Scheme86 – An Architecture for Microcoding a Scheme Interpreter

Henry Manyan Wu

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements
for the Degree of Bachelor of Science

at the

Massachusetts Institute of Technology

May 9, 1986

Signature of Author____

Department of Electrical Engineering and Computer Science
May 9, 1986

Certified by____

Professor Gerald Jay Sussman
Thesis Supervisor

Accepted by____

Chairman, Departmental Committee

Scheme86 – An Architecture for Microcoding a Scheme Interpreter

Henry Manyan Wu

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements
for the Degree of Bachelor of Science

Abstract

I have designed and am currently building a computer that is optimized for a microcoded interpreter for Scheme. The native language of this computer is SCode, a tree-structured, typed-pointer representation of Scheme. The memory system is built with high speed RAM and offers low latency as well as high throughput. Multiple execution units in the processor make it possible to finish most complex operations in less than one memory cycle, thus allowing efficient use of memory bandwidth. The processor provides hardware support for tagged data objects and runtime type checking. This computer is designed as a back-end processor to a Hewlett Packard workstation which will handle bootstrapping and I/O operations. I will discuss the motivation for such a machine, its architecture, why it is expected to interpret Scheme efficiently, and the computer aided design tools I have developed for building this computer.

1. Introduction

Scheme86 is an architecture designed to execute Scheme programs efficiently. Unlike many modern computer designs which stress the use of simplified instructions with clever compilation techniques to improve performance, Scheme86 executes a powerful instruction set called SCode. SCode is a typed-pointer, tree-structured representation of Scheme. Scheme86 uses microcode with a relatively wide instruction to interpret SCode, and hence Scheme. Scheme86 is currently being implemented as a back-end processor using Advanced Schottky TTL technology. A Hewlett Packard workstation will be used as a front-end device for handling I/O operations and loading the microcode.

I will discuss the problems addressed by this machine, together with its architecture, the reasons why it is expected to interpret Scheme efficiently, and the computer-aided design tools implemented for completing the project.

1.1 Background

Optimizing the execution of a high level language at the architectural level is not a new idea. As early as the 60's Burroughs produced the B6500, a stack machine aimed at running Algol. The CADR processors of the LISP Machine system use an architecture tailored to running LISP code [Knight 1979]. Machines with the same architecture are still being produced by both the LISP Machine Incorporated and Texas Instruments. The Symbolics 3600 descended from the CADR and has become the industry's standard in the performance of LISP workstations [Moon 1985, Symbolics 1983]. The SPUR project at the University of California at Berkeley is experimenting with a new class of LISP engines based on simplified instructions [Katz 1985].

In 1979 and 1982 two attempts were made to implement an SCode interpreter with microcode on a microprocessor. Dubbed the SCHEME-79 and SCHEME-81 chips, these VLSI projects produced evidence that a properly designed architecture driven by microcode that inteprets SCode can run Scheme as efficiently as implementations running on conventional hardware [Steele 79, Holloway 80, Batali 82]. However, both chips were plagued by fabrication problems and limitations of VLSI technology at that time. A complete Scheme system never ran on any of them and extensive performance testing was not completed.

1.2 The Scheme86 Architecture

Similar to the SCHEME chips, Scheme86 is primarily an SCode interpreter implemented through the use of a microcoded finite state machine directly in hardware. This particular implementation of the architectural ideas uses TTL technology so that the resulting prototype will be easily debuggable. The amount of parallelism stressed by Scheme86's design would also make its implementation difficult even with today's VLSI technology.

SCode is a typed-pointer variant of Scheme. Each SCode instruction mirrors a Scheme special form. Using a powerful instruction set such as SCode seems to run against the common notion that implementing complicated instructions slows down the speed of a computer. In fact some SCode instructions and Scheme primitives specify enough computation so that most of the time Scheme86 is executing simple and uniform microcode instructions hand-optimized to take advantage of the architecture.

Scheme86 is designed mindful of the fact that interpreting SCode is a memory intensive process, and that executing LISP programs often involves chasing down several levels of indirect references during operand fetching. A high speed main memory system, to be built entirely using static RAM chips, is therefore chosen. Hardware support in the processor is added in a parallel fashion to allow concurrent operations. Instead of having one complicated execution unit as many machines with a complex instruction set do, Scheme86 has four relatively simple execution units that can be made very fast. These execution units ensure that there will be few wasted memory cycles. In addition, Scheme86 is designed so that data returning from a memory request can be used for the next memory reference as quickly as possible.

To summarize, Scheme86 can best be described as a machine with an extremely powerful instruction set, several parallel execution units and a high bandwidth main memory system optimized to execute LISP efficiently.

2. Principles of the Architecture

The Scheme86 processor is an attempt to arrive at an architecture that, given restrictions in technology, will execute Scheme as fast as possible. Not only must individual Scheme expressions run fast, the kind of programs that lend themselves to implementation in Scheme must run efficiently as well.

In Scheme86 user programs are stored as SCode items. SCode is to Scheme86 as the traditional machine language is to a more traditional processor. An SCode interpreter resides in the native control store of the processor. We call this the microcode of the machine. Microcode instructions are interpreted directly by the hardware; each microinstruction bit controls a physical control wire in the data paths. I will use the term "machine language" to mean the level of instructions that a software programmer is likely to use. The "microcode" is the program that interprets those instructions. Some machines use machine language instructions to manipulate hardware directly. Some machines interpret their microcode with another program called the nanocode.

2.1 The Ultimate Sequential Machine

Conceptually, the fastest sequential machine we can build is one which wastes no time performing useless work and which performs as much of the work in parallel as it can, executing in sequence only those actions that depend on a previous result. Of course, approaching this extreme requires an enormous amount of hardware in the processor, but it is indeed the ideal case.

Conventional von Neumann architectures have two critical parts: the execution unit, where operations are performed on data, and the memory system, where the data is kept and the state of the process is maintained. Each operation involves first transferring data from the memory system into the execution unit, operating on it, and then updating the memory system to store the freshly created result. Operations cannot start until the memory system has finished delivering the required data into the execution unit, and the operation hasn't ended until the result is transferred back to the memory system.

Since conventional machines usually require more than one memory cycle to complete an operation, fast registers are put inside the execution unit so that during the course of an operation the memory system is untouched. The execution unit operates on cached data sequentially, and only when it finishes does it write the result back to the memory system. This is, however, not the best a machine can do. Assuming that we are willing to put in enough hardware to ensure parallelism, we can arrange for even the most complicated operations on a reasonable amount of data to take only a very short time. This is particularly true if we notice that the common operations a machine encounters are mostly simple ones, such as simply moving data around, adding, bit manipulation, and multiplication[1]. Registers maintain the execution unit's internal state, which sometimes must be changed before memory needs to be accessed again. However many of these changes can occur simultaneously with useful data manipulation. The only truly sequential operations are thus memory transactions. This argument is more convincing when we consider that it is really only memory transactions that change the state of the computation. All other operations are functional in nature. The merit of an architecture must then be gauged by how often it is able to change the state of the computation, in other words, how fast it can issue sequential memory accesses, or how well it can keep the memory working. Beyond that we are limited by the technology available to construct a memory system that is as fast possible. The fastest computer given a certain technology must be memory bound.

