General-Purpose Memory

Philip C. Treleaven

University of Reading

ABSTRACT

This document discusses how the memory of a fifth generation computer can be made more "general-purpose", to better meet the information representation requirements of very high level languages and operating systems. Three important mechanisms are examined: structured memory, addressing, and the communication of data. The proposals are for basing: memory on nested, variable-size memory cells; addressing on a contextual address space; and communication of data on both shared memory and message passing. For these mechanisms both design issues and implementations are discussed.

INTRODUCTION

If we examine the memory i.e. information requirements of modern high level programming languages and operating systems, we find much commonality. In these systems, memory is viewed as structured (ex. variables, arrays, lists, files, directories, pages, segments); addressing is contextual (ex. descriptors); and data is communicated both by shared memory (ex. code, file store) and message passing (ex. parallel processes, input/output).

At the present time these system concepts must be simulated in software on top of the von Neumann computer mechanisms:

memory	vector of fixed-sized memory cells
addressing	one-level address space
communication	shared memory

For the fifth generation computer it is clearly appropriate to implement structured memory concepts in the fundamental hardware mechanisms of the architecture. This will give a direct representation of the information structures encoded in high level languages and will transfer many of the mechanisms normally found in operating systems into the computer's architecture. In turn this should lead to more efficient representation and execution of programs. We therefore propose the following fifth generation computer mechanisms:

memory	nested, variable-size memory cells
addressing	contextual address space
communication	shared memory + message passing

The memory in such a computer allows each memory cell to contain a vector of subsidiary memory cells. For instance, a vector of the numbers 0 to 9 can be represented by one memory cell containing the vector, and subsidiary cells containing the individual numbers:

```
 ------------------------------------------------------------
 ¦ ---   ---   ---   ---   ---   ---   ---             ---  ¦
 ¦ ¦ 0 ¦ ¦ 1 ¦ ¦ 2 ¦ ¦ 3 ¦ ¦ 4 ¦ ¦ 5 ¦ ¦ 6 ¦   . . .    ¦ 9 ¦ ¦
 ¦ ---   ---   ---   ---   ---   ---   ---             ---  ¦
 ------------------------------------------------------------
```

while the statement "a=(b+1)*(b-c)" can be encoded as the instruction:

```
 ----------------------------------------------------
 ¦ ---   ------   ------   ---                      ¦
 ¦ ¦ * ¦ ¦+ b 1 ¦ ¦- b c ¦ ¦ a ¦         . . .      ¦
 ¦ ---   ------   ------   ---                      ¦
 ----------------------------------------------------
```

Addressing is based on each memory cell being considered a context and each subsidiary cell having a unique "selector" within this context:

```
 -       ------------------------------------------------      -
 |    |     ---     ---     ---              ---     |    |
 | 3: | 1:| 0 | 2:| 1 | 3:| 2 |    . . .   10:| 9 |  | 4: |
 |    |     ---     ---     ---              ---     |    |
 -       ------------------------------------------------      -
```

Thus within the surrounding context the address "3" may be used to access the whole vector of numbers while the address "3/2" allows access to the cell containing the number "1".

Lastly there is communication of data. In traditional computers two operations may be performed on the contents of a memory cell; STORE and LOAD. STORE overwrites the contents of the accessed cell, and LOAD takes a copy of the cell's contents. This supports the shared memory semantics. To support message passing, and to integrate it with shared memory, two additional operations may be performed on the contents of a memory cell: PUT and TAKE.

```
             shared memory                 message passing

              STORE addr                      PUT addr
                  |                              |
               -------                        -------
       addr:  |       |              addr:  |"empty"|
               -------                        -------
                  |                              |
              LOAD addr                      TAKE addr
```

PUT may only store into an "empty" memory cell and TAKE may only remove a non-empty contents, replacing it with "empty". Both operation may be viewed as polling a memory cell until the cell is in the correct state. (Note STORE and LOAD operations are unaffected by the "empty".)

Below we examine the design and implementation of the memory, addressing and communication mechanisms in more detail.

MEMORY

Memory in the computer consists of a nest organisation of variable-size memory cells. Such a memory could be implemented by traditional LISP nodes; however we believe that delimited strings represent a more idealised implementation. To justify this position, it is necessary for us to examine the conceptual role of an address in a computer, which is to bring together an operation and operand. When an operation and operand are literal values (e.g. +, 1) they are placed physically adjacent, since this is the most efficient representation. Similarly the elements of a vector are usually placed in consecutive memory cells. However where an operand (or an operation if it is a function body) is used in a number of places and it is not physically possible to place the operand adjacent to all its operations, then the address of the operand is used as a "place-holder". At runtime the address must be traversed, incurring an additional overhead.

