The Holy Book of x86

Volume 1

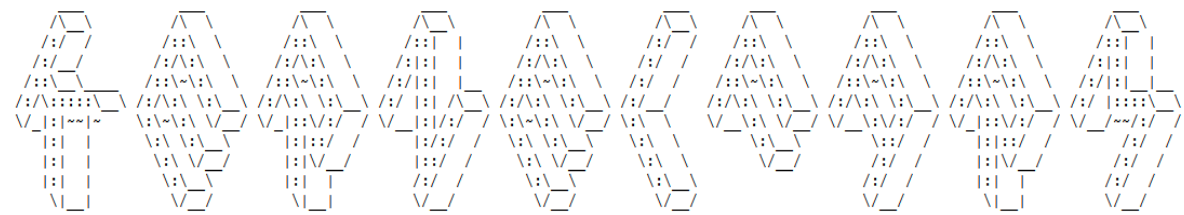
Intoduction to x86 Architecture

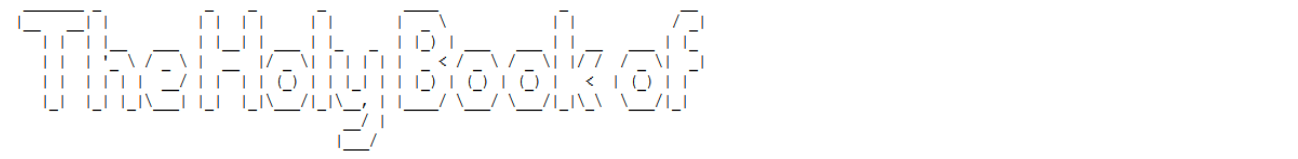
Learn Most Frequently Used Assembly Instructions

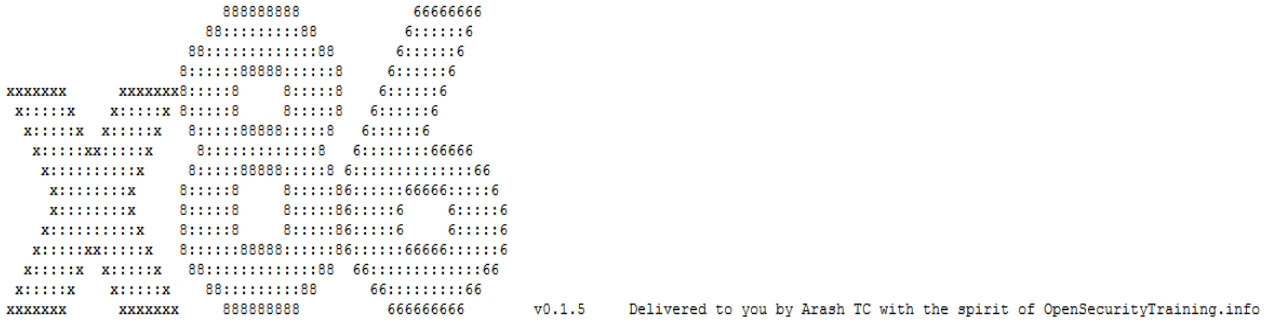
Understand Basic Assembly Conventions

Based on Intel Assembly Flavor

Arash TC







ARE YOU SUCH A DREAMER TO PUT THE WORLD TO RIGHTS?
I STAY HOME FOREVER
WHERE 2 AND 2
ALWAYS MAKES A 5

[THOM YORKE - 2 + 2 = 5]

Table of Contents

Acknowledgement

I owe everything I know about x86 architecture to Xeno Kovah. A man who shared his class videos and slides freely available to everyone which is a noble act. In return to his great efforts, I decided to write this tutorial on x86 architecture and assembly and publish it for free so whoever is interested can learn and contribute.

About the Author[s]

Arash TC is the main author and maintainer of this book. He will appreciate readers' comments, criticisms and contributions. His main interest is security at its deepest level.

Special thanks to **Gerardo Iglesias Galván** for doing a full technical review.

You can contact the author[s] by visiting http://www.kernelfarm.com/

Introduction

This book/guide/tutorial/wiki is about assembly and x86 architecture. It's written by a low-level security dude for low-level security dudes. If you want to learn Assembly and its structure, reversing basics, Segmentation, Paging, etc. keep on reading. I highly recommend you check opensecuritytraining.info website and watch Intro to x86 videos as you read this book.

You need Intel Developer's Manual as a quick reference throughout this book. You can download it from the link below:

https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

Data Types

There are 5 different data types to deal with in the world of Assembly. They are as follows:

Byte: A byte is simply an 8-bit value (1 byte) and the C equivalent of a Byte is when you define a character like:

char alpha = 'a';

Word: A word is twice the size of a byte; 16-bit value (2 bytes) and it translates to this piece of code in C:

short int = 'a'

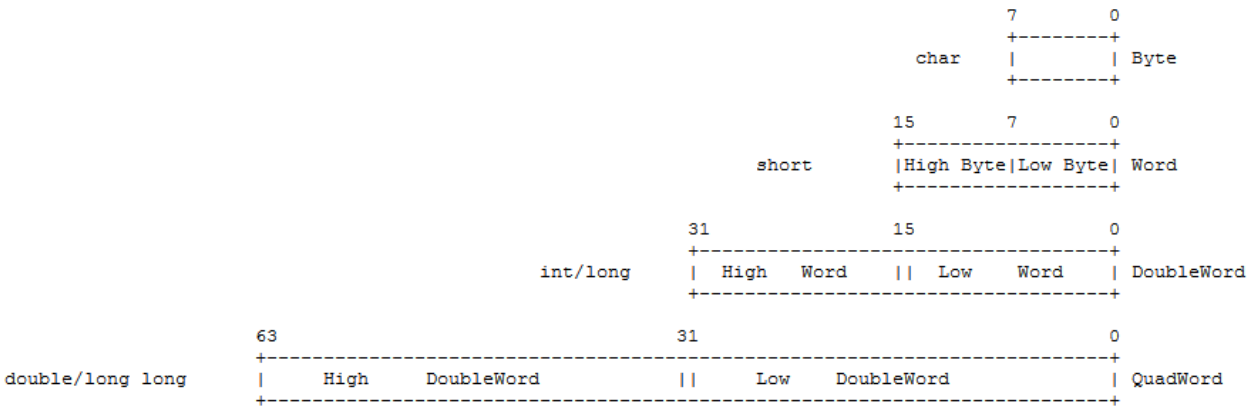
DoubleWord: As its name represents, a Double-Word or a DWORD is a 32-bit value (4 bytes) and it would be translated to:

int var = 3

QuadWord: A 64-bit value (8 bytes) with the C equivalent of:

long int var = ...

Double-QuadWord: You do the math :)



Binary-Decimal-Hex Refresher

If you don't know how to work with Hexadecimal and Binary values, I have to tell you to stop reading this and go kill yourself. Seriously? Ok. Here's a refresher for you:

Decimal (base 10)	Binary (Base 2)	HEX (Base 16)
00	0000	0x00
01	0001	0x01
02	0010	0x02
03	0011	0x03
04	0100	0x04
05	0101	0x05
06	0110	0x06
07	0111	0x07
08	1000	0x08
09	1001	0x09
10	1010	0x0A
11	1011	0x0B
12	1100	0x0C
13	1101	0x0D
14	1110	0x0E
15	1111	0x0F

Negative numbers

Negative numbers in x86 architecture may seem a little bit weird at first. A negative Number named N with the positive value of P, is P's two's complement which is equal to one's complement plus one.

Holy shit! What was that again? OK! It is very simple and clear if you see it in action.

P = 1 in decimal = 0x01 in Hex = 00000001 in Binary.
One's complement is when you flip all the bits of the number P. So:
P = 00000001 and P's One's complement is all P's bits flipped which equals to:

Flipped_P = 11111110 in Binary.
Now what happens if you add one to it and convert it to Hex?
Flipped_P + 1 = 11111110 + 00000001 = 11111111 ---> 0xFF in Hex.

So negative one in x86 Hex format would be 0xFF. You can take a look at the following table to completely comprehend it.

P	Flipped_P (One's Complement	Two's Complement
0x00000001	0xFFFFFFFE	0xFFFFFFFF

For signed integers, we have these scopes:

-From byte 0x01 to byte 0x7F, all bytes are positive.

-From byte 0x80 to byte 0xFF, all bytes are negative (They are in reversed order, 0xFF is -1 and as you approach a smaller hex value, you approach a smaller negative number).

-From DWORD 0x00000001 to DWORD 0x7FFFFFFF, all DWORDs are positive.

-From DWORD 0x80000000 to DWORD 0xFFFFFFFF, all DWORDs are negative (They are in reversed order, which means that the last one (0xFFFFFFFF) is -1 all the way down to the smallest).

	positive	negative
from	0x00000001	0x80000000
to	0x7fffffff	0xffffffff

Little Endian or Big Endian?

Endianness comes from Jonathan Swifts' "Gulliver's Travels". It doesn't matter which way you eat your eggs and it certainly matter in computer architecture, right? Well, I don't know exactly. Probably not. But what matters is that Intel Architecture is "Little-Endian". So what's up with that?

In a Little-Endian Architecture, values are stored in RAM starting from the lowest byte. For example, this is what happens if you want to store the address 0x12345678 in memory:

```
0x12345678  -->  0x12 0x34 0x56 0x78  -->
                                        |
                                        |
Into RAM    <--  0x78 0x56 0x34 0x12 <--
```

So it's simple. It just starts storing from that lowest byte up to the highest byte.

In Big-Endian Architecture, values are stored in RAM as they are. Network traffic is Big-Endian. PowerPC, ARM, SPARC, MIPS, etc. are Big-Endian unless otherwise configured.

*** Note: Register values are always Big-Endian. Little-Endian only applies when writing to and reading from RAM (Memory).

REGISTERS

Registers are small memory storage areas built into the processor. They are still a volatile type of memory storage so if you power off your PC, you're gonna lose the state of your registers. Intel architecture defines 8 General Purpose Registers (GPR) for 32 bit platforms and 16 GRP for 64 bit platforms as follows:

32-bit	64-bit
EAX	RAX
EBX	RBX
ECX	RCX
EDX	RDX
ESI	RSI
EDI	RDI
EBP	RBP
ESP	RSP
	R8
	R9
	R10
	R11
	R12
	R13
	R14
	R15

Each of the registers above are 4 bytes long in the 32-bit version, and 8 bytes long in the 64-bit version. Beside those General Purpose Registers, we have EIP (RIP for 64-bit) which is called the Instruction Pointer which holds the current flow of the execution; and we have EFLAGS, which is a 32-bit long register of registers. Yeah, I know that might sound crazy but I will explain them fully later.

EAX is mostly used when a function wants to return a value and it is also used for lots of different purposes. You have to see them in action in order to recognize its usability in different scenarios. EBX is the Base Pointer for data section and EDX is the I/O pointer but let's save these conventions for later. ECX is mostly used as a counter for repetitive instructions i.e. for a loop. ESI and EDI are used as Source Index and Destination Index respectively i.e. copying a string value. ESP is the stack pointer which always points to the top of the current stack. EBP is the base pointer which always (actually not always :D) points to the bottom

of the current stack frame by convention. I have to mention that these are only some conventions and you don't have to use them in the exact way. It is simply for simplicity and readability for your code.

The Stack

Two very important concepts you need to know, Stack and Heap. I'm gonna tell you what Stack is now. Stack is a conceptual area of memory (RAM) which mostly holds a function's local variables. Stack has a Last-In-First-Out data structure, meaning that the first thing that is pushed onto the stack is the last thing that is gonna pop out. Imagine a bucket full of apples. The first apple that you put inside the bucket is the last one that you can pull out (of course if you don't just turn the bucket upside down :D).

Stack grows down from higher memory addresses to lower memory addresses. For example, if the stack starts from address 0x7fff4444 (ESP), the next DWORD (4 bytes, remember?) that you push onto the stack, decrements the stack by 4 bytes and then ESP will point to 0x7fff4444 - 4 = 0x7fff4440. You will see this in greater detail in the next few paragraphs.

OK, you may want me to cut the bullshit and show you some real stuff, right? How about we see a simple C program? Hell no! We're just getting started. Kidding aside, there are still some major things you need to know in order to fully understand even a simple "Hello World" code in Assembly. So stick with me and be patient.

