Design Doc for UM

Ian Cross and Emmett Moore

1.*What problem are we trying to solve?*
We are writing a Universal Turing Machine that can take in any binary program and run it. It will include the following:

- 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

2. *What example inputs will help illuminate the problem?*
We will be using executables (in binary form) that test individual pieces of the the UM. Initially, we will be designing test cases for our segment interface and then later we will test each individual operation (14 UM opcodes) using small programs that test each individual case. For example, just like Norman's code in the lab, we will have a program that just halts and prints something if it doesn't.

3. *What example outputs go with those example inputs?*
For most of the test cases, we can use DDD to check individual registers and memory addresses. This way, we can be sure that each instruction is performing the way we expect. Otherwise, some the instructions use I/O and we can simply see if something is printed.

4. *Into what steps or subproblems can you break down the problem?*

1. The first problem is figuring out how to manage memory and how to store segments with 32 bit identifiers.
    a. This includes keeping track of labels and where you are in the control flow.
2. The second problem is inputing a stream of 64 bit words and converting them into 32 bit instructions that makes up a program in binary. This input stream will be loaded into segment 0.
3. We then have to come up with representations for each instruction in the set of 14 opcodes.

    The above subproblems lend themselves to the following order of operations in our UM:
    a. initialize data (registers, segments)
    b. load the program into segment 0.
    c. iterate through each instruction

d. free all allocated memory

We skipped questions 5-8 because the spec said to only include the high points of these design documents, and we found that these questions are answered elsewhere in our design doc.

9. What invariant properties should hold during the solution of the problem?
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 (if valid) is then executed.

11. What are the major components of your program, and what are their interfaces?

The major components of our program consist 3 interfaces and 4 major aspects:

1. main handles initializing memory and storing the instruction stream, along with iterating through the instruction stream
2. UnivM interface takes in an instruction and then calls the correct functions based on the opcode.
3. segMem interface will handle everything to do with our representation of memory segments. This will be accomplished by using a hanson sequence of arrays of uint32_ts.
4. lastly, we will be using bitpack.h to manipulate the instructions and parse opcode, register, and value information from them.

To see the general flow of the program and how these interfaces interact, see #12. To see the actual interfaces in C, see below.

12. How do the components in your program interact? That is, what is the architecture of your program?

Our main() will deal with 3 layers of abstraction. main() itself will only handle initializing the machine and then loops through the instructions (segment 0). This then sends the instruction to our UnivM interface which handles all of the instructions. For 10 of these instructions, the instruction handler function in the UnivM interface will call static functions that carry out the specified function. The other 4 instructions deal with managing memory segments, which we believe warrants its own interface segMem.h. This interface has 4 functions and no data--it expects that the instruction is already analyzed, and it takes in the memory sequence and the values stored in the registers. It doesn't take in the registers themselves, but rather the values stored in them to simplify the interface. Therefore, much of the heavy lifting of this interface is done in univm.h as opposed to in segMem.h. We believe that this is important, as it makes testing our memory managment much easier.

univm.h will use bitpack.h to unpack the instructions (i.e. determine the opcode and register indices). It will then call one of the 14 possible operations. The four instructions that deal

with memory (map, unmap, seg load and seg store) will also be directly called in our switch statement using segMem. The other 10 will have static functions called in the switch statement. Once these functions are run, they will put values back into the registers specified by the instruction. Again, most of the heavy lifting is done by the univm.h.

13. What test cases will you use to convince yourself that your program works?
Initially, we will be testing the segment management aspect of our UnivM interface. It is imperative that this works before we move on to other parts of our program. We are testing four aspects of our segment interface that correspond to 4 of the 14 possible instructions:

1. load
2. store
3. map
4. unmap

Later, we will be using executables. We will use unit testing to see if each individual opcode performs as expected, and we will use DDD to display the values in registers--this will be useful to see that each operation does as we expect it to.

Interface Architecture

`UnivM`

The first interface that we will build is called `UnivM`.
`Invariants:`

- Upon initializing the struct UnivM via UnivM_new(), Segment 0 will contain the program.
- For each instruction that the UnivM_T takes in via UnivM_handleInst(), it will execute a valid opcode (0-13).
- We will always first map segments with identifiers in the freed_mem array. If freed_mem is empty, a new identifier will be created to map that segment.
- If a new segment is ever unmapped, its identifier will exist until the UnivM struct is freed via UnivM_free().

A brief design checklist for the UnivM:

1. *What is the abstract thing you are trying to represent?*
We are trying to represent segments in memory with unique identifier as well as storing the program. Basically, we are representing a computer/machine that can run any program.

2. *What functions will you offer, and what are the contracts of that those functions must meet?*
The functions are specified in the interface below. The instruction handler function will call 1 of

14 functions to carry out the opcode specified in the 4 most significant bits of the instruction.
3. *What examples do you have of what the functions are supposed to do?*
See the spec for this project
4. *What representation will you use, and what invariants will it satisfy?* (This question is especially important to answer *precisely*.)
See invariants below.

We skipped questions 5-7 because the spec said to only include the high points of these design documents, and we found these are answered elsewhere in our design doc.

this is the data that UnivM will hold:

```
struct UnivM{
      Seq_T segments;    //sequence of segments (which are uint32 arrays)
      Seq_T freed_mem; //sequence of identifiers that have been
                        //unmapped and need to be reused
};

#ifndef UNIVM_INCLUDED
#define UNIVM_INCLUDED
#define T UnivM_T
typedef struct T *T;
#include <stdint.h>
#include <stdio.h>
/*
 * creates a new UnivM struct that contains the sequence of segments and the
 * array of freed memory locations for reuse.
 */
extern T UnivM_new(FILE* program);
/*
 * takes in the UnivM struct, the instruction, and the register array and
 * carries out the specified instruction.
 */
extern void UnivM_handleInst(T program, uint32_t inst, int* regs);
/*
 * frees all of the data in the struct.
 */
extern void UnivM_free(T program);
#undef T
#endif
```

segMem

The second interface we will be using is called `segMem`.

Invariant:

Since there is no data in this interface, we are debating whether there is a “class invariant“. What will be true within this interface are the following:

- The segments in the sequence are either mapped, unmapped, loaded, or stored.
- The invariants in UnivM_T will be upheld during execution.

Since this interface holds no memory, we did not fill out the ADT design checklist.

```
/*
 *     segMem.h - Segment Memory interface
 */
#include <stdint.h>

#ifndef SEGMEM_INCLUDED
#define SEGMEM_INCLUDED
#include <seq.h>

extern void segMem_map(Seq_T segments, uint32_t size, uint32_t id);
extern void segMem_unmap(Seq_T segments, uint32_t id);
extern uint32_t segMem_load(Seq_T segments, uint32_t identifier,
                            uint32_t index);
extern void segMem_store(Seq_T segments, uint32_t id,
                         uint32_t index, uint32_t value);

#endif
```

A diagram of the architecture of our program, and how the interfaces interact with one another.