EMAT10006 Assignment 3

Feb 2016

1 Plagiarism

A big warning at the top here: this is an individual assignment. You must not work together on this or copy each other's code. We will be checking for plagiarism and the penalties are severe.

2 Overview

In this assignment we will make a compiler for the ZX256 assembly language. The compiler will read assembly code and in several steps compile it to machine code giving a range of different output options.

3 Packaging requirements

Your program will be called "hd". You will submit your code to SAFE as a .zip file called "hd-1.0.zip". Inside the zip file is a directory called "hd-1.0" and inside that directory there will be

- Makefile
- README.txt
- zxpremain.c
- zxencmain.c
- zxccmain.c
- a folder called tests
- Other .c and .h files
- a folder called misc

The contents of the “tests” folder can be found as “tests.zip” on Blackboard (I’ll put it up soon). The code will compile to create several programs: zxpre.exe, zxenc.exe and zxcc.exe. These programs do different things that will be described below. Each “*main.c“ file contains the “main“ function for a different program. Only the main function of each program will be located in these files and the main functions will be relatively short. The rest of the code must go into other .c and .h files. You should think about how to break up the tasks required here into functions that are *reusable* so that you can use them in each of the different programs.

4 ZX256 ASM

The ZX256 is explained in more detail in the notes. Here we will briefly summarise the syntax of ZX256 assembly language. An example of ZX256 assembly is the program

```
message: db "Hello world!\n"  ; message

        ; Print hello world
main:   mov %a,0x04             ; sys_write
        mov %b,0x01             ; stdout
        mov %c,message          ; string to print
        mov %d,0x0d             ; length of string
        syscall                 ; invoke sys call

        ; Clean exit
        mov %a,0x01             ; sys_exit
        syscall                 ; invoke sys call
```

Comments are marked with semicolons. On any line all characters after a semicolon including the semicolon itself are not part of the program. The exception to this is if the semicolon character in a string (between double " quotes). Whitespace (spaces, tabs, blank lines) is generally not significant in the meaning of the program except that it separates Some lines are blank or only contain comments and whitespace. Other lines contain *instructions*.

An instruction line may have a *label* (e.g. "message" and "main" above). This syntax for a label is that it is an identifier followed by a colon that comes at the start of the instruction line. The syntax for an *identifier* is that it is a single word containing up to 8 characters consisting only of alphabet letters (a-z and A-Z), underscores (_) and digits (0-9). An additional restriction is that an identifier cannot begin with a digit. For example "loop2", "asd_asd", and "ALPHA_1" are valid but "1loop", "a.b" and "$a" are invalid. If there is a label then it is separated from the instruction by a colon (:). Otherwise the instruction may not contain a colon (except in a string). There may be whitespace before a label and between the label and the colon. A line which has a label but no instruction is invalid.

An instruction consists of *mnemonic* which is a short word that identifies the type of the instruction (e.g. "mov" or "syscall") followed by zero or more *operands*. The syntax rules for this are that the mnemonic is an identifier and that there must be whitespace between the mnemonic and any operands. Operands are separated by commas (,). Whitespace may appear before the mnemonic or between any of the operands and the commas or after any operands. For example:

```
     ; Valid instructions:
mov %a,0x01          ; Comments
            mov           %a                ,                 0x01
mov %a  , 0x01
jmp 0x12
     syscall
     ; Invalid (needs space):
     mov%a,0x01
```

Operands can have several forms. An operand may be a register name e.g. "%a" or a hexadecimal byte literal "0x01", a string literal enclosed in double quotes ("), a label identifier, a direct memory address or an indirect memory address.

Register names begin with % e.g. "%a" and may not contain whitespace. Byte literals begin with "0x", cannot contain whitespace and must always have 2 hexadecimal digits so the number 1 is "0x01". A direct memory address looks like "(0x12)" i.e. it is a byte literal between brackets and may not contain whitespace. An indirect literal looks like "(%a)" i.e. it is a register name enclosed in brackets and may not contain whitespace.

A string literal is marked with double quotes. Between the opening " character and the closing " character any printable characters can appear. Newline characters are marked with \n. Quote characters within the string are marked as \". A \character is given as two i.e. \\. A semicolon or colon may appear in a string and are simply characters in the string without indicating labels or comments.

5 zxpre

This section describes the zxpre.exe program. The role of this program is to preprocess the ZX256 assembly code. Preprocessing is used as part of any compiler and forms the first step in the compilation chain. You should write functions for preprocessing ZX256 assembly and use them in zxpre.exe but then also *reuse* them in the other programs. Think about how to make functions that are generically useful rather than functions that are only useful in one program.

