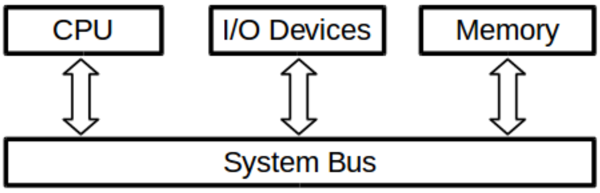
Notes for "Assembly primer for hackers"

Source:

www.securitytube.net
https://www.youtube.com/watch?v=K0g-twyhmQ4

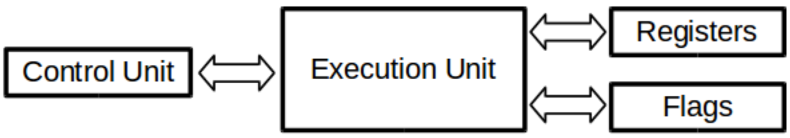
written by avr4L

1 Part 1: System Organization



The CPU, Memory/RAM and I/O Devices such as your keyboard or monitor communicate via the system bus.

1.1 CPU



The CPU consists of four components.

- Control Unit: loads and decodes instructions from RAM in the Execution unit, stores data in RAM
- Execution Unit: executes instructions
- Registers: internal memory for the execution unit to execute calculations / internal variables of the CPU
- Flags depending on events flags/binary variables get set. e.g. the zero flag if any calculation as a zero as return

CPU Registers

There are four different kind of CPU registers

General purpose registers

EAX EBX ECX EDX ESI EDI ESP EBP

Segment Registers

CS DS SS ES FS GS

Control Registers

CR0 CR1 CR2 CR3 CR4

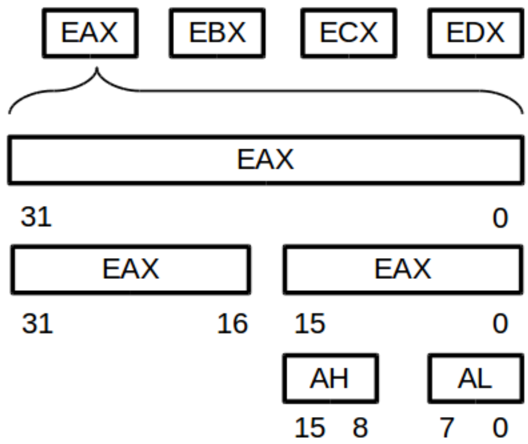
Instruction Pointer Register

EIP

General Purpose Registers

- EAX: Accumulator Register, stores operands and results
- EBX: Base Register, stores pointers to Data
- ECX: Counter Register, stores a counter for loops and strings
- EDX: Data Register, I/O Pointer
- ESI, EDI: Source Index, Destination Index, store pointers for memory operations
- ESP: Stack Pointer, points to the top of the stack
- EBP: Stack Base pointer, points the the base of a stack frame

In 32 bit architecture the general purpose registers store 32 bits. Specific portions of those bits can be accessed the following way.

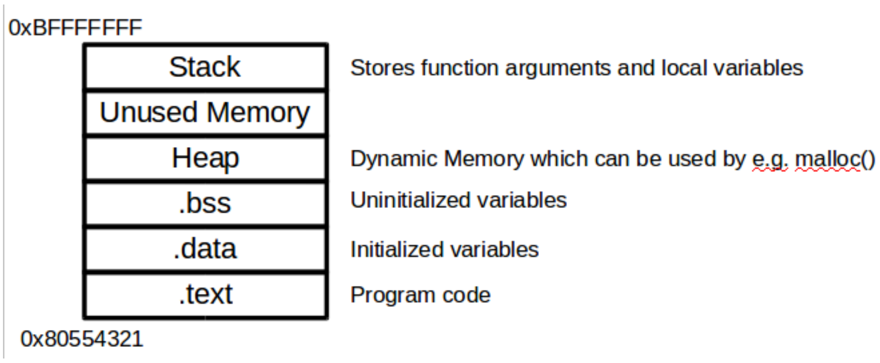


Referencing specific part of bits is done the same way with EBX, ECX and EDX.

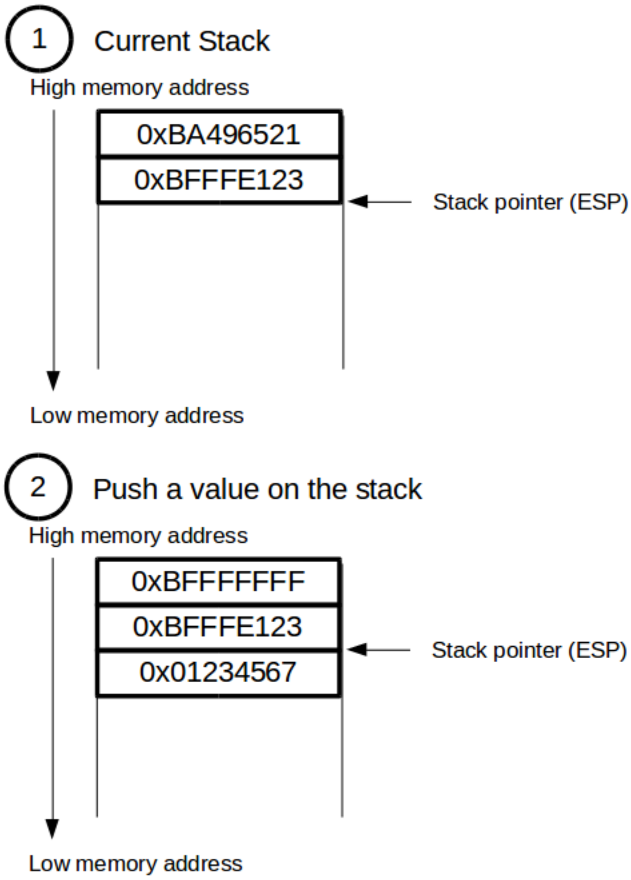
1.2 Memory

Virtual Memory Model

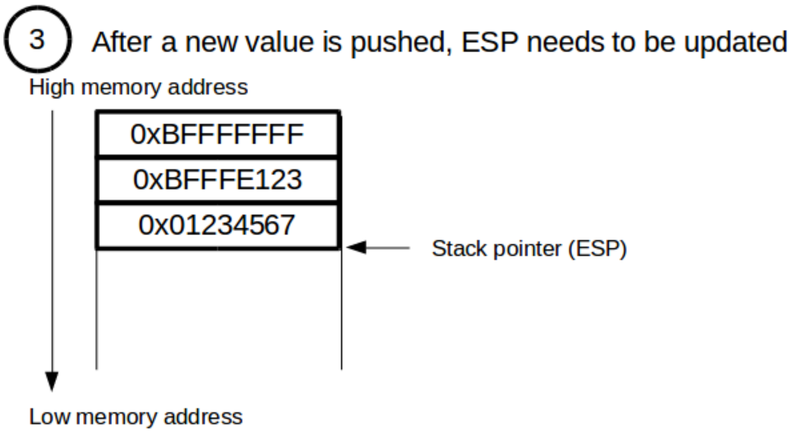
A process does not have to deal with the exact locations of variables in the RAM, but instead has a virtual memory which is always layed out the same way. The memory management unit (MMU) of the computer takes care of which parts of the RAM are still free and maps virtual memory to the physical RAM. This means that two processes can use the same memory locations in virtual memory but are located at completely different places in the physical memory.

