# Built-In Assembler In Delphi 2006

# التجميع المضمن في ديلفي

http://www.arabteam2000-forum.com

إعداد: مراد / (Ikossan)

# بسم الله الرحمن الرحيم

## مقدمة :

هذا المقال موجه لكل من يريد إستخدام لغة التجميع Assembly language في ديلفي، و ليس يختص بشرح لغة التجميع بحد ذاتها. بل لجعل المبرمج بلغة ديلفي يستأنس بلغة التجميع المضمن ويستعمله في حالات يكون الحل الوحيد والأبسط هو لغة التجميع.

لست بصدد كتابة درس عن كيفية البرمجة بلغة التجميع للمبتدئين، فهو درس يطلب لقارئه ومتناوله حد أدنى من الدراية بقواعد البرمجة بلغة التجميع، ومعرفة بنظم العد كنظام العد الثنائيbinary system و نظام العد الستعشري hexadecimal system ، وبعض الأساسايات في طرق العنونة المباشرة والمفهرسة والمؤشرة.

وكذلك معرفة جيدة بمعالجات Intel x86 وبنيتها ومسجلاتها Registries، وكذا تعليمات لغة التجميع، وبطبيعة الحال معلومات جيدة عن لغة الباسكال، ولا يهم إن كنت لا تعلم عن النظم البرمجة في reel mode أو protected و virtual لمعالجات i386 لأنها غير مطلوبة في البرمجة لتطبيقات Win32.

ولن نتطرق لكيفية إستعمال لغة التجميع للتطوير و تحسين مردودية برامج ديلفي، لكن سيكون درسا عن كيفية كتابة لغة التجميع المضمن الذي يأتي مع ديلفي بطريقة سليمة و مفيدة.

ونذكر أن ديلفي المستهدف سيكون إنطلاقا من نسخته الأولى delphi2 لتطبيقات win32.
هناك بعض الميزات تمت إضافتها في نسخ لاحقة لديلفي كتعليمات SSE2 اللتي تم دمجها في delphi6.

Assembler المضمن سيمكنك من كتابة كود لغة التجميع في برامج ديلفي، وهو متوفر فقط في منصة delphi32، ويضم الميزات التالية:

- يمكنك كتابة كود التجميع inline.
- يضم كل تعليمات لغة التجميع المتوفرة لمعالجات pentium4 لإينتل، تعليمات MMX، وتعليمات SSE و تعليمات AMD Athlon بما فيها ! 3D Now.
- لا يمكنك كتابة macro لكن تستطيع كتابة دوال وإجرائيات مكتوبة بالكامل بلغة التجميع.
- تستطيع إستعمال كل معرفات Delphi مثل المتغيرات والتوابث، وtypes في كود assembly.

نذكر أنه بإمكانك إستعمال ملفات obj في برنامجك إذا أردت ربط إجرائيات أو دوال خارجية EXTERN، دون إستعمال التجميع المضمن في ديلفي.

# أ. التجميع المضمن في ديلفي:

## 1. تضمين لغة التجميع داخل الكود في ديلفي:

عملية جد سهلة وبسيطة لكي تكتب كود بلغة التجميع داخل كود برنامج ديلفي، يكفي أن تحيطه بالكلمتين المفتاحيتين asm في بداية النص ، وختمه ب end.

وكل التعليمات المحصورة بين هتين الكلمتين سيتم إدراجها كما هي في كود البرنامج التنفيذي.

مثال:

```
[code]
procedure TForm1.Button1Click(Sender: Tobject) ;
 var
  i,j: Integer;
  sum: Integer;
begin
  i:= 1;
  j:= 9;
  asm
    mov eax,i
    add eax,j
    mov sum,eax
  end;
  Label1.Caption:= IntToStr(sum);
end;
[/code]
```

## 2. تعليمات لغة التجميع الممكن كتابتها:

منذ delphi7 تم إدراج كل تعليمات التجميع لمعالجات intel . سواء كانت التعليمات الأساسية لمعالجات X86، أو تعليمات FPU، تعليمات الميلتيميديا MMX و SSE و SSE2 و كذلك 3d Now ! (طبعا ليست كلها متوفرة في كل المعالجات، بل حسب نوع المعالج).

بما أن ديلفي يبرمج لتطبيقات win32 فالمفروض إستعمال سجلات 32bits للمعالج إفتراضيا، وإن كان متاحا أيضا إستعمال سجلات 16bits و 8bits، فديلفي ستتولى كتابة التعليمة المناسبة وفق حجم المسجل، وحتى إستعمال العنونة extended addressing فهي مستعملة:

```
[code]
   mov eax,dword ptr[i]     //is the same as
   mov eax,i
[/code]
```

مثال آخر:

```
[code]
procedure TForm1.Button1Click(Sender: TObject);
var
  num: array[0..5] of byte;
  sum: Integer;
  I: Integer;
begin
  for I := 0 to 5 do num[i]:=i;
  sum:=0;
  asm
     xor eax,eax
     mov ecx,5
@@l:  mov al,byte ptr [num+ecx]
     add sum,eax
     loop @@l
  end;
  Label1.Caption:= IntToStr(sum);
end;
[/code]
```

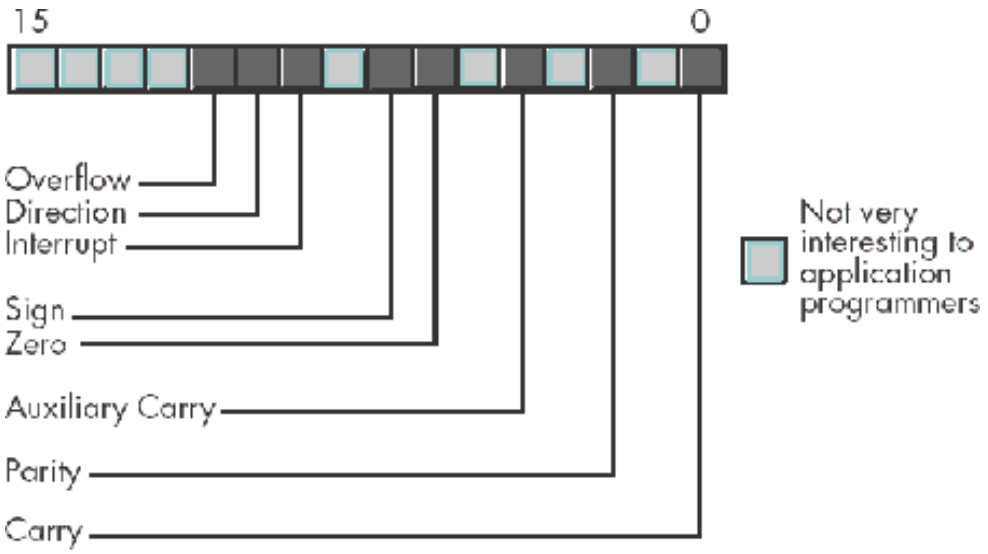
## 3. ما هي السجلات التي يمكن إستخدامها؟:

كل سجلات IA32 لمعالجات Intel، تصلح في التجميع المضمن لديلفي :

- السجلات العامة: eax,ebx,ecx,edx,edi,esi,ebp,esp وكذلك كلتقسيماتها ل 16bits و8bits: ax,al,ah,bx,bl,bh,cx,cl,ch,dx,dl,dh.

80386 Registers (Application Programmer Visible)

- سجل الحالة Eflags:




*Layout of the 80x86 flags register (LO 16 bits)*

- سجلات segments: ك ds وes وfs وgs وcs و ss، رغم أن إستعماله نادر جدا في ديلفي، وينصح بعدم المساس بها إلا للمبرمجين المتقدمين والمحترفين.
- سجلات وحدة الحساب.FPU و هي st0 إلى st7
- سجلات MMX وهي من mm0 إلى mm7
- سجلات SSE/SSE2 وهي xmm0 إلى xmm7 و سجل mxcsr.

| | | | |
|---|---|---|---|
| 32-bit general purpose | EAX EBX ECX EDX | 32-bit pointer or index | ESP EBP ESI EDI |
| 16-bit general purpose | AX BX CX DX | 16-bit pointer or index | SP BP SI DI |
| 8-bit low registers | AL BL CL DL | 16-bit segment registers | CS DS SS ES |
| | | 32-bit segment registers | FS GS |
| 8-bit high registers | AH BH CH DH | Coprocessor register stack | ST |

كل هذه السجلات يمكن كتابتها داخل حيز التجميع المضمن، لكن يجب أخذ بعين الأعتبار السجلات : ebx,esi,edi,ebp,esp,ds,es,cs و cs ومراعات عدم تغيير قيمها إلا بعد حفظها بتعليمات push وإسترجاعها ب pop.

```
[code]
asm
      PUSH    EDI
      PUSH    ESI
      MOV     ESI,EAX
      MOV     EDI,EDX
...........
...........
...........
      POP     ESI
      POP     EDI
end;
[/code]
```

سجلي القاعدة base registers و هما EBX و EBP وكذلك سجلي التأشير index registers (ESI و EDI) يمكن كتابتها بين معقوفتين للدلالة على التأشير indexing.

الزوج base/index المسموح به كتابة هي: [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], و[BP+DI] . وكذلك يمكنك إستعمال جميع سجلات 32Bits في التأشير كمثال: [ESP], [EAX+ECX], و [ESP+EAX+5] .

يمكنك كذلك إستعمال سجلات segments (ES, CS, SS, DS, FS, and GS) غير أنها غير ذي اهمية في تطبيقات 32bits.

---

## 4. سجلات segment في ديلفي:

سجلات segments يتكلف بها ديلفي، وليس من الفطنة المساس بهذه السجلات، بما أن ديلفي يعمل في النظام المحمي protected mode في العنونة 32bits، فإن كل الذاكرة للتطبيق ستكون متاحة دون تغيير قيم هذه السجلات كما كان العهد في win16 وفي 16bits dos.

فالسجلين DS و ES تشير كليهما إلى منطقة data للتطبيق، سواءً كانت هذه متغيرات static أو objects. فليس ضروري إستعمالها في العنونة بإستخدام النقطتين ك ": ES" أو" : DS" بما أنهما معا يشيران إلى نفس المنطقة في الذاكرة العامة للتطبيق. SS تشير إلى منطقة ذاكرة المكدس وهي أيضا في ديلفي لها نفس قيم ds وes. CS تشير إلى منطقة الكود.

أعيدها مرة أخرى، ديلفي تقوم على أكمل وجه بتولي هذه السجلات ولا يوجد داعي لإستعمالها للمبرمج في معظم الحالات. فيجب عدم المساس بها و الإبتعاد عن العبث فيها. فأنت في ديلفي لا تبرمج لتطبيقات 16Bits كما كان سابقل في تطبيقات Dos بل لتطبيقات تعمل في منصة Win32Bits.

## ب. كتابة لغة التجميع المضمن في ديلفي:

### 1. الكتابة الإملائية لتعليمات التجميع Assembler Statement Syntax:

الكتابة الإملائية لتعليمة لغة التجميع المضمن في ديلفي تكون على هذا النحو:

```
[code]
[label :]    Opcode    prefix operand1[, Operand2[, Operand3]]    // comment
[/code]
```

وسنشرح كل جزء من أجزاء هذه الصيغة الإملائية في ما سيلي.

### 2. تعليمات لغة التجميع المضمن في ديلفي Instruction OpCode :

عدد المعاملات يرجع للتعليمة المستعملة، فهناك تعليمات لا تحتاج لمعاملات و وهناك منها ما يحتاج لواحدة أو إثنين أو ثلاث.

