CSMC 305 Final Project: Emulating the SCHEME-79 Chip

Sammy Furr

December 20, 2019

1 Introduction

I first got interested in LISP when I started to use Emacs as my editor at the beginning of college. I was shocked when I learned how old the language is—it predates almost all languages still used today, and was certainly a high-level language decades before C[4]. Dialects of LISP were being developed while we were still figuring out a lot about computing that we take for granted today. In the late 70's, as Guy L. Steele Jr. and Gerald Jay Sussman were creating SCHEME at the MIT Artificial Intelligence Laboratory, swaths of researchers were hard at work trying to fit millions of transistors into chips the size of a fingernail, using new Very Large Scale Integration (VLSI) design techniques. In 1979, these two worlds would join as part of the MPC79 project[3]. Sussman and Steele, along with Jack Holloway and Alan Bell, would create a chip that directly interpreted SCHEME.[2]

I was struggling to find a LISP machine to emulate for my final project. The papers I found described stack machines or PDP clones with some hardware instructions to do the LISP-y functions (cons, car, etc.) thrown in. I was beginning to sour on my idea when I finally stumbled across the SCHEME-79 paper. A chip that directly interpreted s-expressions, the basis of one of the least imperative languages of all time? A processor that was entirely specified in LISP and then "compiled into artwork"[2, 12]? Uniting software and hardware abstraction techniques, the functional and the imperative, s-expressions and silicon, the SCHEME-79 chip is a remarkable piece of computing history. I hope that when you run the emulator and see s-expressions moving through registers you will be as amazed as me!

2 How the Chip Works

The SCHEME-79 chip is remarkably simple. It contains

1. 10 special-purpose registers.
2. 3 Programmable Logic Arrays (PLAs), which combine to form a single Finite State Machine (FSM).
3. 44 pads for connecting to the outside world.

This is all connected by a 32 bit bus. That's really it—s-expressions are directly stored in the registers, each of which contains a type field. Each

7-bit type is the equivalent of an opcode on a traditional processor. On each instruction cycle the FSM sees a type, such as type 005_8-procedure, transitions it's state accordingly, and executes the micro-word that goes with the type.

```
(deftype procedure
  (assign *val* (&cons (fetch *exp*) (fetch *display*)))
  (&set-type *val* closure)
  (dispatch-on-stack))
```

Procedure takes the memory address of the procedure definition in the expression register, conses it to the environment stored in the display register, and sticks all this in the value register to make a closure. This closure will be applied and evaluated by previous types, which have been stored on the stack. To transition to the next state, the FSM dispatches the type in the stack register. Computation proceeds like this, recursively evaluating s-expressions until the program is over, at which point the processor halts.

3 Emulating SCHEME-79

The SCHEME-79 paper comes with a full specification of the microcode on the chip. Emulating the chip is not as simple, of course, as copying the microcode verbatim into a file and interpreting. The real challenge in emulating the chip is writing functions to emulate the functionality of the chip that is implemented directly in hardware, not programmed onto the PLA. These functions are not described in the microcode specification—only named. Doing this involved a fair amount of trial-and-error, and a lot of re-reading the paper for hints. Luckily, some of the microcode is fairly self-documenting. Some of it, however, is decidedly not:

```
...(eval-exp-popj-to internal-apply)) ;Amazing! Where did retpc
   go?
```

"Amazing! Where did retpc go?" is the original authors' comment, not mine.[2, 39]

As it turns out eval-exp-popj-to is a function in hardware. It stores the microcode return-address (type), internal-apply in this example, in the type field of the stack register, saves the stack register onto the stack, and then

evaluates the expression in the exp register. This is where retpc, our return program counter, went. Here's the code to emulate this behavior:

```
(define (eval-exp-popj-to type)
  (&set-type *stack* type)
  (save *stack*)
  (dispatch-on-exp-allowing-interrupts))
```

In the end, the emulator includes a few basic components, just like the chip it emulates:

1. Vectors to hold the types, which are used like addresses of micro-words.
2. Functions and macros to bind micro-code addresses to the microcode given in the paper.
3. A variety of direct hardware functions and macros not given in the paper.
4. A vector and some functions to emulate memory.
5. A function to step through individual instruction cycles, running a program.

4 TODO

4.1 Garbage Collection

The emulator is still lacking one large piece of functionality from the original chip—garbage collection. A large portion of the paper is dedicated to garbage collection, and you can see why when running the emulator. The recursive nature of SCHEME means that values are constantly being thrown on the stack. The stack is implemented on the heap as a LISP list, which is beautifully simple and also incredibly quick to fill memory. The garbage collector is fully specified, but I didn't have time to completely understand the algorithm for it, and decided to forgo implementing it to focus on simply getting the interpreter running.

4.2 Compilation

Programs for the original chip were written in LISP, and then compiled to the S-code that SCHEME-79 actually runs. Though S-code is still recognizable as LISP, it differs in how some operations are connected together, and is more explicit. The paper doesn't come with a compiler, so for now S-code has to be hand-written. For example, the ultra-simple function application:

```
((lambda () 3))
```

Translates into the following S-code:

```
ADDR CAR          . CDR
1001 01100001002 . 00000000000 ;; apply-no-args to addr 1002
1002 00500002000 . 00000000000 ;; procedure at addr 2000
2000 10000000003 . 00000000000 ;; self-evaluating-immediate 3
```

All numbers are specified in octal.

This is both tedious and difficult to write by hand, and a compiler is definitely needed! I still need a fuller understanding of every aspect of the chip in order to write a compiler, however. Barring writing a compiler, the functionality to read S-code from a file is needed, as you must currently type S-code into the file that houses code for the emulator—sorry!

5 Conclusion

Emulating SCHEME-79 was a challenge, but a completely worthwhile one for such an awesome piece of engineering. The chip is truly extraordinary, and the way it was implemented was revolutionary for the time. In the era of the all powerful general-purpose processor a special-purpose chip so influenced by one type of programming language may seem like an antiquity. Looking closely at the features of a modern x86 or ARM processor reveals that this couldn't be further from the truth. When you load a webpage using AES, your connection is encrypted using hardware directly on your chip. The need for fast encryption and decryption has become so ubiquitous that every modern consumer processor includes instructions to perform *a specific encryption algorithm*. Just like the need to quickly interpret SCHEME drove the SCHEME-79's hardware development, the need to quickly decrypt and encrypt webpages led chip-makers to develop direct hardware for the task.

We don't make so many standalone single-purpose processors now. Instead, we put them in every general-purpose processor we make.

The techniques of abstraction used to design the SCHEME-79 chip enable the vast chips we have today. With a minimum feature width of 5 microns[2, 21], SCHEME-79 was on the bleeding edge. Standing in a classroom you can see how far we've come. The classroom is the gate width on SCHEME-79. Hold up a pinky. That's the width of a gate doing the AES cryptography on Intel's latest 10nm node[1]. Just incredible.

References

[1] https://en.wikichip.org/wiki/10_nm_lithography_process.

[2] Bell, J. H. G. L. S. J. G. J. S. A. The scheme-79 chip.

[3] Conway, A. B. M. N. R. L. R. P. L. Implementation documentation for the mpc79 multi-university multiproject chip-set.

[4] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, part i.