A Novel Cache Mechanism

J. A. Gunawardena
Faculty of Engineering
University of Peradeniya, SRI LANKA
+94 (08) 88185

Abstract - **This cache mechanism is transparent but does not contain associative circuits. It does not rely on locality of reference of instructions or data. No redundant instructions or data are encached. Items in the cache are accessed without address arithmetic. A cache miss is detected by the simplest test; compare two bits. These features would result in faster access, higher hit rate, reduced chip area and less power dissipation in comparison with associative systems of similar size.**

1 Introduction

Most computer programs exhibit the property of temporal locality; i.e.,

- Much of the code consists of repetitive loops.
- The same data is accessed several times within a short interval of time.

Often, they also exhibit the property of spatial locality; i.e.,

- Instructions executed in sequence occupy a contiguous area of the main memory.
- Successive instructions access data in a contiguous area of the main memory.

All cache systems rely on temporal locality to meet the basic objective of reducing accesses to the main memory. Conventional cache systems that rely on temporal locality alone must employ fully associative access of the cache. But associative hardware is complex and costly. Hence, most conventional cache systems depend on the property of spatial locality as well. As a result the hit rate decreases when the program exhibits poor spatial locality.

The system proposed in this paper uses no associative hardware at all. Yet, its performance is totally independent of spatial locality of code and data in the main memory. Hence its performance would be as good as or better than that of a fully associative cache of similar capacity.

If a cache memory system is to be completely transparent to software it should cater to the following two attributes of programs:

- A datum may be operated on by several distinct instructions.
- An instruction may receive control from that assigned to the preceding location in the main memory or from a branch instruction.

The following sections show how a hypothetical computer utilizing the proposed cache mechanism provides for these attributes while minimizing accesses to the main memory. For clarity here, the hypothetical computer is assigned a simple architecture. However, with suitable modifications, the proposed mechanism will complement most other features of modern computers including those of multi-processor systems.

2 General Features

This cache system is completely transparent to software. Programs are compiled or assembled and loaded onto a random access main memory as if there were no cache. In contrast to conventional systems, there is no restriction on the location in the cache at which an item may be entered. The Execution Unit (EU) does not perform address arithmetic for instructions or for scalar data. This work is done by a Cache Management Unit (CMU) and is required only when a miss occurs. The miss detection test is the simplest possible; compare two bits.

The cache is accessed as a random access memory. When decaching and encaching, it is operated as a normally full cyclic FIFO buffer. All decached items are returned to the main memory. Thus, only one copy of each item is maintained. Instructions and scalar data are placed in the same cache. Arrays may be placed in the same cache provided that each array is taken in its entirety. Arrays may also be placed in a separate cache. These conditions are necessary for the proper functioning of the miss detection test.

3 Structure

Fig. 1 shows the functional components of the proposed system. For clarity at this stage we consider a computer in which all instructions and data are of fixed length p bits, with each instruction or datum occupying one **word** of the main memory.

