hw6: um

with matt (mung01) and liam (lstran01)

Architecture

Modules:

main.c - command-line-args and driver

- [ ] Parse command line and extract filename (through prepare module)
- [ ] Pass filename to um_state
- [ ] Check command line arguments for proper format (i.e. 2 arguments)

um_state - machine state, control loop

- [ ] Instantiates and maintains the state of the universal machine
- [ ] Memory allocation and subsequent errors handled through hanson alloc
- [ ] The um is represented by several data structures
    - [ ] The 8 4-byte registers are represented by an 8-element array of uint32s
    - [ ] The program counter is represented by a uint32 on the stack
    - [ ] The 0-segment is represented by a standard c-array of uint32s of the appropriate size, on the heap.
    - [ ] The other segments are represented by a Hanson sequence of uarrays of uint32s. Each uarray is one segment in memory, indexed at its segment number.
    - [ ] When segments are unmapped, the memory associated with the segment is recycled, and the index is placed in a uarray of uint32s,

prepare - parse input of .um file and fills segment zero

- [ ] Uses a stat system call to get the size of file to instantiate segment zero with the proper size for the amount of instructions in file
- [ ] Segment Zero is on the heap and the pointer to array of segment zero is returned from the prepare module
- [ ] Parses input um file to fill segment zero with code word instructions
- [ ] Error Checks for invalid um file and violation of um contract (non-existent or invalid access to file)

instructions - executed by control loop for each of the commands

- [ ] Called by `um_state` control loop to execute each of the 14 instructions (excluding halt)
- [ ] One extern function for each instruction
- [ ] Parameters passed in are dependent on action of instruction (vary between the array of registers for access to sequence of segments etc.)

Functions:

main.c

- [ ] **`int main(int argc, char *argv)`**
  - [ ] The program driver reads command line arguments and passes an opened `FILE` pointer to `um_run`.

um_state

- [ ] **`extern void um_run(FILE *input_file)`**
  - [ ] Calls `parse_file` from `prepare` to build the 0-segment
  - [ ] Initializes the program counter and registers to 0
  - [ ] Initializes the Hanson sequence for the other segments to an empty state
  - [ ] Initializes the Hanson sequence for recycled indices to an empty state
  - [ ] Calls `execute_instructions` to begin execution
  - [ ] Call `clean_up` after instruction finishes to recycle all memory
- [ ]
```
void execute_instructions(uint32_t *program_counter,
                          uint32_t *prog_seg,
                          Seq_T     other_segs,
                          Seq_T     available_indices)
```
  - [ ] Calls `unwrap_instruction` to un-bitpack the instruction in `prog_seg[*program_counter]`
  - [ ] Uses a switch on the instruction ID to call the appropriate function from `instructions`
  - [ ] Increments `program_counter`
  - [ ] Loops back to read the next instruction
  - [ ] If the halt instruction is read, stop the loop and exit the function.
- [ ]
```
uint32_t *seg_source(uint32_t *prog_seg,
                     Seq_T     other_segs,
                     uint32_t  seg_num,
                     uint32_t  seg_index)
```
  - [ ] `seg_load` and `seg_store` both require pointers to locations in segments, so this helper function takes references to all segments, and the desired segment number and index within that segment and returns a pointer to the desired location in the desired segment.
- [ ]
```
void clean_up(uint32_t **prog_seg,
              Seq_T     *other_segs,
              Seq_T     *available_indices)
```
  - [ ] Recycles all memory associated with the state of the machine

prepare

- [ ] **`extern uint32 *parse_file(FILE *input_file)`**
  - [ ] Takes in input file pointer and populates a c-array (representing segment zero) with instruction code words

