FUNCTIONALLY OLD
TOWARD AN ATARI 2600 EMULATOR IN CHEZ SCHEME

Chris Frisz
TriClojure Meetup
23 Aug 2018

WHO AM I?

- Software engineer at Cisco since 2012
- Long-time Scheme programmer
- Worked on Chez Scheme
- Lapsed Clojure hacker

WHY AN EMULATOR IN SCHEME?

- Document the experience of starting a Scheme project
- Figure out if Chez Scheme is fast enough
- Do some systems programming
- Make stuff for video games

WHAT THIS TALK *IS*

- An experience report and how Scheme helped (and hurt)
- A slice of Scheme programming life
- A dip into systems programming
- A demonstration of cool stuff in Chez Scheme

WHAT IT *IS NOT*

- A full guide to writing an Atari emulator (or a 6502 emulator)
- The definitive guide to programming with Chez Scheme (or any other Scheme)
- Tips and tricks for getting good at video games (or anything about video games…yet)

INTRO TO THE 6502

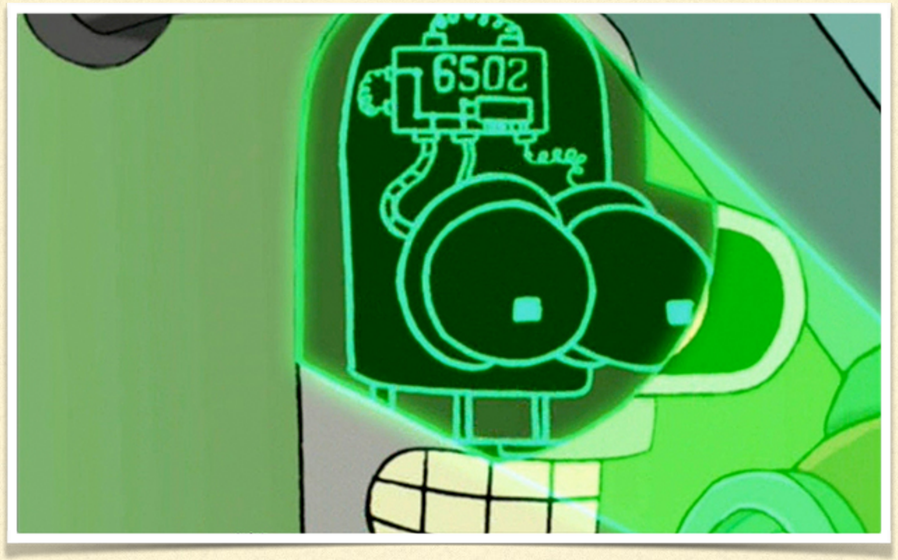
- An inexpensive 8-bit processor
- Originally produced in 1975
- Popular for giving a lot of processing bang for the buck
- Readily available to this very day


apple II
plus


POWER
ON
OFF
TV TYPE
COLOR
B-W
VIDEO COMPUTER SYSTEM™
GAME SELECT
GAME RESET
ATARI®


Nintendo
ENTERTAINMENT SYSTEM™
POWER
RESET

```
 8 ******************
 9
10           ORG   $4000
11 A1        =     $3C
12 A2        =     $3E
13 A4        =     $42
14 AUXMOVE   =     $C311
15
16 ****************************
17 * SETUP - move data for VTOC
18 * and catalog to auxmem at
19 * B000-B3FF (pseudo trk 11
20 * 0-3)
21 ************************
22 SETUP     LDA   #<VTOC
23           STA   A1
24           LDA   #>VTOC
25           STA   A1+1
26           LDA   #<END
27           STA   A2
28           LDA   #>END
29           STA   A2+1
30           LDA   #$00
31           STA   A4
32           LDA   #$B0
33           STA   A4+1
34           SEC
35           JMP   AUXMOVE
36
```



REGISTERS

- 16-bit Program Counter (PC)
- 8-bit stack pointer (S)
- 8-bit Processor Status (P)
- One 8-bit accumulator register (A)
- Two 8-bit index registers (X and Y)

THE CPU IN SCHEME

```
(define-record-type cpu
  (nongenerative)
  (sealed #t)
  (fields
    (mutable pc)
    (mutable s)
    (mutable p)
    (mutable a)
    (mutable x)
    (mutable y))
  (protocol
    (lambda (new)
      (case-lambda
        [() (new *pc-init* *s-init* *p-init* *a-init* *x-init* *y-init*)]
        [(pc s p a x y) (new pc s p a x y)]))))
```