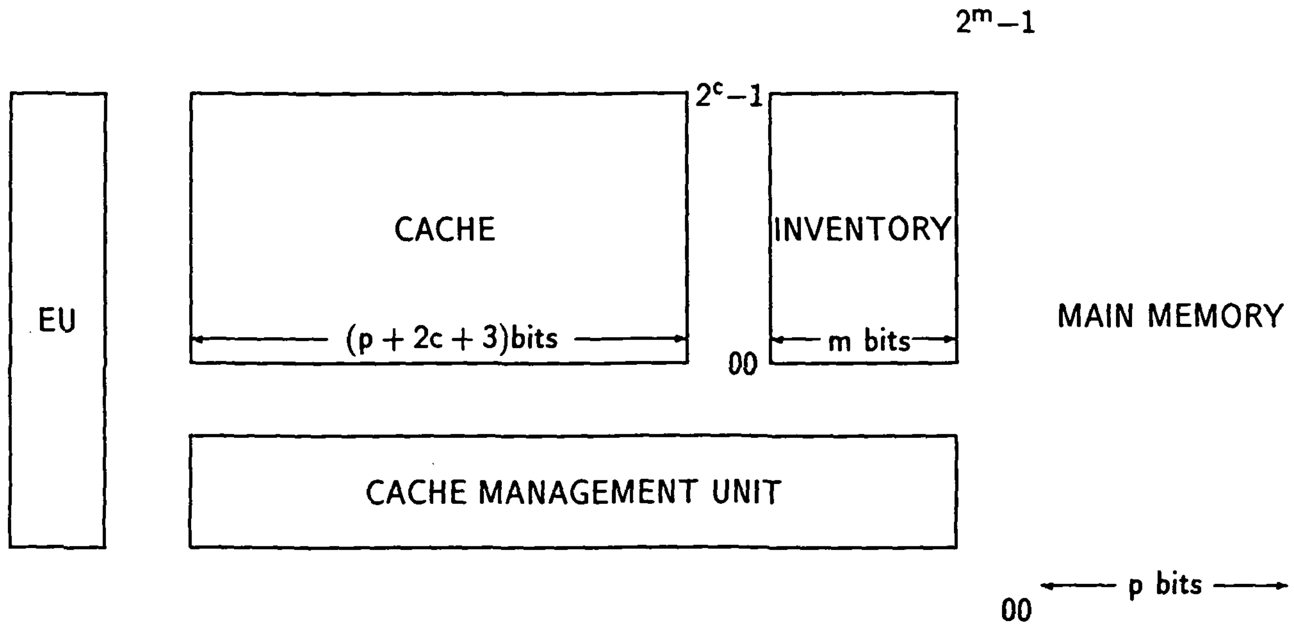

Fig. 1 Functional components of the cache system.

The cache consists of 2^c **lines**; i.e. a line has a physical address of c bits. We use the term "physical address" because, at a later stage we need to introduce the term "logical address" with respect to cache lines.

The cache contains both instructions and data. Each cache line contains one **item**, viz., one instruction or one datum. The cache is normally full. Hence the current item in a line must be decached before a new item can be encached there. A register contained in the CMU holds the address of the line from which an item will be decached next. This is referred to as the Decache Address (DA). The cyclic FIFO rule is implemented by incrementing DA after each encache operation, with wraparound when necessary.

For this simple model it will be shown that each line should contain $(p + 2c + 3)$ bits. Of these, p bits are required for the encached item. The additional $(2c + 3)$ bits comprise fields that hold information required for non-associative operation of the system. The function of each of these fields will be described as the need arises.

The inventory holds the main memory address of each encached item. Thus a cache address refers to a line in the cache as well as the corresponding **entry** in the inventory. If the main memory contains 2^m words then each entry in the inventory takes m bits.

The CMU encaches each item when it is first required by the EU. Hence, the EU needs to communicate with the cache only, unless a miss occurs. In this event it signals the CMU which in turn accesses the cache, inventory and main memory to take corrective action.

4 The Contents of the Cache

An instruction must be in the cache before its execution may commence. Similarly, a datum must be in the cache before it may be operated on.

Consider the execution of a register/memory instruction in which the register is implied in the opcode. Assume that the instruction has been just encached at CACI, the Cache Address of the Current Instruction, and that the datum and the next instruction are in the main memory.

The EU is unaware that the datum is not in the cache and tries to access it using the content of a field from the extra $(2c + 3)$ bits at CACI. The exact line accessed is irrelevant at this point; it is sufficient to note that, because the datum is not in the cache, the miss detection test must return a miss.

The current instruction which is in the line at CACI consists of the opcode and the operand as obtained from the main memory (Fig. 2). The operand is the Main memory Address of the Datum (MAD). The CMU gets MAD from the line at CACI, accesses the main memory, obtains the datum and encaches it in the line at DA. Note that MAD and DA are not related.

The CMU enters DA in a field set apart for it from among the extra $(2c + 3)$ bits in the line at CACI. This field is referred to as the Cache Address of the Datum (CAD). The EU hereafter uses CAD to access the datum in the cache.

For this particular example, since the current instruction and the datum were encached in immediate succession, we have

$$\mathrm{CAD} = \mathrm{CACI} + 1 \tag{1}$$

This relation will not hold true in general because the datum may have been encached earlier for it to be operated on by a preceding instruction. Hence, the pointer CAD in a field in the line at CACI is a necessary requirement.

