Massachusetts Institute of Technology
Artificial Intelligence Laboratory

Working Paper 225 January 1982

The Assq Chip and Its Progeny

Philip E. Agre

Abstract

The Assq Chip lives on the memory bus of the Scheme-81 chip of Sussman *et al* and serves as a utility for the computation of a number of functions concerned with the maintenance of linear tables and lists. Motivated by a desire to apply the design methodology implicit in Scheme-81, it was designed in about two months, has a very simple architecture and layout, and is primarily machine-generated. The chip and the design process are described and evaluated in the context of a proposal to construct a Scheme-to-silicon compiler that automates the design methodology used in the Assq Chip.

Acknowledgements. Clark Baker, John Batali, Howie Shrobe, and Gerry Sussman were extremely helpful during the design of the Assq Chip. Daniel Weise and John Batali read and commented upon various drafts, but the remaining obscurity is my fault.

Overview

The Assq Chip is an nMOS chip, approximately 6mm x 6mm in size, which is capable of computing a number of common Lisp functions associated with the maintenance of linear lists and tables. It was designed as a project for MIT course 6.371 for the Fall 1981 term, as an auxiliary to the Scheme-81 chip of Sussman *et al* (Scheme 1981). It acts as a utility, living on Scheme-81's bus and talking to its list-structured memory. It was motivated by a desire to determine whether the design methodology which produced the Scheme-81 chip could allow a relatively inexperienced designer to produce a complete design from a high-level specification in a short period and to inquire into the possibility of using such a methodology to write a silicon compiler which, given the definition of a simple Scheme function, will produce a chip layout. There is hope on each point. The Assq Chip behaves as a set of memory registers mapped onto Scheme-81's memory that allow a programmer to compute the Lisp functions **assq**, **rassq**, **delassq**, **delrassq**, **memq**, **delq**, and **circularp** simply by placing the arguments in ordinary memory locations and fetching another memory location until a legal result is found. The Assq Chip contains extensive error checking to avoid infinite loops and illegal memory operations and may be interrupted, halted, or redirected to a new problem on any iteration of its algorithm. The main considerations in the design were the short period of time available and the constraints imposed by process and packaging; speed was not an important consideration due to the fact that the clock will be synchronized with that of the much larger Scheme-81 chip. The architecture of the chip is simple, consisting of a finite-state machine implemented with a PLA and one level of subroutining, a small array of flip-flops for internal state, a large data path incorporating much automatic condition-testing circuitry, 60 pads, a small amount of random logic, and a considerable amount of wiring. The PLA and register array were machine-generated, and many routing functions were done automatically. The microcode in the PLA was generated by a small compiler operating on a low-level microcode source language. The input specification for the data path generator was abstracted from a high-level description of the microcode which specified what registers were needed and what sorts of specific operations were required to be performed on each. Placement and routing decisions were dictated almost entirely by the PLA and data path which were by far the largest blocks in the layout. Extensive testing and debugging functions have been incorporated in the microcode, and assuming minimal finite-state machine functionality testing will be done over the Scheme-81's bus by a program written in Scheme. The relative independence of the low-level design details from the functionality of the chip indicates that the methodology employed in its design would, if automated, be the basis of a reasonably powerful Scheme-to-silicon compiler. It was concluded that the design methodology used in Scheme-81 was generalizable at least as far as **assq**, but that it is not, or not yet, reasonable to expect hardware-innocent users to be able to design chips using the existing tools.

The Assq Chip will produce offspring later on this year by means of a Scheme-to-silicon compiler that I propose to write. (Scheme, of course, is a dialect of Lisp distinguished by its use of lexical scoping.) This program will produce chips that have the same basic organization as the Assq Chip and talk to Scheme-81 in the same way, but which implement different functions. The ability to customize the data path introduces some interesting compiler issues, and these are described briefly.

Current State of the Implementation

The chip is described as of the first week in December, 1981. The design appears to pass design-rule and static analyses. The simulation that has been performed to date indicates that the various parts of the chip have reasonable functionality (the finite-state machine really is a finite-state machine and so on), but the fine details of the algorithms have not been simulated due to a shortage of time and the presence of a few unresolved issues with regard to the interactions between the Assq and Scheme-81 chips. The design will be completed and cooked in early 1982.

This Document

This document is meant to serve a number of disparate purposes. First, it is meant as a fairly complete documentation of the Assq Chip project. The various structures and algorithms involved in the Assq Chip are motivated and described in some detail. Second, it is meant as a study in integrated circuit design, one small-to-middling project by a novice. Third, it is meant as a proposal and argument for the construction of a silicon compiler based on lessons learned in the design of the Assq Chip. An attempt has been made to organize this paper so that a reader who is not interested in all three aspects need not read everything. If a paragraph seems boring, it is reasonably safe to go on to the next one. Because the Assq Chip and its prospective progeny are so intimately related to the Scheme-81 chip, this paper contains some rather detailed descriptions of various aspects of the Scheme-81 chip. As of this date, however, the fine details of the design of the Scheme-81 chip have not yet been settled, so a reader wishing the definitive word on the details Scheme-81 chip and system should wait for the final report on that project.

Motivation for the Project

The choice of the Assq Chip as my term project was motivated by two questions of computer science. My hope is to allow ordinary computer users to cause chips to be fabricated by simply entering an editor command over a Scheme function definition. In order for this to happen, we must understand more about high-level VLSI design methodology. The Scheme-81 chip of Sussman *et al*, which was nearing completion in the AI Lab at the time of the start of the Fall 1981 term, had scouted some new territory in high-level design. Both a set of VLSI design tools and something that looked like a design methodology had taken shape and I wanted to determine (1) whether the tools that had been developed would allow a computer person with minimal VLSI experience (such as myself) to design a relatively large chip in a short period, and (2) whether I could use those tools in such a way that the low-level design and layout details were relatively independent of the actual functionality implemented by the chip. If I were successful in the latter, it would follow that by automating what I did and adding a relatively straightforward translator from source code to circuit functionality specification one could build a Scheme-to-silicon compiler.

The design environment. The available design tools fell into three groups. At the high-level end were two programs to automatically generate various circuits according to high-level specifications, Howie Shrobe's data path generator (Shrobe 1982) and John Batali's PLA generator (Batali 1982). Both are immensely powerful and my design made only fairly rudimentary use of each. At a lower level were two well-developed design tools, Shrobe's Daedalus graphic editing system and the Design Procedure Language for circuit description (DPL 1980) (DPL/Daedalus 1980). These two programs build in turn on the advanced interactive capabilities of the Lisp Machine. Also available were a number of low-level design aids including Daniel Weise's node extractor and Ned Goodhue's adaptation to the Daedalus environment of Clark Baker's design rule checker. Heavy use was not made of these tools because their development was not complete and because the ease of sending CIF over the ChaosNet from the Lisp Machine to machines which had more highly developed tools made them dispensable until they are completed and integrated into the Daedalus environment.

Scheme-81 is a system. Scheme-81 was designed to make use of "hardware subroutine" utility chips such as the Assq Chip and to support general parallel computation, both among utility chips on a single Scheme-81 chip's bus and among different interconnected Scheme-81 machines (Scheme 1981). Machines would coordinate through ports, implemented as utilities on the busses of two different Scheme-81 machines. The Scheme-81 garbage collector is capable of supporting cross-machine pointer references to allow the sharing of data and the communication of information in s-expression format. Each machine in such an organization would be based on a Scheme-81 chip and specialized by the incorporation of specialized utilities such as the Assq Chip. There is interest in building chips for doing symbolic pattern matching, infinite-precision arithmetic, and other common and useful operations. Indeed, some of these will be generated by the proposed silicon compiler described at the end of this paper. Even aside from the possibilities for parallelism

amongst utility chips, the main justification for having specialized chips to perform common operations is the gain in speed and storage utilization efficiency that would result from eliminating the overhead of Scheme-81 interpretation (Scheme-81 has no compiler). The specialized design of the Assq Chip and its brethren will allow all computation aside from the memory operations absolutely necessary to manipulate the list-structured data to be hard-wired, thus allowing bus-utilization efficiency to approach 100 percent (a condition more humbly known as the von Neumann bottleneck).

Generalizing to silicon compilation. There have been many proposals for silicon compilers. The approach to the problem I decided to explore involves a high-level source language, the Scheme language in fact, extremely simple and regular chip architectures, mostly machine-generated, and an absolute minimum of complicated wiring and random logic. A major advantage of this approach is that techniques for translating symbolic descriptions of process and data into specifications at the level of input to a PLA generator or data path generator are well-developed and widely understood. Such techniques include register allocation, peephole optimization, source-to-source translations, modular semantic routines, and various other standard methods in the construction of traditional compilers. Optimization techniques derived from traditional Boolean logic design might also be of use. Once a silicon compiler, any silicon compiler, is running, one can experiment at leisure with silicon analogs of these methods as well as develop new ones as suggested by experience. The question is one of trading back and forth between what sorts of representations can be provided by a source-to-middle-level compiler and what sorts of structures can be compiled from middle-level descriptions. The short period of time available to complete the design of the Assq Chip prevented any tendencies to complexify the layout portion of the design in the name of efficiency; it was always a better use of my time to add extra features at the high level than in specific circuitry. Later sections outline the course the design took, describe how one might build a silicon compiler, and consider what the first steps in developing a silicon compiler culture might be.

