AN INTRODUCTION TO THE DIRECT EMULATION OF CONTROL STRUCTURES BY A PARALLEL MICRO-COMPUTER

BY

VICTOR. R. LESSER

STAN-CS-71-191
January, 1971

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

AN INTRODUCTION TO THE DIRECT EMULATION OF CONTROL STRUCTURES BY A PARALLEL MICRO-COMPUTER[*]

Victor R. Lesser
Computer Science Department
Stanford University
Stanford, California

December 1970

The research was carried on while the author was an NSF graduate fellow and partially supported under AT(04-3)326, P.A.23.

[*] Presented as an informal paper at the 3rd Annual Workshop on Micro-Programming, Buffalo, New York. October 12-13, 1970.

Abstract

This paper is an 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 its control structure so that it is appropriate for the particular computer to be emulated. The control structure of this micro-computer is dynamically modified by changing the organization of its data structure for control. The micro-computer contains six primitive operators which dynamically manipulate and generate a tree type data structure for control. 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. An emulator programmed on this micro-computer works as an iterative two-step process similar to the process of dynamic compilation or run time macro-expansion. This dynamic compilation approach to emulation differs considerably from the conventional approach to emulation, and provides a unifying approach to the emulation of a wide variety of sequential and parallel computers.

Table of Contents

I. INTRODUCTION

In the past few years, there have been two trends in computer architecture that have significant implications for the architecture of micro-computers. The first trend is the development of computers whose machine languages are optimized for a particular higher level language. This trend is execmplified by the languages of machines such as the B6500[1] for ALGOL, Abram's APL Machine[2], Melbourne and Pugmire's Fortran Machine[3], etc. These machine languages represent a broader class of languages than is conventionally considered a machine language, e.g., a von Neumann machine language. We shall refer to this broader class as Intermediate Machine Languages (IML's). The tailoring of an IML to a specific higher level language is accomplished by incorporating instructions in the IML which "directly implement (i.e., mirror)" the primitive operations of the higher level language (e.g., a procedure call in ALGOL is directly mirrored, including the modification of the addressing environment, by the VALUE CALL instruction in the B6500). Thus, instead of implementing the semantics of higher level language primitive operations through an unnecessarily long and complicated sequence of instructions, the IML is designed so that there is a single or, at worst, a short sequence of IML instructions that efficiently carry out the operation. Therefore, by tailoring of a machine language more closely to a particular higher level language, the mapping between the higher level language and machine language is simpler and results in more compact and efficient generated code[4].

The second trend in computer architecture is the development of computers which are able to carry out parallel activity at the functional unit level, instruction level, or process level. This second trend is exemplified by machines such as the CDC6600[5], which permits asynchronous parallel operation of functional units, the ILLIAC-IV[6], which permits coordinated execution of multiple copies of a single instruction stream, and the dual processor CDC 6500, which permits the execution of multiple asynchronous instruction streams.

These two trends in computer architecture are not disparate but rather are separate aspects of a more general trend towards the design of complex problem oriented computers whose architecture departs considerably from a classical von Neumann architecture. Both the B6500 and the ILLIAC-IV represent an integration of these two trends in computer architecture; the B6500 is a multi-processor system which permits the allocation of multiple processors to the execution of a single ALGOL program, and the ILLIAC-IV has a parallel organization designed for problems involving an array structured data base.

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., the IBM 360/40[7], with read-only control memories containing micro-programs which emulate a specific von Neumann type computer. Recently, there has been an attempt to combine complex problem-oriented computer design with micro-computer design, implementing a specific architecture by modifying the read-write control memory of the micro-computer. It is hoped that the goal of emulating a wide range of problem-oriented computers can be realized by modifying dynamically the control memory of a single micro-computer. This goal cannot be attained on micro-computers whose architecture is essentially of the von Neumann type. This paper offers an architecture for a micro-computer with a control data structure and primitive operations that permits a systematic approach to the emulation of a wide variety of sequential and parallel intermediate machine languages.

A. Traditional Micro-Computer Architecture

Current micro-computers employ a simple sequential control structure and an instruction semantics for transferring data between registers. These features are quite adequate for emulating machine languages with simple control structures and instructions which operate on simple data structures. However, Intermediate Machine Languages (IML's) that are tailored for the execution of higher level languages are not this simple since the complexity of the higher level language operations is reflected in the semantics of the IML's instructions and control structure. If the current trend in the development of higher level languages is maintained, these problem-oriented IML's will employ increasingly more sophisticated control structures such as recursion, co-routines, parallelism, etc., and instructions for accessing complex data structures, such as lists, trees, arrays, etc. As will be argued below, these IML's call for a more sophisticated control structure in the micro-computer.

The control structure of a language, L , consists of a set of control rules, CR_L , and a data structure for control, CDS_L , commonly called state information, program environment, etc., on which the control rules operate. The control rules determine, at each meaningful unit of activity of the language, which statement or statements of the language will next be executed. For example, the CDS of a simplified computer could consist of a program counter and an interrupt, register; the CR of this simplified computer could be the following paradigm: "if there are no interrupts pending, then execute the instruction at the location specified by the program counter, otherwise, store the program counter at a fixed location in the program memory, turn off the interrupt bit, place the address of the interrupt handling routine in the program counter, and then execute the first instruction of the interrupt handling routine". This definition of a control structure makes a clear distinction between the control structure of a language and the execution of control statements of a language, e.g., conditional branch instructions, etc. The control statements of a language implicitly, rather than explicitly, affect sequencing by modifying only one part of the control structure, namely, the CDS; the actual sequencing of statements occurs only by the interpretation of the control data structure by the control rules. For example, consider the results of executing the control statement "branch to location X" in terms of the control structure of the simplified computer discussed previously; the branch statement, when executed, places the address X in the program counter; however, the next instruction to be executed may not be at address X since during the time the branch instruction was executed, an interrupt could have occurred.

By examining the program structure of an Intermediate Machine Language emulator, we can see the shortcomings of the typical current micro-computer architecture. The program structure is shown conceptually in Figure 1. The "controlprocess" activates the "decoding process" with data that identifies the next instruction of the emulated computer to be executed; the "decoding process" then analyzes the instruction to be executed so as to determine the semantic routine, together with its appropriate calling sequence, whose activation will perform the semantics of the emulated instruction. After the appropriate semantic routine has 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, selects the next instruction to be emulated. This basic cycle is conventionally called the "Do Interpretive Loop" (DIL)[8].

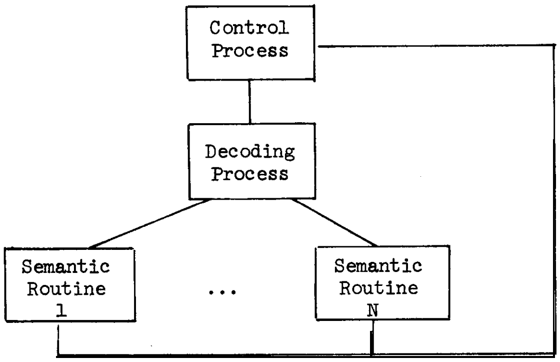


Figure 1. Conceptual Program Structure of an Emulator

In micro-computers designed to emulate a specific computer architecture or family of computers with similar architectures, the control and decoding processes of emulators of these architectures are usually implemented in the hardware of the micro-computer, and the semantic routines of the emulators are implemented as micro-programs. When such a micro-computer is used to emulate an IML which was unanticipated by the designer of the micro-computer, the hardware implementation of the control and decoding processes cannot be used. The micro-language designed as a means of coding the semantic routines must then also be used to code the control process and the decoding process. However, there are no features in the micro-language designed to facilitate the coding of control or decoding processes, nor can the instruction sequencing of the micro-computer be used directly to control the flow of activity among the control process, decoding process and semantic routines of the emulator. This lack of features in the micro-language for general purpose emulation is an especially serious shortcoming when an IML with complex sequential or parallel instruction sequencing is to be emulated. Additionally, existing micro-computer architectures do not contain multiple micro-processors, and thus parallel activity indicated by the IML can only be emulated by sequentializing the parallel activity.

