COMP40 Assignment: A Universal Virtual Machine

Contents

1 Purpose

The purpose of this assignment is to understand virtual-machine code (and by extension machine code) by writing a software implementation of a simple virtual machine. You will demonstrate your ability to design, document, and implement a program with a clean modular structure. You will also begin to learn how the structural choices you make affect the performance of your programs. The primary goal of your design and implementation is clean structure, but there are stated performance goals you must achieve to receive credit.

2 Summary of the Assignment

You will be building an executable named `um` that emulates a "Universal Machine" (UM). The executable takes a single argument: the pathname for a file (typically with a name like `some_program.um`) that contains machine instructions for your emulator to execute. During execution, the UM uses `stdin` and `stdout` for I/O, so you might run your emulator with:

```
./um some_program.um < testinput.txt > output.txt
```

Most of the specifications for your program consist of the specifications of the UM itself. You will find these outlined in the section below. These specifications describe the structure of the UM (how many registers, etc.) as well as the operation of each UM instruction. Start by reading and making sure you understand the UM specifications below. Then, before you design or code anything, read the rest of the spec.

3 Specification of the Universal Machine

3.1 Machine State

The UM has these components:

- Eight general-purpose registers holding one 32-bit word each.
- An 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. The device uses `stdin` and `stdout` to implement the UM's I/O instructions.
- 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.*

3.2 Notation

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

- Registers are designated $\$\mathtt{r}[0]$ through $\$\mathtt{r}[7]$.
- The segment identified by the 32-bit number s is designated $\$\mathtt{m}[s]$. The 0 segment is designated $\$\mathtt{m}[0]$.
- A word at offset n within segment s is designated $\$\mathtt{m}[s][n]$. You might refer to s as the *segment number* and n as the *address within the segment*.

3.3 Initial state

The UM is initialized by providing it with a *program*, which is a sequence of 32-bit words. *When a UM program is stored in a file, the UM instructions are stored using big-endian byte order.* Thus the high order four bits of the first byte of the file will be the first operation code for that program. Initially

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

3.4 Execution cycle

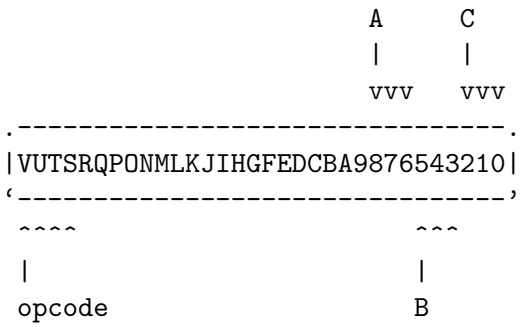
At each time step, an instruction is retrieved from the word in the 0 segment whose address is the program counter. The program counter is advanced to the next word, if any, and the instruction is then executed.

3.5 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*. The remaining 28 bits are used differently based on the instruction being encoded. This section explains these encodings, and the semantics of each instruction are given in Figure **??**.

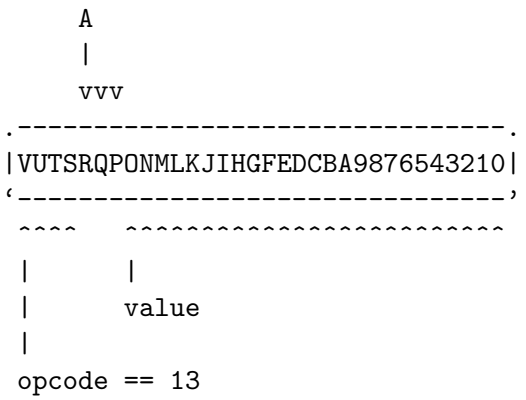
3.5.1 Three-register instructions

Most instructions operate on up to three registers (instructions 7 through 12 ignore one or more of these registers). The registers are identified by number; we'll call the numbers A, B, and C. We code each number as a three-bit unsigned integer embedded in the instruction word, with register C's code starting at the 0th bit, register B's starting at the 3rd bit, and register A's starting at the 6th bit. This image depicts where each register's code and the opcode lie within a 3-register instruction word:



3.5.2 One other instruction

One special instruction, with opcode 13, does not describe registers in the same way as the others. Instead, the three bits immediately less significant than the opcode describe a single register A. The remaining 25 bits are an unsigned binary value. This instruction sets $\$\mathtt{r}[A]$ to that value.



