cl-6502

Brit Butler Dustin Long Justin Caratzas BigEd b8sell

Contents

1 | Introduction

High-Level Emulation by Example

I started working on **cl-6502** to develop a better mental model of assembly and how CPUs work. The project has evolved into a highly correct, concise 6502 emulator. Its nearest neighbors, and inspirations, are lib6502 and py65 which also make for pretty good reading if you prefer C or Python to Lisp. :)

The MOS 6502

The MOS 6502, as noted in the README, is famous for its usage in:

- the Apple II,
- the Nintendo,
- the Commodore 64,
- the BBC Micro,
- Bender, the Terminator, and elsewhere.

It was, therefore, a large component of the microcomputer boom in the 1980s and helped usher in the Modern PC era we enjoy today. The 6502 also comes from a time when assembly programming was necessary for fast, responsive programs. Consequently, the assembly language was designed with the intent that it would be used by humans at least as often as it would be generated by a compiler for a higher-level language. This makes it ideal for education purposes especially relative to the complex x86 assembly that is prevalent today.

Explicit Goals

cl-6502 does have some explicit technical goals:

- Code size excluding tests should be < 1000 loc. (**0.9.7**: 949 lines)
- Able to run at 8 mhz or faster using a single core on an Intel Core CPU. (**0.9.7**: ~50 mhz)
- Cycle-accurate emulation suitable for use in a full NES emulator.
- Readable as a 6502 introduction for someone with or without a copy of CLHS.

A Word on Performance

Performance is **not** an explicit goal of the **cl-6502** project. Currently, it emulates about 15-20x slower than lib6502, a very fast C implementation by Ian Piumarta, and is 25x faster than the original NES. This has less to do with the choice of language than with my naivete about emulation techniques and the emphasis on readable, safe code. For example, **cl-6502** raises an exception upon encountering illegal opcodes while lib6502 prints a message and continues.

Why Lisp?

Common Lisp was chosen for several reasons:

- It is the language the primary author is most familiar with.
- It has strong support for metaprogramming.
- It supports several paradigms and has good facilities to describe low-level behavior.
- It has native code compilers capable of generating fast code.

Emulation Basics

Emulation isn't particularly different from any other kind of programming. You need to store the state of the hardware you're emulating and provide functions to transform that state based on user input to produce correct results. In **cl-6502**'s case, that means:

- a CPU object,
- functions for assembly instructions and interrupts the CPU can execute,
- some RAM (represented as an array of bytes) to store the emulated program in,
- functions to make the CPU run a single instruction or run until the program halts.

More complete coverage of emulation, including whole-system emulation, can be found in Victor Moya del Barrio's excellent Study of the Techniques for Emulation Programming.

Design Decisions

These are the current major design decisions in **cl-6502**:

- Compile-time macros for all status register handling.
- Type declarations for core data structures and memory access functions.
- Storage of opcode metadata and opcode lambdas in separate arrays.
- A naive interpreter loop rather than Context-Threaded Dispatch, JIT, etc.
- Ignore decimal mode support in ADC/SBC since the NES doesn't require it.

If you get stumped by a macro, I'd encourage you to macroexpand one of its callsites. Don't be afraid to skim something and come back to it later, either. Most importantly, I am *very open* to feedback of any kind on this text or the code itself. If you find something confusing, feel free to email and ask why I wrote it that way or suggest alternatives. Patches welcome!

With all that out of the way, let's dive in!

2 | Addressing Modes

`addressing.lisp`: The Addressing Mode Protocol

References:

- Addressing Modes

Addressing Modes are perhaps *the* major difference between high-level languages and assembly. In practically every high-level language, you define named variables to store your data in and then access your data by name. In assembly, you store data at a memory address and interact with the address only. To interact with a given address, you use Addressing Modes.

Conceptually, an addressing mode is just a function that takes a number and returns an index into memory. Each addressing mode indexes into memory differently and comes with a unique syntax which we'll define readers and writers for to aid with assembly and disassembly.

The 6502 has 13 addressing modes that can be roughly divided into 5 groups based on what part of memory they access:

1. Basic: CPU registers - implied, immediate, accumulator
2. Zero-Page: The bottom 256 bytes of RAM - zero-page, zero-page-x, zero-page-y
3. Absolute: Any address in RAM - absolute, absolute-x, absolute-y
4. Indirect: The address stored at another address - indirect, indirect-x, indirect-y
5. Relative: Conditionally go forwards or backwards - relative

To make things more complicated, you can't just pass any address to any CPU instruction. Each CPU instruction supports a limited number of addressing modes. You'll note when we start defining opcodes that certain addressing modes take more CPU time than others. On many older systems, a large part of writing fast assembly code is figuring out how to layout your program's data in memory so you can use the fastest addressing mode possible to access it.

A functioning emulator needs to know how to parse assembly, print disassebly, get data, and set data for each addressing mode. **cl-6502** uses methods on unique symbols for reading and printing, and functions for getting and setting. The reader is a regular expression, with a placeholder character (underscore) from which a number, label or expression is extracted. The printer is a lisp format string that effectively inverts the process. Since the getter and setter need to work on the same address in memory, and use the same code to compute that address, we'll use a `defaddress` macro to factor out their shared code.

Source Code

The Protocol

```
(in-package :6502)

(defparameter *address-modes* (make-hash-table :test 'equal))

(defgeneric reader (mode)
  (:documentation "Return a Perl-compatible regex suitable for parsing MODE.")
  (:method (mode) (error 'invalid-mode :mode mode)))

(defgeneric writer (mode)
  (:documentation "Return a format string suitable for printing MODE.")
  (:method (mode) (error 'invalid-mode :mode mode)))

(defmacro defaddress (name (&key reader writer cpu-reg) &body body)
  "Define an Addressing Mode, NAME, with READER and WRITER methods that take
   NAME as a symbol and return a regex or format expression, respectively,
   and a function and setf function to get and set the data pointed to by the
   given mode."
  `(progn
     (defmethod reader ((mode (eql ',name)))
       ,(cl-ppcre:regex-replace-all "_" reader "([^,()#&]+)"))
     (defmethod writer ((mode (eql ',name))) ,writer)
     (setf (gethash ',name *address-modes*) t)
     (defun ,name (cpu) ,@body)
     (defun (setf ,name) (value cpu)
       ,(if cpu-reg
            `(setf ,@body value)
            `(setf (get-byte ,@body) value)))))

(defun make-getter (name mode raw-p)
  "Generate an appropriate GETTER for NAME based on RAW-P
and whether or not it is a register shift operation."
  (let ((register-shift-op-p (cl:and (member name '(asl lsr rol ror))
                                     (eql mode 'accumulator))))
    (if (or raw-p register-shift-op-p)
        `(,mode cpu)
        `(get-byte (,mode cpu)))))
```

