Processable: A Visual Assembly Debugger and Program Tracer in the Browser

Robert March Whitaker

A Thesis
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Bachelor's of Arts

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Robert Dondero

June 2018

This Thesis represents my own work in accordance with university regulations

Robert March Whitaker

Abstract

We present a web application for debugging and tracing programs written in x86-64 assembly. Given that assembly programming is notoriously difficult for students to learn, this visual environment aims to provide students with intuition for the behavior of programs at a lower level of abstraction than similar tools designed for tracing high level languages. We also present a javascript library for working with fixed-width integers and emulating ALU behavior, and an extensible assembly emulator (also in javascript), that were developed during this project in support of the visual debugger. By keeping the entire project in javascript, the application can be run without a webserver, and once hosted, can be accessed by a student with no installation.

Acknowledgements

This work would not have been possible without the support of a huge number of people, only some of whom I am able to include here.

First of all, I would like to thank my advisor, Bob Dondero, for his unwavering support over the last year and half, for allowing me to take on an ambitious project, and for continuously reminding me to leave time for writing. I'd also like to thank Iasonas Petras and Donna Gabai for volunteering to be guinea pigs for the first version of my project, and for all their helpful feedback, and Professor Szymon Rusinkiewicz for letting my project further tested on COS217 students this Spring. I also owe some thanks indirectly to Professor Andrew Appel, for pushing for the switch of COS217 to ARM, keeping me on my toes and forcing my project to become more modular and extensible.

I want to thank my friends for bearing with me through this whole project, and listening to me go on about debuggers for inhumane amounts of time. Thanks to Josh Becker, Eric Mitchell, and Lukas Novak for looking at early versions of the project, and giving me great feedback and encouragement. Thank you to Terrace F. Club, putting a roof over my head and feeding me half the time, and to the Pink House for adopting me and giving me food and shelter the other half, and making me realize that while food is love, radish butter is better than both. And to Anna Maritz, for being my light at the end of the tunnel and guiding star, bringing me coffee and kombucha all hours of the day, and listening to me rant about computers for hours without holding it against me.

Finally, I would like to thank my parents, for raising me to work hard, stay humble, and remember to take a break sometimes.

To my teachers

Contents

List of Figures

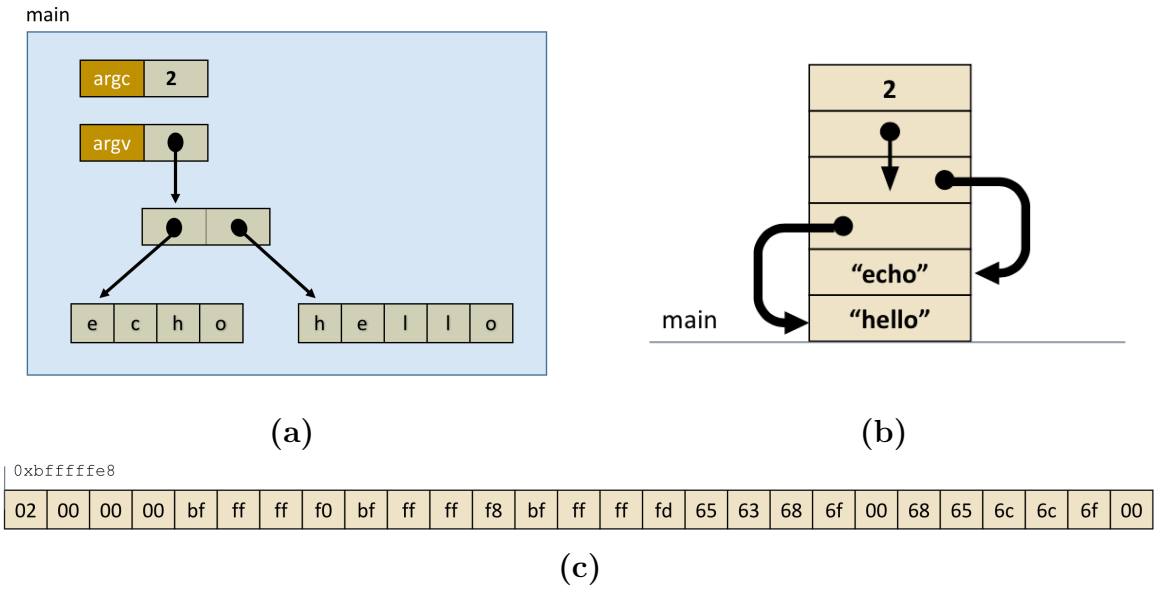
Chapter 1

Introduction

Programming is difficult[27]. Assembly programming is especially difficult, and presents so many potential pitfalls that the entire computer science community has largely abandoned it in favor of structured programming in high level languages. Flat control structures such as the `goto` or `jump` statement, have been derided as harmful[13], however it is still the case that most all computer architectures rely on an instruction set that *requires* such flat programming. As such, writing code at the lowest level presents an inherently difficult problem, and this is difficulty is especially apparent to students as they learn their first assembly language. As the practice is mostly reserved for systems experts and operating systems developers, many of the resources available assume an expert audience. Architecture manuals are extremely dense, and describe chip behavior without developing intuition for how one would compose an assembly program.

For high level languages and assembly languages alike, one of the best teaching tools in the instructor's toolbelt is the *program trace*. That is, one of the most effective ways of conveying programming intuition for any language is to visualize the state of the program's runtime environment at various critical points during execution. Despite program behavior usually being deterministic, the process of tracing

Figure 1.1: Snapshots of program traces at various levels of detail



is most often done ad hoc, by hand at the blackboard. We rely on an instructor's understanding of a program to translate its static specification in code to its dynamic execution as a process. A program trace can be performed in varying levels of depth from a very abstract box and arrows diagram like that in Figure 1.1(a), to a more realistic layout which suggests more about the arrangement of the data in memory as in Figure 1.1(b), to a literal dump of each of the program's bytes in order, Figure 1.1(c).

As useful as the program trace is, it has been found that students often struggle with the ability to generate a trace of their own. Further, being able to correctly trace a program has been shown to be directly correlated with performance on a number of different types of problems[32].

In introductory computer science classes, generating these traces is a task that instructors must perform repeatedly by hand. To make traces that can be reused, often instructors must hand draw a representation of a process at each step in execution, or even more painstakingly generate a series of images by hand or using a presentation tool. For the sake of both novice programmers and computer science instructors, we would like to simplify the process of generating these program traces.

Specifically, this is an application that, given a process in a particular state, can generate a detailed graphical representation of that process. The natural extension of this tool would also be able to interact with the process as a debugger's inferior, and allow the user to set breakpoints, step, and continue the process, all while being able to inspect the visual representation of the program's state.

It is interesting that this task is not so easily automated, despite the fact that the task of mapping a program to a process is exactly what the machine performs whenever a program is run. The problem lies in the fact that the goals of teaching programming and running programs are often in direct conflict with each other. Engineers typically want programs to execute as quickly as possible, without requiring input from users. After all, one does not need to verify each step of a correctly implemented program every time it is executed. Students and educators on the other hand are typically interested in slowing programs down and inspecting their state in order to better understand them. As such, program visualization is an area of active research, and in this paper we will consider it from a number of angles. Many successful program visualizers have been developed already: from simple but incredibly popular tools like `pythontutor.com`, which give box-and-arrows traces of high level programming languages in execution, to such ambitious projects as Luna: a functional programming language in which every program maps one-to-one with a visual representation[3]. However the discipline has also received plenty of criticism, even by such notable figures as Dijkstra:

> I was recently exposed to... what.., pretended to be educational software for an introductory programming course. With its "visualizations" on the screen, it was.., an obvious case of curriculum infantilization .... We must expect from that system permanent mental damage for most students exposed to it.[14]

In order to avoid similar criticisms, we take a methodical approach to developing a program tracer and visual debugger which we name *Processable*. The structure of

this paper is as follows: first, we clearly define the problem we are trying to solve, and identify the concepts we would like to teach, and the properties the final application should have. Then we review background material related to debuggers and emulators, and investigate some similar tools to identify strengths and weaknesses. We then present the *Processable* application, its use cases, features, and limitations, before engaging in a more thorough exploration of its implementation. Finally, we evaluate the success of the project on a number of metrics, and discuss opportunities for future work.

1.1 Problem Definition

What we identify here is a gap in available resources for students for visualizing low-level computer behavior. While significant progress has been made in flexible tools for visualizing high-level programming concepts, the same is not true for systems-level and assembly programs. Given the success of these tools in contributing to students' understanding of structured programs and algorithms, we are motivated to develop an equivalent application that can be used for tracing assembly programs.

To better define the problem we aim to solve, we would like to enumerate the specific concepts that students are being taught by learning an assembly language that are distinct from those taught in higher level languages. We still draw on the principle in computer science education that *concepts* are much more important than any particular language or environment. While narrowing the focus to assembly language seems contradictory to this goal, we note that in addition to their particular syntax, assembly languages also introduce a number of new ideas that are shared between architectures and syntaxes. In programming courses that use an assembly language for teaching, the focus remains on the following low-level and systems concepts:

1. **Addresses and virtual memory areas**: One of the key distinctions between assembly and higher level programming is the explicit use of data addressing, and the separation of data into different virtual memory areas with different runtime properties and permissions. Accessing global data in statically allocated `.data`, `.rodata`, and `.bss` sections are often not distinguished in higher level languages, but must be explicitly specified in an assembly program, and each section has distinct permissions and properties.

2. **Type-agnostic data**: Whether a high level language is statically typed or dynamically typed, there is still the property that a piece of data *exists* in a particular type, and can be manipulated in ways specific to that type. Integers can exist as signed or unsigned, but these cannot be added together, nor can either be added to strings. Asking if the character `'a'` is less than the integer `-1` is bound to draw some complaints from a compiler worth its salt. However in a lower-level view of the program, the distinction does not come from what a variable *is*, but how it is *interpretted*. In an ASCII character encoding scheme on a little-endian system, the 32-bit 2s-complement integer 2,189,672 has the same representation as the null-terminated string `"hi!"`.

3. **Flat control structures**: Another convenience that is lacked by assembly programs are complex control structures, such as *if-else blocks*, *do-while loops*, and *switch statements*. Instead, flow of control is implemented entirely as jumps, which may depend on certain conditions about the running process. This type programming is so riddled with the potential for bugs that the programming world has discouraged the inclusion of similar control structures in high level languages at all.[13] While this luxury is available to high level languages, they still must define the behavior of the easier-to-reason-about structures in terms of a single stream of execution from one instruction to the next.

4. **Function call mechanics**: Finally the low-level details of how functions are called are only exposed at the assembly level. Specifically, this is the pushing of return addresses onto the stack at a function call, and the lack of robustness of this behavior to malicious input via buffer overflow. The tool should help to describe such subtleties in function calling, and how use of stack and base pointer registers establish each call frame.

Returning to the example program traces of Figure 1.1, intuition for these concepts can be built by finding a way to combine traces of different levels of detail. The task of learning assembly language from a higher level language like C, is the process of adding detail to principles of the higher level language, and learning the relationship between traces like Figure 1.1(a-b), to that of Figure 1.1(c).

In designing a system for program visualization, we also look to the literature to identify properties of successful visualization tools. Specifically, the reviews of Sorva[40, 41] and reports from working groups at the ACM Conferences on Innovations and Technology in Computer Science Education (ITiCSE)[35] lead us to identify the following goals for the behavior of the application itself.

1. **Flexible Input**: For a tool to be useful to students in the process of learning new computer science concepts, it is important that they be able to customize their usage with the ability to upload arbitrary programs (subject to certain restrictions of the platform).

2. **Interactive**: It has been shown[36] that simply providing visualization alone does not contribute significantly to student understanding, but that students learn better from interacting with a visual representation. In addition to allowing the student to customize the input, the tool should be able to respond to user interaction throughout the visualization process.

3. **Multiple viewing options**: It was similarly found in [36] that providing multiple views on the same data or algorithm was also very helpful for improving student understanding, and so the user should be able to change the way information is presented to best suit their needs.

4. **Available to the widest possible audience**: Finally, a good teaching tool should have an easy or no installation, and have minimal technological requirements, so that it can be readily used by any student who wishes to use it.

Chapter 2

Background and Related Work

The tool presented in this paper is a hybrid of several existing technologies, and here we review the literature surrounding each to draw inspiration from some of their design decisions. Specifically, what we propose is a combination of a debugger and a program visualizer, which is made possible by the use of hardware and software emulation.

2.1 Debuggers

In general, a *debugger* is an interactive process that can be used to control and examine the state of another process, known as the *inferior*, while it is running. This may include setting *breakpoints* to pause execution at a certain location in the text, *watchpoints* to pause when a certain address in memory is accessed, or *single-stepping* a process to pause after every instruction is executed. While the inferior is paused, a debugger allows the user to probe or edit its state, by displaying or altering the contents of different areas of memory.[5] Put another way, debuggers attempt to slow down its inferior, so that a developer can identify the precise moment when an unintended behavior manifests.

However, as most programs are built to run as fast as possible on their target architecture, there is often no native way to dump the state of a program in the middle of execution without editing the source to do so. Especially for compiled languages, there is no way to take control of a process between instructions to check if a breakpoint exists at the current address, or provide the user an opportunity to inspect the program state. Production debuggers solve this problem by taking advantage of specific system calls provided by the operating system which allow a parent process to "attach" to its child, and then edit its text section while it runs.[1]

The process of setting a breakpoint in a compiled program running under a debugger is incredibly complicated, and requires extensive bookkeeping by the debugger, and significant operating system support. Before any breakpoints can be set, the inferior must make a system call giving its parent permission to trace it. [2] When a breakpoint is to be set at a particular address, the debugger will write over the instruction at that address with a special one-byte trap instruction, making use of a *poke*[9] system call. As the debugger and inferior each have separate virtual address spaces, the actual *poking* (editing of memory) must be done in the kernel, where the virtual address of in the inferior can be translated to a physical address. The special trap instruction must be no longer than a single byte, so as not to overwrite subsequent instructions as well, and the operating system must provide a special handler for this trap which knows how to return control back to the parent debugging process.[3] With the instruction at the breakpoint overwritten, the inferior can be allowed to execute on its own, until the trap is hit, at which point the kernel takes control,

[1]This is the `ptrace` family of system calls in Unix[30]. While this is not the only way that a debugger can be built, it is by far the most popular, and is the only feasible way to take control of another process without a heavily customized operating system.

[2]This is accomplished with the `PTRACE_TRACEME` request as the first argument to `ptrace`[30]. This can be accomplished within the debugger after the child is forked from the parent, but before it `exec`s the inferior

[3]On x86-64, this is the `INT 3` instruction, known only as the "trap interrupt", and is encoded by the one-byte opcode `0xCC`.[39] As most interrupts are two bytes, (`0xCD 0x??`), this illustrates that debuggers require cooperation from the *hardware* as well as the operating system.

recognizes that the process is being debugged, blocks the inferior and finally schedules the debugger.

We mention the complexity of debuggers only to contrast it with *Processable*. In particular, the difficulty of implementing debugging functionality on native code assembly code is prohibitive, and so we are driven to consider hardware *emulation* as an alternative platform on which to build a program visualizer.

2.2 Emulation

As a solution to some of the barriers to debugging posed by real systems, virtual computer systems offer much more fine-grained control over an execution environment. While software emulation incurs a major slowdown, it offers significantly more flexibility than native hardware execution. This flexibility can be summarized by two major advantages offered by emulation: *portability*, and *customizability*. Emulators can be used to run a program compiled for a particular target architecture on another that may lack the necessary hardware. Additionally, replacing physical hardware behaviors with programmed behavior in software makes it much easier to edit or augment the capabilities of the hardware being emulated, for example by adding hooks that can respond to certain events that occur in the emulated environment. In the context of this project, an emulator could have built-in debugging capabilities, that allow it to manage breakpoints or watchpoints without requiring operating system and hardware support.

