4

The Octopus switch

This chapter[1] discusses the interconnection architecture of the Mobile Digital Companion. The approach to build a low-power handheld multimedia computer presented here is to have autonomous, reconfigurable modules such as network, video and audio devices, interconnected by a switch rather than by a bus, and to offload as much as work as possible from the CPU to programmable modules placed in the data streams. Thus, communication between components is not broadcast over a bus but delivered exactly where it is needed, work is carried out where the data passes through, bypassing the memory. The amount of buffering is minimised, and if it is required at all, it is placed right on the data path, where it is needed.

A reconfigurable internal communication network switch called Octopus exploits locality of reference and eliminates wasteful data copies. The switch is implemented as a simplified ATM switch and provides Quality of Service guarantees and enough bandwidth for multimedia applications. We have built a testbed of the architecture, of which we will present performance and energy consumption characteristics.

4.1 Introduction

The interconnection structure of the *Mobile Digital Companion* is based on a switch, called *Octopus*, which interconnects a general-purpose processor, (multimedia) devices, and a wireless network interface. Although not uniquely aimed at the desk-area, our work is related to other projects (like [1][5] and [9]), in which the traditional workstation bus is replaced by a high speed network in order to eliminate the communication bottleneck that exists in current systems.

---

[1] Major parts of this chapter were presented at the fifth annual *ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99)* 1999 [8], and at the *ProRISC'99 workshop on Circuits, Systems and Signal Processing*, 1999 [9].

The exact implementation of the interconnect is not a vital issue in the architecture of the *Mobile Digital Companion*. Just as rings, crossbars and busses have all been used in ATM switches [3], so may they be used in the Companion. It is the connection-oriented approach, using fixed sized cells and asynchronous multiplexing as key factors. In most computer systems a single shared link or a bus interconnects the components. The main attraction of a bus is its low cost and low complexity. Drawbacks are high energy consumption, limited extensibility and limited scalability (that is, the consequences of extending busses are high: increased complexity and high cost in many aspects). Due to the electrical properties of busses these limitations become more evident at high speed. Although an interconnect based on a switch cannot compete in terms of cost, it is attractive because of its high aggregate bandwidth, scalability, and low energy consumption. As in switching networks, the use of a multi-path topology will enable parallel data flows between different pairs of modules and thus will increase performance.

The *n*-by-*n* switch interconnects *n* modules and provides a reliable path for communication between modules. Addressing is based on connections rather than memory addresses (see Chapter 3). This not only eliminates the need to transfer a large number of address bits per access, it also gives the system the possibility to control the QoS of a task down to the communication infrastructure. This is an important requirement since in a QoS architecture all system components, hardware as well as software, have to be covered end-to-end along the way from the source to the destination. In our infrastructure all connections are associated with a certain QoS.

In an ideal model all modules can communicate with each other over a communication channel of zero length and infinite speed. In our prototype the internal bandwidth will be much more than the maximal throughput of a device (e.g. the wireless interface is capable of transferring 2 Mb/s, and the interconnect has a capacity of 32 Mb/s per connection). By having such high bandwidths on the local interconnect, devices can access other devices (including the network) in much the same way as if the device had exclusive access. The bandwidth reservation on the wireless network, which will probably be the bottleneck in this architecture, can in this way be though of as extending into the mobile, all the way to the device. Since we are using ATM cells not only as basic communication mechanism on the network, but also internally in the architecture of the mobile, we do not need to have any packet conversion as well.

Switching theory

One of the main problems of network design is how to ensure sufficient bandwidth, and thus throughput, for all data streams [13]. A network may be blocking, which means that certain connections cannot be made, because of other connections created earlier. Basically there are two types of switches: *time division* (T) switches and *space division* (S) switches. In an S-switch, physical switches are used to connect input wires to output wires. Physical connections are thus created between input and output channels. In a T-switch, a single physical line is used to transport the different connections. Time is divided into periodic cycles, where each cycle consists of a fixed number of *time slots*. A

time slot is a periodically recurring time interval consisting of a fixed number of clock cycles. Each time slot represents a different channel.

A well-known three stage switching network is the TST network [13]. The first and third stage of this network consists of T-switches, whereas the second stage consists of a *time-shared* S-switch. In a time-shared S-switch new physical connections between input and output channels are created for each time slot.

The communication network is built according to the TST network in which the Octopus switch is a time-shared S-switch, and the modules can function as the T-switch. Such a configuration has important advantages over other configurations when used in processor networks [13]. Because a T-switch has intrinsic buffering, the concept of FIFO buffering to allow synchronisation between modules operating at different data rates is readily implemented in such a switch. The data that is to be produced or consumed by a module can be buffered until sufficient data is available and a time slot is granted to the connection.

Outline

Section 4.2 provides the architecture and design of the interconnection network, the Octopus switch. The switch provides the communication infrastructure between functional modules, and is based on connections of two service classes. Topics of special interest are the buffer organisation, the scheduling techniques used, and the internal communication protocol. In Section 4.3 the prototype implementation of the Octopus switch is described, and performance and energy consumption measurements are presented. Finally, we present the summary and conclusions in Section 4.4.

4.2 Architecture of the Octopus switch

In this section we will present the architecture of the Octopus switch. The key goals motivating the design has been simplicity, flexibility and energy efficiency.

4.2.1 Octopus architecture

The Octopus switch provides the interconnection infrastructure between the functional modules in the system of a Mobile Digital Companion. Figure 1 shows an architectural view of the system of a Mobile Digital Companion.

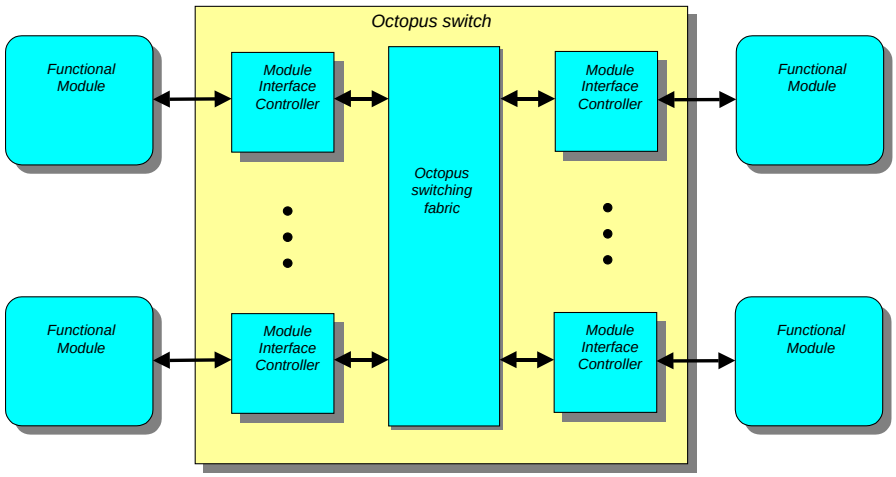

Figure 1: System architecture Mobile Digital Companion.

Around the Octopus switch there are several *Functional Modules* (like network module and video module). Inside the Octopus switch we have the *Module Interface Controllers* and the *Octopus switching fabric*.

In the communication we have three basic protocol layers: the *module layer* that connects the functional modules, the module interface layer, that interconnects the Module Interface Controllers, and the *physical layer* that is performed by the Octopus switching fabric. Figure 2 gives a schematic overview of the interconnection protocol layers.

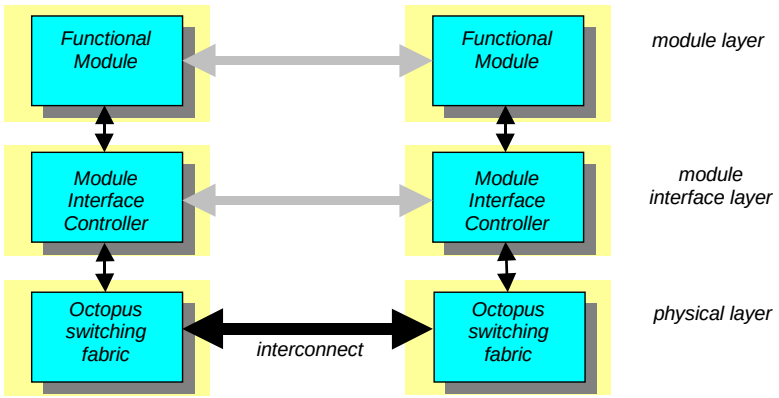

Figure 2: Interconnection protocol layers.

The grey arrows represent a connection at protocol level, and the black arrows represent physical connections. Flow control is done end-to-end rather than link-by-link, that is,

only the modules that send data and not the switch is responsible for controlling the flow of data.

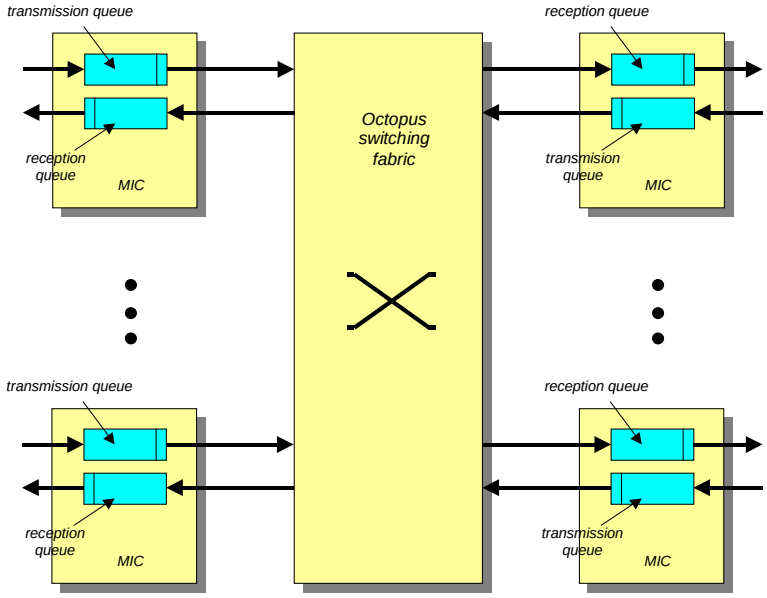

Figure 3: Basic architecture Octopus switch.

