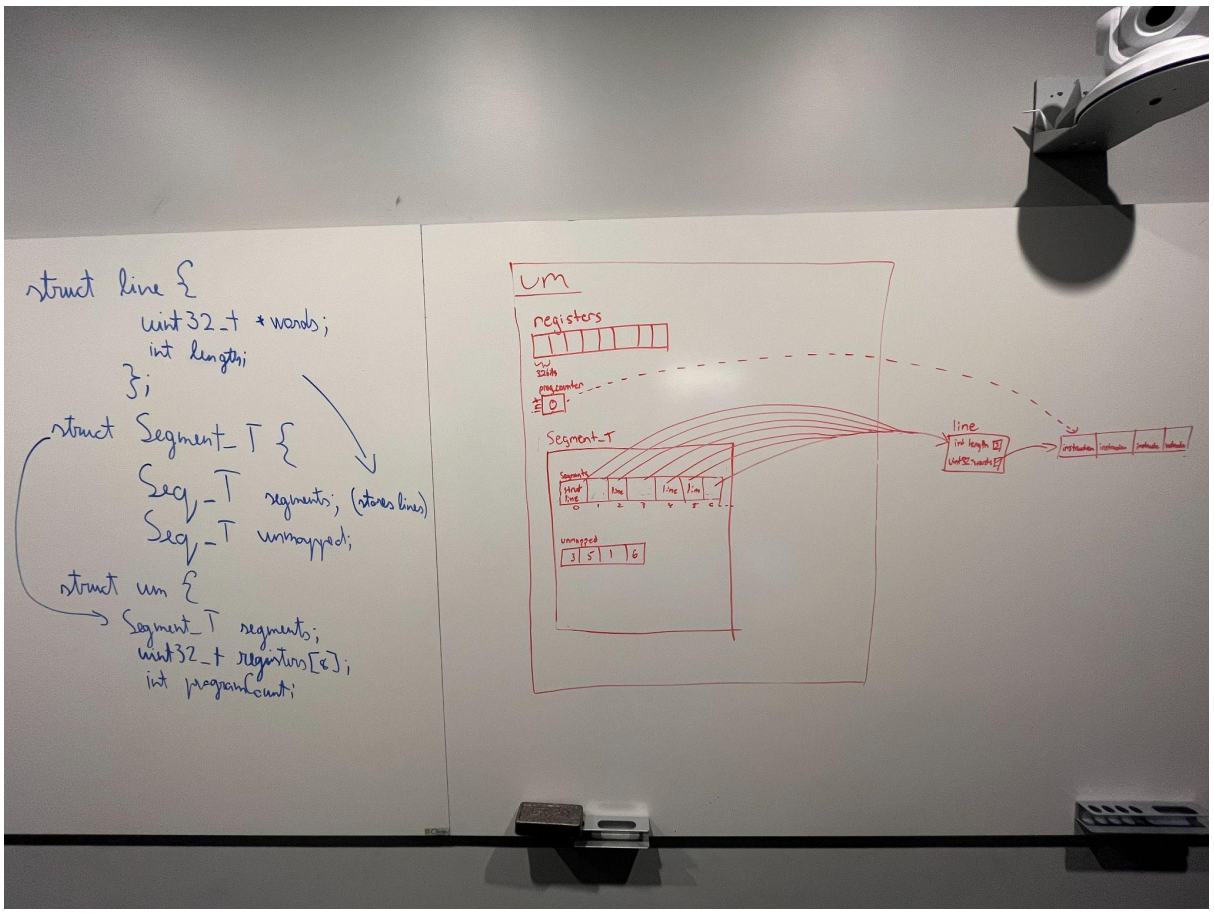
Architecture: We plan to implement the UM with 3 different modules:

- Segment_T:
    - This module contains our implementation for handling segments in the um module.
    - It contains a Hanson Sequence to store and identify each segment, which will each be represented by a struct that contains a size variable to keep track of the number of words in the segment and a pointer to an array of 32-bit words that will contain the actual segment itself.
    - It will also contain a Hanson Sequence to track the identifiers (indexes) allocated within the other sequence but unmapped by the um. This is done to reuse these elements within the other sequence to save time and space.
    - This module is used within the um module to emulate the segmented memory of the um.
- um:
    - This module contains our emulation of the um and how it is meant to use its registers and memory.
    - It contains an array of size 8 with 32-bit unsigned integers, which will emulate our registers.
    - It will also contain a 32-bit integer, programCount, to track where we are in running the current program, stored in segment 0.
    - It will also contain an instance of a Segment_T that emulates the segmented memory of the um for the user to map, unmap, and store data within.
    - The main module uses this module to create an um and run it.
- main:
    - This module contains an instance of the um, runs it, and frees it.



