UM-design doc

Authors: Ella Hou (ehou02) and Darius-Stefan Iavorschi (diavor01)

Architecture:

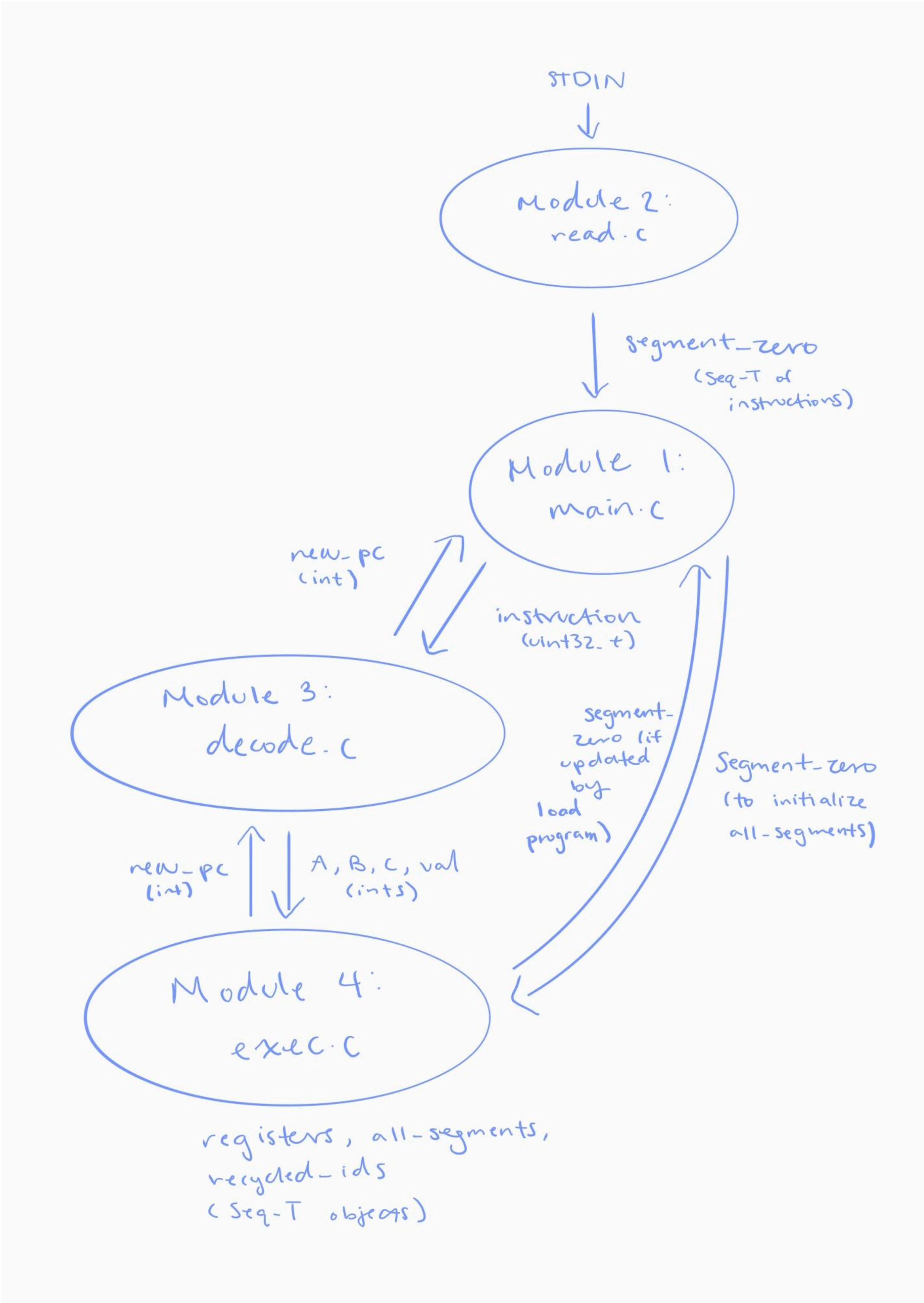
Overview of data structures:

- Storing Segments: We are creating a matrix of instructions using 2 nested Hanson's sequences. The outer sequence is named all_segments. Each row in all_segments is a sequence representing a segment, and each segment contains 32-bit integer instructions.

- Storing recycled IDs: We will also store the ID's of segments no longer used in a Hanson Sequence named recycled_ids.

