

# لغة اسمبلي

---

## ماهي لغة اسمبلي؟

لغة اسمبلي هي لغة Low-level يمكن للبشر قراءتها وفهمها وأيضا يمكن للمعالج ان يترجم هذه الاسطر البرمجية المكتوبة بلغة اسمبلي الى Machine - code بحيث يستطيع فهمها وتنفيذها، أيضا لغة اسمبلي تسمح بالوصول المباشر الى الهاردوير وإمكانية التعديل المباشر على الذاكرة العشوائية والـ registers.

تتنوع لغة اسمبلي الى 4 أنواع رئيسية، وهي:

❖ RISC (Reduced Instruction-Set Computer)

❖ DSP (Digital Signal Processor)

❖ CISC: Complex Instruction Set Computer

❖ VLIW: Very Long Instruction Word

الشرح الموجود والامثلة هي لنوع CISC ومعمارية x86\_64

## ماهي أنواع البيانات في لغة اسمبلي؟

قبل ان نتعرف على أنواع البيانات في لغة اسمبلي، دعونا نتعرف على الفرق بين signed و unsigned:

### :Signed number

ال sign bit يستخدم لجعل الرقم موجبا او سالبا, فإذا كان ال sign bit مساويا ل0 فهذا يعني ان الرقم موجب واذا كان مساويا ل1 فيعني ان الرقم سالبا وتتم عملية حساب اكبر رقم بحسب عدد  $n$  بت بالمعادلة التالية:

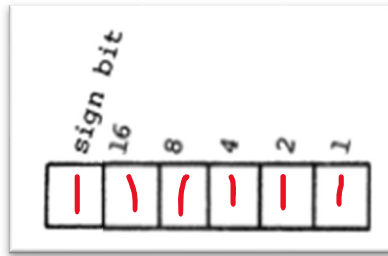
$$\text{from } (2^{n-1}) \text{ to } 2^{n-1}-1$$

- ✓ نستخدم  $2^{n-1}$  إذا كان sign bit مساويا لـ 1 أي ان العدد سالب.
- ✓ نستخدم  $2^{n-1}-1$  إذا كان sign bit مساويا لـ 0 أي ان العدد موجب.

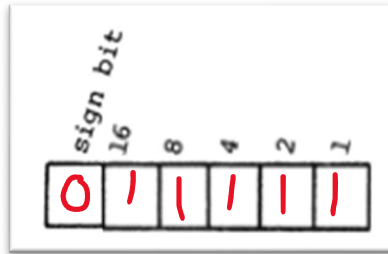
مثال :

افترض ان لدينا 6 bits جميعها 1, سيتم تطبيق المعادلة بهذا الشكل وحجز اخر بت على اليمين لل sign bit

$$2^{(n-1)} = 2^{6-1} = -32$$



افترض ان لدينا 6 bits جميعها 1 ما عدا sign bit يساوي 0, سيتم تطبيق المعادلة بهذا الشكل:



$$2^{(n-1)} - 1 = 2^{6-1} - 1 = +31$$

## :Unsigned number

تكون جميع الاعداد موجبة ولا يوجد لدينا sign bit ويتراوح مداها من 0 الى  $2^n - 1$  حيث ان n تساوي عدد الbits .

## أنواع البيانات في لغة اسمبلي:

**Byte** : يتكون من 8 بت ودائما القيمة موجبة

\* اصغر قيمة هي 0

\* اكبر قيمة هي 255

\* الحرف الواحد عبارة عن 1 بايت = 8 بت , مثلا A = 1 Byte

أيضا يوجد لدينا SByte وهو عبارة عن 8 بت لكن قد تكون سالبة او موجبة

\* اصغر قيمة للSbyte هي -128

\* اكبر قيمة للSbyte هي 127

**WORD** : وهو عبارة عن 16 بت والتي تساوي 2 بايت أيضا وتكون موجبة

\* اكبر قيمة للWORD هي 65535

\* أيضا يوجد لدينا Signed WORD وتختصر SWORD وتكون 16 بت أيضا لكن قد تكون سالبة او موجبة

