UM Design Document

Architecture

Module Overview:

UM.c, segments.c/h, operations.c/h, um_status.c/h.

Data Structures:

Type defined structures:

- We use a `Segments_T` to represent all the memory segments and the unmapped segment IDs:

```
struct Segments_T {
      Seq_T      unmapped_ID; (element type: uint32_t)
      Seq_T      segments (element type: Segment_Info_T)
}
```

- We use a **Hanson sequence** to store pointers to segments, where the sequence index equals the segment number
- We use a **Hanson sequence** to store unmapped IDs for future use when map segment is called:
  - Everytime the user maps a new segment, we check whether the sequence unmapped_ID is empty;
  - if not, it retrieves the first element and uses it as the ID for the new mapped segment; otherwise it assigns a new segment.

```
Seq_T unmapped_ID (element type: uint32_t)
```

- We define a struct `Segment_Info_T` to represent all the words in a segment (using **Hanson sequence)** and the number of words/instructions within that segment

```
struct segment_info {
      Seq_T      words; (element type: uint32_t)
      uint32_t   size;
}
```

- We define a struct `UM_T` to represent the current status of the UM, including the status of the registers, a program counter, and the segments of instructions.

```
struct UM_T {
      Segments_T segments;
```

```
        uint32_t   program_counter;
        UArray_T   registers; (element type: uint32_t)
}
```

➢ We use a Hanson UArray to represent the 8 registers, where each element in the UArray is a uint32_t representing each of $r[0] -$ r[7].

Module Description:

*********UM.c*********

- UM.c functions as the driver code that interacts with the user via command line.
- UM.c takes command line inputs, processes the input program.um and input texts, and initializes segments.
- UM.c interacts with um_status.c to initialize 8 empty registers, as well as with operations.c as the entry point to call the operations in some_program.um.

*********operations.c/h*********

- operations.c can process any of the 14 well-defined operations (opcode 0-13) by calling helper functions in um_status.c.
- operations.c has access to registers values via a struct pointer.
- Contains a query loop which calls different operations depending on the opcode.

*********um_status.c/h*********

- um_status.c/h represents the current state of the UM.
- It contains information about the eight registers.
- It contains functions that initialize the state of the UM (by setting all registers to 0 and load the first sequence of instructions to the 0 segment), change the state of registers, free the memory usage of the UM, extract a specific word from a segment, put a specific word to a segment at given index, and extract values from instructions (ie. opcode, register A, B and C).

*********segments.c/h*********

- segments.c allocates, deallocates, and modifies memory for segments storing instructions.
- It has functions to map and unmap a segment, and recycle an unmapped 32-bit identifier for future mapping use.
- This module is self-contained, managing all aspects of memory without depending on other modules.

Module Interaction:

Segments.h → um_status.h

- ❖ um_status.h contains memory-associated functions as wrapper functions of those in segments.h
- ❖ um_status.h contains word extraction/storing functions as wrapper functions of those in segments.h
- ❖ um_status.h contains an initialization function that uses segments.h to populate the 0 segment with instructions read from the input .um file.

