SLAC-127
UC-32
(MISC)

DIRECT EMULATION OF CONTROL STRUCTURES BY A PARALLEL MICRO-COMPUTER

VICTOR R. LESSER*
STANFORD LINEAR ACCELERATOR CENTER
STANFORD UNIVERSITY
Stanford, California 94305

PREPARED FOR THE U. S. ATOMIC ENERGY
COMMISSION UNDER CONTRACT NO. AT(04-3)-515

October 1970

Reproduced in the USA. Available from the National Technical Information Service, Springfield, Virginia 22151.
Price: Full size copy $3.00; microfiche copy $ .65.

---

*The research was carried on while the author was a NSF graduate fellow and partially supported under NSF2-FCZ-708-94140, AT(043)326, P.A.23.

ABSTRACT

This paper is a preliminary investigation of the organization of a parallel micro-computer designed to emulate a wide variety of sequential and parallel computers. This micro-computer allows tailoring of the control structure of an emulator so that it directly emulates (mirrors) the control structure of the computer to be emulated. An emulated control structure is implemented through a tree type data structure which is dynamically generated and manipulated by six primitive (built-in) operators. This data structure for control is used as a syntactic framework within which particular implementations of control concepts, such as iteration, recursion, co-routines, parallelism, interrupts, etc., can be easily expressed. The major features of the control data structure and the primitive operators are: 1) once the fixed control and data linkages among processes have been defined, they need not be rebuilt on subsequent executions of the control structure; 2) micro-programs may be written so that they execute independently of the number of physical processors present and still take advantage of available processors; 3) control structures for I/O processes, data-accessing processes, and computational processes are expressed in a single uniform framework. This method of emulating control structures is in sharp contrast with the usual method of micro-programming control structures which handles control instructions in the same manner as other types of instructions, e.g., subroutines of micro-instructions, and provides a unifying method for the efficient emulation of a wide variety of sequential and parallel computers.

ACKNOWLEDGEMENTS

I wish to express my sincere thanks to Professor William Miller whose constant support and encouragement of my research efforts have made possible the successful completion of this paper. I would also like to thank Professor Ed Davidson for his detailed reading and criticisms of this paper, and Dr. Harry Saal and Professor William McKeeman for their encouragement of my research efforts and the many fruitful discussions I had with each. Thanks especially to my friends and fellow graduate students Lee Erman and Bill Riddle who have had to suffer through an uncountable number of rewrites and discussion of this paper.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

In the past few years, both the size and diversity of the class of problems being submitted to computers for solution has significantly increased. The programming of many of these new problems on a computer with a von Neumann organization can be very complex and, additionally, can result in programs which execute inefficiently. A significant part of these difficulties can be attributed to the "degree of complexity" of the transformation from the representational framework within which the programmer develops an algorithm (e.g., ALGOL, LISP, Graph Model, etc.) to the representational framework of a von Neumann computer within which the algorithm is executed. The complexity of transformation between these two levels of representation thus makes it difficult to construct an automatic mapping between levels which is both quick and efficient. The perception of this problem has led to the development of computers whose organizations are optimized for either a particular subset of or a higher level language for the problem class. Examples of such machine languages should include those of the B5500 [1] for ALGOL, ILLIAC IV [2] for processing of array structured data, Abram's APL machine,[3] Melbourne and Pugmire's FORTRAN[4] machine, etc. Since these represent a broader class of languages than what is usually meant by machine language, we will refer to them as intermediate machine languages (IML's). This tailoring of IML to a specific higher level language is accomplished by incorporating primitive operators in the IML which directly mirror operations in the higher level language (e.g., recursion in ALGOL is directly mirrored through stack operations in B5500). Thus, by the tailoring of a machine's organization more closely to a particular user representational framework, the mapping between levels is simpler and results in more efficient program execution.[20]

In parallel with the development of problem oriented computers, there has been an effort toward providing a systematic and flexible approach to the hardware design of a specific computer. This effort has led to the development of micro-computers, e.g., 360/40,[5] with read-only control memories programmed to emulate a specific von Neumann type computer.

Recently, there has been an attempt to integrate both of these new directions in computer architecture (machine organizations designed for specific applications and micro-computers) by attaching to the micro-computer writeable control memories. Thus, it is intended that through the ability to modify dynamically the control memory of a micro-computer, a wide range of machine languages of different computer organizations (IML) can be efficiently emulated on a single micro-computer. However, it is the author's contention that this goal cannot be realized by existing micro-computers.

A. Traditional Micro-Computer Architecture

Existing micro-computer architectures are still oriented toward the design of von Neumann type computers rather than a systematic approach to the emulation of a wide variety of different sequential and parallel intermediary machine languages.

The program structure of an IML emulator, in a conceptual sense, is seen in Fig. 1.

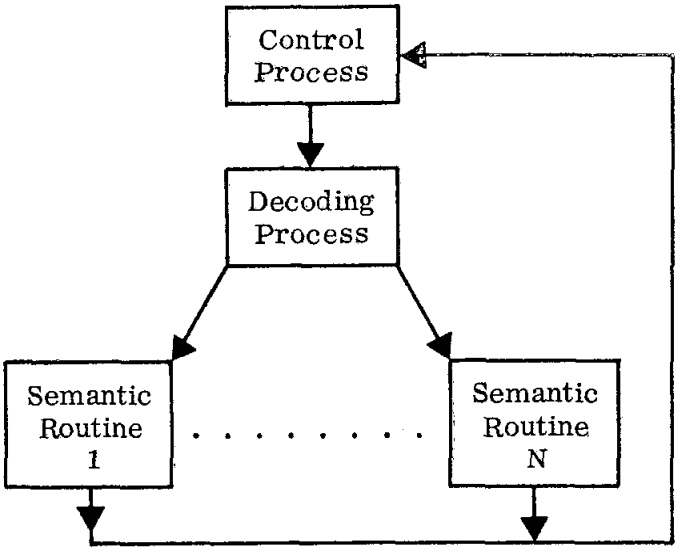


FIG. 1--Conceptual structure of an emulator.

The "control process", which represents the control structure* of the computer to be emulated, activates the "decoding process" with data that identifies the next instruction(s) of the emulated computer to be executed; the "decoding process" analyzes the instruction(s) to be executed so as to determine the semantic routine(s), together with its (their) appropriate calling sequence(s), whose activation will perform the semantics of the emulated instruction(s). After the appropriate semantic routine(s) has (have) been executed, the flow of control returns to the control process which, based on the results of executing the decoding process and the semantic routine(s), selects the next instruction(s) to be emulated.

*The control structure of a computer consists of the set of rules used to define the sequencing of the instructions of the computer.

The organizations of existing micro-computers when applied to the emulation of unanticipated IML's do not reflect this conceptualization of the structure of an emulator, but rather provide a simple, uniform framework for the coding of an emulator. In these machines, the semantics of micro-instructions are generally realized by a short parallel sequence of register transfers, and the control for sequencing among micro-instructions is sequential and based on simple conditional transfer commands. There are no features in the language that distinguish the coding of the control process from that of the decoding process or the semantic routines, nor the relationship, for instance, between the control process and the decoding process. An emulator expressed in this type of micro-computer language "... implements machine instructions as a subroutine of micro-instructions".[6] Thus, due to the simplicity of micro-computer languages and their paucity of control commands, the structure of the emulated computer is not directly observable in the structure of its emulator. The key to efficient emulation is just this missing ability to directly mirror the control structure, instruction formats, and primitive data-accessing operations of an IML in the corresponding control structure, instruction formats and primitive data-accessing operations of its emulator. In particular, a control action by an instruction in the IML program being emulated should be directly mirrored in a modification of the control structure of the emulator.

Thus, the current approach to the design of a micro-computer which stresses simplicity is not unreasonable if the micro-computer is going to emulate computers and IML's that have a simple sequential control and simple instructions. But, IML's that are tailored for a particular subset of a higher level language for a problem class are, in a sense by their very purpose, not simple since the complexity of the higher level language is imbedded in the semantics of the IML's instructions and control structure. If the current trend in higher level languages is maintained, these problem or procedure oriented IML's will have increasingly more sophisticated control structures employing such control concepts as recursion, co-routines, parallelism, etc., and, likewise, their instructions will directly operate on increasingly more complex data structures, e.g., lists, trees, arrays, etc. Therefore, the current structure of existing micro-computers is inadequate for the task of effectively emulating the wide range of such intermediary languages, just as a von Neumann computer in comparison with the B5500 does not efficiently execute ALGOL.

B. Variable Control Structure as the Basis of a Micro-Computer Architecture

The micro-computer architectural design to be presented in this paper is based on the idea that the program structure of an emulator written in this micro-computer should reflect the structure of an IML that is being emulated. It is felt that the key to accomplishing this mirroring process between IML and its emulator lies in the control structure of the micro-processor. Thus, the main emphasis in the design to be presented here is to incorporate a very general control structure in the micro-processor.

The approach conventionally used to design a micro-processor with a powerful control structure is first to develop a basic machine language having a well-defined set of instructions and a simple sequential control structure, and then add instructions and facilities (such as subroutine call instruction, a stack for parameter passage, a fork-join instruction, etc.) for structuring complex sequential and parallel processes. This is not the approach taken here. Instead, the approach is to develop a micro-language specifically designed for the task of dynamically constructing control structures. This control structure definition language, called the Structure Building Language (SBL), is used to dynamically define a wide range of particularized control structures through the generation of a data structure for control. The control data structure acts as a syntactic framework within which dynamic and static control and data environment inter-relationships among processes can be expressed. The control structure of this micro-computer can then be dynamically tailored (through the SBL) into a form which is most suitable for the emulation of a particular IML. An emulator programmed in this micro-computer, as will be seen later, works in a fashion similar to the process of dynamic compilation or run-time macro expansion. This method of emulation differs radically from the conventional form of emulation consisting of a sequence of calls to sub-routines of micro-instructions.

The variable nature of the control structure of this micro-computer distinguishes its architecture (from the viewpoint of form and complexity) from existing micro-computer architecture. It is felt that a variable control structure micro-computer provides a unifying approach to the emulation of an extremely wide variety of computer organizations and IML's. The goals of this micro-computer design are to be able to:

1. Emulate efficiently a wide class of both sequential and parallel IML's (e.g., array processors, pipeline, stack machines, LISP machines, computational graph models, etc.).

2. Program an emulation in a simple and uniform manner, such that the dynamic program structure of an emulator reflects the architecture of the computer it emulates.
3. Incorporate easily and efficiently a changing array of hardware arithmetic units (e.g., square root, inner product, etc.) I/O devices and memory units (e.g., associative memory, bit slice memory, etc.).

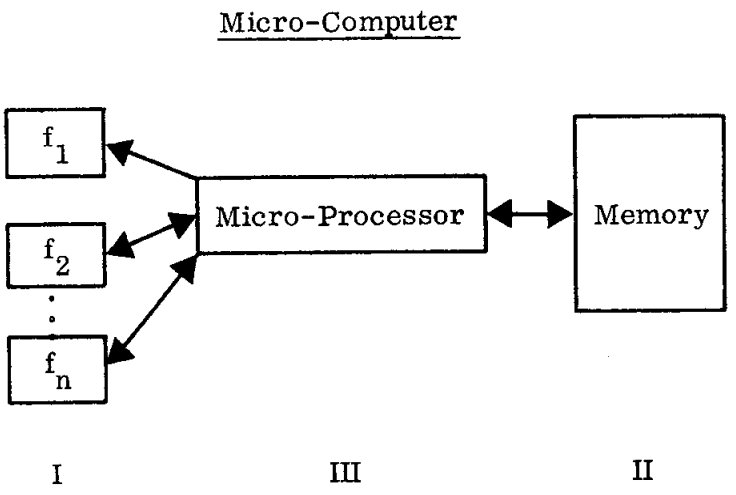


FIG. 2--Micro-Computer subsystems (modules).

II. MICRO-COMPUTER ARCHITECTURE

The micro-computer architecture, as pictured in Fig. 2, can be characterized in terms of three basic hardware subsystems. The first subsystem is composed of an arbitrary set of functional units. Each of these units can be independently activated and can have an arbitrary number of inputs and outputs, where that number need not be fixed but may be data dependent. A functional unit could be a floating point multiplier or, more generally, an arbitrary input/output device. This more general usage of a functional unit is a natural consequence of imposing restrictions neither on the size (or form) of the input and output data sets of a unit nor on the sequencing between units.

The second subsystem is a memory. This memory is bit-addressable and can be activated either to store or retrieve an arbitrary length string of bits. This memory holds the program that is going to be emulated, and additionally, serves as a storage buffer for communication between the functional unit subsystem and the micro-processor subsystem. Other types of memory organizations, such as word-oriented, bit-slice, associative, etc., can also be included in the system's architecture by making them function units.

The third subsystem, which is the major innovation in this micro-computer architecture, is a micro-processor that controls the dynamic interactions between the other two subsystems and among functional units. The programmable nature of the control unit of the micro-processor subsystem allows the tailoring of both the hardware and software of this architecture to various problems. The hardware tailoring involves the addition of specialized functional units which carry out operations commonly used in the problem class (e.g., floating-point multiplier bit-slice memory, etc.) to the functional unit subsystem or addition of more parallelism in the micro-processor subsystem. The variable nature of the control unit of the micro-processor subsystem, as will be discussed later, allows these hardware modifications to be incorporated without modification to the language of the micro-processor.

In order to emulate a computer using this system, the program which is to be run on the emulated computer is stored bit-wise in the memory subsystem in the same order as it would be stored in the emulated computer's memory. The micro-processor must then perform the following tasks: (1) fetch from the memory subsystem the instruction(s) of the emulated computer which is (are) to

be executed in the next step; (2) analyze this (these) instruction(s) in order to generate the appropriate sequence of functional unit activations which will perform the computations specified by the instruction(s). In addition, the sequence of functional unit activations must be coupled with accesses and stores to the memory subsystem so as to provide the input and output data set for each unit. This sequence of functional activations may result in concurrent operation of functional units or a pipelining of functional units.

The major focus of the rest of the paper will be on the organization of the control unit of micro-processor subsystem, especially the syntax and semantics of the SBL.

III. MICRO-PROCESSOR SUBSYSTEM

The main orientation in the design of this micro-computer, as stated in the introduction, is to incorporate a variable control structure definitional facility into the hardware of its processor. This design emphasis has led to a micro-processor that contains two basic classes of instructions. One class of micro-instructions, called the Structure Building Language (SBL), is used to construct dynamically the control structure of an emulator while the other class, called the Interger Function Language (IFL), is used to compute address arithmetic functions.

The SBL dynamically defines an emulator's control structure through the generation of a data structure for control. The basis of the syntax and semantics of the SBL is a fixed set of definitional templates that define particular types (forms) of control structures. An SBL statement (macro) specifies one of the fixed set of templates together with a set of IFL address arithmetic functions. Each definitional template represents a parameterized model of a basic control concept, e.g., iteration, selection, hierarchy, synchronization, etc. The specification of particular values for the parameters of the template defines a particular instance of a basic control concept. These values are computed by the IFL address arithmetic functions specified in the SBL macro. A call to an IFL program results in the generation of either an integer value or a sequence of interger values that are then used in the expansion or execution of a macro. The expansion of a definitional template results in the generation of a structure which contains all the state information necessary to model the execution of this particular instance of the control concept. More complex control structures are constructed through the expansion of a sequence of these definition templates. The binding of parameters to the SBL macro is under the explicit control of other SBL statements. Similarly, the expansion of SBL macros and later execution is explicitly programmable in the SBL. This ability of the SBL to define dynamically the sequencing of other SBL statements is the key to the control structure definitional facility of the micro-processor.