What It Does - The High Level

The Assq chip is meant as a server for a Scheme-81 chip. It communicates with its owner on two levels: Considered from the level of abstraction at which a user's Scheme programs are to be written, the Assq chip performs a number of common Lisp functions. Considered from the level of abstraction at which the bus protocols operate, the Assq chip supports most but not all of the features of the Scheme-81 bus.

Choosing the functions. The Assq chip implements seven Lisp functions, commonly called **assq**, **rassq**, **delassq**, **delrassq**, **memq**, **delq**, and **circularp**. This list was chosen in the following way: I wanted to implement a set of functions that formed a module or a complete set of functions for performing some sort of operation or manipulating some sort of data structure. In order to simplify interactions between the chip and the bus, I made the constraint that the functions do no consing and maintain no arbitrary-depth stack. In order to avoid having to handle interrupts, I made the constraint that the algorithms that implement the functions be decomposable into small parts so that the chip could give up control of the bus frequently to allow Scheme-81 a chance to halt or redirect it or to give other utility chips a chance to use the memory. A third constraint was that the algorithm involve no arithmetic that could not be handled by a Shrobe-style data path.

What the functions do. A set of functions that satisfies these constraints is the standard module for handling what are known in Lisp as *association lists* (also known as *assoc lists* or *tables*), lists of **cons** cells each of whose **car** is an atom[1] and each of whose **cdr** is some s-expression which is said to be *associated* with its

---

1. Actually, any sort of data structure can be used, but the equality test employed by the algorithms which search in the tables is **eq**, not **equal**. **Assq** *et al* are distinguished from their cousins, **assoc** *et al* by the use of **eq** rather than **equal**. The reason why my chip does **assq** and not **assoc** is that **eq** can be done with simple combinational logic whereas **equal** requires a general recursion capability if arbitrarily deep structures are to be handled.

respective atom. Given an atom and an association list, **assq(atom,list)** returns the first **cons** cell which associates **atom** with a value on **list**, and **nil** if there is no such cell, **delassq(atom,list)** returns a version of **list** from which the first **cons** cell in **list**, if any, which associates **atom** with some value has been spliced out destructively, and **setassq(atom,list,sexp)** destructively alters the first **cons** cell on **list** which associates **atom** with some value so that it comes to associate **atom** with **sexp** (if no such cell exist it conses a new **cons** pair onto the front of **list**). The analogous "reverse assoc" functions **rassq**, **delrassq**, and **setrassq** perform the same operations on *rassoc lists*, which are like assoc lists except that the atom with which a value is associated is stored in the **cdr** part of each **cons** cell. While such a data structure is sometimes a useful alternative to a straightforward assoc list in its own right, more frequently the reverse assoc functions are used to invert the ordinary assoc functions, so that if one wishes to find the first atom on an assoc list whose associated value is **YELLOW**, one can say **rassq(YELLOW,table)**. The functions **memq**, **delq**, and **circularp** were added in the design when it was noticed that they could be incorporated without any major change to the design or layout, and without violating any of the design constraints. The **memq** and **delq** functions, with **cons**, form a module for the handling of basic list operations[1]. Given an atom and a list, **memq(atom,list)** returns the first sublist of **list** whose **car** is **atom**, and **nil** if **atom** is not an element of list and **delq(atom,list)** returns list with the first occurrence of **atom** spliced out destructively. Given a list, **circularp(list)** returns **t** if **list** is a circular list and **nil** otherwise.

Algorithms - first pass. All of the assoc list and atom list functions are implemented using the obvious linear search algorithms. The **circularp** function and circularity tests done by the algorithms for the other six functions are implemented by means of an algorithm reputedly due to Floyd which works as follows: two pointers, A and B begin at the first **cons** node in the list. On each operation, A is moved ahead in the list by one list element and B is moved ahead by two list elements. If A and B ever come to point at the same cell after the start of the algorithm, this means that B has traversed the circular portion of the list and has "lapped" A. If on the other hand the B pointer ever becomes anything but a **cons** cell, then the list is not circular. It is easy to show that all of these algorithms always terminate within time linear in the number of list elements. In order to minimize the complexity of the chip only those parts of the various algorithms which are repeated for each element of the list are implemented, so that, for example, once the algorithm for **delrassq** finds the element of the list to splice out, instead of doing the required **rplacd** operation itself it simply returns the previous sublist and allows the calling Scheme-81 code to do with the result as it wishes.

Formal definitions. In lieu of a formal semantic definition of the various functions implemented by the Assq Chip, the equivalent Lisp code is given in Appendix One. (In this code, a function name that begins with a single asterisk refers to a function actually implemented on the Assq Chip, while a function name that does not refers to a function which is part of the module as presented to the user.) To Scheme-81, the *-functions are just memory locations that can be set with **rplaca** and read with **car**. Appendix Two defines the functions that will actually be employed by Scheme-81 to call the Assq Chip. (In each definition there, atoms whose printed representations begin with two asterisks refer to **cons** cells upon whose **car** pointers the various operations of the Assq Chip are memory-mapped.)

What It Does - The Low Level

The bus and the system. Below the level of Scheme algorithms, the Assq Chip must be able to communicate with Scheme-81 and with the memory using Scheme-81's bus, the *sbus*. The Scheme-81 chip is designed to be only one part of a complex computer system. Each Scheme-81 chip has an sbus, which appears to the Scheme-81 chip's bus protocol mechanisms to be a simple memory bus but which has in addition to actual memory units an arbitrary number of servers, such as Assq chips or pattern-matcher chips.

---

1. Note that **memq** and **delq** are to **member** and **delete** as **assq** and **delassq** are to **assoc** and **delassoc**.

Each Scheme-81 - sbus - memory - server collection is called a Scheme-81 computer. Up to 31 such computers can be connected together into a network by means of *portals*, dual-ported servers on the various sbusses which recognize requests for memory operations on memory words belonging to other computers and route those requests to other sbusses, all the while pretending to be memory units or servers on all the busses to which they are connected. By means of portals the various computers in a network can share list-structured data simply by pointing at it. The bus protocol is designed to allow the garbage collectors on the various Scheme-81 chips in a network to cooperate in following inter-computer pointers. Because it does no consing, the Assq chip does not have to worry about the garbage collector[1]. Inter-computer memory references are transparent to the Assq chip.

Sbus organization. The sbus has a simple linear organization, with a single Scheme-81 chip which is in charge, memory units which respond to addresses within their respective ranges and an arbitrary number of utility chips such as the Assq Chip. Each operation that can be performed by some utility chip is mapped onto a memory address, allowing any chip to make a request of a utility chip by pretending that it is performing a memory operation. (In this way the bus structure is similar to that of the I/O bus of the PDP-11.) Although any number of chips on the sbus can be active at a given time, the sbus always belongs to exactly one of them, and only that one may make requests on the sbus. By convention, a utility chip that does not handle interrupts (none discussed in this paper do) gives up the sbus from time to time to allow others to make use of it.

The sbus according to the master. To Scheme-81 or any other chip wishing to make use of utility chips, the bus appears as a large memory, segmented into up to 32 parts by the uppermost five bits of every 29-bit address. The segment codes are known as "machine numbers" because the memory addresses on a given Scheme-81 chip's sbus must all have as their upper five bits a code which distinguishes that Scheme-81 chip from its brethren on other sbusses. Memory references across segment boundaries are perfectly legal and handled properly by the garbage collector, but are somewhat expensive and are not intended to be used liberally. The exception to these rules is the memory segment for machine number -1, that is, 11111-base-2. This segment is reserved for the memory-mapped addresses of the various functions performed by utility chips. The "-1 space" addresses, as they are known, are local to each sbus and the garbage collector ignores them. A Scheme-81 or other calling chip sends data to a utility chip by performing a memory write operation on a -1 space address recognized by that chip and fetches data from a utility chip by performing a read operation on another such address. In the case of the Assq Chip, a two-argument function call is made by writing the first argument to an address which causes the Assq Chip to load its first-argument register (called *atom*) and then by writing the second argument to another address that corresponds to the desired function, whereupon the chip commences processing by attempting to gain control of the sbus.

The sbus according to the slave. To an Assq Chip waiting to be told what to do, each address going by on the bus appears to be made up of a machine number (it is looking for -1), 14 bits of "chip type" code (one particular chip type code means "Assq Chip"), five bits of "identity" code (each of up to 32 Assq Chips on an sbus has its own identity code), and five bits of "opcode" (there are 32 different operations that can be performed by each Assq Chip, a few of them illegal). The input value of the cdr bit is always ignored by the Assq Chip. Each Assq Chip, on each phi1 when -read-mar or -write-mar is low (that is, a memory read or write operation is being attempted by some chip on the sbus) and the chip is waiting to be told what to do, tests whether the topmost five bits are all 1's, whether the next 14 bits have the appropriate values, and whether the next five bits match the inputs from the lines coming from the five identity pads. If all of these conditions hold, then on the succeeding phi2 the PLA will dispatch on the lowest five (opcode) bits and

---

1. Were a garbage collection to happen with an Assq Chip function in progress, the pointers kept by the chip would not be relocated and disaster would follow. The Scheme-81 software is aware of the problem and can choose among a number of painless methods of avoiding it. Anyone interested in this corner of Scheme-81 theory and practice should wait for the final report on Scheme-81.

perform the appropriate actions.