3.6 Failure modes

The behavior of the Universal Machine is not fully defined; under certain circumstances, 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. Go wild!

If any of the following circumstances occur (and only these circumstances), the machine is allowed to *fail*:

- At the beginning of a machine cycle, the program counter points outside the bounds of $\$\mathtt{m}[0]$.
- At the beginning of a machine cycle, the program counter points to a word that does not code for a valid instruction.
- A segmented load or segmented store refers to an unmapped segment.
- A segmented load or segmented store refers to a location outside the bounds of a mapped segment.

Number	*Operator*	*Action*
0	Conditional Move	if $\$\mathtt{r}[C] \neq 0$ then $\$\mathtt{r}[A] := \$\mathtt{r}[B]$
1	Segmented Load	$\$\mathtt{r}[A] := \$\mathtt{m}[\$\mathtt{r}[B]][\$\mathtt{r}[C]]$
2	Segmented Store	$\$\mathtt{m}[\$\mathtt{r}[A]][\$\mathtt{r}[B]] := \$\mathtt{r}[C]$
3	Addition	$\$\mathtt{r}[A] := (\$\mathtt{r}[B] + \$\mathtt{r}[C]) \bmod 2^{32}$
4	Multiplication	$\$\mathtt{r}[A] := (\$\mathtt{r}[B] \times \$\mathtt{r}[C]) \bmod 2^{32}$
5	Division	$\$\mathtt{r}[A] := \lfloor \$\mathtt{r}[B] \div \$\mathtt{r}[C] \rfloor$
6	Bitwise NAND	$\$\mathtt{r}[A] := \neg(\$\mathtt{r}[B] \wedge \$\mathtt{r}[C])$
7	Halt	Computation stops.
8	Map Segment	A new segment is created with a number of words equal to the value in $\$\mathtt{r}[C]$. Each word in the new segment is initialized to 0. A bit pattern that is not all zeroes and that does not identify any currently mapped segment is placed in $\$\mathtt{r}[B]$. The new segment is mapped as $\$\mathtt{m}[\$\mathtt{r}[B]]$.
9	Unmap Segment	The segment $\$\mathtt{m}[\$\mathtt{r}[C]]$ is unmapped. Future Map Segment instructions may reuse the identifier $\$\mathtt{r}[C]$.
10	Output	The value in $\$\mathtt{r}[C]$ is written to the I/O device immediately. Only values from 0 to 255 are allowed.
11	Input	The universal machine waits for input on the I/O device. When input arrives, $\$\mathtt{r}[C]$ is loaded with the input, which must be a value from 0 to 255. If the end of input has been signaled, then $\$\mathtt{r}[C]$ is loaded with a full 32-bit word in which every bit is 1.
12	Load Program	Segment $\$\mathtt{m}[\$\mathtt{r}[B]]$ is *duplicated*, and the duplicate replaces $\$\mathtt{m}[0]$, which is abandoned. The program counter is set to point to $\$\mathtt{m}[0][\$\mathtt{r}[C]]$. If $\$\mathtt{r}[B] = 0$, the load-program operation is expected to be extremely quick.
13	Load Value	See semantics for "other instruction" in Section **??**.

Figure 1: Semantics of UM instructions

- An instruction unmaps either $\texttt{\$m}[0]$ or a segment that is not mapped.
- An instruction divides by zero.
- An instruction loads a program from a segment that is not mapped.
- An instruction outputs a value larger than 255.

3.7 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 in Section **??**, your only recourse is to halt execution with a checked run-time error.

3.8 Contract violations

If the `um` binary is called from the command line in a way that violates its contract, it must print a suitable message to standard error, *and* it must exit with `EXIT_FAILURE`.

4 Advice on the implementation

4.1 A good VM

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. No matter how badly a UM client behaves, *your implementation must ensure that, when the UM finishes running, all available machine resources are recovered* (unless it is a case in which the machine is allowed to fail as described in Sec **??**).

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

If you were emulating the UM on a 32-bit machine, you could simply use a 32-bit pointer as a segment identifier and have `malloc` do your heavy lifting. Since we're using 64-bit machines, you will need an abstraction that maps 32-bit segment identifiers (for the UM) to actual sequences of words in memory (on your physical machine). You cannot simply use the result of a `malloc` call as a segment identifier.

