Last Name: RAMOS First Name: ANTHONY

Computer Science
C.Sc. 342
Take Home TEST No.2

CSc or CPE

Submit by 11:00 PM, April 24, 2021

"I will neither give nor receive unauthorized assistance on this TEST. I will use only one computing device to perform this TEST. I will not use cell while performing this test."

- Anthony Ramos

Start Time and date: 8:00PM @ 4/22/21
End TIME and date: 8:00PM @ 4/23/21

Contents

Objective

The objective of this test is to explore and demonstrate understanding of recursive calls and stack frames in three sets of architecture: MIPS instruction set architecture, Intel x86 ISA via MS Visual Studio 32-Bit compiler and debugger, and Intel X86 64-bit running Linux platform (64-bit GCC and GDB).

Introduction

A Stack Frame is a frame of data (i.e., a region of memory) that gets created (i.e, allocated) within the stack. Each time a function is called, a new stack frame is created. Each *new* frame initialized by pushing the old base pointer onto the stack. A new base pointer is initialized, and space is allocated (via a subtraction) from stack pointer for necessary data. Next, any arguments are pushed onto the stack (using negative offsets w.r.t base pointer for local variables). Lastly, the *return address* (i.e., the address after the function call) is pushed to the top of the stack prior moving on to the next function call. For each function call, this process repeats for a new Stack Frame. The behavior of Stack Frames may differ depending on the program. One of the best ways to understand Stack Frames is through recursion.

I. Factorial

MIPS on MARS Simulator

Consider the following assembly program that computes the factorial of some integer *n*.

```
.data
        n: .word 5
        N_fact: .word 0
.text
main:

        lw $a0, n
        jal factorial
        sw $v0, N_fact

        li $v0, 10
        syscall

factorial: # procedure to calculate factorial(n)
        addi $sp, $sp, -8 # adjust stack pointer for 2 items
        sw $s0, 4($sp) # store the argument
        sw $ra, 0($sp) # store the return address

        # base case
        li $v0, 1
        beq $a0, 0, endcall # if n = 1, go to endcall procedure

        move $s0, $a0
        sub $a0, $a0, 1 # set argument for n-1
        jal factorial # call factorial() with n-1 as the argument

        mul $v0, $s0, $v0 # compute n * factorial(n-1)

        endcall: #return from jal
                lw $ra, 0($sp) # restore the return address
                lw $s0, 4($sp) # restore the argument n
                addi $sp, $sp, 8 # adjust stack pointer to pop twice

                jr $ra # jump to return address
```

Figure 1: Assembly program to compute factorial

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	7: lw $a0, n
	0x00400004	0x8c240000	lw $4,0x00000000($1)	
	0x00400008	0x0c100007	jal 0x0040001c	8: jal factorial
	0x0040000c	0x3c011001	lui $1,0x00001001	9: sw $v0, N_fact
	0x00400010	0xac220004	sw $2,0x00000004($1)	
	0x00400014	0x2402000a	addiu $2,$0,0x0000000a	11: li $v0, 10
	0x00400018	0x0000000c	syscall	12: syscall
	0x0040001c	0x23bdfff8	addi $29,$29,0xffff...	15: addi $sp, $sp, -8 # adjust stack pointer for 2 items
	0x00400020	0xafb00004	sw $16,0x00000004($29)	16: sw $s0, 4($sp) # store the argument
	0x00400024	0xafbf0000	sw $31,0x00000000($29)	17: sw $ra, 0($sp) # store the return address
	0x00400028	0x24020001	addiu $2,$0,0x00000001	20: li $v0, 1
	0x0040002c	0x20010000	addi $1,$0,0x00000000	21: beq $a0, 0, endcall # if n = 1, go to endcall procedure
	0x00400030	0x10240005	beq $1,$4,0x00000005	
	0x00400034	0x00048021	addu $16,$0,$4	23: move $s0, $a0
	0x00400038	0x20010001	addi $1,$0,0x00000001	24: sub $a0, $a0, 1 # set argument for n-1
	0x0040003c	0x00812022	sub $4,$4,$1	
	0x00400040	0x0c100007	jal 0x0040001c	25: jal factorial # call factorial() with n-1 as the argument
	0x00400044	0x72021002	mul $2,$16,$2	27: mul $v0, $s0, $v0 # compute n * factorial(n-1)
	0x00400048	0x8fbf0000	lw $31,0x00000000($29)	30: lw $ra, 0($sp) # restore the return address
	0x0040004c	0x8fb00004	lw $16,0x00000004($29)	31: lw $s0, 4($sp) # restore the argument n
	0x00400050	0x23bd0008	addi $29,$29,0x0000...	32: addi $sp, $sp, 8 # adjust stack pointer to pop twice
	0x00400054	0x03e00008	jr $31	34: jr $ra # jump to return address

Figure 2: Text segment windows showing all program instructions

Initially, the first instructions located at addresses 0x00400000 and 0x00400004 will simply load the argument *n* to register `$a0` which will contain 0x05 in the register window. The memory location of the argument is stored at address 0x10010000 in the Data Segment window.

$v1	3	0x00000000
$a0	4	0x00000005
$a1	5	0x00000000

Figure 3a: Register window showing contents of register `$a0` after executing first two instructions

Data Segment

Address	Value (+0)
0x10010000	0x00000005

Figure 3b: Memory location and value of argument *n*

The next instruction at address 0x00400008 will perform a jump and link (`jal`). Upon executing this instruction, the value stored in register `$ra` is updated to now contain the *return address* (i.e. the instruction address after the `jal` instruction of line 8). We will need this return address to return back *this* location in the program after executing all the necessary function calls.

$fp	30	0x00000000
$ra	31	0x0040000c
pc		0x0040001c

Figure 3c: Register window showing return address stored in register `$ra`

First Call

The next three instructions to be executed are at addresses 0x0040001C to 0x00400028. Upon executing these instruction, memory (8 bytes) is allocated to stack which updates the value stored at the stack pointer ($sp).

$gp	28	0x10008000
$sp	29	0x7fffeff4
$fp	30	0x00000000

Figure 4a: Stack pointer value (first call)

The value in register `$s0` (i.e. 0x00) is stored at address 0x7FFFEFF8 (i.e. 0x7FFFEFE0 + 0x18 offset). Likewise, the value in register $ra is stored at address 0x7FFFEFF4 (i.e. 0x7FFFEFE0 + 0x14 offset) in the Data Segment.

Value (+14)	Value (+18)
0x0040000c	0x00000000

Figure 4b: Memory locations and values of `$ra` and `$s0`

Next, the values of registers `$s0` and `$ra` are pushed onto the stack at address 0x7FFFEFF8 and 0x7FFFEFF4 respectively. Lastly, we load 1 onto register $v0 to be the default return value.

$at	1	0x10010000
$v0	2	0x00000001
$v1	3	0x00000000

Figure 4c: Return value stored in register `$v0`

The next instruction at address 0x0040002C will check (via the `beq` instruction) if the current argument value *n* (i.e. 0x05) stored at register `$a0` is equal to 0. Since it is not, we skip over this instruction and go on to execute the next series of instructions at addresses 0x00400034 and 0x00400038. These instructions will first copy the value in register `$a0` (i.e. 0x05) onto register `$s0`. The value at register `$a0` is then decremented by 1 and now contains 0x04. This will be the next argument value in the next function call.

$t7	15	0x00000000
$s0	16	0x00000005
$s1	17	0x00000000

$v1	3	0x00000000
$a0	4	0x00000004
$a1	5	0x00000000

Figure 4d: Updated contents of `$s0` and `$a0`

The next instruction to be executed (and the last in this first call) is at address 0x00400040 which will update the value in the `$ra` register to 0x00400044 (i.e. the instruction address after the jal instruction).

$fp	30	0x00000000
$ra	31	0x00400044
pc		0x0040001c

Figure 4e: Register window showing return address stored in register `$ra`

Second Call

We return back to the start of the factorial procedure. The first three instructions in this procedure to be executed are at addresses 0x0040001C to 0x00400028 which create a new stack frame. Again, executing these instructions allocates 8 bytes of memory to the stack which updates the value stored at the stack pointer (`$sp`).

$gp	28	0x10008000
$sp	29	0x7fffefec
$fp	30	0x00000000

Figure 5a: Stack pointer value (second call)

The value at register `$s0` (i.e. 0x05) is stored at address 0x7FFFEFF0 (i.e. 0x7FFFEFE0 + 0x10 offset). Likewise, the value at register $ra is stored at address 0x7FFFEFEC (i.e. 0x7FFFEFE0 + 0x0C offset) in the Data Segment.

Value (+c)	Value (+10)
0x00400044	0x00000005

Figure 5b: Memory locations and values of `$ra` and `$s0`

The next instruction at address 0x0040002C will check (via the `beq` instruction) if the current argument value *n* (i.e. 0x04) stored at register `$a0` is equal to 0. Since it is not, we skip over this instruction and go on to execute the next series of instructions at addresses 0x00400034 and 0x00400038. These instructions will first copy the value in register `$a0` (i.e. 0x04) onto register `$s0`. The value at register `$a0` is then decremented by 1 and now contains 0x03. This will be the next argument value in the next function call.

$t7	15	0x00000000
$s0	16	0x00000004
$s1	17	0x00000000

$v1	3	0x00000000
$a0	4	0x00000003
$a1	5	0x00000000

Figure 5c: Updated contents of `$s0` and `$a0`

The next instruction to be executed (and the last in this second call) is at address 0x00400040 which will update the value in the `$ra` register to 0x00400044 (i.e. the instruction address after the `jal` instruction).

$fp	30	0x00000000
$ra	31	0x00400044
pc		0x0040001c

Figure 5d: Register window showing return address stored in register `$ra`

Third Call

We return back to the start of the factorial procedure. The first three instructions in this procedure to be executed are at addresses 0x0040001C to 0x00400028 which create a new stack frame. Again, executing these instructions allocates 8 bytes of memory to the stack which updates the value stored at the stack pointer (`$sp`).

$gp	28	0x10008000
$sp	29	0x7fffefe4
$fp	30	0x00000000

Figure 6a: Stack pointer value (third call)

The value at register `$s0` (i.e. 0x04) is stored at address 0x7FFFEFE8. Likewise, the value at register $ra is stored at address 0x7FFFEFE4 in the Data Segment..

Value (+4)	Value (+8)
0x00400044	0x00000004

Figure 6b: Memory locations and values of `$ra` and `$s0`

The next instruction at address 0x0040002C will check (via the `beq` instruction) if the current argument value *n* (i.e. 0x03) stored at register `$a0` is equal to 0. Since it is not, we skip over this instruction and go on to execute the next series of instructions at addresses 0x00400034 and 0x00400038. These instructions will first copy the value in register `$a0` (i.e. 0x03) onto register `$s0`. The value at register `$a0` is then decremented by 1 and now contains 0x02. This will be the next argument value in the next function call.

$t7	15	0x00000000
$s0	16	0x00000003
$s1	17	0x00000000

$v1	3	0x00000000
$a0	4	0x00000002
$a1	5	0x00000000

Figure 6c: Updated contents of `$s0` and `$a0`

The next instruction to be executed (and the last in this third call) is at address 0x00400040 which will update the value in the `$ra` register to 0x00400044 (i.e. the instruction address after the `jal` instruction).

$fp	30	0x00000000
$ra	31	0x00400044
pc		0x0040001c

Figure 6d: Register window showing return address stored in register `$ra`

Fourth Call

We return back to the start of the factorial procedure. The first three instructions in this procedure to be executed are at addresses 0x0040001C to 0x00400028 which create a new stack frame. Again, executing these instructions allocates 8 bytes of memory to the stack which updates the value stored at the stack pointer (`$sp`).

$gp	28	0x10008000
$sp	29	0x7fffefdc
$fp	30	0x00000000

Figure 7a: Stack pointer value (fourth call)

The value at register `$s0` (i.e. 0x03) is stored at address 0x7FFFEFE0. Likewise, the value at register `$ra` is stored at address 0x7FFFEFDC (i.e. 0x7FFFEFC0 + 0x1C offset) in the Data Segment.

Address	Value (+0)
0x7fffefc0	0x00000000
0x7fffefe0	0x00000003

Value (+1c)
0x00400044

Figure 7b: Memory locations and values of `$ra` and `$s0`

The next instruction at address 0x0040002C will check (via the `beq` instruction) if the current argument value *n* (i.e. 0x02) stored at register `$a0` is equal to 0. Since it is not, we skip over this instruction and go on to execute the next series of instructions at addresses 0x00400034 and 0x00400038. These instructions will first copy the value in register `$a0` (i.e. 0x02) onto register `$s0`. The value at register `$a0` is then decremented by 1 and now contains 0x01. This will be the next argument value in the next function call.

$t7	15	0x00000000
$s0	16	0x00000002
$s1	17	0x00000000

$v1	3	0x00000000
$a0	4	0x00000001
$a1	5	0x00000000

Figure 7c: Updated contents of `$s0` and `$a0`

The next instruction to be executed (and the last in this fourth call) is at address 0x00400040 which will update the value in the `$ra` register to 0x00400044 (i.e. the instruction address after the `jal` instruction).

$fp	30	0x00000000
$ra	31	0x00400044
pc		0x0040001c

Figure 7d: Register window showing return address stored in register `$ra`

Fifth Call

We return back to the start of the factorial procedure. The first three instructions in this procedure to be executed are at addresses 0x0040001C to 0x00400028 which create a new stack frame. Again, executing these instructions allocates 8 bytes of memory to the stack which updates the value stored at the stack pointer (`$sp`).

$gp	28	0x10008000
$sp	29	0x7fffefd4
$fp	30	0x00000000

Figure 8a: Stack pointer value (fifth call)

The value at register `$s0` (i.e. 0x02) is stored at address 0x7FFFEFD8 (i.e. 0x7FFFEFC0 + 0x18 offset). Likewise, the value at register $ra is stored at address 0x7FFFEFD4 (i.e. 0x7FFFEFC0 + 0x14 offset) in the Data Segment.

Value (+14)	Value (+18)
0x00400044	0x00000002

Figure 8b: Memory locations and values of `$ra` and `$s0`

The next instruction at address 0x0040002C will check (via the `beq` instruction) if the current argument value *n* (i.e. 0x01) stored at register `$a0` is equal to 0. Since it is not, we skip over this instruction and go on to execute the next series of instructions at addresses 0x00400034 and 0x00400038. These instructions will first copy the value in register `$a0` (i.e. 0x01) onto register `$s0`. The value at register `$a0` is then decremented by 1 and now contains 0x00.

$t7	15	0x00000000
$s0	16	0x00000001
$s1	17	0x00000000

$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000

Figure 8c: Updated contents of `$s0` and `$a0`

The next instruction to be executed (and the last in this fifth call) is at address 0x00400040 which will update the value in the `$ra` register to 0x00400044 (i.e. the instruction address after the `jal` instruction).

$fp	30	0x00000000
$ra	31	0x00400044
pc		0x0040001c

Figure 8d: Register window showing return address stored in register `$ra`

We return back to the start of the factorial procedure. The first three instructions in this procedure to be executed are at addresses 0x0040001C to 0x00400028 which create a new stack frame. Again, executing these instructions allocates 8 bytes of memory to the stack which updates the value stored at the stack pointer (`$sp`).

$gp	28	0x10008000
$sp	29	0x7fffefcc
$fp	30	0x00000000

Figure 9a: Stack pointer value

The value at register `$s0` (i.e. 0x01) is stored at address 0x7FFFEFCC (i.e. 0x7FFFEFC0 + 0x0C offset). Likewise, the value at register $ra is stored at address 0x7FFFEFD0 (i.e. 0x7FFFEFC0 + 0x10 offset) in the Data Segment.

Value (+c)	Value (+10)
0x00400044	0x00000001

The next instruction at address 0x0040002C will check (via the `beq` instruction) if the current argument value *n* stored at register `$a0` (i.e. 0x00) is equal to 0. Since it is, a branch occurs and the `$pc` register points to the next instruction at address 0x00400048 (i.e. the start of the `endcall` procedure). Instructions at addresses 0x00400048 to 0x00400050 pop registers `$s0` and `$ra` off the stack and deallocate memory used for these registers. Hence the stack pointer is reverted to its state as it was in the fourth call.

