//Readme for y86emul

Usage:

y86emul [-h] <y86 input file>

y86emul is a program that is capable of reading in .y86 input files, then completing the fetch-decode-execute instruction cycle on the instructions contained in the given .y86 input file.

y86emul works in three main phases, setup, read-in and the execution.

//Setup

The setup works by initializing the variables, registers, and flags necessary for the execution and storage of the y86 program. Some notable variables in the setup are:

membuf[]: the memory buffer where all memory is fetched to, decoded, and executed.

flag[]: the logical flags required for jump instructions (overflow flag, zero flag, sign flag).

reg[]: the eight registers used in y86 memory manipulation.

debug: the variable that turns on debug mode, which prints out useful information at runtime.

//Read-in

The read-in is a while loop that utilizes sscanf to move line by line down the given input file. It is capable of reading 5 directives, .size, .string, .long, .byte, and .text, of which .text and .size are required directives for all .y86 files.

.size works by simply callocing the given size value.

.string works through a simple memcpy of the given string to its given location

.long works by a similar memcpy on an unsigned int.

.byte works by a similar memcpy on an unsigned int formed by calling HexAsciiToDec on the given byte.

.text works by creating a loop that essentially runs .byte over and over.

//execution

The execution phase is a loop that moves byte-by-byte through the memory buffer, performing operations as the loop hits opcodes. It stops if it reaches an invalid instruction, or if the opcode is 10 (HALT). There are 26 instructions that alter the memory in some way/

//movls

rrmovl: Register to register movement, works through memcpy.

irmovl: immediate to register movement,^^^^^^^^^^^^^^^^^^^^

rmmovl: register to memory movement,   ^^^^^^^^^^^^^^^^^^^^

mrmovl: memory to register movement , ^^^^^^^^^^^^^^^^^^^^

movsbl: memory to register movement,   ^^^^^^^^^^^^^^^^^^^^, however moves only a single byte which is then sign extended.

//ops

addl: Simple Addition of registers,   (B=B+A) sets flags accordingly.

subl: Simple Subtraction of registers   (B=B-A) ^^^^^^^^^^^^^^^^^^^^^^

andl: Logical ANDing of registers, uses & (B=B&A) ^^^^^^^^^^^^^^^^^^^^^^

xorl: Logical XORing of registers, uses ^ (B=B^A) ^^^^^^^^^^^^^^^^^^^^^^

mull: Simple multiplaction of registers   (B=B*A) ^^^^^^^^^^^^^^^^^^^^^^

//jmps

jmp: Unconditional jump.

jle: Jump when less than or equal: SF xor OF | ZF

jl: Jump when less than: SF xor OF

je: Jump when equal to: ZF

jne: Jump when not equal to: !ZF

jge: Jump when greater than or equal: SF == OF

jg: Jump when greater than: SF == OF &!zf

//stackstuff

push: Pushes register to the stack(SP handled by input code) and decrements pointer.

popl: Pops something from the stack and stores in register and increments pointer.

call: Calls a push, however pushes the instruction pointer of the next instruction.

ret: Calls a pop, and sets the instruction pointer to whatever was popped.

//i.o.

readb: Reads two ASCII characters, then saves their bit value. I.E. input="14" -> value=0x14

readl: Reads in a 4 bit integer and saves its value.

writeb: Prints out a single bit value.

writel: Prints out a 4 bit integer.

//Comments

This program was the best thing I've coded in my entire life. The challenge level was just right, and it was the most fun thing I've ever worked on. Building this whole thing from scratch was an amazing project to get to work on and honestly, it tested basically everything we learned throughout the course of the entire year. Every project we worked on in the past also played a role in building this one, whether it was string management learned in Tokenizer, conversion a la Calc, or even understanding Assembly from Mystery. This was a great cumulative assignment. Even getting stuck and working my way through GDB proved to be enjoyable as I was able to really learn the ins and outs of my program, and what the impact of every function was.

Looking back on the code is weird, because I spent so much time and effort in getting everything right, but I feel like I could redo this in a few hours at most. The time I spent getting so acclimated with memcpy, fgets, scanf, the stack, hex-addresses, jumps, is worth a lot.

The best (Read: worst) part about this program is that small errors cause huge problems in output. After my first run of finishing the commands, my program would simply not work as intended. It took me over 5 hours of GDB and rereading of my own code to change only ~5 lines to fix my errors.