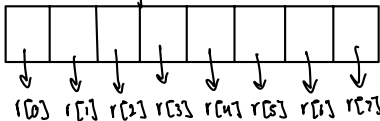
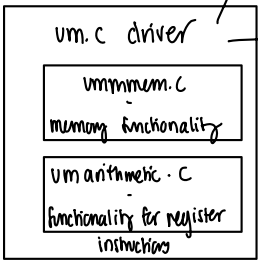
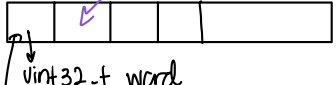
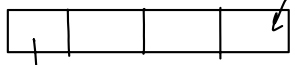
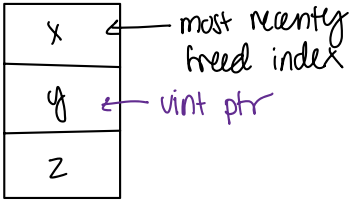


### Architecture:

Module and purpose	Module parts and responsibilities	Applicable drawings
um.c: responsible for driving the UM.	<ul style="list-style-type: none"> <li>- Utilizes ummem.c for memory management commands and functionality</li> <li>- Utilizes umarithmetic.c for arithmetic functionality</li> <li>- Owns the emulator's registers</li> <li>- Owns instance of the memory struct defined in UM               <ul style="list-style-type: none"> <li>- Segmented memory and stack of unmapped memory addresses</li> <li>- 0 segment loaded with the program using exported functions from ummem.c</li> </ul> </li> <li>- Owns the program pointer               <ul style="list-style-type: none"> <li>- (uint32_t)(uintptr_t) pointer to a uint32_t in the 0 segment</li> </ul> </li> <li>- Defines functionality for the I/O device</li> <li>- Owns query loop for opcodes determining which instruction is executed</li> </ul> <p>Data structures:</p> <ul style="list-style-type: none"> <li>- 8 index C array of uint32_t's representing registers r[0] through r[7]</li> <li>- Exported UMmem structure from ummem holding segmented memory and unmapped IDs</li> </ul>	<p>uint32_t registers 1-8</p>  <p>um.c driver</p>  <p>program pointer, IO device</p> <p>\$m[0]:</p>  <p>uint32_t word</p> <p>program ptr starts here [0][0] - after instruction executes, go to [0][1]</p>
ummem.c: Defines memory operations needed by the UM.	<p>Defines functionality for load program, load value, map segment, and unmap segment commands</p> <p>Data structures:</p> <ol style="list-style-type: none"> <li>1) UM_Mem struct holding:           <ol style="list-style-type: none"> <li>1.1) Hanson's Seq_T of UArray_Ts to hold all segmented memory of the program               <ul style="list-style-type: none"> <li>- Members of UArray_Ts are uint32_t pointers holding a UM instruction</li> </ul> </li> <li>1.2) Stack containing uint32_t pointers holding IDs of unmapped segments, such that they can be reused.</li> </ol> </li> </ol> <p>The "ID" for a segment is its slot in the array. Each time a segment is added in the program, it will be loaded into a UArray and the UArray pointer will be added to the sequence at index (current length of sequence)</p>	<p>UM_mem struct:</p> <pre> struct UM_mem {     Seq_T mapped_segments;     Stack_T unmapped_IDS; } </pre> <p>Seq_T mapped_segments:</p> <p>\$m[0] is 0 index of sequence of segments.</p>  <p>UArray_T ptr</p> <p>uint32_t word</p> <p>UArray_T</p>

