Zoom: https://tufts.zoom.us/j/92712511308
VSCode:

UM Architecture

Overview

- Segmented Memory:
  - each **instruction** is a 32-bit word (uint32_t)
  - each **segment** is a **Hanson sequence** of 32-bit instructions
  - the entire memory structure is a collection of segments (more specifically, a **Hanson sequence** of **pointers to segments**)
  - when a new segment is mapped, a pointer to this segment is added to the collection of segment pointers**.** This means that each segment's **unique segment ID** is its **index** in the sequence of segment pointers. This sequence will allow us access a particular segment when given a segment ID number.
  - when a segment is unmapped, we will push its segment ID number onto a **stack** that will keep track of **segment ID numbers that are available for reuse** (in the order that they become available).
- Registers:
  - registers are represented as an **8-element array of uint32_t objects**
  - 0-indexed (i.e. register 1 → element 0, register 2 → element 1, etc.)
- Program counter:
  - the program counter will be represented as a **pointer to a 32-bit instruction (i.e. a uin32_t pointer)**
  - to set the program counter → start the program counter at the first instruction of the sequence in the 0-segment (0-index in segmentID sequence)
  - to increment the program counter → pointer arithmetic (pcounter++) to move it to the next instruction in the segment

---

Implementation/Testing Plan

---

module	description
mainum.c	**input:** pathname for a .um file that contains machine instructions for our UM to execute **tasks:** - usage: ./um [filename] - ensures no more than 2 command line arguments were specified - opens the provided file, which should containing binary machine instructions - if opened successfully, passes a pointer to the opened file into the execute function **output:** file pointer to the opened file containing binary machine instructions

processfile.h	**purpose:** interface for the processfile.c module. will declare the following function: - function that will **read the binary contents of the .um file** into a sequence of **uint32_t instructions**
processfile.c	**input:** pointer to an open file containing machine instructions in binary **tasks:** - initialize a **Hanson sequence** of uint32_t instructions (for the segment) - call **getchar()** to extract each byte and **bitpack_newu** to pack four bytes at a time into a 32-bit word - append each 32-bit word onto the low end of the sequence of instructions. the resulting sequence is our **0-segment** - initialize a **Hanson sequence of pointers to sequences.** this will be our **segmentID sequence** that will hold pointers to all the segments. in the **0th index**, store a pointer to the 0-segment we just built, which now contains all the instructions from the input file **testing:** - test: ensure that the instructions we put into a file are the same as the instructions that we pull out of the file - test: write simple unit tests for both types of instructions, then print out opcodes and register numbers as we read each instruction in) **output:** an initialized segmentID sequence with only one mapped segment, the 0-segment (i.e. there is just one element in the 0-index of the sequence, and it's a pointer to the sequence containing all the program instructions)
execute.h	**purpose:** interface for the execute.c module. will declare the following functions: - function to set the initial state of the machine - function to prompt the execution cycle → get an instruction, pass the instruction to the function that will execute it, then update the status of the machine - functions for actually executing each of the 14 program instructions
execute.c	**input:** segmentID sequence with only the 0-segment, or 0th index, mapped **tasks → initializer function:** - initialize **stack** → declare the stack that will store available 32-bit segmentID numbers (i.e. as segments are unmapped) - initialize **registers** → declare an 8-element array of uint32_t's, and populate each array element with 0 (all registers start at 0). - initialize **program counter** → declare a uint32_t pointer for the program counter and have it point to the first instruction of the 0-segment. **tasks → execution cycle function:** - start **execution cycle** (repeat until we reach a halt instruction or some sort of failure) - (1) get current instruction - (2) check instruction's opcode (using bitpack_getu) to determine

	which function to pass it to - (3) call the corresponding function based on the opcode number (i.e. the instruction being executed) - (4) upon completion of the current instruction, increment the program counter to retrieve the next instruction (pcounter++) **testing:** - (1) → ensure the program counter points to the first instruction of the input file (which we created) at the start of execution (print the instruction?) - (2)/(3) → ensure our function for opcode checking accurately matches each opcode to the correct instruction + corresponding function - (3) → work with one instruction at a time, ensuring we are calling the proper function that corresponds to this instruction - (4) → for everything except the load program instruction, ensure the program counter increments to the next instruction at the end of execution - (4) → following a load program instruction, ensure the program counter points to the correct instruction in the newly loaded segment (print the instruction it points to ?) - test: ensure program counter is pointing where it should be following a load program instruction - (4) → ensure the execution cycle stops upon receiving a halt instruction **output:** none, will continue executing until it's reached the end of the program, outputting anything sent to stdout along the way
CMOV (0)	**input:** uint32_t instruction (with opcode 0000) **tasks:** - use array-indexing on the **registers array** to access the 32-bit values in each register - check the value in register C, and **compare to 0** - if the value is zero, return. - if the value is nonzero, **"move"/copy the value** that is in register B into register A, then return. **unit tests:** - load both zero and nonzero values into register C - load some value into register B - load some value into register A (to be overwritten) - perform a conditional move into register A **output:** none, but modifies the contents of the registers array.
SLOAD (1)	**input:** uint32_t instruction (with opcode 0001) **tasks:** - grab the value in register B and store in a local int variable → this value will be the **segment ID number.** - grab the value in register C and store in a local in variable → this value will be the **offset** (index within segment)

