HW 6: um
Dan Glorioso & Brandon Dionisio (dglori02 & bdioni01)

Architecture

Modules:

- main - main function, argument checks, file opening
- Segment ADT - stores the address space in a UArray of Seq_T and contains functions for the implementation of the module (i.e., `new_segment()`, `map_segment()`, `unmap_segment()`, `get_word()`, `free_segment()`, `free_all_segments()`).
- Execute - reads in each instruction from the opened file and calls appropriate functions to execute the operations. Creates an instance of the segment module.
- Operations – functions that perform conditional move, segmented load, segmented store, addition, multiplication, division, bitwise NAND, and load value
- Getter - get register values and opcode
- Control flow – halt and load program
- User IO – functions that take in input from stdin and output to stdout

main:
The main.c file contains a `main` function that handles the command line and arguments.
Other functionality of main:

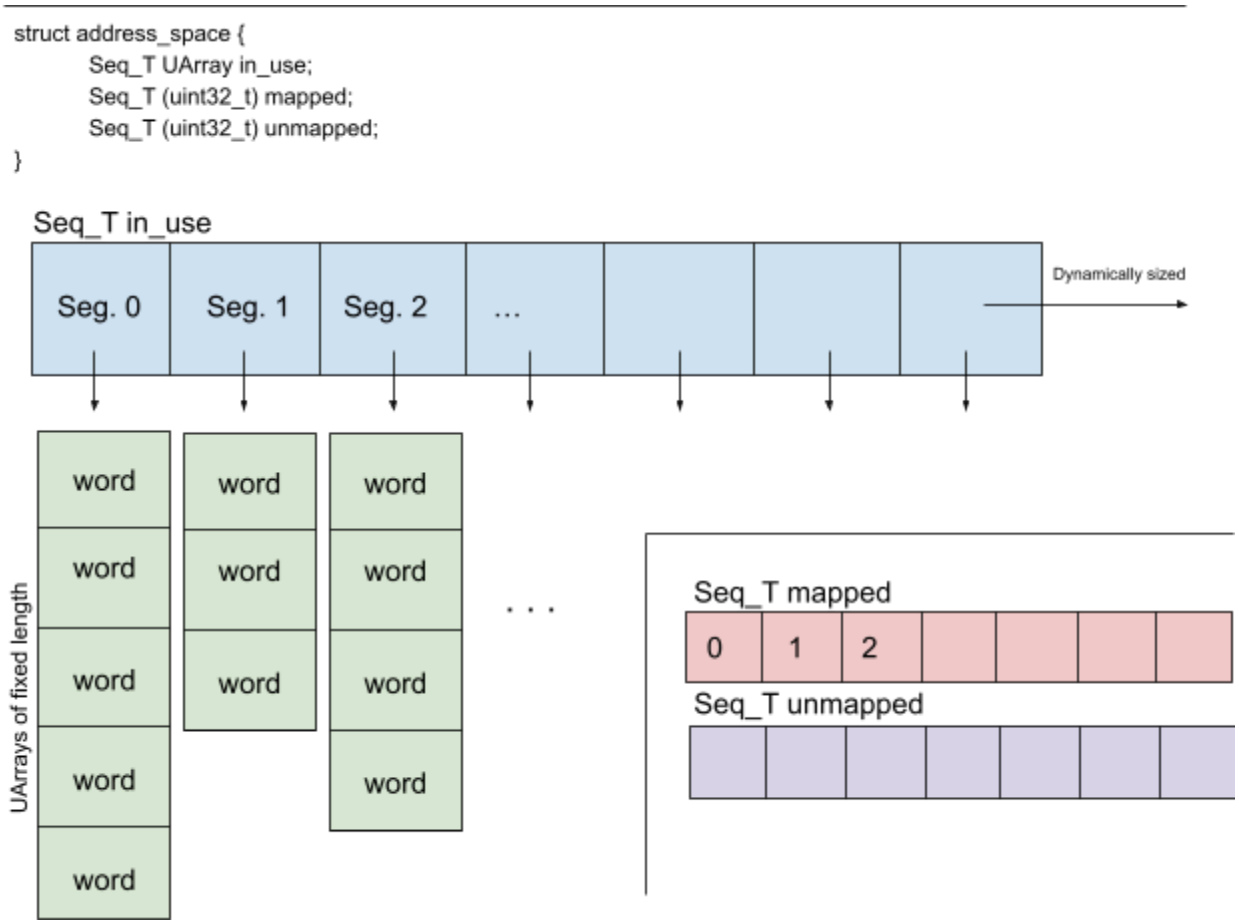
- Checks that the instructions in an input file are of proper length using the `fscan()` function. If the size variable from the returned st_stat is not divisible by 4, then the instructions are not formatted correctly.
- Opens the file specified in the command-line with the helper open_or_die function.
- Calls the execute module to perform the instruction in the input file.

