DESIGN OF A FULLY VARIABLE-LENGTH STRUCTURED MINICOMPUTER

Z. G. Vranesic
V. C. Hamacher
Y. Y. Leung
Departments of Electrical Engineering and Computer Science
University of Toronto
Toronto, Canada

ABSTRACT

Binary-based and fixed-length structure computers are often inconvenient and wasteful of resources. In this paper we present a design for a fully variable-length structured minicomputer. Since all parameters (instructions and data) are unrestricted in length, their boundaries and interpretation are effected by special delimiter codes. For practical reasons (dictated by current technology) the machine utilizes a binary-coded decimal number representation.

I. INTRODUCTION

Present day digital systems show a prevalence of binary, fixed-length structures. This is dictated by the technological ease of implementation, low cost and high reliability. Yet there are a large number of applications where the binary base and fixed-length organization are inconvenient and often wasteful of resources.

Decimal and variable-length data have been implemented in differing degrees from the IBM 1620 era [1] to one of the latest minicomputers, the CIP/2200 [2]. However, most of these machines have achieved these features in an "added-on" fashion in a structure that mainly offers conventional binary, fixed-length operations. The recently reported B1700 computer [3] reflects an attempt to get around the difficulties imposed by fixed-length constraints by providing a highly flexible, reconfigurable structure, where specific lengths may be defined as run time parameters.

In this paper we propose the design of a relatively small-scale decimal, variable-length machine whose structure evolves solely from those two features. In a sense, the work reported here can be interpreted as an elaboration or feasibility study on some conjectures made recently by Foster [4] concerning the architecture of the average computer of the year 2000. The design study described in more detail in the following sections in fact supports the feasibility of the basic concept even in terms of present day technology.

II. MACHINE ORGANIZATION

In order to provide the variable-length characteristic for data, OP-codes and addresses, it is necessary to employ some "length delimiters". Thus it is apparent that a truly binary machine could not be constructed to meet such requirements, since the range of available digits (0,1) leaves no spare codes which could serve as delimiters. Hence we must turn to a higher base system, which for practical reasons might be binary coded.

Our choice is the decimal system with binary coded implementation. This provides us with six codes (other than 0-9) for use as delimiters. We will call them $D = \{\alpha, \beta, \gamma, \delta, +, -\}$.

The machine has a random-access memory with the capacity of 100,000 digits, addressable to the digit (0-99,999).

NUMBER AND CHARACTER REPRESENTATION

In order to represent real numbers of the form N x 10^e, both N and e are expressed as a sign followed by a 10's complement value. The exponent e is stored first, followed by the significant digits N, both number fields occurring low-order digits first.

For example +318.27 x 10^{-12} is represented and stored in memory as -68+72813*, which is equivalent to 31827 x 10^{-14}. The decimal point is always implied at the low order end of N. Note that * could be any delimiter.

Integer form is used for addressing purposes only (e = 0 and it is not shown explicitly) and it is recognized as such from the context of instructions. We will refer to {±} followed by a string of digits as a number field or address, depending on the context, and use the name number or real number to refer to two successive number fields.

It is important to observe that the low-order digits are stored first, because the arithmetic unit must be at least partly serial, to enable it to handle arbitrarily long numbers.

The ASCII character set is represented directly using 2 4-bit digits per character. Any two digit delimiter not in the ASCII set, referenced in this paper as ◫, is used as the character string delimiter. In general, the address of a character string, number, address, or instruction is the address of the leading delimiter, since this delimiter usually describes some property of the information to follow.

INSTRUCTION SET

The machine has thirty-one instructions, including four rudimentary I/O instructions. Instructions are delimited by α. An unsigned decimal opcode follows the leading α and its end is indicated by any single digit delimiter. All instructions except HALT have an operand list which in some cases is preceded by a parameter K. The delimiter δ indicates that K is present, and the possible values for K are integers equal to or greater than 0. The K value may be referenced by any of the addressing modes of Table 1. The interpretation of the delimiter set when used to separate items of the operand list is shown in Table 1. Table 2 gives the complete instruction set. In instructions where the parameter K is called for, it may be omitted if K = 1; there being no ambiguity, since A can never be immediate data. The number of operands is variable in ADSB and MUDI instructions.

A few examples should clarify the appearance in memory of complete instructions, and give an idea of instruction execution.