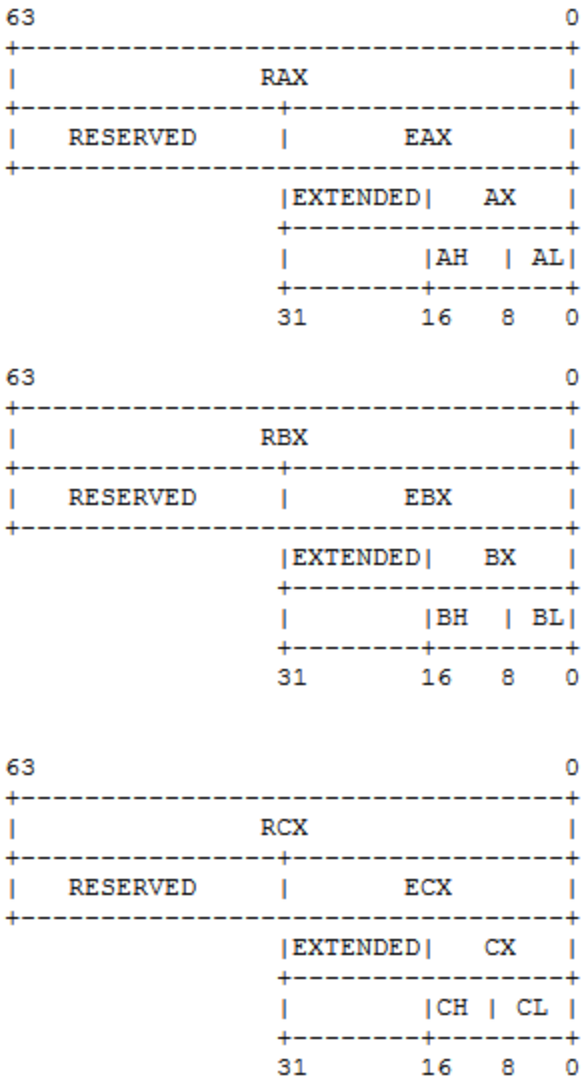
Caller - Callee Convention

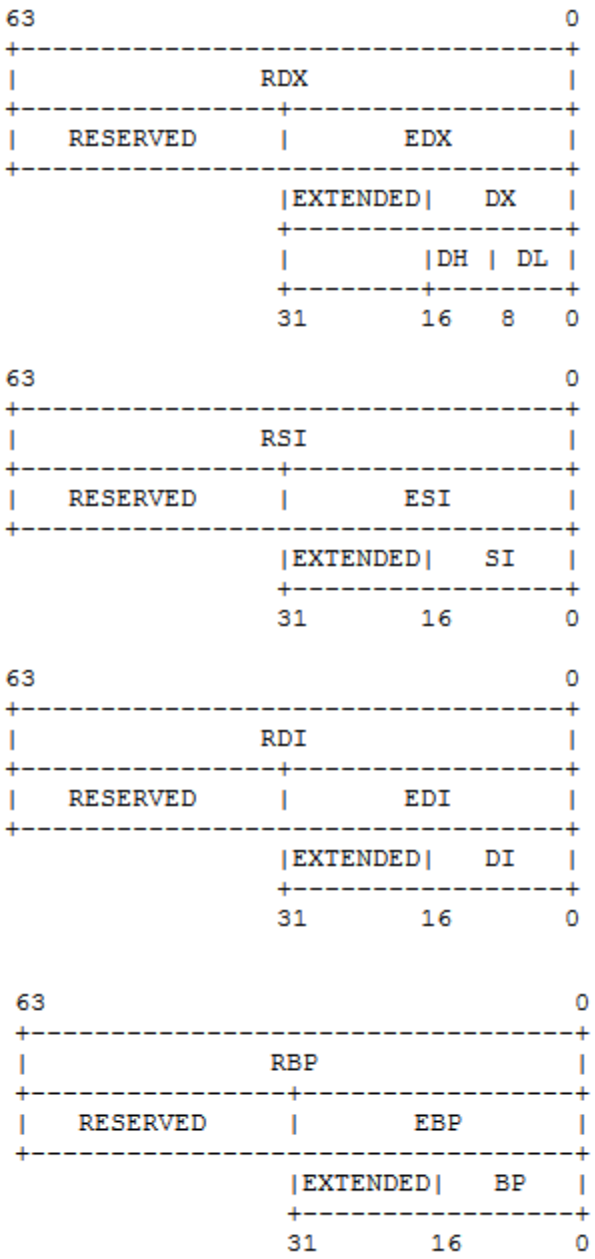
Caller Save Registers mean that whenever you want to call a function, save these registers (EAX,EDX,ECX 32-bit or RAX,EDX,ECX for 64-bit) somehow so when the execution is handed over to the function, your data will remain intact. That means the caller is responsible of saving these registers in order to prevent their destruction when the function modifies the values held in these registers. The caller is also responsible for restoring the saved values in registers when execution gets back to it.

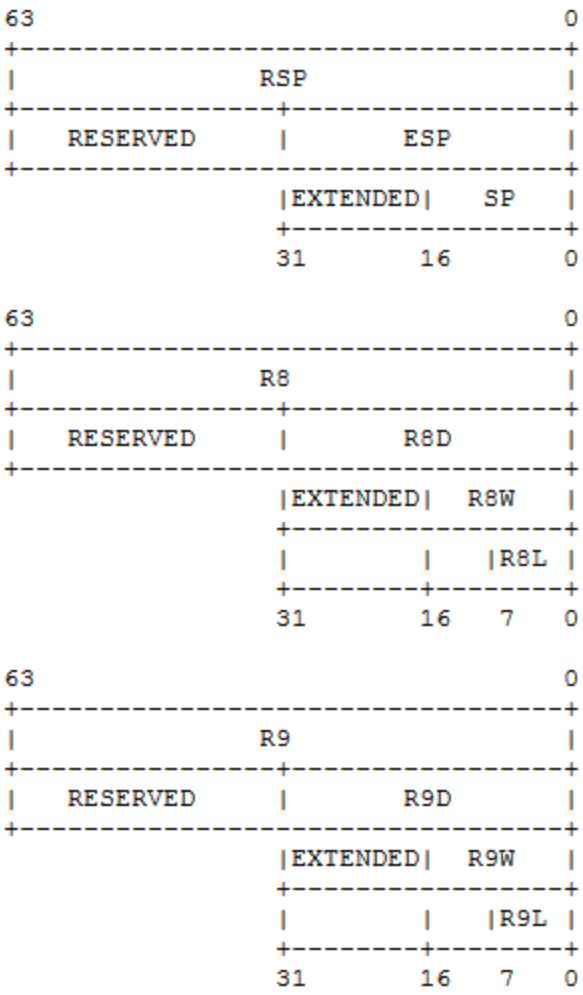
Callee Save Registers means that when the function (the callee) needs more registers than those which are already saved by the caller, callee is responsible for saving those values before going to its actual execution point. The registers that the callee is responsible for are EBP, EBX, ESI, EDI (RBP, RBX, RSI, RDI 64-bit). The callee is responsible for restoring these saved values back to their place before handing the execution back to the caller.

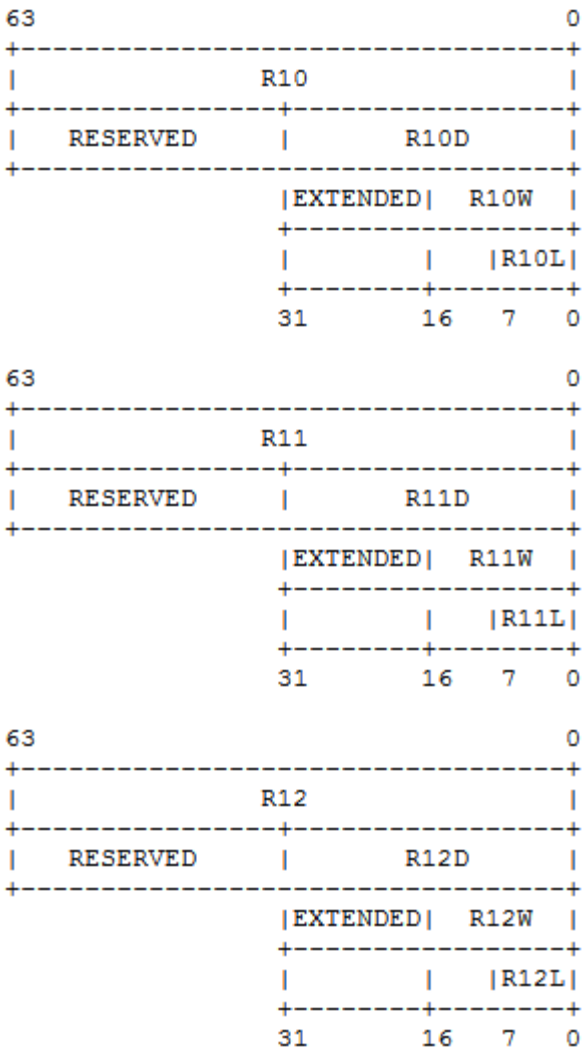
Structure of Registers

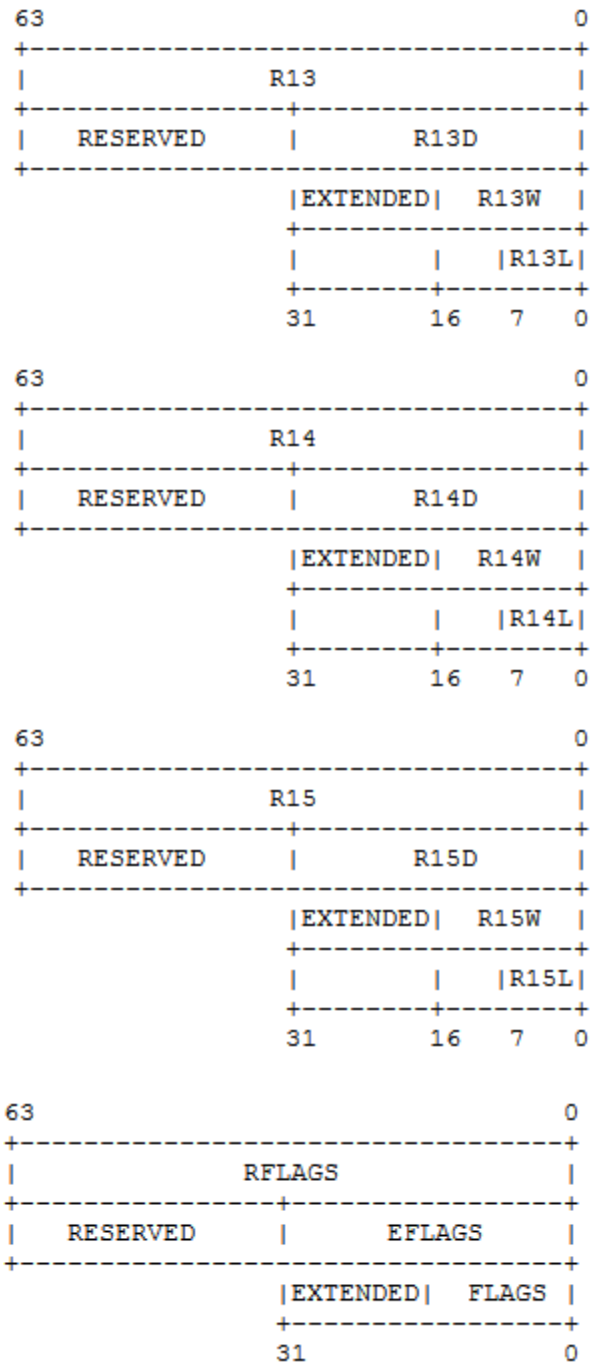
Every register is divided into some smaller pieces like below:




63
0
RDX
RESERVED
EDX
EXTENDED
DX
DH
DL
31
16
8
0
63
0
RSI
RESERVED
ESI
EXTENDED
SI
31
16
0
63
0
RDI
RESERVED
EDI
EXTENDED
DI
31
16
0
63
0
RBP
RESERVED
EBP
EXTENDED
BP
31
16
0


63
0
RSP
RESERVED
ESP
EXTENDED
SP
31
16
0
63
0
R8
RESERVED
R8D
EXTENDED
R8W
R8L
31
16
7
0
63
0
R9
RESERVED
R9D
EXTENDED
R9W
R9L
31
16
7
0


63
0
R10
RESERVED
R10D
EXTENDED
R10W
R10L
31
16
7
0
63
0
R11
RESERVED
R11D
EXTENDED
R11W
R11L
31
16
7
0
63
0
R12
RESERVED
R12D
EXTENDED
R12W
R12L
31
16
7
0



You can access the small portions of registers in an assembly code. The concept of accessing and using these small portions becomes clear in shellcoding when you want your shellcode to be as small as possible.

