Recursion in Assembly

By Alexander Wang

42 Silicon Valley™

Recursive Functions:

"To understand recursion, one must understand recursion" – Stephen Hawking

Now for some fun times for those of you who haven't taken an "advanced" math class. This will be absolutely pure madness to understand but don't worry it'll make your brain hurt less and help you in your college-level discrete mathematics class (or equivalent computer science class).

Recursion quite simply is a function that calls itself and that's it. What is so great about it? It allows for some elegant code (at least in higher-level languages like Python). However, there are some serious downsides:

- Slower than the equivalent iterative solution (function that is called only once).
- Might crash your program

Here are some natural recursive functions that might help:

- Summation of n numbers: $\sum_{i=0}^{n} i$ basically, i starts at 0 and gets added until i = n (the recursion here is the addition part).
- Factorial:

$$fact(n) = \begin{cases} 1\ if\ n = 0 \\ n * fact(n-1) if\ n > 0 \end{cases}$$

Note that the two examples above have a natural stopping point. This stopping point is called a *base case*. Without a base case the program will go on forever and eventually crash the program.

We will do the first as an example of recursion. But before we get to the code, lets plan it out a little bit, as should be obvious:

To reiterate we are going to implement the following function:

$$\sum_{i=0}^{N} i$$

$$f(N) = \begin{cases} N & N \le 1 \\ N + f(N-1) & otherwise \end{cases}$$

So, let's do a quick rundown of what we need to do:

1. Examine the number.
2. Is the number less than or equal to 1?
3. If so the answer is the number.
4. Otherwise: the answer is the number added the addition of the number minus 1.

This would be slightly painful if we didn't have local variables. While in other programs we could have stored the value as a global variable, but global variables only allow one copy of each variable. In this program we will have multiple copies of the same function running at the same time, all of them need their own copy of data. Since local variables are created with each function call we are good to go.

Let's see this program in play:

```
#Purpose: This program finds the summation of numbers up to 5.
#Everything is stored in registers, but for completeness; keep data sect.
.section data

.section .text
.globl _start
.globl summation #unneeded unless we want to share the function with others
_start:
     pushl $5 #Summation takes one parameter so we will push it on
     call summation #run the summation function
     addl $4, %esp #fix the stack
     movl %eax, %ebx  #return value in %eax, move to %ebx
     movl $1, %eax #call the kernel's exit function
     int 0x80

#This is the actual function definition.
.type summation, @function

```

```
19 summation:
20     pushl %ebp #standard stuff to restore %ebp after the function
21     movl %esp, %ebp #don’t want to modify the stack pointer; use %ebp
22     movl 8(%ebp), %eax #Since this only has one parameter
23     cmp $1, %eax #if the number is less or equal 1 then
24     jle end_sum #our base case and we return.
25     decl %eax #otherwise decrease the value by 1.
26     push %eax #push it on for our next call to summation.
27     call summation
28     movl 8(%ebp), %ebx #%eax has our return value, so we reload the value
29                        #into %ebx
30     addl %ebx, %eax #add result of last call to sum (in %eax), the answer
31                     #is stored in %eax, which is good since that’s where
32                     # our return values are.
33     end_sum:
34        movl %ebp, %esp #standard function return stuff.
35        popl %ebp #have to restore %ebp to where they were before function
36        ret
```

Assemble, link and run it with these commands:

```
as summation.s –o summation.o
ld summation.o –o summation
./summation
echo $?
```

The output you should have gotten here is 15. Since 5 + 4 + 3 + 2 + 1 = 15. Presumably you don’t understand all of it so let’s begin the explanation:

In label: “_start” we are pushing 5 onto the stack. When programming functions, you are supposed to put the parameters of the function on the stack right before you call the function. In this case, summation only takes one parameter – the number you want to sum too. The call instruction will then make the function call.