Addressing Modes

```
(defaddress implied (:reader "^$"
                     :writer "")
  nil)

(defaddress accumulator (:reader "^[aA]$"
                         :writer "A"
                         :cpu-reg t)
  (cpu-ar cpu))

(defaddress immediate (:reader "^#_$"
                       :writer "~{#$~2,'0x~}"
                       :cpu-reg t)
  (cpu-pc cpu))
```

```
(defaddress zero-page (:reader ”^_$”
                       :writer ”~{$~2,’0x~}”)
  (get-byte (cpu-pc cpu)))

(defaddress zero-page-x (:reader ”^_,\\s*[xX]$”
                         :writer ”$~{~2,’0x~}, X”)
  (wrap-byte (+ (get-byte (cpu-pc cpu)) (cpu-xr cpu))))

(defaddress zero-page-y (:reader ”^_,\\s*[yY]$”
                         :writer ”$~{~2,’0x~}, Y”)
  (wrap-byte (+ (get-byte (cpu-pc cpu)) (cpu-yr cpu))))

(defaddress absolute (:reader ”^_$”
                      :writer ”$~{~2,’0x~}”)
  (get-word (cpu-pc cpu)))

(defaddress absolute-x (:reader ”^_,\\s*[xX]$”
                        :writer ”$~{~2,’0x~}, X”)
  (let ((result (wrap-word (+ (get-word (cpu-pc cpu)) (cpu-xr cpu)))))
    (maybe-update-cycle-count cpu result)
    result))

(defaddress absolute-y (:reader ”^_,\\s*[yY]$”
                        :writer ”$~{~2,’0x~}, Y”)
  (let ((result (wrap-word (+ (get-word (cpu-pc cpu)) (cpu-yr cpu)))))
    (maybe-update-cycle-count cpu result)
    result))

(defaddress indirect (:reader ”^\\(_\\)$”
                      :writer ”($~{~2,’0x~})”)
  (get-word (get-word (cpu-pc cpu)) t))

(defaddress indirect-x (:reader ”^\\(_\\),\\s*[xX]$”
                        :writer ”($~{~2,’0x~}), X”)
  (get-word (wrap-byte (+ (get-byte (cpu-pc cpu)) (cpu-xr cpu))) t))

(defaddress indirect-y (:reader ”^\\(_\\),\\s*[yY]$”
                        :writer ”($~{~2,’0x~}), Y”)
  (let* ((addr (get-word (get-byte (cpu-pc cpu)) t))
         (result (wrap-word (+ addr (cpu-yr cpu)))))
    (maybe-update-cycle-count cpu result addr)
    result))

(defaddress relative (:reader ”^(&?_)$”
                      :writer ”&~{~2,’0x~}”)
  (let ((offset (get-byte (cpu-pc cpu))))
    (incf (cpu-cc cpu))
    (let ((result (if (logbitp 7 offset)
                      (wrap-word (- (cpu-pc cpu) (- #xff offset)))
                      (wrap-word (+ (cpu-pc cpu) (1+ offset))))))
      (maybe-update-cycle-count cpu result (1+ (cpu-pc cpu)))
      result)))
```

3 | CPU

`cpu.lisp`: The 6502 VM

References:

- General
- Registers

Loosely, the idea of `cpu.lisp` is to define a simple VM the rest of the emulator's behavior is defined in terms of. The first thing to do is construct a model of the CPU's state. Like all CPUs, the 6502 has a Program Counter pointing to the next instruction to execute. The Program Counter (or PC) is 16 bits, meaning the 6502 can address 64k of RAM. It also has Stack Pointer, Accumulator, X, and Y registers each of which is a single byte. There is a Status Register with information about the result of the last computation. For emulation purposes, we'll also add a cycle counter to track CPU execution time.

Once data structures for the CPU and RAM are defined, we'll want helper functions for the rest of the emulator to transform them. Common operations we'll want to support include:

- Getting and setting a byte/word in RAM.
- Pushing and popping items off the stack.
- Getting and setting individual bits in the status register.
- Resetting the state of the system.
- Executing an NMI.

We'll also add a few things to help ensure the correctness of our emulator:

- Wrappers to ensure that bytes and words don't overflow.
- Type definitions for a byte (u8) and machine word (u16).
- A macro to simplify the definition of opcodes.

Most other things we'll need to define our opcodes will be built in language primitives like arithmetic, conditionals, etc.

Source Code

Core Data Types

```
(in-package :6502)
```

```
(deftype u8 () ’(unsigned-byte 8))
(deftype u16 () ’(unsigned-byte 16))

(defstruct cpu
  ”A 6502 CPU with an extra slot for tracking the cycle count/clock ticks.”
  (pc #xfffc :type u16)                ;; program counter
  (sp #xfd   :type u8)                 ;; stack pointer
  (sr #x24   :type u8)                 ;; status register
  (xr 0      :type u8)                 ;; x register
  (yr 0      :type u8)                 ;; y register
  (ar 0      :type u8)                 ;; accumulator
  (cc 0      :type fixnum))            ;; cycle counter

(defmethod initialize-instance :after ((cpu cpu) &key)
  (setf (cpu-pc cpu) (absolute cpu)))

(defun bytevector (size)
  ”Return an array of the given SIZE with element-type u8.”
  (make-array size :element-type ’u8))
```

Tasty Globals

```
(declaim (type (simple-array u8 (#x10000)) *ram*))
(defparameter *ram* (bytevector #x10000)
  ”A lovely hunk of bytes.”)

(defparameter *cpu* (make-cpu)
  ”The 6502 instance used by default during execution.”)

(declaim (type (simple-vector 256) *opcode-funs*))
(defparameter *opcode-funs* (make-array #x100 :element-type ’(or function null))
  ”The opcode lambdas used during emulation.”)

(defparameter *opcode-meta* (make-array #x100 :initial-element nil)
  ”A mapping of opcodes to metadata lists.”)
```

Helpers

```
(defgeneric reset (obj)
  (:documentation ”Reset the OBJ to an initial state.”)
  (:method (obj) (initialize-instance obj)))

(defgeneric nmi (obj)
  (:documentation ”Generate a non-maskable interrupt. Used for vblanking in NES.”)
  (:method (obj)
    (stack-push-word (cpu-pc obj) obj)
    (stack-push (cpu-sr obj) obj)
    (setf (cpu-pc obj) (get-word #xfffa))))

(declaim (inline wrap-byte wrap-word wrap-page)
         (ftype (function (fixnum) u8) wrap-byte))
(defun wrap-byte (value)
  ”Wrap VALUE so it conforms to (typep value ’u8), i.e. a single byte.”
  (logand value #xff))
```

```
(declaim (ftype (function (fixnum) u16) wrap-word))
(defun wrap-word (value)
  ”Wrap VALUE so it conforms to (typep value ’u16), i.e. a machine word.”
  (logand value #xffff))

(defun wrap-page (address)
  ”Wrap the given ADDRESS, ensuring that we don’t cross a page boundary.
e.g. If we (get-word address).”
  (+ (logand address #xff00) (logand (1+ address) #xff)))

(declaim (ftype (function (u16) u8) get-byte))
(defun get-byte (address)
  ”Get a byte from RAM at the given ADDRESS.”
  (aref *ram* address))

(defun (setf get-byte) (new-val address)
  ”Set ADDRESS in *ram* to NEW-VAL.”
  (setf (aref *ram* address) new-val))

(defun get-word (address &optional wrap-p)
  ”Get a word from RAM starting at the given ADDRESS.”
  (+ (get-byte address)
     (ash (get-byte (if wrap-p (wrap-page address) (1+ address))) 8)))

(defun (setf get-word) (new-val address)
  ”Set ADDRESS and (1+ ADDRESS) in *ram* to NEW-VAL, little endian ordering.”
  (setf (get-byte address) (wrap-byte (ash new-val -8))
        (get-byte (1+ address)) (wrap-byte new-val)))

(defun get-range (start &optional end)
  ”Get a range of bytes from RAM, starting from START and stopping at END if
provided.”
  (subseq *ram* start end))

(defun (setf get-range) (bytevector start)
  ”Replace the contents of RAM, starting from START with BYTEVECTOR.”
  (setf (subseq *ram* start (+ start (length bytevector))) bytevector))

(declaim (inline stack-push stack-pop))
(defun stack-push (value cpu)
  ”Push the byte VALUE on the stack and decrement the SP.”
  (setf (get-byte (+ (cpu-sp cpu) #x100)) (wrap-byte value))
  (setf (cpu-sp cpu) (wrap-byte (1- (cpu-sp cpu)))))

(defun stack-push-word (value cpu)
  ”Push the 16-bit word VALUE onto the stack.”
  (stack-push (wrap-byte (ash value -8)) cpu)
  (stack-push (wrap-byte value) cpu))

(defun stack-pop (cpu)
  ”Pop the value pointed to by the SP and increment the SP.”
  (setf (cpu-sp cpu) (wrap-byte (1+ (cpu-sp cpu))))
  (get-byte (+ (cpu-sp cpu) #x100)))

(defun stack-pop-word (cpu)
  ”Pop a 16-bit word off the stack.”
  (+ (stack-pop cpu) (ash (stack-pop cpu) 8)))

(defmacro defenum (name (&rest keys))
  ”Define a function named %NAME, that takes KEY as an arg and returns the
index of KEY. KEYS should be scalar values.”
```

```
  (let ((enum (make-hash-table)))
    (loop for i = 0 then (1+ i)
       for key in keys
       do (setf (gethash key enum) i))
    ‘(defun ,(intern (format nil ”%~:@(~A~)” name)) (key)
       (let ((enum ,enum))
         (gethash key enum)))))

(eval-when (:compile-toplevel :load-toplevel :execute)
  (defenum status-bit (:carry :zero :interrupt :decimal
                       :break :unused :overflow :negative)))

(defmacro status-bit (key)
  ”Test if KEY is set in the status register. KEY should be a keyword.”
  ‘(logand (cpu-sr cpu) ,(ash 1 (%status-bit key))))

(defmacro set-status-bit (key new-val)
  ”Set bit KEY in the status reg to NEW-VAL. KEY should be a keyword.”
  ‘(setf (ldb (byte 1 ,(%status-bit key)) (cpu-sr cpu)) ,new-val))

(defmacro set-flags-if (&rest flag-preds)
  ”Takes any even number of arguments where the first is a keyword denoting a
status bit and the second is a funcallable predicate that takes no arguments.
It will set each flag to 1 if its predicate is true, otherwise 0.”
  ‘(progn
     ,@(loop for (flag pred . nil) on flag-preds by #’cddr
          collecting ‘(set-status-bit ,flag (if ,pred 1 0)))))

(defun overflow-p (result reg mem)
  ”Checks whether the sign of RESULT is found in the signs of REG or MEM.”
  (flet ((sign-of (x) (logbitp 7 x)))
    (not (or (eql (sign-of result) (sign-of reg))
             (eql (sign-of result) (sign-of mem))))))

(defun maybe-update-cycle-count (cpu address &optional start)
  ”If ADDRESS crosses a page boundary, add an extra cycle to CPU’s count. If
START is provided, test that against ADDRESS. Otherwise, use the absolute address.”
  (let ((operand (or start (absolute cpu))))
    (declare (type u16 operand)
             (type u16 address)
             (type (or null u16) start))
    (when (not (= (logand operand #xff00)
                  (logand address #xff00)))
      (incf (cpu-cc cpu)))))

(defmacro branch-if (predicate)
  ”Take a Relative branch if PREDICATE is true, otherwise increment the PC.”
  ‘(if ,predicate
       (setf (cpu-pc cpu) (relative cpu))
       (incf (cpu-pc cpu))))

(defun rotate-byte (integer count cpu)
  ”Rotate the bits of INTEGER by COUNT. If COUNT is negative, rotate right.”
  (let ((result (ash integer count)))
    (if (plusp (status-bit :carry))
        (ecase count
          (01 (logior result #x01))
          (-1 (logior result #x80)))
        result)))
```

