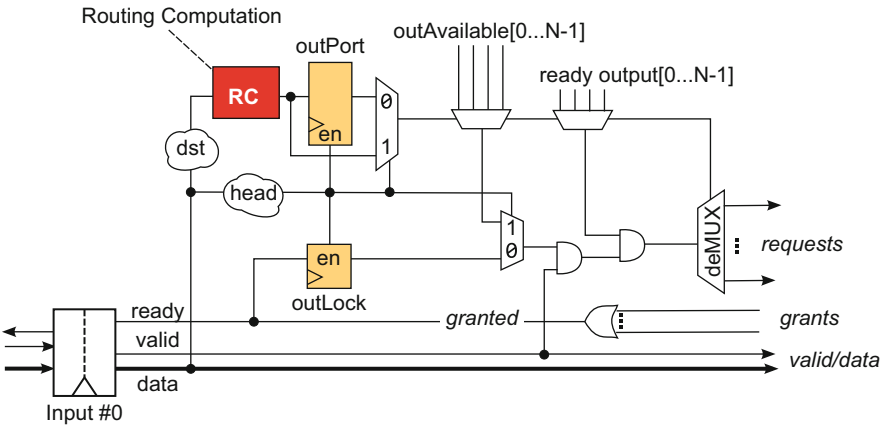

Fig. 3.16 Routing computation selects the output port that each packet should follow according to the packet's destination address. The remaining request generation and grant handling logic remains exactly the same

the request generation logic. The requests used at each router are thrown away by shifting accordingly the bits of the head flit.

In the case of distributed routing the head flit carries only the address of the destination node. Depending on the routing algorithm, each router should translate the destination address to a local output request allowing each packet to move closer to its destination. This translation is an obligation of the routing computation logic. The routing computation logic can be implemented using a simple lookup table or with simple turn-prohibiting logic (Flich and Duato 2008). Routing computation is driven by the destination address of each packet and returns the id of the output port that the packet should use for leaving the current router, as shown in Fig. 3.16. This id will be used for selecting the appropriate output availability flags and ready signals and will be stored to the *outPort* variable of each input controller.

In the case of adaptive routing where each packet is allowed to follow more than one paths to reach its final destination the routing computation logic delivers a set of eligible output ports instead of a single output. Selecting the output port to which the packet can leave the router needs an extra selection step that may take other network-level metrics into account such as the available credits of the eligible outputs or additional congestion notification signals that will be provided outside the routers (Ascia et al. 2008).

3.6.1 Lookahead Routing Computation

Routing computation reads the destination address of the head flit of a packet and translates it to a local *outPort* request following the rules of the network-wide

routing algorithm and the ID of the current router. In this way, request generation, arbitration and multiplexing should wait first RC to complete before being executed. This serial dependency can be removed if the head flit carries the output port request for the current router in parallel to the destination address. Allowing such behavior requires the head flit to compute the output port request before arriving at the current router using lookahead routing computation. Lookahead routing computation (LRC) was first employed in the SGI Spider switch (Galles 1997), and extended to adaptive routing algorithms in Vaidya et al. (1999).

The implementation of LRC can take many forms. In the traditional case without any lookahead in RC, shown in Fig. 3.17, each input port first executes RC and then continues with the rest operations of the router. In this case, the RC unit of input X selects to which outputs, A, B, or C, the arriving packets should move. Then, once the packet arrived to the input of the next router it would repeat RC computation for moving closer to its destination. Instead of implementing RC after a head flit has arrived at the input buffer, we can change the order of execution and implement

