&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

ROZDZIAŁ DZIESIĄTY: **STRUKTURY STERUJĄCE**

Rozdział ten omawia dwa podstawowe typy struktur sterujących : decyzje i powtórzenia (iteracje).Omawia jak skonwertować instrukcje języka wysokiego poziomu takie jak if...then..else, case (switch), while, for itp do odpowiedników sekwencji języka asemblera. rozdział ten również omawia techniki jakie możemy zastosować do poprawienia osiągów tych struktur sterujących. Sekcje poniżej, które mają przedrostek „•" są niezbędne .Te sekcje, z „⊗" omawiają zaawansowane tematy, które może odłożymy na później.

• Wprowadzenie do Decyzji
• Sekwencje IF..THEN..ELSE
• Instrukcje CASE
⊗ Stany maszynowe i skoki pośrednie
• Kod spaghetti
• Pętle
• Pętle WHILE
• Pętle REPEAT..UNTIL
• LOOP..ENDLOOP
• Pętle FOR
• Wykorzystanie rejestru a pętle
⊗ Poprawianie wydajności
⊗ Przenoszenie warunku zakończenia na koniec pętli
⊗ Wykonywanie pętli wstecznych
⊗ Pętla niezmienników
⊗ Pętle rozwijane
⊗ Indukcja zmiennych

10.1 WPROWADZENIE DO DECYZJI

W swoje podstawowej formie, decyzja jest rodzajem krótkiego skoku wewnątrz kodu ,który przełącza pomiędzy dwoma możliwymi wykonaniami ścieżkami wykonania opartymi na jakimś warunku. Normalnie,(chociaż nie zawsze),sekwencje instrukcji warunkowych są implementowane z instrukcjami skoków warunkowych. Instrukcje warunkowe odpowiadają instrukcjom w Pascalu;

IF (warunek jest prawdziwy) THEN instr1 else instr2;

Język asemblera, jak zwykle, oferuje dużo większą elastyczność, kiedy zajmujemy się instrukcjami warunkowymi. Rozważmy poniższą instrukcję pascalowską:

IF ((X<Y) and (Z>T)) or (A <>B) THEN instr1;

Podchodząc do konwertowania tej instrukcji na asembler metodą „brute force" możemy stworzyć:

```
                mov     cl, 1           ;Assume true
                mov     ax, X
                cmp     ax, Y
                jl      IsTrue
                mov     cl, 0           ;This one's false
IsTrue:         mov     ax, Z
                cmp     ax, T
                jg      AndTrue
                mov     cl, 0           ;It's false now
AndTrue:        mov     al, A
                cmp     al, B
                je      OrFalse
                mov     cl, 1           ;Its true if A <> B
OrFalse:        cmp     cl, 1
                jne     SkipStmt1
        <Code for stmt1 goes here>
SkipStmt1:
```

Jak możemy zobaczyć ,w przetwarzaniu tego powyższego wyrażenia bierze udział znaczna liczba instrukcji warunkowych. Pobieżnie odpowiada to (odpowiednikowi) Pascalowych instrukcji:

```
cl := true;
IF (X>=Y) then cl := fałsz;
IF (Z<=T) then cl := fałsz;
IF (A<>B) then   cl := true;
IF (CL = true) then stmt1;
```

Teraz porównajmy to z poniższym „poprawionym" kodem:

```
        mov     ax, A
        cmp     ax, B
        jne     DoStmt
        mov     ax, X
        cmp     ax, Y
        jnl     SkipStmt
        mov     ax, Z
        cmp     ax, T
        jng     SkipStmt
DoStmt:
        <tu umieszczamy kod dla Stmt1>
SkipStmt:
```

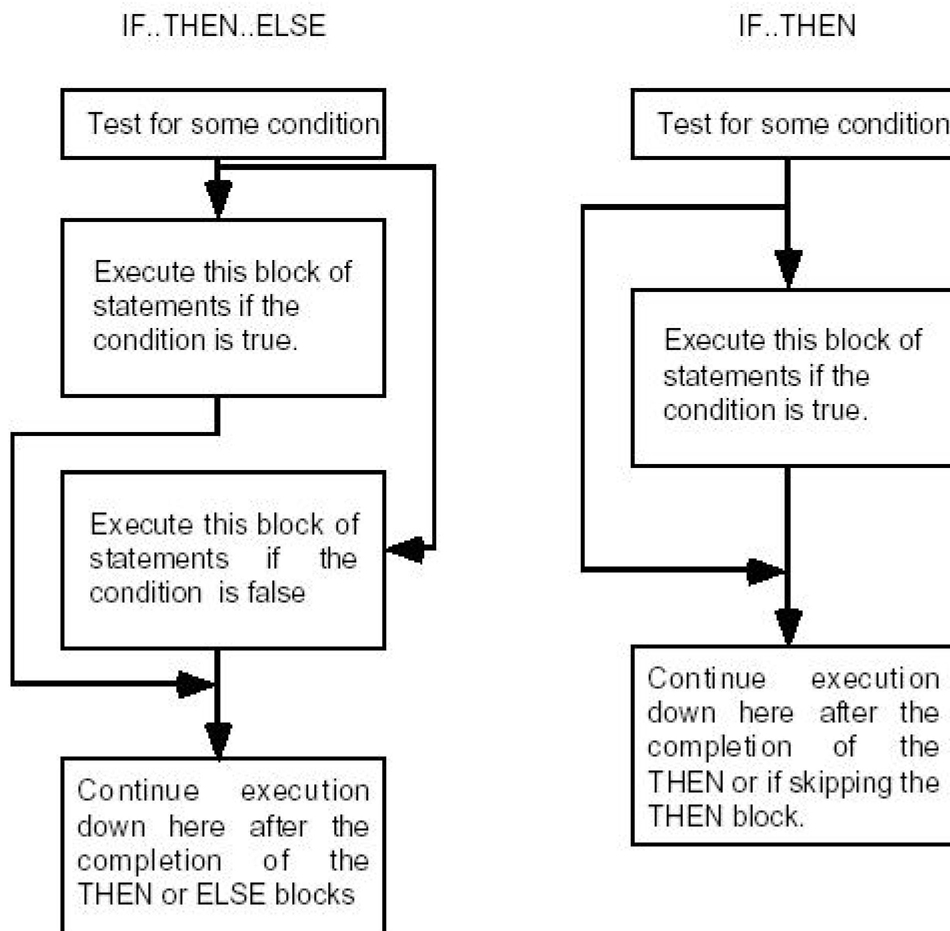
Dwie rzeczy powinny być oczywiste z powyższej sekwencji kodu: po pierwsze, pojedyncza instrukcja warunkowa w Pascalu może wymagać kilku skoków warunkowych w asemblerze; po drugie, organizacja złożonego wyrażenia w sekwencji warunkowej może wpływać na wydajność kodu. Dlatego też ,powinniśmy ostrożnie ćwiczyć kiedy zajmujemy się sekwencjami warunkowymi w asemblerze.

Instrukcje warunkowe mogą być podzielone na trzy podstawowe kategorie: instrukcje if..then..else ,instrukcje case i skoki pośrednie. Poniższa sekcja opisze te struktury programu ,jak je zastosować i jak napisać je w języku asemblera.

10.2 SEKWENCJE IF..THEN..ELSE

Najpowszechniejszym zastosowaniem instrukcji warunkowych jest instrukcja if..then lub if..then..else. Te dwie instrukcje przybierają formę jak pokazano na rysunku 10.1

Instrukcja if..then jest specjalnym przypadkiem instrukcji if..then..else (z pustym blokiem ELSE).Dlatego też, będziemy rozpatrywać bardziej ogólną postać if..then..else. Podstawowa implementacja instrukcji if..then..else w języku asemblera 80x86 wygląda podobnie jak to:



Rysunek 10.1 Instrukcje IF..THEN i IF..THEN..ELSE

```
        {sekwencja instrukcji testujących jakiś warunek}
                jcc     JakiśKod
            {sekwencja instrukcji odpowiadających blokowi THEN}
                jmp     EndOfIf
ElseCode:
            {sekwencja instrukcji odpowiadających blokowi ELSE}
EndOfIf:
```

Notka: Jcc przedstawia jakąś instrukcję skoku warunkowego.

Na przykład do konwersji instrukcji pascalowskie:

IF (a=b) then c:=d else b:= b+1;

Na język asemblera, możemy zastosować poniższy kod 80x86:

```
            mov     ax, a
            cmp     ax, b
            jne     ElseBlk
            mov     ax, d
            mov     c, ax
            jmp     EndOfIf
ElseBlk:
            inc     b
EndOf If:
```

Dla prostego wyrażenia takiego jak (A=B) generowanie właściwego kodu dla instrukcji if..then..else jest prawie trywialne. Kiedy wyrażenie staje się bardziej złożone, również wzrasta złożoność powiązanego kodu asemblerowego. Rozważmy poniższą instrukcję if prezentowaną wcześniej:

```
IF ((X > Y) and (Z <T)) or (A<>B) THEN C :=D;
```

Kiedy przetwarzamy złożone instrukcje if takie jak to, zadanie konwersji stanie się łatwiejsze jeśli podzielimy tą instrukcję if na sekwencję trzech różnych instrukcji if jak następuje:

```
IF  (A<>B) THEN C :=D
IF (X >Y) THEN IF (Z< T) THEN C := D;
```

Konwersja ta pochodzi z Pascalowskich odpowiedników:

```
IF (wyraż1 AND wyarż2) THEN instr;
```

Jest odpowiednikiem

```
IF (wyraż1) THEN IF (wyraż2) THEN  instr;
```

a

```
IF (wyraż1 OR wyraż2) THEN instr;
```

jest odpowiednikiem

```
IF (wyraż1) THEN instr;
IF (wyraż2) THEN instr;
```

W języku asemblera, dawna instrukcja if staje się :

```
        mov     ax, A
        cmp     ax, B
        jne     DOIf
        mov     ax, X
        cmp     ax, Y
        jng     EndOfIf
        mov     ax, Z
        cmp     ax, T
        jnl     RndOfIf
DoIf:
        mov     ax, D
        mov     C, ax
EndOfIf:
```

Jak już prawdopodobnie mówiliśmy, kod konieczny do sprawdzenia warunku może łatwo stać się bardziej złożony niż instrukcje pojawiające się w blokach else i then. Chociaż wydaje się to nieco paradoksalne, że można włożyć większy wysiłek w testowanie warunku niż działanie na wyniku tego warunku, zdarza się to cały czas .Dlatego też powinniśmy być przygotowani na taką sytuację.