OpCode هي كل تعليمات Intel المصرح بها في مطبوعات الشركة لمعالجات:

- Pentium family
- Pentium Pro and Pentium II
- Pentium III
- Pentium 4

بالإضافة لتعليمات:

- AMD 3DNow! (from the AMD K6 onwards)
- AMD Enhanced 3DNow! (from the AMD Athlon onwards)

تعليمت ret كلها من نوع near return.

تقوم ديلفي أيضا بترشيد وتحسين الكود الخاص بالقفز jumping بتحويله حسب موقعه في الكود إلى النوع المناسب، إما قصير short jump أو قريب near jump.
سيقوم ديلفي بتحسين الكود الخاص بالقفزات حسب الحاجة. فمثلا:

```
[code]
    JC          Stop
[/code]
```

ستصبح بعد تعديل ديلفي على هذا النحو:

```
[code]
    JNC     Skip
    JMP     Stop
Skip:
[/code]
```

## 3. المعاملات Operands :

من خلال صيغة الكتابة النحوية، بعد التعليمة تأتيمجموعة من معاملاتها، حسب نوع التعليمة OpCode.

فهناك تعليمات لا تحتاج لمعاملات، وهناك من التعليمات ما تتعدى لواحدة، أو إثنتين أو ثلاث.
يمكن للمعاملات أن تكون:

- قيم لحظية immediate values.
- السجلات العامة للمعالج.
- متغيرات variables لبرنامج ديلفي.
- عنوان ذاكرة مباشر أو مؤشر Direct address memory/ indexed address memory.

بالنسبة للقيم اللحظية فيجب أن تكون معروفة الكم قبل عملية التجميع compilation:

```
[code]
procedure TForm1.Button_Click(Sender: TObject);
const
 ten= 10;
 Max_byte= 255;
 I = $8000;
begin
  asm
    mov eax,$FA13014D
    mov ax,4
    mov cl, Max_byte SHR 3      //  هذه قيمة لحظية يستطيع المجمع معرفت قيمتها قبل التجميع
    mov edx,I + ten*Max_byte
  end;
end;
[/code]
```

يجب أن تكون operand المركبة من عدة قيم، محددة قبل عملية التجميع، فمثلا في المثال أعلاه Max_byte SHR 3 هي قيمة معروفة لأنها تتكون من قيم ثابتة وقيمتها هي 255 مقسومة على 2 أس 3، أي قيمتها هي = 31.

أنظر المثال التالي للتوضيح أكثر:

```
 [code]
procedure TForm1.Button5Click(Sender: TObject);
var
 X,Y,Z: integer;
begin
X:=23;
y:=17;
  asm
    mov eax,x
    add eax,y
    mov z,eax
  end;
  Label1.Caption:= 'z= x+y = '+ IntToStr(Z);
end;
[/code]
```

الكود الثاني:

```
[code]
procedure TForm1.Button6Click(Sender: TObject);
const
 x= 10;
 y= 45;
var
  z: integer;
begin
  asm
     mov z, x+y
  end;
  Label1.Caption:= 'z= x+y = '+ IntToStr(Z);
end;
[/code]
```

الفرق بين كتابة الكود الأول و الثاني تكمن في أن قيمة x+y في الكود الثاني هي قيمة لعددين تابثين و ما هي إلا طريقة لكتابة العدد 40. لهذا كتبنا mov z, x+y . في المثال الأول لا يمكنا كتابتها بنفس الطريقة، فقيم x و y هي قيم لمتغيرات لا يمكن معرفة قيمها إلا في زمن التنفيذ، لهذا كانت كتابة التعليمة في ثلاث سطور.

القيمة اللحظية يجب أن تكون دائما من نفس حجم السجل أو الموضع من الذاكرة التي ستحتويها، فلا يمكن أن تخزن قيمة 355 في المسجل al لأنها تتعدى القيمة القصوى 255 لل byte .

يقوم ديلفي بتمييز هذا الخطأ، لكن عند الإلتباس يجب تحديد نوع القيمة بأحد prefix مثل:
BYTE PTR و WORD PTRأو DWORD PTR...إلخ

*Predefined type symbols*

| Symbol | Type |
|---|---|
| BYTE | 1 |
| WORD | 2 |
| DWORD | 4 |
| QWORD | 8 |
| TBYTE | 10 |

مثال على ذلك :

```
[code]
procedure TForm1.Button4Click(Sender: TObject);
var
 B4: array[0..3] of byte;
 B8: array[0..7]of byte;
 W4: array[0..1] of dword;
 B10: array[0..9] of byte;
begin
  asm
    mov   dl, Byte PTR B4   // read  the 4 bytes of array B4
    mov   cx,Word ptr B4    // read  the 2 first bytes of array B4
    movq  MM1, B8           // read 8 bytes in the registre MM1 of MMX
    fld   TBYTE PTR b10     //  place the 10 bytes in the FPU Stack
    fld   QWORD PTR b10     //  place the 1st 8 bytes in the FPU Stack
    fld   w4
    lea   eax,B4
    mov   BYTE PTR [eax],3Fh
  end;
end;
[/code]
```

Prefix له دور كبير في تحديد Operands، هذه هي كل Prefix التي يمكنك إستعمالها في التجميع المضمن:

*Definitions of built-in assembler expression operators*

| Operator | Description |
|---|---|
| **&** | **Identifier override**. The identifier immediately following the ampersand is treated as a user-defined symbol, even if the spelling is the same as a built-in assembler reserved symbol. |
| **(... )** | **Subexpression**. Expressions within parentheses are evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the parentheses; the result in this case is the sum of the values of the two expressions, with the type of the first expression. |
| **[... ]** | **Memory reference**. The expression within brackets is evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the brackets; the result in this case is the sum of the values of the two expressions, with the type of the first expression. The result is always a memory reference. |
| **.** | **Structure member selector**. The result is the sum of the expression before the period and the expression after the period, with the type of the expression after the period. Symbols belonging to the scope identified by the expression before the period can be accessed in the expression after the period. |
| **HIGH** | Returns the high-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value. |
| **LOW** | Returns the low-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value. |
| + | **Unary plus**. Returns the expression following the plus with no changes. The expression must be an absolute immediate value. |
| **-** | **Unary minus**. Returns the negated value of the expression following the minus. The expression must be an absolute immediate value. |
| + | **Addition**. The expressions can be immediate values or memory references, but only one of the expressions can be a relocatable value. If one of the expressions is a relocatable value, the result is also a relocatable value. If either of the expressions is a memory reference, the result is also a memory reference. |
| **-** | **Subtraction**. The first expression can have any class, but the second expression must be an absolute immediate value. The result has the same class as the first expression. |
| **:** | **Segment override**. Instructs the assembler that the expression after the colon belongs to the segment given by the segment register name (CS, DS, SS, FS, GS, or ES) before the colon. The result is a memory reference with the value of the expression after the colon. When a segment override is used in an instruction operand, the instruction is prefixed with an appropriate segment-override prefix instruction to ensure that the indicated segment is selected. |

| | |
|---|---|
| **OFFSET** | Returns the offset part (double word) of the expression following the operator. The result is an immediate value. |
| **TYPE** | Returns the type (size in bytes) of the expression following the operator. The type of an immediate value is 0. |
| **PTR** | **Typecast operator**. The result is a memory reference with the value of the expression following the operator and the type of the expression in front of the operator. |
| * | **Multiplication**. Both expressions must be absolute immediate values, and the result is an absolute immediate value. |
| / | **Integer division**. Both expressions must be absolute immediate values, and the result is an absolute immediate value. |
| **MOD** | **Remainder after integer division**. Both expressions must be absolute immediate values, and the result is an absolute immediate value. |
| **SHL** | **Logical shift left**. Both expressions must be absolute immediate values, and the result is an absolute immediate value. |
| **SHR** | **Logical shift right**. Both expressions must be absolute immediate values, and the result is an absolute immediate value. |
| **NOT** | **Bitwise negation.** The expression must be an absolute immediate value, and the result is an absolute immediate value. |
| **AND** | **Bitwise AND**. Both expressions must be absolute immediate values, and the result is an absolute immediate value. |
| **OR** | **Bitwise OR**. Both expressions must be absolute immediate values, and the result is an absolute immediate value. |
| **XOR** | **Bitwise exclusive OR**. Both expressions must be absolute immediate values, and the result is an absolute immediate value. |

## 4. Labels:

Label هي وسم لتعليم موضع في الكود، تستعمل عادة في تعليمات القفز و تعليمات الحلقات التكرارية.

**Label تستعمل في لغة التجميع المضمن كمثيلتها في الباسكال، يجب تعريفها قبل استعمالها بالكلمة المفتاحية "Label" :**

```
[code]
procedure TForm1.Button7Click(Sender: TObject);
var
  num: array[0..5] of integer;
  sum: Integer;
  I: Integer;
label lbl_Loop,Ends,Starts;
begin
  for I := 0 to 5 do num[i]:=i;
```

```
  sum:=0;
  asm
starts:
      push  ebx
      push  edi
      xor   eax,eax
      mov   ecx,5
lbl_Loop:
      mov   eax,dword ptr [num+ecx*4]
      add   sum,eax
      loop  lbl_Loop
Ends:
      pop   edi
      pop   ebx
  end;
  Label1.Caption:= IntToStr(sum);
end;
[/code]
```

يمكن إستعمال labels داخل جسم asm/end دون تعريفها، حيث يكفي سبق كلمتها بالرمز @:

```
[code]
@Loops:
      mov   eax,dword ptr [num+ecx*4]
      add   sum,eax
      loop  @Loops
[/code]
```

## 5. توجيهات لغة التجميع Assembly Directives:

ليس كل التوجيهات المستعملة في لغات التجميع يمكن إستعمالها في لغة التجميع المضمن في ديلفي، سوى توجيهات db,dw,dd,dq:

| Directive | Description |
|---|---|
| DB | تعريف Byte : تقوم بحجز مجموعة من البيانات متسلسلة بعرض بايت واحدة. كل بينة من البيانات يمكنها إذن أن تأخذ قيمة ما بين -128و 255، أو قيمة محرفية،أو سلسلة محرفية بأي طول. فأي قيمة عددية لحظية ستأخذ موضع بايت في الذاكرة، وأية سلسلة محرفية ستتحول إلى سلسلة من القيم العددية الموافق لحروفها بكود الأسكي. |
| DW | تعريف كلمة 16بيتWord : تقوم بحجز سلسلة من القيم العددية من عرض كلمة أي 16Bits فلا يمكنها حفظ سوى الأعداد الممتدة من 32768- إلى 65535. و قد تضم تعريف عنوان لذاكرة near من حجم 16Bits . |
| DD | تعريف كلمة مضاعفة Dword: يمكن حجز سلسلة من كلمات مضاغفة بسعة 32Bits. و كذلك تسمح بحفظ عناوين ذاكرة من نوع Far. |
| DQ | تقوم بتعريف كلمة رباعية quad word: تستعمل لحجز سلسلة من الأعداد ذات سعة 64Bits كالأعداد المعفة في ديلفي ب Int64. |