(i) αMVNδ2+419δ-97+32α
This instruction inserts the real number $23\text{x}10^{-21}$, represented by 2 number fields, into the memory starting at the address 914; while αMVNδ4+419-0001α moves the four number fields (2 real numbers or 4 addresses) starting at the address stored at location 1000 (indirect mode) into 914. If the number field (address) stored at 1000 has a "-" leading delimiter, then another level of indirection is indicated.

(ii) $\alpha\text{CPC+001}\delta\underbrace{D_1\ D_2}_{C_1}\underbrace{D_3\ D_4}_{C_2}\ldots\underbrace{D_{n-1}\ D_n}_{C_{n/2}}\ \square\square\ \alpha$
compares the character string $C_1C_2\ldots C_{n/2}$ with the one stored at 100 and sets the 2-bit condition vector in the CPU to 00 if they are equal, and to 11 if they are not; while αCPC+001γ456+27α

TABLE 1
Interpretation of delimiters in instructions

Delimiter	Addressing Mode or Function	Operand Form
+	direct	unsigned integer address.
-	indirect	unsigned integer address.
γ	indexed	unsigned integer address (the location of the index), delimited by + or - indicating a direct or indirect address to follow.
δ	immediate	number, address or character string, appropriately delimited.
α	instruction delimiter	
β	operation change (indicates SUBTRACT instead of ADD and DIVIDE instead of MULTIPLY in the ADSB and MUDI instructions.)	

TABLE 2
Instruction Set

ØP CØDE	ØPERAND LIST	DESCRIPTION
MVN	K,A,B	Move number fields (up to Kth delimiter) from B to A
MVC	K,A,B	Move character strings (up to Kth delimiter) from B to A
MVD	K,A,B	Move K digits from B to A
CPA	A,B	Compare addresses at A and B
CPN	A,B	Compare numbers at A and B
CPC	A,B	Compare character strings at A and B
CPD	K,A,B	Compare digit list at A and B
AND	K,A,B,C	Logical "AND" of K digits at B and C into A
ØR	K,A,B,C	Logical "ØR" of K digits at B and C into A
CMPL	K,A,B	Logical "complement of K digits at B into A
TRCT	K,A	Truncate number at A to K significant digits
CLR	K,A	Clear K digits starting at A
MVPN	K,A	Move pointer at A over Kth number field delimiter
MVPEN	K,A	Move pointer at A to the end of the Kth number field
MVPC	K,A	Move pointer at A over to Kth character string delimiter
MVPEC	K,A	Move pointer at A to the end of the Kth character string
ADSB	A,B,C,...	Add/Subtract B,C,... and put in A
MUDI	A,B,C,...	Mult./Div. B,C,..., and put in A
ADSBA	A,B,C,...	Add/Subtract addresses (single number field)
CMPN	A,B	9's complement of the number field starting at B into A
BRZ	A	conditional branches to A: Br zero
BRNZ	A	conditional branches to A: Br nonzero
BRN	A	conditional branches to A: Br negative
BRP	A	conditional branches to A: Br positive
BR	A	Unconditional branch to A
JMPS	A	Subroutine linkage to A
HALT		stop
IN	K,A	I/O w.r.t. device K;
ØUT	K,A	transfer 16 bits of data
BIN	K,A	I/O w.r.t. device K;
BØUT	K,A	transfer 8 bits of data

compares the character string at [[654]+72] where [...] indicates "the contents of".

(iii) αMVPNδ21+2α takes [2] as an address and, assuming [2] points at a number field delimiter, increases the value of [2] until it points at the 12th number field delimiter from the starting point. This would allow [2] to now point at the 6th number down the list from the starting number. This provides the means for accessing variable length number or character strings in a list of such items where the programmer knows explicitly only the address of the first item.

In the above examples, we have used appropriate mnemonics for the OP-codes, but they are actually specified in memory by unsigned integer codes.

III. HARDWARE DESIGN

Since all parameters may be variable in length, a fully parallel design of the machine cannot be achieved. It is apparent that serial by digit structure would be the simplest solution in terms of hardware costs. However, in order to attain a reasonable processing speed, some degree of parallelism must be introduced.