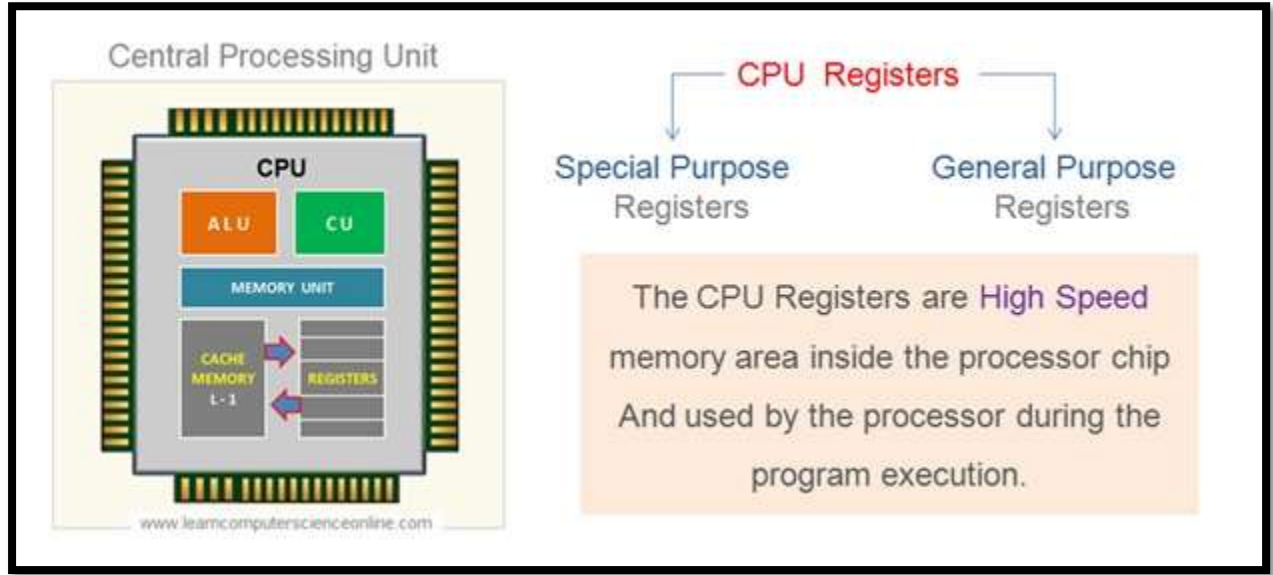
\* اصغر قيمة سالبة للSWORD هي -32768 واكبر قيمة هي +32767

**DWORD** : وهي اختصار لـ Double Word وتكون بمساحة 32 بت , ويتراوح مداها من 0 الى 4,294,967,295

**QWORD** : وهي اختصار لـ quad word وتكون بمساحة 64 بت أي ما يعادل 8 bytes ويتراوح المدى للunsigned من 0 الى 18,446,744,073,703,709,551,615

## ماهي الريجسترات (Registers)؟

هي وحدات تخزينية توجد في الـ CPU وتكون مساحتها صغيرة جدا وتخزن bits , وتعتمد مساحتها التخزينية على معمارية المعالج سواء 32 bit او 64 bit , بحيث في 64 bit تكون مساحة الريجستر الواحد 64 bit. تمتاز الريجسترات بأنها داخل المعالج. فيسهل الوصول لها بسرعة فائقة على عكس الذاكرة العشوائية (RAM).



## أنواع الريجسترات :

1- ريجسترات الاستخدام العام (general purpose registers):

عددها 16 وتستخدم عادة لأغلب الأغراض بحيث تعتمد على حسب المبرمج او المطور, لكن بعض هذه الريجسترات لها استخدامات متعارف عليها .

⚠ الريجسترات التي تكون 64 بت، تبدأ بحرف الـ "r". والريجسترات التي تكون 32 بت، تبدأ ⚠  
بالحرف "e".

rax (Accumulator register): يستخدم للعمليات الحسابية و تخزين قيمة الـ return للفنكشن.

rbx (Base register): ويستخدم كمؤشر (pointer) على بيانات معينة.

rcx (Counter register): يستخدم في الـ loops.

rdx (Data register): يستعمل في العمليات الحسابية و عمليات الادخال والإخراج (I/O).

rbp (base pointer register): يستعمل كمؤشر لبداية الستك (stack).

rsp (stack pointer register): ويستخدم كمؤشر لقمة الستك.

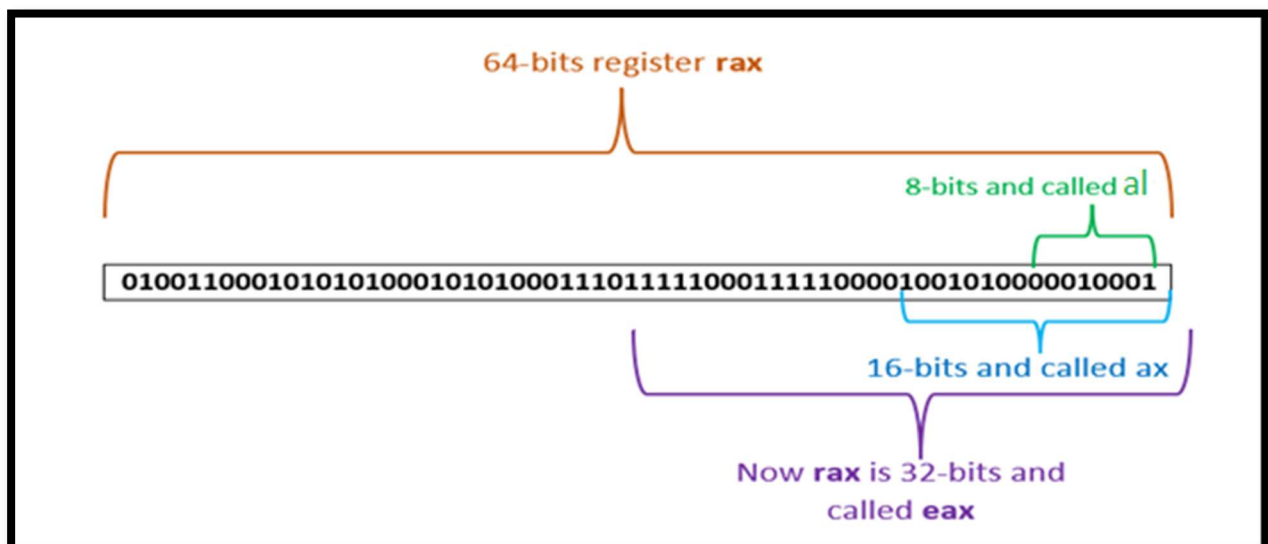
rdi (destination index register): يستعمل كمؤشر على مواقع في الذاكرة في الـ data section في الذاكرة, وأيضا يستعمل لحفظ اول argument عند استدعاء دالة.

