Christian Suleiman

Take Home Test 2: Recursive Functions

CS342 Section EF

Fall 2019

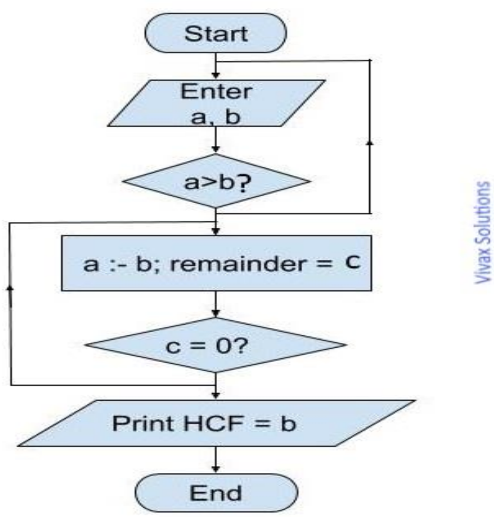
Objective

The objective of this take-home test is to study the process of memory allocation when a processor handles recursive calls. We will look at how the stack is manipulated and accessed during these calls and when it must return to a previous address once the call is complete.

We will be looking at three different environments in which this process takes place. These environments are

- Using the MIPS Instruction Set on MARS simulator
- x86 Intel Processor on Microsoft Visual Studio
- 64-bit Intel Processor on Linux-based GDB

To study recursive functions, we require a recursive function to test on. For this assignment, we will be implementing a recursive algorithm to find the Greatest Common Divisor (GCD) of two positive integers. This algorithm will replicate Euclid's algorithm, the pseudo-code of which can be seen here:



MIPS on MARS

Here is my MIPS Assembly code for implementing Euclid's algorithm.

```
# Christian Suleiman
# GCD Recursive

li $a0, 44 #load values into reg
li $a1, 8
li $a2, 0 # hold the result/temp

sub $sp,$sp,16 # creating space

sw $ra,0($sp)  #storing values in
sw $a0,4($sp)
sw $a1,8($sp)
sw $a2,12($sp)
```

This first set of instructions is to be used for allocating space to hold our local variables(A, B, result) as well as the return address for when we wish to come back to the main() function (four 4-btyes of data → offset of 16 from stack pointer).

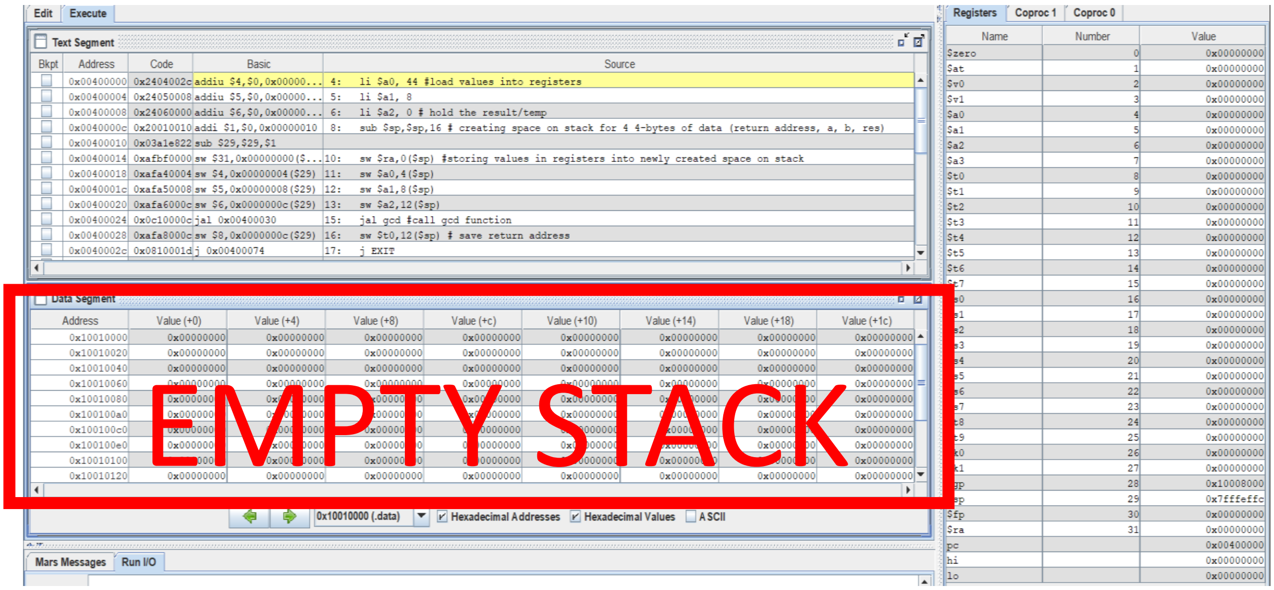
```
gcd:
  # a0 and a1 are the two integer para
  sub $sp,$sp,12
  sw $ra,0($sp)
  sw $a0,4($sp)
  sw $a1,8($sp)
  move $t0, $a0
  move $t1, $a1

#loop:
  beq $t1, $0, done #if second arg(a0 
  div $t0, $t1 #divide our operands . 
  move $t0, $t1 # a = b in euclid's al
  mfhi $t1 # b = remainder
  move $a1,$t1
  move $a0,$t0
  jal gcd #return to start of loop
done: #essentially a loop of returning
  lw $ra,0($sp) # loading top of stack
  addi $sp,$sp,12 # deallocating space
  jr $ra
EXIT:
```

