ROZDZIAŁ JEDENASTY: **PROCEDURY I FUNKCJE**

Budowa modularna jest jednym z kamieni węgielnych programowania strukturalnego. Program modularny zawiera bloki kodu z pojedynczym punktem wejścia i wyjścia. Możemy użyć ponownie dobrze napisaną sekcję kodu w innych programach lub innej części istniejącego programu. Jeśli użyjemy ponownie istniejącą część kodu ,nie musimy tworzyć kodu, ani też debuggować tej części kodu ponieważ (przypuszczalnie) zostało to już zrobione. Przy rosnących kosztach projektowania oprogramowania, budowa modularna staje się bardzo ważna ponieważ oszczędza czas.

Podstawową jednostką program u modularnego jest moduł. Moduły mają różne znaczenia dla różnych ludzi ,tutaj możemy założyć, że terminy moduł, podprogram, podprocedura, jednostka programowa ,procedura i funkcja wszystkie są synonimami.

Procedura jest podstawowa dla stylu programowania. Języki proceduralne to Pascal, BASIC,C++,FORTRAN,PL/I i ALGOL Przykłady języków nie proceduralnych to APL,LISP,SNOBOL4 ICON,FORTH,SETL,PROLOG i inne ,które są oparte o inne konstrukcje programistyczne takie jak funkcjonalna abstrakcja lub dopasowanie do wzorca .Język asemblera jest zdolny pełnić obowiązki języka proceduralnego lub nieproceduralnego .ponieważ prawdopodobnie jesteśmy dużo bardziej zapoznani z paradygmatami programowania proceduralnego, ten tekst trzymać się będzie stymulowania konstrukcji proceduralnych w języku asemblera 80x86

11.0 WSTĘP

Rozdział ten przedstawia wprowadzenie do procedur i funkcji w asemblerze. Omawia podstawowe zasady, przekazywanie parametrów, wyników funkcji, zmiennych lokalnych i rekurencje Zastosujemy większość technik, które ten rozdział omawia w typowych programach asemblerowych. Omawianie procedur i funkcji będzie kontynuowane w następnym rozdziale; ten rozdział omawia zaawansowane techniki, które nie są powszechnie stosowane w programach asemblerowych. poniższa część, która ma przedrostek „•” jest niezbędna. Te części z „⊗” omawiają zaawansowane tematy które możemy zechcieć odłożyć na później.

- • Procedury
- ⊗ Procedury bliskie i dalekie
- • Funkcje
- • Zachowywanie stanów maszyny
- • Parametry
- • Przekazywanie parametrów przez wartość
- • Przekazywanie parametrów przez referencję
- ⊗ Przekazywanie parametrów przez wartość zwrotną
- ⊗ Przekazywanie parametrów przez wynik
- ⊗ Przekazywanie parametrów przez nazwę
- • Przekazywanie parametrów w rejestrach
- • Przekazywanie parametrów w zmiennych globalnych
- • Przekazywanie parametrów na stos
- • Przekazywanie parametrów w strumieniu kodu

⊗ Przekazywanie parametrów przez blok parametrów
• Wyniki funkcji
• Zwracanie wyniku funkcji w rejestrach
• Zwracanie wyniku funkcji na stos
• Zwracanie wyniku funkcji w komórkach pamięci
• Efekty uboczne
⊗ Przechowywanie zmiennych lokalnych
⊗ Rekurencja

11.1 PROCEDURY

W środowisku proceduralnym, podstawową jednostką kodu jest procedura. Procedura jest zbiorem instrukcji, które obliczają jaką wartość lub wykonują jakąś czynność (taką jak drukowanie lub odczytywanie wartości znaku). Definicja procedury jest bardzo podobna do definicji algorytmu. Procedura jest zbiorem zasad następujących po sobie, które jeśli są zakończone tworzą jakiś wynik. Algorytm jest również taką sekwencją, ale algorytm gwarantuje zakończenie podczas gdy procedura takiej gwarancji nie daje.

Wiele języków proceduralnych implementuje procedury stosując mechanizm call / ret. To znaczy, jakiś kod wywołuje procedurę, procedura robi swoje a potem wraca do miejsca skąd została wywołana. Instrukcje call i return dostarczają mechanizmy wywołania procedury 80x86.Kod wywołujący wywołuje procedurę instrukcją call, procedura wraca do miejsca wywołania instrukcją ret .Na przykład, poniższa instrukcja 80x86 wywołuje podprogram Biblioteki Standardowej UCR sL_putcr:

```
        call    sl_putcr
```

sl_putcr drukuje sekwencję carriage return /line feed na wyświetlaczu i zwraca sterowanie do instrukcji bezpośrednio po instrukcji call sl_putrc.

Niestety, Biblioteka Standardowa UCR nie dostarcza wszystkich podprogramów jakich będziemy potrzebowali .Większość czasu, będziemy musieli pisać swoje własne procedury. Prosta procedura może składać się z niczego więcej niż tylko sekwencji instrukcji zakończenia z instrukcją ret. Na przykład, poniższa „procedura' zeruje 256 bajtów zaczynając od adresu w rejestrze bx:

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

Przez załadowanie rejestru bx adresem bloku 256 bajtów i wykonanie instrukcji call ZeroBytes, możemy wyzerować określony blok.

Jako generalna zasada nie możemy zdefiniować własnej procedury w ten sposób, zamiast tego powinniśmy użyć dyrektyw MASM'a proc i endp. Podprogram ZeroByte stosujący dyrektywy proc i endp:

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

Zapamiętajmy, że proc i endp są dyrektywami asemblera. Nie generują żadnego kodu .Są one mechanizmami pomagającymi uczynić nasz program łatwiejszym do czytania. Dla 80x86 dwa ostatnie przykłady są identyczne; jednakże dla istoty ludzkiej, ostatni jest wyraźnie samodzielną procedurą, inna może być po prostu przypadkowym zbiorem instrukcji wewnątrz jakiejś innej procedury. Rozważmy teraz poniższy kod:

```
ZeroBytes:      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
```

Czy są to dwie procedury czy tylko jedna? Innymi słowy, możemy wywoływać program wprowadzając ten kod przy etykiecie ZeroBytes i DoFFs lub tylko ZeroBytes? Zastosowanie dyrektyw proc i endp może pomóc usunąć tę niejasność:

Traktujemy jako pojedynczy podprogram:

```
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
```

Traktujemy jako dwa oddzielne podprogramy:

```
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
```

Zawsze pamiętajmy, że dyrektywy proc i endp są logicznymi separatorami procedury. Mikroprocesor wraca z procedury wykonując instrukcję ret a nie poprzez napotkanie dyrektywy endp .Poniższy kod nie jest odpowiednikiem kodu powyższego:

```
ZeroBytes       proc
                xor     ax, ax
                jcxz    DoFFs
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
;  zagubiona instrukcja RET

ZeroBytes       endp

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

```
                loop        FFLoop

; zaginiona instrukcja RET

DoFFs           endp
```

Bez instrukcji ret na końcu każdej procedury,80x86 przejdzie do następnego podprogramu zamiast wrócić do miejsca wywołania. Po wykonaniu ZeroBytes, 80x86 zacznie wykonywać podprogram DoFFs (zaczynając od instrukcji mov cx, 128).Zaraz potem, 80x86 zacznie kontynuowanie wykonywania następnych instrukcji aż do dyrektywy endp DoFFs

Procedura 80x86 przyjmuje formę:

```
ProcName        proc        {near | far}        ;wybierz near lub far
        <instrukcje procedury>
ProcName        endp
```

Operand near lub far jest opcjonalny, następna sekcja omawia ich cele. Nazwa procedury musi być zarówno w linii proc jaki i endp. Nazwa procedury musi być unikalna w programie.

Każda dyrektywa proc musi mieć dopasowaną dyrektywę endp. Błędne dopasowanie dyrektyw proc i endp wywoła błąd zagnieżdżenia bloku.

11.2 PROCEDURY BLISKIE I DALEKIE

80x86 wspiera podprogramy. Wywołanie bliskie i powrót przekazują sterowanie danymi pomiędzy procedurami w tym samym segmencie kodu. Dalekie wywołanie i powrót przekazują sterowanie między różnymi segmentami. Te dwa mechanizmy wywołania i powrotu odkładają i ściągają adresy powrotne. Generalnie nie stosujemy bliskiej instrukcji call do wywołania dalekiej procedury lub dalekiej instrukcji call do wywołania bliskiej procedury. Opierając się na tej zasadzie, powstaje pytanie :jak możemy sterować emisją bliskiego lub dalekiego call lub ret?"

Większość czasu, instrukcja call stosuje następującą składnię:

```
                call        ProcName
```

a instrukcja ret :

```
                ret
lub             ret         przemieszczenie
```

Niestety, instrukcje te nie mówią MASMowi czy wywołujemy bliską czy daleką procedurę lub czy wracamy z dalekiej czy bliskiej procedury. Dyrektywa proc zajmuje się tym. dyrektywa proc ma opcjonalny operand, który jest albo bliski albo daleki. Bliski jest domyślny ,jeśli pole operandu jest puste .Assembler przydziela typ procedury (bliska lub daleka) do symbolu. Kiedykolwiek MASM asembluje instrukcję call, emituje bliskie lub dalekie wywołanie w zależności od operandu .Dlatego też deklarowanie symbolu proc lub proc near wymusza bliskie wywołanie. Podobnie zastosowanie proc far wymusza dalekie wywołanie.

Poza sterowaniem generowania bliskiego lub dalekiego wywołania, operand proc również steruje generowaniem kodu ret. Jeśli procedura ma bliski operand, wtedy wszystkie powroty instrukcji wewnątrz tej procedury będą bliskie.. MASM emituje dalekie powroty wewnątrz dalekich procedur.

11.2.1 WYMUSZANIE BLISKICH LUB DALEKICH WYWOŁAŃ I POWROTÓW

Raz na jakiś czas możemy chcieć zastąpić mechanizm deklaracji near / far. MASM dostarcza mechanizmu, który pozwala nam wymusić zastosowanie wywołań bliskie/ dalekie i powrotów.

Stosujemy operatory near ptr i far ptr do zastąpienia automatycznie przydzielonego wywołania bliskiego lub dalekiego. Jeśli NearLbl jest bliską etykietą a FarLbl jest etykietą daleką, wtedy następująca instrukcja call wygeneruje odpowiednio bliskie i dalekie wywołanie:

```
                        call    NearLbl                 ;generowanie bliskiego wywołania
                        call    FarLbl                  ;generowanie dalekiego wywołania
```

Przypuśćmy, że musimy wykonać dalekie wywołanie do NearLbl lub bliskie wywołanie do FarLbl .Możemy to wykonać stosując poniższe instrukcje:

```
                        call    far ptr NearLbl         ;generowanie dalekiego wywołania
                        call    near ptr FarLbl         ;generowanie bliskiego wywołania
```

Wywołując bliską procedurę stosujemy daleki call lub wywołując daleką procedurę stosując bliski call, to nie jest coś co będziemy normalnie robić. jeśli wywołamy bliską procedurę stosując daleką instrukcję call, bliski powrót pozostanie wartością cs na stosie. Generalnie, zamiast:

```
                        call    far ptr NearProc
```

powinniśmy zastosować jaśniejszy kod:

```
                        push    cs
                        call    NearProc.
```

Wywoływanie dalekiej procedury bliskim call jest niebezpieczną operacja. Jeśli spróbujemy takiego wywołania, bieżąca wartość cs musi być na stosie. Pamiętajmy, że daleki ret zdejmuje segmentowy adres powrotu ze stosu .Bliska instrukcja call odkłada tylko offset a nie segmentową część adresu powrotnego.

Poczynając od MASM v5.0, są jasne instrukcje, które możemy użyć dla wymuszenia bliskiego lub dalekiego ret. Jeśli ret pojawia się wewnątrz procedury deklarowanej przez proc i endp, MASM automatycznie wygeneruje właściwą instrukcję bliskiego lub dalekiego powrotu. Wykonując to użyjemy instrukcji retn i retf .Te dwie instrukcje generują odpowiednio bliski i daleki ret.

11.2.2 PROCEDURY ZAGNIEŻDŻONE

MASM pozwala nam zagnieżdżać procedury. To znaczy, definicja jednej procedury może być całkowicie otoczoną wewnątrz innej. Poniżej jest przykład takiej pary procedur:

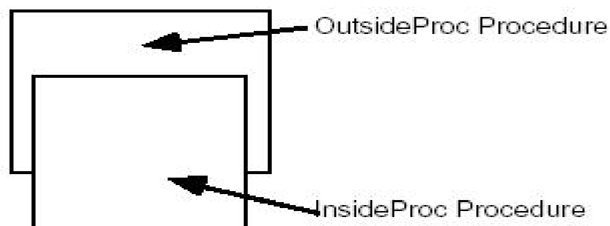
```
OutsideProc:        proc    near
                    jmp     EndofOutside
InsideProc          proc    near
                    mov     ax, 0
                    ret
                    endp
EndofOutside:       call    InsideProc
                    mov     bx,0
                    ret
OutsideProc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

W odróżnieniu od języków wysokiego poziomu, zagnieżdżanie procedur w asemblerze 80x86 ni służy żadnemu użytecznemu celowi. Jeśli zagnieżdżamy procedurę (jak powyższa InsideProc), będziemy musieli wyraźnie zakodować jmp wokoło zagnieżdżonej procedury. Umieszczając procedury zagnieżdżonej po całym kodzie procedury zewnętrznej (ale jeszcze pomiędzy dyrektywami zewnętrznymi proc /endp) nie osiągamy niczego. Dlatego też, to nie jest dobry powód zagnieżdżania procedur w ten sposób.

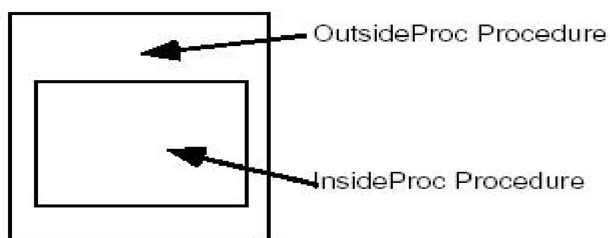
Kiedy zagnieżdżamy jedną procedurę wewnątrz innej, musi być całkowicie zawarta wewnątrz procedury zagnieżdżającej. To znaczy, instrukcje proc i endp dla procedury zagnieżdżanej muszą leżeć pomiędzy dyrektywami proc i endp zewnętrznymi zagnieżdżającej procedury. Poniższy kod nie jest poprawny:

```
OutsideProc         proc    near
                    -
                    -
                    -
InsideProc          proc    near
                    -
                    -
                    -
OutsideProc         endp
                    -
                    -
                    -
InsideProc          endp
```

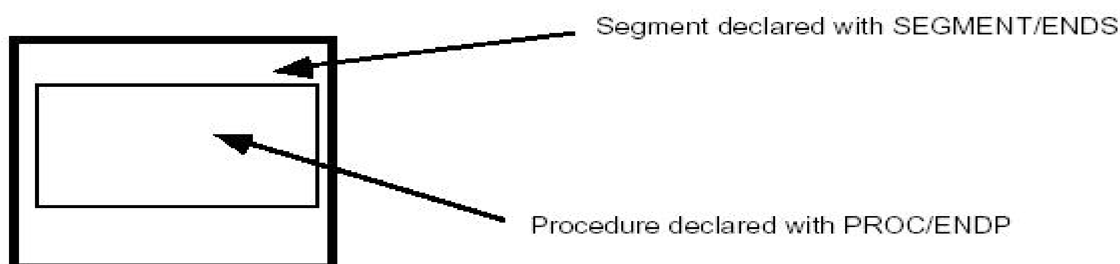
Procedury OutsideProc i InsideProc zachodzą na siebie, nie są zagnieżdżone. Jeśli spróbujemy stworzyć zbiór procedur takich jak ta, MASM zaraportuje „błąd zagnieżdżanego bloku”. Rysunek 11 demonstruje to graficznie:



Rysunek 11.1 Niepoprawne zagnieżdżanie procedur



Rysunek 11.2 Poprawne zagnieżdżanie procedur



Rysunek 11.3 poprawne zagnieżdżanie Procedura/Segment

Jedyną akceptowalną formą przez MASM jest ta pojawiająca się na rysunku 11.2

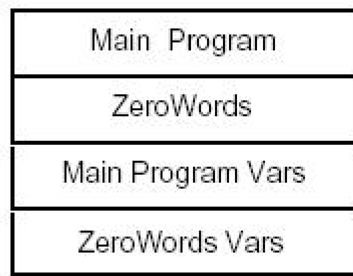
Poza dopasowaniem wewnątrz otaczającej procedury, grupa proc/endp musi dopasować się całkowicie wewnątrz segmentu. Dlatego poniższy kod jest niepoprawny:

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

Dyrektywa endp musi pojawić się przed instrukcją cseg ends ponieważ MyProc zaczyna się wewnątrz cseg. Dlatego też procedury wewnątrz segmentów muszą zawsze przybierać formę pokazaną na rysunku 11.3.

