A High Performance Interface for a 40-bit Machine on the NuBus

by

David C. Douglas

Submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

in partial fulfillment of the requirements

for the degrees of

BACHELOR OF SCIENCE

and

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1986

© David C. Douglas 1986

The author hereby grants to MIT permission to reproduce and to distribute copies of this thesis document in whole or in part.

Signature of Author _____

Department of Electrical Engineering and Computer Science

May 9, 1986

Certified by _____

Thesis Supervisor: Professor Stephen A. Ward

Certified by _____

Company Supervisor: Steve Krueger

Accepted by _____

Arthur C. Smith, Chairman

Committee on Graduate Students

MASS. INST. TECH.
JUL 2 3 1986
LIBRARIES

Archives

A High Performance Interface for a 40-bit Machine on the NuBus

by

David C. Douglas

Submitted to the Department of Electrical Engineering and Computer Science on May 9, 1986, in partial fulfillment for the degrees of Master of Science and Bachelor of Science in Electrical Engineering.

Abstract

In order to achieve full 32-bit capability, a processor based on a tagged architecture must support word sizes greater than 32 bits. While it is desirable to maintain compatibility with existing 32- and 16-bit devices, problems arise in transferring data from the smaller word size of these devices to the larger word size of the tagged processor. Many of these problems are avoided if the tagged processor can be efficiently interfaced to an existing standardized bus. A high speed memory interface allowing efficient use of a 40-bit processor on the 32-bit NuBus is examined. Two functional designs are presented and evaluated through simulation and numerical analysis.

Thesis Supervisor: Stephen A. Ward

Title: Associate Professor, Computer Science

Company Supervisor: Steve Krueger

Title: Senior Member of Technical Staff, Texas Instruments

Acknowledgements

I would like to thank my thesis advisor, Prof. Steve Ward, who taught me that working with computers and having fun didn't have to be mutually exclusive activities, and also for his attention, advice, and help on this work. Also at MIT, I'd like to thank Sharon Thomas for her help and encouragement. I would like to thank all of the many people I worked with at Texas Instruments, where the work for this thesis was conducted. In particular, Steve Krueger, Mike Amundsen, and Pat Bosshart for being my technical mentors, and Gene Matthews for all of his support and guidance.

I would like to thank my parents, grandparents, and my brother Pete for their constant love and encouragement. I would like to thank Pam, who makes all of this fun, and also all of my great friends and fellow students at MIT who've helped me keep my sanity here.

I would like to dedicate my thesis to the memory of my grandfather, Charles Wilken, who taught me the value of hard work, and to the memory of Eric Ritland, Harvard '86, a close friend who passed away while I was completing the work for this paper.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This paper describes and analyzes the virtual machine changes necessary to implement a 40-bit processor on the NuBus[1]. The goals of the designs in this paper are to minimize the average access time of virtual memory requests from the 40-bit processor to existing NuBus devices. Two approaches will be presented and an example implementation of them on an existing machine will be described. The first approach is based on storing 3 40-bit words per 4 word block. The second is based on the storage of 4 40-bit words in a 4 word block and one extra word. The rest of this chapter will discuss the notation used in this paper, give an overview of the two approaches, and discuss how their performance will be evaluated. The next chapter will discuss the range of machine architectures to which these approaches are applicable. The next two chapters will describe the two approaches in detail, the following chapter will present an example of the necessary changes and subsequent performance of an existing 32-bit machine (the Texas Instruments Explorer Lisp Machine[2]) following a hypothetical upgrade to 40 bits. The final chapter will summarize the results.

[1]Nubus is a trademark of Texas Instruments, Inc.

[2]Explorer is a trademark of Texas Instruments, Inc.

1.1 Notation

This paper will use the following notations which are consistent with those described in the NuBus Specification. *Byte* will refer to a group of 8 bits, *half-word* to 16 bits, and *word* to 32 bits. In this paper, *bword* will refer to groups of 40 bits, this being short for "big word". The *least significant bit (LSB)* of a group will always refer to bit 0, while the *most significant bit (MSB)* will refer to bit $n-1$ of a group n bits long. Bit strings will always be listed from the MSB to the LSB, left to right. The hardware which occupies a NuBus slot will be referred to as a *NuBus module.*

In reference to bwords, *pointer* will designate the lower 4 bytes or 32 bits. The upper byte will be called the *tag.* These terms are usually used in reference to a tagged architecture, and are used throughout this paper in anticipation of the tagged architecture example in Chapter 5.

In the cache sections, the word *block* is used for what is referred to as either a *block* or a *line* in cache literature. References to caches as *local* mean that the cache is in the same NuBus module as the processor, *i.e.* a NuBus access is not necessary in order to obtain data from the cache. The *write-back* update policy will be referred to as *store-in.* All other cache terms try to comply with common usage (see [Smit82]). Cache performance statistics will be represented as probabilities, allowing more formal representation and usage. The following conventions will be used: $\mathbf{P}(A)$ will stand for the probability of event A, $\mathbf{P}(A \mid B)$ will signify the probability of event A given event B, and $\mathbf{P}(A \cdot B)$ will signify the probability of both event A and event B.

This paper frequently makes references such as *the necessary changes* or *the only unit which will change* without specifying what the object is changing from (or sometimes to). It should be assumed that such phrases are refering to the changes necessary in converting a 32-bit processor to a functionally equivalent 40-bit architecture.

Variables in equations will often use the *prime* mark ($'$) to indicate that the value of the variable differs from that of its original use in describing the Explorer

performance.

1.2 Overview of Div3 Method

The Div3 Method is based on a transfer unit of 4 words containing three bwords and an extra byte which can be used to serve a variety of useful purposes. The name is derived from the fact that a factor of 3 is introduced into the addressing scheme, causing a divide-by-3 to be done implicitly or explicitly at some point in the virtual address translation process. The performance of an implementation of this method will depend heavily on how efficiently this translation is done.

The pertinent design variables in this method include the following: the organization of the bwords within the 4 word block, the size of the addressable page, the size of the transfer page, the cache organization, and the method of address translation. These variables are not independent, with the actual number of efficient, feasible combinations being quite small. The effects of different designs on performance are reflected in trade-offs between cache access speed, map access speed, hardware complexity, and transfer efficiency. These effects will be discussed in the chapter describing the general Div3 Method and will be quantified in the example in a later chapter.

1.3 The Separate Tag Method

The Separate Tag Method is based on the storage of 4-bword pointer fields in a 4-word block, and their corresponding tag fields together in 1 word. Physical and disk address spaces are divided into two subspaces – pointer space and tag space. These are identified by pointer and tag space base addresses. The offsets of a pointer block (four pointer words) and its corresponding tag word (four tags) into their respective address spaces are related by a simple binary operation.

An implementation of this method on a 40-bit machine will be very similar to the memory interface of a 32-bit version of the same processor. The only major

change requires the virtual memory system to perform two complementary operations, one each in pointer and tag space, for each virtual memory request. Similar performance estimates will also be made for this method.

Besides the necessary virtual machine changes and their performance effects, implementations of the two methods differ in their ability to share memory with non-40-bit devices. The Separate Tag Method can communicate easily with 8-, 16-, and 32-bit processors by using pointer space as shared memory. However, in the Div3 Method, the non-40-bit device would have to be given some knowledge of how bwords are arranged within the virtual and physical address spaces. This may or may not be an design consideration, depending on the proposed use of the 40-bit processor. This topic will be reintroduced later in more detail.

1.4 Performance Evaluation

In using a 40-bit processor on a 32-bit bus, one must pay an unavoidable performance penalty when accessing data over the bus. This is reflected in an increase in both the time to access memory from the processor, and in the time to process a page fault in a virtual memory implementation. If a cache block size other than a power of 2 is used, as in the Div3 Method, the hit ratio of a local cache may also be effected by the change from 32- to 40-bit words. In an attempt to capture all of these effects in one measurable value, this paper will use the average access time of virtual memory requests as a basis for evaluating performance.

The average access time of a virtual memory request on a general architecture can be expressed by the definitions and resulting equation shown in figure 1.1. These equations are based on the general virtual memory system shown in figure 1.2. Note that it is assumed that the cache and TLB are accessed in parallel, and that this is reflected in equation 1.1. Other cache/TLB arrangements can easily be represented by this equation by choosing appropriate values for t_c and t_{tlb}. Note also that the processing time t_{pgf} overlaps t_c and t_{tlb} in equation 1.1. The delay t_{tlb} will be considered to have a second order effect on the value of t_{pgf} so it has been

$\mathbf{P}(h_c) =$ *probability of a cache hit on a local cache*

$\mathbf{P}(m_c) =$ *probability of a miss on a local cache* $= 1 - \mathbf{P}(h_c)$

$t_c =$ *access time of a local cache*

$\mathbf{P}(h_{tlb} \mid m_c) =$ *probability of a hit on the translation lookaside buffer given a local cache miss*

$\mathbf{P}(m_{tlb} \mid m_c) = 1 - \mathbf{P}(h_{tlb} \mid m_c)$

$t_{tlb} =$ *access time of the translation lookaside buffer*

$t_{mem} =$ *access time of block from main memory*

$t_{pgf} =$ *time to process a page fault on a tlb miss*

$t_{avg} =$ *average access time of a virtual memory request*

$$t_{avg} = \mathbf{P}(h_c) * t_c + \mathbf{P}(m_c) * [\mathbf{P}(h_{tlb} \mid m_c) * (t_{mem} + t_{tlb}) + \mathbf{P}(m_{tlb} \mid m_c) * t_{pgf}] \quad (1.1)$$

Figure 1.1: Computing the Average Memory Access Time of a General Architecture

included implicitly in that value instead of explicitly.

Each of these terms can be expanded as necessary to reflect a particular architecture. Such cases would include architectures with a heirarchy of local caches, multiple actions resulting from a TLB miss, a cache using various update policies, or a cache which handles instructions separate from data. Other cases involve the expansion of the t_{pgf} and m_{tlb} terms.

The terms $\mathbf{P}(h_{tlb} \mid m_c)$ and $\mathbf{P}(m_{tlb} \mid m_c)$ can be simplified in the following manner:

$$\mathbf{P}(h_{tlb} \mid m_c) = 1 - \mathbf{P}(m_{tlb} \mid m_c) \quad (1.2)$$

$$\mathbf{P}(m_{tlb} \mid m_c) = \frac{\mathbf{P}(m_{tlb} \cdot m_c)}{\mathbf{P}(m_c)} \quad (1.3)$$

This paper will assume that the main memory is sufficiently large compared to the cache that:

$$\mathbf{P}(m_{tlb} \cdot m_c) = \mathbf{P}(m_{tlb}). \quad (1.4)$$

Certainly, for any system a sequence of events can be invented in which a data object is in the cache but the physical page on which it resides has been flushed from main

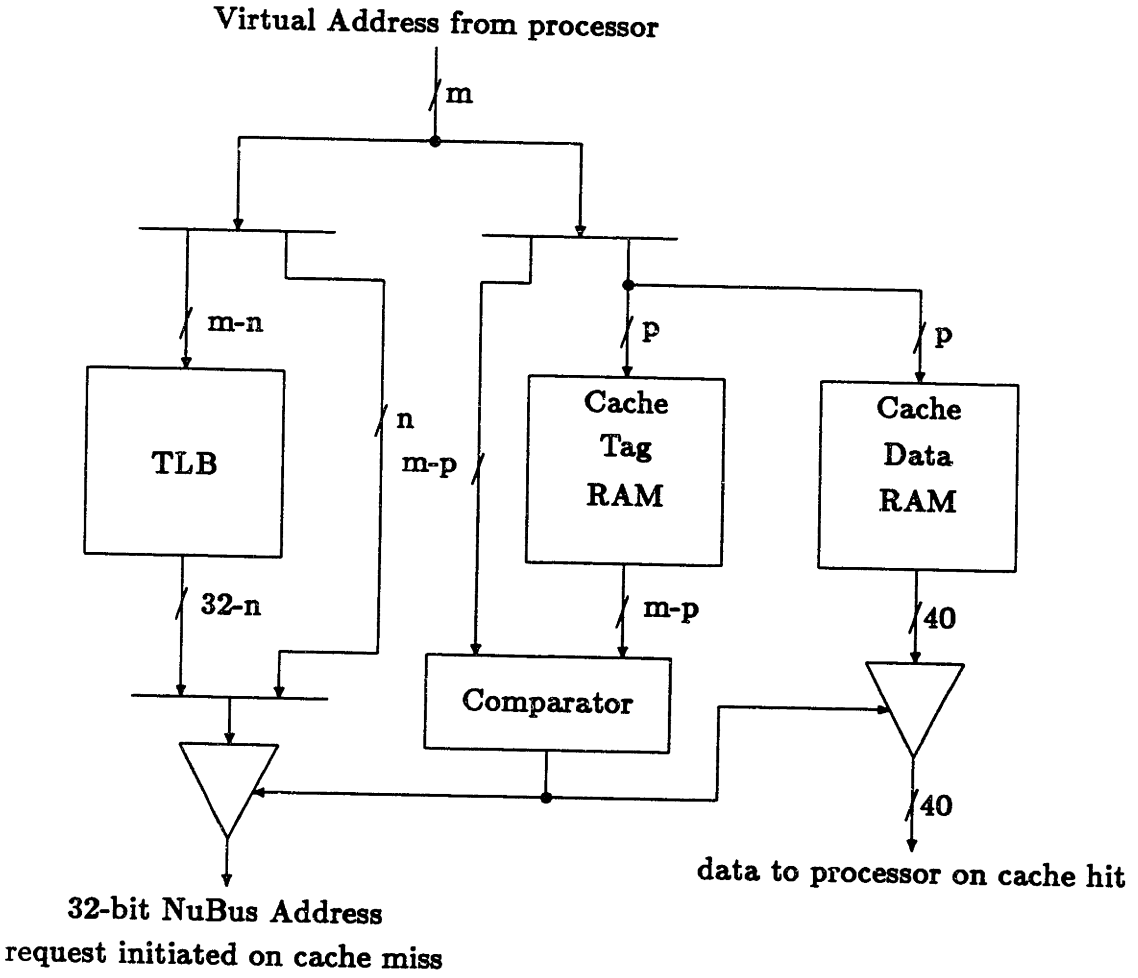


Figure 1.2: General Organization of the Virtual Memory Hardware

memory. It will be assumed that such occurrences are infrequent enough to have no statistical impact on the outcome of these equations. paper which justifies this. Combining equations 1.2, 1.3, and 1.4, it is found that:

$$\mathbf{P}(m_{tlb} \mid m_c) = \frac{\mathbf{P}(m_{tlb})}{\mathbf{P}(m_c)}, \tag{1.5}$$

and

$$\mathbf{P}(h_{tlb} \mid m_c) = 1 - \frac{\mathbf{P}(m_{tlb})}{\mathbf{P}(m_c)}. \tag{1.6}$$

Similar substitutions will be used throughout this paper.

Performance comparisons will be made between the two 40-bit methods and a 32-bit processor on the NuBus. The 32-bit machine is included as a basis for comparisons, assuming that its perfomance is equivalent to that of a 40-bit machine on a 40-bit bus, both running identical access sequences. Specific changes to equation 1.1 will be discussed at the end of the chapters which describe the methods.

Chapter 2

Range of Applications

This chapter discusses the range of 40-bit virtual architectures which are supported by the designs in this paper. By supporting these architectures, these designs should have applications to a wide range of 40-bit machines. If a given application does not require the support of some or any of the features described in this chapter, aspects of the designs may be simplified of left out altogether.

2.1 Compatibility with Existing Hardware

Except in some multiprocessor applications, the designs in this paper require no changes to any existing NuBus devices except the 40-bit processor. Unaffected devices include the NuBus itself, NuBus memory boards, and any other NuBus based peripheral devices. No changes will be required of other processors unless they are intended to share memory with the 40-bit processor. The extent of these changes will depend on the 40-bit method used. All 40-bit processor changes will be noted in the designs, and some notes on the complexity of these changes will be added. All changes can be implemented using commercially available parts, and are easily implemented with semi-custom or custom IC's.

2.2 Virtual Memory Systems

All designs in this paper support a general virtual memory system through the following operations. First, the designs support the dynamic, consistent mapping of virtual addresses onto locations in physical memory. This occurs at the word or bword level. Second, the designs support the transfer of sections of memory between different physical locations or devices, in particular page and disk block transfers. Finally, the designs support unmapped accesses of any memory location. The following sections will discuss each of these operations separately.

2.2.1 Address Mapping

The designs all support the dynamic, consistent mapping of virtual addresses to physical bwords. This means that, even though the physical address of a data object may change over time, it must always be accessible by its virtual address. This operation is among the hardest to meet in implementing a 40-bit machine on a 32-bit bus. The following points about this operation should be noted. First, because of the size difference, a virtual address cannot translate into a single physical address, and a physical address cannot alone represent a virtual address. Second, a physical word may contain parts of one or many virtual bwords, or may never contain any at all. Finally, contiguous, sequential physical words do not necessarily have to represent sequential virtual addresses, even within a page. The address mapping operation supported by these designs guarantees that a set of virtual addresses will always return the same data objects.

2.2.2 Block Transfers

The designs also allow the mapping and transfer of blocks or groups of bwords. The basic operation of this type is virtual memory paging, although I/O transfer blocks, and disk block or sector transfers are also supported by this type of operation. The designs are all compatible with page mapping support such as TLB's and the information necessary to maintain page and block mapping. This includes

the possibility that some portions of these data structures are also stored in virtual memory due to their size. Again, the incompatibility in word sizes presents some anomalies. First, an integral number of bwords does not necessarily occupy a convenient number of words. For this reason, designs may sometimes transfer words which are never used, but are transfered to simplify the management of blocks of bwords. Second, because of the problems described in the last section, knowing the address of a block of bwords does not give one the addresses of the elements of the block. These problems are dealt with differently by the two methods.

2.2.3 Unmapped Accesses

These designs allow the processor to bypass the TLB and access NuBus locations, including memory, with physical addresses. These will be called *unmapped accesses* throughout this paper. This is necessary in order to support boot strapping operations, I/O, and DMA type operations. Other operations which fall below the top level of the virtual architecture also may require unmapped accesses. These include page fault and interrupt handling. Support of this operation requires the ability to access, without hardware support, a single bword using its virtual address. The designs guarantee the ability to obtain the physical addresses of the bword parts from the data in the TLB, and to obtain the bword in a maximum of two NuBus transactions. By obtaining the exact byte and word addresses of the parts of a bword, the unmapped write of a single bword can be done without doing a read to maintain the integrity of neighboring bwords. This is also an important factor in handling mapped writes in most cache organizations.

2.3 Local Cache Memory

Being a crucial part of their efficiency, the designs of this paper fully support local cache memories. For speed purposes, the caches in these designs are addressable with virtual addresses, or in the worst case, virtual addresses which have undergone a single level of translation. All general cache design parameters are supported, in-

cluding address blocks, multiple set sizes, replacement policies, and update policies. The following paragraphs will discuss a couple of these in detail.

In order to reduce the size of the memory used to store cache tags, it is common to store a tag for every n words, where n is the *block size* or number of words transfered into the cache at a time. Notice that in no case is every value of n possible, but is instead a member of the set of values represented by equation 2.1.

$$n = c * 2^i, \quad i \in \{0, 1, 2...\} \tag{2.1}$$

In most designs, c has a value of 1. For the designs in this paper, however, bword storage formats further restrict the possible values of n. For the Div3 Method, c has a value of 3, and for the Separate Tag Method a value of 4.

