STANISŁAW KRUK

ASEMBLER

PODRĘCZNIK UŻYTKOWNIKA

Labor Omnia Vincit

Wergiliusz

Projekt okładki: **Grzegorz Ławniczak**

Redakcja: **Irena Pielińska**

Skład komputerowy: **Dorota Świstak**

Książka uczy posługiwania się asemblerem pozwalając czytelnikom na zapoznanie się z podróżami po wnętrzu komputera. Asembler to narzędzie, które pozwala władać komputerem na poziomie niedostępnym dla przeciętnego użytkownika.

Autor kolejno opisuje rejestry komputera oraz jego oprogramowanie systemowe. Następnie wprowadza czytelników w zasady budowy programów w języku asemblera. Oddzielne miejsce zostało przeznaczone na omówienie przerwań.

Książkę zamykają dodatki, które dotyczą między innymi arytmetyki dwójkowej i szesnastkowej oraz prezentują listę rozkazów procesora Pentium. Dla zainteresowanych autor wraz z wydawnictwem MIKOM przygotowują kolejne książki na temat bardziej zaawansowanych zagadnień związanych z asemblerem.

Zastrzeżonych nazw firm i produktów użyto w książce wyłącznie w celu identyfikacji.

Wydawca: ZNI MIKOM, ul. Andrzejowska 3, 02-312 Warszawa, tel. 823-70-77

Druk: ZWP "HEL", ul. Grenadierów 77, Warszawa, tel. 810-12-71

ISBN 83-7158-181-5

Warszawa, maj 1999

Spis treści

Od Autora

Jest tak wiele rzeczy i zjawisk na świecie, iż często stajemy bezradni wobec tysięcy nowych pojęć. Próba dotknięcia ich naszymi zmysłami nie zawsze bywa udana. Może robimy to zbyt po amatorsku, a może brak nam wystarczającej wiedzy do zrozumienia tego, co próbujemy badać. Napiszmy literkę czy narysujmy kreskę na papierze. Nawet nie zdajemy sobie sprawy, ile w tym czasie musiało zajść w naszym wnętrzu, by stworzona została literka czy kreska. Rozbicie całości na części, na kawałki, po to tylko, by móc zrozumieć całość. Z pozoru przewrotna to myśl, chociaż po chwili dojdziemy do wniosku, iż tak nie jest. To jest normalne. Na pewno niejeden z nas młodych badaczy – czy dorosłych – badał świat „rozbierając go na części", na kawałki, niezależnie od tego, czy tym „światem" był zegarek, radioodbiornik, kalkulator, czy wreszcie, komputer. Nieważne, czego to badanie dotyczyło, ważne w tym wszystkim było to, iż uczyliśmy się dostrzegać, w jakim celu to zostało tak skonstruowane.

W komputerze naciskamy klawisz [Enter] czy klawisz myszki i od tej chwili zaczyna się coś dziać, komputer wyświetla, gra, bawi itp. Ale co takiego się stało, że tak mały impuls wywołał taką lawinę zjawisk. W fizyce opisującej zjawiska atmosferyczne efekt taki przyjęło się nazywać efektem motyla. Jeden drobny ruch skrzydeł motyla wywoła trąbę powietrzną kilkaset kilometrów dalej, zjawiska niestacjonarne, trudne do przewidzenia. W przypadku komputera może nie ma aż takich nieprzewidywalnych zjawisk, chyba że po [Enter] zaczyna się nieoczekiwane kasowanie dysku pełnego danych. Widocznie nieuważnie przeczytaliśmy jakieś polecenie lub ktoś zrobił nam brzydką „niespodziankę" bądź też ożywiliśmy potwora z głębin pamięci. Rodem jak z Asemblera. No właśnie, Asembler, to takie tajemnicze, niebezpieczne..., ale i chyba zarazem ciekawe narzędzie. Taka czarodziejska retorta. Musi być we właściwych rękach, aby coś złego się nie stało. Nie jest to takie zwykłe narzędzie, którym większość programistów się posługuje. Przy jego użyciu obejmiesz władzę nad całym komputerem, nad jego zasobami, nad dyskiem, pamięcią, nad „najodleglejszymi" i „ciemnymi" zakątkami Twojej maszyny. I o tym będzie ta książka, książka dla wszystkich.

Przyjemnej lektury życzy Autor

1. Podstawy Asemblera

1.1. Natura języka Asembler

Niemal wszyscy wiemy, że komputer to takie urządzenie, które posługuje się tylko dwoma cyframi, zerem i jedynką. Jak to jest jednak możliwe, że pomimo tak ubogiego alfabetu komputer może wytworzyć tyle pięknych obrazów, dźwięków, animacji. Cała tajemnica tego swoistego piękna tkwi w szybkości operowania przez maszynę owymi zerami i jedynkami.

Ale od początku. Każda maszyna cyfrowa ma swój mózg-procesor i serce-zegar. Im sprawniejszy mózg i im szybszy zegar, nadający rytm pracy maszyny, tym większą ma ona *siłę* do działania. Pisząc programy za pomocą alfabetu zerojedynkowego, sięgamy aż do najniższego poziomu maszyny, do jej mózgu i serca. Interesujące, prawda? Jest tylko jeden problem; trzeba wiedzieć, jak wpisywać te cyferki, i co najpierw, 0 czy 1? A może kilka jedynek naraz, a potem kilka zer, może jednak przeplatać je odpowiednio? Ale jak? Zero czy jedynka mają swoje zarezerwowane miejsca w ich ustalonym ciągu. Ale kto ustalił ten ciąg? Postaci ciągów zerojedynkowych ustalili twórcy, projektanci procesora, nadając im określone znaczenie, zgodne z jego budową i zastosowaniem. Czy to oznacza, że w kontakcie z procesorem *skazani* jesteśmy na wpisywanie zer i jedynek w odpowiedniej kolejności? Na to by wyglądało... I to ma być ten piękny oraz najskuteczniejszy ze wszystkich języków, język Asembler? To znaczy, jeżeli chcielibyśmy chwilowo zatrzymać procesor, musielibyśmy wpisać 10011011, nie strasząc już tu Czytelnika długimi „tasiemcami" zerojedynkowymi w przypadku bardziej złożonych sytuacji.

Od pewnego czasu nie jest już tak źle. Jak podają książki o historii Asemblera, podczas rywalizacji między mocarstwami zaprogramowano rakietę Vanquard – jako odpowiedź na Sputnika z 1957 r. – w języku wewnętrznym maszyny. Po prostu „nafaszerowano" jej wnętrze zerami i jedynkami. I cóż się stało. Ano, stało się, programista przeoczył zero czy jedynkę, po czym odbył się kosztowny spektakl na nieboskłonie. Czy ten programista dalej tam pracował? Nie wiadomo. W każdym razie poważnie rzecz potraktowano, wzięto pod uwagę omylność człowieka i sięgnięto po pomysł z początku lat 50. Pomysł był prosty; należało napisać program przekształcający monotonne ciągi zerojedynkowe, które oznaczałyby kody rozkazów procesora oraz ich adresy w pamięci, na symboliczne nazwy. Dokonano tego i okazało się, że to działa aż do dziś – tak powstał symboliczny asembler (jako program-translator pisany jest zawsze małą literką, aby nie mylić z nazwą języka Asembler). Wilk syty i owca

cała. Asembler jako język pozostał, zamieniono tylko żmudne i łatwe do pomylenia kody zerojedynkowe rozkazów procesora na nazwy symboliczne, tzw. mnemoniki. I tak np., zamiast wpisywać ciąg 10011011 wprowadzający procesor w stan oczekiwania, wpisywać będziemy słowo angielskie WAIT. A gdybyśmy się pomylili i ciąg 10011011 zamienili z ciągiem 11110100, spowodowalibyśmy spore zamieszanie, bo całkowicie zatrzymalibyśmy procesor. Nietrudno sobie wyobrazić, jakie praktyczne szkody może poczynić procesor, który nagle został „zwolniony z pracy" przez program.

1.2. Architektura sprzętowa komputera

W najbardziej ogólnych kategoriach komputer jest niczym innym jak urządzeniem, które przemieszcza dane z jednego miejsca w inne, czasami je przekształcając do różnych postaci. Posiada pięć „głów", a każda z nich odpowiada za swój odcinek pracy. Jedna „głowa" kontroluje klawiaturę, myszkę, dysk, druga – monitor, drukarkę, dysk, trzecia – czuwa nad działaniami arytmetyczno-logicznymi, czwarta sprawuje władzę nad pamięcią, a ostatnia, piąta, czuwa nad całością. Nad nią czuwa już tylko człowiek.

1.3. Powstawanie i rozwój języka Asembler

Nie jest przypadkiem, że rozkazy, które może wykonać procesor, ściśle odpowiadają akcjom w nim zachodzącym. Czym tak naprawdę jest rozkaz? Czym różni się on od danej, która jest przez niego przetwarzana? Rozkaz jest to elementarna operacja, jaką może wykonać mikroprocesor, i tak naprawdę niczym nie różni się od danej (danych). Oglądając w pamięci ciągi znaków (zapisanych w systemie szesnastkowym), nie potrafimy powiedzieć, który z tych znaków to rozkaz, a który to dana. Owszem, wprawne oko programisty-asemblerowca potrafi z dużym prawdopodobieństwem powiedzieć: – Teraz widzę (chyba) rozkaz, a teraz to dana tego (chyba) rozkazu. Jeśli do pamięci wczytany został „czysty jak łza" program, tzw. mapa pamięci (plik z rozszerzeniem .COM), to prawdopodobieństwo odgadnięcia tego, co jest czym, znacznie rośnie. Takie odgadywanie w Asemblerze, że z „upieczonego placka" określi się ilość i rodzaj składników, nazywa się deasemblacją (bądź desasemblacją) kodu wynikowego (wykonywalnego) na kod źródłowy. Każdy rozkaz posiada swoją wartość, program zaś jest niczym więcej, jak sekwencyjnie poukładanymi wartościami. Ale który rozkaz procesor będzie wykonywał jako następny? Muszą być jakieś wskaźniki pokazujące, gdzie ten rozkaz w pamięci się znajduje. Gdy następny rozkaz jest czytany z pamięci i wykonywany, wskaźnik ustawiany jest na następnym rozkazie. Niektóre rozkazy mogą ustawiać wskaźniki do nowych wartości; pozwala to procesorowi na niesekwencyjne wykonywanie szeregu rozkazów, uzależnione od określonych warunków. W języku Asembler posługujemy się rozkazami procesora, które

tak zostały skonstruowane i nazwane, by ich forma była zorientowana na człowieka. Krótko mówiąc, ludzki wymiar „złotych ścieżek procesora i jego rozkazów". Nie jest więc tak źle, jakby się mogło wydawać. Tak jak asembler (program do tłumaczenia) przekształca program źródłowy z jednego rodzaju tekstu zrozumiałego dla człowieka na tekst „zrozumiały" dla procesora, tak tenże procesor musi ów tekst jeszcze przerobić na tzw. język maszynowy, na zera i jedynki. Podczas gdy język Asembler i język maszynowy są sobie funkcjonalnie równoważne, to jednak język Asembler jest językiem dla ludzi. O wiele, wiele łatwiej zapamiętać rozkaz **ADD AL,3** wyrażony w postaci mnemonicznej, oznaczający dodawanie wartości **3** do rejestru, **AL**, niż gdyby rozkaz był zapisany w formie liczb **04** i **03** sekwencyjnie wprowadzonych w programie.

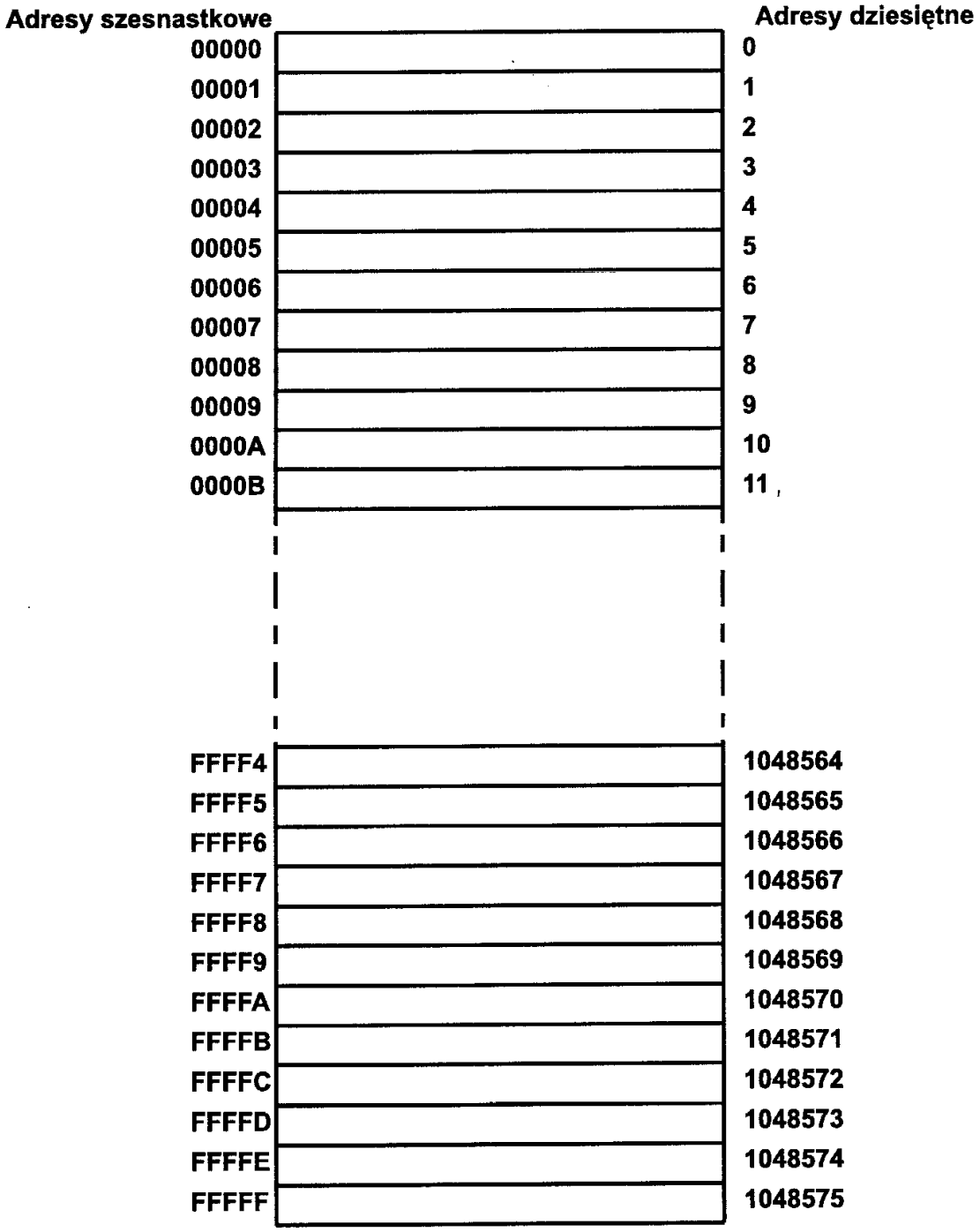
Niezwykle użyteczną stroną Asemblera jest to, iż pozwala on kontrolować akcje procesora jedną za drugą, maksymalnie efektywnie. Owszem, kod źródłowy w Asemblerze jest dłuższy niż w języku C czy Pascalu, ale żaden kod nie jest tak „szybki i zgrabny", jak kod programu w Asemblerze. Asembler to nie tylko język, który pozwala trzymać pełną kontrolę nad komputerem, ale to również filozofia stylu pracy procesora i jego otoczenia.

1.4. O użyteczności programów

Wszystkie procesory – od 8088/8086 aż do Pentium – należą do jednej rodziny procesorów iaPx86. Programistów piszących w Asemblerze interesuje tylko jedna rzecz, a mianowicie, czy asemblerowe programy, które napisano dla procesora 8088, będą równie dobre dla procesora Pentium? Tak, programy napisane dla pierwszego procesora 8088 będą dobre do najnowszych maszyn, ale tylko w sensie programu źródłowego. Jednakże, by programy te dały się uruchomić na tych najnowszych maszynach, trzeba je, przy użyciu odpowiednich programów tłumaczących, ponownie przetłumaczyć.

1.5. Procesory, pamięć i jej adresowanie; przechowywanie odwrotne

Procesor, wykonując programy, sięga po ich kody tam, gdzie się one znajdują, to znaczy do komórek pamięci. Musi on – niczym doręczyciel listów – dokładnie znać adres żądanego miejsca, by pobrać przesyłkę do nadania lub tylko ją przekazać. Procesor, jako główny zarządca, musi umieć komunikować się z pamięcią. Model 8086 zostaje połączony dwudziestoma „drutami", po których biegną dane oraz adresy komórek pamięci i urządzeń. Po jednym „drucie" może przesłać on 0 albo 1, a więc przy dwudziestu „drutach" może uczynić to na 2^{20}=1048576 możliwości. Wykorzystując wszystkie kombinacje ustawień 0 i 1 na 20 „drutach", można jednoznacznie wskazać (zaadresować) 1048576 (1 MB – 1 megabajt) komórek pamięci.



Rysunek 1.1. Przestrzeń adresowa procesorów 8086/8088

Czy oznacza to, iż asemblerowy programista, wkładając daną do pamięci bądź wskazując ją, musi za każdym razem wiedzieć i nieustannie śledzić, gdzie w danej chwili znajduje się ona w pamięci, a może tam już coś się znajduje, np. kod systemu operacyjnego lub programy rezydentne tak samo ważne jak i kod systemu operacyjnego. A gdyby nawet programista miał wiedzę, która komórka pamięci jest wolna i można tam przesłać daną, to jak ma tę komórkę zaadresować? Czy wolno mu wpisać bezwzględną wartość komórki w postaci: prześlij do 123786 komórki pamięci daną 1-bajtową? Taki bezwzględny zapis adresu komórki byłby nieprawidłowym zapisem. Aby procesor, a przede wszystkim programista, mogli względnie łatwo poruszać się

po 1 MB pamięci, podzielono tę pamięć na 16 segmentów, w każdym segmencie po 65536 (64 KB) jednobajtowych komórek. Adres dowolnej komórki pamięci będzie się teraz składał z dwóch części – numeru segmentu i położenia komórki w tym segmencie. W programie asemblerowym zapiszemy: prześlij daną do segmentu nr 10, a w tym dziesiątym segmencie do komórki nr 32456. Gdyby tak faktycznie było, to nadal nie rozwiązano by tego „problemu" adresowania możliwie prosto, tak jak oczekiwaliby tego programiści.

Programista po napisaniu kilku programów miałby serdecznie dosyć wszelkiego adresowania, nie mówiąc już o samej nauce języka i złożoności programowania, gdyby jeszcze nad nim zawisła taka „czarna chmura" niczym miecz Demoklesa. Procesor to niezwykły rygorysta. Ustawia w pamięci wszystko tak, jak to być powinno. Wszystko ma pod swoją kontrolą. Gdy spróbujemy mu coś narzucić na siłę, to na pewno tego nie przyjmie i w takich sytuacjach *zamrozi* całą maszynę, zawieszając jej działanie. Nie ma wówczas innego sposobu, jak zrobić maszynie „zimny prysznic" w postaci RESET. Ale częsty RESET może maszynie poważnie na „zdrowiu" zaszkodzić. Wewnątrz procesora wbudowanych zostało kilka (kilkanaście) rejestrów. Są to bardzo szybkie elementy elektroniczne mające zdolność pamiętania. Tylko niektórym z nich można narzucić swoje zdanie, inne rejestry pilnie słuchają swego procesora. Nawet asemblerowy programista, piszący pod wskazany adres pamięci, pełni jednak rolę specyficznego skazańca, skazanego na taki segment i takie miejsce w tym segmencie (tzw. offset), jakie zostaną mu „zaproponowane" przez procesor. Jest to jednak jedyny pozytywny przypadek skazania.

Samowola programisty, zwłaszcza asemblerowego, mogłaby spowodować wiele zniszczeń, spustoszeń w maszynie, a w najlepszym razie ją zawiesić. A więc procesor musi surowo pilnować, poprzez swoje rejestry, by programista nie wszedł mu w obszar segmentu, w którym znajduje się kod systemu operacyjnego czy coś równie ważnego (o rejestrach napisano w dalszej części tej książki). Programista zapisuje w programach, mniej lub bardziej jawnie, adres komórki (komórek) pamięci. Jest to adres, którego jedna część odnosi się do początku segmentu pamięci, a druga część do jakiegoś miejsca w tym segmencie, liczonego od początku tegoż segmentu. Ta druga składowa adresu nazywa się offsetem, zaś cały adres nazywany jest adresem logicznym, złożonym z pary oznaczanej w postaci: SEGMENT:OFFSET. Procesor korzystając ze swych „umiejętności", jak też z „umiejętności" swoich „kolegów po szynie", musi przekształcić adres logiczny na adres fizyczny, dla modelu 8086/8088 na adres 20-bitowy, dokonując przeliczeń według wzoru: **adres fizyczny = 10×segment+offset**; gdzie segment oznacza zawartość jednego z rejestrów segmentowych, natomiast offset zawartość rejestru wskazującego odległość od początku w tym segmencie, 10 – liczbę w zapisie szesnastkowym, równą 16 w zapisie szesnastkowym (heksadecymalnym); liczbę tę czyta się jako jeden i sześć, a nie szesnaście.

Intel 8086/8088 może odwołać się do pamięci operacyjnej o pojemności 1 MB – procesory te mają 20 wyprowadzeń adresowych. Dla większości przypadków wynik adresu bazowego (adresu fizycznego) segmentu i przemieszczenia (offsetu) będzie liczbą 20-bitową, jednakże może się zdarzyć, że dla niektórych wartości adresu bazowego i przemieszczenia wynik dodania tych wartości będzie liczbą 21-bitową. Gdy na przykład rejestr segmentowy ma wartość równą A000H (szesnastkowo), a rejestr wskaźnikowy ma wówczas wartość przemieszczenia (offsetu) równą 0140H, wówczas pełna wartość adresu fizycznego wynosić będzie 10xA000H+0140H=A0000H+0140H =A0140H=655680D (dziesiętnie) lub 10100000000101000000B (dwójkowo). W sytuacji gdy rejestr segmentowy, np. DS, zawierać będzie wartość 0FFFFH, a przemieszczenie jest równe 0FFFFH, dodanie tych wartości według opisywanego wzoru da wynik 0FFFF0H+0FFFFH= 10FFEFH. Wynik ten jest liczbą 21-bitową.

Gdyby było więcej linii adresowych, byłaby możliwość zaadresowania więcej niż 1 MB pamięci o 64 KB-17 B, gdyż 10FFEFH(1114095)-100000H (1048576) =65519 bajtów=64 KB-17 B.

W procesorach Intel 8086/8088 ze względu na 20-bitową szynę adresową bit 21 jest obcinany. Natomiast w wyższych procesorach, współpracujących z 24- czy 32-bitową szyną, 21 bit umożliwia zaadresowanie trochę więcej pamięci niż 1 MB o te właśnie 64 KB-17 B. W komputerach z procesorami 80286, 80386 i nowszymi obszar 64 KB-17 B nazywany jest pamięcią wysoką (ang. HMA).

Pamięć powyżej tej (pamięci) HMA, tzn. powyżej 1 MB+64 KB-17 B, nazywa się pamięcią rozszerzoną. Dostanie się do tej pamięci przy normalnym biegu pracy komputera z procesorem 80286 i nowszymi nie jest możliwe. Aby procesor mógł zaadresować tę pamięć, musi znaleźć się w trybie wirtualnym. 16-bitowy rejestr segmentowy zawiera teraz na trzynastu bitach tzw. selektor segmentu – wskaźnik do 8-bajtowej struktury opisującej dany segment, tzw. deskryptor segmentu, 2 bity związane są z prawami dostępu do segmentu, 1 bit określa, czy ów wskaźnik do 8-bajtowej struktury dotyczy tzw. tablicy lokalnej, czy globalnej. W tym trybie procesor uzyskuje zawrotne możliwości adresowania aż do 4 GB.

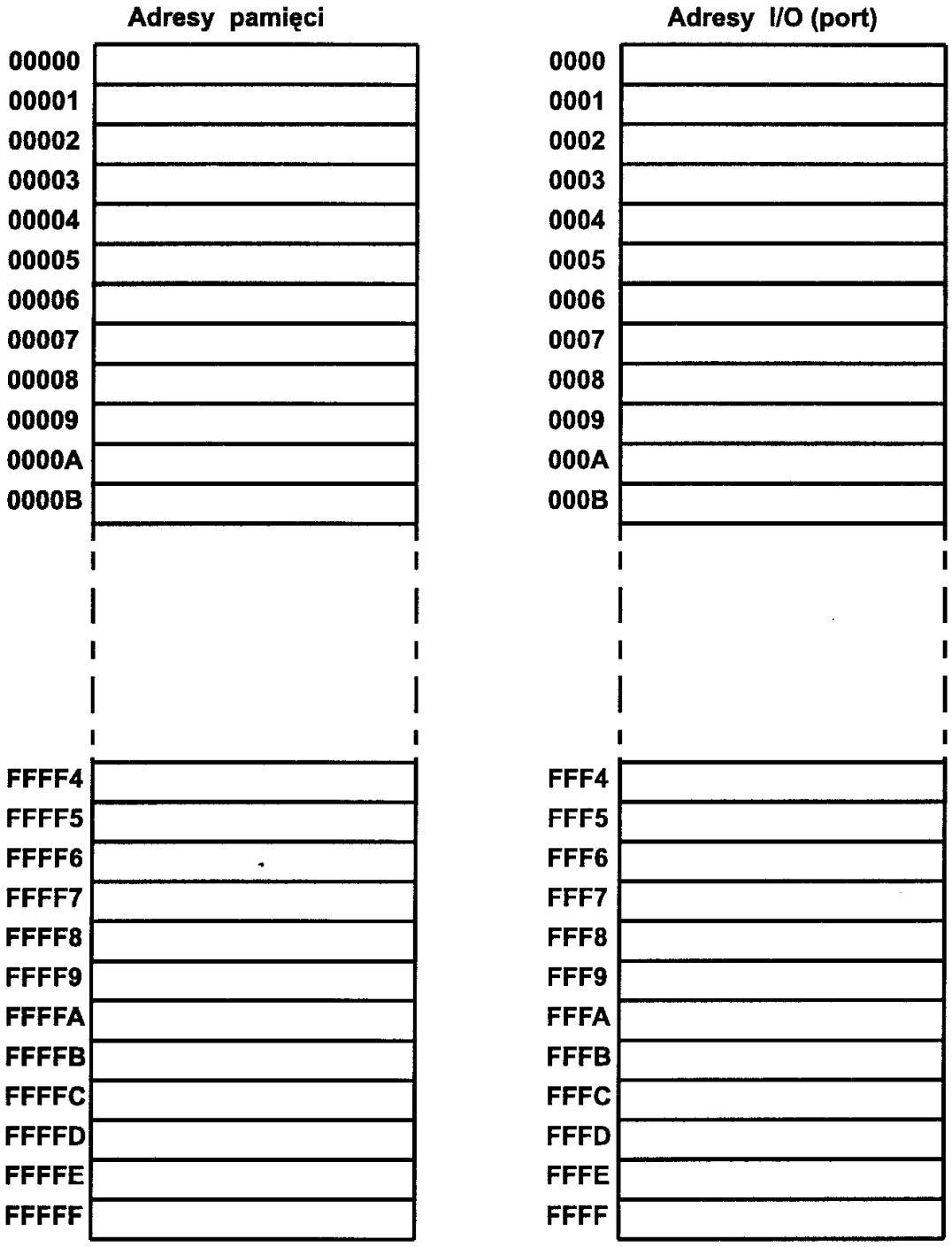
Procesory 32-bitowe składają swój adres logiczny z zawartości 16-bitowego rejestru segmentowego i 32-bitowego przemieszczenia zawartego w rejestrze offsetowym. Mają możliwość zaadresowania aż 64 TB (TB=terabajt) pamięci.

Pamięć komputera PC, mimo iż jest adresowana w jednostkach 8-bitowych (bajtach), to jednak wiele operacji wykonywanych jest na dłuższych niż 1 bajt porcjach bitów; są to słowa (dwa bajty), dwusłowa, poczwórne słowa itp. Jeśli do pamięci wpiszemy liczbę dziesiętną 1234, to okaże się, że cyfry 34 będą występować „wcześniej", pod młodszym adresem, niżeli cyfry 12. Gdy na przykład cyfry 34 występować będą pod adresem DS:0000, to cyfry 12 znajdą się o bajt dalej, pod adresem DS:0001, chociaż wydawałoby się, iż powinno być odwrotnie. Ten rodzaj przechowywania informacji w pamięci nazywa się przechowywaniem odwrotnym. Gdy pra-

cuje się z bajtami, słowami i jeszcze dłuższymi danymi, trzeba mieć się na baczności, by nie wprowadzić zamieszania i nie zapomnieć o odwrotnym przechowywaniu danych w pamięci.

1.6. Wejście/wyjście

Procesor 8086 obsługuje urządzenia wejścia i wyjścia w dwojaki sposób – za pomocą rozkazów wejścia/wyjścia (rozkazów I/O) i za pomocą adresowania pamięci. Niektóre wejścia i wyjścia urządzeń są kontrolowane przez porty, które określone są adresami I/O



Rysunek 1.2. Adresy pamięci i adresy I/O dla procesorów 8086/8088

w 64 KB przestrzeni adresowej oddzielonej od 1 MB przestrzeni adresowej pamięci (patrz rysunek 1.2). Przestrzeń adresowa I/O jest o wiele mniejsza niż 1 MB przestrzeń pamięci i dla 8086 ma wartość 64 KB; w rzeczywistości tylko 4 KB. Tym samym przestrzeń adresowa I/O nie jest używana do zapamiętywania wartości, ale raczej do zapewnienia właściwego sterowania urządzeniami wejścia/wyjścia, np.: modemem, drukarką, kartą dźwiękową itd. Adresy wejścia/wyjścia I/O mogą być dostępne za pomocą specjalnych rozkazów, IN i OUT, które tylko do tego celu służą. Na przykład rozkaz OUT DX,AL przesyła zawartość rejestru AL do portu I/O przez wybrany rejestr DX.