The zxpre.exe program preprocesses assembly code and outputs the preprocessed form. This means that all comments have been stripped out all blank lines removed and whitespace normalised. So for example if the hello world program above was in a file called "hello.asm" then we could do

```
$ ./zxpre.exe hello.asm
message: db "Hello world!\n"
main:     mov %a,0x04
          mov %b,0x01
          mov %c,message
          mov %d,0x0d
          syscall
          mov %a,0x01
          syscall
```

Instructions begin on the 10th character of each line and labels appear in the first 10 characters. Labels appear in the first 9 characters of output (note that labels cannot be more than 8 characters) followed by a colon. There is a single space after each instruction mnemonic. Other operands are separated only by commas and not whitespace. There is no trailing whitespace following the operands.

I recommend to create a function called something like "preprocess_get_line" to which you can pass a file object and receive a line in return much like "fgets". The signature would be something like

```
int preprocess_get_line(char label[], char instruction[], FILE* fp)
{
  /*
   * Read next instruction from the file fp. Reads the file line
   * by line looking for the next non-blank line.
   *
   * The non-blank line will be preprocessed and if it has a label
   * that will be stored in label. Otherwise label will be set to
   * the empty string. The array for label must have space for 8
   * characters (plus a null byte).
   *
   * The remainder of the instruction will be stored in instruction.
   * All white-space and comments will have been removed except for
   * one space after the mnemonic and any spaces in string literals.
   * The operands will be separated only by commas. Instruction must
   * have space for 255 characters (plus a null byte).
   *
   * The return value indicates success/failure. If there was a problem
   * reading from the file or invalid assembly code was found then
   * the return value is 0. If a line of assembly was successfully read
   * then the return value is 1. If we reached the end of the file then
   * EOF is returned.
   */
}
```

Having created a function like this it would be easy to create the zxpre program. A function like this would also be useful in the other programs we need to make though. I would suggest to make this function and put it in a file called say zx256.c (with declaration in zx256.h). Then in zxpremain.c you can #include "zx256.h" and use the function. You should use it by looping over it until it returns EOF similar to the way we loop over fgets when reading a file line by line (probably preprocess_get_line would want to call fgets internally).

5.1 Reading from stdin

The command line usage of zxpre is that it can be run with one argument which is the name of the input file. The program will open the file whose name is given on the command line. If no file is given on the command line then zxpre will assume that it should read from stdin. In this case we can do the following to redirect a file to zxpre's stdin (note the "<" sign in the command):

```
$ ./zxpre.exe < hello.asm
message: db "Hello world!\n"
main:    mov %a,0x04
         mov %b,0x01
         mov %c,message
         mov %d,0x0d
         syscall
         mov %a,0x01
         syscall
```

In the command above zxpre was called with no arguments (argc=1) and hello.asm was redirected to the stdin of zxpre. When zxpre reads from stdin it will be reading from hello.asm. This behaviour (read from a named file or otherwise stdin) is common among command line programs.

6 zxenc

The zxenc.exe program is for manipulating assembly code and gives us a hex representation of the bytes that would appear when compiling assembly code. It has different modes depending in the command line arguments supplied. In the simplest case the user supplies the "-c" argument followed by a another argument which is a single assembly instruction. This instruction should be of the form output by zxpre with whitespace normalised and no comments. Because there needs to be a space after mnemonic we use single quotes ' around the instruction to show that it is a single command line argument. The quotes are not part of the string passed to argv (unless you run this in cmd.exe in which case you should use double quotes ").

```
$ ./zxenc.exe -c 'mov %a,0x01'
0x01 0xc0 0x01
$ ./zxenc.exe -c 'jmp 0x21'
0x20 0x21
$ ./zxenc.exe -c 'mov %b,%a'
```

```
0x01 0x08
$ ./zxenc.exe -c 'syscall'
0x30
```

There is a detailed discussion in the notes about how to encode an assembly instruction as bytes. I recommend that you create a function called "encode" that can do take an instruction string as input and stores its output in an array of 3 bytes e.g.

```
int encode(char instruction[], char ibytes[])
{
  /*
   *  Encode the instruction in the string instruction
   *  storing the resulting bytes in the array called
   *  ibytes. Returns the number of bytes used for the
   *  instruction (i.e. the number of bytes stored in
   *  ibytes). Returns 0 if there is an error.
   *
   *  ibytes should be big enough to store 256 bytes.
   */
}
```

You could put this function in zx256.c and use it as part of zxenc.exe. That way you can use this same code for zxenc.exe and also for zxcc.exe later.