Here are some instructions for you but before you begin, you must know the basic syntax of an assembly instruction. We have 2 different notations of assembly, Intel notation and AT&T notation.

In Intel notation after the instruction, first the destination is mentioned followed by a comma and the source.

instruction destination, source

In AT&T notation after the instruction, first comes the source followed by a comma and then the destination. Every register has a percent sign(%) appended to the beginning of it. It looks like this:

instruction %source, %destination

*** NOTE: The percent sign is only applied to the registers. It is not applied to the immediate values.

NOP

YES! The first instruction for you to learn is NOP. NOP stands for No Operation. Better to wipe that smile off your face and tell me what NOP does. Ha? Nothing? Well, you're wrong! NOP actually does something. A NOP instruction is like this:

XCHG eax,eax

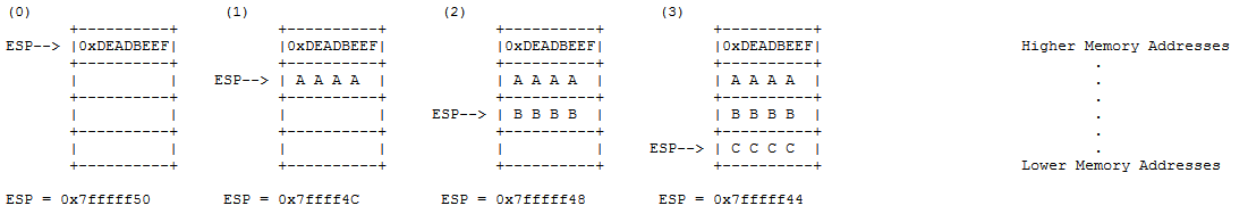
It conceptually does nothing, but behind the scene it exchanges (XCHG as you guessed) the value in EAX with EAX.

PUSH

PUSH instruction pushes either a byte, a word, a dword or a quadword onto the stack. For this part of tutorial, I will only explain pushing a dword (4-byte value) onto the stack. The rest of them takes seconds to understand. In order to fully understand what a push instruction does, you have to see it by demonstration. For the following instructions:

(1) PUSH 0x41414141
(2) PUSH 0x42424242
(3) PUSH 0x43434343

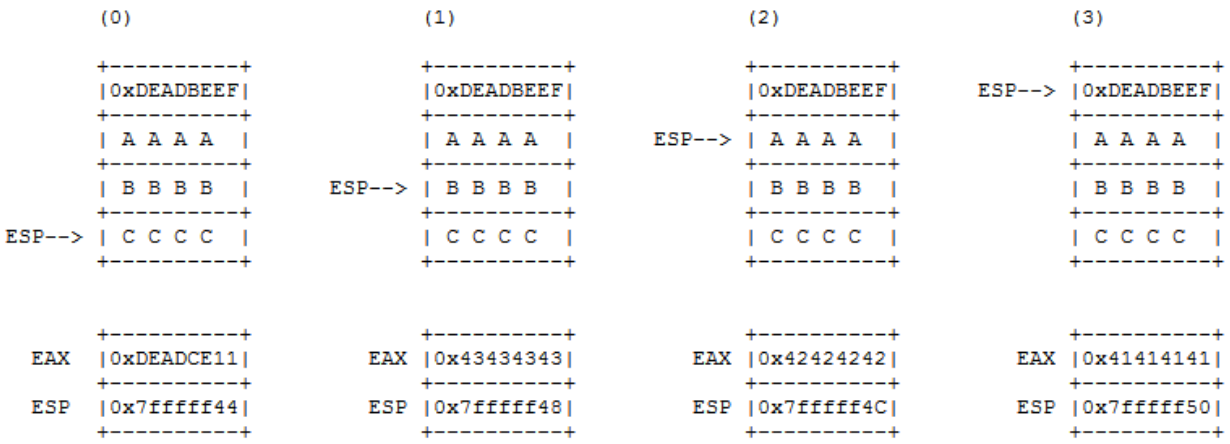
Consider ESP points to some address that holds the content 0xDEADBEEF(0) before executing the 3 lines above. After the execution of each PUSH instruction, ESP gets decremented by 4 and the value will be pushed on to the stack and the new ESP will point to it.



POP

POP is an instruction that is exactly the opposite of a PUSH instruction. It pops(moves) whatever value that ESP is currently pointing at to another register and will increment ESP by 4(in the case of a DWORD). If you look at the demonstration below, assuming EAX holds the value 0xDEADCE11 before the execution of the following 3 lines by issuing a PUSH EAX instruction(1). The current value at the address that ESP is pointing to at the time (CCCC or 0x43434343) will be popped off the stack and it will show up in the EAX register and ESP will be incremented by 4. Notice that popping values off the stack will not completely destroy the popped value. It just moves it to the register as the instruction defines and adds 4 bytes to ESP.

(1) POP EAX
(2) POP EAX
(3) POP EAX



*** POP DWORD to a REGISTER

CALL

One of the most important instructions for you to understand is the CALL instruction and its conventions. Understanding the calling conventions is crucial in the field of reverse engineering. So before I tell you what happens while executing a CALL instruction, let's dive into the calling conventions themselves. The calling conventions define how the code calls a function(subroutine) and how the parameters are passed to the function. It is mostly dependent on the compiler and it can be configured to use a certain convention. But there are few of them and the most commonly used ones are CDECL and STDCALL conventions.

CDECL

"C Declaration" is the most commonly used convention for all C code and some C++. In CDECL, the caller must push the parameters of the function that are gonna be called(callee) onto the stack from right to left. So for example if we have this function in C:

```
func (int a, int b){
      ...
      ...
}

int main (){
      func(100,200);
      int var = 300;
      return 0;
}
```

The value "b" and then "a" must be pushed onto the stack (right to left) before calling "func". After calling the function "func", callee(func) must save the previous stack frame pointer and create a new stack frame. Wait a minute! WTF? What's a stack frame? Oops! I forgot to tell you that. (:D)

Here, I will explain it now. Each function has its own stack frame. A stack frame is simply (by convention) an area that is every function's playground in order to store local variables, etc. By calling a function, after passing the parameters, the called function must set up its own new stack frame by executing 2 simple instructions as below:

(1) PUSH EBP
(2) MOV EBP,ESP

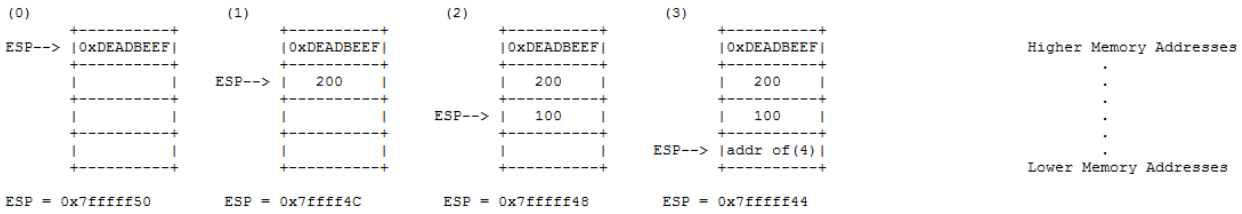
Line(1) saves the current stack pointer onto the stack, then on line(2) it will copy it to the EBP register which always points to the bottom (start of) the stack. Both EBP and ESP hold the same value. Then after the function starts executing its main functions, ESP will point somewhere lower than EBP (a frame full of local variables, etc.). A CALL instruction will push the address of next instruction just after the CALL instruction onto to the stack and will change the EIP with the address of the first line of the function's code section. Here's a demonstration for you to see the whole picture:

.
.
.

(1) PUSH 0xC8
(2) PUSH 0x64
(3) CALL func
(4) PUSH 0x12C

.
.
.

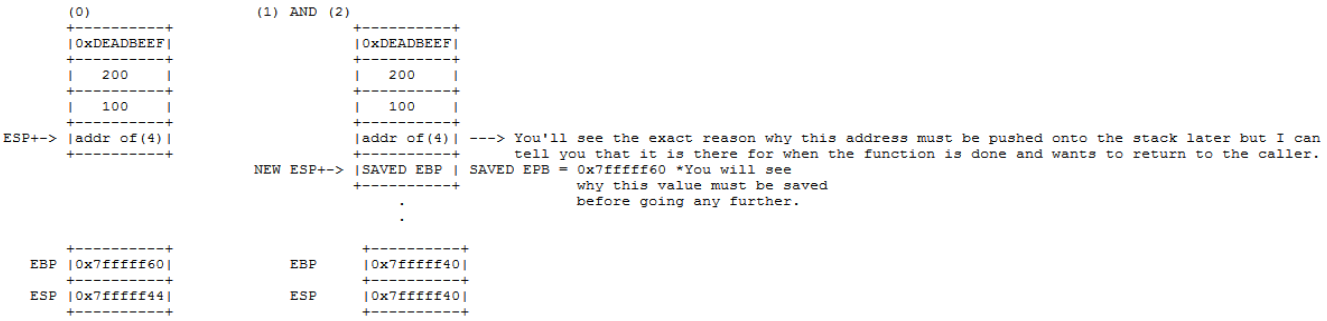
Before the call:



After the call:

func:
(1) PUSH EBP
(2) MOV EBP,ESP

.
.
.



In CDECL calling convention, function's return value will be put in EAX or EDX:EAX for primitive data types and after returning, caller is responsible for cleaning up the stack. So here we wrap it up in the list below:

1. Most common calling convention for all C code and some C++ code.
2. The called function (the callee) expects its parameter to be pushed onto the stack from right to left.
3. First thing that the callee does is saving the old stack frame(PUSH EBP) and setting up a new one (MOV EBP,ESP). This procedure is called "Function Prologue".

4. Returns data in EAX or EDX:EAX registers.
5. Caller is responsible for cleaning up the stack.

STDCALL

The only difference between STDCALL and CDECL is that in STDCALL, the callee is responsible for cleaning up the stack. This calling convention is mainly used by Microsoft C++ code (e.g. WIN32 API).

RET

We have 2 forms of Return Instruction:

1. It translates to the instruction "POP EIP". It means it pops whatever value is on top of the stack and puts it into EIP. This method is used by a CDECL convention as caller is responsible for the stack clean-up.

2. It does exactly as number 1, plus it increments ESP by a given value(i.e. "RET 0x08" in the previous demonstration) after reverting the previous stack frame, it pops the value pointer by ESP(address of (4)) and then increments ESP by 8 which will remove the arguments b and a that were pushed on to the stack before. If you pay close attention, you will recognize that this

action represents a STDCALL convention where the callee is responsible for cleaning up the stack.

*** Note: In terms of exploitation, specifically buffer overflows, changing the return address (address of (4) in above demonstration) means gaining control of the EIP register which holds the key to the program's execution path.

MOV

A MOV instruction simply copies from source to destination(notice that you have this in the background process of your brain since we saw it when explaining the stack and calling conventions). We can move data in 3 different ways:

1. Register to Register
2. Memory to Register / Register to Register
3. Immediate to Register / Immediate to Memory

As you it guessed the MOV instruction can't move data from memory to memory. The memory addresses in most of the assembly instructions are used in a way called r/m32 which will be explained.

