Liam Davies
CS 211 – Computer Architecture
Section: 11

Y86emul.c ReadMe

Description:

The Y86 Emulator program takes a .y86 file name as a command line argument, locates it in the current directory, parses it, then creates and properly sets up the allocated space for program operation and finally executes the .y86 program.

Y86emul.c does this process in the following way. First it checks to see if the proper number of arguments was given or if the "–h" flag was entered. It closes if there was an improper number of arguments and prints "Usage: y86emul <y86 input file>" if the help flag was entered. Next, it makes sure the input file is actually a .y86 file and closes if it isn't. Then it begins to open in file in read mode. It parses the file by first breaking it into lines and then breaking those lines up by spaces. It checks to see if each directive is valid, then performs the action that directive does. If this has an invalid syntax it will be caught. The file closes when there are no further lines.

Once memory is set up the fetch, decode, execute cycle begins. In y86emul.c this consists of a switch statement that calls methods from memory.h when needed.

Switch Statement Operation & Memory.h:

Memory.h keeps track of all of the memory operations of the current program. It contains all of the methods that directly change the state of the allocated memory. These methods are called either by the switch statement or by the program when it reads the directives. Memory.h contains an integer array of size 8 to represent the registers, 3 integers to represent the flags, an integer to track the highest memory position, a *byte* * representing memory and two unions, *byte* and *int_Char*.

- *byte* Union
  - This union contains a struct (the size of a byte) and a char (same size)
  - The struct (called *nibb*) contains two unsigned integers that are nibbles of size 4, meaning they add up to a single byte and can each contain values 0-15 (as is represented by the hex characters in the .y86 file)
  - The char is just a regular char which is the size of a single byte which matches the size of the struct
  - This union allows for the easy conversion of two smaller sub-bytes into a byte to be stored in memory and vice versa to read instructions

- *int_Char* Union
    - This union contains a struct (the size of 4 bytes) and an integer (same size)
    - The struct (called *chars*) contains 4 chars to represent the individual bytes of the larger, little endian integer.
    - The integer is just a regular integer which is the size of 4 bytes which matches the size of the 4 chars
    - This union allows for the easy conversion of 4 separately tracked bytes into an integer

Each iteration of the switch statement calls upon one of the methods in memory.h (except nop and halt which just increment or stop the program respectively) which uses these variables and unions to manipulate the data in memory to replicate the desired action. Each method is also responsible for moving the instruction pointer along as information is read from memory. Each time the instruction pointer is moved, memory.h checks to make sure it is a valid position that has been allocated and throws an error if it is not. If an invalid command is reached, an error is thrown. Everything does what it is supposed to do according to the spec.

Specify readX and writeX:

The methods for readX and writeX read or write to the terminal window, either a single char (if the fn code is 0) or a 4 byte integer (if the op code is 1).

When the program hits a halt command it will end free the allocated memory and print out AOK and HLT to tell the user that the program successfully finished without an error.

<u>Challenges:</u>

The hardest part of writing this program had to be figuring out how I wanted to represent and manipulate the memory. I went through several models before deciding on the *byte* struct array which allows for the easiest access to the information stored in memory, since I don't need to put the char from a char array into the struct every time I need info out of it.

Debugging was also moderately difficult but I had built a test header that had a bunch of testing methods to make sure all of my code worked. This allowed me to isolate each method and test it alone by manipulating and printing memory in a cycle to see how it changed.