6.1 Reading from a file

The other mode for zxenc.exe is that it reads from a file. In this case the "-c" argument is not supplied and the only command line argument is the filename or if no command line arguments are given then zxenc.exe should read from stdin. For example:

```
$ ./zxenc.exe hello.asm
0x48 0x65 0x6c 0x6c   ; message   db "Hello world!\n"
0x6f 0x20 0x77 0x6f
0x72 0x6c 0x64 0x21
0x0a
0x01 0xc0 0x04        ; main      mov %a,0x04
0x01 0xc1 0x01        ;           mov %b,0x01
0x01 0xc2 0x00        ;           mov %c,0x00
0x01 0xc3 0x0d        ;           mov %d,0x0d
0x30                  ;           syscall
0x01 0xc0 0x01        ;           mov %a,0x01
0x30                  ;           syscall
0x0d                  ;           main
```

Note carefully the output format here. We print each instruction with 4 bytes per line. There is a comment after the *first* byte line of the instruction which indicates the label and instruction that were used in the original assembly code. There are 2 spaces between the 4th byte and the semicolon. There are 10 spaces between the semicolon and the instruction mnemonic. The label is printed in the 8 middle spaces of these 10.

Assuming that you created the preprocess and encode functions as described above it is not too hard to put together a function that can do this. So I would suggest something like the following (in Pythonish pseudo-code):

```
def zxenc_dump(filename):
    file = open(filename)
    while True:
        ret = preprocess_get_line(label, instruction, file)
        if ret == EOF:
            break
        elif ret ==  0:
            return 1 # error
        nbytes = encode(instruction, ibytes)
        print_ibytes(ibytes, nbytes, label, instruction)
```

where the function "print_ibytes" produces output in the format shown above. This way we are reusing the same functions that we wrote earlier.

There is one point to be careful of though: we will need to read the file twice. On the first loop through the file we will work out the length of each instruction and keep a count of how many bytes there are *before* each instruction (so we know the memory address of each instruction). This way we can work out the memory addresses of each of the labels e.g. "message" and "main". On the second loop we reread the file and this time whenever we see a label we can replace it with the appropriate byte number. This enables us to generate the full machine code which we can output as shown above. The address of the "main" label is printed as a final byte at the end of the machine code. This is used by the ZX256 to know which instruction should be executed first. Getting the labels right is rather tricky so I would propose to leave that initially (until you've got everything else sorted).

7 zxcc.exe

The final program to make is called zxcc.exe. This is analogous to the gcc program that we have been using to compile our C programs. Unlike the programs above it does not produce a text display for output. Rather it writes its output to a binary file as bytes (not as hex literals). For example:

```
$ ./zxcc.exe hello.asm -o hello.exe
$ hd hello.exe
00000000  48 65 6c 6c 6f 20 77 6f  72 6c 64 21 0a 01 c0 04
|Hello world!....|
00000010  01 c1 01 01 c2 00 01 c3  0d 30 01 c0 01 30 0d
|.........0...0.|
0000001f
```

I'm showing the output from hd here because the contents of the file itself don't display easily in text. It's important to understand that the hello.exe file should not contain text. It should contain the raw bytes whose hex codes are shown above. Another way to view the file contents is from Python:

```
>>> with open('hello.exe', 'rb') as fin: # 'rb' for binary
...     bytes = fin.read()
...
>>> [hex(b) for b in bytes]
['0x48', '0x65', '0x6c', '0x6c', '0x6f', '0x20',
 '0x77', '0x6f', '0x72', '0x6c', '0x64', '0x21',
 '0xa', '0x1', '0xc0', '0x4', '0x1', '0xc1', '0x1',
 '0x1', '0xc2', '0x0', '0x1', '0xc3', '0xd', '0x30',
 '0x1', '0xc0', '0x1', '0x30', '0xd']
```

When no "-o" output file is given e.g. "./zxcc.exe hello.asm" then zxcc will save its output in a file called "a.exe" by default. If no input file is given e.g. "./zxcc.exe" or "./zxcc.exe -o foo.exe" then zxcc will read its input from stdin.

Assuming that the functions you made in the previous sections are good then making zxcc.exe should be easy. You just need to write the code to check the command line arguments provided and call the appropriate functions together. One thing to be careful of is that when writing to a binary file we need to open it with the binary mode set i.e. "wb" - otherwise Windows will screw up all the 0x0a bytes (not a problem on Linux/OSX).

8 Write an assembly program

The very final part of this assignment (worth 20%) is to write a short program in ZX256 assembly. However note that I will only mark this if you've completed all of the previous parts of the assignment. Please do not attempt this part until you've done everything else.

This program can do anything you like. You should explain what it does in your README.txt. Later I will put up a browser-based ZX256 emulator so that you can do this.

9 Final notes

I will put up more information about how to do different parts of this in C in the coming days along with tests and other stuff. Also I want to be clear that you should try to write good clear code. Separate the different parts of your program into sensible pieces that you can reuse. Don't copy/paste code everywhere. Please ensure that you check your program against the test code I provide. I guarantee that if you don't then you will miss some bugs in your program.