Nie tylko możemy gnieździć procedury wewnątrz innych procedur i segmentów ,ale możemy zagnieżdżać również segmenty wewnątrz procedur i segmentów. Jeśli lubimy symulować procedury C lub Pascala w asemblerze, możemy stworzyć zmienną deklarującą sekcje na początku każdej procedury jaka tworzymy, podobnie jak w Pascalu:

```
cgroup      group   cseg1, cseg2

cseg1       segment  para public  'code'
cseg1       ends

cseg2       segment  para public 'code'
cseg2       ends
```



Rysunek 11.4 Przykład rozmieszczenia pamięci

```
dseg        segment     para  public 'data'
dseg        ends

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

MainPgm     proc      near
;deklaracja danych dla głównego programu:

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

; Procedury które są lokalne w głównym programie:

cseg2       segment  para  public 'code'

ZeroWords   proc      near

;zmienne lokalne dla ZeroBytes:

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

;kod dla procedury 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
```

```
; rzeczywisty główny program zaczyna się tutaj:
                mov     bx, offset Array
                mov     cx, 128
                call    ZeroWords
                ret
MainPgm         endp
cseg1           ends
                end
```

System załaduje ten kod do pamięci jak pokazano na rysunku 11.4

ZeroWords występuje w programie głównym ponieważ należy do innego segmentu (cseg2) niż MainPgm (cseg1).Pamiętamy, że asembler i linker łączą segmenty o tej samej nazwie klasy w pojedynczy segment przed załadowaniem do pamięci. Możemy zastosować tą cechę asemblera do „pseudo-paskalizacji" naszego kodu w powyżej pokazany sposób. Jednakże ,prawdopodobnie nie uczynimy naszego programu bardziej czytelnym niż stosując proste podejście nie zagnieżdżania.

11.3 FUNKCJE

Różnica pomiędzy funkcją a procedurą w języku asemblera jest głównie w sposobie definiowania. Celem dla funkcji jest zwrócenie jakiejś konkretnej wartości, podczas gdy celem funkcji jest wykonanie jakiegoś działania. Dla deklarowania funkcji w asemblerze stosujemy dyrektywy proc/ endp. Wszystkie zasady i techniki jakie stosujemy dla procedury stosujemy również dla funkcji. tekst ten będzie omawiał później funkcje w tym rozdziale w sekcji o wynikach funkcji. Na razie procedury będą znaczyły procedura lub funkcja

---

11.4 ZACHOWANIE STANÓW MASZYNY

Spójrzmy na ten kod:

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

Ta część kodu próbuje wydrukować dziesięć linii każda po 40 spacji. Niestety jest subtelny błąd, który powoduje, że drukuje się 40 spacji na linię w pętli nieskończonej. Program główny stosuje instrukcję loop do wywołania PrintSpaces 10 razy. PrintSpaces używa cx do zliczania 40 spacji do druku. PrintSpaces wraca kiedy cx zawiera zero. Program główny drukuje wtedy, carriage return / line feed , zmniejsza cx i zaczyna powtórkę ponieważ cx nie jest zerem (zawsze będzie zawierać 0FFFFh w tym miejscu)

Tu problem jest taki, że podprogram PrintSpaces nie przechowuje rejestru cx. Zachowanie rejestru znaczy, że zachowujemy go na wejściu do podprogramu i przywracamy przed wyjściem. Mając podprogram PrintSpaces zachowujący zawartość rejestru cx, powyższy program funkcjonowałby poprawnie.

Stosujemy instrukcje 80x86 push i pop do przechowania wartości rejestru, podczas gdy używamy ich do czegoś innego. Rozważmy poniższy kod dla PrintSpaces:

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

Zauważmy, że PrintSpaces zachowuje i przywraca ax i cx (ponieważ procedura ta modyfikuje te rejestry). Również zauważmy, że ten kod zdejmuje rejestry ze stosu w odwrotnej kolejności, niż je tam odkładał. Operacje na stosie narzucają taką kolejność.

Albo wywołujący (kod zawierający instrukcję call) albo wywoływany (podprogram) mogą wziąć odpowiedzialność za zachowanie rejestrów .W powyższym przykładzie wywoływany zachowuje rejestry. Poniższy przykład pokazuje jak może wyglądać ten kod, jeśli wywołujący zachowuje rejestry:

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

Są dwie korzyści z zachowywania przez wywoływanego: miejsce i pielęgnowalność. Jeśli wywoływany przechowuje wszystkie wykorzystywane rejestry, wtedy jest tylko jedna kopia instrukcji push i pop, które zawiera ta procedura. Kiedy wywołujący zachowuje wartości w rejestrach, program potrzebuje zbioru instrukcji push i pop przy każdym wywołaniu. To nie tylko czyni nasze programy dłuższymi, czyni je również trudniejszymi do pielęgnacji. Pamiętanie które rejestry odkładamy i zdejmuje przy każdym wywołaniu procedury nie jest czymś łatwym do zrobienia.

Z drugiej strony, podprogram może niepotrzebnie przechować jakieś rejestry, jeśli przechowuje wszystkie rejestry które modyfikuje. W powyższym przykładzie, kod nie potrzebuje przechowywać ax. Chociaż PrintSpaces zmienia al., nie wpłynie to na działanie programu. Jeśli wywołujący przechowuje rejestry, nie musi zachowywać rejestrów nie troszcząc się o to:

```
            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
```

Przykład ten dostarcza trzech różnych przypadków. Pierwsza pętla (Loop0) zachowuje tylko rejestr cx. Modyfikacja rejestru al. Nie wpływa na działanie tego programu. Bezpośrednio po pierwszej pętli, kod ten znowu wywołuje PrintSpaces. Jednakże, ten kod nie zachowuje ax lub cx ponieważ nie troszczy się czy PrintSpaces je zmienia. Ponieważ pętla końcowa (Loop1) stosuje ax i cx, zachowuje je obie.

Jednym dużym problemem z zachowywaniem rejestrów przez wywołującego, jest to ,że nasz program może się zmieniać. Możemy zmodyfikować kod wywołujący lub procedurę, żeby stosowały dodatkowy rejestr. Takie zmiany, oczywiście, mogą zmieniać zbiór rejestrów, które musimy przechować. Gorzej jeszcze, jeśli modyfikowany jest sam podprogram, będziemy musieli zlokalizować każde wywołanie programu i zweryfikować, który podprogram nie zmienia żadnego rejestru stosując kod wywołujący.

Przechowywanie rejestrów nie jest tym co zachowanie otoczenia. Możemy również położyć i zdjąć zmienne i inne wartości które podprogram może zmienić. Ponieważ, 80x86 pozwala nam odłożyć i zdjąć komórki pamięci, możemy łatwo zachować również te wartości.

11.5 PARAMETRY

Chociaż jest dużo klas procedur które są całkowicie zamknięte, większość procedur wymaga jakichś danych wejściowych i zwraca jakieś dane do wywoływującego. Parametry są wartościami które możemy przekazać z i do procedury. Jest wiele aspektów parametrów. Pytania związane z parametrami:

*__gdzie__ jest dana przychodząca?
*__jak__ przekazujemy i zwracamy dane?
*__jaka__ jest ilość danych do przekazania?

Jest sześć głównych mechanizmów dla przekazywania danych do i z procedury, są to:

*przekazywanie przez wartość
*przekazywanie przez referencję
*przekazywanie przez wartość/powrót
*przekazywanie przez wynik, i
*przekazywanie przez nazwę
*przekazywanie przez leniwe wartościowanie

Możemy również martwić się gdzie mamy przekazać parametry. Popularne miejsca to:

*do rejestrów
*do globalnych komórek pamięci
*na stos
*do strumienia kodu lub
*do bloku parametrów odnosząc się przez wskaźnik

W końcu, ilość danych ma bezpośredni związek na to gdzie i jak się je przekazuje. Poniższa sekcja zajmuje się tymi tematami.

11.5.1 PRZEKAZYWANIE PRZEZ WARTOŚĆ

Parametr przekazywany przez wartość to znaczy – program wywołujący przekazuje wartość do procedury. Parametry przekazywane przez wartość są tylko wartościami wejściowymi. To znaczy, możemy przekazać je do procedury, ale procedura nie może ich zwrócić. W HLLach, takich jak Pascal, idea przekazywania parametru przez wartość tylko parametru wejściowego stanowi większy sens .Mamy daną procedurę call;

```
CallProc(I);
```

Jeśli przekażemy I przez wartość, CallProc nie zmieni wartości I, bez względu na to co będzie się działo z parametrem wewnątrz CallProc.

Ponieważ musimy przekazać kopię danej do procedury, powinniśmy tylko stosować tą metodę dla przekazywania małych obiektów, takich jak bajty, słowa, i podwójne słowa. Przekazywanie tablic i ciągów znaków jest bardzo nieefektywne (ponieważ musimy stworzyć i przekazać kopię struktury do procedury)

11.5.2 PRZEKAZYWANIE PRZEZ REFERENCJĘ

Przy przekazywaniu parametrów przez referencję, musimy przekazać adres zmiennej zamiast jej wartości. Innymi słowy, musimy przekazać wskaźnik do danej. Procedura musi odnieść się przez ten wskaźnik aby uzyskać dostęp do danej. Przekazywanie parametrów przez referencję jest użyteczne kiedy musimy zmodyfikować parametr aktualny lub kiedy przekazujemy duże struktury danych pomiędzy procedurami.

Przekazywanie parametrów przez referencję może tworzyć jakieś osobliwe wyniki. Poniższa procedura pascalowska dostarcza przykład jednego problemu jaki możemy napotkać:

```
program main (input, output) ;
var m.: integer;
```

```
            procedure bletch (var i, j: integer);
            begin
                        i := i+2;
                        j := j-1;
                        writeln (i, ' ', j);
            end;
            -
            -
            -
begin       {main}
                        m. := 5;
                        bletch (m., m.);
end.
```

Ta szczególna sekwencja kodu będzie drukowała „00" bez w względu na wartość m. Jest tak ponieważ parametry i i j są wskaźnikami do aktualnej danej i oba wskazują na ten sam obiekt. Dlatego też, instrukcja j := j+i; zawsze daje zero ponieważ i i j odnoszą się do tej samej zmiennej.

Przekazywanie przez referencję jest zazwyczaj mniej wydajne niż przekazywanie przez wartość. Musimy osiągnąć wartość obiektu dla wszystkich parametrów przekazywanych przez referencję przy każdym dostępie; jest to wolniejsze niż proste zastosowanie wartości. Jednakże, kiedy przekazujemy duże struktury danych,, przekazywanie przez referencje jest szybsze ponieważ nie robimy kopii tej struktury przed wywołaniem procedury.

11.5.3 PRZEKAZYWANIE PRZEZ WARTOŚĆ / POWRÓT

Przekazywanie przez wartość / powrót (znane również jako wartość – wynik) jest kombinacją cech przekazywania przez wartość i przekazywania przez referencję. Przekazujemy parametry przez wartość/ powrót poprzez adres, podobnie jak przy przekazywaniu parametrów przez referencję. Jednakże, na wejściu, procedura czyni czasową kopię tego parametru i stosuje kopię podczas wykonywania procedury. Kiedy procedura się kończy, kopiuje kopię czasową do oryginalnego parametru.

Pascalowski kod przedstawiony w poprzedniej sekcji działałby właściwie z przekazywaniem parametrów przez wartość/ powrót. Oczywiście, kiedy Bletch wraca do kodu wywołującego, m może zwierać tylko jedną z dwóch wartości, ale kiedy Bletch jest wykonywana, i i j mogą zawierać różne wartości.

W pewnych przypadkach przekazywanie przez wartość/ powrót jest bardziej wydajne niż przekazywanie przez referencję, w innych mniej wydajne. Jeśli procedura odwołuje się do parametrów parę razy, kopiowanie danych parametrów jest kosztowne. Z drugiej strony, jeśli procedura stosuje ten parametr często, procedura amortyzuje stały koszt kopiowania danych poprzez niekosztowny dostęp do lokalnej kopii.

11.5.4 PRZEKAZYWANIE PRZEZ WYNIK

Przekazywanie przez wynik jest prawie identyczne do przekazywania przez wartość – powrót. Przekazujemy wskaźnik do żądanego obiektu a procedura stosuje lokalną kopię zmiennej a potem przechowuje wynik we wskaźniku kiedy wraca. Jedyna różnica pomiędzy przekazywaniem przez wartość – powrót a przekazywaniem przez wynik polega na tym, że kiedy przekazujemy parametry przez wynik nie kopiujemy danych na wejściu do procedury. Parametry przekazywanie przez wynik są wartościami zwracanymi, nie danymi przekazywanymi do procedury. Dlatego też, przekazywanie przez wynik jest odrobinę bardziej wydajne niż przekazywanie przez wartość – powrót ponieważ oszczędzamy na kopiowaniu danych do lokalnej zmiennej.

11.5.5 PRZEKAZYWANIE PRZEZ NAZWĘ

Przekazywanie przez nazwę jest mechanizmem przekazywania parametrów stosowanym przez makra , przyrównywania tekstu i macro #define w języku C. Ten mechanizm przekazywania parametrów stosuje zastępowanie tekstowe parametrów. Rozważmy poniższe makro MASMa:

```
PassByName          macro   Param1, param2
                    mov     ax, param1
                    add     ax, Param2
                    endm
```

Jeśli mamy wywołanie makra w postaci:

```
            PassByName      bx, I
```

MASM wyemituje poniższy kod, zastępując bx za param1 i I za Param2:

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

Niektóre języki wysokiego poziomu takie jak ALGOL-68 i Panacea, wspierają przekazywane parametrów przez nazwę. Jednakże, implementacja przekazywania przez nazwę stosując zastępowanie tekstowe w językach kompilowanych (jak ALGOL-68) jest bardzo trudne i niewydajne. W zasadzie, musimy rekompilować funkcje za każdym razem kiedy ją wywołujemy. Wiec

języki kompilowane które wpierają przekazywanie przez nazwę generalnie stosują różne techniki dla przekazywania tych parametrów. Rozważmy poniższą procedurę Panacei:

```
PassByName: procedure(name item; integer ; var  index: integer);
begin PassByName;
        foreach  index in 0..10 do
                item := 0;
        endfor;
end PassByName;
```

Zakładając, że wywołujemy ten podprogram instrukcją PassByName(A[i], i); gdzie A jest tablicą liczb całkowitych mającą (przynajmniej) elementy A[0]..A[10].Po zastąpienia przekazywania przez nazwę parametru item dostalibyśmy poniższy kod:

```
begin PassByName;
        foreach index in 0..10 do
                A[I] := 0 (*Zauważmy, że index i  I są aliasami*)
        endfor;
end PassByName;
```

Kod ten zeruje elementy 0..10 tablicy A

Języki wysokiego poziomu jak ALGOL-68 i Panacea kompilują przekazywanie parametrów przez nazwę do funkcji, która zwraca adres danego parametru. Pod tym względem przekazywanie parametrów przez nazwę jest podobne do przekazywania parametrów przez referencję ponieważ przekazujemy adres obiektu. Główna różnica jest taka ,że przy przekazywaniu parametrów przez referencję obliczamy adres obiektu przed wywołaniem podprogramu; przy przekazywaniu przez nazwę podprogram sam wywołuje jakąś funkcję obliczając adres parametru.