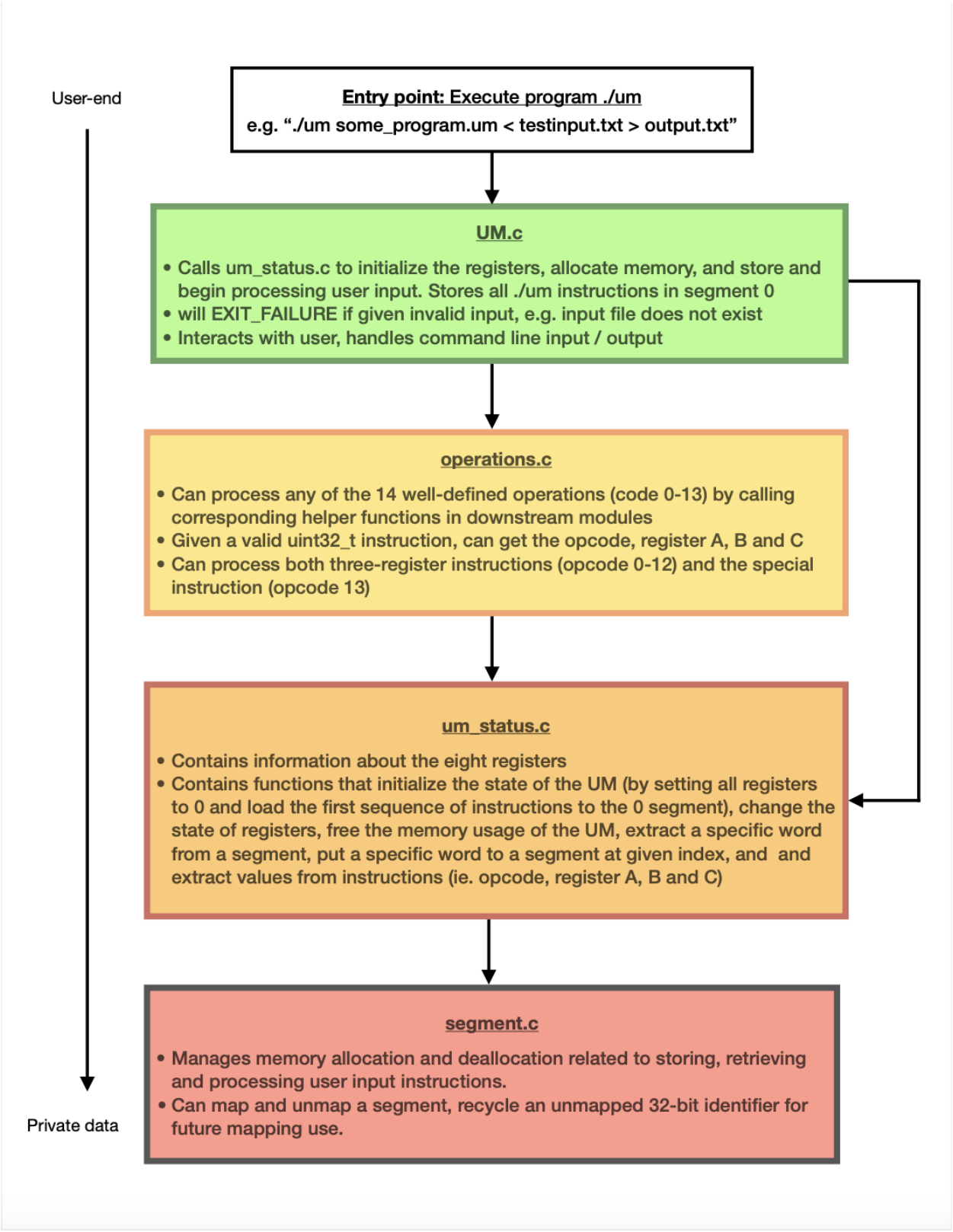
um_status.h → operations.h

- ❖ operations.h would call memory-associated functions from um_status.h when dealing with instructions regarding memory (map, unmap, duplicate, etc,)
- ❖ Call operations based on words(instructions) extracted by extract_word() in um_status.h
- ❖ Update register information by calling set_register_value() in um_status.h

operations.h → um.c
um_status.h → um.c

- ❖ Read input from .um file, and then call the initialization function in um_status.h (0 segment is populated with input)
- ❖ Perform instructions through functions in operations.h

(picture on the next page)


User-end
Entry point: Execute program ./um
e.g. "./um some_program.um < testinput.txt > output.txt"
UM.c
• Calls um_status.c to initialize the registers, allocate memory, and store and begin processing user input. Stores all ./um instructions in segment 0
• will EXIT_FAILURE if given invalid input, e.g. input file does not exist
• Interacts with user, handles command line input / output
operations.c
• Can process any of the 14 well-defined operations (code 0-13) by calling corresponding helper functions in downstream modules
• Given a valid uint32_t instruction, can get the opcode, register A, B and C
• Can process both three-register instructions (opcode 0-12) and the special instruction (opcode 13)
um_status.c
• Contains information about the eight registers
• Contains functions that initialize the state of the UM (by setting all registers to 0 and load the first sequence of instructions to the 0 segment), change the state of registers, free the memory usage of the UM, extract a specific word from a segment, put a specific word to a segment at given index, and and extract values from instructions (ie. opcode, register A, B and C)
segment.c
• Manages memory allocation and deallocation related to storing, retrieving and processing user input instructions.
• Can map and unmap a segment, recycle an unmapped 32-bit identifier for future mapping use.
Private data

