Anne Lau (alau03), Daniel Chan Lee (dlee07)
Homework 6
Design Doc

<u>Design Checklist for Universal Machine</u>

Architecture

- The virtual machine will consist of a few main components:
    - Registers -- An array of 8 uint32_t elements
    - Memory -- Globally declared sequence and an array of pointers to elements in the array
    - Program Counter -- Globally declared pointer to a 32-bit word that will live in the UM
- We will break it down into 3 main interfaces, the um.c interface, that will call functions from the operation.c interface and will occasionally alter the memory.c interface.

What are the major components of your program, and what are their interfaces? Components include functions as well as abstract data types. An interface includes contracts as well as function prototypes.

Our program will have 3 main interfaces:

1. um.c interface
    a. Variables that exist are initialised in this interface:
        i. Pointer Counter
        ii. Memory
    b. This will call run_um which will do the following steps:
        i. Will call read_input ( ) {will read all the input in}
        ii. Will call start_prog ( ) {initialise memory and set the program counter}
        iii. Will call a while loop that calls move_pcount( ) and calls read_op( ) {read_op( ) will be called from the operation.c interface}
        iv. Will call read_out ( ) {read the um input out}
2. operation.c interface
    a. This interface will hold the read_op( ) function that will be a switch statement that will go the respective instruction
    b. This interface will also hold all 14 instructions
3. memory.c interface
    a. This interface will initialise memory and change memory segments accordingly

Um.c interface

// Functions that are called to start the program

- uint32_t *start_prog (FILE *input)
    - Does: calls intial_mem(FILE *input), and initial_pcount(Seq_T mem), which will set the program counter to the beginning of the list of instructions in m[0]
    - Parameters: Takes in the file input, which will be read into the initial_mem
    - Returns: uint32_t pointer to the beginning of the program
    - Error: If FILE does not open

// I/O Device interface

- void write_data (uint32_t) - accesses the virtual memory data structure and calls convert_ASCII.
    - Does: Writes the uint32_t value out to stdout, after calling convert_ASCII
    - Parameters: the uint32_t that is going to be written out
    - Returns: void
    - Error: None

- uint32_t input_data ()
    - Does: Takes value from the I/O and returns a uint32_t that is loaded into the register
    - Parameters: None
    - Returns: Returns the uint32_t value is taken in from the I/O stream
    - Error: If the number is not within the range of 0-255

//Program Counter functions:
The program counter will be a pointer to a uint32_t word and will be initialized in this interface.

- uint32_t * intial_pcount (Seq_T mem)
    - Does: Initialises a pointer to the start of the sequence held in the m[0] of the Sequence in memory
    - Parameters: Seq_T that stores the memory
    - Returns: uint32_t * to the beginning of the sequence
    - Errors: If the memory is NULL
- void move_pcount (uint32 * pcount, Seq_T instructions)
    - Does: Moves the program counter to the next uint32_t of the Seq_T “instructions” in m[0]
    - Parameters: uint32_t * program counter and the Seq_T of the instructions in m[0]
    - Returns: None
    - Errors: If Seq_T is empty

Operation.c interface
The register will represent the register of our UM machine and hold the values that we will use as we move in the program. Our register interface will be represented by a Hansen Sequence of size 8 of uint32_t values.

- Seq_T initial_registers ();
    - Does: Initialises the Seq_T of 8 of uint32_t words
    - Parameters: None
    - Returns: Hansen’s Seq_T of uint32_t words
    - Error: None

- void Read_op (uint32_t)
    - Does: Executes a switch statement that calls a switch statement on the respective op_code and executes the respective function on the three registers passed in
    - Parameters: the uint32_t instruction that the program counter points to
    - Returns: void
    - Error: If the uint32_t instruction is not valid

- Void free_register (Seq_T registers)
    - Does: Frees the registers
    - Parameters: the Seq_T of the register
    - Returns: None
    - Error: None

// Operator functions
- void cond_move (uint32_t *a, uint32_t *b, uint32_t *c)
    - Does: Conditional Move if $r[C]≠ 0 then $r[A] := $r[B]
    - Parameters: The three register values
    - Returns: None

    - Error: None

- Void seg_load (uint32_t *a, uint32_t *b, uint32_t *c)
    - Does: $r[A] := $m[$r[B]][$r[C]], retrieves the information from memory's function mem_access( )
    - Parameters: The three register values
    - Returns: None
    - Error: None