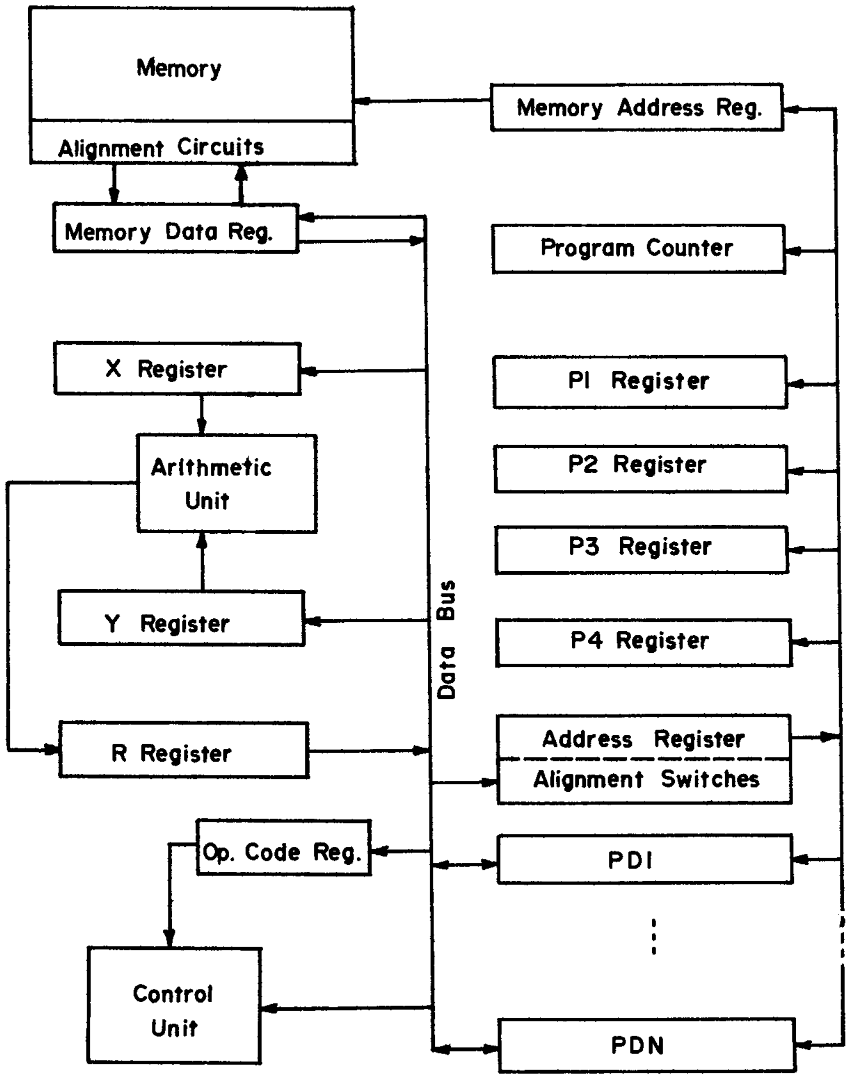
In our prototype design we have chosen serial processing of four-digit (16 bits) blocks of data. Figure 1 shows the block diagram of the machine.

Memory has a 16 bit word length and its addressing is arranged in a 4 x 25 x 1000 digit pattern, giving a total capacity of 100,000 digits. It is digit-addressable, necessitating two internal read cycles if the address is not 0 or divisible by 4. In order to avoid alignment difficulties on the data bus and in the 4-digit parallel arithmetic unit, the memory includes alignment circuits so that the memory data register always contains the addressed digit plus the three digits that follow. Thus it is not necessary to impose any boundary alignment conditions on the programmer for storage of data in the memory.

Internal sequencing and serial control of instruction execution when the operand length exceeds 4 digits is regulated by the pointer registers P1, P2, P3 and P4, each being a 5-digit counter-register.

All addressing is carried out via a 5-digit address bus. Since addresses are obtained directly from instructions they are not necessarily correctly aligned on the data bus. This is remedied by assembling all addresses in the address register which

FIGURE 1
Block Diagram



includes the required alignment switches.

Peripheral devices PD1,..., PDN are addressed through the low order digits of the address bus, with data transfer handled by the data bus.

