THE MU6-G VIRTUAL ADDRESS CACHE

Alan Knowles

Department Of Computer Science

The University Of Manchester

Manchester M13 9PL, England

Shreekant Thakkar

Department of Computer Science and Engineering

Oregon Graduate Center

Beaverton, Oregon 97006

Abstract

A virtual address cache, storing both instruction and operands, for a supermini computer is presented. The cache is based on parallel hash hardware and is a part of the virtual memory management unit. This paper describes the design and implementation of the cache and how it forms a part of the memory hierarchy.

This research was supported by Science and Engineering Research Council of United Kingdom.

1. Introduction

MU6-G is a general purpose 64-bit supermini computer designed to fill the gap between the 32-bit minicomputers and the larger mainframes. Like its predecessor, the MU5 [MORR79], its instruction set is designed to provide support for high-level languages. However, unlike the MU5, MU6-G does not make a distinction between primary (*named*) operands which can be reals, integers or descriptors of arrays, and secondary operands which are array elements.

The *Name Store* [IBBE77] on MU5, which is used to hold most recently used named operands, is replaced on MU6-G by a cache holding both the operands and instructions. This is primarily because the performance of the Name Store fell below the expected level during compilation process [HUSB 76], [THAK78].

The cache in the MU6-G processor is located within the *Memory Access Controller* (MAC) [KNOW84]. Its organisation is based on parallel hash hardware [GOTO77] and data in the cache is accessed using virtual addresses. It forms a part of *one-level store* [KILB62] with the rest of memory hierarchy of the system and is invisible to the system software and the user.

2. The MU6-G Computer System

MU6-G is a high performance minicomputer with 32-bit address and integer registers, and 64-bit data path between memory and the 64-bit floating point arithmetic unit. The instruction set has been designed with requirements of both the high-level languages, compilers and the operating system in mind. A full description is given in reference [EDWA80]; a brief outline (figure 1) follows:

Instruction Fetch Unit (IFU) maintains a small prefetch buffer of instructions from memory via MAC which performs virtual to real address translation and, if the item is not in its cache, it accesses the *Local Store* (LS) via the *Local Store Interface* (LSI) which also provides the route to the outside world via the *System Bus* (SB). Instructions are passed from IFU to *Operand Fetch Unit*

