Procédures et fonctions Chapitre 11

La conception modulaire est l'une des pierres angulaires de la programmation structurée. Un programme modulaire contient des blocs de code avec des points d'entrée et de sortie uniques. Vous pouvez *réutiliser* les sections bien écrites du code dans d'autres programmes ou dans d'autres sections d'un programme existant. Si vous réutilisez un segment de code existant, vous n'avez besoin ni de concevoir, ni de coder, ni de débuguer cette section de code puisque (vraisemblablement) vous l'avez déjà fait. Étant donné les coûts croissants de développement de logiciel, la conception modulaire deviendra de plus en plus importante avec le temps.

L'unité de base d'un programme modulaire est le module. Les modules ont différentes significations pour des personnes différentes, ici, vous pouvez supposer que les termes module, sous-programme, routine, unité de programme, procédure, et fonction sont tous synonymes.

La procédure est la base d'un style de programmation. Les langues procédurales incluent le Pascal, le BASIC, le C++, le Fortran, le PL/I, et l'Algol. Des exemples des langues non procédurales incluent l'APL, le LISP, le SNOBOL4 ICON, le Forth, le SETL, le PROLOG et d'autres qui sont basés sur d'autre constructions de programmation comme l'abstraction fonctionnelle ou la recherche de modèles. Le langage assembleur est capable d'agir comme un langage procédural ou non procédural. Puisque vous êtes probablement beaucoup plus familiarisé avec le paradigme de la programmation procédurale, ce texte s'attachera à simuler des constructions procédurales en langage assembleur 80x86.

11.0 Vue d'ensemble du chapitre

Ce chapitre présente une introduction aux procédures et aux fonctions en assembleur. Il discute des principes de base, du passage de paramètres, des résultats de fonctions, des variables locales, et de la récursion. Vous emploierez la plupart des techniques que ce chapitre discute dans des programmes classiques en assembleur. La discussion des procédures et des fonctions se poursuit dans le chapitre suivant; ce chapitre discute de techniques avançées que vous n'emploierez pas généralement dans des programmes en assembleur. Les sections ci-dessous qui ont "●" comme préfixe sont essentielles. Les sections avec un "❑" discutent de sujets avançés que vous pouvez mettre de côté pendant un moment

- ● Procédures.
- ❑ Procédures *near* et *far.*
- ● Fonctions.
- ● Sauvegarder l'état de la machine.
- ● Paramètres.
- ● Passer des paramètres par valeur.
- ● Passer des paramètres par référence.
- ❑ Passer des paramètres par valeur-retournée.
- ❑ Passer des paramètres par résultat.
- ❑ Passer des paramètres par nom.
- ● Passer des paramètres dans des registres
- ● Passer des paramètres dans des variables globales
- ● Passer des paramètres sur la pile
- ● Passer des paramètres dans le flux de code.
- ❑ Passer des paramètres via un bloc de paramètre
- ● Résultats de fonction
- ● Retourner des résultats de fonction dans un registre
- ● Retourner des résultats de fonction sur la pile
- ● Retourner des résultats de fonction dans des emplacements de mémoire
- ● Effets secondaires.
- ❑ Stockage de variables locales.
- ❑ Récursion.

11.1 Expressions arithmétiques

Dans un environnement procédural, l'unité de base du code est la *procédure*. Une procédure est un ensemble d'instructions qui calculent une certaine valeur ou exécutent une certaine action (telle qu'imprimer ou lire une valeur de caractère). La définition d'une procédure est très semblable à la définition d'un *algorithme*. Une procédure est un ensemble de règles à suivre qui, si elles sont concluantes, produisent un certain résultat. Un algorithme est également une semblable séquence, mais un algorithme est garanti se terminer tandis qu'une procédure n'offre aucune garantie de cet ordre.

La plupart des langages de programmation procéduraux mettent en application des procédures en utilisant le mécanisme d'appel/retour (call/return). C'est-à-dire, un certain code appelle une procédure, la procédure fait ce qu'elle doit faire, et alors la procédure revient à l'appelant. Les instructions d'appel et de retour fournissent le *mécanisme d'invocation de procédures* du 80x86. Le code appelant appelle une procédure avec l'instruction call, la procédure retourne à l'appelant avec l'instruction ret. Par exemple, l'instruction 80x86 suivante appelle la routine de l'UCR Standard Library sl_putcr[111111]:

```
        call    sl_putcr
```

sl_putcr imprime une séquence retour de chariot /passage à la ligne sur l'affichage videol et retourne le contrôle à l'instruction juste après l'instruction call sl_putcr.

Hélas, la bibliothèque standard UCR ne fournit pas toutes les routines dont vous aurez besoin. La plupart du de temps, vous devrez écrire vos propres procédures. Une procédure simple peut ne se composer de rien d'autre qu'une séquence d'instructions finissant avec une instruction de retour. Par exemple, la "procédure" suivante met à zéro les 256 bytes commençant à l'adresse dans le registre bx:

```
ZeroBytes:      xor     ax, ax
                mov     cx, 128
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret
```

En chargeant le registre bx avec l'adresse d'un bloc de 256 bytes et en lançant une instruction call ZeroBytes, vous pouvez mettre à zéro le bloc indiqué.

En règle générale, vous n'allez pas définir vos propres procédures de cette manière. Au lieu de cela, vous devriez employer les directives assembleur proc et endp de MASM. La routine de ZeroBytes, en utilisant les directives proc et endp, est

```
ZeroBytes                               proc
                xor     ax, ax
                mov     cx, 128
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret
ZeroBytes       endp
```

Gardez à l'esprit que proc et endp sont des directives assembleur. Elles ne produisent aucun code. Elles sont simplement un mécanisme qui vous aide à rendre vos programmes plus faciles à lire. Pour le 80x86, les deux derniers exemples sont identiques; cependant, pour un être humain, le dernier est clairement une procédure d'un seul bloc, l'autre pourrait simplement être un ensemble quelconque d'instructions dans une autre procédure. Considérez maintenant le code suivant:

```
ZeroBytes:      xor     ax, ax
                jcxz    DoFFs
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret
```

[1] Normalement vous devriez utiliser la macro putcr pour accomplir ceci, mais cette instruction call fera la même chose.

```
DoFFs:  mov     cx, 128
                mov     ax, 0ffffh
FFLoop:         mov     [bx], ax
                sub bx, 2
                loop FFLoop
                ret
```

Y-a-t-il ici deux procédures ou juste une? Est-ce qu'en d'autres termes, un programme appelant peut entrer dans ce code aux étiquettes ZeroBytes et DoFFs et ou seulement à ZeroBytes? L'utilisation des directives proc et endp peut aider à lever cette ambiguïté:

Traité comme une routine unique:

```
ZeroBytes       proc
                xor     ax, ax
                jcxz    DoFFs
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret

DoFFs:  mov     cx, 128
                mov     ax, 0ffffh
FFLoop:         mov     [bx], ax
                sub     bx, 2
                loop    FFLoop
                ret
ZeroBytes       endp
```

Traité comme deux routines séparées:

```
ZeroBytes       proc
                xor     ax, ax
                jcxz    DoFFs
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret
ZeroBytes       endp

DoFFs           proc
                mov     cx, 128
                mov     ax, 0ffffh
FFLoop:         mov     [bx], ax
                sub     bx, 2
                loop    FFLoop
                ret
DoFFs           endp
```

Gardez toujours à l'esprit que les directives proc et endp sont des séparateurs *logiques* de procédures. Le microprocesseur 80x86 retourne d'une procédure en exécutant une instruction ret, pas en rencontrant une directive endp. Ce qui suit n'est pas équivalent au code ci-dessus:

```
ZeroBytes       proc
                xor     ax, ax
                jcxz    DoFFs
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
;       Note : manque l'instruction RET.
ZeroBytes       endp
```

```
DoFFs           proc
                mov     cx, 128
                mov     ax, 0ffffh
FFLoop:         mov     [bx], ax
                sub     bx, 2
                loop    FFLoop
                ; Note : manque l'instruction ret.
DoFFs           endp
```

Sans instruction ret à la fin de chaque procédure, le 80x86 tombera dans le sous-programme suivant au lieu de retourner à l'appelant. Après avoir exécuté ZeroBytes ci-dessus, le 80x86 tombera jusqu'au sous-programme DoFFs (commençant par l'instruction mov cx, 128). Une fois traversée DoFFs, le 80x86 continuera l'exécution avec l'instruction exécutable suivant immédiatement la directive endp de DoFFs.

Une procédure 80x86 prend la forme:

```
NomProc         proc    {near|far}              ;Choisissez near, far ou rien.
      <Instructions de la Procedure >
NomProc         endp
```

L'opérande near ou far est facultative, la prochaine section discutera de son utilisation. Le nom de procédure doit être sur les deux lignes proc et endp. Le nom de procédure doit être unique dans le programme. Chaque directive proc doit avoir une directive endp associée. Le fait de ne pas associer les directives proc et endp produira une *erreur d'emboîtement de bloc*.

11.2 Procédures near et far

Le 80x86 supporte des routines near et far. Les appels et les retours near transférent le contrôle entre des procédures dans le même segment de code. Les appels et les retours far font passer le contrôle entre différents segments. Les deux mécanismes d'appel et de retour posent sur la pile et en tirent différentes adresses de retour. Généralement, vous n'employez pas une instruction call proche pour appeler une procédure lointaine ou une instruction call lointaine pour appeler une procédure proche. Etant donné cette petite règle, la question suivante est "Comment controlez-vous l'émission d'un call ou d'un ret near ou far?"

La majeure partie du temps, l'instruction call emploie la syntaxe suivante:

```
                call    ProcName
```

et l'instruction ret est soit[222222]:

```
                ret
soit             ret    disp
```

Malheureusement, ces instructions n'indiquent pas à MASM si vous appelez une procédure proche ou lointaine ou si vous retournez d'une procédure proche ou lointaine. Les directives proc se chargent de cette tâche. La directive proc a une opérande facultative qui est soit near soit far. Near est le défaut si la zone d'opérande est vide[333333]. L'assembleur assigne le type de procédure (near ou far) au symbole. Chaque fois que MASM assemble une instruction call, il émet un call proche ou lointain selon l'opérande. Par conséquent, déclarer un symbole avec proc ou proc near, force un appel proche. De même, utiliser proc far, force un appel lointain.

En plus de contrôler la génération d'un appel proche ou lointain, l'opérande de proc contrôle également la génération du code ret. Si une procédure a l'opérande near, alors toutes les instructions de retour à l'intérieur de cette procédure seront near. MASM émet des retours far à l'intérieur des procédures far.

11.2.1 Forcer des appels et des retours NEAR ou FAR

[2] Il y a aussi les instructions retn et retf

[3] À moins que vous n' employiez les *directives de segment simplifiées* de MASM. Voyez les annexes pour plus de details.

De temps à autre, vous pourriez vouloir outrepasser le mécanisme de déclaration de near/far. MASM fournit un mécanisme qui vous permet de forcer l'utilisation des appels et des retours near/far.

Employez les opérateurs near ptr et far ptr pour outrepasser l'attribution automatique d'un call proche ou lointain. Si NearLbl est une étiquette proche et FarLbl est une étiquette lointaine, alors les instructions call suivantes produisent un appel proche et lointain, respectivement:

```
call    NearLbl         ; Genère un appel NEAR.
call    FarLbl          ; Genère un appel FAR.
```

Supposez que vous deviez faire un appel lointain à NearLbl ou un appel proche à FarLbl. Vous pouvez accomplir ceci en utilisant les instructions suivantes:

```
call    far ptr NearLbl ; Genère un appel FAR.
call    near ptr FarLbl ; Genère un appel NEAR.
```

Appeler une procédure proche en utilisant un call far, ou appelant une procédure lointaine employant un call proche n'est pas quelque chose que vous ferez normalement. Si vous appelez une procédure proche en utilisant une instruction call far, le retour near laissera la valeur de cs sur la pile. Généralement, plutôt que:

```
call    far ptr NearProc
```

vous devriez employer probablement le code plus clair:

```
push    cs
call    NearProc
```

Appeler une procédure lointaine avec un call near est une opération très dangereuse. Si vous essayez un tel appel, la valeur courante de cs doit être sur la pile. Rappelez-vous, un ret lointain extrait une adresse de retour segmentée de la pile. Une instruction call proche pousse seulement l'offset de l'adresse de retour, pas la partie segment.

A partir de MASM v5.0, il y a des instructions explicites que vous pouvez employer pour forcer un ret near ou far. Si ret apparaît dans une procédure déclarée via proc et endp, MASM générera automatiquement l'instruction de retour proche ou lointaine appropriée. Pour accomplir ceci, utilisez les instructions retn et retf. Ces deux instructions produisent un ret near et far, respectivement.

11.2.2 Procédures emboîtées

MASM vous permet d'emboîter des procédures. C'est-à-dire, une définition de procédure peut être totalement enfermée à l'intérieur d'une autre. Ce qui suit est un exemple d'une telle paire de procédures:

```
OutsideProc     proc    near
                jmp EndofOutside
InsideProc      proc    near
                mov     ax, 0
                ret
InsideProc      endp

EndofOutside:   call    InsideProc
                mov bx, 0
                ret
OutsideProc     endp
```

A la différence de langages de haut niveau, les procédures emboîtées ne servent pas à grand chose dans l'assembleur 80x86. Si vous emboîtez une procédure (comme avec InsideProc ci-dessus), vous devrez coder un jmp explicite autour de la procédure emboîtée. Placer la procédure emboîtée après tout le code dans la procédure extérieure (mais cependant entre les directives extérieures proc/endp) n'avance à rien. Par conséquent, il n'y a pas de raison d'emboîter des procédures de cette manière.

Chaque fois que vous emboîtez une procédure dans une autre, elle doit être totalement contenue dans la procédure emboîtante. C'est-à-dire, les instructions proc et endp pour la procédure emboîtée doivent se trouver entre les directives proc et endp de la procédure extérieure, emboîtante. Ce qui suit n'est *pas* légal:

```
OutsideProc     proc    near
.
.
.
InsideProc      proc    near
.
.
.
OutsideProc     endp
.
.
.
InsideProc      endp
```

Les procédures OutsideProc et InsideProc se chevauchent, elles ne sont pas emboîtées. Si vous essayez de créer un ensemble de procédures comme ceci, MASM rapporterait une erreur d'emboîtement de bloc ("block nesting error"). La Figure 11.1 démontre ceci graphiquement.

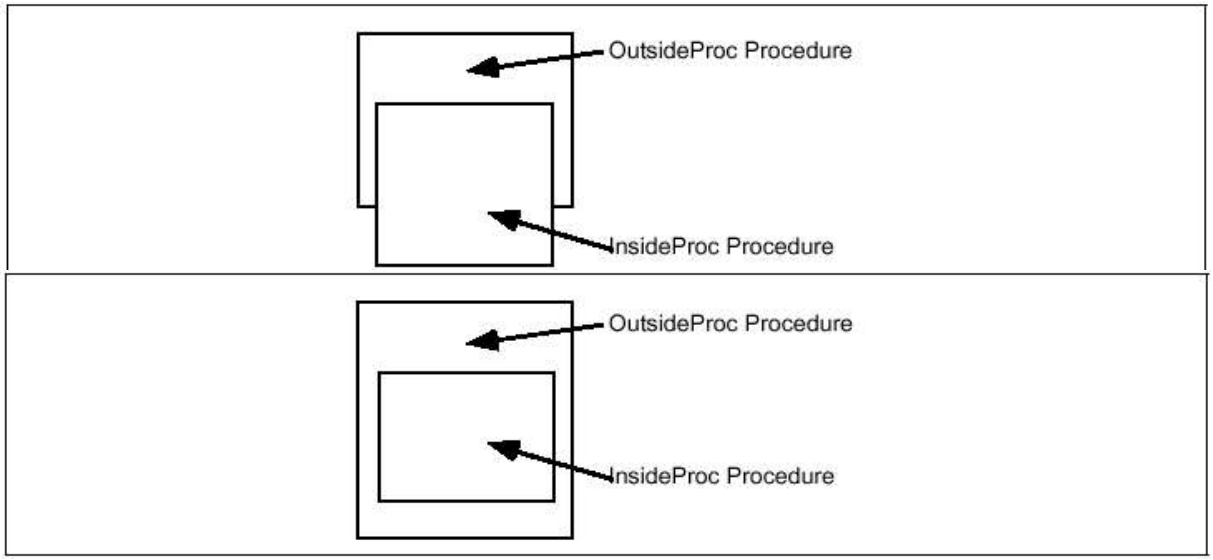


Figure 11.2 Emboîtement légal de procédures

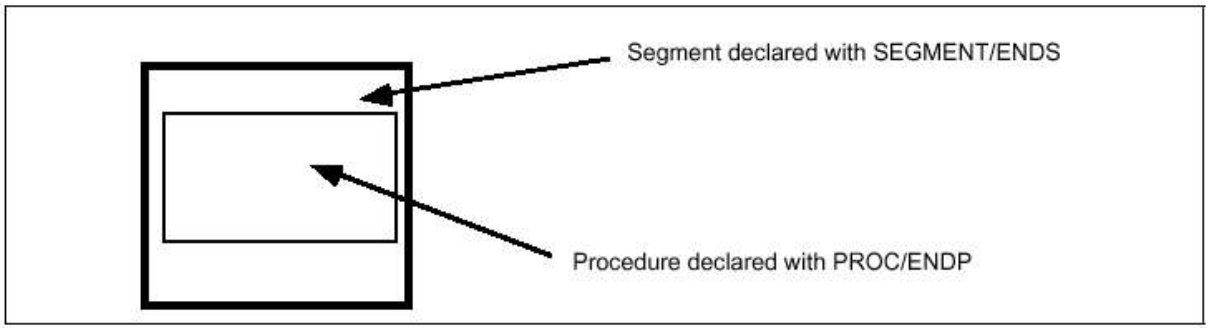


Figure 11.3 Emboîtement légal de procédure / segment

La seule forme acceptable pour MASM apparaît sur la Figure 11.2.

En plus de tenir à l'intérieur d'une procédure enfermante, les groupes de proc/endp doivent tenir entièrement dans un segment. Par conséquent le code suivant est illégal:

```
cseg    segment
MyProc          proc    near
                ret
cseg            ends
MyProc  endp
```

La directive endp doit apparaître avant l'instruction cseg ends puisque MyProc commence à l'intérieur de cseg. Par conséquent, les procédures au sein des segments doivent toujours prendre la forme représentée sur la Figure 11.3.

Non seulement vous pouvez emboîter des procédures à l'intérieur d'autres procédures et segments, mais vous pouvez aussi emboîter des segments à l'intérieur d'autres procédures et segments. Si vous êtes du type à aimer simuler des procédures de Pascal ou C en assembleur, vous pouvez créer des sections de déclaration de variables au début de chaque procédure que vous créez, tout comme en Pascal:

```
cgroup          group           cseg1, cseg2
cseg1           segment         para public 'code'
cseg1           ends
cseg2           segment         para public 'code'
cseg2           ends
dseg            segment         para public 'data'
dseg            ends

cseg1           segment         para public 'code'
                assume          cs:cgroup, ds:dseg

MainPgm         proc            near

; Déclarations de données pour le programme principal:

dseg            segment         para public 'data'
I               word            ?
J               word            ?
dseg            ends

; Procédures qui sont locales au programme principal:

cseg2           segment         para public 'code'

ZeroWords       proc            near

; Variables locales à ZeroBytes:

dseg            segment         para public 'data'
AXSave          word            ?
BXSave          word            ?
CXSave          word            ?
dseg            ends

; Code pour la procédure ZeroBytes:

                mov         AXSave, ax
                mov         CXSave, cx
                mov         BXSave, bx
                xor         ax, ax
ZeroLoop:       mov         [bx], ax
                inc         bx
                inc         bx
                loop        ZeroLoop
                mov         ax, AXSave
                mov         bx, BXSave
```

```
            mov       cx, CXSave
            ret
ZeroWords   endp

Cseg2   ends

; Le program principal réel commence ici:

            mov       bx, offset Array
            mov       cx, 128
            call      ZeroWords
            ret
MainPgm        endp
cseg1          ends
               end
```

Le système chargera ce code dans la mémoire comme sur la Figure 11.4.

ZeroWords *suit* le programme principal parce qu'il appartient à un segment différent (cseg2) que MainPgm (cseg1). Rappelez-vous, l'assembleur et l'éditeur de liens combine les segments avec le même nom de classe dans un segment unique avant de les charger dans la mémoire (voir "Ordre de Chargement des Segments" au chapitre 8 pour plus de détails). Vous pouvez employer cette fonctionnalité de l'assembleur pour "pseudo-Pascaliser" votre code de la façon montrée ci-dessus. Cependant, vous ne trouverez probablement pas que vos programmes sont plus lisibles qu'en utilisant l'approche normale sans-emboîtement.

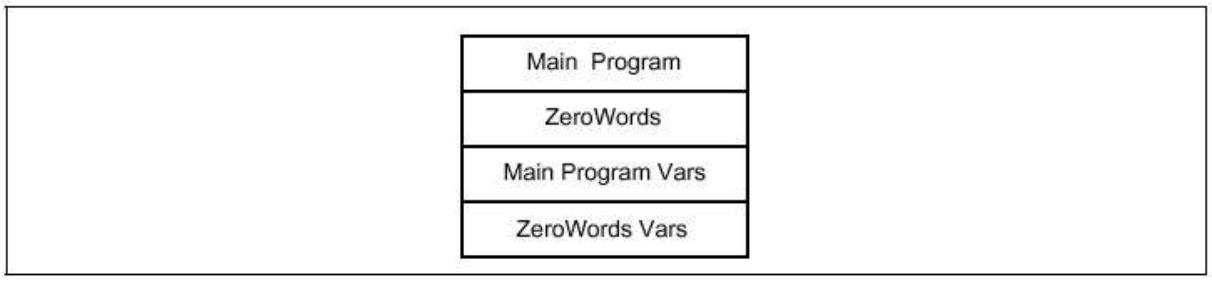


Figure 11.4 Disposition de la mémoire de l'exemple

11.3 Fonctions

La différence entre les fonctions et les procédures en assembleur est surtout une question de définition. Le but pour une fonction est de renvoyer une certaine valeur explicite tandis que le but pour une procédure est d'exécuter une certaine action. Pour déclarer une fonction en assembleur, utilisez les directives proc/endp. Toutes les règles et techniques qui s'appliquent aux procédures s'appliquent aux fonctions. Ce texte jettera un œil sur les fonctions plus tard dans ce chapitre dans la section sur des résultats des fonction. D'ici là, procédure signifiera procédure ou fonction.

11.4 Sauvegarder l'état de la machine

Jetez un coup d'oeil à ce code:

```
            mov     cx, 10
Loop0:      call    PrintSpaces
            putcr
            loop    Loop0
            .
            .
            .
PrintSpaces proc    near
            mov     al, ' '
```

```
                mov     cx, 40
PSLoop:         putc
                loop    PSLoop
                ret
PrintSpaces     endp
```

Cette section de code essaye d'imprimer dix lignes de 40 espaces chacune. Malheureusement, il y a un bogue subtil qui provoque l'impression des 40 espaces par ligne en boucle sans fin. Le programme principal emploie l'instruction loop pour appeler PrintSpaces 10 fois. PrintSpaces emploie cx pour décompter les 40 espaces qu'il imprime. PrintSpaces retourne avec cx contenant zéro. Le programme principal imprime alors un retour de chariot/saut à la ligne, décrémente cx et puis recommence parce que cx n'est pas égal àzéro (il contiendra toujours 0FFFFh à ce point).

Le problème ici est que le sous-programme PrintSpaces ne préserve pas le registre cx. Préserver un registre signifie que vous le sauvegardez à l'entrée du sous-programme et le restituez avant de partir. Le sous-programme PrintSpaces eut-il préservé le contenu du registre cx, le programme ci-dessus aurait fonctionné correctement.

Utilisez les instructions push et pop du 80x86 pour préserver des valeurs de registre alors que vous devez les employer pour autre chose. Considérez le code suivant pour PrintSpaces:

```
PrintSpaces     proc    near
                push    ax
                push    cx
                mov     al, ' '
                mov     cx, 40
PSLoop:         putc
                loop    PSLoop
                pop     cx
                pop     ax
                ret
PrintSpaces     endp
```

Notez que PrintSpaces sauve et restitue ax et cx (puisque ce procédure modifie ces registres). En outre, notez que ce code retire les registres de la pile dans l'ordre inverse qu'il les a poussés. Le mode d'emploi de la pile impose cet ordre..

