FORTH-ICS / TR-124 July 1994

A Memory Controller for Access Interleaving over a single Rambus

Zacharenia Xanthaki

ABSTRACT

While the need for higher memory bandwidth is increasing, the traditional DRAM interface becomes more and more a bottleneck that keeps the bandwidth of DRAM chips at low levels. This fact forces the designers to use expensive techniques for organizing the memory system in order to meet the bandwidth requirements.

The *Rambus* solution is based on new DRAM architecture and a new DRAM interface that provides high bandwidth communication between the DRAM chips and the processing elements. The Rambus Channel uses a synchronous block oriented protocol. Each transcaction consists of 3 packets: request acknowledge and data. The bus runs at a 250MHz clock, and achieves a peak data rate of 500MBytes/s.

We have designed a memory controller for access interleaving over Rambus. Our controller maximizes the utilization of the bus, by interleaving the requests whenever it is possible. The scheduling algorithm used by our controller schedules a new request every 9 Rambus cycles (36ns), with corresponds to a peak data throughput of 222 MBytes/s. Our controller handles individual word accessess, thus it could be used in a system that requires high data rates with increased bandwidth requirements, and accesses to non-sequential memory words, for example a supercomputer accessing non-cached vector elements with random stride. The chip was designed using the ES2 1.0 μm CMOS standard cell proccess and it was successfully simulated for speeds up to 18.8MHz (53ns).

† Institute of Computer Science, F.O.R.T.H., Greece E-mail: xanthaki@csi.forth.gr

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

A Memory Controller for Access Interleaving over a single Rambus

Zacharenia E.M. Xanthaki

Submitted in Partial Satisfaction of the Requirements for the Degree of
Master of Science
in
Computer Science

Iraklio, February 1994

Abstract

While the need for higher memory bandwidth is increasing, the traditional DRAM interface becomes more and more a bottleneck that keeps the bandwidth of DRAM chips at low levels. This fact forces the designers to use expensive techniques for organizing the memory system in order to meet the bandwidth requirements.

The *Rambus* solution is based on new DRAM architecture and a new DRAM interface that provides high bandwidth communication between the DRAM chips and the processing elements. The Rambus Channel uses a synchronous block oriented protocol. Each transcaction consists of 3 packets: request acknowledge and data. The bus runs at a 250MHz clock, and achieves a peak data rate of 500MBytes/s.

We have designed a memory controller for access interleaving over Rambus. Our controller maximizes the utilization of the bus, by interleaving the requests whenever it is possible. The scheduling algorithm used by our controller schedules a new request every 9 Rambus cycles (36ns), with corresponds to a peak data throughput of 222 MBytes/s. Our controller handles individual word accessess, thus it could be used in a system that requires high data rates with increased bandwidth requirements, and accesses to non-sequential memory words, for example a supercomputer accessing non-cached vector elements with random stride. The chip was designed using the ES2 1.0 µm CMOS standard cell proccess and it was successfully simulated for speeds up to 18.8MHz (53ns).

Acknowledgements

I would like to thank all the people who helped me to complete this thesis. First of all, I would like to thank my supervisor Professor Manolis Katevenis for the original idea, and his invaluable guidance throughout this work. On his ideas are based the solutions on many of the problems that I had to deal with in this thesis. I thank him deeply.

I would also like to thank Professor Apostolos Traganitis for his guidance during my studies in the University of Crete. I also thank him as well as Professor Nikos Alvertos, for participating in my thesis committee.

George Dimitriadis help me many times with the problems that I had with the design software, and our discussions help me clarify some implementation issues. Penny Vatsolaki gave me with some useful advice, especially during the place & route phase of the design. Discussions with George Kalokerinos helped in the definition of the processor interface. Aristidis Eythymiou gave me the CAM metrics, which I have used for comparison purposes in 2.3.1. George Xouris helped me with the creation of the Appendix A, and the correction of the greek version of this text. I thank them all.

I would also like to thank the Institute of Computer Science of the Foundation for Research and Technology–Hellas (FORTH) for the financial and technical support provided during my studies. This work was supported by the ESPRIT SHIPS (contract 6253) research project. The VLSI design software used in this work was provided through the ESPRIT EUROCHIP (contract 3603) educational program.

Finally, I would like to thank my parents and my family for their love and understanding.

Table of Contents

Chapter 3: Implementation

Chapter 4: Design Methodology and Chip Metrics

Chapter 1

Introduction

While the need for higher memory bandwidth is increasing, the traditional DRAM interface becomes more and more a bottleneck that keeps the bandwidth of DRAM chips at low levels. This fact forces the designers to use expensive techniques for organizing the memory system in order to meet the bandwidth requirements. The *Rambus* solution is based on new DRAM architecture and a new DRAM interface that provides high bandwidth communication between the DRAM chips and the processing elements. In section 1.1 we describe the traditional DRAM interface and the corresponding memory system architecture, while in section 1.2 we describe the Rambus DRAMs (RDRAMs), the new DRAM interface, and the architecture of a memory system employing RDRAMs.

1.1 Traditional DRAMs and corresponding memory architecture

1.1.1 DRAM interface

In traditional DRAMs, the memory array is organized as a rectangle of rows and columns. The top half of the address specifies the row and the bottom half specifies the column : the position of the word within the row. Each time a new word is read or written the *row address* is used to read the entire row, which is placed in the sense amplifier latches. Then, the *column address* is used to select the requested word. If there are successive accesses to the same row the access time and thus the cycle time can be reduced by using *page mode*. In *page mode* accesses, there is only a column access since the requested row is already found in the sense amplifier latch. In this case the sense amplifier latch is used as a cache.

Traditional DRAMs have seperate pins for address control and data. The number of pins and the cost of the DRAM chip is reduced by time-multiplexing the row address and the column address on the same pins, and by having a small number (usually 1 or 4) of data pins. The typical control signals are RAS, CAS, WE, OE. For *random* accesses first the row address is read by the chip when RAS is lowered and then the column address is read when CAS is lowered. For *page mode* accesses, only the column address is specified and only CAS is lowered. The value of WE determines the type of the access (read/write). The value of OE determines when

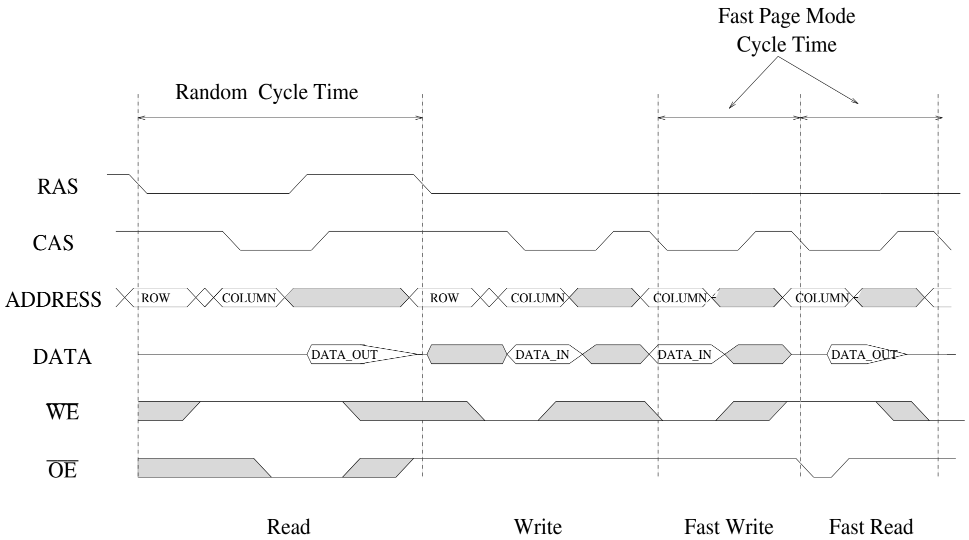


Figure 1.1: Traditional DRAM signals

output data is valid. Traditional DRAM timing is shown in Figure 1.1.

1.1.2 Memory system design

Assuming a typical page mode cycle of 40ns, the peak bandwidth of a traditional DRAM is 4bits/40ns (12.5Mbytes/sec), while the average bandwidth can be significantly smaller if the percentage of page mode accesses is small. One way to increase bandwidth is to transfer more bits in one cycle by using a wide data bus and memory banks of the same width. A memory bank consists of a number of DRAM chips that are accessed in parallel and each provides a portion of the memory word. If more bandwidth is needed, an *interleaved* memory system is built consisting of more than one memory banks. In an interleaved system each bank has seperate address and control circuits and can be accessed indepentendly. The data bus, however, is common for all memory banks so the data from the various banks have to be placed sequentially on it. Figure 1.2 shows a 4-way interleaved memory system with a 64 bit wide data bus, using 16 DRAMs per memory bank and 64 DRAMs in total.

The peak bandwidth of an interleaved memory system is equal to the *non-interleaved bandwidth* times the *number of banks*. The average bandwidth depends on the pattern of memory accesses and the mapping of addresses to memory banks. The mapping of consecutive *words* to different banks optimizes sequential memory accesses, which can also be optimized – perhaps more effectively – using page mode [HePa]. In this case mapping consecutive *rows* to different banks may be more effective. The disadvantages of interleaving is that is complicates control and, since it requires a large number of chips, it becomes more expensive as DRAM capacity increase.

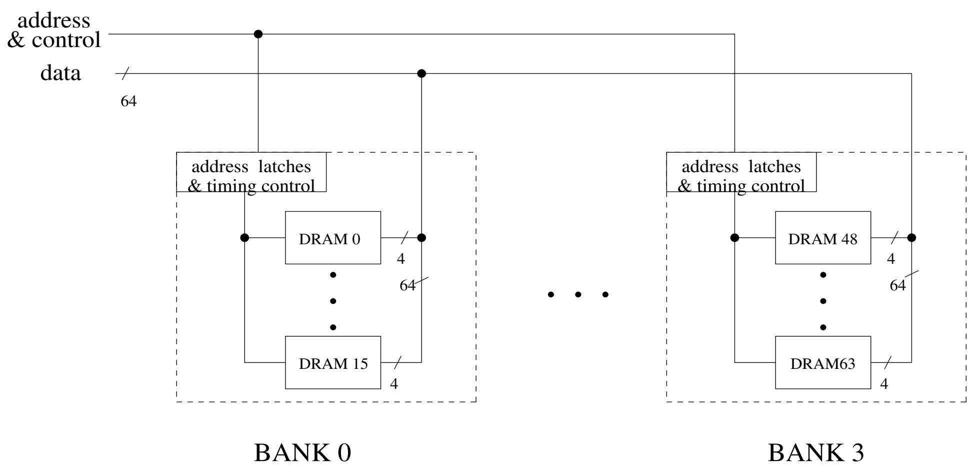


Figure 1.2: An Interleaved Memory System

In the traditional memory systems the DRAM-cache is not used effectively. Each memory bank has a single, very long, cache line which gives a low hit ratio. However, even when requests are addressed to the DRAM's cache the current interface does not provide enough bandwidth for fast access to the large number of bits stored there.

1.1.3 Cached DRAMs

An optimization of the traditional DRAMs are the *cached* or *enhanced* DRAMs, which integrate a small (2K-16K) SRAM cache into the DRAM chip. There are various organizations of the SRAM cache: The Mitsubishi *cached*-DRAM [Mitsu] has a 16K cache organized in 256 lines of 64-bits each, while the Ramtron *enchanced* DRAM [Ramtron] has a single row 2K SRAM cache. The integration of the SRAM cache into the DRAM chip allows for very high bandwidth communication, between the SRAM cache and the DRAM memory array, by using very wide data path. This enables the cache to be refilled in a single cycle, breaking the word-serial bottleneck between the DRAM chip and seperate cache. With the *cached* DRAMs the need for a seperate second level cache is eliminated or reduced since a small embedded cache, with larger block size, is as efficient as a separate larger cache [Mitsu2]. By eliminating the separate cache controller and SRAM chips, required for the second level cache, the system cost, board space, and power consumption is reduced.

The Ramtron *enchanced* DRAM has a hit cycle-time of 15ns, and a miss cycle time of 65ns, the chip peak bandwidth is 33Mbytes/sec (4bits/15ns). The Mitsubishi *cached*-DRAM has a hit cycle-time of 10ns, an average cycle-time of 70ns and chip peak bandwidth 50Mbytes/sec (4bits/10ns). Both employ the traditional DRAM interface, with some extra pins for control and, the second, with seperate cache and DRAM address pins,thus doubling the number of address pins.

In cached DRAMs both the hit/page-mode cycle-time, and the miss/random cycle-time are significantly improved, compared to the traditional DRAMs cycle-times. However, the cached DRAMs employ the traditional interface, which is still the bottleneck that limits the chip bandwidth : The cached DRAM chip bandwidth is 4 times the traditional DRAM chip bandwidth, but 10 times less than the Rambus-DRAM bandwidth as we shall see in the following section. The interface also imposes the traditional memory system achitecture that we have already descibed, so the banwdidth requirements still determine the system size.

The conclusion is that the current DRAM interface leads to the design of memory systems with wide data buses and complicated control which employ a large number of chips that operate in parallel and consume a lot of board space and power. Cached DRAMs are partial solutions, however the Rambus approach is significantly better as we will see next.

1.2 The Rambus approach

In the Rambus approach the traditional RAS−CAS interface and the wide interleaved bus is replaced by the *Rambus Channel*, a narrow, high-speed bus which is used to directly connect Rambus-DRAM chips "RDRAMs" to the processor or to a memory controller. Any device that is connected to the *Rambus Channel* must adhere to the Rambus interface protocol [RaGu]. The Rambus interface protocol is layered : The first (physical) layer describes the electrical and mechanical interface, while the second (logical) layer describes the format of the bus transactions. In the rest of this section we briefly present the physical layer and the RDRAM chip, while the logical layer is described in section 1.2.1

The *Rambus Channel* is synchronous, meaning that data are transferred in synchrony with a clock edge. For Rambus data are transferred on both edges of the clock. The bus clock runs at 250 MHz; the bus has 9 lines for data, (one byte plus one bit for parity) so the bus bandwidth is 500MBytes/sec. This high speed is achieved by using low voltage swing signals, resistor-terminated controlled-impedance lines and a precise clocking method.

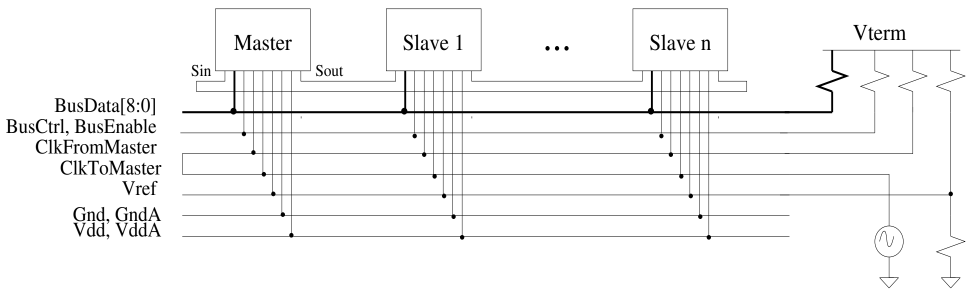


Figure 1.3: The Rambus Channel

The *Rambus master* is the only device that can generate requests and can be either the processor or a memory controller. The master must be located at one end of the Channel and termination resistors at the other. The *Rambus slaves* can only reply to requests issued by the master. The typical Rambus slave is the RDRAM, but it may also be an expansion device which is used to attach a secondary Rambus Channel (RModule). The bus length is limited by the maximum allowable clock skew to 32 RDRAMs or 10 RModules.

As shown in Figure 1.3 the *Rambus Channel* has 16 active signals. Just 9 of these signals are used to carry all address, data, and most control information. There are two additional control signals : *BusEnable* is used to reset/activate bus devices and *BusCtrl* is used to denote the beginning of a new request packet transmit opcode and acknowledgement packets. There are two clock signals, one for each direction of transmition, in order to minimize clock to data skew. Data travelling from the master to the slaves are synchronous with *ClkFromMaster*, while data travelling from the slaves to the master are synchronous with *ClkToMaster*. All the above mentioned signals are low voltage swing signals. The voltage swing is +/- 0.3V around the reference voltage *Vref* = 2.2V. *Sin* and *Sout* signals are CMOS level signals, they form a daisy chain and are used during system initialization.

The *Rambus Interface* circuit implements the physical layer of Rambus Protocol and must be included in any device attached to the Rambus Channel. The Rambus Interface converts low voltage swing signals to normal CMOS levels, and the 250MHz Rambus clock to 4 times slower clock,*SClk*, that has a known phase relationship with the Rambus clock. It syncronizes on-chip with off-chip clocks and provides the clocks for transmiting and receiving *Rambus Channel* data. The Rambus Interface is integrated into the RDRAM and it is also available as ASIC I/O cell or full custom cell so that the application chip designer does not have to implement the physical layer of the Rambus Protocol.

The organization of the core of an RDRAM is similar to a traditonal CMOS DRAM. It includes traditional memory arrays, sense amplifiers decoding and refresh logic. Since a RDRAM chip is connected directly to the Rambus Channel, it also includes the Rambus Interface and the logic that implements the logical layer of the Rambus Slave Protocol. The behavior of the RDRAM is described in the Rambus Protocol section. The first available RDRAM chips have 4.5Mbits of storage, organized in two independed memory banks. Each bank has 256 rows and each row has 1K 9-bit words. The sense amplifier latch is used as a cache line that can be accessed at a rate of 500Mbytes/sec.

1.2.1 Rambus Protocol

Data and addresses on Rambus are transfered in packets. A packet can start only in an even bus-cycle. (In the following we shall refer to the half-clock period (2ns) as *bus-cycle* and the clock period (4ns) as *bus-cycle pair*) There are three types of packets : *request*, *acknowledge* and *data*.

The *request* packet is 6 bus-cycles long and contains the type of operation, the address and the number of bytes requested. The *data* packet can be 1 to 256 bus-cycles long, which means that 1 to 256 bytes can be transferred with a single transaction. The *acknowledge* packet is 2 bus-cycles long. All control and data information is carried in 10 signals : BusCtrl, BusData[8:0]. *Request* packets use all 10 signals, data packets use only *BusData[8:0]*, and *acknowledge* packets use only *BusCtrl* consequently *acknowledge* and *data* packets may be concurrent.

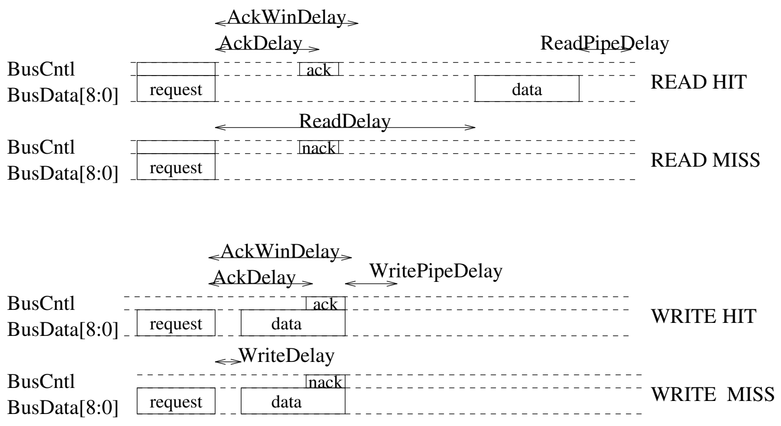


Figure 1.4: Timing of Rambus Transactions

The master initiates a transaction by sending a request packet. The addressed slave responds by sending an acknowledge packet after a delay of *AckDelay*. The master expects the acknowledge packet within the *AckWinDelay* interval. Since the start bit of request packet and the acknowledge packet are both sent on the *BusCtrl*, an acknowledge packet could be mistaken by the slaves as the beginning of new request packet. To prevent this from happening, all responding devices ignore the *BusCtrl* wire as a start bit of a new request packet during the *AckWinDelay* interval.

