46

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 559 EE&CS Integrated Circuit Memo No. 80-6 January 1980

The SCHEME-79 Chip

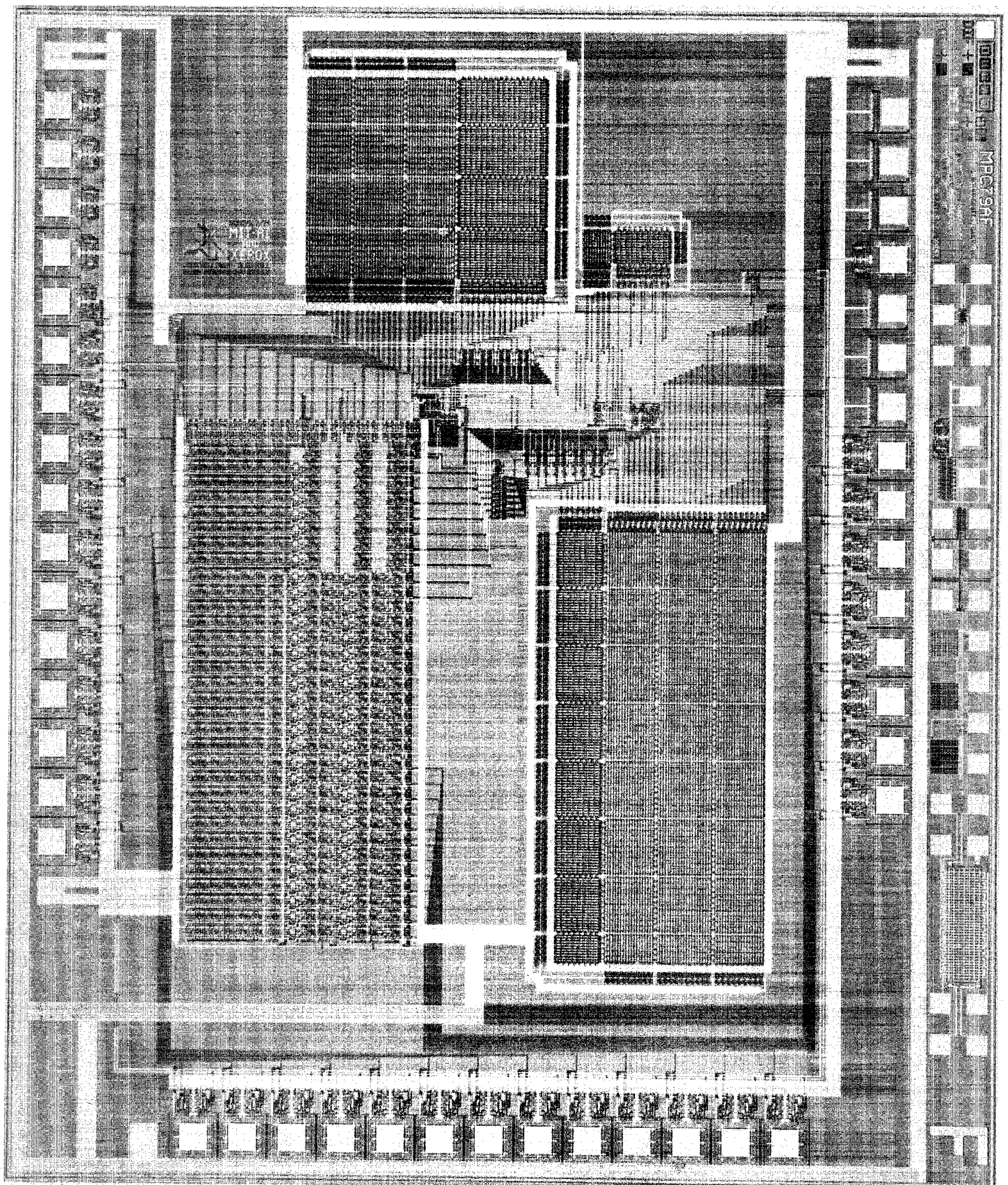
by
Jack Holloway, Guy Lewis Steele Jr., Gerald Jay Sussman
and Alan Bell

Abstract:
We have designed and implemented a single-chip microcomputer (which we call SCHEME-79) which directly interprets a typed-pointer variant of SCHEME, a dialect of the language LISP. To support this interpreter the chip implements an automatic storage allocation system for heap-allocated data and an interrupt facility for user interrupt routines implemented in SCHEME. We describe how the machine architecture is tailored to support the language, and the design methodology by which the hardware was synthesized. We develop an interpreter for SCHEME written in LISP which may be viewed as a microcode specification. This is converted by successive compilation passes into actual hardware structures on the chip. We develop a language embedded in LISP for describing layout artwork so we can procedurally define generators for generalized macro components. The generators accept parameters to produce the specialized instances used in a particular design. We discuss the performance of the current design and directions for improvement, both in the circuit performance and in the algorithms implemented by the chip. A complete, annotated listing of the microcode embodied by the chip is included.

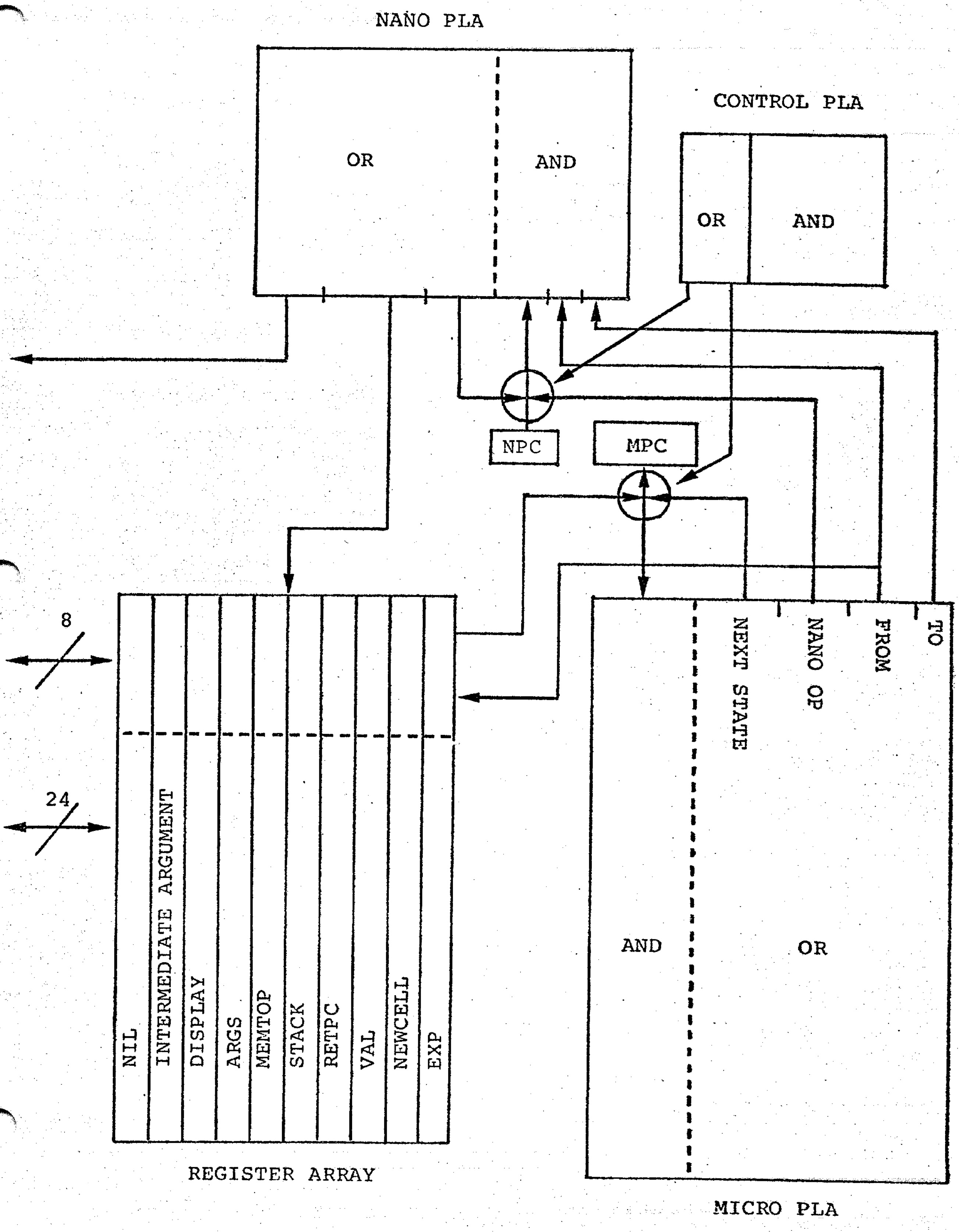
Keywords: VLSI design, Interpreters, LISP, SCHEME, microcode, design methodology.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643. This work was supported in part by National Science Foundation Grant MCS77-04828, and in part by Air Force Office of Scientific Research Grant AFOSR-78-3593.

Guy Steele is a Fannie and John Hertz Fellow, Alan Bell is a staff member of the Xerox Palo Alto Research Center System Science Laboratory, Jack Holloway is a staff member of the MIT Artificial Intelligence Laboratory, and Gerald Jay Sussman is, as usual, a Jolly Good Fellow.


MIT
and
XEROX
MPC79AF

The Major Blocks of the SCHEME-79 Chip



Introduction

We have designed and implemented a single-chip microcomputer (which we call SCHEME-79) which directly interprets a typed-pointer variant of SCHEME [Revised Report], a dialect of the language LISP [LISP 1.5]. To support this interpreter the chip implements an automatic storage allocation system for heap-allocated data and an interrupt facility for user interrupt routines implemented in SCHEME. In this paper we describe why SCHEME is particularly well suited to direct implementation of a LISP-like language in hardware, how the machine architecture is tailored to support the language, and the design methodology by which the hardware was synthesized. We develop an interpreter for SCHEME written in LISP which may be viewed as a microcrode specification. We describe how this specification is converted by successive compilation passes into actual hardware structures on the chip. To help us do this we developed a language embedded in LISP for describing layout artwork. This language allows us to procedurally define generators for architectural elements. The architectural elements are generalized macro components. The generators accept parameters to produce the specialized instances used in a particular design. In conclusion, we discuss the performance of the current design and directions for improvement, both in the circuit performance and in the algorithms implemented by the chip.

Why LISP?

LISP is a natural choice for implementation of a "higher level language" on a single chip (or in any hardware, for that matter — cf. [LISP Machine] [FLATS]). LISP is a very simple language in which only a few primitive operators and data types are sufficient for building a powerful system. In LISP there is a uniform representation of programs as data. We have extended this idea to machine language by representing machine instructions with typed list structure. Thus the same primitive operators which are used by a user's program to manipulate his data are used by the system to effect control.

LISP programs manipulate primitive data such as numbers, symbols, and character strings. What makes LISP unique is that LISP provides a construction material, called list structure, for gluing pieces of data together to make compound data objects. Thus a modest set of primitives gives us the ability to manufacture complex data abstractions. For example, a programmer can make his own record structures out of list structure without the language designer explicitly putting in special features for him. The same economy applies to procedural abstractions because of the convenient extensibility of the primitives.

In addition, LISP supports an integrated operating system, in one language, complete with dynamic linking of procedures. The uniform representation of programs as data supports the construction of interactive editors and debugging systems. These features are particularly welcome in a simple single-chip implementation.

The particular dialect of LISP we have chosen is SCHEME because of further economies it offers in the implementation of our machine. It is tail recursive and lexically scoped. Tail recursion is a call-save-return discipline in which a called procedure is not expected to return to its immediate caller unless the immediate caller wants to do something after the called procedure finishes (rather than just returning to its own caller). The use of tail recursion allows us to conveniently define all common control abstractions in terms of just two primitive notions, procedure call and conditional [Imperative], without significant loss of efficiency [Debunking]. In order for this to work, however, we must adopt lexical scoping of free variables. Fortunately, this is simpler to implement on the chip than shallow dynamic binding schemes used in traditional LISPs, and it is more efficient than the alternative of deep dynamic binding [Shallow]. In some cases, for example when rapidly changing environments as in multiprocessing applications, lexical binding is more efficient than any kind of dynamic binding strategy.

How the machine supports SCHEME

All compound data in the system is built from list nodes which consist of two pointers (called the CAR and the CDR for historical reasons.) A pointer is a 32 bit object with 3 fields: a 24 bit data field, a 7 bit type field, and one bit used only by the storage allocator. The type identifies the object referred to by the data field. Sometimes the datum refered to is an immediate quantity, and otherwise the datum points to another list node. [MDL] [LISP Machine] [ECL]

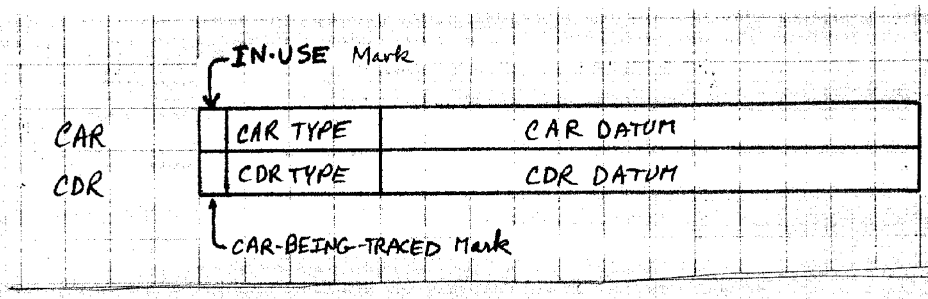


Figure 1. Format of a list node

LISP is an expression-oriented language. A LISP program is represented as list structure which notates the *parse tree* of a more conventional language. LISP expressions are executed by a process called evaluation. The evaluator performs a recursive tree walk on the expression, executing side effects and developing a value. At each node of the expression the evaluator dispatches on the syntactic type of that node to determine what to do. It may decide that the node is an immediate datum which is to be returned as a value, a conditional expression which requires the evaluation of one of two alternative subtrees based on the value of a predicate expression, or an application of a procedure to a set of arguments. The evaluator recursively evaluates the arguments and then passes them to the indicated procedure.

LISP expressions are converted into machine programs for execution by the SCHEME-79 chip. The machine programs are also represented as list structure made of typed pointers. The syntactic type of the expression (which is used by the evaluator) is encoded in the type field of the pointer to the translated expression. The transformation from LISP to machine language preserves the expression structure of the original LISP program. The type fields are also used to annotate the structure of the expression to tell the evaluator what is coming up. The type fields in our machine are analogous to the op-codes of a traditional instruction set. (Indeed, the IBM 650 [650 DPS] had an instruction set in which each instruction pointed at the next one. However, it was used to implement a traditional linear programming style, not to implement runnable parse trees.) In the SCHEME-79 architecture, the recursive tree walk of the evaluator over expressions takes the place of the linear sequencing of instructions in traditional computers. We call the machine language for our chip S-code.

The S-code representation differs from the original Lisp expression in several ways. It distinguishes between two kinds of variables in the original Lisp expression. Global variable references translate into a special kind of pointer which points at the global value of the symbol. A local variable reference is transformed into an instruction containing a lexical address of its value in the environment structure. Constants are transformed into instructions which move the appropriate constant into the accumulated value. Certain primitive procedures (for example, CAR, CDR, CONS, EQ, etc.) are recognized and are realized directly as machine op-codes. Procedure calls are converted from prefix to postfix notation. Conditionals and control sequences (PROGN) are performed somewhat differently than they are in the source language. For example, the following simple LISP program for appending two lists:

```
(defun append (list1 list2)
       (cond ((eq list1 '()) list2)
             (t (cons (car list1)
                      (append (cdr list1) list2)))))
```

is translated into the S-code shown in Figure 2 below. (The details of the S-code language are not explained here; this example is only to give an idea of what the S-code is like. The appendix gives a complete listing of the microcode interpreter for the S-code language, if you want to know the details. A user of the SCHEME-79 chip should never see the S-code.)

