CS-UY 2214 — Homework 9

Jeff Epstein

Introduction

Unless otherwise specified, put your answers in a plain text file named `hw9.txt`. Number each answer. Submit your work on Gradescope.

The slides, available on Brightspace, cover many useful Intel opcodes. You may consult the online Intel assembly language reference to help you understand any unknown opcodes. You can also use the official Intel developer's manual.

To aid in debugging your assembly language programs, you can use `gdb` or similar. Please read the "Reversing tutorial," available on Brightspace.

Problems

1. Consider the following Intel assembly language fragment. Assume that the label `my_data` refers to a region of writable memory.

```
mov eax, my_data
mov ebx, 0x0123456
mov [eax], ebx
add eax, 2
mov bh, 0xff
add [eax], bh
add eax, 1
mov ecx, 0xabcdabcd
mov [eax], ecx
; program ends here
```

   Give the value of all known memory values when the program ends, starting at address `my_data`. Give your answer as a sequence of hexadecimal bytes. Recall that Intel is a little-endian architecture.

2. Consider the following Intel assembly language program.

```
    section .data

arr:
    dw 5, 50, 20, 30, 0
d:
    dd 0

    section .text
    global _start

_start:
    mov ecx, arr    ; store the address of the array
```

```
    xor edx, edx
    xor eax, eax

loop:
    mov dx, [ecx]  ; read word value
    add ecx, 2

    sub eax, edx
    jns L1         ; jump if sign bit is not set
    neg eax        ; 2s complement negate
L1:
    add [d], eax

    mov eax, edx
    cmp edx, 0
    jne loop

    ; end of program
```

(a) What does the program do? What is the final value of `[d]`? What is that value in terms of the initial value of the array `arr`?

(b) How does the program know when it has reached the end of the array?

(c) What is the size of the values in the array? How do you know?

(d) What is the size of the `d` variable (i.e. the memory region pointed to by the label of that name)? How do you know?

(e) What is the total size of the array in bytes?

(f) How does the program calculate absolute value?

(g) At the beginning of the program, we set `edx` to zero using the `xor edx, edx` instruction. Why is this instruction necessary?

3. Consider the following incomplete Intel assembly language program, `avg.asm`, which is provided in the `hw9.zip` file:

```
    section .data
numbers_size:
    dd 8
numbers:
    dd 10, 34, 55, 106, 44, 0, 45, 400

    section .text
    global _start
_start:
    ; Your code here
```

Complete the program so that it calculates the integer average of values of the unsigned dword array `numbers`. The length of the array is stored at address `numbers_size`. Your program should work correctly with an array containing any content, of any length. The suggested approach is to first sum the array, then divide. The final result should be returned as the exit status from the `exit` syscall.

Please examine the documentation for the `div` instruction. This instruction is slightly anomalous. Notice that `div` takes only one argument, the divisor, while the dividend is always implicitly the 64-bit

value in `edx:eax` (that is, `edx` forms the most significant 32 bits of the dividend, and `eax` is its least significant 32 bits). The divsor may not be an immediate value (i.e. `div 10` won't work), so you have to load the divisor into a register. Note, as well, that `div ebx` will divide the 64-bit value in `edx:eax` by `ebx`, and will store the quotient in `eax` and the remainder in `edx`. A consequence of the fact that `div` divides a 64-bit value, whose high dword is in `edx` and whose low dword is in `eax`, is that, if you want to divide only `eax` you must ensure that `edx` is zero before you issue the instruction. Furthermore, the quotient must fit in the 32-bit register `eax`: in case of quotient overflow, the CPU will signal a divide error, and your program will terminate.

Use the Linux `exit` syscall to terminate your program normally. Specify the array's average value as the exit status in `ebx`, as follows:

```
mov eax, 1    ; select exit syscall
mov ebx, ...  ; store the average in ebx
int 80h       ; invoke the syscall
```

Note that the exit value is 8-bit, so the highest possible expressible average is 255. You can assume that the average is less than 256. You can also assume that the sum of the array is less than 2^{32}.

You can assemble and link your program with commands similar to those discussed above:

```
user@ubuntu:~/intel$ nasm -f elf -gstabs avg.asm
user@ubuntu:~/intel$ ld -o avg -m elf_i386 avg.o
```

If your program works, you'll be able to verify the correct average by examining its exit status, as follows:

```
user@ubuntu:~/intel$ ./avg
user@ubuntu:~/intel$ echo $?
86
```

Submit your complete `avg.asm` file.

4. Write an Intel assembly language program in a file named `greet.asm` that will prompt the user to enter their name, and then respond with the greeting "Nice to meet you," followed by the name that the user entered.

   A sample execution of the program should look exactly like this, where the italicized text represents user input:

   ```
   Please enter your name: Spatula
   Nice to meet you, Spatula
   ```

   In other words, your program should be functionally identical to the following Python 3 program:

   ```
   name = input("Please enter your name: ")
   print("Nice to meet you, " + name)
   ```

   You'll have to use Linux syscalls. You will need syscall 4 (write) to print messages on the screen (via file descriptor 1), and syscall 3 (read) to accept data input from the keyboard (via file descriptor 0). Your program should then halt normally, using the syscall 1 (exit).