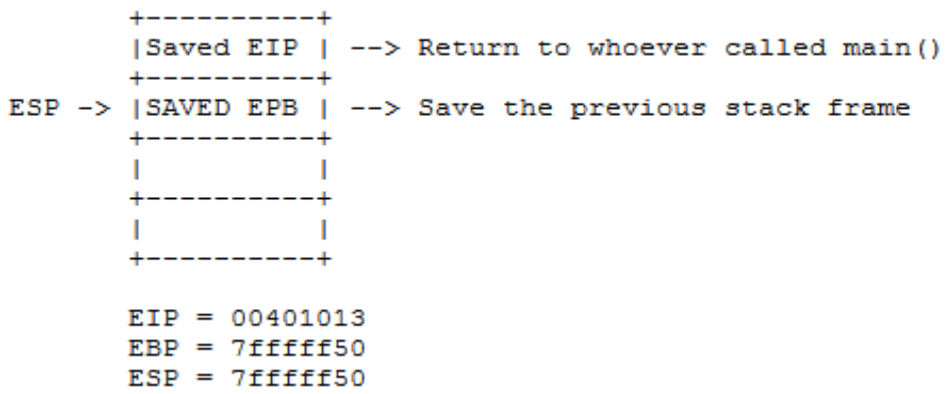
Now let's take a look at a very simple piece of code:

```
example1.c                              sub:                          main:
----------------------------------------+-----------------------------+------------------------------+
        int sub(){                      | 00401000 push ebp           | 00401010 push ebp            |
          return 0xbeef;                | 00401001 mov ebp,esp        | 00401001 mov ebp,esp         |
        }                               | 00401003 mov eax,0xBEEF     | 00401013 call sub(401000h)   |
        int main(){                     | 00401008 pop ebp            | 00401018 mov eax,0xF00D      |
          sub();                        | 00401009 ret                | 0040101D pop ebp             |
          return 0xf00d;                |                             | 0040101E ret                 |
        }                               |                             |                              |
----------------------------------------+-----------------------------+------------------------------+
```

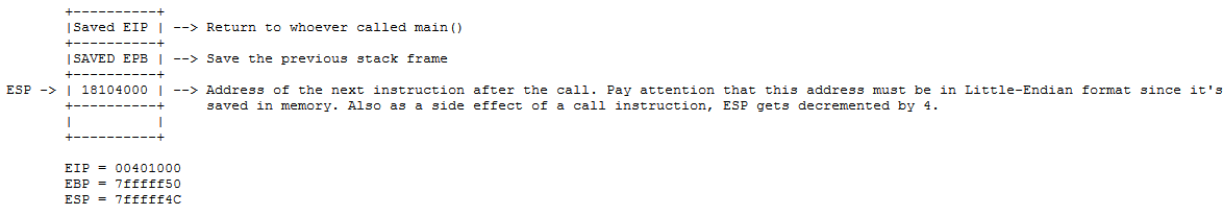
*** Note: This piece of code is compiled without any optimization and security protection. Your assembly instruction may look different but don't
worry because this example serves the educational purpose as we have it in the simplest way. You will see more complicated and up-to-date instructions
as you go along in this book.

If we assume that the first thing that's gonna start executing is main(), this piece of code is gonna call the function sub() and then sub() is
gonna return the hex value 0xBEEF and main is not gonna use it in anyway and return 0xF00D and exit.

In assembly code, we assume the entry point of our program is main(). The first 2 instructions are the function prologue as we discussed before. It saves the previous stack frame (PUSH EBP). This is done based on a simple fact that main() is not the first function that is called to start executing. There are tons of them, you can check it in gdb if you're interested but for now we assume main() is the entry point. Later it creates its own stack frame (MOV EBP,ESP). After executing those 2 lines, the stack should look like this:



Now when the call Instruction is gonna execute, the address of the very next instruction after the call instruction in main() is gonna get pushed on to the stack which in this case is 0x00401018 (MOV EAX,0xF00D) and EIP will be point to the first instruction in sub() which is 0x00401000 (PUSH EBP). See when that happens the stack will look like this:



The only thing that sub() does is returning 0xBEEF. As it was mentioned before, the EAX register is mostly used for the function's return value. After executing the function's prologue(PUSH EBP and MOV EBP,ESP), the hex value 0xBEEF is gonna be put in EAX. Here's how the stack will look like:

```
        +-----------+
        |Saved EIP  | --> Return to whoever called main()
        +-----------+
        |SAVED EPB  | --> Save the previous stack frame
        +-----------+
        | 18104000  | --> Address of the next instruction after the call. Pay attention that this address must be in Little-Endian format since it's
        +-----------+     saved in memory.
 ESP -> | 50ffff7f  | --> Previous EBP(stack frame) will be push on to the stack and ESP will get decremented as a side effect of PUSH instruction.
        +-----------+     This address is also saved in memory in Little-Endian format.
        |           |
        +-----------+

        EIP = 00401008
        EBP = 7fffff48
        ESP = 7fffff48
        EAX = 0000BEEF
```

Now the return value of the function sub() has been put in EAX register, it's time get back to main(). The next instruction to execute is the POP EBP instruction. As mentioned before, a POP instruction, gets whatever value that ESP currently points at and puts in in the register that is written in front of it. ESP currently has the value of 0x7FFFFF48 which points to the value 50FFFF7F(Little-Endian). So after executing POP EBP instruction, the stack will look like this:

```
        +-----------+
        |Saved EIP  | --> Return to whoever called main()
        +-----------+
        |SAVED EPB  | --> Save the previous stack frame
        +-----------+
ESP --> | 18104000  | --> Address of the next instruction after the call. We will use this to return the execution to main. As a side effect of the POP
        +-----------+     instruction, ESP is incremented by 4.
        | 50ffff7f  | --> Previous EBP (stack frame) will be popped off the stack and gets put in the EBP register. This value will not be completely
        +-----------+     wiped off the stack but the program has nothing to do with it and it's the OS' concern not us.
        |           |
        +-----------+

        EIP = 00401009
        EBP = 7fffff50
        ESP = 7fffff4C
        EAX = 0000BEEF
```

Now we got back to our previous stack frame by popping the saved EBP back to the EBP register, it's time to go back to main. When RET is executed, what's gonna happen is that whatever value that ESP currently points at is gonna pop off the stack and appear in the EIP register. So here is the stack after executing the RET instruction:

```
        +-----------+
        |Saved EIP  | --> Return address to whoever called main()
        +-----------+
ESP --> |SAVED EPB  | --> ESP point here after executing the RET instruction.
        +-----------+
        | 18104000  | --> Address of the next instruction after the call. We will use this to return the execution to main. As a side effect of the
        +-----------+     POP instruction, ESP is incremented by 4.
        |undefined  |
        +-----------+
        |           |
        +-----------+

        EIP = 00401018
        EBP = 7fffff50
        ESP = 7fffff50
        EAX = 0000BEEF
```

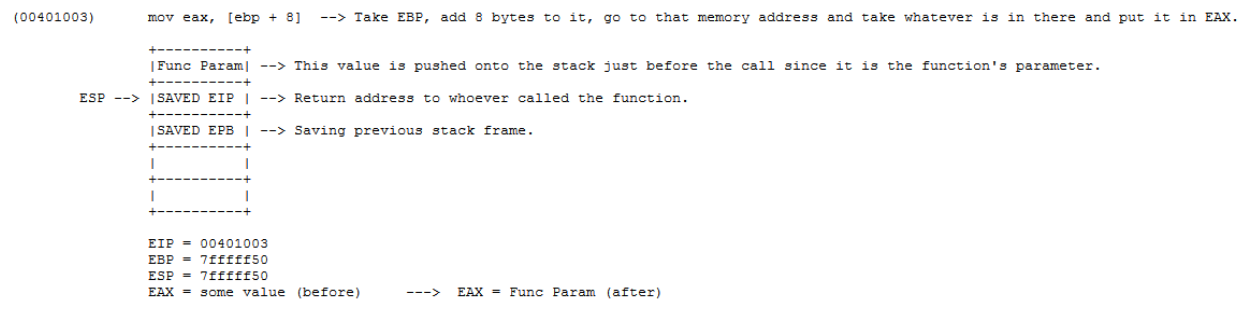
EIP points to the instruction just after the CALL sub() instruction which is MOV EAX,0xF00D. Now after executing this, the EAX register will hold the value 0xF00D and the stack will remain the same. Now what's gonna happen after executing the RET instruction in main()? The same thing we saw in sub(). The saved EBP(previous stack frame before calling main()) will

be popped off the stack and EBP will be reset again to that value. Then RET will put the saved EIP value into EIP and decrement ESP by 4 and return to whatever function it was at(probably a kernel module, I don't know).

Well that was fairly easy but it was a fairly good example for you to understand how calling and returning from calls work. Before we jump to our next example, here I introduce you to R/M32:

Whenever you see the term R/M32 in Intel's manual or such, it means it can get the value you're looking for using a combination of a register pointing to a memory location plus some offset or optionally a scale multiplier. I guess you may be doing your WTF gesture now (:D). What that means is that you specify a register that points to a memory location that your program needs and based on that address you may add some offset to it to access the exact value you want. For example, imagine after calling a function, that function wants to move some value from previous stack frame to some register to work with. That actually happens every time a function wants to access the parameters passed to it. If you remember as mentioned before, right before a call instruction, the parameters passed to the function must be pushed onto the stack from right to left. So when the called function wants to access those parameters, if we assume that the function's prologue is executed and the very first instruction after it is to return its parameter back (just for the sake of simplicity), that instruction would be as follows:

(00401003) mov eax, [ebp + 8] --> Take EBP, add 8 bytes to it, go to that memory address and take whatever is in there and put it in EAX.



Now one thing you may have noticed is the brackets. So here's a rule which applies 99% of the time you see a register inside brackets:

A register(plus the index or scale) simply means that: go to the memory address at that location and get whatever actual value is in there and do whatever is asked for. We need the content in that memory address, not the address itself.

So if we sum up R/M32 it would be:

*[Base index*scale + displacement]*

Where Base is a register such as EAX, EBX, ESP, EBP, etc. and index again is another register multiplied by a scale plus the displacement(offset). In the above example, we only used Base plus displacement which happens to be the most common R/M32 form you will see. Remember that all of these parts in the brackets are optional which means that you can put a hardcoded memory address inside the bracket (i.e. [7FFFFF58]).

Here are some new instructions for you to continue to the next example:

ADD

Fairly easy, right? It takes the source and adds it to the destination and puts the final value in the destination. For example:

add eax,0x10 ---> will add decimal value 16(or hex 10) to EAX and
updates EAX with the result.

SUB

Exactly like the ADD instruction but it does subtraction instead of addition.

LEA

Remember when I told you 99% of the time when you see R/M32 and the brackets, it means go to the memory address and get the actual content not the memory address? Well that 1% applies to the LEA instruction.

lea eax,[ebp + 8]

LEA (Load Effective Address) means that the program only needs the actual address not the content. When you get the address, put it (load it) in the register specified as destination. The above example means take the address in EBP and add it by 8 and put the result (which is an address not the content pointed at by it) and put it in EAX.

SHL

In order to understand SHL or Shift Logical Left we need to use a number as an example and work with it. Imagine you have the instructions below which moves the hex value of 10 (or 16 decimal) in EAX and the does a SHL on EAX by 1:

mov eax, 0x10
shl eax, 1

if we turn 0x10 to binary it would be this:

Decimal (base 10)	Binary (Base 2)	HEX (Base 16)
16	00010000	0x10

SHL is a bitwise operation which means it deals with the binary format data. We take the binary value 00010000 and shift it once to left. The effect of this action is shown below:

```
00010000   ------>  0010000[]  --->   final result: 00100000 ----> To decimal:  32  ---> To hex: 0x20
   ^      shifted      ^    ^
            left           |___ the least significant bit must be filled with zeros
```

I think after seeing the result, you may have guessed that SHL multiplies an integer by 2 when the amount of shifting required is 1. So what about when it's 2 or 3? YES! It multiplies by 2 to the power of the amount of shifting required.

shl register, n ---> register = register x 2^n

SHR

SHR or Shift Logical Right is exactly the same as SHL but it shifts the bits to right. As you probably guessed, it means division by powers of 2.

mov eax, 0x10
shr eax, 2

After executing the instructions above, what would be the value of EAX? (DIY)

AND

AND instruction and the next 2 instructions that you'll see are very useful in terms of stack alignment, addressing, shellcoding and encoding the shellcode that you want to shove into the stack (or heap or other locations when you use an egghunter). AND instruction takes the source and the destination, turns them into binary, and performs a bit-by-bit AND operation and puts the result in destination register. If you don't remember how the AND operation works, here's a refresher:

A	B	A & B
0	0	0
1	0	0
0	1	0
1	1	1

If we have the instructions below:

mov eax, 0x12345678
and eax, 0x45454545

0x12345678 -----> 0001 0010 0011 0100 0101 0110 0111 1000
0x45454545 -----> 0100 0101 0100 0101 0100 0101 0100 0101
AND ----------
0x00044440 <----- 0000 0000 0000 0100 0100 0100 0100 0000

The new value which is 0x44440 will be put in EAX. The AND instruction accepts a register or an R/M32(or an R/M64 for 64-bit) form as destination. As a source, you can use an immediate, a register or an R/M32(or R/M64).

OR

The OR instruction like AND instruction performs a bitwise operation. You can check the table below as a refresher on the OR operation:

A	B	A or B
0	0	0
1	0	1
0	1	1
1	1	1

If we have the instructions below:

mov eax, 0x12345678
and eax, 0x45454545

0x12345678 -----> 0001 0010 0011 0100 0101 0110 0111 1000
0x45454545 -----> 0100 0101 0100 0101 0100 0101 0100 0101
AND ---------- ----------
0x5775577D <----- 0101 0111 0111 0101 0101 0111 0111 1101

XOR

XOR or "eXclusive OR" is a very useful instruction for zeroing out registers in terms of shellcoding. The XOR operation is slightly different than the OR and AND operations.