This paper presents a way to emulate sophisticated IML's by (1) defining a micro-computer with a powerful and general control structure, and (2) supplying features (totally missing in current micro-computers) to tailor the control structure of the micro-computer so that it corresponds to the one needed for the particular IML being emulated. This tailoring results in the control structure, instruction semantics, and primitive data-accessing operations of an IML being more directly implemented (i.e., mirrored) in the corresponding control structure, instruction semantics, and primitive data-accessing operations of the micro-computer on which the IML emulator is executed. For example, if the semantics of an IML instruction require a particular form of iteration, then that form of iteration will appear in the tailored control structure and will not have to be programmed as a sequence of micro-instructions. An additional by-product of tailoring is that the structure of the IML is directly observable in the program structure of its emulator.

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

The control structure of this micro-computer is dynamically modified by changing the organization (form) of its data structure for control. These modifications to the CDS are specified by a control structure definitional language called the Structure Building Language (SBL). The SBL can manipulate and build up the organization of the CDS only in ways understandable to the control rules of the micro-processor; the CDS, in a very general sense, can be considered a control structure definition program which, when interpreted by the CR of the micro-computer, defines a particular sequential or parallel control structure for sequencing of micro-instructions. Current micro-computers consider a micro-program as a linear sequence of instructions with no explicit internal structure (topology). The CDS can be thought of as a variable structure template which defines a particular internal structure for the micro-program memory. An SBL program is quite different from a sequence of control statements since the control structure definition program constructed in the CDS by the SBL program is separated from (external to) the micro-program. This separation of the control structure definition program from the micro-program permits the static part of the internal structure of a micro-program to be generated only once for repeated executions of the micro-program.

An integral part of the technique used by the SBL to define a control structure is to explicitly represent in the CDS the relationships between the execution of micro-instructions and the data environment in which these instructions operate. In particular, the sequencing of micro-instructions involves specifying the address of the micro-instruction to be executed and local and global parameters which define the data context for the execution of that micro-instruction. Thus, a close relationship between control and data environment allows the state of the emulated computer to be integrated directly into the CDS. Thereafter, SBL statements can be used to dynamically modify the CDS to reflect the state transitions occurring in the emulated computer. Additionally, the CDS can be used to define (1) control structures for sequencing micro-instructions which then carry out the semantics of the emulated instruction, (2) control structures for I/O, and (3) control structures for data-accessing operations.

II. MICRO-COMPUTER ARCHITECTURE

The micro-computer architecture pictured in Figure 2 can be characterized in terms of three basic hardware subsystems. The first subsystem is 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, and may be data dependent. For example, a functional unit could be a floating point multiplier, or, more generally, an arbitrary input/output device. A functional unit can receive input data from three sources: the memory subsystem, another functional unit, or the micro-processor subsystem. A functional unit obtains (and stores) data by requests to the micro-processor subsystem, which has complete responsibility for determining the source (or sink) of the data which is requested and for generating the appropriate control signals to accomplish the data transfer. In this manner, the micro-processor subsystem acts as a generalized I/O controller and separates the process of data-accessing from that of computation.

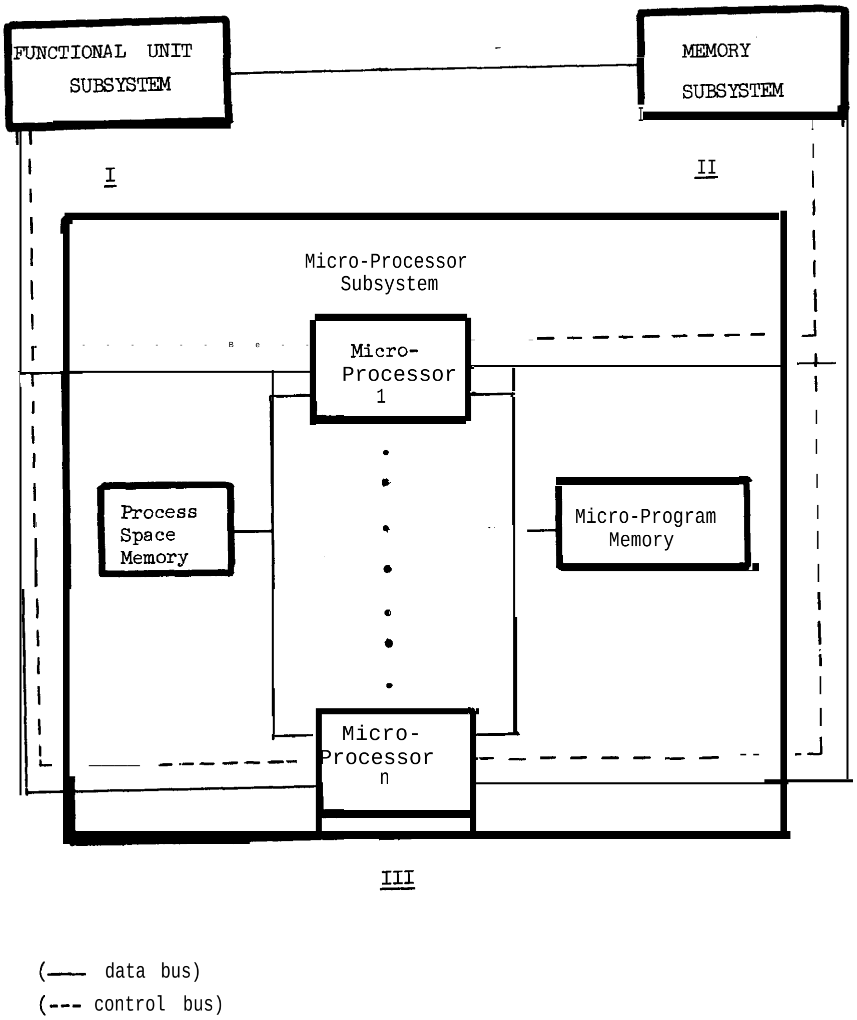


FIGURE 2. Micro-Computer Hardware Organization

The second subsystem is a memory. This memory is bit-addressable and can be activated to store or retrieve an arbitrary length string of bits. This memory holds the program to be emulated, and can serve 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 functional units. The memory subsystem is itself not just another functional unit because the process of field (bit-string) extraction and manipulation is recognized as an integral part of general purpose emulation; the language of the micro-processor contains instructions that directly retrieve and store fields of the emulated computer's program memory. Thus, there exists a direct connection between the micro-processors and the memory subsystem because accessing the memory subsystem is a primitive operation of the micro-processors.

The third subsystem is composed of an arbitrary number of identical micro-processors and two additional memories. The micro-processor subsystem controls the dynamic interactions among functional units, between the functional unit subsystem and the memory subsystem, and among micro-processors. The execution of the micro-processors is controlled through data stored in the micro-program and process-space memories; these two memories contain, respectively, the static and active parts of the control structure of the micro-processor subsystem. The micro-program memory holds micro-programs and is not normally modified during an emulation; the micro-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 frequently modified during emulation. The process space memory stores the state of the emulator, the state of the emulated computer, and the state of micro-processor subsystem. The process space memory, in addition, contains a set of auxilliary working registers which are dynamically allocated to micro-processors for temporary storage.