   The messages you display should be represented as strings in your program's `.data` section. Data you read in from the keyboard should be stored in a region in your program's `.bss` or `.data` section. To simplify the problem, you may assume that user input does not exceed 20 characters.

   To assemble and link your `greet.asm` file, use these commands:

   ```
   nasm -f elf greet.asm
   ld -o greet -m elf_i386 greet.o
   ```

If you don't get any errors, the result will be an executable file named `greet` that you can run like this:

```
./greet
```

Submit your completed `greet.asm` file.

Hint: you will need to use the read syscall to accept keyboard input from the user. Consult the syscall table in the slides and here. Note that the read syscall is number 3, and expects the following arguments: `ebx` will contain the file handle to read from (should be stdin); `ecx` will contain a pointer to a buffer where the user input will be stored; and `edx` will contain the maximum number of bytes to read. After the syscall completes, `eax` will contain the number of bytes that were actually read.

5. Download the file `hw9.zip`. The assembly language program `pr.asm` contained therein should print several unsigned integer values. Your task is to complete this program by writing the `print_int` function. The function should work like this:

   - When `print_int` is called, it expects the `eax` register to contain a 32-bit unsigned integer value.
   - It should convert this value into an ASCII string, stored in the provided `buffer`.
   - It should print this string to the console (via stdout) by using the `write` syscall (number 4).

   One possible algorithm to convert the binary value into an ASCII string is the same technique we use to convert a number to different bases: repeatedly divide the number by 10. The remainder (i.e. modulus) of each division will yield one decimal digit, while the quotient will be used in the next iteration. When the quotient is zero, we're done. You may want to consult the ASCII table. As described in a previous question, recall that the `div` opcode will produce both the quotient *and* the remainder.

   A tricky part of implementing this algorithm is that you need to keep track of how many decimal digits you've converted so far. The given technique will yield the *last* decimal digit first, so I recommend storing digits in the buffer starting from its end.

   Please recall also how to convert a binary value in the range 0–9 to the corresponding ASCII digit: it's sufficient to add the value of the ASCII character "`0`", which happens to be 48. If you have a value in the range 0–9 in register `al`, the following instruction will yield the appropriate ASCII character: `add al, 48`.

   A common mistake is to use dword memory operations on the buffer. Note that because we are storing a string in the buffer, we should treat it as an array of bytes. Therefore, we must be careful to use byte-sized memory operations on it.

   Another common mistake is to print out too many characters. The buffer is 16 characters, but the numeric strings you'll be printing out are shorter than that. You need to specify the correct length and starting address when you call the `write` syscall. If you see question marks (`?`) in your output, you are printing out bytes from a memory location where you didn't store a character.

   The commands for assembling and linking the program are given in the comments in the source code. Please test your program before submitting it. Submit your complete `pr.asm` file.

   If your program is correct, the output should be exactly as follows:

```
0
1
34
1024
9999
32777
20343343
2147483646
```

```
2
Everything is okay!
```

If the output does not match exactly, then something has gone wrong. In particular, if the last line is not `Everything is okay!`, then you have accidentally overwritten some values outside of the allocated buffer.

Modify only the body of the `print_int` function. Test your program before submitting. The code file contains instructions for assembling and linking. Submit your completed `pr.asm`.

Syscall reference

This table summarizes the use of those syscalls needed for this assignment.

eax	Name	Source	ebx	ecx	edx
1	sys_exit	kernel/exit.c	int	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t

Some notes on the use of these syscalls:

- `sys_exit` takes an "error value" in the range $0 - 255$ in `ebx`. By convention, the value zero represents success.
- `sys_read` and `sys_write` take a file descriptor in `ebx`.

  For our purposes, we will use the descriptor 0 (stdin) to read from the keyboard with `sys_read`; and the descriptor 1 (stdout) to write to the screen with `sys_write`.

  In other contexts, the descriptor may correspond to a file, a network socket, a timer, a pipe, or other operating system objects.
- `sys_read` and `sys_write` takes the address of a buffer in `ecx`, and the size of that buffer in `edx`. `sys_read` will read up to the specified number of bytes and will store them in the buffer. `sys_write` will write the specified number of bytes in the buffer to the destination (in our case, to the screen).
- When `sys_read` returns, it will leave a value in `eax` corresponding to the actual number of bytes read, which may be less than the size of the buffer.
- For all syscalls, after you've loaded the appropriate values into the registers, you need to invoke the Linux syscall interrupt with the instruction `int 80h`.

For example, to terminate the program and indicate success to the caller, use this invocation of `sys_exit`:

```
mov eax, 1      # select sys_exit
mov ebx, 0      # error value is success
int 80h         # do it!
```