**Carnegie-Mellon University**

Department of Computer Science
Schenley Park
Pittsburgh, Pennsylvania 15213
[412] 621-2600

June 7, 1976

Professor Wayne Lichtenberger
Department of Electrical Engineering
University of Hawaii at Manoa
2540 Dole St.
Honolulu, Hawaii 96822

Dear Wayne:

When I was in Hawaii, I became intrigued with the fast memory on the 500. Using the document you gave me and talking to Mel last year, I tried to spell out the algorithms for the different parts of the fast memory logic. I think they are quite fascinating. However I will not be able to complete them. If you or someone else would like to complete them and correct the description of the algorithms, I would like that very much. Even at this late date people would find this interesting, I think.

Best of luck with everything.

Sincerely,

Anita K. Jones
Assistant Professor

AKJ/dmj
enc.
cc: M. W. Pirtle

Anita Jones

# FAST MEMORY

## A Technique for Increasing the Effective Access Cycle of Central Memory

## INTRODUCTION

Since the inception of the modern computer system, providing a sufficient quantity of directly addressable memory -- cost effectively -- has been a problem. The solution has been to use a hierarchy of memories: directly accessible central memory, sequentially addressable secondary memory and archival off-line storage. Processing power is used to manage the flow of data between levels in the hierarchy so that the several levels effectively simulated a large directly addressable memory containing all information in the system; that is, whenever data is required for program execution, it can be directly accessed. Such techniques are cost effective because the processing power to support them is cheaper than the fast directly addressable passive memory being simulated.

An additional problem arises when the central memory that is cost effectively available -- or available in the market place at all -- responds so slowly (that is, has a long cycle time) to accesses made by processors that the processor's instruction execution rate is effectively reduced. Again the solution is to expend processing power to effectively increase the performance of the memory by interposing a mechanism between the direct access memory and the processor. This paper narrates one innovative solution to the problem of providing cheap but very fast directly accessible memory.

## THE PROBLEM

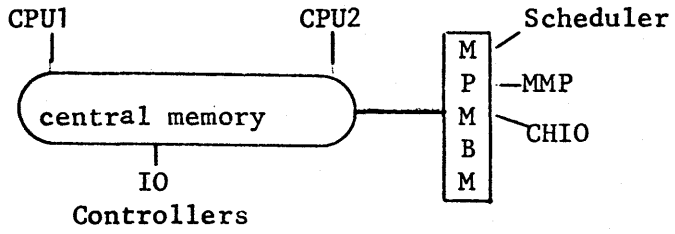
The problem of providing rapidly and directly accessible memory arose in the design of the Berkeley Computer Corporation time sharing utility, the

BCC500 system [ ]. In particular central memory was to be directly addressable by five microprocessors (maximum rate of 15 million operations per second each) and sequentially accessible to four (two drum and two disk) controllers collectively able to sustain a transfer rate of over 12M bytes per second. Any solution was constrained to minimize interference, minimize cost and to be a maintainable component of the system.

At the time of the design (1969) a reliable direct access memory fast enough to satisfy the voracious appetites of the five processors and four controllers was not on the market. Furthermore, the fastest reliable memories on the market were very expensive. Restated the problem was: Is there a way to interface 1 microsecond memory* to the processors and controllers (designed and built at BCC) to meet the economic and performance goals stated above?

MOTIVATING THE SOLUTION

The central memory is referenced through four ports connected to the processors.




The two Central Processing Units (CPUs) are dedicated to executing user jobs. In contrast to conventional operating systems the BCC500 has three microprocessors dedicated to performing physical resource management. The Scheduler

*Judged at the time to be the fastest memory that was (1) reliable, (2) maintainable, (3) quickly obtainable and (4) affordable!

performs user process synchronization and process multiplexing on the two CPUs. The Memory Management Processor (MMP) manages the flow of user process working sets between central memory and secondary storage so that the Scheduler directs a CPU to execute a user process only if that process' working set is resident in central memory. The Character IO (CHIO) processor manages the information streams between the system and the many local and remote console devices. The IO controllers transfer pages between central memory and drum or disk as directed by the MMP.