	<p>Mapping segments allocates memory for a UArray and adds to it to the sequence, while unmapping a segment frees a UArray.</p> <p>When a segment is unmapped, that index of the sequence will be empty. We will push the "ID" (pointer holding the index) of the unmapped segment on a stack to be reused in the future i.e. when a segment is next mapped. The new mapped segment will be placed at the index on top of the unmappedIDs stack and the stack will be popped. If the stack is empty, the sequence will be expanded.</p>	<p>Stack - T unmapped_memory of vint pointers</p>  <pre> graph TD     subgraph Stack_T [Stack - T]         direction TB         x[x]         y[y]         z[z]     end     x -- "most recently freed index" --&gt; x     y -- "vint ptr" --&gt; y </pre>
<p>Uarithmetic.c: Defines arithmetic functions to modify the registers in UM based on given UM instructions.</p>	<p>No data structures: updates UM registers defined in um.c based on given instruction</p> <p>static inline definitions of functions</p> <ul style="list-style-type: none"> <li>- Add</li> <li>- Multiply</li> <li>- Divide</li> <li>- Halt</li> <li>- Conditional move</li> <li>- Segment load</li> <li>- Segment store</li> <li>- Bitwise NAND</li> </ul>	

### Implementation Plan

Module section and corresponding testing section	Substeps for implementation
<b>Module: ummem.c</b> <b>Data structures and load program</b> <ol style="list-style-type: none"> <li>1. Define ummem struct and initialize members</li> </ol> <p>Testing: See testing step 1</p>	<ol style="list-style-type: none"> <li>1. Define struct in ummem module</li> <li>2. Implement constructor- type function which initializes Seq_T mapped_memory and Stack_T unmappedID within the struct and returns a struct. This will be called in the um driver module.</li> </ol>
<b>Module: um.c</b> <ol style="list-style-type: none"> <li>1. Initialize C array holding registers</li> <li>2. Implement I/O device (read from stdin and output to stdout)               <ol style="list-style-type: none"> <li>a. Includes helper function to get length of file</li> </ol> </li> <li>3. Implement query loop</li> <li>4. Initialize program pointer and implement advancement of program pointer</li> </ol> <p>Testing: see testing step 2</p>	<ol style="list-style-type: none"> <li>1. Initialize the uint32_t array of 8 indexes called "um_registers" and initialize all values to 0.</li> <li>2. Implement "in" part of the IO device by reading in from stdin and printing read value to stdout               <ol style="list-style-type: none"> <li>a. Implement a helper function which gets the length of input file using stat functions, determining how many words will be entered into the 0 segment by load_program and therefore how long the UArray holding that segment will be.</li> </ol> </li> <li>3. Implement "out" part of the IO device and test in conjunction with "in" part of IO device</li> <li>4. Implement query loop               <ol style="list-style-type: none"> <li>a. Gets first 4 bits of instruction and calls function corresponding to that opcode</li> </ol> </li> <li>5. Initialize a uintptr cast as a uint32_t pointer to serve as the program pointer in the main driver and initially points to \$m[0][0].</li> <li>6. Implement a looping helper function that moves the pointer to \$m[0][n+1] after instructions at \$m[0][n] have been executed.</li> </ol>
<b>Module: ummem.c</b> <b>Load value and load program</b> <ol style="list-style-type: none"> <li>1. Implement/ transfer loadval from lab code</li> <li>2. Implement <u>load program</u> functionality</li> </ol>	<ol style="list-style-type: none"> <li>1. Copy our loadval function from our umlab.               <ol style="list-style-type: none"> <li>a. In order to do this we will also need to transfer the typedefs for Um_instructions into our um</li> </ol> </li> </ol>

