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

	Page
I. INTRODUCTION.	1
A. Traditional Micro-Computer Architecture	2
B. Variable Control Structure as the Basis of a Micro-Computer Architecture	4
II. MICRO-COMPUTER ARCHITECTURE	6
III. MICRO-PROCESSOR SUBSYSTEM.	8
IV. STRUCTURE BUILDING LANGUAGE (SBL).	13
A. Control Data Structure	14
B. Use of the Six SBL Macro Types	16
C. Format of SBL Macro Calling Sequence	17
D. Subsystem Command Macros	20
E. Structure Building Macros	25
1. Sequential Control Structures	25
2. Nonsequential Control Structures	30
3. Tree Structured Addressing.	34
4. Synchronization, and Control and Data Linkage Among Processes	35
V. INTEGER FUNCTION LANGUAGE (IFL)	42
A. Format and Sequencing of IFL Instructions	43
B. Built-In Arithmetic Operations	48
C. Side Effects in IFL.	50
D. Pseudo-Functional Units	51
VI. FORMAT OF SBL MACROS	53
A. Data-Descriptor Macro	54
B. Selection Macro	55
C. Iteration Macro	56
D. Instruction and Hierarchical Macros	57
E. Control Macro.	59
VII. SUMMARY COMMENT AND FUTURE RESEARCH.	61
REFERENCES.	63

LIST OF FIGURES

		Page
1.	Conceptual structure of an emulator	2
2.	Micro-Computer subsystems (modules)	5
3.	Micro-Processor subsystem's organization	12
4.	The control data structure for an emulator of a von Neumann computer organization with interrupt	32
5.	Fork-join instruction	33

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.

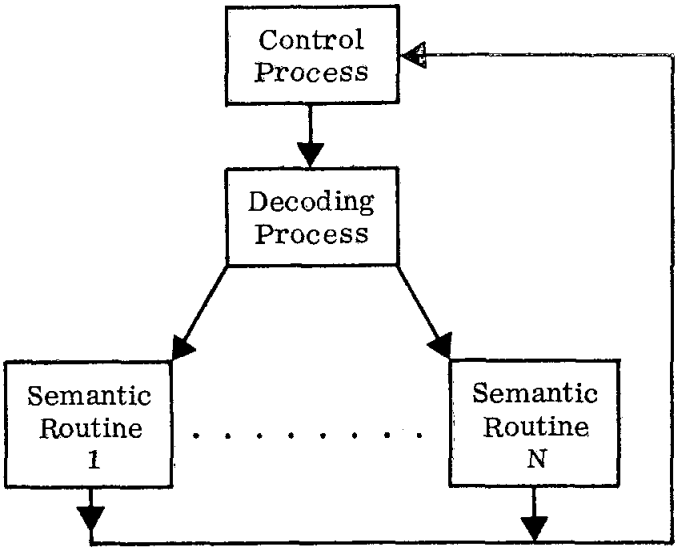


FIG. 1--Conceptual structure of an emulator.

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.).

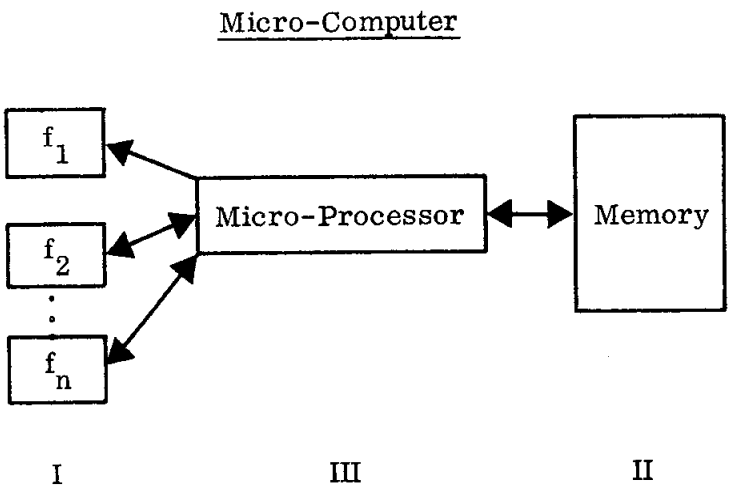


FIG. 2--Micro-Computer subsystems (modules).