Segment ADT:
Using Hanson's Data Structures, our entire address space will be stored in a struct containing a Seq_T (segments) of UArray (words). Additionally, the struct will contain a Seq_T of uint32_t identifiers of the mapped segments in use and a Seq_T of uint32_t identifiers to keep track of the unmapped segments. An instance of this ADT will be initialized within the execute module to allow the program to perform operations on the data while keeping the implementation in this module. In order to be able to ever-change the size of the collection of memory segments (dynamic), we opted to use a Seq_T.

To keep track of which segments within the outer Seq_T are storing data or not, we will use an additional Seq_T within the address_space struct that will hold the uint32_t identifiers of the segments that are not in use. In this way when the unmapped function is called on a certain segment, the index will be added to the unmapped sequence so it can be reused by future mapping instructions, as specified in the spec: "Future Map Segment instructions may reuse the identifier $r[C]" after being unmapped. Since each segment needs to be identified by a distinct 32-bit identifier, we will use a Seq_T of uint32_t within the address space struct to keep a running log of the identifiers of segments in use. When the mapping function is called, either a

new 32-bit uint32_t identifier will be created and added to the Seq_T of mapped identifiers or a former identifier will be reused from the unmapped Seq_T to store the new segment.

```
struct address_space {
        Seq_T UArray in_use;          //  Seq_T of all of segments that contain UArrays of words
        Seq_T (uint32_t) mapped;      //  sequence of distinct 32-bit identifiers of segments in use
        Seq_T (uint32_t) unmapped; //  sequence of distinct 32-bit identifiers of empty segments
}
```



An instance of this struct will be created when the execute module calls the new_address_space() function that allocates this struct to memory.

This module will contain the following functions:

- new_address_space()
    - Create a new instance of the address_space struct which consists of three Seq_T outlined in the struct above

- map_segment(int length)
    - Check the Seq_T unmapped for identifiers of segments that are not in use. If identifiers exist, remove uint32_t from unmapped and override that segment with the new segment created in this function.
    - Create a new segment of specified length
    - Specify distinct 32-bit identifier for new map segment and adds identifier to Seq_T mapped
- unmap_segment(uint32_t ID)
    - If $m[0] is attempted to be unmapped, the machine will fail and an unchecked run-time error will be thrown.
    - Remove a segment from the address_space by freeing that segment
    - Adds the removed segment's identifier to the Seq_T unmapped to keep track of which segments from the mapped sequence are no longer in use and can be overwritten
- uint32_t *word_at(uint32_t ID, uint32_t word_index)
    - Fetches the word at a specified segment identifier's word index
    - Can be used to iterate through all of the words in a segment
    - Returns a pointer to the word at the specified location
- free_segment(uint32_t ID)
    - Used by unmapped to deallocate a specific segment from the address_space
- free_all_segments()
    - Deallocates all of the segments stored in the address_space
    - Used by halt operation/end of program to free all segments