Lines 11-14 are our next lines of code to look at. They take place after the call to summation. Now we have to clean up the stack. The addl will move the stack pointer back to where it was before we pushed on the parameter. It's a good habit to always clean up your stack after every function call unless you have a *very good* reason not to.

So some nice things about function calls:

- Other programmers don't have to know anything about them except its arguments to use them.
- Provided standard blocks by which you can build a program.
- Can be called multiple times from any location and they will always know where to go back to because of the *call* instruction.

Now let's go to our actual function called summation (lines 17-36). The .type directive tells the linker that *summation* is a function. This isn't needed, as mentioned previously, unless we want other programs to be able to use it.

The first real instructions (20-21) creates a stack frame for the function. These two lines is how you should start every function.

Line 22 uses base pointer addressing, and since we know the parameter is in 8(%ebp) we will directly address it. Remember that (%ebp) has the old (%ebp), 4(%ebp) is the return address and 8(%ebp) is the location of the first parameter to the function. Since this function calls itself it will have other values too.

Now, next we check if we hit our base case (a parameter of 1), if so we jump to the instruction at the label **end_sum**, where it will be returned. The answer is already in %eax which we mentioned is where you put return values.

Otherwise we would call summation again with our parameter minus 1. So first we decrease %eax by one and after that we push it back onto the stack and call summation again (Lines 25-27).

Okay since, we just came back from summation, we don't know what our registers will look like so we will pull off our original parameter again from the stack (Line 28). And now we add them together (Line 29). And now we return to the start until the base case is satisfied.

We do the classic clearing of the stack, restoring %ebp and %esp and then finally call ret. The function is done and we now have our answer.

Appendix A Assembly Instructions:

[NOTE: SOME OF THESE YOU WILL NOT NEED UNTIL LATER ON]

ADD (Addition)
Syntax: ADD destination, source

The ADD instruction adds a value to a register or a memory address. It can be used in these ways:

These instruction can set the Z-Flag, the O-Flag and the C-Flag (and some others, which are not needed for cracking).

AND (Logical And)
Syntax: AND destination, source

The AND instruction uses a logical AND on two values.
This instruction *will* clear the O-Flag and the C-Flag and can set the Z-Flag.
To understand AND better, consider those two binary values:

```
1001010110
0101001101
```

If you AND them, the result is 0001000100
When two 1 stand below each other, the result is of this bit is 1, if not: The result is 0. You can use calc.exe to calculate AND easily.

CALL (Call)
Syntax: CALL something

The instruction CALL pushes the RVA (Relative Virtual Address) of the instruction that follows the CALL to the stack and calls a sub program/procedure.

CALL can be used in the following ways:

```
    CALL     404000                  ;; MOST COMMON: CALL ADDRESS
    CALL     EAX                     ;; CALL REGISTER - IF EAX WOULD BE 404000 IT
WOULD BE SAME AS THE ONE ABOVE
    CALL     DWORD PTR [EAX]         ;; CALLS THE ADDRESS THAT IS STORED AT
[EAX]
    CALL     DWORD PTR [EAX+5]              ;; CALLS THE ADDRESS THAT IS STORED
AT [EAX+5]
```

CDQ (Convert DWord (4Byte) to QWord (8 Byte))
Syntax: CQD

CDQ is an instruction that always confuses newbies when it appears first time. It is mostly used in front of divisions and does nothing else then setting all bytes of EDX to the value of the highest bit of EAX. (That is: if EAX <80000000, then EDX will be 00000000; if EAX >= 80000000, EDX will be FFFFFFFF).

CMP (Compare)
Syntax: CMP dest, source

The CMP instruction compares two things and can set the C/O/Z flags if the result fits.

```
    CMP         EAX, EBX                        ;; compares eax and ebx and sets z-flag if they
are equal
    CMP         EAX,[404000]                    ;; compares eax with the dword at 404000
    CMP         [404000],EAX                    ;; compares eax with the dword at 404000
```

DEC (Decrement)
Syntax: DEC something

dec is used to decrease a value (that is: value=value-1)

dec can be used in the following ways:

```
    dec eax                                     ;; decrease eax
    dec [eax]                                   ;; decrease the dword that is stored at [eax]
    dec [401000]                                ;; decrease the dword that is stored at [401000]
    dec [eax+401000]                            ;; decrease the dword that is stored at
[eax+401000]
```

The dec instruction can set the Z/O flags if the result fits.