- Void seg_store (uint32_t *a, uint32_t *b, uint32_t *c)
    - Does: $m[$r[A]][$r[B]] := $r[C], retrieves the information from memory's function mem_access( )
    - Parameters: The three register values
    - Returns: None
    - Error: None

- void add (uint32_t *a, uint32_t *b, uint32_t *c)
    - Does: Add, $r[A] := ($r[B] + $r[C]) mod 232
    - Parameters: The register array, the three register values
    - Returns: None
    - Error: None

- void mult (uint32_t *a, uint32_t *b, uint32_t *c)
    - Does: Multiplies, $r[A] := ($r[B] × $r[C]) mod 232
    - Parameters: The register array, the three register values
    - Returns: None
    - Error: None

- void div (uint32_t *a, uint32_t *b, uint32_t *c)
    - Does: Division, $r[A] := ⌊$r[B] ÷ $r[C]⌋
    - Parameters: The register array, the three register values
    - Returns: None
    - Error: None

- void bit_and (uint32_t *a, uint32_t *b, uint32_t *c)
    - Does: Bitwise NAND, $r[A] := ¬($r[B] ∧ $r[C])
    - Parameters: The register array, the three register values
    - Returns: None
    - Error: None

- Void halt ()
    - Does: Halts the program
    - Parameters: None
    - Returns: None
    - Error: None

- Void map_seg (uint32_t *b, uint32_t *c)
    - Does: Calls the get_id and the new_seg functions from the memory interface according to the registers
    - Parameters: the two registers

- Returns: None
    - Error: None
-
- Void unmap_seg (uint32_t *b, uint32_t *c)
    - Does: Calls the free_seg functions from the memory interface according to the registers
    - Parameters: the two registers
    - Returns: None
    - Error: None

- Void output (uint32_t *c)
    - Does: writes the output out to the I/O stream from register C
    - Parameters: the last register
    - Returns: None
    - Error: Checks the ASCII value

- Void input (uint32_t *c)
    - Does: Takes in the input from the I/O steam and puts it into register C
    - Parameters: the last register
    - Returns: None
    - Error: None

- Void load_prog (uint32_t *b, uint32_t *c)
    - Does: Segment $m[$r[B]] is duplicated, and the duplicate replaces $m[0], which is abandoned. The program counter is set to point to $m[0][$r[C]].
    - Parameters: the two registers
    - Returns: None
    - Error: None

