Język asembler dla każdego

Bogdan Drozdowski

Jak pisać programy w języku asembler?
Wstęp.

Pewnie wiele osób spośród Was słyszało już o tym, jaki ten asembler jest straszny. Ale te opinie możecie już śmiało wyrzucić do Kosza (użytkownicy Linuksa mogą skorzystać z /dev/null), gdyż przyszła pora na przeczytanie tego oto dokumentu.

Na początek, przyjrzyjmy się niewątpliwym zaletom języka asembler:
(przeskocz zalety)

1. Mały rozmiar kodu.
   Tego nie przebije żaden kompilator żadnego innego języka. Dlaczego? Oto kilka powodów:
   - Jako programista języka asembler (asma) wiesz co i gdzie w danej chwili się znajduje. Nie musisz ciągle przeładowywać zmiennych, co zabiera i miejsce i czas. Możesz eliminować instrukcje, które są po prostu zbędne.
   - Do twojego pięknego kodu nie są dołączane żadne biblioteki z dziesiątkami procedur, podczas gdy ty używasz tylko jednej z nich. Co za marnotrawstwo!
   - Jako znawca zestawu instrukcji wiesz, które z nich są krótsze.

2. Duża szybkość działania.
   Znając sztuczki optymalizacyjne, wiedząc, które instrukcje są szybsze, eliminując zbędne instrukcje z pętli, otrzymujesz kod nierzadko dziesiątki razy szybszy od tych napisanych w językach wysokiego poziomu (high-level languages, HLLs). Nieprzebijalne.

3. Wiedza.
   Nawet jeśli nie piszesz dużo programów w asmie, zdobywasz wprost bezcenną wiedzę o tym, jak naprawdę działa komputer i możesz zobaczyć, jak marną czasem robotę wykonują kompilatory HLL-ów. Zrozumiesz, czym jest wskaźnik i często popełniane błędy z nim związane.

4. Możliwość tworzenia zmiennych o dużych rozmiarach, a nie ograniczonych do 4 czy 8 bajtów. W asemblerze zmienne mogą mieć dowolną ilość bajtów.

5. Wstawki asemblerowe.
   Jeśli mimo to nie chcesz porzucić swojego ulubionego dotąd HLLa, to w niektórych językach istnieje możliwość wstawianie kodu napisanego w asemblerze wprost do twoich programów!

Teraz przyszła kolej na rzekome argumenty przeciwko językowi asembler:
(przeskocz wady)

1. Nieprzenośność kodu między różnymi maszynami.
   No cóż, prawda. Ale i tak większość tego, co napisane dla procesorów Intela będzie działało na procesorach AMD i innych zgodnych z x86. I na odwrót.
   Nieprzenośność jest chyba najczęściej używanym argumentem przeciwko asemblerowi. Jest on zwykle stawiany przez programistów języka C, którzy po udowodnieniu, jaki to język C jest wspaniały, wracają do pisania dokładnie w takim samym stopniu nie-przenośnych programów...
   Nie ukrywajmy: bez zmian kodu to tylko programy niewiele przewyższające Witaj, świecie

skompilują się i uruchomią pod różnymi systemami.

2. A nowoczesne kompilatory i tak produkują najlepszy kod...

   Nieprawda, i to z kilku powodów:

   - Kompilator używa zmiennych. No i co z tego, pytacie? A to, że pamięć RAM (o dyskach itp. nie wspominając) jest wiele, wiele razy wolniejsza niż pamięć procesora (czyli rejestry). Nawet pamięć podręczna (cache) jest sporo wolniejsza.
   - Kompilator nie wie, co się znajduje np. w danym rejestrze procesora, więc pewnie wpisze tam tą samą wartość. Co innego z programistą asma.
   - Kompilator nie jest w stanie przewidzieć, co będzie w danej chwili w innych rejestrach. Więc do rejestru, który chce zmienić i tak wpisze jakąś wartość zamiast użyć innego rejestru, co prawie zawsze jest szybsze a więc lepsze. Co innego zrobiłby programista asma.
   - Kompilator może używać dłuższych lub wolniejszych instrukcji.
   - Kompilator nie zawsze może poprzestawiać instrukcje, aby uzyskać lepszy kod. Programista asma widzi, co się w jego kodzie dzieje i może wybrać inne, lepsze rozwiązanie (np. zmniejszyć rozmiary pętli czy pozbyć się zależności między instrukcjami)
   - Kompilator może nie być świadomy technologii zawartych w procesorze. Programista asma wie, jaki ma procesor i za darmo ściąga do niego pełną dokumentację.

3. Brak bibliotek standardowych.
   I znowu nieprawda. Istnieje co najmniej kilka takich. Zawierają procedury wejścia, wyjścia, alokacji pamięci i wiele, wiele innych. Nawet sam taką jedną bibliotekę napisałem...

4. Kod wygląda dziwniej. Jest bardziej abstrakcyjny.
   Dziwniej - tak, ale nie oszukujmy się. To właśnie języki wysokiego poziomu są abstrakcyjne! Asembler przecież operuje na tym, co fizycznie istnieje w procesorze - na jego własnych rejestrach przy użyciu jego własnych instrukcji.

5. Mniejsza czytelność kodu.
   Kod w języku C można tak napisać, że nie zrozumie go nawet sam autor. Kod w asmie można tak napisać, że każdy go zrozumie. Wystarczy kilka słów wstępu i komentarze. W HLLach trzeba byłoby wszystkie struktury objaśniać.
   A wygląd i czytelność kodu zależą tylko od tego, czy dany programista jest dobry, czy nie.
   Dobry programista asemblera nie będzie miał większych kłopotów z odczytaniem kodu w asmie niż dobry programista C kodu napisanego w C.

6. Brak łatwych do zrozumienia instrukcji sterujących (if, while, ...)
   Przecież w procesorze nie ma nic takiego!
   Programista asma ma 2 wyjścia: albo używać prawdziwych instrukcji albo napisać własne makra, które takie instrukcje będą udawać (już są takie napisane). Ale nie ma nic uniwersalnego. Na jedną okazję można użyć takich instrukcji, a na inną - innych. Jednak zawsze można wybrać najszybszą wersję według własnego zdania, a nie według zdania kompilatora.

Asembler może i nie jest z początku łatwy do zrozumienia, ale wszystko przyjdzie wraz z doświadczeniem.

7. Trzeba pisać dużo kodu.
   No, tak. Jak się komuś męczą palce, to niech zostanie przy HLLach i żyje w świecie abstrakcji. Prawdziwym programistom nie będzie przecież takie coś przeszkadzać!
   Mówi się, że ZŁEJ baletnicy nawet rąbek u sukni przeszkadza.
   Poza tym, programista nad samym pisaniem kodu spędza ok 30% czasu przeznaczonego na program (reszta to plan wykonania, wdrażanie, utrzymanie, testowanie...). Nawet jeśli programiście asma zabiera to 2 razy więcej czasu niż programiście HLLa, to i tak zysk lub strata wynosi 15%.
   Dużo pisania sprawia, że umysł się uczy, zapamiętuje składnię instrukcji i nabiera doświadczenia.

8. Assmebler jest ciężki do nauki.
   Jak każdy nowy język. Nauka C lub innych podobnych przychodzi łatwo, gdy już się zna np. Pascala. Próba nauczenia się innych dziwnych języków zajmie dłużej, niż nauka asma.

9. Ciężko znajdować i usuwać błędy.
   Na początku równie ciężko, jak w innych językach. Pamiętacie jeszcze usuwanie błędów ze swoich pierwszych programów w C czy Pascalu?

10. Programy w asemblerze są ciężkie w utrzymaniu.
    Znowu powiem to samo: podobnie jest w innych językach. Najlepiej dany program zna jego autor, co wcale nie oznacza, że w przyszłości będzie dalej rozumiał swój kod (nawet napisany w jakimś HLLu). Dlatego ważne są komentarze. Zdolność do zajmowania się programem w przyszłości także przychodzi wraz z doświadczeniem.