In this next set of instructions, we define our GCD function as well as handle memory allocation/deallocation for each recursive call.

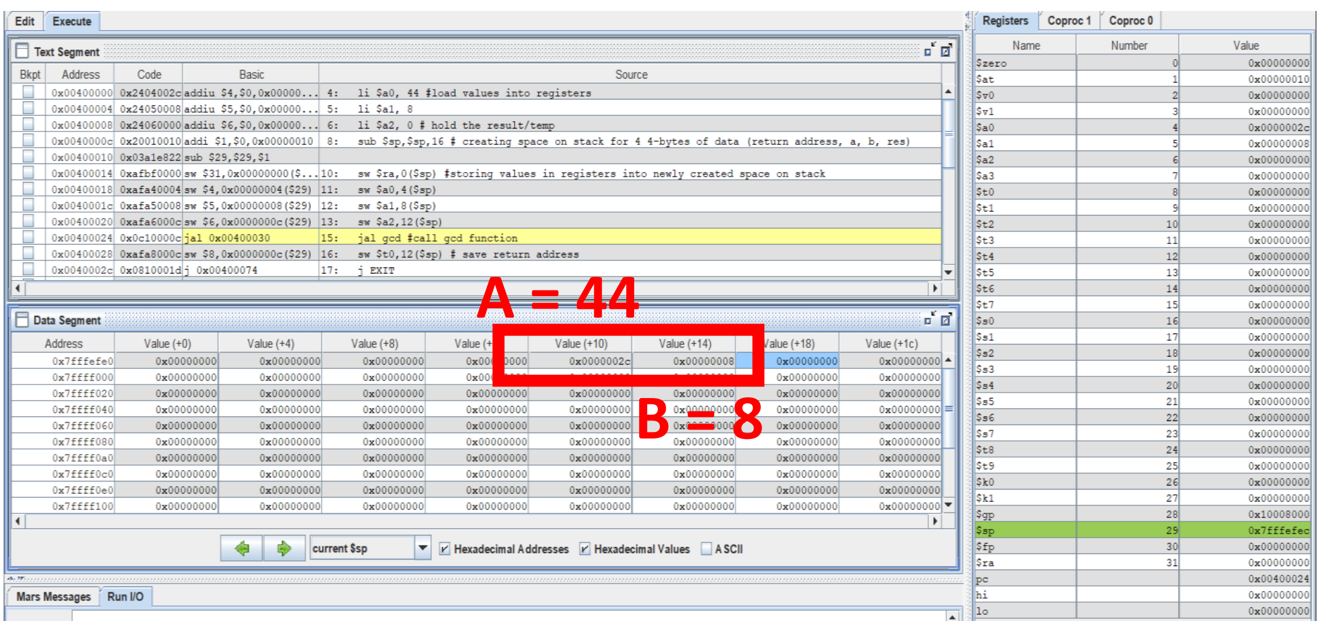
At the beginning of the GCD function, we allocate additional space for the current iteration’s set of local variables. So each time we call GCD, we call will create a new stack.