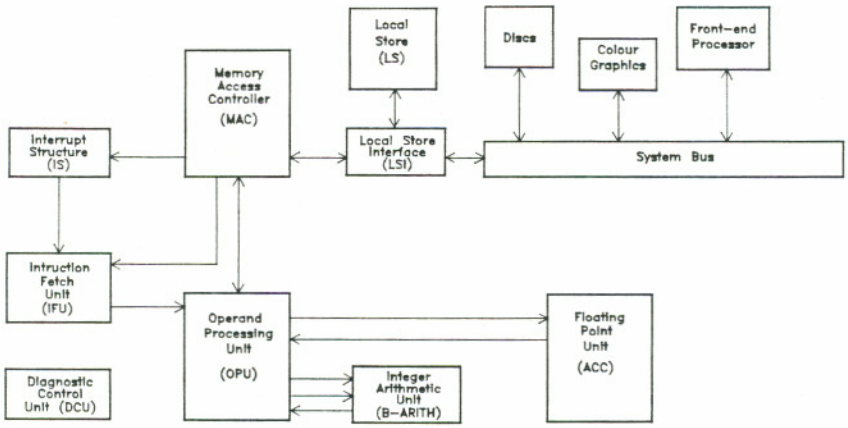

Figure 1: MU6-G System

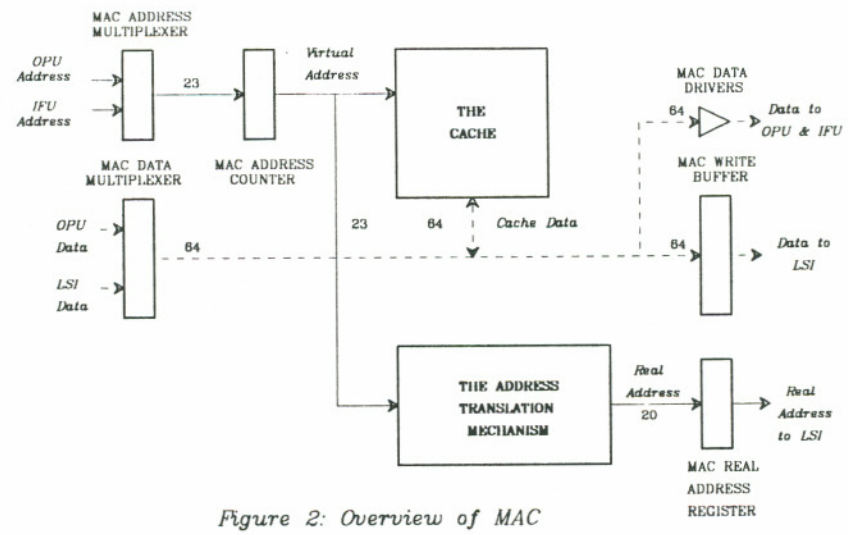

Figure 2: Overview of MAC

(OPU) where they are decoded, operand accesses made via MAC if required, and, for most orders, appropriate functions are passed on to either the *Integer Arithmetic Unit* (B-Arith) or the *Floating Point Arithmetic Unit* (ACC). The *Interrupt Structure* (IS) deals with interrupts of all types, generating appropriate identifiers and initiating action in both OPU and IFU. The *Diagnostic Control Unit* (DCU) has read/write access to all flip/flops in the MU6-G processor and, as implied by its name, is used to diagnose faults. Its use during normal operation is to initialise control flip/flops on start-up, to generate the system clock and to act as the system console.

3. The MAC

In addition to performing address translation and providing high speed buffer storage for operands and instructions MAC also performs the virtual memory management functions. It handles communication between the operating system and the machine hardware.

Unlike most memory systems architectures, the virtual memory management unit combines the high speed access to data and the management of the virtual store. These functions are performed simultaneously, and are independent of each other. Operations which require manipulation of the memory mapping registers are provided and these keep the software action required to manage the virtual store to a minimum.

A full description of MAC is given in the reference [THAK82]; a brief outline (figure 2) follows:

The virtual address from OPU or IFU is selected by MAC's priority arbitration logic and the selected address is then concatenated with the PROCESS (ID) number from a control register to form a system virtual address. This address is then presented to both the cache and the address translation mechanism simultaneously. In the event of a cache hit the appropriate 64-bit value will be sent to the requesting unit, otherwise a miss is indicated to the microinstruction sequencer which is a part of MAC control unit (not shown in the figure). The address translation mechanism performs translation from virtual page number to real page number. This is then concatenated

with the untranslated line number to form the real address. In the event of a cache hit and the address translation being successful the statistics information within the memory mapping registers is updated and MAC becomes ready to accept the next request. In the event of a cache miss and address translation being successful a request to the Local Store is made via the Local Store Interface. If it was a read request then the requesting unit will be sent data from Local Store when it is available otherwise the data to be written is loaded into the write buffer and MAC becomes free again. When the address translation is not successful then an interrupt is generated by MAC and the operating system will have to take the necessary actions.

An operation mode control register within MAC can switch on/off the cache and the address translation mechanism. This can only be done in *executive (supervisor)* mode by using the *V-Store* mechanism. The V-Store also enables the system to purge the cache selectively or completely and allows the manipulation of memory mapping registers.

4. The Cache

4.1 The Design Rationale

A study of cache organisations [THAK78] alternative to that of MU5 Name Store was done with trace driven simulations. These revealed that an organisation based on parallel hash hardware, storing both the operands and instructions (a *unified cache*) and using virtual addresses had a hit-rate in excess of 95% during the compilation and execution process for selected benchmarks. The benchmark programs were selected from a suite of thirty Algol and Fortran benchmarks after a preliminary simulation study. These were a Fortran Compiler (FTN-COMP) and Ackerman's function. These programs were selected because they consistently exhibited very high miss-rate on all of the different cache organisations simulated.

Ideally the cache design should be based on fully associative organisation such as the MU5 Name Store since this organisation is least sensitive to mapping [RAO78] and performs very well when a rea-

sonably sized cache is implemented [HUSB76], [THAK78]. It also has additional advantages, such as the ease of performing masked searches, over other organisations. The main reason this organisation has not been widely used is the impracticability of implementation of a reasonably sized cache. For example, to implement a 512 word fully associative cache with a 32-bit tag field using the available *content addressable memory* (CAM) (4 word x 4 bit) would require 1000 CAM chips. This does not include the random access memory required to store the data associated with the tags. Therefore alternative solutions are adopted which do not require the use of content addressable memory. Hence a study was performed to evaluate the performance of parallel hash hardware as an alternative cache organisation.

