At the beginning of this semester, I set out to work on a senior project that would be accessible to any student who had taken CPSC 201, our introductory course on computer science. Having worked as an Undergraduate Learning Assistant for the department, I had anecdotal evidence that many students that took 201 hadn’t been especially fond of the TC-201 problem set, in which they were asked to build a virtual machine architecture with an assembler, and then write programs for it. I was saddened to hear this, as when I had taken 201 my first year at Yale the TC-201 was my favorite p-set, although it was still quite difficult for me at the time. After countless hours debugging, however, what I was left with was a computer, that I had made. My computer was little more than a binary calculator, but it would execute any instruction I could think of how to implement, and building it had taught me quite a lot about how other computers, such as the one I built it on, were designed. Given a semester, I thought, maybe I could turn this into something worth interacting with.

What I came up with was the MC-490, an expanded architecture with additional instructions, a display monitor, and keyboard input. Modeled after various microcontrollers like the Arduino, my computer was designed to be a bare-bones system with a minimalist assembler, owing its true potential to the programs it could execute. To demonstrate this capability, I wanted to write an easily recognizable program using the MC-490’s assembly language, and settled on attempting to reproduce the classic arcade game Space Invaders. If I was going to create a program with hundreds of variables and thousands of possible states, however, I was going to need to either learn to write extremely efficient code or add some enhancements to my

---

1 albeit, a tiny one
virtual machine. I decided the latter seemed easier and added a small bank of registers to reduce RAM accesses and a dedicated stack, in addition to separating the program from the RAM and encoding it as a byte string.

I implemented my registers using a similar approach to my RAM as a vector of vectors. Initially I settled on 16 one-byte registers, but wanted the flexibility of handling two-byte numbers so I added functionality to allow for registers to be indexed and set as pairs. Drawing from the Atmel AVR (the chip on the Arduino Uno)’s index registers, I named every other register a letter starting from the back of the alphabet, leaving space for a one-byte status register that never came to be and a three-byte accumulator, which was similarly underused. In practice, I ended up using the computer a bit more like a 6-register machine, using only the named registers as they were the ones I could remember my own calling convention for.

Rather than place my stack at the end of my RAM growing downwards, I chose to have a dedicated stack that took advantage of Racket pair/list properties to allow constant-time accesses to the top, as well as avoid the possibility of the stack overflowing the memory. In a similar vein of protection, I chose to load programs separately into the computer as ROM rather than allow for the possibility of self-modifying code. This also presented an opportunity for me to optimize my ROM implementation, and I decided to store my assembled program as a Racket byte string due to their compact size and automatic numerical conversion. I also believed that I would get a performance boost out of limiting the number of accesses to the ROM, but it turned out that I overestimated both the eventual size of my program and its memory usage, as both would have easily fit in memory. After assembling, my program was only 738 characters, or roughly two and a half 2018 tweets, long.

To draw to the screen and read from the keyboard, I started off the semester torn between two different Racket graphics libraries, the How To Design Programs image library (htdp) and the Racket Graphical Interface Toolkit (gui). The htdp library was designed for CS students and features easy to understand procedures for windowing, drawing, and keyboard handling. The gui library, on the other hand, seems to be the standard Racket GUI library, that it’s possible even DrRacket uses. Since I knew the basic abilities that whichever library I ended up using needed to have, I decided to write a small series of tests for both libraries in order to determine which, if

\[X, Y,\text{ and } Z\text{ are caller-save and } U, V,\text{ and } W\text{ are callee-save}\]
any, would enable me to finish my project. As a bonus, testing both languages would force me to learn the basics of them both for when it was time to add graphics to my instruction set.

At its core, an MC-490 program should be able to read user input from a keyboard, and display some sort of feedback to the screen. For my test, I decided to try and reproduce in the gui library a program I adapted from a post in Stack Overflow, which uses the htdp library to draw a ball to the screen that is controlled by the arrow keys (see Appendix Figure 1). Unlike the htdp example, which uses the built-in big-bang procedure and simple functions that consume a world state and return a modified one, the gui library required a little bit more tooling to draw a window and update it (see Appendix Figure 2). In both examples, however, multiple key presses are not handled correctly, and since I wanted to provide the ability to move and shoot at the same time in my Space Invaders implementation - and for key combinations more generally - I next challenged myself to modify my programs to allow for multiple key presses. After struggling for a while with many different failed approaches using the htdp library, I found a thread on the Racket community page with the code I was looking for, although even the author wrote that it was rather complex and potentially difficult to understand. What’s more, the code only really allowed for one or two keys to be pressed at a time, rather than monitoring the state of all keys and responding depending on which were pressed in conjunction. At the suggestion of my advisor, I tried using a hash table with the gui library to store the state of each key being tracked, then handling keyboard input based off of the state of the virtual keyboard in the hash table, rather than worrying about receiving key presses one at a time. Unlike the htdp implementation, my code (see Appendix Figure 3) would easily scale to any number of keys being monitored at once without introducing hundreds of new possible states.