Niektóre wyjścia urządzeń mają odwzorowania w pamięci i są one kontrolowane przez adresy pamięci, a nie przez porty I/O. Tak częściowo rzecz się ma z monitorami, które mają swe odwzorowania w 1 MB pamięci, a równocześnie, dla niektórych ich funkcji, można nimi sterować za pomocą rozkazów I/O.

1.7. Przerwania; wektory przerwań

Przerwania są działaniami, za pomocą których zewnętrzne układy – w stosunku do jednostki centralnej, CPU, (ang. Central Procesor Unit) – sygnalizują zajście jakiegoś zdarzenia (np. wciśnięcia klawisza) i żądają określonego działania. System BIOS i system operacyjny (zwykle jest to DOS) wprowadzają swoje przerwania programowe, aby przywołać i wykonać specjalne programy obsługi (patrz BIOS – rozdział 3). Przyjrzyjmy się jednak najpierw mechanizmowi samego przerwania.

Gdy zachodzi przerwanie, sterowanie nad komputerem przejmuje program obsługi przerwań, który zwykle znajduje się w pamięci ROM. Program ten przywoływany jest przez załadowanie jego adresu segmentowego do odpowiednich rejestrów procesora; pełny adres zawarty jest w parze rejestrów CS i IP (patrz Rejestry – rozdział 2). Adresy segmentowe potrzebne do zlokalizowania programów obsługi przerwań nazwane zostały wektorami przerwania.

Wektory przerwań ustawione są w stan początkowy w czasie uruchamiania komputera i wskazują na programy obsługi przerwań zawarte w pamięci ROM (ROM, pamięć tylko do odczytu – patrz rozdział 3). Te wektory przerwań przechowywane są w postaci tabeli w pamięci RAM jako para słów z adresem względnym – pierwsze słowo, i z częścią segmentową – drugie słowo. (RAM – pamięć do odczytu i zapisu, początek obszaru konwencjonalnej pamięci RAM w komputerze PC zaczyna się od adresu 0, a kończy na 640 KB). Ta para słów przechowywana jest w pamięci w sposób odwrotny (np. odczytany w pamięci wektor przerwań 54FF00F0 należy czytać F000:FF54, SEGMENT:OFFSET). Wektory przerwań można zmieniać tak, aby wskazywały nowe programy obsługi przerwań.

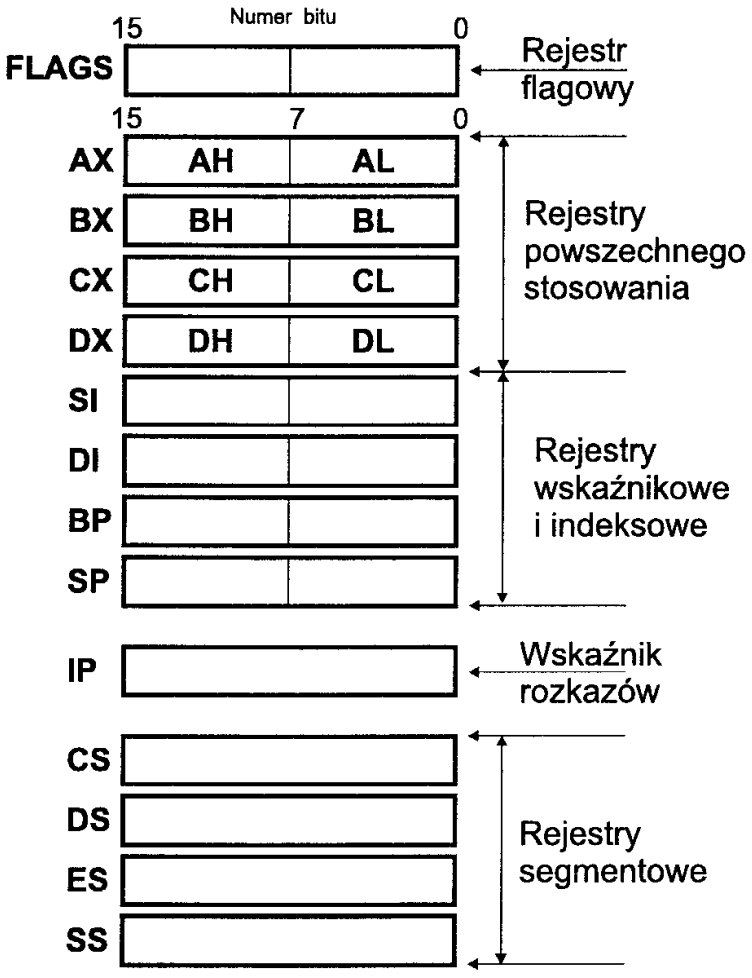
W rodzinie komputerów PC istnieją trzy główne kategorie przerwań. Pierwsza, to takie przerwania, które generowane są przez obwody komputera w odpowiedzi na jakieś zdarzenia, np. naciśnięcie klawisza. Przerwania te są obsługiwane przez ste-

rownik(i) przerwań 8259, który ustawia te przerwania w kolejności ich ważności, zanim prześle je do jednostki centralnej (CPU) w celu ich wykonania. Druga kategoria to przerwania generowane przez CPU jako pewnego rodzaju produkt uboczny wyjątkowych działań programu, np. dzielenie przez 0. Trzecia kategoria przerwań to przerwania celowo generowane przez programy jako sposób wywołania odległych podprogramów przechowywanych w pamięci RAM lub ROM. Przerwania te, zwane przerwaniami programowymi, są częścią składową ROM-BIOS i systemu operacyjnego. Możliwa jest zmiana programów obsługi przerwań bądź napisanie nowych (gdy zaistnieje taka potrzeba). Aby w programie przywoływać przerwania programowe, należy używać rozkazu INT z argumentem określającym numer przerwania, liczbą od 0 do 255. Na przykład: INT 10H – przerwanie należące do systemu BIOS obsługujące ekran monitora, INT 21H – przerwanie należące do systemu operacyjnego DOS zawierające różne usługi systemowe itp. (patrz rozdział – Tablica wektorów przerwań).

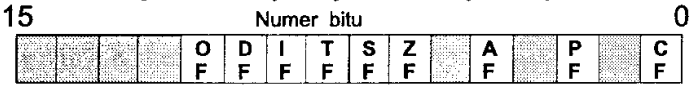
Istnieje jeszcze jeden, specjalny typ przerwań, zwany przerwaniami niemaskowalnymi (NMI – ang. *non-maskable interrupts*), stosowany do wywołania natychmiastowej uwagi jednostki centralnej. Przerwania te najczęściej sygnalizują jakieś sytuacje niebezpieczne, np. spadek napięcia, błąd pamięci. Gdy przesyłane jest to przerwanie, zachowywane jest bezwzględne pierwszeństwo i jednostka centralna, CPU, przetwarza je przed wszystkimi innymi przerwaniami. Niezależnie jednak od tego, w jaki sposób przerwanie zostało wytworzone, urządzenie wywołujące je nie musi znać adresu pamięci odpowiedniej obsługi przerwania, musi znać tylko numer przerwania. Numer przerwania wskazuje tabelę przechowywaną od najniższych adresów pamięci, zawierającą adres segmentowy podprogramu obsługi przerwania. Ten adres obsługi przerwania nazywa się wektorem przerwania. Gdy sami tworzymy nowe podprogramy obsługi przerwań, wówczas nadajemy im już istniejące numery wektory przerwań (*podpinamy* procedurę pod istniejący numer przerwania) albo nadajemy im nowe numery.

2. Rejestry

Jak już wcześniej wspomniano, rejestry to bardzo szybkie elementy elektroniczne posiadające zdolność zapamiętywania informacji. Wewnątrz 8086/8088 znajduje się 14 rejestrów, za pomocą których można przesyłać i przetwarzać dane. Te wewnętrzne



Poszczególne bity rejestru flagowego



OF - flaga nadmiaru (ang. *overflow flag*)
DF - flaga kierunku (ang. *direction flag*)
IF - flaga zezwolenia na przerwanie (ang. *Interupt enable flag*)
TF - flaga pracy krokowej (ang. *trap flag*)
SF - flaga znaku (ang. *sign flag*)
ZF - flaga zera (ang. *zero flag*)
AF - flaga przeniesienia pomocniczego (ang. *auxiliary flag*)
PF - flaga parzystości (ang. *parity flag*)
CF - flaga przeniesienia (ang. *carry flag*)

Rysunek 2.1. Rejestry procesora 8086

rejestry, rozciągające się na obszarze 28 bajtów, są w stanie czasowo przechowywać dane, adresy pamięci, wskaźniki rozkazów i stanów oraz znaczniki (flagi) sterujące; poprzez nie procesor ma dostęp do ponad 1 MB pamięci i do 64 KB portów I/O. Zasadniczo każdy z rejestrów ma swoje zadanie do spełnienia, każdy z nich pełni swoją rolę i dają się one pogrupować według podobnych zadań:

- rejestry powszechnego zastosowania,
- rejestry segmentowe,
- rejestry wskaźnikowe i indeksowe,
- wskaźnik rozkazów,
- rejestr znaczników (rejestr flagowy) – FLAGS.

2.1. Rejestry powszechnego zastosowania

Osiem rejestrów powszechnego zastosowania (każdy o długości 16 bitów) są używane do najczęściej stosowanych rozkazów, jako miejsce, skąd pobieramy dane, miejsce przeznaczenia, wskaźniki do pamięci i wreszcie jako liczniki. Każdy z tych ośmiu rejestrów może być załadowany zarówno z pamięci, jak też z nich można do pamięci załadować, można ich używać do operacji arytmetycznych i logicznych. Na przykład:

```
...
MOV AX,89          ;prześlij do rejestru AX daną 89
MOV DX,10          ;prześlij do rejestru DX daną 10
ADD AX,DX          ;dodaj te dwie liczby 89 i 10, a wynik prześlij do rejestru AX
...
```

Rejestry CX, SI lub inne rejestry powszechnego zastosowania mogłyby zastąpić użyte w tym przykładzie rejestry AX lub DX z równie dobrym powodzeniem.

Pomimo wspólnych cech rejestrów powszechnego zastosowania, każdy z nich, z osobna, ma swoją „osobowość".

- **Rejestr AX** – znany jako akumulator. Używany zawsze tam, gdzie zachodzi mnożenie, dzielenie. Jest to najbardziej efektywny rejestr używany w operacjach arytmetycznych, logicznych, przesyłania danych. Dolna, 8-bitowa część rejestru AX nosi nazwę AL (ang. *A-Low*), część górna, też 8-bitowa, nosi nazwę AH (ang. *A-High*). Taki podział rejestru na dwie 8-bitowe części jest wygodny podczas działań na danych 1-bajtowych, tworząc dwa niezależne rejestry.

```
...
MOV AH,1           ;prześlij do rejestru AH wartość 1
MOV AL,AH          ;kopiuj wartość AH do AL
DEC AL             ;odejmij (zmniejsz, dekrementuj) o 1 zawartość rejestru AL
...
```

W wyniku tych rozkazów rejestr AX ma wartość 1.

W powyższym przykładzie rejestry BX, CX i DX mogłyby też wziąć udział jako rejestry 16-bitowe lub jako dwa rejestry 8-bitowe.

- **Rejestr BX** – może wskazywać położenie, lokalizację w pamięci. 16-bitowa wartość zapamiętana w tym rejestrze może być po części użyta do adresowania położenia w pamięci. Na przykład, do rejestru AL ładowana jest zawartość pamięci spod (umownego) adresu 6:

```
...
MOV AX,0                ;prześlij do rejestru AX wartość 0
MOV DX,AX               ;kopiuj wartość AX do DX
MOV BX,6                ;prześlij do rejestru BX wartość 6
MOV AL,[BX]             ;prześlij do rejestru AL zawartość pamięci spod adresu 6
...
```

Domyślnie rejestr BX, wraz z rejestrem segmentowym DS, jest używany jako wskaźnik pamięci. Rejestr BX może być traktowany jako dwa 8-bitowe rejestry – BH i BL.

- **Rejestr CX** – używa się go głównie jako licznika odliczającego powtarzające się fragmenty programów bądź pojedynczych rozkazów. Na przykład:

```
...
MOV CX,5                ;prześlij do rejestru CX wartość 5
Zaczynaj_petle:         ;etykieta o nazwie Zaczynaj_petle
<. . .>                 ;powtarzane rozkazy
SUB CX,1                ;odejmuj (za każdym razem) od rejestru CX wartość 1
jnz Zaczynaj_petle      ;jeśli nie zero (w CX) skocz do etykiety Zaczynaj_petle
...
```

Przytoczony wyżej fragment programu, przedstawiający sposób tworzenia pętli, może też być zorganizowany w inny sposób, przy użyciu innego rozkazu **LOOP**. Rozkaz **LOOP** odejmuje 1 od rejestru CX, następuje skok, jeśli CX nie jest równe zeru, jak w poniżej podanym przykładzie:

```
...
MOV CX,5                ;prześlij do rejestru CX wartość 5
Zaczynaj_petle:         ;etykieta o nazwie Zaczynaj_petle
<. . .>                 ;powtarzane rozkazy
loop Zaczynaj_petle     ;jeśli nie zero (w CX) skocz do etykiety Zaczynaj_petle
...
```

Rejestr CX może być traktowany jako dwa 8-bitowe rejestry, CH i CL.

- **Rejestr DX** – jego głównym przeznaczeniem jest użycie go jako wskaźnika adresów w rozkazach **IN** i **OUT** – rozkazy I/O (wejścia/wyjścia). Nie ma bowiem

innej drogi do zaadresowania portów, aniżeli użycie rejestru DX. Następujący przykład pokazuje zapis danej o wartości 35 do portu o numerze 45:

```
...
MOV AL,35          ;prześlij do rejestru AL wartość 35
MOV DX,45          ;prześlij do rejestru DX wartość 45, nr portu
OUT DX,AL          ;wyprowadź bajt z portu 45 do rejestru AL
...
```

Rejestr DX może być użyty w operacjach mnożenia i dzielenia. Gdy dzielimy 32-bitową dzielną przez 16-bitowy dzielnik, „górne" 16 bitów dzielnej musi być umiejscowione w DX; po dzieleniu reszta z dzielenia zapamiętywana zostaje w DX. („Dolne" 16 bitów dzielnej musi być umieszczone w rejestrze akumulatora AX, iloraz zapamiętany jest w AX). Podobnie, gdy mnożymy dwie 16-bitowe liczby; „górne" 16 bitów iloczynu jest zapamiętane w DX („dolne" 16 bitów iloczynu zapamiętane jest w AX). Rejestr DX może być traktowany jako dwa 8-bitowe rejestry, DH i DL.

2.2. Rejestry wskaźnikowe i indeksowe

- **Rejestr SI** – podobnie jak rejestr BX, może być użyty jako wskaźnik pamięci. Na przykład:

```
MOV AX,0           ;prześlij do rejestru AX wartość 0
MOV DS,AX          ;prześlij do rejestru DX zawartość rejestru AX
MOV SI,18          ;prześlij do rejestru SI wartość 18
MOV AL,[SI]        ;prześlij do rejestru AL zawartość pamięci spod adresu
                   DS:SI
```

Natomiast ciąg rozkazów...

```
...
CLD                ;znacznik kierunku DF przyjmuje wartość 0
MOV AX,0           ;prześlij do rejestru AX wartość 0
MOV DS,AX          ;prześlij do rejestru DS zawartość rejestru AX
MOV SI,18          ;prześlij do rejestru SI wartość 18
LODSB              ;prześlij bajt spod adresu DS:SI do rejestru AL
...
```

... nie tylko ładuje rejestr AX (część AL), ale i dodatkowo zwiększa rejestr SI o 1 (DF=0).

Taki ciąg rozkazów może być bardzo efektywny podczas wykonywania rozkazów łańcuchowych, w których przesyłane są ciągi tekstów, znak po znaku.

- **Rejestr DI** – jest bardzo podobny w użyciu do rejestru SI. Może być użyty jako wskaźnik pamięci; ma specjalne własności, gdy zostanie zastosowany w rozkazach związanych z łańcuchami znakowymi. Na przykład:

```
...
MOV AX,0                ;prześlij do rejestru AX wartość 0
MOV DS,AX               ;prześlij do rejestru DS zawartość rejestru AX
MOV DI,1024             ;prześlij do rejestru DI wartość 1024
ADD BL,[DI]             ;dodaj do BL zawartość pamięci spod adresu DS:DI
...
```

Rejestry DI i SI wespół z rejestrem DS związane są z adresowaniem łańcuchów znakowych, z tym że rejestr DI występuje zawsze, gdy adresowanie dotyczy źródła danych, zaś rejestr SI występuje wówczas, gdy adresowanie to dotyczy przeznaczenia (celu). Rejestry SI i DI użyte jako wskaźniki pamięci z nie łańcuchowymi rozkazami odnoszą się zawsze względem rejestru segmentowego DS. Na przykład:

```
...
CLD                     ;znacznik kierunku DF przyjmuje wartość 0
MOV DX,0                ;prześlij do rejestru DX wartość 0
MOV ES,AX               ;prześlij do rejestru ES zawartość rejestru AX
MOV DI,2048             ;prześlij do rejestru DI wartość 2048
STOSB                   ;prześlij bajt z rejestru AL do pamięci adresowanej rejestrami ES:DI
...
```

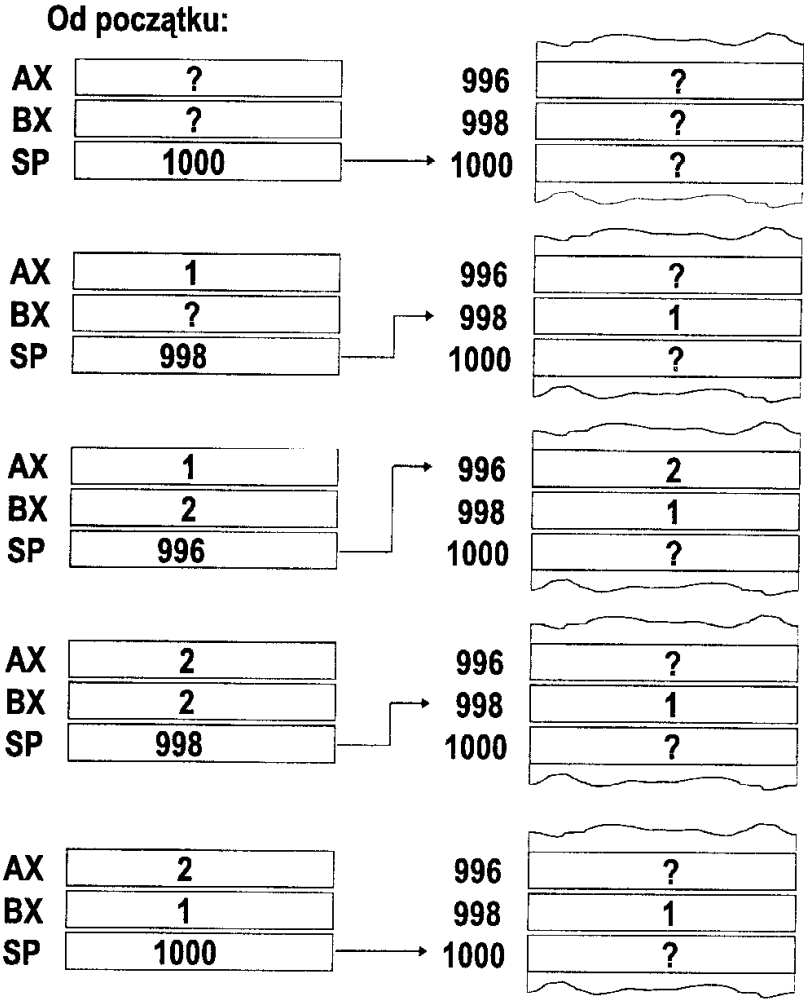
- **Rejestr BP** – podobnie jak BX, SI, DI – może być użyty jako wskaźnik pamięci, ale z pewną różnicą. Podczas gdy rejestry BX, SI, DI, wskazując na adres w pamięci odnoszą się względem rejestru segmentowego DS, to rejestr BP, służąc za wskaźnik pamięci, odnosi się do rejestru SS, rejestru segmentu stosu. Adresowanie poprzez segment stosu, stosowane zwłaszcza w procedurach języków C, Pascal, dokonuje się właśnie przy użyciu rejestru BP. Na przykład:

```
...
PUSH BP                 ;odłóż na stos zawartość rejestru BP
MOV BP,SP               ;prześlij do rejestru bp zawartość rejestru SP
MOV AX,[BP+4]           ;prześlij do rejestru AX zawartość stosu spod adresu SS:[BP+4]
...
```

- **Rejestr SP** – znany jest jako wskaźnik stosu i należy do ostatnich rejestrów powszechnego stosowania. Rejestr SP daje położenie bieżącego wierzchołka stosu i jest analogiczny do IP. Umieszczanie wartości na stosie (ang. *pushing*) dokonywane jest za pomocą rozkazu PUSH, a zdejmowanie wartości ze stosu (ang. *popping*) odbywa się przy użyciu rozkazu POP. Przykład wraz z rysunkiem ilustrują zmiany w rejestrach SP, AX i BX i na stosie, podczas wykonywania się kodu; przyjmujemy, że początkowy stan rejestru SP ma wartość 1:

```
...
MOV AX,1                ;prześlij 1 do rejestru akumulatora AX
```

```
PUSH AX          ;odłóż na stos zawartość rejestru AX
MOV BX,2         ;prześlij do rejestru BX zawartość 2
PUSH BX          ;odłóż na stos zawartość rejestru BX
PUSH AX          ;odłóż na stos zawartość rejestru AX
PUSH BX          ;odłóż na stos zawartość rejestru BX
...
```



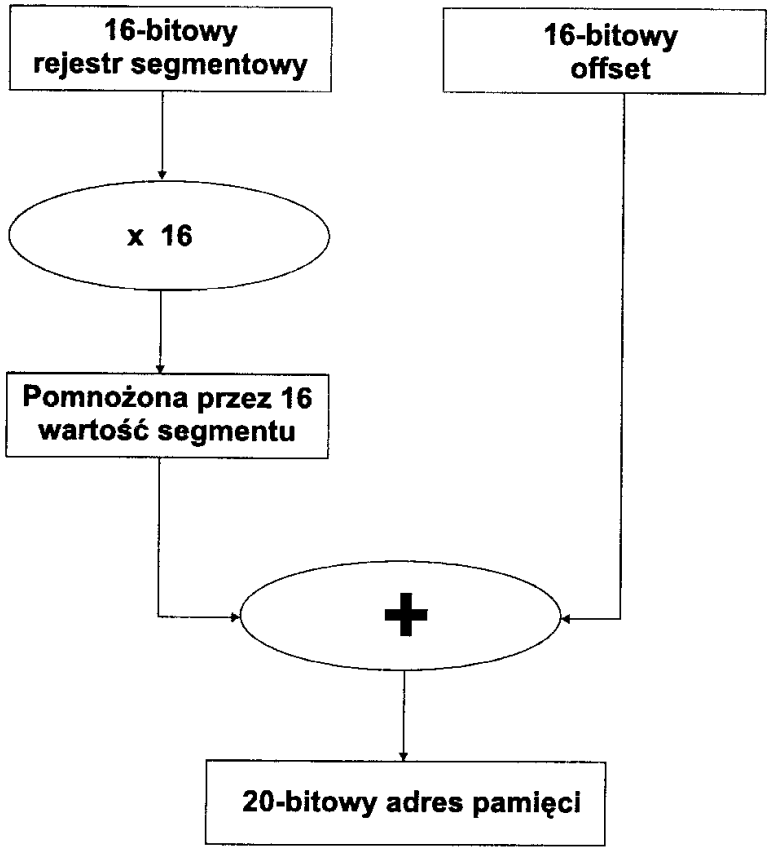
Rysunek 2.2. Rejestry AX, BX, SP i stos

Jeśli zmieniamy rejestr SP, to zmieniamy położenie wierzchołka stosu i wówczas możemy szybko doprowadzić do katastrofy. Zmiany na stosie mogą być dokonywane jedynie za pomocą rozkazów operujących na stosie. Za każdym razem, gdy wywołujemy podprogram lub wracamy zeń do programu, stos jest używany. Reasumując, modyfikację rejestru SP zostawmy rozkazom operującym na stosie, wywołaniom procedur i powrotom z nich do programów; nie dokonujmy bezpośredniej modyfikacji zawartości tego rejestru.

2.3. Rejestry segmentowe

Podstawową przesłanką segmentacji pamięci jest to, iż procesor 8086 ma zdolność adresowania 1 MB pamięci. 20-bitowy sposób adresowania pamięci wystarcza,

by każda komórka 1 MB pamięci została zaadresowana. 8086 używa tylko 16-bitowego wskaźnika do pamięci (dla przykładu należy przypomnieć, że 16-bitowy rejestr BX może być użyty do wskazania w pamięci). Jak wobec tego pogodzić 16-bitowe wskaźniki z 20-bitową przestrzenią adresową? Odpowiedź jest następująca: 8086 składa pełny 20-bitowy adres z dwóch części. Każdy 16-bitowy wskaźnik lub offset pamięci jest składany z zawartością rejestru segmentowego do formy pełnego 20-bitowego adresu. Wartość segmentu jest przesuwana o 4 pozycje w lewo (mnożona przez 16) i dopiero wówczas jest dodawana do offsetu, tak jak pokazano to na rysunku.



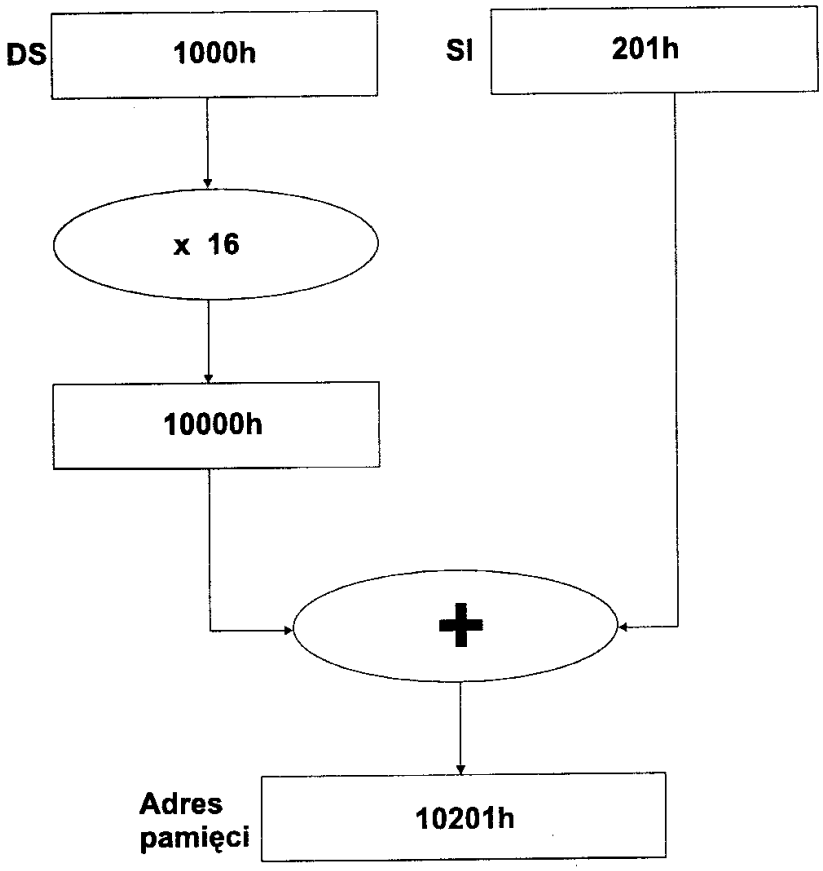
Rysunek 2.3. Obliczanie adresów pamięci – schemat ogólny

Weźmy pod uwagę następujący przykład:

```
...
MOV AX,1000H          ;prześlij 1000h do rejestru akumulatora AX
MOV DS,AX             ;prześlij z rejestru AX do rejestru segmentowego DS
MOV SI,201H           ;prześlij 201h do rejestru SI
MOV DL,[SI]           ;prześlij do rejestru DL zaw. pamięci spod adresu DS:201h
...
```

Do rejestru DL ładowana jest wartość spod adresu ((DS*16)+SI), czyli w tym przykładzie ((1000H*16)+201H)=10000H+201H=10201H szesnaście (dziesiętnie) w systemie szesnastkowym ma wartość 10H.

Poprzedni, ogólny schemat adresowania nabierze teraz konkretnej treści – w sposób pokazany na rysunku.



Rysunek 2.4. Obliczanie adresów pamięci

Podczas programowania musimy zawsze używać adresu pamięci w postaci pary SEGMENT:OFFSET. Wszystkie rozkazy i tryby adresowania procesora 8086 domyślnie odnoszą się do jednego lub wielu rejestrów segmentowych, jednakże niektóre rozkazy mogą być jawnie przypisane do takiego rejestru segmentowego, jakiego sobie życzymy. Rzadko ładujemy rejestry segmentowe za pomocą liczb, natomiast ładujemy je wpisując w programie źródłowym ich nazwy, które następnie przekształcane są w liczby w czasie asemblacji, konsolidacji i uruchamiania programu. Użycie nazw segmentów umożliwia właściwą współpracę między systemem operacyjnym a Asemblerem. W Turbo Asemblerze, o którym tu głównie będzie mowa, najczęściej przywoływaną nazwą segmentu jest **@data**, wskazująca na domyślny segment danych; gdy użyto uproszczonych dyrektyw.

W przykładzie:

```
.MODEL small
.DATA
Zm1 DW 0
...
.CODE
mov ax,@data
mov ds,ax
...
END
```

DS ładowany jest wartością domyślnego segmentu danych, w którym mieści się zmienna Zm1. Użycie segmentów w procesorze 8086 ma wiele implikacji. Po pierwsze, tylko 64 KB bloki pamięci dają się adresować poprzez rejestry segmentowe, ponieważ 64 KB jest maksymalną ilością pamięci, która może być zaadresowana przy użyciu 16-bitowego offsetu (2^{16}=65536:1024=64); ten sposób adresowania jest, niestety, przeszkodą do manipulowania większymi (niż 64 KB) blokami danych. Po drugie, dane położenie w pamięci może być zaadresowane na wiele kombinacji SEGMENT:OFFSET. Na przykład, adres 100H może być zapisywany: 0:100H, 1:F0H, 2:E0H itd.