Prawdopodobnie największym problem z implementacją złożonych instrukcji warunkowych w asemblerze jest próbowanie obliczenia co zrobimy po napisaniu kodu. Największą zaletą oferowaną przez język wysokiego poziomu nad asemblerem jest to że wyrażenia są dużo łatwiejsze do odczytu i pojęcia w języku wysokiego poziomu. Wersja HLL'a jest samodokumentująca podczas gdy język asemblera ma tendencje do ukrywania prawdziwej natury kodu. Dlatego też dobrze napisane komentarze są niezbędnym składnikiem implementacji przez asembler instrukcji if..then..else. Elegancka implementacja powyższego przykładu:

```
;IF  ((X >Y) AND (Z < T)) OR (A<> B) THEN C:=D;
;Implementujemy jako:
;IF (A<>B) THEN GOTO DoIf;
        mov     ax, A
        cmp     ax, B
        jne     DoIf
;IF NOT (X>Y) THEN GOTO  EndOfIf;
        mov     ax, X
        cmp     ax, Y
        jng     EndOf If
;IF NOT (Z < T) THEN GOTO EndOfIf;
        mov     ax, Z
        cmp     ax,T
        jnl     EndOfIf
; Blok THEN:
        mov     ax, D
        mov     C, ax
```

```
;Koniec instrukcji IF
EndOfIf:
```

Trzeba przyznać, że takie przedstawianie jest popadaniem w przesadę dla tak prostego przykładu poniższe byłoby prawdopodobnie wystarczające:

```
;IF ((X > Y) AND (Z < T) OR (A<>B) THEN C:=D;
;test wyrażenia boolowskiego:
        mov     ax, A
        cmp     ax, B
        jne     DoIF
        mov     ax, X
        cmp     ax, Y
        jng     EndOfIf
        mov     ax, Z
        cmp     ax, T
        jnl     EndOfIf
;Blok THEN:
        mov     ax, D
        mov     C, ax
;Koniec instrukcji IF
EndOfIf:
```

Jednakże jeśli nasze instrukcje if stają się złożone, gęstość (i jakość) naszych komentarzy staje się coraz bardziej ważna.

10.3 INSTRUKCJE CASE

Pascalowa instrukcja case przybiera poniższą formę:

CASE zmienna OF
    $stała_1$: $instr_1$;
    $stała_2$:$instr_2$;
    -
    -
    -
    $stała_n$:$instr_n$
END;

Kiedy wykonuje się ta instrukcja, sprawdza wartość stałej $const_1$..$const_n$.Jeśli została znaleziona, wtedy wykonuje się odpowiednia instrukcja. standardowy Pascal umieszcza kilka ograniczeń na instrukcję case. Po pierwsze, wartość zmiennej nie jest na liście stałych, wynik instrukcji case jest niezdefiniowany. Po drugie, wszystkie stałe pojawiające się po etykiecie CASE muszą być unikalne. Powód tych ograniczeń stanie się jasny za chwilę.

Większość wstępnych tekstów o programowaniu wprowadza instrukcję case poprzez wyjaśnienie jej jako sekwencji instrukcji if..then..else. Można twierdzić że poniższe dwa kawałki kodu pascalowego są sobie odpowiednie:

```
CASE I OF
        0: Writeln('I=0');
        1:Writeln('I=1');
        2:Writeln('I=2');
END;

IF I = 0 THEN writeln ('I=0')
ELSE IF I = 1 THEN Writeln ('I= 1')
ELSE IF I= 2 THEN Writeln ('I=2');
```

Podczas gdy semantycznie te dwie części kodu mogą być takie same, ich implementacja jest zwykle różna. Podczas gdy łańcuch if..then..else wykonuje porównanie dla każdej instrukcji warunkowej w sekwencji, instrukcja case normalnie używa skoku pośredniego dla sterowania przesyłaniem danych do jednej z kilku instrukcji pojedynczego obliczenia. Rozważmy dwa przedstawione powyżej przykłady ,które mogą być napisane w asemblerze w poniższym kodzie:

```
                mov     bx, I
                shl     bx, 1         ;Multiply BX by two
                jmp     cs:JmpTbl[bx]

JmpTbl          word    stmt0, stmt1, stmt2

Stmt0:          print
                byte    "I=0",cr,lf,0
                jmp     EndCase

Stmt1:          print
                byte    "I=1",cr,lf,0
                jmp     EndCase

Stmt2:          print
                byte    "I=2",cr,lf,0

EndCase:

; IF..THEN..ELSE form:

                mov     ax, I
                cmp     ax, 0
                jne     Not0
                print
                byte    "I=0",cr,lf,0
                jmp     EndOfIF

Not0:           cmp     ax, 1
                jne     Not1
                print
                byte    "I=1",cr,lf,0
                jmp     EndOfIF

Not1:           cmp     ax, 2
                jne     EndOfIF
                Print
                byte    "I=2",cr,lf,0
EndOfIF:
```

Dwie rzeczy powinny stać się łatwo zauważalne: im więcej (kolejnych) case'ów mamy, tym bardziej wydajna staje się implementacja tablicy skoków (pod względem przestrzeni i szybkości).z wyjątkiem banalnych przypadków ,instrukcja case jest prawie zawsze szybsza i zwykle z dużym marginesem. Tak długo jak etykiety case są kolejnymi wartościami, wersja instrukcji case jest zazwyczaj również mniejsza.

Co stanie się jeśli musimy zawrzeć nie po kolei etykiety case lub nie możemy być pewni, czy zmienna case nie wyjdzie poza zakres? Wiele Pascali ma rozszerzoną definicję instrukcji case zawierającą klauzulę otherwise .Taka instrukcja case przyjmuje postać:

```
CASE zmienna  OF
        stała:instr;
        stała:instr;
        •                               -
        •                               -
        •                               -
        stała:instr;
        OTHERWISE instr
END;
```

Jeśli zwartość zmiennej zgadza się z jedną ze stałych stanowiącą etykietę case, wtedy jest wykonywana powiązana instrukcja .Jeśli wartość zmiennej nie zgadza się z żadną z etykiet case wtedy wykonywana jest instrukcja

po klauzuli otherwise. Klauzula otherwise jest implementowana w dwóch fazach .najpierw musimy wybrać wartości minimalną i maksymalną, która pojawia się w instrukcji case. W poniższej instrukcji case ,najmniejsza etykieta case to pięć a największa 15:

```
CASE I OF
        5:instr1;
        8:instr2;
        10:instr3;
        12:instr4;
        15:instr5;
        OTHERWISE instr6
END;
```

Przed wykonaniem skoku przez tablicę skoków,80x86 implementuje tą instrukcję case , sprawdzając zmienną case aby być pewnym ,że jest w zakresie 5..15.Jeśli nie, sterowanie powinno być przekazane do instr6:

```
mov     bx, I
cmp     bx, 5
jl      Otherwise
cmp     bx, 15
jg      Otherwise
shl     bx, 1
jmp     cs:JmpTbl-10[bx]
```

Jedyny problem z tą formą instrukcji case jaki występuje, jest taki, że niepoprawnie działa w sytuacji gdzie I jest równe 6,7,9,11,13 lub 14.Zamiast dorzucić ekstra kod przed skok warunkowy, możemy dołączyć ekstra wejście w tablicy skoków jak następuje:

```
                mov     bx, I
                cmp     bx, 5
                jl      Otherwise
                cmp     bx, 15
                jg      Otherwise
                shl     bx, 1
                jmp     cs:JmpTbl-10[bx]
Otherwise:      {wkładamy instr6 tutaj}

JmpTbl          word    instr1, Otherwise,Otherwise,instr2,Otherwise
                word    instr3,Otherwise,instr4,otherwise,otherwise
                word    instr5
                itp.
```

Zauważmy ,że wartość 10 jest odejmowana od adresu tablicy skoków. Pierwsze wejście w tablicy ma zawsze offset zero podczas gdy najmniejsza wartość stosowana do indeksowania tej tablicy to pięć (która jest mnożona przez dwa do stworzenia 10).Wejścia dla 6,7,89,11,13 i 14 wszystkie wskazują na kod klauzuli Otherwise, więc jeśli te wartości zawierają jeden, będzie wykonywana klauzula Otherwise.

Jest problem z tą implementacją instrukcji case. Jeśli etykieta case zawiera nie-kolejne wejścia, które są bardziej przestrzenne, poniższa instrukcja case wygeneruje niezmiernie duży kod:

```
CASE I OF
        0:instr1;
        100:instr2;
        1000:instr3;
        10000:instr4
        Otherwise instr5
END;
```

W takiej sytuacji nasz program będzie dużo mniejszy, jeśli zaimplementujemy instrukcję case sekwencją instrukcji if zamiast stosując instrukcje skoku. Jednak, zapamiętajmy jedną rzecz - wielkość tablicy skoków normalnie nie wpływa na szybkość wykonywania programu .Jeśli tablica skoków zawiera dwa wejścia lub dwa tysiące, instrukcja case wykona wielokrotny skok stałą ilość razy. Implementacja instrukcji if wymaga rosnącej

liniowo ilości czasu dla każdej etykiety case pojawiającej się w instrukcji case.

Prawdopodobnie, największą zaletą stosowania języka asemblera nad HLL'ami takimi jak Pascal jest to, że możemy wybrać rzeczywistą implementację. W takim przypadku, możemy zaimplementować instrukcję case jako sekwencję instrukcji if..then..else .lub możemy zaimplementować ją jako tablicę skoków lub zastosujemy ich skrzyżowanie ich dwóch:

```
CASE  I  OF
        0:instr1;
        1:instr2;
        2:instr3;
        100:instr4;
        Otherwise  instr5
END;
```

może stać się:

```
mov     bx, I
cmp     bx, 100
je      Is100
cmp     bx, 2
ja      Otherwise
shl     bx, 1
jmp     cs:JmpTbl[bx]
itd.
```

Oczywiście, możemy zrobić to w Pascalu, w poniższym kodzie:

```
IF I = 100 then   instr4
ELSE   CASE   I   OF
            0:instr1;
            1:instr2;
            2:instr3;
            otherwise  instr5
END;
```

Ale powoduje to zakłócenie czytelności programu pascalowskiego. Z drugiej strony, dodatkowy kod do testowania dla 100 w kodzie asemblera niekorzystnie wpływa na czytelność programu (być może dlatego ,że już jest trudny do odczytu) Dlatego też ,większość ludzi doda dodatkowy kod aby uczynić swój program bardziej wydajnym.

