Asembler

Podstawy programowania w Windows

Stanisław Kruk

www.EscapeMagazine.pl

Asembler. Podstawy programowania w Windows.
Stanisław Kruk

Skład i łamanie: Patrycja Kierzkowska
Korekta: Anna Matusewicz

Wydanie pierwsze, Jędrzejów 2007
ISBN: 978-83-60320-18-1

Wydawnictwo Escape Magazine
ul. Spokojna 14
28-300 Jędrzejów
http://www.EscapeMagazine.pl

darmowy fragment

Rozdział 1

Co to jest Asembler?

W zasadzie każda opowieść lub teoria naukowa, mniej lub bardziej ambitna bazuje na pojęciach elementarnych, dobrze określonych albo znanych od dawna, uznawanych za pewniki niepodlegające dyskusji. Oczywiście te pewniki, aksjomaty, nie wzięły się znikąd, lecz z wnikliwej obserwacji przyrody i dlatego z uwagi na ich oczywistość nie udowadniano ich. To tak jakby udowadniano, że istnieje Ziemia, Słońce, Księżyc. Początek każdej opowieści czy teorii zaczyna się zazwyczaj od uznania czegoś za oczywiste, elementarne, by następnie bazując na takim wartościowaniu tworzyć i rozszerzać wątki tematyczne *wszerz i wzdłuż*.

Aby umieć czytać, rozpoznawać świat, musimy operować jakimś językiem. Nieważne, jakim, bo nawet języki plemienne o słabo rozwiniętej gramatyce, bazujące głównie na emocjach i bogatej gestykulacji niosą całe mnóstwo informacji, rzecz polega na tym by móc właściwie odczytać wszelkie jego odcienie. W dowolnym języku naturalnym wypowiadamy słowo atom i wiemy przecież, co ono oznacza. Słowo to (w naszym języku ojczystym) posiada zaledwie 4 znaki, 4 litery alfabetu. Czy jest jakiś sposób, aby to czy jakiekolwiek inne słowo przekazać maszynie cyfrowej, komputerowi, by mógł on go „zrozumieć", właściwie przetworzyć? Maszyna cyfrowa komunikuje się cyframi, więc zamiast jej podawać słowa składające się z liter, czytelnych tylko dla ludzi, musimy jej dać odpowiedni ciąg cyfr równoważny „znaczeniowo" treści słów.

Na przykład słowo: *atom* zapisujemy w postaci ciągów zerojedynkowych: 01100001 (jako litera *a*) 01110100 (jako *t*) 01101111 (jako *o*) 01101101 (jako *m*). Pisząc zaś w jednym ciągu, mielibyśmy taki oto łańcuch cyfrowy: 011000010111010001101111011011 01. No trudno, mamy coś za coś. Napisane w języku naturalnym słowo *atom* zawiera zaledwie kilka liter, a wypowiedziane zawiera dwie głoski, natomiast zapisane przy pomocy dwóch tylko znaków

0 i 1 słowo *atom* składa się aż z 8*4=32 cyfr; 4 bajty informacji. Jeśli alfabet języka składa się z kilkunastu czy nawet kilkudziesięciu znaków, to wówczas trzeba niewielu znaków alfabetu do zapisania słowa. W przypadku dwuznakowego alfabetu musimy się trochę natrudzić. „Rozmawiając" z maszyną, wystarczy nam język mający w swym alfabecie zaledwie dwie cyfry, 0 i 1, by „opowiedzieć" jej o wszystkim, co się dzieje we Wszechświecie. Naprawdę! Jest taki język? Tak, tak. Ten język nazywa się **Asembler**.

Asembler to język „emocji" procesora – jako „serca" maszyny cyfrowej. Tam nic innego się nie dzieje jak tylko „stwierdzanie": TAK lub NIE, no i ich przeróżne tego kombinacje. Jeśli za TAK podstawimy 1, a za NIE 0, to otrzymamy słowa języka procesora. Taki jest właśnie czysty Asembler, zerojedynkowy. Takim też Asemblerem posługiwali się programiści kilkadziesiąt lat temu. Dzisiejszy Asembler *obrósł nieco w piórka*, bo to już jest Makro- czy Turbo Asembler, i choć nadal w jego wnętrzu siedzą 0 i 1, to jednak z *wierzchu* już ich nie widać.

Współczesny programista, posługując się językami programowania, używa różnych narzędzi programistycznych ułatwiających mu życie, a poważnie mówiąc, służących do poprawnego pisania kodu programu, do jego optymalizacji itp. W niniejszej książce opiszemy również sposób użycia Turbo Debuggera. Oczywiście można wybrać i inne debuggery, jako programy typu freeware'owego czy choćby shareware'owego. Tu, z uwagi na to, że programy asemblerowe przekształcane będą przy pomocy Turbo Assemblera i Turbo Linkera, opisano Turbo Debugger oraz wspomniano o takim debuggerze, który jest dostępny w sieci internet. Wędrówkę po programowaniu zaczniemy oczywiście od przypomnienia programowania w Asemblerze w systemie DOS włącznie z użyciem Turbo Debuggera dla DOS. Dopiero potem, mocno przygotowani do trudniejszej wspinaczki, będziemy zdobywać wyższe szczyty programowania w języku Asembler, czyli programowania w środowisku Windows.

1.1. O procesorze i jego rejestrach – przypomnienie niektórych wiadomości

Programowanie w Asemblerze dla systemu DOS oparte jest na koncepcji nieustannych ingerencji w rejestry procesora, przydzielania odpowiednich fragmentów pamięci odpowiednim rejestrom procesora i wreszcie koncepcji zasadzającej się na odpowiednim rozdzielaniu ról, zadań, odpowiednim aktorom tej specyficznej sceny programistycznej. Głównymi aktorami tej sceny są rejestry procesora (i koprocesora). Przedstawmy ich. Najpowszechniej używanym rejestrem jest akumulator - rejestr powszechnego stosowania, oznaczany literkami AX, a dla procesorów 386 i nowszych jako EAX.

Rejestr AX składa się z dwóch części 8-bitowych, AH i AL. Rejestr EAX ma szerokość 32 bitów, 16 bitów zajmuje wspomniany AX, drugie jego 16 bitów nie zostało nazwane. Rozkaz: MOV AX, 0ABCDh sprawi, że rejestr AX zostanie zapełniony wartością szesnastkową 0ABCD, i rozmieści ją tak, iż w AH znajdzie się wartość AB, zaś w AL, wartość CD. Możemy jednak wyraźnie wskazać określoną część rejestru AH bądź AL. Rozkaz MOV AL,0ABh ulokuje wartość szesnastkową AB w rejestrze AL, natomiast rozkaz MOV AH,0CDh umieści wartość szesnastkową CD w rejestrze AH.

Proszę zwrócić uwagę na zapis liczby szesnastkowej w rozkazie. Przed liczbą szesnastkową stawiamy cyfrę 0, kończymy ją literką h. Literką h sygnalizujemy, że mamy do czynienia z liczbą szesnastkową (heksadecymalną); takie wymagania ma Turbo Asembler (TASM.EXE lub TASM32.EXE) - jeden z programów biorących udział w dwustopniowym przetwarzaniu naszego tekstowego programu asemblerowego o rozszerzeniu ASM do postaci wykonywalnej; drugi stopień przetwarzania *półproduktu* z postaci OBJ do COM lub EXE odbywa się przy pomocy turbo linkera (konsolidatora) występującego pod nazwą pliku TLINK.EXE (lub TLINK32.EXE).

Rejestr AX czy jego rozszerzona wersja 32 bitowa, EAX, otwiera wiele rejestrów, określanych jako rejestry powszechnego stosowania. Kolejnymi rejestrami w tym szeregu są: BX, CX, DX oraz rejestry wskaźnikowe i indeksowe SI, DI, BP, SP. Wszystkie one mają swe 32-bitowe wersje, poszerzone, i występują pod nazwami EBX, ECX, EDX, ESI, EDI, EBP, ESP.

Programując w systemie DOS, ciągle używamy różnego rodzaju segmentów, a głównie kodu. Co to takiego ów segment? Segment - to pewien fragment pamięci. W programie musi być on wskazany przez wyznaczony w tym celu rejestr segmentowy. Rejestry te oznaczono: literami CS, DS, ES, SS oraz FS i GS dla wyższych niż zwykły DOS-owy tryb pracy procesora. I nie posiadają one wersji rozszerzonych. Dodać należy, że te rejestry segmentowe „chodzą zawsze parami" z rejestrami wskaźnikowymi i indeksowymi oraz z rejestrem – wskaźnikiem rozkazów, IP (lub EIP dla 32 bitowych procesorów). Rejestr znaczników, inaczej rejestr flagowy – FLAGS (lub EFLAGS dla 32 bitowych procesorów) zamyka listę rejestrów programowalnych. Ów rejestr jest zbiorem poszczególnych bitów zwanych znacznikami (flagami), które wskazują wystąpienie określonego stanu procesora.

Jest jeszcze coś takiego w komputerze jak koprocesor. Fizycznie występuje on jako dodatkowy, wyspecjalizowany procesor, bądź też jest zintegrowany z procesorem głównym. Koprocesor ma też swoje rejestry w liczbie 8, tworzą one stos. Oznacza się je kolejno symbolami, licząc od wierzchołka stosu: ST(0) lub ST, ST(1), ST(2)...ST(7). Te same rejestry stosowe mogą się stać też rejestrami typu MMX; Technologia MMX™ (ang. **M**anager **M**emory E**X**tended) dostarczyła architekturze intelowskiej nowego środowiska programistycznego.

To środowisko tworzą następujące elementy:

1. Osiem 64 bitowych rejestrów MMX (od MM0 do MM7).
2. Cztery typy danych (packed bytes, packed words, packed doublewords, quadwords).
3. Lista rozkazów MMX.

1.2. Krótko o programowaniu w języku Asembler w środowisku systemu DOS

Zanim przejdziemy do okienek, czyli Windowsa, spróbujmy coś zrobić pożytecznego i konkretnego jeszcze w starym (dobrym?) systemie DOS.

Nasz program ma wyświetlać na ekranie dowolny znak, zatrzymać się aż do czasu, gdy naciśniemy jakiś klawisz, by następnie miękko wylądować w systemie. Nazwijmy ten program znak.asm. A oto i jego treść:

```
Znak Segment
Assume CS:Znak
org 0100h
Start:
Mov ah,02h
Mov dl,'K'
int 21h
mov ah,08h
int 21h
Mov ah,4ch
Int 21h
Znak Ends
End Start
```

Teraz w linii poleceń (w systemie DOS) lub w *trybie okienkowym* w systemie Windows uruchamiamy (Turbo) asembler pisząc: Tasm[.exe] znak[.asm], a następnie naciskamy klawisz Enter. Po tej operacji (asemblacji) otrzymujemy „półprodukt" o nazwie znak.obj. Bierzemy go w dalsze *turbo obroty,* poddając procesowi linkowania (konsolidacji) i pisząc w linii poleceń: Tlink[.exe] znak.[obj]; nawias kwadratowy oznacza, że wpisywanie rozszerzeń jest opcjonalne. Po tych dwóch etapach *miażdżenia* pliku tekstowego o rozszerzeniu ASM otrzymamy plik wykonywalny o rozszerzeniu COM (tu: znak.com). Możemy teraz usłyszeć pytanie: Czy przedstawiony program musi być koniecznie w taki sposób napisany, aby wyświetlał jakiś znak i był to równocześnie program typu COM?