11. Nowoczesne komputery są tak szybkie, że i tak szybkość nie robi to różnicy...
    Napiszmy program z czterema zagnieżdżonymi pętlami po 100 powtórzeń każda. Razem 100 000 000 (sto milionów) powtórzeń. Czas wykonania tego programu napisanego w jakimś HLLu liczy się w minutach, a często w dziesiątkach minut (czasem godzin - zależy od tego, co jest w pętlach). To samo zadanie napisane w asemblerze daje program, którego czas działania można liczyć w sekundach lub pojedynczych minutach!
    Po prostu najszybsze programy są pisane w asemblerze. Często otrzymuje się program 5-10 razy szybszy (lub jeszcze szybszy) niż ten w HLLu.

12. Chcesz mieć szybki program? Zmień algorytm, a nie język
    A co jeśli używasz już najszybszego algorytmu a on i tak działa za wolno?
    Każdy algorytm zawsze można zapisać w asemblerze, co poprawi jego wydajność. Nie wszystko da się zrobić w HLLu.

13. Nowoczesne komputery i tak mają dużo pamięci. Po co więc mniejsze programy?
    Wolisz mieć 1 wolno działający program o rozmiarze 1 MB, napisany w HLLu i robić 1 czynność w danej chwili, czy może wolisz wykonywać 10 czynności na raz dziesięcioma programami w asemblerze po 100kB każdy (no, przesadziłem - rzadko który program w asmie sięgnie aż tak gigantycznych rozmiarów!)?

To był tylko wstęp do bezkresnej wiedzy, jaką każdy z Was zdobędzie.

Ale nie myślcie, że całkowicie odradzam Wam języki wysokiego poziomu. Ja po prostu polecam Wam asemblera.

Najlepsze programy pisze się w czystym asemblerze, co sprawia niesamowitą radość, ale można przecież łączyć języki. Na przykład, część programu odpowiedzialną za wczytywanie danych lub wyświetlanie wyników można napisać w HLLu, a intensywne obliczeniowo pętle pisać w asmie, albo robiąc wstawki w kod, albo pisząc w ogóle oddzielne moduły i potem łączyć wszystko w całość.

Nauka tego wspaniałego języka przyjdzie Wam łatwiej, niż myślicie. Pomyślcie też, co powiedzą znajomi, gdy się dowiedzą, co umiecie!

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Jak pisać programy w języku asembler?
Część 1 - Podstawy, czyli czym to się je.

Wyobraźcie sobie, jakby to było móc programować maszynę bezpośrednio, czyli rozmawiać z procesorem bez pośrednictwa struktur wysokiego poziomu, np. takich jak spotykamy w języku C. Bezpośrednie operowanie na procesorze umożliwia przecież pełną kontrolę jego działań! Bez zbędnych instrukcji i innych śmieci spowalniających nasze programy.

Czy już czujecie chęć pisania najkrótszych i najszybszych programów na świecie?
Programów, których czasem w ogóle NIE MOŻNA napisać w innych językach? Brzmi wspaniale, prawda?

Tylko pomyślcie o tym, co powiedzieliby znajomi, gdybyście się im pochwalili. Widzicie już te ich zdumione miny?

Miła perspektywa, prawda? No, ale dość już gadania. Zabierajmy się do rzeczy!

Zacznijmy od krótkiego wprowadzenia:

Niedziesiętne systemy liczenia

1. Dwójkowy (binarny)

   Najprostszy dla komputera, gdzie coś jest albo włączone, albo wyłączone. System ten operuje na liczbach zwanych bitami (bit = binary digit = cyfra dwójkowa). Bit przyjmuje jedną z dwóch wartości: 0 lub 1.
   Na bajt składa się 8 bitów. Jednym bajtem można przedstawić więc 2^8=256 możliwości.

   Przeliczenie liczby zapisanej w systemie dwójkowym na dziesiętny jest proste. Podobnie jak w systemie dziesiętnym, każdą cyfrę mnożymy przez odpowiednią potęgę podstawy (podstawa wynosi 2 w systemie dwójkowym, 10 w systemie dziesiętnym).

   Oto przykład (niech daszek ^ oznacza potęgowanie):
   1010 1001 dwójkowo =
   1*(2^7) + 0*(2^6) + 1*(2^5) + 0*(2^4) + 1*(2^3) + 0*(2^2) + 0*(2^1) + 1*(2^0) =
   128 + 32 + 8 + 1 =
   169 dziesiętnie (lub dec, od decimal).

   Działanie odwrotne też nie jest trudne: naszą liczbę dzielimy ciągle (do chwili uzyskania ilorazu równego 0) przez 2, po czym zapisujemy reszty z dzielenia wspak:
   (przeskocz konwersję liczby dziesiętnej na dwójkową)

```
169     |
 84     | 1
 42     | 0
 21     | 0
 10     | 1
  5     | 0
  2     | 1
  1     | 0
  0     | 1
```

Wspak dostajemy: 1010 1001, czyli wyjściową liczbę.

2. Szesnastkowy (heksadecymalny, w skrócie hex)

Jako że system dwójkowy ma mniej cyfr niż dziesiętny, do przedstawienia względnie małych liczb trzeba użyć dużo zer i jedynek. Jako że bajt ma 8 bitów, podzielono go na dwie równe, 4-bitowe części. Teraz bajt można już reprezentować dwoma znakami, a nie ośmioma. Na każdy taki znak składa się 2^4=16 możliwości. Stąd wzięła się nazwa szesnastkowy.
Powstał jednak problem: cyfr jest tylko 10, a trzeba mieć 16. Co zrobić?
Postanowiono liczbom 10-15 przyporządkować odpowiednio znaki A-F.
Np.
Liczba 255 dziesiętnie = 1111 1111 binarnie = FF szesnastkowo (1111 bin = 15 dec = F hex)
Liczba 150 dziesiętnie = 1001 0110 binarnie = 96 szesnastkowo.

Należy zauważyć ścisły związek między systemem dwójkowym i szesnastkowym: 1 cyfra szesnastkowa to 4 bity, co umożliwia błyskawiczne przeliczanie między obydwoma systemami: wystarczy tłumaczyć po 4 bity (1 cyfrę hex) na raz i zrobione.

Przeliczenie liczby zapisanej w systemie szesnastkowym na dziesiętny jest równie proste, jak tłumaczenie z dwójkowego na dziesiętny. Każdą cyfrę mnożymy przez odpowiednią potęgę podstawy (podstawa wynosi 16 w systemie szesnastkowym).

Oto przykład:
10A szesnastkowo =
1*16^2 + 0*16^1 + A*16^0 =
256 + 0 + 10 =
266 dziesiętnie.

Działanie odwrotne też nie jest trudne: naszą liczbę dzielimy ciągle (do chwili uzyskania ilorazu=0) przez 16, po czym zapisujemy reszty z dzielenia wspak:
(przeskocz konwersję liczby dziesiętnej na szesnastkową)

```
266    |
16     | 10
 1     | 0
 0     | 1
```

Wspak dostajemy kolejno: 1, 0 i 10, czyli 10A, czyli wyjściową liczbę.

Podczas pisania programów, liczby w systemie szesnastkowym oznacza się przez dodanie na końcu litery h (lub z przodu 0x), a liczby w systemie dwójkowym - przez dodanie litery b.
Tak więc, 101 oznacza dziesiętną liczbę o wartości 101, 101b oznacza liczbę 101 w systemie dwójkowym (czyli 5 w systemie dziesiętnym), a 101h lub 0x101 oznacza liczbę 101 w systemie szesnastkowym (czyli 257 dziesiętnie).

Język asembler i rejestry procesora

Co to w ogóle jest asembler?

Asembler jest to język programowania, należący do języków niskiego poziomu. Znaczy to tyle, że jednej komendzie asemblera odpowiada dokładnie jeden rozkaz procesora. Asembler operuje na rejestrach procesora.

A co to jest rejestr procesora?
Rejestr procesora to zespół układów elektronicznych, mogący przechowywać informacje (taka własna pamięć wewnętrzna procesora).
Zaraz podam Wam podstawowe rejestry, na których będziemy operować. Wiem, że ich ilość może przerazić, ale od razu mówię, abyście NIE uczyli się tego wszystkiego na pamięć! Najlepiej zrobicie, czytając poniższą listę tylko 2 razy, a potem wracali do niej, gdy jakikolwiek rejestr pojawi się w programach, które będę później prezentował w ramach tego kursu.
Oto lista interesujących nas rejestrów:

1. ogólnego użytku:

   - akumulator:
     RAX (64 bity) = EAX (młodsze 32 bity) + starsze 32 bity,
     EAX (32 bity) = AX (młodsze 16 bitów) + starsze 16 bitów,
     AX = AH (starsze 8 bitów) + AL (młodsze 8 bitów)
     Rejestr ten najczęściej służy do wykonywania działań matematycznych, ale często w tym rejestrze lub jego części (AX lub AH) będziemy mówić systemowi operacyjnemu i BIOS-owi, co od niego chcemy.