Opcode Macrology

```
(defmacro defasm (name (&key (docs ””) raw-p (track-pc t))
                  modes &body body)
  ”Define a 6502 instruction NAME, storing its DOCS and metadata in *opcode-meta*,
and a lambda that executes BODY in *opcode-funs*. Within BODY, the functions
GETTER and SETTER can be used to get and set values for the current addressing
mode, respectively. TRACK-PC can be passed nil to disable program counter updates
for branching/jump operations. If RAW-P is true, GETTER will return the mode’s
address directly, otherwise it will return the byte at that address. MODES is a
list of opcode metadata lists: (opcode cycles bytes mode).”
  ‘(progn
     ,@(loop for (op cycles bytes mode) in modes collect
            ‘(setf (aref *opcode-meta* ,op) ’,(list name docs cycles bytes mode)))
     ,@(loop for (op cycles bytes mode) in modes collect
            ‘(setf (aref *opcode-funs* ,op)
                   (named-lambda ,(intern (format nil ”~A-~X” name op)) (cpu)
                     (incf (cpu-pc cpu))
                     (flet ((getter ()
                              ,(make-getter name mode raw-p))
                            (setter (x)
                              (setf (,mode cpu) x)))
                       ,@body)
                     ,@(when track-pc
                         ‘((incf (cpu-pc cpu) (1- ,bytes))))
                     (incf (cpu-cc cpu) ,cycles))))))
```

4 | Opcode Emulation

`opcodes.lisp`: **Enter defasm**

References:

- Opcodes
- Py65

Now comes the core of the project, the CPU opcode definitions. The 6502 has 56 instructions, each with several addressing modes. We'll use our `defasm` macro to define each instruction with all its variants in one place. The variants are given as a list of lists where each variant is specified like so:

```
(opcode-byte cycles bytes addressing-mode)
```

The `opcode-byte` is the how the opcode is stored in memory, the `cycles` are how long it takes the CPU to execute, and the `bytes` are how many bytes in RAM the opcode *and* its arguments use.

You might have wondered why in `defasm` we increment the Program Counter by 1, then *later* check `track-pc` before incrementing the rest. Imagining the opcode's execution makes it obvious. The PC points at the opcode, then we increment the program counter to point at the first argument. The instruction BODY can run without worrying about argument offsets. Afterwards, if there were no arguments, we're already pointing at the next instruction. Otherwise, we just bump the PC past the arguments.

If you were reading carefully earlier, you noticed that we wrap the body of `defasm` in a FLET that defines the `getter` and `setter` functions to access memory for the opcode's addressing mode. It's advantageous for performance to compute as much as possible at compile-time, so we have `make-getter` compute a custom body for the GETTER in `defasm`.

`make-getter` is there only because, unlike **all** the other instructions, shifts and rotations (i.e. ASL, LSR, ROL, and ROR), use "raw" addressing in their accumulator mode but "normal" addressing everywhere else. `make-getter` takes the instruction mnemonic and addressing mode and factors out the special casing. Aren't you glad I already hunted those bugs down?

