An Introductory Low-level Programming Course for Students with a Python Background

by

Grace Quaratiello

S.B. Electrical Engineering and Computer Science and Business Analytics, Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by: Grace Quaratiello
Department of Electrical Engineering and Computer Science
May 12, 2023

Certified by: Joseph D. Steinmeyer
Senior Lecturer
Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

An Introductory Low-level Programming Course for Students with a Python Background

by

Grace Quaratiello

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The study of C and assembly language can provide valuable insight about the innate nature of computing systems and higher level programming languages. However, before September 2022, the MIT Department of Electrical Engineering and Computer Science (MIT EECS) had not required students to take any class that covers this material and these relationships. The classes included in the introductory programming sequence taken by most MIT EECS students place a stronger emphasis on high-level languages such as Python, which abstract away the interactions that a program must have with memory. Previously, if C had been introduced in an introductory-level class, it was one of several simultaneous concepts being taught to the students and therefore was not explored in depth. In September 2022, MIT EECS revised the class requirements for two of its degrees, Electrical Engineering and Computer Science (Course 6-2) and Computer Science and Engineering (Course 6-3) [1] to require a six-unit introductory course that focuses on low-level programming using C and assembly language. This thesis focuses on the establishment of this introductory low-level programming class intended for students positioned early in the EECS curriculum. Students taking this class study C and assembly language so that they can enter later coursework with both the ability to use these programming languages and a basic understanding of computing systems and associated constraints.

Thesis Supervisor: Joseph D. Steinmeyer
Title: Senior Lecturer

Acknowledgments

My involvement in Introduction to Low-level Programming in C and Assembly (6.190) is largely due to the support of two people instrumental to its development – Joseph Steinmeyer and Silvina Hanono Wachman. Thank you both for the countless contributions you have made to this class, and for always welcoming my ideas, valuing my feedback, and providing guidance. I have enjoyed collaborating with you, and I will look back fondly on these two years.

This thesis would be telling a different story if not for the person who presented me with this topic in the first place. Joe, thank you for providing me with this unique opportunity, from which I have been able to learn valuable lessons about course development and education.

Another significant component of my MEng was working as a teaching assistant for 6.191 (previously 6.004) for three semesters. Silvina, thank you for granting me that opportunity, allowing me to play a part in introducing students to the world of digital computing systems.

One major takeaway from this experience is that it takes many people to run a large class smoothly, especially when it has an in-class laboratory component. Thank you to all of the instructors, teaching assistants, and laboratory assistants for all of your hard work, feedback, and contributions throughout the first five offerings of 6.190. It has been a pleasure working with you all!

Thank you to my friends – particularly Daria Bakhtiari, Julia Keenan, Rachel Prevost, Sarah Berube, and Veronica Walsh – for all of the memories, laughs, and fun over the years. You all inspire me, and I am thankful to have you in my life.

Finally, thank you to my parents for supporting me in all of my endeavors. I am grateful for all of the sacrifices you have made to ensure my siblings and I each have the opportunity to realize our fullest potential.

Contents

List of Figures

List of Tables

Chapter 1

Background

There are various software applications, such as systems-level programming and embedded firmware development, where it is critical for a programmer to have control over computing resources such as memory. The C programming language is widely used for these applications due to its nature as a low-level language that allows programmers access to these resources while still being compatible with various computer architectures. After a programmer writes a C program, it is compiled into an assembly language program, which is made up of the basic instructions that can be directly understood and executed by a computer. Different architectures understand different sets of instructions, so the set of assembly language instructions that a C program is ultimately translated into depends on the architecture of the computer being programmed. As a result of this dependence and the basic nature of assembly language instructions, assembly language is used much less frequently by programmers than C. This necessary translation of C programs to assembly language programs is done by sophisticated tools such as compilers. However, the study of assembly language provides valuable insights into how a program is ultimately translated into the binary data that is stored in memory, as well as how a program is executed by a processor.

Although exposure to low-level programming can deepen one's understanding of software programming and underlying computing systems, the MIT Department of Electrical Engineering and Computer Science (MIT EECS) has not historically required its students to take a course dedicated to it. However, in 2022, the department

updated the curricula for some of its majors. One significant change made during this update was the inclusion of a half-semester class that specifically focused on low-level programming [1]. This course is required of all students completing certain MIT EECS majors, and it is a prerequisite to a foundational digital systems and computer architecture course.

This thesis discusses the development of this new low-level programming course, named Introduction to Low-level Programming in C and Assembly (6.190), which has been offered at MIT during the last five academic quarters, starting in the second half of the spring 2022 semester. In the future, it will be offered during both halves of just the spring semester (the third and fourth quarters of the academic year). This course focuses on exploring various low-level programming topics, such as memory, binary encoding, bit manipulation, using C and assembly language. We developed a six-week curriculum that covers these topics, and we designed various hardware and software tools to enhance students' understanding of material covered in the course.

1.1 Massachusetts Institute of Technology (MIT)

The development of this course was primarily motivated by the curriculum transition that MIT EECS underwent in the fall semester of 2022, because the proposed curriculum would include a low-level programming course requirement [1]. We began development of the course during the fall 2021 semester in anticipation of this transition.

1.1.1 Old Curriculum (2017-2022)

Prior to 2022, the MIT Department of Electrical Engineering and Computer Science had three core majors: Electrical Science and Engineering (Course 6-1), Electrical Engineering and Computer Science (Course 6-2), and Computer Science and Engineering (Course 6-3). The most recent curriculum, established in 2017, included a programming track containing three courses, none of which explicitly covered low-level programming. This programming track is shown in Figure 1-1. The introductory pro-

gramming course, required of students in all three EECS majors, was Introduction to Computer Science Programming in Python (6.0001). The following two courses were strictly required of computer science students, and they could fulfill requirements for students in the other two majors. The first class, Fundamentals of Programming (6.009), also used Python almost exclusively, and the second, Software Construction (6.031), used Java, but was updated to utilize TypeScript in 2022.

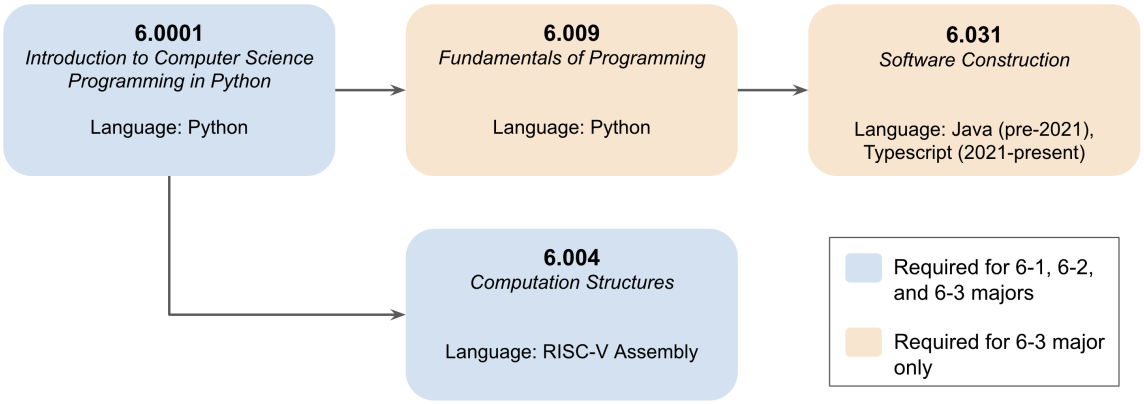


Figure 1-1: Relevant courses in the pre-2022 MIT EECS curriculum, which was introduced in 2017.

Other introductory-level courses utilized low-level languages, but they did not focus on them. Students in all three majors were required to take Computation Structures (6.004), a foundational course that introduced digital systems and computer architecture. It had just 6.0001 as a programming prerequisite, as shown in Figure 1-1. So, students who had taken just this class had only basic familiarity with Python, a high-level programming language. In order to provide the necessary background for the computer architecture portion of the course, the early weeks of 6.004 focused on binary encoding and RISC-V assembly language, bypassing the C programming language entirely. While it was critical for 6.004 to cover binary encoding and RISC-V assembly language, the time spent on those topics left little room for the class to cover more complex topics, such as operating systems. As a result, those topics were only lightly covered. Additionally, topics such as assembly language and operating systems could have been more effectively and efficiently taught if students

had some working knowledge of the C language.

Additionally, another course, named Interconnected Embedded Systems (6.08), commonly fulfilled the "Introductory Subject" requirement for all three majors, and it used a mix of C and C++, a superset of C, as a tool to program an Arduino-based microcontroller. While students gained familiarity with the language, they frequently relied on various external libraries and packages that abstracted away many details of low-level programming, including how the programming language would directly interact with the physical hardware. Additionally, the course was extremely broad. It covered a wide variety of topics, such as the Internet of Things (IoT) and databases, and it also required students to use high-level programming languages such as Python and SQL. While students did gain exposure to C, the course did not provide a solid foundation in the language and associated low-level programming topics.

Because no introductory-level or required course adequately covered low-level programming, upper-level courses that relied on C or assembly language needed to introduce the languages as they were needed, either by dedicating class time to covering them or requiring students to learn them on their own outside of class. These classes would not focus on the fundamentals of either language, but rather on how the languages could be applied to a given area. As a result, students could potentially complete these classes, and graduate with an MIT EECS degree, without truly understanding low-level programming.

1.1.2 New Curriculum (2022-)

In 2022, the department revised the curricula for the Electrical Engineering and Computer Science major (Course 6-2) and the Computer Science and Engineering major (Course 6-3). Additionally, the department updated all of its subject numbers in the fall of 2022.[1] The required programming track subjects did not change, as

[1]All MIT EECS courses were assigned updated numbers, starting in the fall 2022 semester. In the programming track, 6.0001 (Introduction to Computer Science Programming in Python) became 6.100A, 6.009 (Fundamentals of Programming) became 6.101, and 6.031 (Software Construction) became 6.102. Additionally, 6.004 (Computation Structures) became 6.191, 6.08 (Interconnected Embedded Systems) became 6.901, and 6.0004 (Introduction to Low-level Programming in C and Assembly, the subject of this thesis) became 6.190.

shown in Figure 1-2, and it continues to focus on programming using higher level languages.

Although the main programming track remained the same, the department added a required six-unit, half-semester course, Introduction to Low-level Programming in C and Assembly (6.190, previously 6.0004). This course focuses on low-level programming via both C and assembly language, and it serves as a prerequisite course for Computation Structures (6.191, previously 6.004). 6.191 no longer explicitly teaches assembly language because it is taught in this new course. Students are expected to have taken the first course in the programming sequence, 6.100A (previously 6.0001) or equivalent before taking this class, so this course will not need to cover basic programming skills. The order of these courses in the new curriculum is shown in Figure 1-2. Because 6.191 is a prerequisite course for most classes that cover systems-level programming, and the new course is a prerequisite for 6.191, students will enter those upper-level classes with an understanding of basic low-level programming concepts.

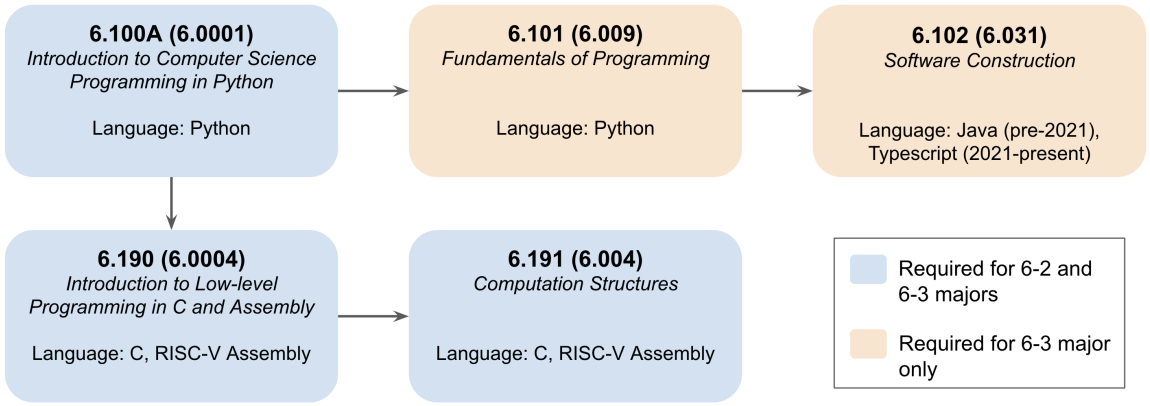


Figure 1-2: Relevant courses in the current MIT EECS curriculum, which was introduced in 2022. Notably, 6.190 was inserted to be taken between 6.100A and 6.191.

1.2 Improvements upon Existing MIT Courses

As previously noted, there were two courses in the old curriculum that did cover low-level programming: Interconnected Embedded Systems (6.08, now 6.901), which

covered C, and Computation Structures (6.004, now 6.191), which covered assembly language. Due to the curriculum change, 6.901 is not currently being offered at MIT, and 6.191 no longer teaches assembly language, instead expecting that students learn RISC-V assembly language prior, while taking 6.190. As a result, some of our course materials are adapted from 6.901's coverage of C and 6.191's coverage of RISC-V assembly language.

In 6.901, students learned C primarily as a tool to program an Arduino-based microcontroller. However, there were various compatible external libraries that students were allowed to use to control the peripherals on the microcontroller. These library functions abstract away key elements of low-level programming and hardware-software interactions, such as bit manipulation and direct memory accesses. 6.190 places more of a focus on low-level programming, such as interpreting and interacting with individual bits stored in memory.

The overall treatment of RISC-V assembly in 6.190 is largely similar to that in 6.191. However, since students have familiarity with low-level programming in C from the first portion of the course, we can assume exposure to ideas such as pointers and bit manipulation. This allows us to focus more on other key concepts, such as how RISC-V instructions are represented and organized in memory. We also developed new educational tools, such as a more convenient and user-friendly RISC-V simulator, to help students better understand how registers and memory evolve as different types of RISC-V assembly instructions are executed.

Chapter 2

Course Content

The main, universal theme of Introduction to Low-level Programming in C and Assembly Language (6.190) is memory and how low-level programming can be used to interact with it. All of the topics covered in the class relate to this overall theme in some way, and the class uses C and RISC-V assembly extensively as tools to explore them. Students, having taken an introductory programming course, should enter this class familiar with the idea that data is stored somewhere in memory. However, only having exposure to high-level programming in Python, they would not have previously needed to explicitly explore this idea in depth or even consciously interact with memory itself. This class and associated materials were designed with the primary goal of giving students an understanding of a program's underlying memory structure, how it naturally evolves throughout a program, and how to access and manipulate it using software.

The primary topics covered in this class are summarized in its listing in the MIT course catalog:

> "Introduction to C and assembly language for students coming from a Python background (6.100A). Studies the C language, focusing on memory and associated topics including pointers, and how different data structures are stored in memory, the stack, and the heap in order to build a strong understanding of the constraints involved in manipulating complex

data structures in modern computational systems. Studies assembly language to facilitate a firm understanding of how high-level languages are translated to machine-level instructions" [3].

6.100A is the introductory Python course offered at MIT that students are required to take, or otherwise show proficiency with its material, before taking this class.

2.1 C

We use the C programming language as the primary tool to understand and interact with memory because it is more similar to higher level languages, such as Python, than assembly language is. We use *The C Programming Language, Second Edition* by Brian Kernighan and Dennis Ritchie [4] as the textbook and C reference for the course. Because students are expected to have been exposed to programming in Python, the course does not introduce or focus on topics such as variables, arithmetic and logical computation, and control flow. Instead, it initially highlights where Python and C differ, primarily syntactically, then it shifts focus to new concepts such as how program variables and their contents reside in memory.

For example, a student entering this class should be able to understand the following line of Python code: `x = 5`. This is simply assigning the variable `x` to hold the value 5. In C, an equivalent line of code would be `int x = 5;`. This idea of variable assignment should be familiar to a student with a Python background, however, this course further enhances their understanding of variable assignment by explaining that:

- An `int` is a 32-bit (on the machine used in our class and in 6.191) integer value. So, 32 bits in memory will be dedicated to holding this variable, `x`.
- One byte corresponds to 8 bits. So, an `int`, which is a 32-bit value, is 4 bytes.
- Memory is byte-addressable, meaning that each address corresponds to one byte. Because an `int` is 4 bytes, the variable `x` will occupy 4 addresses in memory.

- Memory is made up of bits, which can only hold the value 0 or 1. So, numerical values stored in memory are encoded in base-2, or binary, and the base-10 value 5 will be stored in memory as the base-2 value 00000000000000000000000000000101.

This example highlights one key topic in this class – how information is represented in memory. We introduce various information encoding schemes, including binary, hexadecimal, and ASCII so students can become familiar with this idea.

Another area of focus in this class is how a programmer can directly interact with and modify memory, which is a new concept for the average student taking this course. The course discusses bitwise operations to expose students to the idea of manipulating individual bits in memory, and it introduces pointers so students can gain confidence with using addresses to access particular values in memory. The laboratory component of this class especially reinforces these concepts, with each laboratory assignment requiring some form of bit manipulation. This also allows students to understand how more complex data structures, specifically arrays, are represented in memory. The course also explores how pointers enable software control of physical hardware via memory-mapped input and output.

2.2 Assembly Language

This class introduces assembly language with the ultimate goal of providing students with an understanding of how a higher level language, such as Python or C, is translated into the binary instructions that are stored in memory and directly understood and executed by a computer. This understanding is critical for students who plan on taking any computer architecture course in the future. Students taking MIT's foundational digital design and computer architecture course, 6.191, actually implement single-cycle and pipelined RISC-V processors, so this course focuses on the RISC-V instruction set architecture (ISA) [5] to ensure that students have the prerequisite knowledge for those assignments. Specifically, it introduces the subset of the user-level RV32I (RISC-V 32-bit integer) ISA introduced in 6.191. Notably, this subset does not include sub-word load and store instructions, control and status register

instructions, or any other instructions that interact with the operating system. We also introduce a set of relevant RISC-V pseudoinstructions, while emphasizing the fact that psuedoinstructions must be translated into equivalent RISC-V instructions before being assembled into the 32-bit binary encodings understood and executed by a processor.

Students first gain familiarity with the RISC-V ISA and its different instruction types, then they spend time compiling C programs into equivalent RISC-V assembly programs. Although this translation from C to assembly language is a process normally done by a software compiler, it is important for students to understand it, as it can inform how they write higher-level programs and provide the background necessary to explore compilers in future coursework. As this course places a strong emphasis on memory, students also learn how RISC-V assembly instructions are encoded into binary values that are stored in a particular section of memory. This course focuses on the 32-bit instruction encoding of RISC-V instructions because this class, along with MIT's 6.191, uses a 32-bit machine.

Due to the limited amount of time available to introduce material in the class, 6.190 does not introduce compiler design, in favor of focusing on more memory-related topics. If this class were to be extended to be offered over a full semester, this topic would be given priority above the other topics we could introduce, as it would give students the more complete picture of how high level languages are translated into binary machine code. Currently, in 6.190, students act as the compilers, manually translating C programs into assembly language instructions and the corresponding binary, gaining the background knowledge necessary for learning about compiler design in future coursework.

Assembly language also allows students to directly interact with the stack, and they must use the stack in order to adhere to RISC-V calling convention. This is something that is not explicitly necessary when programming in C or a higher-level language, as it is generally handled when those higher-level programs are compiled into assembly language and machine code. Learning about RISC-V calling conventions and stack discipline gives students a deeper understanding of how the stack

section of memory is structured and how it evolves throughout a program.

2.3 Course Structure

Given the goals we established for this class, we identified the topics that we wanted it to cover. This course was designed with the intention to offer it during a half of a given semester at MIT, so, constrained by MIT's academic calendar, we distributed topics to create six weeks' worth of material as shown in Table 2.1.

Week 1	C Syntax Data types in C Unsigned binary encoding Hexadecimal encoding Bitwise operations Pointers in C
Week 2	Signed (two's complement) binary encoding ASCII character encoding Arrays in C Pointer arithmetic in C
Week 3	Strings in C Structures in C
Week 4	RISC-V assembly language RISC-V instruction encoding Instruction memory
Week 5	RISC-V calling convention Stacks
Week 6	Memory layout Heap Dynamic memory allocation

Table 2.1: The topics covered in 6.190 are distributed over six weeks, because the course is offered during half of an academic semester at MIT.

A typical week in 6.190 includes three class meetings: a lecture, a recitation and a laboratory section, and three assignments: a lab, a post-lab, and a set of exercises. There is one exam administered at the end of the class.

The week's material is first introduced during lecture, then it is reinforced through examples and practice problems in recitation. Students will then attend a labora-

tory section where they explore the week's concepts via developing firmware for an embedded system based on a RISC-V microcontroller. Weekly assignments include completing the lab from class, as well as completing a related "postlab" and a problem set consisting of various exercises.

2.3.1 Lecture

Each week includes a 90-minute lecture where the week's material is first introduced by an instructor of the class. The lectures are primarily presentation-based, but certain weeks feature live coding demos. The lecture meeting is scheduled at the beginning of a given week, so students can spend the rest of the week interacting with the material via the other components of the class.

2.3.2 Recitation

After lecture, students attend a 90-minute recitation section led by a course teaching assistant. Recitations focus on guiding students through solving problems using the concepts introduced in lecture.

2.3.3 Exercises

Each week, students are assigned a set of exercises to complete. These exercises range from simple examples, such as converting the binary representation of a number to its decimal representation, to more complex problems, such as writing nontrivial C functions or assembly procedures. Some exercises have been adapted from those used in 6.191 and 6.901, and some are new, original exercises developed specifically for this class.

2.3.4 Laboratory-Based Assignments (Labs and Postlabs)

There are two types of laboratory-based assignments, labs and postlabs, where students explore a week's concepts by developing firmware for an embedded system based

on a RISC-V microcontroller. The labs are attempted during a weekly 2.5-hour in-class session, so they are designed to take students that amount of time, on average, to complete. Postlabs are often continuations or extensions of that week's lab, so they are designed to be completed independently outside of class after completion of the lab. Labs and postlabs both include short check-in conversations with a staff member to verify system functionality and student understanding of their implementation and related course topics. The development of the lab and postlab assignments is a central focus of this thesis, and it is discussed in depth in Chapter 3.

2.3.5 Final Exam

There is an exam during the sixth week of classes that assesses students' understanding of topics covered during the first five weeks of the course. Because the topics at the core of the course are emphasized and applied during the labs and postlabs, exams include questions related to them. Topics, including memory layout, the heap, and dynamic memory allocation, introduced during Week 6 are not included on the final exam due to Institute regulations surrounding scheduling exams at the end of the term.

Chapter 3

Laboratory Component

One noteworthy aspect of this course is its laboratory-based assignments. These hands-on exercises focus on developing firmware for an embedded system that we assembled specifically for the class. The primary goal of these assignments is to provide students with the opportunity to apply particular core topics introduced in class to a practical application, thus enhancing their understanding of the material. Additionally, these assignments introduce students to the field of embedded systems early in their electrical engineering and computer science careers and provide exposure to other areas, such as circuits. We created one laboratory assignment (lab) and one post-laboratory assignment (postlab) for each of the first five weeks of the course.

3.1 Laboratory Infrastructure

The lab and postlab assignments rely on programming a physical embedded system, so we needed to identify and design software and hardware tools to support embedded systems development. Specifically, we needed to design an embedded system and select useful software development tools. While doing this, we considered important factors including availability, cost, user experience, and educational benefits. Availability and cost are critical because we must obtain sufficient quantities of any

necessary items in a timely manner so that hundreds[1] of students at a given time have the necessary tools to complete the assignments. User experience is a significant factor because students completing these assignments are not expected to have any relevant background beyond basic Python programming exposure. Ideally, the tools would be simple to use so that they don't distract students from the educational goals of the assignments. Lastly, this system is being used as an educational tool in a classroom environment, so some components were selected based on how well they could facilitate understanding of the course material.