2.2.1 Qemu and Unicorn

One of the gold standards in hardware emulation is the open-source Qemu software, originally written by Fabrice Bellard. Qemu gives near-perfect emulation of a variety of architectures, including x86 (and x64), ARM, SPARC, MIPS, PowerPC and

others, supporting software virtualization of serial and parallel ports, memory management units, and drives in addition to the CPU [8]. One of its primary use cases is for debugging systems software (or an entire operating system), without requiring a reboot of the host machine. Qemu does not however support built-in debugging, and one must still use something like `gdb` to remotely trace the process running on the emulator.

While Qemu is a remarkable *application*, it does not serve well as a *framework* for developing further applications on top of it. This can be thought of as a side-effect of Qemu's accuracy in emulation; there are no side-channels for hooking into a running process on Qemu, and its fundamental goal is to be a "FAST!" emulator[8]. To answer this shortcoming, the developers Anh Quynh Nguyen and Hoang Vu Dang created the Unicorn Framework, by stripping all but the CPU emulation modules from Qemu, and enhancing it with a set of API bindings (in more than 10 popular programming languages) for manipulating the instruction emulator, without requiring a full hardware context to be instantiated[37]. To illustrate this difference, single-stepping a process in Qemu is almost identical to single-stepping a process on the target architecture: the virtual hardware trap flag would be set, causing each instruction to trigger an exception, which would call a handler in the kernel and eventually pass control to the debugger tracing the process. Using the Unicorn framework, one explicitly sets start and end addresses for emulation with the Unicorn API, and control remains in the hands of the emulating application. While both Qemu and Unicorn are tools for improving software portability, they must in turn be built for the host architecture, so users must either install a precompiled binary, or build them from scratch.

2.2.2 Emscripten and Unicorn.js

In the world of more unorthodox emulation environments, there is the Emscripten compiler and software development kit, which aims to provide a similarly flexible ex-

ecution environment for the web[43]. Unlike Qemu or Unicorn, which sit between the host machine and the emulated code at runtime, Emscripten is invoked at compile-time, and generates an program with equivalent behavior *in javascript*. Emscripten works by defining its own virtual machine written in a subset of javascript known as asm.js[24], and aims for equivalent behavior without necessarily virtualizing any specific hardware. Rather than taking a program compiled for a particular architecture, and running it on Emscripten, one uses Emscripten to compile the program's source to javascript.

Emscripten doesn't quite fit our definition of emulator, as it solves the portability problem (even better than Unicorn and Qemu) but doesn't give much more flexibility over a program's execution on the Emscripten VM. But it does provide an ability to *retarget an exiting emulator for javascript*, and combine the extreme portability offered by javascript to the flexibility provided by unicorn. This is precisely what Alexandro Bach accomplished with Unicorn.js[6], a javascript port of Unicorn using Emscripten and is an inspirtation for and close relative to the emulator we present in this paper. However, Unicorn.js brings with it a few downsides: to start, even the minified javascript bundle for Unicorn.js is over 20MB, around 40 times larger than the 90th percentile among the popular sites in recent years[21]. With much of the world still receiving download speeds under 10 Mbps [16], this would impose a 15 second or longer page load, which is unacceptable. Additionally, the Emscripten compilation process makes the project impossible to maintain from pure javascript, and the Unicorn source is sufficiently complex that reading it is does not make for an educational experience *per se*.[4]

[4]This is not to say that Unicorn.js is not an incredible piece of software, and we will explore other uses for it in Chapter 6. The goal of the project's source being an additional educational resource will be developed further in the discussion of the implementation in Chapter 4.

2.3 Visualization Tools

2.3.1 Evaluation and Classification of Visualization Tools

Visualization tools have frequently been a subject area for papers presented at the ITiCSE, and in addition to a rich collection of visual applications from which we can draw inspiration, a number of papers and working group reports have appeared which provide frameworks for evaluating software visualization techniques.[1] Further, papers as early as those of Myers in 1990[34] have provided taxonomies for classifying visual applications, so that we may be precise in comparing and contrasting our proposal from existing tools in the field. In particular, we can distinguish between *Visual Programming* and *Program Visualization*. The former is defined by a programming environment that uses visual elements to *define* program behavior. Updating some of the examples used by Myers for 2018, we could categorize as Visual Programming some domain-specific languages like Max/MSP[4] or PureData[31] which provide visual interfaces to digital signal processing for music producers, as well as the recently released functional language Luna[3] which boasts an isomorphic visual and textual representation of every program, and a custom visual IDE.

This project however is more concerned with the other taxonomy presented by Myers, of *Program Visualization*. This type of system is not concerned with creating new software, but in augmenting existing software with visualizations with the aim of improving understanding or the debugging experience. In the words of Myers,

> Another motivation for using graphics is that it tends to be a higher-level description of the desired actions (often deemphasizing issues of syntax and providing a higher level of abstraction) and may therefore make the programming task easier even for professional programmers. This may be especially true during debugging, where graphics can be used to present much more information about the program state (such as current variables and data structures) than is possible with purely textual displays.[34]

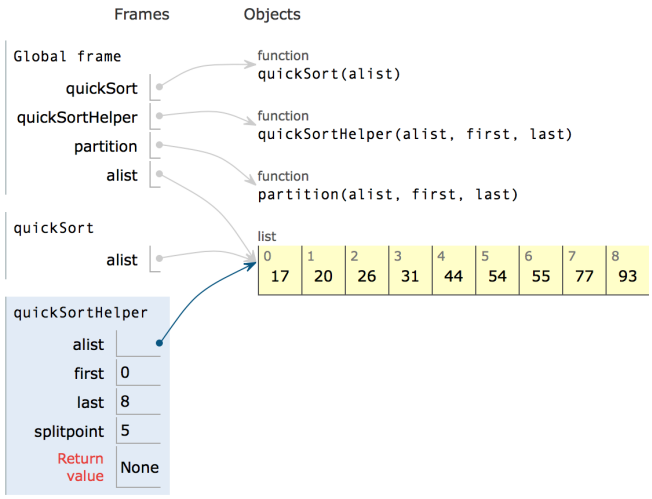
The field of program visualization can be further broken along two dimensions into *static* and *dynamic* visualizers, and visualizations targeted to *code*, *data*, or *algorithms*. The first categorization distinguishes between tools that create images illustrating program state offline, or in a batch fashion, between those that can illustrate program state as the target program is executing, with the opportunity to possibly edit the program's trajectory after visualization has begun. The former restriction may arise from a computational difficulty in generating the desired visualizations in a reasonable amount of time (a problem that has become less pressing in the nearly 30 years since Myers' paper), or from an incompatibility between the environment that executes the program and the one that produces the visualization. The second dimension for categorizing program visualization systems distinguishes between *code* visualization, which attempts to illustrate the logic and flow of control in a program, *data* visualization, which illustrates meaningful aspects of a process' state (i.e. memory), and *algorithm* visualization, which takes a high-level view of a particular concept and attempts to animate its behavior at a level of abstraction above a specific programming language.

The type of tool we propose in this paper would be classified in this taxonomy as a *Dynamic Data Visualization*, and these are the types of visual tools that we evaluate here.

2.3.2 Tools focusing on High Level Languages

The most successful tool in this area by far has been Phillip Guo's Python Tutor[22], which since its launch in 2015 has expanded to support similar visualizations for Java, Javascript, C, C++, and Ruby (as well as Python 2 and 3). The general architecture is as follows: the user types code in the selected language into an online editor and submits with a "visualize execution" button. The code is then sent to the server, and executed in custom sandbox under the supervision of a debugger, with

Figure 2.1: **(a)** A screenshot of the Python Tutor visualizer output, in the middle of executing an implementation of quicksort. **(b)** The structure of the trace object returned from the Python Tutor server



(a)

(b)

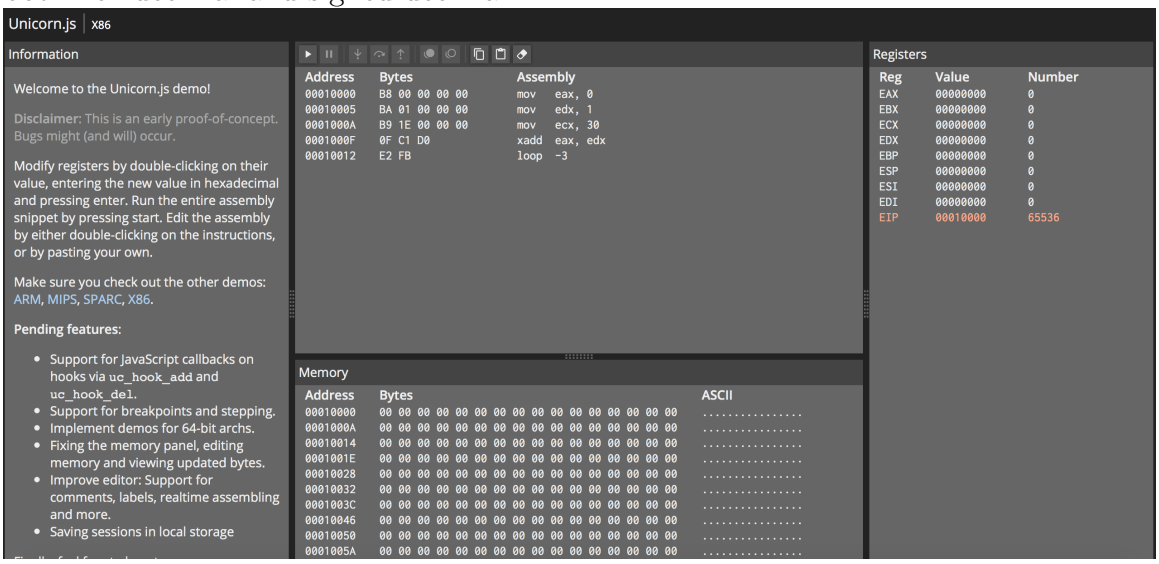
information about the program's state logged after every line is executed. A structure containing this state for each step is then returned to the client, and each is used to generate a boxes-and-arrows visualization for that point in the program's execution (see Figure 2.3.2). While one can step through the program once it has been run, and flip through each illustration one at a time, it is important to note that the entire program must be run before any visuals can be displayed. This provides the user the ability to easily step backwards through a program if desired, but also means that no program state can be updated during execution, purely as result of this design.[5] There also does not appear to be a way to supply the program with command line arguments, and reads from stdin require an additional round trip to compute the rest of the program after reading input. Additionally, the sandbox imposes a maximum of 300 lines of code before it will automatically stop executing[23].

[5]While *Processable* does not currently support editing registers and memory during execution, its runtime environment *does*, and so allowing this support would be a simple change to the interface. In contrast, Python Tutor cannot support state manipulation without changing its execution model.

It seems that this tool is better classified as a static visualizer, as even the "live code execution" mode requires full server round-trips that must run the entire program and send back the same data structure structure shown in Figure 2.3.2**(b)**.[6] At the same time, with the limit on program length, the backend is able to execute each and regain responsiveness with minimal delay, and the ability to step backwards in time is surely a helpful pedagogical tool.

2.3.3 Tools Focusing on Assembly Language

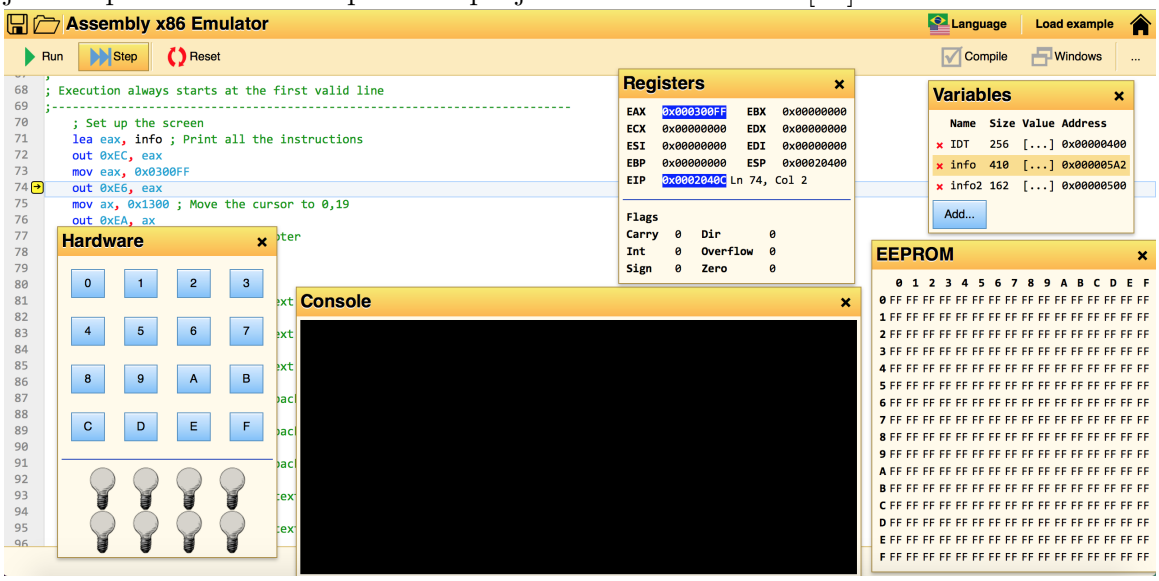
Figure 2.2: Screenshot of a web-based x86 assembly visualizer backed by Unicorn.js[6]. The contents of the registers and memory are editable, but the buttons controlling program flow are all unimplemented. Memory presented as a single continuous chunk, and is decoded only as ASCII; register values are displayed in their 32-bit form in both hexidecimal and signed decimal.



This project is not the first to attempt to apply the success of high level visualization tools to lower level programming, but no others appear to have had success on the level of Python Tutor. One visual debugging tool (shown in Figure 2.2) is built on

[6]It would be interesting to attempt to replicate this system entirely on the client side, by using web-enabled runtimes generated written in javascript (such as Brython[38]) or compiled to javascript with Emscripten, such as pypyjs[29]. This would enable the tool to use dynamic visualization, and perhaps remove some of the limitations of the sandbox on the server side (such as the lines of execution maximum), as well as allow users to edit variable values online, as the program is being visualized.

Figure 2.3: Screenshot of another web-based x86 visualizer, based on a custom javascript runtime developed in a project known as asm86.[12]



top of Unicorn.js, written by Alexandro Bach. Though it appears to be an abandoned project, it shows some promise as a web front-end for an object code interpreter, with a built-in disassembler to show corresponding assembly mnemonics. While its github stars suggest there is certainly interest in this type of tool, it fails to produce more intuitive traces of assembly programs or provide support for customizable decoding, nor does it distinguish between static memory areas, the stack, or the heap. The same front-end is also implemented for MIPS, SPARC, and ARM, though they are all similarly unfinished, and as mentioned above in our discussion of Unicorn.js, they all have substantial loading times.

Another low level visualization tool written by Carlos Neves[12] is similar in scope, but emulates machine behavior with an assembly language interpreter (as does *Processable*). This tool allows users to write assembly language into the browser by hand, or load one precomposed example which handles external interrupts. My own experience with the interface felt fairly clunky and unintuitive, as each area other than the editor opens in its own pseudo-window, as shown in Figure **??**. It is also limited to 32-bit x86 assembly with intel syntax, and appears to put more emphasis

on lower level hardware elements, such as EEPROM and interrupt pins. While we investigated the code for this tool as a potential starting point, it is organized in such a way as to not lend itself to extensibility or easy maintanence, using a single monolithic javascript entry point.