Plan to reuse 32-bit identifiers that have been unmapped. One way is to store them in one of Hanson's sequences (`Seq_T`). A wonderful C99 trick is that you can cast an `uintptr_t` to a `void *`, so statements like

```
Seq_addlo(ids, (void *)(uintptr_t)id);
return (uint32_t)(uintptr_t)Seq_remlo(ids);
```

might be useful. The above assumes `id` is of type `uint32_t`. The casts are needed, not to change the bit representation of the number, but to suppress compiler warnings about assignments between pointers and integers of differing sizes.

4.3 Efficient abstractions

Your choice of abstractions can easily affect performance of your UM by a factor of 1000. We will provide a benchmark (named `midmark`) that your UM should be able to complete in 60 seconds or less on our lab machines; a UM designed without regard to performance might take 20 minutes on the same benchmark.

Remember: mapping and unmapping segments in a UM program serves many of the same purposes as using `malloc` and `free` in a C program; your emulator should be able to handle very large numbers of segment creations and destructions with good performance.

To get decent performance while maintaining clean modular code (the latter is more important for this assignment), focus on three 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`.)
- Decide where you want to use safe abstractions like the ones in the CII library and where you want to use unsafe techniques like pointer arithmetic. Your Universal Machine is permitted to "fail" by misusing a C pointer.
- In some cases you can achieve the benefits of procedural abstraction and type checking without any run-time overhead by writing `static inline` procedures. If such procedures are reusable, it can be appropriate to put them in a `.h` file.

4.4 Controlling use of CPU and memory

When we test your UM, we will limit its CPU time. Because you can easily overlook how much time your UM needs, we provide the command `cpu-limited`, which ensures that your UM runs within specified limits. For example, to run for at most 10 seconds, you can run

```
cpu-limited 10 ./um midmark.um
```

If your UM exceeds its limits, this program will halt it. For more information about forced halts or other failures, run

```
catch-signal cpu-limited 10 ./um midmark.um
```

These commands are not documented, but they are available when you run `use comp40`.

Note that if you let the amount of memory allocated grow without bound, or if you allocate more memory than is really needed to run your UM, you either won't be able to run any nontrivial UM programs or you'll find that nontrivial UM programs run extremely slowly.

4.5 General Hints, Info, and Reminders

- At this point in COMP 40 we expect you to demonstrate all you've learned about building programs that exhibit modularity and abstraction. Look back at your previous assignments' S&O grades and make sure this assignment incorporates any feedback that's been provided!
- It's easy to forget to test the input instruction adequately. This instruction is supposed to read any C `char` as an integer in the range 0 to 255. 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.
- In the C programming language, addition and multiplication of values of type `uint`k`_t` 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.
- Things like wrong exit codes, incorrect use of Checked or Unchecked Runtime Errors, and inappropriate output to stderr tend to cost students lots of credit on otherwise impressive submissions. REREAD ALL THE INSTRUCTIONS AND CHECK ALL THE DETAILS BEFORE YOU SUBMIT!
- Since your program is always given the name of a file (as opposed to a pipe or keyboard device mapped to `stdin`), it is possible to determine the length of the file before opening it. Try:

  ```
  man 2 stat
  ```

  for some hints. Whether you will find `stat` useful may depend on the strategy you adopt for reading in the UM program; there are several reasonable ways to do it.

5 What we provide for you

We provide the following useful items:

- A small collection of Universal Machine binaries that you can use for final system test. The binaries are described by a README file.
- In `/comp/40/build/include` and `/comp/40/build/lib` respectively, you will find header file `um-dis.h` and corresponding library `libum-dis.a`, which you can link with `-lum-dis -lcii`. This library exports a single function `Um_disassemble`, which you may find useful for debugging your UM. Check out the header file for information about the function.
- Program `/comp/40/bin/umdump` will dump the contents of a Universal Machine binary, as in

  ```
  umdump cat.um
  umdump midmark.um | less
  ```

- There is a working `libbitpack.a` in `/comp/40/build/lib` that you can link against with `-lbitpack`. If you use this linking flag when compiling, **it must come before any other linking flags in your compilation command.**
- An upcoming lab will give you some ideas about unit-testing your Universal Machine.