$gp	28	0x10008000
$sp	29	0x7fffefd4
$fp	30	0x00000000

The next instruction at address 0x00400054 will jump to the return address (via `jr`) stored in register `$ra`. This address is 0x00400044 whose instruction is to multiply return value stored in register `$v0` (i.e. 0x01) with the value stored in register `$s0` (i.e. 0x01) and store the result back into register `$v0`. Hence, the value in register `$v0` is 0x01 after this multiplication.

$at	1	0x00000000
$v0	2	0x00000001
$v1	3	0x00000000

Figure: Value in register `$v0` after multiplication

Return to Fourth Call

The `$pc` register points to the next instruction at address 0x00400048 (i.e. the start of the `endcall` procedure). Once again, instructions at addresses 0x00400048 to 0x00400050 pop registers `$s0` and `$ra` off the stack and deallocate memory used for these registers. Hence the stack pointer is reverted to its state as it was in the fourth call.

$t7	15	0x00000000
$s0	16	0x00000002
$s1	17	0x00000000

$gp	28	0x10008000
$sp	29	0x7fffefdc
$fp	30	0x00000000

Figure 9a: Reverted stack pointer and $s0 value

The next instruction at address 0x00400054 will jump to the return address (via `jr`) stored in register `$ra`. This address is 0x00400044 whose instruction is to multiply return value stored in register `$v0` (i.e. 0x01) with the value stored in register `$s0` (i.e. 0x02) and store the result back into register `$v0`. Hence, the value in register `$v0` is 0x02 after this multiplication.

$at	1	0x00000000
$v0	2	0x00000002
$v1	3	0x00000000

Figure 9b: Value in register `$v0` (i.e. return value) after multiplication

Return to Third Call

The `$pc` register points to the next instruction at address 0x00400048 (i.e. the start of the `endcall` procedure). Once again, instructions at addresses 0x00400048 to 0x00400050 pop registers `$s0` and `$ra` off the stack and deallocate memory used for these registers. Hence the stack pointer is reverted to its state as it was in the third call.

$t7	15	0x00000000
$s0	16	0x00000003
$s1	17	0x00000000

$gp	28	0x10008000
$sp	29	0x7fffefe4
$fp	30	0x00000000

Figure 10a: Reverted stack pointer and $s0 value

The next instruction at address 0x00400054 will jump to the return address (via `jr`) stored in register `$ra`. This address is 0x00400044 whose instruction is to multiply return value stored in register `$v0` (i.e. 0x02) with the value stored in register `$s0` (i.e. 0x03) and store the result back into register `$v0`. Hence, the value in register `$v0` is 0x06 after this multiplication.

$at	1	0x00000000
$v0	2	0x00000006
$v1	3	0x00000000

Figure 10b: Value in register `$v0` (i.e. return value) after multiplication

Return to Second Call

The `$pc` register points to the next instruction at address 0x00400048 (i.e. the start of the `endcall` procedure). Once again, instructions at addresses 0x00400048 to 0x00400050 pop registers `$s0` and `$ra` off the stack and deallocate memory used for these registers. Hence the stack pointer is reverted to its state as it was in the second call.

$t7	15	0x00000000
$s0	16	0x00000004
$s1	17	0x00000000

$gp	28	0x10008000
$sp	29	0x7fffefec
$fp	30	0x00000000

Figure 11a: Reverted stack pointer and $s0 value

The next instruction at address 0x00400054 will jump to the return address (via `jr`) stored in register `$ra`. This address is 0x00400044 whose instruction is to multiply return value stored in register `$v0` (i.e. 0x06) with the value stored in register `$s0` (i.e. 0x04) and store the result back into register `$v0`. Hence, the value in register `$v0` is 0x18 (24 in decimal) after this multiplication.

$at	1	0x00000000
$v0	2	0x00000018
$v1	3	0x00000000

Figure 11b: Value in register `$v0` (i.e. return value) after multiplication

Return to First Call

The `$pc` register points to the next instruction at address 0x00400048 (i.e. the start of the `endcall` procedure). Once again, instructions at addresses 0x00400048 to 0x00400050 pop registers `$s0` and `$ra` off the stack and deallocate memory used for these registers. Hence the stack pointer is reverted to its state as it was in the first call.

$t7	15	0x00000000
$s0	16	0x00000005
$s1	17	0x00000000

$gp	28	0x10008000
$sp	29	0x7fffeff4
$fp	30	0x00000000

Figure 12a: Reverted stack pointer and $s0 value

The next instruction at address 0x00400054 will jump to the return address (via `jr`) stored in register `$ra`. This address is 0x00400044 whose instruction is to multiply return value stored in register `$v0` (i.e. 0x18) with the value stored in register `$s0` (i.e. 0x05) and store the result back into register `$v0`. Hence, the value in register `$v0` is 0x78 (120 in decimal) after this multiplication.

$at	1	0x00000000
$v0	2	0x00000078
$v1	3	0x00000000

Figure 12b: Value in register `$v0` (i.e. return value) after multiplication

At this stage in the program, we have computed factorial(5) and stored the final result in register `$v0`.

Return to Main

The `$pc` register points to the next instruction at address 0x00400048 (i.e. the start of the `endcall` procedure). Once again, instructions at addresses 0x00400048 to 0x00400050 pop registers `$s0` and `$ra` off the stack and deallocate memory used for these registers. Hence the stack pointer is reverted to its state as it was prior to the first procedure call.

$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000

$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000

Figure 13a: Reverted stack pointer and $s0 value

The next instruction at address 0x00400054 will jump to the return address (via `jr`) stored in register `$ra`. This address is 0x0040000C which is back into the main procedure right where we left off. The instruction at this address will store the final value of register $v0 into memory location 0x10010004 and then exit.

Data Segment

Address	Value (+0)	Value (+4)
0x10010000	0x00000005	0x00000078

Figure 13b: Storage of final factorial value in memory

Intel X86 on MS Visual Studio

```
int factorial(int n) {
    if (n == 1) return 1;
    return (n * factorial(n - 1));
}

int main() {
    int N_fact = factorial(5);
}
```

Figure 14: C program to compute factorial

Main Call

```
void main() {
00DE1780  push        ebp
00DE1781  mov         ebp,esp
00DE1783  sub         esp,0CCh
00DE1789  push        ebx
00DE178A  push        esi
00DE178B  push        edi
00DE178C  lea         edi,[ebp-0CCh]
00DE1792  mov         ecx,33h
00DE1797  mov         eax,0CCCCCCCCh
00DE179C  rep stos    dword ptr es:[edi]
00DE179E  mov         ecx,offset _4FF3989B_factorial@cpp (0DEC000h)
00DE17A3  call        @__CheckForDebuggerJustMyCode@4 (0DE1203h)
    int N_fact = factorial(5);
00DE17A8  push        5
00DE17AA  call        factorial (0DE12A3h)
00DE17AF  add         esp,4
00DE17B2  mov         dword ptr [N_fact],eax
}
00DE17B5  xor         eax,eax
00DE17B7  pop         edi
00DE17B8  pop         esi
00DE17B9  pop         ebx
00DE17BA  add         esp,0CCh
00DE17C0  cmp         ebp,esp
00DE17C2  call        __RTC_CheckEsp (0DE120Dh)
00DE17C7  mov         esp,ebp
00DE17C9  pop         ebp
00DE17CA  ret
```

Figure 14a: Disassembly Code for main() function

The preliminary instructions (boxed in black) initializes a new stack frame with stack pointer value 0x00D3F7BC and base pointer value 0x00D3F894.

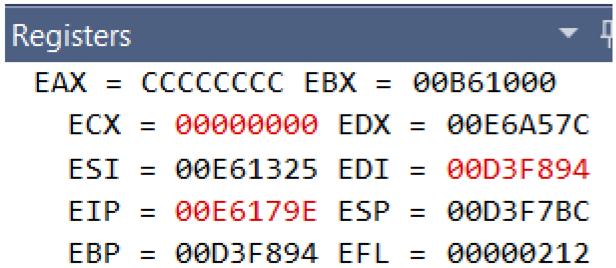

Figure 14b: Initialization of stack frame

The next instructions (in orange) will first push the argument value 5 onto the stack in memory location 0x00D3F7B8 before calling `factorial()`. In addition, the return address (i.e. the address after the call instruction) will be saved in the `EIP` register.

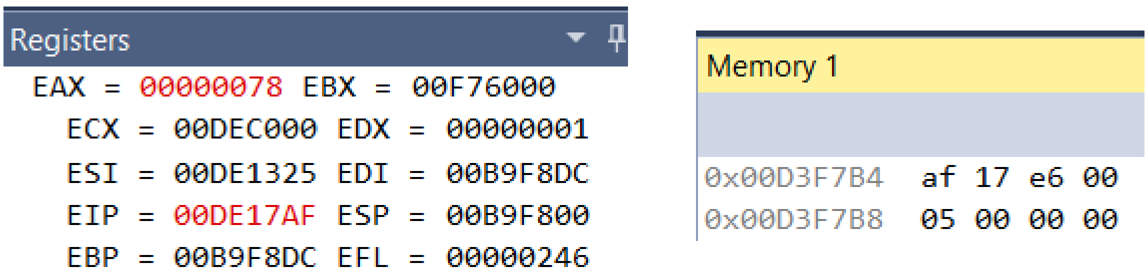

Figure 14c: Storing of return address and memory location of initial argument value

When the call instruction is executed, we will jump to the function `factorial()` whose disassembly code is given below.

First Call

```
int factorial(int n) {
00E616F0  push        ebp
00E616F1  mov         ebp,esp
00E616F3  sub         esp,0C0h
00E616F9  push        ebx
00E616FA  push        esi
00E616FB  push        edi
00E616FC  lea         edi,[ebp-0C0h]
00E61702  mov         ecx,30h
00E61707  mov         eax,0CCCCCCCCh
00E6170C  rep stos    dword ptr es:[edi]
00E6170E  mov         ecx,offset _4FF3989B_factorial@cpp (0E6C000h)
00E61713  call        @__CheckForDebuggerJustMyCode@4 (0E61203h)
    if (n == 1) return 1;
00E61718  cmp         dword ptr [n],1
00E6171C  jne         factorial+35h (0E61725h)
00E6171E  mov         eax,1
00E61723  jmp         factorial+48h (0E61738h)
    return (n * factorial(n - 1));
00E61725  mov         eax,dword ptr [n]
00E61728  sub         eax,1
00E6172B  push        eax
00E6172C  call        factorial (0E612A3h)
00E61731  add         esp,4
00E61734  imul        eax,dword ptr [n]
}
```

Figure 15a: Disassembly Code for factorial() function

The first set of instructions (in black) that will initialize a *new* stack frame with stack pointer and base pointer values 0x00D3F6E4 and 0x00D3F7B0 respectively.

```
Registers
EAX = CCCCCCCC EBX = 00B61000
   ECX = 00000000 EDX = 00000001
   ESI = 00E61325 EDI = 00D3F7B0
   EIP = 00E6170E ESP = 00D3F6E4
   EBP = 00D3F7B0 EFL = 00000206
```

Figure 15b: Initializing a new stack frame (first call)

The next instructions (in orange) will check if the argument value (5) is equal to 1 and perform a jump if they are not equal. Since they are not equal, a jump is performed. The next sequence of instructions (in blue) are executed. First, the argument *n* is copied into register EAX, then it is decremented and pushed back onto the stack at memory location 0x00D3F6E0. This new value will be the argument used in the next function call.

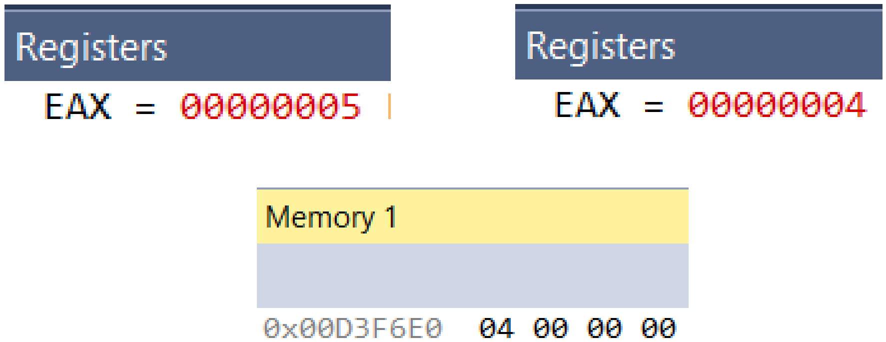

Figure 15c: Value stored EAX register before and after decrementing and location in memory

The call instruction is the last instruction in this first call and will push the return address 0x00E61731 into the EIP register at memory location 0x00D3F6DC on the stack.

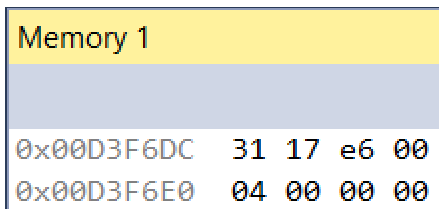

Figure 15d: Memory location and values of current argument value and return address

Second Call

We return back to the first set of instructions (in black) that will initialize a *new* stack frame with stack pointer and base pointer values 0x00D3F60C and 0x00D3F6D8 respectively.

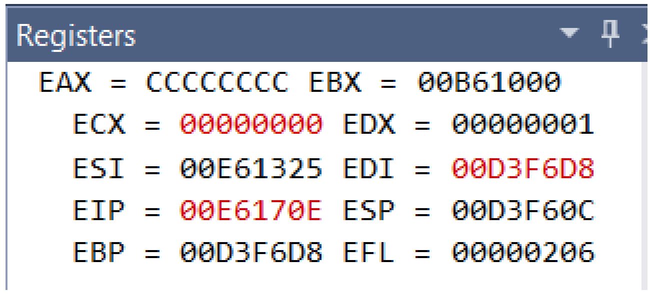

Figure 16a: Initializing a new stack frame (second call)

The next instructions (in orange) will check if the argument value (4) is equal to 1 and perform a jump if they are not equal. Since they are not equal, a jump is performed. The next sequence of instructions (in blue) are executed. First, the argument *n* is copied into register EAX, then it is decremented and pushed back onto the stack at memory location 0x00D3F608. This new value will be the argument used in the next function call.

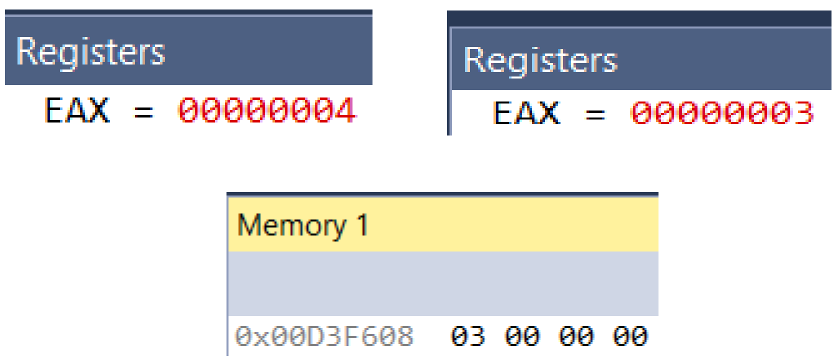

Figure 16b: Value stored EAX register before and after decrementing and location in memory

The call instruction is the last instruction in this first call and will push the return address 0x00E61731 into the EIP register at memory location 0x00D3F604 on the stack.

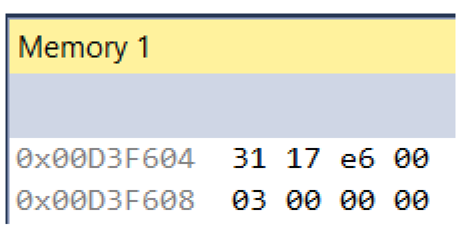

Figure 15c: Memory location and values of current argument value and return address

__Third Call__

We return back to the first set of instructions (in black) that will initialize a *new* stack frame with stack pointer and base pointer values 0x00D3F534 and 0x00D3F600 respectively.

