# COMP40 Assignment: A Universal Virtual Machine

## Contents

# 1 Purpose and overview

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 also have an opportunity to demonstrate your ability to create a design with a clean modular structure, to document that design, to choose suitable data structures and implementation techniques for that program, and to document those choices.

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 Getting Started

You will be building an executable named `um`. This Linux program takes a single argument which is the pathname for a file. That file, typically with a name like `some_program.um`, contains the machine instructions that your emulator is to execute. *When a UM program is stored in a file, words 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. As described below, `stdin` and `stdout` are used for the implementations of the UM `Input` and `Output` instructions respectively. Thus, a typical invocation of your emulator might look like:

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

Your program is to emulate a UM, so 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 remaining sections of these instructions. They include additional specifications that apply to your emulator, e.g. how fast it must execute UM programs. You will also find instructions for writing your design documents, for building unit tests, and for submitting your designs, unit tests and working UM emulator. You will also find hints relating to the trickier aspects of implementing a 32 bit UM on a 64 bit AMD 64 Linux machine.

# 3 Specification of the Universal Machine

## 3.1 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.*

### 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 $i$ is designated $\$\mathtt{m}[i]$. The 0 segment is designated $\$\mathtt{m}[0]$.
- A word at offset $n$ within segment $i$ is designated $\$\mathtt{m}[i][n]$. You might refer to $i$ 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. 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

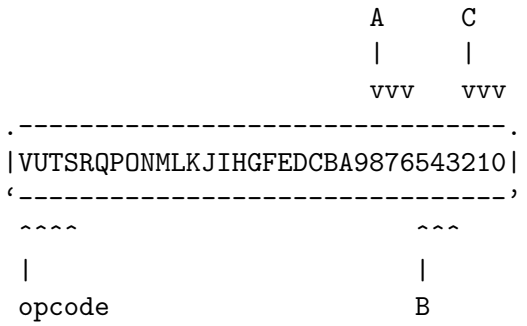
The Universal Machine recognizes 14 instructions. The instruction is coded by the four most significant bits of the instruction word. These bits are called the *opcode*.

| *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 | $\$\mathtt{r}[A] := \neg(\$\mathtt{r}[B] \wedge \$\mathtt{r}[C])$ |
| Some instructions ignore one or more of the register numbers $A$, $B$, and $C$. | | |
| 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 displayed on 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 3.5.2. |

Figure 1: Semantics of UM instructions

### 3.5.1 Three-register instructions

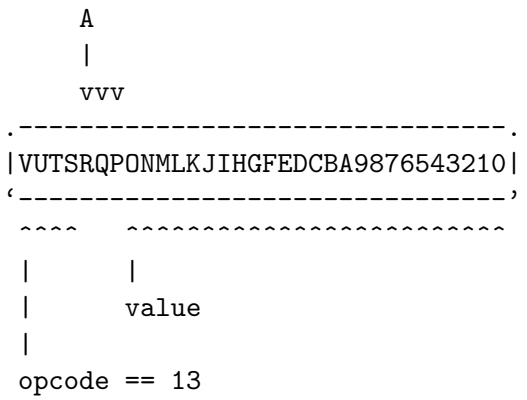
Most instructions operate on three registers. The registers are identified by number; we'll call the numbers $A$, $B$, and $C$. Each number coded as a three-bit unsigned integer embedded in the instruction word. The register $C$ is coded by the three least significant bits, the register $B$ by the three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




Semantics are given in Figure 1.

### 3.5.2 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 $\$\mathtt{r}[A]$.




## 3.6 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 $\$\mathtt{m}[0]$, 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. Go wild!

### 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 3.6, 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

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.

Of course there is at this point in COMP 40 the expectation that you will demonstrate what you've learned about building a clean, well-organized program:

- We expect to see you make good choices for modularizing your design
- We expect to see clean, well-documented interfaces for each module. We expect each module to "perform one function well" and/or to "hide secret(s)"

- We expect to see well named functions, variables and constants
- We expect that, having completed the "arith-challenge" assignment you will look for opportunities to leverage "single points" of truth. (There will not be a "UM challenge", but we will as always grade your code for structure and organization.)

And finally there are some useful things to know:

- In the C programming language running on modern hardware, 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 $2^k$.
- In the C programming language running on AMD64 hardware, division of signed and unsigned integer types rounds toward zero.[1]
- 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.