   - rejestr bazowy:
     RBX (64 bity) = EBX (młodsze 32 bity) + starsze 32 bity,
     EBX (32 bity) = BX (młodsze 16 bitów) + starsze 16 bitów,
     BX = BH (starsze 8 bitów) + BL (młodsze 8 bitów)
     Ten rejestr jest używany np. przy dostępie do tablic.

   - licznik:
     RCX (64 bity) = ECX (młodsze 32 bity) + starsze 32 bity,
     ECX (32 bity) = CX (młodsze 16 bitów) + starsze 16 bitów,
     CX = CH (starsze 8 bitów) + CL (młodsze 8 bitów)
     Tego rejestru używamy np. do określania ilości powtórzeń pętli.

   - rejestr danych:
     RDX (64 bity) = EDX (młodsze 32 bity) + starsze 32 bity,
     EDX (32 bity) = DX (młodsze 16 bitów) + starsze 16 bitów,
     DX = DH (starsze 8 bitów) + DL (młodsze 8 bitów)
     W tym rejestrze przechowujemy adresy różnych zmiennych. Jak wkrótce zobaczymy, do tego rejestru będziemy wpisywać adres napisu, który będziemy chcieli wyświetlić.

   - rejestry dostępne tylko w trybie 64-bitowym:
     - 8 rejestrów 8-bitowych: R8B, ..., R15B
     - 8 rejestrów 16-bitowych: R8W, ..., R15W
     - 8 rejestrów 32-bitowych: R8D, ..., R15D
     - 8 rejestrów 64-bitowych: R8, ..., R15

   - rejestry indeksowe:

◊ indeks źródłowy:
RSI (64 bity) = ESI (młodsze 32 bity) + starsze 32 bity,
ESI (32 bity) = SI (młodsze 16 bitów) + starsze 16 bitów,
SI (16 bitów) = SIL (młodsze 8 bitów) + starsze 8 bitów (tylko tryb 64-bit)

◊ indeks docelowy:
RDI (64 bity) = EDI (młodsze 32 bity) + starsze 32 bity,
EDI (32 bity) = DI (młodsze 16 bitów) + starsze 16 bitów,
DI (16 bitów) = DIL (młodsze 8 bitów) + starsze 8 bitów (tylko tryb 64-bit)
Rejestry indeksowe najczęściej służą do operacji na długich łańcuchach danych, w tym napisach i tablicach.

♦ rejestry wskaźnikowe:
◊ wskaźnik bazowy:
RBP (64 bity) = EBP (młodsze 32 bity) + starsze 32 bity,
EBP (32 bity) = BP (młodsze 16 bitów) + starsze 16 bitów.
BP (16 bitów) = BPL (młodsze 8 bitów) + starsze 8 bitów (tylko tryb 64-bit)
Najczęściej służy do dostępu do zmiennych lokalnych.

◊ wskaźnik stosu:
RSP (64 bity) = ESP (młodsze 32 bity) + starsze 32 bity,
ESP (32 bity) = SP (młodsze 16 bitów) + starsze 16 bitów.
SP (16 bitów) = SPL (młodsze 8 bitów) + starsze 8 bitów (tylko tryb 64-bit)
Służy do dostępu do stosu (o tym nieco później).

◊ wskaźnik instrukcji:
RIP (64 bity) = EIP (młodsze 32 bity) + starsze 32 bity,
EIP (32 bity) = IP (młodsze 16 bitów) + starsze 16 bitów.
Mówi procesorowi, skąd ma pobierać instrukcje do wykonywania.

2. rejestry segmentowe (wszystkie 16-bitowe):

♦ segment kodu CS - mówi procesorowi, gdzie znajdują się dla niego instrukcje.
♦ segment danych DS - ten najczęściej pokazuje na miejsce, gdzie trzymamy nasze zmienne.
♦ segment stosu SS - dzięki niemu wiemy, w którym segmencie jest nasz stos. O tym, czym w ogóle jest stos, powiem w następnej części.
♦ segment dodatkowy ES - często używany, gdy chcemy coś napisać lub narysować na ekranie bez pomocy Windows, DOSa czy nawet BIOSu.
♦ FS i GS (dostępne dopiero od 80386) - nie mają specjalnego przeznaczenia. Są tu na wypadek, gdyby zabrakło nam innych rejestrów segmentowych.

3. rejestr stanu procesora: FLAGI (16-bitowe), E-FLAGI (32-bitowe) lub R-FLAGI (64-bitowe). Służą one przede wszystkim do badania wyniku ostatniej operacji (np. czy nie wystąpiło przepełnienie, czy wynik jest zerem, itp.). Najważniejsze flagi to CF (carry flag - flaga przeniesienia), OF (overflow flag - flaga przepełnienia), SF (sign flag - flaga znaku), ZF (zero flag - flaga zera), IF (interrupt flag - flaga przerwań), PF (parity flag - flaga parzystości), DF (direction flag - flaga kierunku).

Użycie litery R przed symbolem rejestru, np. RCX, oznacza rejestr 64-bitowy, dostępny tylko na procesorach 64-bitowych.

Użycie litery E przed symbolem rejestru, np. EAX, oznacza rejestr 32-bitowy, dostępny tylko na procesorach rodziny 80386 lub lepszych. Nie dotyczy to rejestru ES.

Napisy

> RAX = EAX+starsze 32 bity; EAX=AX + starsze 16 bitów; AX=AH+AL

oznaczają takie zależności między tymi rejestrami:
(przeskocz rozwinięcie rejestru RAX)

```
                              RAX (64 bity)
                                   |                EAX (32b)
00000000000000000000000000000000 | 0000000000000000 | 00000000 | 00000000
                32b                |        16b       |        AX(16b)
                                   |                  |   AH(8b) |   AL(8b)
```

Napisy

> RSI = ESI + starsze 32 bity; ESI = SI + starsze 16 bitów; SI = SIL+starsze 8 bitów

oznaczają:
(przeskocz rozwinięcie rejestru RSI)

```
                              RSI (64 bity)
                                   |                ESI (32b)
00000000000000000000000000000000 | 0000000000000000 | 00000000 | 00000000
                32b                |        16b       |        SI(16b)
                                   |                  |    8b    |  SIL(8b)
```

Tak, w DOSie można używać rejestrów 32-bitowych (o ile posiada się 80386 lub nowszy). Można nawet 64-bitowych, jeśli tylko posiada się właściwy procesor.

Jedna ważna uwaga - między nazwami rejestrów może pojawić się dwukropek w dwóch różnych znaczeniach:

- zapis DX : AX (lub 2 dowolne zwykłe rejestry) będzie oznaczać liczbę, której starsza część znajduje się w rejestrze po lewej stronie (DX), a młodsza - w tym z prawej (AX). Wartość liczby wynosi DX*65536 + AX.
- zapis CS : SI (rejestr segmentowy + dowolny zwykły) będzie najczęściej oznaczać wskaźnik do jakiegoś obiektu w pamięci (o pamięci opowiem następnym razem). Rejestr segmentowy zawiera oczywiście segment, w którym znajduje się ów obiekt, a rejestr zwykły - offset (przesunięcie, adres w tym segmencie) tegoż obiektu.

Na razie nie musicie się przejmować tymi dwukropkami. Mówię to tylko dlatego, żebyście nie byli zaskoczeni, gdyż w przyszłości się pojawią.

Programista może odnosić się bezpośrednio do wszystkich wymienionych rejestrów, z wyjątkiem *IP oraz flag procesora (z wyjątkami).