- Why sequences? Hanson sequences are implemented as circular arrays under the hood, implying that:
    1) They are resizable (since we don't know how many instructions are in a segment or the total number of segments).
    2) There is O(1) extraction time at any index.
    3) There is O(1) insertion time at the end of the sequence (adding new segments or instructions)
    4) There is O(1) removal time at the beginning of the sequence (popping from recycled_ids)

Diagram of Module Interactions:



Module descriptions:

- **Module 1**: (main.c)
    - Purpose: Handles the control flow of the program, updates the program counter
    - Data Structures:
        - Seq_T segment_zero: holds the instructions for segment_zero
    - Interactions with other Modules:
        - Calls store_code() from Module 2 to read in the instructions from the file and initialize segment zero.
        - Calls initialize_program_state(segment_zero) from Module 4 to initialize all_segments (with segment_zero at the beginning), registers, and recycled_ids
        - Passes the current instruction to handle_instruction() in Module 3 to carry out the corresponding function.
        - Receives an int from handle_instruction() in Module 3 about how to update the pc

- **Module 2**: Read the instructions from STDIN (read.h, read.c)
    - Purpose: Read in 32-bit instructions from the file in big endian order, storing them in a Seq_T called segment zero. Return segment zero.
    - Data Structures:
        - Seq_T segment_zero: holds all the 32-bit instructions from the file
    - Interactions with other Modules:
        - Returns segment zero to Module 1 (Main)
            - Caller is responsible for freeing memory

- **Module 3**: Decode Instruction (decode.c, decode.h)
    - Purpose: Decodes the opcode, register, and value information, calling the corresponding function for each instruction using a switch statement
    - Data Structures:
        - None
    - Interactions with other Modules
        - Receives the instruction from Module 1 (Main)
        - Sends the opcode, register, and value information to Module 4
        - Returns to Module 1 (Main) after execution of the instruction is done, returning an int that represents the next program counter

- **Module 4**: Instruction execution (exec.h, exec.c)
    - Purpose: Contains the implementation of each instruction
    - Data Structures:
        - uint32_t registers[8]: global C array of 8 unsigned 32-bit ints
        - Seq_T all_segments: global sequence containing all the segments
        - Seq_T recycled_ids: global sequence containing recycled ids
    - Interactions with other Modules

        - Returns to Module 3 after execution of the instruction is done, with an int that determines where the next pc with be
        - Module 1 calls **initialize_program_state**(segment_zero), a function within Module 4, to initialize the global structures.
        - **loadProg**() may update segment zero, which should simultaneously update in Module 1

<u>C testing functions descriptions and prototypes:</u>

- void print_registers();
    - Prints all the values in registers 0 through 7 as 32-bit unsigned ints
    - Will be used in Module 4
- void print_segments();
    - Prints all the instructions from all the segments in all_segments
    - Will be used in Module 4
- void print_recycled_ids();
    - Print the sequence of ints in recycled_ids
    - Will be used in Module 4
- void print_hex(uint32_t instruction)
    - Prints the hex representation of a 32-bit int
    - Will be used in Modules 1 and 2

<u>Implementation and Testing Plan:</u>

- **Module 1**: main.c, which orchestrates the program execution
    - Create empty main function in main.c
    - Call **store_code()** in main from Module 2, and store the result in a Seq_T named segment_zero
        - Write and test **store_code()** in Module 2, return back when finished
    - Call **initialize_program_state(segment_zero)** to initialize registers, recycled_ids, and all_segments in exec.c
        - Write and test **initialize_program_state()** in Module 4, return back
    - Initialize int pc = 0
    - While pc is less than the length of segment zero:
        - Use Seq_get to get the instruction at segment_zero[pc]
        - TESTING: print the current instruction using print_hex()
    - Call **execute_instruction()** inside the loop, which will be written in Module 3, and store the return value in a int64_t variable called new_pc
        - If the returned value is -1, then the program counter increments. Otherwise, the program counter is set to new_pc
        - TESTING: print new_pc, check that it updates accordingly

- **Module 2**: read.c, read.h, responsible for reading the input from stdin
    - In read.c, create 2 new functions named **read_instruction()** and **store_code()**
        - **read_instruction()** takes no parameters and returns a 32-bit unsigned int