In implementing update policies, the problem of updating single bwords appears again. If a write-through or store-and-invalidate update policy is used, then on every write the corresponding bword in physical memory has to be updated. If a store-in policy is used, then the dirty words or transfer block must be written out to physical memory when that block is flushed. In either case, it is undesirable to have to write out an entire block in order to update one bword in primary memory. On the other hand, one would want to write to memory without complex hardware support. The tradeoffs between these two approaches will be more apparent as the designs are presented.

2.4 Summary

This chapter has shown the architectural features which are supported by the designs of this paper. They also represent the areas in which many designs fail as potential solutions to the problem of a 40-bit machine on the NuBus. As the two methods are presented, the ways in which these methods handle these problem areas will be highlighted. At those times, it will be apparent which portions of the methods have been included to implement these features. These portions of the methods can then be removed if the feature is not required.

Chapter 3

The Div3 Method

The Div3 Method is based on the NuBus block transfer of 4-word blocks containing 3 bwords. The following sections will describe a series of implementations of this method which retain the speed advantages of fetching 4-word blocks, while overcoming the difficulties of operating a virtual memory system in this environment. The first section will discuss an organization of bwords within the 4-word block, and will also introduce the hardware necessary for realigning bwords once they are obtained over the NuBus. The next section will discuss the changes which will be necessary to a virtual memory system in order to support the odd block sized introduced by the factor of 3. Cache organizations in the Div3 Method will then be discussed followed by a description of the changes which are necessary to equation 1.1 in order to quantify the performance of the Div3 Method.

3.1 Bword Organization

Figure 3.1 shows the alignment of 3 bwords within the 4-word transfer block which will be used by the Div3 Method. This arrangement has been selected due to its logic and time saving effects. The extra byte and a byte from each of the three bwords occupy the first word. These are followed by the remaining parts of the 3 bwords, each occupying a word. The byte from each bword which is placed in the first word of the transfer block can be either the first or last byte of the bword

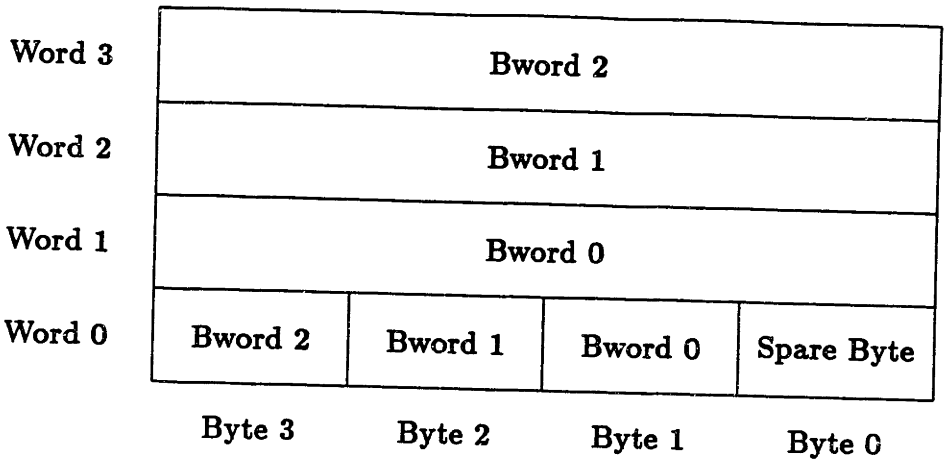


Figure 3.1: Bword Alignment in a 4-Word Transfer Block

without loss of simplicity. Use of any other byte complicates the realignment buffer – the hardware which reorganizes bwords from their parts. The rest of this paper will use the case where the high order byte (the tag field) is stored separate from the low word (pointer field).

When a transfer block is read into the processor from the NuBus, it is desirable to obtain a bword on every cycle after the first one. In this way data can be read out of the realignment buffer in parallel with the last words being read in. For many potential uses of the extra byte in the transfer block it would also be desirable for it to be read in first. Such uses include error correction, caching information, and multiprocessing data consistency information. For these reasons, the extra byte and a byte from each bword have been placed in the first word of the transfer block.

This bword organization also simplifies the addressing of single bwords within the transfer block. No matter what organization is used, at least two memory accesses are required to access a single bword from the block. Physical bword addresses are easily changed to forms which allow them to access the bword through a word access and a byte access, or the whole transfer block through a single block address. Figure 3.2 shows how this is accomplished. The next section will describe how virtual addresses are translated into physical addresses of this form.

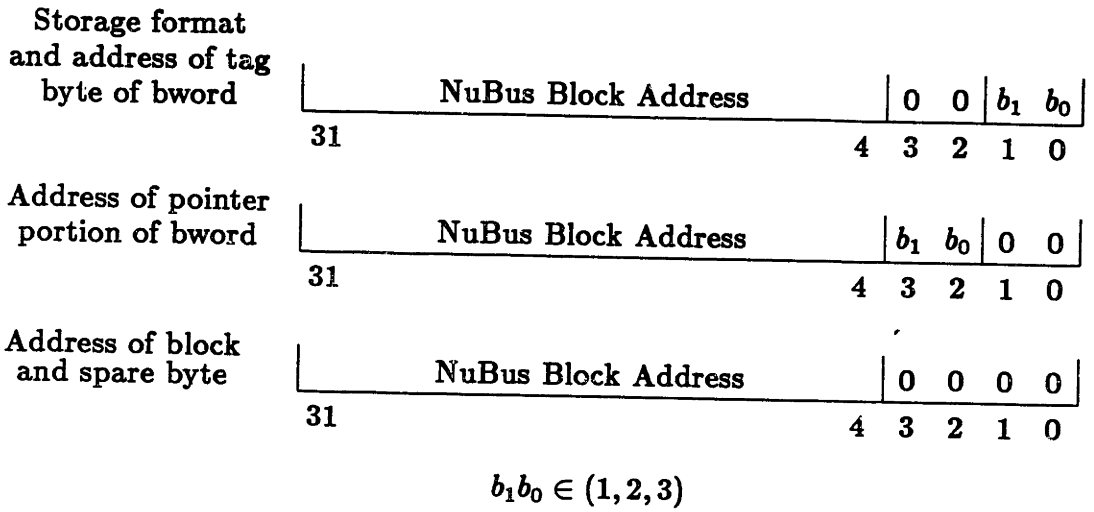


Figure 3.2: Physical Address Formats for Bword Addressing

If memory must be shared with 16- or 32-bit NuBus processor, then the page should be marked as non-lisp and the tag fields of the three bwords will be left empty. Optionally, automatic tag generation hardware similar to that described with the Separate Tag Method could be used in support of this type of page. It should be noted, however, that the other processor must be given knowledge of this format.

The realignment buffer is designed to allow both single bword and 4-word block transfers to occur without complex control logic. The functional diagram in figure 3.3 gives an example implementation.

It will be assumed that the cache will be responsible for organizing the bwords into the transfer block when they are being written over the NuBus. Since the cache must then have access to the tags all at once, then it will also be advantageous to also give the processor access to this tag word. This requires little hardware and may be especially useful for activities such as garbage collection.

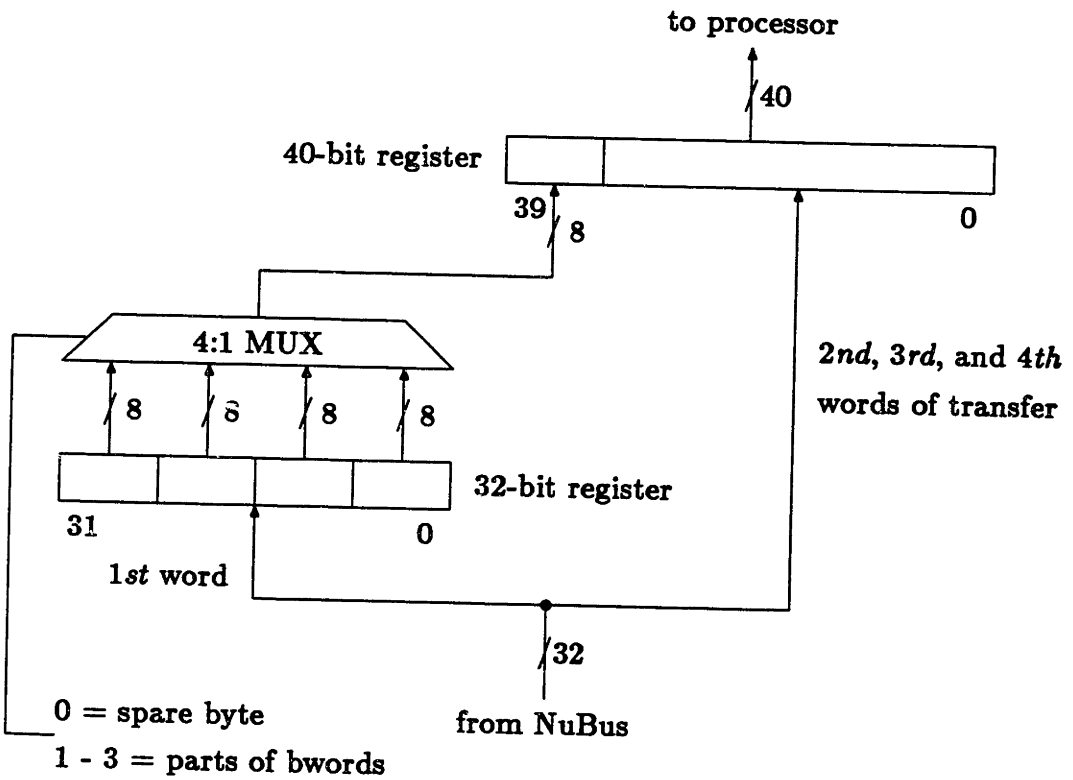


Figure 3.3: Realignment Buffer

Number of bits in offset address	Number of bwords addressed	Number of words necessary for storage
7	128	171
8	256	372
9	512	683
10	1K	1366
11	2K	2731

Table 3.1: Page Sizes in Words and Bwords

3.2 Virtual Memory Management

This section describes how the Div3 Method translates virtual bword addresses into physical addresses of the storage format shown in figure 3.2. By storing 3 bwords per transfer block, a factor of 3 has been introduced into the mapping of addresses into physical memory locations. The level of complexity and efficiency of the Div3 Method is dependent on how this awkward factor is handled by the mapping scheme of the virtual memory system. Mapping schemes for address translation at both the bword and page levels will be examined.

3.2.1 Address Pages

Similar to other virtual memory systems, the Div3 Method uses the low n bits of the virtual address as the word offset into the page, and the remaining virtual address bits as the virtual page number. Since the bword offset into the page does not also represent a physical offset into the page, the offset must also be translated to reference the correct physical locations. This extra translation is not necessary with most standard virtual memory systems. Table 3.1 shows the correlation between the number of offset bits in the virtual address (n), the number of bwords which can be addressed (2^n), and the number of 32-bit words required to hold them ($\frac{4}{3} * 2^n$). Pages of these sizes will be called *address pages*, and will not necessarily be the

same size as the pages which are stored in primary and secondary memory, which will be called *transfer pages*. These differences, along with the translation of the page number will be discussed in the next section.

Notice that the Div3 Method uses virtual page sizes which are a power of 2 number of bwords, as opposed to using page sizes which are a power of 2 words large. If the latter page size were adopted, then the number of bwords in a page would be an odd number, approximately $\frac{3}{4}$ the number of words. In this case, the virtual page number and virtual page offset address fields overlap by 2 bits, causing the following problems: the virtual page offset is now a function of the virtual page number, and the virtual page number depends on the two overlap bits. The first problem exists for the Div3 Method also, but in this case the entire virtual address is needed to obtain the offset. Since this operation lies in the critical path of the cache, this added complication will have adverse effects on the access time of the cache. The second problem causes great difficulty in TLB addressing. It was found that these difficulties cannot be overcome without substantial hardware support.

Given n bits specifying the bword offset into the address page, and given an address which specifies the origin of the physical page, the bword offset must be translated into an offset which points to the correct 4-word transfer block within the physical page. Equation 3.1 shows the mathematical relationship between the virtual bword offset and the physical transfer block offset.

$$TransferBlockOffset = Integer(VirtualBwordOffset * \frac{1}{3 * I}) * 4 * I, \quad I \in \{1, 2, 4\} \tag{3.1}$$

This equation holds for 4- ($I = 1$), 8- ($I = 2$), or 16-word ($I = 4$) NuBus block transfer sizes. Notice that the *Integer* function simply returns the integer portion of its argument.

The following shows how this addressing scheme is implemented for a 4-word, 3-bword transfer block ($I = 1$).

It is clear from equation 3.1 that a divide by three operation cannot be avoided. Assuming that n, the number of bits in the virtual bword offset is not excessive, then the operation can be done by 1 or 2 PALs or ROMs. These would use the

virtual bword offset as an address and return an $(n-1)$-bit quotient and a 2-bit remainder, these being the results of dividing the address by 3. The multiply by 4 is done by shifting the quotient left 2 places. The result of this operation represents the transfer block offset into the physical page.

Another method of doing the divide-by-3 is to notice that the Taylor series expansion of $\frac{1}{3}$ is $\frac{1}{2} - \frac{1}{4} + \frac{1}{8}$..... This operation can be done by a series of shifts, adds, and subtracts. Unfortunately, using current technologies, this elegant method will have a propagation delay at least twice that of a fast ROM or PAL.

Since a transfer block offset represents three virtual bwords, the remainder of the divide-by-3 operation is used to identify one of the 3 bwords in the transfer block. A remainder of a divide-by-3 normally returns values of 0, 1, or 2. Upon examining the alignment of bwords within the transfer block, however, it is clear that the values of 1, 2, and 3 ($remainder+1$) would be more useful. If the remainder is just a PAL or ROM location accessed by the bword address offset, then it is a trivial change to have the PAL or ROM return $remainder+1$ instead of *remainder*. If the Taylor expansion method is being used, then the remainder will have to be incremented following the divide-by-3 operation. This must be done at the end so that only the remainder is affected by this change. This value then represents the byte offset into word 0 and the word offset into the transfer block which correspond to the 5 bytes the addressed bword.

Figure 3.4 gives a functional representation of the translation process which was described in the last two paragraphs. For simplicity the remainder of the paper will refer to the hardware which does the divide-by-3 operation as the Div3 ROM.

This scheme can be easily adapted for block sizes of 8 or 16 words. As equation 3.1 indicates, the divide-by-3 is replaced a divide-by-6 for a block size of 8, and a divide-by-12 for a block size of 16. The multiplication is done by shifting left 3 or 4 bits respectively. Similar arguments about the value of the remainder also apply to these cases. Table 3.2 shows the remainders which should be output instead of the actual values, and figure 3.5 shows an example translation of the same addess for each of the 3 block sizes.

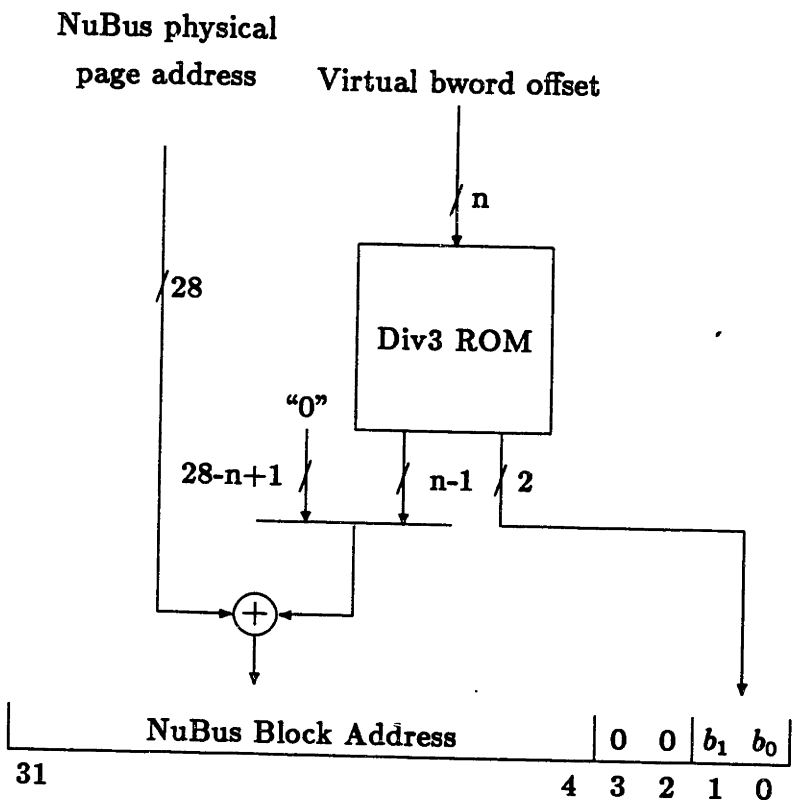


Figure 3.4: Calculating the NuBus Block Address and Block Offset

Real Remainder	Div6 Remainder	Div12 Remainder	Real Remainder	Div12 Remainder
0	1	1	6	9
1	2	2	7	10
2	3	3	8	11
3	5	5	9	13
4	6	6	10	14
5	7	7	11	15

Table 3.2: Remainders Stored in the Div6 and Div12 ROMs

Physical Page Address = F30FF000 (hex)

Virtual Offset = 142 (dec) = 8E (hex)

Output of ROM:

ROM	Block Number		Offset		Physical Block Address
	(dec)	(hex)	(dec)	(hex)	(hex)
Div3	47	2F	1	1	F30FF2F0
Div6	23	17	4	4	F30FF2E0
Div12	11	B	10	A	F30FF2C0

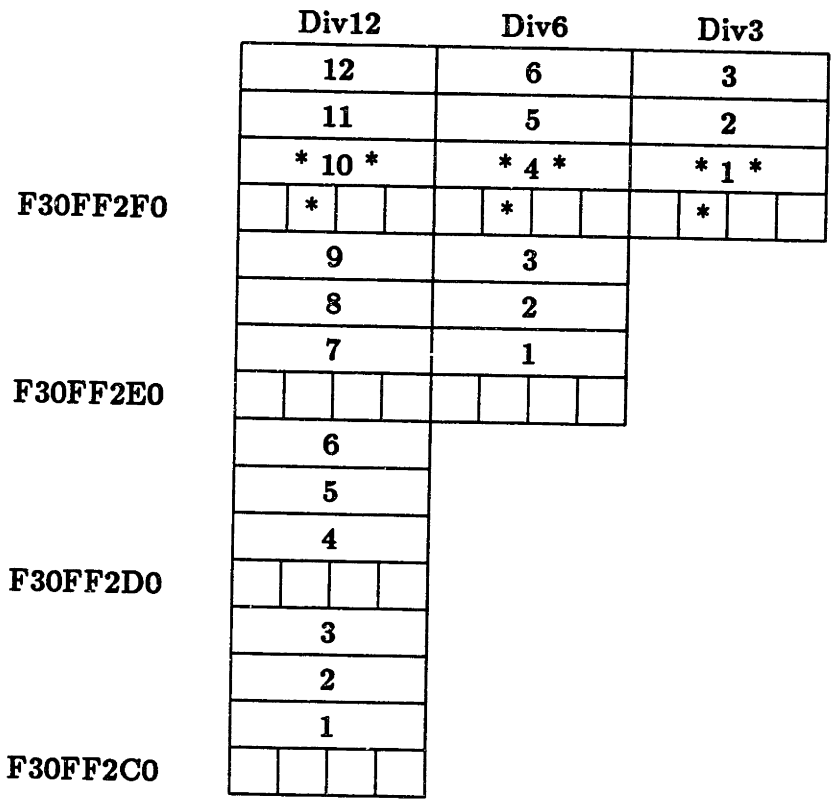


Figure 3.5: Example Address Translation for 3 Different Block Sizes

3.2.2 Transfer Pages

This section describes the translation process of virtual page numbers to physical page addresses, and how the virtual memory system maintains physical pages. The ideas of this section can easily be generalized to apply to the transfer of blocks of data between any devices, not just primary and secondary memory. Due to the odd number of 32-bit words in an address page (recall table 3.1), physical pages in the Div3 Method are often not of a convenient size for systems which normally handle blocks of data whose size is a power of 2. This odd page size can complicate both the physical page number to physical page address translation process, and the derivation of the physical address from the physical page address and the physical page offset. Depending on how the virtual memory system is implemented, some tradeoffs can be made between speed, complexity, and the transfer page size.