The parallel hash hardware [GOTO77] consists of three major components, *hash address generator* (HAG), a *hash table* and a *controller* (figure 3). When presented with a search word (item), HAG generates hash addresses which are used in the interrogation of the hash table.

The hash table is organised as J parallel banks of M words. Entries from each bank are accessed in parallel using the hash address. A comparator is associated with each bank for performing comparison between a given *key* K and the accessed key K_{ij}, ($1 \leq j \leq J$, $1 \leq i \leq M$).

Each entry in the hash table has an *empty* bit associated with it which indicates whether the entry is empty or occupied. *Collisions (conflicts)* are handled using an *open addressing* mechanism; that is, when a clash occurs a second access or probe is made to the hash table, using another hash address. This address is either computed using one of the probing techniques [MAUR68] or rehashed using a different hashing algorithm.

The table is searched until an empty location is found or a match occurs or all the entries have been examined. A *conflict counter* associated with each line of the hash table is used for resolving the next empty location during an *insert* operation (adding a key) and the end of search during *delete* operation (removing a key). The counter is incremented for every probed entry which is not found empty during an insert operation, and decremented for every probe entry with matching key during a delete opera-

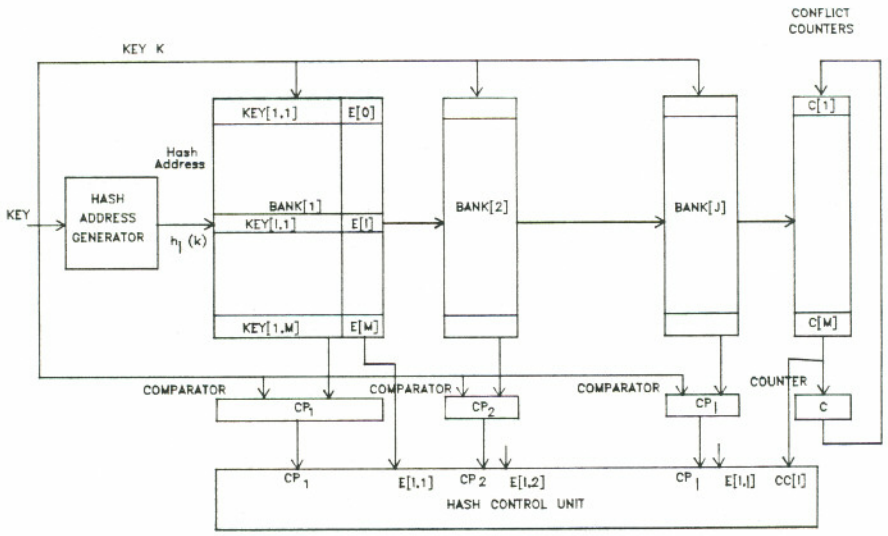

Figure 3: Organisation of Parallel Hash Hardware

tion. In this way, the *search* operation can resolve whether to proceed with the search for a given key.

The parallel hash hardware has to be modified when used as a cache because the luxury of going through a chain list is too expensive if the first access is unsuccessful. In a hash table of J banks, the first entry in a line can be thought of as the primary location for the data when it is accessed by an hash address, and the entries in the rest of the line can be considered as secondary locations (*i.e.*, a *bucket* of J-1 entries). Hence the primary location and all the possible secondary locations are accessed simultaneously. If the search is unsuccessful then one of the entries in the line has to be replaced. In the general case a FIFO replacement strategy is used per line and this points to the next location (*i.e.*, the bank) into which a new block can be loaded. Hence the parallel hash table looks similar to the *Translation Lookaside Buffer* (TLB) organisation found on IBM mainframes [MATI77].

Each bank of the hash table could be accessed using a hash address generated by a different hashing algorithm. However this feature is not used since it would complicate the implementation of replacement strategies and control of the hashing hardware.