        - **store_code()** takes no parameters and returns a Seq_T
    - In **read_instruction():**
        - Use a for loop to getchar() 4 times from stdin
        - Bitpack the 4 characters into a 32-bit instruction
        - TESTING: print the instruction using out print_hex() function, check that it matches the instruction inputted
        - Return the result
    - In **store_code():**
        - Use Seq_new() to create a new sequence and name it segment_zero
        - Use a while loop to repeatedly call read_instruction() and append the resulting instruction to segment_zero
        - Returns segment_zero after loading all the instructions
    - TESTING: use print_hex() to print each instruction, test by inputting hello.um
        - Check that each instruction corresponds to the appropriate instruction

- **Module 3**: decode.c, decode.h, responsible for extracting the opcode and calling the appropriate instruction to execute
    - Create two functions, named **decode_instruction()** and **handle_instructions()**
        - void **decode_instruction**(uint32_t instruction, uint8_t *A, uint8_t *B, uint8_t *C, uint8_t *opcode, uint32_t *val)
        - uint64_t **handle_instructions**(uint32_t instruction, Seq_T all_segments)
    - Inside **decode_instruction()**:
        - Use bitpack to extract the opcode with width 4 and lsb 28, update the pointer to opcode
        - If opcode is LV, use bitpack to extract register A and the value, update the corresponding pointers
        - Else, use bitpack to extract registers A, B, C, update the pointers
    - Inside **handle_instruction()**:
        - Declare opcode, A, B, C, val variables
        - Call decode_instruction to decode the opcode, A, B, C (and the val if the instruction is load_value)
        - Using a switch statement, we call the appropriate function to handle the instruction, passing in the required registers and values
        - Return -1 if the opcode is different from 12 (load program). Otherwise, it returns the new program counter, which is returned by the **load_program()** function.
    - TESTING: print the opcode, A, B, C, val as unsigned ints and check they matches the input instructions

- **Module 4**: exec.c, exec.h
    - In exec.h, declare global program_variables:
        - uint32_t registers[8]
        - Seq_T all_segments
        - Seq_T recycled_ids

- In exec.c, Create void **initialize_program_state**(Seq_T segment_zero) function that:
    - Initializes all the elements of registers array to 0
    - Sets all_segments equal to result of Seq_seq(segment_zero, NULL)
    - Sets recycled_ids to result of Seq_new(10)
- Write each instruction function in exec.c, in this order. Test using unit tests and C functions. We will diff test with the reference and valgrind to ensure no memory errors/leaks:
    - 1) **halt()**
        - Implementation steps:
            - Use a double nested for-loop to free the memory of each word in each segment of all_segments
            - Exit with success code
        - TEST1:
            - halt();
                - Check that the program ends without any output
    - 2) **loadval()**
        - Implementation steps:
            - Set registers[A] to hold val
        - TEST1:
            - loadval(A = 0; val = 30);
            - halt();
                - Use print_registers() to check that registers[0] holds the value 30, and all other registers remain zero
        - TEST2:
            - for (int i = 0; i < 8; i++) {
        loadval(A = i; val = i);
}
for (int i = 0; i < 8; i++) {
        loadval(A = i; val = i + 1) ;
};
                - Use print_registers() after each loadval() call to check that it first prints “0 1 2 3 4 5 6 7”, then “1 2 3 4 5 6 7 8”
                - This tests the edge case that registers are overwritten using loadval
        - TEST3:
            - loadval(A = 7, val = ‘ó’);
            - halt();
                - Use print_registers() to check that register 7 holds the unsigned int 243
                - This tests the edge case that a non-standard ascii character is loaded into a register in char form
        - TEST4:

        - loadval(A = 4, val = 16777216); //2^24
        - halt();
            - Use print_registers to check that register 4 holds the unsigned int 16777216
            - This tests the edge case of a large value that still fits within 25 unsigned bits
- 3) **output()**
    - Implementation steps:
        - Assert that the value in registers[C] is in range 0 to 255
        - Print the contents of registers[C] to stdout as a character
    - TEST1:
        - load_val(A = 0, val = 65);
        - out(C = 0);
        - halt();
            - Check that “A” (the equivalent of 65 in ASCII) is printed to stdout
    - TEST2:
        - loadval(A = 3, val = ‘h’);
        - out(C = 3);
        - loadval(A = 3, val = “~”);
        - out(C = 3);
        - halt();
            - Check that “h`” is printed to stdout
            - Using print_registers, check that register 3 holds the unsigned int 126 (‘~’, overwrites the first value)
    - TEST3:
        - loadval(A = 8, val = 10);
        - out(C = 8);
        - halt();
            - Check that the newline character is printed to stdout
    - TEST4:
        - loadval(A = 0, val = 300);
        - out(C = 0);
        - halt();
            - This tests the edge case that a value above 255 is in a register before being outputted
            - Expected behaviour is a runtime error
- 4) **input()**:
    - Implementation steps:
        - Use getchar() to get the incoming byte, store as an int (signed)
        - If the result is -1, then store 2^32 - 1 (32 bit int of all 1’s) into r[C]