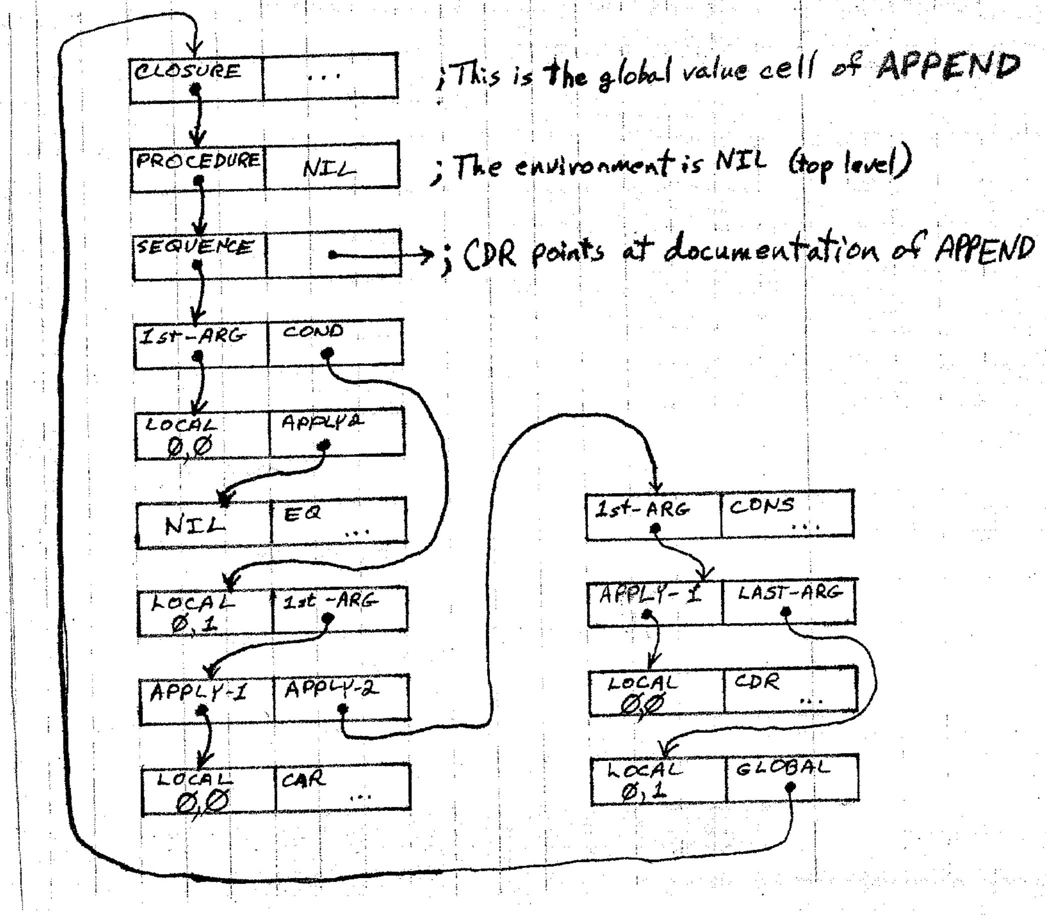


Figure 2. The S-code for Append

At each step the hardware evaluator dispatches on the state of the expression under consideration. The state of the evaluator is kept in a set of registers on the chip. There is a register for the current expression, EXP. When the value of an expression is determined it is put into the VAL register. When the arguments for a procedure call are being evaluated at each step the result in VAL is added to the list of already evaluated arguments kept in ARGS. When the arguments are all evaluated, the procedure is invoked. This requires that the formal parameters of the procedure be bound to the actual parameters computed. This binding occurs by the DISPLAY register getting the contents of the ARGS register prefixed to the environment pointer of the closed procedure being applied. When evaluating the arguments to a procedure, the evaluator may have to recurse to obtain the value of a sub-expression. The state of the evaluator will have to be restored when the sub-expression returns with a value. This requires that the state be saved before recursion. The evaluator maintains a pointer to the stack of pending returns and associated state in the register called STACK.

In our machine, even the stack is represented using list structure allocated from a heap memory. Neither the user nor the system interpreter is interested in how the memory is allocated. But we only have a finite amount of memory and normal computation leads to the creation of garbage. For example, entries built on the stack during the evaluation of subexpressions are usually useless after the subexpression has been evaluated. Thus the storage allocator must have a means of reclaiming memory which has been allocated but which can no longer affect the future of the computation so that this memory can be reused. The problem is to determine which parts of memory are garbage. There are several common strategies for this [Art]. One involves reference counts. Another, *garbage collection* (which we use), is based on the observation that if the computation involves only list operations then the future of the computation can only be influenced by objects which can be referenced by list operations starting with the contents of the interpreter's registers. That is, a cell is only in use if it is part of some accessible memory structure; however, if a cell is part of an accessible structure, it must be glued in by having a pointer to it from some other piece of accessible structure. This structure must be well founded. The only structures we know *a priori* to be in use are the machine registers. The strategy of garbage collection we use is thus the *mark-sweep* plan. We recursively trace the structure pointed at by the machine registers, marking each cell reached as we go. We eventually mark the transitive closure of the list access operations starting with the machine registers; therefore a cell is marked if and only if it is accessible. We then scan all of memory. Any location that was not marked is swept up as garbage and made reusable.

Usually recursive traversal of a structure requires an auxiliary stack. This is unfortunate for our machine since we need list structure to build stack and we are presumably garbage collecting because we ran out of memory. Deutsch, Schorr, and Waite [DSW] developed a clever method of tracing structure without auxiliary memory. We use their scheme.

In our sweep phase we use a *two finger compaction* algorithm [MDL] which relocates all useful structure to the bottom of memory. The allocation process simply increments the free storage pointer. This allows us to sweep only as much memory as has been allocated before the garbage collection. In addition, we allow the user program to set the value of the MEMTOP register, which is compared with the free-storage pointer when allocation is done. When the free storage pointer gets to MEMTOP, the user receives an interrupt (see below). He then has the option of moving the MEMTOP if he has more memory available, or of taking a garbage collection by invoking the garbage-collector as a primitive procedure. This gives the user a convenient control on how distributed he will let his working set become. In the case of a paged memory, this allows the user to trade off garbage collector time against paging time.

Our chip also supports an interrupt system. The interrupt handlers are written in SCHEME. Thus the user can, for example, simulate parallel processing or handle asynchronous I/O. The problem here is that the state of the interrupted process must be saved during the execution of the interrupt routine so that it can be restored when the interrupt is dismissed. This is accomplished by building a data structure which contains the state of the relevant registers and passing it to the interrupt routine as a continuation argument [Continuations]. The interrupt routine can then do its job and resume the interrupted process, if it wishes, by invoking the continuation as a procedure. The interrupt mechanism is also used (as described above) to interface the garbage collector to the interpreter.

The SCHEME-79 Architecture

The SCHEME-79 chip implements a standard von Neumann architecture in which there is a processor attached to a memory system. The processor is divided into two parts: the data paths, and a controller. The data paths consist of a set of special purpose registers, with built-in operators, which are interconnected with a single 32 bit bus. The controller is a finite state machine which sequences through *microcode* implementing the interpreter and garbage collector. At each step it performs an operation on some of the registers (for example, transferring the address of an allocated cell in NEWCELL into the STACK register) and selects a next state based on its current state and conditions developed within the data paths. We will see that this decomposition facilitates the automatic generation of the hardware from a program written in a high-level language that describes the algorithms.

There are 10 registers, which have specialized characteristics. To save space these registers are shared by the interpreter and the garbage collector. This is possible because the interpreter cannot run while a garbage collection is taking place (but see [Concurrent] [Real Time]). Figure 3 (below) shows (schematically) the geometric layout of the register array. The registers and operators are all sitting on the same bus (the bus lines run vertically in the diagram). The arrows in the diagram depict control actions that the controller may take on the register array or branch conditions that the register array develops which may be tested by the controller. The TO controls are used to load the specified register fields from the bus; the FROM controls are used to read the specified register onto the bus.

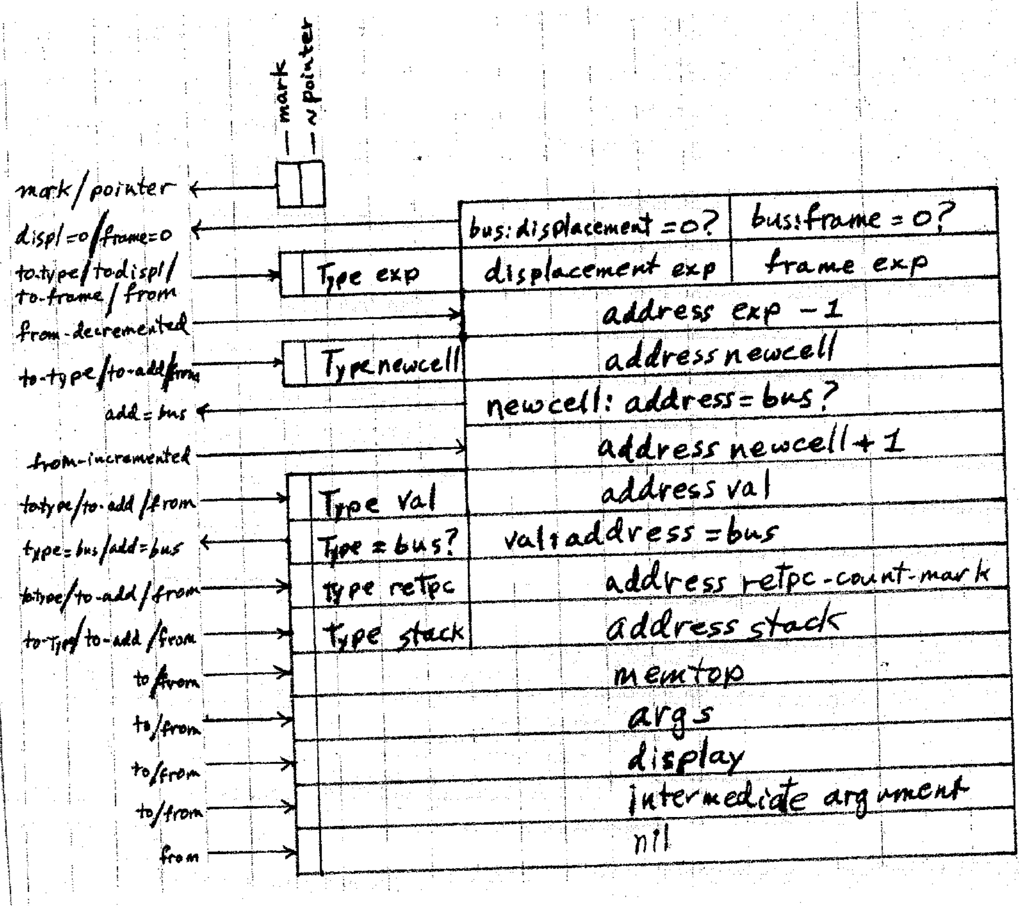


Figure 3. The Register Array

The division of storage words into mark, type, and data fields is reflected in physical structure of most of the registers. In the EXP register, the data field is further decomposed into a FRAME field and a DISPLACEMENT field; thus in some microcycle the microcode can enable TO-DISPLACEMENT-EXP and FROM-RETPC-COUNT-MARK, which will transfer the bits from the DISPLACEMENT part of the data field of the RETPC-COUNT-MARK register to the DISPLACEMENT part of the data field of the EXP register. The data part of the EXP register can be decremented and placed on the bus (that is, a number one less than the data part of EXP can be read onto the bus; the data part of EXP is not itself modified). The DISPLACEMENT and FRAME subfields of the bus can be separately tested for zero; this feature is used to implement the lookup of variables as specified by their lexical address. Since the data field of the EXP register can be decremented as a whole, it can be used as the scan down pointer in the sweep phase of the garbage collector. The register NEWCELL, which contains the free storage pointer during interpretation, can be read onto the bus either directly, or its incremented value can be read onto the bus. The same register is used to scan up in memory during the sweep phase of garbage collection. The NEWCELL register also contains logic that continually tests whether its data field (which here represents the address of the next free cell in memory) is equal to the data field on the bus. This comparison is used for checking during allocation to see if the available memory limit has been exceeded. The register NIL, on the other hand, has only one capability: it can be read onto the bus and its value is always zero in all bits.

On each cycle the registers can be controlled so that one of them is gated onto the bus, and selected fields of the bus gated into another register. The bus is extended off the chip through a set of pads. The external world is conceptualized as a set of registers with special capabilities. The external ADDRESS register is used for accessing memory and can be set from the bus. The pseudo-register MEMORY can be read onto the bus or written from the bus. The actual access is performed to the list cell addressed by the ADDRESS register. The CDR bit controls which half of the cell is being accessed. One more external register, INTERRUPT, which can be read onto the bus, contains the address of a global symbol whose value (its CAR) is an appropriate interrupt handler.

The finite-state controller for the SCHEME-79 chip is a synchronous system composed of a state register and a large piece of combinational logic, the control map (see Figure 4, below.). The control map is used to develop, from the current state stored in the state register, the control signals for the register array and pads, the new state, and controls for selection of the sources for the next sequential state. One bit of the next state is computed using the current value of the selected branch condition (if any). The rest of the next sequential state is chosen from either the new state generated by the control map, or from the type field of the register array bus (dispatching on the type).

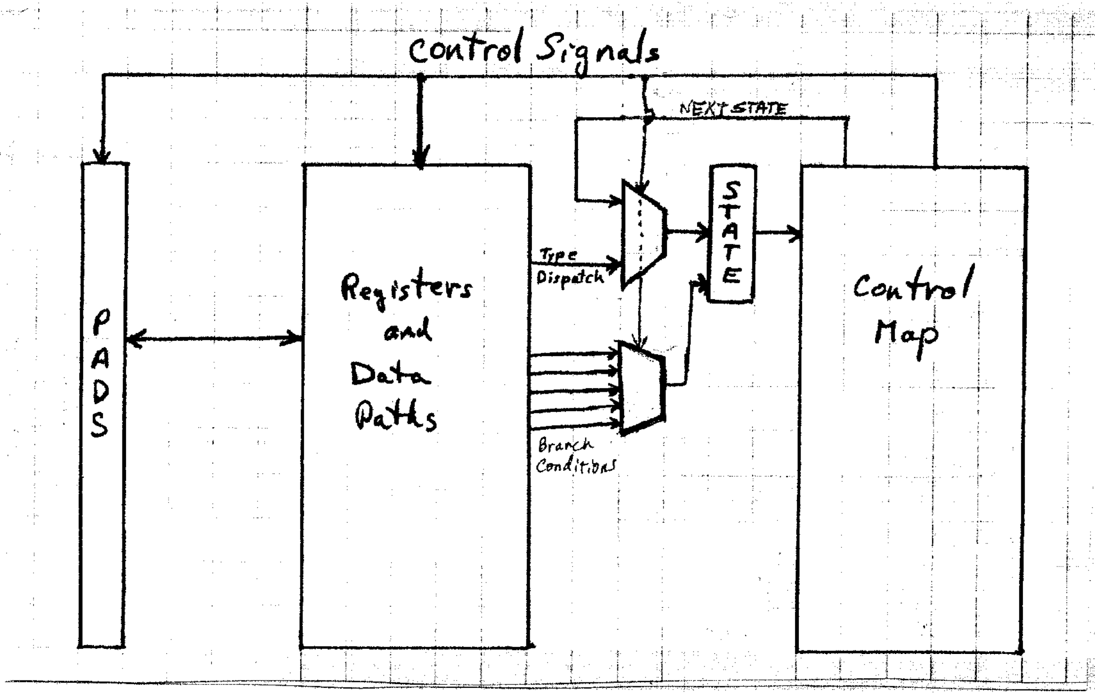


Figure 4. The Finite-state Controller

Including both the interpreter and garbage collector on the chip makes a compact realization of the map from state to new state and control function difficult. If we tried to implement this in the straightforward manner using a programmed logic array (PLA) for the map, this structure would physically dominate the design and make the implementation infeasible. The map logic was made feasible by several decompositions. Only a few of the total possible number of combinations of register controls are actually used. For example, only one register can be gated onto the bus at one time. We can use this interdependence to compress our register control. Instead of developing all of the register controls out of one piece of logic (which would have to be very wide), we instead develop an encoding within the main map of the operation to be performed and the registers involved. This encoding is then expanded by an auxiliary map (also constructed as a PLA) to produce the actual control signals. (See Figure 5 below.) This was inspired by the similar decomposition using both vertical and horizontal microcode in the M68000 [M68000] (cf. [QM-1], [Nanoprogramming]).

