CS40 HW6 Design Doc (UM)

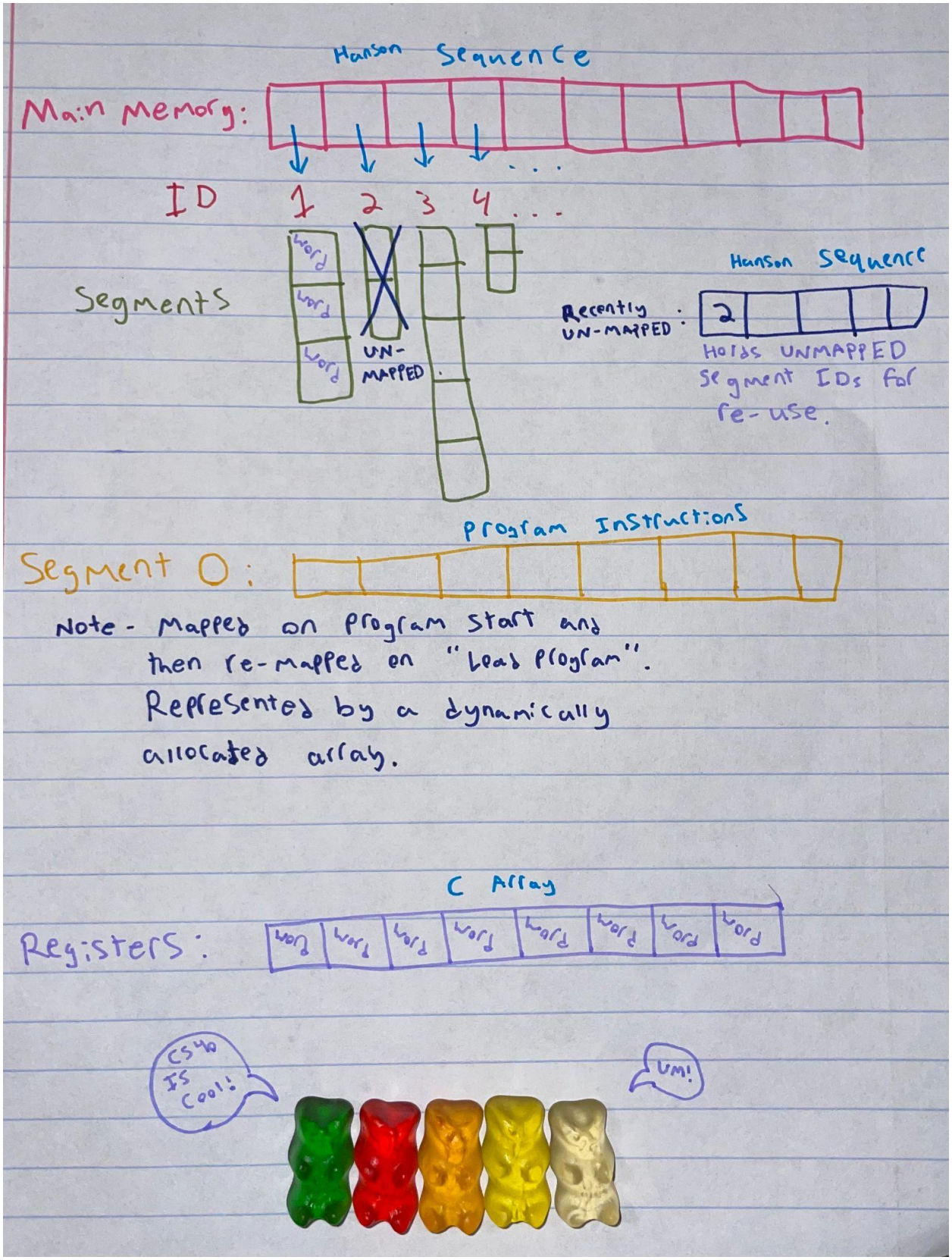
Adam Weiss (aweiss15) and Auriel Wish (awish01)

