الفريق العربي للهندسة العكسية

دورة ATRE لتعليم الأسمبلي للمبتدئين من الصفر

الدرس الأول – أنظمة العد Number Systems

سنتكلم اليوم عن أنظمة العد الأربعة الأساسية ...

Binary :

هذا هو النظام الثنائي وهو اللغة الأم للحاسوب...فالحاسوب من الداخل مكون من دوائر كهربائية...هذه الدوائر لا تعرف الـ assembly أو الـ visual basic أو أي شيء من هذه اللغات..هي تتعامل مع التيار الكهربائي...وجود تيار كهربائي أو عدم وجود كهربائي ، وقد اتفق على ان يرمز لحالة وجود التيار بالرمز 1 وعدم وجوده بالرمز 0 .
النظام الثنائي إذن يتكون من رقمين فقط هما 0 و 1 ...وبالتالي فإن أساس النظام (Base). يساوي 2 .
مثلا فإن تعليمة مثل هذه :

MOV EAX,DWORD PTR DS:[40A710h]

هي بالأصل يتم التعامل معها بداخل الدوائر الكهربائية للحاسوب على أنها

10100001000100001010011101000000000000000

أما تعليمة

Push EAX

فهي

1010000

والآن دعنا نرى...كيف يتم العد بالنظام الثنائي؟ دعني أذكرك...بالنظام العشري...كنا نعد هكذا: 1 2 3 ...9 10 ، هل لاحظت ؟ النظام العشري مكون من 10 رموز هي 0123456789 ، إذن بعد الوصول إلى 9 فان الرموز (الأرقام) تنتهي...ما العمل؟ نبدأ من جديد فتصبح الـ 9 صفرا ، ونضيف 1 إلى الخانة التالية (خانة العشرات) ، إذن 9 يليها 10...بالمثل عندما نصل إلى 399 فان خانة الآحاد قد وصلت إلى آخر رقم وهو 9 إذن نعيد العد ونستبدل الـ 9 بصفر ونضيف واحد للخانة التالية...الخانة التالية بها9 ان أضفت إليها 1 ستصبح 10 إذن سنكتب اصفر والواحد ينتقل للخانة التالية وهي الـ 3 ، فتصبح 4 ، أي سنصل إلى 400 .

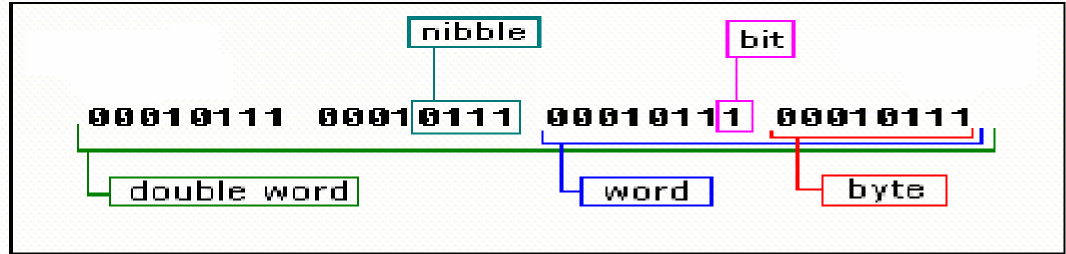
تابع معي كيف نعد بالنظام الثنائي :
0 ثم 1 ثم 10 . ما الذي حصل؟ كنا نعد بشكل طبيعي...0...يليه 1...يليه ماذا ؟ لا يليه شيء فالواحد هو آخر رقم في هذا النظام. إذن استبدله بصفر وأضف واحد للخانة الجديدة فتحصل على 10 . فلنتابع : 10 ثم 11 ثم 100 ، ماذا حصل ؟ الـ 10 يليها 11..يليها 100 .كما ترى فخانة الآحاد قد وصلة إلى آخر عدد إذن استبدله بصفر وأضف واحد للخانة التالية(أي خانة العشرات) ، لكن خانة العشرات بدورها قد وصلت لآخر عدد..إذن عندما نضيف 1 للـ 1 سنحصل على 10 أي نبقي الصفر وننقل الـ1 للخانة التالية فيصبح لدينا 100.

انظر :

binary	decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9

أرجو أن تكون الفكرة قد أصبحت واضحة.

كل رقم (digit) في النظام الثنائي يسمى **BIT** ، كل 4 bits تسمى **NIBBLE** ، كل 8 bits تكون **BYTE** ، كل two bytes تكون **WORD** ، وختاما كل two words تكون **DOUBLE WORD** .
انظر الشكل التالي :



إن أول بت (في اليمين) يسمى low bit أو Least Significant Bit (LSB) أي البت ذو القيمة الأقل ، وفي المقابل فإن آخر بت (في اليسار) يسمى high bit أو Most Significant Bit (MSB) أي البت ذو القيمة الأعلى.

ملاحظة :عندما نقول 10100101b فإننا نعني قيمة بالبايناري ولهذا وضعنا حرف b في نهاية الرقم...لكن هذا ليس شرطا...

ماذا إذا أردنا جمع عددين بالنظام الثنائي ؟ لا بأس دعنا نرى...
0=0+0
1=1+0
10=1+1

والآن لنرى كيف نجمع 1100110 مع 1101

```
    1   1
1 1 0 0 1 1 0
0 0 0 1 1 0 1   +
-------------------
1 1 1 0 0 1 1
```

كما ترى ، فقد أضفنا أصفارا للرقم السفلي...هذا للتوضيح فقط...كي تسهل علينا عملية الحساب...لأنك تعلم ان الصفر على اليسار لا قيمة له...
أول صفر مع أول 1 نتيجتهم 1
ثاني 1 مع الـ 0 نتيجتهم 1
ثالث 1 مع الـ 1 نتيجتهم 10 - إذن نكتب الـ 0 ونضيف 1 للخانة التالية
رابع 0 مع الـ 1 مع الـ 1 السابق نتيجتهم 10 - إذن نكتب الـ 0 ونضيف 1 للخانة التالية
خامس 0 مع الـ 0 مع الـ 1 السابق نتيجتهم 1
سادس 1 مع الـ 0 نتيجتهم 1
سابع 1 مع الـ 0 نتيجتهم 1 .

لن أشرح العمليات الأخرى أي الطرح والضرب والقسمة فهذه ستأخذ وقتا طويلا إضافة إلى انه لا فائدة كبيرة منها...

النظام السداسي العشري Hexadecimal

يتكون هذا النظام من 16 رقم (رمز) هم : (من اليمين لليسار)
F E D C B A 9 8 7 6 5 4 3 2 1 0
أساس النظام (Base) يساوي 16 .دعنا نرى كيف يمكن أن نعد باستخدام هذا النظام...
0 ثم 1 ثم 2 ...ثم 9 ثم A ثم B ...ثم F ثم 10
ما الذي حصل؟ كما في السابق..بدأنا العد وحين انتهت الأرقام ووصلنا إلى أخر رقم ممكن وهو F اضطررنا إلى استبداله بصفر وإضافة 1 للخانة التالية.
انظر إلى هذا : (ابدأ التتبع من اليسار)

… 108 109 10A 10B 10C 10D 10E 10F 110 111 112 113….

… D389 D38A D38B D38C D38D D38E D38F D390…

كما في النظام الثنائي فلن أتطرق لعمليات الحساب في هذا النظام رغم أنها تتبع نفس القاعدة في النظام السابق....

النظام الثماني OCTAL

هذا النظام يتكون من ثمانية رموز هي 0 1 2 3 4 5 6 7
والعد فيه يكون كالتالي (من اليسار لليمين)
...
0 1 2 3 4 5 6 7 10 11 12 13 14 15 16 17 20 21 22 23 24 25 26 27 30 31 32 33 ...

النظام العشري Decimal

هذا هو النظام المألوف لجميع الناس...في النظام العشري هناك 10 أرقام (digits) :
0, 1, 2, 3, 4, 5, 6, 7, 8, 9

التحويل من أي نظام إلى Decimal

للتحويل من أي نظام إلى النظام العشري فإننا نبدأ بالـ MSB ونضربه بأساس النظام مرفوعا إلى القوة التي تساوي ترتيب الرقم ، ثم ننتقل إلى الرقم التالي فنضربه بالأساس مرفوعا إلى القوة التي تساوي ترتيبه ...ونجمع هذه القيم.

مثلا لتحويل 10100101b من النظام الثنائي إلى العشري ،

10100101b =

$$= 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

$$= 128 + 0 + 32 + 0 + 0 + 4 + 0 + 1 = 165$$

(decimal value)

base

digit position

كما ترى...للتحويل من النظام الثنائي إلى العشري فإننا نبدأ بالـ MSB ونضربه ب 2(الأساس) مرفوعة للقوة التي تساوي ترتيبه ، في حالتنا 7 ، ثم نجمع عليها البت التالي مضروبا ب 2 مرفوعة للقوة التي تساوي ترتيبه...وهكذا...تمعن في الصورة فهي واضحة...

دعنا نطبق الفكرة على تحويل القيمة 1234h من سداسي عشري إلى عشري :

$$1 \cdot 16^3 + 2 \cdot 16^2 + 3 \cdot 16^1 + 4 \cdot 16^0 = 4096 + 512 + 48 + 4 = 4660$$

(decimal value)

base

digit position

نبدأ بالـ MSB وهو 1 ، نضربه بالأساس (أي 16) مرفوعا إلى ترتيب الرقم (أي 3) ، ثم نكمل فنجمعه مع 2 مضروبة بالأساس (أي 16) مرفوعا إلى الترتيب (أي 2) وهكذا...
نفس الفكرة تنطبق على التحويل من النظام الثماني إلى النظام العشري.

التحويل من Decimal إلى أي نظام آخر :

الفكرة هي : نقوم بقسمة القيمة العشرية على أساس النظام الذي نريد التحويل عليه ، ولا تنسى أن هناك باق . تابع عملية القسمة إلى ان يصبح ناتج القسمة صفرا...
سنطبق الفكرة على التحويل من Decimal إلى Binary :
دعنا نقوم بتحويل القيمة 359 إلى ثنائي :

Divider	reminder	notes
359/2=179	1	بدئنا بقسمة الـ 359 على 2 والناتج 179.5 أي 179 والباقي 1
179/2=89	1	والآن 179 تقسيم 2 يساوي 89.5 أي 89 والباقي 1
89/2=44	1	89 تقسيم 2 يساوي 44.5 أي 44 والباقي 1
44/2=22	0	44 تقسيم 2 يساوي 22 والباقي 0
22/2=11	0	22 تقسيم 2 يساوي 11 والباقي 0
11/2=5	1	11 تقسيم 2 يساوي 5.5 أي 5 والباقي 1
5/2=2	1	5 تقسيم 2 يساوي 2.5 أي 2 والباقي 1
2/2=1	0	2 تقسيم 2 يساوي 1 والباقي 0
1/2=0	1	1 تقسيم 2 يساوي **0** والباقي 1... نتيجة القسمة **0** إذن فقد انتهينا

والآن الجواب هو ما تراه في عمود reminder **من أسفل لأعلى ، من اليسار لليمين** . أي أن

359=101100111

دعنا نطبق الفكرة على التحويل من النظام السداسي العشري إلى النظام العشري.
نريد تحويل 67024 إلى النظام العشري.

Divider	reminder	notes
67024/16=4189	0	67024 على 16 يساوي 4189 والباقي 0
4189/16=261.8125	0.8125 x 16 = 13 ➔ D	4189 تقسيم 16 يساوي 261 والباقي 0.8125 ، نقوم بضرب الباقي بـ 16 فنحصل على 13 ، أي D
261/16=16.3125	0.3125 x 16 = 5	261 تقسيم 16 يساوي 16 والباقي 0.3125 ، نقوم بضرب الباقي بـ 16 فنحصل على 5
16/16=1	0	16 تقسيم 16 يساوي 1 والباقي 0
1/16=0	1	1 تقسيم 16 يساوي **0** والباقي 1 ، الناتج **0** إذن انتهينا.

والآن الجواب هو ما تراه في عمود reminder **من أسفل لأعلى ، من اليسار لليمين** . أي أن
67024 = 105D0

ملاحظة : عند البرمجة بالاسمبلي فيجب أن يسبق أي قيمة بالسداسي عشري ، الرقم 0 إذا كانت مبتدئة بحرف...فمثلا القيمة 1B2 تبقى كما هي لأنها ليست مبتدئة بحرف ، لكن C441 يجب أن تكتبها 0C441 .

التحويل من و إلى النظام Binary والـ Hex .

الأمر سهل للغاية..أنظر الجدول بالأسفل...لكل رقم بالهكس هناك ما يقابله بالنظام الثنائي (مع الوقت يمكنك حفظه لكن هذا ليس مطلوبا منك) . مثلا افترض أننا نريد التحويل من 1001101101b إلى هكس...عليك أن تبدأ من LSB ، قم بتقسيم الرقم إلى مجموعات كل مجموعة مكونة من 4 خانات. هكذا :
(لاحظ أن المجموعة الثالثة في اليسار مكونة من رقمين فقط لكن نحن نريدها مكونة من 4 أرقام ، لذا نضيف إليها صفرين إلى اليسار)

0 0 1 0   0 1 1 0   1 1 0 1

والآن من الجدول يمكنك أن تلاحظ أن 1101 تقابل D ، و 0110 تقابل 6 ، أما 0010 فتقابل 2 أذن الناتج 26D .
والعكس يتم بنفس الطريقة ، لتحويل 0B45A نستبدل كل رقم بما يقابله ، فـ A يقابلها 1010 و 5 يقابلها 0101 و 4 يقابلها 0100 أما B فيقابلها 1011 ، إذن النتيجة 1011010001011010

Binary	Decimal	Octal	Hex
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	10	8
1001	9	12	9
1010	10	13	A
1011	11	14	B
1100	12	15	C
1101	13	16	D
1110	14	17	E
1111	15	21	F

: Signed Numbers

تمعن في هذه القيمة : 0B521 ...هل هي موجبة أم سالبة ؟ بالطبع لا مجال للحديث عن الموجب أو السالب فهذه قيمة لا تحمل إشارة...

المكمل الأول: First complement

إذا كان لدينا 8-bits (بايت) فيمكننا عمل 256 عدد مختلف (**256** combinations) بما فيها الصفر. إذن يمكننا أن نفترض أن أول 128 عدد (0-127) هي موجبة والأعداد التالية (128-256) سالبة.
للحصول على الـ 1st complement لأي عدد موجب فإننا ببساطة نحوله إلى ما يقابله بالـ binary .
إذن للحصول على 1st c للقيمة 54 فإنا نحولها إلى binary فنحصل على 110110

أما لذا كانت القيمة سالبة ، أي -54 فانا نطبق المعادلة التالية :

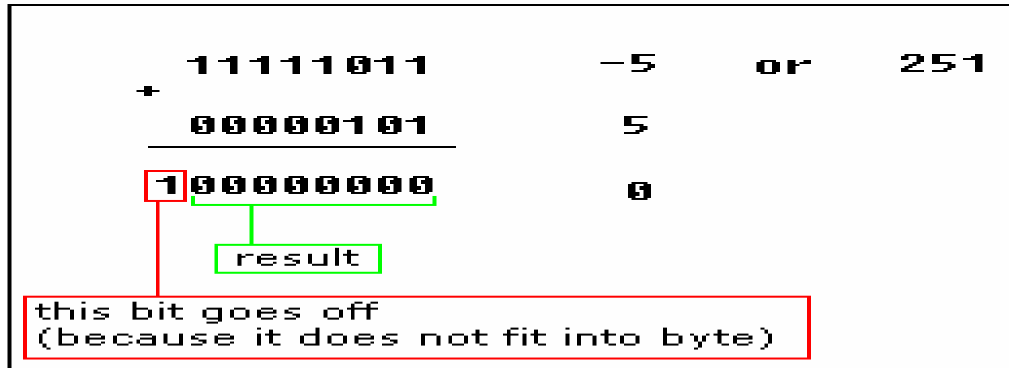
1st c = 2 power n – 1 – number in binary

n التي تراها تعني عدد الخانات ، فان كانت القيمة بايت فـ n=8 وأن كانت word فـ n=16 وهكذا...
2 power n أي 2 مرفوعة للأس n .
Number in binary أي الرقم بعد تحويلها إلى binary . لنطبق المعادلة على القيمة -54 لتحويلها إلى 1st c بصيغة بايت

1st c = 2 power 8 – 1 – 110110
= 1111 1111 – 110110 = 11001001

هناك طريقة أخرى أسهل من هذه...لو لاحظت فان القيمة 54 بالثنائي تساوي 00110110 (أضفنا صفرين لليسار كي يصبح العدد مكونا من 8 خانات) والآن اعكس كل بت فتحصل على 11001001 . هل رأيت ما أسهلها :)

لكن دعنا نتأمل قليلا...لماذا تم اختيار هذه الطريقة (أقصد طريقة الـ 1st complement) وليس أي طريقة أخرى ؟
إذا جمعت -5 مع 5 فستحصل على 0 . أليس كذالك؟ حسنا هذا يحث عندما يجمع الـ processor القيمة 5 مع القيمة 251 ، النتيجة هي 255 ، وبسبب الـ overflow فالنتيجة تعتبر صفرا.



كما ترى الناتج هو 100000000 لكن نحن جمعنا عددين من فئة byte أي 8bit لذا يجب أن يكون الجواب مكون أيضا من 8bit لكن حيث أننا حصلنا على 9bit فان هذه الحالة تسمى في علم الحاسوب overflow ، ويتم تجاهل الـ MSB ويبقى 00000000 أي صفر...
نفس المبدأ ينطبق عند التعامل ما فئة word (16 bit values) حيث أن 16 bits تنشئ **65536 عددا مختلفا ، أول** 32768 عدد (من 0 إلى 32767) تستخدم لتمثيل الأعداد الموجبة ، والباقي (32768 إلى 65536) تستخدم لتمثيل الأعداد السالبة.

المكمل الثاني: Second complement

المكمل الثاني للأعداد الموجبة هو نفسه المكمل الأول والذي بدوره عبارة عن العدد بعد تحويله للـ binary .
فالـ 54 عند تحويلها للمكمل الثاني نحصل على 110110
أما المكمل الثاني للقيم السالبة فيتم الحصول عليه بإضافة 1 إلى المكمل الأول.
في المثال السابق عرفنا أن 54- بالمكمل الأول تساوي 11001001 إذن بالمكمل الثاني :
2nd c = 1st c +1 = 11001001 + 1 = 11001010

لن أتعمق أ:ثر من هذا...يمكنني كتابة عشرات الصفحات كي أشرح هذا النظام لكن هذا ليس محور حديثنا.

Assignment :

I-convert these values from decimal to hexadecimal , then to Binary
a) 15 b)150

II-convert these values from Binary to Decimal , then to Hex.
a) 1001 b)011101101

III-convert these values from Hex to Decimal , then to Binary.
a) 0B51 b)124C

IV- find the 1st and 2nd complement of the following decimal values :
a) 154
b) - 413

solutions should be mailed to revenge-34[at]hotmail[dot]com .
deadline: one week from the lesson's date .

References:

1- http://en.wikipedia.org/wiki/Main_Page
2- http://www.al-ebda3.info/ib/
3- http://www.emu8086.com/
4- http://www.arabteam2000.com/
5- Assembly Language: Step-by-Step , Jeff Duntemann
6- introduction to computer , Jawdat Abu Taha

جهاز الكمبيوتر هو *آلة كهربائية تأخذ البيانات والتعليمات (inputs) ، ثم تقوم بمعالجتها(processing) ، وتخرج لنا المخرجات (outputs) .*
جهاز الكمبيوتر مر بخمسة أجيال كل جيل كان له خصائص معينة...(لمزيد من التفاصيل www.google.com) .
كما و يقسم الكمبيوتر إلى عدة أقسام (Ranges) منها :

- embedded system computer (e.g. : washing machine)
- microcomputers
- minicomputers
- mainframe computers
- super computers

جهاز الحاسوب الذي أمامك ينتمي إلى الجيل الخامس ، والى قسم الثاني.
يتكون جهاز الحاسوب من 5 أقسام أساسية :

I -central processing unit (CPU)
II -input devices
III-memory storage devices
IV-output devices
V -a communication network , called "bus" , that links all the elements of the system and connects the system to the external world.

وحدة المعالجة المركزية : Central Processing Unit (CPU)

الـ CPU قد تكون شريحة (chip) مفردة أو عدة شرائح متصلة مع بعضها البعض حيث تقوم بتنفيذ العمليات الحسابية والمنطقية (arithmetic and logical calculations) وتقوم بتنظيم تزامن العمليات الأخرى في الحاسوب والتحكم بها. مع تطور التكنولوجيا ظهر ما يسمى بالـ microprocessor الذي يتضمن المزيد من الدوائر الكهربائية والذاكرة داخليا. والنتيجة جهاز حاسوب بحجم أصغر بكثير...
معظم الـ CPU تتكون من المكونات الأساسية التالية :

- **arithmetic logic unit (ALU)**
- **registers**
- **control section**
- **internal bus**

أولا ـ وحدة الحساب والمنطق Arithmetic Logical Unit (ALU)

هذه الوحدة مسئولة عن العمليات الحسابية (جمع طرح ...) والمنطقية (or , xor , and , etc...) وعمليات المقارنة بين البيانات (أكبر أصغر...) .

ثانيا – المسجلات Registers

إن الـ CPU يحتوي على وحدات ذاكرة صغيرة لكن سريعة جدا تستخدم لعمليات التخزين المؤقتة(temporary) للبيانات. هذه الذاكرة تحتوي على عدد من المسجلات registers ، كل واحدة تقوم بعملية محددة...
وسيلي شرح تفصيلي عن المسجلات لاحقا.