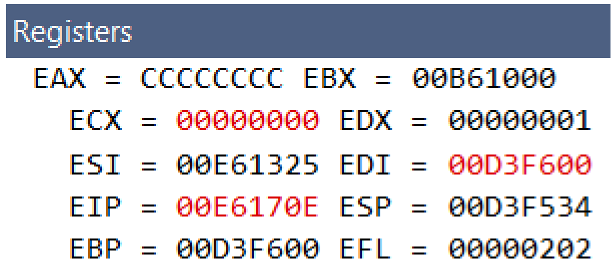

Figure 17a: Initializing a new stack frame (third call)

The next instructions (in orange) will check if the current argument value (3) is equal to 1 and perform a jump if they are not equal. Since they are not equal, a jump is performed. The next sequence of instructions (in blue) are executed. First, the argument *n* is copied into register EAX, then it is decremented and pushed back onto the stack at memory location 0x00D3F530. This new value will be the argument used in the next function call.

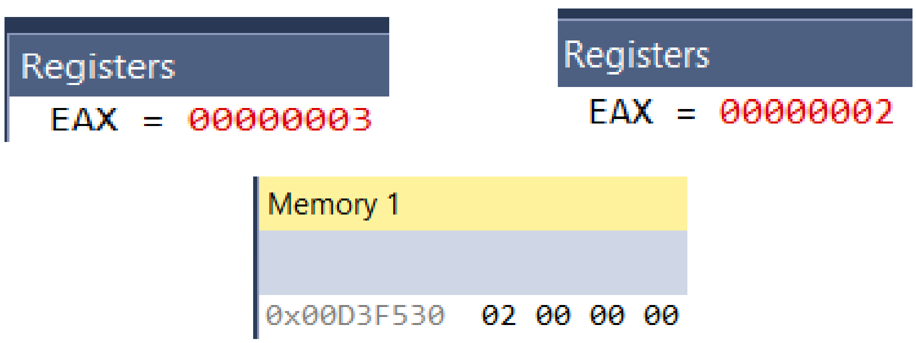

Figure 17b: Value stored EAX register before and after decrementing and location in memory

The call instruction is the last instruction in this first call and will push the return address 0x00E61731 into the EIP register at memory location 0x00D3F52C on the stack.

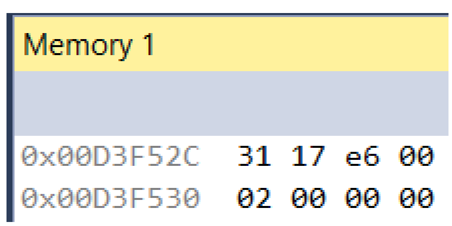

Figure 17c: Memory location and values of current argument value and return address

Fourth Call

We return back to the first set of instructions (in black) that will initialize a *new* stack frame with stack pointer and base pointer values 0x00D3F45C and 0x00D3F528 respectively.

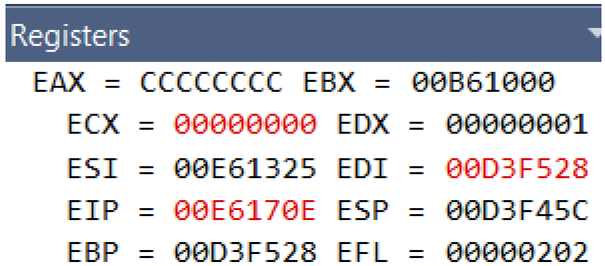

Figure 18a: Initializing a new stack frame (fourth call)

The next instructions (in orange) will check if the current argument value (2) is equal to 1 and perform a jump if they are not equal. Since they are not equal, a jump is performed. The next sequence of instructions (in blue) are executed. First, the argument *n* is copied into register EAX, then it is decremented and pushed back onto the stack at memory location 0x00D3F458. This new value will be the argument used in the next function call.

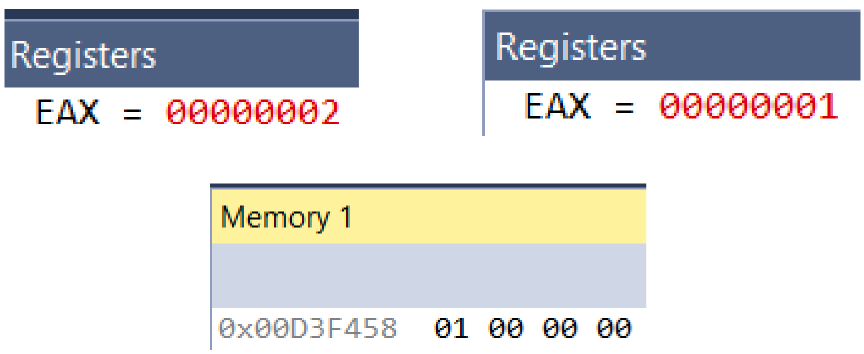

Figure 18b: Value stored EAX register before and after decrementing and location in memory

The call instruction is the last instruction in this first call and will push the return address 0x00E61731 into the EIP register at memory location 0x00D3F454 on the stack.

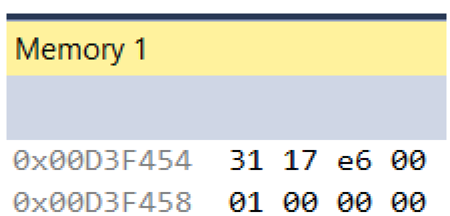

Figure 18c: Memory location and values of current argument value and return address

Fifth Call

We return back to the first set of instructions (in black) that will initialize a *new* stack frame with stack pointer and base pointer values 0x00D3F384 and 0x00D3F450 respectively.

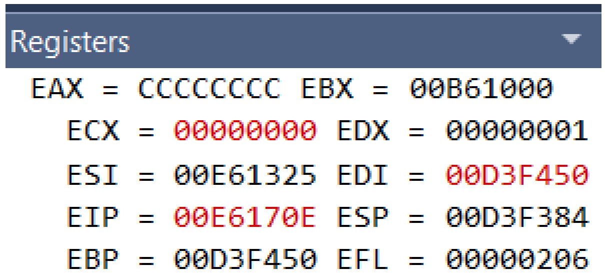

Figure 19a: Initializing a new stack frame (fifth call)

The next instructions (in orange) will check if the current argument value (1) is equal to 1 and perform a jump if they are not equal. Now they are equal and so a jump is not performed. The next sequence of instructions (in green) are executed. The value 1 will be moved to register EAX and a jump will occur to address 0x00E61738 which will deallocate memory off the stack and go to the return address 0x00E61731. Figure 19b shows all the stack frames created throughout the program where the arguments, return addresses, and base pointers that were pushed onto the stack are represented in purple, yellow, and red respectively.

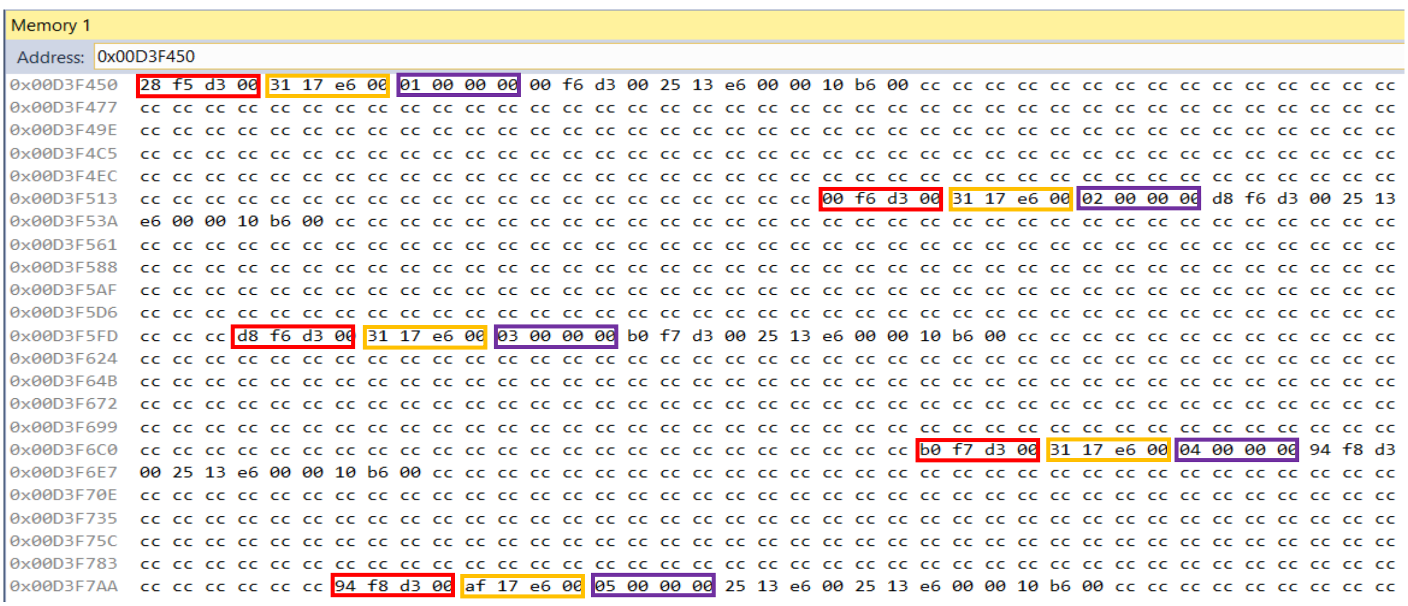

Figure 19b: Stack frames used throughout the program

Return to Fourth Call

The stack frame is reverted as it was during the fourth call after the execution of the instruction at the return address.

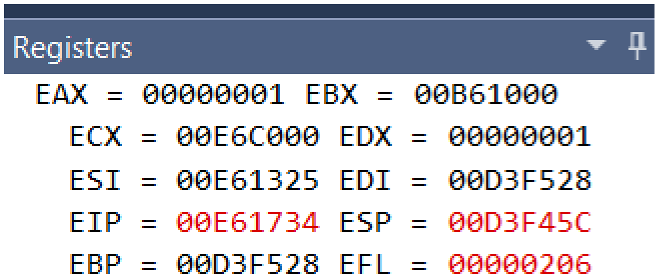

Figure 20a: Reverted stack frame (Return to Fourth Call)

Following that instruction, the instruction (in yellow) will multiply the current value in the EAX register (i.e. 1) with the argument value that was available during the fourth function call (i.e. 2). The result of this multiplication is stored back into register EAX. Hence, the value stored at this register is 2. This EAX value will be used for when we revert back to the previous function call.

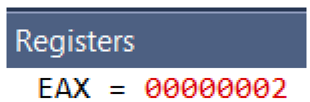

Figure 20b: Contents of register EAX storing the result of multiplication

Return to Third Call

The stack frame is reverted as it was during the third call after the execution of the instruction at the return address.

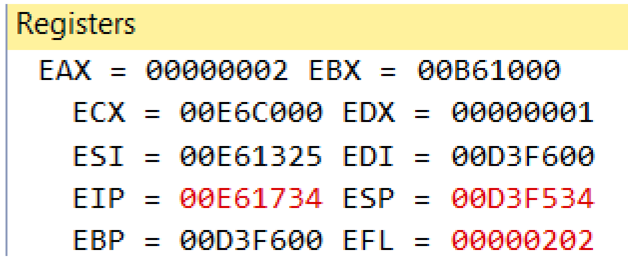

Figure 21a: Reverted stack frame (Return to Third Call)

Following that instruction, the instruction (in yellow) will multiply the current value in the EAX register (i.e. 2) with the argument value that was available during the fourth function call (i.e. 3). The result of this multiplication is stored back into register EAX. Hence, the value stored at this register is 6. This EAX value will be used for when we revert back to the previous function call.

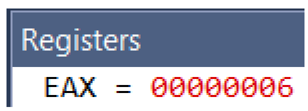

Figure 21b: Contents of register EAX storing the result of multiplication

Return to Second Call

The stack frame is reverted as it was during the second call after the execution of the instruction at the return address.

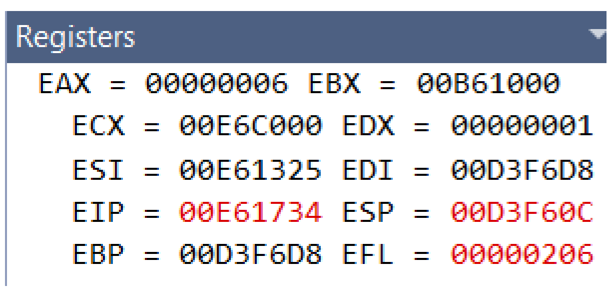

Figure 22a: Reverted stack frame (Return to Second Call)

Following that instruction, the instruction (in yellow) will multiply the current value in the EAX register (i.e. 6) with the argument value that was available during the fourth function call (i.e. 4). The result of this multiplication is stored back into register EAX. Hence, the value stored at this register is 24. This EAX value will be used for when we revert back to the previous function call.

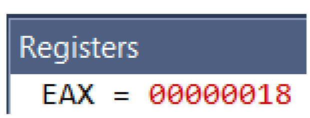

Figure 22b: Contents of register EAX storing the result of multiplication

Return to First Call

The stack frame is reverted as it was during the second call after the execution of the instruction at the return address.

```
Registers
EAX = 00000018 EBX = 00B61000
  ECX = 00E6C000 EDX = 00000001
  ESI = 00E61325 EDI = 00D3F7B0
  EIP = 00E61734 ESP = 00D3F6E4
  EBP = 00D3F7B0 EFL = 00000206
```

Figure 23a: Reverted stack frame (Return to First Call)

Following that instruction, the instruction (in yellow) will multiply the current value in the EAX register (i.e. 24) with the argument value that was available during the fourth function call (i.e. 5). The result of this multiplication is stored back into register EAX. Hence, the value stored at this register is 120 in decimal. This EAX value will is the final value to be used for when we revert back to main().

```
Registers
EAX = 00000078
```

Figure 23b: Final value stored in EAX after multiplication

Return to Main

We have returned to main() due to the return address 0x00E617AF. The stack frame is reverted as it was before any function calls after the execution of the instruction at the return address.

```
Registers
EAX = 00000078 EBX = 00B61000
  ECX = 00E6C000 EDX = 00000001
  ESI = 00E61325 EDI = 00D3F894
  EIP = 00E617B2 ESP = 00D3F7BC
  EBP = 00D3F894 EFL = 00000202
```

Figure 24a: Reverted stack frame (Return to Main)

The final crucial instruction (in blue) will move the return value stored in register EAX (i.e. 120) to local variable `N_Fact`.

Linux 64-Bit on Intel With GCC and GDB

Consider the previous C program used to compute the factorial of some integer. We assign `factorial()` with argument 5 to local variable `N_fact` and debug the program in a Linux environment using GDB.