مثال لتعريف متغيرات بقيم تابثة بواسطة هذه التوجيهات داخل قسم asm/end، رغم أنه يفضل إستخدام موجهة ديلفي لتعريف المتغيرات بإستخدام Var أو Const:

```
[code]
procedure TForm1.Button8Click(Sender: TObject);
var
  C_str: pchar;          //  c strings
  p_str: PShortString;  //  pascal string
begin
asm
  mov eax, TYPE Integer
  mov ecx,  10
  //......
  //......
  jmp   @ToCode

@@my_c_str:
  db  'Welcom In delphi assembler inline',0h
@@my_p_str: { pascal ShortString followed by CR/LF and a 2ndline }
  DB 30,'Hello world...',0DH,0AH,'Arab Team 2000'
@my_varB:
  db  12
@my_Charb:
  db  'C'
@my_varW:
  dw  $f4c2
```

```
@my_varDw:
  dd  $12345678
@my_varQ:
  dq  12

@ToCode
  mov   P_str,offset @@my_p_str
  mov   c_str,offset @@my_c_str
end;
[/code]
```

يفضل إستخدام موجهة ديلفي لتعريف المتغيرات بإستخدام Var، مثل هذه التعريفات في المثال أسفله غير معتمدة في لغة التجميع المضمن في ديلفي، فعوض كتابة متغيرات داخل asm/end بهذا الشكل:

```
[code]
ByteVar    DB     ?        // error
WordVar    DW     ?        // error
IntVarDD   ?      // error
.....
....
..
  MOV      AL,ByteVar
  MOV      BX,WordVar
  MOV      ECX,IntVar
[/code]
```

سينتج عنها رسائل خطأ لهذا قم بتعريف المتغيرات خارج asm/end بواسطة Var:

```
[code]
var
    ByteVar: Byte;
    WordVar: Word;
    IntVar: Integer;
...
...
..
   asm
    MOV AL,ByteVar
    MOV BX,WordVar
    MOV ECX,IntVar
   end;
[/code]
```

تقوم هذه الموجهات بحجز سلسلة من البايتات Bytes مباشرة في شيفرة الكود segment code ، لهذا فإن ديلفي يتعامل معها كما لو كانت سلسلة من bytes خام تزرع كما هي داخل نص الكود، لاحظ إستعمال تعليمة القفز@ToCode jmp عندما عرفنا مجموعة من المتغيرات كتوابث داخل asm/end في المثال فوق.

لهذا يمكنك إستعمالها لزرع جزء من الكود مكتوب مباشرة بلغة الآلة.

أفضل مثال على هذا هي إستعمال تعليمة RDTSC المعروفة في معالجات pentium التي تقوم بقراءة العداد من سعة Int64 و وضعه في السجلين edx :eax حيث edx تضم word الأعلى و eax تضم word الأدنى:

```
[code]
function Read_RDTSC: Int64;
asm
  rdtsc
end;
[/code]
```

يمكنك وضع هذه التعليمة ماشرة وسط الكود بإستعمال هذه التوجيهات على هذا النحو في المثال التالي الذي سيمكن من حساب عدد نبضات المعالج اللازمة لتنفيذ التعليمات التي بين الإستدعاء الأول والإستدعاء الثاني.

فيمكن كتابتها في نسخ ديلفي التي لم تكن تعتمد هذه التعليمة ك ديلفي 2 مثلا ( أدمجت تعليمة RDTSC في ديلفي إنطلاقا من النسخة السادسة):

```
[code]
function Read_RDTSC: Int64;
asm
  dw  $310F     //rdtsc
  // Or
  db  $0F,$31   //rdtsc
end;
[/code]
```

---

## 6. تنبيه في إستعمال بعض المعاملات خاصة بالتجميع المضمن لديلفي:

تستعمل بعض prefix إستعمال خاص في القسم asm/end، لهذا وجب التنبيه لها لأهميتها:
**أنظر جدول prefix المرفق في الفقرة " 3.المعاملات Operands ".**

- **Low:** تقوم بإرجاع الوزن الأدنى لل Operand، يجب عدم الخلط بينها وبين دالة الباسكال low().

```
[code]
asm
  mov   al, low WordVar
  mov   ax, low DwordVar
  mov   eax, low Int64Var
end;
[/code]
```

- **High**: تقوم بإرجاع الوزن الأعلى لل Operand، يجب عدم الخلط بينها وبين دالة الباسكال high().

```
[code]
asm
  mov   al, low WordVar
  mov   ax, low DwordVar
  mov   eax, low Int64Var
end;
[/code]
```

- **Type**: ترجع سعة الOperand بالبايت، فهي ترجع عدد البايتات التي تحتجزها operand، تقابلها في الباسكال ديلفي الدالة SizeOf().

```
[code]
Var
 WordArr: array[0..15] of word;
 Delphi_str: string;
 Pascal_str1: string[45];
 PVar: Pointer;
asm
  mov   eax, TYPE  Delphi_str
  mov   eax, TYPE  Pascal_str1
  mov   eax, Type  pvar
  mov   eax, TYPE  WordArr
end;
[/code]
```

- <:> : تستعمل في تحميل segments عند إستدعاء تعليمات لغة التجميع.

```
[code]
   mov   eax, ss:wvar
   mov   eax, cs:wvar
   mov   eax, es:wvar
   mov   eax, ds:wvar
[/code]
```

إستعمالها ليس بأهمية في تطبيقات win32 كما ذكرت سابقا.

- **Brackets []**: تدل على عنوان في الذاكرة، ويمكن أن يضم بداخلها سجلات أو متغير أو قيم لحظية. وكل قيمة عددية تحسب على أنها عدد البايتات المضافة للحصول على عنوان الذاكرة:
**(أنظر الفقرة ث.1 حول نظم العنونة في معالجات IA32 ، سأتناول شرح حول إستخدامها و معانيها)**

```
[code]
var
 MyArr: array[0..255] of byte;
 i: integer;
begin
for I := 0 to 255 do MyArr[i]:=i;

  asm
    xor   ecx,ecx
    lea   eax, MyArr    // eax = adress of the array in memory
    mov   edx, [eax]
    mov   edx, [eax + ecx*4]
    mov   ecx,1
    mov   [eax][ecx+6], $3f
  end;
end;
[/code]
```

تستعمل المعقوفات كثيرا لفائدتها في العنونة المفهرسة أو المؤشرة. فسجلي القاعدة base registers و هما EBX وEBP وكذلك سجلي التأشير index registers (ESI و EDI) يمكن كتابتها بين معقوفتين للدلالة على التأشير indexing.

الأزواج base/index المسموح بها في كتابة لغة التجميع في ديلفي هي: [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], و[BP+DI] . وكذلك يمكنك إستعمال جميع سجلات 32Bits في التأشير كمثال: [EAX+ECX], [ESP] و[ESP+EAX+5] .
(أنظر الفقرة ث.1)

- **<u>المعاملات المنطقية و المعاملات الحسابية و معاملات Bits</u>:** و هي معاملات bits ماتسمى ب Bitwise و هي **NOT**, **AND**, **OR**, **XOR** و كذلك المعاملات المنطقية و هي SHL **و** SHR، و المعاملات الحسابية و هي + و – و / و * و MOD.

كل هذه المعاملات يمكنك إستعمالها على operand شريطة أن يكون الخارج قيمة لحظية مطلقة معروفة لديلفي قبل compilation.

```
[code]
 asm
   mov   ax,  not $ff
   mov   edx, $FF00 and 234
   mov   edx, MyVar shr 4            // error   "MyVar shr 4" is not a constant
   mov   edx, (type MyVar) shr 4     // ok
```

```
    mov   cl, 1+13-(255/23)
    mov   eax, $12345678 mod -32
  end;
[/code]
```

## 7. كتابة التعليقلت Comments:

كتابة التعليقات في كود لغة التجميع لديلفي تعد مطلوبة لصعوبة مقروئية كود التجميع، فيمكن بعد عدة شهور ستجد نفسك عاجز عن قراءة أجزاء من الكود مكتوبة بلغة التجميع لخلوها من التعليقات حتى ولو كنت أنت من كتبها.

التعليقات تكتب بنفس الطريقة في ديلفي، بإستعمال // أو {} أو (* و *).

## ت. إستعمال أصناف ديلفي في التجميع المضمن:

جميع الأصناف المعرفة في ديلفي، سواءً الأصناف الإفتراضية أو الأصناف التي يعرفها المبرمج يمكن للمبرمج التغامل معها داخل شيفرة كود لغة التجميع.

وتعد الأصناف الستاتيكية الأصناف الأسهل وصولا و الأبسط إستعمالا في لغة التجميع المضمن، وسنتطرق لمعظم هذه الأصناف في الفقرات التالية.

في المقابل، الأصناف الديناميكية مثل المتسلسلات المحرفية لديلفي String أو الجداول الديناميكية مثل Array of... ، لها إستعمال معقد وصعب لأن معظم دوالها هي دوال غير مصرح بها في ديلفي وتعد دوال خاصة مكن لشركة بورلاند أن تعدل فيها دون سابق إخبار. لن نتطرق إليها في هذا الدرس.

في أغلب الأحيان ستلحظ أننا لن نستعمل الموجهة xxx PTR عند إستحدام المتغيرات، فإن ديلفي سيقوم بإخبار المبرمج بكل إلتباس إذا لم يكن هناك توافق مع حجم المصدر أو الوجهة التي ستحتويها.

### 1. الأصناف المرتبة Ordinal Types:

الأصناف المرتبة هي الأصناف البسيطة التي يقوم ديلفي بتخزينها في سعة 8bits أو Bits16 أو 32Bits. و هي أصناف integer و character و Boolean و enumerated و Pointer و subrange types .

يقوم ديلفي بالتعامل مع هذه الأصناف كما لو كانت أعداد صحيحة طبيعية وفق سعتها بالبايت.
كما أن إستعمال المعقوفتين ليس بضروري عند التعامل مع المتغيرات، وكذلك ليس ضروري تحديد صنف المتغير بالموجهات xxx PTR MyVar، فمثلا كل هذه السطور الثلاثة متطابقة وسينتج عنها نفس كود لغة التجميع:

```
[code]
  mov       DWORD PTR [MyVar],$d316
  mov       DWORD PTR MyVar,$d316
  mov       MyVar,$d316
[/code]
```

مثال لطريقة التعامل مع بعض هذه الأصناف في لغة التجميع لديلفي:

```
[code]
TYPE
 TWeekDay = (Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday);
procedure TForm1.Button12Click(Sender: TObject);
VAR
  n: Integer;      //  Dword=32bits
```

```
  x,y: Smallint;    //  both x,y are Word = 16bit
  c: Char;          //  Byte=8bits
  b: Byte;
  P: Pointer;       //  All pointers are a Dword (32Bits)
  Day: TWeekDay;    //  day is a byte in this case
begin
 asm
  mov   n,$d316
  mov   eax,n
  mov   al,c
  mov   c,Edx   //  [Pascal Error]: E2107 Operand size mismatch
  mov   c,dl    // it's ok
  mov   dl,b
  mov   eax,p
  mov   cx,word ptr day
  mov   ax,x
  mov   y,ax
 end;
end;
[/code]
```

ملاحظة هامة بخصوص صنف enumerated الذي يمكنه أن يخزن في Byte أو في word أو في Dword حسب قيم عناصره المكونة له:

```
[code]
TYPE
TenumByte = ( Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday);
 TEnumWord =  (MaxNibell= $F, MaxByte=$FF, MaxWord = $FFFF);
 TEnumDword = (Zerro=0, Kilo=1024, Mega=Kilo *1024, Gega= 1024*Mega);

procedure TForm1.Button13Click(Sender: TObject);
var
  ByteVar: TEnumByte;    //  day is a byte in this case
  WordVar: TEnumWord;    //  day is a byte in this case
  DwordVar: TEnumDword;    //  day is a byte in this case
begin
  asm
     mov   al, ByteVar   // Byte value
     cmp   al, Wednesday

     mov   ax, WordVar   // Byte value
     cmp   ax, MaxWord
```

```
    mov   eax, DwordVar   // Byte value
    cmp   eax, Gega
  end;
end;
[/code]
```

## 2. صنف المجموعة Sets Types:

صنف المجموعة هو صنف اختصت به ديلفي منذ بدايتها عن باقي لغات البرمجة. و تقوم ديلفي بتمثيلها بمجموعة من البتات bits في الذاكرة، وكل بت يمثل إنتماء عنصر إلى المجموعة.
فكل بيت به واحد يرمز لأنتماء هذا العنصر للمجموعة، و لو كان بالبت قيمة 0 فسيرمز بالنفي لأنتماء العنصر للمجموعة.