The micro-processors can be executing concurrently. The process space memory stores the control data structure which coordinates the activity among virtual micro-processors. If there are not enough micro-processors to carry out the parallel activity specified by the CDS, the available micro-processors are scheduled on a first-come first-served basis. The state of the micro-processor subsystem maintains this mapping from virtual activity to actual activity. This transformation from virtual processor activity to actual micro-processor activity may lead to indeterminate results depending upon the number of micro-processors available. However, as will be described in Section IV.C, the SBL contains control primitives that allow the micro-programmer to construct appropriate synchronization rules (e.g., Dijkstra's semaphore, Saltzer's wakeup-waiting switch, lock-step execution, etc.) which preserve the inherent parallelism among processes, while at the same time guaranteeing that the scheduling of virtual parallel activity will always result in determinate computation, independent of the number of actual micro-processors.

The ability to manipulate the control data structure allows the tailoring of both the hardware and software of this architecture to various IML's. 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, matrix multiply unit, etc.) or the addition of micro-processors. These hardware modifications can be made without modifying the language of the micro-processor. The software tailoring involves building up an appropriate control data structure in process space memory which integrates the state of the emulated computer with the state of its emulator. Thereafter, changes in the state of the emulated computer can be directly reflected in changes in the state of its emulator.

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 instruction and generate the appropriate sequence of functional unit activations or calls to micro-programs which will perform the computations specified by the instruction. In addition, the sequence of functional unit activations must be coupled with accesses from and stores to the memory subsystem to provide the input and output data set for each unit. This sequence of functional activations may result either in concurrent operation of functional units or a pipelining of data through a series of functional units.

III. MICRO-PROCESSING SUBSYSTEM

The main orientation in the design of this micro-computer is the incorporation of a facility for defining a variable control structure in the hardware of its processor. This design emphasis has led to a micro-processor that contains two basic classes of instructions. One class, called the Integer Function Language (IFL), is used to program address arithmetic functions while the other, called the Structure Building Language (SBL), is used to construct dynamically the control structure of the micro-processor subsystem.

The Integer Function Language is a highly specialized micro-code language specifically designed for the task of address arithmetic involved in computing the effective address of an operand, decoding an instruction, etc. The IFL is also used to construct programs which simulate the actions of functional units, e.g., a floating point multiplier, etc. The IFL departs from conventional micro-languages in three ways: (1) the result of executing an IFL program is to return an integer value to the calling routine; (2) each IFL program has parameters passed to it; and (3) IFL instructions do not operate on a fixed set of registers, but rather on the parameters which are passed to an IFL program. Each IFL program can be thought of as a definition of a function which, when executed, may have side effects.

The SBL, on the other hand, is used to define the control structure of a process. A process consists of the execution of a sequence of statements which either activate an IFL program or activate a functional unit. The control structure for sequencing these process statements specifies not only the sequence of activations, but also the appropriate input and output data sets for each activation. The generation of these input and output data sets may itself be based on the results of activations of IFL programs or the activations of functional units. The SBL sequences IFL programs both implicitly and explicitly as will be seen more clearly in the next section. The explicit sequencing results from the control structure constructed by the SBL; the implicit sequencing occurs when an IFL program is invoked during the execution of an SBL statement. The SBL statements, in themselves, have no ability to do arithmetic computations other than to add to or subtract from a parameter a constant. Thus, whenever a more sophisticated parameter calculation is required by an SBL statement in order to modify or build up a Control Data Structure, an IFL routine is implicitly invoked by the SBL statement so as to perform the desired parameter calculation.

The SBL is not designed to define the sequencing of individual IFL instructions since the control structure necessary for address arithmetic algorithms is simpler and less variable than that required for processes; the control structure for IFL instructions is build-in rather than dynamically constructed. Address arithmetic functions can therefore be executed without invoking the overhead of a variable control structure.

The rest of this paper will discuss the techniques employed by the SBL to generate a variable control structure. The IFL will not be discussed further in this paper. However, an example of an IFL program is contained in Appendix A, and a complete discussion of the IFL is contained in a previous paper by the author[10].

IV. STRUCTURE: BUILDING LANGUAGE

The basis of the syntax and semantics of the SBL is a fixed set of control structure definitional prototypes (templates) that, when expanded, modify the CDS so as to define a particular type of control structure. An SBL statement (macro) specifies one of the fixed set of prototypes together with a set of IFL address arithmetic functions. Each prototype 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 prototype defines a particular instance of a basic control concept. 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 the set of control structure definition prototypes. This dynamic mapping is represented in the address arithmetic algorithms specified in the SBL macro that are used to calculate the parameters of the prototype. 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 dynamic compilation approach to emulation differs considerably from the conventional approach to emulation (i.e., calling subroutines of micro-instructions[9]) done on existing micro-processors, and directly reflects (as will be seen in the next paragraphs) the conceptualization of the structure of an emulator pictured in Figure 1.

The CDS is in the form of a tree whose non-terminal nodes are calling sequences to SBL macros or to IFL programs, and whose terminal nodes are calling sequences to hardwired control rules (clocking processes[11]) which sequence the non-terminal nodes, control the activity of functional units, or control the accessing of data from the memory subsystem. Terminal nodes can be considered as instructions for a control structure definition programwhich, when interpreted by the control rules of the micro-processor, defines a particular control structure for the micro-processor subsystem. An example of a CDS is pictured in Figure 3.

The execution of an SBL macro is factored into three separable phases: a binding, an expansion, and an activation phase. The binding phase occurs when a macro calling sequence is generated in a non-terminal node in the CDS by the expansion or activation phase of another SBL macro; in particular, the non-terminal node contains the address of an SBL macro stored in the micro-program memory, and the parameters to be associated with expansion phase of the macro. For example, the binding phase analog in an ALGOL procedure is the machine code that sets up the parameters for the procedure call.

> Example 1A:*/ Consider the emulation of an instruction, FAD I 20, where FAD specifies a floating add operation, I specifies indirect addressing, and the accumulator is the second and result operand. The first step in the emulation of this instruction on this micro-processor is the following: An SBL macro generates a non-terminal node in the CDS which is a binding between a parameter, IR (instruction register), whose value is the instruction to be emulated, and an SBL macro IEXEC. This binding phase step is the analog of the control process of an emulator, and is pictured in Figure 4A.

The expansion phase, which is the second phase, occurs when SBL macro calling sequence stored in the non-terminal nodes is executed, and results in the generation of sets of son nodes of the non-terminal node. The number of son nodes generated and the particular calling sequence stored in each son node is defined by the

*/ Examples IA, 1B, 1C, 2, 3, and 4 form an integrated sequence which explains Figure 3.

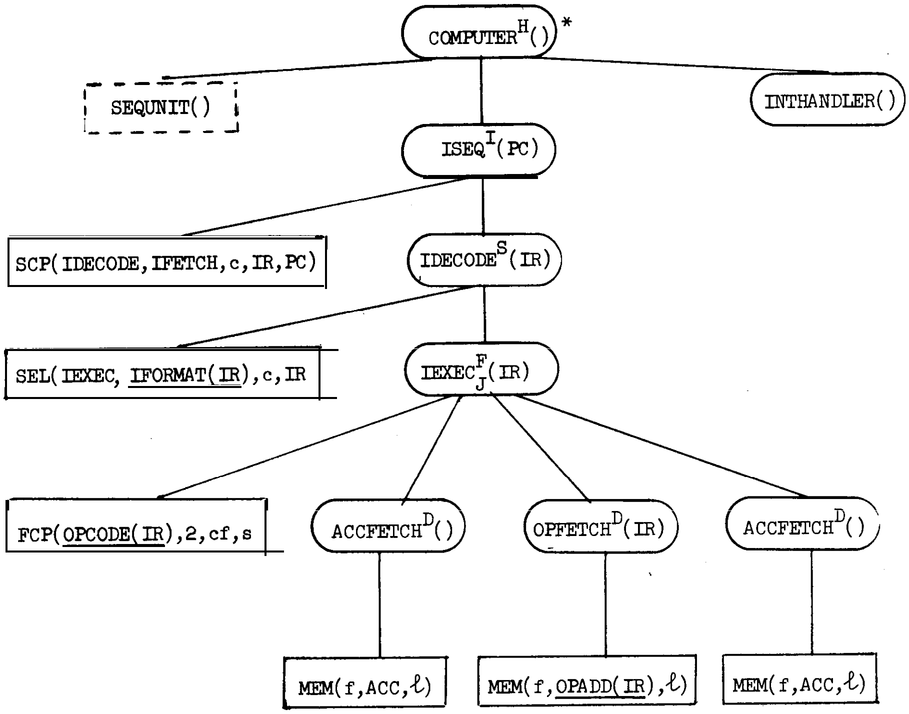


FIGURE 3. The Control Data Structure for an Emulator of a Conventional Computer

*An oblong box is a non-terminal node; a rectangle is a terminal node; an underline indicates an IFL calling sequence; and a superscript on a macro calling sequence, e.g., the D on OPFETCH^D, indicates the control structure prototype that the macro invoked.