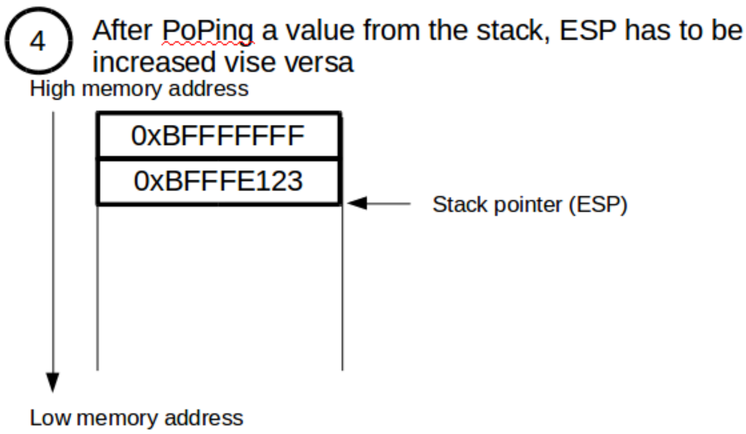


The stack starts at the highest memory address and grows downward into the unused Memory.



The current stack consists of two values and the stack pointer points to the top of the stack. After pushing a third value on the stack the stack pointer points to the wrong location (marked by a 2 in the figure) ESP gets updated such that it points to the top of the stack again (circle with a 3)





When using a PoP operation to remova a value from the stack, ESP has to be updated vise versa by pointing it to a higher memory address such that it points to the top of the stack again.

2 Part 2: Examining memory using GDB

Save the following source code as simpleDemo.c

simpleDemo.c

```
#include <stdio.h>
#include <stdlib.h>

int add(int x, int y)
{
        int z = 10;
        z = x + y;
        return z;
}
```

```

main(int argc, char **argv)
{
        int a = atoi(argv[1]);
        int b = atoi(argv[2]);
        int c;
        char buffer[100];

        gets(buffer); // gets userinput to buffer
        puts(buffer); // prints buffer

        c = add(a,b);
        printf("Sum of %d+%d = %d \n", a, b, c);
        exit(0);
}
```

compile it with gcc
$ gcc -ggdb -o simpleDemo simpleDemo.c
running the program with the inputs "10", "20", and "demo"
$ ./simpleDemo 10 20
$ demo
results in the following output
$ demo
$ Sum of 10+20 = 30

running this program in one terminal by typing: ./simpleDemo 10 20 leaves the process running
While the simpleDemo process is running it can be examined in another terminal

```
$ ps -aux | grep simpleDemo
$ sysgen 3781 0.0 0.0  4352 652 pts/2 S+ 12:12 0:00 ./simpleDemo 10 20
$ sysgen 3876 0.0 0.0 14224 980 pts/3 S+ 12:24 0:00 grep --color=auto simpleDemo
```

This output shows the process ID "3781". In Linux everything is a file. A running process like simpleDemo can be found in the /proc/3781 directory (it includes all runtime information). /proc/3781/maps includes the memory layout

```

$ cat /proc/3781/maps

00400000-00401000 r-xp 00000000 08:03 11797734        ~/myfolder/simpleDemo
00600000-00601000 r--p 00000000 08:03 11797734        ~/myfolder/simpleDemo
00601000-00602000 rw-p 00001000 08:03 11797734        ~/myfolder/simpleDemo
01e58000-01e79000 rw-p 00000000 00:00 0               [heap]
7f408ffe6000-7f40901a6000 r-xp 00000000 08:03 2627199 /lib/x86_64-linux-gnu/libc-2.23.so
7f40901a6000-7f40903a6000 ---p 001c0000 08:03 2627199 /lib/x86_64-linux-gnu/libc-2.23.so
7f40903a6000-7f40903aa000 r--p 001c0000 08:03 2627199 /lib/x86_64-linux-gnu/libc-2.23.so
7f40903aa000-7f40903ac000 rw-p 001c4000 08:03 2627199 /lib/x86_64-linux-gnu/libc-2.23.so
7f40903ac000-7f40903b0000 rw-p 00000000 00:00 0
7f40903b0000-7f40903d6000 r-xp 00000000 08:03 2627197 /lib/x86_64-linux-gnu/ld-2.23.so
7f40905af000-7f40905b2000 rw-p 00000000 00:00 0
```

```
7f40905d5000-7f40905d6000 r--p 00025000 08:03 2627197    /lib/x86_64-linux-gnu/ld-2.23.so
7f40905d6000-7f40905d7000 rw-p 00026000 08:03 2627197    /lib/x86_64-linux-gnu/ld-2.23.so
7f40905d7000-7f40905d8000 rw-p 00000000 00:00 0
7ffc1937a000-7ffc1939b000 rw-p 00000000 00:00 0          [stack]
7ffc193a4000-7ffc193a7000 r--p 00000000 00:00 0          [vvar]
7ffc193a7000-7ffc193a9000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
```

Section 1.2 describes the memory layout and just like described there, the program code is at the lowest address in virtual memory 00400000 - 00602000. Where the first part has an executable flag. libc-2.23.so has been loaded into 7f408ffe6000 -7f40903ac000. The stack is at the highest memory location 7ffc1937a000-7ffc1939b000. If simpleDemo is run in an additional terminal, and the memory layout is examined the nearly same memory layout can be seen. Only the range of the stack varies which is a feature of a linux 2.6 kernel (stack randomization) which makes it harder to exploit buffer overflows. This feature can be turned off by:

```
$ echo 0 > /proc/sys/kernel/randomize_va_space
```