لهذا فإن عدد البتات Bits هو بطبيعة الحال هو عدد العناصر التي يمكنها الإنتماء للمجموعة، وديلفي يمكنها أن تخزن هذا التمثيل من بت واحد 1bit إلى 256bits، يعني سيلزم المجموعة حجز من بايت 1byte إلى 32bytes.

فكل مجموعة تتكون من عنصر إلى 8 عناصر سيلزمها بايت واحدة Byte، وكل مجموعة من 9 إلى 16 سيلزمها Word، وكل مجموعة من 17 إلى 32 سيستعمل لتخزينها Dword.
ولو كان عدد العناصر أكثر من 33 فما فوق فعملية التحزين ستستلزم مجموعة من Bytes الكافية لإحتوائها.

فديلفي سيخصص عدد البايت وفق عدد العناصر، فلو كانت مجموعة يمكنها أن تضم 6 عناصر فسيلزم لتخزينها بالذاكرة إلى بايت واحدة، ولو كان عدد العناصر 45 مثلا، فالأقرب ل 45 من مضاعفات 8 هو 48bits أي 6 بايتات.

عندما يكون عدد العناصر أقل من 32bits، فإن عملين التعامل تكون مباشرة، أما إذا تعدت لأكثر، فستستلزم عدة مراحل كما لو كنا نتعامل مع جدول من bytes أو من words أو Dword.

```
[code]
TYPE
 TWeekDay = (Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday);
 TJob = set of Monday..Friday;    //  set of 5 elements <= 8bits = 1Byte to store
 Tday =set of TWeekDay;           //  set of 7 elements <= 8bits = 1Byte to store
 TCodeRange = set of 'A'..'Z';    //  set of 26 element <=32 = 1Dword to store
 TPorts = set of 0..100;          //  Set of 101 element <= 104Bits = 13Bytes to store
procedure TForm1.Button14Click(Sender: TObject);
var
  JobDay: TJob;
  FreeDay: Tday;
  Codes: TCodeRange;
  Ports: TPorts;
begin
JobDay:= [Monday,Wednesday,Thursday,Friday];
```

```
FreeDay:= [Monday,Wednesday,Thursday,Friday,Saturday];
Codes:= ['A','C','F','K','E','N','Z','M','W','M'];
Ports:= [0..10,30..45,60..63];
asm
  mov   al, jobday
  cmp   al, Monday + Wednesday + thursday

  mov   cl, jobday
  mov   al, freeday
  and   al, cl

  mov   eax, Codes
  cmp   eax, $F000

  mov   eax, Dword PTR Ports[2]
  mov   cx, word ptr[Ports+3]
  mov   byte ptr Ports[13], $01
end;
end;
[/code]
```

## 3. الأصناف الأعداد الحقيقية Real Types:

صنف الأعداد الحقيقية و هي الأصناف الأساسية التالية:

***Fundamental Win32 real types***

| Type | Range | Significant digits | Size in bytes |
|---|---|---|---|
| **Single** | -1.5 x 10^45 .. 3.4 x 10^38 | 7-8 | 4 |
| **Double** | -5.0 x 10^324 .. 1.7 x 10^308 | 15-16 | 8 |
| **Extended** | -3.6 x 10^4951 .. 1.1 x 10^4932 | 10-20 | 10 |
| **Currency** | -922337203685477.5808.. 922337203685477.5807 | 10-20 | 8 |

الأصناف Real48 و Real و Comp لا تعد من الأصناف الأساسية. Real هي Real48 لا تزال تعتمد في ديلفي للتوافقية فقط مع النسخ السابقة، و كذلك Comp فهي تسمية لل Currency.
الأصناف الأساسية يمكن التعامل معها مباشرة بواسطة سجلات وحدة الحساب FPU، فسجلاتها تتعامل معها بديهيا ، مباشرة و بكل سهولة:

```
[code]
procedure TForm1.Button15Click(Sender: TObject);
var
 s1,S2: Single;
```

```
 D1,D2: Double;
 E1,E2: Extended;
 C1,C2: Currency;
begin
asm
  fld   S1
  fadd  S2
  wait
  fst   S1

  fld   D1
  fdiv  D2
  wait
  fst   D1

  fld   E1
  fld   E2
  fmul
  wait
  fstp   E1

  fld   C1
  fsub  C2
  wait
  fst   C1
end;
end;
[/code]
```

## 4. صنف العدد الصحيح الطبيعي In64:

صنف العدد الصحيح الطبيعي Int64 لا يمكنه أن يدخل في سجلات المعالج، لأن أكبر سعة لسجلات المعالج هي 32bits. بل لا توجد حتى تعليمات لغة التجميع التي تتعامل مع 64bits.
لهذا فكل تعالمل مع هذا المتغير يتم عبر تمثيل قيمته بوحدتين من سعة Dword، وغالبا ما يستعمل الزوجEAX: EDX للتعبير عن قيم Int64. حيث Dword الأدنى يكون في Eax و dword الأعلى يسجل في Edx. هذه ليست قاعدة بل يمكنك إستعمال ما تشاء من السجلات العامة لتمثيله بها.

```
[code]
procedure TForm1.Button16Click(Sender: TObject);
var
 a,b: Int64;
 sum: int64;
begin
a:= 55550000;
b:= 00009999;
sum:=0;
  asm
```

```
    mov   eax,dword ptr a   // load the Low dword of A
    add   eax,dword ptr b   // add eax with The Low Dword of B
    mov   dword ptr sum,eax
    mov   eax,dword ptr a+4   // load the High dword of A
    adc   eax,dword ptr b+4   // add eax with The Dword of B
    mov   dword ptr sum+4,eax
  end;
  Label1.Caption:= IntToStr(sum);
end;
[/code]
```

هنالك تعليمات تخص FPU أو وحدات MMX يمكنها التعامل طبيعيا مع الأعداد الصحيحة Int64، وهذا مثال يغني عن أستعمال dword PTR في FPU:

```
[code]
procedure TForm1.Button16Click(Sender: TObject);
var
 a,b: Int64;
 sum: Extended;
begin
a:= 55550000;
b:= 00009999;
sum:=0;
  asm
    fild   a    //  load the 64bits of a
    fild   b    //  load the 64bits of b
    fadd        //  a+b
    wait
    fstp  sum   // sum= result
  end;
  Label1.Caption:= FloatToStr(sum);
end;
[/code]
```

## 5. صنف الجداول الستاتيكية Static Array Types:

الجداول هي صنف من أصناف ديلفي لسلسلة محددة الطول من البيانات من نفس السعة و الصنف. للتعامل مع عنصر من عناصر الجدول فلابد من إستعمال أداة تحديد السعة Ptr xxx مع عدد يؤشر عن موقعه في السلسلة.

فمعرف الجدول هو عنوان في الذاكرة لأول عنصر، والعناصر الأخرى تسجل تباعا لموضع العنصر الأول. فلا يهم إن كان أول مؤشر Index يبدأ من 0 أم من عدد آخر، مثلا الجدولين التاليين مسجلين في الذاكرة بنفس الطريقة.

```
[code]
var
  Arr1: array[0..5] of integer;
  Arr2: array[3..8] of integer;
[/code]
```

للوصول لعناصر الجدولين سنستعمل المعقوفات [..] ،غير أنها ليس بنفس المعنا عند إستعمالها خارج جزء asm/end، فهي لا تعني Index بل تفيد عدد البايتات للتحرك إنطلاقا من عنوان في الذاكرة.

فللوصول للعنصر الثالث من الجدول Arr1 فلن نكتب Arr[3]، بل سنكتب التحرك بعدد البايتات بالنسبة لموقع عنوان الجدول في الذاكرة: Arr[3* Type (Integer)] أي Arr1[12]. و يمكن كذلك كتابتها Arr1 + 12 دون إستعمال المعقوفات. الجداول دائما نتعامل معها كما لو كانت تبتدئ من المؤشر :0

```
[code]
procedure TForm1.Button14Click(Sender: TObject);
var
  MyArr: array[1..20] of integer;
  sum: integer;
begin
asm
    xor   eax,eax   // eax:= 0
    xor   ecx,ecx   // ecx:= 0
@Loop1:
    add   eax, DWORD PTR MyARR[ECX]
    add   ecx, Type INTEGER
    cmp   ecx, Type  MyArr
    JB    @Loop1
    mov   sum, eax
end;
  Label1.Caption:= FloatToStr(sum);
end;
[/code]
```

فالمثال قمنا بجمع قيم عناصر الجدول MyArr ، إستخدمنا كمؤشر للوصول لعناصر الجدول: السجل ECX.

ECX ستأخذ قيم مضاعفات 4 لأنها هي عدد Bytes المكونة لصنف Integer المكون لعناصر الجدول، يمكننا إعادة كتابة نفس المثال بطريقة أخرى بحيث سنؤشر ب Ecx لكن سنقوم بجرد العناصر بتعليمة Loop التي ستنقص ecx بواحد حتي تنعدم.

```
 [code]
procedure TForm1.Button14Click(Sender: TObject);
var
  MyArr: array[1..20] of integer;
  sum: integer;
  I: Integer;
begin
for I := 1 to 20 do MyArr[I]:= I;
asm
    xor   eax,eax   // eax:= 0
    mov   ecx, (Type MyArr) / (Type Integer)   // ecx:= 20 Number of MyArr Elements
@Loop1:
     // ECX* (Type Integer)-4:  -4 because we scaned the array  from 19 to 0
    //  and not from 20 to 1 !! atention :)
    add   eax, DWORD PTR MyARR[ECX* (Type Integer)-4]
    Loop  @Loop1
    mov   sum, eax
end;
  Label1.Caption:= FloatToStr(sum);
end;
 [/code]
```

الكتابة :

DWORD PTR MyARR[ECX* (Type Integer)-4]،

أو DWORD PTR [MyARR+ECX* (Type Integer)-4]،

فهي نفس الشيء ويعود لذوق المبرمج في الكتابة(أنظر الفقرة ث.1). نذكر كذلك أن هنالك بعض التعليمات التي يمكها الوصول للجداول بقراءة 8 بايتات مرة واحدة كما في المثال التالي الذي يستعمل بعضا من تعليمات MMX:

```
 [code]
procedure TForm1.Button1Click(Sender: TObject);
var
  ArrB1,ArrB2: array[0..7] of Byte;
  ArrW1,ArrW2: array[0..3] of Word;
  ArrD1,ArrD2: array[0..1] of Dword;
begin
  asm
    //  ArrB1: = ArrB1 + ArrB2
    MOVQ    MM0, ArrB1
    PADDB   MM0, ArrB2
    MOVQ    ArrB1, MM0

    //  ArrW1: = ArrW1 + ArrW2
    MOVQ    MM0, ArrW1
    PADDW   MM0, ArrW2
    MOVQ    ArrW1, MM0

    //  ArrD1: = ArrD1 + ArrD2
```

```
    MOVQ    MM0, ArrD1
    PADDD   MM0, ArrD2
    MOVQ    ArrD1, MM0
  end;
end;
 [/code]
```

## 6. صنف المتسلسلة المحرفية القصيرة للباسكال ShortString Types :

المتسلسلات المحرفية للباسكال ShortString أو ما يمنكن تعريفها ب String[n] ، تعتبر كأنها جداول من نوع خاص، حيث أن عناصرها هي سلسلة من المحارف من صنف Char و العنصر الأول يحتوي على طول السلسلة. ولا يمكنها أن تتدا 256 خانة، بحساب لحانة الأولى.