Jak widać po ich rozmiarach, do rejestrów 8-bitowych można wpisać liczbę z przedziału 0-255 (lub od -128 do 127, gdy najwyższy, siódmy bit służy nam jako bit oznaczający znak liczby), w 16-bitowych zmieszczą się liczby 0-65535 (od -32768 do 32767), a w 32-bitowych - liczby od 0 do 4.294.967.295 (od -2.147.483.648 do

2.147.483.647)

Dobrym, choć trudnym w odbiorze źródłem informacji są: *Intel Architecture Software Developer's Manual* (IASDM) dostępny ZA DARMO ze stron Intela oraz DARMOWE podręczniki *AMD64 Architecture Programmer's Manual* firmy AMD

Pisanie i kompilowanie (asemblowanie) swoich programów
Jak pisać programy w asemblerze?

Należy zaopatrzyć się w:

- Edytor tekstu, mogący zapisywać pliki formatu TXT (bez formatowania), np. Programmer's File Editor, Quick Editor, The Gun (wszystkie są na www.movsd.com) czy zwykły Notatnik
- Kompilator języka asembler (patrz dalej)
- Odpowiedni program łączący (kosolidator, ang. linker), chyba że kompilator ma już taki wbudowany, jak np. A86, NASM lub FASM (patrz dalej)

Wtedy wystarczy napisać w edytorze tekstu plik zawierający komendy procesora (o tym później), zapisać go z rozszerzeniem .ASM, po czym użyć kompilatora, aby przetworzyć program na kod rozumiany przez procesor.

Jakiego kompilatora użyć?

Istnieje wiele kompilatorów języka asembler. Do najpopularniejszych należą Turbo asembler firmy Borland, Microsoft Macro asembler (MASM), Netwide asembler Project (NASM), A86/A386, NBASM, FASM, HLA.

Można je ściągnąć z internetu:
(przeskocz adresy stron kompilatorów)

- Strona główna NASMa: sf.net/projects/nasm
- A86 z eji.com
- Flat asembler (FASM): flatasembler.net
- MASM z Webster.cs.ucr.edu lub z www.movsd.com (wersje 32-bitowe)
- HLA Webster.cs.ucr.edu

Po skompilowaniu pliku z kodem źródłowym należy użyć programu łączącego, dostępnego zwykle z odpowiednim kompilatorem (np. tlink z tasm, link z masm). Mamy więc już wszystko, co potrzeba. Zaczynamy pisać. Będę tutaj używał składni Turbo asemblera zgodnego z MASMem oraz FASMa i NASMa.
(przeskocz program w wersji TASM)

```
; wersja TASM
.model tiny
.code
org 100h

start:
        mov     ah, 9
        mov     dx, offset info
        int     21h

        mov     ah, 0
```

```
        int     16h

        mov     ax, 4C00h
        int     21h

info    db      "Czesc.$"

end start
```

Teraz wersja NASM:
(przeskocz program w wersji NASM)

```
; wersja NASM

; nie ma ".model" ani ".code"
; tu można wstawić:
; section .text
; aby dać znać NASMowi, że to będzie w sekcji kodu.
; Nie jest to jednak wymagane, bo to jest sekcja domyślna.

org 100h

start:                          ; nawet tego NASM nie wymaga
        mov     ah, 9
        mov     dx, info        ; nie ma słowa "offset"
        int     21h

        mov     ah, 0
        int     16h

        mov     ax, 4C00h
        int     21h

info    db      "Czesc.$"

; nie ma "end start"
```

Teraz wersja FASM
(przeskocz program w wersji FASM)

```
; wersja FASM

format binary

; nie ma ".model" ani ".code"
org 100h

start:                          ; nawet tego FASM nie wymaga
        mov     ah, 9
        mov     dx, info        ; nie ma słowa "offset"
        int     21h

        mov     ah, 0
        int     16h

        mov     ax, 4C00h
        int     21h

info    db      "Czesc.$"
```

```
        ; nie ma "end start"
```

Bez paniki! Teraz omówimy dokładnie, co każda linia robi.

- linie lub napisy zaczynające się średnikiem

  Traktowane są jako komentarze i są całkowicie ignorowane przy kompilacji. Rozmiar skompilowanego programu wynikowego nie zależy od ilości komentarzy. Dlatego najlepiej wstawiać tyle komentarzy, aby inni (również my) mogli później zrozumieć nasz kod.

- `.model tiny` (pamiętajcie o kropce) lub `format binary` (w FASMie)

  Wskazuje kompilatorowi rodzaj programu. Jest kilka takich dyrektyw:
  - tiny: kod i dane mieszczą się w jednym 64kB segmencie. Typowy dla programów typu .com
  - small: kod i dane są w różnych segmentach, ale obydwa są mniejsze od 64kB
  - medium: kod może być > 64kB, ale dane muszą być < 64kB
  - compact: kod musi być < 64kB, dane mogą mieć więcej niż 64kB
  - large: kod i dane mogą być > 64kB, ale tablice muszą być < 64kB
  - huge: kod, dane i tablice mogą być > 64kB

- `.code` (też z kropką)

  Wskazuje początek segmentu, gdzie znajduje się kod programu. Można jednak w tym segmencie umieszczać dane, ale należy to robić tak, aby nie stały się one częścią programu. Zwykle wpisuje się je za ostatnią komendą kończącą program. Procesor przecież nie wie, co jest pod danym adresem i z miłą chęcią potraktuje to coś jako instrukcję, co może prowadzić do przykrych konsekwencji. Swoje dane umieszczajcie tak, aby w żaden sposób strumień wykonywanych instrukcji nie wszedł na nie. Są też inne dyrektywy: .data, deklarująca początek segmentu z danymi oraz .stack, deklarująca segment stosu (o tym później), której nie można używać w programach typu .com, gdzie stos jest automatycznie ustawiany.

- `org 100h` (bez kropki)

  Ta linia mówi kompilatorowi, że nasz kod będzie (dopiero po uruchomieniu!) znajdował się pod adresem 100h (256 dziesiętnie) w swoim segmencie. To jest typowe dla programów .com. DOS, uruchamiając taki program, szuka wolnego segmentu i kod programu umieszcza dopiero pod adresem (czasami zwanym offsetem - przesunięciem) 100h.
  Co jest więc wcześniej? Wiele ciekawych informacji, z których chyba najczęściej używaną jest linia poleceń programu (parametry uruchomienia, np. różne opcje itd.).
  Dyrektywa org podana na początku kodu NIE wpływa na rozmiar programu, ułatwia kompilatorowi określenie adresów różnych etykiet (w tym danych) znajdujących się w programie.
  Jeśli chcemy tworzyć programy typu .com, należy zawsze podać org 100h i opcję /t dla Turbo Linkera.

- `start:` (z dwukropkiem) i `end start` (bez dwukropka)

  Mówią kompilatorowi, gdzie są odpowiednio: początek i koniec programu.

- `mov ah,9`

Do 8-bitowego rejestru AH (górnej części 16-bitowego AX) wstaw (MOV = move, przesuń) wartość 9. Po co i czemu akurat 9? Zaraz zobaczymy.
Najpierw powiem o czymś innym: komenda MOV ma ważne ograniczenia:

1. nie można skopiować jedną komendą MOV komórki pamięci do innej komórki pamięci, czyli takie coś:

```
mov     [a], [b]
```

   (gdzie a i b - dwie zmienne w pamięci) jest zabronione.
   O tym, co oznaczają nawiasy kwadratowe, czyli o adresowaniu zmiennych w pamięci - następnym razem.
2. nie można skopiować jedną komendą MOV jednego rejestru segmentowego (cs,ds,es,ss,fs,gs) do innego rejestru segmentowego, czyli działanie

```
mov     es, ds
```

   jest zabronione.
3. Nie można do rejestru segmentowego bezpośrednio wpisać jakieś wartości, czyli nie można

```
mov     ds, 0
```

   ale można:

```
mov     bx, 0
mov     ds, bx
```

- `mov dx,offset info`

  Do rejestru danych (DX, 16-bitowy) wstaw offset (adres względem początku segmentu) etykiety info. Można obliczać adresy nie tylko danych, ale etykiet znajdujących się w kodzie programu.