ثالثا – وحدة التحكم (Control Unit (CU

إن العمليات التي يتم تنفيذها من قبل الـ CPU هي بالتسلسل التالي :
1- فك تشفير التعليمة
decoding the instruction within the computer.

2- تنظيم تزامن عمليات القراءة والكتابة بداخل الـ CPU وخارجيا في ممرات البيانات
Sequencing the reading and writing of data within the CPU and externally on the Data Bus.

3- التحكم في التزامن الذي يتم من خلاله تنفيذ العمليات
Controlling the sequence in which instructions are executed.

4- التحكم بالعمليات التي يتم تنفيذها من قبل الـ ALU
Controlling the operations performed by the ALU.

رابعا – الممر الداخلي Internal Bus

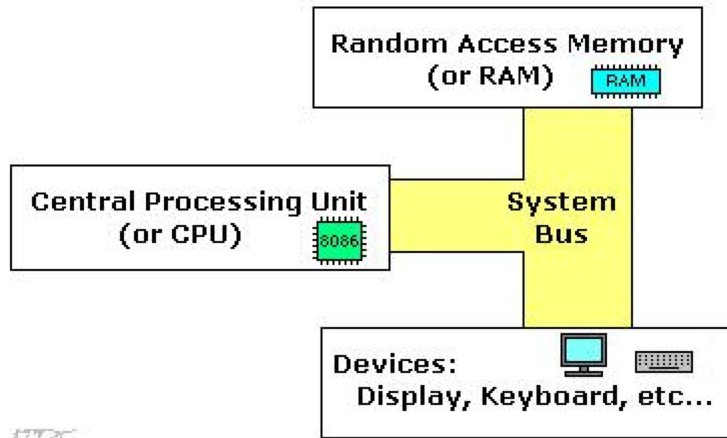
هو عبارة عن شبكة من أسلاك الاتصالات التي تربط العناصر الداخلية للمعالج ببعضها البعض ، وأيضا تؤدي إلى اتصالات خارجية تربط المعالج بالعناصر الأخرى للنظام – أي للكمبيوتر- .
هناك ثلاث أنواع من الـ CPU Buses :
1-Control Bus :يتكون من الخط (line) الذي يستشعر (senses) الإشارات الداخلية بالإضافة إلى خطوط أخرى للتحكم بالإشارات بداخل الـ CPU .
2- The address Bus ، خط باتجاه واحد من المعالج الذي يحوي عناوين الذاكرة للبيانات .
3- the Data Bus ، خط باتجاهين يقوم بقراءة البيانات من الذاكرة وكتابتها إليها .

نظرة تاريخية على المعالجات...

في السبعينيات طرحت شركة انتل معالج 4004 وهو أول µP من نوع single-chip ...وعندما نقول أنه 4-bit فنحن نعني أن الـ Bus Width يساوي 4 bit . كان قادرا على عنونة ذاكرة حتى حجم 640 بايت فقط. تلا هذا الطراز عدة طرازات...ثم جاء الـ 8086 في سنة 1978 وبناقل بيانات Bus باتساع 16-Bit ...وناقل عناوين addresses bus باتساع 20 بت.كان قادرا على عنونة ذاكرة حتى حجم 1 ميجابايت...تلا ذلك عدة أنواع وموديلات أصبحت مزودة بناقل بيانات باتساع 32 بت ، وهو الـ Pentium ، وأيضا صدرت بعض المعالجات بناقل بيانات باتساع 64 بت كـ titanium ، للإطلاع على قائمة بجميع الموديلات وتفاصيلها زر هذه الصفحة :
http://en.wikipedia.org/wiki/List_of_Intel_microprocessors
إن معالجي 8086 و 8088 متشابهين كثيرا من حيث الخصائص...وهذا السبب الذي يدفعنا لتسميتهم باسم x-86 family...

لغة الأسمبلي...ما هي ؟

هي لغة برمجة منخفضة المستوى (low level) ، كي تبرمج الأسمبلي يلزمك معرفة بالـ computer structure ، يمكننا توضيح الـ computer model كالتالي :



والآن دعنا نلقي نظرة مفصلة على المسجلات التي بداخل الـ CPU ...
(ملاحظة : لن أتطرق إلى الـ internal architecture للمعالج...سأكتفي بما يهمنا بطريقة مباشرة فقط)

31 16	15 8	7 0
EAX	AH	AL
EBX	BH	BL
ECX	CH	CL
EDX	DH	DL
ESI	SI	
EDI	DI	
ESP	SP	
EBP	BP	

CS
SS
DS
ES
FS
GS

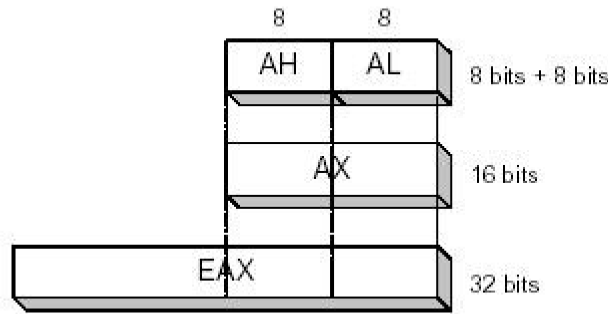
IP

كما ترى هناك في هذا المخطط عدة أقسام للمسجلات...
: **general purpose registers**

إن معالجات 8086 تمتلك 8 مسجلات عامة الغرض ، ولكل منها اسم خاص :

- **EAX** - the accumulator register.
- **EBX** - the base address register.
- **ECX** - the count register
- **EDX** - the data register
- **ESI** - source index register.
- **EDI** - destination index register.
- **EBP** - base pointer.
- **ESP** - stack pointer.

الواقع فإن تلك المسجلات الأربع : EAX ,EBX,ECX,EDX لها التركيب التالي :



عند التعامل مع هذه المسجلات يمكنك الوصول إلى الـ 32 بت عن طريق EAX أو إلى أول 16 بت عن طريق AX أو حتى إلى أول وثاني 8 بت عن طريق AL و AH على الترتيب. لاحظ أن القسم العلوي من EAX لا يمكن الوصول إليه على انفراد. فقط القسم السفلي يمكنك التعامل معه بشكل منفرد . طبعا ما ينطبق على EAX ينطبق على EBX,ECX,EDX .

على الرغم من التسمية لكل مسجل ، فإن المبرمج هو الذي يحدد استخدامات كل مسجل منها...
جدير بالذكر أن التعامل مع هذه الذواكر يتم بكل سرعة لأنها موجودة بداخل المعالج بعكس الذاكرة RAM (أو غيرها) حيث التعامل معها يتطلب استخدام الـ Buses مما يسبب تأخيرا زمنيا...

: **segment registers**

- CS - points at the segment containing the current program.
- DS - generally points at segment where variables are defined.
- ES - extra segment register, it's up to a coder to define its usage.
- SS - points at the segment containing the stack.

أيضا هناك مسجلين إضافيين في المعالجات الحديثة (نسبيا) هما : FS و GS .
على الرغم من أنه يمكنك تخزين أي قيمة في الـ segment registers ، فإن هذه لم ولن تكون فكرة صائبة...لأن هذا النوع من المسجلات ليس عام الغرض ، بل له مهمة محددة وهي pointing at accessible blocks of memory (لا تسألني عن الترجمة :).

إن الـ segment registers تعملا جنبا إلى جنب مع الـ general purpose registers للوصول إلى أي عنوان ذاكرة.
على سبيل المثال إذا أردنا الوصول إلى عنوان الذاكرة الفيزيائي 12345h يجب أن نضع DS = 1230h و SI = 0045h . هذه طريقة جيدة لأنه هكذا يمكننا الوصول إلى عدد أكبر من العناوين بدلا من التقيد بالـ 16 بت التي تملكها الـ segment registers.
إن الـ CPU يقوم بعمليات حسابات للعنوان الفيزيائي بضرب الـ segment register بـ 10h وجمعه مع الـ general purpose register . هكذا :

```
 12300
+ 0045
------
 12345
```

إن العنوان المكون بمسجلين إثنين يسمى effective address . بشكل افتراضي ، فإن BX, SI and DI تعمل مع DS أما BP and SP فتعمل مع SS .
المسجلات العامة الغرض الأخرى لا يمكنها أن تكون effective address.

: special purpose registers

- EIP the instruction pointer.
- flags register - determines the current state of the microprocessor.

إن EIP دائما يعمل مع CS ، وهو يشير دائما وأبدا إلى التعليمة التي يجري تنفيذها حاليا.

الرايات : Flag registers
إن الرايات يتم تغييرها تلقائيا من قبل الـ CPU بعد تنفيذ عمليات رياضية ومنطقية . إنها تسمح بمعرفة نتيجة العملية وتحديد الشروط لنقل التحكم إلى أجزاء أخرى من البرنامج .
بشكل عام لا يمكنك تغيير قيم هذه المسجلات بطريقة مباشرة...هناك طرق غير مباشرة قد نأتي على ذكرها في دروس قادمة...

31	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	AG	VM	RF	0	NT	IOPL	O	D	I	T	S	Z		A		P		C

تقسم الرايات إلى ثلاثة أقسام: (مع الشكر للأخ afeef على دروسه حيث نسخت التالي كما هو :))
- *رايات غير مستعملة* أي أنها لا تفيد في الحكم على اخر عملية وهي موجودة فقط في حالة تطوير المعالج ربما يحتاجون إلى رايات إضافية فيمكن استغلال هذه الرايات. في الوقت الحالي نحن لسنا بحاجة إليها
- *رايات الوضع*: وهي الرايات التي تتأثر وتتغير حسب وضع العمليات التي تقوم بها وحدة الحساب والمنطق في المعالج
- *رايات السيطرة*: وهي رايات المبرمج يتحكم بوضعها فإذا وضع بداخلها القيمة 1 تبقى هذه القيمة حتى يغيرها المبرمج في البرنامج عن طريق أوامر برمجة خاصة بها

مسجل الرايات في المعالج 8086 يحتوي على 7 رايات غير مستخدمة 6 رايات وضع و 3 رايات سيطرة

CF	Carry flag
PF	Parity flag
AF	Auxiliary flag
ZF	Zero flag
SF	Sign flag
TF	Trap flag
IF	Interrupt flag
DF	Direction Flag
OF	Overflow flag

إلى هنا يكون درس اليوم قد انتهى...لا يوجد assignment لهذا الدرس...أراكم في الدرس القادم

References:

1- http://www.emu8086.com/
2- http://www.arabteam2000.com/
3- Fundamentals of Computer Organization and Architecture , Mostafa Abd-El-Barr and Hesham El-Rewini.
4- Assembly Language for Intel-Based Computers , Kip. R. Irvine

الفريق العربي للهندسة العكسية

دورة **ATRE** لتعليم الأسمبلي للمبتدئين من الصفر
الدرس الثالث -- نظام التشغيل DOS – العنونة في البرامج

allko

بداية أود أن أشير إلى أن أجزاء كبيرة من هذا الدرس مأخوذة – حرفيا – من درس للأخ xacker فله الشكر ...طبعا مع القليل من التصرف.

نظام التشغيل DOS

يمثل DOS نظاما لتشغيل الحاسوب يؤمن عملية استخدام موارد وأجهزة الحاسوب دون الخوض في تفاصيلها التقنية والفيزيائية . تشمل هذه الأجهزة لوحة المفاتيح ، الشاشة وسواقات الأقراص . وعلى هذا فان DOS مسئول عن المعالجة الأدنى مستوى لموارد وأجهزة الحاسوب . ومن بين هذه الوظائف الكثيرة للنظام سنهتم بمجموعة الوظائف التالية :

- إدارة الملفات : يعتبر نظام DOS مسئولا عن إدارة وتنظيم الفهارس directories والملفات الموجودة على القرص. فجميع برامجنا تستطيع إنشاء وتعديل الملفات عن طريق خدمات نظام DOS (هذا بالنسبة لتطبيقات 16 بت طبعا) تاركين في ذلك عبء عملية إدارة مواقع تخزينها إلى DOS.
- Input/output : يريح نظام DOS المبرمج من عنا الخل / الخرج المباشر ، فهو يسمح بطلب إدخال/إخراج المعطيات عن طريق المقاطعات Interrupts .
- تحميل البرنامج : يؤمن DOS تنفيذ أي برنامج يريده المستخدم. فهو يقرا البرنامج من القرص ليحمله في مكان في الذاكرة ثم يسلم التنفيذ له.
- إدارة الذاكرة : عندما يحمل DOS احد البرامج لتنفيذه ، يخصص قبل ذلك حيزا كافيا من الذاكرة لشفرة التعليمات والمعطيات ، ويسمح للبرنامج بطلب المزيد من الذاكرة أو تحريرها .
- المقاطعات Interrupts : يؤمن DOS عملية الاتصال بينه وبين المبرمج عن طريق المقاطعات ، كما يترك لنا حرية تغيير خدمات هذه المقاطعات ، فعلى سبيل المقال يمكننا ترك البرنامج مقيما في الذاكرة (Terminate-Stay Resident) وجعله جزءا من نظام المقاطعات لإنجاز وظائف معينة.

عملية الإقلاع The Boot Process

ينشأ عن إعادة وصل الكهرباء إلى الحاسوب حدوث عملية تسمى الإقلاع البارد cool boot والتي ينتج عنها دخول المعالج في حالة إعادة التهيئة التي تفرض عليه تصفير كافة مواقع الذاكرة وإجراء تدقيق لخانات التحقق في الذاكرة ، ومن ثم تحميل المسجل CS
بالعنوان FFFF[0]h والمسجل IP بالقيمة صفر ، وعلى هذا فان التعليمة الأولى التي ستجلب من الذاكرة تقع عند العنوان FFFF[0]h والتي تشكل البداية لروتينات(BIOS (Basic Input Output System في الذاكرة ROM . تبدأ بعدها روتيناتBIOS بفحص المنافذ المختلفة
بغية التعرف على كافة الأجهزة الموصولة بالحاسوب وتهيئتها. تبني روتينات BIOS بعد ذلك منطقتين في الذاكرة وهما :

1- جدول خدمة المقاطعات Interrupt Services Table : تحتوي هذه المنطقة على عناوين برامج خدمات المقاطعات ، وذلك ابتداء من العنوان 00000h في الذاكرة .
2- منطقة معطيات BIOS : وهي منطقة تبدأ عند العنوان 00400h من الذاكرة وتضم معلومات عديدة عن الأجهزة المحيطة بالحاسوب.

تبحث روتينات BIOS بعد ذلك عن قرص النظام في سواقات الأقراص لينتقل التنفيذ في حال العثور عليه إلى برنامج التحفيز الذاتي bootstrap الموجود في ذلك القرص . يحمل البرنامج الأخير الملفين io.sys و msdos.sys من القرص إلى الذاكرة وينتقل التنفيذ نقطة إلى نقطة بداية التنفيذ في io.sys . يعيد io.sys

توضعه في الذاكرة وينقل التنفيذ بدوره إلى msdod.sys . يهيئ الأخير جداول معطيات DOS الداخلية والقسم الخاص للنظام من جدول عناوين خدمات المقاطعات ثم يقرأ الملف config.sys من القرص وينفذ أوامره. أخيرا ينقل msdos.sys التنفيذ إلى command.com الذي بدوره سينفذ أوامر الملف autoexec.bat ويظهر محث الأوامر ويراقب المعطيات المدخلة من لوحة المفاتيح.
هذا طبعا بالنسبة لنظام DOS القديم التقليدي.

Loading System Program

يدعم النظام DOS نوعين اثنين من البرامج التنفيذية exe و com . يتألف برنامج com من مقطع وحيد عليه أن يضم شفرة التعليمات والمعطيات والمكدس. وهو نوع مناسب للبرامج الصغيرة الخدمية والبرامج المقيمة في الذاكرة.
أما برامج exe فهي تتألف من مقاطع منفصلة هي مقطع شفرة التعليمات والمعطيات والمكدس (على الأقل...قد يكون هناك مقاطع أخرى) وهي تناسب البرامج الأكثر جدية.
يؤمن DOS عملية التعامل مع كلا النوعين السابقين من خلال برنامج التحميل command.com الذي يقوم بتحميل البرنامج التنفيذي إلى الذاكرة قبل تسليمه التحكم. وبالطبع سيختلف تعامله مع كلا النوعين بسبب طبيعة اختلافهما....سنستعرض الآن الخطوط العريضة التي ينجزها برنامج التحميل عند طلب تنفيذ برنامج من النوع exe وسنؤجل الحديث عن ملفات com إلى وقت لاحق.

تشمل خطوات تنفيذ ملفات exe ما يلي :

1- الوصول إلى برنامج exe الموجود على القرص.
2- بناء منطقة تسمى بادئة مقطع البرنامج Program Segment Prefix (PSP) والتي تتألف من 256 بايت على حدود فقرة ضمن الذاكرة المتوفرة للبرنامج
3- تحميل البرنامج بعد منطقة PSP مباشرة.
4- تحميل المسجلات DS و ES بعنوان مقطع PSP
5- تحميل المسجل CS بعنوان مقطع شفرة التعليمات للبرنامج ، والمسجل IP بإزاحة التعليمة الأولى والتي تساوي عادة 0 في مقطع شفرة التعليمات.
6- تحميل المسجل SS بعنوان مقطع الـ stack والمسجل SP بحجم الـ stack المطلوب
7- تسليم التنفيذ إلى البرنامج ابتداء من نقطة بداية التنفيذ Original Entry Point (OEP)

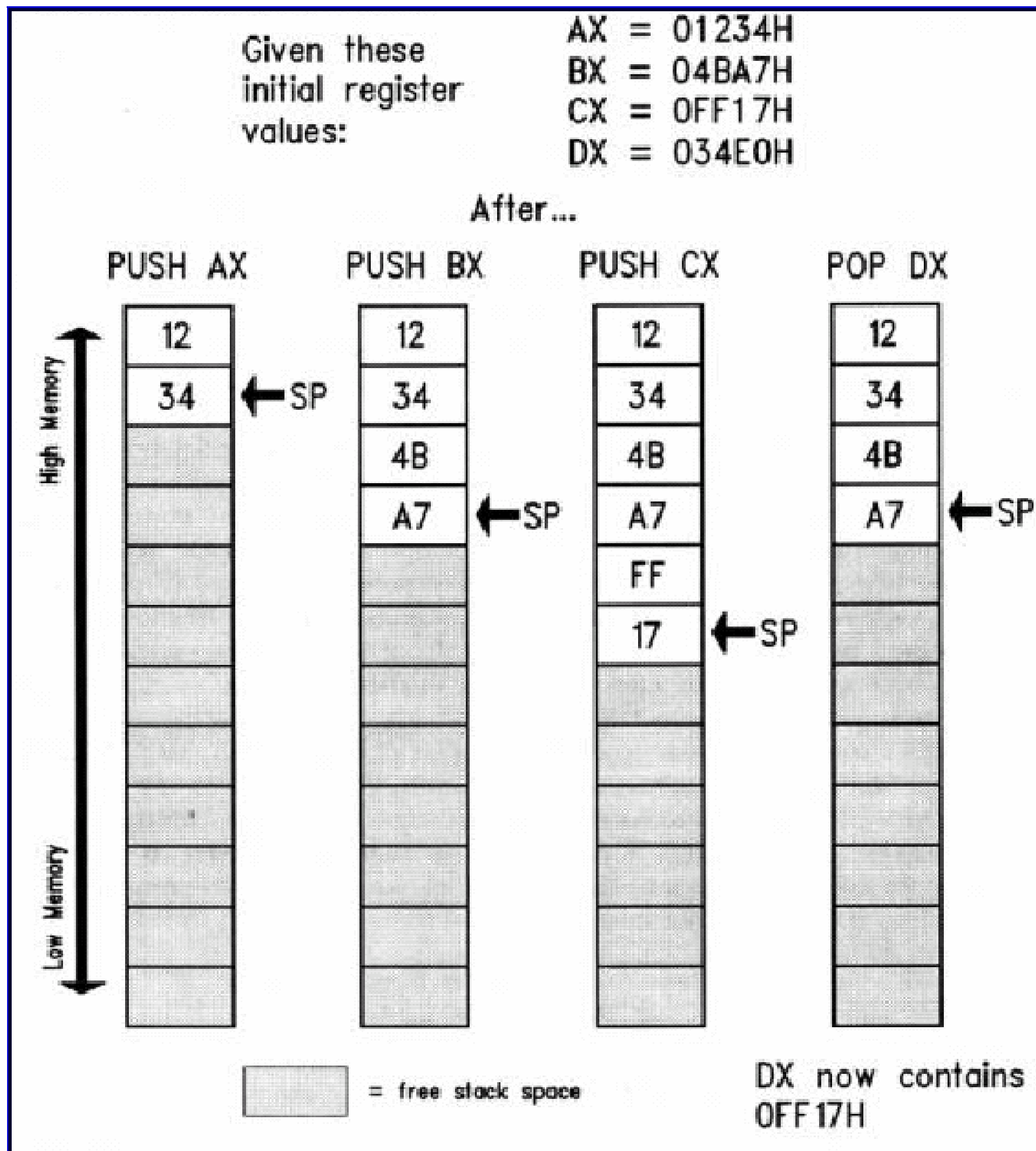
لاحظ أنه قد تم بهذه الطريقة تحميل زوجي المسجلات CS:IP و SS:SP بالقيم المناسبة الصحيحة إلا أن المسجل DS لا يحتوي على عنوان مقطع المعطيات وكذلك الأمر مع ES فكلاهما يحملان عنوان مقطع PSP وعلى هذا يمكن الاستنتاج أن على البرنامج exe أن يغير من قيمة DS و ES بنفسه. للوصول إلى مقطع المعطيات في البرنامج .