The SBL consists of six types of macro bodies (definitional templates): data-descriptor (D), instruction (I), selection (S), iteration (IT), hierarchical (H), and control (C). The first two types of macro bodies are called subsystem command macros while the remaining four are called structure building macros. The subsystem command macros specify the interaction between the functional unit

subsystem and the memory subsystem. Only these two macros actually produce computational results through the action of functional units. More complex computational processes are constructed through the execution of a sequence of structure building macros that use as their basic building block calling sequences to subsystem command macros. When the basic building blocks are just data-descriptor macro calling sequences, then the structure building macros defines a data-accessing procedure.

The programming of an emulation on this micro-computer is done by creating a dynamic mapping between the control structure and instructions of the emulated computer and a set of structure building macros and subsystem command macros. This dynamic mapping is represented in the address arithmetic algorithms that are used to expand the definitional templates. Thus, an emulator programmed in this micro-computer works as an iterative two-step process (i.e., it generates an instance and then executes the instance) similar to the process of dynamic compilation or run-time macro expansion. This two-step approach to emulation differs from the conventional one-step approach to emulation (i.e., calling sub-routines of micro-instructions) done on existing micro-processors, and directly reflects the conceptualization of an emulator pictured in Fig. 1. The binding of a parameter list to a SBL macro is the analog of the control process of the emulator; the expansion of a SBL macro is the analog of the decoding process of the emulator, and the execution of SBL macros is the analog of the semantic routines of the emulator.

Example 1

Consider the emulation of an instruction, FAD I 20, stored at location 10 in the emulated computer where FAD specifies a floating add operation, I specifies indirect addressing, and the accumulator is the second and result operand. The sequence of steps involved in emulation of this instruction on this micro-processor is the following: (1) An SBL instruction generates and then stores as a node in the control data structure a binding between a pointer to the current value of the program counter of the emulated computer: 10, and a subsystem command macro A. (2) The macro A with a parameter whose value is 10 is then expanded. This expansion results in the generation of a subsystem command in the control data structure. The expansion of a subsystem command macro is based on

a template having the following format: "functional unit", "address of input 1", "address of input 2", "address of output 1". Macro A fills in the slots of the template by calling with parameter 10 two IFL programs B and C whose integer value outputs respectively, fill in the "functional unit", and "address of input operand 1" fields. The other two fields are always constants specifying the address of the accumulator of the emulated computer. The IFL program B extracts the op-code field of the instruction at location 10, and then based on this value, determines the functional unit in the functional unit subsystem that carries out the operation specified by the op-code. The IFL program C does the address arithmetic, in this case indirect addressing, required to locate the address of the operand specified by the instruction at location 10.
(3) The instance of a subsystem command generated by step 2 is then executed. The execution of this command results in the activation of the floating point add functional unit with two operands and then the storage of the result of the floating point operation in the accumulator of emulated computer. Thus, the subsystem command carries out the semantics of the emulated instruction FAD I 20. This example indicates the three phases involved in emulating IML instructions. However, it should be pointed out that for the emulation of additional IML instructions with the same basic format (e.g., op-code, indirect bit, address) the binding and expansion phases can be eliminated. Thus the overhead involved in the binding and expansion phases need be incurred only once for each different instruction format of the emulated computer. The control data structure for an idealized von Neumann computer is pictured in Fig. 4 on page 32, and will be used in the next section as a basis for discussing the six SBL macro types.

The basic hardware organization of this micro-processor subsystem at the functional level is pictured in Fig. 3. The micro-processor subsystem contains an arbitrary number of identical micro-processors. The execution of the micro-processors are controlled through data stored in the program and process-space memories. These two memories differentiate the static and active parts of the control structure of the micro-processor subsystem. The "program memory" holds SBL and IFL statements and is not normally modified during an emulation;

the program memory is similar to the control memory of a conventional micro-processor. The "process space" memory holds the control data structure constructed by the SBL and is constantly being modified during an emulation. The contents of the process space memory is in essence the state of the emulator which is currently being executed by the micro-processor subsystem.

The micro-processor subsystem can carry on parallel activity since the number of micro-processors contained in the micro-processor subsystem is arbitrary and these processors can be executed concurrently. The process space memory holds the definition of the control structure which coordinates, in a virtual sense, the activity among micro-processors. In the case that there are not enough micro-processors to carry out the parallel activity specified by the control structure in the process space memory, then the available micro-processors are scheduled on a first come-first serve basis. This transformation from virtual processor activity to actual processor activity may lead to indeterminate results depending upon the number of micro-processors available. However, as will be described in Section IV.E.4 the SBL contains control primitives that allow the programmer to construct the appropriate synchronization rules (Dykstra's semaphore, Saltzer's wakeup-waiting switch, lock-step execution, etc.) which preserve the inherent parallelisms among processes, while at the same time guarantee the scheduling of virtual parallel activity will always result in determinate computation independent of the number of actual mirco-processors.

Micro-Computer Hardware Organization

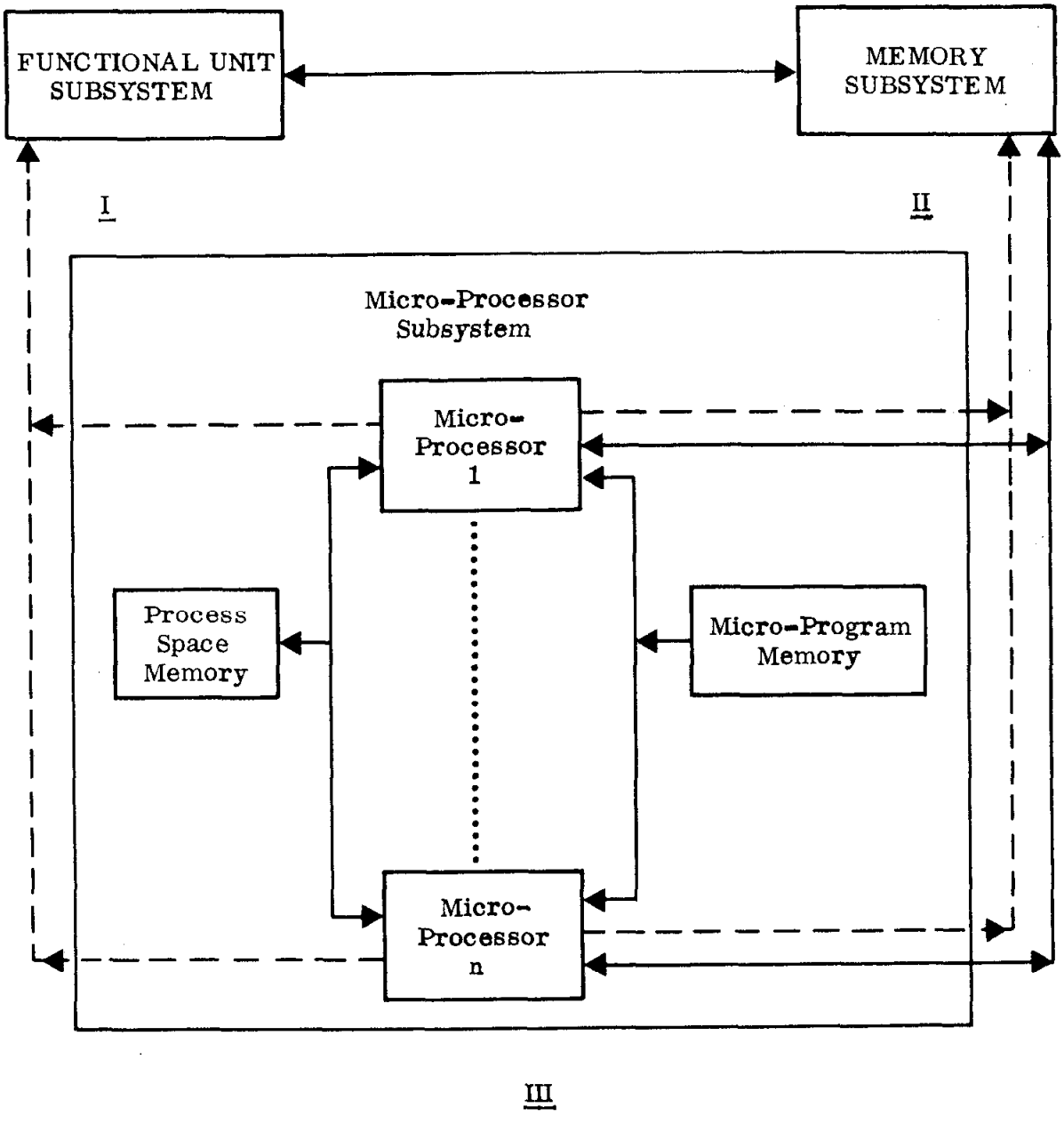
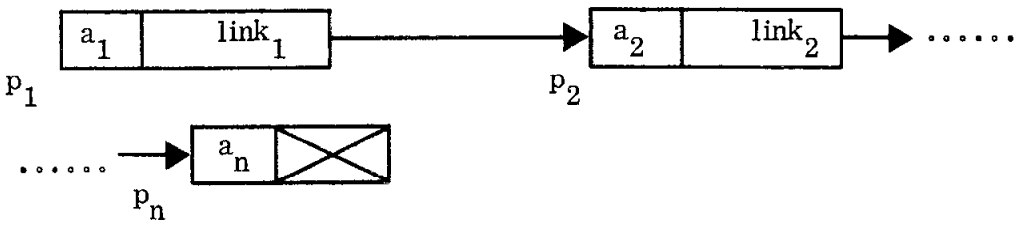


FIG. 3--Micro-Processor subsystem's organization.

IV. STRUCTURE BUILDING LANGUAGE (SBL)

The SBL is used to define control structures for I/O processes, data-accessing processes, and computational processes. The SBL defines each of these types of control structures in a single uniform framework. This use of a single framework for data-accessing and computational processes came from the following observation: if a set of instructions are considered to form a data structure, then the control structure associated with the sequencing of these instructions can be considered as a data-accessing procedure where the data being retrieved are instructions. For example, consider the following representation of a typical list structure:



where p_i is the address of the ith word in the list, a_i is the data-item stored at the ith word, and $link_i$ is data stored at the ith word used in computing p_{i+1}. A data-accessing procedure to extract $a_1,\ldots a_n$ from this typical list structure would generate the sequence $p_1,\ldots,p_n$ from the link information $link_1,\ldots link_{n-1}$. After the generation of each p_i (i=1,n) the corresponding a_i can then be extracted.

Similarily, consider $a_1\ldots a_n$ as machine instructions. They can be sequenced by a program counter p which takes on a succession of values $p_1,\ldots p_n$. After the generation of each p_i, the instruction a_i located at p_i is executed, and then based on p_i and a_i, p_{i+1} is calculated. The only difference between instruction sequencing and data-accessing of a list structure is that in instruction sequencing the link information, $link_i$, is always encoded in the instruction, a_i (an instruction includes an implicit or explicit link). Thus, the general paradigms developed to sequence through arbitrary list structure can also be used to define conventional sequential control structures.

The IFL is specifically designed to efficiently sequence through an arbitrary formatted list structure, and generate either the address of the final list element p_n or the addresses of the intermediate list elements $p_1,\ldots p_{n-1}$. In the latter case, the SBL uses the addresses of these intermediate list elements to generate

a series of macro calling sequences (the binding of a parameter p_i to a macro body). The execution of the macro with parameter p_i then results in the carrying out of the semantics associated with a_i, where a_i can be a data-item, an emulated instruction, or the name of a process. These semantics involve, respectively, the retrieval of the data-element from the memory subsystem, the execution of a functional unit with appropriate input and output sets, or the generation and execution of further macro calling sequences. The first two cases are handled by subsystem command macros while the latter case by structure building macros. Thus, depending on the types of the macros bound to the sequence of parameter $p_1 \cdots p_{n-1}$, a data-accessing process, an I/O process, or a computational process can be defined.

A. Control Data Structure

The SBL defines a control structure through the dynamic generation of a tree type data structure in the process space memory whose nonterminal nodes contain calling sequences to either a subsystem command macro or a structure building macro. The process space memory also holds all temporary information structures, which will be considered as terminal nodes of control data structure, needed in the expansion and the execution of a macro. The data structure for control is in the form of a tree due to the ease of specifying such control concepts as hierarchical structure (functional decomposition), parallelism, co-routines, and recursion. The representation of hierarchical structure and recursion is possible because additional levels (sibling groups) may be dynamically built in the tree through the expansion of nonterminal nodes (macro calling sequences). The representation of parallel and co-routine control structures is possible because brother nodes in the tree may be treated as distinct independent processes each with its own state information. A tree data structure is also a convenient syntax framework (father, brother, etc., relationship between nodes) for defining distributed control systems. Namely, the control structure of a complex system can sometimes be conveniently represented through hierarchical structure where in each sibling set (structural level) of the tree there is embedded a simple control process (clocking process)[9] that initially sequences its brother nodes. If additional clocking processes are contained in the sibling set, control may pass to these processes after initialization. Thus, instead of one complex control process for the entire system, the control can be distributed throughout the

system. In addition, if these simple control processes can be coded so their addressing structure is not based on their absolute locations in the tree, but only on their relative position in terms of father and brother addressing in the tree, then relative addressing allows copies of a single process to be used at different levels in the tree. The simultaneous execution of many calling sequences to the same macro body is permitted because information local to each macro expansion and its subsequent execution is stored with the activating calling sequence.

Another important feature of the SBL is the separation that is made between the generation of a macro calling sequence (e.g., the binding of parameters to the macro body) from the expansion and execution of that calling sequence. The rules for the dynamic sequencing of the nodes of the control data structure can, therefore, be different from the rules for building of the control data structure. The only built-in sequencing associated with the tree is that a father node must be expanded before any of its son's. The form of control data structure is thus just a convenient syntax framework within which sequencing rules can be expressed. This allows control structures which cannot be conveniently represented in a tree structure (e.g., fork-join control as will be seen in example 9, computational graphs, etc.) to still be programmed in the SBL since the tree is the form for generation of the control data structure but not necessarily the form for the passage of control during execution. The SBL also separates the expansion of a macro calling sequence (which results in the generation of a control structure that defines a process) from the subsequent execution of the expanded macro (which results in the execution of the process). Through this separation, the SBL can control the relative rate of execution of the control structure defined by the expanded macro, e.g., executing a macro that defines an iteration control structure for only one cycle (loop) and then suspending the execution of the macro.

A tree node (macro calling sequence) has seven states of activity: (1) it is unexpanded; (2) it is being expanded; (3) it is expanded; (4) it is being executed; (5) it is being suspended*; (6) it is suspended; and (7) it is terminated. By controlling the activity rate of a node, namely the rules (conditions) for transition between the seven node states, the SBL can produce an arbitrary "time grain". The time grain of a process refers to the smallest unit of a process activity that can be controlled. Time grain, as will be seen later, can be employed to represent concisely such control concepts as co-routines, interrupts, monitoring, lock-step execution, etc.

*The fifth state indicates the node is currently executing but will be suspended at the end of its current time grain.