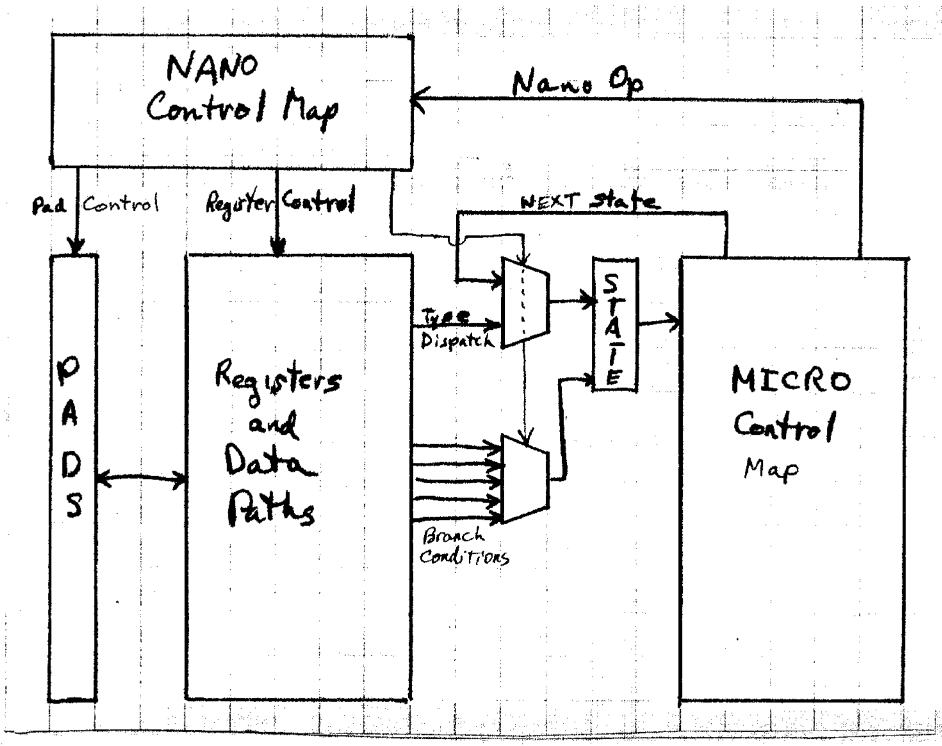


Figure 5. Two stage map

Unfortunately, this is not adequate to fix the problem. When examining the microcode, we found many microcode instructions that were nearly identical, differing only in the particular registers being manipulated. To take advantage of this regularity we extended the previous decomposition so that the encoded operation is specified independently of the registers taking part in that operation. Thus, the Micro control map develops a full-blown vertical microcode, consisting of an operation specification (the Nano opcode) and separate specifications of registers to be used as source (`from`) and destination (`to`) operands. The Nano opcode determines which of the operands are enabled for decoding (that is, whether one, both, or neither of the operands is to be used.) The vertical microcode is then expanded by the Nano control map into a horizontal microcode which is the actual register, selector, and pad controls.

Further savings were realized by noticing that there were many identical sequences of microinstructions (except for renaming of registers) which contained no conditional branches. For example, to take the CAR of a specified source register and put it in a specified destination register takes two cycles – one to put the source out on the pads and tell the memory to latch the address, and another to read the data from memory into the destination register. Since sequences like CAR are common, the microcode could be compressed if we allowed a limited form of microcode subroutine. We did this rather painlessly by extending the horizontal (Nano) microcode map to be a full state machine (See Figure 6 below.). These common subsequences are then represented as single cycles of the Micro engine which stimulate several cycles of the Nano engine. To make this work, the vertical (Micro) sequencer must be frozen while the Nano-sequencer is running. This mechanism was needed in any case to make it possible for the chip to wait for memory response.

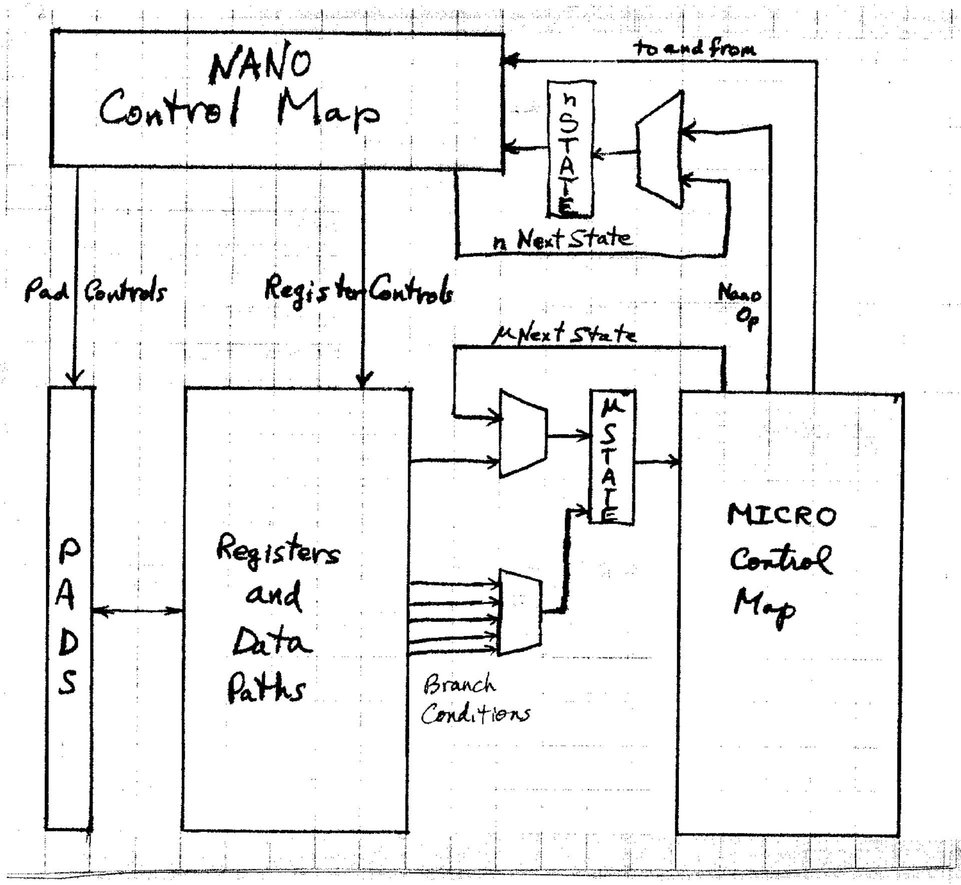


Figure 6. SCHEME-79 Control

Even further savings were realized by clever subroutinization of common subsequences in the source microcode. Some of the common subsequences could not be captured by the Nano engine because they involved conditional branch operations, which the Nano engine was incapable of performing. These subsequences could be subroutinized in the source code by the conventional use of a free register to hold the return microcode address (Micro state) in its type field. Many microcode instructions were saved by this analysis.

Synthesizing the SCHEME-79 chip

The major problem introduced by VLSI is one of coping with complexity. Logic densities of over 1 million gates per chip are predicted from extrapolating fabrication techniques that are appearing in laboratories. In the design of the SCHEME-79 chip we take advantage of the techniques and perspectives we have inherited from confronting the complexity problem in the software domain. We feel that the activity of hardware design will increasingly become analogous to the programming of a large software system.

Current practice in chip design uses the idea of a *cell*, a particular combination of primitive structures which can be repeated or combined with other cells to make more complex structure. The advantage of this approach is that one only has to define a cell which performs a common function once. More importantly, the cell encapsulates much detail so that the problem of design is simplified by suppressing that detail. As in programming we extend this notion (following Johannsen [Bristle Blocks]) by parameterizing our cells to form compound abstractions which represent whole classes of cells which can vary both in structure and function. We call these abstract parameterized cells *architectural elements*. For example, a simple nMOS depletion load pullup comes in a variety of sizes. The particular length-to-width ratio of such a transistor is a simple numerical parameter. A trivial architectural element is just such a parameterized pullup. It encapsulates the particular rules for constructing the pullup including the placement of the contact from the poly to the diffusion layer and the placement of the ion-implant. A more interesting architectural element is an n-way selector parameterized by the number of inputs. In this case higher level detail is suppressed, such as the means of driving the input lines and the particular logic by which the selector is implemented. Small selectors may be more effectively implemented with one kind of logic and large selectors may be more effectively implemented with another.

But these are still only simple parameters. We have developed much more powerful architectural elements. For example, a finite-state machine controller can be implemented as a PLA and state register. We have constructed a PLA generator which is parameterized by the logical contents and physical structure. This generator is used with a compiler which takes a program to be embodied in the state machine to produce the state machine controller, an architectural element parameterized by a program written in a high-level language. Although we have not completely parameterized it, we can think of our register array generator as an architectural element. In fact, we believe that an entire high-level language interpreter module can be successfully defined as an architectural element parameterized by the interpreter program augmented with declarations which describe how the interpreter data is represented in registers.

To achieve this goal, we need a high level language adequate for describing the algorithm to be embodied in the hardware. We chose LISP as a convenient base language for embedding the description of the SCHEME interpreter. By adding primitive operations that operate on the machine structures we arrive at a form of LISP (micro-LISP) which is suitable for expressing our microcode. Although micro-LISP has the structure of LISP, all of its basic operations are ultimately for side effect rather than for value. They represent actions that can be performed on the registers on the chip. However, by using the form of LISP we can take advantage of the features of a high level language such as conditionals, compound expressions, and user macro definitions. For example, the following fragment of the on-chip storage allocator is written in micro-LISP:

```
(defpc mark-node
          (assign *leader* (&car (fetch *node-pointer*)))
          (cond ((and (&pointer? (fetch *leader*))
                           (not (&in-use? (fetch *leader*))))
                   (&mark-car-being-traced! (fetch *node-pointer*))
                   (&rplaca-and-mark! (fetch *node-pointer*) (fetch *stack-top*))
                   (go-to down-trace))
                  (t
                   (&mark-in-use! (fetch *node-pointer*))
                   (go-to trace-cdr))))
(defpc down-trace
          (assign *stack-top* (fetch *node-pointer*))
          (assign *node-pointer* (fetch *leader*))
          (go-to mark-node))
```

This example shows two subsequences of the microcode, which are labeled `mark-node` and `down-trace`. `Down-trace` is a simple sequence that transfers the contents of the machine registers `*node-pointer*` and `*leader*` into `*stack-top*` and `*node-pointer*`. Although the storage allocator was written as if it uses a set of registers distinct from those of the evaluator, there are micro-LISP declarations that make these names equivalent to the registers used by the evaluator. The `mark-node` sequence illustrates the use of a compound expression that refers to the result of performing the `CAR` operation on the contents of the `*node-pointer*` register. The `CAR` operation is itself a sequence of machine steps that access memory. (This is an example of a nano-operation which performs a step to output the address to be accessed, followed by a step that reads the data from the external memory into the `*leader*` register.) The compound boolean expression that tests the contents of the `*leader*` register for being a pointer to an unused cell is compiled into a series of microcode branches that transfer to separate chains of microinstructions corresponding to the consequent and alternative clauses.

One benefit of embedding the microcode language in LISP is that by providing definitions for the primitive machine operators that simulate the machine actions on the registers, we can simulate the operation of the machine by running our microcode as a LISP program.

More importantly, micro-LISP is also an easy language to compile. Micro-LISP is compiled into artwork by a cascade of three compilation steps. The first phase transforms Micro-LISP into a relatively conventional machine language. It removes all embedded structure such as the composition of primitive operators. This involves the allocation of registers to hold the necessary intermediate results. The compiler tries to optimize these computations by storing intermediate results in the ultimate target of a computation, because that cannot be needed later. Conditionals are linearized and the compound boolean expressions are transformed into simple branches. The machine language is specialized to the actual target machine architecture by the code generators which transform the conceptually primitive operators of Micro-LISP into single major cycle machine operations, which can be encoded as single transitions of MICRO (the vertical microcode state machine). This phase does not know how these operations will be executed by NANO (the horizontal microcode state machine).

For example, the fragment of Micro-LISP from the storage-allocator shown above is translated into the following microcode instruction sequences:

```
MARK-NODE-1
   ((FROM *DISPLAY*) (TO PADS) ALE)
   ((FROM PADS) (TO *INTERMEDIATE-ARGUMENT*) READ)
   ((FROM *INTERMEDIATE-ARGUMENT*) (TO PADS) WRITE MERGE-MARK-1
        (GO-TO TRACE-CDR))
MARK-NODE-2
   ((FROM *DISPLAY*) (TO PADS) ALE)
   ((FROM PADS) (TO *INTERMEDIATE-ARGUMENT*) READ CDR)
   ((FROM *INTERMEDIATE-ARGUMENT*) (TO PADS) WRITE MERGE-MARK-1 CDR)
   ((FROM *DISPLAY*) (TO PADS) ALE)
   ((FROM *VAL*) (TO PADS) WRITE MERGE-MARK-1 (GO-TO DOWN-TRACE))
MARK-NODE-3
   ((FROM *ARGS*) (TO PADS) ALE)
   ((FROM PADS) READ (BRANCH MARK-BIT-BUS MARK-NODE-1 MARK-NODE-2))
MARK-NODE
   ((FROM *DISPLAY*) (TO *ARGS*) DO-CAR)
   ((FROM *ARGS*) (BRANCH TYPE=POINTER-BUS MARK-NODE-3 MARK-NODE-1))
DOWN-TRACE
   ((FROM *DISPLAY*) (TO *VAL*))
   ((FROM *ARGS*) (TO *DISPLAY*) (GO-TO MARK-NODE))
```

Each micro-instruction specifies a source register (`from ...`), destination register (`to ...`), a set of operations and controls, and possible branch conditions. For example, the alternative of the conditional in `mark-node`,

```
(t
 (&mark-in-use! (fetch *node-pointer*))
 (go-to trace-cdr))))
```

translates into micro-sequence labeled `mark-node-1`. Notice that the registers named in the Micro-LISP code have been translated into their equivalent evaluator register names (`*node-pointer*` $\longrightarrow$ `*display*`). The `&mark-in-use!` operation performs a memory cycle that merges a mark bit into the contents of the CAR of the node addressed by the register. The first micro-instruction puts the register contents onto the chip's data bus and emits the ALE control (to set the ADDRESS pseudo-register). The following step reads CAR of the ADDRESSed node into a temporary register. Finally, the temporary is re-written into memory with the `merge-mark-1` control which modifies the bus contents (by forcing the mark bit to be 1) on the way to the pads, and the sequence continues at `trace-cdr`.