rsi (source index register): ينفذ نفس ما يفعله الـ rdi لكن على الـ source وليس الـ destination, وأيضا يستعمل لحفظ ثاني Argument عند استدعاء دالة.

r8, r9, r10, r11, r12, r13, r14, r15 هذه الريجسترات تستعمل لأغراض عدة, مثل حفظ الـ arguments او تستعمل كمؤشر على البيانات او حفظ قيم تستخدم لوقت طويل اثناء الـ function calls.

هذه الاستعمالات هي استعمالات متعارفة عليها ويمكن للمبرمج تغيير وظيفة كل ريجستر حسب الحاجة فهي في الأخير ريجسترات للاستعمال العام.

❖ يمكن تقسيم الريجستر للوصول الى جزء معين فيه، بدون تغيير الأجزاء الباقية (تغيير الـ lower 8bits فقط مثلا) ولكن سوف تختلف تسمية الريجستر بهذا الشكل:



## ❖ تقسيمة باقي الريجسترات

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

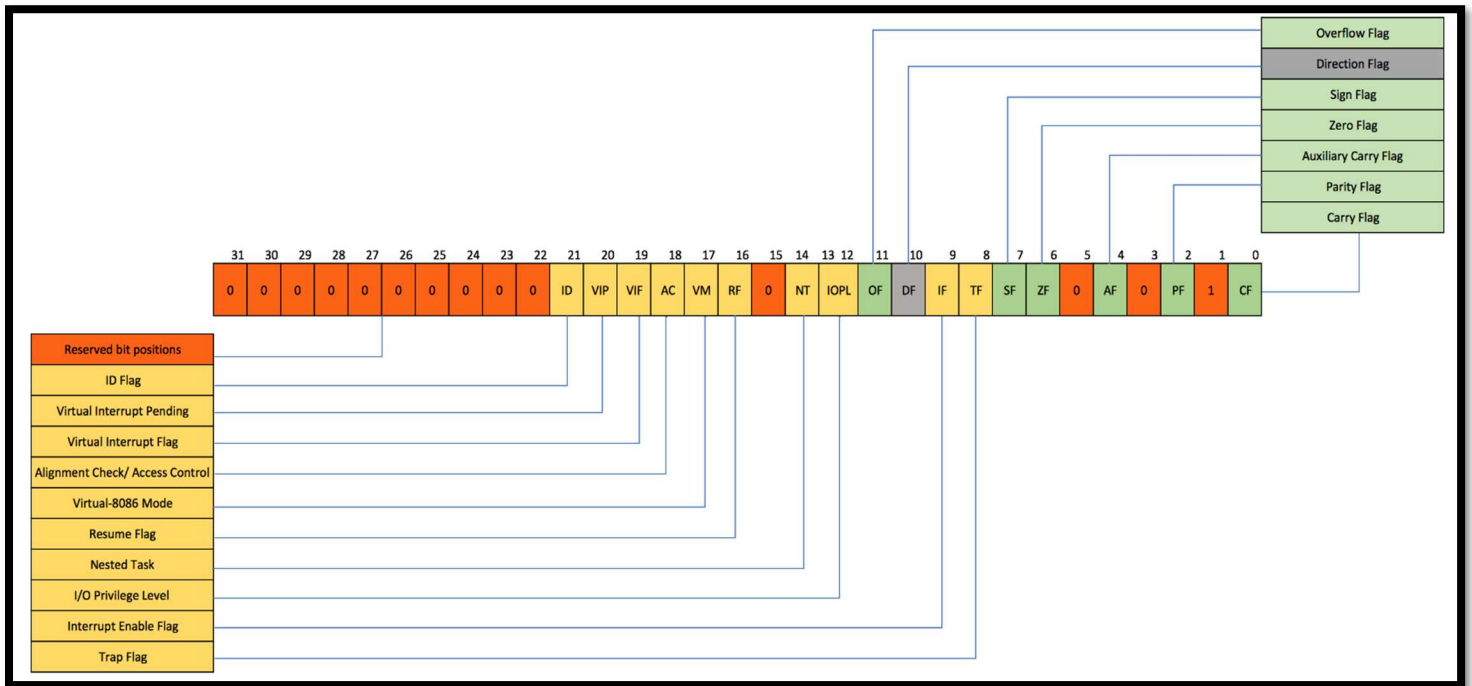
## :segment registers-2

هي ريجسترات تستعمل كمؤشر (pointer) على مناطق في الذاكرة العشوائية مثل data section و code section و stack. وعددها 6:

- SS (stack Segment): يعمل كمؤشر على الستاك , وقد يحتوي على عنوان البداية للدالة, ويستعمل أيضا لتخزين return address للـ procedures.
- CS (Code Segment): يعمل كمؤشر يحتوي على الأوامر التي سوف تنفذ, يحتوي ال CS أيضا على عنوان بداية ال Code Section في الذاكرة.
- DS (Data Segment): يحتوي على عنوان البداية للـ data section التي بدورها تحتوي على المتغيرات والثوابت وبعض المعلومات الأخرى.
- ES و FS و GS: يستعملون كمؤشرات على المزيد من البيانات , وسبب وجودهم هو تمكين البرامج الى الوصول الى اكبر عدد من البيانات في الذاكرة.