Functions and Contracts

segments.h/.c

Self-contained

```
/********** map_segment ********
 *
 * Allocates a new segment of memory with the specified length.
 * Initializes all values to 0. Attach the segment to the back of the sequence of
 * segments
 * Reuse an available ID if able
 *
 * Parameters:
 *     uint32_t length: length of the segment for memory allocation
 *     Segments_T segments: modify segments to add one with given length
 * Return:
 *     segment ID
 *
 * Expects:
 *     There is enough heap space to allocate
 *
 * Notes:
 *     Will CRE if input_um is NULL
 *
 ****************************/
```

uint32_t **map_segment**(Segments_T segments, uint32_t length)

```
/********** unmap_segment ********
 *
 * Free a segment of memory based on a segment ID
 * Recycle the ID
 *
 * Parameters:
 *     uint32_t seg_ID: The identifier of the segment to be deallocated.
 *     Segments_T segments: Where the unmap happens
 *
 * Return: None
 *
 * Expects:
 *     Segment ID is valid
 *
 * Notes:
 *
 ****************************/
```

void **unmap_segment**(Segments_T segments, uint32_t seg_ID)

```
/********** get_word ********
 *
 * extract a word (instruction) from a given segment ID and offset within the segment
 *
 * Parameters:
 *     Segments_T segments: where the words are extracted from
 *     uint32_t seg_ID: The identifier of the segment to be accessed.
 *     int offset: where the word is within the segment of instructions
 *
 * Return: the word at seg_id and offset
 *
```

```
 * Expects:
 *     Segment ID is valid, offset is not out of range
 *
 * Notes:
 *
 *****************************/
```

uint32_t **get_word**(Segments_T segments, uint32_t seg_ID, int offset)

```
/********** set_word ********
 *
 * Sets a word in the specified segment at the given offset to the provided value.
 *
 * Parameters:
 *     Segments_T segments: memory where word is stored
 *     seg_ID, offset, value: Segment identifier, position within the segment, and the
 *     value to set.
 *
 * Return: none
 *
 * Expects:
 *     Segment ID is valid, offset is not out of range
 *
 * Notes:
 *
 ****************************/
```

void **set_word**(Segments_T segments, uint32_t seg_ID, int offset, uint32_t value)

```
/********** duplicate ********
 *
 * copy segment from source ID to destination ID
 *
 * Parameters:
 *     Segments_T segments: where the duplication takes place
 *     uint32_t source_ID, uint32_t destination_ID: positions of the source segment
 *     and destination segment
 *
 * Return: none
 *
 * Expects:
 *     Segment IDs are valid which means there are segments at given indices.
 *
 * Notes:
 *
 ****************************/
```

void **duplicate**(Segments_T segments, uint32_t source_ID, uint32_t destination_ID)

um_status.h/.c

```
#include "segments.h"


/********** initialize_um ********
 *
 * Initializes a new UMState representing the state of a UM machine.
 * All registers are set to 0, and only segment 0 is initially mapped.
 *
 * Parameters: char* input: all instructions from .um file
 *             all_segments segments: the memory handling ADS to store instructions
 *
 * Return: a UM_T struct representing the UM internal machine structure
 *
 * Expects:
 *     the 0 segment is mapped and contain instruction from the .um file
 *
 * Notes:
 *     Will CRE if input_um is NULL
 *
 ****************************/
```

UM_T **initialize_um**(char* input)

```
/********** set_register_val ********
 *
 * Sets the value of a specified register.
 *
 * Parameters:
 *     uint32_t register_index: Index of the register to be set (0-7)
 *     uint32_t value: value to be set
 *     UM_T um: UM information
 *
 * Return:
 *     none
 *
 * Expects:
 *     Register index is within valid range (0-7)
 *
 * Notes:
 *     Directly modifies the value of a specific register.
 *
 ****************************/
```

void **set_register_val**(UM_T um, uint32_t register_index, uint32_t value)

```
/********** get_register_val ********
 *
 * Retrieves the value of a specified register.
 *
 * Parameters:
 *     uint32_t register_index: Index of the register whose value is to be retrieved
 *     UM_T um: UM information
 *
 * Return:
 *     Value of the specified register.
 *
 * Expects:
 *     Register index is within valid range (0-7)
 *
 * Notes:
 *     Provides read access to a specific register's value.
```

```
 *
 ****************************/
```

