CSC 411 Assignment: A Universal Virtual Machine

Design due Friday, November 17th at 11:59 PM. *Implementation* due Friday, December 1st at 11:59 PM.

Purpose and overview

The purpose of this assignment is to understand virtual-machine code (and by extension machine code) by writing a software implementation, in Rust, of a simple virtual machine, which we will call `rum`.

Specification of the Universal Machine

Machine State

The UM has these components:

- Eight general-purpose registers holding one word each
- A very large address space that is divided into an ever-changing collection of *memory segments*. Each segment contains a sequence of words, and each is referred to by a distinct 32-bit identifier. The memory is *segmented* and *word-oriented*; you cannot load a byte
- An I/O device capable of displaying ASCII characters and performing input and output of unsigned 8-bit characters
- A 32-bit program counter

One distinguished segment is referred to by the 32-bit identifier `0` and stores the *program*. This segment is called the '0' segment.

Notation

To describe the locations on the machine, we use the following notation:

- Registers are designated `$r[0]` through `$r[7]`
- The segment identified by the 32-bit number i is designated `$m[i]`. The '0' segment is designated `$m[0]`.

- A word at offset n within segment i is designated `$m[i][n]` You might refer to i as the *segment number* and n as the *address within the segment*.

Initial state

The UM is initialized by providing it with a *program*, which is a sequence of 32-bit words. Initially:

- The '0' segment `$m[0]` contains the words of the program.
- A segment may be *mapped* or *unmapped*. Initially, `$m[0]` is mapped and all other segments are unmapped.
- All registers are zero.
- The program counter points to `$m[0][0]`, i.e., the first word in the '0' segment.

Execution cycle

At each time step, an instruction is retrieved from the word in the 0 segment whose address is the program counter. The program counter is advanced to the next word, if any, and the instruction is then executed.

Instructions' coding and semantics

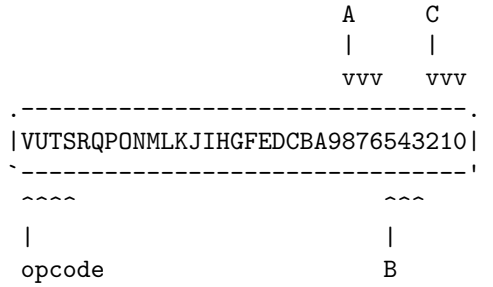
The Universal Machine recognizes 14 instructions. The instruction is coded by the four most significant bits of the instruction word. These bits are called the *opcode*.

Opcode	Operator	Action
0	Conditional Move	if `$r[C]` $\neq 0$ then `$r[A] := $r[B]`
1	Segmented Load	`$r[A] := $m[$r[B]][$r[C]]`
2	Segmented Store	`$m[$r[A]][$r[B]] := $r[C]`
3	Addition	`$r[A] := ($r[B] + $r[C]) mod` 2^{32}
4	Multiplication	`$r[A] := ($r[B]` $\times$ `$r[C]) mod` 2^{32}
5	Division	`$r[A] := ($r[B]` $\div$ `$r[C])` (integer division)
6	Bitwise NAND	`$r[A] :=`$\neg$`($r[B]`$\wedge$`$r[C])`
7	Halt	Computation stops
8	Map segment	A new segment is created with a number of words equal to the value in `$r[C]`. Each word in the new segment is initialized to zero. A bit pattern that is not all zeroes and does not identify any currently mapped segment is placed in `$r[B]`. The new segment is mapped as `$m[$r[B]]`.
9	Unmap segment	The segment `$m[$r[C]]` is unmapped. Future Map Segment instructions may reuse the identifier `$r[C]`.

Opcode	Operator	Action
10	Output	The value in `$r[C]` is displayed on the I/O device immediately. Only values from 0 to 255 are allowed.
11	Input	The UM waits for input on the I/O device. When input arrives, `$r[c]` is loaded with the input, which must be a value from 0 to 255. If the end of input has been signaled, then `$r[C]` is loaded with a full 32-bit word in which every bit is 1.
12	Load Program	Segment `$m[$r[B]]` is duplicated, and the duplicate replaces `$m[0]`, which is abandoned. The program counter is set to point to `$m[0][$r[C]]`. If `$r[B]=0`, the load program operation should be extremely quick, as this is effectively a jump.
13	Load Value	See semantics for "other instruction".

