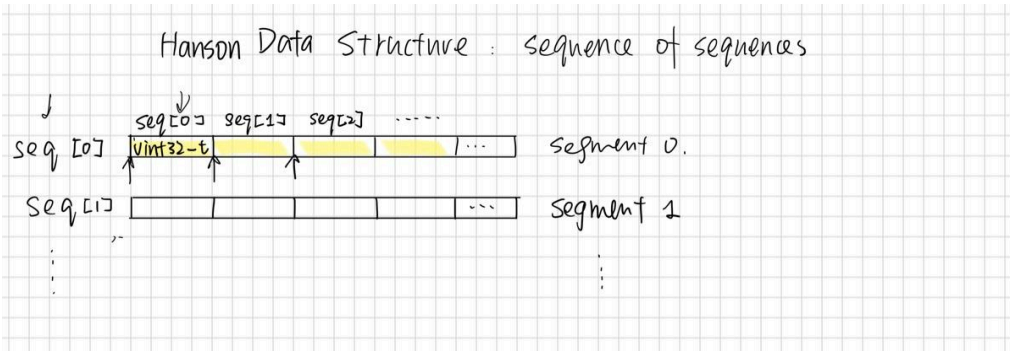
Architecture

Module 1: SegMem.h (Segmented Memory)

Data Structure: a struct that contains:

1. Next segment id: stores the next id to use when mapping a new segment
2. Pointer to the array of empty id: An array of empty segments that are unmapped
3. The representation of segmented memory: The outer sequence represents segments, while the inner sequence contains words representing instructions, where each word is a uint32_t. (The drawing depicts how we represent the segmented memory.)



The data structure is shown below:

```
struct SegMem_T {
        unsigned next_id;
        Uarray_T empty_id;
        Seq_T memory;
};

typedef struct SegMem2_T *SegMem2_T;

/* initialize_seg
*
* Initialize the struct SegMem_T, and initialize the structure of segmented memory which is
* represented by sequence of sequences
*
* Parameters:
*      None
*
* Returns: an initialized struct SegMem_T
* Expects: None
*
```

```
* Notes: Allocates new memory for Seq_T (sequence of sequences); memory will be
deallocated when finishing using the segmented memory by calling the seg_free() or deleting a
segment by calling unmap_seg()
*/
SegMem_T initialize_seg();

/* new_seg
*
* Initialize and create a new segment with the provided number of elements and add the new
* segment to the segmented memory. Return new segid. Updates next_id. This function will be
* used when opcode Map Segment is used, which communicates with um.h
*
* Parameters:
*      SegMem_T seg_mem:      The segmented memory to be updated.
*      unsigned num_words:       number of words in the new segment.
*
* Returns: the id of the new segment created.
* Expects: the passed in SegMem_T is not NULL.
*
* Notes: It's a CRE for seg_mem to be null.
*/
unsigned new_seg(SegMem_T seg_mem, unsigned num_words);

/* unmap_seg
*
* Delete and Deallocate $m[index] and update the segmented memory represented by
* sequence of sequences. This function will be used when the opcode Unmap Segment is used
* which communicates with um.h
*
* Parameters:
*      SegMem_T seg_mem:            The segmented memory to be updated
*      unsigned index:              The index of the segment to be updated
*
* Returns: None
* Expects: the seg_mem cannot be NULL
*
* Notes:
* CRE if the seg_mem is NULL
* this function deallocates the memory of the unmapped segment when the opcode
* Unmap Segment is used
*/
void unmap_seg(SegMem_T seg_mem, unsigned index);

/* seg_get
```

```
*
* Return the value stored in $m[segid][offset], which communicates with um.h when the op code
* Segmented Load is used
*
* Parameters:
*      SegMem_T seg_mem:          The segmented memory to be updated
*      unsigned segid:            The id of the segment to be accessed
*      unsigned offset:           The offset in the segment of the value to be accessed
*
* Returns: The value (uint32_t) stored in the designated segment and offset.
* Expects: The seg_mem cannot be NULL, segid cannot refer to an empty segment or out of
* range, and offset can not be longer than the length of the segment accessed.
*
* Notes: It's a CRE for seg_mem to be null or any of segid or offset to be accessing empty
* segment or out of range.
*/
uint32_t seg_get(SegMem_T seg_mem, unsigned segid, unsigned offset);

/* seg_put
*
* Update a word in the segmented memory, which is an element of the inner sequence. This
* communicates with um.h when the op code Segmented Store is used
*
* Parameters:
*      SegMem_T seg_mem:          The segmented memory to be updated
*      unsigned segid:            The id of the segment to be updated
*      unsigned offset:           The offset of the word to be updated to locate the word
*      uint32_t value:            The new value to replace the original word
*
* Returns: the original value before the update
* Expects: The seg_mem cannot be NULL, segid cannot refer to an empty segment or out of
* range, and offset can not be longer than the length of the segment accessed.
*
* Notes:
* CRE if seg_mem is NULL or any of segid or offset to be accessing empty
* segment or out of range.
*/
uint32_t seg_put(SegMem_T seg_mem, unsigned segid, unsigned offset, uint32_t value);

/* seg_free
*
* Deallocate the whole segmented memory when finished using
*
* Parameters:
```