Implementation and Testing Plan:

1. Implement the struct for an individual segment line containing a size variable to keep track of the number of words in the segment and a pointer to an array of 32-bit words containing the actual segment itself.
    a. Create a .c file where we declare the struct and define the related functions, which will be a new and free function.
        i. The new function takes in an integer as a parameter: the number of words in that segment. It allocates memory for a 32-bit integer array of the size provided and keeps track of that using the pointer in the struct.
        ii. The free function takes in a pointer to an instance of the struct and frees it.
        iii. Testing: The new function can be tested by creating a separate file with a main where we allocate memory for a segment and using Valgrind, ensuring that the right amount of memory was allocated. To test the free function, we can allocate memory using new and free it using the free function. Then, we check for memory leaks with Valgrind.
2. Implement the Segment_T data structure that contains a Hanson Sequence to store and identify each segment and another Hanson Sequence to track the identifiers (indexes) allocated within the other sequence but unmapped by the um.

a. Create a .c file for Segment_T where we declare a struct for it and define the functions related to it.
b. It will have the following functions: new, free, mapSegment, unmapSegment, at, loadProgram, loadWord, and loadSegment.
    i. The new function initializes a Segment_T variable with one segment containing all zeros.
    ii. The free function takes in a pointer to an instance of the struct and frees it.
    iii. The mapSegment function takes a uint32_t, signifying the number of words to allocate in a new segment. It creates an individual segment struct with that amount of words. It then checks the tracker sequence to see if there are any open slots from previously unmapped segments. If there is not, it adds the new individual segment struct to the sequence of segments and returns the index in the sequence it was stored at. If a slot is open in the sequence, it does the same process in the open slot.
    iv. The unmapSegment function takes in an identifier and deallocates the memory associated with the segment identified by the identifier.
    v. The at function takes in an identifier and returns a pointer to a segment struct described earlier.
    vi. The loadProgram function takes in an identifier, duplicates the segment it identifies, and puts it in segment[0]. The value in segment[0] will be freed to ensure no memory leaks.
    vii. The loadWord function will take an identifier, an offset, and a 32-bit word value, and it will set the segment identified by the given identifier to have the given word at the given offset.
    viii. The loadSegment function will take an individual segment struct and an index and store the segment at the given index in the sequence of segments.
    ix. Testing: The new function can be tested by creating a separate file with a main where we allocate memory for a segment and using Valgrind, ensuring that the right amount of memory was allocated. To test the free function, we can allocate memory using new and free it using the free function. Then, we check for memory leaks with Valgrind. To test the mapSegment function, we can map a segment and check that the right amount of memory was allocated using Valgrind. To test unmapSegment, we can map a segment and then unmap it using the function and check using Valgrind that no memory leaks are possible. For the at function, we can store some segments and then access them using at and ensure the right struct was accessed and that no memory leaked during the storing/accessing. For loadProgram, we can store multiple segments and then load them into the

first segment and check with Valgrind that we are not leaking memory because of the overwriting that is going on. To test the loadWord function, we can store some value using the function, access it using the at function, and assert that the value stored and received is the same by accessing the right index within the segment that the at function returned. For loadSegment, store it using the function and access it using at. Then, iterate through the data stored and accessed and ensure that it is all the same.

3. Implement the um data structure, the array of 32-bit registers, and the functions that help emulate our segmented memory and registers for the universal machine. We also implement the operations for the um in this file.
    a. Create a .c file for the data structure where we declare a struct and its associated functions.
    b. It will have a function for each of the 14 operations necessary for the um and a few others, like new and free.
        i. The new function initializes an um variable that is a universal machine.
        ii. The free function takes a pointer to an instance of an um and frees it.
        iii. The conditional move function takes 3 values identifying registers and moves the value from one to the other dependent on the value in the 3rd
        iv. The segmented load function takes in an identifier and an offset value and stores the value in that location in a segment in register A.
        v. The segmentedStore function utilizes the functionality of the Segment_T data structure to store the data from a register in a segment at an offset.
        vi. The addition function takes the values in registers B and C, gets their sum, and stores that in register A.
        vii. The multiplication function takes the values in registers B and C, gets their product, and stores that in register A.
        viii. The division function takes the values in registers B and C, divides the value from B with the value from C, and stores that in register A.
        ix. The bitwise NAND operator takes the values of registers B and C, uses the bitwise and operator on them, then negates the result of the and operation and stores it in register A.
        x. The halt function stops processing any code, frees all memory, and cleans up the program.
        xi. The mapSegment function utilizes the functionality of the Segment_T data structure to create a new segment with a certain amount of words.
        xii. The unmapSegment function utilizes the functionality of the Segment_T data structure to clear the segment at a given identifier.
        xiii. The output function prints the value at a given register to output.

xiv. The input function gets input from standard input and stores that in register C. Input must be between 0 and 255.

xv. The LoadProgram function utilizes the functionality of the Segment_T data structure to put a given segment into segment 0 and set the programCounter to the given index.

xvi. The LoadValue function takes a value and a register id and stores the value in the register.

xvii. Testing: The new function can be tested by creating a separate file with a main where we allocate memory for a segment and using Valgrind, ensuring that the right amount of memory was allocated. To test the free function, we can allocate memory using new and free it using the free function. Testing of the operations of the um is described later.