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

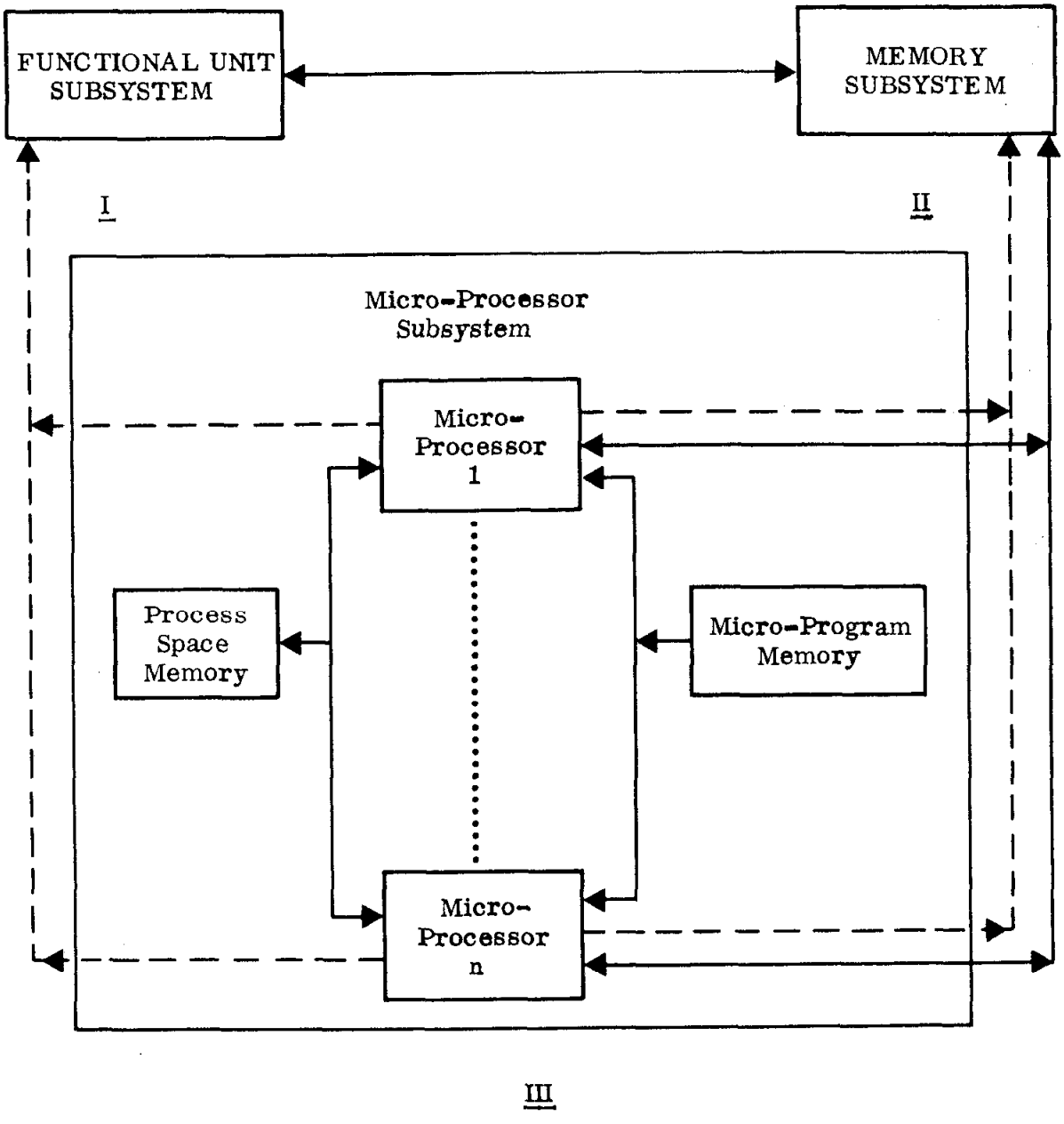
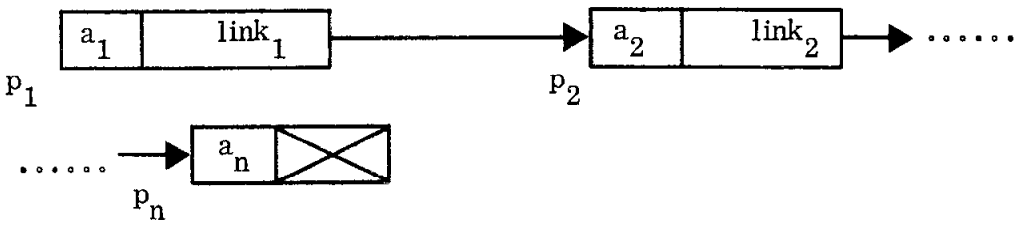


FIG. 3--Micro-Processor subsystem's organization.