Soit l'appelant (le code contenant l'instruction call) soit l'appelé (le sous-programme) peut prendre la responsabilité de préserver les registres. Dans l'exemple ci-dessus, l'appelé a préservé les registres. L'exemple suivant montre à quoi pourrait ressembler ce code si l'appelant préserve les registres:

```
                mov     cx, 10
Loop0:                                                          push
                                        ax
                push    cx
                call    PrintSpaces
                pop     cx
                pop     ax
                putcr
                loop    Loop0
                .
                .
                .
PrintSpaces     proc    near
                mov     al, ' '
                mov     cx, 40
PSLoop:         putc
                loop    PSLoop
                ret
PrintSpaces                             endp
```

Il y a deux avantages à la préservation par l'appelé: l'espace et la maintenance. Si l'appelé préserve tous les registres affectés, alors il y a seulement une copie des instructions push et pop, celles que la procédure contient. Si l'appelant sauve les valeurs dans les registres, le programme a besoin d'un ensemble d'instructions push et pop autour de chaque appel. Non seulement ceci rend vos programmes plus longs, il les rend également plus difficiles à maintenir. Se rappeler quels registres il faut pousser et extraire à chaque appel de procédure n'est pas chose facile.

Par contre, un sous-programme peut inutilement préserver quelques registres s'il préserve tous les registres qu'il modifie. Dans les exemples ci-dessus, le code n'a pas besoin de sauver ax. Bien que PrintSpaces change al, ceci n'affectera pas l'exécution du programme. Si l'appelant préserve les registres, il n'a pas à sauver des registres qui ne l'intéresent pas:

```
                mov     cx, 10
Loop0:          push    cx
                call    PrintSpaces
                pop     cx
                putcr
                loop    Loop0
                putcr
                putcr
                call    PrintSpaces
                mov     al, '*'
                mov     cx, 100
Loop1:          putc
                push    ax
                push    cx
                call    PrintSpaces
                pop     cx
                pop     ax
                putc
                putcr
                loop    Loop1
                .
                .
                .
PrintSpaces     proc    near
                mov     al, ' '
                mov     cx, 40
PSLoop:         putc
                loop    PSLoop
                ret
PrintSpaces     endp
```

Cet exemple fournit trois cas différents. La première boucle (Loop0) préserve seulement le registre cx. Modifier le registre al n'affectera pas l'exécution de ce programme. Immédiatement après la première boucle, ce code appelle PrintSpaces une nouvelle fois. Cependant, ce code ne sauve pas ax ou cx parce qu'il s'en fiche si PrintSpaces les change. Comme la boucle finale (Loop1) utilise ax et cx, elle les sauve tous les deux.

Un gros problème en faisant préserver les registres par l'appelant est que votre programme peut changer. Vous pouvez modifier le code appelant ou la procédure de sorte qu'ils utilisent des registres supplémentaires. De tels changements, naturellement, peuvent changer le jeu de registres que vous devez préserver. Pire encore, si la modification est dans le sous-programme lui-même, vous devrez localiser chaque appel à la routine et vérifier que le sous-programme ne change aucun registre que le code appelant utilise.

La préservation de l'environnement ne se limite pas à la préservation des registres. Vous pouvez également pousser et retirer des variables et d'autres valeurs qu'un sous-programme pourrait changer. Puisque le 80x86 vous permet de pousser et extraire des endroits de mémoire, vous pouvez aussi facilement préserver ces valeurs.

11.5 Les paramètres

Bien qu'il y ait une grande classe de procédures qui sont totalement indépendantes, la plupart d'elles demandent des données en entrée et renvoient des données à l'appelant. Les paramètres sont des valeurs que vous échangez avec une procédure. Il y a bien des facettes aux paramètres. Les questions sur les paramètres incluent:

- *D'où* viennent les données?
- *Comment* passez-vous et renvoyez-vous des données?
- *Quelle* est la quantité de données à passer?

Il y a six mécanismes principaux pour échanger des données avec une procédure, ce sont

- passage par valeur,
- passage par référence,
- passage par valeur/retournée
- passage par résultat,
- passage par nom et
- passage par évaluation paresseuse

Vous devez également vous soucier de l'endroit où vous pouvez passer des paramètres. Les endroits habituels sont

- dans des registres
- dans des emplacements de mémoire globaux
- sur la pile
- dans le jet de code, ou
- dans un bloc de paramètre référencé par l'intermédiaire d'un pointeur.

En conclusion, la quantité de données a une influence directe sur l'endroit et la façon de les passer. Les sections suivantes reprennent ces questions.

11.5.1 Le passage par valeur

Un paramètre passé par valeur n'est que cela - l'appelant passe une valeur à la procédure. Les paramètres passés par valeur sont des paramètres d'entrée seule. C'est-à-dire, vous pouvez les passer à une procédure mais la procédure ne peut pas les renvoyer. Dans des HLLs (High Level Langages ou langages de haut niveau), comme le Pascal, l'idée qu'un paramètre passé par valeur est un paramètre d'entrée seulement paraît très logique. Étant donné l'appel de procédure Pascal:

```
CallProc(I);
```

Si vous passez I par valeur, CallProc ne change pas la valeur de I, indépendamment de ce qui arrive au paramètre à l'intérieur de CallProc.

Puisque vous devez passer une copie des données à la procédure, vous devriez seulement employer cette méthode pour passer de petits objets comme des bytes, des mots, et des doubles mots. Le passage de tableaux et de chaînes par valeur est très inefficace (puisque vous devez créer et passer une copie de la structure à la procédure).

11.5.2 Le passage par référence

Pour passer un paramètre par référence, vous devez passer l'adresse d'une variable et non sa valeur. En d'autres termes, vous devez passer un pointeur sur les données. La procédure doit déréférencer ce pointeur pour accéder aux données. Le passage des paramètres par référence est utile quand vous devez modifier le paramètre effectif ou quand vous passez de grandes structures de données entre les procédures.

Le passage des paramètres par référence peut produire des résultats bizarres. La procédure Pascal suivante montre un exemple d'un problème que vous pourriez rencontrer:

```
program main(input,output);
var m:integer;
procedure bletch(var i,j:integer);
begin
     i := i+2;
     j := j-i;
     writeln(i,' ',j);
end;
.
.
```

```
.
begin {main}

m := 5;
bletch(m,m);

end.
```

Cette séquence de code particulière imprimera "00" indépendamment de la valeur de m. Ceci parce que les paramètres i et j sont des pointeurs sur les données réelles et qu'ils pointent tous les deux sur le même objet. Par conséquent, l'instruction j:=j-i; produit toujours zéro puisqu'i et j se réfèrent à la même variable.

Le passage par référence est habituellement moins efficace que le passage par valeur. Vous devez déréférencer tous les paramètres passés par référence à chaque accès; c'est plus lent qu'utiliser simplement une valeur. Cependant, quand on passe une grande structure de données, le passage par référence est plus rapide parce que vous ne devez pas copier une grande quantité de données avant d'appeler la procédure.

11.5.3 Le passage par valeur retournée

Le passage par Valeur-Retournée (également connu comme *valeur-résultat*) combine les aspects des mécanismes à la fois du passage par valeur et du passage par référence. Vous passez un paramètre valeur-retournée par son adresse, tout comme les paramètres passés par référence. Cependant, à l'entrée, la procédure fait une copie provisoire de ce paramètre et utilise la copie pendant que la procédure s'exécute. Quand la procédure se termine, elle copie la copie provisoire de nouveau dans le paramètre original.

Le code de Pascal présenté dans la section précédente fonctionnerait correctement avec des paramètres passés par valeur-retournée. Naturellement, quand Bletch revient au code appelant, m pourrait seulement contenir une des deux valeurs, mais pendant que Bletch s'exécute, i et j contiendraient des valeurs distinctes.

Parfois, le passage par valeur-retournée est plus efficace que le passage par référence, dans d'autres cas, il est moins efficace. Si une procédure référence seulement le paramètre une fois ou deux, copier les données du parametre est coûteux. D'autre part, si la procédure utilise ce paramètre souvent, elle amortit le coût fixe de copier les données par de nombreux accès peu coûteux à la copie locale.

11.5.4 Le passage par résultat

Le passage par résultat est presque identique au passage par valeur-retournée. Vous passez un pointeur sur l'objet désiré et la procédure utilise une copie locale de la variable et puis stocke le résultat par l'intermédiaire du pointeur au retour. La seule différence entre le passage par valeur-retournée et le passage par résultat est qu'en passant des paramètres par résultat, vous ne copiez pas les données lors de l'entrée dans la procédure. L'utilité des paramètres passés par résultat est de renvoyer des valeurs pas de passer des données à la procédure. Par conséquent, le passage par résultat est légèrement plus efficace que le passage par valeur-retournée puisque vous économisez le coût de copier les données dans la variable locale.

11.5.5 Le passage par nom

Le passage par nom est le mécanisme de passage de paramètre employé par les macros, les "text equates" et la macro utilitaire "#definine" dans le langage de programmation de C. Ce mécanisme de passage de paramètre utilise la substitution textuelle sur les paramètres. Considérez la macro suivante de MASM:

```
PassByName      macro   Parameter1, Parameter2
                mov             ax, Parameter1
                add             ax, Parameter2
                endm
```

Si vous avez une invocation de la macro de la forme:

```
            PassByName bx, I
```

MASM émet le code suivant, substituant bx à Parameter1 et I à Parameter2:

```
            mov ax, bx
            add ax, I
```

Quelques langages de haut niveau, telles qu'algol-68 et Panacea, supportent les paramètres passés par nom. Cependant, mettre en application le passage par nom utilisant la substitution textuelle dans un langage compilé (comme ALGOL-68) est très difficile et inefficace. Fondamentalement, vous devriez recompiler une fonction chaquer fois que vous l'appelez. Les langages ainsi compilés qui soutiennent des paramètres passés par nom utilisent généralement une technique différente pour passer ces paramètres. Considérez la procédure Panacea suivante:

```
PassByName: procedure(name item:integer; var index:integer);
begin PassByName;

      foreach index in 0..10 do
            item := 0;

      endfor;

end PassByName;
```

Supposez que vous appellez cette routine avec l'instruction PassByName(A[i], i); où A est un tableau de nombres entiers ayant (au moins) les éléments A[0]..A[10]. Si vous deviez substituer le paramètre passé par nom *item*, vous obtiendriez le code suivant:

```
begin PassByName;

      foreach index in 0..10 do

            A[I] := 0; (* Notez que index et I sont des alias *)

      endfor;

end PassByName;
```

Ce code met à zéro les éléments 0..10 du tableau A.

Des langages de haut niveau comme ALGOL-68 et Panacea compilent les paramètres passés par nom dans des *fonctions* qui renvoient l'adresse d'un paramètre donné. Ainsi à un égard, les paramètres passés par nom sont semblables aux paramètres passés par référence dans la mesure où vous passez l'adresse d'un objet. La différence principale est qu'avec le passage par référence, vous calculez l'adresse d'un objet avant d'appeler un sous-programme; et qu'avec le passage par nom le sous-programme lui-même appelle une fonction déterminée pour calculer l'adresse du paramètre.

Ainsi quelle différence est-ce que cela fait? Eh bien, reconsidérez le code ci-dessus. Si vous aviez passé A[I] par référence plutôt que par nom, le code appelant calculerait l'adresse d'A[I] *juste avant l'appel* et passerait cette adresse. À l'intérieur de la procédure PassByName, la variable item se serait toujours référée à une adresse simple, pas une adresse qui change avec I. Avec des paramètres passés par nom, item est vraiment une fonction qui calcule l'adresse du paramètre dans lequel la procédure stocke la valeur zéro. Une telle fonction pourrait ressembler à ce qui suit:

```
ItemThunk       proc    near
                mov     bx, I
                shl     bx, 1
                lea     bx, A[bx]
                ret
ItemThunk       endp
```

Le code compilé à l'intérieur de la procédure PassByName pourrait ressembler quelque chose comme ce qui suit:

```
; item := 0;

            call ItemThunk
            mov word ptr [bx], 0
```

Thunk est le terme historique pour ces fonctions qui calculent l'adresse d'un paramètre passé par nom. Il vaut la peine de noter que la plupart des langages de haut niveau qui supportent des paramètres passés par nom n'appellent pas des thunks directement (comme l'appel ci-dessus). Généralement, l'appelant passe l'adresse d'un thunk et le sous-programme appelle le thunk *indirectement*. Ceci permet à la même séquence d'instructions d'appeler plusieurs thunks différents (correspondant à différents appels au sous-programme).

11.5.6 Le passage par évaluation paresseuse

Le passage par nom est semblable au passage par référence dans la mesure où la procédure accède au paramètre en utilisant l'adresse du paramètre. La différence principale entre les deux est qu'un appelant passe directement l'adresse sur la pile avec le passage par référence, et qu'il passe l'adresse d'une fonction qui calcule l'adresse du paramètre en passant un paramètre par nom. Le mécanisme paresseux de passage par évaluation paresseuse partage cette même relation avec les paramètres passés par valeur - l'appelant passe l'adresse d'une fonction qui calcule la valeur du paramètre si le premier accès à ce paramètre est une opération de lecture.

Le passage par évaluation paresseuse est une technique utile de passage de paramètre si le coût pour calculer la valeur du paramètre est très élevé et que la procédure peut ne pas employer la valeur. Considérez l'en-tête de procédure suivant de Panacea:

```
PassByEval: procedure(eval a:integer; eval b:integer; eval c:integer);
```

Quand vous appelez la fonction PassByEval, elle n'évalue pas les paramètres effectifs et ne passe pas leurs valeurs à la procédure. Au lieu de cela, le compilateur produit des thunks qui calculeront la valeur du paramètre tout au plus une fois. Si le premier accès à un paramètre eval est une lecture, le thunk calculera la valeur du paramètre et stockera cela dans une variable locale. Il placera également un drapeau de sorte que tous les futurs accès n'appellent pas le thunk (puisqu'il a déjà calculé la valeur du paramètre). Si le premier accès à un paramètre eval est une écriture, alors le code active le drapeau et de futurs accès dans la même activation de procédure utiliseront la valeur écrite et ignoreront le thunk.

Considérez la procédure PassByEval ci-dessus. Supposez que cela prend plusieurs minutes pour calculer les valeurs pour les paramètres a, b et c (ceux-ci pourraient être, par exemple, trois chemins possibles différents dans un jeu d'échecs). Il se peut que la procédure PassByEval utilise seulement la valeur d'un de ces paramètres. Sans passage par évaluation paresseuse, le code appelant devrait passer du temps à calculer chacun des trois paramètres quoique la procédure utilise seulement une des valeurs. Avec le passage par évaluation paresseuse, cependant, la procédure ne perdra du temps qu'à calculer la valeur du seul paramètre dont elle a besoin. L'évaluation paresseuse est une technique commune que l'intelligence artificielle (AI) et les systèmes d'exploitation utilisent pour améliorer les performances.

11.5.7 Le passage de paramètres dans les registres

Ayant traité de la façon dont passer des paramètres à une procédure, la prochaine chose dont il faut discuter est où passer des paramètres. Là où vous passez des paramètres dépend, en grande partie, de la taille et du nombre de ces paramètres. Si vous passez un nombre restreint de bytes à une procédure, alors les registres sont un excellent endroit pour passer des paramètres. Et ils sont un endroit idéal pour passer des paramètres de valeur à une procédure. Si vous passez un paramètre unique à une procédure vous devriez utiliser les registres suivants pour les types de données qui vont avec:

Taille de Données	Registre à Utiliser
Byte:	al
Mot:	ax
Double-mot:	dx:ax ou eax (si 80386 ou mieux)

Ceci est, nullement, une règle pure et dure. Si vous trouvez plus commode de passer des valeurs de 16 bits dans les registres si ou bx, faites le. Cependant, la plupart des programmeurs utilisent les registres ci-dessus.

Si vous passez plusieurs paramètres à une procédure dans les registres du 80x86, vous devriez probablement utiliser les registres dans l'ordre suivant:

Premier Dernier

ax, dx, si, di, bx, cx

En général, vous devriez éviter d'utiliser le registre bp. Si vous avez besoin de plus de six mots, peut-être devriez-vous passer vos valeurs ailleurs.

La livraison de la Bibliothèque Standard de l'UCR fournit plusieurs bons exemples de procédures qui passent des paramètres par valeur dans les registres. putc, qui sort un code de caractère ASCII à l'affichage vidéo, attend une valeur ASCII dans le registre al. De même, puti attend la valeur d'un nombre entier signé dans le registre ax. Comme autre exemple, considérez la routine suivante putsi (put short integer) qui affiche la valeur dans al comme nombre entier signé:

```
putsi           proc
push            ax              ;Sauve la valeur d'AH.
                cbw             ;Étend le signe d'AL -> AX.
                puti            ;Donne le travail effectif à puti.
                pop ax          ;Restaure AH.
                ret
putsi           endp
```

Les quatre autres mécanismes passage de paramètre (passage par référence, valeur-retournée, résultat et nom), exigent généralement que vous passiez un pointeur sur l'objet désiré (ou sur un thunk dans le cas du passage par nom). En passant de tels paramètres dans des registres, vous devez prendre en compte si vous passez un offset ou une adresse segmentée complète. Les offsets de seize bits peuvent être passés dans n'importe lequel des registres de 16 bits d'usage général du 80x86, si, di et bx sont les meilleurs endroits pour passer un offset puisque vous aurez probablement besoin de le charger dans un de ces registres de toute façon[444444]. Vous pouvez passer des adresses segmentées de 32 bits comme dx:ax comme d'autres paramètres de double-word. Cependant, vous pouvez également les passer dans ds:bx, ds:si, ds:di, es:bx, es:si ou es:di et pouvoir les employer sans faire de copie dans un registre de segment.

La routine de l'UCR Stdlib puts, qui imprime une chaîne sur l'affichage vidéo, est un bon exemple d'un sous-programme qui utilise le passage par référence. Elle demande l'adresse d'une chaîne dans la paire de registre es:di. Elle passe le paramètre de cette façon, non pas parce qu'elle modifie celui-ci, mais parce que les chaînes sont plutôt longues et que les passer autrement serait inefficace. Comme autre exemple, considérez la routine strfill(str, c) qui copie le caractère c (passé par valeur dans al) dans chaque position de caractère dans str (passé par la référence dans es:di) jusqu'à un byte de terminaison zéro:

```
; strfill-              copie la valeur dans al dans la chaîne pointée par es:di
;                       jusqu'à un byte de terminaison zéro.

byp             textequ         <byte ptr>
strfill         proc
                pushf           ;Sauve le flag de direction.
                cld             ;Pour incrémenter D avec STOS.
                push    di      ;Sauvé, parce qu'il est modifié.
                jmp             sfStart
sfLoop:         stosb                           ;es:[di] := al, di := di + 1;
sfStart:        cmp     byp es:[di], 0  ;Fini?
                jne     sfLoop
                pop     di      ;Restaure di.
                popf            ;Restaure flag de direction.
                ret
strfill         endp
```

Quand on passe des paramètres par valeur-retournée ou par résultat à un sous-programme, on pourrait passer l'adresse dans un registre. À l'intérieur de la procédure, on copierait la valeur pointée par ce registre dans une variable locale (seulement pour la valeur-retournée). Juste avant que la procédure revienne à l'appelant, elle pourrait stocker le résultat final de nouveau à l'adresse dans le registre.

Le code suivant demande deux paramètres. Le premier est un paramètre passé par valeur-retournée et le sous-programme attend l'adresse du paramètre réel dans bx. Le second est un paramètre passé par résultat dont l'adresse est

[4] Ceci ne s'applique pas aux thunks. Vous pouvez passer l'adresse d'un thunk dans n'importe quel registre de 16 bits. Naturellement, sur un processeur 80386 ou postérieur, vous pouvez utiliser n'importe lequel des registres de 32 bits du 80386 pour contenir une addresse.

dans si. Cette routine incrémente le paramètre passé par valeur-retournée et stocke le résultat précédent dans le paramètre passé par résultat:

```
; CopyAndInc-   BX contient l'addresse d'une variable. Cette routine
;                       copie cette variable à l'emplacement specifié dans SI,
;                       puis incrémente la variable sur laquelle pointe BX.
;                       Note: AX and CX contiennent les copies locales de ces
;                       paramètres pendant l'exécution.

CopyAndInc      proc
                push    ax              ;Préserve AX pendant l'appel.
                push    cx              ;Préserve CX pendant l'appel.
                mov     ax, [bx]        ;Obtient une copie locale du 1er paramètre.
                mov     cx, ax          ;La stocke dans la var locale du 2ème parm.
                inc     ax              ;Incrémente le 1er paramètre.
                mov     [si], cx        ;Stocke le parm passé par resultat.
                mov     [bx], ax        ;Stocke le parm passé par valeur/ret.
                        pop     cx      ;Restaure la valeur de CX.
                        pop     ax      ;Restaure la valeur de AX.
                        ret
CopyAndInc              endp
```

Pour faire l'appel à CopyAndInc(I, J) vous utiliseriez du code comme ce qui suit:

```
                        lea     bx, I
                        lea     si, J
                        call    CopyAndInc
```

Ceci est, naturellement, un exemple banal dont l'exécution est très inefficace. Néanmoins, il montre comment passer des paramètres par valeur-retournée et par résultat dans les registres du 80x86. Si vous êtes disposé à échanger un peu d'espace contre de la vitesse, il y a une autre manière d'arriver aux mêmes résultats que par le passage par valeur-retournée ou le passage par résultat en passant des paramètres dans des registres. Considérez l'implémentation suivante de CopyAndInc:

```
CopyAndInc              proc
                        mov     cx, ax  ;Faire une copie du 1er paramètre,
                        inc     ax      ; puis l'incrémenter de un.
                        ret
CopyAndInc              endp
```

Pour faire appel à CopyAndInc(I, J), comme avant, vous utiliseriez le code 80x86 suivant:

```
                        mov     ax, I
                        call    CopyAndInc
                        mov     I, ax
                        mov     J, cx
```

Remarquez que ce code ne passe absolument aucune adresse; pourtant il a la même sémantique (c'est-à-dire, effectue les mêmes opérations) que la version précédente. Les deux versions incrémentent I et stockent la version pré-incrémentée dans J. Clairement la dernière version est plus rapide, bien que votre programme soit légèrement plus grand s'il y a beaucoup d'appels à CopyAndInc dans votre programme (six ou plus).

Vous pouvez passer un paramètre par nom ou par évaluation paresseuse dans un registre en chargeant simplement ce registre avec l'adresse du thunk à appeller. Considérez la procédure PassByName de Panacea (voir la section "Le Passage par Nom", plus haut). Une implémentation de cette procédure pourrait être la suivante:

```
;PassByName-    Attend un paramètre index, passé par référence
;                       dans si et un paramètre item, passé par nom
;                       dans dx (le thunk renvoie l'addresse dans bx).
PassByName              proc
                        push    ax                      ;Préserve AX pendant l'appel
                        mov     word ptr [si], 0        ;Index := 0;
ForLoop:                cmp     word ptr [si], 10 ;Le loop ForLoop finit à dix.
                        jg      ForDone
```

```
                        call    dx                      ;Appelle le thunk item.
                        mov     word ptr [bx], 0        ;Stocke zéro dans item.
                        inc     word ptr [si]   ;Index := Index + 1;
                        jmp     ForLoop
ForDone:                pop     ax                      ;Restaure AX.
                        ret                             ;Fini!
PassByName              endp
```

Vous pourriez appeler cette routine avec du code qui ressemble à ce qui suit:

```
                        lea     si, I
                        lea     dx, Thunk_A
                        call    PassByName
                        .
                        .
                        .
Thunk_A                 proc
                        mov     bx, I
                        shl     bx, 1
                        lea     bx, A[bx]
                        ret
Thunk_A                 endp
```

L'avantage de cet arrangement, par rapport à celui présenté dans la section précédente, est que vous pouvez appeler différents thunks, pas simplement la routine ItemThunk apparaissant dans celui-ci.

