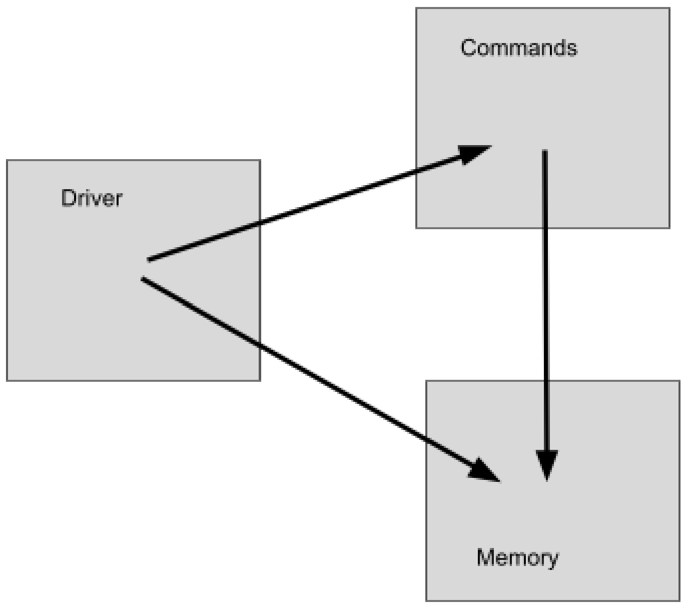
Homework 6 design doc
Addison Mirliani and Jake Seidman
4/12/21

Architecture



*An arrow represents one module using another’s interface

- Driver (main)
    - Contains main function
    - Reads in all input commands
        - Maps segment 0
        - One at a time reads in command words from stdin and stores them in segment 0 using segmented store
    - Calls a function in the command module that starts a command loop
- Commands (command.h)
    - Contains a function with a command loop
    - Contains functions which implement each command
    - Has a function that runs gets the current command word
    - Unpacks the op code of the instruction and passes it to the appropriate function to run the command
    - Interface used by other modules:
        - void runCommandLoop()
            - Called by the driver module
            - Repeatedly gets the next word from segment 0 and calls runCommand

            - runCommand() unpacks the op code (and ABC if applicable) and passes these values and the word to the appropriate function
- Memory (memory.h)
    - Contains an array of 8 registers
        - Each register is a uint32_t
    - Each segment is an array of uint32_t words and all segments are stored in a sequence of arrays
    - Segments can be mapped or unmapped using the map and unmap, which are called by the driver
    - Each individual segment is an array of uint32_t words
    - Uninitialized words contain 0
    - Program instructions are stored in m[0]
        - Contains a program counter which tracks the current instruction being run
    - Has a get_current_command() function that returns the next word to execute in segment 0
        - Increment the counter by 1
    - Has a set_command_counter() function that sets the command counter
    - Has functions that can add/get values to/from a given segment at a given index
    - Contains a mapSegment() function
        - Loops through sequences of arrays until one of the pointer is NULL
        - Once one is found add a new segment of length r[c] at that spot
            - If no NULL is found, add the new segment at the end
    - Contains an unmapSegment() function
        - Go to the index given and free the array at that index
        - Set the value in the sequence to NULL
    - Interface used by other modules:
        - uint32_t mapSegment(uint32_t length)
            - Called by Driver and Commands
            - Iterates over sequence of segments until an unmapped index is found
                - If no unmapped index is found, adds a segment on the end of the sequence
            - Allocates memory for an array of uint32_t, with a length equal to the length provided
            - Returns the index that the segment was mapped to
        - void unmapSegment(uint32_t index)
            - Called by Driver and Commands
            - Frees the segment at the given index
            - Sets the element in the sequence to be NULL
        - void setWord(uint32_t segmentID, uint32_t wordID, uint32_t word)
            - Called by Driver and Commands
            - Sets the word at the given wordID in the given segment to the given word

- uint32_t getWord(uint32_t segmentID, uint32_t wordID)
    - Called by Commands
    - Returns the value of the word at the given segmentID and wordID
- void freeAllMemory()
    - Called by the Driver
    - Iterates through our sequence of arrays and frees each array
    - Frees the sequence
    - Frees the array of registers
- uint32_t getRegister(uint32_t index)
    - Called by Commands
    - Returns the value stored in a given register (index 0-7)
- void setRegister(uint32_t index, uint32_t value)
    - Called by Commands
    - Sets the value of the given register (index 0-7) to the specified value
- void loadProgram(uint32_t segmentID, uint32_t counterPosition)
    - Called by Commands
    - Duplicates the segment at the segmentID and puts it into segment 0
    - Resets the position of the counter to the given position

