# The Design of the Node for the Single Chip Message Passing (SCMP) Parallel Computer

by

Mark Bucciero

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in a partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Engineering

March 23, 2004

Blacksburg, Virginia

Keywords: Processor, parallel, node, system on chip, single chip computer, SCMP

# The Design of the Node for the Single Chip Message Passing (SCMP) Parallel Computer

Mark Bucciero

James M. Baker, PhD, committee chair

Bradley Department of Electrical and Computer Engineering

## Abstract

Current processor designs use additional transistors to add functionality that improves performance. These features tend to exploit instruction level parallelism. However, a point of diminishing returns has been reached in this effort. Instead, these additional transistors could be used to take advantage of thread level parallelism (TLP). This type of parallelism focuses on hundreds of instructions, rather than single instructions, executing in parallel. Additionally, as transistor sizes shrink, the wires on a chip become thinner. Fabricating a thinner wire means increasing the resistance and thus, the latency of that wire. In fact, in the near future, a signal may not reach a portion of the chip in a single clock cycle. So, in future designs, it will be important to limit the length of the wires on a chip.

The SCMP parallel computer is a new architecture that is made up of small processing elements, called nodes, which are connected in a 2-D mesh with nearest neighbor connections. Nodes communicate with one another, via message passing, through a network, which uses dimension order worm-hole routing. To support TLP, each node is capable of supporting multiple threads, which execute in a non-preemptive round robin manner. The wire lengths of this system are limited since a node is only connected to its nearest neighbors.

This paper focuses on the System C hardware design of the node that gets replicated across the chip. The result is a node implementation that can be used to create a hardware model of the SCMP parallel computer.

## Acknowledgements

An extra special thanks goes to Dr. James M. Baker and his family. Dr. Baker, you always gave me the freedom to work on what I wanted, when I wanted. This approach taught me to become self motivated in my work. It may have taken me a little longer to realize what was happening, but I do appreciate your style and patience. You always listened to my research opinions, good or bad. We talked about them, and you always made sure I understood what I had not considered. Thanks for always inviting me and whomever I wanted to bring to tailgate with you before football games. The food and company was always outstanding.

To all those who have worked on or are currently are working on the SCMP project, thank you for keeping me in the loop and making sure I was thinking about all aspects of the problem, rather than just my own.

Thanks to Brian Gold for not being shy about telling me what I did wrong. I am a much humbler person having worked with you. I always enjoyed our days working and our nights out.

To Alicia Petta for always being supportive of my work/life and empathetic about my complaining even when I did not work enough to complain about it. You always make me think about things other than work, and make sure I keep the big picture in mind.

I would like to thank Alison Lazarevich for her listening ear and constant encouragement. You inspire me to be more than I think I can be.

Special thanks go to Michael Carr for keeping me sane and always making the time for video games. I feel better knowing there is someone out there with similar problems to mine.

Thanks to Marie Dotson for her constant friendship and loving me just the way I am. My college years would not have been nearly as good without you.

To my mother, for making me feel special no matter what she had going on in her life. To my father, for his constant wisdom and for reading the technical babble that follows. Thanks to Paul and Sarah for always making sure I sharpen the saw by attending their holiday parties.

# Table of Contents

## Table of Figures

## Table of Tables

# Chapter 1 Introduction

Will processor speeds continue to increase with Moore's Law? Will a paradigm shift in processor design be necessary to make full use of shrinking technologies and increasing clock speeds? This thesis presents a portion of a new processor architecture that will hopefully allow processor speeds to continue to increase with Moore's Law.

This thesis begins by discussing the limitations of current processor technology and introduces a new architecture that can work within these limitations, while extending the bounds of processor speeds that can be achieved. The significance of this research and a brief outline of the remainder of this document are also given.

## *1.1 Motivation*

Today's processor designs use reduced transistor sizes to improve processor performance in two ways. First, as the transistor has gotten smaller, its switching speed has increased. Faster switching speeds allow for the same processor design to execute the same program in less time. Second, when a smaller transistor technology is introduced, more transistors can be placed on a die of the same size. In fact, the International Technology Roadmap for Semiconductors (ITRS) estimates that a chip will contain one billion transistors by the year 2007 [1]. These additional transistors are used to add functionality to a design to make a program execute in fewer clock cycles. However, these design techniques cannot continue to improve processor performance at the rate of Moore's law.

With these smaller transistors, the cross sectional area of the wires on a silicon die will be reduced. This thinner wire leads to increased resistance per unit area. One study projects that when transistor technology reaches 70 nm, less than 20% of a chip could be reached in a single clock cycle [2]. Eventually, processor speeds will be determined by the lengths of the wires on the chip and not by the switching speed of a transistor [3].

One effect of making smaller transistors is that a larger number of them can be placed on the same die. Current processor architectures use these extra transistors to add new features to a design. However, the extra logic needed to support the new features tends to use the additional transistors, so the wire lengths still span the entire chip. Thus, a more complex processor design will have wire latency issues in the near future [3][4][5].

The additional transistors on a chip tend to be used to exploit instruction level parallelism (ILP). It makes sense because if more instructions can be executed at once, then the program will execute faster. The additional hardware necessary to increase performance by exploiting ILP creates additional design and test time. The design of a new algorithm to make use of ILP must first be tested on a number of applications to determine its worth. Then, a hardware implementation of that algorithm needs to be developed and tested. Finally, when the new design is fabricated, it must also be tested. The process of creating and testing a new design has shown little improvement in ILP performance for a general application. Thus, the concern is that exploiting ILP has reached a point of diminishing returns [5].

This concern has lead to an increased interest in thread level parallelism (TLP). TLP takes small groups of instructions and executes them in parallel. Therefore, its concept is coarser than ILP, which exploits the finest grain of parallelism, a single instruction. Diefendorff believes that coarser grained parallelism will become the dominant force in microprocessor design [6].

Processor designs of the future will also need to consider the 'Memory Wall' effect [7]. The basic idea is that memory speeds will not continue to increase with processor speeds. Thus, the processor will have to delay for long periods of time during an off chip memory access. One way to counter this effect would be to put all the memory needed by the processor on the chip itself. This configuration would allow the memory speed to be the same as level 1 cache, which is orders of magnitude faster than an off chip memory access.

## 1.2 Overview of SCMP

One possible architecture alternative is the single chip message passing (SCMP) parallel computer. The SCMP architecture supports thread level parallelism and attempts to minimize the length of the wires on the chip [8]. This architecture uses the transistors on the chip to create a number of simple processing elements, called nodes. The individual nodes are small and global signals are avoided to limit the lengths of the wires on the chip. Thus, the clock speed of the SCMP processor can be determined by transistor switching speed rather than the wire latency.

The nodes on the chip exploit TLP and, to keep them simple, provide no support for ILP. Each node can maintain multiple threads, with each thread being on the order of tens to hundreds of instructions. The threads on a node execute in round robin non-preemptive fashion. These threads are used to keep a node busy so context switches occur while one thread is waiting for data. Some multimedia applications, such as videoconferencing, contain enough TLP to use up to 64 nodes for processing [6]. Current parallel architectures cannot efficiently support thread level granularity because of the cost of communication. Also, the uniprocessors used in other parallel architectures were designed to make use of ILP instead of TLP.

Nodes on the chip each have their own local memory to counter the 'Memory Wall' effect [7]. Memory is distributed on the chip to keep the wire lengths as short as possible. When a program is loaded, the same image of that program is copied into the memory on each node. This fact simplified programming for SCMP since one node could write data values to known locations on another node.

The SCMP parallel computer is a message passing system similar to that of the Pica and the J-Machine [9][10]. The nodes on the chip are connected in a 2-D mesh, so each node is connected to its four nearest neighbors. Messages are sent between nodes using dimension order worm-hole routing. Since each node is only connected to its nearest neighbors, the wire lengths on the chip remain short. Thus, communication between nodes can happen in less than one clock cycle. In addition, there is only one physical channel between nodes. Therefore, messages that attempt to use the same physical channel can be multiplexed over the single link using virtual channels [11]. For more information on the network, see "Balancing Performance, Area, and Power in an On-Chip Network" by Brian Gold [12].

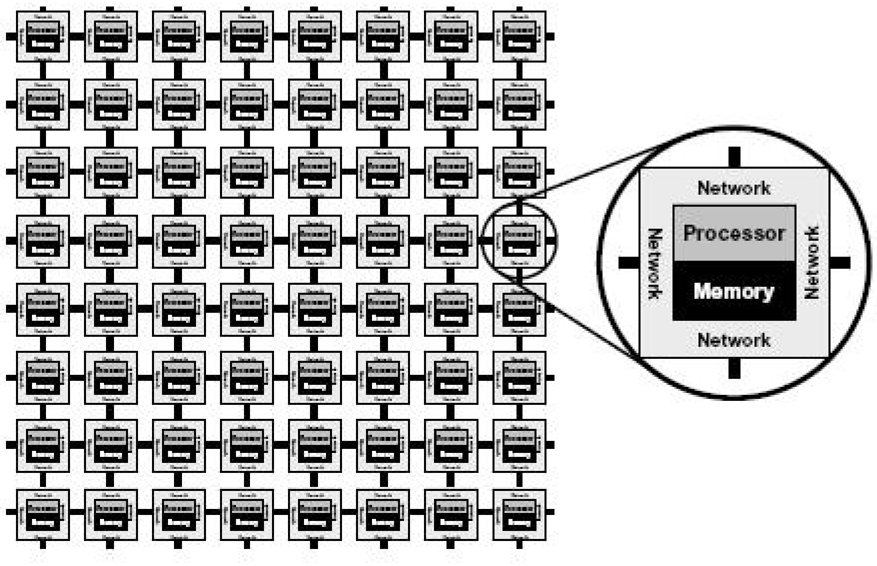


**Figure 1.1**: The SCMP parallel computer connects nodes in a 2-D mesh with nearest neighbor connections.

## 1.3 Significance of Thesis

The major parts of the SCMP parallel computer are the node that is replicated across the chip and the network that connects the nodes. The crux of this thesis effort was the design and implementation of a single node for the SCMP architecture. Some aspects of the SCMP node were determined before this thesis effort began. These features will be explained later in this document and include: the instruction set, the message format, and the Context Management Table (CMT) format. The goal of this research was to design the node around these requirements and still have the node function properly.

Without nodes, the SCMP chip becomes a network back plane with no computational abilities. The design includes everything from formatting messages for the network, to thread management, to memory access and control. The goal was to create a fully functional 32 bit RISC processor that would fully support all of the needs of the SCMP instruction set.

Currently, the SCMP software simulator assumes a certain number of clock cycles per instruction. Once this work is completed, the simulator can be modified to mimic the actual hardware design. Thus, a clock cycle accurate software simulator would be created. This new simulator could then be used to test a set of benchmarks on the SCMP parallel processor. These benchmarks were chosen so that the performance of the SCMP processor could be compared to others like it.

## *1.4 Thesis Organization*

The requirements for the SCMP node must be defined before its design and implementation can be discussed. Chapter 2 of this thesis gives a detailed description of all of the parts of a node. Chapter 3 presents the implementation of the majority of the SCMP node. Chapter 4 discusses the implementation of the context and context management portion of the node. A full chapter was dedicated to this topic because of its complexity.

After the node was initially designed, some changes in the requirements for the node were identified. The design was changed to include floating point support and exception handling. These changes and their implementation are discussed in Chapter 5.

Once the hardware was designed and implemented, it was necessary to test its functionality. Chapter 6 describes the software used and the hardware simulation support created for a SCMP node. This software includes a test bench that creates a node and a module to load a program into memory and start the simulation. After it completed, the simulation results were verified. Finally, a summation and some possibilities for future work are given in Chapter 7.

# Chapter 2  The SCMP Node

The node that is replicated across the SCMP parallel computer needs to be simple enough so up to 64 nodes can be included on a single chip. Also, the wire lengths need to be short enough so that a node can perform all of its operations in one clock cycle. However, it is complex enough to support thread level parallelism, message passing, and memory management [8]. Since each node also has its own local memory, support is necessary to share data between nodes. To support these extra features, the SCMP node has a unique instruction set (see Chapter 2.3). This chapter describes what is required of an SCMP node and how it was subdivided for hardware development.

## 2.1 Description

Each node of the SCMP parallel computer acts as a generic processor, so it must be designed to handle any general task. To satisfy this requirement, the node supports all types of operations. These operations include but are not limited to: arithmetic, logic, memory, thread access, thread management, and networking. In addition, a central control unit, in this case a pipeline, is needed to manage all of these operations.

This processor architecture is proposed as a design alternative to combat the problems of wire latencies and unmanageable complexity. In order to minimize the wire latency, global signals are not allowed in the SCMP design. Therefore, the on chip memory must be distributed to each node rather than being shared between nodes, which would require wires to be as long as the chip. Figure 2.1 depicts the shared memory design approach. For example, the IBM Power4 takes this approach to placing two 64 bit nodes on a single chip [13]. Similarly, the MAJC [14], Hydra [15], Blue Gene [16], and CMP [17] architectures also use this configuration.

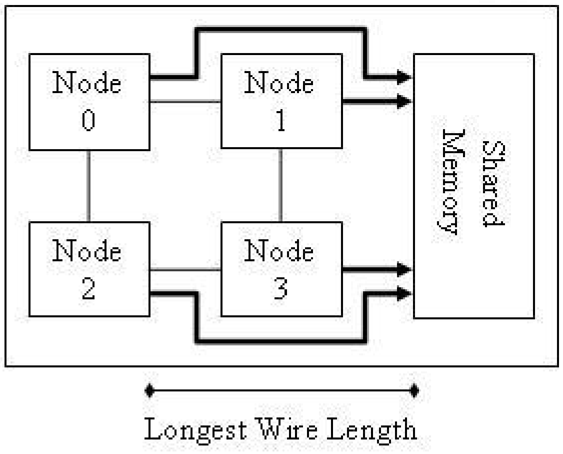
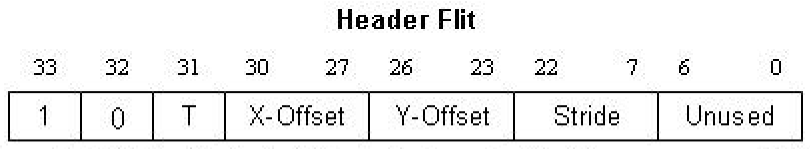


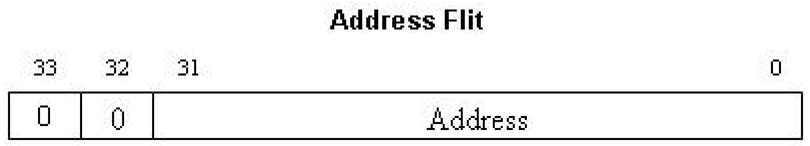
**Figure 2.1**: The bold wires show the additional wires needed and the longest wire length added in a shared memory system.

The SCMP parallel computer is a MIMD system because each node can operate independently on its own data stream. Therefore, it is important to have an easy way to share data among the processors on the chip. The SCMP network uses an active message passing scheme. When a message reaches its destination node, it enters through the network interface unit (NIU). The NIU, then, determines the type of message from the information contained in the message header. The incoming message is either a thread message, which creates a new thread to execute, or a data message, which writes directly to memory.

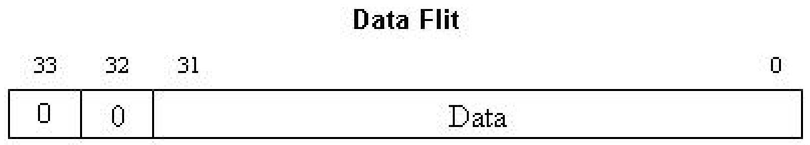
Messages in the SCMP network are sent in small pieces, called flits (flow control digits). A message consists of a header flit, an address flit, numerous data flits, and a tail flit. The header flit, shown in Figure 2.2, contains which type of message (thread or data) is being sent; the destination node (given in x and y coordinates); and the bit that signals it is a header flit set to one. In the case of a thread message, the address flit contains the instruction pointer at which a thread is to begin execution. On the other hand, the address flit of a data message contains the memory address at which data will be written. The data for the data flits is generated directly from the registers or from the memory. This method limits the amount of data that is copied when creating a message, which should improve message throughput. There can be any number of data flits following the address flit. The tail flit is the last data flit of a message and is signified by the tail bit being set to one.



(a) Bit 33 is set to '1' to indicate that this is the header flit of the message. Bit 31 is used as the message type ('1' for thread and '0' for data).



(b) Bits 33 and 32 are always '0'. The address is either the IP address to start a thread or a memory location to begin copying.



(c) Data is sent in these flits. Bit 32 is used to indicate the tail flit. When that bit is set, the message is complete.

**Figure 2.2**: Images (a), (b), and (c) show the different parts of a message in the SCMP network.

The routing of messages between nodes is done through an on chip router that uses dimension order worm-hole routing. This type of routing means that the header flit makes a path through the network for the rest of the message to follow. In addition, messages are always routed first in the x direction, then in the y direction. Routing messages in this pattern ensures that dead lock will not occur in the network.

When a thread message arrives at the NIU of a node, a space to store the thread, called a context, is requested. Since the SCMP node is designed to handle multiple threads, there are multiple contexts (currently 16). Each context contains 32 32-bit general purpose registers that are used to store data. The context management table (CMT) keeps track of those contexts that are in use and those that are free. Each CMT entry has four main fields: the Alloc bit – set to one if a context is in use; the Thread bit – set to one if this context was to be scheduled for execution; the Data Context Register (DCR) – an identifier that points to the context that can be used as additional data storage; and the instruction pointer – the memory address for the thread to begin its execution. It is important to note that the IP value in a CMT field is only valid when a thread begins execution or restarts after a suspension. There is an IP register within the node pipeline that keeps track of the actual instruction that is executing. Figure 2.3 depicts a CMT entry with the named fields.

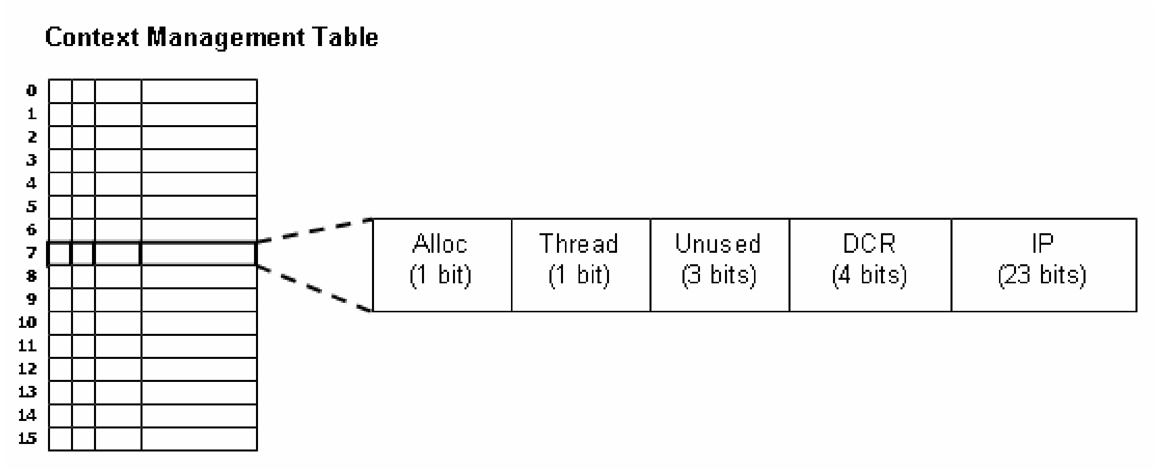


**Figure 2.3**: The CMT contains the information needed to determine if a context is ready to execute and where that execution should begin.

Once a CMT entry is assigned to the incoming message, the address flit value is written to the IP value of the CMT entry. The data flits following the address flit are written directly in to the register block for that context. Therefore, only 32 data values can be sent in a thread message. If more flits are sent, the data values did not have a place to be stored. As a result, if a thread message contains more than 32 data flits, the additional flits are ignored.