When making the choice of hashing algorithms for this study, the complex algorithms were not considered because of the necessity to generate the hash address very rapidly. Hence, three simple hashing algorithms were chosen for investigation. The study showed that the algorithm using *hash-bit extraction* had better performance than those involving logical and folding operations.

A larger but an easier to implement cache based on parallel hash hardware gave equivalent performance in simulation [THAK78] to a reasonable size fully associative cache. Hence, a decision was made to use this organisation to implement the cache for MU6-G.

If the cache were split into two parts, one for instructions and one for operands, it would create several problems [SMIT82]. First is the problem of consistency. There may be two copies of information, one in both caches, specifically in the case of self modifying code. Second, data and instructions may coexist in the same word for reasons of storage efficiency or because of operands accessed relative

to the program counter.

The simulation study showed that there was little difference in the performance of a unified cache and a split cache for instruction and operands. This analysis did not take account of any performance increase which could result from having split caches nearer to an appropriate functional unit such as the Instruction Fetch Unit or the Operand Processing Unit. This is mainly due to the fact the Instruction Fetch Unit and the Operand Processing Unit may be competing for the access to the cache. This is avoided on MU6-G as described below.

The instructions on MU6-G have two formats:

a *short* 16-bit format for majority of orders operating on small literal values, register contents, or names local to current procedures.

a *long* 32-bit format required to exploit more sophisticated operand accessing methods.

The IFU has two 64-bit buffers each of which can hold a mixture of short and long instructions. IFU always prefetches a buffer while it is decoding instructions from another buffer. Fetching more than one instruction from the MAC also ensures that the IFU is not stealing cycles required to process operand requests from OPU.

During the specification of MU6-G it was observed that when operating at its peak rate, MAC could be required to service five requests per microsecond with the ratio of OPU to IFU requests being 4:1. IFU requests are fetched instructions but OPU requests are for operands actually needed. Therefore, OPU requests to MAC are given priority over the IFU requests. However, OPU can give advance notification of an IFU request to MAC when a control transfer instruction is encountered, enabling the MAC to switch its priority logic to accept the IFU request and thus eliminate any arbitration delay.

Hence a decision was made that a unified cache is more appropriate for the MU6-G architecture.

Most cache designs have been based on using a real address to access the cache. This means that the virtual address translation must be done first. However if this stage can be eliminated then the cache access time would be significantly reduced. The MU5 Name Store eliminated this stage and had a fully associative cache for named operands accessed using a virtual address. Hence the study of alternative cache organisations was carried out with a virtual address cache as the prime objective.

A virtual address cache requires that the *tag* of each block of data to have a PROCESS number or an identifier as a part of the address field; otherwise, the cache has to be purged every time task switching occurs. This is not a problem and the only cost is extra memory. Fast address translation is still necessary since the real address will be required to access the Local Store in the event of a cache miss; but, this activity can be done in parallel. Also virtual memory housekeeping requires updating of *dirty* bit in the translation tables to signal that, on rejection, the page needs writing back to backing store.

The other problem faced by the designers of a virtual address cache is writeable *synonyms,* where two or more virtual address can map to the same real address. Synonyms occur whenever two address spaces share cache data. However on MU6-G sharing is not permitted between user address spaces. The only sharing that is permitted is for system address space. This allows sharing of compilers, editors, libraries and other system utilities and is done via a *common segment* feature. If the most significant bit of the SEGMENT field in the virtual address (figure 6) is set then MAC detects this to be a common segment and masks out the PROCESS number from that virtual address.

The cache also stores the access permission bits and hence the access check is performed every time the cache is accessed.

The results of the simulation study showed that a unified cache of two banks with 2048 words each had a hit-rate well in excess of 95%, for a large single job, both during the compilation and execution phases. Hence a unified virtual address cache based on parallel hashing hardware was used for MU6-G.

However, multiprogramming is bound to degrade this performance because no account is taken of PROCESS number in bit selection of the hash address (figure 6); but to what extent is uncertain. Features described later allow experiments to be performed in order to reduce this degradation. An earlier study [STRE76] has shown that the system software plays a critical role in determining the performance of the cache under conditions prevailing in a multiprogramming environment. Specifically if process switching occurs infrequently, then it has very little impact on the cache performance and vice versa. This is true whether the cache is based on virtual or real address space. If the number of processes is small and switched frequently then the performance of the cache might be even better because the data belonging to that process may still be present in the cache.