The three management processors each have private memory though they do share central memory data structures such as process descriptors and thus require access to central memory. Since the resource management processors make relatively few accesses to central memory (for example, they execute their microprograms and do not fetch instructions from central memory as do the CPUs), their requests to central memory are multiplexed onto one bus by the MPMBM.*

The goal is to increase the apparent speed of the central memory. In fact the goal is for the processors and controllers to see a central memory which appears to be an order of magnitude faster than the actual 1 microsecond core memory.

One approach to this problem is to place a small fast memory called a cache in the processor into which small blocks of central memory contents are 'demand paged'. Effectiveness of cache depends on the locality property of program execution. The locality property predicts that if a word is referenced, it or a near neighbor word will be referenced shortly. Thus if the word is referenced, it and its neighbors (a cache page) should be brought to

*Mel Pirtle's Miraculous Bus Multiplexor

the cache so that references to that page which are expected to occur in the near future can be performed at cache speed.

The cache solution presents several difficulties. First, to solve the present problem, every access should be fast, not just the second and succeeding references to a 'cache page'. Second, if multiple processors are to reference a shared central memory, the cache solution (a cache per processor) introduces the problem of maintaining the accuracy and consistency among the contents of the multiple caches and the central memory. Third, many of the accesses to central memory are by secondary storage device controllers whose accesses are sequential by nature of the devices. Caches are not generally used for these references.

It must be concluded that cache does not solve the present problem; indeed it would introduce substantial difficulties in maintaining consistency.

## THE SOLUTION -- AN OVERVIEW

In contrast to cache, the solution implemented associates with each of the eight central memory (core) modules a buffer of six FAST REGISTERS. These registers together with the logic to manage them are called the FAST MEMORY. The logic in each fast memory selectively assigns registers in the fast memory to locations in the associated core module. Then the fast register is effectively substituted for the assigned core location. If the required data for reads is in a fast register, then both reads and writes can be serviced at fast memory speeds so that the apparent speed of central memory is increased. In the BCC500 each fast memory can accept a request each 100 nanoseconds, an order of magnitude faster than the core memory.

Note that the fast memory solution does not have the consistency difficulties of the cache solution. References from all processors to central memory go

to the single fast memory associated with the core module addressed so that all processors see an updated value as soon as it is changed. Description of the fast memory solution will demonstrate that it does not have the other disadvantages of the cache:

→ every processor or controller access can be serviced by the fast memory, not just the second and succeeding accesses to the artificially defined 'cache page'. Indeed the fast memory is even designed to take advantage of the fact that a substantial number of memory accesses (at least those made by the controllers) have a sequential pattern.

There will be cases in which the fast memory will fail to have data ready to satisfy a read request. To minimize the occurrence of this event, the memory interface to the processor is altered so that the fast memory can REJECT requests which it cannot honor immediately out of the fast registers. In fact a response is generated in the same clock cycle as the request so that the central memory appears to a processor to be a component internal to the processor! If the fast memory does not reject a request, then any data to be provided by the memory will be available on the bus during the next clock cycle.

The processors and controllers are adapted to the fast memory in two ways:

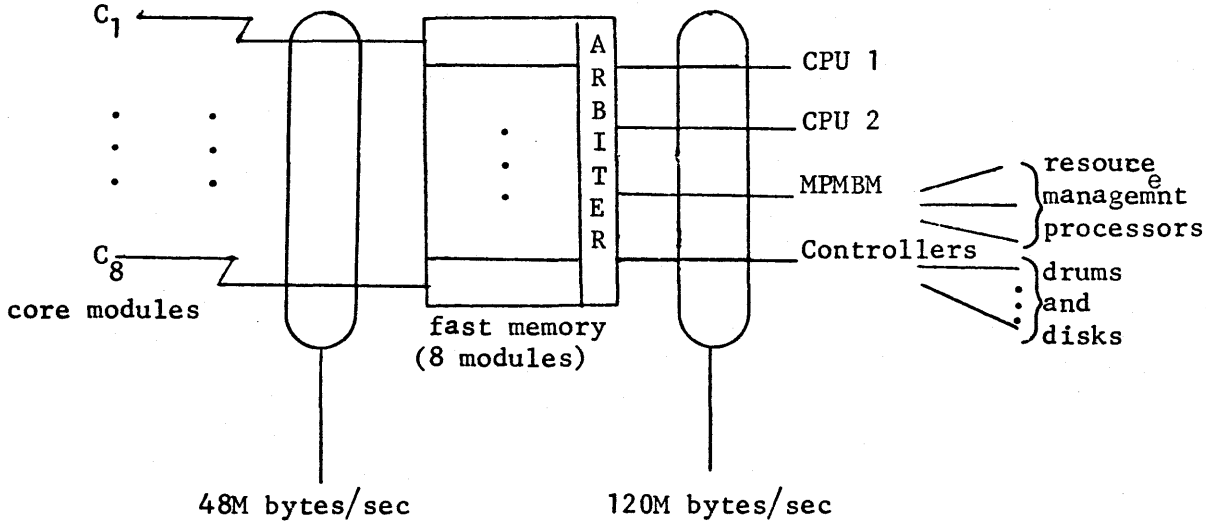
- a request to memory can fail (be rejected) and must be retried
- prefetch requests can be made to alert the fast memory to acquire data for an imminent access.

The fast memory is another example of using plentiful and inexpensive processing power to upgrade the effective use of a resource. In this case the logic associated with the fast registers and the logic to augment the processor

and controller interface to memory as just suggested is the price paid to make 1 microsecond memory appear to be a 100 nanosecond memory. For a discussion of this philosophy see reference [ ].

We will now discuss the interconnections between the fast memory and the other BCC500 system components, then give an overview of the logic which interfaces the fast memory with core (the 'inboard' side of the fast memory) and the 'outboard' interface between the fast memory and the processors and controllers.

If access time is to be minimized, it is essential that the memory be near the components making requests of it. When dealing with memory response times of 100 nanoseconds, every foot of cable separating the fast memory and the requesting components will noticeably increase the effective access time. (That is the reason the MPMBM is backplane cabled to the resource management processors.) The BCC500 components are connected as follows:



Because the core cycle time is so slow, the length of the cables between core and fast memory is relatively unimportant. So the fast memory is located in the same physical package as the CPUs, the MPMBM and the controllers so that the cabling between them is backplane cabling, no more than a few feet in length.

The bus between memory and the requestors (processors and controllers) has the following lines:

| to requestor clock time | line name | to fast memory number of lines |
|---|---|---|
| t | ADDRESS → | 19 |
| t | REQUEST → | 4 |
| t | DATA IN → | 26 |
| t | CPRIORITY → | 2 |
| t | PPRIORITY → | 1 |
| t+1 | ← DATA OUT | 26 |
| t | ← REJECT | 1 |
| t+1 | ← ACCEPT | 1 |
| t+1 | ← SATISFY | 1 |

The bus is clocked. At each clock time t the requestor can place values on the ADDRESS, REQUEST, CPRIORITY and PPRIORITY lines and perhaps the DATA IN line if the request is a 'store'. At the end of the clock cycle (100 nanoseconds) a REJECT response will have been set or reset by the fast memory. If the request was not REJECTed then at time t+1 the fast memory will set the ACCEPT, SATISFY and perhaps the DATA OUT lines. At the fast memory side the bus enters an arbiter which resolves the (hopefully) infrequent conflicts arising when two requestors attempt to reference a location in the same core module. The arbiter uses the PPRIORITY (port priority) value, which is either low or high, to select one request to be gated into the fast memory logic.