parsmeterized prototype specified by the called SBL macro stored in the micro-program memory. The prototype parameters are either specified as immediate data or computed by either adding immediate data to one of the SBL calling sequence parameters, or invoking an IFL program whose parameters are the SBL calling sequence parameters. The set of son nodes generated by the parsmeterized prototype consists of a terminal node (i.e., calling sequence to a hardwire control rule) and a possibly empty set of non-terminal brother nodes which define the immediate environment on which the hardwired control rule operates. For example, the expansion phase analog in an ALGOL procedure is the execution of machine code that performs procedure initiation when a procedure is called, e.g., allocation of memory for a run-time stack, transferring the procedure parsmeters to the run-time stack, etc.

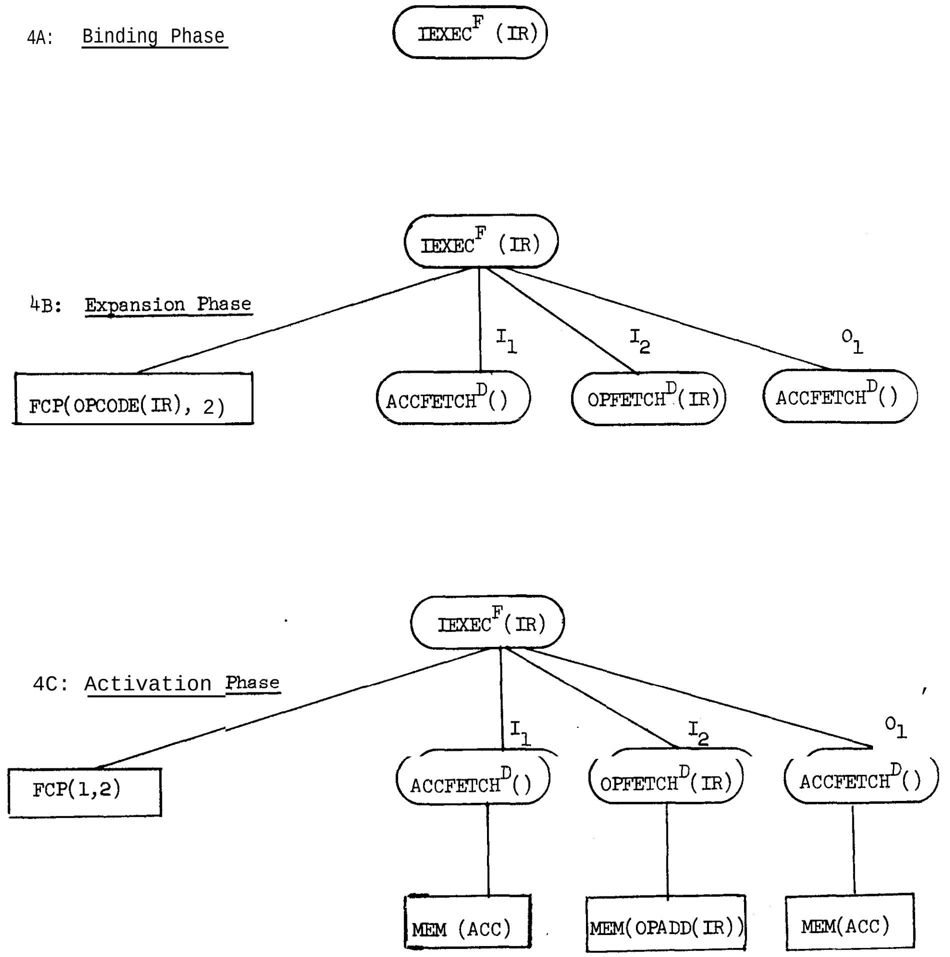


FIGURE 4. The Three Phases of SBL Macro Execution

Example 1B: The second step in the emulation of the instruction FAD I 20 is the expansion of the macro calling sequence IEXEC(IR). The SBL macro, IEXEC, specifies a control structure definitional prototype which generates a CDS that controls a functional unit. The parameters of this prototype specify the functional unit to be activated, and the set of non-terminal nodes that will be used to generate the input and output data set of the functional unit. These prototype parameters, with the exception of the name of the functional unit, do not vary from one execution to another execution of the macro IEXEC; thus, these non-varying parameters are specified in the body of macro IEXEC as immediate data. Whereas, the varying prototype parameter is computed by an IFL program called OPCODE; this IFL program, called with parameter IR, extracts the op-code of the instruction and then determines the name of the appropriate functional unit. The expansion of the SBL macro calling sequence, IEXEC(IR), results in a CDS pictured in Figure 4B; the terminal node in Figure 4B contains a calling sequence to an FCP control rule where the first parameter specifies the functional unit, e.g., floating pointer adder, to be activated and the second parameter specifies the number of non-terminal input nodes, e.g., the nodes I1 and I2. This expansion phase step is the analog of the decoding process of an emulator.

The activation phase, which is the third phase, results in the execution of a hardwired control rule whose calling sequence is stored in the terminal node generated by the expansion phase. The execution of the hardwired control rule, in turn, results in further execution of SBL statements, the activation of IFL programs, or the activation of functional units. For example, the activation phase analog in an ALGOL procedure is the execution of the machine code for the procedure.

> Example 1C: The third step in the emulation of the instruction FAD I 20 is the execution of the calling sequence FCP(1,2) stored in the terminal node. The hardwired control rule FCP activates functional unit 1 with two inputs, and then stores the output; the two inputs are generated by executing nodes I_1 and I_2 , and the output stored by executing node O_1 ; the expansion of these nodes results in Figure 4C where the hardwire control rule MEM controls access to the memory subsystem. In this case of node I_2 , the expansion of the data-descriptor macro calling sequence, OPFETCH(IR), requires the implicit execution of an IFL program, OPADD; this IFL program does the address arithmetic, in this case indirect addressing, required to locate the operand of the instruction. The other two nodes . I_1 and O_1 generate a fixed data-descriptor which specifies the area in the memory subsystem set aside as the accumulator. This activation phase step is the analog of the semantic routine of an emulator.

The examples IA, 1B, and 1C indicate 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. Additionally, if there does not exist a functional unit to carry out the semantics of the emulated instruction, then these semantics can be programmed in terms of the IFL; this IFL program, called a pseudo-functional unit, is then activated by the FCP control rule in the place of a functional unit.