As previously mentioned, the *transfer page* is defined which is transfered between primary and secondary storage. In most existing machines this is the same size as the address page, which is obviously optimal as far as memory usage and transfer efficiency are concerned. If this is not an efficient size for some other reason, it is possible to use a transfer page size which is larger than the address page. The bytes past the end of the address page will never be accessed, but *will* be transfered between primary and secondary memory.

The difference between the address page size and the transfer page size is accepted as wasted physical storage space. This waste is tolerated if the transfer page boundaries fall on addresses which simplify the virtual memory system, especially if much of it needs to be implemented in hardware. This waste will also be tolerated if the transfer page size is fixed due to block size requirements of secondary storage devices, or the use of NuBus block transfers for paging.

If transfer pages are of a size which is not a power of 2, then the physical page address cannot be easily obtained from the physical page number. This is a problem since many systems use physical page numbers instead of physical page addresses in managing the pages of physical memory. To resolve this problem, physical page addresses must be regenerated from physical page numbers on each

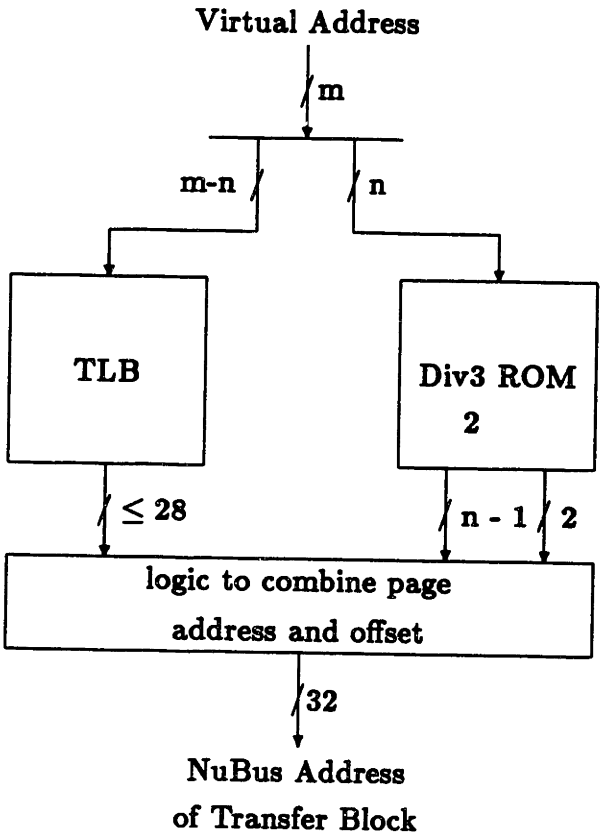


Figure 3.6: Organization of the TLB and Div3 ROM

page fault, or the additional information must be maintained when physical pages become available (virtual pages are flushed), or somewhere in the virtual memory management tables. In addition, there must be a way to generate physical page addresses to initialize the virtual memory system.

Figure 3.6 shows the general organization of the TLB and the logic to generate a physical address of the form shown in figure 3.2. The next few paragraphs will examine the cases where the transfer page is the same(or nearly the same) size as the address page, where the transfer page size is fixed to a power of 2 larger than the address page size, and where the transfer page size is an integral multiple to a power of 2 which is less than the address page size. The differences between these implementations are in the initial values of the physical page addresses and the logic

which follows the TLB.

In the first case the transfer page size is the same or nearly the same size as the address page size. This implies that the physical page address is an odd size and must be maintained separate of the physical page number, since it would be complicated to generated on every page fault. The output of the TLB will be a full 28 bits (the bottom 4 bits are 0's for a 4-word block transfer), and the logic which follows the TLB in figure 3.6 will consist of a 28-bit full adder. By maintaining the full physical page address any size of transfer page could be used, but a size very close to the address page size would be the most efficient in physical memory usage. Therefore, a transfer page size equal to the address page size or the next higher multiple of 8 or 16 words would most likely be used.

The second case fixes the transfer page size to a power of 2 that is greater than the address page size. Assuming a page size of 2^m, the advantage of this case is that the physical page address ends in m bits of 0. Since the transfer page size is larger than the address page size, the output of the Div3 ROM will be m bits or less. The logic following the TLB then needs to append the physical page address to the phyical page offset. This requires no additional hardware and results in no additional delay. It also means that the physical page address and page number are simply related as in most systems. Consequently, the physical page address need not be maintained seperate of the physical page number.

While simplifying physical page management and address generation, the second case is very inefficient in memory usage. If the transfer page size is 2^m words, the address page size is at most 2^{m-1} bwords or $\frac{2^{m+1}}{3}$ words. This results in $\frac{1}{3}$ of physical memory being left unused by the virtual memory system.

The final case is a hybrid version of the other two. It has the efficiency of memory usage of the first case and the simplicity of physical page management of the second case. This is achieved by picking a transfer page size which is easy to multiply the physical page number by. For example, with 256 bwords per page, the address page size is 342 words. If a transfer page size of 384 words is chosen, then the physical page address is found by appending the NuBus slot address to the

Case	Transfer Page Size (words)	Wasted Memory (words/%)	Remarks
1	342	0/0%	Address page size = transfer page size
1	352	10/2.84%	Nearest multiple of 16
2	512	170/33.2%	Nearest power of 2
3	384 (256 + 128)	42/10.9%	Minimum block size of 128, 3 additions
3	352 (256 + 64 + 32)	10/2.84%	Minimum block size of 32, 4 additions

Table 3.3: Consequences of the Different Transfer Page Handling Schemes

physical page size multiplied by 384, which can also be more conveniently expressed as (256 + 128). The physical address is then obtained by the following formula:

$$NuBusBlockAddr = shl24(NuBusSlotNo) + shl10(PhysPageNo) + shl9(PhysPageNo) + shl4(PhysPageOffset). \quad (3.2)$$

Here, *shln(value)* indicates that *value* is shifted left n bits. Note that this generates a NuBus byte address, so multiplying by 256 words is the same as multiplying by 1K bytes. This is easily done in hardware by using a series of adders for the logic following the TLB in figure 3.6. If the adders are pipelined, the delay of 3 or 4 adders will not be much greater than the delay of one.

This final case can be modified in many ways to minimize wasted physical memory, delay through the adder circuitry, or be made to fit the minimum block size requirements of the memory system. Table 3.3 shows a few implementations listed by their case (1, 2, or 3) as presented in the preceding discussion. They all assume an address page size of 256 bwords or 342 words.

3.3 The Div3 Cache

Along with the address translation, the local cache is the feature upon which the performance of the Div3 Method depends. Without memory local to the proces-

sor, every memory access would require a 4 word transfer to take place. If processor slot space is critical, then a small cache or buffer should be implemented, hopefully taking some advantage of the extra bwords which have already been fetched. If space or cost restrictions of this degree do not exist, then a large cache should be implemented. The next sections will discuss cache addressing in the Div3 Method, cache update policies, and small caches and buffers.

3.3.1 Div3 Cache Addressing

This section discusses some of the issues which apply to addressing any cache used with the Div3 Method. Many of these issues are a result of the fact that while memory lies in the 32-bit world and the processor in the 40-bit world, the cache must operate between the two. The bwords are now being stored in 40-bit locations, but the factor of 3 is still present in the block size, and must be dealt with accordingly.

Since 3, 6, or 12 bwords at a time are transfered between memory and the processor, it would seem natural for any cache to have a line or block size equal to the NuBus transfer block size being used. This reduces the cache addressing complexity, allows the cache to store only one tag per block, and uses the minimum number of locations in storing the fetched block. In order to address words in groups of three, however, a divide-by-3 scheme must again be used. This is easily accomplished by using the Div3 ROM. Figure 3.7 shows 2 general cache addressing schemes which support base-3 block sizes using this ROM or PAL.

In discussing cache addressing schemes, it may not always be possible address every cache memory location. These discussions will refer to *effective cache size*, the size of cache counting only those locations which are used.

The first scheme stores bwords in 40-bit locations. Unfortunately, due to the fact that the remainder only takes on 3 different values, the effective cache size will only be $\frac{3}{4}$ of the cache capacity. Any cache data location whose address ends in two 0's will never be used. Note that this problem does not apply to the cache tag memory since only one entry per block is stored there.

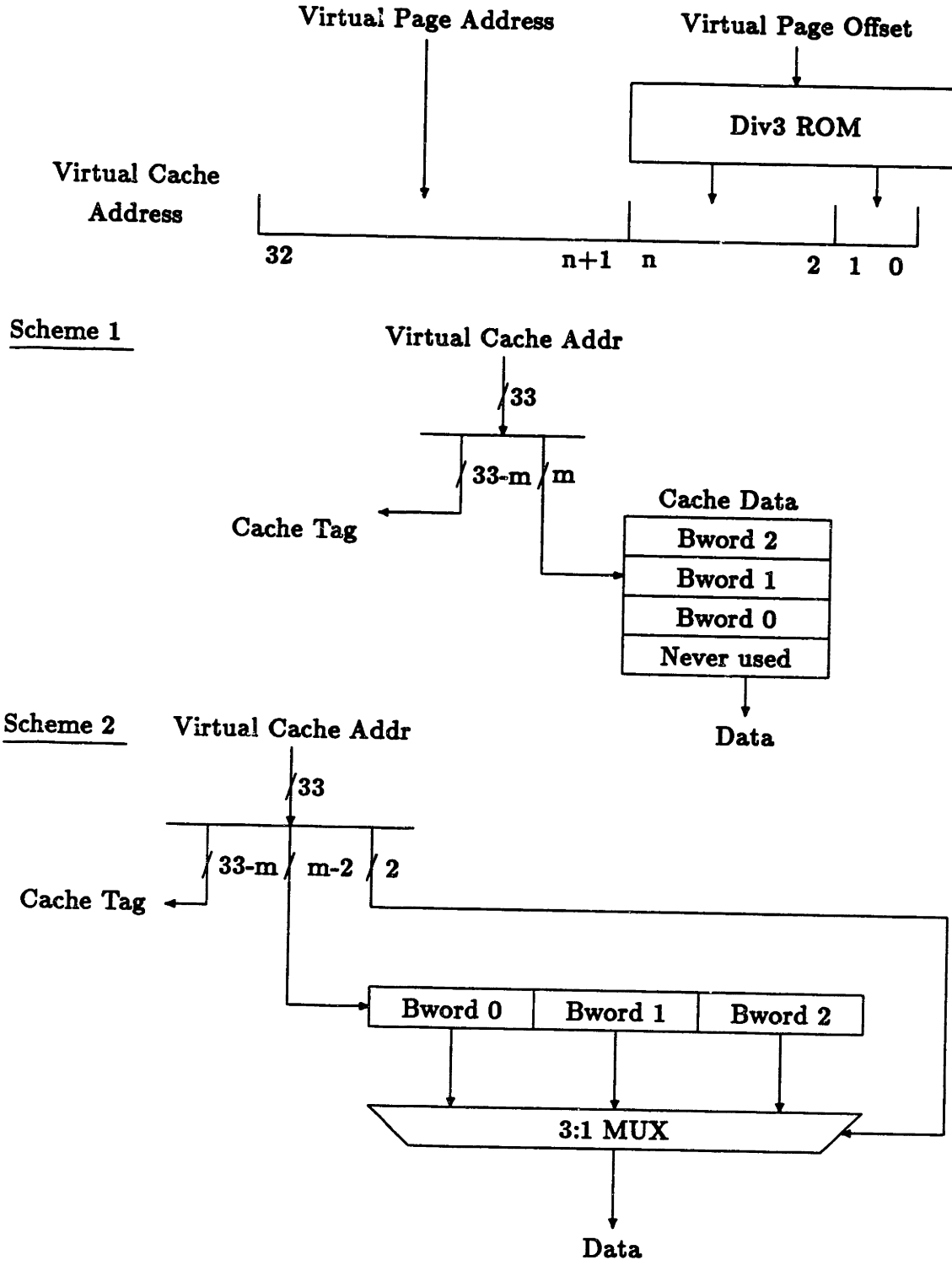


Figure 3.7: Two Cache Addressing Schemes

The second scheme stores bwords in 120-bit locations, 3 per location. In this scheme no memory is wasted, but 120-bit locations must be constructed. The correct bword of the 3 is then selected by the 2 remainder bits. The feasibility of this scheme depends on the implementation technology. Building this cache out of 8-bit SRAMS would not be practicle, especially if a set size larger than 1 was desired. On the other hand, a semicustom integrated circuit implementation of this cache could be practical.

As can be seen in figure 3.7, there is a cutoff point above which the accessing characteristics of the cache change. This is due to the incongruity introduced in the caching system by joining a base-3 address with a base-2 one. The following equations will be used to define three classes of cache size:

$$AddressPageSize < NumberOfSets * BlockSize \tag{3.3}$$

$$AddressPageSize > NumberOfSets * BlockSize \tag{3.4}$$

$$AddressPageSize = NumberOfSets * BlockSize \tag{3.5}$$

Caches which correspond to equation 3.3 will be known as *large caches*. Likewise, caches which correspond to equation 3.4 will be known as *small caches*. The final case, corresponding to equation 3.5 constitute a special case and will be considered separately.

Large caches operate as shown in figure 3.7 without any additional logic. Unfortunately, the high 2 bits of the output of the Div3 ROM will only assume values of 0, 1, and 2. This means that $\frac{1}{4}$ of the usable cache memory will never by used and the effective cache size is further reduced by a factor of $\frac{1}{4}$. For a large cache, this may result in a considerable amount of wasted cache memory. Furthermore, this memory cannot be easily reclaimed since it is scattered throughout the cache in chunks of 2^{n-2}

Small caches do not have the memory waste problem of large caches. This is due to the fact that, except in one specific case, the high 2 bits of the Div3 ROM are not used in addressing the cache memory, but instead form part of the cache tag. The exception occurs when the second highest bit of the output of the Div3 ROM

is being used to address the cache, but the highest one is not. In this configuration, $\frac{2}{3}$ of the address space maps into one half of the cache and the other $\frac{1}{3}$ maps into the other half. This will clearly have a negative effect on cache performance when compared to a cache of comparable size.

The final case is the special case where the address page size is equal to the size of each cache bank, n banks making up an n-way set associative cache. Like the large caches, this size has an effective cache size of $\frac{3}{4}$ of the usable cache memory since the high 2 bits of the Div3 ROM output are not being used. However, unlike the large caches, the unused memory is not scattered about in chunks, but is located all together in the top $\frac{1}{4}$ of the cache memory. This offers the opportunity for it to be used by the processor for some other purpose.

If space allows, an extra Div3 ROM can be used for cache addressing. This ROM would be used for the cache only and would use the correct number of bits so that the output is used to address the cache and the rest of the virtual address is used as the tag. This is identical to the case where equation 3.5 was true, but allows more freedom in picking cache parameters.

3.3.2 Update Policies

The choice of update policy is particularly important to the Div3 Method since it will determine how many single bword memory accesses will be required. The choices are either store-and-invalidate, write-through, or store-in (otherwise known as write-back). There is also the side issue of whether to write single bwords or whole transfer blocks on writes in the write-through policy and with dirty flushes in the store-in policy.

Write-Through Update Policy

The write-through policy is often chosen over the faster store-in policy for two main reasons. First, the cache consistency issues of multiprocessing environments is avoided since primary memory is constantly up to date. Second, the logic necessary to implement write-through is small since the additional information of dirty blocks

and the logic needed to do dirty flushes is not needed. Unfortunately, the write-through cache used with the Div3 Method must pay the penalty of two NuBus accesses on every write.

Store-and-Invalidate Update Policy

While saving some logic compared to write-through, the store-and-invalidate policy becomes less efficient as the block size grows. This is due to the fact that the chance that something else in the block will be accessed while the block is in the cache is proportional to the block size. Since the Div3 Method cache is based on transfer blocks of 3, 6, or 12 bwords, the store-and-invalidate cache will not be very efficient. Instead, the write-through policy, requiring comparable hardware support, should be used instead.

Store-In Update Policy

Store-in has two big advantages over the other two update policies. First, the average access time is lower since the processor does not need to do the write to memory over the NuBus on write misses. Second, bus traffic is reduced for the same reason. The major disadvantage of this policy is the cache consistency problem. Whether this problem will prevent the store-in policy from being used will depend on the environment that the 40-bit machine is to be used in. Another disadvantage is the additional logic needed to maintain dirty bits for, and potentially flush each bword or block.

If cache consistency does present a problem, then either a method to cure the problem must be implemented, or store-in is abandoned as a potential update policy. There are many proposed solutions to the cache consistency problem, and those will not be covered here. Many of these solutions, however, rely on status information about the block in memory. The extra byte in the transfer block could be used as a location with which to pass status information about the block from processor to processor.

Bword or Transfer Block Writes?

In using the any of these policies, the choice of whether to write the whole block or just the dirty bword should depend on how single bword accesses are being handled. If the logic for single bword accesses already exists for unmapped memory accesses and the write-through or store-and-invalidate policy is used, then single bword writes should be used. If this logic does not exist or store-in is being used it may be far less complex to write the entire block back to memory. This has the advantage of having to initiate only one NuBus transaction, while the single bword write will require a word access and a byte access. Additionally, using store-in, more than one bword per block may be dirty and need to be written back anyway.

3.3.3 Small Caches and Buffers

This section will discuss small caches or buffers which can be used when a larger cache is not practicle. For small caches, such as one that would be implemented on a processor chip, a fully-associative cache organization yields the best performance results (this is true for all size caches, but large fully associative caches are usually not practicle to implement). Fortunately, VLSI design supports fast, parallel search over many elements making an efficient fully-associative organization possible. If the Div3 ROM is included on the processor chip along with the cache, then one tag can be stored per 3 bword block, and the parallel search will be $\frac{1}{3}$ as extensive. If the Div3 ROM is off-chip, then a tag will have to be stored for each bword.