Instrukcja switch C/C++ jest bardzo podobna do pascalowskiej instrukcji case. Jest tylko jedna semantyczna różnica; programista musi wyraźnie umieścić instrukcję break w każdej klauzuli dla przekazania sterowania do pierwszej instrukcji za switch. To break odpowiada instrukcji jmp na końcu każdej sekwencji case w powyższym kodzie asemblera .Jeśli odpowiednie break nie jest obecny ,C/C++ przekazuje sterowanie do kodu następującego po case .Jest to odpowiednik pozostawienia jmp na końcu sekwencji case:

```
switch  (1)
{
case 0:   instr1;
case 1:   instr2;
case 2:   instr3;
          break;
case 3:   instr4;
          break;
default:  instr5;
}
```

Tłumaczymy to na kod 80x86:

```
                mov     bx, 1
                cmp     bx, 3
                ja      DefaultCase

                shl     bx, 1
                jmp     cs:JmpTbl[bx]
JmpTbl[bx]      word    case0, case1, case2, case3
        case0:  <kod instr1>
```

```
                case1:  <kod instr2>
                case2:  <kod instr3>
                        jmp     EndCase             ;emitowana dla instrukcji break
                case3:  <kod instr4>
                        jmp     EndCase             ;emitowany dla instrukcji break
DefalutCase:            < kod instr5>
EndCase:
```

10.4 STANY MASZYNOWE I SKOKI POŚREDNIE

Inną strukturą sterująca powszechnie stosowaną w programach asemblerowych jest „stan maszynowy". Stan maszynowy używa zmienną stanu do sterowania przebiegiem programu. Język FORTRAN dostarcza tej możliwości z powiązaną instrukcją goto. Pewne warianty C (np. GNU's GCC z Free Software Foundation) dostarczają podobnych cech. W języku asemblera, skok pośredni dostarcza mechanizmu do łatwej implementacji stanów maszynowych

Więc co to jest ten stan maszynowy? W bardzo podstawowej terminologii ,jest to część kodu który śledzi historię wykonania poprzez wprowadzanie i pozostawiania pewnych „stanów". Dla potrzeb tego rozdziału ,nie będziemy stosować bardzo formalnej definicji stanu maszynowego. Założymy ,że stan maszynowy jest kawałkiem kodu który (jakoś) zapamiętuje historię wykonywania (stan) i wykonuje część kodu opartego na tej historii.

W sensie rzeczywistym, wszystkie programy są stanami masynowymi. Rejestry CPU i wartości w pamięci stanowią „stan" tej maszyny. Jednakże, my zastosujemy dużo bardziej wymuszony pogląd. Istotnie, dla większości potrzeb tylko pojedyncza zmienna (lub wartość w rejestrze IP) będzie oznaczała bieżący stan.

Teraz rozpatrzymy konkretny przykład. Przypuśćmy ,że mamy procedurę, która wykonuje jedną operacje pierwszy raz kiedy ją wywołujemy, różne operacje za drugim wywołaniem, jeszcze coś innego za trzecim wywołaniem a potem znowu coś nowego za czwartym wywołaniem. Po czwartym wywołaniu powtarza te cztery różne operacje w kolejności. Na przykład przypuśćmy że chcemy aby procedura dodała ax i bx za pierwszym razem, odjęła je za drugim wywołaniem, pomnożyła je za trzecim i podzieliła za czwartym.

Możemy zaimplementować tą procedurę jak następuje:

```
State           byte    0
StateMatch      proc
                cmp     stan, 0
                jne     TryState1
;jeśli jest to stan 0, dodaje BX do AX i przełącza do stanu 1:
                add     ax, bx
                inc     State                   ;ustawia go na stan 1
                ret
;Jeśli jest to stan 1, odejmuje BX od AX i przełącza na stan 2
TryState1:      cmp     Stan, 1
                jne     TryState2
                sub     ax, bx
                inc     State
                ret
;jeśli jest to stan 2, mnoży Ax i BX i przełącza do stanu 3:
TryState2:      cmp     Stan, 2
                jne     MustBeState3
                push    dx
                mul     bx
                pop     dx
                inc     State
                ret
;jeśli żaden z powyższych, zakładamy, że to stan 4.Więc dzielimy AX przez BX
MustBeState3:   push    dx
                xor     dx, dx                  ;powielenie zera AX do DX
                div     bx
                pop     dx
                mov     State, 0                ;przełączenie z powrotem do stanu 0
                ret
```

StateMatch endp

Technicznie procedura ta nie jest stanem maszyny. Zamiast tego, jest zmienna State i instrukcje cmp/jne, które stanowią stan maszyny.

Nie ma niczego specjalnego w tym kodzie .Jest trochę bardziej niż instrukcja case implementowany przez konstrukcję if..then..else. Jedyna specjalna rzecz w tej procedurze jest to, że zapamiętuje ile razy została wywołana i zachowuje się różnie w zależności od liczby wywołania. Chociaż jest to poprawna implementacja żądanego stanu maszynowego, nie jest ona szczególnie wydajna. Bardziej powszechną implementacją stanu maszynowego w asemblerze jest zastosowanie skoku pośredniego .Zamiast stosować zmienną stanu zawierającą wartości takie jak zero, jeden, dwa lub trzy, możemy załadować zmienną stanu adresem kodu do wykonania na wejście do procedury. Poprzez prosty skok do tego adresu, stan maszynowy może zapisać powyższe testy potrzebne do właściwego wykonania fragmentu kodu. Rozważmy poniższą implementację stosującą skok pośredni:

```
State           word    State0
StateMach       proc
                jmp     State

; If this is state 0, add BX to AX and switch to state 1:

State0:         add     ax, bx
                mov     State, offset State1            ;Set it to state 1
                ret

; If this is state 1, subtract BX from AX and switch to state 2

State1:         sub     ax, bx
                mov     State, offset State2            ;Switch to State 2
                ret

; If this is state 2, multiply AX and BX and switch to state 3:

State2:         push    dx
                mul     bx
                pop     dx
                mov     State, offset State3            ;Switch to State 3
                ret

; If in State 3, do the division and switch back to State 0:

State3:         push    dx
                xor     dx, dx          ;Zero extend AX into DX.
                div     bx
                pop     dx
                mov     State, offset State0            ;Switch to State 0
                ret
StateMach       endp
```

Instrukcja jmp na początku procedury StateMatch przekazuje sterowanie do lokacji wskazywanej przez zmienna State. Przy pierwszym wywołaniu StateMAtch wskazuje na etykietę State0.Od tego czasu każda podsekcja kodu ustawia zmienną State aby wskazywała właściwy następny kod.

10.5 KOD SPAGHETTI

Jednym z głównych problemów z językiem asemblera jest to, że zabiera on kilka instrukcji dla realizacji prostej idei ujętej przez pojedynczą instrukcję HLL'a. Zbyt często programista zauważa, że może zaoszczędzić kilka bajtów lub cykli poprzez skok do środka jakiejś struktury programowej. Po kilku takich obserwacjach (i odpowiednich modyfikacjach),kod zawiera całą sekwencję skoków do lub z części kodu. Jeśli namalujemy linię każdego skoku do jego przeznaczenia, listing końcowy będzie się kończył tak jak gdyby ktoś rzucił miską spaghetti w nasz kod., stąd termin „spaghetti kod"

Spaghetti kod cierpi z powodu jednej głównej wady - jest trudny (w najlepszym razie) do odczytania taki program i dojścia do tego co on robi. Dużo programów zaczynających się w postaci „strukturalnej" staje się kodem niestrukturalnym (spaghetti) na ołtarzu wydajności. Niestety kod spaghetti rzadko jest wydajny .Ponieważ, trudno jest obliczyć dokładnie, co się stanie, jest bardzo trudno określić czy możemy zastosować lepszy algorytm do poprawy systemu. W związku z tym kod spaghetti może okazać się mniej wydajny.

Chociaż jest prawdą ,że stworzenie jakiegoś kodu spaghetti w naszym programie może poprawić jego wydajność, powinniśmy zawsze robić to w ostateczności (kiedy wypróbowaliśmy wszystkie inne możliwości a jeszcze nie osiągnęliśmy tego co chcieliśmy),nigdy automatycznie. Zawsze zaczynajmy pisanie naszych programów z prostymi instrukcjami if i case .Oczywiście ,nigdy nie powinniśmy zacierać struktury naszego kodu chyba, że wynikające z tego korzyści są tego warte.

Znane powiedzenie z kręgu programowani strukturalnego mówi: „Po goto , wskaźniki są kolejnym, niebezpiecznym elementem w języku programowania" .Podobne powiedzenie mówi: „Wskaźniki są strukturą danych a goto'a są strukturami sterującymi". Innymi słowy, unikajmy nadmiernego stosowania wskaźników. Jeśli wskaźniki i goto'a są złe ,wtedy skok pośredni musi być najgorszą ze wszystkich ponieważ wymaga obu goto i wskaźników! Jednak serio ,instrukcji skoku pośredniego powinniśmy unikać poza sporadycznym zastosowaniem .Mogą one uczynić program trudniejszym do odczytania. W końcu, skok pośredni może (teoretycznie) przekazać sterowanie danymi do każdej etykiety wewnątrz programu. Wyobraźmy sobie jak trudno byłoby kontrolować przepływ w programie gdybyśmy nie mieli pojęcia co zawiera wskaźnik i napotkany skok pośredni stosujący ten wskaźnik .Dlatego też ,powinniśmy zawsze być ostrożni stosując instrukcje skoku pośredniego.