Source Code

```
(in-package :6502)

(defasm adc (:docs "Add to Accumulator with Carry")
    ((#x61 6 2 indirect-x)
     (#x65 3 2 zero-page)
     (#x69 2 2 immediate)
     (#x6d 4 3 absolute)
     (#x71 5 2 indirect-y)
     (#x75 4 2 zero-page-x)
     (#x79 4 3 absolute-y)
     (#x7d 4 3 absolute-x))
  (let ((result (+ (cpu-ar cpu) (getter) (status-bit :carry))))
    (set-flags-if :carry (> result #xff)
                  :overflow (overflow-p result (cpu-ar cpu) (getter))
                  :negative (logbitp 7 result)
                  :zero (zerop (wrap-byte result)))
    (setf (cpu-ar cpu) (wrap-byte result))))

(defasm and (:docs "And with Accumulator")
    ((#x21 6 2 indirect-x)
     (#x25 3 2 zero-page)
     (#x29 2 2 immediate)
     (#x2d 4 3 absolute)
     (#x31 5 2 indirect-y)
     (#x35 4 2 zero-page-x)
     (#x39 4 3 absolute-y)
     (#x3d 4 3 absolute-x))
  (let ((result (setf (cpu-ar cpu) (logand (cpu-ar cpu) (getter)))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm asl (:docs "Arithmetic Shift Left")
    ((#x06 5 2 zero-page)
     (#x0a 2 1 accumulator)
     (#x0e 6 3 absolute)
     (#x16 6 2 zero-page-x)
     (#x1e 7 3 absolute-x))
  (let* ((operand (getter))
         (result (wrap-byte (ash operand 1))))
    (set-flags-if :carry (logbitp 7 operand)
                  :zero (zerop result)
                  :negative (logbitp 7 result))
    (setter result)))

(defasm bcc (:docs "Branch on Carry Clear" :track-pc nil)
    ((#x90 2 2 relative))
  (branch-if (zerop (status-bit :carry))))

(defasm bcs (:docs "Branch on Carry Set" :track-pc nil)
    ((#xb0 2 2 relative))
  (branch-if (plusp (status-bit :carry))))

(defasm beq (:docs "Branch if Equal" :track-pc nil)
    ((#xf0 2 2 relative))
  (branch-if (plusp (status-bit :zero))))

(defasm bit (:docs "Test Bits in Memory with Accumulator")
    ((#x24 3 2 zero-page)
     (#x2c 4 3 absolute))
  (let ((operand (getter)))
    (set-flags-if :zero (zerop (logand (cpu-ar cpu) operand))
                  :negative (logbitp 7 operand)
                  :overflow (logbitp 6 operand))))

(defasm bmi (:docs "Branch on Negative Result" :track-pc nil)
    ((#x30 2 2 relative))
  (branch-if (plusp (status-bit :negative))))
```

```
(defasm bne (:docs ”Branch if Not Equal” :track-pc nil)
    ((#xd0 2 2 relative))
  (branch-if (zerop (status-bit :zero))))

(defasm bpl (:docs ”Branch on Positive Result” :track-pc nil)
    ((#x10 2 2 relative))
  (branch-if (zerop (status-bit :negative))))

(defasm brk (:docs ”Force Break”)
    ((#x00 7 1 implied))
  (let ((pc (wrap-word (1+ (cpu-pc cpu)))))
    (stack-push-word pc cpu)
    (set-status-bit :break 1)
    (stack-push (cpu-sr cpu) cpu)
    (set-status-bit :interrupt 1)
    (setf (cpu-pc cpu) (get-word #xfffe))))

(defasm bvc (:docs ”Branch on Overflow Clear” :track-pc nil)
    ((#x50 2 2 relative))
  (branch-if (zerop (status-bit :overflow))))

(defasm bvs (:docs ”Branch on Overflow Set” :track-pc nil)
    ((#x70 2 2 relative))
  (branch-if (plusp (status-bit :overflow))))

(defasm clc (:docs ”Clear Carry Flag”)
    ((#x18 2 1 implied))
  (set-status-bit :carry 0))

(defasm cld (:docs ”Clear Decimal Flag”)
    ((#xd8 2 1 implied))
  (set-status-bit :decimal 0))

(defasm cli (:docs ”Clear Interrupt Flag”)
    ((#x58 2 1 implied))
  (set-status-bit :interrupt 0))

(defasm clv (:docs ”Clear Overflow Flag”)
    ((#xb8 2 1 implied))
  (set-status-bit :overflow 0))

(defasm cmp (:docs ”Compare Memory with Accumulator”)
    ((#xc1 6 2 indirect-x)
     (#xc5 3 2 zero-page)
     (#xc9 2 2 immediate)
     (#xcd 4 3 absolute)
     (#xd1 5 2 indirect-y)
     (#xd5 4 2 zero-page-x)
     (#xd9 4 3 absolute-y)
     (#xdd 4 3 absolute-x))
  (let ((result (- (cpu-ar cpu) (getter))))
    (set-flags-if :carry (not (minusp result))
                  :zero (zerop result)
                  :negative (logbitp 7 result))))

(defasm cpx (:docs ”Compare Memory with X register”)
    ((#xe0 2 2 immediate)
     (#xe4 3 2 zero-page)
     (#xec 4 3 absolute))
  (let ((result (- (cpu-xr cpu) (getter))))
    (set-flags-if :carry (not (minusp result))
                  :zero (zerop result)
                  :negative (logbitp 7 result))))

(defasm cpy (:docs "Compare Memory with Y register")
    ((#xc0 2 2 immediate)
     (#xc4 3 2 zero-page)
     (#xcc 4 3 absolute))
  (let ((result (- (cpu-yr cpu) (getter))))
    (set-flags-if :carry (not (minusp result))
                  :zero (zerop result)
                  :negative (logbitp 7 result))))

(defasm dec (:docs "Decrement Memory")
    ((#xc6 5 2 zero-page)
     (#xce 6 3 absolute)
     (#xd6 6 2 zero-page-x)
     (#xde 7 3 absolute-x))
  (let ((result (wrap-byte (1- (getter)))))
    (setter result)
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm dex (:docs "Decrement X register")
    ((#xca 2 1 implied))
  (let ((result (setf (cpu-xr cpu) (wrap-byte (1- (cpu-xr cpu))))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm dey (:docs "Decrement Y register")
    ((#x88 2 1 implied))
  (let ((result (setf (cpu-yr cpu) (wrap-byte (1- (cpu-yr cpu))))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm eor (:docs "Exclusive OR with Accumulator")
    ((#x41 6 2 indirect-x)
     (#x45 3 2 zero-page)
     (#x49 2 2 immediate)
     (#x4d 4 3 absolute)
     (#x51 5 2 indirect-y)
     (#x55 4 2 zero-page-x)
     (#x59 4 3 absolute-y)
     (#x5d 4 3 absolute-x))
  (let ((result (setf (cpu-ar cpu) (logxor (getter) (cpu-ar cpu)))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm inc (:docs "Increment Memory")
    ((#xe6 5 2 zero-page)
     (#xee 6 3 absolute)
     (#xf6 6 2 zero-page-x)
     (#xfe 7 3 absolute-x))
  (let ((result (wrap-byte (1+ (getter)))))
    (setter result)
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm inx (:docs "Increment X register")
    ((#xe8 2 1 implied))
  (let ((result (setf (cpu-xr cpu) (wrap-byte (1+ (cpu-xr cpu))))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm iny (:docs "Increment Y register")
    ((#xc8 2 1 implied))
  (let ((result (setf (cpu-yr cpu) (wrap-byte (1+ (cpu-yr cpu))))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm jmp (:docs "Jump Unconditionally" :raw-p t :track-pc nil)
    ((#x4c 3 3 absolute)
     (#x6c 5 3 indirect))
  (setf (cpu-pc cpu) (getter)))

(defasm jsr (:docs "Jump to Subroutine" :raw-p t :track-pc nil)
    ((#x20 6 3 absolute))
  (stack-push-word (wrap-word (1+ (cpu-pc cpu))) cpu)
  (setf (cpu-pc cpu) (getter)))

(defasm lda (:docs "Load Accumulator from Memory")
    ((#xa1 6 2 indirect-x)
     (#xa5 3 2 zero-page)
     (#xa9 2 2 immediate)
     (#xad 4 3 absolute)
     (#xb1 5 2 indirect-y)
     (#xb5 4 2 zero-page-x)
     (#xb9 4 3 absolute-y)
     (#xbd 4 3 absolute-x))
  (let ((result (setf (cpu-ar cpu) (getter))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm ldx (:docs "Load X register from Memory")
    ((#xa2 2 2 immediate)
     (#xa6 3 2 zero-page)
     (#xae 4 3 absolute)
     (#xb6 4 2 zero-page-y)
     (#xbe 4 3 absolute-y))
  (let ((result (setf (cpu-xr cpu) (getter))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm ldy (:docs "Load Y register from Memory")
    ((#xa0 2 2 immediate)
     (#xa4 3 2 zero-page)
     (#xac 4 3 absolute)
     (#xbc 4 3 absolute-x)
     (#xb4 4 2 zero-page-x))
  (let ((result (setf (cpu-yr cpu) (getter))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm lsr (:docs "Logical Shift Right")
    ((#x46 5 2 zero-page)
     (#x4a 2 1 accumulator)
     (#x4e 6 3 absolute)
     (#x56 6 2 zero-page-x)
     (#x5e 7 3 absolute-x))
  (let* ((operand (getter))
         (result (ash operand -1)))
    (set-flags-if :carry (logbitp 0 operand)
                  :zero (zerop result)
                  :negative (logbitp 7 result))
    (setter result)))

(defasm nop (:docs "No Operation")
    ((#xea 2 1 implied))
  nil)

(defasm ora (:docs "Bitwise OR with Accumulator")
    ((#x01 6 2 indirect-x)
     (#x05 3 2 zero-page)
     (#x09 2 2 immediate)
     (#x0d 4 3 absolute)
     (#x11 5 2 indirect-y)
     (#x15 4 2 zero-page-x)
     (#x19 4 3 absolute-y)
     (#x1d 4 3 absolute-x))
  (let ((result (setf (cpu-ar cpu) (logior (cpu-ar cpu) (getter)))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm pha (:docs "Push Accumulator")
    ((#x48 3 1 implied))
  (stack-push (cpu-ar cpu) cpu))

(defasm php (:docs "Push Processor Status")
    ((#x08 3 1 implied))
  (stack-push (logior (cpu-sr cpu) #x10) cpu))

(defasm pla (:docs "Pull Accumulator from Stack")
    ((#x68 4 1 implied))
  (let ((result (setf (cpu-ar cpu) (stack-pop cpu))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm plp (:docs "Pull Processor Status from Stack")
    ((#x28 4 1 implied))
  (let ((result (logior (stack-pop cpu) #x20)))
    (setf (cpu-sr cpu) (logandc2 result #x10))))

(defasm rol (:docs "Rotate Left")
    ((#x2a 2 1 accumulator)
     (#x26 5 2 zero-page)
     (#x2e 6 3 absolute)
     (#x36 6 2 zero-page-x)
     (#x3e 7 3 absolute-x))
  (let* ((operand (getter))
         (result (wrap-byte (rotate-byte operand 1 cpu))))
    (setter result)
    (set-flags-if :carry (logbitp 7 operand)
                  :zero (zerop result)
                  :negative (logbitp 7 result))))

(defasm ror (:docs "Rotate Right")
    ((#x66 5 2 zero-page)
     (#x6a 2 1 accumulator)
     (#x6e 6 3 absolute)
     (#x76 6 2 zero-page-x)
     (#x7e 7 3 absolute-x))
  (let* ((operand (getter))
         (result (wrap-byte (rotate-byte operand -1 cpu))))
    (setter result)
    (set-flags-if :carry (logbitp 0 operand)
                  :zero (zerop result)
                  :negative (logbitp 7 result))))

(defasm rti (:docs "Return from Interrupt")
    ((#x40 6 1 implied))
  (setf (cpu-sr cpu) (logior (stack-pop cpu) #x20))
  (setf (cpu-pc cpu) (stack-pop-word cpu)))

(defasm rts (:docs "Return from Subroutine" :track-pc nil)
    ((#x60 6 1 implied))
  (setf (cpu-pc cpu) (1+ (stack-pop-word cpu))))

(defasm sbc (:docs "Subtract from Accumulator with Carry")
    ((#xe1 6 2 indirect-x)
     (#xe5 3 2 zero-page)
     (#xe9 2 2 immediate)
     (#xed 4 3 absolute)
     (#xf1 5 2 indirect-y)
     (#xf5 4 2 zero-page-x)
     (#xf9 4 3 absolute-y)
     (#xfd 4 3 absolute-x))
  (flet ((flip-bit (position x) (logxor (expt 2 position) x)))
    (let* ((operand (getter))
           (carry-bit (flip-bit 0 (status-bit :carry)))
           (result (- (cpu-ar cpu) operand carry-bit)))
      (set-flags-if :zero (zerop (wrap-byte result))
                    :overflow (overflow-p result (cpu-ar cpu) (flip-bit 7 operand))
                    :negative (logbitp 7 result)
                    :carry (not (minusp result)))
      (setf (cpu-ar cpu) (wrap-byte result)))))

(defasm sec (:docs "Set Carry Flag")
    ((#x38 2 1 implied))
  (set-status-bit :carry 1))

(defasm sed (:docs "Set Decimal Flag")
    ((#xf8 2 1 implied))
  (set-status-bit :decimal 1))

(defasm sei (:docs "Set Interrupt Flag")
    ((#x78 2 1 implied))
  (set-status-bit :interrupt 1))

(defasm sta (:docs "Store Accumulator")
    ((#x81 6 2 indirect-x)
     (#x85 3 2 zero-page)
     (#x8d 4 3 absolute)
     (#x91 6 2 indirect-y)
     (#x95 4 2 zero-page-x)
     (#x99 5 3 absolute-y)
     (#x9d 5 3 absolute-x))
  (setter (cpu-ar cpu)))

(defasm stx (:docs "Store X register")
    ((#x86 3 2 zero-page)
     (#x8e 4 3 absolute)
     (#x96 4 2 zero-page-y))
  (setter (cpu-xr cpu)))

(defasm sty (:docs "Store Y register")
    ((#x84 3 2 zero-page)
     (#x8c 4 3 absolute)
     (#x94 4 2 zero-page-x))
  (setter (cpu-yr cpu)))

(defasm tax (:docs "Transfer Accumulator to X register")
    ((#xaa 2 1 implied))
  (let ((result (setf (cpu-xr cpu) (cpu-ar cpu))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm tay (:docs "Transfer Accumulator to Y register")
    ((#xa8 2 1 implied))
  (let ((result (setf (cpu-yr cpu) (cpu-ar cpu))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm tsx (:docs ”Transfer Stack Pointer to X register”)
    ((#xba 2 1 implied))
  (let ((result (setf (cpu-xr cpu) (cpu-sp cpu))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm txa (:docs ”Transfer X register to Accumulator”)
    ((#x8a 2 1 implied))
  (let ((result (setf (cpu-ar cpu) (cpu-xr cpu))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))

(defasm txs (:docs ”Transfer X register to Stack Pointer”)
    ((#x9a 2 1 implied))
  (setf (cpu-sp cpu) (cpu-xr cpu)))

(defasm tya (:docs ”Transfer Y register to Accumulator”)
    ((#x98 2 1 implied))
  (let ((result (setf (cpu-ar cpu) (cpu-yr cpu))))
    (set-flags-if :zero (zerop result) :negative (logbitp 7 result))))
```