3 Part 3: Examining memory using GDB

```
$ gdb ./simpleDemo
(gdb) list 1 --> shows the sourcecode
(gdb) run 10 20 --> runs the program with 10 and 20 as input
(gdb) disassemble main --> shows the assembler code of
                               the main function

   memory location                    instruction
   0x0000000000400667 <+0>:      push    %rbp
   0x0000000000400668 <+1>:      mov     %rsp,%rbp
   0x000000000040066b <+4>:      sub     $0x90,%rsp
   0x0000000000400672 <+11>:     mov     %edi,-0x84(%rbp)
   0x0000000000400678 <+17>:     mov     %rsi,-0x90(%rbp)
   0x000000000040067f <+24>:     mov     %fs:0x28,%rax
   0x0000000000400688 <+33>:     mov     %rax,-0x8(%rbp)
   0x000000000040068c <+37>:     xor     %eax,%eax
   0x000000000040068e <+39>:     mov     -0x90(%rbp),%rax
   ...
```

```
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3
4       int add(int x, int y)
5       {
6               int z = 10;
7               z = x + y;
8               return z;
9       }
```

```
10

(gdb) break 8 --> sets a breakpoint after the addtion
(gdb) run 10 20
qw
qw

Breakpoint 1, add (x=10, y=20) at simpleDemo.c:8
8               return z;

(gdb) print x --> only works while a break is set
       withing the function / within the stack frame
$1 = 10
info registers --> shows values of registers

rax            0x1e     30
rbx            0x0      0
rcx            0x7ffff7b042c0   140737348911808
rdx            0xa      10
rsi            0x14     20
rdi            0xa      10
rbp            0x7fffffffddf0   0x7fffffffddf0
rsp            0x7fffffffddf0   0x7fffffffddf0
...

x --> examine command to look at the value of
        specific memory locations
(gdb) help x
Examine memory: x/FMT ADDRESS.

e.g. (gdb) x/10xb 0x7fffffffddf0 --> starting at
      0x7fffffffddf0 , show the next 10, b=bytes,
      in x=hex
```

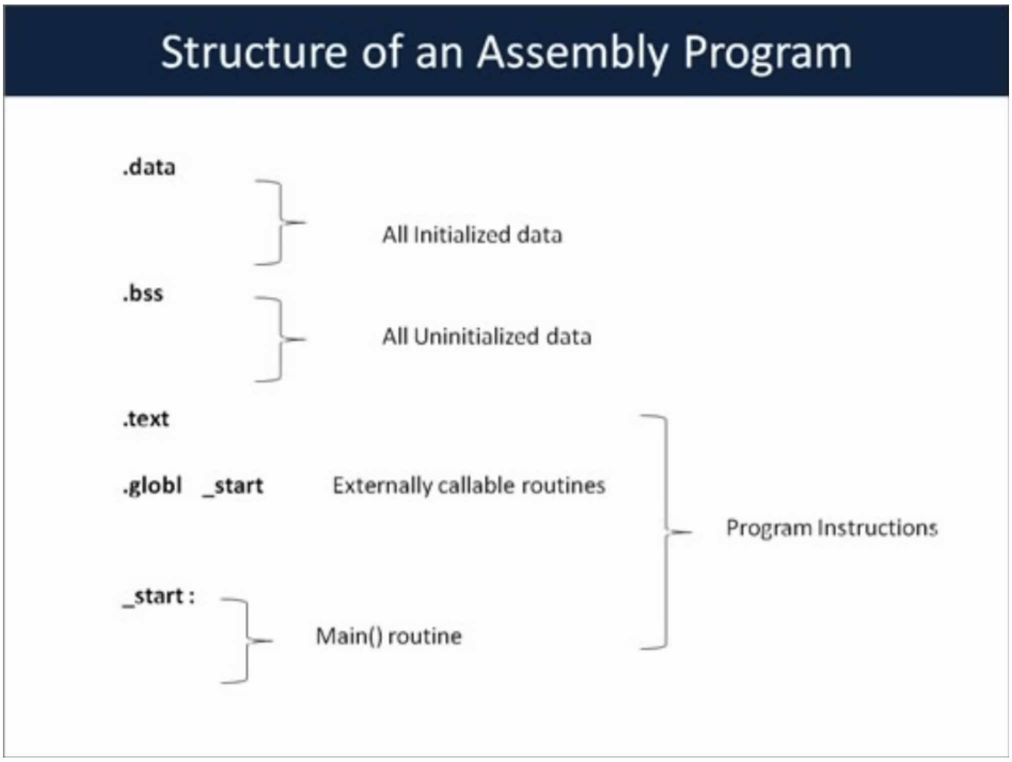
When something is pushed or poped from the stack, its usually a word (4 bytes) therefore when examining the stack, (gdb) x/12xw 0x7fffffffddf0 shows the next 12 values. Since $(gdb)run1020$ was used, somewhere we should find a 10 and 20 which are the values in x and y. In hex these are 10 = 0x0a and 20 = 0x14

```

(gdb) x/12xw 0x7fffffffddf0
0x7fffffffddf0: 0xffffde90      0x00007fff      0x004006ec      0x00000000
0x7fffffffde00: 0xffffdf78      0x00007fff      0x00000000      0x00000003
0x7fffffffde10: 0x00000000      0x0000000a      0x00000014      0x00000000
```

Those can be seen in the last row.

```
(gdb) s --> execute one instruction further
(gdb) continue --> execute until end of program
```

4 Part 4: Structure of an Assembly Program



- .text includes the executable code
- _start defines the starting point of the assembler program (first line of main)

- .globl _start defines the location for the code of libraries
- Linux System Calls are libraries which the kernel provides to get various tasks done
- List of system calls available in `/usr/include/asm/unistd.h`
- e.g. exit(), read(), write()

How do we pass arguments to syscalls?

- EAX - System Call number
- EBX - first argument

- ECX - second argument Counter
- EDS - third argument
- ESI - fourth argument Source Index
- EDI - fifth argument Destination index

Example of exit() in assembly

- Calling exit(0) to exit a program
- Function definition void_exit(int status);

In assembler

1. Sys call number for exit() is 1, so load EAX with 1
   movl $1, %eax (writes a one in the eax register)
2. ”status” is lets say ”0” - EBX must be loaded with ”0”
   movl $0, %ebx
3. Raise the software interrupt 0x80 (invokes the syscall)
   int 0x80

justExit.s (An assembly program which just exits)

```
.text

.globl _start

_start:
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

compiling an assembler program

```
$ as -o justExit.o justExit.s --> creates an object file
$ ld -o justExit justExit.o --> links object file to executable
$ ./justExit
```

Writing a Hello World in assembler

To create a Hello World, we need the write() syscall for console output ssize_t write(int fd, const void *buf, size_t count);
With fd=filedescriptor, *buf=buffer which is printed, Sys call number for write is 4 (store in EAX)
fd = 1 for stdout (in EBX)
Buf = pointer to memory location containing "hello world" string (in ECX)
Count = string length (in EDX)
(first to third arguments)

