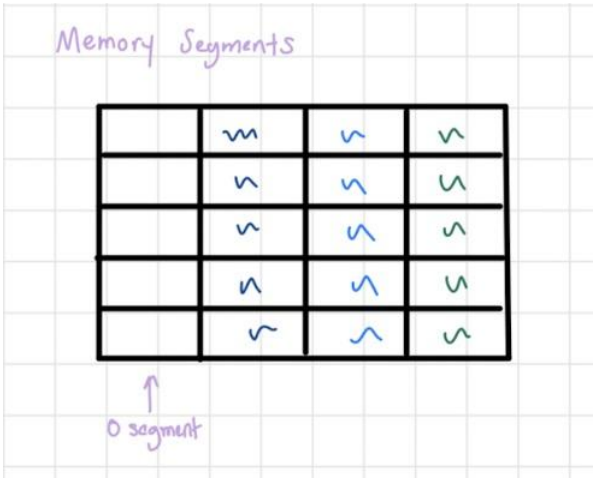
UM Design
Yoda Ermias (yermia01) & Maiah Islam (mislam07)

Architecture:

Hanson Data Structures:

One Sequence for Segmented Memory

- Hanson Sequence of type Hanson Sequence
- Each “large slot” in the sequence represents a memory segment
- Each slot within a larger sequence represents the offset



One UArray_T for the Registers

- All 8 registers will be housed in a UArray_T since we will always have 8 of them.
- Each index in our UArray_T of registers is a uint32_t word which represents the data held in a register

Modules

- Um.c → contains main that processes the command line and calls the program
- Run_um.c → One file that contains wrapper function
  - Declare segment sequence and register array here
  - Pass file pointer, instruction data structure of sequences into reading function (which was created in File_io.c)
- File_io.c → Responsible for reading the file and setting up the initial state of the UM

    - Read in words from file and add to sequence
    - Call wrapper function from um_instructions.c, pass in sequence and the array
    - This also contains a wrapper function (will be called in main, “run_um”)
- Um_instructions.c → Contains the implementation for unpacking a 32 bit word and the instructions (at this point the file has been read in)
    - Big wrapper function that takes the Segmented Memory Sequence and Register Array
    - Unpacking of 32 bit words
    - All 14 commands

Implementation Plan:

File Key:
= um.c
= um_instructions.c
= file_io.c

High Level Steps	Implementation Plan
Process command line and open file	1. Make sure correct number of arguments are provided from command line 2. If yes, pass file into run_um
Read words (instructions) from the file and add to 0-segment sequence Failures: ● At the beginning of a machine cycle, the program counter points outside the bounds of $m[0]. ● At the beginning of a machine cycle, the program counter points to a word that does not code for a valid instruction.	1. Open file and check for proper format 2. Declare memory sequence and register array 3. Populate sequence with 32 bit words from file 4. Once this is done, call the function to properly unpack the words and process the instructions
Unpack each word	1. Use the bitpack functions to take the relevant bits from the word. 2. Ensure that instruction and registers are in the proper range 3. Store the bits inside variables for the opcode, value (for loadval), register A, register B, and register C.
Process instruction	1. Call the necessary command

	functions for the specific opcode read. 2. The relevant parameters will then be passed in.

Plan for Instructions

Instruction	Implementation Plan
Conditional Move	if $r[C] ≠ 0 then $r[A] := $r[B] 1. Takes in registers A, B, C and checks if register C is not equal to 0. 2. If C≠0 then A := B
Segmented Load Failures: refers to a location outside the bounds of a mapped segment, refers to an unmapped segment.	$r[A] := $m[$r[B]][$r[C]] 1. Traverse memory sequence to find location [B][C] 2. Set register A to be what [B][C] contains
Segmented Store Failures: refers to a location outside the bounds of a mapped segment, refers to an unmapped segment.	$m[$r[A]][$r[B]] := $r[C] 1. Traverse the sequence architecture to find the segment index at index stored in $r[A]. 2. Traverse the segment for the instruction stored at the $r[B] index of the sequence 3. Store the information from $r[C] into the current location in the data structure
Addition	$r[A] := ($r[B] + $r[C]) mod 232 1. Add registers B and C and mod the result by 232 2. Load the result into register A
Multiplication	$r[A] := ($r[B] × $r[C]) mod 232 1. Multiply registers B and C and mod result by 232 2. Load result into register A
Division Failures: an instruction divided by zero.	$r[A] := ⌊$r[B] ÷ $r[C]⌋ 1. Make sure value in register C does not equal zero, proceed with division 2. Store result in register A
Bitwise NAND	$r[A] := ¬($r[B] ∧ $r[C]) 1. & the values in registers B and C 2. Not them 3. Store result in A

