

ASSEMBLER

WYKŁADY I ĆWICZENIA

Stanisław Kruk

WYDAWNICTWO NAUKOWE PWN

STANISŁAW KRUK

ASSEMBLER

WYKŁADY I ĆWICZENIA



WYDAWNICTWO NAUKOWE PWN
WARSZAWA 2009

[Kup książkę](#)

Projekt okładki: **MacGraf**
Redakcja: **Krystyna Knap**
Skład komputerowy: **Krzysztof Świstak**

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

Copyright © by Wydawnictwo Naukowe PWN SA
Warszawa 2009

ISBN: 978-83-01-15931-3

Wydawnictwo Naukowe PWN SA
02-676 Warszawa, ul. Postępu 18
tel. (0 22) 69 54 321
faks (0 22) 69 54 031
infolinia 0 801 33 33 88
e-mail: pwn@pwn.com.pl
www.pwn.pl

[Kup książkę](#)

Spis treści

Od Autora.....	8
1. Podstawy Asemblera.....	9
1.1. Natura języka Asembler.....	9
1.2. Architektura sprzętowa komputera	10
1.3. Powstawanie i rozwój języka Asembler	10
1.4. O użyteczności programów	11
1.5. Procesory, pamięć i jej adresowanie; przechowywanie odwrotne.....	11
1.6. Wejście/wyjście	14
1.7. Przerwania; wektory przerwai	15
2. Rejestry	17
2.1. Rejestry powszechnego zastosowania	18
2.2. Rejestry wskaźnikowe i indeksowe	20
2.3. Rejestry segmentowe	22
2.4. Wskaźnik rozkazów	25
2.5. Rejestr znaczników	25
3. Oprogramowanie systemowe DOS i BIOS	27
3.1. Funkcje BIOS	29
4. „Towarzysze” głównego procesora.....	31
5. Różne systemy liczenia.....	33
5.1. Dwójkowy system liczenia	34
5.2. Szesnastkowy system liczenia	35
5.3. Elementarne operacje logiczne NOT, OR, AND; prawa de Morgana.....	36
5.4. Liczby dwójkowe bez znaku i ze znakiem; uzupełnienie do dwóch; kod BCD	39
5.5. Proste działania na liczbach szesnastkowych	42
6. Program uruchomieniowy DEBUG.....	44
6.1. Polecenia.....	44
6.2. Proste programy pod DEBUG-iem.....	51
6.3. Zalety, wady i możliwości programu DEBUG.....	55
7. Podstawy konstruowania programów w języku Asembler	57
7.1. Pole etykiety	61
7.2. Pole operacji (pole mnemonika)	62
7.3. Pole argumentów (operandów)	62
7.4. Pole komentarza.....	62
7.5. Tryby adresowania.....	62
7.6. Schemat tworzenia programu asemblerowego	66

8. Tablica wektorów przerwań.....	72
9. Dwa przerwania najczęściej używane w programach asemblerowych.....	76
INT 10H	76
INT 21H	78
10. Działania arytmetyczne na liczbach dwójkowych oraz w kodzie BCD.....	82
10.1. Dodawanie (ADD, ADC)	82
10.2. Odejmowanie (SUB, SBB)	85
10.3. Mnożenie (MUL, IMUL).....	86
10.4. Dzielenie (DIV, IDIV; CBW, CWD)	87
10.5. Negacja lub uzupełnienie do dwóch (NEG)	91
10.6. Inkrementacja i dekrementacja (INC, DEC).....	92
10.7. Porównywanie (bajtów lub słów) przeznaczenia ze źródłem, CMP.....	92
11. Operacje logiczne	94
11.1. Negacja logiczna (bajtu lub słowa), NOT	94
11.2. Testowanie lub porównywanie logiczne (bajtu lub słowa), TEST	94
11.3. Mnożenie logiczne (bajtu lub słowa), AND	94
11.4. Dodawanie logiczne (bajtów lub słów), OR.....	95
11.5. Logiczna nierównoważność (bajtów lub słów), XOR	95
12. Rotacje i przesunięcia logiczne oraz arytmetyczne.....	96
12.1. Rotacja w lewo (bajtu lub słowa), ROL	96
12.2. Rotacja w prawo (bajtu lub słowa), ROR	97
12.3. Rotacja w lewo (bajtu lub słowa) z przeniesieniem, RCL.....	98
12.4. Rotacja w prawo (bajtu lub słowa) z przeniesieniem, RCR	98
12.5. Przesunięcie logiczne lub arytmetyczne (bajtu lub słowa) w lewo, SHL/SAL	99
12.6. Przesunięcie logiczne (bajtu lub słowa) w prawo, SHR	100
12.7. Przesunięcie arytmetyczne (bajtu lub słowa) w prawo, SAR.....	101
13. Przetwarzanie łańcuchów	103
13.1. Kopiowanie (bajtu lub słowa) z jednego miejsca pamięci do drugiego, MOVS, MOVSB, MOVSW	103
13.2. Kopiowanie zawartości rejestru AL lub AX do pamięci, STOS, STOSB, STOSW	105
13.3. Ładowanie (bajtu lub słowa) łańcucha z pamięci do AL lub AX, LODS, LODSB, LODSW	105
13.4. Porównywanie (bajtu lub słowa) dwóch łańcuchów CMPS, CMPSB, CMPSW; przedrostki REPE i REPNE	106
13.5. Porównywanie (bajtu lub słowa) łańcucha z zawartością rejestru akumulatora, AL lub AX, SCAS, SCASB, SCASW	108
14. Rozkazy sterujące.....	109
14.1. Rozkazy sterujące transmisją.....	109
14.1.1. Bezwarunkowe rozkazy skoku	109
14.1.2. Rozkazy skoków warunkowych	113
14.1.3. Rozkazy pętli programowych, LOOP, LOOPZ, LOOPNZ.....	115