IV. PROGRAMMING CONSIDERATIONS

Consistent with the theme of Foster's [4] brief sketch of the average computer of the year 2000 which "... will-be a monoprocessor doing its own I/O, - most probably be privately owned and monoprogrammed, - be an interpretive engine capable of executing directly one or more high-level languages...", it is claimed that although we do not interpretively execute several high-level languages, the instruction set of Table 2 makes possible efficient processing on a "one-shot" basis of relatively small user programs. This is a reasonable goal for a small, general, privately owned and monoprogrammed computer in any event. The efficient processing we mentioned above is from the programmer standpoint. This means that the machine language, represented in some assembly form, should have instructions and formats that make coding of normal problems somehow natural and concise.

MATRIX MULTIPLY ROUTINE

We first present a complete program to multiply two matrices of real numbers. All matrix entries are of variable length, so normal indexing would not work on any machine, and the equivalent program in a fixed word length structure would be somewhat clumsy and unnatural.

The program performs the computation

C = A x B where A is ID rows by JD columns,
B is JD rows by KD columns,
and C is ID rows by KD columns.

Matrices are stored in column order and the program variables for the matrix dimensions are the same as above.

Assuming that the matrices A and B have been loaded in core and ID, JD, and KD have been appropriately initialized, the program is:

```
        ADSBA II←ID+ID       ;increment step for PT1
        ADSBA K←-KD          to access successive
        MVN   JV←0           row entries.
        MVN   PT3←#C         ;load address of C into
KLØØP:  MVN   M←0            PT3.
        ADSBA I←-ID
ILØØP:  MVN   PT2←#B
        MVPN  JV,PT2         ;sets PT2 to appropriate
        MVN   PT1←#A         column of B.
        MVPN  M,PT1          ;sets PT1 to appropriate
        ADSBA J←-JD          row of A.
        MVN   2,'PT3←0·0     ;clear $c_{i,k}$(initial length
                             unimportant.)
JLØØP:  MUDI  TEMP←'PT1*'PT2 ;$a_{i,j}xb_{j,k}$
        ADSB  'PT3←'PT3+TEMP ;accumulate into $c_{i,k}$
        MVPN  2,PT2          ;move PT2 across 2 delimi-
                             ters to $b_{(j+1),k}$
        MVPN  II,PT1         ;move PT1 to $a_{i,(j+1)}$
        ADSBA J←J+1
        BRN   JLØØP
        MVPEN 2,PT3          ;move PT3 to next C entry
        ADSBA M←M+2
        ADSBA I←I+1
        BRN   ILØØP
        ADSBA JV←JV+JD+JD    ;sets B column accessing
        ADSBA K←K+1          variable.
        BRN   KLØØP
        HALT
```

In this program, and in the remainder of this section, we have used a suitable assembler notation for the parameter and operand lists for instructions. For example, #C refers to the address of C, and 'PT3 indicates indirect addressing through location PT3. There are no macro references; and there is a strict one-to-one correspondence between the lines in the program and machine instructions.

The previous example was concerned with arithmetic operations and array accessing. We now illustrate some aspects of non-numerical programming. The example chosen can be taken as a model of some aspects of symbol table manipulation in a language processor. The main idea here is to illustrate the ease of building and searching tables of variable length mixed data types.

SYMBOL TABLE MANIPULATION

A particular type of character string made up of ASCII symbols $\underbrace{A,B,\ldots,Z}_{<A>},\underbrace{:,;}_{<D>},\$$ is to be processed.

A syntactically correct string must start with a member of <A> and end with a member of <D> followed by $, with no other occurrences of $, and with all other occurrences of members of <D> isolated by members of <A>.

There are also some semantic rules that must be met. First, we need some definitions. Each occurrence of a member of <D> will be said to "terminate" the previous contiguous substring of members of <A>, and the class name <LABEL> will be used to describe any such contiguous substring of members of <A>. Now, a syntactically correct string is also semantically correct if all <LABEL>'s terminated by ":" are unique and any <LABEL> terminated by ";" also appears in the string terminated by ":".

Examples of correct and incorrect strings are:

(i) START:LØØP:CTR:LØØP;OUT:$ is both syntactically and semantically correct.

(ii) $\underline{\text{A:A}}$;SRCH$\underline{\text{:;}}$COMP:SRCH:$\underline{\text{A:}}$$ is both syntactically and semantically incorrect (see the underlined places).