- Enter `define-record-type`
- Specify mutable fields (but they could be immutable)
- Make records using zero or six arguments

MEMORY

- 16-bit address space: 64k addresses
- Memory access is *little endian*

AN ASIDE ON ENDIANNESS

- “Endianness” refers to the order in which bytes in a multi-byte value are stored in memory
- Little-endian means the least significant byte is stored first:

```
Storing value 0xABCD:

  Little endian:                Big endian:

      Address | Value               Address | Value
      ----------------            ----------------
       0x0000 |  0xCD               0x0000 |  0xAB
       0x0001 |  0xAB               0x0001 |  0xCD
```

MEMORY IN SCHEME

- Enter bytevectors
- Provide an interface for interacting with sequences of raw bytes
- Endian access built into the interface

```
(define make-memory
  (lambda ()
    (make-bytevector (*memory-size*) 0)))

(define memory-get-byte bytevector-u8-ref)

(define-syntax memory-get-word
  (syntax-rules ()
    [(_ memory address) (bytevector-u16-ref memory address 'little)]))

(define memory-set-byte! bytevector-u8-set!)

(define-syntax memory-set-word!
  (syntax-rules ()
    [(_ memory address value) (bytevector-u16-set! memory address value 'little)]))
```

BYTEVECTORS IN ACTION

```
Storing value 0xABCD:

  Little endian:

      Address | Value
      ---------------
       0x0000 |  0xCD
       0x0001 |  0xAB
```

```
> (define bv (make-bytevector 2))
> (bytevector-u8-set! bv 0 #xCD)
> (bytevector-u8-set! bv 1 #xAB)
> (bytevector-u16-ref bv 0 'little)
#xABCD
>
```

INSTRUCTIONS

- ~50 instructions, e.g., `adc` and `sta`
- 11 address modes for retrieving operands, e.g., `immediate` and `absolute`
- 152 total instruction variants

INSTRUCTIONS IN SCHEME

```
(define-instruction and
  (advance-pc: #t)
  (raw-operand: #f)
  ([#x29 2 immediate]
   [#x25 2 zero-page]
   [#x2D 3 absolute])
  (lambda (cpu memory)
    (lambda (opnd)
      (let ([result (fxlogand (cpu-a cpu) opnd)])
        (update-flags! cpu
          ([zero (fxzero? result)]
           [sign (fxlogbit? 7 result)]))
        (cpu-a-set! cpu result)))))
```

- Create a `define-instruction` syntax
- Specify instruction declaratively:
  - Attributes
  - Opcodes
  - Operand address modes
  - Instruction semantics

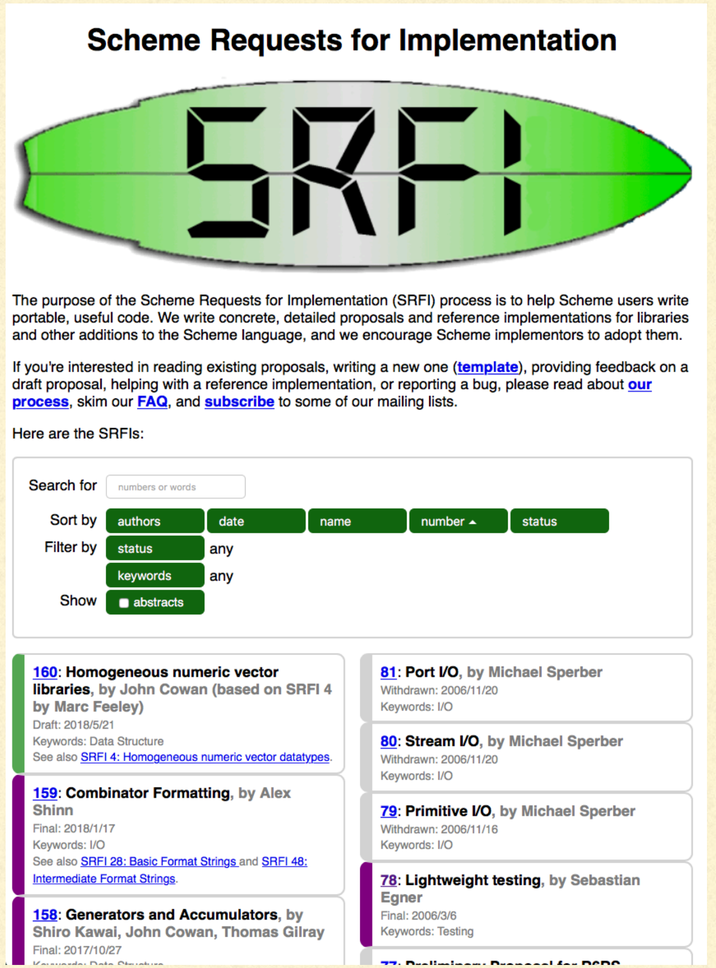
INSTRUCTIONS IN SCHEME (CONT.)

```
(define-syntax define-instruction
  (syntax-rules (advance-pc: raw-operand: lambda)
    ;; Pattern
    [(_ mnemonic
       (advance-pc: advance-pc?)
       (raw-operand: raw?)
       ([opcode* bytes* address-mode*] ...)
       (lambda (cpu memory)
         (lambda (operand)
           ?body ?body* ...)))
     ;; Expansion
     (begin
       (vector-set! opcode-function-vector opcode*
         (lambda (cpu memory)
           (cpu-pc-set! cpu (fx1+ (cpu-pc cpu)))
           ((lambda (operand)
              ?body ?body* ...)
            (get-operand cpu memory address-mode* raw?))
           (when advance-pc?
             (cpu-pc-set! cpu (fx+ (cpu-pc cpu) (fx1- bytes*))))))
       ...)]))
```