11.5.8 Le passage de paramètres dans des variables globales

Dès que vous manquez de registres, la seule autre alternative (raisonnable) que vous avez est la mémoire centrale. Un des endroits les plus faciles pour passer des paramètres est dans des variables globales dans le segment de données. Le code suivant en fournit un exemple:

```
                mov     ax, xxxx                ;Passe ce paramètre par valeur
                mov     Value1Proc1, ax
                mov     ax, offset yyyy         ;Passe ce paramètre par ref
                mov     word ptr Ref1Proc1, ax
                mov     ax, seg yyyy
                mov     word ptr Ref1Proc1+2, ax
                call    ThisProc
                .
                .
ThisProc        proc    near
                push    es
                push    ax
                push    bx
                les     bx, Ref1Proc1           ;Trouve l'addresse du parm ref.
                mov     ax, Value1Proc1         ;Trouve le paramètre valeur
                mov     es:[bx], ax             ;Stocke dans mem pointée par
                pop     bx                      ; le paramètre ref.
                pop     ax
                pop     es
                ret
ThisProc endp
```

Passer des paramètres dans des emplacements globaux est inélégant et inefficace. En outre, si vous utilisez des variables globales de cette façon pour passer des paramètres, les sous-programmes que vous écrivez ne peuvent - pas employer la récursion (voir "La Récursion" à la section 11.9). Heureusement, il y a de meilleurs méthodes de passage de paramètre pour passer des données qui sont dans la mémoire, aussi vous n'avez pas besoin de prendre sérieusement en compte cette méthode.

11.5.9 Le passage de paramètres sur la pile

La plupart des langages de haut niveau utilisent la pile pour passer des paramètres parce que cette méthode est assez efficace. Pour ce faire, poussez-les juste avant d'appeler le sous-programme. Le sous-programme lit ensuite ces données depuis la mémoire de pile et agit sur elles de manière appropriée. Considérez l'appel de procédure Pascal suivant:

```
CallProc(i, j, k+4);
```

La plupart des compilateurs Pascal poussent leurs paramètres sur la pile dans l'ordre où ils apparaissent dans la liste de paramètres. Par conséquent, le code 80x86 habituellement émis pour cet appel de sous-programme (à supposer que vous passez les paramètres par valeur) est

```
push    i
push    j
mov     ax, k
add     ax, 4
push    ax
call    CallProc
```

A l'entrée dans CallProc, la pile du 80x86 ressemblera à la Figure 11.5 (pour une procédure near) ou à la Figure 11.6 (pour une procédure far).

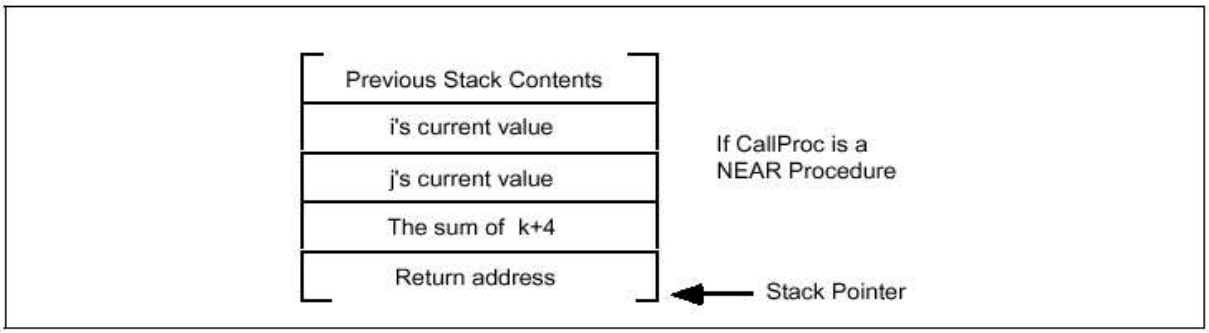


Figure 11.5 Disposition de la pile de CallProc pour une procédure near

Vous pourriez accéder aux paramètres passés sur la pile en extrayant les données de la pile (A supposer un appel de procédure near):

```
CallProc                        proc                                    near
                pop     RtnAdrs
                pop     kParm
                pop     jParm
                pop     iParm
                push    RtnAdrs
                .
                .
                .
                ret
CallProc        endp
```

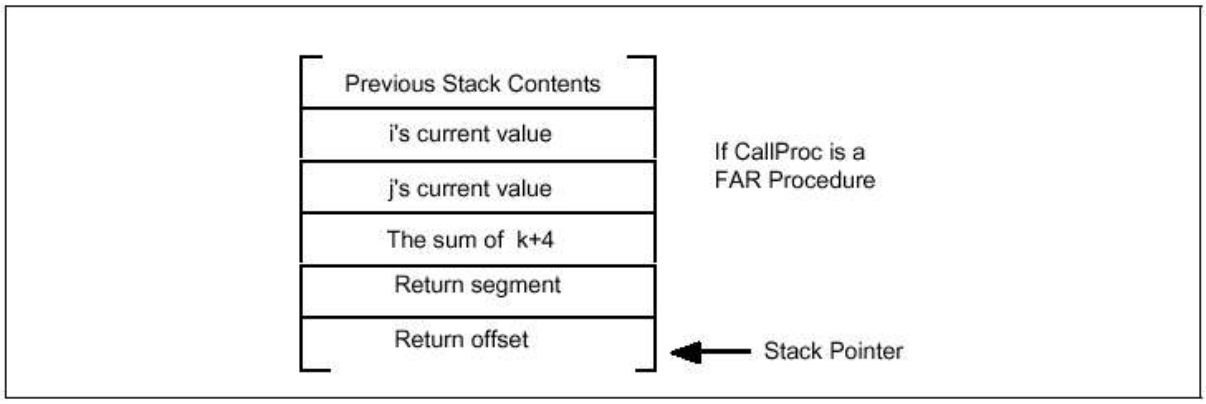


Figure 11.6 Disposition de la pile de CallProc pour une procédure Far

Il y a, cependant, une meilleure manière. L'architecture du 80x86 vous permet d'utiliser le registre bp (base pointer) pour accéder à des paramètres passés sur la pile. C'est l'une des raisons pour lesquelles les modes d'adressage disp[bp], [bp][di], [bp][si], disp[bp][si] et disp[bp][di] utilisent le segment de pile plutôt que le segment de données.

Le segment de code suivant donne le code *standard d'entrée et de sortie de procédure*:

```
StdProc         proc    near
                push    bp
                mov     bp, sp
                .
                .
                .
                pop     bp
                ret     ParmSize
StdProc         endp
```

ParmSize est le nombre de bytes de paramètres poussés sur la pile avant d'appeler la procédure. Dans la procédure CallProc, il y avait six bytes de paramètres poussés sur la pile, aussi ParmSize serait égal à six.

Jetez un coup d'oeil à la pile juste après l'exécution de mov bp, sp dans StdProc. En supposant que vous avez poussé trois paramètres words sur la pile, cela devrait sembler quelque peu à la Figure 11.7.

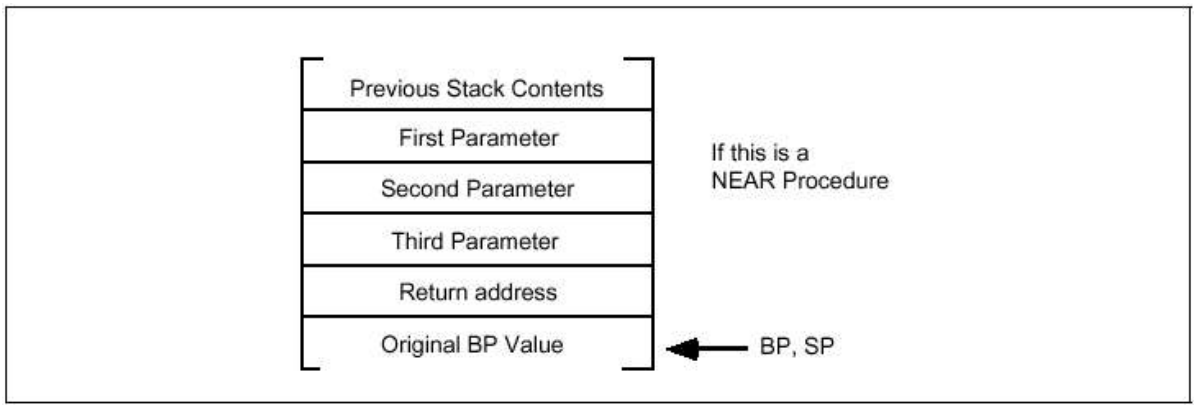


Figure 11.7 Accès aux paramètres sur la pile

Maintenant les paramètres peuvent être obtenus en en indexant le registre bp:

```
            mov     ax, 8[bp]           ;Accède au premier paramètre
            mov     ax, 6[bp]           ; Accède au second paramètre
            mov     ax, 4[bp]           ; Accède au troisième paramètre
```

En retournant au code appelant, la procédure doit enlever ces paramètres de la pile. Pour réaliser ceci, extrayez l'ancienne valeur de bp de la pile et exécutez une instruction ret 6. Ceci extrait l'adresse de retour de la pile et additionne six au pointeur de pile, enlevant efficacement les paramètres de la pile

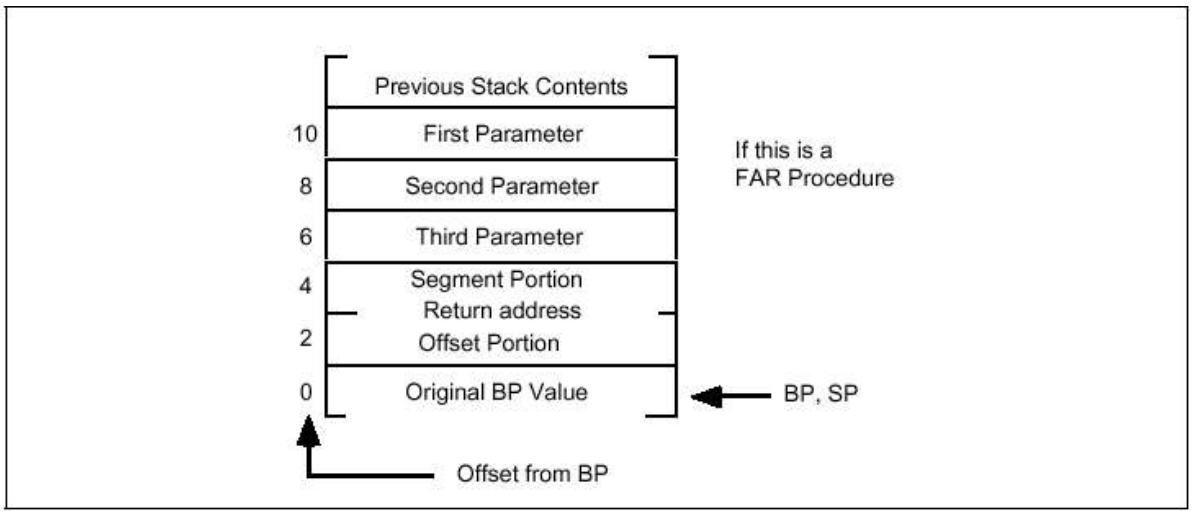


Figure 11.8 Accès aux paramètres sur la pile dans une procédure FAR

Les déplacements donnés ci-dessus sont pour des procédures proches seulement. Quand on appelle une procédure lointaine

- 0[BP] pointera sur l'ancienne valeur de bp
- 2[BP] pointera sur la partie offset de l'adresse de retour
- 4[BP] pointera sur la partie de segment de l'adresse de retour et
- 6[BP] pointera sur le dernier paramètre poussé sur la pile.

Le contenu de pile lors d'un appel à une procédure lointaine est montré sur la Figure 11.8.

Cette collection de paramètres, adresse de retour, registres sauvés sur la pile, et autres objets, est un *cadre de pile (stack frame)* ou *bloc d'activation (activation record)*.

Quand vous sauvez d'autres registres sur la pile, assurez-vous de toujours sauver et positionner bp avant de pousser les autres registres. Si vous poussez les autres registres avant d'établir bp, les offsets dans le cadre de pile changeront. Par exemple, le code suivant dérange l'ordre présenté ci-dessus:

```
FunnyProc       proc    near
                push    ax
                push    bx
                push    bp
                mov     bp, sp
FunnyProc       proc    near
                push    ax
                push    bx
                push    bp
                mov     bp, sp
                .
                .
                .
                pop     bp
                pop     bx
                pop     ax
                ret
FunnyProc       endp
                pop     bp
                pop     bx

                pop     ax
```

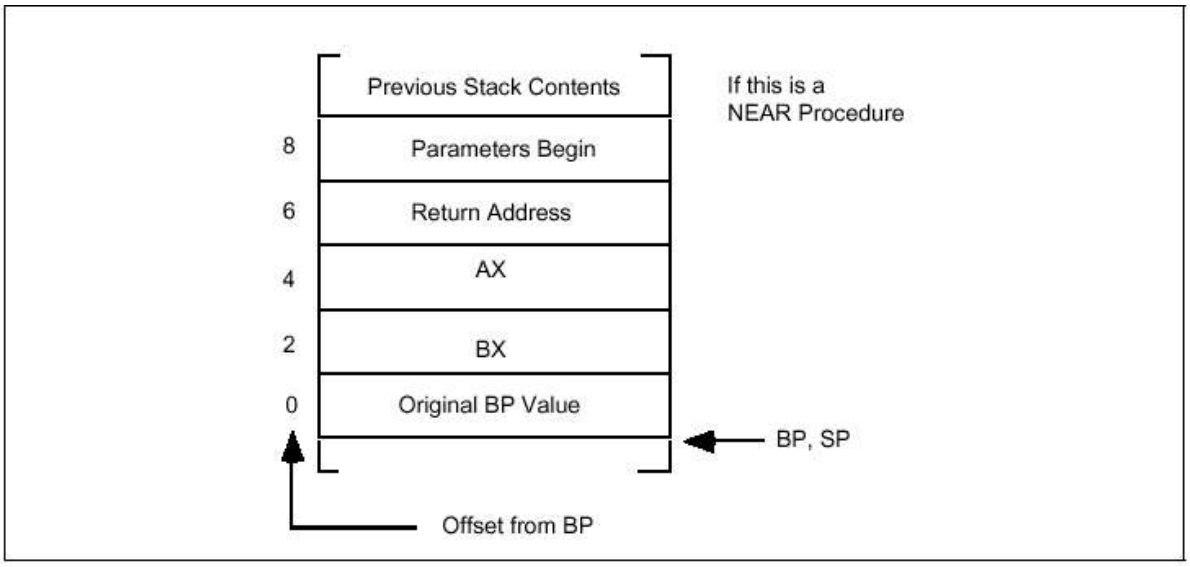
```
                ret
FunnyProc       endp
```

Comme ce code pousse ax et bx avant de pousser bp et de copier sp dans bp, ax et bx apparaissent dans le bloc d'activation avant l'adresse de retour (qui commencerait normalement à l'emplacement [bp+2]). En conséquence, la valeur de bx apparaît à l'endroit [bp+2] et la valeur de ax apparaît à l'emplacement [bp+4]. Ceci repousse l'adresse de retour et des autres paramètres plus loin vers le haut de la pile comme représenté sur la Figure 11.9.

Bien que ce soit une procédure near, les paramètres ne commencent pas avant l'offset huit dans le bloc d'activation. Si vous aviez poussé les registres ax et bx après avoir établi bp, l'offset jusqu'aux paramètres aurait été quatre (voir la Figure 11.10).

```
FunnyProc       proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                .
                .
                .

                pop     bx
                pop     ax
                pop     bp
                ret
FunnyProc       endp
```



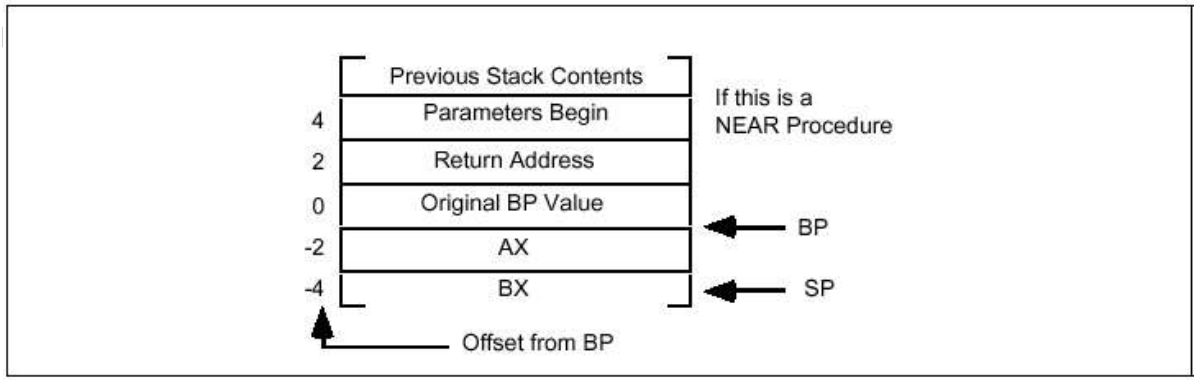


Figure 11.10 Pousser d'abord BP pour la cohérence des offsets

Par conséquent, les instructions push bp et mov bp, sp devraient être les deux premières instructions que n'importe quel sous-programme exécute quand il a des paramètres sur la pile.

L'accès aux paramètres avec des expressions comme [bp+6] peut rendre vos programmes très durs à lire et à maintenir. Si vous voulez employer des noms signicatifs, il y a plusieurs manières de procéder. Une manière de référencer des paramètres par un nom est d'utiliser des equates. Considérez la procédure suivante Pascal et son code équivalent en assembleur 80x86:

```
procedure       xyz(var i:integer; j,k:integer);
begin
                i := j+k;
end;
```

Séquence d'appel:

```
        xyz(a, 3, 4);
```

Code assembleur:

```
xyz_i           equ     8[bp]  ; Utilise des equates pour pouvoir referencer
xyz_j           equ     6[bp]  ; des noms symboliques dans le corps de
xyz_k           equ     4[bp]  ; la procédure.
xyz             proc    near
                push    bp
                mov     bp, sp
                push    es
                push    ax
                push    bx
                les     bx, xyz_i     ; Obtient l'addresse de I dans ES:BX
                mov     ax, xyz_j     ; Obtient le paramètre J
                add     ax, xyz_k     ; Additionne au paramètre K
                mov     es:[bx], ax   ; Stocke résultat dans paramètre I
                pop     bx
                pop     ax
                pop     es
                pop     bp
                ret     8
xyz             endp
```

Séquence d'appel:

```
                mov     ax, seg a     ; Ce paramètre est passé par
                push    ax            ; référence, aussi passer son
                mov     ax, offset a  ; adresse sur la pile.
                push    ax
```

```
        mov     ax, 3           ; Ceci est le second paramètre
        push    ax
        mov     ax, 4           ; Ceci est le troisième paramètre.
        push    ax
        call    xyz
```

Sur un processeur 80186 ou postérieur, vous pourriez employer le code suivant au lieu de ce qui précède:

```
        push    seg a           ; Passer l'adresse de "a" sur la
        push    offset a        ; pile.
        push    3               ; Passer second parm par val.
        push    4               ; Passer troisième parm par val.
        call    xyz
```

A l'entrée dans la procédure xyz, avant l'exécution de l'instruction les, la pile ressemble au schéma représenté sur la Figure 11.11 :

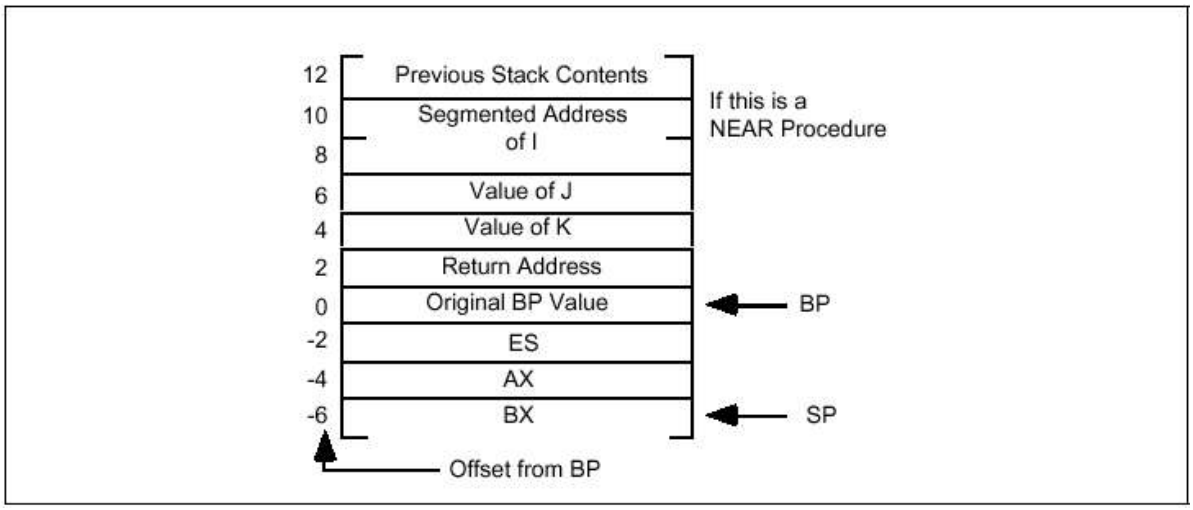


Figure 11.11 La pile de XYZ avant l'entrée dans la procédure

Puisque vous passez I par référence, vous devez pousser son adresse sur la pile. Ce code passe des paramètres de référence en utilisant desadresses segmentées de 32 bits. Notez que ce code utilise ret 8. Bien qu'il y ait trois paramètres sur la pile, le paramètre de référence I consomme quatre bytes puisque c'est une adresse far. Par conséquent il y a huit bytes de paramètres sur la pile, ce qui rend nécessaire l'instruction ret 8.

Si vous deviez passer I par référence en utilisant un pointeur near plutôt qu'un pointeur far, le code ressemblerait à ce qui suit:

```
xyz_i           equ     8[bp]   ; Utiliser des equates pour pouvoir référencer
xyz_j           equ     6[bp]   ; les noms symboliques dans le corps de
xyz_k           equ     4[bp]   ; la procédure.
xyz             proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                mov     bx, xyz_i       ; Obtenir adresse de I dans BX.
                mov     ax, xyz_j       ; Obtenir  paramètre J
                add     ax, xyz_k       ; Additionner au paramètre K
                mov     [bx], ax        ; Stocker résultat dans paramètre I
                pop     bx
                pop     ax
                pop     bp
                ret     6
```

```
xyz             endp
```

Notez que puisque l'adresse de I sur la pile est seulement de deux bytes (au lieu de quatre), cette routine extrait seulement six bytes quand elle retourne.

Séquence d'appel:

```
            mov     ax, offset a    ; Passer adresse near de a.
            push    ax
            mov     ax, 3           ; Ceci est le second paramètre
            push    ax
            mov     ax, 4           ; Ceci est le troisième paramètre.
            push    ax
            call    xyz
```

Sur un processeur 80286 ou postérieur, vous pourriez employer le code suivant au lieu de ce qui précède:

```
            push    offset a        ; Passe adresse near de a.
            push    3               ; Passe second parm par val.
            push    4               ; Passe troisième parm par val.
            call    xyz
```

Le cadre de pile pour le code ci-dessus apparaît sur la Figure 11.12.