A context is scheduled for execution when both its Alloc and Thread bits are set in the CMT entry for that context. Threads are scheduled in round robin non-preemptive fashion. This type of scheduling ensures that no thread will be starved as long as one of them does not enter an infinite loop. The Active Thread Register (ATR) acts as a pointer in to the CMT for the currently executing context. When a thread ends or suspends, a search occurs for the next thread that is ready to execute. If the thread suspends, the ATR is used to identify which CMT entry should be modified to save the suspended thread information. The IP field of the CMT entry is updated with the address of the instruction that will begin its execution the next time the thread is scheduled.

All of the context switches occur in hardware. This design allows for minimal delay from the end/suspend operation on one context to the start of the next.

Once a thread begins execution, it runs until it either completes or suspends. A thread can be suspended specifically with a suspend instruction or due to failure of an instruction in the pipeline. For example, the send instructions are used to inject messages into the network. If the network buffers are full, then the send instruction fails, and the thread suspends.

When a data message arrives at a node, the address flit contains the first address in memory to which data is written. The appropriate values are extracted from the data flits and written sequentially into memory. This message type is used frequently when large blocks of data need to be transferred from one node to another. Unlike the thread message, the data message type does not have a limitation on the number of data flits that can be sent.

In addition, no memory space on the SCMP node is protected. A careless programmer could easily write into the code space of a program. In this situation, the outcome of the program would be unknown. Therefore, it is very important that space be allocated for the incoming message before it arrives. Since all addresses used are physical addresses, space cannot be dynamically allocated for an incoming message.

Now that the SCMP parallel computer has been described, let us consider other similar projects currently being researched. As previously noted, the SCMP node does not support instruction level parallelism within a thread. The SCMP node is different from the RAW architecture [18], which does support ILP in addition to coarser levels of parallelism. The RAW processor also uses a message passing model between nodes and distributes memory among the nodes. However, only a small amount of memory is included on the chip, so many memory accesses will need to access off chip memory. Therefore, the RAW architecture may still suffer the Memory Wall effect. Also, unlike SCMP, the compiler for the RAW machine is allowed to access some of the hardware features on the chip. This permission allows the compiler to statically schedule certain events. For example, the compiler can insert an instruction to set a switch to allow a certain message to pass through that router.

The focus of the IRAM project [19] is to integrate DRAM on the chip with a single processing element. It is different from SCMP since SCMP uses many processing elements. However, the IRAM venture is nearly identical to a single SCMP node. The difference is in the network capabilities provided to the SCMP node.

An SMT system [20] uses a different paradigm than the systems described above, but it does exploit thread level parallelism. The chip of an SMT processor contains many functional units. Each thread in the system is given the capability to use as many of these units as possible. In other words, in a given clock cycle, instructions from different threads are executing at the same time in different functional units. This design, however, does not account for either the Memory Wall effect or the wire latency phenomenon expected in smaller transistor technologies.

## 2.2 Hardware Partitioning

This design of an SCMP node lends itself to a hardware partitioning. The partitioning of such a large design effort made it more manageable, since the different parts could be developed, modified, and tested individually. The division was done with the objective of decreasing design time, while increasing design accuracy. A single node of the SCMP parallel computer was divided into the following blocks: ALU, contexts and context management, instruction cache, memory management, network interface unit (NIU), and pipeline. Table 2.1 lists these blocks with a short description of the responsibilities for each unit.

**Table 2.1**: The hardware division of an SCMP node.

| Block Name | Description |
|---|---|
| Arithmetic Logic Unit (ALU) | Responsible for calculations, comparisons, and shifting |
| Contexts and Context Management | Manages the registers and threads running on a given processor |
| Instruction Cache | Stores instruction words for faster execution times |
| Memory Management | Controls accesses to the node's local memory |
| Network Interface Unit (NIU) | Injects and ejects messages to and from the network |
| Pipeline | Acts as the node controller |

One major concern about partitioning the hardware design was creating and maintaining the interfaces between the blocks. This concern arose because if one block had its interface changed, the affected block must also change its interface. Therefore, it was of paramount importance to create interfaces that covered the scope of the required operations but allowed the flexibility to add or change the functionality. Creating the interfaces in this way eliminated the above concern because the interfaces were not required to change or accommodate change in the functionality of a block in a node. Figure 2.4 depicts an SCMP node with its interconnections. The connections show which blocks interact with which other blocks but not the actual interface that was developed.

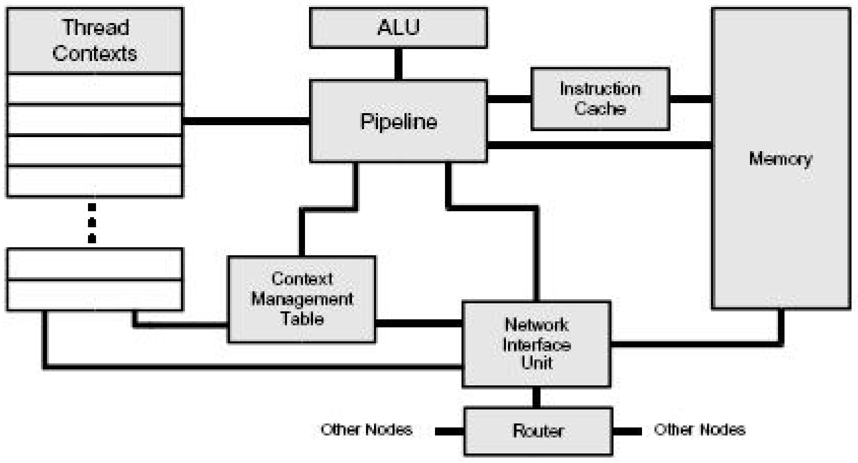


**Figure 2.4**: A block diagram of the SCMP node.

The pipeline is modeled after the MIPS 32 processor core and consists of five stages: fetch, decode, execute, memory access, and write back. Documentation for this processor family can be found at MIPS Technologies, Inc. [23]. The MIPS pipeline functionality was modified to accept the SCMP instruction set and to control the various blocks of the node.

The interfaces and operations that each block performs are considered the hardware implementation of the node. These descriptions are given in Chapter 3, for the ALU, instruction cache, memory management, NIU, and pipeline. In Chapter 4, the details of the contexts and context management are given. The contexts and CMT are described in a separate chapter because of their complexity. The testing and verification of each block and the complete node are presented in Chapter 6. Please refer to the appropriate chapter for further information

## *2.3 Instruction Set*

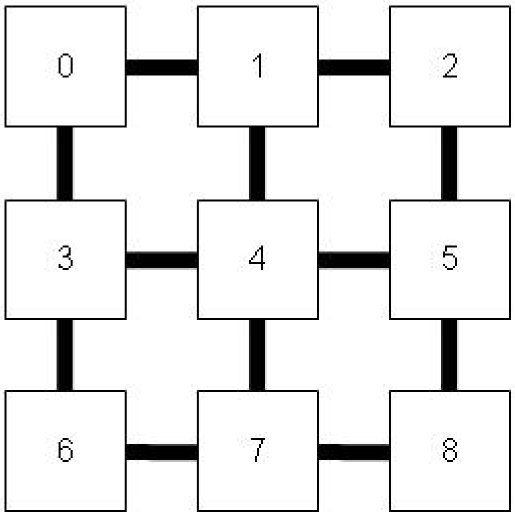
With the extra hardware support required for the SCMP node, special instructions were needed in the instruction set to make use of these features. The special instructions are discussed here. The full list of instructions is given in Appendix A.

In addition to the NIU, the node pipeline can also manipulate the CMT. The pipeline can perform a memory write to the CMT or use the alloc instruction. A memory write could succeed in allocating a context, but there is a potential conflict if the NIU requests a new context in the same clock cycle. In this case, both the pipeline and the NIU viewed the context as free, and thus both allocate it. Therefore, an instruction, alloc, was required to atomically allocate a context to the pipeline. Using the alloc instruction, if both the pipeline and the NIU request a context in the same cycle, the pipeline would be given priority and succeed, while the NIU request would fail. The alloc instruction is used to set the Alloc bit of a free context to one. The return value of this instruction is a pointer to the CMT entry that has been allocated. This command effectively removes a context from the list of available contexts for use by the NIU or the pipeline. After the alloc instruction succeeds, the user can write to the Thread bit, the DCR, and the IP fields in the CMT entry. The free instruction is used to clear the Alloc bit of the designated CMT entry. However, a thread cannot use the free instruction on its own CMT entry.

If a thread contains an intensive amount of processing, it is able to give up control of the processor and let the other active threads have execution time. The suspend instruction is used to accomplish this operation. Since the contexts are not preemptive, operations are guaranteed to be atomic. Therefore, the suspend instruction also allows for synchronization to take place. For example, let us say a thread is waiting for a memory location to change from zero to one. The value in the designated memory location is accessed and its value checked. If the value is zero, then the thread would use the suspend instruction to allow other threads to execute. The likely case is that one of the other threads will change the value at that memory location to one. Then, when the original thread checked the memory value again, if it changed to one, the thread would continue its execution. If it had not changed, the thread would repeat the suspend instruction until the value changed.

The last instruction of any thread is the end instruction. This instruction signals the CMT to clear the Alloc bit of the entry pointed to by the ATR. The freed CMT entry is then added back to the list of contexts available for use by the NIU or the pipeline. Then, the next context to execute can begin its execution. As previously stated, contexts are scheduled in round robin fashion, and both the Alloc and Thread bits must be set for a context to be scheduled.

The other special SCMP instructions involve sending a message. As previously described, a message begins with the header and address flits. Both of these flits can be generated using the sendh instruction, which has two variations. The first type takes the form: sendh <register>, type, address. The register operand contains the destination node number. Nodes on the SCMP chip are numbered from left to right and top to bottom. Figure 2.5 shows such a numbering scheme. Since there are two types of messages, the type field names the message type (thread or data). The third operand is an address. If it is a thread message, then the context begins execution at this address. The address is the location to start writing to memory, if it is a data message.



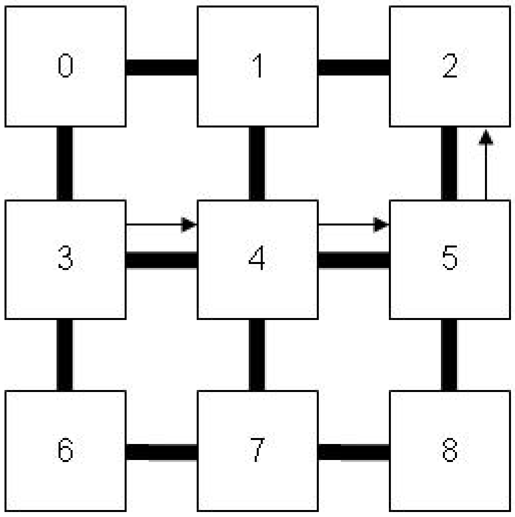


**Figure 2.5**: The left figure shows how the nodes are numbered on the SCMP parallel computer. The right figure depicts how a message would be routed from node 3 to node 2.

The second form of the sendh instruction is as follows: sendh <register1>, type, <register2>[, stride]. The first three values are the same as before, but the address is contained in the second register reference rather than in an immediate value. The stride value is optional and signifies how far apart values should be stored in memory for a data message. The first value will be stored at the address given in register2, and the second value will be stored at <register2>+4*stride. For example, if a data message writes to a column in a matrix, the addresses written to would be spaced evenly, but by more than one address, in memory.

After the header and address flits are sent, data follows. There are three different ways to send data. The first two are very similar and take the forms: send <register> and send2 <register1>, <register2>. Both of these instructions send data directly from a register(s) into the network. This procedure limits the amount of data copying that occurs, with the objective being that messages will reach their destination faster. The worm-hole routing allows the header flit to pave the way for the other flits of the message, which do not have specific knowledge of the destination node. The third way to send data is the sendm instruction. It takes the form, sendm <register1>, <register2>. This instruction copies bytes from memory and injects them directly into the network. As before, this tactic minimizes the copying that occurs before a message is sent. The first register contains the address of the first four bytes to copy from memory. The second register contains the number of four byte values to copy and the stride between the values.

To end a message, a tail flit needs to be sent. Similar to the above, there are three different ways to end a message: sende, send2e, and sendme. A bit is contained in each flit to indicate if this one is the tail flit. The data is sent in the same fashion as described above for the send instructions, but the tail bit is set in the flit with the last data value to indicate it is the end of the message.

The discussion of the instruction set for the SCMP node completes the description of its operational design and functional requirements. The major components of the node were introduced in this chapter. The design objectives of the SCMP node and the hardware partitioning used to optimize its design were presented. The required messaging and message formats used by the SCMP components were described, along with the instruction set developed to provide the expected functionality. The next 2 chapters will discuss the hardware implementation of the SCMP node that was developed for this thesis.

# Chapter 3  Hardware Implementation

After the functionality of each block of the node had been designed, the interfaces between the blocks were created.  A specific naming convention was developed so that each signal between blocks could be identified quickly and easily.  If a signal was an input to a block, it began with a lower case ‘i’.  Similarly, a block output signal started with a lower case ‘o’.  The name of the signal took the following form:
<source block name><description>To<destination block name>.

The description of each signal was intended to be straightforward enough so that the purpose of the signal would be understood.  For example, if an enable signal was generated from the pipeline to the memory, then its signal name would be PipelineEnableToMemory.  The signal name at the pipeline would start with an ‘o’, and at the memory, it would start with an ‘i’.

Once the interfaces were defined, each block was implemented in System C.  System C is a hardware modeling language, like VHDL or Verilog, but with C++ syntax.  System C is a good example of a system description language that can be used to model Systems-On-a-Chip [21].  This chapter and the one following describe in detail the interface and implementation of each of the SCMP blocks.  Research for this thesis focused on the design and implementation of the SCMP memory controller, the Network Interface Unit, and the ALU.  SystemC models of the on-chip memory and instruction cache were also developed.

## *3.1 Memory*

The memory block of the SCMP node was created for simulation purposes only.  It is expected that a synthesized SCMP chip will use a third party vendor static RAM or embedded DRAM chip, if the latter technology continues to improve.  However, there are restrictions that must be considered when choosing a third party chip.  The memory must be synchronous, byte addressable, and allow accesses of 1, 2, and 4 bytes on even boundaries.

Since the SCMP parallel computer attempts to store everything it needs for a program on the chip, the memory access time becomes paramount.  Therefore, it is assumed that the local on chip memory for each node is fast enough to be accessible in one clock cycle.  However, there is not a guaranteed response time for gaining access to the memory space of another node.  The response time depends on things like network traffic, if the destination node is already receiving a message, and accessibility to the actual memory.  Three units in a node can access the memory: the pipeline, the instruction cache, and the network interface unit.  See Chapter 3.2 to learn how each unit can gain access to memory.

The implementation of this block is straightforward.  The memory is single ported, so no collision avoidance is required.  Each memory address is 32 bits wide with the bytes stored in big endian format.  The inputs and outputs of this block are depicted in Figure 3.1.  Access to the memory is accomplished through a request/acknowledge sequence.

That is, the unit that wishes to access memory makes a request and then waits one clock cycle for the acknowledge signal. If the acknowledge is set, the data is valid, but if not, the access is denied.

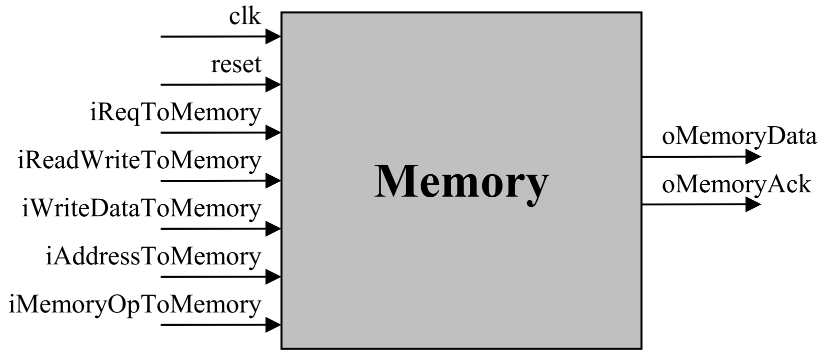


**Figure 3.1**: A block diagram of the memory on the SCMP processor.

The reset of this block is asynchronous and sets the oMemoryAck line to '0'. Any request made to memory during a reset is ignored. A request is made on a rising clock edge, if the iReqToMemory signal is set. Then, a read or a write operation is performed, based on the input iReadWriteToMemory, on the number of bytes given by iMemoryOpToMemory. For the appropriate setting of the signals to this block, see Table 3.1.

**Table 3.1**: The input and output ports and descriptions of the memory block's signals.

| Signal Name | Type | Bit Width | Description |
|---|---|---|---|
| clk | In | 1 | Rising Edge active clock |
| reset | In | 1 | Asynchronous reset to put the unit into a known state |
| iReqToMemory | In | 1 | Signal is a 1 if a request is made to read or write to memory, 0 otherwise |
| iReadWriteToMemory | In | 1 | Signal is a 1 to signify a read, 0 for a write. Is only used when a request is made. |
| iWriteDataToMemory | In | 32 | If a write request is made, this bus contains the data that will be written. |
| iAddressToMemory | In | 23 | The address in memory that is to be manipulated |
| iMemoryOpToMemory | In | 2 | If set to "00", do a 4 byte operation<br>If set to "01", do a 1 byte operation<br>If set to "10", do a 2 byte operation<br>If set to "11", do a 8 byte operation (See Chapter 5) |
| oMemoryData | Out | 32 | If a read request is made, this bus contains the data. The data is valid the cycle after the address was presented. |
| oMemoryAck | Out | 1 | Set to 1 if the oMemoryData bus is carrying valid data for the pipeline, 0 otherwise. |

If a 1 or 2 byte read access is made, the result is shifted to the least significant bits, so the correct value would be stored in a register. Similarly, on a write operation, the 1 or 2 byte value to be written is given in the least significant bits and shifted to the appropriate location in the 32-bit value. The output on a read would be valid on the next rising clock edge. This data is also signified when the oMemoryAck line is set. On a write operation, the acknowledge signal signifies success.

## 3.2 Memory Controller

Since multiple units need access to the memory, their requests must be arbitrated. The pipeline needs memory access to retrieve/modify data values. The instruction cache needs to read instructions from memory. The NIU also needs to read or write data values to memory. However, the pipeline is not deep enough to allow this unit to be synchronous. If it is activated by a clock edge, then the pipeline would not receive the read value until after it should be written to a register. Therefore, this block is asynchronous and is triggered whenever a request signal changes or the acknowledge signal from memory changes. Figure 3.2 below shows the inputs and outputs to each of the three blocks and to memory.