If the requested address is found in the cache of the addressed RDRAM, the RDRAM returns an *Ack* to the master and the request is considered a *hit*. In the case of a read hit the responding device sends the data packet to the master after a delay of *ReadDelay*. In a write hit the master sends the write data packet to the addressed slave after a delay of *WriteDelay*. The end of the current transaction is either the end of the acknowledge window or the end of the data packet, whichever is longer. The next transaction may begin *ReadPipeDelay/WritePipeDelay* after the end of the current read/write transaction.

If the requested address is not found in the cache of the addressed RDRAM, the RDRAM returns a *Nack* to the master and the request is considered a *miss*. When a Nack is returned, the current transaction is completed at the end of the acknowledge window or data packet window whichever is longer. After a Nack, the addressed slave starts the access to the requested row which will be available after some device depended time. The master must retry after this device depended time has elapsed, early retrials will also fail. A Nack is also returned if there is a violation of *WritePipeDelay* or a pending refresh.

The values of *AckDelay, AckWinDelay, ReadDelay, WriteDelay* are programmed, within the manufacturer's allowable range, in the delay registers of the responding devices and are set during system initialization. The *ReadPipeDelay* is zero. In this work we assume a *WritePipeDelay* of two cycles.

1.2.2 System Design

The design of a memory system using RDRAMs is straitforward. If the CPU has a Rambus Interface then it can be directly connected to the Rambus Channel. If it doesn't then a RDRAM controller is placed between the CPU and the Rambus Channel. The maximum number of RDRAMs that can be directly connected to a Rambus Channel is 32 which gives 16 Mbytes for the 4.5Mbit RDRAMs. If more *memory* is needed the Rambus Channel can be expanded by attaching one or more secondary Rambus Channels via a transceiver. The bus bandwidth is 500Mbytes per second and it independed of the number of RDRAMs. If more *bandwidth* is needed, more than one Rambus Channels can be used to form an interleaved memory system. Each channel increments the bandwidth by 500 Mbytes/sec.

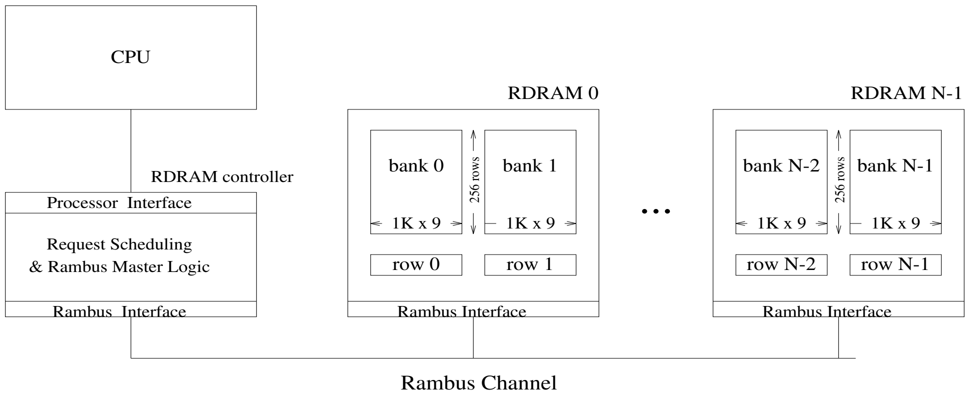


Figure 1.5: Our Rambus System - A Main Memory Application

In the rest of this work we assume the Rambus system is shown in Figure 1.5: it consists of a single Rambus Channel and a memory controller between the CPU and the Rambus Channel. The main parts of the controller are shown in Figure 1.5. It includes the interface to the processor generating the requests and the Rambus Interface that implements the physical layer of the Rambus protocol. The function of a simple Rambus controller is to store processor requests, put them in the Rambus request packet format and schedule the requests in the appropriate Rambus cycles to ensure that there wont be any collisions. It also reads the read data responses and forwards them to the processor.

A single Rambus Channel with only 4.5Mbit RDRAMs attached to it may have 2-64 intependend memory banks and an equal number of 1K by 9 cache-lines that comprise the second level cache with a hit ratio much higher than a traditional memory system of equal size and the contents of each cache-line can be transferred at 500Mbytes/sec. The designer has the option to map consecutive row addresses to consecutive RDRAMs maximizing the locality of the second level cache.

1.3 Comparison

For comparison purposes, let us assume a traditional 4Mbit DRAM with 4 data pins and 1Kx4 cache. For a 64 bit CPU, at least 16 DRAMs are required, organized in a single memory bank, while the minimum increment of the system is one memory bank (8Mbytes). Each bank has a single 8 Kbyte cache line which gives a low hit ratio. Assuming a 40ns page-mode cycle, a 120ns random cycle, and 30% cache hit ratio [RaOv] the average chip bandwidth is 5.2Mbytes/sec.

The corresponding Rambus memory system could operate with only one RDRAM, and has a minimum increment of 0.5Mbytes (one RDRAM). A system with 16 RDRAMs (the minimum number of chips for the 64-bit DRAM-based system) will have 32 independed memory banks, and 32 cachelines, 1Kbyte each, giving a high hit ratio. Assuming a read to write ratio of 15/85 for system with a write-through first level cache,and 90% hit ratio for the Rambus accesses, the average data rate per chip is 170 MBytes/sec.

As shown above, for a memory system with few DRAM chips, the Rambus system significantly outperforms a traditional system of equal size. Unlike the traditional system, the complexity of of Rambus system does not have to increase as bandwith requirements are increasing. The average data bandwidth icreases significantly with larger data packets.

1.4 Overview of this work

This work is the design of a memory controller for access interleaving over a single Rambus. By exploiting the interleaving capability provided by the Rambus protocol we can increase bandwidth utilization for the Rambus Channel by interleaving requests. By predicting whether a request will result in hit or a miss, the missed request is repeated only once. We also exploit the MissDelay and the ReadDelay intervals during which a non-interleaved Rambus is normally idle.

The rest of this work deals with access interleaving on Rambus, and the design and the implementation of the Rambus controller. Chapter 2 discusses access interleaving on Rambus, the scheduling algorithm and how it affects the architecture of the memory controller. Chapter 3 describes the implementation of various parts of the chip. Chapter 4 describes the design methodology used and the chip metrics. Chapter 5 gives our conclusions.

Chapter 2

Access Interleaving on Rambus

2.1 Interleaving capability of the Rambus Channel

Access interleaving is inherently supported in the Rambus Protocol. From the description of the Rambus Protocol we can see that access interleaving is interesting in the following two cases: The *first* is when a request results in a *miss* then the transaction is completed at the end of the Acknowledge window, but the master will have to wait for Read/Write MissDelay cycles before reissuing the request. During this interval the bus is free and, for the 4.5Mbit RDRAMs, this interval could be used for 1 to 5 transactions, as shown in Figure 2.1.b. The *second* case is when the request is a *read-hit*, then the time between the end the acknowledge window and the beginning of the returned data packet, is also free and it is enough for one miss request, or a read-hit request packet, as shown in Figure 2.1.a.

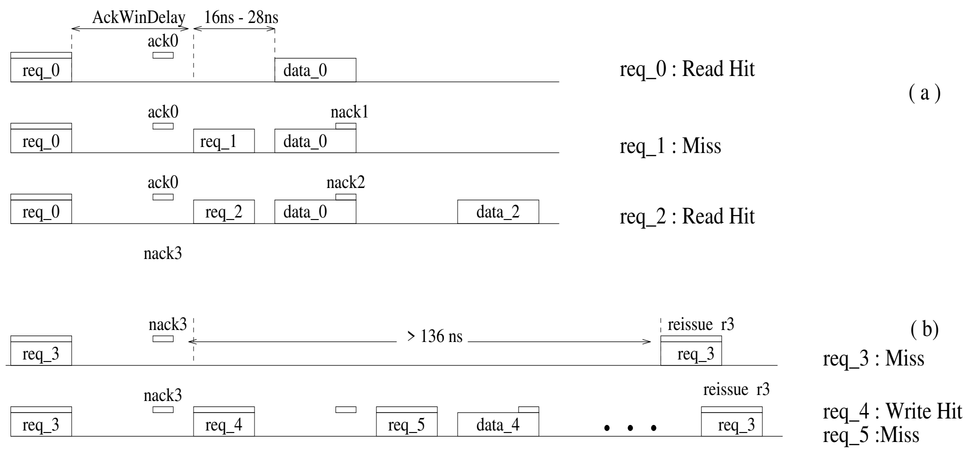


Figure 2.1: Interleaving capability of the Rambus Channel

We have already mentioned in the Rambus Protocol section that *request* and *acknowledge* packets can not overlap. This limits the interleaving ability, because a new request packet can only be sent *AckWinDelay* time after the end of the previous request packet. In section 2.1.1 we show a *modified* architecture for the Rambus Channel that does not impose this limitation. In section 2.2 we shall examine what kinds of request can be interleaved and how the interleaving affects the design of the memory controller. One of the major issues involved in the design of the memory controller is the scheduling algorithm, which will be discussed in section 2.3.

2.1.1 How Rambus could be modified for acknowledge/request overlap

As saw in Chapter 1, the *BusCtrl* wire is used both for request and for acknowledge packets. The beginning of a request packet occurs when the *BusCtrl* is asserted. The acknowledge packet is also sent on the *BusCtrl* wire, within *AckWinDelay* time after the request packet. Slaves ignore the *BusCtrl* wire for *AckWinDelay* time to avoid mistaking the acknowledge packet for the beginning of a new request packet. As a result the possibility of interleaving is limited, because the earliest time when we can send a new request packet is *AckWinDelay* after the end of the previous request packet.

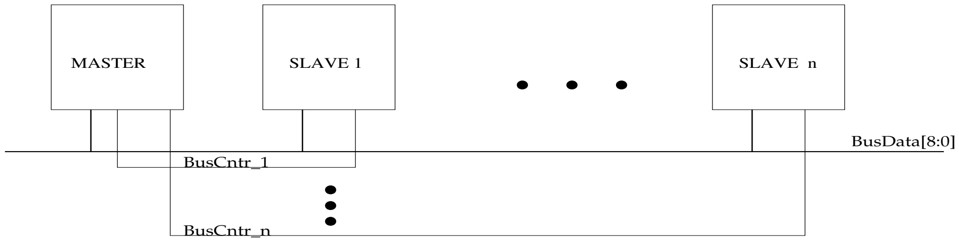


Figure 2.2: The Modified Rambus

The Rambus could be modified in order to overcome this limitation by using a separate *BusCtrl* line for each slave. In the modified Rambus, shown in Figure 2.2, only the addressed slave would see the beginning of a new request packet, while the other slaves keep monitoring the bus, so they can see a new request packet even if it issued right after the previous request packet.

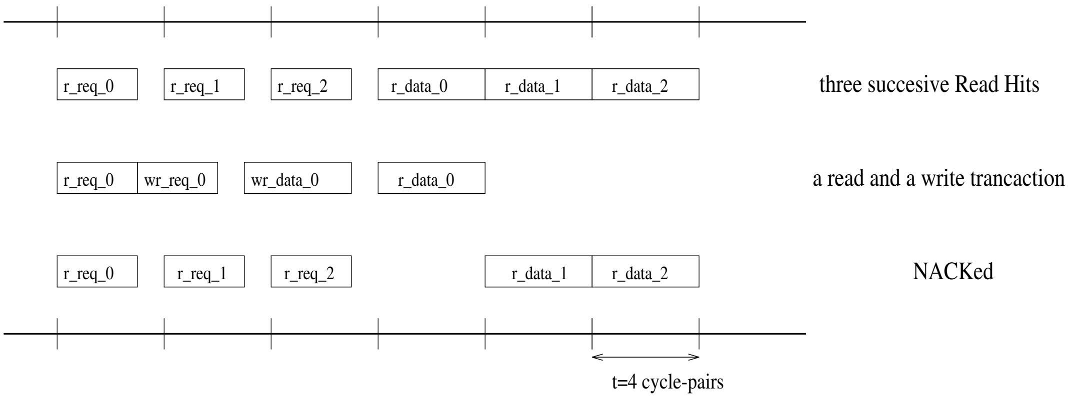


Figure 2.3 :Interleaving on the Modified Rambus

The kind of *interleaved* accesses possible with this modified Rambus are shown in Figure 2.3. For a 64-bit data packet we could have a rate of one request every 8 cycle-pairs. As we shall see later the original, unmodified Rambus, can achieve a rate of one request every 9 cycle-pairs. So the idea of the modified Rambus was finally rejected, since we achieve only a small increase in performance at the cost of tripling the bus width, by having 32 separate *BusCtrl* lines.

2.2 Interleaving on Rambus and effect on Memory Controller.

2.2.1 Request Scheduling

The controller must ensure that there will be no conflicts on the Rambus Channel, by scheduling the request and data packets in proper bus cycles. If request interleaving is not allowed, conflicts can be avoided simply by starting a new transaction after the end of the previous one. In an *interleaved* Rambus, however, the problem is more complicated because a new request can be issued in the middle of some previous transactions. In this case, the controller should know the state of the Rambus Channel: which time slots are free and which are reserved, because an already active transaction will use them, as well as the timing requirements of the waiting requests in order to decide whether each of them would cause any conflicts. Since the delays between the request,acknowledge, and data packets are fixed during the system operation, the timing requirements of each request are fully specified if we know the size of the data packet and whether the request will be a *hit* or a *miss*.

In section 2.3 we shall describe several ways to model the Rambus state and how these affect both the scheduling algorithm and the hardware complexity. In section 2.3.4 we shall describe the model chosen and the resulting scheduling algorithm.

2.2.2 Request Queues

Let us examine now what kinds of requests can be interleaved and how this affects the design of the controller. After a *miss* request any other request, destined to the *same* bank will also result in a miss, if it is issued *before* the Read/Write MissDelay. If it is issued *after* that interval, it may result in incorrect execution, if it is accessing the same word as the original miss request. So, like traditional interleaving, interleaving requests destined to the same memory bank is not meaningful, since it may result in waste of bandwidth or incorrect execution. So the only requests that can be interleaved are those destined to *different* banks. To make this possible the controller should keep a separate queue for every memory bank in the system.

Reordering of the requests destined to the *same* bank could reduce the number of misses. For example consider the sequence of requests rA, rB, rC, rD access the rows 1, 2, 1, 2. The number of misses is reduced from 3 to 1 by reordering this sequence into rA, rC, rB, rD. The requests that access the same *row* need not be reordered, since this will not affect the number of RDRAM misses. This guarantees that the requests accessing the same memory words will always arrive to the RDRAM *in order*, so there exists no danger for incorrect program execution.

Out of order execution could possibly cause a request to wait in the queue for a very long time though, because it would be bypassed by later requests that would be favored by the reordering algorithm. Thus the reordering algorithm should include a mechanism to prevent starvation.

Out of order execution may increase performance, but it would also increase the complexity of the hardware, since it would require special reordering hardware per request-queue. For the above reasons, we did not implement the idea of trying to improve further the bus utilization by reordering the requests destined to the same bank. In our controller requests destined to the same bank enter a FIFO queue, requests are only removed from the head of the FIFO only when they are finally served.

As we shall see next, the scheduling algorithm needs to know, whether a request will result in a *hit* or a *miss*, before it is transmitted. This can be easily done with the FIFO structure described above: Each time a new request enters a FIFO its row address is compared to the row address of the previous tail request in order to determine whether the new request will produce a miss in the RDRAM. This one bit information is also kept in the FIFO.

2.2.3 Return Data Logic

Interleaving may cause out of order execution of the requests addressed to different banks. So, the controller receives the read data packets from the Rambus Channel, in the order that the corresponding read requests send to the Channel, which may be different from the order that the requests were issued by the processor. If the controller delivers the data to the processor out of order with respect to the processor read requests, the processor will not be able to distinguish the read responses from each other. There are two alternatives : The *first* is that the controller puts the data packets in the correct order before returning them to the processor, at the cost of adding extra delay cycles to some read requests. The *second* is to return the data to the processor out of order with respect to the processor requests. In this later case we need a mechanism for the processor to distinguish the read responses from each other. The second alternative was adopted.

Each processor request contains an extra field with its ID: a small number of bits identifying the request. In the case of a read request, the controller will have to match the returned data packet to its ID and return them both to the processor. The mechanism used for matching the data packets with their corresponding ID's is based on the fact that the Rambus Channel will return the data packets in order, since the *ReadDelay* will be the same for all RDRAMs in the system. When a request is sent to the Rambus, we put its ID in a fifo. If varying data packet sizes are allowed, then the packet size must also be kept in the FIFO. When a data packet is returned, the controller matches the data packet with the first ID in the queue.

2.2.4 Controller Overview

In this section we discussed the additional features that a memory controller should have for performing access interleaving over the Rambus Channel. The controller should keep a separate fifo for each memory bank in the system. It should also keep the latest requested row for each memory bank in order to predict for each request whether will be a hit or a miss. The processor will have to add an ID to each request in order to distinguish the read responses because they arrive out of order. The scheduling algorithm will affect the complexity of the scheduling logic and the desing of the fifos. The Implementation of the various parts of the chip will be discussed in detail in Chapter 3.

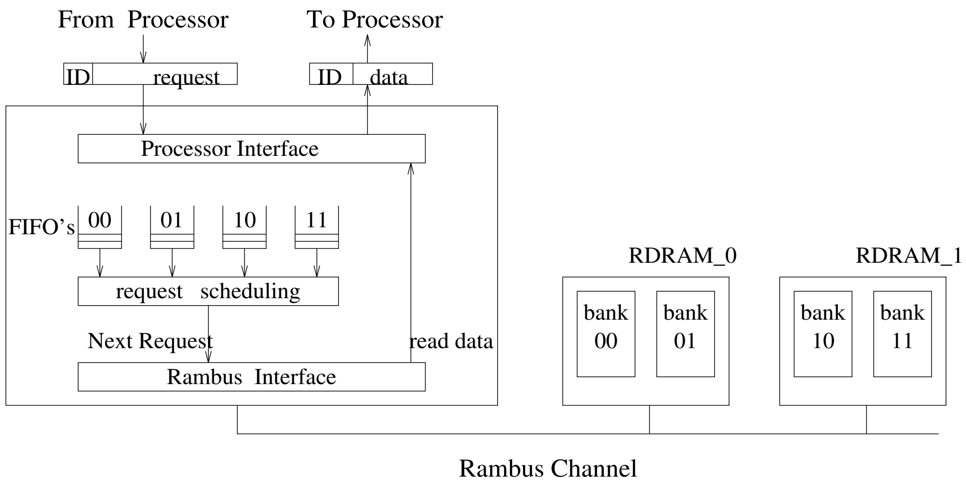


Figure 2.4: Controller Overview

2.3 The scheduling algorithm

The scheduling algorithm determines which of the waiting requests will be sent next on the Rambus Channel. The scheduling algorithm should maximize the bus utilization by interleaving requests, whenever this is possible, and also ensure that there will be no conflicts on the Rambus Channel by scheduling the request and data packets in proper bus cycles.