	- use a sequence **getter function** on the sequence of segment pointers, passing in the **segment ID number as the index.** this function call will return a pointer to the correct segment to be loaded from. - use a sequence **getter function** on the segment sequence, passing in the **offset** as the index. this function call will return a pointer to the uint32_t word to load into the register - after retrieving the 32-bit word, **"load"/copy** its value into register A, using array-indexing **unit tests:** - load values into registers B and C for the segment ID and offset - print out the value in each register as we grab it to ensure it is correct - map a new segment, and load one simple instruction into this segment at a specific index/offset - call segmented load on the new segment we mapped at the specific offset where we know the instruction is - print out the instruction after we grab it to ensure it is the correct instruction **output:** none, but register A now contains the 32-bit word located in segment B with an offset C
SSTORE (2)	**input:** uint32_t instruction (with opcode 0010) **tasks:** - grab the value in register A and store in a local int variable → this value will be the **segment ID number.** - grab the value in register B and store in a local in variable → this value will be the **offset** (index within segment) - grab the word in register C and store in a local uint32_t variable → this 32-bit value is the **word being stored** in memory - use a sequence **getter function** on the sequence of segment pointers, passing in the **segment ID number as the index.** this function call will return a pointer to the segment we will be storing in. - use a sequence **getter function** on the segment, passing in the **offset** as the index → this will return a pointer to the position within the segment where we will store the word. - modify the pointer at the given position to point to the word from register C **unit tests:** - map a new segment to store our word in - load a value into a register, then call SSTORE to store this value in our segment at a specific offset/index - test in tandem with SLOAD → call SLOAD to load this value back into a register, then print it out to ensure it is the same thing we stored

	output: none, but the word in register C is now in segment A, offset B position in memory
ADD (3)	**input:** uint32_t instruction (with opcode 0011) **tasks:** - use array-indexing on the **registers array** to access/modify the values in each register - grab the values in register B and register C and store in local int variables - **add** the two values together, and store the result in register A **unit tests:** - load values into registers B and C - add them manually so we know the ground truth answer - call ADD instruction to load their sum into register A - print out the contents of register A and compare to ground truth answer **output:** none, but register A now contains the **sum** of the values in registers B and C
MULT (4)	**input:** uint32_t instruction (with opcode 0100) **tasks:** - use array-indexing on the **registers array** to access/modify the values in each register - grab the values in register B and register C and store in local int variables - **multiply** the two values together, and store the product in register A **unit tests:** - load values into registers B and C - compute the product manually so we know the ground truth answer - call MULT instruction to load their product into register A - print out the contents of register A and compare to ground truth answer **output:** none, but register A now contains the **product** of the values in registers B and C
DIV (5)	**input:** uint32_t instruction (with opcode 0101) **tasks:** - use array-indexing on the **registers array** to access/modify the values in each register - grab the values in register B and register C and store in local int variables - **multiply** the two values together, and store the product in register A **unit tests:** - load values into registers B and C - divide them manually so we know the ground truth answer - call DIV instruction to load their quotient into register A - print out the contents of register A and compare to ground truth answer **output:** none, but register A now contains the **product** of the values in registers B and C

NAND (6)	**input:** uint32_t instruction (with opcode 0110) **tasks:** - use array-indexing on the **registers array** to access/modify the values in each register - grab the values in register B and register C and store in local uint32_t variables - compute the **bitwise NAND** (by anding the values, then complementing the result) of the two values, and store the result in register A **unit tests:** - load values into registers B and C - NAND them manually so we know the ground truth answer - call NAND instruction to load the result into register A - print out the contents of register A and compare to ground truth answer **output:** none, but register A now contains the **result of bitwise NAND**-ing the values in registers B and C
HALT (7)	**input:** uint32_t instruction (with opcode 0111) **tasks:** - stops the execution cycle → call exit with the right exit code (exit_failure or exit_success ?) **unit tests:** - end each instruction set test with a halt instruction - write instruction test that is two halt instructions (ensure it stops after the first one) **output:** none
MAP (8)	**input:** uint32_t instruction (with opcode 1000) **tasks:** - build a new segment with the specified number of words, where every word/instruction is initialized to 0 - grab the value from register C and store in a local int variable → this value indicates the **size** of the segment (i.e. how many instructions it will hold) - call **seq_new** to build an instance of a sequence, initially empty - create a uint32_t variable of all 0's - push [size] zero's onto the sequence - if the **stack** of available segment ID's is **empty** → **append** a pointer to the new segment onto the end of the sequence of segment pointers. - if the **stack** of available segment ID's is **nonempty** → pop a value off the top of the stack and use the sequence **getter function**, passing in the recycled segmentID number as the index. this will return a pointer to where we want to store the newly mapped segment. assign this pointer to the address of the new segment. - store the newly mapped segment's ID number in register B **unit tests (C code):**

