Comp40 Homework 6: UM Design Document

Names: Isabelle Lai (ilai01) and Andrea Foo (afoo01)

Table of Contents:

ARCHITECTURE

Modules

1. Main UM module
    a. Setup 8 32-bit registers
        i. Hanson UArray with 8 elements of size 4 bytes
        ii. Set registers to 0
    b. Setup main memory
        i. Main memory represented by a Hanson Sequence
        ii. Each element is a pointer to a UArray containing the words of the segment
        iii. Each segment's id will be its index in the Hanson Sequence
        iv. Create a Stack that will hold the segment indices of all the unmapped segments
            1. When a new segment is mapped, the stack will be first checked for any unmapped segments. The new segment will then be mapped at the segment id of an unmapped segment and the id will be removed from the stack. If not, map will just map to the end of the sequence
    c. Read in program and store in segment 0
        i. Use stat to determine length of file, so we know how much memory to map for segment 0
        ii. Use Memory Management Instruction module to map segment 0
        iii. Store program counter (32 bits) (uint32_t current instruction)
            1. Should be at index 0, which is word 0 in 0 segment
    d. Implement Function to read opcodes and perform instructions by calling the instruction functions from other modules
    e. Implementation for halt instruction
        i. Free memory associated with registers (UArray) and main memory (Sequence)
        ii. Exit
2. Arithmetic and data instructions Implementation
    a. Add Instruction
    b. Multiply Instruction
    c. Divide Instruction
    d. Load Value Instruction (***different format)
        i. Store value specified in the least significant 25 bits of the instruction word in the register specified by the next 3 bits of the instruction word.
    e. Conditional Move Instruction
3. Memory management instructions Implementation
    a. Map Segment Instruction
        i. Create UArray based on number of words to map
        ii. Check Hanson Stack for available segment id
            1. If Stack is empty, create new segment id

        2. If Stack is not empty, pop segment id off the stack
    iii. Store pointer returned to UArray in Hanson Sequence at segment id found previously
b. Unmap segment Instruction
    i. Free memory segment -> have the pointer at memory segment point to null
    ii. Add segment id to Hanson stack holding indices unmapped segments
c. Segmented load Instruction
    i. Get value at specified indices of sequence to get value in segment (using Sequence and UArray functions)
    ii. Store in specified register (which is an element of a UArray)
d. Segmented store Instruction
    i. Get word at specified register (Specific element in UArray) and store it in a specified segment (Access main memory location using Sequence and UArray functions)
e. Load program Instruction
    i. Duplicate specified segment
    ii. Free memory of segment 0
    iii. Replace pointer to segment 0 with pointer to duplicated segment
    iv. Update program counter to be set to index of specified word
f. Bitwise NAND

4. I/O instructions (1 byte instructions) Implementation
    a. Input
        i. Read from stdin one byte (unsigned 8 bit characters)
    b. Output
        i. Print to stdout one byte (unsigned 8 bit characters)
5. Bitpack
    a. Use to parse instruction words

Module Interaction (.h files)