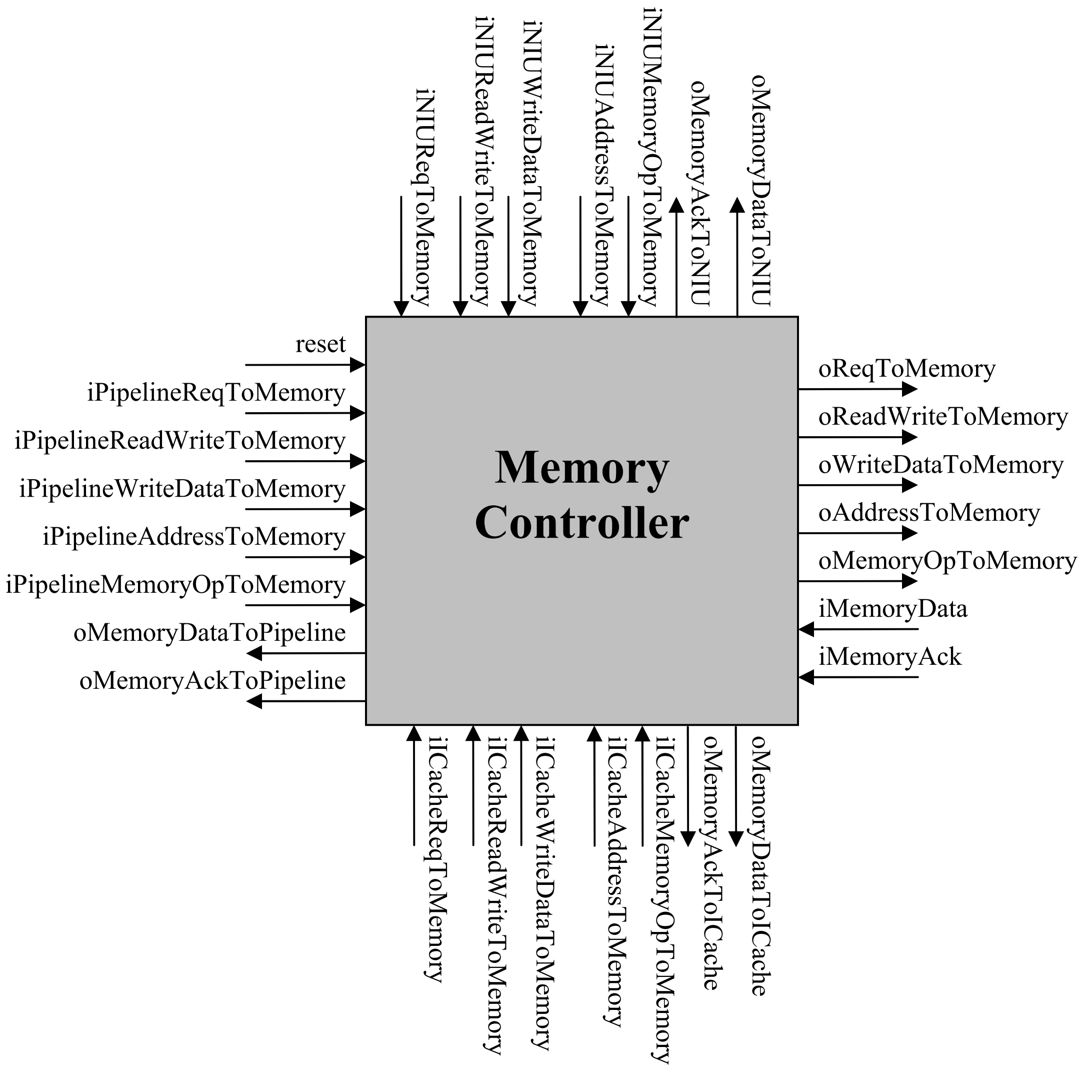


**Figure 3.2**: A block diagram of the memory controller.

If two or three blocks request access at the same time, a method is needed to determine which block has the highest priority. The priorities are chosen with the objective of limiting the number of clock cycles that the pipeline has to stall. This measure is used because contexts are executed in non-preemptive fashion. Therefore, the faster one thread finishes, the faster the next one can start/continue its execution. As a result, the pipeline receives the highest priority, so its operations are unaffected. Then, the instruction cache has the next highest priority. The only time the cache can access memory is on a cache miss, since the pipeline needs an instruction to continue thread execution. Finally, the NIU received the lowest priority. When the NIU accesses memory, it cannot affect the execution of a context unless it wrote to a synchronization variable. If so, the context waiting for a sync would suspend, and the NIU would be able to access the memory before the pipeline could restart after a context switch. In addition, a message can back up in the network for a few cycles without harming overall performance.

The implementation of this block multiplexes the three inputs over one channel. The interesting part of this implementation is in the acknowledgement execution. See Table 3.2 for a list of the acknowledge signals and refer to Table 3.1 for a reference to the memory interface. Since this unit is asynchronous and its outputs are stable by the next rising clock edge, the acknowledge signal is not used to indicate valid data, but is used as an indicator of the acceptance of the request. Figure 3.3 depicts the timing of a memory operation through the memory controller.

The Context Management Table (CMT) is memory mapped to addresses 0xFFFFFF00 - 0xFFFFFFFF. Therefore, the port from the memory controller to the memory is also split to the context logic. When the address is in the appropriate range, the CMT logic responds with the appropriate value while the memory ignores the request.

**Table 3.2**: The signal names with corresponding descriptions of the memory controller block. The other signals are those of a memory interface (See Chapter 3.1) with the appropriate naming convention.

| Signal Name | Type | Bit Width | Description |
|---|---|---|---|
| Reset | In | 1 | Asynchronous reset to put the unit into a known state |
| oMemoryAckToICache | Out | 1 | Set to 1 if the oMemoryData bus is carrying data for the instruction cache, 0 otherwise. |
| oMemoryAckToNIU | Out | 1 | Set to 1 if the oMemoryData bus is carrying data for the NIU, 0 otherwise. |
| oMemoryAckToPipeline | Out | 1 | Set to 1 if the oMemoryData bus is carrying data for the pipeline, 0 otherwise. |
| oMemoryAck | Out | 1 | Set to 1 if the oMemoryData bus is carrying data for the pipeline, 0 otherwise. |

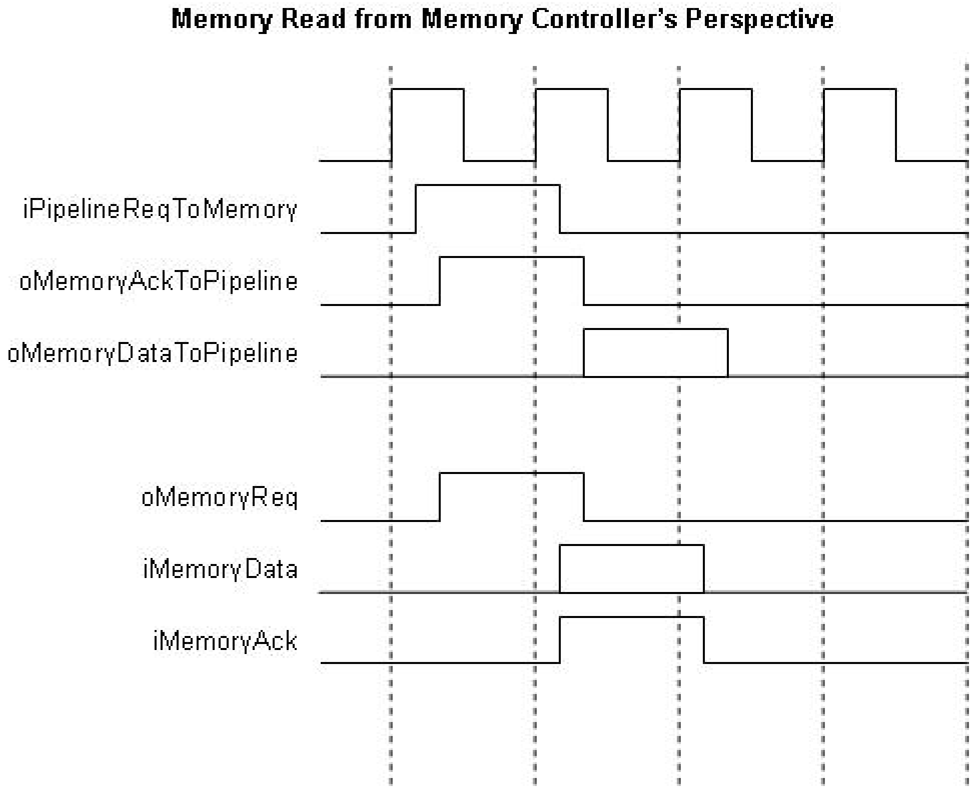


**Figure 3.3**: When a read request is made, the acknowledge signal is received on the next clock edge. If the acknowledge signal is positive, the actual data is valid two cycles after the request was made.

If the request is accepted, the operation is completed by the next rising clock edge. In other words, it is known if the memory operation is permitted on the clock edge after it is requested, but would not be completed until two clock edges later. This design also means that requests to memory did not need to be buffered since a unit can request again as soon as it is denied.

The acknowledge signal from memory triggers the memory controller. On a read operation, data is passed back to the appropriate unit and the acknowledge signal to the requesting unit is set to '0'. Once the requester knows the operation is permitted, data is known to be valid on the next clock cycle.

Another feature of the memory controller is that it abstracts away the memory interface from the three requesting units. When a third party vendor memory design is used, only the memory interface from the controller to the memory will need to change. The three requesting units can keep the same interface, and the memory controller can be changed to translate the current interface to the new one

## 3.3 Instruction Cache

Even though a data value can be read from the memory in one cycle, the memory is not designed to be dual ported. If the memory has two separate access points with collision control, then one port can be dedicated to be the instruction cache and the other to the pipeline and the NIU. Since that is not the case, a separate storage space is needed for the instructions since one needs to be read each clock cycle for an executing context. Therefore, an instruction cache was created with the interface shown in Figure 3.4 and described in Table 3.3.

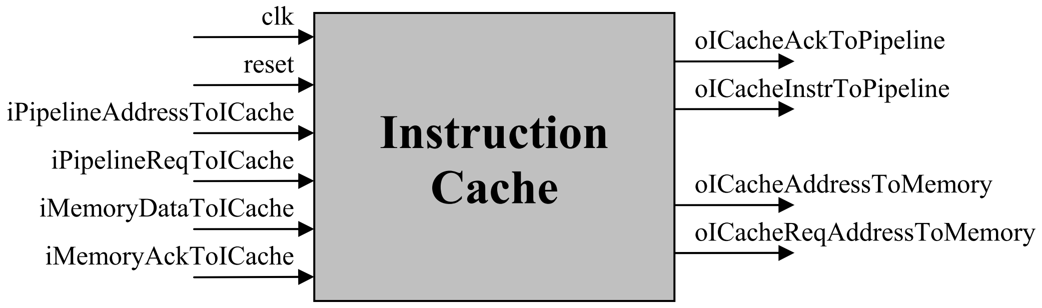


**Figure 3.4**: A block diagram of the instruction cache for the SCMP parallel computer.

When it is possible to fabricate the SCMP parallel computer, this block will probably be substituted with a third party instruction cache. Thus, this block was used for simulation and testing of the hardware and the interfaces. It was important, for simulation purposes, that the instruction cache be flexible so the different types of cache line replacement could be attempted. Therefore, the instruction cache was implemented so that it could be direct mapped, x-way set associative, and fully associative by changing the constants in the icacheconf.h header file.

The constants that control the instruction cache are as follows: iCacheSizeLength, controlled the size of the instruction cache in bytes; iCacheLinesPerSetLength, determines the associativity of the cache; and iCacheInstPerLineLength, sets the number of instructions that are in each cache line. Each of these cache parameters determines its actual value by shifting a ‘1’ to the left the number of times indicated in these constants. For example, if iCacheSizeLength is 15, then iCacheSize would be 32 KB, or $2^{15}$.

**Table 3.3**: The signal names and descriptions for the instruction cache block.

| Signal Name | Type | Bit Width | Description |
|---|---|---|---|
| clk | In | 1 | Rising Edge active clock |
| reset | In | 1 | Asynchronous reset to put the unit into a known state |
| iPipelineAddressToICache | In | 23 | The address of the instruction the pipeline is trying to fetch. |
| iPipelineReqToICache | In | 1 | When set to a ‘1’, the pipeline is making a request for an instruction. On a ‘0’, no request is made. |
| iMemoryDataToICache | In | 32 | On a cache miss, the data for the cache line |

| | | | |
|---|---|---|---|
| | | | will come back on this port. |
| iMemoryAckToICache | In | 1 | When set to a ‘1’, the data coming from memory is valid. When a ‘0’, the data is not valid. |
| oICacheAckToPipeline | Out | 1 | Set to a ‘1’ when there is a cache hit. ‘0’ otherwise. |
| oICacheInstrToPipeline | Out | 32 | The instruction that was fetched. |
| oICacheAddressToMemory | Out | 23 | On a cache miss, this is the address of the 4 byte value being fetched. |
| oICacheReqAddressToMemory | Out | 1 | Set to a ‘1’, when cache needs to fill a line. The signal is a ‘0’ otherwise. |

The important part of the instruction cache implementation is to understand what a set consists of and how many sets are in the cache. When a cache is x-way set associative, it means that there are x lines in a set. Those lines are filled associatively. Each line in the set has a valid signal to indicate when a line is full. And a line identifier is associated with each line to identify which instructions are in that line. Line replacement for each set is done in round robin fashion. The hope is that a context contains few enough instructions so that there would be a high locality of reference and adjacent lines could be reused often. Therefore, each set also needs a reference to which line was to be filled next.

The other important aspect of the instruction cache is how to calculate the number of sets needed to create the cache. For example, if the cache is 32 KB in size, it could hold 8 K of instructions (4 bytes per instruction). Assuming that each cache line has 8 instructions and the cache is 4-way set associative, then the number of sets in this cache is calculated as follows:

$$32KB * \frac{1instruction}{4B} * \frac{1line}{8instructions} * \frac{1set}{4lines} = 2^8 or 256 sets$$

Therefore, 8 bits are needed to identify a set in the cache. An address in the SCMP node is currently 23 bits (8MB of addressable space). The lowest 2 bits of an address must be 0 so that a request is made on an even 4 byte boundary. The next 3 bits are needed to identify which instruction is referenced from the cache line. Therefore, the line identifier needs to be 23-8-3-2=10 bits.

When a request is made to the instruction cache, its implementation accesses the appropriate set and queries all of the lines in that set for a valid line ID match. If the line is present, then it is a cache hit. On a cache miss, the address requested and the address of the start of the corresponding line are saved. The nextLineToFill value is used to determine which line should be replaced in the set. The cache uses the request/acknowledge sequence described in Chapter 3.2 about the memory controller. Once the line is filled, the instruction can be positively acknowledged the next time it gets requested.

## 3.4 ALU

The Arithmetic Logic Unit (ALU) is a synchronous logic block, triggered by the falling edge of the clock, used for any integer based mathematical function. On a reset, this unit sets the acknowledgement signal to ‘0’. The figure below depicts the ALU as a black box with its inputs and outputs labeled. Table 3.4 then describes the use of each of the signals.

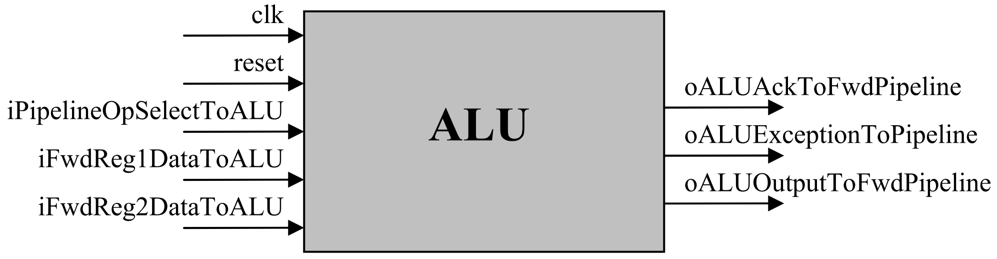


**Figure 3.5**: A block diagram of the ALU.

**Table 3.4**: Input and output signals for the ALU.

| Signal Name | Type | Bit Width | Description |
|---|---|---|---|
| clk | In | 1 | Rising Edge active clock |
| reset | In | 1 | Asynchronous reset to put the unit into a known state |
| iPipelineOpSelectToALU | In | 7 | The instruction opcode tells the ALU which operation to perform. Only one can be calculated at a time. |
| iFwdReg1DataToALU | In | 32 | Data on which operation is performed |
| iFwdReg2DataToALU | In | 32 | Data on which operation is performed |
| oALUOutputToFwdPipeline | Out | 32 | The result of the operation performed. |
| oALUExceptionToPipeline | Out | 32 | Set to the ESTATUS value of the corresponding exception. |
| oALUAckToFwdPipeline | Out | 1 | Acknowledge that was set when the data and exception values were valid. ‘0’ otherwise. |

The pipeline controls when the ALU performs an operation via the OpSelect input. The pipeline identifies which operations the ALU is responsible to perform since all ALU opcodes occurred sequentially in the instruction set. For a complete list of instructions, see Appendix A.

The ALU is a simple block because it is synchronous and each ALU function is implemented behaviorally without optimizations. When an opcode that the ALU is responsible for is present at the OpSelect input, the appropriate operation is performed on Reg1 and Reg2. It is assumed that all operations except multiply, divide, and modulo take only one clock cycle to execute. The other instructions took 5, 19, and 19 cycles respectively. Once an operation begins, the pipeline must wait until the acknowledge signal is set to be sure that the output data is valid. Since the pipeline does not execute instructions out of order, it is forced to stall until the ALU operation is completed.

## 3.5 NIU

The Network Interface Unit (NIU) is used to create flits to inject into the network and to accept and perform operations on flits from the network. Both the inject and eject functions of the NIU are further divided into two parts. When sending a message into the network, it can be generated from either the pipeline, using register values, or from the on chip memory, reading data. When a message arrives at a node, it can either create a new context, using the CMT/context interface or write values to memory, using the memory interface. Figure 3.6 below is a block diagram of the NIU.

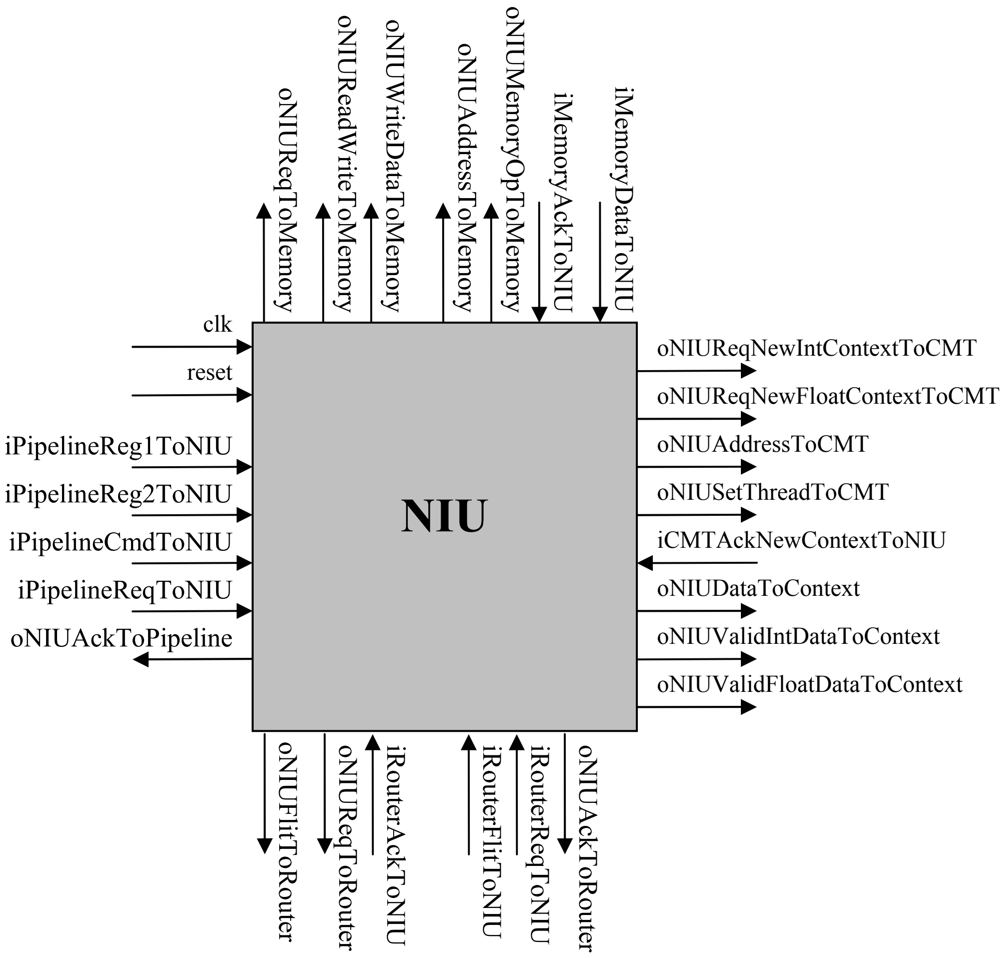


**Figure 3.6**: A block diagram of the Network Interface Unit (NIU).

The signals from the NIU to the memory controller are identified in Chapter 3.1, which describes the memory interface. Similarly, those signals between the NIU and the context management logic will be discussed in Chapter 4. Since those signals are described elsewhere, they will not be discussed here. This section describes the operation of the NIU in terms of the NIU/pipeline and NIU/router signals. Table 3.5 describes these signals in detail.