Tak jak rejestry powszechnego stosowania odgrywają swoje specyficzne role, tak też rejestry segmentowe przeznaczone są do wyspecyfikowanych zadań.

Rejestr CS wskazuje kod programu, rejestr DS wskazuje dane, rejestr SS związany jest ze stosem, zaś rejestr ES (ang. *extra segment*) jest dodatkowym rejestrem, który może być gdzieś potrzebny.

- **Rejestr CS** – wskazuje na początek 64 KB bloku pamięci lub na segment kodu, w którym rezyduje następny do wykonania rozkaz. Dokładne położenie tego rozkazu w segmencie kodu wskazywane jest poprzez offset, którego wartość zawiera rejestr IP. Pełny adres położenia rozkazu w segmencie kodu ma postać: CS:IP. Procesor 8086 nigdy nie pobiera rozkazów z segmentu innego niż segment CS. Rejestr CS może być zmieniony przez niektóre rozkazy, włączając w to rozkazy skoków, wywołań i powrotów. Rejestru tego nie można ładować bezpośrednio.
- **Rejestr DS** – wskazuje początek segmentu danych, czyli 64 KB blok pamięci zawierający argumenty. Zazwyczaj rejestrami stowarzyszonymi z DS, określającymi offset w tym segmencie są rejestry BX, SI lub DI.
- **Rejestr ES** – wskazuje początek 64 KB bloku pamięci, zwanego dodatkowym segmentem. Jak wskazuje nazwa segmentu, (ang. *extra segment*), nie jest on przypisany do pojedynczych zastosowań; stosowany jest do różnych pojawiających się potrzeb. Niekiedy ten *extra segment* użyty jest do utworzenia dodatkowego 64 KB bloku pamięci potrzebnego do przechowywania danych, jednakże z nieco mniejszą dostępnością do tych danych aniżeli z przeznaczonego w tym celu segmentu danych. Rejestr ES używany jest w rozkazach łańcuchowych. Wszystkie te rozkazy (łańcuchowe), które zapisują do pamięci, używają pary rejestrów ES:DI. Rejestr ES używany jest w tych rozkazach wszędzie tam, gdzie dokonywane są operacje na blokach pamięci, kopiowanie, porównywanie, przeszukiwanie, czyszczenie.
- **Rejestr SS** – wskazuje początek 64 KB bloku pamięci, zwanego segmentem stosu. Wszystkie rozkazy, które niejawnie używają rejestru SP – odkładanie na stos, zdejmowanie ze stosu, wywołania i powroty – używają segmentu stosu, ponieważ rejestr SP jest zdolny tylko do adresowania pamięci w obszarze segmentu

stosu SS. Również rejestr BP – o czym była już mowa – odnosi się też do segmentu stosu; stąd też rejestr BP jest używany do adresowania parametrów i zmiennych zawartych na stosie.

2.4. Wskaźnik rozkazów

- **Rejestr IP** – nazywany jest wskaźnikiem rozkazów i zawiera zawsze offset pamięci, w którym zawarty jest następny rozkaz do wykonania. Bazowy adres segmentu kodu zawarty jest w rejestrze CS. Pełny adres logiczny wykonywanego rozkazu wskazywany jest więc przez parę rejestrów CS:IP.

Gdy jeden rozkaz jest wykonywany, wskaźnik rozkazów ustawiany jest do następnego adresu pamięci, pod którym znajduje się rozkaz do wykonania. Zazwyczaj następnym rozkazem w pamięci jest właśnie rozkaz, który będzie wykonywany. Jednakże niektóre rozkazy, takie jak rozkazy wywołania i skoku, mogą spowodować rozgałęzienie w wykonywaniu kodu programu, a tym samym w rejestrze IP nie wystąpi kolejna wartość offsetu wskazująca następny rozkaz do wykonania.

2.5. Rejestr znaczników

- **Rejestr znaczników (rejestr flagowy) – FLAGS** jest czternastym i ostatnim rejestrem procesora 8086. Rejestr ten jest zbiorem poszczególnych bitów kontrolnych, zwanych znacznikami (flagami), które wskazują wystąpienie określonego stanu. Znaczniki mogą być wykorzystywane zarówno przez procesor, jak też i programistę. Niedoświadczony programista nie powinien dokonywać jakichkolwiek zmian stanu znaczników rejestru FLAGS, jeśli nawet potrafi dostać się do tego rejestru (patrz: Rysunek 2.1. Rejestry procesora 8086, oraz Tabela 5.1. Stan znaczników (flag) procesora).

3. Oprogramowanie systemowe DOS i BIOS

Oprogramowanie systemowe kieruje całą złożonością połączeń występujących pomiędzy poszczególnymi urządzeniami. Dwoma głównymi składnikami tego oprogramowania są BIOS (Basic Input/Output System) i DOS (ew. Windows).

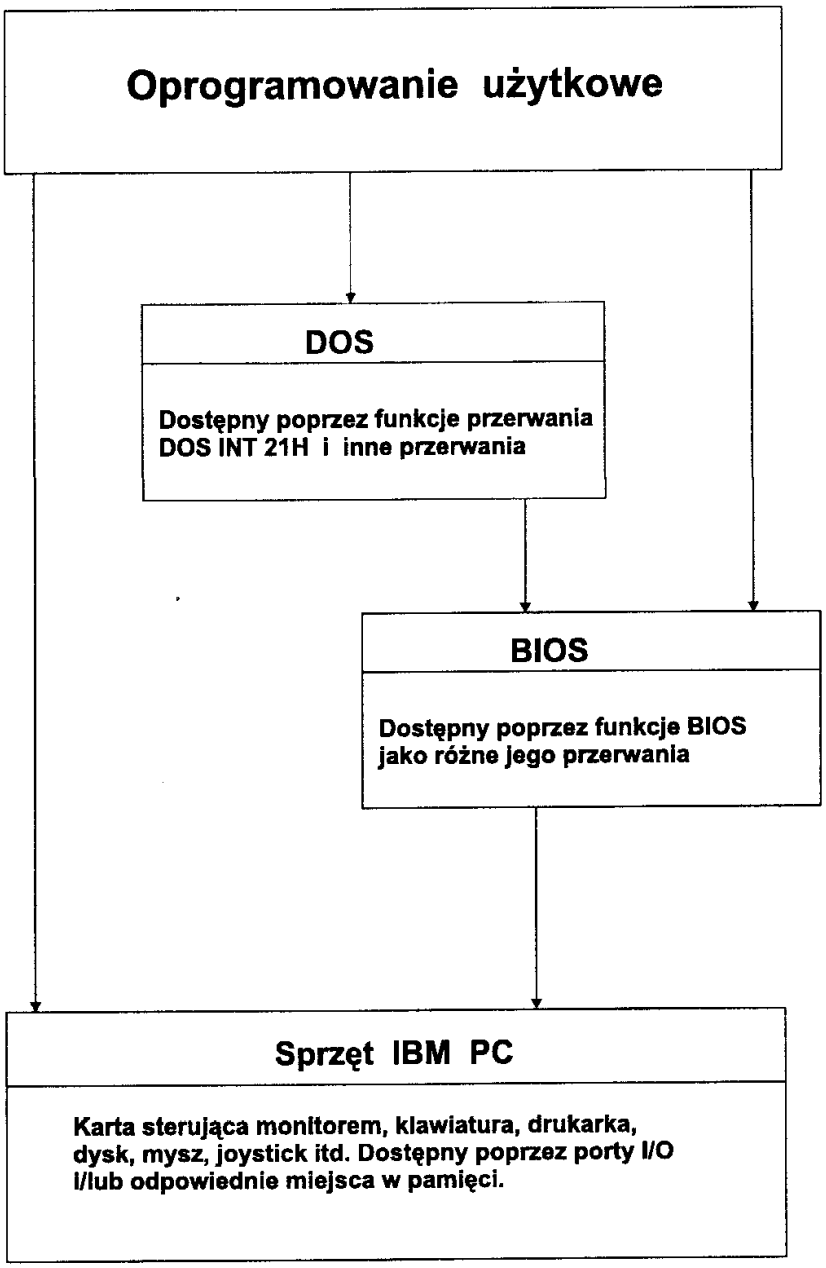
Programy zawarte w pamięci BIOS logicznie umieszczone są między naszymi programami, wraz z DOS/Windows, a sprzętem. BIOS z jednej strony otrzymuje od (naszych) programów wykonania standardowych usług związanych z obsługą urządzeń wejścia/wyjścia. Usługi te wzywane są przez programy za pomocą kombinacji numeru przerwania (przerwania BIOS) i numeru usługi. Użycie tej kombinacji wskazuje na rodzaj usługi, np. obsługa drukarki. W drugą stronę BIOS komunikuje się z urządzeniami komputera, monitorem, dyskami itd., używając odpowiednich dla danego urządzenia kodów rozkazów. Od tej strony BIOS obsługuje też przerwania sprzętowe, które są generowane przez urządzenia „chcące zwrócić na siebie uwagę". Na przykład, naciśnięcie klawisza powoduje, że klawiatura generuje przerwanie, by o tym co zaszło, zaraz zawiadomić BIOS.

Oprogramowanie BIOS zapisane jest w kostce pamięci ROM, znane pod nazwą ROM-BIOS (Read Only Memory – Basic Input/Output System) i można odczytać je tylko jako ciąg asemblerowych rozkazów. DOS (Disc Operating System) ew. Windows jest programem, który kontroluje komputer i jego zasoby od momentu jego włączenia aż do wyłączenia.

Poprzez funkcje DOS programy użytkowe mogą czytać pliki, zapisywać je do pamięci, kontrolować naciskanie klawiszy klawiatury, uruchamiać inne programy, ustawiać datę i czas. Na przykład:

```
...
MOV AH,2          ;funkcja DOS wyświetlająca znaki
MOV DL,'S'        ;S jest znakiem do wyświetlenia
INT 21H           ;wezwij DOS w celu wykonania tej funkcji
...
```

Funkcji DOS używa się w celu wsparcia operacji związanych z wprowadzaniem plików przy użyciu klawiatury, wyprowadzeniem pliku na ekran monitora bądź na drukarkę. Tam gdzie tylko powinno użyć się funkcji DOS, należy zdecydowanie ich używać, aczkolwiek w niektórych przypadkach wyraźnie trzeba użyć funkcji BIOS.



Rysunek 3.1. Oprogramowanie systemowe BIOS i DOS

Funkcje DOS w pełni spełniają potrzeby programisty związane z wejściem/wyjściem i wykonaniem programu. Takim typowym przeznaczeniem funkcji DOS jest „obserwacja" naciskania klawiszy na klawiaturze komputera PC. DOS dostarcza wielu funkcji dających wiele cennych rezultatów dla programisty asemblerowego. Chociażby w przypadku „prostego" naciskania klawiszy najprostszą funkcją DOS z tym związaną jest funkcja numer 1, funkcja czytania znaku z klawiatury z echem. Funkcje DOS są przywoływane, gdy numer tej funkcji umieścimy w rejestrze AH, a następnie wykonamy rozkaz wywołania przerwania 21H, czyli rozkaz INT 21H. Na przykład:

```
...
MOV AH,1                ;funkcja DOS odczytująca znaki z klawiatury
INT 21H                 ;wezwij DOS w celu wykonania funkcji (odczytaj znak)
...
```

Po wykonaniu tej sekwencji rozkazów w rejestrze AL znajdzie się odczytany znak wpisany z klawiatury. Funkcja czeka, aż zostanie naciśnięty klawisz. Jeśli odebrany (funkcją numer 1) znak chcemy uwidocznić na ekranie, musimy użyć funkcji systemowej o numerze 2. W funkcji tej wartość uzyskanego w rejestrze AL kodu znaku należy wpisać do rejestru DL. Na przykład:

```
...
MOV AH,1          ;funkcja DOS odczytująca znaki z klawiatury
INT 21H           ;wezwij DOS w celu wykonania funkcji (odczytaj znak)
MOV AH,2          ;funkcja DOS wyświetlająca znaki
MOV DL,AL         ;przenieś znak (uzyskany z funkcji numer 1) z AL do DL
INT 21H           ;wezwij DOS w celu wykonania funkcji (wyświetl znak)
...
```

DOS ma jeszcze wiele innych ciekawych funkcji operujących na znakach. Można przy jego użyciu odczytać bądź wyświetlić nie tylko pojedynczy znak, ale też cały łańcuch znaków. (Podobnie też działają funkcje w języku C i Pascal: instrukcja **scanf** i **printf**, a w Pascalu **write**). Aby móc prawidłowo zakończyć program, bez jego zawieszenia się, **koniecznie trzeba** użyć funkcji 4CH, i o tej funkcji należy bezwzględnie pamiętać. Poniżej pokazano prosty program z poznanych funkcji DOS, bez dokładniejszego komentarza nie poznanych jeszcze instrukcji programowych.

(**Uwaga!** We wszelkich programach asemblerowych wielkość znaków nie odgrywa roli, jednakże dla zaznaczenia ważności rozkazów, przerwań itp. w książce tej zapisywane są dużymi znakami).

```
.MODEL small
.STACK 100h
.CODE
Echo_Skok:        ;etykieta Echo_Skok (etykieta zakończona dwukropkiem)
MOV AH,1          ;funkcja DOS odczytująca znaki z klawiatury
INT 21H           ;wezwij DOS w celu wykonania tej funkcji
CMP AL,13         ;czy naciśnięto klawisz [Enter]?
                  ;(kod klawisza [Enter]=13)
JZ Echo_Skok      ;jeśli tak (tzn. gdy wartość w AL=13), to idź do Echo_Skok
MOV DL,AL         ;przenieś znak z AL do DL
MOV AH,2          ;funkcja DOS wyświetlająca znaki
INT 21H           ;wezwij DOS w celu wykonania funkcji (wyświetl znak)
JMP Echo_Skok     ;skocz (bezwarunkowo) do etykiety Echo_Skok
Echo_Gotowe:      ;etykieta o nazwie Echo_Gotowe
MOV AH,4CH        ;funkcja wykonania programu
INT 21H           ;wezwij DOS w celu wykonania funkcji (wykonaj program)
END
```

3.1. Funkcje BIOS

Czasami funkcje DOS nie zaspokajają żądań programistów, wówczas należy się zwrócić do BIOS. Odmiennie niż DOS oraz programy użytkowe, BIOS nie jest ładowany z dysku, lecz jest on zapamiętany w pamięci ROM (ang. *Read Only Memory*), z której można tylko go odczytywać. BIOS jest oprogramowaniem komputera PC położonym najniżej; funkcje BIOS uzupełniają DOS w zakresie kontroli nad sprzętem. Ze względu na pewne różnice w oprogramowaniu BIOS powinniśmy raczej używać funkcji DOS niż BIOS, by tym samym uniknąć konfliktów programowych dla różnych modeli komputerów PC. Spośród wielu zastosowań funkcji BIOS jednym z nich jest użycie go w celu sterowania monitorem ekranowym. Tylko przez przywoływanie funkcji BIOS można ustawić tryb pracy monitora, mieć kontrolę nad kolorami, sposobem wyświetlania itp., na przykład:

```
...
MOV AH,0                ;tryb ustawienia BIOS
MOV AL,4                ;nr trybu dla 4-kolorowej grafiki 320x200 (CGA)
INT 10H                 ;wykonaj przerwanie BIOS ustawiające tryb video
...
```

Niekiedy też się zdarza, że musimy mieć programowy kontakt bezpośrednio z portami szeregowymi, wówczas używamy rozkazów IN i OUT.

4. „Towarzysze" głównego procesora

Główny procesor nie może sterować całym komputerem, a nawet nie powinien. Ma tyle własnych zadań, iż nie ma czasu na inne zajęcia. A tych innych zajęć jest co nie miara. Przecież trzeba sterować przepływem informacji w obwodach wewnętrznych lub między komputerem a urządzeniami do niego podłączonymi, takimi jak monitor ekranowy, dysk, podawać wielofazowe równomierne sygnały dla głównego procesora i tzw. urządzeń peryferyjnych. Nie sposób byłoby w tej książce nazwać wszystkich elektronicznych towarzyszy procesora głównego. Wymieńmy zatem i pokrótce opiszmy przynajmniej kilka tych najważniejszych, czy może najczęściej spotykanych w praktyce programistycznej. Zdarza się, iż młodzi adepci asemblerowi, gdy tylko zakosztują „owocu z drzewa Dobrego i Złego", od razu chcą oprogramowywać wszystkie te chipy, których – przynajmniej na początku swoich doświadczeń z Asemblerem – nie powinno się oprogramowywać. Tych chipów jest całkiem sporo, a najbardziej znane są:

a) sterownik przerwań 8259,

b) sterownik DMA, 8237A,

c) licznik czasu 8253/8254,

d) UART 16450,

e) układ MC14818, związany z pamięcią CMOS-RAM i zegarem czasu rzeczywistego,

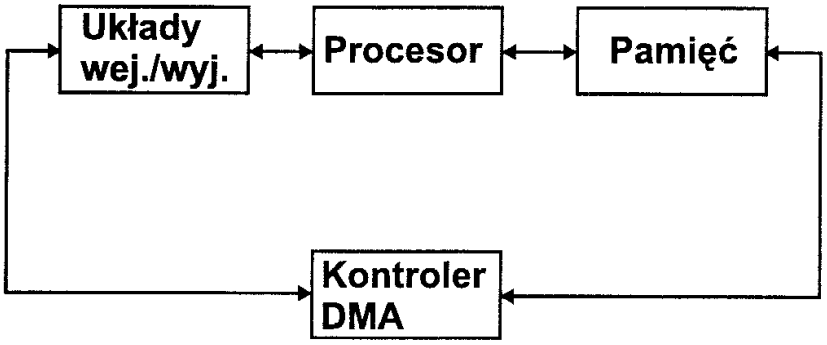
f) układ 8048 (8049) nadzorujący siatkę połączeń klawiszy na klawiaturze.

Nowe typy maszyn mają rozwiązania oparte na układzie 82C206, który zastępuje kilka układów: sterownik(i) przerwań, sterownik(i) DMA, generator czasowy, układ CMOS-RAM. Stosowane układy 82C301 i 82C302 odpowiednio kontrolują relacje czasowe w systemie oraz pamięć.

Ad (a). Sterownik przerwań 8259 nadzoruje działanie przerwań, przejmuje wszystkie sygnały idące od sprzętu, określa ich poziom ważności w stosunku do innych odbieranych sygnałów oraz wysyła przerwanie do CPU na podstawie poziomu ważności. Sterownik przerwań może obsłużyć 8 zadań przerwania równocześnie i może być dołączony do innego układu 8259 w celu zwiększenia liczby obsługiwanych przerwań do 15. Zazwyczaj bezpośrednio nie programujemy sterownika przerwań, gdyż może

to spowodować zakłócenie w podstawowych funkcjach działania komputera, jednakże istnieje programowa możliwość zmiany priorytetów przerwań w taki sposób, by móc dostosować sterownik do własnych potrzeb.

Ad (b). Sterownik DMA odciąża główny procesor od wielu czynności związanych z przesyłaniem danych, np. z dysku i na dysk, w sposób bezpośredni, bez angażowania przy tym procesora głównego. Ten bezpośredni dostęp do pamięci (DMA – ang. *Direct Memory Access*) realizowany jest za pomocą układu 8237A.



Rysunek 4.1. Bezpośrednia komunikacja układów wejścia/wyjścia z pamięcią

Urządzeniem wejścia/wyjścia może być kontroler dysku, kontroler streamera itp. Do każdego z tych urządzeń przyporządkowany jest jeden z kanałów DMA, logicznych strumieni danych inicjowanych przez procesor. Każdy układ 8237A może obsługiwać cztery takie strumienie, a kaskadowe połączenie tych dwóch układów umożliwia obsługę siedmiu kanałów DMA. Urządzenie stowarzyszone z danym kanałem DMA wysyła sygnał żądający od DMA jego obsługi. Układ 8237A posiada swoje wewnętrzne 16-bitowe rejestry, za pomocą których adresuje się obszary pamięci RAM (tzw. strony DMA) o wielkości 64 KB; 2^{16}= =65536/1024=64. Programowanie układu 8237A polega na wpisywaniu, poprzez procesor główny, odpowiednich wartości do jego rejestrów. Układ 8237A posiada 27 rejestrów i są one „widziane" przez procesor główny poprzez porty wejścia/wyjścia o odpowiednich adresach; rozkazy IN i OUT służą do działania na adresach portów wejścia/wyjścia.

Ad (c). Licznik czasu – układ 8253/8254 jest bardzo precyzyjnym, wielozadaniowym licznikiem czasowym. Pełni on funkcje zegara systemowego, odświeża pamięć dynamiczną, obsługuje głośnik. Gdy w komputerze PC znajdują się dwa układy 8254, drugi z nich chroni system przed jego zupełnym załamaniem się. Układ 8253/8254 otrzymuje od generatora zegara (układ 8284A) prostokątny sygnał o częstotliwości 1,19318 MHz. Programowanie licznika czasu sprowadza się do odczytu portów i ich zapisu odpowiednimi wartościami.

Ad (d). UART 16450 – transmisja danych pomiędzy urządzeniami dołączonymi do komputera może mieć charakter transmisji równoległej lub szeregowej; z kolei szeregowa transmisja danych może się dokonywać w sposób synchroniczny i asynchroniczny. Nie wnikając w szczegóły, należy jednak wyjaśnić, że szeregowy strumień

informacji, zanim wpłynie do równoległej magistrali danych, nie może być w żadnym stopniu użyteczny, gdy wcześniej nie podda się go specjalnemu przekształceniu. Przekształcenie danych z postaci szeregowej na równoległą i odwrotnie dokonywane jest poprzez układ scalony typu 8250, 16450 lub 82450 zwany jako UART (ang. Universal Asynchronous Receiver-Transmitter). UART posiada wiele wyspecyfikowanych rejestrów dostępnych programowo w przestrzeni adresów portów wejścia/wyjścia. Poprzez te porty istnieje możliwość dostosowania parametrów pracy układu UART zależnie od naszych potrzeb, zaprogramowania prędkość transmisji, formatu danych itp.

Ad (e). Układ MC14818 spełnia dwa bardzo ważne zadania w komputerze PC, a mianowicie zapamiętuje konfigurację komputera PC oraz zawiera kalendarz i zegar czasu rzeczywistego. Najważniejszą częścią tego układu jest 64-bitowa pamięć CMOS-RAM, z której można dane odczytywać i do niej zapisywać. Dostęp do tego układu scalonego możliwy jest poprzez funkcje BIOS i poprzez porty.

Ad (f). Układ 8048(8049) nadzoruje siatkę połączeń klawiszy na klawiaturze. W węzłach siatki umieszczone są poszczególne klawisze. Głównym zadaniem układu 8048 jest śledzenie klawiszy i przekazywanie informacji do ROM-BIOS, gdy jakikolwiek klawisz zostanie wciśnięty lub zwolniony. Układ 8048(8049), kontroler klawiatury, jest układem, który może być zaprogramowany.

5. Program uruchomieniowy DEBUG

5.1. Polecenia

Programy uruchomieniowe, tzw. debuggery (ang. *debugger*, uruchamiacz), służą do uruchamiania programów zapisanych w postaci plików *.EXE lub *.COM, umożliwiają testowanie zarówno programów w całości, jak też pewnych jego fragmentów, modyfikację programu, danych, wartości początkowych rejestrów, zmiennych itp. Za pomocą programów uruchomieniowych możemy utworzyć plik dyskowy, którego zawartością będzie wybrany obszar pamięci operacyjnej.

Jednym z najbardziej znanych programów uruchomieniowych jest program DEBUG, dostarczany wraz z systemem operacyjnym. Po wpisaniu nazwy tego programu i naciśnięciu klawisza [Enter] pojawi się kreska jako znak gotowości, do wprowadzania poleceń. Gdy wpiszemy znak zapytania, wyświetli się zbiór poleceń dostępnych w programie DEBUG.

```
-?
asembluj                            A [adres]
porównaj                            C zakres adres
zrzuć                               D [zakres]
wprowadź                            E adres [lista]
wypełnij                            F lista zakresu
idź do                              G [=adres] [adresy]
hex                                 H wartość1 wartość2
wprowadź                            I port
załaduj                             L [adres] [dysk] [pierwszy sektor] [liczba]
przenieś                            M zakres adres
nazwa                               N [ścieżka] [lista arg]
wyprowadź                           O port bajt
przejdź                             P [=adres] [liczba]
zakończ                             Q
rejestr                             R [rejestr]
szukaj                              S zakres lista
śledź                               T [=adres] [wartość]
dezasembluj                         U [zakres]
zapisz                              W [adres] [dysk] [pierwszy sektor] [liczba]
alokuj pamięć rozszerzoną typu expanded      XA [#stron]
dezalokuj pamięć rozszerzoną typu expanded   XD [dojście]
mapuj strony pamięci rozszerzonej            XM [Lstrona] [Pstrona] [dojście]
wyświetl stan pamięci rozszerzonej           XS
```

Zacznijmy od polecenia **A** [adres] – asembluj (parametry umieszczone w nawiasach kwadratowych są opcjonalne). Polecenie **A** pozwala na wprowadzenie rozkazów mikroprocesora do pamięci operacyjnej komputera w postaci rozkazów, mnemoników. Parametr [adres] określa miejsca umieszczenia asemblowanego programu, brak tego parametru lokuje program pod adresem, który wynika z aktualnej zawartości rejestrów CS i IP mikroprocesora. Programy, czy nawet pojedyncze rozkazy, które będziemy wpisywać, nie są zrozumiałe dla procesora i muszą być przed załadowaniem do pamięci i wykonaniem przetłumaczone na język wewnętrzny. W tym przypadku tłumaczenie to odbywa się za pomocą programu tłumaczącego DEBUG, który czyta wpisywane kody źródłowe w postaci mnemoników i zamienia je w program wynikowy w języku wewnętrznym.

Należy przypomnieć, iż program tłumaczący – translator języka asemblerowego – również nazywa się asemblerem, tak jak język, jednakże w kontekście zawsze wiadomo, o jaki asembler chodzi. W piśmie nazwę języka Asembler rozpoczynamy dużą literą, a jako translator języka asemblerowego – małą literą.

Uwaga! Wszelkie komentarze umieszczane będą po znaku średnika, polecenia zostaną wytłuszczone. Po znaku średnika „;" umieszcza się też komentarze w źródłowych programach asemblerowych.

```
-A                ;wpisujemy polecenie A (mała lub duża litera), naciskamy [Enter]
                  ;aby wyjść z trybu asemblacji naciskamy klawisz [Enter] lub [Ctrl+C]
10A2:0100         ;wartość 10A2 znajduje się w rejestrze CS, natomiast 0100 w IP, wartość
                  ;rejestru CS jest w danej chwili taka, jaką wskaże procesor, natomiast wartość
                  ;rejestru IP będzie zawsze w tym programie DEBUG jednakowa, 0100H.
```

Zawartość wszystkich rejestrów procesora oraz stan flag będzie można zobaczyć, jeśli wprowadzi się polecenie **R**, bez parametru [rejestr]. Gdy natomiast chcemy obejrzeć zawartość żądanego rejestru, wpisujemy **R** z parametrem rejestr, np. **R**AX.

Wydajmy polecenie **A**; kursor ustawia się za aktualną wartością adresu CS:IP do trybu wpisywania rozkazów. Gdy naciśniemy [Enter], wprowadzamy kolejne polecenie **R** i znowu [Enter]. Przyjrzyjmy się zawartości rejestru AX; na pewno są tam same zera. Zróbmy mały eksperyment. Wracamy do trybu asemblacji, ponownie po kresce piszemy polecenie **A** i [Enter]. Wprowadzamy rozkaz MOV AX,1234 i naciskamy [Enter] (wartości liczbowe są podawane i przyjmowane w postaci szesnastkowej). Rozkaz MOV AX, 1234 został na razie wprowadzony do pamięci, program uruchomieniowy wskazuje nam kolejny adres, od którego zaczynać się będzie następny rozkaz, jeśli tylko zechcemy go wpisać. Gdy w trybie asemblacji rezygnujemy z wprowadzania kolejnych rozkazów, to naciskamy klawisz [Enter] i program DEBUG gotów jest do przyjmowania innych poleceń. Wprowadźmy znów polecenie **R**; widać wyraźnie, iż do pamięci operacyjnej zapisano rozkaz MOV AX,1234. Aby jednak rozkaz ten został wykonany (w tzw. trybie krokowym), wydajemy polecenie **T** (bez

parametrów). Rozkaz MOV AX,1234 został wykonany, tzn. do rejestru AX przesłana została wartość 1234.

Mimo iż program uruchomieniowy DEBUG nie jest programem złożonym, to jednak pozwala na używanie przedrostków segmentów w postaci: CS:, DS:, ES:, SS:. Takie przedrostki wpisuje się w osobnej linii przed rozkazem. Można też używać pewnych skrótów symboli czy operatorów, zamiast symbolu RET FAR symbolu RETF, zamiast operatorów BYTE PTR i WORD PTR skrótów BY, WO. Dopuszczalne jest również użycie pseudorozkazów DB i DW rezerwujących miejsce w pamięci, np. DB 'Napis',0D,0A.

C zakres adres – porównaj

Za pomocą polecenia **C** możemy porównać zawartość dwóch obszarów pamięci. Wielkość porównywanych obszarów pamięci operacyjnej wynika z parametru zakres. Na przykład polecenie **C** CS:0100,01AA,0200 porównuje bajt po bajcie obszaru rozpoczynającego się od adresu CS:0100 do 01AA z obszarem rozciągającym się od 0200 do 02AA względem domyślnego rejestru DS. Wszelkie niezgodności między porównywanymi obszarami zapisywane są w postaci:

```
adres_obszaru1 bajt_obszaru1  bajt_obszaru2  adres_obszaru2
```

D [zakres] – zrzuć

Polecenie **D** wyprowadza na ekran zawartość obszaru pamięci w kodach szesnastkowych lub jako znaki ASCII; program DEBUG zastępuje kropkami znaki, których nie można wydrukować. Jeśli nie podano parametru, pamięć będzie pokazywana od następnego bajtu za obszarem wyświetlonym przez ostatnie polecenie **D** i obejmuje 128 kolejnych bajtów. Podczas pierwszego użycia polecenia zawartość pamięci będzie wyświetlana od adresu DS:0100.