Our main module will call the functions from the other modules in order to perform the correct instructions (arithmetic, memory management, etc.). During setup, the main module will call map segment to map segment 0. The main module will pass pointers to the data structures representing the registers and main memory into each function as needed.

```
/*
* um.h
* Andrea Foo (afoo01) and Isabelle Lai (ilai01)
* Comp40, HW7 um
* Due: April 1, 2020
* Purpose: Interface file for the main UM implementation
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <bitpack.h>
#include <assert.h>
#include "mem_instructions.h"
#include "arith_instructions.h"
#include "io_instructions.h"

#ifndef um_INCLUDED
#define um_INCLUDED

/*
* set_up_machine
* Input: filename containing instruction words
* Returns: none
* Purpose: initializes data structures to store main memory, registers, segment ids
*          then reads in instructions from file and stores in segment 0
*/
void set_up_machine(char *filename);

/*
* run_program
* Input: none
* Returns: none
* Purpose: performs each instructions stored in segment 0
*/
void run_program();


#endif
```

```c
/*
* mem_instructions.h
* Andrea Foo (afoo01) and Isabelle Lai (ilai01)
* Comp40, HW7 um
* Due: April 1, 2020
* Purpose: Interface file for the memory instructions implementation
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <assert.h>
#include <bitpack.h>
#include <seq.h>
#include <uarray.h>
#ifndef MEM_INSTRUCTIONS_INCLUDED
#define MEM_INSTRUCTIONS_INCLUDED

/*
* map_segment
* Input: pointer to main memory Sequence, pointer to registers UArray,
*        index of registers B and C, pointer to unmapped ids Stack
* Returns: none
* Purpose: creates a new segment in main memory Sequence and with num words
*          equal to value in regC. all words are initialized to 0, and new
*          segment id is placed in regB
*/
void map_segment(Seq_T main_memory, UArray_T registers, int regB, int regC, Stack_T
unmapped_ids);

/*
* unmap_segment
* Input: pointer to main memory Sequence, pointer to registers UArray,
*        index of register C, pointer to unmapped ids Stack
* Returns: none
* Purpose: unmaps segment with segment id held in regC and adds segment id to unmapped ids
Stack
*/
void unmap_segment(Seq_T main_memory, UArray_T registers, int regC, Stack_T unmapped_ids);

/*
* segmented_load
* Input: pointer to main memory Sequence, pointer to registers UArray,
*        index of registers A, B, and C
* Returns: none
* Purpose: loads value of word at $m[$r[B]][$r[C]] in regA
*/
void segmented_load(Seq_T main_memory, UArray_T registers, int regA, int regB, int regC);

/*
* segmented_store
* Input: pointer to main memory Sequence, pointer to registers UArray,
*        index of registers A, B, and C
* Returns: none
* Purpose: stores value regA in word at $m[$r[B]][$r[C]] in regA
*/
void segmented_store(Seq_T main_memory, UArray_T registers, int regA, int regB, int regC);

/*
* load_program
* Input: pointer to main memory Sequence, pointer to registers UArray,index of registersB & C
*        * Returns: none
* Purpose: duplicates segment $m[$r[B]] and replaces segment 0, then resets program
*          counter to point to $m[0][$r[C]]
*/
void load_program(Seq_T main_memory, UArray_T registers, int regB, int regC);
#endif
```

```c
/*
* arith_instructions.h
* Andrea Foo (afoo01) and Isabelle Lai (ilai01)
* Comp40, HW7 um
* Due: April 1, 2020
* Purpose: Interface file for the arithmetic instructions implementation
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <bitpack.h>
#include <assert.h>
#include <uarray.h>

#ifndef ARITH_INSTRUCTIONS_INCLUDED
#define ARITH_INSTRUCTIONS_INCLUDED


/*
* add
* Input: a pointer to the uarray registers and 3 integers
*        that specify the indices of the registers
* Returns: none
* Purpose: add contents of register B and register C and
*          store in register A
*/
void add(UArray_T registers, int regA, int regB, int regC);


/*
* multiply
* Input: a pointer to the uarray registers and 3 integers
*        that specify the indices of the registers
* Returns: none
* Purpose: multiply contents of register B and register C
*          and store in register A
*/
void multiply(UArray_T registers, int regA, int regB, int regC);


/*
* divide
* Input: a pointer to the uarray registers and 3 integers that
*        specify the indices of the registers
* Returns: none
* Purpose: divide contents of register B and register C and
*         store in register A
*/
void divide(UArray_T registers, int regA, int regB, int regC);


/*
* load_value
* Input: a pointer to the uarray registers and 3 integers
*        that specify the indices of the registers
* Returns: none
* Purpose: Stores given value at the index of the
*          register given
*/
void load_value(UArray_T registers, int regA, int value);
```

```
/*
* load_value
* Input: a pointer to the uarray registers and 3 integers
*        that specify the indices of the registers
* Returns: none
* Purpose: If value in register C is not 0, then store contents of
*          register B in register A
*/
void conditional_move(UArray_T registers, int regA, int regB, int regC);


/*
* bitwise_NAND
* Input: a pointer to the uarray registers and 3 integers
*        that specify the indices of the registers
* Returns: none
* Purpose: Bitwise NAND register B and C and store result in register A
*/
void bitwise_NAND(UArray_T registers, int regA, int regB, int regC);

#endif
```

```c
/*
* io_instructions.h
* Andrea Foo (afoo01) and Isabelle Lai (ilai01)
* Comp40, HW7 um
* Due: April 1, 2020
* Purpose: Interface file for the input output instructions implementation
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <bitpack.h>
#include <assert.h>
#include <uarray.h>

#ifndef IO_INSTRUCTIONS_INCLUDED
#define IO_INSTRUCTIONS_INCLUDED


/*
* read_input
* Input: UArray_T registers, int regC
* Output: no output
* Purpose: Read one byte from std in and store as unsigned int
*          and store it in register C
*/
void read_input(UArray_T registers, int regC);

/*
* read_input
* Input: UArray_T registers, int regC
* Output: value in Register C (0 to 255)
* Purpose: Get value in register C and output to stdout
*/
void output(UArray_T registers, int regC);


#endif
```