Więc co różni te wykonania? Cóż, spójrzmy ponownie na powyższy kod. Mamy przekazane A[I] przez referencję zamiast przez nazwę, kod wywołujący obliczył adres A[I] tuż przed wywołaniem i przekazał tym samym ten adres. Wewnątrz procedury PassByName zmienna item odnosiła się zawsze do pojedynczego adresu, a nie do adresu ,który zmienił się wraz z I. Przy przekazywaniu przez nazwę item jest rzeczywiście funkcją, która oblicza adres parametru, do którego procedura przekazuje wartość zero. Funkcja taka może wyglądać jak poniższa:

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

Skompilowany kod wewnątrz procedury PassByName może wyglądać czasami jak poniższy:

```
; item := 0;
              call     ItemThunk
              mov      word ptr [bx], 0
```

Thunk jest historycznym terminem dal tej funkcji, która oblicza adres parametru przekazywanego przez nazwę. Nie jest nic warte, że większość HLLi wspiera przekazywanie parametrów przez nazwę nie wywołując bezpośrednio thunks'ów (jak powyższy call). Ogólnie, program wywołujący przekazuje adres thunk a podprogram wywołuje thunk pośrednio . Pozwala to tej samej sekwencji instrukcji wywoływać kilka różnych thunks'ów (odpowiadającym różnym wywołaniom podprogramu)

11.5.6 PRZEKAZYWANIE PRZEZ LENIWE WARTOŚCIOWANIE

Przekazywanie przez nazwę jest podobne do przekazywania przez referencję ponieważ procedura uzyskuje dostęp do parametru stosując adres parametru. Pierwszą różnicą pomiędzy nimi dwom jest to , że program wywołujący przekazuje bezpośrednio adres na stos kiedy przekazujemy przez referencję, przekazuje adres funkcji, która oblicza adres parametru kiedy przekazujemy parametry przez nazwę. Mechanizm przekazywania parametrów przez leniwe wartościowanie dzieli te same związki z przekazywaniem parametrów przez wartość – program wywołujący przekazuje adres funkcji która oblicza wartość parametru jeśli pierwszym dostępem do tego parametru jest operacja odczytu.

Przekazywanie przez leniwe wartościowanie jest użyteczną techniką przekazywania parametrów jeśli koszt obliczenia wartości parametru jest bardzo wysoki a procedura może nie używać wartości. Rozważmy poniższą nagłówkową procedurę Panacei:

```
PassByEval1: procedure (eval a:integer; eval b:integer; eval c:integer);
```

Kiedy wywołujemy funkcję PassByEval nie oblicza aktualnego parametru i przekazuje swoje wartości do procedury,. Zamiast tego kompilator generuje thunksy, które będą obliczały wartość parametru przynajmniej raz. Jeśli pierwszym dostępem do parametru eval jest odczyt, thunk obliczy wartość parametru i przechowa ją w lokalnej zmiennej. Ustawi również flagi żeby wszystkie przyszłe dostępy nie wywoływały thunka (ponieważ już jest obliczona wartość parametru). Jeśli pierwszy dostęp do parametru eval to zapis, wtedy kod ustawia flagi a przyszły dostęp wewnątrz tej samej aktywowanej procedury zastosuje zapisaną wartość i zignorowanie thunka.

Rozważmy powyższą procedurę PassByEval. Przypuśćmy, że zabiera kilka minut obliczenie wartości dla parametrów a ,b i c (które mogą być na przykład trzema możliwymi różnymi ścieżkami w grze Szachy). Być może procedura PassByEval zastosuje tylko wartość jednego z tych parametrów. Bez przekazanie przez leniwie wartościowanie, kod wywołujący musiałby tracić czas na obliczenie wszystkich trzech parametrów, nawet jeśli procedura stosować będzie tylko jedną wartość. Z przekazaniem przez leniwe wartościowanie jednakże, procedura spędzi czas na obliczeniu jednego potrzebnego parametru. Leniwe wartościowanie jest popularną techniką sztucznej inteligencji (AI) i systemów operacyjnych używających poprawy wydajności.

11.5.7 PRZEKAZYWANIE PARAMETRÓW DO REJESTRACH

Mając już poruszony temat jak przekazywać parametry do procedury, następną rzeczą do omówienia jest to gdzie przekazywać parametry. Gdzie są przekazywane parametry zależy w dużej mierze do rozmiaru i liczby tych parametrów. Jeśli przekazujemy małą liczbę bajtów do procedury, wtedy rejestry są doskonałym miejscem do przekazania parametrów. Rejestry są idealnym miejscem do przekazania wartości parametrów do procedury. Jeśli przekazujemy pojedynczy parametr do procedury powinniśmy użyć poniższych rejestrów dla stowarzyszonych typów danych:

Rozmiar Danej	Przekazanie do Rejestru
Bajt:	al
Słowo:	ax
Podwójne Słowo:	dx: ax lub eax (jeśli 80386 lub lepszy)

Taka jest zasada. Jeśli dogodniej będzie przekazać 16 bitowa wartość do rejestru si lub bx, jak najbardziej zróbmy to. Jednak większość programistów stosuje powyższe rejestry do przekazania parametrów.

Jeśli przekazujemy kilka parametrów do procedury do rejestrów 80x86, powinniśmy zastosować rejestry w następującym porządku:

Pierwszy Ostatni

ax, dx, si, di, bx, cx

Generalnie powinniśmy unikać stosowania rejestru bp. Jeśli potrzebujemy więcej niż sześć słów, być może powinniśmy przekazać nasze wartości gdzie indziej

Pakiet Standardowej Biblioteki UCR dostarcza kilku dobrych przykładów procedur, które przekazują parametry przez wartość do rejestrów. Putc, która wysyła kod znaku ASCII na wyświetlacz, spodziewa się wartości ASCII w rejestrze al. Podobnie jak puti spodziewa się wartości liczby całkowitej ze znakiem w rejestrze ax. Jako inny przykład, rozważmy poniższy podprogram putsi , który wysyła wartość do al jako liczbę całkowitą ze znakiem:

```
putsi       proc
            push    ax              ;przechowuje wartość AH
            cbw                     ;rozszerzenie znaku AL -> AX
            puti
            pop     ax              ;przywrócenie AH
            ret
putsi       endp
```

Pozostałe cztery mechanizmy przekazywania parametrów (przekazywanie przez referencję, wartość/powrót, wynik i nazwę) generalnie wymagają aby przekazać wskaźnik do żądanego obiektu (lub do thunka w przypadku przekazywania przez nazwę). Kiedy przekazujemy takie parametry do rejestrów, musimy rozważyć czy przekazujemy offset czy pełny adres segmentowy. Szesnastobitowy offset może być przekazany do każdego 16 bitowego rejestru ogólnego przeznaczenia 80x86. Si, di i bx są najlepszymi miejscami do przekazania offsetu ponieważ będziemy musieli prawdopodobnie załadować go i tak do jednego z tych rejestrów. Możemy przekazać 32 bitowy adres segmentowy do dx:ax podobnie inne parametry podwójnego słowa. Jednakże możemy również przekazać je do ds.:bx, ds.:si, ds.:di, es:bx, es:si lub es:di i zastosować je bez kopiowania do rejestru segmentowego.

Podprogram puts z UCR Stdlib, który drukuje ciąg znaków na wyświetlaczu jest dobrym przykładem podprogramu który stosuje przekazywanie przez referencję. Chce adres ciągu w parze rejestrów es:di. Przekazuje parametry w ten sposób nie dlatego, że modyfikuje parametry ale dlatego, że ciągi są dosyć długie i przekazywanie ich w jakiś inny sposób byłoby nieefektywne. Jako inny przykład rozważmy strfill(str,c) który kopiuje znak c (przekazywany przez wartość do al) na każdą pozycję znaku w str (przekazywaną przez referencję do es:di) aż do zerowego bajtu kończącego:

```
; strfill -             kopiuje wartość do al ciągu wskazywanego przez es:di
```

```
;                       aż do zerowego bajtu kończącego.

byp                     textequ         <byte ptr>

strfill                 proc
                        pushf
                        cld
                        push    di
                        jmp     sfStart
sfLoop:                 stosb                           ;es:[di]  := al, di := di+1;
sfStart:                cmp     byp es:[di], 0
                        jne     sfLoop

                        pop     di
                        popf
                        ret
strfill                 endp
```

Kiedy przekazujemy parametry przez wartość/powrót lub wynik do podprogramu, możemy przekazać adres do rejestru. Wewnątrz procedury skopiujemy wartość wskazywaną przez ten rejestr do zmiennej lokalnej (tylko wartość/powrót). Dopiero przed powrotem do programu wywołującego, możemy przekazać wynik końcowy z powrotem jako adres do rejestru.

Poniższy kod wymaga dwóch parametrów. Pierwszy jest przekazywany przez wartość/powrót a podprogram oczekuje adresu parametru aktualnego w rejestrze bx. Drugi jest przekazywany przez wynik, którego adres jest w si. Ten podprogram zwiększa parametr przekazywany przez wartość/powrót i przechowuje poprzedni wynik w parametrze przekazywanym przez wynik:

```
;CopyAndInc-            BX zawiera adres zmiennej. Podprogram ten kopiuje tą zmienną do lokacji wyszczególnionej
;                       w SI a potem zwiększa zmienną BX. Notka: AX  i CX przechowują lokalne kopie tych
;                       parametrów podczas wykonywania
CopyAndInc              proc
                        push    ax              ;przechowanie AX
                        push    cx              ;przechowanie CX
                        mov     ax, [bx]        ;pobranie lokalnej kopii pierwszego parametru
                        mov     cx, ax          ;przechowanie w zmiennej lokalnej drugiego parametru
                        inc     ax              ;zwiększenie pierwszego parametru
                        mov     [si], cx        ;przechowanie parametru przekazanego przez wart /powrót
                        pop     cx              ;przywrócenie wartości CX
                        pop     ax              ;przywrócenie wartości AX
                        ret
CopyAndInc              endp
```

Czyniąc wywołanie CopyAndInc (I,J) zastosujemy kod podobny do tego:

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

Jest to oczywiście przykład banalny, którego implementacja jest bardzo niewydajna. Niemniej jednak, pokazuje jak są przekazywane parametry przez wartość/powrót i wynik w rejestrach 80x86. Jeśli chętnie wymienimy trochę przestrzeni za szybkość, jest inny sposób osiągnięcia tego samego rezultatu jak przekazanie przez wartość/powrót lub wynik kiedy przekazujemy parametry do rejestrów. Rozważmy poniższą implementację CopyAndInc:

```
CopyAndInc              proc
                        mov     cx, ax          ;robi kopię pierwszego parametru,
                        inc     ax              ;potem zwiększa go o jeden.
                        ret
CopyAndInc              endp
```

Czyniąc wywołanie CopyAndInc(I, J),jak poprzednio, zastosujemy poniższy kod 80x86:

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

Zauważmy, że kod ten nie przekazuje wcale żadnego adresu; a mimo to ma taką samą semantykę (to znaczy wykonuje te same operacje) jak wersja poprzednia. Obie wersje zwiększają I i przechowują pre-inkrementowaną wersję w J. Wyraźnie ostatnia

wersja jest szybsza, chociaż nasz program będzie odrobinę większy jeśli będzie wiele wywołań CopyAndInc w programie (sześć lub więcej).

Możemy przekazać parametry przez nazwę lub leniwe wartościowanie do rejestru poprzez proste załadowanie tego rejestru adresem thunka wywołującego. Rozważmy procedurę Panacei PassByName. Jedna z implementacji tej procedury może wyglądać następująco

```
;PassByName-      Expects a pass by reference parameter index
;                 passed in si and a pass by name parameter, item,
;                 passed in dx (the thunk returns the address in bx).

PassByName        proc
                  push    ax                    ;Preserve AX across call
                  mov     word ptr [si], 0      ;Index := 0;
ForLoop:          cmp     word ptr [si], 10     ;For loop ends at ten.
                  jg      ForDone
                  call    dx                    ;Call thunk item.
                  mov     word ptr [bx], 0      ;Store zero into item.
                  inc     word ptr [si]         ;Index := Index + 1;
                  jmp     ForLoop

ForDone:          pop     ax                    ;Restore AX.
                  ret                           ;All Done!
PassByName        endp
```

Możemy wywołać ten podprogram kodem, który wygląda następująco:

```
                  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
```

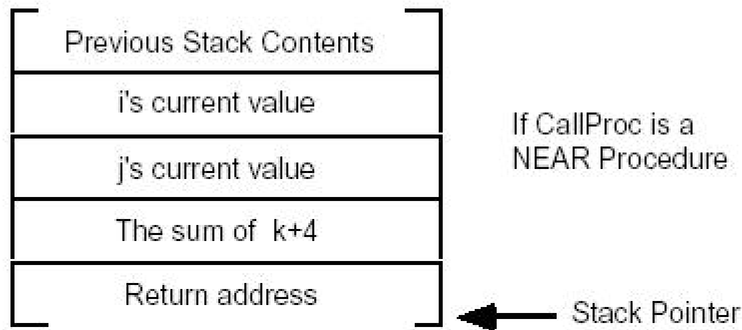
Korzyścią z tego schematu jest to,że możemy wywoływać różne thunksy , nie tylko podprogram ItemThunk pojawiający się we wcześniejszym przykładzie

---

11.5.8 PRZEKAZYWANIE PARAMETRÓW DO ZMIENNYCH GLOBALNYCH.

Kiedy już skończyliśmy z rejestrami, inną jedyną (rozsądną) alternatywą jaką mamy jest pamięć główna. Jednym z łatwiejszych miejsc do przekazania parametrów jest zmienna globalna w segmencie danych. Poniższy kod dostarcza przykładu:

```
                  mov     ax, xxxx                  ;przekazanie tego parametru przez wartość
                  mov     Value1Proc1, ax
                  mov     ax, offset yyyy           ;przekazanie tego parametru przez referencję
                  mov     word ptr Ref1Proc1, ax
                  mov     ax, seg  yyyy
                  mov     word ptr Ref1Proc1+2, ax
                  call    ThisProc
                  -
                  -
                  -
```



Rysunek 11.5 Ułożenie na stosie CallProc dla bliskiej procedury

```
ThisProc        proc    near
                push    es
                push    ax
                push    bx
                les     bx, Ref1Proc1           ;pobranie adresu parametru  referencyjnego
                mov     ax, Value1Proc1         ;pobranie parametru przez wartość
                mov     es:[bx], ax             ;przechowanie w lokacji wskazywanej przez parametr referencyjny
                pop     bx
                pop     ax
                pop     es
                ret
ThisProc        endp
```

Przekazywanie parametrów przez globalne lokacje jest mało elegancji i nie wydolne. Co więcej, jeśli stosujemy zmienne globalne w ten sposób do przekazywania parametrów, podprogramy, które piszemy nie mogą stosować rekurencji. Na szczęście jest lepszy schemat przekazywania parametrów dla przekazywania danych w pamięci wiec nie musimy poważnie rozważać tego schematu.

11.5.9 PRZEKAZYWANIE PARAMETRÓW NA STOS

Wiele HLLi używa stosu do przekazywania parametrów ponieważ metoda ta jest dosyć wydajna. Przekazując parametry na stos, odkładamy je bezpośrednio przed wywołaniem podprogramu. Wtedy podprogram odczytuje je ze stosu pamięci i działa na nich odpowiednio. Rozważmy poniższą pascalowską procedurę call:

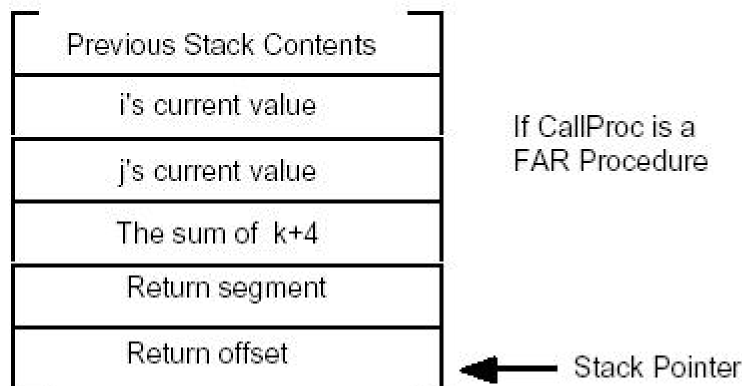
```
                CallProc(i, j,k+4);
```

Większość kompilatorów Pascala odkłada swoje parametry na stos, w kolejności w jakiej się pojawiają na liście parametrów. Dlatego też, kod 80x86 typowo emituje dla tego podprogramu wywołania (zakładając, ze mamy przekazywanie parametrów przez wartość):

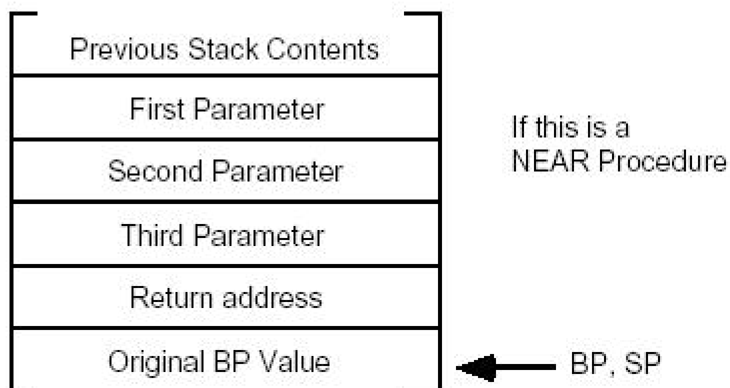
```
                push    i
                push    j
                mov     ax, k
                add     ax, 4
                push    ax
                call    CallProc
```

Na wejściu do CallProc, stos 80x86 wygląda jak ten pokazany na rysunku 11.5 (dla bliskiej procedury) lub Rysunku 11.6 (dla dalekiej procedury).

