Quaestiones Informaticae

Vol. 2 No. 1 November, 1982

Quaestiones Informaticae

An official publication of the Computer Society of South Africa
'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika

Editor: Prof G. Wiechers
Department of Computer Science and Information Systems
University of South Africa
P.O. Box 392, Pretoria 0001

Subscriptions

Annual subscriptions are as follows:

	SA	US	UK
Individuals	R6	$7	£3,0
Institutions	R12	$14	£6,0

Quaestiones Informaticae is prepared for publication by Thomson Publications South Africa (Pty) Ltd for the Computer Society of South Africa.

NOTE FROM THE EDITOR

After an absence of two years we are happy to announce that we are now in a position to continue the publication of Quaestiones Informaticae. The first Volume of QI consists of three numbers, and appeared during the period June 1979 till March 1980 under the editorship of Prof Howard Williams. Because Prof Williams took up a post at the Herriot-Watt University in Edinburgh, he had to relinquish his position as editor. The Computer Society of South Africa, which sponsors the publication of QI, appointed me as editor, whereas Mr Peter Pirow took over the administration of the Journal. The editorial board functions under the auspices of the Publications Committee of the CSSA.

The current issue is Number 1 of Volume 2. It is planned to publish altogether three issues in the Volume, with most of the papers coming from the Second South African Computer Symposium on Research in Theory, Software and Hardware. This Symposium was held on 28th and 29th October, 1981. At present it appears that most of the material published in this Journal comes from papers read at conferences. We invite possible contributors to submit their work to QI, since only the vigorous support of researchers in the field of Computer Science and Information Systems will keep this publication alive.

G WIECHERS

November, 1983

The Design and Microprogrammed Implementation of a Structured Language Machine

G. R. Finnie
Computer Science Department, University of Cape Town

Abstract

A computer architecture is described which is suitable for the execution of structured languages (such as Pascal or Algol) at the intermediate language level. The design is heavily stack oriented and consists basically of a dual processor with one processor dedicated largely to address mapping and stack maintenance. The structure has been implemented using microprogrammable bit-slice processors.

The stack organisation is described and their possible utilisation during process execution is examined. The use of a tag facility for memory is also considered.

Software development for the system includes a flexible micro-assembler and a program for the placement of micro-instructions in the two-dimensional address space of micromemory, both of which are briefly described.

1. Introduction

The increasing popularity of the structured programming languages coupled with the rise of structured programming as a useful concept for efficient program development has to some extent highlighted the disparity between the high-level concepts and the hardware constructs on which a process must be executed. In particular the Von Neumann architecture has few hardware facilities for the efficient handling of block structured languages such as Pascal, Algol or Coral 66.

This disparity has stimulated the design and development of machines suitable for the execution of such languages e.g. the Burroughs B5700/B6700 [12]. Wortman [18] has stated that the distinguishing characteristic of language directed computer design is a conscious effort to match the structure of a computer to the syntax and semantics of one or more programming languages in several ways:

(i) data structures used by programmers map easily onto data structures in the machine
(ii) control constructs in the language match control constructs in the machine
(iii) semantic errors in using the language are detected by the hardware.

High-level language computer architectures fall essentially into two classes: Direct execution architectures in which the source languages are executed relatively directly in source program form and intermediate language machines (syntax-oriented) which execute programs translated to some internal form for more efficiency of execution. The intermediate language is often some form of Polish code [5]. The Structured Language Machine (SLM) considered in this paper is designed for use at an intermediate language level.

2. Basic Architecture

The architecture of the SLM (Figure 1) is briefly described in this section. A more detailed discussion is available in [10].

The system has been built from microprogrammable two-bit slice processors (Intel 3000 series). It consists essentially of a 32-bit main processor for arithmetic/logical manipulation of data and an 18-bit slave processor for stack handling and addressing functions. Their operation is defined by a sequence of instructions from micromemory, the sequence being determined by the operation of a Microprogram Control Unit (MCU).

Both processors are under control of a single micromemory word. This dual processor organisation thus allows tightly coupled parallel operation of each processing unit with a corresponding increase in the macro-operation execution speed e.g. auto-incrementing of a stack pointer may proceed in parallel with the main processor operation. The processors are linked via their address, data and input buses and the transfer of information between them is under control of a micro-instruction field which organises the multiplexing operations.