A high level schematic of the Octopus architecture is shown in Figure 3. At the heart of this architecture is the *Octopus switching fabric*. The fabric connects eight *Module Interface Controllers* (MIC) that interface to the attached modules. These MICs decouple the modules from the timing of the Octopus switch and the other modules. Each module can communicate with the others, so the switch is an 8-by-8 switch. The MICs contain small transmission and reception FIFOs that store ATM cells. The MICs further perform operations like connection setup and arbitration for the connections between the modules.

4.2.2 Packet size

The amount of data that is transported within a single time slot between functional modules is called a *packet*. Several factors determine a suitable packet size for the communication network.

- Firstly, as we want to minimise the buffering in the system, the packet size should be chosen as small as possible. However, the locality of reference principle suggests that we should preserve a sufficient amount of data correlation between the data items, so that the packet size should be sufficiently large. If we would have a too small packet size, of say one byte, then data items of different streams are sent alternating over the same physical link, thereby removing almost all data

correlation. This usually results in a higher transition activity, which causes higher energy consumption.

- Secondly, the minimum packet size is further determined by the overhead in transmitting a packet over the network. This overhead includes connection setup, arbitration of the connection network, error control and possibly also flow control. A small overhead (i.e. fast execution of the internal arbitration and protocol) allows for a smaller packet size, but it more expensive in area and energy consumption than a slower arbitration and protocol.
- Another aspect that determines the packet size is the amount of bits that can be transported over the network in parallel. A fully serial network is efficient in terms of the amount of wiring needed, and can therefore simplify layout. In contrast, a fully parallel implementation is more costly in terms of wiring and complicated layout, but can be very fast.
- Finally, since in a mobile computer the external (wireless) network places an important and prominent role, the packet size over the external network also determines the internal packet size. If the internal packet size were a multiple of the packet size that is transported over the external network then the interfacing would be practical, simple and efficient.

Asynchronous Transfer Mode (ATM) is used in communication systems. The ATM scheme is an advanced version of packet switching: small fixed-size cells are switched at high speeds. We will now only describe briefly the packet organisation of B-ISDN ATM, for a more comprehensive study on ATM we refer to [13]. A cell consists of an information field of 48 bytes and a 5 byte header. The primary function of the header is to identify cells that belong to a connection called the virtual channel, identified by a Virtual Channel Identifier (VCI). The VCI is used for routing ATM cells in a switching fabric. In addition ATM supports virtual paths, identified by Virtual Path Identifiers (VPI). A virtual path is a group of virtual channels. Further information contained in the header is the Payload Type, Cell Loss Priority, and a Header Error Control. The information field is transported transparently by the ATM layer, without any error detection or correction.

In the architecture we have chosen to adopt the structure of an ATM cell, i.e. a cell header and a small payload. This format has shown to be sensible for several reasons, among a practical one is that it allows for a simple connection to the ATM network that we are using. The size is small enough to allow for a fast and flexible scheduling of communication streams, and small enough to have a relatively small overhead. Within the system, the subdivision between virtual paths and virtual channels is not significant, and so the term circuit identifier is used to mean the whole of the header. The 5-byte header seems a bit overdone, but a practical implementation can easily implement a simple header compression mechanism that uses just a one-byte VCI header. A one-byte VCI header allows for 256 simultaneous virtual connections, which we expect to be large enough for our applications and system in a portable computer. Another format of the cell might be more efficient, e.g. a payload of 64 bytes and a header of 1 byte will

result in less overhead. The current testbed uses the B-ISDN ATM cell format, but future implementations might use a more efficient cell structure.

4.2.3 Buffer organisation

A problem that can occur on any communication network is blocking. Blocking results from sharing common resources such as common links or buffers. An important aspect of the architecture of a communication system is therefore the buffer organisation. To obtain a non-blocking communication network, sufficient bandwidth should be available on each communication path through the connection network.

A number of techniques can be applied to decrease the impact of blocking and to increase the throughput [15].

- First the system can provide an adequate buffering at the source, this is called *input buffering*.
- Secondly, it can try to buffer the traffic at the destination module, this is called *output buffering*.
- Finally, buffering can be provided *inside the Octopus switch*, using input buffering, output buffering, a congestion control mechanism, or a combination of these.

It is well known that output buffering yields much better performance than input buffering since cells can only be delayed when the bandwidth of the output link is saturated and never due to internal contention. Simulations showed that pure input buffering has a throughput of less than 60% compared to pure output buffering under a uniform workload [12]. This inferior performance is mainly due to head-of-line (HOL) blocking. HOL blocking occurs with FIFO queuing when a message at the head of an input queue is blocked, all messages behind it in the queue are prevented from being transmitted, even when the output link they need is idle. Link utilisation of an input buffered switch can be improved if the buffered cells can be randomly accessed rather than in FIFO order. This approach, however, requires a more complex buffer management and scheduling of the switching fabric [15].

When comparing the cost of an implementation, output buffering is more expensive than input buffering, in particular for large switches. For an output buffered n by n switch, the bandwidth b of the switching fabric and attached buffer memories grows as $O(n^2)$ since every output queue must be able to simultaneously receive data from n input ports. In contrast, the bandwidth of the fabric and the buffer memories for input buffering grows as $O(n)$ since an output port can only receive data from one input queue at a time.

The disadvantage of output buffering is thus the cost of the memory and the interconnection bandwidth. To overcome the congestion that can occur with input buffering, a scheduler can use a *congestion control mechanism* that (statically and/or dynamically) schedules the traffic in the switch.

Minimal buffering – The amount of buffering required in the system depends strongly on the traffic characteristics of a connection. It is, however, not always easy to know on beforehand the exact traffic characteristics. For example, video and audio data streams

are often bursty. Therefore the system must be able absorb large bursts and buffer the data during the scheduling latency interval. In the design of the Octopus architecture we have tried to minimise the amount of buffering that is required. Having no, or minimal buffering, can result in several advantages for both performance and energy consumption:

- *High performance* – Because the data is copied with no or just minimal buffering from the source module directly to the destination in the sink-module, the latency is reduced.
- *Energy efficient* – There is less energy needed for the storage of the data, and, if buffering can be omitted, also for the transfer to and from the buffers,
- *Simple flow control* – If buffering is used, then, in order to prevent overflow of the buffers, these buffers must be either very large or there must be a good and fast data flow mechanism. Large buffers require extra area and energy, which is wasted because most of the time they will not be used. Flow control induces also extra energy consumption and extra area because it requires extra communication and makes the design more complex.
- *Predictable QoS* – Having a large buffer also influences the Quality of Service because it increases the latency and jitter of the data packets between the modules.

In our model, the buffering of the data should thus be avoided as much as possible. Reduction in the required amount of buffering on the mobile can be achieved in several ways. Firstly, the applications that source the traffic can try to adapt the traffic rate to the rate the sink module can handle. For example in the case of a communication stream between a video camera and a display, the camera could lower its frame rate or use a smaller picture size. Secondly, the modules could try to adapt their implementation such that the system can handle the traffic rate. For example, the video source could use another coding mechanism. Thirdly, if the communication stream is between a mobile and an application or service that is running on a system with plenty of energy (in general on a wired network), then the required buffering could be migrated from the mobile to the fixed station. This implies that the data must be absorbed and processed as quickly as possible. Note that the same techniques can be used if the communication bandwidth is a bottleneck.

Octopus buffer organisation – In our model, the interconnection network is transparent and provides only a direct connection between two functional modules. The Octopus switch transmit and receive ports are simple, containing only minimal buffering and arbitration functionality. The buffering allows ATM cells to be read and written at a rate, which is independent of the functional modules. Each of the transmit and receive ports operates independently. This allows fast and slow modules alike to have a simple interface and see the interconnection network as a place to write and read ATM cells. Logically, the implementation of the Octopus switch represents a crossbar switch with both input and output buffering.

A buffering system can not be generic for all modules. Each individual module should use the buffering system that is dedicated to the traffic characteristics of that module. If the Octopus switch should provide an adequate buffering system and data-flow

mechanism that is capable of handling the buffering requirements of the modules, then the buffering system would become static, and designed for worst-case traffic characteristics that hardly occurs. It would need a large buffer capacity in the switch and an adequate flow control to prevent overflow of the buffers. However, in general most modules do not need such buffering system, and the introduced extra complexity and area requirements would be wasted.

In the Octopus architecture buffering can be performed at three logical units: the switching fabric, the MIC, and the modules. The most significant buffering is located in the functional modules, and with just a minimal buffering in the Module Interface Controllers. The switching fabric does not contain buffers.

- *Octopus switching fabric buffering* – We have omitted buffering of ATM cells in the switching fabric for the reasons mentioned before. Buffering in the fabric is only needed for synchronisation of a connection stream between two MICs.
- *Module Interface Controllers buffering* – These units provide a minimal amount of buffering, just enough to decouple the timing of the attached modules from each other and to buffer the data during the scheduling latency interval. Therefore, the rate at which the functional modules can handle traffic must thus be lower than the capacity of the MICs and the interconnection network.
- *Functional modules buffering* – This is the place where the most significant buffering can occur. Because the raw interconnect data rate of the Octopus switch is much higher than the rate of which the modules will probably handle the traffic, the main bottleneck will thus be not in the switch, but at the end-points of a connection between the modules. The responsibility of the buffer organisation is thus transferred to the module.

