**Name**: Xiaoyan Xie (xxie05) & Rene Zhao (nzhao05)
**Assignment**: CS40 HW6 um Design
**Date**: Nov 13, 2024

# Architecture

The architecture of Um is designed with three modules: driver module, instructions module and memory module, which have distinct functionalities. The driver module oversees file I/O, manages program initialization, and calls the instruction driver for execution. The memory module abstracts memory management, including mapped and unmapped segments, through the use of Hanson Sequence and Hanson Unboxed Array. This can ensure the efficient and safe access and modification of memory segments. The instructions module decodes and executes the instructions, maintaining program flow by interacting with the memory module. The details for each module and their interaction are shown as below.

<table>
<tr><th>Driver</th><th colspan="3">Instructions</th><th>Memory</th></tr>
<tr><td rowspan="2">File I/O</td><td>⬅️Output</td><td>Conditional Move</td><td>Segmented Load ➡️</td><td rowspan="6">struct Memory {<br>Seq_T segments;<br><br>Seq_T unmapped_identifiers;<br>}</td></tr>
<tr><td>⬅️Input</td><td>Addition</td><td>Segmented Store ➡️</td></tr>
<tr><td></td><td></td><td>Multiplication</td><td>Map Segment ➡️</td></tr>
<tr><td></td><td></td><td>Division</td><td>Unmap Segment ➡️</td></tr>
<tr><td></td><td></td><td>Bitwise NAND</td><td>Load Program ➡️</td></tr>
<tr><td></td><td></td><td>Halt</td><td></td><td></td></tr>
<tr><td></td><td></td><td>Load Value</td><td></td><td></td></tr>
</table>

## Module 1: Memory module

- Where segments are stored in a struct
  - 0 segment
  - Other segments (initially unmapped)
  - Use Hanson Sequence as the wrapper for segments
  - Use Hanson Uarray as the segment which is pointed to by pointers stored in wrapper sequence
  - Each element in the Uarray stands for a 32-bit word

    - Use Hanson Sequence to store identifier for unmapped segments
    - Use struct as a wrapper structure for segment sequence and unmapped segment sequence in order to control the access to memory segment via helper functions
- Contains functions that handle memory. These functions are supposed to be low-level and are not used to directly perform any instructions.
    - Initialization functions at the beginning of program
        - A main initialization function that calls other helper functions.
        - Initialize an empty Hanson Uarray of segments based on size of file
        - Initialize an empty Hanson Sequence to store identifier for unmapped segment
        - Initialize Hanson Sequence as the wrapper for segments
            - Initialize mapped segment 0 which contains words inside file
    - Other memory operations that are used for performing instructions
        - Map an unmapped segment
        - Create a segment
        - Get the size of a segment
        - Access a segment given its identifier
        - Copy a certain segment to segment 0
        - Unmapping and freeing a mapped segment after storing its id
    - Functions that frees memory at the end of program
        - Free Hanson Uarray of segments
        - Free Hanson Sequence of identifier
        - Free memory for Memory struct

<table>
<tr><th colspan="3">Extern Functions</th></tr>
<tr><th>Function Name</th><th>Contract</th><th>Interactions</th></tr>
<tr><td rowspan="4">init_memory</td><td>Purpose: Sets up the Memory struct and segment 0 with program data.</td><td rowspan="4">• Called by Driver module at beginning of program<br><br>• Calls helper function to initialize Uarray based on size of file<br>• Calls helper function to initialize Seq_T of identifier<br>• Calls helper function to initialize mapped segment 0</td></tr>
<tr><td>Parameters:<br>• st_size file_size: size of file<br>• uint32_t Num_segments: number of segments<br>• uint32_t instruction_size: size of one element in uarray</td></tr>
<tr><td>Return: Reference to the memory struct</td></tr>
<tr><td>Expects:<br>• Seq_T segments and Seq_T unmapped_identifiers are properly initialized</td></tr>
<tr><td>seg_new</td><td>Purpose: Creates a new segment of</td><td>• Called by Instructions</td></tr>
</table>

