

Lab 2

Ai Nguyen

October 7, 2015

1 GCD in Math

This lab explores how we can create and use functions in the three different languages focused on in this course: SML, C, and ASM. The function we will be trying to implement is GCD, or the greatest common divisor of two positive integers.

This should give us the largest number that is exactly divides both of them such that there is no remainder. For example, the GCD of 14 and 12 is 2 and the GCD of 14 and 11 is 1.

The GCD is given by the following specification:

$$gcd : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$gcd(m, n) = \max\{d \in \mathbb{N} \mid m \bmod d = 0 \wedge n \bmod d = 0\}$$

One algorithm for calculating the gcd follows Euclid's method. If, for 2 natural numbers m and n, if $m > n$, then the gcd is defined by:

$$euclid(m, n) = \begin{cases} euclid(n, m \bmod n), & \text{if } n > 0 \\ m & \end{cases}$$

2 GCD in SML

The GCD can be implemented in SML like this:

SML

```
fun euclid (m, n) = if n > 0 then euclid (n, (m mod n))  
    else m;  
euclid(14,12);  
euclid(14,11);  
euclid(558,198);
```

```
> fun euclid (m, n) = if n > 0 then euclid (n, (m mod n))  
#     else m;  
val euclid = fn: int * int -> int  
> euclid(14,12);  
val it = 2: int  
> euclid(14,11);  
val it = 1: int  
> euclid(558,198);  
val it = 18: int  
> |
```

Since SML is naturally a functional and mathematics based language, it is quite easy to implement the GCD function in SML.

3 GCD in C

GCD can also be implemented in C. The following recursive function is a very simple and effective implementation of the Euclid method of getting the greatest common divisor

C source code written to file lab2.c

```
#include<stdio.h>
int euclid(int m, int n){
    if(n > 0) return euclid(n, m % n);
    else return m;
}
```

Let's try to write an equivalent function that is more like ASM. For example, do not use function composition. Do each operation on a separate line.

C source code appended to file lab2.c

```
int euclidasm(int m, int n){
    int r;
    if (n>0)
        r = euclid(n,m %n);
    else
        r = m;
    return r;
}
```

Now we can test the two functions and see if they both work.

C++ source code appended to file lab2.c

```
int main()
{
    printf("%d\n", euclid(14,12));
    printf("%d\n", euclid(14,11));
    printf("%d\n", euclid(558,198));
    printf("The euclid-asm version of the previous line: %d\n", euclidasm(558,198));
}
```

```
debian@debian:~/labs/lab2$ ccmp lab2
```

```
2
```

```
1
```

```
18
```

```
The euclid-asm version of the previous line: 18
```

When we run the program, we get the same results as SML gives us, (2, 1, 18), which is a good sign. Note that the C implementation is similar to the SML implementation in which we give a clear algorithm for calculating the GCD (if, else...).

4 GCD in ASM

Although these previous implementations are direct translations from math to SML to C, it is difficult to continue onto ASM because of the one-to-many relationship between C and ASM (and IA32 instruction set). For example, ASM does not have direct support for function composition.

So instead of doing something like this:

```
printf("%d\n", euclid(558,198));
```

In ASM we have to do something more like this:

```
int t = euclid(558/198);  
printf("%d\n",t);
```

However this is not possible since we can't assign a returned value into a memory location. Instead we always put it in the `eax` register for return values.

asm source code written to file lab2.s

```
.data  
fmt: .string "%d\n"  
.text  
.global _start
```

The following code calculates `euclid(558,198)` and prints it to the screen

asm source code appended to file lab2.s

```
|_start:  
|  push $198  
|  push $558  
|  call euclid  
|  add $8, %esp  
|  push %eax  
|  push $fmt  
|  call printf  
|  add $8, %esp
```

Since we have finished printing the output of `euclid`, we can now end the program

asm source code appended to file lab2.s

```
|  mov $1, %eax  
|  mov $0, %ebx  
|  int $0x80
```

Definition of Euclid function. Problems to solve:

- Create stack frame for function call (use the ebp (“base pointer”), in conjunction with esp (“stack pointer”))
- Get parameters from frame (by indirect accessing mode using offsets from ebp)
- Calculate modulus (remainder after integer division using idiv command)
- Implement if/else statement (using cmp and jump instructions)
- Call euclid recursively
- Assign a return value (use mov instruction to move into eax)

asm source code appended to file lab2.s

```
euclid:  
| push %ebp  
| mov %esp,%ebp  
| mov 8(%ebp),%eax  
| mov $0, %edx  
| mov 12(%ebp),%ebx  
| cmp $0,%ebx
```


After setting up the registers for integer division, we can check if we need to call euclid again if n is not greater than 0.

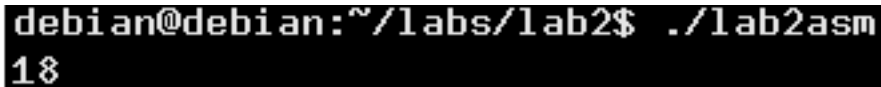
asm source code appended to file lab2.s

```
| jng _else  
  
| idiv %ebx  
| push %edx  
| push %ebx  
| call euclid  
| add $8, %esp  
  
| jmp endif  
| _else:
```

In each call of euclid, we set m into eax. So if we run the function and we find that immediately $n = 0$, we can simply return m by calling ret.

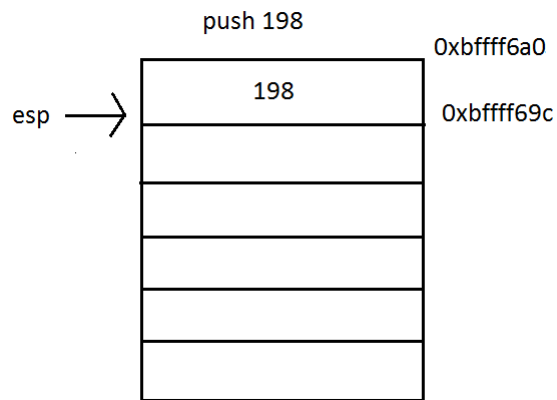
asm source code appended to file lab2.s

```
| endif:  
| mov %ebp,%esp #move stack pointer back onto the main stack  
| pop %ebp # remove the old %ebp (which is on top)  
| ret
```

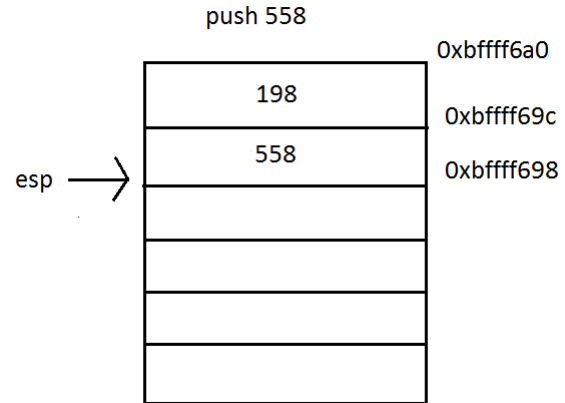
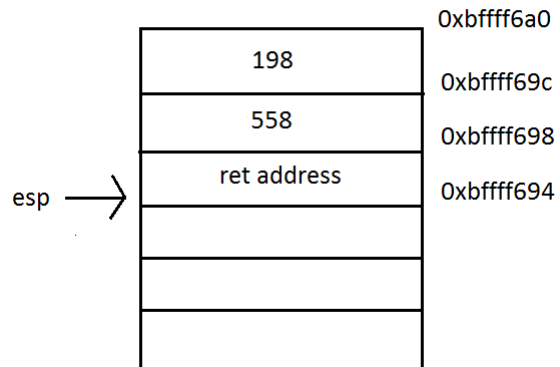


```
debian@debian:~/labs/lab2$ ./lab2asm  
18
```

5 Stack Frame



call euclid
(it will push ret address to stack and change instruction pointer to where the function start)



push %edx