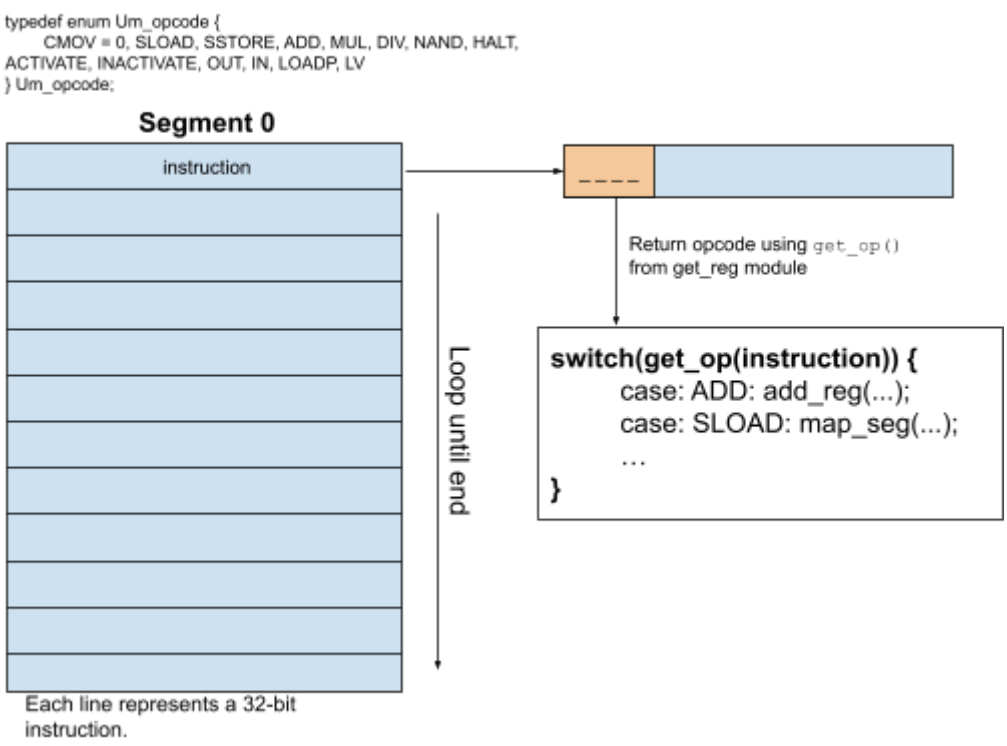
Execute:
First, this function allocates a new address space by calling the new_address_space function that will initially consist of empty Seq_T's (described more in the Segment module). Then, this module will create a segment of the size of the number of instructions by using the map_segment(int length) function in the Segment module. Since the 0 segment contains the program and the 0 segment is identified by the 32-bit identifier 0, the newly created segment will be populated by iterating through the input file instruction and using the uint32_t *word_at(0, uint32_t word_index) function from the Segment module to add each instruction to the UArray of words in segment 0. After each instruction the word_index will increment to the next word position. At the end of the file, the program instruction will all be stored in segment 0 of the address_space struct.

Also, this module will allocate a 8-element array of uint32_t that will represent the 8 registers designated $r[0] through $r[7]. Each of these registers in the array are initially set to 0.

The program counter will be set to $m[0][0] before the instructions are started. Before performing the instructions, proper error checking will be run to ensure that the program counter is not outside the bounds of $m[0] and that the program counter points to a word that codes for a valid instruction. If either of these conditions is not met, the machine will fail and an unchecked run-time error will be raised.

To perform the instruction from the input file that is stored in the 0 segment of the address_space, this module includes the Um_opcode enums from lab for each of the operations. To execute the proper operations, each instruction is passed into the `get_op()` function in the getters module that returns just the opcode of that instruction, which is then used in a switch statement to call the appropriate operation in either the operations module or the control flow module. A diagram of this process is provided below.