Architecture

- **Modules:**
    - **um.c**
        - Contains memory structures and calls functions through command loop (in main)
    - **arithmetic.c**
        - Contains arithmetic functions such as add and multiply
        - These do not need (and therefore do not have) direct access to memory segments or registers
    - **memory.c**
        - Contains functions that manipulate memory segments and have direct access to segments and registers
        - Module-to-module interaction (used for abstraction since the arithmetic functions shouldn't have direct access to registers):
            - Function to **get value in a register**
                - * getRegisterValue
                - * Purpose: Get the value in a register
                - * Parameters: The emulated registers, the desired register
                - * Returns: The value in the requested register
                - * Notes: None
            - Function to **set the value in the register**
                - * setRegisterValue
                - * Purpose: Set the value in a register
                - * Parameters: The registers in the instruction, the register number to change, the value to put in the register
                - * Returns: None

        - * Notes: None
    - Function to **gain access to a desired segment**
        - * getSegmentLocation
        - * Purpose: Get the location of a memory segment
        - * Parameters: The segment ID
        - * Returns: A pointer to the segment
        - * Notes: None
- **How the program works: (see picture below)**
    - The **memory segments** will each be **arrays of uint32_t**. The **segments** will be **stored in a sequence** such that their **index** in the sequence will be the **same as their segment ID**.
    - **Segment 0 will not be in the sequence** - it will be its own separate array that gets passed in as its own argument to functions. This is because we want to make sure it is **placed in a register** (or at least as much of it as can fit) because it will be accessed a lot.
    - **Segment IDs for new segments will be assigned** as follows:
        - Integer maxSegmentID will keep track of the highest ID that has been used
        - Sequence recentlyUnmapped will contain all IDs that were once in use but were then unmapped
            - Whenever a segment is unmapped, add its ID to recentlyUnmapped
        - When a new segment is mapped, check to see if recentlyUnmapped has any IDs in it
            - If it does, then use the most recent one as the ID for the new segment (and remove the ID from recentlyUnmapped)
            - If it does not, then use 1 + maxSegmentID as the ID for the new segment and increment maxSegmentID

- The **8 registers will be a C array** that will contain uint32_t data values. The choice to make this a C array is based on the idea that the array will have a constant length (8), so the size can be defined at compile time.
- After booting up the UM and setting the initial state, main will run a **command loop** that **will continuously fetch, decode and execute instructions**.
  - An instruction is **fetched from segment zero**.
  - That **instruction is decoded into its opcode and registers** (and value if it's a load value instruction.)
  - The command loop will **call a function based on the opcode** and pass in the relevant data and data structures as parameters.
  - The command loop will terminate when it decodes an instruction with the opcode of HALT.
  - The heap memory of the emulation is then freed and the program finishes.


Hanson Sequence
Main Memory:
ID 1 2 3 4 . . .
Segments
word
word
word
UN-MAPPED
Hanson Sequence
Recently UN-MAPPED:
2
Holds UNMAPPED Segment IDs for re-use.
Program Instructions
Segment 0:
Note - Mapped on program start and then re-mapped on "Load Program". Represented by a dynamically allocated array.
C Array
Registers:
word
CS40 IS Cool!
UM!

Implementation

1. Read in original file and place instructions in segment 0
    a. Testing: print the words to output (as numbers) as they are put into segment 0. Since we know what instructions we are inputting, we know what should be outputted.
2. Initialize all variables (sequences, integers, …etc) to keep track of segments/memory, IDs, instructions, …etc
3. Create command loop that runs until halt instruction is given
    a. Fetch: Get the current instruction using the program counter
        i. Testing: Print the current instruction to standard output, convert to binary, and make sure it is the expected instruction
    b. Decode: Write functions to get the opcode and register(s) in the given instructions
        i. Testing: Print out the opcode and registers and make sure they match what was in the printed instruction
    c. Execute: Each instruction gets its own function. Note: unit testing is used for testing these functions. The functions are written and tested in order from top to bottom because earlier functions will be needed to test later ones.
        i. Write the halt function
            1. Testing: call halt with other instructions afterward and make sure that none of them run
        ii. Write load value function
            1. Testing:
                a. Load different values into different registers (not many test cases for this)
                b. Load a value that is greater than 25 bit into a register. (Should shave off the higher order bits).
        iii. Write I/O functions
            1. Testing:
                a. Output - load different values into registers and output…
                    i. Typeable ASCII values