5 | Exceptional Conditions

`conditions.lisp`: **Just in Case**

There’s really not much to say about our conditions. The primary illegal state our emulator can encounter is trying to execute an invalid opcode, i.e. a byte for which no opcode definition exists. For our assembler, we have also defined a condition for invalid syntax. Finally, there is a condition for invalid addressing modes, mostly to provide better debugging in case a new opcode definition was fat-fingered.

Source Code

```
(in-package :6502)

(define-condition illegal-opcode ()
  ((opcode :initarg :opcode :reader opcode))
  (:report (lambda (condition stream)
             (format stream ”~X is not a legal opcode.” (opcode condition))))
  (:documentation ”Illegal opcodes are not currently implemented.”))

(define-condition invalid-syntax ()
  ((line :initarg :line :reader line))
  (:report (lambda (condition stream)
             (format stream ”Syntax for line ~S is invalid.” (line condition))))
  (:documentation ”Assembly must conform to the syntax in the README.”))

(define-condition invalid-mode ()
  ((mode :initarg :mode :reader mode))
  (:report (lambda (condition stream)
             (format stream ”~A is not a valid addressing mode.” (mode condition))))
  (:documentation ”Only the 6502 addressing modes have readers and printers.”))
```

6 | Stepping and Execution

`utils.lisp`: Bringing it all Together

Now we need to actually connect the dots and get our CPU emulating! An `EXECUTE` function that emulates code in a loop until a halt instruction and a `STEP-CPU` function to execute a single opcode suit nicely. Here we see that our opcodes array finally gets put to good use as `STEP-CPU` just grabs the appropriate lambda and calls it, leaving `EXECUTE` to handle errors if necessary.

Source Code

```
(in-package :6502)

(defun execute (cpu)
  ”Step the CPU until a BRK instruction.”
  (loop for opcode of-type u8 = (get-byte (cpu-pc cpu))
     do (handler-case (step-cpu cpu opcode)
          (undefined-function ()
            (error ’illegal-opcode :opcode opcode)))
     until (zerop opcode)))

(defun step-cpu (cpu opcode)
  ”Step the CPU through the next OPCODE.”
  (funcall (aref *opcode-funs* opcode) cpu))
```

7 | A Naive JIT

`jit.lisp`: “JIT” is a bit generous, isn’t it?

Proper JITs involve much careful thinking and engineering to get good results. This “jit” was more of a quick, fun Sunday evening hack. The whole point of a compiler is to rewrite code into a faster form that preserves the original semantics. That entails doing analysis to understand the code and transformations to improve the parts the compiler understands. As littledan says, every compiler has a “special style” that it knows how to optimize. Thus, speeding up a language really corresponds to increasing the parts of the language the compiler understands, growing the special style. All this JIT does is eliminate the opcode dispatch between branches. It never invalidates cached code which means that if a program modifies itself while running we ignore it, producing incorrect results. As Mike Pall has written, a fast interpreter can outrun a naive compiler. **cl-6502** sticks with interpretation. I just think the JIT is too cute to remove.