Assq and the outside world (begin gory details). The Scheme-81 sbus, for present purposes, has 44 lines, including 37 data lines (7 bits of type field, 29 bits of address, and 1 cdr bit) and lines for the bus protocol called -read-mar, -write-mar, -read-mbr, -write-mbr, -dma-request, -sbus-error, and -wait. All of these lines are high by default, and a chip "asserts" one of them by pulling it down. The Assq chip has one pad for each of these bus lines. Part of the bus protocol but not part of the bus itself are the lines that connect to the pads dma-grant-in-pad and dma-grant-out-pad; these implement the daisy-chaining mechanism used by Scheme-81 to give control over the bus to a utility chip which is requesting DMA (direct memory access). Five pads of the Assq Chip are set by DIP switches to give each of a possible 32 Assq Chips on a given Scheme-81's bus a separate identity. Finally, there is an init-pad, which should be driven high for one clock cycle whenever the Assq Chip needs to be initialized or re-initialized.

Gaining control of the sbus. For a utility chip such as the Assq Chip, the protocol for obtaining direct memory access (DMA) privileges is: First it must wait for the -dma-request sbus line to go high, indicating that all utility chips which requested DMA the last time around have finished and control has been recovered by Scheme-81. Then it lowers the -dma-request sbus line by turning on its dma-request flip-flop, whose output sense line is connected directly to the output-enable line of -dma-request-pad, the output-data line being wired to GND. When Scheme-81 wishes to allow those who have made DMA requests to be granted them, it raises its dma-grant-out line, which causes a high input on the dma-grant-in-pad of the first utility chip in line. (A strict priority among utility chips is maintained by this daisy-chain ordering.) When a chip receives a high signal on its dma-grant-in-pad it will allow the signal to propagate to the next chip along unless it had requested DMA privileges by lowering its -dma-request line. A chip that requests DMA privileges and is granted them may use the sbus for as long as it wishes (although courtesy requires moderation); when it is done it ceases pulling down its -dma-request line and the grant signal is thereby allowed to propagate to the next chip along. Because this propagation of DMA grant signals must proceed with great speed, each chip which supports it must have two gates of random logic to constantly compute the relation dma-grant-out-pad := and(dma-grant-in-pad, not(-dma-request)).

Memory read protocol. The bus protocol for memory read operations (**car**s and **cdr**s) is as follows: First the requesting chip, A, gains control of the bus by doing a DMA request in the manner just described. Then on some phi2 A lowers the -read-mar line (for one cycle only) and sets the bus address/data lines to the address to be read, including the **cdr** bit. The memory unit or utility chip upon which the read is occurring, B, recognizes this signal on phi1, pulls down the -wait line, and computes the answer. When the answer is ready, B lets the -wait line go high and assigns the address/data lines to whatever the answer is, not including the **cdr** bit. Meanwhile, upon ceasing to pull down -read-mar, A has lowered the -read-mbr line, letting it raise one cycle after the rise of -wait, having read in the answer. During the time when A is holding down the -read-mbr line, it is latching the contents of the address/data lines on every phi1, on the chance that the real answer is there. If an error of some kind occurs within B, it lowers the -sbus-error line in addition to setting the address/data lines; A must check for this.

Memory write protocol. The bus protocol for memory write operations (**rplaca**s and **rplacd**s) is: First the requesting chip, A, gains control of the bus by doing a DMA request in the manner described above. Then on some phi2 A lowers the -write-mar line (for one cycle only) and sets the bus address/data lines to the address to be written to, including the **cdr** bit. The memory unit or utility chip upon which the write is occurring, B, recognizes this signal on phi1 and pulls down the -wait line. On the following phi2 A lowers the -write-mbr line and sets the address/data lines to whatever is to be written in the word it has specified, not including the **cdr** bit. It maintains these values until it finds -wait high again on some phi1, whereupon it checks to see if B has generated an sbus error.

Other sbus features. The bus protocol can also handle everything needed to do garbage collecting and the creation of new cons nodes (which is done by Scheme-81), but the Assq Chip supports neither of these features.

Sbus function calling protocol (one step up). A great deal of complexity is introduced into the Assq Chip's

algorithms for bus line handling because at different times it must play both memory and user of memory. In fact the Scheme-81 bus protocol has a subtle flaw in this respect: if Scheme-81 were to make a request to a utility chip to, say, look up entries in a data base, and if this chip were in turn to make a request to the Assq Chip to, say, retrieve a variable binding, the Assq Chip would not be able to hold down the -wait line while working and then allow it to rise when done because this would be misinterpreted by Scheme-81 as meaning that the lookup chip itself was done. There is no clean resolution to this problem, especially given current constraints on the number of pins an integrated circuit's package can have, and so a convention has been established that utility chips such as the Assq Chip must provide an answer to all requests on the very next phi1 after they are called. Since the Assq Chip in general needs to perform DMA operations and the like, this means that a request to an Assq Chip from another chip that cannot be satisfied immediately must be answered with a "not ready yet" code (which in the case of the Assq Chip is an object of the UNBOUND type). The requesting chip must make periodic result requests until a legal value is returned. All of this, of course, can be made invisible to the ordinary user.

A utility chip has two modes. To summarize so far, the Assq Chip has two operating modes, a passive mode in which it waits for commands from Scheme-81 in the form of "memory requests", and an active mode in which it performs some useful work. During the active mode it has control of the sbus. Its state alternates between active and passive with some frequency to allow others a chance to use the bus, but if there is still more work to be done it will try to get the bus back as soon as it is legal to try. Once it has control of the bus it turns the bus protocols around and makes memory requests of its own. Although in principle it could perform any bus operations when it is in its active mode, it actually has need only for memory reads. Thus the Assq Chip supports the slave end of memory read and write operations and the master end of memory read operations.

Important opcodes and others. All 32 possible five-bit opcodes are recognized and handled by the Assq Chip. (Although 64 opcodes could be coded by using the cdr bit, this is not done in order to avoid undue complexity.) Thus the Scheme code to do a memory write on the virtual memory location corresponding to one of these opcodes would be **(rplaca **assq-op-opcode)** and the Scheme code to do a memory read on the location would be **(car **assq-op-opcode)**. The opcodes fall into six groups. A set named table-loc, atom-loc, and so on for each register are virtual memory locations which can be read to retrieve the value of the corresponding Assq Chip register or written to store a value in that register. A set named assq-op, rassq-op, and so on after the various functions the chip can perform are the virtual memory locations into which a table is written to begin processing. (The calling Scheme program should take care to have put a legal value in the atom register beforehand by doing a write on atom-loc.) A read on result-op returns either the result of the last function call to the chip or a not-ready value. Int-assq-op, int-rassq-op, and so on are interruptible versions of the standard function ops which differ only in that if a result-op request is made before the answer is ready the current contents of the table register are returned anyway; this is just for debugging. Also for debugging are the ops called dma-request-loc, working-loc, and so on after the various internal state flip-flops. Doing a read on one of these virtual memory locations returns either t[1] or nil according to whether the flip-flop in question is in state 1 or state 0. Doing a write on one of these virtual memory locations inverts the state of the relevant flip-flop (the data for the write operation is ignored for simplicity, though it must, of course, be provided). The debugging ops allow all testing of the Assq Chip to be done over the sbus by a Scheme program running on Scheme-81. The remaining three opcodes are illegal. Attempting to perform a read or write on an address corresponding to an illegal opcode will result in the generation of an sbus error, as will attempting, for example, a write operation on the address corresponding to result-op, which is defined only for reads.

---

1. Actually not t but a fixnum that serves the same purpose.

Design Considerations

Constraints on the design process. This design project had an unusual set of constraints. The primary constraint was time: the functional specification was laid out in early October and the final deadline was in early December, leaving eight weeks to design a chip of very great functional complexity. An early constraint that the design fit within a 2mm x 4mm portion of a multi-project chip was abandoned early on as impossible. However, the ultimate packaging limitation of 6mm x 6mm in size and 64 pads at most 16 to a side was more rigid. Unlike most design projects, the speed of the circuit was not a major consideration in the design of the Assq Chip because for reasons of bus protocol timing the chip is tied to Scheme-81's clock. Since Scheme-81 is a much larger chip with, among other things, a 30-bit carry chain, it was assumed that the Assq Chip could be made no slower than Scheme-81 by paying attention to only the grossest details of timing and by performing such local optimizations as trying to turn long polysilicon and diffusion wires into metal.

Under pressure, complexity flows upward. This combination of design constraints made a highly regular design not only desirable but necessary if the project was to be finished. The collection of available tools (listed above) immediately dictated the outlines of the design. As a result, there were only two decisions in the entire design process that could be considered major: the selection of the functionality and a decision to change the chip architecture to the simplest possible form from one that had seemed more efficient and aesthetically appealing at the start. Another result was that the wiring became largely modular, as sets of wires went from arrays of starting points to arrays of ending points in "ribbons". Although an automated router existed, I did the routing myself for compactness, since the design looked like it would just barely make 6mm x 6mm. Whenever faced with a choice between adding a feature to the physical layout of the chip or to the more abstract structures from which the machine-generated parts of the layout were created, it was always easier to go the latter route. An extra PLA output line or even an extra register was preferable to an extra gate of random logic in terms of the efficient use of my time, and the design constraints made this decision less reprehensible than it would have been otherwise. The abstract is better understood than the concrete.