We then begin the loop that exits when B = 0, the answer being A. If not then we perform another GCD call with B, A%B as the arguments. Once we find the solution, we loop through done and deallocate an offset of 12 for each stack frame we generate.

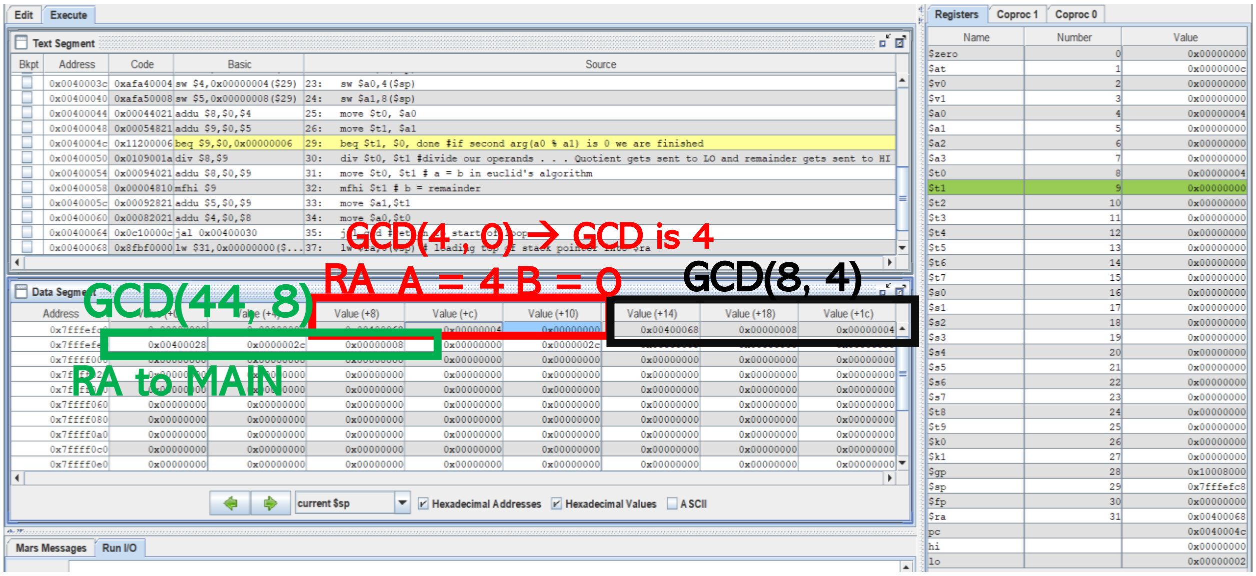
The following figures will show how the stack is modified when performing recursive calls.