Testing the um:

- void build_halt_test(Seq_T stream);
    - This unit test tests the functionality of the halt operation, which should shut down the program. It will just contain the halt function.
- void build_output_test(Seq_T stream);
    - This unit test tests the functionality of the output operation, which should print out the value stored in one of the registers. For this, we will use loadVal to store different values in registers and print them out using the output function. Finally, we will stop the program by calling the halt function.
- void build_load_value_test(Seq_T stream);
    - This unit test tests the functionality of the load value operation. Calls the load value operation to load a specific value into a register, then outputs the contents to assert it was done correctly. Finally, halt will stop the test.
- void build_verbose_halt_test(Seq_T stream);
    - This unit test tests the functionality of the halt operation, which should shut down the program. It will contain the halt function but also include loadVal and output operations to test that halt shut the program down. Otherwise, it will print out "Bad!"
- void build_input_test(Seq_T stream);
    - This unit test tests the functionality of the input operation, which should read in a value from standard output and store it in a register. For this, we will use the input function to read in values from standard input and then print them out using the output function. Finally, halt will be used to stop the test.
- void build_add_test(Seq_T stream);

    - This unit test tests the functionality of the add operation, which should get the sum of elements in registers b and c and store the sum in a. For this, values will be stored in registers a, b, and c using loadVal and printed out using output. We expect the value in register a to be overwritten by the addition.
- void build_multiplication_test(Seq_T stream);
    - This unit test tests the functionality of the multiply operation, which should get the product of elements in registers b and c and store the sum in a. For this, values will be stored in registers a, b, and c using loadVal and printed out using output. We expect the value in register a to be overwritten by the multiplication.
- void build_division_test(Seq_T stream);
    - This unit test tests the functionality of the divide operation, which should get the division of elements in registers b and c and store the sum in a. For this, values will be stored in registers a, b, and c using loadVal and printed out using output. We expect the value in register a to be overwritten by the division.
- void build_nand_test(Seq_T stream);
    - This unit test tests the functionality of the NAND operation, which should use the bitwise NAND operation on values in registers b and c and store it in register a. For this, we will use the loadVal function to store values in registers, output to print them out, NAND to use the bitwise NAND operator on values in registers, output to print out the value that is the result of the NAND, and finally halt to stop the test from running.
- void build_conditional_move_test(Seq_T stream)
    - This unit test tests the functionality of the conditional move operation. It will utilize other previously tested operations to load values into registers and, based on the value in one register, move a value from one to the other, then print the result register to check.
- void build_segment_load_test(Seq_T stream);
    - This unit test tests the functionality of the segment load operation. It will use previously implemented and tested operations to load values into registers and store the value from a slot in segmented memory into a given register. It will output the register contents to assert it is as expected.
- void build_segment_store_test(Seq_T stream);
    - This unit test tests the functionality of the segment store operation. It will use previously implemented and tested operations to load values into registers and store the value from a given register into a slot in segment 0. It will output the segment contents to assert it is as expected.
- void build_map_segment_test(Seq_T stream);

    - This unit test tests the functionality of the map segment operation. It maps a new segment and uses segment store to store values within the new segment. It then calls segment load to access the same value and confirms the value we loaded in and the value we accessed is the same.
- void build_unmap_segment_test(Seq_T stream);
    - This unit test tests the functionality of the unmap segment operation, which should unmap a segment allocated in memory. We will first allocate memory using map and then store information in the segment and ensure the data is stored using segmented load and output. Then, we can unmap the segment and try to reaccess it using segmented load, which should fail since it will be out of the range of the segments.
- void build_load_program_test(Seq_T stream);
    - This unit test tests the functionality of the load program operation. It uses previously implemented and tested operations to map a segment, load it with specific values pertaining to a different program, and then load it into segment 0. It can be checked by whether or not the program begins running the new given program. We can also use segmented load to access values from the segment and print them out using the output to ensure the right segment was stored. Halt will stop the test from running.
- void build_combined(Seq_T stream);
    - This unit tests the functionality of all operations included in the um. It will use the different operations to perform actions on values in registers and segments printed using output. This test ensures that all operations can work in the same program.