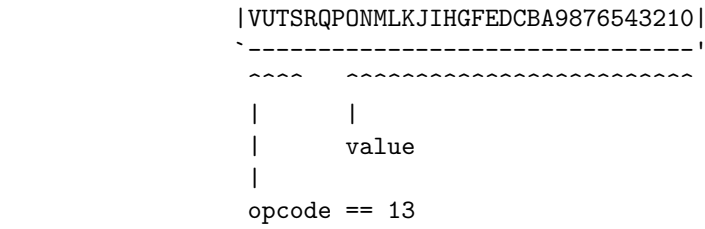
Three-register instructions

Most instructions operate on three registers. The registers are identified by number; we'll call the numbers A, B, and C. Each number coded as a three-bit unsigned integer embedded in the instruction word. The register C is coded by the three least significant bits, the register B by the three next more significant than those, and the register A by the three next more significant than those:



One other instruction

One special intruction, with opcode 13, does not describe registers in the same way as the others. Instead, the three bits immediately less significant than opcode describe a single register A. The remaining 25 bits indicate a value, which is loaded into `$r[A]`.



Failure modes

The behavior of the Universal Machine is not fully defined; under circumstances detailed below (and only these circumstances), the machine may *fail*.

- If at the beginning of a machine cycle the program counter points outside the bounds of `$m[0]`, the machine may fail.
- If at the beginning of a cycle, the word pointed to by the program counter does not code for a valid instruction, the machine may fail.
- If a segmented load or segmented store refers to an unmapped segment, the machine may fail.
- If a segmented load or segmented store refers to a location outside the bounds of a mapped segment, the machine may fail.
- If an instruction unmaps `$m[0]`, or if it unmaps a segment that is not mapped, the machine may fail.
- If an instruction divides by zero, the machine may fail.
- If an instruction loads a program from a segment that is not mapped, then the machine may fail.
- If an instruction outputs a value larger than 255, the machine may fail.

In the interests of performance, *failure may be treated as an* ***unchecked*** *run-time error*. Even a core dump is OK, though a segfault is not.

Resource exhaustion

If a UM program demands resources that your implementation is not able to provide, and if the demand does not constitute *failure* as defined above, your only recourse is to halt execution with a checked run-time error.

Advice on the implementation

This problem presents two challenges:

- Emulating a 32-bit machine on 64-bit hardware
- Choosing abstractions that are efficient enough

There are also some pitfalls:

- It's easy to forget to test the input instruction, or to test it inadequately.

- It's easy to let the amount of memory allocated grow without bound. If you fall into this pit, you won't be able to run any nontrivial UM programs.
- It's easy to allocate more memory than is really needed to solve the problem. If you fall into this pit, you'll find that nontrivial UM programs run very, very slowly.

And finally there are some useful things to know:

- In the Rust programming language running on modern hardware, addition and multiplication of values of type `u`k keeps only the least significant k bits of each result. Mathematically, the least significant k bits of a value is equivalent to that value modulo 2^k.
- Just as in C, in the Rust programming language running on AMD64 hardware, division of signed and unsigned integer types rounds toward zero. (For signed types, rounding toward zero is a crime.
  Rounding toward minus infinity would be much more useful. Alas, we are stuck with this legacy feature.)

Emulating a 32-bit machine: Simulating 32-bit segment identifiers

On a 32-bit machine, you could simply use a 32-bit pointer as a segment identifier and have `malloc` do your heavy lifting. On the 64-bit machine, you will need an abstraction that maps 32-bit segment identifiers to actual sequences of words in memory. (Any representation of segments I can think of requires at least 64 bits to store.)

Plan to reuse 32-bit identifiers that have been unmapped. One way is to store them in some collection such as a `Vec<T>`.

Efficient abstractions

Your choice of abstractions can easily affect performance of your UM by a factor of 1000. We will provide a benchmark that a well-optimized UM should be able to complete in about 1 second; a UM designed without regard to performance might take 20 minutes on the same benchmark. To get decent performance, focus on two decisions:

- Think about what parts of the machine state are most frequently used, and to the degree you can, be sure that frequently used state is in local variables that the compiler can put in registers. (You can verify placement in registers by using `objdump`.)

For this assignment, you should trust the Rust compiler to get you reasonable performance; you'll have the opportunity to speed up your UM in the next assignment.

