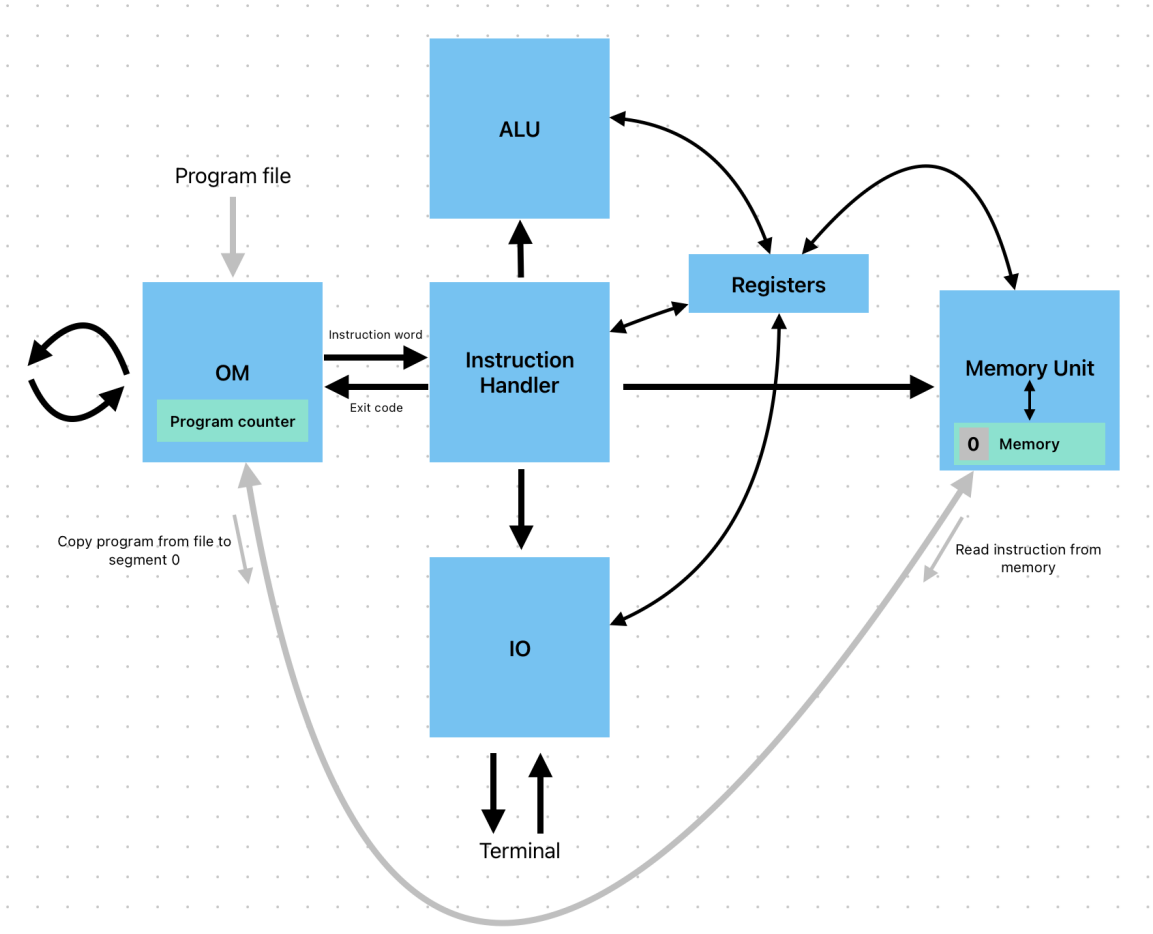
Universal Machine Design Doc

Liam Drew (wdrew02)
Peter Wolfe (pwolfe02)



UM Architecture

Operation module

Input: The initial UM program file

Output: Exit code; Crash if undefined behavior

Interactions: Cycle -> Instruction Handler; Load/halt -> Memory & Registers

- Operations
  - Load Initial state
    - Initialize program counter as 0
    - Accept argument for program file
    - **Load program from file into 0 segment**
    - Start processor cycle
  - Processor cycle
    - Fetch Instruction

- **Call Instruction Handler**
- Update program counter
- Repeat

Instruction Handler

Input: The 32-bit word to decode

Output: Return an exit code to the OM

Interactions: Calls instructions from the ALU, IO, Memory with arguments A, B, C.