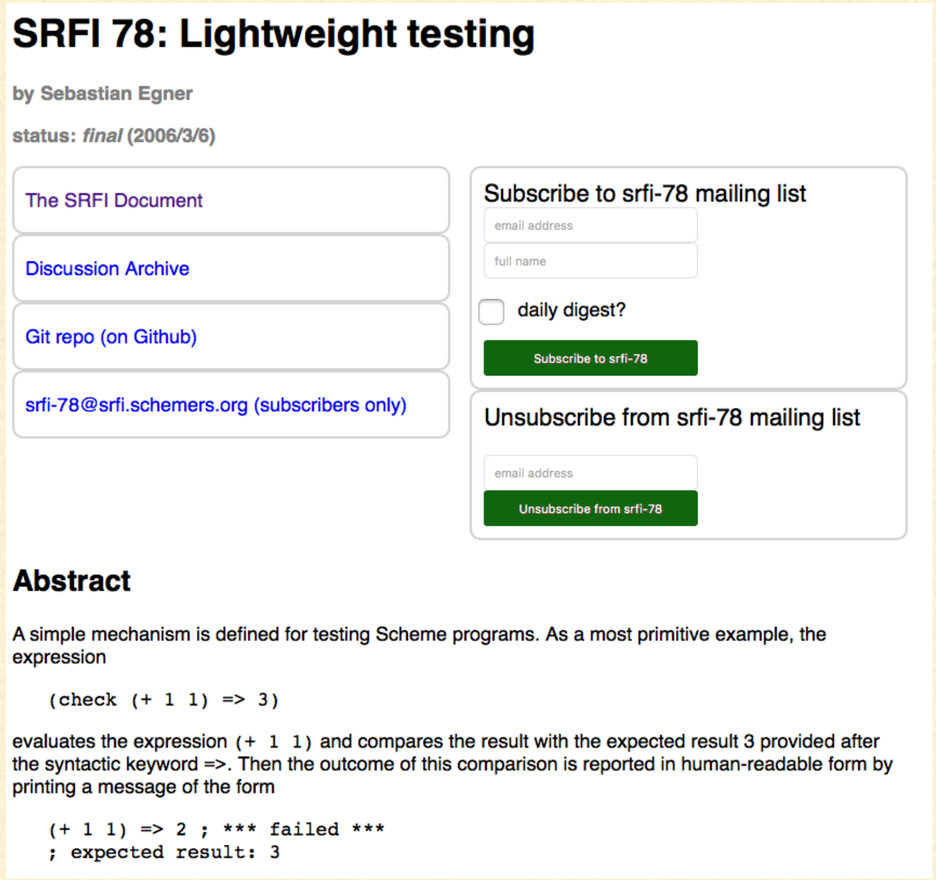
- Use a `syntax-rules` macro
- Expand into a function per opcode
- Take care of instruction “plumbing”
- Set up the runtime opcode lookup data structure

TESTING

- No built-in or standard testing library
- Turn to SRFIs—Scheme Request For Implementation
- Provide a description of an interface—not an actual interface



TESTING (CONT.)



SRFI 78: Lightweight testing

by Sebastian Egner

status: *final* (2006/3/6)

The SRFI Document

Discussion Archive

Git repo (on Github)

srfi-78@srfi.schemers.org (subscribers only)

Subscribe to srfi-78 mailing list

email address

full name

daily digest?