نتعامل مع هذه المتسلسلات كما تعاملنا مع الجداول، وهناك ميزة أخرى تنطبق عليها هي أن لغة التجميع تضم عددا من التعليمات التي تسهل التعامل مع السلسلات المحرفية بسهولة، هذه التعليمات يمكن أيضا تطبيقها على الجداول أيضا دون قيد أو شرط، سوى أن أول عنصر في السلاسل المحرفية يحتوي على طولها:

(المثال فيه طريقة التعامل مع السلاسل التابثة ذات تعامل جد خاص.)

```
 [code]
procedure TForm1.FormCreate(Sender: TObject);
var
 str1,str2: ShortString;
const
  TeamStr = 'www.ArabTeam2000-forum.com';
  TeamStrP: Pointer = Pchar(TeamStr);
  strlen= Length(TeamStr);
begin
asm
  push  edi
  push  esi

  //  edx = adress of the first char of str1
  lea   edi,str1[1]
  mov   ecx, 33
  mov   al,'A'
  rep   stosb     // fill str1 with 33 char = 'A'
  // set the lenght=33 of the string in the first element str1[0]
  mov   byte ptr str1[0],33

  //  copy the TeamStr into str2
  cld
  mov   esi, TeamStrP       //  esi = adress of source (TeamStr)
  lea   edi, str2[1]        //  esi = adress of destination (Str2)
```

```
  mov   ecx, strlen        //  cl= lenght of the TeamStr
  rep   movsb
  // set the lenght of the string in the first element str2[0]
  mov   byte ptr str2, strlen

  pop   esi
  pop   edi
end;
Label1.Caption:= str1;
Label2.Caption:= str2;
end;
[/code]
```

## 7. صنف التسجيلات Records Types :

التسجسلات لها إستعمالات كبيرة في ديلفي، وهي مثل الجداول تسجل في الذاكرة متسلسلة تباعاً. غير أنها تضم يانات متباينة الأصناف ليس كالجداول.

في التجميع المضمن، نتعامل مع التسجيلات كما تعاملنا مع الجداول، أي بمسماها الذي يدل على عنوانها في الذاكرة ، والميزة الكبيرة هي أن التجميع يقبل المعامل "نقطة" كما تستعمل في الوصول لحقول التسجيلات في باسكال.

```
[code]
type
 TMyRec = Record
   b: byte;
   w: word;
   b2: byte;
   d: dword;
   b3: byte;
   q: int64;
 End;
procedure TForm1.Button17Click(Sender: TObject);
Var
MyRec:TMyRec;
begin
  Asm
    MOV al,MyRec.b
    MOV AX,MyRec.w
    MOV ah,MyRec.b2
    MOV EAX,MyRec.d
    MOV cl,MyRec.b3
    MOV EAX, dword ptr MyRec.q
  end;
end;
[/code]
```

يمكن إستعمال المعقوفات في العنونة Indexing adress للوصول لحقول التسجيل، لكن يجب توخي الحذر لأن ديلفي يقوم بتصفيف records على 1Byte أو 2 أو 4 أو 8، مايسمى ب Record Field alignment. أنظر help لديلفي حول تنسيق التسجيلات في:

. Project ▸ Options ▸ Compiler

أنظر إستعمال الوسم {$A-} الذي يلغي تنسيق التصفيف في الذاكرة لل records.

```
[code]
{$A-}
type
 TMyRec = Record
   b: byte;
   w: word;
   b2: byte;
   d: dword;
   b3: byte;
   q: int64;
   b4: byte;
 End;
asm
    MOV al,MyRec.b
    MOV al,byte ptr MyRec              //  MyRec.b

    MOV AX,MyRec.w
    MOV AX,word  ptr MyRec[1]           //  MyRec.w

    MOV ah,MyRec.b2
    MOV ah,byte  ptr MyRec.[3]          //  MyRec.b2

    MOV EAX,MyRec.d
    MOV EAX,dword ptr MyRec[4]          //  MyRec.d

    MOV cl,MyRec.b3
    MOV cl,byte  ptr MyRec[8]           //  MyRec.b3

    MOV EAX, dword ptr MyRec.q
    MOV EAX,dword ptr MyRec[9]          //  low(MyRec.q)

    MOV cl,MyRec.b4
    MOV cl,byte  ptr MyRec[17]          //  MyRec.b4
end;
[/code]
```

الأرقام التي بين المعقوفات هي عدد البايتات للتحرك Offset للوصول للحقول. الأمر معقد لهذا أفضل إستعمال المعامل "نقطة " كما في الباسكال.

## ث. كتابة الدوال و الإجرائيات بلغة التجميع المضمن في ديلفي:

### 1. تقديم حول نظم العنونة لمعالجات x86 ما سمى ب Memory Adressing Modes:

كل الأصناف التي تكلمنا عليها آنفا من أصناف الديلفي ستمثل في خانات بالذاكرة، لكل خانة عنوان يقوم ديلفي بحجزها لتضم المتغيرات لمختلف هذه الأصناف يرمز لموضعها بالذاكرة.

فاستخدام عنوان ما في الذاكرة للوصول إلى محتواه يمكن أن يختلف من حالة برمجية إلى أخرى، أو من طبيعة تسجيل القيم بالذاكرة.

معالجات 80x86 تملك عدة طرق للقراءة و الكتابة في الذاكرة، وهذه الطرق كثيرة و متنوعة توفر للمبرج حرية كبيرة في الولوج للمتغيرات لكافة الأصناف و التعامل معها بتعليمات لغة التجميع .

ربما تعد نظم العنونة من أهم أبواب لغة التجميع لكل من أراد إحتراف هذه اللغة و الإبداع فيها.

أهم أنظمة العنونة هي العنونة الغير مباشرة و صيغ أخرى يمكن تحميلها نفس المعنى سننقاشها كلها أو بعضها بعجالة قدر الأمكان.

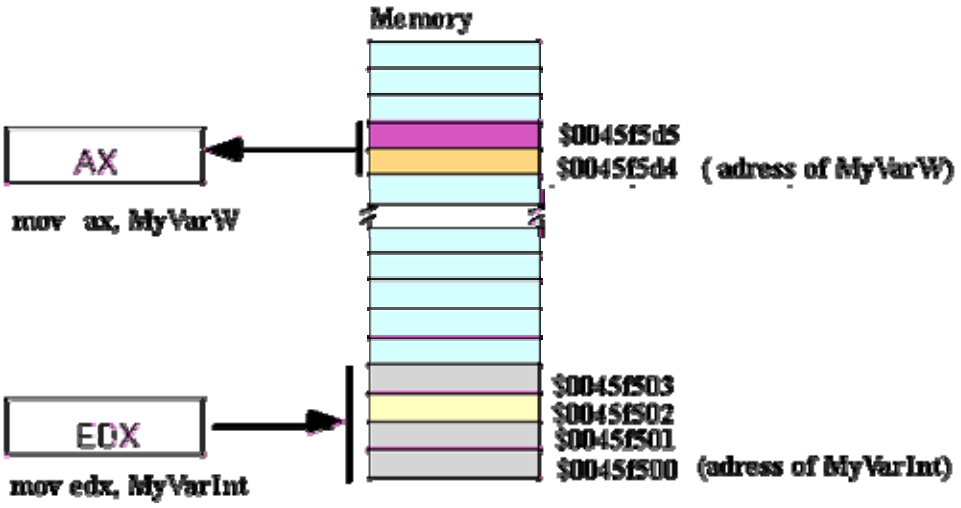
**a) نظم العنونة المباشرة و الغير مباشرة و اللحظية:**

العنونة الغير مباشرة تجعل سجلات المعالج كما لو كانت مؤشرات، كما هي ال Pointers في الباسكال. للدلالة على أن سجل يستخدم كمؤشر في لغة التجميع، نستحدم المعقوفات، ولقد قدمنا أمثلة كثيرة إستعملنا فيها العنونة الغير مباشرة. ففي الباسكال أو في اللغات العالية المستوى الأخرى مثل C و++C نستعمل مفهوم قيمة المتغيرات ومفهوم المؤشرات ومفهوم المرجع : variables Value و pointer و reference :

```
[code]
@L1:   mov   eax, MyVar        // eax:= MyVar العنونة المباشرة لقيمة المتغير في الذاكرة
@L1:   mov   eax, [MyVar]      // the same as first line
@L2:   mov   ebx, offset MyVar  // ebx:= @MyVar ( In C++  ebx= &MyVar)
@L3:   mov   edx,  [ebx]       // edx:= ebx^   ( In C++  edx= *ebx)  = Myvar
[/code]
```

في السطر الأول نقوم بقراءة مباشرة لقيمة من موضع في الذاكرة التي يرمز إليه إسم المتغير MyVar، و الكتابة MyVar أو [MyVar] لا تفرق في شيء فهي متكافئة و لها نفس المعنى.(حذف المعقوفات في هذه الصيغة يعد تساهل في كتابة لغة التجميع في syntax لبورلاند و ميكروسوفت، لكن لانجده في مجمعات أخرى كمجمع Nasm). هذه الطريقة في الوصول للذاكرة تسمى بالعنونة المباشرة direct adressing mode أو مايسمى أيضا ب absolute adressing.

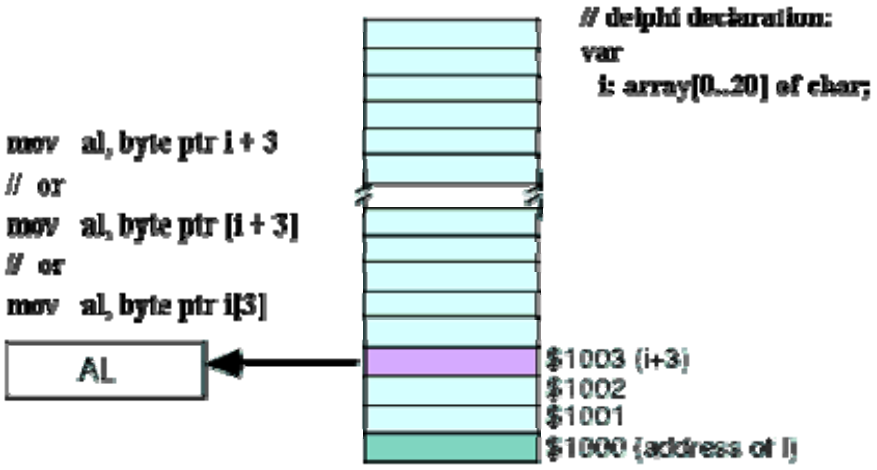
مثال عن كيفية الوصول للقيم بهذه العنونة موضحة في الصورة التالية:




في هذه الطريقة من العنونة يمكن ان نظيف إلى العنوان قيمة ثابتة أو نطرحها منه في نفس سطر التعليمة، فديلفي ستحسبها قبل التصريف بحيث سيتم قراءة عنوان في الذاكرة لكن بتنقل displacement نحو خانة أخرى في الذاكرة بنسبة القيمة المضافة على العنوان. ما يسمى أيضا displacement addressing mode و يكون على هذا النحو:

```
[code]
@L1:   mov   eax, MyVar ± disp        //  eax:=  (@MyVar ± disp)^
       //      Or
@L1:   mov   eax, [MyVar ± disp ]
       //      Or
@L1:   mov   eax, MyVar[ ± disp ]
 [/code]
```

تستعمل هذه الطريقة مثلا في الوصول لعناصر جدول أو موضع وسط متسلسلة ، فبمعرفة العنوان للمصفوفة يمكننا زيادة عدد البايتات اللازمة لكي نصل إلى عنصر من عناصر الجدول:




ملاحظة مهمة displacement تحسب دائما على أنها تنقل بالبايت وليس ب word او dword !!