Możemy zyskać na dostępie do parametrów przekazywanych na stos przez usunięcie danych ze stosu (Zakładając wywołanie bliskiej procedury):



Rysunek 11.6 : Położenie CallProc na stosie dla Dalekiej procedury



Rysunek 11.7 : Dostęp do parametrów na stosie

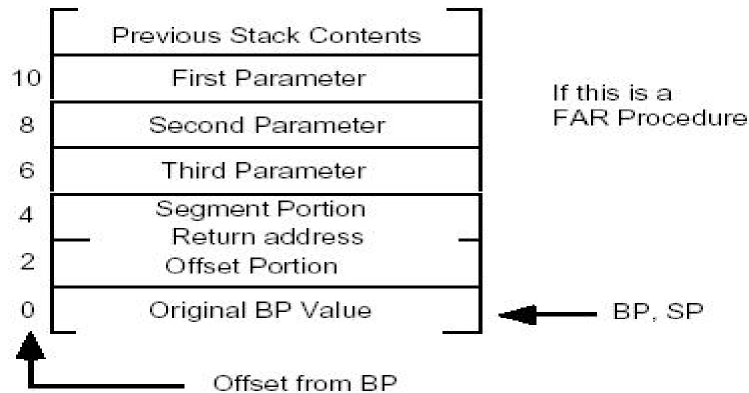
```
CallProc    proc    near
            pop     RtnAdrs
            pop     kParam
            pop     jparam
            pop     iParam
            push    RtnAdrs
            -
            -
            -
            ret
CallProc    end
```

Jednakże jest lepszy sposób. Architektura 80x86 pozwala nam na stosowanie rejestru bp (wskaźnik bazowy) przy dostępie do parametrów przekazywanych na stos. Jest to jeden z powodów dla którego tryby adresowania disp[bp], [bp][di], [bp][si], disp[bp][si] i disp[bp][di] używają segment stosu zamiast segmentu danych. Poniższy fragment kodu daje nam kod standardowego wejścia i wyjścia procedury:

```
StdProc     proc    near
            push    bp
            mov     bp, sp
            -
            -
            -
            pop     bp
            ret     ParamSize
StdProc     endp
```

ParamSize jest liczbą bajtów parametrów odkładanych na stos przed wywołaniem procedury. W procedurze CallProc było sześć bajtów parametrów odkładanych na stos więc ParamSize będzie maił sześć.

Spójrzmy na stos bezpośrednio po wykonaniu mov bp, sp w StdProc. Zakładając, że odłożyliśmy trzy parametry słowa na stos, powinien on wyglądać jak na rysunku 11.7



Rysunek 11.8 Dostęp do parametrów na stosie w dalekiej procedurze

Teraz parametry mogą być pobierane poprzez indeksowanie rejestru bp:

```
mov     ax, 8[bp]       ;dostęp do pierwszego parametru
mov     ax, 6[bp]       ;dostęp do drugiego parametru
mov     ax, 4[bp]       ;dostęp do trzeciego parametru
```

Kiedy wracamy do kodu wywołującego, procedura musi usunąć te parametry ze stosu. Dla osiągnięcia tego zdejmujemy starą wartość bp ze stosu i wykonujemy instrukcję ret6. To zdejmuje adres powrotny ze stosu i dodaje sześć do wskaźnika stosu, skutecznie usuwając parametry ze stosu.

Przesunięcia dane powyżej są tylko dla bliskiej procedury. Kiedy wywołujemy daleką procedurę.

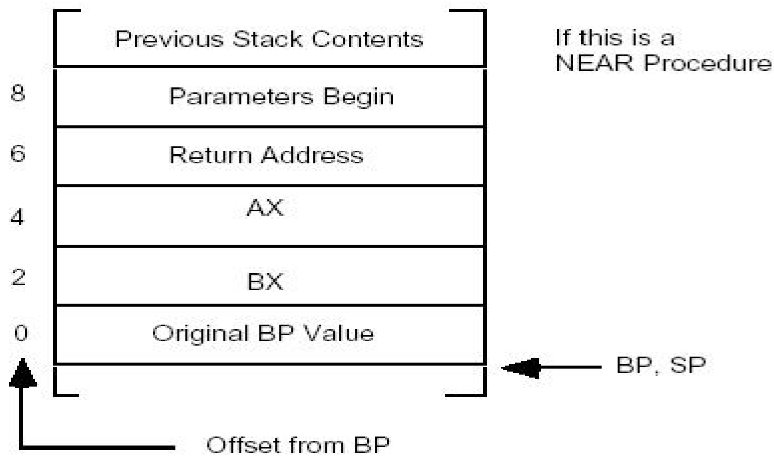
*0[BP] będzie wskazywał na starą wartość BP
*2[BP] będzie wskazywał na przesunięcie części adresu powrotu
*4[BP] będzie wskazywał na segment części adresu powrotu a
*6[BP] będzie wskazywał ostatni parametr na stosie

Zawartość stosu kiedy wywołujemy daleką procedurę jest pokazana na rysunku 11.8
Zbiór parametrów, adres powrotu, rejestry zachowane na stosie i inne pozycje to tzw. Ramka stosu lub rekord aktywacji.

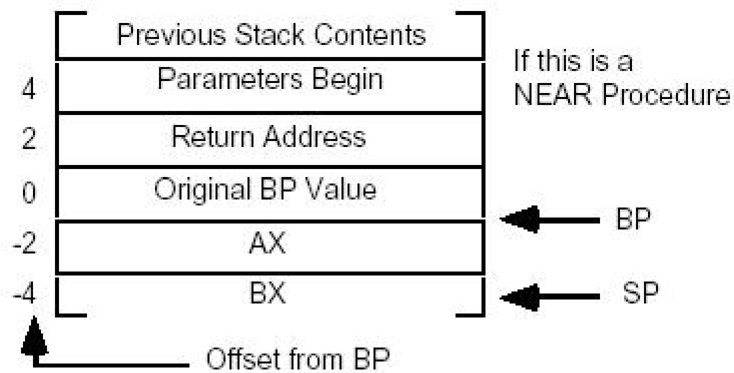
Kiedy zachowujemy inne rejestry na stosie, zawsze upewniajmy się, że zachowaliśmy i ustawili bp przed odłożeniem innych rejestrów na stos. Jeśli odłożymy inne rejestry na stos przed ustawieniem bp, offset ramki stosu zmieni się. Na przykład, poniższy kod zakłóca porządek prezentowany powyżej:

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

Ponieważ kod ten odkłada ax i bx przed odłożeniem bp i skopiowaniem sp do bp, ax i bx pojawią się w rekordzie aktywacji przed adresem powrotu (który zastartowałby normalnie z pod lokacji]bp+2]). Jako wynik, wartość bp pojawi się w lokacji [bp+2] a wartość ax pojawi się w lokacji [bp+4]. Takie odłożenie adresu powrotu i innych parametrów na stosie jest pokazane na rysunku 11.9



Rysunek 11.9: Popsucie offsetu przez odłożenie na stos innych rejestrów przed BP



Rysunek 11.10: Zachowanie stałego offsetu przez odłożenie na stos pierwszego BP

Chociaż jest to bliska procedura, parametry nie zaczynają się od offsetu osiem w rekordzie aktywacji. Po odłożeniu rejestrów ax i bx po ustawieniu bp, offset parametrów będzie wynosił cztery (zobacz rysunek 11.10)

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

Dlatego też, instrukcje push bp i mov bp, sp powinny być pierwszymi instrukcjami każdego wykonywanego podprogramu, kiedy ma swoje parametry na stosie.

Dostęp do parametrów stosując wyrażenie takie jak [bp+6] może uczynić nasz program bardzo trudnym do czytania i pielęgnacji. Jeśli chcielibyśmy użyć sensownych nazw, jest kilka sposobów zrobienia tego. Jeden sposób do odniesienia się do parametrów przez nazwę jest zastosowanie przyrównań. Rozważmy poniższą Pascalową procedurę i jej odpowiednik w kodzie języka asemblera 80x86:

```
procedure xyz (var i: integer; j:integer);
begin
```

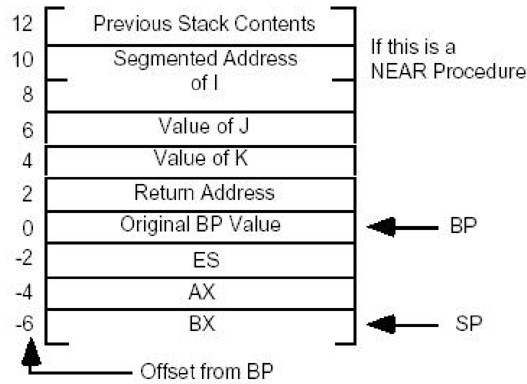
```
                i :=j + k;
        end;
```
Sekwencja wywołania:
```
        xyz(a, 3, 4);
```
Kod języka asemblera:

```
xyz_i       equ     8[bp]           ;Stosujemy przyrównanie więc możemy odnieść się do nazwy
xyz_j       equ     6[bp]           ;symbolicznej w treści procedury
xyz_k       equ     4[bp]
xyz         proc    near
            push    bp
            mov     bp, sp
            push    es
            push    ax
            push    bx
            les     bx, xyz_i       ;pobranie adresu I do ES:BX
            mov     ax, xyz_j       ;pobranie parametru J
            add     ax, xyz_k       ;dodanie parametru K
            mov     es:[bx], ax     ;przechowanie wyniku w parametrze I
            pop     bx
            pop     ax
            pop     es
            pop     bp
            ret     8
xyz         endp
```

Sekwencja wywołania:
```
            mov     ax, seg a       ;parametr ten jest przekazany przez referencję, więc przekazujemy
            push    ax              ; jego adres na stos
            mov     ax, offset a
            push    ax
            mov     ax, 3           ;to jest drugi parametr
            push    ax
            mov     ax, 4           ;to jest trzeci parametr
            push    ax
            call    xyz
```
Na procesorach 80186 i późniejszych możemy zastosować poniższy kod w miejsce powyższego:
```
            push    seg a
            push    offset  a
            push    3
            push    4
            call    xyz
```
Na wejściu do procedury xyz, przed wykonaniem instrukcji les, stos wygląda jak pokazano na rysunku 11.11

Ponieważ przekazujemy I przez referencję, musimy odłożyć jego adres na stos. Ten kod przekazuje parametr referencyjny stosując 32 bitowy adres segmentowy. Zauważmy, że kod ten używa ret 8. Chociaż są trzy parametry na stosie, referencyjny parametr I używa czterech bajtów ponieważ jest to daleki adres. Dlatego też jest osiem bajtów parametrów na stosie wymaganych przez instrukcję ret8

Gdybyśmy przekazali I przez referencję stosując bliski wskaźnik zamiast dalekiego wskaźnika, kod mógłby wyglądać tak:
```
xyz_i       equ     8[bp]           ;stosujemy przyrównanie więc możemy odnieść się do nazwy
xyz_j       equ     6[bp]           ;symbolicznej w treści procedury
xyz_k       equ     4[bp]
xyz         proc    near
            push    bp
            mov     bp, sp
            push    ax
            push    bx
            mov     bx, xyz_i       ;pobranie adresu I do BX
```



Rysunek 11.11: Stos na wejściu do procedury XYZ

```
            mov     ax, xyz_j               ;pobranie parametru J
            add     ax, xyz_k               ;dodanie parametru K
            mov     [bx], ax                ;przechowanie wyniku w parametrze I
            pop     bx
            pop     ax
            pop     bp
            ret     6
xyz         endp
```

Zauważmy, że ponieważ adres I na stosie jest tylko dwu bajtowy (zamiast cztero), ten podprogram zdejmuje tylko sześć bajtów kiedy wraca.

Sekwencja wywołania:

```
            mov     ax, offset a                    ;przekazanie bliskiego adresu a
            push    ax
            mov     ax, 3                           ;to jest drugi parametr
            push    ax
            mov     ax, 4                           ;to jest trzeci parametr
            push    ax
            call    xyz
```

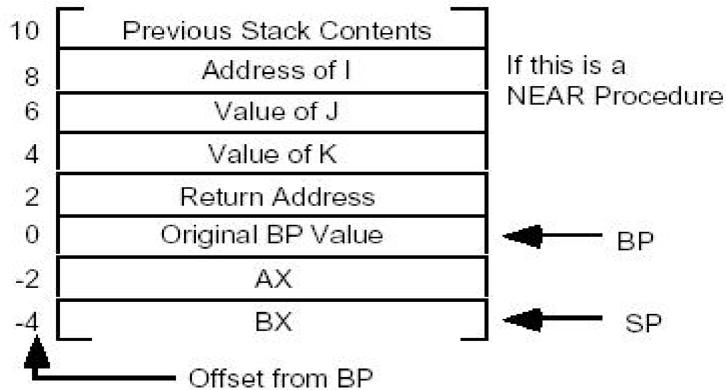
Na procesorze 80286 i późniejszych możemy zastosować poniższy kod w miejsce powyższego:

```
            push    offset a                        ;przekazanie bliskiego adresu a
            push    3                               ;przekazanie drugiego parametru przez wartość
            push    4                               ;przekazanie trzeciego parametru przez wartość
            call    xyz
```

ramkę stosu powyższego kodu pokazano na Rysunku 11.12

Kiedy przekazujemy parametry przez wartość/powrót lub wynik ,przekazujemy adres do procedury, dokładnie jak przekazując parametry przez referencję. Jedyna różnica jest taka, że stosujemy lokalną kopię zmiennej wewnątrz procedury zamiast pośredniego dostępu do zmiennej przez wskaźnik. Poniższa implementacja dla xyz pokazuje jak przekazać I przez wartość-powrót i przez wynik:

```
; wersja xyz stosująca przekazanie przez wartość –powrót dla xyz_i
xyz_i       equ     8[bp]                   ;stosujemy przyrównanie więc możemy odnieść się do nazw
xyz_j       equ     6[bp]                   ;symbolicznych w treści procedury
xyz_k       equ     4[bp]

xyz         proc    near
            push    bp
            mov     bp, sp
            push    ax
            push    bx
```



Rysunek 11.12 :Przekazanie parametrów przez referencję stosując bliski wskaźnik zamiast wskaźnika dalekiego

```
                push    cx                      ;Tu lokalna kopia
                mov     bx, xyz_i               ;pobranie adresu I do BX
                mov     cx, [bx]                ;pobranie lokalnej kopii parametru I
                mov     ax, xyz_j               ;pobranie parametru J
                add     ax, xyz_k               ;dodanie parametru K
                mov     cx, ax                  ;przechowanie wyniku w lokalnej kopii
                mov     bx, xyz_i               ;pobranie wskaźnika do I, znowu
                mov     [bx], cx                ;przechowanie wyniku
                pop     cx
                pop     bx
                pop     ax
                pop     bp
                ret     6
xyz             endp
```

Jest parę niekoniecznych instrukcji mov w tym kodzie. Są one obecne tylko dla dokładnej implementacji przekazywania parametrów przez wartość-powrót. Łatwo jest poprawić ten kod stosując przekazanie parametrów przez wynik. Kod zmodyfikowany:

```
;wersja xyz stosująca przekazywanie przez wynik dla xyz_i

xyz_i           equ     8[bp]           ;stosujemy przyrównanie więc możemy odnieść się do nazwy
xyz_j           equ     6[bp]           ;symbolicznej w treści procedury
xyz_k           equ     4[bp]

xyz             proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                push    cx              ;zachowanie lokalnej kopii
                mov     ax, xyz_i       ;pobranie parametru J
                add     ax,  xyz_k      ;dodanie parametru K
                mov     cx, ax          ;przechowanie wyniku w kopii lokalnej
                mov     bx, xyz_i       ;pobranie wskaźnika do I, znowu
                mov     [bx], cx        ;przechowanie wyniku
                pop     cx
                pop     bx
                pop     ax
                pop     bp
                ret     6
xyz             endp
```

Z przekazywaniem parametrów do rejestrów przez wartość-powrót i wynik, możemy poprawić wydajność tego kodu stosując zmodyfikowaną postać przekazania przez wartość. Rozważmy poniższą implementację xyz:
;wersja xyz stosująca zmodyfikowane przekazywanie przez wartość-powrót dla xyz_i

```
xyz_i           equ     8[bp]           ;stosujemy przyrównanie więc możemy odnieść się
xyz_j           equ     6[bp]           ;do nazwy symbolicznej w treści procedury
xyz_k           equ     4[bp]

xyz             proc    near
                push    bp
                mov     bp, sp
                push    ax
                mov     ax, xyz_j       ;pobrane parametru J
                add     ax, xyz_k       ;dodanie parametru K
                mov     xyz_i, ax       ;przechowanie wyniku w lokalnej kopii
                pop     ax
                pop     bp
                ret     4               ;zauważmy, że nie zdejmujemy parametru I
xyz             endp
```

Sekwencja wywołania dla tego kodu;

```
                push    a       ;przekazanie a jako wartości do xyz
                push    3       ;przekazanie drugiego parametru przez wartość
                push    4       ;przekazanie trzeciego parametru przez wartość
                call    xyz
                pop     a
```

Zauważmy, że wersja z przekazaniem przez wynik nie byłaby praktyczna ponieważ musimy odłożyć coś na stos aby uczynić miejsce dla lokalnej kopii I wewnątrz xyz. Możemy również odłożyć wartość a na wejściu pomimo, że procedura xyz ignoruje ją. Procedura ta zdejmuje tylko cztery bajty ze stosu na wyjściu. Pozostawia wartość parametru I na stosie, żeby kod wywołujący mógł przechować ją dalej dla właściwego miejsca przeznaczenia.

