COMP40 Assignment: A Universal Virtual Machine

Assignment due Thursday, November 19 at 11:59 PM. Design (including unit tests) due Tuesday, November 17 at 11:59 PM.

Professor Ramsey will not be available for office hours on Wednesday, November 18; he will be available Tuesday November 17 instead.

Contents

1 Purpose and overview

2 Specification of the Universal Machine
   2.1 Machine State
   2.2 Notation
   2.3 Initial state
   2.4 Execution cycle
   2.5 Instructions’ coding and semantics
      2.5.1 Three-register instructions
      2.5.2 One other instruction
   2.6 Failure modes

3 Advice on the implementation
   3.1 Emulating a 32-bit machine: Simulating 32-bit array identifiers
   3.2 Efficient abstractions

4 What we expect of you
   4.1 Your design and its documentation
   4.2 Implementation
   4.3 What to submit
      4.3.1 Design
      4.3.2 Implementation

5 What we provide for you
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.

2 Specification of the Universal Machine

2.1 Machine State

The UM has these components:

- Eight general-purpose registers holding one word each
- An ever-changing collection of arrays of words, each referred to by a distinct 32-bit identifier
- A 1x1 character resolution console capable of displaying ASCII characters and performing input and output of unsigned 8-bit characters
- A 32-bit program counter

One distinguished array is referred to by the 32-bit identifier 0 and stores the program. This array is called the ‘0’ array.

2.2 Notation

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

- Registers are designated $r[0]$ through $r[7]$
- The array identified by the 32-bit number $i$ is designated $a[i]$. The ‘0’ array is designated $a[0]$.
- A word at offset $n$ within array $i$ is designated element($a[i], n$).

2.3 Initial state

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

- The ‘0’ array $a[0]$ contains the words of the program.
- An array may be active or inactive. Initially, $a[0]$ is active and all other arrays are inactive.
- All registers are zero.
- The program counter points element($a[0], 0$), i.e., the first word in the ‘0’ array.

2.4 Execution cycle

At each time step, an instruction is retrieved from the word pointed to by the program counter. The program counter is advanced to the next word, if any, and the instruction is then executed.
<table>
<thead>
<tr>
<th>Number</th>
<th>Operator</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Conditional Move</td>
<td>if $r[C] \neq 0$ then $r[A] := r[B]$</td>
</tr>
<tr>
<td>1</td>
<td>Array Index</td>
<td>$r[A] := \text{element}(a[r[B]], r[C])$</td>
</tr>
<tr>
<td>2</td>
<td>Array Update</td>
<td>$\text{element}(a[r[A]], r[B]) := r[C]$</td>
</tr>
<tr>
<td>3</td>
<td>Addition</td>
<td>$r[A] := (r[B] + r[C]) \mod 2^{32}$</td>
</tr>
<tr>
<td>4</td>
<td>Multiplication</td>
<td>$r[A] := (r[B] \times r[C]) \mod 2^{32}$</td>
</tr>
<tr>
<td>5</td>
<td>Division</td>
<td>$r[A] := \lfloor r[B] \div r[C] \rfloor$</td>
</tr>
<tr>
<td>6</td>
<td>Bitwise NAND</td>
<td>$r[A] := \lnot (r[B] \land r[C])$</td>
</tr>
</tbody>
</table>

Some instructions ignore one or more of the register numbers $A$, $B$, and $C$.

- **7** Halt: Computation stops.
- **8** Activate Array: A new array is created with a number of words equal to the value in $r[C]$. Each word in the new array is initialized to 0. A bit pattern that is not all zeroes and that does not identify any currently active array is placed in $r[B]$. Array $a[r[B]]$ becomes active.
- **9** Inactivate Array: The array $a[r[C]]$ becomes inactive. Future allocations may reuse the identifier $r[C]$.
- **10** Output: The value in $r[C]$ is displayed on the console immediately. Only values from 0 to 255 are allowed.
- **11** Input: The universal machine waits for input on the console. 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 word in which every bit is 1.
- **12** Load Program: Array $a[r[B]]$ is duplicated, and the duplicate replaces $a[0]$, which is abandoned. The program counter is set to point to $\text{element}(a[0], r[C])$. If $r[B] = 0$, the load-program operation is expected to be extremely quick.
- **13** Load Value: See semantics for “other instruction” in Section 2.5.2.