Subscribe to srfi-78

Unsubscribe from srfi-78 mailing list

email address

Unsubscribe from srfi-78

Abstract

A simple mechanism is defined for testing Scheme programs. As a most primitive example, the expression

```
(check (+ 1 1) => 3)
```

evaluates the expression `(+ 1 1)` and compares the result with the expected result 3 provided after the syntactic keyword =>. Then the outcome of this comparison is reported in human-readable form by printing a message of the form

```
(+ 1 1) => 2 ; *** failed ***
; expected result: 3
```

- Two testing SRFIs: SRFI-64 and SRFI-78
- Used Andy Keep's chez-srfi repository

BUILDING

- No Leiningen or Boot equivalent
- Have fun with Make
- (I forgot how to write Makefiles)

```
all : run-tests

target-dirs :
	mkdir -p target/{main,test}/scheme target/dependency

submodules/chez-srfi : target-dirs
	git submodule init
	git submodule update

link-srfi-dirs : submodules/chez-srfi
	( cd submodules/chez-srfi && scheme --script link-dirs.chezscheme.sps )

target/test/scheme/srfi : link-srfi-dirs submodules/chez-srfi target-dirs
	ln -fs ../../../submodules/chez-srfi target/test/scheme/srfi

target/main/scheme/maf6502.so : src/main/scheme/maf6502.ss target-dirs
	echo '(begin (compile-profile #t) (compile-file "src/main/scheme/maf6502.ss" "target/main/scheme/maf6502.so"))' | scheme -q --compile-imported>

target/test/scheme/maf6502-test.so : src/test/scheme/maf6502-test.ss target/main/scheme/maf6502.so target/test/scheme/srfi target-dirs
	echo '(compile-file "src/test/scheme/maf6502-test.ss" "target/test/scheme/maf6502-test.so")'\
	  | scheme -q --compile-imported-libraries --libdirs target/main/scheme:target/test/scheme

repl : src/test/scheme/maf6502-repl.ss target/main/scheme/maf6502.so
	scheme --libdirs target/main/scheme src/test/scheme/maf6502-repl.ss

run-tests : target/test/scheme/maf6502-test.so
	scheme --debug-on-exception --libdirs target/main/scheme:target/test/scheme --program target/test/scheme/maf6502-test.so

clean :
	rm -rf target/*
```

DEBUGGING

- Built-in Chez Scheme debugger—even if it's a little intimidating at first
- `trace-define` **and** `trace-lambda`

DEBUGGING: THE INSPECTOR

- Interactive debugger
- Trigger on exception or breakpoints
- Interrogate the system:
  - Values in scope
  - Procedure and call code
  - Source file locations

```
(define-instruction adc
  (advance-pc: #t)
  (raw-operand: #f)
  ([#x6D 3 absolute])
  (lambda (cpu memory)
    (lambda (opnd)
      (let ([result (+ (cpu-a cpu) opnd)])
        (break)
        (update-flags! cpu
          ([carry (fx> result #xFF)]
           [sign (fxlogbit? 7 result)]
           [zero (fxzero? (wrap-byte result))]))
        (cpu-a-set! cpu (wrap-byte result))))))
```

```
break> i
#<continuation>                                                    : s
  continuation:          #<continuation in execute>
  procedure code:        (lambda (cpu memory) ...)
  call code:             (break)
  frame and free variables:
  0. cpu:                #[...]
  1. result:             #x5
#<continuation>                                                    : r 0
#[...]                                                             : p

#[#{cpu pfwdf1gupe5nwxwvty7bzewir-12} #xC01D #xFC #x24 #x0 #x4 #x5]

#[...]                                                             :
```