**Table 3.5**: The signal names and descriptions for the NIU of the SCMP node.

| Signal Name | Type | Bit Width | Description |
|---|---|---|---|
| clk | In | 1 | Rising Edge active clock |
| reset | In | 1 | Asynchronous reset to put the unit into a known state |

| | | | |
|---|---|---|---|
| iPipelineReg1ToNIU | In | 32 | A data value from a register to be sent in a message. |
| iPipelineReg2ToNIU | In | 32 | A second value from a register to be sent in a message. |
| iPipelineCmdToNIU | In | 7 | The command tells the NIU which data values are valid and what operation to perform on the data. |
| iPipelineReqToNIU | In | 1 | When set to '1', the input command and data are valid. '0' otherwise. |
| oNIUAckToPipeline | Out | 1 | Set to a '1' when the request is satisfied. If a '0', the requesting thread will suspend. |
| oNIUFlitToRouter | Out | 34 | A flit contains a 32 bit data value, a bit to indicate the start of a message, and a bit to specify the end of a message. |
| oNIUReqToRouter | Out | 1 | When set to a '1', the flit data is valid. '0' otherwise. |
| iRouterAckToNIU | In | 1 | If set to a '1', the router has stored the flit in a buffer successfully. '0' otherwise. |
| iRouterFlitToNIU | In | 34 | A flit arriving at the node. |
| iRouterReqToNIU | In | 1 | When set to a '1', the flit data is valid. '0' otherwise. |
| oNIUAckToRouter | Out | 1 | Set to a '1' when the NIU accepts the flit. When a '0', the flit remains buffered in the network. |

When the pipeline executes any of the send instructions, data is sent to the NIU from the pipeline. The type of instruction is identified by the <iPipelineCmdToNIU> signal, which is the opcode decoded by the pipeline. One instruction can send up to two flits. To accept two flits per clock cycle, either two flits need to be sent in one clock cycle or flits need to be buffered in the NIU. For this implementation, it is assumed that the router is capable of receiving two flits in one clock cycle. Therefore, a flit could be sent out on the rising and falling edge of the node clock and no buffering is necessary. If this method is found not to be realistic, then buffers can be added to this unit.

Only one message at a time can be constructed by the NIU and injected into the network. This limitation means that once a sendh instruction is received, a sende instruction is required before another sendh. In other words, after a message begins, any other request to start a new message is denied. Both the send and sendm instructions have a version of the instruction that indicates the tail flit of a message. The end of a message is known by the opcode passed to the NIU.

When a sendh instruction is detected, the <iPipelineReg1ToNIU> bus contains the address needed by either the thread message or the data message. The <iPipelineReg2ToNIU> signal contains the following data: the type of message is in the highest bit position, the destination node is in the next 6 most significant bits (for 64 nodes), and the lowest 11 significant bits contain the immediate stride value, if needed. The sendh instruction generates two flits. First, the header flit contains the type of message and the routing information needed to reach the destination node. Second, the address flit contains the starting place for either type of message.

When a sendm instruction is detected, then the <iPipelineReg1ToNIU> bus contains the start address to read from memory. Also, the stride value between memory accesses and the number of values to read are given by the <iPipelineReg2ToNIU> signal. Only one flit is injected into the network per clock cycle because the memory can only process one request per clock cycle.

On a send instruction, <iPipelineReg1ToNIU> contains the register value to inject in to the network. Similarly, when a send2 instruction is detected, both the <iPipelineReg1ToNIU> and <iPipelineReg2ToNIU> buses contain register values to send as part of a message.

A message is guaranteed to be sent in the same order the instructions are issued. For example, if a sendm is followed by a send2 instruction, then the sendm must complete its execution before the send2 instruction is accepted by the NIU.

Similar to the inject portion of the NIU, the eject section can only accept one message at a time. When a header flit arrives at the NIU from the router, the type of message is determined and, if necessary, the stride value is stored. Once the address flit arrives at the NIU, the incoming message manipulates the appropriate interface, either the memory interface or the CMT/context interface. If a data flit arrives at the NIU before its associated header and address flits are successfully written to memory or allocated to a context, then the data flit is denied access, and the flit will be buffered by the router.

With the completion of the NIU implementation, the major components of the SCMP node were designed and implemented, and the messaging, sequencing, and interfaces between the various components were defined. System C models of the on-chip memory and instruction cache were also developed. The heart of the SCMP node and the focus of this thesis were the design and implementation of the SCMP memory controller, the Network Interface Unit, and the ALU. The remaining components of the SCMP node, the CMT and the contexts, are responsible for managing the threads on a node and are described in the next chapter.

# Chapter 4 Contexts and Context Management

As was previously explained, the context management scheme is broken up into two main blocks: the Context Management Table (CMT) and control, and the Context Register File (CRF) and control. This division of hardware allows for the parts to be developed and tested both individually and concurrently. Figure 4.1 depicts the overall context management scheme with the CMT and CRF sub-blocks.

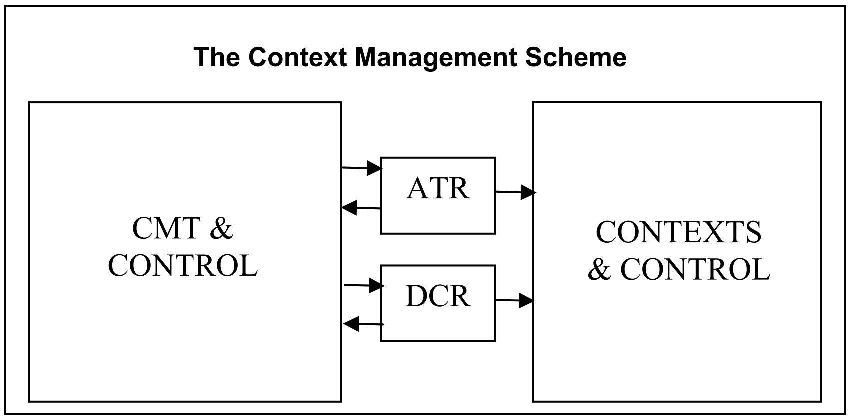


**Figure 4.1**: A block diagram of the context management scheme.

The CMT and Control block is responsible for allocating and deallocating contexts and for scheduling threads. Meanwhile, the Contexts and Control block is in charge of accessing the registers of each context. These blocks cannot, however, be separated because both interact with the same control registers. This chapter presents a detailed implementation of the hardware for each of the sub blocks and the shared hardware between the two. The Contexts and Control block was designed and both the CMT and Contexts were implemented in System C as a part of this research effort.

## *4.1 Shared Hardware*

There are two main registers that are shared between the CMT control and the CRF control: the Active Thread Register (ATR) and the Data Context Register (DCR). Each of these registers is 4 bits long in order to access the 16 possible contexts. The actual length is determined by the constant that set the number of contexts in the node.

The ATR contains the context identifier of the currently executing thread. The CMT needs this value for many of its commands (see Chapter 4.2). For example, when a thread ends, the CMT control needs to modify the CMT entry corresponding to that thread. Therefore, the ATR is used as an index into the CMT so the end instruction can be executed. Similarly, the CRF control uses the ATR as an index into its register set (see Chapter 4.3). As an example, if an instruction reads or writes to a register(s), then the ATR is used to access the appropriate register set in the CRF. The ATR requires a valid signal to ensure that a context is currently executing. If the ATR is not valid, then a context to execute cannot be found and one is searched for during each clock cycle.

A context may increase the number of registers it uses through the DCR. If a program requires more that its own 32 registers, it can allocate a second context to itself and use these registers in its own calculations. In this case, the DCR is used as the context identifier of this second context. When a register in this extended range (32-63) is accessed, the DCR is used as the index into the CRF. Unlike the ATR, the DCR does not require a valid signal because context 0 cannot be used as a data context (see Chapter 5.2). Although the CMT and contexts cannot be separated, the following sections describe each block's implementation in detail.

## *4.2 Context Management Table and Control*

Each entry in the context management table has four parts:

- The Alloc bit - If the context is allocated, this bit is set to 1. The context is free for use as a data context or a thread context, if this bit is set to 0.

- The Thread bit – If the context was allocated and this bit is a 1, the context is a thread and will be scheduled for execution. When the context is allocated and this bit is a 0, the context is currently being used as a data context, or is being filled by the Network Interface Unit (NIU).

- The DCR - A 4 bit value that stores the context ID of a thread's data context. If the value is 0x0, the context does not have a data context. (The number of bits in this field is determined by a constant, which is based on the number of contexts on each processor.)

- The IP Address - A 23 bit value that contains the instruction pointer of a thread context. This value does not have meaning if the context is a data context. (The number of bits in this field is determined by a constant, which is based on the size of memory for each processor.) It is important to note that the 2 lowest significant bits will always be "00" because instructions are 4 byte aligned.

There are a total of 29 bits in each CMT entry. Each CMT entry is chosen to be 32 bits wide to allow easy expansion of either the IP, if the memory size increases, or DCR field, if the number of contexts on a processor is increased. Figure 4.2 shows the breakdown, by bit, of these fields in a CMT entry.

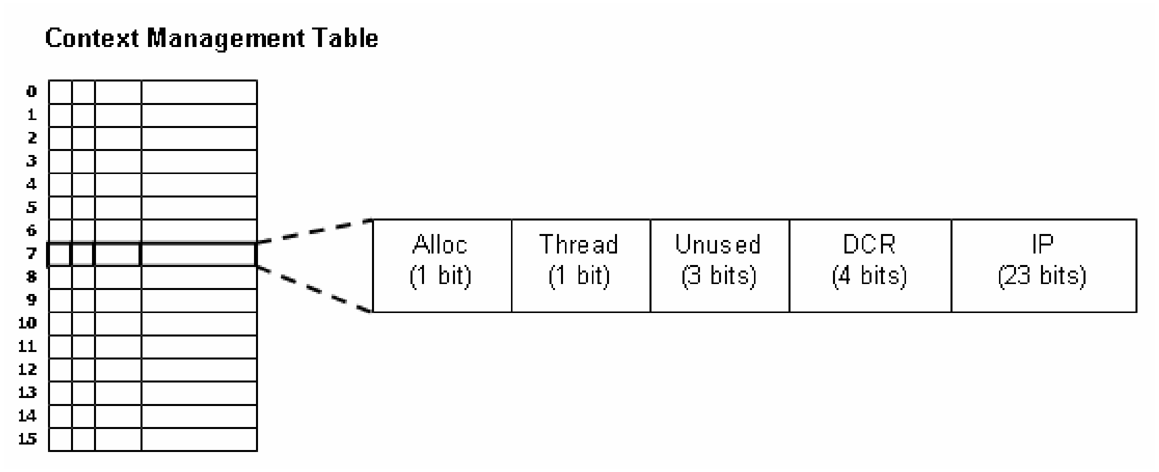


**Figure 4.2**: A visual representation of a CMT entry.

Hardware support is used to manage the threads of a node to minimize the number of clock cycles during a context switch [9]. In theory, from the time the pipeline detects an end or a suspend instruction to the time when the pipeline receives another valid instruction from the instruction cache, it takes four clock cycles. This calculation assumes that there is another context ready for execution and the instruction requested is already in the instruction cache. Figure 4.3 shows the timing of this ideal context switch from the pipeline's perspective.

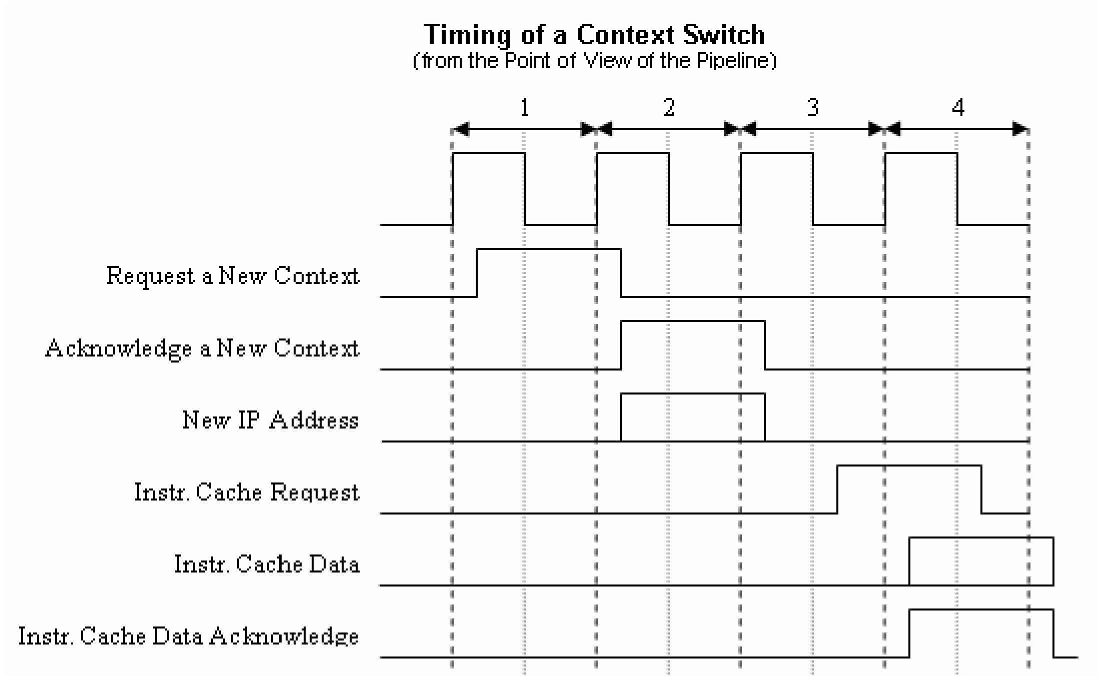

Timing of a Context Switch
(from the Point of View of the Pipeline)
1
2
3
4
Request a New Context
Acknowledge a New Context
New IP Address
Instr. Cache Request
Instr. Cache Data
Instr. Cache Data Acknowledge

**Figure 4.3**: The timing of a context switch.

Both the pipeline and the NIU can control the bits of a CMT entry. The pipeline can control any entry in the CMT. The CMT is memory mapped, so the pipeline can read and write to an entry in the CMT. However, a context cannot change its own entry. Also, the pipeline can use the alloc and free instructions to modify the CMT. The Network Interface Unit (NIU) can also allocate a context by using a thread message. Figure 4.4 depicts the CMT and control with the interfaces to each of the units that could access it. Then, the input and output signals of the CMT control are described in the sections that follow.

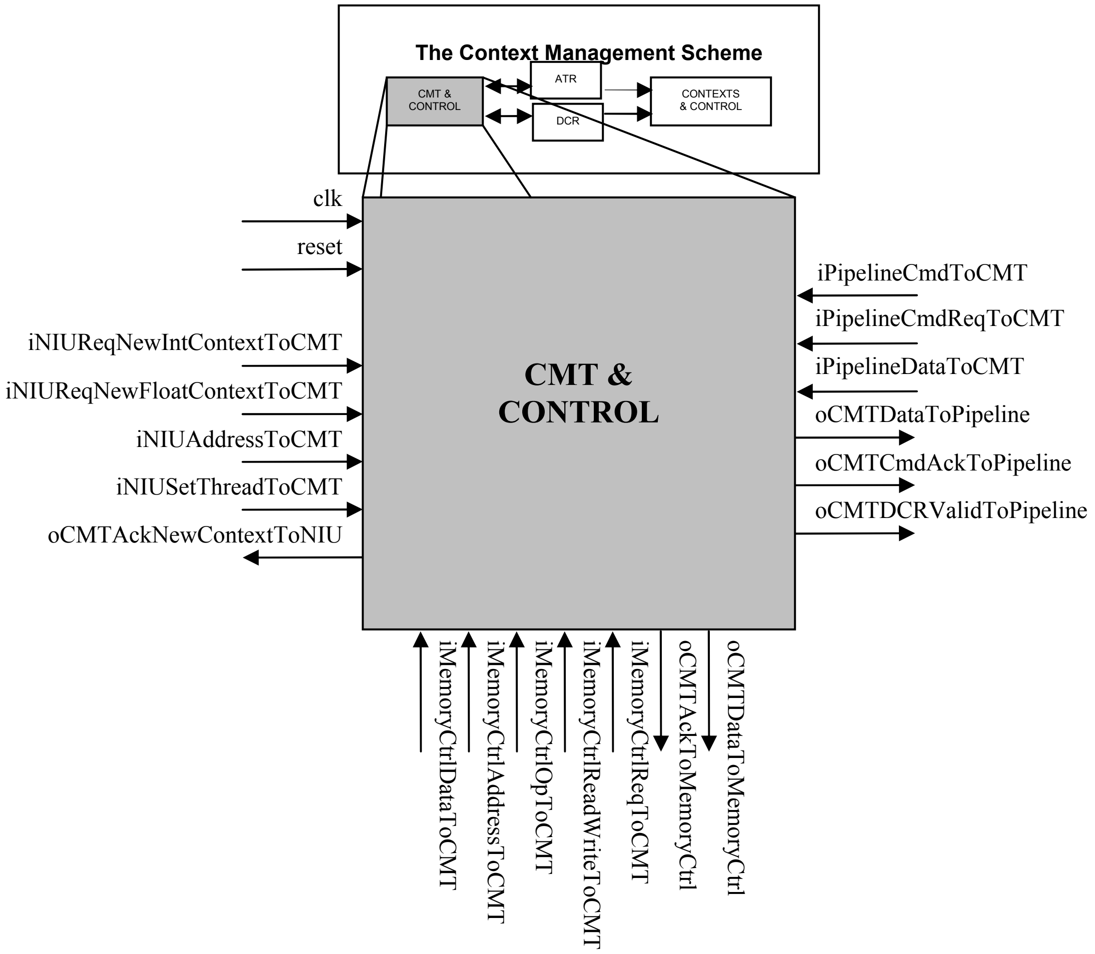


**Figure 4.4**: A block diagram of the CMT & control part of the context management scheme.

| Signal Name | Type | Bit Width | Description |
|---|---|---|---|
| clk | In | 1 | Rising Edge active clock |
| reset | In | 1 | Asynchronous reset to put the unit into a known state |

## Memory Controller Accesses to the CMT

The CMT is memory mapped to addresses 0xFFFFFF00 - 0xFFFFFFFF. Since it is memory mapped, the pipeline is able to read from and write to the CMT just as if it was memory. Table 4.1 below defines the interface between the memory controller and the CMT control.

**Table 4.1**: Describes the interface between the CMT and the memory controller.

| Signal Name | Type | Bit Width | Description |
|---|---|---|---|
| iMemoryCtrlReqToCMT | In | 1 | Set to a '1' when a read or write request is made to the CMT. '0' otherwise. |
| iMemoryCtrlReadWriteToCMT | In | 1 | When a request is made, set to a '1' for a read, a '0' for a write. |
| iMemoryCtrlOpToCMT | In | 2 | If set to "00", do a 4 byte operation<br>If set to "01", do a 1 byte operation<br>If set to "10", do a 2 byte operation<br>If set to "11", do nothing |
| iMemoryCtrlAddressToCMT | In | 23 | The address that is read from or written to when a request is made. |
| iMemoryCtrlDataToCMT | In | 32 | The data written to the CMT. |
| oCMTAckToMemoryCtrl | Out | 1 | The active high acknowledge signal when a request is made to read/write to the CMT. |
| oCMTDataToMemoryCtrl | Out | 32 | The data read from the CMT. |

The CMT control checks for a valid memory access each clock cycle. If a request was made, <iMemoryCtrlReqToCMT> is set to '1', the read or write operation, determined by <iMemoryCtrlReadWriteToCMT>, occurred from the address specified on the <iMemoryCtrlAddressToCMT> bus. The above table defines the different values for <iMemoryCtrlOpToCMT>, which controls how many bytes of the CMT are accessed. The CMT is byte addressable and allows 3 types of reading/writing: 1-byte, 2-byte, and 4-byte. Therefore, CMT entry 0 is represented by addresses 0xFFFFFF00 through 0xFFFFFF03, inclusive. Thus, there are enough addresses to expand to 64 contexts. Currently, only 16 of these 64 possible entries are used, so some of these addresses are unused and invalid. However, the number of contexts per node is configurable, so the remainder of the address space can be used. If an address is outside the bounds of the CMT, all 0's are returned. All memory exceptions, out of the bounds of the CMT or bus alignment, are handled by the memory controller and will not be discussed here.