Przykłady:

```
D CS:020A
D DS:0120
```

E adres [lista] – wprowadź

Polecenie **E** modyfikuje zawartość pamięci operacyjnej. Poleceniem tym można wprowadzić zarówno łańcuch znaków, jak też spowodować wyświetlenie kolejnych bajtów, począwszy od parametru adres, z oczekiwaniem na zmianę ich zawartości. Na przykład:

```
-E ES:0000 'Komputer'
```

Po wydaniu polecenia **D** (jak poniżej) otrzymujemy po lewej stronie ekranu szesnastkowy zapis słowa 'Komputer', po prawej stronie ekranu zwykły zapis znakowy (i ewentualnie zobaczymy jeszcze inne znaki).

```
-D ES:0000
10A2:0000 4B 6F 6D 70 75 74 65 72-1D F0 4F 03 27 0A 8A 03  Komputer
```

Polecenie **E** umożliwia też wprowadzenie pojedynczych liczb szesnastkowych; wprowadzanie kolejnych znaków wymaga naciśnięcia klawisza [SPACJA], na przykład:

```
-E ES:0000
10A2:0000 4B._
```

W miejsce wskazywane przez kursor (dolna kreska) wpisujemy liczbę szesnastkową, naciśnięcie klawisza [SPACJA] powoduje przejście do wpisywania następnego znaku, naciśnięcie klawisza [ENTER] – wyjście z tego polecenia.

F lista zakresu – wypełnij

```
-F ES:0000 L 20 41
```

Po wpisaniu polecenia **F** wprowadzać będziemy 32 znaki (szesnastkowo 20) o wartości szesnastkowej 41 (kod dużej litery A), począwszy od adresu ES:0000.

```
-F ES:0000 'Mikrokomputer'
```

Po tak użytym poleceniu **F** słowo 'Mikrokomputer' wypełni jeden pełny blok pamięci o wielkości 128 bajtów.

-G [=adres] [adresy] – idź do, wykonaj

Polecenie **G** powoduje rozpoczęcie wykonywania testowanego programu od zadeklarowanego adresu, a następnie zatrzymanie jego wykonywania po dojściu do adresu zadeklarowanego jako tzw. punkt kontrolny (ang. *breakpoint*). Na przykład wywołanie polecenia **G** w formie: **-G** 0102 0128 spowoduje wykonanie kodu programu od adresu początkowego 102 w segmencie CS z ustawieniem punktu zatrzymania (pułapki) na adresie 128. Jeśli zostanie pominięty adres początkowy, to program będzie wykonywany od bieżącego adresu zawartego w parze rejestrów CS:IP.

H wartość1 wartość2 – hex

Polecenie **H** oblicza sumę i różnicę dwóch liczb szesnastkowych, na przykład:

```
-H B 4
000F 0007
```

Liczba 000F jest sumą liczb B (11 dziesiętnie) i liczby 4, 0007 jest natomiast różnicą tych liczb.

I port – wprowadź

Polecenie **I** odczytuje i wyświetla szesnastkową zawartość podanego portu, na przykład:

```
- I 61
2C
```

Port o numerze 61 (szesnastkowo) zawiera wartość 2C.

```
-I 42
44
```

Port o numerze 42 zawiera wartość 44.

L [adres] [dysk] [pierwszy sektor] [liczba] – załaduj

Polecenie **L** ładuje do pamięci operacyjnej plik z dysku o nazwie podanej poleceniem **N**. Za pomocą polecenia **L** możemy też załadować z dysku do pamięci operacyjnej odpowiednią ilość jego sektorów. Gdy nie podano adresu, ładowanie nastąpi od adresu 0100h w segmencie kodu CS (dla pliku o rozszerzeniu COM). Plik z rozszerzeniem EXE załadowany zostanie pod taki adres, jaki wyznaczają reguły rządzące tego typu plikami.

Przykład:

```
-N A:\nc_sk.com
```

Zadeklarowano plik o nazwie nc_sk.com znajdujący się na dysku A.

```
-L
```

Po wydaniu polecenia **L** zawartość pliku nc_sk.com zostaje załadowana do pamięci operacyjnej pod adres CS:IP.

```
-L 100,0,4,6
```

Powyższe użycie polecenia **L** załaduje z dysku A (0=A, 1=B, 2=C itp.) do pamięci operacyjnej 6 kolejnych sektorów logicznych dysku, poczynając od sektora nr 4 pod adres CS:0100.

M zakres adres – przenieś; kopiowanie zadeklarowanego obszaru pamięci pod wskazany adres (uwzględniane jest ewentualne nakładanie się obszarów na siebie).

Przykład:

Wyświetlmy pewien obszar pamięci, który następnie będziemy chcieli kopiować w inne miejsce pamięci – obszar docelowy.

```
-D CS:0100 0120
-D DS:0300 0320
```

Kopiujemy 33 bajty pamięci spod adresu CS:0100 do obszaru zaczynającego się adresem DS:0300.

```
-M CS:0100 0120 DS:0300
```

Ponownie przejrzyjmy zawartości obszarów, CS:0100 do 0120 oraz DS:0300 do 320, ażeby przekonać się, że przekopiowane zostały 33 bajty pamięci spod adresu CS:0100 (CS:IP).

N [ścieżka] [lista arg] – nazwa; określenie nazwy pliku wraz z ewentualną ścieżką dostępu. Polecenie **N** przygotowuje dane zawarte w pliku dyskowym w celu ich załadowania do pamięci operacyjnej lub też przygotowuje dane zawarte w pamięci operacyjnej do zapisania ich w pliku dyskowym poleceniem **W**.

Przykład:

```
-N A:\nc_sk.com
```

O port bajt – wyprowadź

Polecenie **O** wyprowadza bajt do portu, którego adres jest stałą 8- lub 16-bitową.

Przykład:

-O 61 2D

Do portu o adresie szesnastkowym 61 wprowadzono stałą 2D.

Q – zakończ; koniec pracy z programem DEBUG, przekazanie sterowania do systemu operacyjnego.

R [rejestr] – rejestr; wyświetlenie zawartości jednego lub więcej rejestrów.

Polecenie **R** bez argumentów wyświetla zawartość wszystkich rejestrów i znaczników (flag) procesora.

Przykłady:

```
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=10A2 ES=10A2 SS=10A2 CS=10A2 IP=0100 NV UP EI PL NZ NA PO NC
10A2:0100 FFFF ??? DI
-RAX
AX 0000
:                ;po dwukropku istnieje możliwość zmiany zawartości rejestru
-RCS
CS 10A2
:                ;mimo iż po dwukropku istnieje możliwość zmiany zawartości rejestru, to
                 ;w tym rejestrze nie należy nic zmieniać, nacisnąć tylko klawisz [Enter]
```

-RF – wyświetlenie stanu znaczników (flag) procesora, z możliwością ich zmiany. Program oczekuje na podanie nowego stanu wybranych (bądź wszystkich) znaczników. Gdy nic nie zmieniamy, naciskamy klawisz [Enter]. (Początkujący programiści nie powinni praktycznie nic „w ciemno" zmieniać w żadnym z rejestrów).

```
NV UP EI PL NZ NA PO NC -
```

Tabela 5.1. Stan znaczników (flag) procesora

Znacznik (flaga)	=1		=0	
OF	OV	(ang. *over*)	NV	(ang. *not over*)
DF	DN	(ang. *down*)	UP	(ang. *up*)
IF	EI	(ang. *enables int*)	DI	(ang. *disables int*)
SF	NG	(ang. *negative*)	PL	(ang. *plus*)
ZF	ZR	(ang. *zero*)	NZ	(ang. *not zero*)
AF	AC	(ang. *aux. carry*)	NA	(ang. *not aux. carry*)
PF	PE	(ang. *parity even*)	PO	(ang. *parity odd*)
CF	CY	(ang. *carry yes*)	NC	(ang. *no carry*)

S zakres lista – szukaj

Polecenie **S** przeszukuje według podanego wzorca zadeklarowany obszar pamięci. Poszukiwanie jest skończone, gdy został znaleziony wzorzec lub gdy został osiągnięty koniec zakresu zadeklarowanego do poszukiwania. Dla jednobajtowego wzorca program DEBUG wyświetli wszystkie adresy, przy których wzorzec został znaleziony.

Przykłady:

```
-S CS:0100,0400 'tM'          ;poszukujemy łańcucha znaków 'tM' od adresu CS:0100
                              ;do adresu CS:0400
10A2:036C                     ;łańcuch 'tM' występuje pod adresem 10A2:036C
-S CS:0100,0500 4d 02         ;poszukujemy wystąpienia ciągu bajtów 4d 02 w przedziale
                              ;adresów CS:0100 do CS:0500
10A2:032D                     ;bajty 4d 02 znalezione zostały przy adresach
10A2:034D                     ;bajty 4d 02 znalezione zostały przy adresach
10A2:0365                     ;bajty 4d 02 znalezione zostały przy adresach
10A2:036D                     ;bajty 4d 02 znalezione zostały przy adresach
```

T [=adres] [wartość] – śledź; krokowa realizacja programu

Polecenie **T** bez żadnych parametrów wykonuje pojedynczy rozkaz spod bieżącego adresu, określonego stanem rejestrów CS:IP i wyświetla zawartość wszystkich

rejestrów. Gdy podajemy adres początkowy, poprzedzamy go znakiem '='. Domyślną liczbą kroków wykonywania rozkazów wskazywaną przez parametr [wartość] jest liczba 1.

Przykłady:

```
-T              ;wykonaj rozkaz spod adresu wskazanego przez aktualny stan rejestrów CS:IP
-T A            ;wykonaj dziesięć rozkazów, począwszy od adresu CS:IP
-T=CS:013E 5    ;wykonaj pięć kolejnych rozkazów, zaczynając od adresu CS:013E
```

U [zakres] – dezasembluj; tłumaczenie zawartości pamięci operacyjnej na symboliczne kody rozkazów i wyświetlenie ich na ekranie.

Uwaga!!! Początkowy adres musi wskazywać pierwszy bajt rozkazu, w przeciwnym razie wykonane zostanie błędne tłumaczenie. Gdy parametrem polecenia **U** jest offset, standardowo przyjmowany jest rejestr CS.

Polecenie **U** bez parametru wyświetla 32 bajty kodu maszynowego, począwszy od adresu wskazanego przez bieżący stan rejestrów CS:IP (gdy uprzednio wykonano polecenie **R**).

Przykłady:

```
-U 10C7:0100 L6     ;deasemblacja sześciu bajtów od adresu 10C7:0100
10C7:0100 EB74      JMP         0176
10C7:0102 90        NOP
10C7:0103 53        PUSH        BX
10C7:0104 2E        CS:
10C7:0105 4B        DEC         BX
-U                  ;fragment 32-bajtowego ciągu rozkazów, prawidłowo zdeasemblowanych
10C7:0100 EB74      JMP         0176
10C7:0102 90        NOP
10C7:0103 53        PUSH        BX
10C7:0104 2E        CS:
10C7:0105 4B        DEC         BX
10C7:0106 2E        CS:
10C7:0107 4C        DEC         SP
-U 0101 L6          ;fragment nieprawidłowo zdeasemblowanych sześciu bajtów
10C7:0101 7490      JZ          0093
10C7:0103 53        PUSH        BX
10C7:0104 2E        CS:
10C7:0105 4B        DEC         BX
10C7:0106 2E        CS:
10C7:0107 4C        DEC         SP
```

W [adres] [dysk] [pierwszy sektor] [liczba] – zakres

Polecenie **W** zapisuje do pamięci dyskowej określonego obszaru pamięci w formacie mapy pamięci (COM lub BIN). Przed użyciem polecenia **W** należy określić nazwę tworzonego pliku (i ew. ścieżkę) poleceniem **N**. Gdy w poleceniu **W** pominięto adres, zostanie przyjęty adres początkowy CS:0100. Zanim wywołane zostanie pole-

cenie **W**, do zapisu dyskowego należy wpisać liczbę bajtów, odpowiednio w rejestrze CX („młodsza" część) i BX („starsza" część). Program DEBUG nie zapisuje plików w formacie EXE.

Przykłady:

```
-RCX
CX 00DD                    ;aktualny stan rejestru (licznika) CX
:0D                        ;ustawiamy wartość CX na 0D
-N A:\stach.com            ;określamy nazwę i ścieżkę dla zapisywanego pliku
-W CS:0100                 ;zapis na dysku w stacji A, 0D bajtów, począwszy od adresu
                           ;CS:0100
Zapisywanie 000D bajtów    ;na dysku utworzy się plik stach.com
```

Jeżeli będziemy odwoływać się do logicznej organizacji dysku, wówczas polecenie **W** wystąpi z parametrami określającymi odpowiednio: adres w pamięci operacyjnej, skąd kopiujemy dane, numer stacji dysków (0=A,1=B), początkowy sektor dysku, liczbę sektorów; dla dyskietek sektor logiczny zawiera 512 bajtów.

```
-W 0200,0,4,5              ;zapisz na dysk A do jego pięciu kolejnych sektorów
                           ;logicznych, począwszy od sektora nr 4, dane zawarte
                           ;w pamięci operacyjnej zaczynające się od adresu CS:0200
```

Uwaga!!! Z poleceniem W eksperymentować tylko i wyłacznie na dyskietce.

Polecenia XA, XD, XM, XS można stosować, gdy zainstalowano program zarządzania pamięcią stronicowaną (zwykle jest to EMM386.EXE). Dla programisty, zwłaszcza początkującego, polecenia te nie mają większego znaczenia.

XA [#stron] – alokuj pamięć rozszerzoną typu *expanded*,

XD [dojście] – dezalokuj pamięć rozszerzoną typu *expanded*,

XM [Lstrona] [Pstrona] [dojście] – mapuj strony pamięci rozszerzonej,

XS – wyświetl stan pamięci rozszerzonej.

Program DEBUG posiada bardzo niewielki zasób komunikatów informujących o popełnionych błędach:

BF – niewłaściwy symbol stanu flagi,

BP – zadeklarowano więcej niż 10 pułapek,

BR – w poleceniu R użyto niepoprawnej nazwy rejestru,

DF – w poleceniu RF podano dwie wartości dla tej samej flagi.

5.2. Proste programy pod DEBUG-iem

Jednym z najprostszych programów, jakie na samym początku używania programu DEBUG można wykonać, jest wywołanie przerwania obsługującego drukarkę. W tablicy wektorów przerwań znajdujemy przerwanie o numerze 5, przerwanie to powoduje wydruk tego, co widać na ekranie. Wywołajmy polecenie **D**, może i **R** (chodzi o to, aby na ekranie był sporo znaków), a następnie polecenie asemblacji **A**.

```
-A
10A2:0100 INT 5H          ;po adresie CS:IP (tu: 10A2:0100) wpisujemy rozkaz INT 5H
10A2:0102                 ;i jeszcze raz naciskamy klawisz [Enter]. Rozkaz INT 5H
-G=0100 0102              ;zajmuje dwa bajty od 0100 do 0102, wykonajmy ten rozkaz
```

Wykonanie rozkazu INT 5H spowoduje wydruk na drukarce zawartości ekranu.

Po wykonaniu poprzedniego programu w postaci dwubajtowego rozkazu INT 5H program DEBUG jest już gotów do wykonania następnego polecenia. Spróbujmy jeszcze wykonać taki sam krótki program jak poprzednio, wpisując teraz numer przerwania 19H; w rozdziale „Tablica wektorów przerwań", czytamy, iż przerwanie 19H powoduje załadowanie systemu operacyjnego; innymi słowy, efektem wykonania dwubajtowego rozkazu INT 19H będzie restart komputera. W tym miejscu może rodzić się pytanie: co kryje się za tymi rozkazami INT 5H, INT 19H, INT 4H itp. Otóż pod tymi rozkazami INT kryją się różne procedury, często „porozrzucane" po całej pamięci i dość skomplikowane. Można spróbować prześledzić, krok po kroku, jak wygląda taka procedura, aż do jej efektu końcowego. Z takich obserwacji możemy wyciągnąć wiele interesujących wniosków, możemy też wzorować się na zapisanych w pamięci RAM i ROM procedurach i na ich podstawie napisać własne, może nawet lepsze. Tak też można robić, zastępując oryginalne procedury własnymi procedurami, jeśli chce się uzyskać coś więcej lub też zaprezentować procedurę inaczej niż „zaszyto" to w systemie operacyjnym bądź w sprzęcie.

Poniżej dokonujemy asemblacji (oczywiście poleceniem **A**) rozkazu INT 4H. Spójrzmy na kilka kroków wykonywania się tego rozkazu.

```
-A
10A2:0100 INT 4H
10A2:0102
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=10A2 ES=10A2 SS=10A2 CS=10A2 IP=0100 NV UP EI PL NZ NA PO NC
10A2:0100 CD04 INT 04
```

Powyższe wartości CS i IP są aktualnymi wartościami w systemie i będę one różne w każdym innym komputerze, a nawet będą różne w tym samym komputerze po uprzednim uruchomieniu programu.

```
-T
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFE8 BP=0000 SI=0000 DI=0000
```

```
DS=10A2 ES=10A2 SS=10A2 CS=0070 IP=0465 NV UP DI PL NZ NA PO NC
0070:0465 CF IRET
```

Jeśli chodzi o powyższe wartości CS i IP, to koniecznie należy zajrzeć do rozdziału „Tablica wektorów przerwań".

```
-T
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=10A2 ES=10A2 SS=10A2 CS=10A2 IP=0102 NV UP EI PL NZ NA PO NC
10A2:0102 0000 ADD [BX+SI],AL DS:0000=CD
-T
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=10A2 ES=10A2 SS=10A2 CS=10A2 IP=0104 NV UP EI NG NZ NA PO NC
10A2:0104 27 DAA
-T
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=10A2 ES=10A2 SS=10A2 CS=10A2 IP=0105 NV UP EI PL ZR NA PE NC
10A2:0105 0300 ADD AX,[BX+SI] DS:0000=20CD
.
.
.
```

Spójrzmy jeszcze na fragment kodu maszynowego „zaszytego" w pamięci (RAM i ROM), odpowiedzialnego za wydruk na drukarkę. Co więc musimy zrobić? Musimy dokonać asemblacji rozkazu INT 5H, a następnie krok po kroku oglądać kod procedury powodującej wydruk.

```
-A
10A2:0100 INT 5H
10A2:0102
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=10A2 ES=10A2 SS=10A2 CS=10A2 IP=0100 NV UP EI PL NZ NA PO NC
10A2:0100 CD05 INT 05
```

Powyższe wartości CS i IP są aktualnymi wartościami w systemie i, podobnie jak poprzednio, będą różne w każdym innym komputerze, a nawet będą różne w tym samym komputerze, po uprzednim uruchomieniu programu.

```
-T
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFE8 BP=0000 SI=0000 DI=0000
DS=10A2 ES=10A2 SS=10A2 CS=F000 IP=FF54 NV UP DI PL NZ NA PO NC
F000:FF54 60 DB 60
```

Jeśli chodzi o powyższe wartości CS i IP, to koniecznie należy zajrzeć do rozdziału „Tablica wektorów przerwań".

```
-T
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFD8 BP=0000 SI=0000 DI=0000
DS=10A2 ES=10A2 SS=10A2 CS=F000 IP=FF55 NV UP DI PL NZ NA PO NC
F000:FF55 1E PUSH DS
-T
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFD6 BP=0000 SI=0000 DI=0000
DS=10A2 ES=10A2 SS=10A2 CS=F000 IP=FF56 NV UP DI PL NZ NA PO NC
```

```
F000:FF56 B84000 MOV AX,0040
-T
AX=0040 BX=0000 CX=0000 DX=0000 SP=FFD6 BP=0000 SI=0000 DI=0000
DS=10A2 ES=10A2 SS=10A2 CS=F000 IP=FF59 NV UP DI PL NZ NA PO NC
F000:FF59 50 PUSH AX
-T
AX=0040 BX=0000 CX=0000 DX=0000 SP=FFD4 BP=0000 SI=0000 DI=0000
DS=10A2 ES=10A2 SS=10A2 CS=F000 IP=FF5A NV UP DI PL NZ NA PO NC
F000:FF5A 1F POP DS
-T
AX=0040 BX=0000 CX=0000 DX=0000 SP=FFD6 BP=0000 SI=0000 DI=0000
DS=0040 ES=10A2 SS=10A2 CS=F000 IP=FF5B NV UP DI PL NZ NA PO NC
F000:FF5B B001 MOV AL,01
-T
AX=0001 BX=0000 CX=0000 DX=0000 SP=FFD6 BP=0000 SI=0000 DI=0000
DS=0040 ES=10A2 SS=10A2 CS=F000 IP=FF5D NV UP DI PL NZ NA PO NC
F000:FF5D 86060001 XCHG AL,[0100] DS:0100=00
.
.
.
```

Śledząc już chociażby tych kilka kroków w wykonywaniu procedury wydruku zawartości ekranu, zorientujemy się, gdzie „uciekają" adresy kodów maszynowych śledzonej procedury. Rejestr kodu CS wskazuje adres F000H, offset IP na FF5DH. Takie adresy leżą „głęboko" w pamięci, i to w pamięci ROM.

Spróbujmy przeliczyć:

F000×10H+FF5DH=F0000H+FF5DH=FFF5DH=1048413 dziesiętnie.

W 1 MB pamięci mieści się 1048576 bajtów, czyli procedura drukowania jest gdzieś blisko krańca 1 MB obszaru pamięci, obszaru pamięci ROM. W tym też obszarze „zaszyto" całe mnóstwo innych procedur umożliwiających w ogóle działanie komputera i obsługę jego peryferii. Czego jeszcze można się nauczyć, patrząc na kolejne kroki wykonującej się procedury drukowania? Wielu interesujących rzeczy. Między innymi można zobaczyć w działaniu, w akcji, rozkazy MOV, PUSH, POP itp., przyrównując do siebie kolejne stany niektórych rejestrów.

Program DEBUG.EXE, mimo wielu niedogodności, pozwala jednak początkującym programistom uzyskać wiele satysfakcji, zwłaszcza gdy efekty ich pracy będą namacalne, dają się słyszeć, zobaczyć, bądź też gdy – ku naszemu zdziwieniu – nastąpi zresetowanie komputera. To ostatnie zjawisko nie zawsze bywa korzystne; nauka jednak kosztuje dużo wyrzeczeń i nakładów finansowych, koniecznie też trzeba wyrzec się konsumpcyjnego podejścia do rzeczywistości, należy być roztropnym i spokojnym badaczem, mając w sobie dużo stoickiego spokoju i mnisiej wytrwałości. Wróćmy jednak do Asemblera. Spójrzmy do rozdziału „Tablicy wektorów przerwań", przerwanie nr 10H – Obsługa ekranu monitora (wektor: 0040-0043). Przerwanie to oferuje szeroki zakres usług. Numer usługi wpisujemy do rejestru AH; rejestr AH odgrywa w programach asemblerowych kluczową rolę. Weźmy pod uwagę dwie „proste" usługi: dla AH=1 i AH=6. W tym miejscu proszę jednak nie mieć zbyt du-

żych złudzeń, iż wystarczy przesłać (tu: pod DEBUG.EXE) do rejestru AH wymienione wartości, wywołać przerwanie 10H za pomocą rozkazu INT i po kłopocie. Owszem, niekiedy tak bywa, że przerwanie nie potrzebuje więcej uszczegółowiania, aby wywołać zeń jakąś usługę, jednakże w tym przypadku tak nie jest. Przede wszystkim, nie znamy jeszcze języka Asembler, byśmy mogli przesłać daną do wyznaczonego rejestru. To prawda, znamy polecenia DEBUG.EXE, które na tym etapie tworzenia ulotnych programów wystarczą nam do takiej prostej operacji, jaką jest przesłanie danej. Mówimy: – *Prostą operację, jaką jest przesłanie danej*, a tak naprawdę wiele złożonych czynności logiczno-elektronicznych musi zajść, zanim po naszym wpisaniu poleceń ustawią się rejestry procesora w taki stan, o jaki nam chodzi. Mniejsza na razie z tym, niech się o to „martwi" program uruchomieniowy, no i może procesor wraz ze swym otoczeniem.

Wróćmy do głównego wątku. Gdy już uda się nam wpisać (jak?) do rejestru AH wartość równą 1, wówczas zaczniemy kształtować na ekranie monitora rozmiar kursora według naszych upodobań. To dopiero początek. Kursor jest takim sobie małym prostokącikiem, i aby w pełni go zdefiniować na ekranie trzeba podać w rejestrze CH numer linii w wierszu, od której zaczyna się jego wyświetlanie, a w rejestrze CL numer linii w wierszu, na której kończy się wyświetlanie. W normalnym trybie kursor, według standardu IBM, ma postać jednej, a zwykle dwóch poziomych kresek „rysowanych" w obszarze prostokąta o wysokości 16 takich kresek. Długości i grubości tych kresek zmieniać nie możemy, możemy manipulować tylko ich liczbą, mniej lub bardziej wypełniając ów prostokąt. Zapewne wszyscy programiści domyślili się, iż „akcja toczy się" w trybie tekstowym. W trybie graficznym komputera (a dokładniej jego karty graficznej) możemy programować kursor wszerz i wzdłuż, jaki tylko zechcemy, np. zamiast standardowego kursora-prostokąta zbudowanego z kresek zaprogramujemy go w postaci zegarka, dłoni, rakiety, drzewa itp. W przypadku trybu tekstowego, gdy do rejestru CH wstawimy wartość 00, a do CL wartość 01, to powinniśmy otrzymać kursor o dwóch kreskach położonych możliwie najwyżej, gdy zaś do rejestru CH wstawimy wartość 0FH i do CL też wartość 0FH, wówczas będziemy mieli kursor o jednej kresce możliwie najniżej położonej.

Wiemy już w zasadzie wszystko o tym, jak programowo utworzyć taki kursor. Uruchamiamy program DEBUG.EXE. Po „słynnej" debuggowskiej kresce stanowiącej znak gotowości do wprowadzania poleceń, wydajmy polecenie **R**, bez parametru; chcemy tylko zobaczyć stan rejestrów. Na pewno w rejestrach CX, AX, jak i w wielu innych, mamy stan 0000. Musimy zmienić stany tych rejestrów; do AH wstawimy wartość 01, do CH i CL tę samą wartość równą 0FH. Ponownie piszemy polecenie **R**, ale już z parametrem, potem CX, a następnie AX. Po **R**AX [Enter] i dwukropku wpisujemy 0100 [Enter]; AX=AH+AL. Następnie po **R**CX [Enter] i dwukropku wpisujemy 0F0F [Enter]; CX=CH+CL. Ta usługa zmiany kursora pochodzi z procedury przerwania dotyczącego obsługi ekranu monitora, przerwania 10H. Ten numer prze-

rwania wpisujemy do pamięci, wprowadzając najpierw polecenie asemblacyjne **A**, a następnie wpisując INT 10H [Enter] i jeszcze raz [Enter], gdyż w tym programie nie będziemy już wpisywać żadnych rozkazów. Możemy wpisać raz jeszcze polecenie **R** bez parametru, aby przekonać się o stanie rejestrów i wprowadzonym rozkazie INT 10H. Wykonajmy ten rozkaz INT 10H z wprowadzonymi wartościami do AX i do CX. Rozkaz mieści się na dwóch bajtach od offsetu 0100H do 0102H. Po znaku zachęty wpisujemy polecenie **G**=0100 0102. Dwubajtowy program został wykonany, zostawiając na ekranie nowy, cieniutki i inaczej położony niż poprzednio kursor.

Możemy skonstruować jeszcze bardziej efektowny program, używając funkcji 6H lub 7H. Użyjmy funkcji 6H przerwania obsługi ekranu monitora. Program obsługi kryjący się za tą funkcją służy do definiowania prostokątnego pola – okna tekstowego na ekranie i do przesuwania jego zawartości w górę (funkcja 7H powoduje przesuwanie w dół). Funkcję 6H (czy 7H) kierujemy do rejestru AH. Z tą funkcją związanych jest wiele parametrów. Do rejestru AL wpisujemy liczbę wierszy, które mają być wygaszone na dole zdefiniowanego okna, do CH numer wiersza górnego lewego rogu okna, do CL numer kolumny górnego lewego rogu okna, do DH numer kolumny dolnego prawego rogu okna, do DL numer kolumny dolnego prawego rogu okna, do BH atrybut wyświetlania pustych linii. Wywołajmy program DEBUG.EXE, a potem polecenie **R**, do AH wstawimy wartość funkcji 06H, do AL wartość 03 (3 linie okna), do CX wartość 050A (CH=05H, CL=0AH; górny lewy róg wiersz nr 5, kolumna nr 10), do DX wartość 1020H (DH=10H, DL=20H; dolny prawy róg wiersz nr 16, kolumna nr 32). Aby efekt wykonania programu był bardziej dostrzegalny, zapiszmy ekran jakimiś znakami, np. wydając polecenie **D** 0000 L 180. Wreszcie na koniec wpiszmy za pomocą polecenia **A** numer przerwania 10H i wykonajmy program **G**=0100 0102. Jeśli Czytelnik zgubił się po drodze, proszę przeprowadzić następujący ciąg poleceń. Po znaku gotowości programu DEBUG.EXE napiszmy **R**AX [Enter], po dwukropku wpisujemy 0603 [Enter], znowu polecenie **R**, lecz teraz z nazwą rejestru CX. **R**CX [Enter], dwukropek 050A [Enter], i jeszcze raz **R** z parametrem oznaczającym rejestr DX. **R**DX [Enter], dwukropek 1020 [Enter]. Po poleceniu asemblacji **A** wpisujemy INT 10H [Enter][Enter]. Zapełniamy ekran dowolnymi znakami, pisząc: **D** 0000 L 180. Wykonujemy program **G**=0100 0102. Efekt znakomity, „wycięty" został fragment ekranu zapisanego przypadkowymi znakami. Na ekranie mamy czarny, pusty prostokącik o współrzędnych, jakie podaliśmy we właściwych dla funkcji 06H rejestrach.