Main Call

```
Dump of assembler code for function main:
   0x00005555555545fa <+0>:     push   %rbp
   0x00005555555545fb <+1>:     mov    %rsp,%rbp
   0x00005555555545fe <+4>:     sub    $0x10,%rsp
=> 0x0000555555554602 <+8>:     mov    $0x5,%edi
   0x0000555555554607 <+13>:    callq  0x555555554616 <factorial>
   0x000055555555460c <+18>:    mov    %eax,-0x4(%rbp)
   0x000055555555460f <+21>:    mov    $0x0,%eax
   0x0000555555554614 <+26>:    leaveq
   0x0000555555554615 <+27>:    retq
End of assembler dump.
```

Figure 25a: Disassembly for `main()`

At the start of the `main()` function, the first set of instructions (in orange) are executed. A stack frame is initialized with initial base pointer (i.e. old based pointer) value 0x7FFFFFFFE0D0 (in white). The *new* base pointer pushed onto the stack has value 0x7FFFFFFFDFF0 (in magenta). A subtraction is performed on the stack pointer to allocate space (16 bytes) for variables (i.e. local variable `N_fact`). The stack pointer has value 0x7FFFFFFFDFE0 as a result (in green). The argument value 5 is also moved onto register `$edi` but *not* pushed onto the stack (unlike MS Visual Studio). Nowhere in this program will registers be stored on the stack!

```
(gdb) x /xg ($rbp)
0x7fffffffdff0: 0x0000555555554650
(gdb) print /x $rbp
$1 = 0x7fffffffdff0
(gdb) x /xg ($rsp)
0x7fffffffdfe0: 0x00007fffffffe0d0
(gdb) print /x $rsp
$2 = 0x7fffffffdfe0
```

```
(gdb) print /x $edi
$3 = 0x5
```

Figure 25b: Initial stack frame and initial argument value

The next instruction (in blue) performs the execution of the `callq` instruction which pushes the *return address* 0x55555555460C (i.e., the instruction after `callq`) onto the stack. This is the last operation executed in `main()` prior to jumping to `factorial()`.

```
(gdb) x /xg ($rsp)
0x7fffffffdfd8: 0x000055555555460c
```

Figure 25c: Pushing the Return Address to the stack

First Call

The assembler code for the `factorial()` function is given below. Notice the instruction at 0x555555554636 is a call to the same function `factorial()`. Hence, we know that this is a recursive function.

```
Dump of assembler code for function factorial:
=> 0x0000555555554616 <+0>:     push   %rbp
   0x0000555555554617 <+1>:     mov    %rsp,%rbp
   0x000055555555461a <+4>:     sub    $0x10,%rsp
   0x000055555555461e <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555554621 <+11>:    cmpl   $0x1,-0x4(%rbp)
   0x0000555555554625 <+15>:    jne    0x55555555462e <factorial+24>
   0x0000555555554627 <+17>:    mov    $0x1,%eax
   0x000055555555462c <+22>:    jmp    0x55555555463f <factorial+41>
   0x000055555555462e <+24>:    mov    -0x4(%rbp),%eax
   0x0000555555554631 <+27>:    sub    $0x1,%eax
   0x0000555555554634 <+30>:    mov    %eax,%edi
   0x0000555555554636 <+32>:    callq  0x555555554616 <factorial>
   0x000055555555463b <+37>:    imul   -0x4(%rbp),%eax
   0x000055555555463f <+41>:    leaveq
   0x0000555555554640 <+42>:    retq
End of assembler dump.
```

Figure 26a: Assembler Code for `factorial()`

At the beginning of this function call, the instructions (in orange) create a new stack frame as shown in figure 26b below. The *now* old base pointer 0x7FFFFFFFDFF0 (in white) is pushed onto the stack. The *new* base and stack pointers have values 0x7FFFFFFFDFD0 (in magenta) and 0x7FFFFFFFDFC0 (in green). Note that our argument value 0x05 (in purple) initialized in `main()` and previously stored in register `$edi` is also pushed onto the stack.

```
0x7fffffffdfc0: 0x00007ffff7de3b40      0x0000000500000000
(gdb) x /2xg ($rbp)
```

```
(gdb) print /x $rsp
$4 = 0x7fffffffdfc0
(gdb) x /2xg ($rbp)
0x7fffffffdfd0: 0x00007fffffffdff0      0x000055555555460c
(gdb) print /x $rbp
$5 = 0x7fffffffdfd0
```

Figure 26b: Stack Frame (First Call)

The next instructions in blue will compare whether or not our argument (0x0) is equal to 1. If not, then a jump will occur from 0x55555555425 to 0x5555555542E. After jumping to instruction at address 0x55555555462E, the next instructions (in yellow) will first move the argument that was previously pushed onto the stack (i.e. 0x05) to register `$eax`, decrement it by 1, and move the result to register `$edi`. The value now stored in this register (i.e. 0x04) will become the new argument of our `factorial()` function in the next call.

```
(gdb) print /x $eax
$6 = 0x4
(gdb) print /x $edi
$7 = 0x4
```

Figure 26c: Register values of `$eax` and `$edi` after first call

The execution of `callq` will terminate this function call by pushing the return address 0x55555555463B (i.e., the instruction of `imul` after `callq`) onto the stack pointer prior to pushing the base pointer 0x7FFFFFFFDFD0 (which occurs in the next call).

```
(gdb) x /xg ($rsp)
0x7fffffffdfb8: 0x000055555555463b
```

Figure 26d: Pushing the return address onto the stack (after first call)

Second Call

In the next call, we remain within the function `factorial()` and return to the set of instructions in orange. The old base pointer 0x7FFFFFFFDFD0 (in white) is pushed onto the stack. A *new* stack frame is created once again with base pointer 0x7FFFFFFFDFB0 (in magenta) and stack pointer 0x7FFFFFFDFA0 (in green). Note that the updated argument value 0x04 (in purple) is also pushed onto the stack.

```
(gdb) x /2xg ($rsp)
0x7fffffffdfa0: 0x00007fffffffe008      0x0000000400f0b5ff
(gdb) print /x $rsp
$8 = 0x7fffffffdfa0
(gdb) x /2xg ($rbp)
0x7fffffffdfb0: 0x00007fffffffdfd0      0x000055555555463b
(gdb) print /x $rbp
$9 = 0x7fffffffdfb0
```

Figure 27a: Stack Frame (Second Call)

The next set of instructions in blue will compare the argument value (i.e., 0x04) and perform a jump to 0x55555555462E since it is not equal to 1. The next instructions in yellow will first move the argument that was previously pushed onto the stack (i.e. 0x04) to register `$eax`, decrement it by 1, and move the result to register `$edi`. The value now stored in this register (i.e., 0x03) will become the new argument of our `factorial()` function in the next call.

```
(gdb) print /x $eax
$10 = 0x3
(gdb) print /x $edi
$11 = 0x3
```

Figure 27b: Register values of `$eax` and `$edi` after second call

The execution of `callq` will terminate this function call by pushing the return address 0x5555555546A1 (i.e., the instruction of `imul` after `callq`) onto the stack pointer prior to pushing the base pointer 0x7FFFFFFFDFB0.

```
(gdb) x /xg ($rsp)
0x7fffffffdfb8: 0x000055555555463b
```

Figure 27c: Pushing the return address onto the stack (after second call)

Third Call

In the next call, we remain within the function `factorial()` and return to the set of instructions in orange. The old base pointer 0x7FFFFFFFFB0 (in white) is pushed onto the stack. A *new* stack frame is created once again with base pointer 0x7FFFFFFFFDF90 (in magenta) and stack pointer 0x7FFFFFFFDF80 (in green). Note that the updated argument value 0x03 (in purple) is also pushed onto the stack.

```
(gdb) x /2xg ($rsp)
0x7fffffffdf80: 0x0000000000000000      0x0000000300000000
(gdb) print /x $rsp
$13 = 0x7fffffffdf80
(gdb) x /2xg ($rbp)
0x7fffffffdf90: 0x00007fffffffdfb0      0x000055555555463b
(gdb) print /x $rbp
$14 = 0x7fffffffdf90
```

Figure 28a: Stack Frame (Third Call)

The next set of instructions in blue will compare the argument value (i.e., 0x03) and perform a jump to 0x55555555462E since it is not equal to 1. The next instructions in yellow will first move the argument that was previously pushed onto the stack (i.e. 0x03) to register `$eax`, decrement it by 1, and move the result to register `$edi`. The value now stored in this register (i.e., 0x02) will become the new argument of our `factorial()` function in the next call.

```
(gdb) print /x $eax
$15 = 0x2
(gdb) print /x $edi
$16 = 0x2
```

Figure 28b: Register values of `$eax` and `$edi` after third call

The execution of `callq` will terminate this function call by pushing the return address 0x55555555463B (i.e., the instruction of `imul` after `callq`) onto the stack pointer prior to pushing the base pointer 0x7FFFFFFFDF__.

```
(gdb) x /xg ($rsp)
0x7fffffffdfb8: 0x000055555555463b
```

Figure 28c: Pushing the return address onto the stack (after third call)

Third Call

In the next call, we remain within the function `factorial()` and return to the set of instructions in orange. The old base pointer 0x7FFFFFFFFB0 (in white) is pushed onto the stack. A *new* stack frame is created once again with base pointer 0x7FFFFFFFFDF90 (in magenta) and stack pointer 0x7FFFFFFFDF80 (in green). Note that the updated argument value 0x03 (in purple) is also pushed onto the stack.

```
(gdb) x /2xg ($rsp)
0x7fffffffdf80: 0x0000000000000000      0x0000000300000000
(gdb) print /x $rsp
$13 = 0x7fffffffdf80
(gdb) x /2xg ($rbp)
0x7fffffffdf90: 0x00007fffffffdfb0      0x000055555555463b
(gdb) print /x $rbp
$14 = 0x7fffffffdf90
```

Figure 29a: Stack Frame (Third Call)

The next set of instructions in blue will compare the argument value (i.e., 0x03) and perform a jump to 0x55555555462E since it is not equal to 1. The next instructions in yellow will first move the argument that was previously pushed onto the stack (i.e. 0x03) to register `$eax`, decrement it by 1, and move the result to register `$edi`. The value now stored in this register (i.e., 0x02) will become the new argument of our `factorial()` function in the next call.

```
(gdb) print /x $eax
$15 = 0x2
(gdb) print /x $edi
$16 = 0x2
```

Figure 29b: Register values of `$eax` and `$edi` after third call

The execution of `callq` will terminate this function call by pushing the return address 0x55555555463B (i.e., the instruction of `imul` after `callq`) onto the stack pointer prior to pushing the base pointer 0x7FFFFFFFFDF90.

```
(gdb) x /xg ($rsp)
0x7fffffffdfb8: 0x000055555555463b
```

Figure 29c: Pushing the return address onto the stack (after third call)

Fourth Call

In the next call, we remain within the function `factorial()` and return to the set of instructions in orange. The old base pointer 0x7FFFFFFF90 (in white) is pushed onto the stack. A *new* stack frame is created once again with base pointer 0x7FFFFFFFDF70 (in magenta) and stack pointer 0x7FFFFFFDF60 (in green). Note that the updated argument value 0x02 (in purple) is also pushed onto the stack.

```
(gdb) x /2xg ($rsp)
0x7fffffffdf60: 0x00007ffff7ffb2a8      0x00000002f7ffe710
(gdb) print /x $rsp
$17 = 0x7fffffffdf60
(gdb) x /2xg ($rbp)
0x7fffffffdf70: 0x00007fffffffdf90      0x000055555555463b
(gdb) print /x $rbp
$18 = 0x7fffffffdf70
```

Figure 30a: Stack Frame (Fourth Call)

The next set of instructions in blue will compare the argument value (i.e., 0x02) and perform a jump to 0x55555555462E since it is not equal to 1. The next instructions in yellow will first move the argument that was previously pushed onto the stack (i.e. 0x02) to register `$eax`, decrement it by 1, and move the result to register `$edi`. The value now stored in this register (i.e., 0x01) will become the new argument of our `factorial()` function in the next call.

```
(gdb) print /x $eax
$19 = 0x1
(gdb) print /x $edi
$20 = 0x1
```

Figure 30b: Register values of `$eax` and `$edi` after fourth call

The execution of `callq` will terminate this function call by pushing the return address 0x55555555463B (i.e., the instruction of `imul` after `callq`) onto the stack pointer prior to pushing the base pointer 0x7FFFFFFFDF70.

```
(gdb) x /xg ($rsp)
0x7fffffffdfb8: 0x000055555555463b
```

Figure 30c: Pushing the return address onto the stack (after fourth call)

Fifth Call

In the next call, we remain within the function `factorial()` and return to the set of instructions in orange. The old base pointer 0x7FFFFFFFDF70 (in white) is pushed onto the stack. A *new* stack frame is created once again with base pointer 0x7FFFFFFFDF50 (in magenta) and stack pointer 0x7FFFFFFDF40 (in green). Note that the updated argument value 0x01 (in purple) is also pushed onto the stack.

```
(gdb) x /2xg ($rsp)
0x7fffffffdf40: 0x0000000000000000      0x00000001ffffdf70
(gdb) print /x $rsp
$21 = 0x7fffffffdf40
(gdb) x /2xg ($rbp)
0x7fffffffdf50: 0x00007fffffffdf70      0x000055555555463b
(gdb) print /x $rbp
$22 = 0x7fffffffdf50
```

Figure 31a: Stack Frame (Fifth Call)

In the next set of instructions in blue, the value of the argument is now 0x01 (i.e., equal to 1) and so no jump will occur. Figure 31b below shows all stack frames that were used in the program. The stack pointers, base pointers, return addresses, and argument values are shown in green, magenta, amber, and purple respectively.

```
0x7fffffffdf40: 0x0000000000000000      0x00000001ffffdf70
0x7fffffffdf50: 0x00007fffffffdf70      0x000055555555463b
0x7fffffffdf60: 0x00007ffff7ffb2a8      0x00000002f7ffe710
0x7fffffffdf70: 0x00007fffffffdf90      0x000055555555463b
0x7fffffffdf80: 0x0000000000000000      0x0000000300000000
0x7fffffffdf90: 0x00007fffffffdfb0      0x000055555555463b
0x7fffffffdfa0: 0x00007fffffffe008      0x0000000400f0b5ff
0x7fffffffdfb0: 0x00007fffffffdfd0      0x000055555555463b
0x7fffffffdfc0: 0x00007ffff7de3b40      0x0000000500000000
0x7fffffffdfd0: 0x00007fffffffdff0      0x000055555555460c
```

Figure 31b: Stack frames of entire program

Return to Fourth Call

```
Dump of assembler code for function factorial:
   0x0000555555554616 <+0>:     push   %rbp
   0x0000555555554617 <+1>:     mov    %rsp,%rbp
   0x000055555555461a <+4>:     sub    $0x10,%rsp
   0x000055555555461e <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555554621 <+11>:    cmpl   $0x1,-0x4(%rbp)
   0x0000555555554625 <+15>:    jne    0x55555555462e <factorial+24>
=> 0x0000555555554627 <+17>:    mov    $0x1,%eax
   0x000055555555462c <+22>:    jmp    0x55555555463f <factorial+41>
   0x000055555555462e <+24>:    mov    -0x4(%rbp),%eax
   0x0000555555554631 <+27>:    sub    $0x1,%eax
   0x0000555555554634 <+30>:    mov    %eax,%edi
   0x0000555555554636 <+32>:    callq  0x555555554616 <factorial>
   0x000055555555463b <+37>:    imul   -0x4(%rbp),%eax
   0x000055555555463f <+41>:    leaveq
   0x0000555555554640 <+42>:    retq
End of assembler dump.
```

Figure 32a: Assembler Code of factorial() after five calls

Since no jump occurs, the next instructions in blue first value 0x01 is moved to register `$eax`. Then a jump will occur to the instruction at address 0x55555555463F which will terminate the fourth call. The `leaveq` instruction will revert the stack frame to its previous state prior to the fifth function call by restoring the stack and base pointers. Also, `retq` will jump back to the return address stored on top of the stack. Hence, the next instruction to be executed (in orange) is at address 0x55555555463B (i.e., the return address). Figure 32b shows the reverted stack frame (prior to the fifth call). The return address (in amber) will also be popped off the stack. Note that figure 32b below is the stack frame as it was shown in figure 30a.

```
(gdb) x /2xg ($rsp)
0x7fffffffdf60: 0x00007ffff7ffb2a8      0x00000002f7ffe710
(gdb) print /x $rsp
$25 = 0x7fffffffdf60
(gdb) x /2xg ($rbp)
0x7fffffffdf70: 0x00007fffffffdf90      0x000055555555463b
(gdb) print /x $rbp
$26 = 0x7fffffffdf70
```

Figure 32b: Reverted stack frame prior to fifth call

The value of the argument (in purple) in this stack frame is 0x02. This is multiplied by value contained in register `$eax` (i.e., 0x01) where the result is stored back into `$eax`. At this point, we have obtained the final return value of `factorial(2)` function (i.e., 2! = 0x02 = 2).

```
(gdb) print /x $eax
$27 = 0x2
```

Figure 32c: Value stored in register `$eax` after multiplication

Return to Third Call

After the execution of the `imul` instruction, we return back to the set of instructions in blue. Now, The `leaveq` instruction will revert the stack frame to its previous state prior to the fourth function call by restoring the stack and base pointers. Also, `retq` will jump back to the return address stored on top of the stack. Again, the next instruction to be executed (in orange) is at the return address. Figure 33a shows the reverted stack frame (prior to the fourth call). The return address (in amber) will also be popped off the stack. Note that figure 33a below is the stack frame as it was shown in figure 29a.

```
(gdb) x /2xg ($rsp)
0x7fffffffdf80: 0x0000000000000000      0x0000000300000000
(gdb) print /x $rsp
$28 = 0x7fffffffdf80
(gdb) x /2xg ($rbp)
0x7fffffffdf90: 0x00007fffffffdfb0      0x000055555555463b
(gdb) print /x $rbp
$29 = 0x7fffffffdf90
```

Figure 33a: Reverted stack frame prior to fourth call (return to third call)

The value of the argument (in purple) in this stack frame is 0x03. This is multiplied by value contained in register `$eax` (i.e., 0x02) where the result is stored back into `$eax`. At this point, we have obtained the final return value of `factorial(3)` function (i.e., 3! = 0x6 = 6).

```
(gdb) print /x $eax
$30 = 0x6
```