However, there is one type of memory access that is not allowed. If a write occurs to the currently executing thread (context ID found in the ATR), it cannot change either the Alloc or Thread bit. This write is not allowed since the pipeline would continue its execution even though the context had been freed or the type of the context had changed from thread to data. In addition, the pipeline can change these bits directly via the end instruction (see the following section on Pipeline Accesses to the CMT). The other bits of the CMT entry for the currently executing thread can be modified. Since there are unused bits, additional information about a thread can be written to the CMT. For example, a thread can write to an unused bit in its CMT entry to indicate if it is sending a message or not. This information can be useful, in the future, if contexts could be temporarily stored in memory to allow the network to inject another thread context. If a thread is sending a message, it could be detrimental to remove it from those being scheduled since no other thread could send a message until that one finished.

**Pipeline Accesses to the CMT**
The pipeline can perform the following operations on the Context Management Table: allocate context, free context, end context, suspend context, read special register, and write special register. Each of these operations is given a command code. Table 4.2 below shows the signal names that control the pipeline accesses to the CMT and its control. The command code is 3 bits long since there are 6 commands to perform. The logic for allocating a context is discussed later since both the pipeline and the NIU can perform that particular function.

**Table 4.2**: Signals that control pipeline accesses to the CMT.

| Signal Name | Type | Bit Width | Description |
|---|---|---|---|
| iPipelineCmdToCMT | In | 3 | '000' – nothing<br>'001' – alloc instruction<br>'010' – free instruction<br>'011' – end instruction<br>'100' – suspend instruction<br>'101' – read special register instruction<br>'110' – write special register instruction<br>'111' – nothing |
| iPipelineCmdReqToCMT | In | 1 | Set to a '1' to indicate a valid command. '0' otherwise. |
| iPipelineDataToCMT | In | 32 | When a writesr instruction, a free instruction, or a context suspends, this data is valid. |
| oCMTDataToPipeline | Out | 32 | When data is valid, this bus contains the IP address of the next context to execute or the data from the readsr instruction. |
| oCMTCmdAckToPipeline | Out | 1 | Set to a '1' when a command is successful or data is valid. '0' otherwise. |
| oCMTDCRValidToPipeline | Out | 1 | Set to a '1' when a context has a valid DCR. '0' otherwise. |

When the free command is sent to the context management control, the pointer to the context that is to be freed is read from <iPipelineDataToCMT>. If the value read is not the same as the currently executing thread, then the Alloc bit of the context ID is set to 0 and <oCMTAckCmdToPipeline> is set to 1 for success. If the value read is the same as the executing context ID, then the request is denied and <oCMTAckCmdToPipeline> is set to 0 for failure.

The end command is used as the free command for the currently executing thread. The Alloc bit of the currently executing thread is set to 0 and <oCMTAckCmdToPipeline> is set to 0 until a new thread to execute is found. If there is not a thread executing, then no action is taken in the CMT, and <oCMTAckCmdToPipeline> is set high when a new thread is found for execution. When the end command is acknowledged, <oCMTDataToPipeline> is set to the IP address of the thread that is starting/resuming its execution.

A suspend instruction from the pipeline to the CMT is similar to the end instruction. Both types of commands use the same method to find the next thread to execute and acknowledge the command to the pipeline. However, the CMT is modified differently. The Alloc and Thread bits both remain set at '1'. The DCR is copied from its shared register back to the CMT entry. When the suspend instruction is sent, the IP address of the next instruction to execute when the context is rescheduled is sent from the pipeline to the CMT control on the <iPipelineDataToCMT> signal. This input is written to the appropriate bits in the suspending context's CMT entry. See Figure 4.2 above for the bit locations of the DCR and IP address in a CMT entry.

The pipeline also has the ability to manipulate the shared registers and read some constant information about a node. A program can read both the ATR and DCR and write to the DCR. The ATR is read, for example, so a program would not write to that CMT entry and mistakenly free it. The DCR can be read to determine if the appropriate data context will be accessed by an instruction referring to registers 32 through 63. Since the alloc instruction returns the context ID of the allocated context, the DCR can be written so that a context can allocate as many contexts as it needs.

There are some constant registers that are stored in the context management control that can be read. The node identification register (NIR) stores the numeric value of the processor. For example, in an SCMP system with 64 processors, the node IDs range from 0 through 63. When a processor sends a function to execute on another processor, it must include the node ID so the remote function can reply to the appropriate processor. The number of nodes in each row is given in the xDim register. Similarly, the register for the number of nodes in the y-direction is called yDim. These values can also be read using the read special register hardware. The NIR is used with xDim and YDim to locate a node on the chip. As an example, in the Figure 4.5 below xDim is 4 and yDim is 4. The NIR of this processor is 14, and the SCMP processor has 16 nodes.

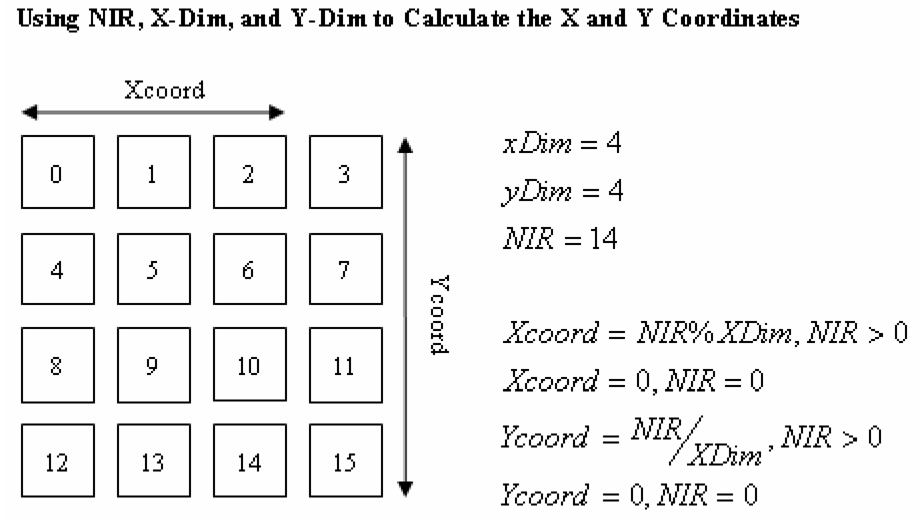

Using NIR, X-Dim, and Y-Dim to Calculate the X and Y Coordinates
Xcoord
Ycoord
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
xDim = 4
yDim = 4
NIR = 14
Xcoord = NIR%XDim, NIR > 0
Xcoord = 0, NIR = 0
Ycoord = NIR/XDim, NIR > 0
Ycoord = 0, NIR = 0

**Figure 4.5**: Shows how to find the location of a node on the SCMP parallel computer.

**NIU Accesses to the CMT**

The Network Interface Unit (NIU) accesses the Context Management Table when a thread message arrives at a node. See Table 4.3 for an explanation of the interface between these components. When a message arrives at a node, <iNIUReqNewIntContextToCMT> is set to ‘1’. The <iNIUReqNewFloatContextToCMT> signal is discussed in Chapter 5.1, which refers to floating point operations. If a request is made and there is a free context available, the Alloc bit of that context’s entry in the CMT is set to ‘1’, and <iNIUAddressToCMT> is the instruction pointer for the new context. Its value is copied into the appropriate bits in the CMT entry and <oCMTAckNewContextToNIU> is set to ‘1’.

**Table 4.3**: A description of the interface between the NIU and the CMT and its control.

| Signal Name | Type | Bit Width | Description |
|---|---|---|---|
| iNIUReqNewIntContextToCMT | In | 1 | Set to a ‘1’ if the NIU receives a thread message and needs to allocate a context. ‘0’ otherwise. |
| iNIUReqNewFloatContextToCMT | In | 1 | Set to a ‘1’ if the NIU has knowledge that floating point data will be received (see Chapter 5). ‘0’ otherwise. |
| iNIUAddressToCMT | In | 23 | The IP address from the address flit of a message. This value is written to the CMT entry which was allocated for integer data. |
| iNIUSetThreadToCMT | In | 1 | When the tail flit of a thread message is received, this flag is set to indicate the end of the message, so the context can be scheduled for execution. |
| oCMTAckNewContextToNIU | Out | 1 | Set to a ‘1’ to indicate that a context(s) is successfully allocated. ‘0’ otherwise. |

This request for a free context creates a potential problem. If the pipeline and the NIU request to allocate a context at the same time, what happens? Let us consider the following possibilities: (1) the NIU receives access to the free context and (2) the pipeline receives access to the free context.

First, if the NIU is granted permission to access the free context, the pipeline command would be denied access until another free context can be found. When the pipeline command is denied access to a context, that context will suspend and re-execute the alloc instruction when that context is restarted.

Second, if the pipeline is granted the free context, the NIU would be left without a free context to use until another is found. However, the network has buffers to hold the requesting message. These buffers allow the NIU and router to receive messages and store them without slowing down the network. The worst case is if a free context is not found, and messages do back up in the network. In the case when a free context can be found, the message would only back up in the network for a cycle or two. The pipeline continues its execution, since it is able to allocate a free context. This method is used because it provides continued pipeline execution and minimal delay to the NIU.

After the NIU receives a positive acknowledgement that a free context is allocated for the thread message, the incoming flits are written to the register block associated with that context. Once the NIU receives the tail flit, the <iNIUSetThreadToCMT> signal is set to '1'. This active high signal signifies that the thread message is finished accessing the register block and is ready for execution. The thread bit of the CMT entry that was allocated to the NIU is set to '1', and the context is scheduled for execution the next time around in the round robin.

To prepare itself for an access from the pipeline or the NIU, two basic operations are performed in the background by the CMT: (1) find the next context to execute and (2) find the next free context. It is important to find the next thread to execute in order to minimize the amount of time it takes to switch contexts. Similarly, finding the next free context limits the number of cycles that it takes to allocate a new context when one is requested. These two processes will be discussed in the following sections.

### 4.2.1 Find a Context to Execute

When the currently executing thread ends, a different thread is able to execute in the processor's pipeline. An algorithm for finding the next context to execute is needed for use in the CMT control. Threads are scheduled in a round robin, non-preemptive fashion. This type of scheduling prevents thread starvation unless a thread enters an infinite loop, which is the programmers fault.

Two ways to locate the next thread to execute were considered during the design effort. First, a counter is used to locate the next thread available for execution. The counter, or next thread to execute register (NTE), starts at the current thread ID, and increments itself until a context is found that has both the Alloc bit and Thread bit set to '1'. When such a thread is found, a valid signal for the NTE register is set. The counting occurs once per clock cycle to avoid a race condition between the NTE and its associated valid signal. An overflow of the counting register causes the scheduling to continue in a round robin manner by making the counter be just wide enough to cycle through all the contexts once. The problem with this approach is the amount of time it potentially takes to find the next executable thread. In the worst case, if a context is ready to execute, it could take up to the <number of contexts> clock cycles to find the next thread to execute. Therefore, with a small number of contexts the time delay will be short, but it will increase linearly with the number of contexts.

To overcome this potential time delay, a second design approach to finding the next thread to execute was developed and implemented. This method needed to be fast and not dependent on the number of contexts in the processor. A new register was added to the CMT management logic that latched a boolean value for each of the contexts in the node. This register, called activeContexts, contains one bit for each entry in the CMT. If a context is available to execute, its corresponding bit in the register is set to '1'. For example, if context 5 has its Alloc and Thread bits set to '1', then in the latched register, bit 5 would be set to 1. Conversely, if context 3 did not have both its Alloc and Thread bit set to '1', then bit 3 of activeContexts is a '0'. Context 0 occupies the lowest significant bit of the latched values. Now, if this register is an integer and its value is 0, then there is not a context to execute and values are re-latched. If the activeContexts register value is 1, then context 0 is executed. If its value is 2 or 3, then context 1 is executed. If its value is in the range 4-7, then context 2 is executed, and so on. Once a context is chosen for execution, its bit in activeContexts is set to 0. This assignment assures that all threads have the chance to execute before a thread has the chance to execute a second time. Thus, it keeps the scheduling in a round robin fashion.

Since the activeContexts register contains latched values, the thread that is scheduled to execute may no longer have both its Alloc and Thread bits set to ‘1’. For example, if activeContexts is latched and context 6 is scheduled as the next context to execute, the currently executing thread frees context 6 by setting its Alloc bit to ‘0’, even though it is still scheduled as the next context to execute. Therefore, before a thread actually begins to execute, both its Alloc and Thread bits must be verified to still be ‘1’.

A second potential problem was thought to be identified in using this context scheduling method. If the activeContexts register is latched, a new context cannot be scheduled to execute until the activeContexts register is re-latched. This situation does not create any deadlock or process starvation, as initially thought, since the thread is scheduled the next time through the round robin scheme. So even though this situation exists, it does not create a scheduling problem.

This type of scheduling, in the average case, takes just one clock cycle. In the worst case, when the Alloc bit or the Thread bit changed, it can take up to two more clock cycles to find a thread to execute. One cycle is used to latch new values, if necessary, and a second cycle is used to find the next thread to execute. However, this delay, in the worst case, is still less than the potential delay of the counter method. Therefore, this approach was implemented.

### 4.2.2 Find a Free Context

To allocate a context, the context management control needs to keep track of the next free context. The same two methods discussed for finding the next context to execute were also considered for finding the next free context. The counter method used the same theory, but with a different counter. For the latched method, only the Alloc bits were latched in a register called freeContexts.

When a free context is found, the context ID is stored in a register, called nextFreeContext, and nextFreeContextValid is set to ‘1’. If a request for a free context is made and nextFreeContextValid is ‘1’, as before, the entry in the CMT must be verified because nextFreeContext is set based on a latched value. If the context had not been allocated (the Alloc bit was ‘0’), then the context was allocated and nextFreeContextValid was set to ‘0’. Again, similar to finding the next context to execute, if a context is freed, it would not be found as a free context until freeContexts is latched again. This glitch does not cause any problems since the newly freed context will be used, but is delayed until the round robin scheme gets back to the freed context. The latch method is used since the delay in the worst case is less than the delay of the counter method.  These two operations allow the CMT to be used as a hardware context scheduler and allocation unit.

## 4.3 Context Register File and Control

Each entry in the CMT has its own set of 32 32-bit all purpose registers.  These registers are not part of main memory, but are stored in their own register file.  In the initial SCMP design, this file was used in a hierarchy, so when a register was referenced, it was loaded into a context cache.  The context cache stored the register value, the register ID, and the context ID from which the register was loaded.  This cache was the size of one context, 32 32-bit registers, so that each register of a context could be loaded, if necessary.  Using a context cache presented one large problem; when there was a context cache miss, the pipeline of the SCMP processor had to stall until the proper value(s) was loaded.  An objective of the revised design was to find an approach that would minimize or eliminate the register access delay.  To eliminate the delay from a context cache load, the registers needed to be accessed directly.  For this to be possible, register accesses needed to be atomic.  Therefore, the potential methods that could be used to access the registers of each context needed to be analyzed.

Two functional units have access to the register file: the pipeline and the network interface unit (NIU).  The pipeline accesses registers in the currently executing thread (the context ID is found in the ATR) and possibly its second data context (the context ID is found in the DCR).  However, the CMT entries of the ATR and DCR have their Alloc bits set and they are guaranteed to be different register sets.  As a result, accesses to the ATR and DCR are atomic.

The NIU also has access to the registers in the register file.  The question is could the NIU ever access the ATR's and/or the DCR's registers?  The answer is no.  The NIU, first, accesses the CMT to start a new thread.  Therefore, it would only access a context where the Alloc and Thread bits are initially 0.  Again, register operations are atomic.

Thus, a context cache was not needed and the registers could be accessed directly.  The register file, then, became as fast as a cache and its 'hierarchy' became one level.  Figure 4.6 depicts the implemented register file.

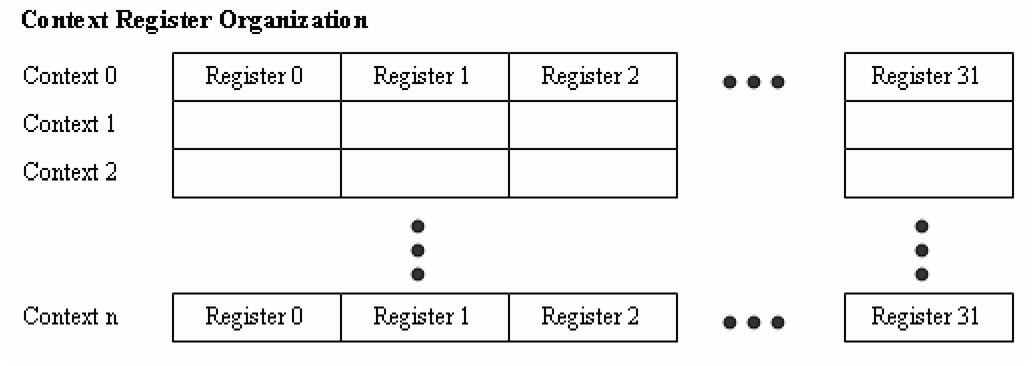
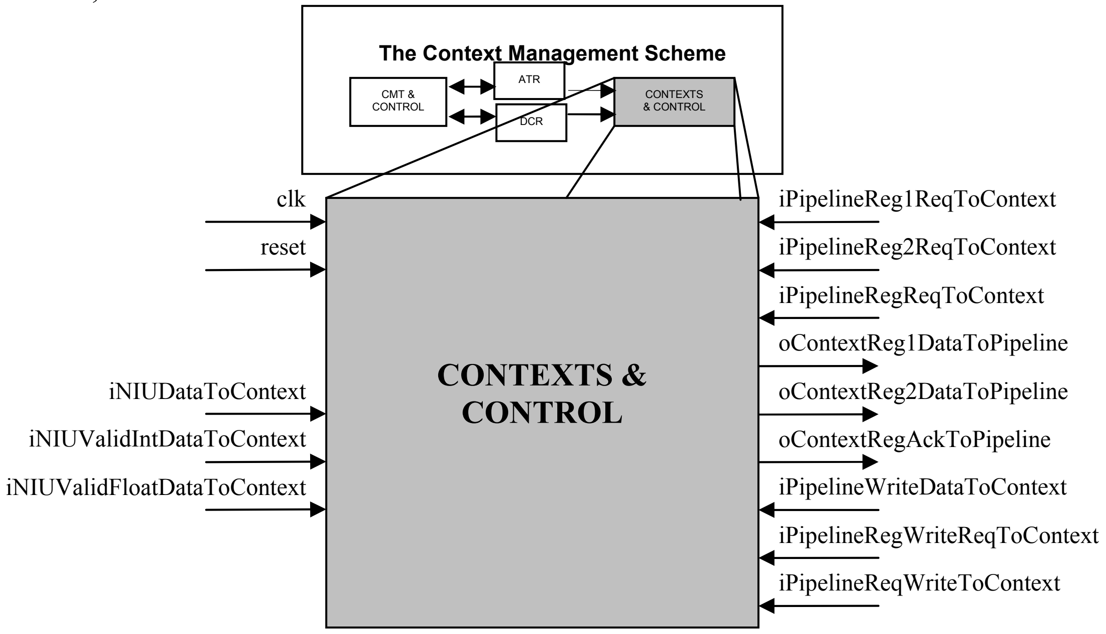


**Figure 4.6**: The registers of a node are organized in a matrix to allow single cycle accesses.

Each row represents the registers in a context. Each column is a register. To read or write a register, two values are needed: (1) a context ID - from the ATR, the DCR, or the NIU - and (2) a register ID. Using this implementation, there is no delay required in the pipeline to access the registers of any context. Figure 4.7 shows the interfaces to the register file. An access from the ATR or DCR comes from the pipeline, so only the pipeline and NIU can write to the registers in a node. This implementation takes advantage of the fact that register references are made with locality to the registers in the rows indexed by the ATR, DCR, and possibly the NIU.

A vertical register file was also considered when implementing the context register file. This file contains 512 (16 contexts * 32 registers per context) general purpose registers. The ATR or DCR entry serves as a pointer to access the base location for a context's register block. The incoming register identifier is used as an index from the base location. This method resembles a paging system in that access to a register requires a base address and an offset. For example, if the ATR refers to context 3, and register 12 is requested, the following operations would occur. The ATR value is multiplied by 32 to obtain the base address (register 0 of context 3). Then, an offset of 12 is added to base address to complete the register reference. These operations can be summarized in the equation: ATR<<5+regID. When compared to the multiplexers used in the 2-D implementation, however, these calculations could increase the critical path to retrieve a register value. Therefore, this method was discarded.



