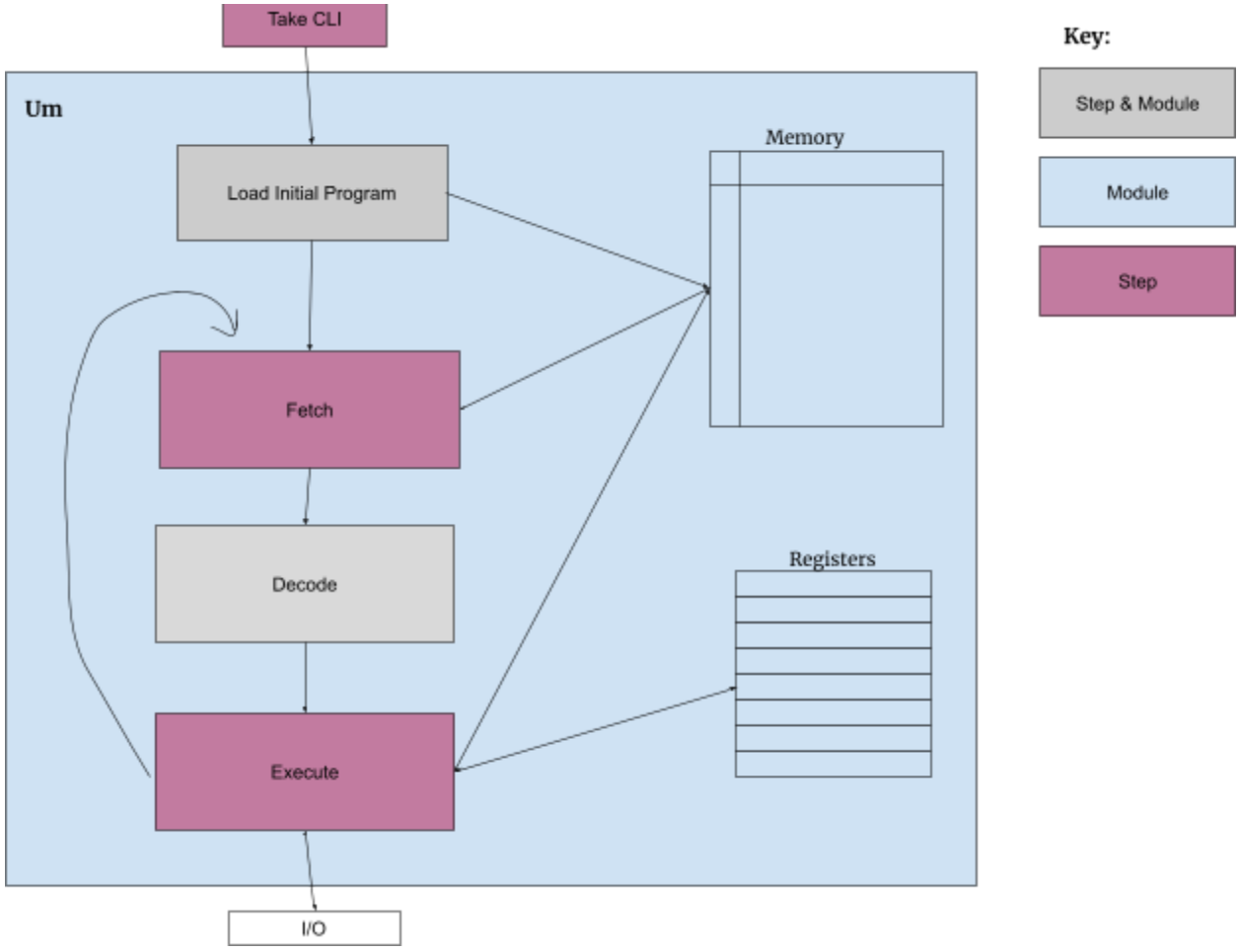
# UM Design Doc

*Tom Lyons (tlyons01) & Noah Stiegler (nstieg01)*

## Architecture



## Modules:

### UM:

Our UM module is the overarching module that handles the fetch, decode, and execute operations of the universal machine. It is a client of the registers and memory modules, as well as the decode module.