3.1.1 Hardware

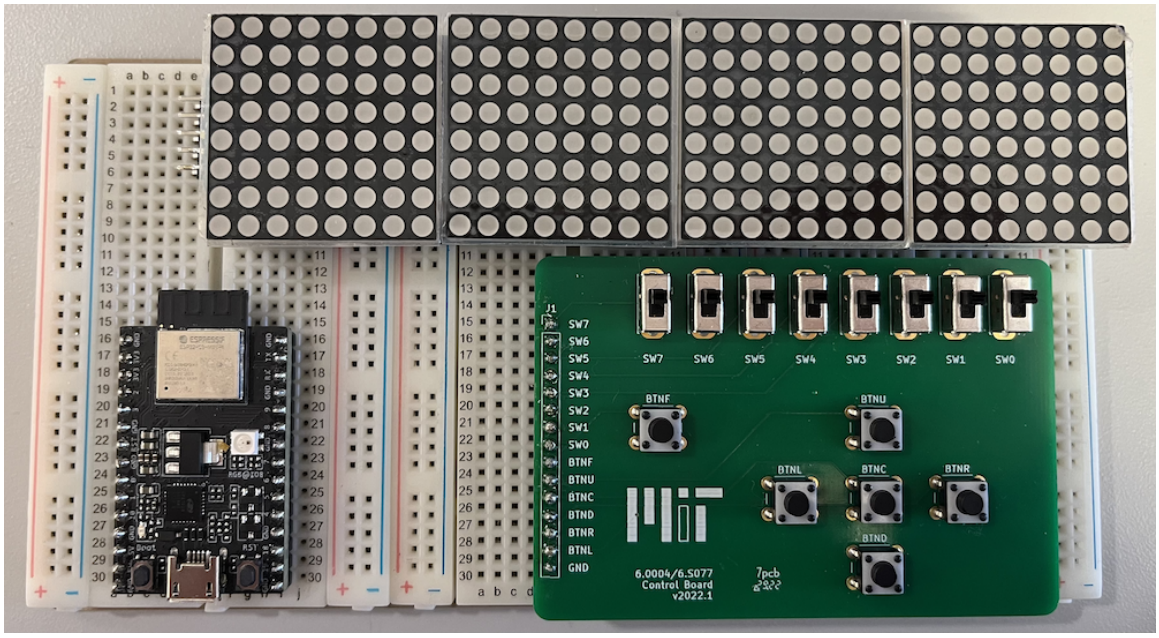


Figure 3-1: A 6.190 lab kit containing a RISC-V development board, course-specific printed circuit board, and an LED display.

Each student taking 6.190 is given a physical kit that they are required to use while completing labs and postlabs throughout the class. This kit consists of three main hardware components: a development board based on a RISC-V microcontroller, a printed circuit board (PCB) containing buttons and switches, and an 8x32 LED

[1]Based on historical enrollment in this class and in 6.191, we anticipate that enrollment in future offerings of 6.190 may reach upwards of 300 students per offering.

display. It also includes a wiring kit and breadboards to connect these components, as well as a cable and corresponding adapter to connect the microcontroller to their computer to power the development board and program the microcontroller. A lab kit, as it would be provided to students at the beginning of the class, is shown in Figure 3-1.

The focal component of the lab kit is the development board containing the RISC-V microcontroller that the students will program. We initially developed the labs and postlabs around the SparkFun RED-V Thing Plus development board, which features a SiFive FE310 system-on-a-chip with a RISC-V processor. However, during development, we encountered an intermittent problem with uploading programs to the board, so we decided to base the assignments on the Espressif Systems ESP32-C3-DevKitM-1 development board. This development board includes an Espressif Systems ESP32-C3-MINI-1 module with an ESP32-C3 RISC-V microprocessor. One additional advantage of the Espressif board is the cost: on Digi-Key Electronics, as of May 2023, the Espressif development board cost $8.00,[2] while the Sparkfun development board cost $32.50.[3] At scale, when hundreds of students are taking the class at a given time, the $24.50 price difference per individual board can result in a significant reduction in cost.

Most of the designs the students are targeting in these assignments require user input in the form of pressing a button or toggling a switch. In total, the assignments require eight switches and six buttons. We designed a printed circuit board containing these components so they are positioned in a uniform arrangement convenient for interacting with the systems developed for the assignments. Students can easily connect the buttons and switches to the microcontroller using just one wire for each input and one wire for ground, reducing the amount of time and thought required to set up the hardware system for each lab. The circuit is designed so that when a switch

[2]Espressif ESP32-C3-DEVKITM-1 Development Board (Digikey): `https://www.digikey.com/en/products/detail/espressif-systems/ESP32-C3-DEVKITM-1/13684315`. Price reference in text accurate as of April 29, 2023.

[3]Sparkfun DEV-15799 Development Board (Digikey): `https://www.digikey.com/en/products/detail/sparkfun-electronics/DEV-15799/10715591`. Price referenced in text accurate as of April 29, 2023.

is in the down position or a button is pressed, a connection can form between the corresponding input pin and ground, resulting in a digital low value on the pin. When a switch is in the up position or a button is not being pressed, the corresponding input pin would be disconnected from ground and an internal pull-up resistor would make the pin input have a digital high value.

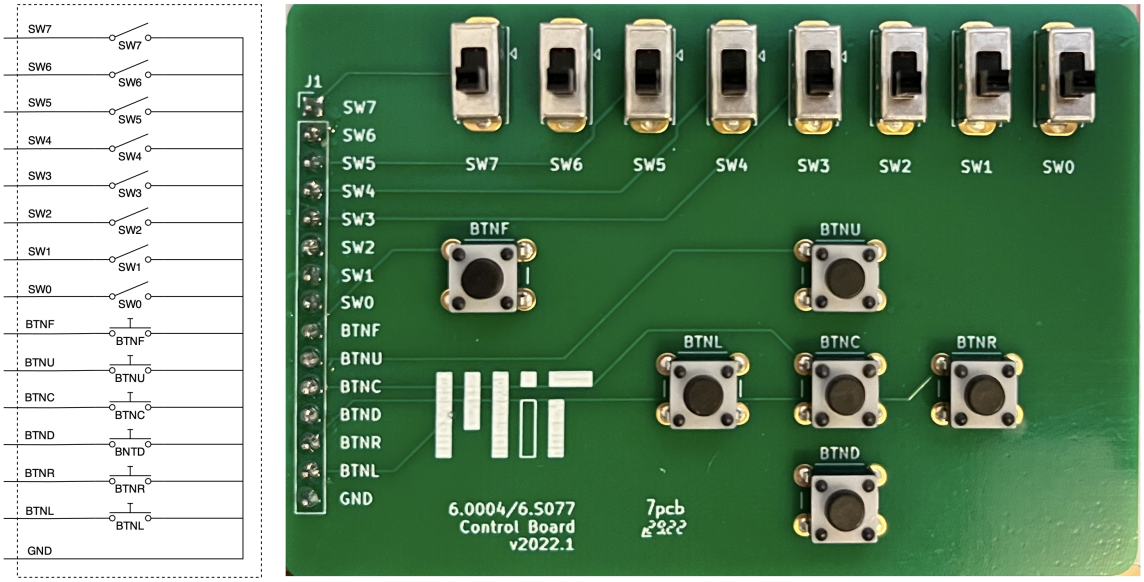


Figure 3-2: The printed circuit board (PCB), containing six buttons and eight switches, designed specifically for use in the 6.190 embedded system.

Some assignments require the system to provide feedback to students on a display. We could have simply used the serial output from the microcontroller that students could view on their computer screens to achieve the necessary functionality. However, we wanted to use something more compelling that could further motivate our choice to focus the labs on embedded systems applications and that could potentially assist students in understanding the material presented in the course. We ultimately chose to use an LED display with 8 rows and 32 columns, choosing it over an LCD display due to the educational benefits of its low resolution and simplicity.

A simple, low-resolution display is especially favorable for use in a class focused on memory because it can be used to visualize the individual bits in a fathomable data structure, such as a one-dimensional array. In this class, we use the 8x32 LED display

to represent an array with eight unsigned 32-bit integer elements. Each element in the array corresponds with a row on the display, and each bit in a given element corresponds with a pixel in that row.

We strategically chose the origin point, or orientation, of the display so that the LEDs would directly match the contents of memory. The first element of the array is be rendered in the top row of the display, and each row is a direct representation of the binary encoding of the corresponding value in the array – the leftmost LED in a given row corresponds to the most significant bit, and the rightmost LED in a given row corresponds with the least significant bit. For example, putting this together, if a student modifies the least significant bit of the first array element, the rightmost pixel in the top row of the display would change. Figure 3-3 provides an example of how the contents of a one-dimensional array in memory correspond to the contents of the LED display.

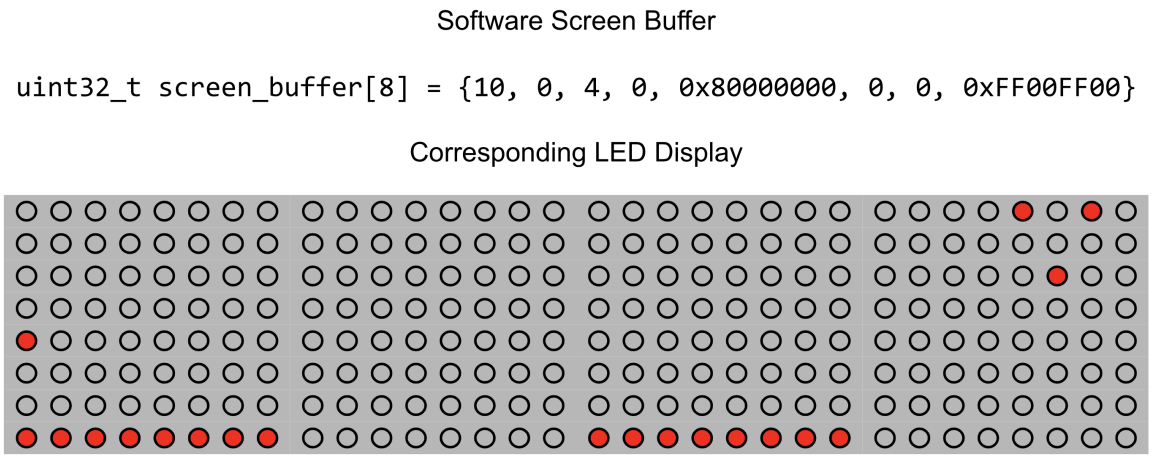


Figure 3-3: The contents of a software screen buffer array directly correspond to the contents of the 8x32 LED display included in the 6.190 lab kit.

We provide students with the SPI interface that transmits data from an array in memory to the LEDs on the display. We chose to implement this interface purely using software, rather than using the hardware SPI peripheral available on the microcontroller. This software SPI interface is educationally beneficial for two main reasons: it is primarily driven by functions that are ultimately implemented by students during the labs, and the interface can be understood by the students taking the

class.

3.1.2 Software

Students are instructed to use the PlatformIO Visual Studio Code extension to complete the lab and postlab assignments. We chose to use PlatformIO for this class because it is simple for beginners to learn and use, supports a myriad of microcontroller hardware platforms, and is compatible with all of the major computer operating systems (Windows, MacOS, and Linux). Students are instructed to use Visual Studio Code as their code editor due to its integration with PlatformIO.

PlatformIO is also available to be used as a command-line interface (CLI), which would remove the dependency on Visual Studio Code, but we decided against using it. We do not expect students to have any familiarity with using the command line and, because this class is only six weeks long, it would be suboptimal for the tools to have a learning curve that would distract students from the course material. If the prerequisite course, 6.100A, were to explore using the command line and related tools, this option would become more compelling, as it would not require as much overhead to become familiar with the tools.

3.2 Laboratory Assignments

There are two types of laboratory-based assignments: labs and postlabs. They are designed to be completed sequentially, with the lab coming before the postlab. Students are provided with detailed instructions and starter code (see Appendix A), that includes any code necessary for the system's functionality but will not be implemented by students due to time constraints, for each assignment.

3.2.1 Lab 1: Introduction

The first lab provides an overview of the lab kit and PlatformIO, then it guides students through writing C functions that interact with the general purpose input and

output (GPIO) pins on the ESP32-C3 microcontroller. Students are given a header file (see Appendix A.1.2) with declarations of board-specific parameters and functions that are used extensively in future labs and postlabs. Some of these functions are already implemented for students, and the rest of them are implemented by students during this lab.

Conceptually, this lab connects to the central theme of memory because it focuses on writing low-level software to interact with individual bits in memory using C. This requires the use of bitwise operations and pointers. The lab first introduces students to three main bit operations: setting bits, clearing bits, and reading bits. Then, students apply these operations to three GPIO-related functions:

- `pinSetup`, a function to configure a GPIO pin on the ESP32-C3 as either an input or an output
- `pinWrite`, a function to write a digital value to an output pin on the ESP32-C3
- `pinRead`, a function to read a digital value from an input pin on the ESP32-C3

These functions are used for the same purposes as Arduino's `pinMode` [6], `digitalWrite` [7], and `digitalRead` [8], respectively. Oftentimes, hobbyists or beginner-level students taking classes such as MIT's 6.901 would have used these functions without needing to understand the details of their implementations. However, this class requires them to actually implement them and think about the underlying memory structure.

For example, the ESP32-C3 microcontroller has a designated memory address, `0x6000403C`, that holds a 32-bit value where each bit corresponds to the digital value of a GPIO input pin [9]. For a program to obtain the value of a GPIO input pin, it must first access the contents of that memory address, then extract the individual bit that corresponds with the input pin of interest. This requires the use of pointers, bit shifting, and bitwise logical operations, which are key topics central to the first week of 6.190.

Students develop each individual function on the class website, using checkers that run the their code submissions against various test cases to verify functionality and

assign a grade accordingly. Once a student's code passes the online test cases, they can transfer it locally to Visual Studio Code, then use PlatformIO to compile the project and upload the resulting binary to the development board. They further verify the behavior of the GPIO output pins by connecting two LEDs to two output pins on the microcontroller, and they use a button input and serial monitor output to verify the functionality of the input pins. Then, they expand their program so that they use logical operations based on two button inputs to control the two LED outputs – for example, they are instructed to make one LED consistent with the output of an exclusive-or function between the two button inputs. This exercise is included with the lab to require students to reason about logical and bitwise operations in C.

3.2.2 Postlab 1: Hit the Bit

In Postlab 1, students build a game named "Hit the Bit," which is based on an early computer game called "Kill the Bit." For this game, the system uses eight switch inputs and eight adjacent LEDs on the LED display. The game starts with one illuminated LED that continuously shifts to the right until it reaches the rightmost column, then it wraps around to the leftmost column and continues to move to the right. This behavior, independent of the game logic, is displayed in Figure 3-4.

Each switch maps to one of the eight LEDs, and on each step, the state of each switch is compared to the state of each LED on the display. A switch in the up position matches an illuminated LED, and a switch in the down position matches an LED that is turned off. If a switch does not match its corresponding LED, then that LED should be illuminated on the next step. Otherwise, that LED should become or stay turned off. The player's objective is to turn off all of the LEDs by flipping the switches so that they match the state of the eight LEDs. Figure 3-5 provides an example of typical gameplay.

Students implement three main components of the game. First, they implement the right-shifting and wrapping behavior. The LEDs are represented by an unsigned eight-bit integer variable. Next, they write a function to store the states of the eight switches in another unsigned eight-bit integer variable, which also requires the use of

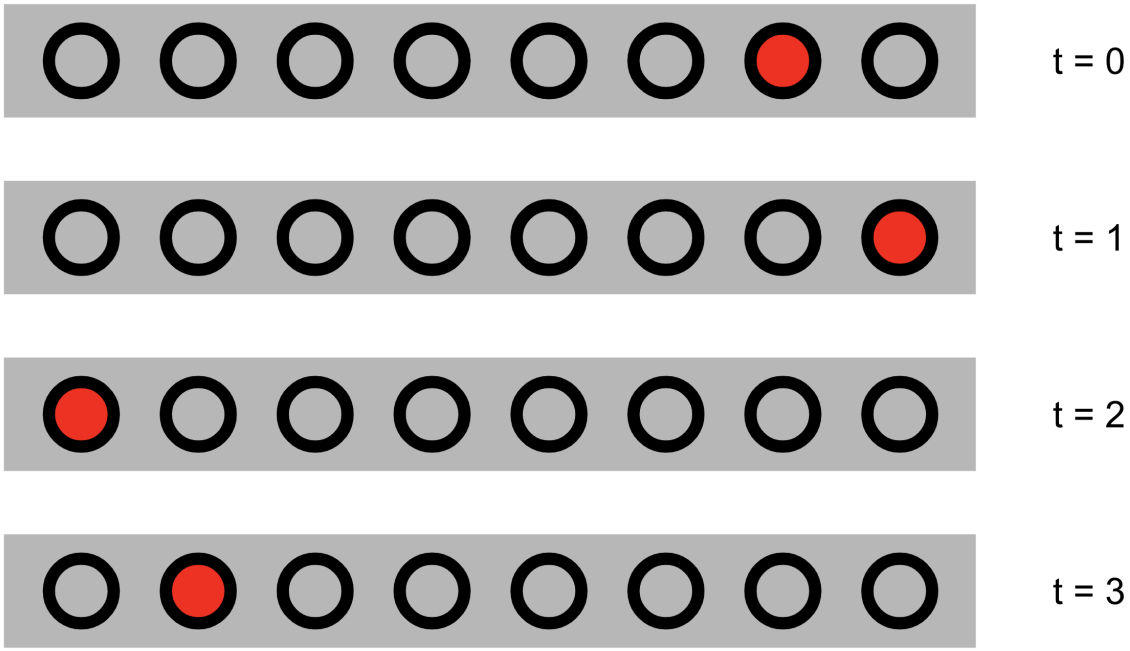


Figure 3-4: Postlab 1 initial right shifting behavior independent of switch inputs and game logic. The state of the rightmost column wraps around to the leftmost one on each step.

their `pinRead` function from Lab 1 and bit shifting. Last, they write a line of code to compare the state of the eight LEDs to the state of the eight switches. Throughout these exercises, students are required to write C code that uses bit shifting and bitwise operations to achieve a working game.

3.2.3 Lab 2: ASCII

In Lab 2, students create a system that displays messages on an LED display, as shown in Figure 3-6. While developing this system, students are introduced to higher-level, relevant ideas such as ASCII encoding, null-terminated strings, state, and character rendering while still being required to interact with individual bits in memory. Students also gain familiarity with arrays – this lab is the first where students use an array in memory to represent the contents of the 8x32 LED display, and the message ultimately rendered on the LED display is initially stored in a character array.

Students first write a function to read the input pins connected to seven switches

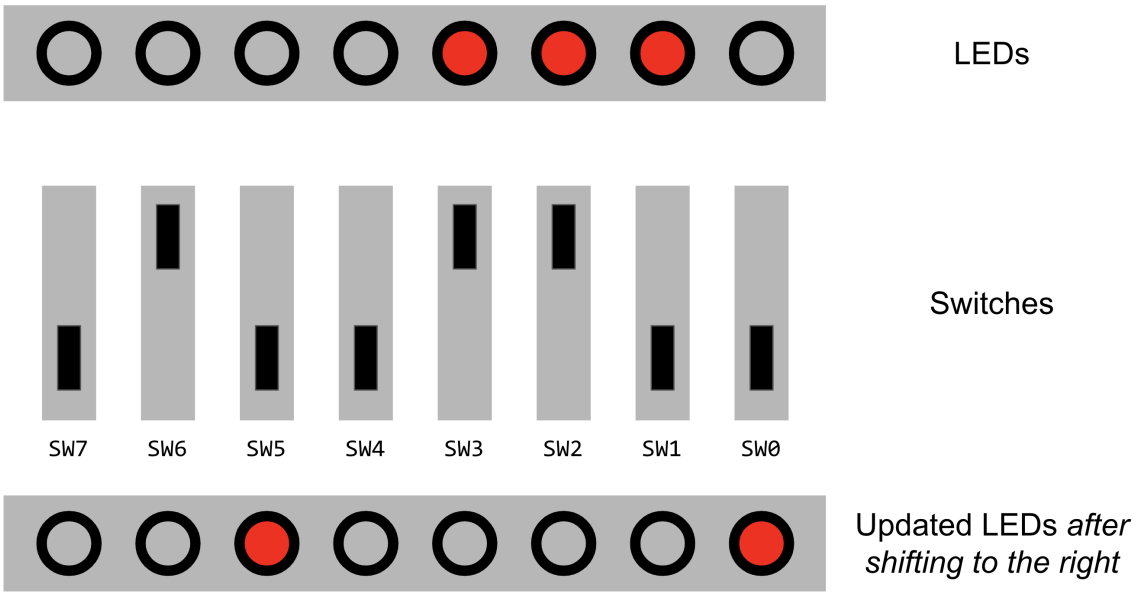


Figure 3-5: Example of typical Postlab 1 gameplay. Two switches, SW6 and SW1, are in the incorrect positions, while the rest are correct. As a result, the LED corresponding with SW6 will turn on, and the LED corresponding with SW1 will stay on. The rest of the LEDs will become or remain turned off. Then, all of the LEDs will shift to the right, and the rightmost LED will wrap around to the leftmost column.

and interpret them as an eight-bit integer, with the most significant bit set to 0. Switch 0 (SW0) on the printed circuit board (PCB) corresponds to the least significant bit in the value, and switch 6 (SW6) corresponds to the most significant bit, as displayed in Figure 3-7. This orientation was chosen because it matches that of a binary number, making each switch input equivalent to the corresponding bit in an eight-bit value. This function should use bit shifting and bit masking, requiring students to recall core concepts from the previous week. The eight-bit value returned from this function is then interpreted as an ASCII character when it is printed on the serial monitor.

Students then add a button input to their system so that the switch inputs are only sampled once upon a button press. After verifying that the system correctly prints a single character upon a button press, students are instructed to write a program that uses the switches and button inputs to fill an array of characters with a null-terminated string. A character should be added to the string upon each button

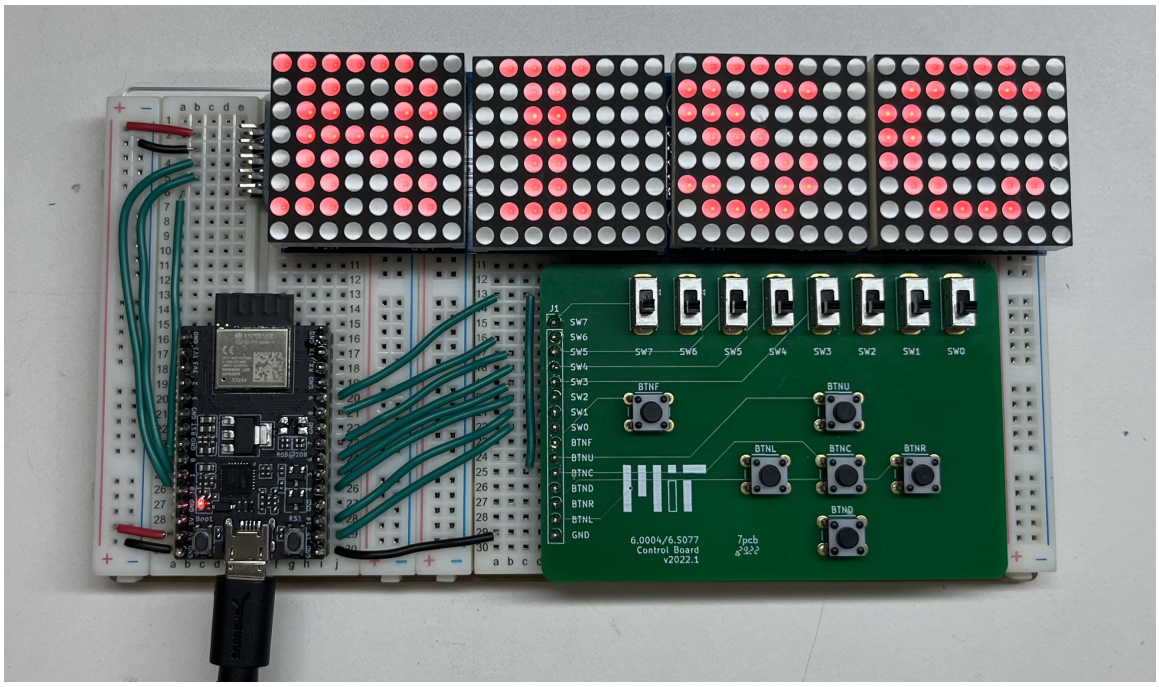


Figure 3-6: Lab 2 system displaying the message "RISC" on the 8x32 LED array.

press, and the string should be printed to the serial monitor when the null-terminating character is entered. Once the string has been printed, the array should be reset so that the same section of memory can be used again to store another string. This exercise requires students to work with arrays using C, and it helps them to become familiar with how strings are represented in memory.

The remainder of the lab focuses on rendering these messages on the system's LED display. First, students are introduced to an adapted header file, named `ascii.h`, containing an 8x8 bitmap for each character [10]. The file consists of a two-dimensional array with one entry for each of the 128 ASCII characters. Each entry is an array containing eight unsigned eight bit integers – each integer maps to a row, and each bit indicates whether or not a pixel in that row should be illuminated, as displayed in Figure 3-8.