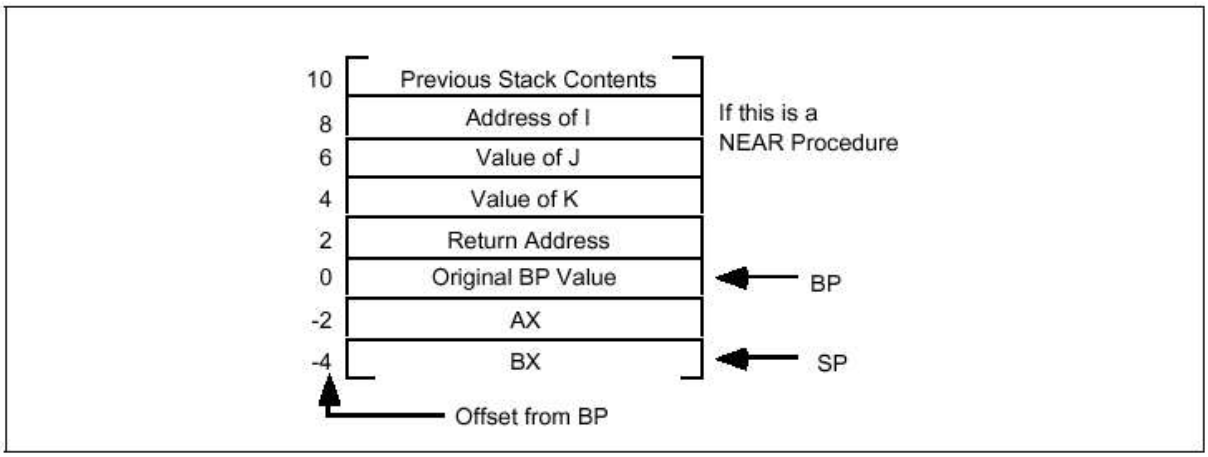


Figure 11.12 Passer des paramètres par référence avec des pointeurs near au lieu de far

En passant un paramètre par valeur-retournée ou par résultat, vous passez une adresse à la procédure, exactement comme en passant le paramètre par référence. La seule différence est que vous utilisez une copie locale de la variable à l'intérieur de la procédure plutôt que d'accéder à la variable indirectement par le pointeur. Les implémentations suivantes pour xyz montrent comment passer I par valeur-retournée et par résultat:

```
; version de xyz employant le passage par Valeur-Retournée pour le xyz_i
xyz_i           equ     8[bp]   ;Utilise equates pour pouvoir référencer
xyz_j           equ     6[bp]   ; des noms symboliques dans le corps de
xyz_k           equ     4[bp]   ; la procédure.
xyz             proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                push    cx              ; Garde une copie locale ici.
                mov     bx, xyz_i       ; Obtient adresse de I dans BX
                mov     cx, [bx]        ; Obtient copie locale du paramètre I.
                mov     ax, xyz_j       ; Obtient paramètre J
                add     ax, xyz_k       ; Additionne au paramètre K
                mov     cx, ax          ; Stocke résultat dans copie locale
                mov     bx, xyz_i       ; Obtient ptr sur I, de nouveau
```

```
                mov     [bx], cx        ; Stocke résultat.
                pop     cx
                pop     bx
                pop     ax
                pop     bp
                ret     6
xyz             endp
```

Il y a une ou deux instructions mov inutiles dans ce code. Elles sont présentes seulement pour mettre en application avec précision le passage par paramètres valeur-retournée. Il est facile d'améliorer ce code en utilisant le passage par des paramètres par résultat. Le code modifié est

```
; version de xyz employant le Passage par Résultat pour xyz_i
xyz_i           equ     8[bp]   ; Utilise equates pour pouvoir référencer
xyz_j           equ     6[bp]   ; des noms symboliques dans le corps de
xyz_k           equ     4[bp]   ; la procédure.

xyz             proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                push    cx              ; Garde une copie locale ici.
                mov     ax, xyz_j       ; Obtient paramètre J
                add     ax, xyz_k       ; Additionne au paramètre K
                mov     cx, ax          ; Stocke résultat dans copie locale
                mov     bx, xyz_i       ; Obtient ptr sur I, de nouveau
                mov     [bx], cx        ; Stocke résultat.
                pop     cx
                pop     bx
                pop     ax
                pop     bp
                ret     6
xyz             endp
```

Comme en passant des paramètres par valeur-retournée et par résultat dans des registres, vous pouvez améliorer l'exécution de ce code en utilisant une forme modifiée du passage par valeur. Considérez l'exécution suivante de xyz :

```
; version de xyz employant le passage par valeur-résultat modifié pour xyz_i
xyz_i           equ     8[bp]   ; Utilise equates pour pouvoir référencer
xyz_j           equ     6[bp]   ; des noms symboliques dans le corps de
xyz_k           equ     4[bp]   ; la procédure.
xyz             proc    near
                push    bp
                mov     bp, sp
                push    ax
                mov     ax, xyz_j       ; Obtient paramètre J
                add     ax, xyz_k       ; Additionne au paramètre K
                mov     xyz_i, ax       ; Stocke résultat dans copie locale
                pop     ax
                pop     bp
                ret     4               ; Notez que nous n'extrayons pas parm I.
xyz             endp
```

La séquence d'appel pour ce code est :

```
                push    a               ; Passe le valeur de a à xyz.
                push    3               ; Passe second paramètre par val.
                push    4               ; Passe troisième paramètre par val.
                call    xyz
                pop     a
```

Note qu'une version par passage par résultat ne serait pas économique puisque vous devez pousser *quelque chose* sur la pile faire de la place pour la copie locale de I à l'intérieur de xyz. Vous pouvez alors aussi bien pousser la valeur de a à

l'entrée quoique la procédure xyz l'ignore. Cette procédure extrait seulement *quatre* bytes de la pile à la sortie. Ceci laisse la valeur du paramètre I sur la pile pour que le code appelant puisse la *stocker* à la destination appropriée.

Pour passer un paramètre par nom sur la pile, vous poussez simplement l'adresse du thunk. Considérez le code suivant en pseudo-Pascal:

```
procedure swap(name Item1, Item2:integer);
var temp:integer;
begin

      temp := Item1;
      Item1 := Item2;
      Item2 := Temp;
end;
```

Si swap est une procédure near, le code 80x86 pour cette procédure pourrait ressembler à ce qui suit (notez que ce code a été légèrement optimisé et ne suit pas la séquence exacte donnée ci-dessus):

```
; swap-         échange deux paramètres passés par nom sur la pile.
; Item1 est passé à l'adresse [bp+6], Item2 est passé
; à l'adresse [bp+4]

wp              textequ         <word ptr>
swap_Item1      equ             [bp+6]
swap_Item2      equ             [bp+4]
swap            proc            near
                push            bp
                mov             bp, sp
                push            ax              ; Préserve valeur de temp.
                push            bx              ; Préserve bx.
                call            wp swap_Item1   ; Obtient adrs de Item1.
                mov             ax, [bx]        ; Sauve dans temp (AX).
                call            wp swap_Item2   ; Obtient adrs de Item2.
                xchg            ax, [bx]        ; Echange temp <-> Item2.
                call            wp swap_Item1   ; Obtient adrs de Item1
                mov             [bx], ax        ; Sauve temp dans Item1.
                pop             bx              ; Restaure bx.
                pop             ax              ; Restaure ax.
                ret             4               ; Retourne and extrait Item1/2.
swap            endp
```

Des exemples d'appels à swap suivent:

```
; swap(A[i], i) -- version 8086.
                lea             ax, thunk1
                push            ax
                lea             ax, thunk2
                push            ax
                call            swap

; swap(A[i],i) -- version 80186 & postérieur.
                push            offset thunk1
                push            offset thunk2
                call            swap
                .
                .
                .
; Note: ce code suppose que A est un tableau d'entiers de deux octets.

thunk1          proc            near
                mov             bx, i
                shl             bx, 1
                lea             bx, A[bx]
                ret
thunk1          endp
```

```
thunk2          proc            near
                lea             bx, i
                ret
thunk2          endp
```

Le code ci-dessus suppose que les thunks sont des procs near qui résident dans le même segment que la routine swap. Si les thunks sont des procédures lointaines l'appelant doit passer des adresses lointaines sur la pile et la routine d'échange doit manoeuvrer des adresses lointaines. L'implémentation suivante de swap, thunk1, et thunk2 démontrent ceci

```
; swap-         échange deux paramètres passés par nom sur la pile.
; Item1 est passé à l'adresse [bp+10], Item2 est passé
; à l'adresse [bp+6]

swap_Item1      equ     [bp+10]
swap_Item2      equ     [bp+6]
dp              textequ <dword ptr>

swap            proc    far
                push    bp
                mov     bp, sp
                push    ax              ; Préserve valeur temp.
                push    bx              ; Préserve bx.
                push    es              ; Préserve es.
                call    dp swap_Item1   ; Obtient adrs de Item1.
                mov     ax, es:[bx]     ; Sauve dans temp (AX).
                call    dp swap_Item2   ; Obtient adrs de Item2.
                xchg    ax, es:[bx]     ; Echange temp <-> Item2.
                call    dp swap_Item1   ; Obtient adrs de Item1.
                mov     es:[bx], ax     ; Sauve temp dans Item1.
                pop     es              ; Restaure es.
                pop     bx              ; Restaure bx.
                pop     ax              ; Restaure ax.
                ret     8               ; Retourne et pop Item1,
                                        ; Item2.
swap            endp
```

Des exemples d'appels à swap suivent:

```
; swap(A[i], i) -- version 8086.

                mov     ax, seg thunk1
                push    ax
                lea     ax, thunk1
                push    ax
                mov     ax, seg thunk2
                push    ax
                lea     ax, thunk2
                push    ax
                call    swap

; swap(A[i],i) -- version 80186 & postérieur.

                push    seg thunk1
                push    offset thunk1
                push    seg thunk2
                push    offset thunk2
                call    swap
                .
                .
                .

; Note: ce code suppose  que A est un tableau d'entiers de deux bytes.
; Notez aussi que nous ne savons pas quel(s) segment(s) contient
```

```
; A et I.

thunk1          proc    far
                mov     bx, seg A       ; Il faut retourner seg A dans ES.
                push    bx              ; Sauve pour plus tard.
                mov     bx, seg i       ; Il faut segment de I pour
                mov     es, bx          ; y accéder.
                mov     bx, es:i        ; Obtient valeur de I.
                shl     bx, 1
                lea     bx, A[bx]
                pop     es              ; Retournr segment de A[I] dans es.
                ret
                thunk1  endp
thunk2          proc    near
                mov     bx, seg i       ; Il faut retourner seg de I dans es.
                mov     es, bx
                lea     bx, i
                ret
thunk2          endp
```

Le passage de paramètres par évaluation paresseuse est laissé pour les projets de programmation.

Une information suplémentaire sur les blocs d'activation et les cadres de pile apparaît plus tard dans ce chapitre dans la section sur les variables locales.

11.5.10 Le passage des paramètres dans le code lui-même

Un autre endroit où vous pouvez passer des paramètres est dans le code immédiatement après l'instruction d'appel. La routine print dans la distribution de la Bibliothèque Standard de l'UCR en fournit un excellent exemple:

```
                print
                byte        "Ce paramètre se trouve dans le code lui-même.", 0
```

Normalement, une routine retourne le contrôle à la première instruction juste après l'instruction call. Si cela devait se produire ici, le 80x86 essayerait d'interpréter le code d'ASCII pour "Ce paramètre…" comme une instruction. Ceci produirait des résultats indésirables. Heureusement, vous pouvez sauter au-dessus de cette chaîne en revenant du sous-programme.

Alors, comment accédez-vous à ces paramètres? Facile. L'adresse de retour sur la pile pointe sur eux. Considérez l'implémentation suivante de print:.

```
MyPrint         proc    near
                push    bp
                mov     bp, sp
                push    bx
                push    ax
                mov     bx, 2[bp]               ; Charge adresse de retour dans BX
PrintLp:        mov     al, cs:[bx]             ; Obtient caractère suivant
                cmp     al, 0                   ; Vérifie fin de chaîne
                jz      EndStr
                putc                            ; Si pas fin, imprime ce car
                inc     bx                      ; Passe au caractère suivant
                jmp     PrintLp
EndStr:         inc bx                          ; Pointe au premier byte après zéro
                mov     2[bp], bx       ; Sauve comme nouvelle adresse de retour
                pop     ax
                pop     bx
                pop     bp
                ret
MyPrint         endp
```

Cette procédure commence par pousser tous les registres affectés sur la pile. Elle cherche alors l'adresse de retour, à l'offset 2[BP] et imprime chaque caractère successif jusqu'à ce qu'elle rencontre un byte zéro. Notez la présence du préfixe de surcharge du segment cs: dans l'instruction mov al, cs:[bx]. Puisque les données viennent du segment de code, ce préfixe garantit que MyPrint cherche les données-caractères dans le segment approprié. En rencontrant le byte zéro, MyPrint pointe bx sur le premier byte après le zéro. C'est l'adresse de la première instruction suivant le byte de terminaison zéro. Le CPU utilise cette valeur comme nouvelle adresse de retour. Maintenant l'exécution de l'instruction de retour renvoie le contrôle à l'instruction suivant la chaîne.

Le code ci-dessus fonctionne impec si MyPrint est une procédure near. Si vous devez appeler MyPrint d'un segment différent vous devrez créer une procédure far. Naturellement, la différence principale est qu'une adresse retour far sera sur la pile à ce moment - vous devrez employer un pointeur far plutôt qu'un pointeur near. L'implémentation suivante de MyPrint traite ce cas.

```
MyPrint         proc    far
                push    bp
                mov     bp, sp
                push    bx              ; Préserve ES, AX, et BX
                push    ax
                push    es
                les     bx, 2[bp]       ; Charge adresse de retour dans ES:BX
PrintLp:        mov     al, es:[bx]     ; Obtient caractère suivant
                cmp     al, 0           ; Vérifie fin de chaîne
                jz      EndStr
                putc                    ; Si pas fin, imprime ce car
                inc     bx              ; Passe au caractère suivant
                jmp     PrintLp
EndStr:         inc     bx              ; Pointe sur premier byte après zéro
                mov     2[bp], bx       ; Sauve comme nouvelle adresse de retour
                pop     es
                pop     ax
                pop     bx
                pop     bp
                ret
MyPrint         endp
```

Notez que ce code ne stocke pas es de nouveau dans l'emplacement [bp+4]. La raison en est tout à fait simple - es ne change pas pendant l'exécution de cette procédure; le stockage de es dans l'emplacement [bp+4] ne changerait pas la valeur à cet endroit. Vous noterez que cette version de MyPrint va chercher chaque caractère à l'emplacement es:[bx] au lieu de cs:[bx]. C'est parce que la chaîne que vous imprimez est dans le segment de l'appelant, qui pourrait ne pas être le même segment que celui qui contient MyPrint.

A part montrer comment passer des paramètres dans le code, la routine MyPrint démontre également un autre concept: les *paramètres de longueur variable*. La chaîne après call peut être de n'importe quelle longueur réelle. Le byte de terminaison zéro marque la fin de la liste de paramètres. Il y a deux manières faciles de manipuler des paramètres de longueur variable. Soit on utilise une valeur de terminaison spéciale (comme zéro), soit on peut passer une valeur de longueur spéciale qui indique au sous-programme combien de paramètres on passe. Les deux méthodes ont leurs avantages et inconvénients. L'utilisation d'une valeur spéciale pour terminer une liste de paramètre exige que vous choisissiez une valeur qui n'apparaît jamais dans la liste. Par exemple, MyPrint utilise zéro comme valeur de terminaison, aussi elle ne peut imprimer le caractère NULL (dont le code ASCII est zéro). Parfois ce n'est pas une limitation. Spécifier un paramètre spécial de longueur est un autre mécanisme que vous pouvez employer pour passer une liste de paramètre de longueur variable. Bien que ceci n'exige pas de code spécial ni ne limitela gamme des valeurs possibles qui peuvent être passées à un sous-programme, définir le paramètre de longueur et maintenirt le code en résultant peut être un vrai cauchemar[555555].

Bien que le passage des paramètres dans le code est une manière idéale de passer des listes de paramètre de longueur variable, vous pouvez passer les listes de paramètre de longueur fixe de la même façon. L'espace de code est un excellent endroit pour passer des constantes (comme les constantes de chaîne passées à MyPrint) et des paramètres de référence. Considérez le code suivant qui attend trois paramètres par référence:

[5] Surtout si la liste de paramètres change fréquemment.

Séquence d'appel:

```
            call      AddEm
            word      I,J,K
```

Procédure:

```
AddEm proc near
               push    bp
               mov     bp, sp
               push    si
               push    bx
               push    ax
               mov     si, [bp+2]            ; Obtient adresse de retour
               mov     bx, cs:[si+2]         ; Obtient adresse de J
               mov     ax, [bx]              ; Obtient valeur de J
               mov     bx, cs:[si+4]         ; Obtient adresse de K
               add     ax, [bx]              ; Additionne avec valeur de K
               mov     bx, cs:[si]           ; Obtient adresse de I
               mov     [bx], ax              ; Stocke résultat
               add     si, 6                 ; Saute après parms
               mov     [bp+2], si            ; Sauve adresse de retour
               pop     ax
               pop     bx
               pop     si
               pop     bp
               ret
AddEm          endp
```

Ce sous-programme additionne J et K ensemble et stocke le résultat dans I. Notez que ce code utilise des pointeurs 16 bits near pour passer les adresses de I, J et K à AddEm. Par conséquent, I, J et K doivent être dans le segment de données courant. Dans l'exemple ci-dessus, AddEm est une procédure proche. Eut-ce été une procédure lointaine qu'elle aurait dû chercher un pointeur de quatre bytes sur la pile au lieu d'un pointeur de deux bytes. Ce qui suit est une version far d'AddEm:

```
AddEm          proc    far
               push    bp
               mov     bp, sp
               push    si
               push    bx
               push    ax
               push    es
               les     si, [bp+2]            ; Obtient adrs far ret dans es:si
               mov     bx, es:[si+2]         ; Obtient adresse de J
               mov     ax, [bx]              ; Obtient valeur de J
               mov     bx, es:[si+4]         ; Obtient adresse de K
               add     ax, [bx]              ; Additionne valeur de K
               mov     bx, es:[si]           ; Obtient adresse de I
               mov     [bx], ax              ; Stocke résultat
               add     si, 6                 ; Saute après parms
               mov     [bp+2], si            ; Sauve adresse de retour
               pop      es
               pop     ax
               pop     bx
               pop     si
               pop     bp
               ret
AddEm          endp
```

Dans les deux versions d'AddEm, les pointeurs sur I, J, et K passés dans le code sont les pointeurs proches. Les deux versions supposent qu'I, J, et K sont tous dans le segment de données courant. Il est possible de passer des pointeurs far à ces variables, ou même des pointeurs near sur certaines et des pointeurs far sur d'autres, dans le code. L'exemple suivant n'est pas si ambitieux, c'est une procédure near qui attend des pointeurs far, mais il montre certaines des différences majeures. Pour des exemples suplémentaires, voyez les exercices.

Séquence d'appel:

```
                call    AddEm
                dword   I,J,K
Code:
AddEm           proc    near
                push    bp
                mov     bp, sp
                push    si
                push    bx
                push    ax
                push    es
                mov     si, [bp+2]      ; Obtient adrs retour near dans si
                les     bx, cs:[si+2]   ; Obtient adresse de J dans es:bx
                mov     ax, es:[bx]     ; Obtient valeur de J
                les     bx, cs:[si+4]   ; Obtient adresse de K
                add     ax, es:[bx]     ; Additionne valeur de K
                les     bx, cs:[si]     ; Obtient adresse de I
                mov     es:[bx], ax     ; Stocke résultat
                add     si, 12          ; Saute après parms
                mov     [bp+2], si      ; Sauve adresse de retour
                pop     es
                pop     ax
                pop     bx
                pop     si
                pop     bp
                ret
AddEm           endp
```

Notez qu'il y a 12 bytes de paramètres dans le flux de code, cette fois. C'est pourquoi ce code contient l'instruction add si, 12 plutôt que add si, 6 qui apparaissait dans les autres versions.

Dans les exemples donnés jusqu'ici, MyPrint s'attend à un passage de paramètre par valeur, elle imprime les caractères effectifs suivant l'appel et AddEm attend trois paramètres passés par référence - leurs adresses suivent dans le code. Naturellement, vous pouvez également passer des paramètres par valeur-retournée, par résultat, par nom ou par évaluation paresseuse dans le code de la même manière. Le prochain exemple est une modification d'AddEm qui utilise le passage par résultat pour I, le passage par valeur-retournée pour J et le passage par nom pour K. Cette version est légèrement differerent dans la mesure où elle modifie J aussi bien que I, afin de justifier l'utilisation du paramètre par valeur-retournée.

```
; AddEm(Result I:integer; ValueResult J:integer; Name K);
;
;       Calcule         I:= J;
;                       J := J+K;
;
; Présume qur tous les pointeurs dans le code sont des pointeurs near.

AddEm           proc    near
                push    bp
                mov     bp, sp
                push    si                      ; Pointeur sur le bloc de paramètres.
                push    bx                      ; Pointeur général.
                push    cx                      ; Valeur temp pour I.
                push    ax                      ; Valeur temp pour J.
                mov     si, [bp+2]              ; Obtient adrs retour near dans si
                mov     bx, cs:[si+2]           ; Obtient adresse de J dans bx
                mov     ax, es:[bx]             ; Crée une copie locale de J.
                mov     cx, ax                  ; Exécute I:=J;
                call    word ptr cs:[si+4]      ; Appelle thunk pour avoir adrs de K
                add     ax, [bx]                ; Calcule J := J + K
                mov     bx, cs:[si]             ; Obtient adresse de I
                mov     [bx], cx                ; et stocke I.
```

```
                mov     bx, cs:[si+2]           ; Obtient adresse de J
                mov     [bx], ax                ; et stocke valeur de J.
                add     si, 6                   ; Saute après parms
                mov     [bp+2], si              ; Sauve adresse de retour
                pop     ax
                pop     cx
                pop     bx
                pop     si
                pop     bp
                ret
AddEm           endp
```

Exemples de séquences d'appel:

```
; AddEm(I,J,K)

                call    AddEm
                word    I,J,KThunk

; AddEm(I,J,A[I])

                call    AddEm
                word    I,J,AThunk
                .
                .
                .
KThunk          proc    near
                lea     bx, K
                ret
KThunk          endp

AThunk          proc    near
                mov     bx, I
                shl     bx, 1
                lea     bx, A[bx]
                ret
AThunk          endp
```

Note: si vous aviez passé I par référence, plutôt que par résultat, dans cet exemple, l'appel

AddEm(I, J, A[i])

aurait produit des résultats différents. Pouvez-vous expliquer pourquoi?

Le passage de paramètres dans le jet de code vous permet d'accomplir quelques tâches vraiment intelligentes. L'exemple suivant est considérablement plus complexe que les autres dans cette section, mais il démontre la puissance du passage de paramètres dans le code et, en dépit de la complexité de cet exemple, comment cela peut simplifier vos tâches de programmation.

Les deux routines suivantes implémentent une instruction for/next, semblable à celle du BASIC, en assembleur. La séquence d'appel pour ces routines est la suivante:

```
                call    ForStmt
                word    «VarControlBoucle», «ValDebut», «ValFin»
                .
                .
                « instructions du corps de la boucle»
                .
                .
                call    Next
```

Ce code définit la variable de contrôle de la boucle (dont vous passez l'adresse near comme premier paramètre, par référence) à la valeur de début (passée par valeur comme deuxième paramètre). Il commence alors l'exécution du corps de boucle. En exécutant l'appel à Next, ce programme incrémente la variable de contrôle de la boucle, puis la compare à la valeur de fin. Si elle est inférieure ou égale à la valeur de fin, le contrôle revient au début du corps de boucle (la première instruction après la directive word). Sinon il continue l'exécution avec la première instruction après l'appel à Next.

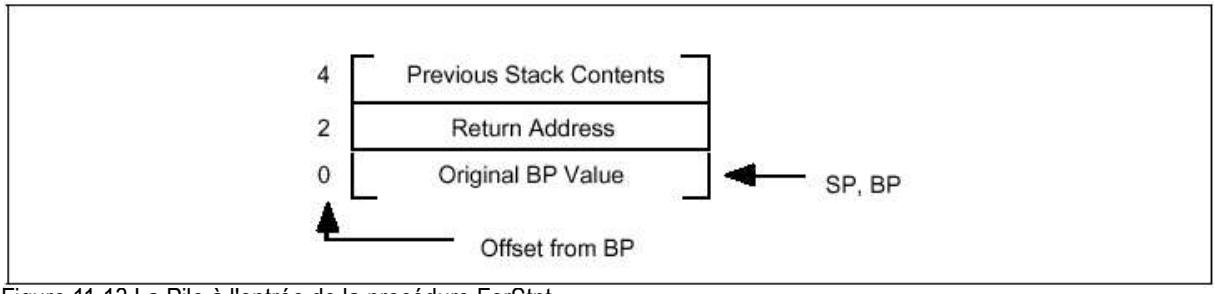


Figure 11.13 La Pile à l'entrée de la procédure ForStpt

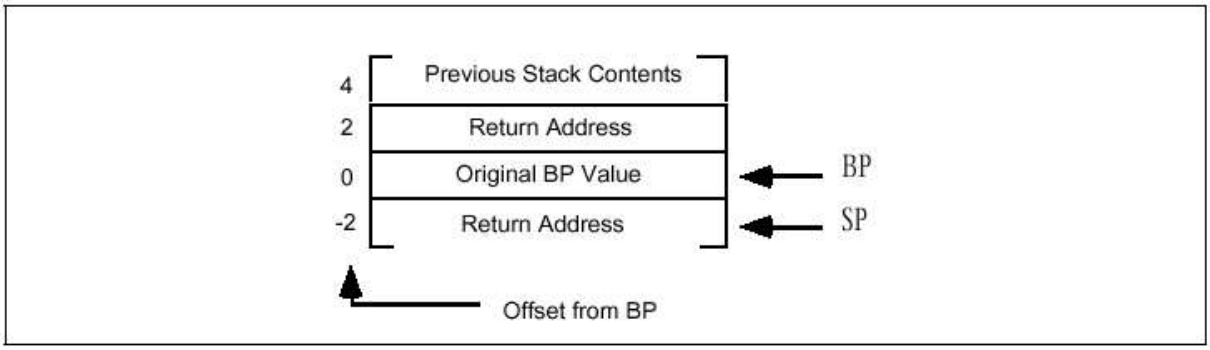


Figure 11.14 La pile juste avant de quitter la procédure ForStmt

Maintenant vous vous demandez probablement, "Comment diable le contrôle est-il transféré au début du corps de la boucle ?" Après tout, il n'y a aucune étiquette à cette instruction et il n'y a aucune instruction de branchement qui saute à la première instruction après la directive word. Bien, il s'avère que vous pouvez faire ceci avec une manipulation un peu osée de la pile. Considérez à quoi ressemblera la pile à l'entrée dans la routine ForStmt, après avoir poussé bp sur la pile (voir la Figure 11.13).