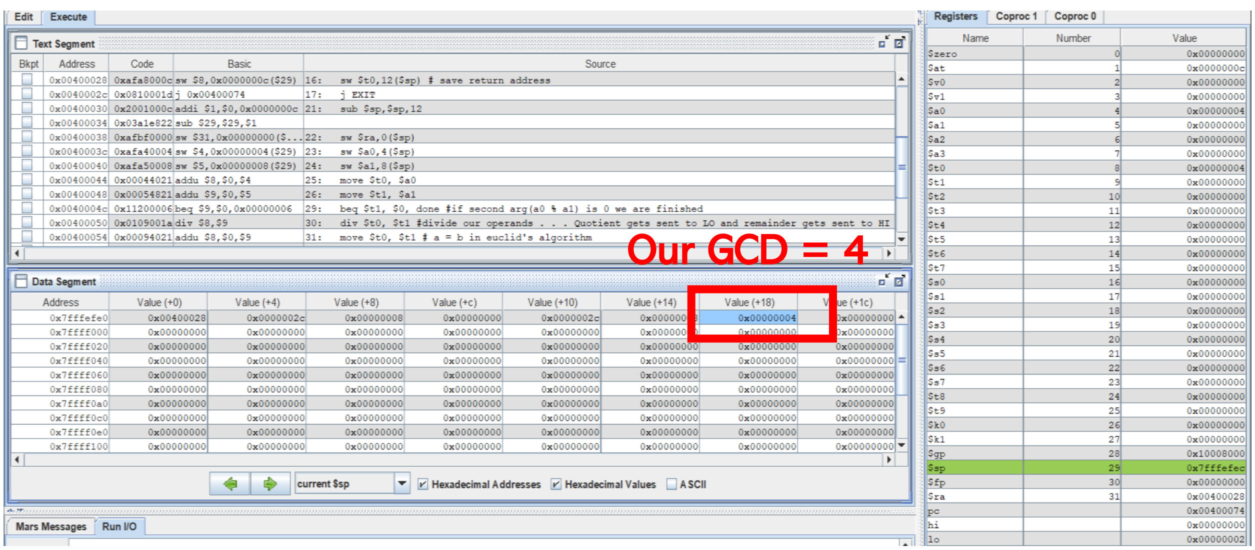
This is the stack before we begin executing any instructions, viewed via the MARS debugging Window.



This figure shows the stack allocating space to our local variables A and B on the first call of GCD(44, 8). We also allocate space to hold our result but it will remain empty until we finish the recursive procedures. Also important to note that integers are stored in BIG Endian Format, that is MSB is on the left.



Memory after we have made all of our recursive calls. Each stack contains the return address to the previous stack with the first stack returning to main(). The final stack GCD(4,0) begins the deallocation and return process.

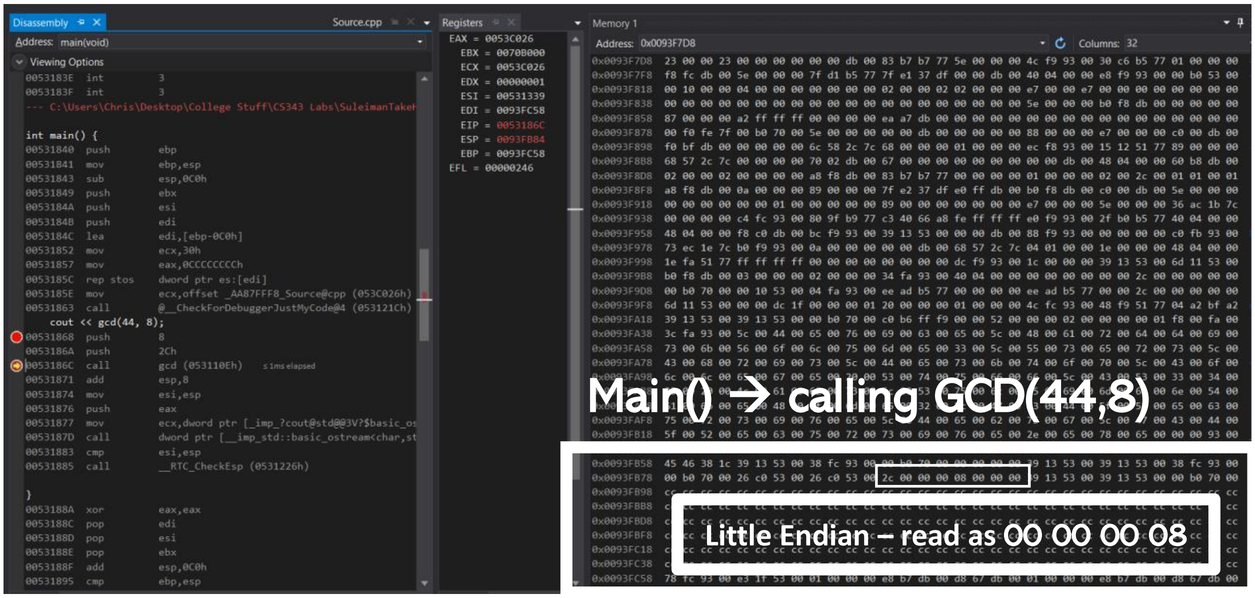


Here we see that our after deallocation we have our result stored in the initial space in memory we allocated when we first ran the program.