في السطر الثاني من مثالنا،قمنا بقراءة عنوان الذاكرة نفسه و ليس القيمة المخزنة في ذلك الموضع. لهذا إستعملنا الموجهة directive التي سبق و أشرنا إليها في الفقرات السابقة و هي OffSet التي تعيد عنوان الذاكرة للمتغير و ليست قيمته، فهو يكافئ في الباسكال المعامل @ أو في C++ المعامل &.

فهذا العنوان ليس متغيرا من متغيرات التطبيق، بل له قيمة عددية ثابتة، كما لو قمنا بوضع قيمة ثابتة بسجل المعالج. هذه العنونة تسمى بالعنونة الللحظية immediate addressing أو ب Litteral adressing . ليس فيها أية ولوج للذاكرة، وهذه القيمة هي قيمة ثابتة سيقوم المصرف بتكويدها مباشرة مع التعليمة في كود لغة الآلة. فليس هناك أي تبادل مع الذاكرة.

في السطر الثالث نقوم بقراءة قيمة لموضع في الذاكرة، عنوان هذا الموضع يوجد في قيمة السجل ebx.

فلو قمنا بقراءة كيف ستقوم ديلفي بتكويد السطور الثلاثة بلغة التجميع عند تصريفها للتطبيق، حتة نستوعب ماذا يعني هذا المفهوم من العنونة:

```
[code]
   mov        eax,[$0045f5d4]    // Loads contents
   mov        ebx,$0045f5d4      // Loads Adress
   mov        edx,[ebx]          // Loads contents in adress stored in ebx
[/code]
```

العدد $0045f5d4 يمثل عنوان MyVar في الذاكرة. فوضعه بين معقوفتين في السطر الأول يفيد العنونة المباشرة Direct adressing (هنا ضروروي وضع المعقوفات عكس التساهل لو أننا إستعملنا معرف المتغير myvar) ، قيمة eax ستكون قيمة dword المخزنة في خانة بذاكرة عنوانها هو $0045f5d4.

أما في السطر الثاني فهو بدون معقوفات مما يفيد العنونة اللحظية اللحظية immediate adressing فهي قيمة ثابتة لحظية ليست مسجلة في مكان ما في الذاكرة، فقيمة ebx بعد تنفيذ التعليمة ستكون هي $0045f5d4. فديلفي سيقوم بتكويد هذا البارامتر الثابت مباشرة في كود لغة الآلة. فهو عدد ثابت لا يأخذ حيز من الذاكرة، أمثلة عليه:

```
[code]
        mov   eax, $5FAD0023
        mov   cx, 0f34ah
        mov   al, 64
[/code]
```

هذه الأرقام كلها بدون معقوفات تحيطها تفيد قيم ثابتة. لو وضعت بين معقوفات فالأمرسيحتلف لأنها ستفيد مواضع لعناوين بالذاكرة.

أما السطر الثالث فهنا نقوم بعملية قراءة لخانة بالذاكرة لكن بعملية غير مباشرة، حيث أن هذا العنوان لا يمرر في سطر تعليمة لغة التجميع ، بل هو مسجل في سجل ebx، فالسجل ebx وضع بين معقوفتين، أي أن السجل ebx يضم عنوان لمكان في الذاكرة، فالمعالج سيقوم أولا بقراءة ebx ليعرف قيمة العنوان، ثم يستعمل هذا العنوان لقراءة القيمة المسجلة بداخله، ليضعها في ما بعد بالسجل edx. طريق منعرجة و غير مستقيمية، لهذا يسمى هذا النوع من العنونة بالعنونة الغير مباشرة Indirect adressing.

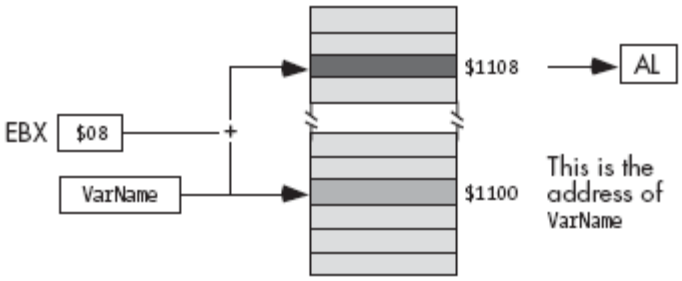
بصفة عامة فإن كل السجلات العامة من 32bits : eax,ebx,ecx,edx وكذلك سجلات التأشير Index register: وهما edi و esi ، تصلح لهذا النوع من العنونة الغير مباشرة وسجلات esp و ebp. نستثني جميع سجلات 8bits و 16bits فهي لا تستطيع العنونة بهذه الطريقة في العنونة 32bits .
ونذكر أن سجلي ebp و esp يستعملان سجل cs لمقطع المكدس Stack Segment. فيحين المسجلات الأخرى تستعمل سجل ds لمقطع المعطيات Code Segments.

**b) نظم العنونة المؤشرة و المفهرسة و ما يشتق منها:**

نظم العنونة التي سنستعرضها كلها تعد نظم عنونة مشتقة من نظام العنونة الغير مباشرة، فهي تطوير و إمتداد لتقنية الولوج لالذاكرة بإستعمال سجلات المعالج، فرغم تعدد صيغها (أكثر من 17 صيغة) و تشعبها إلا أنها تبقى سهلة الفهم و الحفظ إذا تمكنت من ظبط معنى التأشير الغير المباشر.
و يمكن تلخيص تسمياتها بالإنجليزية في:

- indexed addressing modes.
- addressing modes based/indexed.
- Displacement plus based/indexed addressing modes.

أبسط هذه الصيغ هي العنونة المؤشرة ما تسمى ب indexed addressing modes. هي طريقة من طرق الوصول للقيم مسجلة في الذاكرة، حيث نقوم بجمع قيمة لحظية مع قيمة مسجلة في سجل Reg32 من سجلات الحاسب للحصول على عنوان بالذاكرة التي سنقرأ منها القيمة المسجلة فيها.




وهي تقبل صيغ كثيرة في كتابتها ( الإختلاف في إستعمال المعقوفات) :

varName[reg32]
[reg32+varName]
[varName][reg32]
[varName+reg32]
[reg32][varName]
varName[reg32+const]
[reg32+varName+const]
[varName][reg32][const]
varName[const+reg32]
[const+reg32+varName]
[const][reg32][varName]

varName[reg32-const]
[reg32+varName-const]
[varName][reg32][-const]

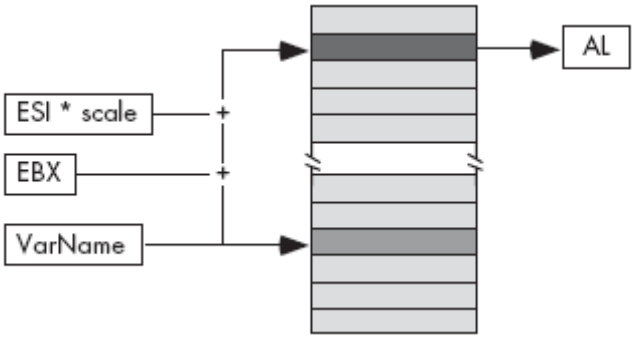
فالصيغ كلها تفيد أن العنوان الحقيقي effective adress الذي سنقرأ منه القيمة المسجلة به سيحسب بهذه المعادلة:

Effective adress := Var_adress+ Reg32 ± Const.

Var_adress تعتبر عنوان المتغير VarName في الذاكرة، و Reg32 هو سجل من سجلات المعالج العامة 32bits، و Const قيمة ثابتة.

```
[code]
        mov   eax, [ebx + 23]
        mov   cx, word ptr MyArr[eax -4]
        mov   edx, dword ptr [MyVar+eax + Type(Integer)]
[/code]
```

وهناك أيضا ما يسمى ب Scaled-Indexed Addressing، وهو يسمح بضرب قيمة السجل بعدد (scaling). هذا العدد يكون أحد هذه القيم فقط 1 أو 2 أو 4 أو 8.




ومن صيغها التي يمكن إستعمالها:

VarName[ IndexReg32 * scale ]
VarName[ IndexReg32 * scale + displacement ]
VarName[ IndexReg32 * scale - displacement ]
[ BaseReg32 + IndexReg32 * scale ]
[ BaseReg32 + IndexReg32 * scale + displacement ]
[ BaseReg32 + IndexReg32 * scale - displacement ]
VarName[ BaseReg32 + IndexReg32 * scale ]
VarName[ BaseReg32 + IndexReg32 * scale + displacement ]
VarName[ BaseReg32 + IndexReg32 * scale - displacement ]

فالصيغ كلها تفيد أن العنوان الحقيقي effective adress الذي سنقرأ منه القيمة المسجلة به سيحسب بهذه المعادلة:

Effective adress :=
Var_adress+ BaseReg32 + IndexReg32 * scale ± displacement.