Normalement, la routine ForStmt devrait extraire bp et retourner avec une instruction ret, qui enlèverait le bloc d'activation de ForStmt de la pile. Supposez qu'au lieu de cela, ForStmt exécute les instructions suivantes:

```
add     word ptr 2[b], 2        ; Saute les paramètres.
push    [bp+2]                  ; Fait une copie de l'adrs retour.
mov     bp, [bp]                ; Restaure la valeur de bp.
ret                             ; Retourne à l'appelant.
```

Juste avant l'instruction ret ci-dessus, la pile a les entrées représentées sur la Figure 11.14.

En exécutant l'instruction ret, ForStmt reviendra à l'adresse de retour appropriée *mais elle laissera son bloc d'activation sur la pile !*

Après exécution des instructions dans le corps de boucle, le programme appelle la routine Next. A l'entrée initiale dans Next (et après avoir établi bp), la pile contient les entrées apparaissant sur la Figure 11.15[6666]

La chose importante à voir ici est que l'adresse de retour de ForStmt, qui pointe sur la première instruction après la directive word, est toujours sur la pile et disponible pour Next à l'offset [bp+6]. Next peut utiliser cette adresse de retour pour accéder aux paramètres et revenir à l'endroit approprié, si nécessaire. Next incréments la variable de contrôle de la boucle et la compare à la valeur de fin. Si la valeur de la variable de contrôle de la boucle est moindre que la valeur de fin, Next extrait son adresse de retour de la pile et retourne par l'adresse de retour de ForStmt. Si la variable de contrôle de la boucle est plus grande que la valeur de fin, Next retourne par sa propre adresse de retour et enlève le bloc d'activation de ForStmt de la pile. Ce qui suit est le code pour Next et ForStmt:

[6] En supposant que la boucle ne pousse rien sur la pile ou n'extrait rien de la pile. Si l'un ou l'autre cas se produit, la boucle ForStmt/Next ne fonctionnerait pas correctement.

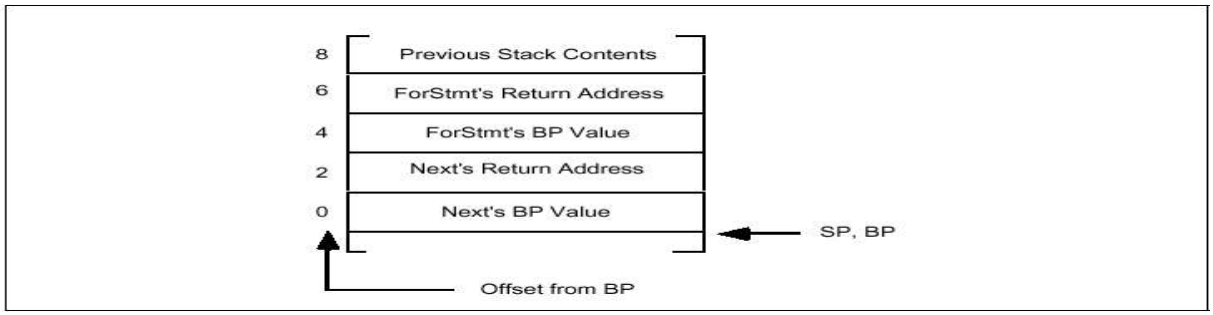


Figure 11.15 La pile à l'entrée de la procédure Next

```
                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list
dseg            segment para public 'data'
I               word    ?
J               word    ?
dseg            ends

cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

wp              textequ <word ptr>

ForStmt         proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                mov     bx, [bp+2]              ; Obtient adresse de retour
                mov     ax, cs:[bx+2]           ; Obtient valeur début
                mov     bx, cs:[bx]             ; Obtient adresse de var
                mov     [bx], ax                ; var := valeur début
                add     wp [bp+2], 6            ; Saute les paramètres
                pop     bx
                pop     ax
                push    [bp+2]                  ; Copie adresse de retour
                mov     bp, [bp]                ; Restaure bp
                ret                             ; Laisse Bloc d'Act sur la pile
ForStmt         endp

Next            proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                mov     bx, [bp+6]              ; adrs de retour de ForStmt
                mov     ax, cs:[bx-2]           ; Valeur de fin
                mov     bx, cs:[bx-6]           ; Ptr sur la var de ctrl de boucle
                inc     wp [bx]                 ; Incrémente ctrl de boucle
                cmp     ax, [bx]                ; Val de fin < ctrl de boucle?
                jl      QuitLoop

; Si nous arrivons ici, la variable de contrôle de boucle est inférieure
; ou égale à la valeur de fin. Alors, il faut repéter la boucle encore une
; fois. Copie l'adresse de retour de ForStmtover par dessus la nôtre et
; puis retourne, en laissant le bloc d'activation de ForStmt intact.
```

```
                mov     ax, [bp+6]              ; Adresse de retour de ForStmt
                mov     [bp+2], ax              ; Ecrase notre adresse de retour
                pop     bx
                pop     ax
                pop     bp                      ; Retourne au début corps de bcle
                ret

; Si nous arrivons ici, la variable de contrôle de boucle est supérieure à
; la valeur de fin, alors il faut quitter la boucle (en retournant à
; l'adresse de retour de Next) and enlever le bloc d'activation de ForStmt.

QuitLoop:       pop     bx
                pop     ax
                pop     bp
                ret     4
Next            endp

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit
                call    ForStmt
                word    I,1,5
                call    ForStmt
                word    J,2,4
                printf
                byte    "I=%d, J=%d\n",0
                dword   I,J
                call    Next                    ; Fin de la boucle J
                call    Next                    ; Fin de la boucle I
                print
                byte    "Fini!",cr,lf,0
Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

Le code d'exemple dans le programme principal montre que ces boucles for s'emboîtent exactement comme on peut s'y attendre dans un langage de niveau élevé comme le BASIC, le Pascal, ou le C. Bien sûr, ce n'est pas une manière particulièrement efficace de construire une boucle for en assembleur. C'est beaucoup plus lent qu'en utilisant les techniques standard de génération de boucle (voir les "Boucles" au chapitre 10 pour plus de détails). Naturellement, si vous ne vous souciez pas de la vitesse, ceci est une manière parfaitement régulière d'implémenter une boucle. Ellel est certainement plus facile à lire et à comprendre que les méthodes traditionnelles pour créer une boucle for. Pour une autre implémentation (plus efficace) de la boucle for , voyez les macros ForLp au Chapitre Huit (voir "Un Exemple de Macro pour Implémenter des Boucles For" à la section , chapitre 8).

Le flux de code est un endroit très commode pour passer des paramètres. La Bibliothèque Standard UCR fait une utilisation considérable de ce mécanisme de passage de paramètre pour rendre facile l'appel à certaines routines. Printf est, peut-être, l'exemple le plus complexe, mais d'autres exemples (particulièrement dans la bibliothèque de chaînes) abondent.

Malgré la commodité, il y a quelques inconvénients à passer des paramètres dans le code. D'abord, si vous ne fournissez pas le nombre exact de paramètres que la procédure exige, le sous-programme ne saura plus où il en est. Considérez la routine print de la Bibliothèque Standard UCR. Elle imprime une chaîne des caractères jusqu'à un byte de terminaison zéro et puis renvoie le contrôle à la première instruction après le byte de terminaison zéro. Si vous oubliez le byte de terminaison zéro, la routine print imprime gentiment les bytes suivants d'opcode comme des caractères ASCII jusqu'à ce qu'elle trouve un byte zéro. Puisque les bytes zéro apparaissent souvent au milieu d'une instruction, la routine print pourrait retourner au milieu d'une autre instruction. Ceci plantera probablement la machine. L'insertion d'un zéro supplémentaire, qui se produit plus souvent que vous pourriez penser, est un autre problème que les programmeurs rencontrent avec la routine print. En ce cas, la routine print retournerait en rencontrant le premier byte zéro et essayerait d'exécuter les caractères ASCII suivants comme du code machine. De nouveau, ceci plante habituellement la machine.

Un autre problème avec le passage des paramètres dans le code est que cela prend plus longtemps pour accéder à de tels paramètres. Le passage des paramètres dans les registres, dans des variables globales, ou sur la pile est légèrement plus efficace, particulièrement dans des routines courtes. Néanmoins, l'accès aux paramètres dans le code n'est pas extrêmement lent, aussi la commodité de tels paramètres peut être supérieure au coût. En outre, beaucoup de routines (print est un bon exemple) sont si lentes de toute façon que quelques micro-secondes supplémentaires ne feront pas la différence.

11.5.11 Le passage de paramètres via un bloc de paramètres

Une autre manière de passer des paramètres dans la mémoire est par *un bloc de paramètres*. Un bloc de paramètres est un ensemble d'emplacements de mémoire contigus contenant les paramètres. Pour accéder à de tels paramètres, vous passez au sous-programme un pointeur sur le bloc de paramètres. Considérez le sous-programme de la section précédente qui additionne J et K ensemble, stockant le résultat dans I; le code qui passe ces paramètres via un bloc de paramètres pourrait être

```
Séquence d'appel:
ParmBlock       dword   I
I               word    ?               ; I, J et K doivent apparaître dans
J               word    ?               ; cet ordre.
K               word    ?
                .
                .
                .
                les     bx, ParmBlock
                call    AddEm
                .
                .
                .
AddEm           proc    near
                push    ax
                mov     ax, es:2[bx]    ; Obtient valeur de J
                add     ax, es:4[bx]    ; Additionne valeur de K
                mov     es:[bx], ax     ; Stocke résultat dans I
                pop     ax
                ret
AddEm           endp
```

Notez que vous devez assigner les trois paramètres dans des emplacements contigus de mémoire.

Cette forme de passage de paramètres marche bien pour passer plusieurs paramètres par référence, parce que vous pouvez initialiser des pointeurs sur les paramètres directement dans l'assembleur. Par exemple, supposez que vous voulez créer une routine rotate à laquelle vous passez quatre paramètres par référence. Cette routine copierait le deuxième paramètre dans le premier, le troisième dans le second, le quatrième dans le troisième et le premier dans le quatrième. Une manière facile d'accomplir ceci en assembleur est :

```
; Rotate-       À l'entrée, BX pointe sur un bloc de paramètres dans le segment
;               données qui pointe sur quatre pointeurs far. Ce code exécute
;               une rotation des données référencées par ces pointeurs.

Rotate  proc    near
```

```
                push    es              ; Il faut préserver ces
                push    si              ; registres
                push    ax
                les     si, [bx+4]      ; Obtient ptr sur 2ème var
                mov     ax es:[si]      ; Obtient sa valeur
                les     si, [bx]        ; Obtient ptr sur 1ère var
                xchg    ax, es:[si]     ; 2ème->1ère, 1ère->ax
                les     si, [bx+12]     ; Obtient ptr sur 4ème var
                xchg    ax, es:[si]     ; 1ère->4ème, 4ème->ax
                les     si, [bx+8]      ; Obtient ptr sur 3ème var
                xchg    ax, es:[si]     ; 4ème->3ème, 3ème->ax
                les     si, [bx+4]      ; Obtient ptr sur 2ème var
                mov     es:[si], ax     ; 3ème -> 2ème
                pop     ax
                pop     si
                pop     es
                ret
Rotate          endp
```

Pour appeler cette routine, vous lui passez un pointeur sur un groupe de quatre pointeurs far dans le registre bx. Par exemple, supposez que vous vouliez faire tourner les premiers éléments de quatre tableaux différents, les deuxièmes éléments de ces quatre tableaux, et les troisième éléments de ces quatre tableaux. Vous pourriez faire ceci avec le code suivant:

```
                lea     bx, RotateGrp1
                call    Rotate
                lea     bx, RotateGrp2
                call    Rotate
                lea     bx, RotateGrp3
                call    Rotate
                .
                .
                .
RotateGrp1      dword   ary1[0], ary2[0], ary3[0], ary4[0]
RotateGrp2      dword   ary1[2], ary2[2], ary3[2], ary4[2]
RotateGrp3      dword   ary1[4], ary2[4], ary3[4], ary4[4]
```

Notez que le pointeur sur le bloc de paramètres est lui-même un paramètre. Les exemples de cette section passent ce pointeur dans les registres. Cependant, vous pouvez passer ce pointeur dans n'importe quel endroit où vous passeriez un autre paramètre par référence - dans des registres, dans des variables globales, sur la pile, dans le code, même dans un autre bloc de paramètres ! De telles variations sur ce thème, cependant, seront laissées à votre propre imagination. Comme avec n'importe quel paramètre, le meilleur emplacement pour passer un pointeur à un bloc de paramètre est dans les registres. Ce texte adoptera généralement cette politique.

Bien que les programmeurs débutants en assembleur utilisent rarement des blocs de paramètre, ceux-ci ont certainement leur place. Certaines des fonctions du BIOS du PC IBM et de MS-DOS utilisent ce mécanisme de passage de paramètres. Les blocs de paramètres, puisque vous pouvez initialiser leurs valeurs pendant l'assemblage (avec byte, word, etc.), fournissent une manière rapide et efficace de passer des paramètres à une procédure.

Naturellement, vous pouvez passer des paramètres par valeur, référence, valeur-retournée, résultat, ou par nom dans un bloc de paramètres. Le morceau de code suivant est une modification de la procédure rotate ci-dessus où le premier paramètre est passé par valeur (sa valeur apparaît à l'intérieur du bloc de paramètre), le second est passé par référence, le troisième par valeur-retournée, et le quatrième par nom (il n'y a aucun passage par résultat puisque rotate a besoin de lire et d'écrire toutes les valeurs). Pour la simplicité, ce code utilise des pointeurs near et suppose que toutes les variables apparaissent dans le segment de données:

```
; Rotate-       À l'entrée, BX pointe sur un bloc de paramètres dans le segment
;               données qui pointe sur quatre pointeurs. Le premier est un
;               paramètre par valeur, le second est passé par référence,
;               le troisième est passé par valeur-retournée, le quatrième
;               est passé par nom.
Rotate          proc    near
```

```
                push    si              ;Utilisé pour accéder aux parms ref
                push    ax              ;Temporaire
                push    bx              ;Utilisé par parm passé par nom
                push    cx              ;Copie locale du parm val/ret
                mov     si, [di+4]      ;Obtient une copie du parm val/ret
                mov     cx, [si]
                mov     ax, [di]        ;Obtient 1er parm (valeur)
                call    word ptr [di+6] ;Obtient ptr sur 4ème var
                xchg    ax, [bx]        ;1ère->4ème, 4ème->ax
                xchg    ax, cx          ;4ème->3ème, 3ème->ax
                mov     bx, [di+2]      ;Obtient adrs du 2ème parm (ref)
                xchg    ax, [bx]        ;3ème ->2ème, 2ème ->ax
                mov     [di], ax        ;2ème ->1er
                mov     bx, [di+4]      ;Obtient ptr sur parm val/ret
                mov     [bx], cx        ;Sauve parm val/ret.
                pop     cx
                pop     bx
                pop     ax
                pop     si
                ret
Rotate          endp
```

Un exemple raisonnable d'un appel à cette routine pourrait être:

```
I               word    10
J               word    15
K               word    20

RotateBlk                       word                    25, I, J, KThunk
                .
                .
                .
                lea     di, RotateBlk
                call    Rotate
                .
                .
                .
KThunk          proc    near
                lea     bx, K
                ret
KThunk          endp
```

11.6 Résultats de fonction

Les fonctions renvoient un résultat, qui n'est rien d'autre qu'un paramètre de résultat. En assembleur, il y a très peu de différences entre une procédure et une fonction. C'est probablement pourquoi il n'y a pas de directives "func" ou "endf". Les fonctions et les procédures sont habituellement différentes dans les HLL, les appels de fonctions apparaissent seulement dans des expressions, les appels de procédures comme instructions[777]. L'assembleur ne les distingue pas.

Vous pouvez renvoyer des résultats de fonction aux mêmes endroits où vous passez et renvoyez des paramètres. Généralement, cependant, une fonction renvoie seulement une valeur unique (ou une structure de données unique) comme résultat de fonction. Les méthodes et les emplacements utilisés pour renvoyer des résultats de fonction forment le sujet des trois sections suivantes.

11.6.1 Retour de résultats de fonction dans un registre

Comme pour les paramètres, les registres du 80x86 sont le meilleur endroit pour renvoyer des résultats de fonction. La routine Getc de la Bibliothèque Standard UCR est un bon exemple d'une fonction qui renvoie une valeur dans un des

[7] C fait exception à cette règle. Les procédures et les fonctions de C sont toutes appelées fonctions. PL/I est une autre exception. Dans PL/I, elles sont toutes appelées procedures.

registres du CPU. Elle lit un caractère du clavier et renvoie le code ASCII pour ce caractère dans le registre al. Généralement, les fonctions renvoient leurs résultats dans les registres suivants :

Utilisation	Premiers	Derniers
Bytes :	al, ah, dl, dh, cl, ch, bl, bh	
Words :	ax, dx, cx, si, di, bx	
Doubles-words :	dx:ax	Sur pre-80386
	eax, edx, ecx, esi, edi, ebx	Sur 80386 et plus récent.
Offsets 16-bits :	bx, si, di, dx	
Offsets 32-bits :	ebx , esi, edi, eax, ecx, edx	
Pointeurs de segment :	es:di, es:bx, dx:ax, es:si	N'utilisez pas DS.

De nouveau, cette table ne représente que des orientations. Si vous le sentez ainsi, vous pourriez renvoyer une valeur de double-mot dans (cl, dh, al, bh). Si vous renvoyez un résultat de fonction dans certains registres, vous ne devriez pas sauver et restaurer ces registres. Faire ainsi irait à l'encontre du but de la fonction.

11.6.2 Retour de résultats de fonction sur la pile

Un autre bon endroit où vous pouvez renvoyer des résultats de fonction est sur la pile. L'idée ici est de pousser quelques valeurs factices sur la pile pour faire de la place pour le résultat de la fonction. La fonction, avant de partir, stocke son résultat dans cet emplacement. Quand la fonction revient à l'appelant, elle dépile tout excepté ce résultat de fonction. Beaucoup de HLLs utilisent cette technique (bien que la plupart des HLLs sur PC retournent les résultats de fonction dans les registres). Les séquences de code suivantes montrent comment des valeurs peuvent être retournées sur la pile:

```
function PasFunc(i,j,k:integer):integer;
begin
       PasFunc := i+j+k;
end;
m := PasFunc(2,n,l);
```

En assembleur :

```
PasFunc_rtn     equ     10[bp]
PasFunc_i       equ     8[bp]
PasFunc_j       equ     6[bp]
PasFunc_k       equ     4[bp]

PasFunc         proc    near
                push    bp
                mov     bp, sp
                push    ax
                mov     ax, PasFunc_i
                add     ax, PasFunc_j
                add     ax, PasFunc_k
                mov     PasFunc_rtn, ax
                pop     ax
                pop     bp
                ret     6
PasFunc         endp
```

Séquence d'appel :

```
                push    ax              ; Place pour résultat retour fonction
                mov     ax, 2
                push    ax
                push    n
```

```
                push    l
                call    PasFunc
                pop     ax              ; Obtient résultat retour fonction
```

Sur des processeur 80286 ou postérieurs, vous pourriez également employer le code :

```
                push    ax              ; Place pour résultat retour fonction
                push    2
                push    n
                push    l
                call    PasFunc
                pop     ax              ; Obtient résultat retour fonction
```

Bien que l'appelant ait poussé huit bytes de données sur la pile, n'en dépile que six. Le premier « paramètre » sur la pile est le résultat de fonction. La fonction doit laisser cette valeur sur la pile quand elle retourne.

11.6.3 Retour de résultats de fonction dans des emplacements en mémoire

Un autre endroit raisonnable pour renvoyer des résultats de fonction est dans un emplacement de mémoire connu. Vous pouvez renvoyer des valeurs de fonction dans des variables globales ou vous pouvez renvoyer un pointeur (en général dans un registre ou une paire de registres) sur un bloc de paramètres. Ce processus est pratiquement identique à passer des paramètres à une procédure ou à une fonction dans des variables globales ou par l'intermédiaire d'un bloc de paramètres.

Le retour des paramètres par l'intermédiaire d'un pointeur sur un bloc de paramètre est une excellente manière de renvoyer de grandes structures de données comme résultats de fonction. Si une fonction retourne un tableau entier, la meilleure manière de renvoyer ce tableau est d'assigner de l'espace, stocker les données dans cet espace, et laisser à la routine d'appel le soin de libérer le stockage. La plupart des langages évolués qui vous permettent de renvoyer de grandes structures de données comme résultats de fonction utilisent cette technique.

Bien sûr, il y a très peu de différence entre renvoyer un résultat de fonction en mémoire et le mécanisme de passage de paramètre par résultat. Voir « Passage par Résultat » à la section 11.5.4 pour plus de détails.

11.7 Effets de bord

Un *effet de bord* ou effet secondaire est n'importe quel calcul ou opération par une procédure qui n'est pas la fonction première de cette procédure. Par exemple, si vous choisissez de ne pas préserver tous les registres affectés dans une procédure, la modification de ces registres est un effet de bord de cette procédure. La programmation par effet de bord, c'est à dire, la pratique d'utiliser les effets de bord des procédures, est très dangereuse. Beaucoup trop souvent, un programmeur se fiera à un effet de bord d'une procédure. Les modifications postérieures peuvent changer l'effet de bord, invalidant tout le code se fondant sur cet effet de bord. Ceci peut rendre vos programmes durs à débuguer et à maintenir. Par conséquent, vous devriez éviter la programmation par effets de bord.

Peut-être quelques exemples de programmation par effet de bord vous aideront à éclaircir les difficultés que vous pouvez rencontrer. La procédure suivant met à zéro un tableau. Pour des raisons d'efficacité, elle laisse l'appelant responsable de la préservation des registres nécessaires. En conséquence, un effet de bord de cette procédure est que les registres bx et cx sont modifiés. En particulier, le registre cx contient zéro au retour.