A small cache such as this could also be used with a larger off-chip cache. Access times on-chip will be better than those off-chip, especially since the offset ROM is avoided, and a small cache is easy an easy thing to implement in any spare silicon. Unlike other cache organizations, it is not important for the fully-associative organization to have a size which is a power of two, making its size very flexible. Also, if used to complement a larger, off-chip cache, a very small cache operates best as an instruction-only cache since instructions tend to demonstrate better locality of reference (see Appendix B for results which show this). In this

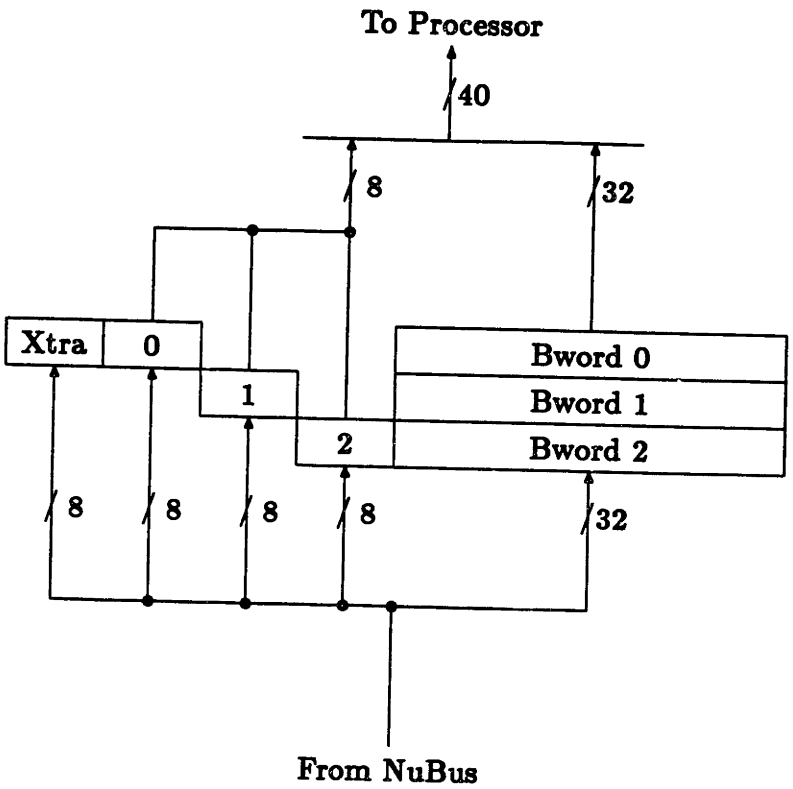


Figure 3.8: Caching Realignment Buffer

way the logic associated with update policies can be ommitted, assuming that the executing instruction set is not self-modifying.

Caches of this type will be very small, and the cost of bringing something into the cache is compounded by the fact that you must flush something that you will probably need later. Since the cache is small due to size or logic restrictions, write-through should be used since it avoids the extra logic of dirty bits and writing back to memory when that block is finally cached.

Finally, if it is impossible to implement a local cache, then the bword realignment logic should be turned into a buffer which stores the last transfer block which was accessed. An example of this is shown in figure 3.8. At least some advantage could be gotten from having transfered the extra words, but this method should only be used when no other arrangement is possible.

3.4 Performance of the Div3 Method

The performance a 40-bit processor which implements the Div3 Method can be compared to that of a functionally equivalent 32-bit processor by noting that a 4-, 8-, or 16-word NuBus block transfer is necessary to fill the block of either cache. For example, the Div3 cache has a block size of 3 bwords which are filled by a 4-word block transfer, while the corresponding 32-bit cache with a block size of 4 words also requires a 4-word block transfer. In a similar manner, the 6-bword block cache and the 8-word block cache both use 8-word block transfers, while the 12-bword block cache and the 16-word block cache both use 16-word block transfers. The differences in their performance will be reflected in the respective hit ratios which will differ due to the number of elements per block.

Some additions to equation 1.1 must also be made in order to accurately predict the performance of the Div3 Method. These are the addition of the access time of the Div3 ROM to the cache access time and the addition of the logic following the TLB to the main memory access time.

The average access time for a general Div3 Method implementation can be represented by the equation 3.6. It is assumed that the page fault occurrence ($\mathbf{P}(m_{tlb} \mid m_c)$) will be identical for the two systems, and that the access times associated with disk (reflected in t_{pgf}) will be similar, although this will may not be true depending on the size of primary memory and the manner in which page faults are resolved.

3.5 Summary of Div3

Figure 3.10 shows the layout of a generalized Div3 Method system. The only hardware additions which have been made are the Div3 ROM and the appropriate logic following the TLB. The Div3 ROM will divide by 3, 6, or 12, depending on the desired NuBus transfer and cache block size. This is implemented with 1 or 2 8-bit PALs or ROMs, or shift and add hardware to implement the Taylor series expansion of $\frac{1}{3}$. The delay which this logic introduces in cache addressing will depend on the

$\mathbf{P}(h_c)l =$ *probability of a cache hit on a local cache*

$\mathbf{P}(m_c)l =$ *probability of a miss on a local cache* $= 1 - \mathbf{P}(h_c)l$

$t_{div3} =$ *access time of Div3 ROM*

$t_cl =$ *access time of a local cache* $= t_c + t_{div3}$

$\mathbf{P}(h_{tlb} \mid m_c)l =$ *probability of a hit on the translation lookaside buffer given a local cache miss*

$\mathbf{P}(m_{tlb} \mid m_c)l = 1 - \mathbf{P}(h_{tlb} \mid h_c)l$

$t_{log} =$ *access time of logic after the TLB*

$t_{tlb}l =$ *access time of the TLB* $= t_{tlb} + t_{log}$

$t_{mem} =$ *access time of block from main memory*

$t_{pgf} =$ *time to process a page fault on a tlb miss*

$t_{avg}l =$ *average access time of a virtual memory request*

$$t_{avg}l = \mathbf{P}(h_c)l * t_cl + \mathbf{P}(m_c) * [\mathbf{P}(h_{tlb} \mid m_c)l * (t_{mem} + t_{tlb}l) + \mathbf{P}(m_{tlb} \mid m_c)l * t_{pgf}] \quad (3.6)$$

Figure 3.9: Computing the Average Access Time of the Div3 Method

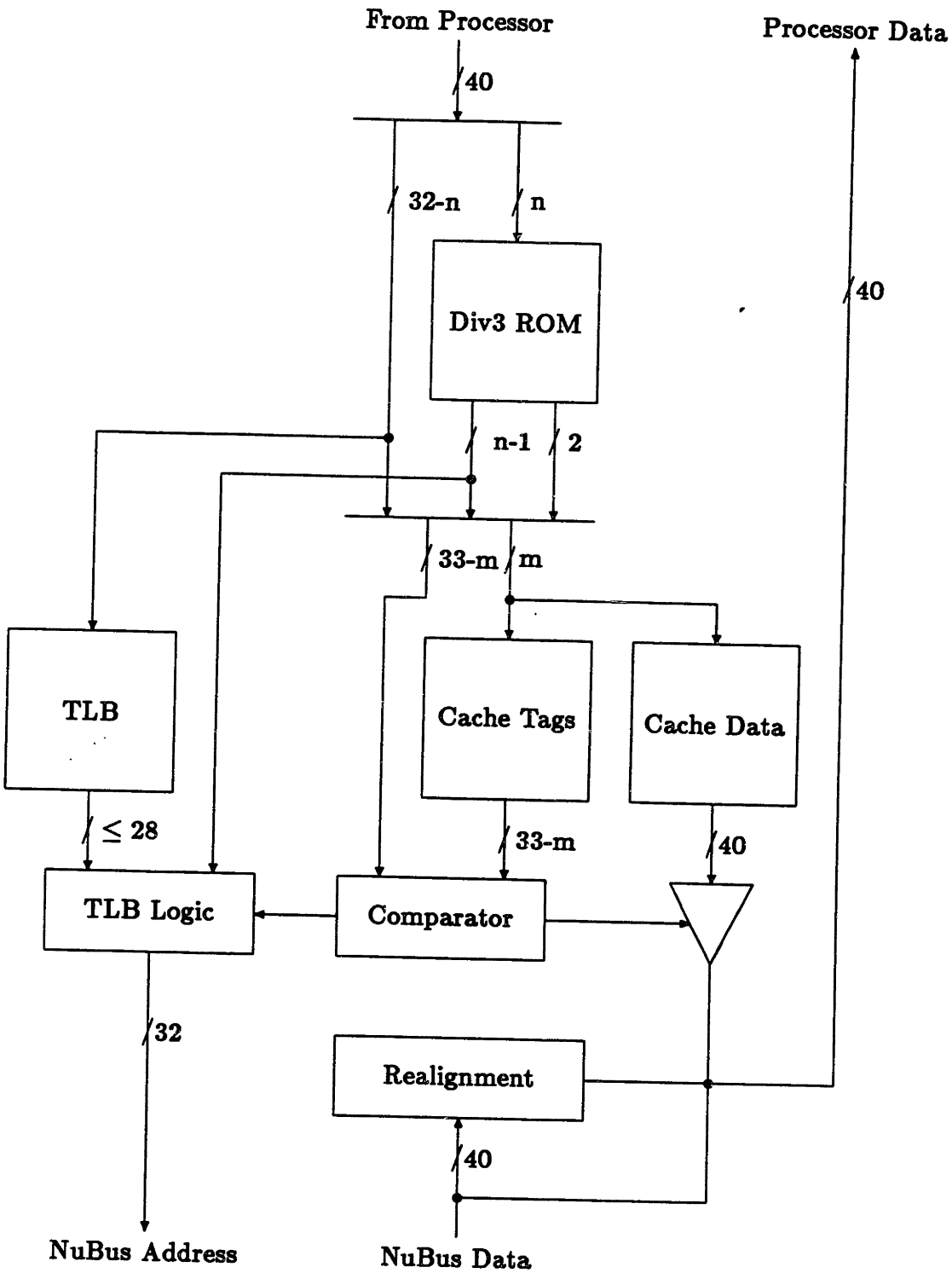


Figure 3.10: Functional Diagram of a General Div3 System

implementation, but could be made as low as 15 ns with currently available devices. The logic following the TLB will consist of either nothing, a single full adder, or multiple full adders. These can be implemented with a variety of commercially available parts with delays as small as 31 ns. The speed of these parts is not as critical as the Div3 ROM which lies in the critical path far more frequently.

With the exception of unmapped memory accesses, the management of bwords is totally transparent to the processor. The Div3 Method has been designed to allow the processor to initiate unmapped accesses with minimal knowledge of the organiation of bwords in memory. This is done by offering a format for storing NuBus addresses from which the addresses for both parts of the bword are easily derived.

The Div3 Method is also attractive because of range of ways in which it can be implemented. Most notably it has a transfer page size can be selected which is compatible with the other NuBus devices of the system. An efficient cache design can also be selected. All cache organizations correspond to a NuBus block transfer size so that the NuBus is utilized in its most efficient manner and bus traffic is kept to a minimum.

Chapter 4

The Separate Tag Method

The Separate Tag Method is based on the storage of 4 bwords in 5 words. In this method, as in the Div3 Method, bwords are divided into two parts. The pointer portion is stored one per word in 4-word blocks, and the tag portion is stored four per word. All types of physical memory spaces are then divided into two parts, a pointer space and a tag space. As the names suggest, tags are stored in tag space and pointers in pointer space. The exact locations of a pointer-tag pair are related to their respective space base addresses by a simple binary operation. The following sections will discuss the organization of bwords in physical memory, the handling of virtual and unmapped memory requests, and the effects of the method on local caches. The final section will discuss performance evaluation of the Separate Tag Method.

4.1 Bword Organization

Bwords are organized in the two spaces as shown in figure 4.1. As in the Div3 Method, it is desirable to transfer the tag word first, allowing a bword to be obtained on each additional NuBus transfer. A realignment buffer similar to the one used in the Div3 Method is necessary to reorganize the 5 words into their appropriate bwords. This is shown in figure 4.2.

It will be again assumed that the cache will be responsible for reorganizing

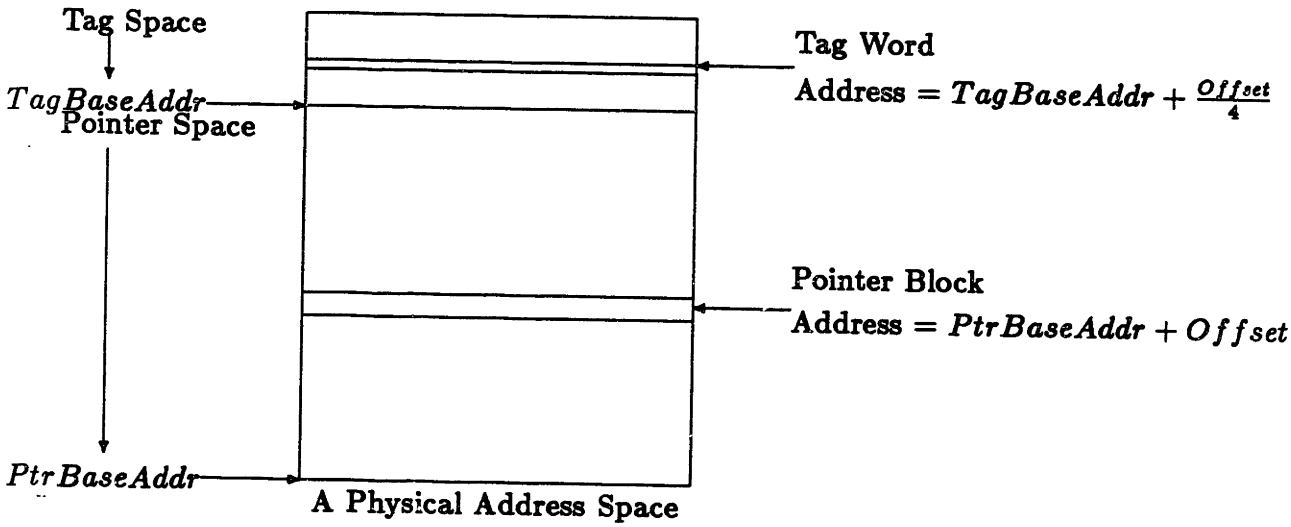


Figure 4.1: Arrangement of the Pointer and Tag Space

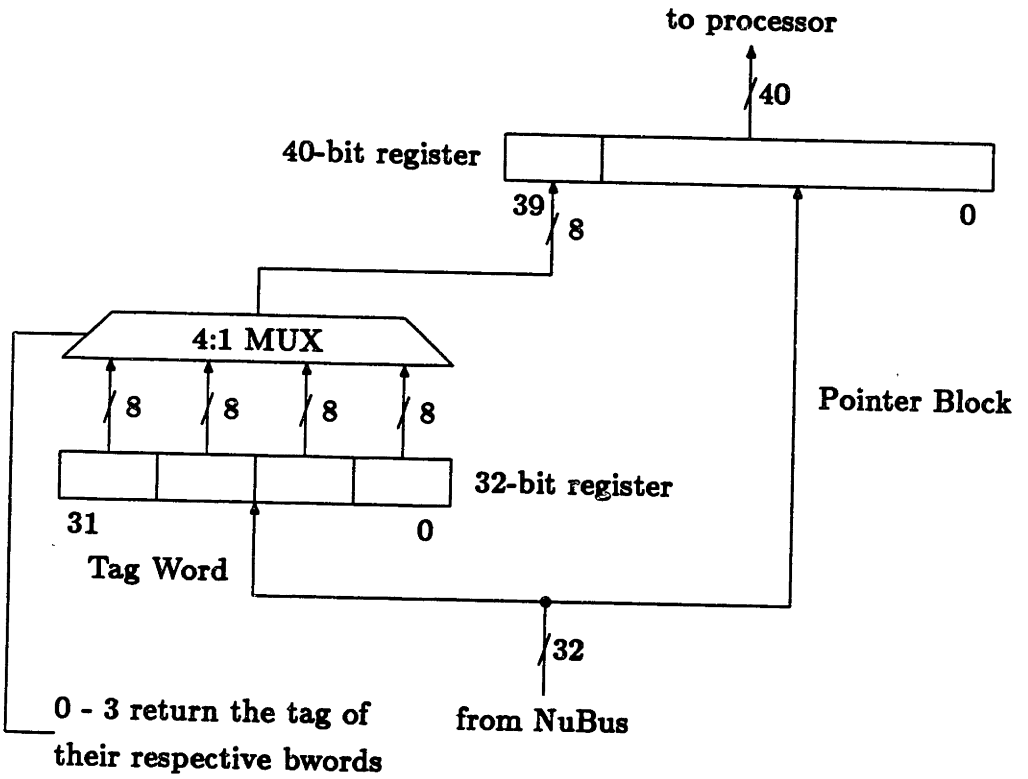


Figure 4.2: Realignment Buffer

blocks going to the NuBus, and it will again be advantageous for the processor to have access to the tag word without the associated pointers (see section 3.1).

One advantage of the Separate Tag Method over the Div3 Method is that it allows memory to be easily shared with 8-, 16-, and 32-bit processors without changes to those processors. This can be accomplished in the Separate Tag Method using either of two ways. First, some or all of the pointer space can be used as shared memory, requiring only that the base address and range of this space be made known to the non-40-bit processors. The words in the tag space should be marked appropriately to indicate that their pointers are in shared memory. This is necessary because the tag words will not be updated by a non-40-bit processor which accesses the shared pointer space.

The other way involves setting aside a portion of physical memory as shared memory, and not maintaining tags for that part of memory. These physical pages must be marked appropriately in the physical and virtual page information of the 40-bit processor. When a memory request to one of these areas is made, the tag fetch of the memory access is not done since there is no tag space associated with the physical address. In its place, a word of tags must be read in from the processor or from somewhere in the processor's memory management hardware. Figure 4.3 shows an example implementation of this idea.

4.2 Memory Management

To support the Separate Tag Method, all physical memory spaces (primary and secondary) which are to be accessed over the NuBus must be divided into a pointer and tag space. It is possible to divide a device into more than one pointer-tag space pair. This would be a good idea for large spaces such as a disk in order to maintain some locality between the two parts of a page. Unless a portion of the memory is to be set aside for pure 32-bit transactions, tag space will occupy $\frac{1}{5}$ of each physical memory device with the remaining $\frac{4}{5}$ set aside for pointer space. Both spaces are defined by a base address and a size, given in words. This section will describe

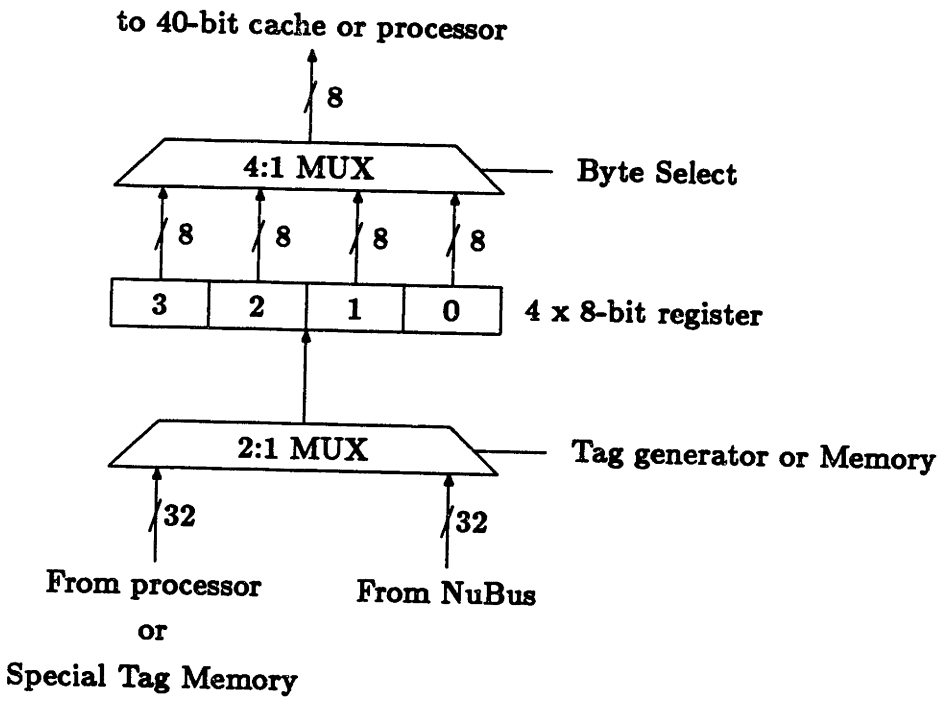


Figure 4.3: An Example of Automatic Tag Generation Hardware

how virtual memory requests and unmapped accesses of bwords are handled in this method.