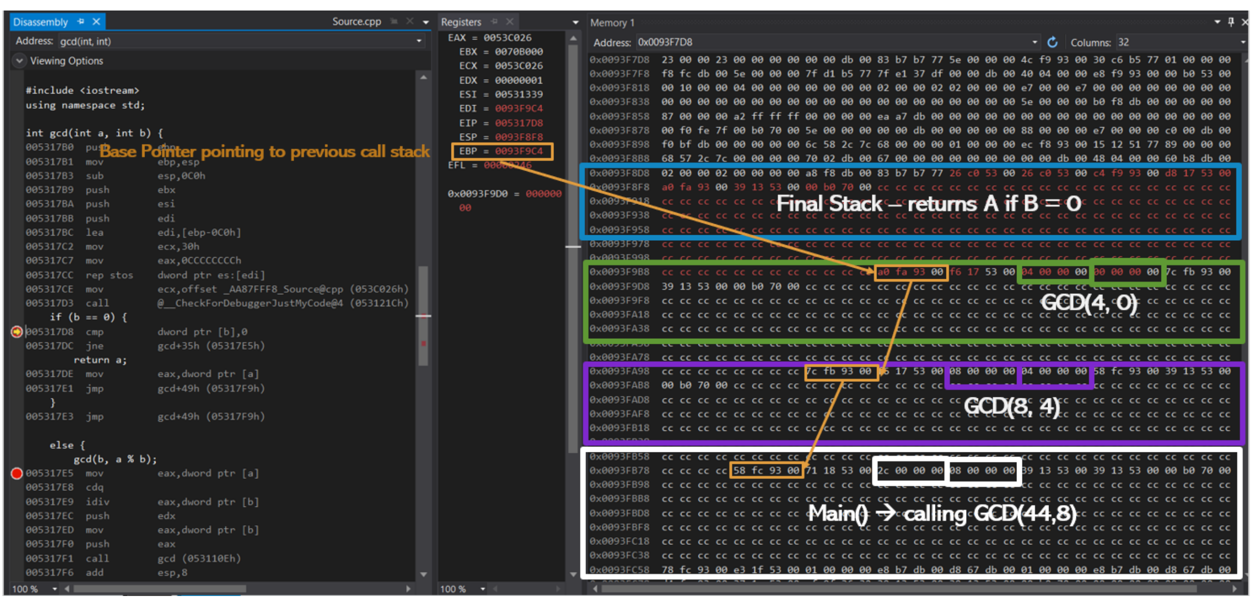
X86 Intel

Next we will look at Intel's x86 Processor's handling of recursive calls. Here is my C++ code written for this program. It very simply follows Euclid's algorithm pseudo code given to us by Professor Gertner.

```cpp
// Christian Suleiman
// Recursive GCD

#include <iostream>
using namespace std;

int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }

    else {
        gcd(b, a % b);
    }
}

int main() {
    cout << gcd(25, 10);

}
```



Here we should note that Intel follow LITTLE endian format for storing integers so 8 is stored as 08 00 00 00. This is the main body calling GCD.



Here we see the stacks being constructed for each call, similarly to the MIPS implementation. When we are in the final stack, EBP is pointing to the address of the previous stack so that we can return to it.

64-Bit Intel Processor on Linux Kali

For this assignment I simulated the Kali Linux OS through Oracle's VirtualBox software since I am on a Windows computer. Here is my C code written in the Linux terminal using Vim, almost identical to the previous implementation.