Avoiding conflicts becomes more complicated with request interleaving, since a new request maybe scheduled in the middle of some previous transactions, so some future time slots may be *"reserved"* for the read data packets, corresponding to some previously initiated requests. The scheduling logic should keep those future reservations in a data structure and have a mechanism to determine whether each of the waiting requests would cause any conflicts if it was scheduled next. The way we model the bus reservations will affect both the size and the complexity of the data structures and the resulting scheduling algorithm.

The scheduling algorithm can be decomposed into two parts: The *first* part finds the *"candidates"* among all the waiting requests. A *"candidate"* is a request that can be scheduled in the next cycle without causing any conflicts on the Rambus Channel. The *second* part selects the request to be scheduled next, among the candidates.

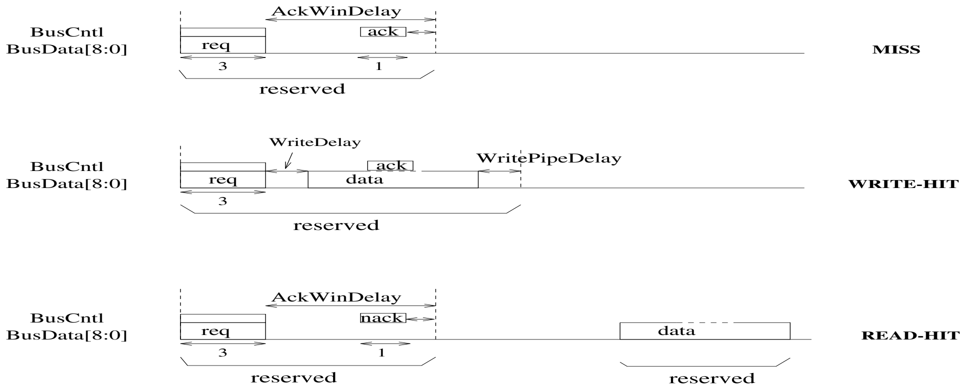


Figure 2.5: The bus reservations

The sizes of the reserved request intervals is *3+AckWinDelay* for read-hit and miss requests and *3+WriteDelay+data_packet_size+WritePipeDelay* for the write-hit request

The future reservation patterns are very much depended on the Rambus protocol. For example, the beginning of the furthest future reserved interval can not start later than *ReadDelay* since it corresponds to some read data packet. The fact that the *miss* transaction is terminated at the end of the acknolwedge window, and then the request has to be repeated until is is a hit, may waste some bandwith by repeating the request, but on the other hand it simplifies the bus-reservation scheme. Otherwise, we would have a very late data-packet, for each a read-miss, thus the range of the future schedule would be much larger than it is now. This would increase the size of the data structures and the complexity of the scheduling algorithm since the future reservation patterns would be far more complicated. In the following paragraphs we shall describe the reserved intervals for each type of request, and we show what future reservation patterns are possible on Rambus.

As we saw in Chapter 1, a *miss* or a *write-hit* transaction occupies the bus for the entire transaction. On the other hand, a *read-hit* transaction occupies the bus for two separate intervals, one for the request and one for the data; between these two intervals the bus is free. The sizes of the above intervals are shown in Figure 2.5: In this figure the intervals marked "reserved" are those when no new request can be issued, because of confict with the already scheduled request, acknowledge or data packets.

In the non-modified Rambus and with the delays for the 4.5Mbits RDRAMs we can only interleave read-hit transactions with other read-hits and misses. Also we can have at most *two* outstanding read-hits: Having *three* outstanding read-hits would require a *ReadDelay* of 19 cycle-pairs (76ns), which is too long for a read-hit, and it is also not supported by the 4.5Mbit RDRAMs. In Figure 2.6 we can see the kind of interleaved transaction we can have on the Rambus Channel: On the right of the vertical axis we can see the transactions currently scheduled and the future reservation intervals, while to the left we can see the past *read* requests corresponding to some the expected data packets. The most complicated reservation pattern is having one request interval at the beginning of the future schedule, and one or two data intervals.

In the following sections we present several ways to model the Rambus State and show how these will affect the scheduling algorithm and the harware complexity.

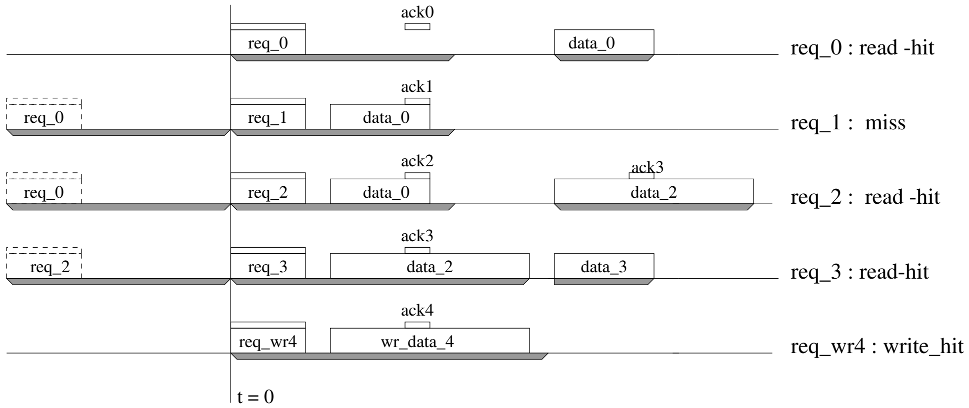


Figure 2.6: Access Interleaving on Rambus

2.3.1 First alternative

The future reservations of the Rambus Channel can be accurately modeled by using a shift register, the *Rambus Reservation Register*. Each bit of the register corresponds to a Rambus-cycle-pair and it is set if the corresponding Rambus cycle-pair is free. Similarly for each possible new request we can have a bit mask – one bit per cycle pair - which represents the times at which this request will need the Channel in the future, if this request is issued now. We can easily determine whether such a request can be scheduled at the current time by comparing the contents of the Rambus Reservation Register with the above bit mask. If there is any overlap in the reserved bus-cycles, that request will cause a conflict, if it is scheduled at the current time.

We can keep an array with one entry per request queue, each entry holding the timing requirements of the first request in the queue. We can find all candidate requests by comparing *in parallel* the contents of the *Rambus Reservation Register* to the contents of *each* entry in the array. The results can be placed on a candidate register or can be directly fed to the circuit that will select among all the candidates the request to be scheduled at the current time.

In the request bitmask the size of the request interval will be 4 cycle pairs (= *request_packet_size* + *bus_turnaround*) unlike Figure 2.5. This is because the acknowledgement

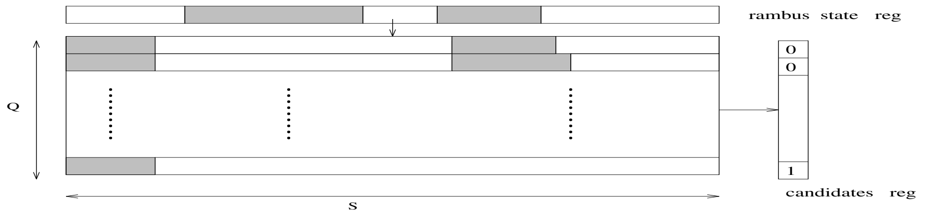


Figure 2.7: The CAM alternative

packet can be interleaved, with data packets. And since a request packet can only be found in the first 3 bits of the RRR, this representation will cause no conflict.

Every cycle the *Rambus Reservation Register* is updated: Each time a new transaction is started the old contents are ORed with the bitmask (as shown in Figure 2.5) corresponding to the selected request to give the new contents, and the bits corresponding to past Rambus cycles are shifted out of the RSR.

This model requires a content addressable memory (CAM) with Q entries, where Q is the number of request queues kept by the controller. The size of each entry, which is equal to the size of the Rambus Reservation Register, depends on the maximum length of a transaction. For the 4.5Mbit RDRAM it is 22 cycle-pairs for a data packet up to 64 bits and 146 cycle-pairs for a data packet up to 256 bytes.

For the ES2 1 micron technology the CAM will be very expensive since it can only be designed using static flip-flops and gates, because the ES2 does not provide a CAM generator, as for other regular structures such as ROMs RAMs and PLAs. We estimated a 64x22 CAM would be $22mm^2$, for a chip of $36mm^2$ the CAM would occupy the 60% of the chip area. For other technologies however, the cost of the CAM is much lower: We estimated that for scmos 1 micron teschnology the size of the CAM array would be $0.334mm^2$, that is only 1% of the chip area of a $36mm^2$ chip. Our conclusion is that for the ES2 1 micron technology the cost of the CAM is prohibitive, for other technologies it is feasible.

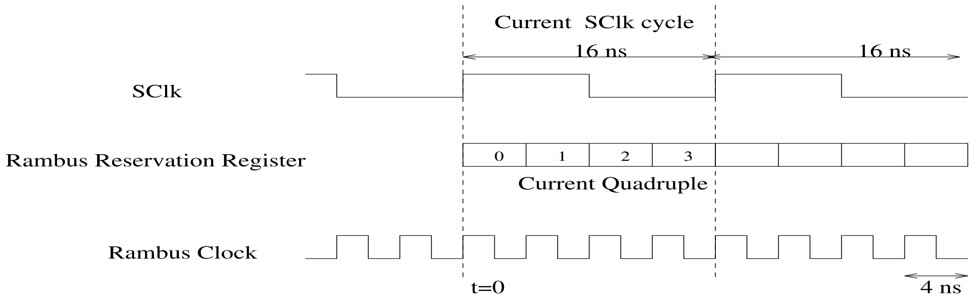


Figure 2.8: The Slower Synchronous Clock.

Since the scheduling algorithm deals with Rambus cycle-pairs, the scheduling logic should operate with a clock that is a multiple of the Rambus clock. Let us assume that the scheduling logic operates with SClk, which is 4 times slower that the Rambus clock. So each SClk cycle corresponds to a quadruple of Rambus-cycle pairs, as shown in figure 2.8. In a single SClk cycle the scheduling algorithm should examine, whether a new request can be initiated in *any* of the 4 Rambus cycle-pairs of the current quadruple. Considering the 4 alternatives cannot be done serially, since it would require the scheduling algorithm (for each alternative) to operate in 4ns. Thus 4 copies of the scheduling logic are required, that would operate in parallel, each considering a different Rambus cycle pair as the beggining of the RRR, and each having a separate copy of the CAM. The copy corresponding to earliest Rambus cycle-pair will have the highest priority, in case more than one copies have candidate requests, so the request scheduled in the earliest cycle-pair will be chosen.

If the data packet sizes, and the programmable delays are such that the request packets are aligned to SClk cycles, then there is no need for multiple copies: The scheduling hardware will consider starting a new request *only* at the first Rambus cycle-pair of the current quadruple. In this case we can further reduce the hardware, by assigning one bit of the Rambus Reservation Register for 4 Rambus cycle-pairs, not one bit per Rambus-cycle-pair, as we assumed so far. In order to avoid collisions we assume that a quadruple will be marked as reserved if at least one of its 4 cycle-pairs is reserved. This would make the reserved intervals seem larger, thus it may result in lost cycles. In this reduced scheme the size of the data structures is reduced by 16 compared to the scheme discussed in the previous paragraph. This scheme will be sufficiently accurate only if most intervals, and data packets are multiples of SClk.

The present alternative models the Rambus future reservations very accurately, and can handle variable packet sizes. The scheduling algorithm is able to detect any possibility of interleaving so is does not result in waste of bandwidth. As a matter of fact it is a general reservation scheme, that does not take into account the special characteristics of the Rambus transactions, and it is also insensitive to any changes in the data packet sizes or the RDRAM delays. On the other hand, its generality is also its disadvantage because it is far more expensive than any other of the alternatives we shall present next. For example, the alternative chosen, gives the same request schedule as the current alternative, but needs only one bit for keeping all the necessary information about the bus' future reservations, also it does not require extra storage for each request queue, as the CAM is this alternative.

2.3.2 Second alternative

In this alternative we use again a *Rambus Reservation Register* similar to the previous case. We can schedule a new request at the current time, only if there is a free interval, at the "beginning" of the *Rambus Reservation Register*, which is long enough for the new request. Using a priority encoder, we can count the size of that interval - let this be F. Knowing F we can tell if any miss or read hit request can be scheduled at the current time. For a write hit we also need to know the data packet size. These are illustrated in Figure 2.9. We can schedule:

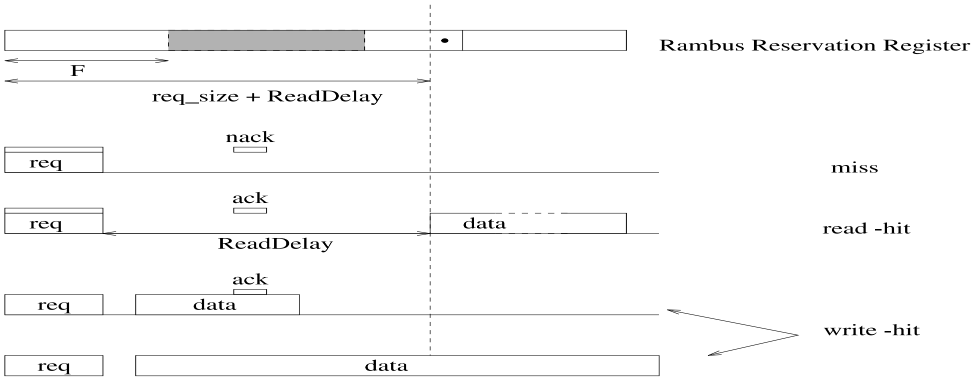


Figure 2.9: The third alternative

[a] a miss request if *request_packet_size* < F.

[b] a write hit if *(request_packet_size+WriteDelay)* + *data_packet_size* <F.

[c] a read hit if *request_packet_size* < F and the cycle-pair at *(request_packet_size+ReadDelay)* is free: Since all read requests have the same *ReadDelay* the data packet corresponding to some previous read request can not start later than the packet of the current request.

Since the *request_packet_size* and the *ReadDelay* are constant, this scheme allows us to know whether a *read-hit* or a *miss* request can be scheduled, by using only the information of RRR. Allowing variable data packet sizes imposes individual checks, for write-hit requests, to determine whether they can be scheduled at the current time. The advantage of this alternative over the previous one is that the only information needed by the scheduling algorithm, for each waiting request, is the type of the request (R/W, hit/miss) and the size of the data packet. This information, except the hit/miss bit, is already found in the request packet, so no extra storage is required, as the CAM in the previous alternative. On the other hand, the algorithm requires Q parallel comparisons for the write data packets, thus Q extra comparator. The comparisons are not needed if the packet size is fixed.

For the 4.5 Mbit RDRAMs the *ReadDelay* is not long enough for having any write-hit transaction interleaved with a previous read-hit. We can only have up to 16 bit write-hits, and this is only if we use the maximum allowable *ReadDelay* (12 cycle-pairs). So unless we care for 16-bit data transfers, the comparisons can be totally be eliminated and write hits maybe scheduled only when the bus is free, which can be detected by the priority encoder.

The algorithm also adds the delay of a priority encoder in the calculation of candidates. As in the previous section the scheduling logic can operate with SClk. If requests and data packets are alligned to SClk cycles, each bit of the RRR can be a quadruple of Rambus cycle-pairs. Otherwise each bit of the RRR corresponds to one cycle pair, and the RRR consists of 4 shift registers, as shown Figure 2.10. Figure 2.10 shows the RRR at the time that a read-hit, with

AckWinDelay 6 cycle-pairs and *ReadDelay* 10 cycle-pairs, is scheduled. In the free time slot before the data_packet a new miss or read request can be scheduled starting from (1,3). However if we used a single bit for each quadruple the first five quadruple would appear as reserved.

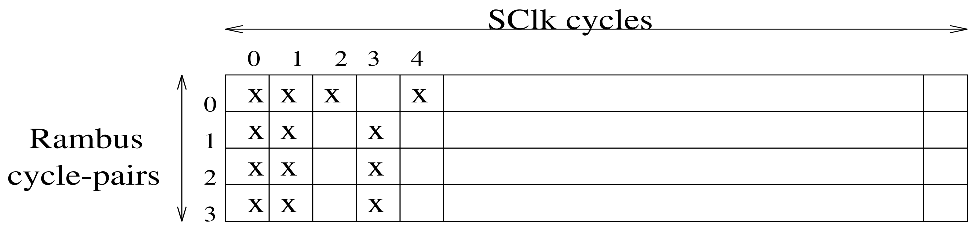


Figure 2.10: The Rambus Reservation Register

2.3.3 Third alternative

As we saw in 2.3, the most complicated reservation pattern that may appear on the Rambus Channel is having one request interval and one or two data intervals. In general, we can have zero to one request intervals, always starting at the beginning of the future schedule, and zero to two data intervals. Since we have a small number of intervals, we can use pointers to mark the beginning and the end of each reserved interval instead of having a bitmask with one bit per Rambus cycle-pair, as the Rambus Reservation Register that was used in the previous alternatives.

In our case five pointers are required: one to mark the end of the request interval, and two for each data interval. As shown in figure 2.11.a *t0* marks the end of the request interval, while *t1* and *t2* mark the first data interval, and *t3* and *t4* mark the second data interval.

Figure 2.11.b shows how the pointers are adjusted when a new request is scheduled, and the kind of request interleaving we can have with this alternative. The scheduling algorithm will be simplified if we make [*t1,t2*], to be always the earliest interval, by swapping the value of *t1* with *t3* and *t2* with *t4*, when *t2* becomes zero. This is illustrated in the last example of Figure 2.11.b.

The pointers can be implemented by counters, that are set each time a new request is scheduled, and are decremented every clock cycle. The scheduling algorithm is shown in Figure 2.12. The scheduling algorithm would require 5 parallel comparisons: each pointer will be compared to zero to determine the number of the reservation intervals. For example if *t0*>0, *t1>0* and *t3*=0 then there is one request interval, [0,*t0*], and one data interval, [*t1,t2*]. In parallel with the above comparisons, 3 more comparisons are performed to determine if each waiting request may be scheduled at the existing intervals (see b.1 and b.3 in Figure 2.12). Then the results are combined to give the *candidate* requests. Thus the algorithm requires 7 comparators, which operate in parallel.