4.2 The Design

The cache is organised as two parallel banks of 1024 blocks (figure 4). Each block holds two 64-bit words. With each block of data is a 23-bit key, a valid bit, 4 access bits and 2 information bits (figure 5).

The *association* is performed by indexing the two banks with the 10 least significant bits from the virtual address (figure 6), and comparing the keys with the 23 most significant bits of the virtual address. A comparator is associated with each bank for this purpose. The 10-bit *hash address* or *index* can be altered if the hashing function proves to be unsatisfactory in a multiprogramming environment. The 3-bit overlap of index and key field (figure 6) allow future variation of the hashing function.

A cache hit occurs if one of the comparisons gives equivalence. However when neither of the keys read out of the banks match the key field of the virtual address, a cache miss is said to have occurred.

Which of the four 64-bit data words from the banks is selected is determined by the bank which gave equivalence (bank select) and by the least significant bit of the virtual address (word select).

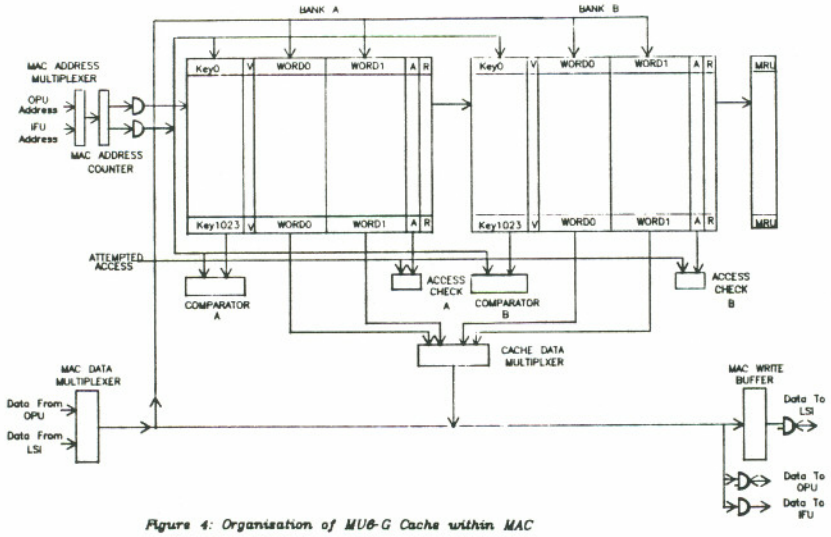

Figure 4: Organisation of MU6-G Cache within MAC

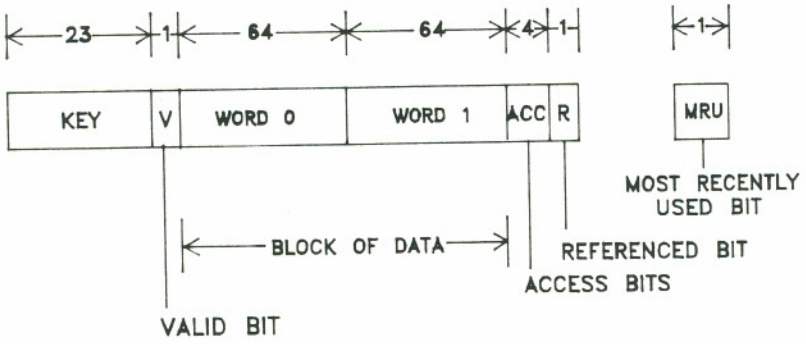

Figure 5: A Line in a Cache Bank

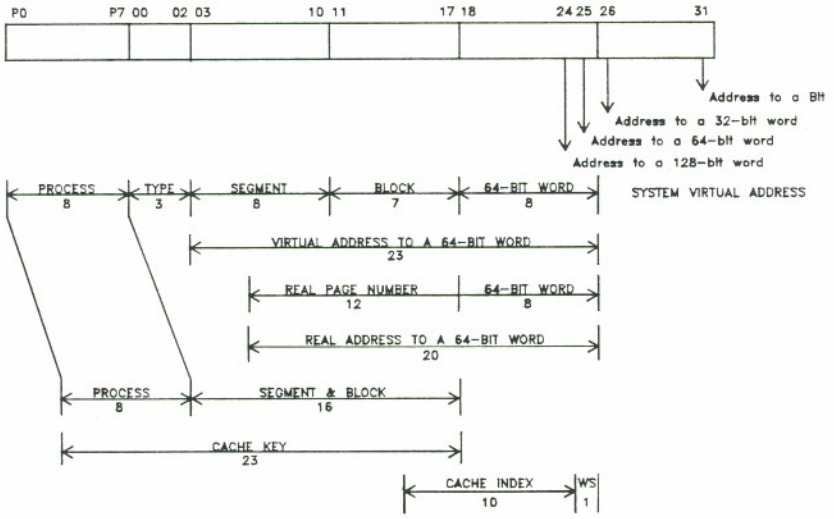

Figure 6: Address Partitioning & Cache Addressing in MU6-G

The valid bit indicates whether the block accessed contains valid data. This bit, initially cleared, is set when the block is loaded from the Local Store after a cache miss. The cache is purged by clearing all the valid bits after a cold start and whenever the memory mapping registers are manipulated by the system software.

The 4 access bits stored with each block determine the use to which the data may be put. These include *write, read* and *obey* permission, in addition to whether the data is for use exclusively by procedures running in the *executive* mode.

The information bits are the *Referenced bit* (R), used by the *prefetching* strategy and the *Most Recently Used bit* (MRU), used by the *replacement* strategy.

A replacement strategy is needed in order to determine into which of the two banks a new block is to be placed after a cache miss. The strategy chosen is to overwrite the least recently used block. For each line of the cache the bank which is most recently accessed is indicated by the MRU bit which is shared between two corresponding banks. Had there been more banks this selection would have been far more complex but with two the solution is simple and easy to implement. This strategy was shown, during the simulation [THAK78], to give a 2:1 improvement in miss-rate over using one bank, for the same total size.