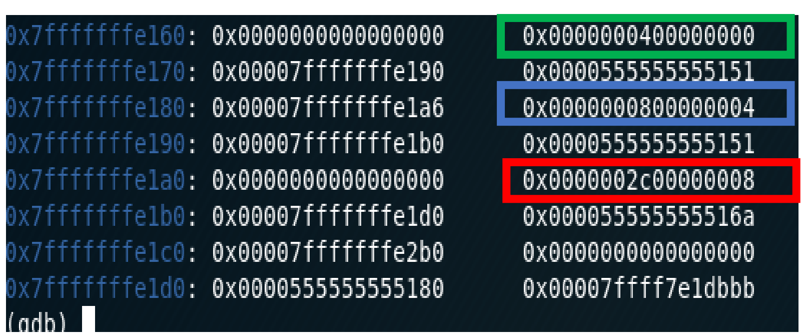
```c
int gcd(int a, int b) {
        if (b == 0) {
                return a;
        }
        else {
                gcd(b, a % b);
        }
}

int main() {
        int res = gcd(44, 8);
        return 0;
}
```

```
(gdb) nexti
0x0000555555555165      11              int res = gcd(44, 8);
   0x000055555555515b <main+8>: be 08 00 00 00  mov    $0x8,%esi
   0x0000555555555160 <main+13>:        bf 2c 00 00 00  mov    $0x2c,%edi
=> 0x0000555555555165 <main+18>:        e8 bb ff ff ff  callq  0x555555555125 <gcd>
   0x000055555555516a <main+23>:        89 45 fc        mov    %eax,-0x4(%rbp)
(gdb) print res
$1 = 0
(gdb) print &res
$2 = (int *) 0x7fffffffe1cc
(gdb) x/32xw $rsp
0x7fffffffe1c0: 0xffffe2b0      0x00007fff      0x00000000      0x00000000
0x7fffffffe1d0: 0x55555180      0x00005555      0xf7e1dbbb      0x00007fff
0x7fffffffe1e0: 0x00000000      0x00000000      0xffffe2b8      0x00007fff
0x7fffffffe1f0: 0x00040000      0x00000001      0x55555153      0x00005555
0x7fffffffe200: 0x00000000      0x00000000      0xa836139b      0x31384163
0x7fffffffe210: 0x55555040      0x00005555      0xffffe2b0      0x00007fff
0x7fffffffe220: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe230: 0xc8f6139b      0x646d1436      0xbdd0139b      0x646d040a
(gdb)
```

In our code, the return value of GCD is stored in int res. Here we can check the address of res and see that it does not contain a value at the beginning of the program.