A corollary of this argument is that since the memory system should always be fulfilling a data reference, there ought to be no time left for instruction fetches. The machine should use all available memory bandwidth for actual data transfer between the execution unit and storage system. Any other type of activity must be counted as overhead. Following from this a separate control store must be built to supply instructions. Of course keeping instructions in the primary storage has its advantages. A microcoded

---

[1] The Scheme86 prototype will not contain a multiplier, though one can be added without any penalty in speed.

architecture combines the benefits of storing user instructions in the main memory system while delegating the burden of controlling the cycle-to-cycle operation of the machine to the microcode store.

Conventional architectures attempt to efficiently use the memory system without putting an extraordinary amount of hardware in the execution unit by noticing that in many cases memory reference patterns can be predicted. Registers are put inside the processor to cache operands. Before starting to operate on the cached data, the processor issues a memory request for the next operation, and while the memory system is working the processor carries on the current execution. Because of this approach the emphasis on the memory system is its throughput, rather than its latency. In this scheme it is not important how long it takes for a given memory reference to return, as long as memory references can be initiated frequently. However, this approach is effective only if the subsequent memory request does not depend on the result of the first operation. If the memory address or data is the result of the first instruction, then the processor must wait for it to complete before the second memory transaction can start. This phenomenon occurs frequently in LISP. As a matter of fact we found that in critical sections of the SCode interpreter the processor frequently has no work to do pending the return of operands from the memory system. This is also true in the garbage collector, and in general for code that relies on `CAR-CDR` chaining. For this kind of application a machine must therefore stress not only the throughput, but also the latency of the memory system.

Although it is true that adding extra hardware inside the processor will increase the chance of it being able to issue memory requests more frequently, a complicated processor may have a huge propagation delay so that in the end the memory system is not efficiently used. Since most operations encountered in the execution of LISP programs are simple ones such as dispatching and register shuffles, an efficient architecture should not spend its hardware dollars optimizing the execution of complex atomic functions which may take a long time. It should focus on putting in a lot of execution paths so that many simple events can happen concurrently.

2.2 The Scheme86 approach

Scheme86 is a particular application of the above ideas. It recognizes the fact that LISP programs frequently manipulate structures which are made up of pointers to other chunks of data. A substantial part of the work in executing LISP is thus to resolve chains of pointer references to reach a piece of data. The memory system in a LISP computer must then be of extremely high bandwidth. Moreover, successive memory requests for dereferencing pointers are chronologically dependent, requiring that the memory system must not only have a high throughput, but very low latency as well. To prevent wasted memory cycles, any hardware added in the processor to make it possible to start memory references faster is a worthwhile investment.

Accordingly Scheme86 uses multiple execution units to ensure that minimal time is spent inside the processor between dependent memory transactions. In a single cycle it can perform both address and data calculations in parallel, while still being able to do some internal house keeping computation. The memory system is interfaced to the processor in a way that memory requests issued in one cycle returns with valid data in the next cycle. The next request can start, using processed data from the first request, in this second cycle. The memory system is built with fast static memory with a total latency of less than 90 nanoseconds.

Scheme86 has four distinct execution units. A complete functional description of each unit appears in the next chapter. Three of these units work on full machine words, with the fourth one specialized to handle only the type code part. Sixty internal registers in a register array are attached to each execution unit. In addition each unit is fed by different, specialized external registers, with the Memory Data Register, which contains data returning from memory, common to all of the units. Execution Unit 1 (`EU1`) has a full arithmetic unit and feeds one of two Memory Address Registers. `EU2` has a similar arithmetic unit and feeds one of two Memory Buffer Registers. The third execution unit has dedicated hardware to detect pointer equality and with its two operands supplies the remaining Memory Address and Memory Buffer Registers. One of the operands can also be routed back into the register array to implement a register transfer.

With the above arrangement a memory transaction can start once the address and/or data arrives at the Memory Address/Buffer Registers. It need not return until the next cycle when other operands from the register array have been fetched. The cycle time of memory and hence the machine is thus determined by adding the latency of the memory subsystem, a couple of interface registers, and the propagation delay of the arithmetic unit. With the technology we are using the number is approximately 120 to 150 nanoseconds. Note that during this time two arithmetic operations plus one register transfer have occurred. The next memory reference can start using results from the first.

The power of this arrangement can be exemplified by characteristic portions of microcode. Consider an operation like pushing the sum of two registers onto a post-increment stack. In Scheme86 this sequence of events can all occur in one single cycle:

```
(define-state PUSH
  (assign SP (+ (fetch SP) 1))                 ; EU1
  (assign MBF (+ (fetch REG1) (fetch REG2)))   ; EU2
  (assign MAR (fetch SP)))                     ; EU3
```

A common LISP indirect data reference such as the expression `(cdadr x)` with type checking can be coded as follows:

```
(define-state DO-CDR-1
  (assign MAR (cdr (fetch X)))                 ; CDR is 1+ on addresses
  (if (not (eq? (type (fetch X)) (type CONS-CELL)))
      (goto ERROR-WRONG-TYPE)))
(define-state DO-CAR-1
  (assign MAR (car (fetch MDATA)))             ; CAR passes pointer only
  (if (not (eq? (type (fetch MDATA)) (type CONS-CELL)))
      (goto ERROR-WRONG-TYPE)))
(define-state DO-CDR-2
  (assign MAR (cdr (fetch MDATA)))
  (if (not (eq? (type (fetch MDATA)) (type CONS-CELL)))
      (goto ERROR-WRONG-TYPE)))
```

It is often desirable to have runtime array bounds checking. Assuming that each array contains a header specifying its length, an array reference takes at least two memory references and some arithmetic comparisons. In Scheme86 the primitive functions VECTOR-REF and VECTOR-SET! can be coded in three instructions, the last of which is so trivial that it can be merged into subsequent instructions:

```
;;; Vector is in Arg1. Index is Arg2. Result returned in Val.
;;; Vector length in vector header.
(define-state VECTOR-REF
  ;; Get address of cell. +CARRY a carry to bump past header.
  (assign Tmp (+carry (integer (fetch Arg1))
              (integer (fetch Arg2))))  ; EU1
  ;; Set up test for whether index is negative
  (assign Arg2 (fetch Arg2) (test VECTOR-REF))  ; EU2
  ;; Fetch header
  (assign MAR (pointer (fetch Arg1)))           ; EU3
  (if (or (not (eq? (type Arg1) (type VECTOR))) ; TCU
          (not (eq? (type Arg2) (type INTEGER))))
      (goto VECTOR-REF-WRONG-ARG)))
(define-state VECTOR-REF-2
  ;; Test index against length of actual vector
  (assign Tmp (- (integer (fetch MDATA)) (integer (fetch Arg2)))
      (test VECTOR-REF-2))
  ;; Fetch cell. This is bogus if tests fails.
  ;; For a VECTOR-SET! the value can flow through EU3 or EU2 into
  ;; the memory buffer register and hence be written out.
  (assign MAR (pointer (fetch Tmp)))            ; EU3
  (if (negative? (test VECTOR-REF))
      (goto OUT-OF-BOUNDS)))
(define-state VECTOR-REF-3
  ;; Transfer result to Val. This cycle can be merged with real work.
  ;; Its only real function is see whether the previous bounds check
  ;; has failed.
  (assign Val (fetch MDATA))
  (if (>? (test VECTOR-REF-2))
      (goto OUT-OF-BOUNDS)))
```