The SLM uses a 45 bit microword which comprises 13 fields and controls all microfunctions and the I/O structure. The fields include the next micro-instruction address, function control for both processors, flag control, carry bit control and conditional clocking of the processors. The use of a masking bus (K) permits bit, byte, halfword or word addressing in the master processor.

The main memory is organised into 32 bit words with the probable addition of tag bits. The SLM is basically a one-address machine (with an 18-bit address field) with a number of zero-address operations which may be packed to one word.

The master processor and slave processor each contain twelve working registers of width 32-bits and 18-bits respectively. A number of these are used as scratchpad registers by the microroutines and the rest are used by the target machine for such functions as index registers, accumulators, loop control registers, stack pointers, etc. The latter functions are detailed in section 3.1.

3. Some High-Level Language Facilities

3.1 Stack Organisation

The SLM is intended for use with block-structured languages allowing recursion to any (practical) depth so that the natural construct which reflects the high-level structure is that of a stack. The system is based on five stacks manipulated by the microroutines.

These are: 1) Accumulator Stack
2) Index Register Stack
3) Loop Control Stack
4) Return Address Stack
5) Activation Record Stack

The concept of an activation record stack to define the current state of execution of a block structured process is well known [12]. This stack is used to link the static nesting of block or procedure activations. To some extent the components of an activation record stack may be considered as being separately