In summary, LISP cells build all structures from addresses, whereas delimited strings allow nesting of structures and only use addresses when structures cannot be placed physically adjacent. Thus delimited strings provide a more direct representation of information structures and therefore should provide a more efficient implementation.

A delimited string (a variable-size memory cell) consists of two alphabets of characters, namely (i) delimiting characters left bracket "(" and right bracket ")", and (ii) data characters binary "0" and "1". Thus the above array of numbers 0 to 9 is represented as:

((0) (1) (2) (3) (4) (5) . . . (9))

In the computer this can be implemented using two bits per character:

<u>character</u>	<u>bit pattern</u>
(	10
)	11
0	00
1	01

Thus the array is encoded as:

```
10 10 00 11 10 01 11 10 01 00 11 10 01 01 11      10 01 00 00 01 11 11
(  (  0  )  (  1  )  (  1  0  )  (  1  1  )  ... (  1  0  0  1  )  )
```

and the physical memory may be viewed as consisting of two bit words.

ADDRESSING

Addressing of information in the computer is based on contextual address space in which each delimited string (i.e. memory cell) is considered a context. Relative to this context a related delimited string is identified by a selector. An address is a sequence of selectors specifying a path from the point of reference in the structure to the target memory cell. For instance to access the whole of the above array from elsewhere in the surrounding context its selector, say "3", is used, whereas to access a subsidiary number "100" the address "3/5" is used:

```
      3:( 1:(0) 2:(1) 3:(10) 4:(11) 5:(100) ... 10:(1001) )

3   =        (0)   (1)   (10)   (11)   (100) ...   (1001)

3/5 =                                  100
```

We assume that these address selectors are not explicitly represented in the information structure but are obtained by counting delimited strings from the left. Basically three types of selector are required for traversing contexts "{ }", which we refer to as:

Type	Before	After
direct 2	({ 1:( ) 2:( ) ... })	(1:( ) 2:({ }) ...)
superior 0	(1:( ) 2:({ }) ...)	({ 1:( ) 2:( ) ... })
expression (1)	({ 1:(2) 2:( ) ... })	(1:(2) 2:({ }) ...)

An address is encoded as a delimited string, with each selector being encoded as a subsidiary delimited string, for example "((3)(5))" which is "((11)(101))". Each selector moves the address from the current context to a new context. The three types of selector have the following meaning. "Direct" is an integer such as "2" specifying directly a delimited string in the current context "2:( )" counting from the left. "Superior" represented, for example by the selector "0", specifies the new context as the one surrounding the current context. Lastly, the exact encoding of the "expression" selector depends on its intended meaning: an "indirect" selector might be distinguished as "(...((1))...)" and define "(1)" as the address of a memory cell containing the required selector, whereas a selector computed by an instruction could be encoded as "(...((+)(1)(2))...)" and defined as returning as its result the required selector.

COMMUNICATION

Communication of data in the computer consists of both shared memory and message passing, the two fundamental forms of communication found in computing. In the past, computer architectures have either supported only one of these forms or have implemented them as disjoint mechanisms. However shared memory and message passing communications have complementary advantages and disadvantages for the representation and execution of programs. For instance shared memory supports program sharing and filestores, while message passing supports synchronised communication between processes and input/output. It is our belief that these forms of communication should be unified in a fifth generation computer.

As discussed above, shared memory and message passing communication is supported by four system-wide operations: STORE, LOAD, PUT, and TAKE. At present the semantics of these operations have been kept purposely close to the memory operations of traditional computers:

```
        shared memory          message passing

          STORE addr             PUT addr
              |                      |
      addr:(     )          addr:(      )
              |                      |
          LOAD addr              TAKE addr
```

However PUT and TAKE could be easily redefined to operate on a list, with PUT appending to the list and TAKE removing the first element:

```
                                        PUT addr
                                            |
                      addr:( ( ) ( )  ...  ( ) ( ) )
                              |
                          TAKE addr
```

Next, before we examine the implementation of memory, addressing and communication, we will briefly discuss the representation and execution of programs.