<p><b>Map and unmap</b></p> <ol style="list-style-type: none"> <li>3. Implement map segment function <ol style="list-style-type: none"> <li>a. Includes helper function that checks for available unmapped IDs to be reused</li> </ol> </li> <li>4. Implement unmap segment function</li> </ol> <p>Testing: see testing step 3</p>	<p>module.</p> <ol style="list-style-type: none"> <li>2. Implement load_program functionality, which will interact with the ummethe program into segment[0] of the mapped_memory sequence</li> <li>3. Implement a helper function (ID_generator) that checks for unmapped IDs by checking the size of the unmappedIDs stack. <ol style="list-style-type: none"> <li>a. If there is an unmapped ID on the stack, the new mapped segment will go at this ID (index in the mapped_IDs sequence)</li> <li>b. If the stack is empty, the mapping functionality will place the new segment at Seq_length(mapped_memory).</li> </ol> </li> <li>4. Implement instruction that maps a segment:</li> </ol> <p>Note: The function will take input from the Input functionality of the I/O device</p> <ol style="list-style-type: none"> <li>a. Initialize a new UArray of length determined by the value in \$r[C] with all values set to 0.</li> <li>b. Call helper function described in substep 1 to determine ID of the new segment and thus its position in the mapped_memory sequence.</li> <li>c. Put the UArray representing the segment into the mapped_memory sequence and the ID/ index of the segment in \$r[B]</li> </ol> <ol style="list-style-type: none"> <li>5. Implement instruction that unmaps a sequence <ol style="list-style-type: none"> <li>a. Get value from \$r[C] in UM and go to that index in the mapped_segments sequence</li> <li>b. Free the UArray at that address</li> <li>c. Push the value of \$r[C] onto the unmapped_IDs stack</li> </ol> </li> </ol>
<p><b>Module: umarithmetic.c</b>  <b>Implements functions that perform opcodes indicated by query loop</b></p> <ol style="list-style-type: none"> <li>1. Implement/ transfer halt function from lab</li> <li>2. Implement/ transfer add function from lab</li> <li>3. Implement multiplication</li> <li>4. Implement division</li> </ol>	<ol style="list-style-type: none"> <li>0. Move all necessary typedefs for registers and Um_instructions from the lab into the umarithmetic module</li> <li>1. Transfer halt function from lab and alter to be independent of lab three_registers function i.e. directly perform instruction</li> <li>2. Transfer add function from lab and alter to be independent of lab three_registers i.e. directly</li> </ol>

5. Implement bitwise NAND
6. Implement conditional move
7. Implement segment load
8. Implement segment store
9. Implement input
10. Implement output

Testing: see testing step 4

- perform computation
3. Use same structure as add function to implement function that multiplies values held in the registers array in um.c at indexes B and C and places value in index A
4. Use same structure as add and multiply functions to implement function that divides values of registers B and C and places value in register A
5. Use same structure as add, multiply, and divide functions to bitwise NAND values in registers B and C and store in register A
  - a. And B and C
  - b. Get two's complement of that value
  - c. Store in rA
6. Conditional move
  - a. Check if the value in \$r[C] is 0
  - b. If it is, do nothing
  - c. If it is not, put the value of \$r[B] into r[A]
7. Segmented load
  - a. Go to segment \$r[B] (index in mapped\_segments Seq\_T)
  - b. Go to index \$r[C] in the UArray and retrieve value
  - c. Store value in \$r[A]
8. Segmented store
  - a. Go to segment \$r[A] (index in mapped\_segments Seq\_T)
  - b. Go to index \$r[B] in the UArray and retrieve value
  - c. Store value in \$r[C]
9. Input
  - a. Activate I/O device and wait for input
  - b. Check that value can fit into 32 bits (0-255)
  - c. If so, put in \$r[C]
  - d. If EOF, put 255 in \$r[C] (all ones)
10. Output
  - a. Check if value is between 0-255
  - b. If so, sent to I/O device to be outputted

## Testing

Step and module	Segment abstraction/ overall program testing	Instruction set testing
<u>Testing Step 1</u> Module: umem.c Initialize ummem struct and initialize members	<p>Test the initialization of the UM struct:</p> <p>To be called in um.c testing main Calls ummem struct constructor, tries adding and removing uint32_t pointers from the unmapped_ID Stack_T void init_test_stack()</p> <p>To be called in um.c testing main Calls ummem struct constructor, tries adding and removing UArrays from the mapped_memory Seq_T. void init_test_mem()</p>	n/a - no instructions implemented at this step
<u>Testing Step 2</u> Module: um.c	<p>1. Test the initialization of the UM's registers after they are initialized in the UM: to be called in um testing main</p> <ul style="list-style-type: none"><li>- Assert value of registers 0 - 8 is 0</li></ul> <pre>void test_reg_init(uint32_t um_registers[]);</pre> <p>Test loading of the UM's registers: to be called in um testing main</p> <ul style="list-style-type: none"><li>- Load each register with a non-zero value and assert each value</li><li>- Reload each register with a different non-zero value and assert each value</li></ul> <pre>void test_reg_load(uint32_t um_registers[]);</pre> <p>2. Test I/O device</p> <p>Test input section: read from stdin and a redirected file on the command line and redirect output to file. To be called in um.c testing main, stream will be initialized from file or from stdin. Diff</p>	n/a - no instructions implemented at this step