In this alternative we assume that the *ReadDelay* is not long enough for having a write-hit transaction interleaved with a previous read-hit, which is the case for the 4.5Mbit RDRAMs,as we

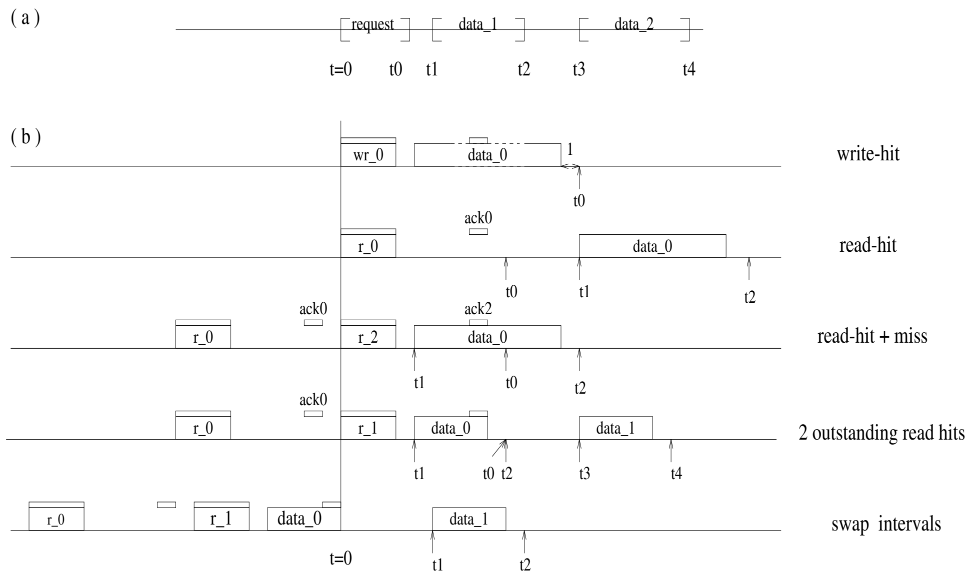


Figure 2.11: The pointer solution

have already explained in 2.3.2. If this is not the case one more comparison would be required to determine whether a write hit can be scheduled. (see b.2 in figure 2.12). This comparison involves the size of the write data packet, thus it has to be performed seperately for each request. In this case the algorithm would require 5+3+Q comparators where Q is the number of fifos kept in the controller.

---

[a] If t0>0 : The bus is busy transmitting the previous request.

[b] If t0=0 & t1>0: The bus is free but there are some future reservations. A request can be scheduled if it does not cause a conflict.

1. Miss request if *request_size* < t1.
2. Write Hit request if *request_size+WriteDelay+Data_packet_size* < t1 (optional)
3. Read Hit if *request_size*<t1 and *request_size+ReadDelay* > t2
   and *request_size+ReadDelay+Data_packet_size* < t3

[c] t1=0 & t2>0: The data is being returned to the processor.

[d] t1=0 & t2=0: The bus is free, any request can be scheduled.

Figure 2.12: The scheduling algorithm.

As the pointers count bus cycle-pairs, the system should run with a clock that is a multiple of the Rambus clock, for example SClk. In this case the pointers can still hold Rambus cycle-pairs, but must be decremented by 4. This would allow requests to be issued only at multiples of SClk cycles. If the size of delays and data packets allow request packets to be alligned to SClk this scheme is efficient, otherwise it will result in lost cycles. This can be overcome by considering whether a request can be scheduled in any of the 4 cycle-pairs of the current quadruple, not only the first one. This would increase the comparison circuits, by 4.

The advantage of this model is that it requires only five counters to model the state of bus, and a few comparisons (Figure 2.12) to determine whether a request could be send or not.

2.3.4 The alternative chosen

One reason for the complexity of the models described in the previous sections is that we assumed more than two outstanding requests, and a variable packet size. For the delay-range of the 4.5 MBit RDRAMs,however, having three outstanding requests is impossible, since it requires a ReadDelay of 19 cycle-pairs. On the other hand, the shortest value for *ReadDelay* that allows interleaving is 10 cycle-pairs. This is only one cycle-pair above the minimum *ReadDelay*, and allows interleaving two requests.

The range of the programmable delays for the 4.5Mbit RDRAMs and the delays chosen in this scheme are shown in Table 2.1. As shown, the delays have their minimum values except *ReadDelay* because a *ReadDelay* of 9 cycle pairs does not allow interleaving.

Register	range	value
AckDelay	4-7	4
AckWinDelay	5-12	5
ReadDelay	9-12	10
WriteDelay	1-4	1

Table 2.1: Programmable Delays for the 4.5Mbit DRAM.

If the length of data transfers is not fixed the only kind of transaction that has a fixed size is a miss transaction, both read and write transactions may have data packets of varying size and this would complicate the scheduling algorithm. For this reason we assume a fixed data-packet size of 64 bits (transfer time 4 cycle-pairs). This corresponds to a computer system like e.g. a supercomputer, where interleaving elements is used to support individual word accesses (for example non-cached vector elements with random stride), rather than cache-line requests,i.e. accesses to blocks of multiple contiguous words - which can use directly the Rambus protocol without need for interleaving. In a system with mostly cache-line accesses, interleaving would only be interesting for the case of RDRAM misses.

The resulting request timing and the kind of interleaving that we can achieve with the present scheme is shown in Figure 2.13. In this scheme a new request can be scheduled every 9 cycle pairs, giving a data bandwith of 220Mbytes/sec (64bits/36ns). The only limitation we have is that a *read-hit* cannot be followed by a *write-hit*. So the only Rambus state information required, is whether there is an outstanding *read-hit*. A single flag RH can be used, which will

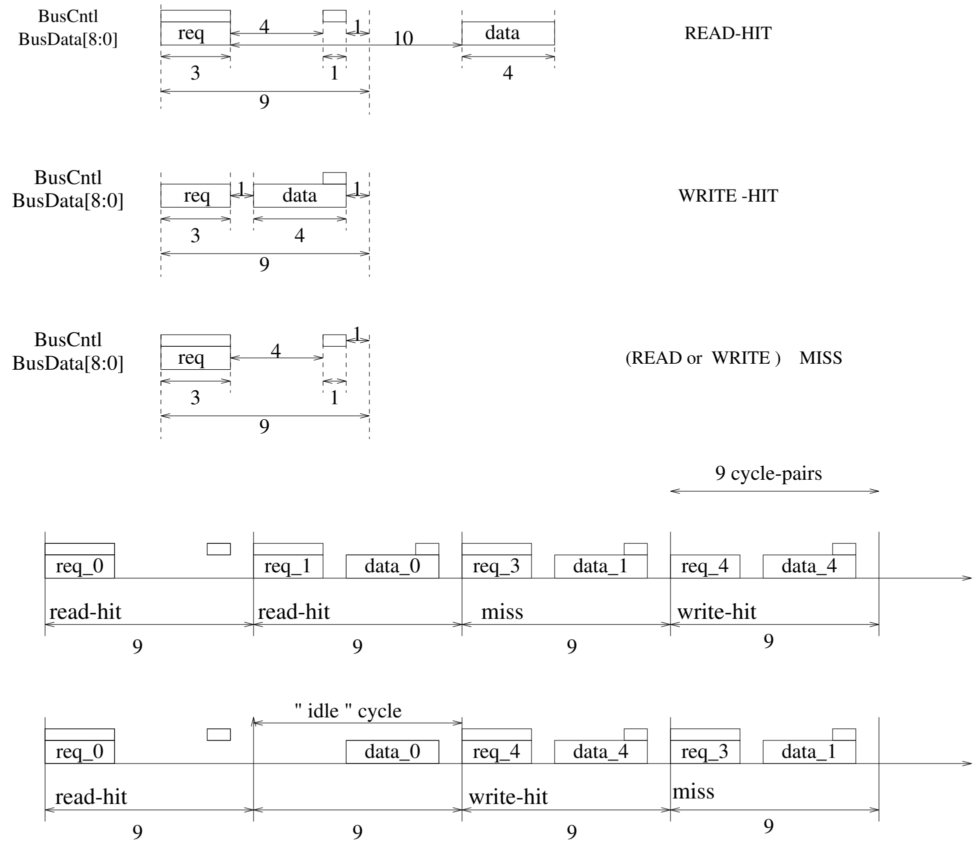


Figure 2.13: The alternative adopted

be set in the above case. Only the type of the request is needed to determine the candidates. Getting the candidates is straitforward:

If RH is not set any waiting request can be scheduled.

If RH is set only a Read-Hit or a Miss can be scheduled.

The scheduling logic requires a clock with a frequency 9 times the Rambus clock. This provides the Rambus Interface with 60 bits of request, and 64 bits of data every 9 cycle pairs. The scheduling logic could also operate with SClk, at the cost of adding an extra cycle-pair to Read-Delay and scheduling one request every 10 cycle-pairs.

Selection logic

The second part of the scheduling algorithm is to select among the candidates the request that will be scheduled at the next cycle. A simple selection scheme is to scan the request FIFO's circularly, and select the first candidate we found. In our case, each time a *write-hit* is found, and the RH is set, we shall have to wait for one cycle before scheduling the write-hit. In order to reduce these idle cycles, we define two classes of candidates: The first contains the read-hits and the misses, while the second contains the write-hits. Instead of scanning the queues once, before wrapping around, we perform two scans, one for each class. This way we have at most one idle cycle for each pair of scans. A more detailed desciption of the scheduling and the selection algorithm and their implementation is given in section 3.5.

Chapter 3

Implementation

In this chapter we discuss the implementation of our RDRAM controller. First, we present the available library (§ 3.1). Then follows an overview of the chip design and operation (§ 3.2). The rest of this chapter describes the implementation of the various parts of the chip: section 3.3 describes the processor interface, section 3.4 the implementation of the scheduling algorithm, and section 3.5 the FIFOs for keeping the processor requests.

3.1 The available library

The technology used in this design is 1.0−micron CMOS, with double metal and a single poly layers, provided to us by the company European Silicon Structures (ES2) through the CEC-funded "EUROchip" program. The library includes standard cells such as gates, 2-to-1 and 4-to-1 multiplexors, 1 and 2-bit adders, and edge triggered flip-flops and latches with various combinations of reset and clear. The gates available as standard cells are: 2− to 4−input *nand* and *and*, 2 and 3 input *nor* and *or*, AOI and OAI, 2 input *xor* and *xnor*, and *inverters buffers* and *tristates buffers* with different driving strengths. A macrocell library is also available with more complex gates and MSI cells built out of standard cells. There are also megacell compilers for generating the following megacells: RAM, dual-port RAM, ROM, PLA, asynchronous FIFO, Multiplier, and DataPath.

For the design of time critical blocks, we used the *nand* gates, the 2−input *nor* gate, the *inverters* and the AOI and OAI gates. The rest of the gates were excluded because they were slower than synthesizing them using the above gates. For example the 2−input *and* gate is almost 2 times slower than a *nand* followed by an *inverter*. The "slower" standard cells were preferred over the faster combinations of multiple standard cells in the design of non−time−critical blocks, because they allow us to save both in total area and in the number of standard cells, which also reduces the area required for routing. Some standard cells (2−to−1 multiplexor, register, tristate buffer) are also available as groups of 4. The 4-bit cells were slightly slower than the corresponding 1-bit cells but resulted in significantly smaller circuits.

In our design we had to include multiplexors of all sizes (2−to−1 up to 64−to−1). For designing those muxes, we had to choose among the 2-input multiplexor, the 4-input multiplexor, and two tristate buffers with different driving strengths. The 4−to−1 multiplexor was excluded in cases where speed was important, because it was slower than two 2−to−1 multiplexors in series. The tristate buffers seemed to be faster than the 2-input multiplexor, but they had smaller driving strengths and resulted in bigger circuits so the 2-input multiplexor was preferred in most cases. We also used the dual port RAM compiler for builting a 128x96 dual port RAM. Our design included a 128x96 dual-port RAM but since the maximum word size supported by the corresponding megacell compiler was 64, we had to use two separate dual-port RAM megacells: 128x64 and 128x32.

3.2 Chip Overview

This section is a brief overview of the chip. A more detailed description is given in the remaining sections of this Chapter.

3.2.1 Chip Interface

The controller is placed between the processor and the memory. The chip interface signals are shown in figure 3.1. The top signals are used for the processor interface, while the bottom signals are the Rambus Channel. The 16 Rambus interface signals were described in section 1.2 so in this paragraph we shall discuss only the processor interface signals. The processor sends the requests and write data on pins *req[0:35]* and the corresponding validity signal on *vreq* pin. The controller sends a control signal to the processor, on the *ack* pin, to denote whether it can accept a new processor request in the next cycle. The read data are returned on the pins *read_data*[0:63] together with the ID of the corresponding request on pins *ID[0:9]* and a validity signal on pin *vdata*. We assume that the *clock* and *reset* signals are provided by the processor.

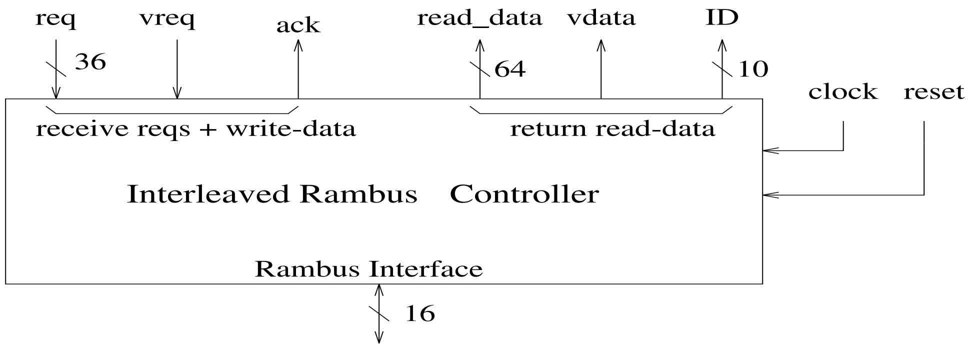


Figure 3.1:The chip interface signals

The controller's scheduling logic clock has to be a multiple (9 times) of the Rambus clock, in this work we assume this clock is provided by the processor, thus the processor interface also runs at the same clock. We also assume that read request are supplied in one processor cycle while write requests in three processor cycles. Since the scheduling algorithm can schedule one request every processor cycle, the processor interface becames the bottleneck. This could be easily cured by modifying the processor interface to increase the throughput of the writes. One way to do this is by adding 28 more bins and transfer the write data the cycle following the write-request. This increases the write-throughput to one write every 2 cycles. The write-throughput can be further increased to one write every cycle by assuming that the processor sends new data at both clock edges (the write data will be send at the negative clock edge).

3.2.2 Chip operation

As we have already seen, the controller keeps a number of FIFOs (one for each memory bank of the system) for storing the processor requests. The controller receives from the processor the requests and write data packets. The requests and the corresponding write data enter the FIFO assigned to the addressed memory bank, and stay there until they are serviced (they are scheduled as *hits* to the RAMBUS Channel). The order in which the requests are serviced depends on the scheduling algorithm. The read-data are returned to the processor four cycles after the corresponding read request is serviced. The block diagram of the chip is shown in Figure 3.2. The chip constists of five blocks: the *InputLogic*, the *FIFO Logic*, the *FIFO memory*, the *Scheduling Logic* and the *Rambus Interface* circuit. In the rest of this section, we discuss the interface and the function of each of the above parts in order to give an overview of the chip operation, before presenting the function and the implementation of each part in detail.

The *InputLogic* handles the reception of processor requests and read data packets: The processor requests are sent on the *req[0:35]* pins in one cycle. After a write request, the processor sends the write-data on pins *req[0:31]* in the two subsequent clock cycles. The controller latches the input data at the rising clock edge. If the input corresponds to a *read* request then this request will be presented to the FIFO Logic right after the latching clock edge. If the input corresponds to a *write* request, then it will be presented to the *FIFO Logic* on cycle later, as we shall explain further in section 3.3.

The controller can hold up to 64 FIFOs with two elements each. Since each request requires 96 bits of storage (32 for the request plus 64 for the write data), 128x96 memory bits are required to hold the processor requests. This number of bits is too big for implementing them using discrete flip-flops; this leds us to use a RAM. If less than 64 FIFOs are required the RAM is partitioned into fewer blocks, so that the size of each "active" FIFO is increased for effective utilization of the available memory. For example, in a system with 32 RDRAMs, all 64 (2-element) FIFOs are used, while in a system with 16 RDRAMs only 32 of the available FIFOs are used, but they contain 4 elements each.

The *Scheduling Logic* requires 3 bits of information from each FIFO:

- the RW-bit denotes whether the first request in the FIFO is a read or a write.
- the HM-bit denotes whether the first request in the FIFO will be a hit or a miss
- the EMPTY-bit denotes whether the FIFO is empty.

The signals RW[0:63], HM[0:63], and Empty[0:63] comprise the *Qstate[0:191]*. The EMPTY bits and the RW/HM bits have to be kept in registers (not in the FIFO memory like the rest of the request), because the *scheduling algorithm* requires them to be available in parallel.

The *Scheduling Logic* uses the *Qstate[0:191]* to determine which request will be scheduled. The selected request (*NextReq[0:63]*) is supplied, in a decoded form, to the *FIFO Logic*: *NextReq[0:63]* is a bitmap indicating the FIFO whose head request is scheduled. *NextReq[0:63]* is

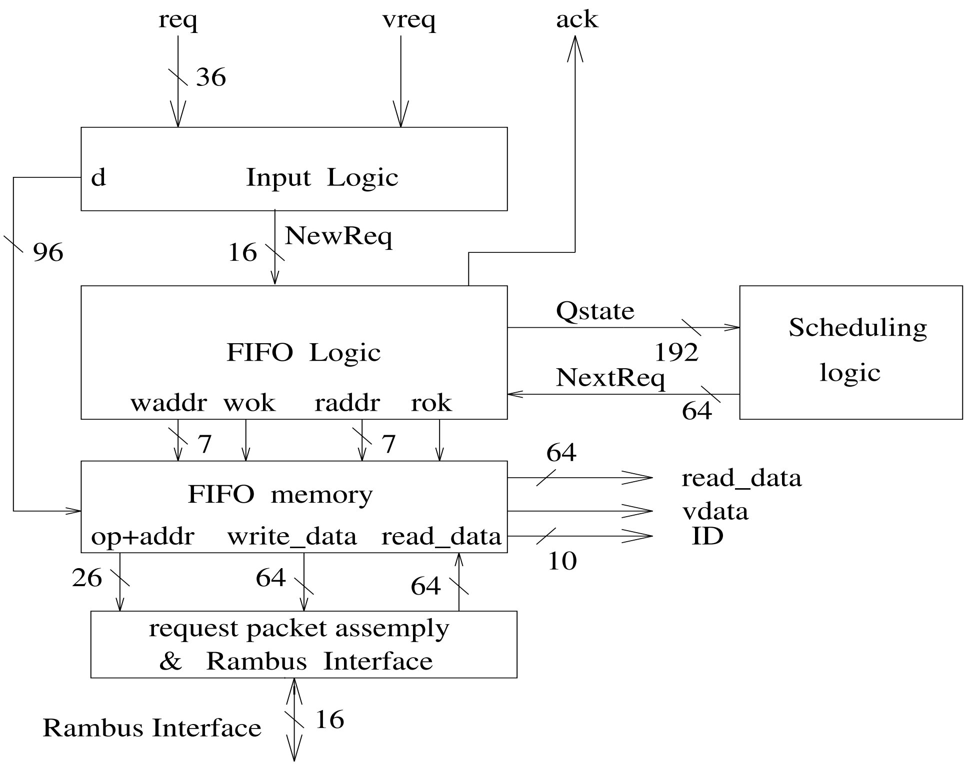


Figure 3.2: Block diagram of the memory controller

used to updated the *Qstate[0:191]* and calculate the SRAM address of the scheduled request (*raddr[0:6]*).

The FIFO's and their manipulation circuits are broken into two blocks: The *FIFO Logic* block includes all the logic required for the FIFOs: the Head and Tail registers and the *Qstate* registers and the logic required for the FIFO operations. It also includes the logic for hit-miss prediction (*Row Logic*). The *FIFO memory* block includes the FIFO's RAM, the corresponding timing and control logic, and some address and data registers.