An optional prefetching strategy is provided, to allow a study of its advantages and overheads. This is similar to the *tagged-prefetch* strategy described by Gindle [GIND77], and its objective is to ensure that the block succeeding the one being referenced is always present in the cache. Smith [SMIT78] has indicated that a properly implemented prefetching strategy would reduce miss rate by at least half and improve the CPU performance by 10 to 15 percent. Against this must be weighed the increase in store traffic interfering with external store activity, and keeping MAC busy for longer than strictly necessary, thus possibly impeding the processing of subsequent requests. Experiments will be performed to evaluate the technique in practice.

A referenced bit is associated with each block, to indicate whether the block has been used or merely been prefetched but not yet accessed. The bit is set when the block is first accessed by the processor and this triggers the prefetch of the succeeding block. However the cache is first examined to check if the succeeding block is already present, before a request to the LSI is made. The referenced bit is reset when the block is loaded into the cache by the prefetch action. The prefetched block will also be signalled as most recently used. Hence it is treated as though it were used at the time of prefetching so that it is then least likely to be replaced.

The correctness of signalling the prefetched block as the most recently used is debatable since as a result of the prefetch operation that the block which is now least recently used was more useful. Hence the setting of the MRU bit for the prefected block can be disabled if necessary.

The prefetch sequence, only entered if the option is selected, examines the referenced bit of the cache block currently accessed, and if set (indicating the prefetching has already been performed) MAC exits from the sequence and accepts another request. Otherwise the referenced bit is first set and then the cache is examined to check if the succeeding block is present, and if not a request is made to LSI. There are three options: non, partial or full prefetch. With partial selected, the prefetch sequence is only entered after a cache miss, while with full, it is entered after every read access.

The cache must also cope with operands not aligned on 64-bit boundaries. Each byte of a 64-bit word can individually be written to. Hence for example a 64-bit operand can use 3 bytes of one 64-bit word and 5 bytes of another. The bytes which do not belong to that operand are not affected. When a 64-bit boundary is crossed by such a request, a double cycle of the cache is performed (including, if appropriate, a write-through to Local Store).

A *write-through* strategy is used for updating the copy of the data in the main store at the same time as in the cache. This policy has been chosen instead of the *write-back* policy primarily because it preserves the integrity of the data in the Local Store at all times. Its other main advantage is easy

implementation. To overcome its main disadvantage [SMIT79], namely additional write accesses to Local Store potentially keeping MAC busy for longer than necessary, a single write buffer has been implemented. The data to be written is buffered together with the corresponding address, freeing MAC to process further requests.