Figure 1: Semantics of UM instructions
2.5 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.

2.5.1 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:

\[
\begin{array}{ccc}
A & C \\
\mid & \mid \\
nvvv & nnv
\end{array}
\]

\[\text{-----------------------------------} \]

\[
|VUTSRQPONMLKJIHGFEDCBA9876543210| \\
\text{-----------------------------------}
\]

Semantics are given in Figure 1.

2.5.2 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 opcode describe a single register $A$. The remaining 25 bits indicate a value, which is loaded into $r[A]$.

\[
\begin{array}{ccc}
A \\
\mid \\
nvv
\end{array}
\]

\[\text{-----------------------------------} \]

\[
|VUTSRQPONMLKJIHGFEDCBA9876543210| \\
\text{-----------------------------------}
\]

Semantics are given in Figure 1.

2.6 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 $a[0]$, 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.

• Array index or array update of an inactive array may fail.

• Array index or array update of a word outside the bounds of an array may fail.

• If an instruction inactivates $a[0]$, or if it inactivates an array that is not active, the machine may fail.

• If an instruction divides by zero, the machine may fail.

• If an instruction loads a program from an array that is not active, 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.

3 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 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.\footnote{For signed types, rounding toward zero is a crime. Rounding toward minus infinity would be much more useful. Alas, we are stuck with a legacy feature.}
3.1 Emulating a 32-bit machine: Simulating 32-bit array identifiers

On a 32-bit machine, you could simply use a 32-bit pointer as an array identifier and have `malloc` do your heavy lifting. On the 64-bit machine, you will need an abstraction that maps 32-bit array identifiers to actual arrays. (Any representation of arrays 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 inactivated. One way is to store them in one of Hanson’s sequences (`Seq_T`). You can cast an `unsigned long` to a `void *`, so statements like

```c
Seq_addlo(ids, (void *) (unsigned long) id);
```
```
return (Umarray_Id)(unsigned long)Seq_remlo(ids);
```

might be useful. (I have written `typedef uint32_t Umarray_Id`.)

3.2 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`.)

- 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 What we expect of you

4.1 Your design and its documentation

The documentation of your design should include

- The high points of a design checklist for UM arrays\(^2\), emphasizing the representation of arrays and its invariants.

- The high points of a design checklist for the full Universal Machine\(^3\), emphasizing the architecture and test plan (items 11, 12, and 13).

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


\(^3\)See URL http://www.cs.tufts.edu/comp/40/handouts/design-pgm.pdf.
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 arrays represented by 32-bit identifiers.
- 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.
- Excellent design documentation will describe 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.

And so on.

4.2 Implementation

We expect you to write a complete and correct implementation of the Universal Machine. Moreover, we ex-pect it to be efficient enough to execute 50 million UM instructions in less than 100 CPU seconds.

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 “console” should be implemented using standard input and standard output.

4.3 What to submit

4.3.1 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 array interface and its unit tests as a collection of .h and .c files. Your unit tests should include all relevant .h files and should compile correctly, but they need not run.
4.3.2 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 three executable binaries:
  - The `um` binary takes one argument, which is the name of a file containing a UM program, and it runs that program to completion.
  - The `um-test` binary runs your short unit tests. If you prefer, you may make `um-test` an executable shell script. A substantial portion of your grade will be based on our evaluation of whether you have tested your UM thoroughly.
  - The `um-test-long` binary runs at least one long unit test, which executes at least 50 million UM instructions.
- 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
  - 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 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.

My 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.

5 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.
• 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 -lci. 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.