Performs the specified operation based on the opcode from the inputted instruction to affect the registers and memory until the program ends or an unspecified operation occurs.

```
/* Um_run
 * Purpose:
 *      Run the UM emulator on the code stored in the provided filename
 * Arguments:
 *      (char *) filename - The name of the file to read the program from
 * Notes:
 *      - CRE for filename to be NULL
 *      - CRE if filename can't be opened
 */
void Um_run(char *filename);
```

### Decode:

Exports a method for decoding the 32 bit word to get the opcode, and, regardless of the opcode, also gets the value of register A, register B, and register C for logical instructions and the register and immediate values for the load-val instruction.

```
/* decode_word
 * Purpose:
 *      To decode a 32-bit um instruction into its component op code,
 *      register(s) and possible value. It is up to the client of this
function
 *      to determine which of these values to use and which to ignore based
 *      on the op-code of the word
 * Arguments:
 *      (uint32_t) word - The instruction to decode
 *      (Um_register ) *rA - If a regular instruction, which register is A
 *      (Um_register) *rB - If a regular instruction, which register is B
 *      (Um_register) *rC - If a regular instruction, which register is C
 *      (Um_register) *load_rA - If a load value instruction, which is rA
 *      (uint32_t) *load_val - If a load value instruction, the value to be
 *                             loaded into rA
 * Returns:
 *      (Um_opcode) that specifies the operation to be performed in the UM
 */
Um_opcode decode_word(uint32_t word, Um_register *rA, Um_register *rB,
                      Um_register *rC, Um_register *load_rA,
                      uint32_t *load_val);
```

**Registers:**
Our registers module is in charge of representing the data stored in the 8 general purpose registers of our universal machine.

We plan on making use of a C array to represent the eight registers as an array of uint32_t's.

```
/* Registers_init
 * Purpose:
 *      Initialize a new register file where each register contains 0
 * Arguments:
 *      (unsigned) num_registers - The number of registers in the file
 * Returns:
 *      A pointer to an array of registers
 * Notes:
 *      - CRE for num_registers to be 0
 *      - CRE if memory cannot be allocated for registers
 *      - All registers start initialized to 0
 *      - Makes registers which hold 32-bit values
 */
uint32_t *Registers_init(unsigned num_registers);


/* register_free
 * Purpose:
 *      To free the registers instance
 * Argument:
 *      (uint32_t **) register_p - Pointer to the address of the
                                   registers to free
 * Notes:
 *      - CRE for register_p or *register_p to be NULL
 *      - Sets *register_p to be null
 */
void register_free(uint32_t **register_p);
```

**Segmented Memory:**
Implements segmented memory. It keeps track of the instruction pointer (program counter). Stores the currently executing code as well as data in memory. It fetches the next instruction, fetches words from memory, stores words in memory, maps new segments and unmaps existing segments.