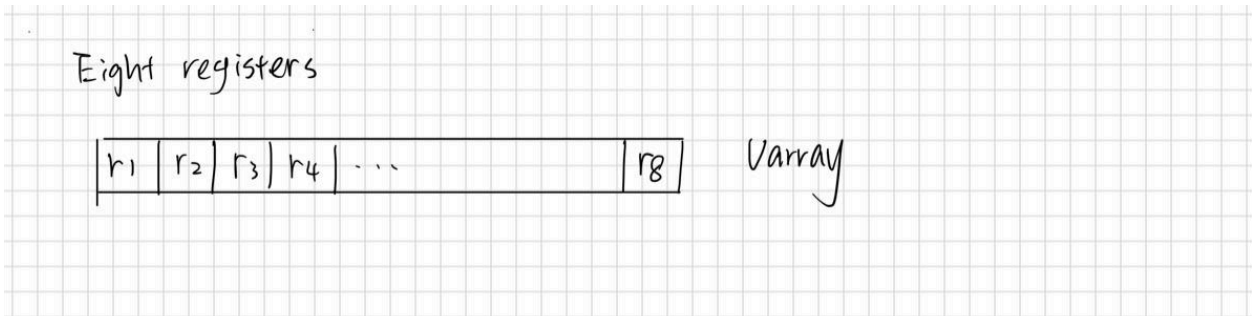
```
*      SegMem_T seg_mem:          The segmented memory to be freed
*
* Returns: None
* Expects: The seg_mem cannot be NULL
*
* Notes:
* CRE if seg_mem is NULL
* this function deallocates the memory of the segmented memory
*/
void seg_free(SegMem_T seg_mem);
```

Module 2: um.h (fetch, decode, execute loop and I/O device)

Take in a file stream with instructions and execute them. This module interacts with the other two modules and prevents main from accessing values in the registers and segmented memory. Also takes in stdin and stdout for I/O devices.

```
struct UM {
        uint32_t* program_counter;
        UArray_T registers;
        SegMem_T seg_mem;
        FILE * input;
        FILE * output;
}
```

Picture demonstrating the data structure of registers as a Hanson UArray.




```
/* new_um
*
* Initialize the UM struct by reading from the file
*
* Parameters:
*      FILE * instructions:       The input file with instructions in it
*      FILE* input:               the input stream used in I/O device
*      FILE* output:              the output stream used in I/O device
*
* Returns: An initialized UM struct
```

```
* Expects: The instructions cannot be NULL
*
* Notes:
* CRE if instructions is NULL
* this function allocates the memory for the segmented memory, and registers, which will be
* deallocated by calling the um_free after finishing executing the program
*/
UM new_um(FILE * instructions, FILE* input, FILE* output);

/* fetch_decode_execute
*
* Executes the program stored in $m[0]. Communicates with the registers and the segmented
* memory using the functions defined in SegMem.h and registers.h.
*
* Parameters:
*      UM um:              The UM to be executed
*
* Returns: None
* Expects: The UM cannot be NULL
*
* Notes:
* CRE if UM is NULL
* this function deallocates the memory of the segmented memory
*/
void fetch_decode_execute(UM um);

/* um_free
*
* Deallocate um when finished executing
*
* Parameters:
*      UM um:              The UM struct to be freed
*
* Returns: None
* Expects: UM to be not NULL.
*
* Notes:
* this function deallocates the memory of the segmented memory(SegMem_T) and registers
* (UArray_T)
*/
void um_free(UM um);
```

