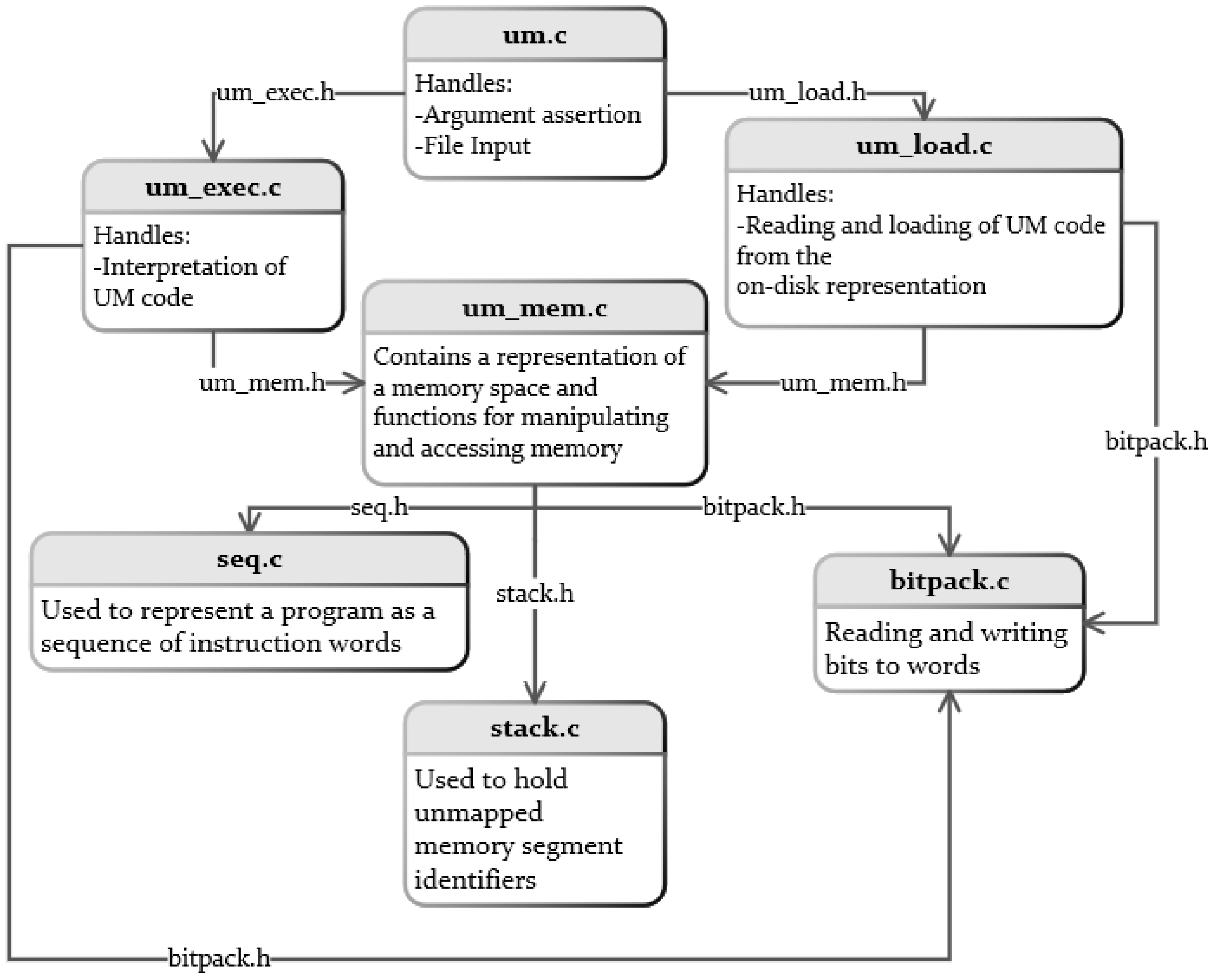
Timothy Colaneri
No partner
CSC411 Universal Machine README
Spring 2020

I believe this universal machine implementation functions correctly for all instructions, albeit slowly.

Acknowledgements:

I was pretty comfortable with the assignment here, I understood bit packing and I've done interpreters for stack machine languages before. Outside of the course material the only reference I had to make here was how to get the size of a file in bytes for the purpose of determining how many instruction words I will need to process. I found a method that uses fseek and ftell from the standard io library.

Final Implementation Architecture:


um.c
Handles:
-Argument assertion
-File Input
um_exec.h
um_load.h
um_exec.c
Handles:
-Interpretation of UM code
um_load.c
Handles:
-Reading and loading of UM code from the on-disk representation
um_mem.c
Contains a representation of a memory space and functions for manipulating and accessing memory
um_mem.h
um_mem.h
bitpack.h
seq.h
bitpack.h
seq.c
Used to represent a program as a sequence of instruction words
stack.h
bitpack.c
Reading and writing bits to words
stack.c
Used to hold unmapped memory segment identifiers
bitpack.h

um.c – Standard input and argument handling. Nothing special here.

um_load.c – This module reads the 32-bit instruction words from a file and populates those words into a memory space representation. It holds the secret of how on-disk UM programs are read into memory.

Um_exec- this module handles the interpretation of the UM code after it has been populated into a memory space. It uses a struct to represent an unpacked word, which consists of 4 integer components: the opcode, and registers a, b, c. Each instruction word is unpacked and then passed through a switch which will send the register values to the correct interpretation function. It holds the secret of how the UM language is actually interpreted.

Um-mem.c- this module contains a representation of a memory space. A struct is used for this, which consists of a stack and a sequence. The stack is used to hold onto and pass back in constant time any unmapped memory segments. The expandable sequence is used to represent a program as a sequence of instructions. It holds the secret of how UM memory is represented and accessed.

Benchmarking:

I inserted a counter into my main processing loop along with a switch to exit the program when 50,000,000 instructions have been executed. Midmark consists of ~82 million instructions, so I was able to run most of this program before it cut out. I ran midmark 4 times.

I received an average of 6 seconds.

Time Spent:

Analysis: ~1 hour
Design: ~1 hour
Again, as I was familiar with most of the concepts, I did not need to spend much time on analysis and design. I just really needed to figure out how to do it in C.
Implementation: ~16 hours
I managed to implement a hard-to-find bug that took me a few extra hours to solve here.