Diagram of Instruction to Operation Execution:



After each execution step, the program counter will be incremented to allow it to advance to the next word_index in the 0 segment, which is the next instruction to be performed. This process will continue through all of the instructions stored in the 0 segment, executing the proper instructions with the switch statement until all of the instructions from the 0 segment have been completed or a fault is raised.

Operations:
This module will contain the actual operations for the UM instructions. For each instruction from the program, each of its corresponding functions from this module will be executed.

- void cmov(uint32_t *reg_a, uint32_t *reg_b, uint32_t *reg_c)
    - Conditional move: if $r[C] ≠ 0 then $r[A] := $r[B]
    - Parameters are pointers to the elements of the register array specified by the instruction. This allows this function to update the values in the register array.

    - Checks that the value in reg_c is not equal to 0
    - If not, then deference value at reg_b and set reg_a's value to that new value
- void seg_load(Address_space space, uint32_t *reg_a, uint32_t *reg_b, uint32_t *reg_c)
    - Calls the uint32_t *word_at(uint32_t id, int word_index) function in the segment ADT to return the word stores in the segment identified by the value of reg_b and, within that, the word located at word_index of the value of reg_c.
    - The word returned from the word_at(...) function will then be set to reg_a using the pointer passed in the parameters.
- void seg_store(Address_space space, uint32_t *reg_a, uint32_t *reg_b, uint32_t *reg_c)
    - Deference the value passed in through the pointer to reg_c and save that register's value to the word_index of reg_b in the segment of identifier of the value of reg_a. Essentially, this will store the value of reg_c at the index of segment reg_a, index reg_b.
- void add(uint32_t *reg_a, uint32_t *reg_b, uint32_t *reg_c)
    - Parameters are pointers to the elements of the register array specified by the instruction. This allows this function to update the values in the register array.
    - Dereference the values of reg_b and reg_c. Calculate the sum of the two values and mod by 2^{32}. Then, using the pointer passed through the parameter, set reg_a to the calculated sum.
- void multiply(uint32_t *reg_a, uint32_t *reg_b, uint32_t *reg_c)
    - Parameters are pointers to the elements of the register array specified by the instruction. This allows this function to update the values in the register array.
    - Dereference the values of reg_b and reg_c. Calculate the product of the two values and mod by 2^{32}. Then, using the pointer to reg_a, set the value to the calculated product.
- void divide(uint32_t *reg_a, uint32_t *reg_b, uint32_t *reg_c)
    - Dereference the value of reg_b and reg_c.
    - If the reg_c passed into the function is 0, the machine will fail since dividing by 0 is invalid. An unchecked run-time error will be thrown.
    - If no error, calculate the quotients of reg_b and reg_c and take the floor. Save this value to the pointer to reg_a.
- void nand(uint32_t *reg_a, uint32_t *reg_b, uint32_t *reg_c)
    - Deference the value at reg_b and reg_c and perform a bitwise exclusive OR operation on the two values. Then, take the inverse of the result and save to the pointer of reg_a.
- void load_value(uint32_t instruction, uint32_t *load_reg_a)
    - Calls get_val from the getter module on the instruction to return the first 25 bits of the instruction, which is an unsigned binary value.
    - Using the pointer to load_reg_a passed in through the parameters, set the return value from get_val to the value of that register.

Getter:

This module returns the bits of a register in an instruction when called. This module relies on the bitpack module for fetching bits from a uint32_t at the specified locations based on the three-register instruction format in the UM spec.
typedef uint32_t Um_instruction;

- int get_A(Um_instruction)
    - Return the bits in register A of the specified instruction (3 bits starting at the 6th bit)
- int get_A_loadval(Um_instruction)
    - Return the single register A of the specified instruction (3 bits starting at the 25th bit)
- int get_B(Um_instruction)
    - Returns the bits in register B of the specified instruction (3 bits starting at the 3rd bit)
- int get_C(Um_instruction)
    - Returns the bits in register C of the specified instruction (3 bits starting at the 0th bit)
- int get_op(Um_instruction)
    - Returns the opcode of the specified instruction (4 bits starting at the 28th bit)
- int get_val(Um_instruction)
    - Return the value of an opcode 13 instruction, which is the unsigned binary value of the first 25 bits of the instruction.