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.

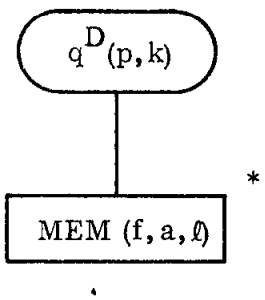
The following notation will be employed in the paper for specifying a macro name, a macro type, and a macro calling sequence. A macro name is specified in one of three following ways: (1) as a symbolic name which is optionally subscripted, e.g., M, a_i, a_{10} etc.; (2) as an absolute address in the 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:



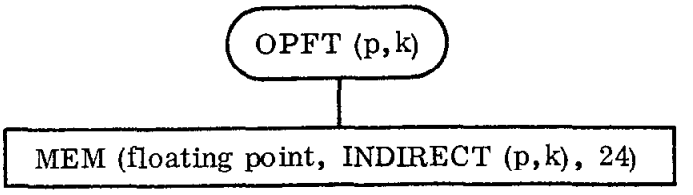
*a box will represent a terminal node

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:

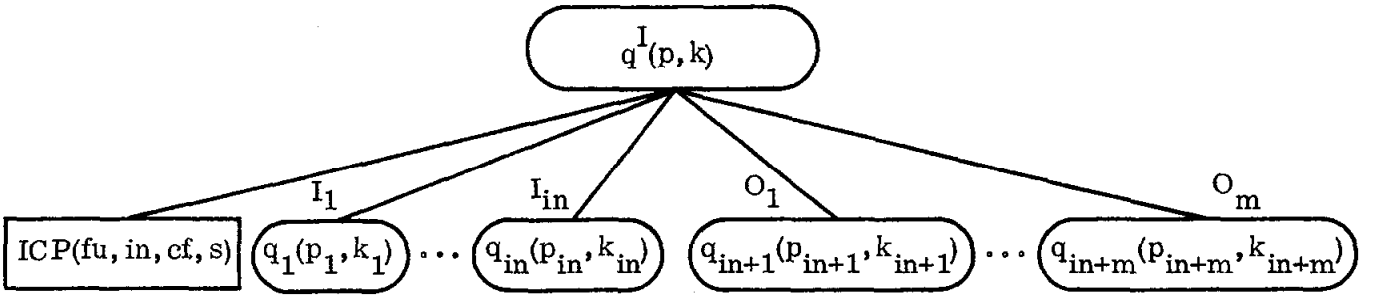


The IFL program INDIRECT is not invoked at macro expansion time but rather at macro execution time since the address field, a, of the memory subsystem command will be recalculated for each execution of the macro OPFT.

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:

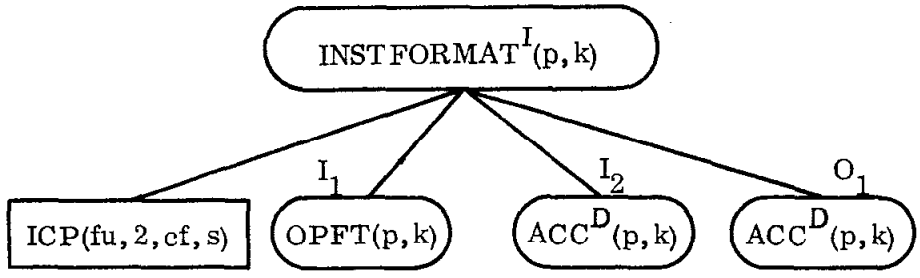


The clocking process ICP when executed, activates the functional unit fu with control information cf, and then waits for a request by the functional unit for input or output data. When input data is requested, the calling sequence $q_1(p_1,k_1)$ is activated to generate a single input value. Upon further 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