Przekazując parametr przez nazwę na stos, po prostu odkładamy adres thunka. Rozważmy poniższy pseudo – pascalowy kod:

```
Procedure swap (name Item1, Item2: integer);
var     temp: integer;
begin
        temp := Item1;
        Item1  := Item2;
        Item2 := temp;
End;
```

Jeśli swap jest bliską procedurą, kod 80x86 dal tej procedury może wyglądać jak poniższy (zauważmy, że ten kod został zoptymalizowany odrobinę i nie następuje dokładna sekwencja jak powyżej):

```
;swap-          zamienia dwa parametry przekazywane przez nazwę na stos Item1 jest przekazywany pod adres
;               [bp+6], Item2 jest przekazywany pod adres [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                      ;zachowanie wartości temp
                push     bx                     ;zachowanie bx
                call    wp swap_Item1           ;pobranie adresu Item1
                mov     ax, [bx]                ;zachowanie w temp (AX)
                call    wp swap_Item2           ;pobranie adresu Item2
                xchg    ax, [bx]                ;zamiana temp <-> Item1
                call    wp swap_Item1           ;pobranie adresu Item11
                mov     [bx], ax                ;zachowanie temp w Item1
```

```
                pop     bx                      ;przywrócenie bx
                pop     ax                      ;przywrócenie ax
                ret     4
swap            endp
```

Kilka przykładów wywołań swap:

```
;swap (A[i], i) --  wersja 8086
                lea     ax, thunk1
                push    ax
                lea     ax, thunk2
                push    ax
                call    swap
;swap (A[i],i)  -- wersja 80186 i wersje późniejsze
                push    offset thunk1
                push    offset thunk2
                call    swap
                -
                -
                -

;Notka: kod ten zakłada, że A jest dwubajtową tablicą liczb całkowitych

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

thunk2      proc        near
            lea         bx, 1
            ret
thunk2      endp
```

Powyższy kod zakłada, że thunksy są bliskimi procedurami które tkwią w tym samym segmencie co podprogram swap. Jeśli thunksy są dalekimi procedurami, program wywołujący musi przekazać daleki adres na stos a podprogram swap musi manipulować dalekim adresem. Demonstruje to poniższa implementacja swap, thunk1 i thunk2

```
;swap-                  zamienia dwa parametry przekazane przez nazwę na stos. Item1 jest przekazany pod
;                       adres [bp+10], Item2 przekazany pod adres [bp+6]

swap_Item1      equu    [bp+10]
swap_Item2      equ     [bp+6]
dp              textequ <dword ptr>
swap            proc    far
                push    bp
                mov     bp, sp
                push    ax                      ;zachowanie wartości temp
                push    bx                      ;zachowanie bx
                push    es                      ;zachowanie es
                call    dp  swap_Item1          ;pobranie adresu Item1
                mov     ax, es:[bx]             ;zachowanie w temp (AX)
                call    dp swap_Item2           ;pobranie adresu Item2
                xchg    ax, es:[bx]             ;zamiana temp <-> Item2
                call    dp  swap_Item1          ;pobranie adresu Item1
                mov     es:[bx], ax             ;zachowanie temp w Item1
                pop     es                      ;przywrócenie es
                pop     bx                      ;przywrócenie bx
                pop     ax                      ;przywrócenie ax
                ret     8                       ;powrót i zdjęcie Item1 i Item2
```

```
swap                    endp
```

Kilka przykładów wywołania swap:

```
;swap(A[i], i)  -- wersja 8086
                        mov     ax, seg thunk1
                        push    ax
                        lea     ax, thunk1
                        push    ax
                        mov     seg, thunk2
                        push    ax
                        lea     ax, thunk2
                        push    ax
                        call    swap

;swap(A[i], i) –wersja 80186 i wersje późniejsze
                        push    seg thunk1
                        push    offset thunk1
                        push    seg thunk2
                        push    offset thunk2
                        call    swap
                        -
                        -
                        -
;Notka:                 kod ten zakłada, że A jest tablicą dwubajtową liczb całkowitych. Zauważmy również, że
;                       jaki segment(y) zawierają A i I

thunk1                  proc    far
                        mov     bx, seg A               ,musimy zwrócić  seg A w ES
                        push    bx                      ;zachowujemy na później
                        mov     bx, seg   i             ;potrzebujemy segment I, żeby mieć do niego dostęp
                        mov     es, bx
                        mov     bx, es:i                ;pobranie wartości I
                        shl     bx, 1
                        lea     bx, A[bx]
                        pop     es                      ;zwrócenie segmentu A[I] w es
                        ret
thunk1                  endp
thunk2                  proc    near
                        mov     bx, seg i               ;potrzebujemy zwrócić  segment I w es
                        mov     es, bx
                        lea     bx, i
                        ret
thunk2                  endp
```

Dodatkowe informacje o rekordzie aktywacji i ramce stosu pojawią się później w tym rozdziale w sekcji o zmiennych lokalnych

---

11.5.10 PRZEKAZYWANIE PARAMETRÓW W STRUMIENIU KODU

innym miejscem gdzie możemy przekazać parametry jest strumień kodu bezpośrednio po instrukcji call. Podprogram print w pakiecie Standardowej Biblioteki UCR dostarcza doskonałego przykładu:

```
                        print
                        byte   „Ten parametr jest w strumieniu kodu.:,0
```

Normalnie podprogram zwraca sterowanie do pierwszej instrukcji bezpośrednio następującej po instrukcji call. Ale co wydarzyłoby się tu gdyby 80x86 próbował zinterpretować kod ASCII dla „To.....” jako instrukcję. Dało by to nam niewłaściwy wynik. Na szczęście, możemy przeskoczyć ten ciąg kiedy wracamy z podprogramu

Więc jak korzystamy z dostępu do tych parametrów? Łatwo. Adres powrotu na stosie wskazuje na nie. Rozważmy poniższą implementację print:

```
MyPrint                 proc    near
                        push    bp
```

```
                mov     bp, sp
                push    bx
                push    ax
                mov     bx, 2[bp]          ,Ładowanie adresu powrotu do BX
PrintLp;        mov     al, cs:[bx]        ;pobranie nowego znaku
                cmp     al, 0              ;sprawdzenie czy koniec ciągu
                jz      EndStr
                putc                       ;jeśli nie koniec, drukujemy ten znak
                inc     bx                 ;przejście do nowego znaku
                jmp     PrintLp
EndStr;         inc     bx                 ;wskazuje pierwszy bajt poza zerem
                mov     2[bp], bx          ;zachowanie jako nowego adresu powrotu
                pop     ax
                pop     bx
                pop     bp
                ret
MyPrint         endp
```

Procedura ta zaczyna się od odłożenia wszystkich zmienianych rejestrów na stos. Pobierany jest adres powrotu pod offset 2[BP] i drukowany każdy kolejny znak aż do napotkania bajtu zerowego. Odnotujmy obecność przedrostka przesłonięcia segmentu cs: w instrukcji mov al, cs:[bx]. Ponieważ dane pochodzą z segmentu kodu, prefiks t gwarantuje, że MyPrint pobierze dany znak z właściwego segmentu.. Po napotkaniu bajtu zerowego, MyPrint wskaże bx jako pierwszy bajt po zerze. Jest to adres pierwszej instrukcji występującej po zerowym bajcie kończącym. CPU stosuje tą wartość jako nowy adres powrotu. Teraz instrukcja ret zwróci sterowanie do instrukcji następnej po ciągu.

Powyższy kod pracuje dobrze jeśli MyPrint jest bliską procedurą. Jeśli musimy wywołać MyPrint z różnych segmentów, musimy stworzyć daleką procedurę. Oczywiście, główna różnica jest taka, że daleki adres powrotu w tym momencie będzie na stosie – będziemy musieli zastosować daleki wskaźnik zamiast wskaźnika bliskiego. Poniższa implementacja MyPrint pokazuje ten przypadek.

```
MyPrint     proc    far
            push    bp
            mov     bp, sp
            push    bx                 ;zachowanie ES, AX i BX
            push    ax
            push    es
            les     bx,2[bp]           ;załadownie adresu powrotu do ES:BX
            mov     al, es:[bx]        ;pobranie nowego znaku
            cmp     al, 0              ;sprawdzenie czy koniec ciągu
            jz      EndStr
            putc                       ;jeśli nie koniec, drukuj ten znak
            inc     bx                 ;przejście do następnego znaku
            jmp     PrintLp
EndStr:     inc     bx                 ;wskazuje na pierwszy bajt po zerze
            mov     2[bp], bx          ;zachowanie jako nowy adres powrotu
            pop     es
            pop     ax
            pop     bx
            pop     bp
            ret
MyPrint     endp
```

Zauważmy, że ten kod nie przechowuje es w lokacji [bp+4]. Powód jest całkiem prosty – es nie zmienia się podczas wykonywania tej procedury; przechowanie es w lokacji [bp+ 4] nie zmienia wartości pod tą lokacją. Zauważmy, że ta wersja MyPrint pobiera każdy znak z lokacji es:[bx] zamiast cx:[bx]. Jest tak ponieważ ciąg jaki drukujemy jest w części wywołującej, a to może nie być ta sama część zawierająca MyPrint.

Poza pokazaniem jak przekazać parametry w strumieniu kodu, podprogram MyPrint również wykazuje inne pojęcie: parametry o zmiennej długości. Ciąg następny po call może być każdej praktycznie długości . Zerowy bajt kończący oznacza koniec listy parametrów. Są dwa łatwe sposoby operowania parametrami o zmiennej długości. Albo zastosować jaką specjalną wartość kończącą (jak zero) lub możemy przekazać specjalną wartość długości która powie podprogramowi ile parametrów jest przekazywanych. Obie metody mają swoje zalety i wady. Stosowanie wartości specjalnej dla zakończenia listy parametrów wymaga, żebyśmy wybrali wartość, która nigdy nie pojawia się na liście. Na przykład, MyPrint używa zera jako wartości kończącej, więc nie może wydrukować znaku NULL (którego kodem ASCII jest zero). Czasami to nie jest ograniczeniem.

Wyszczególnienie specjalnej długości parametru jest innym mechanizmem, który możemy użyć do przekazania listy parametrów o zmiennej długości .Chociaż nie jest wymagane żaden specjalny kod lub ogranicznik zakresu możliwych wartości, które mogą być przekazane do podprogramu, ustawienie długości parametrów i pielęgnacja kodu wynikowego może być prawdziwym koszmarem.

Chociaż przekazanie parametrów w strumieniu kodu jest idealnym sposobem do przekazania listy parametrów o zmiennej długości, możemy przekazać również listę parametrów o stałej długości. Strumień kodu jest doskonałym miejscem do przekazania stałych (podobnie jak ciąg stałych przekazanych do MyPrint) i parametrów referencyjnych, Rozważmy poniższy kod, który oczekuje trzech parametrów przez referencję:

Sekwencja wywołania:

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

Procedura:

```
AddEm           proc    near
                push    bp
                mov     bp, sp
                push    si
                push    bx
                push    ax
                mov     si, [bp+2]          ,Pobranie adresu powrotu
                mov     bx, cs:[si+2]       ;pobranie adresu J
                mov     ax, [bx]            ;pobranie wartości J
                mov     bx, cs:[si+4]       ;pobranie adresu K
                add     ax, [bx]            ;dodanie wartości K
                mov     bx, cs:[si]         ;pobranie adresu I
                mov     [bx], ax            ;przechowanie wyniku
                add     si, 6               ;skok poza parametry
                mov     [bp+2], si          ;zachowanie adresu powrotu
                pop     ax
                pop     bx
                pop     si
                pop     bp
                ret
AddEm           endp
```

Podprogram ten dodaje J i K razem a wynik przechowuje w I. Zauważmy, że kod ten stosuje 16 bitowy bliski wskaźnik dla przekazania adresów I, J i K do AddEm. Dlatego też I, J i K muszą być w bieżącym segmencie danych. W powyższym przykładzie, AddEm jest bliską procedurą. Aby była daleką procedurą musielibyśmy pobrać czterobajtowy wskaźnika ze stosu zamiast dwubajtowego wskaźnika. Poniżej jest daleka wersja AddEm

```
AddEm           proc    far
                push    bp
                mov     bp, sp
                push    si
                push    bx
                push    ax
                push    es
                les     si, [bp+2]          ;Get far ret adrs into es:si
                mov     bx, es:[si+2]       ;Get address of J
                mov     ax, [bx]            ;Get J's value
```

```
        mov     bx, es:[si+4]       ;Get address of K
        add     ax, [bx]            ;Add in K's value
        mov     bx, es:[si]         ;Get address of I
        mov     [bx], ax            ;Store result
        add     si, 6               ;Skip past parms
        mov     [bp+2], si          ;Save return address
        pop     es
        pop     ax
        pop     bx
        pop     si
        pop     bp
        ret
AddEm   endp
```

W obu wersjach AddEm, wskaźniki do I, J i K przekazane w strumieniu kodu są bliskimi wskaźnikami. Obie wersje zakładają, że I, J i K wszystkie są w bieżącym segmencie danych. Jest możliwe przekazanie dalekich wskaźników do tych zmiennych, lub nawet bliskich wskaźników do tych dalekich wskaźników do innych, w strumieniu kodu. Poniższy przykład nie jest całkiem taki ambitny, to jest bliska procedura, która oczekuje dalekiego wskaźnika, ale nie pokazuje jakichś znaczących różnic

Sekwencja wywołania:

```
call    AddEm
dword   I, J, K
```

Kod:

```
AddEm   proc    near
        push    bp
        mov     bp, sp
        push    si
        push    bx
        push    ax
        push    es
        mov     si, [bp+2]          ;Get near ret adrs into si
        les     bx, cs:[si+2]       ;Get address of J into es:bx
        mov     ax, es:[bx]         ;Get J's value
        les     bx, cs:[si+4]       ;Get address of K
        add     ax, es:[bx]         ;Add in K's value
        les     bx, cs:[si]         ;Get address of I
        mov     es:[bx], ax         ;Store result
        add     si, 12              ;Skip past parms
        mov     [bp+2], si          ;Save return address
        pop     es
        pop     ax
        pop     bx
        pop     si
        pop     bp
        ret
AddEm   endp
```

Zauważmy, że jest 12 bajtów parametrów w strumieniu kodu tym razem. Jest tak dlatego, że kod ten zawiera instrukcję add si, 12 zamiast add si, 6 pojawiającą się w poprzednich wersjach.