While I initially was leaning towards the htdp library for its short and easy-to-understand procedures and target demographic, I found that for more complicated functionality, such as monitoring multiple keys at once, the gui library outshined it. In addition, the htdp world state worried me a little bit when I thought about timing and communication between the keyboard and the computer; it seemed as though the library would have me design the entire structure around the htdp big-bang loop or attempt to asynchronously pass messages between the

---

3 https://stackoverflow.com/questions/14540119/on-key-in-racket

graphics/keyboard input portion of my code and the main computer implementation. For these reasons, I settled on the gui library and began reading the documentation\(^5\) online to figure out how to use it.

Creating a screen that could be drawn to ended up requiring little more than defining a frame to hold a canvas, and then defining the canvas to be drawn to. To handle keyboard input, I subclassed the canvas to create kb-canvas\(^\circ\), a canvas class with methods for drawing arrays of bytes to the screen, as well as the hash table of keys and their states. Working with the gui library had the unintended consequence of forcing me to learn how Racket classes behave, which ended up being quite helpful as I quickly outgrew the struct approach of representing my machine and had to define a computer class to hold the CPU and memory portions of the MC-490.

With my virtual hardware in working order, it was time to think about my instruction set architecture. One of the biggest points of confusion for me while working on the TC-201 pset was the trailing zeros in no-argument instructions like output, halt, or skippos. Additionally, I knew I wanted the ability to specify at least two arguments in an instruction so that I was not stuck relying on the accumulator like in the TC-201. Knowing that I wanted to provide a variable number of arguments at assemble-time, I decided that the assembler would take however many arguments it was given and fit them into a set number of bits following the assembled opcode; thus, instructions could take a small number of large arguments or a large number of small ones. There were a few snags to this approach, but none of them were major. First, how would my assembler know what was a label and what was an opcode? I could check whether the symbol was in my instruction set, but many opcode names were general enough that I didn’t want to prevent labels from also holding their names. I settled on using keywords for labels, mostly due to their color scheme being different in DrRacket, but also for their easy conversion to symbols. Finally, I wanted the ability to provide a memory address and an additional argument, but I also wanted to use 12 bits for memory addressing. Here I used a bit of a hack, realizing that if I were to convert that 12-bit number into three 4-bit numbers, my assembler would happily store it and an additional 4-bit number in the two bytes I had reserved for arguments. Thus, I could store opcodes in one byte and, besides data statements, my architecture would have a fixed instruction width of three bytes.

\(^5\) https://docs.racket-lang.org/gui/
Data statements proved similarly easy, since I was only ever using them for representing sprites in Space Invaders. I wanted to store sprites as arrays in memory, with a header byte storing the frame width and the number of frames, followed by the frames drawn out in bits. Since I’m not particularly artistic, I wanted to make the process of animating my game as simple as possible, and opted to represent my bytes of data as strings of bits, allowing me to eyeball my sprites before running anything. These bit strings are thus assembled into individual bytes, and subsequently into a byte string. This is disadvantageous for representing actual numbers, but since MC-490 programs are read-only, data statements are pretty much reserved for bit arrays anyways.

I opted to do conditional branching in a very similar way to the TC-201 using skip instructions, however I fit all skip instructions under one opcode, then defined a sub-instruction set for skip with opcodes for comparing the contents of registers, as well as a poll opcode. Skip poll takes a key code (stored as two 5-bit numbers), and skips if the key is pressed, proceeding to the next instruction if it is not. In a similar fashion, instructions concerning the display are all held under the window opcode, with four bits reserved for sub-opcodes; show, hide, and clear taking no additional arguments, and both draw functions using the remaining 12 bits for a label. Since draw only takes the sprite to be drawn as an argument, I needed a way to indicate the coordinates to draw the sprite at and which frame of the sprite to draw. My unattractive solution to this problem was to store all three in the 24-bit accumulator; ten bits for the x value, ten for the y value, and four for the frame number. I wrote an instruction (xyl) that would take three registers and place the contents of the first two and the least-significant half of the third into the accumulator, separated by two zeros each, effectively multiplying the x and y coordinates by four, the width of a pixel.

As mentioned before, I also wanted to provide my registers and memory to be indexed and set as pairs, providing 16-bit functionality in my machine. To utilize this feature, I planned for many instructions to have a corresponding pair instruction (load2, add2, etc.), although as also mentioned before, this feature ended up being less necessary than I initially imagined. Many pair

---

6 five bits for frame width and three for number of frames, meaning max values of 32 and 8, respectively.