In some cases, like in traffic that uses the wireless communication network interface, buffering can be advantageous and reduce the energy consumption. In such cases the energy consumption required to buffer the data is lower than the energy savings that are achieved. However, this is only possible up to a certain limit due to buffer space limitations and Quality of Service requirements because for example the latency is increased [7].

4.2.4 Octopus switching fabric architecture

The Octopus switching fabric behaves like an 8-by-8 ATM switch. The switching provides a simple mechanism for the exchange of cells, regardless of their payload. The Octopus switch simply routes the traffic according to (a part of) the Virtual Channel Identifier (VCI) in the header. In contrast to full-blown ATM switching fabrics, the responsibility for ATM functions, such as VCI mapping and flow control, has been teased out of the switch fabric and assigned to the devices that plug into the switching fabric. The MICs are responsible for translating the VCI to the address of the destination module, and – when a connection has to be established – initialises the switching fabric with that address.

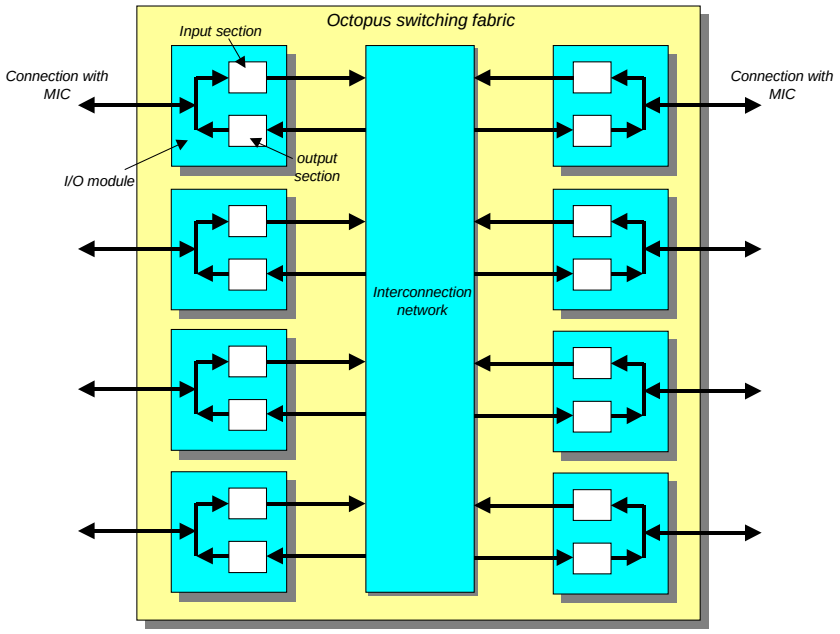

Figure 4: Architecture of the Octopus switching fabric.

The architecture of the switching fabric is therefore basically simple; most of the complexities are migrated to the Module Interface Controllers. As depicted in Figure 4, a switch consists of the following units:

- 8 input sections,
- 8 output sections, and
- an 8 x 8 interconnection structure.

A MIC is connected to an I/O module consisting of an input section and an output section. The connection between the MIC and an input and output section is shared, so the Octopus switch can support half-duplex connections only.

The interconnection network can be implemented in several ways as long as its capacity is large enough to support at least the maximum of four simultaneous connections at a rate that exceeds the data rate of an external module. In our prototype we use a fully connected crossbar. This is an energy efficient, simple and high performance architecture suitable for implementation in a Xilinx gate array. Although such a crossbar needs a large interconnection structure, the data rate on the individual connections can be relatively low.

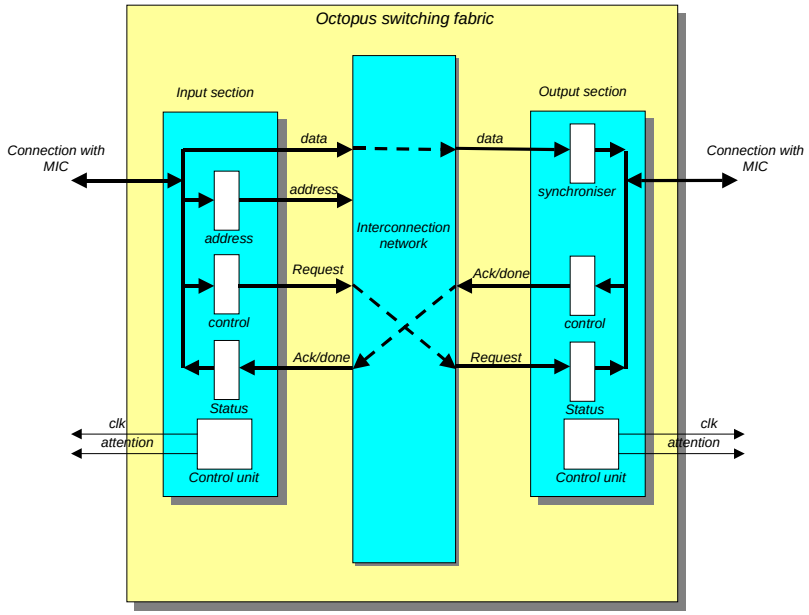

Figure 5: Structure of one input and one output section of the Octopus switch.

The input section basically only consists of three registers: an *address register* that determines the output section of the connection, a *control register* for energy management and general control, and a *status register*. The dataflow of a connection passes the input section transparently. The control register determines when a request for a connection will be made. The status register will contain status information about the current connection.

The output section contains two registers (*control register* and *status register*) and a *synchroniser* that synchronises the datastream to the timing of the receiver MIC. The status register is in fact shared with the input section. All requests for an output section are stored in this register and can be read by the MIC.

Both the input and the output section of an I/O module share the *control unit*. This unit generates the *clock signal* for the attached MIC, and generates an *attention signal* when the MIC has to do an operation. The attention signal can be used to wake-up the MIC from sleep mode.

Figure 5 shows one input section and one output section that are involved in one connection between two modules. The interconnection network uses the address that is stored in the address register of the input section to select the output section it wants to make a connection to.

4.2.5 Module Interface Controller architecture

The Module Interface Controller operates at the module interface layer. Its main task is to provide a data-path between the modules. The MIC basically contains the following units:

- A *transmission queue* that stores ATM cells originating from the module in transit to the switching fabric.
- A *reception queue* receives the ATM cells coming from the switching fabric
- A *VCI mapping table* determines the destination module, and is indexed by the VCI in the header of the ATM cell.
- An *arbiter* performs the actual establishment of a connection and performs scheduling in case multiple simultaneous requests are received via the switch.

Figure 6 shows the basic architecture of a Module Interface Controller with these basic units.

We will briefly describe the basic functions of these units using a typical communication stream between two MICs.

When a MIC receives an ATM cell from the module it is attached to, it will first store this ATM cell in its *transmission queue*. The output port to which the cell must be forwarded is determined by looking up in the VCI mapping table. The VCI mapping table contains the destination MIC of all previously announced virtual connections. The connection identifier contained in the VCI header of the ATM cell indexes the VCI mapping table. The MIC will then establish a connection with this destination MIC. Cells with a VCI that is not known will be forwarded to a default module, which in general will be the CPU-module. The CPU-module contains the *connection manager* (*ConMng*) that is responsible for the management of all connections in the system. The ConMng uses special *management cells* to communicate with the MICs and the modules in the system. The VCI mapping table will be initialised by the ConMng using these management cells.

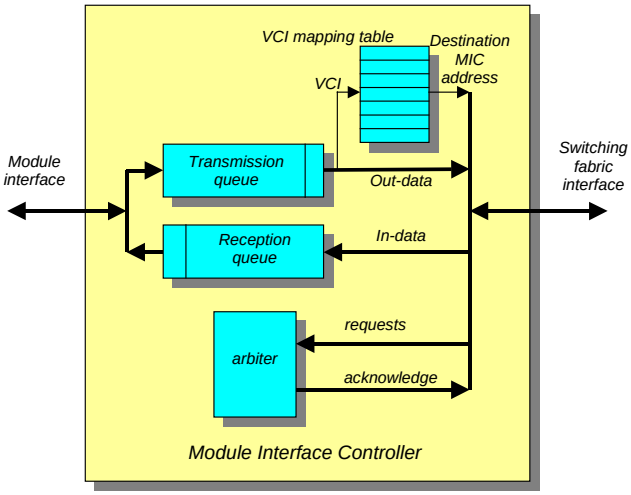

Figure 6: Module Interface Controller architecture.

When the destination MIC receives a request for a connection, the *arbiter* determines when and whether the connection can be established. If there are multiple simultaneous connection requests, it uses a scheduling algorithm to determine which request can be honoured first.

When the connection is established the source MIC forwards the data from its transmission queue over the Octopus switching fabric to the destination MIC. The destination MIC then first stores the ATM cells in its *reception queue* before it is further forwarded to the attached module. Note that the buffering of cells in the transmission and reception queue is not always required: when the module is capable to handle the cells at the required rate, then the buffering can be omitted.

4.2.6 Connections

All communication between modules are based on connections. There is a central Connection Manager (ConMng) that can be used to schedule traffic between the modules. All connections are uni-directional.

Connection types

There are basically two types of connections: *ad-hoc connections* that have no reservations made in the system, and *guaranteed connections* in which the flow through the switch is guaranteed by the connection manager (ConMng) located in the main CPU-module.