Figure 33b: Value stored in register `$eax` after multiplication

Return to Second Call

After the execution of the `imul` instruction, we return back to the set of instructions in blue. Now, The `leaveq` instruction will revert the stack frame to its previous state prior to the fourth function call by restoring the stack and base pointers. Also, `retq` will jump back to the return address stored on top of the stack. Again, the next instruction to be executed (in orange) is at the return address. Figure 34a shows the reverted stack frame (prior to the third call). The return address (in amber) will also be popped off the stack. Note that figure 34a below is the stack frame as it was shown in figure 27a.

```
(gdb) x /2xg ($rsp)
0x7fffffffdfa0: 0x00007fffffffe008      0x0000000400f0b5ff
(gdb) print /x $rsp
$34 = 0x7fffffffdfa0
(gdb) x /2xg ($rbp)
0x7fffffffdfb0: 0x00007fffffffdfd0      0x000055555555463b
(gdb) print /x $rbp
$35 = 0x7fffffffdfb0
```

Figure 34a: Reverted stack frame prior to third call (return to second call)

The value of the argument (in purple) in this stack frame is 0x04. This is multiplied by value contained in register `$eax` (i.e., 0x06) where the result is stored back into `$eax`. At this point, we have obtained the final return value of `factorial(4)` function (i.e., 4! = 0x18 = 24).

```
(gdb) print /x $eax
$36 = 0x18
```

Figure 34b: Value stored in register `$eax` after multiplication

Return to First Call

After the execution of the `imul` instruction, we return back to the set of instructions in blue. Now, The `leaveq` instruction will revert the stack frame to its previous state prior to the fourth function call by restoring the stack and base pointers. Also, `retq` will jump back to the return address stored on top of the stack. Again, the next instruction to be executed (in orange) is at the return address. Figure 35a shows the reverted stack frame (prior to the second call). The return address (in amber) will also be popped off the stack. Note that figure 35a below is the stack frame as it was shown in figure 26b.

```
(gdb) x /2xg ($rsp)
0x7fffffffdfc0: 0x00007ffff7de3b40      0x0000000500000000
(gdb) print /x $rsp
$37 = 0x7fffffffdfc0
(gdb) x /2xg ($rbp)
0x7fffffffdfd0: 0x00007fffffffdff0      0x000055555555460c
(gdb) print /x $rbp
$38 = 0x7fffffffdfd0
```

Figure 35a: Reverted stack frame prior to second call (return to first call)

The value of the argument (in purple) in this stack frame is 0x05. This is multiplied by value contained in register `$eax` (i.e., 0x18) where the result is stored back into `$eax`. At this point, we have obtained the final return value of `factorial(5)` function (i.e., 5! = 0x78 = 120).

```
(gdb) print /x $eax
$39 = 0x78
```

Figure 35b: Value stored in register `$eax` after multiplication

The next instruction to be executed will be the return address 0x55555555460C, which is located in `main()`.

Return to Main

```
Dump of assembler code for function main:
   0x00005555555545fa <+0>:     push   %rbp
   0x00005555555545fb <+1>:     mov    %rsp,%rbp
   0x00005555555545fe <+4>:     sub    $0x10,%rsp
   0x0000555555554602 <+8>:     mov    $0x5,%edi
   0x0000555555554607 <+13>:    callq  0x555555554616 <factorial>
=> 0x000055555555460c <+18>:    mov    %eax,-0x4(%rbp)
   0x000055555555460f <+21>:    mov    $0x0,%eax
   0x0000555555554614 <+26>:    leaveq
   0x0000555555554615 <+27>:    retq
```

Figure 36a: Assembler Code of `main()` after returning from `factorial()`

We've returned back to the `main()` function and starting where we left off (after the `callq` instruction). Figure 36b shows the reverted stack frame as it was in figure 25b.

```
(gdb) x /2xg ($rsp)
0x7fffffffdfe0: 0x00007fffffffe0d0      0x0000000000000000
(gdb) print /x $rsp
$40 = 0x7fffffffdfe0
(gdb) x /2xg ($rbp)
0x7fffffffdff0: 0x0000555555554650      0x00007ffff7a03bf7
(gdb) print /x $rbp
$41 = 0x7fffffffdff0
```

Figure 36b: Reverted stack frame (return to main)

The last crucial instruction to be executed (in blue) is to move the value stored in register `$eax` (i.e., the return value of `factorial(5)`) to the location of local variable `N_fact` on the stack.

```
(gdb) x $rbp - 0x4
0x7fffffffdfec: 0x00000078
(gdb) x &N_fact
0x7fffffffdfec: 0x00000078
```

Figure 36c: Location and value of local variable `N_fact` on stack

The final instructions in white simply deallocate memory and revert the stack frame to its state prior to running the program.

Execution Time of factorial()

To compute the execution time of the factorial function, we utilize some built in tools available in C++. Namely, the previous program was modified to utilize a clock to measure the execution times for the factorial function for the cases N = 10, 100, 1000, 10000.

```
#include <chrono>
#include <iostream>

int factorial(int n) {
        if (n == 1) return 1;
        return (n * factorial(n - 1));
}

int main() {
        using std::chrono::high_resolution_clock;
        using std::chrono::duration_cast;
        using std::chrono::duration;
        using std::chrono::milliseconds;

        int N_fact;

        auto t1 = high_resolution_clock::now();
        N_fact = factorial(10);
        auto t2 = high_resolution_clock::now();

        duration<double, std::milli> execution_time1 = t2 - t1;
        std::cout << "Execution Time: " << execution_time1.count() << " ms\n";

        auto t3 = high_resolution_clock::now();
        N_fact = factorial(100);
        auto t4 = high_resolution_clock::now();

        duration<double, std::milli> execution_time2 = t4 - t3;
        std::cout << "Execution Time: " << execution_time2.count() << " ms\n";

        auto t5 = high_resolution_clock::now();
        N_fact = factorial(1000);
        auto t6 = high_resolution_clock::now();

        duration<double, std::milli> execution_time3 = t6 - t5;
        std::cout << "Execution Time: " << execution_time3.count() << " ms\n";

        auto t7 = high_resolution_clock::now();
        N_fact = factorial(10000);
        auto t8 = high_resolution_clock::now();

        duration<double, std::milli> execution_time4 = t8 - t7;
        std::cout << "Execution Time: " << execution_time4.count() << " ms\n";
}
```

Figure 37a: Modified factorial.cpp program to measure execution time

The execution times were determined by executing 5 program runs and computing their averages and plotting them. It can be seen that as the input increases rapidly, so does the execution time

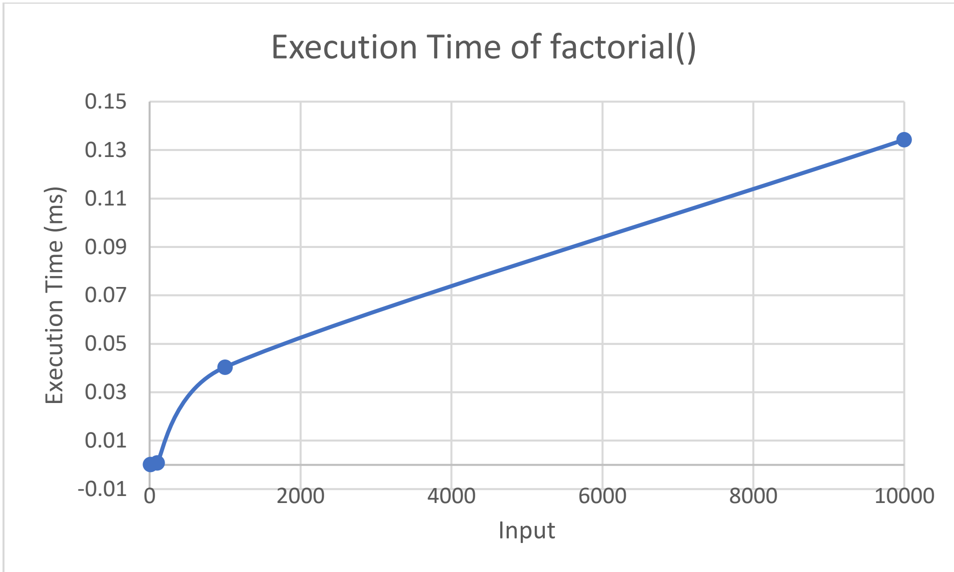

Figure 37b: Plot of execution time as of function of inputs

II. Recursive GCD

The next example is a recursive method of determining the *greatest common factor (GCD)* of two integers a, b > 0.

MIPS on MARS Simulator

```
.data
        a: .word 6
        b: .word 22
        gcd_res: .word 0
.text
main:
        lw $a0, a # load value a
        lw $a1, b # load value b
        jal gcd
        sw $v0, gcd_res # save the return value

        li $v0, 10 # end program
        syscall

gcd: # procedure to calculate gcd(a,b)
        addi $sp, $sp, -12 # adjust stack pointer for 3 items
        sw $s1, 8($sp) # store the first argument
        sw $s0, 4($sp) # store the second arguement
        sw $ra, 0($sp) # store the return address

        add $s0, $a0, $zero # s0 = a0 (s0 = a)
        add $s1, $a1, $zero # s0 = a0 (s1 = b)

        # first iteration
        div $a0, $a1 # divide a by b (remainder is stored in $mfhi)
        mfhi $s0 # store remainder in $a1 (i.e. result of a%b)
        sw $s0, 4($sp) #save the remainder
        bne $s0, $zero, L1 # branch to label if remainder != 0

        # base case
        add $v0, $zero, $a1 # $v0 = $a1 (i.e. return b)
        addi $sp, $sp 12 # adjust stack pointer to pop twice
        jr $ra # jump to return address (i.e. main)

L1: # recursive procedure
        add $a0, $a1, $zero # $a0 = $s1 (a = b)
        lw $s0, 4($sp) # load remainder
        add $a1, $s0, $zero # $a1 = $s0 (b = remainder)

        jal gcd

        # exit
        lw $ra, 0($sp) # restore the return address
        lw $s0, 4($sp) # restore the second argument
        lw $s1, 8($sp) # restore the first argument
        addi $sp, $sp, 12 # adjust stack pointer to pop twice
        jr $ra # jump to return address
```

Figure 38a: gcd.asm

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	7: lw $a0, a # load value a
	0x00400004	0x8c240000	lw $4,0x00000000($1)	
	0x00400008	0x3c011001	lui $1,0x00001001	8: lw $a1, b # load value b
	0x0040000c	0x8c250004	lw $5,0x00000004($1)	
	0x00400010	0x0c100009	jal 0x00400024	9: jal gcd
	0x00400014	0x3c011001	lui $1,0x00001001	10: sw $v0, gcd_res # save the return value
	0x00400018	0xac220008	sw $2,0x00000008($1)	
	0x0040001c	0x2402000a	addiu $2,$0,0x0000000a	12: li $v0, 10 # end program
	0x00400020	0x0000000c	syscall	13: syscall
	0x00400024	0x23bdfff4	addi $29,$29,0xffff...	16: addi $sp, $sp, -12 # adjust stack pointer for 3 items
	0x00400028	0xafb10008	sw $17,0x00000008($29)	17: sw $s1, 8($sp) # store the first argument
	0x0040002c	0xafb00004	sw $16,0x00000004($29)	18: sw $s0, 4($sp) # store the second arguement
	0x00400030	0xafbf0000	sw $31,0x00000000($29)	19: sw $ra, 0($sp) # store the return address
	0x00400034	0x00808020	add $16,$4,$0	21: add $s0, $a0, $zero # s0 = a0 (s0 = a)
	0x00400038	0x00a08820	add $17,$5,$0	22: add $s1, $a1, $zero # s0 = a0 (s1 = b)
	0x0040003c	0x0085001a	div $4,$5	25: div $a0, $a1 # divide a by b (remainder is stored in $mfhi)
	0x00400040	0x00008010	mfhi $16	26: mfhi $s0 # store remainder in $a1 (i.e. result of a%b)
	0x00400044	0xafb00004	sw $16,0x00000004($29)	27: sw $s0, 4($sp) #save the remainder
	0x00400048	0x16000003	bne $16,$0,0x00000003	28: bne $s0, $zero, L1 # branch to label if remainder != 0
	0x0040004c	0x00051020	add $2,$0,$5	31: add $v0, $zero, $a1 # $v0 = $a1 (i.e. return b)
	0x00400050	0x23bd000c	addi $29,$29,0x0000...	32: addi $sp, $sp 12 # adjust stack pointer to pop twice
	0x00400054	0x03e00008	jr $31	33: jr $ra # jump to return address (i.e. main)
	0x00400058	0x00a02020	add $4,$5,$0	36: add $a0, $a1, $zero # $a0 = $s1 (a = b)
	0x0040005c	0x8fb00004	lw $16,0x00000004($29)	37: lw $s0, 4($sp) # load remainder
	0x00400060	0x02002820	add $5,$16,$0	38: add $a1, $s0, $zero # $a1 = $s0 (b = remainder)
	0x00400064	0x0c100009	jal 0x00400024	40: jal gcd
	0x00400068	0x8fbf0000	lw $31,0x00000000($29)	43: lw $ra, 0($sp) # restore the return address
	0x0040006c	0x8fb00004	lw $16,0x00000004($29)	44: lw $s0, 4($sp) # restore the second argument
	0x00400070	0x8fb10008	lw $17,0x00000008($29)	45: lw $s1, 8($sp) # restore the first argument
	0x00400074	0x23bd000c	addi $29,$29,0x0000...	46: addi $sp, $sp, 12 # adjust stack pointer to pop twice
	0x00400078	0x03e00008	jr $31	47: jr $ra # jump to return address

Figure 38b: Text Segment showing all instructions

Main Call

Initially, the first instructions located at addresses 0x00400000 to 0x0040000C will simply load the arguments `a` and `b` into register `$a0` and `$a1` respectively which will contain 0x06 and 0x016 as shown in the register window below.

$a0	4	0x00000006
$a1	5	0x00000016

Figure 39a: Register window showing values of initial arguments

The memory location of these arguments are stored at addresses 0x10010000 and 0x10010004 in the Data Segment window.

Address	Value (+0)	Value (+4)
0x10010000	0x00000006	0x00000016

Figure 39b: Memory locations and values of arguments `a` and `b`

The next instruction at address 0x004000010 will perform a jump and link (`jal`). Upon executing this instruction, the value stored in register `$ra` is updated to now contain the *return address* (i.e. the instruction address after the `jal` instruction of line 10). We will need this return address to return back *this* location in the program after executing all the necessary function calls.

$fp	30	0x00000000
$ra	31	0x00400014
pc		0x00400024

Figure 39c: Register window showing the return address of `main` procedure

First Call

After jumping to the instruction at address 0x00400024, we execute the first instructions within the `gcd` procedure. The instructions 0x00400024 to 0x00400030 allocate 12 bytes of memory to stack which updates value of the stack pointer (`$sp`).

$gp	28	0x10008000
$sp	29	0x7fffeff0
$fp	30	0x00000000

Figure 40a: Register window showing the value of stack pointer (first call)

Next, the value in register `$ra` (i.e. the return address) is stored at address 0x7FFFEFF0 (i.e. 0x7FFFEFE0 + 0x10 offset). Likewise, the values in registers `$s0` and `$s1` are stored at address 0x7FFFEFF4 (i.e. 0x7FFFEFE0 + 0x14 offset) and 0x7FFFEFF8 (i.e. 0x7FFFEFE0 + 0x18 offset) respectively in the Data Segment.

Value (+10)	Value (+14)	Value (+18)
0x00400014	0x00000000	0x00000000

Figure 40b: Data Segment window showing memory locations and values of `$ra`, `$s0`, `$s1` (first call)

The next instructions at addresses 0x00400034 and 0x00400038 move the argument values 0x06 and 0x16 to registers `$s0` and `$s1` respectively.

$s0	16	0x00000006
$s1	17	0x00000016

Figure 40c: Register windows showing the updated contents of registers `$s0` and `$s1`

The instructions at addresses 0x0040003C to 0x00400044 will first divide the values in registers `$s0` and `$s1` (i.e. the argument values) whose remainder will be stored in register `$mfhi`.

pc		0x00400040
hi		0x00000006

Figure 40d: Register window showing remainder stored in `$hi`

Next, the remainder is copied to register `$s0`.