7 instruction sub-set?

8 more on this later
instructions, like loadc2 and pop2, did not even get implemented in the computer’s decode procedure. Two instructions that proved to be a great deal more useful, loadio and storeio, function similar to their cousins loadi and storei, but get their memory address from a register rather than RAM, and use the new surplus bits to store an offset. With these instructions I was able to quickly write arrays and structs in memory, and cut RAM accesses twofold while doing so.

After beefing up the instruction set, it was time to turn my attention to writing the actual program to run on the MC-490. As foreshadowed earlier, I used to only have one draw instruction, which would jump to a label in the program, read the header byte for information about the sprite, and then draw the desired frame bit-by-bit rotated 90 degrees clockwise, drawing a white pixel for a one and a black pixel for a zero. When I began drawing aliens to the screen, I immediately noticed how long it took for me just to draw the 8x5 “rack” of aliens, almost a full second per row if I was also updating the player’s ship and animating bullets. Initially I was using the original Space Invaders method of storing the coordinates of a “reference alien,” and calculating the position of the other 39 aliens off of the reference alien. Thinking that perhaps the lag I was experiencing was due to the process of recalculating these positions every time, I used the aforementioned loadio and storeio (as well as storei in the initialization) to maintain an array of “structs” representing each alien’s status, position, and frame, respectively. I would then update each alien’s position individually, after they were already drawn to the screen, before clearing and redrawing them all together. I figured short of implementing an extremely use-specific instruction to draw grids of sprites, with a mechanism for “turning off” individual sprites, a straightforward loop to draw the aliens was about as optimal as I could get. Still, when I ran the program the aliens would make their way onto the screen at a snail’s pace.

Suspecting thus that the bottleneck was in my actual draw implementation, I pored over my kb-canvas% methods, before realizing that each function call to draw a bit to the screen was first setting the pen and brush according to the bit. Realizing I could save a lot of calls to set-pen and set-brush by keeping them both white and only drawing bits that were ones, I edited my draw function to only draw pixels when the corresponding bit was one, and re-ran my program to discover my speed had increased considerably. Unfortunately, this modification introduced a new problem: faster-moving game objects that I had relied on to overwrite themselves with black

---

9 http://www.computerarcheology.com/Arcade/SpaceInvaders/Code.html
pixels - namely the ship and the bullets - were now leaving trails across the screen until window clear was called at the top of the game loop. I could redraw the ship less frequently, but without overwriting on zero bits, it would either be shooting slow-motion bullets or inchworm laser beams. Deciding neither option was particularly attractive, I reimplemented the original draw instruction, this time named draw-bw, alongside the new and much faster draw. Using the two draw instructions, I was finally able to implement a bare-bones version of Space Invaders, with animated aliens that move in the classic zig-zag pattern towards the bottom of the screen, and trigger a game over if they reach the bottom.

Although my visions were a bit loftier when I began working on this project, now at the end of the semester I find that I have accomplished all of my stated objectives in some form or another. The MC-490 is more architecturally complex than the TC-201, and boasts a considerably larger instruction set. In addition, I was able to write a full program\textsuperscript{10} in MC-490 assembly language. Working at a significantly lower level than I am used to forced me to really think about efficiency, both time and space-wise. Since early in my CS career it has been a dream of mine to “learn to program in assembly language,” so designing, implementing, and programming with an instruction set of my own specification was both a learning opportunity, and an opportunity to check something off my bucket list.

It’s less easy for me to assess whether my project is a success or not in terms of accessibility. While I was hoping to have a substantial portion of the game ready for demo by the time students in 201 were working on the TC-201 p-set, debugging the laggy draw instructions took a considerable amount of time and I was unable to receive feedback from current introductory CS students. Still, I am consistent with my goals in terms of writing the entirety of my code in a language and format that can be understood by someone who has taken CPSC 201, as well as using minimal external libraries and favoring popular 201 programming techniques like car/cdr recursion and auxiliary functions to opaque “clever” approaches.\textsuperscript{11} It is my hope that any 201 student who wished to deepen their understanding of the TC-201 p-set would be able to see how I built the MC-490, and how I implemented Space Invaders to run on it.

\footnotesize
\begin{itemize}
\item \textsuperscript{10} almost
\item \textsuperscript{11} with a few notable exceptions
\end{itemize}
Although I am at the finish line, I find that there are a number of additional tweaks and changes I would make given more time. Some of these are, in no particular order:

• Improve hit-checking in Space Invaders, after devising a better method to test it

• Add shields above the players ship that take damage using a new instruction erase bytes, which sets the pen and brush to black before calling draw and then setting them back

• Add more structure to the game (a startup/pause menu, multiple levels, etc.)