Source Code

```
(in-package :6502)

(defvar *jit-cache* (make-hash-table)
  "The JIT's hot code cache. Currently never invalidated.")

(defun get-basic-block (cpu)
  "Get the opcodes from the current PC to the next branch."
  (flet ((op-length (x) (fourth (aref *opcode-meta* x))))
    (loop for pc = (cpu-pc cpu) then (+ pc (op-length op))
       for op = (get-byte pc) collect op
       until (member op '(#x90 #xb0 #xf0 #x30 #xd0 #x10 #x50 #x70
                          #x00 #x4c #x6c #x20 #x40 #x60)))))

(defun jit-block (opcodes)
  "Given a list of opcodes, JIT compile an equivalent function."
  (flet ((jit-op (x) `(funcall ,(aref *opcode-funs* x) cpu)))
    (compile nil `(lambda (cpu) ,@(mapcar #'jit-op opcodes)))))

(defun jit-step (cpu pc)
  "If the current block has been JIT compiled, run it, otherwise JIT compile it."
  (alexandria:if-let (fn (gethash pc *jit-cache*))
    (funcall fn cpu)
    (let ((code (jit-block (get-basic-block cpu))))
      (setf (gethash pc *jit-cache*) code)
      (funcall code cpu))))
```

8 | Taking Apart Old Code

`disassemble.lisp`: When one is Lost

Disassembly actually isn't very tricky…at least on the 6502. Conceptually, you just want to specify a range of bytes to be viewed as code rather than data and print its representation. Our storage of metadata in the opcodes array makes it trivial. We simply loop from start to end, always incrementing by the length of the opcode rather than 1.

Disassembling a single instruction at a given index works by reading the byte, retrieving the matching entry in the opcodes array, and either printing it using the Addressing Mode's writer or returning a list representing the instruction.

We use keywords rather than symbols for the lispy syntax since `#$` is a standard token in 6502 assembly. There is no way to support the standard syntax for indirect-addressed code in lisp (without readtable hacks) because it uses parens which are illegal in keywords and symbols, so we use an `@` sign instead.

Thus, we can disassemble to either a lispy syntax or the standard 6502 syntax in only a few dozen lines of code. Since we've factored out disassembling a single opcode, its easy to add a `current-instruction` helper to disassemble whatever the CPU is about to execute as well.

Source Code

```
(in-package :6502)

(defmacro with-disasm ((start end &key op) &body body)
  "Loop from START to END, passing each instruction to OP and execute BODY.
OP is PRINT-INSTRUCTION by default. Within BODY, the return value of OP is
bound to RESULT and the length of the instruction in bytes is bound to STEP."
  `(loop with index = ,start while (<= index ,end)
      for (step result) = (disasm-ins index ,@(when op (list op)))
      do (incf index step) ,@body))

(defun disasm (start end)
  "Disassemble memory from START to END."
  (with-disasm (start end)))

(defun disasm-to-list (start end)
  "Disassemble a given region of memory into a sexp-based format."
  (with-disasm (start end :op #'sexpify-instruction) collect result))

(defun disasm-to-str (start end)
  "Call DISASM with the provided args and return its output as a string."
  (with-output-to-string (*standard-output*) (disasm start end)))
```

```
(defun disasm-ins (index &optional (disasm-op #'print-instruction))
    "Lookup the metadata for the instruction at INDEX and pass it to
DISASM-OP for formatting and display, returning the instruction length."
  (destructuring-bind (name docs cycles bytes mode)
      (aref *opcode-meta* (get-byte index))
    (declare (ignore cycles))
    (let ((code-block (coerce (get-range index (+ index bytes)) 'list)))
      (list bytes (funcall disasm-op code-block index name docs mode)))))

(defun print-instruction (bytes index name docs mode)
  "Format the instruction at INDEX and its operands for display."
  (let ((byte-str (format nil "~{~2,'0x ~}" bytes))
        (args-str (format nil "~A ~A" name (arg-formatter (rest bytes) mode))))
    (format t "$~4,'0x   ~9A  ;; ~14A ~A~%" index byte-str args-str docs)))

(defun sexpify-instruction (bytes index name docs mode)
  "Given BYTES and metadata, return a sexp-format representation of it."
  (declare (ignore index docs))
  (alexandria:if-let ((args (rest bytes))
                      (args-str (bytes-to-keyword-syntax bytes mode)))
    (mapcar #'make-keyword (list name args-str))
    (mapcar #'make-keyword (list name))))

(defun arg-formatter (arg mode)
  "Given an instruction's ARG, format it for display using the MODE's WRITER."
  (if (member mode '(absolute absolute-x absolute-y indirect))
      (format nil (writer mode) (reverse arg))
      (format nil (writer mode) arg)))

(defun bytes-to-keyword-syntax (bytes mode)
  "Take BYTES and a MODE and return our assembly representation of the arguments."
  (let ((result (arg-formatter (rest bytes) mode)))
    (flet ((munge-indirect (str)
             (cl-ppcre:regex-replace "\\(\\$(.*)\\)(.*)?" str "@\\1\\2")))
      (cl-ppcre:regex-replace ", " (munge-indirect result) "."))))

(defun current-instruction (cpu &optional print-p)
  "Return a list representing the current instruction. If PRINT-P is non-nil,
print the current address and instruction and return NIL."
  (let ((fn (if print-p #'print-instruction #'sexpify-instruction)))
    (second (disasm-ins (cpu-pc cpu) fn))))
```

9 | Parsing Assembly

parser.lisp: String processing

References:

ca65 syntax

The parser converts source code in string format to intermediate instructions. The syntax is simple, following the standard 6502 style, except without any directives. A single instruction can contain any of the following: a label, an opcode, an operand, and a comment.

Operands have different syntax depending upon the addressing modes as defined in addressing.lisp. The actual value from the operand is still handled by the parser, and can be represented using decimal, or hexadecimal (if preceded by “$” a dollar sign), or binary (if preceded by “%” a percent sign), or can be a label, or a simple expression consisting of some other term plus (“+”) another expression.

The parser returns a list of instructions, with each slot set to the appropriate fields from the assembly statement. There may be more than one address-mode in such instructions, since the syntax is ambiguous.