A *load-through* technique is employed in the cache miss sequence to ensure that the data requested from the Local Store is simultaneously available to the requesting unit and the cache.

On every cache miss two 64-bit words (a block) are fetched from the Local Store. In the case of a read request the requesting unit is sent the required word and the block is placed in the cache. In the case of a write request the block from the Local Store is first stored in the cache and then the required word is updated in the cache. The updated word is then written into the write buffer and a write request to Local Store is made. The replacement policy is used when placing a new block in the cache.

Any alteration to the memory address translation registers can mean that some cached data or access permission are no longer valid. Thus relevant entries must be purged from the cache. A fast scheme could be devised which merely invalidates all entries in the cache but this would cripple the hit-rate. The alternative is to search the cache comparing relevant fields of the key (PROCESS, SEGMENT, BLOCK number determined by the action the operating system is trying to take) and invalidating only those entries which match the comparison. This action is performed by microprogram initiated when the operating system commences the action leading to a translation table change. In the case when a block number is included in the search, only one eighth of cache need to be examined because of the 3-bit overlap of the cache index address and key in the block field (figure 6).

The cache operation is partly controlled by hardwired logic and partly by the microprogram sequencer in the MAC. The hardwired logic is basically sequential logic which generates data available to the requesting unit in the event of a normal data request resulting in a cache hit and address translation success. This is the only way that the high throughput can be achieved. The microprogram

sequencer only performs book-keeping duties in this event and, if prefetching is on ensures that the succeeding block is present in the cache. The sequencer also performs search and purge operations besides handling the cache miss event and prefetching. The blending of hardwired and microprogrammed logic ensures throughput and flexibility without increasing cost and complexity.

4.3 The Implementation

The MU6-G processor is implemented using ECL 10K devices. The cache banks are implemented using the 1024 x 4 bit ECL RAM, with an on-chip worst case address access time of 25ns. The banks are so organised that each requires 41 such chips. The 23-bit comparator, for each bank, is made using equivalence gates and 7 SSI packages are required. The selector, for both banks, is implemented using dual 4 to 1 multiplexers.

The cache is made up on two identical 16" x 8" four layer printed circuit boards (figure 7) compatible with the wirewrap board used elsewhere in the machine.

Each of the boards contains the key field of a cache bank, the access bits, the information bits and 32 of the 64-bits of each of the 4 data words. The power dissipation of each board is about 30 watts.

4.4 The Performance

The timing diagram (figure 8) shows the time duration from the instant when a virtual address is presented to the MAC virtual address multiplexer to the instant when data is available to the requesting unit. The data-available signal is generated after 60ns and data is available after 68ns.

5. Conclusion

The design and implementation of a high performance cache is presented. A unified cache is used to hold all the operands and instructions for MU6-G. The design is based on a parallel hashing scheme using a least significant bit map as a trivial hashing algorithm to achieve high performance. The cache