W przykładach podanych do tego momentu, MyPrint oczekiwał przekazania parametrów przez wartość, drukował aktualne znaki następujące po call a AddEm oczekuje trzech parametrów przekazanych przez referencję – ich adresy występują w strumieniu kodu. Oczywiście, możemy również przekazać parametry przez wartość – powrót, przez wynik, przez nazwę lub leniwe wartościowanie w strumieniu kodu. Następny przykład jest modyfikacją AddEm, który stosuje przekazywanie przez wynik dla I, przekazywanie przez wartość-powrót dla J i przekazywanie przez nazwę dla K. Wersja ta jest odrobinę różna ponieważ modyfikuje zarówno J jak I, żeby uzasadnić stosowanie parametru wartość-powrót.

```
; AddEm(Result I:integer; ValueResult J:integer; Name K);
;
;       Computes            I:= J;
;                           J := J+K;
;
; Presumes all pointers in the code stream are near pointers.

AddEm           proc    near
                push    bp
                mov     bp, sp
                push    si                      ;Pointer to parameter block.
                push    bx                      ;General pointer.
                push    cx                      ;Temp value for I.
                push    ax                      ;Temp value for J.

                mov     si, [bp+2]              ;Get near ret adrs into si

                mov     bx, cs:[si+2]           ;Get address of J into bx
                mov     ax, es:[bx]             ;Create local copy of J.
                mov     cx, ax                  ;Do I:=J;

                call    word ptr cs:[si+4]      ;Call thunk to get K's adrs
                add     ax, [bx]                ;Compute J := J + K

                mov     bx, cs:[si]             ;Get address of I and store
                mov     [bx], cx                ; I away.

                mov     bx, cs:[si+2]           ;Get J's address and store
                mov     [bx], ax                ; J's value away.

                add     si, 6                   ;Skip past parms
                mov     [bp+2], si              ;Save return address
                pop     ax
                pop     cx
                pop     bx
                pop     si
                pop     bp
                ret
AddEm           endp
```

Przykład sekwencji wywołania:

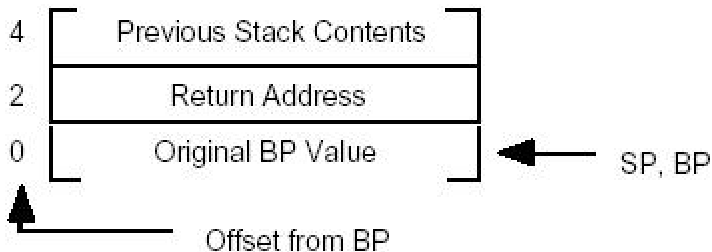
```
; AddEm(I,J,K)

                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
```

Notka: przekazaliśmy I przez referencję zamiast przez wynik , w tym przykładzie wywołując

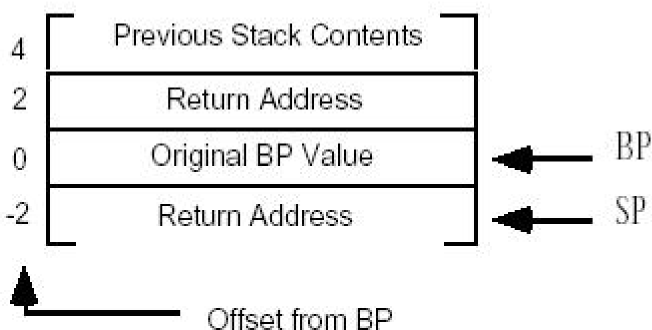
AddEm(I, J, A[i])

stworzymy różne wyniki. Czy można wytłumaczyć dlaczego?

Przekazanie parametrów w strumieniu kodu pozwala nam wykonać naprawdę zmyślne zadania. Poniższy przykład jest znacznie bardziej złożony niż inne w tej sekcji, ale demonstruje



Rysunek 11.13: Stos na wejściu do procedury ForStmt



Rysunek 11.14 Stos przed opuszczeniem procedury ForStmt

potęgę przekazywania parametrów w strumieniu kodu i ,mimo złożoności tego przykładu, jak można uprościć nasze zadania programistyczne.

Poniższe dwa podprogramy implementują instrukcje for /next, podobnie jak w BASICu, w języku asemblera. Sekwencja wywołania dla tych podprogramów jest następująca:

```
call    ForStmt
word    <LoopControlVar>, <StartValue>, <EndValue>
-
-
<instrukcje treści pętli>
-
-
call    Next
```

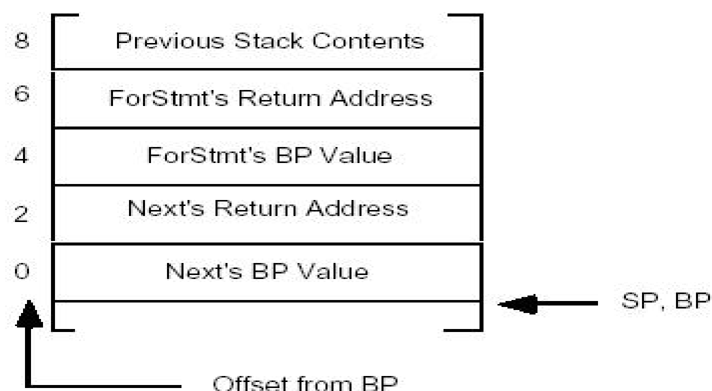
Kod ten ustawia zmienną sterującą pętli (której bliski adres przekazujemy jako pierwszy parametr, przez referencję) na wartość początkową (przekazywaną przez wartość jako drugi parametr). Wtedy zaczyna się wykonywanie treści pętli. Po wykonaniu wywołania Next, program ten zwiększy zmienną sterującą pętli i potem porówna ją z wartością końcową. Jeśli jest mniejsza lub równa wartości końcowej , sterowanie zwracane jest na początek treści pętli (pierwsza instrukcja po dyrektywie word). W innym przypadku jest kontynuowane wykonywanie pierwszej instrukcji po wywołaniu Next.

Teraz prawdopodobnie zastanawiamy się „Jak przekazać sterowanie do początku treści pętli?" W końcu nie ma etykiety przy instrukcji i nie ma instrukcji przekazania sterowania, która skacze do pierwszej instrukcji po dyrektywie word. Cóż, okazuje się, ze możemy zrobić to trochę skomplikowaną manipulacją stosu. Rozważmy jak będzie wyglądał stos na wejściu do podprogramu ForStmt, po odłożeniu bp na stos (zobacz rysunek 11.13)

Normalnie, podprogram ForStmt zdjąłby bp i zwrócił instrukcją ret, która usunęła rekord aktywacji ForStmt ze stosu. Przypuśćmy, że zamiast tego ForStmt wykonuje poniższe instrukcje:

```
add     word ptr 2[bp], 2       ;przeskok parametrów
push    [bp+2]                  ;zrobienie kopii adresu powrotu
mov     bp, [bp]                ;przywrócenie wartości bp
ret                             ;powrót do programu wywołującego
```

Właśnie przed powyższą instrukcją ret, stos ma wejścia pokazane na rysunku 11.14



Rysunek 11.15: Stos na wejściu do procedury Next

Po wykonaniu instrukcji ret, Forstmt będzie zwracał właściwy adres powrotu ale pozostawi swój rekord aktywacji n stosie!

Po wykonaniu instrukcji w treści pętli, program wywoła podprogram Next. Na początkowym wejściu do Next (i ustawieniu bp), stos zawiera wejścia pojawiające się na rysunku 11.15

Ważną rzeczą do ujrzenia tutaj jest to, że adres powrotu ForStmt, który wskazuje na pierwszą instrukcję po dyrektywie word, jest jeszcze na stosie i dostępny Next pod offsetem [bp+6]. Next może użyć tego adresu powrotu do wzmocnienia dostępu do parametrów i powrotu do właściwego miejsca jeśli to konieczne. Next zwiększa zmienną sterującą pętli i porównuje ją z wartością końcową. Jeśli wartość zmiennej sterującej pętli jest mniejsza lub równa wartości końcowej, Next zdejmuje swój adres powrotu i wraca przez adres powrotu ForStmt. Jeśli wartość zmiennej sterującej pętli jest większa niż wartość końcowa, Next wraca przez swój własny adres powrotu i usuwa rekord aktywacji ForStmt ze stosu. Poniżej mamy kod dla Next i ForStmt:

```
                .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]    ;Get return address
                mov       ax, cs:[bx+2];Get starting value
                mov       bx, cs:[bx]   ;Get address of var
                mov       [bx], ax      ;var := starting value
                add       wp [bp+2], 6 ;Skip over parameters
                pop       bx
```

```
                pop     ax
                push    [bp+2]          ;Copy return address
                mov     bp, [bp]        ;Restore bp
                ret                     ;Leave Act Rec on stack
ForStmt         endp

Next            proc near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                mov     bx, [bp+6]      ;ForStmt's rtn adrs
                mov     ax, cs:[bx-2]   ;Ending value
                mov     bx, cs:[bx-6]   ;Ptr to loop ctrl var
                inc     wp [bx]         ;Bump up loop ctrl
                cmp     ax, [bx]        ;Is end val < loop ctrl?
                jl      QuitLoop

; If we get here, the loop control variable is less than or equal
; to the ending value. So we need to repeat the loop one more time.
; Copy ForStmt's return address over our own and then return,
; leaving ForStmt's activation record intact.

                mov     ax, [bp+6]      ;ForStmt's return address
                mov     [bp+2], ax      ;Overwrite our return address
                pop     bx
                pop     ax
                pop     bp              ;Return to start of loop body
                ret

; If we get here, the loop control variable is greater than the
; ending value, so we need to quit the loop (by returning to Next's
; return address) and remove ForStmt's activation record.
```

```
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            ;End of J loop
                call    Next            ;End of I loop
                print
                byte    "All Done!",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
```

Kod przykładowy w głównym programie pokazuje, że pętle for zagnieżdżają się dokładnie jak oczekiwalibyśmy w językach wysokiego poziomu, takich jak BASIC, Pascal lub C. Oczywiście ,nie jest to szczególnie dobry sposób konstruowania pętli for w języku asemblera. Jest wiele razy wolniejszy niż zastosowanie standardowej techniki generowania pętli. Oczywiście, jeśli nie martwimy się o szybkość, jest to doskonały sposób do implementacji pętli. Jest to z pewnością łatwiejsze do odczytu i zrozumienia niż tradycyjne metody tworzenia pętli loop. Dla innej (bardziej wydajnej) implementacji pętli loop, sprawdźmy makro ForLp w Rozdziale Ósmym .

Strumień kodu jest bardzo dogodnym miejscem do przekazywania parametrów. Standardowa Biblioteka UCR czyni istotnym użycie tego mechanizmu przekazywania parametrów czyniąc go łatwym dla wywoływania pewnych podprogramów. Printf jest, być może, najbardziej złożonym przykładem, ale inne przykłady (zwłaszcza w bibliotece string) też obfitują.

Pomimo wygody, są wady przekazywania parametrów w strumieniu kodu. Po pierwsze, zapomnimy dostarczyć stosowną liczbę parametrów wymaganych przez procedurę, podprogram może się pogubić. Rozpatrzmy podprogram print z Biblioteki Standardowej UCR. Drukuje łańcuch znaków aż do zerowego bajtu zakończenia a potem zwraca sterowanie do pierwszej instrukcji po zerowym bajcie zakończenia. Jeśli opuścimy zerowy bajt zakończenia, podprogram print wesoło wydrukuje następne bajty opcodów jako znaki ASCII dopóki nie znajdzie bajtu zero. Ponieważ bajty zero często pojawiają się w środku instrukcji, podprogram print może zwrócić sterowanie w trakcie jakiejś innej instrukcji. To prawdopodobnie zrobi krach sprzętu. Wprowadzenie dodatkowego zera, które może wystąpić dużo częściej niż można sobie pomyśleć, jest innym problemem jaki programiści mają z podprogramem print. W takim przypadku, podprogram print będzie zwracał po napotkaniu pierwszego bajtu zero i próbował wykonać następny znak ASCII jako kod maszynowy. I ponownie mamy krach sprzętu.

Innym problemem z przekazywaniem parametrów w strumieniu kodu jest taki, że mamy trochę dłuższy dostęp do takich parametrów. Przekazanie parametrów w rejestrach, w zmiennych globalnych lub na stos jest odrobinę bardziej wydajne, zwłaszcza w krótkich podprogramach. Niemniej jednak, uzyskiwanie dostępu do parametrów w strumieniu kodu nie jest nadzwyczaj wolne, więc wygoda takich parametrów może przeważyć nad kosztami. Co więcej, wiele podprogramów (print jest dobrym przykładem) jest tak wolnych, że kilka dodatkowych mikrosekund nie robi już różnicy

---

11.5.11 PRZEKAZYWANIE PARAMETRÓW PRZEZ BLOK PARAMETRÓW

Innym sposobem przekazywania parametrów w pamięci jest przekazywanie przez blok parametrów. Blok parametrów jest to zbiór przyległych komórek pamięci zawierających parametry. Aby uzyskać dostęp do takich parametrów, przekazujemy podprogramowi wskaźnik do bloku parametrów. Rozważmy podprogram z poprzedniej sekcji, który oddawał razem J i K, przechowując wynik w I; kod który przekazuje te parametry przez blok parametrów może być następujący:

Sekwencja wywołania:

```
ParamBlock          dword   I
I                   word    ?               ; I, J i K muszą pojawić się w tym porządku
J                   word    ?
K                   word    ?
                    -
                    -
                    -
                    les     bx, ParamBlock
                    call    AddEm
                    -
                    -
                    -
     AddEm          proc    near
                    push    ax
                    mov     ax, es:2[bx]    ;pobranie wartości J
                    add     ax, es:4[bx]    ;dodanie wartości K
                    mov     es:[bx], ax     ;przechowanie wyniku w I
                    pop     ax
                    ret
     AddEm          endp
```

Zauważmy, że musimy zaalokować trzy parametry w sąsiednich komórkach pamięci.

Ta postać przekazywania parametrów pracuje dobrze kiedy przekazujemy kilka parametrów przez referencję, ponieważ możemy zainicjalizować wskaźniki do parametrów bezpośrednio wewnątrz asemblera. Na przykład przypuśćmy, że chcielibyśmy stworzyć podprogram rotate do którego przekazujemy cztery parametry przez referencję. Podprogram ten kopiowałby drugi parametr do pierwszego, trzeci do drugiego, czwarty do trzeciego a pierwszy do czwartego. Łatwy sposób wykonania tego w asemblerze to:

```
; Rotate-           On entry, BX points at a parameter block in the data
;                   segment that points at four far pointers. This code
;                   rotates the data referenced by these pointers.

Rotate              proc    near
                    push    es                      ;Need to preserve these
                    push    si                      ; registers
                    push    ax

                    les     si, [bx+4]              ;Get ptr to 2nd var
                    mov     ax, es:[si]             ;Get its value
                    les     si, [bx]                ;Get ptr to 1st var
                    xchg    ax, es:[si]             ;2nd->1st, 1st->ax
                    les     si, [bx+12]             ;Get ptr to 4th var
                    xchg    ax, es:[si]             ;1st->4th, 4th->ax
                    les     si, [bx+8]              ;Get ptr to 3rd var
                    xchg    ax, es:[si]             ;4th->3rd, 3rd->ax
                    les     si, [bx+4]              ;Get ptr to 2nd var
                    mov     es:[si], ax             ;3rd -> 2nd

                    pop     ax
                    pop     si
                    pop     es
                    ret
Rotate              endp
```

Przy wywołaniu tego podprogramu przekazujemy mu wskaźnik do grupy czterech dalekich wskaźników w rejestrze bx. Na przykład przypuśćmy ,że chcielibyśmy obrócić pierwsze elementy czterech różnych tablic, drugie elementy tych czterech tablic i trzecie elementy tych czterech tablic. Możemy do tego zastosować poniższy kod:

```
                    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]
```

Zauważmy, że wskaźnik do bloku parametrów sam jest parametrem. Przykłady w tej sekcji przekazują ten wskaźnik w rejestrach. Jednakże, możemy przekazać ten wskaźnik gdziekolwiek gdzie przekazujemy inne parametry referencyjne – w rejestrach, w zmiennych globalnych, na stos, w strumieniu kodu, nawet w innym bloku parametrów! Takie są wariacje na ten temat, jednak, zostawmy naszą wyobraźnię. Najlepszym miejscem do przekazania wskaźnika do bloku parametrów są rejestry. Ten tekst generalnie przyjmuje tą zasadę.

