Defining and Using Procedures تحديد واستخدام الإجراءات

A complicated problem is usually divided into separate tasks in order to be understood, implemented, and tested effectively. Thus, the programmer can divides a program into *subroutines*. In assembly language, the term *procedure* means a *subroutine*. In other languages, *subroutine* is called *method* or *function*.

عادة ما يتم تقسيم المشكلة المعقدة إلى مهام منفصلة من أجل فهمها وتنفيذها واختبارها بشكل فعال. وبالتالي ، يمكن للمبرمج تقسيم البرنامج إلى إجراءات فرعية. في لغة التجميع ، يعني مصطلح الإجراء روتينًا فرعيًا. في لغات أخرى ، يسمى الروتين الفرعي الطريقة أو الوظيفة.

1. PROC Directive. توجيه PROC

Defining a Procedure تحديد الإجراء

The programmer can define a *procedure* as a named block of statements that ends in a return statement.

يمكن للمبرمج تعريف الإجراء على أنه كتلة مسماة من العبارات التي تنتهي ببيان عودة.

A procedure is declared using the **PROC** and **ENDP** directives. It must be assigned a name (a valid identifier).

يتم الإعلان عن إجراء باستخدام توجيهات PROC و ENDP. يجب تعيين اسم لها (معرف صالح).

```
main PROC
.
.
main ENDP
```

When the programmer create a procedure other than the program's startup procedure, end it with a **RET** instruction. **RET** forces the CPU to return to the location from where the procedure was called:

عندما ينشئ المبرمج إجراءً بخلاف إجراء بدء تشغيل البرنامج ، قم بإنهائه بتعليمات RET. يجبر RET وحدة المعالجة المركزية على العودة إلى الموقع الذي تم استدعاء الإجراء منه:

```
sample PROC
.
.
ret
sample ENDP
```

Labels in Procedures

By default, labels are visible only within the procedure in which they are declared. This rule often affects jump and loop instructions. In the following example, the label named **Destination** must be located in the same procedure as the JMP instruction:

بشكل افتراضي ، تكون التسميات مرئية فقط ضمن الإجراء الذي تم الإعلان عنه فيه. تؤثر هذه القاعدة غالبًا على تعليمات القفز والتكرار. في المثال التالي ، يجب وضع التسمية المسماة الوجهة في نفس الإجراء مثل تعليمات JMP:

```
jmp Destination
```

It is possible to work around this limitation by declaring a global label, identified by a double colon (::) after its name:

من الممكن التغلب على هذا القيد من خلال إعلان تسمية عمومية ، معرَّفة بنقطتين (: :) بعد اسمها:

```
Destination::
```

In terms of program design, it's not a good idea to jump or loop outside of the current procedure. Procedures have an automated way of returning and adjusting the runtime stack. If the programmer directly transfers out of a procedure, the **runtime stack** can easily become corrupted.

فيما يتعلق بتصميم البرنامج ، ليس من الجيد القفز أو التكرار خارج الإجراء الحالي. الإجراءات لديها طريقة آلية للعودة وتعديل مكدس وقت التشغيل. إذا قام المبرمج بالتحويل مباشرة من أحد الإجراءات ، فيمكن أن يتلف مكدس وقت التشغيل بسهولة.

2. CALL and RET Instructions تعليمات

The **CALL** instruction calls a procedure by directing the processor to begin execution at a new memory location. The procedure uses a **RET** (return from procedure) instruction to bring the processor back to the point in the program where the procedure was called.

From the computer architecture view point, the **CALL** instruction pushes its return address on the stack and copies the called procedure's address into the **I**nstruction **P**ointer. When the procedure is ready to return, its **RET** instruction pops the return address from the stack into the **I**nstruction **P**ointer. In 32-bit mode, the CPU executes the instruction in memory pointed to by **EIP** (instruction pointer register). In 16-bit mode, **IP** points to the instruction.

تستدعي تعليمات CALL إجراءً عن طريق توجيه المعالج لبدء التنفيذ في موقع ذاكرة جديد. يستخدم الإجراء تعليمات RET (العودة من الإجراء) لإعادة المعالج إلى النقطة في البرنامج حيث تم استدعاء الإجراء.من وجهة نظر بنية الكمبيوتر ، تقوم تعليمات CALL بدفع عنوان المرسل الخاص بها على المكدس ونسخ عنوان الإجراء المطلوب في مؤشر التعليمات. عندما يكون الإجراء جاهزًا للعودة ، فإن تعليمات RET الخاصة به تظهر عنوان المرسل من المكدس إلى مؤشر التعليمات. في وضع 32 بت ، تنفذ وحدة المعالجة المركزية التعليمات في الذاكرة المشار إليها بواسطة EIP (سجل مؤشر التعليمات). في وضع 16 بت ، يشير IP إلى التعليمات.