10.6 PĘTLE

Pętle przedstawiają końcowe podstawowe struktury sterujące (sekwencje, decyzje i pętle) ,które stanowią typowy program. Podobnie jak wiele innych struktur w asemblerze, znajdziemy zastosowanie pętli w miejscach o których nam się nie śniło, że można zastosować pętle. Większość HLLi implikuje strukturę pętli jako ukrytą. Na przykład, rozważmy instrukcję BASICa : IF A$ = B$ THEN 100. Ta instrukcja if porównuje dwa łańcuchy i skacze do instrukcji 100 jeśli są równe. W języku asemblera, będziemy musieli napisać pętlę dla porównania każdego znaku w A$ z odpowiednim znakiem w B$ a potem skok do instrukcji 100 jeśli i tylko jeśli wszystkie znaki się zgadzają. W BASICu nie jest widoczna żadna pętla w programie. W języku asemblera ta bardzo prosta instrukcja if wymaga pętli. Ten mały przykład pokazuje, że pętle wydają się pojawiać wszędzie.

Program z pętlami składa się z trzech składników: opcjonalny składnik inicjacyjny, test zakończenia pętli i ciała pętli. Porządek w jakim te składniki są asemblowane może radykalnie zmienić sposób działania pętli. Trzy kombinacje tych składników pojawiają się wielokrotnie. Z powodu ich częstotliwości, tym strukturom pętli zostały nadane specjalne nazwy w HLL'ach: pętle while, pętle repeat..until (do..while w C/C++) i pętle loop..endloop

---

10.6.1 PĘTLE WHILE

Najbardziej ogólną pętlą jest pętla while. Przybiera ona następującą postać:

WHILE wyrażenie boolowskie DO instrukcja;

Są dwa ważne punkty do odnotowania jeśli chodzi o pętlę while. Po pierwsze, test zakończenia pojawia się na początku pętli. Po drugie bezpośrednią konsekwencją umieszczenia testu zakończenia, ciało pętli może nigdy się nie wykonać. Jeśli warunek zakończenia zawsze istnieje, ciało pętli będzie zawsze przeskakiwane.
Rozważmy poniższą pascalowską pętlę loop:

```
I := 0;
WHILE (I <100) do I := I=1;
```

I:=0; jest kodem inicjującym dla tej pętli. I jest zmienną sterującą pętli ,ponieważ steruje wykonaniem ciała pętli (I<100) jest warunkiem zakończenia pętli To znaczy, pętla nie zakończy się tak długo dopóki I będzie mniejsze niż 100. I := I+1 jest ciałem pętli. To jest kod, który wykonuje się w każdym przebiegu pętli. Możemy skonwertować go do języka asemblera 80x86 jak następuje:

```
            mov     I, 0
WhileLp:    cmp     I, 100
            jge     WhileDone
            inc     I
            jmp     WhileLp
WhileDone:
```

Zauważmy ,że pascalowska pętla while może być łatwo zsyntetyzowana przez zastosowanie instrukcji if i goto. Na przykład powyższa Pascalowska pętla while może być zastąpioną przez:

```
I := 0;
```

```
1:      IF (I < 100) THEN BEGIN
            I := I + 1;
            GOTO 1;
        END;
```

Bardziej ogólnie, każda pętla while może być zbudowana według poniższego schematu:

```
        Opcjonalny kod inicjujący
1:      Jeśli warunek zakończenia nie spełniony THEN BEGIN
            ciało pętli
            GOTO 1;
        END;
```

Dlatego możemy użyć tej techniki dla wcześniejszych konwersji, w tym rozdziale, instrukcji if do języka asemblera. Wszystko co potrzebujemy to dodatkowa instrukcja jmp (goto)

10.6.2 PĘTLE REPEAT..UNTIL

Pętla repeat..until (do..while) testuje warunek zakończenia na końcu pętli zamiast na jej początku. W Pascalu, pętla repeat..until przybiera poniższą postać:

```
            Opcjonalny kod inicjujący
            REPEAT
                treść pętli
            UNTIL  warunek zakończenia
```

Sekwencja ta wykonuje kod inicjujący treść pętli, wtedy testuje jakiś warunek aby zobaczyć czy pętla powinna być powtórzona. Jeśli wyrażenie boolowskie przyjmuje wartość fałsz, pętla jest powtarzana, w przeciwnym razie kończy się. Dwiema rzeczami do zapamiętania o pętli repeat..until jest to, że test zakończenia pojawia się na końcu pętli i, bezpośrednia konsekwencja tego, treść pętli wykonuje się przynajmniej raz.

Podobnie jak pętla while, pętla repeat..until może być zsyntetyzowana instrukcją if i goto. Zastosowalibyśmy coś takiego:

```
            Kod inicjujący
1:              treść pętli
            IF  warunek zakończenia nie spełniony THEN GOTO
```

Opierając się na materiale przedstawionym w poprzedniej sekcji ,możemy łatwo zsyntetyzować pętlę repeat..until w języku asemblera.

10.6.3 PĘTLE LOOP..ENDLOOP

Jeśli pętle while testują warunek zakończenia na początku pętli, a pętle repeat..until sprawdzają ten warunek na końcu pętli, jedynym miejscem pozostawionym do testowania warunku zakończenia jest środek pętli. Chociaż Pascal i C/C++ bezpośrednio nie wspierają takiej pętli, struktura loop..endloop może być znajdowania w HLLach takich jak Ada. Pętla loop..endloop przyjmuje następującą postać:

```
            LOOP
                Treść pętli
            ENDLOOP;
```

Zauważmy że nie ma wyraźnego warunku zakończenia. Chyba że uwzględniono budowę loop..endloop jako prostej formy pętli nieskończonej. Warunek pętli jest obsługiwany przez instrukcje if i goto. Rozważmy poniższy (pseudo) pascalowski kod, który zawiera budowę loop..endloop:

```
                LOOP
                    READ (ch)
                    IF ch = '.' THEN BREAK;
                    WRITE(ch);
                ENDLOOP;
```

W rzeczywistym Pascalu, zastosowalibyśmy poniższy kod do wykonania tego:

```
1:
        READ(ch);
        IF ch = '.' THEN GOTO 2; (*Turbo Pascal wspiera BRAEK!*)
        WRITE(ch);
        GOTO 1
2:
```

W języku asemblera skończymy tak:

```
LOOP1: getc
         cmp    al., '.'
         je     EndLoop
         putc
         jmp    LOOP1
EndLoop:
```

10.6.4 PĘTLE FOR

Pętla for jest specjalną formą pętli while ,która powtarza treść pętli określoną ilość razy. W Pascalu pętla for wygląda podobnie jak ta:

FOR var := wartość początkowa TO wartość końcowa DO instrukcja

lub

FOR var := wartość początkowa DOWNTO wartość końcowa DO instrukcja

Tradycyjnie, w Pascalu pętla for bywa zastosowana do przetwarzania tablic i innych obiektów dostępnych w porządku sekwencji numerycznej. Pętle te mogą być konwertowane bezpośrednio do języka asemblera jak poniżej:

W Pascalu:

FOR var := star TO stop DO instrukcje;

W Asemblerze:

```
        mov     var, start
FL:     mov     ax, var
        cmp     ax, stop
        jg      EndFor
; kod odpowiadający instrukcji przychodzi tutaj
        inc     var
        jmp     FL
EndFor:
```

Na szczęście, większość pętli for powtarza jakieś instrukcje stałą ilość razy. Na przykład,

FOR I := 0 to 7 do write(ch);

W sytuacji takiej jak ta lepiej jest zastosować instrukcję loop 80x86 zamiast symulowania pętli for:

```
        mov     cx, 7
LP:     mov     al., ch
        call    putc
        loop    LP
```

Zapamiętajmy ,że normalnie instrukcja loop pojawia się na końcu pętli podczas gdy pętla for testuje warunek zakończenia na początku pętli. Dlatego też, powinniśmy się zabezpieczyć aby zapobiec pętli nieskończonej w przypadku gdy cx jest równe zero (kiedy będzie to przypadek pętli loop powtarzającej się 65,536 razy) lub gdy wartość zatrzymania jest mniejsza niż wartość początkowa. W przypadku:

FOR var := start TO stop DO instrukcja;

Zakładając ,że nie stosujemy wartości var wewnątrz pętli, prawdopodobnie będziemy chcieli zastosować kod asemblerowy:

```
            mov     cx, stop
            sub     cx, start
            jl      SkipFor
            inc     cx
LP:         instrukcja
            Loop    LP
SkipFor:
```

Pamiętamy, że instrukcje sub i cmp ustawiają flagi i identyczny sposób. Dlatego też, pętla ta będzie pominięta jeśli stop będzie mniejsze niż start. Będzie ona powtarzana (stop-start)+1 raz w przeciwnym razie. Jeśli musimy odnieść się do wartości var wewnątrz pętli, możemy użyć poniższego kodu:

```
            mov     ax, star
            mov     var, ax
            mov     cx, stop
```

```
                sub     cx, ax
                jl      SkipFor
                inc     cx
LP:             instrukcje
                inc     var
                loop    LP
SkipFor:
```

10.7 WYKORZYSTANIE REJESTRÓW A PĘTLE

Zważywszy ,że uzyskujemy dostęp do rejestrów 80x86 dużo szybciej niż do komórek pamięci, rejestry są idealnym miejscem do umieszczenia zmiennych sterujących pętli (zwłaszcza dla małych pętli). Punkt ten jest podkreślony, ponieważ instrukcja loop wymaga zastosowania rejestru cx. Jednakże, jest kilka problemów związanych z zastosowaniem rejestrów wewnątrz pętli. Podstawowym problemem z zastosowaniem rejestrów jako zmiennej sterującej pętli jest to ,że rejestry są ograniczonym zasobami. W szczególności, jest tylko jeden rejestr cx. Dlatego też, poniższy kod nie będzie pracował poprawnie:

```
                mov     cx, 8
Loop1:          mov     cx, 4
Loop2:          instrukcje
                loop    Loop2
                instrukcje
                loop    Loop1
```

Intencją tu było stworzenie zbioru pętli zagnieżdżonych, to znaczy, jedna pętla wewnątrz innej. Pętla wewnętrzna (Loop2) powinna powtarzać się cztery razy dla każdego z ośmiu wykonań pętli zewnętrznej (Loop1). Niestety, obie pętle stosują instrukcję loop. Dlatego też będzie to pętla nieskończona ponieważ cx będzie ustawiane na zero (które loop traktuje jako 65,536) na końcu pierwszej instrukcji loop. Ponieważ cx jest zawsze po napotkaniu drugiej instrukcji loop. Sterowanie zostanie zawsze przekazane do etykiety Loop1.Rozwiązanie tu jest zapisanie i odzyskanie rejestru cx lub użycie różnych rejestrów w miejsce cx dla pętli zewnętrznej:

```
                mov     cx, 8
Loop1:          push    cx
                mov     cx, 4
Loop2:          instrukcje
                loop    Loop2
                pop     cx
                instrukcje
                loop    Loop1
```

lub:

```
                mov     bx, 8
Loop1:          mov     cx, 4
Loop2:          instrukcje
                Loop    Loop2
                Instrukcje
                dec     bx
                jnz     Loop1
```

Niepoprawny rejestr jest jedną z głównych źródeł błędów w pętlach programów asemblerowych.