- *Ad-hoc connections* – Modules that need no guaranteed flow of data use the ad-hoc connections. These type of connections have no real-time guarantees, and a higher protocol is needed e.g. for flow control. Ad-hoc connections are also used during

the connection setup phase in which the module establishes a guaranteed connection with a different module.

- *Guaranteed connections* – Guaranteed connections are used to transfer data between modules that require bandwidth guarantees between the modules. Once a guaranteed connection is established, the actual data transfer still has to be announced by the source. In this way, the destination MIC can determine whether the reserved bandwidth is actually used, and otherwise assign that bandwidth to other connections.

Since the Octopus switch does not provide flow control between the modules, the modules either have to use some flow control mechanism (*end-to-end flow control*), or must be capable to source and sink the data at minimal the rate they both agreed upon. This is in line with our philosophy to have minimal buffering.

There is no flow control mechanism in the interconnection layer between two functional modules. Flow control mechanisms are generally used to control the stream of data between buffers. Our flow control can be much simpler than the ones required in for example ATM networks. In these systems, flow control is normally performed on a link-by-link basis. For guaranteed connections we do not need to have a flow control at the switch layer. End-to-end flow control suffices because

- guaranteed connections have been assigned a certain amount of bandwidth in the Octopus switch. The arbiter in the Octopus switch uses a distributed arbitration mechanism in which ad-hoc connections have a lower priority than guaranteed connections,
- the modules can be trusted in the sense that they never exceed their bandwidth allocation,
- the interconnect diameter (which is the maximum distance over all the nodes of a network) is small (internal in the system just one hop), and
- the Octopus interconnect provides a large bandwidth suitable for many simultaneous communication streams (the testbed it can support three simultaneous connections of 32 Mb/s, see Section 4.3.3).

All these reasons make that congestion internally in the switching fabric is not likely to occur.

Connection management

Prior to a communication transfer between two modules, a connection has to be set up. Individual MICs in the Octopus switch can be remotely configured by using special control cells. The operating system running on the CPU-module issues these control-cells. Once a connection has been setup, control can be taken over by a local processor that is responsible for controlling both the MIC as well as the locally connected device.

During *connection setup*, the source module contacts the *connection manager* (*ConMng*) that is located in the CPU-module that it requires a channel to a module with a certain amount of bandwidth. It therefore transmits an ATM cell containing the request to the

CPU module. Since no connection with this module has been established yet, it establishes an ad-hoc connection with the CPU-module. Since such a connection has no guaranteed bandwidth the connection request might take some time before it can be acknowledged. The CPU module will upon receiving the request determine whether there is enough bandwidth available between both modules involved and the connection can thus be established. The ConMng then might negotiate with the destination module to verify that this module is capable and prepared to connect. If the connection is not possible (because either the ConMng knows that there is not enough bandwidth in the Octopus switch, or the destination module cannot accept the connection), then the ConMng will reply to the requesting module that the connection is currently not possible. If the connection is possible, then it will inform the destination module and the source module that it a new connection has been set-up.

The aggregate bandwidth available inside the Octopus switch allows each module to communicate at a rate that is probably much higher than it can source or sink. The Octopus switch allows up to four parallel connections between eight modules. Given the small number of modules and the rate at which our present modules are able to generate traffic, this provides enough bandwidth.

4.2.7 Scheduling

In principle each MIC needs to be able to buffer just two ATM cells, both for receive traffic as for transmit traffic. The main philosophy is that the Octopus switch buffers as little as possible, and that the modules either have to provide the buffer capacity, or that they can adapt their traffic rate.

The input ports of the MICs can receive multiple simultaneous requests from all other MICs. A scheduling policy has to be applied to choose which input port will be acknowledged, and can forward an ATM cell to the module. The problem of scheduling in switches has been studied extensively in the design of ATM switching fabrics. Although the design of our internal ATM switch is much more limited, many of the peculiarities in full-blown ATM fabrics apply to the Octopus switch.

Basically there are two types of scheduling: static and dynamic scheduling [15].

- In *static scheduling* the bandwidth allocation is made on the basis of time slots in a service cycle. A *service cycle* is a periodically recurring time interval consisting of *s* time slots. In static scheduling each time a new request for a connection is made, the scheduler tries to allocate bandwidth on the path between source and destination by reserving resources for a number of timeslots. A *communication scheduling table* describes which connections can communicate at each time slot. When a new reservation is made, the scheduling table has to be updated, and possibly rearrange the previous schedules. The Slepian-Duguid theorem [11] states that a schedule can be found for any traffic pattern, as long as the number of cells for any input or any output is no more than the number of time slots in a service cycle. Computing the new schedule may require the number of slots in a service cycle *s* times switch size *n* (thus *s* . *n*) number of steps for an *n* by *n* switch [2][15]. The main advantage of static scheduling is that it has a bounded latency and can therefore be used for real-

time multimedia traffic. Disadvantages are that it requires a centralised scheduler, and that for each new connection, the scheduling table has to be adapted. Furthermore, some bandwidth can be wasted when a reserved slot is not used. This, however, can be overcome with some special precautions.

- In *dynamic scheduling* the arbitration is more dynamic, and uses a scheduling mechanism to choose which cell to forward on the output link. Common arbitration schemes, such as round robin scheduling or priority based scheduling, are suitable. Which arbitration scheme is best for a specific set of applications depends on the characteristics of the tasks and on the cost of implementation of the arbitration scheme. Examples of scheduling mechanisms for switches are proposals of Hui [11] and parallel iterative matching [2]. A problem associated with dynamic scheduling is fairness because the scheduling is distributed at all the output ports. Dynamic scheduling is thus not really suitable for real-time traffic.

In the prototype of the Octopus switch we use a combination of both scheduling techniques: *static scheduling* for guaranteed connections, and *dynamic scheduling* for ad-hoc connections. A certain portion of the bandwidth is allocated to traffic of guaranteed connections, and any unused slot can be used for other traffic.

Static scheduling – The bandwidth allocation for the guaranteed connections is made on the basis of time slots in a service cycle. Each time a new guaranteed connection is required, the source MIC issues a connection request to the Connection Manager. The Connection Manager tries to allocate bandwidth on the path between source and destination by reserving resources for a number of timeslots. Given the communication network and a collection of connections, the question is in which time slots a certain connection may use the network, such that its throughput requirements are satisfied and no conflicts with other connections can occur. The *communication scheduling table* that describes which source MIC can communicate at each time slot is distributed to all MICs in the system using a management cell. Figure 7 shows an example of a communication scheduling table. The rows of the table represent the *s* time slots of a service cycle. Furthermore, the columns defines the source MIC, and each cell in the table defines the destination MIC. Cells that are not assigned a destination MIC represent time slots that can be used for ad-hoc connections.

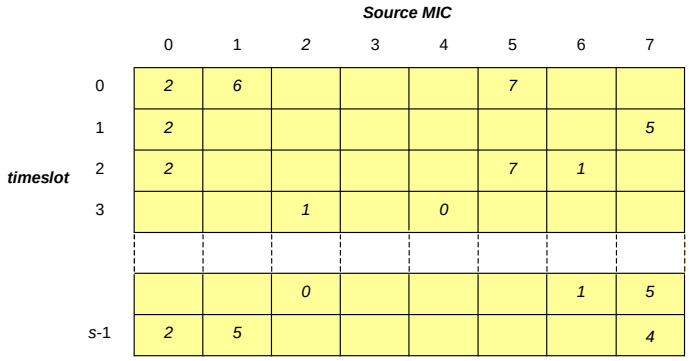

Figure 7: Communication scheduling table example.

Note that in the Octopus switch no distinction is made between connections from one MIC. A guaranteed connection request is issued by the source MIC as if it requires more time slots. When a MIC has multiple connections, it is free to use any slots that it is assigned to, even if they were originally not for that connection. In this way, the source MIC can easily adapt to small fluctuations in the connection streams.

When the guaranteed connection is established, the connection has reserved some bandwidth but these slots are not yet used for traffic. When the source has traffic on the guaranteed connection, it still needs to issue a request to the destination. The arbiter in the destination MIC will always honour this request. When the guaranteed connection has no traffic and does not use the slot, other connections may use the slot instead.

Dynamic scheduling – Dynamic scheduling is much more flexible and the traffic can easily be adapted to the current needs. In the prototype the scheduler of the arbiter in the MICs are based on a round-robin scheduling based on the source MIC (and thus not per connection). Fairness is not a big issue because most connections will be reserved in advance using the static scheduling. These guaranteed connections have a high priority. Ad-hoc connections, however, have a lower priority. We use two separate schedulers for both types of connections. Slots that are reserved for a guaranteed connection, but that are not used at some moment, can be used for other connections, both guaranteed connections as ad-hoc connections. In this way no bandwidth is wasted. Only when no requests are made for traffic on a guaranteed connection, the scheduler for the ad-hoc connections becomes active.

Although ad-hoc connections have no strict timing requirements, some bandwidth needs to be reserved in order to allow new connections to be established and also to allow progress in the processing of these connections. The Connection Manager can satisfy this requirement by ensuring that over a specified time interval sufficient time slots are available for ad-hoc connections. This time interval may be relatively large (e.g. span a large number of service cycles) because there will be sufficient time slots that remain unused. We can safely say this because the aggregate bandwidth is designed to be sufficient, and guaranteed connections are not always active since the bandwidth required by real-time connections may vary over time. During the time that no

bandwidth is required by such a connection, bandwidth can be used by other connections.