The ability to separate the expansion of a macro calling sequence from its execution also avoids the unnecessary rebuilding of the control data structure when the form of the control data structure (e.g., the number of son nodes at a particular level in the tree) does not vary from execution to execution. The SBL is defined so that only the dynamic parts of the control structure are rebuilt; the static parts of the control structures once defined are not regenerated. Additionally, the parameters used to execute and to rebuild parts of the control structure can be different from those used to initially generate the control structure.

B. Use of the Six SBL Macro Types

In a recent report by D. Fisher,[10] the control concepts underlying all control structures were specified as the following: "(1) there must be means to specify a necessary chronological ordering among processes and (2) a means to specify that processes can be processed concurrently. There must be (3) a conditional for selecting alternatives, (4) a means to monitor (i.e., nonbusy waiting) for given conditions, (5) a means for making a process indivisible relative to other processes, and (6) a means for making the execution of a process continuous relative to other process ... A process A will be called continuous relative to another process B if and only if communication is established between A and B in such a way that state changes in B are temporarily delayed while the entire action of A is carried to completion."

These underlying control concepts are implemented in terms of the structure building macros in the following ways, respectively: (1) Sequential control is implemented through the iteration macro. The iteration macro generates a list of macro calling sequences where each calling sequence is executed to completion before the next calling sequence in the list is generated. (2) Parallel control is implemented by the hierarchical macro. The hierarchical macro generates a list of macro calling sequences as its son nodes in the control data structure plus specifying a clocking process that controls the initial sequencing of the son nodes. The clocking process, in turn, executes control macros that control the execution of son nodes. These control macros can activate a node without the control macro's completion being delayed until the completion of the activated node, and therefore, the clocking process does not have to wait for the completion of a node before it activates other nodes. Thus, a clocking process can activate two or

more son nodes so that they are concurrently executing. (3) Conditional sequencing is implemented by either a selection macro or a hierarchical macro in which case the son nodes are possible alternatives and the clocking process selects the alternative. (4) Monitoring and continuous sequencing is implemented through the idea of time grain. The control structure of a process that is being monitored for a specified condition can be constructed so that the process is activated so as to suspend itself after it has performed the smallest unit of work which can effect the condition being monitored. Thus, before reactivating the suspended process the condition being monitored can be checked, and if necessary, an appropriate interrupt process activated. The concept of time grain is realized through the use of a clocking process for a group of son nodes together with the ability to execute via a control macro an iteration macro for only one cycle (calling sequence) per execution. (5) Indivisibility of processes is realized by not allowing a control macro to execute a node which is currently executing or being expanded.

The subsystem commands macros in conjunction with structure building macro are used to define an I/O control structure which, for example, can duplicate the effect of an I/O channel on a conventional computer. An I/O control structure defined by a subsystem command macro can be considered a macro-instruction when the functional unit being controlled in an arithmetic device. This use of a subsystem command was exemplified by example 1. The idea of a generalized I/O control structure to control arithmetic units has been proposed in a previous paper by the author,[7] and also has been proposed by Lass[8] as basis of the design of a high speed computer.

C. Format of SBL Macro Calling Sequence

An SBL macro calling sequence has a fixed format, and consists of an address, q, and two integer parameters, p and k. The address, q, specifies the location of a macro body in the program memory. The integer values defined by p and k are the external parameters used in the expansion of the macro body. These external parameters are stored in the control data structure as integer values, pointers to p or k parameters in other macro calling sequences stored in the control data structure, or pointers to fields in the memory subsystem. In the latter case, the pointer has two components, the first component is the beginning bit address of the field while the second component is the length of the field.

This field in the memory subsystem is interpreted as an integer value where the length of the field is smaller than the length of fixed size integer data that the IFL operates on.

This option of storing pointers instead of values for the external parameters p and k greatly increases the ability to program emulators that directly mirror the control actions of the emulated computer. The first type of pointer allows the representation of the static data relationships between p and k parameters in the control data structure. In particular, the first type of pointer facilitates the representation of broadcast type control structures, and allows modifications at one level in the control data structure to be reflected in changes at other levels in the tree which are not normally accessible from the first level. The second type of pointer allows the state of emulator to be directly mapped on to the state of the emulated computer. This mapping is accomplished by storing part of the state of emulator in the memory subsystem instead of entirely in the process space memory. Thus, SBL operations on p and k parameters can be directly reflected back into changes in the contents of the memory subsystem. In particular, this second type of pointer capability is very valuable in the programming of an emulator for a computer whose state vector is not separated from its memory (e.g., the PDP-11[16] computer whose program counter is stored as register 7 in its memory) since the state of emulator (e.g., the address of current instruction being processed, etc.) and the state of the emulated computer (e.g., its program counter, etc.) can be made equivalent. Thus, the emulator does not have to process in a special way instructions of the emulated computer that modify memory registers which contain parts of the state vector of the emulated computer. Further, the second type of pointer capability allows the state vector of an emulated computer to be stored in a single field in the memory subsystem and references to it to be distributed throughout the control data structure. Thus, by modifying a single field in the memory subsystem, the control data structure can be modified to reflect a new state vector for the emulated computer.

The expansion of a SBL macro q, based on p and k, generates the form of a control structure and the internal parameters of the control structure definition that are not modified (constant) from one execution to another. After the expansion of the macro q, the value of the expansion parameters p and k can be changed by a control macro to $\bar{p}$ and $\bar{k}$, and used as execution parameters of the process

defined by the expanded macro. The internal parameters, which vary from execution to execution, are not calculated at macro expansion time, but instead, are recalculated based on the execution parameters $\bar{p}$ and $\bar{k}$, upon each new execution* of the process defined by the control structure. The programmer can define which of internal parameters vary by setting appropriate fields in the macro body. Varying internal parameters are distinguished from constant internal parameters in the control data structure by storing, respectively, the name of an IFL program in the parameter field instead of an integer value. Thus, only dynamic parts of a control structure need be rebuilt on each execution, and only parameters with varying values need be recalculated.

A macro call contains only two parameters, p and k, because most sequential control rules can be expressed in terms of the modification of, at most, two variables at each step of the sequencing. Thus, the two parameters p and k represent the variables or pointer to the variables which are modified at each step of the sequence. The semantics usually associated with these two parameters will be the following: the first parameter, p, represents the address of the data (e.g., instruction, parameter list, etc.) to be processed at the current step of the sequence, and the second parameter, k, represents the value of a counter that determines the termination of the sequencing.

Example 2

Consider the ALGOL statement: "FOR I ← 1 step 1 until N DO A(I) ← B(I) *C(I),". The sequencing for this statement can be defined in terms of the following list of pairs: (1,N) (2,N-1) ... (i,N-i+1), ... (N,1). The first element of the pair defines the value of I. The value of I is then used as a parameter to a macro that constructs the subsystem commands to carry out A(I) ← B(I) *C(I). The second element of the pair, whose value is the number of iterations that remain before the current iteration is initiated, is used to define the termination condition of the FOR loop. The IFL program that generates this list of pairs, as will be seen later, in example 17, can be stated in just one IFL instruction.

*It may be advantageous to also have the option of recomputing internal parameters when the process goes from the suspended state to the execute state.

The "address" of a data item is used in this discussion in a very general sense to mean information sufficient to determine, possibly by a calculation, either the location of the data-item in the memory subsystem or its explicit value.

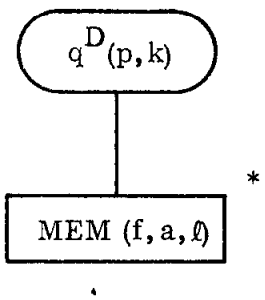
The following notation will be employed in the paper for specifying a macro name, a macro type, and a macro calling sequence. A macro name is specified in one of three following ways: (1) as a symbolic name which is optionally subscripted, e.g., M, a_i, a_{10} etc.; (2) as an absolute address in the program memory enclosed in parentheses, e.g., (0), (10), etc.; (3) as an address arithmetic expression involving symbolic names enclosed in parenthesis, e.g., (a+10), (M_i+i), $(M_0+A_i-B_i)$. The type of macro is specified by appending D, I, S, IT, H, or C, as a superscript to the macro name, e.g., M^I, $(0)^S$, etc. The macro type is optional and is only added for reading clarification. A macro calling sequence is defined by a macro name and optionally its type followed by two parameters which are either symbolic names or integer values enclosed in parentheses, e.g., $M_i(0,5)$, $(10)^D(0,5)$, $(M+5)^D(p,k)$, etc.

D. Subsystem Command Macros

The data-descriptor macro, when expanded, generates a memory subsystem command. The memory subsystem command, when executed, activates the memory subsystem to retrieve (or store) a single data-item. This command is defined in terms of three fields: the first field, f, specifies the format of the data-item (1's complement, floating point, etc.), the second field, a, specifies the address in the memory subsystem of the beginning bit position of the string of bits which denote the data-item, and the third field, ℓ, specifies the length in terms of the number of bits of the data-item. The execution of the memory subsystem command results in the bit string bounded by addresses a and $(a+\ell-1)$ being retrieved from the memory subsystem and then sent together with format field, f, to a functional unit. If $\ell=0$, then address a is used as an immediate operand. The data-descriptor macro neither specifies the particular functional unit that receives or generates the data-item, nor whether the operation is a store or fetch. These specifications of functional unit and operation are defined by the instruction macro that directly or indirectly activates the data-descriptor macro calling sequence. Thus, the same data-descriptor macro can be used with many functional units and may be used either for a store or fetch operation. The use of a format field, f, in the specification of both input and output allows the functional unit to be very sophisticated in being able to perform, if desired, arithmetic operations involving operands and results of different types and lengths. This type of functional unit was proposed for B8502[11] computer.

The data-descriptor macro generates a memory subsystem command by calculating values for the f, a, and ℓ fields (internal parameters). It determines the values for each of these fields by specifying in its body either a constant for the value of the field or the name of an IFL program. In the latter case, the named IFL program is called with the two parameters in the macro calling sequence, and the value returned by the IFL program becomes the value of the field. The IFL program will be executed at the time of either macro expansion or macro execution depending upon whether the value of the internal parameter calculated by the IFL program is a constant for all executions of the generated memory subsystem command.

The IFL program can involve an arbitrarily complex computation and, additionally, as seen in Fig. 2, can access the memory subsystem for data. Thus, the generation of a memory subsystem command, especially the calculation of the address field, a, can be either a simple or complex calculation, depending upon the nature of the IFL program invoked. The data descriptor macro calling sequence, when expanded, is represented by the following figure:



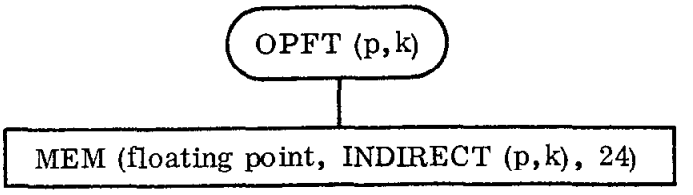
*a box will represent a terminal node

Example 3*

Consider a computer with a 24 bit word in floating point format, and with an instruction format in which bits 0-6 are the op code, bit 7 is an indirect bit, and 8-23 are the address of the next word of the indirect chain. A data-descriptor macro, OPFT, which generates a memory subsystem command that retrieves the desired data-item can be specified in the following manner: Let the p parameter of the macro be the virtual address of an instruction of the emulated computer; the body of OPFT is defined such that the f field is a constant that specifies the floating point data-format, the ℓ field is the constant 24, and the address field, a, is

*Examples 3, 4, 5, 7 and 8 form an integrated sequence that defines the control data structure of an idealized von Neumann computer pictured in Fig. 4 on page 32.

calculated by an IFL program, (INDIRECT) which, using the parameter p, generates the bit address of the last element of the indirect chain. The expansion of the macro calling sequence OPFT (p,k) is then represented by the following figure:

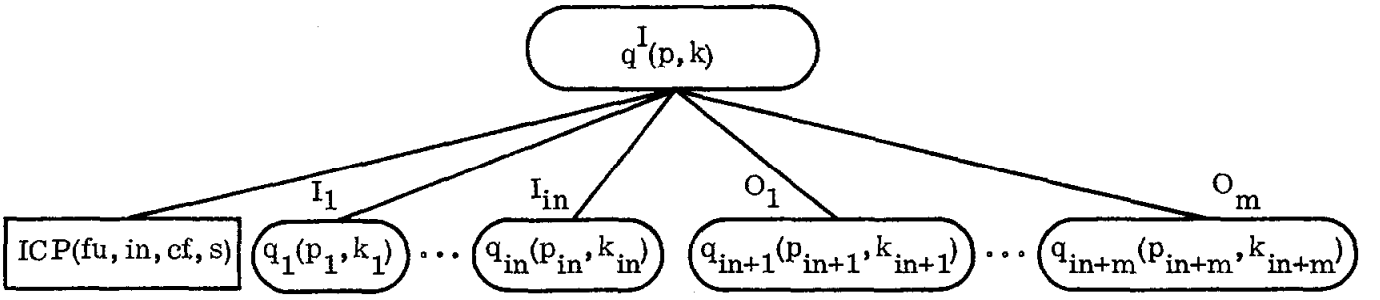


The IFL program INDIRECT is not invoked at macro expansion time but rather at macro execution time since the address field, a, of the memory subsystem command will be recalculated for each execution of the macro OPFT.

The instruction macro, when expanded, generates an I/O control structure that defines the interaction between a functional unit and the memory subsystem. The basic form of the I/O control structure generated by the instruction macro is very similar to the basic form of the control structure generated by the hierarchical macro; that is, a group of son nodes together with a clocking process. The basic difference between these two types of control structures is the format of the clocking process that is used to sequence the son nodes. The hierarchical macro clocking process is an arbitrary process while the instruction macro clocking process has a fixed format. The son nodes of an instruction macro specify the data-accessing procedures which fetch (store) the input (output) data sets of the functional unit. The built-in clocking process of the instruction macro, ICP, is activated with four internal parameters: fu, the name of a functional unit*; in, the number of input set generator nodes (the number of output set generators are the remaining son nodes); cf, control information sent to the functional unit; s, an address in the memory subsystem where the status of the functional unit at the termination of its operation is stored. The internal parameters fu, cf, and s can, if desired, be recalculated for each execution of the

*fu can also refer to an IFL program which simulates the action of a functional unit. The use of a pseudo-functional unit will be discussed in V.D.

instruction macro. However, the parameter, in, can be only calculated at macro expansion time since it relates to the form of the I/O control structure. The instruction macro calling sequence, when expanded, is represented by the following figure:

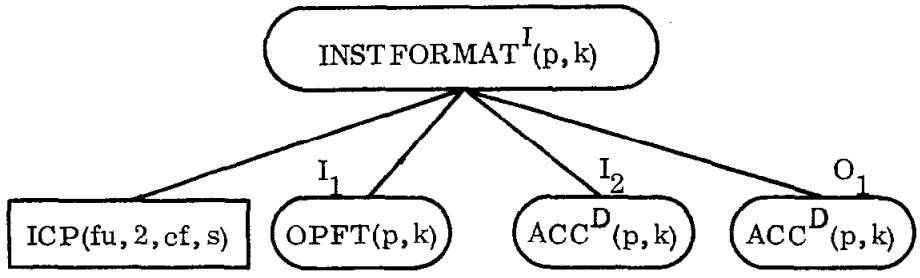


The clocking process ICP when executed, activates the functional unit fu with control information cf, and then waits for a request by the functional unit for input or output data. When input data is requested, the calling sequence $q_1(p_1,k_1)$ is activated to generate a single input value. Upon further requests for input $q_1(p_1,k_1)$ is executed again until it produces no more data (e.g., it is terminated) and then $q_2(p_2,k_2)$ is activated. The same process is then repeated with $q_2(p_2,k_2)$. If an output is requested, $q_{in+1}(p_{in+1},k_{in+1})$ is activated to store a value. Upon further requests for output, an analogous process to the input case just described is carried out. A functional unit can also operate in the mode where it requests all its input data simultaneously, in which case all the input generators $I_1 \ldots I_{in}$ are simultaneously activated to generate inputs. At the termination of operation of the functional unit, the status of the unit is stored starting at address s in the memory subsystem.

