LAB 6

Introduction to IA-32 Assembly Programming

This lab introduces basic instructions of IA32 assembly language on Linux. It will not contain a complete description of the IA32 architecture, but just enough to write simple programs from scratch. Consider the following C program:

```
main()
{
    int x = 10;
    int y = 20;
    int z = 30;
    printf("x y z = %d %d %d\n", x, y, z);
}
```

Save it as **num.c** and compile it with:

```
$ gcc –m32 –O1 –S num.c
```

The compiler will produce assembly output **num.s** rather than a binary executable program. The assembly code generated by the compiler may look like this:

```
.LC0:
      .string     "x y z = %d %d %d\n"
.globl main
main:
      pushl %ebp
      movl  %esp, %ebp
      andl  $-16, %esp
      subl  $32, %esp
      movl  $30, 16(%esp)
      movl  $20, 12(%esp)
      movl  $10, 8(%esp)
      movl  $.LC0, 4(%esp)
      movl  $1, (%esp)
      call  __printf_chk
      leave
      ret
```

Note that it has three distinct parts:

- **Directives** begin with a dot and indicate structural information useful to the assembler or the linker. You will need to know two directives: **`.globl`** and **`.string`**. For example, **`.globl main`** indicates that the label main is a global symbol that can be referenced by other code modules. .string indicates a string constant that the assembler should insert into the output code. You need not be concerned with the other directives shown.
- **Labels** end with a colon and indicate by their position the association between names and locations. For example, the label **`.LC0`**: indicates that the immediately following string should be called **`.LC0`**. The label **`main`**: indicates that the instruction **`pushl %ebp`** is the first instruction of the main function. By convention, labels beginning with a dot are temporary local labels generated by the compiler, while other symbols are user-visible functions and global variables.
- **Instructions** are everything else, typically indented to visually distinguish them from directives and labels.

Compile now the assembly code with:

```
$ gcc –m32 –o num num.s
```

Run it with:

```
$ ./num
```

EXERCISE 1: (10 points) Write an IA32 assembly program (**add.s**) which, given an integer *n*, will extract and print its individual bytes and their sum. For example, if *n* = 0x12345678, then the output should look like:

```
Byte 3 = 0x12
Byte 2 = 0x34
Byte 1 = 0x56
Byte 0 = 0x78
Sum = 0x114
```

EXERCISE 2: (20 points) Write a function **isprime** in IA32 assembly which, given an integer *n*, returns 1 if it is prime; 0 otherwise. Using this function write an IA32 assembly program (**prime.s**), which will print all prime numbers less than 100.

Hint: To determine whether a given number is prime or not, you can use division instruction. Before applying division instruction, 32-bit words in IA32 must be sign-extended into a 64-bit word with the following instruction:

cltd	R[**%edx**]:R[**%eax**] ← *SignExtend* (R[**%eax**])

The following division instructions compute the quotient and remainder:

idiv src	R[**%edx**] ← R[**%edx**]:R[**%eax**] mod src R[**%eax**] ← R[**%edx**]:R[**%eax**] div src	signed
div src	R[**%edx**] ← R[**%edx**]:R[**%eax**] mod src R[**%eax**] ← R[**%edx**]:R[**%eax**] div src	unsigned

So, the following sequence of instructions will divide *a* by *b*:

```
movl  a,%eax
cltd
idivl b
```

Here, *a* and *b* could be immediate, register, or memory. After **idiv** is executed, the quotient is found in **%eax**, and the remainder in **%edx**.

In your main function, use a simple loop and a call to **isprime** to determine and print all prime number less than 100. When completed, upload your **add.s** and **prime.s** as Lab 6. Make sure to include your and your partner's names in the header comments.

EXERCISE 3: (Extra Credit) Write a function **intToStr** in IA32 assembly which, given an integer *n* and a pointer *p*, converts *n* to an equivalent numeric string and places it at the memory location pointed by *p*. For example, if *n* = 1234, the function should place the numeric string "1234" at the memory location pointed by *p*. Write the main routine in IA32 Assembly to test the function. Note that *n* can be negative.