4.2.1 Virtual Memory Requests

The virtual memory system of a Separate Tag Method implementation differs very little from that of a functionally equivalent 32-bit processor. The main difference is the necessity of the hardware to initiate two physical memory accesses for each virtual memory request. This is handled by storing the information of more than one physical location in the TLB, and by maintaining a request queue for the NuBus. Figure 4.10 shows a functional diagram of the basic memory system of the Separate Tag Method.

Unlike the Div3 Method, the sizes of all data blocks in the Separate Tag Method are powers of 2. Assuming a virtual page size of 2^i bwords, the size of the pointer space page will be 2^i words and the tag space page $\frac{2^i}{4}$ or 2^{i-2} words. These convenient sizes simplify the handling of pages. First, every location in a page is used, so no memory is wasted, as in the Div3 Method. Second, the physical page address is found by adding properly shifted versions of the physical page number to the physical base addresses of the pointer and tag spaces. This allows the resolution of page faults to be done with physical page numbers, thereby avoiding the maintenance of physical page addresses outside the TLB.

It should be noted that secondary memory devices must be able to support data blocks of size 2^{i-2}. The transfer page flexibility of the Div3 Method is not possible in this method since the next larger size would be at least twice as big, resulting in half of tag space being wasted. If the secondary memory devices will not support this size of data block, then either the page size must be increased or the Separate Tag Method cannot be used.

Figure 4.5 shows the layout of the TLB entry for one page. The information of two NuBus addresses can be stored in one TLB location since both locations use the same NuBus slot address, and because the TLB entries are assumed to be 40-bits wide. The TLB entry contains the common 8 NuBus slot bits and separate

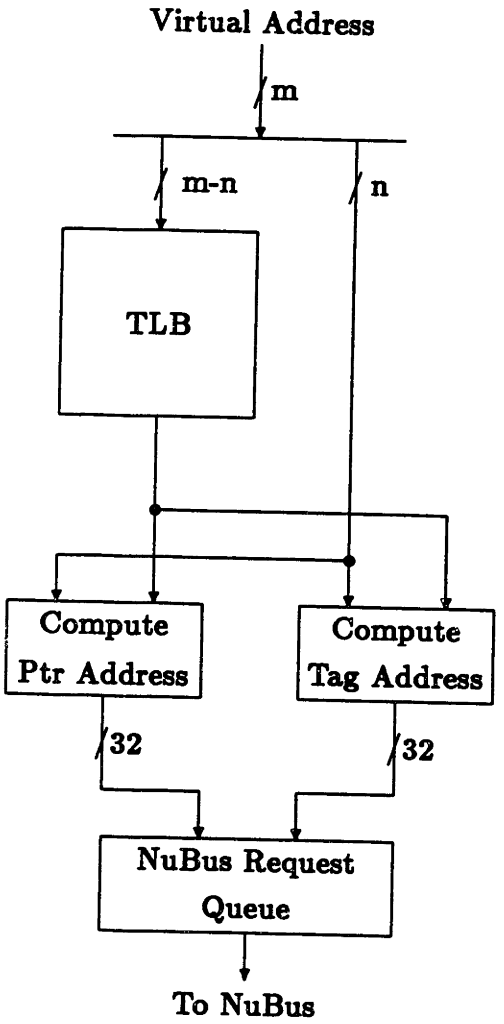
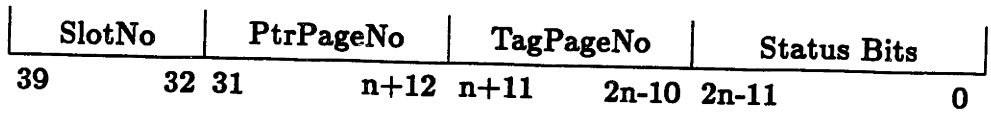


Figure 4.4: General Virtual Memory System of the Separate Tag Method



n = number of bits in virtual page offset
SlotNo = NuBus address of slot where data resides
PtrPageNo = Slot offset address of pointer space page = (20 - n)-bits
TagPageNo = Slot offset address of tag space page = (22 - n)-bits

Figure 4.5: Format of a Translation Lookaside Buffer Entry

page addresses for the pointer block and the tag word. The virtual page offset is saved and appended to both physical page addresses. Figure 4.6 shows how NuBus addresses are generated in this fashion.

Note that this TLB entry organization assumes that it is desirable to fit each entry in 40 bits. If the number of virtual page offset bits (n) is 8 or less, then the number of bits left for status information may be very small. This, combined with the fact that the penalty for a TLB entry size different from 40 bits may be small, suggests that a different size may be optimal. Figures 4.5 and 4.6 do, however, show that the relationship between the two addresses is such that they can be compactly stored together, in one bword if necessary.

The TLB usually contains some status bits for each entry which give information about that particular page in memory. There are two particular status fields which might be especially useful for the Separate Tag Method. The first would indicate whether a given page was part of shared memory or not, and give some indication about the type of sharing that is taking place. The second field indicates if that page has a corresponding tag space or not. Unlike the last field which was related to the physical page, this trait is related to the virtual page. If a page has no tag space, then that half of the access can be ignored, and a set of tags must be generated elsewhere.

The two least significant bits of the virtual address are used to point to the

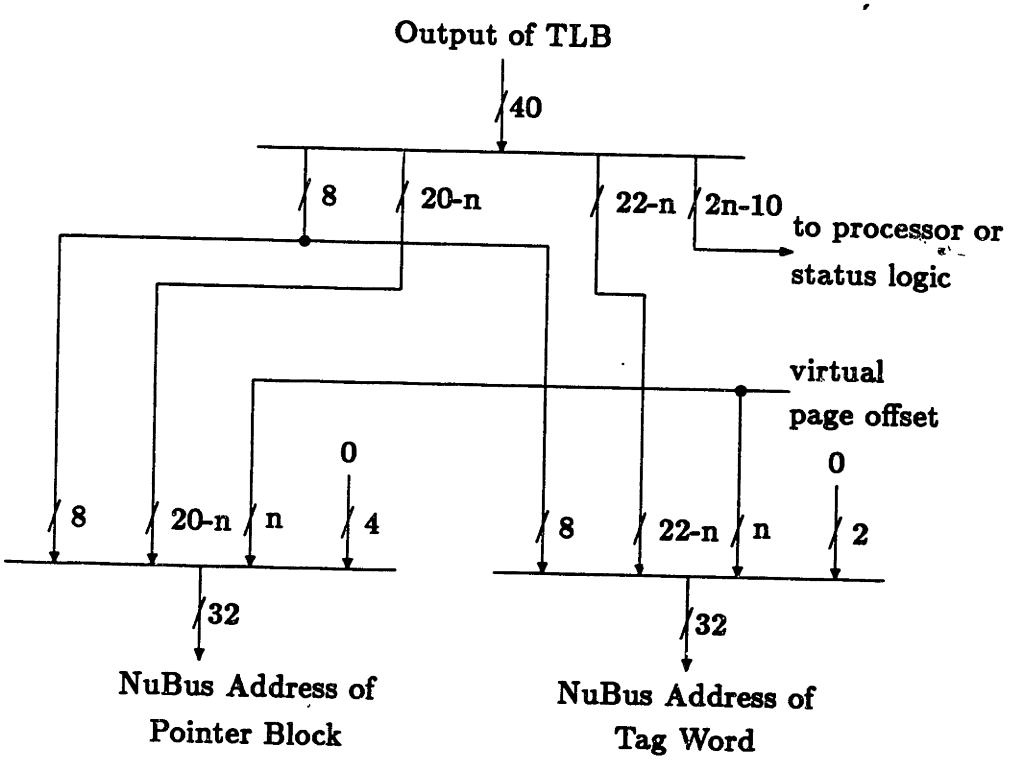


Figure 4.6: Physical Page Generation Using the Output of the TLB

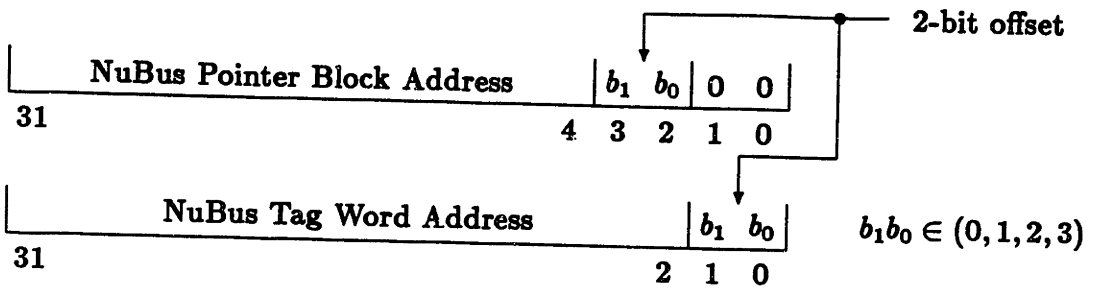


Figure 4.7: Method of Addressing Single Bwords

appropriate word of an accessed block. They can also be used in single bword operations to point to the two parts of the bword. Figure 4.7 shows how single bword addresses are generated in this fashion. Note that these bits are always appended and never cause an addition.

4.2.2 Unmapped Memory Requests

Unmapped memory requests in the Separate Tag Method are complicated by the fact that the physical addresses of the two parts of a bword are, in general, unrelated. If one assumes that the only relation between the two addresses is that they reside on the same NuBus module, then two bwords are required for physical address storage. On the other hand, if a more specific relationship is assumed, then the addresses can be stored in one bword. Finally, if unmapped accesses are prevalent, then it may be worthwhile to add logic which stores an encoded address space number, allowing the two physical addresses to be stored in the same bword. The next paragraphs will examine these three alternatives.

The easiest solution to this problem is to store the addresses to the two parts of the bword in consecutive bwords. Since the accesses are over the NuBus, they will take longer than two microinstructions. Therefore, implementing the second part of the operation explicitly from microcode will show no performance difference for the processor. This method of storing physical addresses adapts easily to the

cases where the pointer and tag portion of the bword are not both needed. All of these cases are handled by just sending one physical address to the NuBus request queue.

The second solution stores the address in one bword. This will be the tag address, and its access will be started immediately. While this access is completing, the pointer block address can be calculated from the tag address, and its address will be placed on the queue as soon as it is generated. Presumably the relationship between these two addresses can be made simple enough to be calculated during the first word access, allowing the second access to start immediately afterward.

The other method takes advantage of the fact that the pointer and tag portion are offset from their space base addresses by the same number of words or bytes. In this method, a pointer-tag space number is stored with the two offsets into the two spaces. The pointer and tag space base addresses are obtained by ROM or register lookup using the space number. Figure 4.8 shows how the addresses are generated. This involves either additional hardware or microcode support. This case would only be used if there were extensive unmapped accesses of both the pointer and tag sections of a pointer-tag space. While saving register, stack, or memory space by storing both addresses in one bword, the additional logic or microcode support of this method may outweigh its benefits.

With the NuBus request queue already built into the system, the problems related to cache update policies and single word accesses are greatly lessened. Since the cache only deals with virtually addressed bwords, the TLB and queue enable single bword and block transfers from the cache without additional logic. Other update policy issues will be dealt with in the next section.

4.3 Caching with the Separate Tag Method

Unlike the Div3 Method, the Separate Tag Method cache lies fully in the 40-bit addressing world. Therefore, the cache is organized in the traditional manner. This means that no cache memory is wasted in this method. The only restriction is that

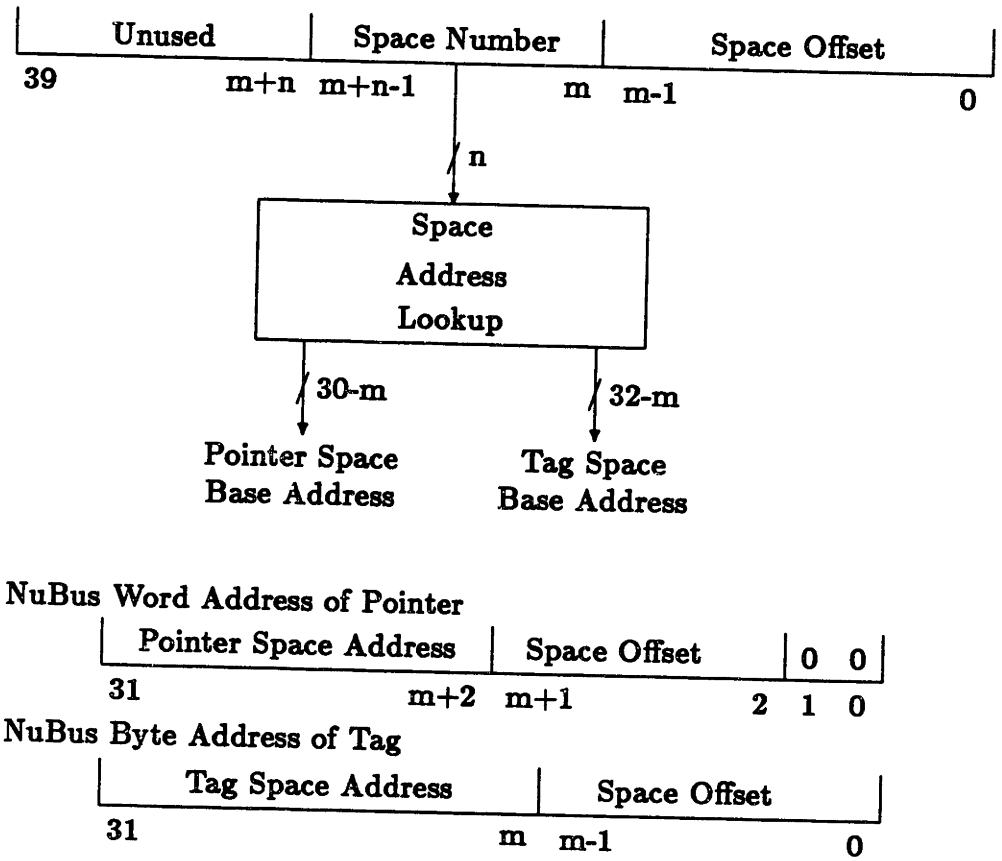


Figure 4.8: Unmapped Bword Address Generation Using Space Lookup

block size must be 4, 8, or 16 bwords. As in the Div3 Method, the cache block size should correspond to the NuBus transfer block size which is selected. In the case of the Separate Tag Method, this means that the number of words in the transfer block which contains the pointer fields should be the same number of bwords in a cache block.

The Separate Tag cache is compatible with all update policies. Cache consistency problems still apply to the store-in policy, however. Since this is a traditional cache organization, any cache consistency solution which has been applied to another cache should also be applicable here.

In matters involving the writing or access of single bwords from the cache to main memory, the situation is a little simpler than in the Div3 Method. This is due to the fact that a NuBus request queue has already been established in order to implement the Separate Tag Method, so the hardware for such activity is already in place. There will be some additional control logic involved with write-throughs or dirty flushes, but the problem has been reduced to the same state that it would be in on any 32-bit machine.

The Separate Tag Method cache will not be described in any further detail since it resembles a conventional 32-bit cache in all aspects. This fact can be especially useful if a cache was already designed for a 32-bit version of the 40-bit machine, in which case it can be stretched to 40 bits and used as designed in the larger machine.

4.4 Performance of the Separate Tag Method

The performance of the Separate Tag Method will depend solely on the hit ratio of its cache. Due to the nature of its storage format, handling a cache miss will take twice as long as it would normally since two NuBus accesses must be initiated. Aside from cache access times, the actual implementation of the Separate Tag Method will not have as large of an effect on the overall average access time as the Div3 Method. This is because no hardware has been added to the critical timing path of cache hits.

$\mathbf{P}(h_c) =$ *probability of a cache hit on a local cache*

$\mathbf{P}(m_c) =$ *probability of a miss on a local cache* $= 1 - \mathbf{P}(h_c)$

$t_c =$ *access time of a local cache*

$\mathbf{P}(h_{tlb} \mid m_c) =$ *probability of a hit on the translation lookaside buffer given a local cache miss*

$\mathbf{P}(m_{tlb} \mid m_c) = 1 - \mathbf{P}(h_{tlb} \mid m_c)$

$t_{tlb} =$ *access time of the translation lookaside buffer*

$t_{mem'} =$ *access time of block from main memory*

$t_{pgf} =$ *time to process a page fault on a tlb miss*

$t_{avg} =$ *average access time of a virtual memory request*

$$t_{avg} = \mathbf{P}(h_c) * t_c + \mathbf{P}(m_c) * [\mathbf{P}(h_{tlb} \mid m_c) * (t_{mem'} + t_{tlb})\mathbf{P}(m_{tlb} \mid m_c) * t_{pgf}] \quad (4.1)$$

Figure 4.9: Computing the Average Memory Access Time of a the Separate Tag Method

The equations for the performance of the Separate Tag Method are identical to the 32-bit version, with the exception of the delay to handle a cache miss, i.e. perform two NuBus accesses instead of one. These are shown in figure 4.9.

4.5 Summary of the Separate Tag Method

Figure 4.10 shows the layout of a general Separate Tag Method implementation. The only hardware additions are the NuBus request queue, the realignment buffer, and the optional tag generation and address space registers. These add no additional delay to the memory system.

Due to the nature of the addressing scheme, the Separate Tag Method makes full and efficient use of main and cache memories. It also allows a conventional cache design to be used, making better use of current cache design experience.

The Separate Tag Method provides a format for sharing memory with 8-, 16-, and 32-bit processors over the NuBus. It also provides a manner for accessing data in areas which are known to be only 32-bits wide, thus having no corresponding tag field.

Finally, the performance of the Separate Tag Method is good as long as the cache is being accessed, but is poor otherwise since 2 separate NuBus accesses must be made to fill the cache. In this method, as in the Div3 Method, the cache performance is the key to the overall system performance. This will be more apparent in the next chapter.