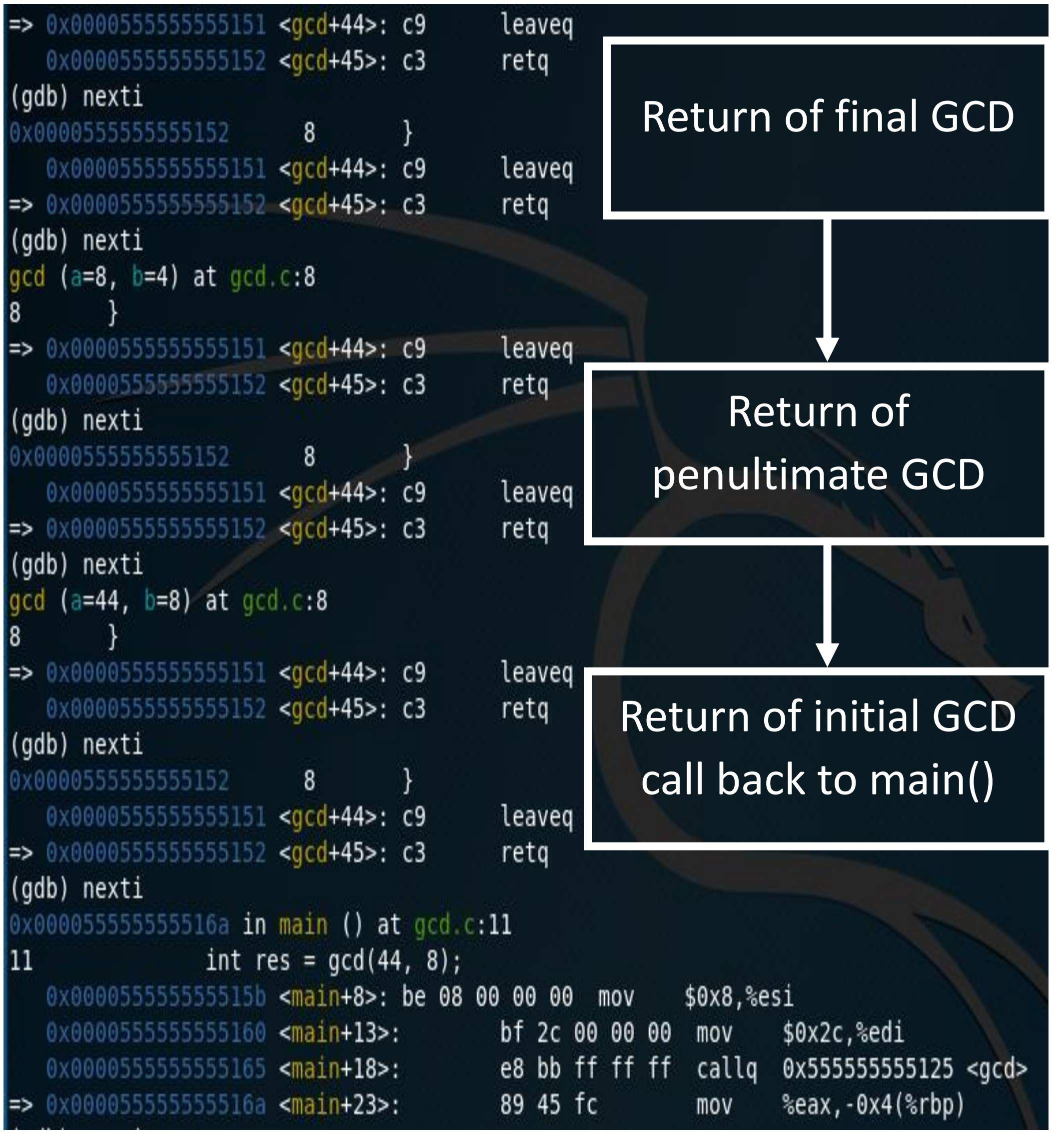
```
Breakpoint 2, gcd (a=44, b=8) at gcd.c:2                GCD(44,8)
2               if (b == 0) {
=> 0x0000555555555133 <gcd+14>: 83 7d f8 00     cmpl   $0x0,-0x8(%rbp)
   0x0000555555555137 <gcd+18>: 75 05   jne    0x55555555513e <gcd+25>
(gdb) next 5

Breakpoint 2, gcd (a=8, b=4) at gcd.c:2                 GCD(8, 4)
2               if (b == 0) {
=> 0x0000555555555133 <gcd+14>: 83 7d f8 00     cmpl   $0x0,-0x8(%rbp)
   0x0000555555555137 <gcd+18>: 75 05   jne    0x55555555513e <gcd+25>
(gdb) next 5

Breakpoint 2, gcd (a=4, b=0) at gcd.c:2                 GCD(8, 4)
2               if (b == 0) {
=> 0x0000555555555133 <gcd+14>: 83 7d f8 00     cmpl   $0x0,-0x8(%rbp)
   0x0000555555555137 <gcd+18>: 75 05   jne    0x55555555513e <gcd+25>
(gdb) nexti
0x0000555555555137      2               if (b == 0) {
   0x0000555555555133 <gcd+14>: 83 7d f8 00     cmpl   $0x0,-0x8(%rbp)
=> 0x0000555555555137 <gcd+18>: 75 05   jne    0x55555555513e <gcd+25>
(gdb) x/32xw $rsp
0x7fffffffe160: 0x00000000      0x00000000      0x00000000      0x00000004
0x7fffffffe170: 0xffffe190      0x00007fff      0x55555151      0x00005555
0x7fffffffe180: 0xffffe1a6      0x00007fff      0x00000004      0x00000008
0x7fffffffe190: 0xffffe1b0      0x00007fff      0x55555151      0x00005555
0x7fffffffe1a0: 0x00000000      0x00000000      0x00000008      0x0000002c
0x7fffffffe1b0: 0xffffe1d0      0x00007fff      0x5555516a      0x00005555
0x7fffffffe1c0: 0xffffe2b0      0x00007fff      0x00000000      0x00000000
0x7fffffffe1d0: 0x55555180      0x00005555      0xf7e1dbbb      0x00007fff
(gdb) print {a,b}
$3 = {4, 0}
(gdb) print {&a,&b}
$4 = {0x7fffffffe16c, 0x7fffffffe168}
```

Here we can see the construction of each stack that occurs after each function call. We see the arguments for each call are stored in their respective stacks.