Control flow:

- void halt(Address_space space)
    - Calls free_all_segments() on the address space.
    - Terminates the program with return EXIT_SUCCESS
- void load_program(Address_space space, uint32_t *reg_b, uint32_t *reg_c)
    - Segment $m[$r[B]] is duplicated and replaces $m[0], which is abandoned. The program counter is set to point to $m[0][$r[C]].

User IO:
Includes standard IO library: <include “stdio.h”>

- void input(uint32_t *reg_c, FILE *fp)
    - Wait for input from the specified filestream from the parameter when called.
    - Checks that the inputted character is of value between 0 and 255. The program fails if this condition is not met.
    - Once the end of input has been signaled, the inputted value is stored through the pointer to reg_c with the full 32-bit word in which every bit is 1.
- void output(uint32_t *reg_c, FILE *fp)
    - Checks that the value reg_c is between 0 and 255. The program fails if this condition is not met.
    - Writes the value at reg_c to the output stream specified in the parameter.

Implementation Plan

Note: Blue text represents UM segment abstraction testing

1. Within the main module .c file, create a `main` function and check execution with a simple print statement
2. Write a check inside main to ensure that the correct number of arguments are provided (`argc == 2`).
    a. Run the program without providing any arguments and with more than one argument to ensure that the program functions properly and prints the proper error message to stderr if not.
    b. Also check with proper argument input and ensure there is no output or errors.
3. Using the `fstat()` function, write a check inside main that ensures the instructions are of proper length in the provided input file. To do this, we will check that the `st_size` attribute within the `stat` struct returns a size (in bytes) that is evenly divisible by 4. This means that the instructions in the inputted file are valid. In the case that it is not evenly divisible by 4, the program will throw an *unchecked run-time error*.
    a. Check by inputting files that we know have both a valid number of bytes per instruction and files that do not have a valid number of bytes and confirm the program functions properly in both instances.
    b. To ensure that the program is checking correctly, print out the values returned from the st_stat and compare to the value we know were in the test input files.
4. Implement the file opening aspect of the program inside main.
    a. Open a valid .um file of machine instruction with the helper `open_or_die` function. If the input is not valid, the program will exit with a EXIT_FAILURE.
    b. Test with a valid .um file from the unit test files created in the lab to see that the program can open the file.
    c. Test with an invalid file to see if the program provides the error message to stderr and exits with code EXIT_FAILURE.
    d. Test with dev/null/ file to double-check the program's expected behavior.
5. Implement the Segment ADT module.
    a. Create the `Address_space` struct, which contains a sequence of UArrays representing the segments and two sequences of unsigned 32-bit integers: one holding mapped segment identifiers and the other holding unmapped segment identifiers (which can be used again).
    b. Implement the `Address_space new_address_space()` function, which creates a new instance of an `Address_space` object.
    c. Initialize an Address_space object in main and check that space was allocated correctly and that there's no errors.
    d. Assert that the size of the newly allocated object is correct from the beginning for each of the sequences in the struct.