```
ClrArray        proc    near
                lea     bx, array
                mov     cx, 32
ClrLoop:        mov     word ptr [bx], 0
                inc     bx
                inc     bx
                loop    ClrLoop
                ret
ClrArray        endp
```

Si votre code s'attend à ce que cx contienne zéro après l'exécution de ce sous-programme, vous comptez sur un effet de bord de la procédure ClrArray. La fonction principale de ce code est de mettre à zéro un tableau, non de mettre le registre cx à zéro. Plus tard, si vous modifiez la procédure ClrArray de la manière suivante, votre code qui dépend de ce que cx contienne zéro, ne fonctionnera plus correctement :

```
ClrArray        proc    near
                lea     bx, array
ClrLoop:        mov     word ptr [bx], 0
                inc     bx
                inc     bx
                cmp     bx, offset array+32
                jne     ClrLoop
                ret
ClrArray        endp
```

Aussi, comment pouvez vous éviter les pièges de la programmation par effets de bord dans vos procédures ? En structurant soigneusement votre code et en prêtant une attention particulière à la manière exacte dont votre code appelant et les procédures associées s'interfacent. Ces règles peuvent vous aider à éviter les problèmes de la programmation par effet de bord :

- Documentez toujours correctement les conditions d'entrée et de sortie d'une procédure. Ne jamais se fier à d'autres conditions d'entrée ou de sortie que ces opérations documentées.
- Divisez vos procédures de sorte qu'elles calculent une valeur unique ou exécutent une opération unique. Les sous-programmes qui font deux tâches ou plus produisent, par définition, des effets de bord à moins que chaque invocation de ce sous-programme nécessite tous les calculs et opérations.
- En mettant à jour le code d'une procédure, assurez-vous qu'il obéit toujours aux conditions d'entrée et de sortie. Sinon, soit modifiez le programme de sorte qu'il le fasse, soit mettez à jour la documentation pour que cette procédure reflète les nouvelles conditions d'entrée et de sortie.
- Évitez de passer l'information entre les routines dans le registre flags du CPU. Le passage d'un statut d'erreur dans le drapeau carry est la limite de ce que vous pouvez vous permetttre. Trop d'instructions affectent les drapeaux et il est trop facile de dénaturer une séquence de retour de sorte qu'un drapeau important est modifié lors du retour.
- Sauvez et restaurez toujours tous les registres qu'une procédure modifie.
- Évitez de passer des paramètres et des résultats de fonction dans des variables globales.
- Évitez de passer des paramètres par référence (avec l'intention de les modifier pour une utilisation ultérieure par le code appelant).

Ces règles, comme toutes règles, sont faites pour être enfreintes. De bonnes pratiques de programmation sont souvent sacrifiées sur l'autel de l'efficacité. Il n'y a rien mal à enfreindre ces règles aussi souvent que vous le jugez nécessaire. Cependant, votre code sera difficile à déboguer et à maintenir si vous violez souvent ces règles. Mais tel est le prix de l'efficacité[888]. Jusqu'à ce que vous acquériez assez d'expérience pour faire un choix judicieux au sujet de l'utilisation des effets de bord dans vos programmes, vous devriez les éviter. Le plus souvent, l'utilisation d'un effet de bord posera plus de problèmes qu'elle n'en résoudra.

11.8 Stockage de variables locales

Parfois une procédure aura besoin de mémoire temporaire, dont elle n'a plus besoin quand la procédure retourne. Vous pouvez facilement assigner un tel stockage local variable sur la pile.

Le 80x86 supporte le stockage variable local avec le même mécanisme qu'il utilise pour les paramètres - il utilise les registres bp et sp pour accéder à et assigner de telles variables. Considérez le programme Pascal suivant :

[8] Ce n'est pas seulement une remarque oiseuse. Les programmeurs experts qui doivent extorquer le petit plus de performance dans une section de code ont souvent recours à de mauvaises pratiques de programmation afin de réaliser leurs buts. Ils sont préparés, cependant, à faire face aux problèmes qui se rencontrent souvent dans de telles situations et ils font beaucoup plus attention en ayant affaire à un tel code.

```
program LocalStorage;
var      i,j,k:integer;
         c: array [0..20000] of integer;

         procedure Proc1;
         var     a:array [0..30000] of integer;
                 i:integer;
         begin

         {Code that manipulates a and i}

         end;

         procedure Proc2;
         var     b:array [0..20000] of integer;
                 i:integer;
         begin

         {Code that manipulates b and i}

         end;

begin

{main program that manipulates i,j,k, and c}
end.
```

Pascal assigne normalement des variables globales dans le segment de données et des variables locales dans le segment de pile. Par conséquent, le programme ci-dessus assigne 50.002 mots de stockage local (30.001 mots dans Proc1 et 20.001 mots dans Proc2). C'est au-dessus et au-delà des autres données sur la pile (comme les adresses de retour). Puisque 50.002 mots de stockage consomment 100.004 bytes de stockage, vous avez un petit problème - le CPU 80x86, en mode réel, limite le segment de pile à 65.536 bytes. Pascal évite ce problème en affectant dynamiquement le stockage local à l'entrée d'une procédure et en désaffectant le stockage local au retour. A moins que Proc1 et Proc2 soient toutes deux en activité (ce qui peut seulement se produire si Proc1 appelle Proc2 ou vice versa), il y a suffisamment de stockage pour ce programme. Vous n'avez pas besoin des 30.001 mots de Proc1 et des 20.001 mots de Proc2 en même temps. Aussi Proc1 assigne et utilise 60.002 bytes de stockage, puis désaffecte ce stockage et retourne (libérant les 60.002 bytes). Ensuite, Proc2 assigne 40.002 bytes de stockage, les utilise, les désaffecte, et revient à son appelant. Notez que Proc1 et Proc2 partagent beaucoup des mêmes emplacements de mémoire. Cependant, ils le font à différents moments. Aussi longtemps que ces variables sont des variables temporaires dont vous n'avez pas besoin de sauver les valeurs d'une invocation de la procédure à une autre, cette forme d'attribution de stockage local fonctionne très bien.

La comparaison suivante entre une procédure Pascal et son code correspondant en assembleur, vous donnera une bonne idée de la façon d'assigner le stockage local sur la pile:

```
procedure LocalStuff(i,j,k:integer);
var l,m,n:integer; {local variables}
begin

       l:= i+2;
       j:= l*k+j;
       n:= j-l;
       m:= l+j+n;

end;
```

Séquence d'appel :

```
       LocalStuff(1,2,3);
```

Code assembleur:

```
LStuff_i        equ     8[bp]
LStuff_j        equ     6[bp]
LStuff_k        equ     4[bp]
LStuff_l        equ     -4[bp]
```

```
LStuff_m        equ     -6[bp]
LStuff_n        equ     -8[bp]

LocalStuff      proc    near
                push    bp
                mov     bp, sp
                push    ax
                sub     sp, 6           ; Alloue variables locales.
L0:             mov     ax, LStuff_i
                add     ax, 2
                mov     LStuff_l, ax
                mov     ax, LStuff_l
                mul     LStuff_k
                add     ax, LStuff_j
                mov     LStuff_j, ax
                sub     ax, LStuff_l    ; AX contient déjà j
                mov     LStuff_n, ax
                add     ax, LStuff_l    ; AX contient déjà n
                add     ax, LStuff_j
                mov     LStuff_m, ax

                add     sp, 6           ; Désalloue stockage local
                pop     ax
                pop     bp
                ret     6
LocalStuff      endp
```

L'instruction sub sp, 6 fait de la place pour trois mots sur la pile. Vous pouvez assigner l, m, et n dans ces trois mots. Vous pouvez référencer ces variables en indexant à partir du registre bp en utilisant des offsets négatifs (voir le code ci-dessus). En atteignant l'instruction à l'étiquette L0, la pile ressemble quelque peu à la Figure 11.16.

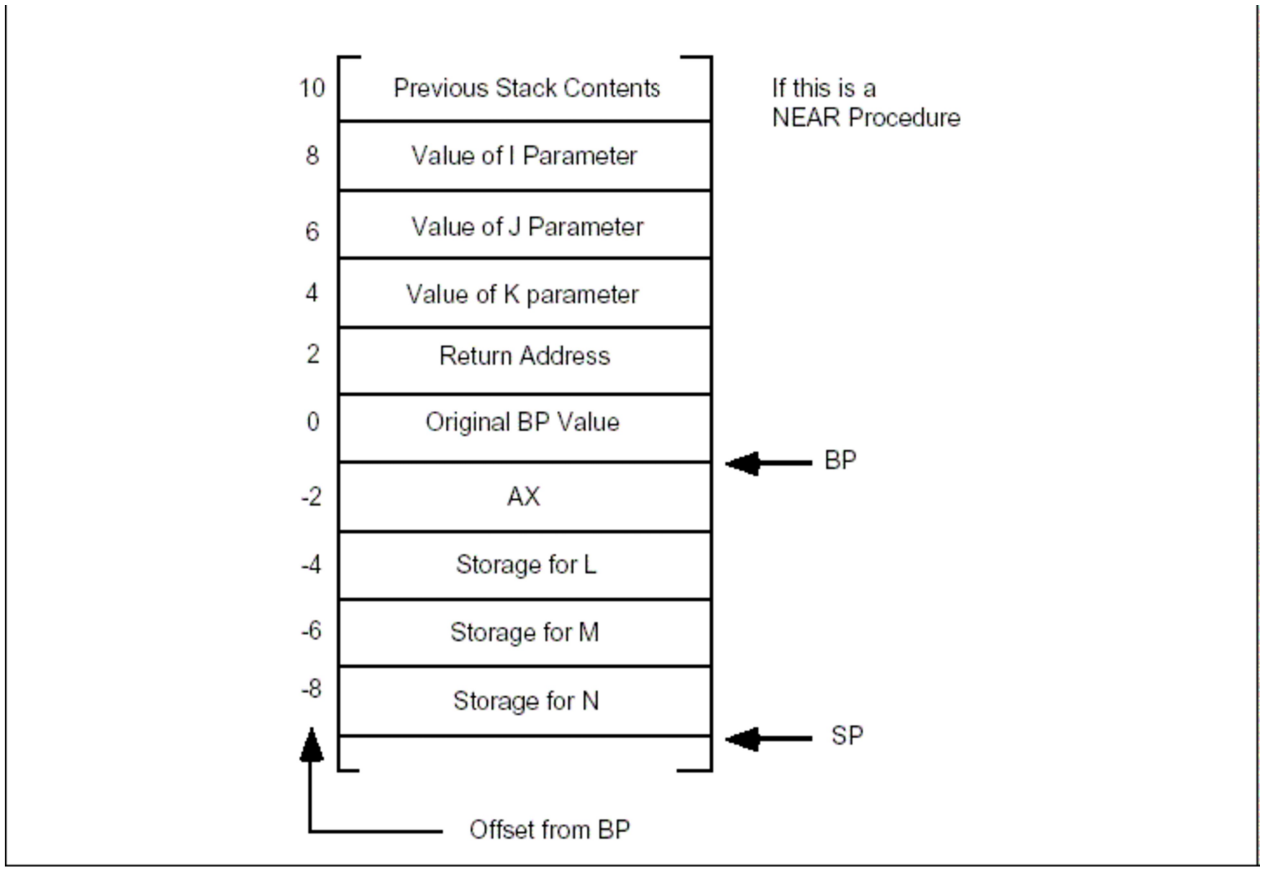


Figure 11.06 La pile à l'entrée de la procédure LocalStuff

Ce code utilise l'instruction correspondante add sp, 6 à la fin de la procédure pour désaffecter le stockage local. La valeur que vous ajoutez au pointeur de pile doit exactement correspondre à la valeur que vous soustrayez en affectant ce stockage. Si ces deux valeurs ne correspondent pas, le pointeur de pile à l'entrée de la routine ne correspondra pas au pointeur de pile à la sortie; c'est comme pousser ou extraire trop d'éléments à l'intérieur de la procédure.

À la différence des paramètres, qui ont un offset fixe dans le bloc d'activation, vous pouvez assigner des variables locales dans n'importe quel ordre. Dans la mesure où vous êtes cohérent dans vos allocations d'emplacement, vous pouvez les assigner de la manière que vous voulez. Gardez à l'esprit, cependant, que le 80x86 supporte deux formes du mode d'adressage disp[bp]. Il utilise un déplacement d'un byte quand il est dans la plage -128..+127. Il utilise un déplacement de deux bytes pour des valeurs dans la plage -32.768..+32.767. Par conséquent, vous devriez placer tous les types de données primitifs et d'autres petites structures près du pointeur de base, de manière à utiliser des déplacements d'un byte unique. Vous devriez placer de grands tableaux et d'autres structures de données au-dessous des variables plus petites sur la pile.

La plupart du temps vous n'avez pas à vous faire du souci pour assigner des variables locales sur la pile. La plupart des programmes n'exigent pas plus de 64K de stockage. Le CPU traite les variables globales plus rapidement que les variables locales. Il y a deux situations où assigner des variables locales en tant que globales dans le segment de données n'est pas pratique: lorsque vous interfacez l'assembleur avec des langages de haut niveau comme le Pascal, et en écrivant du code récursif. Quand vous interfacez avec le Pascal, votre code assembleur peut ne pas avoir de segment de données qu'il peut utiliser, la récursion exige souvent des instances multiples de la même variable.

11.9 La récursion

La récursion se produit quand une procédure s'appelle elle-même. Ce qui suit, par exemple, est une procédure récursive:

```
Recursive       proc
                call    Recursive
                ret
Recursive       endp
```

Naturellement, Le CPU n'exécutera jamais l'instruction ret à la fin de cette procédure. À l'entrée dans Recursive, cette procédure se fera immédiatement appel de nouveau et le contrôle ne passera jamais à l'instruction ret. En ce cas particulier, la récursion en ligne de fuite a comme conséquence une boucle infinie.

A bien des égards, la récursion est très semblable à l'itération (c'est-à-dire, l'exécution réitérée d'une boucle). Le code suivant produit également une boucle infinie:

```
Recursive       proc
                jmp     Recursive
                ret
Recursive       endp
```

Il y a, cependant, une différence principale entre ces deux implémentations. L'ancienne version de Recursive pousse une adresse de retour sur la pile à chaque invocation du sous-programme. Ceci ne se produit pas dans l'exemple immédiatement ci-dessus (puisque l'instruction de jmp n'affecte pas la pile).

Comme une structure de boucle, la récursion exige une condition d'arrêt afin d'arrêter la récursion infinie. Recursive pourrait être récrite avec une condition d'arrêt comme suit :

```
Recursive       proc
                dec     ax
                jz      QuitRecurse
                call    Recursive
QuitRecurse:    ret
Recursive       endp
```

Cette modification de la routine amène Recursive à s'appeller elle-même le nombre de fois qui apparait dans le registre ax. A chaque appel, Recursive décrémente le registre ax de un et s'appelle de nouveau. Finalement, ax Recursive décrémente ax à zéro et retourne. Une fois que ceci se produit, le CPU exécute une chaîne d'instructions ret jusqu'à ce que le contrôle revienne à l'appel original à Recursive.

Jusqu'ici, cependant, on n'a pas vraiment eu besoin de récursion. Après tout, vous pourriez efficacement coder cette procédure comme suit:

```
Recursive                               proc
RepeatAgain:    dec     ax
                jnz     RepeatAgain
                ret
Recursive       endp
```

Les deux exemples répétent le corps de la procédure le nombre de fois passé dans le registre ax[99]. Il s'avère qu'il il y a très peu d'algorithmes récursifs que vous ne puissiez pas implémenter d'une mode itérative. Cependant, beaucoup d'algorithmes implémentés récursivement sont plus efficaces que leurs correspondants itératifs et la plupart du temps la forme récursive de l'algorithme est beaucoup plus facile à comprendre.

L'algorithme de tri quicksort est probablement l'algorithme le plus célèbre qui apparaît presque toujours en forme récursive. Une implémentation Pascal de cet algorithme suit:

```
procedure quicksort(var a:ArrayToSort; Low,High: integer);
       procedure sort(l,r: integer);
       var i, j, Middle, Temp: integer;
       begin
              i:=l;
              j:=r;
              Middle:=a[(l+r) DIV 2];
              repeat
                     while (a[i] < Middle) do i:=i+1;
                     while (Middle < a[j]) do j:=j-1;
                     if (i <= j) then begin

                            Temp:=a[i];
                            a[i]:=a[j];
                            a[j]:=Temp;
                            i:=i+1;
                            j:=j-1;

                     end;

              until i>j;
              if l<j then sort(l,j);
              if i<r then sort(i,r);

       end;

begin {quicksort};
       sort(Low,High);
end;
```

La sous-routine sort est la routine récursive dans cet extrait. La récursion se produit aux deux dernières instructions if de la procédure sort.

En assembleur, la routine sort ressemble en gros à ceci:

```
                        include         stdlib.a
                        includelib      stdlib.lib
cseg                    segment
                        assume  cs:cseg, ds:cseg, ss:sseg, es:cseg
```

[9] Bien que la dernière version le fasse considérablement plus vite puisqu'elle n'a pas à gérer les instructions CALL/RET

```
; Programme pour tester la routine de tri

Main                    proc
                        mov     ax, cs
                        mov     ds, ax
                        mov     es, ax

                        mov     ax, 0
                        push    ax
                        mov     ax, 31
                        push    ax
                        call    sort

                        ExitPgm                 ; Retourne au DOS
Main                    endp

; Données à trier

a                       word    31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16
                        word    15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0

; procédure sort (l,r:integer)
; Trie le tableau A entre les index l et r

l                       equ     6[bp]
r                       equ     4[bp]
i                       equ     -2[bp]
j                       equ     -4[bp]

sort                    proc    near
                        push    bp
                        mov     bp, sp
                        sub     sp, 4           ; Fait de la place pour i et j.

                        mov     ax, l           ; i:= l
                        mov     i, ax
                        mov     bx, r           ; j:= r

                        mov     j, bx

; Note: Ce calcul de l'adresse de a[(l+r) div 2] est un peu
; étrange. Au lieu de diviser par deux, puis multiplier par deux
; (puisque A est un tableau de mots), ce code met simplement à zéro
; le bit de L.O. de BX.

                        add     bx, l           ; Middle:= a[(l+r) div 2]
                        and     bx, 0FFFEh
                        mov     ax, a[bx]       ; BX*2, parce qu'il s'agit
                                                ; d'un tableau de mots,annulle
                                                ; “div 2” ci-dessus.
;
; Répéte jusqu'à ce que i > j: Bien sûr, I et J sont dans BX et SI.

                        lea     bx, a           ; Calcule l'adresse de a[i]
                        add     bx, i           ; et la laisse dans BX.
                        add     bx, i
                        lea     si, a           ; Calcule l'adresse de a[j]
                        add     si, j           ; et la laisse dans SI.
                        add     si, j

RptLp:

; Tant que (a [i] < Middle) fait i:= i + 1;

                        sub     bx, 2           ; Nous l'incrémenterons bientôt.
WhlLp1:                 add     bx, 2
                        cmp     ax, [bx]        ; AX contient toujours middle
                        jg      WhlLp1

; Tant que (Middle < a[j]) fait j:= j-1.
```

```
                        add     si, 2           ; Nous le décrémenterons en boucle
WhlLp2:                 add     si, 2
                        cmp     ax, [si]        ; AX contient toujours la valeur
                        jl      WhlLp2          ; de middle.
                        cmp     bx, si
                        jnle    SkipIf

; Echange, si nécessaire

                        mov     dx, [bx]
                        xchg    dx, [si]
                        xchg    dx, [bx]
                        add     bx, 2           ; Augmente de 2 (valeurs entières)
                        sub     si, 2

SkipIf:                 cmp     bx, si
                        jng     RptLp

; Convertit SI et BX de nouveau en I et J

                        lea     ax, a
                        sub     bx, ax
                        shr     bx, 1
                        sub     si, ax
                        shr     si, 1

; Maintenant la partie récursive:

                        mov     ax, l
                        cmp     ax, si
                        jnl     NoRec1
                        push    ax
                        push    si
                        call    sort

NoRec1:                 cmp     bx, r
                        jnl     NoRec2
                        push    bx
                        push    r
                        call    sort
NoRec2:                 mov     sp, bp
                        pop     bp
                        ret     4

Sort                    endp

cseg                    ends
sseg                    segment stack 'stack'
                        word    256 dup (?)
sseg                    ends
                        end     main
```

A part quelques optimisations de base (comme garder plusieurs variables dans les registres), ce code est presque une traduction littérale du code Pascal. Notez que les variables locales i et j ne sont pas nécessaire dans ce code assembleur (nous pourrions utiliser des registres pour contenir leurs valeurs). Leur utilisation démontre simplement l'allocation des variables locales sur la pile.

Il y a une chose qu'on devrait garder à l'esprit en utilisant la récursion - les routines récursives peuvent dévorer un espace considérable de pile. Par conséquent, en écrivant des sous-programmes récursifs, assignez toujours suffisamment de mémoire dans votre segment de pile. L'exemple ci-dessus a un espace de pile extrêmement anémique de 512 bytes, cependant, il ne trie que 32 nombres, donc une pile de 512 bytes est suffisante. En général, vous ne connaîtrez pas la profondeur à laquelle la récursion vous amènera, ainsi le fait d'assigner un grand bloc de mémoire pour la pile peut être approprié.

Il y a plusieurs considérations d'efficacité qui s'appliquent aux procédures récursives. Par exemple, le deuxième appel (récursif) à sort dans le code assembleur ci-dessus n'a pas besoin d'être un appel récursif. En définissant quelques

variables et registres, une instruction jmp unique peut remplacer les pushes et l'appel récursif. Ceci améliorera l'exécution de la routine quicksort (de beaucoup, en fait) et réduira la quantité de mémoire que la pile exige. Un bon livre sur des algorithmes, tels que *The Art of Computer Programming* de D.E. Knuth, Volume 3, serait une excellente source de matériel suplémentaire sur quicksort. D'autres sources sur les algorithmes complexes, la théorie et les algorithmes de la récursion seraient un bon endroit pour rechercher des idées sur comment implémenter efficacement des algorithmes récursifs.

11.10 Exemple de programme

Le programme d'exemple suivant démontre plusieurs concepts apparaissant dans ce chapitre, plus particulièrement, le passage des paramètres sur la pile. Ce programme (Pgm11_1.asm apparaissant sur le CD-ROM d'accompagnement) manipule l'écran d'affichage vidéo texte du PC mappé en mémoire (à l'adresse B800:0 pour des affichages couleur, B000:0 pour un affichage monochrome). Il fournit les routines qui "capturent" toutes les données sur l'écran dans un tableau, écrivent le contenu d'un tableau à l'écran, effacent l'écran, font défilert une ligne vers le haut ou vers le bas, placent le curseur à une coordonnée (X,Y) et récupèrent la position actuelle de curseur.

Notez que ce code a été écrit pour démontrer l'utilisation des paramètres et des variables locales. Par conséquent, il est plutôt inefficace. Comme les commentaires le précisent, plusieurs des fonctions que cet utilitaire fournit pourraient être écrites pour fonctionner beaucoup plus rapidement en utilisant des instructions de chaîne du 80x86. Voyez les exercices de laboratoire pour une version différente de certaines de ces fonctions écrites d'une telle manière. En outre notez que ce code fait quelques appels au BIOS du PC pour définir et obtenir la position du curseur de même qu'effacer l'écran. Voyez le chapitre sur le BIOS et le DOS pour plus de détails sur ces appels au BIOS.

```
; Pgm11_1.asm
;
; Utilitaires d'affichage
;
; Ce programme vous fournit quelques routines utiles de manipulation
; d'écran qui vous permettent des actions telles que positionner
; le curseur, sauver et restaurer le contenu de l'écran, effacer l'écran,
; etc.
;
; Ce programme n'est pas très efficace. Il a été écrit pour démontrer
; le passage de paramètres, l'utilisation de variables locales, et
; la conversion directe des boucles en assembleur. Il y a de bien
; meilleures méthodes pour réaliser ces tâches (qui tournent
; environ 5-10x plus vite) en utilisant les instructions de chaine
; du 80x86.

            .xlist
            include stdlib.a
            includelib stdlib.lib
            .list

            .386                    ; Mettre ces instructions en
            option    segment:use16 ; commentaires si vous
                                    ; n'utilisez pas un 80386.

; ScrSeg- Ceci est le segment d'adresse de l'écran vidéo. Ce devrait
; être B000 pour les écrans mono et B800 pour les écrans couleur.

ScrSeg  =       0B800h

dseg            segment         para public 'data'

XPosn           word    ?               ; Coordonnées Curseur X (0..79)
YPosn           word    ?               ; Coordonnées Curseur Y (0..24)

; Le tableau suivant contient une copie des données de l'écran initial.

SaveScr         word    25 dup (80 dup (?))

dseg            ends

cseg            segment para public 'code'
```

```
                assume  cs:cseg, ds:dseg.