Architecture -- Summary

The architecture is trivial. The Assq Chip has two major components, a finite-state machine incorporating a one-deep stack and an array of flip-flops for maintaining internal state, and a data path incorporating a number of condition tests on the registers and the bus.

Finite-state machine statistics. The finite-state machine consists of a PLA with 32 input lines, 52 output lines, and 140 minterms among 20 state labels. The stack is an array of 5 single-bit load-read Mead&Conway register cells. There are 9 set-clear Mead&Conway register cells which serve as internal state; the 18 inputs and 9 outputs are finite-state machine outputs and inputs, respectively. There are control lines from the PLA which determine whether to push a state on the stack and whether to take the next state from the stack or from PLA outputs.

Data path statistics. The data path is an array of five 36-bit registers with buffered control lines. Predicate circuitry constantly provides *listp* tests on three registers (i.e., equal-to-constant tests on the type fields of those registers) and equality tests between two pairs of registers. In addition, various circuitry tests the incoming bus lines to determine if a request is being made of the Assq Chip by Scheme-81 (or, perhaps someday, some other device on Scheme-81's bus).

Clocking. The chip uses a two-phase non-overlapping clock. By Scheme-81 convention, state changes occur on phi2 and everything has a meaningful state on phi1. In particular, bus operations happen on phi1, and everything in the register array changes state on phi2. Thus a bit of state code making its way around the finite-state machine loop might go out the push state output field of the PLA, which is clocked on phi2, into the stack, whose input is clocked on phi1, through one of the stack register cells, which pass data between their inverters on phi2, and into the input of the PLA, which is clocked on phi1.

The Finite-state Machine

Control on the Assq Chip is provided by a finite-state machine made of a PLA with 32 input lines, 51 output lines, and 140 minterms, a one-deep stack, and an array of 9 flip-flops for internal state. The PLA was generated by Batali's PLA generator from a specification provided by a simple compiler from a fairly low-level microcode language. The stack and flip-flops were custom-designed although they are not unusual in their design or layout. This section discusses the process by which the original high-level description of the algorithm was converted into the final finite-state machine circuitry. The retelling will be somewhat idealized; one major false start will be left out. The reader should consider at each point the plausibility of automating the design process described here.

Algorithm to PLA. The translation from algorithm to PLA proceeded in four steps, two manual and two automated. The design began with a description of the algorithms to be performed in an informal Algol-like notation. From this description was taken a list of all the registers and internal state bits that the algorithm required. The algorithm was then rewritten so that memory operations alternated with other computations such as tests and the setting and clearing of flip-flops. From this description was extracted a list of the operations that the various registers had to be able to perform between each memory reference. Memory operations, such as "take the **cdr** of the *table* register", were subroutinized. The algorithms were then rewritten again in the source microcode language. A compiler was run on this code to tabulate some lists, do some error checking, and remove some syntactic sugar to produce the object microcode language. Another compiler was then run on the object microcode language to produce input to the PLA generator. The details follow.

The algorithm - bear with me. The original high-level algorithm was no more than the iterative versions of **assq** and **circularp**; the other functions were included because their algorithms were slight variants on that of **assq** that could be encoded in various flip-flops: the seven functions chosen "ring changes" on the steps of the **assq** algorithm. After the seven functions had all been integrated, the algorithm consisted of a loop with a "top" and a "bottom". In between was the code that was to be executed on each iteration of the algorithm; if a given table is 47 elements long, then up to 47 iterations will be required to finish the operation. The algorithm begins each iteration by gaining control of the bus (through the DMA protocol) and pushing forward by two (when possible) the *ilreg* register, which serves only as the "faster" of the two pointers in the circularity testing algorithm. There are three cases: *ilreg* hits a non-**nil** sublist (in which case the list is not circular), comes to be equal to the *table* register (in which case it is; this will cause an appropriate value to be put into the *table* register, which by convention is where answers come from), or neither (in which case the algorithm proceeds normally). Then the *circularp* flip-flop is tested to see if that is all that is required; if so the algorithm goes to the bottom, where the *table* register is **cdr**'ed, the bus is given up, and control returns to the top. If the function being executed is not a circularity test and if *table* is not discovered to be either empty or circular then the body of the algorithm executes. The body does one iteration of **assq**, **delassq**, **rassq**, **delrassq**, **memq**, or **delq**, depending upon the settings of three flip-flops called *shallowp*, *car-version*, and *deletep*. The body of the algorithm is: set the *table-element* register to the **car** of the *table* register, if not-*shallowp* then set the *table-element* register to its (if *car-version* then **car** else **cdr**), if the *table-element* and *atom* registers are **eq** then we have won, so move the appropriate result value (which can be **t**, **nil**, the *table* register, or the *previous-table* register depending on the various flip-flops[1]) into the *table* register, otherwise if *deletep* is on then move *previous-table* into *table*, and then go on in any event to the bottom of the loop, where *table* will be **cdr**'ed and the iteration will end with the surrendering of the bus. If at any point a result value is moved to the *table* register a flip-flop called *working*, which indicates that useful work is still to be done, is cleared, the bus is surrendered, and control is returned to the top of the loop. If any of the memory operation subroutines gets an error, it is assumed that the *got-sbus-error* flip-flop is set (by the memory operation

1. There are lots of special cases here; see the more formal definitions in Appendix One for more details.

subroutines) and the algorithm proceeds as if a value were ready to be returned. The code at the top of the algorithm that listens to the bus, waiting for commands and setting registers and flip-flops when they arrive, is totally magical at this point in the design process, as is all of the mechanics of how bus operations happen, in particular memory fetches and the getting and giving up of the bus. All reasoning about the functionality of the chip is done at this level; all else is compilation by hand and program. From this code, a list was made of the registers, register tests, and flip-flops required:

table -- the table argument to every function
atom -- the atom argument to the searching functions
table-element -- the **car** of table on each iteration
previous-table -- the previous value of table, used by deletion functions; cdr(previous-table)=table
ilreg -- "fast" table register for doing the circularity test

listp-table -- does the table register point to a **cons** cell?
listp-table-element -- likewise for table-element (error checking)
listp-ilreg -- likewise for ilreg
table-element=atom -- have we found what we're looking for?
ilreg=table -- is the table actually circular?

circularp -- is the request a **circularp** call?
shallowp -- if not, is it either **memq** or **delq**?
car-version -- if not, does it work with **car**'s or **cdr**'s?
deletep -- is it a deletion operation?
first-elementp -- is this the first iteration on this job?
working -- is there useful work to be done?
interruptible -- can partial results be returned?
got-sbus-error -- have I gotten an sbus error from memory?
dma-request -- am I trying to get the bus?

The first five of the flip-flops are dependent on the algorithm being implemented in the chip, but the last four are not.

Abstraction of memory operations. Five different memory operations must be performed in different places in the code; each of these was made into a subroutine so as to make the code halfways manageable: *ilreg* := **cdr**(*ilreg*), *table* := **cdr**(*table*), *table-element* := **car**(*table*), *table-element* := **car**(*table-element*), and *table-element* := **cdr**(*table-element*). Each iteration of the algorithm is seen to be an alternation of memory operations and tests and dispatches on various conditions, both of the flip-flops and of the registers. Only when the contents of the *previous-table* register are moved into the *table* register or vice-versa need anything happen in the data path besides memory operations. Even these movements could have been done without, given a bit of microcode and data path hackery, but this is the sort of complexity that doubles the error-proneness of hand-coded microcode while one's back is turned. (It is also what computers are for.) In principle, though, the algorithm could do all the data path transformations and condition tests it needed in-between or in parallel with bus operations.

You look like you could use an example. Let us consider what happens according to the analysis so far when the Assq Chip is asked to compute **(assq 'b '((a . x) (b . y) (c . z)))**. (Strict type checking is maintained throughout, but most of it will be suppressed here for clarity.) The algorithm wakes up at the top of the loop, the (as yet unanalyzed) bus operations having arranged for **b** to appear in the *atom* register and for **((a . x) (b . y) (c . z))** to appear in the *table* register and the *ilreg* register as well. Flip-flops: *circularp* is 0 because it isn't a **circularp** operation, *shallowp* is 0 because it isn't a **memq** operation, *car-version* is 1 because it isn't a **rassq** operation, *deletep* is 0 because it isn't a **delassq** operation, *working* is 1 because the job is not yet done,

interruptible is 0 because we're not in a debugging mode, *first-elementp* is irrelevant because it isn't a deletion operation, and *got-sbus-error* and *dma-request* are part of the low-level bus protocols and effectively hidden by subroutines. Control of the bus is obtained, possibly after a few cycles. On the first iteration, *ilreg* is set to **((b . y) (c . z))** and then **((c . z))** without becoming eq to *table*, so the table isn't circular (yet). *Table-register* is set to **(a . x)**, the car of table. Because it is a list (it would be an error otherwise) and because *shallowp* is 0 and *car-version* is 1, *table-register* is set to **a**, its car. But *table-element* and *atom* aren't eq, so *table* is set to **((b . y) (c . z))**, its cdr, (*first-elementp*, incidentally, is set to 0), the bus is given up, and control is returned to the top of the loop. Because *working* is still 1, the top of the loop tries between requests to get ahold of the bus, and assuming that it receives no commands to change the chip's internal state (as it might if, for example, the calling program has gotten tired of waiting and decided to clear *working* or set *table* to **nil**) it eventually gets ahold of the bus again. *Ilreg* is set to **nil**, its cdr, and then ignored, as it is thereby established that the table was not circular. As before, *table-element* is set to the car, **(b . y)**, of *table*, and then to its own car, **b**. Since *table-element* and *atom* is at this point discovered to be eq, the *working* flip-flop is set to 0, the bus is given up, and the chip sits back and waits for the answer, **((b . y) (c . z))**, to be requested. Quite a bit happens between successive memory operations, but the bus rarely gets a rest because tests are performed instantly and because large numbers of flip-flop and data path operations can happen at once without interfering with one another. The algorithm can be analyzed in terms of what needs to happen in each interval between successive memory instructions, and it turns out that each set of operations produced by this analysis can be performed simultaneously will little fuss.

