HW7 DESIGN - UNIVERSAL MACHINE EMULATOR
Ballard Blair (bblair01) and Siddarth Kapoor (skapoo01)
Comp40-Spring 2016

Interfaces

UM - module to start the UM and load in the initial .UM program into segmented memory
This module will contain the main() function for our universal machine. The general outline of our main will be very simple:

```
Main (argc, argv*[])
{
        read input;
        load instruction (UM_mem memory, file* fp);
        close input;
        execute (UM_mem memory);
        free_memory(UM_mem);
        return;
}
```

UM_Mem - Module to handle segmented memory of universal machine.

Data Types:
We'll be using a struct called UM_Mem to hold a Sequence_T of pointers to segments, each represented by a UArray that holds 32 bit elements. The struct also contains a Stack_T mem_tracker to tell which segments are unmapped.

```
        struct UM_Mem {
                Sequence_T memory; (A sequence of pointers to UArray_T segments)
                Stack_T mem_tracker; (A stack of integer seg_id's)
        };

        UArray_T segment; (An array of 32-bit words/uint32_t)
```

A sequence is incapable of holding null values, and so we chose to make it a sequence to pointers to UArray_T's so that the underlying UArray could be null without the sequence having to contain a null value. Our choice of sequence as our main data structure for representing the UM's memory is based on the fact that indices of elements in the sequence are fixed, which is necessary for the maintenance of unique segment_id's for our segments, as segments are identified by their index in the sequence.

Our UM_Mem will exist on the stack of the UM's main, and will be passed to our controller via a pointer.

Invariants:

- We will reference our segments in the sequence by their index, this should remain constant throughout the program i.e. there's a 1-to-1 mapping between segment_id's and sequence indices.
- When a segment is cleared, only the pointer the underlying UArray is set to NULL so that the indices of the segments do not change and the segment can be reused.

Functions:

extern UM_Mem new_memory()

Returns a new UM_Mem with an empty memory sequence capable of holding UArray segments, and an empty mem_tracker stack

extern void free_memory(UM_Mem memory)

frees all associated memory in the UM_mem

extern int map_seg(UM_Mem memory, int num_words)

creates a segment in UM_mem capable of holding num_words, 32-bit words, returns an integer seg_id for the new segment. If the mem_tracker stack is empty, add new segment to sequence, else pop unmapped segment and return mapped segment index

extern void unmap_seg(UM_Mem memory, int seg_id)

pushes segment id onto mem_tracker and frees segment memory

extern void mem_address(UM_Mem memory, int seg_id, int offset)*

returns address of a particular offset in a particular segment in memory. The user may access addresses and offsets which are invalid, and results of this are unpredictable.

extern void load_instruction(UM_Mem memory, file fp)*

loads 32-bit word into segment 0

extern int segment_length(int seg_id)

returns the length of the segment associated with seg_id

Functions Private to implementation:

static void clean_segment (int seg_id)

frees associated memory in seg_id

extern void print_mem_map(UM_Mem memory)

prints out the sequence memory, and corresponding segments, and a summary of total heap allocated memory currently in use

UM_Controller - Module to run the execution of the UM and instructions, uses the following loop structure:

```
program_counter = mem_adress(0,0);
while (program_counter < segment_length(0))
    fetch instruction
    decode instruction - interpret opcode and register usage
    Exit_code = execute instruction
    if (exit_code == 1) {
        call a cre
        break;
    }
```

Data Structures:

We'll be using an array to store the 8 registers. The registers are used by the the controller's instructions, and thus need only exists on the stack of the execute function where the main execution cycle is situated. There is also a uint32_t pointer called program_counter which points to the instruction to be executed in the current cycle. Finally, there is an instruction struct which is used by the UM_Interpreter module to parse instructions stored in the segment while hiding the format of said instructions by putting them in a separate module.This struct is also declared on the stack of the execute function and passed to the interpreter module via a pointer, as it needs to exists only as long as execution is ongoing.

```
uint32_t registers[8];
uint32_t *program_counter;
struct instruction {
    int opcode;
    int register_a;
    int register_b;
    int register_c;
    int value;
};
```

Functions:

Void execute (UM_mem memory)