- `int 21h`

  INT = interrupt = przerwanie. Nie jest to jednak znane np. z kart dźwiękowych przerwanie typu IRQ. Wywołując przerwanie 21h (33 dziesiętnie) uruchamiamy jedną z funkcji DOSa. Którą? O tym zazwyczaj mówi rejestr AX. W spisie przerwań Ralfa Brown'a (RBIL) patrzymy:
  (przeskocz opis int 21h, ah=9)

```
INT 21 - DOS 1+ - WRITE STRING TO STANDARD OUTPUT
AH = 09h
DS:DX -> $-terminated string
```

  Już widzimy, czemu do AH poszła wartość 9. Chcieliśmy uruchomić funkcję, która wypisuje na na ekran ciąg znaków zakończony znakiem dolara. Adres tego ciągu musi się znajdować w parze rejestrów: DS wskazuje segment, w którym znajduje się ten ciąg, a DX - jego adres w tym segmencie. Dlatego było mov dx,offset info.
  Zaraz, zaraz! Ale przecież my nic nie robiliśmy z DS, a dane znajdują się przecież w segmencie kodu! I to działa?
  Oczywiście! Programy typu .com są małe. Tak małe, że mieszczą się w jednym segmencie pamięci. Dlatego przy ich uruchamianiu DOS ustawia nam CS=DS=ES=SS. Nie musimy się więc o to martwić.

- `mov ah,0`

  Do rejestru AH wpisz 0. Czemu? Zaraz zobaczymy. Ale najpierw wspomnę o czymś innym. Otóż,

  ```
  mov     rejestr, 0
  ```

  nie jest najlepszym sposobem na wyzerowanie danego rejestru. Szybsze lub krótsze są dwa inne:

  ```
  xor     rej1, rej1 ; 1 xor 1 = 0 oraz 0 xor 0 = 0.
                     ; Stąd   "coś XOR to_samo_coś"
                     ; zawsze daje 0.

  sub     rej1, rej1      ; sub=substract=odejmij.
                          ; rej1 - rej1 = 0
  ```

  Ja zwykle używam XOR.

- `int 16h`

  Kolejne przerwanie, więc znowu do listy Ralfa Brown'a:
  (przeskocz opis int 16h, ah=0)

  ```
  INT 16 - KEYBOARD - GET KEYSTROKE
  AH = 00h
  Return: AH = BIOS scan code
          AL = ASCII character
  ```

  Ta funkcja pobiera znak z klawiatury i zwraca go w rejestrze AL. Jeśli nie naciśnięto nic, poczeka, aż użytkownik naciśnie.

- `mov ax,4c00h`

  Do rejestru AX wpisujemy wartość 4c00 szesnastkowo.

- `int 21h`

  Znów przerwanie DOSa, funkcja 4ch. Patrzymy do RBIL:
  (przeskocz opis int 21h, ah=4ch)

  ```
  INT 21 - DOS 2+ - "EXIT" - TERMINATE WITH RETURN CODE
        AH = 4Ch
        AL = return code
        Return: never returns
  ```

  Jak widzimy, ta funkcja powoduje wyjście z powrotem do DOSa, z numerem błędu (errorlevel) w AL równym 0. Przyjmuje się, że 0 oznacza, iż program zakończył się bez błędów. Jak widać po rozmiarze rejestru AL (8 bitów), program może wyjść z 2^8=256 różnymi numerami błędu.

- `info db "Czesc.$"`

  Etykietą info opisujemy kilka bajtów, w tym przypadku zapisanych jako ciąg znaków.
  A po co znak dolara $? Jak sobie przypomnimy, funkcja 9. przerwania DOSa wypisuje ciąg znaków zakończony właśnie na znak dolara $. Gdyby tego znaczka nie było, DOS wypisywałby różne śmieci

z pamięci, aż trafiłby na przypadkowy znak dolara $ nie wiadomo gdzie. O deklarowaniu zmiennych będzie w następnej części.

- `end start`

  Koniec programu.

Programik kompilujemy poleceniem:

```
tasm naszplik.asm
tlink naszplik.obj /t
```

(opcja /t - utworzy plik typu .com).
Lub, dla NASMa:

```
nasm -o naszplik.com -f bin naszplik.asm
```

(-o = nazwa pliku wyjściowego
-f = format pliku. Bin = binarny = np. .com lub .sys).

Lub, dla FASMa:

```
fasm naszplik.asm naszplik.com
```

Uruchamiamy naszplik.com i cieszymy się swoim dziełem.
Miłego eksperymentowania.

> Na świecie jest 10 rodzajów ludzi:
> ci, którzy rozumieją liczby binarne i ci, którzy nie.

Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia:

1. Poeksperymentujcie sobie, wstawiając z różne znaki do napisu. Na przykład, znaki o kodach ASCII 10 (Line Feed), 13 (Carriage Return), 7 (Bell). Pamiętajcie tylko, że znak dolara $ musi być ostatni, dlatego róbcie coś w stylu:

   ```
   info db "Czesc.", 00, 01, 02, 07, 10, 13, 10, 13, "$"
   ```

Jak pisać programy w języku asembler?
Część 2 - Pamięć, czyli gdzie upychać coś, co się nie mieści w procesorze.

Poznaliśmy już rejestry procesora. Jak widać, jest ich ograniczona ilość i nie mają one zbyt dużego rozmiaru. Rejestry ogólnego przeznaczenia są co najwyżej 32-bitowe (4-bajtowe). Dlatego często programista musi niektóre zmienne umieszczać w pamięci. Przykładem tego był napis, który wyświetlaliśmy w poprzedniej części artykułu. Był on zadeklarowany dyrektywą DB, co oznacza declare byte. Ta dyrektywa niekoniecznie musi deklarować dokładnie 1 bajt. Tak jak widzieliśmy, można nią deklarować napisy lub kilka bajtów pod rząd. Teraz omówimy rodzinę dyrektyw służących właśnie do rezerwowania pamięci.

Ogólnie, zmienne można deklarować jako bajty (dyrektywą DB, coś jak char w języku C), słowa (word = 16 bitów = 2 bajty, coś jak short w C) dyrektywą DW, podwójne słowa DD (double word = dword = 32bity = 4 bajty, jak long w C), potrójne słowa pword = 6 bajtów - PW, poczwórne słowa DQ (quad word = qword = 8 bajtów, typ long long), tbyte = 10 bajtów - DT (typ long double w C).

Przykłady (zakomentowane zduplikowane linijki są w składni TASMa):
(przeskocz przykłady)

```
dwa             db 2
szesc_dwojek    db 2, 2, 2, 2, 2, 2
litera_g        db "g"
_ax             dw 4c00h        ; 2-bajtowa liczba całkowita
alfa            dd 12348765h    ; 4-bajtowa liczba całkowita

;liczba_a       dq 1125         ; 8-bajtowa liczba całkowita. NASM
                                ; tego nie przyjmie, zamienimy to na
                                ; postać równoważną:
liczba_a        dd 1125, 0      ; 2 * 4 bajty

liczba_e        dq 2.71         ; liczba zmiennoprzecinkowa
                                ; podwójnej precyzji (double)

;duza_liczba    dt 6af4aD8b4a43ac4d33h  ; 10-bajtowa liczba całkowita.
                                        ; NASM/FASM tego nie przyjmie,
                                        ; zrobimy to tak:
duza_liczba     dd 43ac4d33h, 0f4aD8b4ah; czemu z zerem z przodu?
                                        ; Czytaj dalej
                db 6ah

pi              dt 3.141592
;nie_init       db ?            ; niezainicjalizowany bajt.
                                ; Wartość nieznana.
                                ; NASM tak tego nie przyjmie.
                                ; Należy użyć:
nie_init        resb 1
                                ; zaś dla FASMa:
;nie_init       rb 1

napis1          db "NaPis1."
xxx             db 1
                db 2
                db 3
                db 4
```

Zwróćcie uwagę na sposób rozbijania dużych liczb na poszczególne bajty: najpierw deklarowane są młodsze bajty, a potem starsze (np. dd 11223344h jest równoznaczne z db 44h, 33h, 22h, 11h). To działa, gdyż

procesory Intela i AMD (i wszystkie inne klasy x86) są procesorami typu little-endian, co znaczy, że najmłodsze bajty danego ciągu bajtów są umieszczane przez procesor w najniższych adresach pamięci. Dlatego my też tak deklarujemy nasze zmienne.