- Operations
  - Decode Instruction
    - Will rely on Bitpack interface
  - **Execute Instruction by calling ALU, IO, or Memory**
  - **Return exit code to OM. -1 if normal; -2 if halt; >=0 if changing program counter**

Registers

Input: The register index to get or modify; the 32-bit word to change it to

Output: A register's value

- **Accessible from every module**
- Eight registers, each a 32-bit word
- 13: Load value

Arithmetic Logic Unit

Input: A: the output register, B: the first register, C: the second register

Interactions: Gets the values from registers B and C. Modifies register A.

- Operations
  - 3: Addition
  - 4: Multiplication
  - 5: Division
  - 6: Bitwise NAND
  - 0: Conditional Move

I/O Unit

Input: Reads from stdin

Output: Writes to stdout

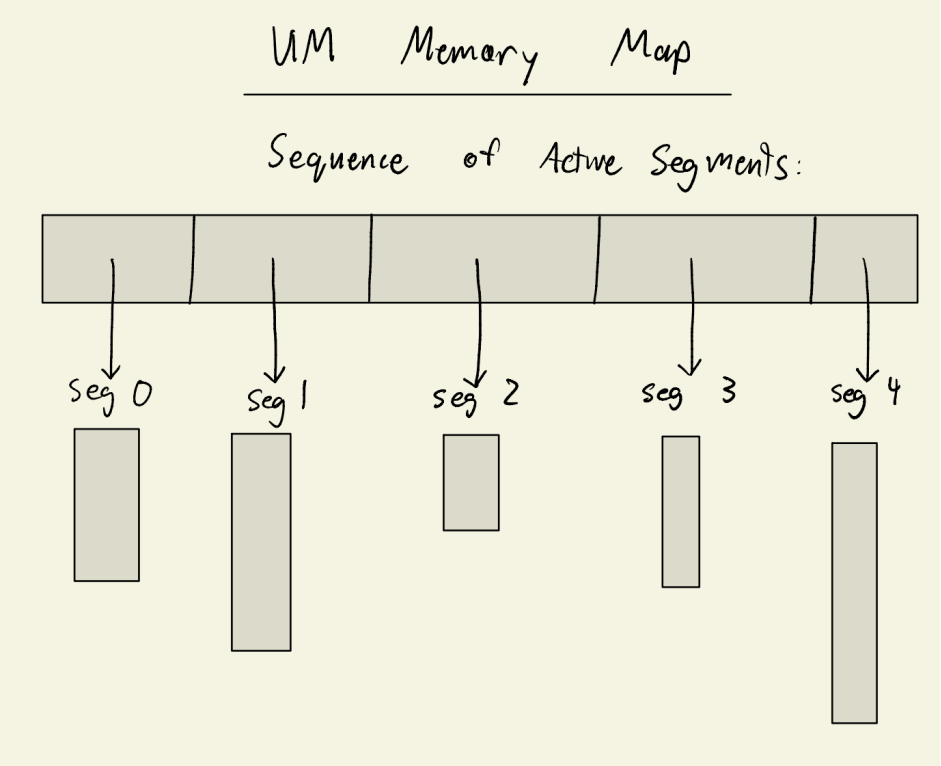
Interactions: Stores in register A when using instruction 11

- Operations
  - 11: Read input (from stdin) and **store in register**
  - 12: Print register to output (stdout)

Memory Unit

Interactions: Accepts instruction from IH. Reads and writes registers.

