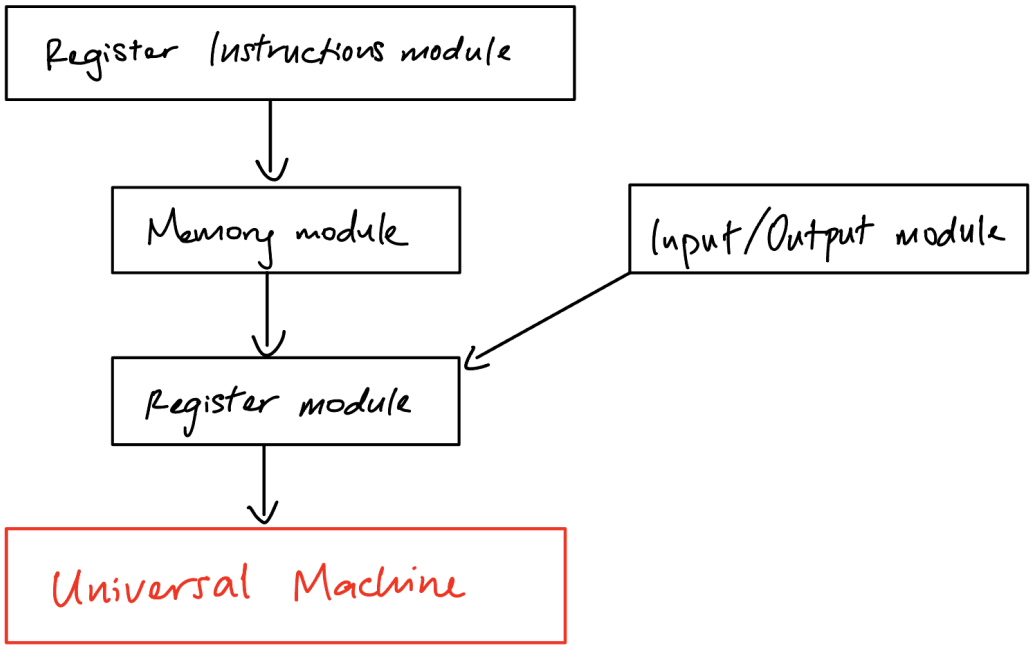
HW 6 Universal Machine: Kerwin Teh (kteh01) & Youssef Ezzo (yezzo01)

Architecture

- Register module
    - This module manages all data stored in registers
    - We will use an array consisting of 8 elements of uint32_t type
    - Each of these 8 elements represents one register, denoted by %r[0] through %r[7]
- Register Instructions module
    - This module handles the implementation of all instructions that do not alter memory (opcode 0-7, 13)
    - For instructions with opcode 0-7, no memory needs to be managed. For these instructions, we simply perform the desired ‘calculations’ using the registers and segment access.
    - Load Value (instruction 13) will insert the value from the instruction into $r[A].
    - For other instructions, additional work will be required as described below
- Memory module (instructions 8-9, 12)
    - A Sequence (Seq_T) will be used to store UArrays of 32-bit words which are called segments. This Sequence will be called “segment_sequence”.
        - Each element in the Sequence points to a UArray, where each UArray stores contiguous 32-bit words
        - Each UArray represents a Segment
    - Seq[X] corresponds to a Segment whose ID is X
    - We will use another Sequence to keep track of unmapped/free segments that can be used/reused by the user.
        - We will store the IDs of unmapped segments
        - Whenever a user wants to map a segment, we will have a record of segments that can be ‘reused’, eliminating the need to create new segments and thereby conserving space

        - We will call this “free_segments”
    - The load program instruction (opcode 12) will first create a new UArray of ‘words’ which is a duplicate of segment at $m[$r[B]] (the Uarray pointed to by the element in the sequence).
    - This new Uarray will then replace the UArray pointed to by seq[0] so that it is now the new segment0.
- Input/Output module (10-11)
    - For the output instruction, we will simply need to output the value in $r[C] to the output stream (stdout)
    - For input instruction, we read in from the input stream (stdin) and store the value in $r[C]
        - When reading in, we will have to set $r[C] to all 1’s once all the input has been read.



The diagram above shows how the modules will interact with one another. The Register module is needed by all other modules. The input/Output module only interacts with the Register module as does the memory module. The Register Instructions module

interacts with both the memory module (for accessing elements) and the register module. All of these modules will then work cohesively in the Universal Machine.

Implementation (testing in red):

1. Read in instructions from .um file and store in segment0
    a. If the .um file is not called correctly in the command line by the user, print "Error: um file not inputted correctly" to stderr and exit the program with EXIT_FAILURE
    b. Declare an array of registers by creating an array of 8 elements of uint32 type
    c. Declare a Sequence of UArrays, where each UArray represents a segment of 32-bit words.
    d. Initialize all registers to be 0.
    e. Set the program counter to point to the first word in segment0
    f. Print out the values in each register and the program counter to ensure all elements have been initialized correctly.
    g. Unit test by passing in a known sequence of words and make sure the instructions match what we expect and that segment0 refers to the right element. This will be done using a similar framework as in the lab.
2. "Unpack" the word at the program counter to get the desired instruction/registers
    a. Use the bitpack module to get the instruction (first 4 bits of the word)
    b. Print instruction out to make sure that the most significant 4 bits of the word correspond to the instruction we expect
3. Implement all 14 UM instructions as specified. Registers A, B, and C refer to specific registers passed in as parameters to these instructions when called. (Test for the following explained after this section)
    a. Conditional Move:
        i. If Register C is not equal to 0, copy value in Register B to Register A
    b. Segmented Load