Even more revealing is the microcode for the LISP function `ASSQ`, an example of an algorithm which has chronologically dependent memory references:

```
;;; A-LIST is a register for holding the pointer to the association
;;; list.
;;; KEY holds the value to match against.
(define-state ASSQ-LOOP-1
  ;; Assume we just read the association list from memory, so it
  ;; is in MDATA at this point.
  ;; Take its CAR to read in the association pair.
  (assign MAR (car (fetch MDATA)))            ; EU1 (or EU3)
  ;; Store the list for later use
  (assign A-LIST (fetch MDATA))               ; EU2
  ;; Check for the end of the list
  (if (eq? (fetch MDATA) (fetch Null-Reg))    ; EU3
      (goto LOSING-TERMINATION)))
(define-state ASSQ-LOOP-2
  ;; Take CAR of pair to read in the key
  (assign MAR (car (fetch MDATA)))            ; EU1
  ;; Save association pair, in case we match
  (assign VAL (fetch MDATA)))                 ; EU2 (or EU3)
(define-state ASSQ-LOOP-3
  ;; CDR down the list for next value, in case we don't match
  (assign MAR (cdr (fetch A-LIST)))           ; EU1
  ;; Try to match key in pair against required key.
  (if (eq? (fetch MDATA) (fetch KEY))         ; EU3
      (goto WINNING-TERMINATION)
      (goto ASSQ-LOOP-1)))
```

As is evident, a three cycle loop is required to implement `ASSQ`. Since this can be put in the microcode, there is no need for instruction fetching. Each iteration hence takes 450 nanoseconds. In other words we can search through a one million element list is less than a second. The same inner loop, written in the Motorola MC68020 assembly language, requires five instructions and more than forty cycles.

2.3 Scode: The Scheme86 Instruction Set

Because of the amount of parallelism and flexibility inherent in the Scheme86 architecture, its native control instruction must be wide. In order that all of the available memory bandwidth be usable for data references rather than instruction fetches, instructions should be placed in a separate control store so that fetching can be performed in parallel with ordinary memory transactions. Because of the width of each instruction, the control store must be compact, and fast. It must also have a wide interface to the rest of the processor. These requirements point to a design that uses microcode to interpret user instructions (assembly code).

Common microcode architectures have certain serious pitfalls. They offer a large number of machine instructions, often of long and variable length. The processor has to spend considerable time fetching an instruction and decoding it to figure out what to do next. Since each instruction is atomic, the programmer cannot freely schedule each microcycle to make use of memory and branch interlocks. These drawbacks have led designers to turn to machines with a small number of fixed length machine instructions that can be completed in a single cycle. These instructions control the hardware directly without the need for further microcode interpretation.

By using SCode as its machine language, Scheme86 combines the benefits of both of these approaches. Since SCode instructions are few in number (less than 256) and are fixed in length (8 bits), it is easy to implement in hardware a fast dispatch to interpret them. Each SCode instruction has powerful semantics. That means a large number of microcycles will elapse before an instruction completes. The ratio of microcode instructions versus machine instructions is thus considerable, implying that only a small overhead for interpretation has to be paid in comparison to the amount of useful work that actually is performed between machine instructions. This also means that most of the time the machine is executing microinstructions, which can be hand optimized to make use of scheduling and other hacks like heuristic memory fetch prediction to fully utilize the hardware. Since microinstructions, being fetched from the control store, can afford to be a lot wider than machine instructions, parallelism can be exploited and fine grain control can be kept.

Like in FORTRAN, a large number of library functions are provided in LISP. With a microcoded architecture, frequently used functions can be hand written in microcode for maximum efficiency and utility of the hardware. The `ASSQ` function given above is an example.

2.4 Support for Scheme

Scheme is a lexically scoped dialect of LISP which features runtime data typing, tail recursion, and dynamic storage allocation. To implement some of these features efficiently, hardware support is indispensable.

Scheme86 is built around a tagged architecture. Each word, or item, has an 8 bit type code field and a 24 bit datum field. The datum field contains either an immediate piece of data or a pointer to another cell. The type code field is used to identify the type of the datum, in the case of an immediate object, or the type of the reference, in the case of a pointer. For SCode items an "opcode" is placed in the type code field.

Scheme86 maintains separate data paths for the type code portion and the pointer portion of a word. On each microcode instruction, separate operations are specified in each execution unit for the two parts. Arithmetic is performed only on the datum field, while type code generation is handled by dedicated hardware. This approach eliminates the need in conventional architectures to mask, insert, or alter type information before and after operating on a cell.

In addition, one of the execution units in Scheme86 is dedicated to testing the type portion of a word. During every cycle, the types of two operands can be tested for equality with two microcode specified constants. Not only is the microcode allowed to branch on any logical combination of the two results, the machine can trap on these combinations to move execution to any one of eight microcode specified handlers. All of this can happen in a single cycle, concurrent with full cell operations taking place in the other three execution units. The result is a powerful mechanism for performing dynamic type checking and implementing generic operations without any cost in speed.

Because the type code is kept in the same part of a cell as SCode instructions, the instruction decoding hardware can be used to implement an N-way dispatch for type codes. This is important for a stop and copy garbage collector like the one to be used for this implementation. Combined with the low latency of the memory and all the parallelism available for doing pointer manipulation and data transfer concurrently, garbage collection will be very efficient.

Besides being block structured, Scheme features first class environments objects which the user can incrementally modify. Because of this, lexical addresses cannot be trivially cached to speed up variable lookup. Implementations of Scheme solve this problem by adding a "danger bit" to the type code of each piece of datum to warn of modified environment frames, invalidating erroneous values looked up using cached addresses that are out of date. In Scheme86 this bit is built into the hardware and can be tested easily by the microcode.

3. The SCode Instruction Set

The instruction set that Scheme86 executes is SCode, which is a typed-pointer, tree-structured representation of Scheme. SCode instructions may recursively point to other SCode forms as arguments. The type on the pointer serves as the "opcode" to be dispatched on by the machine. Scheme86 interprets the tree-structured SCode expressions recursively. Translating Scheme into SCode is called "syntaxing".

3.1 SCode Instructions

Each SCode instruction corresponds to a special form in Scheme . There are instructions for procedure calls, conditionals, variables, primitives, and other Scheme features. The following is a list of the essential SCode forms:

1. `ASSIGNMENT`. Corresponds to `SET!` in Scheme. The instruction points to a pair containing the variable to change and the value to change to.

2. `COMBINATION_N`. Corresponds to a procedure call. The operator and operands, themselves SCode expressions [1], are kept in a vector pointed to by the instruction. There are special SCode instructions for short, fixed length combinations.

3. `CONDITIONAL`. Used for `COND`, `IF`, and `AND`. The arguments, kept in a triple, are the predicate, the consequent, and the alternative. Related is the `DISJUNCTION` instruction used to implement `OR` efficiently.

4. `DEFINITION`. Extends the current environment, binding a name to a value. This is the Scheme form `DEFINE`.

5. `ENVIRONMENTS`. Each Scheme environment is represented as a vector containing the bound values and some housekeeping information. Matching this vector against the parameter list kept in a `LAMBDA` creates an association between names and values.