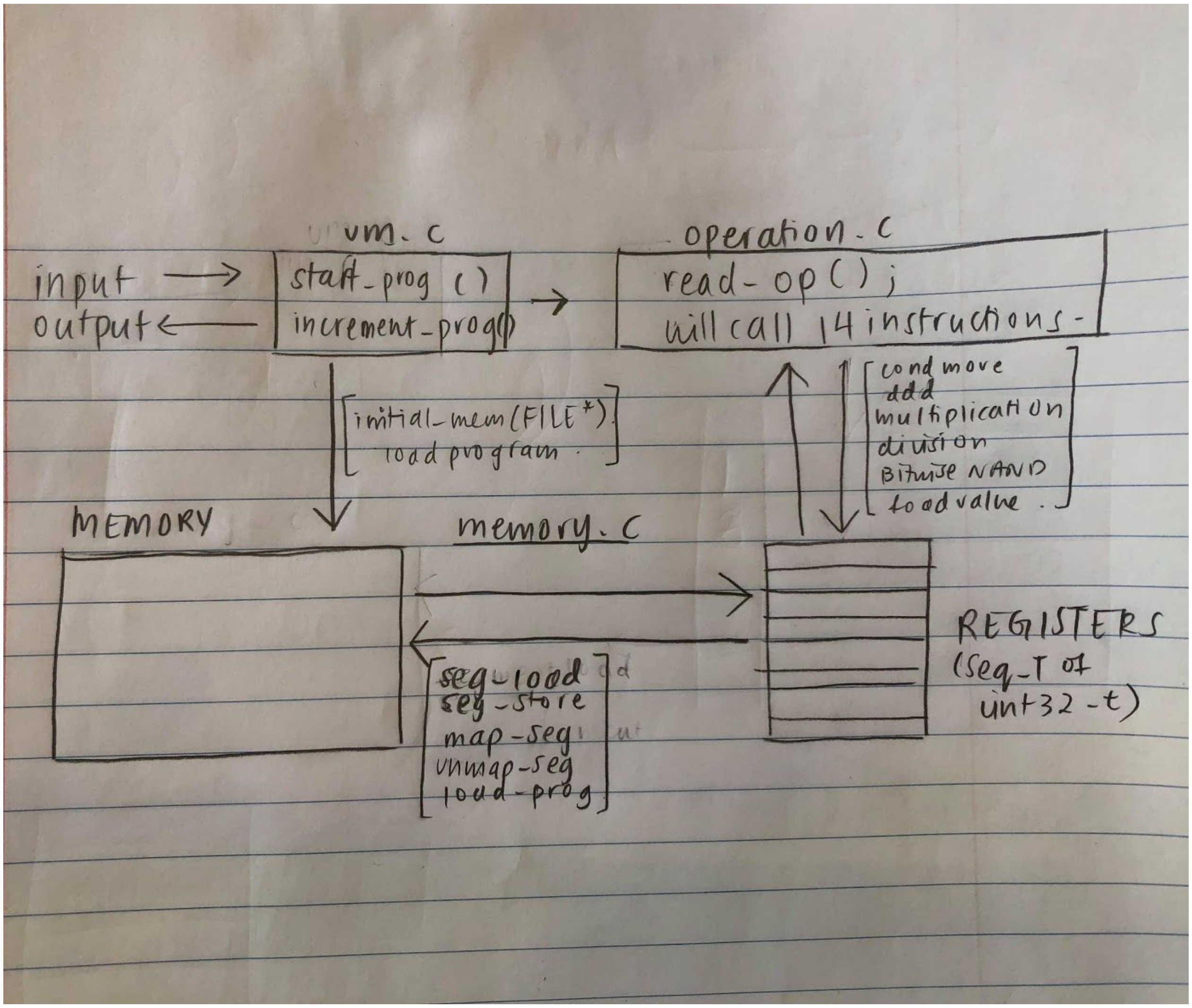
- void load_value (UArray_T registers, uint32_t a, uint32_t value)
    - Does: Sets a to the value of the remaining 25 bits that are an unsigned binary value
    - Parameters: The register array, the one register and the value
    - Returns: None
    - Error: If the Um_register is not within the range of 0-7

// Check the ASCII values for output and input operator
- unsigned convert_ASCII (uint32_t a)
    - Does: converts a value at a register in ASCII, checks if it fits within 0-255
    - Parameters: The uint32_t value that is being outputted
    - Returns: The ASCII value that is correct
    - Error: If the number is not within the range of 0 - 255
- uint32_t convert_uint32 (unsigned a)
    - Does: converts the unsigned value to a uint32_t value, if it is not within the range of 0 - 255, halt
    - Parameters: The unsigned value
    - Returns: The uint32_t value
    - Error: If the number is not within the range of 0 - 255

Memory interface

- Detailed Below

How do the components in your program interact? That is, what is the architecture of your program?



Test Cases

Operator	**Interface used**	**Input**	**Expected Output**	**Exceptions**
Conditional Move	Operation.h	r[a] = 3 r[b] = 5 r[c] = 4	**r[a] = 5** r[b] = 5 r[c] = 4	If r[c] = 0
		r[a] = 3 r[b] = 5 r[c] = 0	**r[a] = 3** r[b] = 5 r[c] = 0	
Segmented Load	Operation.h Memory.h	r[a] = 3 r[b] = 4	**r[a] = $m[4][4]** r[b] = 4	