	<p>input and output</p> <ul style="list-style-type: none"> <li>- Case where input is out of 0-255 range</li> </ul> <pre>void input_IO_test(Seq_T stream)</pre> <p>Test output section: same test as described above, but instead of redirecting output, output comes from the output functionality of the I/O device.</p> <ul style="list-style-type: none"> <li>- Case where input is out of 0-255 range</li> </ul> <pre>void output_IO_test(Seq_T stream)</pre> <p>3. Test query loop:</p> <ul style="list-style-type: none"> <li>- Both tests assert extracted field corresponds to correct opcode and we are extracting top MSBs reliably: This has to be done before we can reliably input instruction tests</li> </ul> <p>Test will declare instruction from which opcode will be extracted</p> <pre>void query_test()</pre> <p>Test will extract instruction from stream read in from I/O device and p</p> <pre>void query_test_stream(Seq_T stream)</pre> <p>4. Test initialization of program pointer and movement</p> <ul style="list-style-type: none"> <li>- Assert initialization to \$m[0][0] before advancement</li> <li>- After each advancement of the pointer, assert it has only moved one index</li> <li>- Edge case where we are at the end of the segment - pointer should not go out of bounds in order to avoid segfaulting</li> </ul> <pre>void pointer_adv(uint32_t</pre>	
--	---	--

	*programp)	
<p><u>Testing Step 3.</u> Module: ummem.c</p> <p>Load value and load program, map and unmap</p>	<p><b>See instruction testing steps 3 and 4 in box directly to the right for map/ unmap</b></p>	<ol style="list-style-type: none"> <li>1. Loadval: Create different inputs with loadval opcode and run to test following aspect and cases: <ol style="list-style-type: none"> <li>a. Assure we are only extracting 25 bits</li> <li>b. Test when extracted value is 0</li> <li>c. Test when extracted value is top of range</li> <li>d. Test when we load val in the same register back to back from a high value to a low value: entire value should be replaced</li> </ol> </li> <li>2. Load program: create different inputs with load program opcode and run to test following aspects and cases: <ol style="list-style-type: none"> <li>a. Assure \$m[\$r[B]] is the segment being copied <ol style="list-style-type: none"> <li>i. Print/ assert value before and after load program operation is performed</li> </ol> </li> <li>b. Assure old \$m[0] is freed (run valgrind)</li> <li>c. Program has no words</li> <li>d. \$r[B] is 0</li> <li>e. Assure pointer is set \$m[0][\$r[c]] and that the user going out of bounds will not cause an illegal segfault</li> </ol> </li> <li>3. Map segment: create different inputs with map segment opcode and run to test following aspects and cases: <ol style="list-style-type: none"> <li>a. Assure the ID generation helper function takes IDs from the unmapped_ID stack when they are available, filling empty slots in the Seq_T <ol style="list-style-type: none"> <li>i. Assert that the stack is popped after ID is used so we do not overwrite an index holding a mapped segment</li> </ol> </li> <li>b. Assure the ID generation does not pop an empty stack</li> <li>c. Assert UArray to hold segment is initialized to the length of the value in \$r[C]</li> <li>d. Assert UArray segment is</li> </ol> </li> </ol>