A. SBL Macro Prototypes*/

There are six possible SBL macro prototypes: data-descriptor (D) , function (F) , selection (S), iteration (I) , hierarchical (H) , and activation (C) . An SBL macro whose prototype is either a data-descriptor prototype or a function prototype is called a subsystem command macro, while a macro whose prototype is one of the remaining four is called a structure building macro. The subsystem command macros generate a CDS that controls the interaction between the memory subsystem and the functional unit subsystem. The CDS of a more complex process is constructed through the execution of a seqeunce of structure building macros that use as their basic building block calling sequences to subsystem command macros. If the basic building blocks are just data-descriptor macro calling sequences, then the structure building macros define the control structure of a data-accessing procedure. The SBL macros can also be used to define an I/O control structure which, for example, duplicates the effect of an I/O channel. An I/O control structure can be considered the definition of a macro-instruction when the functional unit being controlled in an arithmetic device. This later use of SBL macros was shown 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[11], and by Lass[12] as a basis of the design of a high speed computer.

*/ This section discusses the semantics and use of SBL macro prototype but does not discuss the internal – machine language format for SBL *macros.* A discussion of internal machine language format of the SBL macro and also IFL instructions is contained in a previous paper by the author.[10]

The control data structures generated by expanding each of the six macro prototypes are pictured in Figure 5. The non-terminal nodes pictured in Figure 5 contain SBL macro calls which have a fixed format consisting of an address q and a parameter p , e.g., q(p) . The address, q , specifies the location of an SBL macro in the micro-program memory, and the parameter p is used in the computation of the parameters of the prototype. The terminal nodes contain calling sequences to one of five hardwired control rules: MEM (Memory Subsystem Command), FCP (Function Clocking Process), SEL (Selection Clocking Process), SCP (Sequential Clocking process), and ASP (Activation and Synchronization Clocking Process). In order to

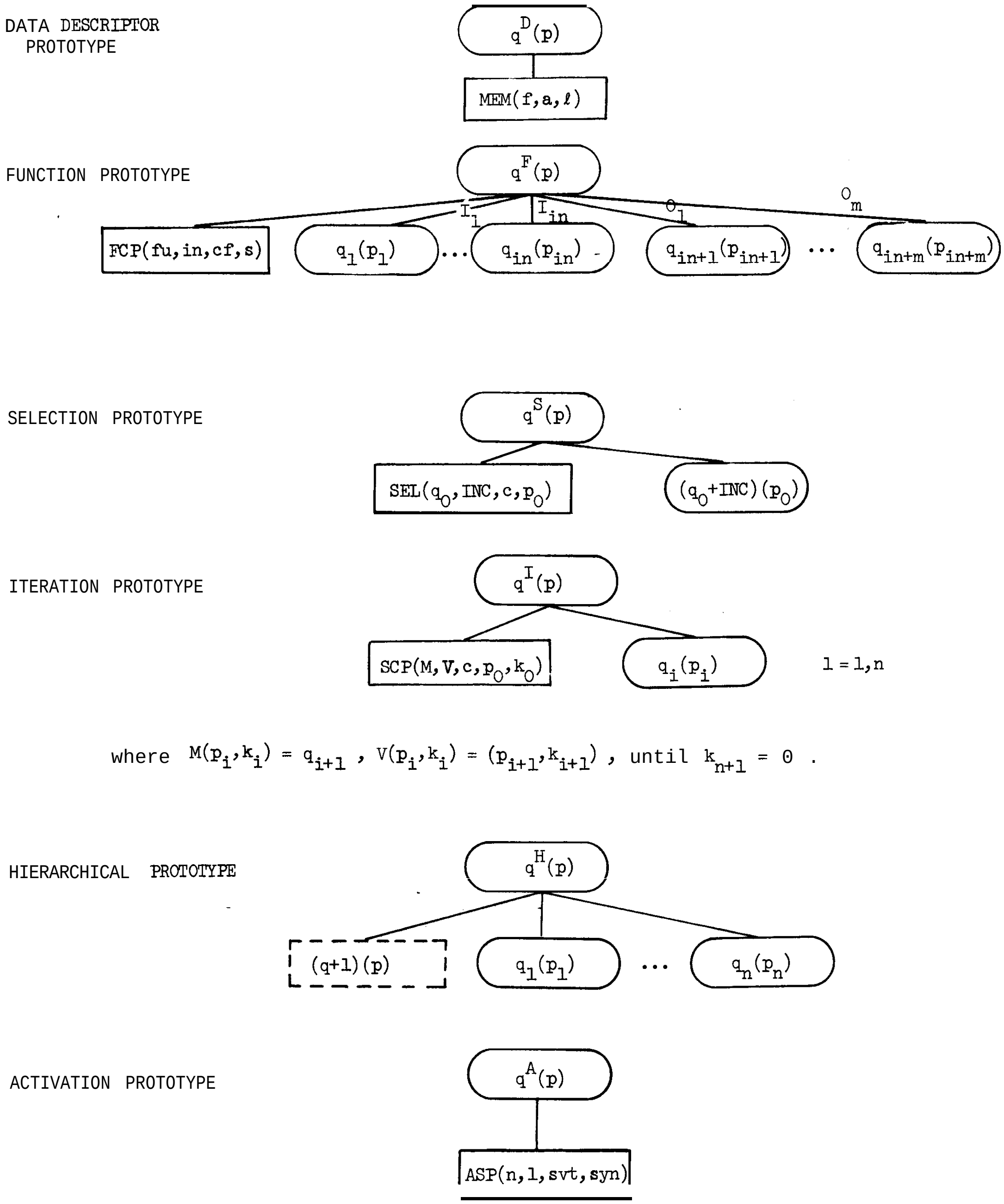


FIGURE 5. CDS Resulting from Macro Prototype Expansion

distinguish for future discussions the parsmeters of SBL macro calling sequences from those of the hardwired control rule calling sequences, the SBL macro calling sequence parameters will be called external parameters and the hardwired control rules calling sequence parameters will be called internal parameters. A more detailed discussion of the format and use of the external and internal parameters stored in the CDS will be delayed to Section IV.B.

The MEM control rule defines an access path to the memory subsystem; the MEM control rule is activated with three internal parameters: $\underline{f}$, the format of the data-item, e.g., floating-point format, etc.; $\underline{a}$, the beginning bit address of the data-item in the memory subsystem; and, $\underline{\ell}$, the bit length of the data-item. When executed, the MEM control rule activates the memory subsystem to fetch (or store into) a bit string bounded by addresses a and $(a+\ell-1)$, and then sends this data together with the format field, f , to the requesting functional unit or IFL program. The MEM control rule calling sequence neither specifies the particular functional unit or IFL program 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 FCP control rule which 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.

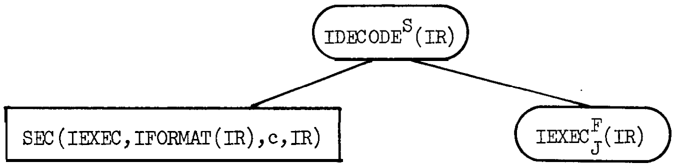
The FCP control rule activates a functional unit and then controls the generation of the input and output data set of the functional unit; the FCP control rule is activated with four internal parameters: $\underline{fu}$, the name of a functional unit or IFL program; $\underline{in}$, the number of input set generator nodes (the number of output set generators are the remaining brother nodes); $\underline{cf}$, control information sent to the functional unit upon its activation; and $\underline{s}$, an address in the memory subsystem where the status of the functional unit at the termination of its operation is stored. When executed, the FCP control rule activates the functional unit fu with control information, cf , and then waits for a request by the functional unit for input or output data. If input data is requested, then the calling sequence $q_1(p_1)$ is activated to generate a single input value. Upon further requests for input $q_1(p_1)$ is executed again until it produces no more data (e.g., it is terminated) and then $q_2(p_2)$ is activated. The same process is then repeated with $q_2(p_2)$. If an output is requested, $q_{in+1}(p_{in+1})$ 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. The function macro prototype generates a CDS that clearly divorces data-accessing from the computational algorithm. This separation facilitates the definition of control structures which (1) directly emulate different types of IML instruction formats, e.g., one address, two address, etc.; (2) specify interconnection patterns among functional units,*/ e.g., a pipeline of functional units, a tree of functional

*/ The method of generating a CDS for these alternative functional unit control structures is discussed more fully in a previous paper by the author[10].

units, etc.; and (3) allow the incorporation of functional units into the functional unit.subsystem that have complex input and output requirements, e.g., a matrix multiply unit.