Students must write a C function that, for up to the first four characters in a given string, places the contents of the 8x8 bitmap from the header file into the screen buffer array so that,when the data in the array is transmitted to the LED

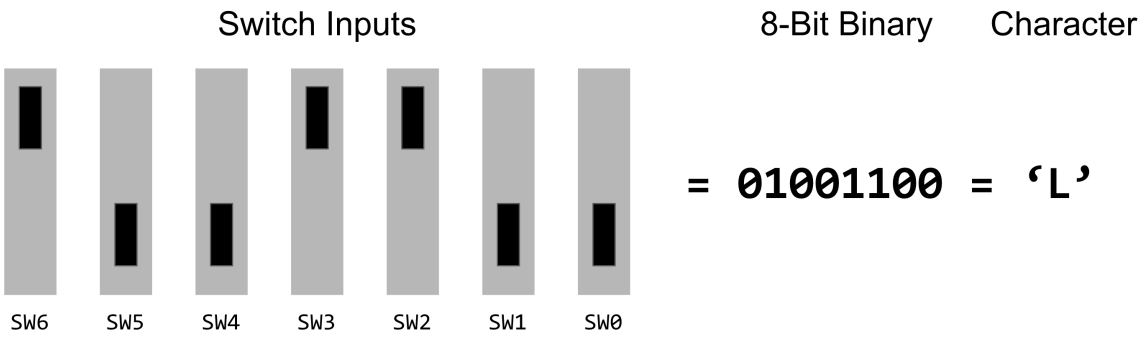


Figure 3-7: In Lab 2, switch inputs on the lab kit are interpreted as eight-bit binary numbers and ASCII characters.

display, the message will correctly appear on it. This requires students to interact with memory by accessing or modifying both entire array elements and the individual bits in those elements.

Due to the unconventional way we chose to orient the LED display, the bitmaps from the original ASCII header file resulted in the letters appearing to be flipped horizontally on our display, as shown in Figure 3-9. During the first three offerings of the course, students were required to correct this using a provided helper function to reverse the bits in each value they used from the ASCII header file before placing it in the screen buffer. This ended up being a large source of confusion for students, and it distracted them from the intended purpose of the lab, so in more recent iterations of the class, we provided students with an updated ASCII header file where the bits in each array entry are oriented so that students can use the values directly, as shown in 3-9.

3.2.4 Postlab 2: Scrolling Text

For the second postlab, students develop a system that displays a scrolling message on the 8x32 LED display. The rest of the system is fairly similar to that from Lab 2, where students wrote a program that would statically display a message, except, rather than requiring students to enter the message through switches and button presses, messages are hard-coded in the program. The scrolling effect is created by

```
ascii[71] =  { 0x3C, 0x66, 0xC0, 0xC0, 0xCE, 0x66, 0x3E, 0x00}

ascii[71][0] = 0x3C = 0b00111100
ascii[71][1] = 0x66 = 0b01100110
ascii[71][2] = 0xC0 = 0b11000000
ascii[71][3] = 0xC0 = 0b11000000
ascii[71][4] = 0xCE = 0b11001110
ascii[71][5] = 0x66 = 0b01100110
ascii[71][6] = 0x3E = 0b00111110
ascii[71][7] = 0x00 = 0b00000000
```

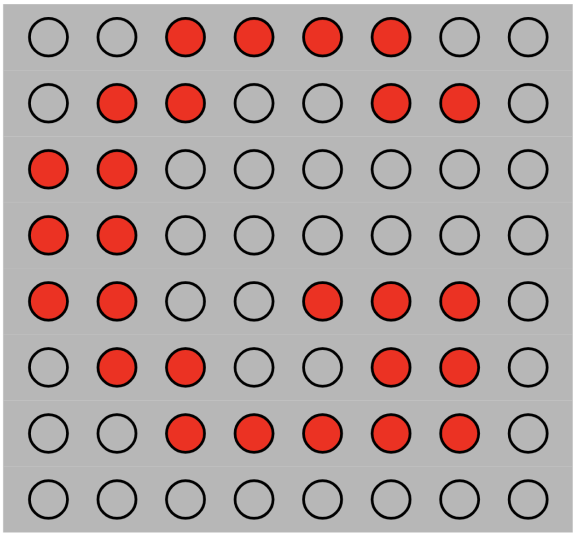


Figure 3-8: An example of rendering an 8x8 character on the LED display using a provided bitmap.

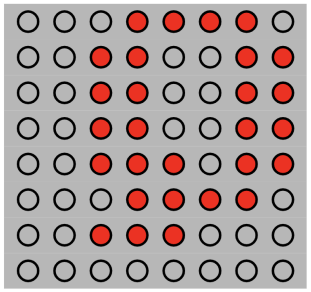
moving the message on the LED display one spot to the left periodically, as shown in Figure 3-10. In the lab, we update the display approximately every 100 milliseconds to create a smooth scrolling effect.

For this assignment, students first write a function that, given a null-terminated C string, calculates the number of characters in the message. This is similar to the C library function `strlen` [4]. The program then uses the calculated length of the string to determine the number of images, or frames, that must be rendered in total to display the entire message – each successive frame effectively replaces the leftmost column in the image by shifting every column to the left and filling the rightmost column with new data, so this can also be thought of as the number of columns required to render the entire message. The starter code for the lab then iterates through each column in the message, showing the relevant portion of the message on the LED display on each iteration.

The majority of the postlab involves writing a function that, given an array containing a null-terminated string and the number of loop iterations that have occurred since the beginning of the message, displays the correct portion of the message on

Rendering 'Q' (ASCII value 81) using the original values from ascii.h directly.

```
ascii[81] = { 0x1E, 0x33, 0x33, 0x33, 0x3B, 0x1E, 0x38, 0x00}
```



Rendering 'Q' (ASCII value 81) using the updated values from ascii.h directly.

```
ascii[81] = { 0x78, 0xCC, 0xCC, 0xCC, 0xDC, 0x78, 0x1C, 0x00}
```

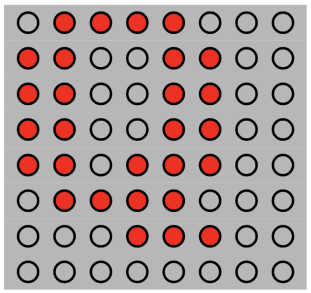


Figure 3-9: Using the values from the original ASCII header file directly resulted in letters appearing flipped on our LED display. The values needed to be reversed in order for the letter to appear correctly. Students originally had to use a helper function in their program to reverse the bits in each value they used from the file, but in later iterations of the course we provided them with an updated ascii.h file with the entries flipped for them.

the LED display. In Lab 2, the LED display rendered four 8x8 characters, which were aligned in the 32-bit wide screen buffer. In this assignment, these characters may not be perfectly aligned, so portions of the leftmost and rightmost character may be displayed. This function must first determine which characters, as well as what portions of those characters, to render. Then, it must obtain the bitmap for each of those characters from the ASCII header file used in Lab 2 and set the bits in the screen buffer accordingly, with the new consideration that only portions of the first and last character may be displayed. This assignment is intended to give students more exposure to arrays in C, as well as to manipulating individual bits in memory through their interactions with the screen buffer.

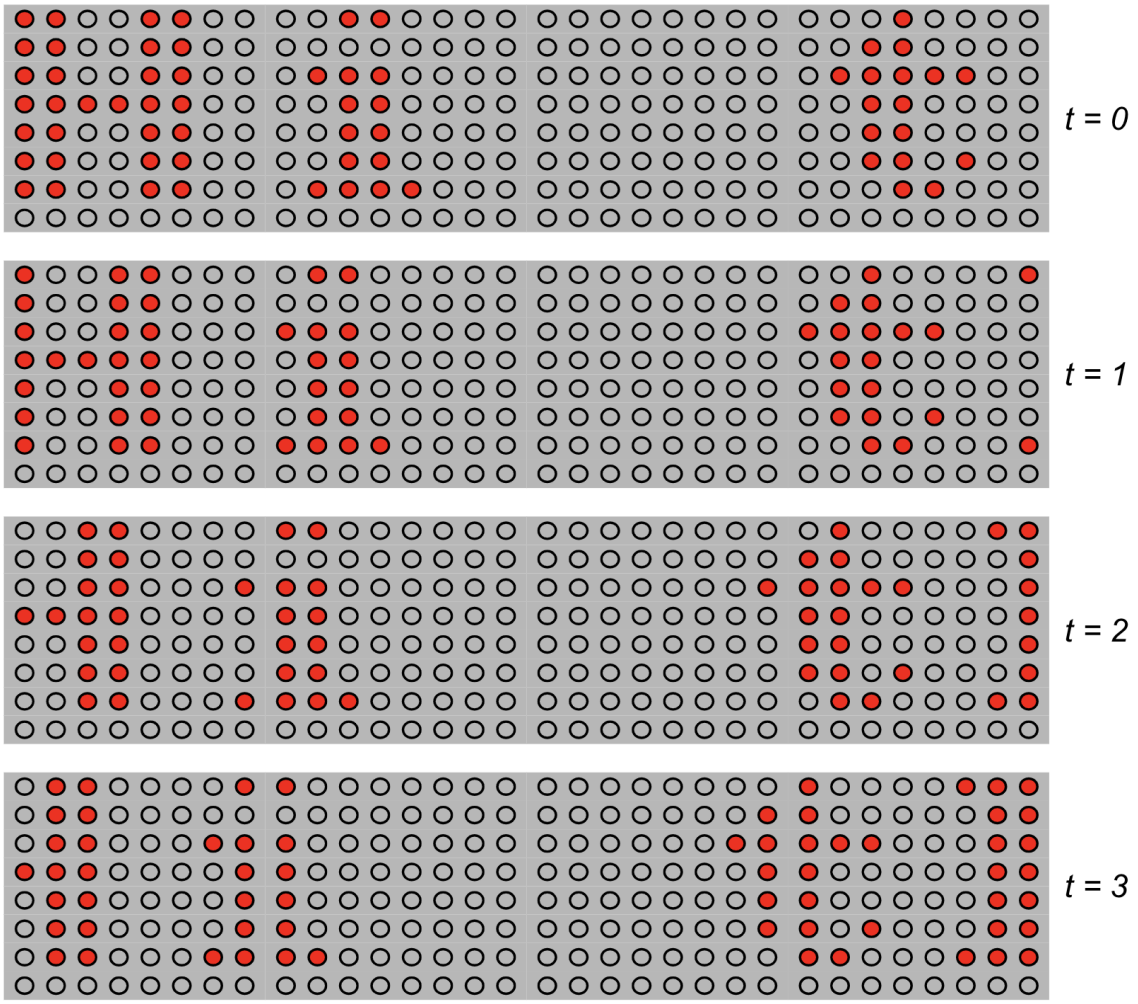


Figure 3-10: A scrolling effect can be created on the LED display by periodically shifting the contents of the display. Above, the message. "Hi, there!" begins to scroll across the display.

3.2.5 Lab 3: Snake I

During Lab 3 and Postlab 3, students implement a Snake game, where the objective is to grow a continuously-moving snake as large as possible without allowing it to collide with itself. A player navigates the snake around the board in search of food, and, when the snake encounters food, it eats it and grows. The game ends when the snake's head collides with its body. Some game variations also end the game when the snake collides with the edge of the game board, but our implementation instead has the snake wrap around to the other side of the board when it encounters

an edge. This lab requires five button inputs: four for navigation and one for game reset. Additionally, the LED display is used to render the game board.

The lab opens with a discussion of how to represent a given location on the game board. We use the 8x32 LED display as our game board, so the representation must be able to encode at least 256 unique location values. We initially considered representing each location with two unsigned eight bit integers, one for the x-coordinate and one for the y-coordinate. The primary benefit of this representation is that it's straightforward and intuitive; many students would have likely chosen this approach if not given any guidance. We ultimately decided to represent each location as a single unsigned eight-bit integer, where the x-coordinate is encoded in the upper five bits and the y-coordinate is encoded in the lower three bits, as shown in Figure 3-11. This approach requires half of the memory than the original, and it employs a more interesting and educational encoding scheme that requires students to use bitwise operations and shifting to extract and modify individual bits in the variable. Lastly, there is a modulus built into this encoding, since there are not any extra, unused bits. This results in the game having the desired "wrap-around" behavior without needing to explicitly implement it.

Students first implement four helper functions to extract information from variables representing locations:

- `getX`: returns the x-coordinate of a location
- `getY`: returns the y-coordinate of a location
- `setX`: updates the x-coordinate of a location
- `setY`: updates the y-coordinate of a location

These functions require the use of operations that modify or extract bits stored in memory, which strongly reinforces key concepts that have been introduced during the first three weeks of the course.

The following section of the lab involves rendering the game board on the LED display. Each of the objects in the Snake game is associated with an unsigned eight-

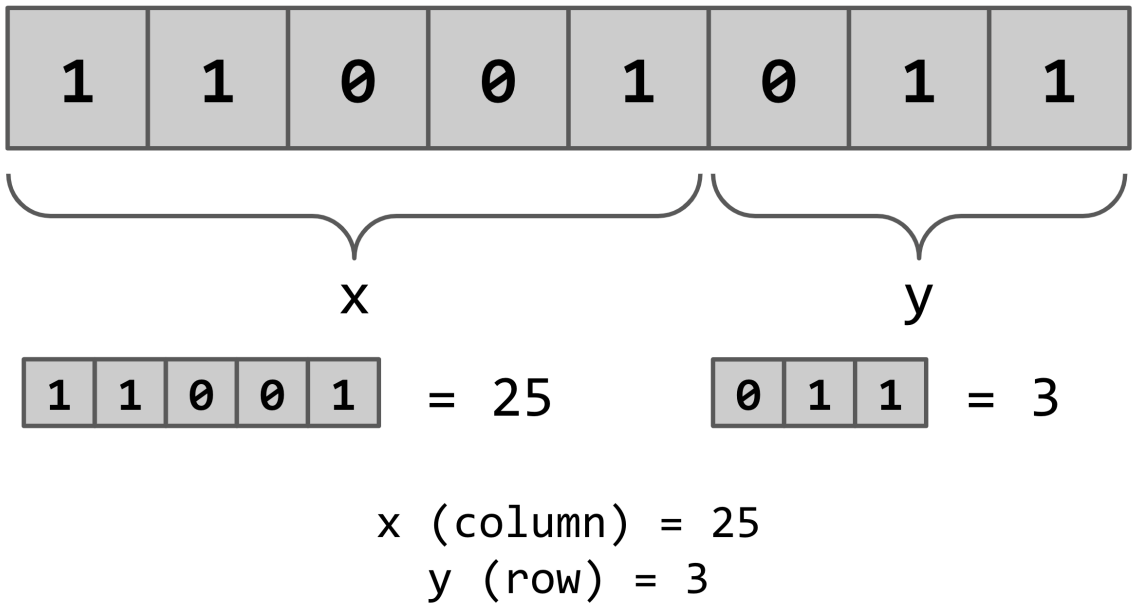


Figure 3-11: In Lab 3 and Postlab 3, locations on the 8x32 LED display are represented using a single unsigned eight-bit variable, with the upper five bits encoding the column and the lower three bits encoding the row.

bit value indicating its location on the board, using the representation previously discussed. The program uses these locations to set the corresponding bits in the software screen buffer, which is then transmitted to the LED display. Students are first instructed to write a function named `setPixel` that sets a bit in the software screen buffer given a location and a binary value indicating whether or not the LED at that location should be on. This function is conceptually relevant because it requires the manipulation of individual bits in the screen buffer array.

Structures in C are introduced during the third week of the class, so the snake used in the game is represented using a C structure containing three members: an array representing the snake's body, the direction of the snake's movement, and the length of the snake. The elements of the body array are unsigned eight-bit integers representing the locations on the game board occupied by the snake. The first element in this array is the front of the snake, or its head, and the number of array elements considered when rendering the snake is dictated by the length member of the structure. The other object in the game is the food eaten by the snake, and the program keeps track

of its location in an unsigned eight-bit integer variable. Students are instructed to write a function, `drawBoard`, that renders both the snake and the food on the game board using the `setPixel` function they had previously implemented.

The program must then be updated so that the player can use button presses to change the snake's direction of travel. Then, students implement a function named `updateSnake` that moves the snake one step in its current direction of travel. This function requires students to update the Snake structure by using a pointer to it, which is a topic introduced in the third week of the class. The final task in the lab portion of the Snake project is implementing a function to randomly generate food for the snake. After this lab is completed, the system is in a state where a player can interact with a continuously-moving snake, but the game logic has yet to be implemented.

3.2.6 Postlab 3: Snake II

In Postlab 3, students complete their Snake game by implementing the game logic. First, the game must be able to detect when the snake has eaten food, which occurs when the snake runs into it. Students implement a function, `snakeAteFood`, that, given the locations of the snake's body and food, returns whether or not the snake's head is occupying the same location on the game board as the food. The game must also be able to identify when the snake collides with itself, resulting in a loss. Students implement a function, `snakeCollisionCheck`, that, given a pointer to a Snake structure, returns whether or not the snake's head has collided with its body. After these functions have been implemented, the system works like a conventional Snake game.

Students must update their program so that the game will restart if the user presses a reset button after the previous game has ended. Lastly, because this is the final lab that focuses on C, students are given the opportunity to implement one open-ended enhancement to their snake game. These enhancements include:

- Ending the game if the snake travels out of bounds.

- Adding a pause button to the game.
- Having the snake move faster every time it consumes food, and having the snake slow down each time it moves without eating food. The game ends if the snake slows down too much.
- Keeping track of the score throughout the game and reporting it at the end.
- The snake should visibly decay until it's no longer visible when the game ends.

Students are not provided any written guidance for any of these enhancements so they can independently reason about how they should design and implement them.

3.2.7 Lab 4: Introduction to Assembly

The fourth lab is the first that focuses on RISC-V assembly language, and it is specifically designed to help students become comfortable translating C programs into assembly language programs so they can better understand the exact operations carried out by the processor during program execution. Students are given an assembly file (shown in Appendix A.6.2), where they will put their RISC-V assembly implementations of various C functions. This file will be used along with the existing 6.190 C header file throughout the remaining lab and postlab assignments.

First, students translate some of the C functions written during Lab 1 into RISC-V assembly language. The lab instructs students to translate `pinRead` first because its RISC-V assembly implementation does not require any control flow instructions beyond the provided return instruction. This allows students to become comfortable with using computational instructions before moving on to implement more complex procedures. Then, students implement `pinWrite` and `pinSetup`, which do require the use of control flow instructions to implement conditional statements. All of these exercises require students to use RISC-V assembly instructions to interact with memory and perform computation on registers. Students verify functionality of these functions in the same way they did during Lab 1, using two LEDs connected to two output pins and printing messages to the serial monitor in response to the button input. Students

also implement a function extremely similar to their `setPixel` C function from Lab 3 in RISC-V assembly, as future assignments will involve setting bits in the software screen buffer.

Writing programs in RISC-V assembly language can be a large jump from writing programs using higher-level languages, including C, so we looked for opportunities to simplify other aspects of the lab. In particular, we chose to have students translate C functions they had previously written, rather than introducing a new application requiring new C functions. Students should already be familiar with these functions, so they can focus on how to translate them into assembly, rather than first needing to familiarize themselves with new C functions. Additionally, these functions in particular allow students to review concepts core to the class, such as bit manipulation.

The fourth week of the class, during which Lab 4 and Postlab 4 are assigned, does not cover some critical elements of assembly language, including procedure calls, RISC-V calling conventions, the stack, or the specific purpose of each RISC-V register. This both limits the types of procedures that students can write and requires the lab instructions to provide strategic guidance to students. First, none of the procedures that students are instructed to implement in this lab call other assembly procedures, so, for every procedure, a correct implementation that does not require any additional instructions for adherence to RISC-V calling conventions exists. However, we must impose a constraint on the registers that students may use in order to ensure that their implementations will follow RISC-V calling conventions without the use of additional instructions.

We also limit the set of registers that students may use in their implementations to register `x0` and registers `a0` through `a7`. We allow the use of `x0` because its value is zero, which is convenient for assembly language programs, and it is immutable, meaning that no procedure needs to be concerned with about its value being overwritten by another procedure. We chose to use registers `a0` through `a7` for two main reasons. First, they are caller-saved registers, meaning that a procedure is not responsible for maintaining their values unless it is making procedure calls itself. None of of the procedures in this assignment call any other procedures, they do not need to include

additional instructions to adhere to RISC-V calling conventions. Additionally, students will have needed to use these registers as function arguments and return values anyways.

Although the assembly procedures implemented by students in this lab do not call procedures, they are called by other functions. We cannot ignore some aspects of these interactions, even though students have not yet been introduced to them. First, each procedure implementation will need to use specific registers as function arguments and return values. Students have not yet been introduced to the specific purposes of registers, so the lab instructions tell students which register holds each function argument at the start of the procedure and which register to place the function's return value in before the procedure ends. Furthermore, each assembly language procedure requires a dedicated instruction to return control back to the procedure that called it. This instruction is included in the lab starter code at the end of every procedure.

Additionally, during the second quarter of the spring 2023 semester, we banned students from using RISC-V pseudoinstructions during the fourth week of the course, which includes Lab 4 and Postlab 4. This requires students to write assembly code using just operations that are directly carried out by a RISC-V processor.

3.2.8 Postlab 4: Bubble Sort

The fourth postlab is adapted from part of an MIT 6.191 assignment, where students translate a Python or C implementation of the bubble sort algorithm into RISC-V assembly. For this class, we removed the reference Python implementation of the algorithm because students taking this course should be comfortable with reading C programs by the time they are given this assignment. Additionally, students will add code to their bubble sort implementation that will render the algorithm on the LED display. However, after students implement the bubble sort algorithm, they will need to add a procedure call to invoke a C function that renders an eight-element array on the LED display. Students must strategically place this procedure call within their implementation so that the display updates every time a swap takes place. After

completion of the postlab, the system displays the bubble sort algorithm repeatedly on the LED display.

Just as with Lab 4, this postlab is assigned before students are familiar with topics related to assembly procedure calls, so students are provided with the same guidance regarding these topics and their implementations are subject to the same register usage constraints. We also provide students with the code necessary to make the single procedure call that they must add to their bubble sort algorithm to render the array on the LED display. This code includes both the procedure call itself, as well as the instructions necessary to adhere to RISC-V calling conventions, so that student implementations will still work as expected after the addition of the procedure call. Additionally, during the second half of the spring 2023 semester, students were not permitted to use pseudoinstructions while completing this assignment.

The postlab also includes an exercise from the same 6.191 lab where students explore the disassembly of an assembly procedure, particularly focusing on the binary encoding of RISC-V instructions. We especially prioritized keeping this exercise because this class aims to emphasize how RISC-V instructions are represented in memory in addition to how RISC-V instructions can be used.

3.2.9 Lab 5: Game of Life

The fifth lab guides students through implementing Conway's Game of Life using RISC-V assembly language, placing a particular focus on RISC-V procedure calls. In particular, it reinforces concepts pertaining to control transfer among procedures, RISC-V calling conventions, the stack, and the purposes of the different RISC-V registers. In this lab and the following postlab, students may use all RISC-V registers, and they are responsible for ensuring that their procedures all adhere to RISC-V calling convention. Students completing this lab during the second quarter of the spring 2023 semester, who were not permitted to use pseudoinstructions to complete the previous week's assignments, are allowed to use them during this lab and Postlab 5. Students completing this lab during the other offerings of the course were allowed to use psuedoinstructions for all assignments, including this lab.

The lab first introduces students to the rules of the game [2]:

1. A living cell with less than two living neighbors dies due to underpopulation.
2. A living cell with two or three living neighbors stays alive.
3. A living cell with more than three living neighbors dies due to overpopulation.
4. A dead cell with three living neighbors becomes a living cell due to reproduction.

These rules are simplified to:

1. A living cell with two or three living neighbors stays alive.
2. A dead cell with three living neighbors becomes a living cell.
3. All other living cells die, and all other dead cells remain dead.

The reduced set of rules is used for the lab's implementation of the game. These rules are applied repeatedly to a game board full of cells, as shown in Figure 3-12.

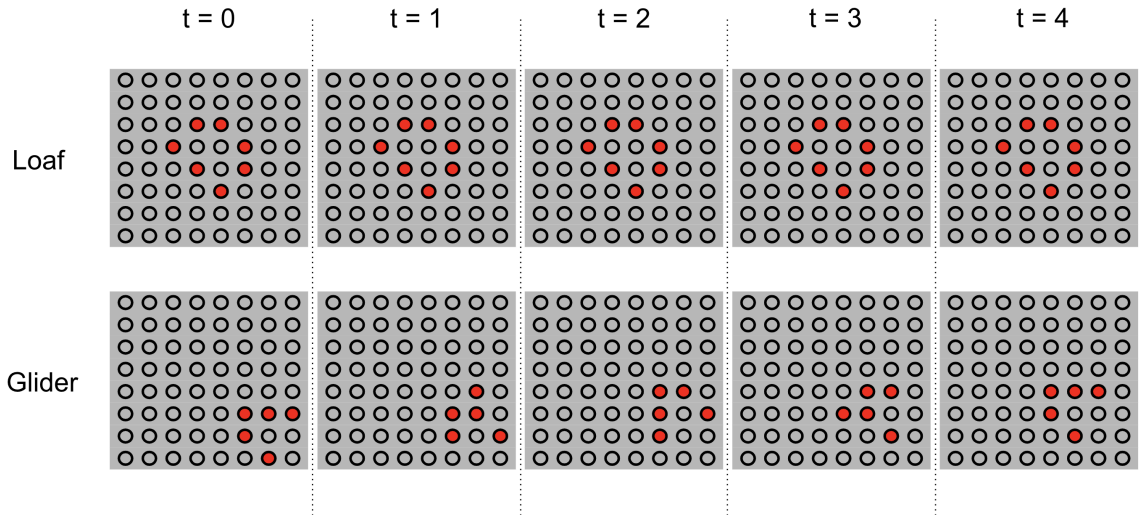


Figure 3-12: The Game of Life rules [2], applied four times to two initial configurations of an 8x8 game board. The Loaf configuration does not change in response to the rules, while the Glider configuration does.

