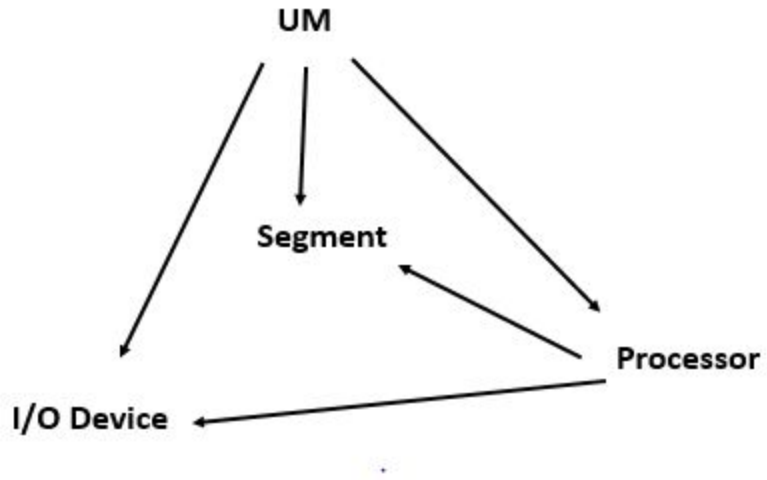
George Pesma and Vivek Ramchandran
Comp 40 UM Machine Design Document
11/14/16

Design:

Overall Architecture

The following graph illustrates the architecture of our UM. Arrows represent direction of access of different abstractions.



There will be a um.c file that contains the main function and three interfaces with their respective implementations.

- The Segment interface will model the memory. Every memory segment will be represented by a Hanson UArray, and every UArray is an element of a Hanson sequence. The segment identifiers of segments that have been unmapped will be stored in a queue, in order to be reused.
- I/O device will read in the .um file and save all the instructions in segment 0, as well as writing the output to a file or standard output. It will also be used along with some of the instructions (as output).
- Processor will contain a function for each of the 14 instructions. Its parameter will be a word. Registers will be declared globally in this file and will be represented by an array of length 8, that contains words. The program counter's only

purpose is to iterate through the different instructions, and can be simply just the counter in a for loop.

- Um will be the file containing the main function. It will use the I/O device to save the program to be run on segment 0, then will go through that segment, word by word and call the processor to execute the instructions.

I/O DEVICE

The I/O (Input/Output) interface (io_devicce.h) performs input and output of 8-bit characters. It is incredibly simple, and should take very little code.
Functions:
Public:
scanbyte() - reads in 1 byte from standard input
printbyte() - prints out 1 byte to standard output

UM

UM holds the main function. It reads in the program instructions from a file using the I/O device and saves them in segment 0. After that, the process abstraction runs the program.

Functions:
main() - calls everything
read_program(file pointer) loop through the file, saving instructions in segment 0
Read_two words(file pointer) - uses scan byte four times and bitpacks them together into a 32-bit word

SEGMENT

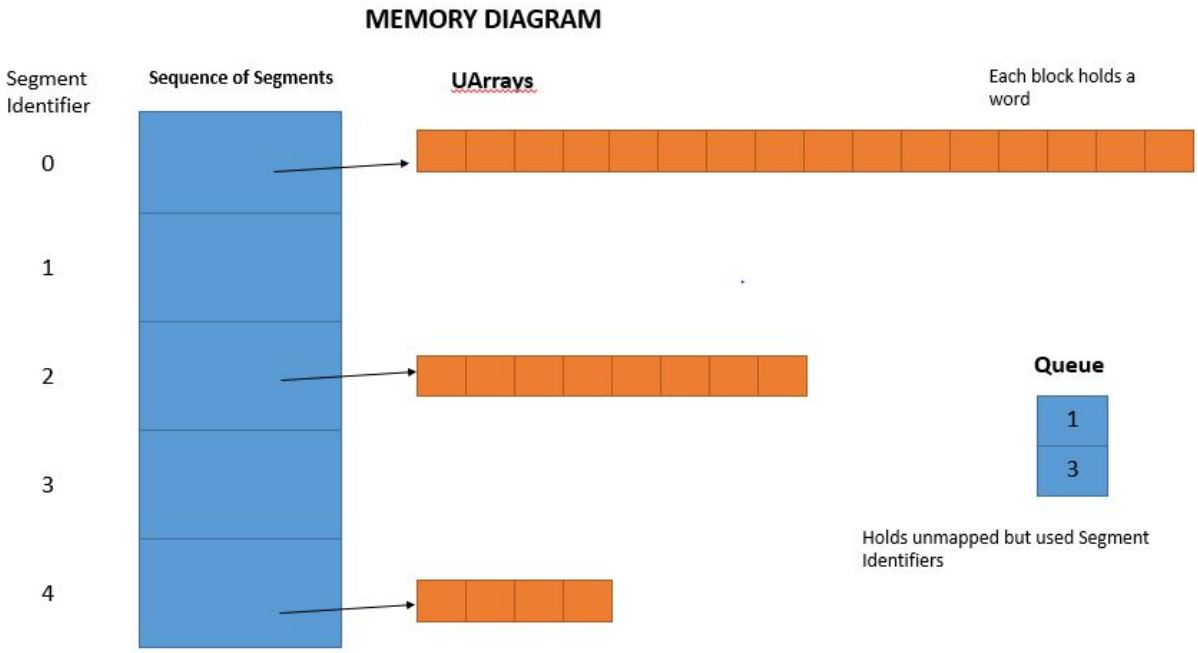
The segment module will handle storing and loading values from memory. Each segment will be represented by a Hanson UArray and every UArray is a different element of a sequence. We will use the indices of the sequence as the 32 bit segment identifiers. When a segment is deleted, the UArray is freed and that element in the sequence will subsequently point to null. We will create a queue (using a Hanson sequence) that will hold segment identifier values of unmapped segments - that is, when we delete a segment and free the memory, that index will be saved in the queue. When more memory is to be allocated, the first element is dequeued (if the queue is not

empty) and the corresponding element in the sequence is reused to hold another segment. If the queue is empty, the sequence has to be added on to for a new segment to be created.