The second phase of the compilation process converts the microcode into specifications for the PLA's that control the Micro and Nano sequencers. For each sequencer, a micro-word (which will be the contents of one row of the OR-plane of the PLA) must be constructed for each microinstruction of the microprogram. Then each microword must have a state number (i.e. program address) assigned to it. These state numbers are used to control sequencing from one microinstruction to another. The assignment of state numbers is affected by certain constraints (described below). Once state numbers are assigned, then references to tags (which are symbolic names of states) must be replaced by the equivalent state numbers. The assembled microwords will become the programming for the PLA OR-plane, and the AND-plane will decode the input state number to enable the appropriate microword (in the case of the Nano PLA, more than one micro-word may be enabled, and all enabled micro-words are ORed together).

The contents of the Nano PLA are constructed first. Now the Nano PLA actually performs three separate decoding functions; it decodes the `from`, `to`, and operation fields of the microinstructions in the Micro PLA. Each of these three decoding functions is compiled separately. First, all the symbols which appear in `from` specifications in the symbolic microcode are gathered together (in the above sample microcode, these symbols include `*display*`, `pads`, `*intermediate-argument*`, `*val*`, and `*args*`). Each of these symbols is assigned a number; these numbers will be used to encode the `from` field in microwords in the Micro PLA. Microwords are then assembled for the Nano PLA which will decode the `from` field number and raise the appropriate register from-control line. For example, suppose that `*args*` is assigned the number 6 for the `from` field. Then there will be a Nano microword which is enabled iff the `from` field is 6, and which has all its bits zero except for the "from-args" bit.

The same gathering and number assignment is then performed for all `to`-field specifications in the symbolic microcode. In general, the set of symbols appearing in `to` specifications is not the same as for `from` specifications. (For example, there is a "from-NIL" signal, but no "to-NIL" signal; it is meaningless to have "`(to nil)`". Conversely, there are signals "to-stack-type" and "to-stack-address", so that each can be written separately, but there is only one signal "from-stack".) It follows that the assignment of numbers is unrelated. For example, `(from args)` might be assigned the number 6, but `(to args)` might be assigned the number 15. The two signals are distinct and are encoded in different fields of Micro microwords.

Next a similar operation is performed for all distinct combinations of operators appearing in symbolic microinstructions. In the example, such combinations include "`ale`", "`do-car`", and "`read cdr`". Each such combination is assigned a number to be used in the operation field of Micro microwords. If the combination takes more than one Nano microword to express, then several numbers are assigned, one for each Nano-word; the Micro-word will then contain the number of the first Nano-word of the sequence, and each Nano-word except the last will specify the number of the next Nano-word in the sequence, and also assert the `nano-run` bit to indicate that the sequence is not complete (asserting the `nano-run` bit inhibits the stepping of the Micro sequencer to the next Micro-word).

Nano-instructions (whether single ones or sequences) can be viewed as macros or subroutines invoked by Micro-instructions. The operation field of the Micro-instruction selects the Nano-instruction; the Nano-instruction can then selectively enable decoding of the `from` and `to` fields. For a single-word Nano-routine, both are automatically enabled. Multiple-word Nano-routines are predefined, and specify when to enable the decodings. For example:

```
(defnano (do-car)
   ((from*) (ale))
   ((to*) (read)))

(defnano (do-cdr)
   ((from*) (ale))
   ((to*) (read cdr)))

(defnano (do-restore)
   (() (from-stack ale))
   ((to*) (read))
   (() (read cdr to-address-stack to-type-stack)))
```

The definition of `do-car` says that it expands into two Nano-words. The first enables `from` decoding and asserts `ale`; the second asserts `read` and enables `to` decoding. (The symbols `from*` and `to*` may be considered to be pronouns, meaning "whatever `from` or `to` was specified in the Micro-word".) This definition is assembled into two Nano-words as follows: (This fragment of printed output from the compiler is included to show its flavor, the details are irrelevant.)

```
(NANO-RUN ALE (NANO-OR-STATE 100))
——→       ((000400 177400) (00 00 200) 20000000000020100)
                                        ;Enable FROM, ALE, next-state 100
(READ)
——→       ((001000 177400) (00 00 100) 00100000000000000)
                                        ;Enable TO, READ
```

The first one is assigned state number 200, and the second one number 100. The first one contains the number of the second one, and also asserts `nano-run`. Any Micro-word which specifies the `do-car` operation will have state number 200 assembled into its operation field.

Once the Nano PLA contents have been assembled, then the Micro PLA contents are assembled. As for the Nano PLA, each Micro-word must be asigned a state number. There are several constraints that affect the assignment of state numbers for the Micro PLA. One complication is introduced by the technique used to implement conditional Micro-code branches. When a branch occurs, the condition being tested is developed as a boolean bit value which is then merged into the low bit of the next state value. This means that the two targets of the branch must be allocated in an even/odd pair of states. If a given Micro-word is the target of more than one branch, then several copies of it may have to be made (for example, one in an odd location and another in an even location). Another complication is that the assignment of states has to be compatible with the assignment of type numbers (the S-code operations) used by the dispatch mechanism of the Micro state machine.

For example, consider a fragment of microcode as produced by the compiler from Micro-LISP, and its corresponding PLA specification:

```
MARK-NODE
   ((FROM *DISPLAY*) (TO *ARGS*) DO-CAR)
   ((FROM *ARGS*) (BRANCH TYPE=POINTER-BUS MARK-NODE-3 MARK-NODE-1))
```

Once the assignments for nano-ops have been chosen (durring assembly of the Nano PLA), the contents of the Micro PLA are assembled from the original microcode as follows:

```
MARK-NODE ((FROM *DISPLAY*) (TO *ARGS*) DO-CAR))
——→       (554 1503401250 (250 200 007 15))
                                        ;GO-TO 250, DO-CAR, FROM 7, TO 15
MARK-NODE+1 ((FROM *ARGS*)
               (BRANCH TYPE=POINTER-BUS MARK-NODE-3 MARK-NODE-1)))
——→       (052 0003343044 (044 307 006 00))       ;GO-TO 44, BRANCH, FROM 6
```

The first Micro-word specifies `do-car`, and so has 200 in its operation field. It also specifies `(to *args*)`, and so has 15 in its `to` field. The *second* micro-word has been assigned Micro-state number 250, and so the *first* Micro-word contains 250 in its next-state field. The second Micro-word contains a branch, and so the two branch targets must be allocated in an even/odd pair. Hence a copy of the Micro-word labelled `mark-node-3` has been assigned Micro-state number 44, and a copy of the Micro-word labelled `mark-node-1` has been assigned number 45.

The third major phase of the compilation is performed by the PLA architectural element generator. One parameter is the PLA specifications output by the previous phases. Other parameters control special details of the clocking, the order of bits in fields in the inputs and output wiring, the ground mesh spacing, and the option of folding the PLA for adjustment of the aspect ratio. These parameters provide flexibility in accommodating the PLA layout to the other structures on the chip.

The Layout Language

In our system, an architectural element generator is a procedure, written in LISP augmented by a set of primitive database and layout operators. This augmented LISP is the Layout Language. An architectural element generator builds an annotated representation of the particular artwork that implements an instance of the element described by the parameters.

The Layout Language primitives create representations of elementary geometric entities such as points and boxes on a particular mask layer. For example:

```
(PT 3 4)    is the point at location (3,4)
(THE-X (PT 3 4))    is 3
(BOX 'POLY 3 4 5 6) is a box on the poly layer from (3,4) to (5,6)
```

Sequences of connected boxes on a single layer can be conveniently constructed by specifying a width, an initial point and a sequence of directions for proceeding along the desired path. The paths must be rectilinear; they are specified as a sequence of movements in either coordinate direction (X or Y). Each movement can be either incremental or to an absolute position.

```
(BOXES (POLY 3) (PT 3 4) (+X 20) (Y 35) (X 70))
        specifies a path in 3 wide poly from (3,4) to (23,4)
        to (23,35) to (70,35).
```

The layout language also provides, as primitive, certain common structures which are used in nMOS layouts, such as contact cuts between the various layers.

```
(CONTACT V POLY (PT 3 4)) makes a metal-to-poly contact which is
        vertically oriented and is centered at point (3,4).
```

We can combine pieces of layout by instantiating each piece separately. We can then make compound cells by giving a name to a program which instantiates all of its pieces. The cell can be parameterized by means of arguments to the procedure which generates it. The following layout procedure creates a depletion-mode pullup of a given width and length. It encapsulates knowledge of the design rules by referring to globally declared process parameters.

```
(deflayout general-pullup (length width)
   (boxes (diff width) (pt 0 -1) (+Y (+ length 2)))
   (boxes (poly (+ (* 2 *poly-overhang*) width)) (pt 0 0) (+Y length))
   (call (butting-contact))
   (boxes (implant (+ (* 2 *implant-overhang*) width))
          (pt 0 (- 0 *implant-overhang*))
          (+Y (+ length (* 2 *implant-overhang*)))))
```

A compound cell can be instantiated as part of a larger structure by invoking its name. The instance created by a call to a compound cell can be translated, rotated and reflected to place it appropriately in the larger structure (cf. CIF in [VLSI Systems]). For example, we could get a pullup where we want it by writing:

```
        (call (general-pullup 8 2) (rot 0 1) (trans (pt 24 35)))
```

Each object can be given a local symbolic name relative to the larger structure of which it is a part. These names can be referred to by means of `paths` which describe the sequence of local names from the root of the structure [Multics]. These symbolic names, which are attached to the coordinate systems of cells, are useful for describing operations to be done when creating artwork without explicitly writing the numerical values of the operands. For example, if we wish to place a driver cell at the end of a register, so that its control outputs align with (and connect to) the control inputs of the first cell of the register column (think of the rows of the register array as bit-slices and the columns as the individual registers), we can call the driver cell generator to get an instance of the driver. We then can align the newly called out instance so that the appropriate points are coincident. We also can inherit names from substructure to make new local names.

```
            (set-the 'exp-driver (call (regcell-driver)))
            (align (the exp-driver)
                   (the-pt end ph1-to ph2-driver exp-driver)
                   (the-pt to-type-exp array))
            (set-the 'to-type-exp (the-pt start sig to-driver exp-driver))
            (set-the 'from-exp (the-pt start sig from-driver exp-driver))
```

(A form `(the-pt ...)` is a path-name: "the point which is the `end` of the `ph1-to` of the `ph2-driver` of the `exp-driver` (of me)".)

Virtual instances of cells can be made. These do not actually create artwork but they can be interrogated for information about the properties of the virtual artwork, such as the relative position of a particular bus metal or the horizontal pitch. In the following fragment `pullup-pair` is made to be the local name of a virtual instance of the `pullup-pair` cell. This is then used to extract parameters to control the elaboration of `pullup-gnd` so that it is compatible with the `pullup-pair` cell and can be abutted with it.

```
(deflayout pullup-gnd (gnd-width)
   (set-the' v-pitch -10)
   (set-the' pullup-pair
             (invoke* (pullup-pair gnd-width)))
   (set-the' vdd
             (wire (metal 4) (pt (the-x end vdd pullup-pair) 0) (Y -10)))
   (set-the' vdd2
             (wire (metal 4)
                   (pt (the-x end vdd pullup-pair) -5)
                   (X (the-x vdd2 pullup-pair))))
   (set-the' gnd
             (wire (metal gnd-width)
                   (pt (the-x end gnd pullup-pair) 0)
                   (Y -10)))
   (boxes (metal 4) (pt (the-x end gnd) -5) (X (the h-pitch pullup-pair))))
```

The layout language system is intended to be part of an interactive environment in which designs are developed by combining incremental modifications to and adaptions of existing fragments of design. Thus the layout procedures are not just a file transducer which takes an input design and cranks out artwork. The layout procedures produce a data-base which describes the artwork to be constructed. The output artwork is just one way of printing out some aspects of this data base. Other information contained in the data base is the user's conception of the structure of the artifact he is constructing. He has mnemonic names by which he refers to parts of his design. This data base can be interrogated by the user to help him examine his design, to annotate it, and to help him produce incremental changes when debugging is necessary.

SCHEME-79 Interface Description

The SCHEME chip interfaces to the external world through a 32 bit bi-directional data bus (*chip-bus*) that is used for specifying addresses, reading and writing heap memory, referencing I/O devices, reading interrupt vectors, and accessing the internal microcode state during debugging. On any cycle where the chip is not performing an explicit transfer over the chip-bus it outputs whatever is on the register array's internal data bus. Besides the chip-bus and power connections, there are also pins for control inputs (`ph1`, `ph2`, `freeze`, `read-state`, `load-state`, and `interrupt-request`) and outputs (`ale`, `read`, `write`, `cdr`, `read-interrupt`, and `GC-needed`). Figure 7 is the logic symbol for the SCHEME-79 chip.

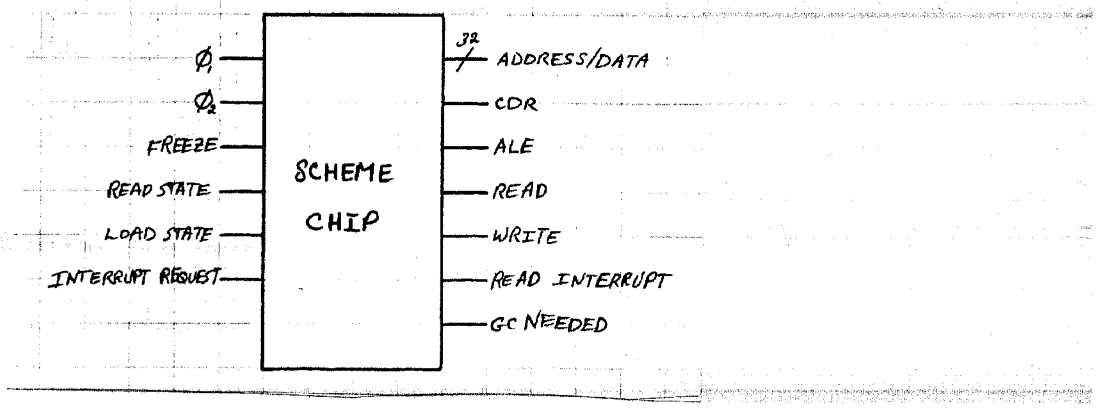


Figure 7. Chip Logic Symbol

The chip is driven by a two-phase non-overlapping clock supplied externally on the `ph1` and `ph2` pins. During the `ph1` phase, the chip performs the data manipulations specified by the current microcode state and computes the new microcode state specified by the MICRO and NANO PLAs. If the chip is outputting onto the chip-bus, the data will become valid by the *end* of `ph1`. If external data is being read it should be valid soon after the beginning of `ph1` so that any microcode branch that tests the read data will have had time to propagate through both of the PLAs. During `ph2` the chip transitions to the new state and the control outputs change accordingly.