The SEL control rule proveides a mechanism for the conditional expansion of the CDS, and serves a purpose in the SBL analogous to the CASE statement in ALGOL, or the Computed Go To statement in FORTRAN. The SEL control rule is activated with four internal parameters: q_0 , an address in the micro-program memory of the base of an array of SBL macros; INC, an index into the array of SBL macros; c , control information which specifies the mode of execution of the generated macro calling sequence; and p_0 , the external parameter of the generated macro calling sequence. When executed, the selection clocking process (SCP) stores in its brother node the SBL macro calling sequence $(q_0+\text{INC})(p_0)$, and then expands and activates this calling sequence based on the control information c . The possible modes of executing a macro-calling sequence will be discussed in the next section.

Example 2: Consider a machine language which has several instruction formats. The emulation of instructions of this machine language could be programmed, e.g., have a separate function macro, IEXEC_J^F , for each format, J . A selection macro IDECODE^S , could be used to implement a decoding process that selects the appropriate function macro given a parameter, IR, whose value is the emulated instruction:



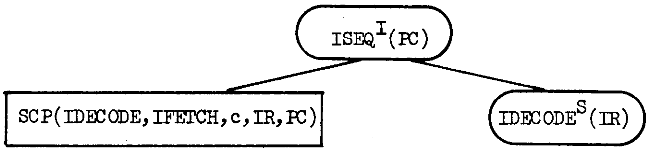
where IFORMAT is an IFL program that selects the appropriate index into the array of function macros based on the format, J , of the emulated instruction.

The SCP control rule is used to define the control structure of a sequential process, and serves a purpose in the SBL analogous to the FOR-LOOP in ALGOL, the DO-LOOP in FORTRAN, or the MAPCAR function in LISP. When executed, the SCP control rule sequentially generates and executes as its brother nodes a list of macro calling sequences:

$$q_1(p_1),\ldots,q_i(p_i) \; , \; q_{i+1}(p_{i+1}),\ldots,q_n(p_n) \; .$$

The sequential clocking process (SCP) defines only a sequential control structure since each macro calling sequence $q_i(p_i)$ is completely executed before the next calling sequence $q_{i+1}(p_{i+1})$ is generated. The SCP control rule is activated with five internal parameters: the first two parameters, $\underline{M}$ and $\underline{V}$, are the addresses of IFL programs; the third parameter, $\underline{c}$, specifies the mode of executing a list of macro calling sequences; and the remaining parameters, p_0 and k_0 , are used to construct the initially macro calling sequence in the list; the M program called with parameters (p_i,k_i) computes q_{i+1} , the address of a macro. "q_i- is constant, then M can be the address of q_i rather than the address of an IFL program that computes q_i . The V program, also called with parameters (Pi'ki) computes (p_{i+1},k_{i+1}) . The generation of calling sequences continues until $k_{n+1} = 0$.

Example 3: The iteration macro can be used to construct a control structure that implements the instruction fetch cycle of an emulated computer. Consider the iteration macro calling sequence, $ISEQ^1(PC)$, which generates an SCP control rule with following internal parameters: $M = IDECODE^S$ (in this case M is an address of SBL macro rather than an IFL program), V = IFETCH , p_0 = IR , and k_0 = PC where the IR parsmeter contains the emulated instruction just executed, and the PC parameter the address of the next instruction to be emulated. The IFL program IFETCH(IR,PC) extracts the instruction of the emulated computer specified by PC, stores this instruction as the new value of IR, and updates PC by one. The SCP control rule, with the above parameters, generates a sequence of calls to the selection macro $IDECODE^S$ with a parameter whose value is the next instruction to be emulated:

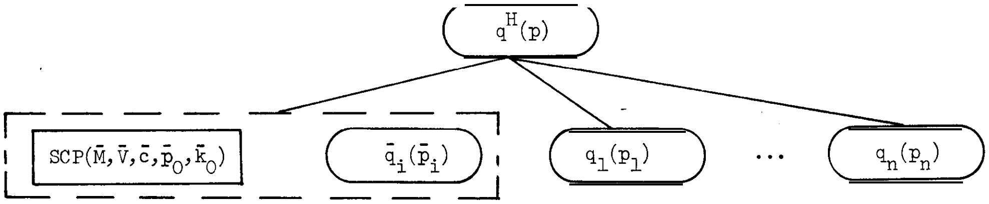


There are two modes of activating an SCP control rule; these two modes are indicated through the control information, c , used to activate the iteration macro calling -sequence that generated the SCP control rule. In the normal mode, the SCP generates and executes macro calling sequences until $k_{n+1} = 0$ while in the other mode, called single cycle, the SCP suspends its activity after the execution of each macro calling sequence, $q_i(p_i)$ i = 1,n ; upon the reactivation of the suspended SCP, depending upon whether $q_i(p_i)$ is itself terminated or suspended, either the next calling sequence $q_{i+1}(p_{i+1})$ will be generated and executed or else the suspended $q_i(p_i)$ will be reactivated. This latter mode of activating an SCP allows the concept of "time grain" to be introduced into the SBL definition of a process's control structure. The time grain of a process refers to the smallest unit of a process's activity that can be controlled; the boundaries of a time grain are the only discrete points in the activity of a process where the process's state may be considered. The two modes of activating an SCP control rule allows the time grain boundaries of a sequential process to be defined as either the termination of the process or the suspension or termination of each process statement.

Example 3A: Consider the iteration macro calling sequence A(p) which, when activated, generates and executes the following list of macro calling sequences $B^I(\ell_1)...B^I(\ell_n)$. Likewise, consider the iteration macro calling sequences $B^I(\ell_i)$ which, in turn, generates and executes the following list of macro calling sequences $C_i(J_1)...C_i(J_m)$. If the macro A^I is activated for a single cycle, and the internal parameter, c , of the SCP node of A specifies execution for all cycles, then the time grain boundary of A is the completion of each macro calling sequence $B^I(\ell_i)$. However, if the c parameter associated with the SCP node of A is set for a single cycle, which implies that SCP node of $B(\ell_i)$ is activated for a single cycle, then the time grain of the A is the time grain of the $C_i(J)$. This successive functional decomposition of a sequential process through a hierarchy of iteration macros can be continued until the desired time grain of the process is achieved.

The concept of time grain can be employed to represent concisely such control concepts as interrupts, monitoring, etc. In particular, the control structure of a sequential process that is being monitored for a specified condition can be constructed so that the process is suspended after the smallest unit of work which can effect the condition being monitored is performed. Thus, before reactivating the suspended process the condition being monitored can be checked, and if necessary, an appropriate interrupt process activated.

The ASP control rule, unlike the other three built-in control rules FCP, SEL and SCP, does not implement a predefined pattern of sequencing non-terminal nodes, but rather is a building block upon which arbitrary, possibly non-sequential, clocking processes can be defined. The activation of the ASP control rule results in the modification of the "state" of a single non-terminal node where the location of the non-terminal node is not fixed but rather specified by parameters of an ASP calling sequence. The ASP control rule is used in conjunction with a hierarchical macro prototype to define a clocking process which sequences the son nodes of the hierarchical macro. A hierarchical macro calling sequence, $q^H(p)$, when expanded, generates a list of macro calling sequences $q_1(p_1) \cdots q_n(p_n)$ and expands the macro calling sequence $(q+1)(p)$. The dotted box in Figure 5 is used to indicate that results of expanding $(q+1)(p)$ is placed in the CDS instead of a non-terminal node containing the calling sequence $(q+1)(p)$. For example, if $(q+1)$ is an iteration macro, then the expansion of $q^H(p)$ results in the following CDS:



The activation of $q^H(p)$ then results in the activation of the terminal node which is generated by the expansion of $(q+1)(p)$, e.g., in the above case, the SCP control rule would be activated. The CDS generated by the expand macro $(q+1)$ defines the control structure of the clocking process which initially sequences the sone nodes of the hierarchical macro; the basic statements of this clocking process are calling sequences to activation macros. The execution of these activation macro calling sequences results in the activation of ASP control rules; these ASP control rules, in turn, control the expansion and activation of the son nodes of the hierarchical macro.