We will implement memory as a Hanson Table storing Hanson sequences of 32-bit words whose keys are 32-bit segment identifiers. The table will only store mapped memory, segment 0 will be handled separately. We will assign identifiers based on a 32 bit counter (which maps directly to a word), which will use a one indexed counter that will rollover to 1 once it gets to the max 32 bit unsigned value. We will test if an identifier has already been used by seeing if that key is in the Hanson table. If it has, we increment again until we get a value that isn't in use. We will keep a separate counter to make sure that no more than 2^32  segments are mapped at a given time (including segment 0).

```c
/* SegMem_new
 * Purpose:
 *      Creates a new segmented memory instance with the contents of the file
 *      passed in as the contents of Segment 0. Sets the instruction pointer
 *      to be word 0.
 * Arguments:
 *      (FILE *) input_file - An opened filestream to a file containing a .um
 *                            program
 * Returns:
 *      (SegMem_T) that contains the entirity of the new memory instance for
 *                 the universal machine
 * Notes:
 *      - CRE for input_file to be NULL
 */
SegMem_T SegMem_new(FILE *input_file);

/* SegMem_fetch_next_i
 * Purpose:
 *      Fetches the next um instruction from the program loaded in segment 0
 * Arguments:
 *      (SegMem_T) mem - The memory to get the instruction from
 * Returns:
 *      (uint32_t) that contains the next instruction for the um to execute
 * Notes:
 *      - CRE for mem to be NULL
 *      - URE for the next instruction to be asked for when there are no more
 *        instructions
 */
uint32_t SegMem_fetch_next_i(SegMem_T mem);

/* SegMem_map
 * Purpose:
 *      Maps a new segment in the memory of a given size and gives back its id
 * Arguments:
 *      (SegMem_T) mem - The memory to map a new segment in
 *      (uint32_t) size - the number of words the segment can store
 * Returns:
 *      (uint32_t) that contains the segment identifier for the newly mapped
 *                 segment
 * Notes:
 *      - CRE for mem to be NULL
 *      - CRE for a segment to be mapped if 2^32 segments are already mapped
 *      - CRE if there isn't enough memory to map a segment of the given size
 */
uint32_t SegMem_map(SegMem_T mem, uint32_t size);

/* SegMem_unmap
 * Purpose:
 *      Unmap a segment of memory allowing for its segment ID to be reused
 * Arguments:
 *      (SegMem_T) mem - The memory to unmap the segment in
 * Notes:
 *      - CRE for mem to be NULL
 *      - URE if unmapping a segment that is not mapped.
 */
void SegMem_unmap(SegMem_T mem);

/* SegMem_get_word
 * Purpose:
 *      Retrieve a word stored in memory at the given segment and index
 * Arguments:
 *      (SegMem_T) mem - The memory to get the word from
```

```
 *      (uint32_t) seg_id - The segment to get the word from
 *      (uint32_t) word_idx - Which word in the segment to get
 * Notes:
 *      - CRE for mem to be NULL
 *      - URE for seg_id to refer to a segment which doesn't exist
 *      - URE for word_idx to refer to a word outside the bounds of the segment
 */
uint32_t SegMem_get_word(SegMem_T mem, uint32_t seg_id, uint32_t word_idx);


/* SegMem_put_word
 * Purpose:
 *      Puts a word into memory at the given segment and index
 * Arguments:
 *      (SegMem_T) mem - The memory to put the word into
 *      (uint32_t) seg_id - The segment to put the word into
 *      (uint32_t) word_idx - Which word in the segment to set
 *      (uint32_t) word - the word to be put into memory
 * Notes:
 *      - CRE for mem to be NULL
 *      - URE for seg_id
 */
uint32_t SegMem_put_word(SegMem_T mem, uint32_t seg_id, uint32_t word_idx,
                         uint32_t word);

/* SegMem_load_program
* Purpose:
*      Loads a new program stored in the given segment to run by copying the
*      contents of that segment into segment 0 and setting the program counter
*      to the provided value
* Arguments:
*      (SegMem_T) mem - The memory to load the program from/into
*      (uint32_t) seg_id - The segment in mem to load the program from
*      (uint32_t) new_program_counter - The word index in the new program to
*                                       start reading instructions from
* Notes:
*      - CRE for mem to be NULL
*      - URE for seg_id to not refer to a mapped segment
*      - URE for new_program_counter to refer to an instruction out of bounds
*        of the new program
*      - Incredibly quick to load segment 0
* */
void SegMem_load_program(SegMem_T mem, uint32_t seg_id,
                         uint32_t new_program_counter);


/* SegMem_free
 * Purpose:
 *      To free the memory used to store SegMem information
 * Argument:
 *      (SegMem_T) mem - The memory to free
 * Notes:
 *      - CRE if mem is null
 *      - CRE if mem->seg0 or mem->data_segments is NULL
 */
```

```
void SegMem_free(SegMem_T &mem);
```

# Implementation/Testing

Testing has 2 parts: .um files which unit test our implementation and c functions which test our segment/memory mapping