; Capture-      Copie les données de l'écran dans le tableau passé
;               par référence comme paramètre.
;
; procedure Capture(var ScrCopy:array[0..24,0..79] of word);
; var x,y:integer;
; begin
;
;       for y:= 0 to 24 do
;               for x:= 0 to 79 do
;                       SCREEN[y,x]:= ScrCopy[y,x];
; end;
;
;
; Bloc d'Activation pour Capture:
;
;       |                                       |
;       | Contenu précédent de pile             |
;       -----------------------------
;       |       Adrs Seg ScrCopy                |
;       --                                      --
;       |       Adrs offset ScrCopy             |
;       -----------------------------
;       |       Adrs Retour (near)              |
;       -----------------------------
;       |               Ancien BP               |
;       -----------------------------           <- BP
;       |       valeur coordonnée X             |
;       -----------------------------
;       |       valeur coordonnée Y             |
;       -----------------------------
;       |       Registres, etc.                 |
;       -----------------------------           <- SP

ScrCopy_cap     textequ         <dword ptr [bp+4]>
X_cap           textequ         <word ptr [bp-2]>
Y_cap           textequ         <word ptr [bp-4]>

Capture         proc
                push    bp
                mov     bp, sp
                sub     sp, 4           ; Alloue de l'espace pour les locaux.

                push    es
                push    ds
                push    ax
                push    bx
                push    di

                mov     bx, ScrSeg      ; Définit le pointeur sur mémoire
                mov     es, bx          ; SCREEN (ScrSeg:0).

                lds     di, ScrCopy_cap ; Obtient ptr sur tableau de capture.

                mov     Y_cap, 0
YLoop:          mov     X_cap, 0
XLoop:          mov     bx, Y_cap
                imul    bx, 80          ; Mémoire écran est un tableau 25x80
                add     bx, X_cap       ; ordonné en ordre par rangée
                add     bx, bx          ; avec deux bytes par élément.

                mov     ax, es:[bx]     ; Lit code caractère à l'écran.
                mov     [di][bx], ax    ; Le stocke dans tableau capture.

                inc     X_Cap           ; Répète pour chaque caractère sur cette
                cmp     X_Cap, 80       ; ligne de caractères (chaque caractère
```

```
                jb      Xloop           ; dans la ligne fait deux bytes).
                inc     Y_Cap           ; Répète pour chaque ligne à l'écran
                cmp     Y_Cap, 25
                jb      YLoop
                pop     di
                pop     bx
                pop     ax
                pop     ds
                pop     es
                mov     sp, bp
                pop     bp
                ret     4
Capture         endp


; Fill-         Copie le tableau passé par référence sur l'écran.
;
; procedure Fill(var ScrCopy:array[0..24,0..79] of word);
; var x,y:integer;
; begin
;
;       for y:= 0 to 24 do
;               for x:= 0 to 79 do
;                       ScrCopy[y,x]:= SCREEN[y,x];
; end;
;
;
; Bloc d'Activation pour Fill:
;
;       |                                       |
;       | Contenu précédent de pile             |
;       -----------------------------
;       |       Adrs Seg ScrCopy                |
;       --                                      --
;       |       Adrs offset ScrCopy             |
;       -----------------------------
;       |       Adrs Retour (near)              |
;       -----------------------------
;       |               Ancien BP               |
;       -----------------------------           <- BP
;       |       valeur coordonnée X             |
;       -----------------------------
;       |       valeur coordonnée Y             |
;       -----------------------------
;       |       Registres, etc.                 |
;       -----------------------------           <- SP


ScrCopy_fill    textequ         <dword ptr [bp+4]>
X_fill          textequ         <word ptr [bp-2]>
Y_fill          textequ         <word ptr [bp-4]>

Fill            proc
                push    bp
                mov     bp, sp
                sub     sp, 4
                push    es
                push    ds
                push    ax
                push    bx
                push    di
```

```
                mov     bx, ScrSeg      ; Définit le pointeur sur mémoire
                mov     es, bx          ; SCREEN (ScrSeg:0).

                lds     di, ScrCopy_fill ; Obtient ptr sur tableau de données.

                mov     Y_Fill, 0
YLoop:          mov     X_Fill, 0
XLoop:          mov     bx, Y_Fill
                imul    bx, 80          ; Mémoire écran est un tableau 25x80
                add     bx, X_Fill      ; ordonné en ordre par rangée
                add     bx, bx          ; avec deux bytes par élément.

                mov     ax, [di][bx]    ; Stocke dans tableau capture.
                mov     es:[bx], ax     ; Lit code caractère à l'écran.

                inc     X_Fill          ; Répète pour chaque caractère sur cette
                cmp     X_Fill, 80      ; ligne de caractères (chaque caractère
                jb      Xloop           ; dans la ligne fait deux bytes).

                inc     Y_Fill          ; Répète pour chaque ligne de l'écran.
                cmp     Y_Fill, 25
                jb      YLoop

                pop     di
                pop     bx
                pop     ax
                pop     ds
                pop     es
                mov     sp, bp
                pop     bp
                ret     4
Fill            endp


; Scroll_up-    Fait défiler l'écran d'une ligne vers le haut. On réalise
;                       cela en copiant la deuxième ligne sur la première,
;                       la troisième ligne sur la seconde, la quatrième ligne
;                       sur la troisième, etc.
;
; procedure Scroll_up;
; var x,y:integer;
; begin
;       for y:= 1 to 24 do
;               for x:= 0 to 79 do
;                       SCREEN[Y-1,X]:= SCREEN[Y,X];
; end;
;

; Bloc d'Activation pour Scroll_up:
;
;       |                                       |
;       | Contenu précédent de pile             |
;       -----------------------------
;       |       Adrs Retour (near)              |
;       -----------------------------
;       |               Ancien BP               |
;       -----------------------------           <- BP
;       |       valeur coordonnée X             |
;       -----------------------------
;       |       valeur coordonnée Y             |
;       -----------------------------
;       |       Registres, etc.                 |
;       -----------------------------           <- SP

X_su            textequ         <word ptr [bp-2]>
Y_su            textequ         <word ptr [bp-4]>
```

```
Scroll_up       proc
                push    bp
                mov     bp, sp
                sub     sp, 4           ; Fait de la place pour X, Y.

                push    ds
                push    ax
                push    bx

                mov     ax, ScrSeg
                mov     ds, ax
                mov     Y_su, 0
su_Loop1:       mov     X_su, 0

su_Loop2:       mov     bx, Y_su        ; Calcule index dans le tableau
                imul    bx, 80          ; de l'écran.
                add     bx, X_su
                add     bx, bx          ; Attention: Tableau de mots.

                mov     ax, [bx+160]    ; Prend un mot de la ligne source.
                mov     [bx], ax        ; Stocke dans ligne dest.
                inc     X_su
                cmp     X_su, 80
                jb      su_Loop2

                inc     Y_su
                cmp     Y_su, 80
                jb      su_Loop1

                pop     bx
                pop     ax
                pop     ds
                mov     sp, bp
                pop     bp
                ret
Scroll_up       endp


; Scroll_dn-    Fait défiler l'écran d'une ligne vers le bas. On réalise
;                       cela en copiant la 24ème ligne sur la 25ème, la 23ème
;                       ligne sur la 24ème, la 22ème ligne sur la 23ème, etc.
;
; procedure Scroll_dn;
; var x,y:integer;
;       begin
;               for y:= 23 downto 0 do
;                       for x:= 0 to 79 do
;                               SCREEN[Y+1,X]:= SCREEN[Y,X];
; end;
;
; Bloc d'Activation pour Scroll_dn:
;
;       |                                       |
;       | Contenu précédent de pile             |
;       -----------------------------
;       |       Adrs Retour (near)              |
;       -----------------------------
;       |               Ancien BP               |
;       -----------------------------           <- BP
;       |       valeur coordonnée X             |
;       -----------------------------
;       |       valeur coordonnée Y             |
;       -----------------------------
;       |       Registres, etc.                 |
;       -----------------------------           <- SP
```

```
X_sd            textequ         <word ptr [bp-2]>
Y_sd            textequ         <word ptr [bp-4]>

Scroll_dn       proc
                push    bp
                mov     bp, sp
                sub     sp, 4           ; Fait de la place pour X, Y.

                push    ds
                push    ax
                push    bx

                mov     ax, ScrSeg
                mov     ds, ax
                mov     Y_sd, 23
sd_Loop1:       mov     X_sd, 0

sd_Loop2:       mov     bx, Y_sd        ; Calcule index dans le tableau
                imul    bx, 80          ; de l'écran.
                add     bx, X_sd
                add     bx, bx          ; Attention: Tableau de mots.

                mov     ax, [bx         ; Prend un mot de la ligne source.
                mov     [bx+160], ax    ; Stocke dans ligne dest.

                inc     X_sd
                cmp     X_sd, 80
                jb      sd_Loop2

                dec     Y_sd
                cmp     Y_sd, 0
                jge     sd_Loop1

                pop     bx
                pop     ax
                pop     ds
                mov     sp, bp
                pop     bp
                        ret
Scroll_dn       endp


; GotoXY-       Positionne le curseur aux coordonnées X, Y spécifiées.
;
; procedure gotoxy(x,y:integer);
; begin
;       BIOS(posnCursor,x,y);
; end;
;
; Bloc d'Activation pour GotoXY
;
;       |                                       |
;       | Contenu précédent de pile             |
;       -----------------------------
;       |       valeur coordonnée X             |
;       -----------------------------
;       |       valeur coordonnée Y             |
;       -----------------------------
;       |       Adrs Retour (near)              |
;       -----------------------------
;       |               Ancien BP               |
;       -----------------------------         <- BP
;       |       Registres, etc.                 |
;       -----------------------------         <- SP

X_gxy           textequ         <byte ptr [bp+6]>
Y_gxy           textequ         <byte ptr [bp+4]>
```

```
GotoXY          proc
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                push    dx

                mov     ah, 2           ; Valeur magique BIOS pour gotoxy.
                mov     bh, 0           ; Affiche page zero.
                mov     dh, Y_gxy       ; Définit paramètres BIOS (X,Y).
                mov     dl, X_gxy
                int     10h             ; Appelle le BIOS.

                pop     dx
                pop     bx
                pop     ax
                mov     sp, bp
                pop     bp
                ret     4
GotoXY          endp

; GetX-         Retourne les coordonnées X du curseur dans le registre AX.

GetX            proc
                push bx
                push cx
                push dx

                mov ah, 3               ; Lit coordonnées X, Y depuis le
                mov bh, 0               ; BIOS
                int     10h

                mov     al, dl          ; Retourne coord. X dans AX.
                mov     ah, 0

                pop     dx
                pop     cx
                pop     bx
                ret
GetX            endp

; GetY-         Retourne coordonnée Y du curseur dans le registre AX.

GetY            proc
                push    bx
                push    cx
                push    dx

                mov     ah, 3
                mov     bh, 0
                int     10h

                mov     al, dh          ;Retourne Coord. Y dans AX.
                mov     ah, 0

                pop     dx
                pop     cx
                pop     bx
                ret
GetY            endp

; ClearScrn-    Efface l'écran et positionne le curseur à (0,0).
;
; procedure ClearScrn;
; begin
;       BIOS(Initialize)
; end;
```

```
ClearScrn       proc
                push    ax
                push    bx
                push    cx
                push    dx

                mov     ah, 6           ; Numéro Magique du BIOS.
                mov     al, 0           ; Efface tout l'écran.
                mov     bh, 07          ; Efface avec des espaces noirs.
                mov     cx, 0000        ; Le coin haut/gauche est (0,0)
                mov     dl, 79          ; Coordonnée X la plus basse
                mov     dh, 24          ; Coordonnée Y la plus basse
                int     10h             ; Appelle le BIOS.

                push    0               ; Positionne le curseur à (0,0)
                push    0               ; après l'appel.
                call    GotoXY

                pop     dx
                pop     cx
                pop     bx
                pop     ax
                ret
ClearScrn       endp

; un court programme principal pour tester nos routines:

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

; Sauve l'écran tel qu'il est quand ce programme est lancé.

                push    seg SaveScr
                push    offset SaveScr
                call    Capture

                call    GetX
                mov     XPosn, ax

                call    GetY
                mov     YPosn, ax

; Efface l'écran pour préparer nos trucs.

                call    ClearScrn

; Positionne le curseur au milieu de l'écran et affiche quelque chose.

                push    30              ; valeur X
                push    10              ; valeur Y
                call    GotoXY

                print
                byte    "Screen Manipulation Demo",0

                push    30
                push    11
                call    GotoXY

                print
                byte    "Press any key to continue",0

                getc

;Fait défiler l'écran deux lignes vers le haut

                call    Scroll_up
                call    Scroll_up
```

```
                getc

;Fait défiler l'écran quatre lignes vers le bas:

                call    Scroll_dn
                call    Scroll_dn
                call    Scroll_dn
                call    Scroll_dn
                getc

; Restaure l'écran à son état premier avant cet appel.

                push    seg SaveScr
                push    offset SaveScr
                call    Fill

                push    XPosn
                push    YPosn
                call    GotoXY

Quit:           ExitPgm                 ; macro DOS pour quitter programme.
Main            endp
cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

11.11 Exercices de laboratoire

Cet exercice de laboratoire démontre comment un programme C/C++ appelle quelques fonctions en assembleur. Il se compose de deux unités de programme: un programme Borland C++ (Ex11_1.cpp) et un programme MASM 6.11 (Ex11_1a.asm). Puisque vous pouvez ne pas avoir accès à un compilateur C++ (et Borland C++ en particulier)[110], le fichier EX11.EXE contient une version precompilée et liée de ces fichiers. Si vous avez une copie de Borland C++, puis vous pouvez compiler/assembler ces fichiers en utilisant le fichier makefile qui apparaît également dans le sous-répertoire du chapitre 11.

Le listing du programme C++ apparaît dans la section 11.11.1. Ce programme efface l'écran et puis fait rebondir un signe dièse ("#") à travers l'écran jusqu'à ce que l'utilisateur appuie sur n'importe quelle touche. Alors il reconstitue l'écran à l'affichage précédant l'exécution et stoppe. Toutes les manipulations d'écran, aussi bien que les tests des touches, sont gérées par des fonctions écrites en assembleur. Les instructions "extern" au début du programme fournissent la liaison à ces fonctions[111] en assembleur. Il y a quelques choses importantes à noter au sujet de la façon dont C/C++ passe des paramètres à une fonction en assembleur:

- C++ pousse des paramètres sur la pile dans l'ordre *inverse* qu'ils apparaissent dans une liste de paramètres. Par exemple, pour l'appel "f(a, b);" C++ pousserait b d'abord et ensuite a. C'est le contraire de la plupart des exemples de ce chapitre.
- En C++, l'appelant est responsable de l'enlèvement des paramètres de la pile. Dans ce chapitre, l'appelé (la fonction elle-même) en général enlevait les paramètres en indiquant une certaine valeur après l'instruction ret. Les fonctions en assembleur que C++ appelle ne doivent *pas* le faire.

[110] Il n'y a rien de spécifique à Borland dans ce programme C++. Borland a été choisi parce qu'il fournit une option qui génère du code assembleur bien annoté.

[111] L'expression *"C" extern* demande à Borland C++ de générer des noms externes C standard plutôt que des noms modifiés (*mangled*) par C++. Un nom externe C est le nom de la fonction avec un soulignage devant lui (par exemple, GotoXY devient _ GotoXY). C++ change complètement le nom pour manipuler le chargement de la fonction et il est difficile de prévoir le nom réel de la fonction correspondante en assembleur.

- C++ sur le PC utilise différents modèles de mémoire pour contrôler si les pointeurs et les fonctions sont near ou far. Ce programme particulier utilise le modèle mémoire *compact*. Celui-ci gère des procédures near et des pointeurs far. Par conséquent, tous les appels seront near (avec une adresse de retour à deux bits seulement sur la pile) et tous les pointeurs sur les objets de données seront far.
- Le Borland C++ exige d'une fonction qu'elle préserve les registres de segment, BP, DI, et SI. La fonction n'a pas besoin de préserver les autres registres. Si une fonction en assembleur doit renvoyer un résultat de fonction de 16 bits à C++, elle doit renvoyer cette valeur dans le registre de AX.
- Voir le Guide du Programmeur C++ de Borland (ou le manuel correspondant pour votre compilateur C++) pour plus de détails au sujet de l'interface entre C/C++ et assembleur.

La plupart des compilateurs C++ vous donnent l'option de produire une sortie en langage assembleur plutôt qu'en code binaire machine. Borland C++ est bien parce qu'il produit du code assembleur bien annoté avec des commentaires indiquant quelles instructions C++ correspondent à une séquence donnée d'instructions en assembleur. Le code assembleur produit par BCC apparaît dans la section 11.11.2 (c'est une version légèrement éditée pour enlever l'information superflue). Examinez ce code et notez que, soumis aux règles ci-dessus, le compilateur C++ émet du code qui est très semblable à celui décrit dans tout ce chapitre.

Le fichier Ex11_1a.asm (voir la section 11.11.3) est le code réel en assembleur que le programme C++ appelle. Celui-ci contient les fonctions pour les routines GotoXY, GetXY, ClrScrn, tstKbd, Capture, PutScrn, PutChar et PutStr qu'ex11_1.cpp appelle. Pour éviter des problèmes légaux de distribution de logiciel, ce programme particulier C/C++ n'inclut aucun appel aux fonctions de la bibliothèque standard C/C++. En outre, il n'utilise pas le fichier standard C0m.obj de Borland qui appelle le programme principal. L'accord de licence libéral de Borland ne permet *pas* de distribuer leurs bibliothèques et modules objets détachés de leur code. Le code en assembleur fournit les routines d'E/S nécessaires et il fournit également une routine de démarrage (StartPgm) qui appelle le programme principal C++ quand DOS/Windows transfère le contrôle au programme. En fournissant les routines de cette façon, vous ne avez pas besoin du code de bibliothèque ou d'objet de Borland pour lier ces programmes ensemble.

Un effet secondaire de lier les modules ainsi est que le compilateur, l'assembleur, et l'éditeur de liens ne peuvent pas stocker l'information correcte de débugage de niveau de source dans le fichier .exe. Par conséquent, vous ne pourrez pas utiliser CodeView pour regarder le code source réel. Au lieu de cela, vous devrez travailler avec du code machine désassemblé. C'est où le code en assembleur de Borland C++ (Ex11_1.asm) devient pratique. Alors que vous tracez pas à pas le programme principal C++ vous pouvez tracer le flux du programme en regardant le fichier d'Ex11_1.asm.

Pour votre rapport de laboratoire: tracez pas à pas le code de StartPgm jusqu'à ce qu'il appelle la fonction de principale C++. Quand ceci se produit, localisez les appels aux routines dans Ex11_1a.asm. Placez des points d'arrêt sur chacun de ces appels en utilisant la touche F9. Exécutez jusqu'à chaque point d'arrêt, puis tracez pas à pas dans la fonction en utilisant la touche F8. Une fois à l'intérieur, affichez les emplacements mémoire commençant à SS:SP. Identifiez chaque paramètre passé sur la pile. Pour des paramètres de référence, vous devriez regarder les emplacements mémoire dont l'adresse apparaît sur la pile. Reportez vos résultats dans votre rapport de laboratoire.

Incluez un listing imprimé du fichier d'Ex11_1.asm et identifiez les instructions qui poussent chaque paramètre sur la pile. Lors de l'exécution, déterminez les valeurs de paramètre que chaque séquence de push pousse sur la pile et incluez ces valeurs dans votre rapport de laboratoire.

Beaucoup des fonctions du fichier en assembleur prennent un temps considérable pour s'exécuter. Par conséquent, vous ne devriez pas tracer pas à pas chacune de ces fonctions. Au lieu de cela, assurez-vous que vous avez installé des des points d'arrêt sur chacune des instructions d'appel dans le programme C++ et utilisez la touche F5 pour exécuter (à pleine vitesse) jusqu'à l'appel de fonction suivant.

11.11.1 Ex11_1.cpp

```
extern "C" void GotoXY(unsigned y, unsigned x);
extern "C" void GetXY(unsigned &x, unsigned &y);
extern "C" void ClrScrn();
extern "C" int tstKbd();
```

```
extern "C" void Capture(unsigned ScrCopy[25][80]);
extern "C" void PutScr(unsigned ScrCopy[25][80]);
extern "C" void PutChar(char ch);
extern "C" void PutStr(char *ch);

int main()
{
	unsigned SaveScr[25][80];

	int	dx,
		x,
		dy,
		y;

	long	i;

	unsigned savex,
		savey;

	GetXY(savex, savey);
	Capture(SaveScr);
	ClrScrn();

	GotoXY(24,0);
	PutStr("Press any key to quit");

	dx = 1;
	dy = 1;
	x = 1;
	y = 1;
	while (!tstKbd())
	{

		GotoXY(y, x);
		PutChar('#');

		for (i=0; i<500000; ++i);

		GotoXY(y, x);
		PutChar(' ');

		x += dx;
		y += dy;
		if (x >= 79)
		{
			x = 78;
			dx = -1;
		}
		else if (x <= 0)
		{
			x = 1;
			dx = 1;
		}
		if (y >= 24)
		{
			y = 23;
			dy = -1;
		}
		else if (y <= 0)
		{
			y = 1;
			dy = 1;
		}
	}
```

```
        PutScr(SaveScr);.Procedures and Functions
        GotoXY(savey, savex);
        return 0;
}
```

11.11.2 Ex11_1.asm

```
        _TEXT   segment byte public 'CODE'
        _TEXT   ends

        DGROUP  group   _DATA,_BSS
                assume  cs:_TEXT,ds:DGROUP
        _DATA   segment word public 'DATA'
        d@      label   byte
        d@w     label   word
        _DATA   ends

        _BSS    segment word public 'BSS'
        b@      label   byte
        b@w     label   word
        _BSS    ends

        _TEXT   segment byte public 'CODE'
        ;
        ; int main()
        ;
        assume cs:_TEXT
        _main proc near
                push bp
                mov bp,sp
                sub sp,4012
                push si
                push di
        ;
        ; {
        ;       unsigned SaveScr[25][80];
        ;
        ;       int     dx,
        ;               x,
        ;               dy,
        ;               y;
        ;
        ;       long    i;
        ;
        ;       unsigned        savex,
        ;                       savey;
        ;
        ;
        ;
        ;       GetXY(savex, savey);
        ;
                push    ss
                lea     ax,word ptr [bp-12]
                push    ax
                push    ss
                lea     ax,word ptr [bp-10]
                push    ax
                call    near ptr _GetXY
                add     sp,8
        ;
        ;       Capture(SaveScr);
        ;
```

```
                push    ss
                lea     ax,word ptr [bp-4012]
                push    ax
                call    near ptr _Capture
                pop     cx
                pop     cx
;
;               ClrScrn();
;
                call    near ptr _ClrScrn
;
;
;               GotoXY(24,0);
;
                xor     ax,ax
                push    ax
                mov     ax,24
                push    ax
                call    near ptr _GotoXY
                pop     cx
                pop     cx
;
;               PutStr("Press any key to quit");
;
                push    ds
                mov     ax,offset DGROUP:s@
                push    ax
                call    near ptr _PutStr
                pop     cx
                pop     cx
;
;
;               dx = 1;
;
                mov     word ptr [bp-2],1
;
;               dy = 1;
;
                mov     word ptr [bp-4],1
;
;               x = 1;
;
                mov     si,1
;
;               y = 1;
;
                mov     di,1
                jmp     @1@422
@1@58:
;
;               while (!tstKbd())
;               {
;
;                       GotoXY(y, x);
;
                push    si
                push    di
                call    near ptr _GotoXY
                pop     cx
                pop     cx
;
;               PutChar('#');
;
```

```
        mov     al,35
        push    ax
        call    near ptr _PutChar
        pop     cx
;
;
;       for (i=0; i<500000; ++i);
;
        mov     word ptr [bp-6],0
        mov     word ptr [bp-8],0
        jmp     short @1@114
@1@86:
        add     word ptr [bp-8],1
        adc     word ptr [bp-6],0
@1@114:
        cmp     word ptr [bp-6],7
        jl      short @1@86
        jne     short @1@198
        cmp     word ptr [bp-8],-24288
        jb      short @1@86
@1@198:
;
;
;       GotoXY(y, x);
;
        push    si
        push    di
        call    near ptr _GotoXY
        pop     cx
        pop     cx
;
;       PutChar(' ');
;
        mov     al,32
        push    ax
        call    near ptr _PutChar
        pop     cx
;
;
;
;
;       x += dx;
;
        add     si,word ptr [bp-2]
;
;       y += dy;
;
        add     di,word ptr [bp-4]
;
;       if (x >= 79)
;
        cmp     si,79
        jl      short @1@254
;
;       {
;               x = 78;
;
        mov     si,78
;
;       dx = -1;
;
        mov     word ptr [bp-2],-1
;
```

```
;         }
;
          jmp     short @1@310
@1@254:
;
;         else if (x <= 0)
;
          or      si,si
          jg      short @1@310
;
;         {
;                 x = 1;
;
          mov     si,1
;
;         dx = 1;
;
          mov     word ptr [bp-2],1
@1@310:
;
;         }
;
;         if (y >= 24)
;
          cmp     di,24
          jl      short @1@366
;
;         {
;                 y = 23;
;
          mov     di,23
;
;         dy = -1;
;
          mov     word ptr [bp-4],-1
;
;         }
;
          jmp     short @1@422
@1@366:
;
;         else if (y <= 0)
;
          or      di,di
          jg      short @1@422
;
;         {
;         y = 1;
;
          mov     di,1
;
;         dy = 1;
;
          mov     word ptr [bp-4],1
@1@422:
          call    near ptr _tstKbd
          or      ax,ax
          jne     @@0
          jmp     @1@58
@@0:
;
;         }
;
```

```
        ;
        ; }
        ;
        ; PutScr(SaveScr);
        ;
                 push    ss
                 lea     ax,word ptr [bp-4012]
                 push    ax
                 call    near ptr _PutScr
                 pop     cx
                 pop     cx
        ;
        ; GotoXY(savey, savex);
        ;
                 push    word ptr [bp-10]
                 push    word ptr [bp-12]
                 call    near ptr _GotoXY
                 pop     cx
                 pop     cx
        ;
        ; return 0;
        ;
                 xor     ax,ax
                 jmp     short @1@478
        @1@478:
        ;
        ; }
        ;.
                 pop     di
                 pop     si
                 mov     sp,bp
                 pop     bp
                 ret
        _main    endp

        _TEXT    ends

        _DATA    segment word public 'DATA'
        s@       label   byte
                 db      'Press any key to quit'
                 db      0
        _DATA    ends
        _TEXT    segment byte public 'CODE'
        _TEXT    ends
                 public  _main
                 extrn   _PutStr:near
                 extrn   _PutChar:near
                 extrn   _PutScr:near
                 extrn   _Capture:near
                 extrn   _tstKbd:near
                 extrn   _ClrScrn:near
                 extrn   _GetXY:near
                 extrn   _GotoXY:near
        _s@      equ s@
 end
```