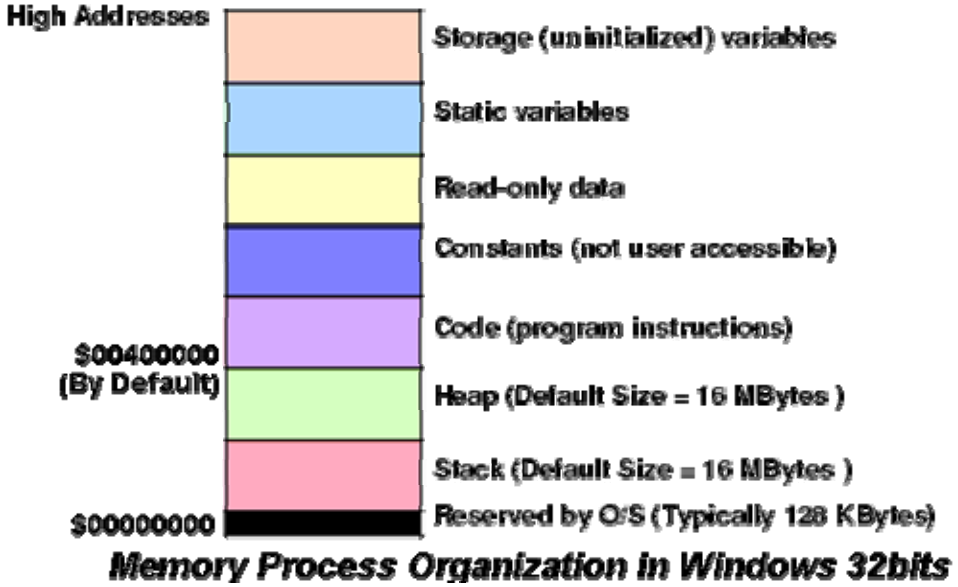
حيث Var_adress تمثل عنوان المتغير VarName في الذاكرة، BaseReg32 هو سجلات المعالج العامة ذات 32bits و هي (EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP)، و IndexReg32 هي السجلات العامة 32bits للمعالج ما عدا سجل esp، و OffSet_address هو عدد ثابت، أما scale فيكون أحد هذه القيم 1,2,4,8.

```
[code]
        mov   eax, MyArr[edx + ecx*4]
        mov   cx, word ptr MyArr[eax +esi*2 - 4]
        mov   edx, dword ptr [ebx+eax +12]
[/code]
```

---

## 2. المكدس the stack:

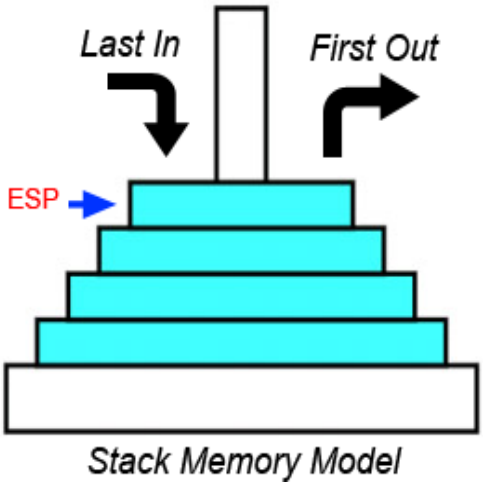
عندما يقوم نظام التشغيل بتنفيذ الملف التنفيذي لتطبيق Win32، فهو يقوم بإسقاط كود التطبيق التنفيذي في الذاكرة، موفرا للتطبيق جزءا من الذاكرة تحت تصرفه ليسجل بها ما يريده من البيانات. هذا الجزء من الذاكرة يسمى بالمكدس stack، فهو جزء من الذاكرة أثناء التنفيذ مخصص يحجزه النظام للتطبيق كمساحة خاصة به لتسجيل بياناته فيها. stack جزء من الأجزاء الأساسية لكل تطبيق يسقط في الذاكرة الإفتراضية لويندوز، كباقي الأجزاء الأخرى مثل جزء الكود و جزء data.



*Memory Process Organization in Windows 32bits*

يعد stack حيز من الذاكرة يتميز بطريقة خاصة للولوج إليه مخالفة لباقي الأجزاء الأخرى. عندما نسجل مجموعة من البينات بالذاكرة تسلسليا إنطلاقا من عنوان ما في الذاكرة فإن البيانات ستسجل متتابعة ، إبتداءا من العنوان البدئي و صعودا في خانات الذاكرة. لكن المكدس يعمل بطريقة أخرى، فالبيانات تسجل إنطلاقا من عنوان مؤشر بالسجل esp ( عنونة غير مباشرة) ، تسلسليا لكن نزولا في خانات الذاكرة، هذا مما يميزها عن باقي أجزاء الذاكرة المخصصة للتطبيق.

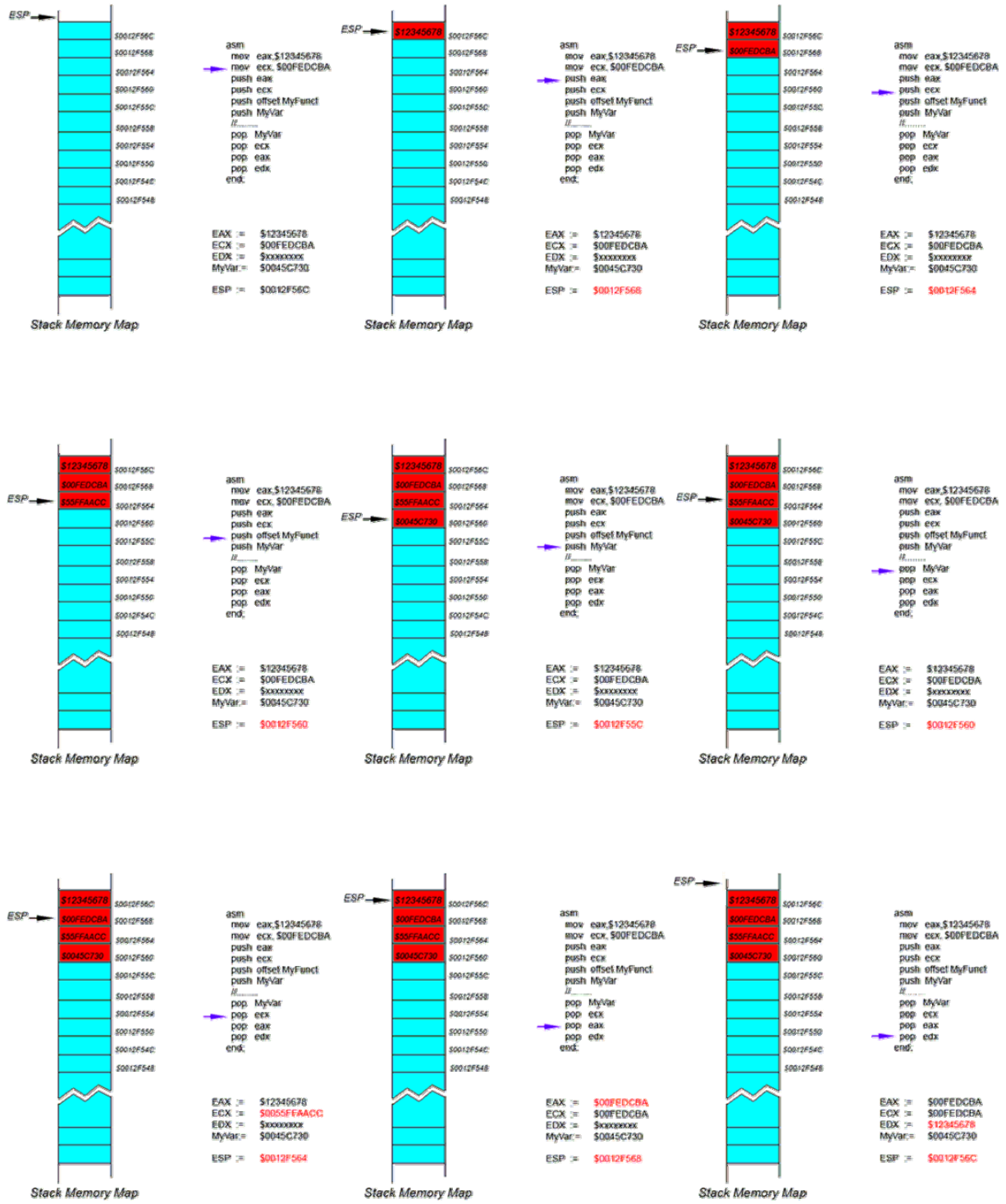
هذه الطريقة في التعامل مع المعطيات في المكدس تسمى ب" الأول دخولا، الأخر خروجا" أو LIFO أي « last in first out ». نفس فكرة لغز أعمدة هانوي Towers of Hanoi puzzle.




يوفر المعالج مجموعة من التعليمات للتعامل مع المكدس، أهم هذه التعليمات هي الزوجين Call/Ret و Push/Pop. بالإضافة إلى السجلين esp و ebp. السجل esp يؤشر دوما إلى أعلى المكدس، هذا السجل من عرض 32bits تنقص قيمته كلما تمت إضافة قيمة للمكدس، وتزداد إذا تم إستخراجها منه.
لإدخال قيمة للمكدس نسنعمل تعليمة push، التي تقوم بنسخ المعامل نحو العنوان الذي يشير إليه السجل esp، ثم تقوم بنقص قيمة esp ليشير للعنوان الموالي مباشرة.
التعليمة المقابلة لها هي pop التي تعمل العكس تماما. رغم أن التعليمتين يمكنه أن تتعامل مع معاملات من سعة word و dword و لا يمكنها التعامل مع byte، يفضل إستعمال معاملات من سعة dword. المعاملات يمكنها أن تكون قيم ثابتة أو عناوين في الذاكرة أو سجلات المعالج :

```
[code]
asm
  mov   eax,$12345678
  mov   ecx, $00FEDCBA
  push  eax              //  [esp] <- eax ,And esp:= esp + 4
  push  ecx              //  [esp] <- ecx ,And esp:= esp + 4
  push  offset MyFunct   //  [esp] <- @MyFunct ,And esp:= esp + 4
  push  MyVar            //  [esp] <- MyVar ,And esp:= esp + 4
  //........
  pop   MyVar            //  [MyVar] <- [esp] and esp:= esp - 4
  pop   ecx              //  ecx <- [esp] and esp:= esp - 4
  pop   eax              //  eax <- [esp] and esp:= esp - 4
  pop   edx              //  edx <- [esp] and esp:= esp - 4
end;
[/code]
```

هذه الرسوم تبين التسلسل الزمني لمنطقة المكدس أثناء تنفيذ الكود أعلاه في ديلفي:




نرى كيف أن esp كيف أنها تشير دائما لأعلى خانة في المكدس، وكذلك كيف أن أول كلمة dword إدخالا ستكون آخرها إخراجا، وهذا هو مفهوم LIFO. تعليمة Push تقوم بتخزين dword في العنوان المؤشر ب esp ثم بعد ذلك تقوم بعملية حسابية esp := esp -4. (-4) لأن عرض dword هو 4 بايت. لو خزنا قيمة بعرض word ف إن قيمة esp بعد تعليمة Push ستكون esp := esp -2.

تعليمة Pop تقوم بعكس عملية push، تقوم أولا بنسخ القيمة من العنوان المؤشر ب esp ثم بعد ذلك تقوم بالعملية esp := esp +4.
تعليمتي Push و Pop تستعمل كثيرا لحفظ قيم السجلات بتعليمة Pop قبل التعديل عليها ثم نستعيد فيما بعد قيمها البدئية بواسطة Pop.

و من أهم إستعمالاتها أيضا هو تخزين مجموعة من البارامترات لتمريرها للدوال والإجرائيات قبل إستدعائها. وسيكون هذا هو موضوع الفقرات القادمة حول كيفية تمرير المعاملات عبر المكدس للإجرائيات.

من التعليمات الأساسية التي تستخدم المكدس إضافة لتعليمتي Push و Pop هي تعليمتي Call و Ret. تعليمة Call هي تعليمة تقوم بتخزين عنوان التعليمة التي تليها في المكدس وتحيين قيمة esp بطرح 4 منها، و تقوم بالقفز للعنوان الممر لهذه التعليمة. أما ret فيقوم بعملية إستخراج dword من المكدس ثم يقفز إليه مباشرة على أنه عنوان في الكود، وتحيين قيمة esp بإضافة 4 لها.

تعليمتي call وret تعد التعليمتين الأساسايتين لتحرير الدوال. فلإستدعاء دالة يكفي تمرير عنوانها لتعليمة call، لتقومة هذه التعليمة أوتوماتيكيا بحفظ العنوان الذي يليها في المكدس، مما سيسهل عملية العودة لنفس المكان التي أستدعيت منه الدالة بعد إنتهائها بتعليمة Ret.

```
[code]
   //........
   push  $06
   push  $05
   push  ecx
   call  MyFunction  // call the function Myfunction(ecx,$05,$06)
   //........
[/code]
```

يجب أن تتوخى الحذر في إستعمال Push و Pop عند البرمجة للدوال، فالمكدس كما ذكرنا هو من نوع Lifo.

```
[code]
   function MyFunction(A,B: integer): integer;
   asm                          // the adress of return is in the top of Stack
     mov   eax,a
     add   eax,b
     mov   @result,eax
     push  eax                  // eax is pushed in the top of Stack
     mov   eax,1
   end;                         // error : the fuction return to [EAX] and not to the Call Line
[/code]
```

الدالة لن ترجع للمكان الذي إستدعيت منه لأنها سترجع لأخر كلمة مكدسة في Stack و هي محتوى eax.