Note that traffic between two modules may consist of both a guaranteed connection, as an ad-hoc connection. For example, a video connection requires a guaranteed throughput to maintain some QoS, and more bandwidth only to improve the quality for which it uses the ad-hoc connection.

4.2.8 The stages of the internal communication protocol

During the operation of the switch, several phases are executed as an ATM cell is transmitted from one module to another. We identify the following phases: wake-up phase, arbitration phase, data phase, and release phase. Note that the basic transfer size is one ATM cell, but several cells can be grouped in a frame.

In the following figures the units and signals that are not used at a certain stage during the protocol are not shown. Units that are idle are coloured white, and fifos that are empty are also white.

Sleep phase – in the sleep phase most units are in a low-power mode, receiving no clock. Only a small part of the Octopus switch is active, waiting for an external event from a functional module indicating that it needs to communicate.

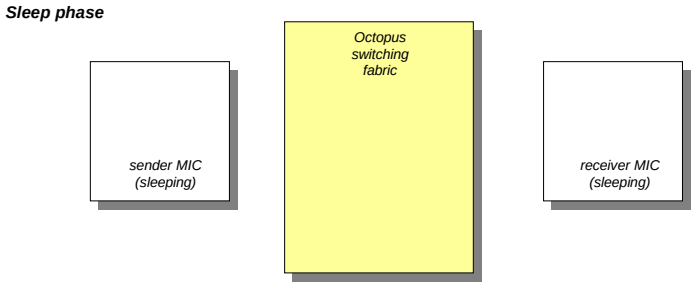

Figure 8: Sleep phase.

Wake-up phase – the first operational phase is the wake-up phase. Normally, i.e. when the Module Interface Controller has no communication to handle, the MIC is sleeping and does not receive a clock from the Octopus switch. If a module needs to transmit data to another module, it notifies the Octopus switch that in its turn wakes the Module Interface Controller by generating a wake-up event. The Octopus switch turns on the clock that goes to the MIC and gives an *Attention (att)* signal. The MIC can then receive an ATM cell from the module. Prior to the actual transfer, the destination MIC has to be notified that it has traffic waiting. It uses the VCI of the ATM cell to determine the destination module, and sends a *Request* to the receiver-MIC of the destination module. The output port of the Octopus switch that is connected to the receiver MIC contains a *Status register* that collects all outstanding requests (maximal 7). If the receiver-MIC is sleeping, the Octopus switch turns the clock on, and wakes the receiver-MIC by sending an *Attention* signal. Now both MICs that are involved in the communication are awake and the arbitration phase is entered.

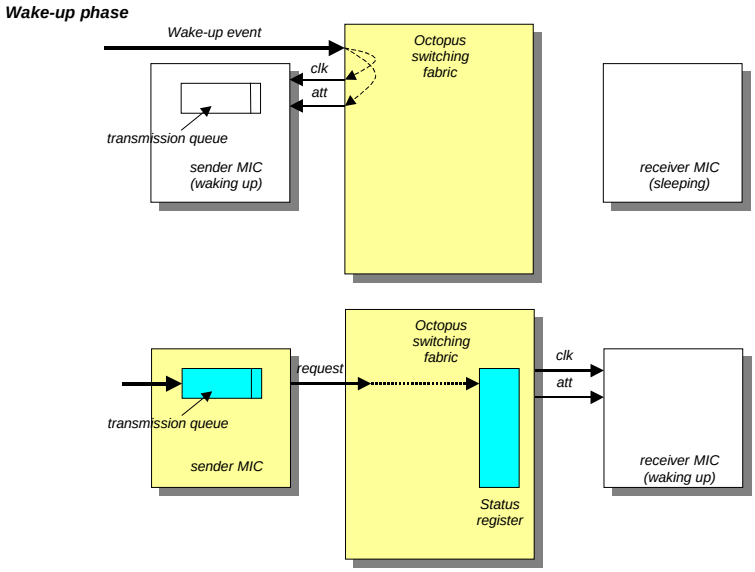

Figure 9: Wake-up phase.

Arbitration phase – The receiver MIC is signalled via the *Attention* signal that it needs to respond to a request. It reads the *Status register* that contains the outstanding requests to its module. It uses a scheduling mechanism to arbitrate between possible multiple requests and replies an acknowledge to the sender-MIC it has selected. Figure 10 shows the arbitration phase in which two simultaneous requests are received by the receiver MIC. The arbiter determines which sender-MIC will be granted the connection, and sends an acknowledge to that sender-MIC. The request of the other sender-MIC will not be honoured yet and has to wait.

The acknowledge is received in the Status register of the sender-MIC, and the MIC receives an attention signal that it should read the register. The attention signal allows the sender-MIC to enter sleep mode as soon as it has made a request. The attention signal will wake-up the MIC when it receives the acknowledge. This mechanism can save energy, since – especially for ad-hoc connections – it can take some time before the connection is established. The MIC reads the status register and can enter the data-transfer phase.

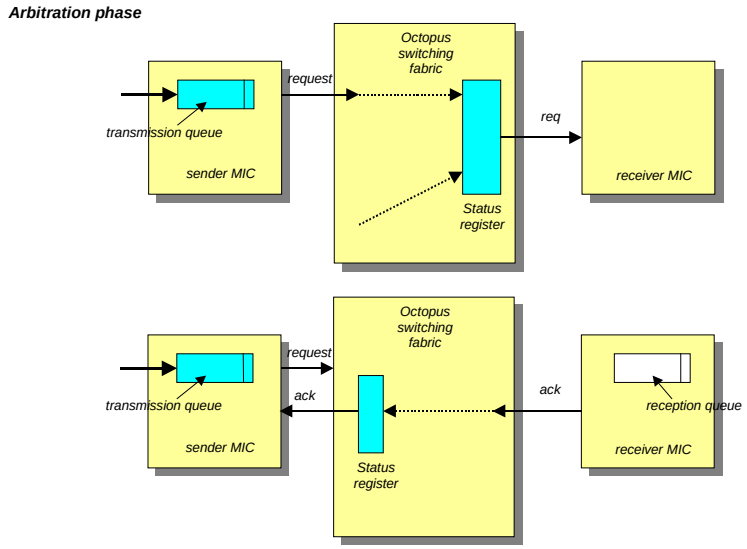

Figure 10: Arbitration phase.

Data-transfer phase – In this phase the transmission queue of the sender-MIC is read and transferred via the switching fabric to the reception queue of the receiver-MIC. The receiver module is notified that it should read the reception queue of its MIC.

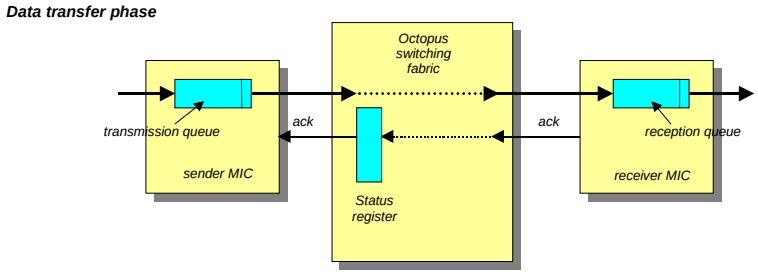

Figure 11: Data-transfer phase.

The acknowledge signal remains active during the communication. When the data-transfer is completed (which in general shall be after one ATM cell), then the release phase is entered.

Release phase – When all bytes of the ATM cell are transmitted, the source MIC releases its request, to signal to the destination MIC that it is ready. If the transfer was successful, i.e. the correct number of bytes were received, the destination MIC returns a *done* signal (i.e. it releases its acknowledge). This is the only error detection performed in the switch since we expect very little errors, and want to keep the forwarding delay as short as possible. This is mainly due to our software implementation of the data-flow. If we would have implemented the data-flow in the MIC with a hardware engine, then we

could more easily implement a better error detection, or even an error correction mechanism.

If the sender module has no more data to send, the sender-module can go to sleep mode.

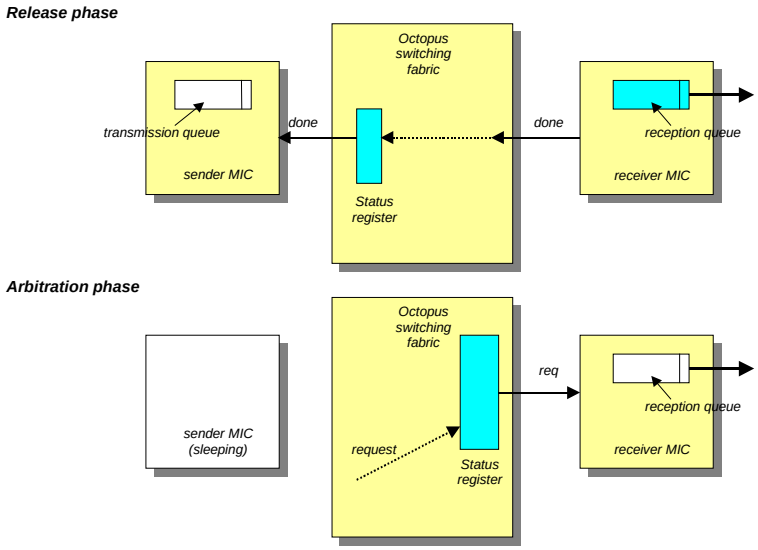

Figure 12: Release phase (and subsequent arbitration phase).

The receiver MIC can now enter the arbitration phase again. Figure 12 shows such a situation. The receiver-MIC schedules again and establishes a connection for the other still outstanding connection request.