Chociaż początkujący programiści asemblerowi rzadko używają bloku parametrów, one z pewnością mają swoje miejsce. Niektóre funkcje IBM PC BIOS i MS-DOS używają tego mechanizmu przekazywania parametrów. Bloki parametrów, ponieważ możemy inicjalizować ich wartości podczas asemblacji (stosując byte, word itd.) dostarczają szybkiego, wydajnego sposobu przekazywania parametrów do procedury.

Oczywiście, możemy przekazać parametry przez wartość, referencję, wartość-powrót, wynik lub nazwę w bloku parametrów. Poniższy kawałek kodu jest modyfikacją powyższej procedury Rotate, gdzie pierwszy parametr jest przekazywany przez wartość (jego wartość pojawia się wewnątrz bloku parametrów), drugi jest przekazywany przez referencję, trzeci przez wartość-powrót a czwarty przez nazwę (nie ma przekazywania przez wynik ponieważ Rotate odczytuje i zapisuje wszystkie wartości) . Dla uproszczenia kod ten używa bliskich wskaźników i zakłada, że wszystkie zmienne pojawiają się w segmencie danych:

```
; Rotate-        On entry, DI points at a parameter block in the data
;                segment that points at four pointers. The first is
;                a value parameter, the second is passed by reference,
;                the third is passed by value/return, the fourth is
;                passed by name.

Rotate           proc     near
                 push     si                    ;Used to access ref parms
                 push     ax                    ;Temporary
                 push     bx                    ;Used by pass by name parm
                 push     cx                    ;Local copy of val/ret parm

                 mov      si, [di+4]            ;Get a copy of val/ret parm
                 mov      cx, [si]

                 mov      ax, [di]              ;Get 1st (value) parm
                 call     word ptr [di+6]       ;Get ptr to 4th var
                 xchg     ax, [bx]              ;1st->4th, 4th->ax
                 xchg     ax, cx                ;4th->3rd, 3rd->ax
                 mov      bx, [di+2]            ;Get adrs of 2nd (ref) parm
                 xchg     ax, [bx]              ;3rd->2nd, 2nd->ax
                 mov      [di], ax              ;2nd->1st

                 mov      bx, [di+4]            ;Get ptr to val/ret parm
                 mov      [bx], cx              ;Save val/ret parm away.

                 pop      cx
                 pop      bx
                 pop      ax
                 pop      si
                 ret
Rotate           endp
```

Przykład wywołania tego podprogramu może być taki:

```
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 WYNIKI FUNKCJI

Funkcje zwracają wynik, który jest niczym więcej niż wynikiem parametru. W języku asemblera jest kilka różnic pomiędzy procedurą a funkcją. Jest tak prawdopodobnie dlatego że nie ma dyrektyw „func" i „endf" Funkcje i procedury są zazwyczaj różne w HLLach, wywołanie funkcji pojawia się tylko w wyrażeniach, podprogramy wywoływane są jako instrukcje. Język asemblera nie rozróżnia pomiędzy nimi.

Możemy zwrócić wynik funkcji i w to samo miejsce przekazać i zwrócić parametry. Typowo jednakże funkcja zwraca tylko pojedynczą wartość (lub pojedyncza strukturę danych) jako wynik funkcji. Metody i lokacje używane do zwracania wyniku funkcji są tematem następnych trzech sekcji.

11.6.1 ZWRACANIE WYNIKU FUNKCJI W REJESTRRZE

Podobnie jak przy parametrach, rejestry 80x86 są najlepszym miejscem do zwracania wyniku funkcji. Podprogram getc ze Standardowej Biblioteki UCR jest dobrym przykładem funkcji, która zwraca wartość w jednym z rejestrów CPU. Odczytuje znak z klawiatury i zwraca kod ASCII dla tego znaku w rejestrze al. Ogólnie, funkcje zwracają swój wynik w następujących rejestrach:

Use	First		Last
Bytes:		al, ah, dl, dh, cl, ch, bl, bh	
Words:		ax, dx, cx, si, di, bx	
Double words:		dx:ax	On pre-80386
		eax, edx, ecx, esi, edi, ebx	On 80386 and later.
16-bitOffsets:		bx, si, di, dx	
32-bit Offsets		ebx, esi , edi, eax, ecx, edx	
Segmented Pointers:		es:di, es:bx, dx:ax, es:si	Do not use DS.

Jeszcze raz, tablica ta przedstawia ogólne wskazówki. Jeśli mamy skłonności do robienia czegoś innego, możemy zwrócić wartość podwójnego słowa w (cl, dh, al ,bh). Jeśli zwracamy wynik funkcji w jakichś rejestrach, nie powinniśmy zachowywać i przywracać tych rejestrów. Robiąc tak możemy zniszczyć cały cel tej funkcji.

11.6.2 ZWRACANIE WYNIKU FUNKCJI NA STOS

Innym dobrym miejscem gdzie możemy zwrócić wynik funkcji jest stos. Pomysłem jest odłożenie jakichś fikcyjnych wartości na stos dla stworzenia przestrzenia dla wyniku funkcji. Funkcja, przed opuszczeniem, przechowuje swój wynik w tej lokacji. Kiedy funkcja wraca do programu wywołującego, zdejmuje wszystko ze stosu za wyjątkiem tego wyniku funkcji. Wiele HLLi stosuje tą technikę 9chociaż większość HLLi na IBM PC zwraca wynik funkcji w rejestrach). Poniższa sekwencja kodu pokazuje jak wartości mogą być zwracane na stos:

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

m := PasFunc(2,n,l);
```

In assembly:

```
PasFunc_rtn     equu    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
```

Sekwencja wywołująca:

```
push    ax                  ;miejsce dla zwracanego wyniku funkcji
mov     ax, 2
push    ax
push    n
push    l
call    PasFunc
pop     ax                  ;pobranie zwracanego wyniku funkcji
```

Na 80286 lub późniejszych procesorach możemy również użyć kodu:

```
push    ax                  ;miejsce dla zwracanego wyniku funkcji
push    2
push    n
push    l
call    PasFunc
pop     ax                  ;pobranie zwracanego wyniku funkcji
```

Chociaż program wywołujący odkłada osiem bajtów danych na stos, PasFunc usuwa tylko sześć. Pierwszy „parametr" na stosie jest wynikiem funkcji. Funkcja musi zostawić tą wartość na stosie kiedy wraca.

11.6.3 ZWRACANIE WYNIKU FUNKCJI W KOMÓRKACH PAMIĘCI

Innym sensownym miejscem zwracania wyniku funkcji są komórki pamięci. Możemy zwracać wartość funkcji w zmiennych globalnych lub możemy zwrócić wskaźnik (przypuszczalnie w rejestrze lub parze rejestrów) do bloku parametrów. Proces ten jest praktycznie identyczny do przekazywania parametrów do procedury lub funkcji w zmiennych globalnych lub przez blok parametrów.

Zwracanie parametrów przez wskaźnik do bloku parametrów jest doskonałym sposobem zwracania dużych struktur danych jako wyniku funkcji. Jeśli funkcja zwraca całą tablicę, najlepszym sposobem do zwrócenia tej tablicy jest przydzielenie pamięci, przechowanie danych w tym obszarze i wychodząc do podprogramu wywołującego zwolnienie tej pamięci. Większość języków wysokiego poziomu, które pozwalają nam na zwracanie dużych struktur danych jako wyniku funkcji stosuje tą technikę.

Oczywiście jest bardzo mała różnica pomiędzy zwracaniem wyniku funkcji w pamięci a mechanizmem przekazania parametru przez wynik.

11.7 EFEKT UBOCZNY

Efekt uboczny jest jakimś obliczeniem lub działaniem procedury, które nie jest pierwszoplanowe dla tej procedury. Na przykład, jeśli wybierzemy nie chronione wszystkie rejestry wewnątrz procedury, modyfikacja tych rejestrów jest efektem ubocznym tej procedury. Programowy efekt uboczny, to znaczy praktyczne zastosowanie efektów ubocznych procedury jest bardzo niebezpieczne. Wszyscy programiści zbyt często polegają na efekcie ubocznym procedury. Późniejsze modyfikacje mogą zmienić efekt uboczny, unieważniając cały kod polegający na tym efekcie ubocznym. Może uczynić to nasz program trudnym do zdebuggowania i pielęgnacji. Dlatego też powinniśmy unikać programowania efektów ubocznych.

Być może niektóre przykłady programowych efektów ubocznych będą pomocne przy rozjaśnianiu trudności jakie możemy napotkać. Poniższa procedura wyzerowuje tablicę. Z powodu wydajności, czynimy program wywołujący odpowiedzialnym za zachowanie niezbędnych rejestrów. W wyniku jednym efektem ubocznym tej procedury jest to, że rejestry bx i cx są modyfikowane. W szczególności rejestr cx zawiera zero przy zwracaniu.

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

Jeśli nasz kod oczekuje aby cx zawierał zero po wykonaniu tego podprogramu, będziemy polegać na efekcie ubocznym procedury ClrArray. Głównym celem tego kodu jest wyzerowanie tablicy a nie ustawienie rejestru cx na zero. Później, jeśli zmodyfikujemy procedurę ClrArray do następnej, nasz kod zależny od cx zawierającego zero , już nie będzie pracowała poprawnie:

```
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
```

Więc jak możemy uniknąć pułapek programowych efektów ubocznych w naszych procedurach? Poprzez ostrożne budowanie naszego kodu i zwrócenie uwagi dokładnie jak nasz kod wywołujący i podporządkowana procedura wzajemnie ze sobą oddziałują. Poniższe zasady pomogą uniknąć nam problemów z programowymi efektami ubocznymi:

- Zawsze właściwie opisuj warunki wejściowe i wyjściowe procedury. Nigdy nie polegaj na innych warunkach wejściowych i wyjściowych jak tylko tych operacji opisanych
- Dziel swoje procedury tak, żeby obliczały pojedynczą wartość lub wykonywały pojedynczą operację. Podprogramy, które robią dwa lub więcej zadań są , z definicji, twórcami efektów ubocznych ,chyba ,że każde wywołanie tego podprogramu wymaga wszystkich obliczeń i operacji
- Kiedy aktualizujemy kod w procedurze, upewnijmy się ,że spełniliśmy warunki wejścia i wyjścia. Jeśli nie, albo modyfikujemy program, żeby to robił albo uaktualniamy dokumentację tej procedury odzwierciedlającą nowe warunki wejścia i wyjścia
- Zawsze zachowujmy i zwracajmy wszystkie rejestry modyfikowanej procedury
- Unikajmy przekazywania parametrów i wyników funkcji w zmiennych globalnych.
- Unikajmy przekazywania parametrów przez referencję (z intencją modyfikowania ich dla użytkowania przez kod wywołujący)

Zasady te, podobnie jak wszystkie inne zasady, pewnie będą złamane. Dobre praktyki programistyczne często są poświęcane na ołtarzu wydajności. Nie ma nic złego w łamaniu tych zasad tak często jak to wydaje się konieczna. Jednakże nasz kod będzie trudny do zdebuggowania i pielęgnacji jeśli naruszamy te zasady często. Ale taka jest cena wydajności. Dopóki nie nabierzemy dosyć doświadczenia aby uczynić rozsądny wybór o użyciu efektów ubocznych w naszych programach powinniśmy ich unikać. Dużo częściej zastosowanie efektów ubocznych powoduje więcej problemów niż daje rozwiązań

11.8 ZMIENNA PAMIĘCI LOKALNEJ

Czasami procedura będzie wymagała czasowego przechowania, które nie jest wymagane, kiedy procedura wraca. Możemy łatwo alokować taka zmienną pamięci lokalnej na stosie

80x86 wspiera zmienną pamięci lokalnej takim samym mechanizmem jaki stosuje dla parametrów – używa rejestrów bp i sp przy dostępie i alokacji takich zmiennych. Rozważmy poniższy program pascalowski;

```
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 zwykle alokuje zmienne globalne w segmencie danych a lokalne zmienne w segmencie stosu. Dlatego też powyższy program alokuje 50,002 słów pamięci lokalnej (30,001 słów dla Proc1 i 20,001 dla Proc2). Ponieważ 50,002 słów zajmuje 100,004 bajtów pamięci mamy mały problem – CPU 80x86 w trybie rzeczywistym ogranicza segment stosu do 65,536 bajtów. Pascal unika tego problemu przez dynamiczną alokację pamięci lokalnej przy wchodzeniu do procedury i dealokowanie pamięci lokalnej przy powrocie. Chyba ,że Proc1 i Proc2 są obie aktywne (co może zdarzyć się jeśli Proc1 wywołuje Proc2 lub vice versa), jest wystarczająco pamięci dla tego programu. Nie potrzebujemy 30,001 słów dla Proc1 i 20,001 słów dla Proc2 w tym samym czasie. Więc Proc1 alokuje i używa 60,002 bajtów pamięci, potem dealokuje tą pamieć i zwraca (zwalnia 60,002 bajty). Następnie Proc2 alokuje 40,002 bajty pamięci, używa jej, dealokuje ją i wraca do programu wywołującego. Zauważmy ,że Proc1 i Proc2 dzielą wiele tych samych komórek pamięci. Jednakże robią to w różnym czasie. Tak długo jak te zmienne są tymczasowe, czyli wartości nie potrzebujemy zachować od jednego wywołania procedury do drugiego, ta postać alokacji pamięci lokalnej pracuje dobrze.

Poniższe porównanie pomiędzy procedurą pascalowską a jej odpowiednikiem w języku asemblera daje nam kod, który jest dobrym pomysłem jak alokować pamięć lokalną na stosie:

```
procedure LocalStuff (i, j,k:integer);
var l,m,n:integer; {zmienna lokalna}
begin
    i := i+2;
    j := l*k+j;
    n := j-l;
    m := l+j+n;
end;
```

Sekwencja wywołująca:

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

Kod języka asemblera:

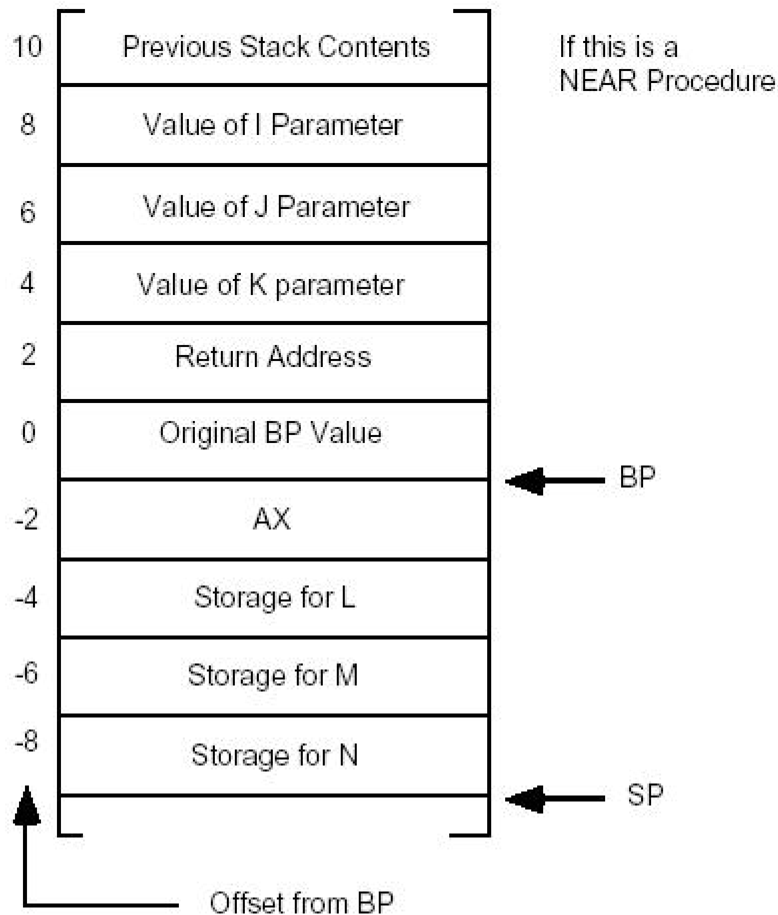
```
LStuff_i        equ     8[bp]
LStuff_j        equ     6[bp]
LStuff_k        equ     4[bp]
LStuff_l        equ     -4[bp]
LStuff_m        equ     -6[bp]
LStuff_n        equ     -8[bp]