1) Implement opening a file in main
    a) Ensure the file is opened by printing it
2) Start writing Um_run by creating a SegMem, passing it an opened file
3) Implement SegMem, creating unit tests as we go
    a) Start by implementing the constructor by making sure it can read and print the file opened by main. Implement reading big-endian. Check by reading in an example .um program and comparing to umdump
    b) Implement a destructor, check with valgrind for no errors
        i) Void check_constructor_destructor()
            (1) Creates and then destroys a SegMem_T, checking with valgrind. Uses SegMem_new and SegMem_free
    c) Change reading a file so it loads contents into segment 0. Implement a program counter and fetch_next_i. Use it to print out a file loaded into segment 0. Print out program counter as we go to make sure it works
        i) Void check_fetch_next_i_basic
            (1) Reads a simple example file with the constructor. Compare instructions fetched with those in the hardcoded file. Uses SegMem_new() and SegMem_free().
    d) Implement get_word and put_word. Check on segment 0
        i) Void check_get_word_seg_0
            (1) Reads a simple example file with the constructor. Compares words read at various indices with those in the file. Uses SegMem_new(), _get_word(), and _free()
        ii) Void check_put_word_seg_0
            (1) Read a simple example file with the constructor. Replace every other word with various values. Compare values read out by fetch_next_i
        iii) Void check_put_get_word_seg_0
            (1) Reads a simple example file with the constructor. _put_word()s words in every other value, then checks that those values are what they should be with _get_word()
        iv) Check that getting or putting a word out of bounds segfaults. Not one of our usual unit tests (will comment out) because it's not an exception we can catch
    e) Implement map and unmap functions by using a Hanson table and keeping track of which identifiers we have given out
        i) Test by mapping and unmapping 2^32 + 1 times to make sure we can re-use identifiers
            (1) Void map_unmap_at_limit()

(a) Maps memory many times, unmapping each time, checking the identifiers given back to make sure they're correct and wrap around

f) Update free so it frees all mapped memory
   i) Void check_destructor_mapped()
      (1) Maps memory then frees the SegMem
g) Test map by calling get_word and put_word to fill and then read out memory from a new segment
   i) Void check_new_segment_all_0s
      (1) Map a few new segments and ensure that _get() tells us each segment is filled with 0s
   ii) Void get_put_word_new_segments
      (1) Map several new segments then put and get words from them to ensure they are as they should be
h) Implement load_program, making sure it's quick to load segment 0.
   i) Test on segment 0 by loading a program, fetching its instructions and printing them out, then loading segment 0 at program counter 0, and fetching the same instructions and printing them out to make sure they match.
      (1) Void check_load_seg_0
         (a) Uses _put() and _load_program() and _fetch_next_i() as well as the constructor to make sure that loading segment 0 works to change the program counter and doesn't change data
   ii) Test on a different segment by printing out instructions fetched and program counter, then instructions fetched and program counter after loading a program in a different segment created by mapping and then put_wording values into it
      (1) Void check_load_seg_other
         (a) Uses _put() and _load_program() and _fetch_next_i() as well as the constructor to make sure that loading segment other than the zero segment works to change the program counter and doesn't change data
   iii) Check above with valgrind

4) Implement Registers, unit testing as we go (in a separate unit test file)
   a) Start by allocating the instance onto the heap. Then make destructors.
      i) Void check_constructor_destructor()
         (1) Allocates a register with the constructor, then deallocates with the destructor
   b) Ensure instance can be created and freed without memory leaks in valgrind
   c) Try placing values in and taking values out, printing them to ensure they work and no leaks with valgrind
      i) Void check_register_read_write()
         (1) Makes a register with the constructor, writes some values into it, reads some values out, makes sure they're the same, frees it with the destructor
5) Implement decode
   a) Using bitwise operation from Arith, read in the values for each register and value.

    b) Input our own theoretical instructions to decode
        i) Print out and check that all components are as we expect for all 13 instructions with various different registers
6) Keep implementing Um_run, putting all these modules together
    a) Initialize memory and registers
    b) Build instruction fetch decode execute loop
        i) Incrementally implement how we handle each of the 13 instructions. For each one, create a unit test (.um file, input, and output), and ensure that the behavior matches as we expect for small tests. Add one at a time.
7) Run our unit test function to test the instruction set.
8) Run the midmark.um file to ensure runtime is under 60 seconds
9) Try running other .um files to ensure out program behaves as expected

Unit test the UM instruction set.

- Start by only testing Halt.
- Test might test Output and Halt.
- Your third test might Output, Load Value, and Halt.
- Test add, multiply, halt and so on.