14.2. Rozkazy sterujące procesorem: CLC, CLD, CLI, CMC, ESC, HLT, LOCK, NOP, STC, STD, STI, WAIT	116
15. Operacje na stosie i adresowanie	119
15.1. Rozkazy PUSH, POP	119
15.2. Rozkazy operujące na znacznikach PUSHF, POPF, LAHF, SAHF	119
15.3. Rozkazy wejścia-wyjścia: IN, OUT	120
15.4. Adresowanie pamięci, adres fizyczny, adres logiczny; przechowywanie odwrotne	120
15.5. Przedrostki (prefiksy) CS:, DS:, ES:, SS: i LOCK	121
15.6. Rozkazy operujące na adresach LEA, LES, LDS	123
16. Krótko o koprocesorze 8087 i jego programowaniu	124
16.1. Kilka dodatkowych uwag na temat programowania koprocesora	126
17. O liczbach w koprocesorze i nie tylko	127
18. Narzędzia programisty	132
18.1. Turbo Debugger dla DOS	132
18.2. Turbo Librarian, Bibliotekarz (TLIB.EXE) – dla systemu DOS i Windows	138
19. Nowa era procesorów	151
20. Typy danych	158
21. Technologia MMX	160
22. Streaming SIMD Extensions (SSE)	163
23. Technologia AVX (Advanced Vector Extensions)	167
24. Asembler w środowisku Windows	168
25. Wydruk na pulpit	181
25.1. Wprowadzenie do aplikacji okienkowych w systemie Windows	185
25.2. Wybrane funkcje API – spis alfabetyczny	197
26. Nowe narzędzia programistyczne; program FASM.W.EXE i OLLYDBG.EXE	199
Dodatek A. Kod ASCII	204
Dodatek B. Wpływ rozkazów na stan flag (znaczników) rejestru EFLAGS	213
Dodatek C. Lista rozkazów: Intel 64 i IA-32	217
C.1. Rozkazy powszechnego stosowania	217
C.2. x87 FPU i SIMD zachowania i odtworzenia stanu rejestru kontrolnego i statusu, MXCSR	221
C.3. Rozkazy zaimplementowane w technologii MMX™	223
C.4. Rozkazy SSE	225
C.5. Rozkazy SSE2	227
C.6. Rozkazy SSE3	230
C.7. Rozkazy SSSE3	231
C.8. Rozkazy SSE4	232
C.9. Rozkazy systemowe	234

C.10. Rozkazy trybu 32: podtryb 64	235
C.11. Virtual-Machine Extensions (VMX)	236
C.12. Safer Mode Extensions (SMX).....	236
C.13. Rozkazy Intel AVX, FMA i AES	237
Dodatek D. Wybrane rejestry.....	239
D.1. Rejestr flagowy (EFLAGS)	239
D.2. Rejestry sterujące: CR0, CR1, CR2, CR3, CR4	239
D.3. MXCSR – rejestr sterujący/statusu	242
D.4. Rejestr XFEATURE_ENABLED_MASK (XCR0 ;Extended Control Registers)	242
Dodatek E. Fizyczne podstawy komputera kwantowego.....	244
Mały słownik asemblerowy	247
Epilog	255
Ćwiczenia.....	256
Odpowiedzi do ćwiczeń	271
Indeks.....	298