---

[1] Scheme has a uniform namespace for operators and operands; the operator is evaluated to give a procedural value just like its arguments.

6. LAMBDA. A PROCEDURE object is created by associating the current execution environment with a LAMBDA expression.

7. PRIMITIVE. An implementation primitive operator. On Scheme86 this would be a microcoded routine. The pointer contains an entry point that can be dispatched to by the microcode.

8. PRIMITIVE_COMBINATION. Similar to COMBINATION_N, except that the operator is known to be a primitive. Up to three arguments are currently supported. This is a major optimization performed by the syntaxer: known primitive calls are integrated into this instruction rather than a COMBINATION_N. The primitive to be called is pointed to by the first argument.

9. PROCEDURE. It points to a LAMBDA expression and its parent ENVIRONMENT [2]. When it is applied, a new environment frame is created, with ENVIRONMENT as parent.

10. SEQUENCE_N. This is a collection of expressions to be executed one by one. Scheme requires that the execution of sequences be tail recursive, meaning that minimum state should be kept around when the expressions are evaluated.

11. VARIABLE. Corresponds to identifiers which are not special forms in Scheme. The value of a variable is initially found by searching down the environment chain, matching parameter names in lambda lists to the identifier. This process is called a "deep search". The frame and slot offsets are then cached inside VARIABLE, so the next time through indexing can be used to fetch the value. This is called a "compiled" lookup. However, the ability to incrementally modify Scheme environments makes it possible for this mechanism to fail. The remedy is an extra bit, called the "danger bit", which is allocated for each Scheme object to mark whether compiled lookup is safe for this value [3].

[2] Since Scheme is lexically scoped, the parent environment is the one in which the LAMBDA was evaluated.

[3] The actual implementation is more complicated than this. Certain environments

As is evident, SCode instructions correspond closely to Scheme special forms. Interpreting SCode is recursive; evaluating an SCode form might lead to the interpreter first having to "reduce" another sub-expression. Since SCode is just like Scheme , the evaluation rules are also the same [Abelson 1985].

3.2 The Case for SCode

SCode has two major problems which makes it not an obvious choice as an instruction set for a fast computer. First it is merely a typed-pointer variant of Scheme. Little optimization is performed during the translation. It is also too complicated to be used to control hardware directly. A layer of interpretation by microcode is necessary. The second problem is that the tree-structured rather than linear organization makes instruction caching extremely difficult.

In fact SCode could be expanded so that traditional techniques of compilation can be used to generate more efficient code. For example we can add SCode instructions that encapsulate special cases of argument evaluation or sequencing which are trivial and do not require that the processor save its current state. This would reduce stack usage and also result in a more compact SCode sequence and shorter execution times. However, compiling Scheme is not a trivial task itself [Rozas 1984, Miller 1985]. In many cases a good interpreter can match compiled code's performance on a poorly designed machine.

Interpretive code leaves a lot more state and debugging information behind. Since SCode is so similar to Scheme, it is possible to reverse the syntaxing process and produce equivalent source text from SCode expressions. When an error occurs the debugger can then pinpoint the irritant and print it out to the user at source level. If we consider also that syntaxing is fast and incremental, we can see why an efficient implementation of an SCode interpreter such as Scheme86 makes the perfect development environment for software [Miller 1985]. Block or cross-procedure compilation relied upon by many

---

have to be marked dangerous as well, for reasons too complicated to explain here.

architectures to ensure performance takes a lot of time, is not incremental, and thus not suitable for languages such as LISP which stress an interactive user environment.

Some optimizations cannot be performed during compilation but can easily be done dynamically. The "rack" idea used in the SCHEME81 implementation is an on-the-fly optimization for reducing stack operations. Although a similar hack is not expected for Scheme86, using SCode as its instruction set certainly opens doors for such optimizations.

4. The Scheme86 Architecture

4.1 Rationale

Although the basic Scheme86 architecture was arrived at after considering the principles outlined in the previous chapter, actual microcode was written and evaluated before the design was finalized. Since the machine is designed to execute SCode, critical sections of a Scheme interpreter using the SCode abstraction were studied. These included code for performing function calling, variable lookup, conditional dispatches, and garbage collection. A pipelined and a non-pipelined memory model were both investigated. As many data paths and execution units as needed were first assumed, and then the code was analyzed to try to find a reasonable number to implement.

These points were evident from the code written:

1. Interpreting Scheme is memory intensive. With few exceptions the interpreter would be able to start a memory reference on every cycle, if not more.

2. It is not easy to utilize memory interlocks in a pipelined memory model to do useful work. After a memory reference is initiated, the processor frequently must stay idle until the memory transaction is completed. To start the next request requires first operating on the result from the previous transaction.

3. In order that memory cycles can be initiated as frequently as possible, two to four distinct paths for operating on data are required.

4. The most frequent addressing mode is "indexed indirect". This is not surprising since SCode is tree-structured. Executing SCode requires that the processor follow pointers through chunks of SCode structure.

These findings more or less agree with the theory described in Chapter Two.

Figure 1. Block Diagram of the Scheme86 data path architecture. The micromachine and clock are not shown.

4.2 Constraints

Although the initial investigation of Scheme86's architecture centered on a machine with an infinite number of datapaths and an extremely wide microcode, real life constraints were considered when the architecture was being finalized. These points were the most important:

1. The Scheme86 prototype will be built from stock parts. TTL technology will primarily be used. This precludes the use of custom VLSI or even gate array chips. A further limitation is that only currently available parts must be considered. This ensures that the project can be completed within schedule.

2. The prototype must fit on a single board. The consensus is that a more ambitious attempt is inappropriate for a thesis. This constraint limits the degree of parallelism that we can afford.

3. Scheme86 will work as a back-end processor to a host computer. This way all of the I/O operations can be delegated to a commercially available machine. The processor can also have all of its microcode be downloaded from the host. Both aspects are important to an experimental architecture. A parallel interface is chosen because it is easy to implement.

4.3 The Datapath Architecture

A Scheme86 machine word consists of 24 bits of datum and 8 bits of tag information. Since word addressing is used, the machine can address 16 Megawords or 64 Megabytes of data. All addresses are physical – virtual addressing is not supported.

Scheme86's main memory system is built entirely out of high speed static RAM chips. Initially 2 Megabytes of memory will be populated. Using 64K $\times$ 1 static RAM with an access time of 45 nanoseconds, the memory system as a whole is expected to have a latency of around 80 to 90 nanoseconds. Each memory board contributes 1 Megabyte. [1] The processor/memory bus is synchronous. Since the main memory system already offers extremely high throughput and low latency, there is no need for a cache. The control store (where the microcode resides) is separately constructed using 35 nanosecond [2] static RAM. Twelve bits of address provide a maximum of 4 Kwords of microcode. Each microcode word is 160 bits in width.

Because analysis have shown that memory interlocks cannot be usefully employed, Scheme86's timing requires that memory references take only one cycle to complete. A request initiated in the current cycle is fulfilled and the data is ready for use in the next.