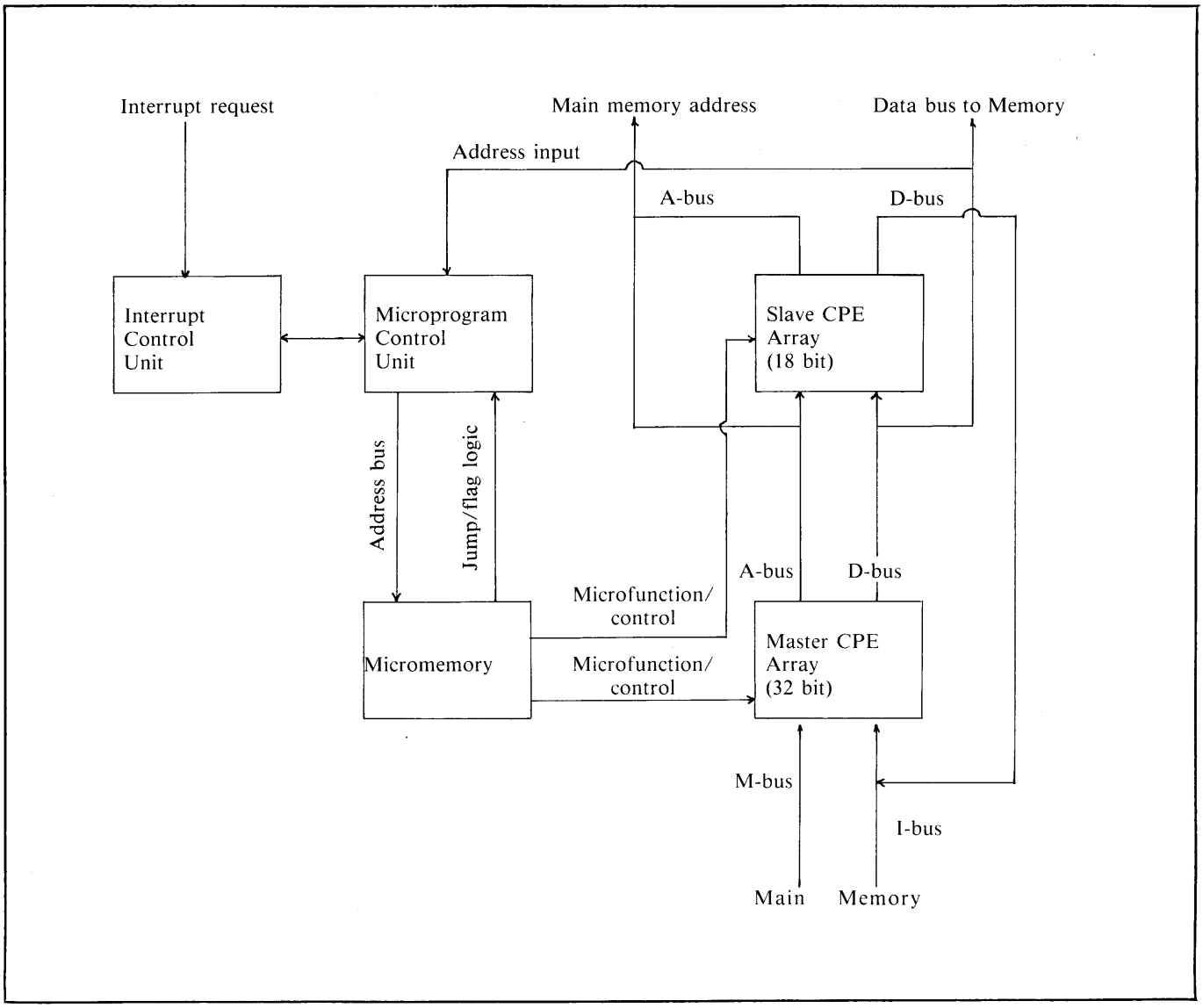

FIGURE 1

stack organised although their overall operation is defined by the movement of the activation stack i.e. all stacks involved most push or pop as the activation record stack. The first four of the above stacks may be considered as components of the activation record stack for any process in the system. The current top element of the activation record stack defines the initial point of the currently active blocks activation record. The second element of the return address stack contains the return of execution point in the enclosing process.

Most operations involving elements at the top of the stack include a default pop or push operation on the stack. Operations modifying the activation record stack involve resetting all other stack pointers to reflect the new status of the process. All stack pointers reside in the slave processor registers.

In a block structured programming language one may distinguish between the code portion and the value portion of a block which represent the program and the dynamic data storage respectively. A desirable feature of an architecture for such languages is one which facilitates the separation of the code and the data areas to allow pure procedures for re-entrant programs i.e. individual jobs must have different records of execution (execution stacks) while sharing the identical code segments. In addition the design should enable the efficient access of data elements defined within blocks containing the current block.

Block structured processes have the distinction that a value declared in any block enclosing the definition of the current block will have the value within the current block unless redeclared within this block. However, data defined within another block at the same or a lower level has no value within the current block. It is thus necessary not only to retain a stack of activation records defining the static nesting of block activations but it is also necessary to maintain a record of the accessing environment of the current block i.e. an indication of which values may be treated as defined for this block.

An example of this is illustrated in Figures 2-5.

The skeleton Pascal program in Figure 2 is used to show the sequence of procedure definitions. Figure 3 shows the logical state of the activation record stack immediately prior to calling procedure A at line 11. The dashed line indicates the access environment i.e. blocks whose data values will be defined within the current procedure and the solid lines indicate the nesting of procedure activations. Figure 4 indicates the logical state of the stack during execution of A. In this case the access environment pointer references only the EXAMPLE record since A is defined within this block.

In the SLM the nesting of process activations is controlled by the activation record pointer stack. Each element of this stack is a pointer to the base of an activation record. When a new procedure or block is entered a new activation record pointer is pushed onto the stack. The access environment control is handled by a field in the activation record defining the link to the base of the enclosing block. Addressing of data elements within a block may be defined relative to the start of the activation record. Figure 5 illustrates the detailed activation record structuring during execution of A. To facilitate fast access to most data structures a copy of the current environment base is held in a fast 18-bit register as is a copy of the global (outer-

```
1   program EXAMPLE (input, output);
2   const     m = 4 ; n = 3;
3   var       i:1..m ; j ; 1 .. n ;
4   procedure A ;
5             var x,y,z : real;
6             begin
                .
                .
                .
7             end ; {A}
8   procedure B ;
        .
        .
        .
9             procedure C ;
10                          begin
11                            A ;
                              .
                              .
12                            end ; (C)
13            begin
                .
                .
                .
14              C ;
15            end ; {B}
16    begin
        .
        .
        .
17    B ;
18  end ;
```

