Y86 Assembler & Disassembler Emulator

Liu Liu

July 15, 2016

1 Introduction

Y86 can be treated as a "educational purpose" X86 language. It has similar syntax as X86, which we've been talking about. This assignment is designed to help you really understand how the fetch-decode-execute cycle works as well as the idea of program-as-data. It will require a substantial implementation effort. The usual warning goes double for this assignment: *do not procrastinate.*

2 Y86 Architecture

The Y86 architecture has eight registers, three condition codes, a program counter and memory that holds machine instructions and data. All addresses, immediate values and displacements are **32 bit little-endian** values.

Each of the eight registers has a 4-bit ID that fits into the Y86 instructions. The eight registers and their encoding in the Y86 machine instructions are as follows:

- %eax 0
- %ecx 1
- %edx 2
- %ebx 3
- %esp 4

- %ebp 5
- %esi 6
- %edi 7

The condition codes are single-bit flags set by arithmetic or logical instructions. The three condition codes are:

- OF overflow
- ZF zero
- SF negative

You might have noticed that %eip is not here. The program counter is the address of the next machine instruction to execute. It will be simulated by a variable in your simulator. Total memory size will have to be determined as part of emulator execution.

The Y86 instruction set is modeled on the larger Intel-x86 instruction set, but is not a direct subset.

3 Y86 Emulator

An emulator is hardware or software that duplicates (or emulates) the functions of one computer system (in this case Y86 instructions) on another computer system (the Intel host). The Y86 instructions are different from the Intel x86 instructions. Your assignment is to write an emulator for the Y86 instruction set.

Implement a program **y86emul** that executes Y86 executable files. Your program **y86emul** should support the following user interface(Command-line):

$ ./y86emul [-h] [y86 input file]

where [y86 input file] is the name of a Y86 input file, whose structure is defined in Section 5. If -h is given as an argument, your program should just print out help for how the user can run the program and then quit.

Erroneous inputs should cause your program to print out:

ERROR: [an informative error message]

Otherwise, your program should run the Y86 code which may read whatever Y86 inputs from the terminal and/or write Y86 outputs to the terminal as your Y86 program executes.

Your emulator will read the input file, allocate a chunk of memory of appropriate size which will act as the Y86 address space, populate that chunk of memory with data and machine instructions from the input file and then starts execution of the Y86 machine instructions. The entire address space of the Y86 program fits within this block of allocated memory. **The lowest byte of the address of the allocated block is Y86 address 0** and all other Y86 addresses are offsets within this block.

Your emulator will fetch, decode and execute Y86 instructions. This execution is tied to an status code that may take on the value **AOK, HLT, ADR, INS**.

- AOK means that everything is fine, no detected errors during execution.
- HLT means a halt instruction has been encountered, which is how Y86 programs normally end.
- ADR means some sort of invalid address has been encountered, which also stops Y86 program execution.
- INS is set for an invalid instruction, which also stops Y86program execution.

Your emulator should print out how the Y86 program execution ended.

4 Y86 Instructions

The definition and encoding of the Y86 instructions are presented in an attachment and in the Bryant and OHalloran book in Chapter 4.1. The instruction set presented there is minimal and almost functionally complete.

What is missing are instructions for input and output.

Your Y86 emulator will also handle instruction to read from and write to the terminal. Read byte and read long instructions.

For example the read instruction. If the instruction is **readB D(%eax)**. The first byte should be the instruction opcode : C0. Second byte would be the operands %eax. %eax is represented by 0 and we don't have the second operand. In this case we use 'F' to represent no-operand. So the second byte would be 00001111(0F). The next 4 bytes would be used to place displacement 'D'. The readL instruction is changing the second half of the opcode byte to '1'.

encoding bytes					
0	1	2	3	4	5
C0	0F	4 Bytes offset			
C1	0F	4 Bytes offset			

Table 1: read instruction

Another example on multiplication instruction. The opcode for mull is "64" and the two arguments are %eax and %ecx.

The instruction **mull %eax, %ecx** will have the following byte representations.

encoding bytes	
0	1
64	01

Table 2: mull instruction

5 Y86 Input File Format

The input file to your Y86 emulator does not contain Y86 assembler instructions. Instead, it contains an ASCII representation of the information needed to start and execute a ready-to-run program, including Y86 machine

instructions. An input file will contain directives that specify data and Y86 machine instructions.