```
# My first Assembly program
.data
        HelloWorldString:
        .ascii "Hello World\n"

.text

.globl _start

_start:
        # Load all the arguments for write ()
        movl $4, %eax
        movl $1, %ebx
        movl $HelloWorldString, %ecx
        movl $12, %edx
        int $0x80

        # Exit the program
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

5 Part 5: Data Types in .DATA

- .byte = 1byte
- .ascii = string
- .asciz = Null terminated string
- .int = 32 bit integer
- .short = 16 bit integer
- .float = Single precision floating point number
- .double = Double precision floating point number

The Space in RAM is reserved at compile time

Data Types in .BSS

.comm -declares common memory area

.lcomm - declares local common memory area

The space in RAM is reserved at RUNTIME! Which is when you come to the line of code which creates the variable

```
# Demo program to show how to use Data types and MOVx instructions
.data
        HelloWorld:
                .ascii "Hello World!"

        ByteLocation:
                .byte 10

        Int32:
                .int 2

        Int16:
                .short 3

        Float:
                .float 10.23

        IntegerArray:
                .int 10,20,30,40,50

.bss
        .comm LargeBuffer, 10000

.text

        .globl _start

        _start:
                nop
                # Exit syscall to exit the program
                movl $1, %eax
                movl $0, %ebx
                int $0x80
```

```
as -gstabs -o VariableDemo.o VariableDemo.s
ld -o VariableDemo VariableDemo.o
gdb ./VariableDemo
(gdb) list
(gdb) break *_start+1
Breakpoint 1 at 0x4000b1: file VariableDemo.s, line 32.
this sets a breakpoint at "movl $1, %eax"
(gdb) run
(gdb) info variables
All defined variables:

Non-debugging symbols:
0x00000000006000bd   HelloWorld
0x00000000006000c9   ByteLocation
0x00000000006000ca   Int32
0x00000000006000ce   Int16
```

```
0x00000000006000d0   Float
0x00000000006000d4   IntegerArray
0x00000000006000e8   __bss_start
0x00000000006000e8   _edata
0x00000000006000f0   LargeBuffer
0x0000000000602800   _end
"memory location", "Label from assembly code"

(gdb) x/12cb 0x00000000006000bd
0x6000bd:        72 'H'  101 'e' 108 'l' 108 'l' 111 'o' 32 '
           '     87 'W'  111 'o'
0x6000c5:        114 'r' 108 'l' 100 'd' 33 '!'

(gdb) x/1dw 0x00000000006000ca
0x6000ca:        2

(gdb) x/1fw 0x00000000006000d0
0x6000d0:        10.2299995

(gdb) x/5dw 0x00000000006000d4
0x6000d4:        10       20       30       40
0x6000e4:        50

(gdb) x/10db 0x00000000006000f0 <— shows the first 10 bytes of our reserved buffer
```

6 Part 6: Basic Instructions - MOVx

Usage format: MOVx source, destination
This is AT& T syntax which also adds a % to almost every register. In intel syntax the % sign is missing and the usage format is MOVx destination, source MOV is a copy operation and does not change the value in the source! depending of the sice of the data, 3 options are available

- movl=moves a 32 bit value from eax to ebx (just a copy, eax is not changed)
movl %eax, %ebx

- movw = moves a 16 bit value
movw%ax,%bx

- movb = moves a 8 bit value
movb%ah,%bh

e.g. movl $10, %ebx moves the value 10 into the register ebx

Moving Data into an indexed memory location

```
integerArray:
.int 10,20,30,40,50
```

Selecting the 3rd integer "30"
BaseAddress(Offset, index, Size) = IntegerArray(0, 2, 4)
0 because we have no offset,
2 because we want to write into the 3rd element
4 because integers have a size of 4 byte
movl %eax, IntegerArray(0,2,4)

Placing the $ sign before a label name takes the memory address of the variable and not the value

```
movl $location , %edi \\
Movl $9, (%edi) --> place value "9" in memory location pointed to by EDI
Movl $9, 4(%edi) --> place value "9" in memory location pointed to by (EDI+4)
Movl $9, -2(%edi) --> place value "9" in memory location pointed to by (EDI-2)
```

```
# Demo program to show how to use Data types and MOVx instructions
.data
        HelloWorld:
                .ascii "Hello World!"

        ByteLocation:
                .byte 10

        Int32:
                .int 2

        Int16:
                .short 3

        Float:
                .float 10.23

        IntegerArray:
                .int 10,20,30,40,50

.bss
        .comm LargeBuffer, 10000

.text

        .globl _start

        _start:
                nop

                # 1. MOV immediate value into register
                movl $10, %eax

                # 2. MOV immediate value into memory location
                movw $50, Int16

                # 3. MOV data between registers
                movl %eax, %ebx
```

```

        # 4. MOV data from memory to register
        movl Int32, %eax

        # 5. MOV data from register to memory
        movb $3, %al
        movb %al, ByteLocation

        # 6. MOV data into an indexed memory location
        # Location is decided by BaseAddress(Offset, Index, DataSize)
        # Offset and Index must be registers, Datasize can be a numerical value

        movl $0, %ecx
        movl $2, %edi
        movl $22, IntegerArray(%ecx, %edi, 4)



        # Exit syscall to exit the program

        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

```
gdb ./MovDemo
(gdb) list
(gdb) break *_start+1
(gdb) run
(gdb) info registers    (eax has a value of 0)
(gdb) s
(gdb) info registers    (eax has a value of 10)
(gdb) x/1dh &Int16      (examining the variable Int16)=3
(gdb) s
(gdb) x/1dh &Int16      (examining the variable Int16)=50
```

7 Part 7: Working with Strings

For moving Strings from one memory location to another we have the MOVSx functions:
MOVSB - move a byte (8bits)
MOVSW - move a word (16 bits)
MOVSL - move a double word (32 bits)
These functions copy a certain amout of bytes from ESI to EDI and increment both registers for the corresponding number of bytes
Source - ESI points to memory location
Destination - EDI points to memory location

```
.data
    HelloWorldString:
        .asciz "Hello World of Assembly!"
```

```
        H3110:
                .asciz "H3110"

.bss
        .lcomm Destination, 100
        .lcomm DestinationUsingRep, 100
        .lcomm DestinationUsingStos, 100

.text
        .globl _start

        _start:
                nop
                # 1. Simple copying using movsb, movsw, movsl

                movl $HelloWorldString, %esi
                movl $Destination, %edi

                movsb # copies the "H" Letter into the Destination and
                      # increments $esi and $edi one byte
                movsw
                movsl

