Danny Atik
Mike Hu
HW6 Design Document
Comp40

Overall Design

In this assignment we are building a Universal Virtual Machine that emulates the most basic instructions of a computer. The hardwares we emulate are a CPU with 8 registers, and a memory that is segmented into different pieces. To keep our program modular, each emulated hardware is separated into their own components and the behavior of each components is defined inside the driver called um.c.

Major Components and Architecture

1. **ADTs**
    a. The segments of memory in our UM will be represented by a hanson sequence of Uarray_t's, with each Uarray holding 32 bit word instructions.
    b. The registers of our UM will be represented by an array of size 8, holding uint32_t's, which are single instructions/words.
    c. We will implement the program counter as a struct with two int variables, responsible for keeping track of which instruction is currently being executed.
2. **Architecture**
    a. The registers, memory segments, and I/O device will be separate modules. The modules come together in the driver called um.c.
    b. The different modules ensures that we can test each components without the rest of the program interfering with the module we are testing and they also keep secrets from each other. Some instructions that require access to different modules don't share secrets. They come together inside um.c and performances the desired instruction.
    c. Each module containing individual .c and .h files will interact inside the driver um.c. The driver will call functions from the different modules and compute the instructions by passing them from memory segments to um and to registers or the other way around.
3. **Modules**
    a. registers.c and register.h
        i. `uint32_t registers [8];`
        ii. Functions:
            1. `void conditional_move(int rA, int rB, int rC);`
                a. The function takes in three integers as array indices and sets the information in rA to rB if rC is not 0.

2. `uint32_t get_reg(int rA);`
    a. Gets the value from rA and returns it.
3. `void set_reg(int rA, uint32_t val);`
    a. Sets the segment specified by rA with value.
4. `void addition(int rA, int rB, int rC);`
    a. Set rA equal to the value of rB + rC mod32.
5. `void multiplication(int rA, int rB, int rC);`
    a. Set rA equal to the value of rB x rC mod32.
6. `void division(int rA, int rB, int rC);`
    a. Set rA equal to the value of floor(rB/rC).
7. `void bitwise_nand(int rA, int rB, int rC);`
    a. Set rA equal to the value of (rB & rC).
8. `void halt();`
    a. Stops the computation, pauses the loop.
9. `void output(int rC);`
10. `void input(int rC);`

iii. Invariants:

b. memory.c and memory.h

i.
```
typedef struct Memory {
        seq_t segment_manager;
        seq_t queue;
} Memory*;
```

ii. Functions:

1. `uint32_t load_instruction(uint32_t valrB, uint32_t valrC);`
    a. Goes into the segmented memory to returns the word specified by segment rB and offset rC.s
2. `void store_instruction(uint32_t valrC, uint32_t valrA, uint32_t valrB);`
    a. Goes into the segmented memory at segment valrA offset *valrB* and sets the value to *valrC*.
3. `Uarray_t load_memory(uint32_t valrB);`
    a. Casts valrb to an int and calls seq_get(T seq, int valrB). Cast the void * returned by seq_get to a Uarray_t and creates a new Uarray_t, filling the new Uarray with what was returned by seq_get() and returning the duplicate Uarray.
4. `void store_memory(uint32_t valrB, Uarray_t mem);`

a. Casts valrb to an int, casts mem to a void *, and calls seq_put() with casted valrB and casted mem, and the sequence as arguments.

5. `uint32_t map_memory(uint32_t valrC);`

   a. Checks our *queue* which holds the unmapped segment identifiers that have been previously used and freed. If the queue is empty we add a new Uarray to *sequence_manager*. If the queue is not empty, we pop off the first element from the queue, go to that address within our segment manager and create a new segment of the desired length, setting the values to zero. This function returns the segment identifier of the new place in memory.

6. `void unmap_memory(uint32_t valrC);`

   a. Pushes *valrC* onto the queue and calls Uarray_free on the segment.

iii. ***Invariants:***

-All mapped segment identifiers will be stored in the segment manager and all unmapped segment identifiers will be stored in the queue, the number of mapped segments is the number of elements inside segment_manager minus the number of elements in the queue(number of unmapped memory).
-All mapped memory segments have a specified length.
-All unmapped memory segments are null.
-If a segment is unmapped then the next mapped segment will have that segment id.

c. um.c and um.h

i. An instance of memory.c, an instance of register.c and an instance of the program counter which is a struct of two ints.

ii.
```
typedef struct prog_count {
            int segment;
            int instruction;
} prog_count*;
```

iii. Functions:

1. `void segment_load(int rA, int rB, int rC);`

   a. Segment load operation is achieved by calling get_reg(rB) and get_reg(rC). The value returned by those two functions, *valrB* and *valrC* will be used as arguments in load_memory() from memory.c. The word specified by segment *valrB* and offset *valrC* is returned by load_memory(*valrB, valrC*). The word is then passed into set_reg() with rA as arguments.