While both of the tools discussed above are squarely in the same category as our intent for *Processable* (Dynamic Data Visualization), neither of them satisfied the main goal of providing a strong intuition for the organization of a program in memory, nor did any of the other less popular or relevant visualizers that were investigated but didn't warrent discussion. However, we were accelerated in the re-implementation of the concepts from scrath by basing our tool on the general structure of those presented above, with the freedom to shape the code and visuals in such a way as to meet the needs outlined in the previous chapter.

Chapter 3

Functionality

3.1 Use Cases

The piece of software described in this project is primarily motivated by a single problem: learning assembly language is difficult. Thus all of the potential uses we imagine for it are united under the idea of helping students in the process of learning (their first) assembly language. Beneath this umbrella, we can imagine a number of different specific use cases in support of the goal of computer science education:

- **A student wishes to visualize the execution of a sample assembly program.**

  It remains the case that one of the best ways to become acquainted with a new programming language or paradigm is to inspect simple example programs and trace their execution. This can be done without a visual environment, but visuals help to build intuition, and can expedite the absorbtion of new concepts, as has been found by the creators of Python Tutor[22]. *Processable* ships with four examples assembly programs of different complexity, and together provide a gentle introduction to assembly language concepts.

- **An instructor wishes to present a program trace to a class.**

  Related to the previous case, instructors often wish to present a similar type of walkthrough as part of their instruction. As an alternative to painstakingly composing a visual for each step of a program by hand, or drawing the trace in front of students on a blackboard, *Processable* can be used to present a clean and reproducible instructional trace that can be projected onto a screen during a class.

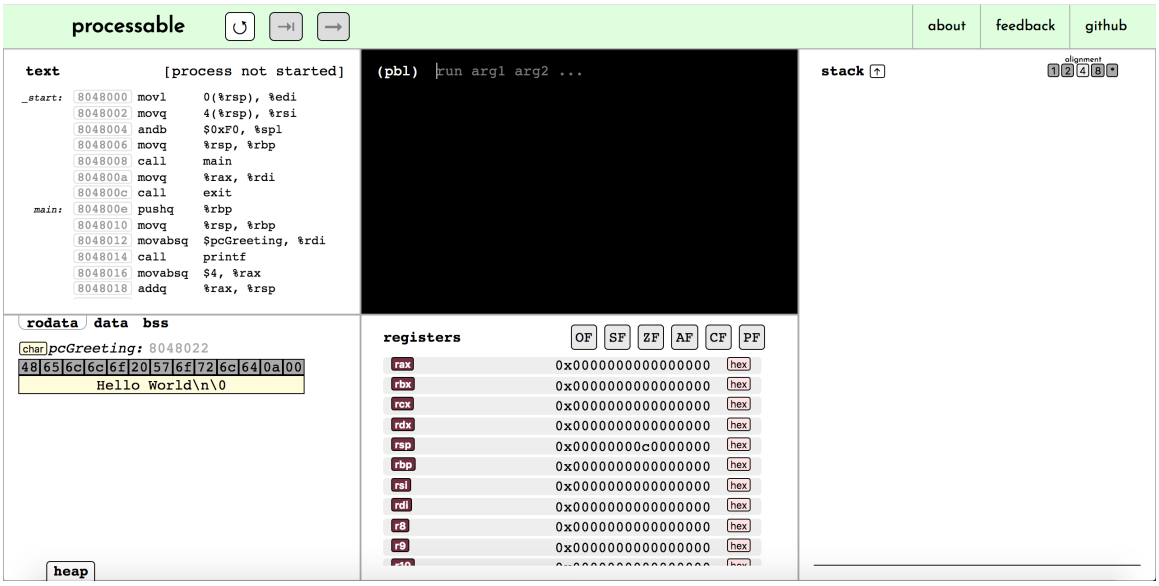
- **A student wishes to debug an assembly program.**

  As the title of this paper suggests, a more specific use case than simply tracing a program to gain intuition is attempting to diagnose a bug in a student's program. Here Processable presents an alternative to terse and complex command-line debuggers such as `gdb` with an intuitive, button-oriented interface for manipulating the inferior process. Further, inspecting program state to determine the source of a bug becomes a matter of simply scanning the UI panel for the relevant area of memory.

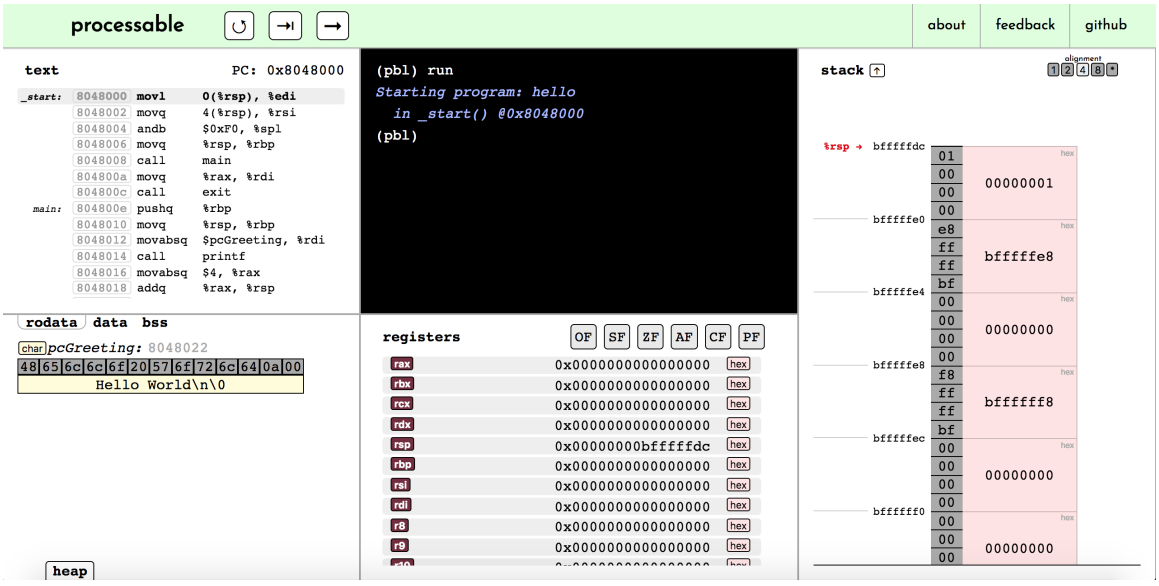
- **A student wishes to visualize a C program at the assembly level.**

  Finally, an ambitious student could apply one of the same processes described above for a C program of their choice, after compiling it to assembly. The general-purpose nature of the emulator on which Processable is built, and the support for integration with the C standard library allow for most of the programming constructs used by student programs to be emulated accurately, and the environment is equipped to deal witht the particulars of compiler-generated code.

Figure 3.1: A series of screenshots of the Processable interface illustrating the process of loading and running a program.
More screenshots of the application can be found in Appendix A



(a) Statically loaded program



(b) Program in execution

3.2 Features

In addressing the assembly language concepts defined in Chapter 1, the final product provides rich functionality for allowing students to engage directly with these ideas.

The flow of a typical user interaction proceeds as follows (see screenshots in Figure 3.1): first, the user is presented with a welcome screen offering them the option to upload an assembly file or load one of the provided examples (Figure A.4). After selecting a file, the application loads the program, and populates the panels dedicated to viewing the static sections of memory (i.e., the `text`, `rodata`, `data`, and `bss` sections), (Figure 3.1(a)). Finally, after beginning the process with the `run` command in the console, the stack is set up with the command-line arguments, and debugging can begin (Figure 3.1(b)). From there, the user can control the inferior process with *Processable*'s button-oriented interface, toggling breakpoints by clicking on addresses in the text section, and stepping or continuing the process with the respective buttons in the toolbar.

3.2.1 Toggleable Decodings

To help convey the concept of type-agnostic data, all of the memory areas provide some method for toggling between decoding options for a particular group of bytes, in addition to a "raw" view of each underlying hex byte wherever possible. The supported decoding formats are hexadecimal, binary, ASCII characters, and signed and unsigned decimal integers. Clicking a decoding group next to any set of bytes in memory will cycle through the available options. These options for the stack are shown in Figure 3.2(a), and for other memory areas in Figures A.1-A.3

On the stack in particular, the size of the decoding groups can be set globally (as 1,2,4, or 8 bytes) or customized to display variables of different sizes on the stack at the same time, as shown in Figure 3.2(b). Additionally, the stack is annotated with the current values of stack and base pointers (`%rsp` and `%rbp`), with labeled arrows pointing to the addresses which they contain.

Additionally, the most recent version of the tool includes support for the heap and dynamic memory allocation. The screen space required to show the heap is quite

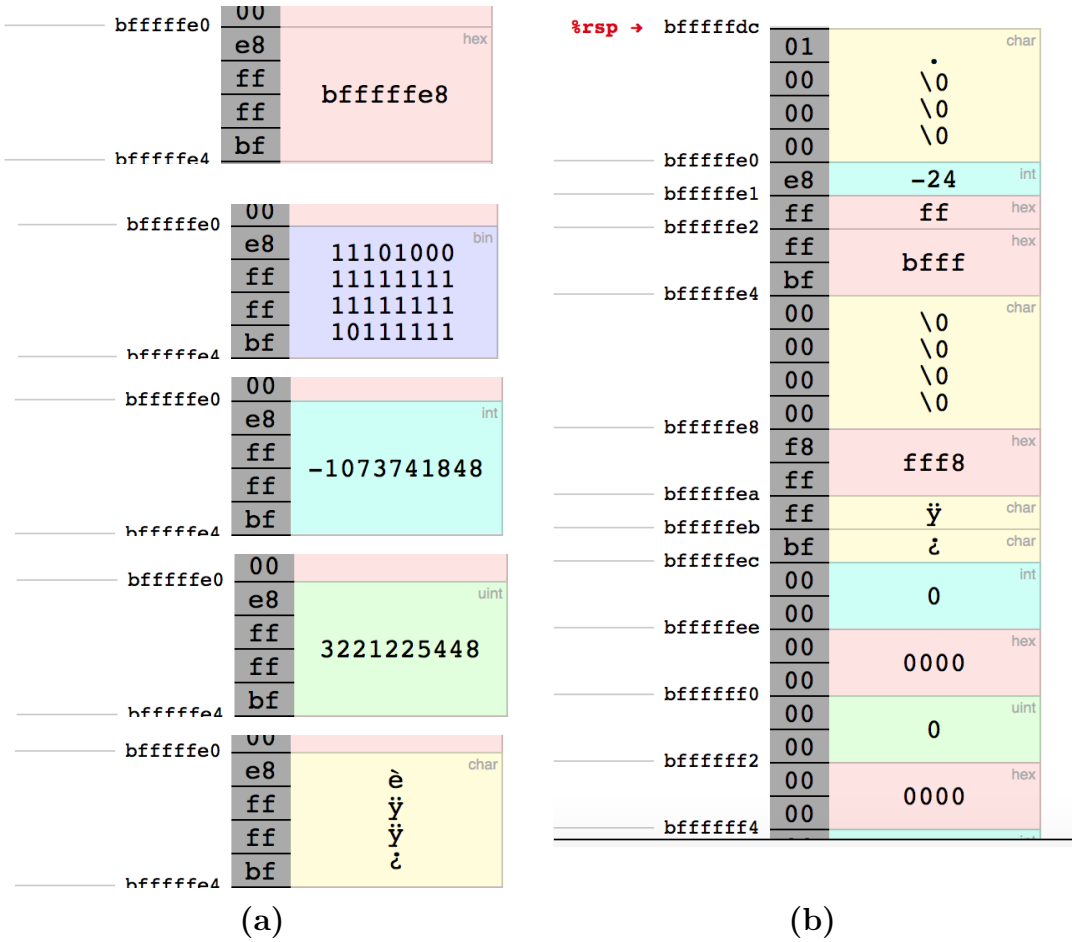


Figure 3.2: Screenshots of the decoding options available for the stack on (a) the same 4-byte region and (b) a larger region with multiple decodings of different sizes

large, and so by default it is hidden. The bottom left of the screen contains a tab for toggling display of the heap, and when it is on, the normal display is condensed to take up half of the screen, and the Heap takes up the rest. The heap supports the same decoding options as the stack, and is shown in the Appendix A in Figure A.5

3.2.2 `gdb` Commands

As a way of bridging the gap between Processable and traditional command-line debuggers, we also have support for a small subset of common `gdb` commands. These commands are entirely redundant to the functionality provided by the debugger's UI, with the exception of the `run` command which must be used to start a process and

provide it with command-line arguments. Those that are implemented include `break`, for toggling breakpoints, `step` for advancing to the next instruction, `continue`, for resuming execution (until the next breakpoint) and `help` for displaying command options.

3.3 Limitations

3.3.1 x86 Instructions

We implmented a minimal but quite usable subset of x86-64, including most control flow and integer arithmetic instructions. Important limitations to note include

- There are no floating point instructions or registers
- No support for the x86 AF (auxiliary / half-carry) and PF (parity) flags
- 64-bit multiplication is not supported (`mulq`, `imulq`)
- No direct support for assembly-level I/O (`in`, `out`)

Despite these limitations, *Processable* supports approximately 97% of the instructions found in production code by frequency of occurence. The full list of supported instructions along with their frequency in production code can be found in Figure 5.1, with accompanying discussion in Section 5.2.1

3.3.2 C Standard Library

A subset of the C standard library has been implemented with javascript functions that produce similar behavior on the emulation environment as their corresponding functions on a linux C runtime. Additionally, variadic functions in the standard library cannot be called with more than six arguments. The list of currently supported functions is presented in Figure 3.3.

Figure 3.3: C standard library functions supported by *Processable*

`printf`[1]	`scanf`[1]	`getchar`	`putchar`
`brk`	`sbrk`	`malloc`[2]	`free`[3]
`atoi`	`abs`	`labs`	`exit`

[1] Supported format specifiers: `%d, %i, %x, %s`

[2] Minimal implementation, simply wraps `sbrk`

[3] Free does nothing (a valid, albeit very wasteful, implementation)

3.3.3 Addresses and Machine code

As this application interprets assembly language directly from its string representation, there is no assembling to machine code, and so addresses in the text section are not accurate. As a convention, all instructions are assumed to be 2 bytes, and the addressing of the text reflects this. As a corollary, machine code injected into memory via a buffer overflow will not be executable.

3.3.4 Runtime

Programs are loaded by something analagous to an `exec()` call by the debugger. This pushes the command-line arguments onto the stack, as well as the `argv` array and finally the pointer to `argv`, and `argc`. Note that there are no environment variables pushed onto the stack, and no corresponding envp array. Additionally, when an assembly file is loaded, the "assembler" injects a minimal C runtime (commonly referred to as crt0.s) which does 4 things: First it copies `argc` and `argv` from the stack into `%edi` and `%rsi` respectively, then it ands the last byte of the stack pointer (`%spl`) with `0xF0`, to force a 16-byte alignment, and calls `main()`. When main returns, it moves the return value `%rax` into `%rdi` and calls `exit()`. Notable omissions include a call to `__libc_start_main`, or other initialization code.

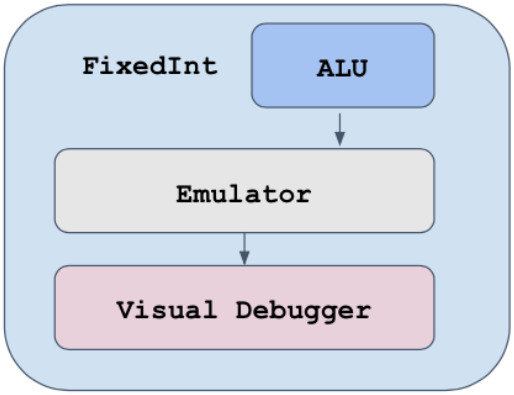
Chapter 4

Implementation

In considering the goals of this software, there were two in particular that led to the implementation described here. First, given the ever-evolving nature of software engineering, and the similar instability of course content, we would like the software to be easily maintained and expanded. Midway through the development process, a movement began within the Princeton CS department to transition the COS217 course from using x86-64 as the architecture of choice to ARM64. While this could have been an existential threat to the usability of this project beyond an academic exercise, we took this as an opportunity to design the application to be as modular and architecture-agnostic as possible. That is, each module should be designed to work with any architecture, with the exception of those features which are unique to it. This led to a more general guiding philosophy that *all* modules should be replaceable if possible.