		<p>placed at the correct ID/index</p> <ol style="list-style-type: none"> <li>4. Unmap segment: create different inputs with unmap segment opcode and run to test following aspects and cases: <ol style="list-style-type: none"> <li>a. Assure unmapped IDs are added to the stack by asserting value of the top of the stack as well as the size of the stack</li> <li>b. Assert we unmap the sequence indicated in \$r[C] and not another</li> <li>c. Valgrind to make sure unmapped UArray has been freed</li> </ol> </li> </ol>
<p><u>Testing Step 4</u> Module: uarithmic.c</p> <p><b>Implements functions that perform opcodes indicated by query loop</b></p>	<p>n/a no segments managed at this step of implementation</p>	<ol style="list-style-type: none"> <li>1. halt function: create different inputs with halt opcode and run to test if halt test stops computation. All following tests depend on the functionality of halt since it ends every instruction set</li> <li>2. Add function: create different inputs with add opcode and run to test following aspects and cases: <ol style="list-style-type: none"> <li>a. Registers rb and rc are added and placed in ra, and none of them are switched around in our computation</li> <li>b. Ensure full value is stored in indicated register if it is within the 32 bit range and</li> <li>c. Values to be added are at the top of 32 bit range and sum therefore exceeds 32 bit range</li> </ol> </li> <li>3. Multiplication function: create different inputs with mult opcode and run to test following aspects and cases: <ol style="list-style-type: none"> <li>a. Registers rb and rc are multiplied and placed in ra, and none of them are switched around in our computation</li> <li>b. Assert full value is stored in indicated register if it is within the 32 bit range</li> <li>c. Values to be added are at the top of 32 bit range and product therefore exceeds 32 bit range</li> </ol> </li> <li>4. Division function: create different</li> </ol>

		<p>inputs with division opcode and run to test following aspects and cases:</p> <ol style="list-style-type: none"> <li>Register rb is divided by rc and quotient is placed in ra, and none of them are switched around in our computation</li> <li>Ensure full value is stored in indicated register if it is within the 32 bit range</li> <li>Quotient is not an integer</li> </ol> <p>5. BNAND function: create different inputs with BNAND opcode and run to test following aspects and cases:</p> <ol style="list-style-type: none"> <li>Registers rb and rc are anded and the result is placed in ra, and none of these registers are switched around in our computation</li> <li>Ensure full value is stored within indicated register if it is within 32 bit range</li> </ol> <p>6. Conditional move: create different inputs with conditional move opcode and run to test following aspects and cases:</p> <ol style="list-style-type: none"> <li>r[C] is not 0 - jump should be made</li> <li>r[C] is 0 jump should not be made</li> <li>Ensure registers \$r[B] and \$r[A] are the ones being updated</li> </ol> <p>7. Segmented load: create different inputs with segmented load opcode and run to test following aspects and cases:</p> <ol style="list-style-type: none"> <li>Referenced segment is not mapped (we don't have to handle this but we should make sure another segment isn't altered in the process)</li> <li>Ensure full value is loaded into \$r[A] if it is within the 32 bit range and all other registers remain unchanged</li> <li>Referenced register is out of bounds (we don't have to handle this but we should make sure another register isn't altered in the process.)</li> </ol> <p>8. Segmented store: create different inputs with segmented store opcode and run to test following aspects and cases:</p>
--	--	--

		<ul style="list-style-type: none"> <li>a. Referenced segment is not mapped (we don't have to handle this but we should make sure another segment isn't altered in the process)</li> <li>b. Referenced register is out of bounds (we don't have to handle this but we should make sure another register isn't altered in the process.)</li> <li>c. Ensure full value is loaded into correct segment if it is in range</li> </ul> <p>9. Input: create different inputs with input opcode and run to test following aspects and cases</p> <ul style="list-style-type: none"> <li>a. Value read is outside of 0-255</li> <li>b. There isn't input <ul style="list-style-type: none"> <li>i. In this case, ensure we load 32 bits of 1s into the \$r[C]</li> </ul> </li> <li>c. Ensure we place input into \$r[C] and all other registers are unaltered</li> </ul> <p>10. Output: create different inputs with output opcode and run to test following aspects and cases:</p> <ul style="list-style-type: none"> <li>a. Value being outputted is outside of 0-255 range</li> </ul>
<u>Additional Unit Tests</u>	<p>Additional Unit tests will combine instructions and follow the same procedure as described in the individual unit tests.</p> <p>We will create two instructions, for example add then multiply, and then run these instructions consecutively in the UM testing main to test that registers are kept consistent when they should be consistent and are altered when they should be altered.</p> <p>We will expand to three or more instructions to check the functionality of the instructions manually and individually before testing the program functionality as a whole.</p>	