Halt	1. Returns EXIT_SUCCESS and terminates the UM
Map Segment	A new segment is created with a number of words equal to the value in $r[C]. Each word in the new segment is initialized to 0. A bit pattern that is not all zeroes and that does not identify any currently mapped segment is placed in $r[B]. The new segment is mapped as $m[$r[B]]. 1. Register B contains location of segment 2. Register C contains how long the sequence should be 3. Add 0 to sequence C number of times
Unmap Segment Failures: An instruction unmaps either $m[0] or a segment that is not mapped	The segment $m[$r[C]] is unmapped. Future Map Segment instructions may reuse the identifier $r[C] 1. The segment in the sequence at segment index $r[C] is cleared of data and size is set to 0.
Output Failures: An instruction outputs a value larger than 255.	The value in $r[C] is written to the I/O device immediately. Only values from 0 to 255 are allowed. 1. Mod value in register by 256 2. Print
Input	The universal machine waits for input on the I/O device. When input arrives, $r[C] is loaded with the input, which must be a value from 0 to 255. If the end of input has been signaled, then $r[C] is loaded with a full 32-bit word in which every bit is 1. 1. Value from input is converted into a value between 0 and 255 2. That integer is stored in register C. 3. If at the end of the input (AKA end of text input file), set to all 1s.
Load Program Failures: An instruction loads a program from a segment that is not mapped.	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]]. If $r[B] = 0, the load-program operation is expected to be extremely quick.

	1. Make sure segment $m[$r[B] exists. 2. Delete what is at $m[0] 3. Copy $m[$r[B] over into $m[0] 4. Reset counter to $m[0][$r[C]] and start executing from there. (also check that value in register C not greater than length of 0 segment sequence)
Load Value	One special instruction, with opcode 13, does not describe registers in the same way as the others. Instead, the three bits immediately less significant than the opcode describe a single register A. The remaining 25 bits are an unsigned binary value. This instruction sets $r[A] to that value. 1. Make sure register A is in range 2. Set register A to be the last 25 bits

Testing:

** If any of the failure modes are encountered, UM will exit with EXIT_FAILURE **

Map_Test:
We will make a map_error_check test function that takes the memory segment sequence and registers as input. We will call map_error_check with the invalid inputs described below and make sure the proper error/failure occurs.

- Check that the provided registers are within the correct range
    - If ((register C > 8 or < 1 or register B > 8 or < 1) → error
- Check if the segment already exists
    - if(r[B] > sequence length) → error
    - What should happen if you map a segment that already exists?
- Check if sequence is correct length after adding a segment
    - assert(seq.length >= r[B]) → error
- Check if segment is correct length after being added
    - Does it contain the number of words specified in register C?
    - assert(seq.at(r[B]).length == r[C]) → error
- Check to make sure not mapping 0 segment
    - assert(seq.at(m[0].length == 0) → error (will always be mapped)

Unmap_test:
We will make an unmap_error_check test function that takes the memory segment sequence and registers as input. We will call unmap_error_check with the invalid inputs described below and make sure the proper error/failure occurs.

- Make sure not unmapping 0 segment
- Check that the provided register is within the correct range
    - If (register B > 8 or < 1) → error
- Check to make sure segment exists (not greater than sequence length)
    - assert(r[B] <= seq.length)
- Check if segment is unmapped (seq length = 0)
    - assert(seq.at(r[B]).length != 0)

Load_test:
We will make a load_error_check test function that takes the memory segment sequence and registers as input. We will call load_error_check with the invalid inputs described below and make sure the proper error/failure occurs.

- Check to make sure registers are within range
    - If ((register C > 8 or < 1 or register B > 8 or < 1) → error
- Check to make sure register B within length of sequence
    - assert(r[B] = length of segment sequence)
- Check to make sure register C is within the length of the specific sequence
    - assert(r[B] = length of instruction sequence at m[r[B]])
- Check to make sure sequence at register B length is not 0 (it is unmapped)
    assert(seq.at(r[B]).length != 0)

* Fits into overall program: These error check functions will be called each time the specific command is called *

Further testing
All function testing will be done inside a separate testing file *playground.c*

- For each command, we will create a .um file that will test a single command. We will print out what was passed into each register, and make sure each register was updated with its correct value.
- For each function, we will make use of playground.c to test if the section of code written executes as intended.
    - Processing the Command Line -
        - Call UM with the incorrect number of arguments and double check if the proper CRT was thrown.
        - Call UM with incorrect inputs in the command line and double check if the proper CRT was thrown
    - Reading in the instructions -
        - Print out the instructions read from the file to stdout to ensure that things are read correctly.
        - The position of the current segment will be printed to ensure that we are inside the 0th-segment
        - We will test the failure case by setting the counter pointer to outside the 0th segment when starting the UM

        - We will test the failure case by providing an instruction that isn't valid to check if a CRT is raised
    - Unpacking word
        - We will print out the unpacked instruction and compare the output to what the opcode, value, and registers are supposed to be.
    - Processing Instructions
        - We will make use of print statements inside the command functions to see if they are being called properly after our instructions are unpacked
        - We will print out the passed in values to ensure that they are passed correctly into the command functions