المكدس Stack

تحتاج البرامج التنفيذية إلى منطقة محجوزة في الذاكرة تدعى stack. تفيد هذه المنطقة في توفير حيز من الذاكرة لتخزين العناوين والمعطيات بشكل مؤقت ، علما أن طول عنصر المعطيات الأساسي في stack هو كلمة واحدة word.
ينبغي على المبرمج تعريف الـ stack في برامج exe في حينه إلا انه لا حاجة لذلك في برامج com إذا يتولى نظام التشغيل تعريف مكدسة آليا. يهيئ DOS مسجل المقطع SS بعنوان بداية مقطع الـ stack، كما يهيئ المسجل SP بحجم الـ stack ليشير إلى نهايته أو قيمته الحالية.

تتعامل مجموعة من التعليمات مع الـ stack بشكل مباشر كالتعليمات POP,PUSH,CALL...فالتعليمة PUSH مثلا تنقص محتويات المسجل SP بمقدار 2 عند تنفيذها وتخزن قيمة ما في تلك الكلمة المحجوزة التي أصبح SP يشير إليها . أما POP فهي على العكس حيث تنقل الكلمة التي يشير إليها SP إلى موقع ما وتزيد قيمة المسجلين AX و BX إلى قمة الـ stack ثم سحبهما من قمة الـ stack إلى نفس المسجلين.

والآن تمعن في الصورة التالية...هناك قيم مخزنة في المسجلات العامة الأربعة ، ونريد أن نرى ماذا يحدث أثناء تنفيذ بعض التعليمات المتعلقة بالمكدس...



لاحظ أننا إذا حفظنا قيم المسجلات AX,BX,CX,DX على الترتيب بواسطة التعليمات التالية :

```
Push ax
Push bx
Push cx
Push dx
```

فعلينا أن نسترجعها كالتالي:

```
Pop dx
Pop cx
Pop bx
Pop ax
```

يسمى هذا بمبدأ **LIFO** (Last In First Out) أي أن آخر قيمة تنقل للـ stack هي أول قيمة تخرج منه. لاحظ أن القيم التي تم استرجاعها من الـ stack تبقى موجودة فيه لكن لا يؤشر عليها المسجل SP ، مما ينبغي ذكره هنا هو وجود حالتين يجب إلا يمر فيها الـ stack .
الأولى هي محاولة سحبنا قيمة ما من الـ stack وهو في حالة underflow أي فارغ يحتوي على القيمة FFFFh)
أما الحالة الثانية فهي محاولة دفع قيمة للـ stack وهو ممتلئ في حالة overflow (أي SP تحوي القيمة 0) ولذلك ينبغي توخي بعض الحذر في تعاملنا مع الـ stack . يشكل ذلك مطلبا بالغ الأهمية لعمل البرنامج في حالة تعطل مفاجئ وانتقال النظام إلى حالة غير مستقرة تستدعي غالبا إعادة استنهاض أو إقلاع الحاسب من جديد.

العنونة في البرنامج : Addressing

تستطيع المعالجات x86 تنفيذ عمليات الوصول إلى كلمات الذاكرة بفعالية أكبر إذا كانت عناوينها زوجية ففي تعليمة MOV يستطيع المعالج الوصول إلى الكلمة ذات الإزاحة 0012 مثلا بشكل كامل ونسخها مباشرة إلى المسجل AX مثلا لأنها تملك عنوانا زوجيا أما إذا كانت الكلمة ذات عنوان فردي كعنوان الإزاحة 0013 عندئذ سينفذ المعالج خطوتين للوصول إلى هذه الكلمة التي سنفرض أن محتوياتها كما في الشكل التالي :

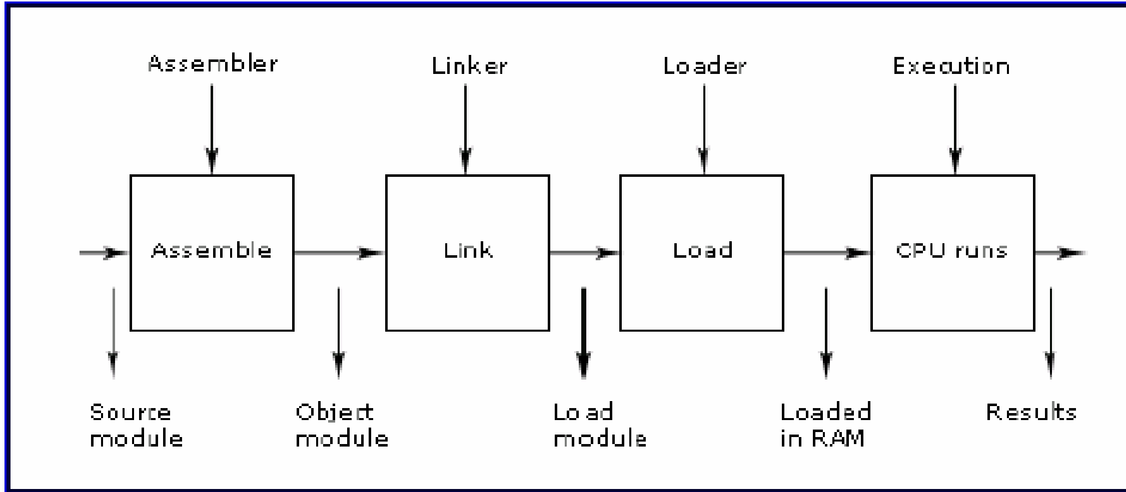
محتويات الذاكرة	xx	01	23	xx
الازاحة	0015	0014	0013	0012

في الخطوة الأولى ينسخ المعالج محتويات الحجرة ذات الإزاحة 0013 إلى المسجل AL بعد وصوله إلى كلا الحجرتين عند الازاحتين 0012 و 0014 ، أما في الخطوة الثانية فهو ينسخ الحجرة ذات الإزاحة 0014 إلى المسجل AH بعد وصوله إلى كلتا الحجرتين ذات الازاحتين 0014 و 0015 وبذلك تصبح محتويات المسجل AX هي 0123h .
لا تحتاج برمجة المواقع الزوجية برمجة تختلف عن الفردية / ولا إلى اهتمام بمعرفة فيما إذا كانت هذه المواقع فردية أم زوجية ففي كلا الحالتين سنحصل على نتائج صحيحة لان عملية الوصول إلى مواقع كلمات الذاكرة تضمن قراءتها وكتابتها ونقلها بشكل صحيح.

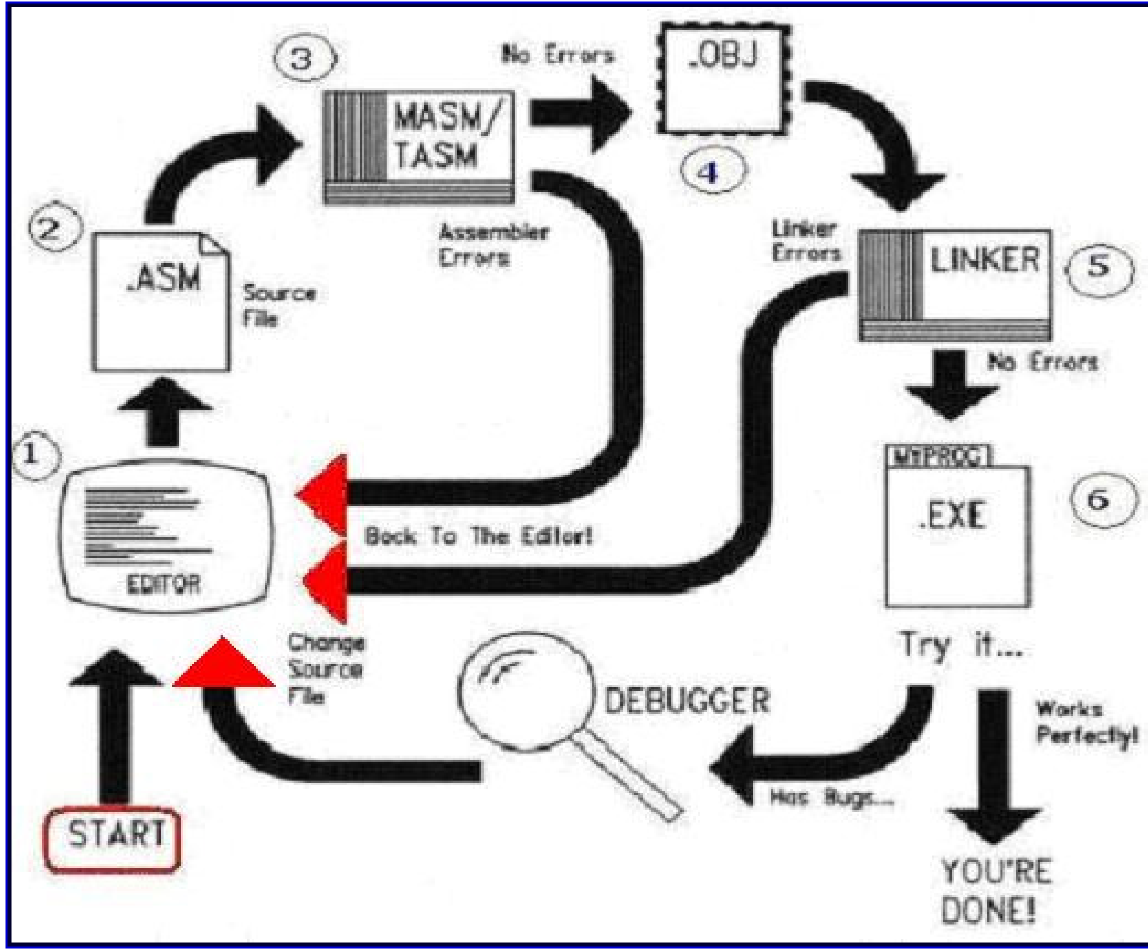
نستطيع استخدام التوجيه(directive) المسمى even في برامجنا لرصف أو توضيع كافة المعطيات في الذاكرة في المواقع ذات العناوين الزوجية لضمان فعالية أكبر للبرامج.

: Assembling and linking

ما تقوم بكتابته من تعليمات يسمى source code ، وعند تجميعه بواسطة assembler كـ masm أو tasm فانك تحصل على ملف وسيط يسمى object code ، ثم نقوم باستخدام احد الـ Linkers لربط ملفات الـ obj. مع بعضها البعض والحصول على ملف exe. واحد.
الصورة التالية توضح المقصود :



والآن أنظر الى الصورة التالية فهي توضح مراحل تطوير برامج الأسمبلي :



1- باستخدام أي محرر (winasm , radasm , notepad , ...) نقوم بكتابة برنامج الأسمبلي
2- بعد حفظ المشروع سنحصل على ملف (أو أكثر) بامتداد asm.
3- باستخدام أي مجمع assembler (MASM , TASM , FASM , ...) نقوم بتجميع ملف أو ملفات الـ asm.
4- بعد التجميع ، نحصل على ملف يسمى relocatable object module, بامتداد obj. (كل ملف asm. يقابله ملف obj. واحد)
5- نقوم باستخدام أي linker (link , tlink , ...) لنربط جميع ملفات الـ obj. ونحصل على الملف التنفيذي.
6- ها قد انتهينا...والناتج هو ملف exe. جاهز للتشغيل.

طبعا لاحظ أنه في كل خطوة كنا نفترض أنه لم يحدث أي خطأ...لان حدوث خطا يتطلب اعادة تلك الخطوة من جديد...هل ترى السهمين باللون الأحمر ؟ كل منهما يعبر عن حدوث خطأ في إحدى المراحل...
أمر آخر...اذا لم يحدث خطأ وحصلنا على ملف exe. لكنه لم يعمل فهنا نحتاج الى استخدام debugger لتنقيح البرنامج واكتشاف موطن الخطأ فيه...وبعد اكتشاف مكان الخطأ نعود للسورس كود ونقوم بتغيير الكود الذي سبب الخطأ.
اذا أردت التوسع في مفهوم المجمع والرابط فهناك العديد من الدروس والمقالات التي تتحدث عن هذا... كما قلت سابقا لا أريد لهذه الدورة أن تطول كثيرا...

مزيد من التوضيح حول الـ effective addresses

قلنا بأن العناوين في المعالجات x86 تتكون من 20 بت ، لكن معظم المسجلات registers في هذه المعالجات تتكون من 16 بت ، إذن كيف سيتم التعامل مع عناوين ذاكرة بحجم 20 بت رغم أن المسجلات حجمها الأقصى هو 16 بت ؟ الحل يكمن في استخدام مسجلين بدلا من واحد...المسجلين سويا حجمهما 32 بت أي أكبر من 20 وبالتالي فقد حلت المشكلة...

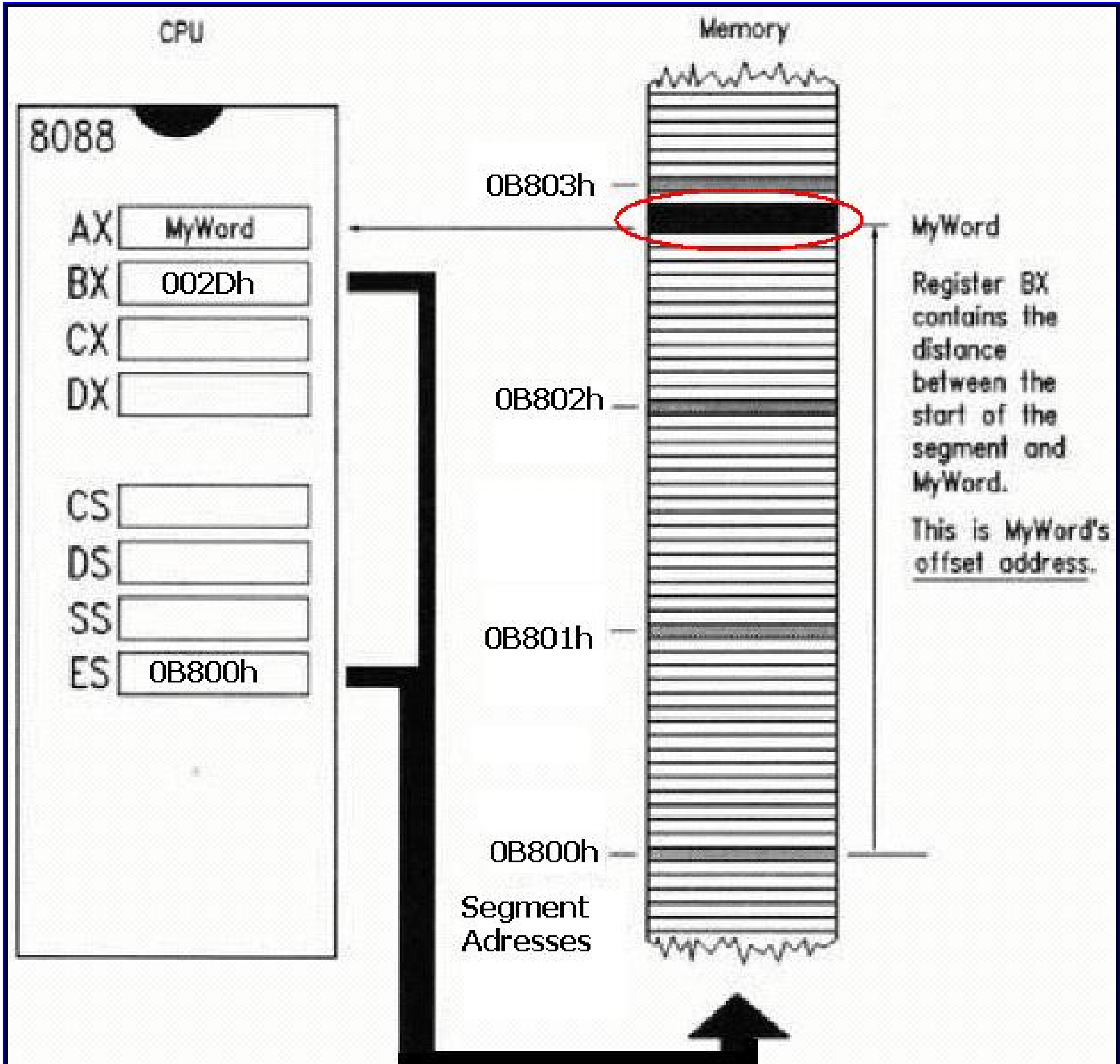
دعنا نرى ما يحدث عند تنفيذ تعليمة مثل

MOV AX, ES:[BX]

كما ترى فإننا نريد نقل بيانات معينة موجودة في الذاكرة (ولنسم البيانات هذا : MyWord) . نريد أن نحدد للمعالج العنوان الذي توجد فيه MyWord ، فكما ترى استخدمنا مسجلين لنصف هذا العنوان كالتالي ES:[BX] ، تذكر ان الصيغة العامة للـ effective addresses هي كالتالي :

Segment: offset

وهذا يعني أن جزء الـ segment من العنوان موجود في ES وجزء الـ offset من العنوان موجود في BX ونريد من المعالج أن يذهب الى هناك ويحضر الـ MyWord ويذهب بها إلى AX . الشكل التالي يوضح لنا كل ما يحدث أثناء تنفيذ التعليمة MOV AX, ES:[BX]



في اليسار ترى المعالج 8088 وترى جزءا من مسجلاته...في اليمين ترى الذاكرة (RAM) وهناك على يسارها عناوين الـ Segment Adresses (الـ Segment Adresses تكون مقسمة evenly الى مجموعات فالسيجمنت تبدأ كل 16 بايت أي 10h.أو دعنا نقول يمكن أن تبدأ كل 16 بايت ، فاذا تذكرنا أن 8088 يمكنه التعامل مع 1mb – أي 1048576 بايت - من الذاكرة ، وبالأخذ بالحسبان أن السيجمنت يمكن أن تبدأ كل 16 بايت ، اذن 1048576 تقسيم 16 الناتج 65536 ، أي أنه هناك 65536 مكانا مختلفا يمكن للسيجمنت أن تبدأ عنده . كما ترى فالفرق بين عنوان كل سيجمنت هو 10h . قد تقول : لكني أرى الفرق 1h ...هناك 0 في النهاية تم الاستغناء عنه من باب التسهيل ، فالعناوين التي على اليسار مثل 0B800h كانت بالأصل 0B8000h لكن الصفر الأخير سيبقى صفرا لأنه كما قلت الفرق بين كل عنوان والذي يليه هو 10h اذن فهناك صفر دائما لذا استغنينا عنه)

إن MyWord التي نريد نسخها موجودة في مكان ما بالذاكرة ، هذا المكان يبعد 02Dh عن السيجمنت 0B800h لذا فإن عنوان هذه الـ MyWord هو 0B800 : 02D . لكن لاحظ أنه يمكننا استخدام سيجمنت أخرى ، ولتكن السيجمنت التي فوقها مباشرة 0B801 عندها ستختلف الإزاحة offset وستصبح 01Dh وبالتالي يمكننا أن نقول أن جميع العناوين التالية لها نفس التأثير :

0B800 : 02D
0B801 : 01D
0B802 : 00D

أعتقد أن الصورة أصبحت أوضح الآن...ملاحظة صغيرة...عن كتابة العناوين بصيغة Segment: offset فيجب ألا تضيف اللاحقة h في نهاية الرقمين التي تدل على ان النظام هو الهكس...

فلنعد الى الصورة مرة أخرى . الآن أصبحنا نعلم أين توجد MyWord و أين قد خزننا عناوينها الاثنين.
بعد تنفيذ التعليمة يكون المعالج قد ذهب الى العنوان المطلوب و أخذ MyWord ووضعها في AX .
لاحظ أنه بإمكانك استخدام أي Segment Register آخر كـ DS مثلا.

إن الـ offset يمكن وضعه فقط في BX,BP,SP,SI and DI .

References:

1 http://www.arabteam2000.com/
2 Assembly Language: Step-by-Step , Jeff Duntemann

الفريق العربي للهندسة العكسية

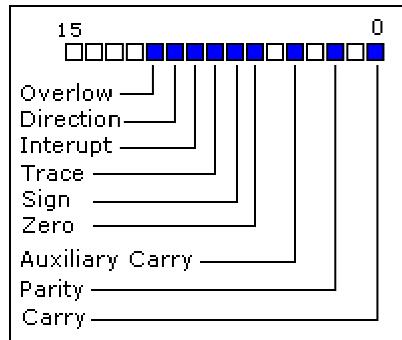
دورة ATRE لتعليم الأسمبلي للمبتدئين من الصفر

الدرس الرابع – التعليمات الحسابية و المنطقية

allko

سنتكلم اليوم عن التعليمات الحسابية(Arithmetic) والمنطقية (Logic)
سنقسم الدرس الى أربع مجموعات بناء على المعاملات (operands) لكل تعليمة .

لكن قبل البدء دعنا نفصل الحديث قليلا عن الـ flags .



كما ترى فإن هذا المسجل هو بحجم 16 بت ، كل بت يسمى flag ويأخذ القيمة 0 أو 1 .
كنت أريد ترجمة الفقرة التالية لكن رأيت أنها مكونة من مصطلحات ولا يوجد ما يحتاج الترجمة...