Ale z kolei takie coś:

```
beta    db aah
```

nie podziała. Dlaczego? KAŻDA liczba musi zaczynać się od cyfry. Jak to obejść? Tak:

```
beta    db 0aah
```

czyli poprzedzić zerem.
Nie podziała również to:

```
0gamma  db      9
```

Dlaczego? Etykiety (dotyczy to tak danych, jak i kodu programu) nie mogą zaczynać się od cyfr.

A co, jeśli chcemy zadeklarować zmienną, powiedzmy, składającą się z 234 bajtów równych zero? Trzeba je wszystkie napisać?
Ależ skąd! Należy użyć operatora duplicate. Odpowiedź na pytanie brzmi (TASM):

```
zmienna         db      234     dup(0)
nazwa           typ     ilość       co zduplikować
```

Lub, dla NASMa i FASMa:

```
zmienna:        TIMES    234    db      0
nazwa                   ilość   typ    co zduplikować
```

A co, jeśli chcemy mieć dwuwymiarową tablicę podwójnych słów o wymiarach 25 na 34?
Robimy tak (TASM) :

```
Tablica         dd      25      dup (34 dup(?))
```

Lub, dla NASMa i FASMa na przykład tak:

```
Tablica:        TIMES    25*34  dd      0
```

Do obsługi takich tablic przydadzą się bardziej skomplikowane sposoby adresowania zmiennych. O tym za moment.

Zmiennych trzeba też umieć używać.
Do uzyskania adresu danej zmiennej używa się operatora (słowa kluczowego) offset (TASM), tak jak widzieliśmy wcześniej. Zawartość zmiennej otrzymuje się poprzez umieszczenie jej w nawiasach kwadratowych. Oto przykład:

```
rejestr_ax      dw      4c00h
rejestr_bx      dw      ?               ; nie w NASMie/FASMie.
                                        ; użyć np. 0 zamiast "?"
rejestr_cl      db      ?               ; jak wyżej
...
        mov     [rejestr_bx], bx
```

```
        mov     cl, [rejestr_cl]
        mov     ax, [rejestr_ax]
        int     21h
```

Zauważcie zgodność rozmiarów zmiennych i rejestrów.
Możemy jednak mieć problem w skompilowaniu czegoś takiego:

```
mov     [jakas_zmienna], 2
```

Dlaczego? Kompilator wie, że gdzieś zadeklarowaliśmy jakas_zmienna, ale nie wie, czy było to

```
jakas_zmienna   db      0
```

czy

```
jakas_zmienna   dw      22
```

czy może

```
jakas_zmienna   dd      "g"
```

Chodzi o to, aby pokazać, jaki rozmiar ma obiekt docelowy. Nie będzie problemów, gdy napiszemy:

```
mov     word ptr [jakas_zmienna], 2     ; TASM
mov     word [jakas_zmienna], 2         ; NASM/FASM - bez PTR
```

I to obojętnie, czy zmienna była bajtem (wtedy następny bajt będzie równy 0), czy słowem (wtedy będzie ono miało wartość 2) czy może podwójnym słowem lub czymś większym (wtedy 2 pierwsze bajty zostaną zmienione, a pozostałe nie). Dzieje się tak dlatego, że zmienne zajmują kolejne bajty w pamięci, najmłodszy bajt w komórce o najmniejszym adresie. Na przykład:

```
xxx     dd      8
```

jest równoważne:

```
xxx     db      8,0,0,0
```

oraz:

```
xxx     db      8
        db      0
        db      0
        db      0
```

Te przykłady nie są jedynymi sposobami adresowania zmiennych (poprzez nazwę). Ogólny schemat wygląda tak:

Używając rejestrów 16-bitowych:
[(BX albo BP) lub (SI albo DI) lub liczba]
słowo albo wyklucza wystąpienie obu rejestrów na raz
np.

```
        mov     al, [ nazwa_zmiennej+2 ]
        mov     [ di-23 ], cl
        mov     al, [ bx + si + nazwa_zmiennej+18 ]
```

nazwa_zmiennej to też liczba, obliczana zazwyczaj przez linker.

W trybie rzeczywistym (np. pod DOSem) pamięć podzielona jest na segmenty, po 64kB (65536 bajtów) każdy, przy czym każdy kolejny segment zaczynał się 16 bajtów dalej niż wcześniejszy (nachodząc na niego). Pamięć adresowalna wynosiła maksymalnie 65536 (maks. liczba segmentów) * 16 bajtów/segment = 1MB. O tym limicie powiem jeszcze dalej.
(przeskocz ilustrację ułożenia segmentów)

```
Ułożenie kolejnych segmentów względem siebie

        segment o numerze 0
0       +-----------------+
        |                 |     segment o numerze 1
10h     +-----------------+     +-----------------+
        |                 |     |                 |     segment o numerze 2
20h     +-----------------+     +-----------------+     +-----------------+
        |                 |     |                 |     |                 |
30h     +-----------------+     +-----------------+     +-----------------+
        |                 |     |                 |     |                 |
```

Słowo offset oznacza odległość jakiegoś miejsca od początku segmentu. Adresy można było pisać w postaci SEG:OFF. Adres liniowy (prawdziwy) otrzymywało się mnożąc segment przez 16 (liczba bajtów) i dodając do otrzymanej wartości offset, np. adres segmentowy 1111h:2222h = adres bezwzględny 13332h (h = szesnastkowy).
Należy też dodać, że różne adresy postaci SEG:OFF mogą dawać w wyniku ten sam adres rzeczywisty. Oto przykład: 0040h:0072h = (seg*16+off) 400h + 72h = 00472h = 0000h:0472h.

Na procesorach 32-bitowych (od 386) odnoszenie się do pamięci może (w kompilatorze TASM należy po dyrektywie .code dopisać linię niżej .386) odbywać się wg schematu:

zmienna [rej_baz + rej_ind * skala +- liczba] (tylko TASM/MASM)
lub
[zmienna + rej_baz + rej_ind * skala +- liczba]
gdzie:

- zmienna oznacza nazwę zmiennej i jest to liczba obliczana przez kompilator lub linker
- rej_baz (rejestr bazowy) = jeden z rejestrów EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- rej_ind (rejestr indeksowy) = jeden z rejestrów EAX, EBX, ECX, EDX, ESI, EDI, EBP (bez ESP)
- mnożnik (scale) = 1, 2, 4 lub 8 (gdy nie jest podany, przyjmuje się 1)

Tak, tego schematu też można używać w DOSie.

Przykłady:

```
        mov     al, [ nazwa_zmiennej+2 ]
        mov     [ edi-23 ], cl
        mov     dl, [ ebx + esi*2 + nazwa_zmiennej+18 ]
```

Na procesorach 64-bitowych odnoszenie się do pamięci może (w kompilatorze TASM nie jest to obsługiwane) odbywać się wg schematu:

zmienna [rej_baz + rej_ind * skala +- liczba] (tylko TASM/MASM)
lub
[zmienna + rej_baz + rej_ind * skala +- liczba]
gdzie:

- zmienna oznacza nazwę zmiennej i jest to liczba obliczana przez kompilator lub linker
- rej_baz (rejestr bazowy) = jeden z rejestrów RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, ..., R15, a nawet RIP (ale wtedy nie można użyć żadnego rejestru indeksowego)
- rej_ind (rejestr indeksowy) = jeden z rejestrów RAX, RBX, RCX, RDX, RSI, RDI, RBP, R8, ..., R15 (bez RSP i RIP)
- mnożnik (scale) = 1, 2, 4 lub 8 (gdy nie jest podany, przyjmuje się 1)

Tak, tego schematu też można używać w DOSie.

Dwie zasady:

- między nawiasami kwadratowymi nie można mieszać rejestrów różnych rozmiarów
- w trybie 64-bitowym nie można do adresowania używać rejestrów cząstkowych: R*D, R*W, R*B.