PROGRAMS

The concepts of nested, variable-size memory cells and contextual addressing can be easily used in conjunction with both low-level and high-level instruction formats. For example, traditional von Neumann instructions may be encoded as:

```
          ( (operator) (modes) (operand) (operand) ... )
                ¦       ¦ ¦                  ¦
         + etc.-       xy xy                 ¦
    shared memory x=0 -¦¦- y=0 literal       ¦- data
  message passing x=1 -   - y=1 address       - data address
```

By allowing the modes to qualify the operation, as well as the operands, it is possible to have built-in program calling:

```
                     ( (modes) (operation) (operand) (operand) ... )
                        ¦ ¦          ¦          ¦
                       xy xy         ¦          ¦
    shared memory x=0 -¦¦- y=0 lit.  ¦- operator¦- data
  message passing x=1 -   - y=1 addr. - code     - data
                                        addr.      addr.
```

Lastly, by extending the modes it is easy to obtain a high level format with nested instructions:

```
                     ( (modes) (operation) (operand) (operand) ... )
                        ¦ ¦          ¦          ¦
                       xy xy         ¦          ¦
    shared memory x=0 -¦¦- y=00 lit  ¦- operator¦- data
  message passing x=1 - ¦- y=01 addr.¦- code    ¦- data address
                        ¦            ¦  addr.   ¦
                         - y=11 instr - instr.   - instruction
                                       (returns   (returns operand)
                                        operation)
```

The reader may be interested to note, that since the computer accesses bits from the left, if it is to perform bit-serial arithmetic (which may be the most efficient), then the least significant bit of a number must be on the left.

Next we will examine implementation of the general-purpose memory.

IMPLEMENTATION

An essential concept in the implementation of the general-purpose memory, particularly in a multi-computer environment, is the direct functional correspondence (at the higher levels) between the physical system and the logical information structure of a fifth generation computer system:

<u>hardware</u>

```
      1:             2:             3:
  ---------      ---------      ---------
 |communicn|<-->|communicn|<-->|communicn|<-->
 |---------|    |---------|    |---------|
 |processor|    |processor|    |processor|      . . .
 |---------|    |---------|    |---------|
 |memory   |    |memory   |    |memory   |
  ---------      ---------      ---------
```

<u>software</u>

```
   1:(       )    2:(       )    3:(       )      . . .
```

In the system the memory of each computer is an implicit delimited string whose selector is the explicit hardware selector of the computer. Inside a memory, delimited strings are explicitly represented and are dynamically manipulated, but the selectors are implicit. (Thus computers in a fifth generation system are analogous to traditional memory cells, and may be allocated and used in a similar way.)

We will now examine the implementation of the general-purpose memory concepts, initially in terms of a single computer and then in a multi-computer system. In these implemntations the following assumptions are made:

1. it is relatively easy to detect unused space between brackets
   ")x...x(" or ")x...x)"
   . to ignore unused cells
   . to allocate unused cells
2. by storing locally the physical address of a delimited string
   . accesses can be optimised
   . physical addresses can be remapped, when strings are relocated

In a single computer the memory is a vector of 2-bit cells, each with a unique physical address, that stores the delimited string information structure. Access to the information is via registers that hold the physical address of the corresponding cell. When a logical address is processed its sequence of selectors is translated into the physical address of the cell containing the left bracket of the delimited string:

```
                     - s1.s2.s3...  address
                     |
                 ---------
                |    8    |
                 ---------
                     |
           6:  7:  8:  9:
            )   )   (   (  . . .  )     )
```

The appropriate cell is located by the memory controller scanning the delimited string structure from the left.

The main difficulty in accessing a delimited string is that STORE and PUT operations may cause a string to expand and hence require the remapping of the memory. A number of possible strategies may be adopted, for instance firstly all empty space can be relocated adjacent to the delimited string before it is accessed, or secondly empty space is assumed to be adjacent to the accessed string (from the previous contents) and memory is only relocated when this is no longer the case.

We will adopt the second strategy because it appears to be more efficient. The initial task of the memory controller is to locate the target delimited string and to create the string if it does not exist. Then the four memory operations are implemented as follows:

LOAD -	a copy of the contents of the delimited string is taken
STORE -	the contents of the old string are deleted and then the new contents are inserted
TAKE -	if the delimited string is empty then reject the request otherwise remove the contents, replacing it by an empty string
PUT -	if the delimited string is non-empty then reject the access otherwise insert the new contents

Finally, we examine the support of the general-purpose memory in a multi-computer environment. As discussed above, each computer in the system behaves as a delimited string with a unique selector. (This is particularly important for programming since each computer system may be programmed as a single computer and accessed using its address as if it is a single memory cell.) As in the single computer, access to a memory is via the local registers, a register being allocated for the duration of each access.