The Scheme86 processor has four distinct function units. Sixty general registers are implemented using 64×18 bit register files that are dual ported for both read and write. Four more special purpose registers may be mapped into the register address space to make up a total of 64 registers per execution units. One of the external registers present on all four execution unit is the Memory Data Register (`MDATA`) which is the interface to data returning from memory.

Each execution unit takes two independent operands (A and B) and generates results and conditions. The four execution units are:

1. Execution Unit 1 (`EU1`). It incorporates a full 24 bit arithmetic logic unit with carry lookahead generation. Both operands can be any of the 60 internal registers. Three external registers are available on the A operand bus. They are the memory input data register (`MDATA1`), the `PCLOSER` register, and the `MARLOSER` register. The latter two capture the most recent microinstruction address and memory bus address prior to an exception. A two-to-one multiplexer selects whether the resulting type code from this execution unit is the one supplied as operand A of the

[1] Initial estimate. It may be able to fit 1.5 to 2 Megabytes depending the final board size chosen.

[2] 25 nanosecond replacements expected late 1986

ALU or a constant supplied by the microcode. The result can be written back into the general registers. It can also be used to load the first of two possible memory address registers (`MAR0`), and the Host Output Register (`HOUT`) for sending data to the host computer. The memory address registers only hold the 24 bit pointer portion of the result, so no explicit masking of the type code is necessary. The top bit of operand A is routed to the micromachine as the danger bit flag.

2. Execution Unit 2 (`EU2`). Similar to `EU1`. The external registers, besides the `MDATA2`, are the host input register (`DATAIN`), a 16 bit constant register (`CONST`), whose value is a concatenation of the two 8 bit type code constants supplied to EU1 and EU2, and a software (Scheme) interrupt register (`EXT-INT`). Full word (32 bit) constants are obtained by assembling two 16 bit chunks over two cycles. The result can be loaded into one of the two memory write buffer registers (`MBF0`), in case the current cycle initiates a memory write. Before the result is written back into the register array, it is passed through a 32 bit barrel shifter for an arbitrary amount of shift or rotation. As in `EU1` the danger bit is routed to the sequencer.

3. Execution Unit 3 (`EU3`). This execution unit contains only a 32 bit identity comparator for generating the LISP `EQ?` function. Operand A can be written back into the register array and so provides an extra path for performing register transfers. It is also loadable into `MAR1`. Operand B is loadable into `MBF1`. The top 8 bits of Operand A are sent to the microsequencer for dispatching. This execution unit has `MDATA3` as the sole external register, which can feed operand A.

4. Type Code Unit (`TCU`). The operands are 8 bits wide and contain only the tag portion of the registers. Operand A is tested for equality against the type code constant supplied to `EU1` and operand B is tested against the one supplied to `EU2`. Distinct condition flags are set according whether either, both, or each of the pairs are equal. These condition flags are supplied to the microsequencer both as branch conditions and as addresses into a handler table for type code dispatches.

Figure 2. Timing Diagram. The various signals are generated by taking `ECLOCK` and passing it through a delay line and combinational logic. These signals are used to trigger various events within a microcycle.

The two arithmetic execution units generate sign, carry and overflow information that can be used in the next cycle to conditionalize branching. However, the special `EQ?`, danger bits, and type code flags can be used in the current cycle to determine what the next instruction is. In addition the microcode can enable the type code flags to signal an exception.

A memory transaction can be initiated using the address and data latched in either `MAR0`, `MAR1`, `MBF0`, or `MBF1`. On a memory read the data returns in the next cycle in `MDATA1`, `MDATA2`, `MDATA3`, and `MDATA-TC` for each of the execution units.

A major cycle of the processor can be divided into four phases:

1. Register Read Phase. Delimited by the signal `REGLE`. Data is read from the registers into the arithmetic or predicate unit. Data from a memory request issued during the previous cycle returns and is loaded into and read from the `MDATA` registers.

2. Operation Phase. The arithmetic and predicate units do their job. Output is valid at the end of this phase. This phase starts on the falling edge of `REGLE` and ends on the rising edge of `REGWR1`.

3. Register Writeback Phase 1. The result from the operand A bus of `EU3` (a register transfer) is written back into the register file. All other results are loaded into selected external registers, and memory transactions can start immediately. The barrel shifter on `EU2` starts operating. All condition and interrupt flags [3] are latched and supplied to the microsequencer, which can then start computing the next microaddress.[4] This phase is delimited by `REGWR1`.

4. Register Writeback Phase 2. Since three result must be written back into the dual

[3] For arithmetic conditions, the flags are the ones generated in the previous cycle. The `EQ?` and type code flags are those generated in the first two phases of the same cycle.

[4] See next section.

ported register files, a second register write phase is used. The result from `EU1` and the barrel shifter are written in this phase. By the end of phase 4 the next instruction is fetched from the control store and ready for latching when phase 1 starts up again in the next cycle. This phase starts on the rising edge of `REGWR2` and ends on its falling edge.

Each phase takes about 30 nanoseconds to complete. Since the architecture specifies that memory references complete in one cycle, the minimum cycle time is determined by how long that takes. Using 45 nanosecond memory parts and assuming two levels of buffers driving the address to memory and receiving data from it, the memory board should have a latency of about 85 nanoseconds. Memory address decoding occurs in parallel with the operation of the memory chips and need not be entered into consideration. The received data on the processor must be buffered and clocked into `MDATA` prior to the operation phase (20 ns). It can then pass through the ALU and be clocked into the memory output registers (35ns). Another layer of buffering conditions the signal to be sent onto the bus again (10ns). Adding the propagation delays gives a cycle time of 150 nanoseconds. These numbers are derived using worse than the maximum propagation delays of components in the Advanced Schottky logic family used and already allow for slack on the bus and circuit board traces.

4.4 The Microcontroller Architecture

The microcontroller used for controlling the data paths is a fairly generic finite state machine built using memory chips and pipeline latches. Complications arise only to implement microcode loading from the host, the instruction dispatch and typecode instruction dispatch, and non-existent memory exception.

Since all the microcode resides in RAM, the host computer is responsible for coordinating the microcode transfer. For purposes of control the host communicates with Scheme86 through eight synchronous status lines. Another 16 lines supply data in two chunks to make up a full 32 bit word on the processor. These lines are bidirectional to implement a similar pathway back from the processor to the host. Five 32 bit chunks of loading latches with 3-state output supplies the microcode memory chip I/O bus. The latches are loaded with appropriate data when they are addressed through the status word and a write enable line is asserted. Loading these chunks sequentially assembles a full 160 bit microword. The microcode address can then be forced to any specified location by loading it into the `BOOT` register and asserting a `BOOT` interrupt. This is the highest level priority and ensures that the microaddress correspond to the one residing in `BOOT`. At this point a write enable line can be asserted to write the microcode word into the control store at the microinstruction specified.

Booting the microsequencer is a similar process. A desired address is clocked into the `BOOT` register. When the `BOOT` interrupt line is taken high by the host, execution is forced to and held at that address. The host can then release `BOOT` to resume execution starting at the same address.

There is only one type of instruction in the Scheme86 micro-machine. Each instruction explicitly specifies the address for the next instruction. Also specified is an alternative address to use if some condition is false. Each address is 12 bits long, making it possible to address four Kwords of instructions. The remaining 136 bits in each instruction are assertion signals. They are used to specify the operands and functions of the execution units, control loading of registers, select branch conditions, and enable interrupts. Using only one type of microinstruction with explicit addresses may at first seem like a waste of hardware and instruction width. However the increased decoding speed and the removal of complicated sequencing hardware make the increased width worthwhile.