Example 4

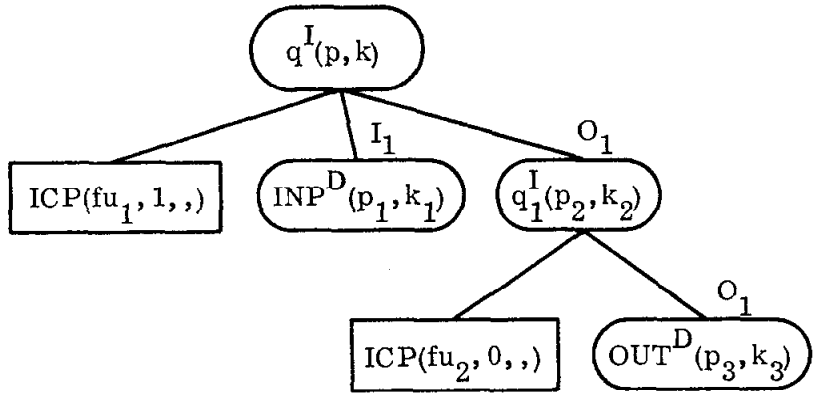
Consider the computer detailed in the previous example. An instruction macro $INSTFORMAT^I(p,k)$ which generates a functional unit subsystem command that emulates instructions of this computer can be defined in the following manner. Let the p parameter of the instruction macro be the virtual address of the instruction to be emulated, and assume that the implicit second operand and result operand of the instruction is the accumulator. The body of INSTFORMAT is defined such that the following

control structure is generated.



where fu is calculated by an IFL program, defined in the macro body INSTFORMAT^{I}, that extracts bits P0-P6 from the memory subsystem, and ACC^{D}(p,k) generates a fixed data-descriptor which represents the area in the memory subsystem set aside as the accumulator.

The instruction macro can also be used to construct I/O control structures that represent a pipeline of functional units. The pipelining of functional units makes unnecessary the use of the memory subsystem as a temporary storage buffer for data that passes directly from one functional unit to another. An example of a control structure for a two level pipeline (inp→ $\boxed{fu_1}$ → $\boxed{fu_2}$ → out) is the following:



The semantics associated with execution of this control structure is the following. The execution of q^{I} activates functional unit, fu_1, with input generated by INP^{D}. The output of fu_1 is then stored by q_1^{I}. But, q_1^{I} is an instruction macro. In that case, the output directed to q_1^{I} is sent as an input value to fu_2 after all the input data generators of q_1^{I} are exhausted. In this particular example, there are no input generators so that output of fu_1 is immediately gated into fu_2. Thus,

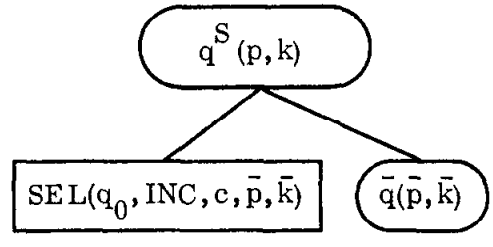
creating a two-level pipeline. Trees of functional units can also be created by this same mechanism; except in this case of a tree of functional units, the control structure is set up so that the instruction macro is requested to produce an input instead of storing an output. The output generated by the instruction macro is then outputted when all the output set generators of the functional unit are exhausted.

The semantics of the data-descriptor macro and the instruction macro have been chosen so as to clearly divorce the function of data-accessing from the computational algorithm (functional unit). This separation then facilitates 1) the definition of I/O control structures which directly emulate different types of IML instruction formats and 2) the incorporation of functional units into the functional unit subsystem that have complex input and output requirements (e.g., a matrix multiply unit, etc.).

E. Structure Building Macros

1. Sequential Control Structures

The selection macro serves the same purpose in the SBL as does the Case statement in ALGOL, the Computed Go To statement in FORTRAN, or the data-dependent jump instruction in machine language. The selection macro provides a mechanism which allows the conditional expansion of a node in the control data structure. In essence, the selection macro defines a one-level decoding tree which results in the generation of an arbitrary macro calling sequence. The expansion of a selection macro, $q^{S}(p,k)$, results in the generation of another macro $\bar{q}(\bar{p},\bar{k})$ where the values of $\bar{q}, \bar{p}$, and $\bar{k}$ are either constants specified in the macro body or are computed by an IFL program using p and k as parameters. The selection macro, when expanded, produces the following structure in the process space memory:



where SEL is a built-in control process with five internal parameters that generates and then executes the macro calling sequence $\bar{q}(\bar{p},\bar{k})$ as its brother node. The

internal parameter q_0 is an address in the program memory, and is added to the integer value, INC, so as to generate the address of macro $\bar{q}$. The parameter q_0 can be thought of as the base address of a vector of alternative processes while INC is an index into the vector that determines the desired alternative. The internal parameter q_0 relates to the form of the selection control structure, and thus cannot be computed after each new execution. The internal parameter c is control information that defines how the macro calling sequence $\bar{q}(\bar{p},\bar{k})$ will be activated when q^S is executed.

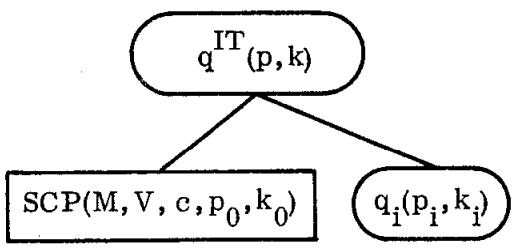
Example 5

Consider a computer with several different instruction formats. The emulation of instructions of this computer could be programmed by having a separate instruction macro $INSTFORMAT^I_J$, for each instruction format J. A selection macro $INSTDECODE^S$ could then be used to select the correct instruction macro for each emulated instruction.

The iteration macro serves the same purpose in the SBL as does the FOR-LOOP in ALGOL, the DO-LOOP in FORTRAN, or the MAPCAR function in LISP. The iteration macro provides a mechanism for building sequential processes. An iteration macro, $q^{IT}(p,k)$, defines a sequential process by generating and executing a list of macro calling sequences:

$$q_1(p_1,k_1),\ q_2(p_2,k_2)\ \ldots\ q_i(p_i,k_i),\ q_{i+1}(p_{i+1}k_{i+1})\ \ldots\ q_n(p_n,k_n);$$

The iteration macro defines only a sequential process because each macro calling sequence $q_i(p_i,k_i)$ is completely executed before the generation of the next calling sequence $q_{i+1}(p_{i+1},k_{i+1})$. The iteration macro, q^{IT}, when expanded produces the following structure in the process space memory;



where SCP (Sequential Clocking Process) is a built-in clocking process that generates and then executes successive elements of the list of macro calling sequences. The SCP, after the generation of each calling sequence $q_i(p_i,k_i)$, then executes this calling sequence as its brother node. The iteration macro may be activated by a control macro so that only a single macro calling sequence $q_i(p_i,k_i)$ is executed, and then after the termination or suspension of this calling sequence the iteration macro is suspended. Upon reactivation of the suspended iteration macro, depending upon whether $q_i(p_i,k_i)$ is terminated or suspended, respectively, either the next calling sequence $q_{i+1}(p_{i+1},k_{i+1})$ will be generated and then executed or else $q_i(p_i,k_i)$ will be reactivated.

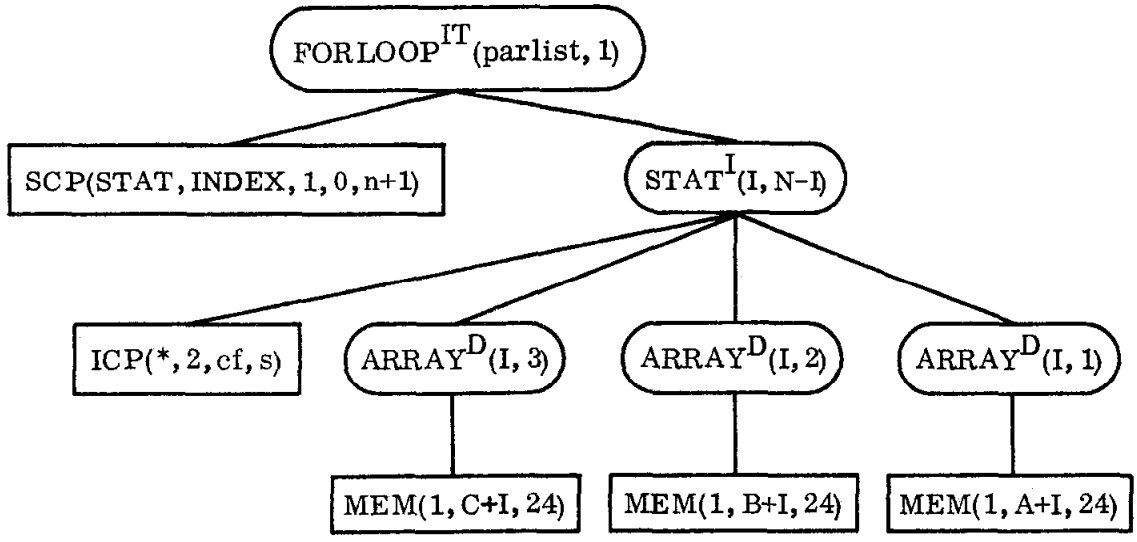
The clocking process SCP is activated with five internal parameters: the first two parameters, M and V, are the addresses of IFL programs; the third parameter, c, specifies control information; the remaining parameters p_0,k_0 are used to construct the initial calling sequence in the list. The M program called with parameters (p_i,k_i) computes q_{i+1}, the location of a macro. The V program, also called with parameters (p_i,k_i), computes (p_{i+1},k_{i+1}), which are the corresponding parameters for q_{i+1}. The M and V internal parameters relate to the form of the iteration control structure and thus cannot be varied from execution to execution. The clocking process SCP terminates the generation of calling sequences when $k_{n+1} = 0$.

Example 6

Consider the Algol Procedure:

```
PROCEDURE FORLOOP (A, B, C, N);
ARRAY A [1:N], B [1:N], C [1:N];
INTEGER I;
FOR I ← 1 step 1 until N
DO A [I] ← B [I] * C [I];
END
```

This procedure can be represented in terms of the following control data structure:



where parlist is a pointer to the parameter list (A, B, C, N); INDEX is an IFL program that generates the sequence of pairs (1, N) (2, N-1) ... (N, 1); and ARRAY is a data-descriptor macro that retrieves (stores) the ith word of an array. It is assumed the data elements of the array are 24 bits in width. This control structure, once expanded, need not be reconstructed for further procedure calls, only the value of parameters A, B, C, and N need be recomputed on each execution.

The control information c is used to define how the macro calling sequence will be activated; namely, if q_i is itself an iteration macro, whether it will be activated either for a single cycle and then suspended, or whether it will be activated for the entire list of macro calling sequences and then terminated. Thus, the time grain (smallest unit of work which can be controlled) of a control structure that is constructed out of a series of successive functional decomposition of a sequential process can be set at any desired level in the decomposition.

Example 6A

Consider the iteration macro, $A^{IT}(p,k)$, which when executed generates and executes the following list of macro calling sequences $B^{IT}(p_1,k_1)$, ..., $B^{IT}(p_n,k_n)$. Likewise, consider $B^{IT}(p_i,k_i)$ which when executed generates

and executes the following list of macro calling sequences $C^D(\bar{p}_1,\bar{k}_1)$, ..., $C^D(\bar{p}_m,\bar{k}_m)$. If the iteration macro A^{IT} is executed for a single cycle, and the c parameter associated with SCP node of A is set for a single cycle execute, then A^{IT} will be suspended after the completion of each data-descriptor macro $C^D(\bar{p}_i,\bar{k}_i)$. Thus, in this above case, the time grain of A^{IT} is the complete execution of macro C^D. While if the c parameter is set for execution until termination, then A^{IT} when executed for a single cycle will be suspended after the termination of iteration macro $B^{IT}(p_i,k_i)$. Thus, in this latter case, the time grain of A^{IT} is the complete execution of B^{IT}.

Another important property of the iterated macro is that generation of the macro calling sequence $q_{i+1}(p_{i+1},k_{i+1})$ may be affected by the results of executing the macro calling sequences $q_1(p_1,k_1)$ ... $q_i(p_i,k_i)$. The execution of a macro may produce side effects by modifying the contents of the memory subsystem or the control data structure which in turn may effect the execution of the M and V programs. This ability to alter the generation pattern of iteration macro via side effects is crucial to defining the sequencing of machine language instructions.

Example 7

Consider an iteration macro $INSTEXEC^{IT}(p,k)$ which generates the following sequence: $INSTDECODE^S(p_1,k_1)$, ... $INSTDECODE^S(p_i,k_i)$, ... where p_i is interpreted as the address of an instruction of an emulated computer, and k_i is the state vector of the emulated computer. The selection macro $INSTDECODE^S$ in turn generates an instructor macro $INSTFORMAT^I_J(p_i,k_i)$, where J refers to the format of the instruction stored at p_i. $INSTFORMAT^I_J$ when executed carries out the semantics of the instruction at location p_i. Therefore, the iterated macro can be thought of as the sequencing unit of a computer, the selection macro as the decode unit, and the instruction macro as the arithmetic and logic unit. This control structure in this example can be very easily extended to include an interrupt structure. All that is required is to set up a clocking process that activates $INSTEXEC^{IT}$ for one cycle at a time, and then checks whether an interrupt requires processing. In this case, the time grain is set as the execution of a single emulated instruction.

The iteration macro can also be used to construct data-accessing procedures when $q_i(p_i,k_i)$ is a data-descriptor macro calling sequence. The iteration macro in this case can be considered an operand name generator and the data-descriptor macro a value generator. An additional use of the iteration macro is the building up of a co-routine structure since the iterated macro holds its state when suspended. By combining these two uses of the iterated macro (as a data-accessing procedure and a co-routine), a stack data-accessing structure can be constructed.

2. Nonsequential Control Structures

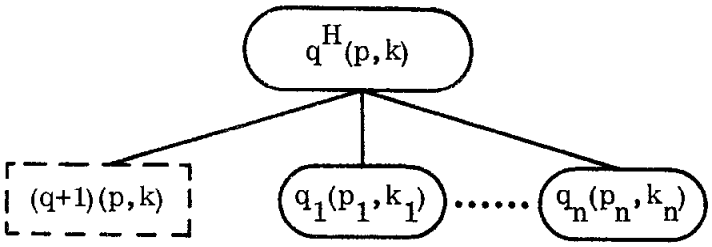
The hierarchical macro provides a mechanism for defining control structures that contain more than one clocking process (path of control),[12] especially control structures that distribute control through a hierarchy of control levels. A distributed control structure, constructed by a sequence of hierarchical macros, can be used to define, depending upon the number of clocking processes that are simultaneously executed, either quasi-parallel[13] or parallel control structures. In addition, many sequential control structures can also be easily defined in terms of a distributed (quasi-parallel) control structure, e.g., a subroutine call mechanism: the execution of the subroutine call suspends the clocking process of the caller, and activates the clocking process of the subroutine; the return from the subroutine then terminates the clocking process of the subroutine and reactivates the clocking process of the caller. The block structure and procedure calls of ALGOL and co-routines are other examples of sequential distributed control structures. In essence, the hierarchical macro allows the structure of a complex process to be functionally decomposed into a set of executions of less complex processes. Thus, the hierarchical macro, in order to represent this functional decomposition, must define (1) the set of less complex processes, and (2) the sequencing algorithm (clocking process) for this set of processes.