Implementation Plan

- Write the makefile, include all necessary libraries
    - **Test:** Compile and run with “Hello World” program
- Write a function in main that reads in 1 binary command from stdin, and print it out
- Make a memory.c and memory.h file and compile
- Create a sequence of arrays that will hold each segment
- Use stat to find the size of the file
    - **Test:** Print out the size in bytes and check its correctness
- Write a function that creates a segment (sequence) of uint32_t of length n
    - Adds 0 to the segment n times
    - **Test:** Iterate through the segment and print the zeroes to make sure it’s the right size
- Call this function to create sequence m[0] of size (# of bits)/32 which will store the program instructions
- Make a setWord() function that stores a word at the specified location within a segment
    - Takes in an index for which segment to add to and the number segment
    - **Test:** Make test segments and try inputting words to different indices
        - Try putting a word at index zero, index length - 1, and an index that is out of bounds (should result in an error)
- One at a time, read in all words from stdin, and store them into segment 0 using this function.
    - Use a counter to keep track of the index of where they should be added
    - **Test:** Print out all words after they have been read in
- Make a function that retrieves a word from a specified location in memory called getWord()
    - Takes in a segment number and a position within that segment
    - **Test:** Make a test segment and test our ability to retrieve different values
        - If the word is empty, we get zero back
        - If the word is out of bounds, we get an error
        - Test that we can get words at [0] and [length - 1] within a segment
- Make a function loadProgram()
    - Write a load_program() function in memory.c
        - This function duplicates a segment
        - It then frees m[0] and replaces it with the duplicated segment
        - Change the counter to the location specified by the user
- Make a function mapSegment()
    - Write a function in memory.c that loops through sequences of arrays until one of the pointer is NULL
        - Once one is found add a new segment of length r[c] at that spot
            - If no NULL is found, add the new segment at the end
        - Add the index of the segment to r[b]
- Make a function unmapSegment()

    - Write in memory.c that goes to the index in r[c] and frees the segment at that index, and sets the value in the sequence to NULL
- Make a command.c and command.h file and compile
- Make a function that retrieves the first 4 bits of the 32-bit words (use bitpack interface)
    - **Test:** Pass in binary commands and check that we print out the correct instruction number
        - Check that we get an error if the command is not in the range 0 to 13
- Update the function so that if the instruction is 13, we call the load value function (currently empty)
- Update the function so that it unpacks the A, B, and C values if the command is not load value (command 13)
    - **Test:** Pass in different words with a variety of command, A, B, and C values and check that we print out the correct values
- Make a function in main that starts the command loop, which will be handled in the commands module
- Make a run_command() function that calls each function depending on the command number
    - **Test:** Check that if the command is an invalid number (outside the range 0-13), the program fails
    - **Test:** Make a function for each command and have each print which function it is
        - Run a program and just print each command that we should execute
- The command loop loops through all of the words in segment zero and calls run_command for each command
    - **Test:** Print each command and A, B, and C values while looping through
        - Try passing in some of the example files given to us to see that we can read the binary correctly
- Implement the commands in command.c in the following order **(see testing below).**
    - Halt (7)
        - Make a function in memory.c that frees all memory in the memory module
        - Call this freeAllMemory() function in the halt function
        - Call exit()
    - Load value (13)
        - Write a function that unpacks the register number and the value given the input word
        - Set the register r[A] to the given value
    - Map segment (8)
        - Write a function in command.c that calls the mapSegment() function in memory.c
    - Unmap segment (9)
        - Write a function in command.c that calls the mapSegment() function in memory.c
    - Output (10)
        - Write a function that prints out the value in the given register as an ASCII character

    - Input (11)
        - Use scanf(“%c”, &c) to read in a single character from stdin and load it into r[c]
        - If c is a newline character, then load ~0 into r[c]
    - Conditional move (0) and Bitwise NAND (6)
        - Implement the logic commands as specified in the spec
    - Segmented load (1)
        - Call the function get_word() in memory.c for the specified indices
    - Segmented store (2)
        - Call the function set_word() in memory.c for the specified indices and pass in the value to set
    - Addition (3), Multiplication (4), and Division (5)
        - Implemented the arithmetic commands as specified in the spec
    - Load program (12)
        - Call the loadProgram() function in memory.c
    - **Test:** See the implementation plan below for the unit testing for each function
    - We chose this order of implementation so we could use previously implemented commands to test the later ones

Architecture Testing

During testing we will temporarily make our sequence of arrays accessible through the .h file. This is only for testing direct access and we will remove it after testing.