| | | |
|---|---|---|
| | memory, assigns it an identifier, and stores identifier in Memory struct.<br><br>**Parameters**:<br>● uint32_t seg_size: the number of words in new segment<br>● int instruction_size: size of an element in uarray which stores instructions<br>● Memory struct<br><br>**Return**:<br>● uint_32: the identifier of new segment<br><br>**Expects**:<br>● Number of words in segment is greater or equal to 0<br>● Memory struct is not NULL | module for Map Segment.<br>● Called by init_memory when initializing mapped segment 0 at beginning of program |
| seg_free | **Purpose**: Frees a segment and makes its identifier available for reuse.<br><br>**Parameters**:<br>● uint_32: the identifier of freed segment<br>● Memory struct<br><br>**Return**: None<br><br>**Expects**:<br>● The provided identifier refers to a mapped segment<br>● Memory struct is not NULL<br>● The segment to be freed is not NULL | ● Called by Instructions module for Unmap Segment<br>● Called by seg_copy to delete the original segment 0, which will be replaced by new segment<br>● Called by free_memory to free remaining mapped segments when program ends |
| seg_at | **Purpose**: Accesses a word in a segment based on its identifier and offset.<br><br>**Parameters**:<br>● uint32_t id: the identifier of accessed segment<br>● uint32_t offset: the index of accessed word in segment<br>● Memory struct<br><br>**Return**:<br>● uint_32*: pointer to accessed word | ● Called by Instructions module for Segmented Load and Segmented Store |

| | | |
|---|---|---|
| | **Expects**:<br>● Provided identifier refers to an existing segment<br>● Provided offset does not exceed length of segment<br>● Memory struct is not NULL | |
| seg_copy | **Purpose**: Copies a segment into Segment 0.<br>**Parameters**:<br>● uint32_t id: the identifier of copied segment<br>● Memory struct<br>**Return**: None<br>**Expects**:<br>● Provided identifier refers to an existing segment<br>● Original segment 0 is properly freed<br>● Memory struct is not NULL | ● Called by Instructions module for Load Program<br>● Calls seg_unmap to free original segment 0, if applicable |
| seg_store | **Purpose:** Stores a certain word in to specified segment.<br>**Parameters:**<br>● uint32_t id: the identifier<br>● uint32_t offset: the index of accessed word in segment<br>● int word<br>● Memory struct<br>**Return**: None<br>**Expects**:<br>● Provided identifier refers to an existing segment<br>● Provided offset does not exceed length of segment<br>● Memory struct is not NULL | ● Called by Instructions module for Segmented Store |
| seg_size | **Purpose**: Gets the size of a segment.<br>**Parameters**:<br>● uint_32t id: the identifier of accessed segment<br>● Memory struct<br>**Return**: | ● Called by Instructions module for initializing program counter and bound checking instructions |

| | | |
|---|---|---|
| | • int: number of words inside the segment | |
| | **Expects:**<br>• Memory struct is not NULL<br>• Uarray storing words is not NULL | |
| free_memory | **Purpose**: Frees all segments and allocated memory, free all the sequences, free the struct | • Called by Driver module at the end of program<br><br>• Calls seg_unmap to free all mapped segments including segment 0<br>• Calls helper function to free Uarray of segments<br>• Calls helper function to free Seq_T of identifier |
| | **Parameters**:<br>• Memory struct | |
| | **Return**: None | |
| | **Expects**:<br>• All allocated memory are freed and there's no memory leaks<br>• There is no double-freeing | |

## Module 2: Instructions module

- Where registers are stored
- Where program counter is stored
- Contains functions for each instruction
- Contains instruction driver to call related helper functions according to opcodes extracted from a certain word