FIGURE 2

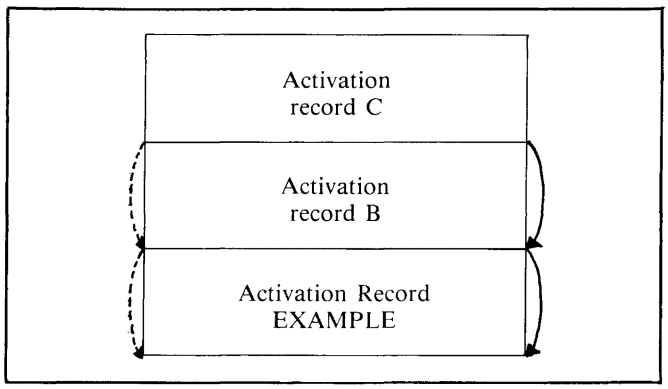

FIGURE 3 Activation Stack prior to procedure call A

most) base. Most data references fall into these two classes. Any reference to data defined in one of the intervening blocks will require backthreading through the access environment links.

The location of the next instruction to be executed within the code portion of a block is held within the Program Counter (PC). The PC is the head of the Jump Return stack used to define the return point to any calling procedure. When a new procedure is activated the current PC value is pushed onto the stack automatically (as part of the activation stack manipulation) and the PC is initialised to the base of the new procedures code segment.

Relocation and protection are provided by a Base and Limits Register structure. An essentially segmented system is envisaged with a Base Register defining the entry point to the current code segment.

The accumulator stack serves as a set of scratchpad registers for intermediate results during expression evaluation. The two accumulators at the top of the stack are both registers in the main processor with the accumulator stack pointer being held

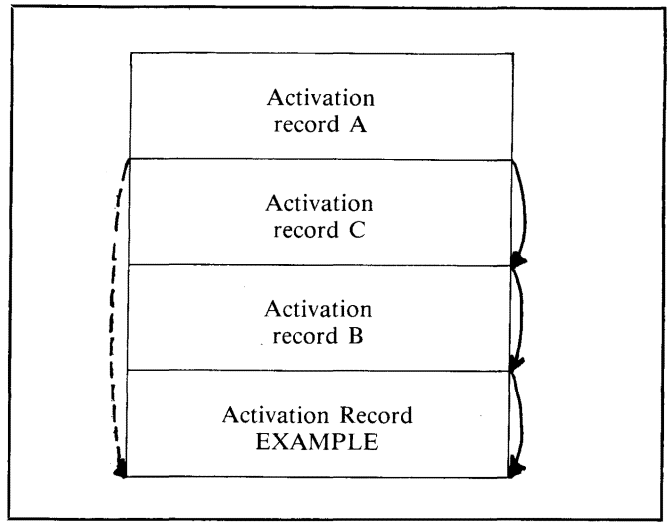

FIGURE 4 Activation Stack during execution of A

in the slave processor. The arithmetic and logical operations are both one and zero-address applied to the top of stack elements. They include a reverse option for the efficient implementation of non-commutative operations.

The index registers consist of an increment portion and a modifier portion with the head of the stack being the Index Register in the main processor.

The use of the stack of index registers may simplify the task of the compiler writer in accessing from multi-level structures and arrays, although such simplification may not provide the most machine efficient form of access.

Similarly the loop control stack is intended to simplify the handling of nested loops with the top stack element being the Loop Count Register in the main processor. Depending on the features of the language being catered for its form may be relatively simple or some form of structured record may be required. For languages such as Pascal the register may be a simple counter manipulated by instructions of the 'Jump and Decrement' type. However, for languages such as Coral 66 a more sophisticated structure may be required containing information such as step size, loop termination value, etc.

3.2 Tagged Data

Although not currently built into the hardware the provision of tag bits appears a useful feature for the effective implementation of a number of HLL constructs. The concept of tagged architecture has appeared fairly extensively in the literature [3,4,6,7,11], particularly as an aid to HLL architectures. It essentially involves the addition of a number of bits to each memory location to provide self-identifying data i.e. any locations can be identified by class at run-time. Interpretation of instructions can thus to some extent be made data-dependent.

Modification of the existing system would not be extensive. In particular the identification of data types is relatively simple given the flexibility of the Intel 3000. The tag bits feed directly into the Microprogram Control Unit which has the facility to use these for the direct control of conditional microprogram jumps.