Co do sposobu konstrukcji programu, to istotnie program wyświetlający jakiś znak można zbudować jeszcze w inny sposób. Natomiast, jeśli chodzi o to, czy ma to być COM czy EXE, to trzeba pamiętać, że kandydat na plik COM nie może zawierać linii sygnalizującej segment stosu ani segment danych, musi się też zaczynać od dyrektywy Org z wartością 0100h. Poza tym w programie musi istnieć dyrektywa (etykieta z dwukropkiem) określająca początek i koniec segmentu kodu (tu: nazwana Start). Wreszcie na sam koniec: turbo linker (program Tlink.exe) należy koniecznie uruchomić z parametrem */t*.

Ostatecznie więc wydajemy kolejno takie oto polecenia: Tasm znak a następnie: Tlink/t znak; znak, to nazwa naszego programu asemblerowego. Dopiero w taki sposób otrzymamy plik znak.com. Jeżeli nie zastosujemy się do opisanych powyżej działań odnoszących się zarówno do kwestii zapisu programu jak też do sposobu uruchamiania turbo asemblera i turbo linkera, to wówczas utworzymy plik znak.exe. Plik ten będzie jednak większy od COM-a o całe 512 bajtów, COM zajmuje zaledwie *śmieszne* 14 bajtów.

Zanim przejdziemy dalej przedstawione zostaną dwie równoważne formy programu znak.asm, w postaci szesnastkowej oraz w postaci binarnej; postać binarna - dla mocno niedowierzających, iż nadal programy można zapisywać w postaci zer i jedynek, chociaż trzeba mieć tu anielską cierpliwość i pokorę mnicha.

Postać szesnastkowa programu znak.asm:

```
Znak Segment
Assume CS:Znak
org 0100h
Start:
db 0b4h
db 02h
db 0b2h
db 4bh
db 0cdh
db 21h
db 0b4h
```

```
db 08h
db 0cdh
db 21h
db 0b4h
db 04ch
db 0cdh
db 21h
Znak Ends
End Start
```

Postać binarna programu znak.asm:

```
Znak Segment
Assume CS:Znak
org 0100h
Start:
db 10110100b
db 10b
db 10110010b
db 1001011b
db 11001101b
db 100001b
db 10110100b
db 1000b
db 11001101b
db 100001b
db 10110100b
db 1001100b
db 11001101b
db 100001b
Znak Ends
End Start
```

W *poważnych*, obszernych programach, korzystających z danych wewnętrznych jak i zewnętrznych koniecznie musi wystąpić jakiś łącznik, po którym *spływać* będą te dane. Tym łącznikiem będą segmenty stosu, segmenty danych itp. Taki program asemblerowy może być

utworzony tylko jako program typu EXE.

Spójrzmy na poniższy program napis1.asm:

```
Title napis1.asm
model small
.stack 100h
.data
Napis db 'Programujemy w Asemblerze, z segmentem danych – dla
DOS!',13,10,'$'
.code
Start:
mov    ax,@data
mov    ds,ax
mov    ah,9
mov    dx,offset napis
int    21h
mov ah,08h
int 21h
mov    ax,4c00h
int    21h
end   Start
```

W programie napis1.asm wyraźnie występuje segment stosu o wielkości 100h, segment danych zadeklarowany jest dyrektywą z kropką na przodzie, ma nazwę: .data, segment kodu zaczyna się od dyrektywy .code (też z kropką na przodzie). Uwaga! Program napis1.asm zapisano w innym trybie niż poprzedni program znak.asm, jednak w niczym to nie zmienia sposobu jego działania. Oznacza to tylko powolne przyzwyczajanie się zapisu programów w środowisku Windows, gdyż w takim trybie będą one prezentowane. Niekiedy można spotkać programy typu COM, które na pozór wydawałoby się, że takimi COM-ami być nie powinny, gdyż w tekście są napisy, a więc jest jawne odwołanie się do danych? Na przykład spójrzmy na poniżej zamieszczony program, nazwany tu jako napis2.asm:

```
Title napis2.asm
.code
org 0100h
Start:
mov    ah,9
mov    dx,offset napis
int    21h
mov ah,08h
int 21h
mov    ax,4c00h
int    21h
Napis  db  'Programujemy  w  Asemblerze,  z  segmentem    danych  -  dla
DOS!',13,10,'$'
end   Start
```

Okazuje się, że powyższy program może być typu COM, gdyż dane do tego programu, a konkretnie tekst o nazwie Napis, *wyrzucony* został poza ostatni rozkaz, który znajduje się oczywiście w segmencie kodu. Rejestry CS i DS mają tę samą wartość, więcej objaśnień tu nie potrzeba. Zresztą spójrzmy na te programy z okien Turbo Debuggera (dla DOS).

Rozdział 2

Narzędzia programisty

2.1. Turbo Debugger (TD.EXE) – dla DOS

Turbo Debugger – program, który w najogólniejszym ujęciu służy do analizy kodu i danych zawartych w analizowanym (debuggowanym) programie. W celu pełnej analizy programu należy go uprzednio odpowiednio przygotować, a mianowicie poddać go asemblacji i linkowaniu w odpowiedniej opcjach. Z uwagi na to, iż Turbo Debugger posiada bardzo wiele funkcji, a co za tym idzie wiele możliwości analitycznych, ograniczymy się tylko do tych elementarnych, które z punktu widzenia początkującego programisty są najistotniejsze.

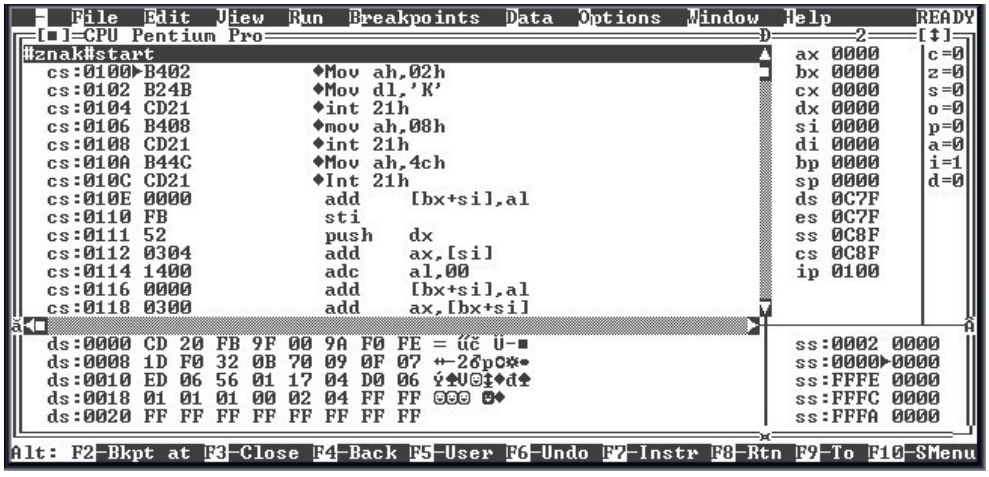
Weźmy pod uwagę nasz pierwszy program znak.asm. Poddajemy go asemblacji: Tasm/zi znak.asm. Następnie linkowaniu: Tlink/v znak.obj. Otrzymany plik znak.exe zawiera teraz tablicę symboli potrzebną w trakcie (turbo) debuggingu. Uruchamiamy turbo debugger: Td.exe znak.exe. Na ekranie otrzymujemy taki oto obraz, jak przedstawiono to na rys. 2.1.1.

Rys. 2.1.1. – Ekran edycji Turbo Debuggera (opcjonalny po wejściu do TD)

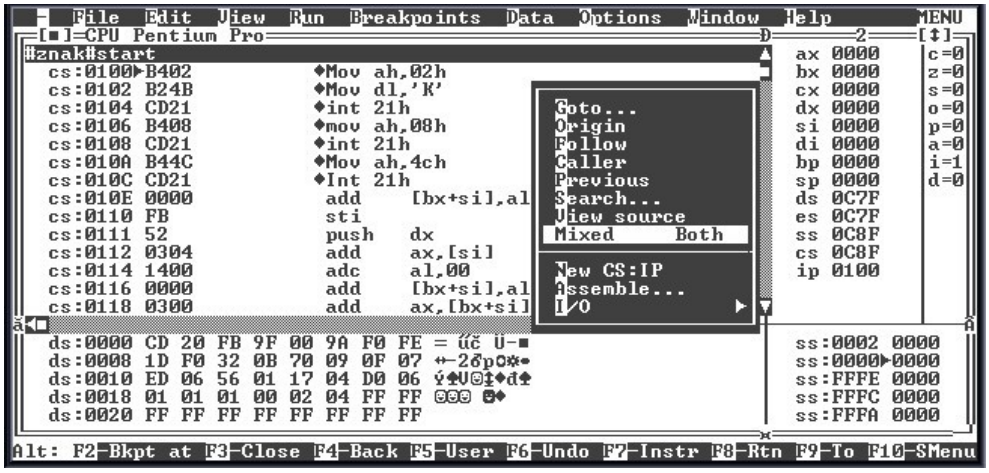
Na dole i u góry mamy całą mnogość funkcji. Nie należy się tym zrażać. Kliknijmy w menu View – rys. 2.1.2.

Rys 2.1.2.– rozwinięcie funkcji View

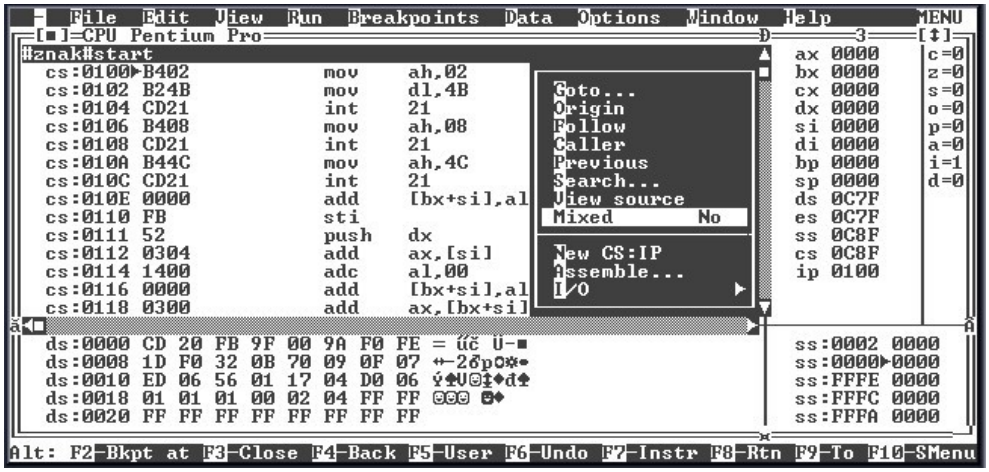
Wybierzmy z tej listy funkcji, CPU – rys. 2.1.3. Funkcja CPU dotyczy jednostki centralnej, czyli krótko mówiąc samego procesora. Z tej funkcji korzystać będziemy najczęściej, aczkolwiek zajrzymy też i do innych ważnych funkcji. Okno CPU składa się z kilka okienek. Okno główne o nazwie CPU *nazwa procesora* pokazuje w swym wnętrzu kod debuggowanego programu. Okno to może pokazywać kod programu na różne sposoby, (wybieramy je, klikając prawym klawiszem myszki, będąc w oknie CPU – rys. 2.1.3., 2.1.4., 2.1.5.,2.1.6.)