REJECT responses are sent to all others. We will not describe the fast memory arbiter any further.

REQUEST is composed of four lines

F fetch request

S store request

H hold or defer (used by the inboard logic to determine when to transfer data between a register and core memory)

Q signifies whether a request is from a sequential device

The REQUEST line values (H and Q) are essentially control information to allow the fast memory logic to optimize the service it provides to requestors.

In the remainder of this section we will describe the fast memory. The explanation ignores details irrelevant to the basic idea of the fast memory solution. For example, the core memory is double word interleaved fetching a double word every cycle, so each fast register contains a pair of data words. This is ignored in the remainder of this section.

Looking more closely at the fast memory structure one finds an array of six registers each containing four kinds of information:

ADDR specifying the address of the core location assigned to the register

DATA contents of the assigned core location

DATASTATE specifying whether
R correct data is in fast register
C correct data is in core location *

STATUS status of the register

H requestor intends to reuse register in immediate future
IP core cycle is 'in progress' for this register
Q signifies whether last requestor addresses memory in a sequential pattern
P priority information

We will first give a version of the outboard algorithm which processes requests. Again, it masks many of the problems and nuances of the logic but suggests roughly what takes place. The outboard algorithm first determines

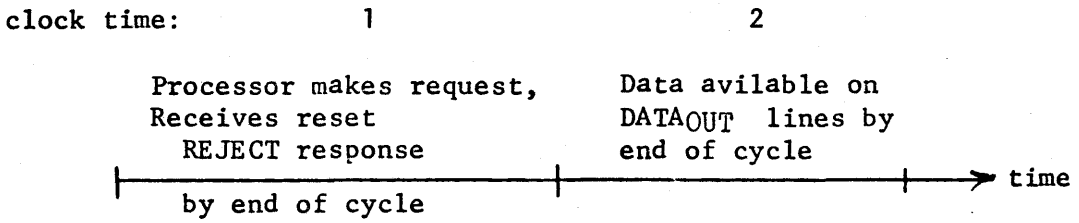
---

* We will freely mix terminology by referring to a bit, say RL, as being 'set' or 'reset', or 'true' or 'false'. (A set bit is interpreted to encode the boolean value true.)

a fast register assigned to the ADDRESS in the request. The second portion responds to the specific request made. Described in English prose augmented by familiar programming language control constructs, the algorithm stated below narrates what the outboard logic of each fast memory module does every clock cycle.

```
if ADDRESS in range this module then
 begin if ¬ (fast register X assigned to ADDRESS)
        then if ∃ X, an available* register
                      then (assign X to ADDRESS;  ACCEPT ← true)**
                      else (ACCEPT ← false;  exit begin block);
      update X.STATUS using REQUEST and CPRIORITY;
      case
        store request:      (latch DATAIN to X.DATA;  adjust X.DATASTATE;
                             SATISFY ← true)
        fetch request:      if X.R then (latch X.DATA to DATAOUT; SATISFY ←true)
                                   else SATISFY ← false
        prefetch or reserve request:  SATISFY ← true
 end
```

This outboard algorithm can respond to a request every 100 nanoseconds. Consequently fetched data is returned to a requestor in the clock cycle following that in which the request is made. For example, if a read request is made when the required data is in a fast register:




*Register availability for assignment will be discussed in detail later.

**Where no confusion should arise (mostly because of indention and spacing), parentheses are used to denote block begin and block end.

The time for a single successful fast memory request is 200 nanoseconds. However, during each clock cycle a processor may make a request so that a request and the results of the previous request are on the bus during the same clock time making the effective fast memory access time 100 nanoseconds. The MPMBM costs an extra clock time so that the resource management processors receive REJECT responses during the clock time after the request is made and any data is received two clock times after the request is made.

