Jordan Pittigano (jpitti01) and Javier Gonzalez (jgonza20)
COMP 40
HW 6: um
Design Document

Architecture

Modules (1)

NOTE: every .c file will have a corresponding .h file containing the function declarations and the libraries used in the respective .c file

um.c (the program's driver)

- This file will contain the main implementation of the code (the main function)
- Going to have the main function. Here we will:
    - Initialize the UM to the initial state described in 3.3 in spec
    - An execution cycle (3.4) that executes instructions one by one
        - This function will read in the struct returned by loadProgram.c and will run until we run out of instructions
        - We will have a separate case-switch base function to know what operation to execute
            - We will make use of bitpack.c to get the opcode from the instruction
        - Call execute_instruction on each instruction
        - This will call the according function from instructions.c
    - Return Exit_success if successful

memory.c (segmented memory utilization)

- This file will be in charge of all of the segment-related usage and will replicate the use of segmented memory
- It is a separate file because the user should not know the details about memory
- Contains the Memory struct
    - This struct will contain a sequence of the segments that represent memory, a sequence with all of the unmapped segment IDs and the program counter to serve as the index for the sequence of segments
        - Each segment will be represented as another sequence that will contain whatever the segment contains (eg: $m[0] the instructions)
- Will contain functions such as segment_map(), segment_unmap(), segment_free(), segment_new(), segment_at() and others that we may find useful to include

loadProgram.c (read in and store the information relating the program to run)

- This file will contain a single function that will be in charge of populating the $m[0]
    - We will use bitpack.c to pack the instructions according to the read opcode's format and add it to the $m[0] segment
- Return an instance of the struct Memory containing all of the information read

instructions.c (contain the definitions of the operator functions)

- Going to contain the 14 operator definitions
    - We will contain a struct called Instruction storing the opcode and an uint32_t word containing the instruction itself
    - We will also contain a function called Register containing simply the array of 8 registers to hide them from the client
- Three_register() and loadval()
    - These latter functions will make use of the "bitpack.h" to unpack the instructions

segment_test.c (test the functionality of our segmented memory)

- This file will be sort of a unit_test file but specifically for the functions defined in Memory.c
    - We will have tests for each and every function defined that related to the segmentation and will ensure that the memory works as expected

<u>Data Structures in our modules (2)</u>

Instruction Struct → Defined in um.c

- Contains the information that we have to pass as parameters for the operators
    - Uint8_t Opcode
    - Uint32_t word

Registers Struct → Defined in um.c

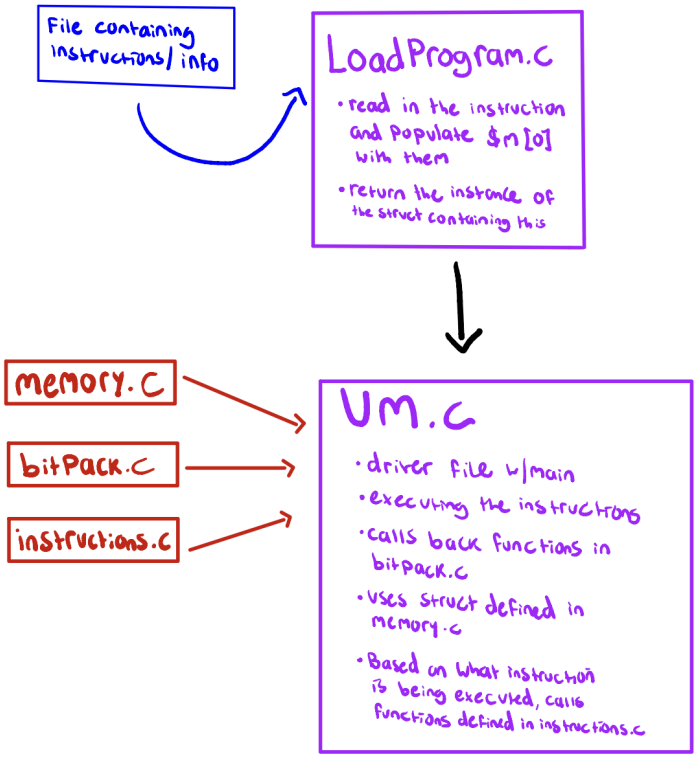
- Contains an array of the 8 registers
    - Will help us with abstraction as we will hide this information from the client
    - We will also pass this as a parameter for the operators

Memory Struct → Defined in memory.c

- Will have a Hanson Seq of the segments that represent memory
- Will have another Hanson Seq with all of the unmapped segment IDs
    - The sequence of unmapped IDs will store a sequence of indices corresponding to the segments that have been unmapped to avoid for unnecessary creation of new segments if there is an unused one

- Will have the program counter to serve as the index for the sequence of segments
- Each segment will be represented as another sequence that will contain whatever the segment contains (eg: $m[0] the instructions)

How our modules interact (3)