	- **map_multiple()** → function that will map multiple (10+) new segments consecutively. all of various sizes. - **map_unmap()** → function that will map a segment, then immediately unmap it, then map another segment. if we are properly reusing segment ID's, the two segments mapped should have the same ID number. - **map_empty()** → function that will attempt to map a segment of size 0 - **map_large()** → function that will map a really large segment **Output:** none
UNMAP (9)	**input:** unit32_t instruction (with opcode 1001) **tasks:** - grab the value in register C and store in a local int variable → this will be the **segment ID.** - use a sequence getter function, passing in the **segment ID** as the index. this will return a pointer to the segment to unmap - use a sequence freeing function to **free the segment** - set the pointer to the unmapped segment to **NULL** - push the segment ID onto the **stack** of available segment ID's **unit tests (C code):** - **unmap_all()** → maps a bunch of segments, then unmaps each of them. a we unmap each segment, assert that the pointers to each of the previously mapped segments is NULL. check segmentID stack to ensure that each segment's ID is getting pushed onto the stack as it is unmapped **output:** none, but
OUTPUT (10)	**input:** uint32_t instruction (with opcode 1010) **tasks:** - grab the value in register C (ensure it is between 0 and 255?) - write this value to standard output using fprintf **unit tests:** - load a value into register C - ensure this value gets printed to standard output - edge case: load a 255+ value into register C and ensure that we exit in the proper failure mode **output:** value from argument register written to stdout
INPUT (11)	**input:** uint32_t instruction (with opcode 1011) **tasks:** - use standard input to accept input value, and store in a local uin32_t variable - place the input value in register C - if the end of the input has been "signaled," place a uint32_t object of all 1's in register C **unit tests:** - call the input instruction and pass in a value through stdin

	- print out the contents of register C to ensure it is the value we entered - edge case: try passing in “signaled” input, whatever this means? **output:** none

LPROG (12)	**input:** uint32_t instruction (with opcode 1100)
	tasks:
	- grab the value in register B and store in a local int variable → this will be the **segment ID** of the segment being duplicated.
	- grab the value in register C and store in a local int variable → this will be the **offset of the program counter**
	- if the segmentID is 0, don’t do any duplication → adjust the position of the program counter (using a sequence getter function) and return.
	- otherwise, **free the sequence** currently being pointed to by the 0-segment. set the pointer at this position to null.
	- use a sequence **getter function** to get the segment to be loaded, passing it the segmentID number as the index. this will return a pointer to the segment to be loaded → **duplicate** this segment (create an empty instance and assign this sequence to it)
	- have the pointer at the 0-segment point to the address of the newly duplicated segment. have the program counter point to the instruction at the specified offset; this will be the next instruction ran.
	unit tests (C code):
	- **load_once()** → map a new segment, then immediately load this segment into the 0 segment. by default, have the program counter start at the first instruction of the new 0-segment.
	- **load_repeat()** → map a segment, load that segment, map another segment, load that segment, and repeat. by default, have the program counter start at the first instruction of the new 0-segment.
	- **load_unmap()** → map a segment, load the segment, then unmap that segment. by default, have the program counter start at the first instruction of the new 0-segment.
	- **load_offset()** → map a segment, load that segment, and have the program counter start at an offset in the middle of the segment.
	- in each unit test, we want to **avoid memory leaks**:
	  - make sure the segment we want to load is being properly duplicated and relocated to the 0-segment (i.e. make sure it does not still exist at its old location)
	  - make sure the segment we are replacing is properly abandoned (i.e. ensure the 0-segment
	output: none, but new program begins executing in 0-segment

LVAL (13)	**input:** uint32_t instruction (with opcode 1101) **tasks:** - use **bitpackgetu** to grab the **high 3 bits** immediately following the opcode and store in a local int → this value describes which **register** we will be loading into - use **bitpackgetu** to grab the low 25 bits of the instruction and store in a uint32_t local variable→ this value will be the value we load into the specified register - use array indexing to “load” the value into the register

	unit tests: - assert the high 3 bits are the load value opcode (13) - ensure that the value to be loaded is the correct value (test with smaller values, then larger) - loading 8+ values (i.e. some register values get overwritten) **output:** none, but contents of registers array get modified.

Note on Testing:
We intend to test each function individually as we implement them. We think that it would make sense to test the mapping functions together, so we can see that space is properly allocated and deallocated on our end. We will also test the arithmetic functions together with the load value function so that we can see that values of each register are manipulated correctly and that the loaded values are not interpreted differently in the computer. We will also test the segmented load and store functions together, so we can make sure the things are placed in the right spots and we are deallocating space when needed. We intend to do all of this via our unit_test framework we have implemented in our lab11–which gives us a .um file with what instructions we give the script.
Idea that we are still thinking about:
Putting all of the execution data structures in a struct.