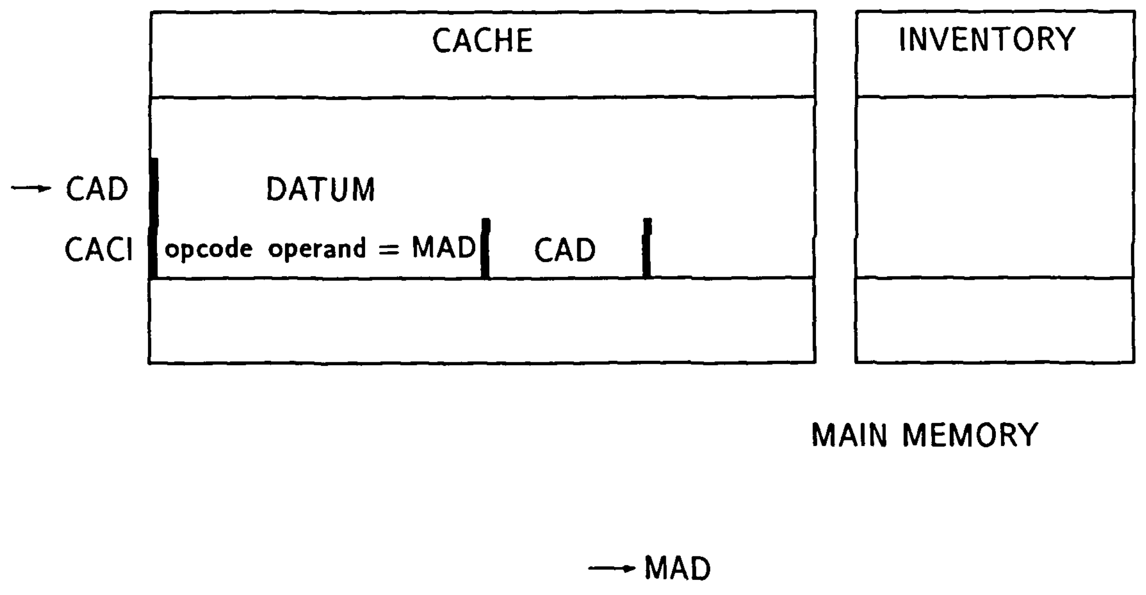

Fig. 2 Current instruction and its datum encached in succession.

In the case of a datum which has been encached earlier, the CMU must first locate the line containing the datum in order to update the CAD field at CACI. The procedure to locate this line will be explained in a later section.

After the datum is operated on, the EU tries to pass control to the instruction at the next sequential location in the main memory. But, since this instruction is not in the cache the EU scores a miss. The CMU then takes corrective action as described below.

At the time that the current instruction was encached the inventory would have been updated. Thus the inventory entry at CACI contains MACI, the Main memory Address of the Current Instruction (Fig. 3). Let the Main memory Address of the Next Instruction be denoted by MANI. Then,

$$\mathrm{MANI} = \mathrm{MACI} + 1 \tag{2}$$

The CMU accesses the inventory at CACI and obtains MACI. It then increments MACI and accesses the main memory at MANI to obtain the next instruction which it encaches at CANI, the Cache address of the Next Instruction. It then enters CANI in a field in the line at CACI.

From the FIFO rule, for this particular example

$$\mathrm{CANI} = \mathrm{CAD} + 1 = \mathrm{CACI} + 2 \tag{3}$$

Again, this relation does not hold true in general because what is referred to here as the "next instruction" may have been executed earlier, after which a branch may have transferred control to the current instruction. In that case the next instruction would have been in the cache before execution of the current instruction commenced. Hence the pointer CANI in the line at CACI is a necessary requirement.

The two pointers CAD and CANI account for $2c$ bits at CACI. The remaining 3 bits are employed for the miss detection test which will be described later.

If the instruction at CACI receives control at a subsequent time and if its datum and the next instruction have not been decached in the meantime, the pointers CAD and CANI will remain valid. The instruction at CACI will then be executed with no access to the main memory or to the CMU. The only additional work for the EU is the miss detection test.

Now, consider the execution of a branch instruction. Let the branch condition be implied in the opcode. Here, the operand MAD points to the "datum" which is in fact the branch destination in the main memory. If the branch is not taken the EU does not need the "datum" and hence does not access the cache at CAD. As before, CANI passes control to the next instruction. When the branch is taken for the first time, the EU scores a miss when accessing the "datum" in cache. At this point the CMU locates the "datum" and updates the CAD field of the line at CACI. The EU then branches to CAD. Thus, a loop containing one or more branch instructions will, after the first pass, be executed with no reference to the CMU, the inventory or the main memory. The only condition is that the code and data of the loop do not require more lines than the total number in the cache.