Each clock cycle two separate FIFO operations may be initiated: One request may enter the FIFO (enqueue), and another request may be scheduled (read or dequeue). Each operation (enqueue/dequeue/read) requires two cycles. In the *first* cycle the memory addresses corresponding to the request to be enqueued/dequeued/read are calculated in order to be presented to the *FIFO memory* in the next clock cycle: The *waddr[0:6]* is the memory address corresponding to the request that will be enqueued (written to the memory) while the *raddr[0:6]* corresponds to the scheduled request (that will be read from the memory). In the *second* cycle of each FIFO operation the actual memory access takes place. The FIFO operations are pipelined: The first cycle of an *enqueue* operation can overlap with the second cycle of a previous *enqueue*. Similarly first cycle of a *dequeue* or *read* operation can overlap with the second cycle of a previous *dequeue* or *read*. Thus the FIFOs' throughput is one *enqueue* operation and one *dequeue* (or *read*) operation per clock cycle. This corresponds to one memory read and one memory write from the FIFO memory every clock cycle, thus a dual port RAM is required.

The *Qstate[0:191]* is updated during the first cycle of each FIFO operation, i.e. a request appears to be enqueued (dequeued) in the FIFO, before it is actually written to (read from) the memory. This reduces the minimum number of cycles that a request stays in the FIFO: Since the *Qstate[0:191]* is updated in the first cycle of the *enqueue* operation, the corresponding request can be scheduled in the next cycle, because it will be visible to the *Scheduling Logic*. Thus each request has to stay in the FIFO only one cycle before it is dequeued.

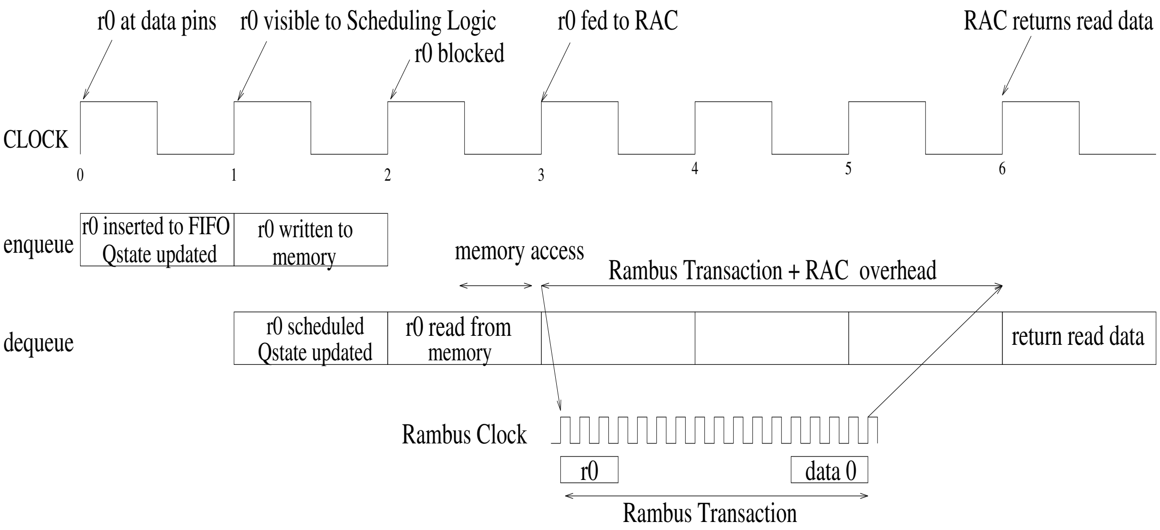


Figure 3.3: Controller operation

At positive edge 0 the request r0 appears in the req pins. At positive edge 1 the Qstate is updated, so the r0 is visible to the Scheduling Logic. Thus r0 maybe scheduled any cycle after positive edge 1. r0 is presented to the Rambus Interface one cycle after it is scheduled.

After a request is read from the FIFO memory, it is fed to the *Rambus Interface* that will transmit the request to the Rambus Channel. The *Rambus Interface* block contains both the RAC cell and the circuits required for interfacing the RAC. In the case of a *write* request, the *Rambus Interface* will transmit both the request and the write data. In the case of a *read*, the Rambus Interface will return the read data to the processor after some known delay time. The delay of the interface has no influence on the design of the controller. As we did not have a detailed description of the Rambus Interface we assumed that the data will be returned 3 cycles after the request is sent to the RAC: two cycles for the Rambus read transaction and one cycle for the interface overhead.

Figure 3.3 shows the timing of enqueue and dequeue operations and the extra cycles added to the latency of a read request. The earliest time that the read data can be returned to the processor is six cycles after the read request appeared on the controller's input pins. The cost of each the request going through the request FIFO is three cycles. In case all request-FIFOs are empty, the processor requests could be forwarded from the *Input Logic* FIFO to the Rambus Interface, bypassing the memory–write and memory–read stages. Forwarding would reduce the latency of a read hit request by two cycles, giving a latency of four cycles for the forwarded read-hits.

Since the *Scheduling Logic* is bypassed in the case of forwarding, some extra logic would be required to control the request scheduling, so that the forwarded requests would not cause any conflicts on the Rambus Channel, and the (forwarded) read requests destined to the same bank are not interleaved. This logic would include a register holding the type (read/write, hit/miss) of the request provided to the *Rambus Interface* in the previous clock cycle, and the memory bank corresponding to that request. It would also include some comparison logic, to determine whether the current request can be scheduled right after the previous request.

3.3 Processor Interface

The processor interface can be decomposed into two parts: the *input* part handles the reception of the processor requests and the write-data, and the *output* part returns the read-data to the processor. In the following paragraphs we shall describe the input part of the processor interface, and in 3.3.1 we shall describe in detail the *Input Logic* block that implements this part of the processor interface. The output part of the processor interface is integrated into the *FIFO memory* block and will be described in section 3.5.7.

The communication between the processor and the controller is synchronous: all data exchanged are valid around a rising edge of the processor clock. The processor requests are 36 bits wide: they constist of a 24-bit address field, a 2-bit opcode field, and a 10-bit ID field. The format of the processor requests is shown in Figure 3.4. We assume that the processor sends the requests in one cycle, on the *req[0:35]* pins. The write data (64-bits) are sent on the same pins (*req[0:31]*), in the two consecutive cycles following the corresponding write request cycle. So a *read* is send in one cycle while a *write* in three cycles.

	OP		ADDRESS		
ID	R	W	BANK	ROW	COLUMN
35:26	25	24	23:18	17:10	9:0

Figure 3.4:The format of the processor requests

Each request received by the controller is enqueued into the FIFO assigned to the memory bank addressed by that request. If that particular FIFO is full, the request can not be enqueued, and it is temporarily stored in a special FIFO, kept by the *Input Logic*, until there is a place in the appropriate request FIFO. In this case the controller clears the *ack* signal, to indicate that it can not accept any more requests. As soon as the processor receives the "negative" *ack* it stops sending requests. The processor will restart the transmission of new requests as soon as *ack* is set (this will happen when the waiting request is finally enqueued).

It is the responsibility of the controller to ensure that no request or write-data is lost because of overflow in the controller's request FIFOs. So the controller should store all the requests that were sent by the processor after the FIFO overflow and before the processor receives the negative *ack*. In this work, we assume that the processor will not stop the transmission in the middle of a write (3 cycles). This assumption simplifies the FSM of the *Input Logic* but increases the size of the *Input Logic* FIFO by two entries.

3.3.1 Input Logic

The *Input Logic* receives the processor requests and the write-data, provides the necessary information to the *FIFO Logic* for enqueueing the request, and provides the *FIFO memory* with the request and the corresponding data packet. It provides the storage required for not losing request packets, in the case of request overflow, and some extra registers for the assembly of the request and the corresponding write data packets until they are presented to the *FIFO memory*.

Interface Signals

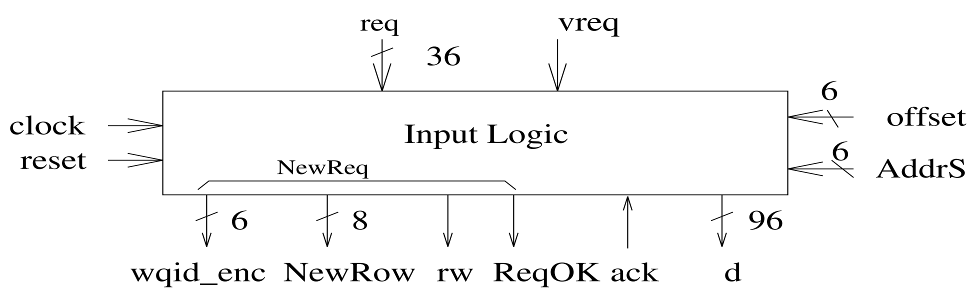


Figure 3.5 : The Input Logic interface signals

The interface signals for the *Input Logic* are shown in Figure 3.5. The processor requests and the write-data are placed on pins *req[0:35]*, as we have already explained, while pin *vreq* is used for the corresponding validity signal. The input request is analyzed before it is presented to the *FIFO Logic*. If the input is a *read* request, it is presented in the same cycle that the request is received, while a *write* request will be presented to the FIFOs logic at the next cycle.

The signals that refer to the request that is presented to the *FIFO Logic* (*NewReq*) are: *wqid_enc*, *NewRow*, *rw*, *reqOk* and provide all the necessary information for the first stage of the enqueue operation (updating the *Qstate* and calculating *waddr* – the SRAM address for *NewReq*). More specifically:

- *wqid_enc[0:5]* is the FIFO address assigned to the memory bank accessed by the new request.
- *NewRow[0:5]* is the row address accessed by the new request.
- *rw* (1 bit) indicates whether the new request is a *read* or a *write*.

• *reqOk* (1 bit) is the validity signal for the above signals.

Ack is a control signal returned by the *FIFO Logic*, indicating whether the new request is actually enqueued. The only case when a negative *ack* is returned is when the corresponding FIFO (specified by *wqid_enc*) is full.

The *d[0:95]* field contain the request and the write-data that will be written into *FIFO Memory*, and is delayed one cycle with respect to the signals presented to *FIFO Logic*.

Offset and *AddrS* are parameters that are set during system initialization.

Offset[0:5] indicates the real number of FIFOs, and it is equal to the smallest power of 2 that is larger or equal to the actual number of banks of the memory system.

The RDRAMs provide the option of remapping the Rambus addresses to the RDRAMs, so that consecutive row addresses are mapped to consecutive memory banks, thus increasing the locality of the second level cache. The new address is produced by swapping individual bits of the bank (*Addr[23:18]*) and row (*Addr[15:10]*) fields of the initial address, under the control of the *AddrS[5:0]* value. More specifically, the row bit Addr[10+i] and the bank Addr[18+i] are swapped, if the *AddrS[i]* bit is set. Since our controller allows this option, it has to perform this remapping as well, in order to estimate the actual row and bank addresses.

Circuit Description

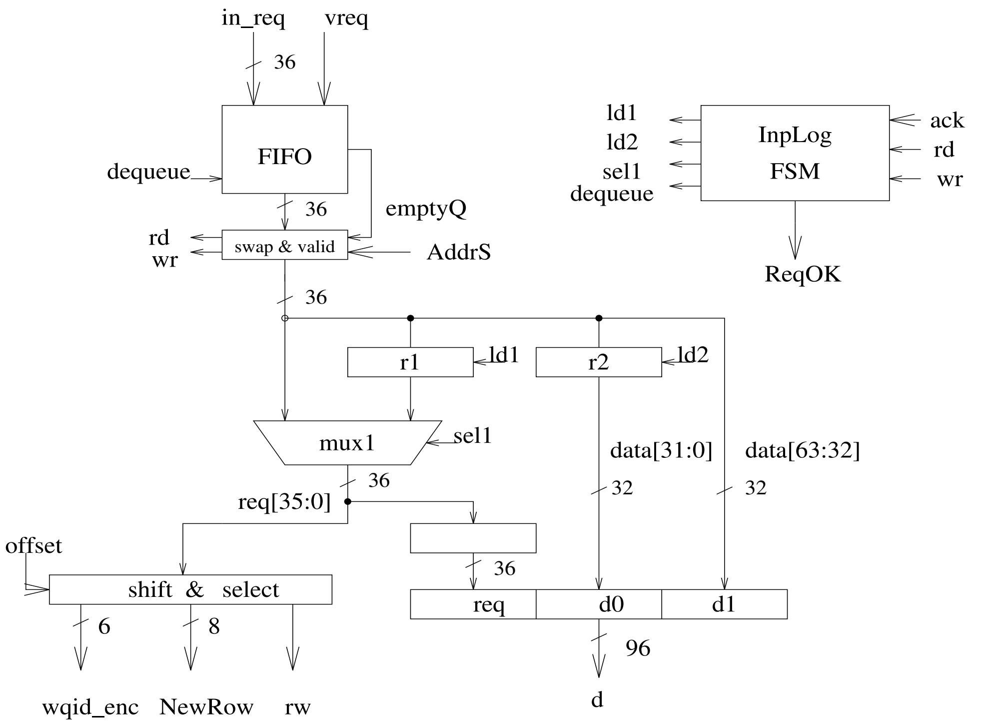
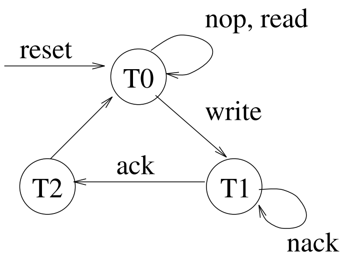


Figure 3.6: Block Diagram of the Input Logic

The block diagram of the *Input Logic* is shown in Figure 3.6. It includes the logic required for the processor interface, that is the registers for holding the input requests and write data packets, and a 3-state FSM for receiving the write data. The state diagram of this FSM is shown in Figure 3.7. The *Input Logic* also does the assignment of the request address to a request-FIFO. This is done by the *swap&valid* and the *shift&select* blocks:

In the *swap&valid* block the bits of the *row[5:0]* and the *bank[5:0]* fields of the request address are swapped according to the values of the corresponding bits of AddrS[5:0] parameter, giving the *NewRow[5:0]* (=*req[15:10]*) and the *NewBank[5:0]* (=*req[23:18]*) values. Also the request opcode is set to "nop" if *vreq* is not asserted.

The *shift&select* part assigns the input request to a FIFO. The FIFO address is a function of the *NewBank[5:0]* and the *Offset* values more specifically : *wqid_enc* = *NewBank* << (*Offset*-1).



$ld1 = T0 \& write$

$ld2 = T1$

$sel1 = T0$

$ReqOK = T0 \ \& \ read \mid T1$

$dequeue = \overline{ReqOK \ \& \ \overline{ack}}$

Figure 3.7: State Diagram of the Input Logic

T0 : The input data correspond to a request T1 : The input data are the first half of the write data. T2 : The input data are the second half of the write data. The read requests are presented to FIFOs Logic at T0 while the write requests at T1.

The *Input Logic* FIFO is used for storing the processor requests in the case of a request-FIFO overflow so that no data is lost. As shown in Figure 3.6 the input data always go through the FIFO, that is the FIFO is also used as a register for the input data. When there is no overflow in some request-FIFO the *Input Logic* FIFO has only one element: the data latched at the current cycle. In the following paragraph we shall describe how this FIFO was designed so that it does not add to the overhead cycles imposed by the controller.

Since the validity signal (*vreq*) and the request (*req[0:35]*) are sent by the processor in the same clock cycle, if we allowed only valid data to enter the *Input Logic* FIFO then the input data would have to be enqueued at the next clock edge (because we need time to check the (*vreq*) signal) thus the input request would be visible to the *FIFOs Logic* with a delay of half clock cycle. Another alternative is letting non valid requests (nops) to be enqueued as well. The later solution would not increase the FIFO's size, but it would delay the requests that would be inserted after the nops from being served. For the above reasons we have implemented a modified FIFO (made of discrete registers and counter cells): Each cycle the data are inserted to the tail of the FIFO but the tail pointer is advanced only if the inserted data are valid. So any invalid data will be overwritten in the next cycle. This modified FIFO requires that req[op] = NOP for non valid requests, because if some invalid data are inserted in the FIFO's head, these data may be visible (in the same clock cycle) by the *FIFOs Logic*. This is enforced by the *swap&valid* block as we have already explained.

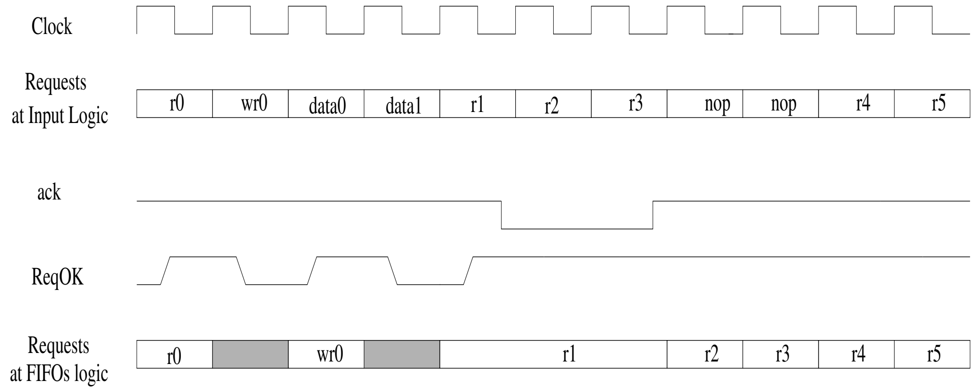


Figure 3.8:Communication protocol

Reads are visible to the FIFOs Logic the same cycle they enter the chip, while writes with one cycle delay. The request r1 can not enqueued because the corresponding FIFO is full. The processor will stop transmitting the requests two cycles later. After two cycles the request can be enqueued and the processor will start transmitting two cycles later.

The current processor interface forces the write-requests to be writen to the SRAM one cycle later than the read-request, because they have to wait until the write-data are available. If the write requests are provided to the *Scheduling Logic* one cycle after they enter the controller (like read request), the SRAM will have to be bypassed, if a write is scheduled as soon as it is visible by the *Scheduling Logic*. In this implementation the write request is visible by the *Scheduling Logic* with one cycle delay with respect to a read request. By delaying the writes for one cycle we favor the scheduling of reads over the writes.

3.4 The Scheduling Logic.

3.4.1 The Scheduling Algorithm

Definition of candidates

In Chapter 1 we defined as *candidates* the requests that can be scheduled for the next cycle without causing any conflicts on the Rambus Channel. For finding the *candidates* we consider all the FIFOs that are not empty and not *blocked*. A FIFO can be *blocked* for two reasons : After a *miss* request is scheduled the corresponding FIFO is blocked for MISS_DELAY cycles, to guarantee that the next time that request will be repeated it will be a *hit*. Thus we avoid wasting bandwidth by repeating miss requests, and we know when we schedule a request whether it will result in a *hit* or *miss*. The *second* reason for blocking a queue is when a *read-hit* request is scheduled. In this case the queue is blocked for next cycle, since we assume we cannot interleave requests destined to the same RDRAM.