The fast memory is analogous to a demand paging system. Demands are in the form of memory access requests. Each register corresponds to a page frame in the demand paging system. One difference between the two is that the processors cooperate with the fast memory to the extent of announcing imminent accesses by prefetches so that the fast memory can preassign registers and fetch data from core memory if necessary. The inboard fast memory logic attempts to keep the values in the register and core the same.

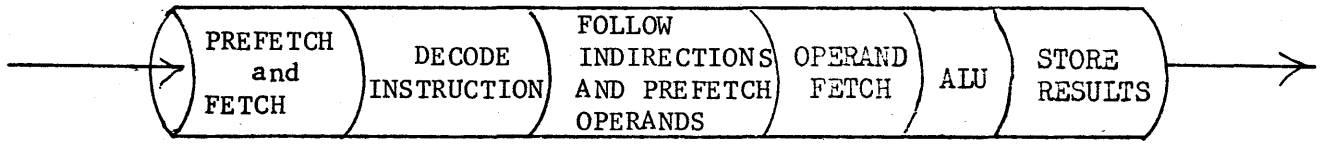
Except for the shared fast registers the inboard and outboard logic implement quite separate algorithms. While the outboard logic executes at processor and fast register speed, the inboard logic necessarily executes at core cycle speed (10 times more slowly). Thus the two algorithms are synchronous by the clock, but asynchronous by activity.

Each core cycle the inboard logic of the fast memory <u>functionally</u> performs the following operation. (Again, the fact that the core modules transfer a double word at a time is ignored.)

```
if ∃ fast register X with DATASTATE and STATUS values (R ∧ C̄ ∧ H̄) or (R̄ ∧ C)
   then begin select the highest priority such register X;
          X.IP ← true;
          if X.R̄ ∧ X.C then (X.DATA is loaded from core location X.ADDR;
                                X.R ← true)
                        else (X.DATA is stored in core location X.ADDR;
                                X.C ← true)
          X.IP ← false
        end
```

The goal of the inboard logic is to cause the DATASTATE flags R (right in the register) and C (right in core) to be set.

As mentioned earlier, the microprocessors are adapted to match the fast memory. The CPUs are pipeline machines in order to be able to decode instructions fetched from memory and prefetch operands before they are required for performance of operations in the ALU. The CPU pipeline can be graphically described by



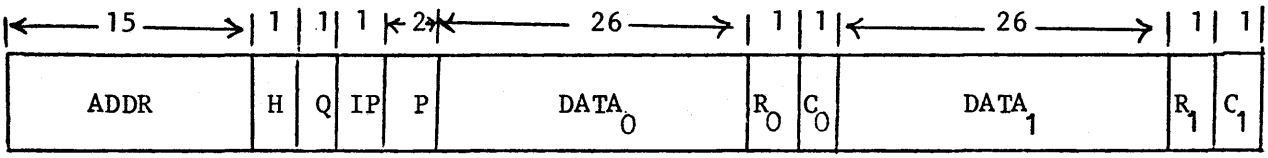

The first section of the pipe logic prefetches and fetches instructions, possibly pursuing both paths following a branch. The instruction prefetch and fetch portion of the pipe increment shadow program counters. Instructions are decoded and the operands determined and prefetched. This may involve indirection and indexing. The arithmetic and logic unit (ALU) is the only portion of the pipe in which the state of the machine is actually changed. It is the ALU that alters the actual program counter. The ALU will request its operands and find them in fast memory. The last portion of the pipe stores results in the operand location which may be in central memory. It is this portion of the CPU that can respond to the rare event that the fast memory cannot accept a store request.

THE SOLUTION - A PRECISE DESCRIPTION

To make the fast memory description more precise the inboard and outboard algorithms will be refined. Special attention will be paid to:

1. ramifications of the fact that core modules transfer double words
2. the inboard priority selection algorithm (used to determine for which register a core transfer is to be performed)
3. the register assignment algorithm
4. the special handling of sequential devices.

