Homework 6: Design Document | Leila Minowada & Drake Nguyen

Modules:

- `instructions.h`
- `bitpack_um.c`
- `write.c`
- `segments.c`
- `memory_seg.c`
- `get_instructions.h`
- `um.c`

Abstractions:

Instruction module: `instructions.h, instructions.c`

- Functions: implementations for each of the 14 UM instructions
    - parameters: registers a, b, and/or c, value for load_value instruction
    - calls/executes actual instructions using memory_seg interface
- Secret: how to execute each of the 14 UM instructions

Bitpack module: `bitpack_um.h, bitpack_um.c`

- Contains: functions that bitpack, check, and remove bits
- Functions: same functions as bitpack from arith, except these work on a 32-bit vector, not a 64-bit vector
    - `bool Bitpack_fitsu(uint32_t n, unsigned width);`
        - checks if num will fit in unsigned 32 bit int
    - `bool Bitpack_fitss(int32_t n, unsigned width);`
        - checks if num will fit in signed 32 bit
    - `uint32_t Bitpack_getu(uint32_t word, unsigned width, unsigned lsb);`
        - gets specific bits out of unsigned word
    - `int32_t Bitpack_gets(uint32_t word, unsigned width, unsigned lsb);`
        - get specific bits out of signed word
    - `uint32_t Bitpack_newu(uint32_t word, unsigned width, unsigned lsb, uint32_t value);`
        - get new unsigned int
    - `uint32_t Bitpack_news(uint32_t word, unsigned width, unsigned lsb, int32_t value);`
        - get new signed int
- Secret: how to bitpack opcode, registers, and values into a UM instruction as well as how to unpack them from instructions given

Outputting module: `write.h, write.c`

- Contains: emit, write functions that use bitpack to print in big endian order, an I/O device capable of displaying ASCII characters and performing input and output of unsigned 8-bit characters
- Secret: how to write instructions to output and deal with streams

Segment module: `segments.h, segments.c`

- Contains: how to create a segment: made out of a Hanson sequence of uint32_t words
- Functions:
    - `Seq_T create_segment(int num_words);`
        - Given how many word, creates a Hanson sequence of unsigned 32 bit words
    - `void delete_segment(Seq_T seg);`
        - frees this specific segment
    - `uint32_t get_word(Seq_T seg, uint32_t id);`
        - get word (32 bit unsigned int) at certain id
    - `void insert_word(Seq_T seg, uint32_t id, uint32_t value);`
        - inserts word into spot
- Secret: how to make/delete a segment, what it is composed of, and how to change/get words

Memory segment module: `memory_seg.h, memory_seg.c`

- Contains:
    - Implementation of an incomplete struct representing the UM's segmented memory. The struct contains a sequence of in-use segments and sequence of unmapped ids
- Functions:
    - `Memory_T map_seg(int num_words, Memory_T memory);`
        - returns to user struct with updated segment sequence
    - `Memory_T unmap_seg(Memory_T mem, uint32_t segid);`
        - returns to user struct with updated segment and id sequence
    - `uint32_t load_value(Memory_T mem, uint32_t segid, uint32_t offset);`
        - returns to user value at specific segment/word
    - `Memory_T store_value(Memory_T mem, uint32_t value, uint32_t  segid, uint32_t offset);`
        - returns to user struct with updated segment sequence
    - `Memory_T initialize_memory();`
        - initializes id sequence and segment sequence as well as returns struct with seq of ids (0) and seq of segments (seg 0 is initialized)
    - `void free_memory(Memory_T);`
        - frees sequences in struct

- Secret: how to deal with segments and unmapped ids when creating/deleting, as well as how registers are created and edited in memory

Register module: `registers.h, registers.c`

- Contains: implementation of the 8 registers of a Universal Machine, implemented as an array of 8 integers
- Functions:
    - `void put_reg(Memory_T, uint32_t reg_id, uint32_t value);`
        - put value in a specific register, returns void
    - `uint32_t get_reg(Memory_T, uint32_t reg_id);`
        - get value from a specific register, return as uint32_t

`get_instructions.c`

- Contains: wrapper functions to unbitpack the UM instructions and call correct instruction execution
- Functions:
    - wrapper functions to unbitpack UM
    - function to ID the correct instruction implementation to call
        - uses the instructions.h interface to execute the instruction
- Secret: how to decide which instruction to run, and get all the information out of a bitpacked instruction

`um.c`

- Contains: main function that runs the UM
- Functions:
    - uses memory_seg.h interface to implement initial state of UM
    - uses write.h interface to take in file/stdin of instructions
    - uses memory_seg.h interface to start running the 0 segment instructions
    - uses get_instruction.h interface to unbitpack instruction and call correct execution
    - uses write.h interface to print correct output
    - holds 32 bit program counter
- Secret: the running code and 32 bit program counter

Architecture:

The main will call the overall functions in our UM. It will interface with memory_seg.h to implement the initial state of the UM, as well as with write.h to read in a file/stdin to get instructions for the UM to run. Memory_seg.h will also be used to run the 0 segment instructions for the whole time the program is running in the UM. get_instruction.h interface will be used to unbitpack and instruction. To unbitpack, get_instruction.h will interface with bitpack_um.h, which is a modified version of our bitpack.h from arith, but modified to work with the 32 bit ints we are using for our UM. get_instruction.h will then interface with instructions.h in order to execute the instruction on memory. instructions.h will interface with memory_seg.h to change the memory representations to implement the instructions. If necessary, it interfaces with write.h to write to the correct output. Finally, the main iterates its program counter, and moves onto the next instruction.

Segmented Memory Implementation:
Our segmented memory implementation is made up of Hanson's sequences. Each segment is a sequence of 32 bit words, with a size specified by the instruction given. The implementation of a single segment is held in `instructions.h`. In `memory_seg.h`, a struct `memory` is defined, which contains a sequence of segments, a sequence of 32 bit unsigned ints, and an array of 32 bit ints. The first sequence holds all of the segments that are currently in use. Since each segment is technically a sequence, this structure is technically a sequence of sequences. The second sequence is used to keep track of each of the segment IDs that have been unmapped. When a segment is unmapped, its position in the sequence of segments is pushed onto the back of the sequence of IDs, and its sequence of 32 bit ints is freed, but it is not removed from the sequence of segments, thus keeping the position of all the other segments the same. When there is a call to map a segment, we first check to see if there is a segment ID that has been unmapped, and create the segment at that spot in the sequence of segments, instead of creating the segment at the back of the sequence of segments.

Invariants:
- segment 0's instructions are always being run
- if we switch out code in segment 0, it immediately resets the program counter to 0 and starts executing the next instruction
- there are always 8 registers of 32 bits, which are set to 0 initially
- an instruction cannot be executed without unbitpacking it to get the individual parts first
- instructions are read in big endian order
- the operator is in the 4 most significant bits
- registers/values cannot be unbitpacked until the operator has been unbitpacked, so that we know how many registers to unbitpack, and whether to unbitpack a value
- values to load cannot be larger than 2^25
- cannot have more than 3 registers in one instruction
- can only output a value between 0-255
- input must only be a value between 0 - 255

Testing:
general:
- To unit test, we will use the umlabwrite.c/ umlab.c programs that we edited in lab in order to decouple the UM from the program loader/main and test specific parts.

  - an example of this is our emit-print6-test, which should print the ascii character 6 if the UM executes correctly. It relies on the emit() function, which removes all the instructions from the sequence of instructions created and outputs the instruction to a stream, which is then executed by our UM.

    ```
    void emit_print6_test(Seq_T stream)
    {
            emit(stream, loadval(r1, 48));
            emit(stream, loadval(r2, 6));
            emit(stream, three_register(ADD, r3, r1, r2));
            emit(stream, output(r3));
            emit(stream, halt());
    }
    ```
  - To be sure that our UM runs correctly, we will test it with a program that only executes one of the instructions and a halt, against our expected output, and assure that each individual instruction works properly. We will also test all of the individual instructions with all of the different registers to make sure that it works with all of them. We will also test with registers that are 0, as well as with registers that already have a value in them. Once we have tested all of our instructions individually and with specific edge cases, we will test our UM against the benchmark programs provided to be sure that it runs correctly as a whole.
- conditional move:
  - test with register C = 0
  - test with register C != 0
- division:
  - test with division without a remainder
  - test with division that has a remainder
  - Test with division by 0 and expect an error
- bitwise NAND:
  - test with all 0's and all 1's, as well as mix
- halt:
  - test halt at beginning, middle, and end of a program
- map segment:
  - test when there are IDs that have already been unmapped
  - test with no IDs having been unmapped
  - test with word length of 1
  - test with word length of 0
  - test with word length = 2^32
  - test with bit pattern that is all 0's
  - test with bit pattern that represents already mapped segment

    - After mapping, print out all words to see if they are initialized to 0
- unmap segment:
    - test with segment that has not yet been mapped/has already been unmapped and expect nothing to happen
    - test with segment that has word length of 1
    - test with segment that has word length of 2^32
- Output
    - Output a value greater than 255 or smaller than 0 and expect an error
    - Output a value within the range of 0-255
- Input
    - Input a value out of the range 0-255, expect an error
    - Input a value within the range 0-255, print out the value in r[c] to check
    - Input a EOF, expect r[c] to be loaded with a full 32-bit vector of 1's
- load program
    - Pass in r[b] = 0
    - Pass in r[b] != 0
- load value
    - Load a value of size 25 bits
    - Load a value of size more than 25 bits and expect an error
- Additionally, we can test instructions in pairs by pairing up functions that do opposite task such as multiplication/division, input/output