Joseph Jaeschke
CS 211
Section 11
y86 Emulator

y86 Emulator (pa4)

The y86 emulator and the y86 disassembler are very similar in design. Both programs needed to have file I/O in order to read in the y86 files and create a memory image. For the dissembler, the only necessary parts of the y86 file that needed to be read was the .size directive and the .text directive. Since the disassembler only needs to print out the commands, it does not need to know any of the values that are to be stored in memory therefore it is not necessary to read in all the .byte, .long, and .string directives. On the other hand, the emulator very much needs to know every single value that will be mapped into memory so nothing from the y86 file can be ignored, and it all must be read in and mapped to the memory image.

In both the disassembler and the emulator the body of the main function, besides file I/O, is a switch case to determine what the current command to be executed is. Every opcode is represented by one case in the switch statement. From there the disassembler will call the function corresponding to that opcode in order to print the relevant information about it such as the mnemonic, the register names, and the offset. The emulator however will still call the function corresponding to the opcode, but rather than printing it will change the states of its registers, main memory array, the flags, the instruction pointer, or the emulator status in order to simulate what the y86 code does. This effectively runs a y86 program. For the disassembler, this is pretty much as far as it goes, only accessing memory linearly and printing, but the emulator goes a lot more in depth.

The main memory of the emulator is an array of unsigned characters. The size of this array gets set to the value marked by the .size directive. The array then gets initialized with the machine instructions and the values to be stored in memory. The registers of the emulator are an array of eight unsigned integers. Because the values in these registers are not represented in the same format as the

memory, they have to be converted back and forth when going to and from memory. Writing to memory uses bit shifting to break up the four byte integer into four separate bytes and then write each one to memory. Reading from memory to a register makes use of some cleaver type casting to access four bytes of memory at once and save it to an integer.

For the emulator states each one is handled differently. The status will become “HLT” when the halt instruction is called which will then cause the program to stop. This is how most programs will end given that everything ran normally and there were no errors. Another possible ending status is “ADR”. The emulator will end with an ADR status if somewhere in the code there was an invalid memory access. If an invalid access is attempted, the emulator will recognize this, change its status to ADR then exit. Another ending status is “INS” which would become the ending state if the emulator was trying to run an invalid opcode. The last possible ending status is “AOK”. The emulator will only end on this if it is terminated using the ctrl+c command from terminal. If the user was to enter ctrl+c while the emulator is waiting for input, the emulator will immediately stop. Since all the memory accesses and opcodes are valid and since a halt command was not reached, the emulator will have an end state of AOK.