		r[c] = 4	r[c] = 4	
Segmented Store	Operation.h Memory.h	r[a] = 3 r[b] = 4 r[c] = 4	r[a] = 3 r[b] = 4 **r[c] = $m[3][4]**	
Addition	Operation.h	r[a] = 3 r[b] = 4 r[c] = 5	**r[a] = 9** r[b] = 4 r[c] = 5	
Multiplication	Operation.h	r[a] = 3 r[b] = 4 r[c] = 5	**r[a] = 20** r[b] = 4 r[c] = 5	
Division	Operation.h	r[a] = 3 r[b] = 4 r[c] = 5	**r[a] = 4/5** r[b] = 4 r[c] = 5	
Bitwise NAND	Operation.h	r[a] = 3 r[b] = 4 r[c] = 5	**r[a] = 5** r[b] = 4 r[c] = 5	
Halt	N/A	N/A	N/A	
Map Segment	Operation.h Memory.h	r[a] = 3 r[b] = 4 r[c] = 5	r[a] = 3 r[b] = ????? r[c] = 5 M[4] = 5 words	A bit pattern that is not all zeroes and that does not identify any currently mapped segment is placed in $r[B].
Unmap Segment	Operation.h Memory.h	r[c] = 5	r[c] = 5 m[5] = unmapped	m[5] can be reused
Output	io.h	r[c] = 5	Stdout << 5;	Only values from 0 to 255 are allowed.
Input	io.h	stdin >> 6	r[c] = 6	Only values from 0 to 255 are allowed.
Load Program	Operation.h Memory.h	r[a] = 3 r[b] = 4 r[c] = 5	m[0] = m[4] Program counter points to m[0][5]	
		r[a] = 3 r[b] = 0 r[c] = 5	The load-program operation is expected to be fast	
Load Value	Register	r[a] = 3 other value = 15	r[a] = 15 other value = 15	Different reading method

UM.c

- The pointer always points to the program in m[0]

Operations.c

- If the I/O interface returns the values specified in the above table.
- Ensure that the ASCII value returned when converted from uint32_t are between 0 and 255.
- If the Register interface returns the values specified in the above table.
- If the Instruction divides by 0, the program should fail.
- Instructions cannot be divided by 0, or else Halt
- Testing method for the Operations.c, we will create uint32_t instructions that will call the 14 different instructions to test them individually and see if it returns the expected output

Memory Interface

If the Memory interface returns the values specified in the above table, and more detailed testing is placed is described below.

Universal machine unit tests -- explanation

- What data structure will be used to represent each part of the state of a UM -- where the data structure will be stored
- How the parts will be organised
- How the implementation of the UM will be decoupled from the program loader
    - Instruction unmaps $m[0], or if it unmaps a segment that is not mapped, the machine will fail
    - If an instruction divides by zero, the machine may fail
    - If an instruction loads a program from a segment that is not mapped, the machine may fail
    - If an instruction outputs a value larger than 255, the machine may fail

Invariants for the Universal Machine

- The elements on the stack always have to refer to unmapped segments
- We know if we are given size of 10, and there are only 5 mapped segments, then there should be 5 segment IDs on stack of segmented IDs. In other words, the size of the stack must be size of our memory structure minus the number of mapped seg_IDs
- When we are adding a uint32_t to a Segment ID that has been mapped, then the stack should not be affected
- Any operations that are called, the Segment IDs that are involved should not be unmapped, other than in the unmap segment operation
- If the segment is unmapped, the sequence representing that segment should point to NULL
- Any inputs and outputs must have a value from 0 to 255, meaning the 24 highest bits must be 0.
- Every time a value is added to its respective ID, and there are no unmapped segments, its seg_id must be the length of the memory sequence + 1
- When a new instruction is read, the program counter should only move by 1 uint32_t word. For every instruction, the program counter should increment by 1.

Decoupling and testing of UM

Both the Operations and Memory interface will have its own test main. For the operations interface, the main will consist of a program that takes in a file of small uint32_t inputs. The first test phase will check if the Opcode, Register A, Register B, and Register C are valid values. The second test phase would consist of 14

test functions that will check that each operation is performing the operation correctly. The third test will be more sophisticated as it will call a function that contains a series of operations.
For the Memory interface, we will first run the basic test cases provided under the description of the memory interface.. Because the test cases for the Memory interface provided below are to test exception catching, we will couple this interface with the operations interface. The reason for this is that we want to test that values are being mapped and unmapped efficiently and correctly. The level in complexity of inputs will increase by increasing the amount of inputs in a file that would be passed into the test main of the operations interface.