A central problem in the design of the multi-computer system is the choice of communications mechanisms, made difficult by the variable-size information structures that must be passed. Two sets of decisions must be made, firstly whether to use shared buses or point-to-point connections; and secondly whether to pass an information structure as an atomic unit of contiguous packets or as packets interleaved with other structures.

A shared bus (or buses) has the advantage of providing rapid communication, but the disadvantages of allocation and extensibility in a decentralised system. Point-to-point connections have the advantages of localised allocation and extensibility, but the disadvantages of slower communications. For the second decision, of how to pass an information structure, the atomic unit has the advantage of reducing the control information that must be associated with a structure, but the disadvantage of monopolising the communications. In contrast, interleaving many structures has the advantage of distributing the usage of the communications, but has the disadvantage of requiring additional control information in packets, such as source and destination addresses in each packet.

The approach we will investigate is analogous to a token passing ring; point-to-point connections on which atomic structures are sent as contiguous packets started and terminated by empty packets:

Communications

```
   ---      ---------      ---------      ---------
      |-->|          |-->|          |-->|          |-->
      |<--|          |<--|          |<--|          |<--
      |    |processor|    |processor|    |processor|
```

Atomic message

```
  -----  -----  -----  -----  ----- ... -----  -----  -----
 |empty||(destination)(source)(operation) (operand) ||empty|
  -----  -----  -----  -----  ----- ... -----  -----  -----
```

In such a system, each connection consists of 5 five wires:

```
        request          <----  |
        acknowledge      ---->  |   control
        empty/data       <----  |

        character        <----  |   data
        ex. ( ) 0 1      <----  |
```

The "request" and "acknowledge" lines are used to transfer a packet between adjacent computers, and the "empty/data" indicates whether the packet is empty or contains a character "(", ")", "0" or "1". A computer may transmit an information structure when it receives an empty packet, by delaying the stream of packets from its neighbour computer. When a computer receives a non-empty packet, immediately following an empty packet, it must check the destination to see if it must service the packet.

Two difficulties remain. The first is to ensure that the pattern of messages cannot cause the communications to "deadlock", as for example when one message needs to over take another (but is physically unable to) so that execution may proceed. This should be achievable by ensuring that only one message may be issued (and must be completed) at a time by an instruction. The second is the handling of rejected PUT and TAKE accesses. This can be supported by requiring an acknowledgement for every access.

OPERATION

To illustrate the operation of the general-purpose memory and the communications, let us consider a request from computer "1" to "LOAD" the contents of the subsidiary delimited string "5" stored in an array in computer "3":

<u>Computer 3:</u>

<u>logical</u>

```
1:      2:        3:           4:           5:          6:
 ( 0 )  ( 1 ) X ( 1 0 )    ( 1 1 )     ( 1 0 0 )   (
```

<u>physical</u>

```
0: 1: 2: 3: 4: 5: 6: 7: 8: 9: 10: 11: 12: 13: 14: 15:
 10 00 11 10 01 11 00 10 01 00  11  10  01  01  11  10 01 00 00 11 10 ...
```

To achieve this, computer 1 sends the following message:

```
     -----  ------- ... ------  -----
    |empty||((3)(5)) (1) (LOAD)||empty|
     -----  ------- ... ------  -----
                |      |      |
  destination -     source     - operation
```

where the logical address "((3)(5))" is translated into the physical address "15", held in a local register. (Notice that physical location "6" contains an empty field "X" which is ignored in locating the target delimited string.) Then computer "3" acknowledges "ACK" the "LOAD" by generating the following reply for computer "1":

```
     -----  ------- ... ---------  -----
    |empty||((1)) (3) (ACK) (100)||empty|
     -----  ------- ... ---------  -----
               |     |    |      |
 destination -  source    |       - operand
                       operation
```

CONCLUSIONS

For a fifth generation computer to better meet the information representation requirements of very high level languages, we have argued for basing its memory on nested variable-size cells (i.e. delimited strings), its addressing on a contextual address space, and its communication of data on both shared memory and message passing.

Although the structured memory concepts seem correct, their implementation require significant further investigation. In particular managing the delimited strings memory and the passing of delimited strings over the communications network requires careful study and analysis.