## :RFLAGS register-3

هو ريجستر واحد طوله 64 bits، لكن ما يميز هذا الريجستر هو ان كل bit فيه مخصص لغرض معين. ويتكون من 32 بت محجوزة + EFLAGS.



وكل bit يسمى flag وتنقسم الflags الى ثلاثة اقسام:

- (a) Status Flags باللون الأخضر
- (b) Control Flags باللون الرمادي
- (c) System Flags باللون الأصفر

ما يلي بعض الـ flags واستخداماتها:

CF(carry flag): تكون قيمته 1 اذا حصل هنالك استلاف اثناء عملية الطرح.

PF(Parity Flag) : تكون قيمته 1 في حال كان عدد 1 في الـ lower 8 bits عددا زوجيا .

مثل 0101001010010101 اذا  $ZF = 1$

ZF (zero flag): يكون 1 اذا كانت نتيجة العملية تساوي 0 ، وتكون قيمته 0 عدا ذلك.

SF (sign flag): تكون قيمته 0 اذا كان العدد موجبا و 1 اذا كان العدد سالبا كما ذكر سابقا.

OF (Overflow Flag) : تكون قيمته 1 اذا حصل هنالك overflow أي ان القيمة المدخلة اكبر من المساحة التخزينية المخصصة لها.

#### 4- المؤشر على الأوامر instruction pointer register:

هو ريجستر يحتوي على عنوان الامر قيد التنفيذ (instruction pointer) RIP ، مثلا :

RIP=0x34F039AA



0x34F039AA	print(x)
0x38945F8	1 + 2

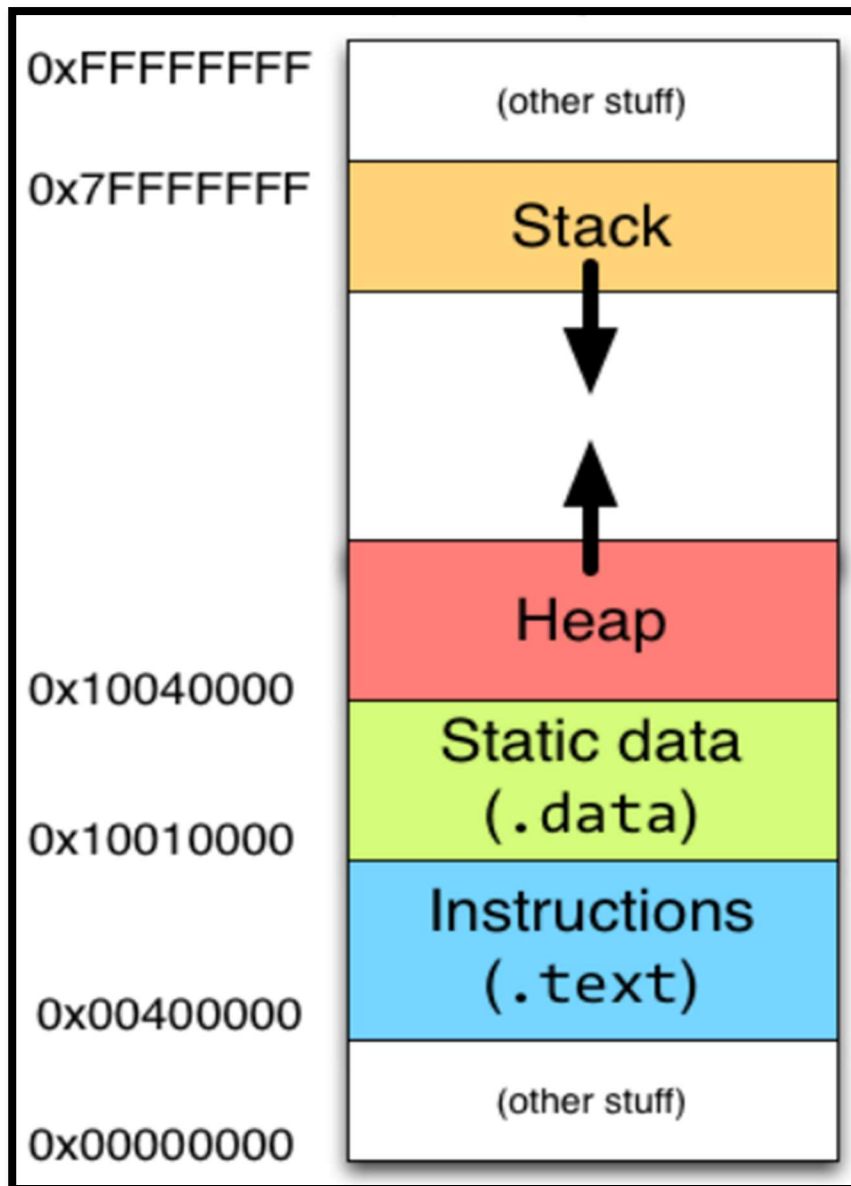
ملاحظة: يسمى rip في معمارية 64 bits (x86\_64) ويدعى eip في معمارية 32 bits (x86)



## ما هو الستاك وما هي أوامره؟

هي في الأساس احد تراكيب البيانات (طريقة لتخزين وعرض البيانات) وتتبع طريقة FILO وهي اختصار لfirst in last out أي ان اول من يدخل الى الستاك هو اخر من تتم ازالته، يمكنك ان تتخيل الستاك على انها مجموعة صحنون متراكمة فوق بعضها البعض ولا يمكنك إزالة الصحن الأول الى ان تتم إزالة جميع الصحنون التي فوقه.