### Function contracts for register instructions:

Instruction	Contract specifying interaction with other modules note Um_register is typedefed as an int holding the index in the registers array in um.c ex: r1 is index 1 <i>Note any arithmetic operations resulting in a value greater than 32 bits will C2E</i>
0 Conditional move	<p>If \$r[C] is not 0 then \$r[B] is put into \$r[A]. Moves the value of register B into register A.</p> <p>Parameters: note Um_register is typedefed: is an index in the registers array in um.c</p> <p>Um_register ra – Indicates which UM register is being copied into</p> <p>Um_register rb - Indicates which UM register is having value copied</p> <p>Um_register rc – Indicates which UM register holds the 0 condition for the jump</p> <p>Returns: none</p> <p>Effects: changes value of \$r[A]</p> <p>Interacts with um.c registers by copying the value of register B into register A</p>
1 Segmented load	<p>Copies value at segment \$m[\$r[B]][\$r[C]] into \$r[A]</p> <p>Parameters:</p> <p>Um_register ra – Indicates which UM register where value at indicated segment will be stored</p> <p>Um_register rb - Indicates which register holds the value indicating the segment</p> <p>Um_register rc – Indicates which UM register holds the value of the address within the segment</p> <p>Returns: none</p> <p>Effects: changes value of \$r[A]</p> <p>Interacts with ummem module by accessing the mapped_memory sequence in the ummem struct and with um.c by changing the value of \$r[A]</p>
2 Segmented store	<p>Copies value at segment \$m[\$r[A]][\$r[B]] into \$r[C]</p>

	<p>Parameters:</p> <p>Um_register rc – Indicates which UM register where value at indicated segment will be stored</p> <p>Um_register ra - Indicates which register holds the value indicating the segment we will access</p> <p>Um_register rb – Indicates which UM register holds the value of the address within the segment</p> <p>Returns: none</p> <p>Effects: alters the value of mapped memory at <math>\\$m[\\$r[A]][\\$r[B]]</math></p> <p>Interacts with ummem module by accessing and altering mapped memory at <math>\\$m[\\$r[A]][\\$r[B]]</math> and with um.c module by accessing <math>\\$r[C]</math></p>
3 addition	<p>Adds the values in <math>\\$r[B]</math> and <math>\\$r[C]</math> and mods the result by <math>2^{32}</math> to ensure it is within 32 bits. Then loads result into register A</p> <p>Parameters:</p> <p>Um_register ra – Indicates in which UM register the sum of the addition will be stored</p> <p>Um_register rb - Indicates first register whose value is being added</p> <p>Um_register rc – Indicates second register whose values is being added</p> <p>Returns: none</p> <p>Effects: Alters value of <math>\\$r[A]</math></p> <p>Interacts with um.c by accessing it's registers</p>
4 multiplication	<p>Multiplies the values in <math>\\$r[B]</math> and <math>\\$r[C]</math> and mods the result by <math>2^{32}</math> to ensure it is within 32 bits. Then loads result into register A</p> <p>Parameters:</p> <p>Um_register ra – Indicates in which UM register the product of the addition will be stored</p> <p>Um_register rb - Indicates first register whose value is being multiplied</p> <p>Um_register rc – Indicates second register whose values is being multiplied</p> <p>Returns: none</p> <p>Effects: Alters value of <math>\\$r[A]</math></p> <p>Interacts with um.c by accessing its registers</p>
5 division	<p>Divides the values in <math>\\$r[B]</math> by <math>\\$r[C]</math> and mods the result by <math>2^{32}</math> to ensure it is within 32 bits. Then loads result into register A</p> <p>Parameters:</p> <p>Um_register ra – Indicates in which UM register the quotient will be stored</p> <p>Um_register rb - Indicates register whose value is the numerator</p> <p>Um_register rc – Indicates register whose value is the denominator</p> <p>Returns: none</p> <p>Effects: Alters value of <math>\\$r[A]</math></p>