The second goal which we hope is satisfied by this implementation is to learn about and explore the properties of real systems. That is, in emulating hardware and low-level kernel and systems software, we would like to draw on the concepts that have allowed them to succeed in providing specific functionalities to the end user. As the process of emulating a particular software involves many of the same challenges

Figure 4.1: A high-level overview of the packages that comprise *Processable*



The `FixedInt` package provides an Arithmetic and Logic Unit (ALU) for hardware emulation. The emulator builds on top of that for a full process environment, and the debugger provides an interface to a running process. The `FixedInt` data type pervades the entire application and allows for the passing of fixed-width data between modules.

as writing the original software, maintaining analogies as faithfully as possible is a helpful tool. Further, the structure of this project itself may serve more advanced students as a way of exploring the intricate relationships between computer systems from the comfort of an expressive high-level language, as it has done for me during the course of development.

In accordance with the goals above, the project is divided into three distinct software packages, with cascading dependencies between the three of them. That is, each higher level package only imports from the package(s) below it. The first is a library (which we name `FixedInt`) for working with fixed-width, two's complement integer values, and manipulating them in the style of an Arithmetic Logic Unit (ALU). This forms our "hardware" layer of abstraction, and implements arithmetic and logical operations independently of any particular architecture. The middle layer is the most technologically challenging, and represents a retargetable, debugging-aware process emulator. Finally, on top of these is a GUI application that serves as a visual debugger, taking advantage of public methods exposed by the emulator to control the process being executed.

Drawing inspiration from the Emscripten virtual machine, the fundamental javascript data structure that makes accurate emulation possible is the `ArrayBuffer`.

The `ArrayBuffer` is just a thin wrapper around the host platform's native `malloc`, and simply returns to the user a pointer to a chunk of memory of the requested number of bytes[17]. There are several wrapper objects for reading values of different types from an `ArrayBuffer` including: `Uint8Array, Uint16Array, Uint32Array, Int8Array, Int16Array, Int32Array,` and `DataView`.[1] All but the latter behave as normal arrays, interpreting the underlying data as the type in the name. The `DataView` is a wrapper that allows one to read beginning at any byte in the buffer, and interpret the following bytes according to the desired type, using methods such as `DataView.readUint32(`*`offset, litteEndian`*`)`. We make extensive use of the ArrayBuffer and its associated views throughout the project, and also note that it is part of the javascript subset known as *asm.js*, a project maintained by Mozilla as a set of ahead-of-time compilable and highly optimizable javascript[24].

We will describe the implementation of each module starting from the lowest level, the FixedInt library, followed by the emulator, and finally the visual debugger. For context regarding the size of each part of the codebase, the `FixedInt` module contains 1430 lines of javascript, including tests; the emulator contains around 3300 lines of javascript, including tests; and the Debugger application contains around 2000 lines of javascript and JSX templates, and 1000 worth of CSS stylesheets. Bundled and minified, the entire application is served with approximately 370KB of javascript, and 20KB of CSS.

4.1 The `FixedInt` Library

At the assembly level, types do not exist as they do in higher level languages. While there is some hardware distinction between floating point and integral types, conceptually different types like signed and unsigned integers, memory addresses, and

[1] There are also the wrapper classes `Float32Array` and `Float64Array`, but we make no use of floating point data and will not discuss them.

character arrays all look the same in memory: lists of bytes. It's then up to the program to decide how to interpret these values, imposing the concept of type on them when we ask certain questions. Important questions for control flow, like "is this number greater than that number" don't make sense in the world of bytes until we decide what the bytes mean. In the level of C, the decision making is abstracted away, and we can imagine that every piece of data we manipulate exists with only one interpretation – unless we explicitly change what that interpretation is with a cast.

The closest thing to type on the machine level is size: we manipulate values of specific, fixed widths in bytes. In most cases, instructions in the x86 architecture require the size of the operands to be specified, as slightly different hardware may be used for manipulating values of different length.

Somewhat ironically, Javascript lacks explicit types in a totally different way. Rather than pedantically checking every operation to make sure you haven't used your data in a way other than you promised you would, javascript lets you reassign variables as you please, allows direct comparison between objects of different types, and doesn't distinguish between floating point and integer values. This last part is especially tricky – javascript uses a single 64-bit IEEE double precision floating point type to hold all numbers[18], even those returned by a call to `parseInt()`!

The ArrayBuffer can support fixed width values, but `ArrayBuffer`s get clunky for passing data around, and there is no easy way to do arithmetic on `ArrayBuffer`s as they exist. So we need some intermediate form into which we will read from `ArrayBuffer`s for doing any arithmetic and data movement. 1,2, and 4-byte values all fit inside the javascript `Number` type with full precision, but for values greater than 2^{53}, the underlying double precision type can no longer represent every integer. While we still technically have 64-bits, we don't have 64-bit *integers*.

While tempting, we can't just throw away precision for values between 2^{53} and $2^{64} - 1$, even though the most common use for 64-bit values in x86-64 is addressing,

and the largest virtual address spaces used in practice these days are only 48-bit[], we want our system to actually work, not only work as long as you don't try to add really big 64-bit values. So, we need a new data type. The first workaround that came to mind was a special 64-bit integer class, simply containing two regular-old javascript numbers. There are no precision problems if we keep lo and hi below 2^{32}, and because of Javascript's lax type system, we can pass Int64 instances to a function the same as any other `Number`. If precision matters for some operation, we just handle the `(x instanceof Int64)` case separately, and otherwise can simply extend the `DataView` prototype to support reading and writing 64-bit integers.

But large integers are not the only problem in hardware emulation; we would like the result of adding two fixed width integers of the same size to be a valid fixed width integer of the same size. So what happens when we add `0xFF + 0x01`? If these are 1-byte values, the sum should be `0x00`, and we would indicate that an overflow has occurred, which is not the case for the addition of two javascript `Number`s. Similarly, our representation should not care whether the values are interpreted as signed or not; one of the beauties of twos-complement arithmetic is that sums and differences are the same regardless of whether we interpret each operand as signed or unsigned.

4.1.1 The `FixedInt` data type

The answer to our problem is to wrap all values in a single fixed-width integer type, with different sizes corresponding to the four common data sizes on 64-bit architectures: 1, 2, 4, and 8-bytes. Internally, this is represented by two IEEE double-precision floating points (javascript `Number`s), which we call `hi` and `lo`, each of which is limited to 32-bit integer values. `FixedInt`s of 1,2, or 4-bytes have `hi` set to zero, and the value stored in `lo`. The 8-byte `FixedInt` uses `hi` to store the high-order 4-bytes of its value, and `lo` for the low-order 4-bytes.

Figure 4.2: API for the FixedInt data type

```
            FixedInt(Number size, Number lo, Number hi)
            FixedInt(Number size, DataView buf, Number offset, bool le)
            FixedInt(Object other)
       bool  isNegative(void)
       bool  isOdd(void)
       bool  isSafeInteger(void)
       bool  isLessThan(FixedInt that)
       bool  equals(FixedInt that)
   FixedInt  clone(void)
   FixedInt  toSize(Number size)
ArrayBuffer  toBuffer(DataView buf, Number offset [, bool le])
     Number  valueOf(void)
     String  toString(Number radix, bool signed)
```

While this allows us to represent values up to 64-bits, it doesn't guarantee that the representation is unique. In a two's-complement integer scheme, the 1-byte signed value -1 has the same representation as the 1-byte unsigned value 255. Javascript `Number`s on the other hand have two distinct values for -1 and 255, and still more representations with different values in the irrelevant upper bytes.

To solve this, we propose an invariant: internally all `hi` and `lo` values will be nonnegative, and for 1 and 2-byte `FixedInt`s, `lo` will be in the range $[0, 2^8 - 1]$, or $[0, 2^{16} - 1]$, respectively. We should still be able to construct a `FixedInt` from a negative number, but it should not stay negative. So how do we convert -1 to 255? Bitwise operations `|`, `&`, $\wedge$, `>>`, `<<`, `>>>` are defined on the `Number` class, but are only valid if the value of that number is a 32-bit integer. More precisely, each operand to a bitwise operation is first converted internally to 32-bit integer, and the result is cast back to a `Number`.[18] Further, the results of bitwise operations are interpreted as signed when casting to `Number`, with the exception of the logical right shift `>>>`, which always returns a positive `Number`. Also notable, all shift operations are taken *mod 32*. To illustrate these quirks, we can enter a couple of basic bitwise

operations into the nodejs REPL:

```
(node) 2 << 33
4
(node) 4294967295 >> 0
-1
(node) -1 >>> 0
4294967295
```

In practice, this just means that we must be aware of these edge cases, and be careful when casting: first limit the input to the desired bit length, and then right shift by zero to force the result to be positive. By keeping a set of bitmasks indexed by byte length we can easily convert automatically in the constructor, and making the class immutable gives us a guarantee that our invariant will not be violated. This looks something like

```
const MASK = {1: 0xFF, 2: 0xFFFF, 4: 0xFFFFFFFF};

class FixedInt {
  constructor(size, lo, hi) {
    // ...
    if (size === 8) {
      this.hi = (hi || lo / (MASK[4] + 1)) >>> 0;
      this.lo = (lo & MASK[4]) >>> 0;
    } else {
      this.lo = (lo & MASK[size]) >>> 0;
    }
  }
}
```

This one fiddling of bits takes care of a lot of our issues for us. Then, when we actually do arithmetic on two `FixedInts`, we just do arithmetic on the constituent `Numbers`: `hi` and `lo`, and when we construct a new `FixedInt` to hold the result, our constraints will be enforced. For 8, 16, and 32-bit `FixedInts`, we can just do `Number` arithmetic and cast the result. In the above example, `0xFF + 0x01 = 0x100` and `(0x100 & 0xFF) >>> 0 = 0` as required.

Figure 4.3: API for the FixedInt ALU

```
static FixedInt  add(FixedInt a, FixedInt b)
static FixedInt  sub(FixedInt a, FixedInt b)
static FixedInt  mul(FixedInt a, FixedInt b)
static FixedInt  div(FixedInt a, FixedInt b)
static FixedInt  sar(FixedInt a, FixedInt b)
static FixedInt  shr(FixedInt a, FixedInt b)
static FixedInt  shl(FixedInt a, FixedInt b)
static FixedInt  and(FixedInt a, FixedInt b)
static FixedInt  xor(FixedInt a, FixedInt b)
static FixedInt  or(FixedInt a, FixedInt b)
static FixedInt  not(FixedInt a)
static FixedInt  neg(FixedInt a)
```

4.1.2 The `FixedInt` ALU

The other part of the `FixedInt` package is an implementation of an Arithmetic Logic Unit (ALU), which handles real operations on `FixedInt`s. The `ALU` class is distinct from `FixedInt` and exposes only static methods that all return new `FixedInt`s. Two other designs were also considered: the first made arithmetic and logical operations instance methods of `FixedInt`, and operations took the form `a.add(b)`. This had the performance advantage of not having to construct a new object for each operation, but would mutate the value. While faster, this introduced the possibility of unexpected mutations after passing a `FixedInt` instance to a funciton, as well as confused the concept of status flags (i.e., does it make sense to refer to the Carry Flag of a particular `FixedInt`?). Another potential solution involved adding static ALU methods directly onto the `FixedInt` prototype, and having each return a new `FixedInt`. While an improvement, it would also make it more difficult to later change or augment the behavior of the ALU, and in the spirit of supporting arbitrary architectures, one could imagine a situation in which the `FixedInt` type was sufficient, but the `ALU` would need to be updated. To allow for this possibility, the `ALU` was encapsulated as its own class.

Given the immutability of the `FixedInt` type, and the guarantee that javascript `Numbers` can be coerced to a unique two's complement representation of the appropriate size, the implementations of individual arithmetic and logical operations were mostly straightforward for 1,2, and 4-byte operations. The 8-byte case would always require some special handling, but often logic could be shared for all sizes. The typical ALU function is structured as follows:

```
function (a, b) {
        const {a, b, size} = _validateInputs(a,b);
        // common logic
        // ...
        if (size === 8) {
                // special 8-byte logic
        } else {
                // special 1,2,4-byte logic
        }
        // common logic
        // ...
        const result = new FixedInt(size, x, y);
        // set flags based on a, b, and result
        // ...
        return result;
}
```

To handle ALU flags like a real processor, we can just write to a couple of static booleans from inside the arithmetic and logical functions. The carry flag is set if the value being passed to the `FixedInt` constructor is larger than the max for that size. The zero flag is trivial, and the sign flag is just a test of the highest valid bit for the size. We can encapsulate the high bit test into an `isNegative()` method, and the overflow flag can be computed by comparing the signs of the operands and the result. With addition for example, the overflow flag should be set if the operands are the same sign, and the result is different.

Additionally, to handle operations which may return a larger value than the input, we also add a read-only property to the `ALU` class which we call `aux`, for storing extra/auxiliary values relevant to the most recent computation. In the case of multiplication, this is the upper n bits of a $2n$-bit product, the result of two n-bit numbers

being multiplied. For division, this is the modulus, which is incidentally computed by the recursive division algorithm used, and required for emulation of x86.

4.2 The Emulator

Like the rest of the project, the Emulator is written in Javascript, but makes no use of Web APIs such as the browser event loop or the window object and its descendents. This allows the emulator to run outside of a web context (e.g. in a Node.js or embedded V8 context[2]), and facilitates more rigorous and automated testing from the command line, as well good logical separation of concerns. However, for the sake of the application being described, being written in javascript allows the emulation to occur *in the browser* and without any communication with a server. This alleviates the vast majority of the security concerns associated with hosting a debugger on the web. Rather than allow users to upload untrusted code and relying on complicated sandboxing schemes to insulate the server, we deliver the full runtime environment to the user and run their code on the emulator running on their own browser.

In terms of module design, dependencies try to model relationships between the real pieces of hardware and software that are being emulated. For example, the CPU chip or instruction set has access to registers and memory, but memory by itself has no concept of registers, and does not know where the stack pointer is at a given point in time. This also allows for the emulator to be readily extended, as any module can be substituted for a different implementation without affecting the behavior of the others. We will primarily describe the emulator in the context of x86-64, which is the only reasonably fully implemented architecture. However it should be clear from this discussion how support could be extended for other architectures, memory models, or libraries, and we will address this further in Chapter 6.

