CS 211: Computer Architecture, Summer 2015
Programming Assignment 4: Y86 Emulation

1 Introduction

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
```

The program counter is the address of the next machine instruction to execute. Total memory size will have to be determined as part of emulator execution.

The Y86 instruction set is modeled on the larger Intelx86 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:

```
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 Y86 program 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 the slides (also an attachment) and in the Bryant and O'Halloran 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

Encoding Bytes
0 1 2 3 4 5

readb d(rA) C0 rA F D
readl d(rA) C1 rA F D

The readb instruction reads a single byte from the terminal into memory, and the readl instruction reads a single 4-byte little-endian word into memory. The little-endian word is already compatible with the little-endian Intel architecture, where your emulator will run. Both instructions set the ZF condition code. On normal input, the ZF flag is set to zero, on end-of-file the ZF flag is set to one. Testing the conditon code is how the Y86 code can detect end of file. Note that the F in the second half of the second byte means ”no register”, just as it does for some of the other Y86 instructions. Both instructions are six bytes long with a 4-byte offset D.

Write byte and write long instructions

> Encoding Bytes
> 0 1 2 3 4 5
>
> writeb d(rA) D0 rA F D
> writel d(rA) D1 rA F D

The writeb instruction writes a single byte from memory to the terminal, and the writel instruction writes a single 4-byte little-endian word from memory to the terminal. Neither instruction alters the condition codes. Both instructions are six bytes long with a 4-byte offset D.

Multiplcation Instruction

> Encoding Bytes
> 0 1 2 3 4 5
>
> mull rA,rB 64 D

The mull instruction multiplies the values in rA and rB and leaves the product in rB. This instruction set the condition codes. The instruction is two bytes long.

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` directive

> .size hex-address

This specifies 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` directive

> .string hex-address ”double-quoted string”

specifies a string contained in the double quotes. The hex-address specifies 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` directive

> .long hex-address decimal-number

specifies a 4-byte signed integer. The hex address specifies 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` directive

> .bss hex-address decimal-size

specifies a chunk of uninitialzed 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` directive

> .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, inclusive.

5.6 Specifying Y86 Machine Instructions

The `.text` directive

> .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 file should also cause termination of the emulator with an informative error message.

6 Y86 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. You are free to make this as complete as posible by adding features of your own for more extra credit.

7 Submission

You have to e-submit the assignment using Sakai. Your submission should be a tar file named `pa4.tar` that can be extracted using the command:

```
tar -xf pa4.tar
```

Extracting your tar file must give a directory called `pa4`. This directory should contain:

The `pa4` directory in your tar file must contain:

- `readme.pdf`: This file should describe the design and implementation of your Y86 emulator and disassembler, and any challenges you encountered in this assignment.

- `Makefile`: there should be at least three rules in your Makefile:

    1. `y86emul`: build your `y86emul` executable.
    2. `y86dis`: build your `y86dis` executable.
    3. `clean`: prepare for rebuilding from scratch.

- source code: all source code files necessary for building `y86emul` nd `y85dis`. Your source code should contain at least 2 files: `y86emul.c` and `y86dis.c`.

We will compile and test your programs on the iLab machines so you should make sure that your programs compile and run correctly on these machines. You must compile all C code using the gcc compiler with the `-ansi -pedantic -Wall` flags.

8 Grading Guidelines

8.1 Functionality

This is a large class so that necessarily the most significant part of your grade will be based on programmatic checking of your program. That is, we will build a binary using the Makefile and source code that you submitted, and then test the binary for correct functionality against a set of inputs. Thus:

- You should make sure that we can build your program by just running `make`.
- You should test your code as thoroughly as you can. *In particular, your code should be adept at handling exceptional cases.*

Be careful to follow all instructions. If something doesn't seem right, ask.

8.2 Design

Having said the above about functionality, design is a critical part of any programming exercise. In particular, we expect you to write reasonably efficient code based on reasonably performing algorithms and data structures. Thus, the explanation and discussion of your design and implementation in the `readme.pdf` files will comprise a non-trivial part of your grade. *Give careful thoughts to your writing of this file, rather than writing whatever comes to your mind in the last few minutes before the assignment is due.*

8.3 Coding Style

Finally, it is important that you write "good" code. Unfortunately, we won't be able to look at your code as closely as we would like to give you good feedback. Nevertheless, *a part of your grade will depend on the quality of your code.* Here are some guidelines for what we consider to be good:

- Your code is modularized. That is, your code is split into pieces that make sense, where the pieces are neither too small nor too big.

- Your code is well documented with comments. This does not mean that you should comment every line of code. Common practice is to document each function (the parameters it takes as input, the results produced, any side-effects, and the function's functionality) and add comments in the code where it will help another programmer figure out what is going on.
- You use variable names that have some meaning (rather than cryptic names like `i`).

Further, you should observe the following protocols to make it easier for us to look at your code:

- Define prototypes for all functions.
- Place all prototype, `typedef`, and `struct` definitions in header (.h) files.
- Error and warning messages should be printed to `stderr` using fprintf.