The assignment uses the lab kit's LED display to visualize the game board. A living cell will correspond to an illuminated LED, and a dead cell will correspond

to an LED being off. Due to the LED display only having eight rows and thirty-two columns, which is much smaller than the size of a typical game board used for the Game of Life, the lab implementation will allow cells to "wrap around" the game board. That is, a cell in the top row of the board will have neighbors in the corresponding columns of the bottom row, and a cell in the leftmost column of the board will have neighbors in the corresponding columns of the rightmost row.

At a high level, the Game of Life can be implemented using an infinite loop that calls a function that updates the board on each iteration. Students are provided with a C implementation of this update function, and they must translate it into RISC-V assembly. This update function loops through every cell on the game board, each of which maps to a pixel on the LED display, and:

- Determines its current status.
- Determines how many of its neighbors are alive.
- Determines the new status of the cell accordingly, using the rules of the game.

Students are provided with a helper procedure named `getPixel` that returns the status of a cell given its location (in x and y coordinates) and a pointer to the game board array.

The update function uses a helper function named `checkNeighbors` that, given a pointer to the game board and a set of coordinates, returns how many of the corresponding cell's neighbors are alive. Students must implement this function using RISC-V assembly language. One key aspect of this `checkNeighbors` procedure is that it uses a helper procedure named `tallyNeighbors` to actually count and return the number of living neighbors that the given cell has. So, students must pay special attention to make sure they adhere to RISC-V calling conventions, which is a central theme of this week's content, when implementing this procedure and using the helper procedure.

After successfully implementing the `checkNeighbors` procedure using RISC-V assembly, students then complete an exercise pertaining to the `tallyNeighbors` helper

function used in their implementation. For this exercise, students are provided with a complete RISC-V implementation of `tallyNeighbors`, that will work as expected once it adheres to RISC-V calling convention. Students must add instructions to the procedure to make it follow calling conventions. The instructions they may add are limited to those that increment or decrement the stack pointer, push data onto the stack, or pop data from the stack. We only require students to handle the calling convention portion of this procedure so that they can focus on understanding those concepts in particular.

The final portion of this lab requires students to translate the board update function from C to assembly language, making sure to follow RISC-V calling conventions where applicable. After students complete this, their system will be able to run the Game of Life. Students are provided with a couple of initial game configurations and are encouraged to create their own.

3.2.10 Postlab 5: Quicksort

The fifth postlab assignment is another adaptation of an existing MIT 6.191 assignment. It is similar to Postlab 4, however, the sorting algorithm to be translated to RISC-V assembly is quicksort rather than bubble sort. We modified the assignment from the 6.191 version so that students implement and test the partitioning procedure separately from the main quicksort procedure that calls it. Implementing quicksort in assembly requires strategic use of the stack to make the procedure adhere to calling convention. Similar to Postlab 4, the lab instructs students to call an array visualization procedure from their quicksort procedure so that they can visualize the sorting algorithm on their LED display. However, in this assignment, students are not given the code to call this procedure, and instead must write it themselves, because procedure calls and calling conventions are a central focus of this week's material.

Chapter 4

RISC-V Debugging Tool

The low-level nature of assembly language programs makes it difficult to gain an understanding of their general behavior by just reading them. This results in assembly language programs often being difficult to debug when they don't work as expected, requiring a programmer to walk step-by-step through each of the individual instructions to find the root cause of the unexpected behavior. While it is possible for a programmer to manually keep track of the state of registers and memory as they step through the program, that approach is tedious, inconvenient, and prone to error.

All exercises and assignments in 6.190 are submitted via CAT-SOOP [11] [12] checker questions embedded in the course website. CAT-SOOP has many question types used to verify different types of submissions, from numerical answers to programs [12]. While preparing for the preliminary offering of the course, we had created a RISC-V assembly program checker question type that uses a derivation of RiscEmu, a RISC-V emulator written in Python [13], to run assembly programs submitted by students against sets of test cases.[1] This chapter discusses this extension of this RISC-V assembly program checker to include a convenient, user-friendly debugging tool. In addition to helping students debug their assembly programs, this tool can enhance their understanding of how individual RISC-V instructions interact with registers and memory.

[1]Future mentions of RiscEmu refers to our derivation of the package, rather than the package from the original source [13]. The derivation can be found at this link: `https://github.com/jodalyst/riscemu`

4.1 Features and Typical Use

The debugger is integrated with the existing RISC-V assembly program checker questions on the course website, and it is activated when a student submits a program including at least one breakpoint instruction. This breakpoint instruction, named `csbreak` (for CAT-SOOP-Breakpoint), is unique to this class and is not in the RV32I instruction set. If a student submits a program without any breakpoints, the question will behave as it did before: the debugger will not be invoked, the submission will receive a grade, and the results of each test case will be generated for the student to view.

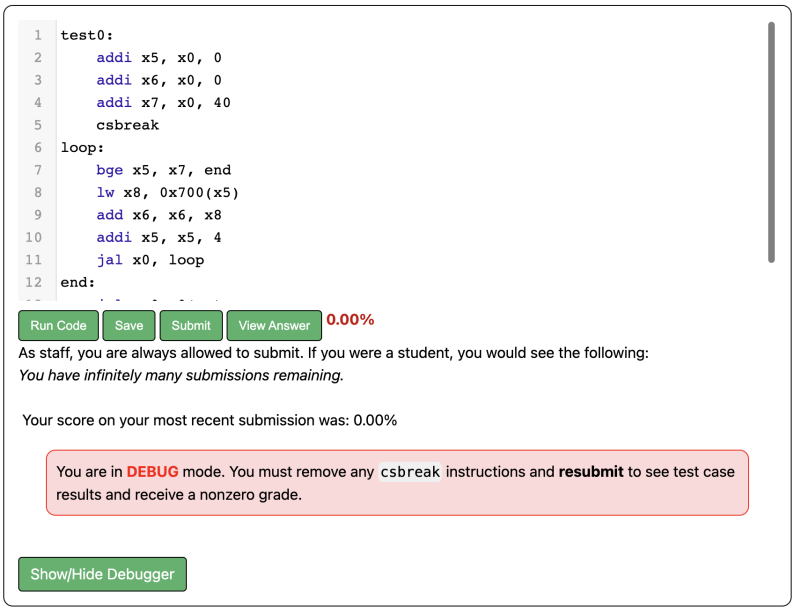


Figure 4-1: A RISC-V assembly checker question after a student submits code including a breakpoint instruction.

When the debugger is invoked, the submission is given a grade of 0% and an informative message will be displayed as shown in Figure 4-1. The question type assigns all submissions containing breakpoints a grade of 0%, because the breakpoint instruction is not an actual RISC-V instruction and would trigger assembler errors if the code was transferred to any other system. This issue is extremely pertinent to the lab and postlab assignments in 6.190, where the programs developed in the

online checkers must ultimately be run on a physical microcontroller. Lastly, a toggle button appears that, when clicked, will show or hide the debuggers.

Every test case in the checker has a separate debugger that is generated automatically when a student submits code with a breakpoint. Each debugger highlights the instruction that is about to be run and also shows the instructions immediately surrounding it. It also shows the stack and has collapsible containers for registers and memory. The debugger is initially paused at the beginning of the submitted program, and there are three buttons that the user can use to interact with the program:

- **Step**: Advances the debugger to the following instruction.
- **Continue**: Advances the debugger to the next breakpoint instruction.
- **Reset**: Returns the debugger to the beginning of the program.

The number of instructions executed since the beginning of program execution is also displayed. This number includes every instruction executed for the test case, not just those included in the submission. Figure 4-2 shows an example debugger at the beginning of the program. The registers and memory are collapsed to conserve space in the web browser, but they can be expanded if the user wants to view them. The stack is empty because this program has not yet put any data on it. This particular test case, as well as the majority of test cases, ran assembly instructions before invoking the submitted code, so the displayed instruction count is greater than zero.

The primary benefit of a debugger comes from observing how the values in registers and memory change as each instruction is executed. The "Registers" section of the debugger displays the values stored in the 32 RISC-V registers at the current point in the program, which is before the highlighted instruction is executed. Additionally, if the previous instruction had modified a register, that register will be highlighted for emphasis, as shown in Figure 4-3.

The "Memory" section of the debugger displays a section of memory. In order for a particular address (and its neighboring addresses) to be shown automatically,

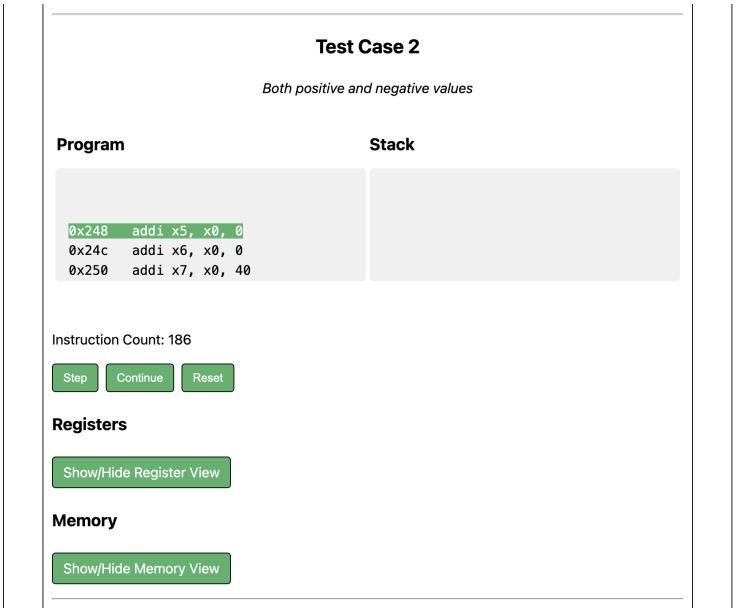


Figure 4-2: The initial state of a RISC-V debugger instance, prior to executing any of the program's instructions. The highlighted instruction is the one that is about to be executed.

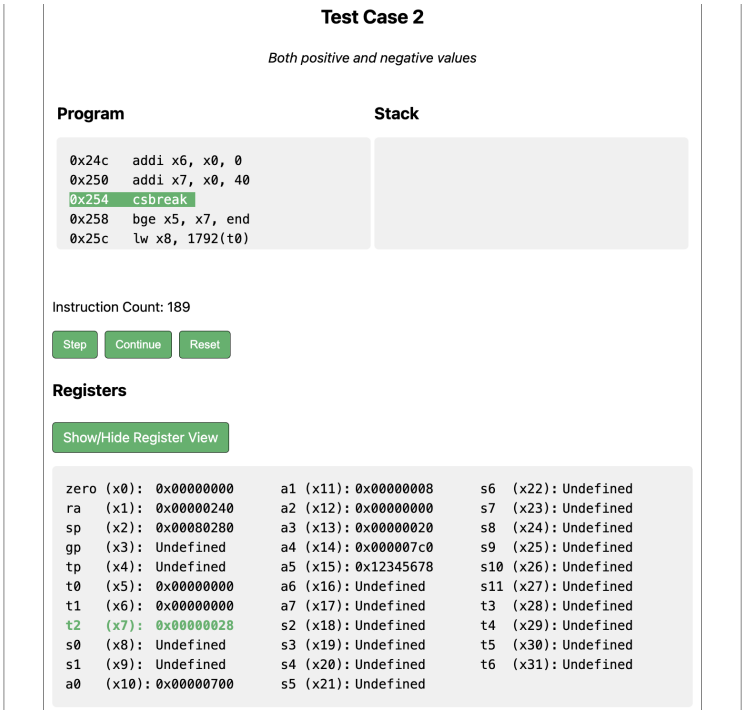


Figure 4-3: The RISC-V assembly debugger highlights the register modified by the most recently executed instruction. Here, the previous instruction, `addi x7, x0, 40` modified register `x7`, so it is highlighted in the "Registers" section of the debugger.

it must be modified by an instruction that accesses memory. Otherwise, the user can manually search for specific memory addresses using the search bar. Similar to registers, a memory address will be highlighted for emphasis if the previous instruction had written to it or if the user had searched for it. This behavior is displayed in Figure 4-4.

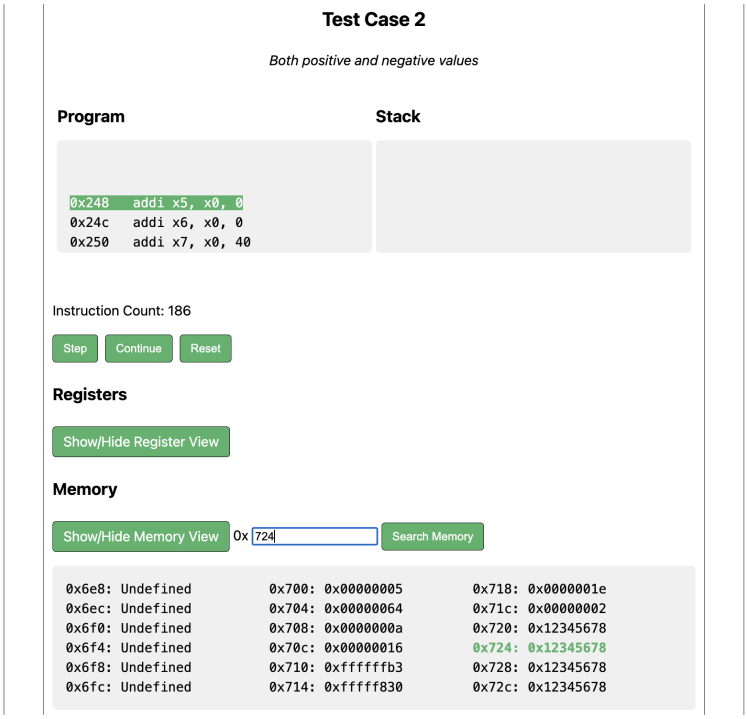


Figure 4-4: The RISC-V assembly debugger highlights either the memory address either modified by the instruction executed most recently or an address explicitly searched for by the user. Here, the user had searched for the memory address `0x724`, so the contents of that and the surrounding memory addresses are shown.

Students can also monitor the evolution of the stack throughout a program in the "Stack" window. To illustrate the last-in-first-out nature of the stack, the stack shown on the debugger grows as the stack pointer decrements and shrinks as it increments. The stack is displayed separately from the rest of memory to emphasize how it grows and shrinks. If the stack grows so that it no longer fits in the provided window, the user can scroll to view all of its contents. An example stack is shown in Figure 4-5.

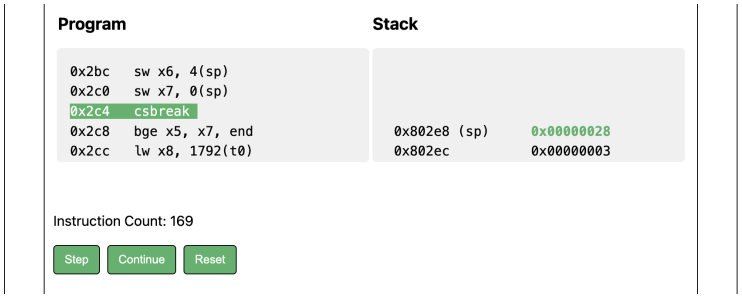


Figure 4-5: The RISC-V assembly debugger displays the stack section of memory separately, highlighting the element added to the stack by the previous instruction.

4.2 System Design and Implementation

The RISC-V debugging tool is integrated with the existing RISC-V assembly checker questions located on the assignment's webpage to reduce the overhead associated with using it compared to if we were to use a separate debugging tool. An overview of the new RISC-V assembly checker with debugging functionality is provided in Figure 4-6. If a student would like to use the debugger, they must simply submit a RISC-V assembly program that includes a breakpoint instruction. Once finished debugging, they must remove all breakpoints from their program and submit it again to receive a grade.

4.2.1 Preserving Existing RISC-V Code Submission Behavior

It is imperative that the original functionality of the online RISC-V checker questions be preserved after adding the debugging capabilities, as students still need to use these checkers to submit and receive grades for their RISC-V assembly language programs.

When a student submits their RISC-V assembly code on the website, a Python function is invoked to handle the submission. Originally, this function iterated through every test case, comparing the behavior of the student's submission agsinst the solution and assigning a grade accordingly. After adding debugging capabilities, this behavior comparison and grade assignment only occurs when a student submits a RISC-V assembly program that does not contain a breakpoint.

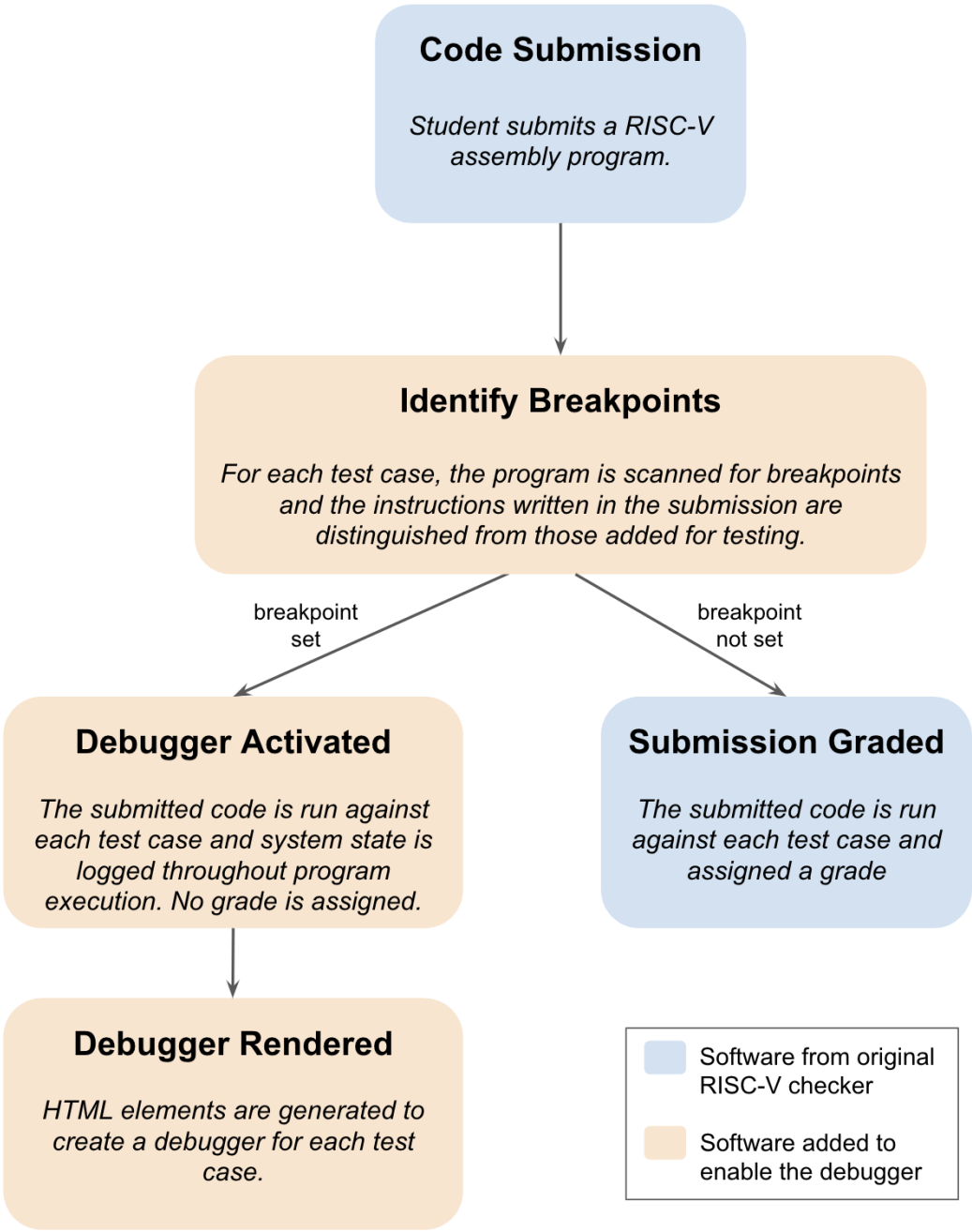


Figure 4-6: Overview of the RISC-V checker question with debugging capability. The debugger is activated when a student submits code including a breakpoint instruction named `csbreak`. Otherwise, the debugger will not be activated and the submitted code will receive a grade.

4.2.2 Identifying Breakpoints and Instructions to Log

The debugger is only activated when the submitted RISC-V assembly program contains a breakpoint, so each submission must first be scanned to identify the presence or absence of breakpoints in the program. We implemented this scan using RiscEmu in a sandboxed environment to parse every instruction in the program. The program counter is initialized to the first memory address containing an instruction, and it is incremented until has gone through the entirety of instruction memory.

Because this breakpoint instruction needs to be parsed and used like any other RISC-V instruction, we added a custom instruction named `csbreak` to the RiscEmu instruction set. It behavior is comparable to that of a no-operation instruction – the checker software just uses its presence or absence to determine whether or not the debugger should be activated. If a `csbreak` instruction is encountered during this preliminary scan of the program, a Boolean flag is set to indicate that the debugger must be activated. Otherwise, the original behavior of the RISC-V assembly checker is carried out and the submission will receive a grade.

When a student's submission is run, it is integrated with additional test code that may include solutions to other procedures they may have to write to complete the assignment. For example, an assignment could instruct students to write a procedure to calculate the sum of squares of values in an array, and this sum of squares procedure must call a multiplication procedure that is defined and implemented in the test code. It would be undesirable for the debugger to reveal the implementation of this multiplication procedure in the case that students need to implement it at some point during the class, so the debugger may only reveal instructions that are implemented by the student. This requires the checker to determine which instructions may be logged for debugging purposes.

During the checker's initial scan for breakpoints, which is run in a sandboxed environment, a Python dictionary mapping each memory address encountered during the scan to whether or not the behavior of the corresponding instruction should be logged for debugging purposes is created. To determine whether or not an instruction

and its behavior should be logged, the code submitted by the student must be distinguished from the test code. We added two more custom instructions to RiscEmu, named `startlog` and `stoplog`, that simply mark the beginning and end of the student's code submission, as shown in Figure 4-7. These instructions do not modify the state of the system and are each implemented as a no-operation instruction for the RiscEmu emulator, just like the `csbreak` instruction used to set breakpoints. We assume that RiscEmu stores instructions in memory contiguously, so that the instructions encountered between `startlog` and `stoplog` are those written by the student. Additionally, this initial scan maps each program counter to the instruction stored at the corresponding memory address. Only instructions that should be logged for debugging purposes, as described previously, are included in this mapping.

4.2.3 Logging Program State

After the initial program scan, the RISC-V assembly checker's submission function has the information necessary to determine whether or not the debugger should be activated. If the debugger is activated, it will run the program submitted by the student against each test case in a sandboxed environment using RiscEmu. Additionally, during program execution, RiscEmu is also used to gather and log the state of the emulated system before specific instructions are run. This program state is stored in a Python dictionary that is ultimately converted to a JSON object to be used in the front-end JavaScript code that renders the debugger on the webpage. Examples of entries in this JSON object are shown in Figures 4-8, 4-9, 4-10, and 4-11.

Using RiscEmu extensively, the program creates and configures an instance of a RV32I processor, loads the program, consisting of the student submission and test code as shown in Figure 4-7, into its memory. Then, the emulated processor executes the program. For each instruction, the program first uses the current program counter to determine whether or not the current system state should be logged. If it should not be logged, the emulated processor simply executes the instruction and proceeds to the next one. If the instruction should be logged, the current state of the system must be saved.