If the EU exits the loop more items may need to be encached. By the FIFO rule some items will be decached. If the EU enters the loop again it may attempt to access an item which has been decached. On attempting access using the now erroneous content of a CAD field or a CANI field it scores a miss. The CMU then encaches the item or finds where it is in the cache and updates CAD or CANI.

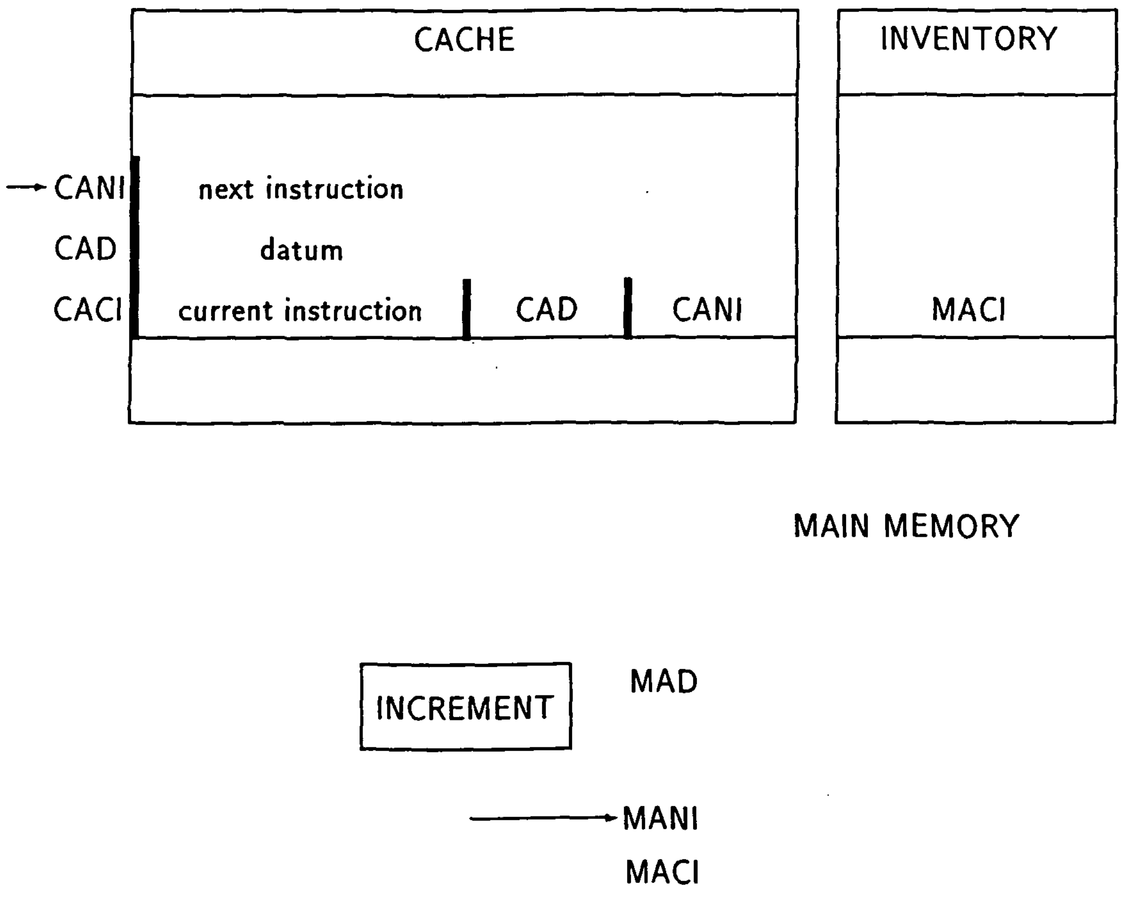

Fig. 3 Current instruction, its datum and the next instruction encached in succession.

In summary, after encaching an instruction the CMU augments it with two cache addresses. The EU runs the program with no address arithmetic.