$t7	15	0x00000000
$s0	16	0x00000006
$s1	17	0x00000016

Figure 40e: Register window showing remainder in register `$s0`

This value is then stored at the memory address of `$s0` (0x7FFFEFF4).

Value (+14)
0x00000006

Figure 40f: Updated contents of memory value `$s0`

The instruction at address 0x00400048 will check (via the `bne` instruction) if the remainder (i.e. 0x06) stored at register `$s0` is equal to 0. Since it is not, we jump to execute the instructions at addresses 0x00400058 to 0x00400060. These instructions will set `$a0 = $s1` (i.e. `a = b`) and `$a1 = $s0` (i.e. b = `a%b`). These updated values will serve as the new arguments for the next function call.

$a0	4	0x00000016
$a1	5	0x00000006
$a2	6	0x00000000

Figure 40g: Updated contents of registers `$a0` and `$a1`

The next instruction to be executed (and the last in this first call) is at address 0x00400064 which will update the value in the `$ra` register to 0x00400068 (i.e. the instruction address after the jal instruction).

$fp	30	0x00000000
$ra	31	0x00400068
pc		0x00400024

Figure 40h: Storing the return address (first call)

Second Call

In the next function call, we again execute the instructions at addresses 0x00400024 to 0x00400030 which first allocate 12 bytes of memory to stack. This updates value of the stack pointer (`$sp`).

$gp	28	0x10008000
$sp	29	0x7fffefe4
$fp	30	0x00000000

Figure 41a: Register window showing the value of stack pointer (second call)

Next, the value in register `$ra` (i.e. the return address) is stored at address 0x7FFFEFE4 (i.e. 0x7FFFEFE0 + 0x04 offset). Likewise, the values in registers `$s0` and `$s1` are stored at address 0x7FFFEFE8 (i.e. 0x7FFFEFE0 + 0x08 offset) and 0x7FFFEFEC (i.e. 0x7FFFEFE0 + 0x0C offset) respectively in the Data Segment.

Value (+4)	Value (+8)	Value (+c)
0x00400068	0x00000006	0x00000016

Figure 41b: Data Segment window showing memory locations and values of `$ra`, `$s0`, `$s1` (second call)

The next instructions at addresses 0x00400034 and 0x00400038 move the updated argument values 0x16 and 0x06 to registers `$s0` and `$s1` respectively.

$s0	16	0x00000016
$s1	17	0x00000006

Figure 41c: Register windows showing the updated contents of registers `$s0` and `$s1`

The instructions at addresses 0x0040003C to 0x00400044 will first divide the values in registers `$s0` and `$s1` (i.e. the argument values) whose remainder will be stored in register `$mfhi`.

pc		0x00400040
hi		0x00000004
lo		0x00000003

Figure 41d: Remainder stored in register `$mfhi`

Next, the remainder is copied to register `$s0`.

$t7	15	0x00000000
$s0	16	0x00000004
$s1	17	0x00000006

Figure 41e: Register window showing remainder in register `$s0`

This value is then stored at the memory address of `$s0` (0x7FFFEFE8).

Value (+8)
0x00000004

Figure 41f: Updated contents of memory value `$s0`

The instruction at address 0x00400048 will check (via the `bne` instruction) if the remainder (i.e. 0x04) stored at register `$s0` is equal to 0. Since it is not, we jump to execute the instructions at addresses 0x00400058 to 0x00400060. These instructions will set `$a0 = $s1` (i.e. `a = b`) and `$a1 = $s0` (i.e. b = `a%b`). These updated values will serve as the new arguments for the next function call.

$a0	4	0x00000006
$a1	5	0x00000004
$a2	6	0x00000000

Figure 41g: Updated contents of registers `$a0` and `$a1`

The next instruction to be executed (and the last in this second call) is at address 0x00400064 which will update the value in the `$ra` register to 0x00400068 (i.e. the instruction address after the jal instruction).

$fp	30	0x00000000
$ra	31	0x00400068
pc		0x00400024

Figure 41h: Storing the return address (second call)

Third Call

In the next function call, we again execute the instructions at addresses 0x00400024 to 0x00400030 which first allocate 12 bytes of memory to stack. This updates value of the stack pointer (`$sp`).

$gp	28	0x10008000
$sp	29	0x7fffefd8
$fp	30	0x00000000

Figure 42a: Register window showing the value of stack pointer (Third call)

Next, the value in register `$ra` (i.e. the return address) is stored at address 0x7FFFEFD8 (i.e. 0x7FFFEFC0 + 0x18 offset). Likewise, the values in registers `$s0` and `$s1` are stored at address 0x7FFFEFDC (i.e. 0x7FFFEFC0 + 0x1C offset) and 0x7FFFEFE0 respectively in the Data Segment.

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefc0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00400068	0x00000004
0x7fffefe0	0x00000006	0x00400068	0x00000004	0x00000016	0x00400014	0x00000006	0x00000000	0x00000000

Figure 42b: Data Segment window showing memory locations and values of `$ra`, `$s0`, `$s1` (third call)

The next instructions at addresses 0x00400034 and 0x00400038 move the updated argument values 0x06 and 0x04 to registers `$s0` and `$s1` respectively.

$s0	16	0x00000006
$s1	17	0x00000004

Figure 42c: Register windows showing the updated contents of registers `$s0` and `$s1`

The instructions at addresses 0x0040003C to 0x00400044 will first divide the values in registers `$s0` and `$s1` (i.e. the argument values) whose remainder will be stored in register `$mfhi`.

pc		0x00400040
hi		0x00000002
lo		0x00000001

Figure 42d: Remainder stored in register `$mfhi`

Next, the remainder is copied to register `$s0`.

$t7	15	0x00000000
$s0	16	0x00000002
$s1	17	0x00000004

Figure 42e: Register window showing remainder in register `$s0`

This value is then stored at the memory address of `$s0` (0x7FFFEFDC).

Value (+1c)
0x00000002

Figure 42f: Updated contents of memory value `$s0`

The instruction at address 0x00400048 will check (via the `bne` instruction) if the remainder (i.e. 0x02) stored at register `$s0` is equal to 0. Since it is not, we jump to execute the instructions at addresses 0x00400058 to 0x00400060. These instructions will set `$a0 = $s1` (i.e. `a = b`) and `$a1 = $s0` (i.e. b = `a%b`). These updated values will serve as the new arguments for the next function call.

$a0	4	0x00000004
$a1	5	0x00000002
$a2	6	0x00000000

Figure 42g: Updated contents of registers `$a0` and `$a1`

The next instruction to be executed (and the last in this third call) is at address 0x00400064 which will update the value in the `$ra` register to 0x00400068 (i.e. the instruction address after the jal instruction).

$fp	30	0x00000000
$ra	31	0x00400068
pc		0x00400024

Figure 42h: Storing the return address (third call)

Fourth Call

In the next function call, we again execute the instructions at addresses 0x00400024 to 0x00400030 which first allocate 12 bytes of memory to stack. This updates value of the stack pointer (`$sp`).

$gp	28	0x10008000
$sp	29	0x7fffefcc
$fp	30	0x00000000

Figure 43a: Register window showing the value of stack pointer (Fourth call)

Next, the value in register `$ra` (i.e. the return address) is stored at address 0x7FFFEFCC (i.e. 0x7FFFEFC0 + 0x0C offset). Likewise, the values in registers `$s0` and `$s1` are stored at address 0x7FFFEFD0 (i.e. 0x7FFFEFC0 + 0x10 offset) and 0x7FFFEFD4 (i.e. 0x7FFFEFC0 + 0x14 offset) respectively in the Data Segment.

Value (+c)	Value (+10)	Value (+14)
0x00400068	0x00000002	0x00000004

Figure 43b: Data Segment window showing memory locations and values of `$ra`, `$s0`, `$s1` (fourth call)

The next instructions at addresses 0x00400034 and 0x00400038 move the updated argument values 0x04 and 0x02 to registers `$s0` and `$s1` respectively.

$s0	16	0x00000004
$s1	17	0x00000002
$s2	18	0x00000000

Figure 43c: Register windows showing the updated contents of registers `$s0` and `$s1`

The instructions at addresses 0x0040003C to 0x00400044 will first divide the values in registers `$s0` and `$s1` (i.e. the argument values) whose remainder will be stored in register `$mfhi`.

pc		0x00400040
hi		0x00000000
lo		0x00000002

Figure 43d: Remainder stored in register `$mfhi`

Next, the remainder is copied to register `$s0`.

$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000002

Figure 43e: Register window showing remainder in register `$s0`

This value is then stored at the memory address of `$s0` (0x7FFFEFD0).

Value (+10)
0x00000000

Figure 43f: Updated contents of memory value `$s0`

The instruction at address 0x00400048 will check (via the `bne` instruction) if the remainder (i.e. 0x00 stored at register `$s0` is equal to 0. It is and so no branching will occur. Instead, the next instruction to be executed is at address 0x0040004C which will load the value stored in register `$a1` (i.e. argument b which holds the remainder) to register `$v0`. This is the return value after all function calls.

$at	1	0x10010000
$v0	2	0x00000002
$v1	3	0x00000000

Figure 43g: Return value stored in register `$v0`

Return to Third call

The next instruction to be executed at address 0x00400050 will deallocate 12 bytes of memory and revert the stack pointer to its previous state as it was during the third call.

$gp	28	0x10008000
$sp	29	0x7fffefd8
$fp	30	0x00000000

Figure 44a: Register window showing the reverted state of stack pointer (return to third call)

Likewise, the instructions at addresses 0x00400068 to 0x00400070 restore the values of registers `$s0` and `$s1` as they were during the third call.

$s0	16	0x00000002
$s1	17	0x00000006
$s2	18	0x00000000

Figure 44b: Register window showing the reverted values of registers $s0 and $s1 (return to third call)

We then jump to the return address 0x00400068 which will bring us back into the second function all.

Return to second call

The next instruction to be executed at address 0x00400074 will deallocate 12 bytes of memory and revert the stack pointer to its previous state as it was during the second call.

$gp	28	0x10008000
$sp	29	0x7fffefe4
$fp	30	0x00000000

Figure 45a: Register window showing the reverted state of stack pointer (return to second call)

Likewise, the instructions at addresses 0x00400068 to 0x00400070 restore the values of registers `$s0` and `$s1` as they were during the second call.

$s0	16	0x00000004
$s1	17	0x00000016
$s2	18	0x00000000

Figure 45b: Register window showing the reverted values of registers $s0 and $s1 (return to second call)

We then jump to the return address 0x00400068 once again.

Return to first call

The next instruction to be executed at address 0x00400074 will deallocate 12 bytes of memory and revert the stack pointer to its previous state as it was during the first call.

$gp	28	0x10008000
$sp	29	0x7fffeff0
$fp	30	0x00000000

Figure 46a: Register window showing the reverted state of stack pointer (return to first call)

Likewise, the instructions at addresses 0x00400068 to 0x00400070 restore the values of registers `$s0` and `$s1` as they were during the second call.

$s0	16	0x00000006
$s1	17	0x00000000
$s2	18	0x00000000

Figure 46b: Register window showing the reverted values of registers $s0 and $s1 (return to second call)

Return to Main

The next instruction to be executed at address 0x00400074 will deallocate 12 bytes of memory and revert the stack pointer to its previous state as it was during the first call.

$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000

Figure 47a: Register window showing the reverted state of stack pointer (return to main)

We will then jump to the return address 0x00400014 which located in the main procedure. The instruction to be executed next is the instruction right after the `jal` instruction (where we left off). This instruction will simply save the value stored in register `$v0` (i.e. the gcd value) into memory location 0x10010008 (i.e. the location of local variable `gcd_res`.

Address	Value (+0)	Value (+4)	Value (+8)
0x10010000	0x00000006	0x00000016	0x00000002

Figure 47b: Memory location and value of local variable `gcd_res`

Intel X86 on MS Visual Studio

```
int main() {
004F1780  push        ebp
004F1781  mov         ebp,esp
004F1783  sub         esp,0CCh
004F1789  push        ebx
004F178A  push        esi
004F178B  push        edi
004F178C  lea         edi,[ebp-0CCh]
004F1792  mov         ecx,33h
004F1797  mov         eax,0CCCCCCCCh
004F179C  rep stos    dword ptr es:[edi]
004F179E  mov         ecx,offset _D11ECB56_gcd_recursive@cpp (04FC000h)    ≤ 1ms elapsed
004F17A3  call        @__CheckForDebuggerJustMyCode@4 (04F1208h)
    int gcd_res = gcd(6, 22);
004F17A8  push        16h
004F17AA  push        6
004F17AC  call        gcd (04F1104h)
004F17B1  add         esp,8
004F17B4  mov         dword ptr [gcd_res],eax
    return 0;
004F17B7  xor         eax,eax
}
004F17B9  pop         edi
004F17BA  pop         esi
004F17BB  pop         ebx
004F17BC  add         esp,0CCh
004F17C2  cmp         ebp,esp
004F17C4  call        __RTC_CheckEsp (04F1212h)
004F17C9  mov         esp,ebp
004F17CB  pop         ebp
004F17CC  ret
```

Figure 48: Assembly Code for main()

Main Call

The preliminary instructions (boxed in black) initializes a new stack frame with stack pointer value 0x0093FD18 and base pointer value 0x0093FD18.

```
Registers
 EAX = CCCCCCCC EBX = 007BB000
   ECX = 00000000 EDX = 004FA57C
   ESI = 004F1325 EDI = 0093FD18
   EIP = 004F179E ESP = 0093FC40
   EBP = 0093FD18 EFL = 00000212
```

Figure 49a: Initialization of stack frame (main call)

The next instructions (in orange) will first push the argument values 0x16 and 0x06 onto the stack in memory locations 0x00D3F7B8 and respectively before calling `gcd()`. In addition, the return address (i.e. the address after the call instruction) will be saved in the `EIP` register.

```
Memory 1

0x0093FC34  b1 17 4f 00
0x0093FC38  06 00 00 00
0x0093FC3C  16 00 00 00
```

Figure 49b: Memory locations of return address and initial argument values

First Call

When the call instruction is executed, we will jump to the function `gcd()` whose disassembly code is given below.

```
int gcd(int a, int b) {
004F16F0  push        ebp     ≤ 1ms elapsed
004F16F1  mov         ebp,esp
004F16F3  sub         esp,0C0h
004F16F9  push        ebx
004F16FA  push        esi
004F16FB  push        edi
004F16FC  lea         edi,[ebp-0C0h]
004F1702  mov         ecx,30h
004F1707  mov         eax,0CCCCCCCCh
004F170C  rep stos    dword ptr es:[edi]
004F170E  mov         ecx,offset _D11ECB56_gcd_recursive@cpp (04FC000h)
004F1713  call        @__CheckForDebuggerJustMyCode@4 (04F1208h)
    if (b == 0)
004F1718  cmp         dword ptr [b],0
004F171C  jne         gcd+35h (04F1725h)
        return a;
004F171E  mov         eax,dword ptr [a]
004F1721  jmp         gcd+49h (04F1739h)
    else
004F1723  jmp         gcd+49h (04F1739h)
        gcd(b, a%b);
004F1725  mov         eax,dword ptr [a]
004F1728  cdq
004F1729  idiv        eax,dword ptr [b]
004F172C  push        edx
004F172D  mov         eax,dword ptr [b]
004F1730  push        eax
004F1731  call        gcd (04F1104h)
004F1736  add         esp,8
}
```

Figure 50a: Disassembly of gcd()

The first set of instructions (in black) that will initialize a *new* stack frame with stack pointer and base pointer values 0x0093FB64 and 0x0093FC30 respectively.

```
Registers
 EAX = CCCCCCCC EBX = 007BB000
   ECX = 00000000 EDX = 00000001
   ESI = 004F1325 EDI = 0093FC30
   EIP = 004F170E ESP = 0093FB64
   EBP = 0093FC30 EFL = 00000202
```

Figure 50b: Initializing a new stack frame (first call)

The next instructions (in orange) will check if the second argument value (0x16) is equal to 0 and perform a jump if they are not equal. Since they are not equal, a jump is performed. The next sequence of instructions (in blue) are executed. First, the argument `a` is copied into register EAX and a division occurs between the `a` and `b`. The result is stored back into register EAX and remainder is stored in register EDX.

```
EDX = 00000006
```

Figure 50c: Remainder of a%b

After computing a%b, the remainder becomes the new argument for b while the previous argument for b is new argument for a. They will be used for the next function call. These new argument values are pushed onto the stack at memory locations 0x0093FB60 and 0x0093FB5C. Likewise, the call instruction is the last instruction in this first call and will push the return address 0x004F1736 into the EIP register at memory location 0x0093FB58 on the stack.

```
Memory 1

0x0093FB58  36 17 4f 00
0x0093FB5C  16 00 00 00
0x0093FB60  06 00 00 00
```

Figure 50d: Memory locations and values of return address and new argument values.