10.8 POPRAWIANIE WYDAJNOŚĆI

Mikroprocesory 80x86 wykonują sekwencje instrukcji z oślepiająca szybkością Rzadko będziemy spotykać program który jest wolny nie zawierający pętli. Ponieważ pętle są głównym źródłem problemów z wydajnością wewnątrz programów ,są one miejscem na które spoglądamy, kiedy próbujemy przyspieszyć nasze oprogramowanie. Rozważania na temat jak pisać bardziej wydajne programy są poza zakresem tego rozdziału, jednak jest parę rzeczy, na które powinniśmy uważać kiedy projektujemy pętle w naszych programach. Namierzają one i usuwają wszystkie niepotrzebne instrukcje z pętli, aby zredukować czas jaki zabiera im wykonanie jednej iteracji pętli.

10.8.1 PRZENOSZENIE WARUNKU ZAKOŃCZENIA NA KONIEC PĘTLI

Rozważmy poniższych wykres przepływu dla trzech typów pętli przedstawionych wcześniej:

Pętla repeat..until :

    Kod inicjujący
        Treść pętli
    Testowanie warunku zakończenia
    Kod następujący po pętli

Pętla while:

    Kod inicjujący
    Test zakończenia pętli
        Treść pętli
        Skok do testu
    Kod następujący po pętli

Pętla loop..endloop:

    Kod inicjujący
        Treść pętli, cześć jeden
        Test zakończenia pętli
        Treść pętli część druga
        Skok do treści pętli część 1
    Kod występujący po pętli

Jak możemy zobaczyć, pętla repeat ..until jest najprostsza. Ma to odzwierciedlenie w kodzie asemblerowym wymaganym do implementacji tych pętli .Rozważmy poniższe pętle repeat..until i while, które są identyczne:

```
SI := DI - 20;                  SI := DI - 20;
while (SI <= DI) do             repeat
begin
        instrukcje                      instrukcje
        SI := SI +1;                    SI := SI +1;
End;                            until  SI > DI;
```

Kod w języku asemblera dla tych dwóch pętli:

```
        mov     si, di                  mov     si, di
        sub     si, 20                  sub     si, 20
WL1:    cmp     si, di          U:      instrukcje
        jnle    QWL                     inc     si
        instrukcje                      cmp     si, di
        inc     si                      jng     RU
        jmp     WL1
QWL:
```

Jak widzimy, testowanie warunku zakończenia na końcu pętli pozwala nam usunąć instrukcję jmp z pętli .To może być znaczące jeśli pętla ta jest zagnieżdżona wewnątrz innej pętli. W poprzednim przykładzie nie było problemu z wykonaniem treści przynajmniej raz W poddanej definicji pętli łatwo możemy zobaczyć ,że pętla będzie wykonywana dokładnie 20 razy. Zakładając ,że cx jest dostępny, łatwo zredukujemy tą pętlę do:

```
        lea     si, -20[di]
        mov     cx, 20
WL1:    instrukcje
        inc     si
        loop    WL1
```

Niestety, nie jest to zawsze takie łatwe. Rozważmy poniższy Pascalowski kod:

```
WHILE  (SI <= DI) DO BEGIN
        Instrukcje
        SI  := SI +1;
END;
```

W tym szczególnym przykładzie, mamy niewielkie pojecie co zawiera si na wejściu do pętli. dlatego też, nie możemy zakładać, że treść pętli zostanie wykonana przynajmniej raz. Dlatego też, musimy zrobić badanie przed wykonaniem treści pętli .Test może być umieszczony na końcu pętli włącznie z pojedynczą instrukcją jmp:

```
        jmp     short Test
```

```
RU:         instrukcje
            inc     si
Test:       cmp     si, di
            jle     RU
```

Chociaż kod jest tak długi jak oryginalna pętla while, instrukcja jmp wykonuje się tylko raz zamiast przy każdym powtarzaniu pętli. Zauważmy, że ten drobny zysk na wydajności jest uzyskany kosztem utraty czytelności. Druga sekwencja kodu jest bliska kodu spaghetti, niż oryginalnej implementacji. Taka jest często cena małego wzrostu wydajności. Dlatego też, powinniśmy ostrożnie analizować nasz kod, dla zapewnienia, że wzrost wydajności jest wart utraty przejrzystości. Dużo częściej programiści asemblerowi poświęcają przejrzystość dla wątpliwej zyskowności w wydajności, tworząc niemożliwe do zrozumienia programy.

10.8.2 WYKONYWANIE PĘTLI WSTECZNYCH

Z powodu właściwości flag na 80x86, pętle z zakresem od jakiejś liczby w dół do (lub w górę do) zera są bardziej wydajne niż inne .Porównajmy poniższe pętle pascalowskie i kodach, które generują:

```
for  I := 1 to 8 do                     for I := 8 downto 1 do
        K := K+I-J;                             K := K+I - J;

        mov     I, 1                            mov     I, 8
FLP:    mov     ax, K                   FLP:    mov     ax, K
        add     ax, I                           add     ax, I
        sub     ax, J                           sub     ax, J
        mov     K, ax                           mov     K, ax
        inc     I                               dec     I
        cmp     I, 8                            jnz     FLP
        jle     FLP
```

Zauważmy że poprzez uruchomienie pętli od osiem do jeden (kod po prawej) zachowujemy porównanie przy każdym powtórzeniu pętli.

Niestety nie możemy uczynić wszystkich pętli wstecznymi. Jednak dzięki lekkiemu wysiłkowi i koercji możemy doprowadzić ,żeby większość pętli pracowała jako wsteczne. Jeśli raz uczynisz pętle wsteczna jest dobrym kandydatem instrukcja loop (która będzie poprawiała wydajność pętli na pre-procesorach 486).

Powyższy przykład powiedzie się dobrze ponieważ pętla działa od osiem do jeden. Pętla kończy się kiedy zmienna sterująca pętli ma wartość zero. Co się stanie jeśli musimy wykonać pętlę kiedy zmienna sterująca pętli idzie do zera. Na przykład przypuśćmy, że powyższa pętla ma zakres od siedmiu w dół do zera. Tak długo jak górna granica jest dodatnia, możemy zamienić instrukcję jns w miejsce powyższej instrukcji jnz dla powtarzania pętli jakąś określoną ilość razy:

```
            mov     I, 7
FLP:        mov     ax, K
            add     ax, I
            sub     ax, J
            mov     K, ax
            dec     I
            jns     FLP
```

Pętla ta będzie powtarzana osiem razy przyjmując wartości od siedem do zera w każdym wykonaniu pętli. Kiedy zostanie zmniejszona z zera do minus jeden, zostanie ustawiona flaga znaku i pętla się zakończy.

Zapamiętajmy, że jakieś wartości mogą wyglądać na dodatnie ale są ujemne. Jeśli zmienna sterująca pętli jest bajtowa, wtedy wartość z zakresu 128.255 jest ujemna. Podobnie 16 bitowa wartość z zakresu 32768..65535 jest ujemna. Dlatego też zainicjowanie zmiennej sterującej pętli wartością z zakresu 129.. 255 lub 32769..65535 (lub oczywiście zerem) spowoduje, że pętla zakończy się po jednokrotnym wykonaniu. Może to sprawić nam dużo kłopotu jeśli nie będziemy ostrożni

---

10.8.3 OBLICZANII NIEZMIENNIKÓW PĘTLI

Obliczanie niezmienników pętli jest jakimś obliczeniem, które pojawia się wewnątrz pętli, które zawsze przynosi ten sam wynik. Nie potrzebujemy robić takiego obliczania wewnątrz pętli. Poniższy Pascalowski kod demonstruje pętlę, która zawiera obliczanie niezmienników:

```
FOR  I := 0 TO N DO
        K := K+(I+J-2);
```

Ponieważ J nigdy nie zmienia się w trakcie całego wykonywania tej pętli, podwyrażenie „J-2" może być obliczone na zewnątrz pętli a jej wartość jest używana w wyrażeniu wewnątrz pętli:

```
temp  := J-2;
FOR I := 0 TO N DO
        K := K+ (I+temp);
```

Oczywiście ,jeśli jesteśmy faktycznie zainteresowani poprawianiem wydajności tej szczególnej pętli,dużo lepszym sposobem obliczenia K jest stosowanie formuły

$$K = K + ((N+1) \times temp) + \frac{(N+2) \times (N+2)}{2}$$

To obliczenie dla K jest oparte na formule:

$$\sum_{i=0}^{N} i = \frac{(N+1) \times (N)}{2}$$

Jednakże, proste obliczanie takie jak to nie jest zawsze możliwe. poza tym pokazuje to, że lepszy algorytm jest prawie zawsze lepszy niż skomplikowany kod jaki możemy wykombinować.