The control signal `freeze` causes the chip to inhibit any state change. It must be stable during `ph1`. When asserted, it inhibits the `from-x` and `to-x` controls in the register array, and causes the two state machines to refetch their current state. The control outputs are not specifically disabled, so the external interface should ignore them during cycles that are frozen.

A reference to heap memory normally consists of two cycles: First, the chip sources the 24-bit node address on the low-order bits of the chip-bus and asserts address-latch-enable (`ale`). Then a following cycle asserts either `read` or `write` to access the node, simultaneously specifying which half (CAR or CDR) of the node with the `cdr` signal. For memories that are slower than the chip, the external memory control should assert `freeze` until the cycle where the memory can complete the memory operation. A timing diagram for a typical read cycle is shown in Figure 8, Figure 9 shows a double write cycle (changing both the CAR pointer and the CDR pointer of the LISP node pointed at by the address latched by the `ale`).

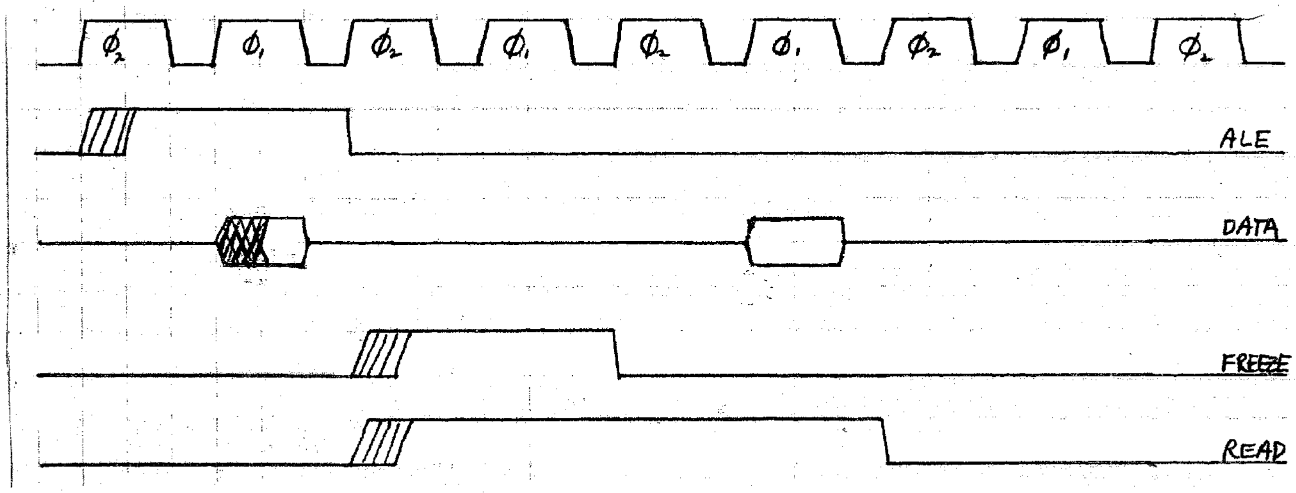


Figure 8. Read cycle (with one freeze cycle)

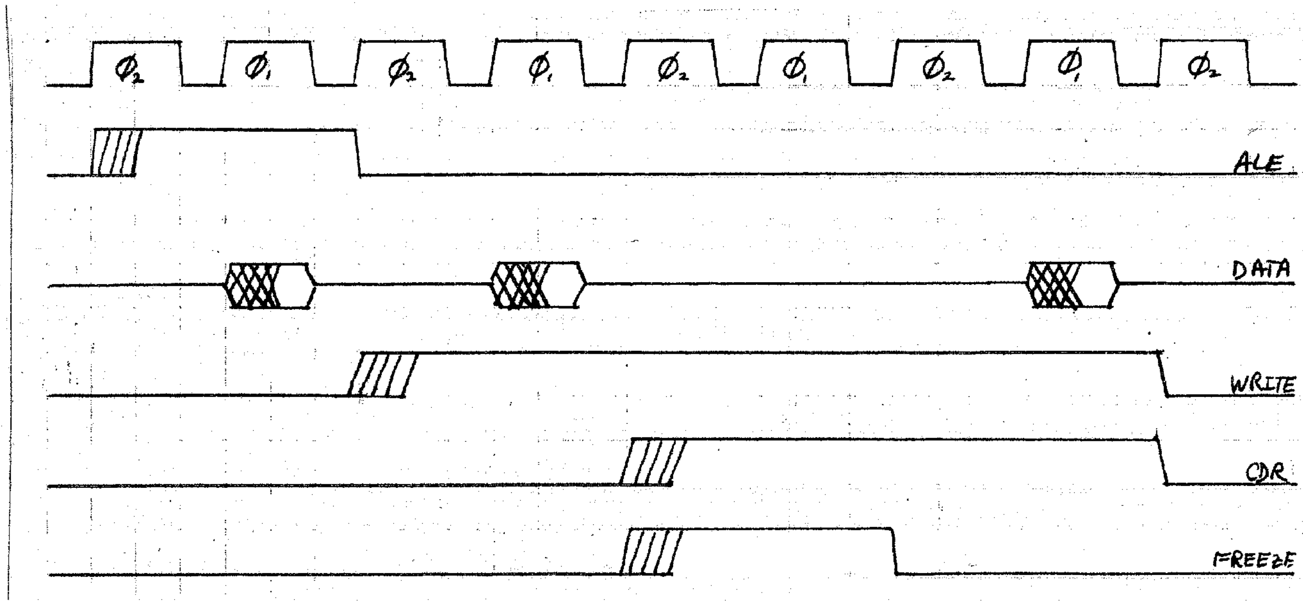


Figure 9. Double write cycle (with 2nd cycle frozen)

I/O devices are mapped into a high part of the node space that is never reached by the allocation/collection mechanism. They are accessed by `CAR`, `CDR`, and `RPLACx` operations on `self-evaluating-immediate` quantities (numbers) that point to the I/O locations. To be useful, the chip interface should supply data with valid and consistent data types to read references. Besides the normal sort of I/O devices, other pseudo-memory locations are assumed to exist that provide those primitive functions not implemented internal to the SCHEME-79 processor, such as fixed and floating point arithmetic.

Interrupts

The SCHEME-79 chip provides a simple facility for interrupting user processes. At points where interrupts can occur, the microcode examines the external input `interrupt-request`. If it is true, an *interrupt-point* is created which encapsulates the state of the current process, and a `read-interrupt` cycle is performed to obtain an address from the interrupting device. The CAR of this address should be a user procedure of one argument (the interrupt-point) which will handle the interrupt. We assume some external I/O locations that control interrupt priority level and re-enabling of the interrupt system.

Interestingly, in our system garbage collections are initiated as if they were external I/O interrupts. This allows the user to provide a *GC-handler* function that can be invoked whenever a garbage collection is called for, a convenient system feature. During all `CONS` operations, the on-chip nano-code checks the allocation pointer (`*newcell*`) against the limit register (`*memtop*`). If these registers are equal, a flip-flop is set that appears externally on the `GC-needed` pin. The external interrupt interface is expected to turn this around as an interrupt request and provide the appropriate vector when the `read-interrupt` is given. The interrupt-point passed to the GC-handler serves as both the continuation of the interrupted process and also as the root from which the garbage collector marks. GC interrupts can occur during user interrupt sequences, but they ought not in turn be interrupted. The S-code primitive operation `mark` clears the internal GC-needed flip-flop and performs a garbage collection.

Debugging

Three control lines give us the ability to initialize the chip microcode state, and indirectly let us examine from and deposit into the internal data registers. For these functions to work, the chip must be stopped with `freeze` asserted. `Load-state` sets the microcode state number from 9 bits of the chip-bus (bits 23,22,30-24). `Read-state` will read out the current `MICRO` state onto the same bits. Then, for instance, the chip can be single stepped by lowering `freeze` for one cycle, and the internal data observed on the chip-bus. There are special sequences in the microcode that have been provided just for scanning in and out the internal registers, so that that a `load-state` to the right state followed by single-stepping the chip will accomplish an examine or deposit from the "switches".

Boot-load sequence

The SCHEME-79 chip requires some assistance when cold starting after power-up. We assume that memory has been initialized to the following contents:

	CAR	CDR	
0:	`NIL`	`NIL's property list`	`;has OBARRAY`
1:	`initial *MEMTOP*`	`---`	

The bootload sequence starts at MICRO-state 111111110, and as the first thing, loads the internal *memtop* from the CAR of location 1, and then does a read-interrupt to get an interrupt vector which designates a cell whose CAR has the initial (and sole) procedure to be executed. (This may be a *read-eval-print* loop for example.)

History

This project started with the (painful) hand layout of a prototype interpreter chip by Steele [SCHEME Chip 0] [SCHEME Chip 1] as part of the 1978 MIT class project chip. The prototype differs from our current design in several ways. It used a separate set of registers for the storage allocator process and the evaluator process while our new design shares the same set of registers between the processes. Although this saves space (by time multiplexing) it precludes the use of a concurrent garbage collector algorithm. The prototype chip was fabricated but it was never tested because a fatal artwork error was discovered (through a microscope!).

We started the design for the SCHEME-79 chip at the MIT AI Lab on (or about) 18 June 1979. The first task was to construct the microcode we wanted the machine to interpret. We adapted a previous experimental implementation of the SCHEME language (written in LISP) for the PDP-10, by defining a storage representation and adding a garbage collector. We studied the microcode to determine what architecture would be effective for its implementation. The next step was to create a layout language so that we could write procedures that would build the chip artwork. This was followed by the simultaneous construction of a compiler of the microcode to the PLA implementation, and the layout of the register array. Complete, preliminary implementations of the main structures of the chip were ready in our database by 8 July 1979.

At this point we went to XEROX Palo Alto Research Center (SSL) to use their automated draftsman program [ICARUS] to assemble and interconnect these pieces. This was the hardest part of the job. It took almost two weeks to do this. During this time we had much support and encouragement from Lynn Conway and the XEROX staff. The first completely assembled version of the SCHEME-79 chip was completed at 0600 PDT on 19 July 1979. At this point the implementation was considered done except for errors to be discovered.

Some of the errors were discovered by the sharp eyes of people at XEROX and MIT. We had an 8x10 foot check plot with features of approximately 1/10 inch. Within a few weeks we had discovered and corrected about 10 design rule violations or artwork errors, and also 2 non-fatal logic bugs. At this point, a program was written by Clark Baker that extracted an electrical description (in terms of nodes and transistors) from our artwork. This discovered an implausible circuit fragment which, when investigated, turned out to be an extra power pad superimposed on one of the data pads! This electrical description was then simulated with a program written by Chris Terman (based upon an earlier program developed by Randy Bryant [MOSSIM]). Before we were done, the simulator helped find 5 additional serious logic errors, a rare PLA compiler bug, an invisible (1/8 minimum feature size) error introduced in the artwork conversion process, and an extraneous piece of polysilicon that had appeared magically during the repair of previous errors. The final simulations checked out a complete garbage collection and the evaluation of a trivial (450 microstep) user program. This experience indicates that efficient tools for checking both the physical and logical design are essential.

The chip went out for fabrication as part of the MPC79 Multi-University Multiproject Chip-Set compiled by the LSI Systems Area, of the System Science Laboratory, of the Xerox Palo Alto Research Center on 4 December 1979. Using a process with a minimum line width of 5 microns (LAMBDA = 2.5 microns) the

SCHEME-79 chip was 5926 microns wide and 7548 microns long, for a total area of 44.73 square millimeters. The masks were made by Micro Mask, Inc. and the wafers were fabricated by Hewlett-Packard's Integrated Circuit Processing Laboratory.

We received 4 chips bonded into packages on 9 January 1980. By that time Howard Cannon had designed and fabricated a board to interface the SCHEME-79 chip to the MIT LISP Machine. This was a substantial project which included an interface to allow our chip to access LISP Machine memory. The interface board contains a map for chip addresses to LISP Machine memory addresses, a programmable clock to allow the LISP Machine to vary the durations of the clock phases and the interphase spaces, debugging apparatus to allow the LISP Machine to set and read the state and internal registers of the chip, circuitry for allowing the LISP Machine to interrupt the SCHEME chip, circuitry to allow the LISP Machine to single-step the chip, and an interface from chip bus protocols to LISP Machine bus protocols. This interface project materially contributed to the success of the SCHEME-79 chip project by allowing us to begin to test the chip almost instantly on receipt.

The first chip we unpacked had a visible fatal flaw in the metal layer. The second one we tried could load and read state but would not run. The third chip seems to work! It has successfully run programs, garbage collected memory, and accepted interrupt requests. We have found two (non fatal) design errors which had escaped our previous simulation and testing. One is a subtle bug in the garbage-collector microcode which could cause rare disasters, but which will never be a problem in actual programs likely to be run by the chip. Another is a race condition in the logic associated with the pad which is used by the chip to signal the need for an interrupt to collect garbage. Luckily, this function is redundant and can be assumed by the interface.

Performance of the SCHEME-79 Chip

The main goal of the design of the SCHEME-79 chip was to have a project that would provide us with experience in VLSI design and to force us to test our ideas about design methodology and tools. Many simplifications were made to allow us to complete a design within a short time. (The entire project, including prototype tool building and chip synthesis, was completed in 5 weeks.) In every instance where there was a tradeoff between performance and simplicity, we opted for simplicity. But it is still interesting to consider what some of those tradeoffs were, and what could be done to improve the performance of a machine based upon this design.

Although we cannot definitively characterize the performance of the SCHEME-79 chip, because we have not had enough experience yet with working parts, we can estimate the performance. We will try to identify the critical determiners of this performance and offer suggestions for future modifications which will improve it. There are three areas which independently limit the performance of our machine. The algorithms embodied by the chip are not optimal. There are architectural improvements that can reduce the number of machine cycles per step in the interpreter. And there are electrical performance limits in our design.

For the sake of uniformity, the interpreter on the chip uses the heap memory system for all of its data structures, including the interpreter's stack. Since usually the interpreter stack is not retained from subexpression evaluation to subexpression evaluation (although it can be retained if an interrupt occurs or the user creates a control point) it is not usually necessary that the stack be allocated from heap memory. In fact, most implementations of LISP use a linear stack structure. This is more efficient because the garbage cells resulting from evaluating a subexpression are reclaimed explicitly rather than through garbage collection. There is also a factor of two more pointers needed to represent the stack stored in the heap than in the linear case (this would not be true if the SCHEME-79 machine used cdr-coded list structure [CDR Coding]). This becomes the dominant source of garbage produced by execution of a program, and is thus the dominant cost of execution.

Specifically, we have estimated by examining fragments of the microcode for the interpreter that we allocate a heap cell for every 10 cycles of computation. The cost of allocating this cell is broken into two parts: approximately 8 cycles are required to perform the allocation, and, under the assumption that one half of the heap is reclaimed per garbage collection, 35 cycles are required to reclaim each cell. This, of course, implies that the current machine will spend 80% of the time in the storage allocator. The garbage collector itself has respectable performance. Assuming a 1 MHz clock (which is what preliminary measurements seem to indicate is the speed of the current part), collection of a 1 megabyte heap (128K LISP nodes) takes less than 6 seconds. Approximately a third of the total time is spent in each of the mark, sweep, and relocate phases of the garbage collection. It would be difficult to make a significant impact on the performance of the machine by improving the process of garbage collection. Obviously, we must reduce the amount of garbage generated in the interpretation process.

There are several ways to attack this problem. One way is to make the evaluator use a more traditional stack structure. This can only be done at the cost of increasing complexity in many parts of the system. For example, there would have to be a mechanism for allocating non-list structures in memory. The garbage collector would need to treat these structures specially, and would also have to be able to mark from pointers stored in the stack. Also, a linear stack regime makes retention of control environments very complicated [Spaghetti]. Specifically, this impacts user features concerned with multiprocessing such as interrupts and non-local catch points.