                # 2. Setting/Clearing the DF flag
                std # set the DF flag
                cld # clear the DF flag

                # 3. Using Rep
                movl $HelloWorldString, %esi
                movl $DestinationUsingRep, %edi
                movl $25, %ecx # set the string length in ECX
                cld # clear the DF
                rep movsb          # repeat movsb 25 times
                std

                # 4. Loading string from memory into EAX register
                cld
                leal HelloWorldString, %esi # load helloworldstring-address
                                            #into esi, "lea"= load effective
                                            # address instruction,
                                            # "l"=Long=double word
                lodsb
                movb $0, %a1

                dec %esi # movb has incremented esi and edi, this line
                         # decrements so the next movw starts at the
                         # beginning of the string
                lodsw
                movw $0, %ax

                subl $2, %esi # Make ESI point back to the original string
                lodsl

                # 5. Storing strings from EAX to memory
                leal DestinationUsingStos, %edi
                stosb
                stosw
                stosl

```

```
                # 6. Comparing Strings
                cld
                leal HelloWorldString, %esi
                leal H3110, %edi
                cmpsb

                dec %esi
                dec %edi
                cmpsw

                subl $2, %esi
                subl $2, %edi
                cmpsl

                # The exit() routine
                movl $1, %eax
                movl %10, %ebx
                int $0x80
```

```
as -ggstabs -o StringBasics.o StringBasics.s
ld -o StringBasics StringBasics.o
gdb ./StringBasics
(gdb) list
(gdb) break 19
(gdb) run

(gdb) print /x &HelloWorldString
$1 = 0x60011f

(gdb) print /x $esi
$2 = 0x60011f

(gdb) s
20 movl $Destination, %edi

(gdb) print /x $esi
$3 = 0x60011f

(gdb) s
22              movsb

(gdb) x/10cb &HelloWorldString
0x60011f:       72 'H' 101 'e' 108 'l' 108 'l' 111 'o'  32 ' ' 87 'W' 111 'o'
0x600127:       114 'r' 108 'l'

(gdb) x/1s &HelloWorldString
0x60011f:       "Hello World of Assembly!"

(gdb) s
23              movsw
# s executed "22 movsb and the next command is "23      movsw"

(gdb) x/1s &Destination
0x600140 <Destination>: "H"

(gdb) print /x $esi
$1 = 0x600120

(gdb) print /x $edi
```

```
$2 = 0x600141

# after movsb has copied a byte from source to destination
# $esi and $edi get incremented by 1 byte

(gdb) info registers
...
eflags         0x202    [ IF ] # gdb shows only the flags which are set, so the DF=0 is not shown
...

(gdb) s
24                      movsl
# $esi and $edi get incremented by two bytes
```

Direction Flag (DF) is a part of the EFLAGS registers
It decides whether to increment or decrement ESI and EDI after a MOVSx instruction
If DF is set to "1" $-->$ ESI, EDI registers are decremented
If DF is set to "0" $-->$ ESI, EDI registers are incremented
You can set DF using the STD instruction
You can clear DF using the CLD instruction

The REP instruction
is used to repeat a string instruction over and over again till the ECX register has a value > 0

Simple use:
- load the ECX register with the string length
- Use the REP MOVsx instruction to copy the string from source to destination

```
# 3. Using Rep
movl $HelloWorldString, %esi
movl $DestinationUsingRep, %edi
movl $25, %ecx # set the string length in ECX
cld # clear the DF
rep movsb           # repeat movsb 25 times
std
```

Loading Strings from Memory into Registers
- loads into the EAX register
- String source pointed to by ESI

LODSx
LODSB $-->$ Load a byte from memory location into AL
LODSW $-->$ Load a word from memory into AX
LODSL $-->$ Load a double word from memory into EAX

ESI is automatically incremented / decremented based on the DF flag after the LODSx instruction executes

```
# 4. Loading string from memory into EAX register
```

```
cld # clear direction flag
leal HelloWorldString, %esi     # load helloworldstring-address
                                #into esi,
                                "lea"= load effective address instruction,
                                "l"=Long=double word
lodsb
movb $0, %al
```

Storing Strings from Registers into Memory

- Storing Strings
- Stores into memory from EAX
- EDI points to destination memory

- STOSx:
- STOSB −− > store AL to memory
- STOSW −− > store AX to memory
- STOSL −− > store EAX to memory

- EDI is incremented / decremented based on the DF flag after every STOSx instruction is executed

Comparing Strings
- ESI contains Source string, EDI the Destination string
- DF flag decides whether ESI/EDI are incremented / decremented

- CMPSx
- CMPSB −− > compares byte value
- CMPSW −− > compares word value
- CMPSL −− > compares double word value

- CMSPx subtracts the destination string from the source string and sets the EFLAGS register appropriately (if equal strings the zero flag is set)

8 Part 8: Unconditional Branching

1. JPM:
1.1 Compare it with GOTO: statement in C
1.2 Syntax-JMP Label
1.3 Short, Near and Far jump possible

UnconditionalBranching.s

```
.data
        HelloWorldString:
                .asciz "Hello World!"
        CallDemo:
                .asciz "Call works!"

.text
        .globl _start

        _start:
                nop
                # Write HelloWorld
        movl $4, %eax
        movl $1, %ebx
                movl $HelloWorldString, %ecx
        movl $12, %edx
        int $0x80

    ExitProgram:
        #Exit the program
        movl $1, %eax
        movl $10, %ebx
        int $0x80

    CallMe:
        movl $4, %eax
        movl $1, %ebx
        movl $CallDemo, %ecx
        movl $11, %edx
        int $0x80
        ret
```

```
$ as -ggstabs -o UnconditionalBranching.o UnconditionalBranching.s
$ ld -o UnconditionalBranching UnconditionalBranching.o
$ ./UnconditionalBranching
Hello World!
```

adding a jump instruction before "Hello World" is printed should neglect any output **UnconditionalBranching2.s**

```
...

        _start:
                jmp ExitProgram
                # Write HelloWorld
...
```

```
$ as -ggstabs -o UnconditionalBranching2.o UnconditionalBranching2.s
$ ld -o UnconditionalBranching2 UnconditionalBranching2.o
$ ./UnconditionalBranching 2
```

and it does so.

```
$ gdb ./UnconditionalBranching2
(gdb) disas _start
Dump of assembler code for function _start:
   0x00000000004000b0 <+0>:     jmp    0x4000c8 <ExitProgram>
   0x00000000004000b2 <+2>:     mov    $0x4,%eax
```

```
    0x00000000004000b7 <+7>:     mov    $0x1,%ebx
    0x00000000004000bc <+12>:    mov    $0x6000eb,%ecx
    0x00000000004000c1 <+17>:    mov    $0xc,%edx
    0x00000000004000c6 <+22>:    int    $0x80
End of assembler dump.