        - Otherwise, assert that the input is between 0 and 255, convert to uint32_t, and store in r[C]
- TEST1:
    - input(C = 0);
    - input(C = 1);
    - input(C = 2);
    - input(C = 3) (register 3 holds EOF);
    - out(C = 0);
    - out(C = 1);
    - out(C = 2);
    - out(C = 3);
    - halt();
        - Type “Hey” into the keyboard, then command + d
        - Check that “Hey” is printed to stdout
        - Use print_registers to check that:
            - register 0 holds 72 (the ASCII equivalent of ‘H’),
            - register 1 holds 101 (‘e’)
            - register 2 holds 121 (‘y’)
            - register 3 holds 255 (all 1’s in binary, the EOF)
- TEST2:
    - input(C = 0);
    - input(C = 1);
    - input(C = 2);
    - input(C = 3);
    - out(C = 0);
    - out(C = 1);
    - out(C = 2);
    - out(C = 3);
    - input(C = 0);
    - input(C = 4);
    - out(C = 0);
    - out(C = 4);
    - halt();
        - When running the program, redirect the input file containing “\n9* \0p” into stdin
            - This tests redirecting files and non-standard character inputs outside of the range 33-126
        - Check that “9* p” is printed to stdout (on the second line)
        - Use print_registers to check that:
            - register 0 holds 10 (the ASCII equivalent of ‘\n’)
            - register 1 holds 57 (‘9’)
            - register 2 holds 42 (‘*’)
            - register 3 holds 32 (‘ ’)
            - register 4 holds 0 (‘\0’)

            - register 5 holds 112 (‘p’)
- 5) **cmov()**
    - Implementation steps:
        - If the value in registers[C] is 0, set registers[A] equal to the value in registers[B]
    - TEST1:
        - loadval(A = 0, val = 0);
        - loadval(A = 1, val = 50);
        - loadval(A = 2, val = 70);
        - cmov(A = 1, B = 2, C = 0);
        - out(C = 1);
        - out(C = 2);
        - halt();
            - This tests the case where no move is made.
            - Check that the output is ‘2F’ (the ASCII representations of 50 and 70)
    - TEST2:
        - loadval(A = 0, val = 1);
        - loadval(A = 1, val = 50);
        - loadval(A = 2, val = 70);
        - cmov(A = 1, B = 2, C = 0);
        - out(C = 1);
        - out(C = 2);
        - halt();
            - This tests the case where a move is made.
            - Check that the output is ‘FF’ (the first register gets overwritten)
- 6) **add()**
    - Implementation steps:
        - Initialize a variable to 1 and shift it left by 32 bits to get 2^32
            - int mod = 1 << 32;
        - Calculate (registers[B] + registers[C]) % mod, and store in registers[A]
    - TEST1:
        - loadval(A = 0, val = 67);
        - loadval(A = 1, val = 12);
        - loadval(A = 2, val = 9);
        - out(C = 0);
        - add(A = 3, B = 0, C = 1);
        - out(C = 3);
        - out(C = 3);
        - add(A = 4, B = 0, C = 2);
        - out(A = 4);
        - halt();