Functions that call back other modules (3)

```
/* Name: extract_instruction
 * Description: makes use of bitgetu() to get the opcode packed in each instruction
 * Parameters: a 32-bit packed word containing the instruction
 * Returns: an int with the opcode that represents instruction in the enum defined
 * Expects: an opcode to be stored in the 4 MSB of the word
 * Notes:
 */
int extract_instruction(uint32_t word)
{
        /* use bitgetu(word, 4, 28) and store it as an int */
        /* return that value so that we can execute the instruction */
}
```

```
/* Name: execute_instruction
 * Description: checks what opcode extract_instruction read in to execute the instruction
 * Parameters: the opcode returned by extract_instruction
 * Returns: nothing
 * Expects: expects a value that lies between 0-13
 * Notes:
 */
void execute_instruction(int operator)
{
        /* this function is based on a case-switch */
        /* "operator" will be compared to any of the 14 values in the enum and the matching
            function will be called with its corresponding parameters */
        /* we will call functions in instructions.c where all of our instructions will be defined */
}

/* Name: loadVal
 * Description: reads in all of the instructions read in from the input file
 * Parameters: FILE *fp (file pointer) and the uint32_t num_instructions to know what size to
 * create the sequence of (we'll do some research on built in function that allow us to measure
 * this based on the size of the file)
 * Returns: a Memory struct with its $m[0] populated with all of the instructions
 * Expects: a valid file pointer
 * Notes: will assert the fp before using it
 */
loadVal(FILE *fp, uint32_t num_instructions)
{
        /* create an instance of the Memory struct and initialize the sequence 0 */
        /* loop through the file while using getc() taking into account that the order of storage is
        in big-endian */
        /* store these instructions in the $m[0] */
        /* return the instance of the struct with the program loaded with instructions */
}
```

Implementation Plan

NOTE: we will be testing by compiling after each step and running specific tests to make sure our UM is on the right track. The specificity of our tests is detailed in the testing section

4/3

1. Set of all of the files mentioned above and make sure that everything compiles correctly **(20 minutes)**
    a. Create the structs in their according files and make sure to include the libraries we are going to use in each of the steps of the UM
2. Code the loadValue function **(2 hours)**
    a. Make use of the memory struct and create the function to read the file containing the instructions and populate the 0 index in the sequence of segments with them
    b. Do some research to check how to find the size of the segment we are going to have to allocate as the segment is created with a known size

4/4

3. Set of the main function in the um.c file **(2 hours)**
    a. Initialize the state of the UM according to the spec (3.3)
    b. Create the loop that is going to iterate through the different instructions stored in the $m[0]
4. Code the functions in charge of determining what the opcode is and return the corresponding integer value **(20 minutes)**
5. Read in that opcode and input it as a parameter for the function that will call the instruction (the case-switch based) **(30 minutes)**
    a. Check that the operation read is the one expected

4/5

6. Start working on the segmented memory representation. **(4 hours)**
    a. Functions such as Segment_new(), Segment_free(), Segment_map(), Segment_unmap(), Get_content() - to get the information stored at a specific segment of memory) - and maybe some other to add functionality

4/6

7. Code the three_register() and the loadVal() functions to allow us to be able to run the other 14 operators **(10 minutes)**
    a. Use um lab to test this
8. Code the 14 operator functions **(3 hours)**
    a. Will test each operator, before moving onto coding another one

4/7

9. Test the program on small cases **(1 hours)**
    a. Create some tests with the lab_um machine and run them on our program
10. Test the program on the larger given .um files (games…etc) **(1.5 hours)**

4/8 - 4/11

11. Build tests for every function of the program that may need it **(4 hours)**
12. Debug the program **(Multiple hours)**

Testing Plan

The UM instruction set

- We will create a set of 1-2 tests for every operator function that we define as soon as we finish it
- To do this we will include out um.c in the lab_um.c file to be able to access our information
    - We will then create some tests here and we will run the executable write_tests to create the .um files
    - We will then use our run_tests.sh to verify the functionality of the code done

UM segment abstraction

**Note: every testing function will make use of the file Memory.c and will be called in the own file's main function*

- For this we will create a file called segment_test.c
- segment _map_test() // the creation of a new index of the segment and adjust in the size of the new sequence
- segment _unmap_test() // check that the contents on the index wanted have been deleted and the index has been correctly added to the unmappedID section
- segment _new_test() // check for the creation of a new sequence of segments
- segment _free_test() // check for the correct deletion of the whole sequence of information and that the information has been freed correctly
- segment_at_test() // check that we are able to access a specific segment anytime we call its index with this function
- segment_print_test() // to check these functions with more information (not only getting the size of the sequence returned (for example), we will create this function that will iterate through each segment printing its contents

Testing checked errors and failures

- Um executable called in a way that violates its contract:
    - Stderr message
    - EXIT_FAILURE
- If the UM program dementia resources that our program cannot provide and it is not an allowed failure, we have to halt() and c.r.e

Edge cases we thought about

- Halt after the execution loop in case the last instruction wasn't a halt
- Empty file
- Just read one instruction
- If we map a segment without having a segment_new call before
- Same with unmap or any other Memory.c function