The Context Management Scheme
CMT & CONTROL
ATR
DCR
CONTEXTS & CONTROL
clk
reset
iNIUDataToContext
iNIUValidIntDataToContext
iNIUValidFloatDataToContext
CONTEXTS & CONTROL
iPipelineReg1ReqToContext
iPipelineReg2ReqToContext
iPipelineRegReqToContext
oContextReg1DataToPipeline
oContextReg2DataToPipeline
oContextRegAckToPipeline
iPipelineWriteDataToContext
iPipelineRegWriteReqToContext
iPipelineReqWriteToContext

**Figure 4.7**: A block diagram of the Contexts and Control section of the context management scheme.

| Signal Name | Type | Bit Width | Description |
|---|---|---|---|
| clk | In | 1 | Rising Edge active clock |
| reset | In | 1 | Asynchronous reset to put the unit into a known state |

Now that the register organization has been discussed, the methodology of accessing the registers is presented. This block is sensitive to the positive edge of the clock. Therefore, unless otherwise stated, all references to action occur on the rising edge. As mentioned earlier, the contexts are accessible to the pipeline and the NIU.

**Pipeline Accesses to the Contexts**

The interface between the pipeline and the contexts is through two read ports and one write port. All three ports are necessary in order to read two values and write one in one clock cycle. Table 4.4 below describes the interface between the two units.

**Table 4.4**: Description of the interface between the pipeline and the contexts.

| Signal Name | Type | Bit Width | Description |
|---|---|---|---|
| iPipelineReg1ReqToContext | In | 6 | A reference to a register in the currently executing context. It is 6 bits because there are up to 64 registers to choose from. |
| iPipelineReg2ReqToContext | In | 6 | A reference to a register in the currently executing context. It is 6 bits because there are up to 64 registers to choose from. |
| iPipelineRegReqToContext | In | 1 | When set to a '1', a request is made for registers with column numbers Reg1Req and Reg2Req. '0' otherwise. |
| oContextReg1DataToPipeline | Out | 32 | The register value read from register Reg1Req. |
| oContextReg2DataToPipeline | Out | 32 | The register value read from register Reg2Req. |
| oContextRegAckToPipeline | Out | 1 | Set to a '1' when data is valid on ports Reg1Data and Reg2Data. '0' otherwise. |
| iPipelineWriteDataToContext | In | 32 | The data to write to a register of the currently executing context. |
| iPipelineRegWriteToContext | In | 6 | The register reference to write the data to. |
| iPipelineReqWriteToContext | In | 1 | Set to a '1' when the incoming WriteData is valid. '0' otherwise. |

The names of these signals are given based on the conceptual place of origin rather than the actual place. These signals breech the protocol set up in Chapter 3 but arranging the signals this way made more sense. The <iPipelineReg1ReqToContext>, <iPipelineReg2ReqToContext>, and <iPipelineRegReqToContext> signals are taken from the output of the instruction cache. The register references in each instruction are always made in the same bit locations of the instruction words. Therefore, the register references are split off from the instruction word to go to both the pipeline and the contexts. The acknowledge signal from the instruction cache is used as the register request. Therefore, when a valid instruction word is sent to the pipeline, a request is made for registers from the contexts. The other signals in this interface use the proper naming convention.

Once a request is made, the data values requested are valid before the falling edge of the current clock cycle. This speed is necessary, so the pipeline can forward the register data to the appropriate unit. Therefore, the delay of this unit may be the critical path for half a clock cycle of work.

A data value is written back to a register using the signals with the term 'Write' in them. The data, <iPipelineWriteDataToContext>, is written to register <iPipelineRegWriteToContext> when <iPipelineReqWriteToContext> is set to a '1'.

A potential design problem in this unit occurs when the register being written to is also being read. However, this case was simple to fix. If a value is being written to a register that was being requested, the new value is forwarded to the pipeline.

**NIU Accesses to the Contexts**

The NIU cannot successfully access the context register block whenever it wants. First, the NIU has to receive a positive acknowledge that a context has been allocated for its use. As described above, the context register block needs two values to be accessed: a context identifier and a register number. If no allocation is made, then all of these data values could be lost because no context ID can be given.

When a context is allocated for use by the NIU, the ID for that context is stored in a temporary register. In addition, a 5 bit counter, which is used as the next register number to write, is reset to 0. Table 4.5 below describes the interface used by the NIU to access the contexts.

**Table 4.5**: Description of the interface between the NIU and the contexts.

| Signal Name | Type | Bit Width | Description |
|---|---|---|---|
| iNIUDataToContext | In | 32 | The data to be written to a register. |
| iNIUValidIntDataToContext | In | 1 | Set to a '1' to indicate valid integer data is present on the data bus. '0' otherwise. |
| iNIUValidFloatDataToContext | In | 1 | Set to a '1' to indicate valid floating point data is present on the data bus (See Chapter 5). '0' otherwise. |

The active high <iNIUValidIntDataToContext> signal is used to indicate the data bus contains a value to be written to the next register. The counter is incremented to the next register, so the next data value will be written to a different location. It is important to note that only 32 data values can be sent in a thread message, since the values are written directly to the registers of one context from the network.

Once the NIU to context interface was finalized, the design and implementation of the Context Management Table (CMT) and the Context Register File (CRF) were complete for the basic SCMP node. Together, the CMT and CRF control units are capable of managing multiple threads and accessing the registers associated with each thread. This chapter identified the signal interfaces between the contexts and the memory controller, the pipeline, and the NIU. The strategies for context management to locate the next thread to execute or the next available context for data storage were also presented and explained. The alternative design approaches that were considered as part of the context interface development were also presented and discussed, and the reasons for including or discarding these alternative designs were explained. With the completion of the design and implementation of the basic the SCMP node, it is now capable of executing any fixed point application. The next chapter presents some changes to this basic node design.

# Chapter 5 Additions to the Basic SCMP Node

After the basic node had been designed and implemented, its operational shortcomings were brought to the forefront. For example, there was no construct in place to handle a divide by zero. Also, any thread could write to the CMT and stop another thread without reason. These examples lead to the addition of an exception handler.

Another inadequacy of the basic SCMP node was identified to be that floating point operations were not supported. For example, image rendering algorithms rely heavily on floating point calculations. Since one of the main targets of the SCMP parallel computer is image processing, the node needed to support functions necessary for that type of computation.

This chapter presents the implementation of floating point support and exception handling for the SCMP node as a part of this research.

## *5.1 Floating Point Support*

A floating point unit was added to each SCMP node so that the processor could support a wider variety of applications. The design conformed to IEEE Standard 754-1985 for binary floating point arithmetic. It is important to note that the implementation described in this section has not been thoroughly tested, and thus, has not been verified.

One of the first considerations when adding the float point capability to the SCMP node was how the values would be stored. Single precision values (32 bits) map directly to the current hardware design. For example, a register could store the entire floating point value, since the registers were also 32 bits. However, double precision values (64 bits) required additional design effort, since each double required two registers for storage. Thus, each context could only store up to 16 double precision values, since there were 32 registers in each context. To keep the design changes and implementation as simple as possible, a maximum of 16 floating point values (singles and doubles) could be stored in one context, registers labeled f0 – f15. In addition, the buses that carry register data from/to the contexts to/from the pipeline were expanded to 64 bits, so that two doubles could be retrieved and one written in one clock cycle.

Rather than designing a system that could intermix integers and floating point values, a second context was allocated to a thread to store floating point values. The SCMP system already had the capability to use two contexts in one thread via the DCR register. Any reference to f0 - f15 in a program would require a valid DCR value so a second context could be accessed. This arrangement allowed the user to use the SCMP processor for integer calculations, and extend a thread's capabilities to floating point arithmetic when needed.

The second context could be allocated to a thread in one of three ways. First, the user can specifically allocate a context for floating point operations by using the alloc instruction. The new context identifier is returned to the pipeline and is written to the DCR for the currently executing thread. If a free context is not available, the context will suspend, and the next thread scheduled to execute will gain control of the processor. When the suspended thread executes again, the alloc instruction will be executed again. The process will repeat until a context is successfully allocated to the thread.

Second, an exception occurs to allocate a floating point context if a reference is made to f0 – f15, but the DCR value is invalid. The exception handler will allocate a context, if one was free. Similar to the first case, if a free context is not available, then the exception handler will exit, and the next thread scheduled to execute will gain control of the processor. When the suspended thread executes again, the exception routine will be triggered to run again. This process will continue until a context is successfully allocated to the thread. However, even after the exception handler succeeds the next thread to execute still gains control of the processor because of how exceptions are implemented (see Chapter 5.2).

Third, if a context is going to send a thread message with floating point data in it, the header of the message indicates that the receiving node's NIU must request two contexts from the CMT, one for integer data and one for floating point data. See Figure 5.1 for a description of the modified message format. If the contexts cannot be allocated, the NIU must continue to request both contexts until it succeeds. If both contexts are allocated, then a bit, which was added to each flit, indicates if the data will be stored in a register as integer data or floating point data.

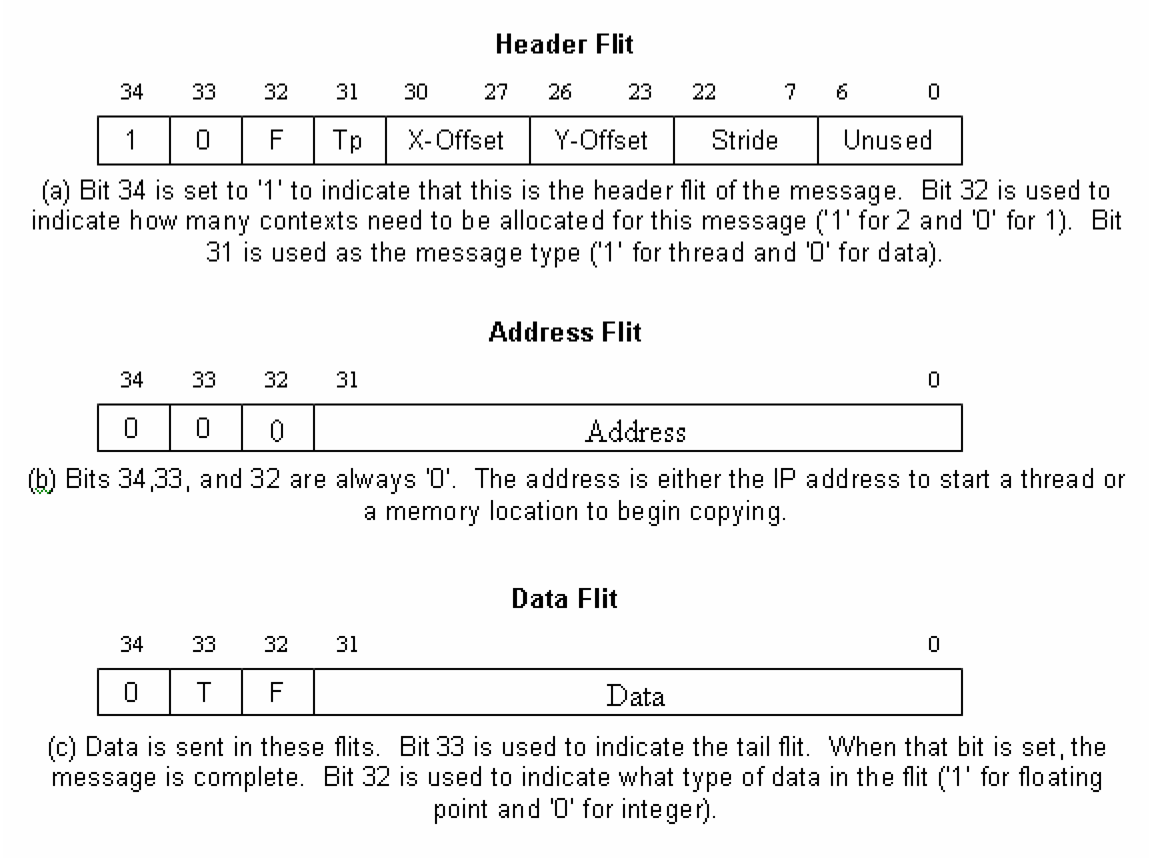

Header Flit
34 33 32 31 30 27 26 23 22 7 6 0
1 0 F Tp X-Offset Y-Offset Stride Unused
(a) Bit 34 is set to '1' to indicate that this is the header flit of the message. Bit 32 is used to indicate how many contexts need to be allocated for this message ('1' for 2 and '0' for 1). Bit 31 is used as the message type ('1' for thread and '0' for data).
Address Flit
34 33 32 31 0
0 0 0 Address
(b) Bits 34,33, and 32 are always '0'. The address is either the IP address to start a thread or a memory location to begin copying.
Data Flit
34 33 32 31 0
0 T F Data
(c) Data is sent in these flits. Bit 33 is used to indicate the tail flit. When that bit is set, the message is complete. Bit 32 is used to indicate what type of data in the flit ('1' for floating point and '0' for integer).

**Figure 5.1**: A bit was added to the flits to indicate whether or not floating point data was being used.

Now, when a thread message arrives at the NIU of a node, the ‘F’ bit in the header flit is used to determine whether or not floating point data is present in this message. If not, then the ‘F’ bit is ‘0’, and would be ignored for the remainder of the message. When the ‘F’ bit is a ‘1’, a second context needs to be allocated for floating point data storage. This case is the third one described above. An active high signal, called <NIUReqNewFloatContextToCMT>, was added between the NIU and the CMT to accommodate the request of a second context. Because a float context cannot be requested without an integer context, the CMT control assumes this signal is valid only when the <NIUReqNewIntContextToCMT> signal is a ‘1’. Therefore, the case when ReqNewIntContext is ‘0’ and ReqNewFloatContext is ‘1’ is ignored. The CMT control uses two registers to store the context identifiers of the integer context and the floating point context so each can have its registers written to with the flits that followed the header.

Once two contexts have been allocated for the message, the <NIUValidFloatDataToContext> and <NIUValidIntDataToContext> signals are used between the NIU and the contexts to determine which context the incoming data will be stored. The values of these signals are determined by the ‘F’ bit in the incoming data flit. If the ValidIntData signal is high and the ValidFloatData signal is low, then the data is an integer. Conversely, if ValidIntData is low and ValidFloatData is high, then the data is stored as a float. If both signals are high or both low, then the data between the NIU and the contexts is invalid. When the tail flit is received, the integer context is scheduled for execution and its DCR value is set to the identifier of the floating point context. This feature was added to help support floating point, but it also provides a way for a node to send a thread message with 64 data values rather that just 32.

It is important to note that the ‘F’ bit has no meaning in a data message. A data message contains a number of 32 bit values that are written to memory starting at a specified location. The type of data does not matter until the values are read by the pipeline. In other words, the memory unit does not care about the type of data, just that bits are being written to a location.

Since the NIU can request two contexts at once, the next two free contexts need to be known. The same logic is used to find a free context, but a test is done to see if another free context needs to be found. For example, when a node is reset, no free context is known. On the next clock edge, the first free context is found. On the second edge, the second free context is found. On the third clock cycle, there is no need for another free context to be found since two were already discovered. Then, at some cycle in the future, one of these free contexts will be allocated. On the next clock edge, the second free context will become the first. Then, in the next cycle, a second free context is found, if one is available. This additional logic also allows the pipeline to allocate a context and the NIU to request an integer context at the same time. Being able to service both units at once potentially decreases the amount of delay in the network since the NIU is always able to request one context, if one is available.

The other form of required storage is local memory. Again, support for double precision values caused a change in the hardware. Since the memory is used just for simulation, it was modified to output two 32 bit values at once. This change also meant that the buses between the pipeline and the memory controller and between the memory and the memory controller were expanded to 64 bits. In addition, the <MemoryOpToMemory> signal was modified to include an 8 byte operation to accommodate floating point accesses.

Finally, a floating point arithmetic unit (FPU) was added to the node. The FPU performs the operations set forth in the IEEE standard. These operations have yet to be optimized for hardware and are assumed to execute in one cycle. The FPU was implemented like the ALU, with the same interface except 64 bits wide instead of 32, so its block will not be discussed in detail.

The addition of floating point support makes each SCMP more versatile. Thus, each node is capable of executing more complex algorithms, and hopefully, in less time than on a uniprocessor.

## 5.2 Exception Handling

When an exception occurs, a software routine is required to manage the error. The way SCMP executes software, either a dedicated thread context or user provided software is necessary to handle the exception. It is unreasonable for everyone who writes a program for SCMP, to write all of the exception handlers in addition to the program itself. Therefore, a software library routine to handle each exception was written by Dr. James Baker. A list of exceptions is given in Table 5.1.

**Table 5.1**: Exceptions handled in the SCMP node.

| Exception Description | When It Occurs | ESIGNAL values |
|---|---|---|
| No Exception | | 0x00000000 |
| Integer Overflow | If result is > INT_MAX<br>If result is < INT_MIN<br>If INT_MIN/-1<br>Or Negate INT_MIN | 0x00000001 |
| Integer Divide By 0 | If second source operand is 0 | 0x00000002 |
| Floating Point Underflow | CA:QA, 3rd edition, Appendix H, pp. H-36 - H-37 | 0x00000004 |
| Floating Point Overflow | For single-precision, exponent is 128 or greater<br>For double-precision, exponent is 1024 or greater | 0x00000008 |
| Floating Point Divide By 0 | If second source operand is 0 | 0x00000010 |
| Floating Point Inexact | Occurs if result overflows or must be rounded | 0x00000020 |
| Floating Point Invalid | Result does not have a natural representation as a number or +/- ∞<br>sqrt(-1), 0/0, ∞ - ∞, etc. | 0x00000040 |
| Bus Alignment | Occurs when memory accesses are not on an even boundary | 0x00000080 |
| Bus Invalid | Load or store to invalid address: Valid | 0x00000100 |

| | | |
|---|---|---|
| | addresses are 0x00000000 to MEM_SIZE-1 for RAM, 0xffffff00 to 0xffffffff for CMT<br>Or Instruction fetch from a non-word-aligned address | |
| Opcode Invalid | The instruction opcode does not exist | 0x00000200 |
| No Free Context | All entries in the CMT have the ALLOC bit set to 1, and the NIU requests a free context | 0x00000400 |
| Invalid Context | Attempt to access a context that was not allocated | 0x00000800 |
| No Data Context (DCR Invalid) | Reference to R32-R63 is made before the DCR is valid | 0x00001000 |

When an exception occurs, the pipeline is flushed so that no operation occurs after the error. This stall is necessary since the exception condition is not known a priori. Since the pipeline is flushed, the exception code gains control of the processor, just as a thread context would. Also, the IP address of the routine and the context identifier that caused the exception must be known. If one entry in the CMT is dedicated to the exception handler, it would be a thread context; and the IP address of the routine could be stored in the IP field of the CMT entry. Therefore, the exception handler is given a context and is always resident for use. Context 0 was chosen arbitrarily.

The method of scheduling the error handling routine was considered. Either the exception handling routine could run as usual, in a round robin fashion, or it could execute out of order. If context scheduling continued in round robin fashion, a second context could throw an exception before the exception handler was executed. The main question becomes, which exception would be handled? If it was the first exception, then the second context would have to recognize that an exception was already pending, and suspend rather than throwing the exception before suspending. While this solution is not complex, it could keep other threads from executing. For example, assume the thrown exception ended thread execution (e.g. divide by 0) and there were no free contexts. If the exception was handled in round robin fashion, a thread message would wait in the network until the exception was handled since there was not a free context. However, if the exception was handled immediately, this delay would not occur.

Conversely, if context 0 executed out of order, then whenever an exception is raised, it would be handled. The trick is not to modify the value in the next thread to execute register (NTE). When the exception handler finishes its execution, the next context that received control of the processor is the context pointed to by the next thread to execute register (NTE). This implementation allows execution to continue in round robin fashion and ensures that no context is starved. A potential negative of this implementation is that a thread will execute until it completed or raised an exception, at which time the thread gives up the processor until the next time it is scheduled. This scenario only involves a check when a context suspends, to determine if an exception needed to be handled or not. If so, then context 0 is executed. If not, then the next thread to execute gains control of the processor.