W asemblerze, obliczenia niezbędników są równie skomplikowane. Rozważmy taką konwersję powyższego kodu pascalowego:

```
            mov     ax, J
            add     ax, 2
            mov     temp, ax
            mov     ax, n
            mov     I, ax
FLP:        mov     ax, K
            add     ax, I
            sub     ax, temp
            mov     K, ax
            dec     I
            cmp     I, -1
            jg      FLP
```

Oczywiście, pierwszym udoskonaleniem jakie możemy zrobić, jest przeniesienie zmiennej sterującej pętli (I) do rejestru. To daje nam poniższy kod:

```
            mov     ax, J
            inc     ax
            inc     ax
            mov     temp, ax
            mov     cx, n
FLP:        mov     ax, K
            add     ax, cx
            sub     ax, temp
            mov     K, ax
            dec     cx
            cmp     cx, -1
            jg      FLP
```

Działanie to przyspiesza pętlę poprzez usuniecie dostępu do pamięci przy każdym powtórzeniu pętli. Idąc krok dalej, dlaczego nie zastosować rejestru do podtrzymania wartości temp zamiast komórki pamięci:

```
            mov     bx, J
            inc     bx
            inc     bx
            mov     cx, n
FLP:        mov     ax, K
            add     ax,cx
```

```
                sub     ax, bx
                mov     K, ax
                dec     cx
                cmp     cx. -1
                jg      FLP
```
Co więcej, dostęp do zmiennej K może być również usunięty z pętli:
```
                mov     bx, J
                inc     bx
                inc     bx
                mov     cx, n
                mov     ax, K
FLP:            add     ax, cx
                sub     ax, bx
                dec     cx
                cmp     cx, -1
                jg      FLP
```
Końcową poprawką jest zastąpienie instrukcji loop przez instrukcje dec cx / cmp cx, -1 / JG FLP. Niestety pętla ta musi być powtarzana kiedykolwiek zmienna sterująca pętli trafia na zero, instrukcja loop nie może tego zrobić. Jednakże, możemy rozwinąć ostatnie wywołanie pętli (zobacz następną sekcję) i zrobić to obliczenie na zewnątrz pętli jak pokazano poniżej:
```
                mov     bx, J
                inc     bx
                inc     bx
                mov     cx, n
                mov     ax, K
FLP:            add     ax, cx
                sub     ax, bx
                loop    FLP
                sub     ax, bx
                mov     K, ax
```
Jak widzimy, te udoskonalenia redukują liczbę instrukcji wykonywanych wewnątrz pętli a te instrukcje, które pojawiają się wewnątrz pętli są bardzo szybkie ponieważ wszystkie odnoszą się do rejestrów zamiast do komórek pamięci.

Usuwanie obliczeń niezmienników i niepotrzebnych komórek pamięci z pętli (szczególnie pętli wewnętrznej w zbiorze pętli zagnieżdżonej) może stworzyć radykalną poprawę wydajności w programie.

10.8.4 PĘTLE ROZWIJANE

Dla małych pętli, to znaczy takich, których treść składa się z kilku instrukcji ,koszt wymagany dla przetwarzania pętli może stanowić znaczący procent całego czasu przetwarzania. Na przykład, spójrzmy na poniższy kod pascalowski i powiązany kod asemblera 80x86:

FOR I := 3 DOWNTO 0 DO A[I} := 0;

```
                mov     I, 3
FLP:            mov     bx, I
                shl     bx, 1
                mov     A[bx], 0
                dec     I
                jns     FLP
```
Każde wykonanie pętli wymaga pięciu instrukcji. Tylko jedna instrukcja wykonuje żądaną operację (przesuniecie zero do elementu A) Pozostałe cztery instrukcje konwertują zmienną sterującą pętli na indeks do A i sterują powtarzaniem pętli .Dlatego też, zabiera 20 instrukcji zrobienie operacji logicznej wymaganej przez cztery.

Ponieważ jest wiele poprawek które możemy uczynić w tej pętli opierając się na informacjach przedstawionych do tej pory, rozważmy ostrożnie dokładnie co jest takiego, że ta pętla się wykonuje - jest to proste przechowanie czterech zer od A[0] do A[3]. Bardziej wydajnym podejściem jest zastosowanie czterech instrukcji mov dla wykonania tego samego zadania. Na przykład, jeśli A jest tablicą słów wtedy poniższy kod inicjuje A dużo szybciej niż kod powyższy:

```
mov     A, 0
mov     A+2, 0
mov     A+4, 0
mov     A+6, 0
```

Możemy poprawić szybkość wykonania i rozmiar kodu przez zastosowanie rejestru ax do przechowywania zera:

```
xor     ax, ax
mov     A, ax
mov     A+2, ax
mov     A+4, ax
mov     A+6, ax
```

Chociaż jest to przykład banalny, pokazuje korzyści pętli rozwijanej. Jeśli ta prosta pętla pojawi się zamknięta wewnątrz zbioru pętli zagnieżdżonych, będzie możliwa redukcja instrukcji 5:1 podwójnej wydajności tej sekcji naszego programu.

Oczywiście nie możemy rozwijać wszystkich pętli. Pętle które wykonują się zmienną liczbę razy nie mogą być rozwinięte ponieważ jest to rzadki sposób określania (w czasie asemblacji) liczby razy, ile pętla będzie się wykonywała. Dlatego rozwinięcie pętli jest procesem najlepszym dla pętli, które wykonują się znaną liczbę razy.

Nawet jeśli powtarzamy pętlę jakąś stałą liczbę iteracji, może ona nie być dobrym kandydatem na pętlę rozwijaną. Pętle rozwijane tworzą imponującą poprawę wydajności kiedy liczba instrukcji wymaganych do sterowania pętlą (i działania na innych kosztownych operacjach) przedstawia znaczący procent całkowitej liczby instrukcji w pętli. Mając powyższą pętle zawierającą 36 instrukcji w treści pętli (wyłączając cztery instrukcje nadmiarowe), wtedy poprawa wydajności będzie ,w najlepszym razie, tylko 10% (porównując z 300-400%). Dlatego tez, koszt pętli rozwijanej , tj wszystkie dodatkowe kody, które muszą być wprowadzone do naszego programu, szybko osiąga małego zwrotu ponieważ treść pętli rośnie bardziej lub zwiększa się liczba iteracji. Co więcej, wprowadzanie tego kodu do naszego programu może stać się całkiem przykre .Dlatego też pętla rozwijana jest najlepszą techniką stosowaną w małych pętlach.

Zauważ, chipy superskalarne x86 (Pentium i późniejsze) mają sprzętowe przewidywanie rozgałęzień i stosują inne techniki dla poprawy wydajności. Pętle rozwijane na takich systemach w rzeczywistości spowalnia kod ponieważ te procesory są optymalizowane do wykonywania krótkich pętli

---

10.8.5 INDUKCJA ZMIENNYCH

Poniżej jest przedstawiona lekko zmodyfikowana pętla przedstawiona w poprzedniej sekcji:

```
FOR I := 0 TO 255 DO A[I] := 0;
        mov     I, 0
FLP:    mov     bx, I
        shl     bx, 1
        mov     A[bx], 0
        inc     I
        cmp     I, 255
        jbe     FLP
```

Chociaż rozwinięcie tego kodu stworzy olbrzymią poprawę wydajności, zajmuje 257 instrukcji do wykonania tego zadania. Jednakże, możemy sporo zredukować czas wykonania treści pętli stosując indukcje zmiennych. Zmienna indukcyjna jest zmienną, której wartość zależy wyłącznie od wartości jakiejś innej zmiennej, W powyższym przykładzie , indeks do tablicy A śledzi zmienna sterująca pętli (jest zawsze równa wartości zmiennej sterującej pętli razy dwa)Ponieważ I nie pojawia się nigdzie indziej w pętli ,nie ma sensu wykonywanie wszystkich obliczeń na I. Dlaczego nie działać bezpośrednio na wartościach indeksu tablicy? Poniższy kod demonstruje tą technikę:

```
        mov     bx, 0
FLP:    mov     A[bx], 0
        inc     bx
        inc     bx
        cmp     bx, 510
        jbe     FLP
```

Tu, kilka instrukcji uzyskuje dostęp do pamięci, gdzie były zastąpione instrukcjami które uzyskują dostęp tylko do rejestrów. Inną poprawą jaką możemy uczynić jest skrócenie instrukcji MOV A][bx],0 stosując poniższy kod:

```
                lea     bx, A
                xor     ax, ax
FLP:            mov     [bx], ax
                inc     bx
                inc     bx
                cmp     bx, offset A+510
                jbe     FLP
```

Ten stransformowany kod poprawia wydajność pętli nawet bardzo. Jednakże, możemy poprawić wydajność poprzez zastosowanie instrukcji loop i rejestru cx do wyeliminowania instrukcji cmp:

```
                lea     bx, A
                xor     ax, ax
                mov     cx, 256
FLP:            mov     [bx], ax
                inc     bx
                inc     bx
                loop    FLP
```

Ta końcowa transformacja tworzy najszybszy wykonywalną wersję tego kodu.

10.8.6 INNE SPOSBY POPRAWY WYDAJNOŚCI

Jest wiele innych sposobów poprawy wydajności pętli wewnątrz naszego programu asemblerowego. Dodatkowa sugestia, dobry tekst o kompilatorach taki jak „Compilers, Principles,Techniques and Tools"Aho, Sethi i Ullmana będzie doskonałym dodatkiem.

10.9 INSTRUKCJE ZAGNIEŻDŻONE

Tak długo jak się będziemy trzymali szablonów dostarczonych w przykładach prezentowanych w tym rozdziale ,bardzo łatwo jest zagnieździć instrukcje jedną wewnątrz drugiej. Tajemnicą uczynienia naszej sekwencji asemblerowej zagnieżdżonej jest zapewnienie, że każda konstrukcja ma jeden punkt wejścia i jeden punkt wyjścia .Jeśli jest to ten przypadek ,wtedy przyjdzie nam z łatwością połączyć instrukcje. Wszystkie instrukcje omawiane w tym rozdziale są zgodne z tą zasadą.