        i. If errors occur, use print statements within the ADT module to output the size of the objects when being allocated.
    e. Implement the `void map_segment(int length)` function, which initializes a new `UArray_T` segment, initializes the words to 0, and then pushes the segment onto the address space.
    f. Check the functionality of this function by mapping on segments in main and ensuring that space was allocated correctly, with the segment being pushed onto the Address_space object and the values all being set to 0.
        i. We will use assert statements to ensure that each value for the size of the sequence is equal to 0.
    g. Check that a new uint32_t ID gets pushed onto the mapped sequence for each sequence that is created.
    h. Test the error handling of the function when given a negative or 0 length parameter. Ensure the program is terminated with an unchecked run-time error.
    i. Implement the `uint32_t *word_at(uint32_t id, int word_index)` function, which returns the word from the sequence at the given identifier at the given index.
    j. Try to set and get words from different indices in different segments.
    k. Test the error handling of the function by providing an id or word_index that are out of range of the address space and ensure that this does not work.
    l. Implement the `void unmap_segment(uint32_t ID)` function, which unmaps the segment at the given identifier.
    m. Map several segments and change the values of words in these segments using word_at. Test if this function can properly unmap these segments and add their uint32_t IDs to the unmapped sequence of IDs. We may also test this with Valgrind to ensure the segment was properly deallocated.
    n. Test that map_segment will map new segments to IDs that are unmapped instead of creating new IDs. We will do this by creating a number of segments and then unmapping segments and then mapping new segments, ensuring that the identifiers of the unmapped segments are used first to create the new segment.
    o. Implement the `free_segment()` and `free_all_segments()` functions.
    p. Test that memory for each function gets properly deallocated using Valgrind.
    q. Map a single segment and then unmap it to test free_segment() and map multiple segments and then free all segments in main to test free_all_segments()
6. Initialize the UM program with the opened file inside the execution module.
    a. Initialize the 8 registers as an 8-length array of type `uint32_t`; set them all to 0.
    b. Check that all of the registers are initially zero by using print statement that traverse through all of the value stored in each register
    c. Parse through the input file until `!EOF`, obtaining 32-bit words (`uint32_t`) by using the `bitpack` interface. For each word, we will call `getc` four times to be put into a 64-bit word with `Bitpack_newu` and then we will cast this word into type `uint32_t`.

    d. After successfully opening the input file in main, call this function in the execution module to parse through the file.
    e. With a file that contains 32 bits, check to see if we read it in correctly as a single uint32_t word.
    f. Test again with files that contain more words and see if we can read those correctly and in the proper big-endian byte order.
    g. Use the /dev/urandom to test that we can read in any C char as an integer.
7. Create the 0 segment in the execution module.
    a. Create a new instance of an `Address_space` object and map a new segment with `map_segment(int length)` that contains a length equal to the length of the input file (`st_size` / 4).
    b. With an input file of a known length, check that the new Address_space object gets initialized correctly: the unmapped sequence is empty, the mapped sequence contains 1, and the in_use sequence is 1-long and contains a UArray that has the length of the input file.
8. Set the 0 segment in the execution module.
    a. With each word read in from the file, use the `word_at(uint32_t id, int word_index)` function to put the word into the 0 segment.
    b. First, test with a file containing one instruction and see if the one word gets correctly inserted into the 0 segment.
    c. Then, try with a file containing several instructions and see if these words get correctly inserted.
9. Run additional segmenting tests.
    a. Map 100 segments and then store a unique word inside each. Parse through the sequence to retrieve each word and make sure that the segment has been mapped correctly.
    b. Try to map segment 0 and ensure an unchecked runtime error.
    c. Map 100 segments and then unmap them all. Check to see if the sequence is empty.
    d. Try to unmap segment 0 and ensure the program throws an *unchecked runtime error*.
10. Initialize the program counter to point to `$m[0][0]` with the `word_at(uint32_t id, int word_index)` function.
    a. Test with known instructions that the program counter points to the first instruction at this point. This checks again that the instructions were read-in and stored in the proper order.
11. Implement the getter module.
    a. Write the `int get_A(Um_instruction)` function that returns the bits representing register A at the given instruction.
    b. Write the `int get_A_loadval(Um_instruction)` function that returns the 3 bits starting at the 25th bit in the given instruction.
    c. Write the `int get_B(Um_instruction)` function that returns the bits representing register B at the given instruction.