After each instruction, execution continues at `NEXT` if the branch condition being tested is true, and `ALT` if it is false. There are four more sources of address that can determine the next state of the controller. The `DISP` address, supplied by the top 8 bits of operand A in `EU3`, when enabled, replaces the lower 8 bits in both `NEXT` and `ALT`.

This implements a 256-way dispatch into two possible dispatch tables depending on a branch condition. The top four bits of `TC-XCP-ADR` is loadable from `ALT`, while the lower eight bits are made up of one constant bit, three microcode controlled signals, and four conditions generated by `TCU`. `TC-XCP-ADR` is enabled if one of the four conditions (or their complements) matches one specified by the microcode. The third source is the `NXM-XCP-ADR`. It is loadable from `ALT` and is enabled if a reference into non-existent memory occurs. This exception can be disabled. The fourth source is `BOOT` described above. `BOOT` takes the highest priority, followed by `NXM-XCP-ADR`, `TC-XCP-ADR`, `DISP`, and the normal addresses.

The operation of the sequencer can also be described in four phases:

1. Execution Phase. A microinstruction is read and latched into loading registers. Control signals propagate to control various parts of the datapath and the sequencer. This phase is marked by `ELE`.

2. Address Latch Phase. With the current instruction safely clocked inside holding registers, all microcode address sources are clocked with addresses for the next state. This phase starts on the falling edge of $\overline{\texttt{ALE}}$ and ends on its rising edge.

3. Decision Phase. Depending on interrupt and branch conditions, one of the address sources is selected. Enables for these sources are clocked. This means that all condition flags that may affect the next state must be valid at the beginning of this phase. Phase 3 and phase 4 are not well delimited.

4. Instruction Fetch Phase. After the address has stabilized, it is used to index into the microstore to produce the next instruction. Phase 4 ends at the start of the next cycle.

A cycle starts on the rising edge of the main clock signal (`ECLOCK`) which has a duty cycle of 50 percent. The other clock signals used to signify the start and end of the various phases are derived from `ECLOCK` using a delay line and combinational logic. Positive multiphase clocking is used throughout the design. In fact the machine can tolerate skews of up to 10 percent of the main cycle time.

4.5 Construction Details

The prototype Scheme86 machine is being built using mainly Advanced Schottky TTL technology. The data paths and micro machine reside on the processor board. It will contain over two hundred and fifty standard TTL chips, seven register files in 120 pin grid array packages, and a barrel shifter in an 84 pin package. The board dimension is estimated to extend over 22 inches on each side. A four-layered printed circuit board is used, with two signal layers on the surface, and power and ground planes in the middle. Much of the board's area is occupied by wires rather than chips. The board communicates to the backplane carrying the data and address buses through two pin-connectors with 108 pins each. The processor board is expected to draw about fifteen to twenty amperes of current, half of which is used to supply the register files. At the time of writing the processor board layout is close to complete.

The memory board will have roughly the same number of packages on it. The preliminary design calls for putting one Megabyte of storage per board. As mentioned before this will be implemented using 64K $\times$ 1 CMOS static RAM with access times of 45 ns. Each memory board draws 20 amperes of current.

The machine will be dissipating over 300 watts of power in a very small area. The design specifies that a total of twelve high-powered muffin fans, each delivering 100 CFM, be mounted around the card cage.

5. The Logic Simulator

Before Scheme86's design was committed to hardware, it was entered into a simulator to verify the logic and timing. The simulator was based on a similar system by Gerald J. Sussman and Michael Douglas for testing the Digital Orrery. The original version was written using Portable Standard Lisp and a message passing system developed at the Hewlett Packard Laboratories. A simplified implementation in Scheme appears in [Abelson 1985]. For Scheme86 this simulator was rewritten using Scheme and the "class" object oriented programming system designed by Chris Hanson for implementing the Scheme text editor Edwin.

5.1 Overall Design

The simulator works at the logic level and is event driven. Each node in the circuit is entered as a wire which is represented in the simulator by an instance of the *wire* type. Each wire object keeps track of its internal state with instance variables. The information includes what signal level it is at (high, low, unknown, or undriven), which gates are driving it, and which gates receive input from it. Each wire instance can represent a single wire or a bus of arbitrary width. In the case of buses the integer value of all the bits, rather than just a binary value, is maintained. The value may also be symbolic, but this requires the components attached to the wire be able to handle non-integer values. Operators exist to merge a number of narrower buses into a wide bus and to break a bus down into smaller buses.

Gates are also object instances. They are associated with methods that produce output signals computed from input signals. The output signals may be delayed by an amount depending on the type of the component. A number of component types are currently implemented, and new types can easily be added when the need arises. The components are sliced according to the width of the wire they are connected to, rather than individually or according to actual physical packages. This provides a lot of convenience entering the circuit besides ensuring a more meaningful and efficient simulation. Complicated component types can be constructed by internally wiring together more primitive functional elements. The new compound type is then modeled by the internal circuit of sub-components. This hierarchical abstraction mechanism can be generalized to implement complex components which represent an entire sub-system in an overall design.

When a component is added to the system, it notifies the wires it is attached to by sending the wire objects an appropriate message. To wires it receives input from, it passes a handler that will trigger a response by the component when the wire switches state. The handlers are recorded in the wires. Whenever the logic level or status of a wire changes, it executes the handlers one by one, thereby propagating the signal along. When a component wants to alter the logic level of a wire, it sends the wire object a message along with the desired signal. The system hence propagates by first having components drive and change the state of wires, and then letting the wires trigger attached components in turn to drive their output nodes. This way a single change in one node of the circuit influences the entire network of components to respond according to their functional specification.

The concept of propagation delay is implemented by maintaining an agenda of action. The component handlers can work not only by performing their action immediately, but also by encapsulating the desired action in a thunk[1] submitted to the agenda with a specified amount of delay. The system goes down the agenda and executes each thunk in chronological order. At the same time it accumulates the delays to maintain a notion of elapsed time. Timing jiggles are modeled by inserting or deleting a random amount from the specified delays.

The agenda mechanism also works for installing handlers, in the case of monostable devices, to check for hold times. Setup times are ensured by inserting a delay equal to the minimum setup requirement between when the input wires change state and when the

[1] A thunk may be viewed as an object encapsulating both code to be run and the execution environment. In Scheme they are implemented as closures.

actual inputs to the gates are affected.

Normally only one gate can drive a wire at a time, and the wire object complains if this is not the case. Tri-state buses are implemented by temporarily lifting this constraint so that there may be some minimum overlap.

Any node in the circuit can be monitored by attaching a probe component to it. The handler for probes simply notifies the user with typeout when the signal changes, informing him of the old and new values and the current time. The trace component can actually plot the signal level versus elapsed time on the screen. This implements a simulated logic analyzer for debugging clock generation and timing constraints.

5.2 Result and Evaluation

The simulator helped catch a handful of mistakes, all concerned with clocking and timing. I was able eventually to simulate the execution of many microinstructions to ensure that critical sections of the hardware have no basic design flaws. In particular the microcode loading sequence is thoroughly tested, the sequencer was found to be bug free, and the datapath design correctly produced results with no apparent race conditions.