5 The Contents of the Main Memory

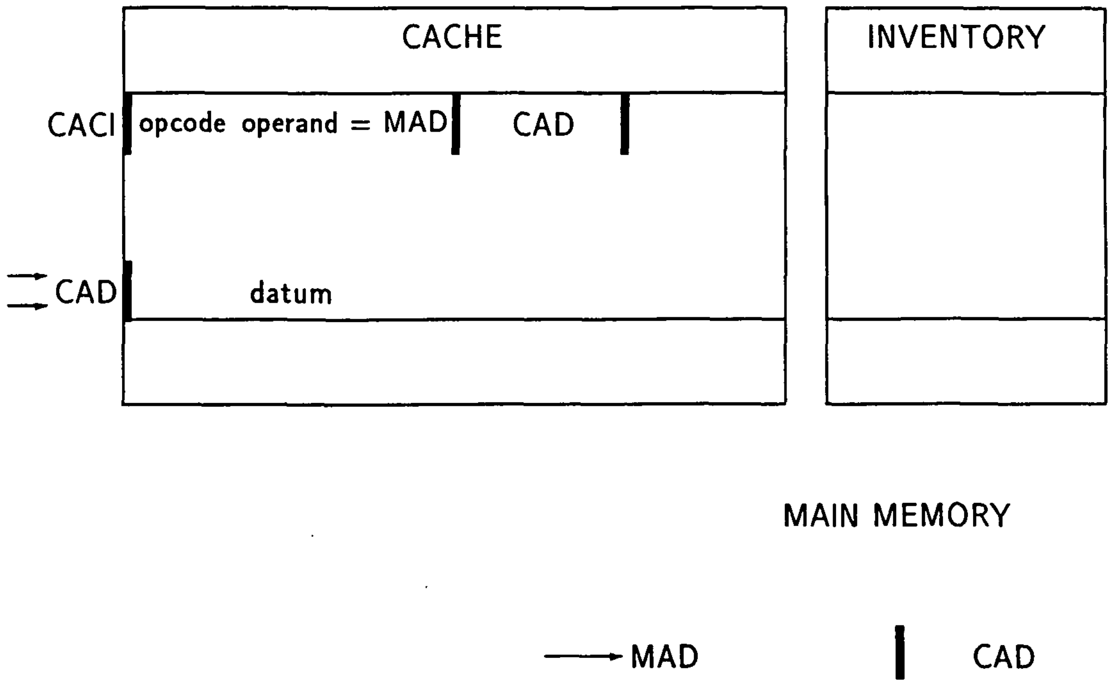

Fig. 4 Datum encached by a preceding instruction.

In trying to access a datum in the cache the EU scores a miss whenever the pointer CAD is invalid. CAD can be invalid under two conditions:

- The datum is in the main memory.
- The datum has been encached by a preceding instruction.

In the former case, the CMU encaches the datum and updates CAD by inserting DA as explained in the previous section. In the latter case the CMU must find where the item is before it can update CAD. In order to provide for this, whenever a datum is encached DA is written in the word in the main memory from which the datum is taken. Thus only one copy of the datum exists at any instant; after a datum is encached the word that originally carried it will contain a pointer to it. MAD becomes an indirect pointer to the datum and CAD becomes a direct pointer. This procedure guarantees that the CMU can link a datum to any number of instructions. Thus, for the case where a miss occurs while the datum is in the cache, the CMU obtains MAD from the operand field of the instruction itself and accesses the main memory at MAD to get the cache address of the datum (Fig. 4). A similar procedure is carried out whenever the EU scores a miss in trying to access an instruction.

The main memory now contains a mix of items and cache pointers. The CMU must distinguish between these two types. An obvious way to provide for this would be to increase the length of a main memory word to $(1+p)$ bits, with the extra bit carrying a boolean field INMAIN. An alternative technique which maintains the length of a main memory word at p is described in a later section.

6 The Encache Procedure

Whenever a miss occurs the EU supplies CACI to the CMU. It also indicates whether the fault was in CAD or in CANI. The CMU then accesses the main memory at MAD or MANI as appropriate. If the required item is in the cache it updates the faulty field at CACI. If the item is in the main memory the CMU encaches it before updating the field.

The steps of the decache/encache procedure and the reason for carrying out each step are listed below.

1. Get the inventory entry at DA. This entry is the main memory address of the word from which the current item in the line at DA was taken.

2. Return the current item in the line at DA to the main memory word at the address given by the inventory entry. Only one copy of an item is maintained.

3. Invalidate the content of the CAD and CANI fields at DA. If the new item to be encached is an instruction it must be forced to score misses when first attempting to access the datum and the next instruction.