Lux in tenebris
[Światło w ciemnościach]

[Kup książkę](#)

Od Autora

Język Asembler to najwspanialsza forma komunikacji programisty z maszyną cyfrową. To narzędzie, które pozwala stworzyć nawet najpotężniejszą budowlę świata komputerowego z podstawowych *mnemonicznych* cegiełek. W tego rodzaju budowaniu kryje się też jego precyzja, bowiem oczywistym, aczkolwiek nie zawsze uświadomionym jest fakt, iż tylko z małych elementów daje się skonstruować najbardziej finezyjną budowlę. **0** i **1** to zaledwie dwie cyfry, a dają przecież taką nieprzeliczalną wręcz wielość form i treści, *ucieleśnianych* w konkretnych programach komputerowych.

Niniejsza książka przeznaczona jest dla tych wszystkich Czytelników, którzy są zainteresowani przygodami ludzkiego umysłu, nienasyconego, ciągle zgłębiającego tajniki krzemowych logik.

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 jedyneką. 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 jedynekami.

Ale zacznijmy 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 zero-jedynkowego, 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 zero-jedynkowych 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 Czytelnika długimi „tasiemcami” zero-jedynkowymi 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 jedynekami. I cóż się stało. Ano, stało się! Programista przeoczył zero albo może jedynekę, po czym odbył się kosztowny spektakl na nieboskłonie. Czy ten programista dalej tam pracował? Nie wiadomo. W każdym razie rzecz potraktowano poważnie, wzięto pod uwagę omylność człowieka i sięgnięto po pomysł z początku lat 50. Pomysł był prosty; należało napisać program przekształcający monotonne ciągi zero-jedynkowe, 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 zero-jedynkowe rozkazów procesora na nazwy symboliczne, tzw. mnemoniki.

I tak np., zamiast wpisywać ciąg 10011011 wprowadzający procesor w stan oczekiwania, wpisywać będziemy słowo angielskie WAIT. Bo gdybyśmy się pomylili i ciąg 10011011 zamienili z ciągiem 11110100, spowodowałibyśmy spore zamieszanie, gdyż całkowicie zatrzymalibyśmy procesor. Nietrudno sobie wyobrazić, jakie niewyobrażalne 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 kontrolę sprawuje 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-asmemblerowca potrafi z dużym prawdopodobieństwem powiedzieć: – Teraz widzę (chyba) rozkaz, a teraz to dana tego (chyba) rozkazu. Jeśli do pamięci wczytany został „czysty jak łąza” 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 dezasemblacją) 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 kolejnym 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, stanowi to 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 jedyńki. 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 maksymalnie efektywnie kontrolować akcje procesora jedną za drugą. 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 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 działać na najnowszym maszynach, 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 żadanego miejsca, by pobrać przesyłkę do nadania lub tylko ją przekazać. Procesor, jako główny zarządcą, musi umieć komunikować się z pamięcią. Model 8086 zostaje połączony z pamięcią dwudziestoma „drutami”, po których biegą 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 $2^{20}=1\ 048\ 576$ możliwości. Wykorzystując wszystkie kombinacje ustawień 0 i 1 na 20 „drutach”, można jednoznacznie wskazać (zaadresować) 1 048 576 (1 MB – 1 megabajt) komórek pamięci.

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 jest już tam coś innego, np. kod systemu operacyjnego lub programy rezydentne, tak samo ważne jak kod systemu operacyjnego? A gdyby nawet programista wiedział, która komórka pamięci jest wolna i chciał tam przesłać daną, to jak ma tę komórkę zaadresować? Czy wolno mu wpisać bezwzględną wartość komórki w postaci: prześlij do 123 786 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, z których każdy zawiera po 65 536 (64 KB) jednobajtowych komórek. Adres dowolnej komórki pamięci składa się teraz z dwóch części – numeru segmentu i położenia komórki w tym segmencie. W programie asemblerowym zapiszemy to następująco: prześlij daną do segmentu nr 10, a w tym dziesiątym segmencie do komórki nr 32 456. Gdyby tak faktycznie było, to nadal nie rozwiązano by „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 Damoklesa. Procesor to niezwykle rygorysta. Ustawia w pamięci wszystko tak, jak być powinno. Wszystko ma pod 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 tylko rolę specyficznego skazańca, skazanego na taki

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, wtedy pełna wartość adresu fizycznego wynosić będzie, $10 \times A000H + 0140H = A0000H + 0140H = A0140H = 655680D$ (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 chociaż o jedną, wtedy byłaby możliwość zaadresowania więcej niż 1 MB pamięci o 64 KB-17 B, gdyż $10FFEFH(1114095) - 100000H(1048576) = 65519$ bajtów = 64 KB-17 B.