uint32_t **get_register_val**(UM_T um, uint32_t register_index)

```
/********** free_um ********
 *
 * Frees all memory used by the Universal Machine, including segments and registers
 *
 * Parameters:
 *     UM_T um: UM information
 *
 * Return:
 *     none
 *
 * Expects:
 *     All allocated memory segments and register array are accessible.
 *
 * Notes:
 *     Ensures clean exit with no memory leaks.
 *
 ****************************/
```

void **free_um**(UM_T um)

```
/********** um_extract_instruction ********
 *
 * Frees all memory used by the Universal Machine, including segments and registers
 *
 * Parameters:
 *     uint32_t seg_ID: The identifier of the segment
 *     uint32_t offset: The offset within the segment where the instruction is
 *     located
 *     UM_T um: UM information
 *
 * Return:
 *     The extracted instruction (uint32_t)
 *
 * Expects:
 *     Segment ID is valid, offset is not out of range.
 *
 * Notes:
 *     Fetches an instruction from memory for execution
 *
 ****************************/
```

uint32_t **um_extract_instruction**(UM_T um, uint32_t seg_ID, uint32_t offset)

```
/********** um_set_instruction ********
 *
 * Frees all memory used by the Universal Machine, including segments and registers
 *
 * Parameters:
 *     uint32_t seg_ID: The identifier of the segment
 *     uint32_t offset: The offset within the segment where the instruction is
 *     located
 *     UM_T um: UM information
 *
 * Return:
 *     The extracted instruction (uint32_t)
 *
 * Expects:
 *     Segment ID is valid, offset is not out of range.
```

```
 *
 * Notes:
 *     Fetches an instruction from memory for execution
 *
 ****************************/
```

void **um_set_instruction**(UM_T um, uint32_t seg_ID, uint32_t offset, uint32_t word)

```
/********** um_map ********
 *
 * Allocates a new segment of memory with the specified length.
 * Initializes all values to 0. Attach the segment to the back of the sequence of
 * segments
 * Reuse an available ID if able
 *
 * Parameters:
 *     uint32_t length: length of the segment for memory allocation
 *     UM_T um: UM information
 *
 * Return:
 *     segment ID
 *
 * Expects:
 *     There is enough heap space to allocate
 *
 * Notes:
 *     Will CRE if input_um is NULL
 *
 ****************************/
```

uint32_t **um_map**(UM_T um, uint32_t length)

```
/********** um_unmap ********
 *
 * Free a segment of memory based on a segment ID
 * Recycle the ID
 *
 * Parameters:
 *     uint32_t seg_ID: The identifier of the segment to be deallocated.
 *     UM_T um: UM information
 *
 * Return: None
 *
 * Expects:
 *     Segment ID is valid
 *
 * Notes:
 *
 ****************************/
```

void **um_unmap**(UM_T um, uint32_t seg_ID)

```
/********** um_duplicate ********
 *
 * copy segment from source ID to destination ID
 *
 * Parameters:
 *     UM_T um: UM information
 *     uint32_t source_ID, uint32_t destination_ID: positions of the source segment
 *     and destination segment
 *
 * Return: none
```

```
 *
 * Expects:
 *     Segment IDs are valid which means there are segments at given indices.
 *
 * Notes:
 *
 *****************************/
```

void **um_duplicate**(UM_T um, uint32_t source_ID, uint32_t destination_ID)

```
/********** get_PC ********
 *
 * get the current program counter value
 *
 * Parameters:
 *     UM_T um: UM information
 *
 * Return:
 *     value of program counter in uint32_t
 *
 * Expects:
 *     Extracts the program counter from um
 *
 *****************************/
```

uint32_t **get_PC**(UM_T um)

operations.h/.c

```
#include "um_status.h"


/********** cmov ********
 *
 * Call reg_cmov to move r[B] into r[A] if r[C] != 0 and update register value
 * accordingly
 *
 * Parameters:
 *      UM_T um: a struct containing register values and instruction info

 * Return: None
 *
 * Notes:
 *     operations.c will call this function from a switch case
 *     based on the operation code
 *
 ************************/
```