6 Organizing and submitting your work

6.1 Submitting your design document

Across your previous assignments, you have honed your skills at many aspects of the design process. This is your chance to show us the total of what you have learned. Your design submission must contain the following components:

Architecture: We expect your final implementation to exemplify two of the main themes in this course: abstraction and modularity. The architecture portion of your design submission should clearly convey (1) how you plan to break your implementation into modules, (2) which Hanson data structures each module will make use of (if any), and (3) how these modules will interact with each other. Interactions between modules should be specified as function contracts. Be clear. Draw pictures. Help us give you feedback on whether or not your plan is likely to yield a functioning implementation.

Implementation Plan: You've been here before. Show us that you can thoughtfully break this assignment down into a logical progression of small, testable steps.

Testing Plan: It is fine to interleave your testing plan with your implementation plan. However, keep in mind that testing for the UM will involve tests for two different aspects of your solution:

1. The UM instruction set (i.e., does the "Add" instruction get disassembled and executed properly)
2. Your UM segment abstraction (i.e., does the emulator map/unmap/load segments without losing memory or segfaulting)

Instruction set tests will be in the form of compiled `.um` files. Your design submission need only outline your plans for such tests, since you will be submitting the tests themselves as part of your final submission (see Section **??**).

Your architecture tests, by contrast, will be in the form of `C` functions that validate aspects of your architecture. You should provide the prototypes and descriptions for these functions in your `.pdf` submission. Specifically, for each architecture test you plan to create, list the following information about that test in your submission:

1. A C function prototype for the eventual function you will write to perform the test.
2. Documentation above that prototype indicating what the eventual function will test and how it will use other functions and modules in your architecture's overall design.

In essence, we are looking for you to demonstrate a high level of synchronicity between your architecture design and your plan to test that architecture.

Please submit your design in a file called `design.pdf` via Gradescope.

6.2 Submitting your instruction set tests and implementation

Your final submission will consist of (1) your instruction set unit tests and (2) your UM implementation. The expectations for both of these components are described in the following sections. The submission of both components can be accomplished with the following, single command:

`submit40-um`

6.2.1 Instruction set tests

Your design submission will outline how you plan to test the UM instruction set. Your final submission should include the actual unit tests that you built to accomplish this. Each unit test may rely on the correct functioning of instructions from previous unit tests. For example:

- Your first unit test might test only Halt.
- Your second unit test might test Output and Halt.
- Your third test might Output, Load Value, and Halt.

And so on.

Every unit test must include a compiled UM binary (you will write the code to create these). The filename of each of these binaries *must* end in the `.um` extension. You may not use the names `midmark.um` or `selfcheck.um`.

Unit tests may also include files that specify input and expected output. For example, if your UM binary is called `hello.um`, your unit test may include the following additional files:

- File `hello.0` contains the input required for the unit test `hello.um`. If there is no file `hello.0`, we will run your unit test with an empty file on standard input.
- File `hello.1` contains output that the unit test `hello.um` is expected to write, assuming that the UM under test is correct. If there is no file `hello.1`, we will assume that the test program `hello.um` is not supposed to write anything.

Each of your unit tests will be evaluated as follows:

1. We will run the test using a correct Universal Machine, using the input you provide. For example, we will run

   ```
   good-um hello.um < hello.0
   ```

   or if you do not provide a `hello.0`,

   ```
   good-um hello.um < /dev/null
   ```

We will expect this command to produce output identical to `hello.1`, or if you do not provide a `hello.1`, we will expect it to produce no output.

2. If your test produces unexpected output, or if it causes the reference machine to fail (including any of the failures listed in Section **??**), your test is *invalid*. Invalid tests lower your grade.

6.2.2 Implementation

Your implementation is expected to include the following:

- All `.c` and `.h` files you have written.
- A `Makefile`, which when run with `make`, compiles all your source code and produces a `um` binary.
- A `UMTESTS` file which lists each of your UM unit tests, one test per line. Specifically, each line must be the name of a file with the `.um` extension. The name must be properly capitalized to match the name of the file you are submitting. Do not list the names of data files with extensions like `.0` or `.1`.
- 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
  - Mentions each UM unit test (from `UMTESTS`) by name, explaining *what* each one tests and *how*
  - 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*