    d. Write the `int get_C(Um_instruction)` function that returns the bits representing register C at the given instruction.
    e. Write the `int get_op(Um_instruction)` function that returns the bits representing the opcode at the given instruction.
    f. Test each of these functions by creating a main function in a separate file that creates a known uint32_t and obtains each of the bits representing the corresponding register. Go one-by-one through each function to check it is returning the expected value. If necessary, we will write out these instructions and then break them up into the register pieces we expect the program to return.
12. Loop and execute each instruction in the 0 segment.
    a. Loop through the words in the 0 segment with the `word_at()` function and use the getting functions to retrieve all information necessary for that instruction.
    b. Construct a switch statement such that each opcode will bring you to a new case.
    c. Update the program counter at the end of each loop.
    d. Read in the instructions from the 0 segment and write print statements for each instruction case from the switch statement to verify that we are able to differentiate the correct operators.
    e. Verify that the instruction opcodes result in the correct case for 0-14. We will test the switch statement by inputting each of these opcodes one-by-one and checking if the correct function is called. Since the operations are not yet implemented, we will just use print statements after each switch so that we can see which case is called for each opcode input.
    f. Check error handling for when the opcode equals 15.
13. Implement the Operations module which contains the functions driving the operation instructions.
    a. These operations include: move, segment load, segment store, add, multiply, divide, nand, and load value.
    b. Test the functionality of these functions with the unit testing framework and the UM instruction set tests included under the Testing Plan section.
    c. Run the segmented load instruction on a known instruction and check that `$r[A]` gets set to the value at `$m[$r[B]][$r[C]]`.
    d. Run the segmented store instruction on a known instruction and check that `$m[$r[B]][$r[C]]` gets set to the value at `$r[A]`.
    e. Run the segmented load and segmented store instructions on a word in a segment that is unmapped and verify that it throws an *unchecked run-time error*.
    f. Run the segmented load and segmented store instructions on a mapped segment such that the word is outside the bounds of the segment and verify that it throws an *unchecked run-time error*.
    g. Check error handling for the segmented load and segmented store instructions when the functions are not given the correct arguments.
14. Implement the Control flow operations module which contains the functions driving the operation instructions.
    a. These operations include: halt and load program

    b. Test the functionality of these functions with the unit testing framework and the UM instruction set tests included under the Testing Plan section.
    c. Run the load program instruction on segment `$m[$r[B]]`, where `$r[B]` = 0 and then halt. Verify that the operation was fast and that no errors were produced.
    d. Run the load program instruction on segment `$m[$r[B]]`, where `$r[B]` != 0 and then halt. Verify that no errors were produced and that the program counter is pointing to `$m[0][$r[B]]`. Also check that `$m[0]` has been replaced.
    e. Run the load program instruction on a segment that is unmapped and verify that it throws an *unchecked run-time error*.
15. Implement the User IO module which contains the functions driving the operation instructions.
    a. These operations include: output and input
    b. Test the functionality of these functions with the unit testing framework and the UM instruction set tests included under the Testing Plan section.
    c. We will also manually test this by typing in our own input through standard input to the program and check that the input is stored correctly in the proper registers.
    d. We will error check this function by inputting values that should not be inputted and check that the program functions correctly.
16. Inside each switch statement, call the corresponding function from its module.
    a. Segment load and store will make use of the `word_at()` function from the segment module.
    b. Create files that will individually test each instruction when read in.
    c. Test to see that segments are mapped and unmapped and are able to store and load words from the segmenter operations.
17. Test all the failure modes
    a. Create a case where:
        i. at the beginning of a machine cycle, the program counter points outside the bounds of $m[0].
        ii. at the beginning of a machine cycle, the program counter points to a word that does not code for a valid instruction.
        iii. a segmented load or segmented store refers to an unmapped segment.
        iv. a segmented load or segmented store refers to a location outside the bounds of a mapped segment.
        v. an instruction unmaps either $m[0] or a segment that is not mapped.
        vi. an instruction divides by zero.
        vii. an instruction loads a program from a segment that is not mapped.
        viii. an instruction outputs a value larger than 255
18. Test the program with reference input files.
    a. Diff our UM with the reference good-UM with the following files: cat.um, hello.um, midmark.um, sandmark.umz, advent.umz, and codex.umz.
    b. Verify that our UM will run midmark.um in under a minute.