Implementation & Testing Plan

1. Implement unit test for operations as what we did in umlab
2. Implement segmented memory
    a. Initialize a sequence of sequences, where each element is a uint32_T
    b. Initialize the struct:

```
struct SegMem_T {
        unsigned next_id;
        Uarray_T empty_id;
        Seq_T memory;
};
```

    Testing: test if the elements in the struct is initialized correctly by printing the elements out in a testing main. Specifically, check for: 1) next_id should be 1; 2) empty_id initialized to 5 elements, all elements are 0; 3) Seq_T initially has length of 1 ($m[0]).
3. Implement um
    a. Implement initialization function:
        i. Initialize SegMem_T by calling the seg_new function
        Testing: check that SegMem is not null.
        ii. Reading from program file:
            - Extract 4 bytes (32 bits) each time and store each word into segment 0 of module 2 (segmented memory), until end of file.
            Testing: Print the uint32_t fetched as bits/hex and check if the elements are stored correctly and in the right format. Specifically, check for the first and last instructions (edge cases) and make sure that all instructions are correctly read in.
            - Initialize a 32-bit program counter as a pointer, which points to the start of segment 0 initially. Advances to the next word of segment 0 as the program executes.
            Testing: Print the element that the counter is pointing to and check if it equals to the element in $m[0][0]
        iii. Implement 8 registers
            - Initialize an UArray of 8 registers
            Testing: Fully test the registers using a testing main, which calls the initializing functions in registers.h. Check if all elements are initialized to 0.
    b. Implement fetch_decode_execute()
        i. We plan to use enum to organize the operation code:

```
typedef enum Um_opcode {
    CMOV = 0, SLOAD, SSTORE, ADD, MUL, DIV,
    NAND, HALT, ACTIVATE, INACTIVATE, OUT, IN, LOADP, LV
} Um_opcode;
```

        ii. We plan to use switch statements to decide which function to call.

- First, we interpret the first four bits of the instruction word (opcode). If the opcode is between 0-12, then extract according to the three register instruction format to execute the instruction; if the opcode is 13, then use the load-value instruction format
    a. Implement every operation in the corresponding case: if the operation code is 0, then it executes the case 0, where has the implementation of the operation 0: Conditional Move, etc.

  Testing: testing each operation through unit-test (the one implemented in the lab), give the function a word, and see if the operation works as intended (whether the correct operation is carried out. Check for

- In terms of IN and OUT: we plan to use an I/O device (stdin and stdout) to display ASCII characters and perform input and output of unsigned 8-bit characters.
  Testing: testing input operation, either by run the operation input and see if the program loads value to $r[C] and if the program waits for an input or run it by using main and see if $r[C] is loaded with the correct value; testing the output by checking if it prints the value in $r[C] correctly

Testing: test the um as a whole:

- Add print statement to see what's in the registers, what's in the segmented memory (what's in the SegMem Struct) in every iteration, to see if it outputs as intended
- Print out what the counter is pointing to for each iteration, to see if the program counter is proceeded as intended

4. Implement main:
    a. Write the program execution check and contracts (exit with EXIT_FAILURE when not used properly.
       Testing: Make sure the contract check works by inputting invalid and valid usage.
    b. Pass in stdin and stdout into UM as the I/O device, and the instructions file stream.
    c. Execute UM by calling fetch_decode_execute().
    d. Free UM memory.
       Testing: Run the provided instructions file.

Loop Invariant

*A loop invariant is a statement that should be **true** when entering a loop and after every iteration of the loop (checked right before the test of the loop).*

The one meaningful and necessary invariant for the fetch_decode_execute loop is the **program counter**. The program will make sure that the counter should not be null at the start of the loop and is updated to the next instruction in $m[0] at the end of each iteration, which is $m[0][current_index + 1]. In the special case of calling a load_value, the program updates the program counter to $m[0][r[C]].