DEBUGGING: TRACING

```
(define-syntax define-instruction
  (syntax-rules (advance-pc: raw-operand: lambda)
    [(_ mnemonic
       (advance-pc: advance-pc?)
       (raw-operand: raw?)
       ([opcode* bytes* address-mode*] ...)
       (lambda (cpu memory)
         (lambda (operand)
           ?body ?body* ...)))
     (begin
       (vector-set! opcode-function-vector opcode*
         (lambda (cpu memory)
           (cpu-pc-set! cpu (fx1+ (cpu-pc cpu)))
           ((trace-lambda mnemonic (operand)
              ?body ?body* ...)
            (get-operand cpu memory address-mode* raw?))
           (when advance-pc?
             (cpu-pc-set! cpu (fx+ (cpu-pc cpu) (fx1- bytes*))))))
       ...)]))
```

```
> (let ([*cpu* (make-cpu)]
        [*memory* (make-memory)]
        [fact-bytes '#vu8( #xA9 #x05 #xC9 #x00 #xD0 #x03 #xA9 #x01 #x00 #xAA #xCA #xE0
0 #x07 #x6D #x00 #x00 #xCA #x4C #x18 #xC0 #xA8 #x68 #xAA #x98 #x4C #x0A #xC0 #x00 )])
    (load-program! fact-bytes *memory* #xC000)
    (cpu-pc-set! *cpu* #xC000)
    (execute *cpu* *memory*)
    *cpu*)
|(lda #x5)
|#<void>
|(cmp #x0)
|#<void>
|(bne #xC009)
|#<void>
|(tax #<void>)
|#<void>
|(dex #<void>)
|#<void>
|(cpx #x1)
|#<void>
|(beq #xC02A)
|#<void>
|(tay #<void>)
|#<void>
|(txa #<void>)
|#<void>
|(pha #<void>)
|#<void>
|(tya #<void>)
|#<void>
```

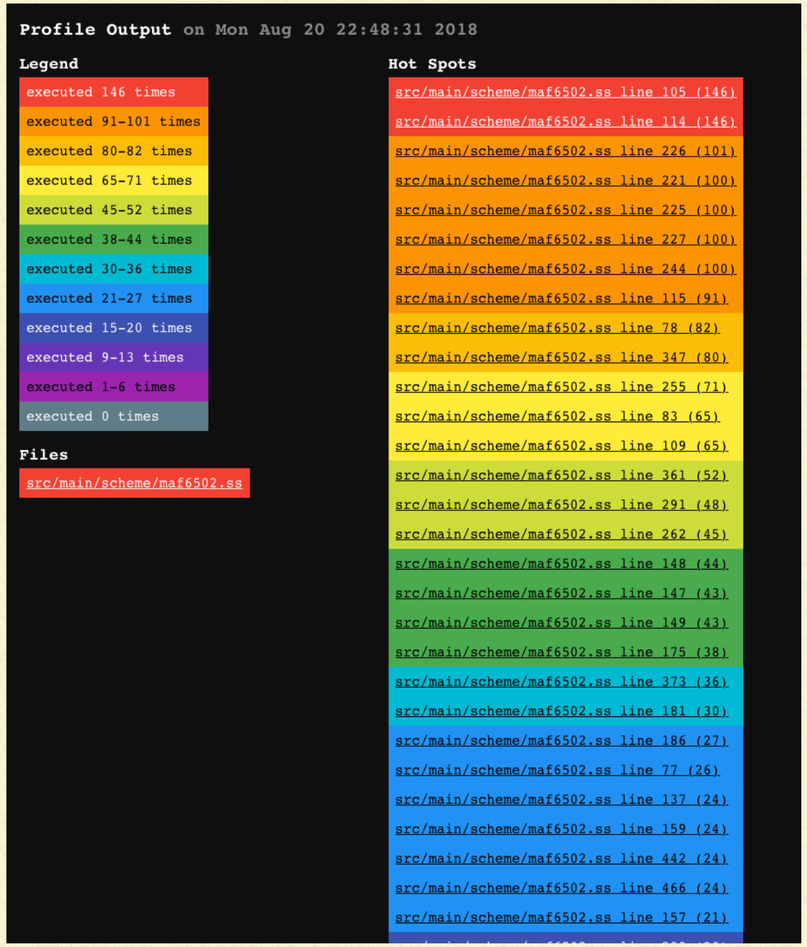
- Enter `trace-define` and `trace-lambda`
- Show calls with argument values
- Print `lambda` return value

PERFORMANCE DEBUGGING

- Built-in profiler with HTML output
- Cost centers for metrics on specific sections of code

PERFORMANCE DEBUGGING: PROFILING

- Profiling with expression-based execution counts
- HTML output with color-coded hotspots