5.3. Zalety, wady, możliwości programu DEBUG

Program DEBUG jest prostym programem uruchomieniowym, umożliwiającym uruchamianie małych programów, oglądanie stanu rejestrów, zawartości wybranego bloku pamięci, proste przesyłanie czy szukanie bajtów w pamięci, no i wreszcie pisanie prostych programów, ich uruchamianie, z możliwością zapisywania kodu wykonywalnego do pliku dyskowego. Faktycznie, istnieje w programie DEBUG taka moż-

liwość, która pozwala zapisywać kod programu, zanim go uruchomimy. Jednakże tu jest pewne, bardzo ważne zastrzeżenie..., ale po kolei.

W programie DEBUG istnieje polecenie **N**, dzięki któremu można określić nazwę pliku wraz z pełną ścieżką dostępu (tego polecenia najlepiej używać tuż po wejściu do programu). Załóżmy, że dokonujemy asmeblacji wektora przerwania nr 5H, powodującego wydruk ekranu na drukarce, i chcemy zapisać wynik tej asemblacji na dysku w postaci pliku dyskowego formatu COM (zapisywać możemy tylko w formacie COM, natomiast wczytywać również i w formacie EXE). Wydajemy polecenie **A**, wpisujemy INT 5H [Enter], [Enter]. Po adresach widać, iż rozkaz INT 5H zajmuje dwa bajty. Załadujmy te dwa zasemblowane bajty rozkazu INT 5H do uprzednio nazwanego pliku dyskowego, wpisując wielkość tworzonego pliku dyskowego do rejestru CX. Wydajemy więc polecenie **R**CX, po dwukropku piszemy 0002 i [Enter], i polecenie **W**, zapisywania na dysku. Podsumujmy wszystkie wydane polecenia:

```
-N A:\drukuj.com
-A
10A2:0100 INT 5H
10A2:0102
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=10A2 ES=10A2 SS=10A2 CS=10A2 IP=0100 NV UP EI PL NZ NA PO NC
10A2:0100 CD05 INT 05
-RCX
CX 0000
10A2:0100 INT 5H
10A2:0102
AX=0000 BX=0000 CX=0002 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=10A2 ES=10A2 SS=10A2 CS=10A2 IP=0100 NV UP EI PL NZ NA PO NC
10A2:0100 CD05 INT 05
-W
Zapisywanie 00002 bajtów
```

Na dysku A w katalogu głównym utworzy się plik, któremu nadaliśmy nazwę *drukuj.com* o wielkości dwóch bajtów, bo dwa bajty ma rozkaz INT 10H i tyle też musieliśmy wpisać do rejestru CX; pamiętamy, że tylko przez rejestr CX możemy wpisać rozmiar pliku do zapisu, gdy plik jest mały, bądź dodatkowo przez rejestr BX dla dużych plików. Tu, i jedynie w nielicznych przypadkach, zdarza się, że rejestr CX nie jest wykorzystywany podczas programowania. Praktycznie zawsze, w każdym programie asemblerowym rejestr CX jest używany (zwykle jako licznik). W programach, w których kod źródłowy zapisujemy jawnie, stosując reguły, pełną składnię i możliwości języka Asembler, możemy się jeszcze od biedy obyć bez rejestru CX, mamy bowiem stos, możemy zadeklarować jedno- czy dwubajtową komórkę pamięci, nazywając ją np. 'licznik', 'kolumna', czy jeszcze inaczej, i do tej komórki przesłać daną, do której potem w programie możemy się zawsze odwołać; rejestr CX będziemy mieli wolny i możemy go wykorzystać do jeszcze bardziej „zbożnych" celów niż przechowalnia na 'licznik' czy na wartość oznaczającą liczbę kolumn tekstowych na

ekranie. Niestety, a może i na szczęście, nie mamy takich możliwości manewrowania komórkami pamięci, rejestrami, stosami itp. w programie DEBUG.EXE. Tu musimy się zadowolić tym, co mamy. Zresztą, to narzędzie służy do tych tylko celów, a inne, bardziej rozbudowane, do bardziej złożonych zastosowań.

6. Podstawy konstruowania programów w języku Asembler

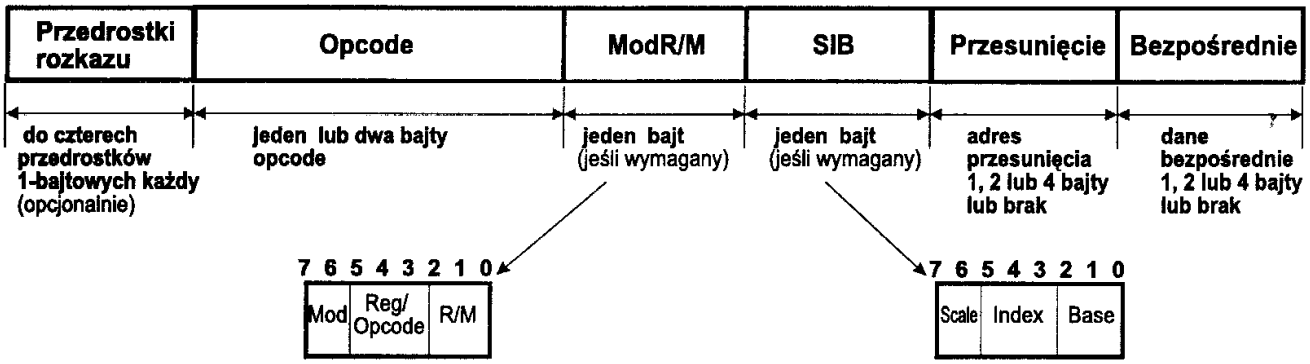
W poprzednich rozdziałach aż nadto stało się oczywiste, że my, programiści, potrzebujemy czegoś więcej, niż tylko obserwowania niejasnych skutków programów, programów wydobywających się gdzieś z „głębin" pamięci i nieuchwytnych na dysk. W kodzie plików COM, a jeszcze gorzej w plikach EXE, nie da się nic zrobić, by ręcznie coś zmieniać, a przede wszystkim to przecież jesteśmy poważnymi ludźmi, byśmy bawili się w asemblerową zgaduj-zgadulę, edytując plik wykonywalny i ewentualnie na chybił trafił usuwając coś z pliku czy też do niego dopisując, jakieś „znaczki", np. θ = -, tworząc w ten sposób nowy program. Te „znaczki" w pliku to „echo" kodów szesnastkowych, kodów, które odzwierciedlają rozkazy procesora. Chcąc użyć w programie COM przerwania 10H, nie będziemy przecież wpisywać jego odpowiednich kodów ASCII =>, tylko musimy stworzyć surowiec do „produkcji" takich różnych „znaczków". Z takich „znaczków", kodów ASCII zbudowane są programy typu COM lub EXE. Tworzenie owego surowca to nic innego, jak umiejętność pisania programów, zapisywanie ich asemblerowego kodu do pliku tekstowego czytelnymi dla programisty znakami i z użyciem pełnej semiotyki języka Asembler. W tym celu, aby napisać program w języku Asembler, trzeba przejść trochę specyficznej, niemalże spartańskiej i bardzo konsekwentnej szkoły myślenia. Nie wystarczy znajomość notacji, semantyki i syntaktyki języka. Trzeba zapoznać się z rozkazami i z ich działaniem.

O ile w językach wysokiego poziomu możemy jeszcze sobie pozwolić na pewną „niedbałość" w programowaniu, o tyle w języku Asembler wymagana jest żelazna konsekwencja w działaniu. W językach wysokiego poziomu możemy sobie pozwolić na trochę luzu, gdyż wiele delikatnych zagadnień załatwia za nas sam ów język, jego specyficzna budowa. Jest to i dobrze, i źle. Dobrze, bo mamy mniej pracy, a źle, bo nie pozwala to nam być samodzielnym i całkowicie panować nad komputerem.

Jednak, jak to w życiu bywa, jest zawsze coś za coś. W języku Asembler wystarczy niekiedy mała nieuwaga, a cały program wysadzi komputer w powietrze, może nie dosłownie, choć – jak już wiemy – tak się zdarzało w przeszłości z rakietami źle zaprogramowanymi („surowym" kodem zerojedynkowym). Niestety, to prawda, ale pisząc w Asemblerze programy tak trochę na oślep, przepisując skądś kody o bliżej nieznanym znaczeniu, można poważnie uszkodzić sprzęt, jeśli nawet nie zniszczyć. Tu trzeba uważać. Obecnie, kiedy w BIOS są funkcje powodujące generowanie

punktu graficznego, tzw. piksela, na ekranie, nie ma specjalnej groźby spalenia monitora, ale gdy karta graficzna była obca komputerowi i nie miał on w swoim BIOS funkcji bezpośrednio oprogramowującej ekran graficzny, wówczas nietrudno było zrobić z monitora bardzo silną latarkę. Każde graficzne oprogramowanie monitora sprowadza się do działania na rejestrach karty graficznej, na przekładaniu zerami i jedynkami. A taki rodzaj programowania jest dosyć delikatny i zawsze niesie ze sobą ryzyko.

Na razie powróćmy do bezpiecznej i nieco naukowej tematyki z dziedziny języka Asembler. W czasie wykonywania programu asemblerowego procesor nieustannie wykonuje wiele złożonych czynności składających się na organizacyjnie zamknięty cykl. Niezależnie od najnowszych rozwiązań konstrukcyjnych procesorów, cykl ten sprowadza się do dwóch najistotniejszych faz – fazy pobrania kodu operacji i fazy wykonania operacji. Dotychczas w tej książce jawnie nie występowało pojęcie kodu operacji, aczkolwiek intuicyjnie na pewno już wiemy o co chodzi. Dla dobra sprawy i precyzji myśli zatrzymajmy się chwilę nad tym pojęciem. Najogólniej, w kodzie rozkazu daje się wyróżnić dwa pola – pole kodu operacji i pole adresu argumentu (operandu) lub argumentu (operandu).



Opcode - kod operacyjny definiujący operację wykonywaną przez rozkaz

ModR/M - tryb adresowania rejestru/pamięci

SIB - skalowany indeks bazy

Przesunięcie - adres przesunięcia (przemieszczenia)

Bezpośrednie - dane bezpośrednie

Mod - dwa bity określające czy oba argumenty są rejestrami, czy też jeden z nich jest w pamięci operacyjnej

Reg/Opcode - trzy bity określające rejestr jako argument rozkazu lub będące rozszerzeniem kodu operacji

R/M - trzy bity wskazujące rejestr będący argumentem lub rejestr wykorzystywany do obliczenia adresu względem początku segmentu

Scale - dwa bity określające współczynnik skali

Index - trzy bity określające numer rejestru z rejestru indeksowego

Base - dwa bity określające numer rejestru z rejestru bazowego

Rysunek 6.1. Format rozkazu

Pole kodu operacji, zajmujące jeden lub dwa bajty, zawiera kod operacji definiujący operację wykonywaną przez rozkaz. Natomiast pole adresu argumentu (operandu) lub argumentu określa miejsce tegoż argumentu. Weźmy pod lupę szesnastkowy,

lub lepiej zerojedynkowy, kod operacji związany z jednym z najczęściej występujących w programach asemblerowych rozkazów – z rozkazem przesłania, a precyzyjniej, z rozkazem kopiowania. Jeśli skopiujemy aktualną zawartość rejestru akumulatora, jego części „dolnej" AL do rejestru licznika też do części „dolnej" CL, wówczas kod tego działania ma postać: 8AC8, natomiast dla kopiowania z rejestru BL do AL kod ma postać: 8AC3. W zapisie szesnastkowym widać już podobieństwo między tymi działaniami, natomiast całą prawdę o tych operacjach pokaże nam ich kod dwójkowy, kod binarny. Dla 8AC8=10001010 11001000 (w grupach 8-bitowych dla 8A i C8) i odpowiednio dla 8AC3=10001010 11000011. W celu lepszego zobrazowania kodów rozkazów umieśćmy je jeden pod drugim:

```
10001010 11001000 = 8AC8=kopiuj AL do CL (MOV CL,AL)
10001010 11000011 = 8AC3=kopiuj BL do AL (MOV AL,BL)
```

W przesłaniu zawartości rejestru do rejestru pierwszy bajt jest interpretowany: 1000101w (w=0 lub 1, zależy, czy działanie oparte jest na bajtach, czy na słowach; tu w obydwu przypadkach: w=0), drugi bajt: 11rejrej (rej=rejestr, trzy bity określające rodzaj rejestru; przy w=0, rejestr AL kodowany jest jako 000, rejestr BL jako 011, rejestr CL – 001). Umieszczone w pamięci kody operacji i liczby (argumenty) są nierozróżnialne, a więc skąd procesor wie, która z tych liczb jest rozkazem, a która daną? Sposób potraktowania przez procesor danego ciągu bitów uzależniony jest od fazy pracy procesora. Jeśli ciąg odczytany jest w fazie pobrania, wówczas umieszczony zostaje on w rejestrze rozkazów (wskaźnik rozkazów) i będzie potraktowany jako kod rozkazu; jeśli ciąg odczytany zostanie w fazie wykonania, umieszczony zostanie w innych rejestrach procesora i będzie potraktowany jako liczba (argument lub adres).

Podstawy konstruowania programów ściśle związane są ze sposobem rozmieszczenia argumentów (operandów) w pamięci. Dla większości rozkazów ich argumenty (operandy) umieszczone są w pamięci. Adres komórki pamięci, gdzie umieszczony jest kod wykonywanego rozkazu, jest zawarty w liczniku rozkazów (wskaźniku rozkazów) IP, natomiast adres argumentu może być określony w bardzo różny sposób, zależny od zastosowanego w rozkazie trybu adresowania. Tryb adresowania określa miejsce, gdzie jest umieszczony adres argumentu (operandu), lub sposób, w jaki jest on obliczany. I w tym kontekście taka uwaga: mimo iż programy napisane w języku Asembler wykonywane są najszybciej ze wszystkich programów napisanych w dowolnym innym języku, to jednak podczas pisania programu w Asemblerze też należy zwracać uwagę na sposób prawidłowego adresowania, aby w programie nie było zbędnych elementów wydłużających jego wykonywanie bądź też fragmentów niepotrzebnie obciążających pamięć itp.

Samo konstruowanie programu źródłowego należy zacząć od postawienia sobie pytania: jakie zadanie ma spełnić program, po czym powinno się naszkicować sieć działania programu, zwłaszcza jeśli będą w nim warunkowe i liczne rozgałęzienia. W tym miejscu oczywiście zakładam, iż zasiadający przed komputerem początkujący

programista zna podstawowe elementy języka oraz środowiska, w którym język funkcjonuje. Edytor, który służy nam do wpisywania treści programu źródłowego, powinien być edytorem czystym, tzn. nie powinien dodawać do zapisywanego na dysku pliku żadnych kodów końca linii, zmiany linii, zmiany strony itp. Edytor ma tylko umożliwić nam, programistom, wpisanie kodu źródłowego programu. Kod źródłowy programu zawiera rozkazy języka i instrukcje asemblera (translatora). Po napisaniu programu źródłowego, poddajemy go asemblacji. Gdy asembler znajdzie błędy w programie, wówczas musimy je niezwłocznie usunąć z programu, korzystając z edytora, i ponownie dokonujemy asemblacji programu; nowoczesne asemblery mają wbudowaną obszerną listę ewentualnych błędów asemblacji.

Chociaż w otrzymanym po asemblacji programie wynikowym wszelkie symbole zostały już zastąpione odpowiednimi wartościami związanymi z ulokowaniem w pamięci, i dla rozkazów zostały wygenerowane już odpowiednie rozkazy kodów maszynowych, to mimo to program taki nie nadaje się jeszcze do uruchomienia. Po asemblacji standardowo generowany jest plik OBJ; mogą być generowane jeszcze inne pliki dyskowe – LST i CRF. Pierwszy z nich zawiera listing programu – raport asemblacji, a drugi plik – informacje dla listy odwołań. Plik ten może być jeszcze dalej przekształcony na plik dyskowy gotowy do wydruku REF. Interesujący nas plik wynikowy OBJ przekształcamy za pomocą odpowiedniego narzędzia programistycznego, konsolidatora, na program maszynowy gotowy już do wykonania. Konsolidator, linker, pozwala utworzyć program maszynowy gotowy do wykonania z wielu niezależnie tłumaczonych modułów OBJ oraz procedur zawartych w plikach bibliotecznych mających rozszerzenie LIB. W zależności od tego, jakich narzędzi używamy do asemblacji i konsolidacji (linkowania), możliwe jest opcjonalne uzyskanie programu formatu EXE lub COM (COM – jeśli w ogóle pozwala na to konstrukcja programu). Program łączący, konsolidator, może również dodatkowo utworzyć plik z rozszerzeniem MAP, zawierający listę połączeń oraz informujący o uporządkowaniu segmentów w programie.

Powróćmy znowu do programu źródłowego i spójrzmy do jego wnętrza. Program źródłowy zawiera instrukcje asemblera albo dyrektywy asemblera, które w procesie asemblacji tłumaczone są na rozkazy maszynowe. Dyrektywy sterują asemblerem, tzn. „podpowiadają" mu, co on ma robić z instrukcjami i z danymi. Dane te mogą mieć postać dwójkową (binarną), ciąg zerojedynkowy zakończony jest literką B; postać dziesiętną – ciąg cyfr zakończony literką D, i wreszcie postać szesnastkową – ciąg cyfr zakończony literką H. Liczby ujemne w postaci dziesiętnej wprowadza się, poprzedzając je znakiem minus. Liczbę ujemną, kodowaną dwójkowo lub szesnastkowo wprowadza w postaci tzw. uzupełnienia do dwóch. Podczas wprowadzania ciągów znaków i napisów należy je ujmować w cudzysłów bądź w apostrofy.

Gdy z czasem będziemy pisać bardzo długie programy, pamiętajmy o komentarzach, bardzo łatwo bowiem stracić koncepcję podczas pisania kodu źródłowego pro-

gramu posiadającego kilka, a może nawet kilkanaście tysięcy wierszy, a takie programy pisze się często przez wiele dni, a nawet tygodni.

Pamiętajmy! Bardzo długie programy pozostawione bez komentarza to czysta strata czasu, to praca na marne. Nie mamy większych szans, wracając po kilku dniach do niedokończonego złożonego programu, abyśmy wiedzieli, co dalej z tym programem mamy robić. Często lepiej jest zasiąść od nowa do pracy niż szukać – w starym, nie skomentowanym programie – porzuconych idei. Wiemy już, że do tworzenia programu źródłowego potrzebne są instrukcje i dyrektywy, których znaczenie trzeba doskonale rozumieć. Zręcznie też musimy poruszać się w środowisku, w którym „zanurzany" jest źródłowy program asemblerowy. W tym środowisku każda instrukcja programowa ma swoje własne pole.

Tabela 6.1. Podział instrukcji asemblerowej na pola

Pole etykiety	Pole operacji	Pole operandów	Pole komentarza
`Start:`	`MOV`	`CX, Zliczaj`	*;kopiuj bajt do CX*
`Powtarzaj:`	`DEC`	`CX`	*;zmniejszaj CX o 1*
	`STD`		
`Zliczaj:`	`DW`	`35`	

Obowiązkowe jest tylko pole operacji. Pole operandów (argumentów) wystąpi wówczas, gdy będzie tego wymagać rozkaz, np. rozkaz STD nie posiada tego pola. Pole etykiety nie musi być w każdej linii programu. Pole komentarza powinno wystąpić zawsze, i to tylko dla naszego dobra, a nie dla dobra asemblera. Niekiedy w programie może się zdarzyć, że zapis instrukcji asemblerowej w jednej linii będzie zbyt długi i przez to nieczytelny. W takim przypadku – bez żadnego uszczerbku dla programu – zapis programu możemy kontynuować, przenosząc wiersz do następnej linii, nie zapominając o umieszczeniu na końcu poprzedniego wiersza znaku kontynuacji w postaci ukośnika (\). Poniżej zaprezentowano przykład w dwóch równoważnych dla asemblera postaciach.

```
Skok: MOV AX,\
BX

Skok: MOV AX,BX
```

Poprzednio dużo było na temat komentarza, jednak nie wszystko. Otóż, jeśli chcemy objąć nim dużą liczbę linii programu naraz, koniecznie musimy zastosować w programie źródłowym dyrektywę COMMENT (zapisywaną dowolnej wielkości literkami) ze znakiem umieszczonym za nią, i to co najmniej w odległości jednej spacji, oraz z tym samym znakiem zamykającym wybrany blok linii, tak jak to przedstawiono na poniższym przykładzie:

```
Licznik DB ?
X DW 44 DUP(35)
ORG 100H
Start:
COMMENT #                    ;Od tej linii programu asembler pomija wszystkie...
MOV AH,4
ADD AX,CX
...
...
#                            ;... linie programu, dopóki znowu nie „zobaczy" znaku #
END Start
```

Po tym elementarnym wstępie o znaczeniach pól występujących w instrukcji programu przejdźmy do ich bardziej szczegółowego opisu, chociaż nadal pozostaniemy w kręgu ogólnych rozważań, bliżej ducha tej książki.

6.1. Pole etykiety

Etykieta w programie asemblerowym (jeśli już w ogóle wystąpi) może być zakończona dwukropkiem (np. Start:) lub bez dwukropka (np. Licznik). Etykieta bez dwukropka oznacza konkretny obiekt. Na przykład etykieta o nazwie Licznik DB ? umożliwia zarezerwowanie jednej komórki w pamięci operacyjnej wielkości jednego bajta i o nie określonej początkowej wartości, etykieta X DW 44 DUP (14, 35) pozwala zarezerwować 44 słowa (podwójnych bajtów) o wartościach początkowych 14 i 35. Trzeba też wiedzieć, że do etykiet bez dwukropka można odwoływać się z dowolnego miejsca w programie. Inny sens ma etykieta z dwukropkiem. Określa ona pewne przesunięcie adresowe w bieżącym segmencie. Na przykład etykieta Start: wystąpi w programie pod offsetem (przesunięciem) 102H, ponieważ występuje tuż po (pseudorozkazie) ORG 100H, wyznaczającym początkową wartość przesunięcia w programie. Do etykiet z dwukropkiem można odwoływać się tylko z wewnątrz bieżącego segmentu.

Z jakich znaków mogą być budowane etykiety? Do budowy etykiet mamy sporo znaków i mogą być one bardzo długie. Muszą się jednak składać z określonego budulca, a mianowicie z liter od A do Z (asembler nie rozróżnia wielkości liter), cyfr od 0 do 9, znaków specjalnych, np.: ? . @ _ $.

Etykieta nie może zaczynać się od cyfry, chyba że poprzedzimy ją znakiem kropki. Nazwami etykiet nie mogą być symbole oznaczające nazwy rejestrów. Nie wolno też umieszczać znaku spacji wewnątrz etykiety. W sytuacjach gdy w nazwach etykiet koniecznie musimy stawiać widoczne przerwy, to asembler dał nam taką możliwość, a mianowicie zamiast niedozwolonej spacji możemy zastosować znak podkreślenia (_), który również pozwoli nam uzyskać dobry efekt rozdzielenia wyrazów.

Niezależnie o tego, ile i jakie znaki mogą być użyte w nazwie etykiety, zaleca się tworzenie etykiet o nazwach krótkich i łatwych do identyfikacji.

6.2. Pole operacji (pole mnemonika)

Pole operacji zawiera skrót mnemoniczny nazwy rozkazu od dwóch do kilku bajtów. Na przykład najczęściej stosowany rozkaz **MOV** jest skrótem mnemonicznym od angielskiego słowa *move* (przenieś, prześlij). Mnemonik wskazuje asemblerowi ilość i typ operandów (argumentów) potrzebnych do pobrania z pola operandów (argumentów). W rozkazie **SUB** (odejmowanie) rozkaz musi wyraźnie wskazywać, które wyrażenia trzeba od siebie odejmować.

6.3. Pole argumentów (operandów)

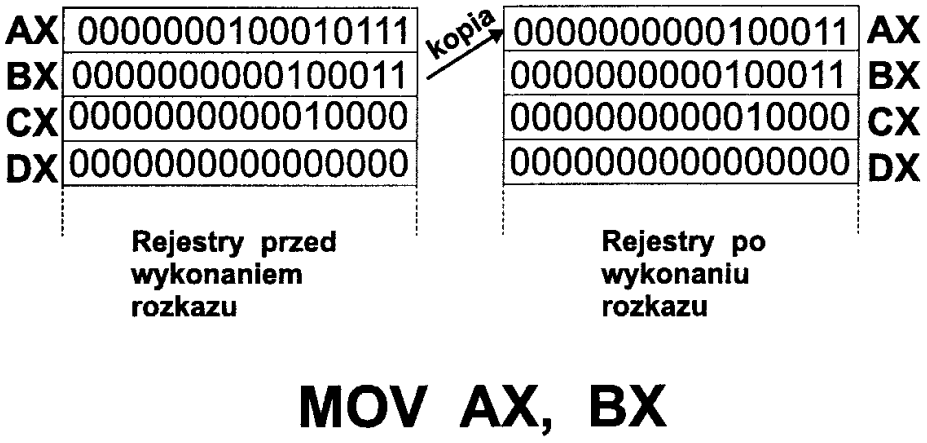
Pole argumentów (operandów) informuje procesor o miejscu przetwarzania danych. Jeśli pole to występuje w rozkazie, to może zawierać jeden, dwa, a nawet trzy (dla najnowszych procesorów) argumenty, oddzielone od mnemonika przynajmniej jedną spacją lub znakiem tabulacji; poszczególne argumenty oddzielane są od siebie przecinkiem. W przypadku rozkazu dwuargumentowego pierwszy argument (tzn. ten bliżej umiejscowiony w stosunku do mnemonika) nazywany jest argumentem przeznaczenia, a drugi argumentem źródłowym. Na przykład w rozkazie **MOV CL, AL** rejestr **AL** jest argumentem źródłowym, a rejestr **CL** argumentem przeznaczenia, docelowym. Zawartość argumentu źródłowego nigdy nie ulega zmianie, natomiast argument przeznaczenia prawie zawsze podlega modyfikacji.

6.4. Pole komentarza

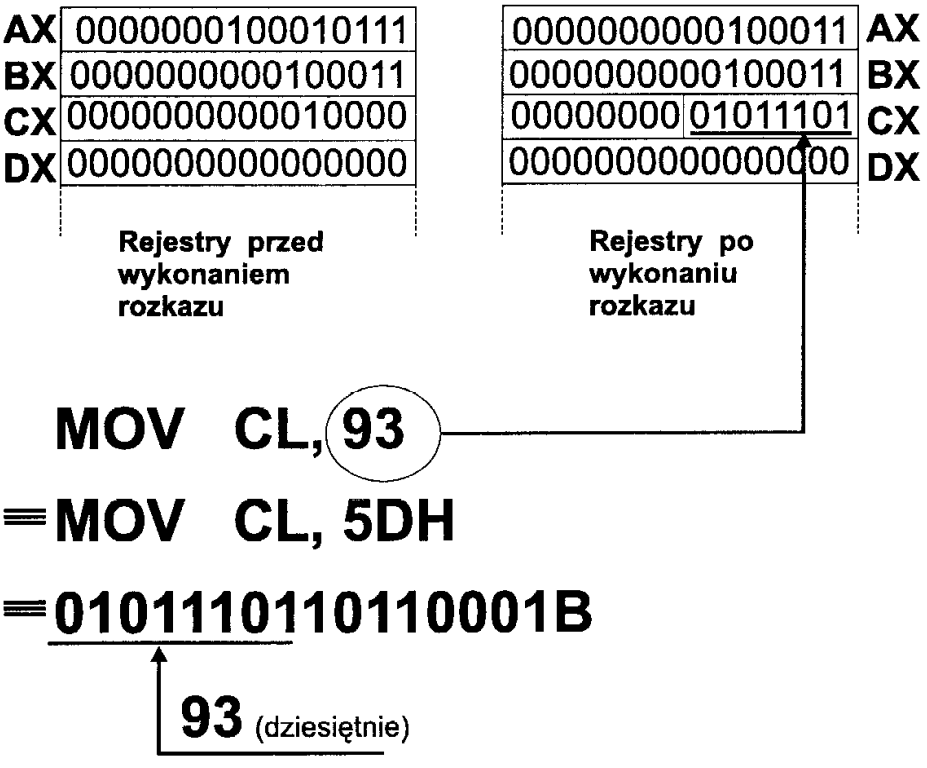
Pole komentarza zostało już dość wyczerpująco omówione, można jedynie przypomnieć, iż każde umieszczenie znaku średnika w linii instrukcji zostanie zignorowane przez asembler podczas asemblacji programu źródłowego. Asembler umieszcza jednak komentarz w tzw. listingach, wydrukach powstałych po asemblacji.

W tym miejscu moglibyśmy zadać sobie proste pytanie, czy mając tyle różnej wiedzy możemy już skonstruować program asemblerowy, zapisując go następnie w pliku dyskowym o rozszerzeniu ASM. Oczywiście moglibyśmy spróbować pokusić się o konstruowanie programu. Jednakże dopóty, dopóki nie poznamy szczegółów poprawnego konstruowania programowania, programowanie w języku Asembler pozbawione będzie sensu i swoistego smaku. Asemblerowe programowanie to nie tylko notacja, rozkazy, dyrektywy, pseudodyrektywy, operatory i inne cegiełki, to również sposoby sięgania po dane do rozkazów, czyli adresowanie, a tego jeszcze nie przerabialiśmy. Najprostszym sposobem (trybem) adresowania jest adresowanie poprzez rejestr. Wiemy już, że w rozkazie – w polu trybu adresowania – zawarta jest (też) in-

formacja o miejscu danej potrzebnej do danego rozkazu. Procesor na podstawie tej informacji określa, który tryb adresowania ma być użyty. Rozpoznanie przez asembler trybu adresowania następuje na podstawie programu źródłowego. Jeżeli argumentami rozkazu będą dwa rejestry, wówczas asembler zakoduje obydwa argumenty w trybie adresowania (adresacji) rejestru, np. **MOV AX, BX**. Taki rodzaj adresowania nazywa się **adresowaniem poprzez rejestr** lub **adresacją rejestrową**.