11.11.3 Ex11_1a.asm

```
; Code assembleur à lier avec un programme C/C++.
; Ce code manipule directement l'écran donnant à C++
; le contrôle d'accès direct à l'écran.
;
```

```
; Note: Comme PGM11_1.ASM, ce code est relativement inefficace.
; Il pourrait être beaucoup accéléré en utilisant les instructions
; de chaîne du 80x86. Cependant, son inefficacité est en fait un plus ici,
; dans la mesure où on ne veut pas que le programme C/C++ (Ex11_1.cpp)
; fonctionne trop vite de toutes façons.
;
;
; Ce code présume que Ex11_1.cpp est compilé en utilisant le modèle de
; mémoire LARGE (procs far et pointeurs far).

                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list

                .386                    ; Mettre ces instructions en
                option  segment:use16   ; commentaire si on
                                        ; n'utilise pas de 80386.

; ScrSeg-       C'est l'adresse du segment de l'écran vidéo. Cela devrait
;               être B000 pour les écrans mono et B800 pour les écrans
;               couleur.

ScrSeg  =       0B800h

_TEXT           segment para public 'CODE'
                assume  cs:_TEXT

; _Capture-     Copie les données à l'écran dans le tableau passé
;                       par référence comme paramètre.
;
; procedure Capture(var ScrCopy:array[0..24,0..79] of word);
; var x,y:integer;
; begin
;
;       for y:= 0 to 24 do
;               for x:= 0 to 79 do.
;                       SCREEN[y,x]:= ScrCopy[y,x];
; end;
;
;
; Bloc d'activation pour Capture:
;
;       |                                       |
;       | Précédent contenu de pile             |
;       -------------------------------
;       |       Adrs Seg ScrCopy                |
;       --                                      --
;       |       Adrs offset ScrCopy             |
;       -------------------------------
;       |       Adrs Retour (offset)            |
;       -------------------------------
;       |       valeur coordonnée X             |
;       -------------------------------
;       |       valeur coordonnée Y             |
;       -------------------------------
;       |       Registres, etc.                 |
;       -------------------------------<- SP

ScrCopy_cap     textequ         <dword ptr [bp+4]>
X_cap           textequ         <word ptr [bp-2]>
Y_cap           textequ         <word ptr [bp-4]>
```

```
public _        Capture
_Capture        proc    near
                push    bp
                mov     bp, sp

                push    es
                push    ds
                push    si
                push    di
                pushf
                cld

                mov     si, ScrSeg      ; Établit pointeur sur
                mov     ds, si          ; mémoire SCREEN (ScrSeg:0).
                sub     si, si

                les     di, ScrCopy_cap ; Ptr sur tableau capture.

                mov     cx, 1000        ; 4000 dwords sur l'écran
        rep     movsd

                popf
                pop     di
                pop     si
                pop     ds
                pop     es
                mov     sp, bp
                pop     bp
                ret
_Capture        endp


; _PutScr-      Copie le tableau passé par référence sur l'écran.
;
; procedure PutScr(var ScrCopy:array[0..24,0..79] of word);
; var x,y:integer;
; begin
;
;       for y:= 0 to 24 do
;               for x:= 0 to 79 do
;                       ScrCopy[y,x]:= SCREEN[y,x];
; end;
;.
;
; Bloc d'activation pour PutScr:
;
;       |                                       |
;       | Précédent contenu de pile             |
;       -------------------------------
;       |       Adrs Seg ScrCopy                |
;       --                                      --
;       |       Adrs offset ScrCopy             |
;       -------------------------------
;       |       Adrs Retour (offset)            |
; ----------------------------------
;       |               Valeur BP               | <- BP
;       -------------------------------
;       |       valeur coordonnée X             |
;       -------------------------------
;       |       valeur coordonnée Y             |
;       -------------------------------
;       |       Registres, etc.                 |
;       -------------------------------         <- SP
```

```
ScrCopy_fill    texteq          <dword ptr [bp+4]>
X_fill          texteq          <word ptr [bp-2]>
Y_fill          texteq          <word ptr [bp-4]>

                public  _PutScr
_PutScr         proc    near
                push    bp
                mov     bp, sp
                push    es
                push    ds
                push    si
                push    di
                pushf
                cld

                mov     di, ScrSeg      ; Établit pointeur sur la
                mov     es, di          ; mémoire SCREEN (ScrSeg:0).
                sub     di, di
                lds     si, ScrCopy_cap ; Ptr sur tableau capture.
                mov     cx, 1000        ; 1000 dwords sur l'écran
                rep     movsd

                popf
                pop     di
                pop     si
                pop     ds
                pop     es
                mov     sp, bp
                pop     bp
                ret
_PutScr         endp

; GotoXY-       Positionne le curseur aux coordonnées spécifiées X, Y.
;
; procedure gotoxy(y,x:integer);
; begin
;       BIOS(posnCursor,x,y);
; end;
;
; Bloc d'activation pour GotoXY
;
;       |                                       |
;       | Précédent contenu de pile             |
;       -------------------------------
;       |       valeur coordonnée X             |
;       -------------------------------
;       |       valeur coordonnée Y             |
;       -------------------------------
;       |       Adrs Retour (offset)            |
; ---------------------------------
;       |               Ancien BP               |
;       -------------------------------         <- BP
;       |       Registres, etc.                 |
;       -------------------------------         <- SP

X_gxy   texteq          <byte ptr [bp+6]>
Y_gxy   texteq          <byte ptr [bp+4]>

                public  _GotoXY
_GotoXY         proc    near
                push    bp
                mov     bp, sp
```

```
                mov     ah, 2           ; Valeur Magique BIOS pour gotoxy.
                mov     bh, 0           ; Affiche page zéro.
                mov     dh, Y_gxy       ; Etablit paramètres BIOS (X,Y).
                mov     dl, X_gxy
                int     10h             ; Appelle le BIOS.

                mov     sp, bp
                pop     bp
                ret
_GotoXY         endp

; ClrScrn-      Efface l'écran et positionne le curseur à (0,0).
;
; procedure ClrScrn;
; begin
;       BIOS(Initialize)
; end;
;
; Bloc d'activation pour ClrScrn
;
;       |                                       |
;       | Précédent contenu de pile             |
;       -------------------------------
;       |       Adrs Retour (offset)            |
; ---------------------------------
;       |       Registres, etc.                 |
;       -------------------------------         <- SP

                public _ClrScrn
_ClrScrn        proc    near

                mov     ah, 6           ; Numéro Magique BIOS.
                mov     al, 0           ; Efface tout l'écran.
                mov     bh, 07          ; Efface avec espaces noirs.
                mov     cx, 0000        ; Coin gauche haut est (0,0)
                mov     dl, 79          ; Coordonnée X du bas
                mov     dh, 24          ; Coordonnée Y du bas
                int     10h             ; Appelle le BIOS.

                push    0               ; Positionne le curseur à
                push    0               ; (0,0) après l'appel.
                call    _GotoXY
                add     sp, 4           ; Enlève params de la pile.

                ret
_ClrScrn        endp

; tstKbd-       Vérifie si une touche est disponible au clavier.
;
; function tstKbd:boolean;
; begin
;       if BIOSKeyAvail then avale la touche et renvoie true
;       else renvoie false;
; end;
;
; Bloc d'activation pour tstKbd
;
;       |                                       |
;       | Précédent contenu de pile             |
;       -------------------------------
;       |       Adrs Retour (offset)            |
;       -------------------------------         <- SP
```

```
                public _tstKbd
_tstKbd         proc    near
                mov     ah, 1           ; Vérifie si touche disponible.
                int     16h
                je      NoKey
                mov     ah, 0           ; Avale la touche si oui.
                int     16h
                mov     ax, 1           ; Retourne true.
                Ret

NoKey:          mov     ax, 0           ; Pas de touche, alors retourne false.
                ret
_tstKbd         endp

; GetXY-        Retourne les coordonnées courantes du curseur X et Y.
;
; procedure GetXY(var x:integer; var y:integer);
;
; Bloc d'activation pour GetXY
;
;       |                                       |
;       | Précédent contenu de pile             |
;       -----------------------------
;       |               Adresse                 |
;       ---         Coordonnée     ---
;       |                 Y                     |
;       -----------------------------
;       |               Adresse                 |
;       ---         Coordonnée     ---
;       |                 X                     |
;       -----------------------------
;       |       Adrs Retour (offset)            |
;       -----------------------------
;       |               Ancien BP               |
;       ----------------------------            <- BP
;       |       Registres, etc.                 |
;       ----------------------------            <- SP

GXY_X           textequ         <[bp+4]>
GXY_Y           textequ         <[bp+8]>

                public  _GetXY
_GetXY          proc    near
                push    bp
                mov     bp, sp
                push    es

                mov     ah, 3           ; Lit coordonnées X, Y via le
                mov     bh, 0           ; BIOS
                int     10h

                les     bx, GXY_X
                mov     es:[bx], dl
                mov     byte ptr es:[bx+1], 0

                les     bx, GXY_Y
                mov     es:[bx], dh
                mov     byte ptr es:[bx+1], 0

                pop     es
                pop     bp
                ret
_GetXY          endp
```

```
; PutChar-      Imprime un unique caractère à l'écran à la position
; courante du curseur.
;
; procedure PutChar(ch:char);
;
; Bloc d'activation pour PutChar
;
;       |                                   |
;       | Précédent contenu de pile         |
;       -------------------------------
;       |       char (dans L.O. byte)       |
;       -------------------------------
;       |       Adrs Retour (offset)        |
;       -------------------------------
;       |               Ancien BP           |
;       -------------------------------     <- BP
;       |       Registres, etc.             |
;       -------------------------------     <- SP

ch_pc           textequ <[bp+4]>
                public  _PutChar
_PutChar        proc    near
                push    bp
                mov     bp, sp

                mov     al, ch_pc
                mov     ah, 0eh
                int     10h

                pop     bp
                ret
_PutChar        endp

; PutStr-       Imprime une chaîne à l'écran à la position courante du curseur.
; Notez qu'une chaîne est une sequence de caractères qui finit par
; un byte à zéro.
;
; procedure PutStr(var str:string);
;
; Bloc d'activation pour PutStr
;
;       |                               |
;       | Précédent contenu de pile     |
;       -------------------------------
;       |               Adresse         |
;       ---        de la chaîne         --
;       |                               |
;       -------------------------------
;       |       Adrs Retour (offset)    |
;       -------------------------------
;       |               Ancien BP       |
;       -------------------------------   <- BP
;       |       Registres, etc.         |
;       -------------------------------   <- SP

Str_ps  textequ         <[bp+4]>

                public  _PutStr
_PutStr         proc    near
                push    bp
                mov     bp, sp
                push    es
```

```
                les     bx, Str_ps
PS_Loop:        mov     al, es:[bx]
                cmp     al, 0
                je      PC_Done

                push    ax
                call    _PutChar
                pop     ax
                inc     bx
                jmp     PS_Loop

PC_Done:        pop     es
                pop     bp
                ret
_PutStr         endp


; StartPgm-     C'est l'endroit où le DOS commence le programme. C'est
;                       un substitut pour le fichier C0L.OBJ normalement lié par
;                       le compilateur Borland C++. Ce code fournit cette routine
;                       pour éviter des problèmes légaux (c.à.d., distribuer des
;                       bibliothèques Borland non liées). Vous pouvez ignorer
;                       sans souci ce code. Notez que le program principal C++
;                       est une procédure near aussi ce code doit faire partie
;                       du segment _TEXT.

                extern  _main:near
StartPgm        proc    near

                mov     ax, _DATA
                mov     ds, ax
                mov     es, ax
                mov     ss, ax
                lea     sp, EndStk

                call    near ptr _main
                mov     ah, 4ch
                int     21h
StartPgm        endp

_TEXT           ends

_DATA           segment word public "DATA"
stack           word    1000h dup (?)
EndStk          word    ?
_DATA           ends

sseg            segment para stack 'STACK'
                word    1000h dup (?)
sseg            ends
                end     StartPgm
```

11.12 Projets de programmation

1) Écrivez une version du programme de multiplication de matrice qui reçoit deux matrices de 4x4 nombres entiers de l'utilisateur et calcule leur produit matriciel (voir l'exposé de la question au Chapitre Huit). L'algorithme de multiplication de matrice (qui calcule C : = A * B) est

```
for i := 0 to 3 do
       for j := 0 to 3 do begin
```

```
                c[i,j] := 0;
                for k := 0 to 3 do
                        c[i,j] := c[i,j] + a[i,k] * b[k,j];
    end;
```

Le programme devrait avoir trois procédures: InputMatrix, PrintMatrix, et MatrixMul. Elles ont les prototypes suivants:

```
Procedure InputMatrix(var m:matrix);
procedure PrintMatrix(var m:matrix);
procedure MatrixMul(var result, A, B:matrix);
```

Notez en particulier que ces routines passent toutes leurs paramètres par référence. Passez ces paramètres par référence sur la pile.

Maintenez toutes les variables (par exemple, I, j, k, etc...) sur la pile en utilisant les techniques décrites dans "Stockage Local de Variables" à la section 11.8. En particulier, ne maintenez pas les variables de contrôle de boucle dans un registre.

Écrivez un programme principal qui fait les appels appropriés à ces routines pour les tester.

2) Un passage de paramètre par évaluation paresseuse est généralement une structure avec trois champs: un pointeur sur le thunk à appeler pour la fonction qui calcule la valeur, un champ pour contenir la valeur du paramètre, et un champ booléen qui contient faux si le champ de valeur est non-initialisé (le champ de valeur devient initialisé si la procédure écrit dans le champ de valeur ou appelle le thunk pour obtenir la valeur). Toutes les fois que la procédure écrit une valeur dans un paramètre passé par évaluation paresseuse, elle stocke la valeur dans le champ valeur et met le champ booléen à vrai. Toutes les fois qu'une procédure veut lire la valeur, elle contrôle en premier ce champ booléen. S'il contient la valeur vraie, elle lit simplement la valeur du champ valeur; si le champ booléen contient faux, la procédure appelle le thunk pour calculer la valeur initiale. Au retour, la procédure stocke le résultat de thunk dans le champ valeur et met le champ booléen à vrai. Notez que pendant une activation unique d'une procédure, le thunk pour un paramètre sera appelé, tout au plus, une fois. Considérez la procédure Panacea suivante:

```
SampleEval: procedure(select:boolean; eval a:integer; eval b:integer);
    var
            result:integer;
    endvar;
    begin SimpleEval;

            if (select) then

                    result := a;
            else

                    result := b;

            endif;
            writeln(result+2);

 end SampleEval;
```

Écrivez un programme en assembleur qui implémente SampleEval. Depuis votre programme principal appellez SampleEval plusieurs fois en lui passant différents thunks comme paramètres a et b. Vos thunks peuvent simplement renvoyer une valeur unique quand ils sont appelés.

3) Écrivez une routine qui bat des cartes à laquelle vous passez un tableau de 52 nombres entiers par référence. La routine devrait remplir le tableau de valeurs 1..52 et puis mélanger aléatoirement les éléments du tableau. Utilisez les routines de la Bibliothèque Standard random et randomize pour choisir un index dans la tableau pour le permuter. Voir le Chapitre Sept, "Génération de Nombres Aléatoires: Random, Randomize" pour plus de détails au sujet de la fonction random. Écrivez un programme principal qui passe une tableau à cette procédure et imprime le résultat.

11.13 Résumé

Dans un programme en assembleur, tout ce dont vous avez besoin est une instruction call et une instruction ret pour implémenter des procédures et des fonctions. Le Chapitre Sept couvre l'utilisation basique des procédures dans un programme de langage en assembleur 80x86; ce chapitre décrit comment organiser des unités de programme comme des procédures et des fonctions, comment passer des paramètres, allouer et accéder à des variables locales, et autres questions relatives à ce sujet.

Ce chapitre commence par un examen de ce qu'est une procédure, comment implémenter des procédures avec MASM, et la différence entre les procédures proches et lointaines sur le 80x86. Pour des détails, voyez les sections suivantes:

- "Procedures" (11.1)
- "Procédures Near et Far" (11.2)
- "Forcer des appels et des Retours NEAR ou FAR" (11.2.1)
- "Procédures Emboîtées" (11.2.2)

Les fonctions sont une construction très importante dans les langages de haut niveau comme le Pascal. Cependant, il n'y a pas vraiment de différence entre une fonction et une procédure dans un programme en assembleur. Logiquement, une fonction renvoie un résultat et une procédure non; mais vous déclarez et appelez les procédures et les fonctions de manière identique dans un programme en assembleur. Voyez

- "Fonctions" (11.3)

Les procédures et les fonctions produisent souvent des *effets de bord (side effects*). C'est-à-dire, elles modifient les valeurs des registres et des variables non-locales. Souvent, ces effets secondaires sont indésirables. Par exemple, une procédure peut modifier un registre que l'appelant a besoin de préserver. Il y a deux mécanismes de base pour préserver de telles valeurs: sauvegarde par l'appelé et sauvegarde par l'appelant. Pour des détails sur ces schémas de sauvegarde et autres questions importantes voyez

- "Sauvegarder l'état de la machine" (11.4)
- "Les effets de bord" (11.7)

Un des avantages principaux d'utiliser un langage procédural comme le Pascal ou le C++ est que vous pouvez facilement passer des paramètres à des procédures et des fonctions. Bien que cela demande un peu plus de travail, vous pouvez aussi passer des paramètres à vos fonctions et procédures en assembleur. Ce chapitre discute comment et où passer des paramètres. Il discute également comment accéder aux paramètres à l'intérieur d'une procédure ou d'une fonction. Pour avoir connaissance de ceci, voir les sections

- "Paramètres" (11.5)
- "Passage par valeur" (11.5.1)
- "Passage par référence" (11.5.2)
- "Passage par valeur-retournée" (11.5.3)
- "Passage par nom" (11.5.5)
- "Passage par évaluation paresseuse" (11.5.6)
- "Passage de paramètres dans des registres" (11.5.7)
- "Passage de paramètres dans des variables globales" (11.5.8)
- "Passage de paramètres sur la pile" (11.5.9)
- "Passage de paramètres dans le code lui-même" (11.5.10)
- "Passage de paramètres via bloc de paramètres" (11.5.11)

Puisque l'assembleur ne supporte pas vraiment la notion de fonction, per se, implémenter une fonction consiste à écrire une procédure avec un paramètre de retour. En tant que tels, les résultats de fonction sont tout à fait semblables aux paramètres à bien des égards. Pour voir les similitudes, voyez les sections suivantes:

- Résultats de fonction" (11.6)
- " Retour de résultats de fonction dans un registre" (11.6.1)

- "Retourner des résultats de fonction sur la pile" (11.6.2)
- "Retourner des résultats de fonction dans des emplacements de mémoire (11.6.3)

La plupart des langages de haut niveau fournissent le *stockage de variables locales* lié à l'activation et à la désactivation d'une procédure ou d'une fonction. Bien que peu de programmes en assembleur utilisent des variables locales d'une façon identique, il est très faciles d'implémenter l'allocation dynamique des variables locales sur la pile. Pour des détails, voir la section

- "Stockage de variables locales" (11.8)

La récursion est une autre facilité de HLL qu'il est très facile d'implémenter dans un programme en assembleur. Ce chapitre discute la technique de la récursion et puis présente un exemple simple en utilisant l'algorithme Quicksort. Voyez

- "La récursion" (11.9)

11.14 Questions

1. Expliquez comment les instructions CALL et RET fonctionnent.
2. Quelles sont les opérandes pour la directive assembleur PROC ? Quelle est leur fonction ?
3. Récrivez le code suivant en utilisant PROC et ENDP.

```
FillMem:        mov     al, 0FFh
FillLoop:       mov     [bx], al
                inc     bx
                loop    FillLoop
                ret
```

4. Modifiez votre réponse au problème (3) de sorte que tous les registres affectés soient préservés par la procédure FillMem.
5. Que se produit-il si vous ne mettez pas un transfert de contrôle d'instruction (tel qu'un JMP ou un RET) juste avant la directive ENDP dans une procédure?
6. Comment l'assembleur détermine-t-il si un CALL est proche ou lointain? Comment détermine-t-il si une instruction RET est proche ou lointaine?
7. Comment pouvez-vous surcharger la décision par défaut de l'assembleur pour utiliser un CALL ou un RET proche ou lointain?
8. Y a-t-il parfois besoin de procédures emboîtées dans un programme en assembleur? Si oui, donnez un exemple.
9. Donnez un exemple de raison pour laquelle vous pourriez vouloir emboîter un segment à l'intérieur d'une procédure.
10. Quelle est la différence entre une fonction et une procédure?
11. Pourquoi les sous-programmes devraient-ils préserver les registres qu'ils modifient?
12. Quels sont les avantages et les inconvénients de la sauvegarde des valeurs par l'appelant et de la sauvegarde des valeurs par l'appelé?
13. Qu'est-ce que sont des paramètres?
14. Comment fonctionnent les mécanismes de passage de paramètre suivants ?

    a) Passage par valeur

    b) Passage par référence

    c) Passage par nom
15. Quel est le meilleur emplacement pour passer des paramètres à une procédure?
16. Énumérez cinq endroits/méthodes différent pour passer des paramètres à une procédure ou d'une procédure.
17. Comment est-ce que des paramètres qui sont passés sur la pile sont accédés dans une procédure?
18. Quelle est la meilleure manière de désallouer des paramètres passés sur la pile quand la procédure termine son exécution?
19. Étant donné la définition de procédure Pascal suivante :

```
procedure PascalProc (i, j, k:integer);
```

    Expliquez comment vous accéderiez aux paramètres d'un programme équivalent en assembleur, à supposer que la procédure est une procédure near.
20. Répétez le problème (19) en supposant que la procédure est une procédure far.

21. À quoi ressemble la pile pendant l'exécution de la procédure dans le problème (19)? Dans le problème (20)?
22. Comment est-ce qu'une procédure en assembleur accède aux paramètres passés dans le code?
23. Comment fait le 80x86 pour sauter par-dessus des paramètres passés dans le code et continuer l'exécution du programme au delà de ceux-ci quand la procédure revient à l'appelant?
24. Quel est l'avantage de passer des paramètres par l'intermédiaire d'un bloc de paramètre?
25. Quels sont les résultats de fonction sont généralement retournés?
26. Qu'est-ce que qu'un effet de bord ?
27. Où sont généralement allouées les variables locales (provisoires)?
28. Comment allouez-vous des variables locales (provisoires) dans une procédure?
29. En supposant que vous avez passé trois paramètres par valeur sur la pile et 4 variables locales différentes, à quoi le bloc d'activation ressemble-t-il après que les variables locales aient été allouées (présumez qu'il s'agit d'une procédure proche et qu'aucun registre autres que bp n'ont été poussés sur la pile).
30. Qu'est-ce que la récursion ?