Richard Stallman has observed that one can make stack out of list nodes, as we do, but by allocating it from a separate region, the stack will usually be a linear sequence of cells. Thus, if we can be sure that there are no pointers to the current stack region, pops can be done by just moving the stack pointer as in a traditional machine. If, however, the control environment is to be retained, we cannot just deallocate linearly, but rather we must defer to the garbage collector as usual. Since it is only the construction of retained control environments which makes the stack non-linear, and since these retained control environments are only constructed at known times by either the user or the interrupt system, it is possible to know just which control environments are to be retained. Stallman suggests having a movable base register which points to the base of the linear (not retained) portion of the stack. When the stack is retained, the base register is just moved up to the stack pointer. When the stack is popped, it may be deallocated unless the pointer is below the base register. Assuming that such retained control environments are rare by comparison to the usual uses of the stack, one can usually treat most of the stack as a linear array, getting most of the efficiency of normal stack operations on conventional machines.

A more speculative approach for improving the performance of our interpreter is to optimize the use of the stack by exploiting the observation that the stack discipline has regularities which make many of the stack operations redundant. In the *caller-saves* convention (which is what the SCHEME-79 chip implements) the only reason why a register is pushed onto the stack is to protect its contents from being destroyed by the unpredictable uses of the register during the recursive evaluation of a subexpression. Therefore one source of redundant stack operations is that a register is saved even though the evaluation of the subexpression may not affect the contents of that register. If we could look ahead in time we could determine whether or not the register will retain its contents through the unknown evaluation. This is one standard kind of optimization done by compilers, but even a compiler cannot optimize all cases because the execution path of a program depends in general on the data being processed. However, instead of looking ahead, we can try to make the stack mechanism *lazy* in that it postpones pushing a register until its contents are about to be destroyed. The key idea is that each register has a state which indicates whether its contents are valuable. If such a valuable register is about to be assigned, it is at that moment pushed. In order to make this system work, each register which may be pushed has its own stack so that we can decouple the stack disciplines for each of the registers. Each register-stack combination can be thought of as having a state which encodes some of the history of previous operations. It is organized as a finite-state automaton which mediates between operation requests and the internal registers and stack. This automaton serves as an on-the-fly peephole optimizer, which recognizes certain patterns of operations within a small window in time and transforms them so as to reduce the actual number of stack operations performed. We have investigated this strategy [Dream] and we believe that it can be implemented in hardware easily and will substantially improve the performance of the algorithm. In pilot studies, we have determined that this technique can save 3 out of every 4 stack allocations in the operation of the interpreter.

Another way of building prescience into the machine language is to include S-code instructions (type fields) that encapsulate those special cases of argument evaluation or sequencing which are trivial and do not require that the interpreter save state. This would reduce the stack usage and also result in a more compact S-code and shorter S-code execution sequences. For example, evaluation of *trivial* expressions containing only variables, constants, or applications of primitive operators to trivial expressions does not require saving of state. But the augmented instruction set would considerably expand the size of the on-chip microcode.

Other tradeoffs were made in the architecture of the chip. For example, we chose to make our register array have a single bus. This forced the microcode to serialize many register transfer operations which logically could have been done in parallel. A more powerful bus structure would result in a significant speed up. We also decided to use simple register cells rather than buffered registers. This means that a register cannot be read and written at the same time. This is not usually a problem because on a single bus machine it is not useful to read out and load the same register. But it does cause increment and decrement operations to take two microcycles rather than one microcycle. This is significant because it is in the innermost loop of the local variable lookup routine. Decrementing the frame and displacement field take twice as long as is really necessary. Finally, we could have used more registers for intermediate values to be stored. In several cases, having an extra intermediate would result in fewer register shuffles and CONSes to get something done. For example, having special argument registers for holding the arguments for primitive one- and two-argument functions could make a serious dent in the storage allocated in argument evaluation and hence ultimately in time taken to garbage collect. This optimization may be combined with our stack optimizer strategy (mentioned above). In fact, the entire argument about stack optimization may be thought of as an architectural issue, because of the simple implementation of our peephole optimizing automata in hardware.

We also made significant tradeoffs in the electrical characteristics of the SCHEME-79 chip. The bus is not precharged. The PLAs are ratio logic, not switched. There is no on-chip clock generator. There are many long runs which were made on poly and diffusion which should have been made on metal. Some buffers are insufficient to effectively drive the long lines to which they are connected. We also used a selector design with implanted pass transistors which is an exceptionally slow circuit. We feel that careful redesign of some of the circuitry on the chip would greatly improve the performance.

We include the actual measured performance of our chip on a sample program. We calculated the values of Fibonacci numbers by the doubly recursive method. (This explodes exponentially; computing each Fibonacci number takes $\frac{1+\sqrt{5}}{2}$ times the time it takes to compute the previous one.) This is an excellent test program because it thoroughly exercises most of the mechanisms of the interpreter. Additionally, sums were computed using Peano arithmetic, which is the only arithmetic available on the chip. The program is as follows:

```
(DEFINE (+ X Y)
   (COND ((ZEROP X) Y)
         (T (+ (1- X) (1+ Y)))))
(DEFINE (FIB X)
  (COND ((ZEROP X) 0) ;If 0, result is 0.
        ((ZEROP (1- X)) 1) ;If 1, result is 1.
        (T (+ (FIB (1- X)) (FIB (1- (1- X)))))))
```

We computed `(fib 20.) = 6765.` with two different memory loadings, with a clock period of 1595 nanoseconds (not the top speed for the chip), and a memory of 32K LISP cells. If the memory was substantially empty (so that garbage collection was maximally efficient) the SCHEME-79 chip took about 1 minute to execute the program. With memory half full of live structure (a typical load for a LISP system) the SCHEME-79 chip took about 3 minutes.

We also compared this performance with that of our standard MACLISP interpreter on the same program running on the DEC KA10 and with a SCHEME interpreter written in MACLISP. LISP did not garbage collect during this operation, but it took about 3.6 minutes. The MACLISP SCHEME interpreter (with unknown memory loading) took about 9 minutes with about 10% of the time spent in the garbage collector.

Acknowledgments

We would like to thank Lynn Conway and Bert Sutherland of the Xerox Palo Alto Research Center System Science Laboratory, and Jon Allen and Paul Penfield of MIT, for help, support and encouragment of this project. We would also like to thank all those people who gave us ideas and technical assistance. Particularly, Neil Mayle, Thomas F. Knight, Howie Shrobe, and Richard Zippel at MIT, and Lynn Conway, Jim Cherry, Dick Lyon, and Gerry Roylance at XEROX PARC, who provided important moral and technical support during our design period. These people read and commented on designs, found bugs, and proposed patches. The resulting chip still had many bugs, which were found by sharp-eyed friends and by the simulation support provided by Clark Baker and Chris Terman of the MIT Laboratory for Computer Science. We would like to thank Howard Cannon for designing and constructing the apparatus for testing SCHEME-79 chips using our MIT LISP machines. Recently, we have had excellent suggestions for improving the next design from Richard Stallman and John Rees. Paul Penfield, Ron Rivest, Neil Mayle, John Batali, and Howie Shrobe have had good ideas about better design tools. We are sure that it would have been impossible to undertake so large a design without such extensive help provided by so many friends.

References

[650 DPS]

650 Data Processing System Bulletin G24-5000-0. International Business Machines Corporation (1958).

[Art]

Knuth, Donald E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley (Reading, Mass., 1968).

[Bristle Blocks]

Johannsen, Dave. "Bristle Blocks: A Silicon Compiler." Proc. Caltech Conference on VLSI, (January 1979).

[CDR Coding]

Hansen, Wilfred J. "Compact List Representation: Definition, Garbage Collection, and System Implementation." Comm. ACM 12, 9 (September 1969), 499-507.

[Concurrent]

Steele, Guy Lewis Jr. "Multiprocessing Compactifying Garbage Collection." Comm. ACM 18, 9 (September 1975), 495-508.

[Continuations]

Reynolds, John C. "Definitional Interpreters for Higher Order Programming Languages." ACM Conference Proceedings 1972.

[DSW]

Schorr, H. and Waite, W. M. "An efficient machine-independent procedure for garbage collection in various list structures." Comm. ACM 10, 8 (August 1967), 501-506.

[Debunking]

Steele, Guy Lewis Jr. "Debunking the 'Expensive Procedure Call' Myth." Proc. ACM National Conference (Seattle, October 1977), 153-162. Revised as MIT AI Memo 443 (Cambridge, October 1977).

[Declarative]

Steele, Guy Lewis Jr. *LAMBDA: The Ultimate Declarative.* AI Memo 379. MIT AI Lab (Cambridge, November 1976).

[Dream]

Steele, Guy Lewis Jr. and Sussman, Gerald Jay. *The Dream of a Lifetime: A Lazy Scoping Mechanism.* MIT AI Memo 527, Cambridge Massachusetts, (November 1979).

[ECL]
Conrad, William R. *Internal Representations of ECL Data Types.* Technical Report 5-75. Center for Research in Computing Technology, Harvard U. (Cambridge, March 1975).

[FLATS]
Goto, Eiichi; Ida, Tetsuo; Kei, Hiraki; Suzuki, Masayuki; and Inada Nobuyuki. "FLATS, A Machine for Numerical, Symbolic, and Associative Computing". Proc. Sixth Annual IEEE/ACM Symposium on Computer Architecture (April 1979), 102-110.

[ICARUS]
Fairbairn, Doug and Rowson, Jim. "ICARUS: An Interactive Integrated Circuit Layout Program". *Proceedings of the 15th Annual IEEE Design Automation Conference,* (June 1978).

[Imperative]
Steele, Guy Lewis Jr. and Sussman, Gerald Jay. *LAMBDA: The Ultimate Imperative.* AI Memo 353. MIT AI Lab (Cambridge, March 1976).

[Interpreters]
Steele, Guy Lewis Jr. and Sussman, Gerald Jay. *The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two).* MIT AI Memo 453 (Cambridge, May 1978).

[LISP 1.5]
McCarthy, John, et al. *LISP 1.5 Programmer's Manual.* The MIT Press (Cambridge, 1962).

[LISP Machine]
The LISP Machine Group: Bawden, Alan; Greenblatt, Richard; Holloway, Jack; Knight, Thomas; Moon, David; and Weinreb, Daniel. *LISP Machine Progress Report.* AI Memo 444. MIT AI Lab (Cambridge, August 1977).

[M68000]
Stritter, Skip and Tredennick, Nick. "Microprogrammed Implementation of a Single Chip Microprocessor". Proc. IEEE 11th Annual Microprogramming Workshop, (November 1978).

[MDL]
Galley, S.W. and Pfister, Greg. *The MDL Language.* Programming Technology Division Document SYS.11.01. Project MAC, MIT (Cambridge, November 1975).

[MOSSIM]
Bryant, Randy. *The MOSSIM User Manual.* Unpublished working paper, MIT Laboratory for Computer Science (1979).

[Multics]

Daley, R.C. and Neumann, P.G. "A General-Purpose File System for Secondary Storage". Proc. Fall Joint Computer Conference, Las Vegas Nevada, (November 1965).

[Nanoprogramming]
Grasselli, A. "The Design of Program-Modifiable Micro-Programmed Control Units", IRE Transactions on Electronic Computers, EC-11 no. 6, (June 1962).

[QM-1]
Frieder, G. and Miller, J. "An Analysis of Code Density for Two Level Programmable Control of the Nanodata QM-1", Proc. IEEE 10th Annual Microprogramming Workshop, (October 1975).

[RABBIT]
Steele, Guy Lewis Jr. *Compiler Optimization Based on Viewing LAMBDA as Rename plus Goto.* S.M. thesis. MIT (Cambridge, May 1977). Published as *RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization).* AI TR 474. MIT AI Lab (Cambridge, May 1978).

[Real Time]
Baker, Henry B., Jr. *List Processing in Real Time on a Serial Computer.* Comm. ACM 21, 4 (April 1978), 280-294.

[Revised Report]
Steele, Guy Lewis Jr. and Sussman, Gerald Jay. *The Revised Report on SCHEME: A Dialect of LISP.* MIT AI Memo 452 (Cambridge, January 1978).

[SCHEME Chip 0]
Steele, Guy Lewis Jr. and Sussman, Gerald Jay. "Storage Management in a LISP-Based Processor." Proc. Caltech Conference on Very Large Scale Integration (Pasadena, January 1979).

[SCHEME Chip 1]
Steele, Guy Lewis Jr. and Sussman, Gerald Jay. *Design of LISP-Based Processors; or, SCHEME: A Dielectric LISP; or, Finite Memories Considered Harmful; or, LAMBDA: The Ultimate Opcode.* AI memo 514. MIT AI Lab (Cambridge, March 1979).

[Shallow]
Baker, Henry B., Jr. *Shallow Binding in LISP 1.5.* Comm. ACM 21, 7 (July 1978), 565-569.

[Spaghetti]
Bobrow, Daniel G. and Wegbreit, Ben. "A Model and Stack Implementation of Multiple Environments." Comm. ACM 16, 10 (October 1973) pp. 591-603.

[VLSI Systems]
Mead, Carver A. and Conway, Lynn A. *Introduction to VLSI Systems.* Addison-Wesley Publishing Co., Reading Mass, (1980).