لكن في حالتنا، الستاك هي منطقة في الذاكرة تتبع هذه الطريقة في تخزين البيانات وتخزينها تخزيناً مؤقتاً.

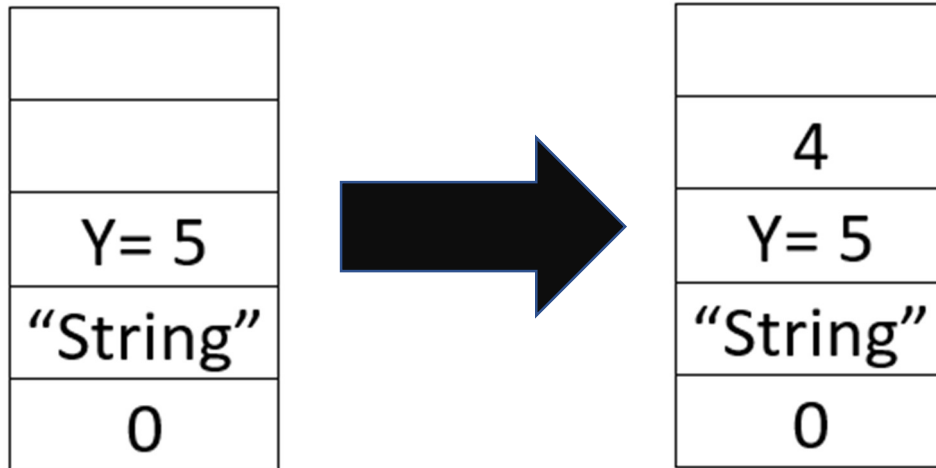


عملية الإضافة الى الستاك تسمى: push

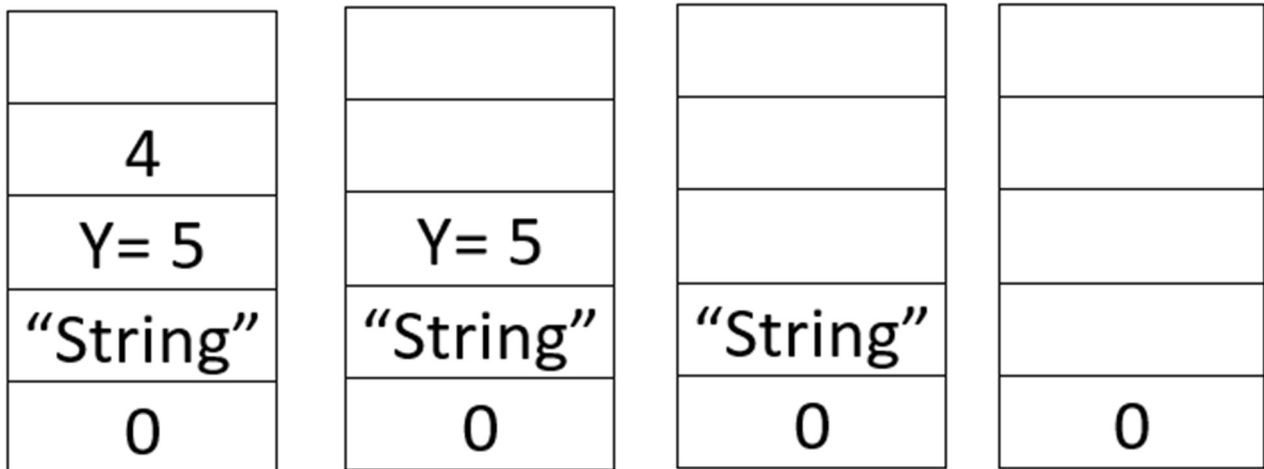
وعملية الازالة من الستاك تسمى: pop

مثال:

push 4



افترض انك تريد الوصول الى 0، كيف ستفعلها؟ عن طريق عمل pop ثلاث مرات



## على ماذا يحتوي الستاك؟

يحتوي الستاك على هذه الخمس:

Arguments-1

2-المتغيرات الموجودة في scope معين (local variables)

3-استدعاء الدوال

4- الدوال المعرفة

5-عنوان الرجوع (يستخدم للتكملة والعودة الى المكان الذي تم الاستدعاء منه)

ما هو stack prologue؟

افترض ان الكود البرمجي يبدأ التنفيذ من الدالة main() وحصل هنالك استدعاء لدالة أخرى على سبيل المثال تدعى sum() كيف ستتم تهيئة الستاك لاستقبال هذه الدالة، هنا يأتي دور الـ stack prologue وهي عبارة عن أوامر بلغة التجميعي :

Push rbp

mov rbp, rsp

sub rsp, 0x60

في البداية قبل الـ stack prologue تتم عملية push للـ rip بحيث انه عند الرجوع الى الـ main يتم معرفة المكان التي توقفت عنده دالة main

main()