The source microcode format. After this analysis, the algorithm was recoded in the microcode source language. A program in the microcode source language is a list of instructions and commands, where an instruction has the format:

(state predicate where-to ups-and-downs), where
state is either X (= don't care) or a state mnemonic like SEARCH3
predicate is a list of conditions, where a condition is
-- the name of an input sense line, like ilreg=table
-- the form (OPCODE op-mnemonic)
-- (NOT condition) for the negation of one of those two
where-to is made up of one or more of
-- NEXT state, meaning go to this state next
-- CALL state THEN state, meaning call here returning there
-- RETURN, meaning do a popj from the stack
ups-and-downs is a sequence of commands to turn outputs on or off
-- (+ line line line) turns those lines on the next cycle
-- (- line line line) turns those lines off the next cycle

The commands are only used to relieve some of the drudgery of microcoding; the various command types are:

(MN-ON state-mnemonic) = use this as the default NEXT
(MN-OFF) = no more default NEXT
(MC-ON input input) = make all conditions use these by default
(MC-OFF input input) = turn off some default conditions
(M1-ON output output) = make all instructions turn these on by default
(M1-OFF output output) = remove some default on outputs
(M0-ON output output) = make all instructions turn there off by default
(M0-OFF output output) = remove some default off outputs

Originally, the high-level code was compiled directly into this format. Eventually I gave up doing it that way because in the limited time available it was not possible to write a compiler able to produce sufficiently compact code and because the functions were so small that they could be easily hand-coded. No attempt was made to write the microcode source in a way that might have been generated by a compiler, on the grounds that writing a compiler would have been easier. Hand-coding actually produced worse code than a properly written compiler would have, largely because there are optimizations available which increase bus-utilization efficiency and decrease the total number of instructions but which produce code that is hard to read and impossible to modify by hand. One such optimization is to move the data path commands from the first instruction of a subroutine to each instruction that calls it, redirecting the call to the second line of the subroutine. This allows the subroutine to begin using the bus immediately, the initial conditions for doing so having been computed "during" the jump. Deciding when and how such optimizations can be done depends upon, among other things, a representation of the timing and ordering constraints on the various actions, and such information is not represented explicitly in any of the algorithm notations I defined.

Translation to a more primitive form. A small compiler was constructed to take this source microcode and remove the syntactic sugar to produce a uniform, order-independent microcode format which maps directly onto the fields of the PLA. The object microcode is organized as a simple list of instructions, each of which is made up of various PLA fields, in this format:

(state pred popjp nextp next holdp pushp push actions), where
the instruction fires if the current state is right and pred holds
state is a state mnemonic, such as SEARCH3
pred is a list of conditions, like before
popjp says whether to get the next state from the stack
nextp says whether to get the next state from the PLA output
next is a state mnemonic for where to go to if nextp is on.
holdp says whether the stack should keep its current contents
pushp says whether to take new stack contents from the PLA output[1]
actions is a list of output lines to turn high (the rest go low)

A program was run on the object microcode to make sure that every possible combination of inputs to the PLA would be handled by some microcode instruction, so as to avoid having the chip shut down completely in case of a design error or electrical glitch[2].

And then to the PLA. The object microcode instructions were translated into the input format of the PLA generator by yet another program, which assigned binary values to the opcodes and state names, computed the widths of the various fields, and other such bookkeeping. The result of this process was a list of PLA minterms, organized in the format required by Batali's PLA generator, with 1's and 0's and X's. The format of the microcode word that was imposed on the PLA's bits can be read from the object microcode instruction format:

opcode -- 5 bits, X's if it doesn't matter
input lines -- 22 in number

---

1. Yes, holdp and pushp are always complements of one another. This is a crock to save a gate of random logic.

2. An exception to this is in the case of an illegal microcode state being generated somehow; the extra microcode instructions to handle all the illegal state codes would probably not be such a good investment, because state codes are generated internally in quite well-understood ways. Were general recursion to be implemented, frequent memory fetches to get microcode states would seem like a much better excuse to handle this particular contingency.

current state -- 5 bits

popjp and nextp -- one bit each
next state field -- 5 bits
holdp and pushp -- one bit each
push state field -- 5 bits
output lines -- 37 in number

The orderings of the input and output lines to and from the pads and data path and flip-flops and so on was given by a collection of lists, which had the effect of further decomposing the microcode word so that the input sense lines fell into the following fields:

input lines from various pads -- 6 in number
sense lines from the data path -- 7 in number
sense lines from the flip-flops -- 9 in number

and the output control lines fell into the following fields:

control lines for various pads -- 6 in number
alternating clear and set lines for the flip-flops -- 18 in number
control lines for the data path -- 13 in number

The lists of 0's, 1's, and X's generated by this last compiler were then fed to the PLA generator, which made the PLA. The various structures created as side-effects by the various compilers were used in deciding where to connect all the wires. Among these wires were control lines that computed the next state, deciding whether to use the next-state field of the output or the contents of the stack and whether to load the stack with the push-state field of the output or to have it maintain its existing value. DPL code was written to route these wires explicitly.

The density of the PLA is not high, about 16 percent. This could presumably be improved by the use of encoders and decoders of the sort created more or less automatically by the PLA generator for Scheme-81. This was not done because of time constraints.

The Data Path

The form of the data path is already decided. It was determined in analysis of the algorithms used in the Assq Chip that there would be required five registers, *ilreg*, *table*, *atom*, *table-element*, and *previous-table*, and various condition tests on the registers and the bus: *ilreg*=*table*, *table-element*=*atom*, listp-*ilreg*, listp-*table*, and listp-*table-element*. The rest is easy; these specifications (together with the magic required by the bus-watching microcode to notice that a request is being made of it) were converted into source code for Shrobe's data path generator; this code is in Appendix Three. The data path generator returned a list of all the places where wires needed to be connected; this list was used by the various routing routines:

Input control lines driven by the PLA
-- Write the bus into the table register
-- Read the table register onto the bus
-- Read **nil** onto the bus
-- Read **t** onto the bus
-- Read **unbound** onto the bus
-- Read the ilreg register onto the bus

-- Write the bus into the ilreg register
-- Write the bus into the table-element register
-- Read the table-element register onto the bus
-- Read the atom register onto the bus
-- Write the bus into the atom register
-- Write the bus into the previous-table register
-- Read the previous-table register onto the bus

Unused input control lines tied to ground so they won't float
-- Load the table register from its local bus
-- Load the ilreg register from its local bus
-- Load the table-element register from its local bus
-- Load the atom register from its local bus
-- Load the previous-table register from its local bus
-- Freeze all data transactions within the data path

Input lines to be sent to pads
-- 36 bus lines (horizontal)
-- 5 identity lines (vertical)

Output sense lines used as inputs to the PLA
-- The proper identity code appears on the bus lines
-- The proper chip-type code appears on the bus lines
-- table-element=atom
-- listp-table-element
-- listp-ilreg
-- ilreg=table
-- listp-table

The data path generator can do obvious things easily. The important point here is that once the algorithm had been written in a high-level form, a first version of the data path was fully generated within half an hour. Although a data path designed manually would probably have been more compact, it is doubtful whether its design could have been changed as radically as the design of the Assq data path was between its various incarnations with as little difficulty.

A few notes regarding my use of the data path generator:

1. For reasons of routing, the ordering of the bits in the data word was reversed from that of Scheme-81; here bit 0 is at the top near the input drivers and bit 35 is at the bottom. This change was implemented by simply rewriting the constant definitions at the top of the data path generator's input file.

2. Because it turns out that at most one register-to-register data movement happens between any two memory operations in the Assq Chip algorithms, the local-bus feature of the data path generator is not used. Time permitting, the library of cells used by the data path generator could have been expanded to include cells without circuitry to handle the local busses. This could have been done without changing the data path generator at all, and would have resulted in a somewhat smaller data path.

3. Another result of the algorithms' not using all of the features provided by the data path is that some of the control lines are not driven by outputs from the PLA. These lines are grounded so as to prevent spurious data path operations due to random electrical floating.

4. Unlike Scheme-81, the Assq data path has no 37th row for the cdr bit, due to the fact that the latter is only used when requesting a car or cdr from the memory. Instead, pad for the cdr bit is driven directly by an output from the PLA.