- **Carry Flag (CF)** - this flag is set to **1** when there is an **unsigned overflow**. For example when you add bytes **255 + 1** (result is not in range 0...255). When there is no overflow this flag is set to **0**.
- **Zero Flag (ZF)** - set to **1** when result is **zero**. For none zero result this flag is set to **0**.
- **Sign Flag (SF)** - set to **1** when result is **negative**. When result is **positive** it is set to **0**. Actually this flag take the value of the most significant bit (MSB).
- **Overflow Flag (OF)** - set to **1** when there is a **signed overflow**. For example, when you add bytes **100 + 50** (result is not in range -128...127).

باقي الرايات ليس من المهم معرفتها (على الأقل بالنسبة لمبتدئ)

First group: ADD, SUB,CMP, AND, TEST , OR, XOR

هذه التعليمات لها المعاملات(operands) التالية

REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate

REG: EAX, AX , AL , AH , EBX , etc...
memory: [BX], [BX+SI+7], variable, etc...
immediate: 5, -24, 3Fh, 10001101b, etc...

- بعد هذه التعليمات ، فالنتيجة تخزن دائما في المعامل الأول . لكن تعليمة **CMP** و **TEST** تؤثر على الرايات فقط ولا تخزن أي شيء .
- الرايات التي تتأثر بهذه التعليمات هي **CF**, **ZF**, **SF**, **OF**, **PF**, **AF**
- عندما يكون هناك two operands فيجب أن يكونا من نفس الحجم ، مثلا لا يجوز أن تكتب mov ecx,ax لانهما ليسا من نفس الحجم.
- أيضا مما يجب ذكره هو أنه اذا رايت قيمة ما كـ operand (معامل) لإحدى التعليمات ، فهذه القيمة هي بالنظام العشري إذا لم يكن هناك لاحقة ، مثلا mov ebx,30 تعني أننا سننقل القيمة 30 عشري ، بينما mov ebx,30h تعني نقل القيمة 30 بالهكس وهكذا.
- في حالة عنوان الذاكرة فيجب أن تحدد الحجم الذي تريده . تذكر أننا قلنا ان الـtwo operands يجب أن يكونا من نفس الحجم ، لذا في حالة عنوان الذاكرة فاننا نبدأ الـ operand بالحجم المطلوب أما dword أو word أو حتى byte ، ثم ptr وهي اختصار لـ Pointer أي مؤشر ، ثم عنوان الذاكرة (DS:[BX] مثلا) . أنظر الى هذه التعليمة مثلا : ADD DWORD PTR DS:[ECX],EAX أو هذه SUB AX , WORD PTR DS:[ECX]

1- تعليمة ADD تقوم بجمع المعامل الثاني الى المعامل الأول .
2- تعليمة Sub تطرح المعامل الثاني من الأول .
3- تعليمة cmp تطرح المعامل الثاني من الأول (لكن النتيجة يتأثر بها الرايات فقط) .
4- دعنا نرى تعليمة AND . هذه تعليمة منطقية . كل عملية منطقية لها ما يسمى بـ Truth Table أو جدول الحقيقة. بالنسبة لتعليمة AND فالـ Truth Table هو : (هذه العملية هي عملية ضرب منطقي لكن بين الصفر والواحد فقط)

1 AND 1 = 1
1 AND 0 = 0
0 AND 1 = 0
0 AND 0 = 0

5- تعليمة TEST تقوم بنفس ما تقوم به AND لكن النتيجة تؤثر على الرايات فقط.

6- تعليمة OR تقوم بعملية منطقية ، الـ Truth Table هو كالتالي :

1 OR 1 = 1
1 OR 0 = 1
0 OR 1 = 1
0 OR 0 = 0

7- تعليمة XOR تقوم بعملية منطقية ، الـ Truth Table هو كالتالي :
(اذا كان المعاملان متشابهين فالنتيجة 0 ، واذا كانا مختلفين فالنتيجة 1)

1 XOR 1 = 0
1 XOR 0 = 1
0 XOR 1 = 1
0 XOR 0 = 0

Second group: MUL, IMUL, DIV, IDIV

هذه التعليمات لها المعاملات(operands) التالية

REG
memory

- تعليمة MUL مسئولة عن الضرب بدون إشارات ، IMUL للضرب بإشارات . نفس الشيء ينطبق على DIV و IDIV .
- تعليمة MUL و IMUL تؤثران على الرايات التالية فقط : **CF , OF**
فعندما تكون النتيجة أكبر من حجم المعاملات ، عندها تحمل هاتين الرايتين القيمة 1 (set to 1) فيما عدا ذلك فتحملان القيمة 0 (set to 0) . أما بالنسبة لتعليمة DIV و IDIV فالرايات غير معروفة القيمة (undefined) .
- القواعد التي يتم على أساسها الضرب في تعليمتي MUL و IMUL هي :

- A **byte** operand is multiplied by **AL**; the result is left in **AX**.
- A **word** operand is multiplied by **AX**; the result is left in **DX:AX**. DX contains the high-order 16 bits of the product..
- A **doubleword** operand is multiplied by **EAX** and the result is left in **EDX:EAX**. EDX contains the high-order 32 bits of the product.

-2 * -4 = 8 = 8 h = 1000 b

```
MOV AL, -2
MOV BL, -4
IMUL BL
; AX = 8 h = 1000 b
```

1700 * 520 = 0D7D20 h = 11010111110100100000 b

MOV AX, 1700
MOV BX,520
MUL BX
; DX =000D h = 0000000000001101 b
; AX= 7D20 h = 0111110100100000 b

12345678h * 99999999 h = AEC33E18EAD65B8 h =
1010111011000011001111100001100011101010110101100101101111000 b

MOV EAX, 12345678h
MOV EBX, 99999999h
MUL EBX
; EDX = 0AEC33E1 = 00001010111011000011001111100001
; EAX = 8EAD65B8 = 10001110101011010110010110111000

لاحظ أنه اذا كتبت العدد كما هو فهذا يعني أنه بالعشري ، اما اذا أضفت اللاحقة h فهذا يعني أنه بالهكس ، بالمثل فإن إضافة اللاحقة b تعني أن العدد بالبايناري.

أما بالنسبة لتعليمة IDIV و DIV فلهما الجدول التالي الذي يوضح الحالات الثلاث للقسمة (تبعا لحجم ما تريد قسمته)

Size	Quotient (الناتج)	Reminder (الباقي)	Dividend (المقسوم عليه)
Byte	AL	AH	AX
Word	AX	DX	DX:AX
dword	EAX	EDX	EDX:EAX

11CCCEEE44BBBAAA h /33A33A33 = 583EF4DF h = 1011000001111101111010011011111 b

MOV EDX, 11CCCEEE h
MOV EAX, 44BBBAAA h
MOV ECX, 33A33A33 h
IDIV ECX
; EAX = 583EF4DF h = 1011000001111101111010011011111 b
; EDX = 15B96C3D h

Third group: INC, DEC, NOT, NEG

هذه التعليمات لها المعاملات(operands) التالية

REG
memory

- تعليمة INC, DEC تؤثران على الرايات التالية : ZF, SF, OF, PF, AF
- تعليمة NOT لا تؤثر على أي من الرايات.
- تعليمة NEG تؤثر على الرايات التالية : .CF, ZF, SF, OF, PF, AF

1- تعليمة INC تقوم بزيادة المعامل بمقدار واحد
2- تعليمة DEC تقوم بإنقاص المعامل بمقدار واحد
3- تعليمة NOT تقوم بعكس كل بت من بتات المعامل (الـ 0 يصبح 1 والعكس صحيح)
4- تعليمة NEG تقوم باستبدال القيمة بالـ two's complement لها .

Fourth group : MOV instruction

هذه التعليمة لها المعاملات(operands) التالية

```
MOV REG, memory
MOV memory, REG
MOV REG, REG
MOV memory, immediate
MOV REG, immediate
```

أما في حالة كان أحد المعاملات segment register ، فهذا هو المسموح فقط :

```
MOV SREG, memory
MOV memory, SREG
MOV REG, SREG
MOV SREG, REG
```

هذه التعليمة تقوم بنسخ المعامل الثاني (المصدر source) إلى المعامل الأول (الوجهة destination)

والآن الى القليل من الأمثلة...لن نكتب البرامج كاملة فنحن لا نعرف بعد كيف يمكن عمل ذلك ، ما سنقوم به هو كتابة الكود الذي ينفذ ما نريده فقط وليس كل البرنامج.

الأمثلة التالية تشمل جميع الأوامر السابق ذكرها

9553h+35h=9588h

```
mov eax,9553h
add  eax,35h
```

111011101b * 100100001b=1000011010011111101b

```
mov eax,111011101b
mul  100100001
```

01100001 AND 11011111 = 01000001

```
mov cx,01100001b
and  cx,11011111b
```

ملاحظة : يمكنك التعامل مع الأحرف ، أي أن تنقل قيمة الـ ascii لحرف ما الى مسجل ما وذلك بوضع الحرف بين علامتي تنصيص مفردتين هكذا :   'a' كما في المثال التالي :

01100001 AND 11011111 = 01000001

```
mov al, 'a'           ; al = 01100001b
and al, 11011111b  ; al  = 01000001b  ('A')
```

203 / 4 = 50.75

```
MOV AX, 203   ; AX = 00CBh
MOV BL, 4
DIV BL        ; AL = 50 (32h), AH = 3
```

NEG 5

```
MOV AL, 5   ; AL = 05h         = 0000 0101
NEG AL      ; AL = 0FBh (-5)  = 1111 1011
NEG AL      ; AL = 05h (5)
```

NOT 00011011b

```
MOV AL, 00011011b
NOT AL        ; AL = 11100100b
```

في هذا المثال ، لنفرض ان القيمة 00011011b موجودة في الذاكرة عند العنوان DS:[2156] عدنها يمكن عمل التالي :

NOT 00011011b

```
NOT [2156]        ; [2156]  = 11100100b
```

الى هنا ينتهي درس اليوم...آمل أن أكون قد وفقت في إيصال المعلومة ببساطة وبدون تعقيد...

Assignment:

Write a code that perform the following arithmetic operation : (with minimum instruction)
[(555AAA h + 10001000100010001 b) * 666777888] / 0E777C h

References:

1- http://www.emu8086.com/
2- http://www.arabteam2000.com/

الفريق العربي للهندسة العكسية

دورة **ATRE** لتعليم الأسمبلي للمبتدئين من الصفر
الدرس الخامس– تعليمات القفز – المتغيرات
allko

درس اليوم يتناول قسمين مهمين...تعليمات القفز إضافة الى المتغيرات...

I- jumping instructions

كما تعلم فإن المعالج يقوم بتنفيذ الأوامر بالترتيب...لكن إن أردنا نقل التنفيذ الى جزء أخر في البرنامج فيمكننا عمل ذلك بواسطة أوامر القفز...تقسم هذه الأوامر الى قفز مشروط conditional jump وقفز غير مشروط unconditional jump .
الجدول التالي يوضح أهم أوامر القفز المشروط وهناك الكثير غيرها سنذكرها لاحقا...

Instruction	*Description*	*Affected flags*
JZ	اقفز اذا كانت النتيجة صفر	ZF=1 → jump
JNZ	اقفز اذا كانت النتيجة لا تساوي صفر.	ZF=0 → jump
JE	اقفز اذا كانت النتيجة صفر	ZF=1 → jump
JNE	اقفز اذا كانت النتيجة لا تساوي صفر.	ZF=0 → jump
JC	اقفز اذا الـ MSB أخرج 1 للخارج (carry out) او احتاج لواحد من الخارج (borrow)	CF=1 → jump
JNC	اقفز اذا الـ MSB لم تخرج 1 للخارج(carry out) او لم يحتاج لواحد من الخارج (borrow)	CF=0 → jump

انتبه الى أن jz تعمل مثل عمل je ، و jnz تعمل مثل عمل jne ...
والآن دعنا نوضح مفهوم الـ carry والـ borrow قليلا...هذه الجزئية من الدرس مقتبسة – بتصرف - من أحد الدروس للأخ afeef من الفريق العربي للبرمجة ، فله الشكر.
أنظر الجدول التالي لتعرف الحالات الأربعة الممكنة لمفهوم الـ carry :

1st number	*2nd number*	*sum(S)*	*Carry (C)*
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

كما ترى ففي حالات الجمع الثلاثة الأولى (الصفوف الثلاثة الأولى) فإن الـ carry يساوي 0 ، لكن الحالة الرابعة عند جمع 1 مع 1 فالناتج هو اثنان أي 10 إذن خانة المجموع (sum) يساوي 0 والـ carry يساوي 1 (بالعامية نقول : باليد واحد ، وهذه بالأسمبلي نقول عنها : carry)

والآن دعنا نرى ما الذي يحدث عند جمع العددين 87H+93H

87H=10000111 *83H=10010011*

					C	C	C		*Carry*
	1	0	0	0	0	1	1	1	*1st number*
	1	0	0	1	0	0	1	1	*2nd number*
1	0	0	0	1	1	0	1	0	*result*
ignored	MSB							LSB	

النتيجة هي 1A h = 00011010 ... والآن ما هي قيمة الراية ZF (Zero Flag) ؟ النتيجة ليست بصفر (واضح !!!) اذن ZF=0 .
والآن ما هي قيمة الراية SF(Sign Flag) ؟ بالتأكيد 0 ، لأن النتيجة موجبة...كيف عرفت؟ راجع الدرس الأول...أنظر الى الـ MSB كما ترى إنه 0 مما يدل على أن العدد موجب.
ماذا عن قيمة الراية CF (Carry Flag) ؟ أنظر الى الخانة الأخيرة في كلا العددين (الـ MSB) ، كلاهما قيمتهما 1 ، وقد قلت للتو أنه في حالة جمع 1 و 1 فالنتيجة 10 ، وكما ترى نضع الـ 0 أما الـ 1 فينقل للخانة التالية (هذا الـ 1 يظهر باللون الأحمر و أسفل منه مكتوب ignored يعني تم تجاهله) لكننا هنا تجاهلناه ولم نعتبره جزءا من النتيجة...لماذا؟ هذا يعتمد على التعليمة...فتعليمة كهذه

```
MOV BL , 83H
ADD BL , 87H
```

تظهر لنا أن النتيجة ستخزن في BL ومعلوم لنا أن حجمه يتسع لـ 8 بتات فقط ، اذن النتيجة ستقتصر على 8 بتات والبت الأخير يذهب الى Carry Flag .
وهذا ما يسمى بمفهوم الـ **overflow**.

أما تعليمة كهذه

```
MOV BX , 83H
ADD BX , 87H
```

فلن يحدث فيها overflow أي أن البت الأخير (1) الذي تجاهلناه في المرة السابقة ، لن نتجاهله هنا بل سنعتبره جزئا من الحل (لأن BX يتسع لـ 16 بت أي لا مشكلة)...أي أن الناتج هو 10A h = 100011010 .
والآن لنتعرف على مفهوم الـ borrow .

1st number	*2nd number*	*sub(S)*	*Borrow (B)*
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

لن أقوم بالمزيد من التوضيح كي لا يخرج الدرس عن غايته الرئيسة...

مثال / دعنا نرى هذا الكود :

```
  CMP BL,2
  JE allko
  MOV DX,44
  JMP at4re
allko:
  MOV BL ,7
at4re:
  NOP
```

في البداية يتم مقارنة BL مع القيمة 2 ، ثم هناك قفزة مشروطة JE ، فإذا تساوا ، سينتقل التنفيذ الى allko ، و allko هو عبارة عن label . كما ترى يمكن وضع label في أي مكان من البرنامج وذلك ببساطة بكتابة الاسم الذي تريد متبوعا بنقطتان رأسيتان.
ماذا ان لم يتساوا؟ لن تنفذ القفزة وبالتالي سيتابع المعالج تنفيذ الأوامر بشكل طبيعي...الأمر التالي الذي سينفذ في حالة عدم المساواة هو mov dx,44 . يليه قفزة غير مشروطة الى الـ label المسمى at4re .
إذن في حالة المساواة سنصل الى تعليمة mov bl,7 ويتم تنفيذها. ثم ننتقل الى التعليمة NOP لتنفيذها.
أما في حالة عدم المساواة سنصل الى تعليمة nop ويتم تنفيذها.
تعليمة nop هي اختصار لـ no operation أي أنها لا تقوم بعمل شيء.

مثال / دعنا نرى هذا الكود ..وظيفته هي نقل القيمة 25H الى تسع خلايا ذاكرة (من 401200 الى 401208) ...هناك عدة طرق لعمل هذه العملية منها الطريقة التالية :

```
    MOV EAX,25H
    MOV ECX,8
BEGINNING:
     MOV BYTE PTR DS:[401200+ECX],EAX
     DEC ECX
     JNZ BEGINNING
      RET
```

في البداية نضع القيمة 25H في أحد المسجلات وليكن EAX ثم نضع القيمة 8 في المسجل ECX (ليس شرطا لكن جرت العادة أن نستخدم ECX كعداد) ثم هناك LABEL باسم BEGINNING ... يليه أمر النقل MOV ، لاحظ أن التوجيه (Directive) المستخدم هو BYTE PTR لأن القيمة التي نود نقلها (25 h) هي بحجم بايت . ثم نقوم بإنقاص قيمة العداد (ECX) بمقدار 1 وبالتالي فإن القيمة 25h سيتم نقلها أول مرة الى 401208 = 8 + 401200 ثم هناك قفزة مشروطة فإذا وصلت قيمة ECX الى الصفر فإن ZF=1 (بالتالي نكون قد نقلنا القيمة الى الأماكن التسعة) ولن يتحقق شرط القفز وسنكمل الى الأمر الذي بعده...
..لكن بالطبع بعد أول عملية نقل وتنفيذ DEC ECX لأول مرة ، فأن ECX=7 اذا الشرط تحقق وبالتالي سنقفز الى BEGINNING
ونقوم بعملية النقل مرة أخرى وهكذا...الى ان نقوم بعملية النقل في المرة الأخيرة عندها 401200+0=401200 ونكون قد انتهينا إذا ECX=0 ← إذا ZF=1 ← الشرط لم يتحقق إذا ← ننفذ الأمر التالي وهو RET وهذا يعني RETURN أي اعادة السيطرة (control) الى نظام التشغيل.

بقي أن نشير الى جزء آخر من تعليمات القفز وتستخدم بكثرة...انتبه الى أنه في هذه التعليمات فاننا نتعامل مع أعداد ذات إشارة إما سالبة و إما موجبة . هذا يعني أن هذه الأوامر تعتبر آخر بت (أي MSB) هو بت الإشارة فاذا كان مساويا لـ 0 فالعدد موجب واذا كان مساويا لـ 1 فالعدد سالب.

CMP OPERAND 1 , OPERAND 2

instruction	description
JG	JMP IF OP1>OP2 (JMP IF GREATER)
JNG	JMP IF OP1 <= OP2
JL	JMP IF OP1<OP2 (JMP IF LESS THAN)
JNL	JMP IF OP1>=OP2
JGE	JMP IF OP1>=OP2 (JMP IF GREATER THAN OR EQUAL)
JNGE	JMP IF OP1<OP2
JLE	JMP IF OP1<=OP2
JNLE	JMP IF OP1>OP2

أما هذه التعليمات فهي خاصة بالأعداد الموجبة فهي تعتبر أن آخر بت (أي MSB) هو جزء من العدد وليس دالا على الإشارة.

CMP OPERAND 1 , OPERAND 2

instruction	description
JA	JMP IF OP1>OP2 (JMP IF above)
JNA	JMP IF OP1 <= OP2
JB	JMP IF OP1<OP2 (JMP IF BELOW)
JNB	JMP IF OP1>=OP2
JAE	JMP IF OP1>=OP2 (JMP IF ABOVE OR EQUAL)
JNAE	JMP IF OP1<OP2
JBE	JMP IF OP1<=OP2
JNBE	JMP IF OP1>OP2

دعنا نوضح الأمر قليلا...
تأمل المقطع التالي :

```
    MOV AL , -1
    CMP AL,0
    JG SECOND
FIRST:
    MOV ECX,3
    RET
SECOND:
    XOR ECX,ECX
    RET
```

إن ***-1*** يتم تمثيلها بالنظام الثنائي (بحجم بايت) كالتالي : 1111 1111 .
هناك عملية مقارنة بين الـ ***-1*** وبين الـ 0 ، واضح أن الشرط لن يتحقق فالـ ***-1*** ليست أكبر من 0 ، اذا لن تنفذ القفزة وبالتالي ستتوجه الى تعليمة MOV ECX,3 فيصبح ECX=3 ثم هناك تعليمة RET مما يعني إرجاع السيطرة الى النظام وإنهاء البرنامج .
لكن ألق نظرة على الكود التالي...نفس السابق لكن استخدمت JA بدلا من JG .

```
    MOV AL , -1
    CMP AL,0
    JA SECOND
FIRST:
    MOV ECX,3
    RET
SECOND:
    XOR ECX,ECX
    RET
```

إن ***-1*** يتم تمثيلها بالنظام الثنائي (بحجم بايت) كالتالي : 1111 1111 لكن بالنسبة لتعليمة JA فهذه لا تعني ***-1*** لأننا هنا نتعامل مع أعداد موجبة اذن البت الأخيرة MSB ليست لتدل على الإشارة بل هي جزء من الرقم...اذن لو حولنا القيمة 1111 1111 الى عشري سنجدها تساوي 255 ...
والآن هل 255 أكبر من 0 ؟ بكل تأكيد...اذا فقد تحقق الشرط...وسننتقل الى التعليمة XOR ECX,ECX التي ستجعل ECX=0 ثم هناك تعليمة RET للعودة للنظام..