Call and Return Example

Suppose that in main, a **CALL** statement is located at offset **00000020**. Typically, this instruction requires **5** bytes of machine code, so the next statement (a MOV in this case) is located at offset **00000025**:

مثال على الاتصال والعودة

لنفترض أنه في الأساس ، توجد عبارة CALL في offset 00000020. عادةً ما تتطلب هذه التعليمات 5 بايت من كود الآلة ، لذا فإن العبارة التالية (MOV في هذه الحالة) موجودة في offset 00000025:

```
main PROC
00000020 call MySub
00000025 mov eax,ebx
```

Next, suppose that the first executable instruction in **MySub** is located at offset **00000040**:

بعد ذلك ، افترض أن أول أمر قابل للتنفيذ في MySub يقع في الإزاحة 00000040:

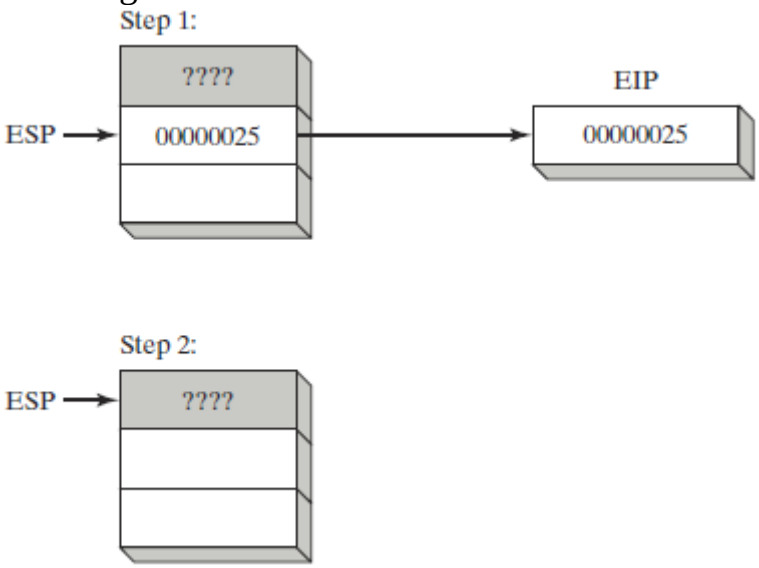
```
MySub PROC
00000040 mov eax,edx
.
.
ret
MySub ENDP
```

When the **CALL** instruction executes (Figure 5), the address following the call (**00000025**) is pushed on the stack and the address of **MySub** is loaded into **EIP**. All instructions in **MySub** execute up to it's **RET** instruction. When the **RET** instruction executes, the value in the stack pointed to by **ESP** is popped into **EIP** (step 1 in Figure 6). In step 2, **ESP** is incremented so it points to the previous value on the stack (step 2).

عند تنفيذ تعليمات CALL (الشكل 5) ، يتم دفع العنوان الذي يلي المكالمة (00000025) على المكدس ويتم تحميل عنوان MySub في EIP. جميع التعليمات الموجودة في MySub تنفذ حتى تعليمات RET الخاصة بها. عند تنفيذ تعليمات RET ، تظهر القيمة الموجودة في المكدس المشار إليها بواسطة ESP في EIP (الخطوة 1 في الشكل 6). في الخطوة 2 ، تتم زيادة ESP بحيث يشير إلى القيمة السابقة على المكدس (الخطوة 2).

Figure 5: Executing a CALL Instruction.

Figure 6: Executing the RET Instruction.



Nested Procedure Calls

A nested procedure call occurs when a called procedure calls another procedure before the first procedure returns. Suppose that main calls a procedure named **Sub1**. While **Sub1** is executing, it calls the **Sub2** procedure. While **Sub2** is executing, it calls the **Sub3** procedure. The process is shown in Figure 7.