- Functions needed:
    - Static functions that do not interact with other modules
        - Instructions such as conditional move, addition, multiplication, division, bitwise NAND, output, input, load value
        - Get opcode
        - Get register
    - Functions that do interact with other modules
        - Segment load, segment store, map segment, unmap segment, load program

| **Extern Functions** | | |
|---|---|---|
| **Function Name** | **Contract** | **Interactions** |
| init_instructions | **Purpose**: Initializes the Instructions | • Called by Driver |

| | | |
|---|---|---|
| | struct and sets up register values and the program counter.<br>**Parameters**: None<br>**Return**: None<br>**Expects**: None | module at beginning of program<br>● Calls helper functions to initialize a Uarray of register values |
| instruction_driver | **Purpose**: Executes each word in segment 0 by decoding instructions and incrementing the program counter.<br>**Parameters**:<br>● Memory struct<br>● uint32_t arr_registers<br>**Return**: None<br>**Expects**:<br>● The identifier for segment 0 is expected to be 0<br>● Segment 0 is properly initialized and mapped<br>● Opcode ranges from 0 to 13 | ● Called by Driver module at beginning of program, after Memory and Instructions are initialized<br>● Calls move_counter every time an instruction is performed<br>● Calls get_opcode<br>● Calls get_registers<br>● Calls functions corresponding to opcode, and sometimes pass in register value as arguments |
| segmented_load | **Purpose**: Sets a register to a word at the provided segment identifier and index.<br>**Parameters**:<br>● uint32_t id: the identifier of accessed segment<br>● uint32_t offset: the index of accessed word in segment<br>● Memory struct<br>**Return**: None<br>**Expects**:<br>● Provided identifier refers to an existing and mapped segment<br>● Provided offset does not exceed length of segment<br>● New register value properly replaced original value | ● Calls seg_at in Memory module<br>● Calls seg_store in Memory module |
| segmented_store | **Purpose**: Sets the word at the | ● Calls seg_store in |

| | | |
|---|---|---|
| | provided segment identifier and index to a value at a register. | Memory module |
| | **Parameters**:<br>● uint32_t id: the identifier of accessed segment<br>● uint32_t offset: the index of accessed word in segment<br>● Uint32_t word: the word that needs to store<br>● Memory struct | |
| | **Return**: None | |
| | **Expects**:<br>● Provided identifier refers to an existing and mapped segment<br>● Provided offset does not exceed length of segment<br>● Original word is properly replaced by new value in register | |
| map_segment | **Purpose**: Maps a new segment of memory with a specified size and assigns a unique identifier. | ● Calls seg_new in Memory module |
| | **Parameters**:<br>● uint32_t size: the number of words in the mapped segment<br>● uint32_t registerC<br>● Memory struct | |
| | **Return**: None | |
| | **Expects**:<br>● The provided identifier refers to a unmapped segment | |
| unmap_segment | **Purpose**: Unmaps a mapped segment and frees its memory. | ● Calls seg_unmap in Memory module |
| | **Parameters**:<br>● uint32_t id: the identifier of unmapped segment<br>● Memory struct | |
| | **Return**: None | |

| | | |
|---|---|---|
| | **Expects**:<br>● The provided identifier refers to a existing and mapped segment<br>● The unmapped segment is not segment 0 | |
| input | **Purpose**: Reads in a value from standard input and stores it in a register.<br>**Parameters**: None<br>**Return**: None<br>**Expects**:<br>● Value ranges from 0 to 255<br>● When input ends with a signal, all bits in the register are loaded as 1 | ● Part of I/O in Driver module<br>● Calls load_value to store value from standard input to register C |
| output | **Purpose**: Prints the value from a register to standard output.<br>**Parameters**:<br>● int32_t registerC<br>**Return**: None<br>**Expects**:<br>● Value ranges from 0 to 255 | ● Part of I/O in Driver module<br>● Calls get_register to get value to be printed from register C |
| load_program | **Purpose**: Replaces segment 0 with a specified segment to load a new program. Sets program counter to a specific word in that segment<br>**Parameters**:<br>● uint32_t id: the identifier of new segment<br>● uint32_t offset: the index of word to start execution<br>● Memory struct<br>● uint32_t counter: program counter<br>**Return**: None<br>**Expects**:<br>● Provided identifier refers to an existing and mapped | ● Called by Driver module at the beginning of program<br>● Calls seg_copy in Memory module<br>● Modifies program counter using set_counter |