اذن في حالة تعليمة JG حصلنا على ECX=3 ، لكن في حالة JA حصلنا على ECX=0

II- variables

المتغير هو مكان في الذاكرة له اسم وحجم معينين. يتم تعريف المتغيرات في مقطع data. من البرنامج .
أنظر الى الجدول التالي :

db	Define byte	1 byte = 8 bit
dw	Define word	2 byte = 16 bit
dd	Define double word	4 byte = 32 bit

- يتم تعريف متغير باسم allko (مثلا) و بحجم word (مثلا) وبقيمة ابتدائية 4445h (مثلا) كالتالي :

Allko dw 4445h

- ماذا لو أردنا تعريف نفس المتغير لكن لا نريد إعطائه أي قيمة ابتدائية ؟ ستكون الصيغة كالتالي :

Allko dw ?

علامة الاستفهام تشير الى عدم وجود قيمة ابتدائية..لاحظ أنه اذا لم تعط قيمة ابتدائية للمتغير فعليك تعريفه في مقطع ?data.

- أيضا يمكنك تعريف مصفوفة (array) كالتالي :

xxx dd 7,5,4,8,1

هنا قمنا بتعريف مصفوفة مكونة من 5 عناصر وكل عنصر بحجم dword كما هو واضح.
و للوصول الى العنصر الثالث مثلا يمكننا عمل التالي :

Add xxx[2],400

لاحظ أن أول عنصر هو xxx[0] والثاني xxx[1] وهكذا...

- لنفرض أننا أردنا حجز 4 بايتات من الذاكرة ونريد وضع القيمة الابتدائية 450 في كل بايت ، ونريدهم جميعا تحت نفس الاسم...هذا ما علينا فعله :

Alalme db 450,450,450,450

لكن هناك طريقة أسهل كالتالي :

Alalme db 4 dup (450)

يمكننا عمل نفس الشيء لكن دون إعطاء قيمة ابتدائية...كالتالي:

Alalme db 4 dup (?)

- أما النصوص فيمكن تعريفها كالتالي :

X db 'A'

لاحظ أنه يجب تعريف الأحرف والنصوص (strings) كمتغيرات بحجم بايت ، أي يجب استخدام db .
ولاحظ أن الحرف أو الـ string توضع بين بين علامتي تنصيص مفردتين .

- تعريف string لا يختلف عن تعريف حرف...لاحظ :

msg db 'allko is da best',0

الفرق الوحيد هو وجود الصفر (NULL) في النهاية ليدل على انتهاء الـ string ...إن كنت قد تعاملت مع أي من لغات المستوى العلوي من قبل فأنت تعرف عن ماذا أتحدث.

- أمر آخر يجب عليك معرفته ، هو أنه بدلا من كتابة الحرف A (أو أي حرف آخر) يمكنك استخدام القيمة المقابلة له في جدول ASCII ...

Y db 65

الى هنا يكون درس اليوم قد انتهى. أتمنى منكم الدعاء لي بالتوفيق بالامتحانات (:

Assignment:

I- What is the value of the SF , CF , ZF and OF flags after the following code is carried out?

```
MOV BX , 81E1 h
ADD BX , 0C0FD h
```

II- what is the value of EAX after the following code is carried out ?

```
     MOV BX , -5
     CMP BX , 5
     JA SECOND
FIRST:
     MOV EAX,3
     RET
SECOND:
     AND EAX,0
     RET
```

III- Rewrite the following code without changing what it do.

```
     MOV EAX,25H
      MOV ECX,8
BEGINNING:
      MOV BYTE PTR DS:[401200+ECX],EAX
      DEC ECX
      JNZ BEGINNING
       RET
```

IV - define a word-sized variable , named "allko" , with a 8100 h initail value.

V - define a word-sized variable , named "allko2" , without an initail value.

VI – define an array of character , a,b,c and d .

VII – define a string of your choice.

References :

1- www.arabtem2000.com
2- www.emu8086.com

درس اليوم يتحدث عن تعليمات الدوران والإزاحة... سنتناول أربع تعليمات فقط ، لكن هناك العديد من التعليمات الأخرى ...لا مجال للحديث عنها جميعا لذا سنتناول أهمها فقط...
أيضا سنتعرض لبعض الأوامر الأخرى...

ROL

هذا الأمر هو اختصار لـ Rotate Left أي تدوير لليسار ...لن أكثر من الكلام فالشكل التالي سيوضح كل شيء بعد تنفيذ **ROL AX,1**

ROL AX,1

Before	1	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0
After	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0	1

CF :

1

لاحظ كيف أن آخر بت (MSB) تم تدويره و أصبح أول بت (LSB) ، أيضا فإن هذا البت قد ذهبت نسخة منه الى CF كما هو واضح.

ROR

نفس الفكرة في حالة ROL لكن الفرق أن اتجاه الدوران هو لليمين. لاحظ ما يحدث عند تنفيذ أمر **ROR CH,3** (هذا الأمر يعادل تنفيذ أمر ROR CH,1 ثلاث مرات متتالية)

Before:

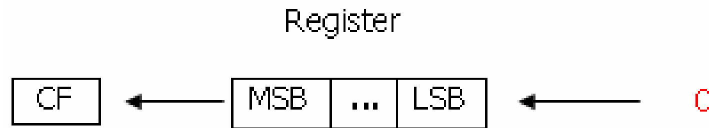
before	0	0	1	1	1	1	1	1
	1	0	0	1	1	1	1	1
	1	1	0	0	1	1	1	1
after	1	1	1	0	0	1	1	1

CF :

1

SHL

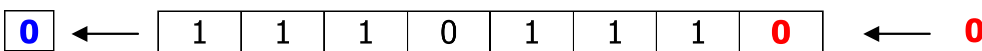
هذا الأمر يقوم بعمل إزاحة لليسار (Shift Left). يتم إضافة 0 من اليمين (إلى LSB) ويذهب الـ MSB الى CF . أي هكذا :



فمثلا عند تنفيذ هذه التعليمة : SHL DL,1 يتم التالي:

Before:

After:
CF:



نقطة هامة في أوامر الإزاحة... إن تنفيذ أمر SHL لمرة واحدة يقوم بضرب العدد (الذي سيتم تدويره) بـ 2 .
أنظر التالي :

```
shl    ax, 1   ;Equivalent to AX*2
shl    ax, 2   ;Equivalent to AX*4
shl    ax, 3   ;Equivalent to AX*8
etc...
```

لكن انتبه ، فيجب ان يكون المسجل المستخدم في أمر SHL كافيا لاستيعاب نتيجة الضرب...

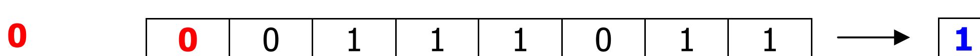
SHR

نفس المبدأ ...لاحظ :

دعنا نرى هذا المثال : SHR AL,1 ...لاحظ كيف يخرج الـ LSB الى الـ CF ...وكيف يأتي 0 ليحل محل الـ MSB ...

Before:

After:
CF:



إن تنفيذ أمر SHR لمرة واحدة يعني قسمة القيمة التي سيتم تدويرها على 2 ...

```
shr    ax, 1          ;Equivalent to AX/2
shr    ax, 2          ;Equivalent to AX/4
shr    ax, 3          ;Equivalent to AX/8
etc…
```

ملاحظات عامة :

- *في معالجات 8086/8088 لا يمكنك التدوير إلا بمقدار 1 . أي أن تعليمة كـ ROL AX,3 غير مسموح بها ، لكن بدئا من معالج 80286 أصبح التدوير بأكثر من 1 ممكنا. هذا الأمر ينطبق على تعليمات ROR / ROL / SHR / SHL*

- *إن تنفيذ تعليمة مثل SHL ECX,32 هي مجرد مضيعة للوقت...فهذه سوف تجعل قيمة ECX صفرا...لأنه هناك 32 عملية إزاحة وكل عملية تقوم بإدخال 0 من اليمين ، في النهاية ستستبدل الـ 32 بت في ECX بـ 32 صفرا...أيضا فتعليمة مثل SHL ECX,33 ستؤدي الى نفس النتيجة .باختصار فإن الإزاحة لليمين أو لليسار يجب أن تتم بعدد مرات أقل من حجم المسجل الهدف و إلا فهي مضيعة للوقت .*

- *بالمثل فإن تعليمة مثل ROR BX,16 هي أيضا مضيعة للوقت فتدوير المسجل BX 16 مرة يعني أنه سيعود إلى حالته الأولى...لاحظ أيضا أن ROR BX,17 تكافئ ROR BX,1 تماما. الفكرة نفسها تنطبق على تعليمة ROL .*

- *عملية القسمة على مضاعفات 2 تنفع في حالة كانت القيمة التي سيتم تدويرها هي unsigned أي بدون إشارة..أما ان كانت ذات اشاراة فالنتيجة لن تكون صحيحة. في حالة قمت بعملية الإزاحة لليمين لمرة واحدة ، أي قمت بعملية القسمة على 2 ، فإن كان هناك باقي فستجده في CF ...لكن لو نفذت عملية الإزاحة (ألقسمة) أكثر من مرة فلن تستطيع الحصول على الباقي* .

- *إن تنفيذ تعليمة مثل SHL AX,8 هو مكافئ لتنفيذ التعليمتين التاليتين :*

```
MOV AH,AL
MOV AL,0
```

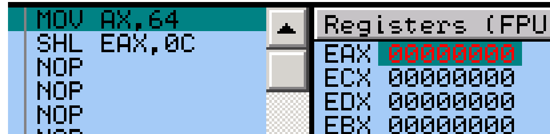
- *إن تنفيذ تعليمة مثل SHR BX,8 هو مكافئ لتنفيذ التعليمتين التاليتين :*
هو مكافئ لتنفيذ التعليمتين التاليتين:

```
MOV BL,BH
MOV BH,0
```

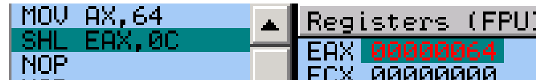
مثال / انظر الى هذه الصور المأخوذة من برنامج OLLY والتي توضح عملية تنفيذ هاتين التعليمتين :
MOV AX,64
SHL EAX,0C

لاحظ أنه في برامج التنقيح كـ OLLY فيفترض أن الأعداد بنظام HEX بينما في برامج التجميع كـ MASM فيفترض أن الأعداد بنظام DECIMAL . أي أنه لكتابة الكود السابق في برنامج MASM فيكتب كالتالي :
MOV AX,100
SHL EAX,12

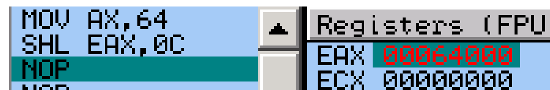
هذه هي الصورة الأولى توضح كيف قمت بفتح برنامج ما وغيرت أول تعليمتين فيه...



والآن F8 :



F8 مرة أخرى



أي أننا قمنا بضرب 64 h بـ ((2 مرفوعة للقوة C)) أي 64 h * 1000 h = 64000 h
أو بالعشري ، ضربنا 100 بـ((2 مرفوعة للقوة 12)) أي 100 * 4096 = 409600

CALL & RET

يتم استدعاء الدالة بمساعدة هذا الأمر...
الامر call يضع عنوان الرجوع في المكدس (stack). عنوان الرجوع بطول 16bit او 32bit حسب نوع الدالة.

الامر ret n يستخدم هذا الامر للرجوع من الدالة.
الرمز n يمثل عدد البايتات (bytes) التي يجب "تنظيفها" في المكدس عند الرجوع من الدالة.
المقصود بعملية التنظيف هو القفز عن البايتات وذلك عن طريق تغيير مؤشر المكدس المسجل ESP
اذا اردنا ان لا نقوم بعملية تنظيف بايتات في المكدس نسجل الامر ret بدون n

```
Call function
…
…
…
ret
```

Assignment:

I - write a code that multiply 62 by 8
II - write a code that divide 4000 by 16
III- what is the value of CF after the following code is carried out

```
Mov ax,10
Shl  ax,12
```

References:

- http://www.arl.wustl.edu
- http://www.arabteam2000.com

الفريق العربي للهندسة العكسية

دورة **ATRE** لتعليم الأسمبلي للمبتدئين من الصفر
الدرس السابع- keygening I
allko

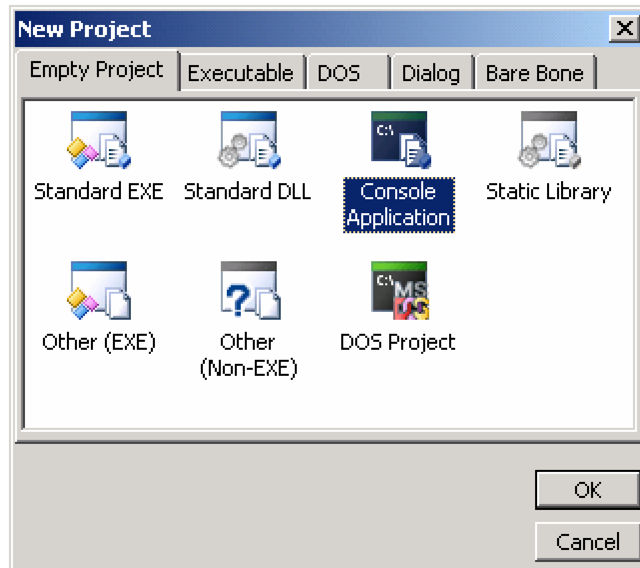
هذا الدرس سيكون مخصصا لشرح كيفية كتابة keygenerater بلغة الاسمبلي.. أتمنى أن يكون الدرس مفيدا...وأود قبل أن أبدا الدرس أن أتوجه بالشكر لـ *Goppit* فقد استفدت كثيرا من كتابه عن الاسمبلي...

أنوه أن هذا الدرس لن نقوم فيه بعمل keygen بل سأقوم بتوضيح كيفية عمل بعض الأمور الأساسية ومن ثم في الدرس التالي (8 و 9) سنقوم بصنع الـ KeyGen سوية.

CONSOLE

ما أعنيه بهذه الكلمة أن البرنامج عند عمله ستفتح لك نافذة كنافذة الدوس يعمل منها البرنامج...لا يوجد واجهات مرئية...أنوه إلى أننا سنركز على البرامج ذات الواجهة المرئية الـ *GUI* ...

ما نحتاجه بداية هو أن تقوم بتثبيت برنامج masm32 . أيضا يلزم تثبيت برنامج WinAsm ...قم بتثبيته بطريقة عادية والآن شغله ثم من قائمة File اختر new project لترى التالي :



كما ترى عليك اختيار Console Application . ثم اضغط انتر أو OK . والآن في المكان المخصص لكتابة الكود قم بكتابة الكود الذي تراه في الصورة التالية (ملاحظة : كان من الممكن ان اضع الـ source code بالمرفقات وبالتالي تقوموا بعمل copy و paste بكل سهولة لكن أريدكم ان تتعودوا على الكتابة بالاسمبلي...) :

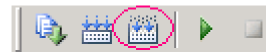
```
.386
.MODEL flat, stdcall
OPTION CASEMAP:NONE

Include windows.inc
Include kernel32.inc
Include masm32.inc
IncludeLib kernel32.lib
IncludeLib masm32.lib

.data
HelloMsg DB "Hello World", 0
CRLF     DB 00Ah, 00Dh, 0
ExitMsg  DB "Enter to Exit", 0

.data?
buffer  DB ?

.code
Start:
invoke StdOut, addr HelloMsg
invoke StdOut, addr CRLF
invoke StdOut, addr ExitMsg
invoke StdIn, addr buffer, 1
invoke ExitProcess, 0
End Start
```

سنقوم بتحليل الكود سطرا سطرا...

386.
هذا السطر يخبر masm32 ان يستخدم مجموعة التعليمات الخاصة بالمعالج 80386 . هذا السطر لن نغيره.
.model flat, stdcall
لن نتكلم عن ماهية الـ model . ولا عن الـ stdcall . اطمئن فهذا السطر ايضا لن يتغير في جميع برامجنا.
option casemap:none
هذا السطر يعني ان الـlabels (ستشرح لاحقا) ستكون حساسة لحالة الأحرف.
include windows.inc
يخبر masm32 بأن يقوم بتضمين windows.inc في ملفنا (وكأننا كتبنا محتوياته هنا) .
include kernel32.inc
نفس الشيء
include masm32.inc
نفس الشيء
includelib kernel32.lib
يخبر الـ linker أن يربط برنامجنا مع ملف kernel32.lib
includelib masm32.lib
نفس الشيء
.data
تعني بداية مقطع البيانات
HelloMsg db "Hello World",0
هنا نقوم بتعريف string (سلسلة من الأحرف) اسمها MsgBoxCaption ومحتوياتها هي ما بين علامتي التنصيص...الصفر بالنهاية يعني أن الـ string ستنتهي بـ 0 (كما هو الحال في الـ ++c)
CRLF db 00Ah, 00Dh , 0
هنا نقوم بتعريف متغير باسم CRLF (CARRIAGE RETURN , LINE FEED) وهاذان المتغيران معا يقومان بانزال مؤشر الكتابة سطرا واحدا للأسفل.
ExitMsg db "Enter to Exit",0
سبق شرح سطر مماثل له.
.data?

تعني بداية مقطع البيانات (المتغيرات) التي لا قيمة ابتدائية لها.
Buffer db ?
متغير ليس له قيمة ابتدائية.
.code
تعني بداية مقطع الكود التنفيذي
start:
كلمة start عبارة عن label (لأنه يوجد بعدها نقطتان رأسيتان) وهي تشير إلى بداية الكود...يمكن اختيار أي كلمة أخرى كـ beginning مثلا...
invoke StdOut , addr HelloMsg
هنا نستدعي دالة StdOut المسؤولة عن إظهار string ما ، أما addr HelloMsg فهي عنوان السترنج التي نود عرضها. أي أن هذا هو البارامتر الوحيد لهذه الدالة.
invoke StdOut , addr CRLF
نفس الشيء...هنا السترنج التي سنعرضها هي CRLF أي اننا سننزل مؤشر الكتابة للأسفل بمقدار سطر واحد.
invoke StdOut , addr ExitMsg
سبق شرح سطر مماثل.
invoke StdIn , addr buffer,1
هنا نستدعي دالة StdIn الخاصة باستقبال النص من المستخدم. في الواقع نحن لا نريد استقبال أي نص لكن لولا هذه الـ "إضافة" لأغلق البرنامج فورما يتم تشغيله لأن عرض السترنج لا يستغرق إلا ثوان معدودة. أما الـ 1 فهو البارامتر الثاني للدالة وهو يشير الى أننا سنستقبل محرفا (character) واحدا من المستخدم.
invoke ExitProcess, 0
هنا نستدعي دالة ExitProcess الخاصة بإنهاء البرنامج ونقوم بإعطائها البارامتر الوحيد وهو 0 .
end start
هذه تشير إلى نهاية البرنامج . لاحظ اننا كتبنا start لاننا كتبناها في الاعلى.فما يكتب بالاعلى نكتبه هنا لكن بدون النقطتان الرأسيتان .

والآن دعنا نجرب برنامجنا...سترى بالشريط العلوي بعض الأزرار كهذه

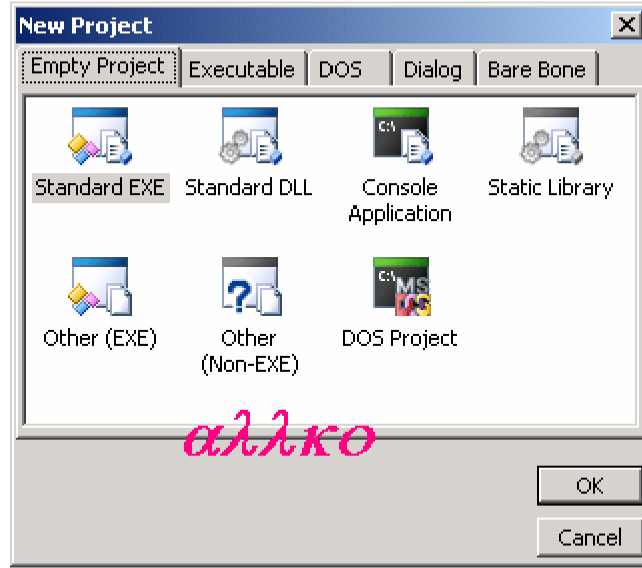


اضغط على زر Go ALL الموضح بالصورة..او يمكنك فتح قائمة Make ومن ثم اختر Go All (هذا الخيار يشمل خيارين اثنين : الاول assembe والثاني link) . بعد الضغط عليه ستخرج لك نافذة تطلب منك حفظ الملفين (الملف الاول ملف المشروع والثاني ملف السورس كود) . **الأفضل** ان تحفظ الملفين بنفس الاسم وفي نفس مجلد WinAsm .
والآن بعد الحفظ سترى التالي :

```
E:\WinAsm\console\console.exe
Hello World
Enter to Exit
```

GUI

قم بفتح مشروع جديد...



