COMP 40 Laboratory: Test code for a Universal Machine

November 4, 2011

1 Introduction

In this lab, you'll practice creating simple test programs to be run on a Universal Machine. You'll use such programs for unit testing. To *create* a unit test, we build it up in an *instruction stream*, one instruction at a time.

This lab will get you started on unit testing—do not expect to finish your unit tests in 75 minutes. The lab gives you *guidelines*, not *directions*. You should get the code from this handout by running

```
git clone /comp/40/git/um
```

We expect you to fill in missing pieces as needed—this may require that you think about and interpret the material in the handout.

We expect that you do not yet have a working Universal Machine. We therefore recommend that you create test programs, write them to disk, and examine them using `umdump`:

1. Use the ideas in Section 3 below to create a `Seq_T` containing UM instructions.
2. Write a function that takes a `Seq_T` and writes the instructions to a file, in big-endian byte order, as specified in Section 4.3 of the homework handout.
3. To see if you have coded and backpatched the instructions correctly, run `umdump` on the resulting file. (To put `umdump` on your `PATH`, you will first need to run `use comp40`.)

   You can learn the ASCII character codes by running `man ascii`.

If you *do* have a Universal Machine, you then start testing it.

The directions for the lab are therefore as follows:

1. Read through the entire lab before starting work.
2. Decide whether you are going to try to run each unit test or if you prefer simply to examine it using `umdump`.
3. For each unit test in the lab, build the test and write it to a `.um` file. To hold the output expected from the test, create a suitable `.1` file. Finally, if the test requires input, create a suitable `.0` file.

In 75 minutes, we think you may be able to complete the first two unit tests and the infrastructure through Section 6. There are more tests in Section 7 than you can implement during the lab.

2 Creating individual instructions

Any unit test for the Universal Machine will require a program written in Universal Machine binary code. And any such program will consist of a sequence of 32-bit words, each containing an instruction or data. Your first step in testing is therefore to be able to make instructions.

We recommend that you define one instruction-making function for each instruction format. You'll need only two functions:

```
typedef uint32_t Um_instruction;
Um_instruction three_register(Um_opcode op, int ra, int rb, int rc);
Um_instruction loadval(unsigned ra, unsigned val);
```

For the opcode type `Um_opcode`, choose a representation that seems good to you.

Not every instruction requires all three registers. For example, the Halt instruction behaves the same no matter what registers appear in fields A, B, and C. We recommend you create a macro or `static inline` function for every instruction, e.g.,

```
static inline Um_instruction halt(void) {
  return three_register(HALT, 0, 0, 0);
}
```

To name registers within an instruction, we'll use a convenient enumeration type:

```
enum regs { r0 = 0, r1, r2, r3, r4, r5, r6, r7 };
```

3 Accumulating instructions in streams

To create a unit test, you will accumulate its instructions one at a time in an *instruction stream*. (You will be able to use the same functionality in the Universal Machine itself, to load a program from a disk file.)

To accumulate instructions in a stream requires three interesting steps:

- Append an instruction to the stream.
- Note the current position of the instruction stream. The position can be used as a kind of *label*.
- *Backpatch* an earlier instruction by putting in a label value.

To represent an instruction stream, we'll use Hanson's `Seq_T`. Each element will contain a `void *` that has been cast from `uintptr_t`. You'll need to define a way to write such a `Seq_T` to disk using the UM binary format:

```
extern Um_write_sequence(FILE *output, Seq_T stream);
```

The most common operation on instruction streams is to append an instruction to the stream, which is called *emitting* the instruction:

```
static inline void emit(Seq_T stream, Um_instruction inst) {
  assert(sizeof(inst) <= sizeof(uintptr_t));
  Seq_addhi(stream, (void *)(uintptr_t)inst);
}
```

The type `uintptr_t` is an unsigned integer type that is large enough to hold a pointer value.