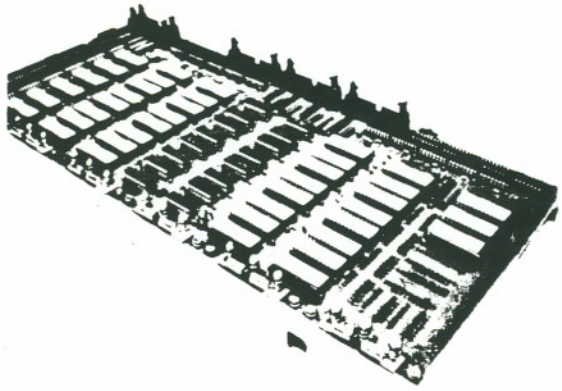

Figure 7: A Cache Board

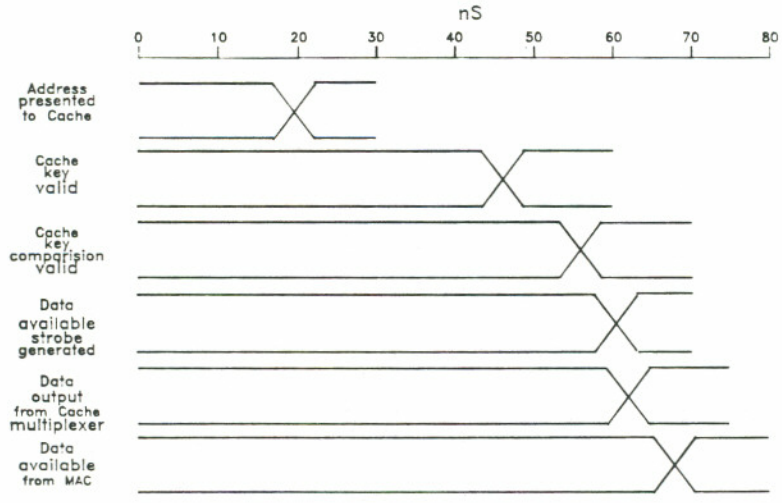

Figure 8: Cache Timing Diagram

uses virtual addresses and can hold entries for up to 256 processes. In simulation studies using benchmark programs a cache of similar configuration and size (*i.e.*, 32K bytes) has been shown to have a hit-rate in excess of 95% for both compilation and execution tasks. Performance degradation due to multiprogramming has been forseen and consideration has been taken into the design to be able to alter the hash-bit extraction algorithm used for generating hash address if necessary.

The cache is entirely controlled by the hardware and is transparent to software. Access checking is also performed in the cache and hence the operation of the cache is totally independent of the address translation and memory management operations.

MU6-G has been fully operational since the fall of 1982 and its architectural features are now being evaluated. However during the commissioning phase the cache and rest of the MAC performed within specification.

References

EDWA80 Edwards, D. B. G., Knowles, A. E., AND Woods, J. V. *MU6-G: A new design to achieve mainframe performance from a mini-sized computer.* In Proc. ACM SIGARCH Symp. on Computer Architecture, SIGARCH Newsletter (ACM) 8, 6 (June 1980).

GIND77 Gindle, J.D. *Buffer Block Prefetching", IBM Technical Disclosure Bulletin,* 20, 2 (July 1977), 696-697.

GOTO77 Goto, E. AND Ida, T. *Performance of a Parallel Hash Hardware with Key Deletion,* IFIP Proc., (1977).

HUSB76 Husband, Y. L. *Operand Buffering in High Speed Computers,* Ph.d Thesis, The University of Manchester, (1976).

IBBE77 Ibbett, R.A., and Husband, M. A. *The MU5 Name Store,* The Computer Journal, 20, 3 (1977), 227-231.

KILB62 Kilburn, T., Edwards, D. B. G., Lanigan, M. J., AND Sumner, F. H. *One-level Storage System,* IEE Transactions Electronic Computers EC-11, 2 (1962).

KNOW84 Knowles, A. E., AND Thakkar, S. S. *Virtual Memory Management Within MU6-G,* To be submitted.

MATI77 Matick, R. E. *Computer Storage Systems and Technology,* John Wiley & Sons, New York, (1977).

MAUR68 Maurer, W. D. *An Improved Hash Code for Scatter Storage,* CACM 11, 1 (Jan. 1968), 35-38.

MORR79 Morris, D., AND Ibbett, R. N. *The MU5 Computer System,* Springer Verlag, New York, (1979).

RAO78 Rao, G. S. *Performance Analysis of Cache Memories,* JACM 25, 3 (July 1978), 378-395.

SMIT78 Smith, A. J. *Sequential Program Prefetching in Memory Hierarchies,* IEEE Computer 11, 12 (December 1978), 7-12.

SMIT79 Smith, A. J. *Characterising the Storage Process and its effect on the Update of Main Memory by Write Through,* JACM 26, 1 (January 1979) 6-27.

SMIT82 Smith, A. J. *Cache Memories,* Computing Surveys, 14, 3 (September 1982), 473-530.

STRE76 Strecker, W. D. *Cache Memory for PDP11 Family Computers,* In Proc. 3rd Annual Symp. on Computer Architecture, (January 1976), ACM, 155-158.

THAK78 Thakkar, S. S. *Investigation of Buffer Store Organisation,* MSc Thesis, The University of Manchester, (1978).

THAK82 Thakkar, S. S. *A High Performance Virtual Memory Management Unit for a Supermini Computer,* Ph.D Thesis, The University Of Manchester, (1982).