(gdb) break *_start+1
(gdb) s
(gdb) print $rip
$4 = (void (*)()) 0x4000b1 <_start+1>
(gdb) print $rip
$5 = (void (*)()) 0x4000c9 <ExitProgram>
```

1. JPM:
1.1 Compare it with GOTO: statement in C
1.2 Syntax-JMP Label
1.3 Short, Near and Far jump possible

2. Call:
2.1 Just like Calling a function in C
2.2 Syntax-Call location
2.3 There is an associated "RET" statement with every call
2.3.1 Similar to "return" statement in other languages
2.4 Using "Call" pushes the next instruction address onto the stack
2.4.1 This instruction is popped back into EIP on hitting the RET instruction

UnconditionalBranching3.s

```
...

        _start:
                call CallMe
                # Write HelloWorld
...
```

```
$ as -ggstabs -o UnconditionalBranching3.o UnconditionalBranching3.s
$ ld -o UnconditionalBranching3 UnconditionalBranching3.o
$ ./UnconditionalBranching3
Call works!Hello World
$ gdb ./UnconditionalBranching3
(gdb) list
...
12              call CallMe
...
(gdb) break 12
(gdb) run

(gdb) print /x $rip
$4 = 0x4000b1

(gdb) print /x $rsp
$5 = 0x7fffffffdf70

```

```
(gdb) disas _start
Dump of assembler code for function _start:
   0x00000000004000b0 <+0>:     nop
=> 0x00000000004000b1 <+1>:     callq  0x4000d8 <CallMe>
   0x00000000004000b6 <+6>:     mov    $0x4,%eax
   0x00000000004000bb <+11>:    mov    $0x1,%ebx
   0x00000000004000c0 <+16>:    mov    $0x6000ef,%ecx
   0x00000000004000c5 <+21>:    mov    $0xc,%edx
   0x00000000004000ca <+26>:    int    $0x80

(gdb) s
CallMe () at UnconditionalBranching3.s:27
27              movl $4, %eax

(gdb) print /x $rip
$6 = 0x4000d8

(gdb) disas CallMe
Dump of assembler code for function CallMe:
=> 0x00000000004000d8 <+0>:     mov    $0x4,%eax
   0x00000000004000dd <+5>:     mov    $0x1,%ebx
   0x00000000004000e2 <+10>:    mov    $0x6000fc,%ecx
   0x00000000004000e7 <+15>:    mov    $0xb,%edx
   0x00000000004000ec <+20>:    int    $0x80
   0x00000000004000ee <+22>:    retq

(gdb) print /x $rsp
$7 = 0x7fffffffdf68 # the stack pointer has changed
```

9 Part 9: Conditional Branching

JXX - JA, JAE, JE, JG, JZ, JNZ ... etc
Dictated by the state of the
Zero flag (ZF)
Parity flag(PF)
Overflow flag (OF)
Sign Flag (SF)
Carry Flag (CF)

In order to use conditional Jumps you must have an operation which sets the EFLAGS register appropriately

In conditional Jumps - only Short and Near jumps are supported. Far jumps are not supported

conditionalBranching.s

```
.data
        HelloWorld:
                .asciz "Hello World!\n"

        ZeroFlagSet:
```

```
                .asciz  "Zero Flag was Set!"

        ZeroFlagNotSet:
                .asciz  "Zero Flag Not Set!"

.text
        .globl _start

        _start:
        nop
        movl $10, %eax # if this operation results in zero, jz FlagSetPrint
                       # is executed
        # e.g. xorl %eax, %eax
        jz FlagSetPrint

    FlagNotSetPrint:
        movl $4, %eax
        movl $1, %ebx
        leal ZeroFlagNotSet, %ecx
        movl $19, %edx
        int $0x80
        jmp ExitCall

    FlagSetPrint:
        movl $4, %eax
        movl $1, %ebx
        leal ZeroFlagSet, %ecx
        movl $19, %edx
        int $0x80
        jmp ExitCall

    ExitCall:
        movl $1, %eax
        movl $0, %ebx
        int $0x80

    PrintHelloWorld:
        movl $10, %ecx
        PrintTenTimes:

            # Save %ecx before it is overwritten
            pushl %ecx
            movl $4, %eax
            movl $1, %ebx

            #leal loads effective address not value
            # this would overwrite the loop counter in ecx
            leal HelloWorld, %ecx
            movl $14, %edx
            int $0x80
            popl %ecx
        #Loop will decrement from %ecx
        loop PrintTenTimes
jmp ExitCall
```

```
$ as --32 -ggstabs -o conditionalBranching.o conditionalBranching.s
$ ld -m elf_i386 -o conditionalBranching conditionalBranching.o
$ ./conditionalBranching.o
```

LOOP Instruction
Used to loop through a set of instructions a predetermined number of times
The number of loops is stored in ECX, LOOP automatically decrements ECX after every run
Sample usage:

```
<code>
movl $10, %ecx # 10 is the number of times to loop
LoopThis:
        <code>
        <code>
        LOOP LoopThis
```

LOOPZ - loop until ECX is not zero or the zero flag (ZF) is not set
LOOPNZ - loop until ECX is not zero or the zero flag (ZF) is set

10 Part 10: Functions in Assembly

Defining a function in assembly:
.type MyFunction, @function

```
MyFunction:
        <code>
        <code>
        ret
```

The function is invoked using "call MyFunction"

Passing Arguments to functions

- Registers
- Global Memory locations
- Stack

The return value can be outputed by Registers or global memory locations.

function.s

```
.data
    HelloWorld:
        .asciz "Hello World!\n"

    HelloFunction:
        .asciz "Hello Function!\n"

.text
    .globl _start
    .type MyFunction, @function
    MyFunction:
        movl $4, %eax
```

```
        movl $1, %ebx
        int $0x80
        ret

    _start:
        nop
        leal HelloWorld, %ecx
        movl $14, %edx # length of "Hello World!\n"
        call MyFunction
        leal HelloFunction, %ecx
        movl $17, %edx
        call MyFunction

    ExitCall:
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

```
$ as -ggstabs -o function.o function.s
$ ld -o function function.o
$ ./function
```