| | | |
|---|---|---|
| | segment<br>● Original segment 0 is properly freed | |

| **Static Functions** | |
|---|---|
| **Function Name** | **Description** |
| load_value | Set a register to a given value. |
| get_opcode | Gets the opcode from a given word. |
| get_register | Gets a register value (provided which register we want to access) from a given word. |
| conditional_move | Moves the value from one register to another if a condition is met (register C is not equal to 0). |
| addition | Adds values from two registers and stores the result in a third register. |
| multiplication | Multiplies values from two registers and stores the result in a third register. |
| division | Divides the value in one register by another and stores the result in a third register. |
| bitwise_nand | Computes the bitwise NAND of two registers and stores the result in a third register. |
| halt | Stops program execution immediately. |
| set_counter | Initializes the program counter at the beginning of the program, or set counter to a specific word in segment 0. |
| move_counter | Increments the program counter after each instruction. |

## Module 3: Driver module

- Contains the main function
- Handles I/O
- Entrance of emulator
- Read the file and get the number of instructions by using st_size
- Contain a driver function to initialize the memory struct and call instruction driver to execute instructions

- Functions needed:

- Static functions that do not interact with other modules, such as main function that
    - Count the number of instructions
    - Read file
- Functions that do interact with other modules such as emulator driver that
    - Create memory struct
    - Store instructions in segment 0
    - Call instruction driver to process instructions
    - A program counter that goes through each instruction

| **Static Functions** | |
|---|---|
| **Function Name** | **Description** |
| main | Entry point of the emulator; initializes memory; call instruction; frees memory at program end. |
| read_and_open_file | Opens and reads the program file; determines file size to calculate the number of instructions. |

# Implementation Plan

1. Create a main function that ends with exit(0)

2. Create read file function
    - Test: Using test_read_file (shown below) to verify that files open correctly and file sizes are properly calculated as expected.

3. Create Memory module
    a. Initialize Memory struct which contains a sequence of unmapped identifiers and a sequence of segments
        - Test: Using void test_init_memory to make sure that the memory struct is successfully initialized as expected.
    b. Create freeing function to free all the memory
        - Test: Using void test_free_memory(shown below) to check if all the segments, unmapped IDs and memory struct are freed as expected
    c. Implement seg_size function to return the size of a segment
        - Test: Using test_seg_size to test if size of a segment is successfully returned as expected

d. Implement seg_new for creating new segments of a given size.
    - Test: Using void test_seg_new() to test if a new segment is created as expected with a specific ID.
e. Implement seg_free function to free a certain segment and recycle id
    - Test: Using test_seg_free() to check if the segment is freed and id is recycled and stored in identifier sequence
f. Implement seg_store to store values at specific indices.
    - Test: Using test_seg_store() to check if information is stored correctly in correct place as expected
g. Implement seg_at to retrieve data within segments.
    - Test: Using test_seg_at() to check if information in specified segment is returned correctly as expected
h. Implement seg_copy to copy a certain segment to segment 0 for program loading
    - Test: Using test_seg_copy() to test if the information in a certain segment is copied correctly as expected

4. Create Instructions module
    a. Initializes registers and program counter for emulator.
        - Test: Using void test_init_instructions() to verify that init_instructions successfully set all registers and initialize the program counter to 0 as expected.

5. Create instructions driver
    a. Extract information from a certain word, executes instructions with a loop
    b. Create static helper functions introduced in the instructions module.
        - Test: listed in test plan part.
    c. Create segemented_load function
        - Test: Using void test_segement_load() to test if the function retrieves the correct value from the specified segment and offset as expected.
    d. Create segmented_store function
        - Test: Using void test_segement_store() to test if the function stores correct value in correct place.
    e. Create map_segment function
        - Test: Using void test_map_segment() to test if a new segment is created and returns correct ID as expected.
    f. Create unmap_segment function
        - Test: Using void test_unmap_segment() to check if a certain segment is freed and its ID is recycled as expected.
    g. Create load_program function

- Test: Using void test_load_program() to check if the specified segment is correctly copied to segment 0 and sets program counter to 0.