Avoid common mistakes

Following this advice will help you avoid common mistakes:

- The Input instruction is supposed to read any C `char` as an integer in the range 0 to 255. In Rust, this is any `u8` value. Standard printable ASCII characters live in the range 33 to 126. You'll want to test on a larger range of inputs. One source of inputs is the special file `/dev/urandom`. Used together with the `dd` and `cmp` commands, it should provide an easy way to test more characters.
- If the `um` binary is called from the command line in a way that violates its contract, it should print a suitable message to standard error, and it should exit nonzero.

What we expect of you

Your design and its documentation

The documentation of your design should include

- the representation of segments and its invariants.
- the architecture and test plan.

For this assignment in particular, we have high expectations for your test plan.

In this assignment we are raising the bar for your design work:

- Excellent design documentation will say what data structure will be used to represent each part of the state of a Universal Machine, and where that data structure will be stored.
- Excellent design documentation will show how the parts will be organized, and in particular, how the implementation of the Universal Machine will be decoupled from the program loader and the `main()` function, so that the Universal Machine can be unit tested.

Implementation

We expect you to write a complete and correct implementation of the Universal Machine. Moreover, we expect it to be efficient enough to execute a UM benchmark of 50 million instructions in less than 100 CPU seconds on the homework server; that's half a million instructions per second.

The UM is a virtual machine. One of the purposes of virtualization is to insulate the real ("host") hardware from bad behavior by client ("guest") software.
For example, in the Amazon Elastic Compute Cloud, no matter how badly the client binaries behave, Amazon makes sure that when a virtual server halts, all machine resources are recovered. (Any other strategy would leave Amazon with machine resources that aren't earning any revenue.) Similarly, no matter how

badly a UM client behaves, *your implementation must ensure that, when the UM finishes running, all available machine resources are recovered.*

For testing, you will find it useful to implement the UM as a library. However, we will be evaluating a command-line version which is a command-line program that expects exactly one argument: the name of a file containing a UM program. *When a UM program is stored in a file, words are stored using big-endian byte order.*

The UM "I/O device" should be implemented using standard input and standard output.

What to submit

Design

On Gradescope, please submit a PDF file describing your design.

Implementation

On Gradescope, please submit:

- Your `rum` project. By now, you should know how to zip it up to exclude `.git` and `target` and `Cargo.lock`
- A `README` file which
  - Identifies you and your programming partner by name
  - Acknowledges help you may have received from or collaborative work you may have undertaken with others
  - Identifies what has been correctly implemented and what has not
  - Briefly enumerates any significant departures from your design
  - Succinctly describes the architecture of your system. Identify the modules used, what abstractions they implement, what secrets they know, and how they relate to one another. Avoid narrative descriptions of the behavior of particular modules.
  - Explains how long it takes your UM to execute 50 million instructions, and how you know
  - Says approximately how many hours you have spent analyzing the assignment
  - Says approximately how many hours you have spent preparing your design
  - Says approximately how many hours you have spent solving the problems after your analysis

On a 32-bit machine, most experienced programmers can understand the Universal Machine specification and build an implementation in a total of two hours. On a 64-bit machine, the need to emulate 32-bit segmented memory (since you can't use 32-bit pointers as segment identifiers) will add an hour of work. We expect you to take about two hours to analyze the assignment, four hours to prepare your design and unit tests, and four hours to build a working implementation.

My implementation is a little less than 300 lines of Rust code; almost one-third is devoted to conversions between 64-bit pointers and 32-bit Universal machine identifiers. Reading arguments and loading the initial program takes about 35 lines, so the Universal Machine itself is just around 150 lines of code.

What we provide for you

We provide the following useful items:

- In `/csc/411/um` you will find a small collection of Universal Machine binaries that you can use for final system test. The binaries are described by a README file.
- They are also available at `https://github.com/ndaniels/rum-binaries`
- The `rumdump` you wrote in lab will dump the contents of a Universal Machine binary, as in:

```
rumdump cat.um
rumdump midmark.um | less
```

Alternatively, the program `/usr/local/bin/umdump` will dump the contents of a Universal Machine binary. It is the closest counterpart I have to `objdump`.

- There is a working `bitpack` on EdStem that you may use if your bitpack from the Arith assignment doesn't work. It only needs to work for unsigned values, and if you use mine, you must mention so in the Readme.