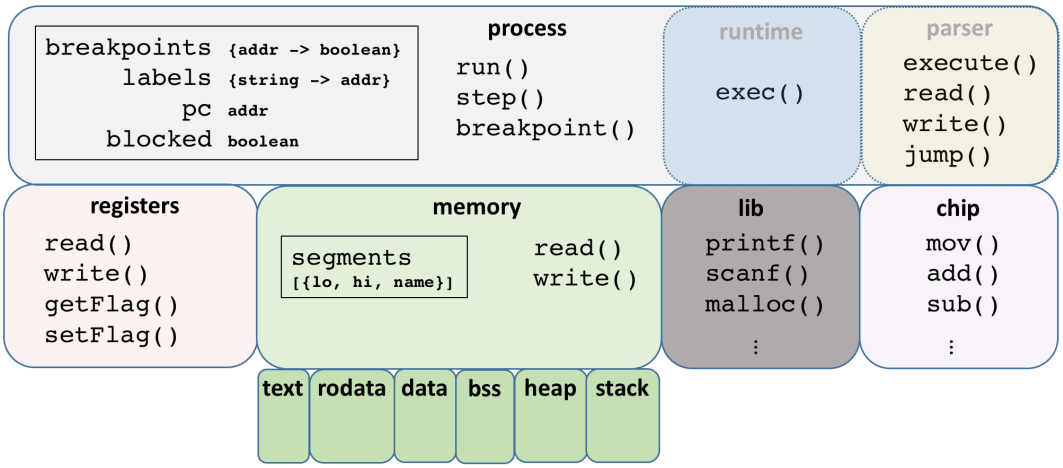
[2]Node.js is a standalone implementation of Google's V8 javascript engine. Additionally, V8 may be embedded into other C++ applications that may require running javascript, and this emulator can run in such an environment[33]

4.2.1 The Assembler

Though this project does not actually assemble any assembly language to machine code, there are several tasks performed at assemble time that are still necessary for accurate emulation. In particular, the assembler (e.g. `gas`[19]) must generate a symbol table which maps from *labels* defined in the assembly file to *addresses* in the final executable. It also must reserve space in the object file for any data that is statically allocated in the `.rodata, .data,` or `.bss` sections of memory. In real compilation pipelines, the addresses of objects in the final executable may not be known at assemble time, as more assembly files may be combined with the one currently being assembled before an executable is generated. Instead of real addresses then, the Assembler will generate *relocation records*, which indicate an address that must be computed at link time, and contain an offset relative to some address that can be computed easily. At link time, the linker gathers all of the individual object files, and resolves all of the relocation records to generate addresses in the final executable.

Our assembler takes a similar approach, first mapping labels to a set of pseudo line-numbers on a first pass, and then making a second pass in which each instruction is written to memory, and space is allocated in the returned `Image` object for each variable declared at assemble-time. On this second pass, line numbers are converted to addresses as the `Image` is generated, and a dictionary of labels mapping to addresses is returned as well. Fortunately, emulating behavior on the assembly level saves one of more cumbersome tasks of the linker, and labels and addresses don't need to be relocated within instructions. Because the runtime environment of the emulator will retain access to the dictionary of labeled addresses, jumps, calls, and other uses of labeled memory can be resolved dynamically at runtime.

Figure 4.4: A diagram of the `Process` object and its subordinates. The `parser` and `runtime` modules each inject methods directly onto the process object, while the `memory`, `registers`, `lib`, and `chip` modules are tacked on as extensions.



4.2.2 The `Process` Object

The `Process` object is the top-level data structure for the emulator. It is instantiated with both the image of the program to be executed, as returned by the assembler, and any additional context which determines the execution environment for the process, including the processor chip, assembly parser, register definition, and standard library. To some extent then, the `Process` object contains in its abstraction it's own version of everything that is necessary for a process to run, including CPU, memory space, registers, libraries, and system resources. While traditionally processes are provided as an abstraction to users so that programs may be written *as if* they have complete access to the address space and registers, and complete control of the CPU, in this emulator each process *does* have exclusive access to its own memory, registers, and CPU chip. There is no implementation of address translation or CPU time-slicing, as software versions of these resources can be instantiated multiple times, unlike their hardware equivalents which require such tricks.

Each of the abstractions discussed in the following sections are subordinate to the `Process`, and are *bound* to it in the javascript sense of sharing the same `this`. Most

of the work of emulating a process is done in one of its subordinate objects, and the `Process` object itself serves mostly to administer the task with its execution loop. The main execution loop consists of three steps:

1. The process consults the breakpoint dictionary to determine if execution should proceed.

2. The process fetches the next instruction from memory. If the next instruction is non-null, it passes the result to the parser to decode and execute.

3. The parser passes control to the appropriate mnemonic in the chipset, which makes changes to the program state, using further helper functions provided by the parser.

Given this main loop, the rest of the Process' behavior is determined by the individual modules described below, which can each be altered indepedently to support any process that can be described by this "check breakpoint, load, execute" loop.

4.2.3 Parsing

While real machines operate on binary machine code, with each instruction and operand given a unique encoding, assembly language often condenses multiple instructions or operand types into a single form of mnemonic. The (albeit limited) expressiveness of assembly language can get in the way of a direct translation between instructions and the process' resources such as memory and registers. While a first approach could be that instruction parsing should be provided with the architecture, this creates a tight coupling in a place where one doesn't exist in real systems. That is, in an x86-64 architecture, we should be able to switch out AT&T syntax for Intel without rewriting every function on the entire chip. Since the actual machine behavior is the same, this can be accomplished by a parser module that can translate

the idiosyncacies of a particular syntax into the proper reads, writes, and jumps in the process.

Accordingly, the parser provides four public functions; `execute()`, which is called by the `Process` object's execution loop, and `read()`, `write()`, and `jump()`, which abstract away the operand syntaxes of the assembly language flavor being parsed, and are available within the chipset to bridge between emulated CPU behavior and the string representation of instructions in assembly. In essence, `execute()` takes two arguments: a string *mnemonic*, and an array of strings *operands*. It parses the mnemonic, and determines the correct chipset function to call, passing along the operands and any pertinent other information obtained from the mnemonic (e.g., operand sizes inferred from the suffix in AT&T syntax). Within the appropriate chipset function, `read()`, `write()`, and `jump()` can be called on the individual operands, according to the behavior of the particular instruction. This greatly simplifies the implementation of the chipset functions, and each may leave it to the parser to distinguish between immediate, register, or memory operands when reading or writing. Similarly, `jump()` is charged with distinguishing between labels, indirect jumps to an address in a register, or external calls to the standard library. Any new parser can be substituted for the existing AT&T/x86-64 parser by supplying these four methods.

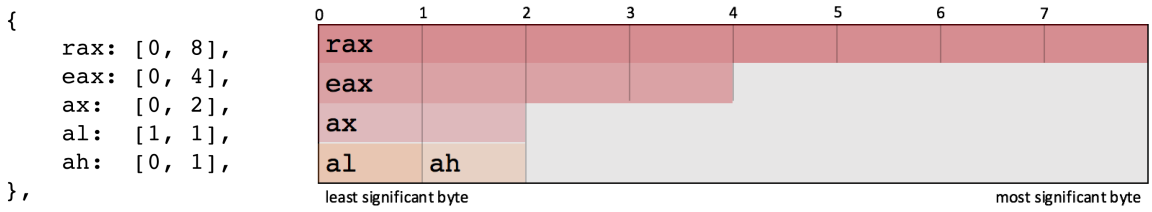
4.2.4 Registers

Registers, the type of storage closest to the CPU in the memory hierarchy, come in many shapes and sizes, and in many architectures multiple named registers may overlap. In particular, hardware registers do not enforce *Load-store consistency*;[3] data written to one register may be read from a different, overlapping register. In

[3]Here "consistency" should not be confused with *data consistency* in the distributed computing context. Here it is referring to the pattern of access, i.e. is one *consistently* accessing data in the same way, or does the usage depend on the side effects of writing to a particular destination.

x64, all general purpose registers come in at least four sizes, and all but those added in the architecture's transition to 64-bits (`%r8-r15`) have two different 1-byte versions that can be accessed. As this particular fact can be tricky for students to internalize, and is typically not a feature of higher level languages, we wanted to ensure that the emulator handled cases of load-store *inconsistency* correctly. An example of load-store inconsistency would be writing a value to each of `%ah` and `%al`, and then reading from `%ax`. On a real machine, the two-byte value `%ax` would have its lower-order byte equal to the value written to `%al`, and it's higher byte equal to the value written to `%ah`.

Figure 4.5: Illustration of overlapping registers and encoding of the register description object.



To provide this guarantee, we need a byte-addressable structure from which we can read and write values of 1, 2, 4, or 8 bytes at a time. As discussed above, the `ArrayBuffer` is perfect for this case. However, to provide a clean bridge between the parser and registers, we would like to be able to access each register by it's string name, as one would in the text of an assembly program. To achieve both, we add a single layer of indirection provided by a register description object, which is mapping from string names to tuples of index and length, indicating where in the underlying `ArrayBuffer` a particular register's value can be found. On little-endian architectures such as x86, most overlapping registers begin at the same index, and just vary in length, such as the `%rax` family shown in Figure 4.5. Each register group is laid out sequentially in a single ArrayBuffer, but the Register object only exposes methods `read()` and `write()`, to access values defined by the register description.

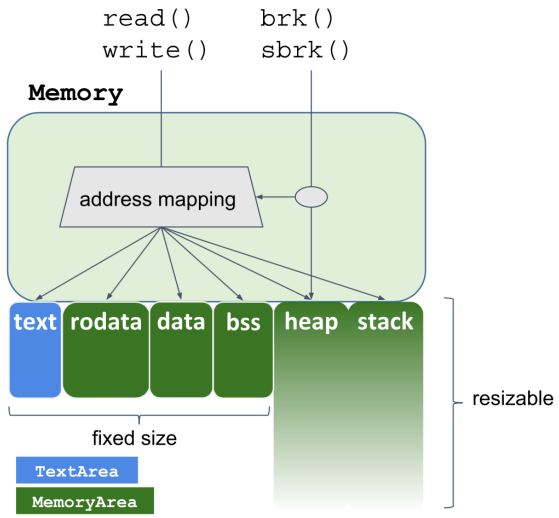
While we use the ArrayBuffer to provide *byte*-level accuracy for registers, we are less concerned with *bit*-level accuracy, as it pertains to status registers such as `EFLAGS` in x86. Though the different status flags are usually implemented as different bits of a single register, they are conceptually independent quantities, and instructions even reference them as such.[4] Drawing on this conceptual interpretation, we implement flags as an additional mapping from flag names to booleans, where the list of flag names is also defined in the register description object.

4.2.5 Memory

Unlike registers, the abstraction of a process' memory space is not dependent on CPU architecture. While precise methods of accessing memory may vary chip to chip, and address translation methods may vary from OS to OS, the concept of portable code requires that the simplest form of the address space abstraction must be platform-independent. Specifically, we want to be able to write to and read from any byte in our space, from 0 to $2^{64} - 1$ on a 64-bit machine, knowing that some operations may trigger a fault if we do not have permission to read or write from that address. The permission concept gets implemented in the form of virutal memory areas (VMAs), where each area spans some subset of the address space, and has a particular set of permissions among read, write, and execute. On real systems, permissions are managed on the page level by the Memory Management Unit (MMU), which will trigger a page fault or general protection fault if a page is accessed in a way that violates its specified permissions (and do nothing if not). The page fault handler will then consult the VMA structure to obtain more information and determine whether it can recover, or whether it should crash the program.

[4] Beyond the many jump instructions which test some logical combination of the status flags, x86 also includes instructions such as `stc`, `clc` and `cmc`, which set, clear, or complement the carry flag respectively[26]

Figure 4.6: Cartoon depicting the organization of the `Memory` object and its subordinates



We implement this in a similar but simpler way. Without an MMU or memory paging, we use VMAs as the fundamental unit of memory, and use a wrapper Memory object to check permissions and delegate reads and writes to the different memory areas, as shown in Figure 4.6. We implement two distinct types of VMA: one for data (`MemoryArea`), and one for code (`TextArea`). The `MemoryArea` class wraps an `ArrayBuffer`, and (just like our register implementation) exposes `read()` and `write()` methods, which each take addresses. Internally, the `MemoryArea` maintains its high and low addresses, and translates each access to an index into the `ArrayBuffer`. Some areas, such as the Stack and Heap, may grow dynamically during a process' execution, and so the `MemoryArea` class also has a `resize()` method which can be called when a VMA is accessed outside of its current bounds. The resize behavior is specified on the instantiation of the `MemoryArea`, as a function for determining whether a particular address is valid for the area to resize[5]. Like registers, the `read/write` API provides a convenient way for moving `FixedInt`s around

[5]For the stack, this is specified as the size of the "red zone"[10] beneath the current stack pointer, which a process can safely reference. The heap is only resized manually by the containing `Memory` object after calls to `brk()` or `sbrk()`

the emulator; the `write()` function takes a `FixedInt` value to be written to memory, and `read()` returns a `FixedInt`.

The `TextArea` class handles the text section, and exposes only a `read()` method. However, as our emulator operates on code in its assembly form (i.e., strings), the return value of `TextArea.read()` is a regular javascript `Array` of strings representing mnemonics and operands. The effect of this is that the `Memory` object's `read()` may return an `Array` or `FixedInt`, depending on whether the address argument refers to the text section or elsewhere in memory, respectively. As mentioned earlier, this also implies that code cannot be injected into memory and executed, nor can the text section be introspected by a process, as the code and data in the emulator have fundamentally incompatible representations.

4.2.6 Instruction Set(s)

The instruction set is one of the more straightforward modules in the emulator, and defines the javascript implementations of each assembly mnemonic. More precisely, the chip is simply a javascript object (conceptuatlly a hashtable) which maps from mnemonic families to functions. Mnemonic families in this case describes a group of mnemonics that perform the same funciton for different sizes of data (e.g., `movq, movl, movw,` and `movb` are all in the `mov` family). For the implementation of the x86-64 chipset, each mnemonic accepts an array of string operands, and an integer size in bytes (1, 2, 4, or 8). Very little computation is actually implemented in the mnemonic functions, and they serve primarily as a way of delegating arithmetic to the `FixedInt ALU`, and mnemonic parsing to the parser's `read()`, `write()`, and `jump()` functions, and updating the status flags when appropriate.

An ARMv8 chipset was also begun as a proof of concept for multiple-architecture support, and for the most part is just a new translation between operand positions,

ALU functions, and some specific register names, while reading, writing and jumping are tasks of the complementary ARMv8 parser.

4.2.7 C Library Calls

While our primary aim for the tool was to allow students to debug assembly language programs, it is unlikely that anyone learning assembly for the first time would not be learning it in the context of a larger runtime environment. To that end we would like to provide some of the comforts of the C runtime environment within our emulator, to allow for a greater range of assembly programs to be debugged on the platform.

There are many ways in which libraries can be linked against a compiled binary, and all of them are difficult. Usually libraries are included at link time (for static libraries) or load time (for shared object libraries), when the final executable binary is produced. In the case of dynamically linked libaries, procedure calls involve a layer of indirection, in which the first call to a particular function instead calls a dynamic loader in the Procedure Linkage Table, which writes the actual address of the function into the Global Descriptor Table[11]. Implementing this in our emulator would be prohibitively complicated, and extraneous information for most students. Not to mention it would require that we distribute a disassembly of a C standard library with the app, and would require us to support some obscure instructions that could turn up there. Static linking is only slightly better, as we would then require students to compile their own binaries, and disassemble them in the application. Both approaches suffer from the fact that many library functions make a system call at some point, which would require us to jump out of the emulator and provide some functionality in javascript. Instead of adding this complexity, we instead decided to implement a subset of the standard library in javascript, using a foreign function

interface (FFI) that conforms to the System V ABI (Application Binary Interface) for x86-64[6] (or whichever architecture is currently being emulated).

The FFI itself consisted of only two functions, provided by the architecture module: `abi.arg()` mapping an integer index to the `FixedInt` value of the argument read from the appropriate register, and `abi.ret()` which writes its argument to `%rax`, pops the address from the top of the stack, and jumps to it. The library functions each begin with calls to `abi.arg()` to retrieve the proper arguments, and the body of the function can perform any necessary computations in javascript with full access to manipulate the process object, before calling `abi.ret()`. Each of the library functions are invoked without any (javascript) argument, and it is the job of the parser when making a jump to identify when a particular label is a part of the attached library, and call the appropriate function. In another nod to the extensibility of this design, the same library implementation can be used with multiple architectures, so long as each provides its own implmentation of the function call ABI for the target architecture.