We can also get and put instructions at individual locations in the stream:

```
static inline Um_instruction get_inst(Seq_T stream, int i) {
  assert(sizeof(Um_instruction) <= sizeof(uintptr_t));
  return (Um_instruction)(uintptr_t)(Seq_get(stream, i));
}

static inline void put_inst(Seq_T stream, int i, Um_instruction inst) {
  assert(sizeof(inst) <= sizeof(uintptr_t));
  Seq_put(stream, i, (void *)(uintptr_t) inst);
}
```

4 First unit test: Halt

Here I'm going to create a sequence of instructions that begins with a Halt. But in case the Halt doesn't work, I will have the Universal machine write `"Bad!\n"` before failing. I'm assuming you have defined a function `output` which creates an Output instruction.

```
void emit_halt_test(Seq_T stream) {
  emit(stream, halt());
  emit(stream, loadval(r1, 'B'));
  emit(stream, output(r1));
  emit(stream, loadval(r1, 'a'));
  emit(stream, output(r1));
  emit(stream, loadval(r1, 'd'));
  emit(stream, output(r1));
  emit(stream, loadval(r1, '!'));
  emit(stream, output(r1));
  emit(stream, loadval(r1, '\n'));
  emit(stream, output(r1));
}
```

You may prefer shorter names for the instructions, so your C code looks more like assembly language.

5 Second unit test: goto

This more ambitious test shows two advanced ideas:

- Control flow with labels and backpatching

- Output of whole strings (with the aid of a temporary register)

Here's the assembly code we would like to write:

```
  r7 := L
  goto r7 in program m[r0]
  output "GOTO failed.\n" using r1
  halt
L:
  output "GOTO passed.\n" using r1
  halt
```

This code is a simple test of Load Program with the same program as currently in use (because register `r0` contains zero), but as simple as it is, it presents two difficulties:

1. At the time we create the initial assignment, we don't know the value of label `L`. We'll solve this problem by *backpatching*.

   - If we have a Load Value instruction is trying to load the value of label `L`, we remember its location in the variable `patch_L`.
   - When we reach the *definition* of `L`, we add the value of `L` to the instruction word at location `patch_L`

   Here is a useful auxiliary function:

   ```
   static void add_label(Seq_T stream, int location_to_patch, int label_value) {
     Um_instruction inst = get_inst(stream, location_to_patch);
     unsigned k = Bitpack_getu(inst, 25, 0);
     inst       = Bitpack_newu(inst, 25, 0, label_value+k);
     put_inst(stream, location_to_patch, inst);
   }
   ```

   This code updates an instruction by adding the label value into the immediate field of the Load Value instruction.

2. We want to write out string literals with the aid of an auxiliary register. I won't give you code for that, but I will give you a signature:

   ```
   static void emit_out_string(Seq_T stream, const char *s, int aux_reg);
   ```

   This function should loop through all the characters in string `s`. For each character c, it should emit two instructions:

   - Load Value of c into auxiliary register `aux_reg`
   - Output the value in `aux_reg`

Here is a unit test that uses these techniques. It assumes that register 0 always contains zero.

```
  void emit_goto_test(Seq_T stream) {
    int patch_L = Seq_length(stream);
    emit(stream, loadval(r7, 0));  // will be patched to 'r7 := L'
    emit(stream, loadprogram(r0, r7)); // should goto label L
    emit_out_string(stream, "GOTO failed.\n", r1);
    emit(stream, halt());
    add_label(stream, patch_L, Seq_length(stream)); // define 'L' to be here
    emit_out_string(stream, "GOTO passed.\n", r1);
    emit(stream, halt());
  }
```

6 Infrastructure for running unit tests

Here is a main program which can run unit tests. It will run the unit tests that are named on the command line, or if no test is named, it will run all of them. You should be able to figure out how to link this main program with your other code, how to add your own tests, and how to run everything.

(In `struct test_info`, the structure field declared as

```
void (*emit_test)(Seq_T stream);
```

is a pointer to a function that, when called, appends a sequence of instructions to its argument.)