Appendix - The SCHEME-79 Microcode

In this appendix we present a complete listing of the microcode for the SCHEME-79 chip. In this listing, symbols surrounded by single asterisks (e.g. `*nil*`) are names or aliases of machine registers. Symbols beginning with ampersand (&) are the names of hardware operations. The S-code (macro) opcodes are (some of) the data types. These are given specific numerical values. When a DEFTYPE is used to define the microcode that is used when that type is executed, I usually place a comment showing how that data type is expected to be used in a piece of S-code. Some data types are pointer types (meaning that the garbage collector must mark the thing pointed at by the address part) and others are immediate data (hence terminal to the garbage collector). The folowing are the data types with preassigned numerical values:

```
(defschip **pointer-types**
      '((self-evaluating-pointer 0)
        (symbol 1)
        (global 2)
        (set-global 3)
        (conditional 4)
        (procedure 5)
        (first-argument 6)
        (next-argument 7)
        (last-argument 10)
        (apply-no-args 11)
        (apply-1-arg 12)
        (primitive-apply-1 13)
        (primitive-apply-2 14)
        (sequence 15)
        (spread-argument 16)
        (closure 17)
        (get-control-point 20)
        (control-point 21)
        (interrupt-point 22)
        (self-evaluating-pointer-1 23)
        (self-evaluating-pointer-2 24)
        (self-evaluating-pointer-3 25)
        (self-evaluating-pointer-4 26)
        ))

(defschip **non-pointer-types**
      '((self-evaluating-immediate 100)
        (local 101)
        (tail-local 102)
        (set-local 103)
        (set-tail-local 104)
        (set-only-tail-local 105)
        (primitive-car 106)           ;Built-in operators
        (primitive-cdr 107)
        (primitive-cons 110)
        (primitive-rplaca 111)
        (primitive-rplacd 112)
        (primitive-eq 113)
        (primitive-type? 114)
        (primitive-type! 115)
        (gc-special-type 116)         ;Never appears except during gc
(self-evaluating-immediate-1 117)
(self-evaluating-immediate-2 120)
(self-evaluating-immediate-3 121)
(self-evaluating-immediate-4 122)
(mark 123)
(done 124)
(primitive-add1 125)
(primitive-sub1 126)
(primitive-zerop 127)
(primitive-displacement-add1 130)
(primitive-not-atom 131)
(boot-load 776)                  ;micro-address forced by RESET
(process-interrupt 777)          ;micro-address forced by EXT INT RQ
))
```

Pointer data is indicated by the 100 bit being off in the data type.

```
(defschip **pointer** 100)                ;100 bit means non-pointer.
```

Next we find the definitions of the registers. The following registers are used by EVAL: `*val*`, `*exp*`, `*args*`, `*display*`, `*stack*`. In addition, `*newcell*` contains the pointer to the beginning of free storage. It is changed by CONSing or saving something on the stack. Whenever `*newcell*` is changed, it is compared with `*memtop*` which contains the (user set) memory limit. When these are equal an interrupt is signalled, setting the one bit register `*gc-needed*` – indicating need for garbage-collection. `*Nil*` is a dummy register, it cannot be set, and its value is nil. The garbage collector uses the following registers: `*stack-top*`, `*node-pointer*`, `*leader*` are used by the GC mark phase only. `*Scan-up*` `*scan-down*` and `*memtop*` are used by the GC sweep phase. `*Rel-tem-1*`, `*rel-tem-2*` are temporaries used in GC relocate phase only. `*Intermediate-argument*` is used by microcode compiler for storing anonymous temporaries and `*retpc-count-mark*` is used in increment operations to store intermediate values because our registers are not dual rank. It is also used for storing microcode return addresses for use in microcode subroutines. There is overlap in the use of the registers, for example, `*scan-up*` is the `*newcell*` pointer when in EVAL. Thus the above names are really aliases for the real underlying physical registers. The mapping is made below:

```
(defschip **machine-registers**
      '((*nil*)
        (*memtop*)
        (*newcell*)
        (*scan-up* *newcell*)
        (*exp*)
        (*scan-down* *exp*)
        (*val*)
        (*rel-tem-2* *val*)
        (*stack-top* *val*)
        (*args*)
        (*leader* *args*)
        (*rel-tem-1* *args*)
        (*display*)
        (*node-pointer* *display*)
        (*stack*)
        (*retpc-count-mark*)
        (*intermediate-argument*)))
```

Each physical register has certain capabilities which are controlled by particular control wires which go into it. Thus the `to-displacement` wire on the `*exp*` register will, if raised, allow the displacement field of the register to be set from the corresponding field of the bus. The registers also develop certain conditions which are reflected by the states of sense wires. So, for example, the `type=bus` wire coming out of the `*val*` register indicates if the type field of the register is equal to the corresponding field on the bus. The following expressions define the control lines and sense wires on the registers.

```
(defreg *exp* (to-type to-displacement to-frame from from-decremented) ())

(defreg *newcell* (to-type to-address from from-incremented) (address=bus))

(defreg *val*
        (to-type to-address from)
        (type=bus address=bus =bus))     ;=bus is AND of type, address=bus

(defreg *retpc-count-mark* (to-type to-address from) ())

(defreg *stack* (to-type to-address from) ())

(defreg *memtop* (to from) ())

(defreg *args* (to from) ())

(defreg *display* (to from) ())

(defreg *intermediate-argument* (to from) ())

(defreg *nil* (from) ())
```

Additionally, the bus is sensitive to several conditions:

```
(defreg bus
        () (mark-bit type-not-pointer frame=0 displacement=0 address=0))
```

A register has two basic operations which can be done to it. Its contents can be `fetch`ed, and its contents can be `assign`ed from some source. In addition, for the LISP simulator we define two additional operations which are used for stacks:

```
(defmacro save (quantity)
          (assign *stack* (&cons ,quantity (fetch *stack*))))

(defmacro restore (register)
          (progn (assign ,register (&car (fetch *stack*)))
                 (assign *stack* (&cdr (fetch *stack*)))))
```

At this point we begin to look at the microcode proper. `Boot-load` is the place where the chip is initialized to run. The first thing it does is initialize the memory limit register and then it picks up (as an interrupt address) a pointer to the expression to begin executing. It stashes this away in the stack and goes off to MARK to get memory organized and set up a reasonable value for `*scan-up*` (`*newcell*`). The (micro) return address is stored as the type of the stack pointer. Thus all micro return addresses must be pointer types – something the compiler must know about!

```
(deftype boot-load
         (assign *scan-up* (fetch *nil*))
         (&increment-scan-up)                              ;to location 1
         (assign *memtop* (&car (fetch *scan-up*)))
         (assign *scan-up* (fetch *memtop*))
         (assign *stack* (&get-interrupt-routine-pointer))     ;from pads
         (&set-type *stack* boot-load-return)
         (go-to mark))

(defreturn boot-load-return
         (assign *exp* (&car (fetch *stack*)))
         (assign *stack* (fetch *nil*))
         (&set-type *stack* done)
         (dispatch-on-exp-allowing-interrupts))
```

When there is nothing more to do the machine halts.

```
(deftype done
         (go-to done))
```

The next section of the microcode is the SCHEME-79 chip storage allocator and garbage collector. `Mark` is the garbage-collector entry point. It is a user function with no arguments which returns NIL when it is done. We use the Deutsch-Schorr-Waite mark algorithm. There are 3 registers containing pointer data: `*stack-top*`, `*node-pointer*`, `*leader*`. A datum may be a pointer or a terminal datum; this may be tested by `&pointer?` Objects have 2 data parts which can fit a pointer -- the CAR and CDR. These are accessed by `&car` and `&cdr` functions of the pointer to the object. They are clobbered by `&rplaca` and `&rplacd` functions of a pointer to the object and the replacement datum. Objects also have two mark bits, the `in-use` and `car-trace-in-progress` bit. The `in-use` bit is stored in the CAR of the node and the `car-trace-in-progress` bit is stored in the CDR of the node. They are accessed by `&in-use?` and `&car-being-traced?` of a pointer to the object. They are set and cleared by `&mark-in-use!`, `&mark-car-being-traced!`, `&mark-car-trace-over!` and `&unmark!` of the pointer to the object. In addition, any `&rplaca` or `&rplacd` operation will clear the associated mark bit. This requires the introduction of `&rplaca-and-mark!` to change the CAR pointer while setting the `in-use` bit.

```
(deftype mark                             ;MARK(?)
       (&rplaca (fetch *nil*) (fetch *stack*))
       (assign *node-pointer* (fetch *nil*))
       (assign *stack-top* (fetch *nil*))
       (&set-type *stack-top* gc-special-type)
       (go-to mark-node))

(defpc mark-node
       (assign *leader* (&car (fetch *node-pointer*)))
       (cond ((and (&pointer? (fetch *leader*))
                   (not (&in-use? (fetch *leader*))))
              (&mark-car-being-traced! (fetch *node-pointer*))
              (&rplaca-and-mark! (fetch *node-pointer*) (fetch *stack-top*))
              (go-to down-trace))
             (t
              (&mark-in-use! (fetch *node-pointer*))
              (go-to trace-cdr))))
```

```
(defpc down-trace
       (assign *stack-top* (fetch *node-pointer*))
       (assign *node-pointer* (fetch *leader*))
       (go-to mark-node))

(defpc trace-cdr
       (assign *leader* (&cdr (fetch *node-pointer*)))
       (cond ((and (&pointer? (fetch *leader*)) (not (&in-use? (fetch *leader*))))
              (&rplacd (fetch *node-pointer*) (fetch *stack-top*))
              (go-to down-trace))
             (t (go-to up-trace))))

(defpc up-trace
       (cond ((&=type? (fetch *stack-top*) gc-special-type)
              (go-to sweep))
             (t (assign *leader* (fetch *stack-top*))
                (cond ((&car-being-traced? (fetch *leader*))
                       (&mark-car-trace-over! (fetch *leader*))
                       (assign *stack-top* (&car (fetch *leader*)))
                       (&rplaca-and-mark! (fetch *leader*)
                                          (fetch *node-pointer*))
                       (assign *node-pointer* (fetch *leader*))
                       (go-to trace-cdr))
                      (t (assign *stack-top* (&cdr (fetch *leader*)))
                         (&rplacd (fetch *leader*) (fetch *node-pointer*))
                         (assign *node-pointer* (fetch *leader*))
                         (go-to up-trace))))))
```

The sweep algorithm for this garbage collector is the simple *two finger compacting* method. The two "fingers" are: `*scan-up*` and `*scan-down*`. Remember, `*scan-up*` is the newcell register for cons. Thus, because `mark` does not disturb it, initially, `*scan-up*` points at the last successfully completed cons.

```
(defpc sweep
       (&increment-scan-up)
       (assign *scan-down* (fetch *scan-up*))          ;initialization
       (assign *scan-up* (fetch *nil*))                ;make address = 0
       (&set-type *scan-up* gc-special-type)
       (&clear-gc-needed)
       (go-to scan-down-for-thing))

(defpc scan-down-for-thing
       (&decrement-scan-down)
       (cond ((&scan-up=scan-down?) (go-to relocate-pointers))
             ((&in-use? (fetch *scan-down*)) (go-to scan-up-for-hole))
             (t (go-to scan-down-for-thing))))

(defpc scan-up-for-hole
       (cond ((&in-use? (fetch *scan-up*))
              (&increment-scan-up)
              (cond ((&scan-up=scan-down?) (go-to relocate-pointers))
                    (t (go-to scan-up-for-hole))))
             (t (go-to swap-thing-and-hole))))
```

The following code is rather tricky. The last `rplaca` operation performs several important operations at once. Since the type of `*scan-up*` is `gc-special-type`, the cell pointed at by `*scan-down*` (which is above the eventual `*scan-up*` and thus will be free storage) is marked as a "broken heart" pointing at where its contents has gone. This will be looked at later by the relocation phase. This free-cell-to-be is also unmarked by this operation.

```
(defpc swap-thing-and-hole
       (&rplaca-and-mark! (fetch *scan-up*) (&car (fetch *scan-down*)))
       (&rplacd (fetch *scan-up*) (&cdr (fetch *scan-down*)))
       (&rplaca (fetch *scan-down*) (fetch *scan-up*))
       (go-to scan-down-for-thing))
```

The relocation phase now adjusts all live pointers which point at object which have been moved, leaving behind broken hearts. At the entry to `relocate-pointers`, `*scan-up* = *scan-down*` and they point at the highest occupied location in memory. `*Scan-up*` is left there to become the future `*newcell*` and `*scan-down*` is used to count down until we get to the bottom of memory.

```
(defpc relocate-pointers
       (assign *rel-tem-1* (&car (fetch *scan-down*)))
       (cond ((&pointer? (fetch *rel-tem-1*))
              (assign *rel-tem-2* (&car (fetch *rel-tem-1*)))
              (cond ((&=type? (fetch *rel-tem-2*) gc-special-type)
                     (&set-type *rel-tem-2* (fetch *rel-tem-1*))
                     (&rplaca (fetch *scan-down*) (fetch *rel-tem-2*))))))
       (assign *rel-tem-1* (&cdr (fetch *scan-down*)))
       (cond ((&pointer? (fetch *rel-tem-1*))
              (assign *rel-tem-2* (&car (fetch *rel-tem-1*)))
              (cond ((&=type? (fetch *rel-tem-2*) gc-special-type)
                     (&set-type *rel-tem-2* (fetch *rel-tem-1*))
                     (&rplacd (fetch *scan-down*) (fetch *rel-tem-2*))))))
       (&unmark! (fetch *scan-down*))
       (cond ((&scan-down=0?)
              (&set-type *scan-up* self-evaluating-pointer)
              (assign *stack* (&car (fetch *nil*)))     ;might have been relocated
              (&rplaca (fetch *nil*) (fetch *nil*))
              (assign *val* (fetch *nil*))
              (dispatch-on-stack))
             (t (&decrement-scan-down)
                (go-to relocate-pointers))))
```

Congratulations, you have just survived the garbage collector! We now proceed to examine the evaluator proper. The first part of the evaluator is the stuff for dealing with variable references. The opcodes which take a lexical-address as their data field decode that field into a frame number and a displacement number in the `*exp*` register. Lexical access of local variables uses `lookup-exp` to get a locative to the value. The CAR of the locative is the value cell for that variable. `Micro-call` is a microcode macro operation which stashes the (micro code) return address specified by its second argument in the type field of `*retpc-count-mark*` and then goes to the micro-code address specified by its first argument. `Micro-return` is used to dispatch on this saved type field.

```
(deftype local                    ;LOCAL(lexical-address)
         (micro-call lookup-exp local-return))
```

```
(defpc local-return
       (assign *val* (&car (fetch *display*)))
       (dispatch-on-stack))
```

Tail local variables give SCHEME an *LSUBR* option. That is, a procedure may be passed the list of evaluated arguments as the value of a variable rather than having an explicit local variable for each value passed. For example: in `((lambda x (foo x)) 1 2 3)`, x is a tail variable which is bound to the list `(1 2 3)`. additionally, this is extended to give the power of *rest* variables as follows: in `(lambda (x y . z) ---)` x and y are bound to the first two arguments while z is a tail variable which is bound to the remaining arguments.

```
(deftype tail-local              ;TAIL-LOCAL(lexical-address)
        (micro-call lookup-exp tail-local-return))

(defpc tail-local-return
       (assign *val* (fetch *display*))
       (dispatch-on-stack))
```

Global variables are stored in the value cell of a symbol. The value cell is assumed to be in the CAR of the symbol. The CDR may be used for other purposes (such as property lists). Thus the `global` type may be thought of as an alias for CAR.

```
(deftype global                  ;GLOBAL(symbol)
       (assign *val*             ;global-value=&car
               (&global-value (fetch *exp*)))
       (dispatch-on-stack))
```

The following stuff is for assignment to variables. These types are to be used as part of a `sequence` whose provious entry develops a value in `*val*` which will be the value stuffed into the variable's value locative.

```
(deftype set-local               ;SET-LOCAL(lexical-address)
        (micro-call lookup-exp set-local-return))

(defpc set-local-return
       (&rplaca (fetch *display*) (fetch *val*))
       (dispatch-on-stack))

(deftype set-tail-local          ;SET-TAIL-LOCAL(lexical-address)
        (micro-call lookup-exp set-tail-local-return))

(defpc set-tail-local-return
       (&rplacd (fetch *display*) (fetch *val*))
       (dispatch-on-stack))
```

The following is a tricky little critter. It is needed because if we have a tail only variable (e.g. `(lambda x ---)`) we need to be able to get at the header of the sublist of the display referred to by the tail variable.

```
(deftype set-only-tail-local     ;SET-ONLY-TAIL-LOCAL(lexical-address)
        (if (&frame=0?)
            (progn (&rplaca (fetch *display*) (fetch *val*))
                   (dispatch-on-stack))
            (progn (assign *display* (&cdr (fetch *display*)))
                   (&decrement-frame)
                   (go-to set-only-tail-local))))
```

```
;&set-global-value = &rplaca
(deftype set-global                    ;SET-GLOBAL(symbol)
         (&set-global-value (fetch *exp*) (fetch *val*))
         (dispatch-on-stack))

(defpc lookup-exp
       (if (&frame=0?)
           (progn (assign *display* (&car (fetch *display*)))
                  (go-to count-displacement))
           (progn (&decrement-frame)
                  (assign *display* (&cdr (fetch *display*)))
                  (go-to lookup-exp))))

(defpc count-displacement
       (if (&displacement=0?)
           (micro-return)
           (progn (&decrement-displacement)
                  (assign *display* (&cdr (fetch *display*)))
                  (go-to count-displacement))))
```