      - Check that “COOL” is printed to stdout
      - Use print_registers to check that:
        - after the first add() call, register 3 holds 79 (‘O’)
        - After the second add() call, register 4 holds 76 (‘L’)
  - TEST2:
    - loadval(A = 0, val = 2^31);
    - loadval(A = 1; val = 2^31);
    - loadval(A = 2; val = 450);
    - add(A = 3, B = 0, C = 1);
    - add(A = 2, B = 3, C = 2);
    - halt();
      - This tests the edge case that the result of addition is greater than 2^32, meaning the remainder mod 2^32 should be the result
      - It also tests the edge case that the result of the addition is placed in the same register as one of the inputs, replacing them (the second add function).
      - Use print_registers after the first add() call to check that register 3 holds 0
        - This is because 2^31 + 2^31 = 2^32 = 0 mod 2^32
      - Use print_registers after the second add() call to check that register 2 holds 450
        - This is because 0 + 450 = 450
- 7) **multiply()**
  - Implementation steps:
    - Initialize a variable to 1 and shift it left by 32 bits to get 2^32
      - int mod = 1 << 32;
    - Calculate (registers[B] * registers[C]) % mod, and store in registers[A]
  - TEST1:
    - loadval(A = 0, val = 2);
    - loadval(B = 1, val = 33);
    - multiply(A = 1, B = 0, C = 1);
    - out(C = 1);
    - halt();
      - Check that ‘B’ is printed to stdout (the char equivalent of 2*33 = 66)
      - Use print_registers to check that register 1 holds 66 after the multiply() call, which should replace the 33
  - TEST2:
    - loadval(A = 7, val = 2^30);
    - loadval(A = 8, val = 2^8);
    - multiply(A = 5, B = 7, C = 8);
    - out(C = 5);

            - halt();
                - This tests the case that the result of multiplication is greater than 2^32, meaning the remainder mod 2^32 should be the result
                - Check that ‘@’ is printed to stdout
                    - This is because 2^30 * 2^8 = 2^38 = 2^6 (mod 2^32) = 64, which corresponds to ‘@’ in ascii.
                - Use print_registers to check that register holds the value 64 after the multiply() call.
- 8) **divide()**
    - Implementation steps:
        - Assert that registers[C] is different from 0;
        - Calculate registers[B] / registers[C] and put the result in registers[A]
            - Note: remember that this is integer division, so it should round down
    - TEST1:
        - loadval(A = 0, val = 131);
        - loadval(B = 1, val = 2);
        - divide(A = 1, B = 0, C = 1);
        - out(C = 1);
        - halt();
            - Check that ‘A’ is printed to stdout (the ASCII equivalent of 131/2 = 65)
            - This tests the edge case that the value in register b is not evenly divisible by the value in register c, so the result should be rounded down
    - TEST2:
        - loadval(A = 0, val = 10);
        - loadval(B = 1, val = 0);
        - divide(A = 1, B = 0, C = 1);
        - out(C = 1);
        - halt();
            - Check for the corresponding CRE (division by 0)
    - TEST3:
        - loadval(A = 0, val = 122);
        - out(C=0);
        - loadval(A = 1, val = 2);
        - divide(A = 2; B = 0; C = 1);
        - out(C = 2);
        - loadval(A = 5, val = 29);
        - add(A = 4, B = 2, C = 5);
        - out(C = 4);
        - halt();

- Check that “z=Z” is printed to stdout
- Use print registers to check the values
    - registers[0] must be 122;
    - registers[1] must be 2;
    - registers[2] must be 61;
    - registers[4] must be 90;
    - registers[5] must be 29;

- 9) **nand()**
    - Implementation steps:
        - Calculate the Bitwise AND between registers[B] & registers[C]
        - Take the complement of the result and store it in registers[A]
    - TEST1:
        - loadval(A = 0, val = 241);
        - loadval(A = 1, val = 143);
        - nand(A = 2; B = 0; C = 1);
        - out(C = 2);
        - halt();
            - The output should be ‘~’ (126 in decimal)
    - TEST2:
        - loadval(A = 0, val = 100);
        - loadval(A = 1, val = 0);
        - nand(A = 2, B = 0, C = 1);
        - out(C = 2);
        - halt();
            - Check if the final result is ‘ÿ’ (255 in decimal) if one of the registers is 0

10) **map()**
    - Implementation Steps:
        - If recycled_ids is not empty:
            - pop from front to get id
            - Use Seq_new to create a new segment with hint = the value in registers[C]
            - Initialize all values in new sequence to 0
            - Set the value at all_segments[id] to be the new sequence
            - Set registers[B] to id
        - If recycled_ids is empty:
            - Set id to Seq_length()
            - Use Seq_new to create a new segment with hint = the value in registers[C]
            - Initialize all values in new sequence to 0
            - Append new sequence to end of all_segments
            - Set registers[B] to id
    - TEST1:
        - map(B = 2, C = 4);