4.2.9 Clock Gating

Chapter 3 already showed the locally synchronous, globally asynchronous timing methodology that was used in our architecture. The Octopus switch internally uses a synchronous design methodology. However, the local clock of various building blocks is generated under the control of incoming data tokens. Arrival of a data token restarts the clock signal, and when the processing is done, the clock signal is disabled. Such a mechanism is also know as *clock gating*, which is the most popular method for power reduction of clock signals [17]. When the clock signal of a hardware unit (which in general can be ALUs, memories, state machines, etc.) is not required for some period, the clock feeding the unit can be turned off. The generation of the enable signals increases the complexity of the circuitry, and the timing relation of the signals has to be evaluated carefully to avoid signal glitches at the clock output. Note that the gating signal should be enabled and disabled at a much slower rate compared to the clock frequency. Otherwise the energy required to drive the enable signal may outweight the energy saving.

In the design of the Octopus switch clock gating has been applied at several layers and with different timing granularity. The trade off is to justify the additional hardware and design complexity in managing the various functional units.

Clock gating at the architecture level

At the architecture level, clock gating is a particularly attractive technique because little hardware and design complexity is needed to achieve substantial energy saving. At the system architecture level layer the Octopus interface controllers can be left idle and be sleeping for an extensive period of time. The Octopus switch controls the clock signals of its attached interface controllers. When there is no traffic flow in the system, then all interface controllers are sleeping and receive no clock. When, due to an external event, the Octopus switch notices that it requires an interface controller, it enables the clock signal to the interface controller, and wakes the controller from its sleep. The effectiveness of clock gating in our testbed is described in Section 4.3.3.

Note that, although the design complexity of this method is low, and the energy savings can be high, the startup time can be significant in a particular implementation. The PIC micro-controller that is used in the prototype implementation has several energy saving modes. The intent is to put more functional modes of the controller in idle when the processor goes into a deeper sleep mode. However, it requires more latency to resume computation from a deeper sleep mode. Therefore, the tolerable latency of the attached module determines the sleep mode of the interface controller. In general a module that interfaces at a high data rate requires also a low latency, because otherwise it would require substantial buffering. The PIC controller requires 1024 clock signals (which is equivalent to 51 μs on 20 MHz) to power up from its deepest sleep (internal clock oscillator also turned off) and less than one μs when the internal clock oscillator keeps running.

Clock gating at logic level

Clock gating at the logic level in the Octopus switch is used in various blocks and at different hierarchies. The Octopus is divided into eight basic and identical blocks. Each block interfaces one functional module of the system. All blocks are interconnected to each other via a fully connected crossbar switch. If the block is not needed to handle traffic, then the whole block is inactive, and receives no clock. When a block takes part in the communication stream between two functional modules, then the block becomes partially active. Only those parts of the block that are required during a certain stage in the communication protocol are active and receive clocks.

This principle is applied at both the control flow (for state-machines) and at the data-flow. The advantages for the state-machines are mainly local: no energy is wasted just to stay in the same state. The advantages when applying this principle in the data flow can have more impact. The data-flow in the Octopus switch is based on a pipelined and *guarded* architecture. The basic principle of a guarded architecture is to identify logical conditions at some inputs to a logic circuit that is invariant to the output. To reduce the switching activities inside the Octopus switch, latches (registers) are added into the data

flow that guard switching activities to propagate further inside the switch. The latches are transparent when the data is to be used. Otherwise, if the outputs of a unit are not used, then they do not change.

4.3 Implementation of the Octopus switch

The previous section described the architecture of the Octopus switch. In this section we will present our testbed implementation of the Octopus switch that is used as the interconnection network of the Mobile Digital Companion.

A key goal motivating the design has been simplicity and flexibility. Our goal was to build a testbed from off-the-shelf VLSI components that was easy to design and test. With this testbed we are able to quickly explore the design space of the system. Therefore, the prototype interconnection module is build using a Field Programmable Gate Array surrounded by several low-end and low-power micro-controllers.

4.3.1 Basic components of the testbed

A prototype of the Octopus switch has been implemented on a single small printed circuit board with a standard Field Programmable Gate Array of Xilinx (i.e. XC4010XL) and six low-end micro-controllers (i.e. Microchip PIC 16C66). So, in the testbed we are able to interconnect only six modules instead of eight.

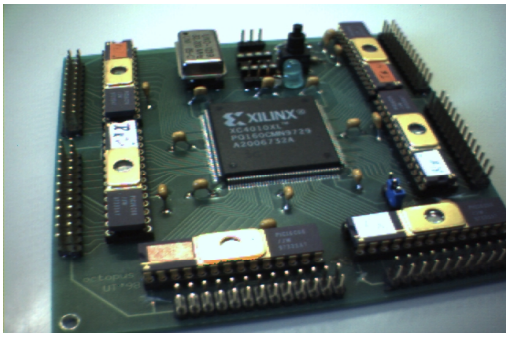

Figure 13: Testbed implementation *Octopus*.

Field Programmable Gate Arrays – Xilinx's FPGA architecture is similar to other gate arrays, with an interior matrix of configurable logic blocks and a surrounding ring of I/O interface blocks. A logic block consists of a combinatorial section and a sequential section. A logic block can also be configured as a small memory (32 by one bit), but this feature was not used in the prototype. The functions of the FPGA (logic, memory, and their interconnects) are stored in an on-chip memory. This technology allows gate arrays to be (re)programmed an unlimited number of times. With proper tools, the design cycle of implementing a design in an FPGA is short. In the last years traditional FPGAs have

gained new positions in the semiconductor industry moving from their initial use for 'glue logic' and fast prototyping purposes to be adopted for typical co-processor tasks. Still, speed area and power consumption are considerably lagging behind the more traditional ASIC designs.

The basic reason of the high energy consumption of an FPGA is due to the interconnect capability within an FPGA: the intrinsic need of routing flexibility implies that more than 90% of the total area is due to interconnect resources. The fact that general connection patterns are provided through switching points instead of dedicated wiring implies that the resulting routing is more resistive and capacitive. As a consequence the relative weight of interconnect contributes on average no less than 50%, and possibly as much as 80% to delay and power consumption.

The main reason for using FPGAs in our design is therefore *flexibility*. We use FPGAs as a dynamic programmable unit, whose function can be changed under program control. This approach creates a test-bed for interconnection structures, arbitration mechanisms, and clocking mechanisms.

Micro-controllers – With an FPGA it is relatively simple and efficient to implement the datapath of a system. However, the FPGA is not suitable for high control complexity. For example, it would require relatively large area if the FPGA has to implement the connection setup protocol, handle timeouts and perform retransmissions. Traditional microprocessors are much better equipped to deal with larger and more complex control structures, and although they are capable of handling the data-flow as well, they typically can only achieve this with a much lower performance than with a hardware implementation.

By combining an FPGA with a traditional processor, it is possible to build a system that uses a mixture of hardware and software in order to exploit the best features of both domains effectively. Another major advantage of using micro-controllers (at least of the type that we have used), is that they consume very little energy, and still provide a reasonable performance. The micro-controllers used have power savings modes in which they consume little energy (less than 1 μA typical standby current).

4.3.2 Implementation

The FPGA can be programmed to operate as the interconnection switch, that connects the modules. Each port of the switch is connected to one micro-controller, so in our prototype we have six micro-controllers. The basic function of the micro-controllers is to perform routing, to establish a connection, and to interface between the switch and the connected modules.

The micro-controllers implement the *Module Interface Controllers* (*MIC*) as described in the architecture. The datapath between the micro-controllers and the FPGA is eight bits wide. The internal datapath in the switch is also 8 bits wide. All data in the system is based on the size of an B-ISDN ATM cell (48 bytes, 5 bytes header). This not only allows us to easily interconnect with an ATM environment, but the size is also adequate: it is small enough to buffer several cells and have a small latency, and large enough to keep the overhead small. Using an other frame format is quite well possible. E.g. a frame

format that uses 64 bytes of payload data with a one byte connection identifier would be more efficient, but would complicate the interface to a B-ISDN ATM network.

We have implemented the interconnection as a fully connected crossbar switch. The switch does not have ATM sized buffers, but just some synchronisation and pipeline registers.

In the current prototype the MICs perform several tasks:

- Connection establishment with the other MICs that are connected to the switch.
- Routing of traffic between the modules
- Scheduling of traffic at the output port of the switch destined for the module
- The actual data-transfer between the module's device and the input port of the switch

Note that the MICs also perform the actual data transfer. The reason for this is that they can perform the data transfer at a sufficient data-rate, and with very low energy consumption. We were able to achieve such a high data-rate because we added special synchronisation circuitry in the Octopus switch.

Connection synchroniser

The switch is capable of having three simultaneous data streams between three disjoint pairs of MICs. Since the data traffic to and from the switch is handled in software by the MICs, the data rate is determined by the rate at which the MICs can handle this. The switch should provide the connection between the two MICs involved in the connection. Both the sender-MIC as the receiver-MIC must participate in the communication at the same time. A complicating factor in the communication is that although the MICs operate at the same frequency and share the same clock signal, they operate completely asynchronously from each other and can be in another internal operating phase. For a reliable connection between the MICs we thus need a *handshake protocol*. However, to achieve the highest speed, it is not possible to have such a handshaking protocol that is implemented in software between the MICs. Therefore the Octopus switch contains some *synchronisation circuitry* at all output ports of the switch that synchronises the traffic between two modules.