Testing Plan

Our testing plan consists of the UM segment abstraction testing which is interleaved within our implementation plan along with the following UM instruction set tests:

- We will assume that each unit test produced from the instruction set tests will contain a proceeding halt instruction and will be built off of other instruction set tests; this is for conciseness.

Instruction Set Tests

- Halt:
  - Create a .um file with only the halt instruction. Check to see if the program correctly halts and doesn't produce any output or error.
- Load program:
  - Cannot test with unit testing framework.
- Map segment:
  - Cannot test with unit testing framework.
- Unmap segment:
  - Cannot test with unit testing framework.
- Conditional move:
  - Set `$r[C]` = 0, `$r[A]` = 10, `$r[B]` = 20. Run the conditional move instruction and check that `r[A]` = 10.
  - Set `$r[C]` = 9, `$r[A]` = 10, `$r[B]` = 20. Run the conditional move instruction and check that `r[A]` = 20.
  - Check error handling when the function is not given the correct arguments.
- Segmented load:
  - Cannot test with unit testing framework.
- Segmented store:
  - Cannot test with unit testing framework.
- Addition:
  - Set `$r[A]` = 10, `$r[B]` = 14, `$r[C]` = 7. Run the additional instruction and check that `r[A]` = 21.
  - Set `$r[A]` = 3, `$r[B]` = -4, `$r[C]` = -6. Run the additional instruction and check that `r[A]` = -10.
  - Set `$r[A]` = 1, `$r[B]` = 4294967296, `$r[C]` = 5. Run the additional instruction and check that `r[A]` = 5.
  - Check error handling when the function is not given the correct arguments.
- Multiplication:
  - Set `$r[A]` = 10, `$r[B]` = 4, `$r[C]` = 7. Run the multiplication instruction and check that `r[A]` = 28.
  - Set `$r[A]` = 10, `$r[B]` = -4, `$r[C]` = -6. Run the multiplication instruction and check that `r[A]` = 24.

    - Set `$r[A]` = 1, `$r[B]` = 2147483650, `$r[C]` = 2. Run the multiplication instruction and check that `r[A]` = 4.
    - Check error handling when the function is not given the correct arguments.
- Division:
    - Set `$r[A]` = 10, `$r[B]` = 24, `$r[C]` = 6. Run the division instruction and check that `r[A]` = 4.
    - Set `$r[A]` = 10, `$r[B]` = 49, `$r[C]` = -7. Run the division instruction and check that `r[A]` = -7.
    - Set `$r[A]` = 1, `$r[B]` = 10, `$r[C]` = 0. Run the division instruction and check that the program throws an *unchecked runtime error*
    - Check error handling when the function is not given the correct arguments.
- Bitwise NAND:
    - Set `$r[A]` = 10, `$r[B]` = 24, `$r[C]` = 24. Run the bitwise NAND and check that `r[A]` = 1.
    - Set `$r[A]` = 10, `$r[B]` = 24, `$r[C]` = 36. Run the bitwise NAND and check that `r[A]` = 0.
    - Check error handling when the function is not given the correct arguments.
- Output:
    - Set `$r[C]` to a variety of different values and verify that those values are printed to stdout.
    - Check that the program throws an *unchecked run-time error* when output tries to print out a value greater than 255.
    - Check error handling when the function is not given the correct arguments.
- Input:
    - Test with an input stream containing a variety of different **valid** values and verify that `$r[C]` is set to those values after the instruction call.
    - Test with an input stream containing a variety of different **invalid** values (out of range from 0 to 225) and check that error handling is as expected.
    - Check error handling when the function is not given the correct arguments.
- Load value:
    - Set the three less significant bits from the opcode to 5 and check that `$r[A]` = 5 after the function call.
    - Check error handling when the function is not given the correct arguments.
- Additional tests:
    - Test each command with registers that don't exist, say, `r[10]`.