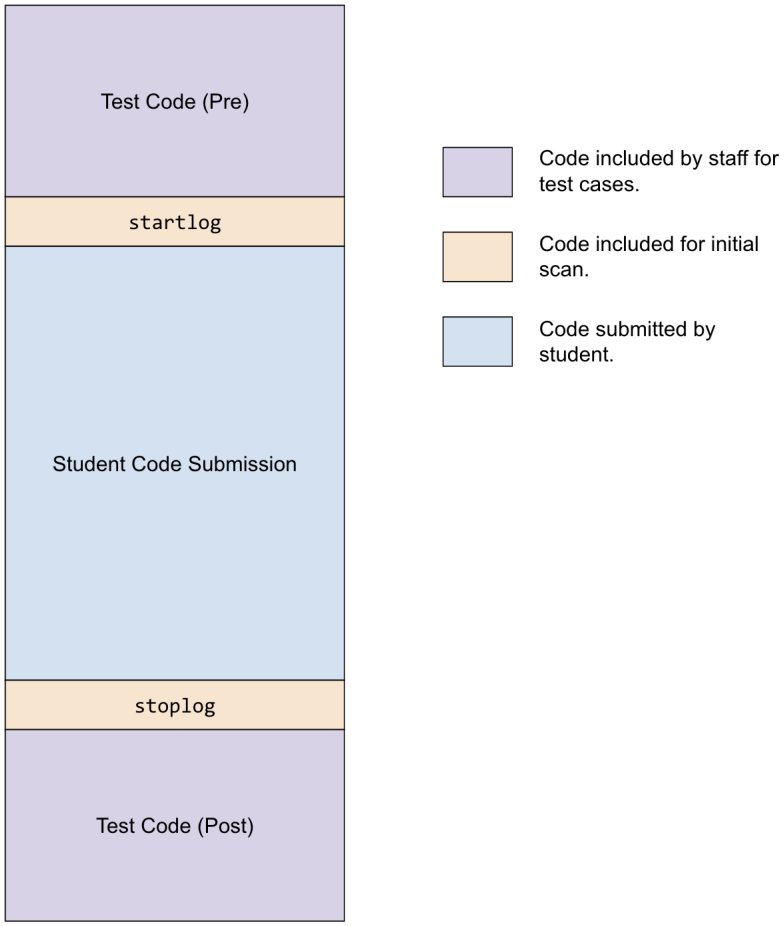


Figure 4-7: The structure of the assembly code used for the initial scan to identify breakpoints and distinguish the assembly instructions submitted by the student from the assembly instructions injected by course staff.

We use a Python dictionary to store the state of the system at each necessary point in the program before executing the instruction. The state consists of:

- `inscount`: The number of instructions that have been executed since the beginning of the program, including those not logged for debugging purposes.
- `pc`: The current value of the program counter.
- `ins`: The instruction about to be executed.
- `final_state`: Indicates whether or not this state is the final one of the program.
- `updated_register`: The register updated by the previous instruction.
- `updated_addr`: The memory address updated by the previous instruction.
- `new_reg_vals`: The updated register value. If the previous instruction was not logged, or the current instruction is a breakpoint, it includes all 32 register values.
- `new_mem_vals`: The updated memory value. If the previous instruction was not logged, or the current instruction is a breakpoint, it includes all memory values of interest.
- `stack_top`: The value of the stack pointer.
- `stack_bottom`: The memory address indicating the bottom of the stack.
- `stack_data`: The contents of the stack.

Each "current state" Python dictionary, like the ones shown in Figures 4-8, 4-9, 4-10, and 4-11, is then added to an "overall state" Python dictionary, and its key is the number of instructions that have been logged since the beginning of the program. This data structure is used extensively when rendering the front-end of the debugger.

```
"0": {
        "pc": 460,
        "ins": "addi a0, x0, 1",
        "inscount": 1,
        "final_state": false,
        "updated_register": null,
        "updated_addr": null,
        "new_mem_vals": {},
        "new_reg_vals": {
          "zero": 0,
          "ra": "Undefined",
          "sp": 524792,
          "gp": "Undefined",
          "tp": "Undefined",
          "t0": "Undefined",
          "t1": "Undefined",
          "t2": "Undefined",
          "s0": "Undefined",
          "s1": "Undefined",
          "a0": "Undefined",
          "a1": "Undefined",
          "a2": "Undefined",
          "a3": "Undefined",
          "a4": "Undefined",
          "a5": "Undefined",
          "a6": "Undefined",
          "a7": "Undefined",
          "s2": "Undefined",
          "s3": "Undefined",
          "s4": "Undefined",
          "s5": "Undefined",
          "s6": "Undefined",
          "s7": "Undefined",
          "s8": "Undefined",
          "s9": "Undefined",
          "s10": "Undefined",
          "s11": "Undefined",
          "t3": "Undefined",
          "t4": "Undefined",
          "t5": "Undefined",
          "t6": "Undefined"
        },
        "stack_top": 524792,
        "stack_bottom": 524792,
        "stack_data": {}
    },

```


Figure 4-8: An example of the first entry in a program state JSON object. It records the state of the system before the first student-written instruction is executed.

```
"1": {
        "pc": 464,
        "ins": "addi sp, sp, -4",
        "inscount": 2,
        "final_state": false,
        "updated_register": "a0",
        "updated_addr": null,
        "new_reg_vals": {
          "a0": 1
        },
        "new_mem_vals": {},
        "stack_top": 524792,
        "stack_bottom": 524792,
        "stack_data": {}
    },

```

Figure 4-9: An example entry in the program state JSON object, where the previous instruction had modified register `a0`. This information is communicated via the `updated_register` and `new_reg_vals` fields in the entry. Note that this is the state of the system before the instruction included in this entry, `addi sp, sp, -4`, is executed.

```
"3": {
        "pc": 472,
        "ins": "lui t0, 6",
        "inscount": 4,
        "final_state": false,
        "updated_register": null,
        "updated_addr": 524788,
        "new_reg_vals": {},
        "new_mem_vals": {
          "524788": 1
        },
        "stack_top": 524788,
        "stack_bottom": 524792,
        "stack_data": {
          "524788": 1
        }
    },


```

Figure 4-10: An example entry in the program state JSON object, where the previous instruction had written the value `1` to the memory address `524788`. This is communicated through the `updated_addr` and `new_mem_vals` fields in the entry. This memory address modified was also a part of the stack, which is determined using the fields `stack_top` and `stack_bottom`, so the `stack_data` field is also updated to reflect this memory write.

```
"5": {
        "pc": 480,
        "ins": "csbreak ",
        "inscount": 6,
        "final_state": false,
        "updated_register": null,
        "updated_addr": 24576,
        "new_mem_vals": {
          "24576": 1,
          "524788": 1
        },
        "new_reg_vals": {
          "zero": 0,
          "ra": "Undefined",
          "sp": 524788,
          "gp": "Undefined",
          "tp": "Undefined",
          "t0": 24576,
          "t1": "Undefined",
          "t2": "Undefined",
          "s0": "Undefined",
          "s1": "Undefined",
          "a0": 1,
          "a1": "Undefined",
          "a2": "Undefined",
          "a3": "Undefined",
          "a4": "Undefined",
          "a5": "Undefined",
          "a6": "Undefined",
          "a7": "Undefined",
          "s2": "Undefined",
          "s3": "Undefined",
          "s4": "Undefined",
          "s5": "Undefined",
          "s6": "Undefined",
          "s7": "Undefined",
          "s8": "Undefined",
          "s9": "Undefined",
          "s10": "Undefined",
          "s11": "Undefined",
          "t3": "Undefined",
          "t4": "Undefined",
          "t5": "Undefined",
          "t6": "Undefined"
        },
        "stack_top": 524788,
        "stack_bottom": 524792,
        "stack_data": {
          "524788": 1
        }
    },

```


Figure 4-11: An example entry in the program state JSON object, where the current instruction is a `csbreak` instruction, indicating that the program has reached a breakpoint. Due to the way the state of the system is rendered on the front-end of the debugger, the entire state of the system must be logged at each breakpooint.

4.2.4 Debugger Front-end

The central submission Python function included in the checker software will conditionally generate certain HTML elements depending on whether or not the debugger is activated. A separate debugger is instantiated for every individual test case, so that interactions with the debugger for one test case does not impact the status of any of the other debuggers. After initially being rendered, these HTML elements are each controlled by JavaScript listener functions that respond to user interaction in the form of button clicks. There are four buttons that a user can interact with: Step, Continue, Reset, and Memory Search.

First, JavaScript objects are generated so that information about the program's state can be used for the debugger's front end. The first object is simply the dictionary containing the state of the system, as defined in the previous section, throughout program execution.

The second object contains information about the system state that should be rendered on the debugger at a single point in time, as shown in Figure 4-12. It has the following properties:

- `ins`: The number of instructions simulated by the debugger.
- `done`: Whether or not the debugger has reached the end of the program.
- `reg_state`: A mapping between the 32 registers and their values at the current point in time.
- `mem_state`: A mapping between the memory addresses of interest and their associated values at the current point in time.
- `min_pc`: The lowest memory address containing an instruction from the student's code submission.
- `max_pc`: The highest memory address containing an instruction from the student's code submission.

```
{
  "ins": 1,
  "done": false,
  "reg_state": {
    "zero": 0,
    "ra": "Undefined",
    "sp": 524800,
    "gp": "Undefined",
    "tp": "Undefined",
    "t0": "Undefined",
    "t1": "Undefined",
    "t2": "Undefined",
    "s0": "Undefined",
    "s1": "Undefined",
    "a0": 0,
    "a1": "Undefined",
    "a2": "Undefined",
    "a3": "Undefined",
    "a4": "Undefined",
    "a5": "Undefined",
    "a6": "Undefined",
    "a7": "Undefined",
    "s2": "Undefined",
    "s3": "Undefined",
    "s4": "Undefined",
    "s5": "Undefined",
    "s6": "Undefined",
    "s7": "Undefined",
    "s8": "Undefined",
    "s9": "Undefined",
    "s10": "Undefined",
    "s11": "Undefined",
    "t3": "Undefined",
    "t4": "Undefined",
    "t5": "Undefined",
    "t6": "Undefined"
  },
  "mem_state": {},
  "min_pc": 460,
  "max_pc": 492,
  "pc_to_inst": {
    "460": "addi a0, x0, 0",
    "464": "addi a1, x0, 0",
    "468": "addi a2, x0, 40",
    "472": "csbreak ",
    "476": "bge a0, a2, end",
    "480": "lw a4, 1792(a0)",
    "484": "add a1, a1, x8",
    "488": "addi a0, a0, 4",
    "492": "jal x0, loop"
  },
  "addr_viewing": null
}

```


Figure 4-12: An example of the JavaScript object containing data about the current state of the system that is used to render the front-end of the debugger at that point in time.

- `pc_to_inst`: A mapping between each program counter and its corresponding instruction, for every instruction in the program submitted by the student.
- `addr_viewing`: The memory address being highlighted at the current point in time.

The third object contains variables that refer to each HTML element that will either invoke or be updated by JavaScript event listeners. These HTML elements are: a step button, a reset button, a continue button, a container to display the registers and their values, a container to display memory addresses and their values, a memory search bar, a memory search button, a memory search error message container, a container to display the stack, a container to highlight the current instruction, and an instruction count container. These three objects are then passed into a JavaScript function that will provide them with the necessary functionality.

There are four main JavaScript functions that are invoked by user interaction – step, continue, reset, and memory search. The step operation simulates the execution of a single instruction, the operation function simulates the execution of multiple instructions until a breakpoint is encountered, the reset operation resets the state of the debugger to be at the beginning of the program, and the memory search operation allows the user to view the contents of memory at certain addresses.

Chapter 5

Discussion

Introduction to Low-level Programming in C and Assembly (6.190) has been offered by the MIT Department of Electrical Engineering and Computer Science for five consecutive academic quarters, beginning in the second half of the spring 2022 semester. For the rest of this chapter, we will refer to each academic quarter as follows:

- SP22 H2: The second half of the spring 2022 semester.
- FA22 H1: The first half of the fall 2022 semester.
- FA22 H2: The second half of the fall 2022 semester.
- SP23 H1: The first half of the spring 2023 semester.
- SP23 H2: The second half of the spring 2023 semester.

Throughout these offerings, we have collected data and information from in-class surveys and observation that can inform the effectiveness and impact of the class.

5.1 Enrollment

One key metric of a class is its enrollment, because it is an indicator of overall student interest in the class as well as the amount of resources needed for its successful execution. Approximately 622 students have taken 6.190 across the first five offerings

of the 6.190, as shown in Table 5.1. This is not the number of unique students who have taken the class, as students have repeated the class under circumstances such as receiving a non-passing grade. We consider enrollment to be the number of students who have completed the class as demonstrated by taking the final exam. As of writing, the final exam had not yet been administered for the SP23 H2 offering of the class, so we define that term's enrollment as the number of students. taking the class for-credit, who are eligible to submit a MIT-administered subject evaluation for the class.

Offering	SP22 H2	FA22 H1	FA22 H2	SP23 H1	SP23 H2	Total
Enrollment	8	89	199	176	150	622

Table 5.1: Student enrollment in Introduction to Low-level Programming in C and Assembly during each of the first five offerings of the class. Here, we define enrollment as the number of students who completed the final exam, with the exception of the SP23 H2 offering of the class because the exam had not been administered as of writing. Instead, the enrollment for the SP23 H2 offering of the class is defined as the number of students taking the course for credit who are eligible to complete a MIT subject evaluation for the class.

The very first offering of the class, during SP22 H2, had just eight students. Enrollment for that quarter was low for three main reasons. First, it was the initial, pilot offering of the class, so students may not have heard about it or planned to take it. Additionally, only students entering MIT in fall 2022, the following semester, would need to take 6.190 to fulfill the new requirements for the Course 6-2 or Course 6-3 majors [1]. Students who were enrolled at MIT before the fall 2022 semester could choose between completing the old major requirements, that did not include 6.190, and the new ones. Finally, 6.190 was not an enforced corequisite for 6.191 until the spring 2023 semester, one year later [1]. Consequently, the enrollment figure for the second half of the spring 2022 semester can be treated as an outlier, as the class was offered under various conditions that were unique to that academic quarter and are no longer applicable.

Student enrollment in the class quickly grew. It increased to 89 students in the FA22 H1 offering and reached a maximum of 199 students during the FA22 H2 offering

of the class. This increase between the first and second halves of the semester could, in part, be due to students needing to complete the prerequisite class, 6.100A, prior to enrolling in 6.190. Because 6.100A is offered during the first half of the semester, students could conveniently take 6.100A during the first half and 6.190 during the second. Additionally, word may have simply started to spread about the class itself.

Enrollment then declined slightly, decreasing to 176 students in the SP23 H1 offering and 150 students in the SP23 H2 offering. The spring 2023 semester was the first where 6.190 was an enforced corequisite for 6.191, so many students took 6.190 during the first half so that they could enroll in 6.191.

If we consider only the four offerings of the class throughout the 2022-2023 academic year, this class was offered to 614 students. On average, there were 153.5 students per offering. In the future, 6.190 will only be offered twice per year, during both halves of the spring semester. Under this planned decrease in annual offerings of the class, if the number of students who need to take the class annually remains the same, at 614, the average enrollment would double to 312 students per offering. The maximum number of students taking the class during a given offering thus far has been 199, so this potential significant increase in enrollment must be considered when allocating resources, such as staff, toward the 6.190 in the future.

5.2 Time Commitment

Introduction to Low-level Programming in C and Assembly (6.190) is a six-unit class that students take during one half of a given semester, so the expectation set by MIT is that a student will need to dedicate 12 hours per week to the class [14]. We designed the class with this expectation in mind. During the first three offerings of the course (the SP22 H2, FA22 H1, and FA22 H2 offerings), we administered a weekly survey, prompting students to report how much time they spent on various components of the course. This data allows us to understand how well the actual weekly time commitment associated with the class aligns with the expected time commitment set by its units. Additionally, administering this survey on a weekly

basis allows us to identify weeks, and even particular assignments, that require a different amount of time than expected. It is important to emphasize that this data is self-reported by the students, so it may not be a completely accurate representation of the actual amount of time students spent on the class. However, this data does provide us with an idea of how much time students devoted toward completing the class.

A typical week in 6.190 consists of five main components: a lecture, a recitation, a lab, a postlab, and exercises. Lecture and recitation are each scheduled for an hour and a half every week, and our calculations assume that the average student is in attendance at both of these class sessions weekly.

On average, the self-reported weekly time commitment for 6.190 is 12.08 hours, as shown in Table 5.2, which is very close to the expected weekly time commitment of 12 hours – it is only about five minutes more than expected. This overall weekly average is broken down into 3 hours spent in lecture and recitation, 4.02 hours on exercises, 3.04 hours on the lab, and 2.01 hours on the postlab. Week 5, which is dedicated to RISC-V calling conventions and the stack, had the largest time commitment, with students, on average, dedicating 13.14 hours to it. Week 1, which focuses on introducing students to C and the idea of manipulating individual bits in memory, had the smallest time commitment, with students spending an average of 10.26 hours on the class that week.

5.3 Effectiveness of Course Components

During the final week of each of the first three offerings of the course, we administered a survey prompting students to indicate their level of agreement with various statements on a scale of 0 through 4. A rating of 0 means that the respondent strongly disagreed with the statement, a rating of 2 means that the respondent was neutral toward the statement, and a rating of 4 means that the respondent strongly agreed with the statement. The first set of statements pertained to how students felt that various course components contributed to their learning. We can use this feedback to eval-

Week	1	2	3	4	5	Overall (Average)
Lecture	1.50	1.50	1.50	1.50	1.50	1.50
Recitation	1.50	1.50	1.50	1.50	1.50	1.50
Exercises	2.90	4.11	4.54	4.32	4.24	4.02
Lab	2.63	3.44	3.04	2.88	3.23	3.04
Postlab	1.73	2.09	1.80	1.77	2.67	2.01
Total (Hours)	10.26	12.64	12.39	11.97	13.14	12.08

Table 5.2: The average self-reported amount of time (in hours) that a student would spend on Introduction to Low-level Programming in C and Assembly weekly. The overall average is provided, as well as averages for each week and course component. The data pertaining to the time spent on exercises, labs, and postlabs was collected from weekly surveys administered to students during the first three offerings of the course. We assume that the average student attends the 1.5-hour lecture and 1.5-hour recitation each week.

uate the effectiveness of the different aspects of the course and make improvements accordingly. The statements provided in the survey and their average responses are shown in Table 5.3. Additionally, histograms presenting the frequency distributions of the responses to each statement are shown in Figures 5-1 and 5-2.

Students found the labs to be the aspect of the class most helpful to their understanding of a given week's material, rating their agreement with the statement, "The labs helped me better understand the material presented in lecture and recitation," 3.65/4 on average. The matching statements about hands-on components of the class, such as the postlabs and exercises, that required students to practice applying the concepts learned in lecture and recitation, were also generally agreed with. However, students found more the more traditional class session components of the course least helpful, rating their agreement with the statements, "The lectures were helpful to my learning," and "The recitations helped me better understand the material presented in lecture" as 2.79/4 and 2.82/4, respectively. While both of these average responses indicate agreement, they indicate less agreement than those to affirmative statements about other aspects of the course.

On average, students agreed with the statement that they were provided with enough exercises to gain comfort with the course material, however, they also were

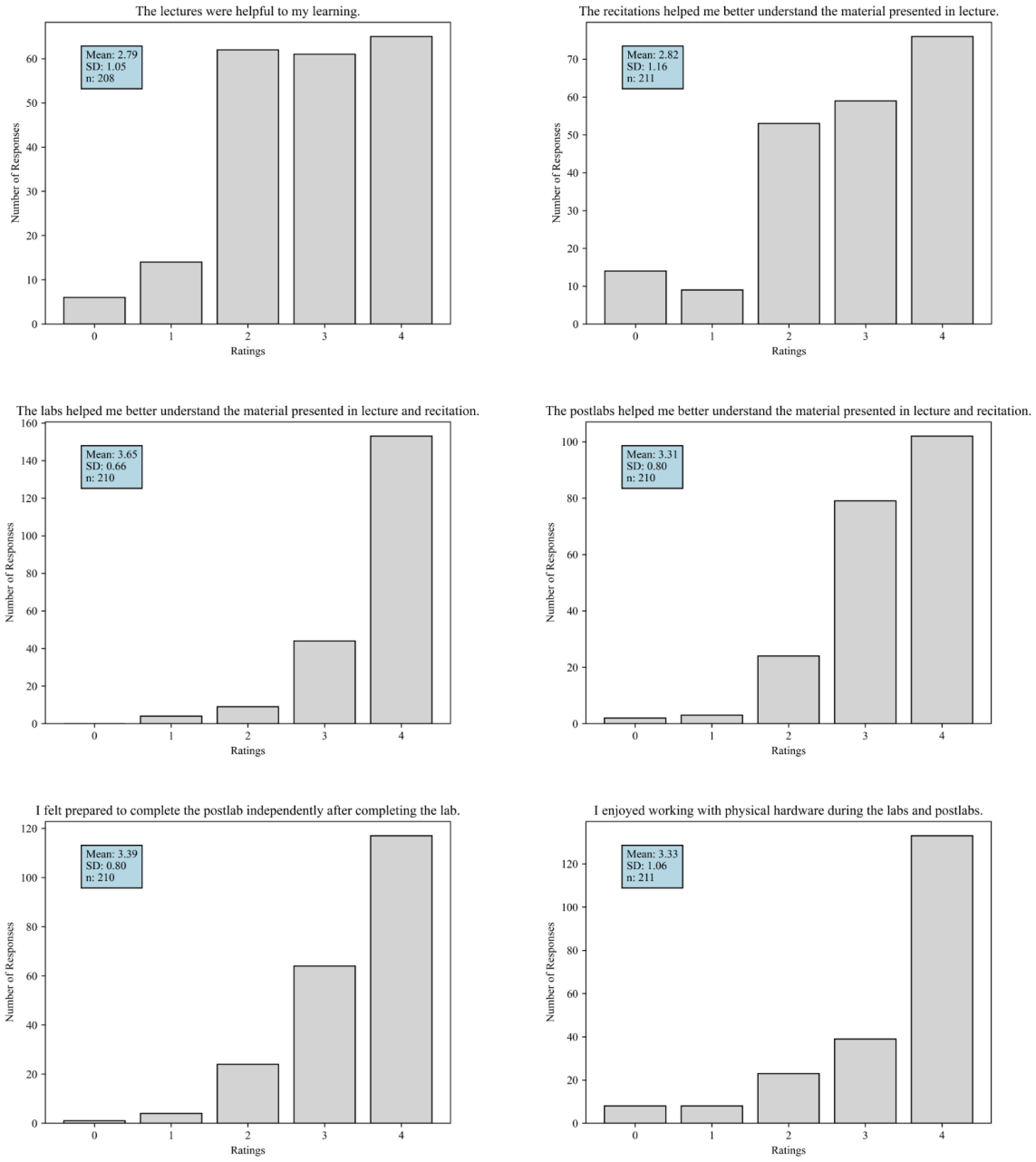


Figure 5-1: Histograms displaying the frequencies of student responses to first six final survey prompts pertaining to their experience with the different components of the class. A response of 0 indicates that the respondent strongly disagreed with the statement, a 2 indicates that the respondent was neutral towards the statement, and a 4 indicates that the respondent strongly agreed with the statement.

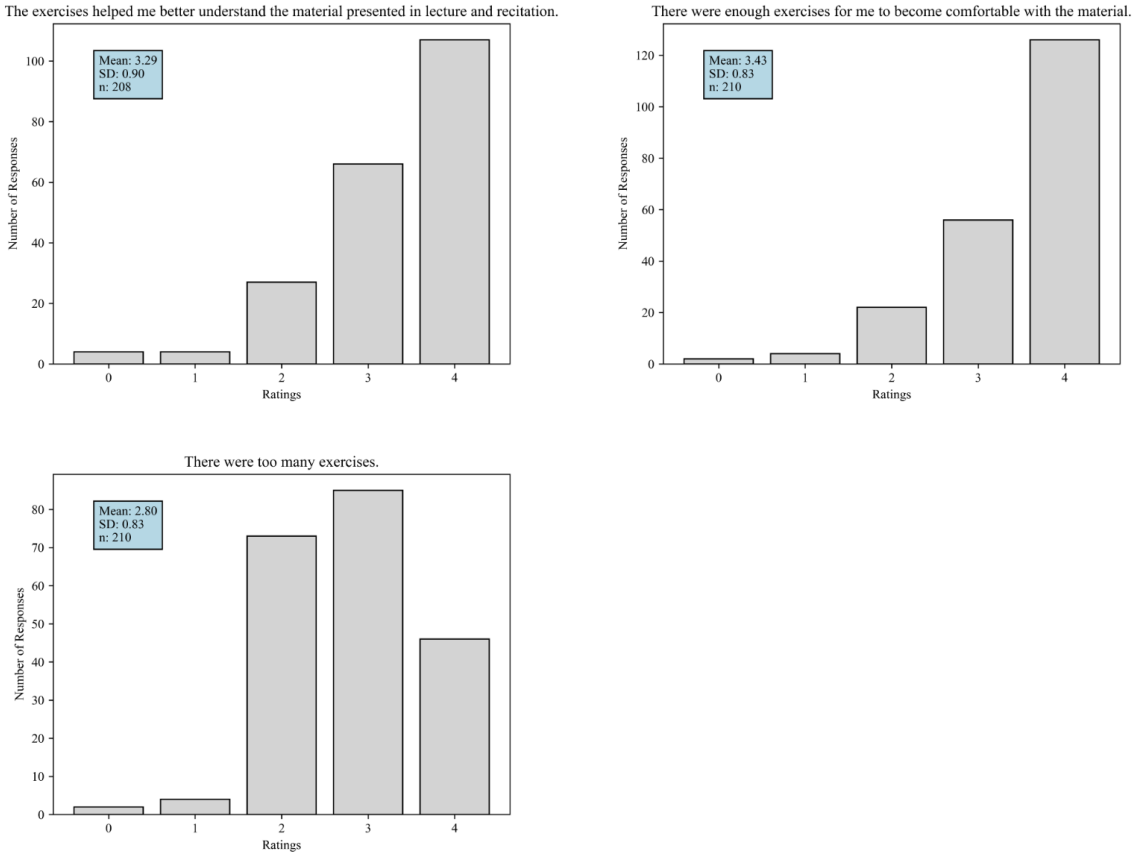


Figure 5-2: Histograms displaying the frequencies of student responses to the seventh through ninth final survey prompts pertaining to their experience with the different components of the class. A response of 0 indicates that the respondent strongly disagreed with the statement, a 2 indicates that the respondent was neutral towards the statement, and a 4 indicates that the respondent strongly agreed with the statement.