Using global memory:

function2.s

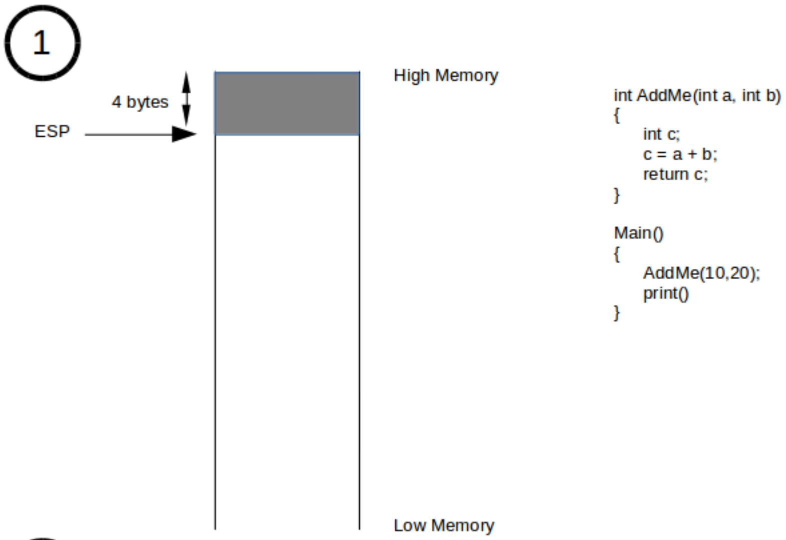
```
.data
    HelloWorld:
        .asciz "Hello World!\n"

    HelloFunction:
        .asciz "Hello Function!\n"

.bss
    .lcomm StringPointer, 4
    .lcomm StringLength, 4

.text
    .globl _start
    .type MyFunction, @function
    MyFunction:
        movl $4, %eax
        movl $1, %ebx
        movl StringPointer, %ecx
        movl StringLength, %edx
        int $0x80
        ret

    _start:
        nop

        # Print the Hello World String
        movl $HelloWorld, StringPointer
        movl $14, StringLength
        call MyFunction

        # Print Hello Function
        movl $HelloFunction, StringPointer
        movl $17, StringLength
        call MyFunction
```

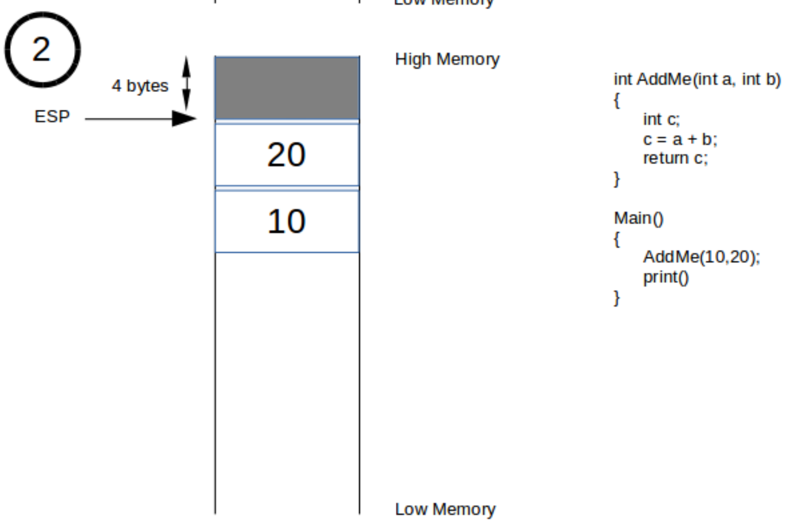
```

    ExitCall:
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

```
$ as -ggstabs -o function2.o function2.s
$ ld -o function2 function2.o
$ ./function2
```

11 Part 11: Passing Arguments via the Stack

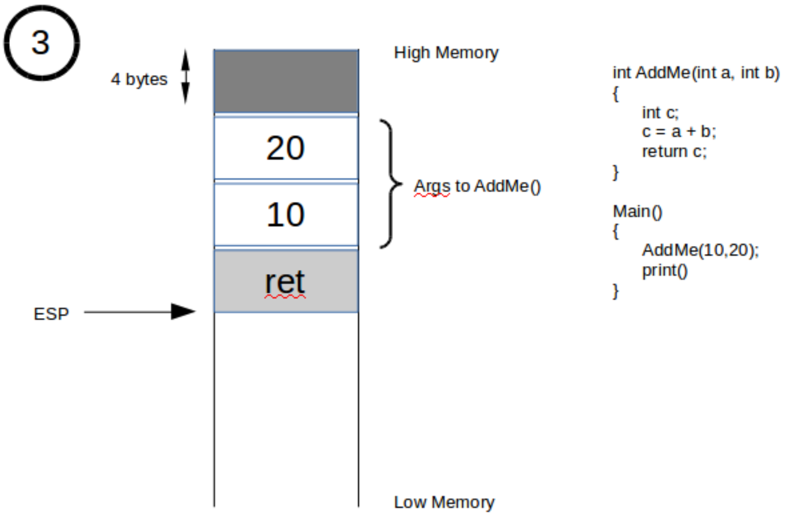
When main is called, the AddMe function is called. Before this function is executed the arguments have to be passed to the function, so they get placed on the stack. This is the transition from image 1 to image 2.

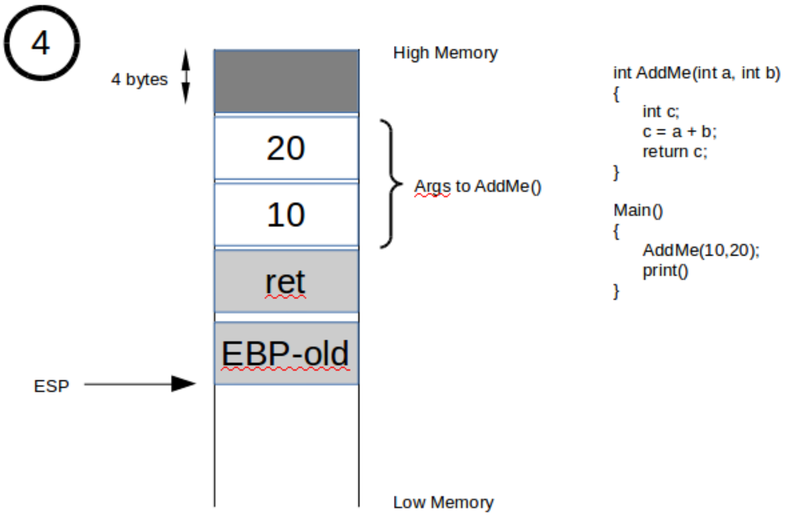




Afterwards the ESP is updated such that it points to the top of the stack again. Somehow the program needs to know where to continue after AddMe is done. To do this, the "return value" which is the address of the next instruction is pushed onto the stack (shown in image 3 as "ret") and the ESP is updated again.

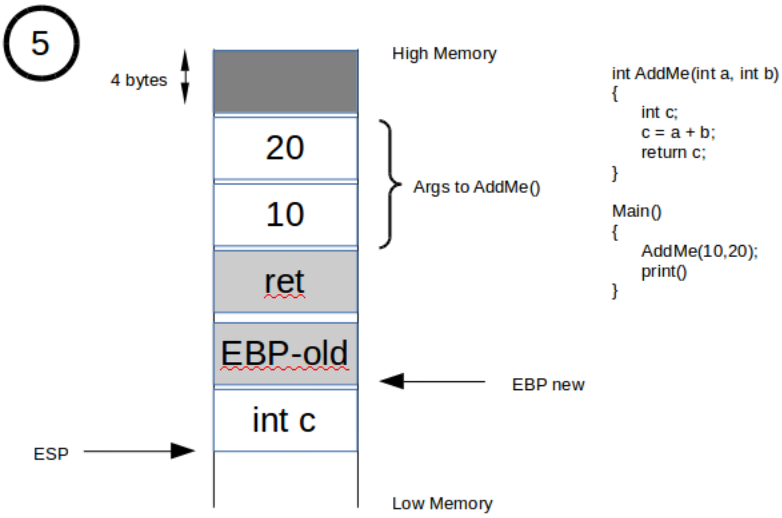
After doing this the program continues with the AddMe function. In this function ESP might be used which would overwrite ESP but ESP is needed to have a reference where the variables of main are placed. The EBP (base pointer) is therefore used but it could also contain a value from the main function. The EBP is therefore saved by placing it on the stack and ESP is updated (image 4).

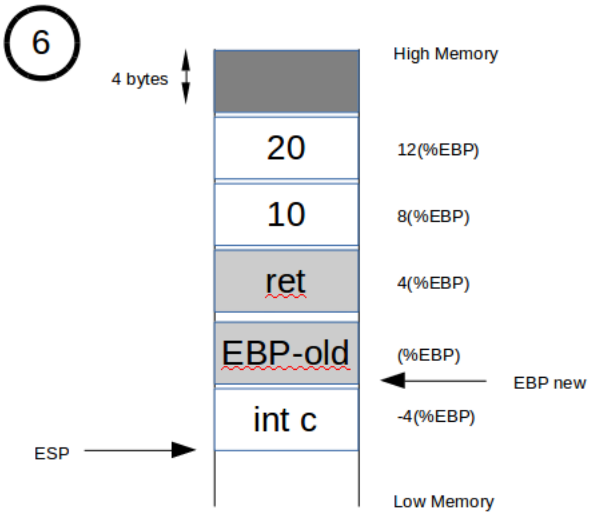




Once this is done ESP is copied into EBP such that ESP can be used further and still have a reference to the variables of the previous function (a so called stack frame). If local variables are required like c in the current function AddMe, they can simply be placed on the stack and ESP gets updated like usual (image 5).

The values of the previous or current function can now be accessed like shown in image 6 by the EBP register.





function3.s