4.3 Front-end and Debugger

The user interface for the visual debugger application is quite straightforward given the emulator on top of which it is built. This is accomplished by using the javascript framework React[15]: a declarative, component-based library for making responsive user interfaces. In particular, the declaritive syntax allows us to define the application's entire view state in terms of arbitrary data. For *Processable*, the data that defines our view is the process object defined in the emulator layer. Much of the relevant data is deeply nested, and so we try to similarly nest the components which

[6]Specifically, this is the convention that the first six argument are passed in registers `%rdi,%rsi,%rdx,%rcx,%r8d,` and `%r9d`, respectively, and the return value is passed in `%rax`

display it, threading only the relevant members of the process object through to their corresponding presentational elements.

Fundamentally we would like to display two different types of data: string assembly instructions, and type-agnostic, variable-length byte fields. The former is accomplished simply by mapping each instruction to an HTML element containing the instruction's text and address. The address is formatted as a button, with a click handler bound to the breakpoint toggle method of the currently executing Process object. In addition, the text section displays the current value of the Program Counter (PC), and conditionally formats the instructions to clearly indicate the next one to be executed.

For non-text data, we build a display hierarchy up from the `FixedInt` data type. At the bottom of the hierarchy for each virtual memory area we have an `<Item/>` component, which holds the items value as its main property. For the purporses of display, the `FixedInt` module is augmented with a set of `decode` functions, which define an enumeration type for the five decoding options we provide.[7] The `<Item/>` maintains via the `state` instance variable its current decoding, and uses that to call the appropriate decoding function in its `render()` method. The `<Item/>` registers a click handler which cycles the decoding through the available options, changing its color and decoding label along the way. For all memory areas (i.e., excluding only the register section), the `<Item/>` also renders each byte of its underlying `FixedInt` to provide the raw data which does not change with each decoding.

On top of this, each VMA in the display must map from its corresponding `MemoryArea` in the emulator, to a list of `<Item/>`s to be rendered. For the static sections and Heap, we rely on a simple mapping which divides the entirety of the VMA in question into equal-sized chunks: the largest of 1, 2, 4, or 8 which evenly divides the area in question. For static sections, this process is repeated between

[7] Again, these are unsigned hexadecimal (`hex`), unsigned decimal integer (`uint`), signed decimal integer (`int`), binary (`bin`), and ASCII character (`char`).

each label, and for the heap, this simply defaults to 8-bytes accross the entire section. The Stack is much more flexible, as it tends to have the largest variety of data stored within each function call frame, and so we allow the user to customize the way in which it is decomposed into individual `<Item/>`s. This is done by providing both a global chunk size, which a user may toggle between 1, 2, 4, or 8-bytes, as well as a separate custom mode in which the stack can be divided into chunks of unequal size. This is accomplished by maintaining an array of addresses in the top level stack component which we call *breaks*. When rendered, the stack iterates over each break, and constructs a `FixedInt` from the stack by reading the data between consecutive breaks. We then allow the user to add or remove breaks from this list with a click handler on the address adjacent to each stack byte. As `FixedInt`s are limited to specific byte sizes, there is additional logic in the break toggle handler to add additional breaks if necessary to keep each chunk a valid size, or reject the toggle altogether if it is impossible at the given address.

Chapter 5

Evaluation and Conclusions

5.1 User Evaluation

To gauge the success of the application as a potential teaching tool, we conducted a user evaluation in two phases. The first phase was an in-depth walkthrough with three expert users, and consisted of a cognitive walkthrough using an early, stable version of the software to trace an assembly-language program with which each of them were quite familiar. Following this evaluation and a round of revisions based on their feedback, the alpha version of the software was published and the link was distributed via piazza to the students enrolled in COS217 at Princeton in Spring 2018. The alpha version included a link to a feedback form which the students were encouraged to fill out. We now present the results of each of these rounds of evalutation.

5.1.1 Evaluations by Instructors

To perform an expert evaluation, we needed participants who were not just well-versed in assembly language and debugging, but in particular experts in *the process of learning assembly language*. While this is a strange designation, it is exactly the expertise of instructors who have been teaching introductory systems programming

for several years. The three instructors selected to evaluate the first version have each been preceptors for COS217 for multiple offerings of the course, and had extensive experience introducing assembly language to computer science students, and tracing assembly language programs by hand.

To evaluate the application, each instructor was first introduced to the software by loading a simple Hello World program, (`hello.s`, provided in Appendix B). Following a brief tour of the user interface, the evaluators were instructed to return to the landing page, and load a second program, `euclid.s`[1], wich computes the greatest common demoninator of two integers read from stdin. Each evaluator was then asked a series of simple questions about values in registers or memory during the execution of the program, (the full text of the evaluation script can be found in Appendix C).

All three evaluators exhibited a similar learning curve for the interface, which was fairly steep in the sense that initial confusion gave way to comfort with the interface after a few minutes of use. The feedback from the preliminary evalution was uniformly positive, and all three were able to answer all questions without much trouble. At the same time, most sticking points were shared between evaluators, and the evaluation process resulted in the following takeaways:

Problem *The direction of stack growth is a conceptual block*

When asked to find a value on the stack, all three instructors intially stumbled while navigating it, remarking out loud that they expected it to be growing upwards rather than downwards. Discussing the issue with each of them, they admitted a bias that stems from the way they draw the stack in class, which may not be shared by those learning assembly for the first time. Further, the popular Computer Systems textbook by Bryant and O'Halloran[11] (used in COS217) depicts the stack growing in the opposite direction, to indicate that it grows towards lower addresses on x86. The

[1]This program is one of the examples used in the precepts for COS217, and all three instructors were familiar with the program. It's full text is also available in Appendix B

confusing disparity between the conceptual stack which "grows up" and the literal stack on x86 which expands towards lower addresses was also noted as a common stumbling block for students learning assembly, and so we concluded that regardless of direction, there should be a visual cue indicating which direction is being shown.

Solution: The view code used for showing the stack was refactored in such a way that the order of items could be reversed with a single toggle. To make the direction still more clear, an invisible padding element was added after the stack top to ensure that top was distinguishable from the origin. The current orientation of the stack was displayed in a button to the right of the stack area's title, containing either an up or down arrow as appropriate (↑/↓). Clicking on the button would toggle the stack direction as well as the direction of the arrow. At the request of the instructors, growing up was set as the default behavior.

Problem: *The presence of some funcitonality was unclear*

Likely as a result of previous experience with gdb, the evaluators would often rely on the console to perform tasks for which there was a convenient button provided instead.

Solution: Clickable objects were given mouseover tooltips to explain their function, which appear after hovering over an item for 500ms or longer. Further, an "about" page was added with a description of the various functionality provided, so that users may consult more information when stuck.

Problem: *The values of flags were not clear*

When asked about the values of flags in the status register, displayed as colored boxes with a two-letter abbreviation for each flag, evaluators noted that users unfamiliar with assembly language may need more information to disambiguate the flags and their states.

Solution: Flag boxes were given mouseover tooltips giving the flag's long name (e.g. CF = "Carry flag"), and whether or not it was set.

5.1.2 Evaluations by Students

Following the incorporation of feedback from the instructor evaluation, the application was published as an alpha release, and the link was distributed to students currently enrolled in COS217 during the week in which the course began teaching assembly language. The release coincided with the students' first assignment in assembly language, requiring them to write a word count program, and an optimized addition function for a big integer data type. Unfortunately, a combination of factors lead to a low usage rate for the alpha version, with a total of around 40 unique users visiting the site between March and April. In particular, the assembly assignments for COS217 included one very straightforward translation, (which we presume did not require significant debugging effort,) and one more difficult assignment that used the heap and required multiple files. As the alpha version supported neither visualization of the heap nor multi-file uploads, it was less useful to students during this week than we had hoped.

However, after soliciting additional reviews from former students of the class we were able to achieve a second round of feedback which guides some of the future work discussed below. The student reviewers were very enthusiastic about the software, but having been presented with it without any introduction (as was given to the expert reviewers), they uninanimously requested a tutorial animation of some sort. Beyond that, no one offered significant proposals for additional features or reported any other inconveniences or bugs.

5.2 Performance Evaluation

Figure 5.1: Instructions supported by *Processable* and their popularity.

Statistics for mnemonic popularity was determined by parsing the output of objdump for binaries found in `/usr/bin/` and `/bin/` on MacOS Sierra 10.12.6. The same tests were also run on Amazon Linux with comparable results.

instruction	**popularity**		
`mov`	38.30%	`or`	0.46%
`j`*cc*[1]	8.99%	`shl`[3]	0.34%
`call`	7.78%	`dec`	0.30%
`lea`	6.78%	`movabsq`	0.18
`cmp`	5.39%	`imul`	0.15%
`test`	3.91%	`sar`	0.10 %
`push`	3.8%	`idiv`	0.04%
`pop`	3.73%	`adc`	0.04%
`xor`	3.60%	`not`	0.03%
`jmp`	3.53%	`neg`	0.03%
`add`	2.85%	`stc`	0.01%
`movs`*xx*[2]	1.41%	`cmc`	0.01%
`sub`	1.08%	`cltq`	$< 0.01\%$
`ret`	1.04%	`cqto`	$< 0.01\%$
`and`	0.79%	`jcxz`	$< 0.01\%$
`movz`*xx*[2]	0.75%	`jecxz`	$< 0.01\%$
`inc`	0.67%	`shr`	$< 0.01\%$
`nop`	0.58%	`hlt`	$< 0.01\%$
		unsupported[4]	2.71%

[1] Each conditional jump is implemented separately, the full set consists of `jns`, `jne`, `jo`, `jno`, `ja`, `jae`, `jb`, `jbe`, `jg`, `jge`, `jl`, `jle`, `js`, `jz`, `jnz`, `jc`, `jnc`, `jnb`, `jnbe`, `jna`, `jnae`, `jng`, `jnge`, `jnl`, and `jnle`

[2] These are the intel syntax mnemonics for casting moves. The mnemonics implemented by *Processable* are in AT&T syntax, and are `mov`*x*`bw`, `mov`*x*`bw`, `mov`*x*`bq`, `mov`*x*`wl`, `mov`*x*`wq`, `mov`*x*`lq`, with *x* replaced by `z` or `s`, for zero-extension and sign-extension, respectively

[3] This also includes `sal`

[4] Among the most popular unsupported instructions are `cmov`*cc* (0.47%), `movaps` (0.25%), `rol` (0.16%), `xchg` (0.04%)

5.2.1 Instruction Support

A good way to evaluate an emulator is to see just how well it actually emulates the desired architecture. More precisely, we can ask the question, how many of the architecture's instructions are implemented? While it's hard to precisely determine how many instructions there are in the x86-64 architecture, one estimate says there are 981 distinct mnemonics[25], ignoring operand sizes and types.[2] The x86 module in this projects only implements 69 different mnemonics (ignoring size), which at first glance seems like pitiful coverage. However, we must take into consideration that the vast majority of these mnemonics are fairly special-purpose, and most programs don't use them at all. So how can we better estimate the coverage? Expanding on an approach used by Peter Kankowski to answer a similar question in 2006,[28] we can analyze the results of disassembling different collections of programs with `objdump`, and count the occurence of each instruction to determine its popularity, and arrive at a measure of coverage weighted by popularity. Using this metric, and using the Unix utilties found in `/bin` and `/usr/bin` as a sample, we conclude that *Processable* can run **97.3%** of the disassembled code found there. The breakdown by mnemonic, and the full list of supported instructions can be found in Figure 5.1.

5.3 Future Work

5.3.1 Minor Improvements

There are a number of small issues and areas for improvement that have become clear during the project's evaluation that form the immediate next steps before a public beta release. From the common advice of the several students who completed the feedback form for the hosted alpha version, it seemed that the interface was

[2]This is the number of distinct Intel syntax mnemonics. The author also claims that there are 1,279 mnmonics in AT&T syntax, which distinguishes many instructions by their operand sizes.[25]

intuitive only after a short but steep learning curve. An immediate improvement to the workflow then would be the inclusion of a short tutorial or walkthrough of the available features, especially including how to start a process.

Another helpful feature that couldn't be implemented because of time constraints would be allowing multi-file uploads. Though the assembler module of the emulator is written to accomodate multiple files, it does not correctly implement some of the subtler behavior usually handled by the linker, such as the resolution of weak and strong symbols[11]. As such, the file upload button on *Processable* accepts only a single assembly file, limiting its ability to debug larger, multi-file programs.

5.3.2 Improving Architecture Support

Though the focus of this project has been the x86-64 architecture, a remarkable feature of the codebase is its factorization which allows for the substitution of new architectures. In particular, every software component of the emulator described in Figure **??** can be substituted independently. Given that COS217, the main case study for this project, may transition to ARMv8 from x86-64, and that ARM processors dominate the smartphone and embedded computer markets, it makes sense to first aim for ARM support. The necessary reorganization to support ARM has already been performed, and the register description and function call ABI have been written, as well as a skeletal implementation of an ARM instruction set.

Another way to improve architecture support–at the cost of code size–would be to integrate an existing emulator like Unicorn.js with the Debugger part of *Processable*. This would allow execution of arbitrary x86 instructions and accurate addressing in the text section, but also would require the inclusion of a disassembler to display assembly mnemonics. A candidate for this, also ported to javascript by Alexandro Bach, would be Capstone.js[7], and would add another several MB to the application bundle. Some of the *Processable* emulator code would likely persist, for example

the C standard libarary interface and I/O handling are intentionally simplified from their native behavior to coexist with a browser, and to customize the operating system level services that are provided to to each process. These modules could be ported to a new interpreter just by writing a new foreign function interface like those described in Chapter 4.2.7. Together with the original *Processable* interpreter, the application could offer a fast-loading, "simplified" environment, suitable for most students' programs with the previously described limitations, as well as a slower-loading "professional" environment, which would support multiple architectures and their full range of instructions. Educational use cases could expand to include more advanced systems and assembly-level concepts such as vectorized computations, multithreading, and signal handling. Going even further, a full emulator could also benefit from integration with the emscripten virtual file system, which offers a POSIX-like API[44].

5.3.3 Compilation from C

With unlimited time, this project could expand in a number of areas to continue to support students learning assembly language. In particular, offering the option to compile C code directly in the application would greatly expand the number of use cases we could serve, as well as make it easiser for students to use the tool with larger programs. While this could be done by a webserver or even serverless via function-as-a-service providers like AWS, it would be more in line with the spirit of this project to run the compiler *in javascript*. Using emscripten, one could compile a C compiler, and allow uploaded C files to be compiled to assembly or object code *in the browser*, after which they could be *executed in the browser* on top of the *Processable* emulator and *debugged in the browser* from the *Processable* debugger.

To take the compilation one step further, previous work by Matt Godbolt (*Compiler Explorer* [20]) and myself (*Assemblance*[42]) has focused on annotating the compilation process between higher level languages and assembly. Assemblance in par-

ticular contains a python module capable of parsing DWARF debugging information generated by `gcc` or `clang`, and returning JSON describing the mapping between local variables (in the high level source) and their stack offsets or register locations (in assembly), as well as the mapping from lines of source code to lines of assembly. Combined with these tools, *Processable* could graduate to a full-fledged debugger, allowing symbolic debugging of a C program with accurate emulation on the assembly level. Further, tools such as **transcrypt**[2] allow for transpilation between python and javascript, so we could still keep the entire debugger in the browser without connecting to a remote webserver.