When the **RET** instruction at the end of **Sub3** executes, it pops the value at **stack[ESP]** into the **I**nstruction **P**ointer. This causes execution to resume at the instruction following the call **Sub3** instruction. The following diagram shows the stack just before the return from **Sub3** is executed:

استدعاءات الإجراءات المتداخلة

يحدث استدعاء إجراء متداخل عندما يستدعي إجراء يسمى إجراء آخر قبل عودة الإجراء الأول. افترض أن الرئيسي يستدعي إجراء يسمى Sub1. أثناء تنفيذ Sub1 ، فإنه يستدعي الإجراء Sub2. أثناء تنفيذ Sub2 ، فإنه يستدعي الإجراء Sub3. العملية موضحة في الشكل 7.

عندما يتم تنفيذ تعليمات RET في نهاية Sub3 ، فإنها تنبثق القيمة الموجودة في المكدس [ESP] في مؤشر التعليمات. يؤدي هذا إلى استئناف التنفيذ عند التعليمات التالية لتعليمات استدعاء Sub3. يوضح الرسم البياني التالي المكدس قبل تنفيذ الإرجاع من Sub3:

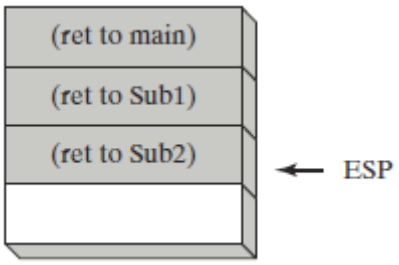
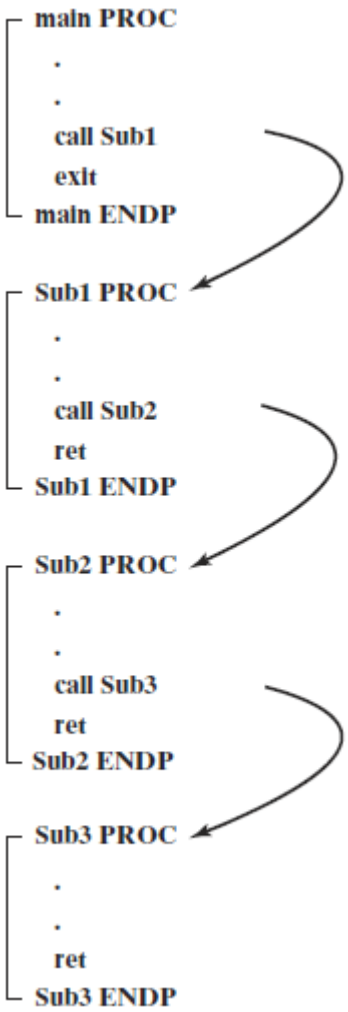
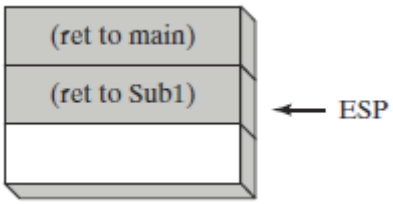


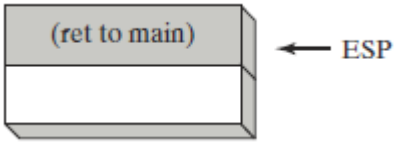
Figure 7: Nested Procedure Calls.



After the return, **ESP** points to the next-highest stack entry. When the **RET** instruction at the end of **Sub2** is about to execute, the stack appears as follows:
بعد الإرجاع ، يشير ESP إلى ثاني أعلى إدخال للمكدس. عندما تكون تعليمات RET في نهاية Sub2 على وشك التنفيذ ، تظهر المكدس على النحو التالي:



When the **RET** instruction at the end of **Sub1** is about to execute the stack appears as follows:
عندما تكون تعليمات RET في نهاية Sub1 على وشك تنفيذ المكدس تظهر على النحو التالي:



Finally when the sub1 ends, **stack[ESP]** is popped into the **I**nstruction **P**ointer, and execution resumes in main:
أخيرًا ، عندما ينتهي sub1 ، ينبثق المكدس [ESP] في مؤشر التعليمات ، ويستأنف التنفيذ بشكل رئيسي:

Passing Register Arguments to Procedures

If the programmer writes a procedure that performs some standard operation such as calculating the sum of an integer array, it's not a good idea to include references to specific variable names inside the procedure. If the programmer did, the procedure could only be used with one array. *A better approach is to pass the offset of an array to the procedure and pass an integer specifying the number of array elements.* We call these *arguments* (or *input parameters*). In assembly language, it is common to pass arguments inside general-purpose registers.