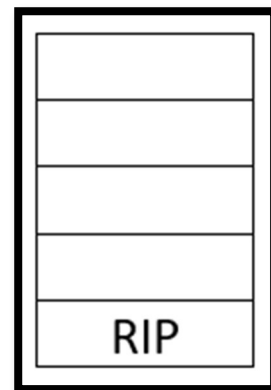
{

Int x = 5;

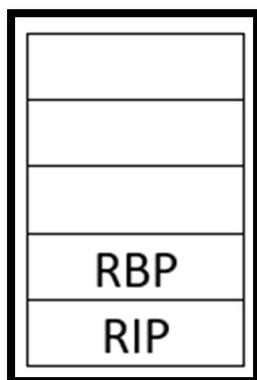
sum();

x = x + 5 ;

}



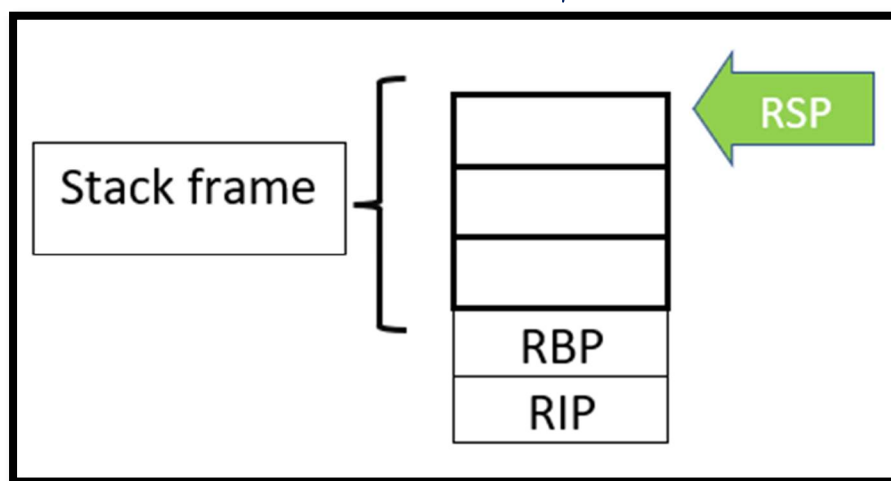
حسب الـ stack prologue فالبداية هي push rbp وذلك لتحديد أساس بداية الدالة sum:



بعد ذلك mov rbp, rsp ، وهذا الامر يقتضي بجعل قيمة الـ rsp مساوية لقيمة الـ rbp بحيث انه الان بداية الدالة ونهايتها في المكان ذاته:



بعد ان تتم العملية السابقة، يتبقى علينا فقط الطرح من قيمة الـ rsp بحيث نزيد من حجم الـ **الستاك (الستاك يكبر من high memory address الى low memory address)** وقد تختلف القيمة عن 0x60 بحسب حجم الـ stack frame المراد إنشاؤه.

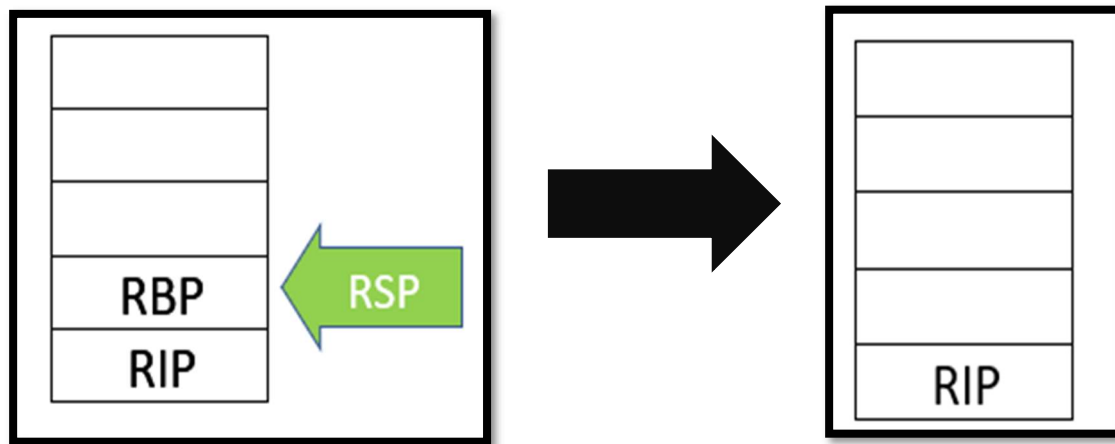


عكس عملية الـ stack prologue هي عملية الـ stack epilogue فعلى النقيض تماماً، هي مجموعة أوامر مخصصة عند الانتهاء من الـ stack frame والعودة إلى دالة main على حسب المثال الذي تم ذكره. ويتكون الـ stack epilogue من هذه الأوامر:

leave

ret

أمر leave يجعل rsp مساوياً لـ rbp ويعمل عملية pop لـ rbp:



أمر ret يقوم بعملية pop لقيمة الـ rip، ويقوم بجعل القيمة الموجودة في الـ stack، تحفظ في الـ rip الحالي بحيث أن يعود تنفيذ البرنامج إلى دالة main

main()