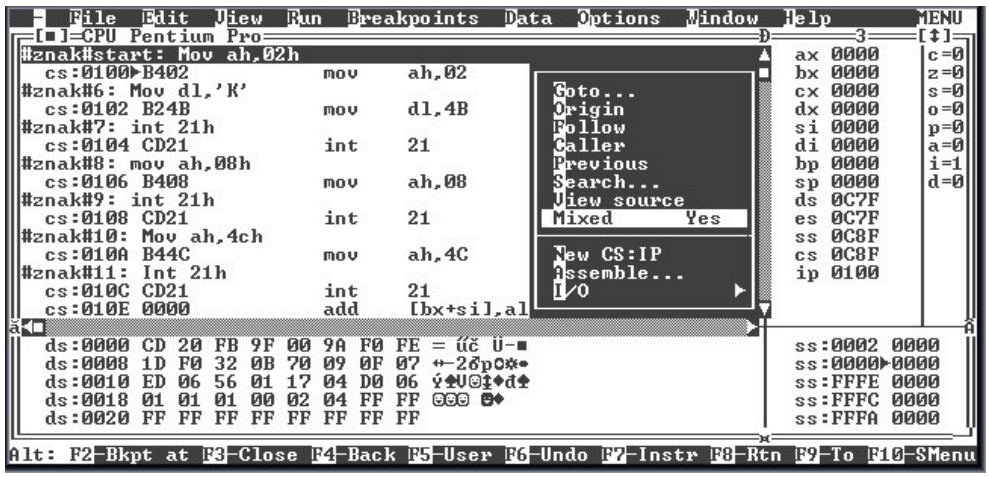
rys. 2.1.3.



rys. 2.1.4.



rys. 2.1.5.



rys. 2.1.6.

Rys. 2.1.3 do 2.1.6 – Wnętrze procesora i pamięci komputera *widziane* przez Turbo Debugger

W górnej części okna CPU widać kod programu (tu: znak.exe) znajdujący się w segmencie kodu, segmentu kontrolowanego przez rejestr CS, w dolnej części tego okna poniżej poziomego podziału okna widać segment danych zawartych w pamięci operacyjnej. Segment ten jest kontrolowany przez rejestr segmentowy DS.

Powyższe obrazki mogą nie przekonywać o potrzebie stosowania Turbo Debuggera, bo przecież wiemy, jaki kod piszemy w Asemblerze, więc w jakim celu obserwować go w TD? Turbo Debugger (TD) pozwala nam nie tylko na obserwację i analizę kodów programu i danych użytych do programu, ale na zachowanie kodu w trakcie wykonywania. Widać co dzieje się z poszczególnymi rejestrami procesora, komórkami pamięci, stosem, ze stanem koprocesora. Również debuggować można programy napisane w innych językach niż Asembler, np. w C, w Pascalu.

Na przykład poniższy program napisany w języku C drukujący na ekran tekst „Asembler” w TD będzie widoczny tak, jak na rys. 2.1.7.

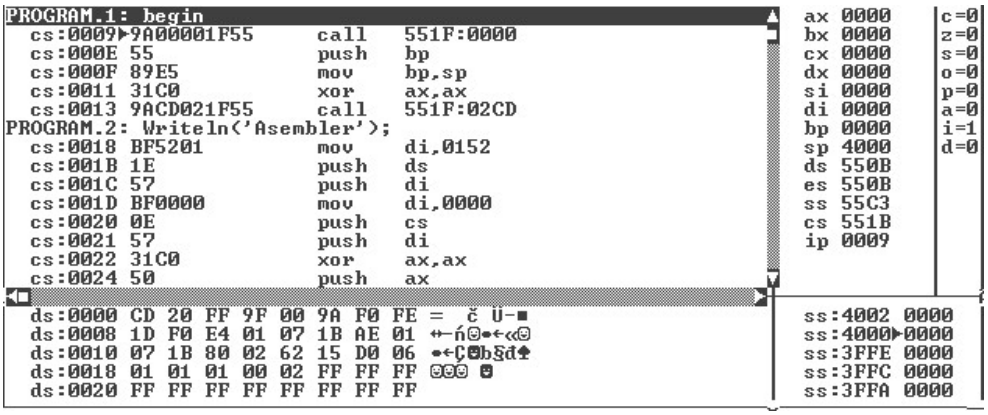
```
#include <stdio.h>
int main()
{
      printf("Asembler\n");
      return 0;
}
```

```
_main: int main()                                    ax 0100   c=0
  cs:02C2▶55              push   bp                  bx 033A   z=1
  cs:02C3 8BEC            mov    bp,sp               cx 0001   s=0
#ZNAK_C#9:  printf("Asembler\n");                    dx 033A   o=0
  cs:02C5 B8AA00          mov    ax,00AA             si 0334   p=1
  cs:02C8 50              push   ax                  di 033A   a=0
  cs:02C9 E89A0C          call   _printf             bp 0000   i=1
  cs:02CC 59              pop    cx                  sp FFF8   d=0
#ZNAK_C#10:  return 0;                               ds 5664
  cs:02CD 33C0            xor    ax,ax               es 5664
  cs:02CF EB00            jmp    #ZNAK_C#11 (02D1)   ss 5664
#ZNAK_C#11: }                                        cs 551B
  cs:02D1 5D              pop    bp                  ip 02C2
  cs:02D2 C3              ret
_atexit
  ds:0000 00 00 00 00 42 6F 72 6C     Borl           ss:FFFA 0000
  ds:0008 61 6E 64 20 43 2B 2B 20 and C++            ss:FFF8▶015B
  ds:0010 2D 20 43 6F 70 79 72 69 - Copyri           ss:FFF6 3246
  ds:0018 67 68 74 20 31 39 39 31 ght 1991           ss:FFF4 551B
  ds:0020 20 42 6F 72 6C 61 6E 64  Borland           ss:FFF2 02C3
```

Rys. 2.1.7 – Postać programu napisanego w języku C w oknie CPU Turbo Debuggera.

Program w Pascalu robiący to samo, co poprzednio przedstawiony, widoczny jest w Turbo Debuggerze w postaci takiej jak widać to na rys. 2.1.8.:

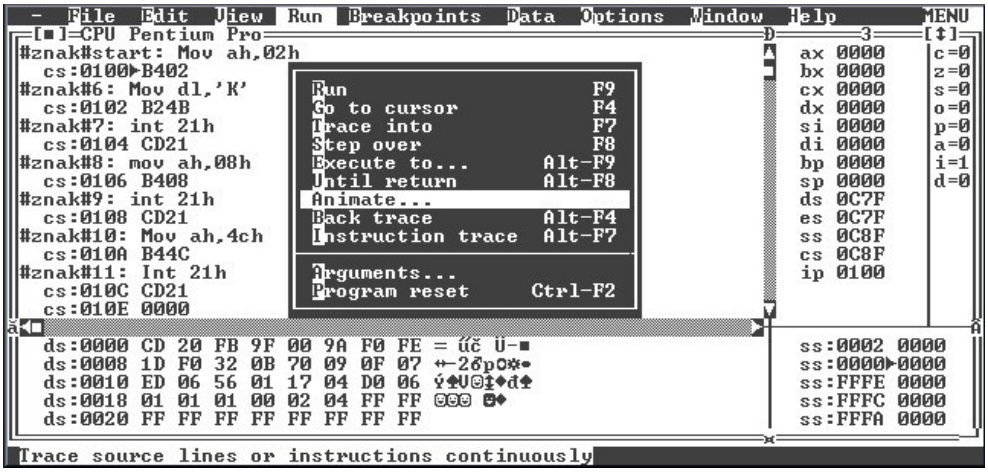
```
begin
Writeln('Asembler');
end.
```



Rys. 2.1.8 – Postać programu napisanego w języku Pascal w oknie CPU Turbo Debuggera.

W pierwszym jak i drugim przykładzie w oknie CPU Turbo Debuggera widać tylko fragmenty kodu asemblerowego tychże programów. Miejsce wywołań procedur printf (w C) czy call *adres* (w Pascalu) jest tu wyraźnie widoczne. Jeślibyśmy przeskoczyli do tego wywołania, to moglibyśmy popaść w tarapaty, albowiem tam jest całe *kłębowisko* odniesień do różnych procedur rozciągających się *wszerz i wzdłuż* komputera.

Jeśli programy jawnie odwołują się za naszym przyzwoleniem do rejestrów koprocesora lub jako rejestrów MMX, wówczas Turbo Debugger możemy tak ustawić, by pokazywał nam to, co dzieje się we wszystkich rejestrach koprocesora. Nie na tym jednak kończą się możliwości TD. Wszystko co się dzieje w rejestrach i w pamięci możemy oglądać w oknach TD bez dotykania palca do klawiatury czy myszki, w tempie takim, jakim chcemy, ustawiając czas w funkcji Animate. – rys. 2.1.9.



Rys. 2.1.9. – Funkcja Animate... umożliwia samoczynne wykonywanie się programu (w TD i w TD32) z dowolną szybkością.

W pewnym ograniczonym jednak zakresie możemy przy pomocy funkcji Back trace Alt-F4 wykonywać program w kierunku wstecznym. Jak widać możliwości programu TD są ogromne, nie wnikając tu już w szczegóły funkcji Breakpoints (punkty wstrzymań), używanej głównie przez szperaczy w kodach programów.

Spójrzmy jeszcze na prawą stronę okna Turbo Debuggera. Widać tam stan rejestrów procesora (ax, bx, cx....ip), stan rejestru flagowego (c=0,z=0,...d=0). Poniżej tych okienek rejestrów znajduje się okienko stanu stosu procesora (ss:0002 0000 itd.)

Bardzo interesujące są też obserwacje rejestrów stosu koprocesora lub jako „typowych" rejestrów MMX (te ostatnie widoczne będą wówczas, gdy zanurzymy się w system Windows). A gdzie znajdujemy ten koprocesor, by zobaczyć jak on pracuje? Spójrzmy najpierw na prosty program z bezpośrednim użyciem koprocesora.