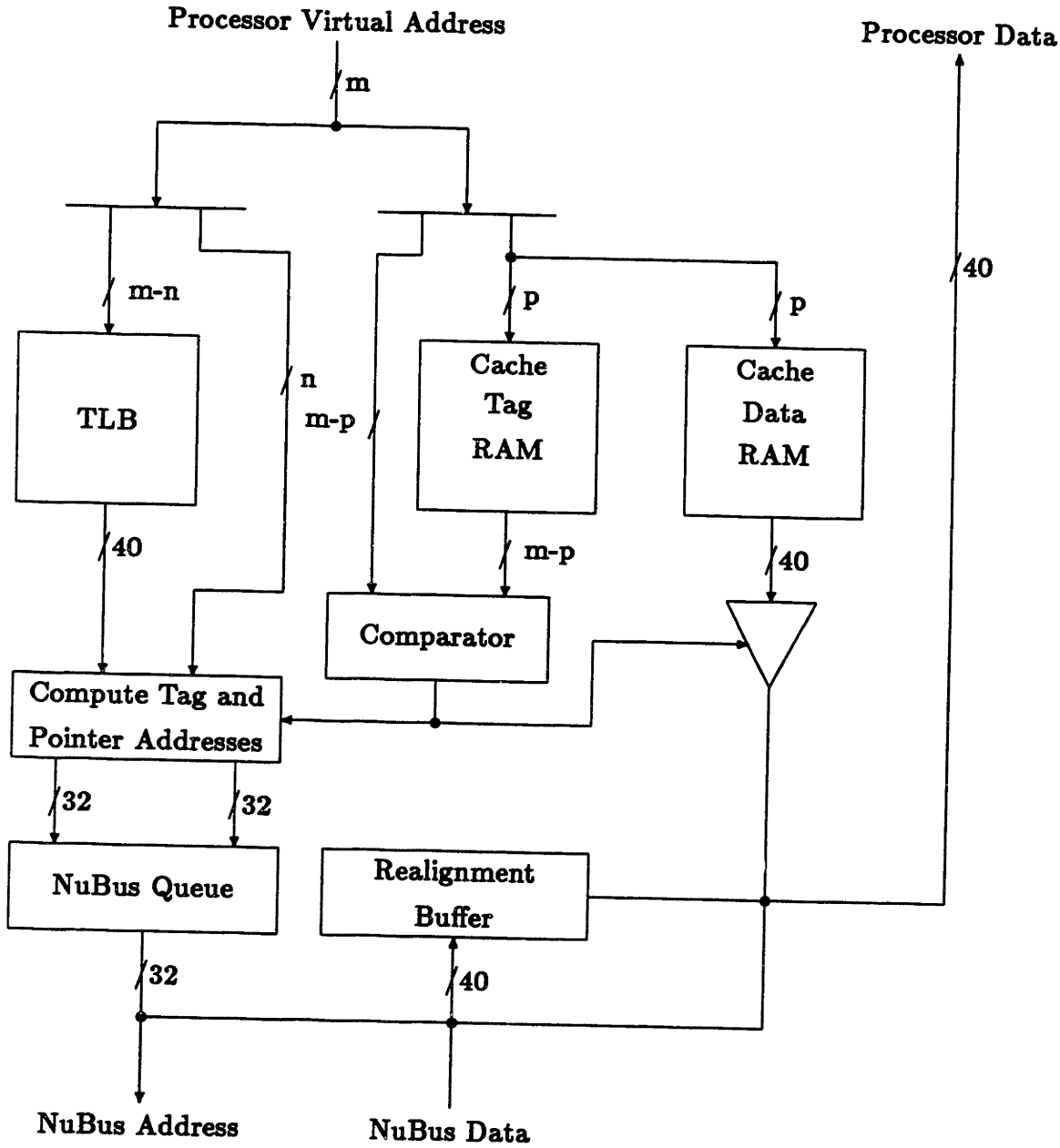


Figure 4.10: Organization of the Separate Tag Virtual Memory Hardware

Chapter 5

Example Implementations of the Two Methods

This chapter describes hypothetical implementations of the two methods on a Texas Instruments Explorer Lisp Machine. The Explorer is a NuBus based, 32-bit architecture. The performance of the Explorer will be simulated in three different configurations. The first will be the normal Explorer memory configuration with simulated caches, and the other two will be simulations of the two methods implemented on the Explorer. The following sections will describe the virtual architecture of the Explorer, the simulation techniques, an implementation of each method on the Explorer, and derive some performance figures for the methods using the simulation results and the equations developed in earlier chapters.

5.1 The TI Explorer

The TI Explorer is a Lisp-based, single-user machine which is a descendant of the MIT CADR. The Explorer has a fixed word size of 32 bits which are separated into a 7-bit tag and a 25-bit pointer field. The tagged architecture is one of many features which is designed to enhance the performance of the Lisp programming language on the machine. The Explorer processor resides in a multi-slot NuBus chassis, and is implemented in doublesided, triple-high Eurocard format. Other

optional NuBus boards include an Ethernet interface, 1-, 2-, 4-, and 8-megabyte memory boards, a system interface for terminal and keyboard control, and peripheral interface which supports the SCSI (Small Computer Systems Interface) bus standard. The SCSI bus is used by large volume secondary storage units such as Winchester disks and mag tape drives.

5.1.1 The Explorer Virtual Machine Implementation

The Explorer processor is a microcoded to implement a virtual machine which is designed to support the Lisp programming language. Lisp is compiled into macrocode, an intermediate langauge which is interpreted by the microcoded processor. Many macrocode, microcode, and hardware features are included to enhance the performance of common Lisp operations on the machine. This section will only discuss the portions of the Explorer which are related to the performance simulation and the implementation of the two methods. If any more information is required, the machine is examined in depth in [TI85] and [TI84].

Interrrupts are at the lowest level of the Explorer virtual architecture. Since they fall below page faults in the system organization, they must be implemented in wired memory – virtual memory with is never paged, or through unmapped accesses using physical addresses.

Above interrupts in the virtual architecture is the virtual memory system. Anything higher than this in the architecture is free to use virtual addresses. The implementation of the virtual memory system is discussed in detail in the next few paragraphs since it is important in understanding of how the methods are implemented on the hypothetical 40-bit Explorer.

Memory accesses are initiated by the processor by writing to one of two registers: VMA (virtual memory address), or MD (memory data). These registers are represented as a number of functional sources, each representing a different type of memory access. These include mapped and unmapped accesses, reads, writes, and fetches.

The Explorer uses a number of tables to store virtual memory information.

The two-level memory map, or TLB stores the data of the virtual pages which are in physical memory for convenient access by the processor. Other data necessary for virtual memory management is stored in the page hash table (PHT), physical page data table (PPD), and disk page map table (DPMT).

Macrocode instructions are 16-bits wide, and are stored two per word. Since macrocode handling is above the virtual memory system in the virtual architecture, macrocode words are in virtual memory and their fetches are monitored explicitly in microcode to check for page faults. In this manner, the simulator is able to count macroinstruction fetches separately from data reads and writes.

Many of the statistics shown in this paper rely on the accessing characteristics of the Explorer, i.e. the portion of the accesses which are reads, writes, or instruction fetches. These figures certainly vary over time, but the following values are representative: reads – 54%, writes – 7%, and instruction fetches – 39%.

5.1.2 The Explorer Virtual Machine Model

The Explorer virtual architecture model which is used in the performance estimates of this paper does not exactly emulate the Explorer. This was done intentionally for four reasons. First, due to some tradeoffs made at design time, the Explorer does not communicate with its main memory board over the NuBus. Instead it used as faster local bus over which it is the sole master. In keeping with the NuBus philosophy and future Explorer family designs, the local bus is not included in the model. All inter-board communications will be assumed to take place over the NuBus.

Second, though fully capable of supporting one, the Explorer lacks a local cache, a crucial part of the designs presented in this paper. All cache performance numbers used in this paper have been generated by simulation.

Third, non-NuBus delay or access times do not exactly match those of the Explorer. Delay times of currently available parts will be used to reflect the performance of these designs as if they were being built at the time of the writing of this paper.

Finally, the Explorer uses a 2-level TLB, but access times of a single level TLB will be used in this machine model. The two level memory map was selected at design time because of its efficiency of storage, and because no cache exists on the current Explorer implementation. If the memory map is considerably slower than the cache, then the machine may be slowed down by the TLB, even during periods of cache hits. It is desirable for the TLB and cache to have comparable access times, and a TLB has been selected here which better complements the caches which have been simulated.

5.1.3 The 40-bit Explorer Virtual Machine Model

The Div3 and Separate Tag methods will be simulated as they would be implemented on an Explorer. Since there are no 40-bit versions of the Explorer, the virtual architecture model which has been selected was chosen for its simplicitly and its ability to be accurately simulated using results from the 32-bit Explorer.

In the 40-bit Explorer model, all bwords will be divided into a 32-bit pointer and an 8-bit tag. This means that the virtual memory space has been extended from 25 to 32 bits since the pointer contains virtual addresses. Macrocode instructions will still be stored two per word, their actual length being undetermined. All virtual memory tables wil be expanded to 40-bits wide. The larger Explorer will probably still communicate with most other devices in 32- or 16-bit words, but thesef all below the level of the virtual memory system, so are of no consequencce here.

It will be assumed that in this configuration, the memory accessing characteristics of this machine will be identical to those of the 32-bit model described above. In this way, the use of the simulation results of the the Explorer as applied to this model will be justified.

5.2 Simulation Techniques

This section describes how the prerformance of the Explorer and the two methods were estimated. The simulator and its use will be discussed, followed by a

description of how the simulation results were evaluated to estimate average access times for the three configurations.

5.2.1 The Simulator

All Explorer performance statistics presented in this chapter were obtained using a microcoded cache and memory system simulator on an Explorer (see [Doug86] for a detailed description of the simulator). The simulator allows runtime performance statistics to be obtained in real time during a natural execution environment. The output of the simulator has cache performance statistics divided into reads, writes, and instruction fetches. For simulating the store-in update policy, the simulator also gives the portion of all accesses which caused a dirty cache block to be flushed.

All figures represent the execution of the Explorer averaged over approximately 27 million memory accesses. They were obtained while running the Boyer benchmark of the Gabriel Benchmark Suite (a short AI-like program) and the compilation of a small program, each representing about half of the 27 million accesses. In each run the machine was configured with one 8-megabyte memory board, and the network and garbage collector both disabled.

A special version of the microcoded cache simulator was built to simulate the Div3 Method. This simulator maintained a copy of the contents of the Div3 ROM in physical memory. This data was accessed exactly as it would be in the Div3 Method, and the cache address is built just as it is described in figure 3.7. In this way, the simulator was able to accurately reproduce the activity of a Div3 cache and obtain information describing its performance.

The raw data from the simulator (read, write, and instruction fetch miss ratios) are included in Appendix A.

5.2.2 Access Time Computation

Performance results of the Explorer and the two methods will be derived from mathematical expressions based on equations 1.1, 3.6, and 4.1. These equations will

be modified to reflect the store-in and write-through update policies for each of the three different machine configurations. The rest of this section will develop each of these expressions and give values which will be used for the time delay constants in them.

All computations in the rest of this paper it will be asumed that no page faults have occurred, so $\mathbf{P}(m_{tlb} \mid m_c)$ will be zero. It is acknowledged that the methods of this paper might alter this value, affecting the average access time values. This effect was ignored for the following reasons. First, after the system had settled down, the programs which were being run for simulation fit easily in the 8-megabyte primary memory which was being used, so page faults were infrequent. Second, there are many different levels of page faults in the Explorer, each requiring very different amounts of time to resolve. Page fault resolution also includes disk wait time, backround writes, and prefetches – figures which are difficult to measure or estimate. Since the raw cache data was known to be accurate, the uncertainty of page fault measurement was seen as a distortion of the final results. Finally, the effects of any differences in paging perfomance are definitely second order effects when compared to the cache and main memory access characteristics.

A FIFO replacement policy was used for set sizes of 2, 4, and 8. Each set stores the number of the set member which was last flushed. The next highest member of the set is chosen next time a cache flush is necessary. In this method, the replacement information is updated on every flush instead of every access. Note that for a set size of 2 with this replacement policy, one would expect cache performance to be comparable to the same size cache with a set size of 1.

Equation 5.1 shows how NuBus memory access times were computed.

$$t_{mem}(BlockSize) = 200ns + 100ns + 300ns + (BlockSize * 100)ns \qquad (5.1)$$

The $200ns$ delay is for bus arbitration, the $100ns$ delay is the address transfer and setup, and the $300ns$ delay is the memory board access time. The final delay is $100ns$ for the transfer of each word of the block. Table 5.1 shows the values obtained from using equation 5.1. The values used for the Separate Tag Method

32-bit Explorer		Div3 Method		SepTag Method	
Words	t_{mem}	Bwords	t_{mem}	Bwords	t_{mem}
4	1000	3	1000	4	1700
8	1400	6	1400	8	2200
16	2200	12	2200	16	3200

Table 5.1: Memory Access Times over the NuBus

include the arbitration times for both memory accesses, assuming that the processor is operating in a multiprocessor environment, and that, in general, bus parking will not be possible. Using Appendix A, the results of this chapter can be recomputed using different NuBus accessing characteristics.

The Store-In Update Policy

The store-in update policy stores write cache hits in the cache and updates memory ony when that block of the cache gets flushed. Read and instruction fetch accesses of the cache are handled normally. Describing the store-in policy mathematically involves using the diry flush statistic from the simulator. Figure 5.1 shows how equation 1.1 is expanded to calculate the average access time of a system using the store-in update policy. Note that the no-page-fault assumption has been made, so $\mathbf{P}(m_{ub} \mid m_c)$ is zero, and write cache misses are not cached.

The Write-Through Update Policy

The write-through update policy updates main memory on every virtual memory write and only updates the cache if the write produces a cache hit. Read and instruction fetch accesses are handled normally. Using the definitions of figure 5.1, the following expression for the average access time with a write-through cache can be derived.

$$t_{avg} = [\%read * \mathbf{P}(h_{read}) + \%fetch * \mathbf{P}(h_{fetch})] * t_c$$

$$
\begin{aligned}
\mathbf{P}(h_{read}) &= \textit{probability of a cache hit on a data read} = 1 - \mathbf{P}(m_{read}) \\
\mathbf{P}(h_{write}) &= \textit{probability of a cache hit on a data write} = 1 - \mathbf{P}(m_{write}) \\
\mathbf{P}(h_{fetch}) &= \textit{probability of a hit on a instruction fetch} = 1 - \mathbf{P}(m_{fetch}) \\
\%read,\ \%write,\ \%fetch &= \textit{portion of total accesses that were data reads,} \\
&\quad \textit{writes, and instruction fetches} \\
\mathbf{P}(h_c) &= \%read * \mathbf{P}(h_{read}) + \%write * \mathbf{P}(h_{write}) + \%fetch * \mathbf{P}(h_{fetch}) \\
&\quad = 1 - \mathbf{P}(m_c) \\
\mathbf{P}(h_{tlb} \mid m_c) &= 1 - \mathbf{P}(m_{tlb} \mid h_c) = 1 \\
t_c &= \textit{access time of a local cache} \\
t_{mem} &= \textit{access time of block from main memory} \\
t_{tlb} &= \textit{access time of the TLB} \\
t_{word} &= \textit{access time of one word from main memory} \\
t_{avg} &= \textit{average access time of a virtual memory request} \\
\%dirty &= \textit{fraction of all accesses which cause a dirty page to be flushed}
\end{aligned}
$$

$$
\begin{aligned}
t_{avg} = {} & \mathbf{P}(h_c) * t_c + [\%read * \mathbf{P}(m_{read}) + \%fetch * \mathbf{P}(m_{fetch}) + \%dirty] * (t_{mem} + t_{tlb}) \\
& + \%write * \mathbf{P}(m_{write}) * (t_{word} + t_{tlb})
\end{aligned} \tag{5.2}
$$

Figure 5.1: Average Access Time (ns) of the Store-In Policy

$$+[\%read * \mathbf{P}(m_{read}) + \%fetch * \mathbf{P}(m_{fetch})] * (t_{mem} + t_{tlb})$$
$$+\%write * (t_{word} + t_{tlb}) \quad (5.3)$$

Again, the no-page-fault assumption is made and write misses are not cached.

The Explorer

Estimating the average access time of the Explroer can be done using equation 5.2 for store-in and 5.3 for write-through. The following values were used for time delays. The value of t_{tlb} was set at 125ns, indicating a single level memory map implemented in static RAM with additional control logic. The value of t_c will be 100ns, again implemented in static RAM. The appropriate value of t_{mem} can be found in table 5.1, and the value of t_{word} will be 700ns as computed by equation 5.1.

The Div3 Method

The version of the Div3 Method which will be simulated uses an address page size equal to or near the transfer page size. This means that the logic following the TLB will be a single 32-bit full-adder. Simple changes could be made to the following equations to allow for the other cases, but they will not be examined here.

The expression for estimating the average access time of the Div3 Method using the store-in policy is found by combining equations 3.6 and 5.2. This results in the following equation:

$$\begin{aligned} t_{avg}\prime \;=\; & \mathbf{P}(h_c)\prime * t_c\prime \\ & +[\%read * \mathbf{P}(m_{read})\prime + \%fetch * \mathbf{P}(m_{fetch})\prime + \%dirty\prime] * (t_{mem} + t_{tlb}\prime) \\ & +\%write * \mathbf{P}(m_{write}) * (t_{2word} + t_{tlb}\prime) \end{aligned} \quad (5.4)$$

Equations 3.6 and 5.3 are combined to give an expression which estimates the performance of the Div3 Mehod with a write-through update policy:

$$\begin{aligned} t_{avg}\prime \;=\; & [\%read * \mathbf{P}(h_{read})\prime + \%fetch * \mathbf{P}(h_{fetch})\prime] * t_c\prime \\ & +[\%read * \mathbf{P}(m_{read})\prime + \%fetch * \mathbf{P}(m_{fetch})\prime] * (t_{mem} + t_{tlb}\prime) \\ & +\%write * (t_{2word} + t_{tlb}\prime) \end{aligned} \quad (5.5)$$

The following additional time delays are used by the Div3 Method. Since the Div3 Method uses bwords, all single bword accesses require two NuBus accesses – one word and one byte transfer. This is represented by a value of t_{2word} which is 1400ns. The value of $t_{tlb'}$ is broken up into t_{tlb} which remains 125ns and t_{log} which whill be 70ns, representing a 32-bit full-adder. The value of $t_{mem'}$ can be obtained from table 5.1.

The Separate Tag Method

By combining equation 4.1 with equations 5.2 and 5.3, the following equations can be obtained to calculate the performance of the Separate Tag Method with the store-in and write-through update poplicies:

$$\begin{aligned} t_{avg'} &= \mathbf{P}(h_c) * t_c + [\%read * \mathbf{P}(m_{read}) + \%fetch * \mathbf{P}(m_{fetch})] * (t_{mem'} + t_{tlb}) \\ &\quad + \%write * \mathbf{P}(m_{write}) * t_(2word + t_{tlb}) \end{aligned} \tag{5.6}$$

$$\begin{aligned} t_{avg'} &= [\%read * \mathbf{P}(h_{read}) + \%fetch * \mathbf{P}(h_{fetch})] * t_c \\ &\quad + [\%read * \mathbf{P}(m_{read}) + \%fetch * \mathbf{P}(m_{fetch})] * (t_{mem'} + t_{tlb}) \\ &\quad + \%write * (t_{2word} + t_{tlb}) \end{aligned} \tag{5.7}$$

No new time delay values are necessary for the Separate Tag Method. The values for $t_{mem'}$ can be found in table 5.1.

5.3 Results from the 32-bit Explorer

Table 5.2 shows the results of the Explorer performance simulations using the store-in update policy. These were derived from the equations of the last section and the raw data given in Appendix A. Similar results for the write-through policy are given in table 5.3.