Placement and routing

Almost all of the placement decisions in this design were dictated by the PLA and the data path, which were the two dominant blocks in the design. Given their shapes, there were only two plausible placements for these two blocks, the one chosen with the PLA "pointing down" and the same one with the PLA "pointing up". Subjective ease-of-routing was the determining factor in the choice between the two. The objects to be placed and routed among were:

the finite-state machine: the PLA, 5 stack cells, 9 flip-flops
the data path
60 pads: 37 bus lines, 9 bus protocol pads, 5 chip identity pads, 2 clock pads, 4 GND pads, and 3 VDD pads
2 gates of random logic
the logo

The placement of the various parts was done by trial-and-error. The result is reasonably compact, although considerable further local compaction is prevented only by time constraints. The choice of 4 GND and 3 VDD pads was dictated by the unavoidable difficulty of routing the stack and flip-flop power lines, given that they are buried in amongst so much wiring, by the desire to make sure that GND is not allowed to come much above 0V in real operation, and by a general policy of extreme conservatism in all electrical matters.

Wiring was primarily regular. All but a few of the wires were routed in "ribbons", i.e., collections of wires which go from consecutive points along some row of objects to consecutive points along some other row, or "rings", i.e., wires tracing most or all of the outer circumference of the circuit in the vicinity of the pads. The ribbons, numbering 15 and averaging about 8 wires, were each implemented by means of a DO loop in the DPL code for the top-level circuit. The rings correspond to the signals vdd, gnd, phi1, phi2, bus-input-enable, and bus-output-enable, and each was implemented by means of its own DPL WIREQ command. The random logic wires, the power, ground, and clock lines, and the lines connecting the PLA's bus-input-enable and bus-output-enable outputs and the clock pads' inputs to their respective rings were routed individually.

Parameterized wiring doesn't get in the way. The wiring programs, written in DPL, were parameterized so as to allow wires to stretch or shrink with movements in the various components of the design. An almost-general-purpose program was written to make the connections between the 16 pads on the right-hand side of the chip and the 16 corresponding bus line connections on the right-hand side of the data path. Much of the rest of the routing was done with the aid of a "careful router", a set of alternative versions of the DPL wiring commands which try to run a line in metal as much as possible but which avoid collisions by moving a new metal line into poly when it approaches an existing metal line. This was useful because the various ribbons had much opportunity to cross one another, and explicit programming of the jumps to poly was very difficult.

Random logic and logo. The two gates of random logic necessary to implement the daisy-chaining of DMA grant signals were placed above the relevant pads near the upper-left-hand corner of the chip. As mentioned, these were placed and routed in separate DPL code. The logo for the design, which represents the Scheme definition of the assq function, runs vertically between the right-hand pads and the metal wires that connect them to the data path bus lines. It was laid out using a program written by Batali that converts AST format font files into DPL's internal format.

Discussion -- The project as a high-level design experiment

Half an environment. Although I am a fan of silicon compilers, more traditional design methods will probably never be completely replaced, and will certainly be around for some time. If some of the chips we are to design are to be very large, it would seem that we are stuck with Mead and Conway abstractions, machine generated structures, hierarchical representations, design rule checkers, and so on. In designing the Assq chip, I dealt with a particular set of these: DPL, Daedalus, the Shrobe data path generator, the Batali

PLA generator, Baker's node extractor, design rule checker, and static analyzer, and Chris Terman's simulator. Since one of the main reasons for designing the Assq Chip was to test out the available design environment, it will be worthwhile to consider briefly how that environment held up. The programs I mentioned all have their strengths and weaknesses, but those aside, everything in my experience with the design of the Assq chip has convinced me of the need for completely integrated design environments in which all the various representations of a chip -- geometric, electrical, even raster-scan -- are hung from the same hierarchical representation of cell types and cell tokens.

I feel like an ingrate analyzing DPL, but... DPL is a fantastically expressive language for describing hierarchical layouts, but I encountered three difficulties with it that require attention. First, it takes a large amount of memory to run. This, it seems, can probably be fixed. Second, designing with DPL needs to be somewhat more incremental. Presently, small changes frequently require large amounts of recomputation. This is partially because the full potential of the already heavily exploited idea of caching of partial results was not realized, but more because the great expressive power of DPL allows one to express such intricate interrelationships between objects that such caching won't avoid the need for much recomputation. The worst case of this phenomenon, it would seem, is the sort of involved routing done in the Assq chip. It is an interesting question how much of a tradeoff there is between expressive power and incrementality. The third problem is a conceptual one. Because of the necessity of augmenting DPL structures when instances are aligned and realigned, DPL sets up a system of variable evaluation separate from that of Lisp. This leads to confusion when Lisp and DPL are mixed. For example, there is no DPL analog of **mapc**, and there is no clean way to augment arbitrarily complicated data structures made up of DPL objects, such as those created by the careful router mentioned above. Either some design language should be built on top of DPL, or DPL should be changed to be more friendly in this respect; I would favor the former.

Circuit-generation tools. The PLA and data path generators worked pretty much the way I expected they would. The major disappointment was that it was not feasible for me to redesign the cells used by the data path generator so as to eliminate functions I didn't use. The library of cells available to the data path generator will need to be extended beyond the set used in the Scheme-81 and Assq chips if it is to find wider application.

Design environments must be integrated. (Surprise!) The most important flaw in the design environment I used was the low bandwidth of communication between those parts of the world we might divide into the design tools and the analysis tools. The output of the node extractor, design-rule checker, static analyzer, and simulator was nearly meaningless in many cases because the considerable structural information available in DPL's internal structures was not used by the node extractor, which took a rectangle file generated from CIF as its input. Sufficient information is available in the system for, say, the static analysis program to say that "the output of the right-inverter of the 17th register-cell of the table-register of the main-register-array cannot be set to zero" instead of "the electrical node at coordinates (1324,2048) cannot be set to zero". This information would allow iterated errors to be recognized and summarized instead of being listed separately. Just about every piece of information that DPL has about a design could used to advantage by these other programs. (Consider a simulator that labels the color display with the values on each electrical node after each cycle.) In the limit, this argues for every part of the design process to live on the same machine and use the same data structures.

Replicated bogus design rule violations are no fun. Finally, until silicon compilers take over most integrated circuit design tasks, large portions of most designs are likely to be machine-generated, as were the PLA and data path for the Assq chip, or taken directly from cell libraries, as were the Assq chip's pads. Creators of design environments must come up with ways to make apparent errors and the like inside such structures meaningful to the designer without recourse the expedient of asking advice of the author of the cell layout or its generating program. (I got this observation from Clark Baker.) Perhaps a structured on-line documentation system is called for. High-level design needs to proceed at the high level; otherwise the complexity of the process is guaranteed t · be prohibitive.

A Modest Proposal

Why AI Research Needs Silicon Compilers

The debate over whether the real world needs custom LSI is not yet done, but it seems clear to me that it will quite soon become a necessity in artificial intelligence research. Computers of traditional design would have to operate at or beyond theoretical limitations in order to support some of the programs we want to write right now, so we're going to have to build our own. Building a machine with, say, 10^{10} transistors[1] in it is going to be impractical without custom LSI, at the very least because of the physical size of such a machine built out of off-the-self TTL. Microprocessor networks will be a workable stopgap, but a network of 1024 μP's is at best 1024 times faster than one μP (which, by the way, is many times slower than a KL-10), and current off-the-shelf μP's have not proven themselves well-adapted to large-scale networking. AI should never count on the real world to provide its processing needs.

Economics of silicon compilation. Custom LSI design is an extraordinarily painful and time-consuming process, labor-intensive in a labor-bound field. As a solution to this problem, I propose to build a silicon compiler. This program will remove all human labor from the process of turning a description of chip behavior into a layout specification suitable for cooking. Given that we need to build custom LSI circuits, I believe that the economics of LSI design in AI research will favor silicon compilers over hand design in almost all cases. This is because of the trade-off between the effort expended in a design and the size and efficiency of the resulting circuit. An ordinary person, such as a novice designer or a compiler, can produce on short notice *some* design that will work, but with few promises as to size or efficiency; the Assq Chip is an example. A group of experienced designers can produce a design for an extremely small and fast microprocessor in two years. The real world puts so much effort into size and efficiency because (1) when you're selling millions of copies of a chip, processing rivals or overtakes design on the balance sheet, and a 10% smaller chip saves 10% of processing costs in these conditions; and (2) a customer will just go across the valley if someone else's product is faster. Neither consideration is as important for custom LSI design in AI research. The small numbers involved will make size-dependent costs a much smaller proportion of the total cost, and the experimental nature of the applications will allow some inefficiencies to be tolerated, especially when those who must tolerate the inefficiencies would have to redesign the chips themselves to eliminate them. Turn-around is another important consideration. Researchers want their custom chips right now and manual design takes time; the program I propose to write should take a few hours to a day. And so it would seem that silicon compilation is the way of the future.

What I propose to do

Big lambda to little lambda. The program I propose to write will take as input a function definition in the Scheme language and produce as output a representation suitable for cooking of a circuit that will perform that function when placed in an appropriately configured Scheme-81 computer (Scheme 1981). (The program will also provide all the information necessary to perform a simulation of the chip layout.) Although some modifications to the Scheme input format may be forced upon me by harsh reality, my hope is to write an editor command that will convert a function definition into a chip without requiring any special knowledge of its user. My reason for using the Scheme language as an input format is quite straightforward: programming languages are the only proven way to express ideas about how a computing machine should work, and Scheme is the best programming language going. The reason why the output chips will be

---

1. Human brains have 10^9 to 10^{11} neurons in them, depending on how you count, and most neurons are considerably more complex than transistors, so we're talking about artificial iguanas here.

designed to live in the Scheme-81 environment is also straightforward: the Scheme-81 computer will solve all problems of storage allocation, speed[1], interrupt handling, testing[2], and input/output for me, allowing me to concentrate on compiler issues.