Być może najpowszechniej zagnieżdżanymi instrukcjami są instrukcje if..then..else. Aby zobaczyć jak łatwo zagnieździć te instrukcje w asemblerze rozważmy poniższy kod pascalowski:

```
if (x = y) then
        if (I >= J) then writeln ('At point 1')
        else writeln ('At point 2')
else write ('Error condition');
```

Konwersja tego zagnieżdżenia do języka asemblera zaczyna się od znajdującego się na zewnątrz if, konwertujemy go do asemblacji, potem pracujemy na wewnętrznym if:

```
; if (x = y) then
                mov     ax, X
                cmp     ax, Y
                jne     Else0
;kładziemy wewnętrzne if tutaj
                jmp     IfDone0
;Else write ('Error condition');

Else0:          print
                byte   „Error condition",0
IfDone0:
```

Jak możemy zobaczyć, powyższy kod działa z instrukcją „if (X=Y)" pozostawiając miejsce dla drugiego if. Teraz dodamy drugie if jak następuje:

```
; if (x=y) then
                mov     ax ,X
                cmp     ax, Y
                jne     Else0
; If (I >=J)  then writeln ('At point 1')
```

```
                mov     ax, I
                cmp     ax, J
                jnge    Esle1
                print
                byte    „At point1",cr,lf,0
                jmp     IfDone1
; Else writeln ('At point2');

Else1:          print
                byte    „At point 2",cr,lf,0
IfDone1:
                Jmp     IfDone0
; Else write ('Error condition');

Else0:          print
                byte    „Error condition",0
IfDone0:
```

Jest to oczywista optymalizacja, której nie chcemy rzeczywiście robić dopóki szybkość nie stanie się prawdziwym problemem. Zauważmy w wewnętrznej instrukcji if, że instrukcja JMP IFDONE1 po prostu skacze do instrukcji jmp, która przekazuje sterowanie do IfDone0 Jest bardzo kuszące zastąpić pierwszy jmp przez inny który skacze bezpośrednio do IfDone0.Istotnie,kiedy wchodzimy do środka i optymizujemy nasz kod, będzie to dobrze zrobiona optymalizacja. Jednakże, nie powinniśmy robić takiej optymalizacji naszego kodu, chyba, że rzeczywiście potrzebujemy szybkości. Robiąc tak uczynimy nasz kod trudniejszym do odczytania i zrozumienia. Pamiętajmy ,że chcemy aby wszystkie nasze struktury sterujące mają jedno wejście i jedno wyjście. Zmieniając ten skok jak opisano, dostaniemy dwa punkty wyjścia z wewnętrznej instrukcji if.

Pętla for jest innym powszechną zagnieżdżoną strukturą sterującą. Jeszcze raz, kluczem do zbudowania struktury zagnieżdżonej jest skonstruowanie najpierw obiektu zewnętrznego i wypełnienie go później obiektami wewnętrznymi .jako przykład rozważmy poniższą zagnieżdżoną pętlę for, która dodaje elementy pary dwu wymiarowych tablic razem

```
for  i := 0 to 7 do
        for k := 0 to 7 do
                A[i,j]:= B[i,j] + C[i,j];
```

Jak przedtem zaczniemy od skonstruowania najpierw najbardziej zewnętrznej pętli. Kod ten zakłada, że dx będzie znamienną sterującą pętli dla pętli zewnętrznej (to znaczy dx jest odpowiadające „i"):

```
;for    dx := 0 to 7 do

                mov     dx, 0
ForLp0:         cmp     dx, 7
                jnle    EndFor0
; tu wkładamy wewnętrzną pętle FOR
                inc     dx
                jmp     ForLp0
EndFor0:
```

Teraz dodamy kod dla zagnieżdżonej pętli loop. Zauważmy że zastosowano rejestr cx dla zmiennej sterującej pętli w wewnętrznej pętli for tego kodu

```
;for  dx := 0 to 7 do
                mov     dx,0
ForLp0:         cmp     dx, 7
                jnle    EndFor0
; for cx := 0 to 7 do
                mov     cx,0
```

```
ForLp1:     cmp     cx, 7
            jnle    EndFor1
; Wkładamy  tu kod dla A[dx,cx] := b[dx,cx]+C [dx,cx]
            inc     cx
            jmp     ForLp1
EndFor1:
            inc     dx
            jmp     ForLp0
EndFor0:
```

Końcowym krokiem jest dodanie kodu, który wykonuje to rzeczywiste obliczenie.

10.10 PĘTLE OPÓŹNIENIA CZASOWEGO

Większość czasu komputer pracuje zbyt wolno jak dla większości ludzi. Jednakże ,są sytuacje, kiedy on w rzeczywistości pracuje zbyt szybko .Jednym popularnym rozwiązaniem jest stworzenie pustej pętli dla tracenia małej ilości czasu. W Pascalu powszechnie występują pętle takie jak;

```
            for i := 1 to 10000 do;
```

W asemblerze, możemy zobaczyć porównywalne pętle:

```
            mov     cx, 8000h
DelayLp:    loop    DelayLp
```

Przez ostrożny wybór liczby iteracji, możemy uzyskiwać stosunkowo dokładne przedziały opóźnienia jest jednak małe „ale". To stosunkowo dokładne przedziały opóźnienia są dokładne tylko na Twojej maszynie. Jeśli przeniesiemy nasz program na inną maszynę z innym CPU, szybkością zegara, liczbą stanów oczekiwania, innym rozmiarem pamięci cache lub innymi cechami, odkryjemy, że nasz pętla opóźniająca zajmuje kompletnie inną ilość czasu. Ponieważ jest lepiej niż sto do jednego różnic w szybkości pomiędzy dzisiejszymi PC wysokiej klasy a niskiej klasy, powinniśmy przyjąć żadnej niespodzianki, że powyższa pętla będzie wykonywała się 100 razy szybciej na jednych maszynach niż na innych.

Fakt, że jeden CPU działa 100 razy szybciej niż inny nie oznacza, że nie musimy mieć pętli opóźniającej wykonującej się stałą ilość razy. Istotnie, czyni to problem dużo bardziej poważniejszym. Na szczęście, PC dostarczają sprzętowej matrycy tajmera, które działają z tą samą szybkością bez względu na szybkość CPU. Ten tajmer utrzymuje czas dzienny dla systemu operacyjnego, więc jest bardzo ważne, że pracuje z ta samą szybkością niezależnie czy pracuje na 8088 czy Pentium. W rozdziale o przerwaniach nauczymy się o rzeczywistych łatkach do tych urządzeń wykonujących różne zadania. Teraz po prostu wykorzystamy fakt, że ten układ tajmera wymusza z CPU zwiększenie 32 bitowej komórki pamięci (40:6ch) o 18.2 razy na sekundę .Poprzez patrzenie na tą zmienną możemy określić szybkość CPU i zmodyfikować licznik wartości dla pustej pętli odpowiednio.

Podstawową ideą poniższego kodu jest obserwacja zmiennej tajmera BIOSA dopóki się zmienia. jedna zmiana, zaczyna liczyć liczbę iteracji przez jakąś część pętli dopóki zmienna tajmera BIOSA nie zmieni się ponownie. Jeśli wykonujemy podobną pętlę tą sam ilość razy będzie wymaga około 1/18.2 sekundy na wykonanie.

Poniższy program demonstruje jak stworzyć taki podprogram Delay

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

; PPI_B is the I/O address of the keyboard/speaker control
; port. This program accesses it simply to introduce a
; large number of wait states on faster machines. Since the
; PPI (Programmable Peripheral Interface) chip runs at about
; the same speed on all PCs, accessing this chip slows most
; machines down to within a factor of two of the slower
; machines.

PPI_B           equu      61h

; RTC is the address of the BIOS timer variable (40:6ch).
; The BIOS timer interrupt code increments this 32-bit
; location about every 55 ms (1/18.2 seconds). The code
; which initializes everything for the Delay routine
; reads this location to determine when 1/18th seconds
; have passed.

RTC             textequ  <es:[6ch]>

dseg            segment  para public 'data'


 ; TimedValue contains the number of iterations the delay
 ; loop must repeat in order to waste 1/18.2 seconds.

 TimedValue      word     0

 ; RTC2 is a dummy variable used by the Delay routine to
 ; simulate accessing a BIOS variable.

 RTC2            word     0


 dseg            ends



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

 ; Main program which tests out the DELAY subroutine.

 Main            proc
                 mov      ax, dseg
                 mov      ds, ax

                 print
                 byte     "Delay test routine",cr,lf,0
```

```
; Okay, let's see how long it takes to count down 1/18th
; of a second. First, point ES as segment 40h in memory.
; The BIOS variables are all in segment 40h.
;
; This code begins by reading the memory timer variable
; and waiting until it changes. Once it changes we can
; begin timing until the next change occurs. That will
; give us 1/18.2 seconds. We cannot start timing right
; away because we might be in the middle of a 1/18.2
; second period.

                mov     ax, 40h
                mov     es, ax
                mov     ax, RTC
RTCMustChange:  cmp     ax, RTC
                je      RTCMustChange

; Okay, begin timing the number of iterations it takes
; for an 18th of a second to pass. Note that this
; code must be very similar to the code in the Delay
; routine.
```

```
                mov     ax, 40h
                mov     es, ax
                mov     ax, RTC
RTCMustChange:  cmp     ax, RTC
                je      RTCMustChange

; Okay, begin timing the number of iterations it takes
; for an 18th of a second to pass. Note that this
; code must be very similar to the code in the Delay
; routine.

                mov     cx, 0
                mov     si, RTC
                mov     dx, PPI_B
TimeRTC:        mov     bx, 10
DelayLp:        in      al, dx
                dec     bx
                jne     DelayLp
                cmp     si, RTC
                loope   TimeRTC

                neg     cx                      ;CX counted down!
                mov     TimedValue, cx          ;Save away

                mov     ax, ds
                mov     es, ax

                printf
                byte    "TimedValue = %d",cr,lf
                byte    "Press any key to continue",cr,lf
                byte    "This will begin a delay of five "
```

```
                byte     "seconds",cr,lf,0
                dword    TimedValue

                getc

                mov      cx, 90
DelayIt:        call     Delay18
                loop     DelayIt

Quit:           ExitPgm  ;DOS macro to quit program.
Main            endp

; Delay18-This routine delays for approximately 1/18th sec.
;       Presumably, the variable "TimedValue" in DS has
;       been initialized with an appropriate count down
;       value before calling this code.