Source Code

```
(in-package :6502)

(defun make-stream (text)
  ”Make a string displaced onto the given text.”
  (make-array (length text) :element-type ’character :adjustable t
              :displaced-to text :displaced-index-offset 0))

(defun try-fetch (stream regex)
  ”If the stream begins with a regex match, returns the matched text and move
   the stream past it. Otherwise, returns nil.”
  (let ((result (cl-ppcre:scan-to-strings regex stream)))
    (when result
      (multiple-value-bind (original index-offset) (array-displacement stream)
        (adjust-array stream (- (length stream) (length result))
                      :displaced-to original
                      :displaced-index-offset (+ index-offset (length result))))
      result)))

(defun substream-advance (stream start end)
  ”Set the stream to a substream at START positions ahead and finishing
```

```
   at END position."
  (multiple-value-bind (original index-offset) (array-displacement stream)
    (unless original
      (error "substream-advance called with a string, not a stream"))
    (adjust-array stream (- end start) :displaced-to original
                  :displaced-index-offset (+ index-offset start))))

(defun skip-white-space (stream)
  "Fetches white space from the stream, ignores it and returns the stream."
  (try-fetch stream "^\\s+")
  stream)

(defun fetch-label (stream)
  "Fetches a label from the stream, or returns nil."
  (let ((result (try-fetch stream "^[a-zA-Z][a-zA-Z0-9_]*:")))
    (when result (string-right-trim ":" result))))

(defun fetch-opcode (stream)
  "Fetches an opcode from the stream as a keyword, or returns nil."
  (let ((result (try-fetch stream "^[a-zA-Z]{3}")))
    (when result (intern (string-upcase result) :keyword))))

(defun fetch-literal (stream)
  "Fetches a literal value from the stream and returns it as an alist
   containing the integer value and address mode, or returns nil."
  (let ((result (try-fetch stream "^((\\$|\\&)[a-fA-F0-9]+|%[0-1]+|[0-9]+)")))
    (cond
      ((not result) nil)
      ((find (aref result 0) "$&") (parse-integer (subseq result 1) :radix 16))
      ((char= (aref result 0) #\%) (parse-integer (subseq result 1) :radix 2))
      (t (parse-integer result :radix 10)))))

(defun fetch-name (stream)
  "Fetches a name from the stream, or returns nil."
  (try-fetch stream "^[a-zA-Z][a-zA-Z0-9_]*"))

(defun fetch-term (stream)
  "Fetches a literal or name from the stream, or returns nil."
  (or (fetch-literal stream) (fetch-name stream)))

(defun match-operand-address-modes (stream)
  "Matches the stream against all address-mode readers, returning those that
   match, as well as the positions where the match occurs."
  (let ((value-match (list nil nil)))
    (list (loop for address-mode being the hash-keys of *address-modes*
             when (multiple-value-bind (start end match-start match-end)
                      (cl-ppcre:scan (reader address-mode) stream)
                    (when start
                      (setf value-match (list match-start match-end))))
             collect address-mode) value-match)))

(defun operand-possible-modes-and-value (stream)
  "Returns all matching address-modes for the operand, along with positions
   where the match occurs."
  (destructuring-bind (address-modes (match-starts match-ends))
      (match-operand-address-modes stream)
    (cond
      ((find 'implied address-modes) (list (list 'implied) 0 0))
      ((find 'accumulator address-modes) (list (list 'accumulator) 0 1))
      (t (list address-modes (aref match-starts 0) (aref match-ends 0))))))
```

```
(defun fetch-expression (stream)
  ”Fetches an expression from the stream, either a term or a term plus another.”
  (let ((term-1 (fetch-term stream)))
    (if (try-fetch (skip-white-space stream) ”^\\+”)
        (list ’+ term-1 (fetch-expression stream))
        term-1)))

(defun fetch-operand (stream)
  ”Fetches the operand, returning its numerical value and possible
   address-modes.”
  (destructuring-bind (possible-modes value-start value-end)
      (operand-possible-modes-and-value stream)
    (substream-advance stream value-start value-end)
    (list (fetch-expression stream) possible-modes)))

(defun parse-line (text)
  ”Converts a line of text into an instruction representing the assembly code.”
  (let* ((stream (make-stream text))
         (label (fetch-label (skip-white-space stream)))
         (opcode (fetch-opcode (skip-white-space stream)))
         (operand (fetch-operand (skip-white-space stream)))
         (value (first operand))
         (address-modes (second operand)))
    (make-instruction :label label :opcode opcode :value value
                      :address-mode address-modes)))

(defun strip-comment (text)
  ”Removes comment and white space from end of string.”
  (let ((pos (position #\; text)))
    (when pos (setf text (subseq text 0 pos))))
  (string-right-trim ” ” text))

(defun parse-code (text)
  ”Parses the assembly source text and returns the assembled code as a list of
   alists.”
  (loop for line in (cl-ppcre:split ”\\n” text)
        when (parse-line (strip-comment line)) collect it))
```

10 | Creating New Code

`assemble.lisp`: This is only the Beginning

References:

- A port of Henry Baker's comfy-6502
- Fun with Lisp: Programming the NES

Since our disassembler can disassemble to either a string or sexp format, we'd like our assembler to be similarly versatile. Therefore, we'll define a Generic Function `asm` that works on lists or strings. In addition it takes an optional environment as a hash table, and a starting address.

The assembler works by looping over its input, recording any labels, and assembling each statement one at a time, using delayed functions for any statements that use a label. Then finally, delayed functions are resolved, and all the results are concatenated together. Both lists and strings are converted into an intermediary format, a struct called instruction, which represents a single statement.

The parser determines which possible address modes can match the given syntax, which may be ambiguous due to labels, and matches these modes against what the given opcode can use.

Source Code

```
(in-package :6502)

(defconstant +relative-branch-size-byte+ 2)
(defconstant +max-byte+ 256)
(defparameter +absolute-modes+ '(absolute absolute-x absolute-y))
(defparameter +zero-page-modes+ '(zero-page zero-page-x zero-page-y))

(defgeneric asm (source &optional init-env org-start)
  (:documentation "Assemble SOURCE into a bytevector and return it."))

(defmethod asm ((source list) &optional init-env org-start)
  (assemble-code-block (list-to-instructions source) init-env org-start))

(defmethod asm ((source string) &optional init-env org-start)
  (assemble-code-block (parse-code source) init-env org-start))

(defstruct instruction
  "Represents a single line of code."
  (label       nil :type (or null string))
  (opcode      nil :type (or null symbol))
  (address-mode nil :type (or null symbol list))
  (value        nil :type (or null u16 list string)))

(defun list-to-instructions (instructions)
  "Given a list of assembly tuples, convert them to instructions."
  (unless (listp (first instructions))
    (setf instructions (list instructions)))
  (loop for tuple in instructions collect (apply #'tuple-to-instruction tuple)))

(defun tuple-to-instruction (opcode &optional operand)
  "Given an opcode and value, as symbols, convert them to an instruction."
  (unless operand
    (return-from tuple-to-instruction
      (make-instruction :opcode opcode :address-mode 'implied)))
  (let ((token (transform-sexp-syntax operand)))
    (destructuring-bind (possible-modes value-start value-end)
        (operand-possible-modes-and-value token)
      (let ((stream (make-stream (coerce (subseq token value-start value-end)
                                         '(vector character)))))
        (make-instruction :opcode opcode :value (fetch-literal stream)
                          :address-mode possible-modes)))))

(defun transform-sexp-syntax (sexp-token)
  "Given a SEXP-token using an indirect, *.x or *.y addressing mode, transform
   it to use the classic string assembly syntax."
  (substitute #\, #\. (cl-ppcre:regex-replace "\@([^.]*)(.*)?"
                                              (string sexp-token) "($\\1)\\2")))

(defmacro resolve-byte (place env)
  "Given a place and an environment, if that place is a function, call that
   function with the environment and assign the result to the place."
  (let ((byte-name (gensym)))
    `(when (functionp ,place)
       (let ((,byte-name (funcall ,place ,env)))
         (setf ,place ,byte-name)))))

(defun assemble-code-block (code-block &optional init-env org-start)
  "Given a list of instructions, assemble each to a byte vector."
  (let ((env (or init-env (make-hash-table :test 'equal)))
        (output (make-array 0 :fill-pointer 0 :adjustable t))
        (pc-start (or org-start 0)))
    ; Build byte vector, without labels.
    (loop for instruction in code-block
       do (let ((bytes (assemble-instruction instruction
                                             (+ pc-start (length output)) env)))
            (loop for b in bytes
               do (vector-push-extend b output))))
    ; Resolve labels in the byte vector.
    (loop for i from 0 below (length output)
       do (resolve-byte (aref output i) env))
    output))

(defun assemble-instruction (instruction pc env)
  "Given an instruction, and the current program counter, fill the environment
   with any labels and assemble instruction to a list of bytes."
  (with-slots (opcode value label) instruction
    (when label
      (setf (gethash label env) pc))
    (when opcode
      (let ((mode (decide-address-mode instruction env)))
        (list* (find-opcode opcode mode) (process-args value mode pc))))))
```

```
(defun find-opcode (opcode mode)
  "Finds an opcode matching OPCODE and MODE, raising ILLEGAL-OPCODE otherwise."
  (let ((match (position-if #'(lambda (e) (match-opcode-data e opcode mode))
                            *opcode-meta*)))
    (or match (error 'illegal-opcode :opcode (list opcode mode)))))

(defun process-args (value address-mode pc)
  "Given the operand value, address-mode, and program counter, return a list of
   assembled bytes, using delayed functions for labels or expressions."
  (case address-mode
    ((absolute absolute-x absolute-y indirect)
     (list (make-byte value pc :low) (make-byte value pc :high)))
    ((implied accumulator) nil)
    (relative (list (make-byte value pc :relative)))
    (otherwise (list (make-byte value pc :low)))))

(defun decide-address-mode (instruction env)
  "Finds the desired address mode, matching what the opcode allows to what was
   parsed from the operand's syntax."
  (with-slots (opcode address-mode value) instruction
    (let ((modes (if (listp address-mode) address-mode (list address-mode)))
          (opcode-modes (get-opcode-address-modes opcode)))
      (if (common-lisp:and (zero-page-address value env)
               (intersection opcode-modes +zero-page-modes+))
          (setf modes (set-difference modes +absolute-modes+))
          (setf modes (set-difference modes +zero-page-modes+)))
      (first (intersection modes opcode-modes)))))

(defun get-opcode-address-modes (opcode)
  "Given an opcode, return the possible address modes for that operation."
  (loop for e across *opcode-meta*
     when (match-opcode-data e opcode :any) collect (fifth e)))

(defun match-opcode-data (data opcode &optional (address-mode :any))
  "Returns whether the asm metadata matches the given opcode, and address-mode
   if it is provided."
  (common-lisp:and (eq (first data) (intern (symbol-name opcode) :6502))
                   (or (eq address-mode :any) (eq address-mode (fifth data)))))

(defun zero-page-address (addr env)
  "Returns whether the address is a zero page access."
  (cond
    ((numberp addr) (< addr +max-byte+))
    ((stringp addr)
     (let ((addr (gethash addr env)))
       (common-lisp:and (numberp addr) (< addr +max-byte+))))
    ((listp addr) nil)
    (t (error "Invalid address" :argument addr))))

(defun make-byte (value pc type)
  "Given an integer, return a single byte for the required type. Given a label,
   return a delayed function to calculate the same, once labels are defined."
  (cond
    ((stringp value)
     (lambda (env)
       (let ((addr (or (gethash value env)
                       (error "Undefined label" :argument value))))
         (when (eq type :relative)
           (setf addr (- addr pc +relative-branch-size-byte+)))
         (make-byte addr pc type))))
    ((common-lisp:and (listp value) (eq (first value) ’+))
     (lambda (env)
       (destructuring-bind (unused-plus operand-1 operand-2) value
         (declare (ignore unused-plus))
         (+ (make-and-resolve-byte operand-1 pc type env)
            (make-and-resolve-byte operand-2 pc type env)))))
    ((numberp value)
     (if (eq type :high) (floor (/ value +max-byte+)) (mod value +max-byte+)))
    (t (error ”Cannot make-byte” :argument value))))

(defun make-and-resolve-byte (operand pc type env)
  ”Given an operand, convert it to a byte, resolving any delayed functions.”
  (let ((value (make-byte operand pc type)))
    (resolve-byte value env)
    value))
```