A	B	A xor B
0	0	0
1	0	1
0	1	1
1	1	0

Imagine we're trying to encode our shellcode and we need a clean state of the EAX register. You may say "Hah! I just do a MOV EAX,0" but are you sure that works? Think again! You can't pass a Null-Byte(0x00) to your buffer so that is a big problem if you don't know about XOR. Here's how we can zero out the EAX register with a bit of creativity:

xor eax, eax

That's all it takes. Remember the result of an XOR operation is 1 if and only if the operands are different (look again at the table above). So XOR-ing a value with itself will always result in zero.

Now is the time for our second example but this time I want you to do some exercises step by step. I'm using an Ubuntu VM version 16.04.1 LTS (on VMware). First of all, here's the code in C which we are going to rip apart:

example2.c

```
#include <stdlib.h>
int sub(int x, int y){
        return 2*x+y;
}

int main(int argc, char ** argv){
```

```
        int a;
        a = atoi(argv[1]);
        return sub(argc,a);
}
```

Compile and build the program using clang (not GCC) just for convenience:

bash#> clang -o example2 example2.c -m32 -fno-stack-protector -Wl,-z,relro,-z,now,-z,noexecstack -static

Now you need to create this file named cmd in order to make it easy for us while debugging and checking registers:

```
bash#> cat assembly/cmd
        display/10i $eip
        display/x $eax
        display/x $ebx
        display/x $ecx
        display/x $edx
        display/x $edi
        display/x $esi
        display/x $ebp
        display/16xw $esp
        break main
```

using our previous cmd file:

```
bash#> gdb -x cmd example2
```

We configure the debugger to use Intel syntax for the code

```
(gdb) set disassembly-flavor intel
```

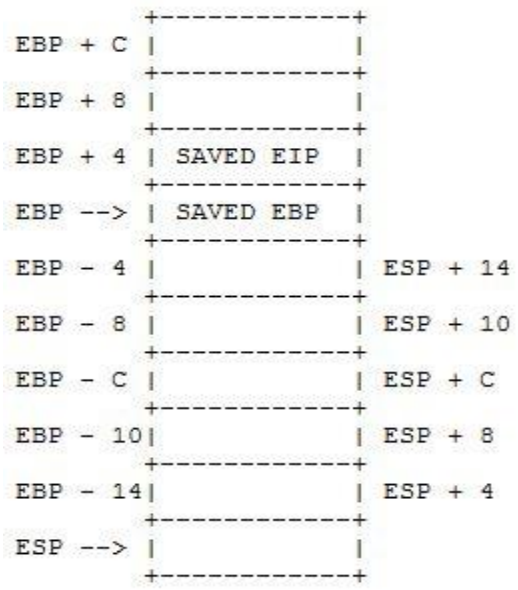
Now issuing the commands:

```
(gdb) disassemble main
Dump of assembler code for the main function:
   0x080488a0 <+0>:     push   ebp
   0x080488a1 <+1>:     mov    ebp,esp
   0x080488a3 <+3>:     sub    esp,0x18
   0x080488a6 <+6>:     mov    eax,DWORD PTR [ebp+0xc]
   0x080488a9 <+9>:     mov    ecx,DWORD PTR [ebp+0x8]
   0x080488ac <+12>:    mov    DWORD PTR [ebp-0x4],0x0
```

```
0x080488b3 <+19>:    mov    DWORD PTR [ebp-0x8],ecx
0x080488b6 <+22>:    mov    DWORD PTR [ebp-0xc],eax
0x080488b9 <+25>:    mov    eax,DWORD PTR [ebp-0xc]
0x080488bc <+28>:    mov    eax,DWORD PTR [eax+0x4]
0x080488bf <+31>:    mov    DWORD PTR [esp],eax
0x080488c2 <+34>:    call   0x804d890 <atoi>
0x080488c7 <+39>:    mov    DWORD PTR [ebp-0x10],eax
0x080488ca <+42>:    mov    eax,DWORD PTR [ebp-0x8]
0x080488cd <+45>:    mov    ecx,DWORD PTR [ebp-0x10]
0x080488d0 <+48>:    mov    DWORD PTR [esp],eax
0x080488d3 <+51>:    mov    DWORD PTR [esp+0x4],ecx
0x080488d7 <+55>:    call   0x8048880 <sub>
0x080488dc <+60>:    add    esp,0x18
0x080488df <+63>:    pop    ebp
0x080488e0 <+64>:    ret
End of the assembler dump.
(gdb) disassemble sub
Dump of assembler code for the sub function:
0x08048880 <+0>:     push   ebp
0x08048881 <+1>:     mov    ebp,esp
0x08048883 <+3>:     sub    esp,0x8
0x08048886 <+6>:     mov    eax,DWORD PTR [ebp+0xc]
0x08048889 <+9>:     mov    ecx,DWORD PTR [ebp+0x8]
0x0804888c <+12>:    mov    DWORD PTR [ebp-0x4],ecx
0x0804888f <+15>:    mov    DWORD PTR [ebp-0x8],eax
0x08048892 <+18>:    mov    eax,DWORD PTR [ebp-0x4]
0x08048895 <+21>:    shl    eax,0x1
0x08048898 <+24>:    add    eax,DWORD PTR [ebp-0x8]
0x0804889b <+27>:    add    esp,0x8
0x0804889e <+30>:    pop    ebp
0x0804889f <+31>:    ret
End of the assembler dump.
```

***Note: If you need some guidance or tutorial on using gdb or linux, look for it elsewhere, not here!!

Alright! We start analyzing the assembly line by line and match it with our high-level C code. Starting with line <+0> of main(), we can see it's saving the old stack frame and creating its own. Then it pushes ESP down by issuing a SUB ESP,0x18 on line <+3>. In order to make sense of that, we convert 0x18 to decimal which gives us 24. So it is reserving 24 bytes on the stack(why?). Here is how the stack looks like:

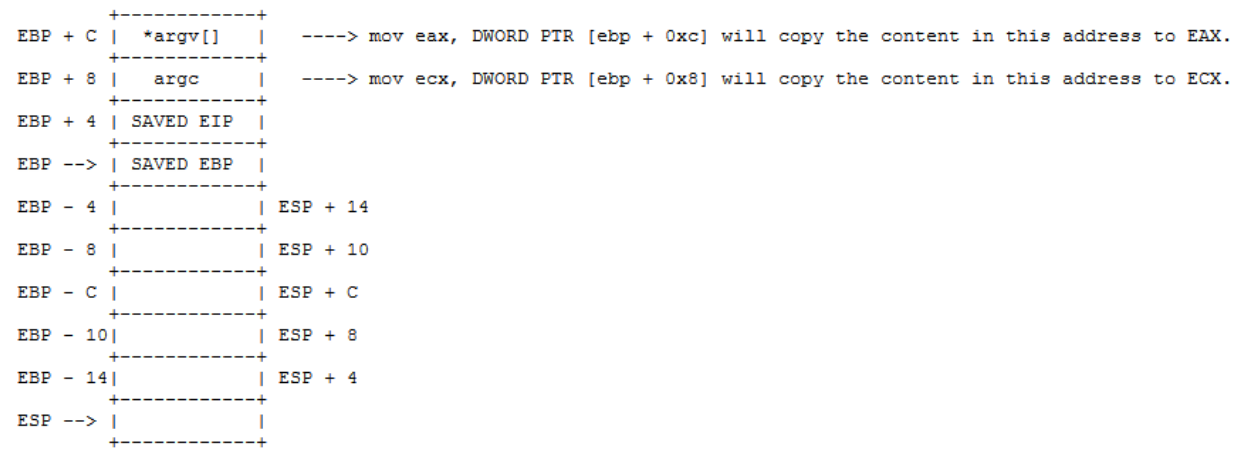


Then we have these instructions:

mov eax,DWORD PTR [ebp+0xc]
mov ecx,DWORD PTR [ebp+0x8]

** NOTE: DWORD PTR is Double-Word Pointer. You can just ignore it!

They are asking for some values beyond our current stack frame. Looks weird? No! Remember the calling conventions and how the parameters of a function get pushed onto the stack? That's exactly it and it's asking for main()'s parameters argc and *argv[]. One important note here is that whenever you see EBP plus something, 99% of the time it's referring to some parameters passed for the called function. In this case, some function called main() and main() is asking for its parameters argc and argv. So we complete our picture of the stack:



EAX = Pointer to the start of argv[] array

ECX = Holds the number of command line arguments when executing the program in the command line. For example:

bash#> ./program 100

has 2 command line arguments.

Line <+12> corresponds to the first line in main(). The uninitialized variable "a" gets a NULL(0x00) value. Quickly note that different compilers
and architectures handle the uninitialized variables differently. For example they may end up in HEAP or even don't end up anywhere until they get
initialized.

On lines <+19> and <+22> of main, those command-line arguments that we saw previously which were moved to EAX and ECX, now get copied onto the stack. This time relative addressing of EBP minus something is used. This type of addresses(EBP minus something) 99% of the time represent local variables and the function's own variables and procedures. SO after these 2 line:

mov DWORD PTR [ebp-0x4],ecx
mov DWORD PTR [ebp-0x8],eax

the stack will look like this:

```
        +-------------+
EBP + C |  *argv[]    |   ----> mov eax, DWORD PTR [ebp + 0xc] will copy the content in this address to EAX.
        +-------------+
EBP + 8 |   argc      |   ----> mov ecx, DWORD PTR [ebp + 0x8] will copy the content in this address to ECX.
        +-------------+
EBP + 4 | SAVED EIP   |
        +-------------+
EBP --> | SAVED EBP   |
        +-------------+
EBP - 4 |int a = argc| ESP + 14   ***Note: this address was first filled with 0x00 and then filled with the value of argc.
        +-------------+
EBP - 8 | ECX = argc | ESP + 10
        +-------------+
EBP - C |EAX= *argv[]| ESP + C
        +-------------+
EBP - 10|             | ESP + 8
        +-------------+
EBP - 14|             | ESP + 4
        +-------------+
ESP --> |             |
        +-------------+
```

Line <+25> gets the pointer to the start of argv[] array and puts that address in EAX and on line <+28> it says: go to the start of the array, then go 4 bytes forward and take whatever value is in there and put it in EAX. If we picture it, it would make more sense:

<+25>: mov eax,DWORD PTR [ebp-0xc]
<+28>: mov eax,DWORD PTR [eax+0x4]

```
argv[] --> {./example2,0x100,.....}
           ^           ^
           |           |__ the command-line argument that we pass to the program. As an example, here we used the number 256
           |__ start of argv array
```

bash#> ./example2 256

argv[0] is ./example2
argv[1] is 0x100 or 256

I hope it's clear now ;D

Then on line <+31> just before we call atoi() funtion, we need to push its parameter onto the stack which is argc.