As noted earlier the core modules accept or provide a double word of data at a time. Memory locations N and N+1 (N even) are in module N mod16. Consequently a fast register also holds a double word of data. We now revise our earlier sketch of the structure of the fast registers. The 76 bits of a fast register are divided into the fields shown below:




The $DATA_0$ and $DATA_1$ fields hold the contents of the double word (in the associated core module) indicated by ADDR. $DATA_0$ ($DATA_1$) holds the left (right) half of a double word. The status of the $DATA_0$ ($DATA_1$) field is recorded in $R_0$ and $C_0$ ($R_1$ and $C_1$). $R_0$ and $R_1$ indicate whether the correct value is in the associated data field. $C_0$ and $C_1$ indicate whether the correct value is in core. Thus these four bits together compose what we called DATASTATE earlier.

The objective of the inboard logic is to maintain every register in its normal state -- i.e. one in which $R_0$, $R_1$, $C_0$, $C_1$ are set meaning that both

the fast memory register and the assigned core locations (the double word) are correct. A transfer between a fast register and core is required if the boolean expression CORETRANS is true when CORETRANS = FETCH $\vee$ (STORE $\wedge \overline{H}$). A fetch is required just in case FETCH is true, where

$$\text{FETCH} = (\overline{R_0} \wedge C_0) \vee (\overline{R_1} \wedge C_1).^*$$

A store is required just in case STORE is true, where

$$\text{STORE} = (R_0 \wedge \overline{C_0}) \vee (R_1 \wedge \overline{C_1})$$

H is used in conjunction with prefetch requests to insure that in most cases the requested location will be held in fast memory until the fetch request is made. It is used in conjunction with fetch requests to hold a double word in a fast register until each half has been fetched and in conjunction with stores to defer the store of a word to core until both halves of a double word have been stored into the fast register. Thus H is usually set when the first of two references to a double word is made or when a prefetch is made. A timer is associated with each fast register hold bit. It is used to limit the time a processor can attempt to force the fast memory to maintain a particular register assignment. Each time H is set, the associated timer is set. After an interval measured by the timer elapses (approximately 4 $\mu$sec if the Q bit is set and 2 $\mu$sec if it is reset) the hold bit is reset.

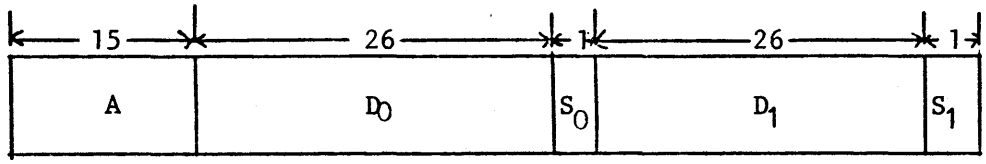
Shortly after a core cycle begins the inboard logic computes the boolean expression CORETRANS and selects the first register of highest priority (consulting the P fields) encountered in a cyclic scan of the registers beginning with the register indicated by the SCAN pointer. Functionally this can be described as a boolean procedure SELECT which, if successful, determines a register for which a core transfer is to be performed and places the name of the register in a variable X accessible to all the inboard logic. We

assume SCAN points to a register and will not change during the execution of SELECT.

```
boolean procedure SELECT =
  begin local I,J,PRIOR;  PRIOR ← -1;  X ← -1;
        for I ← 0 to 5 do
          begin J ← (SCAN + I) mod 6;
                if J.CORETRANS ∧ PRIOR>J.P then (PRIOR ← J.P;  X ← I)
          end
        if PRIOR > -1 then (X.IP ← true;  return true)
                      else return false
  end
```

The major portion of the inboard logic can now be described. If SELECT returns a value of true, the X.IP 'in progress' bit is set so that register X will not be considered by the outboard register assignment algorithm.

All core transfers are via a single high speed memory register M associated with the core module and containing five fields:




The A field contains the address (within the module) of the double word being accessed during a core cycle. $S_0$ ($S_1$) is true if the corresponding data field $D_0$ ($D_1$) has been loaded from a fast memory register.

