Joseph A. Boyle

Three programs have been created:

- **Y86emul**: Given a Y86 program in opcode format, will execute the program.
- **Y86dis**: Given a Y86 program in opcode format, will recreate the program in Y86 assembly instructions (converts opcodes into mnemonics).
- **Y86asmbl**: Given a Y86 program in assembly, creates the opcodes to be used by **Y86emul**. Thus, **Y86dis** and **Y86asmbl** effectively reverse one another.

There are also two sample programs provided:

- prog3.y86_gen (mnemonics stored in prog3.y86): calculates the sum of 1 + 2 + 3 +... + 100.
- prog4.y86_gen (mnemonics stored in prog4.y86): simple function call, nothing fancy, used for testing the assembler.

How It Works

All programs reference Y86tools, which creates a collection of functions and definitions:

- **Memory**: set and get byte/long from/to memory.
- **Registers**: set and get register values.
- **Program Loading**: program interpretation, loading values into memory.
- **Instruction Translations**: Converts opcodes to mnemonics, and vice-versa.
- Definitions for: register IDs, Opcodes, Status codes.
- Status flags

Tricks:

- Every opcode is stored as an enum for easier switch statements (ie: NOP = 0x00, SUBL = 0x61, etc). Thus, interpreting an instruction is an O(1) operation.
- When interpreting an instruction, opcode is set by the above method, and then rA is set to the next 4 bits, rB to the 4 bits after, and d to the 32 bits after.
    - This occurs regardless of whether or not the instruction actually *uses* these variables.
    - In doing so, we can pass the instruction struct to execute and everything is fine and dandy, regardless of what the instruction is. Execute just has to focus on doing what it is told.
    - Decode doesn't have to figure out which parameters to pull. In doing all of them, a switch statement isn't needed.
- In futil, we define several helpful file operation functions, such as open/close, string appending, and token fetching.
    - The tokenizer function reads the file in place and finds the first token that either is encapsulated by quotation marks or is bounded by escape characters. O(N) operation.
    - Tokens are only accessed during program loading to set memory as expected. Afterwards, they are not needed.

Initiating:

- Load file passed by user, die if it does not exist.

- Process the file. If this returns an ecd rror, die before proceeding to program execution.

<u>Program Loading </u>(loadProgramIntoMemory(FILE *file)):

- Grab the next token (consecutive characters with characters <= 32, unless enclosed by quotation marks) from the file.
- If this token is ".size", grab the next token and set the size of the memory block to this token. Continue.
- Otherwise, if the token is ".string": starting at the memory address indicated by the next token, store the token that follows. Continue.
- Otherwise, if the token is ".text": starting at the memory address indicated by the next token, store the token that follows. Continue.
- Otherwise if the token is ".long" or ".byte": starting at the memory address indicated by the next token, store the token that follows. Continue.
- Upon reaching EOF, if .size and .text never set, return -1 to indicate an error.

<u>Program Execution</u>:

- While status is AOK:
    - execute(decode(fetch()))
        - Fetch: return program counter and then increment it by one.
        - Decode: Create a new Instruction struct, setting the opcode to the integer at the indicated addy.
            - Set rA to the left 4 bits of memory[addy + 1], and rB to the right 4 bits of memory[addy + 1]
            - Set d to the long stored starting at addy + 2.
        - Execute: Using a switch, find the opcode that's currently being executed.
            - Utilizing rA, rB, and d, execute the instruction.
            - Increase count as needed (ie: if the instruction uses the register byte and immediate bytes, will increase count by 5 so that count points to the new instruction).
            - If instruction isn't found, throw an error.
- If status isn't HLT, print the error that occurred.

Y86dis works functionally the same, except it does not execute any instructions. It uses the same form of **Y86emul**'s execute function, but only to correctly increment the count variable. During every call to execute, **y86dis** will print the instruction to both console and a file via the methods in y86tools:

- Convert the opcode to mnemonic equivalent
- Given rA, rB, and d, print whichever arguments are relevant to the given instruction.
    - EG: rrmovl will print the values of rA and rB
    - EG: nop will print no arguments.

Y86asmbl works in two stages:

- First, we create a symbol table. This symbol table converts all "symbols" of the form ".symbolName:" into the addresses at which they occur. Thus, we can refer to functions by ASCII names instead of addresses.

    - To do so, we search through the entire file, searching for any occurrences of a symbol. We then store it in a symbol table along with its address relative to the beginning of the program.
- Next, we iterate over the file again, this time replacing all occurences of “.smybolName” with the address that matches with the symbol “symbolName”.
- When we find non-symbols, we simply convert the data into the instruction structure we previously discussed. Thus, converting from the instruction struct to opcodes is as simple as using the previously defined conversion functions for y86emul & y86dis.

Efficiency

In terms of space efficiency, both Y86emul and Y86dis will roughly be O(N + C), where N is the size of the memory chunk requested by .size, and C is the constant size of the Instruction struct and several constant-sized variables that are statically shared. Thus, the space efficiency is roughly O(N).

In terms of run-time efficiency, both programs run in roughly O(N) time. Of course, the execution time varies moreso on the instructions within a program and not the length of the program -- a Y86 program which produces an infinite loop with just a few instructions will take infinitely longer to execute than one thousand Y86 instructions.

Y86asmbl runs in O(N) time and space complexity. It is limited by the size of the file.