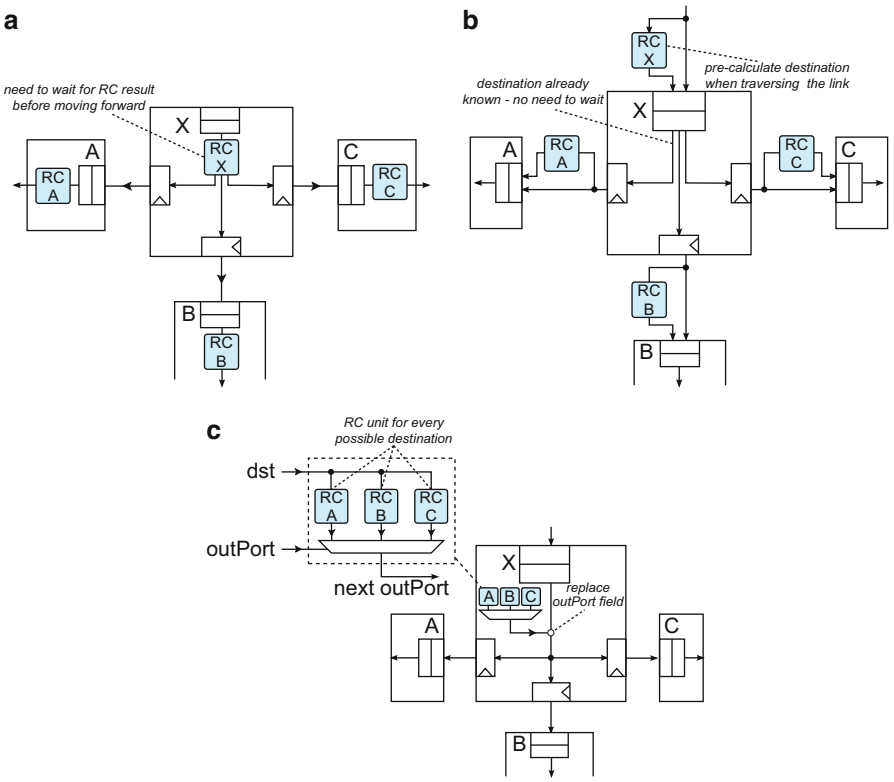

Fig. 3.17 (**a**) Baseline RC placement, (**b**) Lookahead RC in parallel to Link traversal, and (**c**) the implementation of Lookahead RC at each input port that runs in parallel to arbitration and uses multiple routing computation units one for each possible output port

RC in parallel to link traversal. Therefore, at input X the head flit of the packet has already computed RC_X at the link, and presents to the router the pre-computed *outPort* requests. The same happens also for inputs A, B, and C. Once the packet follows the link towards these inputs it computes RC for the next router just before it gets stored to the corresponding input buffer.

Lookahead routing computation can move one step forward and the let RC_A, RC_B and RC_C modules exist at input X instead of the links. When a packet arrives at input X, it has already computed beforehand the output port that should select for arbitration and multiplexing. The output port request vector of a head flit at input X would point to one of the links connecting to the next inputs A, B or C. Therefore, depending on which output the packet of input X is heading to, it should select the result of the appropriate routing computation logic RC_A, RC_B, or RC_C. The organization of the LRC unit at input X is illustrated in Fig. 3.17c. All RC units receive the destination address of the packet and based one the output port request of the packet arriving at input X selects the output port request for the next router. For example, if the incoming packet moves to input A of the next router, the output port requests of RC_A should be selected and attached to the header.

In this way, LRC runs in parallel to the rest tasks of the router, since the *outPort* request vector is ready on the header of the packet. The organization of the request generation logic that relies on LRC is shown in Fig. 3.18. Instead of the RC's result, now the head flits use their own field containing the output port, that was calculated at the previous router. In parallel to request generation, the same field feeds the LRC unit that computes the output port for the next router.

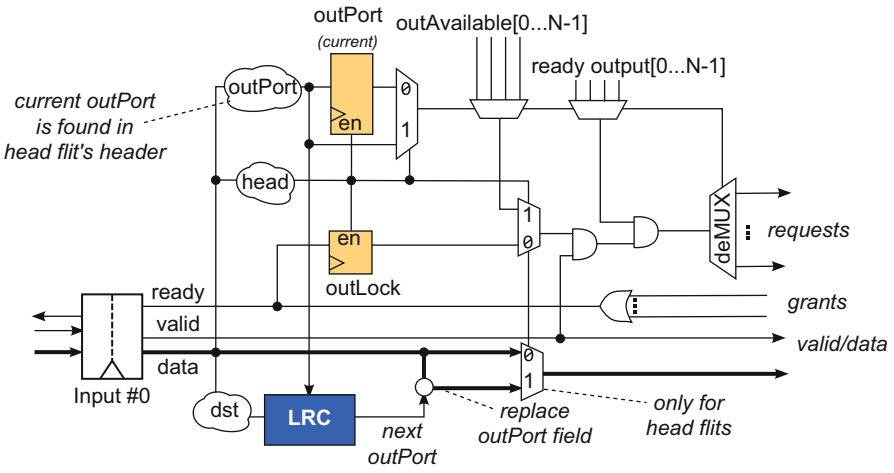

Fig. 3.18 In the traditional RC placement, a flit must wait for the RC result before being able to perform request generation and arbitration. In Lookahead RC, the output port is pre-computed and can be found in the flit's header, thus allowing the tasks of the router to execute in parallel to the LRC for the next router