Functionally, the main inboard logic can be described by the following program when { and } act as begin and end block delimiters and in addition indicate that the bracketed statements can be executed in parallel:

```
procedure INBOARD =
    repeat if SELECT then
        begin { D0 ← 0; D1 ← 0; S0 ← false; S1 ← false; A ← ADDR}
              {if X.R0 ∧ ¬X.C0 then {D0 ← DATA0; S0 ← true}
               if X.R1 ∧ ¬X.C1 then {D1 ← DATA1; S1 ← true}}

              {if ¬S0 then load D0 from core location A;
               if ¬S1 then load D1 from core location A+1}

              {DATA0 ← D0; X.R0 ← true; load core location A from D0; X.C0←true
               DATA1 ← D1; S.R1 ← true; load core location A+1 from D1;
               X.C1 ← true}
               X.IP ← false
        end
```

After a register is SELECTed, the core module high speed register M is initialized. If the value of the locations A and A+1 are correctly represented in the fast registers X, but not in core the fast register is used to load M. Loading of M is completed using core values (readout is destructive). M is used to update core and the fast register X.

REGISTER ASSIGNMENT ALGORITHM

We now consider the fast register assignment algorithm. When a memory request specifies an ADDRESS not currently assigned to a register, an attempt is made to find a register to be assigned to the ADDRESS*. The assignment

*ADDRESS is 19 bits in length and the ADDR field is 15 bits in length. The least significant bit in ADDRESS selects half of a double word. The next three least significant bits select one of the eight core modules and thus it is the remaining 15 most significant bits which are compared to ADDR in a fast register for assignment purposes. We will write this ADDRESS[1:15].

named ASSIGN
algorithm, cyclically scans the registers up to three times searching for a register to assign. The only registers to be considered are free registers, to ADDRESS.
i.e. those for which FREE = $\overline{IP} \wedge \overline{CORETRANS} \wedge \overline{H}$ is true. Stated more intuitively, a free register is one which is not involved in the current core transfer ($\overline{IP}$), is not a candidate for a core transfer ($\overline{CORETRANS}$) and for which no deferral has been specified ($\overline{H}$). The assignment algorithm reads and sets the cyclic SCAN pointer (also used by the inboard selection algorithm). The assignment algorithm can be functionally described by:

```
boolean procedure ASSIGN =
  begin local I, J;
      if     ∃X, X.ADDR = ADDRESS[1:15]
           then      (Y ← X; return true )
           else begin
                  for I ← 0 to 5 do begin J ← (SCAN+I) mod 6;
                                          if  J.FREE  ∧ J.Q*  then FOUND(J)
                                    end
                  for I ← 0 to 5 do begin J ← (SCAN+I) mod 6;
                                          if  J.FREE  then (SCAN ← (J+1) mod 6;
                                                            FOUND(J))
                                    end
                  if REQUEST.Q ∧ CPRIORITY = high
                     then for I ← 0 to 5 do begin J ← (SCAN+I) mod 6;
                                                  if J.Q ∧ J.IP ∧ J.STORE
                                                     then FOUND(J)
                                                end
                  return false
      end

macro  FOUND(X) =
      begin  Y ← J;  J[1:76] ← 0;  J.C0 ← true;  J.C1 ← true;

           return true
      end
```

*Recall that the Q bit indicates that a word in that register was last used by a sequential device (e.g. a drum). If the hold bit H is not set, then the register is a prime candidate for assignment since the sequential device which last used it will probably not do so again for a long while.

The algorithm first tries to locate a register which was used temporarily by a sequential device (FREE ∧ Q is true), for it is unlikely to be used again. This increases the probability that a word recently referenced by a non-sequential device (like a CPU) will remain in a register for awhile. The algorithm insures that Q requests, made almost exclusively by disks and drums, will almost always be assigned a register. The algorithm takes cognizance of high priority (as opposed to low or medium priority requests) from sequential devices to prevent drum or disk overrun. The algorithm will not reassign a register with the hold bit set unless overrun is impending (as indicated by CPRIORITY = high).