With the given time intervals and data packet lengths, a *read-hit* request can not be followed immediately by a *write-hit* request. If we absolutely must do so there must be a idle cycle between the two requests. In order to reduce the idle cycles, we try to minimize the number or read-hits followed immediately by write-hits. This is achieved by having two classes of requests. The *first* class contains the read-hit and any-miss (read and write miss) requests while the *second* class contains write-hit requests. The algorithm has two phases of operation. In each phase we only consider candidates who belong to one of the above classes.

Circular scanning and Phase Switching

In the current implementation we scan the queues from right to left. Initially the scan pointer is set to the rightmost queue. If a request is scheduled the pointer is moved to the left neighbour of the selected queue. We switch phase when there are no requests of the current type to the left of the pointer, and there are requests of the opposite type. When we switch phase the pointer is reset to the rightmost queue. In case there are neither current−type requests to the left of the pointer, nor opposite−type requests; we reset the pointer and stay in the current phase.

The above description of the algorithm was written in "sequential" fashion for clarity; in reality, all of the above searches are performed in parallel, in the same clock cycle, so that if there is any candidate request we are guaranteed to make a selection within this single cycle.

The only case that we need to have an *idle cycle*, due to Rambus limitations, is when there is a switch from "read-phase" to "write-phase" and the RH flag is set (the RH flag is set if the request that was scheduled in the previous cycle was a *read-hit*). The number of idle cycles can be further minimized if the number of scans for each phase is increased. Another possibility is to have more scans for one of the phases, –e.g. two scans for read-phase and one scan for write – or some other combination, based on the percentage of requests that we expect to have for each phase. This decision depends on the specific application, and could be made after analyzing the performance of the various configurations. In the present design we adopted the simplest solution: we perform one scan for each phase.

3.4.2 Circuit Description

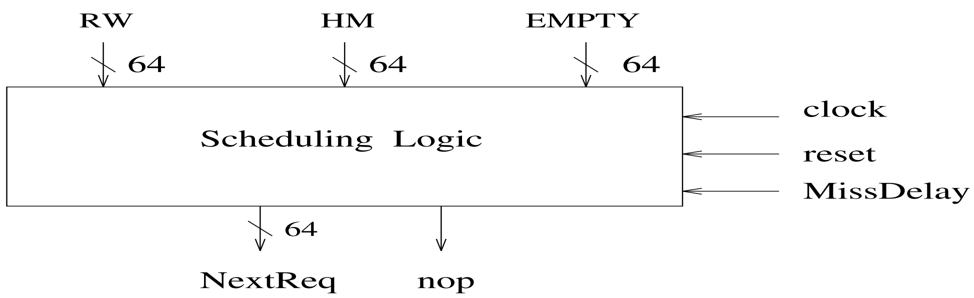


Figure 3.9:The scheduling interface

The scheduling logic uses the *Qstate* provided by the *FIFO Logic* in order to produce the request to be scheduled in the next cycle. The interface signals for the *Scheduling Logic* are shown in Figure 3.9. The signals RW, HM and EMPTY comprise the *Qstate* as explained in 3.2.2. The *MissDelay* is a parameter that is set during system initialization and corresponds to the number of cycles that a miss request should wait before being serviced. The *NextReq* and *nop* outputs are fed to the *FIFO Logic*: *NextReq[0:63]* indicates (in a decoded fashion) the request FIFO whose head request is scheduled in the next cycle. The *nop* bit is set when there is no candidate to be scheduled currently.

The block diagram of the *Scheduling Logic* is shown in Figure 3.10. There are two pointers one for each phase: *RTurn* for read-phase requests, and *WTurn* for write-phase requests. The pointer of the current phase is set to the left of the request that was scheduled in the previous cycle. The pointer of the other phase is set to the rightmost request. The pointers are implemented using 64-bit registers having set the bits to the left of the "pointed" FIFO. We define four types of Candidates.

- A-Read : read-phase candidates to the left of RTurn.
- B-Read : read-phase candidates to the right of RTurn.
- A-Write : write-phase candidates to the left of WTurn.
- B-Write : write-phase candidates to the right of WTurn.

There are 4 priority encoders, one for each type of candidates. The input of each priority encoder is the 64-bit candidates of the corresponding type. Each priority encoder has two outputs: a 64-bit output that gives the rightmost candidate of his class and an carry-out bit indicating if there were any candidates in this class. These four carry-out bits together with the phase-bit and the RH-bit are used to determine which queue is finally selected and the next values of

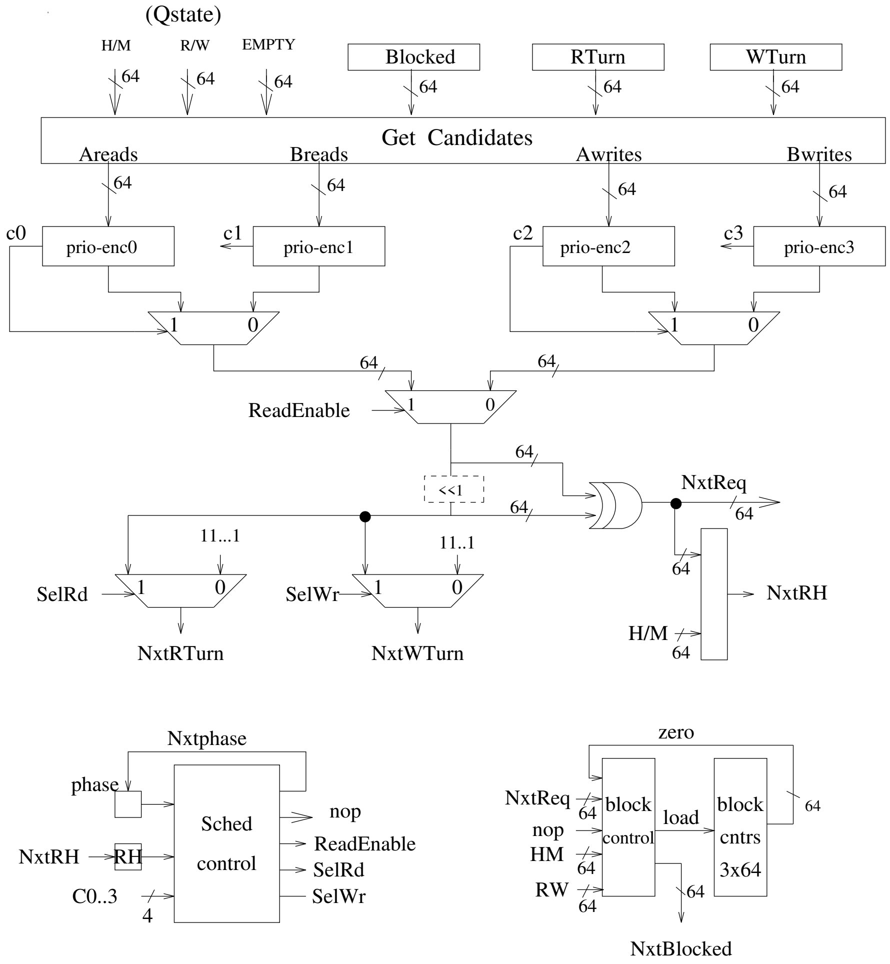


Figure 3.10:The Scheduling Logic

RH	phase	c0	c1	c2	c3	RdEn	nop	switch	NxtrTurn	NxtwTurn
x	0	0	0	0	x	x	1	0	reset	reset
x	0	0	1	0	x	1	0	0	<<	reset
0	0	0	x	1	x	0	0	1	reset	<<
1	0	0	x	1	x	x	1	1	reset	reset
x	0	1	x	x	x	1	0	0	<<	reset
x	1	0	x	0	0	x	1	0	reset	reset
x	1	0	x	0	1	0	0	0	reset	<<
x	1	x	x	1	x	0	0	0	reset	<<
x	1	1	x	0	x	1	0	1	<<	reset

Table 3.1

R/W-pointers and phase/RH bits, as shown in Table 3.1.

In the above scheme one priority encoder is redundant since the B-type candidates of the inactive phase are always 0...00. We could modify this scheme and use only three priority encoders: Two priority encoders for the A- and B-type candidates of the "active" phase, and one priority encoder for the A-type candidates of the "inactive" phase. In this case a 64-bit 2−to−1 multiplexor would be required (to select the input candidates of the appropriate phase) for each priority encoder. This alternative was rejected for two reasons: The three 64-bit multiplexors added, take up almost triple standard cell area than the single priority encoder eliminated, and the delay of a 2−to−1 multiplexor is added to the critical path.

Implementation of the priority encoder

The priority encoder is a circuit that selects the 1 with the highest priority from a sequence of 1s and 0s. In the following we assume that the priorities are increasing from right to left, thus the priority encoder will select the least significant 1 of the input. The circuit has two inputs: *A[N-1:0]*, *Cin* and two outputs *Y[N-1:0]*, *Cout*. The input *A[N-1:0]* can be any sequence of 0s and 1s. The output *Y[N-1:0]* is a sequence of 1s followed by a sequence of 0s, having its least significant 1 in the same place as the least significant 1 of the input. The *Y* output is given by the following equation: Y[i] = A[i] + A[i-1] + ... + A[0] + *Cin*. We can see that Y[N-1] is the OR of all N+1 input bits, and is set if any of the *Cin*, A[i] is set. *Cout* is identical to Y[N-1].

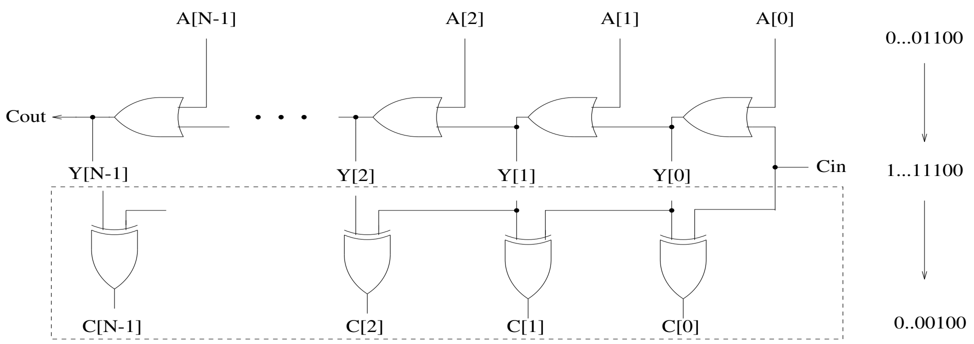


Figure 3.11: Typical priority encoder

A straitforward implementation of the priority encoder is shown in Figure 3.11. It constists of a chain of OR gates and (optionally) a chain of XOR gates: Output *Y[N-1:0]* can be fed to the XOR circuit that produces *C[N-1:0]*, that has only one bit is set: the one corresponding to the least significant 1 of the input (*A[N-1:0]*). The delay of the above priority encoder is N-1 OR-gates + 1 XOR gate. In our design the 64-bit priority encoder required is on the critical path so we tried to minimize the delay of the priority encoder by using a carry look-ahead tree. The tree structure (width, height) was dictated by the gate-delay of the available library.

The priority encoder is broken into 16 4-bit priority encoders. The 4-bit priority encoder is shown in Figure 3.12. It operates on 4 bits of the input (*A[0:3]*) and produces 4 bits of the output(*B[0:3]*). It also produces a lookahead carry (*G*) that is the OR of its 4-bit inputs (G = A[0] + A[1] + A[2] + A[3]). G is fed to the lookahead circuit that will eventually produce *Cin*. *Cin* is the input carry corresponding to the output carry of all the least significant (4-bit) priority encoders. *Cin* is then used to produce the 4-bit output: B[i] = (A[i] + A[i-1] +...+ A[0]) + Cin.

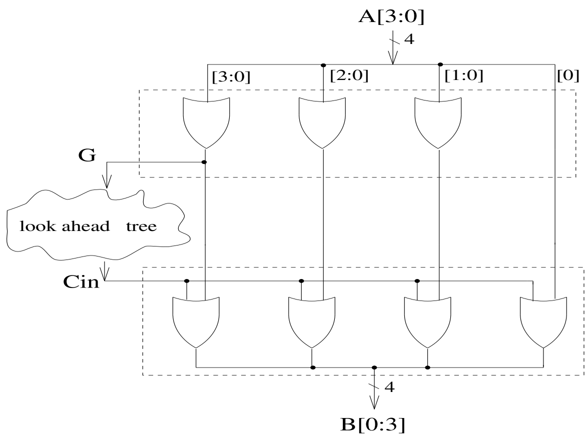


Figure 3.12: The 4-bit priority

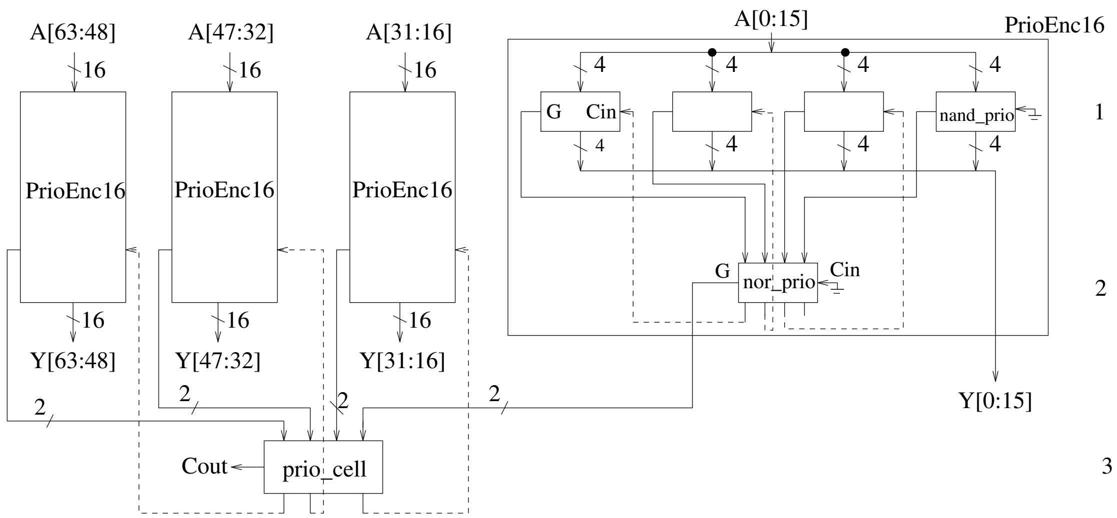


Figure 3.13: The lookahead tree

As shown in Figure 3.13, the Gs of the first level are fed to the next level which consist of 4 similar priority encoders. The only difference from the priority encoders of the first level is that 2-bit G's are produced at the second level (G[0] = A[0] + A[1], G[1] = A[0] + A[2]), in order to minimize the logic levels required for carry genaration. This is forced by the available library: In the first level we used 4-bit NANDs but there are not 4-bit NORs required for the second level. The third level consists of a single node that calculates the actual carry_in for the 3 subtrees (the carry_in of the left-most subtree is zero) Each of the 4 second−level nodes calculates the Cin of its 4 priority encoders. When the Cin is available the priority encoders of the first level will produce the output. In the above scheme, five (= 1+1+3) gate levels are required for carry generation, plus two gate levels for going back through the tree, for a total of 7 gate levels. The schematic diagrams for the priority encoder is found in the Appendix A.

3.5 The organization of the request FIFOs

In this section we describe the organization of the request FIFOs, that is the operations provided by a single request-FIFO (3.5.1), the additional logic required for each FIFO (3.5.2), how multiple FIFOs are implemented using a single dual-port RAM (3.5.3), and an overview of the FIFOs operation (3.5.4). In the rest of this section we discuss the implementation of some parts of the *FIFO Logic* and the *FIFO memory*.

3.5.1 Operation of a single request FIFO.

Each FIFO has two pointers : the *Head* pointer points to the first (oldest) element of the queue, while the *Tail* pointer indicates the first available place in the queue where the next element that enters the queue will be inserted. The FIFO supports three operations :

Enqueue : A new element is inserted in the FIFO and it is placed at the position pointed to by the Tail pointer. The Tail pointer advances to the next empty position in the FIFO. If the FIFO is full, the enqueue operation is rejected and there is no change to the state of the queue.

Dequeue : The element pointed by the Head is read, and the Head pointer is advanced to the next element of the queue. If the FIFO is empty the dequeue operation will not change the state of the FIFO.

Read : The element pointed by the Head is read, but there is no change in the position of the Head, so the element read remains at the Head of the queue. The Read operation is useful when a miss request is scheduled. In this case we want the request to remain in the queue and to be removed only when it is serviced (when it becomes a hit).

The Head and Tail pointers move circularly within the FIFO range. The two pointers point to the same position when the FIFO is either empty or full. We distinguish these two states by using an extra bit (Empty bit) which is set when the FIFO is empty. In Figure 3.14 we can see three instances of the FIFO operation. The positions of the Head and Tail pointers and the Empty bit is shown in three cases, when the queue is empty, when the first element is inserted, and when

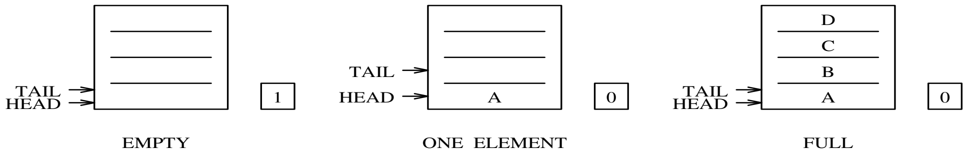


Figure 3.14: Single FIFO operation

the FIFO is full.

3.5.2 Additional logic per FIFO

Except for the Head and Tail registers and Empty bit, and the logic that handles the FIFO operations, some additional logic is required for each request FIFO: the logic required for *hit/miss* prediction, and the logic for handling the *Qstate* bits.

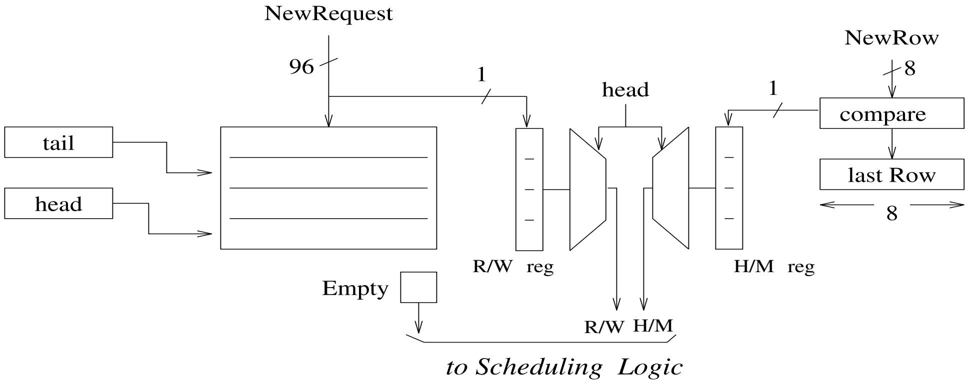


Figure 3.15: Additional logic required for a single FIFO.

In order to know whether a request will result in a *hit* or a *miss*, the row address of the last request of each FIFO is kept in a register (the *last Row* register shown in Figure 3.15). This address is compared to the row address of each new request (*NewRow*). It there is a match then the new request will be a *hit*, otherwise the new request will be a *miss*, and the *last ROW* register is updated. We will see below why this *last Row* information needs to be kept in registers outside the FIFO memory.