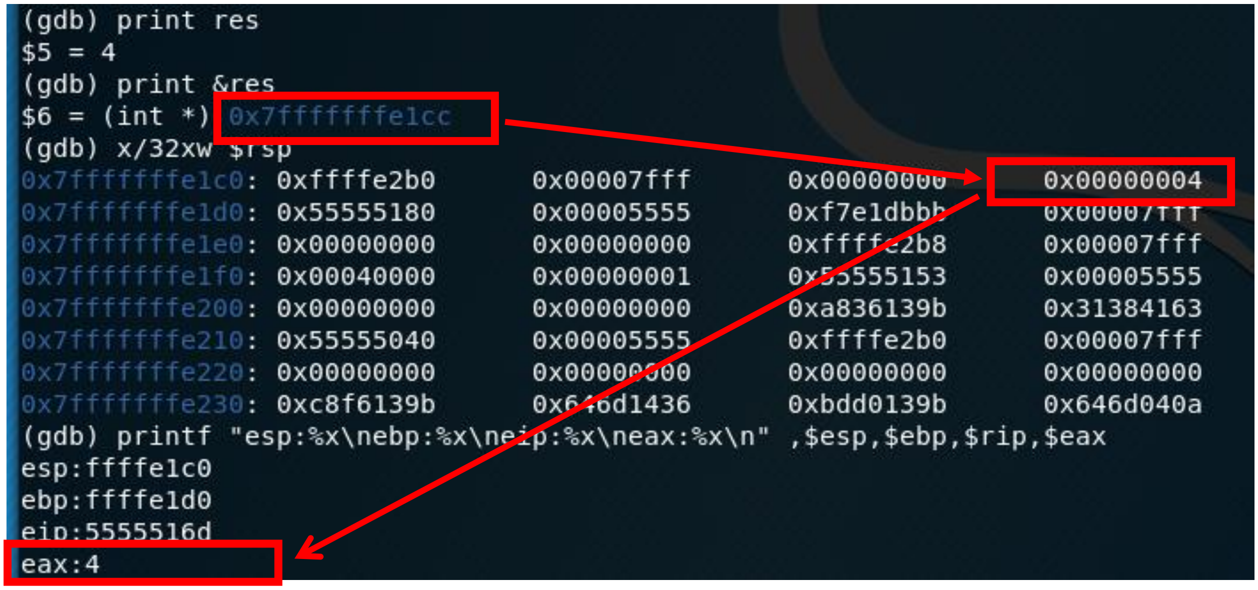


```
0x7fffffffe160: 0x0000000000000000      0x0000000400000000
0x7fffffffe170: 0x00007fffffffe190      0x0000555555555151
0x7fffffffe180: 0x00007fffffffe1a6      0x0000000800000004
0x7fffffffe190: 0x00007fffffffe1b0      0x0000555555555151
0x7fffffffe1a0: 0x0000000000000000      0x0000002c00000008
0x7fffffffe1b0: 0x00007fffffffe1d0      0x000055555555516a
0x7fffffffe1c0: 0x00007fffffffe2b0      0x0000000000000000
0x7fffffffe1d0: 0x0000555555555180      0x00007ffff7e1dbbb
(gdb)
```

We should also note that the arguments are being stored in 64-bits here.
Although integers seem to be Big Endian, they are actually stored as Little Endian.



Here we see the returning to previous stacks until we return to main()



When we finally have our result, it is stored in register EAX. Once we return to main(), we store the value of EAX into our 'res' variable.

Conclusion

We have studied how recursive functions are handled by three different processors. Although they are relatively quick, the main disadvantage of recursive calls is the large amount of memory used up in creating stacks. This can lead to stack overflow. We have also observed the key differences and similarities between these three environments.