5.1 Specifying Total Program Size and Base of Stack

The **.size hex-address**

This species the total size of the program in memory (in bytes). The hex address also specifies the address of the bottom of the stack. The Y86 stack grows from larger addresses toward smaller addresses. There should be only one .size directive in the input file.

5.2 Specifying String Constants

The **.string hex-address "double-quoted string"**

specifies a string contained in the double quotes. The hex-address species the location of the string in the memory block allocated by your emulator. The input string will contain only printable characters and nothing that requires a backslash.

5.3 Specifying Integer Values

The **.long hex-address decimal-number**

specifies a 4-byte signed integer. The hex address species the location of the value and the decimal number is the initial value at that Y86 address. All Y86 arithmetic is 4-byte signed integer arithmetic.

5.4 Setting Aside Chunks of Memory

The **.bss hex-address decimal-size**

specifies a chunk of uninitialized memory in the Y86 address space. The hex address specifies the location of the uninitialized chunk and the decimal size specifies the size.

5.5 Specifying One-Byte Values

The **.byte hex-address hex-number** specifies a one-byte value. The hex address specifies the location of the byte and the initial value is the hex number whose value is between 00 and FF.

5.6 Specifying Y86 Machine Instructions

The **.text hex-address ASCII string of hex Y86 instructions** specifies the Y86 machine instructions. The hex address specifies where the machine instructions should be placed in the Y86 address space. This same address is also the initial value of the Y86 program counter. The ASCII string in a single long encoding of the hex bytes on the machine instructions, two characters per byte, no leading "0x".

Note that the directives may come in any order. Some directives may appear more than once, but the **.text** directive and **.size** directive will appear only once. Your emulator may warn about overlapping directives, but that would not stop Y86 program execution. A missing .text directive is an error and should cause termination of the emulator with an informative error message. Any other detected deformity of the input le should also cause termination of the emulator with an informative error message.

6 Extra Credit: Y86 Disassembler

For extra credit, you can write a Y86 disassembler as a separate executable. A disassembler reads machine instructions and produces an assembly language listing as output.

You can implement a program y86dis that disassembles Y86 machine instructions. Your program **y86dis** should support the following user interface:

$ ./y86dis [-h] [y86 input file]

where [y86 input file] is the name of a Y86 input file, whose structure is defined in Section 5.

If -h is given as an argument, your program should just print out help for how the user can run the program and then quit.

Erroneous inputs should cause your program to print out:

ERROR: [an informative error message]

Otherwise, your program should disassemble the Y86 code in the **.text** section of the input file.

Your disassembler program will read the input file, find the **.text** section and print out assembly instruction, one instruction per line. The assembly instruction will include instruction mnemonic and operands. You can also print out the Y86 address of each instruction (in hex), and the ASCII representation of the hex bytes of the corresponding machine instructions.

7 Submission

IMPORTANT NOTE: You may write your code on any machine and operating system you desire, but the code you turn in MUST be able to compile and run on ILAB or a zero grade will be given, since our TAs will grade your work on ILAB. Be sure to compile and execute your code on an Ilab machine before handing it in. This has been clearly stated here and NO EXCEPTIONS will be given.

A tar ball named pa2.tar.gz shall be submitted with your code files:

- yourcode.c
- yourcode.h
- makefile
- Report.pdf, it contains a brief description of the program. No code needed here, but only a general discussion of how do you solve the problem and what part do you think it's tricky and how you solved it.

The makefile should have at least 2 rules:

- y86emul : build y86emul executable
- clean : clean up the objects + executable
- *y86dis, build y86dis executable if you go for the extra credit

Suppose that you have a directory called pa2 in your account (on the iLab machine(s)), containing the above required files. Here is how you create the required tar ball. (The ls commands are just to help show you where you should be in relation to pa1. The only necessary command is the tar command.)

```
$ ls pa2
$ tar -zcvf pa2.tar.gz pa1
```

You can check your pa2.tgz by either untarring it or running tar tfz pa2.tgz (see man tar).