IMPLEMENTATION PLAN

1. Implement bitpack.c using solutions from Arith
2. Implement main UM module
    a. Create um.c
    b. Create UArray for registers
    c. Create Sequence for main memory
    d. Write function to read in instruction words and store in segment 0
        i. Create program counter and set to 0 (uint_32t that stores index of current instruction)
        ii. Create UArray to store words of segment 0
    e. Create Stack to store unmap segment ids
    f. Implement Function to read opcodes and parse instructions by calling the instruction functions from other modules
        i. Use bitpack to get opcode from word
        ii. If opcode is for load value instruction:
            1. Use bitpack to get register A and value
        iii. Else:
            1. Use bitpack to get registers A, B, C from word
        iv. Call corresponding function for instruction
    g. Implement for halt instruction
        i. Free all memory associated with UM
        ii. Exit
    h. Account for contract violations (EXIT_FAILURE)
3. Implement memory management module
    a. Create file mem_management.c
    b. Implement Map segment instruction function
        i. Create UArray at sequence index to store words based on size passed in
        ii. Get unmapped segment id from stack (unless stack is empty)
        iii. Store UArray in Sequence at segment id index
    c. Implement Unmap segment instruction function
        i. Free memory associated with the pointer at that memory segment id and set it to null
        ii. Append the index of the unmapped segment to the stack that keeps track of all the unmapped segment ids
    d. Implement Segmented load Instruction function
        i. Get value at specified segment using Sequence function to get correct segment and UArray function to get correct word in segment
        ii. Store value in specified register, by storing in corresponding index of UArray
    e. Implement Segmented store Instruction function
        i. Access corresponding index of UArray to get value in specified register
        ii. Get pointer at specified segment using Sequence function to get correct segment and UArray function to get correct word in segment

        iii. Store value from register in segment
    f. Implement Load program Instruction function
        i. Duplicate specified segment
        ii. Free memory of segment 0
        iii. Replace pointer to segment 0 with pointer to duplicated segment
        iv. Update program counter to be set to specified value
4. Implement Arithmetic and data instructions Module
    a. Add Instruction
        i. Function will take in four parameters: a pointer to the uarray registers and 3 integers that specify the indices of the registers
        ii. $r[A] := ($r[B] + [$r[C]) mod 2^32 (add contents of r[B] and r[C] and store in r[A])
    b. Multiply Instruction
        i. Function will take in four parameters: a pointer to the uarray registers and 3 integers that specify the indices of the registers
        ii. $r[A] := ($r[B] * [$r[C]) mod 2^32 (multiply contents of r[B] and r[C] and store in r[A])
    c. Divide Instruction
        i. Function will take in four parameters: a pointer to the uarray registers and 3 integers that specify the indices of the registers
        ii. $r[A] := ($r[B] / [$r[C]) mod 2^32 (divide contents of r[B] by r[C] and store in r[A])
    d. Load Value Instruction (***different format)
        i. Function will take in three parameters: a pointer to the uarray registers, an integer specifying the index of a register, and an integer value
        ii. Store value in given register
    e. Conditional Move Instruction
        i. Function will take in four parameters: a pointer to the uarray registers and 3 integers that specify the indices of the registers
        ii. If $r[C] not equal to 0: $r[A] := $r[B] (store contents of r[B] in r[A])
    f. Bitwise NAND
        i. Function will take in four parameters: a pointer to the uarray registers and 3 integers that specify the indices of the registers
        ii. $r[A] := ~($r[B] & $r[C])
5. Implement I/O instructions (1 byte instructions) Module
    a. Input: Read one byte from stdin and store as unsigned integer
        i. Store value in register C
    b. Output: Get the value in register C (check that value is within range 0 to 255)
        i. Print to stdout one byte as unsigned 32 bit integer