The mainline outboard algorithm responds to requests. There are six types of requests encoded in the F, S and H bits:

| Type of Request | F | S | H |
|---|---|---|---|
| fetch | 1 | 0 | 0 |
| fetch and hold | 1 | 0 | 1 |
| prefetch | 1 | 1 | 1 |
| store | 0 | 1 | 0 |
| store and hold | 0 | 1 | 1 |
| reserve | 0 | 0 | 1 |

We do not treat any other combinations here.

We now define the mainline outboard algorithm with more precision. Each clock cycle (100 nanoseconds), the outboard logic functionally* executes:

*Note that we state that the program functionally describes the corresponding logic. No promises of precise implementation are to be inferred.

```
procedure OUTBOARD =
  begin local T;
    if ADDRESS in range of this module
      then
        if ASSIGN then begin Y.ADDR ← ADDRESS[1:15]; Y.H ←      REQUEST.H;
                             Y.Q ←      REQUEST.Q; if CPRIORITY>Y.P then Y.P←CPRIORITY;
                             ACCEPT ← true;
                             case
                                prefetch request:  SATISFY ← true;

                                store request:   (T← ADDRESS mod 1;
                                                  Y.DATA_T ← DATAIN;  Y.R_T ← true;
                                                  Y.C_T ← false;  SATISFY ← true)

                                fetch request:   (T← ADDRESS mod 1;
                                                  if Y.R_T then (DATAOUT← DATA_T;
                                                                 SATISFY ← true)
                                                       else (Y.H ← true; SATISFY ←false)

                           end
                           else ACCEPT ← false
  end
```

The outboard logic invokes ASSIGN to find an existing or new register to assign to ADDRESS. If ASSIGN is successful the outboard logic updates the status information in the assigned register Y from the request.

For example, the register priority P should reflect the highest priority ascribed to it by either the current or a previous request.

Setting the hold bit in the fast register to have the same value as REQUEST.H is sufficient to handle the 'hold' portion of fetch and hold, store and hold, and reserve. Note that if a fetch request fails, the hold bit is set so that when the request is repeated, the register will not have been reassigned and the data will probably be available in the fast register. The remaining three types of requests are handled separately. Note again that though the memory request was not REJECTed in the last clock cycle, it may still fail to be honored in two ways: failure of ASSIGN to find a register in which case the request is not ACCEPTed and failure to find data to be fetched correctly recorded in the fast register in which case the request is not SATISFYed.

SEQUENTIAL DEVICES

Drums and disks generally access memory sequentially. Consequently they can accurately predict and issue prefetches or reserves far ahead of the actual operations. This permits the fast memory to schedule core transfers to minimize interference with transfers related to other requests.

For the purpose of allocation and scheduling of core transfers there exist three priorities: low, medium and high.* A fourth priority, warning, is used by processors and controllers to forewarn a memory module of an imminent high priority request. A warning is treated as a low priority except that when a warning priority request is accepted by a module any warning priorities already present are changed to high priorities. This tends to reduce the frequency with which a module has multiple high-priority requests to process, reducing the number of times a high priority fetch or store cannot be satisfied. Again it is critical that drum requests, in particular, be satisfied to prevent overrun. The dynamic priority scheme allows processors to make low priority requests which will not interfere in the more important high priority requests, but to increase the priority when response becomes critical.

EPILOGUE

The fast register memory has been implemented in the BCC500 running routinely since 1972. Unfortunately due to the press of finances, only two fast registers are connected to each of the eight core modules. Experience shows that . . .

---

*A memory request carries two types of priority PPRIORITY to be used to adjudicate port entry conflicts and the core request priority (CPRIORITY) we are now discussing.