> Example 4: Consider the emulation of a conventional computer with an interrupt structure. The control structure of the emulator for this computer can be constructed by combining together the control structures discussed in Examples 1, 2 and 3, and then adding as a superstructure a macro, COMPUTER', which specifies an interrupt control structure. Figure 3 represents this combined control structure, where SEQUNIT is a clocking process that activates the node $ISEQ^I(pc)$ for one cycle at a time (which means the grain boundary of ISEQ is the emulation of a single instruction), and then checks whether an interrupt requires servicing; if it does, then the node INTHANDLER is executed, else the node $ISEQ^I$ is reactivated and the basic sequencing cycle is repeated.

A more detailed explanation of the hierarchical macro prototype and the ASP control rule is contained in Section IV.C.

B. Control Data Structure

The CDS 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. Representation of hierarchical structure and recursion is possible because additional levels (e.g. son nodes) may be dynamically built in the tree by expansion of non-terminal nodes (macro calling sequences). 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 (using father, brother, etc., relationships between nodes) for defining distributed control systems. The control structure of a complex system can sometimes be conveniently represented through hierarchical structure where in each sibling set (or structural level) of the tree there is embedded a simple control rule (via a clocking process[14]) that initiates the sequencing of 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 rule for the entire system, the control can be distributed throughout the system. In addition, since the control rules can be coded such that their addressing structure is not based on their absolute locations in the tree, but only on their relative position in the tree, one copy of a single clocking process can be used at different points 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 activation is stored in the CDS.

A tree is the form of a CDS generated by SBL macros but does not necessarily reflect the dynamic sequencing of nodes. This separation between generation and sequencing is possible because the execution of an SBL macro is factored into three separable phases: the generation of a CDS, caused by the binding and expansion phases, can thus be separated from the execution of a CDS, caused by the activation phase. The only built-in sequencing associated with the tree is that a father node must be expanded before any of its sons. The CDS is just a convenient framework within which sequencing rules can be expressed. Thus, control structures whose CDS's are not normally represented as tree structures can also 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 passing of control during execution.

Example 6: Consider the parallel control structure defined by a fork-join instruction[15]. The fork-join control structure is normally represented in terms of the directed graph in Figure 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 Figure 5b: the clocking process Control-l 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.

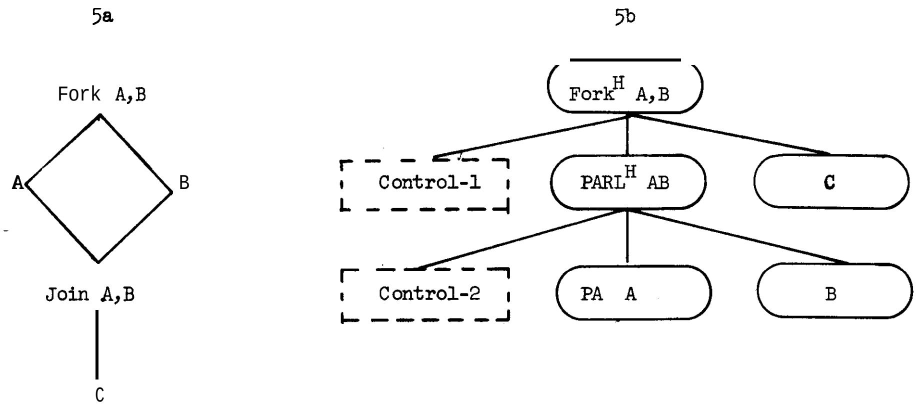


FIGURE 5. Fork-join Instruction

The separability of the three phases of macro executions also avoids the unnecessary rebuilding of the CDS when the form of the control data structure (e.g., the number of sone 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 CDS must be rebuilt; the static parts of the CDS once defined need not be regenerated. For example, the only parts of the CDS pictured in Figure 3 that need to be rebuilt for the emulation of each instruction are the values of the internal parameters calculated by IFL programs. As well, the separation between the binding phase, and the expansion and activation phase allows the external parsmeters used in executing and rebuilding parts of the CDS to be different from those used to initially generate the CDS.

Each non-terminal node in the CDS can be considered the state vector of a process, where the process is defined by the control structure generated by the expansion of the macro calling sequence stored at the node. The built-in control rule that is used to activate a non-terminal node will be called the external clocking process (ECP) of the node, while the built-in control rule that is activated as a consequence of the node being activated will be called the internal clocking process (ICP) of the node. For example, in Figure 3, the ECP of non-terminal node IDECODE(IR) is the SCP control rule contained in its brother node, while the ICP of this node is the SEL control rule contained in its first son node. The process state vector in each non-terminal node has six componenets (q,p,s,c,r,d) where q(p) 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 current state, s , of the process refers to the state of the ICP associated with the node, and has seven possible values: (1) ICP is unexpanded; (2) ICP is being expanded; (3) ICP is expanded; (4) ICP is being executed; (5) ICP is being suspended[*/];(6) ICP is suspended; and (7) ICP is terminated. The control information, c , specifies (1) the time grain of the process, (2) the conditions under which the process will signal its external clocking process, and (3) the conditions under which the process's control structure is rebuilt. The time grain of a process can be either the activation of a single process statement or the duration of the process. The internal clocking process will either return control to the external clocking process after each activation or it will retain control until the process is terminated. The immediate global control pointer, r J (conventionally called a return link) specifies the address of the external clocking process that will be reactivated. 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 J 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 the addresses of, respectively, its father node's internal clocking process and its father node. However, these are default options for r and d . They can be overridden by the activation macro in order to create a control data structure to pass control in a manner different from that of a tree structure.

The p component of a process state vector, which is the parameter of the macro calling sequence (e.g., the external parameter), contains either an immediate integer value or a pointer; there are three types of pointers: (1) a pointer to the p component of the process state vector of another node;

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

(2) a pointer to a field in the memory subsystem; and (3) a pointer to a block of auxilliary registers. The first type of pointer allows the representation of the static data relationships among external parameters contained in the CDS. In particular, the first type of pointer facilitates the definition of broadcast type control structures and control structures for parameter passage. It allows modification of external parameters at one level in the tree to be directly reflected in other levels of the tree without explicitly modifying those parameters at other levels. 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 external 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 in its memory (e.g., the PDP-11[16] computer whose program counter is stored as register 7 in its memory); this can be done by having the CDS contain a pointer to the field in the memory subsystem which contains the program counter, rather than storing the value of the program counter in the CDS itself. Thus, the emulator does not have to process in a special way instructions of the emulated computer that can possibly modify the field in the memory subsystem that contains the program counter. 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 by external parameters 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 third type of pointer allows a macro calling sequence to contain multiple parameters rather than just a single parameter. An IFL program which is called with this third type of pointer as a parameter can then directly store and retrieve data from a block of auxilliary registers.

The expansion of a SBL macro q , using parameter p , 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 parameter p can be changed by an activation macro to $\bar{p}$ J 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 parameter $\bar{p}$, upon each new execution of the process defined by the control structure. The programmer can define which of the internal parametrs 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 of the terminal node instead of an external parameter. Thus, only dynamic parts of a-control structure need be rebuilt on each execution, and only parameters with varying values need be recalculated.