i. Find value located at $m[$r[B]][$r[C]] and store that value in Register A

c. Segmented Store
   i. Take value in Register C and store it in memory located at $m[$r[A]][$r[B]]

d. Addition
   i. Sum up values in Registers B and C, mod it with 2^{32}, and store the result in Register A

e. Multiplication
   i. Multiply values in Registers B and C, mod it with 2^{32}, and store the result in Register A

f. Division
   i. Divide value in Register B by value in Register C and store the result in Register A

g. Bitwise NAND
   i. Perform a Bitwise AND operation on values in Registers B and C
   ii. Take the negation of that result and store it in Register A

h. Halt
   i. Stop running the program

i. Map Segment
   i. Get the size of segment in Register C. This size corresponds to the number of words that will be in the segment.
   ii. Create a new UArray of size in Register C, and initialize all values to 0.
   iii. If free_segments is not empty (there are empty segments that can be reused), we get the index of that segment from the 'free_segments' sequence and go to that index in segment_sequence and set that pointer to be the new UArray that has just been created
   iv. If there are no empty sequences to be reused, call Seq_addhi to add the new UArray to the end of segment_sequence.

v. By doing this, the segment is mapped as $m[$r[B]]
vi. Wherever the mapped segment has been added, we will store that index in $r[B]

j. Unmap Segment
i. Store the index (segment ID) of the Uarray we are about to unmap into our ‘free_segments’ sequence
ii. Set the pointer at that index in the original sequence to NULL
iii. That index can now be reused the next time we map new data.

k. Output
i. Send the value in Register C to stdout
ii. Only values from 0 to 255 can be outputted

l. Input
i. Read input from stdin and store it in Register C
ii. Once all input has been read, set Register C to a 32-bit value that is all 1s

m. Load Program
i. Check if Register B is 0
- If it is, just continue to the next instruction
- Otherwise, get the index in Register B and go to segment_sequence[index] to get the UArray. Call UArray_copy on that UArray to duplicate it, and replace segment0 with the duplicated UArray.
- Set the program counter to point to $m[0][r[C]]

n. Load Value
i. Unpack the value from the word (least significant 25 bits) and store it in register A.

4. Execute the instruction and update the program counter to point to the next word.
a. Based on the instruction from unpacking, call the appropriate function (one for each instruction) with the appropriate registers
5. Repeat until all words have been read (thus all instructions have been executed)

Testing Plan

Unit Tests:

For each of these tests we will create a new test that appends previously tested instructions to make sure they all work cohesively together.

1. Halt
    a. Make sure the program exits
2. Output
    a. Append output instruction to stream
    b. Print value in that register
    c. Call objdump to make sure that the value in the register is as supposed
3. Load value
    a. Append load val instruction to stream
    b. Load given value into register
    c. Use output to verify the value in the register
4. Cmove
    a. Append cmove instruction to stream
    b. Use output to verify the the value has been moved correctly
5. Segmented Load
    a. Append segmented load to stream
    b. Go to memory in segment_sequence and print the value at $m[$r[B]][$r[C]]
    c. Use output to verify that the value in Register A is the same as the value in memory above
    d. Verify that both values match
6. Segmented Store:
    a. Append segmented store to stream
    b. Use output to print the value in $r[C]
    c. Go to address in segment_sequence and print the value at the specified indices
    d. Verify that both values match

7. Addition
    a. Append add to stream
    b. Output values in registers A, B, C and make sure that the value in $r[A] is the sum of values in $r[B] and $r[C]
8. Multiplication
    a. Append multiplication to stream
    b. Output values in registers A, B, C and make sure that the value in $r[A] is the product of values in $r[B] and $r[C]
9. Division
    a. Append division to stream
    b. Output values in registers A, B, C and make sure that the value in $r[A] is the quotient of values in $r[B] and $r[C]
10. Bitwise NAND
    a. Append bitwise NAND to stream
    b. Output values in registers A, B, C and make sure that the value in $r[A] is the negation of the result of values ($r[B] AND $r[C])
11. Input
    a. Append input to stream
    b. Use output to verify that values being stored in $r[C] match the values that should be getting read in
12. Map
    a. Append map segment to stream
    b. Verify that the value placed in $r[B] is the correct index of segment by calling output, passing in $r[B] as its parameter
13. Unmap
    a. Append unmap segment to stream
    b. Verify that the pointer in memory at $m[$r[C]] is NULL
    c. Verify that the correct index was stored in “free_segments”

Functions to test memory

1. Load program
    a. void test_load_program() {
        /* map_segment returns the index of new segment in 32-bits*/
        UArray index = load_program(uint32_t register);
        assert(Seq_get(segment_sequence, index) != NULL;
    }
2. Map
    a. void test_map() {
        /* map_segment returns the index of new segment in 32-bits*/
        uint32_t index = map_segment(uint32_t register);
        assert(Seq_get(segment_sequence, index) != NULL;
    }

    b. Call the function map segment with known inputs and make sure that the new segment is mapped to a valid address
3. Unmap
    a. void test_unmapped() {
        uint32_t index = unmap_segment(uint32_t register);
        assert(Seq_get(segment_sequence, index) == NUL;L
        assert(Seq_get(free_segments, 0) == index);
        /* test unmap segment again and to make sure the next element in the sequence is equal to the index of segment that is free */
        index = unmap_segment(uint32_t register);
        assert(Seq_get(free_segments, 1) == index);
    }
    b. Call the function unmap segment with known inputs and make sure that the segment unmapped is NULL and free_segment holds the index of freed segment