Currently four classes of tag are envisaged which would require an additional two bits per memory location. The classes used are similar to those suggested by Iliffe [7] and by McMahan and Feustal [11]. The data value class contains the primitive data elements used by the language e.g. integer, boolean, real, character, etc. If a type field is included for each word the individual data types could be dynamically recognised simplifying both the format and interpretation of instructions e.g. automatic type conversion, unified arithmetic instructions for reals or integers, etc.

Reference elements provide address pointers which are in fact segment descriptors of a structure of values. Again a type field may be used to indicate the data type of the referenced segment.

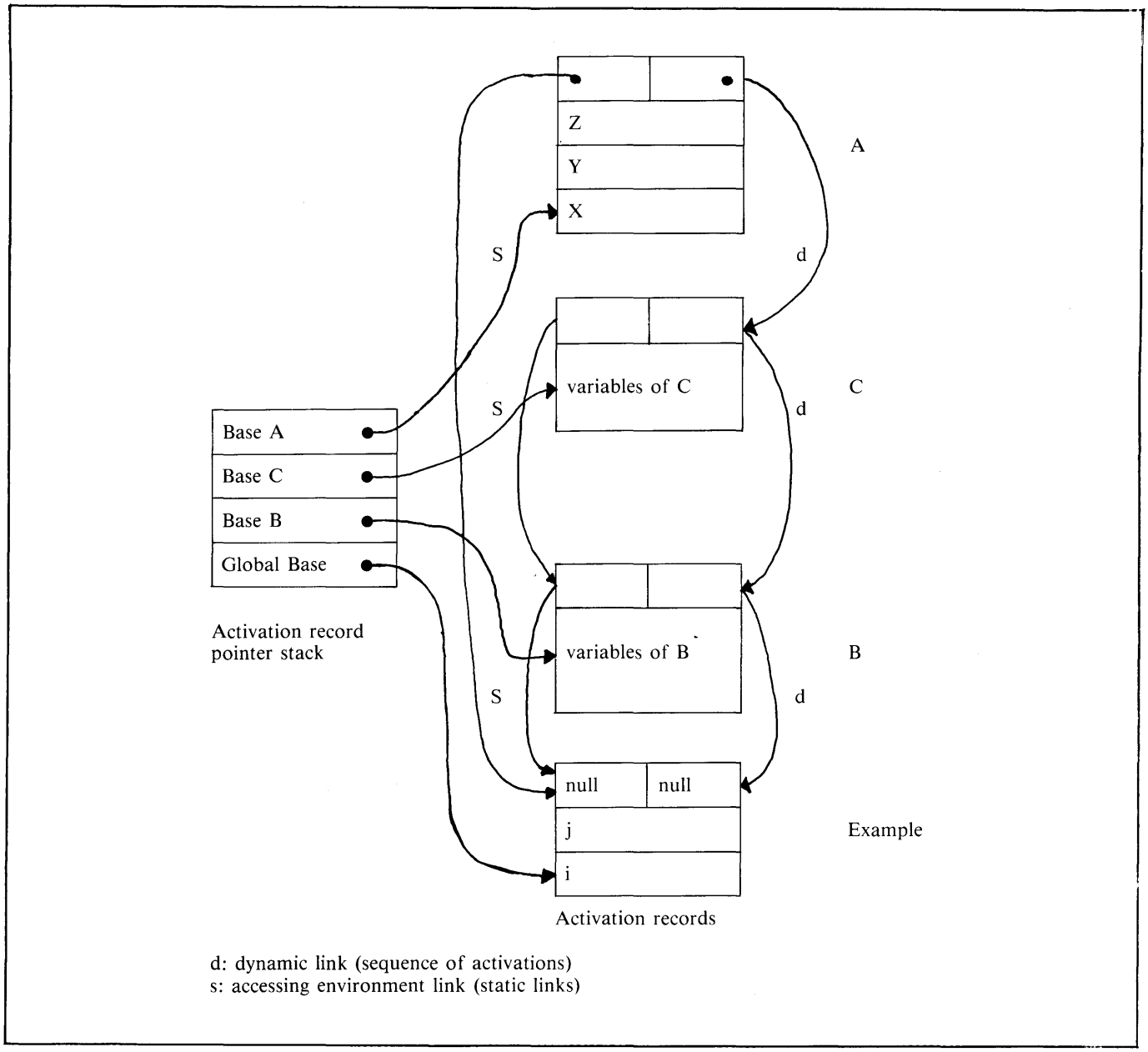

FIGURE 5 Activation Structure

Basically these elements would provide links to data structures.