The hierarchical macro, $q^H(p,k)$, when expanded, generates a list of macro calling sequences:

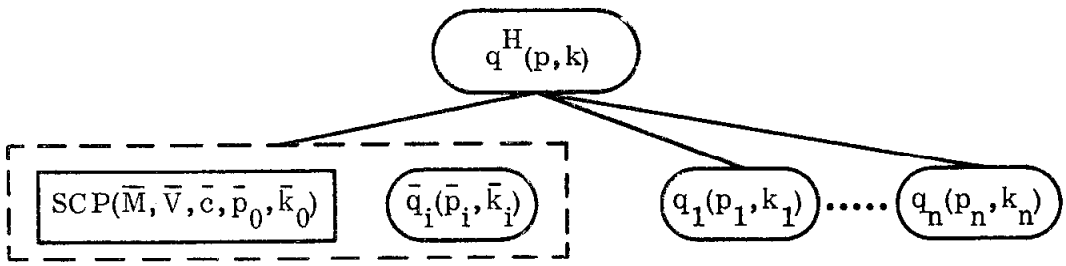
$$q_1(p_1,k_1),\ q_2(p_2,k_2)\ \ldots,\ q_n(p_n,k_n)$$

and then expands a macro calling sequence $(q+1)(p,k)$. The macro $(q+1)$ is a clocking process that controls through the execution of control macros the initial sequencing of the list of macro calling sequences. The list of macro calling sequences is generated using the same mechanism, $SCP(M,V,c,p_0,k_o)$, employed by the iterated macro to generate a list. Except, in this case, the generation

pattern of the list cannot be altered through side effects since a macro calling sequence in the list is not executed until the entire list is generated. The control field c in SCP in the case of hierarchical macro is used to define a default value for control information associated with the execution of each $q_i(p_i,k_i)$. The list of macro calling sequences after its generation is stored as son nodes of the hierarchical macro in the control data structure. The expansion of a hierarchical macro results in the generation of the following structure in the process space memory:



The macro calling sequence $(q+1)(p,k)$ is enclosed in a dotted box to indicate that the results of expanding the calling sequence $(q+1)(p,k)$ is placed in the process space memory rather than the actual calling sequence $(q+1)(p,k)$. Thus, if $(q+1)$ is an iteration macro, then the expansion of $q^H(p,k)$ would result in the following control data structure:



The execution of $q^H(p,k)$ in this above case results in the execution of the built-in clocking process $SCP(\bar{M},\bar{V},\bar{c},\bar{p}_0,\bar{k}_0)$ which sequentially generates and executes a list of macros calling sequences $\bar{q}_1(\bar{p}_1,\bar{k}_1) \ldots \bar{q}_i(\bar{p}_i,\bar{k}_i) \ldots$. The results of executing this list of macro calling sequences, in turn, define the initial sequencing of $q_1(p_1,k_1) \ldots q_n(p_n,k_n)$. The clocking process call sequence $(q+1)(p,k)$ does not have any characteristics which distinguish it from other processes defined by the SBL. Thus, a clocking process can be of arbitrary complexity and only the parts of its structure which are changed on each execution need be modified. A

tree of arbitrary width and depth can then be dynamically generated since the macro q_i may itself be a hierarchical macro.

Example 8

Consider the emulation of a conventional von Neumann computer organization with an interrupt structure. The basic form of the control structure for an emulator for this type of computer can be constructed by combining together the control structures discussed in examples 3, 4, 5, and 7, and then adding a hierarchical macro that specifies the interrupt structure. Figure 4 represents this control structure, where SEQUNIT is a clocking

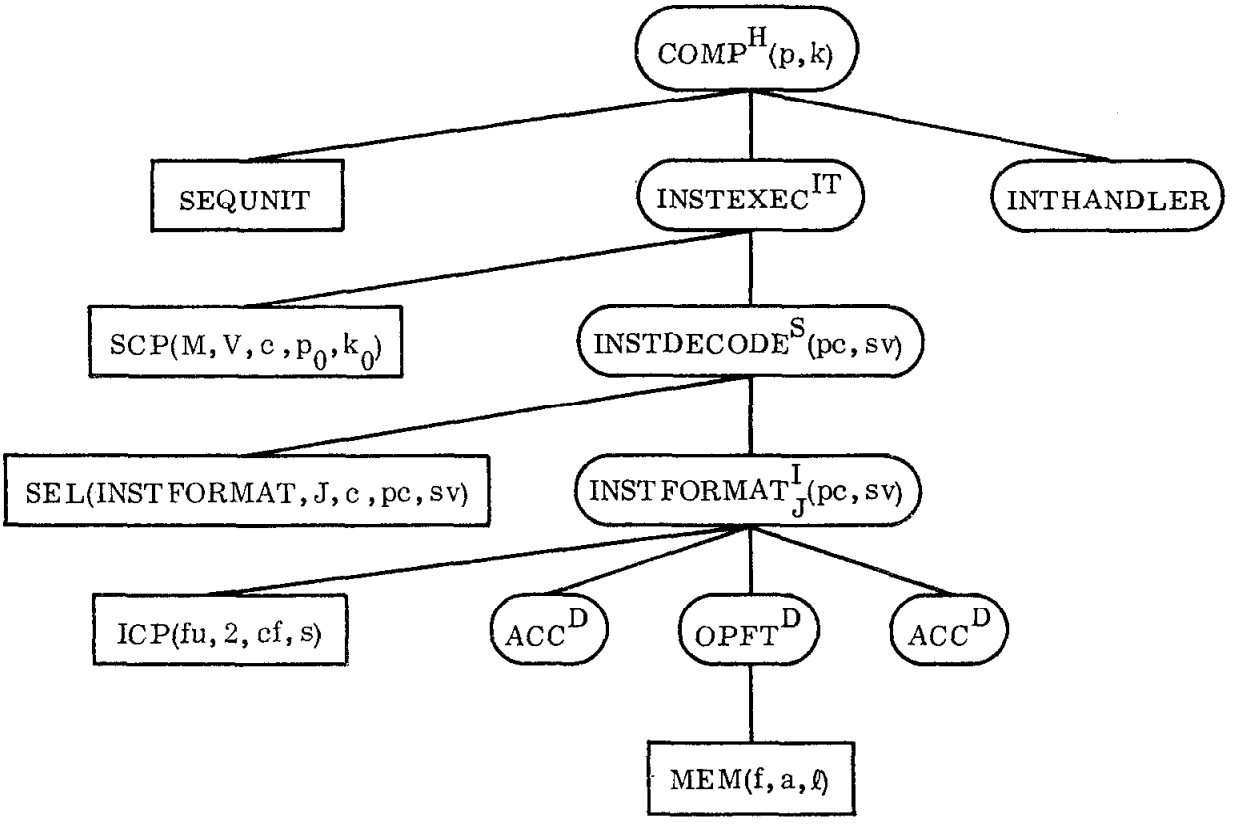


FIG. 4--The control data structure for an emulator of a von Neumann computer organization with interrupt.

process that activates $INSTEXEC^{IT}$ for one cycle (instruction) at a time, and then checks whether an interrupt requires servicing; if it does, then INTHANDLER is executed, else $INSTEXEC^{IT}$ is reactivated and the basic sequencing cycle is repeated.

The hierarchical macro can also be used to construct distributed control structures which are not conventionally represented in terms of a tree structure. Nontree like control structures can be represented, because, as previously discussed, the dynamic sequencing of the tree (which is defined by clocking processes of arbitrary complexity) is separated from the generation of the tree structure. The sequencing of sibling nodes is, therefore, not restricted to a predefined set of built-in sequencing patterns since the clocking process is an arbitrary program. In addition, the time grain of a process defined by a hierarchical macro also can be arbitrary since the time grain of the clocking process is programmable.

Example 9

Consider the parallel control structure defined by a fork-join instruction.[14] The fork-join control structure is normally represented in terms of the directed graph in Fig. 5a. However, if the correct clocking processes are attached to a tree of processes, then the fork-join control structure can be represented in terms of a tree, as viewed in Fig. 5b: the clocking process Control-1 sequentially executes the process specified by macros "PARL AB" and C. Control-2 clocking process executes processes A and B in parallel, and is not terminated until both processes A and B are terminated.

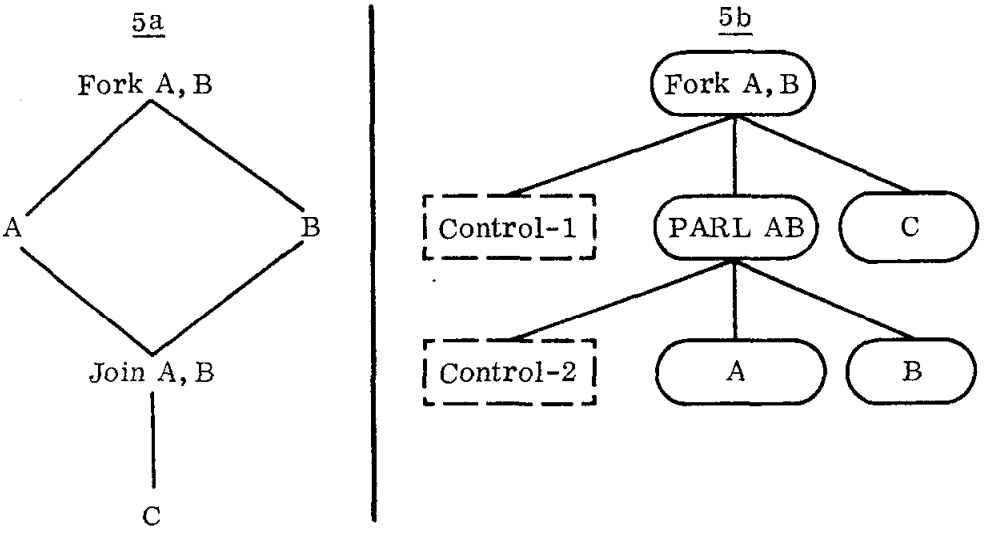


FIG. 5--Fork-join instruction.

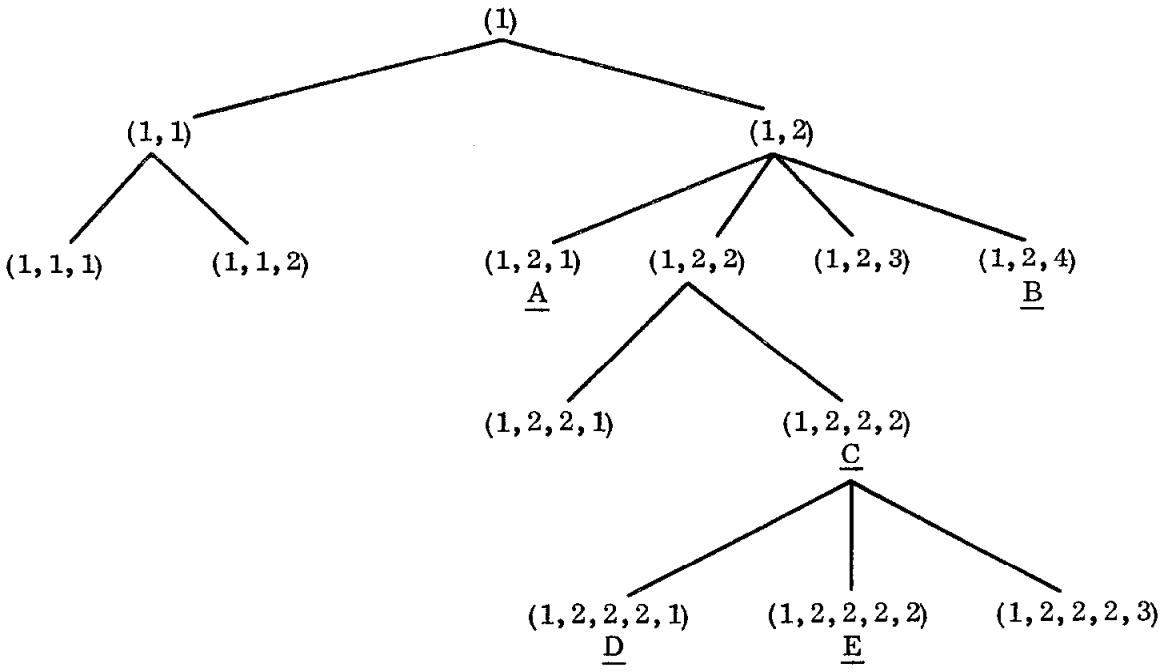
3. Tree Structured Addressing

The control macro and IFL refer to (address) processes (macro calling sequences) in the process space memory either through their absolute location in the process space memory or their relative location in the control data structure tree with respect to the address of either the control macro calling sequence or the macro calling sequence that invokes the IFL program. In general, a node in an arbitrary tree structure requires k parameters to specify its address uniquely, where k is the depth of the node in the tree. However, by employing relative addressing for node specification and restricting the part of the tree that can be addressed from any node, the address of a process can be specified in terms of two parameters. The restriction on accessing only part of the tree corresponds very closely to the restriction placed on accessing variables in a nested block structure in ALGOL and is not a serious practical limitation. Further, this relative addressing mode, if necessary, can be overridden by using absolute addressing node.

The relative addressing schema is a two step process, each step using one of the parameters. The first step, using a parameter to indicate the number of times applies the father (antecedent) relation recursively to the relative base node. The second step, using a parameter to specify the number of the brother, locates a particular brother of the node which results from the first step. The address schema, where (n,l) are the two parameters, can then be specified by the following formula: (brother^{l}.father^{n}.base-node). In the case of the absolute address node, the addressing schema is (brother^{l}.n) where the parameter N is the absolute address of a node.

Example 10

Consider the following tree:



then using E(1,2,2,2,2) as a relative base node

(2,-1) addresses A (1,2,1)

(2,2) addresses B (1,2,4)

(1,0) addresses C (1,2,2,2)

(0,-1) addresses D (1,2,2,2,1)

In general, if a base node address is $(a_1, a_2, \ldots a_n)$ then relative address (i,j) refers to node $(a_1, a_2, \ldots a_{(n-i-1)}, (a_{(n-i)}+j))$.

This relative address capability can be used very advantageously in the definition of recursive distributed control structures since a clocking process does not have to know the exact level of the tree it is controlling. Thus, the copies of a single clocking process can be used to control different levels of the tree.

4. Synchronization, and Control and Data Linkage Among Processes

The previous sections in this chapter have described the form, the method for constructing and the addressing structure of the control data structure. This section will now detail how the control macro, which is the basic building block

of clocking processes, uses the control data structure as a syntactic framework within which to define nonsequential control structures.