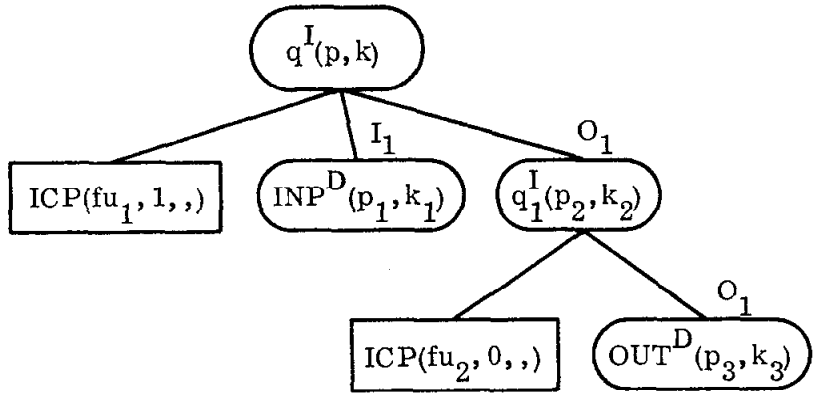
Consider the computer detailed in the previous example. An instruction macro $INSTFORMAT^I(p,k)$ which generates a functional unit subsystem command that emulates instructions of this computer can be defined in the following manner. Let the p parameter of the instruction macro be the virtual address 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,

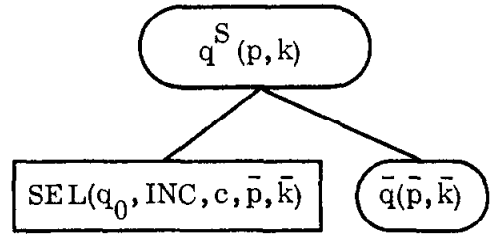
creating a two-level pipeline. Trees of functional units can also be created by this same mechanism; except in this case of a tree of functional units, the control structure is set up so that the instruction macro is requested to produce an input instead of storing an output. The output generated by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

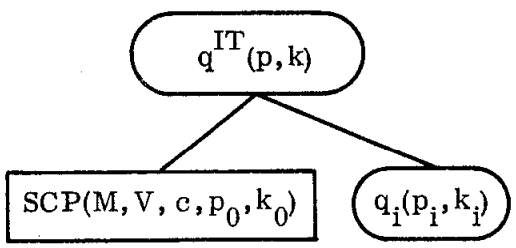
Example 5

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.

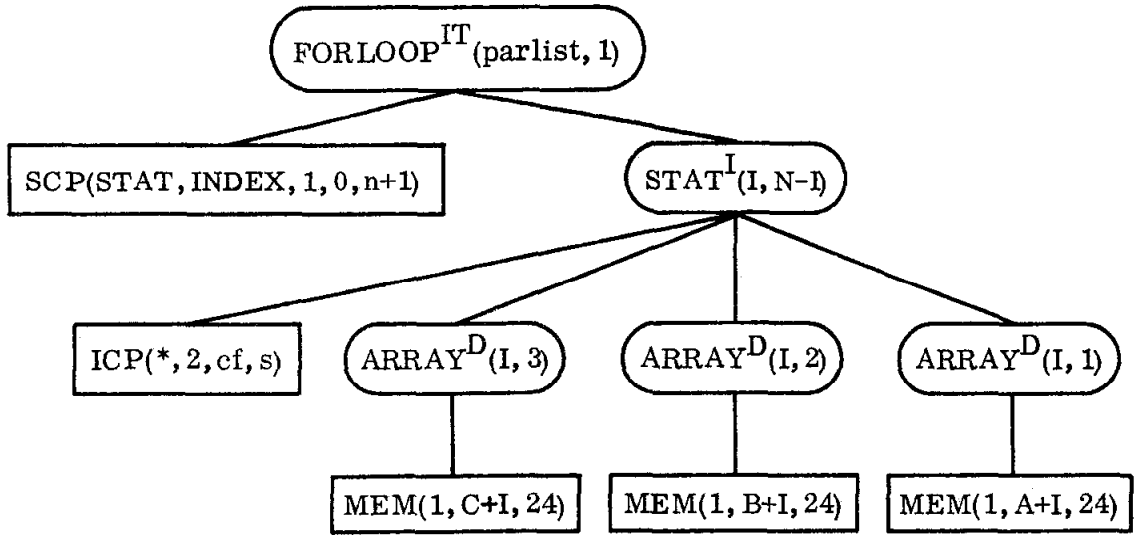
The clocking process SCP is activated with five internal parameters: the first two parameters, M and V, are the addresses of IFL programs; the third parameter, c, specifies control information; the remaining parameters p_0,k_0 are used to construct the initial calling sequence in the list. The M program 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.