The class of control elements are used to provide references to the code segments of a process. These may be used to extend the concept of segmentation to that of Iliffe's tree-structured address space which simplifies sharing of code, protection of code segments and the separation of parallel processes.

Instruction elements correspond to the individual process instructions. Any attempt to locate an instruction via an address reference element would lead to an addressing exception (essentialy a data-driven interrupt).

The above four classes of tags considerably extend the power and flexibility of the system.

4. Software

A flexible micro-assembler (VAMPIRE) has been developed for use with the Intel 3000 series [15]. The assembler operates in two modes. For portability and generality an initialising phase allows the user to define a particular configuration i.e. the format of the micro-instructions as well as specifying a set of mnemonics for use with each field of the micro-instruction. This flexibility is essential for the development of systems under research conditions with the probability of design modifications. Once initialised the routine may be used to assemble mnemonic microroutines.

A routine is currently under development to control the automatic placement of micro-instructions in micromemory. This problem is non-trivial due to the two-dimensional nature of Intel control store. The address space is organised as a two dimensional array of 32 rows and 16 columns with any location being accessed by a 9-bit address (512 locations). Each micro-instruction contains a seven-bit field specifying a conditional or unconditional jump as well as some bits for use with the address register. The address register in the Microprogram Control Unit (MCU) is modified to a combination of these jump code bits, the current register contents and certain condition bits. The location of any instruction is thus constrained by the address of its predecessor(s). This restricted form of addressing enables a reduction in the length of the micro-instruction word and a reduction in the pin count of the MCU chip.

Instructions referenced by an unconditional jump are constrained to the same row or column as their predecessor. Those referenced by conditional jumps are restricted to a fixed group of rows containing the predecessor. Depending on jump type this row group may consist of four, eight or sixteen adjacent rows. In addition the referenced instructions may be constrained to certain column groups within a row group.

Microroutines written at the assembler level are in symbolic form without regard to the final location of instructions in

memory. The placement routine then has the responsibility for allocation of memory to individual instructions.

Essentially the process works with a graphical structure defining the flow of control within the microroutines. From this is extracted a set of constraints for each instruction in terms of its predecessor(s), successor(s) and siblings (for conditional jump targets). In addition an ordering and grouping of instructions into classes is obtained (in terms of probable difficulty of placement). This heuristic information is then used to map the instructions to the memory in terms of the ordering. As an instruction is placed the constraints on any related instructions are tightened (as the number of possible locations for these are reduced). The placing of sequences of unconditional jumps is of low priority as these are relatively unrestricted.

During placement a weighting is calculated for divisions of control memory in terms of the instruction load in any division and an attempt is made to place new groups into lightly loaded zones. If an instruction cannot be placed backtracking is initiated and the constraints are reconstituted for another attempt. An initial hand placement of selected micro-instructions may be used to simplify the task of the process.

5. Conclusion

The cost of hardware forms a small proportion of the total cost of a computer system with the bulk of the cost being in software development and maintenance. Most large software systems are currently written in high-level languages, often whose which support the concept of structured programming. Thus any advances in hardware design should enhance the use of such languages.

The Structured Language Machine is an attempt to mirror high-level features at the machine architecture level and thus facilitate the compilation and execution of block structured languages. The design should result in the production of shorter, more compact code sequences as compiler output, simpler sharing and protection of code and data segments and faster access and manipulation of sophisticated data structures than that obtained using a conventional Von Neumann design.

References

[1] D.G. Bobrow and B. Wegbreit, A model and Stack Implementation of Multiple Environments, CACM Vol. 16, **10**(1973).

[2] Y. Chu, High-level Language Computer Architecture (Academic Press, 1975).

[3] E.A. Feustal, On the Advantages of Tagged Architecture, IEEE Trans. on Computers, Vol. c-22, **7** (1973).

[4] E.A. Feustal, Tagged Architecture and the Semantics of Programming Languages: Extensible Types, ACM-IEEE, Third Annual Symp. on Computer Architecture, (1976).

[5] L.S. Haynes, Structure of a Polish String Language for an Algol 60 Language Processor, ACM-IEEE, Symp. on High-Level-Language Computer Architecture, (1973).