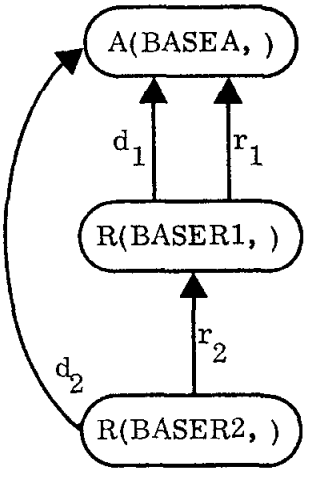
The control macro combines the control functions of process activation (including parameter passage) and process synchronization. The control macro performs these control functions through operations on the data stored at a node in the process space memory. This data can be considered the state vector of a process, where the process is defined by the control structure generated by the macro calling sequence stored at the node. This process state vector contains seven components (q,p,k,s,c,r,d) where q,p, and k is a macro calling sequence, s is the current state of the process, c is control information associated with the activation of the process, and r and d are pointers to nodes that, respectively, define the immediate global control and data environment of the process. The control information, c, specifies the time grain of the process, the conditions for the process signalling its external clocking process, and the conditions for rebuilding the process' control structure; the time grain of a process can be defined to be the execution of the process' internal clocking process for either a single cycle or until it is terminated; the time grain of a process defines at what points a process' activity can be suspended. A process can signal its external clocking process when the process' state is expanded, suspended, terminated, or either suspended or terminated. The immediate global control environment pointer, r, conventionally called a return link specifies the address of this external clocking process that will be signalled. The c component also specifies whether a process' control structure will be partially rebuilt after each execution of the process, or either partially or completely rebuilt after the process is terminated. The immediate data environment pointer, d, is used by the tree address mechanism to locate nodes in the process space memory. The values of r and d when a node is initially generated are, respectively, the addresses of node's clocking process and father. However, these default options for r and d can be overridden by the control macro so as to create a control data structure for the passage of control which is not a tree structure.

Example 11

Consider the following ALGOL program, discussed in a report by Shaw[15]:

```
A: begin real a_1, ..., a_n;
   procedure r;
   begin real r_1, ..., r_m;
      .
      .
      .
     R: r
   end r;
A1: r;
end A
```

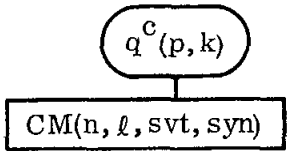
r is called at A1; after r is called recursively for the first time at R, the control data structure has the following form:



In this example, the immediate global data environment is the same for both instantiations of R. This example points up the distinction between the control and data environment of a process, and the necessity for being able to construct each of these environments independent of one another and to override the normal tree structure discipline for representing control and data relationships among processes.

A node contains, in addition to process state vector, the address of the node's first son. The locations of a node's siblings are implicitly defined since sibling nodes are stored in consecutive locations in the process space memory.

The control macro, $q^c(p,k)$, when expanded, generates the following control structure:



where CM is a built-in clocking process with four internal parameters. The first two parameters, n and l specify the relative address of a node in the control data structure. The third parameter, svt, is a template for a process state vector where for each of the component of vector there is stored in the template either a value or null symbol. The fourth parameter, syn, is used to synchronize the activity of the control macro with the activity of the process located at (n,l).

The execution of the built-in clocking process CM results in the modification of the state vector of the process located at relative address (n,l) in the process space memory. This process' state vector is modified by replacing the value of each of its components by the corresponding svt component whenever this corresponding svt component is not null. Thus, only the components of the state vector of the activated process which vary from execution to execution of the process need be recalculated and then replaced by the control macro. The static components of a process state vector (the fixed control and data linkages of a process) are defined either by default options when the process' state vector is initially generated or by the control macro which initially expands the macro calling sequence that defines control structure of the process. Thereafter, the control macro that activates the process has a template state vector whose components are null whenever the corresponding components of the process' state vector are static. At the same time as the modification of the process' state vector is completed, the s component of the state vector of the CM clocking process is modified, depending upon the syn parameter, to be either the suspended or the terminated state. Through this mechanism of simultaneous modifying of two state vectors, the activity of one process can be synchronized with activity of another process.

Example 12

Consider two processes A and B, where process A calls process B as a subroutine. Process A performs the subroutine call by executing and then waiting for termination of a CM clocking process. In turn, the CM clocking process activates the process B and modifies B's state vector so that process B will signal a return when it is terminated, and this return will be to the CM clocking process. At the same time, the syn parameter of CM is set up so that after process B's state vector is modified the CM clocking process is suspended. When process B is terminated, CM will then be re-awoken and will go to the terminated state. This action in turn will allow process A to continue processing since process A has been waiting on the completion of the CM clocking process. If process A was not synchronized with the activity of process B then syn parameter of CM would be set up so that after process B is activated the CM process is terminated. Thus, process A after process B is activated will immediately continue processing. Process A while waiting for CM process to terminate is not suspended because the action of suspending process A may be significant to A's external clocking process since the suspending of A means that process A has completed a time grain. Thus, this implementation of subroutine call permits A's external clocking process to view A as executing while process B is executing, but at the same time A's internal clocking process is waiting on B's completion.

The CM clocking process can only activate a process for execution (e.g., change the s component of the process' state vector to executing) when the process' current state is unexpanded, expanded, suspended or terminated. In the case that CM clocking process attempts to execute an already executing process, the CM clocking process either is suspended or goes into a busy wait until the process to be executed is no longer executing. The time grain of the node that generates the CM determines which one of these options is taken: if the time grain is a single cycle the CM is suspended, otherwise it busy waits. Thus, if two processes simultaneously issue CM's which activate the same node (shared process), only one CM will be allowed to execute the shared process. The other CM will then either wait till the shared process is completed, or possibly at some later time try to

execute the shared process. This paradigm for sequentializing the execution of a shared process can then be used as basis for constructing synchronizing primitives for cooperating processes.

Example 14

Consider the implementation of Dykstra's P and V semaphores in terms of the CM clocking process. Let PV be a shared process where the p component of its state vector is the name of semaphore variable to be operated on, the k component of its state vector indicates whether a P or V operation is to be performed, and the r component is the address of the process that activated PV. A process L_i performs a P or V semaphore operation by generating a CM clocking process whose time grain is termination, syn parameter in the case of P operation specifies suspended while for a V operation specifies terminated, (n, 1) parameters specify the relative address of the PV process, and the svt contains the correct calling sequence for either a P or V operation. The PV process when executed by CM for a P operation checks whether the semaphore variable specified in the calling sequence can be decremented, if it can, then the operation is completed and the PV process is suspended. This suspension of PV results in termination of CM which then permits process L_i to continue. In the case that semaphore can not be decremented, the PV process modifies its own state vector component so that it does not return to CM when it is suspended. It then extracts the address of the CM process from its state vector, places this address in queue associated with the semaphore name, and suspends itself. Thus, the CM clocking process still remains in the suspended state, and therefore process L_i can not continue. The PV process when executed for V operation increments the semaphore variable, and then checks whether there is a queued CM process on that semaphore variable that can now be executed. If there is, this CM process address is stored in the r component of PV state vector, and PV process then suspends itself which results in the queued CM process to be re-awoken. The CM clocking process that executed the PV process for a V operation terminates immediately after the PV process state vector has been modified, and thus L_i can continue processing while V operation is being done. If the PV process is busy, when CM attempts to execute it, then CM goes

into a busy wait, however, this busy wait is not on a semaphore variable but only on the process which updates the semaphore.

The CM can also be used to create a new copy of a process (node) instead of calling a shared process. This creation of new node occurs when the (n, l) parameter are (0, 0). The new node is the root node of a separate tree, and only the CM clocking process can access this tree. It may be also advisable, for efficiency reasons, for a CM clocking process to be able to simultaneously activate all the sibling nodes at level in tree, and then be able to wait for all of them to signal a return.

V. INTEGER FUNCTION LANGUAGE (IFL)

The IFL is a highly specialized micro-code language designed specially for the task of address arithmetic computations. The output of the address arithmetic computations performed by the IFL are then used in the expansion and execution of SBL macros. The format of IFL instructions and SBL macros are very similar; each is called with two parameters, and each has an expansion and execution phase. In fact, the execution of SBL* and IFL statements can be intermixed, and the same syntax will be used to define an IFL instruction calling sequence as is used to define an SBL macro calling sequence (e.g., $q^F(p,k)$, $(10)(p,5)$, etc., where F is for the convenience of the reader to differentiate an IFL calling sequence). The basic difference in a conceptual sense between the IFL and SBL is that the execution of an IFL instruction results in the execution of a function which returns an integer value whereas the execution of SBL macros results in the execution of a process. Thus, the IFL instruction can be considered a "functional macro". The basic reason for not defining address arithmetic algorithms in terms of an SBL control structure stems from the use of a different control structure for address arithmetic functions than that for processes. In particular, the control structure for defining address arithmetic functions can be much less complex and variable than that required for processes. Thus, the sequencing schema for IFL instructions is built-in rather than explicitly defined, as in the case of the sequencing for SBL macros. Address arithmetic algorithms can, therefore, be executed without the overhead of a variable control structure used for defining a process control structure.

Before formally defining the syntax and semantics of the IFL, it is worthwhile to note the following characteristics of the IFL which differentiate it from conventional micro-code instruction sets:

1. Parallelism: the IFL can execute, wherever appropriate, parallel (concurrent) activity in an address arithmetic computation.
2. Modularity: complex IFL program can be easily constructed out of calls to other IFL programs (the concept of a recursive function is an integral part of the control structure of the IFL); each IFL instruction is called with two parameters and then returns an integer value.

---

*The execution of an SBL macro in the context of an IFL program results in the expansion and then complete execution of the macro. After its termination, the next statement in the program memory is executed, and the control data structure in process space memory resulting from its execution is garbage collected.

3. Reentrancy: the IFL interpreter does not have a fixed set of registers but, instead, registers are dynamically created through the passage of parameters.
4. Simplicity: the syntax (format) and semantics of an IFL instruction are uniform (regular) and simple.

A. Format and Sequencing of IFL Instructions

The format of an IFL instruction q^F, which is stored in a word in the program memory, consists of five identically formatted fields, f_q, A_q, B_q, K_q, Q_q. The format of a field consists of an m bit interger*, c, plus a 2-bit descriptor field, d, that defines the method for calculating the value associated with the field; a field is represented by the following notation: (d,c). The value associated with a field is calculated, depending upon d, in one of the four following ways:

1. c;
2. p+c;
3. k+c;
4. $(q+c)^F$ (p,k) .

where p and k are the parameters used to call an IFL instruction $q:q^F(p,k)$. These three parameters, q, p and k, of an IFL instruction calling sequence, can be considered in terms of a conventional micro-computer organization as the current values, respectively, of the program counter, accumulator, and index register.

The expansion phase of the instruction calling sequence q(p,k) is the parallel evaluation of the five fields, f_q, A_q, B_q, K_q, Q_q, based on the above rules. These five fields have the following semantics associated with their values: f_q (the name of dyadic integer function which is the op-code of the IFL instruction q); A_q and B_q (parameters for the function f_q), K_q (a counter used to define termination of sequencing), and Q_q (an increment used to indicate the relative address with respect to q of the next IFL instruction).

---

*The maximum size of c which is dependent on m does not have to have an relationship to the maximum size of addressing space in the program memory or the memory subsystem. The choice of a value for m will be based on considerations of code density and speed of execution.

Example 14

Consider the IFL instruction FACT which has the following five field formats: (1,"*"), (2,0), (3,0), (3,-1), (1,0). The expansion phase of FACT(p,k) then results in the five fields having the following values: f_{FACT}=*(multiply), A_{FACT}=p, B_{FACT}=k, K_{FACT}K,-1, and Q_{FACT}=0.

These five values generated by the expansion phase are then used in the execution phase of q(p,k) to define the execution semantics and sequencing of IFL instructions in the following manner:

$$q(p,k)^{F} \leftarrow \text{if } k=0 \text{ then } p \text{ else } (q+Q_q)^{F}\,(f_q(A_q,B_q),K_q)$$

This paradigm for execution of IFL instructions results in the generation of a sequence of triplets: $(q_0,p_0,k_0)\ (q_1,p_1,k_1)\ \ldots\ (q_n,p_n,k_n)\ (q_{n+1},p_{n+1},0)$, where q_0=q, p_0=p, k_0=k, and for $i \geq 0$, $p_{i+1} = f_{q_i}(A_{q_i},B_{q_i})$, $k_{i+1}=K_{q_i}$ and $q_{i+1}=q_i+Q_{q_i}$. The value p_{n+1} is then returned to the process that called q_0.

Example 15

Consider the IFL instruction, FACT, discussed in Example 14. The execution of the calling sequence FACT(p,k) then results in the following calculation:

$$FACT(p,k) \leftarrow \text{if } k=0 \text{ then } p \text{ else } (FACT+0)\quad (p*k,\ k-1)$$

If p and k have the following initial values of, respectively, 1 and N, then FACT(1,N) when executed calculates N! in the following iterative manner:

$$FACT(1,N) \leftarrow \text{if } N=0 \text{ then } 1 \text{ else } (FACT+0)\,(N*1,N-1) \equiv FACT(N,N-1)$$

$$\vdots$$

$$FACT(N!,i) \leftarrow \text{if } i=0 \text{ then } N!/i! \text{ else } FACT(N!/(i-1)!,i-1)$$

$$\vdots$$

$$FACT(N!/0!,0)=N!$$

This calculation generates a sequence of triplets: (FACT,1,N) (FACT,N,N-1) ... (FACT,N!/i!,i) ... (FACT,N!,0).

The sequencing part of the execution paradigm for IFL instructions is very general, and allows as special cases, iterative, straight-line, and conditional sequencing of IFL instructions, plus a value return mechanism.

1. iterative sequencing, as seen in example 15, occurs when $Q_q \equiv 0$, then $q(p,k) \leftarrow$ if k=0 then p else $q(p_1,k_1)$.
2. straight line sequencing occurs when $Q_q \equiv 1$, then $q(p,k) \leftarrow$ if k=0 then p else $(q+1)(p_1,k_1)$; in addition, if k is always not zero when q is called, then $q(p,k) \leftarrow (q+1)(p_1,k_1)$.
3. conditional sequencing occurs when the field Q_q is an expression rather than a constant.
4. a value return occurs when $K_q=0$, then $q(p,k) \leftarrow$ if k=0 then p else $(q+Q_q)(p_1,0)$, but $(q+Q_q)(p_1,0) \equiv p_1$, and thus $q(p,k) \leftarrow$ if k=0 then p else p_1; in addition, if k is always not zero when q is called, then $q(p,k) \leftarrow p_1$.

In addition, parallel and recursive sequencing of IFL instructions may be programmed. Parallel and recursive sequences of IFL instructions arise because the values associated with five fields can be calculated in parallel and may result (when the descriptor, d, of a field is equal to 4) in the calling of an other IFL instruction.

Example 16

Consider the following two IFL instructions, X and Y, which have the following field values:

$X = \{f_x=*,\ A_x=Y(p,k),\ B_x=k,\ K_x=0,\ Q_x=0\}$
$Y = \{f_y=+,\ A_y=p,\ B_y=0,\ K_y=k-1,\ Q_y=-1\}$
where Y = (X+1)

The execution of the calling sequence X(p,k) then results in the following calculation:

$X(p,k) \leftarrow$ if k=0 then p else $(X+0)\ (Y(p,k)*k, 0)$ which is equivalent to
$X(p,k) \leftarrow$ if k=0 then p else $Y(p,k)*k$, where
$Y(p,k) \leftarrow$ if k=0 then p else $(Y-1)(p,k-1)$, however
X(p,k) only calls Y(p,k) when $k \neq 0$, and also $(Y-1)=X$.
Thus $Y(p,k) \leftarrow X(p,k-1)$; and then
$X(p,k) \leftarrow$ if k=0 then p else $X(p,k-1)*k$
If p and k are initially, respectively, 1 and N, then X(1,N) calculates N! in a recursive manner since:
$X(1,N) \leftarrow$ if N=0 then 1 else $X(1,N-1)*N$.