LocalStuff      proc    near
                push    bp
                mov     bp, sp
                push    ax
                sub     sp, 6                   ;Allocate local variables.
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 already contains j
                mov     LStuff_n, ax
                add     ax, LStuff_l            ;AX already contains n
                add     ax, LStuff_j
                mov     LStuff_m, ax

                add     sp, 6                   ;Deallocate local storage
                pop     ax
                pop     bp
                ret     6
LocalStuff      endp
```

Instrukcja sub sp, 6 czyni miejsce dla trzech słów na stosie. Możemy zaalokować l, m i n w tych trzech słowach. Możemy odnieść się do tych zmiennych poprzez indeksowanie rejestrem bp stosując offset ujemny (zobacz kod powyżej) Przy osiągnięciu instrukcji przy etykiecieL0, stos wygląda tak jak na rysunku 11.15.

Kod ten używa dopasowującej instrukcji add sp, 6 na końcu procedury dla dealokowania pamięci lokalnej. Wartość , którą dodajemy do wskaźnika stosu musi dokładnie zgadzać się z wartością jaką odjęliśmy, kiedy alokowaliśmy tą pamięć. Jeśli te dwie wartości się nie zgadzają, wskaźnik stosu na wejściu do podprogramu nie zgadza się ze wskaźnikiem stosu na wyjściu; jest tak jak odłożenie i zdjęcie zbyt wielu pozycji wewnątrz procedury

W odróżnieniu od parametrów, które mają stały offset w rekordzie aktywacji, możemy alokować zmienne lokalne w dowolnym porządku. Tak długo jak jesteśmy zgodni z naszą przydzieloną lokacja, możemy alokować je w wybrany przez nas sposób. Zapamiętajmy jednak ,że 80x86 wspiera dwie formy trybu adresowania disp[bp]. Stosuje jednobajtowe przemieszczenie kiedy jest to zakres –128..+127. Stosuje dwubajtowe przemieszczenie dla wartości z zakresu –32,786...+32,767. Dlatego też, powinniśmy umieszczać wszystkie elementarne typy danych i inne małe struktury blisko wskaźnika bazowego, aby można było użyć jednobajtowego przemieszczenia. Powinniśmy umieszczać duże tablice i inne struktury danych poniżej mniejszych zmiennych na stosie.

Nie musimy się martwić większość czasu o alokowanie zmiennych lokalnych na stosie. Większość programów nie wymaga więcej niż 64Kb pamięci. CPU działa na zmiennych globalnych szybciej niż zmiennych lokalnych. Są dwie sytuacje, gdzie alokowanie zmiennych lokalnych jako globalnych w segmencie danych nie jest praktyczne: kiedy łączymy język asemblera z HLLem takim jak Pascal i kiedy piszemy kod rekurencyjny. Kiedy łączymy z Pascalem, nasz kod asemblerowy może nie mieć segmentu danych, który może używać, rekurencja często wymaga mnożenia egzemplarzy tej samej zmiennej lokalnej.



Rysunek 11.16 Stos na wejściu do procedury Next

11.9 REKURENCJA

Rekurencja występuje wtedy kiedy procedura wywołuje samą siebie. Poniżej mamy na przykład procedurę rekurencyjną:

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

Oczywiście, CPU nigdy nie będzie wykonywał instrukcji ret na końcu tej procedury. Na wejściu do Recursive, procedura ta będzie bezpośrednio wywoływała samą siebie znowu i sterownie nigdy nie zostanie przekazane do instrukcji ret. W tym szczególnym przypadku, wynik rekurencji ucieknie nam w pętlę nieskończoną

Pod wieloma względami rekurencja jest podobna do iteracji (to znaczy wykonywanie się powtarzających pętli0 Poniższy kod również tworzy pętlę nieskończoną:

```
Recursive   proc
            Jmp     Recursive
            Ret
Recursive   endp
```

Jest jednakże jedna znacząca różnica pomiędzy tymi implementacjami.. ta pierwsza wersja Recursive odkłada na stos adres powrotu przy każdym wywołaniu podprogramu. To nie dzieje się w powyższym przykładzie (ponieważ instrukcja jmp nie ma wpływu na stos).

Podobnie jak struktury pętlujące, rekurencja wymaga warunku zakończenia aby zatrzymać nieskończoną rekurencję. Recursive może być przepisana z warunkiem zakończenia jak następuje:

```
Recursive       proc
                dec     ax
                jz      QuitRecursive
                call    Recursive
QuitRecursive   ret
Recursive       endp
```

Ta modyfikacja podprogramu powoduje, że Recursive wywołuje się liczbę razy pojawiającą się w rejestrze ax Przy każdym wywołaniu, Recursive zmniejsza rejestr ax o jeden i wywołuje się znowu. Ewentualnie Recursive zmniejsza ax do zera i powraca. Jedno co się wydarzy, to CPU wykona łańcuch instrukcji ret aż do przekazania sterowania do oryginalnego wywołania Recursive.

Dotychczas jednakże nie było rzeczywistej potrzeby dla rekurencji. W końcu możemy wydajnie zakodować tą procedurę jak poniżej pokazano:

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

Oba przykłady będą powtarzały treść procedury ilość razy przekazaną w rejestrze ax. Okazuje się, że jest kilka algorytmów rekurencji, których nie można zaimplementować w trybie iteracyjnym. Jednakże wiele implementacji algorytmów rekurencyjnych jest bardziej wydajnych niż ich iteracyjne odpowiedniki i o wiele bardziej postać algorytmu rekurencyjnego jest dużo łatwiejszy do zrozumienia.

Algorytm szybkiego sortowania jest chyba najbardziej znanym algorytmem, który prawie zawsze pojawia się w formie rekurencyjnej. Pascalowska implementacja tego algorytmu:

```
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;
```

Podprogram sort jest podprogramem rekurencyjnym. Rekurencja pojawia się przy ostatnich dwóch instrukcjach if w procedurze sort.
W języku asemblera , podprogram sort wygląda podobnie jak to:

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

; Main program to test sorting routine

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

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

                ExitPgm                         ;Return to DOS
Main            endp

; Data to be sorted

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

; procedure sort (l,r:integer)
; Sorts array A between indices l and 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                   ;Make room for i and j.

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

```
                mov     j, bx

; Note: This computation of the address of a[(l+r) div 2] is kind
; of strange. Rather than divide by two, then multiply by two
; (since A is a word array), this code simply clears the L.O. bit
; of BX.

                add     bx, 1                   ;Middle := a[(l+r) div 2]
                and     bx, 0FFFEh
                mov     ax, a[bx]               ;BX*2, because this is a word
;                                               ; array,nullifies the "div 2"
;                                               ; above.
;
; Repeat until i > j: Of course, I and J are in BX and SI.

                lea     bx, a                   ;Compute the address of a[i]
                add     bx, i                   ; and leave it in BX.
                add     bx, i

                lea     si, a                   ;Compute the address of a[j]
                add     si, j                   ; and leave it in SI.
                add     si, j

RptLp:

; While (a [i] < Middle) do i := i + 1;

                sub     bx, 2                   ;We'll increment it real
soon.
WhlLp1:         add     bx, 2
                cmp     ax, [bx]                ;AX still contains middle
                jg      WhlLp1

; While (Middle < a[j]) do j := j-1




                add     si, 2                   ;We'll decrement it in loop
WhlLp2:         add     si, 2
                cmp     ax, [si]                ;AX still contains middle
                jl      WhlLp2                  ; value.
                cmp     bx, si
                jnle    SkipIf

; Swap, if necessary

                mov     dx, [bx]
                xchg    dx, [si]
                xchg    dx, [bx]

                add     bx, 2                   ;Bump by two (integer values)
                sub     si, 2

SkipIf:         cmp     bx, si
                jng     RptLp

; Convert SI and BX back to I and J

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

; Now for the recursive part:

                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
```

Inaczej niż w podstawowej optymalizacji, (przechowanie kilku zmiennych w rejestrach) kod ten jest zawsze tłumaczeniem kodu pascalowskiego. Zauważmy ,że zmienne lokalne i i j nie są konieczne w tym kodzie asemblerowym (możemy użyć rejestrów do przechowania ich wartości) Ich zastosowanie demonstruje alokację zmiennych lokalnych na stosie.

Jest jedna ważna rzecz jaką musimy zapamiętać kiedy używamy rekurencji – podprogramy rekurencyjne mogą „zjadać" znaczną przestrzeń stosu. Dlatego też, kiedy piszemy podprogram rekurencyjny, zawsze alokujmy dostateczną pamięć w naszym segmencie stosu. Powyższy przykład ma niezwykle anemiczną 512 bajtową przestrzeń stosu, jednak, sortuje on tylko 32 liczby dlatego też 512 bajtów stosu jest wystarczające. Ogólnie nie będziemy znali głębokości naszej rekurencji więc alokowanie dużego bloku pamięci na stos może być właściwe.

Jest kilka wydajnych czynników, które nakładają się na procedury rekurencyjne. Na przykład, drugie (rekurencyjne) wywołanie sort w powyższym kodzie asemblera nie musi być wywołaniem rekurencyjnym. Poprzez ustawienie pary zmiennych i rejestrów, prosta instrukcja jmp może zastąpić odkładanie i wywołanie rekurencji. Poprawi to wydajność podprogramu szybkiego sortowania (rzeczywiście trochę) i zredukuje ilość pamięci wymaganą przez stos. Dobra książka o algorytmach, tak jak „The Art. Of Computer Programming, Tom III" D.E. Knutha jest doskonałym dodatkowym źródłem o szybkim sortowaniu .Inne teksty na temat złożonych algorytmów, teorii rekurencji i algorytmów będą dobrym miejscem do szukania pomysłów na wydajne implementowanie algorytmów rekurencji.

11.13 PODSUMOWANIE

W programie języka asemblera, wszystko co musimy zrobić to wykorzystać instrukcje call i ret dla implementacji procedur i funkcji. Rozdział Siódmy omawiał podstawowe zastosowanie procedur w programie asemblerowym 80x86; ten rozdział omówił jak zorganizować program podobnie do procedur i funkcji, jak przekazać parametry, zaalokować i uzyskać dostęp do zmiennych lokalnych i powiązane z tym tematy.

Rozdział ten zaczął się od spojrzenia co to jest procedura, jak zaimplementować procedurę z MASMem i różnic pomiędzy bliskimi i dalekimi procedurami w 80x86. Po szczegóły zajrzyj do poniższych sekcji:

*Procedury
*Bliskie i Dalekie Procedury
*Wymuszanie bliskich i dalekich wywołań i powroty
*Procedury zagnieżdżone

Funkcje są bardzo ważnymi konstrukcjami w językach wysokiego poziomu takich jak Pascal. Jednakże, nie ma rzeczywistych różnic pomiędzy funkcją a procedurą w programie asemblerowym. Logicznie, funkcja zwraca wynik a procedura nie; ale deklarujemy i wywołujemy procedury i funkcje identycznie w programie asemblerowym
Zobacz:

*Funkcje

Procedury i funkcje często tworzą „efekt uboczny". To znaczy, modyfikują one wartości rejestrów i nie-lokalnych zmiennych. Często, te efekty uboczne są niepożądane. Na przykład procedura może modyfikować rejestr , który kod wywołujący potrzebował zachować. Są dwa podstawowe mechanizmy dla zachowania takich wartości: zachowanie przez kod wywołany i zachowanie przez kod wywołujący. Po szczegóły dotyczące tych schematów zachowania i innych ważnych kwestii zobacz

*Zachowanie stanów maszynowych
*Efekty uboczne

Jedną ze znaczących korzyści stosowania proceduralnych języków takich jak Pascal lub C++ jest to, że możemy łatwo przekazać parametry do i z procedury i funkcji. Chociaż jest to trochę więcej pracy, możemy również przekazać parametry do naszych asemblerowych funkcji i procedur. Ten rozdział omówił jak i gdzie przekazać parametry. Omówił również jak uzyskać dostęp do parametrów wewnątrz procedury i funkcji. Poczytamy o tym w sekcjach:

*Parametry
*Przekazywanie przez wartość
*Przekazywanie przez referencję

*Przekazywanie przez wartość-Powrót
*Przekazywanie przez nazwę
*Przekazywanie przez leniwe wartościowanie
*Przekazywanie parametrów do rejestrów
*Przekazywanie parametrów do zmiennych globalnych
*Przekazywanie parametrów na stos
*Przekazywanie parametrów w strumieniu kodu
*Przekazywanie parametrów przez blok parametrów

Ponieważ język asemblera w rzeczywistości nie wspiera zapisu funkcji jako takiego, implementacja funkcji składa się z napisania procedury z parametrem zwracanym. Jako takie, wynik funkcji są całkiem podobne do parametrów pod wieloma względami.

*Wyniki funkcji
*Zwracanie wyniku funkcji do rejestru
* Zwracanie wyniku funkcji na stos
* Zwracanie wyniku funkcji do komórek pamięci

Większość HLLi dostarcza zmiennej lokalnej pamięci powiązanej z aktywacją i deaktywacją procedury lub funkcji. Chociaż kilka programów asemblerowych stosuje lokalne zmienne w identyczny sposób, łatwo jest zaimplementować dynamiczną alokacje zmiennych lokalnych na stosie

*Zmienne lokalnej pamięci

Rekurencja jest innym HLLowym udogodnieniem, który jest bardzo łatwy do implementacji w asemblerze. Rozdział ten omawia techniki rekurencji i przedstawia prosty przykład użycia algorytmu Szybkiego sortowania

---

11.14 PYTANIA

1) Wyjaśnij jak działają instrukcje CALL i RET.
2) Jakie są operandy dla dyrektywy asemblerowej PROC? Jaka jest jej funkcja?
3) Przepisz poniższy kod stosując PROC i ENDP

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

4) Zmodyfikuj swoją odpowiedź pytania 3 tak aby wszystkie wymagane rejestry były przechowane przez procedurę FillMem
5) Co się wydarzy jeśli opuścimy instrukcję przekazania sterowani (taką jak JMP lub RET) bezpośrednio przed dyrektywą ENDP w procedurze?
6) Jak asembler określa czy CALL jest bliski czy daleki? Jak określa czy instrukcja RET jest bliska czy daleka?
7) Jak możemy zastąpić domyślną decyzję asemblera czy zastosować bliski czy daleki CALL lub RET?
8) Czy zawsze musimy zagnieżdżać procedury w programie asemblerowym? Jeśli tak daj przykład.
9) Daj przykład dlaczego możemy chcieć zagnieździć segment w procedurze.
10) Jaka jest różnica pomiędzy funkcją a procedurą ?
11) Dlaczego podprogramy powinny przechowywać rejestry które modyfikują?
12) Jakie są zalety i wady wartości przechowywanych przez kod wywołujący a wartości przechowywanych przez kod wywoływany?
13) Co to są parametry?
14) Jak działają poniższe mechanizmy przekazywania parametrów
    a) przekazanie przez wartość
    b) przekazanie przez referencję
    c) przekazanie przez wartość – powrót
    d) przekazanie przez nazwę
15) Gdzie jest najlepsze miejsce do przekazania parametrów do procedury?
16) Wypisz pięć lokacji/ metod dla przekazania parametrów z lub do procedury
17) Jakie są parametry, które są przekazane na stos, dostępne wewnątrz procedury?
18) Jaki jest najlepszy sposób dealokacji parametrów przekazanych na stos, kiedy procedura kończy wykonywanie?

19) Podaj definicję poniższej pascalowskiej procedury:
    procedure PascalProc(i,j,k:integer)
20) Powtórz pytanie 19 zakładając, że procedura jest daleką procedurą
21) Jak wygląda stos podczas wykonywania procedury z pytania 19 ? Pytania 20?
22) Jak procedury w asemblerze uzyskują dostęp do parametrów przekazanych w strumieniu kodu?
23) Jak 80x86 przeskakuje parametry przekazane w strumieniu kodu i kontynuuje wykonywanie programu poza nimi kiedy procedura wraca do kodu wywołującego?
24) Jakie są zalety przekazywania parametrów przez blok parametrów?
25) Gdzie typowo są zwracane wyniki funkcji?
26) Co to jest efekt uboczny?
27) Gdzie typowo są alokowane zmienne lokalne (czasowe)?
28) Jak zaalokować lokalną (czasową) zmienną wewnątrz procedury?
29) Zakładając, że mamy trzy parametry przekazane przez wartość na stos i cztery różne lokalne zmienne, jak wygląda rekord aktywacji po zaalokowaniu zmiennych lokalnych (zakładamy bliską procedurę i żadnych rejestrów innych niż BP odłożonych na stos)
30) Co to jest rekurencja?