void **cmov**(UM_T um);

```
/********** segmented_load ********
 *
 * Call reg_segmented_load to move $m[$r[B]][$r[C]] into r[A] and update register
 * value accordingly
 *
 * Parameters:
 *      UM_T um: a struct containing register values and instruction info
 * Return: None
 *
 * Notes:
 *     operations.c will call this function from a switch case
 *     based on the operation code
 *
 ************************/
```

void **segmented_load**(UM_T um);

```
/********** segmented_store ********
 *
 * Call reg_segmented_store to move $r[C] into $m[$r[B]][$r[C]] and update register
 * value accordingly
 *
 * Parameters:
 *      UM_T um: a struct containing register values and instruction info
 * Return: None
 *
 * Notes:
 *     operations.c will call this function from a switch case
 *     based on the operation code
 *
 ************************/
```

void **segmented_store**(UM_T um);

```
/********** add ********
 *
 * Perform addition and update register value accordingly
 *
 * Parameters:
```

```
 *      UM_T um: a struct containing register values and instruction info
 *
 * Return: None
 *
 * Notes:
 *     operations.c will call this function from a switch case
 *     based on the operation code
 *
 ************************/
```
void **add**(UM_T um);

```
/********** mult ********
 *
 * Perform multiplication and update register values accordingly
 *
 * Parameters:
 *      UM_T um: a struct containing register values and instruction info
 *
 * Return: None
 *
 * Notes:
 *     operations.c will call this function from a switch case
 *     based on the operation code
 *
 ************************/
```
void **mult**(UM_T um);

```
/********** division ********
 *
 * Call reg_division to perform division and update register value accordingly
 *
 * Parameters:
 *      UM_T um: a struct containing register values and instruction info
 *
 * Return: None
 *
 * Notes:
 *     operations.c will call this function from a switch case
 *     based on the operation code
 *
 ************************/
```
void **division**(UM_T um);

```
/********** bitwise_nand ********
 *
 * Call reg_bitwise_nand to perform bitwise NAND and update register value accordingly
 *
 * Parameters:
 *      UM_T um: a struct containing register values and instruction info
 *
 * Return: None
 *
 * Notes:
 *     operations.c will call this function from a switch case
 *     based on the operation code
 *
 ************************/
```
void **bitwise_nand**(UM_T um);

```
/********** halt ********
 *
 * Call reg_halt to stop computation and clear register value and downstream memory
 *
 * Parameters:
 *      UM_T um: a struct containing register values and instruction info
 * Return: None
 *
 * Notes:
 *     operations.c will call this function from a switch case
 *     based on the operation code
 *
 ************************/
```

void **halt**();

```
/********** map_segment ********
 *
 * Call reg_map_segment to map a segment and update memory
 *
 * Parameters: UM_t um
 *      UM_T um: a struct containing register values and instruction info
 * Return: None
 *
 * Notes:
 *     operations.c will call this function from a switch case
 *     based on the operation code
 *
 ************************/
```

void **map_segment**(UM_T um);

```
/********** unmap_segment ********
 *
 * Call reg_unmap_segment to unmap a segment and update memory
 *
 * Parameters:
 *      UM_T um: a struct containing register values and instruction info
 * Return: None
 *
 * Notes:
 *     operations.c will call this function from a switch case
 *     based on the operation code
 *
 ************************/
```

void **unmap_segment**(UM_T um);

```
/********** output ********
 *
 * Print the value in $r[C] to stdout
 *
 * Parameters:
 *      UM_T um: a struct containing register values and instruction info
 * Return: None
 *
 * Notes:
 *     operations.c will call this function from a switch case
 *     based on the operation code
 *
 ************************/
```

void **output**(UM_T um);

```
/********** input ********
 *
 * Call reg_input to write value in $r[C]
 *
 * Parameters:
 *      UM_T um: a struct containing register values and instruction info
 *
 * Return: None
 *
 * Notes:
 *     operations.c will call this function from a switch case
 *     based on the operation code
 *
 ************************/
```

void **input**(UM_T um);

```
/********** load_program ********
 *
 * Calls helper function to load program with
 *
 * Parameters:
 *     UM_T um: a struct containing register values and instruction info
 * Return: None
 *
 * Notes:
 *     operations.c will call this function from a switch case
 *     based on the operation code
 *
 ************************
```

void **load_program**(UM_T um);