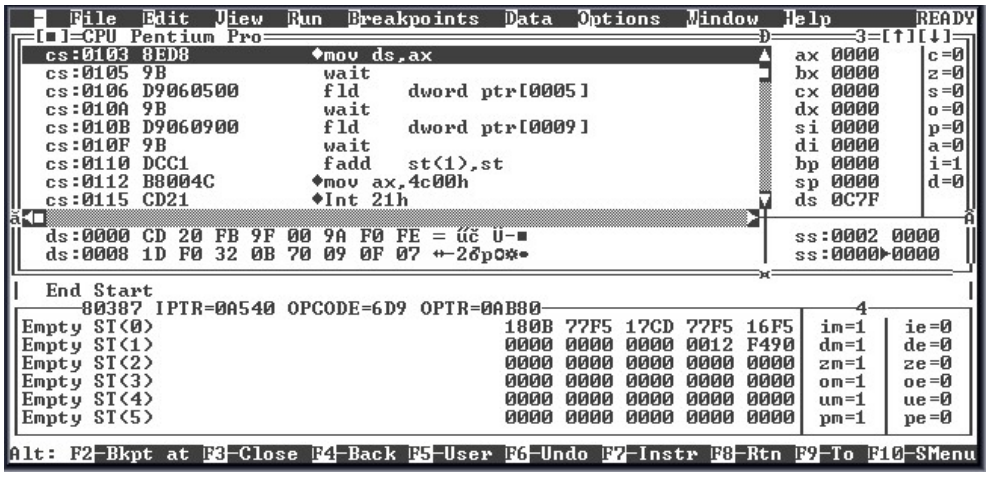
```
Title Fadd.asm
Kopro SEGMENT 'code'
ASSUME CS:Kopro,DS:Dane
;;;
Dane   Segment
Pocz_danych    DB 'Start'
```

```
a dd 3.8
b dd 5.3
Koniec_danych  DB 'Stop'
Dane Ends
;;;
ORG 100H
Start:
mov ax,Dane
mov ds,ax
fld a
fld b
fadd ST(1),ST
;;;
mov ax,4c00h
Int 21h
Kopro Ends
End Start
```

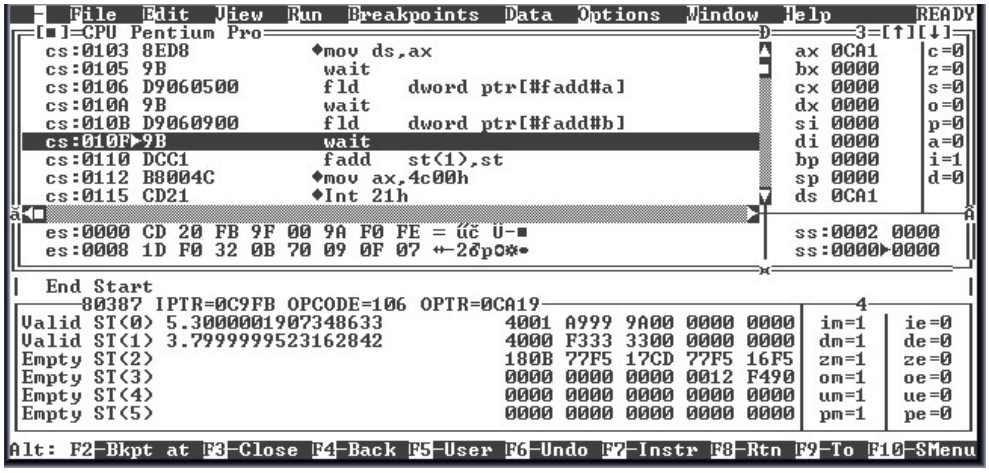
Zadaniem programu jest dodanie do siebie dwóch liczb rzeczywistych z pozostawieniem sumy w jednym z rejestrów stosowych (koprocesora); (*Czytelników chcących dokładnie poznać tematykę programowania koprocesora mogę odesłać do jednej z moich prac na ten temat*). Przełączenie się na okno koprocesora dokonujemy w Turbo Debuggerze w sposób taki, jaki pokazano na rys. 2.1.10.

Możemy też tak porozmieszczać okna, iż będziemy mieć ich tyle, *ile tylko dusza zapragnie*. Klikamy na poszczególnych funkcjach z menu głównego View. Funkcje te wygenerują różne okna i okienka, a my tylko je rozmieszczamy w sposób dla nas wygodny, rys. 2.1.11.

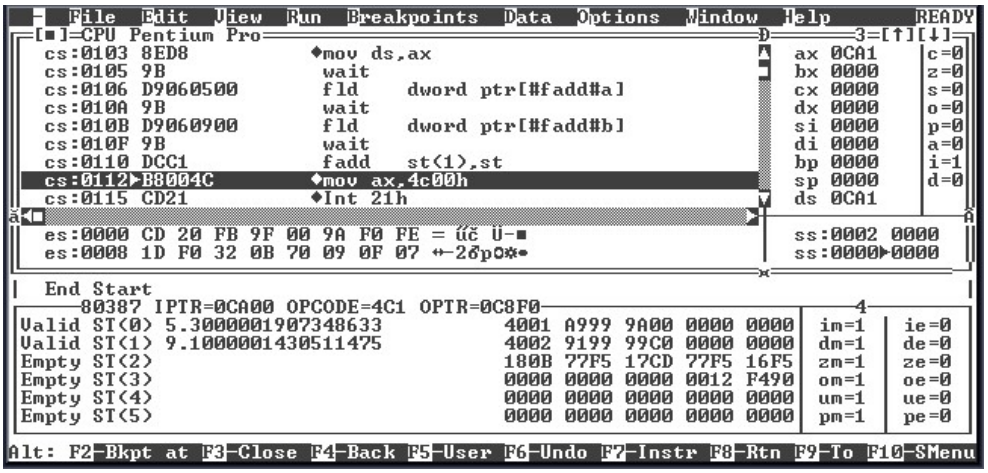
Rys. 2.1.10. – Funkcja Numeric processor pozwala uzyskać okno stanu procesora numerycznego, czyli koprocesora



Rys. 2.1.11. – „Na wprost” pokazano kolejno od góry trzy okna: okno kodu programu, poniżej okno danych i na samym dole okno koprocesora wraz z jego rejestrami; wszystkie okna przedstawiono tu w sposób fragmentaryczny; stan koprocesora – przed wykonaniem powyższego programu



Rys. 2.1.12. – Stan koprocesora po złożeniu składników na jego stos, ale tuż przed obliczeniem sumy



Rys. 2.1.13. – Stan koprocesora tuż po obliczeniu sumy i odłożeniu jej na stosie koprocesora.

Na zakończenie tych kilku uwag o koprocesorze. Proszę zwrócić uwagę na bezpośredni zapis liczb 5.3 i 3.8 na stosie koprocesora oraz na ich sposób kodowania. Dlaczego tak jest? Skoro zapisaliśmy 5.3 i 3.8, to tyle tam powinno być, a nie jakieś tasiemcowe ich postaci i rozwinięcia. Nie wchodząc w skomplikowane szczegóły można tylko powiedzieć tyle, iż fakt ten związany jest z brakiem możliwości dokładnego przedstawienia niektórych liczb w postaci sumy szeregu arytmetycznego. Podobnie też wartości liczb uwidocznione po prawej stronie (na stosie koprocesora) są odpowiednio obliczonymi wartościami szesnastkowymi.

2.2. Turbo Librarian, Bibliotekarz (TLIB.EXE) – nie tylko dla DOS

To narzędzie jest bardzo interesujące, ważne i użyteczne dla programisty, zwłaszcza asemblerowego. Programu Bibliotekarz zawartego w pliku **TLIB.EXE** używa się do tworzenia biblioteki modułów programowych. Jeśli w programie asemblerowym istnieje potrzeba użycia choćby jednego elementu zawartego w bibliotece, wystarczy wówczas w tworzonym programie odpowiednio zaznaczyć połączenie z daną biblioteką, by w czasie konsolidacji programu nastąpiło pobranie żądanego elementu bibliotecznego.

Bibliotekarz umożliwia:

- Utworzenie nowej biblioteki z grupy modułów obiektowych,
- Dodawanie modułów obiektowych do istniejącej biblioteki,
- Usunięcie modułów obiektowych z istniejącej biblioteki,
- Zamianę modułów obiektowych na inne moduły w istniejącej bibliotece,
- Wyłączenie modułów obiektowych z istniejącej biblioteki (i utworzenie pliku obiektowego,
- Wygenerowanie listy zawierającej wszystkie moduły wchodzące w skład istniejącej biblioteki.

Podczas modyfikacji biblioteki, program TLIB zawsze tworzy kopię oryginalnej biblioteki nadając jej rozszerzenie BAK. Plik zawierający oryginalną bibliotekę ma rozszerzenie LIB.
Mamy teraz następujące zagadnienie programistyczne. Program, który nazwiemy tu D1.asm, ma za zadanie współpracować z programem D2.asm. Na jakiej zasadzie ma się odbywać ta współpraca? Otóż program D1.asm o treści przedstawionej poniżej.

```
Title D1.asm
GLOBAL Wynik:BYTE,Dodaj:FAR
;Etykieta Wynik jest typu BYTE, to widać w programie D2.asm;
;Procedura Dodaj jest zaś typu dalekiego, FAR
;GLOBAL to dyrektywa do eksportu i importu
PUBLIC Liczba, Wstaw_wynik
;Powyżej upubliczniono etykietę Liczba i procedurę o nazwie Wstaw_wynik
ASSUME CS:KOD,DS:DANE
```

```
;----------
DANE SEGMENT
  Liczba DB 2
;ta Liczba o wartości 2 będzie wysłana do programu D2.asm, do procedury
o Dodaj, tam znajdującej się.
DANE ENDS
;----------
KOD SEGMENT
  Start:
Wstaw_wynik Proc
mov ax,Dane
mov ds,ax
Call FAR PTR Dodaj   ;dalekie wywołanie procedury Dodaj z D2.asm
mov bl,[Wynik] ;tu następuje pobranie liczby z procedury Dodaj z programu
D2.asm
Ret
Wstaw_wynik Endp
mov ah,4ch
int 21h
KOD ENDS
End START
```

… przekazuje do programu D2.asm liczbę 2 z segmentu danych. W programie D2.asm (patrz poniżej) liczba ta dodawana jest do liczby tam zawartej, po czym przesyłana zostaje w tym programie D2.asm do komórki pamięci o nazwie Wynik. Ale żeby tego *poplątania* było jeszcze mało, to wartość komórki pamięci o nazwie Wynik z programu D2.asm przekazana zostaje do rejestru BL w programie D1.asm:

```
Title D2.asm
ASSUME CS:KOD,DS:DANE
GLOBAL Wynik:BYTE,Dodaj:FAR, Liczba:BYTE
;-----------
DANE SEGMENT
Wynik DB ?
DANE ENDS
;-----------
```

```
KOD SEGMENT
Dodaj Proc Far
mov ah,[Liczba]
add ah,6
mov [Wynik],ah
ret
Dodaj  ENDP
KOD ENDS
END
```

Kilka spraw należy wpierw krótko wyjaśnić. W programach D1.asm i D2.asm pojawiły się tajemnicze (?) dyrektywy: GLOBAL, PUBLIC (mogą być też jeszcze dyrektywy: EXTRN i inne). GLOBAL – definiuje symbol globalny, który może być zdefiniowany w bieżącym lub zewnętrznym module. EXTRN – wskazuje symbol używany w danym module, który zdefiniowany jest w innym module (*dyrektywy tej użyjemy w następnym przykładzie*). PUBLIC – udostępnia definiowany symbol innym modułom. Jeśli symbol nie ma charakteru publicznego, to będzie dostępny tylko z bieżącego pliku. Tyle teorii na ten temat, i wystarczy, bo jeszcze trzeba wspomnieć o właściwościach Bibliotekarza, TLIB.EXE.