| Step | Test Function Prototype | Description |
|---|---|---|
| 2 | void test_read_file(FILE* input); | Calls read file function, checks if the returned FILE pointer is not NULL, and checks if file size is correctly calculated. |
| 3.a | void test_init_memory(st_size file_size, uint32_t Num_segments, uint32_t instruction_size); | Calls init_memory function, checks if the pointer to struct and sequences are not NULL. |
| 3.b | void test_free_memory(Memory memory) | Calls init_memory and free_memory function, check if it frees all allocated memory in Memory struct, including segments and struct. |
| 3.c | void test_seg_size() | Calls init_memory and seg_size function, confirms the size of a segment is accurately returned |
| 3.d | void test_seg_new() | Calls init_memory and seg_new function, adds a segment with 0s, can create/reuse ID and return the correct ID; can handle CRE as expected |
| 3.e | void test_seg_free() | Calls init_memory, seg_new and seg_free, free_memory functions, releases a specific segment, add its ID to sequence for unmapped identifiers |
| 3.f | void test_seg_store() | Calls init_memory, seg_new and seg_store function, seg_free, free_memory functions, confirms correct information is stored at a specific segment |
| 3.g | void test_seg_at() | Calls init_memory, seg_new seg_at and seg_at and free functions, check if the information in a specified segment is returned correctly |
| 3.h | void test_seg_copy() | Calls init_memory, seg_new, seg_at, seg_copy and free functions to make sure that specified segment is copied. |

| | | |
|---|---|---|
| 4.a | void test_init_instructions() | Calls init_instructions function, verifies that init_instructions successfully set all registers and initialize the program counter to 0 |
| 5.a | void test_instruction_driver() | Calls init_instructions and instruction_driver functions, ensures the diver function correctly fetches and executes instructions, updates program counter and modifies registers as expected. |
| 5.c | void test_segmented_load() | Calls init_memory(), seg_new(), seg_store, free_memory(), init_instructions(), instruction_driver() and segmented_load function, verifies if segmented_load retrieves the correct value from the specified segment and offset. |
| 5.d | void test_segmented_store() | Using test_segmented_load() function and add segmented_store function call, check if the correct value is stored in correct segment with specified index |
| 5.e | void test_map_segment() | Using test_segmented_store() function and add map_segment function call,check if a segment with specified size is created and returns correct segment ID. |
| 5.f | void test_unmap_segment() | Using test_segmented_store() function and add unmap_segment function call,check if a specified segment is freed and segment ID is recycled. |
| 5.g | void test_load_program() | Using test_map_segment() function and add load_program function call to check if it copies a specified segment to segment 0 and program counter is 0. |

# Testing Plan

1. The UM instruction set (i.e., does the “Add” instruction get disassembled and executed properly) tests are listed below. The test for abstraction contains two parts:

during implementation, for each step the unit test will be run along with each step, which is shown in above implementation plan. After finishing a module, comprehensive tests will be performed to test the functionality of the whole module.

Testing each instruction along implementation:

| **Unit Test for** | **Instructions Set** |
|---|---|
| Halt | Halt |
| Output | Output, Halt |
| Load Value | Load Value, Output, Halt |
| Conditional Move | Load Value, Conditional Move, Output, Halt |
| Addition | Load Value, Addition, Output, Halt |
| Multiplication | Load Value, Multiplication, Output, Halt |
| Division | Load Value, Division, Output, Halt |
| Bitwise NAND | Load Value, Bitwise NAND, Output, Halt |
| Input | Input, Output, Halt |
| Map Segment | Load Value, Map Segment, Halt |
| Unmap Segment | Load Value, Map Segment, Unmap Segment, Halt |
| Segmented Load | Load Value, Map Segment, Segmented Load, Halt |
| Segmented Store | Load Value, Map Segment, Segmented Store, Halt |
| Load Program | Load Value, Map Segment, Load Program, Halt |

2. Testing after the modules are completed:
    - Test mixed instructions
        - Use self-created instruction set with mixed instructions with void test_self_created_instruction_set();
        - A set of instructions with various operations (e.g.,Load Value, Addition, Segmented Load, Output, Halt) to verify instruction interaction within the module.

    - Tests instructions with boundary values (e.g., maximum and minimum register values) to verify they handle overflow, underflow, and boundary limits.
- Test CPU time
    - Use spec-provided instruction set midmark and test with void test_midmark_instruction_set();
    - Use spec-provided command cpu-limited 10 ./um midmark.um