5.4 Project Evaluation

Looking back to the project goals defined in Section 1.1, would like to offer some commentary regarding how well *Processable* has achieved these goals, both as a teaching tool for assembly language, and as an effective program visualizer. Regarding the assembly language concepts, the distinction between virtual memory areas is made explicitly clear by the different panels of the interface. Further, the difference between the visualizations provided for static sections and the dynamic heap and stack areas emphasizes their distinct qualities, the former being allocated before the process begins, and the latter two changing in size throughout the program's execution. The type-agnostic property of data in all memory areas is emphasized by the toggleable decodings options enjoyed by each section. Further, the annotation of the stack with the stack pointer (`%rsp`) and base pointer (`%rbp`) registers help to emphasize the bounds of each stack frame, and help to elucidate function call mechanics at the assembly level. Flat control structures remain a difficult concept to convey clearly, but with clear visual elements for each of the status flags, we note that loops and

branches can be explored by setting breakpoints on conditional jump instructions and inspecting the flag area before continuing.

As for the program visualization goals, the customizable decoding feature connects to the desirable properties of multiple viewing options, and response to user interaction, as while the underlying data does not change while the process is paused, the user may completely alter its interpretation and presentation. The discussion of Section 5.2.1 indicates that even the limited instruction set implemented by *Processable* allows for very flexible input, with the potential for extension to multiple architectures only increasing this flexibility. Finally, by combining all of this functionality into a single, manageably-sized bundle of javascript and css, we believe that we have achieved the widest audience possible given currently available technology, as a purely client-side webapp can be run from any device with a (reasonably modern) web browser and internet connection.

5.5 Conclusion

Though the world of tools dedicated to Computer Science Education is rich and densely populated, there remain many niche content areas that are not well covered, and the barriers to creating an effective visual application are quite high. As the experience of building this application has taught me, to effectively teach even a simplified model of a complex system, one must first thoroughly understand the real system in order to identify which concepts may be simplified. Though I expected this to be the case, it was still surprising to me just how many times the solution to a tough problem in system emulation lay somewhere else within the original system! Suffice it to say that while *using* single program visualization tool may help the user understand the system in question, *building* a program visualization tool has (for me at least) resulted in the study of far more computer systems than I ever imagined

would be necessary. As we have set up this project with the goal of teaching systems level concepts for many years into the future, we hope at least a few students who use it will take the opportunity to dive deeper into the world of computer systems, and perhaps extend the software to continue to handle more use cases and broaden understanding of one of the more nuanced and frustrating areas of computer science.

Appendix A

Supplemental Figures

Figure A.1: Screenshots of decoding options available for static sections.

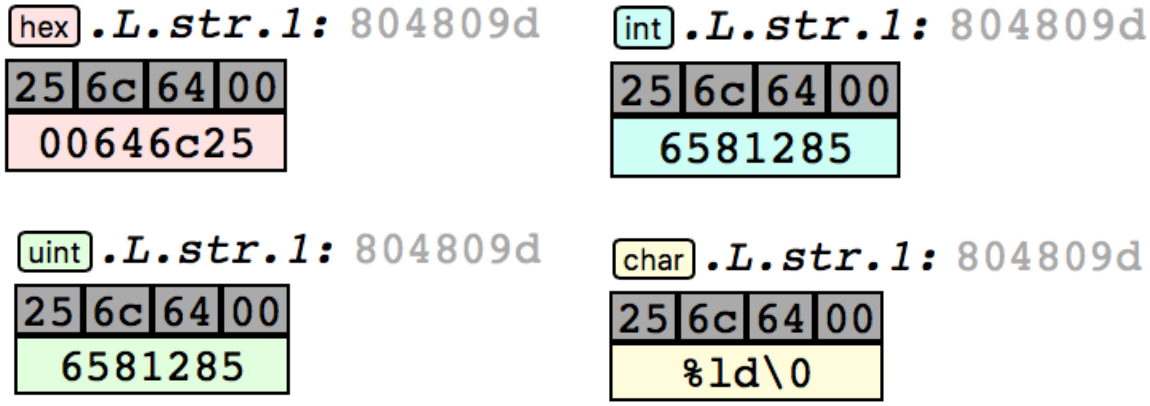


Figure A.2: Screenshots of decoding options available for the heap

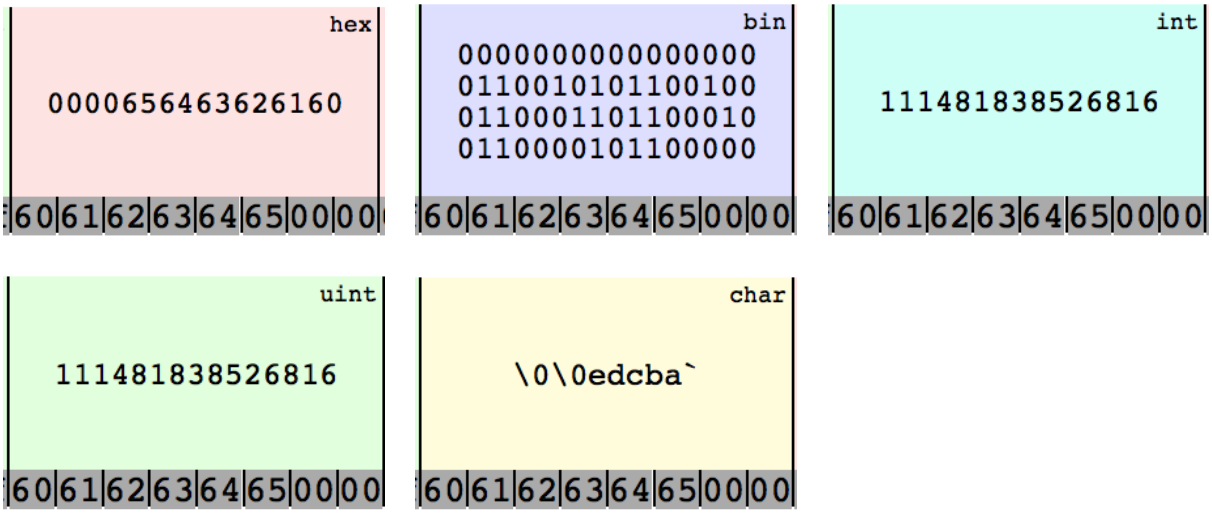


Figure A.3: Screenshots of decoding options available for registers

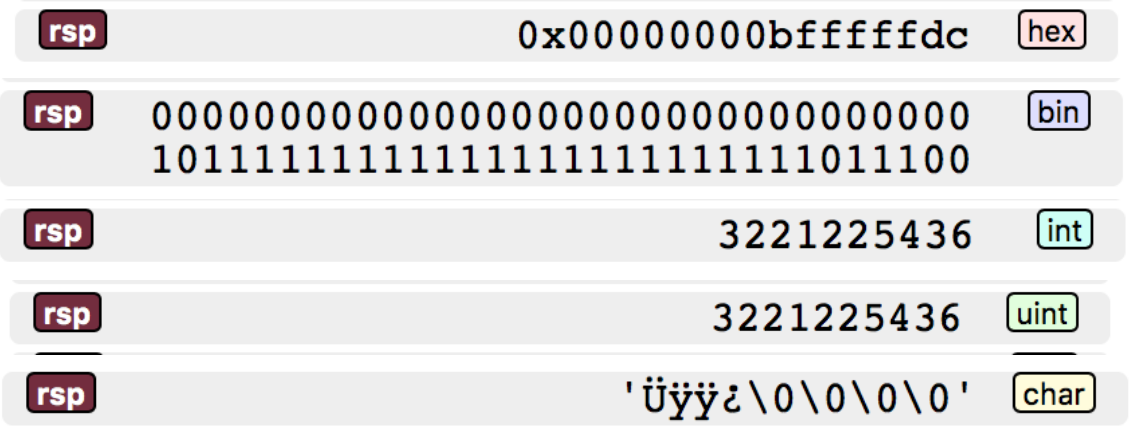


Figure A.4: Screenshot of the *Processable* welcome screen, with options to upload a file or select an example.

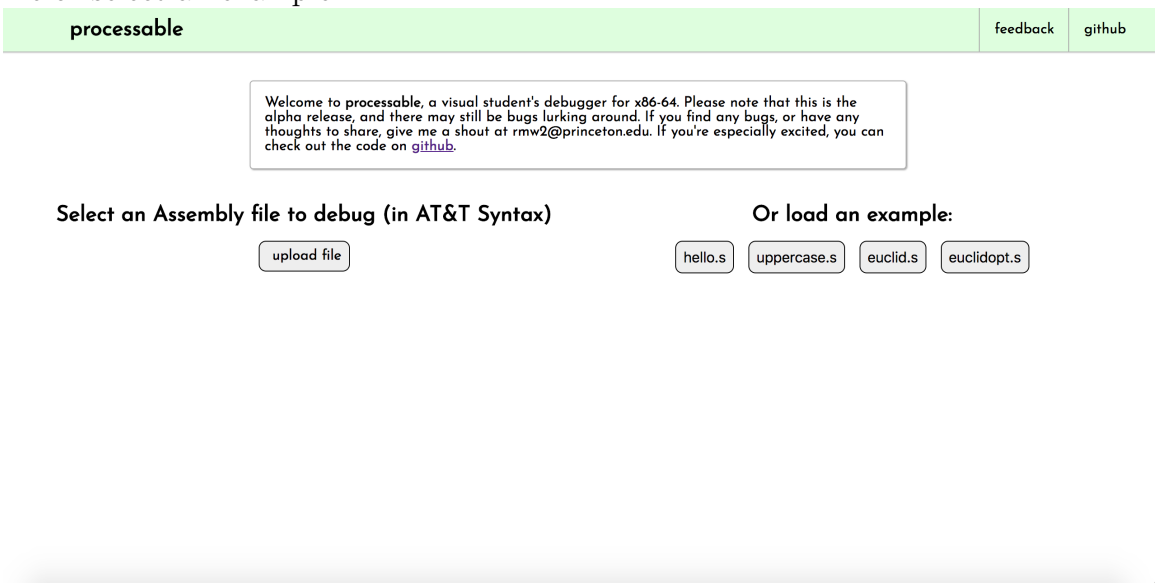


Figure A.5: Screenshot of the *Processable* interface during program execution, with the heap view toggled on

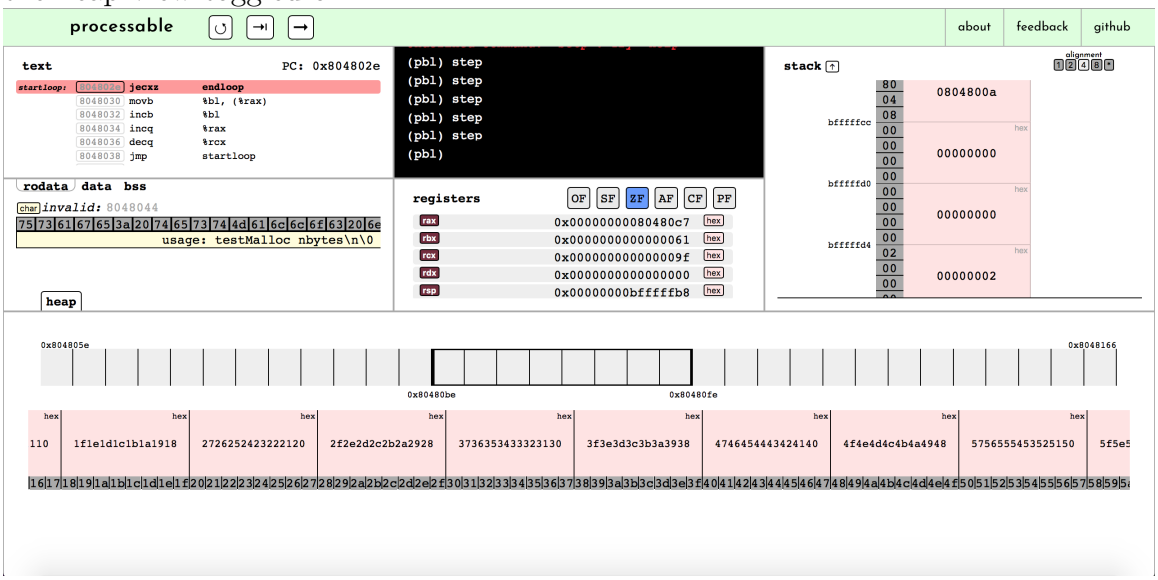
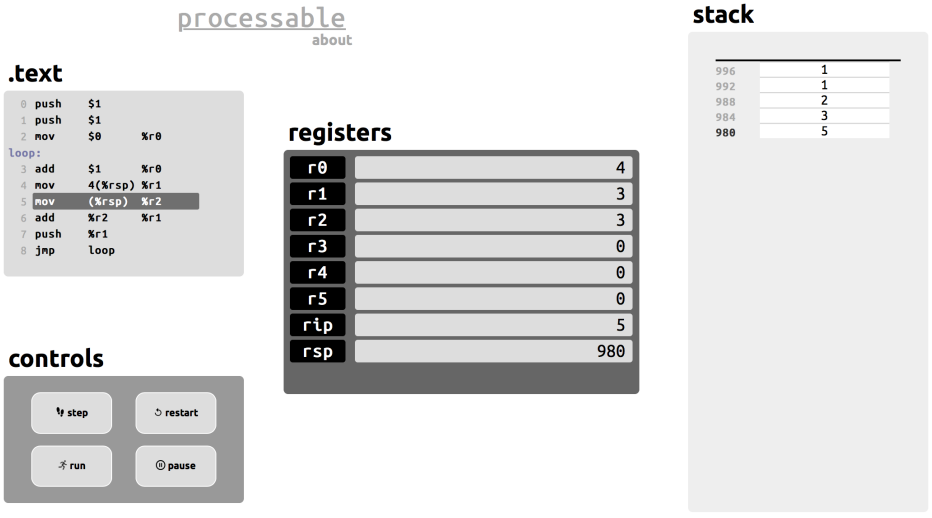


Figure A.6: A screenshot of the processable prototype, developed in July 2017

The prototype was developed as a single-file AngularJS application, using a mock register set, used only a single (4-byte integer) data type. It supported the `push`, `pop`, `mov`, `jmp`, and `add` instructions, and is still currently hosted at `https://rmw2.github.io/processable/prototype/`



Appendix B

Sample Assembly Programs

The following are the sample programs that are provided as examples with the application. These are four programs of increasing complexity, designed to showcase a few of the features of the debugger, and provide a gentle introduction to assembly language concepts.