Rysunek 6.2. Tryb adresowania poprzez rejestr



Rysunek 6.3. Tryb adresowania natychmiastowego

Ten tryb adresowania jest najszybszym i najprostszym sposobem wskazywania argumentów (operandów), podobnie jak **tryb adresowania natychmiastowego**, w którym stała 8-, 16- czy 32-bitowa jako argument źródłowy zawarta jest w rozkazie (umieszczona tam przez asembler), a nie w rejestrze czy w komórce pamięci, np. **MOV CL,-35 MOV CX, 44** czy **MOV EAX,123456**. Argumentem natychmiasto-

wym może być też symbol zdefiniowany przez dyrektywę **EQU**, tak jak to przedstawiono na poniższym przykładzie:

```
LICZBA1 EQU 16
...
MOV CX,LICZBA1
...
```

W asemblerze mamy nieustannie do czynienia z różnego typu liczbami. Zapisując je w instrukcjach programowych musimy zawsze pamiętać, jak one są duże, czy zmieszczą się do zadeklarowanej komórki lub rejestru, który jest zawsze tak samo szeroki.

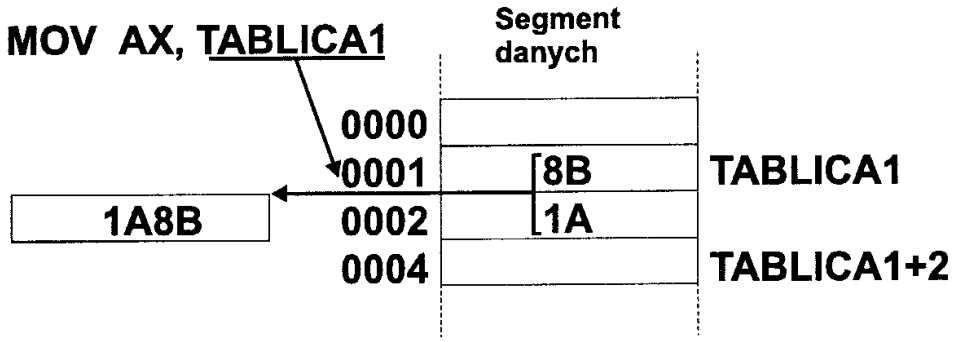
Trzeba wiedzieć, iż liczby 8-bitowe ze znakiem mieszczą się w zakresie od +127(7FH) do -(minus) 128(80H), największa 8-bitowa liczba bez znaku wynosi 255 (0FFH); liczby 16-bitowe ze znakiem mieszczą się w zakresie od +32767 (7FFFH) do -(minus) 32768 (8000H), największa 16-bitowa liczba bez znaku wynosi 65535 (0FFFFH); 32-bitowe liczby ze znakiem mieszczą się w zakresie od + 2147483647 (7FFFFFFFH) do – (minus) 2147483648 (80000000H), największa 32-bitowa liczba bez znaku wynosi 4294967295 (0FFFFFFFFH).

Podczas kopiowania liczby do argumentu przeznaczenia, np. podczas działania rozkazu **MOV AX,1000**, asembler traktuje liczbę 1000 jako tysiąc (w systemie dziesiętnym). W przedstawieniu dwójkowym liczba tysiąc mieć będzie postać: 1111101000. Gdy ta liczba ładowana będzie do 16-bitowego rejestru przeznaczenia (AX), asembler dopisuje do niej z przodu sześć zer, rozszerzając ją do 16-bitowego rejestru. Analogicznie dzieje się dla przypadków z 8- i 32-bitowym rejestrem, gdy wpisywana do nich liczba dodatnia lub ujemna jest o wiele mniejsza niż rozmiar rejestru.

Po tej dłuższej, a jakże ważnej dygresji o rodzajach liczb powróćmy do trybów adresowania. W programach asemblerowych nie tylko stosujemy dwa proste tryby adresowania, poprzez rejestr i natychmiastowe, w których to trybach sięganie po dane lub ich adresy odbywa się bez udziału rejestrów segmentowych lub rejestru stosu; byłoby to zbyt proste, by było prawdziwe.

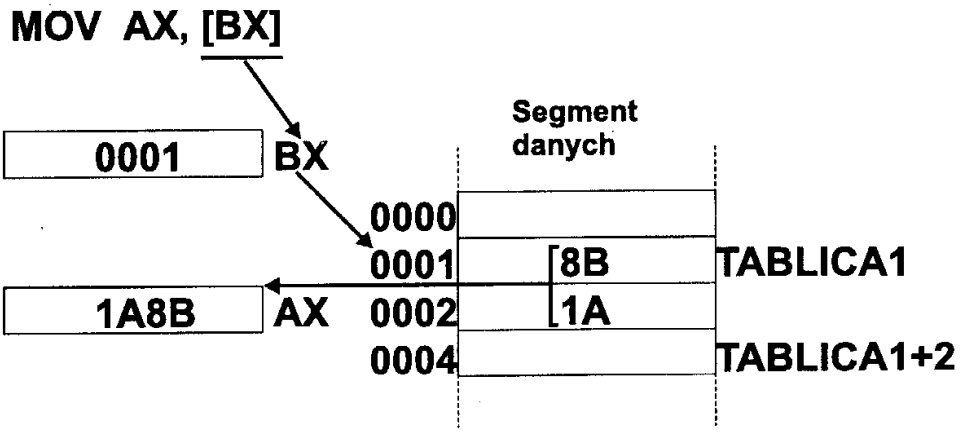
Tak jak w trybie adresowania natychmiastowego, w którym dana zawarta jest w rozkazie, tak też podobnie jest z trybem **adresowania bezpośredniego**, z tą jednakże różnicą, że teraz w rozkazie zawarty jest adres efektywny, a nie dane „natychmiastowe"; adres efektywny wyznacza położenie argumentu od początku segmentu. Argumentem adresowania bezpośredniego jest głównie etykieta, np.:

```
MOV AX, TABLICA1
```



Rysunek 6.4. Tryb adresowania bezpośredniego

Do rejestru **AX** ładowana jest zawartość komórki pamięci z daną o nazwie Tablica1.



Rysunek 6.5. Tryb adresowania pośredniego poprzez rejestr

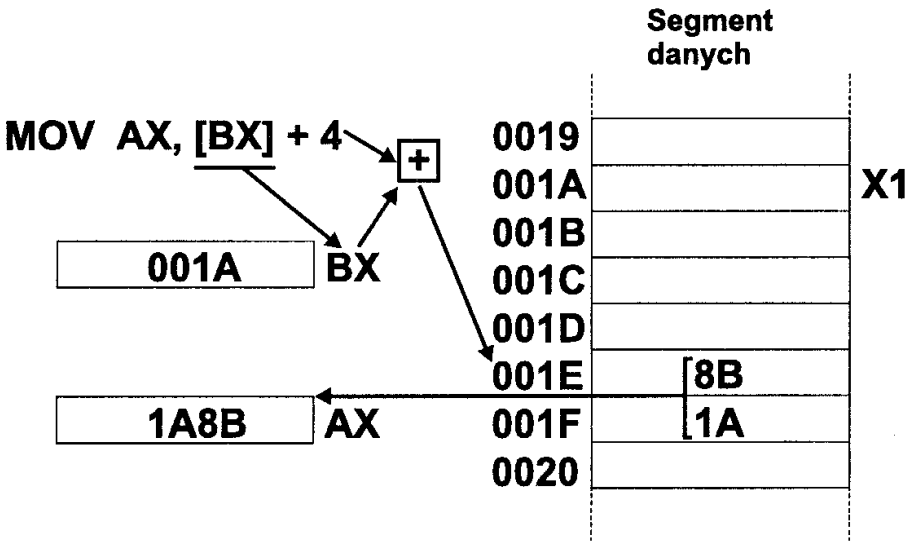
W trybie **adresowania pośredniego poprzez rejestr** adres efektywny argumentu znajduje się w rejestrze bazowym **BX** lub we wskaźniku bazy **BP** (wskaźnik bazy **BP** odnosi się do rejestru **SS**) albo w rejestrze indeksowym **SI** lub **DI**. Rejestry, które są argumentami pośrednimi, zapisujemy w programie obejmując je nawiasem kwadratowym, by móc odróżnić je od rejestrów-argumentów. Na przykład **MOV AX,BX** to zupełnie co innego aniżeli **MOV AX,[BX]**. W pierwszym przypadku do rejestru **AX** ładowana jest zawartość rejestru **BX**, w drugim przypadku ładowana jest zawartość komórki, której adres (względem początku segmentu – tzw. offsetu) wskazuje rejestr **BX**. Aby umieścić offset w rejestrze **BX**, używa się przed adresem pamięci przedrostka OFFSET, na przykład:

```
...
MOV BX,OFFSET TABLICA1
MOV AX,[BX]
...
```

Dwie powyższe instrukcje asemblera wykonują tę samą czynność co instrukcja **MOV AX, TABLICA1**, z tą różnicą, że o ile bezpośrednie przyporządkowanie nazwy komórce pamięci jest jednoznaczne i właściwe przy operowaniu na pojedynczej komórce, o tyle operowanie na wielu komórkach z nieustannym odwoływaniem się do nowego adresu komórki w adresowaniu bezpośrednim byłoby uciążliwe. Stąd też

adresowanie pośrednie stwarza możliwość elastycznych działań na adresach komórek, bez pobierania za każdym razem nowego adresu kolejnej komórki.

Jest jeszcze inny tryb adresowania – **adresowanie pośrednie poprzez rejestr bazowy**. W tym trybie adresowania asembler oblicza adres efektywny przez dodanie zawartości offsetu (przesunięcia) do zawartości rejestru **BX** lub **BP**.



Rysunek 6.6. Tryb adresowania pośredniego poprzez rejestr bazowy

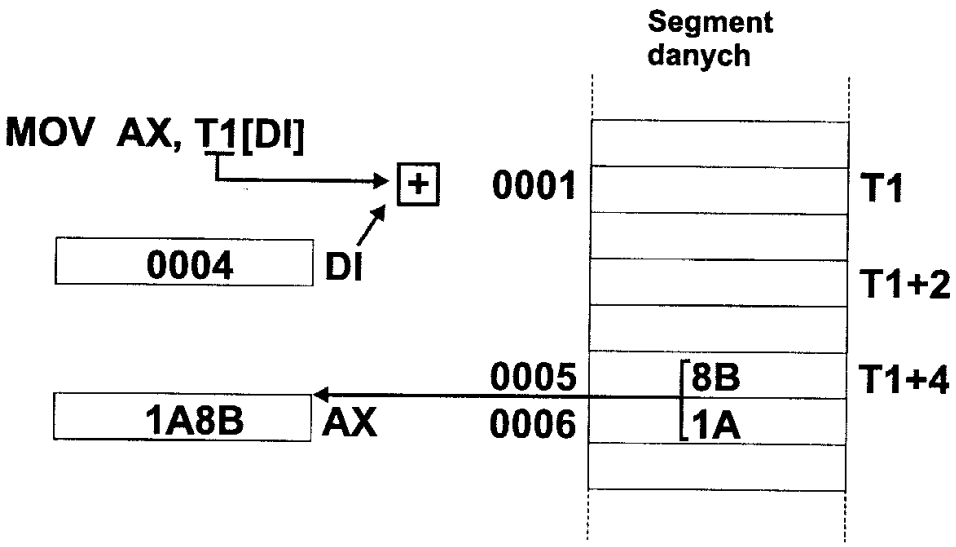
Taki tryb adresowania jest bardzo wygodny w przypadku uzyskiwania dostępów do struktur danych umieszczonych w różnych miejscach pamięci. Adres bazowy struktury umieszczamy w rejestrze bazowym, natomiast do poszczególnych elementów struktury odwołujemy się podając offset (przesunięcie) od adresu bazowego.

W trybie **adresowania indeksowanego bezpośrednio** adres efektywny jest sumą dwóch składników – przesunięcia (offsetu) i rejestru indeksowego **DI** lub **SI**. Taki rodzaj indeksowania jest bardzo wygodny do operowania na tablicach. Przesunięcie wskazuje początek tablicy, natomiast rejestr indeksowy jej element. Na poniższym rysunku w tablicy słów elementy są odległe od siebie o dwa bajty. Dla tablicy T1 instrukcje:

```
MOV DI,4
MOV AX,T1[DI]
```

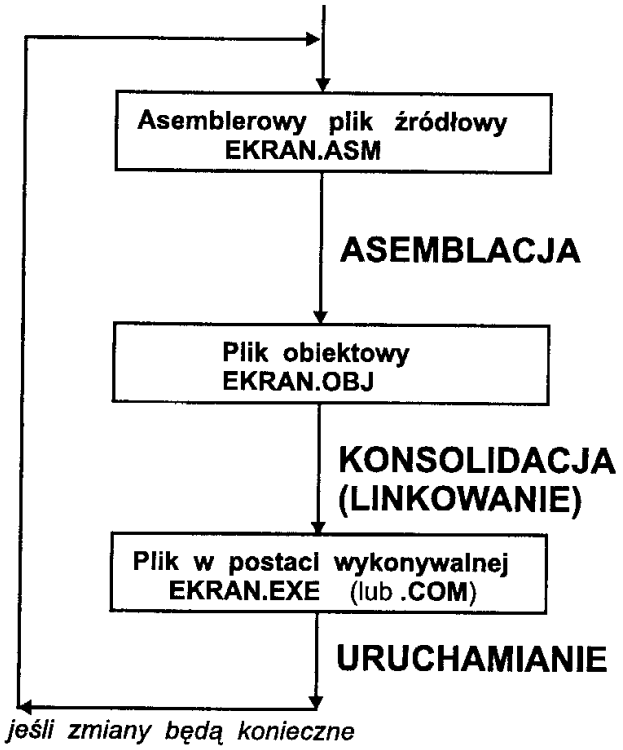
załadują trzeci element tablicy T1 do rejestru **AX**.

Na zakończenie omawiania trybów adresowania zasygnalizujmy jeszcze jeden tryb, a mianowicie **adresowanie indeksowane bezpośrednie poprzez rejestr bazowy**. W tym trybie adresowania adres efektywny jest sumą rejestru bazowego, rejestru indeksowego oraz (ewentualnie) przesunięcia. Ze względu na stosowane w tym trybie adresowania dwa odrębne przesunięcia nadaje się on do operowania na tablicach dwuwymiarowych. W rejestrze bazowym przechowywany będzie adres początkowy tablicy, natomiast przesunięcie i rejestr indeksowy dotyczyć będą wiersza i kolumny.



Rysunek 6.7. Tryb adresowania indeksowego bezpośredniego

Nasza wiedza na temat podstaw programowania w języku Asembler jest już prawie pełna. Wystarczy jeszcze poznać przynajmniej kilka dyrektyw potrzebnych do wytworzenia kodu maszynowego i możemy napisać prosty program asemblerowy. Gotowy już źródłowy program asemblerowy o rozszerzeniu ASM poddajemy – za pomocą odpowiednich narzędzi – asemblacji, a następnie konsolidacji według schematu pokazanego na poniższym rysunku.



Rysunek 6.8. Schemat tworzenia nowego programu

Programów do asemblacji i konsolidacji jest wiele. Najczęściej spotykanymi są: MASM i LINK oraz TASM i TLINK; dwóch różnych firm. Pisanie programów pod

asembler MASM praktycznie różni się (szczegółami) od pisania programów dla TASM. Autor niniejszej książki używać będzie programu TASM, kierując się osobistym przyzwyczajeniem do tego narzędzia.

Spróbujmy skonstruować pierwszy program asemblerowy, tym samym wprawiając całą machinę asemblerową w ruch. Zanim jednak to zrobimy, krótko opisane zostaną niektóre dyrektywy, konieczne do stworzenia programu.

- SEGMENT – dyrektywy SEGMENT i ENDS dzielą program źródłowy na segmenty. Program może mieć cztery rodzaje segmentów: danych, kodu, stosu i dodatkowy. Dyrektywa SEGMENT może posiadać trzy argumenty: typ segmentu, połączenie, klasę. Typ segmentu określa początek adresu rozpoczynającego segment przechowywany w pamięci. Połączenie określa sposób łączenia segmentu z innymi segmentami o tej samej nazwie. Klasa segmentu ma wpływ na kolejność przechowywania segmentów.
- ENDS – jw.
- END – dyrektywa sterująca asemblerem, wyznaczająca asemblerowi koniec asemblacji programu źródłowego. Wszystko to, co wystąpi za dyrektywą END, będzie pomijane podczas asemblacji.
- ASSUME – dyrektywy SEGMENT i ENDS zaznaczają początek i koniec segmentu programu, jednak nie wskazują asemblerowi, jaki rodzaj segmentu jest definiowany. To wskazanie umożliwia dopiero dyrektywa ASSUME. Dyrektywa ma postać ogólną: ASSUME rejestr segmentowy:nazwa segmentu lub ASSUME rejestr segmentowy:NOTHING. Na przykład ASSUME DS: Dane oznacza, iż *segmentem danych jest segment o nazwie Dane, a rejestr DS będzie zawierał* początek segmentu Dane.
- ORG – dyrektywa sterująca asemblerem. Ustawia wskaźnik asemblera na wartość występującą po dyrektywie ORG, powodując przechowywanie przez asembler danych i instrukcji od miejsca wskazanego dyrektywą ORG. Dyrektywa ORG 100H używana jest w konstrukcji programów typu COM, wskazując asemblerowi, aby przechowywanie programu nastąpiło 0100H, czyli 256 bajtów za bieżącą pozycją.

```
MALUJ SEGMENT
ASSUME CS:MALUJ
MOV AX,0000H          ;(1) przesyłamy wartość 0000H do rejestru AX...
MOV DS,AX             ;(2)... gdyż bezpośrednio do DS nie można tego zrobić
MOV AX,0B800H         ;(3) przesyłamy wartość 0B800H pośrednio do...
MOV ES,AX             ;(4)... rejestru AX, a docelowo do ES
MOV SI,0000H          ;(5) adres offsetu pamięci, skąd kopiujemy
MOV DI,1*160D         ;(6) adres offsetu pamięci, dokąd kopiujemy
MOV CX,3*80D          ;(7) 240 znaków (włącznie z ich atrybutami)
CLD                   ;(8) kopiowanie „w przód"
```

```
REP MOVSW                    ;(9) powtarzaj rozkaz MOVSW, aż CX=0
MOV AH, 4CH                  ;(10) funkcja wyjścia z programu...
INT 21H                      ;(11)... do DOS
MALUJ ENDS
END
```

Nadajmy nazwę plikowi, w którym znajduje się powyżej napisany program – **ekran.asm.**

Dokonujemy asemblacji pliku **ekran.asm** przy użyciu programu **TASM.EXE:**

```
TASM ekran[.asm]             ;w nawiasach [ ] – domyślnie
Turbo Assembler Version xxx Copyright xxx
Assembling file: ekran.ASM   ;nazwa asemblowanego pliku
Error messages: None         ;brak błędów asemblacji
Warning messages: None       ;brak ostrzeżeń
Passes: 1                    ;jedno przejście asemblera
Remaining memory: 434k       ;pozostało pamięci operacyjnej (tu: 434KB)
```

Po asemblacji otrzymamy na dysku plik obiektowy **ekran.obj**.

Plik **ekran.obj** poddajemy konsolidacji za pomocą programu **TLINK.EXE**:

```
TLINK ekran[.obj]            ;w nawiasach [ ] – domyślnie
Turbo Link Version xxx Copyright xxx
Warning: No stack            ;ostrzeżenie: brak stosu
```

Na dysku utworzył się gotowy plik do wykonania – **ekran.exe**.

Komentarz dotyczący znaczenia (ponumerowanych) linii pliku **ekran.asm**. Program **ekran.asm** kopiuje obszar pamięci zaadresowany za pomocą rejestrów **DS:SI** (0000H:0000H) – obszar źródłowy, do obszaru pamięci zaadresowanego przez parę rejestrów **ES:DI** (0B800H:00A0H) – obszar docelowy. Kopiowania tego dokonujemy przy użyciu rozkazu **MOVSW**. Obszar docelowy tak został dobrany, aby widać było wyraźnie efekty działania programu. Po prostu obszar docelowy jest częścią pamięci karty VGA (B800-BFFF) i na ekranie zaczynać się będzie od drugiej linii (od góry ekranu – linia programu nr *(6)*), a kończyć na linii piątej (linie programu nr *(6)+(7)*). Powodzenia!

7. Tablica wektorów przerwań

Tablica wektorów przerwań zajmuje obszar pierwszego kilobajta pamięci operacyjnej; 256 wektorów × 4 bajty dla jednego wektora. Adresy programów obsługi przerwań są wpisywane do tablicy wektorów przerwań przy ładowaniu systemu operacyjnego przez BIOS i przez system operacyjny. Przerwania obsługiwane przez BIOS mają numery od 0 do 1FH, chociaż system operacyjny może przechwytywać niektóre z tych przerwań. Zasadniczo pozostałe przerwania obsługiwane są przez system operacyjny, część z nich jest jednak przeznaczona na potrzeby użytkownika. W celu prezentacji tablicy wektorów przerwań zawartej w pamięci skorzystajmy ze znanego nam programu DEBUG.

```
-D 0000:0000
0000:0000 9E 0F CB 00 65 04 70 00-16 00 CF D8 65 04 70 00 ....e.p.....e.p.
0000:0010 65 04 70 00 54 FF 00 F0-58 85 00 F0 6F EF 00 F0 e.p.T...X...o...
0000:0020 00 00 82 09 D2 08 1A DC-6F EF 00 F0 6F EF 00 F0 ........o...o...
0000:0030 6F EF 00 F0 6F EF 00 F0-9A 00 CF D8 65 04 70 00 o...o.......e.p.
0000:0040 13 00 E2 09 4D F8 00 F0-41 F8 00 F0 F7 24 62 FD ....M...A....$b.
0000:0050 39 E7 00 F0 3A 05 98 02-2D 04 70 00 28 0A 83 D0 9...:...-.p.(...
0000:0060 A4 E7 00 F0 2F 00 D9 03-6E FE 00 F0 04 06 83 D0 ..../...n.......
0000:0070 1D 00 82 09 A4 F0 00 F0-22 05 00 00 10 65 00 C0 ........"....e..
```

Ten fragment pamięci pokazuje pierwsze 32 wektory w zapisie szesnastkowym wraz z umieszczonymi obok znakami w kodzie ASCII; kody znaków niedrukowalnych zastępowane są kropkami.

W przedstawionym fragmencie pamięci przykładowo wyróżniono adres czwartego i piątego wektora przerwania: INT 4H, wartość **CS**=0070, **IP**=0465 oraz INT 5H CS=F000, **IP**=FF54. Czcionka opisująca adres przerwania piątego została dodatkowo podkreślona (w pamięci widać zapis zgodny z przechowywaniem odwrotnym).

Tabela 7.1. Tablica przerwań

Nr przerwania – dziesiętnie	**Nr przerwania** – szesnastkowo	**Adres wektora*** – szesnastkowo	**Znaczenie**
0	0	0000-0003	Dzielenie przez zero
1	1	0004-0007	Praca krokowa
2	2	0008-000B	Przerw. niemaskowalne
3	3	000C-000F	Pułapka (gener. przez INT3)
4	4	0010-0013	Nadmiar

Nr przerwania – dziesiętnie	Nr przerwania – szesnastkowo	Adres wektora* – szesnastkowo	Znaczenie
5	5	0014-0017	Wydruk ekranu monitora
6	6	0018-001B	Zarezerwowane
7	7	001C-001F	Zarezerwowane
8	8	0020-0023	Przerw. zegara; IRO0
9	9	0024-0027	Przerw. klawiatury; IRQ1
10	A	0028-002B	Zarezerwowane; IRQ2
11	B	002C-002F	Przerw. sterow.(2).; IRQ3
12	C	0030-0033	Przerw. łącza szereg.; IRQ4
13	D	0034-0037	Sterow. dysk. tward.; IRQ5
14	E	0038-003B	Sterow. dysk. mięk.; IRQ6
15	F	003C-003F	Sterow. drukark.; IRQ7
16	10	0040-0043	Obsługa ekranu monitora
17	11	0044-0047	Konfiguracja systemu
18	12	0048-004B	W AX wielk. pam; (40:13)
19	13	004C-004F	Obsługa dyskietki/dysku
20	14	0050-0053	Obsł. transmisji szeregowej
21	15	0054-0057	Rozmiar pamięci rozszerz.
22	16	0058-005B	Obsługa klawiatury
23	17	005C-005F	Obsługa drukarki
24	18	0060-0063	Basic
25	19	0064-0067	Ładowanie systemu operac.
26	1A	0068-006B	Obsługa zegara system.
27	1B	006C-006F	Obsł. klaw. CTRL-BREAK
28	1C	0070-0073	Obsł. przerw. zegarowego
29	1D	0074-0077	Param. sterow. ekranu
30	1E	0078-007B	Param. napędu dyskietek
31	1F	007C-007F	Tabl. znak. graf. (128-255)
32	20	0080-0083	Zakończenie programu
33	21	0084-0087	Wywołanie funkcji DOS
34	22	0088-008B	Adres powrot. po zak. progr.
35	23	008C-008F	Obsł. przerw. wykon. progr.
36	24	0090-0093	Obsł. błędów krytycznych
37	25	0094-0097	Odczyt sektorów z dysku

Nr przerwania – dziesiętnie	Nr przerwania – szesnastkowo	Adres wektora* – szesnastkowo	Znaczenie
38	26	0098-009B	Zapis sektorów na dysku
39	27	009C-009F	Zak. i pozost. progr. w pam.
40	28	00A0-00A3	DOS w stanie jałowym
41	29	00A4-00A7	Szybkie wysł. zn. na konsol.
42	2A	00A8-00AB	DOS w stanie jałowym
43-45	2B-2D	00A4-00B4	Zarezer. dla DOS
46	2E	00B8-00BB	Wyk. polecenia systemowe
47	2F	00BC-00BF	Przerw. multipleksowe
48	30	00C0-00C3	We. do fun. sys. DOS
49	31	00C4-00C7	We. do fun. sys. DOS/DPMI
50-63	32-3F	00C8-00FF	Zarezer. dla system. operac.
64	40	0100-0103	Obsł. dyskie. (gdy jest dysk)
65	41	0104-0107	Param. dysku. twardego 1
66	42	0108-010B	Ob. ka. CGA(gdy jest EGA)
67	43	010C-010F	Znaki graf. kart EGA/VGA
68	44	0110-0113	Znaki graf. karty EGA
69	45	0114-0117	Zarezer. dla BIOS
70	46	0118-011B	Param. dysku twardego 2
71-73	47-49	011C-0127	Zarezer. dla BIOS
74	4A	0128-012B	Alarm (AT)
75-95	4B-5F	012C-017F	Zarezer. dla BIOS
96-103	60-67	0180-019F	Zarezer. dla prog. użytk.
104-111	68-6F	01A0-01BF	Nie używane
112	70	01C0-01C3	Zarezer.;IRQ8
113	71	01C4-01C7	Zarezer.;IRQ9
114	72	01C8-01CB	Zarezer.;IRQ10
115	73	01CC-01CF	Zarezer.;IRQ11
116	74	01D0-01D3	Zarezer.;IRQ12
117	75	01D4-01D7	Przerw. koprocesora;IRQ13
118	76	01D8-01DB	Sterow. dysk. tward.;IRQ14
119	77	01DC-01DF	Zarezer.;IRQ15
120-127	78-7F	01E0-01FF	Nie używane
128-133	80-85	0200-0217	Zarezer. dla Basica

Nr przerwania – dziesiętnie	**Nr przerwania** – szesnastkowo	**Adres wektora*** – szesnastkowo	**Znaczenie**
134-240	86-F0	0218-03C3	Używane przez Basic
241-255	F1-FF	03C4-03FF	Nie używane

Skróty i oznaczenia:

Adres powrot. po zak. progr. – Adres powrotny po zakończeniu programu

gener. – generowane

Ładowanie systemu operac. – Ładowanie systemu operacyjnego

Obsł. – Obsługa

Param. – Parametry

Przerw. – Przerwanie

Sterow. – Sterownik(a)

Zarezer. – Zarezerwowane

Ob. ka. CGA – Obsługa karty graficznej CGA

Obsł. dyskie. – Obsługa dyskietki

Obsł. przerw. wykon. progr. – Obsługa przerwania wykonywanego programu

Przerw. łącza szereg. – Przerwanie łącza szeregowego

Przerw. sterow. (2) – Przerwanie sterownika (2)

Sterow. drukark. – Sterownik drukarki

Sterow. dysk. mięk. – Sterownik dysku miękkiego

Sterow. dysk. tward. – Sterownik dysku twardego

Szybkie wysł. zn. na konsol. – Szybkie wysłanie znaku na konsolę

Tabl. znak. graf. – Tablica znaków graficznych

W AX wielk. pam. – W rejestrze AX wielkość pamięci

We. do fun. sys. – Wejście do funkcji systemowej

Wyk. polecenia systemowe – Wykonano polecenie systemowe

Zak. i pozost. progr. w pam. – Zakończenie i pozostawienie programu w pamięci

Zarezer. dla prog. użytk. – Zarezerwowane dla programów użytkowych

Zarezer. dla system. operac. – Zarezerwowane dla systemu operacyjnego

Znaki graf. – Znaki graficzne

* Adres wektora przerwania (jego początek) obliczamy według następującego wzoru: **nr przerwania** × **4**; każdy wektor w tablicy wektorów przerwań zajmuje 4 bajty **adres końca wektora przerwania** = **adres początku+3**; wektory liczone są od zera, np.: 4A×4=128H, 128+3=12BH (szesnastkowo) lub dziesiętnie 74×4=296(128H), 296+3=299=12BH.

8. Dwa przerwania najczęściej używane w programach asemblerowych

Najczęściej używanymi przerwaniami w programach asemblerowych są:

- przerwanie 10H (16D) – obsługa ekranu monitora,
- przerwanie 21H (33D) – wywołanie funkcji DOS.