	Interacts with um.c by accessing its registers
6 Bitwise NAND	<p>Bitwise ands the values in \$r[B] by \$r[C] and negates it (two's complement). Then loads the result into \$r[A]</p> <p>Parameters:  Um_register ra – Indicates in which UM register the result will be stored  Um_register rb - Indicates first register whose value is being anded  Um_register rc – Indicates second register whose value is being anded  Returns: none  Effects: Alters value of \$r[A]</p> <p>Interacts with um.c by accessing its registers</p>
7 Halt	<p>Stops the program's computation</p> <p>Parameters:  none  Returns: updated um_instruction  Effects: stops program</p>
8 Map segment	<p>Adds a new segment of memory into mapped_memory Seq_T in the ummem struct of size held in \$r[C] and sets all slots to 0. Put segment ID into \$r[B] and maps segment to \$m[\$r[B]]</p> <p>Parameters:  Um_register rb – register into which segment ID will be placed  Um_register rc - register where size of new segment is held</p> <p>Returns: none</p> <p>Effects: allocates memory for a new segment, may decrease size of stack if unmapped ID is reused. Increases size of Seq_T otherwise.</p> <p>Interacts with ummem.c by altering mapped_memory Seq_T  Interacts with um.c by accessing its registers</p>
9 unmap segment	<p>Removes a segment of memory from the mapped_memory Seq_T in the ummem struct at index indicated by value or \$r[C]  Pushes the index that was unmapped onto the unmapped_IDs Stack_T</p> <p>Parameters:  Um_register rc - register where ID of unmapped segment will be held</p> <p>Returns: none  Effects: deallocates memory for a segment, allocates memory for ID pointer on the stack</p>

	<p>Interacts with ummem.c by altering mapped memory Seq_T</p> <p>Interacts with um.c by altering its registers</p>
10 output	<p>Writes value of \$r[C] to output through the I/O device, checking if these values are within the 0-255 range.</p> <p>Parameters: Um_register rc - register where value to be written is held</p> <p>Returns: none</p> <p>Effects: none</p> <p>Interacts with um.c by accessing its registers and I/O device</p>
11 input	<p>Opens input of I/O device and loads input into \$r[C]. If there is no input/ input is EOF, put 32 bits of 1's into \$r[C]. Checks if input is in 0-255 range.</p> <p>Parameters: Um_register rc- register where input will be loaded</p> <p>Returns: none</p> <p>Effects: alters \$r[C]</p> <p>Interacts with um.c by accessing its registers and I/O device</p>
12 load program	<p>Segment \$m[\$r[B]] is duplicated and loaded into \$m[0]. \$m[0] is deallocated. The program counter is set to point to \$m[0][\$r[C]].</p> <p>Parameters:  Um_register rb - register indicating which segment will be duplicated and loaded into \$m[0]</p> <p>Um_register rc- register indicating which address in the 0 segment the program pointer will be set to</p> <p>Returns: none</p> <p>Effects: Alters \$m[0] by deallocating one segment and allocating a new one. Changes position of program pointer</p> <p>Interacts with um.c by accessing its registers and changing the value of the program pointer. Interacts with ummem.c by altering the position of the program pointer.</p>
13 load value	<p>Extracts and loads lower 25 bits of a word into \$r[A]</p>

	<p>Parameters:</p> <p>UInt32_t word/ instruction from which value is extracted</p> <p>Um_register ra - register indicating which segment the value will be loaded into</p> <p>Returns: none</p> <p>Effects: Alters \$r[A]</p> <p>Interacts with um.c by accessing its registers</p>
--	---