When the actual traffic from the transmitting MIC arrives at the output port of the switch, the synchroniser can introduce some delay to adapt the timing of the traffic to the phase of the receiving MIC. The synchroniser therefore has to know the phase of both the sender-MIC and the receiver-MIC. When the receiving MIC acknowledges a connection request, the synchronisation circuitry knows the phase of the receiving MIC. Each ATM cell is preceded by a special *Start of Cell header*. The output port of the switch uses this header to determine when the sender starts transmitting an ATM cell. This header is also used as a synchronisation byte to determine the difference in phase between the sender MIC and receiving MIC and if needed introduces a little delay. In this way, the sender MIC can transfer the ATM cell that is in its transmit queue at the highest speed possible without having to care about any handshaking protocol. Figure 14

shows the datapath of a connection between two modules. The synchroniser is located at the output port of the switching fabric.

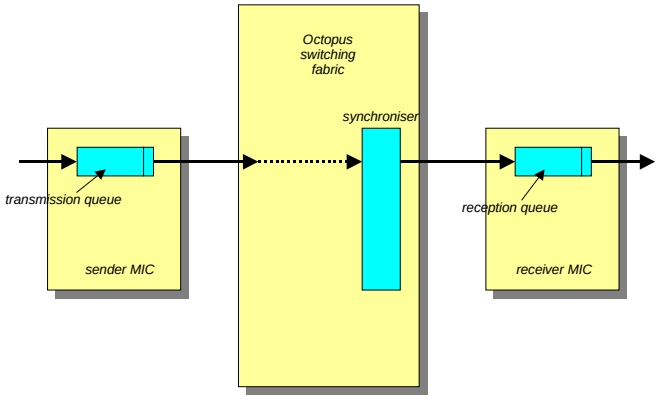

Figure 14: The datapath of a connection between two modules.

The MICs can thus receive and transmit cells at the highest speed possible. Figure 15 shows a simplified part of the code that is used to transmit an ATM cell in our testbed[2]. The ATM cells are stored in dedicated buffers (internal registers). During a communication between two MICs, they can perform no other tasks. The sender-MIC just has to read one byte from the buffer and then write it to the input port of the Octopus switching fabric. The receiver MIC does similar operations, but writes the data to a dedicated receive buffer. As can be seen, just two operations (or 8 clock ticks) are needed to transfer one byte. The achievable data-rate is thus equal to one bit per clock-tick.

[2] This code is only valid for our testbed, and it is shown here to give an indication of the complexity of the required functionality. When integrated in a chip-design, then the micro-controllers will be dedicated state-machines.

```
;*****************************************************************
Tx1Cell
; transmits an ATM cell from buffer 1 to receiver DEST
;*****************************************************************
       movf           DEST,w          ; DEST: the destination MIC
       call           sndReq          ; generate a request
       call           wait4ack        ; (busy) wait for the acknowledge
       ; now the acknowledge has been received start sending the cell
       bsf            PortA,0         ; command Write Data
       set_Octopus_O                  ; set Octopus port (B) to output
       movlw          B'10101010'     ; write sync
       movwf          PortB           ; to Octopus port
       nop
       movf           ATMb1,w         ; read 1st byte from buffer
       movwf          PortB           ; write to Octopus port
       movf           ATMb0,w         ; read 2nd byte from buffer
       movwf          PortB           ; write to Octopus port
       . . . .                        ; more bytes . . .
       movf           ATMb53,w        ; read last byte from buffer
       movwf          PortB           ; write to Octopus port
       set_Octopus_I                  ; set Octopus port to Input
       bcf            PortA,0         ; command Read Status
       call           Release         ; release the connection
       return
```

Figure 15: Code fragment to transmit one ATM cell.

The design methodology used is data driven and based on asynchronous methods, combined with synchronous parts. Each individual part of the switch that is not used at some time, does not receive any data and clock changes, thus minimising energy consumption. The attached modules can be similar to devices found on today's PDAs or notebook computers, but can also include multimedia devices like a camera. If the functionality of the attached module is small and requires little resources, then parts of this functionality can be integrated in the micro-controller as well.

The design has been implemented and tested. The design allows us to do experiment with and make performance measurements of various architectures and interconnection protocols. We used VHDL as a design tool.

4.3.3 Performance

We have measured the performance and energy consumption of the Octopus switch (implemented in an FPGA) including the MICs (implemented in six micro-controllers). All measurements are performed with a clock frequency applied to the switch and MICs ranging from 0.1 to 32 MHz. This is equivalent to a raw data-rate per connection of 0.1 to 32 Mb/s. The Octopus switch is capable to support up to three simultaneous active connections when the connections are disjoint. This makes the total maximal throughput to be 96 Mb/s. This data-rate is more than sufficient to support all our expected data

streams on the mobile computer. A full-custom implementation in an Integrated Circuit will give a higher throughput.

Performance and energy consumption during data transfer

In Figure 16 we have plotted the power consumption for varying frequencies, and with a different number of simultaneous data-flows. In this setup at most three disjoint connections were active, involving six modules, three sending and three receiving MICs. Since the connections are disjoint, no congestion can occur and all units are able to operate at maximum speed.

The graphs show clearly that the energy consumption increases linearly with the frequency, which was expected. It also shows that the required amount of energy depends strongly on the number of data flows in the switch. This effect is mainly due to our aggressive power management. All parts of the system that at some moment have no functionality, are in a low-power mode. The micro-controllers are in a very low-power mode when they don't have traffic, and their contribution to the energy consumption can be ignored. The switch, however, always has a large energy overhead due to the implementation in an FPGA.

Although most parts of the switch idle, it still requires a significant amount of energy. The parts of the switch that remain active are the clock generation and the wake-up circuitry.

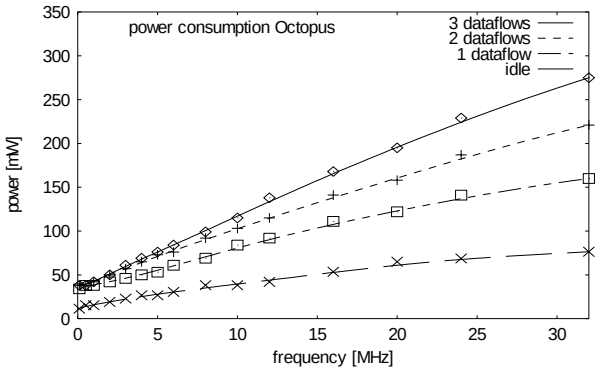

Figure 16: Power consumption Octopus switch with various simultaneous data-streams.

In these measurements the micro-controllers are actively transferring data from their modules to the switch. There are three transmitting modules, and three receiving modules. The transmitting micro-controllers operate in the following sequential phases:

phase 1: reading an ATM cell from the module (*module I/O phase*),

phase 2: establishing a connection with a destination module (*arbitration phase*),

phase 3: transferring the cell to the switch (*data-transfer phase*),

phase 4: waiting for an acknowledge and releasing the connection (*release phase*).

The receiving micro-controllers have three related phases:

Phase 1: arbitration and connection establishment with the source module,

Phase 2: receiving a cell from the switch (*data-transfer phase*),

Phase 3: sending an acknowledge and releasing the connection (*release phase*),

Phase 4: writing the received cell to the module (*module I/O phase*).

Note that the phases of both micro-controllers that take part in the connection operate in parallel, and that the effect is that there is a pipelined dataflow between the modules. During phase 1 of the transmitting module (i.e. an ATM cell is transferred from the module to the micro-controller), the receiving module is in phase 4 and transferes the just received ATM cell to its attached module. The synchronisation point is when a connection is being established.

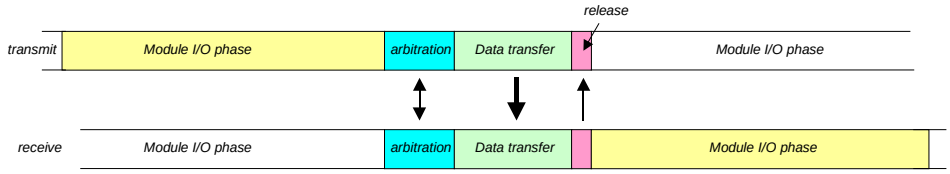

Figure 17: Phases during switch communication.

We have also measured the time in which the individual phases contributed to the total time needed to transfer one ATM cell. In the setup the arbitration phase took 9% of the time, the data-transfer between the micro-controller and the switch, including the release phase, took 27%, and the transfer between module and micro-controller took 64%. The contributions to the time of the receiving and transmitting controller were thus irrespective to their function, and only depended on the phase. Note that the interface with the switch is highly optimised, and that the interface with the module is more general, and requires more time. To determine the effect of the data-rate with the module to the energy consumption, we have made another setup. In this setup the data-rate with the module was infinite (in fact we removed the module I/O phase (1) from the transmitting controller, and the module I/O phase (4) from the receving module). Figure 18 show the result of these measurements with three simultaneous traffic flows.

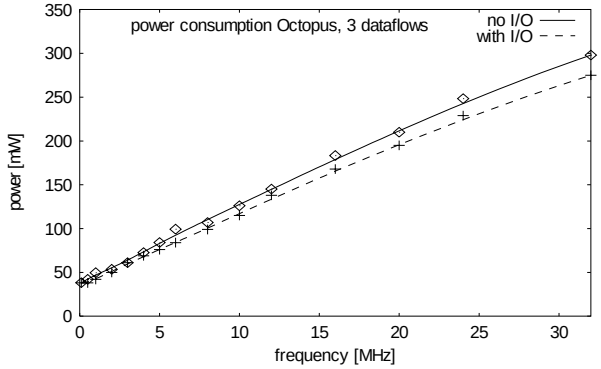

Figure 18: Power consumption Octopus switch with three data-streams, with and without I/O to the module.