- Relies on a Hanson sequence of pointers to arrays as a data structure (see our data structures discussion below)
- **No other module has access to the memory segments (with the exception of OM to segment 0)**
- Operations
  - 1: Segmented Load
  - 2: Segmented Store
  - 8: Map Segment (like calloc)
  - 9: Unmap Segment (like free)
  - 12: Load program
    - The Instruction Handler passes the program counter index to the OM
  - 7: Halt
    - Unmap every segment
    - The Instruction Handler passes a Halt exit code to the OM
- Memory Implementation



Data Structures

- Memory: A Hanson sequence of structs: Each struct holds a pointer to a heap-allocated array of uint32_t words and the array's size
    - Map Segment creates a new array and adds its pointer and size to the end of the sequence
    - Unmap Segment frees the array and sets the pointer to NULL and size to -1
- Enumerations for opcodes (from the lab)
- Registers are an array of eight uint32_t's

Implementation Plan

- When building out UM, we will start by implementing the two modules that are absolutely necessary to run the UM: the Operations Module and the Memory Interface.
    - Once these modules have been created, we will test them with our memory unit tests (see testing plan below)
- Next, we will create the register and instruction handler modules
    - The registers are simple as they are merely setters and getters
    - The instruction handler will require extensive testing to ensure it decodes bits correctly and handles corrupt instructions.
    - After these modules are implemented, we will be able to test our memory module more thoroughly. The memory module of the UM presents the biggest challenge with the most potential for errors (especially when switching from an active program to a different program when the load program instruction is encountered). Having these modules implemented will help us test this fundamental piece of UM infrastructure before we implement any other functionality
- After our program is capable of handling instructions and accessing registers, we will implement the IO module and the ALU.
    - At this point, all functionality of the UM should be implemented.
- Writing unit tests. Will do this using the um provided in lab. These unit tests will test every single one of the 14 operations that the um is capable of executing. Doing this early has multiple benefits: first, we will get very familiar with the functionality of the um, and that understanding will help us as we try to build our own. Additionally, these unit tests will be useful for testing our um after we build it.
- Performance testing & benchmarking

Testing Plan

Write UM unit tests and test them on the reference UM we saw in lab 10.

After we confirm that our um tests work as expected on the reference UM, we will run the unit tests on the UM we build, and verify that our UM behaves the way we expect.

We will implement the following functions as debugging tools:

- Print registers
- Print memory
- Breakpoints (halt machine if program counter reaches certain #)

These Unit Tests are:

- Arithmetic Correctness confirmation:
    - Addition
    - Multiplication
    - Division
    - Bitwise NAND
- Halt test:
    - Make sure program correctly halts with no memory leaks (Valgrind)
- Load Segment:
    - Make sure program can load a new memory segment and start executing it without errors
- Register test:
    - Flood the registers with random values
    - Put a value in a register out of bounds
- Instruction decoder tests:
    - While programming: Print values to ensure bitpack interface is decoding the correct bits
    - Make samples for all fourteen instructions. Ensure they are decoded correctly.
    - Input corrupted instructions where the opcode > 14 or the middle bits are filled.
- I/O Tests
    - Make sure the UM can write and read from disk as expected. Take a string of chars from input and print the same chars back.
- Memory Tests
    - We plan to test our memory module outside of the UM before testing it in the UM

- First, we plan on making sure that we can allocate and access memory in the way we expect.
    - void allocateAndAccess(int *array, int count);
        - Map a memory segment with the given array of ints. Access it at the given segment index. Assert both are the same.
- Next, we will make sure that we can unmap (deallocate) memory without any memory leaks.
    - void firstMiddleLast();
        - Map nine memory segments of random sizes. Unmap #1,2,3,5,6,7. Halt the program. Assert that #0,4, and 8 were unallocated with no null pointer errors.
- We want to make sure our program is correctly copied into memory. This uses the OM's program loader and the Memory Unit
    - void loadProgramInitial(char *filename);
        - Load a program from file into memory segment 0. Print it out. Diff test to see if they are the same.
- We want to make sure that the Load Program instruction works. We can also use this function to test that loading from 0 to 0 is quick.
    - void loadProgramTest(char *filename, int seg);
        - Load a program from file into segment m[seg]. Run Load Program instruction to copy into segment 0.