Piszemy teraz różne pliki .BAT, aby ułatwić sobie życie z asemblacją, konsolidacją i w ogóle. Nazwijmy plik BAT o treści:

```
tasm/zi D1+D2
TLIB Mojalib +D1.obj+D2.obj
```

jako D1_2_lib.bat

Pierwsza linia pliku spowoduje asemblację plików D1.asm i D2.asm za jednym zamachem w opcji dla Turbo Debuggera, po tej linii muszą się utworzyć dwa pliki obiektowe, D1.obj i D2.obj, *i łaski nie robią*. Natomiast druga linia uruchamia program Bibliotekarz, TLIB.EXE, który właśnie w taki sposób z plików OBJ utworzy nam bibliotekę, tu o nazwie Mojalib, na dysku, w katalogu bieżącym. Na pewno zauważymy tam plik Mojalib.lib.

Co tak naprawdę jest w tej bibliotece? Powiemy o tym w następnym przykładzie, jako skompresowanej wersji tych przykładów. Na tę chwilę potrzebowaliśmy mieć coś takiego jak bibliotekę i umieć z niej, gdy *zastukają programiści asemblerowi do drzwi o północy*, by z niej pilnie skorzystać!

Po co to wszystko, zapyta Czytelnik? Otóż Szanowny Czytelniku robimy to w tym celu, by nauczyć się nie tylko dobrze pisać programy w Asemblerze, ale umieć skorzystać z napisanych programów, tym bardziej, że to co będzie się dziać za chwilę i jeszcze dalej, towarzyszyć nam już będzie przez cały czas, programując, asemblując, linkując programy asemblerowe dla środowiska Windows. Po prostu jest to przygrywka do Windowsa.

Napiszmy teraz kolejnego batcha, który połączy te dwa pliki OBJ, wytworzy nowy, złączony z dwóch, plik D12.EXE na podstawie informacji zawartych w bibliotece Mojalib.lib. Gdy już wszystko będzie gotowe „wskoczymy" do Turbo Debuggera i będziemy obserwować jak nasz program sprawnie działa. Ten batch, nazwijmy go D12.bat, niech będzie posiadał treść:

```
tlink/v D1+D2, D12,,Mojalib.lib
td D12
```

Zanim rozkoszować się będziemy okienkami Turbo Debuggera i naszymi wspaniałymi umiejętnościami programistycznymi, zobaczmy co tak naprawdę w tej bibliotece siedzi. Znowu utwórzmy kolejny batch o treści jak poniżej. Nazwijmy go, jak chcemy, na przykład: Tlib_list.bat i uruchamiamy go.

```
TLIB mojalib, lista
```

Po uruchomieniu takiego batcha utworzy się nam plik o nazwie lista.lst i zawartości:

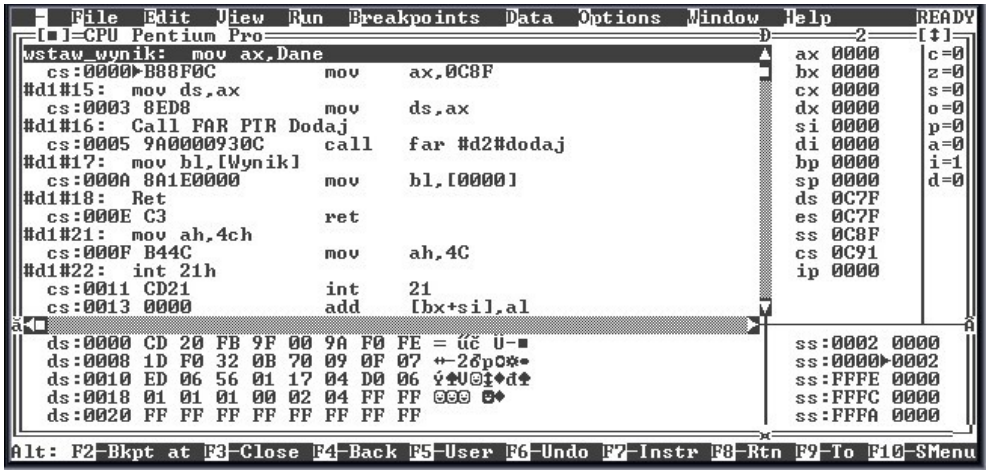
```
Publics by module
D1          size = 20
      LICZBA                            WSTAW_WYNIK
D2          size = 13
      DODAJ                             WYNIK
```

No i co my tu widzimy? Oh, wiele tu ważnego! W pliku lista.lst jest zawarty spis plików – bez ich rozszerzeń – (modułów) obiektowych (OBJ) mających elementy publiczne, nazwy tych elementów oraz i ich rozmiar.

Po wejściu do Turbo Debuggera (TD.EXE) przywita nas ekranem:

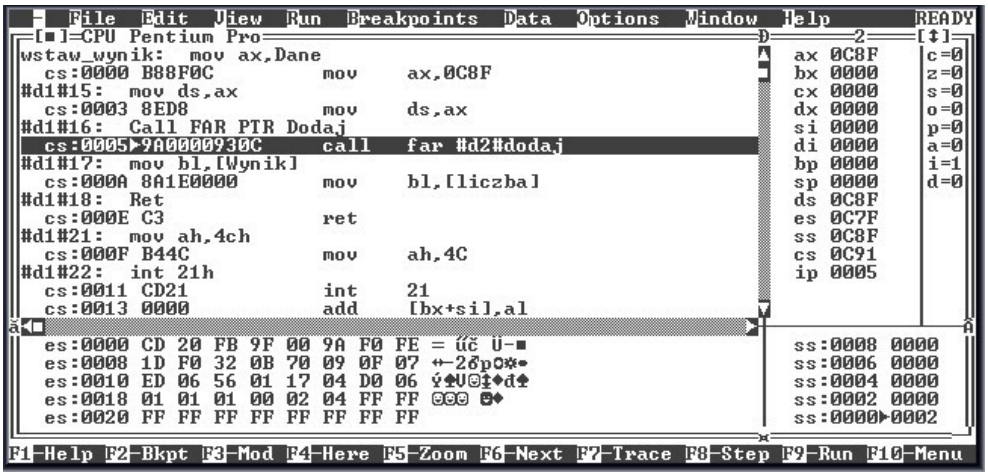
Rys. 2.2.1. – Edycja programu D1 pod Turbo Debuggerem

Wybieramy z górnej belki menu funkcję: View, a stąd CPU. Okno to ustawmy prawym klawiszem myszki na Mixed Yes, aby równocześnie widzieć oryginalny zapis programu i jego odniesienia, patrz poniżej przedstawiony rysunek 2.2.1.

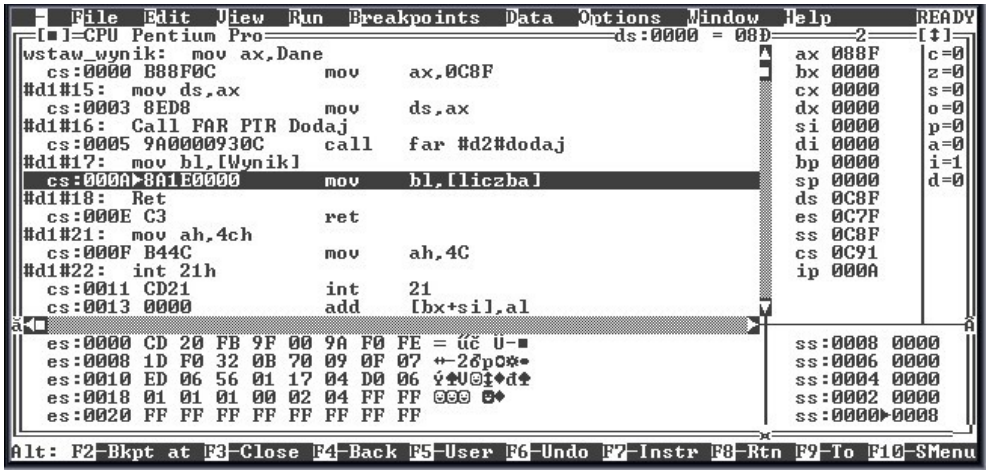


Rys. 2.2.2. – Okno CPU dla programu D1 pod Turbo Debuggerem

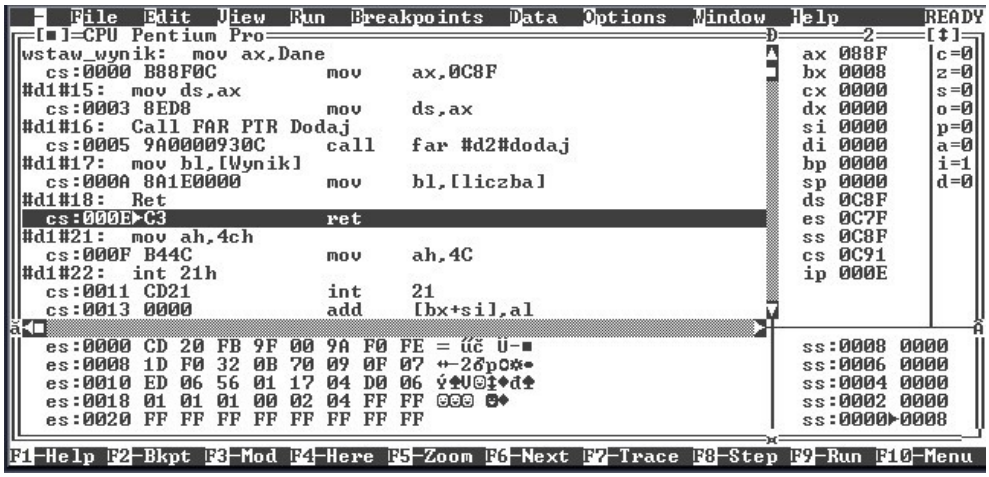
Zaobserwujmy teraz wykonywanie się programu D12.EXE. Naciskając klawisz F8, umożliwiamy wykonywanie się programu bez wchodzenia w jego głąb; klawisz F7 pozwala na wejście w daną procedurę, przerwanie. Nasza obserwacja ma głównie dotyczyć stanu rejestrów AX i BX, gdyż w tych rejestrach *toczy się akcja* programu, w AH odbywa się dodawanie liczb, w BL przesyłanie ich sumy. Popatrzmy więc na kolejne „zdjęcia migawkowe" Turbo Debuggera na rysunkach: 2.2.3., 2.2.4.,2.2.5.