instructions

- [ ] **`extern void c_move(uint32_t *source, uint32_t *dest, uint32_t *check)`**
  - [ ] Checks if check is not zero, and moves source into destination if true
- [ ] **`extern void seg_load(uint32_t *source, uint32_t *dest)`**
  - [ ] Moves instruction in segment seg_num at index seg_index into the destination register
- [ ] **`extern void seg_store(uint32_t *seg_num, uint32_t *seg_index, uint32_t *reg_src)`**
  - [ ] Moves instruction in the source register (reg_src) and moves it into segment seg_num at index seg_index
- [ ] **`extern void add(uint32_t *reg_one, uint32_t *reg_two, uint32_t *dest)`**
  - [ ] Adds the values in the two registers and pushes it into the desired destination register
  - [ ] Keeps added value in bounds by modding by 2^{32}
- [ ] **`extern void mult(uint32_t *reg_one, uint32_t *reg_two, uint32_t *dest)`**
  - [ ] Multiplies the values in the two registers and pushes it into the desired destination register
  - [ ] Keeps multiplied value in bounds by modding by 2^{32}
- [ ] **`extern void div(uint32_t *reg_one, uint32_t *reg_two, uint32_t *dest)`**
  - [ ] Divides the values in the two registers and pushes it into the desired destination register
- [ ] **`extern void nand(uint32_t *reg_one, uint32_t *reg_two, uint32_t *dest)`**
  - [ ] Bitwise ORs the values in the two registers and pushes the not'ed value into the desired destination register
- [ ] **`extern void map(Seq_T    other_segs, Seq_T available_indices, uint32_t num_words)`**
  - [ ] Maps a segment by checking if there are any recycled segments first in available indices
  - [ ] Creates or recycles a new segment and instantiates the number of words to parameter num_words (equal to $r[C])

- [ ] **`extern void unmap(Seq_T other_segs, Seq_T available_indices, uint32_t seg_num)`**
  - [ ] Segment in other segs (identified with $r[C]) is recycled
  - [ ] Its segment may be reused and is kept track by pushing to available_indices
- [ ] **`extern void out(uint32_t reg_val)`**
  - [ ] Prints to stdout the value passed in (which will be the value at $r[C] of instruction)
  - [ ] Only vals allowed are from 0 to 255 inclusive
- [ ] **`extern void in(uint32_t *reg_C)`**
  - [ ] Takes the value passed in from I/O and pushes to register (which will be $r[C] where C is from 32-bit instruction)
- [ ] **`extern void load_p(uint32_t **prog_seg_p, Seq_T oth_segs, uint32_t *reg_B, uint32_t *reg_C, uint32_t *p_counter)`**
  - [ ] reg_B is used to indicate which segment to duplicate ($m[$r[B]]) and replace segment zero with case where reg_B is 0, doesn't duplicate
  - [ ] Duplicates by deleting old segment zero and mallocs new c-array with same size as $m[$r[B]]
  - [ ] Either case, always adjusts the program counter to $m[0][$r[C]] by altering p_counter
- [ ] **`extern void load_v(uint32_t value, uint32_t *dest_reg)`**
  - [ ] `value` is stored in `dest_reg`…nothing too complicated here

Testing:

Register Testing:

- [ ] `void direct_reg_test(uint32_t *reg)`
  - [ ] Directly accessing register w/ pointer to specific register
  - [ ] We would print the value in register, then change it, then print again

- [ ] `void indirect_reg_test(uint32_t *registers, uint32_t reg_num)`
  - [ ] In-directly accessing register w/ pointer to array of registers and indexes with reg_num
  - [ ] We would print the value in register, then change it, then print again

Segment Testing:

- [ ] `void direct_zseg_test(uint32_t *seg)`
  - [ ] Directly accessing zero segment w/ pointer to zero segment.
- [ ] `void many_segs_test(Seq_T oth_segs)`
  - [ ] Fills the oth_segs sequence with a thousand segments populated with an instruction holding its number and we'll iterate through the entire sequence in order and output to ensure that all segments are properly added in
- [ ] `void recycling_segs_test(Seq_T oth_segs, Seq_T recycled_segs)`
  - [ ] Tests recycled segments by unmapping sections then trying to map and checking if that specific segment has now been overwritten (since we want to make sure that it continue to use places in the seq which have been unmapped)
  - [ ] Stress Test: This repeatedly maps and maps a million times to ensure that the sequence is properly recycling segments and will not overload after repeated mapping and unmapping
- [ ] `void print_zseg_test(uint32_t *seg)`
  - [ ] Goes through segment and prints opcode for each instruction
  - [ ] Use this to print zero segment and compare where umdump to see if we properly parsed file (instructions in seg zero before execution should be the same as the umdump)
- [ ] `void print_seg_test(Seq_T oth_segs, uint32_t num_seg)`
  - [ ] Load instructions into a segment and print them to ensure they are translated properly
  - [ ] Use this to print other segments and make sure there are the proper number of instructions per segment

Integration Testing:

- [ ] `void regs_to_segs(Seq_T oth_segs, uint32_t *regs)`
  - [ ] Load information into other segments from register and check if the destination holds the proper value
  - [ ] Make sure to access segment memory using `seg_source` function
- [ ] `void segs_to_regs(Seq_T oth_segs, uint32_t *regs)`
  - [ ] Load information into register from other segments and check if the destination holds the proper value
  - [ ] Make sure to access segment memory using `seg_source` function
- [ ] `void regs_to_prog(uint32_t *prog_seg, uint32_t *regs)`
  - [ ] Load information into zero segment from register and check if the destination holds the proper value
  - [ ] Access segment using `prog_seg`
- [ ] `void prog_to_regs(uint32_t *prog_seg, uint32_t *regs)`
  - [ ] Load information into register from zero segment and check if the destination holds the proper value
  - [ ] Access segment using `prog_seg`
- [ ] `void segs_to_prog(Seq_T oth_segs, uint32_t *regs)`
  - [ ] Load information into zero segment from other segments and check if the destination holds the proper value
  - [ ] Make sure to access other segment memory using `seg_source` function
- [ ] `void prog_to_segs(Seq_T oth_segs, uint32_t *regs)`
  - [ ] Load information into zero segment from other segments and check if the destination holds the proper value
  - [ ] Make sure to access other segment memory using `seg_source` function

Test Framework:

- [ ] We have developed a python-based unit testing framework for um instructions.
- [ ] The test framework expands on the capabilities of the um-lab test framework by automating the running of our um implementation and diffing the output against the expected output automatically.
- [ ] It is also possible to automatically generate test output using this by running test input through the reference implementation.

NOTES:

Cases allowed to fail (Do whatever when fails):

- 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.
- A segmented load or segmented store refers to an unmapped segment.
- A segmented load or segmented store refers to a location outside the bounds of a mapped segment
- An instruction unmaps either $m[0] or a segment that is not mapped.
- An instruction divides by zero.
- An instruction loads a program from a segment that is not mapped.
- An instruction outputs a value larger than 255.

Not allowed to fail:

- Resource exhaustion (heap memory allocation fails)
    - Result: Halt and throw checked runtime error
- Contract Violations (extra/invalid command line arguments)
    - Result: Stderr message & EXIT_FAILURE

Edgecases to think about:

- Test for invalid input um file
    - Check argc/argv and if file can be accessed properly
- Invalid commands
    - Opcode is not one of the 14 implemented instructions
- Halt testing
    - Make sure that when halted memory and other machine state processes are reset/deleted
    - Test by valgrinding to check if heap memory is properly freed following the desired end of the program
- Test for each command (i.e. math operations, output etc)
    - Efficiently done by testing disassembling of code word (print out opcode, portions A, B, and C of instruction)
    - Check that sections of the machine were properly altered following the execution of instruction (program counter shifted, registers/segments updated etc)
- Test input properly (beyond printable characters 33-126)
    - Test using /dev/urandom
- Resource exhaustion test
    - Run memory allocation that would result in error