4. Place the new item in the line at DA. The EU will access this directly hereafter.

5. Copy DA into the main memory word. The CMU will use this as a pointer to the item if it becomes necessary to link it to other instructions.

6. Enter the main memory address of the encached word, in the inventory at DA. This entry is required for proper decaching of this line. Decaching will become necessary once DA has been incremented through one wraparound. This entry is also necessary to find MANI and explicitly link an instruction to its successor in the cache.

7. Insert DA in the appropriate field, i.e., CAD or CANI, in the line at CACI. This ensures a hit if the current instruction is executed again.

8. Increment DA with wraparound if necessary. This implements the cyclic FIFO rule.

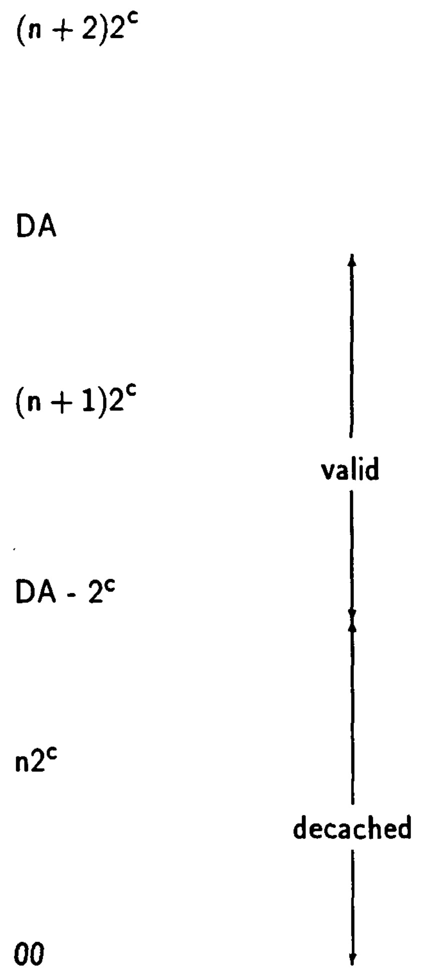

Fig. 5 Logical addresses of cyclic FIFO buffer represented as a quasi-finite stack.

7 The Miss Detection Test

For decache/encache operations, the CMU treats the cache as a FIFO buffer with physical addresses ranging from 0 to $(2^c - 1)$. Logically, this FIFO buffer may be treated as a quasi-infinite stack in which only the top 2^c locations contain valid items (Fig. 5). In this logical representation, to encache an item the CMU pushes it on to the stack. Since only the top 2^c items are valid a decache operation is not required.

The logical address of a line in the stack lies in the range 00 to (DA - 1) and is given by

$$\textit{logical address} = \textit{physical address} + n2^c \tag{4}$$

where n is the count of the number of wraparounds since power-up. n is held in a field of MSBs in DA. This field may be referred to as the wraparound part of the logical address.

Whenever an item is encached the wraparound field of DA is stored in a field set apart for this purpose at each location in the cache. The logical address of a cache location is determined by appending the physical address to the content of the wraparound field at the accessed location, treating the former as LSBs.

Recall that whenever an item is encached, its cache address CAD or CANI as the case may be, is stored in a field at CACI. If CAD and CANI consist of logical addresses a miss may be detected by comparing the wraparound part of CAD or CANI with the content of the wraparound field at the accessed location. Equality indicates a hit.

As the program runs, more and more items get encached and n increases steadily. It appears that the miss detection test would be effective only if the wraparound field is large enough to accommodate all wraparound from power-up to shutdown. The upper bound to the length of the wraparound field may be determined as follows.

At the instant that CAD is updated to a value say CADi, the datum and the current instruction are both present in the valid portion of the stack. Hence,

$$0 <| \text{CADi} - \text{CACI} |< 2^c \tag{5}$$

Let the contents of the physical location represented by CADi be replaced zero or more times. Let the resulting logical address of that location be CADj. If the same instruction receives control again then CACI and CADj must fall within the valid portion of the stack. Hence,

$$0 <| \text{CADj} - \text{CACI} |< 2c \tag{6}$$

Also CADj > CADi Therefore, from equations (4) and (5),

$$0 < \text{CADj} - \text{CADi} < 2^{1+c} \tag{7}$$

Lemma If K, L, M are non-negative integers such that $K - L < 2^M$
then, $K = L$ iff $[K = L]$ modulo 2^M

Thus, CADi = CADj iff [CADi = CADj] modulo 2^{1+c}

Hence, the maximum length required for logical addresses is $(1+c)$ bits. **The wraparound field need not be more than 1 bit long.**

Now, since CADi and CADj both refer to the same physical cache location

$$[\text{CADi} = \text{CADj}] \text{ modulo } 2^c. \tag{8}$$

Therefore the miss detection test reduces to comparing two bits; the MSB of CAD or CANI and the wraparound bit stored at the accessed line.