Implementation & Testing Plan

Phase 1: Initial Setup and Basic Functionality

1. Setup Basic Structure
    a. Implementation: Create project structure with um.c, segments.c/h, operations.c/h, um_status.c/h.
    b. Testing:
        i. Ensure the modules link and compile correctly

2. Implement initialize_registers in um_status.c
    a. Implementation: Define a function to initialize eight registers to zero.
    b. Testing:
        i. Write tests to check each register is zero-initialized
        ii. Print the values of the registers
            1. Call initialize_registers.
            2. Loop through each register using a get_register_val function and check its value.
            3. Print the values of the registers for visual verification.

3. Implement Basic I/O in UM.c
    a. Implementation: Basic I/O operations for reading .um files.
    b. Testing:
        i. Test reading a small .um file
        ii. Check for correct handling of non-existent files.
        iii. Test reading a file in different format

Phase 2: Core Memory Management (segments.c)

1. Implement map_segment in segments.c
    a. Implementation: map_segment() which allocates and initializes a new memory segment.
    b. Testing:
        i. Write tests to check a new segment is initialized to zeros.
        ii. Map a new segment of a known size, e.g., 100 words.
            1. Verify that the returned segment ID is as expected (e.g., the first available ID).
            2. Check that the segment is initialized to zeros.

3. Unmap the segment.
4. Map a new segment of the same or different size.
5. Verify that the new segment ID is the same as the previously unmapped segment (recycling of IDs).
6. Check the initialization of the new segment.

2. Implement unmap_segment in segments.c
    a. Implementation: Function to deallocate a memory segment.
    b. Testing: Ensure proper deallocation and recycling of segment IDs.
        i. Map 10 segments, unmap 9 segments, and map 1 segment; expected behavior: program should recycle previously unmapped IDs
        ii. Run valgrind after unmap to ensure all heap memory has been deallocated
        iii. Map a new segment of a known size, e.g., 100 words.
            1. Verify that the returned segment ID is as expected (e.g., the first available ID).
            2. Check that the segment is initialized to zeros.
            3. Unmap the segment.
            4. Map a new segment of the same or different size.
            5. Verify that the new segment ID i6s the same as the previously unmapped segment (recycling of IDs).
            6. Check the initialization of the new segment.

3. Implement get_word and set_word in segments.c
    a. Implementation:
    b. Testing:
        i. Unit Test for get_word and set_word: Create tests to ensure that words are correctly set and retrieved from various segments and offsets.
        ii. Integration Test: Check integration with map_segment and unmap_segment functions.
            1. Map a new segment of a known size (e.g., 50 words).
            2. Use set_word to write a specific value (e.g., 12345) at a known offset (e.g., 10th word) in the segment.
            3. Use get_word to read the value at the same offset.

4. The value retrieved by get_word should match the value set by set_word (12345 in this case).

iii. Attempt to set and get words at invalid offsets (e.g., negative offset, offset beyond the segment length).

1. The system should handle invalid offsets gracefully, possibly through error handling or assertions.

Phase 3: Basic Operations and Integration

1. **Halt** Instruction
    a. Implementation: Implement the halt operation to stop the program execution.
    b. Test:
        i. Use a test program (halt_test.um) containing the halt instruction to ensure the program stops execution as expected.
        ii. Test Case:
            1. Input: A .um program where the halt instruction is placed after a few operations. Print program counter, which should point to the correct value depending on the instructions before halting.
            2. Command: Run the program and monitor if execution stops after the halt instruction.
            3. Expected Outcome: Program execution stops immediately after the halt instruction.

2. **Input** Instruction
    a. Implementation: Implement the input operation to read input and store it in a register.
    b. Test:
        i. Create a test that reads from a file, and print out the register values to verify that the input is correctly stored in the specified register.
        ii. Test Case:
            1. Input: A file containing a sequence of characters or numbers.
            2. Command: Read the contents of the file using the input instruction and store them in a register.

3. Expected Outcome: The values read match the contents of the file exactly.

3. **Output** Instruction
    a. Implementation: Implement the output operation to output the content of a register.
    b. Test:
        i. After the input operation is implemented, use it to store a value and then output it, checking for the correct display.
        ii. Test Case:
            1. Input: Store a known value (e.g., ASCII value 65) in a register.
            2. Command: Output the content of the register.
            3. Expected Outcome: The output should display the character 'A' (corresponding to ASCII 65).