إن خيار Standard EXE هو الـ default أي يكون محددا تلقائيا لذا اضغط ok .
والآن تظهر شاشة فارغة...سنقوم بعمل اول برنامج لنا...برنامج message box . لذا قم بكتابة الكود الذي تراه في الصورة التالية :

```
.386
.model flat, stdcall
option casemap:none

include windows.inc
include kernel32.inc
include user32.inc
includelib kernel32.lib
includelib user32.lib

.data
MsgBoxCaption  db "www.alalame.net/vb",0
MsgBoxText     db "allko",0

.code
start:
push MB_OK
push offset MsgBoxCaption
push offset MsgBoxText
push NULL
call MessageBox

push NULL
call ExitProcess

end start
```

αλλκο

سنقوم بتحليل الكود سطرا سطرا... لكن سنشرح فقط الأسطر الجديدة والتي لم يتم شرحها في البرنامج السابق.

push MB_OK
هنا يتم دفع البارامتر الرابع من بارامترات دالة MessageBox . سنتحدث عنه لاحقا.
push offset MsgBoxCaption
هنا يتم دفع عنوان (offset) الـ string التي ستظهر في أعلى المسج . هذا هو البارامتر الثالث
push offset MsgBoxText
هنا يتم دفع عنوان (offset) الـ string التي ستظهر في وسط المسج . هذا هو البارامتر الثاني.
push NULL
هذا أول بارامتر . كلمة NULL تعني صفرا أو لا شيء. يمكن استبدالها بـ 0 .
call MessageBox
هنا يتم استدعاء الدالة بعد ان دفعنا بارامتراتها الأربعة (يتم تخزين البارامترات في الـ stuck)
push NULL
هنا يتم دفع البارامتر الوحيد لدالة ExitProcess
call ExitProcess
هنا نستدعي دالة ExitProcess الخاصة بإنهاء البرنامج

والآن دعنا نجرب برنامجنا...
اضغط على زر Go ALL... بعد الضغط عليه ستخرج لك نافذة تطلب منك حفظ الملفين (الملف الاول ملف المشروع والثاني ملف السورس كود) . **الأفضل** ان تحفظ الملفين بنفس الاسم وفي نفس مجلد WinAsm .
والآن بعد الحفظ سترى التالي :

اذا كان هذا أول برنامج أسمبلي لك...فــ...مبارك !!!

اذا اردت التفاصيل عن السطر الثاني فاقرا كتاب اسمبلي..او اقرا مقالات Xacker في منتديات الفريق العربي للبرمجة فمقالاته مختصرة ومفيدة. ان دالة MessageBox التي استخدمناها تأخذ 4 بارامترات كما لاحظت. اول بارامتر يتعلق بالـ style لهذه المسج (النافذة) .
الأن سنعمل تعديل على الكود ...لاحظ اننا استخدمنا توجيه offset للحصول على عنوان للـ string . هناك طريقة اخرى كالتالي :

```
push MB_OK
lea eax,MsgBoxCaption
push eax
lea eax,MsgBoxText
push eax
push NULL
call MessageBox
```

ان تعليمة lea تعني load effective address أي انها تستخدم للحصول على عنوان لـ string او غيرها...كما ترى فاننا نخزن العنوان في المسجل eax (يمكنك اختيار أي مسجل اخر) ومن ثم ندفع تلك القيمة إلى المكدس(stuck) بواسطة امر الـ push .
لاحظ اننا استخدمنا طريقة push و call لاستدعاء الدالة سواء في الاسلوب الاول او الثاني...لكن حيث ان المجمع (assembler) الذي يستخدمه برنامج winasm هو masm32 (بالمناسبة فهو يدعم مجمع fasm أيضا) فيمكننا استخدام صيغة اخرى. في الواقع فان masm32 يقدم صيغا تجعل الاسمبلي في منتهى السهولة . لاحظ التالي : يمكن الاستغناء عن :

```
push MB_OK
push offset  MsgBoxCaption
push offset MsgBoxText
push NULL
call MessageBox
```

بسطر واحد...وهو :

```
invoke MessageBox, NULL, addr MsgBoxText, addr MsgBoxCaption, MB_OK
```

لاحظ مدى سهولة هذه الصيغة...كلمة invoke هي عبارة عن توجيه (directive) ولا يمكن استخدامها مع المجمعات الاخر كـ tasm مثلا...فهي مخصصة لـ masm . لاحظ الترتيب الذي تكتب فيه البارامترات. أيضا لاحظ كلمة addr التي تشير إلى دفع عنوان الـ string . انتبه إلى انه لا يمكنك استخدام كلمة offset في صيغة invoke .
نفس الشيء ينطبق على الدالة الأخرى فيمكن كتابتها كالتالي :

```
invoke ExitProcess,0
```

الآن قم بالتجريب...احذف الصيغ السابقة واستبدلها بهذه...كما ترى حصلنا على نفس النتيجة...
ملاحظة : من الآن فصاعدا سنستخدم طريقة الـ invoke فهي اسهل بكثير...لكن مع ذلك انصحك وبشدة بتطبيق الطريقتين الاخرتين كي تتعلم اكثر...)
أما بخصوص MB_OK فهي كما قلت عبارة عن نمط الـ messagebox . جرب ان تستبدلها بـ MB_OKCANCEL ستجد أنه بدل وجود زر ok فقط أصبح هناك ok و cancel . الآن جرب
MB_YESNO or MB_ICONASTERISK or MB_DEFBUTTON2 . كلمة or يمكنك استبدالها بإشارة + . أي اننا سننفذ الأنماط الثلاثة معا. لاحظ انه اثناء كتابتك لكل نمط سيخرج لك مربع من قبل winasm بحيث يمكن اختيار النمط الذي تريد من قائمة بجميع الأنماط المتوفرة. الآن نفذ السابق لتحصل على :

لاحظ...النمط الأول يعني وجود زرين yes,no والنمط الثاني يعني وجود أيقونة خطأ (التي تراها باللون الأحمر) والنمط الثالث يعني أن الزر الثاني هو الزر المختار افتراضيا.(ملاحظة : قمت بتغيير الكلام الذي يظهر بالأعلى وبالوسط...).
قلت أن مجمع masm له العديد من المزايا...من بينها هو إمكانية استخدام بعض الـ directives التي تستخدم مع لغات المستوى العلوي...هذه المجموعة تشمل .if و .elseif و .while وغيرها...أكثر ما يهمنا هو الأولى والثانية...
لاحظ معي:

The C version :	**The MASM version:**

if (var1 == var2)
{
//code goes here
}
else
if (var1 == var3)
{
//code goes here
}
else
{
//code goes here
}
	.if (var1 == var2)
	;code goes here
	.elseif (var1 == var3)
	;code goes here
	.else
	;code goes here
	.endif

جملة الشرط تبدأ بـ if. وتنتهي بـ endif. كما هو واضح..إذا كان هناك حالات أخرى فنستخدم elseif.
ستتضح الصورة أكثر في الدرس القادم حينما يتم استخدام هذه الصيغ في كتابة dialog procedure

Assignment:
I- write a console program that receive the name of the user then print it , like this :

```
E:\WinAsm\console\console.exe
Please enter your name :allko
allko  is da best
Enter to Exit_
```

note that the program should take the name then print it followed by the sentence "is da best" .

II- write a simple message box GUI program.

References:
ARTeam Win32 Assembly for Crackers – Goppit

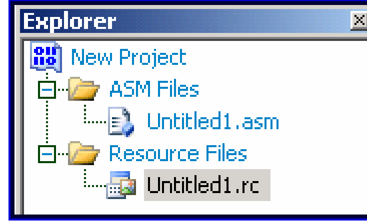
الفريق العربي للهندسة العكسية

دورة **ATRE** لتعليم الأسمبلي للمبتدئين من الصفر

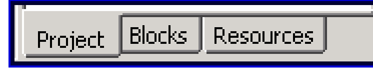
الدرس الثامن- keygening II

allko

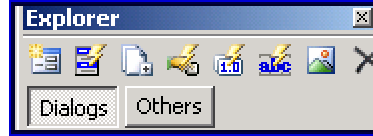
اليوم سنتعلم شيئا جديدا...سنرى كيف يمكن عمل قوائم وأزرار وغيرها بالاسمبلي...
أنشئ مشروعا جديدا كما فعلنا في الدرس السابق...والآن من قائمة Project اختر add new Rc ...لاحظ ظهور شاشة جديدة بخلفية زرقاء...أيضا فانه في الـ explorer bar سترى التالي: (ان لم يكن الـ explorer bar ظاهرا لديك فمن قائمة view اختر explorer



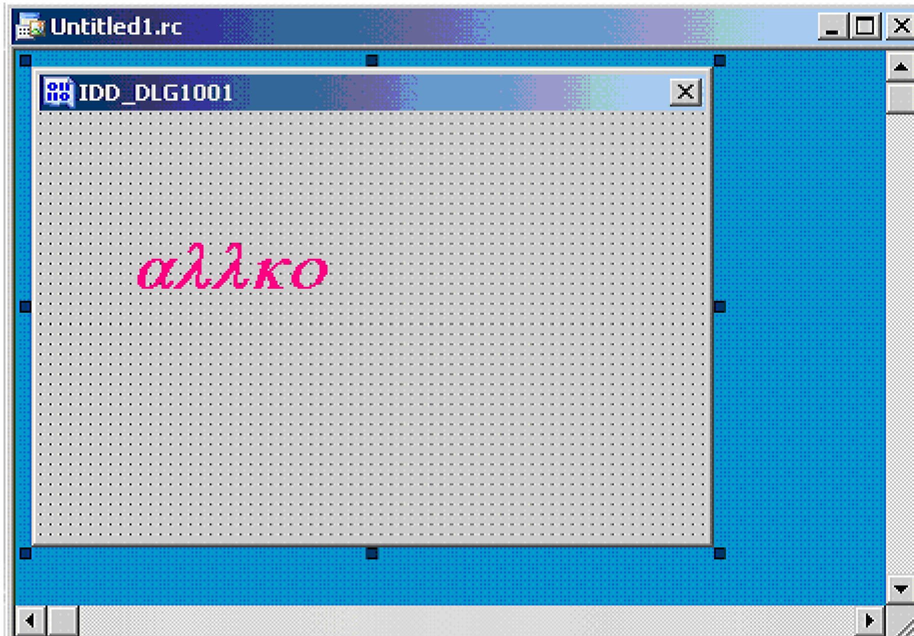
لاحظ ان المشروع الآن يتكون من قسمين...قسم مخصص للـ source code (ASM Files) وقسم مخصص للـ Resources (Resources Files) . الآن في أسفل الـ explorer bar سترى التالي :



اختر Resources . الآن تغير الـ explorer bar وأصبح كما في الصورة التالية :



والآن اختر أول زر من اليسار (Add new dialog) ..يفترض أن ترى التالي في شاشة الـ resources :



الآن نريد إضافة بعض الأمور لهذا الـ dialog ...من قائمة view اختر Tool Box و Dialog .
سيظهر شريط الـ Tool Box و شريط الـ Dialog .

اختر سادس زر من اليسار في شريط ToolBox وهو خاص بإضافة الأزرار. ستلاحظ ان مؤشر الماوس تغير فأصبح على شكل علامة + . قم برسم الزر في أي مكان من الـ dialog وليكن في الأسفل.
ستلاحظ أنه أصبح بالشكل التالي :

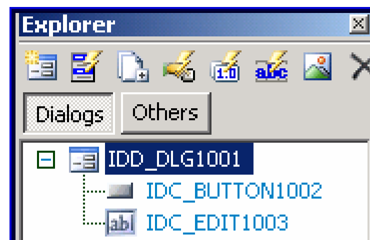
أما في نافذة الـ explorer bar فسترى التالي :

Name	IDC_BUTTON1002
ID	1002
Left	7
Top	98
Width	86
Height	22
Style	0x50010000
ExStyle	0x00000000
Visible	TRUE
Caption	IDC_BUTTON1002

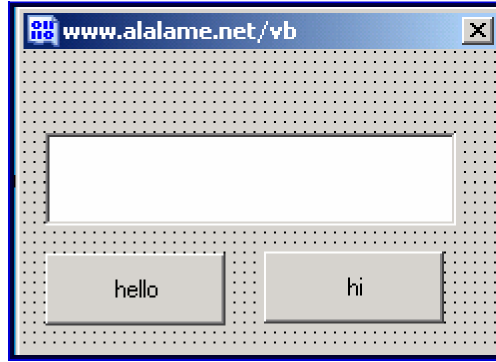
Project | Blocks | Resources

أول خانة هي خانة الاسم...لا تغيرها...ثانية خانة هي الـ ID الخاص بالزر...فكل resource له ID خاص به. آخر خانة Caption هي الكلام المكتوب على الزر. اخترها وامسك الكلام واكتب أي شيء وليكن : hello . (ملاحظة : يمكنك الكتابة بالعربية على الرغم من أن الحروف سوف تظهر بشكل غريب إلا انه عند عمل assembling and linking للبرنامج ستظهر الكتابة العربية بشكل سليم) .
نعود إلى الـ toolbox والآن اختر رابع عنصر في الشريط (من اليسار) وهو عنصر edit . والآن ارسم مربع في وسط الـ dialog فوق الزر .
تلاحظ ظهور مربع تحرير (edit) والآن من الـ explorer bar اختر أخر عنصر ألا وهو text وامسح الكلام المكتوب واتركه فارغا (لا تكتب شيئا) . سيصبح الـ dialog بهذا الشكل (ملاحظة : يمكنك تغيير حجم الـ dialog ...انقر في أي مكان عليه لترى بأن الزوايا اصبح بها نقاط....ضع الماوس عليها وتحكم بالحجم وقم بتصغيره او تكبيره كما تشاء).

الآن دعنا نغير الكلام الموجود بالأعلى (IDD_DLG1001) من الـ explorer bar سترى بالأعلى التالي :



أول عنصر هو الـ dialog والثاني الزر والثالث مربع التحرير . انقر على الأول واذهب إلى عنصر Caption وغير الكلام الى : www.alalame.net/vb . أضف زرا آخر – بنفس الطريقة – باسم hi .



والآن قد تريد إضافة الـ dialogs من نماذج جاهزة فما الذي يجب فعله؟ انظر أسفل قائمة Make هناك زر كالتالي :

اضغط الزر اليمين ...كي ننتقل الى وضع visual mode : off . ستلاحظ اختفاء الـ dialog وبدلا منه هنالك التالي :

```
;This Resource Script was generated by WinAsm Studio.

#define IDD_DLG1001 1001
#define IDC_BUTTON1002 1002
#define IDC_EDIT1003 1003
#define IDC_BUTTON1004 1004

IDD_DLG1001 DIALOGEX 0,0,158,92
CAPTION "www.alalame.net/vb"
FONT 8,"MS Sans Serif"
STYLE 0x10cc0000
EXSTYLE 0x00000000
BEGIN
	CONTROL "hello",IDC_BUTTON1002,"Button",0x50010000,7,62,60,22,0x00000000
	CONTROL "",IDC_EDIT1003,"Edit",0x50010080,7,25,137,28,0x00000200
	CONTROL "hi",IDC_BUTTON1004,"Button",0x50010000,80,62,60,22,0x00000000
END
```

إذن لإضافة زر يمكنك كتابة بضعة اسطر برمجة...لكن بلا شك فالعمل من خلال الوضع المرئي اسهل بكثير.
أيضا اذا كان هناك dialog جاهز تريد إضافته فانسخه والصقه هنا .
الآن من أسفل شريط الـ explorer bar اختر Project

والآن في أعلى الـ explorer bar اختر ملف untiteled1.asm . الآن ما نريد عمله هو عندما نضغط على ذلك الزر نريد ان تظهر جملة في المربع الأبيض...الكود المطلوب سيكون كالتالي :

```
.386
.model flat,stdcall
option casemap:none

include windows.inc
include kernel32.inc
include user32.inc
includelib kernel32.lib
includelib user32.lib

mydialog proto :DWORD,:DWORD,:DWORD,:DWORD

.data
msg1 db  "allko is the best",0
msg2 db "hello",0
.data?
hinstance HINSTANCE  ?

.code
start:
        invoke GetModuleHandle,NULL
        mov hinstance,eax
        invoke DialogBoxParam,hinstance,1001,NULL,addr mydialog,NULL
        invoke ExitProcess,NULL

mydialog PROC hwnd:HWND , uMsg:UINT,wParam:WPARAM, lParam:LPARAM
mydialog EndP

end start
```

αλλκο

كالمعتاد سأشرح الكود سطرا سطرا...أول سطر غير مألوف هو

```
mydialog proto :DWORD,:DWORD,:DWORD,:DWORD
```

في لغات البرمجة الأخرى ، عندما تريد عمل function مثلا فيجب ان تقوم بالإعلان (declaration)عنها ، أيضا هنا يجب أن نعلن عن هذه الإجرائية(procedure) . في هذا السطر عرفنا إجرائية سميناها mydialog ومن ثم كتبن التوجيه proto ومن ثم البارامترات لهذه الإجرائية ، وكما ترى جميع البارامترات من نوع DWORD=Double Word.
هذا السطر لن يتغير لجميع الـ dialogs .
السطر غير المألوف التالي هو

```
.data?
hinstance HINSTANCE  ?
```

إن .data? هو مقطع للبيانات مثله مثل .data لكن الفرق أنه مخصص للـ uninitialized data أي البيانات التي ليس لها قيمة ابتدائية ، فكما تلاحظ فان المتغير hinstance بجانبه علامة ؟ دلالة على أنه لا يحمل قيمة ابتدائية.

ان دالة ***GetModuleHandle*** تقوم بإرجاع مقبض (handle) للـ module المحددة ... تعريف الدالة (بالنسبة للـ c++) هو:

HMODULE WINAPI GetModuleHandle(LPCTSTR lpModuleName);

أي أن البارامتر لهذه الدالة هو مؤشر(pointer) على string بها اسم الـ module الذي نريد مقبض له . وبالتالي إن أردنا مقبضا للـ kernel32.dll مثلا فإننا نخزن هذه الـ string (أي kernel32.dll) في مكان ما بالبرنامج ثم نعرف مؤشرا عليها...لكن كما تلاحظ في حالتنا فإن البارامتر كان صفرا. في هذه الحالة فإن الدالة تقوم بإرجاع مقبض للملف الذي استخدم في إنشاء الدالة...أي الملف الذي توجد دالة GetModuleHandle فيه. جدير بالذكر أن المقبض المرجع يخزن في eax .

أما الدالة الثانية ***DialogBoxParam*** فتعريفها (بالنسبة لـ c++) هو كالتالي :

```
int DialogBoxParam(

HINSTANCE hInstance,

LPCTSTR lpTemplateName,

HWND hWndParent,

DLGPROC lpDialogFunc ,

PARAM dwInitParam);
```

البارامتر الأول هو مقبض للـ module التي يحوي ملفها التنفيذي على الـ dialog box .
الثاني هو مؤشر إلى الـ dialog box . الثالث مقبض للنافذة الأم . أي النافذة الأصلية التي تفرع منها هذا الـ dialog
الرابع مؤشر إلى إجرائية(procedure) الـ dialog box . الخامس يحدد القيمة التي ستمرر الـ dialog box (حيث سيستقبلها عن طريق بارامتر *lParam* من المسجـ(الرسالة) المسماة WM_INITDIALOG .

الدالة الثالثة سبق شرحها وهي لإنهاء البرنامج. الآن دعنا نرى محتويات تلك الـ procedure :

```
mydialog PROC hwnd:HWND , uMsg:UINT,wParam:WPARAM, lParam:LPARAM
    .if uMsg==WM_COMMAND
        .if wParam==1002
            invoke SetDlgItemText,hwnd,1003,addr msg1
        .elseif wParam==1004
            invoke SetDlgItemText,hwnd,1003,addr msg2
        .endif
    .elseif uMsg==WM_CLOSE
        invoke EndDialog,hwnd,0
    .endif
xor eax,eax
Ret
mydialog EndP

end start
```

αλλκο

أول سطر هو تعريف الإجرائية. إن تعريف الإجرائية هو كالتالي :

```
INT_PTR CALLBACK DialogProc(

   HWND hwndDlg,

   UINT uMsg,

   WPARAM wParam,

   LPARAM lParam );
```

إن هذه العناصر الأربعة هي messages . فالـ dialog يتعامل مع النظام عن طريق رسائل.
البارامتر الأول (أو الرسالة لأولى) هي مقبض للـ dialog Box والبارامتر الثاني يحدد الرسالة الواصلة الى الـ dialog أما البارامتر الثالث فيحدد معلومات إضافية عن الرسالة الواصلة الى الـ dialog ، بالمثل البارامتر الرابع يحدد معلومات إضافية عن الرسالة الواصلة . عندما نقول رسالة واصلة فنحن نعني بذلك أن يضغط المستخدم على زر ما –مثلا- أو أن يغلق البرنامج من زر x بالأعلى...

والآن لنرى تعريف دالة SetDlgItemText كما هو موجود في مكتبة MSDN :

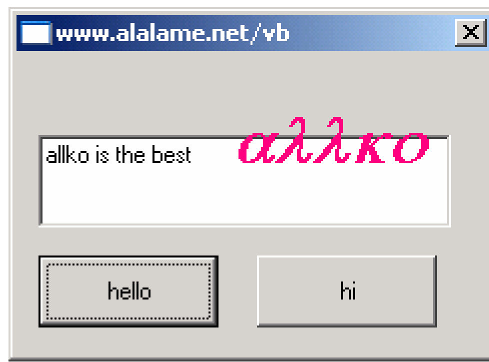
```
BOOL SetDlgItemText( HWND hDlg, int nIDDlgItem, LPCTSTR lpString);
```

أول بارامتر هو مقبض للـ dialog box . ثاني بارامتر يحدد العنصر الذي سيعرض النص . الثالث هو مؤشر إلى الـ string التي سوف يتم عرضها.

جدير بالذكر أنه هنالك خياران فيما يتعلق بتحديد العنصر –سواء بالنسبة للدالة هذه أو لبقية الدوال – فإما أن تكتب اسم العنصر ، أي DC_BUTTON1002 مثلا ، أو أن تكتب الـ ID الخاص به...بالتأكيد كتابة الـ ID أسهل بكثير وهذا ما فعلناه.