<+31>: mov DWORD PTR [esp],eax

After that we call atoi on line <+34> which makes our stack like this:

```
          +-------------+
EBP + C |  *argv[]    |   ----> mov eax, DWORD PTR [ebp + 0xc] will copy the content in this address to EAX.
          +-------------+
EBP + 8 |   argc      |   ----> mov ecx, DWORD PTR [ebp + 0x8] will copy the content in this address to ECX.
          +-------------+
EBP + 4 | SAVED EIP   |
          +-------------+
EBP --> | SAVED EBP   |[]
          +-------------+
EBP - 4 |int a = argc| ESP + 14   ***Note: This address was first filled with 0x00 and then filled with the value of argc.
          +-------------+
EBP - 8 | ECX = argc | ESP + 10
          +-------------+
EBP - C |EAX= *argv[]| ESP + C
          +-------------+
EBP - 10|             | ESP + 8
          +-------------+
EBP - 14|             | ESP + 4
          +-------------+
ESP --> |    argc     |
          +-------------+
```

We're not gonna go through atoi() function right now. As we learned before, it's gonna pick up its parameter from [EBP + 0x8] (note that after SAVED EIP is on address [EBP + 0x4]) and return the value in EAX and use the saved EIP to get back to main(). Then the returned value in EAX will be placed onto stack at [EBP - 0x10]. Then on line <+42> pointer to argv[] is copied to EAX and on line <+45> the integer value of argc is copied to ECX. These values EXA,ECX will be placed at [ESP] and [ESP + 0x4] respectively to be ready to call sub().

```
        +-------------+
EBP + C |  *argv[]    |   ----> mov eax, DWORD PTR [ebp + 0xc] will copy the content in this address to EAX.
        +-------------+
EBP + 8 |   argc      |   ----> mov ecx, DWORD PTR [ebp + 0x8] will copy the content in this address to ECX.
        +-------------+
EBP + 4 | SAVED EIP   |
        +-------------+
EBP --> | SAVED EBP   |
        +-------------+
EBP - 4 |int a = argc| ESP + 14   ***Note: this address was first filled with 0x00 and then filled with the value of argc.
        +-------------+
EBP - 8 | ECX = argc | ESP + 10
        +-------------+
EBP - C |EAX= *argv[]| ESP + C
        +-------------+
EBP - 10| int argv[1]| ESP + 8
        +-------------+
EBP - 14|    argc    | ESP + 4
        +-------------+
ESP --> | int argv[1]|
        +-------------+
```

Now we have everything set up to call sub(). After calling sub() and past the function's prologue, it reserves 8 bytes for its stack frame. On line <+6> and <+9> it takes the local variables from previous stack frame(main()'s) and copies them in EAX and ECX respectively. Then it places those values onto its own stack frame at [EBP - 0x4] for ECX and [EBP -0x8] for EAX. Then again on line <+18> it places the value of [EBP - 0x4] which is integer value of 2 in EAX. Then on line <+21> we have a shift logical left instruction with shifting of one bit which translates to multiply by 2 to the power of one and then the next line of instruction, the value of [EBP - 0x8] which is 256(the imaginary value we passed on command line) will be added to EAX. The final result will be in EAX and returned to main().

```
                  +-------------+
          EBP + C |    argc     |
                  +-------------+
          EBP + 8 | int argv[1]|
                  +-------------+
          EBP + 4 | SAVED EIP   |
                  +-------------+
          EBP --> | SAVED EBP   |
                  +-------------+
          EBP - 4 | int argv[1]|
                  +-------------+
ESP -->   EBP - 8 |    argc     |
                  +-------------+

                  EAX = 2*2+256 = 260
```

Now it's time to return to main(). Knowing that sub's return value of 260 in in EAX, it wipes down its stack frame in line <+27> by adding 8 to ESP. Then it pops the value in ESP(after adding 8 to ESP) which is the SAVED EBP into EBP register. Then it uses the SAVED EIP to return to main() (line <+60>).

The same story fits to the remaining instruction in main. It wipes up the stack frame by adding 0x18(24 decimal) to ESP. Pops SAVED EBP back
to EBP register and returns to whoever called main().

Now that we successfully analyzed our second example, it's a good time to introduce you to concept of jumps. There are some ways to change EIP's value in order to change the execution path to somewhere else in the program.

One way of doing that is by using a CALL instruction as mentioned earlier. Remember that a CALL will push the address of the next instruction onto the stack which was referred to as SAVED EIP in our demonstrations of stack. A JUMP instruction won't do that. It simply changes the EIP value and this cause the execution to "jump" to that address and follow execution. We have different kinds of jump and each of them are a convenient way to assemble a control flow or a loop in a high-level language. JUMPs are categorized into 2 groups: Conditional and Unconditional.

Unconditional Jump

JMP

A JMP instruction is an unconditional jump. All it does is simply changing the value of EIP with the given address. For example, the piece of code below will simply change the EIP value with a hardcoded address:

```
          ____ EIP before = 0xsomething
         /
JMP 0x00401000
         \______ EIP after = 0x00401000
```

Using hardcoded addresses is rarely seen and also not recommended because it highly reduces the reliability of code. Hence we use the relative form of addressing by using registers. For example the assembly code below is highly used in vanilla stack-based overflows when we want to redirect the execution flow to our injected shellcode on the stack:

```
jmp esp
```

ESP = 0x7fffff50 (start of our injected shellcode)

Or when our shellcode is at some offset to the address pointed by EAX:

```
sub eax,0x100
jmp eax
```

**NOTE: We're gonna see more in-depth discussions on stack overflows later. So don't worry if you don't have a clue what I'm talking about. Just know that the JMP instruction is an unconditional jump and will change the execution flow, no questions asked.

Conditional Jumps

A conditional jump is based on checking a single value in EFLAGS register that is the result of a comparison. For us to understand how they work, we need to dive into explaining the EFLAGS register and all of its tiny parts. Obviously that's not convenient to explain every little thing inside it. You're not reading Intel's manual anyway. So here we go:

EFLAGS Register

EFLAGS is a register that has a lot of flags! No, shit! (:D). But it is really what it's about. Flags that control certain parts of the program and its connection with the OS. These flags are the bits of the EFLAGS register which can be set to 1 or 0. These values are mostly changed after a comparison or after executing some instruction that has a direct effect on the state of these bits (flags). Here's the structure of EFLAGS register containing its different flags:

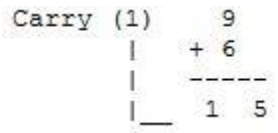
Bit	Abbreviation	Description	Category
		FLAGS (16-bit)	
0	CF	Carry Flag	Status
1		Reserved, Always 1 in EFLAGS	
2	PF	Parity Flag	Status
3		Reserved	
4	AF	Adjust Flag	Status
5		Reserved	
6	ZF	Zero Flag	Status
7	SF	Sign Flag	Status
8	TF	Trap Flag	Control
9	IF	Interrupt Enable Flag	Control
10	DF	Direction Flag	Control
11	OF	Overflow Flag	Status
12-13	IOPL	I/O Privilege Level	System
14	NT	Nested Task Flag	System
15		Reserved (Always 1 on x86)	
		EFLAGS (32-bit)	
16	RF	Resume Flag	System
17	VM	Virtual x86 Mode Flag	System
18	AC	Alignment Check	System
19	VIF	Virtual Interrupt Flag	System
20	VIP	Virtual Interrupt Pending	System
21	ID	Able to use CPUID instruction	System
22 - 31		Reserved	
		RFLAGS (64-bit)	
32 - 63		Reserved	

Well that's a long list and I don't expect you to memorize them. For now, we just have to know about these flags:

Zero Flag (ZF): This flag will be set to 1 if comparing 2 things results in equality.

Sign Flag (SF): As its name represents, it determines if you're dealing with a signed integer or an unsigned integer.

Carry Flag (CF): Remember when you were in elementary school and the teacher said if we want to add 9 to 6:



Ehem. That's the best way I can describe it. You will see it more in action later. What were we talking about? Aha! Conditional jumps.

CMP

Before learning a conditional jump, you need to know how condition is determined, right? So CMP (Compare) instruction determines that condition. It compares the source and destination value with each other and set the appropriate bits (flags) accordingly in EFLAGS register. The CF, OF, SF, ZF, AF, and PF flags are set according to the result.

Jcc

- Jump if Condition is Met

Conditional jumps have too many different instructions, each designed for a specific purpose. It's not really convenient to mention all of them here since it will not do you any good. We just mention a couple of them here with examples. If you are interested to read more about them (highly recommended), check the Intel's manual (Now you're old enough to check the manual yourself :p).

JNE

Jump if Not Equal or (JNE) checks if the Zero-Flag (ZF) is set or not. As the result of a CMP instruction sets this flag, it will take jump if ZF is not set (0) and will pass to the next instruction after the JNE instruction if ZF is set (1). Here's an example:

```
(1)  xor ecx,ecx
(2)  mov eax,0x10
(3)  inc ecx
(4)  cmp eax,ecx
(5)  jne (3)
(6)  ....
```

This piece of assembly represents a very simple and neat loop that loops 16 times. The first instruction does an "exclusive OR" on ECX. If you remember correctly, this will zero out ECX. On second line we put decimal value of 16 (hex 0x10) in EAX. After incrementing ECX by 1, we compare ECX with EAX. Comparing the values 16 and 1 is done by subtracting 1 from 16. Obviously ZF will not be set to 1 and the jump will be taken. After taking the jump, EIP will point to line (3). This will increment ECX again by 1. After looping 16 times, ECX will be 0x10 and the CMP instruction (subtracting 16 - 16) will result in 0 and ZF will be set to 1. This time the condition for JNE is not met and jump will not be take and EIP will point to line (6).

It's a good time to take a look at our 3rd example in order to better understand conditional jumps. This time we won't show you the source code first and we dive into assembly. This will be our first reversing experience. This time, our assembly is 64-bit.

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000100000f70 <+0>:     push   rbp
   0x0000000100000f71 <+1>:     mov    rbp,rsp
   0x0000000100000f74 <+4>:     mov    DWORD PTR [rbp-0x4],0x0
   0x0000000100000f7b <+11>:    mov    DWORD PTR [rbp-0x8],0x5
   0x0000000100000f82 <+18>:    mov    DWORD PTR [rbp-0xc],0x6
   0x0000000100000f89 <+25>:    mov    eax,DWORD PTR [rbp-0x8]
   0x0000000100000f8c <+28>:    cmp    eax,DWORD PTR [rbp-0xc]
   0x0000000100000f8f <+31>:    jle    0x100000fa1 <main+49>
   0x0000000100000f95 <+37>:    mov    DWORD PTR [rbp-0x4],0xffffffff
   0x0000000100000f9c <+44>:    jmp    0x100000fa8 <main+56>
   0x0000000100000fa1 <+49>:    mov    DWORD PTR [rbp-0x4],0x0
   0x0000000100000fa8 <+56>:    mov    eax,DWORD PTR [rbp-0x4]
   0x0000000100000fab <+59>:    pop    rbp
   0x0000000100000fac <+60>:    ret
End of assembler dump.
```

Okie Dockie! First two lines (line <+0> and <+1> in main) are pretty familiar; function prologue. On line <+4> through <+18> after setting up the new stack frame, we save the value zero at RBP - 4, value 5 at RBP - 8 and value 6 at RBP - 12. This pretty much looks like the process of initializing local variables. We can be sure value 5 and 6 are local variables of integer type but as we saw in our previous examples, 0 can be an uninitialized variable or something that only exists in the assembly not the source. By now we can guess that our source code might look like this:

```
main()
{
int a = 5;
int b = 6;
}
```

On line <+25> we see one of these local variables is moved to EAX followed by a compare instruction that checks these 2 local variables together. On line <+31> we see a new instruction (Here's a good time to check the manual ;p). JLE (Jump if Less than or Equal to) will check the value at RBP - 12 which is number 6 with the value in EAX which is 5. This compare instruction will change the affected flags as follows:

ZF = 0
SF = 1
OF = 0
CF = 1

JLE instruction will take the jump if and only if ZF = 1 or SF not be equal to OF (Overflow Flag). As we can see, EFLAGS register shows that this condition is met, so the jump is taken. After taking the jump we land on line <+49> which will put zero in EAX, pop the previous stack frame back into RBP and return. We now noticed that RBP - 4 (the suspicious value of 0) was indeed something in assembly only and not in the source. It was only used for our return value. Our best guess for the source code is:

```
int main() //int is used because the return values are integers
{
  int a = 5;
  int b = 6;
  if(a<b)
    return 0;
  else
    return -1;     //if JLE does not meet its condition, the jump will not be
                   //taken and the value 0xffffffff will be return which is -1
}
```

That's pretty much the source code that was used but do you want to see the real source code?