4. **Divide** Instruction
    a. Implementation: Implement the divide operation for division of register values.
    b. Test:
        i. Divide various numbers and verify the results.
        ii. Test division by zero and ensure it throws an unchecked runtime error.
        iii. Combine divide and halt, e.g. divide, halt, divide, and use program counter to check whether the program stops at halt() correctly.
        iv. Test Cases:
            1. Division Test:
                a. Input: Set registers with values, e.g., r[A] = 20, r[B] = 5.
                b. Command: Divide r[A] by r[B].
                c. Expected Outcome: The result in r[A] should be 4.
            2. Division by Zero Test:
                a. Input: Set r[B] = 0.
                b. Command: Attempt division by zero.
                c. Expected Outcome: An unchecked runtime error occurs.

5. **Conditional Move**

a. Implementation: Implement the conditional_move operation to conditionally move data between registers.
b. Test:
    i. Create a test where r[C] is zero and non-zero and verify that r[A] is updated only when r[C] is non-zero.
    ii. Test Cases:
        1. Zero Condition Test:
            a. Input: r[A] = 5, r[B] = 10, r[C] = 0.
            b. Command: Perform conditional move.
            c. Expected Outcome: r[A] remains 5.
            d. Non-zero Condition Test:
        2. Input: r[C] = 1.
            a. Command: Perform conditional move again.
            b. Expected Outcome: r[A] changes to 10.

6. Segmented Load

a. Implementation: Implement the segmented_load operation to load data from memory segments into registers.
b. Test:
    i. Load values into registers and verify that the correct data is loaded from the memory segments.
    ii. Test Case:
        1. Input: Map a segment and set a word at offset 3.
        2. Command: Load the word from the segment into a register.
        3. Expected Outcome: The register contains the word set in the segment.

7. Segmented Store

a. Implementation: Implement the segmented_store operation to store data from registers into memory segments.
b. Test:
    i. Store values from registers into memory segments and verify that the data is correctly stored.
    ii. Test Case:
        1. Input: Store a known value in a register and have a mapped segment ready.

2. Command: Store the register's content into the segment at a specific offset.
3. Expected Outcome: The segment's content at the offset matches the register's value.

8. Addition

a. Implementation: Implement the addition operation to add values of two registers and store the result in a third register.
b. Test:
    i. Test addition of various numbers including edge cases like overflow.
    ii. Specifically test cases where the result overflows 32 bits; expected behavior: program should store the value after modding by 2^32.

9. Multiplication

a. Implementation: Implement the multiplication operation for multiplying register values.
b. Test:
    i. Multiply two numbers and verify the result.
    ii. Test cases where the product overflows 32 bits; expected behavior: program should store the value after modding by 2^32.

10. Bitwise NAND

a. Implementation: Implement the bitwise_nand operation.
b. Test:
    i. Test NAND operation with various number pairs and verify results.
    ii. Test basic cases like NAND of 0 and 0.

11. Map Segment

a. Implementation: Implement the map_segment operation to allocate new memory segments.
b. Test:

i. Verify that the new segment size matches the value in $r[C] and that the segment is correctly initialized using the tests written for the segments module

12. Unmap Segment

a. Implementation: Implement the unmap_segment operation to deallocate memory segments.
b. Test:
    i. Ensure that the function throws a runtime error if the segment ID is invalid (e.g., greater than 255).
    ii. Check to see if segment is freed using the tests written for the segments module

13. Load Program

a. Implementation: Implement the load_program operation to load and execute instructions from a specified segment.
b. Test:
    i. Create a test where the last instruction of a loaded program is executed to verify correct loading.

14. Load Value

a. Implementation: Implement the load_value operation to load a value directly into a register.
b. Test:
    i. Load various values into a register and verify correctness using print statements.

Phase 4: Advanced Memory and Operations

1. Implement Advanced Memory Functions
    a. Implementation: Add functions like duplicate in segments.c.
    b. Testing: Test duplication correctness, including edge cases like duplicate to the same ID.

2. Implement Remaining Operations
    a. Implementation: Add remaining operations like multiplication, division, input/output.
    b. Testing: Comprehensive tests for each new operation, ensuring accuracy.