أول سطر في تلك الإجرائية هو if uMsg==WM_COMMAND. وهذا يعني أنه اذا كانت الرسالة الواصلة من المستخدم هي أمر فنفذ الأوامر التالية. ضمن هذه الـ if هناك 2 if *الأولى* تقول أنه اذا كانت المعلومات التابعة للرسالة الواصلة ، صادرة عن العنصر ذي الرقم 1002 <الزر الأيمن> فاستدعي دالة SetDlgItemText واطلب منها أن تعرض msg1 في العنصر ذي الرقم 1003 (أي في مربع التحرير edit) . *الثانية* تقول أنه اذا كانت المعلومات التابعة للرسالة الواصلة ، صادرة عن العنصر ذي الرقم 1004<الزر الأيسر> فاستدعي دالة SetDlgItemText واطلب منها أن تعرض msg 2 في العنصر ذي الرقم 1003 (أي في مربع التحرير edit) . بعد هاذين الشرطين هناك endif التي تنهي عمل if .
بعد ذلك هناك elseif uMsg==WM_CLOSE. وهذه تعني أنه اذا كانت الرسالة الصادر من المستخدم هي طلب إغلاق فنفذ التالي : استدعي الدالة EndDialog التي سوف تقوم بإغلاق الـ dialog box . وبعد ذلك هناك endif التي تنهي عمل if .
بعدها تأتي تعليمة بتصفير المسجل eax . xor eax,eax وهذه الخطوة احتياطية .ثم هناك تعليمة RET التي تعيد السيطرة الى النظام.
وفي النهاية نجد mydialog end معلنة إنتهاء الاجرائية .

الآن اضغط على زر Go All في الأعلى ...أحفظ الملفات في مكان مناسب (يفضل نفس مجلد winasm) والآن يفترض أن ترى التالي :
(هذه الصورة مأخوذة بعد الضغط على الزر الأيمن) :



إن ظهر لك خطأ فراجع الخطوات جيدا...في المرفقات هناك ملفان الأول باسم asm والثاني باسم resource أنشئ مشروعا جديدا وانسخ محتويات ملف asm الى قسم السورس كود ثم من قائمة make اختر new rc والآن انتقل الى الوضع الكتابي بالضغط على زر الذي أسفل قائمة make وانسخ محتويات ملف resource اليه. اضغط على زر go all واحفظ الملفات في نفس مجلد winasm . الأن سترى النتيجة بلا أخطا...حاول ان تجد خطأك بمقارنة ما كتبته مع ما هو مرفق...

وبهذا يكون درسنا لهذا اليوم قد انتهى آمل أن تكونوا قد استفدتم .

الفريق العربي للهندسة العكسية

دورة ATRE لتعليم الأسمبلي للمبتدئين من الصفر

الدرس التاسع- keygening III

allko

في الدرسين الماضيين (السابع والثامن) بدأنا بتعلم كيفية استخدام winasm في كتابة برامج اسمبلي مرئية...اليوم سنتعلم كيفية صناعة كيجين...وإضافة موسيقى له أيضا...الشرح سيكون مخصصا لـ crackme الذي وضعه الأخ ColdFever كتحدي في منتدى البرمجة والحماية / نادي التحدي .
لن أخوض كثيرا في تفاصيل كسر الحماية..فالتحدي بسيط جدا...تجده بالمرفقات باسم coldfever

افتح البرنامج مستخدما olly . كما ترى هناك دالة GetDlgItemTextA عند 401172. ضع bp عليها ثم F9 . أدخل أي اسم وليكن allko وأي رقم وليكن 123456 . الآن اضغط register . ها قد عدنا الى olly . تتبع(F8) وعندما تصل الى الـ CALL عند العنوان 4011B6 فاضغط F7 كي نرى ماذا يوجد هناك.

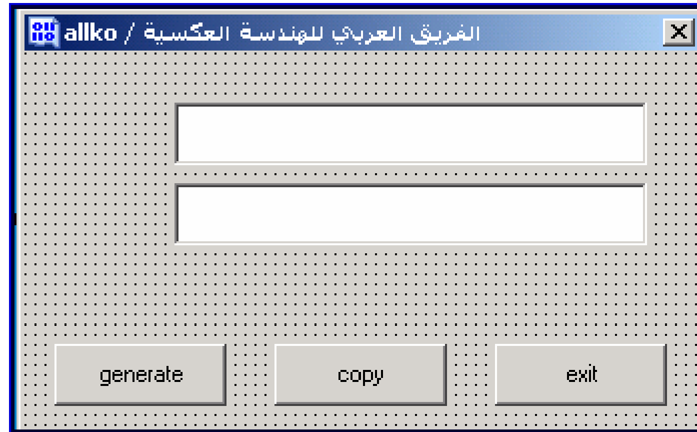
```
PUSH keygenMe.0040361A                    [String2 = "FIT-"
PUSH keygenMe.00403256                    [String1 = keygenMe
CALL <JMP.&kernel32.lstrcpyA>             [lstrcpyA
PUSH keygenMe.0040339A                    [String = "allko"
CALL <JMP.&kernel32.lstrlenA>             [lstrlenA
MOV DWORD PTR DS:[403396],EAX
CMP EAX,5
JL SHORT keygenMe.004013F1
LEA EBP,DWORD PTR DS:[403638]
LEA EDI,DWORD PTR DS:[403396]
PUSH EDX
PUSH EBX
PUSH EDI
PUSH ECX
PUSH EAX
MOV EDX,4E6AF4BC                          allko
LEA EBX,DWORD PTR SS:[EBP-4]
MOV EDI,DWORD PTR SS:[EBX]
MOV ECX,DWORD PTR DS:[EDI-4]
SUB ECX,3
MOV EAX,DWORD PTR DS:[EDI]
XOR EDX,EAX
INC EDI
DEC ECX
JNZ SHORT keygenMe.004013AC
MOV DWORD PTR SS:[EBP-8],EDX
POP EAX
POP ECX
POP EDI
POP EBX
POP EDX
MOV EAX,DWORD PTR SS:[EBP-8]
PUSH EAX                                  [<%d>
PUSH keygenMe.0040361F                    [Format = "%d"
PUSH keygenMe.004034DA                    [s = keygenMe.00403
CALL <JMP.&user32.wsprintfA>              [wsprintfA
ADD ESP,0C
PUSH keygenMe.004034DA                    [StringToAdd = ""
PUSH keygenMe.00403256                    [ConcatString = ""
CALL <JMP.&kernel32.lstrcatA>             [lstrcatA
PUSH keygenMe.00403256                    [String2 = ""
PUSH keygenMe.00403116                    [String1 = "123456"
CALL <JMP.&kernel32.lstrcmpA>             [lstrcmpA
RETN
```

الشرح
بداية هناك دالة lstrcatA التي ستضيف FIT- إلى السيريال (أي إلى الذاكرة عند 403256) . يليها دالة حساب الطول lstrlenA (يخزن الطول في eax ثم ينقل إلى 403396)
هنا يتم مقارنة الطول مع القيمة 5
تحميل القيمة 403638 إلى EBP و القيمة 403396 إلى EDI .
هنا يتم حفظ قيم المسجلات الخمسة: EDX , EBX , ECX , EDI , EAX
تحميل القيمة4E6AF4BC(قيمة عشوائيةمن اختيار المبرمج)
تحميل بعض القيم والعناوين.تتبع بنفسك لمزيد من التفاصيل
يتم طرح 3 من طول اليوزر(allko=5). الآن ECX قيمته 2 .
يتم تحميل أول حرف إلى eax . ثم يتم عمل عملية xor له مع القيمة 4E6AF4BC . ثم يتم زيادة EDI ليشير إلى الحرف التالي ثم إنقاص ecx .عندما يصبح ecx=0 تنتهي الـ loop.
يتم نقل النتيجة من EDX الى الذاكرة...ثم يتم إرجاع المسجلات الأربعة لقيمها السابقة قبل تنفيذ الـ loop .
يتم نقل النتيجة الى EAX
يتم دفع البارامترات الثلاثة للدالة wsprintfA. تقوم بتحويل النتيجة التي حصلنا عليها الى أحرف و أرقام (أي الى رموز ASCII) سنتحدث عنها لاحقا...
الآن يتم إضافة الـ string السابقة الى FIT- لنحصل على الصيغة النهائية للسيريال.
هنا تتم عملية مقارنة بي السيريال المدخل والسيريال الصحيح.

ان أول حرف من اليوزر مع القيمة 4E6AF4BC . هذه الـ loop تستمر حتى عدد معين من المرات يساوي طول اليوزر ناقص 3 . فإذا كان طول اليوزر 5 ، فان هذه الـ loop تستمر دورتين اثنتين ، أي يتم عمل عملية الـ xor للحرف الأول والثاني من اليوزر. ثم يتم عمل عملية أشبه بالتشفير تحول النتيجة السابقة من مجرد قيم هكس إلى رموز بصيغة الـ ascii . في النهاية يتم إضافة هذه الرموز إلى FIT- ونحصل على السيريال. الآن دعنا نلخص العملية. بداية هناك 4 أحرف في أول كل سيريال بغض النظر عن اليوزرنيم المدخل وهي FIT- .

الآن شغل برنامج winasm . أنشئ مشروعا جديدا . والآن من قائمة project اختر new rc .
قم بإنشاء dialog جديد (أول زر من اليسار في شريط explorer) . الآن أنشئ مربعي تحرير وثلاث أزرار كما بالصورة

التالية :



انتبه الى أن الـ ID لكل resource هي كالتالي (غيرها إن لم تكن لديك هكذا)
الـ dialog الرئيسي : 1001
مربع التحرير العلوي :1002
مربع التحرير السفلي :1003
زر generate :1004
زر copy :1005
زر exit :1006

الآن فلنذهب الى قسم السورس كود لنكتب كود الكيجين :
كما تعلم فأول أسطر هي :

```
.386
.model flat,std call
option casemap:none

include windows.inc
include kernel32.inc
include user32.inc
includelib kernel32.lib
includelib user32.lib
```

يليها تعريف الـ procedures الاثنتان...واحدة للـ dialog وأخرى للـ generating

```
mydialog   proto :DWORD,:DWORD,:DWORD,:DWORD
generating proto
```

لاحظ أن إجرائية توليد السيريال لا تستقبل شيء...بينما الأولى تستقبل 4messages . والآن في مقطع البيانات لدينا التالي:

```
.data
special1          db          "FIT-",0
special2          db          "%d",0
```

الـ string الأولى لإضافة تلك المحارف الأربعة للسيريال أما الثانية فخاصة بالدالة wsprintfA وسنتحدث عنها لاحقا. نأتي الآن الى مقطع .data? ولدينا الآتي :

```
.data?
hInstance          HINSTANCE ?
NameBuffer         32 dup(?)
SerialBuffer        32 dup(?)
mystr             32 dup(?)
```

الأولى هي متغير سنستخدمه لنضع فيه مقبض (handle) . هذه "الصيغة" احفظها كما هي . إن أردت المزيد من التفاصيل فابحث على الانترنت...
الثانية تحجز مساحة فارغة بحجم 32 بايت خاصة بالمتغير NameBuffer . لو كتبت مثلا ('d')32 dup فسيتم حجز مساحة بحجم 32 بايت بها 32 حرف d . طبعا هذا ما لا نريده لذلك نكتب علامة الاستفهام (لاحظ أنها بدون ' ') التي تدل على أن المساحة فارغة ليس بها شيء . أيضا ربما تسأل : أليس من المفروض أن نكتب تلك الجملة هكذا :

```
NameBuffer   db      32 dup(?)
```

لا مشكلة...لأننا نتعامل مع uninitialized data فيمكنك الاستغناء عن تلك الـ directive (أقصد db) . لكن في قسم data. حيث لا يوجد سوى initialized data فإياك أن لا تكتبها.
نفس الشيء ينطبق على المتغير الثالث SerialBuffer . أما الرابع فسنستخدمه كمتغير مؤقت...سيلزمنا عند استخدام دالة wsprintfA . يلي ذلك الأسطر التالية :

```
.const
IDD_KEYGEN      equ     1001
IDC_NAME        equ     1002
IDC_SERIAL      equ     1003
IDC_GENERATE    equ     1004
IDC_COPY        equ     1005
IDC_EXIT        equ     1006
```

هذا هو مقطع الثوابت (constants) هنا كل ما نفعله هو أن نخبر المجمع بأن زر الكيجين له الـ ID 1001 و مربع حوار السيريال له الـ ID 1003 وهكذا مع بقية الـ resources . إن الـ directive المسماة equ تستخدم لإعطاء متغير ما قيمة ثابتة طوال عمل البرنامج. أي محاولة لتغيير هذه القيمة ستعطيك error. الآن لدينا التالي :

```
.code
start :
   invoke GetModuleHandle, NULL
   mov    hInstance,eax
   invoke DialogBoxParam, hInstance, IDD_KEYGEN, NULL, addr DlgProc, NULL
   invoke ExitProcess,eax
```

أول سطر يعني أننا بدئنا بمقطع الكود. ثاني سطر يعني بداية الكود التنفيذي . يلي ذلك استدعاء لدالة GetModuleHandle باختصار ستعيد مقبض للملف الموجودة فيه أي kernel32.dll . الثالث سيستدعي الإجرائية الخاصة بالـ dialog . البارامترات الأربعة : الأول مقبض ملف kernel32.dll ، الثاني اسم الـ dialog ويمكنك استبداله بالـ ID الخاص به (أي 1001) ، الثالث مقبض للنافذة الأم (صفر لأنه لا يوجد نافذة أم) ، الرابع مؤشر إلى إجرائية (procedure) الـ dialog box ، الخامس يحدد القيمة التي ستمرر الـ dialog box (حيث سيستقبلها عن طريق بارامتر *lParam* من المسج(الرسالة) المسماة WM_INITDIALOG . الدالة الثالثة خاصة بإنهاء البرنامج.

والآن سنكتب إجرائية الـ dialog .

```
DlgProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
.if uMsg == WM_INITDIALOG

.elseif uMsg == WM_COMMAND
    mov    eax,wParam
    .if eax==IDC_GENERATE
            invoke GetDlgItemText,hWnd,IDC_NAME,addr NameBuffer,32
            call Generate
            invoke SetDlgItemText,hWnd,IDC_SERIAL,addr SerialBuffer
    .elseif eax==IDC_COPY
            invoke SendDlgItemMessage,hWnd,IDC_SERIAL,EM_SETSEL,0,-1
            invoke SendDlgItemMessage,hWnd,IDC_SERIAL,WM_COPY,0,0
    .elseif eax==IDC_EXIT
            invoke SendMessage,hWnd,WM_CLOSE,0,0
    .endif
```

```
.elseif     uMsg == WM_CLOSE
    invoke  EndDialog,hWnd,0
.endif
    xor     eax,eax
    ret
DlgProc endp
```

لن أشرح مرة أخرى فقد سبق وشرحت كودا مشابها في الدرس السابق.بشكل عام هناك ثلاث messages (باللون الأزرق) أساسية . الأولى وكما ترى فارغة ، وهي خاصة بعملية الـ initialization (البدء) . يمكنك هنا أن تضع الكود الخاص باستخدام icon للكيجين مثلا...أو الكود الخاص بتشغيل موسيقى (كما سنرى لاحقا) . مؤقتا اتركه فارغا.

ربما التغيير الجديد هو في الكود الخاص بزر copy .

```
.elseif eax==IDC_COPY
        invoke SendDlgItemMessage,hWnd,IDC_SERIAL,EM_SETSEL,0,-1
        invoke SendDlgItemMessage,hWnd,IDC_SERIAL,WM_COPY,0,0
```

حسنا لا داعي لإطالة الوقت في شرح تلك الدوال فأنا نفسي لم أذهب الى msdn كي أفهمهم. يكفي أن تحفظهم كما هم وإن أردت التفاصيل فعليك بـ msdn . فقط انتبه الى البارامترات...البارامتر الثاني (في الدالتين) هو المكان الذي تريد أن ننسخ منه وهو هنا IDC_SERIAL .

أما زر generate فتجده أسفله الكود :

```
.if eax==IDC_GENERATE
        invoke GetDlgItemText,hWnd,IDC_NAME,addr NameBuffer,32
        call Generate
        invoke SetDlgItemText,hWnd,IDC_SERIAL,addr SerialBuffer
```

أي ان الضغط على ذلك الزر يعني : 1- استدعاء دالة GetDlgItemText التي ستأخذ اليوزرنيم(IDC_NAME) وتخزنه في NameBuffer . بالطبع فهي لن تأخذ الا الـ 32 حرفا الأولى كما هو واضح من البارامتر الرابع. ثم يتم استدعاء إجرائية توليد السيريال ، وفي النهاية يتم استدعاء دالة SetDlgItemText التي تعرض السيريال في المكان المخصص.

والآن لنرى الكود الخاص بإجرائية توليد السيريال (قمت بتلوين الكود لتسهيل عملية التتبع) :

1	*Generate proc*
2	mov edi,offset NameBuffer
3	invoke lstrlen, edi
4	cmp eax,5
5	jl **NOINPUT**
6	invoke lstrcat,addr SerialBuffer,addr special1
7	xor eax,eax
8	xor ecx,ecx
9	invoke lstrlen, edi
10	mov ecx, eax
11	sub ecx,3
12	mov edx,4E6AF4BCh
13	xor eax,eax
14	xor ebx,ebx
15	**begin:**
16	mov eax,dword ptr [ebx+edi]
17	inc ebx
18	xor edx,eax
19	dec ecx
20	jnz **begin**
21	invoke wsprintf,addr str3,addr special2,edx
22	invoke lstrcat,addr SerialBuffer,addr str3
23	**NOINPUT:**

24	ret
25	Generate endp

السطر 1 و 25 يوضحان أين تبدأ تلك الإجرائية وأين تنتهي...
السطر 2 يقوم بأخذ عنوان اليوزرنيم ويضعه في edi (غالبا يستخدم edi لمثل هذه المهام) .
السطر 3 يستدعي دالة lstrlen ويدفع لها مؤشرا الى السترنج الخاصة باليوزرنيم . هذه الدالة تحسب طول السترنج التي نعطيها مؤشرا عليه (نحن أعطيناها مؤشرا على اليوزرنيم) وتقوم بإعادة الطول في EAX .
السطر 4 يفحص ما إذا كان السيريال المدخل طوله اقل من 5 أم لا...إذا كان صفرا (أي لم يتم إدخال أي شيء) أو كان اقل من 5 فان السطر 5 به تعليمة jl (أي jump if low) فسيتم الذهاب الى NOINPUT (هل عرفت الآن ماذا قصدت في الدرس السابق بقولي : label ؟ إن NOINPUT و begin هما مثالان على ذلك) وبعد الوصول الى هناك سيتم تنفيذ الكود الذي هناك..ببساطة ما يوجد هناك هو تعليمة ret التي تعيد السيطرة الى البرنامج و تنهي هذه الإجرائية .
السطر 6 يستدعي دالة lstrcatA التي ستضيف -FIT الى بداية السيريال .
السطر 7 و 8 يقومان بتصفير eax و ecx تمهيدا لاستخدامها.
السطر 9 يستدعي دالة lstrlen مرة أخرى لوضع الطول في eax . (((ملاحظة : كان بالإمكان عندما حسبنا الطول للمرة الأولى أن نضعه في متغير ما...ونستخدمه كلما احتجناه في البرنامج...))) .
السطر 10 ينقل الطول من eax الى ecx .
السطر 11 يطرح 3 من الطول .(لماذا 3 ؟ لا علاقة لي !!! هكذا يريد المبرمج اسأله !! إنها مجرد قيمة اختيارية :)) . السطر 12 يضع القيمة 4E6AF4BCh في edx تمهيدا لاستخدامه في الـ loop .
السطر 13 و 14 يصفران eax و ebx تمهيدا لاستخدامهما في الـ loop .
السطر 15 هو label سنتحدث عنه بعد قليل.
السطر 16 هو بداية الـ loop . كما ترى يتم نقل محتويات الذاكرة التي عنوانها يساوي ebx+edi الى eax ، 4 بايت ف المرة الواحدة (أي DWORD) . لاحظ أنه كلما رأيت تلك الصيغة فإن ما يوجد بين القوسين يشير الى عنوان ذاكرة قد يكون مثلا هكذا [ecx+8] أو هكذا [3* edi + ebx] أو أي صيغة أخرى...أما DWORD PTR فكلمة DWORD هي اختصار double word وكل word تساوي 2 بايت ، إذن DWORD تساوي 4 بايت. انتبه فلا يمكنك أن تكتب التالي:

mov eax,word ptr [ebx+edi]

لأن eax حجمه 32 بت أي 4 بايت أي dword وبالتالي من الخطأ أن تنقل إليه word . أيضا العكس خطأ فالصيغة

mov ax,dword ptr [ebx+edi]

ستتسبب في خطأ فليس من المنطق نقل 32 بت الى مسجل بحجم 16 بت فقط. يجب أن يكون الـ source والـ destination متساويين بالحجم...طبعا هذا يخص تعليمة mov لا أعرف إذا ما كان هناك تعليمات تخرج عن هذه القاعدة.
جدير بالذكر أنني اخترت ebx كعداد لليوزرنيم ...اخترت edx كمكان لوضع القيمة 4E6AF4BCh ، اخترت edi كمؤشر الى اليوزرنيم...لماذا؟ يمكنك نظريا اختيار ما تريد...لكن هناك بعض الضوابط...حاول الابتعاد عن eax فكما ترى نحن نستخدمه مع الدوال فمعظم الدوال تعيد قيمها فيه...أما ecx فهو من اسمه extended counter أي اعتدنا استخدامه كعداد. edi و ebi يستخدمان غالبا كمؤشر وبالتالي لا تضع فيهم قيما ثابتة...ضع فيهم عناوين ذاكرة فهذا هو السبب الأساسي الذي دفع Intel لصناعتهما :) .
السطر 17 يقوم بزيادة ebx بمقدار واحد. لاحظ أنه بداية كان edi يشير الى أول حرف (راجع السطر الثاني) و ebx قيمته صفر إذن المجموع سيشير الى أول حرف...لكن في الدورة الثاني من الـ Loop نريد أن نتعامل مع ثاني حرف وبالتالي يجب زيادة ebx بمقدار واحد فيصبح المجموع يشير الى الحرف الثاني وهكذا.
السطر 18 هو أساس الـ loop وهو الذي سيولد السيريال...ببساطة يتم عمل xor لكل حرف مع القيمة المخزنة في edx وهي 4E6AF4BCh (حرف h ليس جزئا من القيمة فكما تعلم فإن النظام السداسي العشري ينتهي عند حرف F ، لكنها ضرورية لأنه بدونها سيعتبر المجمع أن هذه القيمة هي بنظام decimal وليس hex). لاحظ أن القيمة النهائية تخزن في edx .
السطر 19 ينقص ecx بمقدار واحد .
السطر 20 فيه قفزة (Jump if NOT Zero) . لاحظ أن ecx يتناقص تدريجيا...فورما تصبح قيمته صفر فآن مسجل الأعلام المسمى zero flag يتحول الى الوضع set أي تصبح قيمته واحد دلالة على أن مسجلا من المسجلات قد تم تصفيره...ما تفعله تلك القفزة هو النظر الى مسجل الأعلام هذا (البعض يقول الـ "راية" بدل مسجل الأعلام) هل هو set ؟ أي هل قيمته 1؟ اذا كان نعم فانها لن تقفز...بل ستتابع البرنامج وكأنها غير موجودة...أما اذا كان لا أي ان مسجل الأعلام zero flag في وضع reset أي قيمته 0 ، فإنها ستقفز...الى اين؟ الى الـ label المسمى **begin** . وبالتالي ستعيد الـ loop مرة أخرى...ستستمر هذه الدورة الى ان ينتهي العد ويصل ecx الى الصفر...
السطر 21 يستدعي دالة wsprintfA ...ما تقوم به هذه الدالة هو أشبه بعملية فك التشفير إن صح التعبير...نعطيها قيمة معينة...و نعطيها Format Control (يمكن أن تقول : معامل التشفير :)) ، هذا الـ Format Control هو %d (لا تسألني لماذا ، تتبع البرنامج وستجد المبرمج قد اختاره...)