Ze względu na to, iż niektóre funkcje dotyczące zarówno przerwania 10H, jak też 21H są zbyt trudne w użyciu, zwłaszcza dla początkującego programisty, w rozdziale tym opisane zostaną tylko te funkcje, które są stosunkowo łatwe w użyciu i często stosowane w programach.

INT 10H

Funkcje przerwania 10H realizowane są wpisaniem zawartości do rejestru AH:

AH=0 – ustawienie trybu pracy karty graficznej

AH=1 – zdefiniowanie rozmiaru kursora

Wejście:
CH – numer linii w wierszu, w której zaczyna się kursor
CL – numer linii w wierszu, na której kończy się kursor

AH=2 – ustawienie pozycji kursora na ekranie

Wejście:
BH – numer strony
DH – numer wiersza
DL – numer kolumny

AH=3 – odczytanie położenia kursora na ekranie

Wejście:
BH – numer strony
Wyjście:

DH – numer wiersza
DL – numer kolumny
CH i CL – aktualnie ustawiony typ kursora (jak dla funkcji 1H)

AH=5 – zmiana strony aktywnej

Wejście:
AL – numer strony aktywnej

AH=6 – przewijanie strony aktywnej w górę

AH=7 – przewijanie strony aktywnej w dół

Wejście:
AL – liczba wierszy, które mają być wygaszone na dole okna
CH – numer wiersza górnego lewego rogu okna
CL – numer kolumny górnego lewego rogu okna
DH – numer wiersza dolnego prawego rogu okna
DL – numer kolumny dolnego prawego rogu okna
BH – atrybut (wygaszonych) wierszy

AH=8 – odczytanie znaku i atrybutu w miejscu ustawienia kursora

Wejście:
BH – numer strony
Wyjście:
AL – kod znaku
AH – atrybut znaku

AH=9 – zapisanie znaku i atrybutu w miejscu ustawienia kursora

Wejście:
AL – kod znaku
AH – atrybut znaku
CX – liczba znaków do zapisu

AH=10 (0AH) – zapisanie znaku bez atrybutu w miejscu ustawienia kursora

Wejście:
AL – kod znaku
CX – liczba znaków do zapisu

AH=11 (0BH) – wybór palety kolorów

Wejście:
BH – numer palety
BL – numer koloru z wybranej palety kolorów

AH=12 (0CH) – wyświetlenie punktu

Wejście:
BH – numer strony
DX – numer wiersza
CX – numer kolumny
AL – kolor punktu

AH=13 (0DH) – odczytanie punktu

Wejście:
BH – numer strony
DX – numer wiersza
CX – numer kolumny
Wyjście:
AL – kolor punktu (podanego miejsca)

AH=15 (0FH) – odczytanie stanu ekranu

Wejście:
AL – tryb pracy
AH – liczba kolumn
BH – numer aktywnej strony

INT 21H

Funkcje przerwania 21H realizowane są wpisaniem zawartości do rejestru AH:

AH=01H – odczytanie znaku z klawiatury i wysłanie echa na ekran

Wyjście:
AL – odebrany znak

AH=02H – wysłanie znaku na ekran

Wejście:
DL – kod znaku do wysłania

AH=05H – wydrukowanie znaku

Wejście:
DL – znak do wydrukowania

AH=07H – odczytanie znaku z konsoli

Wyjście:
AL – znak odebrany z klawiatury

AH=08H – odczytanie znaku z klawiatury

Wyjście:
AL – znak odebrany z klawiatury; brak echa, wykrywany jest CTRL-BREAK

AH=09H – wyświetlanie na ekranie łańcucha znaków (tekstu)

Wejście:
DS:DX – adres początku łańcucha, łańcuch kończy się znakiem $

AH=0BH – sprawdzenie, czy w buforze jest podany znak z klawiatury

Wyjście:
AL – FFH – znak gotowy do odebrania
AL – 00H – brak znaku

AH=0EH – wybór stacji dysków

Wejście:
DL – numer stacji dysku (A = 0, B = 1 itd.)
Wyjście:
AL – liczba dysków logicznych w systemie

AH=19H – dysk roboczy

Wyjście:
AL – numer dysku roboczego (A = 0, B = 1 itd.)

AH=1AH – deklaracja bufora transmisji dyskowych

Wejście:
DS:DX – wskaźnik do obszaru deklarowanego bufora

AH=25H – zapisanie wektora przerwań

Wejście:
AL – numer przerwania
DS:DX – adres nowej procedury obsługi przerwania

AH=2FH – informacja o położeniu bufora transmisji dyskowych

Wyjście:
ES:BX – wskaźnik do bieżącego bufora transmisji dyskowych

AH=31 – zakończenie procesu z pozostawieniem w pamięci

Wejście:
AL – kod powrotu
DX – rozmiar programu pozostawionego w pamięci (w paragrafach)

Uwaga! Zamiast przerwania INT 21H – funkcja 31H, bywa jeszcze stosowane starsze przerwanie 27H(39D), które również umożliwia zakończenie wykonywanego

programu z pozostawieniem go w pamięci. Rejestry CS:DX powinny zawierać adres następnego bajtu po ostatnim bajcie kodu programu.

AH=35H – odczytanie wektora przerwań

Wejście:
AL – numer przerwania
ES:BX – adres procedury obsługi przerwania

AH=39H – utworzenie podkatalogu

Wejście:
DS:DX – wskaźnik do specyfikacji katalogu (ciąg ASCII+0)
Wyjście:
znacznik CF = 0 – poprawne wykonanie
znacznik CF = 1 – błąd; kody błędów w AX

AH=3AH – usunięcie podkatalogu

Wejście:
DS:DX – wskaźnik do specyfikacji katalogu (ciąg ASCII+0)
Wyjście:
znacznik CF = 0 – poprawne wykonanie
znacznik CF = 1 – błąd; kody błędów w AX

AH=3BH – zmiana katalogu roboczego

Wejście:
DS:DX – wskaźnik do specyfikacji nowego katalogu roboczego (ciąg ASCII+0)
Wyjście:
znacznik CF = 0 – poprawne wykonanie
znacznik CF = 1 – błąd; kody błędów w AX

AH=3CH – utworzenie pliku

Wejście:
CX – atrybuty pliku
DS:DX – wskaźnik do specyfikacji pliku (ciąg ASCII+0)
Wyjście:
znacznik CF = 0 – poprawne wykonanie, w AX uchwyt pliku
znacznik CF = 1 – błąd; kody błędów w AX

Uchwyt pliku – liczba 16-bitowa przekazywana przez funkcje systemu podczas tworzenia pliku lub podczas otwarcia istniejącego już pliku. W systemie określono pięć uchwytów pliku (od 0 do 4):

0000H – standardowe wejście,

0001H – standardowe wyjście,

0002H – standardowe urządzenie do wysyłania komunikatów o błędach,

0003H – standardowe łącze szeregowe,

0004H – standardowa drukarka,

AH=3DH – otwarcie pliku

Wejście:
AL – tryb otwarcia pliku
DS:DX – wskaźnik do specyfikacji pliku (ciąg ASCII+0)
Wyjście:
znacznik CF = 0 – poprawne wykonanie, w AX uchwyt pliku
znacznik CF = 1 – błąd; kody błędów w AX

AH=3EH – zamknięcie pliku

Wejście:
BX – uchwyt pliku
Wyjście:
znacznik CF = 0 – poprawne wykonanie
znacznik CF = 1 – błąd; kody błędów w AX

AH=3FH – odczytanie z pliku

Wejście:
BX – uchwyt pliku
CX – liczba bajtów do odczytania
DS:DX – wskaźnik do bufora (miejsca odczytania danych)
Wyjście:
znacznik CF = 0 – poprawne wykonanie; AX – liczba odczytanych bajtów
znacznik CF = 1 – błąd; kody błędów w AX

AH=40H – zapis do pliku

Wejście:
BX – uchwyt pliku
CX – liczba bajtów do odczytania
DS:DX – wskaźnik do bufora (miejsca pobierania zapisywanych danych)
Wyjście:
znacznik CF = 0 – poprawne wykonanie; AX – liczba zapisanych bajtów
znacznik CF = 1 – błąd; kody błędów w AX

AH=41H – skasowanie pliku

Wejście:
DS:DX – wskaźnik do specyfikacji do kasowanego pliku (ciąg ASCII+0)
Wyjście:

znacznik CF = 0 – poprawne wykonanie
znacznik CF = 1 – błąd; kody błędów w AX

AH=42H – przesunięcie wskaźnika odczytu-zapisu pliku

Wejście:
AL – rodzaj przesunięcia (0 – względem początku pliku, 1 – względem bieżącej pozycji, 2 – względem końca pliku)

Uwaga! Dla AL = 1 lub 2 wartość podawana w CX:DX jest traktowana jako liczba całkowita ze znakiem (kod U2).

BX – uchwyt pliku
CX:DX – wielkość przesunięcia (liczba 4-bajtowa, w CX – część najbardziej znacząca)
Wyjście:
znacznik CF = 0 – poprawne wykonanie; DX:AX – nowa pozycja wskaźnika odczytu-zapisu względem początku pliku (liczba 4-bajtowa, w DX część najbardziej znacząca)
znacznik CF = 1 – błąd; kody błędów w AX

AH=4CH – zakończenie procesu

Wejście:
AL – kod powrotu; dostępny przez funkcję 4DH

Dodatek A. Kod ASCII

Kod ASCII (American Standard Code for Information Interchange) jest specjalnym kodem lub systemem, który zamienia duże litery, małe litery, liczby, znaki interpunkcyjne oraz znaki specjalne na liczby od 0 do 127, również na odwrót. W przypadku komputera IBM PC i w wielu komputerach kompatybilnych używany jest rozszerzony kod ASCII, zaprojektowany na 8 bitach. Kod ten zawiera dodatkowo: symbole matematyczne, proste znaki graficzne, znaki specjalne, które są reprezentowane liczbami z przedziału od 128 do 255. Znaki ASCII z przedziału 0–31 oraz 127, 128 do 159 (włącznie) i 255 są znakami sterującymi np.: kursorem na ekranie, drukarką.

Tabela A.1. Znaki ASCII

Znak	Dziesiętnie	Szesnastkowo	Dwójkowo
NUL	0	00	0000 0000
☺ (SOH)	1	01	0000 0001
☻ (STX)	2	02	0000 0010
♥ (ETX)	3	03	0000 0011
♦ (EOT)	4	04	0000 0100
♣ (ENQ)	5	05	0000 0101
♠ (ACK)	6	06	0000 0110
● (BEL)	7	07	0000 0111
◘ (BS)	8	08	0000 1000
○ (HT)	9	09	0000 1001
◙ (LF)	10	0A	0000 1010
♂ (VT)	11	0B	0000 1011
♀ (FF)	12	0C	0000 1100
♪ (CR)	13	0D	0000 1101
♫ (SO)	14	0E	0000 1110
☼ (SI)	15	0F	0000 1111
► (DLE)	16	10	0001 0000
◄ (DC1)	17	11	0001 0001
↕ (DC2)	18	12	0001 0010
‼ (DC3)	19	13	0001 0011

Znak	Dziesiętnie	Szesnastkowo	Dwójkowo
¶ (DC4)	20	14	0001 0100
§ (NAK)	21	15	0001 0101
– (SYN)	22	16	0001 0110
↨ (ETB)	23	17	0001 0111
↑ (CAN)	24	18	0001 1000
↓ (EM)	25	19	0001 1001
→ (SUB)	26	1A	0001 1010
← (ESC)	27	1B	0001 1011
∟ (FS)	28	1C	0001 1100
↔ (GS)	29	1D	0001 1101
▲ (RS)	30	1E	0001 1110
▼ (US)	31	1F	0001 1111
odstęp	32	20	0010 0000
!	33	21	0010 0001
"	34	22	0010 0010
#	35	23	0010 0011
$	36	24	0010 0100
%	37	25	0010 0101
&	38	26	0010 0110
'	39	27	0010 0111
(	40	28	0010 1000
)	41	29	0010 1001
*	42	2A	0010 1010
+	43	2B	0010 1011
,	44	2C	0010 1100
–	45	2D	0010 1101
.	46	2E	0010 1110
/	47	2F	0010 1111
0	48	30	0011 0000
1	49	31	0011 0001
2	50	32	0011 0010
3	51	33	0011 0011
4	52	34	0011 0100
5	53	35	0011 0101
6	54	36	0011 0110
7	55	37	0011 0111

Znak	Dziesiętnie	Szesnastkowo	Dwójkowo
8	56	38	0011 1000
9	57	39	0011 1001
:	58	3A	0011 1010
;	59	3B	0011 1011
<	60	3C	0011 1100
=	61	3D	0011 1101
>	62	3E	0011 1110
?	63	3F	0011 1111
@	64	40	0100 0000
A	65	41	0100 0001
B	66	42	0100 0010
C	67	43	0100 0011
D	68	44	0100 0100
E	69	45	0100 0101
F	70	46	0100 0110
G	71	47	0100 0111
H	72	48	0100 1000
I	73	49	0100 1001
J	74	4A	0100 1010
K	75	4B	0100 1011
L	76	4C	0100 1100
M	77	4D	0100 1101
N	78	4E	0100 1110
O	79	4F	0100 1111
P	80	50	0101 0000
Q	81	51	0101 0001
R	82	52	0101 0010
S	83	53	0101 0011
T	84	54	0101 0100
U	85	55	0101 0101
V	86	56	0101 0110
W	87	57	0101 0111
X	88	58	0101 1000
Y	89	59	0101 1001
Z	90	5A	0101 1010
[	91	5B	0101 1011
\	92	5C	0101 1100

Znak	Dziesiętnie	Szesnastkowo	Dwójkowo
]	93	5D	0101 1101
^	94	5E	0101 1110
_	95	5F	0101 1111
`	96	60	0110 0000
a	97	61	0110 0001
b	98	62	0110 0010
c	99	63	0110 0011
d	100	64	0110 0100
e	101	65	0110 0101
f	102	66	0110 0110
g	103	67	0110 0111
h	104	68	0110 1000
i	105	69	0110 1001
j	106	6A	0110 1010
k	107	6B	0110 1011
l	108	6C	0110 1100
m	109	6D	0110 1101
n	110	6E	0110 1110
o	111	6F	0110 1111
p	112	70	0111 1000
q	113	71	0111 0001
r	114	72	0111 0010
s	115	73	0111 0011
t	116	74	0111 0100
u	117	75	0111 0101
v	118	76	0111 0110
w	119	77	0111 0111
x	120	78	0111 1000
y	121	79	0111 1001
z	122	7A	0111 1010
{	123	7B	0111 1011
|	124	7C	0111 1100
}	125	7D	0111 1101
~	126	7E	0111 1110
⌂ (DEL)	127	7F	0111 1111
Ç	128	80	1000 0000

Znak	Dziesiętnie	Szesnastkowo	Dwójkowo
ü	129	81	1000 0001
é	130	82	1000 0010
â	131	83	1000 0011
ä	132	84	1000 0100
à	133	85	1000 0101
å	134	86	1000 0110
ç	135	87	1000 0111
ê	136	88	1000 1000
ë	137	89	1000 1001
è	138	8A	1000 1010
ï	139	8B	1000 1011
î	140	8C	1000 1100
ì	141	8D	1000 1101
Ä	142	8E	1000 1110
Å	143	8F	1000 1111
É	144	90	1001 0000
æ	145	91	1001 0001
Æ	146	92	1001 0010
ô	147	93	1001 0011
ö	148	94	1001 0100
ò	149	95	1001 0101
û	150	96	1001 0110
ù	151	97	1001 0111
ÿ	152	98	1001 1000
Ö	153	99	1001 1001
Ü	154	9A	1001 1010
¢	155	9B	1001 1011
£	156	9C	1001 1100
¥	157	9D	1001 1101
₧	158	9E	1001 1110
ƒ	159	9F	1001 1111
	160	A0	1010 0000

Znak	Dziesiętnie	Szesnastkowo	Dwójkowo
í	161	A1	1010 0001
ó	162	A2	1010 0010
ú	163	A3	1010 0011
ñ	164	A4	1010 0100
Ñ	165	A5	1010 0101
ª	166	A6	1010 0110
º	167	A7	1010 0111
¿	168	A8	1010 1000
⌐	169	A9	1010 1001
¬	170	AA	1010 1010
½	171	AB	1010 1011
¼	172	AC	1010 1100
¡	173	AD	1010 1101
«	174	AE	1010 1110
»	175	AF	1010 1111
░	176	B0	1011 0000
▒	177	B1	1011 0001
▓	178	B2	1011 0010
│	179	B3	1011 0011
┤	180	B4	1011 0100
╡	181	B5	1011 0101
╢	182	B6	1011 0110
╖	183	B7	1011 0111
╕	184	B8	1011 1000
╣	185	B9	1011 1001
║	186	BA	1011 1010
╗	187	BB	1011 1011
╝	188	BC	1011 1100
╜	189	BD	1011 1101
╛	190	BE	1011 1110
┐	191	BF	1011 1111
└	192	C0	1100 0000
┴	193	C1	1100 0001
┬	194	C2	1100 0010
├	195	C3	1100 0011
─	196	C4	1100 0100

Znak	Dziesiętnie	Szesnastkowo	Dwójkowo
┼	197	C5	1100 0101
╞	198	C6	1100 0110
╟	199	C7	1100 0111
╚	200	C8	1100 1000
╔	201	C9	1100 1001
╩	202	CA	1100 1010
╦	203	CB	1100 1011
╠	204	CC	1100 1100
═	205	CD	1100 1101
╬	206	CE	1100 1110
╧	207	CF	1100 1111
╨	208	D0	1101 0000
╤	209	D1	1101 0001
╥	210	D2	1101 0010
╙	211	D3	1101 0011
╘	212	D4	1101 0100
╒	213	D5	1101 0101
╓	214	D6	1101 0110
╫	215	D7	1101 0111
╪	216	D8	1101 1000
┘	217	D9	1101 1001
┌	218	DA	1101 1010
▉	219	DB	1101 1011
▄	220	DC	1101 1100
▌	221	DD	1101 1101
▐	222	DE	1101 1110
▀	223	DF	1101 1111
α	224	E0	1110 0000
ß	225	E1	1110 0001
Γ	226	E2	1110 0010
π	227	E3	1110 0011
∑	228	E4	1110 0100
σ	229	E5	1110 0101
µ	230	E6	1110 0110
τ	231	E7	1110 0111
Φ	232	E8	1110 1000

Znak	Dziesiętnie	Szesnastkowo	Dwójkowo
Θ	233	E9	1110 1010
Ω	234	EA	1110 1010
δ	235	EB	1110 1011
∞	236	EC	1110 1100
ø	237	ED	1110 1101
ε	238	EE	1110 1110
∩	239	EF	1110 1111
≡	240	F0	1111 0000
±	241	F1	1111 0001
≥	242	F2	1111 0010
≤	243	F3	1111 0011
⌠	244	F4	1111 0100
⌡	245	F5	1111 0101
÷	246	F6	1111 0110
≈	247	F7	1111 0111
°	248	F8	1111 1000
∙	249	F9	1111 1001
·	250	FA	1111 1010
√	251	FB	1111 1011
n	252	FC	1111 1100
2	253	FD	1111 1101
■	254	FE	1111 1110
□	255	FF	1111 1111

NUL – znak pusty;

SOH – *start of heading* (początek nagłówka zawierającego adres lub polecenie);

STX – *start of text* (początek tekstu);

ETX – *end of text* (koniec tekstu);

EOT – *end of transmission* (koniec transmisji);

ENQ – *enquiry* (zapytanie);

ACK – *acknowledge* (potwierdzenie);

BEL – *bell* (dzwonek);

BS – *backspace* (cofnięcie mechanizmu drukującego lub kursora o jedną pozycję wstecz);

HT – *tab* (ht) (tabulacja, przesunięcie mechanizmu drukującego lub kursora do następnej pozycji tabulacji);

LF – *line feed* (wysunięcie papieru lub przejście kursora do następnego wiersza);

VT – *vertical tabulation* (tabulacja pionowa, ruch mechanizmu drukującego lub kursora do następnego ustalonego wiersza);

CR – *carriage return* (powrót karetki, ruch mechanizmu drukującego lub kursora do początkowej pozycji w tym samym wierszu);

SO – *shift out* (wyjście z kodu, ostrzeżenie, że następne znaki nie mogą być interpretowane jako znaki ASCII, aż do zatrzymania znaku SI);

SI – *shift in* (powrót do kodu, dalsze znaki należą już do znaków ASCII);

DLE – *data line escape* (zmiana znaczenia następnego znaku, który nie powinien być traktowany jako znak ASCII, lecz jako kombinacja bitów sterująca pracą urządzenia);

DC1, DC2, DC3, DC4 – *device control* (sterowanie urządzeniami nr 1, 2, 3, 4);

NAK – *negative acknowledge* (brak potwierdzenia, np. na nieprawidłową informację);

SYN – *synchronous file* (znak synchronizacyjny);

ETB – *end of transmission block* (koniec transmisji bloku, znak stosowany do podziału przesyłanego tekstu na mniejsze bloki);

CAN – *cancel* (anulowanie danych zawartych w bloku);

EM – *end of medium* (koniec nośnika informacji);

SUB – *substitute* (postawienie, znak wstawiony w odbiorniku w miejsce błędnie odebranego znaku, np. po wykryciu błędu parzystości);

ESC – *escape* (zmiana kodu dla określonej liczby następnych znaków);

FS, GS, RS, US – kody sterujące kursorem we wszystkich czterech kierunkach;

DEL – unieważnienie znaku.

Dodatek B. Potęgi liczby 2 i liczby 16

Tabela B.1. Potęgi liczby 2

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1 024
11	2 048
12	4 096
13	8 192
14	16 384
15	32 768
16	65 536
17	131 072
18	262 144
19	524 288
20	1 048 576

n	2^n
21	2 097 152
22	4 194 304
23	8 388 608
24	16 777 216
25	33 554 432
26	67 108 864
27	134 217 728
28	268 435 456
29	536 870 912
30	1 073 741 824
31	2 147 483 648
32	4 294 967 296
33	8 589 934 592
34	17 179 869 184
35	34 359 738 368
36	68 719 476 736
37	137 438 953 472
38	274 877 906 944
39	549 755 813 888
40	1 099 511 627 776

Tabela B.2. Potęgi liczby 16

n	16^n
0	1
1	16
2	256
3	4 096
4	65 536
5	1 048 576
6	16 777 216
7	268 435 456
8	4 294 967 296
9	68 719 476 736
10	1 099 511 627 776

Dodatek C. Arytmetyka dwójkowa i szesnastkowa

C.1. Zamiana liczby dziesiętnej na dwójkową i odwrotnie

W systemie dwójkowym mamy tylko dwie cyfry: 0 i 1.

Przedstawmy liczbę dziesiętną **1996** w postaci dwójkowej:

1996:2=998 reszty 0
998:2=499 reszty 0
499:2=249 reszty 1
249:2=124 reszty 1
124:2=62 reszty 0
62:2=31 reszty 0
31:2=15 reszty 1
15:2=7 reszty 1
7:2=3 reszty 1
3:2=1 reszty 1
1:2=0 reszty 1

Odczytujemy reszty od końca do początku podziału: **1 1 1 1 1 0 0 1 1 0 0** i jest to właśnie liczba dwójkowa.

Liczbę dwójkową 11111001100 zamieńmy z powrotem na dziesiętną:

Tabela C.1. Zamiana liczby dwójkowej na dziesiętną

1	1	1	1	1	0	0	1	1	0	0
2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
=1024	=512	=256	=128	=64	=32	=16	=8	=4	=2	=1
1*1024	+ 1*512	+1*256	+1*128	+ 1*64	+ 0*32	+ 0*16	+ 1* 8	+ 1*4	+ 0*2	+ 0*1=
=1996										

C.2. Zamiana liczby dziesiętnej na szesnastkową i odwrotnie

W systemie szesnastkowym mamy szesnaście „cyfr":

0 1 2 3 4 5 6 7 8 9 A B C D E F, gdzie A=10 (dziesięć), B=11 (jedenaście),
C=12 (dwanaście), D=13 (trzynaście), E=14 (czternaście),
F=15 (piętnaście).

Przedstawmy liczbę dziesiętną **1996** w postaci szesnastkowej:

1996:16 = 124 reszty 12
124:16 = 7 reszty 12
7:16 = 0 reszty 7

Reszty z tego podziału odczytujemy od końca, zamieniając je na cyfry szesnastkowe: 7CC = 1996.

Liczbę szesnastkową 7CC zamienimy z powrotem na dziesiętną:

Tabela C.2. Zamiana liczby szesnastkowej na dziesiętną

7	C←12	C←12
16^2	16^1	16^0
=256	=16	=1
7*256	+ 12 * 16	+12 * 1 =
1996		

C.3. Dodawanie liczb dwójkowych i szesnastkowych

```
  10 10 10 11
   2  2  2  3   10      1  10
   1  1  1  1   2       1   2
   1  1  1  1  1  0  0  1  1  0  0  = 1996D
+        1  1  1  0  1  0  1  1  =  235D
-----------------------------------------
   1  0  0  0  1  0  1  1  0  1  1  1 = 2231D
```

Rysunek C.1. Dodawanie binarne

```
 8D=8H      27D=1BH                23D=17H
 7+1+=8D    C+1+E=12+1+14=27D      B+C=11+12=23D
   1            1
   7          C          C      = 1996D
+             E          B      =  235D
-----------------------------------------
   8          B          7      = 2231D
```

Rysunek C.2. Dodawanie szesnastkowe

Dodatek D. Lista rozkazów procesora Pentium – opis ogólny

1. Rozkazy przesłania danych

MOV – prześlij (kopiuj)
CMOVE/CMOVZ – prześlij warunkowo, jeśli równe/jeśli zero
CMOVNE/CMOVNZ – prześlij warunkowo, jeśli nie równe/jeśli nie zero
CMOVA/CMOVNBE – prześlij warunkowo, jeśli powyżej/jeśli nie poniżej lub równe
CMOVAE/CMOVNB – prześlij warunkowo, jeśli powyżej lub równe/jeśli nie poniżej
CMOVB/CMOVNAE – prześlij warunkowo, jeśli poniżej/jeśli nie powyżej lub równe
CMOVBE/CMOVNA – prześlij warunkowo, jeśli poniżej lub równe/jeśli nie powyżej
CMOVG/CMOVNLE – prześlij warunkowo, jeśli większe/jeśli nie mniejsze lub równe
CMOVGE/CMOVNL – prześlij warunkowo, jeśli większe lub równe/jeśli nie mniejsze
CMOVL/CMOVNGE – prześlij warunkowo, jeśli mniejsze/jeśli nie mniejsze lub równe
CMOVLE/CMOVNG – prześlij warunkowo, jeśli mniejsze lub równe/jeśli nie większe
CMOVC – prześlij warunkowo, jeśli przeniesienie
CMOVNC – prześlij warunkowo, jeśli brak przeniesienia
CMOVO – prześlij warunkowo, jeśli nadmiar
CMOVNO – prześlij warunkowo, jeśli brak nadmiaru
CMOVS – prześlij warunkowo, jeśli znak (ujemny)
CMOVNS – prześlij warunkowo, jeśli brak znaku (nie ujemny)
CMOVP/CMOVPE – prześlij warunkowo, jeśli parzyste/jeśli parzystość parzysta
CMOVNP/CMOVPO – prześlij warunkowo, jeśli nieparzyste/jeśli parzystość nieparzysta
XCHG – wymiana
BSWAP – odwrócenie kolejności bajtów
XADD – zamiana operandów i dodawanie
CMPXCHG – porównywanie i zamiana
CMPXCHG8B – porównywanie i zamiana 8 bajtów
PUSH – odłóż na stos
POP – zdejmij ze stosu
PUSHA/PUSHAD – umieszczenie na stosie wszystkich rejestrów ogólnego zastosowania
POPA/POPAD – zdejmowanie ze stosu wszystkich rejestrów ogólnego przeznaczenia
IN – odczyt z portu wejściowego
OUT – zapis do portu wejściowego
CWD/CDQ – konwersja słowa na dwusłowo/konwersja dwusłowa na poczwórne słowo
CBW/CWDE – konwersja bajtu na słowo/konwersja słowa na podwójne słowo

MOVSX – przesłanie danych do rejestru o większej długości z zachowaniem znaku
MOVZX – przesłanie dłuższych danych z uzupełnieniem zerami

2. Rozkazy arytmetyki binarnej

ADD – dodawanie całkowite
ADC – dodawanie całkowite z przeniesieniem
SUB – odejmowanie
SBB – odejmowanie całkowite z pożyczką
IMUL – mnożenie ze znakiem
MUL – mnożenie bez znaku
IDIV – dzielenie ze znakiem
DIV – dzielenie bez znaku
INC – zwiększenie o 1 (inkrementacja)
DEC – zmniejszanie o 1 (dekrementacja)
NEG – negacja
CMP – porównywanie

3. Rozkazy arytmetyki dziesiętnej

DAA – korekcja upakowanego kodu BCD po dodawaniu
DAS – korekcja upakowanego kodu BCD po odejmowaniu
AAA – korekcja po dodawaniu
AAS – korekcja po odejmowaniu
AAM – korekcja po mnożeniu
AAD – korekcja przed dzieleniem

4. Rozkazy logiczne

AND – iloczyn logiczny
OR – suma logiczna
XOR – alternatywa wykluczająca (logiczna nierównoważność)
NOT – negacja logiczna

5. Rozkazy przesunięć logicznych i arytmetycznych

SAR – przesunięcie arytmetyczne w prawo
SHR – przesunięcie logiczne w lewo
SAL/SHL – przesunięcie arytmetyczne w lewo/przesunięcie logiczne w lewo
SHRD – podwójne przesunięcie w prawo
SHLD – podwójne przesunięcie w lewo
ROR – obrót w prawo
ROL – obrót w lewo
RCR – obrót w prawo z wykorzystaniem flagi przeniesienia, CF
RCL – obrót w lewo z wykorzystaniem flagi przeniesienia, CF

