Introduction

In this lab, you’ll practice creating simple test programs to be run on a Universal Machine. You’ll be able to 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 gives you guidelines, not directions. Get code from this handout by git clone /comp/40/git/um. We expect you to fill in missing pieces using your intelligence as guided by experience. We are shorthanded today, but please also ask the course staff.

Creating individual instructions

To add instructions to a unit test, you first have to be able to make instructions. We recommend that you define instruction-making functions for each instruction format. There are just two:

```c
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);
```

To represent the opcode type `Um_opcode`, choose some 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.,

```c
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:

```c
enum regs { r0 = 0, r1, r2, r3, r4, r5, r6, r7 };
```
Representing a unit test

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.

We’ll use Hanson’s Seq_T as a convenient representation for instruction streams—it plays a role like that of an assembly-language program. Each element will contain a void * that has been cast from uintptr_t. We’ll assume that you have a way of running a UM program that is represented as a Seq_T:

```c
extern Um_run_sequence(Seq_T asm);
```

This page does not endorse (or fail to endorse) Seq_T as a representation suitable for use within your UM; Um_run_sequence might convert its argument to an internal representation before running the code.

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

```c
static inline void emit(Seq_T asm, Um_instruction inst) {
    Seq_addhi(asm, (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:

```c
static inline Um_instruction get_inst(Seq_T asm, int i) {
    return (Um_instruction)(uintptr_t)(Seq_get(asm, i));
}

static inline void put_inst(Seq_T asm, int i, Um_instruction inst) {
    Seq_put(asm, i, (void *)(uintptr_t) inst);
}
```

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 asm) {
    emit(asm, halt());
    emit(asm, loadval(r1, 'B'));
    emit(asm, output(r1));
    emit(asm, loadval(r1, 'a'));
    emit(asm, output(r1));
    emit(asm, loadval(r1, 'd'));
    emit(asm, output(r1));
    emit(asm, loadval(r1, '!'));
    emit(asm, output(r1));
    emit(asm, loadval(r1, '
'));
    emit(asm, output(r1));
}

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

**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:

```assembly
r7 := L
goto r7 in program a[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:

```c
static void add_label(Seq_T asm, int location_to_patch, int label_value) {
    Un_instruction inst = get_inst(asm, location_to_patch);
    unsigned k = Bitpack_getu(inst, 25, 0);
    inst = Bitpack_newu(inst, 25, 0, label_value+k);
    put_inst(asm, 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:

```c
static void emit_out_string(Seq_T asm, 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.

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

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.

```c
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include "seq.h"
```
extern void Um_run_sequence(Seq_T asm);

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

static struct {
    const char *name;
    void (*emit)(Seq_T asm);
} tests[] = {
    { "halt", emit_halt_test },
    { "goto", emit_goto_test },
};

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

static void run(void (*emit)(Seq_T asm)) {
    Seq_T asm = Seq_new(0);
    emit(asm);
    Um_run_sequence(asm);
    Seq_free(&asm);
}

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

The structure field declared as

    void (*emit)(Seq_T asm);

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

  ```asm
  r7 := L1
  r6 := L2
  if (r0 != 0) r7 := r6 // should leave r7 unchanged
  goto r7 in program a[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 Array Index and Array Update instructions on array 0.

- Test Activate Array and Inactivate Array, 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.

**System tests**

At [http://www.cs.tufts.edu/comp/40/um](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:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>midmark.um</td>
<td>medium benchmark; over 80 million instructions</td>
</tr>
<tr>
<td>advent.umz</td>
<td>Adventure game; 700 million instructions</td>
</tr>
<tr>
<td>codex.umz</td>
<td>UMiX operating system; 1.6 billion instructions</td>
</tr>
<tr>
<td>sandmark.umz</td>
<td>large benchmark; over 2 billion instructions</td>
</tr>
</tbody>
</table>