The operation of the simulator was sufficiently efficient. When simulating all of the processor board design the simulator can execute one microinstruction (150ns) every second. Since the system was implemented in Scheme it is also easy to replace working subsystems by custom written component modules with similar behaviour but are no longer faithfully represented by large numbers of primitive logic elements. This greatly speeded up the simulation process. For example, once the microcode loading sequence was certified to work, the ten minute process in the simulator was replaced by a procedure that took negligible time to complete. Further simulation was then centered around the datapath design.

Appendix Five contains a sample of the code describing a part of the machine to the simulator.

6. The Layout System

An important computer-aided design tool implemented was the procedural layout system used to specify the design of the printed circuit boards on which Scheme86 is to be built. Graphical features on each board is described with Scheme code. The layout system evaluates the Scheme expressions and then either displays the design on a graphics monitor or generates CIF, a low level specification language recognized by the printed circuit board manufacturing facility. The system provides means of combining primitive geometric figures into compound structures which in turn can be cut and pasted to form yet more abstract structures. Locations within structures can be referred to symbolically.

The system bears resemblance to the Design Procedure Language, a similar tool developed at the MIT Artificial Intelligence Laboratory [Batali 1980]. Some features and functionality were inspired by EARL, another layout language [Kingsley 1982]. Currently the system is oriented towards laying out printed circuit boards. Adapting it for VLSI development or as a general graphics language should not be difficult.

6.1 Implementation

The system is embedded in a Scheme interpreter. A handful of new special forms and functions that explicitly manipulate the layout are made available to the user. Since the geometric objects are represented as Scheme objects, all the power of the Scheme language can be used in conjunction with the layout functions to describe and manipulate the geometry of the design. For example, buses can be constructed by iteratively calling a Scheme procedure that produces one wire, each time with slightly different arguments.

There are two kinds of primitives. There are purely geometric structures, such as lines, boxes, and points. There are board primitives, for example vias, standard plated-through holes, ground connections, and wires. Conceptually board primitives can be constructed as compound structures of geometric primitives, but it was not done for efficiency reasons. Each primitive is a Scheme procedure. Instances of a primitive can be made by applying the procedure with where it is to be placed and other relevant parameters. Primitives know how to draw themselves relative to the point they are placed.

Compound objects are called structures. The special form `DEFINE-STRUCTURE` creates an environment in which primitives or other structures can be placed. Coordinates specified inside this environment are relative to the structure only. Once made, the entire structure can be instantiated like a primitive, i.e. it can be rotated, translated and incorporated into other structures. Because structures are just like primitives, they are also implemented as procedures. Drawing a structure simply involves translating to the correct origin and then recursively drawing the sub-structures. When a primitive is reached drawing actually occurs.

Several instances can be glued together and manipulated as one through the use of the `ALL-OF` operator. It is a procedure that takes instances as arguments which it incorporates into a new instance. The difference between these compounds and structures is that the latter become system primitives which can be instantiated and translated, while the former cannot.

An instance is implemented as a closure. Its Scheme environment captures its internal states, for example where it is, which layer it is on, and an association list of its inferior structures and their names. The procedure implements a message dispatcher as described in [Abelson 1985]. By applying the closure with messages we can inquire about an instance's internal parameters or cause it to draw itself.

The position of any instance can be found by using the `@` special form. The name of the instance and the list of structures which hierarchically contains it is supplied as arguments. `@` expands into the function `LOC` which traces down the chain of structures, maintaining the correct coordinate transformation, and finally returns the location of the required instance in the coordinate system of the calling environment. The lookup occurs when the Scheme expressions describing the structure are evaluated. This means the symbolic locations must already be defined. It also has the property that when the actual location of a symbolic reference changes through the use of incremental definitions, the new coordinate will not propagate to places that are referencing it. All dependent structures must be evaluated again to capture the new change.

The layout system uses Scheme's graphics interface to plot structures on video monitors and hardcopy devices in color or monochrome. The ability to display structures as they are being defined into the system makes it possible to debug the layout interactively and incrementally. The user can choose a structure to display, and instruct the system to either fit the entire structure on the display or zoom in to examine details.

Ultimately the system generates Caltech Intermediate Form, a low level graphics language that specifies the geometry of the board in a format that the fabrication facility can read. CIF differs from a high level specification like the Scheme layout language in that it does not have symbolic references, procedures or iterative constructs. CIF resembles commands sent to a low level graphics system. The CIF code generator in the layout system is implemented by consistently replacing each primitive's drawing function with a procedure that writes out an equivalent CIF command to a file. Since each primitive in the system has a counterpart in CIF, the translation process is simple.

6.2 Example

The following piece of code places a couple of chips side by side. It is assumed that BYPASSED-UD-24 is a defined structure that creates a 24-pin DIP package oriented in the upright direction.

```
(define-structure two-chips
  ;; Place first chip, call it U001
  (add-component! 'U001
    (bypassed-ud-24 (make-point 0 0)))
  ;; Place U002 one chip spacing on the right of U001
  (add-component! 'U002
    (bypassed-ud-24 (+point (@ (U001 chip pin20))
                    (make-point *chip-spacing* 0))))
  ;; Ground certain pins
  (add-component! 'ground-pins
    ;; Ground things on the solder side of the board. PIN10 is chip ground
    (all-of (solder-connect (@ (U001 chip pin8))
                            (@ (U001 chip pin10)))
    (solder-connect (@ (U002 chip pin11))
                            (@ (U002 chip pin10)))))
  ;; Route pin 13 to top of the chip, on component side
  (add-component! 'enable
    ;; Notice use of Scheme language features
    (let ((from (@ (U001 chip pin13)))
  (end (@ (U001 chip pin20))))
      ;; We clear the pin itself by 1 minimum clearance in the X-direction
      (component-connect from (pad-clear 1 0 from)
                 (pad-clear 1 1 end))))
  ;; End DEFINE-STRUCTURE
  )
```

For more examples of how the system works please read Appendix Four which lists code actually used for describing the processor board. Appendix One contains a summary of special forms and functions defined by the layout system.

7. Results and Conclusion

After many months of work the layout of the processor board and memory boards is near completion. Once they are fabricated hardware debugging work can start. A full SCode interpreter can then be installed on the machine to allow performance tests to be run.

It is hard to gauge the potential of this architecture without actually running any programs on it. Based on the microcode written and the cycle time expected, this machine will be able to interpret SCode substantially faster than any Scheme interpreter currently implemented. Because of the difficulties in compiling Scheme and the nature of the types of programs usually written in it, Scheme86 is expected to compare favorably against compiled Scheme programs running on conventional architectures implemented using similar technology. These expectations, though certainly backed up by the examples of routines we are able to analyze, is preliminary. Actual tests must be performed to prove the merits of the design.

8. Possible Improvements and Future Work

Because of the severe time constraint imposed on the project, some compromises in the architecture have to be made. A few genuine errors were also made in the design.

Owing to some bad experiences dealing with memory systems, I have chosen to map all of the specialized I/O and debugging registers into the register array space rather than the memory space. This moved a lot of hardware and wiring into the processor area, severely complicating the layout process. While this approach is efficient for the memory interface, non-critical registers should not take up space in the execution units.