The SBL and IFL interact through the generation by the SBL of an IFL calling sequence, $q^F(p,k)$. The result of executing this calling sequence is the return of either a single value p_{n+1}, or each pair in the sequence (p_1,k_1) ... (p_n,k_n). This latter type of return is used to define the execution of the iteration macro and results in a co-routine type interaction between the SBL and IFL since the IFL program is suspended after each pair (p_i,k_i) is generated.

Example 17

Consider the sequence of pairs (1,N)(2,N-1) ... (I,N-I+1) ... (N,1) used in defining the iteration macro that represents the ALGOL statement for I←1 step 1 until N do A[I]←B[I]*C[I]. This sequence of pairs can be generated by the IFL instruction, INDEX, which has the following field values, f_{INDEX}="+", A_{INDEX}=p, B_{INDEX}=1, K_{INDEX}=k-1, and Q_{INDEX}=0. The IFL instruction calling sequence INDEX(0,N+1) when executed then generates the following sequence of pairs: (1,N) ... (i,N+1-i), (i+1,N-i) ... (N,1) in the following manner:

INDEX(i,N+1-i)←if (N+1-i)=0 then i else INDEX(i+1,N-i)

In order to clarify the discussion of IFL programs presented in later sections, the assembler notation specified in Table 1 will be used for describing IFL instructions and programs. This symbolic notation for IFL instructions can be mapped directly (one-one) into actual IFL instructions. The major purpose of the assembler notation is to represent the special IFL sequencing cases, previously described, with a symbolic notation that indicates each of the special cases. Table 2 indicates these relationships between assembler syntax and special cases of IFL sequencing.

Example 17A

Consider the IFL instructions FACT,X,Y discussed previously. These IFL instructions can be symbolic represented as follows:

FACT:ITERATE(p:=P*k, k:=k-1);
X: if k=0 then p else [Y]*k;
Y: k:=k-1, go to X .

TABLE 1: Syntax of IFL Assembler

<IFL-PROGRAM> ← <IFL-STATEMENT>; /<IFL-STATEMENT>; <IFL-PROGRAM>

<IFL-STATEMENT> ← <STATEMENT-LABEL>: <STATEMENT>

<STATEMENT> ← if k=0 then p else <IFL-INSTRUCTION>/<IFL-INSTRUCTION>

<IFL-INSTRUCTION> ← <P-OP>, <K-OP>, <Q-OP> / ITERATE (<P-OP>, <K-OP>)/<PEXP>

<P-OP> ← p: = <PEXP>

<K-OP> ← k: = <EXP-FIELD>

<Q-OP> ← go to <STATEMENT-LABEL>/go to <EXP-FIELD> (<LIST-STAT>)

<LIST-STAT> ← <STATEMENT-LABEL>/<STATEMENT-LABEL>, <LIST-STAT>

<PEXP> ← <EXP-FIELD> <INFIX> <EXP-FIELD> / <PREFIX> (<EXP-FIELD>, <EXP-FIELD>)

<EXP-FIELD> ← <C> / - <C> / p / p + <C> / p - <C> / k / + <C> / k - <C> / <STATEMENT-LABEL>

<INFIX> ← + / * / - / > / < / = / ≠ / ...

<PREFIX> ← M/SHIFT/IA/IB/<EXP-FIELD>/...

<C> ← "intergers less than 2^{m-1}"

TABLE 2

Special Case	Syntax of Special Case
$k \neq 0$	<STATEMENT> ← <IFL-INSTRUCTION>
$Q_q \equiv 0$	<IFL-INSTRUCTION> ← ITERATE (<P-OP>, <K-OP>)
$Q_q \equiv 1$	<IFL-INSTRUCTION> ← <P-OP>, <K-OP>
$K_q \equiv 0$	<IFL-INSTRUCTION> ← <PEXP>

B. Built-In Arithmetic Operations

The types of functions which f_q field can represent fall into three classes: interger arithmetic operations, conditional and selection operations, and memory access operations. The interger arithmetic operations contain the conventional arithmetic and logical operations, concatenation, and shifting. Therefore, if f_q=+ then $f_q(A_q, B_q)$ equals A_q+B_q. The concatenation and shifting operators allows building a larger size constant from two smaller size constants or the combining of disjoint memory fields. The conditional operations test a condition between the two operands and, depending upon the satisfaction of the condition, produces either 0 or 1:

$$\text{if } f_q = \text{">"} \text{ then } f_q(A_q, B_q) = \text{ if } A_q > B_q \text{ then 1 else 0}$$

The selection operation, IA and IB, which are, respectively defined by field value for f_q of 1 or 0 have the following definition:

$$IA(A_q, B_q) = A_q \text{ and } IB(A_q, B_q) = B_q$$

The conditional and selection operators can then be combined to construct if-then-else arithmetic statements.

Example 18

Consider the function X(I) which has the following definition: if I> 5 then I else 5. The function X can be programmed in terms of two IFL instructions C and D, where

C: [D](p, 5);

D: p> 5;

Let C be called with parameter (I, 1) then $C(I, 1) \leftarrow [D(I, 1)] (I, 5)$, and $D(I, 1) \leftarrow$ if I> 5 then 1 else 0.

Since [1] $(I, 5) \equiv I_A(I, 5) = I$ and [0] $(I, 5) \equiv I_B(I, 5) = 5$ then $C(I, 1) \leftarrow$ if I> 5 then I else 5.

There are two types of memory access operators: one to access the memory subsystem and the other to access the process space memory. The memory subsystem operator $M(A_q, B_q)$ extracts from the memory subsystem starting at bit A_q a string of length B_q. This string of bits is interpreted as an integer value. The process space memory operations $P1(A_q, B_q)$, $P2(A_q, B_q)$ ... $P6(A_q, B_q)$,

retrieve, respectively, one of the six components of the process state vector located at relative address (A_q, B_q) in the process space memory.

Example 19

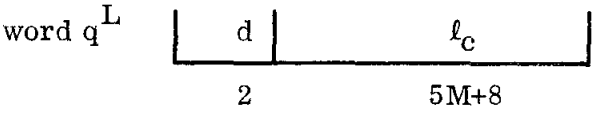
Consider the addressing structure of the PDP-6. Each PDP-6 word is 36 bits long and is divided into three fields for addressing: an indirect field, I, (Bit 13), an index field, B, (Bits 14-17), and an address field, A, (Bits 18-35). The index registers in the PDP-6 are the first 16 words in memory. The addressing structure of PDP-6 is indirect addressing with indexing at each level of the (arbitrarily long) indirect chain. The first problem is how to represent the 36-bit wide word memory in the memory subsystem. Let us lay out PDP-6 memory starting at bit 0 in the memory subsystem so that word K of the PDP-6 begins at address M[K*36] and ends at M[K*36+35]. The following IFL program determines the address of the last word in the indirect chain giving the address of the first word of the chain:

	Comments
PD6ADD: if k=0 then p else p:=p*36, go to [k] (CHAIN, EXTRACT-A);	Converts virtual address to physical address and then gets value associated with physical address
CHAIN:p:= [EXTRACT-A] + [EXTRACT-B], k:= [EXTRACT-I], go to PD6ADD;	Basic sequencing of indirect addressing
EXTRACT-A: M(p+18, 18);	Extracts address field
EXTRACT-B: p:=M(p+14, 4), k:=2, go to PD6ADD;	Extracts index field and then calls procedure to get value of index
EXTRACT-I: M(p+13, 1);	Extracts indirect field

The IFL program PD6ADD is called with parameters $(a_1, 1)$ where a_1, is the address of the first word of the chain.

Large size address constants can be generated by IFL instructions either through the concatenation of smaller size address constants or by storing

beforehand the constant in the memory subsystem and then, when the constant is required, extracting it from the memory subsystem. Though the above is a conceptually adequate solution, for reasons of execution efficiency and code density, an additional instruction format has been added to the IFL to handle large size address constants. This added type of instruction format, L, has the following format:



where d has the same semantics as it has in the basic instruction format of IFL instruction, and ℓ_c is an address constant which fills the rest of the program memory word. Thus, the value of $q^L(p,k)$ depending upon d is computed in one of the four possible ways: ℓ_c, $p+\ell_c$, $k+\ell_c$, or $(q+\ell_c)$ (p,k).

C. Side Effects in IFL

The IFL, as so far presented, is very similar in two significant ways to "pure LISP"; each creates temporary storage solely through parameter passages and each has no side effects other than the return of a value. These characteristics of the IFL, though theoretically interesting since they guarantee the determininacy of parallel IFL computations, severely limited the ability of this microcomputer to emulate existing computers. In particular, the programming of address arithmetic computations for emulator may involve more than just the return of a value; e.g., an effective address calculation may also involve checking for an address alignment error, and, if necessary, then updating the state vector of the emulated computer to indicate the addressing error. Thus, the IFL contains provisions for the programming of side effects.

The IFL contains memory operations which can modify the contents of either the memory subsystem or the process space memory. These memory operations SM, SP1, ... SP6, are the store counterparts, respectively, of the memory access operation M, P1, ... P6. The memory store operation stores the p parameter of an IFL calling sequence in the designated place in the memory subsystem or process space memory. In addition, an IFL can be executed in a call by value or call by name mode. In the call by value case, the values of $f_q(A_q, B_q)$ and K_q are stored in, respectively, new temporary storage locations p_1 and k_1,

while in the call by name case,* the values of the parameters p and k are, respectively, replaced by $f_q(A_q, B_q)$ and K_q. The call by name case is used when the p and k parameters are pointers either to fields in the memory subsystem or to p or k components in the process space memory. Thus, side effects in IFL can be programmed in two ways: directly through memory store operations or indirectly through the call by name mode.

The call by name mode is distinguished from the call by value mode through two control bits attached to the IFL instruction format previously discussed. There are also three other control bits, attached to each IFL instruction. These three other control bits are used to sequence the evaluations of the five fields in the IFL instruction and, thus, override the normal parallel evaluation.† These field sequence control bits allow the programmer to specify the order of evaluation of fields so as to avoid indeterminancy in IFL computations when one or more of field evaluations result in side effects. In addition the IFL, contains three other memory operations, PM, PP2, and PP3, whose execution results in the generation of a pointer to, respectively, a field in the memory subsystem or to a p or to a k component in the process space memory.

D. Pseudo-Functional Units

An IFL program can be used to simulate the actions of a functional unit in the functional unit subsystem. This use of an IFL program occurs when the value of the fu parameter of an instruction macro is greater than the number of functional units in the functional unit subsystem. In this case, the fu parameter is interpreted as the starting address in the program memory of an IFL program. The IFL program activated by the instruction macro then interacts with the input data and output data generators of the instruction macro through the following operations: FI, FIF, FIA, FIL, FO, FOF, FOA, FOL, FC, FS and FIN. The FI operation activates an input generator of the instruction macro and the bit

---

*The notation used in the assembler to distinguish a call by name from that of a call by value is the following: p←<PEXP> is a call by name while p:=<PEXP> is a call by value.

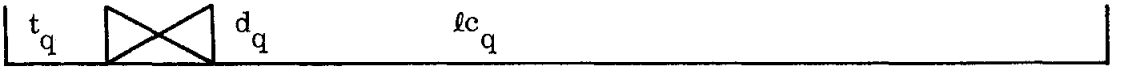
†The eight possible strategies for evaluation of the five fields have not yet been fixed.

string produced by the input generator is the output of the FI operation. The FIF, FIA, and FIL operations retrieve from the input generator, respectively, the format, address, and length of the next input data item to be generated. The FO, FOF, FOA, and FOL operations activate an output generator and are the output analogs of FI, FIF, FIA and FIL operations. The FC and FS operations retrieve, respectively, the c and s parameters of the instruction macro. The FIN operations is used to determine whether there is any more input data to be processed. These operations in conjunction with the other IFL operations previously discussed allow IFL programs to simulate an arbitrarily complex functional unit.

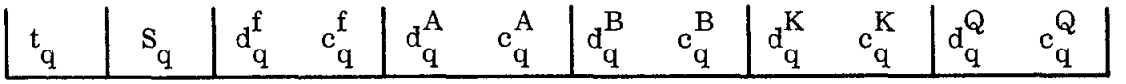
VI. FORMAT OF SBL MACROS

An SBL program, like an IFL program, is stored in the program memory. The format of a word in the program memory that defines an SBL macro body is identical to the format of a word that defines an IFL instruction. Further, the definition of a macro body, stored at address q, is specified in terms of the values of the fields f_q, A_q, B_q, K_q, and Q_q. These fields are computed for an SBL macro definition in the same manner as they are computed for an IFL instruction, where the two parameters p and k that are used in computing the values of the five fields are derived from the macro-calling sequence q(p,k). Thus, the only difference between the definition of an SBL macro body is the definitional semantics associated with the values of the five fields. The different types of macro bodies are specified in terms of a fixed set of definitional templates (skeletons). The values of the five fields are then used in the expansion of a template (to fill in the blank spaces in a skeleton) where the usage of the five fields is fixed according to the particular template being expanded. This method for specifying the body of a macro is simple and uniform but at the same time very powerful since the value of each of the five fields can be the result of an arbitrarily complex address arithmetic computation.

The possible formats of a word q in the program memory are the following:



or



where t_q specifies either the type of (template for) the macro body or the type of IFL instruction stored at word q, and s_q is control information used in the evaluation of the five fields. The type field, t_q, is 3 bits long and specifies either one of the six possible macro bodies or one of two possible IFL instruction types. The control field, s_q, is five bits long, and in the case of an IFL instruction s_q, specifies whether the instruction is called by name or by value and the order of evaluation of the five fields, while in the case of an SBL macro s_q specifies whether each of five fields will be evaluated at the time of either macro expansion or macro execution.

A. Data-Descriptor Macro

The data-descriptor macro, q^D, when expanded, results in the generation of a terminal node MEM(f, a, ℓ), where f specifies the format of a data item, a its address, and ℓ its length. The values of these three parameters are computed, based on the values of the five fields stored at location q, in the following manner:

$$a = f_q(A_q, B_q)$$
$$\ell = K_q$$
$$f = Q_q$$

These three parameter values are computed at either macro expansion or macro execution time depending upon the value of s_q. Base relative addressing can be programmed by setting f_q = '+', A_q the value of a base register, and B_q the address displacement relative to the base: $a=A_q+B_q$.

The following symbolic notation will be used to represent the data-descriptor macro body:

<DATA-DESCRIPTOR>:=<STATEMENT-LABEL>: ^{D} MEM(<F>, <A>, <L>)
<F> := <EXP-FIELD>
<A> := <PEXP>
<L> := <EXP-FIELD>

The underlining of a parameter of MEM indicates the parameter is computed at macro execution time rather than at the time of macro expansion. This convention of underlining will be used for all symbolic representation of SBL macro bodies.