Maybe somewhat surprisingly, the effect of having I/O with the module or not, is very small. If there is no I/O with the module, the energy consumption is even a little higher than when there is actual data traffic with the modules. This effect is due to the fact that the activity of the switch relatively to the activity of the MICs is much higher than in the previous setup. This requires more energy since the FPGA (i.e. switch) is less energy efficient than the microcontroller (i.e. MIC). The arbitration phase takes 31% of the total time needed to transfer one ATM cell, and the actual data-transfer phase takes 69% of the total time.

The effect of clock gating to the energy consumption

To determine the usefulness of our clock gating strategy and guarding of data on the energy consumption of the switch, we have made two versions of the switch: one optimised that uses clock gating and data guarding wherever possible, and one traditional design that did not use these techniques. Figure 19 shows the increase in power consumption of the traditional design versus the frequency. The figure plots two graphs, one when no data flow occurs and the switch is idle, and another when one connection between two modules is active.

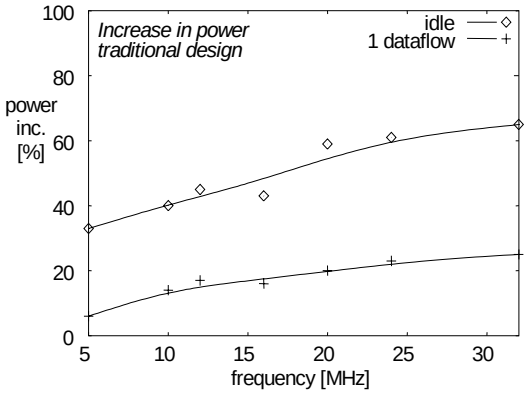

Figure 19: Effect of clock-gating to the power consumption of the Octopus switch.

When the switch is idle, the increase in energy consumption of the traditional design can be up to 65%. The optimised design is then in an energy efficient mode, and no useless clocks or data transitions occur. The main source of energy consumption that is left is due to the inherent energy consumption of the FPGA. This shows that – even for an FPGA that has a high energy consumption overhead – it is indeed worthwhile to optimise the clocking system.

When the switch is active, and there is one connection that communicates at maximum speed, the advantage is less: the increase in energy consumption of the traditional design is 25%. This can be expected since, when there is a connection, the attached MICs are also active and consume energy.

4.3.4 Conclusion

The realised prototype shows that it is relatively simple to implement an architecture that is suitable for becoming the interconnection structure of a mobile computer. The combination of a FPGA surrounded by several small micro-controllers proved to be a flexible prototyping testbed. The complexity of the architecture is low, which make it feasible to built the switch in a custom IC.

The measurements show that the costs of having a flexible dynamic scheduling can be significant. The overhead introduced by the arbitration and release phase is significant (i.e. 30%) when only one ATM cell is actually being transferred. A solution can be to have larger packets of multiple ATM cells, but this will lead to a bigger latency.

The performance of this prototype provides bandwidth guarantees and enough bandwidth for many multimedia applications. The power management that was used in the switch showed to be very effective. The energy consumption of the switch is strongly related to the number of communication streams in the switch. When the switch does not need to communicate, it is in an energy saving mode.

Clock gating and data guarding is being used in the switching fabric whenever possible. The effect of these techniques in energy savings is quite significant, and can be up to 65% when the switch is idle.

4.4 Summary and conclusions

In this chapter we discussed the design of an interconnection architecture for a handheld mobile multimedia computer. Energy management is the general theme in the design of the system architecture since battery life is limited and battery weight is an important factor. We have shown that there is a vital relationship between hardware architecture, operating system architecture and applications architecture, where each benefits from the others. In our architecture we have applied several supplementary energy reduction techniques on all levels of the system. Achieving high energy efficiency requires first of all the elimination of the waste that typically dominates the energy consumption in general-purpose processors. The second main principle used is to have a high locality of reference. The philosophy is that all operations that are required on the data should be done at the place where it the most efficient, thereby also minimising the transport of data through the system.

The interconnect of the architecture is based on a switch, called *Octopus*, which interconnects a general-purpose processor, programmable (multimedia) devices (called modules), and a wireless network interface. In our model, the interconnection network is transparent and provides only a direct connection between two functional modules. The Octopus switch transmit and receive ports are simple, containing only minimal buffering and arbitration functionality. The switch supports two basic connection types: ad-hoc connections for traffic with no hard real-time requirements, and guaranteed connections for traffic with (hard) real-time requirements. To assign the bandwidth in the switch we use static scheduling for guaranteed connections, and dynamic scheduling for ad-hoc connections. A certain portion of the bandwidth is allocated to traffic of guaranteed connections, and any unused bandwidth can be used for other traffic.

The architecture uses a locally synchronous, globally asynchronous timing methodology. The data paths through the switch only consume energy when data is being transferred, leaving most of the switch turned off nearly all the time. This is achieved by using clock gating and data guarding techniques in the switching fabric and energy management mechanisms in the module interface controllers.

We have built a prototype of this architecture from off-the-shelf VLSI components that was easy to design and test. A key goal motivating the design has been simplicity and flexibility. A Field Programmable Gate Array can be programmed to operate as the interconnection switch that connects the modules. Each port of the switch is connected to a micro-controller that performs routing, connection establishment, and provides the interface between the switch and the connected modules.

The performance of this prototype provides bandwidth guarantees and enough bandwidth for many multimedia applications. The power management that was used in

the switch showed to be very effective. The energy consumption of the switch is strongly related to the number of communication streams in the switch. When the switch does not need to communicate, it is in an energy saving mode. Clock gating and data guarding is being used in the switching fabric whenever possible. The effect of these techniques in energy savings is quite significant, and can be up to 65% when the switch is idle.

The prototype showed that it is already feasible with standard components to build an energy efficient architecture that allows many devices in the system to be turned off (including the CPU), while still providing enough performance to support multimedia applications. Having an energy efficient architecture that is capable to handle adaptability and flexibility in a mobile multimedia environment requires more than just a suitable hardware platform. First of all we need to have an operating system architecture that can deal with the hardware platform and the adaptability and flexibility of its devices. Optimisations across diverse layers and functions, not only at the operating systems level, is crucial. Managing and exploiting this diversity is the key system design problem. A model that encompasses different levels of granularity of the system is essential in the design of an energy management system and in assisting the system designer in making the right decisions in the many trade-offs that can be made in the system design. Finally, to fully exploit the possibilities offered by the reconfigurable hardware, we need to have proper operating system support for reconfigurable computing, so that these components can be reprogrammed adequate when the system or the application can benefit from it.

References

[1] Adam J.F., Houh H.H., Tennenhouse D.L.: “Experience with the VuNet: a network architecture for a distributed multimedia system”, *Proceedings of the IEEE 18th Conference on Local Computer Networks*, pp. 70-76, Minneapolis MN, September 1993.

[2] Anderson T.E., Owicki S.S., Saxe J.B., Tacker C.P.: “High speed switch scheduling for local area networks”, Proceedings *ACM ASPLOS V*, pp. 98-110, 1992.

[3] Barham P., Hayter M., McAuley D., Pratt I.: “Devices on the Desk Area Network”, March 1994.

[4] Benini L., De Micheli G.: “Dynamic Power Management, design techniques and CAD tools”, *Kluwer*

[5] Doyle van Meter, R.: “A brief survey of current work on network attached peripherals”, *ACM Operating Systems Review*, Jan. 1996.

[6] Eberle H., Oertli E.: “Switcherland: a QoS communication architecture for workstation clusters”, *Proceedings ISCA ’98 – 25th annual Int. Symposium on Computer Architecture*, Barcelona, June 1998.

[7] Havinga P.J.M., Smit G.J.M., Bos M.: “Energy efficient wireless ATM design”, *proceedings second IEEE international workshop on wireless mobile ATM implementations (wmATM’99)*, pp. 11-22, June, 1999.

[8] Havinga P.J.M., Smit G.J.M.: “Octopus: embracing the energy efficiency of handheld multimedia computers” , *proceedings fifth annual ACM/IEEE international conference on mobile computing and networking (Mobicom’99)*, pp.77-87, August 1999.

[9] Havinga P.J.M., Smit G.J.M.: “Octopus – an energy-efficient architecture for wireless multimedia systems”, *Proceedings Program for Research on Integrated Systems and Circuits (ProRISC’99)*, pp. 185-192, November 1999.

[10] Hayter M.D., McAuley D.R.: “The desk area network”, *ACM Operating systems review*, Vol. 25 No 4, pp. 14-21, October 1991.

[11] Hui J.: “Switching and traffic theory for integrated broadband networks”, *Kluwer Academic Press*, 1990.

[12] Karol M., Hutchy M. Morgan S.: “Input versus output queueing on a space-division packet switch”, *IEEE transactions on communication*, 35(12), pp. 1347-1356, 1987.

[13] Leijten J.A.J.: “Real-time constrained reconfigurable communication between embedded processors”, *Ph.D. thesis, Eindhoven University of Technology*, November 1998.

[14] Prycker: “Asynchronous Transfer Mode”, 1991.

[15] Smit G.J.M.: “The design of central switch communication systems for multimedia applications”, *Ph.D. thesis, University of Twente*, 1994.

[16] Truman T.E., Pering T., Doering R., Brodersen R.W.: The InfoPad multimedia terminal: a portable device for wireless information access”, *IEEE transactions on computers*, Vol. 47, No. 10, pp. 1073-1087, October 1998.

[17] Yeap G.K.: “Practical low power digital VLSI design”, *Kluwer Academic Publishers*, ISBN 0-7923-80.

[18] Zhang H., Wan M., George V., Rabaey J.: “Interconnect architecture exploration for low-energy reconfigurable single-chip DSPs”, *Proceedings of the WVLSI*, Orlando, Fl, April 1999.