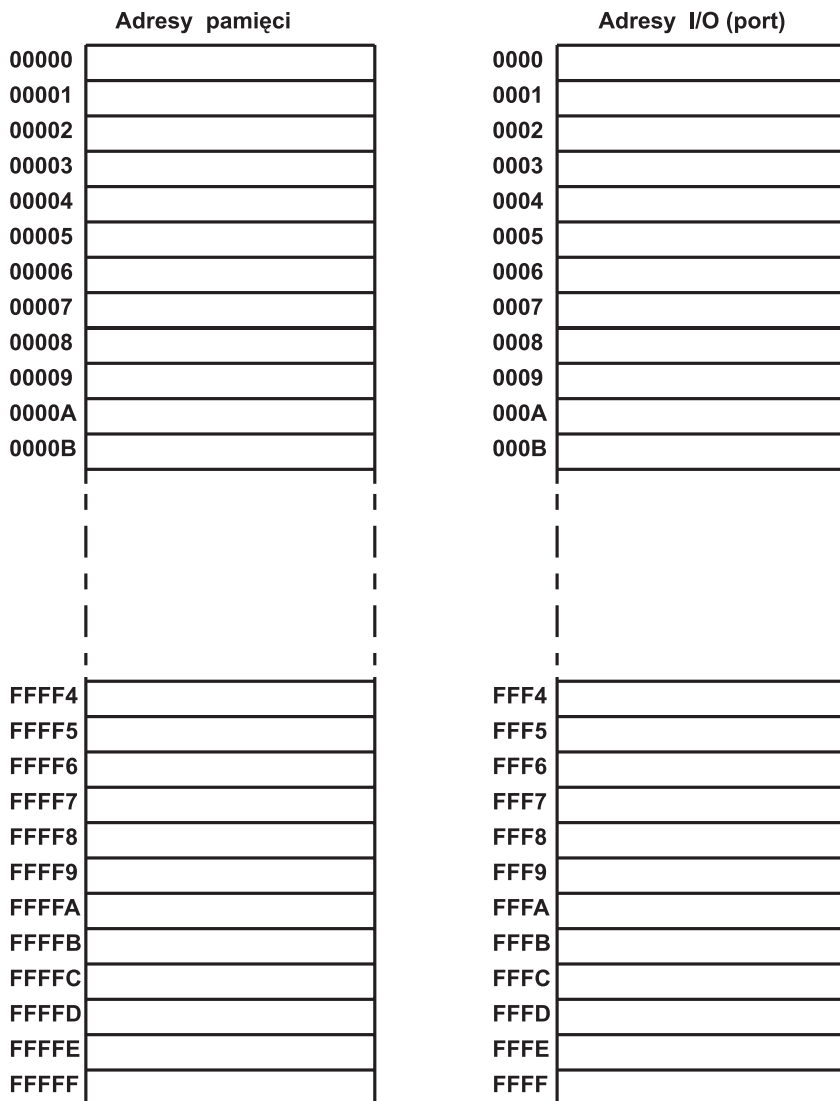
W procesorach Intel 8086/8088 ze względu na 20-bitową szynę adresową bit 21 jest obcinany. Natomiast w wyższych procesorach, współpracujących z 24- czy 32-bitową szyną, 21 bit umożliwia zaadresowanie trochę więcej pamięci niż 1 MB o te właśnie 64 KB-17 B. W komputerach z procesorami 80286, 80386 i nowszymi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



Rysunek 1.6.1. Adresy pamięci i adresy I/O dla procesorów 8086/8088

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 w 64 KB przestrzeni adresowej oddzielonej od 1 MB przestrzeni adresowej pamięci (patrz rysunek 1.6.1.). Przestrzeń adresowa I/O jest o wiele mniejsza niż 1 MB przestrzeń pamięci i dla 8086 ma wartość 64 KB; w rzeczywistości zaś 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