Przykłady:

```
mov     al, [ nazwa_zmiennej+2 ]
mov     [ rdi-23 ], cl
mov     dl, [ rbx + rsi*2 + nazwa_zmiennej+18 ]
mov     rax, [rax+rbx*8-34]
mov     rax, [ebx]
mov     r8d, [ecx-11223344]
mov     cx, [r8]
```

A teraz inny przykład: spróbujemy wczytać 5 elementów o numerach 1, 3, 78, 25, i 200 (pamiętajmy, że liczymy od zera) z tablicy zmienna (tej o 234 bajtach, zadeklarowanej wcześniej) do kilku rejestrów 8-bitowych. Operacja nie jest trudna i wygląda po prostu tak:

```
mov     al, [ zmienna + 1 ]
mov     ah, [ zmienna + 3 ]
mov     cl, [ zmienna + 78 ]
mov     ch, [ zmienna + 25 ]
mov     dl, [ zmienna + 200 ]
```

Oczywiście, kompilator nie sprawdzi za Was, czy takie elementy tablicy rzeczywiście istnieją - o to musicie zadbać sami.

W powyższym przykładzie rzuca się w oczy, że ciągle używamy słowa zmienna, bo wiemy, gdzie jest nasza tablica. Jeśli tego nie wiemy (dynamiczne przydzielanie pamięci), lub z innych przyczyn nie chcemy ciągle pisać zmienna, możemy posłużyć się bardziej złożonymi sposobami adresowania. Po chwili zastanowienia bez problemu stwierdzicie, że powyższy kod można bez problemu zastąpić czymś takim (i też będzie działać):

```
        mov     bx, OFFSET zmienna      ; w NASMie/FASMie: mov bx, zmienna
        mov     al, [ bx + 1 ]
        mov     ah, [ bx + 3 ]
        mov     cl, [ bx + 78 ]
        mov     ch, [ bx + 25 ]
        mov     dl, [ bx + 200 ]
```

Teraz trudniejszy przykład: spróbujmy dobrać się do kilku elementów 2-wymiarowej tablicy dwordów zadeklarowanej wcześniej (tej o rozmiarze 25 na 34). Mamy 25 wierszy po 34 elementy każdy. Aby do EAX wpisać pierwszy element pierwszego wiersza, piszemy oczywiście tylko:

```
        mov     eax, [Tablica]
```

Ale jak odczytać 23 element 17 wiersza? Otóż, sprawa nie jest taka trudna, jakby się mogło wydawać. Ogólny schemat wygląda tak (zakładam, że ostatni wskaźnik zmienia się najszybciej, potem przedostatni itd. - pamiętamy, że rozmiar elementu wynosi 4):

```
Tablica[17][23] = [ Tablica + (17*długość_wiersza + 23)*4 ]
```

No więc piszemy (użyjemy tutaj wygodniejszego adresowania 32-bitowego):

```
        mov     ebx, OFFSET Tablica     ; w NASMie/FASMie:
                                        ; MOV BX, Tablica
        mov     esi, 17
jakas_petla:
        imul    esi, 34         ; ESI = ESI * 34 =
                                ; 17 * długość wiersza
        add     esi, 23         ; ESI = ESI + 23 =
                                ; 17 * długość wiersza + 23
        mov     eax, [ ebx + esi*4 ]    ; mnożymy numer elementu
                                        ; przez rozmiar elementu
        ...
```

Można było to zrobić po prostu tak:

```
        mov     eax, [ Tablica + (17*34 + 23)*4 ]
```

ale poprzednie rozwiązanie (na rejestrach) jest wprost idealne do pętli, w której robimy coś z coraz to innym elementem tablicy.

Podobnie ((numer_wiersza*długość_wiersza1 + numer_wiersza*długość_wiersza2 + ...)*rozmiar_elementu) adresuje się tablice wielowymiarowe. Schemat jest następujący:

```
Tablica[d1][d2][d3][d4]   - 4 wymiary o długościach wierszy
                            d1, d2, d3 i d4

Tablica[i][j][k][m] = [ Tablica + (i*d2*d3*d4+j*d3*d4+k*d4+m)*
                        *rozmiar elementu ]
```

Teraz powiedzmy, że mamy taką tablicę:

```
dword tab1[24][78][13][93]
```

Aby dobrać się do elementu tab1[5][38][9][55], piszemy:

```
        mov     eax, [ tab1 + (5*78*13*93 + 38*13*93 + 9*93 + 55)*4 ]
```

Pytanie: do jakich segmentów odnosi się to całe adresowanie? Przecież mamy kilka rejestrów segmentowych, które mogą wskazywać na zupełnie co innego.
Odpowiedź:
Na rejestrach 16-bitowych obowiązują reguły:

- jeśli pierwszym rejestrem jest BP, używany jest SS
- w pozostałych przypadkach używany jest DS

Na rejestrach 32-bitowych mamy:

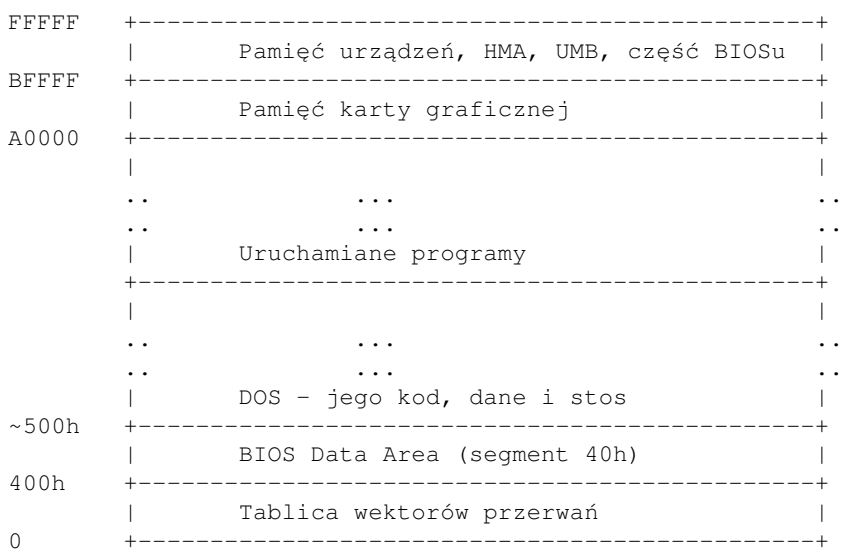
- jeśli pierwszym w kolejności rejestrem jest EBP lub ESP, używany jest SS
- w pozostałych przypadkach używany jest DS

W systemach 64-bitowych segmenty odchodzą w zapomnienie.
Domyślne ustawianie można zawsze obejść używając przedrostków, np.

```
; TASM:
        mov     ax, ss:[si]
        mov     gs:[eax+ebx*2-8], cx

; NASM/FASM:
        mov     ax, [ss:si]
        mov     [gs:eax+ebx*2-8], cx
```

Organizacja pamięci w komputerze.

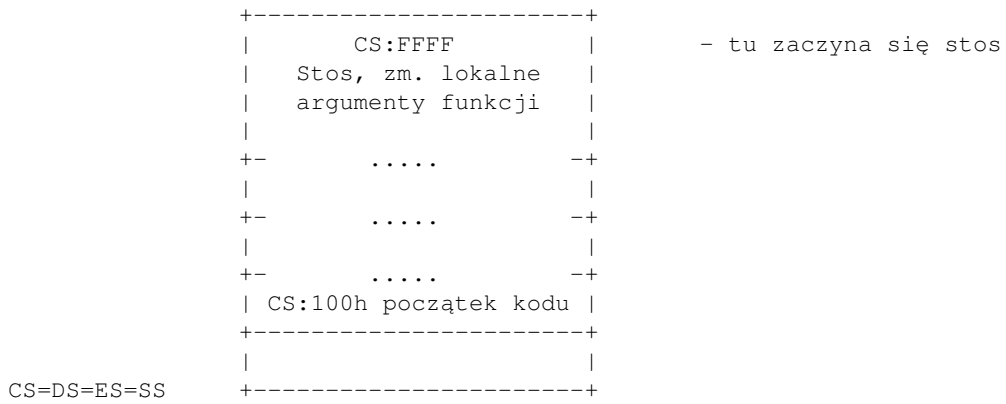
Po załadowaniu systemu DOS, pamięć wygląda z grubsza tak (niektóre elementy zostaną zaraz opisane) : (przeskocz ilustrację pamięci w DOSie)



Od segmentu A0000 zaczyna się pamięć karty graficznej. Pamięć ta jest bezpośrednim odwzorowaniem ekranu i pisząc tam, zmieniamy zawartość ekranu (więcej o tym w innych artykułach). Po przeliczeniu A0000 na system dziesiętny dostajemy 655360, czyli ... 640kB. Stąd wziął się ten sławny limit pamięci konwencjonalnej.
Powyżej znajduje się DOSowy Upper Memory Block i High Memory Area. Na samym końcu granic adresowania (czyli tuż pod 1MB) jest jeszcze skrawek BIOSu i to miejsce (a właściwie to adres FFFF:0000) jest punktem startu procesora tuż po włączeniu zasilania. W okolicach tego adresu znajduje się instrukcja skoku, która mówi procesorowi, gdzie są dalsze instrukcje.
Od adresu zero zaczyna się Tablica Wektorów Przerwań (Interrupt Vector Table, IVT), mająca 256 adresów procedur obsługi przerwań po 4 bajty (segment+offset) każdy.
Potem jest BIOS Data Area (segment 40h), powyżej - kod DOSa, a po nim miejsce na uruchamiane programy.