Rys. 2.2.3. – Wygląd ekranu Turbo Debuggera z wykonywania się programu D12.EXE tuż przed wykonaniem wywołania procedury Dodaj



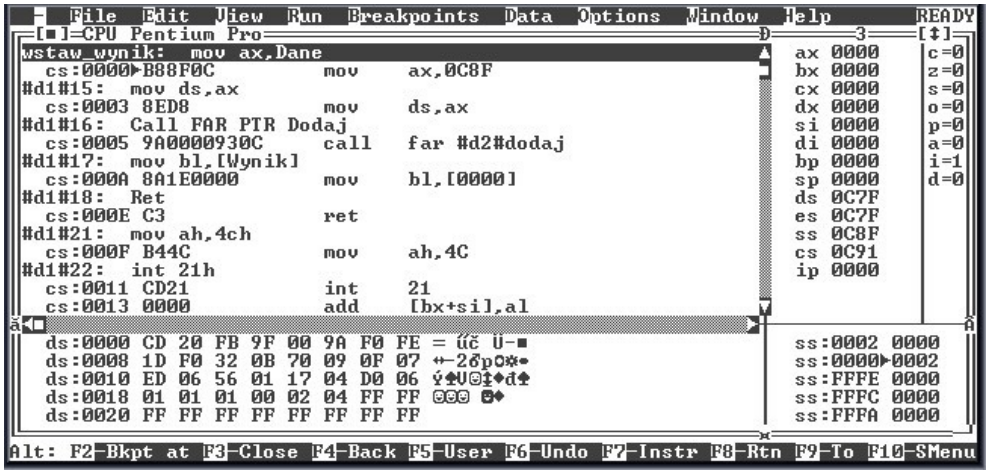
Rys. 2.2.4. – Wygląd ekranu Turbo Debuggera z wykonywania się programu D12.EXE tuż po wykonaniu procedury Dodaj; wartość rejestru AH wynosi teraz 08 (przed, wynosiła 0C, Dlaczego? Wiadomo! Wystarczy popatrzeć na rozkaz: mov ax,0C8F, stąd 0C „wpada" do AH a 8F do AL).



Rys. 2.2.5. – Wygląd ekranu Turbo Debuggera z wykonywania się programu D12.EXE tuż po przesłaniu zawartości komórki pamięci o nazwie Liczba do rejestru BL; teraz BL ma wartość 08, poprzednio było 00 - rys. 2.2.4.)

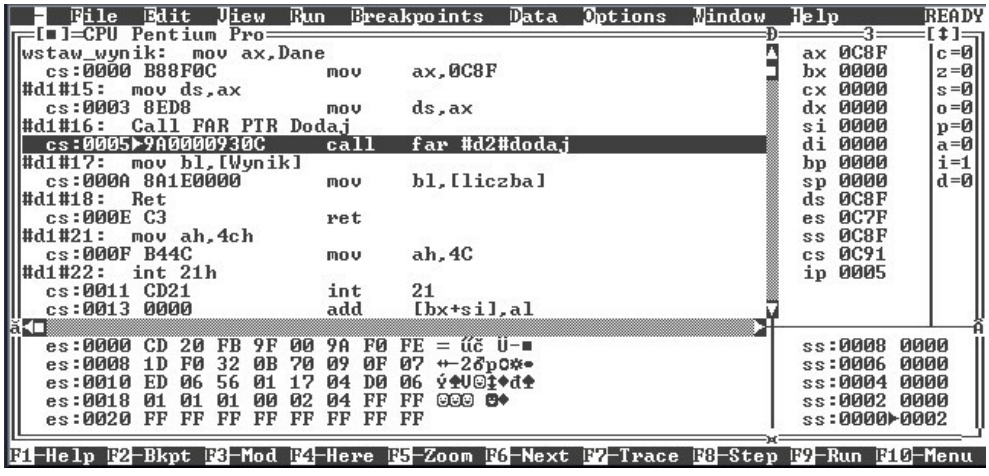
Szczególnie zainteresowani Turbo Debuggerem, bo tak sądzę, że takie zainteresowanie od tego momentu nagle wzrosło, zapytają, a jak zobaczyć zmiany, które dokonują się w międzyczasie w komórkach pamięci.

Wynik? To bardzo prosto, odpowiadamy fanom TD. Weźmy na przykład pod *lupę* tę komórkę, bo tu się coś naprawdę dzieje. Uruchamiamy znowu TD z naszym programem D12.exe; możemy napisać w linii poleceń TD D12 lub uruchomić to z pliku BAT.



Rys. 2.2.6. – Wygląd ekranu Turbo Debuggera z wykonywania się programu D12.EXE

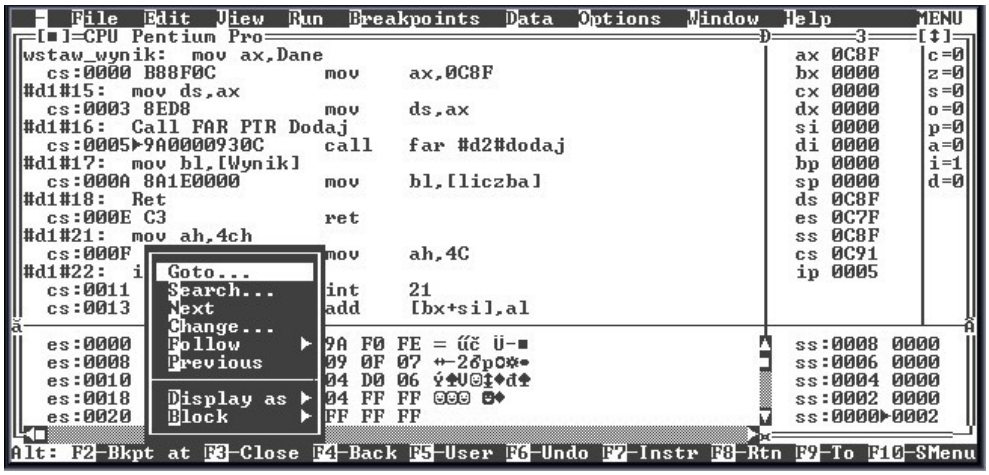
Teraz nasza uwaga bardziej skupi się na dolnym okienku, tu gdzie mamy zapisane wartości komórek pamięci, aczkolwiek również musimy obserwować w oknie kodu, w którym kolejny rozkaz się wykonuje. Po pierwszych dwóch rozkazach koniecznie należy uważnie przyjrzeć się rejestrom, zwłaszcza rejestrowi DS odpowiedzialnemu za kontrolę nad segmentem danych. Na rys. 2.2.6. wartość ta wynosi jeszcze 0C7F, ale już na następnym rysunku 2.2.7?



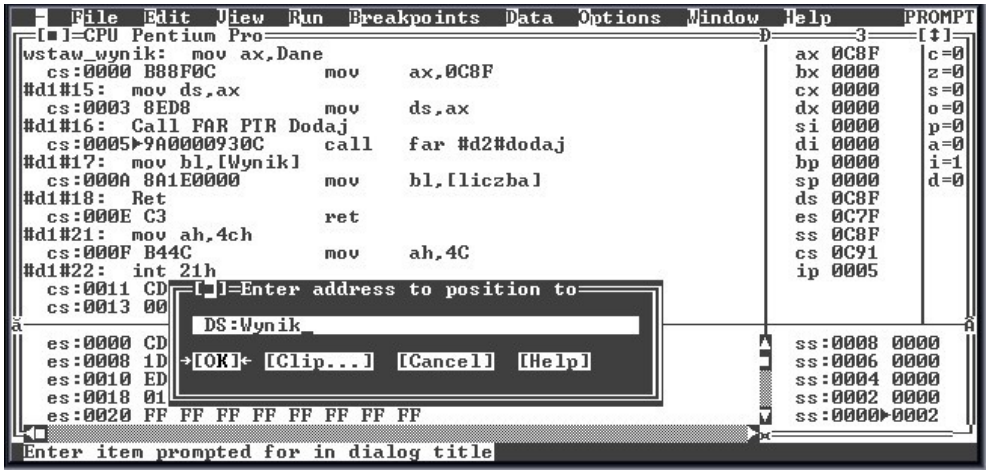
Rys. 2.2.7. – Wykonano dwa pierwsze rozkazy: mov ax, Dane (inna forma: mov ax,0C8F) oraz mov ds,ax. Zmianie uległ m.in. rejestr segmentu danych DS.

Po rozkazie mov ds,ax rejestr „nastawiony” został na segment z danymi do programu D12.exe (dotychczas wartość ta była domyślna), teraz musimy przeskoczyć do okienka z tymi danymi. Jak to zrobić? Stajemy myszką na tym okienku, klikamy w prawy jej klawisz. Po ukazaniu się okienka - rys. 2.2.8. naciskamy klawisz enter, by wejść do opcji Goto... - rys. 2.2.8.

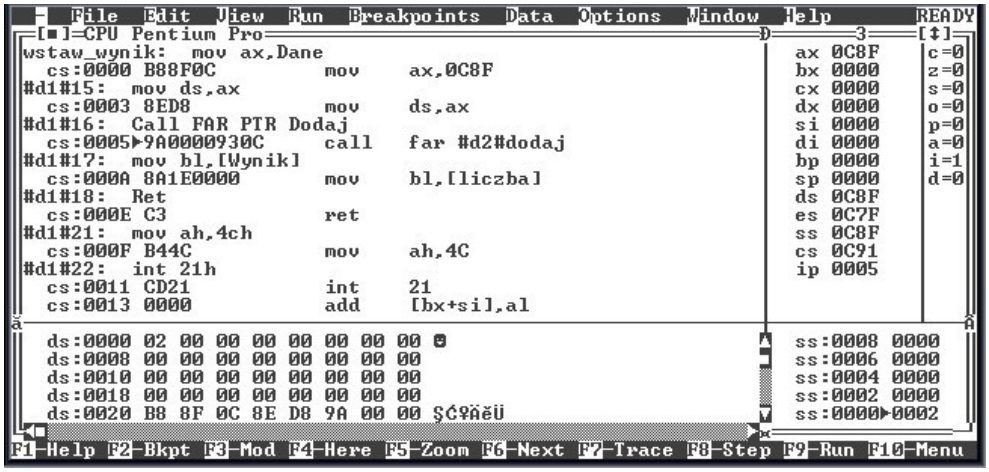
Gdy już weszliśmy tam, ukazuje się naszym oczom kolejne okienko, potrzebne do wpisania adresu komórki pamięci. Adres ten wpisujemy jawnie lub w postaci częściowo niejawnej - rys. 2.2.9. Po wpisaniu adresu śledzonej komórki DS:Wynik naciskamy klawisz enter i nagle pokazują się nam wspaniałe widoki na prawdziwie rzeczywisty segment danych, a nie na jakiś domyślny – rys. 2.2.10. Proszę jeszcze porównać okienko danych z rys. 2.2.9. z okienkiem z rys. 2.2.10. - głównie w kontekście adresów. Poprzednio (domyślny) segment danych kontrolowany był przez dodatkowy rejestr segmentowy ES, a nie przez DS, o jaki nam tu chodziło.