Example 20

Consider the data-descriptor macro, OPFT, discussed in example 3 on page 21. The body of the macro OPFT can be specified in the following manner:

OPFT^D: MEM(1, [INDIRECT]*24, 24)
INDIRECT^F: if k=0 then p else p:=p*24;
p:=M(p+7, 16), k:=[I-BIT], go to INDIRECT;
I-BIT^F: M(p+6, 1);

The 24 bit length words of the emulated computer are laid out in consecutive bit locations starting at 0 in the memory subsystem. The macro

OPFT is invoked with a calling sequence whose first parameter, p, is the virtual address of the first word of the indirect chain. IFL program INDIRECT, invoked with the same calling sequence parameters as OPFT, computes the virtual address of the last element of the indirect chain. Thus, the execution of OPFT(p,k) results in the execution of the memory subsystem command MEM(f, a, ℓ) where

f = 1, specifying floating point format;

a = INDIRECT(p,k)*24, the absolute address of the last word of the indirect chain;

ℓ = 24, the length of data word.

B. Selection Macro

The selection macro, q^S, when expanded, results in the generation of a terminal node SEL(q_0, INC, c, $\bar{p}$, $\bar{k}$). This terminal node, when executed, generates and then executes the macro calling sequence $\bar{q}(\bar{p},\bar{k})$, where $\bar{q}$ is equal to q_0+INC, and c indicates the type of activation. The value of the five parameters of SEL are computed based on the value of the five fields stored at location q in the following manner.

$$q_0 = q + f_q$$

$$INC = A_q$$

$$c = B_q$$

$$\bar{p} = K_q$$

$$\bar{k} = Q_q$$

The following symbolic notation will be used to represent the selection macro body:

<SELECTION>:=<STATEMENT-LABEL> ^{S} : SEL (<QO>, <INC>, <C >, <$\bar{P}$>, <$\bar{K}$>)

<QO>:=<STATEMENT-LABEL>

<INC>:=<EXP-FIELD>

<C >:=<EXP-FIELD>

<$\bar{P}$>:=<EXP-FIELD>

<$\bar{K}$>:=<EXP-FIELD>

Example 21

Consider the selection macro, INSTDECODE, discussed in example 5 on page 26, and suppose that the computer to be emulated has a 24 bit length word where the first 2 bits of the word specify one of four possible instruction formats. The body of the macro INSTDECODE can be specified in the following manner:

INSTDECODE^{S}: SEL (INSTFORMAT, [DEC], 1, p, k);
DEC^{F}: p:=p*24; M(p,2);

where the macro INSTDECODE is invoked with a calling sequence whose first parameter, p, is the virtual address of the instruction to be emulated. The IFL program DEC, when executed, returns the value of the first two bits of the instruction word. This value is then used to choose one of four possible macros: INSTFORMAT, (INSTFORMAT+1), (INSTFORMAT+2) or (INSTFORMAT+3). This macro is then executed with the same parameters as used to call INSTDECODE:

$\bar{q}$ = INSTFORMAT + DEC(p,k)
$\bar{p}$ = p
$\bar{k}$ = k

C. Iteration Macro

The iteration macro, q^{IT}, when expanded, results in the generation of a terminal node SCP(M,V,c,p_0,k_0). The SCP node, when executed, sequentially generates and executes a list of macro calling sequences: $q_1(p_1,k_1)$, ... $q_n(p_n,k_n)$. This list is generated by invoking the IFL program, V, with the initial parameters p_0 and k_0; $V(p_0,k_0)$, executed like a co-routine as previously described on page , generates a sequence of pairs (p_1,k_1) (p_2,k_2) ... (p_n,k_n) $(p_{n+1},0)$. The first n pairs are used to define the parameters pairs in the list of macro calling sequence. The corresponding macro q_i associated with each pair (p_i,k_i) is computed in the following manner:

$$q_i = q + M(p_{i-1},k_{i-1})$$

If the s_q bit associated with M parameter is set to evaluation at the time of macro expansion rather than macro execution, then M is a constant and thus q_i=q+M is a constant. The c parameter of SCP defines whether a macro calling sequence $q_i(p_i,k_i)$ will be evaluated for a single cycle or to completion, and in the case that q_i is a constant whether the macro q_i will be reexpanded for each cycle of the interation macro, reexpanded only for each sequence of parameters $(p_1,k_1)...(p_n,k_n)$, or never reexpanded.

The following symbolic notation will be used to represent the iteration macro body:

<ITERATION>:=<STATEMENT-LABEL>:^{IT} SCP(<M>, <V>, < C >, <PO>, <KO>)

<M>:=<STATEMENT-LABEL>/<EXP-FIELD>

<V>:=<STATEMENT-LABEL>

< C >:=<EXP-FIELD>

<PO>:=<EXP-FIELD>

<KO>:=<EXP-FIELD>

The parameter M is a <STATEMENT-LABEL> when M is evaluated at macro-expansion time.

Example 22

Consider the iteration macro, FORLOOP^{IT}, described in example 6 on page . The body of the macro FORLOOP can be specified in the following manner:

FORLOOP^{IT}: SCP(STAT, [INDEX], 1, -24, [GET]);

INDEX^{F}: ITERATE (p← p+24, k← k-1);

GET^{F}: p:=(p+3)*24; p:=M(p+9, 15); p+1;

The macro is called with parameter pair (parlist, 1), where parlist is a virtual address of the parameter list (A, B, C, N). It is assumed that memory subsystem represents a 24 bit wide computer memory. Thus, in order to get absolute address of the parameter list, parlist must be multiplied by 24. Further, it is assumed that parameters (A, B, C, N) are stored in the last 15 bits of the 24 bit word. The IFL program GET retrieves the value of parameter N and increases its value by 1. The IFL program INDEX generates the sequence of pairs (0, N), (24, N-1) ... ((N-1)× 24, 1), where the first element of each represent the absolute bit offset from the base of the array of the elements A[I], B[I] and C[I]. Example 23 will define the instruction macro STAT.

D. Instruction and Hierarchical Macros

The instruction macro, q^{I}, when expanded results in the generation of a list of macro calling sequences $q_1(p_1,k_1)$ ... $q_n(p_n,k_n)$, and a terminal node ICP(fu, in, cf, s). The body of the instruction macro, q^{I}, is specified in terms of

two words q and q+1. The first word, q, specifies the parameters used to generate the list of macro calling sequence while the second word, q+1, specifies the parameters of the ICP node. The list of macro calling sequences is generated by the same process as used by the SCP node of an iteration macro to generate a list of calling sequences. Thus, the five fields of q have the same semantics as the five fields of an iteration macro body. The fields of q+1 correspond to parameters of the ICP node, e.g., fu=f_q, in=A_q, cf=B_q, s=K_q.

The symbolic notation that will be used to define the body of the instruction macro is the following:

<INSTRUCTION>:=<STATEMENT-LABEL>^{I} : SCP(<M>, <V>, < C >, <PO>, <KO>), ICP(<FU>, <IN>, <CF>, <S>);

<FU>:=<EXP-FIELD>

<IN> :=<EXP-FIELD>

<S> :=<EXP-FIELD>

Example 23

Consider the instruction macro, STAT^{I}, discussed in example 6 on page 27. The body of the macro STAT can be specified in the following manner:

STAT^{I}:SCP(ARRAY,[GEN], 1, [POINT], 4), ICP(*functional unit, 2,,);

BASE^{F}: p:=p2(2, 0);	Comments Extract virtual address of dope vector
p:=p+k-1;	Compute desired element of dope vector, e.g., A, B, or C
p:=p*24;	Convert virtual address to absolute address
p:=M(p+9, 15);	Get virtual address of beginning of array A, B, or C
p*24;	Convert virtual address of array to absolute address, and then return absolute address

GEN^{F}: ITERATE(k:=k-1)	Generates sequence of pairs (I,3), (I,2), (I,1)
POINT^{F}: PP2(0,0)	Generate pointer to p component of the STAT calling sequence which is I

The macro STAT is called with parameters ((i-1)*24, N-i+1) for i=1,N, where the first parameter is the absolute bit offset from the base of array. The first word of STAT generates the following list of macro calls: ARRAY(J,3), ARRAY(J,2), ARRAY(J,1) where J is a pointer to the first parameter of STAT: (i-1)*24. The data-descriptor macro, ARRAY^{D}, when expanded, computes, using the IFL program BASE, the absolute address of the base of the array A, B or C, depending upon its second parameter which is 1, 2 or 3. The ARRAY macro, when executed, computes the effective address of the element of array A[I], B[I], and C[I], by adding the base of the array computed at macro expansion time to the value pointed to by J.

The format of the hierarchical macro, q^{H}, is very similar to the format of the instruction macro. The difference between the formats is that the second word q+1 of hierarchical macro can be an arbitrary SBL macro whereas the second word of the instruction macro defines the parameters of the ICP node. The symbolic notation that will be used to define the body of the hierarchical macro is the following:

<HIERARCHICAL>:=<STATEMENT-LABEL>^{H}: SCP(<M>, <V>, < C>, <PO>, <KO>),
<SBL-MACRO>

<SBL-MACRO>:=<DATA-DESCRIPTOR>/<INSTRUCTION>/<ITERATION>/
<SELECTION>/<HIERARCHICAL>/<CONTROL>

E. Control Macro

The control macro, q^{C}, when expanded results in the generation of a CM clocking process which has four internal parameters n, ℓ, svt, and syn. These four parameters are specified in terms of two words q and q + 1. The five fields of the first word q have the following meaning: n = f_q, ℓ = A_q, syn = B_q, null = K_q, r = Q_q, and the fields of the second word are respectively the q, p, k, c, and d components of the svt

template. The parameter, null, indicates which one of components of the svt are template, are null. In addition, the syn parameter also indicates the new s component of svt template. If the null parameter indicates that the components q, p, k, c, and d are null then the q^c can be specified in terms of only one word q.

The symbolic notation that will be used to define the body of the control macro is the following:

<CONTROL> := <STATEMENT - LABEL>$^{\{c\}}$: CM(<N>, <L>, <SVT>, <SYN>)

<N> := <EXP - FIELD>

<L> := <EXP - FIELD>

<SYN> := SUSPEND/TERMINATE

<SVT> := (<Q>, <P>, <K>, <S>, <C>, <R>, <D>)

<Q> := <EXP - FIELD>/<NULL>

<P> := <EXP - FIELD>/<NULL>

<K> := <EXP - FIELD>/<NULL>

<L> := EXPAND/EXECUTE/TERMINATE/SUSPEND/<NULL>

<C> := <EXP - FIELD>/<NULL>

<R> := <EXP - FIELD>/<NULL>

<D> := <EXP - FIELD>/<NULL>

VII. SUMMARY COMMENT AND FUTURE RESEARCH

This paper is a preliminary investigation of the organization of a parallel micro-computer designed to emulate a wide variety of sequential and parallel computers. This micro-computer allows tailoring of the control structure of an emulator so that it directly emulates (mirrors) the control structure of the computer to be emulated. An emulated control structure is implemented through a tree type data structure which is dynamically generated and manipulated by six primitive (built-in) operators. This data structure for control is used as a syntactic framework within which particular implementations of control concepts, such as iteration, recursion, co-routines, parallelism, interrupts, etc., can be easily expressed. The major features of the control data structure and the primitive operators are: 1) once the fixed control and data linkages among processes have been defined, they need not be rebuilt on subsequent executions of the control structure; 2) micro-programs may be written so that they execute independently of the number of physical processors present and still take advantage of available processors; 3) control structures for I/O processes, data-accessing processes, and computational processes are expressed in a single uniform framework. This method of emulating control structures is in sharp contrast with the usual method of micro-programming control structures which handles control instructions in the same manner as other types of instructions, e.g., subroutines of micro-instructions, and provides a unifying method for efficient emulation of a wide variety of sequential and parallel computers.

Future research on this micro-computer organization will attempt to develop more rigorous arguments for the merits of this proposed method for emulating control structures. In particular, a simulator for this micro-computer organization and emulators for complex sequential and parallel IML's will be programmed. These emulators will then be run on the simulator to gather performance statistics. In addition, it is planned to develop a higher level language, which can be easily compiled into SBL and IFL statements, for representing control structures of machines.

There are two other research areas which will be investigated. The first research area involves the addition to the SBL of primitive operators (macros) which control access to nodes in the process space memory, fields in the memory subsystem, and functional units in the functional unit subsystem. Thus, it is

proposed to integrate the concept of protection (capabilities, access path, etc.) into the definition of the control structure of a process which is where the definition of protection naturally belongs. In the preliminary investigation of this idea, it appears that the concepts of protection discussed by Dennis and Van Horn,[17] Lampson,[18] etc. can be easily specified, with the addition of two or three primitives to SBL, in the framework of the proposed data structure for control. Thus, emulators for operating systems IML's will be more easily implemented, and it will be possible to protect a micro-code from interference by other micro-programs.

The second research area to be investigated involves applying the concept of control structure definition language to the organization of a computer rather than just a micro-computer. The investigation of this research area has been prompted by the work of Mitchell[19] on the organization of an interpreter for LC^2 language.

REFERENCES

1. Burroughs Corporation [1963]. The Operational Characteristics of the Processors for the Burroughs B5000. Burroughs Corporation, Detroit, Michigan.
2. Illiac-IV System Study Final Report [1966]. Burroughs Corporation, University of Illinois No. 09852-B.
3. Abrams, P. S. [1970]. An APL Machine. Report No. SLAC-114, Stanford Linear Accelerator Center, Stanford University, Stanford, California.
4. Melbourne, A. J. and Pugmire, J. M. [1965]. A Small Computer for the Direct Processing of FORTRAN Statements. The Computer Journal, Vol. 8 (April).
5. "System/360 Model 40,2040 processing unit." [1966]. IBM Field Engineering Diagrams Manual, Document No. 0223-2842.
6. Cook, R. W. and Flynn M. J. [1970]. System Design of a Dynamic Microprocessor. IEEE Transactions on Computers, Vol. C-19, No. 3.
7. Lesser, V. R. [1968]. A Multi-Level Computer Organization Designed to Separate Data-Accessing from the Computation. Tech. Rep. CS90, Computer Science Department, Stanford University.
8. Lass, S. [1968]. A Fourth Generation Computer Organization. AFIPS Conference Proceedings, Vol. 32.
9. Horning, J. J. and Randell, B. [1969]. Structuring Complex Processes. Report RC-2459, IBM Watson Research Center, Yorktown Heights, New York.
10. Fisher, D. A. [1970]. Control Structures for Programming Languages, Computer Science Department, Carnegie-Mellon University, Pittsburg, Pennsylvania, Ph.D. thesis.
11. Private communications with Burroughs Corporation on B8502 Organization. [1969].
12. Bingham, H. W. and Reigel, E. W. [1969]. Parallelism Exposure and Exploitation in Digital Computing Systems. Final technical report, Burroughs Corp, Paoli, Pa.
13. Dahl, O., and Yngaard, K. [1966]. SIMULA - an Algol-Based Simulation Language. Comm ACM 9.
14. Conway, M. E. [1963]. A Multiprocessor System Design. Proc. FJCC 24, 139-146.

15. Shaw, A. C. [1966]. Lecture Notes on a Course in Systems Programming, Technical Report No. 52, Computer Science Department, Stanford University, Stanford, California.

16. PDP-11 Reference Manual. [1969]. Digital Equipment Corporation.

17. Dennis, J. B. and van Horn, E. C. [1966]. Programming Semantics for Multiprogrammed Computation. Comm ACM 8,3.

18. Lampson, B. W. [1969]. Dynamic Protection Structures. AFIPS Conference Proceedings (FJCC 69).

19. Mitchell, J. [1970]. Lecture at Stanford University.

20. McKeeman, W. [1967]. Language Directed Computer Design. AIFIPS Conference Proceedings (FJCC67).