Prompt	Average	Standard Deviation	Respondents
The lectures were helpful to my learning.	2.79	1.05	208
The recitations helped me better understand the material presented in lecture.	2.82	1.16	211
The labs helped me better understand the material presented in lecture and recitation.	3.65	0.66	210
The postlabs helped me better understand the material presented in lecture and recitation.	3.31	0.80	210
I felt prepared to complete the postlab independently after completing the lab.	3.39	0.80	210
I enjoyed working with physical hardware during the labs and postlabs.	3.33	1.06	211
The exercises helped me better understand the material presented in lecture and recitation.	3.29	0.90	208
There were enough exercises for me to become comfortable with the material.	3.43	0.83	210
There were too many exercises.	2.80	0.83	210

Table 5.3: Student responses to a series of prompts pertaining to their experience with the different components of the class. This data was collected from a survey administered to students at the end of each of the first three offerings of the class. A response of 0 indicates that the respondent strongly disagreed with the statement, a 2 indicates that the respondent was neutral towards the statement, and a 4 indicates that the respondent strongly agreed with the statement. The responses shown in the table imply that students, on average, found the hands-on portions of the class, such as laboratory assignments and exercises, more useful to their overall understanding of the course material than the other portions, such as lecture and recitation.

in slight agreement with the statement that there were too many exercises. We responded to this feedback by evaluating the necessity of each of the exercises that were assigned to students and making adjustments to them accordingly. For example, Week 3 initially contained an exercise that required students to implement a nontrivial function in Python. We found that this exercise wasn't reinforcing any concepts

covered in the class and removed it.

Students reporting that they found the labs and postlabs to be helpful, and that they enjoyed working with hardware during those assignments, particularly validates the decision to incorporate an embedded systems laboratory component into the curriculum of the class.

5.4 Learning Objectives

The final class survey discussed in the previous section included a second set of statements that primarily asked about the student's perceived understanding of various course topics. These responses allow us to gain insight to what students felt they had learned from taking the class and compare that to what we would have expected them to learn. The statements and their average responses, on a scale from 0 to 4 as defined previously, are displayed in Table 5.4. Additionally, histograms presenting the frequency distributions of the responses to each statement are shown in Figures 5-3, 5-4, and 5-5.

According to these survey responses, students most strongly agreed that they understood how C pointers are translated into RISC-V assembly instructions and how to translate between different numerical representations. Students agreed least with the statements that they understood how floating point numbers and C structures are represented in memory.

One promising insight gained from the responses to these survey prompts is that students overall felt as though they had some level of understanding of each of the topics mentioned in the survey. The average student response to every statement, which affirmed understanding of a particular topics or action, was greater than the neutral response of 2. However, it is possible that these average figures are inflated due to this data being self-reported.

One goal of this class is for other, more advanced, classes offered by the MIT Department of Electrical Engineering and Computer Science to be able to use the C programming language without needing to use class time to introduce it to students.

Prompt	Average	Standard Deviation	Respondents
I am able to write programs using the C programming language	3.44	0.67	209
I understand how data is represented in memory	3.39	0.72	209
I understand how to access data stored in memory using C	3.41	0.73	209
I feel comfortable using pointers in C	3.24	0.77	209
I can translate C programs into RISC-V assembly code	3.50	0.61	208
I understand how C pointers translate into RISC-V assembly instructions.	3.67	0.52	6
I understand how to use RISC-V assembly instructions to access and modify memory.	3.49	0.64	200
I understand how different areas of memory are used throughout a program	3.19	0.81	205
I am comfortable translating between decimal, binary, and hexadecimal encoding of numbers.	3.60	0.64	207
I am comfortable using two's complement encoding of numbers.	3.40	0.79	206
I am comfortable using floating point representation for numbers.	2.62	1.16	207
I understand how characters and strings are represented in memory.	3.40	0.73	207
I understand how structs are represented in memory.	2.87	0.97	199

Table 5.4: Student responses to a series of prompts pertaining to their perceived understanding of main course topics and comfort with using them. A response of 0 indicates that the respondent strongly disagreed with the statement, a 2 indicates that the respondent was neutral towards the statement, and a 4 indicates that the respondent strongly agreed with the statement. These responses were gathered via a survey administered at the end of each of the first three offerings of the course.

Th students who responded to the survey, on average, agreed with the statement that they were able to write programs using the C programming language, and they responded favorably to the other C-related prompts. This indicates that students, on average, comfortable with the C programming language, meaning that they would

potentially be able to use it in future coursework without needing to dedicate a significant amount of time to becoming familiar with it.

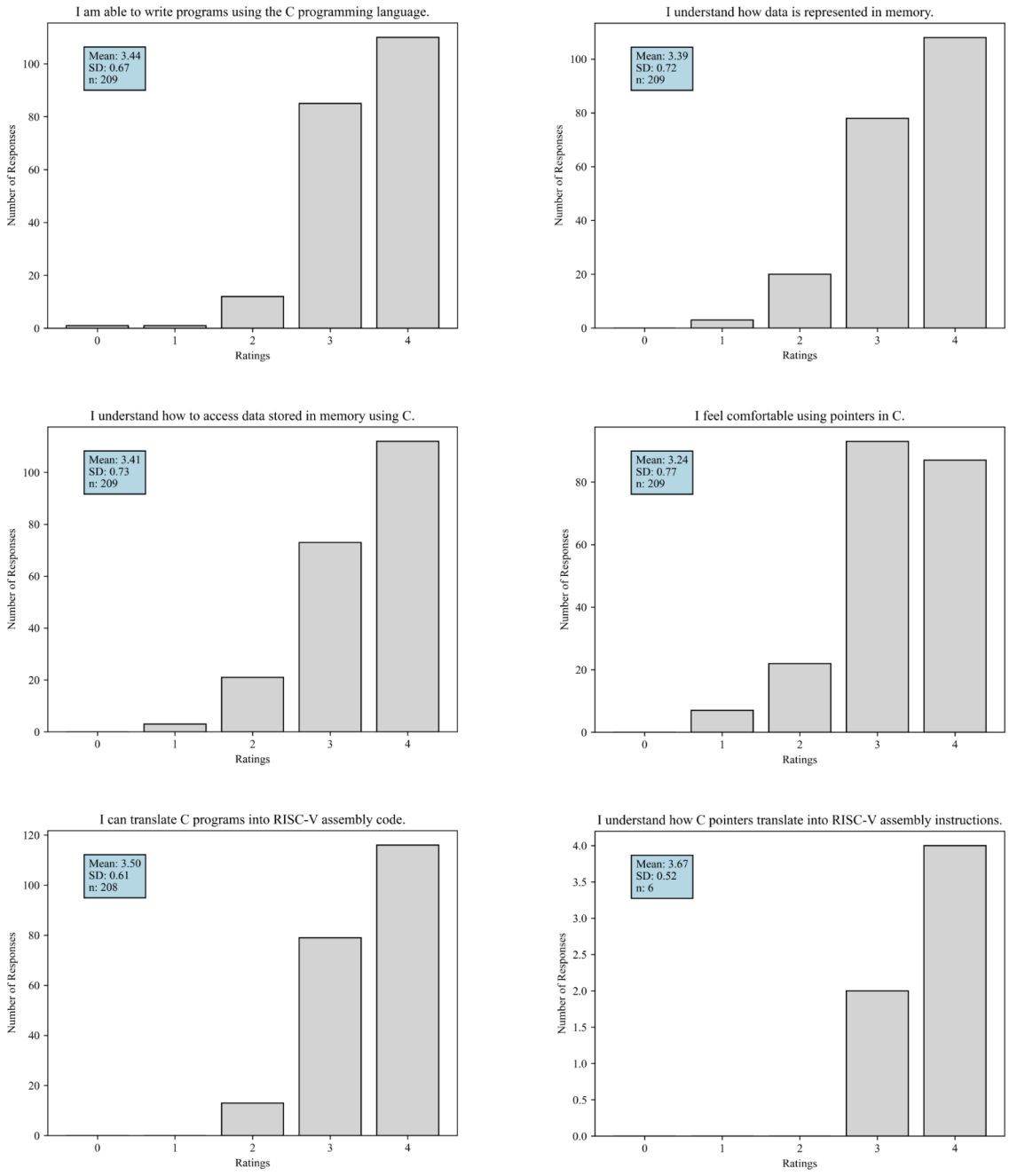


Figure 5-3: Histograms displaying the frequencies of student responses to the first six final survey prompts pertaining to their perceived understanding of main course topics and comfort with using them. A response of 0 indicates that the respondent strongly disagreed with the statement, a 2 indicates that the respondent was neutral towards the statement, and a 4 indicates that the respondent strongly agreed with the statement.

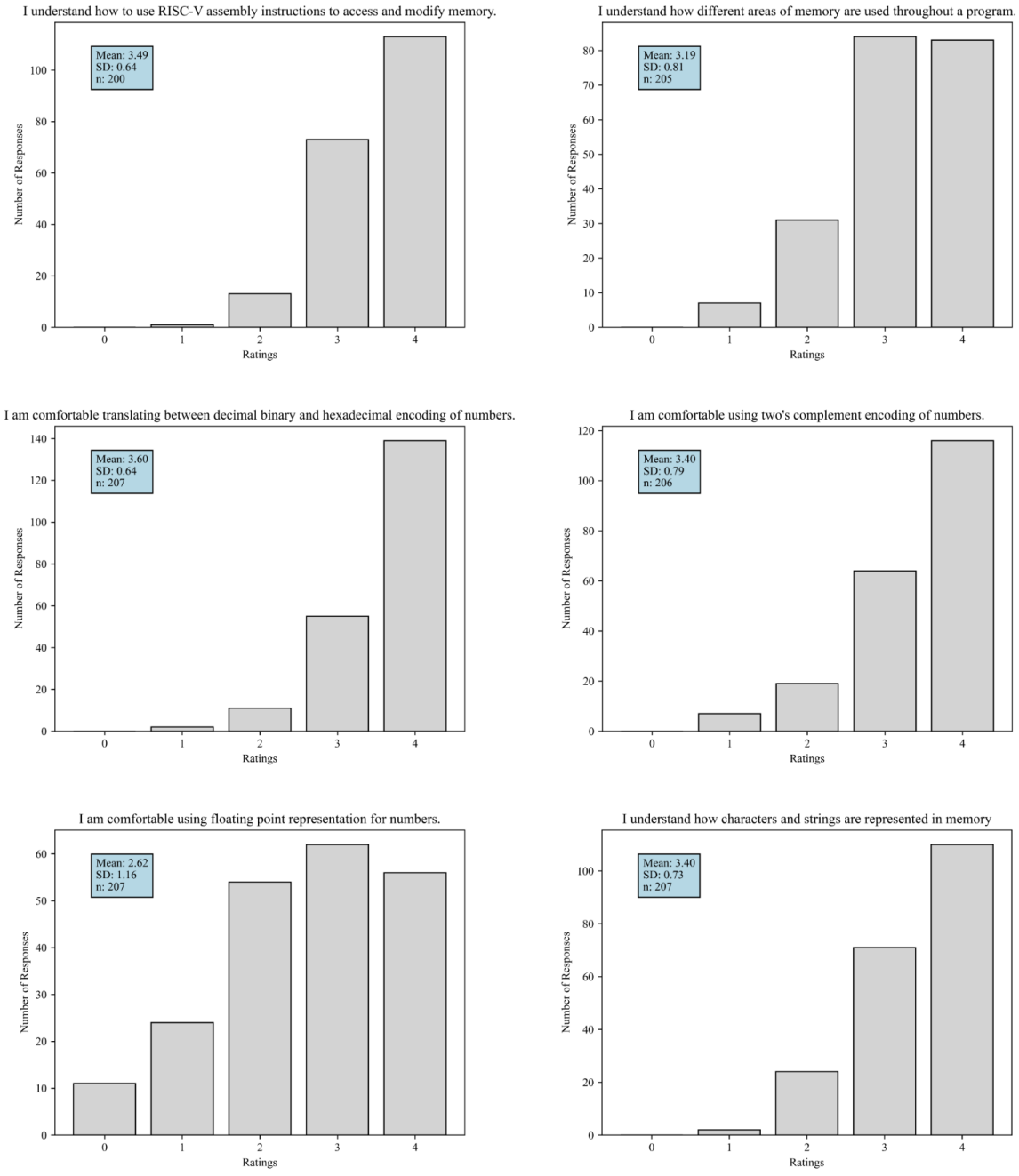


Figure 5-4: Histograms displaying the frequencies of student responses to the seventh through twelfth final survey prompts pertaining to their perceived understanding of main course topics and comfort with using them. A response of 0 indicates that the respondent strongly disagreed with the statement, a 2 indicates that the respondent was neutral towards the statement, and a 4 indicates that the respondent strongly agreed with the statement.

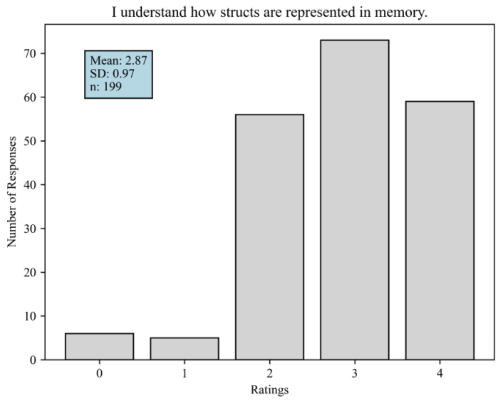


Figure 5-5: Histogram displaying the frequencies of student responses to the thirteenth final survey prompt pertaining to their perceived understanding of main course topics and comfort with using them. A response of 0 indicates that the respondent strongly disagreed with the statement, a 2 indicates that the respondent was neutral towards the statement, and a 4 indicates that the respondent strongly agreed with the statement.

Chapter 6

Future Work

After developing this course and running it for five academic quarters, we have identified four viable extensions of this work. Additionally, it will be important for future instructors of the course to continuously evaluate the effectiveness and relevance of the existing materials and make improvements accordingly.

6.1 Stable Toolchain for Laboratory Assignments

The embedded development toolchain that we use extensively for lab and postlab assignments is reliant on various external software applications and packages working as expected. When these applications and packages update, sometimes automatically, new behavior can be introduced.

For example, the Espressif32 development platform updated from version 5.1.1 to version 5.2.0 during the fall 2022 semester, and the ESP32 microcontroller would consistently enter a boot loop, despite being programmed with a known program using the exact same workflow and settings as we had used successfully many times before. Additionally, students must use their own personal computers for these assignments, some of which require different configuration settings.

It would be ideal if our development tools did not depend on software applications that could potentially update or on students' computers having particular hardware or software specifications. Developing a stable toolchain that is compatible with all

major operating systems and processors would significantly reduce the amount of time students and staff members spend troubleshooting software and hardware issues associated with system setup during the first week of the class. A future project could involve exploring options, such as using Docker, for enabling this.

6.2 C Debugging Tool

A typical student taking this course has only ever been exposed to basic, high-level programming in Python, so they may find C programs difficult to understand, especially early in the course. They could potentially benefit from a debugging tool that would allow them to walk through C programs and observe how memory and the different program variables evolve as the lines are run. Similar to the RISC-V debugging tool, this could be integrated with existing C program checker questions on the course website.

A C program debugging tool would be useful for troubleshooting C programs, but it also has some potentially-significant educational benefits. Strategically-designed visuals could emphasize the different types of variables and data structures and how they are stored in memory. For example, the tool could clearly display that a `uint8_t` variable, upon declaration, occupies a single byte in memory, while a `uint16_t` variable occupies two. Additionally, it could display more complex ideas such as how pointers reference values in memory while still themselves being stored in memory.

6.3 Full-Semester Version of 6.190

One significant constraint faced during the design and development of 6.190 was the short, half-semester length. This only allows for about six weeks' worth of material, and schedule constraints require us to administer the final exam during the sixth week of the course, so it does not test students on the material introduced during the sixth week of classes, which includes dynamic memory allocation. Additionally, there is no lab or postlab this week. Ideally, students would get the opportunity to explore

dynamic memory allocation in an assignment and be asked questions about it during the exam, because this topic is integral to understanding how memory is managed in C programs.

In addition to allowing time for proper coverage of dynamic memory allocation, extending the course to last for an entire semester would allow for the coverage of more related topics, including compilers, software performance and optimization, and operating systems. The additional time would provide the ability to administer multiple quizzes throughout the semester, rather than just one final exam, and the possibility of a guided or open-ended final project could be explored.

6.4 Analyzing Impact on Follow-on Classes

This class (6.190) was designed with MIT Computation Structures (6.191) in mind: moving assembly language content from 6.191 to 6.190 would allow 6.191 to introduce more topics related to computing systems and explore existing ones in more depth. 6.190 became an enforced corequisite for MIT Computation Structures (6.191) in the Spring 2023 semester, so 6.191 updated accordingly – new content includes a more in-depth treatment of virtual memory and exceptions and a lecture dedicated to parallel processing. Additionally, because 6.190 now covers the RISC-V assembly content that the first two 6.191 labs had focused on, 6.191 has replaced those labs with two new ones: the first focuses on implementing a four-stage pipelined RISC-V processor with full bypassing (this had previously been an optional design exercise), and the second focuses on operating systems. Additionally, 6.191 has introduced a new optional design project, as a primary aspect of the previous one is now the required four-stage pipelined processor lab. As these changes were all introduced in the Spring 2023 semester, their success could not be evaluated as part of this thesis. Future work could explore 6.190's impact on 6.191 by analyzing the changes 6.191 made in response to the inclusion of 6.190.

Additionally, the changes made to 6.191 would allow its follow-on classes to adjust accordingly. More future work could analyze how the inclusion of 6.190 ultimately

impacts classes taken later in the curriculum, such as 6.192 (Constructive Computer Architecture) and 6.205 (Digital Systems Laboratory), due to the changes it allowed 6.191 to make.

Appendix A

Code Appendix

A.1 Lab 1 Starter Code

A.1.1 Main C File

```
#include <stdio.h>
#include "6190.h"

void setup(){
    /* Setup function to configure GPIO pins and real-time clock. Today, the real-time
    clock has already been configured for you. */

    timerSetup(); // Initialize timer driven by real-time clock.

    // pinSetup(LED1, GPIO_OUTPUT); // Configure pin connected to LED1 as output
    // pinSetup(LED2, GPIO_OUTPUT); // Configure pin connected to LED2 as output

    // pinSetup(BTNL, GPIO_INPUT); // Configure pin connected to BTNL as input
    // pinSetup(BTNR, GPIO_INPUT); // Configure pin connected to BTNR as input
```

```
}

void app_main() {
    /* Main function, your program starts here. Any functions
        you call here must be defined above or in 6s077.h. */

    setup();

    // pinWrite(LED1, 1);  // Turn LED1 on
    // pinWrite(LED2, 0);  // Turn LED2 off

    while(1){
        // main loop of our program

        // int x = pinRead(BTNL);  // Read value of BTNL
        // if (x == 1){
        //     printf("button not pressed\n");
        // } else if (x == 0){
        //     printf("BUTTON PRESSED!\n");
        // }
    }
}
```

A.1.2 6.190 Header File

```
#include <stdio.h>
#include <stdint.h>

/* Header File for 6.190, using the ESP32-C3 */
/* ESP32-C3 Reference: https://www.espressif.com/sites/default/files
   /documentation/esp32-c3_technical_reference_manual_en.pdf */

////////////////////////////////
// Peripheral Base Addresses //
////////////////////////////////

```

```
#define GPIO_BASE_ADDR      0x60004000  // Base address for memory locations dedicated to GPIO matrix
#define IO_MUX_BASE_ADDR    0x60009000  // Base address for memory locations dedicated to IO multiplexer addresses
#define TIMER_BASE_ADDR     0x6001F000  // Base address for memory locations dedicated to timer


////////////////////////////////////////////////
// General-Purposed Input/Output Definitions //
////////////////////////////////////////////////

/* GPIO Addresses */
#define GPIO_OUT_ADDR       (GPIO_BASE_ADDR + 0x0004)
#define GPIO_ENABLE_ADDR    (GPIO_BASE_ADDR + 0x0020)
#define GPIO_IN_ADDR        (GPIO_BASE_ADDR + 0x003C)
#define IO_MUX_GPIOn_ADDR   (IO_MUX_BASE_ADDR + 0x0004) // 1 memory location/pin, need to add 4n for pin n

/* GPIO Pin Modes */
#define GPIO_INPUT 0
#define GPIO_OUTPUT 1

/* GPIO -> Peripheral Mapping */

// Switches
#define SW7 9
#define SW6 8
#define SW5 7
#define SW4 6
#define SW3 5
#define SW2 4
#define SW1 18
#define SW0 19

// Buttons
```

```
43 #define BTNF 9
44 #define BTNU 8
45 #define BTNC 7
46 #define BTND 6
47 #define BTNR 5
48 #define BTNL 4
49
50 // LEDs
51 #define LED1 2
52 #define LED2 3
53
54
55 /* GPIO Functions */
56 void pinSetup(int pin_num, int mode){
57    /* Setup function to configure GPIO pin (pin_num) as either an
    input
58    or an output, as defined by mode.
59
60    Arguments:
61        pin_num: GPIO pin number (0 - 31)
62        mode: Mode to configure pin as. GPIO_INPUT = 0, GPIO_OUTPUT =
     1
63    */
64
65    if (mode == GPIO_INPUT){
66        // Configure GPIO pin pin_num as input
67        int* io_mux_gpio_addr = (int*) (IO_MUX_GPIOn_ADDR + (4*
    pin_num));
68        *io_mux_gpio_addr =  // YOUR CODE HERE
69    } else if (mode == GPIO_OUTPUT){
70        // Configure GPIO pin pin_num as output.
71        int* gpio_enable_addr = (int*) GPIO_ENABLE_ADDR;
72        *gpio_enable_addr =  // YOUR CODE HERE
73    }
74 }
75
```

```
76
77 void pinWrite(int pin_num, int value){
78     /* Function to write either 0 or 1 (value) to an output GPIO pin
       (pin_num)
79     Arguments:
80         pin_num: GPIO pin number (0 - 31)
81         value: digital value (0 or 1) to write to GPIO pin
82     */
83
84     int* output_val = (int*) GPIO_OUT_ADDR;
85
86     // YOUR CODE HERE...
87
88 }
89
90
91 int pinRead(int pin_num){
92     /* Function to read the value of an input GPIO pin (pin_num).
93     Arguments:
94         pin_num: GPIO pin number (0 - 31)
95     Returns:
96         int: digital value (0 or 1) read from GPIO pin
97     */
98
99   int* input_val = (int*) GPIO_IN_ADDR;
100
101   // YOUR CODE HERE...
102
103   return 0; // replace 0 with return value of pinRead
104
105 }
106
107 ////////////////////////
108 // Timer Definitions //
109 ////////////////////////
110
```

```
111 /* Timer Addresses */
112 #define TIMER_CONFIG_ADDR          (TIMER_BASE_ADDR + 0x0000)
113 #define TIMER_COUNT_LO_ADDR        (TIMER_BASE_ADDR + 0x0004)
114 #define TIMER_COUNT_HI_ADDR        (TIMER_BASE_ADDR + 0x0008)
115 #define TIMER_COUNT_UPDATE_ADDR    (TIMER_BASE_ADDR + 0x000C)
116
117 #define WATCHDOG_CONFIG_ADDR        (TIMER_BASE_ADDR + 0x0048)
118 #define WATCHDOG_WRITE_PROTECT_ADDR (TIMER_BASE_ADDR + 0x0064)
119
120 /* Timer Functions */
121 void timerSetup(void){
122    /* Setup function to initalize Timer 0 in the ESP32-C3 Timer
       Group. */
123
124    // Configure timer by choosing clock source (external clock),
       prescalar value (0x01) and direction (increment).
125    int* timg_t0_config = (int*) TIMER_CONFIG_ADDR;
126    *timg_t0_config = (*timg_t0_config) | (0x01 << 9) | (0x01 << 13)
       | (0x01 << 30);
127
128    // Start timer by setting most sigificant bit (MSB) of value at
       timer configuration memory address
129    *timg_t0_config |= (0x01 << 31);
130
131    // Turn off watchdog timer timeout messages
132    int* wdog_addr = (int*) WATCHDOG_WRITE_PROTECT_ADDR;
133    *wdog_addr = 0x50d83aa1;
134
135    int* wdog_cfg = (int*) WATCHDOG_CONFIG_ADDR;
136    *wdog_cfg &= ~(0x01 << 31); // clear bit 31
137 }
138
139
140 uint64_t millis(void){
141    /* Converts the value of the counter driven by a 40MHz clock
       to a time-stamp in milliseconds.*/
```

```
143
144     // write to MSB of value at timer counter update address to latch
        (update/retain) value of counter
145     int* timg_to_update = (int*) TIMER_COUNT_UPDATE_ADDR;
146     *timg_to_update |= (1 << 31);
147
148     // (2) actually read...
149     int* timer_lo = (int*) TIMER_COUNT_LO_ADDR; // read low 32 bits
        of counter value
150     int* timer_hi = (int*) TIMER_COUNT_HI_ADDR; // upper 22 bits of
        counter value
151     uint32_t timer_lo_val = (uint32_t) *timer_lo;
152     uint32_t timer_hi_val = (uint32_t) *timer_hi;
153     uint64_t bigger_timer_hi = (uint64_t) timer_hi_val;
154
155     return ((bigger_timer_hi << 32) + timer_lo_val) >> 14;
156 }
157
158
159 uint64_t micros(void){
160     /* Converts the value of the counter driven by a 40MHz clock
161     to a time-stamp in microseconds.*/
162
163     // write to MSB of value at timer counter update address to latch
        (update/retain) value of counter
164     int* timg_to_update = (int*) TIMER_COUNT_UPDATE_ADDR;
165     *timg_to_update |= (1 << 31);
166
167     int* timer_lo = (int*) TIMER_COUNT_LO_ADDR; // read low 32 bits
        of counter value
168     int* timer_hi = (int*) TIMER_COUNT_HI_ADDR; // upper 22 bits of
        counter value
169     uint32_t timer_lo_val = (uint32_t) *timer_lo;
170     uint32_t timer_hi_val = (uint32_t) *timer_hi;
171     uint64_t bigger_timer_hi = (uint64_t) timer_hi_val;
172
```

```
173     return ((bigger_timer_hi << 32) + timer_lo_val) >> 4;
174 }
```