5.4 Results from the Div3 Method

Table 5.4 contains the results for the Div3 Method with a store-in update policy,

and table 5.5 shows the results using the write-through policy. Below each average access time is a number which shows the percent increase for this method over the 32-bit Explorer results. The comparisons were made between systems which were using the same total cache size and NuBus block transfer size. For example, an implementation of the Div3 Method with a cache block size of 3 is compare to a 32-bit Explorer cache of block size 4.

5.5 Results from the Separate Tag Method

Tables 5.6 and 5.7 give the results of the Separate Tag Method with the store-in update policy. Tables 5.8 and 5.9 show the results using the write-through update policy. They are shown in the same form as the Div3 Method, except that comparison figures have been given for both the 32-bit Explorer and the Div3 Method.

Set Size	Block Size	Total Cache Size (words)							
		256	512	1K	2K	4K	8K	16K	32K
1	4	393.36	336.04	266.71	231.26	200.14	171.18	158.77	142.17
1	8	471.08	391.97	305.36	246.75	208.62	177.02	155.69	140.49
1	16	655.50	544.15	397.2	314.24	246.87	212.08	170.53	146.69
2	4	375.13	310.83	262.27	225.59	199.65	177.91	159.38	141.62
2	8	423.66	342.23	281.9	239.27	211.24	178.54	163.01	141.54
2	16	571.39	444.91	353.29	293.18	229.00	194.42	168.52	147.83
4	4	350.86	282.00	231.16	201.09	177.65	163.07	148.5	131.16
4	8	404.66	315.81	256.23	208.63	183.34	161.05	146.41	132.51
4	16	545.03	409.12	329.22	247.13	200.97	175.89	151.29	132.3
8	4	351.21	274.26	233.04	203.0	180.87	164.50	145.11	130.73
8	8	404.29	316.15	248.23	211.78	184.61	163.47	146.25	130.47
8	16	541.32	412.81	320.71	240.56	200.15	175.54	152.32	131.85

Table 5.2: Average Access Time of Explorer using Store-In

Set Size	Block Size	Total Cache Size (words)							
		256	512	1K	2K	4K	8K	16K	32K
1	4	396.09	343.17	281.33	249.81	227.0	202.62	191.45	179.84
1	8	469.61	396.43	322.21	264.62	237.85	209.35	191.12	179.69
1	16	637.34	534.71	404.32	328.34	271.75	243.81	206.61	187.94
2	4	380.93	323.19	279.47	244.19	221.28	208.71	192.34	180.95
2	8	424.83	352.49	298.5	258.85	234.38	210.11	198.29	183.23
2	16	558.91	445.93	364.75	306.96	250.42	225.83	204.91	193.79
4	4	358.42	299.82	252.67	227.47	207.82	196.75	185.3	171.94
4	8	407.21	328.85	276.5	235.42	214.14	196.27	185.29	175.02
4	16	533.05	413.71	341.93	269.8	231.25	211.03	191.51	177.1
8	4	358.85	291.75	255.44	230.0	212.07	199.73	184.14	173.14
8	8	407.63	329.88	270.43	240.16	216.35	200.0	187.19	175.06
8	16	529.29	417.63	335.75	266.07	232.19	211.78	193.84	178.63

Table 5.3: Average Access Time of Explorer using Write Through

Set Size	Block Size	Total Cache Size (bwords)							
		256	512	1K	2K	4K	8K	16K	32K
1	3	443.57	391.11	334.06	297.08	251.66	222.96	207.89	185.67
vs. Explorer (%)		12.76	16.389	25.255	28.463	25.74	30.246	30.935	30.596
1	6	531.88	456.46	369.16	314.66	264.25	232.18	206.83	182.78
vs. Explorer (%)		12.90	16.452	20.892	27.522	26.666	31.159	32.849	30.107
1	12	712.18	580.55	452.33	366.17	308.55	238.73	214.35	185.95
vs. Explorer (%)		8.65	6.6905	13.879	16.525	24.985	12.565	25.696	26.765
2	3	430.61	375.50	320.69	285.04	257.75	225.54	209.03	186.85
vs. Explorer (%)		28.426	33.771	35.529	40.84	44.707	45.425	51.27	51.765
2	6	482.08	415.33	352.66	307.19	267.1	229.87	204.45	182.88
vs. Explorer (%)		24.132	32.187	37.166	41.606	41.384	47.272	45.048	49.030
2	12	620.64	521.14	412.3	342.0	286.75	243.12	215.17	186.91
vs. Explorer (%)		16.716	26.396	27.775	29.983	39.241	42.160	45.643	44.753
4	3	371.88	328.18	291.56	258.75	238.39	215.13	193.44	176.97
vs. Explorer (%)		5.99	16.376	26.134	28.676	34.193	31.922	30.268	34.927
4	6	418.64	361.16	320.31	265.14	236.86	217.57	193.15	172.86
vs. Explorer (%)		3.45	14.362	25.007	27.085	29.192	35.093	31.925	30.453
4	12	532.35	433.20	373.87	303.38	250.51	220.98	198.81	172.97
vs. Explorer (%)		-2.33	5.8874	13.565	22.76	24.649	25.635	31.409	30.744
8	3	367.52	314.32	276.01	259.37	240.84	213.19	187.44	172.38
vs. Explorer (%)		4.65	14.607	18.440	27.77	33.156	29.595	29.174	31.857
8	6	416.10	339.31	300.68	265.37	232.35	215.23	187.99	168.46
vs. Explorer (%)		2.92	7.3254	21.131	25.307	25.857	31.660	28.539	29.114
8	12	515.10	412.28	331.87	289.96	248.82	219.97	193.09	169.44
vs. Explorer (%)		-4.84	-0.13	3.4803	20.534	24.314	25.313	26.768	28.510

Table 5.4: Average Access Time (ns) of the Div3 Method using Store-In

Set Size	Block Size	Total Cache Size (bwords)							
		256	512	1K	2K	4K	8K	16K	32K
1	3	495.32	445.93	390.55	354.7	322.04	298.4	285.31	267.38
vs. Explorer (%)		25.05	29.946	38.819	41.987	41.869	47.266	49.023	48.675
1	6	574.44	503.25	420.92	369.41	335.54	307.16	286.15	267.19
vs. Explorer (%)		22.32	26.944	30.635	39.604	41.073	46.722	49.727	48.696
1	12	738.94	615.94	499.70	418.33	376.72	317.5	297.72	274.2
vs. Explorer (%)		21.94	15.193	23.589	27.405	38.629	30.225	44.095	45.893
2	3	489.21	432.34	378.76	343.92	320.21	303.51	290.96	274.62
vs. Explorer (%)		28.426	33.771	35.529	40.84	44.707	45.425	51.27	51.765
2	6	527.35	465.95	409.44	366.55	331.38	309.44	287.61	273.07
vs. Explorer (%)		24.132	32.187	37.166	41.606	41.384	47.272	45.048	49.030
2	12	652.33	563.64	466.05	398.99	348.69	321.04	298.43	280.52
vs. Explorer (%)		16.716	26.396	27.775	29.983	39.241	42.160	45.643	44.753
4	3	428.88	392.3	357.00	330.91	313.34	294.74	277.38	265.16
vs. Explorer (%)		19.66	30.845	41.29	45.476	50.771	49.803	49.697	54.216
4	6	468.76	419.87	390.32	336.52	311.05	298.46	277.32	264.01
vs. Explorer (%)		15.12	27.679	41.167	42.944	45.251	52.069	49.671	50.845
4	12	569.96	485.79	432.79	371.4	326.72	304.44	285.33	268.02
vs. Explorer (%)		6.92	17.423	26.572	37.658	41.286	44.268	48.986	51.342
8	3	430.00	384.88	348.72	333.22	318.87	297.88	275.28	265.0
vs. Explorer (%)		19.83	31.92	36.518	44.880	50.358	49.135	49.495	53.057
8	6	471.10	403.74	375.64	337.77	310.14	300.11	277.61	262.66
vs. Explorer (%)		15.57	22.39	38.902	40.641	43.354	50.058	48.306	50.04
8	12	559.53	469.49	401.29	363.76	327.40	306.15	284.78	268.34
vs. Explorer (%)		5.71	12.419	19.520	36.715	41.009	44.557	46.918	50.222

Table 5.5: Average Access Time (ns) of the Div3 Method using Write-Through

Set Size	Block Size	Total Cache Size (bwords)							
		256	512	1K	2K	4K	8K	16K	32K
1	4	618.11	522.72	407.77	348.66	295.76	247.13	226.30	198.52
vs. Explorer (%)		57.14	55.552	52.892	50.766	47.777	44.365	42.532	39.631
vs. Div3 (%)		39.35	33.648	22.064	17.361	17.526	10.841	8.8571	6.9181
1	8	707.96	584.52	451.18	359.50	297.95	248.6	215.65	191.72
vs. Explorer (%)		50.28	49.123	47.753	45.696	42.823	40.432	38.511	36.470
vs. Div3 (%)		33.11	28.056	22.219	14.251	12.756	7.0698	4.2626	4.8908
1	16	936.57	774.81	562.16	442.22	343.04	290.85	230.72	195.90
vs. Explorer (%)		42.88	42.39	41.531	40.728	38.955	37.139	35.297	33.549
vs. Div3 (%)		31.51	33.460	24.282	20.770	11.178	21.831	7.6382	5.3515
2	4	588.44	481.77	400.40	339.34	296.06	258.35	227.21	197.79
vs. Explorer (%)		56.865	54.995	52.667	50.427	48.291	45.212	42.554	39.667
vs. Div3 (%)		36.653	28.301	24.856	19.052	14.864	14.546	8.6966	5.8545
2	8	635.26	508.32	414.83	348.23	303.93	251.08	226.98	193.33
vs. Explorer (%)		49.946	48.531	47.156	45.539	43.879	40.627	39.246	36.589
vs. Div3 (%)		31.774	22.388	17.629	13.359	13.791	9.2262	11.022	5.7135
2	16	816.35	632.99	499.64	411.38	318.63	265.69	227.62	197.87
vs. Explorer (%)		42.872	42.274	41.426	40.316	39.138	36.656	35.065	33.854
vs. Div3 (%)		31.534	21.463	21.186	20.289	11.117	9.2841	5.7856	5.8666

Table 5.6: Average Access Time (ns) of the Separate Tag Method: Store-In

Set Size	Block Size	Total Cache Size (bwords)							
		256	512	1K	2K	4K	8K	16K	32K
4	4	547.51	433.63	348.02	297.06	258.03	233.39	208.92	180.07
vs. Explorer (%)		56.05	53.770	50.557	47.728	45.249	43.121	40.691	37.292
vs. Div3 (%)		47.23	32.132	19.363	14.806	8.2391	8.4887	8.0010	1.7526
4	8	604.62	466.25	373.57	298.88	258.97	223.73	200.89	179.05
vs. Explorer (%)		49.41	47.637	45.796	43.256	41.249	38.915	37.212	35.123
vs. Div3 (%)		44.43	29.097	16.630	12.725	9.3327	2.8292	4.0076	3.5802
4	16	776.85	580.00	463.08	344.1	275.95	238.61	202.84	174.87
vs. Explorer (%)		42.53	41.769	40.661	39.237	37.310	35.662	34.074	32.18
vs. Div3 (%)		45.93	33.886	23.86	13.423	10.157	7.9809	2.0277	1.0985
8	4	548.09	419.85	350.91	300.21	263.27	235.6	203.16	179.22
vs. Explorer (%)		56.06	53.085	50.581	47.887	45.557	43.220	40.004	37.086
vs. Div3 (%)		49.13	33.574	27.137	15.745	9.3139	10.513	8.3843	3.9661
8	8	604.03	466.74	360.82	303.25	260.87	227.31	200.39	175.65
vs. Explorer (%)		49.40	47.631	45.356	43.192	41.303	39.053	37.022	34.627
vs. Div3 (%)		45.16	37.554	19.999	14.273	12.272	5.6151	6.5997	4.27
8	16	771.31	585.03	450.69	333.71	274.42	238.22	204.23	173.87
vs. Explorer (%)		42.49	41.717	40.527	38.724	37.102	35.707	34.078	31.874
vs. Div3 (%)		47.74	41.902	35.801	15.091	10.287	8.2947	5.7662	2.6171

Table 5.7: Average Access Time (ns) of the Separate Tag Method: Store-In

Set Size	Block Size	Total Cache Size (bwords)							
		256	512	1K	2K	4K	8K	16K	32K
1	4	631.66	543.88	441.27	389.07	351.22	310.74	292.22	272.90
vs. Explorer (%)		59.48	58.49	56.850	55.746	54.724	53.359	52.632	51.749
vs. Div3 (%)		27.53	21.966	12.989	9.6904	9.0612	4.1376	2.4216	2.0672
1	8	717.69	604.61	491.93	401.12	359.85	315.61	287.75	269.95
vs. Explorer (%)		52.83	52.514	52.673	51.587	51.294	50.758	50.560	50.231
vs. Div3 (%)		24.94	20.143	16.87	8.5833	7.245	2.7508	0.5566	1.0327
1	16	925.84	778.24	590.49	481.26	399.77	359.66	306.23	279.33
vs. Explorer (%)		45.26	45.545	46.045	46.571	47.111	47.521	48.214	48.625
vs. Div3 (%)		25.29	26.349	18.17	15.043	6.1182	13.282	2.8590	1.8722
2	4	607.57	511.8	438.86	380.35	342.27	321.73	294.0	275.5
vs. Explorer (%)		59.496	58.358	57.035	55.762	54.678	54.155	52.851	52.247
vs. Div3 (%)		24.193	18.380	15.868	10.595	6.8899	6.0026	1.0453	0.3178
2	8	650.01	537.67	454.36	393.44	355.71	317.67	299.98	276.62
vs. Explorer (%)		53.005	52.534	52.217	51.993	51.764	51.192	51.284	50.966
vs. Div3 (%)		23.261	15.393	10.973	7.3352	7.3421	2.6611	4.2987	1.2987
2	16	814.73	652.11	535.26	452.26	370.37	335.73	305.01	291.05
vs. Explorer (%)		45.772	46.235	46.748	47.339	47.900	48.665	48.852	50.187
vs. Div3 (%)		24.894	15.696	14.849	13.352	6.2188	4.5759	2.2035	3.7534

Table 5.8: Average Access Time (ns) of the Separate Tag Method: Write-Through

Set Size	Block Size	Total Cache Size (bwords)							
		256	512	1K	2K	4K	8K	16K	32K
4	4	569.67	473.56	394.16	352.48	319.76	301.38	282.44	260.16
vs. Explorer (%)		58.94	57.950	55.997	54.954	53.864	53.178	52.428	51.304
vs. Div3 (%)		32.83	20.715	10.41	6.5158	2.0518	2.2531	1.8241	-1.8884
4	8	621.99	501.06	420.17	356.73	323.95	295.98	279.28	263.49
vs. Explorer (%)		52.74	52.368	51.963	51.528	51.277	50.803	50.728	50.547
vs. Div3 (%)		32.69	19.336	7.6476	6.0053	4.1485	-0.8326	0.7059	-0.1980
4	16	776.72	605.12	501.8	397.91	342.46	313.5	285.41	264.57
vs. Explorer (%)		45.71	46.267	46.754	47.484	48.093	48.558	49.027	49.394
vs. Div3 (%)		36.28	24.564	15.945	7.1382	4.8178	2.9738	0.0279	-1.2868
8	4	570.83	459.4	399.18	356.98	327.23	306.73	280.82	262.59
vs. Explorer (%)		59.07	57.461	56.273	55.211	54.301	53.571	52.503	51.665
vs. Div3 (%)		32.75	19.361	14.471	7.1309	2.6227	2.9739	2.0125	-0.9098
8	8	623.42	503.26	411.4	364.59	327.90	302.27	282.84	264.08
vs. Explorer (%)		52.93	52.558	52.124	51.809	51.563	51.136	51.095	50.847
vs. Div3 (%)		32.33	24.65	9.5187	7.9409	5.7258	0.7183	1.8807	0.5378
8	16	772.09	611.48	493.76	393.15	344.65	315.28	289.49	267.54
vs. Explorer (%)		45.87	46.417	47.059	47.765	48.435	48.870	49.346	49.777
vs. Div3 (%)		37.99	30.242	23.041	8.0820	5.2668	2.9836	1.6524	-0.2962

Table 5.9: Average Access Time (ns) of the Separate Tag Method: Write-Through

Chapter 6

Summary

This chapter will discuss the Div3 and Separate Tag Methods in the context of the results of the last chapter and Appendix A. The first section will discuss the hardware requirements of the two methods, and will be followed by a discussion of their relative performance. The final section will summarize the findings of the paper.

6.1 Hardware

As seen in chapters 3 and 4, the hardware which is necessary to implement the Div3 and Separate Tag Methods is minimal, especially in comparison to the complexity of a modern 32- or 40-bit processor or memory management unit. This chapter will discuss each of the hardware components separately, concentrating on their implementation and cost/performance tradeoffs.

6.1.1 The Div3 Hardware

The hardware components of the Div3 Method include the realignment buffer, the Div3 ROM, the logic following the TLB, and the cache memory with associated control and tag storage. Each of these will be discussed separately.

The Realignment Buffer

In its barest form, the realingment buffer of the Div3 Method consists of one 32-bit register, one 40-bit register, and one 8-bit 4:1 multiplexor. This would probably not be practical to implement using discrete parts, but is feasible in a semi-custom standard cell or gate-array design. The part would require 32 inputs from the NuBus, 40 outputs to the processor, and control and power. Depending on how tightly coupled the processor and bus clocks are, the buffer may have some asynchronous qualities which would requre extra buffering of words to the processor or cache.

It is assumed that buffering and organization of words going from the processor to the NuBus will be handled by the cache and/or processor.

The Div3 ROM

The Div3 ROM can be implemented with either PALs, ROMs, or a combination of adders and shifters. Since it is in the critical path for minimizing the cache access time, the divide-by-3 function should be done as quickly as possible, preferably with only one level of logic. For ROMS and PALs, the other critical parameter is the number of addressable locations in each part. With n bits to be divided by 3, each part should have 2^n locations, since this prevents the necessity of passing carry bits from one part to the next. The number of output bits for each part only determines how many parts are needed to execute the function in parallel. For example, with $n = 9$, there will be 10 output bits. This could be imlemented with three 1K x 4-bit ROMS, each appropriately programmed.

TLB Address Logic

The logic following the TLB is takes a NuBus address from the TLB and combines it with the output of the Div3 ROM. Depending on the transfer page size, the complexity of this logic can range from simply appending the page offset to the page address to doing one or more full 24-bit additions (the top 8 bits are fixed by the NuBus slot number, and it will be assumed that no memory overlaps slot

spaces). If a single level of addition is to be done, the function can be accomplished using off-the-shelf parts. If a multilevel shift-and-add function is to be performed, space and time considerations would suggest the use of a custom or semicustom part. The inputs can be latched in series since the output of the Div3 ROM should be available well before the output of the TLB. For speed, the additions should be highly pipelined.

The Div3 Cache

The complexity of the Div3 Cache will be determined by the update policy, cache set size, and general block organization (see figure 3.7 for choices of organization). It is currently common to implement cache memory and management functions using custom or semicustom designs.

6.1.2 The Separate Tag Method

The hardware required to implement the Separate Tag Method includes the realignment buffer, NuBus request queue, and possibly special hardware for automatic tag generation and address space lookup. Special address translation hardware is not necessary since all addresses are generated by appending the offset to the base page address. The Separate Tag Method can use less custom hardware than the Div3 Method since a more conventional cache design can be used, but the extent of the differences in complexity are highly dependent on the how the methods are implemented.