- **void testMapSegment()**
    - Test that we can map segments of the correct size
    - Call the **mapSegment()** function in memory.c
    - Use **sizeof()** on the array of words to determine that the segment was created with the correct size
    - Iterate through the segment by directly accessing the sequence of arrays, and assert that each value in the segment is 0

- **void testUnmapSegment()**
    - Test that we can unmap segments at the specified locations
    - Call the **mapSegment()** function to map 4 segments
    - Now call the **unmapSegment()** function to unmap segment 2
    - Access the sequence directly and confirm that the pointer in the corresponding index of the sequence is NULL
    - Check with valgrind to ensure there is no memory loss

- **void testSegmentIndices()**
    - Map 5 segments by calling **mapSegment()**.
        - Directly access the sequence of arrays and make sure that they have all been mapped correctly indices 0-4.

    - Unmap segments 2 and 3 using **unmapSegment()**
        - Check that segments 2 and 3 have correctly been set to NULL by directly accessing the sequence of arrays
        - Check that segments 0, 1, 3, and 4 are still mapped by directly accessing the sequence of arrays
    - Map 3 additional segments using **mapSegment()**
        - Check that segments 0-5 are correctly mapped by directly accessing the sequence of arrays
- **void testSetWord()**
    - Tests that the setWord() can correctly assign a given value to a given segment at a given word index
    - Call **mapSegment(256)** to map an empty segment (initialized to all 0s) of length 256
    - Use a for loop with i ranging from 0 to 255 to set the value of each word in the segment equal to its index
        - i.e. segment[5] = 5
        - Do this by directly accessing the segment
    - Iterate over the segment a second time, convert each binary value to decimal, and print them out
    - Confirm that the values print out are the sequential numbers 0,1,2 ,3… 256

- **void testGetWord()**
    - Tests that **getWord()** can correctly retrieve a word at a given segment and word index
    - Use **setWord()** to set all values in the segment to their indices and then use **getWord()** to access each of these words and assert than each is equal to its index
- **void testSetRegister()**
    - Tests that **setRegister()** can successfully set the correct value in the register array
        - Temporarily make the register array public so that we can directly access it and check that we can change the values
- **void testGetRegister()**
    - Tests that **getRegister()** can get the correct values of the registers
    - Use **setRegister()** to set each of the values in the array of registers equal to its index
    - Use **getRegister()** on each register and assert that its value is equal to its index
    - Check that when **getRegister()** is called on an empty register it returns 0
- **void testLoadProgram()**
    - Tests that **loadProgram()** can correctly move a segment to segment zero and begin executing commands starting at a specified location in the segment
    - Use **mapSegment()** and **setWord()** to create a segment and add commands to it
    - Next, use **loadProgram()** to replace the current segment 0 with our newly created segment

    - Test that the commands we wrote in our new segment are executed starting at the counter value we give (check that this works if the counter value is in different spots)
    - Check for failure if the counter points somewhere outside of the segment
- **void testFreeAllMemory()**
    - Tests that **freeAllMemory()** can successfully pass a valgrind test and free all of the memory
    - Use **mapSegment()** and **setWord()** to set up multiple segments that contain memory
        - Call **freeAllMemory()** to check that this is freed
    - Use **setRegister()** to give the registers different values
        - Call **freeAllMemory()** to check that this is freed

Unit Testing

All of our unit tests will be put into .um files using umlabwrite.c. We will add unit tests to our array of tests to build and add the expected output for each. This will create .1 files containing the expected outputs of our tests. We will then write a shell script that runs each of the .um tests on our machine and puts the output in the appropriate .out files. We can compare each .out file to its corresponding .1 file in our shell script and print to terminal whether each test has passed or failed.