DIV (Division)
Syntax: DIV divisor

DIV is used to divide EAX through divisor (unsigned division). The dividend is always EAX, the result is stored in EAX, the modulo-value in EDX.

An example:

```
    mov eax,64                                  ;; EAX = 64h = 100
    mov ecx,9                           ;; ECX = 9
    div ecx                                     ;; DIVIDE EAX THROUGH ECX
```

After the division EAX = 100/9 = 0B and ECX = 100 MOD 9 = 1

The div instruction can set the C/O/Z flags if the result fits.

IDIV (Integer Division)
Syntax: IDIV divisor

The IDIV works in the same way as DIV, but IDIV is a signed division.
The idiv instruction can set the C/O/Z flags if the result fits.

IMUL (Integer Multiplication)
Syntax: IMUL value
IMUL dest,value,value
IMUL dest,value

IMUL multiplies either EAX with value (IMUL value) or it multiplies two values and puts them into a destination register (IMUL dest, value, value) or it multiplies a register with a value (IMUL dest, value).

If the multiplication result is too big to fit into the destination register, the O/C flags are set. The Z flag can be set, too.

INC (Increment)
Syntax: INC register

INC is the opposite of the DEC instruction; it increases values by 1.
INC can set the Z/O flags.

INT
Syntax: int dest

Generates a call to an interrupt handler. The dest value must be an integer (e.g., Int 21h).
INT3 and INTO are interrupt calls that take no parameters but call the handlers for interrupts 3 and 4, respectively.

JUMPS
These are the most important jumps and the condition that needs to be met, so that they'll be executed (Important jumps are marked with * and very important with **):

JA*	-	Jump if (unsigned) above	- CF=0 and ZF=0
JAE	-	Jump if (unsigned) above or equal	- CF=0
JB*	-	Jump if (unsigned) below	- CF=1
JBE	-	Jump if (unsigned) below or equal	- CF=1 or ZF=1
JC	-	Jump if carry flag set	- CF=1
JCXZ	-	Jump if CX is 0	- CX=0
JE**	-	Jump if equal	- ZF=1
JECXZ	-	Jump if ECX is 0	- ECX=0

JG*	-	Jump if (signed) greater	- ZF=0 and SF=OF (SF = Sign Flag)
JGE*	-	Jump if (signed) greater or equal	- SF=OF
JL*	-	Jump if (signed) less	- SF != OF (!= is not)
JLE*	-	Jump if (signed) less or equal	- ZF=1 and OF != OF
JMP**	-	Jump	- Jumps always
JNA	-	Jump if (unsigned) not above	- CF=1 or ZF=1
JNAE	-	Jump if (unsigned) not above or equal	- CF=1
JNB	-	Jump if (unsigned) not below	- CF=0
JNBE	-	Jump if (unsigned) not below or equal	- CF=0 and ZF=0
JNC	-	Jump if carry flag not set	- CF=0
JNE**	-	Jump if not equal	- ZF=0
JNG	-	Jump if (signed) not greater	- ZF=1 or SF!=OF
JNGE	-	Jump if (signed) not greater or equal	- SF!=OF
JNL	-	Jump if (signed) not less	- SF=OF
JNLE	-	Jump if (signed) not less or equal	- ZF=0 and SF=OF
JNO	-	Jump if overflow flag not set	- OF=0
JNP	-	Jump if parity flag not set	- PF=0
JNS	-	Jump if sign flag not set	- SF=0
JNZ	-	Jump if not zero	- ZF=0
JO	-	Jump if overflow flag is set	- OF=1
JP	-	Jump if parity flag set	- PF=1
JPE	-	Jump if parity is equal	- PF=1
JPO	-	Jump if parity is odd	- PF=0
JS	-	Jump if sign flag is set	- SF=1
JZ	-	Jump if zero	- ZF=1