### 4.1 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.) Hanson's CII library is there if you need it.

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 (Umsegment_Id)(uintptr_t)Seq_remlo(ids);
```

might be useful. (The above assumes `typedef uint32_t Umsegment_Id`. 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.)

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.

[1] For signed types, rounding toward zero is usually not what you want. Rounding toward minus infinity would be much more useful. Alas, we are stuck with this legacy feature.

## 4.2 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 a well-optimized UM should be able to complete in about 1 second on our lab machines; a UM designed without regard to performance might take 20 minutes on the same benchmark. This is an admittedly informal performance goal but if you're taking tens of seconds that's not good.

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`.)
- 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.

How to balance efficiency considerations with writing clean modular code? Answer: Your main goal should be to write modular code that demonstrates your mastery of the principles of abstraction, secret-keeping, interface design, etc. that you have studied throughout COMP 40. You also MUST meet the minimum performance requirements described in these instructions, I.e. to run the benchmark in a handfull of seconds vs. many tens of seconds. It is quite possible to achieve the performance and structure goals simultaneously if you are careful.

## 4.3 Controlling use of CPU and memory

When we test your UM, we will give it only 1000 MB of memory, and we will limit its CPU time as well. Because you can easily overlook how much time and space your UM needs, we provide the commands `mem-limited` and `cpu-limited`, which ensure that your UM runs within specified limits. For example, to run with 500 MB of memory for at most 10 seconds, you can run

```
mem-limited 500m cpu-limited 10 ./um midmark.um
```

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

```
catch-signal mem-limited 500m cpu-limited 10 ./um midmark.um
```

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

## 4.4 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. 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.

- As always, things like wrong exit codes, incorrect use of Checked or Unchecked Runtime Errors, inappropriate output to stderr, etc. tend to cost students lots of credit on otherwise impressive submissions. REREAD ALL THE INSTRUCTIONS AND CHECK ALL THE DETAILS BEFORE YOU SUBMIT!

# 5 What we expect of you

## 5.1 Your design and its documentation

The documentation of your design should include the following sections:

- The high points of a design checklist for the full Universal Machine[2], emphasizing the architecture and test plan (items 11, 12, and 13).

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

- The high points of a design checklist for UM segments[3], emphasizing the representation of segments and its invariants.

Remember we don't want your *complete* design checklist—show us only the interesting parts.

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

- Excellent design documentation will give explicit, compilable interfaces and *compilable unit tests* for whatever mechanism you decide will best serve to implement memory segments represented by 32-bit identifiers. Unlike the unit tests you will build for the UM itself, these will be `.c` (and maybe `.h` files) that invoke your segment-management methods.

- 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.

## 5.2 Universal Machine unit tests

Your design submission should include unit tests for the Universal Machine segment abstraction (as described above), but not for the Universal Machine instruction set. Remember, your tests for the segment abstraction are likely to be in the form of `.c` and as necessary `.h` files. Your final submission should also include *unit tests for the instruction set*, represented as described below. In your README file, include documentation of a sequence of unit tests that *in toto* cover all of the Universal Machine instructions. Each unit test may rely on 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.

[2] See URL `http://www.cs.tufts.edu/comp/40/handouts/design-pgm.pdf`.
[3] See URL `http://www.cs.tufts.edu/comp/40/handouts/design-adt.pdf`.

And so on.

Every unit test *must* include a compiled UM binary whose name ends in `.um`.[4] You will write code to create these binaries. No unit test may cause any of the failures listed in Section 3.6.

Unit tests may also include additional files. If your UM binary is called `hello.um`, your unit test may include these 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, your test is *invalid*. Invalid tests lower your grade and play no further role in the process.

3. If your test produces the expected output with no failures, it is *valid*. Each valid test is run against all the UMs submitted by all the other pairs working on the problem. It is also run against a selection of faulty UMs that we create.

The more faulty UMs your tests detect, the higher your grade.

## 5.3 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 lab machines; 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.*

[4] You may not use the names `midmark.um` or `selfcheck.um`.

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. Remember, *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.

## 5.4 What to submit

### 5.4.1 Design

Using the script `submit40-um-design`, please submit

- A `DESIGN` or `design.pdf` file describing your design. Because plain text is much easier for us to read, please use PDF only if you have diagrams or other information that is not easily rendered in plain text.
- Source code for the segment interface and its unit tests as a collection of `.h` and `.c` files. Your segment unit tests should include all relevant `.h` files and should compile correctly, but they need not run. You need not submit a `Makefile`. We will compile your code with a script that gives you access to the usual Hanson ADTs and to `.h` files from `/comp/40/include`.

### 5.4.2 Implementation

Using the script `submit40-um`, please submit

- All `.c` and `.h` files you have written.
- A script called `compile` that compiles all your `.c` files into `.o` files and then links 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*

On a 32-bit machine, most experienced C programmers can understand the Universal Machine specification and build an implementation in a total of two hours. 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.

Norman Ramsey's implementation is about 200 lines of C code; almost half 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 well under 100 lines of code.

# 6 What we provide for you

We provide the following useful items:

- At `http://www.cs.tufts.edu/comp/40/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.
- In `/comp/40/include` and `/comp/40/lib64` 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 gives a string representation of a Universal Machine instruction. *You must free the string* returned by `Um_disassemble`, or you will have memory leaks.

  You may find it useful to call `Um_disassemble` from DDD.
- Program `/comp/40/bin/umdump` will dump the contents of a Universal Machine binary, as in

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

  It is the closest counterpart I have to `objdump`.
- There is a working `libbitpack.a` in `/comp/40/lib64`; you can link against it using `-lbitpack`.
- In Friday's lab you will see some examples of C code that you can use to get ideas about unit-testing your Universal Machine.