- Command 0 -> Conditional Move
    - If $r[C] = 0, return
    - Use output to check that we have done this command correctly
        - Output the value in each register
- Command 1 -> Segmented Load
    - Load a variety of values from several different (mapped) segments
        - Load values that have been assigned, and some that have not been assigned
    - Check that words that have been mapped but not assigned contain 0
    - Test in conjunctions with segmented load to ensure that the values returned match the value that were stored
- Command 2 -> Segmented Store
    - Store a variety of values into several different (mapped) segments
    - Test in conjunctions with segmented load to ensure that the values returned match the value that were stored
- Command 3 -> Addition
    - Load values into registers 0, and 1 using load value
    - Run the addition command, and check to see if register 2 contains the correct sum
    - Repeat with different value in registers 0 and 1
    - Repeat with different indices of registers

    - Run with numbers that will overflow above 2^{32}, and check if they are correctly modulused
- Command 4 -> Multiplication
    - Load values into registers 0, and 1 using load value
    - Run the multiplication command, and check to see if register 2 contains the correct product
    - Repeat with different value in registers 0 and 1
    - Repeat with different indices of registers
    - Run with numbers that will overflow above 2^{32}, and check if they are correctly modulused
- Command 5 -> Division
    - Load values into registers 0, and 1 using load value
        - Test with 0 in the second register. Make sure that our program correctly fails when this occurs.
    - Run the multiplication command, and check to see if register 2 contains the correct quotient.
        - Decimal quotients should be rounded down. Ensure that this truncation is happening correctly
    - Repeat with different value in registers 0 and 1
    - Repeat with different indices of registers
- Command 6 -> Bitwise NAND
    - Use register 0 and 1 as input registers, and register 2 as output register. Test with the following inputs, and compare to the following expected outputs

Register 0 (decimal)	Register 1 (decimal)	Register 2 (decimal)
5	10	2^{32} - 1
1	1	2^{32} - 2
2^{32}-1	2^{32}-1	0
15	7	2^{32} - 8

    -
- Command 7 -> Halt
    - Stop executing the program and return
    - First, try having a halt command and no other code in our program
    - Check that we can halt at any time and that no code after a halt will be executed
    - Map several segments, and then umap them before calling halt
        - Run with valgrind to ensure that all memory have been freed correctly
- Command 8 -> Map Segment
    - Call the function in memory.c that creates a segment of the specified length with all zeroes
    - Classify this segment as mapped in memory.c by adding a 1 in this index to the mapped array
        - Test this by printing out the mapped array

    - Print out each of the values in the segment to test that they are all zero and the correct length
        - Try this with different lengths
- Command 9 -> Unmap Segment
    - Classify this segment as unmapped in memory.c by adding a 0 in this index to the mapped array
        - Test this by printing out the mapped array
    - Free the sequence
        - Test this with valgrind
- Command 10 -> Output
    - Load different values into all registers. Try to output the values in each register
    - Test with values greater than 255 to see if the program correctly fails
- Command 11 -> Input
    - If the input is empty, check that it stores a 32-bit value of all ones
    - Use output to print each value we read in
        - Try 0 and 255
        - Try calling input multiple times and reading in multiple characters
- Command 12 -> Load Program
    - Try running a program with an expected output and then calling load program and using a previously-written test program
        - Check that both outputs are written to the .out file
- Command 13 -> Load Value
    - Unpack the values of the register number and the value
    - Print out the value in the specified register before and after this command is called
    - Check that we can load very large and very small values

Command Unit Tests

1. Halt
2. LoadValue(0,65), Output(0), Halt
    a. Should output “A”
3. LoadValue(0, 30), LoadValue(1, 36), Add(2, 1, 0), Output(2), Halt
    a. Should output “B”
4. LoadValue(0, 34), LoadValue(1, 2), Multiply(2, 1, 0), Output(2), Halt
    a. Should output “D”
5. LoadValue(0, 198), LoadValue(1, 3), Divide(2, 1, 0), Output(2), Halt
    a. Should output “D”
6. LoadValue(1, 67), LoadValue(0, 65), Conditional Move(0, 1, 2), Output(0), LoadValue(2, 1), Conditional Move(0, 1, 2), Output(0) , Halt
    a. Should output “AC”
7. LoadValue(0,7), LoadValue(1,8), NAND(2, 1, 0), LoadValue(3,68), Add(2,2,3), Output(2), Halt
    a. Should output “C”
8. Input(0), “A”, Output(0)
    a. Should output “A”