• Fix the check of whether the aliens have reached the bottom in inc-pos (right now it’s only checking the position of the bottom row, not whether any aliens in the row are still alive)

• Refactor run flag into the status register and add side-effects in instructions that set the bits of the status register and the accumulator

• Allow data statements to hold strings of text, instead of just ones and zeros, and assemble each character as a bit or a byte depending on whether the string is pure binary or not

• Add a print instruction and write Hello World

• Allow user input in the REPL without having to show the display, possibly a modified version of do-input from the TC-201

Even without these features, however, I am quite pleased with my end product, and it’s doubtful most of them would have any measurable impact on the final machine anyways. In retrospect, I actually ended up underusing a lot of my own bells and whistles, finding that I could accomplish the same effects using simpler methods. The beauty of assembly language programming, I learned from working on this project for the past semester, is that a handful of basic commands, when combined in the right way, can create endlessly powerful applications.
Appendix

#lang racket

(require 2htdp/image)

(define (ball-image t) ;<-- the t-parameter is our WorldState
  (place-image (circle 5 "solid" "red")
               (car t) ;<-- here now x variable coordinate
               (cdr t) ;<-- here now y variable, instead of 150
               (empty-scene 300 300)))

(define (change w a-key)
  (cond ;w - is the previous worldState, V here we change it
         [(key=? a-key "left")  (cons (- (car w) 3) (cdr w))];and
         [(key=? a-key "right") (cons (+ (car w) 3) (cdr w))];return
         [(key=? a-key "up")    (cons (car w) (- (cdr w) 3))]
         [(key=? a-key "down")  (cons (car w) (+ (cdr w) 3))]
         [else w]) ;<-- If the key of no interest, just return the previous WorldState

(big-bang *(150 . 150) ;<-- initial state
to-draw ball-image) ;<-- redraws the world
(on-key change);<-- process the event of key press

Figure 1. Htdp Ball Moving Test

#lang racket

(require racket/gui/base)

(define frame (new frame% [label "Window"] [width 300] [height 300]))

(define kb-canvas% (class canvas% (init pos) ; pos is a pair of X and Y coordinates
                [super-new [parent frame]
                  [paint-callback (λ dc
                                    (send dc set-pen "red" 10 'solid)
                                    (send dc clear)
                                    (send dc draw-point (car (send c get-pos)) (cdr (send c get-pos)))]])
                [define/override (on-char event)
                                (change-pos (send event get-key-code))]
                [define/public (get-pos) current-pos]
                [define/public (change-pos code)
                              (let ([lr (or (equal? code 'left) (equal? code 'right))])
                               (if lr ((if (equal? code 'left) - +) (car current-pos) 3) (car current-pos))
                               (if ud ((if (equal? code 'up) - +) (cdr current-pos) 3) (cdr current-pos))))])

(define canvas (new kb-canvas% [pos (cons 150 150)]))

(define (loop)
  (send canvas on-paint)
  (sleep/yield .001) ; yield for keyboard handling
  (loop))

(send frame show #t)
(loop)

Figure 2. Gui Ball Moving Test
(require racket/gui/base)

(define frame (new frame% [label "Window"] [width 400] [height 300]))

(define KEYS (list 'left 'right 'up 'down 'space))

(define kb-canvas% (class canvas% (init pos)
  [define current-pos pos]
  [define keys (make-immutable-hash (map (λ (key) (cons key #f)) KEYS))]
  [super-new [parent frame]
    [paint-callback (λ (c dc)
      (send dc set-pen "red" 4 'solid)
      (send dc clear)
      (update-pos)
      (send dc draw-point (car (send c get-pos)) (cdr (send c get-pos))))])
  [define/override (on-char event)
    (cond
      [(member (send event get-key-code) KEYS) (set! keys (hash-set keys (send event get-key-code) #t))]
      [(and (equal? (send event get-key-code) 'release) (member (send event get-key-release-code) KEYS))
        (set! keys (hash-set keys (send event get-key-release-code) #f))]
    )]
  [define/public (update-pos)
    (let [(ud (xor (hash-ref keys 'up) (hash-ref keys 'down)))
           (lr (xor (hash-ref keys 'left) (hash-ref keys 'right)))]
      (when (or ud lr)
        (set! current-pos (cons (if lr ((if (hash-ref keys 'left) - +) (car current-pos) 2) (car current-pos)) (if ud ((if (hash-ref keys 'up) - +) (cdr current-pos) 2) (cdr current-pos))))))]
  [define/public (get-pos)
    current-pos])

(define canvas (new kb-canvas% [pos (cons 150 150)]))

(define (loop)
  (send canvas on-paint)
  (sleep/yield .01)
  (loop))

(send frame show #t)
(loop)

Figure 3. Gui Multi-Key Test