Realignment Buffer

The realignment buffer of the Separate Tag Method is very similar to that of the Div3 Method with the difference in block size of 4 versus 3 bwords being the only change. The previous discussion of the implementation of the Div3 realignment buffer also applies here.

NuBus Request Queue

Since the control for the request queue will be too complex to implement with discrete logic, it should be implemented with a custom or semi-custom part. Depending on the NuBus controller used for the design, this function may or may not be already implemented.

Optional Hardware

The optional hardware of the Separate Tag Method includes the automatic tag generation unit and the address space lookup memory. Both of these are small, register based, non-critical pieces of hardware, and can be implemented using virtually any technology. Unfortunately, both are complicated enough that a semi-custom part would have to be used in order to conserve slot space.

6.2 Performance

This section will use the results of Chapter 5 and Appendix A to analyze the performance of the two methods. The first part will examine the Explorer results which will be used as a basis for comparison.

6.2.1 The Explorer

The following observations are from the data of Appendix A. They pertain to the raw data used in both the Explorer and Separate Tag average access times.

- The write miss ratio is considerably higher than both the read and fetch miss ratios. The read miss ratio is consistently 3 to 4 times higher than the fetch miss ratio.

- The effect of block size on miss ratio is greater for larger caches. Larger block sizes consistently yield lower miss ratios.

- Larger set sizes also yield lower miss ratios for all sizes, and again the effect is greater for larger caches.

- The gain in performance when the size of the cache is doubled is greater for smaller caches. Examining the data, it would appear that an increase in cache size from 32K to 64K would give a very small perfomance improvement.

The following cache performance characteristics were observed from the average access time calculations of tables 5.2 and 5.3.

- The performance difference between the store-in and write-through update policies was greater for larger caches. As miss ratios drop, the cost of doing extra bus accesses becomes more significant.

- As one would expect, the increase in performace for doubling the total cache size drops off as the cache size grows, ending at around 10% performance increase for enlarging from 16K to 32K words.

- Set sizes of 4 or 8 were superior to those of 1 or 2 for all size caches. The difference, however, decreased for larger caches.

- Block sizes of 4 words yielded better performance for small sizes, although a block size of 8 gave comparable results for large caches.

Note again that the performance results of the Explorer are identical to those of the same system implemented in 40-bits and operating on a 40-bit bus.

6.2.2 The Div3 Method

Examining the raw cache results in Appendix A shows that the Div3 cache behaves nearly identically to the corresponding Explorer cache. In this context, the following points relate to tables 5.4 and 5.5.

- Again, the performance difference between store-in and write-through increased as cache sizes increased, but the performance difference was larger in the Div3 cache than in the Explorer (approximately 50% versus 30% for a 32K cache).

- The increase in performance when the cache size is doubled decreases as the cache size goes up. Again, increasing the performance from 16K to 32K words gives only a 10% increase in performance.

- Set sizes of 4 or 8 were better for all cache sizes except 16K and 32K where the set size appeared to have little effect on performance.

- Block size of 3 was better for smaller caches, but block sizes of 3 and 6 yeilded similar results for 16K and 32K word caches.

- Compared to an Explorer cache using store-in, the Div3 cache using store-in had average access times about 30% greater for large caches, independent of other cache parameters. For 256-, 512-, and 1K-word caches, caches with larger block sizes performed closer to the comparable Explorer cache.

- A large cache using write-through accessed memory 50% slower than a comparable Explorer cache also using write-through. Again, smaller caches performed better in comparison, and relative performance was, again, a function of block size for these smaller caches.

To generalize these conclusions, with a medium to high performance cache, the Div3 Method will operate about 30% slower than an Explorer when using the store-in update policy, and 50% slower when using write-through. From examining the results for the 4K through 32K-word caches, it would appear safe to extrapolate this result to include caches larger than 32K words. For smaller caches, the higher the average access time, the better the Div3 Method does in relation to acomparably equipped Explorer.

6.2.3 The Separate Tag Method

Using the data summarized in the section 6.2.1, the following observations can be made about the results of tables 5.6, 5.7, 5.8 and 5.9.

- The differences between the store-in and write-through update policies were similar to those for the Div3 Method and Explorer. The magnitude of the difference increased with cache size, and was close to that of the Div3 Method.

- The increase in performance decreases with larger caches, but not to the extent of the Explorer or Div3 Method. This can be seen by the steadily decreasing comparison figures.

- Set sizes of 4 and 8 outperformed the other two over all cache sizes, including large ones where set size was not a factor in the Div3 Method.

- A block size of 4 words yielded the best performance for caches 2K words and smaller, but a block size of 8 words dominated in the larger caches.

- Using store-in, the performance of the Separate Tag Method versus the Explorer ranged from 50% to 30% slower, decreasing with increasing cache size. Unlike the Div3 Method, however, the gain in relative performance was not slowing down with increasing cache size.

- Using write-through, the performance of the Separate Tag Method remained 45-50% slower than the Explorer regardless of cache size. This, also, differs from the Div3 Method.

- Using the store-in policy, the Separate Tag Method does far worse (30% slower) than the Div3 Method for small caches, but has comparable performance for large caches. Examining the figures shows that the Separate Tag Method should outperform the Div3 Method for caches larger than those tested.

- With the write-through policy, the Separate Tag Method again has performance comparable to the Div3 Method for large caches, and should outperform it for larger ones.

In summary, the Separate Tag Method performs poorly for small caches, but catches the Div3 Method quickly. If a very large cache is used, especially with the

store-in policy, the Separate Tag Method appears to be approaching the performance of the Explorer.

6.3 Conclusion

While the previous chapters were general and theoretical, this chapter has tried to more practical. In keeping with this tone, it is appropriate to discuss the reasons for implementing a 40-bit processor on a 32-bit bus. The paper will be concluded by discussing the last two sections in light of these reasons.

6.3.1 Why 40 Bits on a 32-bit Bus?

There are several reasons for wanting to implement a 40-bit processor on a 32-bit bus. These are divided into two main categories: the reasons for wanting to build a 40-bit processor, and the reasons for putting it on a 32-bit bus. These will be discussed in the next few paragraphs.

Address Space

Current programming languages and applications are constantly requiring larger and larger virtual address spaces. This is especially true of object oriented systems such as Lisp Machines where dynamic memory allocation and long-lived data objects can use virtual memory alarming rates, especially if programmed inefficiently. Many of these object oriented systems also have the disadvantage (in the sense of virtual memory size) of being implemented using tagged architectures which reduce the number of available address bits. This is the case with the Explorer. By going to a 40-bit word size, an 8-bit tag can be used while still maintaining a 40-bit address space.

Numerical Accuracy

As CAD/CAM tools and other math based programs become commmon computer applications, numerical accuracy and representation of numbers increasing

in importance. A 40-bit machine increases the numerical accuracy, especially for a tagged architecture which is incapable of representing 32-bit constants in a single word. This is also important for a tagged architecture if it is intended as a debugging station for 32-bit machines.

For non-tagged architectures, the 40-bit word is a convenient storage format for the IEEE 80-bit floating point standard.

Hardware Compatibility

Once a 40-bit processor has been designed, it must communicate with a wide variety of other devices, including primary and secondary memory, I/O devices, and communications networks. Traditionally the processor has communicated with these devices via a bus which was using a standard protocol. Examples include general bus architectures such as VME bus, Multibus, or NuBus. Many devices have been designed to communicate specifically one of these buses. Unfortunately for the designer of the 40-bit processor, none of these devices are specified for 40-bit data transfer. This problem is the driving force behind the designs of this paper.

By implementing one of the methods described in this paper, the 40-bit processor can be implemented as part of a NuBus-based system architecture and is free to make use of existing NuBus devises. In this manner, the design and implementation of such a system would be greatly simplified, and the cost of the overall system would be reduced.

6.3.2 Cost/Perfomance Analysis

After considering the 40-bit design possibilities, their implementations, their performance, and the motivation for their existence, the final cost/performance anaysis is very straightforward. It is based upon the following observations:

1. The motivation for building a 40-bit machine is based either on compatility/cost, or on performance.

2. There appears to be a practical limit in relative performance between a 40-bit processor on a 32-bit and 40-bit bus. For the case described in this paper, the performance degradation is at least 30%.

The choice of whether to use one of the designs in this paper is clear.

If the motivation for developing a 40-bit machine is cost and compatibility, then the methods of this paper should be heavily considered. First, the performance degradation can be tolerated if performance is not the primary design factor. Second, the additional cost of implementing one of these methods is small compared to the cost developing a new 40-bit bus, memory, and peripherals. The total development time and expense is greatly reduced.

If the motivation for developing a 40-bit machine is performance, then the methods of this paper are probably not applicable. First, the performance degradation would probably be prohibitive. Second, a high performance design is usually started from the bottom up in order to take advantage of the latest technologies and ideas. In this case, special 40-bit support would probably be developed system wide.

A final use of these methods is as an intermediate step between a 32-bit and a full 40-bit system. One could make a plug-compatible 40-bit processor to fit into a 32-bit NuBus based system using one of these methods. Then the other 40-bit system module and peripherals could be developed while the 40-bit software was being debugged on the processor. The qualities of these systems which makes this possible is the fact that the addressing anomalies are transparent above the virtual machine level, and the relatively low expense of implementing these methods.

Appendix A

Tables A.1, A.2, and A.3 give the read, write, and fetch miss ratios (respectively) for the Explorer. Since the block sizes of these results coincided with those for the Separate Tag Method, this raw data was also used to generate the average access times for that method. Tables A.4, A.5, and A.6 show the raw read, write, and fetch data used to generate the results for the Div3 Method.

Set Size	Block Size	Total Cache Size (words)							
		256	512	1K	2K	4K	8K	16K	32K
1	4	0.3113	0.2475	0.1691	0.1272	0.0967	0.0656	0.0502	0.0337
1	8	0.3047	0.2332	0.1626	0.1081	0.0821	0.0555	0.0357	0.0242
1	16	0.3137	0.2498	0.1646	0.1152	0.0769	0.0627	0.0344	0.0218
2	4	0.3045	0.2291	0.1681	0.1209	0.0890	0.0698	0.0506	0.0327
2	8	0.2659	0.1987	0.1456	0.1046	0.0792	0.054	0.0410	0.0257
2	16	0.2686	0.1969	0.1412	0.1032	0.0643	0.0461	0.0321	0.0214
4	4	0.2592	0.1860	0.1273	0.0946	0.0686	0.0542	0.0396	0.0214
4	8	0.2445	0.1696	0.119	0.0781	0.0572	0.0396	0.0282	0.0182
4	16	0.2455	0.1691	0.1210	0.0746	0.0490	0.0360	0.0225	0.0131
8	4	0.2572	0.1764	0.1302	0.0967	0.0717	0.0560	0.0359	0.021
8	8	0.241	0.1689	0.1109	0.0818	0.0576	0.0415	0.0287	0.0167
8	16	0.2417	0.1705	0.1182	0.0712	0.0468	0.0353	0.0232	0.0131

Table A.1: Read Miss Ratios for the Explorer

Set Size	Block Size	Total Cache Size (words)							
		256	512	1K	2K	4K	8K	16K	32K
1	4	0.4801	0.4525	0.4400	0.4117	0.3221	0.2793	0.2621	0.2408
1	8	0.4831	0.4239	0.3959	0.3755	0.2496	0.224	0.218	0.1971
1	16	0.4661	0.4068	0.3584	0.3336	0.2500	0.1713	0.1567	0.1368
2	4	0.4814	0.4681	0.4206	0.4063	0.3879	0.2762	0.2543	0.2433
2	8	0.4745	0.4278	0.4071	0.3692	0.3229	0.2235	0.2118	0.1866
2	16	0.4632	0.4131	0.3587	0.2964	0.2844	0.1696	0.1435	0.1334
4	4	0.4544	0.4269	0.3752	0.3038	0.2885	0.2553	0.2315	0.2215
4	8	0.4342	0.3750	0.3402	0.2842	0.2438	0.208	0.1932	0.1743
4	16	0.4221	0.3700	0.2999	0.2667	0.2044	0.1589	0.144	0.1208
8	4	0.4466	0.396	0.3548	0.2981	0.2774	0.2404	0.2175	0.2066
8	8	0.4210	0.3643	0.3149	0.2579	0.2357	0.1972	0.1774	0.1592
8	16	0.4046	0.3516	0.286	0.2296	0.1866	0.1589	0.1373	0.1057

Table A.2: Write Miss Ratios for the Explorer

Set Size	Block Size	Total Cache Size (words)							
		256	512	1K	2K	4K	8K	16K	32K
1	4	0.1651	0.1214	0.0758	0.054	0.0395	0.0224	0.0156	0.0101
1	8	0.1376	0.1052	0.0633	0.0425	0.0308	0.0172	0.0113	0.0074
1	16	0.1128	0.0838	0.0540	0.0356	0.0244	0.0118	0.0086	0.0054
2	4	0.1333	0.092	0.0699	0.0457	0.0330	0.0254	0.0158	0.0084
2	8	0.1123	0.0758	0.0507	0.0348	0.0252	0.0185	0.0125	0.0071
2	16	0.091	0.0594	0.0426	0.0277	0.017	0.0122	0.0087	0.0058
4	4	0.1399	0.0858	0.0588	0.0397	0.0279	0.0202	0.0112	0.0046
4	8	0.1077	0.0696	0.0465	0.0295	0.0199	0.0141	0.009	0.0040
4	16	0.0894	0.058	0.0422	0.0243	0.0156	0.0100	0.0067	0.0034
8	4	0.1414	0.0854	0.0580	0.0405	0.0308	0.0218	0.011	0.004
8	8	0.1109	0.0708	0.0446	0.0307	0.0209	0.0157	0.0089	0.0038
8	16	0.0892	0.0594	0.0372	0.0234	0.0153	0.0104	0.0067	0.0034

Table A.3: Fetch Miss Ratios for the Explorer

Set Size	Block Size	Total Cache Size (words)							
		256	512	1K	2K	4K	8K	16K	32K
1	3	0.3105	0.2546	0.1907	0.1475	0.1071	0.0776	0.0607	0.0377
1	6	0.3140	0.2504	0.1752	0.1256	0.0928	0.0670	0.0453	0.0273
1	12	0.318	0.2414	0.1702	0.1158	0.0894	0.0513	0.0379	0.0223
2	3	0.297	0.2437	0.1759	0.1295	0.0981	0.0759	0.0597	0.0386
2	6	0.2626	0.2161	0.1623	0.1188	0.0852	0.0634	0.0436	0.0282
2	12	0.2553	0.208	0.1467	0.1032	0.0696	0.0506	0.0361	0.0228
4	3	0.2287	0.1844	0.1426	0.1115	0.0906	0.0671	0.0463	0.0298
4	6	0.2079	0.1623	0.1326	0.0892	0.0651	0.0531	0.034	0.0201
4	12	0.2068	0.1524	0.1197	0.0824	0.0538	0.0397	0.0279	0.0158
8	3	0.2262	0.1760	0.13	0.1114	0.0931	0.0666	0.0405	0.0252
8	6	0.2092	0.1499	0.1163	0.0886	0.0621	0.0522	0.0312	0.017
8	12	0.1971	0.1434	0.0985	0.0759	0.0527	0.0387	0.0252	0.0147

Table A.4: Read Miss Ratios for the Div3 Cache

Set Size	Block Size	Total Cache Size (words)							
		256	512	1K	2K	4K	8K	16K	32K
1	3	0.4872	0.4576	0.4409	0.4196	0.3334	0.29	0.2775	0.256
1	6	0.4814	0.4373	0.4154	0.4014	0.2911	0.2582	0.2391	0.2197
1	12	0.4666	0.4144	0.3765	0.3557	0.2656	0.2042	0.1794	0.1641
2	3	0.4570	0.4624	0.4506	0.4235	0.4013	0.2872	0.267	0.2414
2	6	0.4623	0.4426	0.4029	0.3758	0.3408	0.2376	0.2194	0.2058
2	12	0.4395	0.4116	0.3655	0.3285	0.3071	0.1973	0.1849	0.1632
4	3	0.4339	0.3888	0.3663	0.3163	0.2906	0.2699	0.2420	0.2269
4	6	0.4196	0.3775	0.3283	0.2969	0.2755	0.2407	0.2237	0.1961
4	12	0.3867	0.3447	0.3061	0.268	0.2142	0.1808	0.1711	0.1496
8	3	0.4024	0.3386	0.3162	0.3122	0.2742	0.247	0.2279	0.2124
8	6	0.4020	0.3473	0.306	0.2913	0.2545	0.2197	0.1969	0.1838
8	12	0.3803	0.3164	0.2509	0.2347	0.2053	0.1777	0.1555	0.1280

Table A.5: Write Miss Ratios for the Div3 Cache

Set Size	Block Size	Total Cache Size (words)							
		256	512	1K	2K	4K	8K	16K	32K
1	3	0.1786	0.1365	0.088	0.0627	0.0445	0.0291	0.0186	0.0115
1	6	0.1438	0.1067	0.0698	0.0483	0.0351	0.0208	0.0145	0.0085
1	12	0.1193	0.0861	0.0554	0.0372	0.0279	0.0145	0.0102	0.0065
2	3	0.1525	0.1068	0.0749	0.0541	0.0409	0.0301	0.0188	0.0101
2	6	0.12	0.083	0.0601	0.0418	0.0286	0.022	0.0138	0.0077
2	12	0.1062	0.0721	0.0473	0.0316	0.02	0.0147	0.0102	0.006
4	3	0.1236	0.0950	0.0719	0.0495	0.0381	0.0270	0.0154	0.0064
4	6	0.1015	0.0797	0.0548	0.0354	0.025	0.018	0.011	0.0048
4	12	0.0782	0.0598	0.0450	0.0273	0.018	0.0125	0.0083	0.0039
8	3	0.1202	0.0825	0.0631	0.0506	0.0412	0.0283	0.0140	0.0047
8	6	0.0999	0.0639	0.0476	0.0353	0.0255	0.0185	0.0113	0.0041
8	12	0.0779	0.0509	0.0378	0.0263	0.0178	0.0130	0.0085	0.0034

Table A.6: Fetch Miss Ratios for the Div3 Cache

Bibliography

[Clar81] Douglas W. Clark, Butler W. Lampson, and Kenneth A. Pier, "The Memory System of a High-Performance Personal Computer", Xerox PARC, Palo Alto, CA., 1981.

[Doug86] David C. Douglas, "Cache Evaluation using a Microcoded Cache Simulator", unpublished paper, MIT, Cambridge, MA., 1986.

[Hwan79] Kai Hwang, *Computer Arithmetic: Principles, Architecture, and Design*, John Wiley & Sons, New York, 1979.

[Smit82] Alan Jay Smith, "Cache Memories", ACM Computing Surveys, Vol. 14, No. 3, September, 1982, pp. 473-530.

[Smit85] Allan Jay Smith, "Cache Evaluation and the Impact of Workload Choice", Proc. 12'th Ann. Symp. on Computer Architecture, June, 1985, Boston, MA, pp. 64-73.

[TI85] *Explorer System Software Design Notes*, Texas Instruments, Inc., Dallas, Tx., 1985.

[TI84] *Explorer Processor*, Texas Instruments, Inc., Dallas, Tx., September, 1984.

[TI83] *NuBus Specification*, Texas Instruements, Inc., Dallas, Tx., 1983.