```
.data
    HelloWorld:
        .asciz "Hello World!\n"

.text
    .globl _start
    .type PrintFunction, @function

    PrintFunction:
        pushl %ebp # store the current value of EBP on the stack
        movl %esp, %ebp # Make EBP point to the top of the stack

        # The write function
        movl $4, %eax # syscall number for write
        movl $1, %ebx # sends the write output to stdout
```

```
        movl 8(%ebp), %ecx # %ebp --> interpret ebp as an address,
                           #          add 8 to it, copy it into ecx = 10
        movl 12(%ebp), %edx # stringlength
        int $0x80

        movl %ebp, %esp # Restore the old value of ESP
        popl %ebp # Restore the old value of EBP
        ret # change EIP to start the next instruction

    _start:
        nop

        # push the strlen on the stack
        pushl $13

        # push the string pointer on the stack
        pushl $HelloWorld

        # Call the function
        call PrintFunction

        # adjust the stack pointer
        addl $8, %esp

        # Exit Routine

    ExitCall:
        movl $1, %eax
        int $0x80
```

To recap, the write syscall needs two important parameters. The length of a string and a pointer to the start of the string. Both are pushed on the stack right after the _start. When the call instruction is executed the instruction pointer for the next instruction (addl $8, %esp) is placed on the stack called RET for return address. In the print funtion the current value of ebp is stored on the stack followed by saving the esp (the stack address before the function starts) in ebp. From here ebp can be used as a reference for all arguments.

```
$ as --32 -ggstab -o function3.o function3.s
$ ld -m elf_i386 -o function3 function3.o
$ ./function3
Hello World!

$ gdb ./function3
(gdb) list
...
27              # push the strlen on the stack
28              pushl $13
...
(gdb) break 28 # breaks before line 28 is executed
(gdb) run
(gdb) print /x $esp
$1 = 0xffffd100 # current address of esp
(gdb) x/4xw $esp
0xffffd100:     0x00000001      0xffffd2cc      0x00000000      0xffffd31a
```

```
(gdb) s # executes pushl $13
(gdb) x/4xw $esp
0xffffd0fc:     0x0000000d      0x00000001      0xffffd2cc      0x00000000
                # 0x0000000d = 13

# the next instruction to be executed is pushl $HelloWorld
(gdb) print /x &HelloWorld
$2 = 0x80490a4

(gdb) s
(gdb) x/4xw $esp
0xffffd0f8:     0x080490a4      0x0000000d      0x00000001      0xffffd2cc

(gdb) disas _start
Dump of assembler code for function _start:
   0x0804808d <+0>:     nop
   0x0804808e <+1>:     push   $0xd
   0x08048090 <+3>:     push   $0x80490a4
=> 0x08048095 <+8>:     call   0x8048074 <PrintFunction>
   0x0804809a <+13>:    add    $0x8,%esp # <--- this should be the return address

(gdb) s
(gdb) x/4xw $esp
0xffffd0f4:     0x0804809a      0x080490a4      0x0000000d      0x00000001

(gdb) print /x $ebp
$3 = 0x0

(gdb) s # executes pushl %ebp
(gdb) x/4xw $esp
0xffffd0f0:     0x00000000      0x0804809a      0x080490a4      0x0000000d
esp value       old ebp         ret               arg2    arg1

(gdb) print /x $ebp
$4 = 0x0
(gdb) print /x $esp
$5 = 0xffffd0f0
(gdb) s # copy esp into ebp
(gdb) print /x $ebp
$6 = 0xffffd0f0

(gdb) s
15              movl $1, %ebx
(gdb) s
16              movl 8(%ebp), %ecx


(gdb) x/4xw $ebp
0xffffd0f0:     0x00000000      0x0804809a      0x080490a4      0x0000000d
# ebp points to 0x00000000
# 4 bytes further points to 0x0804809a
# 8 bytes further points to 0x080490a4 which points to "Hello World!\n"
(gdb) x/1s 0x080490a4
0x80490a4:      "Hello World!\n"
```