A.2 Postlab 1 Starter Code

A.2.1 Main C File

```
1      #include <stdio.h>
2 #include "6190.h"
3
4 const int TIME = 1000; // loop time
5 int switch_pins[] = {SW0,SW1,SW2,SW3,SW4,SW5,SW6,SW7};
6
7 // TASK 1:
8 // Get updateLeds working below
9 void updateLeds(int* leds){
10
11     // TODO: right-shift (and wrap) the value pointed to by leds
12
13 }
14
15 // TASK 2:
16 // Get getInputs working below
17 int getInputs(){
18
19     // TODO: return an integer with the lower eight bits
    representing our switches in order
20     // use switch_pins and values at address location
    GPIO_INPUT_VAL_ADDR
21
22     return 0; // replace return 0 with getInputs return value
23 }
24
25 // positions the 8-long led array into the larger screen buffer:
26 void drawLedArray(int leds){
```

```
27      screen_buffer[7] = leds<<8;
28      drawBuffer();
29  }
30
31  void app_main(){
32      timerSetup();
33      setupDisplay();
34      for(int i=0; i<8; i++){
35          pinSetup(switch_pins[i],GPIO_INPUT);
36      }
37      eraseBuffer();
38      int leds = 0x10;        // initial state of LEDs
39      int old_switches =0;    // for remembering switches
40      drawLedArray(leds);     // draw initial state on LED array
41      long timer = millis();
42      while(1){
43          while(millis()-timer < TIME){  // wait for TIME milliseconds
44              int new_switches = getInputs();
45              if (new_switches != old_switches){
46                  leds = leds;  // TASK 3 (update so new_switches
    alters leds!)
47                  drawLedArray(leds); // draw immediately
48                  old_switches = new_switches;   // remember what the
    switches were.
49                  break;
50              }
51          }
52          while(millis()-timer < TIME);  // wait for remainder of loop
     time
53          timer = millis();     // update and remember
54          updateLeds(&leds);   // update LED array
55          drawLedArray(leds);
56      }
57  }
```

A.2.2 6.190 Header File Code for Driving LED Display

This code is added to the 6.190 header file that students used in Lab 1. Students will use this file for the remainder of the labs, making slight modifications to it depending on the assignment.

```
////////////////////////////////////////////////////
// Serial Peripheral Interface (SPI) Functions //
////////////////////////////////////////////////////

/* Mapping of GPIO pin to SPI signal */
#define CLK 0
#define CS 1
#define MOSI 10

void spiPause(){
    /* 1 microsecond delay for SPI timing */

    uint64_t t = micros();
    while(micros()-t<1);
}


void spiWrite(uint8_t *data, uint8_t len, uint8_t clk_pin, uint8_t
    mosi_pin, uint8_t cs_pin ){
    /* Performs software-driven SPI write
     Arguments:
        data: pointer to array of 8-bit values to be sent serially
        len: number of 8-bit values to write
        clk_pin: GPIO pin used to generate clock signal
        mosi_pin: GPIO pin used to transmit data serially
        cs_pin: GPIO pin used for chip select
    */

    pinWrite(cs_pin,0); // bring chip select signal low

    // Send data in buffer in 8-bit chunks
```

```
31      // Generate clock signal and send 1 bit/clock period
32      for (int q=0; q<len; q++){
33          for (int p=0; p<8; p++){
34              pinWrite(clk_pin,0);
35              pinWrite(mosi_pin,((data[q]>>p)&0x01));
36              spiPause();
37              pinWrite(clk_pin,1);
38              spiPause();
39          }
40      }
41      pinWrite(cs_pin,1); // bring chip-select signal high
42  }
43
44  //////////////////
45  // LED Display //
46  //////////////////
47
48  // LED Display Driver (MAX7219) Documentation: https://datasheets.
        maximintegrated.com/en/ds/MAX7219-MAX7221.pdf
49
50  /* Helper functions for flipping bits */
51  void bit_reverse(int value, int *dump){
52      for (int i = 0; i<32; i++){
53          *dump = ((*dump)<<1)|((value>>i)&0x1);
54      }
55  }
56
57  uint8_t flip_8(uint8_t value){
58      uint8_t t =0;
59      for (int i = 0; i<8; i++){
60          t = (t<<1)|((value>>i)&0x1);
61      }
62      return t;
63  }
64
65  /* LED Display Driver Functions */
```

```
void setupDisplay(){
    /* Function to set up LED display */

    // Set up necessary GPIO pins
    pinSetup(CS, GPIO_OUTPUT);
    pinSetup(MOSI, GPIO_OUTPUT);
    pinSetup(CLK, GPIO_OUTPUT);

    // Pull chip select, clock, and data out high
    pinWrite(CS,1);
    pinWrite(MOSI,1);
    pinWrite(CLK,1);

    // Configure LED display
    int DISP_TEST = 0x0F000F00; //display test
    int DISP_OFF = 0x0C000C00;  //display off
    int DISP_ON = 0x0C010C01;   //display on
    int DEC_0 = 0x09000900;     //decode mode 0
    int SCAN_M = 0x0B070B07;    //scan mode setting
    int INTENSITY = 0x0A010A01; //screen intensity

    int config_buffers[] = {DISP_TEST, DISP_OFF, DISP_ON, DEC_0,
   SCAN_M, INTENSITY};

    for (int i = 0; i < 6; i = i + 1){
        int t[2] = {0,0};
        bit_reverse(config_buffers[i],t);
        bit_reverse(config_buffers[i],t+1);
        uint8_t *s = (uint8_t*)t;
        for (int i = 0; i<4; i++){
            spiWrite(s,2,CLK,MOSI,CS);
        }

        int del = millis();
        while(millis()-del<5);
    }
```

```
}

uint32_t screen_buffer [8]; // 8x32 array for storing LED display
    image

void eraseBuffer(){
    /* Function that clears screen buffer by setting all bits to 0.
    */

    for (uint8_t i=0; i<8; i++){
        screen_buffer[i]=0;     //set all 32 bits to 0
    }
}

void drawBuffer(){
    /* Function that sends bytes in screen buffer to be displayed on
    LED array.*/

    uint8_t buffer[8];
    for (int k = 1; k <9; k++){
        for (int m=0; m<4;m++){
            buffer[0+2*m] = (uint8_t)flip_8(9-k);
            uint8_t temp = (uint8_t)(screen_buffer[k-1]>>(8*m));
            buffer[1+2*m] = temp & 0x000000FF;
        }
        spiWrite(buffer,8,CLK,MOSI,CS);
    }
}
```

A.3 Lab 2 Starter Code

A.3.1 Main C File

```
#include <stdio.h>
#include "6190.h"
```

```
3
4  int switch_pins[8] = {SW0, SW1, SW2, SW3, SW4, SW5, SW6, SW7};
5
6  void setup(){
7
8      timerSetup();   // Configure timer
9      setupDisplay(); // Configure display
10     eraseBuffer();  // Erase random data from display
11     drawBuffer();   // Display contents of buffer on LED array
12
13     for (int i = 0; i < 7; i++){
14         pinSetup(switch_pins[i], GPIO_INPUT);
15     }
16     pinSetup(BTNF, GPIO_INPUT);
17
18 }
19
20
21 char switchToBinary(){
22     /*
23     Return:
24         char: 8-bit value. Bits 0-6 correspond to switch inputs, bit
       7 is 0.
25     */
26
27     ////////////////////////////////////////
28     // TO-DO: Complete switchToBinary()    //
29     ////////////////////////////////////////
30
31     return 0; // replace!
32 }
33
34
35 void drawAsciiString(char* string_to_draw){
36     /* Function that draws a string on the LED array using the
      mapping defined in ascii.h
```

```
37      by filling in a buffer.
38
39      Argument:
40          char* string_to_draw: String to draw on the LED array.
41      */
42
43      ////////////////////////////////////////
44      // TO-DO: Complete drawAsciiString()   //
45      ////////////////////////////////////////
46
47
48  }
49
50
51  void app_main(){
52      setup();
53
54      char ascii_input = 0;
55
56      while(1){
57          ascii_input = switchToBinary();
58          printf("Integer value: 0x%x, ", ascii_input);   // printing
    this value using %x formatter gives a hexadecimal value...
59          printf("ASCII character: %c\n", ascii_input);   // ...
    printing the same value with %c formatter gives a character
60
61          // int btnf = pinRead(BTNF);        // read BTNF to obtain
    the current state of the button
62          // if (btnf == 0){
63          //     /* TO-DO: Put your print statements from the last
    part in here */
64          // }
65
66          // Code to slow down loop so that "bouncing" associated with
     a single button press doesn't
67          // look like more than one button press. Write all of your
```

```
    code in above this point in the loop.
68          int start = millis();           // Get "start" time stamp
69          while(millis() - start < 50);   // Wait until current time
    stamp - "start" is greater than 50ms
70      }
71  }
```

A.3.2 ASCII Header File

This file is used for rendering ASCII characters on the labkit's LED display. This header file is derived from [10] – the sole difference is that we reversed the bits in each byte in the array to improve compatibility with the existing laboratory infrastructure.

```
1  #include <stdint.h>
2
3  // Source: https://github.com/dhepper/font8x8/blob/master/
       font8x8_basic.h
4
5  uint8_t ascii[128][8] = {
6      { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0000 (
       nul)
7      { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0001
8      { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0002
9      { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0003
10     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0004
11     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0005
12     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0006
13     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0007
14     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0008
15     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0009
16     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+000A
17     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+000B
18     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+000C
19     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+000D
20     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+000E
21     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+000F
```

```
22     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0010
23     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0011
24     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0012
25     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0013
26     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0014
27     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0015
28     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0016
29     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0017
30     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0018
31     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0019
32     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+001A
33     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+001B
34     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+001C
35     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+001D
36     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+001E
37     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+001F
38     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0020 (
    space)
39     { 0x18, 0x3C, 0x3C, 0x18, 0x18, 0x00, 0x18, 0x00},   // U+0021
    (!)
40     { 0x6C, 0x6C, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0022
    (")
41     { 0x6C, 0x6C, 0xFE, 0x6C, 0xFE, 0x6C, 0x6C, 0x00},   // U+0023
    (#)
42     { 0x30, 0x7C, 0xC0, 0x78, 0x0C, 0xF8, 0x30, 0x00},   // U+0024 (
    $)
43     { 0x00, 0xC6, 0xCC, 0x18, 0x30, 0x66, 0xC6, 0x00},   // U+0025
    (%)
44     { 0x38, 0x6C, 0x38, 0x76, 0xDC, 0xCC, 0x76, 0x00},   // U+0026
    (&)
45     { 0x60, 0x60, 0xC0, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0027
    (')
46     { 0x18, 0x30, 0x60, 0x60, 0x60, 0x30, 0x18, 0x00},   // U+0028
    (()
47     { 0x60, 0x30, 0x18, 0x18, 0x18, 0x30, 0x60, 0x00},   // U+0029
    ())
```

```
48     { 0x00, 0x66, 0x3C, 0xFF, 0x3C, 0x66, 0x00, 0x00},   // U+002A
       (*)
49     { 0x00, 0x30, 0x30, 0xFC, 0x30, 0x30, 0x00, 0x00},   // U+002B
       (+)
50     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x30, 0x30, 0x60},   // U+002C
       (,)
51     { 0x00, 0x00, 0x00, 0xFC, 0x00, 0x00, 0x00, 0x00},   // U+002D
       (-)
52     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x30, 0x30, 0x00},   // U+002E
       (.)
53     { 0x06, 0x0C, 0x18, 0x30, 0x60, 0xC0, 0x80, 0x00},   // U+002F
       (/)
54     { 0x7C, 0xC6, 0xCE, 0xDE, 0xF6, 0xE6, 0x7C, 0x00},   // U+0030
       (0)
55     { 0x30, 0x70, 0x30, 0x30, 0x30, 0x30, 0xFC, 0x00},   // U+0031
       (1)
56     { 0x78, 0xCC, 0x0C, 0x38, 0x60, 0xCC, 0xFC, 0x00},   // U+0032
       (2)
57     { 0x78, 0xCC, 0x0C, 0x38, 0x0C, 0xCC, 0x78, 0x00},   // U+0033
       (3)
58     { 0x1C, 0x3C, 0x6C, 0xCC, 0xFE, 0x0C, 0x1E, 0x00},   // U+0034
       (4)
59     { 0xFC, 0xC0, 0xF8, 0x0C, 0x0C, 0xCC, 0x78, 0x00},   // U+0035
       (5)
60     { 0x38, 0x60, 0xC0, 0xF8, 0xCC, 0xCC, 0x78, 0x00},   // U+0036
       (6)
61     { 0xFC, 0xCC, 0x0C, 0x18, 0x30, 0x30, 0x30, 0x00},   // U+0037
       (7)
62     { 0x78, 0xCC, 0xCC, 0x78, 0xCC, 0xCC, 0x78, 0x00},   // U+0038
       (8)
63     { 0x78, 0xCC, 0xCC, 0x7C, 0x0C, 0x18, 0x70, 0x00},   // U+0039
       (9)
64     { 0x00, 0x30, 0x30, 0x00, 0x00, 0x30, 0x30, 0x00},   // U+003A
       (:)
65     { 0x00, 0x30, 0x30, 0x00, 0x00, 0x30, 0x30, 0x60},   // U+003B
       (;)
```

```
66     { 0x18, 0x30, 0x60, 0xC0, 0x60, 0x30, 0x18, 0x00},   // U+003C
       (<)
67     { 0x00, 0x00, 0xFC, 0x00, 0x00, 0xFC, 0x00, 0x00},   // U+003D
       (=)
68     { 0x60, 0x30, 0x18, 0x0C, 0x18, 0x30, 0x60, 0x00},   // U+003E
       (>)
69     { 0x78, 0xCC, 0x0C, 0x18, 0x30, 0x00, 0x30, 0x00},   // U+003F
       (?)
70     { 0x7C, 0xC6, 0xDE, 0xDE, 0xDE, 0xC0, 0x78, 0x00},   // U+0040 (
       @)
71     { 0x30, 0x78, 0xCC, 0xCC, 0xFC, 0xCC, 0xCC, 0x00},   // U+0041 (
       A)
72     { 0xFC, 0x66, 0x66, 0x7C, 0x66, 0x66, 0xFC, 0x00},   // U+0042 (
       B)
73     { 0x3C, 0x66, 0xC0, 0xC0, 0xC0, 0x66, 0x3C, 0x00},   // U+0043 (
       C)
74     { 0xF8, 0x6C, 0x66, 0x66, 0x66, 0x6C, 0xF8, 0x00},   // U+0044 (
       D)
75     { 0xFE, 0x62, 0x68, 0x78, 0x68, 0x62, 0xFE, 0x00},   // U+0045 (
       E)
76     { 0xFE, 0x62, 0x68, 0x78, 0x68, 0x60, 0xF0, 0x00},   // U+0046 (
       F)
77     { 0x3C, 0x66, 0xC0, 0xC0, 0xCE, 0x66, 0x3E, 0x00},   // U+0047 (
       G)
78     { 0xCC, 0xCC, 0xCC, 0xFC, 0xCC, 0xCC, 0xCC, 0x00},   // U+0048 (
       H)
79     { 0x78, 0x30, 0x30, 0x30, 0x30, 0x30, 0x78, 0x00},   // U+0049 (
       I)
80     { 0x1E, 0x0C, 0x0C, 0x0C, 0xCC, 0xCC, 0x78, 0x00},   // U+004A (
       J)
81     { 0xE6, 0x66, 0x6C, 0x78, 0x6C, 0x66, 0xE6, 0x00},   // U+004B (
       K)
82     { 0xF0, 0x60, 0x60, 0x60, 0x62, 0x66, 0xFE, 0x00},   // U+004C (
       L)
83     { 0xC6, 0xEE, 0xFE, 0xFE, 0xD6, 0xC6, 0xC6, 0x00},   // U+004D (
       M)
```

```
84     { 0xC6, 0xE6, 0xF6, 0xDE, 0xCE, 0xC6, 0xC6, 0x00},   // U+004E (
       N)
85     { 0x38, 0x6C, 0xC6, 0xC6, 0xC6, 0x6C, 0x38, 0x00},   // U+004F (
       O)
86     { 0xFC, 0x66, 0x66, 0x7C, 0x60, 0x60, 0xF0, 0x00},   // U+0050 (
       P)
87     { 0x78, 0xCC, 0xCC, 0xCC, 0xDC, 0x78, 0x1C, 0x00},   // U+0051 (
       Q)
88     { 0xFC, 0x66, 0x66, 0x7C, 0x6C, 0x66, 0xE6, 0x00},   // U+0052 (
       R)
89     { 0x78, 0xCC, 0xE0, 0x70, 0x1C, 0xCC, 0x78, 0x00},   // U+0053 (
       S)
90     { 0xFC, 0xB4, 0x30, 0x30, 0x30, 0x30, 0x78, 0x00},   // U+0054 (
       T)
91     { 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xFC, 0x00},   // U+0055 (
       U)
92     { 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0x78, 0x30, 0x00},   // U+0056 (
       V)
93     { 0xC6, 0xC6, 0xC6, 0xD6, 0xFE, 0xEE, 0xC6, 0x00},   // U+0057 (
       W)
94     { 0xC6, 0xC6, 0x6C, 0x38, 0x38, 0x6C, 0xC6, 0x00},   // U+0058 (
       X)
95     { 0xCC, 0xCC, 0xCC, 0x78, 0x30, 0x30, 0x78, 0x00},   // U+0059 (
       Y)
96     { 0xFE, 0xC6, 0x8C, 0x18, 0x32, 0x66, 0xFE, 0x00},   // U+005A (
       Z)
97     { 0x78, 0x60, 0x60, 0x60, 0x60, 0x60, 0x78, 0x00},   // U+005B
       ([)
98     { 0xC0, 0x60, 0x30, 0x18, 0x0C, 0x06, 0x02, 0x00},   // U+005C
       (\)
99     { 0x78, 0x18, 0x18, 0x18, 0x18, 0x18, 0x78, 0x00},   // U+005D
       (])
100    { 0x10, 0x38, 0x6C, 0xC6, 0x00, 0x00, 0x00, 0x00},   // U+005E
       (^)
101    { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF},   // U+005F (
       _)
```

```
102     { 0x30, 0x30, 0x18, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0060 (‘)
103     { 0x00, 0x00, 0x78, 0x0C, 0x7C, 0xCC, 0x76, 0x00},   // U+0061 (a)
104     { 0xE0, 0x60, 0x60, 0x7C, 0x66, 0x66, 0xDC, 0x00},   // U+0062 (b)
105     { 0x00, 0x00, 0x78, 0xCC, 0xC0, 0xCC, 0x78, 0x00},   // U+0063 (c)
106     { 0x1C, 0x0C, 0x0C, 0x7C, 0xCC, 0xCC, 0x76, 0x00},   // U+0064 (d)
107     { 0x00, 0x00, 0x78, 0xCC, 0xFC, 0xC0, 0x78, 0x00},   // U+0065 (e)
108     { 0x38, 0x6C, 0x60, 0xF0, 0x60, 0x60, 0xF0, 0x00},   // U+0066 (f)
109     { 0x00, 0x00, 0x76, 0xCC, 0xCC, 0x7C, 0x0C, 0xF8},   // U+0067 (g)
110     { 0xE0, 0x60, 0x6C, 0x76, 0x66, 0x66, 0xE6, 0x00},   // U+0068 (h)
111     { 0x30, 0x00, 0x70, 0x30, 0x30, 0x30, 0x78, 0x00},   // U+0069 (i)
112     { 0x0C, 0x00, 0x0C, 0x0C, 0x0C, 0xCC, 0xCC, 0x78},   // U+006A (j)
113     { 0xE0, 0x60, 0x66, 0x6C, 0x78, 0x6C, 0xE6, 0x00},   // U+006B (k)
114     { 0x70, 0x30, 0x30, 0x30, 0x30, 0x30, 0x78, 0x00},   // U+006C (l)
115     { 0x00, 0x00, 0xCC, 0xFE, 0xFE, 0xD6, 0xC6, 0x00},   // U+006D (m)
116     { 0x00, 0x00, 0xF8, 0xCC, 0xCC, 0xCC, 0xCC, 0x00},   // U+006E (n)
117     { 0x00, 0x00, 0x78, 0xCC, 0xCC, 0xCC, 0x78, 0x00},   // U+006F (o)
118     { 0x00, 0x00, 0xDC, 0x66, 0x66, 0x7C, 0x60, 0xF0},   // U+0070 (p)
119     { 0x00, 0x00, 0x76, 0xCC, 0xCC, 0x7C, 0x0C, 0x1E},   // U+0071 (q)
```

```
    { 0x00, 0x00, 0xDC, 0x76, 0x66, 0x60, 0xF0, 0x00},   // U+0072 (r)
    { 0x00, 0x00, 0x7C, 0xC0, 0x78, 0x0C, 0xF8, 0x00},   // U+0073 (s)
    { 0x10, 0x30, 0x7C, 0x30, 0x30, 0x34, 0x18, 0x00},   // U+0074 (t)
    { 0x00, 0x00, 0xCC, 0xCC, 0xCC, 0xCC, 0x76, 0x00},   // U+0075 (u)
    { 0x00, 0x00, 0xCC, 0xCC, 0xCC, 0x78, 0x30, 0x00},   // U+0076 (v)
    { 0x00, 0x00, 0xC6, 0xD6, 0xFE, 0xFE, 0x6C, 0x00},   // U+0077 (w)
    { 0x00, 0x00, 0xC6, 0x6C, 0x38, 0x6C, 0xC6, 0x00},   // U+0078 (x)
    { 0x00, 0x00, 0xCC, 0xCC, 0xCC, 0x7C, 0x0C, 0xF8},   // U+0079 (y)
    { 0x00, 0x00, 0xFC, 0x98, 0x30, 0x64, 0xFC, 0x00},   // U+007A (z)
    { 0x1C, 0x30, 0x30, 0xE0, 0x30, 0x30, 0x1C, 0x00},   // U+007B ({)
    { 0x18, 0x18, 0x18, 0x00, 0x18, 0x18, 0x18, 0x00},   // U+007C (|)
    { 0xE0, 0x30, 0x30, 0x1C, 0x30, 0x30, 0xE0, 0x00},   // U+007D (})
    { 0x76, 0xDC, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+007E (~)
    { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}    // U+007F
};
```