2. `void segment_store(int rA, int rB, int rC);`
   a. Segment store operation is achieved by calling get_reg(rC) which returns *valrC*, get_reg(rB) which returns *segrB*, and get_reg(rA) which returns *offsetrA*. valrC is passed into store_memory() with *segrB* and *offsetrA*.
3. `void map_segment(int rC);`
   a. Map segment operation is achieved by calling get_reg(rC) which returns *valrc,* the desired segment length, and calls map_memory(valrC) from memory.c which returns the segment identifier. We call set_reg() with rB and the returned segment identifier.
4. `void ummap_segment(int rC);`
   a. Unmap segment operation is achieved by calling get_reg(rC), which returns *valrC,* and unmap_memory is called from memory.h with *valrC* as argument.
5. `void load_program(int rB, int rC);`
   a. Load_program is achieved by calling get_reg(rB) and then load_segment(valrB). The Uarray_t returned by load_segment() is then stored in segment 0 by calling store_memory() and passing 0 and the Uarray as arguments. We then call get_reg(rC) and set the program counter equal to segment 0, instruction rC. If rB = 0, then the function only needs to update the program counter.
6. `void load_value(int rA, uint32_t value);`
   a. `Calls set_reg(ra, value).`

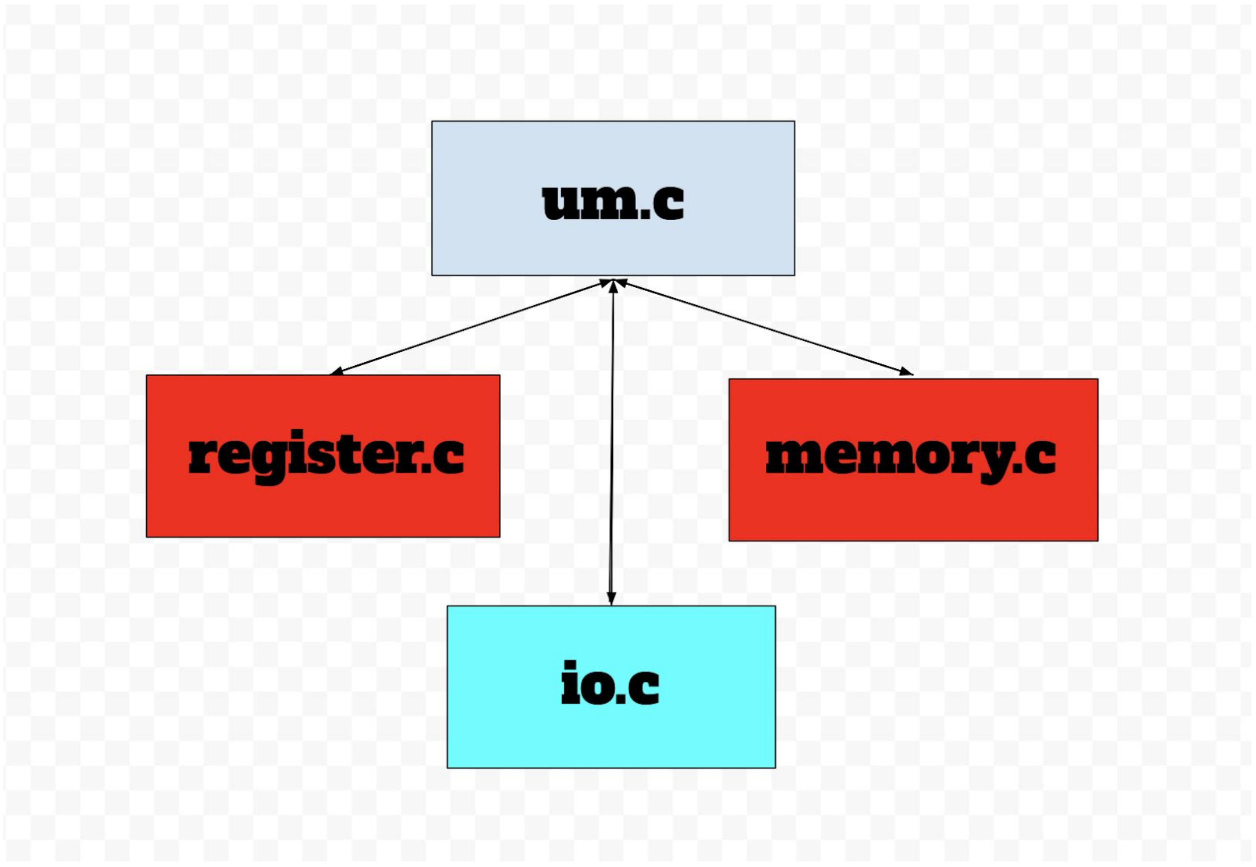
iv. Invariants:
   1. All um functions require secrets from both memory.c and register.c

d. IO.c and IO.h
   i. Functions:
      1. `void input(int rC);`
         a. Uses fgetc() to scan for an input from stdin and store it in rC by calling set_reg(rC);
      2. `void output(int rC);`
         a. Calls get_reg(rC) and passes the returned value to fputc() to output a character into stdout.

Test Cases:

Since each individual module will be unit tested, we will also have a general test case that tests all the modules as a whole. Such will start with the simplest instruction combinations to the more complex combinations that will simulate how the um is actually run. Each stage of this test will give us good feedback on the bugs we will run into.

Diagram:


um.c
register.c
memory.c
io.c