تمرير الحجج سجل إلى الإجراءات
إذا كتب المبرمج إجراءً يؤدي بعض العمليات القياسية مثل حساب مجموع مصفوفة عدد صحيح ، فليس من الجيد تضمين مراجع لأسماء متغيرات معينة داخل الإجراء. إذا قام المبرمج بذلك ، فلا يمكن استخدام الإجراء إلا مع مصفوفة واحدة. أفضل طريقة هي تمرير إزاحة المصفوفة إلى الإجراء وتمرير عدد صحيح يحدد عدد عناصر المصفوفة. نسمي هذه الوسائط (أو معلمات الإدخال). في لغة التجميع ، من الشائع تمرير الحجج داخل سجلات الأغراض العامة.

```
.data
    theSum DWORD ?
.code
main PROC
    mov eax,10000h          ; argument
    mov ebx,20000h          ; argument
    mov ecx,30000h          ; argument
    call Sumof              ; EAX = (EAX + EBX + ECX)
    mov theSum,eax          ; save the sum
SumOf PROC
    add eax,ebx
    add eax,ecx
    ret
SumOf ENDP
```

After the **CALL** statement, the programmer can copy the sum in **EAX** to a variable.

بعد عبارة CALL ، يمكن للمبرمج نسخ المجموع في EAX إلى متغير.

3. Saving and Restoring Registers

The programmer can push the registers on the stack at the beginning of the procedure and popped at the end. This action is typical of most procedures that modify registers in order to be sure that none of its own register values will be overwritten. However, the previous action is not suitable if the programmer need to return value in some registers.

3. حفظ واستعادة السجلات

يمكن للمبرمج دفع السجلات على المكدس في بداية الإجراء وظهرت في النهاية. يعتبر هذا الإجراء نموذجيًا لمعظم الإجراءات التي تعدل السجلات للتأكد من عدم الكتابة فوق أي من قيم التسجيل الخاصة بها. ومع ذلك ، فإن الإجراء السابق غير مناسب إذا احتاج المبرمج إلى إرجاع القيمة في بعض السجلات.

USES Operator

The **USES** operator, coupled with the **PROC** directive, lets the programmer list the names of all registers modified within a procedure. **USES** tells the assembler to do two things: *First*, generate **PUSH** instructions that save the registers on the stack at the beginning of the procedure. *Second*, generate **POP** instructions that restore the register values at the end of the procedure. The **USES** operator immediately follows **PROC**, and is itself followed by a list of registers on the same line separated by spaces or tabs (not commas).

مشغل الاستخدامات

يتيح عامل التشغيل USES ، إلى جانب توجيه PROC ، للمبرمج سرد أسماء جميع السجلات المعدلة في الإجراء. يطلب USES من المجمع القيام بأمرين: أولاً ، إنشاء تعليمات PUSH التي تحفظ السجلات على

المكدس في بداية الإجراء. ثانيًا ، قم بإنشاء إرشادات POP التي تستعيد قيم التسجيل في نهاية الإجراء. يتبع عامل التشغيل USES PROC على الفور ، ويتبعه هو نفسه قائمة من السجلات على نفس السطر مفصولة بمسافات أو علامات تبويب (وليس بفاصلات).

```
ArraySum PROC USES esi ecx
   mov eax,0                    ; set the sum to zero
L1:
   add eax,[esi]                ; add each integer to sum
   add esi,TYPE DWORD           ; point to next integer
   loop L1                      ; repeat for array size
   ret                          ; sum is in EAX
ArraySum ENDP
```

The corresponding code generated by the assembler shows the effect of USES:

```
ArraySum PROC
   push esi
   push ecx
   mov eax,0                    ; set the sum to zero
L1:
   add eax,[esi]                ; add each integer to sum
   add esi,TYPE DWORD           ; point to next integer
   loop L1                      ; repeat for array size
   pop ecx
   pop esi
   ret
ArraySum ENDP
```

H.W. Write a main procedure that uses to read array elements and pass it to ArraySum procedure.

هـ. اكتب إجراءً رئيسيًا يستخدم لقراءة عناصر المصفوفة ومرره إلى إجراء ArraySum.