Interfaces and Function Contracts (for interconnected modules)

operation_manager.h

```c
/**
* @file operation_manager.h
* @authors Peter Wolfe and Liam Drew
* @brief Interface of the operation manager
*/

/**
* @name load_initial_state
* @brief Load the initial state of the UM. This involves:
*        Initializing the program counter to 0.
*        Accepting an argument for the program file.
*        Load program from file into 0 segment of memory.
*        Start processor cycle.
*
*
* @note interactions: Interacts with the load_program() function in the memory
*       interface, since this interaction is necessary to intialize the UM.
* @returns None
*/
extern void load_initial_state();


/**
* @name start_processor_cycle
* @brief Starts the processor cycle of the UM. This invovles
*        Fetching & Handling the program's next instruction.
*        Updates program counter
*        Repeats (continues processor cycle)
*
*
* @note interactions: Interacts with the handle_next_instruction() method of
*       the Instruction Handler module. The handle_next_instruction() method
*       returns different exit codes to its caller (this function) depending
*       on the state of the program. See the handle_next_instruction()
*       function contract in instruction_handler.h for more information.
* @returns None
*/
extern void start_processor_cycle();
```

instruction_handler.h

```
/**
* @file instruction_handler.h
* @authors Peter Wolfe and Liam Drew
* @brief Interface of the operation manager
*/

/**
* @name start_processor_cycle
* @brief Handles the next instruction in the running program's memory segment.
*        Fetches and Decodes the program's next instruction.
*        Executes the program's next instruction by calling the relevant
*        operation module (ALU, IO, or Memory Module)
*
*
* @note This is the most interconnected method in the entire UM. This method
*       returns an exit code to its caller (the Operation Manager module) to
*       inform it of the state of the program after the instruction executes.
*       In order to actually execute this instruction, this function calls
*       the relevant function in one of the other 3 modules in the UM (the
*       ALU, IO module, or Memory Module) to ensure that the instruction gets
*       executed. Please see our design doc for full functionality of each
*       module.
* @returns int Exit Code:
*                    -1: Normal
*                    -2: Halt
*                   >=0: Changing program counter
*/
extern int handle_next_instruction();
```

memory_interface.h

```
/**
* @file memory_interface.h
* @authors Peter Wolfe and Liam Drew
* @brief The interface of the memory module in the UM. This module supports
*        access to and modification of the segmented memory of the UM.
*/

/**
* @name segmented_load
* @brief Put the contents of an address in memory in a register
*
*
* @note interactions: Interacts with the register module to access memory data
*       and store the accessed data.
* @returns None
*/
void extern segmented_load(unsigned a, unsigned b, unsigned c);

/**
* @name segmented_store
* @brief Put the contents of a register into memory
*
*
* @note interactions: Interacts with the register module to access register
*       data and store the data in memory.
* @returns None
*/
void extern segmented_store(unsigned a, unsigned b, unsigned c);

/**
* @name map_segment
* @brief Initialize a new memory segment and store the address of the segment
*        in a specified register.
*
* @note interactions: Interacts with the register module to access register
*       data and store the data in memory.
* @returns None
*/
void extern map_segment(unsigned b, unsigned c);

/**
```

```
* @name unmap_segment
* @brief Frees the memory associated with the address stored in a specified
*        register.
*
* @note interactions: Interacts with the register module to access register
*       data and store the data in memory.
* @returns None
*/
void extern unmap_segment(unsigned c);

/**
* @name load_program
* @brief Copies the memory segment pointed to by a specified register into
*        segment 0 (the active program segment)
*
* @note interactions: Interacts with the register module to access register
*       data and store the data in memory. Additionally, this function is
*       called by the operation manager module when the UM starts up.
* @returns None
*/
void extern load_program(unsigned b);

/**
* @name load_value
* @brief Sets a specified register to store instruction contents.
*
* @note interactions: Interacts with the register module to access register
*       data and store the data in memory.
* @returns None
*/
void extern load_value(unsigned a);

/**
* @name handle_halt
* @brief Handles the halt instruction and frees all program memory
*
* @returns None
*/
void extern handle_halt();
```