[6] J.K. Iliffe, Elements of BLM, Computer Journal Vol. 12, **3** (1969).

[7] J.K. Iliffe, Basic Machine Principles (2nd Ed.), (Macdonald/Elsevier, 1972).

[8] L. Lofgren, Reference concepts in a Tree Structured Address Space, ACM-IEEE, Second Annual Symp. on Computer Architecture (1975).

[9] C. McFarland, A language Oriented Computer Design, Proc. AFIPS FJCC (1970).

[10] K.J. MacGregor and G.R. Finnie, Development of Software for Microcomputers, SACAC, Symp. on Minicomputers and Microprocessors, University of Natal, (1976).

[11] L.N. McMahan and E.A. Feustal, Implementation of a Tagged Architecture for Block Structured Languages, ACM-IEEE, Symp. on High-Level-Language Computer Architecture, (1973).

[12] E.I. Organick, Computer System Organisation: The B5700/B6700 Series, (Academic Press, 1973).

[13] J.W. Paterson, An Abstract Machine for the Implementation of Pascal and Coral 66, (Glasgow University Technical Publication, 1975).

[14] B. Randell and C.J. Kuehner, Dynamic Storage Allocation Systems, CACM, Vol. 11, **5** (1968).

[15] C.D. Rudenberg, The VAMPIRE Users Manual and Systems Manual, University of Cape Town, Internal Publication, (1976).

[16] J.F. Wakerly, C.R. Hollander and D. Davies, Placement of Micro-instructions in a Two-Dimensional Address Space, Proc. Eighth Annual Workshop on Microprogramming, (1975).

[17] H. Weber, A Microprogrammed Implementation of EULER on the IBM System 360 Model 30, CACM, Vol. 10, **9** (1967).

[18] D.B. Wortman, Language Directed Computer Design, International Workshop on Computer Architecture, (1973).

Notes for Contributors

The purpose of this Journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles, exploratory articles of general interest to readers of the Journal. The preferred languages of the Journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to: Prof. G. Wiechers at:

Department of Computer Science
University of South Africa
P.O. Box 392
Pretoria 0001
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. The original ribbon copy of the typed manuscript should be submitted. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name, and the affiliation and address. Each paper must be accompanied by a summary of less than 200 words which will be printed immediately below the title at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Illustrations and tables should not be included in the text, although the author should indicate the desired location of each in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Illustrations should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the Author's name and figure number. Original line drawings (not photoprints) should be submitted and should include all relevant details. Drawings, etc., should be submitted and should include all relevant details. Drawings, etc., should be about twice the final size required and lettering must be clear and "open" and sufficiently large to permit the necessary reduction of size in block-making.

Where photographs are submitted, glossy bromide prints are required. If words or numbers are to appear on a photograph, two prints should be sent, the lettering being clearly indicated on one print only. Computer programs or output should be given on clear original printouts and preferably not on lined paper so that they can be reproduced photographically.

Figure legends should be typed on a separate sheet and placed at the end of the manuscript.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters between the letter O and zero; between the letter l, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetical order of author's name, and cited in the text by number in square brackets. Journal references should be arranged thus:

1. ASHCROFT, E. and MANNA, Z. (1972). The Translation of 'GOTO' Programs to 'WHILE' Programs, in *Proceedings of IFIP Congress 71,* North-Holland, Amsterdam, 250-255.
2. BÖHM, C. and JACOPINI, G. (1966). Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, *Comm. ACM,* **9**, 366-371.
3. GINSBURG, S. (1966). *Mathematical Theory of context-free Languages,* McGraw Hill, New York.

Proofs and reprints

Galley proofs will be sent to the author to ensure that the papers have been correctly set up in type and not for the addition of new material or amendment of texts. Excessive alterations may have to be disallowed or the cost charged against the author. Corrected galley proofs, together with the original typescript, must be returned to the editor within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Fifty reprints of each article will be supplied free of charge. Additional copies may be purchased on a reprint order form which will accompany the proofs.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.

Hierdie notas is ook in Afrikaans verkrygbaar.

Quaestiones Informaticae

Contents/Inhoud

*Presented at the second South African Computer Symposium held on 28th and 29th October, 1981.