TESTING PLAN

Note: UM Failure is an unchecked runtime error

Architecture Testing (C functions)

1. Test that instruction words are read and stored in segment 0 properly

```
/*
* test_segment_zero
* Input: pointer to main memory sequence, number of words in instruction file
* Output: prints whether segment zero is correctly holding the instruction
*         words from the instruction file
* Purpose: test that segment zero correctly stored instruction words
*          It will exit with EXIT_FAILURE and have a error message printed
*          to stderr if it has a contract violation. Else, it will print
*          a success message.
*/
void test_segment_zero(Seq_T main_memory, int num_words)
```

2. Test that program counter points to word 0 in segment 0
    a. Edge Case:
        i. Case: Program counter points outside the bounds of $m[0]
        Result: UM Fails
        ii. Case: Program counter points to word that does not code for a valid instruction
        Result: UM Fails

```
/*
* test_program_counter
* Input: the uint32_t program counter
* Output: prints whether the program counter is
*         correctly set to 0 and pointing to a valid word
* Purpose: test that the program counter is correctly set at
*          the beginning of a machine cycle
*          It will exit with EXIT_FAILURE and have a error message printed
*          to stderr if it has a contract violation. Else, it will print
*          a success message.
*/
void test_program_counter(uint32_t program_counter)
```

3. Test for memory leaks after UM halts: Valgrind
4. Test for contract violations when um is called from command line

```
/*
* test_command_line_input
* Input: int argc
* Output: prints whether the command line has a contract violation
* Purpose: test that the command line has the right number of arguments.
*          It will exit with EXIT_FAILURE and have a error message printed
*          to stderr if it has a contract violation. Else, it will print
*          a success message.
*/
void test_command_line_input(int argc)
```

5. Test that instructions are correctly disassembled and parsed for opcodes and register values using bitpack

```
/*
* test_instruction_parsing
* Input: opcode and registers A, B, C of an instruction word
* Output: prints whether opcode and registers are valid or not
* Purpose: test that an instruction word has been correctly analyzed
*          It will exit with EXIT_FAILURE and have a error message printed
*          to stderr if it has a contract violation. Else, it will print
*          a success message.
*/
void test_instruction_parsing(int opcode, int rA, int rB, int rC)
```

<u>Instruction Set Testing (.um files)</u>

1. Arithmetic and data instructions Testing plan
    a. Unit test for add instruction
        i. Create binary instruction file containing different add instruction words
        ii. Predict result of instruction words
        iii. Run UM with file and print out contents of each register after each instruction to ensure that output matches predicted output
        iv. Edge cases:
            1. Case: Sum is a value that is too large to be represented in 32-bits
               Result:: Add function should mod the sum before storing in register
            2. Case: Register specified is out of bounds of the UArray
               Result: Halt with a checked runtime error
    b. Unit test for Multiply Instruction
        i. Create binary instruction file containing different multiply instruction words
        ii. Predict result of instruction words

iii. Run UM with file and print out contents of each register after each instruction to ensure that output matches predicted output
iv. Edge cases:
1. Case: Multiplied result is a value that is too large to be represented in 32-bits
Result:: Multiply function should mod the product before storing in register
2. Case: A number is multiplied by 0
Result: Result in specified register should be 0
3. Case: Register specified is out of bounds of the UArray
Result: Halt with a checked runtime error

c. Unit test for Divide Instruction
i. Create binary instruction file containing different divide instruction words
ii. Predict result of instruction words
iii. Run UM with file and print out contents of each register after each instruction to ensure that output matches expected output
iv. Edge Cases:
1. Case: Divided result is a value that is too large to be represented in 32-bits
Result:: Divide function should mod the result before storing in register
2. Case: A value is being divided by 0
Result: UM fails
3. Case: 0 is being divided by some value
Result: Result in specified register should be 0
4. Case: Register specified is out of bounds of the UArray
Result: Halt with a checked runtime error
d. Unit Test for Load Value Instruction (***different format)
i. Create binary instruction file containing different load value instruction words
ii. Predict result of instruction words
iii. Run UM with file and print out contents of each register after each instruction to ensure that output matches predicted output
iv. Edge Cases:
1. Case: Register specified is out of bounds of the UArray
Result: Halt with a checked runtime error
e. Unit test for Conditional Move Instruction
i. Create binary instruction file containing different conditional move instruction words
1. Test conditional moves with both true and false values in register C
ii. Predict result of instruction words