### 3. الكتابة الإملائية لدالة بلغة التجميع:

الكتابة الإملائية لدالة أو إجرائية بلغة التجميع المضمن في ديلفي تكون بجعل جسم الدالة كله مكتوب بلغة التجميع، ونبتدئ كتابة الدالة بالكلمة المفتاحية "asm" عوض " begin" و نختمها ب " ; end".

```
[code]
function Foo(MyVar: Type): Type;
var
  LocalsVar: type;
const
  LocalConst= value;
asm
    asm instructions only
    ....
    ....
    ....
end;
[/code]
```

ما بين asm/end يجب أن لا نجد سوى تعليمات لغة التجميع فقط. الكلمة المفتاحية assembler التي كانت تكتب بعد إسم الدالة عند تعريفها على أنها دالة ستكتب بلغة التجميع

```
[code]
function Foo(MyVar: Type): Type; assembler ;
asm
    ....
end;
[/code]
```

لم يعد له فعل و لا يستعمل إلا للتوافقية مع إصدارات ديلفي الأولى. فيكفي أن يكون جسم الدالة مكون من asm/end عوض begin/end لكي يتعرف ديلفي أن هذه الدالة هي دالة مكتوبة كلها بلغة التجميع.هذه الكتابة بلغة التجميع سيترتب عليها أشياء مخفية سيقوم بها ديلفي عند التصريف لتطبيق. فتعويض begin ب asm ، يقوم ديلفي بحذف مجموعة من تعليمات الآلة التي تخص نسخ المتغيرات الممررة بقيمتها by value التي يتعدى عرضها 32bits.

عند كتابة دالة أو إجرائية في ديلفي فإن ديلفي يقوم تلقائيا بتكويد بداية الدالة و إنتهائها و وكذلك كل التفاصيل المتعلقة بتمرير البارامترات لهذه الدالة ، فديلفي تجهز المكدس للإستقبال البارامترات الممرة للدالة و كذالك المتغيرات المحلية Local Variables، فكل من asm أو end هي عبارة عن محموعة من تعليمات لغة التجميع التي تتولى مهمة إطار المكدس. هذا الإطار سيكون محسنا وفق حاجيات الدالة فقط و لن تستعمل فيه إلا التعليمات الضرورية. لننظر للدالة التالية كيف ستكود بواسطة ديلفي:

```
[code]
function MyFunction(A,B: integer): integer; stdcall;
var
  // local variable with 12 bytes ($0C) bytes
```

```
  Lcl_Var1,Lcl_Var2,Lcl_Var3: integer;
asm
  mov   eax,a
  add   eax,b
  lea edx, lcl_var1
  lea ecx, lcl_var2
  lea esi, lcl_var3
end;
[/code]
```

بعد التكويد ستصبح:

```
MainUnit.pas.265: asm
004587A0 55          push ebp
004587A1 8BEC                mov ebp,esp
004587A3 83C4F4              add esp,-$0c
MainUnit.pas.266: mov   eax,a
004587A6 8B4508              mov eax,[ebp+$08]
MainUnit.pas.267: add   eax,b
004587A9 03450C              add eax,[ebp+$0c]
MainUnit.pas.269: lea edx, lcl_var1
004587AC 8D55FC              lea edx,[ebp-$04]
MainUnit.pas.270: lea ecx, lcl_var2
004587AF 8D4DF8              lea ecx,[ebp-$08]
MainUnit.pas.271: lea esi, lcl_var3
004587B2 8D75F4              lea esi,[ebp-$0c]
MainUnit.pas.272: end;
004587B5 8BE5                mov esp,ebp
004587B7 5D          pop ebp
004587B8 C20800              ret $0008
```

asm تم تكويدها ب:

```
[code]
        push ebp
        mov ebp,esp         // save the esp in ebp
        add esp,-$0c        // reserve memory for local variable in stack
                            // in bytes
 [/code]
```

و end تكافئ:

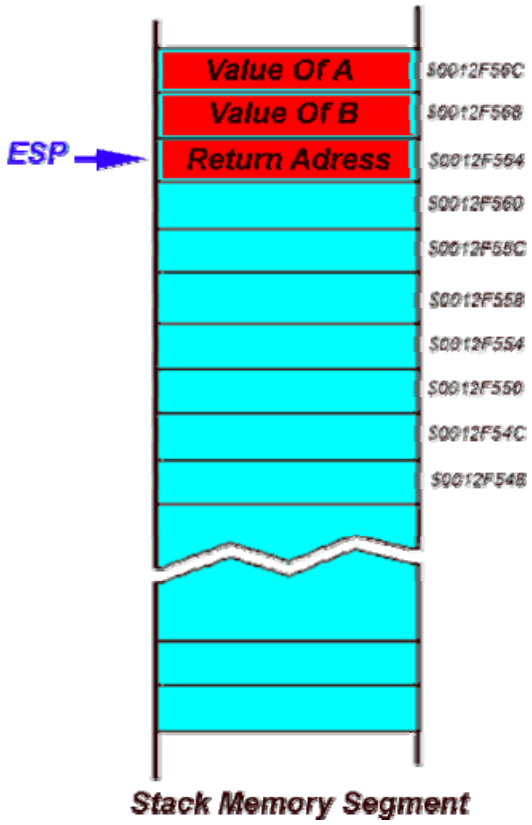
```
[code]
        mov esp,ebp
        pop ebp             // restore the old value of ebp
        ret $0008           // initialize esp to pointe to the returne adress
                            // with the size of parameters in bytes
 [/code]
```

سنلحظ كيف تم إستعمال المكدس لإستقبال البارامترات، و كذلك بحجز مكان في المكدس للمتغيرات المحلية المعرفة داخل كود الدالة.لماذا هذه التعليمات الإضافية؟ وما وظيفتها؟

عند دخول البرنامج لأول تعليمة للدالة ، فإن سجل المكسesp سيؤشر لعنوان العودة، و قبله سنجد قيم البارامترات التي تم تمريرها عبر المكدس عند إستدعاء الدالة :

```
[code]
   //........
  push  $06              //  value of A
  push  $05              //  value of B
  call  MyFunction       // call the function Myfunction(A,B)
   //........
[/code]
```

فمباشرة بعد إستدعاء الدالة سيكون لدينا في المكدس:

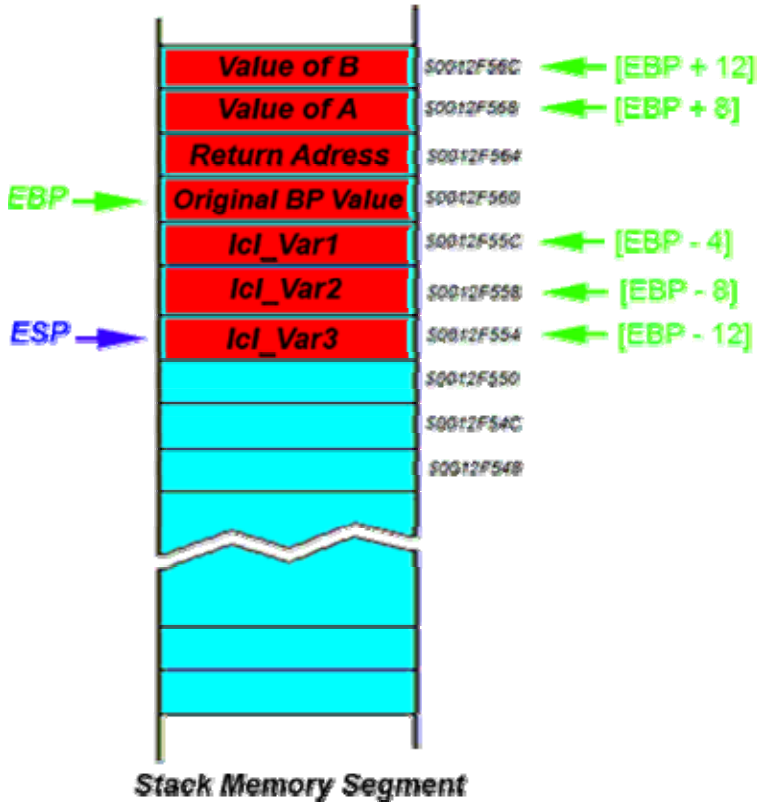



الدالة تستعمل متغيرات محلية وهي Lcl_Var1 و Lcl_Var2 و Lcl_Var3، و هي متغيرات من حجم 4 بايت لكل واحدة اي ما مجموعه 12 بايت. بطبيعة الحال أفضل مكان لحجز متغيرات مؤقتة هو المكدس، لكن لو حجزنا 12bytes فإن esp ستتغير قيمتها و البارامترين A و B سيتغير المؤشر لهما في المكدس ،ناهيك لو أننا احتجنا حفظ بعض القيم في المكدس، كإستدعاء دالة أخرى في نفس كود الدالة. هذا سيجعل البارامترات الممرة للدالة و كذلك المتغيرات تأخذ أماكن مختلفة نسبة لأعلى المكدس، و سيصعب تأشيرها ب esp لأن esp تتغير كلما إستخدمنا pop أو push.

لهذا فالمعالج يوفر مسجل آخر يستعمل كإستعمال esp للعنونة الغير مباشرة نحو خانات المكدس، وهو السجل ebp مثله مثل esp يشيران لمنطقة stack segment. ebp هو سجل مخصص لتأشير المعاملات و المتغيرات المحلية للدوال و هذا هو دوره الأساسي منذ بدايات البرمجة في real mode و protected، رغم أن في virtual mode التي نبرمج فيها بواسطة ديلفي قد فقد هذا الإختصاص بما أن كل من سجلات الأقسام segment لكل من data و الكود متساوية: ss و ds.

فعند إستعمال المكدس لتمرير البارامترات وكذلك للمتغيرات المحلية، نستعمل السجل ebp للتأشير لخاناتها في المكدس، ونترك esp تتغير وفق سير البرنامج دون أن نكترث لقيمتها. فأول شيء نقوم به هو حفظ قيمة ebp في المكدس، ثم نسخ قيمة esp في ebp لكي لا نضيع نسبية البارامترات و المتغيرات في المكدس. وهذا هي ماهية إضافة تعليمات في أول الدالة من طرف ديلفي:

```
[code]
        push ebp        // save ebp in the stack
        mov ebp,esp     // save the esp in ebp
        add esp,-$0c    // reserve memory for local variable in stack
                        // in bytes
[/code]
```




قيمة السجل ebp ستبقى ثابتة خلال تسلسل تعليمات الدالة، وبهذا ستبقى نسبية المتغيرات و البارامترات ثابتة نسبة لebp. وسنستعمب مثلا العنونة الغير مباشرة [ebp+8] للبارامتر A، و [ebp – 12] للمتغير المحلي lcl_Var3.

إنتهى الجزء الثالث بحمد الله، و إنشاء الله نستمر في الجزء الرابع

أخوكم مراد: Ikossan@gmail.com
الفريق العربي للبرمجة

www.arabteam2000-forum.com