Rys. 2.2.8. – Ustawianie się w segmencie danych

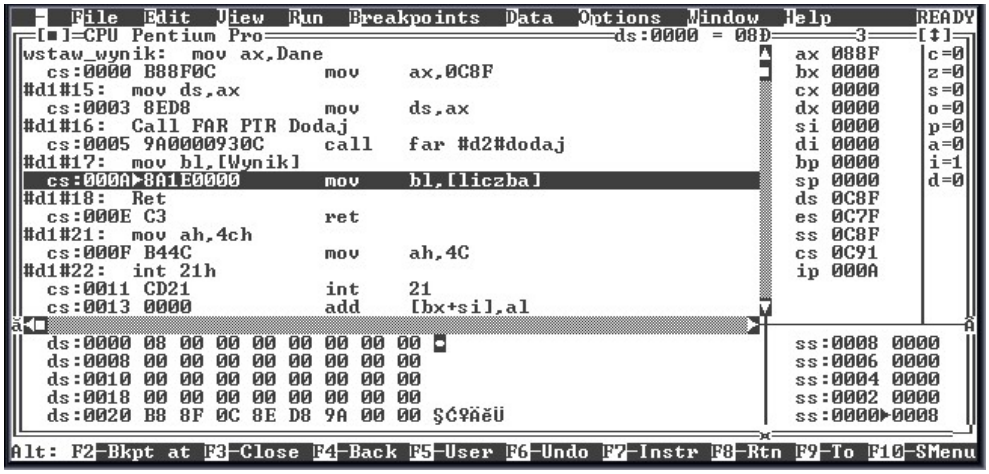


Rys. 2.2.9. – Okienko w segmencie danych do wpisywania adresu komórki (możemy wpisywać go niejawnie, tzn. tak jak na tym rysunku lub jawnie, gdy znamy obydwie części adresu w postaci liczbowej).



Rys. 2.2.10. – Na dole okna, okienko segmentu danych kontrolowane przez rejestr segmentowy DS, którego wartość nie jest domyślna a podyktowana jest ona przez debuggowany program D12.EXE

Na rys. 2.2.10 widzimy początkową (przed dodaniem liczb) wartość komórki o nazwie Wynik o wartości 2. Teraz chwilowo „zeskakujemy” z okienka danych do okienka powyżej leżącego, czyli do okienka kodu - rys. 2.2.11.



Rys. 2.2.11. – Wykonywanie się rozkazu wraz z uwidocznieniem zmian w komórce pamięci segmentu danych.

Wykonując kolejny rozkaz, naciśnięciem klawisza F8 równocześnie, obserwujmy w segmencie danych zmiany wartości komórki z 02 na 08; proszę porównać bajt spod adresu ds:0000 z rys. 2.2.10. z tym samym bajtem z rys. 2.2.11.

Programy asemblerowe pisane dla środowiska Windows i wszelkie czynności przygotowawcze związane z doprowadzeniem takich programów do ich „używalności”, wykazują duże podobieństwa do przedstawianych tu działań. Kolejne działania przedstawione dalej będą tego potwierdzeniem, iż znajdujemy się już u progu programowania w Asemblerze dla środowiska Windows.

Weźmy nadal pod uwagę przedstawione uprzednio programy D1.asm i D2.asm. Gdy programy te wygenerują bibliotekę, Mojalib.lib, napiszmy program asemblerowy, który bezpośrednio wykorzysta zasoby tej biblioteki. Program ten nazwijmy D_extrn.asm. Będzie on mieć następującą treść:

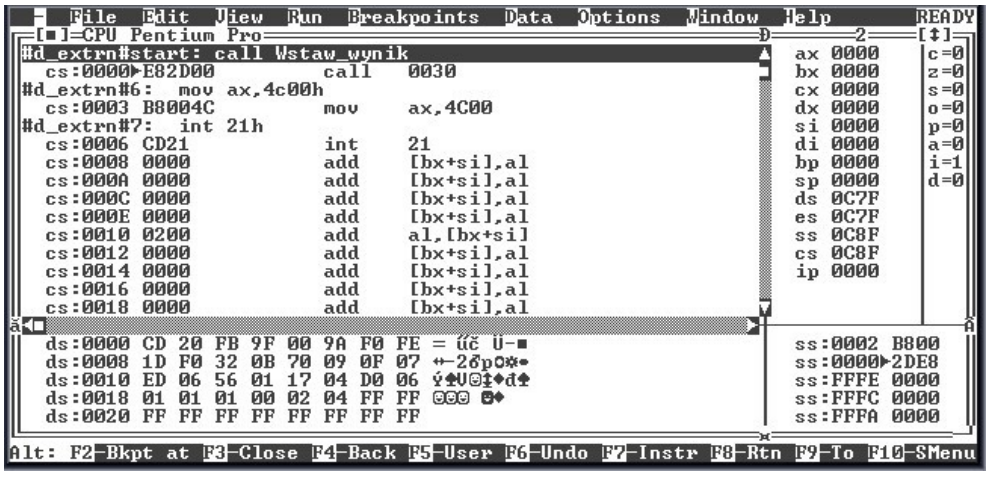
```
Title D_extrn.asm
.model small
extrn Wstaw_liczbe: proc
.code
Start:
```

```
call    Wstaw_liczbe
mov     ax,4c00h
int     21h
end    Start
```

Program przekształcimy do postaci wykonywalnej z użyciem utworzonej biblioteki i w taki sposób, aby można go było prawidłowo debuggować. Nasz batch będzie więc mieć postać:

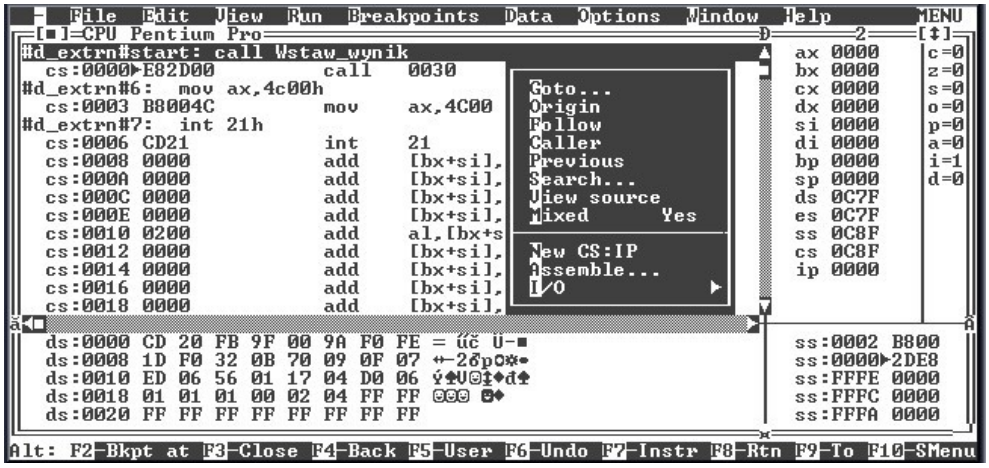
```
tasm/zi D_extrn
tlink/v D_extrn,D_extrn,,Mojalib.lib
td D_extrn
```

Batch automatycznie uruchomi nasz program D_extrn pod TD. W kilku kolejnych obrazach prześledzimy działanie programu D_extrn.exe pod Turbo Debuggerem, wykonując program nie za pomocą naciskania klawisza F8, gdyż wówczas przeskakujemy nad wszystkimi procedurami, lecz za pomocą F7. Przekonamy się, że faktycznie wskakujemy do drugiego modułu, którego w zasadzie nie ma, chociaż musi on być jakoś widziany, w przeciwnym razie program nie funkcjonowałby prawidłowo. Takie „cuda" dzieją się oczywiście za sprawą połączenia z biblioteką i odpowiednim połączeniem między dwoma modułami programowymi. Cały teatr rozgrywa się w pamięci, w różnych jego zaułkach, o adresach wyznaczonych przez program.

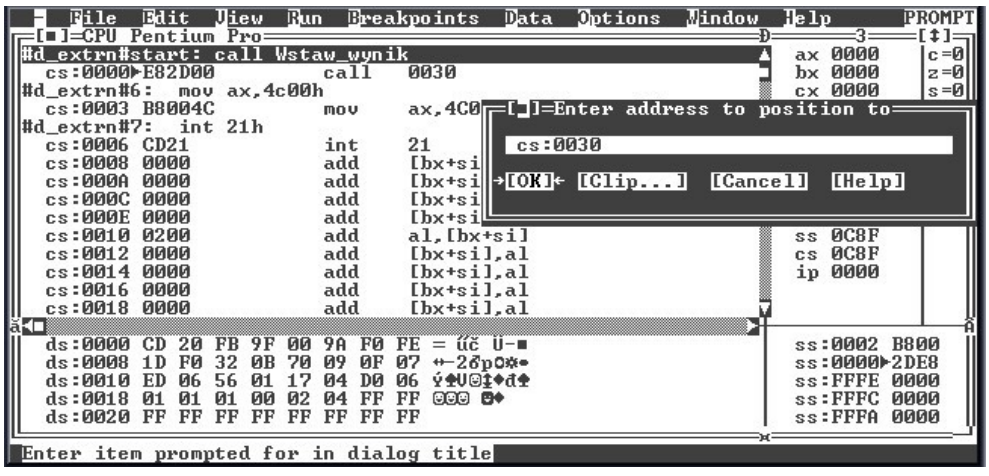


Rys. 2.2.12. – Okno CPU programu Turbo Debugger, *we wnętrzu* gotowy do wykonania program D_extrn.exe; okno ustawiono w opcji Mixed Yes.

Na rys. 2.2.12 w oknie CPU widzimy kod programu. W opcji Mixed Yes, Turbo Debugger pokazuje kod programu w dwojaki sposób, tak jak został on zapisany w programie źródłowym oraz poprzez adresy, gdy w programie występują etykiety, procedury, itp. Widać tu wywołanie procedury w postaci: call Wstaw_wynik jak też w postaci: call 0030. Co oznacza ten drugi zapis? Przekonajmy się! Robiliśmy już podobne doświadczenia, lecz w segmencie danych, jednakże w niczym się one tu w segmencie kodu nie różnią od tamtych doświadczeń z segmentem danych. Kliknijmy prawym klawiszem myszki, będąc w segmencie kodu. Zobaczymy ekran jak na rys. 2.2.13.

