Eli Intriligator (eintri01) and Max Behrendt (mbehre01)

Architecture

Modules

- memory.h
    - Contains functions relating to our segment abstraction
    - Hanson Sequence to simulate 32-bit segment identifiers
        - Index in sequence fits in 32-bits and can be referenced with a 32-bit address
        - Void pointer to actual malloc'd memory location
- instructions.h
    - Contains functions for all 14 instructions, as well as any corresponding helper functions
- main
    - Registers – array of 8 uint32_t elements
    - Hanson Sequence of mapped segments (uses the pointer casting trick to hold the 32-bit segment identifiers of each segment)
        - Each segment is a struct containing:
            - Hanson Sequence of instruction words
            - the segment's 32-bit address identifier

UM Instruction Set – Implementation and Testing

We will implement and test our UM instruction set in the following order:

1. Halt
    a. Test
        i. Call Halt
        ii. Program should stop running
2. Output
    a. Test
        i. Load number 3 into register A
        ii. Call output with register A
        iii. Program should print the number 3
    b. Edge Case
        i. Load a number > 255 into register A and call output; machine should fail
        ii. Test with all ASCII chars, verifying output by converting the printed chars to integers
3. Load Value

   a. Test
      i. Load the value 8 into register A
      ii. Call output on register A
      iii. Program should print 8
4. Add
   a. Test
      i. Place the numbers 100 and 10 into registers B and C, respectively
      ii. Call add on registers B and C, placing result in register A
      iii. Call output on register A
      iv. Check the result is correct
   b. Edge cases
      i. Repeat above steps with max int $(2^{32} - 1)$ in registers B and C
      ii. Call add on registers B and C, placing result in register A
      iii. Should throw a CRE
5. Multiply
   a. Test
      i. Place the numbers 100 and 10 into registers B and C, respectively
      ii. Call each operation on registers B and C, placing result in register A
      iii. Call output on register A after each operation
      iv. Check the result is correct
   b. Edge cases
      i. Test using numbers that yield a result that uses more than 32 bits
      ii. Should throw a CRE
6. Divide
   a. Test
      i. Place the numbers 100 and 10 into registers B and C, respectively
      ii. Call each operation on registers B and C, placing result in register A
      iii. Call output on register A after each operation
      iv. Check the result is correct
   b. Edge cases
      i. Test with a denominator of 0
      ii. Should call an unchecked runtime error
7. Bitwise NAND
   a. Place the numbers 5 and 10 into registers B and C respectively
   b. Call Bitwise NAND on registers B and C
   c. Call output on register A, it should be 1
   d. Do the same with the number 5 in both B and C, the output should be 0
8. Input
   a. Test
      i. Use input to place a value into a register
      ii. Use Output to print that register
   b. Edge Case
      i. Like with Output, try with all ASCII chars, using integer values of the ones we can't see

9. Map Segment
    a. Test
        i. Create a new segment of size 7 (using functions described in next section)
        ii. Use output to check that our memory sequence has a new segment in m[1] that is made up of 7 32 bit words, each containing all 0s
    b. Edge Cases
        i. Make sure we throw a CRE if asked to create a segment of size < 1
10. Unmap segment
    a. Test
        i. Create a new segment of size 7
        ii. Unmap the segment
        iii. Halt the program and use valgrind to check that memory was deallocated
    b. Edge cases
        i. Give the function an invalid segment identifier (i.e. an unmapped segment); machine should fail
        ii. Ask the function to unmap the 0th segment; machine should fail
11. Conditional move
    a. Test
        i. Place a value in register A and register B
        ii. Test Conditional Move with the value 0 in register C, and with a non-zero value
        iii. In the first case, the value in register A should remain unchanged, in the second, it should be come the value held in register B
12. Segmented store
    a. Test
        i. Create a new segment of size 7
        ii. Use segmented store to store a series of values in each of the 7 words in the segment
        iii. Use output to check in our sequence that m[1] holds a size 7 segment with each word corresponding to the correct value that we stored
    b. Edge case
        i. Give the function an invalid segment identifier; machine should fail
        ii. Give the function a word location that is outside the bounds of a mapped segment; machine should fail
13. Segmented load
    a. Test
        i. Create a new segment of size 7
        ii. Use segmented store to store a series of values in each of the 7 words in the segment
        iii. Call segmented load to load the seventh word into register A
        iv. Output the contents of register A and verify that the loaded value matches the stored value
    b. Edge case

        i. Give the function an invalid segment identifier; machine should fail
        ii. Give the function a word location that is outside the bounds of a mapped segment; machine should fail

14. Load program
    a. Test
        i. Create a new segment of size 7
        ii. Use Segmented Store to store a series of 7 instructions inside the segment that will print out a word when executed
        iii. Call load program and check that the new program prints the word properly
    b. Edge case
        i. Give the function the 0th segment and a non-zero desired counter location as parameters; program should advance program counter but not alter the 0th segment
        ii. Give the function an invalid segment identifier; machine should fail

UM Segment Abstraction – Implementation and Testing

We will implement our UM segment abstraction in the following order:

1. Memory identifier system
    a. new function – allocates memory for a given number of uint32_t elements and sets the value of each uint32_t to be zero
        i. Input: number of uint32_t elements (words) to allocate memory for
        ii. Output: 32-bit segment identifier corresponding to an index in the sequence
        iii. Test:
            1. Call new(5)
            2. Print if malloc was successful
            3. Print the value of the first word (should be 0) at the returned index in the sequence
    b. get function – gets pointer to the first word in a segment
        i. Input: 32-bit segment identifier corresponding to an index in the sequence
        ii. Output: returns a uint32_t pointer to first uint32_t that is allocated for a given segment
        iii. Test:
            1. Use the new function to allocate memory for 5 words and print the location of the first word.
            2. Use the get function with the corresponding index
            3. Make sure that the returned pointer points to the same location
        iv. Edge case:
            1. Give the function an invalid segment identifier; machine should fail
    c. free function – frees an indicated segment in the sequence.

i. Input: 32-bit segment identifier corresponding to an index in the sequence
ii. Output: none
iii. Test:
    1. Use new function to allocate memory for 5 words
    2. Use our free function to free the memory
    3. Run with valgrind to ensure no memory was leaked
iv. Edge case:
    1. Give the function an invalid segment identifier; machine should fail