A.4 Postlab 2 Starter Code

A.4.1 Main C File

```
#include <stdio.h>
#include "6190.h"
```

```
3
4  void setup(){
5
6      timerSetup();   // Configure timer
7      setupDisplay(); // Configure display
8      eraseBuffer();  // Erase random data from display
9      drawBuffer();   // Display contents of buffer on LED array
10 }
11
12
13 int messageLength(char* message){
14
15     ////////////////////////////////////////
16     // TO-DO: Complete messageLength()    //
17     ////////////////////////////////////////
18
19     return 0; // Replace!
20 }
21
22
23 void fillScreenBuffer(char* message, int total_offset){
24
25     ////////////////////////////////////////
26     // TO-DO: Complete fillScreenBuffer() //
27     ////////////////////////////////////////
28
29 }
30
31
32 void app_main(){
33     setup();
34
35     char message[] = "     THEY SEE ME SCROLLIN' THEY HATIN'     ";
36     int len = messageLength(message);  // TASK 1: CALCULATE MESSAGE
       LENGTH
37     int offset = 0;
```

```
    while(1){
        if (offset == len * 8){
            offset = 0; // Reset if we have reached the end of the message
        } else {
            eraseBuffer();  // Clear screen buffer
            fillScreenBuffer(message, offset); // TASK 2: FILL SCREEN BUFFER WITH CORRECT DATA
            drawBuffer();   // Transmit screen buffer data to screen
            offset++;        // Move forward
        }

        int start = millis();             // Get "start" time stamp
        while(millis() - start < 100);    // Wait until (current time stamp - "start") >= 100ms
    }
}
```

A.5 Lab 3 Starter Code

A.5.1 Main C File

This starter code is the basis for both Lab 3 and Postlab 3.

```
#include <stdio.h>
#include <stdint.h>
#include "6190.h"

// Constants
#define SCREEN_ROWS 8
#define SCREEN_COLS 32
enum snake_direction { up, down, left, right };

// Setup for random value generator
#define RANDOM_VALUE_ADDR 0x600260B0
#define RTC_CNTL_CLK_CONF_REG 0x60008070
```

```
13
14 void setupRandom(){
15     int *rtc_cntl_clk_conf_rg = (int*) RTC_CNTL_CLK_CONF_REG;
16     *rtc_cntl_clk_conf_rg |= (1 << 10);
17 }
18
19
20 // Snake struct definition
21 struct Snake {
22     uint8_t body[SCREEN_COLS*SCREEN_ROWS]; // can get as large as
    entire screen in theory
23     uint8_t direction;
24     uint8_t length;
25 };
26
27
28 void setup(){
29     timerSetup();
30     setupRandom();
31     setupDisplay();
32     eraseBuffer();
33     int snake_buttons[5] = {BTNF, BTNU, BTND, BTNR, BTNL};
34     for (int i = 0; i < 5; i++){
35         pinSetup(snake_buttons[i], GPIO_INPUT);
36     }
37 }
38
39
40 uint8_t getX(uint8_t location){
41     /* Function that returns the x coordinate from an 8-bit value
    containing both x and y coordinates.
42         The x coordinate is encoded in the upper (most significant)
    5 bits of the input.
43     Arguments:
44         location: an unsigned 8-bit value representing a location on
     an 8x32 gameboard.
```

```
45      Returns:
46          uint8_t representing the x-coordinate (0-31) encoded in the
        upper 5 bits of the argument.
47      */
48
49      ////////////////////////////////
50      // TO-DO: Implement getX() //
51      ////////////////////////////////
52
53      return 0; // replace
54  }
55
56
57  uint8_t getY(uint8_t location){
58      /* Function that returns the y coordinate from an 8-bit value
        containing both x and y coordinates.
59      Arguments:
60          location: an unsigned 8-bit value representing a location on
        an 8x32 gameboard.
61      Returns:
62          uint8_t representing the y-coordinate (0-7) encoded in the
        lower 3 bits of the argument.
63      */
64
65      ////////////////////////////////
66      // TO-DO: Implement getY() //
67      ////////////////////////////////
68
69      return 0; // replace
70  }
71
72
73  void setX(uint8_t *location, uint8_t new_x){
74      /* Function that updates the upper 5 bits of the value stored at
        location to be equal to new_x.
75      Arguments:
```

```
76          uint8_t *location: a pointer to an unsigned 8-bit value
    representing a location on an 8x32 gameboard.
77          uint8_t new_x: an unsigned integer representing the value
    (0-31) that we'd like to set for the x location.
78      */
79
80      ////////////////////////////////
81      // TO-DO: Implement setX() //
82      ////////////////////////////////
83
84  }
85
86
87  void setY(uint8_t *location, uint8_t new_y){
88      /* Function that updates the lower 3 bits of the value stored at
        location to be equal to new_y.
89      Arguments:
90          uint8_t *location: a pointer to an unsigned 8-bit value
    representing a location on an 8x32 gameboard.
91          uint8_t new_y an unsigned integer representing the value
    (0-7) that we'd like to set for the y location.
92      */
93
94      ////////////////////////////////
95      // TO-DO: Implement setY() //
96      ////////////////////////////////
97  }
98
99
100 void setPixel(uint8_t location, uint8_t val){
101     /* Function that sets the value of an indivdual pixel (LED) on
    the display.
102     Arguments:
103         location: The location of the LED to be turned on/off on the
    LED array
```

```
104         val: The binary (0 or 1) value indicating if the LED at the
    location
105             should be off (0) or on (1).
106     */
107
108     ////////////////////////////////
109     // TO-DO: Implement setPixel() //
110     ////////////////////////////////
111
112 }
113
114
115 void drawBoard(struct Snake *snake, uint8_t food){
116     /* Function to render the snake and food on the board on each
    given time-step.
117     Arguments:
118         struct Snake *snake: pointer to snake struct
119         uint8_t food: location of food to render
120     */
121
122     ////////////////////////////////
123     // TO-DO: Implement drawBoard() //
124     ////////////////////////////////
125
126 }
127
128
129 void updateSnake(struct Snake *snake){
130     /* Function to update the snake on each time-step based on its
    direction.
131     Arguments:
132         struct Snake *snake: pointer to snake struct
133     */
134
135     ////////////////////////////////
136     // TO-DO: Implement updateSnake() //
```

```
137     ////////////////////////////////////
138
139 }
140
141
142 int generateFood(struct Snake *snake, uint8_t *food){
143     /* Function to randomly generate food for the snake.
144     Arguments:
145         struct Snake *snake: pointer to snake struct
146         uint8_t *food: pointer to variable holding the food's
    location
147     Returns:
148         int: 0 if food does not conflict with the snake, 1 if it
    does.
149     */
150
151     int *random_value_ptr = (int*) RANDOM_VALUE_ADDR; // read this
    address to obtain a random integer value
152
153     ////////////////////////////////////
154     // TO-DO: Complete generateFood() //
155     ////////////////////////////////////
156
157     return 0; // UPDATE
158 }
159
160
161 uint8_t snakeAteFood(uint8_t *snake_body, uint8_t food){
162     /* Function to determine if the snake collided with (and ate)
    the food.
163     Arguments:
164         uint8_t *snake_body: pointer to the snake body array
165         uint8_t food: variable holding the food's location
166     Returns:
167         int: 0 if the snake did not eat the food, 1 if it did.
168     */
```

```
169
170     ////////////////////////////////////
171     // TO-DO: Implement snakeAteFood() //
172     ////////////////////////////////////
173
174     return 0; // UPDATE
175 }
176
177
178 uint8_t snakeCollisionCheck(struct Snake *snake){
179     /* Function to determine if the snake collided with itself.
180     Arguments:
181         struct Snake *snake: pointer to snake struct
182     Returns:
183         int: 0 if the snake did not collide with itself, 1 if it did
        .
184     */
185
186     ////////////////////////////////////////////
187     // TO-DO: Implement snakeCollisionCheck() //
188     ////////////////////////////////////////////
189
190     return 0; // UPDATE
191 }
192
193
194 void app_main() {
195     setup(); // System setup
196
197     // Snake initialization
198     struct Snake snake;
199     // COMMENT OUT FOR PART 10
200     snake.body[0] = 0;
201     snake.direction = left;
202     snake.length = 1;
203
```

```
204     // UNCOMMENT FOR PART 10
205     // for (int i=0; i<3; i++){
206     //     setX(&snake.body[i], 5);
207     //     setY(&snake.body[i], 3-i);
208     // }
209     // snake.length = 3;
210     // snake.direction = left;
211 
212     uint8_t food = 0; // food in top right corner of the game board
213 
214     while(generateFood(&snake, &food) != 0); // Generate food initially
215     drawBoard(&snake, food);
216 
217     while(1){
218 
219         ////////////////////////////////////////////
220         // INSERT BUTTON PRESS DETECTION LOGIC HERE //
221         ////////////////////////////////////////////
222 
223         updateSnake(&snake);
224         drawBoard(&snake, food);
225 
226         if (snakeAteFood(snake.body, food) == 1){
227             ////////////////////////////
228             // TO-DO: "Grow" snake    //
229             ////////////////////////////
230 
231             // Generate food
232             while (generateFood(&snake, &food) != 0); // generate food
233         }
234 
235         if (snakeCollisionCheck(&snake) == 1){
236             // Game ends
237             eraseBuffer(); // Clear game board
```

```
238             drawBuffer();
239             while(1); // Infinite loop since game is over.
240         }
241 
242         // Code to limit frequency of loop so that gameplay is
    possible.
243         // Put all game code above this point.
244         int start = millis();
245         while (millis() - start < 200); // One loop iteration every
    ~200ms
246     }
247 }
```

A.6 Lab 4 Starter Code

A.6.1 Main C File

```
1 #include <stdio.h>
2 #include "6190.h"
3 
4 void setup(){
5     timerSetup();
6 
7     pinSetup(LED1, GPIO_OUTPUT); // Configure LED1 GPIO pin as
    output
8     pinSetup(LED2, GPIO_OUTPUT); // Configure LED2 GPIO pin output
9 
10    pinSetup(BTNC, GPIO_INPUT);  // Configure BTNC GPIO pin as input
11 
12    setupDisplay();
13    eraseBuffer(screen_buffer);
14    drawBuffer();
15 }
16 
17 void app_main() {
```

```
18      setup();
19
20      // CODE TO TEST pinWrite
21      // pinWrite(LED1, 1);
22      // pinWrite(LED2, 0);
23
24      int evens = 1, odds = 0;
25      while(1){
26
27          // CODE TO TEST pinRead
28          int x = pinRead(BTNC);  // Read value of GPIO pin 7
29          if (x == 1){
30              printf("button not pressed\n");
31          } else if (x == 0){
32              printf("BUTTON PRESSED!\n");
33          }
34
35          // CODE TO TEST setPixel
36          // evens ^= 1;
37          // odds ^= 1;
38
39          // for (int i = 0; i < 32; i = i + 1){
40          //     for (int j = 0; j < 8; j = j + 1){
41          //         if (j%2== 0){
42          //             setPixel(screen_buffer, i, j, evens);
43          //         }
44          //         else{
45          //             setPixel(screen_buffer, i, j, odds);
46          //         }
47          //     }
48          // }
49
50          // drawBuffer();
51          // long start = millis();
52          // while (millis() - start < 1000);
53
```

```
    }
}
```

A.6.2 Assembly Language File for Hardware-Related Procedures

```
.section .text      # indicates a code segment
.align 2            # specifies how data should be arranged in memory
//.globl pinSetup   # makes pinSetup able to be used outside of this file
.globl pinWrite     # makes pinWrite able to be used outside of this file
.globl pinRead      # makes pinRead able to be used outside of this file
.globl setPixel     # makes setPixel able to be used outside of this file
.globl eraseBuffer  # makes eraseBuffer able to be used outside of this file

# C declaration: int pinRead(int pin_num)
# ARGUMENTS a0: pin_num
# RETURNS bit read from GPIO pin
pinRead:
    lui a2, 0x60004
    addi a2, a2, 0x3C # GPIO_IN_ADDR

    # YOUR CODE HERE...

    jalr x0, 0(ra)


# C declaration: void pinWrite(int pin_num, int value)
# ARGUMENTS a0: pin_num, a1: value
# RETURNS: Nothing
```

```
24 pinWrite:
25     lui a2, 0x60004
26     addi a2, a2, 0x4  # GPIO_OUT_ADDR
27 
28     # YOUR CODE HERE...
29 
30     jalr x0, 0(ra)
31 
32 
33 # C declaration: void pinSetup(int pin_num, int mode)
34 # ARGUMENTS a0: pin_num, a1: mode
35 # RETURNS: Nothing
36 pinSetup:
37     lui a2, 0x60009
38     addi a2, a2, 0x4  # IO_MUX_GPIOn_ADDR
39 
40     lui a3, 0x60004
41     addi a3, a3, 0x20 # GPIO_ENABLE_ADDR
42 
43     # YOUR CODE HERE...
44 
45     jalr x0, 0(ra)
46 
47 
48 # C declaration: void setPixel(uint32_t* screen_buffer_addr, uint8_t
       x, uint8_t y, uint8_t val);
49 # ARGUMENTS a0: screen_buffer base address, a1: x, a2: y, a3: val
50 # RETURNS: Nothing
51 setPixel:
52     # YOUR CODE HERE...
53 
54     jalr x0, 0(ra)
55 
56 
57 # C declaration: void eraseBuffer(uint32_t* screen_buffer_addr)
58 # ARGUMENTS a0: screen_buffer base address
```

```
# RETURNS: Nothing

eraseBuffer:
    addi a1, x0, 8      # upper bound on for loop
    addi a2, x0, 0      # "i" for for loop
looping:
    slli a3, a2, 2      # calculate 4*i
    add a4, a0, a3      # get address of array element by adding
   base address + 4*i
    sw zero, 0(a4)      # write 0 to memory address
    addi a2, a2, 1      # increment i
    bne a2, a1, looping # continue looping if i < 8
    jalr x0, 0(ra)      # return from eraseBuffer
```

A.7 Postlab 4 Starter Code

A.7.1 Main C File

```
#include <stdio.h>
#include "6190.h"

// Declaration for part 1 of lab.
int collatz(int n);
void bubblesort(int *p, int n);

void setup(){
    timerSetup();
    setupDisplay();
    eraseBuffer(screen_buffer);
    drawBuffer();
}


void arrayViz(int *arr_to_viz){
    int val, val_for_buf;
```

```
18      for (int i = 0; i < 8; i += 1){
19          val_for_buf = 0;
20          val = arr_to_viz[i];
21          for (int j = 0; j < val; j+=1){
22              val_for_buf |= (1 << j);
23          }
24          screen_buffer[i] = val_for_buf;
25      }
26      drawBuffer();
27      eraseBuffer(screen_buffer);
28      long start = millis();
29      while(millis() - start < 50);
30  }
31
32  int testArr[8] = {32, 16, 28, 12, 24, 20, 8, 4};
33
34  void app_main() {
35
36      setup();
37      srand(13);
38
39      while(1){
40          arrayViz(testArr);
41          long start = millis();
42          while(millis() - start < 2000);
43          bubblesort(testArr, 8);
44
45          // Restarts automatically
46          testArr[0] = rand() % 32;
47          testArr[1] = rand() % 32;
48          testArr[2] = rand() % 32;
49          testArr[3] = rand() % 32;
50          testArr[4] = rand() % 32;
51          testArr[5] = rand() % 32;
52          testArr[6] = rand() % 32;
53          testArr[7] = rand() % 32;
```

```
54
55          start = millis();
56          while(millis() - start < 2000);
57      }
58  }
```

A.7.2 Bubble Sort Assembly File

```
1  .section .text
2  .align 2
3  .globl bubblesort
4
5  bubblesort:
6    # a0: first memory address of array (DO NOT MODIFY)
7    # a1: length of array
8
9    # YOUR CODE HERE!
10
11   jalr x0, 0(ra)
```

A.8 Lab 5 Starter Code

A.8.1 Main C File

```
1  #include <stdio.h>
2  #include "6190.h"
3  #include <stdlib.h>
4
5  uint32_t temp_buffer[8]; //temporarily holds new board
6
7  void initializeLoaf(uint32_t* gb){
8    gb[0] = 0;
9    gb[1] = 0;
10   gb[2] = 0x3 << 15;
11   gb[3] = 0x9 << 14;
```

```
  gb[4] = 0x5 << 14;
  gb[5] = 0x1 << 15;
  gb[6] = 0;
  gb[7] = 0;
}

void initializeGlider(uint32_t* gb){
  gb[0] = 0;
  gb[1] = 0x7 << 1;
  gb[2] = 0x1 << 3;
  gb[3] = 0x1 << 2;
  for (int i = 4; i < 8; i++){
    gb[i] = 0;
  }
}

// Setup for random value generator
#define RANDOM_VALUE_ADDR 0x600260B0
#define RTC_CNTL_CLK_CONF_REG 0x60008070
void setupRandom(){
    int *rtc_cntl_clk_conf_rg = (int*) RTC_CNTL_CLK_CONF_REG;
    *rtc_cntl_clk_conf_rg |= (1 << 10);
}

void initializeRandom(uint32_t* gb){
    int *r = (int *) RANDOM_VALUE_ADDR;
    for (int i =0; i<8; i++){
        gb[i] = *r;
        long start = millis();
        while(millis()-start < 100);
    }
}

void updateBoard(uint32_t *current_board, uint32_t *new_board);

void app_main() {
```

```
    timerSetup();
    setupRandom();
    setupDisplay();
    eraseBuffer(screen_buffer);
    const int TIME = 100; //loop time.
    // Choose an initializer:
    initializeLoaf(screen_buffer);
    // initializeGlider(screen_buffer);
    // initializeRandom(screen_buffer);
    while(1){
        long start = millis();
        while(millis()-start < TIME); //wait for TIME milliseconds
        drawBuffer();
        updateBoard(screen_buffer, temp_buffer);

        // copy game_board back into screen_buffer
        for (int i=0; i<8;i++){
           screen_buffer[i] = temp_buffer[i];
        }
    }
}
```

A.8.2 Game of Life Assembly File

```
.section .text
.align 2
.globl updateBoard

# updateBoard
# ARGUMENTS a0: screen buffer (current board), a1: temporary output
   buffer (for new board)
# RETURNS: Nothing
updateBoard:

    # TO-DO: Translate the C implementation of updateBoard to
   assembly.
```

```
    # Make sure to follow RISC-V calling convention!

    ret

# getPixel
# ARGUMENTS a0: screen_buffer, a1: x, a2: y
# RETURNS: 1 if cell occupied, 0 otherwise
getPixel:
    # your code here
    slli a2, a2, 2  # a2: 4 * y for address
    add a2, a2, a0  # a2: screen_buffer + 4*y
    lw a3, 0(a2)    # a3: screen_buffer[y]
    srl a3, a3, a1  # a3: screen_buffer[y] >>_l x
    andi a0, a3, 1  # a0: a3 & 1
    ret

# checkNeighbors
# ARGUMENTS a0: game_board, a1: x index, a2: y index
# RETURNS: total occupied cells in the eight surrounding cells of (x,y) (game board wraps in x and y)
checkNeighbors:
    # TODO: Determine left, right, up, and down indices for neighbors.

    call tallyNeighbors # tallyNeighbors(game_board, x, y, left, right, up, down)

    # TODO: Return result of tallyNeighbors

    ret

# tallyNeighbors
# ARGUMENTS a0: game_board, a1: current x index, a2: current y index, a3: left neighbor index,
# a4: right neighbor index, a5: up neighbor index, a6: down neighbor index
```

```
42 # RETURNS: total occupied cells in the eight surrounding cells of
       current (x,y)
43 tallyNeighbors:
44     # TODO: This procedure is functionally correct, but doesn't
       follow RISC-V calling convention.
45     # Make this procedure follow calling convention. You may only
       add instructions that:
46     # 1. Increment/decrement the stack pointer
47     # 2. Put elements on the stack
48     # 3. Take elements off the stack.
49
50     mv s0, a0 # s0: game_board
51     mv s1, a1 # a1: x
52     mv s2, a2 # a2: y
53     li s3, 0  # s3: tally
54
55     mv a1, a4
56     mv a2, a5
57     call getPixel # getPixel(game_board, x-1, y-1)
58     add s3, s3, a0
59
60     mv a0, s0
61     mv a1, a4
62     mv a2, s2
63     call getPixel # getPixel(game_board, x-1, y)
64     add s3, s3, a0
65
66     mv a0, s0
67     mv a1, a4
68     mv a2, a6
69     call getPixel # getPixel(game_board, x-1, y+1)
70     add s3, s3, a0
71
72     mv a0, s0
73     mv a1, s1
74     mv a2, a5
```

```
75      call getPixel # getPixel(game_board, x, y-1)
76      add s3, s3, a0
77
78      mv a0, s0
79      mv a1, s1
80      mv a2, a6
81      call getPixel # getPixel(game_board, x, y+1)
82      add s3, s3, a0
83
84      mv a0, s0
85      mv a1, a3
86      mv a2, a5
87      call getPixel # getPixel(game_board, x+1, y-1)
88      add s3, s3, a0
89
90      mv a0, s0
91      mv a1, a3
92      mv a2, s2
93      call getPixel # getPixel(game_board, x+1, y)
94      add s3, s3, a0
95
96      mv a0, s0
97      mv a1, a3
98      mv a2, a6
99      call getPixel # getPixel(game_board, x+1, y+1)
100     add s3, s3, a0
101
102     mv a0, s3
103
104     ret
```

A.9 Postlab 5 Starter Code

A.9.1 Main C File

```
#include <stdio.h>
#include "6190.h"

void quicksort(int *p, int start, int end);

void setup(){
    timerSetup();
    setupDisplay();
    eraseBuffer(screen_buffer);
    drawBuffer();
}


void arrayViz(int *arr_to_viz){
    int val, val_for_buf;
    for (int i = 0; i < 8; i += 1){
        val_for_buf = 0;
        val = arr_to_viz[i];
        for (int j = 0; j < val; j+=1){
            val_for_buf |= (1 << j);
        }
        screen_buffer[i] = val_for_buf;
    }
    drawBuffer();
    eraseBuffer(screen_buffer);
    long start = millis();
    while(millis() - start < 100);
}

int testArr[8] = {32, 16, 28, 12, 24, 20, 8, 4};

void app_main() {

    setup();
    srand(60004);

```

```
37      while(1){
38          arrayViz(testArr);
39          long start = millis();
40          while(millis() - start < 2000);
41          quicksort(testArr, 0, 7);
42
43          // Restarts automatically
44          testArr[0] = rand() % 32;
45          testArr[1] = rand() % 32;
46          testArr[2] = rand() % 32;
47          testArr[3] = rand() % 32;
48          testArr[4] = rand() % 32;
49          testArr[5] = rand() % 32;
50          testArr[6] = rand() % 32;
51          testArr[7] = rand() % 32;
52
53          start = millis();
54          while(millis() - start < 2000);
55      }
56  }
```

A.9.2 Quicksort Assembly File

```
1  .section .text
2  .align 2
3  .globl quicksort
4
5  quicksort:
6
7    ret
8
9
10 partition:
11
12   ret
```

Bibliography

[1] "2022 Curriculum Transition." https://www.eecs.mit.edu/academics/undergraduate-programs/curriculum/2022-curriculum-transition, 2022. Accessed: 2023-05-02.

[2] M. Gardner, "The fantastic combinations of John Conway's new solitaire game "life"," *Scientific American*, vol. 233, no. 4, pp. 120–123, 1970.

[3] "MIT Subject Descriptions 2022-2023." http://catalog.mit.edu/archive/mit-subjects-22-23.pdf, 2022. Accessed: 2023-05-06.

[4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language, 2nd Edition*. Pearson, 1988.

[5] A. Waterman and K. Asanovć, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. RISC-V Foundation, May 2017.

[6] "pinMode()." https://www.arduino.cc/reference/en/language/functions/digital-io/pinmode/. Accessed: 2023-05-12.

[7] "digitalWrite()." https://www.arduino.cc/reference/en/language/functions/digital-io/digitalwrite/. Accessed: 2023-05-12.

[8] "digitalRead()." https://www.arduino.cc/reference/en/language/functions/digital-io/digitalread/. Accessed: 2023-05-12.

[9] Espressif Systems, *ESP32-C3 Technical Reference Manual*, pre-release v0.7 ed., December 2022.

[10] D. Hepper, "8x8 monochrome bitmap fonts for rendering." https://github.com/dhepper/font8x8/blob/master/font8x8_basic.h, 2012. Accessed: 2023-05-02.

[11] "CAT-SOOP." https://catsoop.org/website. Accessed: 2023-05-08.

[12] A. J. Hartz, "CAT-SOOP: A tool for automatic collection and assessment of homework exercises," Master's thesis, Massachusetts Institute of Technology, 2012.

[13] A. Lydike, "RiscEmu - RISC-V (userspace) emulator in python." https://github.com/AntonLydike/riscemu, 2021. Accessed: 2023-05-02.

[14] "Calculating instructional units." https://registrar.mit.edu/sites/default/files/2018-12/calculating_instructional_units.pdf. Accessed: 2023-05-06.