PERFORMANCE DEBUGGING: PROFILING (CONT.)

```
188 (define-syntax wrap-word
189   (syntax-rules ()
190     [(_ value) (fxlogand value #xFFFF)]))
191
192 (define-syntax stack-push-byte!
193   (syntax-rules ()
194     [(_ cpu memory value)
195      (let ([s (cpu-s cpu)])
196        (memory-set-byte! memory (+ #x100 s) (wrap-byte value))
197        (cpu-s-set! cpu (wrap-byte (fx- s 1))))]))
198
199 (define-syntax stack-push-word!
200   (syntax-rules ()
201     [(_ cpu memory value)
202      (begin
203        (stack-push-byte! cpu memory (fxsrl value 8))
204        (stack-push-byte! cpu memory (wrap-byte value)))]))
205
206 (define-syntax stack-pop-byte!
207   (syntax-rules ()
208     [(_ ?cpu ?memory)
209      (let ([cpu ?cpu])
210        (let ([s (wrap-byte (fx+ (cpu-s cpu) 1))])
211          (cpu-s-set! cpu s)
212          (memory-get-byte ?memory (+ #x100 s))))]))
213
214 (define opcode-function-vector
215   (make-vector #xFF
216     (lambda (cpu memory)
217       (errorf 'maf6502 "unimplemented opcode"))))
218
219 (define opcode-lookup
220   (lambda (opcode)
221     (vector-ref opcode-function-vector opcode)))
222
223 (define e
224   (lambda (cpu memory)
225     (do ([opcode (memory-get-byte memory (cpu-pc cpu)) (memory-get-byte memory (cpu-pc cpu))])
226       ((fx= opcode #x0))
227       ((opcode-lookup opcode) cpu memory))))
228
229 (define-syntax define-instruction
230   (syntax-rules (advance-pc: raw-operand: lambda)
231     [(_ mnemonic
232        (advance-pc: advance-pc?)
233        (raw-operand: raw?)
234        ([opcode* bytes* address-mode*] ...)
235        (lambda (cpu memory)
236          (lambda (operand)
237            ?body ?body* ...)))
238      (begin
239        (vector-set! opcode-function-vector opcode*
240          (lambda (cpu memory)
241            ;; NB: advance the PC past the opcode
242            ;; NB: may seem odd, but comes into play with instructions for which
243            ;; NB: the order of operations is specific and important, e.g., BRK
244            (cpu-pc-set! cpu (fx1+ (cpu-pc cpu)))
245            ((lambda (operand)
246               ?body ?body* ...)
247             ;; NB: for conditional branches, the relative address is always
```

line 221 char 6 count 100

- HTML output per source code file
- Expression execution counts on hover

PERFORMANCE DEBUGGING: COST CENTERS

- Instrument specific sections of code
- Gather metrics:
  - Memory allocation
  - Instruction count
  - CPU time

```
(define adc-cc (make-cost-center))
(define-instruction adc
  (advance-pc: #t)
  (raw-operand: #f)
  ([#x6D 3 absolute])
  (lambda (cpu memory)
    (lambda (opnd)
      (with-cost-center #t adc-cc
        (lambda ()
          (let ([result (+ (cpu-a cpu) opnd)])
            (update-flags! cpu
              ([carry (fx> result #xFF)]
               [sign (fxlogbit? 7 result)]
               [zero (fxzero? (wrap-byte result))]))
            (cpu-a-set! cpu (wrap-byte result)))))))))
```

```
> (cost-center-allocation-count adc-cc)
0
> (cost-center-instruction-count adc-cc)
1044
> (cost-center-time adc-cc)
#<time-duration 0.000020801>
>
```

NEXT STEPS

- Write an assembler and disassembler
- Implement the remaining…few…instructions
- Evaluate performance and optimize

BEYOND THE 6502

- Implement other system components like the PPU and sound chip
- Adapt Scheme libraries for graphics and sound to Chez Scheme
- Use Chez Scheme's C FFI to leverage C implementations

WRAPPING UP: THE GOOD

- Data structures—records and bytevectors
- Macros
- Debugging tools—the inspector and tracing
- Performance debugging tools—profiling and cost centers

WRAPPING UP:THE BAD

- Library availability—SRFIs
- Build tools—`make`

WHY CHEZ SCHEME?

- It's consistent—features always compose
- It's fast—great compiler and runtime performance
- It has excellent documentation
- It has a wonderful (if small) community of experts

RESOURCES

- The 6502 Microprocessor resource: http://www.6502.org
- kingcons/cl-6502—a Common Lisp 6502 emulator: https://github.com/kingcons/cl-6502
- Chez Scheme Version 9 User's Guide: http://cisco.github.io/ChezScheme/csug9.5/
- Scheme Requests for Implementation: https://srfi.schemers.org
- akeep/chez-srfi—SRFIs for Chez Scheme: https://github.com/akeep/chez-srfi
- Kent Dybvig's Guide to debugging Chez Scheme programs: https://www.cs.indiana.edu/chezscheme/debug/