C. Nonsequential Control Structures

The hierarchical macro provides a mechanism for defining control structures that contain more than one clocking process (path of control),[17] 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[18] 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. 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. A clocking process is constructed out of calls to the ASP control rule. The ASP control rule combines the control functions of process activation including parameter passage and process synchronization. The ASP control rule performs these control functions through operations on the process state vector stored at a node.

The ASP control rule pictured in Figure 5 is called with four internal parameters. The first two parameters, n and l , specify the relative address of a node in the CDS; the third parameter, svt , is a template for a process state vector where for each of the components of the 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 ASP control rule with the activity of the process located at (n,l) . The relative addressing schema used to locate a node in the CDS is the following: brother' .father" 'base-node , where the base node is the node containing the activation macro calling sequence; the d component of the process space vector is used to locate the father node. This relative addressing 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.

The activation of the built-in clocking process ASP 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, the only components of the state vector of the activated process which vary from execution to execution of the process need be recalculated. 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 activation macro which initially expands the macro calling sequence that defines control structure of the process. Thereafter, the activation 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 ASP 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 the activity of another process.

> Example 7: Consider two processes A and B, where process A calls process B as a subroutine call by executing and then waiting for termination of an ASP clocking process. In turn, the ASP 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 ASP clocking process. At the same time, the syn parameter of ASP is set up so that after process B's state vector is modified the ASP clocking process is suspended. When process B is terminated, ASP will then be re-awakened 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 canpletion of the ASP clocking process. If process A was not synchronized with the activity of process B then syn parameter of ASP would be set up so that after process B is activated the ASP process is terminated. Thus, process A after process B is activated will immediately continue processing. Process A while waiting for ASP process to terminate is not suspended because the action of suspending process A may be significant to A's external clocing process since the suspending of A means that process A has canpleted 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 canpletion.

The ASP 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 ASP clocking process attempts to execute an already executing process, the ASP 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 ASP determines which one of these options is taken: if the time grain is a single cycle the ASP is suspended, otherwise it busy waits. Thus, if two processes simultaneously issue ASP's which activate the same node (shared process), only one ASP will be allowed to execute the shared process. The other ASP 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 a basis for constructing synchronizing primitives for cooperating processes.

> Example 8: Consider the implementation of Dykstra's P and V semaphores in terms of the ASP clocking process. Let PV be a shared process where the p component of its state vector specifies the name of a semaphore variable to be operated on and 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 activating an ASP 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,l) parsmeters 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 activated by ASP 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 th PV process is suspended. This suspension of PV results in termination of ASP which then permits process L_i to continue. In the case that the semaphore cannot be decremented, the PV process modifies its own state vector component so that it does not return to ASP when it is suspended. It then extracts the address of the ASP process from its state vector, places this address in queue

associated with the semaphore name, and suspends itself. Thus, the ASP clocking process still remains in the suspended state, and therefore process L_i cannot continue. The PV process when executed for V operation increments the semaphore variable, and then checks whether there is a queued ASP process on that semaphore variable that can now be executed. If there is, this ASP node address is stored in the r component of PV state vector, and PV process then suspends itself which results in the queued ASP process being re-awakened. The ASP 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 a V operation is being done. If the PV process is busy, when ASP attempts to execute it, then ASP 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 ASP can also be used to create a new copy of a process (node) instead of calling a shared process. This creation of a new node occurs when the (n,l) parameters are (0,0) . The new node is the root node of a separate tree, and only the ASP clocking process can access this tree. It may also be advisable, for efficiency reasons in implementing lock-step (broadcast) control structures, that an ASP clocking process be able to simultaneously activate all the sibling nodes at a level in the tree, and then be able to wait for all of them to signal a return.

SUMMARY COMMENT AND FUTURE RESEARCH

This paper is an investigation of the idea of a micro-computer whose control structure is modifiable. This investigation attempts to indicate the advantages of a variable control structure micro-computer architecture over conventional micro-computer architectures: (1) a wide variety of complex sequential and parallel IMLs can be emulated on a single micro-computer; (2) the programming of emulator is simple and uniform, such that the program structure of the emulator reflects the architecture of the computer it emulates; and (3) a variety of hardware arithmetic units, I/O devices and varying numbers of micro-processors can be easily incorporated into the organization of the micro-computer.

Future research on this micro-canputer organization will attempt to develop more rigorous arguments for the merits of this proposed method for emulation. In particular, a simulator for this micro-computer organization and emulators for complex sequential and parallel IMLs will be programmed. In addition, it is planned to investigate the possibility of adding to the SBL primitive operators which control access to nodes in the CDS, 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[19], 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 IMLs will be more easily implemented, and it will be possible to protect a micro-code from interference by other micro-programs.

References

1. Hauck, E. A. and Dent, B. A. [1968]. "Burroughs B6500/B7500 Stack Mechanism." AFIPS Conf. Proc. Vol. 32.

2. Abrams, P. S. [1970]. "An APL Machine". Report No. SLAC-114, Stanford Linear Accelerator Center, Stanford University, Stanford, California.

3. Melbourne, A. J. and Pugmire, J. M. [1965]. "A Small Computer for the Direct Processing of FORTRAN Statements." The Computer Journal, Vol. 8 (April).

4. McKeeman, W. [1967]. "Language Directed Computer Design." AFIPS Conference Proceedings (FJCC67).

5. Control Data Corporation. [1966]. "6400/6600 Computer Systems Reference Manual."

6. Illiac-IV System Study Final Report [1966]. Burrough Corporation, University of Illinois, No. 09852-B.

7. "System/360 Model 40, 2040 processing unit." [1966]. IBM Field Engineering Diagrams Manual, Document No. 0223-2842.

8. Tucker, S. G. [1965]. "Rnulation of Large Systems." CACM, Vol.8, No. 12.

9. Cook, R. W. and Flynn, M. J. [1970]. "System Design-of a Dynamic Micro-processor." IEEE Transactions on Computers, Vol. C-19, No. 3.

10. Lesser, V. R. [1970]. "Direct Emulation of Control Structures by a Parallel Micro-Computer." SLAC report No. 127, Stanford University, Stanford, California.

11. Lesser, V. R. [1968]. "A Multi-Level Computer Organization Designed to Separate Data-Accessing from the Computation." Technical Report CS 90, Computer Science Department, Stanford University.

12. Lass, S. [1968]. "A Fourth Generation Computer Organization." AFIPS Conference Proceedings, Vol. 32.

13. Dennis, J. B.and Van Horn, E. C. [1966]. "Programming Semantics for Multi-programmed Computation." Comm ACM, Vol. 8, No. 3.

14. Horning, J. J. and Randell, B. [1969]. "Structuring Complex Processes." Report RC-2459, IBM Watson Research Center, Yorktown Heights, New York.

15. Conway, M. E. [1963]. "A Multiprocessor System Design." Proc. FJCC **24**, 139-146.

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

17. Bingham, H. W. and Reigel, E. W. [1969]. "Parallelism Exposure and Exploitation in Digital Computing Systems." Final technical report, Burroughs Corp, Paoli, Pa.

18. Dahl, O., and Yngaard, K. [1966]. ."SIMULA - An Algol-Based Simulation Language." Comm ACM, Vol. 9.

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

APPENDIXA

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. PDP-6 allows indirect addressing with indexing at each level of an arbitrarily long indirect chain. The 36-bit wide memory can be represented in the memory subsystem starting at bit 0 so that work K of the PDP-6 begins at bit address M[K*36] and ends at M[K*36+35] . The following IFL program finds the address of the last word in an indirect chain given the address of the first word of the chain as its p parameter and 1 as its k parameter.

	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.