There was one major concern about adding exceptions to the SCMP node. In general, processors on a parallel system work towards a common goal. The processors in a parallel system communicate, either to share results or synchronize with another node, to achieve that goal. If a synchronization thread threw an exception and was killed by the handler, then the waiting processor will hang indefinitely and the program would never finish. In the SCMP simulator, an error message was output when a thread was killed by an exception. Then, the simulator exited. In hardware, the exception handler could reset the entire processor, since the algorithm failed to complete.

As stated above, context 0 is used as the exception handling context. The following registers are used to identify when an exception is thrown, by which context, and if handling is necessary: EHANDLER, EMASK, ESIGNAL, ESTATUS, and ETHREAD. All of the registers and logic needed to support error handling were added to the context management logic of the processor, since that is where threads are created, ended, suspended, and scheduled. The values of the registers can be read by using the readsr instruction and written, if possible, using the writesr instruction.

EHANDLER contains the starting address of the exception handler. When an exception occurs, this value is copied into the IP field of CMT entry 0 (the exception handling context). This strategy allows the user to write an exception handler and use it by writing to this register. The default value of this register can be set in contexts.h

The EMASK register is used to determine which exceptions would cause the handler to execute. A zero bit value means the exception is disabled, while a one means it is enabled. The organization of the register is shown in Figure 5.2.

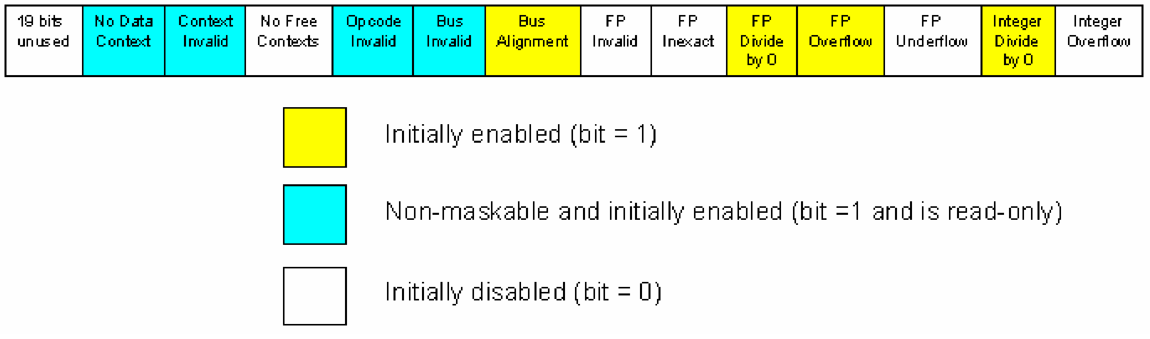


**Figure 5.2**: The bit assignment in the EMASK register.

ESIGNAL is a read-only register used to determine, if any, the most recent exception that was thrown. Table 5.1 includes the ESIGNAL value for each exception. The encoding for this register is a one to one match with the EMASK register. This uniform management is more readable and faster to debug since register comparison is not necessary.

The ESTATUS register contains all of the exceptions that have been thrown, whether enabled or disabled. The bits in this register are 'sticky' and, as before, match the positions in the EMASK register. Sticky bits remain at the current value until cleared using the writesr instruction. As a general rule, the exception handler will clear the corresponding bit in this register when it completes. However, if an exception is disabled, once thrown, the value in the ESTATUS register will remain a one. The user can use the readsr instruction to obtain the status of the exceptions for the node. This read could be useful if the user desires to handle the exception within the program without giving up the processor.

ETHREAD is a read-only register that identifies which context caused the exception. In addition, this value is used to read/write register values to the correct context during the exception handler. In other words, this register acts like the DCR register for another context. It could also be used to access the CMT entry of the context that caused the exception and use/change its values. For example, assume that context 1 caused a divide by 0 exception, and CMT entry 1 contains context 10 in its DCR field. Then, the exception handler needs to free both context 1 and context 10 since neither could continue execution.

The following describes how the new hardware will handle an exception. When an exception occurs, the corresponding bit is set in the ESTATUS register. If the corresponding bit in the EMASK register is a zero, then the exception is disabled and thread execution continues uninterrupted. If the exception is enabled in EMASK, then throw the exception. In CMT entry 0, the THREAD bit is set to '1', EHANDLER is copied to the IP field, and the DCR is set to 0. The current ATR value is copied into ETHREAD, and ESIGNAL is set to the exception identifier. The current thread suspends and context 0 begins to execute. After the exception is handled, the corresponding bit in the ESTATUS register is cleared, and the next thread to execute after the thread that caused the exception is given control of the processor.

In addition to the hardware changes necessary to handle the exception, extra hardware is necessary to identify when an exception occurs. For example, the memory controller is responsible for identifying bus alignment errors to memory. Therefore, the signal MemoryExceptionToPipeline was added between the memory controller and the pipeline. This signal is as wide as the ESIGNAL register, and the signal value corresponds to the appropriate exception. Similar signals were added as outputs for all blocks of the processor that could cause exceptions: ALU, memory controller, and pipeline. The other units indicate an exception through a negative acknowledge. For example, assume that the alloc instruction is executed but no free context is available. The CMT control would indicate an exception of 'no free context' by using a negative acknowledge. The pipeline recognizes the negative acknowledge and throws the 'no free context' exception.

The pipeline receives these signals after each instruction is executed. If an exception occurs, the corresponding ESIGNAL value is sent to the context management control so the exception can be thrown.

The major problem that arose with the addition of exceptions to the design was if a write to memory occurred the instruction after an exception was thrown. However, the SCMP pipeline is able to catch each exception before the write to memory began. This new requirement is achievable since all ALU operations end on the falling edge of the execute stage of the pipeline. There is a half cycle to stop memory from being written. Figure 5.3 depicts the timing diagram to illustrate this point. For bus alignment errors, the memory controller replies in the same cycle since it is asynchronous. Therefore, the pipeline detects the error before the setup for the next instruction is complete.

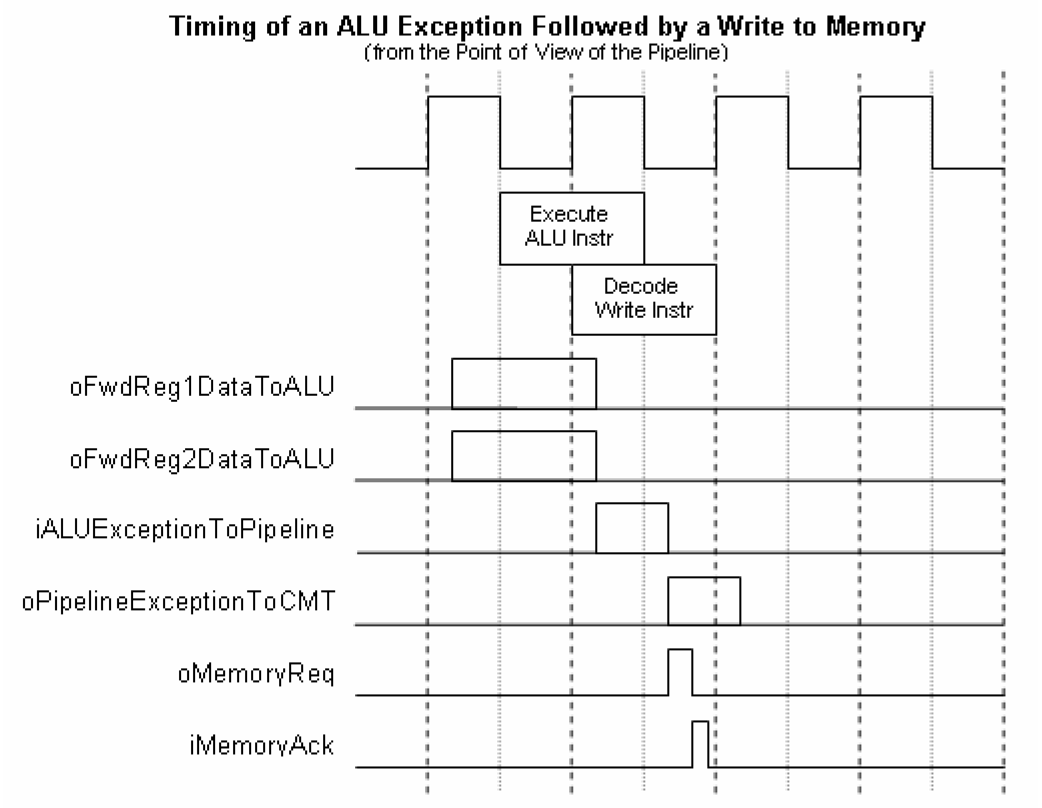


**Figure 5.3**: The timing of an exception between the ALU and the pipeline.

Adding exceptions to each node creates a safer environment for programs to execute. The structure that was put in place creates a context priority scheme where context 0 executes whenever it needs to and the other contexts execute in a round robin non-interruptible fashion.

With the addition of floating point support and exception handling, the SCMP node can support any target application as well as any application error. The floating point design allows for utilization of single precision and double precision floating point values. Extending the node hardware capabilities to accommodate floating point values had the extra benefit of doubling the amount of data that a node could pass in a thread message to 64 values. Now that the SCMP node design and implementation is complete, its operation needs to be tested and verified.

# Chapter 6 Software Support and Simulation

Now that the hardware was designed and behaviorally implemented in System C, it needed to be tested to ensure that all the parts work separately and together. This research effort was responsible for all of the testing and verification presented in this chapter. Once the design was verified, the software simulator for the SCMP parallel computer could be updated to match the hardware design.

## *6.1 Hardware Simulation of the Block of the SCMP Node*

Each block of the SCMP node was tested individually and then as an integrated part of the node. All of the blocks were tested in a similar fashion. A new object was created for each block that instantiated the unit it was testing and was triggered on the appropriate clock edge. Figure 6.1 gives a general block diagram of this testing scenario.

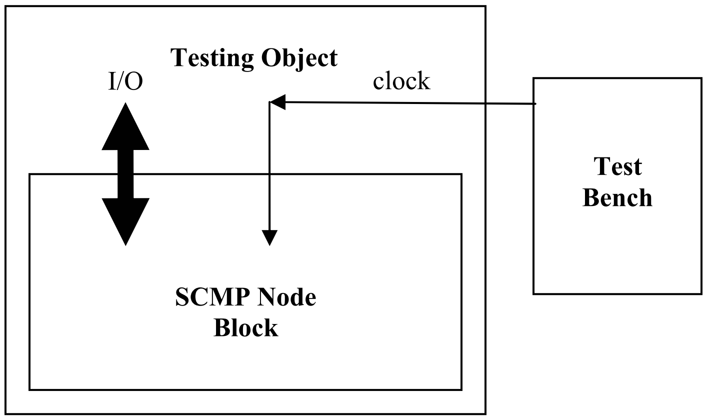


**Figure 6.1**: A general case of how individual blocks were tested.

The testing object was responsible for controlling the test cases. This control was accomplished by a finite state machine, which manipulated the inputs and displayed the appropriate valid outputs. The testing unit also created a trace of the signals of a block.

A test bench was created for each unit. The test benches created the testing object and then controlled the clock so the test could proceed. After the simulation was complete, the output from the tester and the signal traces were used to verify the design of a unit in the node.

Let us consider the testing object for the ALU as an example. When the testing unit, called aluTEST, was instantiated, a reset object and an ALU object were created. The positive edge triggered reset unit controlled the reset signal. The ALU, however, was negative edge triggered because of how the pipeline performed its operations. On the reset object's first event, the reset signal was set high. The opcodes that signaled the ALU to perform an operation were arranged consecutively in the instruction set. Therefore, when the testing object was reset, the OpSelect lines were set to the first value that signaled an ALU operation. The two inputs to the ALU were set to the same value for the entire test since the outputs of the operation could easily be calculated. On the next reset event, the reset signal was set low, so the test could begin.

The aluTEST object waited for the acknowledge signal from the ALU to go high. If, on an ALU event, the acknowledge signal was low, a NOP (0x00) was sent to the ALU because it did not pipeline the operations. The acknowledge event indicated that the ALU had finished its calculation and the output was valid. When a valid output signal was received, the ALU value was output, and the next opcode was sent on the OpSelect lines. The process repeated until all ALU operations were tested. This operation can be seen in Figure 6.2, which depicts a portion of the results of the test on the ALU.

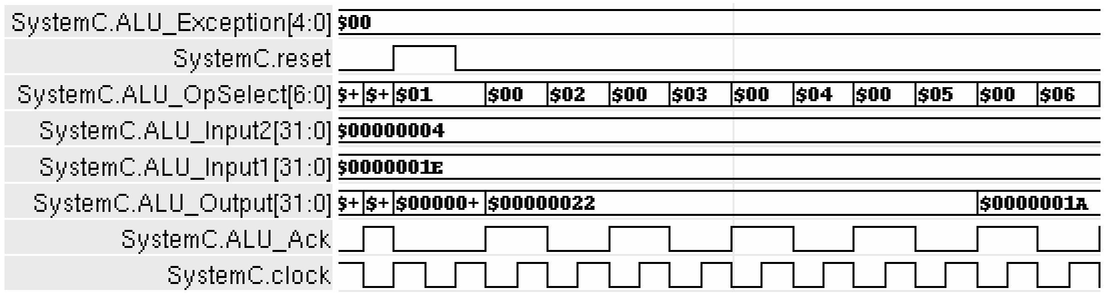


**Figure 6.2**: A timing diagram from the ALU test bench.

In Figure 6.2, when the reset signal was set high, the ALU inputs were set to 4 and 30, and the OpSelect lines were set to the add instruction (0x01). On the falling clock edge after the reset signal went low, the ALU performed the addition of 4 and 30. Because the acknowledge signal was low on the same edge, a NOP (0x00) was sent on the OpSelect lines. The first time the acknowledge signal was high on a falling clock edge, the result was 34 (0x22) signifying success. After the result was read, the next ALU operation to perform was sent out on the OpSelect bus. If all the ALU operations were completed, then the test was complete and the simulation exited.

The asynchronous memory controller was the only block that was not tested individually. After the memory unit and the context management table had their memory interfaces tested, both were used to test the memory controller. These extra blocks simplified the testing unit since the memory and CMT became responsible for the acknowledge signals and data handling. The testing block was only accountable for the interfaces to the requesting units. There were six test cases for access to the memory: when zero units requested access, when only the pipeline requested access, when only the instruction cache requested access, when only the NIU requested access, when all three units requested access, and when the instruction cache and NIU requested access. The test when all three blocks requested access to memory also covered the missing two cases: when the pipeline and the instruction cache requested access, and when the pipeline and the NIU requested access. This deduction was made because it was shown that the pipeline received priority over both of the other units when all three units made a request to memory. Figure 6.3 illustrates part of the test of the memory controller.

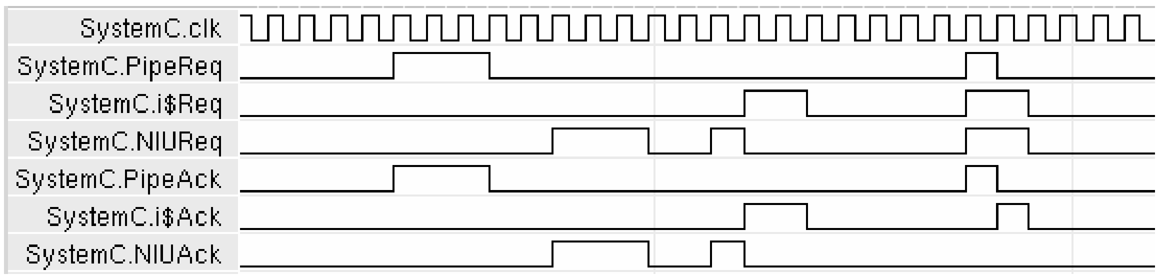


**Figure 6.3**: A timing diagram of the test for the memory controller.

In the diagram above, the request and acknowledge signals for each of the three units that access the memory through the memory controller are shown. The six test cases that were described above can be seen in this figure. As expected, when only one unit requested access to the memory, it received a positive acknowledgement. The cases when more than one unit had a memory request are shown at the right end of the image. Again, as expected, when all three units requested memory access, the pipeline received the positive acknowledge. Similarly, when the instruction cache and the NIU requested access to memory, the cache received the positive acknowledgement.

Of the blocks that make up the SCMP node, only the pipeline was not fully tested. The remaining blocks of the node were tested and their operation was verified. Within the pipeline, the interfaces from the pipeline to the CMT/contexts and from the pipeline to the NIU still need to be implemented and tested. Once this work is completed, the pipeline can be completely analyzed.

## 6.2 Results

After each of the blocks of the SCMP node were tested, they were put together to simulate a single node of the SCMP parallel computer. This simulation excluded the router, as it was not within the scope of this portion of the research project. Also, these tests were conducted without the implementation of floating point or exception handling.

It was not possible to simulate a node's operation without executing a program on the node. Therefore, it was important to consider how a program was created for the SCMP parallel computer. A programmer could write a program for this architecture either using C or assembly. The C-compiler was developed by Sidney Bennett [24] and Dr. James Baker. It was based on the SUIF toolkit developed at Stanford [22].

The C program is first translated into a general assembly language provided by SUIF. Each of these general assembly instructions was implemented in the SCMP instruction set. Then, the SUIF code is translated into SCMP instructions. Next, the SCMP assembler was used to generate the actual machine code for a node. Finally, library code, which provided general functionality, could be linked into the program using the SCMP linker. This explanation is vastly oversimplified, but gives the reader a general idea of how it works.

Once the executable had been generated, it was necessary to test the program. A software simulator was developed to execute SCMP programs. It was written in C, by Dr. James Baker, but it was not cycle accurate. Thus, the operations were performed correctly, but they did not use the correct amount of clock cycles.

Similar to a single unit, a testing object was created that instantiated the entire node with its interconnections. To simulate a program on a node, the tester loaded an executable into the memory on the node. An executable program for the SCMP chip was in the ELF file format. Dr. James Baker wrote a C converter that took an executable file as an input and extracted the bytes that need to be stored in memory. These bytes were then written to the memory of a node. Also, the start address for the program was saved so that a context could be created on the node to start the execution. Once the memory contained the necessary information, the testing object, then, wrote to the Alloc bit, the Thread bit, and the IP address of an entry in the CMT using the memory mapped interface of the CMT. The IP address came from the saved start address indicated by the executable file. The tester could accomplish these tasks because it instantiated all of the units in the node, and thus could control any interface it wanted. After the initial thread was created in the CMT, the testing object waited for the simulation to complete.

The simulation was complete after all of the contexts in the node had finished execution. This case was true when every entry in the CMT had its Alloc bit set to ‘0’. A done signal was added between the CMT and the node tester to indicate when all of the contexts were free. The tester used this signal to indicate the end of the simulation. However, this scenario will not work when multiple nodes were simulated at once. All of the nodes could have completed execution, but there may be a thread message in the network that could start a new thread on one of the nodes. Therefore, when the entire SCMP parallel computer is simulated in hardware, a mechanism will be needed to detect if there are any messages in the network.

Once the program finished its execution, the operation of the node needed to be verified. Similar to the individual unit testing, the simulation generated a trace file of the appropriate signals so the timing of the units could be analyzed. In addition, a function was added to the CMT/context control and the memory that allowed its contents to be output to a monitor screen. The CMT/context output function accepted a single integer, which was used as an index to the CMT and the contexts. The CMT value at the given location was output, and all of the register values for the given context were output.

When considering the memory unit output function, it was not reasonable to output the entire contents of memory. Therefore, two values were sent to the memory output function. These parameters served as the start and end addresses of the values to output from memory. Using these additional functions and the signal trace, the operation of the SCMP node was verified.

Currently, the pipeline only supports instructions that involve the ALU, instruction cache, and memory. The test program that was executed in the SCMP node hardware simulator and a portion of its results are given in Table 6.1 below. For an assembly instruction reference guide, see Appendix A.