Second Call

We return back to the start of the gcd() function and once again execute the first set of instructions (in black) that will initialize a *new* stack frame with stack pointer and base pointer values 0x0093FB54 and 0x0093FA88 respectively.

```
Registers
EAX = CCCCCCCC EBX = 007BB000
  ECX = 00000000 EDX = 00000006
  ESI = 004F1325 EDI = 0093FB54
  EIP = 004F170E ESP = 0093FA88
  EBP = 0093FB54 EFL = 00000202
```

Figure 51a: Initializing a new stack frame (second call)

The next instructions (in orange) will check if the updated second argument (i.e. the value of the remainder computed in the previous call) (0x06) is equal to 0 and perform a jump if they are not equal. Since they are not equal, a jump is performed. The next sequence of instructions (in blue) are executed. First, the updated argument `a` is copied into register EAX and a division occurs between the `a` and `b`. The remainder is stored in register EDX.

```
EDX = 00000004
```

Figure 51b: Remainder of a%b (second call)

After computing a%b, the remainder becomes the new argument for b while the previous argument for b is new argument for a. They will be used for the next function call. These new argument values are pushed onto the stack at memory locations 0x0093FA80 and 0x0093FA84. Likewise, the call instruction is the last instruction in this first call and will push the return address 0x004F1736 into the EIP register at memory location 0x0093FA7C on the stack.

```
Memory 1

0x0093FA7C  36 17 4f 00
0x0093FA80  06 00 00 00
0x0093FA84  04 00 00 00
```

Figure 51c: Memory locations and values of return address and new argument values (second call).

Third Call

We return back to the start of the gcd() function and once again execute the first set of instructions (in black) that will initialize a *new* stack frame with stack pointer and base pointer values 0x0093F9AC and 0x0093FA78 respectively.

```
Registers
EAX = CCCCCCCC EBX = 007BB000
  ECX = 00000000 EDX = 00000004
  ESI = 004F1325 EDI = 0093FA78
  EIP = 004F170E ESP = 0093F9AC
  EBP = 0093FA78 EFL = 00000206
```

Figure 52a: Initializing a new stack frame (third call)

The next instructions (in orange) will check if the updated second argument (i.e. the value of the remainder computed in the previous call) (0x04) is equal to 0 and perform a jump if they are not equal. Since they are not equal, a jump is performed. The next sequence of instructions (in blue) are executed. First, the updated argument `a` is copied into register EAX and a division occurs between the `a` and `b`. The remainder is stored in register EDX.

```
EDX = 00000002
```

Figure 52b: Remainder of a%b (third call)

After computing a%b, the remainder becomes the new argument for b while the previous argument for b is new argument for a. They will be used for the next function call. These new argument values are pushed onto the stack at memory locations 0x0093F9A4 and 0x0093F9A8. Likewise, the call instruction is the last instruction in this first call and will push the return address 0x004F1736 into the EIP register at memory location 0x0093F9A0 on the stack.

```
Memory 1

0x0093F9A0  36 17 4f 00
0x0093F9A4  04 00 00 00
0x0093F9A8  02 00 00 00
```

Figure 52c: Memory locations and values of return address and new argument values (third call).

Fourth Call

We return back to the start of the gcd() function and once again execute the first set of instructions (in black) that will initialize a *new* stack frame with stack pointer and base pointer values 0x0093F8D0 and 0x0093F99C respectively.

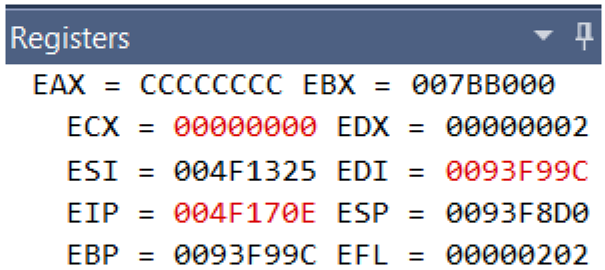

Figure 53a: Initializing a new stack frame (fourth call)

The next instructions (in orange) will check if the updated second argument (i.e. the value of the remainder computed in the previous call) (0x02) is equal to 0 and perform a jump if they are not equal. Since they are not equal, a jump is performed. The next sequence of instructions (in blue) are executed. First, the updated argument `a` is copied into register EAX and a division occurs between the `a` and `b`. The remainder is stored in register EDX.

Figure 53b: Remainder of a%b (fourth call)

After computing a%b, the remainder becomes the new argument for b while the previous argument for b is new argument for a. They will be used for the next function call. These new argument values are pushed onto the stack at memory locations 0x0093F8C8 and 0x0093F8CC. address 0x004F1736 into the EIP register at memory location 0x0093F8C4 on the stack.

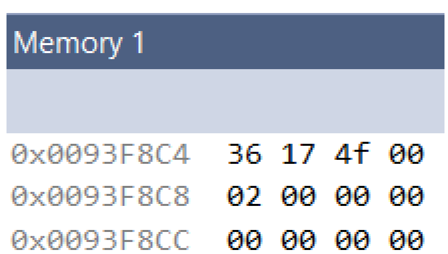

Figure 53c: Memory locations and values of return address and new argument values (fourth call).

Fifth Call

We return back to the start of the `gcd()` function and once again execute the first set of instructions (in black) that will initialize a *new* stack frame with stack pointer and base pointer values 0x0093F7F8 and 0x0093F8C0 respectively.

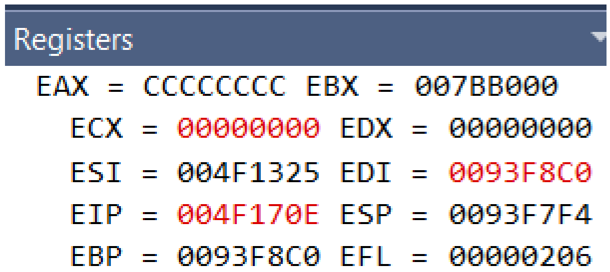

Figure 54a: Initializing a new stack frame (fifth call)

The next instructions (in orange) will check if the updated second argument (i.e. the value of the remainder computed in the previous call) (0x00) is equal to 0 and perform a jump if they are not equal. They are equal and so no jump will occur. Instead, the next instructions to be executed are those in red. These instructions will move the argument value of a (i.e. 0x02) into register EAX. Figure 54b below shows all stack frames used throughout the program where the base pointers, return addresses, first and second arguments are shown in red, yellow, blue, and purple respectively.

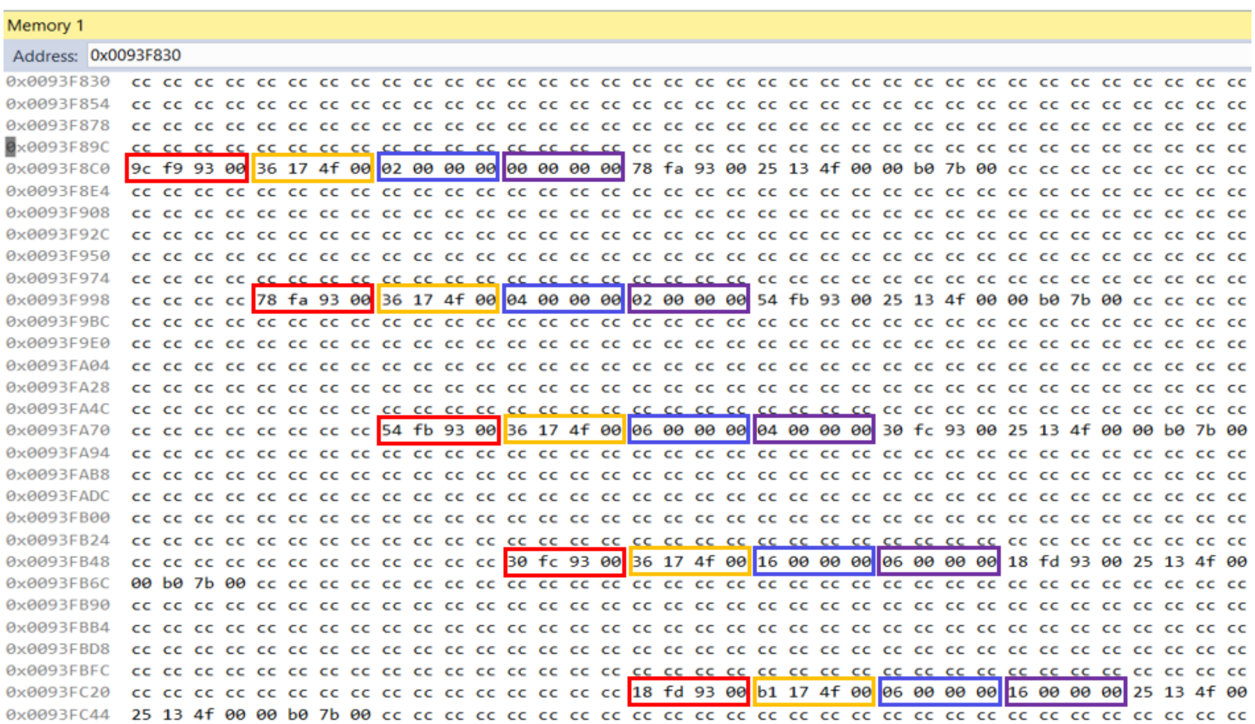

Figure 54b: Stack frames used throughout program

Return to Fourth Call

```
004F1739  pop         edi
004F173A  pop         esi
004F173B  pop         ebx
004F173C  add         esp,0C0h
004F1742  cmp         ebp,esp
004F1744  call        __RTC_CheckEsp (04F1212h)
004F1749  mov         esp,ebp
004F174B  pop         ebp
```

Figure 55a: Instructions to revert stack frame

A jump will then be performed to a set of instructions shown above that will revert the stack frame to its previous state as it was in the fourth function call.

```
Registers
EAX = 00000002 EBX = 007BB000
  ECX = 004FC000 EDX = 00000001
  ESI = 004F1325 EDI = 0093F99C
  EIP = 004F1739 ESP = 0093F8D0
  EBP = 0093F99C EFL = 00000212
```

Figure 55b: Reverted stack frame (return to fourth call)

Return to Third Call

A jump will then be performed to a set of instructions that will once again revert the stack frame to its previous state as it was in the third function call.

```
Registers
EAX = 00000002 EBX = 007BB000
  ECX = 004FC000 EDX = 00000001
  ESI = 004F1325 EDI = 0093FA78
  EIP = 004F1739 ESP = 0093F9AC
  EBP = 0093FA78 EFL = 00000206
```

Figure 56: Reverted stack frame (return to third call)

Return to Second Call

A jump will then be performed to a set of instructions that will once again revert the stack frame to its previous state as it was in the second function call.

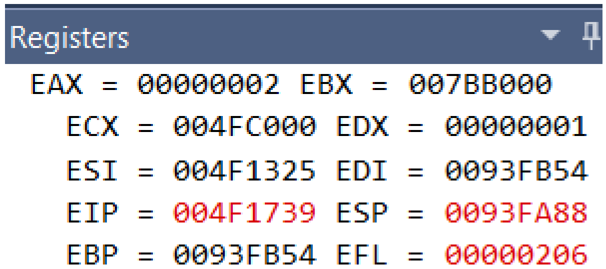

Figure 57: Reverted stack frame (return to second call)

Return to First Call

A jump will then be performed to a set of instructions that will once again revert the stack frame to its previous state as it was in the first function call.

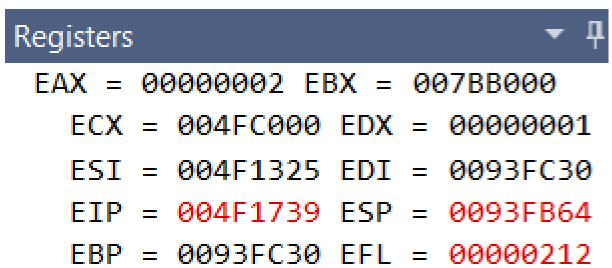

Figure 58: Reverted stack frame (return to first call)

Return to Main

Upon reverting all stack frames of all previous function calls, we revert the stack frame to its previous state as it was in main() before any function calls.

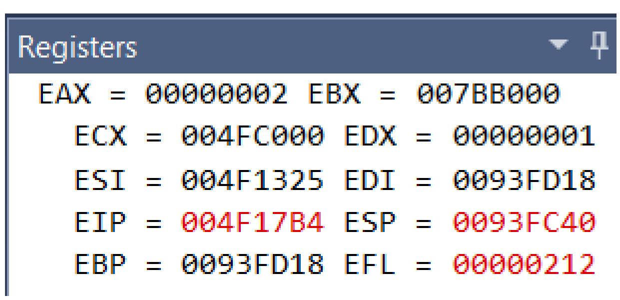

Figure 59: Reverted stack frame (return to main)

Lastly, we move the final return value (stored in register EAX) to local variable `gcd_res` and the program terminates.

Linux 64-Bit on Intel With GCC and GDB

Main Call

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555554628 <+0>:     push   %rbp
   0x0000555555554629 <+1>:     mov    %rsp,%rbp
   0x000055555555462c <+4>:     sub    $0x10,%rsp
=> 0x0000555555554630 <+8>:     mov    $0x16,%esi
   0x0000555555554635 <+13>:    mov    $0x6,%edi
   0x000055555555463a <+18>:    callq  0x5555555545fa <gcd>
   0x000055555555463f <+23>:    mov    %eax,-0x4(%rbp)
   0x0000555555554642 <+26>:    mov    $0x0,%eax
   0x0000555555554647 <+31>:    leaveq
   0x0000555555554648 <+32>:    retq
End of assembler dump.
```

Figure 60a: Disassembly code for main()

At the start of the `main()` function, the first set of instructions (in orange) are executed. A stack frame is initialized with initial base pointer (i.e. old based pointer) value 0x7FFFFFFFE0C0 (in white). The *new* base pointer pushed onto the stack has value 0x7FFFFFFFDFE0 (in magenta). A subtraction is performed on the stack pointer to allocate space (16 bytes) for variables. The stack pointer has value 0x7FFFFFFFDFD0 as a result (in green).

```
(gdb) x /2xg ($rsp)
0x7fffffffdfd0: 0x00007fffffffe0c0      0x0000000000000000
(gdb) print /x $rsp
$1 = 0x7fffffffdfd0
(gdb) x /2xg ($rbp)
0x7fffffffdfe0: 0x0000555555554650      0x00007ffff7a03bf7
(gdb) print /x $rbp
$2 = 0x7fffffffdfe0
```

Figure 60b: Stack frame for main()

The argument values 6 and 22 are also moved onto register `$edi` and `$esi` respectively.

```
(gdb) print /x $edi
$3 = 0x6
(gdb) print /x $esi
$4 = 0x16
```

Figure 60c: Argument values stored in registers `$edi` and `$esi`

The next instruction (in blue) performs the execution of the `callq` instruction which pushes the *return address* 0x55555555463F (i.e., the instruction after `callq`) onto the stack. This is the last operation executed in `main()` prior to jumping to `gcd()`.

```
(gdb) x /xg ($rsp)
0x7fffffffdfd8: 0x000055555555463f
```

Figure 60d: Pushing the Return Address to the stack

First Call

The assembler code for the `gcd()` function is given below.