ii. Untypeable ASCII values
iii. Value greater than 255 - should fail an assertion

b. Input
i. Typeable ASCII values
ii. Untypeable ASCII values (by piping in file that contains them)
iii. Pipe file to test end-of-file input, which should enter -1 into the register

iv. Write arithmetic instructions

1. Testing: all operation solutions will be outputted and compared to manually performed solutions. This means that ultimately, all test case solutions will eventually have to be between 0-255 (and preferably typeable characters)

a. Conditional Move
i. Conditionally move numbers (not many test cases needed)

b. Multiply
i. Multiply small/medium sized numbers
ii. Multiply numbers whose multiply to a number greater than 2^{32} (to make sure modulo is done correctly)

c. Add
i. Add small/medium sized numbers
ii. Add numbers whose sum is greater than 2^{32} (to make sure modulo is done correctly)
1. Since load value can only go up to 2^{25}, use multiply to get numbers in 2 registers that are big enough to go past 2^{32} when they are added together

d. Divide

i. Divide different numbers (we don't need to account for dividing by 0, as stated in the spec)

e. Bitwise NAND

i. NAND 2 very large numbers such that all bits after the 7th least significant bit will end up being 0. This way, we can easily output the result and make sure it is correct

v. Write segment handling functions

1. Testing:

a. Map segment (check with valgrind)

i. Map new segment with different amounts of words

ii. Map new segment with 0 words (when accessing these segments, it is undefined behavior since the index of the requested word will automatically be outside of the segment bounds)

iii. Map new segment once segments have been unmapped (once unmap function is written)

iv. Use segment load and segment store (once written) to test segment contents

b. Unmap segment (check with valgrind)

i. Unmap one segment

ii. Unmap multiple segments in a row

iii. Map and unmap segments consecutively

iv. Unmap IDs that aren't mapped (spec says undefined behavior).

v. Unmap segment 0 (also undefined behavior)

c. Segment Load (check by sending to output)

i. Load words in segment 0

ii. Load words in a newly mapped segment

    d. Segment Store (check results using segment load and sending that to output)
        i. Store words in segment 0
        ii. Store words in a newly mapped segment
    e. Load program (check for deep copy, not just pointer copying)
        i. Load program from segment 0 (should only change where the program counter points to. Make sure that the next instruction is the intended instruction)
        ii. Load program from other segments with different lengths
        iii. Load program from segments with length 0 (undefined behavior since the program counter will automatically point to an instruction outside of segment 0)
4. Free all outstanding memory (segments, sequences, … etc)
    a. Testing: valgrind
5. More Architecture testing (function prototypes)
    a. void debugPrintRegisters(uint32_t regs[])
        i. Prints out the data in the registers array
        ii. This will be used to test the getRegisterValue function and the setRegisterValue function
        iii. It will also be called in the fetch decode execute loop so we can make sure the registers are what we expect them to be
    b. void debugRecentlyUnmapped(Seq_t recentlyUnmapped)
        i. Prints out all recently unmapped IDs, which are held in a sequence
        ii. Makes sure that all IDs that can be reused are placed in the sequence
    c. void viewProgram(Um_instruction program[])
        i. Prints out the instructions in segment 0, which are in an array of instructions

ii. This lets us know what each step of the program is supposed to be doing

d. void viewMainMemory(int lowID, highID, Seq_t mainMemory)

i. Prints out the contents of main memory between two segment IDs in a readable way. Each segment is an array, and each array is in the mainMemory sequence

1. If a segment has been unmapped in the range, there will be a special output stating that this is the case

ii. This allows us to see what is in a segment after calling segment store along with seeing what segments have been mapped within a specific range. The range of IDs allows for the printing of a small amount of memory instead of the entire memory every time.