{

Int x = 5;

sum();

x = x + 5 ;

}



## أوامر اسمبلي الأكثر شيوعاً:

أغلب الأوامر في اسمبلي تأتي بهذا السياق:

<القيمة> , <المكان المراد الحفظ فيه> <العملية>

فعل سبيل المثال: `mov eax, 5`

هذا الأمر يقوم بحفظ 5 في الريجستر `eax`

### ❖ عملية ADD

هذه العملية مخصصة للقيام بعملية الجمع فلو اردنا جمع عددين يمكننا القيام بهذه الأوامر:  $(9=4+5)$

```
mov    eax, 5
mov    ebx, 4
add    eax, ebx
```

هذا الأمر سيقوم بجمع القيمتين ووضع الناتج في `eax`

### ❖ عملية SUB

وهي عملية مخصصة للقيام بطرح قيمتين من بعضها البعض، فلو اردنا القيام بعملية  $5-3=2$  لسوف نقوم بالأوامر التالية:

```
mov r8,5
sub r8,3
```

هذا الأمر سيقوم بطرح 3 من 5 ووضع الناتج في `r8`

## ❖ عملية MUL

تستعمل هذه العملية لضرب عددين unsigned أي لا يوجد نتيجة سالبة ولا يمكن للمعامل ان يكون سالباً ، عملية mul تعتبر غريبةً بعض الشيء بسبب انها تأخذ معامل واحد فقط ولكنها تقوم بعملية الضرب على معامل اخر وتحفظ قيمة الضرب فيه :

مثال :  $6 * 5 = 30$

```
mov ax, 6
mov cx, 5
mul cx
```

هذا ما حصل بطريقة مبسطة:

```
ax = 6
cx = 5
ax = ax * cx
ax = 30
```

## ❖ عملية DIV

هي عملية القسمة في لغة اسمبلي وتقوم على نفس مبدأ mul بحيث انه اذا اردنا ان نحصل على ناتج العملية التالية  $\frac{10}{5}$  فعلينا القيام بالتالي:

```
mov eax, 10
mov ecx, 5
div ecx
```

لكن في القسمة يوجد لدينا الناتج ويوجد لدينا باقي القسمة، يتم الاحتفاظ بالناتج في eax وباقي القسمة يحفظ في الريجستر edx

edx = 0 و eax = 2

## التحكم في سير البرنامج

في البداية عملية jump هي القفز من جزء في الكود الى جزء اخر.

يتم التحكم في سير البرنامج عن طريق الامر jmp في اسمبلي، لكن يختلف نوع ال jmp فقد يكون مشروطا بشرط معين او لا. وهذا يأخذنا الى نوعين من أنواع jmp وهي المشروط (conditional) وغير المشروط (un-conditional).

- Conditional jump

ومن هذه العمليات المشروطة:

je (jump if equal): تقفز الى جزء معين في الكود اذا تساوت القيمتين أي ان ال zero flag مساوي لـ 1

jne (jump if not equal): هذه العملية ستقوم بالقفز الى المكان المراد اذا لم تتساوى القيمتين، أي ان ال zero flag مساوي لـ 0

jg (jump if greater): تتحقق عملية القفز الى الجزء المراد من الكود اذا كانت القيمة اكبر من القيمة المطلوبة .  
وأیضا توجد عملية (jge (jump if greater or equal : وستقوم بعملية القفز الى الجزء المراد من الكود اذا كانت القيمة اكبر او مساوية للقيمة المطلوبة.

(jl (jump if less : تتحقق عملية القفز الى الجزء المراد من الكود اذا كانت القيمة اصغر من القيمة المطلوبة .  
وأیضا توجد عملية (jle (jump if less or equal : وستقوم بعملية القفز الى الجزء المراد من الكود اذا كانت القيمة اصغر او مساوية للقيمة المطلوبة.

- Unconditional jump

وهي بكل اختصار عملية القفز والذهاب الى الجزء المراد في الكود من غير شرط

(jmp (jump : تغير سير البرنامج الى اسم الدالة او الى عنوان الذاكرة المحدد.



## عمليات bitwise

هي عمليات يتم تنفيذها على مستوى bits وتختص في تعديل قيمة المتغير عن طريق تحريك bits:

- SHR (shift to the right)

هي عملية تحريك bits الى اليمين بعدد خانات معين، فإذا أردنا تحريك القيمة الموجودة داخل ebx بمقدار 5 خانات الى اليمين فسنقوم بالأمر التالي:

shr ebx, 5

العدد	عدد خانات التحريك لليمين	العدد بعد التحريك
00101000	3	00000101
01000000	1	00100000
00000001	9	00000000

- SHL (shift to the left)

وهي عملية تحريك bits الى اليسار بعدد خانات معين، فإذا أردنا تحريك القيمة الموجودة داخل eax بثلاثة خانات الى اليسار فسنقوم بالأمر التالي:

shl eax, 3

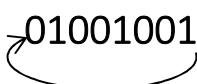
العدد	عدد خانات التحريك لليسار	العدد بعد التحريك
00101000	3	01000000
01000000	1	10000000
00000001	7	10000000

- ROR (rotate to the right)

هي عملية تحريك bits بعدد خانات معين الى اليمين، لكن إذا وصل bit الى النهاية فإنه يعود مرة أخرى الى البداية