Please keep in mind that the full implementation of LRC requires the addition of RC computation units at the network interfaces as well, which prepare the output port requests for the routers that are connected at the edges of the network.

3.7 Hierarchical Switching

The operation of the switch described in the previous paragraphs can be easily decomposed to primitive blocks that handle arbitration and multiplexing in a distributed manner. By using the primitive merge units described in Sect. 3.1.3 (see Fig. 3.5) and splitting the data arriving at each input port to the correct output, one can design an arbitrary distributed router architectures (Huan and DeHon 2012; Roca et al. 2012; Balkan et al. 2009; Rahimi et al. 2011). An example is shown in Fig. 3.19, which depicts a router with 4 inputs and 4 outputs. Upon arrival at the input of the router, each packet performs routing computation (RC). Subsequently, depending on buffer availability, output availability, and the allocation steps involved in each merging unit – the flits of the packet are forwarded to the merging unit of the appropriate output. Integration of the merging units is straightforward, since they all operate under the same ready/valid handshake protocol (or credit-based flow control). All router paths from input to output see a pipeline of merging units of $\log_2 N$ stages. Moving to the next router involves one extra cycle on the link; link traversal does not include any merging units and is just a one-to-one connection of elastic buffers.

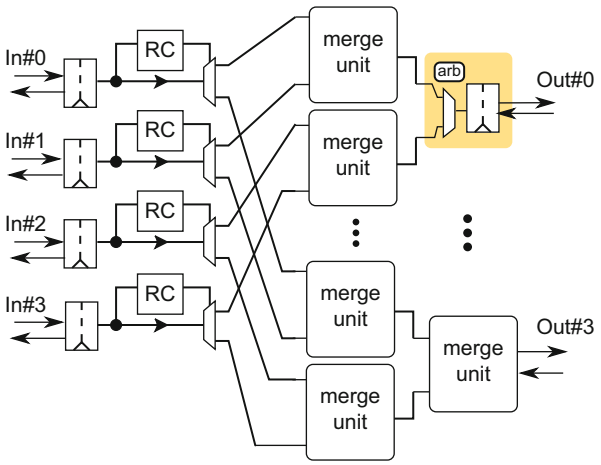

Fig. 3.19 The parallel connection of multiple inputs to multiple outputs can be established using a hierarchical merging tree of smaller switching elements at each output, and a split stage at the inputs that guides incoming packets to their destined output based on the outcome of the routing computation logic

Due to the distributed nature of this architecture, the split connections can be customized to reflect the turns allowed by the routing algorithm. For example, in a 5-port router for a 2D mesh employing XY dimensioned-ordered routing, splitting from the Y+ input to the X+ output is not necessary since this turn is prohibited. Several other deterministic and partially-adaptive routing algorithms can be defined via turn prohibits (Flich et al. 2007; Flich and Duato 2008). When this customization is utilized, significant area savings are expected, due to the removal of both buffering and logic resources. This modular router construction enables packet flow to be pipelined in a fine-grained manner, implementing all necessary steps of buffering, port allocation, and multiplexing in a distributed way inside each merging unit, or across merging units. Also, the placement of merge units does not need to follow the floor-plan of the chosen NoC topology. Instead, merge units can be freely placed in space, provided that they are appropriately connected.

3.8 Take-Away Points

Switching packets of flits from many inputs to one or multiple outputs is a combined operation that merges link-level flow control with arbitration in order to resolve contention for the same output and guaranteeing that there are available buffer slots to host the selected flits. The implementation of flow control and arbitration requires the addition of per-input and per-output state variables that guide all the intermediate steps that a packet should complete before being able to move to the selected output port. The addition of a routing computation module transforms a switch to a network router that can participate in an arbitrary network topology allowing incoming packets to find their path towards their destination.