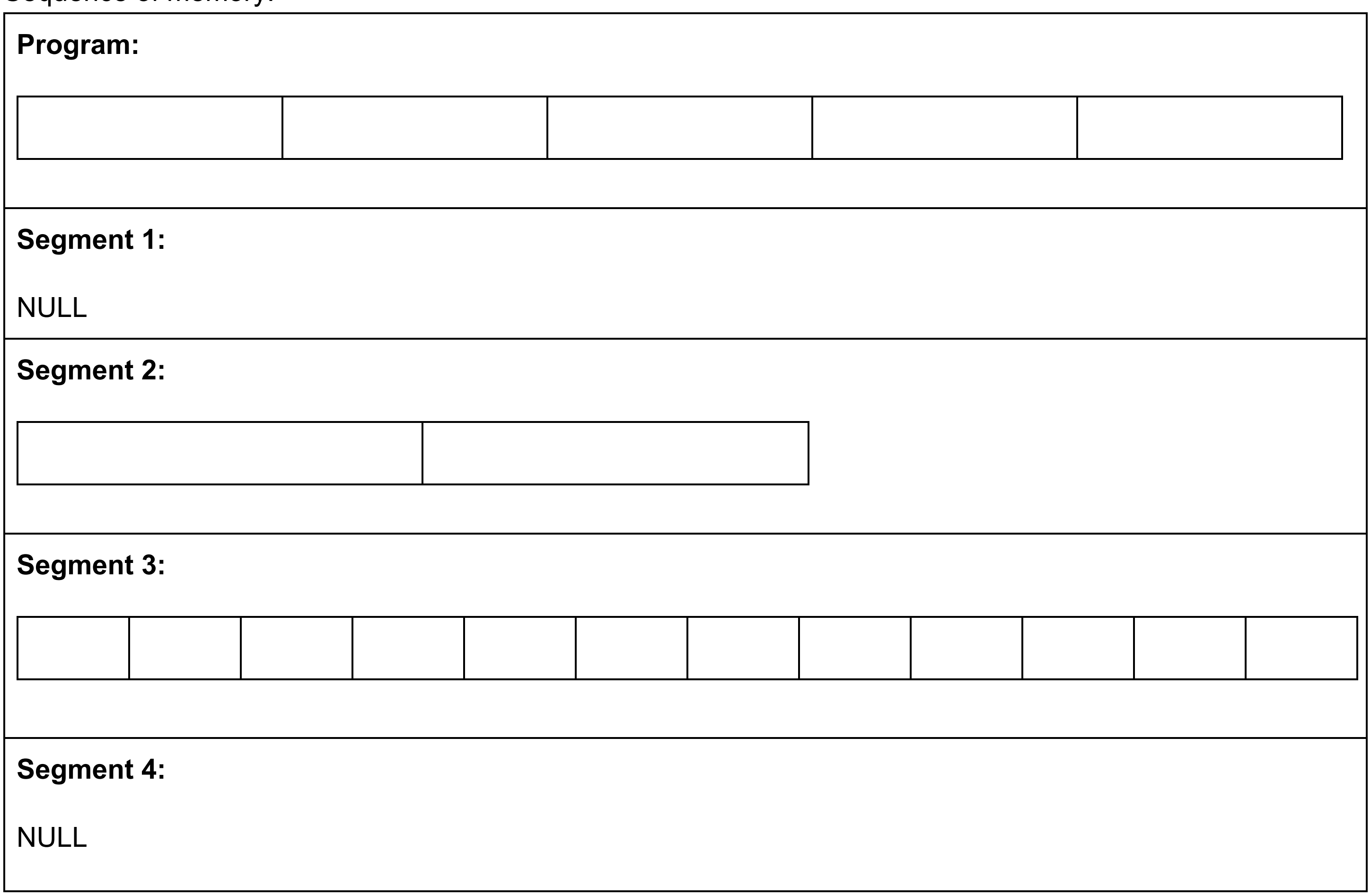
<u>Design Checklist for Universal Segments</u>
What are the major components of your program, and what are their interfaces? Components include functions as well as abstract data types. An interface includes contracts as well as function prototypes.

- We will use a Hansen implemented Seq of Seq of 32-bit words to represent the memory
    - The mapped/unmapped status of the segments will be checked if the Sequence points to NULL.
- We will use a Hansen Seq to represent a stack that will hold all unmapped segments of unsigned
    - Before we map a segment, we will check this stack for the segments that are unmapped, if none, map a new segment

Stack of unsigned representing the unmapped segments:

1
4

Sequence of memory:



How do the components in your program interact? That is, what is the architecture of your program?

