In CPSC 201 Intro to Computer Science, students complete a problem set in which they build a virtual machine architecture and assembler in Racket. This problem set, known as the TC-201 (Tiny Computer 201), is often regarded as one of the most difficult problem sets in the class as it synthesizes what students have learned about the architecture of computers and requires them to write programs in both Racket and their own version of assembly. As a senior in the Computer Science major, I have come to appreciate facing this problem set so early in my college career, as it helped prepare me for numerous other classes in which concepts touched upon by the TC-201 were either revisited or expanded upon. Thus, in my opinion, the TC-201 serves as an incredibly important foundational medium in the Computer Science department for learning and conceptualizing the incredibly complicated mechanisms which operate the modern computer. Working as an Undergraduate Learning Assistant last semester forced me to consider problem sets from a teacher’s perspective, and one of the things that I noticed about the TC-201 was that many students struggled to make the connection between assembly language and higher level languages they were more comfortable with, namely because they had difficulty writing programs in assembly because of the difficulty translating common techniques like loops and conditionals.

The TC-201 provided a fantastic foundation for learning about the inner workings of computers down to the bits, and for my senior project I would like to expand on the model of the TC-201 to create a more comprehensive virtual architecture with a larger instruction set. In addition, I will provide a program (or programs if time permits) that more fully demonstrate the capabilities of the instruction set and allow for more interaction than the assembly programs
written for the TC-201. A logical domain is video games, namely early arcade games such as Space Invaders or Pac-Man. My motivations for this are twofold: designing, implementing, and programming with an instruction set of my specification will force me to really acquaint myself with how a computer operates and the foundational work of early computer scientists; in addition, the implementation of such an architecture in Racket, structurally similar to the TC-201 means that my code will be accessible to anyone who has taken CPSC 201, Yale’s introductory Computer Science course, as I plan to use minimal external libraries and document the use of those that I do in comments with the assumption that the reader is unfamiliar with their function. In this way, the MC-490 serves as a development of the TC-201 concept, both for me and for any CPSC 201 students who may wish to further their learning after completing the TC-201.

Implementation

The biggest feature the TC-201 is lacking for the purposes of video game programming is a method for displaying graphics to the screen (not to mention a screen). For this and other system-level interactions such as keyboard input, I plan to use either the racket gui library\(^1\) or a library called 2htdp, which corresponds to the second edition of the online textbook How To Design Programs\(^2\). Similar to the TC-201, I plan to divide my computer configurations into CPU and RAM structs allowing either or both to be changed independently. Unlike the TC-201, I plan on adding a number of registers to the CPU to reduce the number of memory accesses and dividing memory into RAM and ROM to protect the program from accidental writes. In addition to increasing the number of available RAM addresses, I will most likely need to increase the bit-width of the MC-490 to reflect the larger number of opcodes and potentially larger addresses. Potential additions to the instruction set beyond the addition of graphics and keyboard-related

\(^1\) https://docs.racket-lang.org/gui/index.html

\(^2\) http://www.htdp.org/2018-01-06/Book/index.html
operations include: push/pop instructions for native stack support, call/ret instructions for reusability of code sections (pseudo-functions), and branch instructions for easier conditionals.

For the assembly program, I plan to try and implement Space Invaders as faithful to the original arcade game as possible. Since color is of minimal importance in the game, this would allow me to draw and animate sprites in memory as extended data statements with bits corresponding to individual pixels, rather than worrying about representing both position and color. Certain regions of the game are colored differently, namely the shield and player ship region are green and the UFO is red, but as the coloring is region-specific, this can be handled outside of the program or even inside using a special opcode. Aiding me in both the design of my program and the implementation of necessary functions is an extensive article I found on the game\(^3\), which includes a fully disassembled source code as Z80 opcodes. In addition, there exists an implementation of a Space Invaders clone in Racket on GitHub\(^4\), which uses the 2htdp library. Both provide an excellent starting point for the design of my own Space Invaders clone, especially the original game source, which elucidated both the monitor’s 90-degree rotation and the region-specific coloring that were used to fit sprites into 8-bit instructions and introduce minor coloring to the game rather simply.

\(^3\) [http://www.computerarcheology.com/Arcade/SpaceInvaders/](http://www.computerarcheology.com/Arcade/SpaceInvaders/)

\(^4\) [https://github.com/jeremylinlin/racket-space-invaders](https://github.com/jeremylinlin/racket-space-invaders)