The type code exception mechanism, though elaborate, is very difficult to make use of in software. A better design of this part of the architecture should be worked on once there is more microcode written.

Due to inexperience, the memory bus was not very well designed from both a software and electrical point of view. A minimal handshaking protocol is supported, and it will not be an easy task to implement coprocessors for this architecture. A better design of the bus can speed it up and hence improve the overall performance of the processor.

The timing of the processor will be a lot simpler if I can use a register file with one more write port. Using the dual ported register file we have, two register write-back cycles are needed. Given this situation a fourth write-back, possibly for another register transfer or the result of a floating point unit or a multiplier, could have been implemented. This was not done to reduce hardware and microcode width. Adding the extra write-back would have improved the potential performance substantially. Alternatively the processor's cycle time can be shortened with an extra write port. It will then be able to go through two cycles for each memory reference. This way the memory throughput can be increased without sacrificing its latency. This may however introduce pipeline interlocks in the sequencer, complicating the software model of the microcode.

The microcode address space is a little too small. Although it will not take more than the four kilowords available to code an SCode interpreter, more space will allow more primitives to be microcoded. If a path from the main memory to the control store is put in, it will even be possible to micro-compile user code. When fast static RAM chips with more storage are available[1], this machine can be redesigned to contain more microcode memory.

We conservatively decided to using a four-layered (with two signal planes plus power and ground) circuit board for laying out the components. This turned out to be extraordinarily difficult, given the number of pin-grid array chips and crossing data buses in the design. Although the conservatism will eventually buy us the ability to correct bugs after the board has been fabricated, we also ended up making its area so large that propagation delay across the traces may become a problem. Not only will we be able to pack the components denser on a six or eight layered board, routing the individual lines and buses will also become much easier.

The layout system is procedurally driven. This makes the layout easy to comprehend, but the code is very tedious to write. A more graphics oriented approach may make the work easier, with the penalty that finished pieces may not be very maintainable. Basically the current layout system is adequate for small designs, but a more robust system is desperately needed for projects of this magnitude.

The simulator has some of the same problems. Entering the schematic procedurally is a mundane process. The simulator should be integrated with the layout system so that even the final layout could be checked. The trace feature was a good idea for complicated multiphase timings like the one in this design, but better triggering control could be added to make it easier to use.

As a result of a lack of time, the SCode abstraction was taken wholesale from previous implementations of Scheme as the instruction set for this machine. While we are

[1] An 8K $\times$ 9 NMOS chip made by Toshiba is expected to be available end of this year.

certainly experienced in implementing SCode and it is desirable in many ways, extensions could be added to make the machine run more efficiently. Future work on this type of architecture must focus on designing an instruction set that fully exploits the architecture and many known techniques of compiling Scheme.

The design of the architecture was based on intuition and on our experience with large programs and LISP implementations. The performance expectations were derived from looking at small sections of frequently called routines and critical paths of the interpreter. No empirical data has been taken, and the actual execution of substantial programs has not been simulated. To prove the merits of the design, extensive tests and benchmarks must be done. A comparison of the general architectural principle against other popular designs is necessary. Such a comparison must try to factor out the technology used and constraints in actual implementations to reveal the full potential of each design. In fact I will be working on a Master's thesis attempting to do just that.

Acknowledgements

I would like to thank Professor Gerald J. Sussman for supervising this project. He helped come up with the basic architecture, wrote the first version of the simulator, and got me started writing the layout system.

Much gratitude is due to Dr. Yekta Gürsel, who laid out the processor board while I coded his often brilliant designs in the layout language. He will also work on the memory board and is the sole reason why this machine may ever actually materialize. He gave many suggestions for improving the layout language and contributed code and algorithms with which I refined the system.

Chris Hanson made numerous important contributions to the Scheme86 architecture. He provided lots of ideas while the sample microcode was written and analyzed. He was always a dependable source of technical and moral support. Jim Miller participated in discussions on the feasibility and practicality of this architecture. He got me started working on the Scheme Team and has always been a friend and a mentor.

I would also like to thank Andy Berlin for taking over my role in laying out the processor after I started writing this paper. He will write the actual microcode for Scheme86. Bill Rozas, the resident Scheme wizard, taught me a lot about the SCode implementation. David Espinosa wrote the first pass of a simplified Scheme interpreter for this machine and together with Oded Feingold worked on an overall design of the memory board layout. Dr. Gürsel and I adapted the CIFP plotting program written by Chris Kingsley to run on local hardware. Robert Armstrong allowed us access to the saddle plotters owned by his research group. Professor Rich Zippel graded and provided valuable comments on an earlier paper on this architecture. I struggled to typeset this document in TeX with help from Panayotis Skordos. Mike Marino worked on the blueprints and illustrations for a modest fee.

Finally I would like to thank my family and friends for their moral support. I am especially grateful to my parents, to whom this thesis is dedicated.

References

[Abelson 1985]
Abelson, Harold, and Gerald Jay Sussman, with Julie Sussman. 1985. *Structures and Interpretation of Computer Programs.* Cambridge, Mass.: MIT Press.

[Batali 1980]
Batali, John, and Anne Hartheimer. 1980. The Design Procedural Language. Memo 598, MIT Artificial Intelligence Laboratory.

[Batali 1982]
Batali, John, Edmund Goodhue, Chris Hanson, Howie Shrobe, Richard M. Stallman, and Gerald Jay Sussman. 1982. The SCHEME-81 Architecture – System and Chip. In *Proceedings of the MIT Conference on Advanced Research in VLSI,* edited by Paul Penfield, Jr. (Dedham, Mass.: Artech House).

[Holloway 1980]
Holloway, Jack, Guy Lewis Steele, Gerald Jay Sussman, and Alan Bell. 1980. The SCHEME-79 Chip. Memo 559, MIT Artificial Intelligence Laboratory.

[Katz 1985]
Katz, Randy H., editor. 1985. SPUR Architecture Design Rationale. Computer Science Division, Electrical and Computer Science Department, University of California, Berkeley.

[Kingsley 1982]
Kingsley, Chris. 1982. EARL: An Integrated Circuit Design Language. Master's thesis, Computer Science Department, California Institute of Technology.

[Knight 1979]
Knight, Thomas F., Jr., David A. Moon, Jack Holloway, and Guy L. Steele, Jr. 1979. CADR. Memo 528, MIT Artificial Intelligence Laboratory.

[Miller 1985]
Miller, James S. 1985. Ph. D. Area Examination paper. Department of Electrical Engineering and Computer Science, MIT. Private communications.

[Moon 1985]
Moon, David A. 1985. Architecture of the Symbolics 3600. Submitted to the 12th IEEE International Symposium on Computer Architecture, April 3, 1985.

[Rozas 1984]
Rozas, Guillermo J. 1984. Liar: An Algol-like Compiler for Scheme. S.B. thesis, Department of Electrical Engineering and Computer Science, MIT.

[Steele 1979]
Steele, Guy Lewis, Jr., and Gerald Jay Sussman. 1979. Design of LISP-based Processors or, A dielectric LISP or, Finite Memories Considered Harmful, or LAMBDA: The Ultimate Opcode. Memo 514, MIT Artificial Intelligence Laboratory.