A5: rum

Design Documentation

CSC 411

Marceline Kelly

November 17, 2023

1 Overview

This assignment involves the creation of a primitive virtual machine. This machine has two forms of memory: a set of eight registers that the virtual processor can directly access as well as a segmented memory space. Each segment contains a series of words that can be accessed using a unique identifier. The machine can execute programs constructed from a basic set of instructions, loaded from the "Universal Machine" binary format. The machine also supports basic ASCII input and output, akin to standard I/O on a modern machine.

2 Memory Segments

One might initially consider representing the VM's memory using a statically-sized array. While this would incur a substantial up-front allocation cost, accessing memory during execution would be relatively performant. The problem with this approach, however, is that our memory segments do not have a fixed size.

As a result, a dynamically-sized structure is a better option, in this case. This structure must also be indexed and iterable (sorry, hash tables). Considering that each word is a 32-bit integer, it makes sense that a segment would take the form of a `Vec` of `u32`'s.

We can apply this thought process to our general memory space as well, representing it as a large `Vec`. Noting that memory can be unmapped at will and in no particular order, we consider any segment with length 0 (an empty segment) to be "free" and able to store new data. To free a segment, then, is simply to clear the `Vec` that represents it. Conversely, the size of a memory segment is equal to the number of elements in its `Vec`. As the program requires more memory, more segments can be allocated on the fly.

This yields the layout:

- `Word: u32`
- `Segment: Vec<Word>`

- `Memory: Vec<Segment>`

This segmented memory is accompanied by a program counter (a `Word`) and eight registers (stored in a `Word` array). As such, the machine's state is stored almost entirely on the heap, save for the program counter and registers, which live on the stack. This dynamic structure ensures that the VM will be able to handle the needs of almost any UM program provided.

2.1 Invariants

1. The memory segment at index 0 stores the program and is always mapped.
2. The program counter holds a value between 0 and $n-1$, where n is the length of the program in words.
3. In any segment that has been initialized but has not yet received any data, all words equal zero.

3 Architecture

The program will consist of the following modules:

1. `rumload`
    - Unpacks the raw binary file into a series of 32-bit instruction words
2. `rumdis`
    - Separates each instruction provided by `rumload` into an opcode and associated values
3. `rummem`
    - Allocates and manages the memory of the virtual machine (both registers and segment space)
    - Loads the operations provided by `rumdis` into virtual memory
4. `rumrun`
    - Executes the loaded program
    - Interfaces with `rummem` for memory management
    - Keeps track of the current instruction via the program counter
5. `rumio`
    - Accepts data from standard input
    - Sends data to standard output
6. `main`
    - The entry point of the program
    - Coordinates the primary control flow of the program

4 Test Plan

4.1 Unit testing

This program will support a rigorous suite of unit tests. These tests are designed to ensure that each individual function operates as expected. They also test the program's behavior

4.1.1 rumload

- Is the size of the loaded data exactly the same as that of the input data?
- Is each word unmodified from its representation on disk?

4.1.2 rumdis

- Does the program halt gracefully when presented with an invalid opcode (14 or 15)?

4.1.3 rummem

- If a UM program requests resources that are not available, does execution halt in the expected manner?
- Does the program correctly re-use unmapped memory?
- When more memory is requested, does the program map the first (from index 1) free segment?

4.1.4 rumrun

- Is a program executed in the exact order specified by the binary?
- Do jumps (using instruction 13) land at the correct instruction?

4.1.5 rumio

- Does the input prompt terminate when and only when explicitly signaled to?
- Does the program halt gracefully when asked to output a value larger than 255?
- Does the program halt gracefully when given an input value greater than 255 (i.e. a non-ASCII character)?

4.2 Functional testing

The above unit tests will be supplemented by comprehensive start-to-finish testing—that is, testing that the program as a whole performs a certain way based on the provided input. The use of a debugger will assist in finding bugs that the unit tests may have missed, should the program behave unexpectedly.