```
int main()
{
  int a = 5;
  int b = 6;
  if(a>b)
    return -1;
  else
    return 0;
}
```

The noticeable difference is that in assembly, the condition is check differently compared to the way it's checked in high level source code. This can be due to several reason

like compiler conventions, optimization, OS, etc. First "else" condition is check which is a less than or equal to b (opposite of a greater than b).

By far we know 17 different instructions. 90% of any given program is filled with these instructions that we learned thus far. But we keep pushing forward anyway. Next instruction.

IMUL

IMUL is a Signed Multiply instruction. This instruction is used when we want to multiply by any number that is not a power of 2. IMUL has three different ways to work with (Actually it's not just 3 different ways. It's 13 but we only mention 3, maybe 4. You can take a look at all 13 different ways in Intel's Manual, volume 2A - Page 3-443. BTW it's Page 3-443 not page 3 to 443 :D)

(1) --> imul r/m32 ---------> EDX:EAX = EAX x r/m32 --------> Final result 64 bits
imul r/m64 ---------> RDX:EAX = RAX x r/m64 --------> Final result 128 bits

In this way, it's gonna take the R/M32 value and multiply it by whatever is inside EAX and put the result back in to EDX:EAX in a way that EDX will contain the higher 32 bits and EAX will contain the lower 32 bits. For 64-bit version, you switch EDX and EAX with RDX and RAX respectively.

(2) --> imul reg32, r/m32 -------> reg32 = reg32 x r/m32
imul reg64, r/m64 -------> reg64 = reg64 x r/m64

You really want me to explain this one? Come on!

(3) --> imul reg32, r/m32, immediate -----> reg32 = r/m32 x immediate
imul reg64, r/m64, immediate -----> reg64 = r/m64 x immediate

Yeah, Yeah! I know! It's the first time you see an instruction with 3 operands. Well, you better look in the manual every once in a while. You may find interesting things in there :p

DIV

DIV is Unsigned Division. It has 2 forms in general.

(1) div r/m8

This will divide AX by the given r/m8 and will store the quotient in AL and the remainder in AH. These are small portions of EAX if remember, but I draw the table for you here again:

```
+-----------------+
|       EAX       |
+-----------------+
|EXTENDED|   AX   |  you take whatever inside AX, divide it by the given r/m8 value.
+-----------------+
|        |AH  | AL|  you store the result back to AX; the quotient in AL and the remainder in AH.
+--------+----+----+
31       16   8    0
```

(2) div r/m16
div r/m32
div r/m64

For r/m16 form, it takes the value DX:AX and divides it by the given r/m16 value. Then it stores the quotient in AX and the remainder in DX. If you use your imagination now, for r/m32 and r/m64 it a matter of switch DX and AX with EDX, EAX, RDX and RAX respectively.

Time for our next example. This time we're gonna see something a little bit different. And again we're only gonna see the assembly and not the source code. Don't panic. Alright, here we go:

```
(gdb) set disassembly-flavor intel
(gdb) disassemble main
Dump of assembler code for function main:
   0x0804840b <+0>:     lea    ecx,[esp+0x4]              }
   0x0804840f <+4>:     and    esp,0xfffffff0             } -----> what the hell are these???
   0x08048412 <+7>:     push   DWORD PTR [ecx-0x4]        }
   0x08048415 <+10>:    push   ebp
   0x08048416 <+11>:    mov    ebp,esp
   0x08048418 <+13>:    push   ecx
   0x08048419 <+14>:    sub    esp,0x14
   0x0804841c <+17>:    mov    DWORD PTR [ebp-0x14],0x5
   0x08048423 <+24>:    mov    DWORD PTR [ebp-0x10],0x6
   0x0804842a <+31>:    mov    eax,DWORD PTR [ebp-0x14]
   0x0804842d <+34>:    imul   eax,DWORD PTR [ebp-0x10]
   0x08048431 <+38>:    mov    DWORD PTR [ebp-0xc],eax
   0x08048434 <+41>:    sub    esp,0x8
   0x08048437 <+44>:    push   DWORD PTR [ebp-0xc]
   0x0804843a <+47>:    push   0x80484e0
   0x0804843f <+52>:    call   0x80482e0 <printf@plt>
   0x08048444 <+57>:    add    esp,0x10
   0x08048447 <+60>:    mov    eax,0x0
   0x0804844c <+65>:    mov    ecx,DWORD PTR [ebp-0x4]
   0x0804844f <+68>:    leave
   0x08048450 <+69>:    lea    esp,[ecx-0x4]
   0x08048453 <+72>:    ret
End of assembler dump.
(gdb)
```

Before we begin reversing, we need to understand the first 3 lines and their purpose. Now we start by asking ourselves, what is ESP + 0x4 ? Why is it copied into ECX? The first instruction saves an special address which is an offset to ESP and pastes it into ECX. Now if you take a look at the bottom of this chunk of assembly, on line <+69>, this address is recovered back into ESP. So we understand the first line now: it's saving the ESP value before doing

anything. But hey! Didn't we do that by the famous "Function Prologue" of "Push EBP" and "MOV EBP,ESP"? Yes buy take a look at the next instruction on line <+4>. Let's break it down:

and esp,0xfffffff0

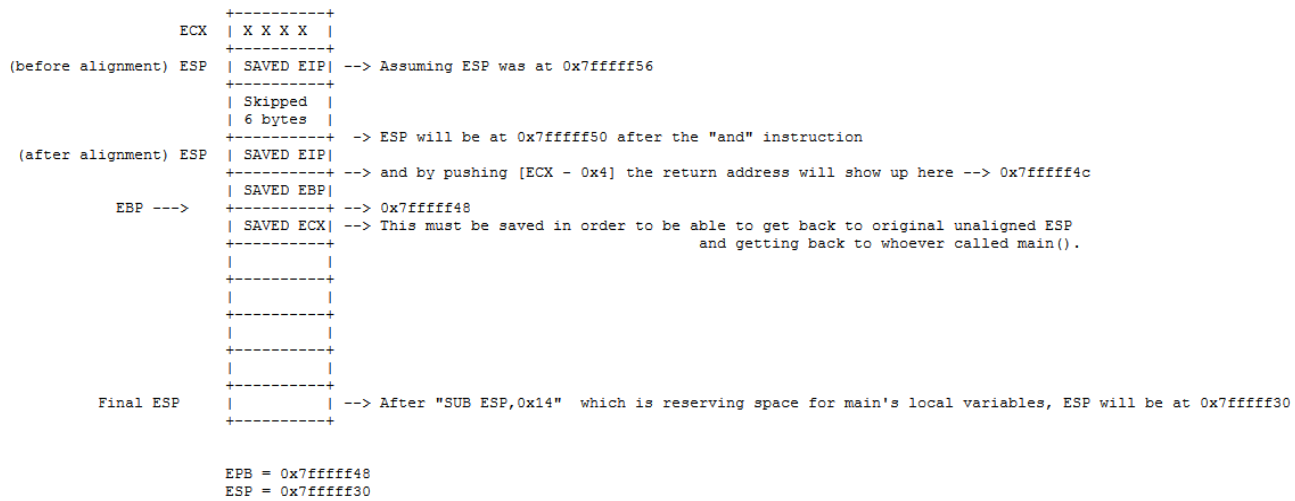
if we assume ESP is at address 0x7fffff56, this instruction will do a bitwise "AND" operation to zero out that least significant nibble in ESP.

```
and 7fffff56
    fffffff0
    ---------
    7fffff50  ---> New ESP
```

But why? Because of optimization and performance. In order to have a better performance on memory calculations, OS optimizes the stack addresses to start at a 16-byte boundary. This speeds up calculations and addressing which will result in a better performance. I think you pretty much got the idea. On line <+7>, the real ESP which is at a 0x4-byte offset to ECX, will be pushed onto the stack. If you take a look at 2-3 line after this, you see ECX is pushed onto the stack. So why pushing [ECX-0x4]? This is getting complicated right? No! The answer requires a keen memory.

[ECX-0x4] is [ESP + 0x4 - 0x4] right? So it's [ESP] which holds the value of the very next instruction after calling main(). It's the return address!

Line <+10> and <+11> is what we new before about a function prologue. Now add those 3 new lines above it to your definition on function prologue. After that, ECX is pushed onto the stack save its value if later instructions and function calls are gonna use ECX register. Maybe a stack representation would give you the full picture. So after executing line <+14> which reserves some space for main's local variables, stack should look like this:



**Note: There is an alternative good answer in this link:

http://stackoverflow.com/questions/1147623/trying-to-understand-the-main-disassembly-first-instructions

Moving on to the next instructions, on line <+17> and <+24> we see two numbers are placed on onto the stack. Number 5 will be placed at EBP - 0x14 and number 6 will be placed on EBP - 0x10. Thus far, we have 2 should have 2 local variables:

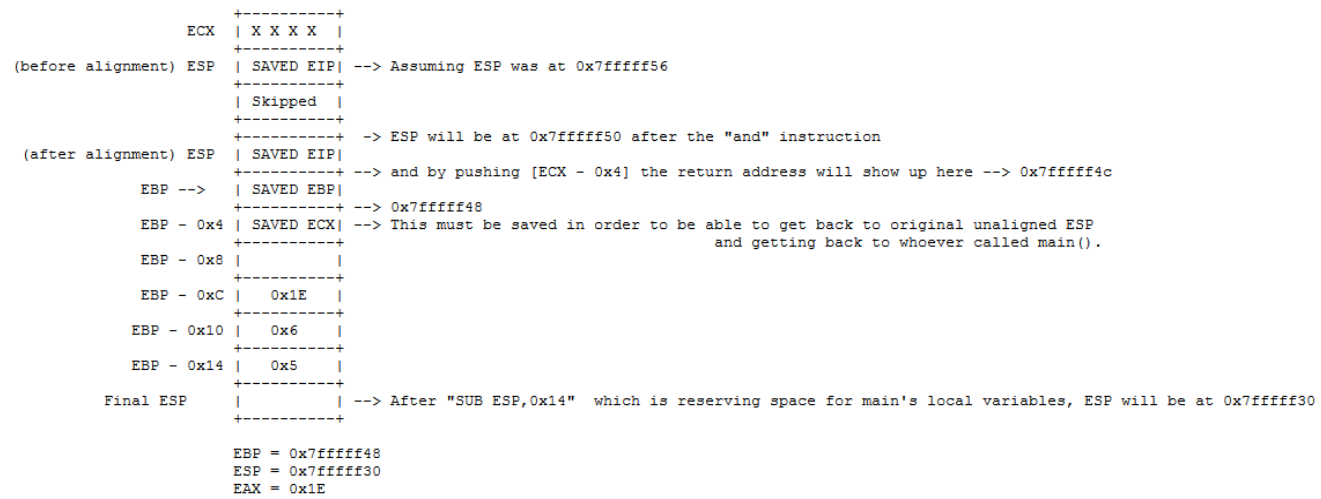
int a = 5;
int b = 6;

**Note2: We can now recognize the sequence of instructions used to define local variables.

Next, on line <+31> we see on of our local variables is copied into EAX followed by an IMUL instruction. If we check closely, we can see, variable a or number 5, is copied into EAX and then, on line <+34> we have:

imul eax, DWORD PTR [EBP-0x10]

...which means multiply whatever is inside [EBP-0x10] by EAX and put the result back in EAX. For this example, it multiplies 5 by 6 and EAX will hold the value 30. Later this value is placed onto the stack at [EBP - 0xC]. Our Stack will be like this:

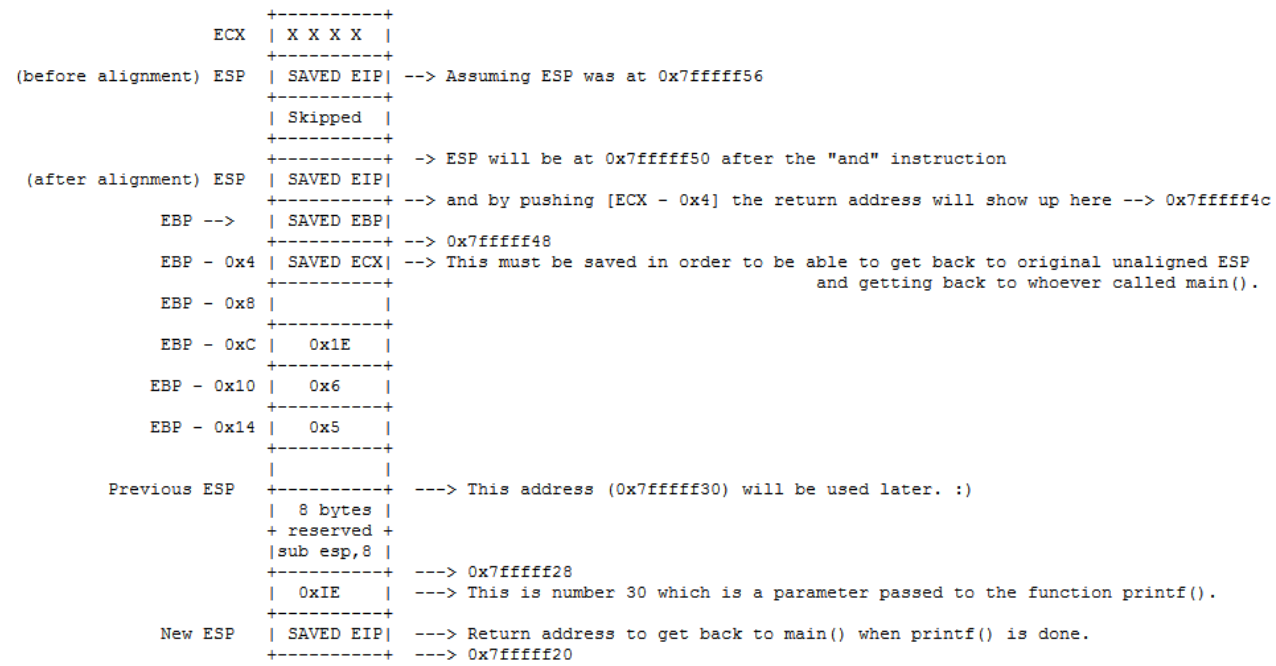


Moving on to the next instruction, we see ESP is again subtracted by 0x8. Reserving space again? Hmm... Although we have empty spaces in our stack but it is reserving more space. Let's move on and see what happens next. We see the result of the multiply, is now push onto the stack. The next two instructions after it, is pushing the address of the very next instruction after the call and then calling printf(). Now we can be sure it is going to print the value 30 on the screen.

0x08048437 <+44>: push DWORD PTR [ebp-0xc] ----> Pushing the parameter needed for the function printf() onto the stack.

0x0804843a <+47>: push 0x80484e0 ----> pointer to
the format string used with
printf()

0x0804843f <+52>: call 0x80482e0 <printf@plt> ----> Calling printf();



Rest of the instructions are closing scene. On line <+57>, we wipe off a part of the stack by adding 16 bytes to it. This will recover ESP to point to "Previous ESP". Then the value 0 is put inside EAX.

0x08048444 <+57>: add esp,0x10
0x08048447 <+60>: mov eax,0x0

Now on line <+65> the saved value of ECX will be recovered into ECX again. And then on line <+69> we use [ECX - 0x4] (ECX was a 4-byte offset to ESP, remember?) to restore the original unaligned ESP. That's it!

0x0804844c <+65>: mov ecx,DWORD PTR [ebp-0x4]
0x0804844f <+68>: leave
0x08048450 <+69>: lea esp,[ecx-0x4]
0x08048453 <+72>: ret

I noticed that I haven't told you about "leave" instruction. Leave is nothing but:

mov esp,ebp
pop ebp

which is the same as what we saw in previous examples when a function wants to restore the caller's stack frame and return. So anyways, we reversed example4 completely and here's the result we got for the high-level source code:

```
#include <stdio.h>     // ---> because we used printf()
int main()             //----> main() must be an integer because the final
                       //return value which we saw in EAX on line <+60> is an
                       //integer.
{
int a = 5;
int b = 6;
int c = a*b;           //because of the IMUL instruction on our 2 local
                       //variables above.

printf("%d", c);       // ---> because the result of the multiplication was
                       //printed to screen.

return 0;              //---> the final return value in EAX is 0.
}
```

While examining a program inside a debugger, often time we see 2 instructions that do a very simple task. These 2 instructions are introduced
here as one:

INC - DEC

Increment/Decrement instruction accepts only 1 r/m32 value and increments/decrements it by 1. That doesn't need anymore explanation :D

How about writing our "Hello World!" program in ASM now, hmm? Perhaps this is the only time you see "Hello World" at almost the end of a book when learning a new language. Well, there is a little twist here, I try to explain as clear as possible. First we need to know the architecture of a program in Assembly. There are 3 different sections (Well, the most important andgeneral ones) that forms up an assembly program:

.TEXT : .TEXT section is where all the code goes. In other words, it's the section that holds the main execution flow of the program. You
may hear folks refer to this section as "CODE" section.

.DATA : In this section, all the constants are defined. You can not change the values here throughout the execution.

.BSS : ...and all the variables reside in this section.

Alright, open up a new file and name it whatever you want and save it with ".asm" extension (I'm doing it on Ubuntu). Let's define 2 sections that we need for our "Hello World" program in the file "hello.asm", .TEXT and .DATA (Since we're only gonna use constants!):

hello.asm

```
---------------------------------------------------------------------------
SECTION .TEXT




SECTION .DATA

---------------------------------------------------------------------------
```

This is the syntax that should be used to define sections in ASM. In almost every programming language, the starting point of the program is the "main()" function. In Assembly, the starting point is "_start" (Underline start). Since this function must be globally recognized throughout whole program, we declare it as GLOBAL:

hello.asm

```
---------------------------------------------------------------------------
SECTION .TEXT
    global _start
_start:



SECTION .DATA

---------------------------------------------------------------------------
```

Now we can start writing our "Hello World" Program. Keep in mind that every function that we have in our program, must be declared in .TEXT section before doing anything (right after SECTION .TEXT). This should be done in order for the program to be linked correctly to its binary executable.

What should we do before printing to screen? Defining the string we want to print. We do that in .DATA section. The syntax for it is:

```
name_of_const    db    value          ;This is a comment! Ehem, db is
                                      ;"Define Byte" and what it does is
                                      ;that it places the value in memory
                                      ;and labels it so we can access it
                                      ;by a name.
```

Later, we need to specify the length of the constant (or string, in this case) that we defined earlier. This may be a little bit tricky:

```
name_of_const    equ  $ - name_of_const_in_question
```

...tha f**k? Yeah, I know :D. "equ" mean "=". $ means "here"! It literally means the address of "here" exactly. This is mostly used right after declaring a variable or constant. So if we say:

```
something    db     "What?"
length       equ    $ - something
```

it means:

Declare the string "what?" and save the address of its beginning to the constant "something". Now something holds the address of the start of that string we declared. So here's our hello.asm file now:

hello.asm

```
---------------------------------------------------------------------------
SECTION .TEXT
     global _start
_start:


SECTION .DATA:
message       db       "Hello World!",0xa     ;0xa is "new-line" character.

length        equ      $ - message
---------------------------------------------------------------------------
```

Now that we have our constants set up, we still can't finish the "Hello World" program and move on with our lives. We still miss something very important in our knowledge base. That important something is "System Calls".

SYSCALLS

A Sys Call or a System Call is a term used to define the process of handing execution to the kernel. To be honest, in order to explain a sys call correctly, you also need to know something called "Ring Level". But I don't wanna throw you in the rabbit hole. So I simplify it for now and later in the 2nd volume, we're gonna live inside the rabbit hole, like it or not. We have 4 rings in computer architecture starting from ring 0 to ring 3. Ring 0 has the most privilege and ring 3 has the least. Here I break it down for you:

Ring 0 : Kernel or the OS, (Kernel Space)
Ring 1 & 2: Nothing Special, it's not used by most of the OSs
Ring 3: Programs, Users (User Space)

A process in ring 3 can't do things in ring 0. It simply doesn't have the privilege. On the other hand, a process in ring 0 can literally do
whatever it wants. So imagine a ring 3 process wants to execute something but that something requires ring 0 privilege. What should be done here? The process should call some process in ring 0 and hand the execution to ring 0. If the process (or the code) is legitimate and ok, ring 0 will accept the request, execute the requested code and returns the execution flow to the caller with the appropriate result. This is called a Sys Call.(in a nutshell).

Sys Calls on Linux are done by "int 0x80"(32-bit) or "syscall"(64-bit only) instruction. Don't mistake it with integer, it's an interrupt. There is a reason why it's called an interrupt. You'll see the magic in volume 2 when we read about Interrupt Handling and IRQL and interrupts in general. On Windows, Sys Calls (or interrupts) are done differently. You'll learn more about it in volume 2 when we talk about windows internals. For now, we focus on Linux. On Linux, after calling the kernel by "int 0x80" or "syscall" instruction, the operator will pick up the phone (this is serious :D) and tells you:

press 0	*if you need read*	*sys_read*	*fs/read_write.c*
press 1	*if you need write*	*sys_write*	*fs/read_write.c*
press 2	*if you need open*	*sys_open*	*fs/open.c*
press 3	*if you need close*	*sys_close*	*fs/open.c*
press 4	*if you need stat*	*sys_newstat*	*fs/stat.c*
press 5	*if you need fstat*	*sys_newfstat*	*fs/stat.c*
press 6	*if you need lstat*	*sys_newlstat*	*fs/stat.c*
press 7	*if you need poll*	*sys_poll*	*fs/select.c*
press 8	*if you need lseek*	*sys_lseek*	*fs/read_write.c*
…			

And it goes on and on and on until it finishes all 313 options that they offer. (Please don't make a conspiracy theory out of it.). Now we all know nobody is down with listening to all 313 options and then press the button, so we must pick our desired option in advance. For that, we have a structure and based on that, we prepare everything that we need in advance and then make the call. The list above is for 64-bit version. Sys call index numbers and order are different in 64-bit mode and the option are also expanded to 313. Before they were only 190. It is also very important to know that "int 0x80" and "syscall" handle the calls differently and each of them have their own uses and conventions. and guess what, they are not the only instructions for this purpose. 'int 0x80' is mostly referred to as "the legacy" way of calling the kernel. So what we're gonna do is to write the hello.asm program in both versions so you would have a clear understanding how they work and recognize their differences. Keep that in mind that the version of the Linux kernel that your using may be also important but it doesn't affect us now.

First, we go for 32-bit, using "int 0x80" instruction. Wait a second, do we know how to pass the arguments to the function which "int 0x80" is calling? As we learned before from Caller-Callee conventions, in order to call a function, we need to push its required parameters onto the stack or move them to proper registers right before the call instruction. The required parameters for sys_write should be copied to EAX, EBX, ECX, EDX.

EAX takes the index number (press 4 if you need sys_write... yeah! that one!)
EBX takes the file descriptor. For standard output, the number is 1.

ECX hold a pointer to the start of the string you want to print.

EDX tells how much is the length of the string. (Where is the end? Until where you should be printing...)

you may wonder how do I know these numbers? I googled. I'm not a robot. Believe me :D So our hello.asm program will be like this:

hello.asm

```
---------------------------------------------------------------------------
SECTION .TEXT
    global _start
_start:
    mov eax,4
    mov ebx,1
    mov ecx,message         ;message is a pointer to the start of our
                            ;message. Look at .DATA section
```

```
    mov edx,length          ;length is telling how many characters should
                            ;be printed starting from "the starting
                            ;pointer"
    int 0x80

SECTION .DATA:
message     db      "Hello World!",0xa
length      equ     $ - message
---------------------------------------------------------------------------
```

Haha! We did it. Yeees! No. I have to stop you there. What's gonna happen next? shouldn't we close the damn program? :D. the index number forexiting a program is 1 for 32-bit and it should be in EAX. So here will be our final hello.asm:

```
hello.asm
---------------------------------------------------------------------------
SECTION .TEXT
    global _start
_start:
    mov eax,4
    mov ebx,1
    mov ecx,message         ;message is a pointer to the start of our
            ;message. Look at .DATA section
    mov edx,length          ;length is telling how many characters should
                    ;be printed starting from "the starting
            ;pointer"
    int 0x80

    mov eax,1
    int 0x80

SECTION .DATA:
message     db      "Hello World!",0xa
length      equ     $ - message
---------------------------------------------------------------------------
```

Now go ahead and create an executable from hell.asm using NASM and execute it. Here's the commands for it:

```
bash#> nasm -f elf hello.asm
bash#> ./hello
Hello World!
bash#>
```

A very important thing to know here is that I ran this code on Ubuntu. On FreeBSD and other UNIX-like systems index numbers or the way we pass the arguments may be different. For example on FreeBSD, you have to push the parameters onto the stack instead of moving them into registers. As I promised before, we gonna write the hello.asm for 64-bit Linux too. This time we use "syscall" instruction (Fast System Call as the manual says). As mentioned before, index numbers in 64-bit mode is different than 32-bit mode. This time sys_write is at index 1. It takes 3 parameters like before: stdout, message, length. These 3 parameters must be in RDI, RSI,RDX respectively. RAX will hold the index number for syscall which is 1 (again, sys_write is at index 1 on 64-bit mode).

hello64.asm

```
---------------------------------------------------------------------------
SECTION .TEXT
    global _start
_start:
    mov   rax, 1        ; sys_write
    mov   rdi, 1        ; stdout
    mov   rsi, message   ; pointer to the start of our string
    mov   rdx, length    ; length of our string
    syscall


SECTION .DATA:
    message     db      "Hello World!",0xa    ;0xa is "new-line" character.
    length      equ     $ - message
---------------------------------------------------------------------------
```

Again, we need to close and exit. Remember in C code we write, return(0)? That zero as you know means successfully executed and now exit. We do that here when we use syscall. Index number for exit in 64-bit mode is 60 (in EAX) and the return value (0) must be in RDI. Here is our final program:

hello64.asm

```
---------------------------------------------------------------------------
SECTION .TEXT
    global _start
_start:
    mov   rax, 1        ; sys_write
    mov   rdi, 1        ; stdout
    mov   rsi, message   ; pointer to the start of our string
    mov   rdx, length    ; length of our string
```

```
    syscall

    mov    rax, 60        ; sys_exit
    mov    rdi, 0         ; return value
    syscall

SECTION .DATA:
    message       db       "Hello World!",0xa     ;0xa is "new-line" character.
    length        equ      $ - message
-------------------------------------------------------------------------------


bash#> nasm -f elf64 -o hello64.o hello64.asm
bash#> ld -o hello64 hello64.o
bash#> ./hello64
Hello World!
bash#>
```