**Table 6.1**: The program used to test the SCMP node with a portion of its results. (This program was created and tested by Priyadarshini Ramachandran.)

| Test Code | Result |
|---|---|
| main: | <PIPE.ID1> Decoding: 0x6e180005 (LLO) |
| llo r3, 5 | <ALU> LLO: in1 = 0 in2 = 5, Result:5 |
| llo r2, 4 | <PIPE.ID1> Decoding: 0x6e100004 (LLO) |
| stw 1(r2), r3 | <ALU> LLO: in1 = 0 in2 = 4, Result:4 |
| ldw r4, 1(r2) | <PIPE.ID1> Decoding: 0x7e004181 (STW) |
| mul r1, r4, r3 | <ALU> STW: in1 = 4 in2 = 1, Result:5 |
| llo r5, 9 | <PIPE.ID1> Decoding: 0x78204001 (LDW) |
| llo r6, 8 | <ALU> LDW: in1 = 4 in2 = 1, Result:5 |
| addi r7, r5, 89 | <PIPE.ID1> Decoding: 0x12088180 (MUL) |
| idiv r8, r6, r2 | <PIPE.ID1> Is a RAW Hazard |
| ldw r9, 1(r2) | <PIPE.ID1> Decoding: 0x12088180 (MUL) |
| stw 1(r2), r1 | <ALU> MUL: |
| ldw r5, 1(r2) | <ALU> MUL: |
| idiv r6, r5, r9 | <ALU> MUL: |
| bra Skip1 | <ALU> MUL: |
| mulh r7, r1, r2 | <ALU> MUL: in1 = 5 in2 = 5, Result:25 |
| mulu r8, r3, r5 | <PIPE.ID1> Decoding: 0x6e280009 (LLO) |
| mului r9, r7, 1 | <ALU> LLO: in1 = 0 in2 = 9, Result:9 |
| Skip1: | <PIPE.ID1> Decoding: 0x6e300008 (LLO) |
| sub r1, r3, r2 | <ALU> LLO: in1 = 0 in2 = 8, Result:8 |
| bgt r1, Skip2 | <PIPE.ID1> Decoding: 0x638a059 (ADDI) |
| muli r0, r0, 0 | <ALU> ADDI: in1 = 9 in2 = 89, Result:98 |
| mul r1, r2, r3 | <PIPE.ID1> Decoding: 0x2240c100 (IDIV) |
| Skip2: | <ALU> IDIV: |
| llo r3, 2 | <ALU> IDIV: |
| end | etc. |

The results section of Table 6.1 shows when each instruction was identified and then executed. In the case of the first multiply instruction (MUL), one of the inputs was read from memory in the previous instruction. Therefore, the pipeline must stall for one cycle to wait for the valid data to return from memory. Once the data returned, the multiply instruction was performed. It is assumed that it will take 5 clock cycles to execute a multiply, which is the reason for the blank <ALU> MUL: lines in the output.

The results of these simulations demonstrate that the current SCMP node has been implemented successfully. As the pipeline begins to support the other portions of the processor, test programs will be executed in the simulator to verify the operation of the node. Also, testing for utilization of the floating point features and the exception handling must also be performed to complete the testing and verification of the SCMP node.

# Chapter 7 Summary and Future Work

The goal of this research was to create a hardware model of a node for the SCMP parallel computer. This hardware model will be used to modify the existing software simulator to become cycle accurate.

## *7.1 Summary*

An SCMP node was designed and implemented as a 32 bit RISC, integer based, multi-threaded processor with local memory. Up to sixteen threads could be simultaneously scheduled to execute on a node. These threads were scheduled in round robin, non-preemptive fashion. The node was designed to communicate with other nodes on the SCMP chip. To meet this requirement, an active message passing system was used. Two types of messages, thread and data, were supported. A thread message was sent to another node on the chip to start a new thread on that node. Data messages were handled much like DMA transfers. The data was read from one node's memory and written directly to another node's memory. These requirements were provided prior to the beginning of this thesis research.

After the prerequisites for the node had been identified, this research began by partitioning the node into a number of hardware units. The design and behavioral implementation, in System C, of each of the basic units was then created.

The five stage pipeline was modeled after the MIPS32 [23]. Three of the components of the node - the ALU, the memory, and the instruction cache – were created for simulation only, and the design of each was common to those found in many other processors. The memory controller was an asynchronous unit responsible for arbitrating requests to memory. Priorities were given to the units that could access memory so that the pipeline would stall as few cycles as possible. Therefore, priority was given in the following order: the pipeline, the instruction cache, and the NIU. As a possible point of improvement, the memory may not need to know how many bytes of data are being operated on. The memory controller could read out the maximum number of bytes and modify the memory value to the appropriate number of bytes. The NIU was responsible for creating messages and injecting them into the network between the nodes on an SCMP chip. It accepted data, either from the pipeline or the memory, generated the flits for the message, and sent them in to the network.

With those units, the SCMP node could run a single thread, if it was given an address to start. Since this processor was multi-threaded, support was added to accomplish this goal. A context was responsible for the execution of one thread. The context management table was created to provide a framework for handling multiple threads. Each thread was given its own set of 32, 32 bit, registers for its use. Special instructions were available to expand to 64 registers, if necessary. Contexts were scheduled for execution in round robin non-preemptive fashion. This algorithm guaranteed that no process would be starved unless one entered an infinite loop.

Once the initial design was implemented, two major features were added to the node: floating point support and exception handling. Floating point for the SCMP node adhered to IEEE Standard 754-1985. The major concern when adding this feature was data storage. When a context manipulated floating point values, it was forced to allocate a data context for those calculations. Since the registers were only 32 bits wide, two of them were used when a double precision value was stored in a context. This register usage meant that a context could only manipulate 16 double precision values at once. As a result, for simplicity sake, a data context could only hold 16 floating point values.

Exception handling is an important feature of any program. For example, what happens when the denominator of a fraction is a zero? It was unreasonable to have everyone that wrote a program for the SCMP parallel computer also write an exception handler. Therefore, a general one was created for the SCMP node. Context 0 was dedicated to handling exceptions. When a context threw an exception, the context would suspend and context 0 would gain control of the processor. Then, the exception was handled, and the next thread to execute was given control of the processor. Therefore, contexts 1 through 15 used round robin scheduling and context 0 only executed when an exception was thrown.

Only the initial design, without floating point and exceptions, was verified. First, each individual unit was tested. Then, an entire node was assembled and a short sample program was executed to test its operation.

## 7.2 Future Work

Although significant progress was made as part of this research effort toward developing a node for the SCMP chip, there is still a considerable amount of work to be done. For example, the processor that was verified was only capable of integer calculations. A design for the addition of floating point operations has been presented, but its implementation and verification have not been completed. Similarly, the exception handling scheme for the node has been designed and implemented, but it needs to be verified.

The hardware model of the SCMP node that was developed should also be used in the future to modify the software simulator to become cycle accurate. The same sample programs used to verify the hardware should be tested in the software simulator. If the results of the two simulators do not match, the software simulator has not yet been correctly modified to match the hardware.

Outside of the information presented in the thesis, there are many additions or changes that can be implemented on the SCMP node to possibly improve performance. These items can include the following: give priorities to threads, allow pre-emptive scheduling, optimize the operations in the ALU, and exploit instruction level parallelism within a thread.

It is conceivable that giving priorities to the threads on a node could speed up its performance. When synchronization between nodes occurs, short thread messages are sent so a semaphore value can be decremented. These messages do not require many instructions, but at least one other thread will be waiting for the synchronization thread to execute. So if this thread is given priority, it would have two advantages. First, the thread waiting for the synchronization could continue uninterrupted. Second, since the thread was short, it could free its context quickly so another thread could use it.

Along the same lines, pre-emptive round robin scheduling would prevent one context from monopolizing the processor. The time slice given to each thread would be an important feature to consider. It should be long enough to allow one synchronization thread to execute from start to finish so the benefits described above can still be achieved.

Another alternative would be to allow simultaneous execution of threads on a node. This change would turn the SCMP node in to something like the SMT architecture [20].

Optimizing the operations of the ALU may help to increase performance. Currently, it is assumed how many cycles each operation will take. Designing each of the ALU's operations would give a definite answer to how many clock cycles each operation will take.

One of the motivations to create the SCMP parallel computer was to exploit thread level parallelism since instruction level parallelism had reached a point of diminishing returns. However, it may be possible to exploit some ILP within a thread without drastically increasing the complexity of an SCMP node.

As a complete design change, it may be worthwhile to go to a 64 bit processor instead of a 32 bit one. This change should be considered if most of the applications being developed for the SCMP chip use floating point operations. Most of the buses in the system expanded to 64 bits with the addition of a floating point unit, so 32 bit operations only make use of half of the provided ports.

After the node becomes a stable unit, the entire SCMP parallel computer should be tested. This testing would be accomplished by including a router in the instantiation of a node. Then, the nodes would be connected through the routers and a program could be tested with hardware simulation.

Another future consideration for SCMP would be to design an I/O scheme for the chip. Currently, it is assumed that everything the chip will need is on chip before the program begins execution. In a real system, however, data will likely need to be loaded onto the chip before program execution.

Finally, in considering a real system, the synthesis of the SCMP parallel computer will become important. A transistor count of the chip could be obtained and be used to verify the projected fabrication year for SCMP. One possible way to create a physical system before that date would be to use a stack of FPGAs. Each FPGA could represent one node, and the pins could be connected to form the network.  Once the chip is fabricated, it can be installed as a system processor or a co-processor.

## References

[1] Semiconductor Industry Association, "The International Technology Roadmap for Semiconductors 2003 Edition," 2003.

[2] V. Agarwal et al., "Clock Rate versus IPC: The End of the Road for Conventional Microprocessors." Proc. 27th Ann. Int'l Symp. Computer Architecture, New York: ACM Press, 2000, pp. 248-259.

[3] D. Matzke, "Will Physical Scalability Sabotage Performance Gains?," *Computer*, vol. 30, no. 9, September 1997, pp. 37-39.

[4] P. Ghosh, R. Mangaser, C. Mark, and K. Rose, "Interconnect-Dominated VLSI Design," *20th Conference on Advanced Research in VLSI (ARVLSI 99)*, March 1999.

[5] W.J. Dally and S. Lacy, "VLSI Architecture: Past, Present, and Future*," 20th Conference on Advanced Research in VLSI (ARVLSI 99)*, March 1999.

[6] K. Diefendorff and P. Dubey, "How Multimedia Workloads Will Change Processor Design," *Computer*, vol. 30, no. 9, September 1997, pp. 43-45.

[7] W.A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, vol. 23, no. 1, March 1995, pp. 20-24.

[8] J. M. Baker et al., "SCMP: A Single-Chip Message Passing Parallel Computer," Proc. Parallel and Distributed Processing Techniques and Applications, PDPTA'02, CSREA Press, 2002, pp. 1485-1491.

[9] D.S. Wills, H.H. Cat, J. Cruz-Rivera, W.S. Lacy, J.M. Baker, Jr., J.C. Eble, A. Lopez-Lagunas, and M. Hopper, "High-Throughput, Low-Memory Applications on the Pica Architecture," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 10, October 1997, pp. 1055-1067.

[10] W. Dally, J. Fiske, J. Keen, R. Lethin, M. Noakes, P. Nuth, R. Davison, and G. Fyler, "The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms," *IEEE Micro*, vol. 12, no. 2, April 1992, pp. 23-39.

[11] W. Dally, "Virtual-Channel Flow Control," *IEEE Transactions on Parallel and Distributed Systems*, vol.3, no. 2, March 1992, pp. 194-205.

[12] Gold, Brian, "Balancing Performance, Area, and Power in an On-Chip Network," M.S. Thesis, Virginia Polytechnic Institute and State University, July 2003.

[13] K. Diefendorff, "Power4 Focuses on Memory Bandwidth," *Microprocessor Report*, vol. 13, no. 13, October 6, 1999.

[14] M. Tremblay, J. Chan, S. Chaudhry, A.W. Conigliaro, and S.S. Tse, "The MAJC Architecture: A Synthesis of Parallelism and Scalability," *IEEE Micro*, vol. 20, no. 6, November-December 2000, pp. 12-25.

[15] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The Case for a Single-Chip Multiprocessor," *Seventh International Symp. Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996, pp. 2-11.

[16] F. Allen, et. al., "Blue Gene: A Vision for Protein Science Using a Petaflop Supercomputer," *IBM Systems Journal*, vol. 40, no. 2, 2001, pp. 310-327.

[17] V. Krishnan and J. Torrellas, "A Chip-Multiprocessor Architecture with Speculative Multithreading," *IEEE Transactions on Computers*, vol. 48, no. 9, September 1999, pp. 866-880.

[18] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring It All to Software: Raw Machines," *Computer*, vol. 30, no. 9, September 1997, pp. 86-93.

[19] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent RAM," *IEEE Micro*, vol. 17, no. 2, March/April 1997, pp. 34-44.

[20] S. Eggers, J. Elmer, H. Levy. J. Lo, R. Stamm, and D. Tullsen, "Simultaneous Multithreading: A Platform for Next-Generation Processors," *IEEE Micro*, vol. 17, no. 5, September/October 1997, pp. 12-19.

[21] Mahajan, R.; Govindarajulu, R.; Armstrong, J.R.; Gray, F.G., "A multi-language goal-tree based functional test planning system," *Proceedings of Test Conference*, Oct. 2002, pp. 472 – 481.

[22] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," *Computer*, vol. 29, no. 12, December 1996, pp. 84-89.

[23] MIPS Technologies, Inc., "MIPS32 4K Famly," http://www.mips.com/content/Products/Cores/32-BitCores/MIPS324KFamily/ProductCatalog/P_MIPS324KFamily/productBrief, March 17, 2004.

[24] Bennett, Sidney, "Designing a Compiler for a Distributed Memory Parallel Computing System," MS Thesis, Virginia Polytechnic Institute and State University, November 2003.

# Appendix A The SCMP Instruction Set Architecture

| Instruction | Operands | Opcode | Description |
|---|---|---|---|
| ADD | rd, rs1, rs2 | 0000001 | Adds two registers |
| ADDI | rd, rs, imm | 0000010 | Adds a register and immediate |
| ADDUI | rd, rs, imm | 0000011 | Adds register and immediate (unsigned) |
| AND | rd, rs1, rs2 | 0010110 | Logical AND of two registers |
| ANDI | rd, rs, imm | 0010111 | Logical AND of register and immediate |
| ASH | rd, rs1, rs2 | 0011110 | Arithmetic shift, register operands |
| ASHI | rd, rs, imm | 0011111 | Arith. shift, register and immediate |
| IDIV | rd, rs1, rs2 | 0001111 | Signed division, register operands |
| IDIVI | rd, rs, imm | 0010000 | Signed division, register & immediate |
| IDIVU | rd, rs1, rs2 | 0010001 | Unsigned division, register operands |
| IDIVUI | rd, rs, imm | 0010010 | Unsigned division, reg. & immediate |
| LSH | rd, rs1, rs2 | 0011100 | Logical shift, register operands |
| LSHI | rd, rs, imm | 0011101 | Logical shift, register & immediate |
| MOD | rd, rs1, rs2 | 0010011 | Modulo, register operands |
| MODI | rd, rs, imm | 0010100 | Modulo, register & immediate |
| MUL | rd, rs1, rs2 | 0000111 | Signed multiplication (lower word) |
| MULH | rd, rs1, rs2 | 0001011 | Signed multiplication (high word) |
| MULHI | rd, rs, imm | 0001100 | Signed multiplication (high word) |
| MULHU | rd, rs1, rs2 | 0001101 | Unsigned multiplication (high word) |
| MULHUI | rd, rs, imm | 0001110 | Unsigned multiplication (high word) |
| MULI | rd, rs, imm | 0001000 | Signed multiplication (lower word) |
| MULU | rd, rs1, rs2 | 0001001 | Unsigned multiplication (lower word) |
| MULUI | rd, rs, imm | 0001010 | Unsigned multiplication (lower word) |
| NEG | rd, rs | 0010101 | Negate (two's complement) |
| OR | rd, rs1, rs2 | 0011000 | Logical OR, register operands |
| ORI | rd, rs, imm | 0011001 | Logical OR, register & imm. |
| ROT | rd, rs1, rs2 | 0100000 | Rotate, register operands |
| ROTI | rd, rs, imm | 0100001 | Rotate, register & immediate |
| SGT | rd, rs1, rs2 | 0100110 | Set if greater than, signed |
| SGTI | rd, rs, imm | 0100111 | Set if greater than imm., signed |
| SGTU | rd, rs1, rs2 | 0101000 | Set if greater than, unsigned |
| SGTUI | rd, rs, imm | 0101001 | Set if greater than imm., unsigned |
| SLT | rd, rs1, rs2 | 0100010 | Set if less than, signed |
| SLTI | rd, rs, imm | 0100011 | Set if less than imm., signed |
| SLTU | rd, rs1, rs2 | 0100100 | Set if less than, unsigned |
| SLTUI | rd, rs, imm | 0100101 | Set if less than imm., unsigned |
| SUB | rd, rs1, rs2 | 0000100 | Subtraction, register operands |
| SUBI | rd, rs, imm | 0000101 | Subtraction, reg. & imm. |
| SUBUI | rd, rs, imm | 0000110 | Subtraction, unsigned imm. |
| XOR | rd, rs1, rs2 | 0011010 | Logical XOR, register operands |
| XORI | rd, rs, imm | 0011011 | Logical XOR, reg. & imm. |
| BEQ | reg, disp | 0101011 | Branch if = 0 |
| BGE | reg, disp | 0101110 | Branch if >= 0 |
| BGT | reg, disp | 0101101 | Branch if > 0 |
| BLE | reg, disp | 0110000 | Branch if <= 0 |

|  |  |  |  |
|---|---|---|---|
| BLT | reg, disp | 0101111 | Branch if < 0 |
| BNE | reg, disp | 0101100 | Branch if != 0 |
| BRA | disp | 0101010 | Branch always, imm. disp. |
| BRA | reg | 0110011 | Branch always, reg. address |
| BSR | reg, disp | 0110010 | Branch to subroutine |
| BSR | reg1, reg2 | 0110001 | Branch to subroutine |
| RSR | reg | 0110011 | Return from subroutine |
| LDB | rd, imm(rs) | 0110110 | Load signed byte |
| LDBU | rd, imm(rs) | 0110111 | Load unsigned byte |
| LDH | rd, imm(rs) | 0111000 | Load signed halfword |
| LDHU | rd, imm(rs) | 0111001 | Load unsigned halfword |
| LDW | rd, imm(rs) | 0111010 | Load word |
| LHI | reg, imm | 0110100 | Load high 16 bits of register |
| LLO | reg, imm | 0110101 | Load lower 16 bits of register |
| STB | imm(rd), rs | 0111011 | Store byte in memory |
| STH | imm(rd), rs | 0111100 | Store halfword in memory |
| STW | imm(rd), rs | 0111101 | Store word in memory |
| SEND | rs1 | 1000010 | Send 1 data word |
| SEND2 | rs1, rs2 | 1000011 | Send 2 data words |
| SEND2E | rs1, rs2 | 1000101 | Send 2 data words and end message |
| SENDE | reg | 1000100 | Send 1 data word and end message |
| SENDH | reg, type, imm | 1000001 | Send msg. header, imm. operand |
| SENDH | reg1, type, reg2 | 1000000 | Send msg. header, reg. operand |
| SENDM | reg1, reg2, reg3 | 1000110 | Send data words from memory |
| SENDME | reg1, reg2, reg3 | 1000111 | Send data words and end message |
| ALLOC | reg | 1001010 | Allocates a context |
| END |  | 1001001 | Ends the current thread |
| FREE | reg | 1001011 | Frees a context |
| SUSPEND |  | 1001000 | Suspends current thread |
| NOP |  | 0000000 | No operation |
| OSCALL | reg, type | 1001110 | Simulate OS functions |
| READSR | reg, rs | 0111110 | Read special register |
| WRITESR | rs, reg | 0111111 | Write special register |

## Vita

Mark Benjamin Bucciero was born on January 19, 1979 in Lansdale, PA. In 1997, he graduated from Centreville High School in Clifton, VA. Mark enrolled in Virginia Tech's Computer Engineering program following his high school graduation. He received his bachelor's degree from Virginia Tech in the Spring of 2001.

Mark will complete his Master of Science degree in Computer Engineering in the Summer of 2004. His research was sponsored by the NSF and by Virginia Tech's Bradley Fellowship. Mark will start his career after graduation at Argon Engineering in Fairfax, VA, where he now resides.