هذا ما سيحصل في حال تمت عملية تحريك الخانات الى اليمين بمقدار 1 خانة

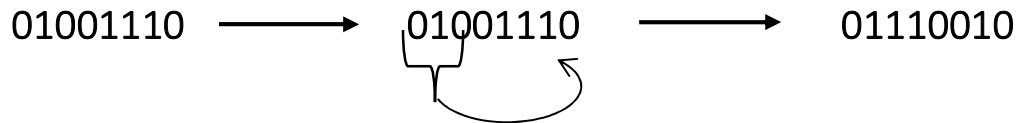
01001001 → 01001001 → 10100100



- ROL (rotate to the left)

وهي عملية تحريك bits الى اليسار، لكن إذا وصل bit الى اقصى اليسار ستم عملية رجوعه مرة أخرى الى اليمين

مثال: تم تحريك العدد 01001110 الى اليسار 3 خانات



## العمليات المنطقية

هي عمليات تتم فيها مقارنة كل bit مع bit الذي يقابله، وتتم المقارنة حسب العملية.

- عملية AND

هي عملية يكون ناتجها true اذا كان كل الطرفين true . عدا ذلك فالنتيجة false.

المعادلة	النتيجة
true AND true	true
true AND false	false
false AND false	false
false AND true	false

ويمكن تنفيذ عملية AND بهذا الشكل أيضا:

```

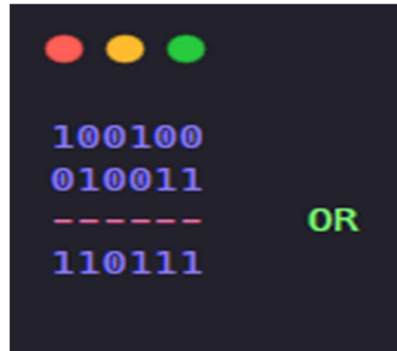
14 = 1110
9  = 1001  and(&)
----
1000 which equal 8 , so 14 & 9 = 8

```

## • عملية OR

يكون الناتج true اذا كان احد الشرطين true

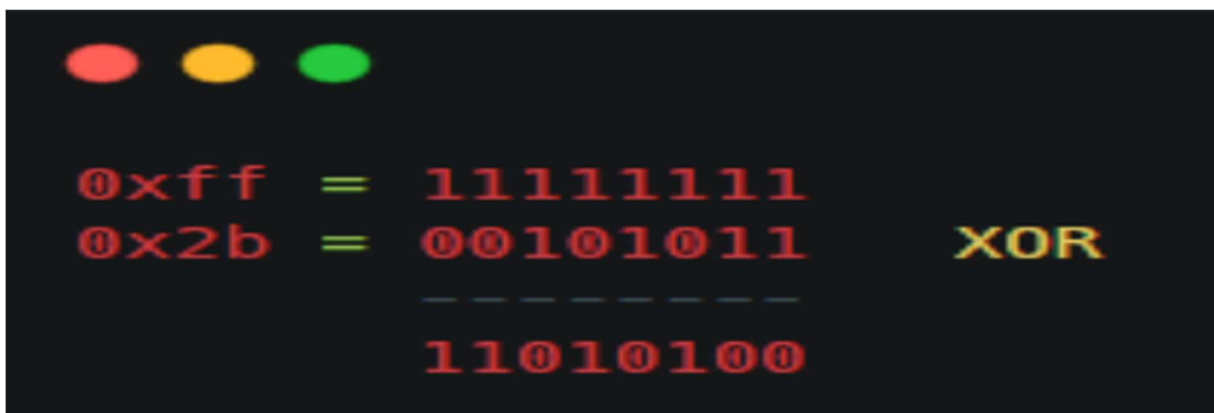
المعادلة	النتيجة
true OR true	true
true OR false	true
false OR false	true
false OR false	false



## • عملية XOR

يكون الناتج true اذا كان الشرطين مختلفين و false اذا كان الشرطين متشابهين:

المعادلة	النتيجة
true XOR true	false
true XOR false	true
false XOR false	true
false XOR false	false



• عملية TEST:

هي عملية مقارنة بين عددين او بين معاملين.

TEST eax, eax

    Jmp sum()

## المصادر:

<https://www.cs.uaf.edu/courses/cs301/2014-fall/notes/bits/>

<https://portal.abuad.edu.ng/lecturer/documents/1595337617Advanced.pdf>

<https://www.forth.com/starting-forth/7-signed-double-length-numbers/>

<https://www.tutorialspoint.com/unsigned-and-signed-binary-numbers>

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture>

<https://www.learncomputerscienceonline.com/>

[https://wiki.cdott.senecacollege.ca/wiki/X86\\_64\\_Register\\_and\\_Instruction\\_Quick\\_Start](https://wiki.cdott.senecacollege.ca/wiki/X86_64_Register_and_Instruction_Quick_Start)

[https://cs.brown.edu/courses/cs033/docs/guides/x64\\_cheatsheet.pdf](https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf)

<http://6.s081.scripts.mit.edu/sp18/x86-64-architecture-guide.html>

[https://en.wikibooks.org/wiki/X86\\_Assembly/X86\\_Architecture](https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture)

<https://www.byteshuffle.com/author/mcsmurf/>

<https://www.cs.virginia.edu/~evans/cs216/guides/x86.html>