11 | Wrap it with a Bow

`packages.lisp`: Exposed Functionality

Finally, we'll want to define packages for our emulator to provide a public API. Who knows, maybe somebody out there is just dying for a drag and drop 6502 emulator in Lisp with a function to single-step instructions. :)

The `6502` package is for use by emulator writers, the test suite, etc. It exposes all the types and interesting high-level functionality. It also shadows Common Lisp's built-in `and` and `bit` symbols since they name 6502 opcodes.

The `cl-6502` package is more limited and designed for public consumption, hiding the Addressing Modes and their protocol, the CPU slot accessors, and other helpers.

Source Code

```
(defpackage :6502
  (:use :cl)
  (:import-from :alexandria #:compose #:emptyp #:flatten #:make-keyword #:named-lambda)
  (:shadow #:bit #:and)
  (:export ;; Public API
           #:execute #:step-cpu #:asm #:disasm #:disasm-to-str #:disasm-to-list
           #:current-instruction #:get-byte #:get-word #:get-range #:*cpu* #:cpu
           #:nmi #:reset #:jit-step #:*opcode-meta*
           ;; CPU struct
           #:make-cpu #:cpu-ar #:cpu-xr #:cpu-yr #:cpu-sr #:cpu-sp #:cpu-pc #:cpu-cc #:u8 #:u16
           ;; Addr modes
           #:implied #:accumulator #:immediate #:zero-page #:zero-page-x #:zero-page-y
           #:absolute #:absolute-x #:absolute-y #:indirect #:indirect-x #:indirect-y #:relative
           #:reader #:writer
           ;; Helpers
           #:wrap-byte #:wrap-word #:status-bit #:defenum #:bytevector))

(defpackage :cl-6502
  (:documentation "Homepage: <a href=\"http://github.com/redline6561/cl-6502\">Github</a>")
  (:use :cl)
  (:import-from :6502 #:execute #:step-cpu #:asm #:disasm #:disasm-to-str #:disasm-to-list
                      #:current-instruction #:get-byte #:get-word #:get-range #:*cpu* #:cpu
                      #:nmi #:reset #:jit-step)
  (:export #:execute #:step-cpu #:asm #:disasm #:disasm-to-str #:disasm-to-list
           #:current-instruction #:get-byte #:get-word #:get-range #:*cpu* #:cpu
           #:nmi #:reset #:jit-step))
```

12 | Lessons Learned - Emulation

High-level emulation is viable

High-level emulation *can* work and the result is great. On the other hand, it entails *extra* work mapping the low-level hardware to your high-level language back to reasonably fast code. This is a time-consuming process.

What must be fast

3 things absolutely must be fast: opcode dispatch, memory reading/writing, and status bit handling. CPUs only do a few things and most of them your language already knows how to make fast; arithmetic and assignment, in particular. Practically every CPU instruction deals with memory somehow, updates the status register, or both. Think carefully about how to map these operations onto your language. They should happen a few million times a second. :)

Profile with intuition

Your language profiler can't show you architectural issues, only hotspots. I used SBCL's statistical profiler a good bit on this project and regard it highly. It tracks both CPU usage and memory allocation/consing very well. However, most of the really big gains came from some mixture of intuition, experimentation, and good old hard thinking.

Get the API right first

Consequently, getting the API right is the hard part and the most important. If you've defined your interfaces correctly, you can rewrite the underlying data representations or execution strategies to achieve good performance.

13 | Lessons Learned - Common Lisp

Structures can be preferable to classes

Structures are much more static than classes. They also enforce their slot types. When you have a solid idea of the layout of your data and really need speed, they're ideal.

CLOS is fast enough

CLOS, for single-dispatch at least, is really quite fast. When I redesigned the emulator to avoid a method call for every memory read/write, my benchmark only ran ~10% faster. I eventually chose to stick with the new scheme for several reasons, performance was only a minor factor.

Destructuring is more expensive than you think

My second big speedup came, indirectly, from changing the arguments to the opcode lambdas. By having the opcode only take a single argument, the CPU, I avoided the need to destructure the opcode metadata in `step-cpu`. You **don't** want to destructure a list in your inner loop, no matter how readable it is!

Eval-when is about data more than code

That is, the times I found myself using it always involved computing data at compile-time that would be stored or accessed at load-time or later. E.g. I used it to ensure that the status-bit enum was created for use by later macros like `set-flags-if`. Regardless, try to go without it if possible.

Use DECLAIM (and DECLARE) wisely

DECLAIM is for global declarations and *DECLARE* is for local ones. Once you've eked out as many algorithmic gains as possible and figured out your hotspots with the profiler, recompile your code with `(declaim (optimize speed))` to see what notes the compiler gives you. Letting the compiler know the *FTYPE* of your most called functions and inlining a few things can make a big difference.

14 | Conclusion

Next Steps

Once you've got a CPU Emulator there's a plethora of options for further exploration:

- You could write a whole-system emulator using that CPU.
- You could write a simple compiler targeting that CPU.
- You could write a static analyzer to compute the CFG of binaries for that CPU.
- You could port it to the web to bring your code to a wider audience.

Whatever you do, I hope you enjoyed reading through **cl-6502**. Happy Hacking!