iii. Run UM with file and print out contents of each register after each instruction to ensure that output result matches predicted result
iv. Edge cases:
1. Case: Register specified is out of bounds of the UArray
Result: Halt with a checked runtime error

f. Unit Test for Bitwise NAND
i. Create binary instruction file containing different bitwise NAND instruction words
ii. Predict result of instruction words
iii. Run UM with file and print out contents of each register after each instruction to ensure that output matches predicted output
iv. Edge Cases:
v. Case: Register specified is out of bounds of the UArray
Result: Halt with a checked runtime error
vi. Case: Value to nand with is 0
Result: Resulting value in specified register should be all 1s
vii. Case: Value to nand with is all 1s
Result: Resulting value in specified register should be all 0s

2. Memory management instructions Testing Plan
a. Unit Test for Map Segment Instruction
i. Create binary instruction file containing different map segment instruction words
ii. Predict result of instruction words
iii. Run UM with file and print out contents of each segment after each instruction to ensure that output matches predicted output
iv. Valgrind to ensure no memory was lost
v. Edge Cases:
1. Case: Register specified is out of bounds of the UArray
Result: Halt with a checked runtime error

b. Unmap segment Instruction
i. Create binary instruction file containing different unmap segment instruction words
ii. Predict result of instruction words
iii. Run UM with file and print out contents of each segment after each instruction to ensure that output matches predicted output
iv. Valgrind to ensure no memory leaks
v. Edge Cases:
1. Case: Register specified is out of bounds of the UArray
Result: Halt with a checked runtime error
2. Case: Segment has not been mapped/does not exist
Result: UM fails

3. Case: Instruction unmaps segment 0
   Result: UM fails

c. Segmented load Instruction
   i. Create binary instruction file containing different segmented load instruction words
   ii. Predict result of instruction words
   iii. Run UM with file and print out contents of each register and segments after each instruction to ensure that output matches predicted output
   iv. Valgrind to ensure no memory leaks
   v. Edge Cases:
      1. Case: Register specified is out of bounds of the UArray
         Result: Halt with a checked runtime error
      2. Case: Segment to load from has not been mapped/does not exist
         Result: UM fails
      3. Case: Offset in segment to load from is out of bounds of a mapped segment
         Result: UM fails

d. Segmented store Instruction
   i. Create binary instruction file containing different segmented store instruction words
   ii. Predict result of instruction words
   iii. Run UM with file and print out contents of each register and segments after each instruction to ensure that output matches predicted output
   iv. Valgrind to ensure no memory leaks
   v. Edge Cases:
      1. Case: Register specified is out of bounds of the UArray
         Result: Halt with a checked runtime error
      2. Case: Segment to store into has not been mapped/does not exist
         Result: UM fails
      3. Case: Offset in segment to store into is out of bounds of a mapped segment
         Result: UM fails

e. Load program Instruction
   i. Create binary instruction file containing different load program instruction words
   ii. Predict result of instruction words
   iii. Run UM with file and print out contents of loaded segment and duplicated segment after each instruction to ensure that output matches predicted output
   iv. Edge Cases:
      1. Case: Register specified is out of bounds of the UArray
         Result: Halt with a checked runtime error

2. Case: Instruction loads a program from a segment that has not been mapped
   Result: UM fails

3. I/O instructions (read/write 1 byte) Testing Plan
   a. Unit Test for Input
      i. Create binary instruction file containing different input instruction words
      ii. Predict result of instruction words
      iii. Run UM with file and print out contents of register after each instruction to ensure that output matches predicted output
      iv. Edge Cases:
         1. Case: Register specified is out of bounds of the UArray
            Result: Halt with a checked runtime error
         2. Case: Input is outside the range of 0 to 255
            Result: Halt with checked runtime error
   b. Unit Test for Output
      i. Create binary instruction file containing different output instruction words
      ii. Predict result of instruction words
      iii. Run UM with file and ensure that output of each output instruction matches predicted output
      iv. Edge Cases:
         1. Case: Register specified is out of bounds of the UArray
            Result: Halt with a checked runtime error
         2. Case: If instruction outputs value outside range of 0 to 255
            Result: UM Fails
4. Halt instruction Testing Plan
   a. Create binary instruction file containing halt
   b. Ensure that program halts with no memory leaks using valgrind