The scheduling algorithm requires the RW and HM bits of all FIFOs to be available to the *Scheduling Logic* in parallel. In order to do this, these bits are kept in registers rather than inside the SRAM, like the rest of the request. Among the RW and HM bits of all entries in a FIFO, only those corresponding to the current head of the FIFO are needed by the Scheduling Logic; thus these bits are supplied to the *Scheduling Logic* through a selection circuit.

As we have already explained in §3.2, in each clock-cycle only two FIFO operations may be initiated: one *enqueue* operation and one *read* or *dequeue* operation, concerning one or at most two FIFOs. Thus, it would be wasteful to implement each of the 64 FIFOs independently – this would only be needed if all FIFOs should operate in parallel. In our case a single dual port RAM, can provide the required bandwidth (one read and one write per cycle). Also the Head and Tail registers can be organized in two register files, each having two read and one write port. Similarly the Empty-bit is organized in a 64–bit register and the RW/HM bits in two 128–bit registers. The ROW registers are also organized in a single register file, and only one comparator is needed. The resulting scheme is illustrated in Figure 3.16

3.5.3 Variable size FIFO's.

As shown in Chapter 1, our memory controller interfaces directly to a single Rambus Channel that can have up to 32 RDRAMs. Since each RDRAM has two independent memory banks, the maximum number of memory banks on a Rambus with our controller is 64. The controller keeps one FIFO per memory bank – 64 FIFOs in total. Assuming each FIFO has two entries, a 128 word dual-port RAM is used to store the requests. If the memory system has less than 64 banks, our controller chip allows each of the now fewer FIFOs to grow larger, so that the entire memory is exploited. The assignment of the memory banks to the existing request–FIFOs depends on the *Offset* value, as given by the following relationship:

fifo_num = bank_num << (Offset-1)

Recall that the *Offset* value was defined as the smallest power of 2, that is larger or equal to the actual number of the system's memory banks.

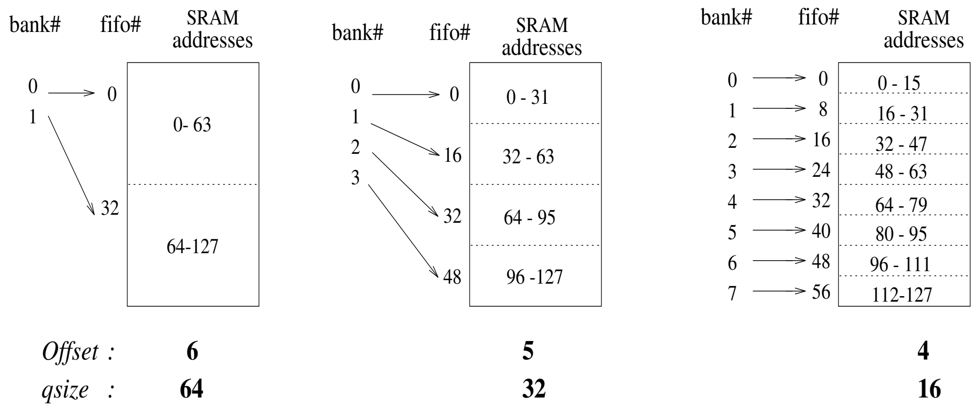


Figure 3.16: Variable size FIFOs sharing all the available SRAM

In figure 3.16 we show how the memory is fully utilized if we have a smaller number of memory banks. We can see that if there are only 2, 4, or 8 memory banks, then the FIFO size is 64, 32, or 16 respectively. The memory addresses assigned to each FIFO are in the interval:

[*fifo_num * 2, (fifo_num * 2)+qsize*].

The number of FIFOs could be reduced, for example by assigning one FIFO to two memory banks, thus reducing in half the number of FIFOs. Even without reducing the total capacity of all the FIFOs, we would still economize in silicon area, because the size of the FIFO logic, Head & Tail pointers and Empty bits would be cut down by half. The *Scheduling Logic* would be significantly reduced, as well, and so would the delays imposed by the *Scheduling Logic*. The cost of this reduction is that it would no longer be possible to interleave requests destined to any two banks that are assigned to the same FIFO.

3.5.4 Overview of the FIFO operation.

We will now present the details of the FIFO operation.

Enqueue

The new request, *InReq*, and a validity signal is presented to Qlog. The corresponding FIFO is specified by wqid_enc<0:5>. The *row* address of the new request is compared to row address of the tail request of the corresponding FIFO to determine whether the request will be a hit or a miss. At the same time, the Head and Tail registers, and the Empty bit of the corresponding FIFO are accessed to determine the state of the FIFO. If the FIFO is full, then a negative acknowledgement is returned and the *enqueue* operation is rejected. Otherwise, the state of the addressed FIFO is updated, the hit/miss bit together with the R/W bit are written to the corresponding 128−bit register, and the memory address (*waddr*) is calculated so that in the next clock-cycle the request is written to the SRAM.

Dequeue

The *Scheduling Logic* supplies the FIFO number for the Next Request to be scheduled (*NextReq*) and the *read* and *dequeue* signals to determine the kind of FIFO operation. The NextReq[0:63] is in decoded form and it used to access the Head/Tail register files and the Empty bit register. NextReq is encoded and combined with the Head value of the corresponding FIFO to give the SRAM address of the scheduled request (*raddr*). The SRAM access will take place at the next clock cycle.

3.5.5 The Head and Tail register files.

The head and tail pointers of the FIFOs are organized into two separate register files. These two register files consist of 64 registers each, and they have exactly the same organization. Since the maximum offset that we can have is 6 bits, we could implement these register files using 64 6-bit registers, 384 bits in total. In that case the bits of some registers would never be used (for any offset values). For example, the FIFO number 2 is used only for the offset value 1, thus the 5 most significant bits of the register number 2 would never be used. In the following paragraph we

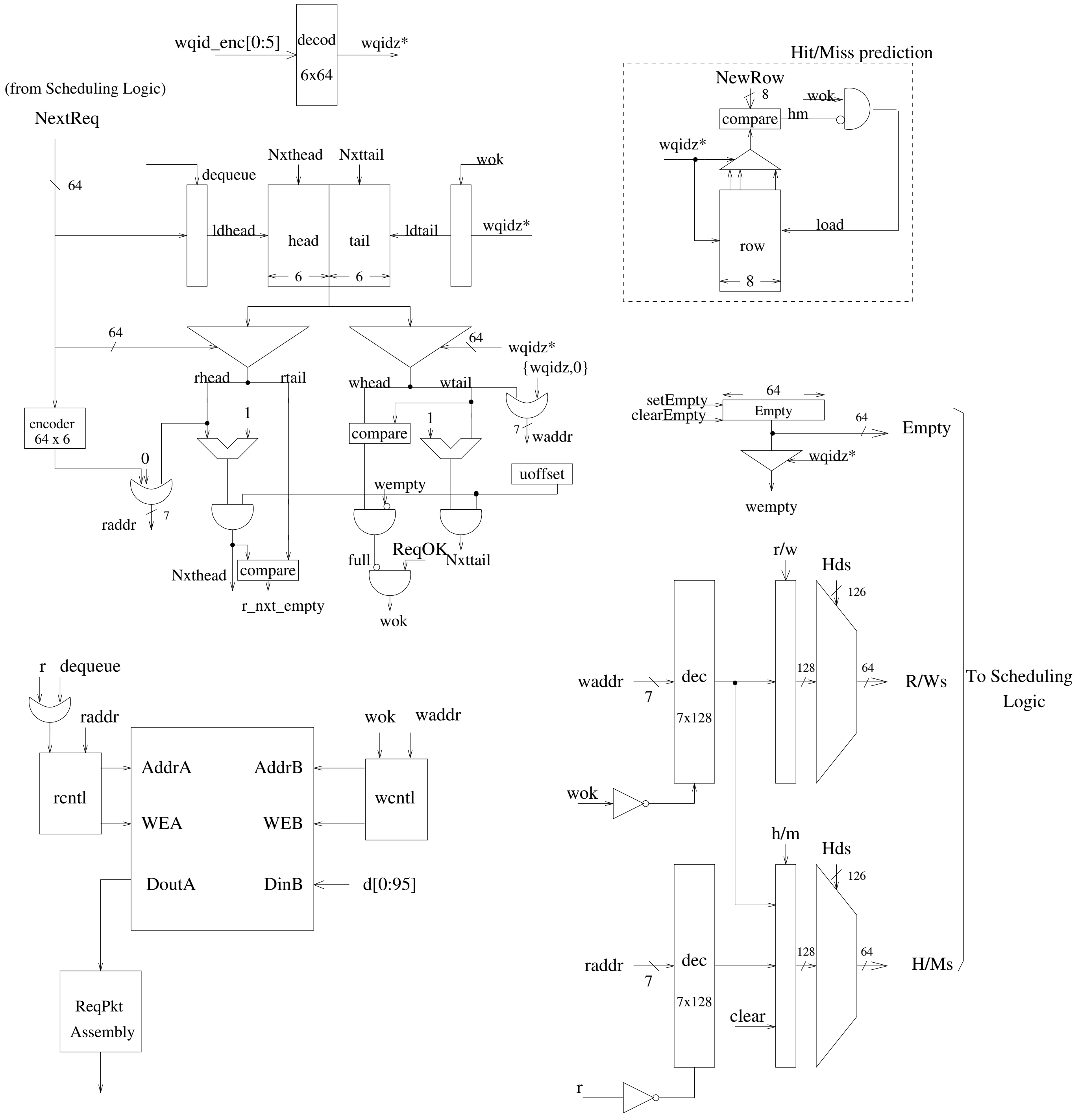


Figure 3.17: Organization of the request fifos

decribe how we can have variable size registers, and thus reduce the number of bits required for each register file to 126.

From the correspondence between the memory banks and the request FIFOs we conclude that the FIFOs number 0 and number 32 will be used for every FIFO configuration (all offset values between 6 and 1) and thus 6 bits are required for the corresponding Head and Tail registers. Similarly the FIFOs 16 and 48 will be used only for *Offset* values 5 through 1, thus only 5 bits are required for the corresponding Head and Tail registers. Simirarly working for every FIFO address we conclude that the number of bit required for each Head/Tail register file is:

126 = (2*6 + 2*5 + 4*4 + 8*3 + 16*2 + 31*1).

This is also shown in Figure 3.18: Each register file has: two 6-bit registers (register #0 and register #32), two 5-bit registers (register #16, and register #48), four 4-bit registers etc.

Each register file has two read ports and one write port. We assume that both read and write addresses are in decoded form, thus each address bit corresponds to a register. The individual bits of the *write* address provide the load enable signals for the corresponding registers. Each *read* port requires its own copy of the selection circuit shown in Figure 3.18 This selection circuit consists of 6 multiplexors that have their selection input in decoded form. As shown in Figure 3.18, the multiplexors have variable sizes: For the bit 0 a 64−to−1 mux is required, while for bit 5 only a 2−to−1 mux is required, since there exist only two 6-bit registers.

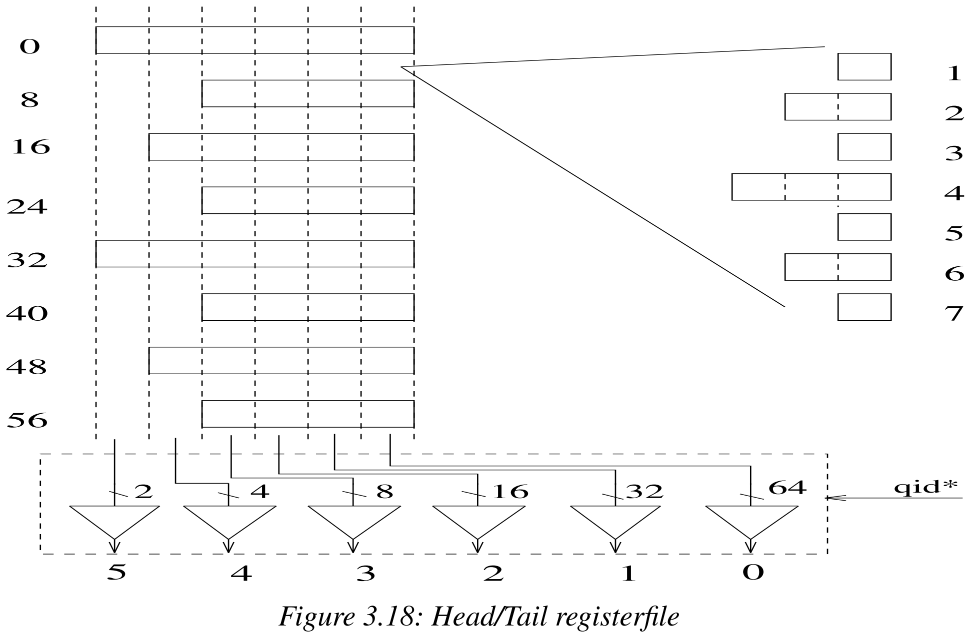


Figure 3.18: Head/Tail registerfile

3.5.6 The RW selection circuit

As shown in Figure 3.17 the RW logic includes two selection circuits, one for each 128-bit register. Each selection circuit has two inputs: the in<0:127> input that is fed form the corresponding 128bit register, and the sel<0:125> input that is fed from the Head register file. The selection circuit also has a 64-bit output, each bit of which corresponds to a FIFO. The function of the circuit is to provide each output with the RW or HM bit that is held in the Head of the corresponding FIFO. We can picture this circuit as 64 smaller selection circuits, one for each FIFO; for each of them the data input constists of all the RW or HM bits held in the corresponding FIFO, and the selection input is the FIFOs head.

This selection circuit is rather complex, because we allow FIFOs to have variable sizes. For example for *Offset*=6, there are only two request-FIFOs thus only two 64−to−1 muxes are required. For *Offset*=5 there are 4 FIFOs, thus 4 32−to−1 muxes are required. Similarly for *Offset*=1, 64 2−to−1 muxes are required.

A straitforward way to implement this selection circuit would be to have 6 groups of muxes (one for each *Offset* value) each time selecting the output form the appropriate group. In this case, 2 64−to−1 + 3 32−to−1 + 8 16−to−1 + 16 4−to−1 + 32 2−to−1 muxes are required plus 32 muxes for the selection of the appropriate group. However, there is a better way:

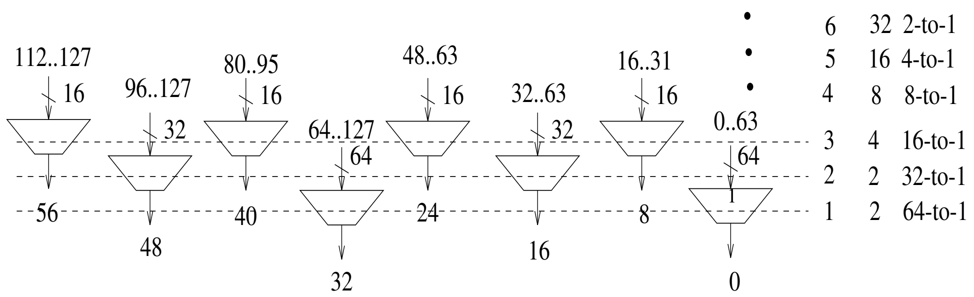


Figure 3.19:The RW selection circuit.

A 2n−to−1 mux can also operate as a mux of (any) smaller dimension if the appropriate number of the more significant of its selection bits are 0. For example, a 8−to−1 mux can operate as a 4−to−1 mux (considering only the least significant quadruple of its data input), if the most significant selection bit is 0. So, instead of having six muxes per FIFO (one for every Offset value), and then selecting the appropriate mux according to the actual *Offset* value, we can have only one mux per FIFO: the largest required.

The resulting scheme requires only 64 muxes and is illustrated in Figure 3.19. In order to clarify the organization of the muxes we classify them in 6 levels. Level one consists of two 64−to−1 muxes, level two consists of two 32−to−1 muxes, level three consists of four 16−to−1 muxes etc. If the Offset value is 1, only the first level muxes are used. If the Offset value is 2 is the fist and second level muxes are used, but in this case the muxes of the first level will operate as 32−to−1 muxes, considering only the 32 least significant bits of their inputs. For example mux #0 will consider only the bits 0..31 of its input. For Offset value is 3, the muxes of the first 3 levels will be used, all operating as 16−to−1 muxes. Finally for Offset value is 6 the muxes of all six levels will be used, operating as 2−to−1 muxes.

3.5.7 The FIFO memory

The SRAM was designed using the available dual-port RAM megacell compiler. Each dual-port RAM megacell has two independent ports: A and B. Port A can be configured either as read/write or read-only, while port B can be configured as either read/write or write-only. The operation of each port is controlled by a separate memory enable signal (ME). MEA corresponds to port A, and MEB corresponds to the port B. There is no constraint on the relative timing of the two ports except in the case of address contention (simultaneous write and read to the same address). For both read and write accesses, the addresses are latched on the rising edge of ME. For write accesses, the data are latched on the falling edge of the ME. The input to the compiler is the number of rows, the number of columns, the word size, the number of words and the configuration of each port.

Our design required a dual port RAM with 128 words, 96 bits each. Since the maximum word size supported by the dual-port RAM megacell compiler was 64, we had to use two separate megacells: One 128x64 with size (3461µm x 1791µm = 6.2mm^2) and one 128x32 with size (1957µm x 1791µm = 3.5mm^2). For both megacells port A was configured as read-only, and port B was configured as write-only. The cycle time for the 128x64 megacell is 27.7ns and the access time 16.2ns (from the rising edge of ME). Thus, clock cycle times of 36ns, or larger, allow latching the read data on the clock edge that follows the address latching edge. The timing of the memory is shown in Figure 3.21. The design of the *Input Logic* and the *Selection Logic* guarantees that there will never be any read/write contention.

In Figure 3.20 we can see the dual-port SRAM and its control logic. Since the two dual-port RAM megacells have common address and control signals, they are shown in this figure as a single block. In order to save memory we do not store all the address bit (28 bits) but only a part of it (22) since a part the rest can be derived by the FIFO address (the "rqid[0:5]" shown in Figure 3.20).