Runs the execution of the UM and instruction utilizing the loop described above

uint32_t get_instruction(UM_mem memory, uint32_t pc)

updates pc and returns the next 32 bit word instruction from segment 0

int handle_instruction (UM_mem memory, instruction op, uint32_t registers, uint32_t *pc)*

function determines which instruction to execute based off of the opcode

void conditional_move(uint32_t registers, int reg_a, int reg_b)*

checks if register C is 0, if not then the contents of reg_b are moved to reg_a

void segmented_load (uint32_t registers, int reg_a, int reg_b, int reg_c)*

the contents in segment[reg_b][reg_c] are moved to reg_a

void segmented_store (uint32_t registers, int reg_a, int reg_b, int reg_c)*

memory segment [reg_a][reg_b] gets the contents of reg_a

void add (uint32_t registers, int reg_a, int reg_b, int reg_c)*

reg_a gets the sum of the contents of reg_b and reg_c

void multiply (uint32_t registers, int reg_a, int reg_b, int reg_c)*

reg_a gets the product of the contents of reg_b and reg_c

void divide (uint32_t registers, int reg_a, int reg_b, int reg_c)*

reg_a gets the quotient of reg_b and reg_c

void bit_nand (uint32_t registers, int reg_a, int reg_b, int reg_c)*

reg_a gets ~(reg_b & reg_c)

void halt ()

computation stops and the UM is shut down

void map_segment (uint32_t registers, int reg_a, int reg_b, int reg_c)*

a new segment is created with a number of words equal to the value of reg_c, each words in the new segment is initialized to 0, bit pattern of not all zeroes and does not identify and mapped segment is put in reg_b, new segment is mapped to segment[reg_b]

void unmap_segment (uint32_t registers, int reg_a, int reg_b, int reg_c)*

segment[reg_c] is unmapped

void output (uint32_t registers, int reg_c)*

value of reg_c is displayed on I/O device, only values 0 to 255 are allowed

void input (uint32_t registers, int reg_c)*

UM waits for input on I/O device, reg_c is loaded with input which must be a value from 0 to 255, if the end of input is signaled, reg_c is loaded with a 32-bit word in which every bit is 1

void load_program (uint32_t registers, int reg_b, int reg_c, uint32_t *pc)*

segment [reg_b] is duplicated and duplicate replaces segment[0], program counter is set to point to segment[0][reg_c]

void load_value (uint32_t registers, int reg_a, int value)*

Value is loaded into reg_a

UM_Interpreter - module to parse 32 bit instruction words, and store parsed values into an Instruction struct. This module uses the bitpack interface for this purpose. We have chosen to implement it in isolation from other modules to isolate the secret of the on-disk format of the instructions. Should this format change, the only thing that will be changed in our implementation is this module.

Data Structures:

```
struct instruction {
        int opcode;
        int register_a;
        int register_b;
        int register_c;
        int value;
};
```

Functions:

void parse_instruction(uint32_t encoded, Instruction decoded)

This function accepts an encoded 32 bit word and parses it using the bitpack module. The parsed details are deposited into the Instruction struct which then contains the instruction in decoded form. The struct is passed by reference to avoid having to heap-allocate it, and then worry about freeing it at a later point.

Architecture

Our UM module accepts a program name from the command-line, and stores it into segment_0 of our UM_Mem. It then passes this non-empty UM_Mem to out UM_Controller, which is

responsible for handling the execution of the program in segment_0. This is also the module where the data structures for our program counter, registers, and instruction interpretation are declared. This module has a simple loop structure that follows a fetch-decode-execute pattern, which manages the control flow according to the program in segment_0. The loop ends when a HALT instruction is encountered, or there are no more instructions to be read.

Test Plan

Testing Module Interpreter

Testing this module is straightforward. We'll need a test main that has an Instruction struct declared inside it. We can accept a filename of a file that contains 32 bit instructions from the command line, and run a loop that calls the parse_instruction() function and then prints the data members of the instructions struct. Testing with small input files, such as files containing one to ten 32 bit words will allow us to inspect if the interpreter functions correctly.

Testing Module UM_Mem

See provided test_um_mem.c file

Testing Module UM_Controller

We will create unit tests for every function in the UM_controller, in the same fashion we have tested each function for UM_Mem.