أول بارامتر لها هو المكان الذي سنخزن النتيجة فيه...وهو str3 (هل يجب أن أقول : يمكنك اختيار أي اسم آخر؟):)
ثاني بارامتر هو الـ Format Contro والثالث هو القيمة التي نود "فك تشفيرها" ان صح التعبير وهي هنا edx لانه كما قلنا يحوي نتيجة تلك الـ Loop .
السطر 22 يضيف ما حصلنا عليه للتو وخزناه في str3 الى السيريال (المبدوء بـ -FIT).
السطر 23 و 24 تم شرحهم.
السطر 25 هو نهاية الـ procedure .

و الآن يجب أن ننهي البرنامج...ببساطة نحتاج الى :

```
end start
```

اضغط زر Go All .احفظ ملفات البرنامج بأي اسم (ويفضل في نفس مجلد البرنامج) ...والآن جرب واستمتع !!

إضافة الموسيقى :

هناك طريقتان لذلك...جربت الطريقتين ونجحتا لكن مع ملفات xm فقط وليس مع ملفات mod ولا أعرف السبب.
سأشرح طريقة واحدة وهي الأسهل ولا تحتاج الى كتابة أكواد كثيرة...ربما 5 أو 6 أسطر فقط !!!
ضمن الملفات المستوردة (أسفل include windows.inc) أضف التالي :

```
include         ufmodapi.inc
includelib      ufmod.lib
```

الآن ضمن مقطع code. أضف التالي :

```
include         allko.inc
xmSize          equ $ - table
```

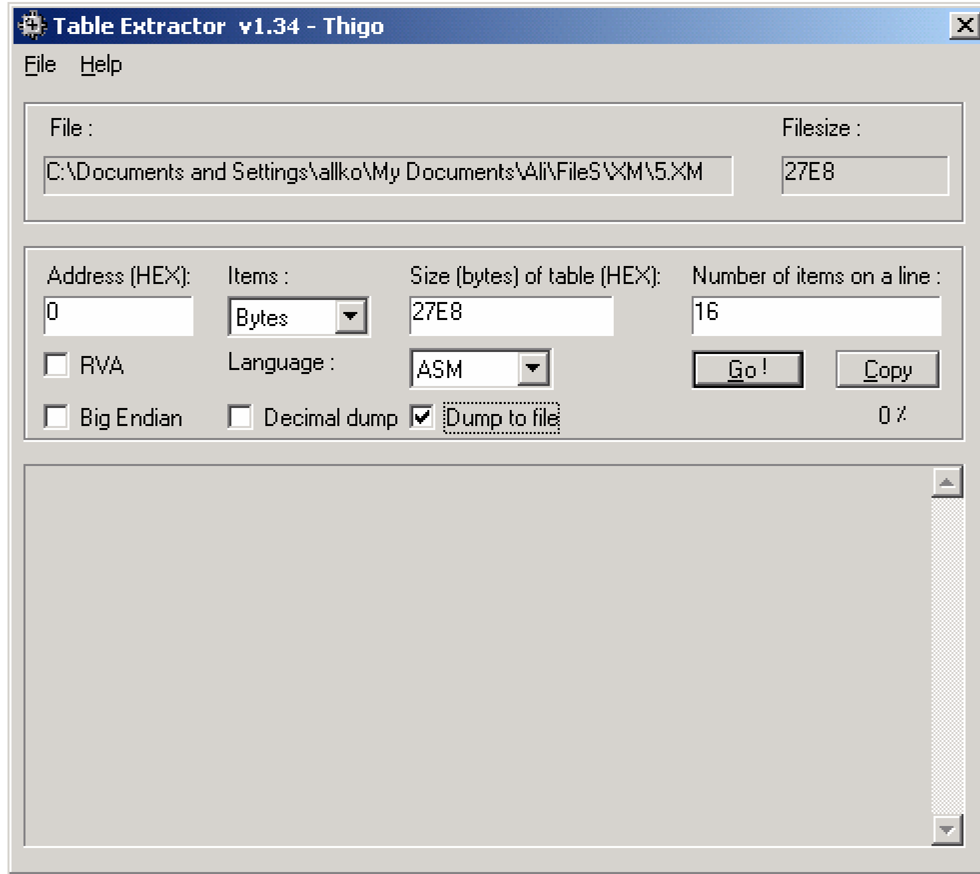
تذكر أننا تركنا مكان الكود الخاص برسالة WM_INITDIALOG فارغا...حان الوقت لكتابة شيء هنا...السطر التالي لا داعي لمعرفة تفاصيله ، فقط أضفه ضمن هذه الرسالة كلما أحببت إضافة موسيقى الى برنامجك.

```
invoke    uFMOD_PlaySong,addr table,xmSize,XM_MEMORY
```

والآن من الطبيعي أنه في المكان الخاص برسالة WM_CLOSE أن نضع كودا لإنهاء الموسيقى...هذا ما يفعله السطر التالي :

```
invoke    uFMOD_PlaySong,0,0,0
```

حسنا ، سأفترض أنك اتبعت ما قلته لك وحفظت المشروع ضمن مجلد winasm . هناك بعض الملفات والمكتبات التي يجب وضعها في نفس المجلد الذي حفظت المشروع فيه. وهي : ufmod.lib و ufmodapi.inc و chiptune.inc
الأول والثاني لن تغير فيهم شيئا . أما الثالث فهو ملف الموسيقى !! كيف؟ طريقتنا تعتمد على اضافة ملف الموسيقى ليس كـ resource (هذا ما تتبعه الطريقة الثانية) بل كـ array of bytes . عظيم لكن كيف سنحول ملف الموسيقى الى array of bytes ؟ أرفقت مع الدرس برنامجا خاصا بذلك(! thx THigo). افتح البرنامج ثم من قائمة file اختر load واختر أي ملف موسيقي ترغب فيه.
الآن بعد تحميل الملف غير الإعدادات الى ما تراه هنا :



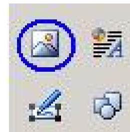
لاحظ أن الملف الذي قمت أنا باختياره حجمه 27E8 لذلك في خانة size كتبت نفس الرقم. أما القيمة 0 و 16 فلن تتغيرا . انتبه الى اختيار اللغة : asm ، أيضا لا تنسى وضع علامة صح على خيار dump to file ، والآن اختر go .
بعد الانتهاء ستحصل على ملف باسم table.txt افتحه ثم احفظه باسم allko.inc (لأننا اخترنا هذا الاسم بالأعلى)
والآن اضغط GO all . استمتع بالموسيقى !!!

الفريق العربي للهندسة العكسية

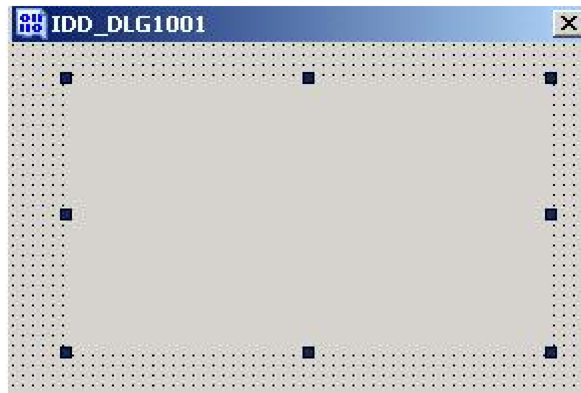
درس اليوم هو الدرس الأخير من دورة ATRE لتعليم الأسمبلي للكراكرز...هذا الدرس سيتناول ثلاث نقاط :

- كيفية إضافة صور من نوع bmp ، وهذه تتم مباشرة وبكل سهولة
- كيفية إضافة صور من نوع jpg ، وهذه تحتاج الى أكواد معينة ودوال خاصة لتنفيذها...
- كيفية إضافة تأثير Fade In و Fade Out .

كيفية اضافة صورة هي عملية بسيطة للغاية...سأفترض أنك قمت بفتح مشروع جديد وقمت بإنشاء dialog جديد... والآن من شريط الـ tool box (ان لم يكن ظاهرا فقم باظهاره من view ➔ toolbox) اختر زر الصورة

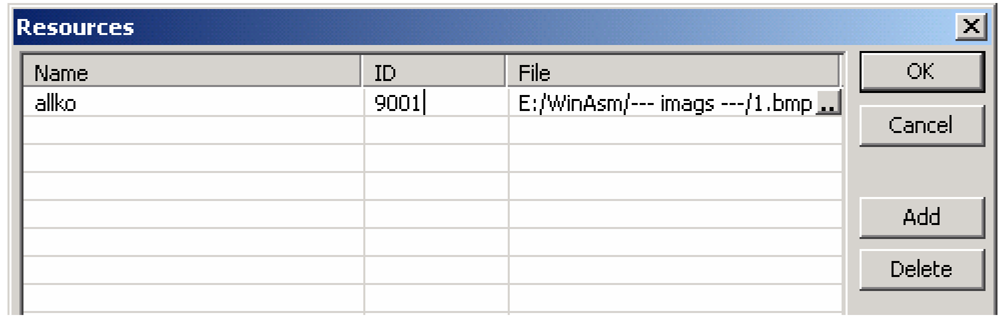


والان حرك المؤشر فوق الـ dialog وستلاحظ أن المؤشر تغير بحيث يمكنك رسم حدود الصورة...ارسم حدودا كما تريد...هكذا مثلا:



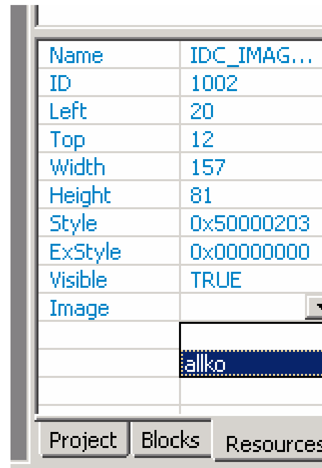
والآن اضغط على زر الـ resources :

سيخرج لك مربع حوار جديد ، اضغط add وحدد صورة ما ثم قم بكتابة اسم و ID لها ...هكذا مثلا :

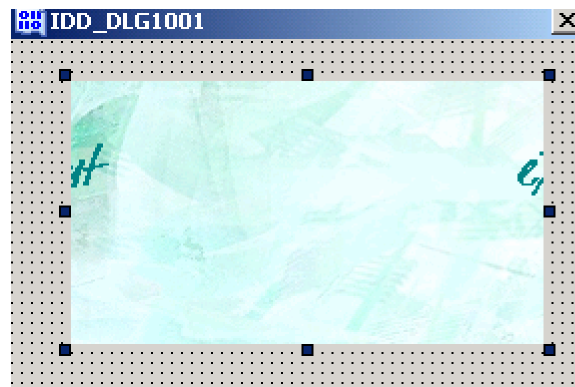


ثم اضغط ok . والآن اضغط على ما تراه بالصورة :

ثم :



والآن ستلاحظ أن الصورة قد ظهرت :



قد تتساءل...ما الدافع الى تعلم كيفية اضافة صور الـ jpg ما دمنا قادرين على اضافة صور الـ bmp وبكل بساطة ؟ لم نقوم باضافة أكواد اضافية ؟

حسنا هناك سبب بسيط...صور jpg مضغوطة لذا فان حجمها يكون اصغر بكثير من صور bmp ، مثلا احدى الصور لدي كان حجمها (bmp) : 197kb ، لكن نفس الصورة بصيغة jpg : 23kb .

هناك مشكلة بسيطة في هذه الحالة – أي عند استخدام صور jpg – وهي ان winasm لن يكون قادرا على تشغيل الوضع المرئي visual وبالتالي لن تستطيع اضافة زر او مربع نص مثلا الا بطريقة يدوية...عموما هذه ليست بمشكلة كبيرة وهناك حل سأذكره لاحقا في هذا الدرس.

والآن قم بإنشاء مشروع جديد...standard exe ➔ new ➔ file ثم add new RC ➔ Project .
في المرفقات هناك ملفان نصيان ، الأول test والثاني testRC ، قم بنسخ محتويات الأول الى المكان المخصص له ، ومحتويات الثاني الى المكان المخصص له .

والآن اضغط زر Go All لترى التالي في الزاوية العليا اليسرى :
(طبعا قبل أن تره هذه الشاشة سيطلب منك winasm ان تقوم بحفظ المشروع ، لا تنسى ان تضع الصورة المسماة image في نفس المجلد الذي ستحفظ فيه ملفات المشروع .)

لنرى ما الجديد...

```
386.
.model flat,stdcall
option casemap:none

    include \masm32\include\windows.inc
    include \masm32\include\masm32.inc
    include \masm32\include\gdi32.inc
    include \masm32\include\user32.inc
    include \masm32\include\kernel32.inc
    include \masm32\include\Comctl32.inc
    include \masm32\include\comdlg32.inc
    include \masm32\include\shell32.inc
    include \masm32\include\oleaut32.inc
    include \masm32\include\ole32.inc

    includelib \masm32\lib\masm32.lib
    includelib \masm32\lib\gdi32.lib
    includelib \masm32\lib\user32.lib
    includelib \masm32\lib\kernel32.lib
    includelib \masm32\lib\Comctl32.lib
    includelib \masm32\lib\comdlg32.lib
    includelib \masm32\lib\shell32.lib
    includelib \masm32\lib\oleaut32.lib
    includelib \masm32\lib\ole32.lib
```

حسنا هناك الكثير من الـ inc. والـ lib. هنا...لا بأس دعك منها...فلنرى ماذا يوجد بعد :

```
dlgproc proto :DWORD ,:DWORD ,:DWORD ,:DWORD

.data
msg   db  "hi",0

.data?
hInstance  HINSTANCE?
hBmp dd?

.code
start:
        invoke GetModuleHandle,NULL
        mov hInstance,eax
        invoke DialogBoxParam,hInstance,1001 ,NULL,addr dlgproc ,NULL
        invoke ExitProcess,NULL
```

حتى الآن لا جديد...هذه الأسطر نكتبها دائما...دعنا نرى الأسطر التالية :

```
dlgproc proc hWnd:HWND ,uMsg:UINT ,wParam:WPARAM, lParam:LPARAM
 LOCAL hDC :DWORD,  LOCAL hOld :DWORD,  LOCAL memDC :DWORD,
 LOCAL ps:PAINTSTRUCT
.if uMsg == WM_COMMAND
        .if wParam==1002
                invoke MessageBox,hWnd,addr msg,addr msg,MB_OK
        .endif
.elseif uMsg==WM_INITDIALOG
        invoke BitmapFromResource, hInstance, 1003
        mov hBmp, eax
        invoke BitBlt,hDC,0,1,550,175,memDC,0,0,SRCCOPY
.elseif uMsg == WM_PAINT
        invoke BeginPaint,hWnd,ADDR ps
        mov hDC, eax
        invoke CreateCompatibleDC,hDC
        mov memDC, eax
        invoke SelectObject,memDC,hBmp
        mov hOld, eax
        invoke BitBlt,hDC,0,1,550,175,memDC,0,0,SRCCOPY
        invoke SelectObject,hDC,hOld
        invoke DeleteDC,memDC
        invoke EndPaint,hWnd,ADDR ps
.elseif  uMsg == WM_CLOSE
        invoke EndDialog, hWnd, 0
.endif
xor eax,eax
Ret
dlgproc EndP

end start
```

حسنا الجديد هو القليل من الأسطر...لكن بالنسبة لعمل كيجين أو كراكمي ، فنحن يهمنا ما هو أسفل

```
if uMsg == WM_COMMAND
```

و أعتقد أنك تعرف ما أقصد...فمثلا في هذا المثال أضفنا الأسطر اللازمة للتعامل مع الضغط على زر allko حيث ستخرج messagebox تبعا لذلك...أنظر :

```
.if wParam==1002
        invoke MessageBox,hWnd,addr msg,addr msg,MB_OK
.endif
```

وبالمثل يمكنك إضافة أسطر لباقي الوظائف...
لكن ماذا عن قسم rc. ؟ عرفنا كيف نتعامل مع الزر لكن كيف نقوم بتعريف الزر (أو أي عنصر آخر) يدويا ؟
العمل اليدوي غير مناسب...لذلك كل ما عليك فعله هو أن تقوم بكتابة كراكمي أو كيجين مي باستخدام صورة bmp بشكل اعتيادي تماما...ثم نقوم بنسخ الأمور الضرورية وهي
1- التعاريف (أي ما يوجد أسفل data. و data?. و cons.)
2- ما يوجد أسفل if uMsg == WM_COMMAND
3- الإجرائيات ... مثلا عند الضغط على زر generate يتم استدعاء أجرائية generate ...تقوم بنسخ الاجرائية كما هي...
4- استبدال محتويات الـ rc. القديمة بالمحتويات الجديدة .

والآن الخطوة الأخيرة هي إضافة تأثير الـ fade in والـ fade out .
في أي برنامج أسمبلي كل ما عليك فعله لإضافة هذين التأثيرين هو التالي :
1- أسفل تعريف المكتبات (inc. و lib. ببداية البرنامج) أضف التالي :

```
FadeIn          proto       :DWORD
FadeOut         proto       :DWORD
```

2- أسفل data?. أضف السطر التالي :

```
Transparencydd        ?
```

3- أسفل const. أضف السطرين التاليين

```
DELAY_VALUE     equ     10
LWA_ALPHA       equ     2
```

4- في تعريف الـ dlgproc proc وأسفل uMsg==WM_INITDIALOG أضف السطر التالي :
(لاحظ وجود hWnd ، قد تكون عرفتها باسم آخر في برنامجك...اكتب الاسم الذي عرفتها به)

```
invoke FadeIn,hWnd
```

5- أسفل uMsg == WM_CLOSE أضف التالي :

```
invoke FadeOut,hWnd
```

في آخر البرنامج (أو في أي مكان ترغب به...لكن بالطبع لا تضعه في وسط إجرائية ما !!!) ضع الإجرائيتين التاليتين :

```
FadeIn proc hWnd:HWND
        invoke ShowWindow,hWnd,SW_SHOW
        mov Transparency,75
@@:
        invoke SetLayeredWindowAttributes,hWnd,0,Transparency,LWA_ALPHA
```

```
        invoke Sleep,DELAY_VALUE
        add Transparency,5
        cmp Transparency,255
        jne @b
        ret
FadeIn endp

FadeOut proc hWnd:HWND
        mov Transparency,255
@@:
        invoke SetLayeredWindowAttributes,hWnd,0,Transparency,LWA_ALPHA
        invoke Sleep,DELAY_VALUE
        sub Transparency,5
        cmp Transparency,0
        jne @b
        ret
FadeOut endp
```

سأترك تغيير القيم لك...غيرها بنفسك واكتشف ما الذي يحدث...

وفي الختام ، ستجد في المرفقات مجلد باسم final به ملفين نصيين وهو بمثابة tamplete يمكن استخدامها لأي كيجين .