1. The m[0] of the Sequence will hold the program
2. The program counter will be initialized and will point to the start of the first instruction of the m[0]
3. When a new segment is mapped, we check the Stack for unmapped segments and if there are unmapped segments, set the new segment to that segment identifier
4. When a segment is unmapped, we set the Sequence to NULL and add it to the Stack that holds the unmapped segments

Memory Segment interface:

- Seq_T initial_mem (FILE *input)
    - Does: Initializes a sequence of sequences with values from input. This data structure will be our memory storage structure.
    - Parameters: a FILE pointer
    - Returns: a pointer to the Sequence of Sequences
    - Errors: N/A
- Seq_T initial_stack ()
    - Does: initializes a stack that will keep in track of unmapped IDs
    - Parameters: none
    - Returns: returns the stack of unmapped ids
    - Errors: N/A
- void free_mem (T mem);
    - Does: frees the sequences of sequences
    - Parameters: Sequence of Sequences
    - Returns: none
    - Errors: none
- Void free_stack (Seq_T stack)
    - Does: frees the stack of unmapped sequences
    - Parameters: Sequence of unsigned
    - Returns: none
    - Errors: none
- Unsigned new_seg (Seq_T mem, unsigned num_words) // or we can call it Map_Seg ()
    - Does: takes in a value and creates an segment ID for the value
    - Parameters: Sequence of sequences, value
    - Returns: unsigned
    - Errors: If the unsigned value is mapped to an unmapped segment
- uint32_t seg_access (Seq_T mem, unsigned seg_id, unsigned offset)
    - Does: accesses the memory structure and retrieves the word stored at the seg_id and offset
    - Parameters: Seq_T of memory, segment ID, location within the segment
    - Returns: a uint32_t value
    - Errors: If segnum and offset are out of bounds
- void seg_store (T mem, unsigned segnum, unsigned offset, uint32_t value)
    - Does: stores a value into Memory[segnum][offset]
    - Parameters: Seq_T, Segment ID, location, uint32_t value
    - Returns: none
    - Errors: if segnum and offset are out of bounds
- void free_seg (Seq_T mem, unsigned seg_id, Seq_T stack)

- Does: Set the Seq_T mem at the seg_id to NULL and add the number of the seq_id to the Stack
    - Parameters: The sequence of sequences, ID, sequence of unmapped statements.
    - Returns: none
    - Errors: If seg_id given refers to an unmapped segment in memory
- unsigned get_id (T stack)
    - Does: gets the next segment_id, checks the stack for unmapped segments, if there are previous unmapped segments, return that value, get the Sequence length and increment it to get the next seg_id.
    - Parameters: The stack of unmapped IDs
    - Returns: returns the ID value of an unmapped segment
    - Errors: N/A

Functions called by our test file
- void check_mapped (Seq_T stack, unsigned seg_id)
    - Does: Check that the seg_id does not point to a NULL value
    - Parameters: Sequence of memory, segment id
    - Returns: None
    - Error: If the seg_id is not in the stack
- void check_size (Seq_T mem, unsigned seg_id)
    - Does: Checks that the seg_id is smaller than the size of the Table
    - Parameters: Sequence of memory, segment id
    - Returns: None
    - Error: If the seg_id is bigger than the sequence
- unsigned test_m0stored (Seq_T mem)
    - Does: Checks that segment 0 holds the program
    - Parameters: sequence of sequences
    - Returns: returns Segment ID of where the program counter is
    - Error: if ID is not 0, its not pointing to the program.

What test cases will you use to convince yourself that your program works?

5. Test exception catching
    a. If $m[0] does not store the program, halt
        i. Function: Check_m0stored
    b. If at the beginning of the machine cycle, program counter points outside the bounds of $m[0]
        i. Function: Check_ptr
    c. If the word pointed to by the program counter does not code for a valid instruction, halt
        i. Function: Check_instruction will either check
            1. Value is between 0 to 255 - check_val
            2. Op code is from 0 to 13 - check_op
            3. Check if the location is not out of bounds of segment
            4. Check if the segment has been allocated.
    d. If segmented load or segmented store refers to an unmapped segment, halt
        i. Function: Check_mapped
    e. If a segmented load or a segmented store refers to a location outside the bounds of a mapped segment, Halt
    f. Ensure that the offset is always within the bounds of size of the seg_id
6. Test functionality

a. Ensure that Seg_Store, stored the instructions into the correct area in segment[n]
b. Ensure that Seg_Load, retrieves the instructions from the specified area in segment[n]
c. Ensure that after a segment is unmapped, the seg_id will be mapped to the next new segment
d. Ensure that we never map it to 0