The processing to be performed on these strings is as follows: Build a table in core of all unique <LABEL>'s with an address associated with each. If the <LABEL> **first** occurs terminated by ":", the address is provided from the contents of a word addressed as LØCCTR; otherwise, the 5-digit value 00000 is associated with the <LABEL>. This "dummy" address will be replaced by the correct value from LØCCTR when the <LABEL> later occurs terminated by ":".

There are two subroutines used in the program which we will list in detail. One, called TBLSCH, is used to search the table for the occurrence of the <LABEL> currently in 'BUFF. The locations TØP1 and BØT1 are pointers to the top and bottom of the table, respectively; and PT1 is a pointer location for accessing the table entries. On exit, put "Y" in ANS if the <LABEL> is found, and leave PT1 pointing at the lead delimiter for the matching <LABEL>; otherwise, put "N" in ANS. The routine is accessed from a JMPS instruction which puts the return address in the first location, TBLSCH, in the routine.
The coding is:

```
TBLSCH: DA    5              ;assembler command to
        MVN   PT1←BØT1       establish a 5-digit "return
        CPA   TØP1↔BØT1      field."
        BNZ   CHECK          ;go to CHECK if table non-
        MVD   2,ANS←"N"      empty.
        BR    'TBLSCH
CHECK:  CPC   'BUFF↔'PT1     ;compare  LABEL  in 'BUFF
        BZ    FØUND          with one in table.
        MVPC  PT1            ;move PT1 to start of next
        ADSBA PT1←PT1+2      <LABEL> in table
        MVPN  PT1
        ADSBA PT1←PT1+1
```

```
         CPA    PT1↔TØP1        ;has whole table been
         BNZ    CHECK           searched?
         MVD    2,ANS←"N"
         BR     'TBLSCH
FOUND:   MVD    2,ANS←"Y"
         BR     'TBLSCH
```

The second routine, called TBLINS, inserts the <LABEL> in 'BUFF onto the top of the symbol table and associates the address in PARAM with it. The pointer TØP1 is adjusted appropriately.

```
TBLINS:  DA     5
         MVN    PT1←TØP1
         MVC    'TØP1←'BUFF     ;add <LABEL>
         MVPEC  PT1
         MVD    2,'PT1←"□□";    ;insert character delimi-
         ADSBA  PT1←PT1+2       ter
         MVN    'PT1←PARAM      ;insert associated address
         MVPEN  PT1
         MVD    'PT1←'+'        ;insert number field deli-
         ADSBA  PT1←PT1+1       miter
         MVD    2,'PT1←"□□"     ;insert table-top delimi-
                                ter
         MVN    TØP1←PT1        ;adjust table-top pointer
         BR     'TBLINS
```

Although we have only presented two of the subroutines used in the complete program, the type of coding used at the assembler level for non-numeric processing should be evident. The complete program required 110 instructions, including the subroutine coding.

Due to the radically different structure, it is difficult to compare our machine with standard minicomputers. Meaningful comparisons will become possible only as a result of extensive experience with it. The machine was simulated and some interesting observations made. For example, the above matrix multiply routine was found to require 400 digits of storage with the delimiter density of 25%.

V. CONCLUSIONS

We have described the design of a fully variable-length general purpose computer. In order to assess the feasibility of such machines it is essential to take a close look at advantages gained and difficulties that might be encountered.

Based on a number of programs that we have written, it is apparent that programming presents fewer difficulties than one usually encounters with standard minicomputers.

Limits on computational accuracy, size of operand labels and data as well as the alignment requirements, are non-existent from the programmer's point of view by the very nature of the machine.

In order to determine the physical feasibility of such machines, we have completed the design on the basic circuit level (using standard TTL MSI components). As a result we have found that the hardware complexity and cost place the machine in the price range of typical minicomputers. Simulator runs have been used to verify the logical correctness and adequacy of the selected instruction set, as well as to obtain an evaluation of memory and cycle time requirements.

VI. ACKNOWLEDGEMENT

This research was partly supported by the National Research Council of Canada.

VII. REFERENCES

1 IBM Reference Manual, 1620 Data Processing System, IBM, 1960.

2 CIP/2200 Reference Manual, Cincinnati Milacron Company, Process Controls Division, Lebanon, Ohio, April 1972.

3 W.T. Wilner, "Burroughs B1700 memory utilization," Proceedings of FJCC, 1972, pp. 579-586.

4 Caxton C. Foster, "The Next Three Generations," Computer, Vol. 5, No. 2, March/April 1972.