        - halt();
            - Use print_segments() to print all segments, check that there is a new segment with id 1, all four values initialized to 0
            - Use print_registers() to check that the value in registers[2] is 1 (the id)
- 11) **unMap()**
    - Implementation Steps:
        - Get the curr_segment = index registers[C] in all_segments
        - Free the elements in the curr_segment, then use Seq_free() to free curr_segment
        - Set the element in all_segments to NULL
        - Push the id (held in registers[C]) into recycled_ids
    - TEST1:
        - map(B = 2, C = 4);
        - unMap(C = 2);
        - halt();
            - Use print_segments() to check that there is only segment_zero after unmapping
            - Use print_recycled_ids() to check that id 1 is in Seq_T recycled_ids
    - TEST2:
        - map(B = 0, C = 3);
        - map(B = 1, C = 4);
        - unMap(C = 0);
        - map(B = 2, C = 5);
        - halt();
            - Use print_segments() after each step to check that:
                - 1st map: seg 1 = {0, 0, 0}
                - 2nd map: seg 2 = {0, 0, 0, 0}
                - unmap: seg 1 = NULL
                - 3rd map: seg 1 = {0, 0, 0, 0, 0}
            - Use print_recycled_ids() after each instruction:
                - 1st map: recycled_ids = {}
                - 2nd map: recycled_ids = {}
                - unmap: recycled_ids = {1}
                - 3rd map: recycled_ids = {}
- 12) **segStore()**
    - Implementation steps:
        - Use Seq_get to get the curr_segment at index registers[A] of all_segments
        - Get instruction at registers[C[
        - Use Seq_put to place the instruction into index registers[B] in curr_segment

- TEST1:
    - map(B = 0, C = 4);
    - segStore(A = 0, B = 2, C = 255);
    - halt();
        - Use print_segments() to check that index 2 of segment 1 holds the value 255
- 13) **segLoad()**
    - Implementation steps:
        - Use Seq_get to get the segment at index registers[B] of all_segments, store in a variable named curr_segment
        - Use Seq_get to get the instruction at index registers[C] of curr_segment, store in registers[A]
    - TEST1:
        - map(B = 0, C = 4);
        - segStore(A = 0, B = 2, C = 126);
        - segLoad(A = 3, B = 0, C = 2);
        - out(C = 3);
        - halt();
            - Use print_segments() to check that index 2 of segment 1 holds the value 126
            - Check that “~” is printed to stdout
    - TEST2:
        - map(B = 0, C = 10)
        - segLoad(A = 3, B = 0, C = 2);
        - halt();
            - Use print_registers() to check that registers[3] holds 0
- 14) **loadProg()**
    - Implementation steps:
        - If r[B] = 0:
            - return r[C] to execute_instructions() to set the pc in main()
        - else:
            - Use Seq_get to get the segment at index r[B], name it to_copy
            - Use Seq_new to create a new segment with the same length, called new_segment
            - Use a for loop to copy the elements of to_copy into new_segment
            - Use Seq_get to get the segment at index 0
            - Free all elements in segment_zero
            - Set segment_zero to new_segment
            - Return r[C] execute_instructions() to set the pc in main()
    - TEST1:
        - loadval(A = 0, val = 65);
        - loadval(A = 1, val = 0);

                - loadProg(B = 1, C = 4);
                - out(C = 0);
                - halt();
                    - Check that nothing is printed, meaning the program counter skips to 4
                    - This tests the case where the segment to put into segment 0 is just 0, meaning the program counter is updated
            - TEST2:
                - loadval(A = 0, val = 65);
                - map(B = 1, C = 5) // id is in register 1, segment 1 is 5 long
                - loadval(A = 2, val = 29360128); // register 2 holds halt instruction before shifting
                - loadval(A = 3, val = 64);
                - multiply(A = 4, B = 2, C = 3) // multiply to get halt instruction
                - segStore(A = 1, B = 3, C = 4) // store halt into index 3 of seg 1
                - loadProg(B = 1, C = 3)
                    - // duplicate segment 1, start program counter at 3 (the halt instruction)
                - out(C = 0); //This should not print out
                - halt();
                    - Check that 'A' does not print
                    - Use print_segments() to check the structure of the segments
                        - After loadprog, segment 0 should just hold the halt instruction at index 3

Final testing:

- Input hello.um, review.um, etc
- Diff check results with reference implementation
- Check no memory errors or leaks

<u>Loop invariant:</u>

- At the start of each iteration in main, 0 <= pc < length of segment_zero