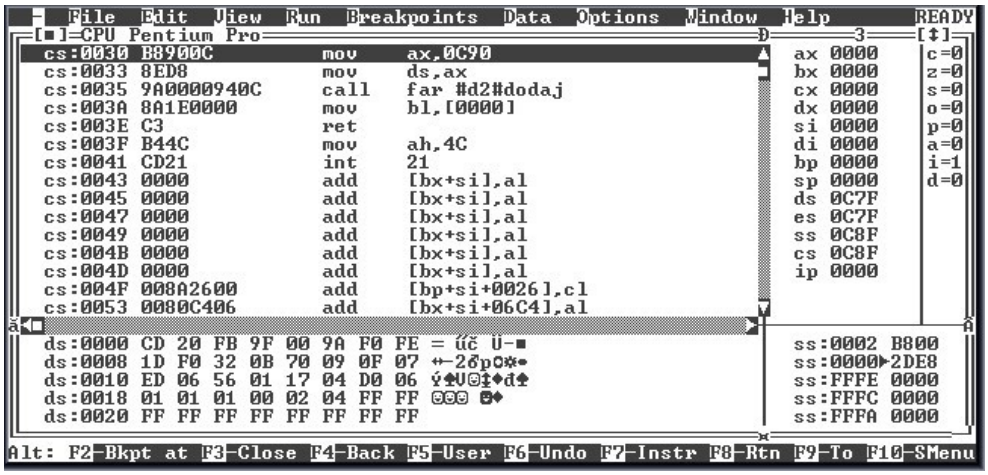


Rys. 2.2.13. – Wybranie opcji: Goto... w segmencie kodu



Rys. 2.2.14. – Wybranie Goto... i wpisanie adresu

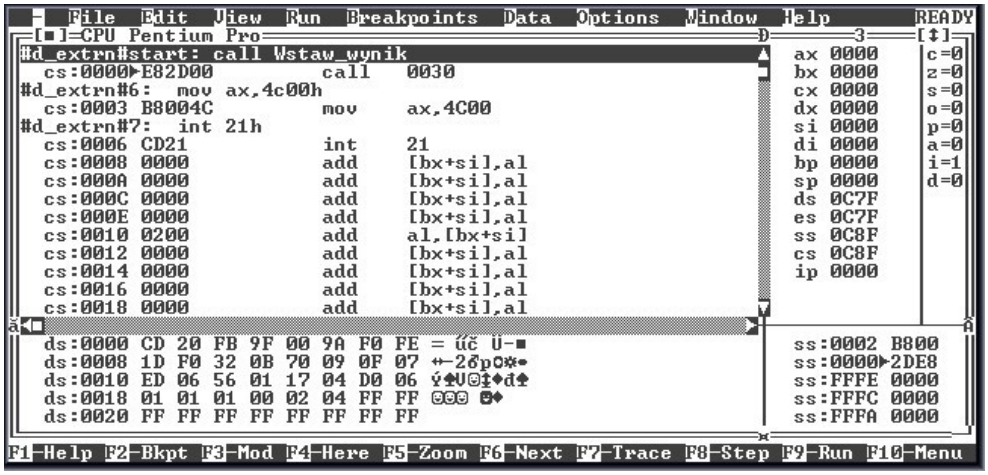
Po wpisaniu adresu znajdziemy się trochę dalej i głębiej w pamięci, tam na pewno poznamy nasz stary kod programu D1.asm, gdzie zamieszczona została procedura Wstaw_wynik z wywołaniem z jej wnętrza dalekiej procedury o nazwie dodaj, z programu D2.asm. Proszę spojrzeć na rys. 2.2.15.



Rys. 2.2.15. – Wejście w „głąb” do programu D1

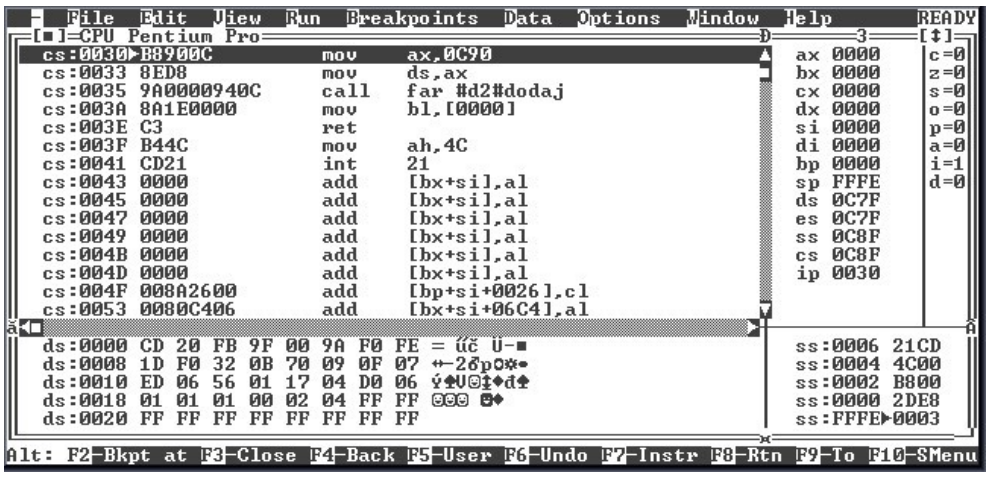
Na rys. 2.2.15. widać kolejne rozkazy programu D1, lecz tu mogą one występować tylko w postaci adresowej, gdyż nie ma bezpośredniego *podkładu* źródłowego tegoż programu; obraz ten pochodzi z pamięci.

Te dwa powyższe rysunki obrazują nam faktyczną sytuację w pamięci. Jednak w ten sposób (skakaniem po adresach) nie możemy wykonać tego programu D_extrn.exe. Jeśli chcemy to zrobić, wówczas musimy ponownie wyjść z TD, ponownie załadować nasz program i teraz zacząć od wykonywania programu, wchodząc do procedur. Oczywiście otrzymamy podobne obrazki jak poprzednio, z tym tylko, że teraz jesteśmy w trakcie wykonywania kodu, a nie tylko jego oglądania.



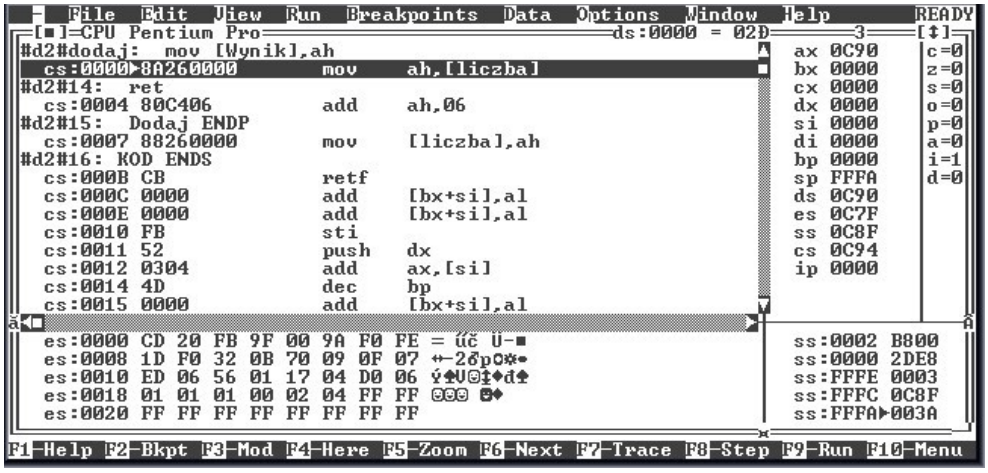
Rys. 2.2.16. – Okno CPU programu Turbo Debugger, *we wnętrzu* gotowy do wykonania program D_extrn.exe; okno ustawiono w opcji Mixed Yes

Po naciśnięciu klawisza F7 „wylądowaliśmy” w procedurze Wstaw_wynik w module D1, rys. 2.2.17.



Rys. 2.2.17. – Okno kodu programu D1

Wykonując dalej w taki sposób program, zaraz napotkamy rozkaz: call far #d2#dodaj (tak TD znaczy odniesienia do etykiet danego modułu programowego). Znowu przeskoczymy na drugą stronę „lustra”, czyli teraz do modułu D2 – rys. 2.2.18.

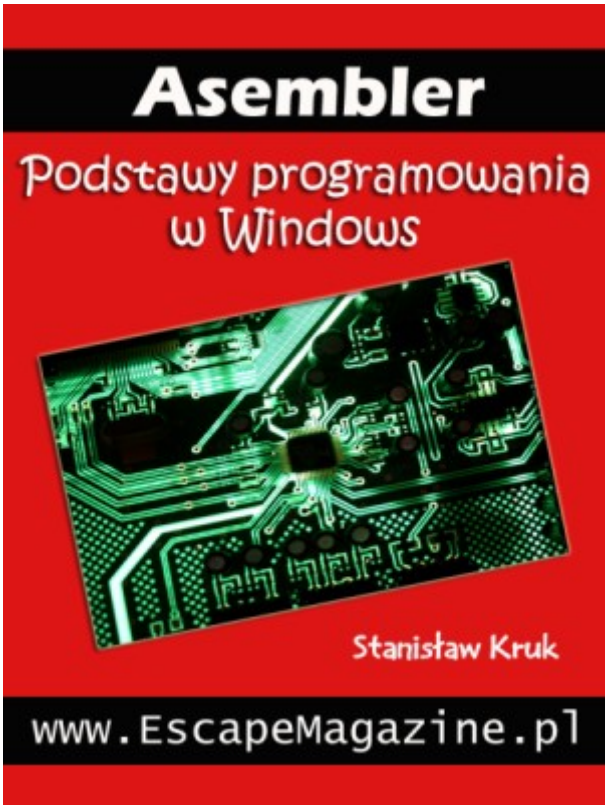


Rys. 2.2.18. – Okno kodu programu D2.

I co dalej? A dalej to w ramach ćwiczeń (nie tylko manualnych, lecz przede wszystkim umysłowych) można przekonać się, jak nasz program skacze po kodach obydwu modułów programowych. Gdy owo ćwiczenie zostanie wykonane poprawnie wówczas uznać można, że powolutku należy przechodzić do programowania w Asemblerze pod Windowsem, bo programować dla systemu DOS i *DOS-owo (turbo) debuggować* już umiemy.

Zanim jednak przywitamy się w asemblerowym Windowsie, musimy wcześniej zapoznać się, przynajmniej tak z grubsza, z nowym i zmienionym nieco środowiskiem programowym, Turbo Asemblerem (TASM32.EXE), Turbo Linkerem (TLINK32.EXE), Turbo Debuggerem (TD32.EXE).

Zobacz pełną wersję ebooka



http://www.escapemagazine.pl/301484-asembler-podstawy