Functions:

Uint32_t map(length): creates new segment, returns identifier
Uint32_t get_element(m,n): returns 32 bit word at segment m, offset n.
void unmap(identifier): deletes segment
void put(m, n, value): store a value in segment m, offset n
Uint32_t size(identifier): get size of a segment
Uint32_t next_instruction(&counter): returns next operation, moves counter

Below is a diagram of our memory structure



PROCESSOR

The processor abstraction defines the program counter as a local variable and sets it to zero. Then reads one instruction at a time from segment 0, decodes the word and calls an appropriate function. Every one of the 14 UM instructions is performed by a specific function.
The registers will be a global array of length 8 that holds 32 bit words.

Functions:
Public:

process(word) - accepts 32 bit word and calls the appropriate function with appropriate arguments

Private:

Call_instruction: uses switch statement to call appropriate function
14 instructions(word): One function per instruction
get_ra(word): returns value of register ra from a 32-bit instruction
get_rb(word): returns value of register rb from a 32-bit instruction
get_rc(word): returns value of register rc from a 32-bit instruction
get_opcode(word): returns value of opcode from a 32-bit instruction

Other Questions:

Below are some example inputs:

- A single argument (the pathname for a file) that contains machine instructions.
- Stdin and stdout are used for the implementation of the UM input and Output instructions: e.g. um program.um < example_input.txt > example_output.txt
- The input file is machine instructions for the UM - each instruction takes in a 32 bit word and stores it in big endian form. The first 4 bits tell us which instruction to execute, and 3 bits will tell us which and how many registers to use (use anywhere from 0 to 3 registers) at the far right.
- The load instruction i unique - immediately after the instruction, the register is denoted, and all the remaining bits are used to represent the value to be loaded into the register.

Below are some example outputs:

- The program will output to stdout - it will print a single 9 bit value from $r[C] as an ASCII character
- If the program is used incorrectly, it will output to stderr.

Representation and Invariants:

For every machine instruction that the client provides, our emulator simply performs the instruction in a single loop calling the modules we create. The program counter must always point to the next instruction word necessary to be executed. That is not counting the tiny amount of time during which the counter must be incremented after a word is picked up. For each segment in memory, there is a single word that corresponds to it - each 32 bit segment indicator will either directly correspond to a segment of memory, or it will be invalid.

Test Plan

Our testing will be very focused on testing each instruction as separately as possible, and then building up to test additional instructions that rely on previously tested cases. Furthermore, we want to focus testing corner cases for several instructions. Testing of the segment is included in the unit tests submitted with the design document, while testing of the UM is below.

We want to test similarly to how we did so in the lab - we will write the test code in a plain file, similar to assembler code (just with far fewer instructions). We'll write a program that reads the example code, uses bitpack to set it to a 32 bit word, and write to a specified output. The instructions can be fed into the UM as input so that we can determine the functionality of the program.

We want to output text under conditionals and then output what is contained in the registers just like we did in the lab. The test program will compare the expected output of each unit text to the actual output produced by our machine. Due to the fact that the test program will run all tests, we will always know if changes we make to the UM program to fail tests that it previously passed.

We will start by testing the halt instruction. We need to do this to make sure the UM stops exactly where we want it to, as it is one of the most basic instruction that is used frequently in other instructions. Next, we want to test our loadval function. We then want to test the input and output functions to make sure that they are performing properly. With these basic tools, we can begin to test our emulator to input a sequence of machine instructions and output what we want it to (it will match the expected output). We are going to use some of the provided binary programs in order to make sure that the UM performs at the speed we want it to.

Once our emulator is efficient in taking all the instructions provided by the user, we will have to test the corner cases: We need to make sure that the emulator can handle a lot of segment creations and destructions (mapping and unmapping) and perform well. We need to make sure that we properly fail on everything we are

supposed to fail on - when working unmapped segments or loading values from out of bounds of a segment. We also need to make sure the emulator can handle all jumps in the machine instruction. We will also need to design initial tests for all 13 instructions.

Below are some examples on how we will test the instructions:

In order to test output, we will use our loadval to load a value into the register and output it, making sure that the character we expect is outputted. Simple tests such as this are incredibly important, as we will build on it and rely that it works on many instructions. We want to test all the instructions by making sure that they first work for simple inputs. For example, we can test the add instruction by loading two values into registers and adding them into another register and then outputting the value. We can test multiplication and division in the same way. Bitwise NAND will be tested by NANDing two values together and evaluating the output. We want to write several different tests that are similar to this - we need to make sure that these tests can add small values like 1 and 2 together with the correct result, but we also need to make sure that the result is what we want when we add very large values together. Segmented Load and Segmented Store can be tested by storing a value into segmented memory and accessing that value and making sure it is unchanged and in the correct spot.

Finally, Map and Unmap Segment can be tested by creating a new segment of space using Map Segment. We will store a value there, and will access it to make sure the value is correct and in the right spot. Then, we'll use Unmap Segment to unmap the segment, and will make sure we can't access it afterwards.