Next come all of the various types of self-evaluating data. There are two different classes -- pointer data and immediate data. A `symbol` is pointer data. We provide several unspecified varieties of such self-evaluating data for the user to assign to things like fixed numbers and floating numbers.

```
(deftype self-evaluating-immediate      ;SELF-EVALUATING-IMMEDIATE(frob)
         (assign *val* (fetch *exp*))
         (dispatch-on-stack))

(deftype self-evaluating-immediate-1    ;SELF-EVALUATING-IMMEDIATE-1(frob)
         (assign *val* (fetch *exp*))
         (dispatch-on-stack))

(deftype self-evaluating-immediate-2    ;SELF-EVALUATING-IMMEDIATE-2(frob)
         (assign *val* (fetch *exp*))
         (dispatch-on-stack))

(deftype self-evaluating-immediate-3    ;SELF-EVALUATING-IMMEDIATE-3(frob)
         (assign *val* (fetch *exp*))
         (dispatch-on-stack))

(deftype self-evaluating-immediate-4    ;SELF-EVALUATING-IMMEDIATE-4(frob)
         (assign *val* (fetch *exp*))
         (dispatch-on-stack))

(deftype symbol                         ;SYMBOL(frob)
         (assign *val* (fetch *exp*))
         (dispatch-on-stack))

(deftype self-evaluating-pointer        ;SELF-EVALUATING-POINTER(frob)
         (assign *val* (fetch *exp*))
         (dispatch-on-stack))

(deftype self-evaluating-pointer-1      ;SELF-EVALUATING-POINTER-1(frob)
         (assign *val* (fetch *exp*))
         (dispatch-on-stack))
```

```
(deftype self-evaluating-pointer-2        ;SELF-EVALUATING-POINTER-2⟨frob⟩
        (assign *val* (fetch *exp*))
        (dispatch-on-stack))

(deftype self-evaluating-pointer-3        ;SELF-EVALUATING-POINTER-3⟨frob⟩
        (assign *val* (fetch *exp*))
        (dispatch-on-stack))

(deftype self-evaluating-pointer-4        ;SELF-EVALUATING-POINTER-4⟨frob⟩
        (assign *val* (fetch *exp*))
        (dispatch-on-stack))
```

A `lambda` expression in the original SCHEME code turns into a `procedure` in the S-code. When executed, a `procedure` constructs and returns a `closure`. `Procedures` may be documented with a description of the original variable names and the context they were compiled in (perhaps even a direct pointer to the source code) thus providing for debugging tools.

```
(deftype procedure                ;PROCEDURE⟨(script . documentation)⟩
        (assign *val* (&cons (fetch *exp*) (fetch *display*)))
        (&set-type *val* closure)
        (dispatch-on-stack))
```

An `if` expression in the SCHEME source code turns into a `sequence` which evaluates the predicate part of the `if` and then falls into a `conditional` to choose between the consequent and alternative expressions on the basis of the value of the `*val*` register.

```
(deftype conditional              ;CONDITIONAL⟨(consequent . alternative)⟩
        (if (&eq-val (fetch *nil*))
            (assign *exp* (&cdr (fetch *exp*)))
            (assign *exp* (&car (fetch *exp*))))
        (dispatch-on-exp-allowing-interrupts))
```

The following macro definition defines a common sequence in the rest of the microcode. This sequence will be the standard way to attack a compound expression. The (micro) return address is stashed in `*retpc-count-mark*` so that it can be used as the type of a stack cell. The top of the stack had better be `standard-return` which knows how to undo this mess.

```
(defmicromacro save-cdr-and-eval-car (return-tag)
          (progn (&set-type *retpc-count-mark* ,return-tag)
                 (go-to standard-eval)))

(defpc standard-eval
      (save (fetch *display*))
      (&set-type *stack* (fetch *retpc-count-mark*))
      (save (&cdr (fetch *exp*)))
      (&set-type *stack* standard-return)
      (assign *exp* (&car (fetch *exp*)))
      (dispatch-on-exp-allowing-interrupts))
```

```
(defreturn standard-return
     (restore *exp*)
     (assign *retpc-count-mark* (fetch *stack*))
     (restore *display*)
     (dispatch (fetch *retpc-count-mark*)))
```

The sequence construct is very important in the S-code language. Not only is it used to implement PROGN but also, it is used to develop values in the `*val*` register to be used by later parts of the sequence such as conditionals or variable assigners.

```
(deftype sequence                        ;SEQUENCE((expression . rest))
        (assign *val* (fetch *nil*))    ;for gc
        (save-cdr-and-eval-car sequence-return))

(defreturn sequence-return
         (dispatch-on-exp-allowing-interrupts))
```

Control points are used to implement the general "catch tags" used in constructing non-standard control structures. It is useful for error exits, and multiprocess sorts of work. It is only to be used with extreme caution since it is easy to screw oneself with constructs such as this which violate the expression structure of the language.

```
(deftype get-control-point      ;GET-CONTROL-POINT((variable-setter . rest))
        (assign *val* (&cons (fetch *stack*) (fetch *nil*)))
        (&set-type *val* control-point)
        (save-cdr-and-eval-car sequence-return))
```

To evaluate a form with more than one argument one starts with a pointer of type `first-argument` which is used to initialize the `*args*` register which will be used to accumulate the arguments. The evaluation of the first argument is to be continued with an evaluation of each successive next argument until the last argument is encountered which should fall into the execution of the body of the procedure being called.

```
(deftype first-argument                  ;FIRST-ARGUMENT((arg1 . rest))
        (save-cdr-and-eval-car first-argument-return))

(defreturn first-argument-return
          (assign *args* (&cons (fetch *val*) (fetch *nil*)))
          (save (fetch *args*))
          (dispatch-on-exp-allowing-interrupts))
```

Next argument just accumulates the value of an argument and continues the evaluation of the form.

```
(deftype next-argument                   ;NEXT-ARGUMENT((arg . rest))
        (save (fetch *args*))
        (save-cdr-and-eval-car next-argument-return))

(defreturn next-argument-return
          (restore *args*)
          (&rplacd (fetch *args*) (&cons (fetch *val*) (fetch *nil*)))
          (assign *args* (&cdr (fetch *args*)))
          (dispatch-on-exp-allowing-interrupts))
```

Finally we get to the evaluation of the last argument. At this time the continuation is an expression which should evaluate to a closure which is to be applied.

```
(deftype last-argument                   ;LAST-ARGUMENT((arg . fun))
         (save (fetch *args*))
         (save-cdr-and-eval-car last-argument-return))

(defreturn last-argument-return
           (restore *args*)
           (&rplacd (fetch *args*)
                    (&cons (fetch *val*) (fetch *nil*)))
           (eval-exp-popj-to internal-apply))    ;Amazing! Where did retpc go?
```

Procedures with zero or one argument are handled specially for efficiency reasons.

```
(deftype apply-1-arg                     ;APPLY-1-ARG((arg . fn))
         (save-cdr-and-eval-car apply-1-arg-return))

(defreturn apply-1-arg-return
           (assign *args* (&cons (fetch *val*) (fetch *nil*)))
           (save (fetch *args*))
           (eval-exp-popj-to internal-apply))

(deftype apply-no-args                   ;APPLY-NO-ARGS((fn . ?))
         (assign *exp* (&car (fetch *exp*)))
         (save (fetch *nil*))            ;ugh! need a place for retpc.
         (eval-exp-popj-to internal-apply))
```

Spread argument is `apply`. It evaluates an argument, takes it as the set of arguments to be passed to the procedure specified by the continuation.

```
(deftype spread-argument                 ;SPREAD-ARGUMENT((arg . fun))
         (save-cdr-and-eval-car spread-argument-return))

(defreturn spread-argument-return
           (save (fetch *val*))
           (eval-exp-popj-to internal-apply))

(defreturn internal-apply                ;function is in *val*
           (restore *args*)
           (assign *exp* (fetch *val*))
           (dispatch-on-exp-allowing-interrupts))
```

Every user procedure is a `closure`. The closures are produced by evaluating `procedures`. A closure has a script which is the body of the procedure to be executed and a display which is the environment which the closure was manufactured in. Notice that there two CAR operations required to get the actual body of the procedure. This is necessary to bypass the documentation packaged in the procedure definition.

```
(deftype closure                         ;CLOSURE((script . display))
         (assign *display*
                 (&cons (fetch *args*) (&cdr (fetch *exp*))))
         (assign *exp* (&car (&car (fetch *exp*))))
         (dispatch-on-exp-allowing-interrupts))
```

When a control point (non-standard continuation) is executed it is interpreted as a procedure with one argument which returns that argument to the constructor of the control point.

```
(deftype control-point                          ;CONTROL-POINT(state)
         (assign *val* (&car (fetch *args*)))
         (assign *stack* (&car (fetch *exp*)))
         (dispatch-on-stack))
```

An interrupt point is similar to a control point except that the `*display*` and `*val*` registers must be restored.

```
(deftype interrupt-point                        ;INTERRUPT-POINT(state)
         (assign *stack* (fetch *exp*))
         (restore *val*)
         (restore *display*)
         (go-to restore-exp-args-dispatch))

(deftype primitive-apply-1                      ;PRIMITIVE-APPLY-1((arg . primop))
         (save (&cdr (fetch *exp*)))
         (assign *exp* (&car (fetch *exp*)))
         (eval-exp-popj-to primitive-apply-1-return))

(defreturn primitive-apply-1-return
           (restore *exp*)
           (dispatch-on-exp-allowing-interrupts))
```

The primitive operators included on the SCHEME-79 chip are implemented in the following microcode.

```
(deftype primitive-car                          ;PRIMITIVE-CAR(?)
         (assign *val* (&car (fetch *val*)))
         (dispatch-on-stack))

(deftype primitive-cdr                          ;PRIMITIVE-CDR(?)
         (assign *val* (&cdr (fetch *val*)))
         (dispatch-on-stack))

(deftype primitive-type?                        ;PRIMITIVE-TYPE?(?)
         (assign *exp* (fetch *val*))
         (assign *val* (fetch *nil*))
         (&set-type *val* (fetch *exp*))          ;build prototype.
         (dispatch-on-stack))

                                              ;PRIMITIVE-NOT-ATOM(?)
(deftype primitive-not-atom
         (if (&pointer? (fetch *val*))
             (progn (assign *val* (fetch *nil*))
                    (&set-type *val* self-evaluating-immediate))          ;T
             (assign *val* (fetch *nil*)))
         (dispatch-on-stack))

(deftype primitive-zerop                        ;PRIMITIVE-ZEROP(?)
         (if (&val=0?)
             (progn (assign *val* (fetch *nil*))
                    (&set-type *val* self-evaluating-immediate))          ;T
             (assign *val* (fetch *nil*)))
         (dispatch-on-stack))

(deftype primitive-sub1                         ;PRIMITIVE-SUB1(?)
         (assign *scan-down* (fetch *val*))
         (&decrement-scan-down-to-val)
         (dispatch-on-stack))
```

```
(deftype primitive-add1                        ;PRIMITIVE-ADD1(?)
         (assign *exp* (fetch *scan-up*))
         (assign *scan-up* (fetch *val*))
         (&increment-scan-up-to-val)
         (assign *scan-up* (fetch *exp*))
         (dispatch-on-stack))

(deftype primitive-displacement-add1           ;PRIMITIVE-DISPLACEMENT-ADD1(?)
         (assign *exp* (fetch *nil*))
         (&decrement-frame)                    ;make -1 in frame part
         (&val-displacement-to-exp-displacement)
         (assign *args* (fetch *scan-up*))
         (assign *scan-up* (fetch *exp*))
         (&increment-scan-up)
         (assign *exp* (fetch *scan-up*))
         (&val-frame-to-exp-frame)
         (assign *val* (fetch *exp*))
         (dispatch-on-stack))
```

Thus cons = list*.

```
(deftype primitive-apply-2                     ;PRIMITIVE-APPLY-2((arg . primop))
         (save (fetch *args*))
         (save (&cdr (fetch *exp*)))
         (assign *exp* (&car (fetch *exp*)))
         (eval-exp-popj-to restore-exp-args-dispatch))

(defreturn restore-exp-args-dispatch
           (restore *exp*)
           (restore *args*)
           (dispatch-on-exp-allowing-interrupts))

(deftype primitive-cons                        ;PRIMITIVE-CONS(?)
         (&rplacd (fetch *args*) (fetch *val*))
         (restore *val*)
         (dispatch-on-stack))

(deftype primitive-eq                          ;PRIMITIVE-EQ(?)
         (restore *args*)
         (assign *args* (&car (fetch *args*)))
         (if (&eq-val (fetch *args*))
             (progn (assign *val* (fetch *nil*))
                    (&set-type *val* self-evaluating-immediate))        ;T
             (assign *val* (fetch *nil*)))
         (dispatch-on-stack))

(deftype primitive-rplaca                      ;PRIMITIVE-RPLACA(?)
         (restore *args*)
         (assign *val* (&rplaca (&car (fetch *args*)) (fetch *val*)))
         (dispatch-on-stack))

(deftype primitive-rplacd                      ;PRIMITIVE-RPLACD(?)
         (restore *args*)
         (assign *val* (&rplacd (&car (fetch *args*)) (fetch *val*)))
         (dispatch-on-stack))
```

```
(deftype primitive-type!                    ;PRIMITIVE-TYPE!(?)
         (restore *args*)
         (assign *exp* *val*)
         (assign *val* (&car (fetch *args*)))
         (&set-type *val* (fetch *exp*))
         (dispatch-on-stack))
```

When an interrupt is requested and the machine is allowing interrupts, the microcode continues from the following place:

```
(deftype process-interrupt
        (save (fetch *args*))
        (save (fetch *exp*))
        (save (fetch *display*))
        (save (fetch *val*))
        (&set-type *stack* interrupt-point)
        (assign *args* (fetch *stack*))
        (assign *exp* (&car (&get-interrupt-routine-pointer)))    ;from pads
        (dispatch-on-exp-allowing-interrupts))
```

The following routines are used to let the user get at the internal storage allocator registers (GOD help him!).

```
(defpc get-memtop
        (assign *val* (fetch *memtop*))
        (dispatch-on-stack))

(defpc set-memtop
        (assign *memtop* (fetch *val*))
        (dispatch-on-stack))

(defpc get-scan-up
        (assign *val* (fetch *scan-up*))
        (dispatch-on-stack))

(defpc set-scan-up
        (assign *scan-up* (fetch *val*))
        (&set-type *scan-up* self-evaluating-pointer)
        (dispatch-on-stack))
```

The following code is put in for debugging purposes. By setting the state of the machine to the appropriate value we can read out or set any of the internal machine registers.

```
(defpc debug-routine
        (&write-to-pads (fetch *exp*))
        (&write-to-pads (fetch *val*))
        (&write-to-pads (fetch *args*))
        (&write-to-pads (fetch *display*))
        (&write-to-pads (fetch *stack*))
        (&write-to-pads (fetch *newcell*))
        (&write-to-pads (fetch *memtop*))
        (&write-to-pads (fetch *retpc-count-mark*))
        (&write-to-pads (fetch *intermediate-argument*))
        (&read-from-pads *exp*)
        (&read-from-pads *val*)
        (&read-from-pads *args*)
        (&read-from-pads *display*)
(&read-from-pads *stack*)
(&read-from-pads *newcell*)
(&read-from-pads *memtop*)
(&read-from-pads *retpc-count-mark*)
(&read-from-pads *intermediate-argument*)
(go-to debug-routine))
```