Delay18         proc     near
                push     ds
                push     es
                push     ax
                push     bx
                push     cx
                push     dx
                push     si

                mov      ax, dseg
                mov      es, ax
                mov      ds, ax
```

```
; The following code contains two loops. The inside
; nested loop repeats 10 times. The outside loop
; repeats the number of times determined to waste
; 1/18.2 seconds. This loop accesses the hardware
; port "PPI_B" in order to introduce many wait states
; on the faster processors. This helps even out the
; timings on very fast machines by slowing them down.
; Note that accessing PPI_B is only done to introduce
; these wait states, the data read is of no interest
; to this code.
;
; Note the similarity of this code to the code in the
; main program which initializes the TimedValue variable.

                mov     cx, TimedValue
                mov     si, es:RTC2
                mov     dx, PPI_B

TimeRTC:        mov     bx, 10
DelayLp:        in      al, dx
                dec     bx
                jne     DelayLp
                cmp     si, es:RTC2
                loope   TimeRTC

                pop     si
                pop     dx
                pop     cx
                pop     bx
                pop     ax
                pop     es
                pop     ds
                ret
Delay18         endp



cseg            ends

sseg            segment para stack 'stack'
stk             word    1024 dup (0)
sseg            ends
                end     Main
```

10.14 PODSUMOWANIE

Rozdział ten omówił implementację różnych struktur sterujących w programach języka asemblera zawierających instrukcje warunkowe (instrukcje if..then..else i case), stany maszynowe i iteracje (pętle while repeat..until (do/while), loop..endloop i for).Podczas gdy asembler daje nam elastyczność w tworzeniu własnych struktur sterujących, robiąc to często tworzymy programy które są trudne do odczytania i zrozumienia. Chyba że sytuacja wymaga absolutnie czegoś innego ,powinniśmy spróbować wzorować nasze struktury sterujące asemblera na tych z języków wysokiego poziomu jeśli to możliwe.

Najbardziej powszechna struktura sterująca pojawiająca się w HLL'ach to instrukcja IF..THEN..ELSE. Możemy łatwo zsyntetyzować instrukcje if..then i if..then..else w asemblerze stosując instrukcję cmp, skoków

warunkowych, i instrukcji jmp. Aby zobaczyć jak skonwertować HLL'ową instrukcję if..then..else do języka asemblera zajrzyj do

*Sekwencja IF..THEN..ELSE"

Drugą popularną instrukcją warunkową HLLi jest instrukcja case (switch).Instrukcja case dostarcza wydajnego sposobu dla sterowania przepływem danych do jednej lub wielu różnych instrukcji w zależności od wartości jakiegoś wyrażenia. podczas gdy jest wiele sposobów implementacji instrukcji case w asemblerze, najpowszechniejszym sposobem jest użycie tablicy skoków .Dla instrukcji case z przyległymi wartościami, jest to prawdopodobnie najlepsza implementacja. Dla instrukcji case, która ma znaczną przestrzeń nie przylegających wartości, implementacja if..then..else lub jakaś inna technika są lepsze szczegóły zajrzyj:

*Instrukcja CASE

Stany maszynowe dostarczają użytecznego paradygmatu w pewnych programistycznych sytuacjach. Sekcja kodu która implementuje stany maszynowe utrzymuje historię wcześniejszego wykonania wewnątrz zmiennej stanu .Późniejsze wykonanie kodu pobiera historię różnych „stanów" w zależności od wcześniejszego wykonania. Skoki pośrednie dostarczają wydajnego mechanizmu dla implementacji stanów maszynowych w asemblerze. Rozdział ten dostarcza krótkiego wprowadzenia do stanów maszynowych. Aby zobaczyć jak implementować stany maszynowe za pomocą skoków pośrednich zobacz:

*Stany maszynowe i skoki pośrednie

Język asemblera dostarcza bardzo mocnych elementów podstawowych dla konstruowania szerokiego wyboru struktur sterujących. Chociaż rozdział ten skupia się na symulowaniu konstrukcji HLL, możemy zbudować jakieś zawiłe struktury sterujące korzystając z instrukcji cmp 80x86 i skoków warunkowych. Niestety, wynik może być bardzo różny do zrozumienia, zwłaszcza przez kogoś innego niż samego autora. Chociaż asembler daje nam wolność do robienia tego co chcemy, dojrzały programista ćwiczy umiar i wybiera tylko taki przebieg sterowania który jest łatwy do odczytu i zrozumienia ;nigdy nie zadawala się kodem zagmatwanym, chyba, że jest to absolutnie konieczne.

*Kod Spaghetti

Iteracja jest jednym z trzech podstawowych składników języka programowania budowanym dla maszyn Von Neumanna. Struktura pętli sterującej dostarcza mechanizmu podstawowej iteracji w większości HLLi. Język asemblera nie dostarcza żadnej pętli podstawowej. Nawet instrukcja 80x86 loop nie jest pętlą rzeczywistą ,jest instrukcją zmniejszania, porównania i odgałęzienia. Pomimo to jest łatwo zsyntetyzować popularne struktury pętli sterujących w asemblerze. Poniższe sekcje omawiają jak zbudować HLL'owską strukturę pętli sterującej w asemblerze:

*Pętle
*Pętla WHILE
*Pętle Repeat..Until
*Pętle LOOP..ENDLOOP
Pętla FOR

Pętle programowe często pochłaniają większość czasu CPU w typowym programie .dlatego też, jeśli chcemy poprawić wydajność naszego programu, pętle są na pierwszym miejscu któremu chcemy się przyjrzeć. Rozdział ten dostarczył kilku sugestii pomocnych przy poprawie wydajności pewnych typów pętli w programach asemblerowych Podczas gdy nie dostarczają kompletnego przewodnika do optymalizacji, poniższe sekcje dostarczają popularnych technik stosowanych przez kompilatory i doświadczonych programistów języka asemblera:

*Stosowanie rejestrów i Pętli
*Poprawianie wydajności
*Przesuwanie warunku zakończenia na koniec pętli
*Wykonywanie pętli wstecznych
*Obliczanie pętli niezmienników
*Pętle rozwijane
*Zmienne wywoływane
*Inne sposoby poprawiania wydajności

10.15 PYTANIA

1) Skonwertuj poniższe Pascalowskie instrukcje do asemblera: (zakładamy, że wszystkie zmienne są dwubajtowymi liczbami całkowitymi ze znakiem)
   a) IF (X=Y) then A := B;
   b) IF (X<=Y) then X:=X+1 ELSE Y:= Y-1;

c) IF NOT ((X=Y) and (Z<>T)) then Z :=T else X := T;
d) IF (X=0) and ((Y-2)>1) then T:=Y-1;

2) Skonwertuj poniższą instrukcję CASE na asembler:

```
CASE I OF
        0: I:=5;
        1: J:=J+1;
        2: K:=I+J;
        3: K:=I-J;
        Otherwise I:=0;
END;
```

3) Która z metod implementacji dla instrukcji CASE (tablica skoków lub forma IF) tworzy najmniejszą ilość kodu (wliczając w to tablicę skoków ,jeśli użyjemy) dla poniższych instrukcji CASE?

a)

```
CASE I OF
        0:instr;
        100:instr;
        1000:instr;
END;
```

b)

```
CASE  I  OF
        0:instr;
        1:instr;
        2:instr;
        3:instr;
        4:instr;
END;
```

4) Dla pytani trzy ,która forma tworzy najszybszy kod?
5) Zaimplementuj instrukcje CASE z pytania trzy używając język asemblera 80x86
6) Jakie są trzy składniki tworzące pętlę?
7) Jakie są trzy główne różnice pomiędzy pętlami WHILE, REPEAT..UNTIL i LOOP..ENDLOOP?
8) Co to jest zmienna sterująca pętli?
9) Skonwertuj poniższe pętle WHILE do asemblera :(Notka: nie optymalizuj tych pętli,)

a)

```
I := 0;
WHILE (I < 100) DO I := I + 1;
```

b)

```
CH := ' ';
WHILE (CH <> '.') DO BEGIN
        CH := GETC;
        PUTC(CH);
END;
```

10) Skonwertuj poniższe pętle REPEAT ..UNTIL do języka asemblera.

a)
```
I := 0;
REPEAT
        I := I + 1;
UNTIL   I >= 100;
```

b)
```
REPEAT
        CH := GETC;
        PUTC(CH);
UNTIL CH = '.';
```

11) Skonwertuj poniższe pętle LOOP..ENDLOOP do języka asemblera

```
a)  I := 0;   LOOP
              I := I + 1;        IF I >= 100 THEN BREAK;
    ENDLOOP;
b)  LOOP
              CH := GETC;    IF CH = '.' THEN BREAK; PUTC(CH);
    ENDLOOP;
```

12) Jakie są różnice ,jeśli, pomiędzy pętlami z pytań 4 ,5 i 6? Czy wykonują one takie same operacje? Która wersja jest bardziej wydajna?
13) Przepisz dwie pętle przedstawione w poprzednim przykładzie w asemblerze, tak sprawnie jak możesz
14) Poprze proste dodanie instrukcji JMP, skonwertuj dwie pętle z pytania cztery do pętli Repeat..Until
15) Poprzez proste dodanie instrukcji JMP, skonwertuj dwie pętle z pytania pięć do pętli WHILE
16) Skonwertuj poniższe pętle FOR na asembler (Notka: masz wolną rękę w użyciu podprogramów dostarczonych w Bibliotece Standardowej UCR):

```
a)  FOR I := 0 to 100 do WriteLn(I);
b)  FOR I := 0 to 7 do
            FOR J := 0 to 7 do
                    K := K*(I-J);
c)  FOR I := 255 to 16 do
            A [I] := A[240-I]-I;
```

17) Słowo zarezerwowane DOWNTO używane wraz z Pascalowską pętlą FOR, uruchamia licznik pętli od najwyższej liczby w dół do najmniejszej liczby. Pętla FOR ze słowem zarezerwowanym DOWNTO jest odpowiednikiem poniższej pętli WHILE:

```
loopvar := initial;
while (loopvar >= final) do begin
          stmt;
          loopvar := loopvar-1;
end;
```

Zaimplementuj poniższe pętle pascalowskie FOR w asemblerze:

a) FOR I := start downto stop do WriteLn(I);

b) FOR I := 7 downto 0 do
```
        FOR J := 0  to 7 do
              K:=K*(I-J);
```
c) FOR I := 255 downto 16 do
```
        A[I] := A[240-I]-I;
```

18) Przepisz pętlę z pytania 11b utrzymując I w BX, J w CX a K w AX
19) Jak przesunąć test zakończenia pętli na koniec pętli poprawiając wydajność tej pętli?
20) Co to jest obliczanie niezbędników pętli?
21) Jak wykonać pętlę wsteczną poprawiając wydajność pętli?
22) Co oznacza rozwinięcie pętli?
23) Jak rozwinąć pętlę poprawiając wydajność pętli?
24) Daj przykład pętli która nie może być rozwinięta
25) Daj przykład pętli, która może być, ale nie powinna być rozwinięta