Phase 5: UM Status and Execution

1. Implement UM_status functions in um_status.c
    a. Implementation: incrementally implement functions that modify register value including initialize_um(), free_um(), get_register_val(), and set_register_val().
    b. Testing:
        i. Exhaustively print values in registers when operations are executed to ensure register values are updated correctly

2. Implement Program Execution in UM.c
    a. Implementation: Load instructions into memory and execute them sequentially.
    b. Testing:
        i. Create simple .um programs containing multiple operations (e.g. add, mult, halt, output) and verify the correct sequence of operations and final state by printing out portions of the instruction (opcode, register A, B & C) and register.

Phase 6: Full Program Testing

Tests applicable to all instructions and/or the entire program:

1. Using the testing framework provided in umlab.c, create different combinations of operations in separate .um programs, including single-operation tests (e.g. add.um) and all_ops .um which covers all operations; diff output.txt against expected output
2. Run valgrind on valid inputs and ensure all heap-allocated memory is deallocated if the program finishes executing without failing.
3. Stress test program counter functional correctness by repeatedly calling Halt between other arithmetic and memory operations; use assertions to check that program counter stops where expected
4. Program should EXIT_FAILURE if called with incorrect command line arguments, e.g. extra command line arguments
5. An opcode extracted from an instruction is not well-defined (ie. opcode > 13). Expected behavior: CRE (spec 3.7 Resource Exhaustion)
6. Heap memory allocation fails when user requests duplicating or mapping a new segment; expected behavior: CRE (spec 3.7 Resource Exhaustion)

Segments.h/.c Testing Functions and Contracts

```
/********** test_map_segment ********
 *
 * Tests the map_segment function for proper segment allocation and
 * initialization.
 *
 * Parameters:
 *  Segments_T segments: The segment handler.
 *  uint32_t length: The length of the segment to be allocated.
 *
 * Return:
 *  Test result status (boolean).
 *
 * Expects:
 *  The allocated segment should be initialized to 0 and have the
 *  specified length.
 *
 * Notes:
 *  Verifies that a new segment is correctly created and initialized.
 *
 ****************************************/
```

bool **test_map_segment**(Segments_T segments, uint32_t length)

```
/********** test_unmap_segment ********
 *
 * Tests the unmap_segment function for correct deallocation and ID
 * recycling.
 *
 * Parameters:
 *  Segments_T segments: The segment handler.
 *  uint32_t seg_ID: The identifier of the segment to be deallocated.
 *
 * Return:
 *  Test result status (boolean).
 *
 * Expects:
 *  The specified segment should be properly deallocated and its ID
 *  recycled.
 *
 * Notes:
 *  Ensures that the unmap operation correctly frees memory and
 *  recycles the segment ID.
 *
```

```
 ****************************************/
```

bool **test_unmap_segment**(Segments_T segments, uint32_t seg_ID);

```
/********** test_get_set_word ********
 *
 * Tests the get_word and set_word functions for correct data
 * retrieval and modification.
 *
 * Parameters:
 *  Segments_T segments: The segment handler.
 *  uint32_t seg_ID: The identifier of the segment.
 *  int offset: The position within the segment.
 *  uint32_t value: The value to be set.
 *
 * Return:
 *  Test result status (boolean).
 *
 * Expects:
 *  set_word should correctly modify the specified location.
 *  get_word should retrieve the value set by set_word.
 *
 * Notes:
 *  Confirms the integrity of data storage and retrieval in segments.
 *
 ****************************************/
```

bool **test_get_set_word**(Segments_T segments, uint32_t seg_ID, int offset, uint32_t value);

```
/********** test_duplicate ********
 *
 * Tests the duplicate function for correct segment duplication.
 *
 * Parameters:
 *  Segments_T segments: The segment handler.
 *  uint32_t source_ID: Source segment identifier.
 *  uint32_t destination_ID: Destination segment identifier.
 *
 * Return:
 *  Test result status (boolean).
 *
 * Expects:
 *  The destination segment should be an exact copy of the source
 *  segment.
 *
```

```
* Notes:
*  Ensures that the segment duplication process works correctly.
*
****************************************/
```

bool **test_duplicate**(Segments_T segments, uint32_t source_ID, uint32_t destination_ID);