In summary, DA is maintained as a register of $(1 + c)$ bits. The current MSB or wraparound bit is recorded at each location when an item is encached. When the datum is encached the $(1 + c)$ bits of DA are entered in the CAD field. The EU accesses the datum using the c LSBs of CAD and compares the MSB of CAD to the wraparound bit stored at the accessed location. Equality indicates a hit. The procedure in the case of the next instruction is similar.

8 Miscellaneous

8.1 Word Length of Main Memory

As explained earlier, whenever an item is encached its cache address is placed in the main memory word. As a result the main memory contains a mix of items and pointers. In order to distinguish between these two types an additional bit may be incorporated in every word of the main memory. The following procedure eliminates the need for an additional bit.

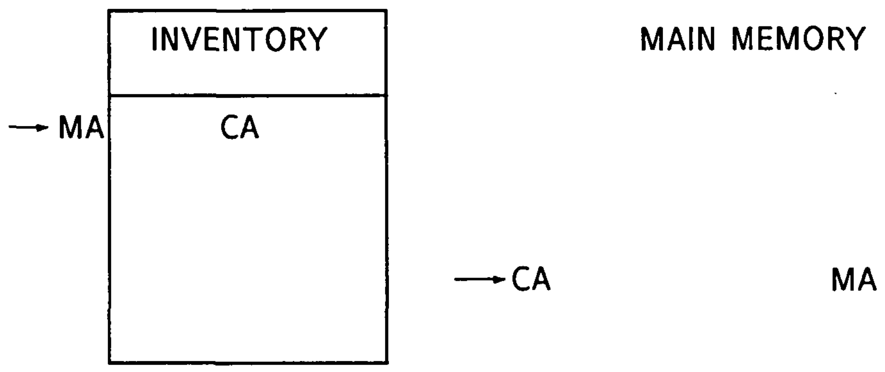

Fig. 6 Pointers at an encached word of the main memory and in the inventory.

Whenever the CMU accesses the main memory, say at MA, it interprets the c LSBs of the content of the accessed word as a cache address, say CA (Fig. 6). It then obtains the inventory entry at CA and compares it with MA. Equality indicates that MA contains a cache pointer. Inequality shows that MA contains a program item.

8.2 Decaching the Current Instruction

It is possible that the EU scores a miss when CACI is equal to DA. This would result in the current instruction being decached. This possibility should be taken into account in organizing the data transfers within the CMU.

9 Conclusions

A novel method of implementing a cache memory is proposed. Its primary features are the absence of associative hardware and complete independence from spatial locality of code or data. The absence of associative hardware makes large caches possible at relatively low cost. Conventional single-address instructions get automatically augmented to a two-address format. This enables the program to be executed from the cache without address arithmetic. A cache miss is detected by the simplest possible test; compare two bits. The correctness of the test has been proved with mathematical rigor.

10 Acknowledgements

The work described in this paper was done by the author during a sabbatical year in 1974/75, at the Department of Computer Science, University of Reading, U.K. The author expresses his gratitude to the Royal Society and the Nuffield Foundation for the grant that enabled him to spend a year in the U.K. and to Dr. J.D. Roberts who helped and encouraged him throughout the course of this work. The author wishes to acknowledge the generous assistance by way of suggestions and comments given by Dr. T. Srikanthan and Dr. A.B. Premkumar during the writing of the present draft. Thanks are also due to Mr. H.G. Wijewansa, Mr. T. Kirubarajan and Miss. S.S. Madahapola who assisted the author in preparing the final version of this paper.

11 References

[1] J.A. Gunawardena, "Improvements in or Relating to Computer Store Mechanisms," United Kingdom Patent No. 1531261 of 1976. Filed by the National Research Development Corporation, London.
[2] J.A. Gunawardena, "Improvements in or Relating to Computer Store Mechanisms," United States Patent No. 4212058 of 1980. Filed by the National Research Development Corporation, London.