LEA (Load Effective Address)
Syntax: LEA dest,src

LEA can be treated the same way as the MOV instruction. It isn't used too much for its original function, but more for quick multiplications like this:

```
lea eax, dword ptr [4*ecx+ebx]
```
which gives eax the value of 4*ecx+ebx

MOV (Move)
Syntax: MOV dest,src

This is an easy to understand instruction. MOV copies the value from src to dest and src stays what it was before.

There are some variants of MOV:

MOVS/MOVSB/MOVSW/MOVSD EDI, ESI: Those variants copy the byte/word/dword ESI points to,
to the space EDI points to.

MOVSX: MOVSX expands Byte or Word operands to Word or Dword size and keeps the sign of the
value.

MOVZX: MOVZX expands Byte or Word operands to Word or Dword size and fills the rest of the
space with 0.

MUL (Multiplication)
Syntax: MUL value

This instruction is the same as IMUL, except that it multiplies unsigned. It can set the O/Z/F flags.

NOP (No Operation)
Syntax: NOP

This instruction does absolutely nothing
That's the reason why it is used so often in reversing ;)

OR (Logical Inclusive Or)
Syntax: OR dest,src

The OR instruction connects two values using the logical inclusive or.
This instruction clears the O-Flag and the C-Flag and can set the Z-Flag.

To understand OR better, consider those two binary values:

```
1001010110
0101001101
```

If you OR them, the result is 1101011111

Only when there are two 0 on top of each other, the resulting bit is 0. Else the resulting bit is 1. You can use calc.exe to calculate OR. I hope you understand why, else write down a value on paper and try ;)

POP
Syntax: POP dest

POP loads the value of byte/word/dword ptr [esp] and puts it into dest. Additionally it increases the stack by the size of the value that was popped of the stack, so that the next
POP would get the next value.

PUSH
Syntax: PUSH operand

PUSH is the opposite of POP. It stores a value on the stack and decreases it by the size of the operand that was pushed, so that ESP points to the value that was PUSHed.

REP/REPE/REPZ/REPNE/REPNZ
Syntax: REP/REPE/REPZ/REPNE/REPNZ *ins*

Repeat Following String Instruction: Repeats ins until CX=0 or until indicated condition (ZF=1, ZF=1, ZF=0, ZF=0) is met. The ins value must be a string operation such as CMPS, INS,
LODS, MOVS, OUTS, SCAS, or STOS.

RET (Return)
Syntax: RET
RET digit

RET does nothing but return from a part of code that was reached using a CALL instruction.
RET digit cleans the stack before it returns.

SUB (Subtraction)
Syntax: SUB dest,src

SUB is the opposite of the ADD command. It subtracts the value of src from the value of dest and stores the result in dest.

SUB can set the Z/O/C flags.

TEST
Syntax: TEST operand1, operand2

This instruction is in 99% of all cases used for "TEST EAX, EAX". It performs a Logical AND(AND instruction) but does not save the values. It only sets the Z-Flag, when EAX is 0
or clears it, when EAX is not 0. The O/C flags are always cleared.

XOR
Syntax: XOR dest,src

The XOR instruction connects two values using logical exclusive OR (remember OR uses inclusive OR).

This instruction clears the O-Flag and the C-Flag and can set the Z-Flag.
To understand XOR better, consider those two binary values:

```
1001010110
0101001101
```

If you OR them, the result is 1100011011

When two bits on top of each other are equal, the resulting bit is 0. Else the resulting bit is 1. You can use calc.exe to calculate XOR.
The most often seen use of XOR is “XOR, EAX, EAX”. This will set EAX to 0, because when
you XOR a value with itself, the result is always 0. I hope you understand why, else write down a value on paper and try ;)