Assq's progeny. This proposal took shape during the process of designing the Assq Chip. While working on the design I noticed that it had the interesting property that the particular set of functions being implemented was effectively a parameter of the design, having almost no impact on the physical layout of the chip. The only function-dependent details of the layout are the dimensions and contents of the microcode PLA (assuming non-recursive microcode) and the data path, and the numbers of flip-flops and register cells. These details were generated in an ad hoc manner, partly by hand and partly by a variety of somewhat general programs, but in at no point was any deep thought involved. This was encouraging even as it was boring.

The form of the compiled chips. My compiler will take advantage of these observations by producing chips which are physically similar in most respects and whose variable parts will be determined by analysis of the source program. The layout will be somewhat more general than that of the Assq Chip. The major change will be the elimination of the hardwired microcode stack in favor of a stack being kept in cons cells, as in the Scheme-81 chip. Other changes will involve modifications in routing to allow for potentially much larger PLA's and data paths and for the increased number of functions the chips might have to perform. (The Assq Chip performs 29 functions, each with a five-bit opcode. Most, of course, are for debugging. Chips produced by the compiler may require larger opcodes.)

Compilation with customizable registers. The technically interesting work to be performed by the compiler involves the translation of the source program into a set of microcode instructions to be fed to the PLA generator and a description of the data path to be fed to the data path generator. What makes this program more than a routine exercise in compiler writing is the great flexibility available in the design of the data path. Whereas the machine languages of most object computers provide a fixed number of standard, interchangeable general purpose registers which can only be used in a small number of ways and usually at most two at a time, a silicon compiler can design special-purpose register arrays which can perform large numbers of moderately complicated register operations in parallel. For example, a data path could be designed so as to allow one microcode instruction to add A to B, move C to D, decrement E, and exchange F and G in one cycle, producing condition bits for listp(A), B=0, C=D, and E<F for use in tests by the next instruction. Close analysis of the source program can thus result in a chip design made highly efficient by data path parallelism. Indeed, given the memory-intensive nature of most Lisp programs, it seems plausible that 100 percent bus utilization efficiency can be attained by designing chips that can perform all the operations that need to occur between any two successive memory references in a single cycle.

Three sets of issues arise. The first are matters of representation: how best to represent potential parallelism at intermediate levels of compilation? One possibility is to represent a program as a directed graph whose links represent ordering constraints. The second set of issues has to do with the process of reasoning with the various possible data path designs. These designs are constrained by planarity, by the set of available register operations and tests, and by other factors as well. The program must execute a mapping between the sets of operations which need to occur in one cycle and the possible data path designs, and not every such collection of sets of operations can be simultaneously accommodated in a single data path design. If a good theory of the possible data path designs can be constructed, it can be used to reason about what collections of sets of operations can and cannot be accommodated. The third set of issues has to do with the algorithmic efficiency of the resulting design. Indeed, the compilation process can be thought of as a very

---

1. The chips will be tied to the Scheme-81 chip's glacial clock, so electrical speed will not be an important consideration. Algorithmic speed, of course, is another thing entirely.

2. The chips will be tested by programs written for the Scheme-81 chip which will use functions of the chip designed specifically for the purpose.

fancy optimizer, taking a straightforward translation of the source program and attempting to squeeze out redundant register capabilities and unnecessary microinstructions. Of particular concern is the consed stack. With luck, the tail-recursion elimination techniques applied by Steele in his traditional (if it can be called that) Scheme compiler (Steele 1978) can be adapted to the silicon domain in order to reduce stack consing, but this will interact with the other considerations in as yet unknown ways.

Plan of action. The tack I will take in writing this compiler will be to begin by getting up a version which produces correct chips without worrying much about issues of optimization. From that point I will experiment with a variety of compilation techniques such as suggested by the speculations just mentioned and by experience. I plan to give the program a series of tests, beginning with the relatively simple equal function, followed by a simple pattern matching program, a number of modules to implement the AMORD language, and finally a number of functions for symbolic algebra, such as an algebraic simplifier and (should a miracle occur and I get this far) a polynomial GCD algorithm. Once the program is running it should not matter much to it how large are the functions it is being asked to compile. However, I do not yet know how quickly the size of the resulting chips will grow with the size of the source programs. In the Assq Chip, that portion of the microcode not devoted to fixed bus protocol and testing operations amounted to about 15 percent of the total, so it seems reasonable expect that the PLA will not double from its Assq Chip size until the functions involved become quite complex indeed.

Other work. Of the existing work on silicon compilation, the project most similar to mine is the quite exciting MacPitts project of Siskind *et al* (1982). The chips that will be produced by our two compilers will be similar in consisting of a control machine and a data path. The data path generators used are quite similar, the major difference being that that of Siskind *et al* allows more complicated internal bus structures. Their control machine uses Weinberger array logic instead of a PLA, thus allowing microcode multiprocessing. The MacPitts chips will differ from mine in electrical details (three-phase clocking and dynamic logic as opposed to the more conservative Mead-and-Conway scheme), but will be quite similar in spirit, with much concern for modularity. The most important difference between our two projects is that Siskind *et al* have in mind a general-purpose compiler applicable to a very large class of VLSI design problems, whereas my concern is more with building an integrated *system* of chips, based on the Scheme-81 environment. Thus their input language, although also a Lisp dialect, is stated in terms of numerical operations and synchronization logic, whereas mine will be in terms of Lisp data structures. It is fully conceivable that one might attempt to write a compiler from Scheme to the MacPitts input language that would provide the necessary bus-protocol logic and so forth; I hope someone attempts this. Depending on the application, my compiler will either eliminate the bother of specifying routine details of memory operations or provide inconvenient restrictions to be circumvented. The former view is probably justified because the applications I have in mind are AI-oriented operations such as pattern matching, and these are usually programmed in symbolic languages such as Scheme.

A Concluding Note

Compiler cultures. My silicon compiler will not be the first and it will not be the last; its chips could be better in many and varied ways; why write it? Because there are so many source and object languages, the development of compilers has always been very much a social phenomenon. Tricks are discovered which work in specific situations but don't generalize interestingly, yet a compiler written using only generalized theory is no good. The writer of a new compiler must size up a list of the tricks that seem applicable and stir them up with what little formal theory is available. In such a world, the vagaries of experience achieve obvious importance. The old tricks generally will not translate well into the terms of silicon compilers. The list of obvious tricks for customizing data paths is endless, but which ones are really useful? What is needed is a silicon compiler culture, and within it a Scheme-to-silicon analog of the Lisp-to-PDP-10 compiler subculture of the last fifteen years.

The future. So let's put up a Scheme-to-silicon compiler and start using it for real applications in AI.

Analysis of the chips it produces will suggest better ways to write it. How useful is encoding of the input and output fields of microcode PLA's? How useful are generalized placement and interconnect programs? When are predicate selector boxes *a la* Scheme-81 useful? I could provide guesses as to the answers to these questions and many others now, but they would only be guesses and my answers would only bring on exponentially more questions of the same sort. As an informed user community develops, so will the social mechanism that can let experience answer these questions and translate the answers into new and better compilers. In order for there to be a user community, there must be something to use, and I claim little for my proposed compiler other than that it can be used.

Appendix One: Lisp-level semantics for the Assq chip

```
(defun assq (atom table)
 (let ((result (*assq atom table)))
  (cond ((null result) nil)
     (t (car result)))))
(defun *assq (atom table (ilreg table))    ; ilreg has default value table
 (cond ((nlistp table) nil)
    ((and (listp ilreg) (eq table ilreg)) nil)
    ((and (listp (cdr ilreg)) (eq table (cdr ilreg))) nil)
    ((nlistp (car table)) nil)
    ((eq atom (caar table)) table)
    (t (*assq atm (cdr table)
           (cond ((nlistp ilreg) ilreg)
              ((nlistp (cdr ilreg)) (cdr ilreg))
              (t (cddr ilreg)))))))
;;; analogously for rassq
(defun delassq (atom table)
 (let ((result (*delassq atom table)))
  (cond ((null result) nil)
     ((eq result t) (cdr table))
     (t (rplacd result (cddr result)) table))))
(defun *delassq (atom table (previous-table nil) (ilreg table))
 (cond ((nlistp table) nil)
    ((and (listp ilreg) (eq table ilreg)) nil)
    ((and (listp (cdr ilreg)) (eq table (cdr ilreg))) nil)
    ((nlistp (car table)) nil)
    ((eq atom (caar table)) (if previous-table previous-table t))
    (t (*delassq atm (cdr table) table
             (cond ((nlistp ilreg) ilreg)
                ((nlistp (cdr ilreg)) (cdr ilreg))
                (t (cddr ilreg)))))))
;;; analogously for delrassq
(defun memq (atom list)
 (*memq atom list))
(defun *memq (atom table (ilreg list))    ; ilreg has default value table
 (cond ((nlistp list) nil)
    ((and (listp ilreg) (eq list ilreg)) nil)
    ((and (listp (cdr ilreg)) (eq list (cdr ilreg))) nil)
```

```
    ((eq atom (car list)) list)
    (t (*assq atm (cdr list)
           (cond ((nlistp ilreg) ilreg)
              ((nlistp (cdr ilreg)) (cdr ilreg))
              (t (cddr ilreg)))))))
(defun delq (atom list)
 (let ((result (*delq atom list)))
  (cond ((null result) nil)
     ((eq result t) (cdr list))
     (t (rplacd result (cddr result)) list))))
(defun *delq (atom list (previous-list nil) (ilreg table))
 (cond ((nlistp list) nil)
    ((and (listp ilreg) (eq list ilreg)) nil)
    ((and (listp (cdr ilreg)) (eq list (cdr ilreg))) nil)
    ((eq atom (car list)) (if previous-list previous-list t))
    (t (*delq atm (cdr list) list
           (cond ((nlistp ilreg) ilreg)
              ((nlistp (cdr ilreg)) (cdr ilreg))
              (t (cddr ilreg)))))))
(defun circularp (list)
 (*circularp list))
(defun *circularp (list (ilreg list))
 (cond ((nlistp ilreg) nil)
    ((eq ilreg list) t)
    ((nlistp (cdr ilreg)) nil)
    ((eq (cdr ilreg) list) t)
    (t (*circularp (cdr list) (cddr ilreg)))))
```