Listing B.1: hello.s by Bob Dondero

```
1
2       .section ".rodata"
3
4   pcGreeting:
5       .asciz  "Hello␣World\n"
6
7       .section ".text"
8       .globl  main
9
10  main:
11      pushq   %rbp
12      movq    %rsp, %rbp
13
14      movabsq $pcGreeting, %rdi
15      call    printf
16      movabsq $4, %rax
17      addq    %rax, %rsp
18
19      movl    $0, %eax
20      movq    %rbp, %rsp
21      popq    %rbp
22      ret
```

Listing B.2: uppercase.s by Bob Dondero

```
1
2       .equ    LOWER_TO_UPPER, -32
3       .section ".bss"
4   cChar:
5       .skip   1
6
7       .section ".text"
8       .globl  main
9       .type   main,@function
10
11  main:
12      call    getchar
13      movb    %al, cChar
14
15      addb    $LOWER_TO_UPPER, cChar
16
17      movsbl  cChar, %edi
18      call    putchar
19
20      movl    $'\n', %edi
21      call    putchar
22
23      movl    $0, %eax
24      ret
```

Listing B.3: euclid.s compiled with Apple LLVM version 8.1 from euclid.c by Bob Dondero

```
1
2       .text
3       .globl  main
4       .type   main,@function
5   main:
6
7       pushq   %rbp
8   .Ltmp0:
9   .Ltmp1:
10      movq    %rsp, %rbp
11  .Ltmp2:
12      subq    $64, %rsp
13      movabsq $.L.str, %rdi
14      movl    $0, -4(%rbp)
15      movb    $0, %al
16      callq   printf
17      movabsq $.L.str.1, %rdi
18      leaq    -16(%rbp), %rsi
19      movl    %eax, -36(%rbp)
20      movb    $0, %al
21      callq   scanf
22      movabsq $.L.str, %rdi
23      movl    %eax, -40(%rbp)
24      movb    $0, %al
25      callq   printf
26      movabsq $.L.str.1, %rdi
27      leaq    -24(%rbp), %rsi
28      movl    %eax, -44(%rbp)
29      movb    $0, %al
30      callq   scanf
31      movq    -16(%rbp), %rdi
32      movq    -24(%rbp), %rsi
33      movl    %eax, -48(%rbp)
34      callq   gcd
35      movabsq $.L.str.2, %rdi
36      movq    %rax, -32(%rbp)
37      movq    -32(%rbp), %rsi
38      movb    $0, %al
39      callq   printf
40      xorl    %ecx, %ecx
41      movl    %eax, -52(%rbp)
42      movl    %ecx, %eax
43      addq    $64, %rsp
44      popq    %rbp
45      retq
46  .Lfunc_end0:
47      .size   main, .Lfunc_end0-main
48
49      .p2align    4, 0x90
50      .type   gcd,@function
```

```
gcd:

    pushq   %rbp
.Ltmp3:
.Ltmp4:
    movq    %rsp, %rbp
.Ltmp5:
    subq    $48, %rsp
    movq    %rdi, -8(%rbp)
    movq    %rsi, -16(%rbp)
    movq    -8(%rbp), %rdi
    callq   labs
    movq    %rax, -32(%rbp)
    movq    -16(%rbp), %rdi
    callq   labs
    movq    %rax, -40(%rbp)
.LBB1_1:
    cmpq    $0, -40(%rbp)
    je  .LBB1_3

    movq    -32(%rbp), %rax
    cqto
    idivq   -40(%rbp)
    movq    %rdx, -24(%rbp)
    movq    -40(%rbp), %rdx
    movq    %rdx, -32(%rbp)
    movq    -24(%rbp), %rdx
    movq    %rdx, -40(%rbp)
    jmp .LBB1_1
.LBB1_3:
    movq    -32(%rbp), %rax
    addq    $48, %rsp
    popq    %rbp
    retq

    .section    ".rodata"
.L.str:
    .asciz  "Enter␣an␣integer:␣"
.L.str.1:
    .asciz  "%ld"
.L.str.2:
    .asciz  "The␣gcd␣is␣%ld\n"
```

Listing B.4: euclidopt.s by Bob Dondero

```

    .section ".rodata"
cPrompt:
    .string "Enter␣an␣integer:␣"
cScanfFormat:
    .string "%ld"
cPrintfFormat:
    .string "The␣gcd␣is␣%ld\n"

    .section ".text"
    .equ    LABSSECOND, %r13
    .equ    LABSFIRST,  %r14
    .equ    LTEMP,      %r15
    .equ    LSECOND,    %rsi
    .equ    LFIRST,     %rdi

    .type   gcd,@function
gcd:
    pushq   %r13
    pushq   %r14
    pushq   %r15

    pushq   %rdi
    pushq   %rsi
    movq    LFIRST, %rdi
    call    labs
    movq    %rax, LABSFIRST
    popq    %rsi
    popq    %rdi

    pushq   %rdi
    pushq   %rsi
    movq    LSECOND, %rdi
    call    labs
    movq    %rax, LABSSECOND
    popq    %rsi
    popq    %rdi

loop1:
    cmpq    $0, LABSSECOND
    je      loopend1

    movq    LABSFIRST, %rax
    cqto
    idivq   LABSSECOND
    movq    %rdx, LTEMP

    movq    LABSSECOND, LABSFIRST

    movq    LTEMP, LABSSECOND

    jmp     loop1

loopend1:
    movq    LABSFIRST, %rax
    popq    %r15
    popq    %r14
    popq    %r13
    ret

    .equ    LGCD, 0
    .equ    L2,   8
    .equ    L1,  16
```

```
64
65      .equ    STACK_BYTECOUNT, 24
66
67      .globl  main
68      .type   main,@function
69
70  main:
71
72      subq    $8, %rsp
73      subq    $8, %rsp
74      subq    $8, %rsp
75
76      movq    $cPrompt, %rdi
77      movl    $0, %eax
78      call    printf
79
80      movq    $cScanfFormat, %rdi
81      leaq    L1(%rsp), %rsi
82      movl    $0, %eax
83      call    scanf
84
85      movq    $cPrompt, %rdi
86      movl    $0, %eax
87      call    printf
88
89      movq    $cScanfFormat, %rdi
90      leaq    L2(%rsp), %rsi
91      movl    $0, %eax
92      call    scanf
93
94      movq    L1(%rsp), %rdi
95      movq    L2(%rsp), %rsi
96      call    gcd
97      movq    %rax, LGCD(%rsp)
98
99      movq    $cPrintfFormat, %rdi
100     movq    LGCD(%rsp), %rsi
101     movl    $0, %eax
102     call    printf
103
104     movl    $0, %eax
105     addq    $STACK_BYTECOUNT, %rsp
106     ret
```

Appendix C

Materials used for Project Evaluation

Here we present the materials used to evaluate the program. These include the script used in the preliminary evaluation, the feedback form used in the alpha evaluation, and the scripts used to automate performance evaluation, all discussed in Chapter **??**.

C.1 Preliminary Evaluation Script

- This is the alpha version of Processable, a student's debugger and program tracer for x86-64
- Begin by opening the example program hello.s
  - This is the Processable interface with hello.s loaded.
  - Quickly touring the display, we have the header bar, which contains buttons to control the process; from left to right these are restart, step, and continue.
  - Below this, from left to right, we first have the text section, which shows the assembly code for this program, and the current value of the program counter. Breakpoints can be set by clicking on any address in the text section.
  - The next section is the console, which is bound to the current process' stdin, stdout, and stderr. Whenever the process is paused, it can also be used to issue commands to the debugger.
  - To begin the process, and push the arguments onto the stack, type the command "run" into the console, and press enter.
  - On the right is the stack section, which will show the value of every byte between the stack's origin and the current value of the stack pointer. The rightmost column of the stack decodes the values into hex, binary, signed integer, unsigned integer, or characters in groups of bytes of equal size.

The size of the grouping can be toggled by the group of buttons titled "alignment," and the decoding for any particular grouping can be toggled by clicking on the corresponding box.

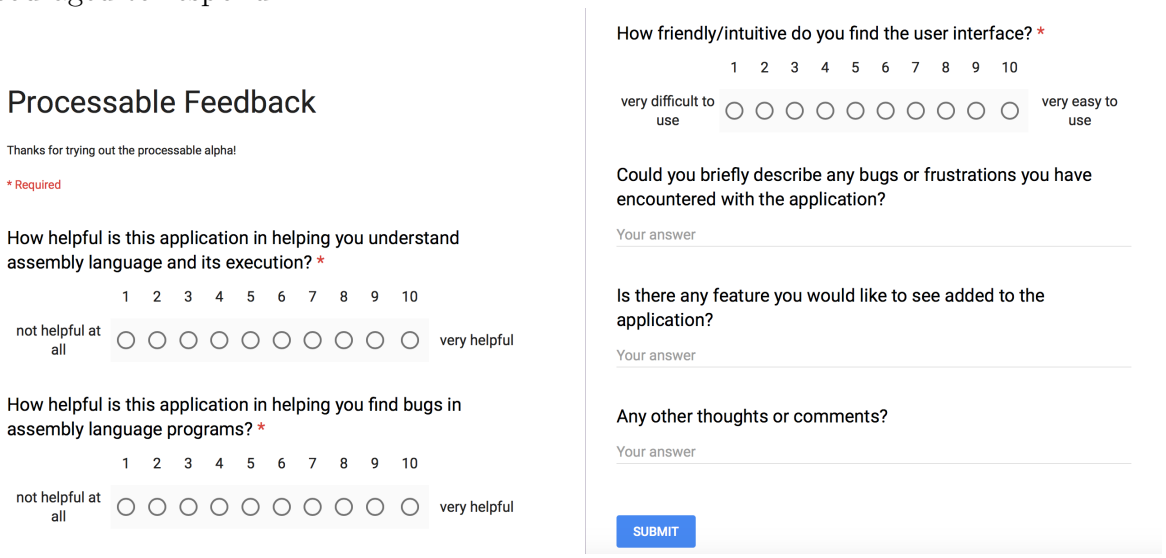
  - On the bottom left is the section for displaying static memory, with tabs to select the rodata, data, or bss section for display. Labels, addresses, and contents of memory are shown as bytes and decodings, and can have their decodings toggled by clicking on the colored button to the left of each labeled byte group.
  - Finally, the Registers are shown in the bottom middle area. Each row corresponds to a different group of overlapping registers, and the member of the group whose value is displayed can be toggled by the button to the left of the row showing the current register's name. Similarly, the decoding scheme for the register can be toggled by the colored button at the right of each row.
  - Press the continue button to allow the program to run until completion. Note that messages printed to stdout show up in the console.
  - Return to the home page

- Open the example program euclid.s
  - Start the program, and step or continue to the beginning of the main() function
    * How would you find the value of the program counter?
    * What is the value in register %rsp?
    * On the top of the stack is the return address, what would you expect that value to be and how would you verify this?
  - Set a breakpoint at the first call to printf, and continue until that point
    * What is the value of %rdi before the call to printf?
    * What is the string at that address?
  - Advance the program until the first call to scanf()
    * What is the value in %rdi, and what is the string that resides at that address?
    * Step past the call to scanf, and enter the value 18 when prompted
    * What is the return value in %rax, interpreted as a long integer?
    * What is the value stored at the address just found in %rsi, interpretted as an integer?
  - Continue past the next call to scanf(), enter the value 12 when prompted
  - Step until the beginning of the gcd() function
  - Set a breakpoint at the je instruction at address 0x804806e
    * Continue until the breakpoint; what is the value of the zero flag?

    - Repeatedly continue until the zero flag is set
- Set a breakpoint at the return statement from gcd
    - What is the long integer interpretation of the value about to be returned in %rax?
    - What is the address to which gcd will return, and what is the instruction at that address?

C.2 Student Feedback Form

The following google form was linked to by the alpha release, and testers were encouraged to respond.



Processable Feedback

Thanks for trying out the processable alpha!

* Required

How helpful is this application in helping you understand assembly language and its execution? *

1 2 3 4 5 6 7 8 9 10

not helpful at all ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ very helpful

How helpful is this application in helping you find bugs in assembly language programs? *

1 2 3 4 5 6 7 8 9 10

not helpful at all ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ very helpful

How friendly/intuitive do you find the user interface? *

1 2 3 4 5 6 7 8 9 10

very difficult to use ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ very easy to use

Could you briefly describe any bugs or frustrations you have encountered with the application?

Your answer

Is there any feature you would like to see added to the application?

Your answer

Any other thoughts or comments?

Your answer

SUBMIT

Bibliography

[1] The annual conference on innovation and technology in computer science education. *https://iticse.acm.org/*.

[2] Transcrypt. *https://www.transcrypt.org/documentation*, 2016.

[3] The luna language. *http://www.luna-lang.org/*, 2018.

[4] Cycling '74. Max 8 and max signal processing. *https://cycling74.com/products/max/*, 2018.

[5] Sanjeev Kumar Aggarwal and M. Sarath Kumar. *The Compiler Design Handbook*, chapter Debuggers for Programming Languages. CRC Press Ltd., 2003.

[6] Alexandro Sanchez Bach. Unicorn.js. *https://alexaltea.github.io/unicorn.js/*, 2016.

[7] Alexendro Sanchez Bach. Capstone js: The capstone disassembler framework in javascript. *https://alexaltea.github.io/capstone.js/*, 2014.

[8] Fabrice Bellard and Stefan Weil. *QEMU version 2.11.93 User Documentation*.

[9] Eli Bendersky. How debuggers work. *https://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1*, January 2011.

[10] Eli Bendersky. Stack frame layout on x86-64. *https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64*, 2011.

[11] Randal Bryant and David O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, 2016.

[12] Carlos Rafael Gimenes das Neves. Assembly x86 emulator. *http://carlosrafaelgn.com.br/Asm86/index.html?language=en*, 2013.

[13] Edgar Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3), March 1968.

[14] Edgar Dijkstra. On the cruelty of really teaching computing science (the sigcse award lecture). *Communications of the ACM*, 32:1403–1404, 1989.

[15] Inc. Facebook. React: A javascript library for building user interfaces. *https://reactjs.org*, 2018.

[16] Fastmetrics. Internet speeds by country. *https://www.fastmetrics.com/internet-connection-speed-by-country.php*, 2017.

[17] Mozilla Foundation. Arraybuffer. *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer*, 2018.

[18] Mozilla Foundation. Number. *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number*, 2018.

[19] Free Software Foundation. *The GNU Assembler*.

[20] Matt Godbolt. How it works: Compiler explorer. *https://xania.org/201609/how-compiler-explorer-runs-on-amazon*, 2016.

[21] Ilya Grigorik. Optimizing content effeciency. *https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/*, January 2018.

[22] Philip J. Guo. Online Python Tutor: Embeddable web-based program visualization for CS education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.

[23] Phillip Guo. Onlinepythontutor. *https://github.com/pgbovine/OnlinePythonTutor*, 2018.

[24] David Herman, Luke Wagner, and Alon Zakai. asm.js specification: Working draft. *http://asmjs.org/spec/latest/*, August 2014.

[25] Stefan Heule. How many x86-64 instructions are there anyway? *https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/*, March 2016.

[26] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

[27] Tony Jenkins. On the difficulty of learning to program. In *Proceedings of the 3rd Annual LTSN-ICS Conference*, 2002.

[28] Peter Kankowski. x86 machine code statistics. *https://www.strchr.com/x86_machine_code_statistics*, 2006.

[29] Ryan Kelly. Pypyjs. *http://pypyjs.org*, 2015.

[30] Michael Kerrisk. *ptrace(2) Linux Programmer's Manual*, September 2017.

[31] Johannes Kreidler. *Programming Electronic Music in Pd.* http://www.pd-tutorial.com/english/index.html, 2009.

[32] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. A multi-national study of reading and tracing skills in novice programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '04, pages 119–150, New York, NY, USA, 2004. ACM.

[33] Peter Marshall. Embedding v8. *https://github.com/v8/v8/wiki/Getting-Started-with-Embedding*, April 2017.

[34] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, pages 97–123, 1990.

[35] Thomas L. Naps and Guido Rößling. Exploring the role of visualization and engagement in computer science education. Technical report, Working group on Improving Educational Impact of Algorithm Visualization, 2002.

[36] Thomas L. Naps and Guido Rößling. A testbed for pedagogical requirements in algorithm visualizations. *Proceedings of the 7th Annual ITiCSE Conference*, 2002.

[37] Anh Quynh Nguyen and Hoang Vu Dang. Unicorn: Next generation cpu emulator framework. In *Blackhat Conference USA*, 2015.

[38] Pierre Quentel. Brython. *https://brython.info*, 2015.

[39] Alexander Sandler. How debuggers work. *http://www.alexonlinux.com/how-debugger-works*, 2008.

[40] J. Sorva, V. Karavirta, and L. Malmi. A review of generic program visualization systems for introductory programming education. *ACM Trans. Comput. Educ.*, 2013.

[41] Juha Sorva. *Visual Program Simulation in Introductory Programming Education.* PhD thesis, Aalto University, 2012.

[42] Rob Whitaker. Assemblance: An interactive assembly explorer. *Spring Independent Work, Princeton University*, 2017.

[43] Alon Zakai. Emscripten: an llvm-to-javascript compiler. *Proceedings of the 26th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–312, 10 2011.

[44] Alon Zakai. Emscripten file system api. *https://kripken.github.io/emscripten-site/docs/api_reference/Filesystem-API.html*, 2015.