```
Dump of assembler code for function gcd:
=> 0x00005555555545fa <+0>:     push   %rbp
   0x00005555555545fb <+1>:     mov    %rsp,%rbp
   0x00005555555545fe <+4>:     sub    $0x10,%rsp
   0x0000555555554602 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555554605 <+11>:    mov    %esi,-0x8(%rbp)
   0x0000555555554608 <+14>:    cmpl   $0x0,-0x8(%rbp)
   0x000055555555460c <+18>:    jne    0x555555554613 <gcd+25>
   0x000055555555460e <+20>:    mov    -0x4(%rbp),%eax
   0x0000555555554611 <+23>:    jmp    0x555555554626 <gcd+44>
   0x0000555555554613 <+25>:    mov    -0x4(%rbp),%eax
   0x0000555555554616 <+28>:    cltd
   0x0000555555554617 <+29>:    idivl  -0x8(%rbp)
   0x000055555555461a <+32>:    mov    -0x8(%rbp),%eax
   0x000055555555461d <+35>:    mov    %edx,%esi
   0x000055555555461f <+37>:    mov    %eax,%edi
   0x0000555555554621 <+39>:    callq  0x5555555545fa <gcd>
   0x0000555555554626 <+44>:    leaveq
   0x0000555555554627 <+45>:    retq
```

Figure 61a: Disassembly code for gcd()

At the beginning of this function call, the instructions (in orange) create a new stack frame as shown in figure 61b below. The *now* old base pointer 0x7FFFFFFFDFE0 (in white) is pushed onto the stack. The *new* base and stack pointers have values 0x7FFFFFFFDFC0 (in magenta) and 0x7FFFFFFFDFB0 (in green). Note that our argument values 0x06 and 0x16 (in lilac and purple) initialized in `main()` and previously stored in registers `$edi` and `$esi` are also pushed onto the stack.

```
(gdb) x /2xg ($rsp)
0x7fffffffdfb0: 0x00007ffff7de3b40      0x0000000600000016
(gdb) print /x $rsp
$5 = 0x7fffffffdfb0
(gdb) x /2xg ($rbp)
0x7fffffffdfc0: 0x00007fffffffdfe0      0x000055555555463f
(gdb) print /x $rbp
$6 = 0x7fffffffdfc0
```

Figure 61b: Stack frame (first call)

The next instructions in blue will compare whether or not our second argument (0x16) is equal to 0. If not, then a jump will occur from 0x55555555460C to 0x555555554613. After jumping to instruction at address 0x555555554613, the next instructions (in yellow) will first move the first argument value 0x06 to register `$eax`. Next, a modulo is performed with the value stored at location `$rbp` – 0x8 (i.e. the second argument 0x16). The result of this modulo is stored into register `$esi`. Likewise, the value stored in register `$edx` (i.e. 0x16) is copied to register `$edi`. These become the *new* argument values to be passed in the next function call.

```
(gdb) print /x $esi
$10 = 0x6
(gdb) print /x $edi
$11 = 0x16
```

Figure 61c: Register values of `$esi` and `$edi` after first call

The execution of `callq` will terminate this function call by pushing the return address 0x555555554626 (i.e., the instruction after `callq`) onto the stack pointer prior to pushing the base pointer 0x7FFFFFFFDFC0 (which occurs in the next call).

```
(gdb) x /xg ($rsp)
0x7fffffffdfa8: 0x0000555555554626
```

Figure 61d: Pushing the return address onto the stack (after first call)

Second Call

In the next call, we remain within `gcd()` and return to the set of instructions in orange. The old base pointer 0x7FFFFFFFDFC0 (in white) is pushed onto the stack. A *new* stack frame is created once again with base pointer 0x7FFFFFFFDFA0 (in magenta) and stack pointer 0x7FFFFFFDF90 (in green). Note that the updated argument value 0x16 and 0x06 (in lilac and purple) is also pushed onto the stack.

```
(gdb) x /2xg ($rsp)
0x7fffffffdf90: 0x00007fffffffdff8      0x0000001600000006
(gdb) print /x $rsp
$11 = 0x7fffffffdf90
(gdb) x /2xg ($rbp)
0x7fffffffdfa0: 0x00007fffffffdfc0      0x0000555555554626
(gdb) print /x $rbp
$12 = 0x7fffffffdfa0
```

Figure 62a: Stack Frame (Second Call)

```
(gdb) print /x $edi
$13 = 0x16
(gdb) print /x $esi
$14 = 0x6
```

Figure 62b: Argument values stored in registers `$edi` and `$esi` (Second Call)

The next instructions in blue will compare whether or not our second argument (0x06) is equal to 0. If not, then a jump will occur from 0x55555555460C to 0x555555554613. After jumping to instruction at address 0x555555554613, the next instructions (in yellow) will first move the first argument value 0x16 to register `$eax`. Next, a modulo is performed with the value stored at location `$rbp` – 0x8 (i.e. the second argument 0x06). The result of this modulo is stored into register `$esi`. Likewise, the value stored in register `$edx` (i.e. 0x06) is copied to register `$edi`. These become the *new* argument values to be passed in the next function call.

```
(gdb) print /x $edi
$15 = 0x6
(gdb) print /x $esi
$16 = 0x4
```

Figure 62c: Register values of `$edi` and `$esi` after second call

The execution of `callq` will terminate this function call by pushing the return address 0x555555554626 (i.e., the instruction after `callq`) onto the stack pointer prior to pushing the base pointer 0x7FFFFFFFDFA0 (which occurs in the next call).

```
(gdb) x /xg ($rsp)
0x7fffffffdfa8: 0x0000555555554626
```

Figure 62d: Pushing the return address onto the stack (after second call)

Third Call

In the next call, we remain within `gcd()` and return to the set of instructions in orange. The old base pointer 0x7FFFFFFFDFA0 (in white) is pushed onto the stack. A *new* stack frame is created once again with base pointer 0x7FFFFFFFDF80 (in magenta) and stack pointer 0x7FFFFFFDF70 (in green). Note that the updated argument value 0x06 and 0x04 (in lilac and purple) is also pushed onto the stack.

```
(gdb) x /2xg ($rsp)
0x7fffffffdf70: 0x0000000000000000      0x0000000600000004
(gdb) print /x $rsp
$17 = 0x7fffffffdf70
(gdb) x /2xg ($rbp)
0x7fffffffdf80: 0x00007fffffffdfa0      0x0000555555554626
(gdb) print /x $rbp
$18 = 0x7fffffffdf80
```

Figure 63a: Stack Frame (Third Call)

```
(gdb) print /x $edi
$19 = 0x6
(gdb) print /x $esi
$20 = 0x4
```

Figure 63b: Argument values stored in registers `$edi` and `$esi` (Third Call)

The next instructions in blue will compare whether or not our second argument (0x04) is equal to 0. If not, then a jump will occur from 0x55555555460C to 0x555555554613. After jumping to instruction at address 0x555555554613, the next instructions (in yellow) will first move the first argument value 0x06 to register `$eax`. Next, a modulo is performed with the value stored at location `$rbp` – 0x8 (i.e. the second argument 0x04). The result of this modulo is stored into register `$esi`. Likewise, the value stored in register `$edx` (i.e. 0x04) is copied to register `$edi`. These become the *new* argument values to be passed in the next function call.

```
(gdb) print /x $edi
$21 = 0x4
(gdb) print /x $esi
$22 = 0x2
```

Figure 63c: Register values of `$edi` and `$esi` after third call

The execution of `callq` will terminate this function call by pushing the return address 0x555555554626 (i.e., the instruction after `callq`) onto the stack pointer prior to pushing the base pointer 0x7FFFFFFFDF80 (which occurs in the next call).

```
(gdb) x /xg ($rsp)
0x7fffffffdfa8: 0x0000555555554626
```

Figure 63d: Pushing the return address onto the stack (after third call)

Fourth Call

In the next call, we remain within `gcd()` and return to the set of instructions in orange. The old base pointer 0x7FFFFFFFDF80 (in white) is pushed onto the stack. A *new* stack frame is created once again with base pointer 0x7FFFFFFFDF60 (in magenta) and stack pointer 0x7FFFFFFDF50 (in green). Note that the updated argument value 0x06 and 0x04 (in lilac and purple) is also pushed onto the stack.

```
(gdb) x /2xg ($rsp)
0x7fffffffdf50: 0x00007ffff7ffb2a8      0x0000000400000002
(gdb) print /x $rsp
$23 = 0x7fffffffdf50
(gdb) x /2xg ($rbp)
0x7fffffffdf60: 0x00007fffffffdf80      0x0000555555554626
(gdb) print /x $rbp
$24 = 0x7fffffffdf60
```

Figure 64a: Stack Frame (Fourth Call)

```
(gdb) print /x $edi
$25 = 0x4
(gdb) print /x $esi
$26 = 0x2
```

Figure 64b: Argument values stored in registers `$edi` and `$esi` (Fourth Call)

The next instructions in blue will compare whether or not our second argument (0x02) is equal to 0. If not, then a jump will occur from 0x55555555460C to 0x555555554613. After jumping to instruction at address 0x555555554613, the next instructions (in yellow) will first move the first argument value 0x04 to register `$eax`. Next, a modulo is performed with the value stored at location `$rbp` – 0x8 (i.e. the second argument 0x02). The result of this modulo is stored into register `$esi`. Likewise, the value stored in register `$edx` (i.e. 0x02) is copied to register `$edi`. These become the *new* argument values to be passed in the next function call.

```
(gdb) print /x $edi
$27 = 0x2
(gdb) print /x $esi
$28 = 0x0
```

Figure 64c: Register values of `$edi` and `$esi` after Fourth call

The execution of `callq` will terminate this function call by pushing the return address 0x555555554626 (i.e., the instruction after `callq`) onto the stack pointer prior to pushing the base pointer 0x7FFFFFFFDF60 (which occurs in the next call).

```
(gdb) x /xg ($rsp)
0x7fffffffdfa8: 0x0000555555554626
```

Figure 64d: Pushing the return address onto the stack (after Fourth call)

Fifth Call

In the next call, we remain within `gcd()` and return to the set of instructions in orange. The old base pointer 0x7FFFFFFFDF60 (in white) is pushed onto the stack. A *new* stack frame is created once again with base pointer 0x7FFFFFFFDF40 (in magenta) and stack pointer 0x7FFFFFFDF30 (in green). Note that the updated argument value 0x06 and 0x04 (in lilac and purple) is also pushed onto the stack.

```
(gdb) x /2xg ($rsp)
0x7fffffffdf30: 0x0000000000000000      0x0000000200000000
(gdb) print /x $rsp
$29 = 0x7fffffffdf30
(gdb) x /2xg ($rbp)
0x7fffffffdf40: 0x00007fffffffdf60      0x0000555555554626
(gdb) print /x $rbp
$30 = 0x7fffffffdf40
```

Figure 65a: Stack Frame (Fifth Call)

```
$30 = 0x7fffffffdf40
(gdb) print /x $edi
$31 = 0x2
(gdb) print /x $esi
$32 = 0x0
```

Figure 65b: Argument values stored in registers `$edi` and `$esi` (Fifth Call)

The next instructions in blue will compare whether or not our second argument (0x00) is equal to 0. We meet this condition and the next instructions (in red) are executed by first copying the value of the first argument (i.e. 0x02) into register `$eax`. Next, a jump will occur to the instruction located at address 0x555555554626. Figure 65c below shows all stack frames that were used in the program. The stack pointers, base pointers, return addresses, and argument values are shown in green, magenta, amber, and purple respectively.

```
(gdb) x /22xg ($rsp)
0x7fffffffdf30: 0x0000000000000000      0x0000000200000000
0x7fffffffdf40: 0x00007fffffffdf60      0x0000555555554626
0x7fffffffdf50: 0x00007ffff7ffb2a8      0x0000000400000002
0x7fffffffdf60: 0x00007fffffffdf80      0x0000555555554626
0x7fffffffdf70: 0x0000000000000000      0x0000000600000004
0x7fffffffdf80: 0x00007fffffffdfa0      0x0000555555554626
0x7fffffffdf90: 0x00007fffffffdff8      0x0000001600000006
0x7fffffffdfa0: 0x00007fffffffdfc0      0x0000555555554626
0x7fffffffdfb0: 0x00007ffff7de3b40      0x0000000600000016
0x7fffffffdfc0: 0x00007fffffffdfe0      0x000055555555463f
0x7fffffffdfd0: 0x00007fffffffe0c0      0x0000000000000000
```

Figure 65c: Stack Frames used throughout program

The instruction at addresses 0x555555554626 and 0x555555554627 will terminate the current function call and return to the return address 0x555555554626.

Return to Fourth Call

We return to the fourth call and revert the stack frame as it was in this stage.

```
(gdb) x /2xg ($rsp)
0x7fffffffdf50: 0x00007ffff7ffb2a8      0x0000000400000002
(gdb) print /x $rsp
$33 = 0x7fffffffdf50
(gdb) x /2xg ($rbp)
0x7fffffffdf60: 0x00007fffffffdf80      0x0000555555554626
(gdb) print /x $rbp
$34 = 0x7fffffffdf60
```

Figure 66: Reverted Stack Frame (Return to Fourth Call)

No additional instructions are required and so we exit this current function call and return to back to the return address 0x555555554626 once again.

Return to Third Call

We return to the third call and revert the stack frame as it was in this stage.

```
(gdb) x /2xg ($rsp)
0x7fffffffdf70: 0x0000000000000000      0x0000000600000004
(gdb) print /x $rsp
$35 = 0x7fffffffdf70
(gdb) x /2xg ($rbp)
0x7fffffffdf80: 0x00007fffffffdfa0      0x0000555555554626
(gdb) print /x $rbp
$36 = 0x7fffffffdf80
```

Figure 67: Reverted Stack Frame (Return to Third Call)

No additional instructions are required and so we exit this current function call and return to back to the return address 0x555555554626 once again.

Return to Second Call

We return to the second call and revert the stack frame as it was in this stage.

```
(gdb) x /2xg ($rsp)
0x7fffffffdf90: 0x00007fffffffdff8      0x0000001600000006
(gdb) print /x $rsp
$37 = 0x7fffffffdf90
(gdb) x /2xg ($rbp)
0x7fffffffdfa0: 0x00007fffffffdfc0      0x0000555555554626
(gdb) print /x $rbp
$38 = 0x7fffffffdfa0
```

Figure 68: Reverted Stack Frame (Return to second Call)

No additional instructions are required and so we exit this current function call and return to back to the return address 0x555555554626 once again.

Return to First Call

We return to the first call and revert the stack frame as it was in this stage.

```
(gdb) x /2xg ($rsp)
0x7fffffffdfb0: 0x00007ffff7de3b40      0x0000000600000016
(gdb) print /x $rsp
$39 = 0x7fffffffdfb0
(gdb) x /2xg ($rbp)
0x7fffffffdfc0: 0x00007fffffffdfe0      0x000055555555463f
(gdb) print /x $rbp
$40 = 0x7fffffffdfc0
```

Figure 69: Reverted Stack Frame (Return to first Call)

No additional instructions are required and so we exit this current function call and return to back to the return address 0x55555555463f.

Return to Main

We return back to main() after completing all function calls.

```
Dump of assembler code for function main:
   0x0000555555554628 <+0>:     push   %rbp
   0x0000555555554629 <+1>:     mov    %rsp,%rbp
   0x000055555555462c <+4>:     sub    $0x10,%rsp
   0x0000555555554630 <+8>:     mov    $0x16,%esi
   0x0000555555554635 <+13>:    mov    $0x6,%edi
   0x000055555555463a <+18>:    callq  0x5555555545fa <gcd>
=> 0x000055555555463f <+23>:    mov    %eax,-0x4(%rbp)
   0x0000555555554642 <+26>:    mov    $0x0,%eax
   0x0000555555554647 <+31>:    leaveq
   0x0000555555554648 <+32>:    retq
```

Figure 70a: Return to main()

The stack frame is reverted as it was in this stage.

```
(gdb) x /2xg ($rsp)
0x7fffffffdfd0: 0x00007fffffffe0c0      0x0000000000000000
(gdb) print /x $rsp
$41 = 0x7fffffffdfd0
(gdb) x /2xg ($rbp)
0x7fffffffdfe0: 0x0000555555554650      0x00007ffff7a03bf7
(gdb) print /x $rbp
$42 = 0x7fffffffdfe0
```

Figure 70b: Reverted Stack Frame (Return to main())

The last crucial instruction at address 0x55555555463f will move the value stored register `$eax` (i.e., the return value of `gcd(6,22)`) to the location of local variable `gcd_res` on the stack.

```
(gdb) x $rbp - 0x4
0x7fffffffdfdc: 0x00000002
(gdb) x &gcd_res
0x7fffffffdfdc: 0x00000002
```

Figure 70c: Location and value of local variable `gcd_res`