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "assert.h"
#include "fmt.h"
#include "seq.h"

extern void Um_write_sequence(FILE *output, Seq_T asm);

extern void emit_halt_test(Seq_T asm);
extern void emit_goto_test(Seq_T asm);

static struct test_info {
  const char *name;
  const char *test_input;          // NULL means no input needed
  const char *expected_output;
  void (*emit_test)(Seq_T stream); // writes instructions into sequence
} tests[] = {
  { "halt", NULL, "",               emit_halt_test },
  { "goto", NULL, "GOTO passed.\n", emit_goto_test },
};

#define NTESTS (sizeof(tests)/sizeof(tests[0]))

static FILE *open_and_free_pathname(char *path);
// open file 'path' for writing, then free the pathname;
// if anything fails, checked runtime error

static void write_or_remove_file(char *path, const char *contents);
// if contents is NULL or empty, remove the given 'path',
// otherwise write 'contents' into 'path'.  Either way, free 'path'.

static void write_test_files(struct test_info *test) {
  FILE *binary = open_and_free_pathname(Fmt_string("%s.um", test->name));
  Seq_T asm = Seq_new(0);
  test->emit_test(asm);
  Um_write_sequence(binary, asm);
  Seq_free(&asm);
  fclose(binary);

  write_or_remove_file(Fmt_string("%s.0", test->name), test->test_input);
  write_or_remove_file(Fmt_string("%s.1", test->name), test->expected_output);
}
```

```
int main (int argc, char *argv[]) {
  bool failed = false;
  if (argc == 1)
    for (unsigned i = 0; i < NTESTS; i++) {
      printf("***** Writing test '%s'.\n", tests[i].name);
      write_test_files(&tests[i]);
    }
  else
    for (int j = 1; j < argc; j++) {
      bool tested = false;
      for (unsigned i = 0; i < NTESTS; i++)
        if (!strcmp(tests[i].name, argv[j])) {
          tested = true;
          write_test_files(&tests[i]);
        }
      if (!tested) {
        failed = true;
        fprintf(stderr, "***** No test named %s *****\n", argv[j]);
      }
    }
  return failed; // failed nonzero == exit nonzero == failure
}


static void write_or_remove_file(char *path, const char *contents) {
  if (contents == NULL || *contents == '\0') {
    remove(path);
  } else {
    FILE *input = fopen(path, "wb");
    assert(input);
    fputs(contents, input);
    fclose(input);
  }
  free(path);
}

static FILE *open_and_free_pathname(char *path) {
  FILE *fp = fopen(path, "wb");
  assert(fp);
  free(path);
  return fp;
}
```

7 Additional unit tests

Unit tests are a primary part of this assignment. Here are some suggestions for other unit tests:

- Make sure Conditional Move does the right thing given a register whose contents are zero. The assembly code might look like this:

```
  r7 := L1
  r6 := L2
  if (r0 != 0) r7 := r6  // should leave r7 unchanged
  goto r7 in program m[r0];
L2:
  output "Conditional Move on zero register failed.\n" using r5
  halt
L1:
  output "Conditional Move on zero register passed.\n" using r5
  halt
```

You'll need to write C code that backpatches *two* labels: `L1` and `L2`.

- Make sure Conditional Move does the right thing given a register whose contents are *not* zero.
- Read a character and print the character.
- Load value 48, add a 1-digit number to it, and print the result, which should print as the digit.
- Read one character from the input, which should be a digit. Use NAND to extract the least significant 4 bits. Add this number to 48, and print the result, which should be the original digit.
- Test all the arithmetic instructions. You can print any positive, single-digit result if you add 48. You can tell a result fits in one digit if when it is divided by 10, you get zero.
- Insert data into the instruction stream, branch around it, and test the Segmented Load and Segmented Store instructions on segment 0.
- Test Map Segment and Unmap Segment, most likely in a loop.
- Since your performance target is to execute 50 million instructions in 100 seconds, write a loop that you expect to execute 500,000 instructions and see if it finishes in one second. *Don't forget to compile with `-O1` and `-O2`; use whichever gives better results.*

8 System tests

At `http://www.cs.tufts.edu/comp/40/um`, you will find some test binaries. If you can get good enough performance, they are quite interesting:

`midmark.um`	medium benchmark; over 80 million instructions
`advent.umz`	Adventure game; 700 million instructions
`codex.umz`	UMIX operating system; 1.6 billion instructions
`sandmark.umz`	large benchmark; over 2 billion instructions