Appendix Two: Lisp-level interface to the chip

```
(defun assq (atom table)
 (rplaca **atom-loc atom)
 (rplaca **assq-table-loc table)
 (do ((fetch (car **result-loc) (car **result-loc)))
   ((boundp 'fetch) (car fetch))))
(defun rassq (atom table)
 (rplaca **atom-loc atom)
 (rplaca **rassq-table-loc table)
 (do ((fetch (car **result-loc) (car **result-loc)))
   ((boundp 'fetch) (car fetch))))
(defun delassq (atom table)
 (rplaca **atom-loc atom)
 (rplaca **delassq-table-loc table)
 (do ((fetch (car **result-loc) (car **result-loc)))
   ((boundp 'fetch)
    (cond ((null fetch) nil)
       ((eq fetch t) (cdr table))
       (t (rplacd fetch (cddr fetch)) table)))))
```

```
(defun delrassq (atom table)
 (rplaca **atom-loc atom)
 (rplaca **delrassq-table-loc table)
 (do ((fetch (car **result-loc) (car **result-loc)))
   ((boundp 'fetch)
    (cond ((null fetch) nil)
       ((eq fetch t) (cdr table))
       (t (rplacd fetch (cddr fetch)) table)))))
(defun memq (atom list)
 (rplaca **atom-loc atom)
 (rplaca **memq-list-loc list)
 (do ((fetch (car **result-loc) (car **result-loc)))
   ((boundp 'fetch) fetch)))
(defun delq (atom list)
 (rplaca **atom-loc atom)
 (rplaca **delq-list-loc list)
 (do ((fetch (car **result-loc) (car **result-loc)))
   ((boundp 'fetch) fetch)))
(defun circularp (list)
 (rplaca **circularp-list-loc list)
 (do ((fetch (car **result-loc) (car **result-loc)))
   ((boundp 'fetch) fetch)))
```

Appendix Three: The input to the data path generator

```
;;; What each register looks like:
;;; Physical top of the word is bit 0
;;; Bits 0 - 6  type field
;;;      7 - 35 data field
;;; The data field for a request by Scheme to the assq machine looks like:
;;;      7 - 11   5 bits, "-1 region" machine number
;;;      12 - 25  14 bits, its-assq code
;;;      26 - 30  5 bits, its-me code, names this particular assq chip
;;;      31 - 35  5 bits, opcode

(defconst type-size 7.)
(defconst type-start 0.)
(defconst type-end type-size)

(defconst datum-start type-end)
(defconst datum-size 29.)
(defconst datum-end (+ datum-size datum-start))

(defconst machine-number-size 5.)
(defconst its-me-size 5.)
(defconst opcode-size 5.)
(defconst its-assq-size (- datum-size (+ machine-number-size its-me-size opcode-size)))
```

```
(defconst machine-number-start datum-start)
(defconst machine-number-end (+ machine-number-start machine-number-size))

(defconst its-assq-start machine-number-end)
(defconst its-assq-end (+ its-assq-start its-assq-size))

(defconst constant-part-start machine-number-start)
(defconst constant-part-size (+ machine-number-size its-assq-size))
(defconst constant-part-end (+ constant-part-start constant-part-size))

(defconst its-me-start its-assq-end)
(defconst its-me-end (+ its-me-start its-me-size))

(defconst opcode-start its-me-end)
(defconst opcode-end (+ opcode-start opcode-size))

;;; Remember that the bits are in the opposite order from the Scheme chip's data path.
(defconst word-start 0.)
(defconst word-size (+ type-size datum-size))
(defconst word-end (+ word-start word-size))

;;; Bit pattern for machine number plus its-assq code.
(defconst constant-part-bit-pattern (+ #o76 (ash #o37 its-assq-size)))

;;; The type number for integers is 2 and the type number for unbound is 4.  Send back
;;; unbound when you're not done yet, and an integer as the non-nil value.  Make sure that
;;; these type constants get put in the right places in the final layout.  Note that
;;; all of these constants must be "backwards" in their field because the bits are
;;; numbered in reverse.  For example, type 2 is fixnum, but the actual constant is
;;; #2r0100000, i.e., seven bits which when reversed in order make 2.
(defconst list-type-constant #2r1000000) ;type LIST
(defconst datum-constant-1 #o-1) ;datum -1
(defconst unbound-type-constant #2r0010000) ;type UNBOUND
(defconst datum-constant-2 #o0)
(defconst fixnum-type-constant #2r0100000)
(defconst nil-pattern (+ (ash #o0 (1- type-start)) (ash #o0 (1- datum-start))))
(defconst non-nil-pattern (+ (ash fixnum-type-constant (1- type-start))
                             (ash #o0 (1- datum-start))))
(defconst not-yet-pattern (+ (ash unbound-type-constant (1- type-start))
                             (ash #o0 (1- datum-start))))

;;; Note that this register array has no cdr bit because the assq chip doesn't need one.
(defconst assq-regs
 '((table normal-register (size ,word-size) (starting-position ,word-start)
          (associates
           (listp-table bit-tester (size ,type-size) (starting-position ,type-start)
                        (bit-pattern ,list-type-constant))))
   (ilreg=table eq-test (size ,word-size) (starting-position ,word-start))
   (ilreg normal-register (size ,word-size) (starting-position ,word-start)
```

```
        (associates
         (listp-ilreg bit-tester (size ,type-size) (starting-position ,type-start)
                      (bit-pattern ,list-type-constant))))
(nil-source constant-source
            (size ,word-size) (starting-position ,word-start)
            (bit-pattern ,nil-pattern))
(non-nil-source constant-source
                (size ,word-size) (starting-position ,word-start)
                (bit-pattern ,non-nil-pattern))
(not-yet-source constant-source
                (size ,word-size) (starting-position ,word-start)
                (bit-pattern ,not-yet-pattern))
(table-element normal-register (size ,word-size) (starting-position ,word-start)
               (associates
                (listp-table-element bit-tester (size ,type-size)
                                     (starting-position ,type-start)
                                     (bit-pattern ,list-type-constant))))
(table-element=atom eq-test (size ,word-size) (starting-position ,word-start))
(atom normal-register (size ,word-size) (starting-position ,word-start))
(previous-table normal-register (size ,word-size) (starting-position ,word-start))
(its-assq bit-tester (size ,constant-part-size) (starting-position ,constant-part-start)
          (bit-pattern ,constant-part-bit-pattern))
(its-me top-level-literal (starting-position ,its-me-start)
        (size ,its-me-size) (bus-size ,its-me-size) (field-start ,its-me-start)
        (field-size ,its-me-size)
        (associates
         (test eq-test (starting-position ,its-me-start) (size ,its-me-size))))))
```

References

(Batali 1982) John Batali, Forthcoming documentation of the PLA generator, 1982.

(DPL 1980) John Batali and Anne Hartheimer, The Design Procedure Language Manual, MIT AI Memo 598, VLSI Memo 80-31, September 1980.

(DPL/Daedalus 1980) John Batali, Neil Mayle, Howie Shrobe, Gerald Jay Sussman, and Daniel Weise, The DPL/Daedalus Design Environment, Proc. VLSI-81, Edinburgh, August 1981.

(Scheme 1981) John Batali, Edmund Goodhue, Chris Hanson, Howie Shrobe, Richard Stallman, and Gerald Jay Sussman, The Scheme-81 Architecture -- System and Chip, Proc. MIT Conference on Advanced Research in VLSI, January 1982.

(Shrobe 1982) Howard E. Shrobe, The Datapath Generator, Proc. MIT Conference on Advanced Research in VLSI, January 1982.

(Siskind 1982) Jeffrey Mark Siskind, Jay Roger Southard, and Kenneth Walter Crouch, Generating Custom High Performance VLSI Designs From Succinct Algorithmic Descriptions, Proc. MIT Conference of

Advanced Research in VLSI, January 1982.

(Steele 1978) Guy Steele, Rabbit: A Compiler for Scheme, MIT AI Lab TR-474, May 1978.