The processor interface

The *FIFO Memory* block, shown in FIgure 3.20, also implements a part of the processor interface. The read data will be returned to the processor 3 cycles after the memory access of the corresponding read request. The *FIFO Memory* keeps a small (2-entry) FIFO. Each cycle a new ID is enqueued, and the head (oldest) ID is dequeued. The head-ID (ID[0:9]) is returned to the processor together with the read-data and a validity signal (vdata), which is also produced by the

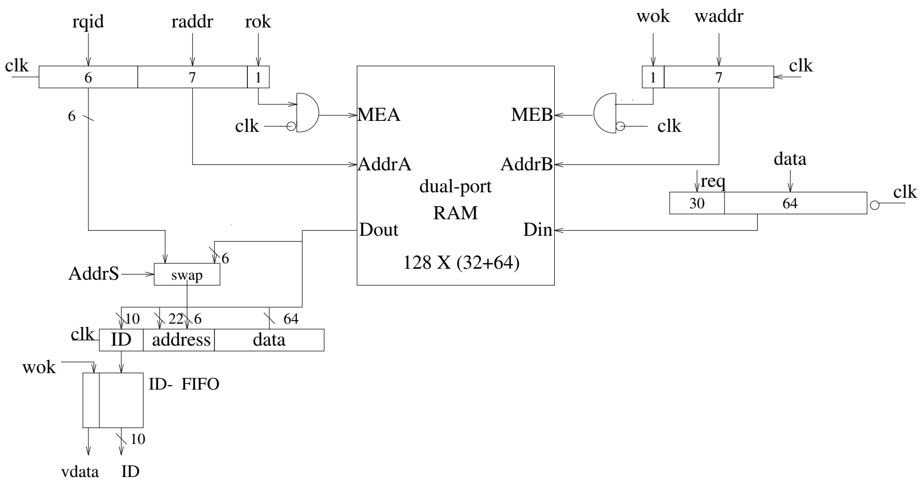


Figure 3.20: Memory block diagram

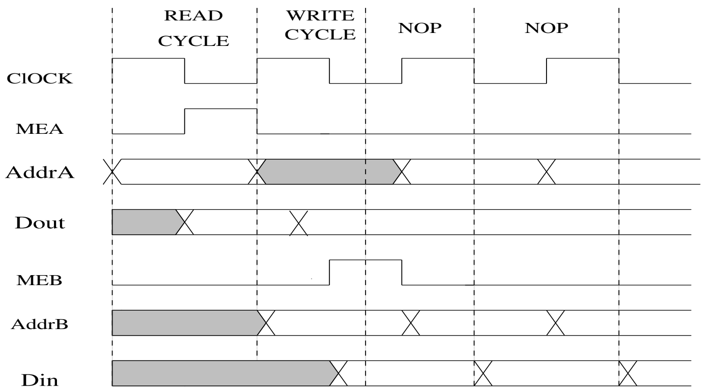


Figure 3.21: Memory timing signals

Read cycle: address is latched at negative clock edge while data is valid at the following (positive) edge. *Write cycle*: address is latched by the memory at negative clock edge and data at the following (positive) edge.

FIFO Memory logic.

Chapter 4

Design Methodology and Chip Metrics

The chip was designed using the Cadence Design Enviroment and the ES2 ecpd10 standard cell library. The chip layout was then produced using the Cadence tools for automatic placement and routing. In this chaper (§ 4.1) we describe the design flow: the schematic entry, the simulation, and the Place and Route process, and then (§ 4.2) we give some chip metrics.

4.1 Design Methodology

First, the design was described using Verilog [VeriXL]: we built behavioral models for the low level blocks such as adders, decoders, multiplexors, and used structural descriptions for the rest of the design. A behavioral model for the processor generating the requests was built, in order to supply the input requests, and test the processor interface protocol. The Verilog model was faster to develop and simulate, and helped in correcting some design errors in this early stage [SteSi]. As we shall see next, the debugging process is much slower in the Cadence Design System, because simulation is significantly slower, and changing the schematic diagrams is more time consuming than modifying the verilog code. Then, based on the Verilog model, we designed our chip using Cadence Design Enviroment and the ES2 1.0 micron standard cell library. The design flow imposed by the design software [SOLO2030] is shown in Figure 4.1.

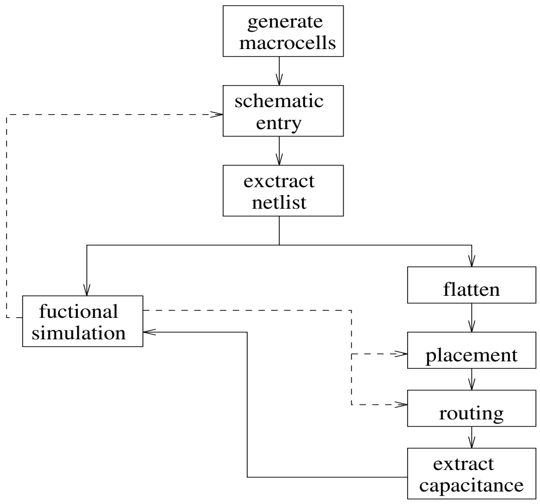


Figure 4.1:Design flow

Each block was designed using the Cadence schematic entry system, its netlist was extracted, and the design was simulated to verify the block's correct operation, before the block was included in the design hierarchy. The simulation results and the delays were observed using the available waveform displays. When the whole design was simulated using the verilog simulator, our initial behavioral verilog module was included for generating the requests according to the processor interface protocol. Then, the chip layout was produced using the Cadence Automatic Tools.

The netlist is flattened to produce the Autolayout representation which is the input to the Place and Route process [PandR]. The Autolayout representation contains the standard cells and the macrocells of the design and the logical connections between them. The output of the Place and Route process is the physical layout, which includes the cells placed in their final positions, and the physical geometries corresponding to the flattened netlist.

Before running the Placement program, the net priorities are set, and the placement regions are defined. The net priorities specify the order that the nets will be routed, and the routing algorithm. The default priority for all nets is 10. Nets with priorities greater than 100 are routed if possible in a single layer of metal. The clock nets were assigned a priority of 110, and the reset nets were assigned a priority of 101. The priority of power and ground nets was set to 101, and their width was set to 39μm. We also raised the priorities of some time−critical nets (still keeping them less than 100).

We grouped standard cells in clusters, and we assigned each cluster a placement region. We tried to cluster the standard cells in such a way that there is a lot of internal communication within a cluster and little communication with other clusters. In order to do this, we had to change slightly the interface of some blocks. For example, we replaced the *Row Logic* input, *wqid[0:63]*, with its encoded form, *wqid_enc[0:5]*. As shown in Figure 4.2, we defined five clusters corresponding to *InputLogic*, *Scheduling Logic*, *ROW Logic*, *Head register file*, and *Tail register file* blocks. Then, the placement planner was used, to estimate the number of standard cell rows that should be assigned to each region.

After setting the net priorities and the placement regions, the megacells and the I/O pads are placed, then the placement algorithm is run to place the standard cells automatically. The standard cells have equal height and will be placed in rows, while the area among the standard cell rows will be used for routing. The placement algorithm builds its own clusters of standard cells, based on their communication pattern, and tries to minimize the wire length. After the placement is over, the area left for routing is divided in vertical and horizontal channels.

During *Global Routing* the nets are assigned to channels. The global router determines the path each net takes through the existing channels based on a global analysis of the design: The routing algorithm evaluates the different possible paths for each net and seeks an optimal path based on net-priorities, wire length, channel congestion and other design constraints. First, nets with priorities greater than 100 are routed by a tree-based router, then the rest of the nets are routed by an expansion router.

During *Detailed Routing* the physical geometries are created according to the logical connections determined during *Global Routing*. Channels are routed independently, in the order that they were generated during channel generation. In this phase, each channel is automatically compacted or expanded, according to the area that is actually required for the routing of all the nets assigned to that channel.

After the end of the *Detailed Routing*, the placement regions are either compacted or expanded, this may result in empty areas in the chip core. We can adjust the placement regions and repeat the Place and Route process to improve the area utilization.

The pre-layout simulations use estimated loads: the load capacitance of each wire is estimated based on the total transistor count of the chip and the fan–in and fan–out the wire. Once the physical layout is produced, we can extract the paracitic capacitances in order to perfom simulations with the actual loads. In post layout simulations, some overloads were produced: the actual loads were in some cases significantly larger than the estimated ones. In the cases of significant overload, the schematic was modified: extra buffering was introduced to reduce the fanout of the overloaded wires. For the rest of the overloaded wires, their priorities were adjusted before repeating the Place and Route process.

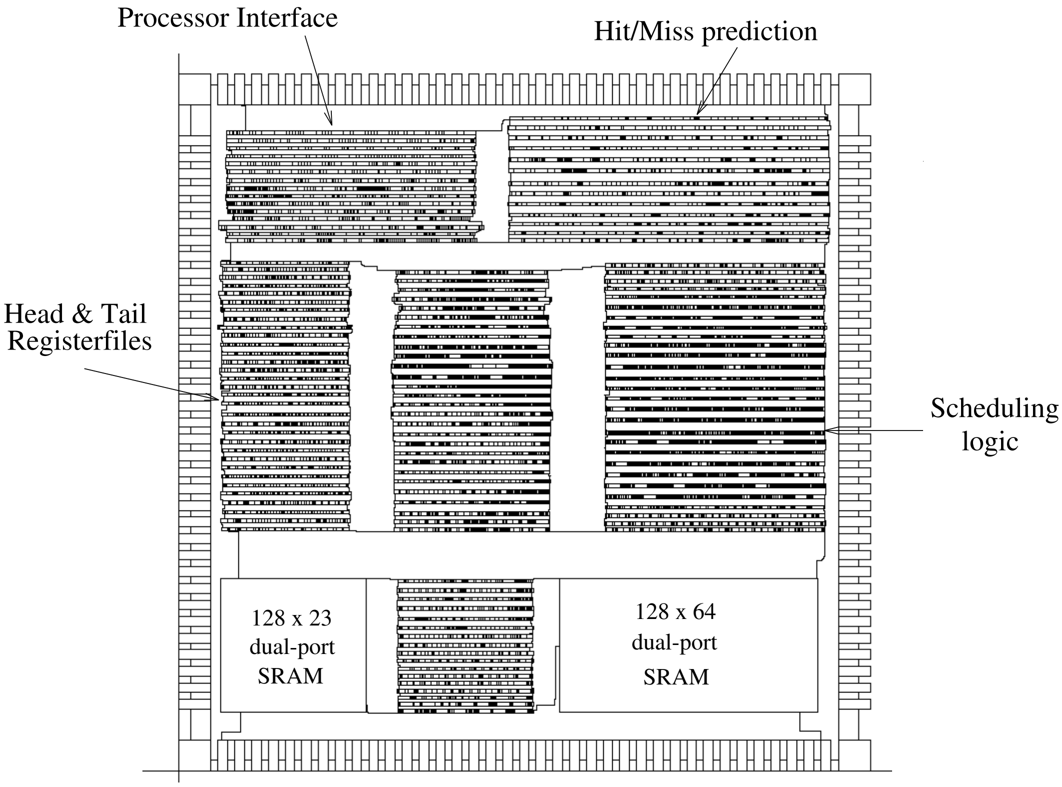


Figure 4.2:The chip layout

4.2 Chip Metrics

The chip layout is shown in Figure 4.2. The chip has 144 pins: 13 for power and ground, 16 for the Rambus Interface and 113 for the processor interface signals. There are separate power and ground pads for the core and the I/O pad ring, in order to decouple output switching. The core area is 8.65 x 8.87 mm^2, and the chip area is 9.5 x 9.7mm^2. The chip contains 11,460 standard cells (30% of which are inverters) and two dual–port RAM megacells (8Kbits + 4Kbits), totaling 68,700 gates or 272,800 transistors. We estimated that the total standard cell and macrocell area is 25.5mm^2 that is 33% of the core area.

The scheduling algorithm required our chip to schedule one request every 9 Rambus cycles, that is every 36ns. The current design was simulated successfully for a much slower clock, 53ns (18.8MHz). In our design there are two time–critical paths: The first critical path is in the first stage of the enqueue pipeline, while the second is in the first stage of the dequeue pipeline (see Figure 3.3).

The first critical path includes : A 64–to–1 mux which is part of the RW selection circuit (11ns). The request scheduling logic (see Figure 3.10). which includes a 64-bit priority encoder (11ns) two 2–to-1 muxes in series (5ns) and XOR gate synthesized by and XNOR gate and an inverter (3.5ns). The total delay added by the *Scheduling Logic* is 22ns. The *Scheduling Logic* circuit supplies the selected FIFO in decoded form. The next step is to access the Head and Tail register files (8ns), in order to estimate the SRAM address of the Head request of the selected FIFO (raddr[0:6]). The delay of this path is 46ns.

The path that turned out to be the critical one includes: Latching the input request and then extracting it from the (8-entry) Input Logic FIFO (10ns). Calculating the address of the FIFO in which the request will be kept (10ns), decoding of the FIFO address, and accessing the Head and Tail register files (8ns). Once the head and tail values are available, the SRAM address corresponding to the input request (waddr[0:6]) is estimated, and the waddr[0:6] address is then decoded, in order to update the RW/HM bits.

The first path is the one that we initially thought that it would the time critical path, thus it was carefully designed. In the design of the second path we gave more emphasis in space rather than speed, as a result it turned out slower than the first one. The delay of the second path can be reduced by bypassing the *Input Logic* FIFO when it is empty, and by modifying the organization of the Head/Tail register files, so that one of the two read ports accepts encoded addresses. After these changes, the first path will be the critical one.

Chapter 5

Conclusions

The traditional DRAM interface limits the DRAM-chip bandwidth to a few MBytes/s, and imposes the use of various expensive techniques in the design of memory systems, in order to provide the required bandwidth. The techniques used so far (wide data buses, traditional interleaving, separate second level caches) increase the number of chips and board space required, and the overall system cost. Novel DRAM architectures are required: cached DRAMs increase the chip bandwidth, but the interface is still the bottleneck. Rambus breaks the DRAM interface bottleneck, and it may be used in a wide range of applications where both high bandwidth and low cost is required.

This work was about the design of a memory controller for interleaved accesses over Rambus. Our controller maximizes the utilization of the bus bandwidth by exploiting the interleaving capability of the Rambus Channel. Our controller handles individual word accesses, and it could be used in a system with increased bandwidth requirements, such as a supercomputer that has to access non-cached vector elements with random stride.

In this work we assumed a single Rambus Channel, without extension cards, which may have up to 32 RDRAMs connected to it, which corresponds to up to 64 memory banks. Our controller keeps the processor requests in FIFOs, one per memory bank. If the system uses less than 64 memory banks, the existing FIFOs grow larger, so that all the available memory is used. Our controller includes a mechanism for predicting whether a request will result in a hit or a miss, and thus we avoid wasting bandwidth by reissuing the missed requests only once, when we can be sure that they will be served. The selection logic scans the FIFOs in a circular fashion in order to select the request to be scheduled in the next cycle.

More specifically our controller schedules a new request every 9 Rambus cycle pairs, which corresponds to a peak throughput of 222 MBytes/s. (64bits/36ns). The controller requires a clock that is 9 times slower than the Rambus clock (cycle time = 36ns). For the interleaved Rambus, the throughput for the (read or write) hits is one request per controller cycle (36ns), while the throughput for the (read or write) misses is one request every two controller cycles (72ns). The corresponding times for the non-interleaved Rambus is: 64ns for a read-hit, 36ns for a write-hit, 236ns for a read-miss and 214ns for a write-miss. Thus our controller doubles the throughput of read-hits and triples the throughput of misses. Assuming that the read to write ratio is 2 to 1 (reads 34%, writes 66%), then for high hit ratios e.g. 90% the average throughput of our system is 210MBytes/s, while for the non-interleaved Rambus it is 131MBytes/s. As the hit-ratios are decreasing our system gets even better, compared to the non-interleaved Rambus: For a 50% hit ratio the average throughput for our system is 166MBytes/s, versus 47MBytes/s for non-interleaved Rambus.

Our chip was designed using the ES2 1.0μm standard cell technology. The chip area is 9.5 x 9.7mm^2 and it was simulated successfully for a 18.8MHz (53ns) clock.

Possible modifications and extensions

If there is a first level cache between the memory system and the processor then memory reads involve cache lines, which makes the interleaving of read requests impossible, because of the size of the read data packets. Interleaving is only possible in case of RDRAM misses. Our controller could be modified to handle both word and block transfers, interleaving requests whenever possible. In many systems, however, such as in PCs and workstations, the bandwidth supplied by a non-interleaved Rambus Channel is sufficient, so interleaving is not necessary.

Our controller maximizes the utilization of the Rambus bandwidth, at the cost of increasing the latency of the read-hit transactions. The minimum latency for a read-hit is 6 controller cycles (216ns). A fraction of this latency is due to the fact that each request has to go though the controller's FIFOs before it presented to the Rambus Interface. In case that there are no pending requests in controller (all of its FIFOs are empty) the FIFOs could be bypassed, and the input request could be fed directly to the Rambus Interface. This modification reduces the minimum latency from 6 to 4 cycles (144ns), when this bypassing is possible (see also 3.2.2).

The number of FIFOs required, the FIFO sizes, and their effect in the system's performance should be studied more carefully, and perhaps be adjusted to the needs of the specific application, since the number and the size of the FIFOs determine the chip size and the chip cycle time. In our design we assumed 64 FIFOs with two elements each. If the number of the FIFOs was 32, the chip size would be reduced by half, at the cost of assigning two memory banks per FIFO, for systems with more than 32 memory banks.

For the processor interface we assumed that the processor sends the write request and the corresponding write data in 3 consecutive cycles. This assumption was wrong, because it makes the processor interface the bottleneck that limits the system's throughput. This can be corrected (as shown in 3.2.1) by modifying the processor interface, so that the rate that processor sends the requests, is equal to the peak rate that the requests can be served by the interleaved Rambus (=1 request/cycle). In this new processor interface, the number of the input pins will be increased by 28, and the processor will send the data at both clock edges.

In this implementation we assumed a simple synchronous processor interface and we also assume that the controller's clock is provided by the processor. This means that the processor clock has to be a multiple of the Rambus clock. Normally the processor clock can be asynchronous, and with any frequency relationship to the Rambus clock, in which case some synchronization logic is required.

Our controller schedules a new request every 9 Rambus cycle-pairs (36ns). This resulted from the assumption that an idle Rambus cycle-pair (4ns) is required for bus turnaround, every time the direction of the bus transfers changes. If this applies only for master requests followed by slave responses, then we may schedule a new request every 8 Rambus cycle pairs (32ns) (provided that the *ReadPipeDelay* and *WritePipeDelay* values can both be zero). This change will not affect the scheduling algorithm: only the controller clock and the MISS_DELAY value would have to be adjusted.

References

[HePa] John L. Hennesy and David A. Patterson, "Computer Architecture A Quantitative Approach", Morgan Kaufmann Publishers 1992

[Ramtron] DM2202 EDRAM 1Mbx4 Enchanced Dynamic RAM (Preliminary Datasheet) RAMTRON

[Ramtron2] D.Bursky, "Combination DRAM-SRAM removes seconadry caches", Electronic Design, Jan 23 1992

[Mitsu] MSM44409TP-10,12,15,20 4Mbit Cached DRAM (Preliminary Spec) Mistubishi ELECTRIC

[Mitsu2] D.Bursky, "Integrated Cached DRAM lets data flow at 100 MHz" D.Bursky, Electronic Design, Feb 20, 1992

[RaOv] "Architectural Overview", Rambus Inc 1992

[RaGu] "Rambus Technology Guide", Revision 0.9 - Preliminary, Rambus Inc, May 1992

[RaDRAM] N.Kushiyama, S.Ohshima, D.Stark, H.Noji, K. Sakurai, S. Takase, T.Furuyama, R.M.Barth, A.Chan, J.Dillon,J.A.Gasparro, M.M.Griffin, M.Horowitz, T.H.Lee, V.Lee "A 500 Mbyte/s Data-Rate DRAM", IEEE Journal of Solid-State Circuits, vol. 28, no. 4, April 1993, pp. 490-496

[ES2Lib] "ECPD10 Library Databook", European Silicon Structures, April 1993

[ES2Gui] ECPD10 Asic Design Guidelines, European Silicon Structures

[SOLO2030] "SOLO 2030 User Manual", European Silicon Structures

[VeriXL] "Verilog-XL Reference Manual" Gateway Design Automation Corporation,1989

[SteSi] E.Sternheim, R.Singh, and Y.Trivedi, "Digital Design with Verilog HDL", Automata Publishing Company

[PandR] "Automatic Place and Route Reference Manual", Cadence Design Systems 1989