6. Rozkázy operujące na pojedynczych bitach i bajtach

BT – testowanie bitu

BTS – testowanie bitu z ustawieniem
BTR – testowanie bitu z zerowaniem
BTC – testowanie bitu z negacją
BSF – przeszukiwanie bitów w przód
BSR – przeszukiwanie bitów wstecz
SETE/SETZ – ustaw bajt, jeśli równe/jeśli zero
SETNE/SETNZ – ustaw bajt, jeśli nie równe/jeśli nie zero
SETA/SETNBE – ustaw bajt, jeśli powyżej/jeśli nie powyżej lub równe
SETAE/SETNB/SETNC – ustaw bajt, jeśli powyżej lub równe/jeśli nie poniżej/jeśli brak przeniesienia
SETB/SETNAE/SETC – ustaw bajt, jeśli poniżej/jeśli nie powyżej lub równe/jeśli przeniesienie
SETBE/SETNA – ustaw bajt, jeśli poniżej lub równe/jeśli nie powyżej
SETG/SETNLE – ustaw bajt, jeśli większe/jeśli nie mniejsze lub równe
SETGE/SETNL – ustaw bajt, jeśli większe lub równe/jeśli nie mniejsze
SETL/SETNGE – ustaw bajt, jeśli mniejsze/jeśli nie większe lub równe
SETLE/SETNG – ustaw bajt, jeśli mniejsze lub równe/jeśli nie większe
SETS – ustaw bajt, jeśli znak (ujemny)
SETNS – ustaw bajt, jeśli brak znaku (nie ujemny)
SETO – ustaw bajt, jeśli powyżej
SETNO – ustaw bajt, jeśli nie powyżej
SETPE/SETP – ustaw bajt, jeśli parzystość parzysta/jeśli parzysta
SETPO/SETNP – ustaw bajt, jeśli parzystość nieparzysta/jeśli nieparzysta
TEST – porównywanie logiczne

7. Rozkazy sterujące przesyłaniem danych

JMP – skok bezwarunkowy
JE/JZ – skok (warunkowy), jeśli równe/jeśli zero
JNE/JNZ – skok, jeśli nie równe/jeśli nie zero
JA/JNBE – skok, jeśli powyżej/jeśli nie poniżej lub równe
JAE/JNB – skok, jeśli powyżej lub równe/jeśli nie poniżej
JB/JNAE – skok, jeśli poniżej/jeśli nie powyżej lub równe
JBE/JNA – skok, jeśli poniżej lub równe/jeśli nie powyżej
JG/JNLE – skok, jeśli większe/jeśli nie mniejsze lub równe
JGE/JNL – skok, jeśli większe lub równe/jeśli nie mniejsze
JL/JNGE – skok, jeśli mniejsze/jeśli większe lub równe
JLE/JNG – skok, jeśli mniejsze lub równe/jeśli nie większe
JC – skok, jeśli przeniesienie, CF
JNC – skok, jeśli brak przeniesienia
JO – skok, jeśli nadmiar
JNO – skok, jeśli brak nadmiaru
JS – skok, jeśli znak (ujemny)
JNS – skok, jeśli brak znaku (nie ujemny)

JPO/JNP – skok, jeśli parzystość nieparzysta/jeśli brak parzystości
JPE/JP – skok, jeśli parzystość parzysta/jeśli parzysta
JCXZ/JECXZ – skok, jeśli rejestr CX zero/jeśli rejestr ECX zero
LOOP – skok z licznikiem w rejestrze ECX
LOOPZ/LOOPE – skok z licznikiem w rejestrze ECX i zero/z licznikiem w rejestrze ECX i równe
LOOPNZ/LOOPNE – skok z licznikiem w rejestrze ECX i nie zero/z licznikiem w rejestrze ECX i nie równe
CALL – wywołaj procedurę
RET – powrót z procedury
IRET – powrót z przerwania
INT – przerwanie programowe
INTO – przerwanie programowe przy nadmiarze
BOUND – wykrywanie wartości poza zakresem
ENTER – wejście do procedury wyższego poziomu
LEAVE – wyjście z procedury wyższego poziomu

8. Rozkazy łańcuchowe

MOVS/MOVSB – prześlij łańcuch/bajt łańcucha
MOVS/MOVSW – prześlij łańcuch/słowo łańcucha
MOVS/MOVSD – prześlij łańcuch/podwójne słowo łańcucha
CMPS/CMPSB – porównaj łańcuch/bajt łańcucha
CMPS/CMPSW – porównaj łańcuch/słowo łańcucha
CMPS/CMPSD – porównaj łańcuch/podwójne słowo łańcucha
SCAS/SCASB – przeszukaj łańcuch/bajt łańcucha
SCAS/SCASW – przeszukaj łańcuch/słowo łańcucha
SCAS/SCASD – przeszukaj łańcuch/podwójne słowo łańcucha
LODS/LODSB – ładuj łańcuch/bajt łańcucha
LODS/LODSW – ładuj łańcuch/słowo łańcucha
LODS/LODSD – ładuj łańcuch/podwójne słowo łańcucha
STOS/STOSB – umieść w pamięci łańcuch/bajt łańcucha
STOS/STOSW – umieść w pamięci łańcuch/słowo łańcucha
STOS/STOSD – umieść w pamięci łańcuch/podwójne słowo łańcucha
REP – powtórz, dopóki ECX nie jest zerem
REPE/REPZ – powtórz, dopóki równe/dopóki zero
REPNE/REPNZ – powtórz, dopóki nie równe/dopóki nie zero
INS/INSB – wprowadź łańcuch z portu/bajt łańcucha z portu
INS/INSW – wprowadź łańcuch z portu/słowo łańcucha z portu
INS/INSD – wprowadź łańcuch z portu/podwójne słowo łańcucha z portu
OUTS/OUTSB – wyprowadź łańcuch do portu/bajt łańcucha do portu
OUTS/OUTSW – wyprowadź łańcuch do portu/słowo łańcucha do portu
OUTS/OUTSD – wyprowadź łańcuch do portu/podwójne słowo łańcucha do portu

9. Rozkazy sterujące flagami

STC – ustaw flagę przeniesienia, CF
CLC – wyczyść (wyzeruj) flagę przeniesienia, CF
CMC – neguj flagę przeniesienia, CF
CLD – zeruj flagę kierunku, DF
STD – ustaw flagę kierunku, DF
LAHF – ładuj flagi do rejestru AH
SAHF – zapamiętaj rejestr AH w rejestrze FLAGS
PUSHF/PUSHFD – odłóż rejestr EFLAGS na stos
POPF/POPFD – zdejmij ze stosu rejestr EFLAGS
STI – ustaw flagę przerwania, IF
CLI – wyczyść flagę przerwania, IF

10. Rozkazy rejestrów segmentowych

LDS – ładowanie dalekiego wskaźnika, używając DS
LES – ładowanie dalekiego wskaźnika, używając ES
LFS – ładowanie dalekiego wskaźnika, używając FS
LGS – ładowanie dalekiego wskaźnika, używając GS
LSS – ładowanie dalekiego wskaźnika, używając SS

11. Rozkazy mieszane

LEA – ładuj adres efektywny
NOP – brak operacji, „nic nie rób"
UB2 – niezdefiniowany rozkaz
XLAT/XLATB – tłumaczenie na podstawie tablicy translacji
CPUID – identyfikacja procesora, CPU

12. Rozkazy systemowe

LGDT – ładuj rejestr GDTR
SGDT – umieść w pamięci rejestr GDTR
LLDT – ładuj rejestr LDTR
SLDT – umieść w pamięci rejestr LDTR
LTR – ładuj rejestr zadania
STR – umieść w pamięci rejestr zadania
LIDT – ładuj rejestr IDTR
SIDT – umieść w pamięci rejestr IDTR
MOV – ładuj i zapamiętaj rejestry sterujące
LMSW – ładuj rejestr słowa stanu maszyny
SMSW – zapamiętaj słowo stanu maszyny
CLTS – zeruj flagę przełączania zadań
ARPL – zmiana pola RPL selektora
LAR – ładuj bajt praw dostępu
LSL – ładuj granicę segmentu

VERR – weryfikuj segment do odczytu
VERW – weryfikuj segment do zapisu
MOV – ładuj i zapamiętaj rejestry uruchomieniowe
INVD – opróżnij wewnętrzną pamięć podręczną (brak opóźnionego zapisu)
WBINVD – opróżnij pamięć podręczną, z opóźnionym zapisem
INVLPG – opróżnij pamięć TLB
LOCK (przedrostek) – uaktywnienie sygnału LOCK
HLT – zatrzymanie procesora
RSM – powrót z trybu zarządzania systemem
RDMSR – odczyt ze specjalnego rejestru (MSR)
WRMSR – zapis do specjalnego rejestru (MSR)
RDPMC – odczyt licznika kontroli wydajności (systemu)
RDTSC – odczyt licznika etykiety czasowej

D.1. Rozkazy zaimplementowane w technologii MMX™

1. Rozkazy przesłania danych

MOVD – prześlij podwójne słowo
MOVQ – prześlij poczwórne słowo

2. Rozkazy konwersji

PACKSSWB – pakowanie słowa na bajty z nasyceniem, ze znakiem
PACKSSDW – pakowanie podwójnego słowa na słowa z nasyceniem, ze znakiem
PACKUSWB – pakowanie słowa na bajty z nasyceniem, bez znaku
PUNPCKHBW – rozpakowywanie „starszego" bajtu na słowo
PUNPCKHWD – rozpakowywanie „starszego" słowa na podwójne słowo
PUNPCKHDQ – rozpakowywanie „starszego" podwójnego słowa na poczwórne słowo
PUNPCKLBW – rozpakowywanie „młodszego" bajtu na słowo
PUNPCKLWD – rozpakowywanie „młodszego" słowa na podwójne słowo
PUNPCKLDQ – rozpakowywanie „młodszego" podwójnego słowa na poczwórne słowo

3. Rozkazy arytmetyczne (danych spakowanych)

PADDB – dodaj spakowane bajty
PADDW – dodaj spakowane słowa
PADDD – dodaj spakowane dwusłowa
PADDSB – dodaj spakowane bajty z nasyceniem
PADDSW – dodaj spakowane słowa z nasyceniem
PADDUSB – dodaj spakowane bez znaku bajty z nasyceniem
PADDUSW – dodaj spakowane bez znaku słowa z nasyceniem
PSUBB – odejmij spakowane bajty
PSUBW – odejmij spakowane słowa
PSUBD – odejmij spakowane dwusłowa
PSUBSB – odejmij spakowane bajty z nasyceniem

PSUBSD – odejmij spakowane słowa z nasyceniem
PMULHW – mnóż spakowane słowa i zapamiętaj „wyższe" słowo
PMULW – mnóż spakowane słowa i zapamiętaj „niższe" słowo
PMADDWD – mnóż i dodawaj spakowane słowa

4. Rozkazy porównywania

PCMPEQB – porównuj spakowane bajty, jeśli równe: wynik = 0xff, jeśli nie: wynik = 0
PCMPEQW – porównuj spakowane słowa, jeśli równe: wynik = 0xff, jeśli nie: wynik = 0
PCMPEQD – porównuj spakowane dwusłowa, jeśli równe: wynik = 0xff, jeśli nie: wynik = 0
PCMPGTB – porównuj spakowane bajty, jeśli większe: wynik = 0xff, jeśli nie: wynik = 0
PCMPGTW – porównuj spakowane słowa, jeśli większe: wynik = 0xff, jeśli nie: wynik = 0
PCMPGTD – porównuj spakowane dwusłowa, jeśli większe: wynik = 0xff, jeśli nie: wynik = 0

5. Rozkazy logiczne

PAND – bitowe logiczne AND
PANDN – bitowe logiczne NAND
POR – bitowe logiczne OR
PXOR – bitowe logiczne XOR

6. Rozkazy przesunięć logicznych i arytmetycznych

PSLLW – przesunięcie logiczne w lewo spakowanego słowa
PSLLD – przesunięcie logiczne w lewo spakowanego dwusłowa
PSLLQ – przesunięcie logiczne w lewo spakowanego poczwórnego słowa
PSRLW – przesunięcie logiczne w prawo spakowanego słowa
PSRLD – przesunięcie logiczne w prawo spakowanego dwusłowa
PSRLQ – przesunięcie logiczne w prawo spakowanego poczwórnego słowa
PSRAW – przesunięcie arytmetyczne w prawo spakowanego słowa
PSRAD – przesunięcie arytmetyczne w prawo spakowanego dwusłowa

7. Rozkaz stanu zarządzania

EMMS – opuszczenie stanu MMX

D.2. Rozkazy koprocesora, FPU

1. Rozkazy przesłania danych

FLD – ładuj liczbę rzeczywistą
FST – zachowaj liczbę rzeczywistą
FSTP – zachowaj liczbę rzeczywistą i zdejmij ze stosu
FILD – ładuj liczbę całkowitą
FIST – zachowaj liczbę całkowitą
FISTP – zachowaj liczbę całkowitą i zdejmij ze stosu
FBLD – ładuj liczbę w kodzie BCD
FBSTP – ładuj liczbę w kodzie BCD i zdejmij ze stosu

FXCH – zamień rejestry
FCMOVE – prześlij warunkowo liczbę zmiennoprzecinkową, jeśli równe
FCMOVNE – prześlij warunkowo liczbę zmiennoprzecinkową, jeśli nie równe
FCMOVB – prześlij warunkowo liczbę zmiennoprzecinkową, jeśli poniżej
FCMOVBE – prześlij warunkowo liczbę zmiennoprzecinkową, jeśli poniżej lub równe
FCMOVNB – prześlij warunkowo liczbę zmiennoprzecinkową, jeśli nie poniżej
FCMOVNBE – prześlij warunkowo liczbę zmiennoprzecinkową, jeśli nie poniżej lub równe
FCMOVU – prześlij warunkowo liczbę zmiennoprzecinkową, jeśli nieuporządkowane
FCMOVNU – prześlij warunkowo liczbę zmiennoprzecinkową, jeśli nie nieuporządkowane

2. Rozkazy arytmetyczne

FADD – dodaj liczbę rzeczywistą
FADDP – dodaj liczbę rzeczywistą i zdejmij ze stosu
FIADD – dodaj liczbę całkowitą
FSUB – odejmij liczbę rzeczywistą
FSUBP – odejmij liczbę rzeczywistą i zdejmij ze stosu
FISUBP – odejmij liczbę całkowitą
FSUBR – odejmij liczbę całkowitą *(odejmowanie odwrócone)*
FSUBRP – odejmij liczbę całkowitą i zdejmij ze stosu *(odejmowanie odwrócone)*
FISUBR – odejmij liczbę całkowitą i zdejmij ze stosu *(odejmowanie odwrócone)*
FMUL – pomnóż liczbę rzeczywistą
FMULP – pomnóż liczbę rzeczywistą i zdejmij ze stosu
FIMUL – pomnóż liczbę całkowitą
FDIV – podziel liczbę rzeczywistą
FDIVP – podziel liczbę rzeczywistą i zdejmij ze stosu
FIDIV – podziel liczbę całkowitą
FDIVR – podziel liczbę rzeczywistą *(dzielenie odwrócone)*
FDIVRP – podziel liczbę rzeczywistą i zdejmij ze stosu *(dzielenie odwrócone)*
FIDIVR – podziel liczbę całkowitą *(dzielenie odwrócone)*
FPREM – oblicz resztę z dzielenia
FPREM1 – oblicz resztę z dzielenia zgodnie z normą IEEE
FABS – oblicz wartość absolutną
FCHS – zmień znak
FRNDINT – zaokrąglij do liczby całkowitej
FSCALE – skalowanie przez liczbę będącą potęgą liczby 2
FSQRT – pierwiastek kwadratowy
FXTRACT – obliczanie mantysy i operandu

3. Rozkazy porównywania

FCOM – porównaj liczby rzeczywiste
FCOMP – porównaj liczby rzeczywiste i zdejmij ze stosu
FCOMPP – porównaj liczby rzeczywiste i dwukrotnie zdejmij ze stosu
FUCOM – nieuporządkowane porównywanie liczb rzeczywistych

FUCOMP – nieuporządkowane porównywanie liczb rzeczywistych i zdjęcie ze stosu
FUCOMPP – nieuporządkowane porównywanie liczb rzeczywistych i dwukrotne zdjęcie ze stosu
FICOM – porównywanie liczb całkowitych
FICOMP – porównywanie liczb całkowitych i zdjęcie ze stosu
FCOMI – porównywanie liczb rzeczywistych i ustawienie EFLAGS
FUCOMI – nieuporządkowane porównywanie liczb rzeczywistych i ustawienie EFLAGS
FCOMIP – porównywanie rzeczywiste, ustawienie EFLAGS, i zdjęcie ze stosu
FUCOMIP – nieuporządkowane porównywanie rzeczywiste, ustawienie EFLAGS, i zdjęcie ze stosu
FTST – test porównywania na liczbach rzeczywistych
FXAM – badanie liczby rzeczywistej

4. Rozkazy funkcji przestępnych

FSIN – sinus
FCOS – cosinus
FSINCOS – sinus i cosinus
FPTAN – tangens
FPATAN – arcus tangens
F2XM1 – 2^x-1
FYL2X – $y \times \log_2 x$
FYL2XP1 – $y \times \log_2 (x+1)$

5. Rozkazy ładowania stałych

FLD1 – ładuj +1.0
FLDZ – ładuj +0.0
FLDPI – FLDPI – FLDL2E – ładuj $\log_2 e$
FLDLN2 – ładuj $\log_e 2$
FLDL2T – ładuj $\log_2 10$
FLDLG2 – ładuj $\log_{10} 2$

6. Rozkazy sterowania koprocesorem

FINCSTP – zwiększenie (inkrementacja) wskaźnika wierzchołka stosu
FDECSTP – zmnicjszenie (dekrementacja) wskaźnika wierzchołka stosu
FFREE – zwolnienie rejestru zmiennoprzecinkowego
FINIT/FNINIT – inicjalizacja jednostki zmiennoprzecinkowej
FCLEX/FNCLEX – zerowanie znaczników błędów
FSTCW/FNSTCW – zapis rejestru sterowania koprocesora, FPU jednostki zmienno-przecinkowej
FLDCW – ładowanie słowa sterującego
FSTENV/FNSTENV – zapis środowiska jednostki FPU

FLDENV – ładowanie środowiska jednostki FPU
FSAVE/FNSAVE – zapis zawartości jednostki FPU
FRSTOR – odtworzenie stanu FPU
FSTSW/FNSTSW – zapis rejestru stanu jednostki FPU
WAIT/FWAIT – stan czekania jednostki FPU
FNOP – „nic nie rób" jednostki FPU

Dodatek E. Mały słownik asemblerowy

Adres – jednoznaczny identyfikator komórki pamięci. Programista pisząc programy w języku Asembler używa tylko adresu logicznego, SEGMENT:OFFSET. Na podstawie adresu logicznego, zapisanego wewnątrz programu, adres przeliczany jest na adres fizyczny rozkazu w pamięci w następujący sposób: adres fizyczny rozkazu = zawartość rejestru segmentowego **CS**×10H (SEGMENT) + zawartość rejestru **IP** (OFFSET), np. **CS** = C018H, **IP** = 0129H, adres fizyczny = C018H × 10H+0129H = C0180+ 0129H= C02A9H= 787113D.

Asembler – program tłumaczący (translator) program źródłowy zapisany w języku Asembler na program wynikowy w języku wewnętrznym. Asembler *(drugie znaczenie pisane dużą literą)* – język programowania niskiego poziomu, język symboliczny, którego budulcem są nazwy symboliczne określające rozkazy procesora, instrukcje języka, adresy komórek pamięci, stałe, parametry i inne elementy używane w programie (źródłowym). Programy napisane w języku Asembler muszą być przed załadowaniem i wykonaniem w pamięci przetłumaczone na język wewnętrzny.

Bajt – ciąg bitów o długości 8 bitów.

BIOS-u obszar roboczy – 256-bajtowy obszar roboczy BIOS-u rozciągający się od adresu 00400H-004FFH.

Bit – pojedyncza cyfra w zapisie dwójkowym.

CMOS-u obszar – 64-bajtowy obszar pamięci zawierający dane konfiguracyjne komputera. Dostęp do obszaru CMOS możliwy jest poprzez funkcje systemu (operacyjnego) lub bezpośrednio poprzez porty: port o adresie 71H – dwukierunkowy rejestr danych pamięci CMOS i port 70H – rejestr adresowy pamięci CMOS.

Deasembler – program-translator działający odwrotnie do asemblera, a mianowicie program tłumaczący kody języka wewnętrznego na symboliczne nazwy w języku Asembler.

DOS/BASIC obszar roboczy – 512-bajtowy obszar roboczy przeznaczony dla DOS/BASIC, rozciągający się od adresu 00500H-006FFH.

ICA – (ang. *intra-application communications area*) obszar komunikacji do wewnętrznych zastosowań. 16-bajtowy obszar pamięci operacyjnej położony między adresami od 004F0 do 004FF. Używany do przechowywania danych, które mogą być wspólne dla kilku różnych programów.

Instrukcja asemblera – instrukcja języka Asembler zawierająca następujące pola: [etykieta[:]] mnemonik [argument (operand)] [;komentarz]. Pola ujęte w nawiasy [] są opcjonalne, pole mnemonik występuje zawsze. Pole argumentu (operandu) występuje wówczas, gdy wymaga tego dany rozkaz procesora posiadający zawsze swoją nazwę mnemoniczną. Gdy w linii instrukcji wystąpi komentarz, to musi być poprzedzony znakiem średnika ';'. Opcjonalne pole etykiety przypisuje nazwę instrukcji asemblera.

Język wewnętrzny – język składający się z kodów dwójkowych.

Lista rozkazów – zbiór wszystkich rozkazów, które mogą być wykonywane przez dany procesor.

Mnemonik – symboliczna nazwa kodu rozkazu, np. MOV...

Model pamięci – pojęcie definiujące największy rozmiar dla programów zawierających segmenty kodu i segmenty danych. Zdefiniowano sześć modeli pamięci: tiny (model zdefiniowany dla programów postaci COM – kod programu i kod danych musi zawierać się w tym samym 64 KB segmencie), small (kod programu musi zmieścić się w pojedynczym 64 KB segmencie, kod danych musi zawierać się wewnątrz oddzielnego segmentu; kod programu i kod danych są typu bliskiego), medium (kod programu może być większy niż 64 KB, ale kod danych musi zmieścić się w obrębie pojedynczego 64 KB segmentu; kod programu jest typu dalekiego a kod danych są typu bliskiego), compact (kod programu musi zmieścić się w pojedynczym 64 KB segmencie, ale kod danych może być większy niż 64 KB; kod programu jest typu bliskiego, kod danych typu dalekiego), large (kody programu i kody danych mogą być większe niż 64 KB, ale pojedynczy obszar danych nie może być większy niż 64 KB; kod programu i kod danych są typu dalekiego), huge (kody programu i kody danych mogą być większe niż 64 KB; kod programu i kod danych są typu dalekiego. Wskaźniki do elementów wewnątrz obszaru są typu dalekiego). Opisywane modele pamięci odpowiadają modelom pamięci używanym przede wszystkim przez języki wysokiego poziomu (Turbo C, Turbo Pascal). W programie asemblerowym poleceniem wyznaczającym właściwy model pamięci jest dyrektywa .model (z kropką na początku nazwy).

Nanosekunda – jedna miliardowa część sekundy, 10^{-9} sek.

Offset – w języku Asembler wielkość zawarta w rejestrze **IP** (wskaźnik rozkazów), określająca adres względem początku segmentu programu.

Pamięć operacyjna – pamięć będąca *areną* dla wykonywanych na niej programów.

Pikosekunda – jedna bilionowa część sekundy, 10^{-12} sek.

Port – miejsce w wyróżnionej przestrzeni adresów wejścia/wyjścia, identyfikowane przez swój adres będący liczbą od 0 do 65535. Każdy port pozwala wysłać/pobrać jeden bajt lub słowo (dwa bajty) do lub z rejestru. W języku Asembler opcracje na portach wykonuje się tylko za pomocą rozkazów: **IN** ... (pobranie da-

nych z portu) i **OUT** ... (wysłanie danych do portu). Adres portu podaje się jako liczbę, gdy jego wartość nie przekracza 255, bądź w rejestrze **DX**.

Program typu COM – (ang. *command* – rozkaz); programy typu COM nazywane są częstą mapą pamięci – postać programu na dysku jest wierną kopią tego, co jest ładowane i uruchamiane w pamięci. Podczas ładowania do pamięci programu typu COM wszystkie rejestry segmentowe (**CS**, **DS**, **ES**, **SS**) otrzymują tę samą wartość, wartość segmentu bloku PSP. Zawartość rejestru **IP** wynosi zawsze 100H; sterowanie jest przekazywane tuż za 256-bajtowy blok PSP. Ze względu na to pierwsza instrukcja w programie musi rozpoczynać się od adresu **CS**:0100H; w programie asemblerowym odpowiednio umieszcza się dyrektywę ORG 100H (ORG 256D). Programy typu COM są krótsze niż programy typu EXE, nie mogą być dłuższe niż 64 KB (to jest ich wadą, ale i zaletą), są szybciej ładowane do pamięci. Ze względu na to, że nie wymagają stosu, rejestru **DS** i **ES**, łatwiej jest je napisać niż programy EXE. Wady programów COM: wymagają bardzo ścisłych (chociaż prostszych niż programy EXE) metod kodowania; mogą mieć tylko jeden segment, przez co brak jest wyraźnego rozdziału między danymi a instrukcjami; nie mogą uzyskać dostępu do procedur lub danych zawartych w innym segmencie. Konstrukcja plików COM utrudnia korzystanie z modułów zewnętrznych, w których nazwy segmentów są akurat inne od nazw użytych w segmencie *macierzystym*.

Program typu EXE – (ang. *execution* – wykonanie); program o charakterze relokowalnym, przesuwalnym. Programy typu EXE mogą być ładowane i uruchamiane w pamięci, począwszy od dowolnego adresu będącego wielokrotnością liczby 16D(10H). Ich budowa może być wielosegmentowa. Segmenty (do 64 KB) mają zapewnioną wzajemną komunikację za pomocą 32-bitowych adresów logicznych. Zawartość programu EXE składa się z nagłówka programu i z modułu (programu) przeznaczonego do umieszczenia go w pamięci. W nagłówku wyróżnia się część sformatowaną o ustalonej długości i zawartości poszczególnych pól oraz tablicę relokacji. W części sformatowanej nagłówka, zaczynającej się znacznikiem ze znakami 'MZ', zawarty jest opis pliku oraz wszelkie wymagania dotyczące przydziału pamięci operacyjnej, informacja o początkowej zawartości rejestrów. Wielkość programu EXE jest ograniczona tylko wielkością pamięci dyskowej.

Program wynikowy – program powstały po translacji programu źródłowego, mający nieczytelną postać dla programisty, zawierający kody zapisane w języku wewnętrznym.

Program źródłowy – program, którego elementami są czytelne dla programisty nazwy instrukcji, rozkazów, poleceń, należące do danego języka programowania.

Programy rezydentne – specjalnie skonstruowane programy pozostawiające swój kod w pamięci (programy TSR – ang. *terminate but stay resident*). Kod programu rezydentnego chroniony jest w pamięci na równi z kodem systemu operacyjnego.

Programy uruchomieniowe (ang. *debbuggers*) – programy służące do uruchamiania programów, modyfikowania, testowania, wykonywania ich w całości lub w określonej części; najbardziej znanym programem uruchomieniowym jest program DEBUG.EXE.

PSP – 256-bajtowy (100H) blok wstępny programu, początek tego obszaru określany jest jako początek segmentu programu. Blok wstępny programu, PSP, stanowi obszar komunikacyjny między programem a systemem.

Przechowywanie odwrotne – sposób przechowywania informacji w pamięci (stosowany w pamięciach komputerów osobistych) polegający na tym, iż dwubajtowe słowa zapisywane są w odwrotny sposób niż przedstawia się je w normalnym zapisie. Na przykład liczba szesnastkowa F07B zapisana zostanie w pamięci, w postaci kolejno występujących bajtów 7B oraz F0.

Przerwanie – sygnał informujący procesor o zajściu jakiegoś zdarzenia. Przerwanie zwykle wywołuje akcję programu; powstaje w wyniku zdarzenia, którego czas wystąpienia jest nieoczekiwany.

Rejestr – podstawowa komórka pamięci wewnątrz procesora funkcjonalnie będąca zespołem przerzutników.

Rozkaz – elementarna operacja, którą może wykonać procesor.

Segment – w języku Asembler wielkość zawarta w rejestrze segmentowym **CS**, określająca wraz z zawartością rejestru **IP** (offset) adres logiczny rozkazu zapisywany w programie jako para SEGMENT:OFFSET.

Stos – rodzaj pamięci tak zorganizowanej, iż dostęp do niej możliwy jest tylko z jednej strony. Informację można zapisywać tylko na wierzchołku stosu, odczytać również tylko z wierzchołka. Każdy zapis informacji na stosie powoduje zwiększenie jego zawartości, każdy odczyt zmniejszenie jego zawartości. Po odczycie odsłaniana jest informacja zapisana na pozycji niższej na stosie. Informacje odczytywane są ze stosu w kolejności odwrotnej niż były na nim zapisane; jako pierwsza będzie odczytana informacja zapisana ostatnio.

Tablica wektorów przerwań – tablica zawierająca adresy wektorów przerwań (przechowywanych w pamięci w sposób odwrotny, na przykład wektor przerwań 54FF00F0 należy odczytać jako F000:FF54, SEGMENT:OFFSET). Tablica wektorów przerwań umieszczona jest w najniższym, pierwszym kilobajcie pamięci od położenia 0000H do 03FFH.

Tryb adresowania – określanie miejsca, gdzie jest umieszczony adres argumentu (operandu) rozkazu lub sposób, w jaki jest on obliczany.

Wektor przerwania – wskaźnik do procedury obsługi przerwań. Para słów w pamięci zawierająca przesunięcie (offset) i adres segmentu programu, który ma być wykonany przez procesor, gdy otrzyma on odpowiedni sygnał generowany instrukcją INT.

Skorowidz