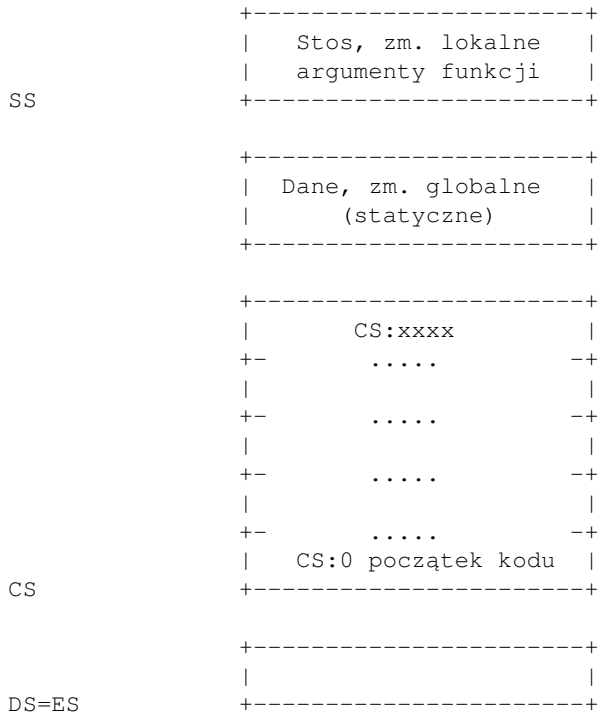
Ale chwileczkę! DOS nie może korzystać z więcej niż 1 MB pamięci? A co z EMS i XMS?
Megabajt pamięci to wszystko, co może osiągnąć procesor 16-bitowy. Procesory od 80386 w górę są co najmniej 32-bitowe, co daje łączną możliwość zaadresowania 2^32 = 4GB pamięci, o ile tylko jest tyle zainstalowane.
Menadżery EMS i XMS są to programy (napisane dla procesorów 32-bitowych), które umożliwiają innym programom dostęp do pamięci powyżej 1 MB. Sam DOS nie musi mieć aż tyle pamięci, ale inne programy mogą korzystać z dobrodziejstw większych ilości pamięci RAM. Zamiast korzystać z przerwania DOSa do rezerwacji pamięci, programy te korzystają z interfejsu udostępnianego przez np. HIMEM.SYS czy EMM386.EXE i udokumentowanego w spisie przerwań Ralfa Brown'a.

Struktura pamięci dla poszczególnych programów zależy od ich typu. Jak pamiętamy z części pierwszej, program typu .com mieści się w jednym segmencie, wykonywanie zaczyna się od adresu 100h (256. bajt), a wcześniej jest między innymi linia poleceń programu.
Wygląda to tak:
(przeskocz ilustrację pamięci programu COM)



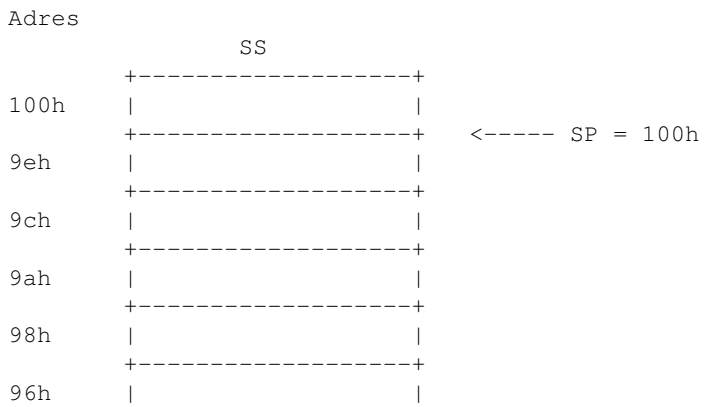
Kod zaczyna się od CS:100h, wszystkie rejestry segmentowe mają rowne wartości. Od CS:FFFF zaczyna się stos rosnący oczywiście w dół, więc pisząc taki program trzeba uważać, by ze stosem nie wejść na kod lub dane.
Programy .exe mają nieco bardziej złożoną strukturę. Kod zaczyna się pod adresem 0 w danym, wyznaczonym przez DOS, segmencie. Ale rejestry DS i ES mają inną wartość niż CS i wskazują na wspomniane przy okazji programów .com 256 bajtów zawierających linię poleceń programu itp. Dane programu, jeśli zostały umieszczone w kodzie w osobnym segmencie, też mogą dostać własny segment pamięci. Segment stosu zaś jest całkowicie oddzielony od pozostałych, zwykle za kodem. Jego położenie zależy od rozmiaru kodu i danych. Jako że programy .exe posiadają nagłówek, DOS nie musi przydzielać im całego segmentu. Zamiast tego, rozmiar segmentu kodu (i stosu) odczyta sobie z nagłówka pliku.

Graficznie wygląda to tak:
(przeskocz ilustrację pamięci programu EXE)



Stos

Przyszła pora na omówienie, czym jest stos.
Otóż, stos jest po prostu kolejnym segmentem pamięci. Są na nim umieszczane dane tymczasowe, np. adres powrotny z funkcji, jej parametry wywołania, jej zmienne lokalne. Służy też do zachowywania zawartości rejestrów.
Obsługa stosu jest jednak zupełnie inna.
Po pierwsze, stos jest budowany od góry na dół! Rysunek będzie pomocny:
(przeskocz rysunek stosu)



```
...             ....
```

Na tym rysunku SP=100h, czyli SP wskazuje na komórkę o adresie 100h w segmencie SS.
Dane na stosie umieszcza się instrukcją PUSH a zdejmuje instrukcją POP. Push jest równoważne parze pseudo-instrukcji:

```
sub     sp, ..  ; rozmiar zależy od rozmiaru obiektu w bajtach
mov     ss:[sp], ..
```

a pop:

```
mov     .., ss:[sp]
add     sp, ..
```

Tak więc, po wykonaniu instrukcji PUSH AX i PUSH DX powyższy stos będzie wyglądał tak:
(przeskocz rysunek działania stosu)

```
Stos po wykonaniu PUSH AX i PUSH DX, czyli
        sub     sp, 2
        mov     ss:[sp], ax
        sub     sp, 2
        mov     ss:[sp], dx

                SS
        +-------------------+
100h    |                   |
        +-------------------+
9eh     |       AX          |
        +-------------------+
9ch     |       DX          |
        +-------------------+   <----- SP = 9ch
...             ....
```

SP=9ch, pod [SP] znajduje się wartość DX, a pod [SP+2] - wartość AX. A po wykonaniu instrukcji POP EBX (tak, można zdjąć dane do innego rejestru, niż ten, z którego pochodziły):
(przeskocz drugi rysunek działania stosu)

```
Stos po wykonaniu POP EBX, czyli
        mov     ebx, ss:[sp]
        add     sp, 4

                SS
        +-------------------+
100h    |                   |
        +-------------------+   <----- SP = 100h
9eh     |       AX          |
        +-------------------+
9ch     |       DX          |
        +-------------------+
...             ....
```

Teraz ponownie SP=100h. Zauważcie, że dane są tylko kopiowane ze stosu, a nie z niego usuwane. Ale w żadnym przypadku nie można na nich już polegać. Dlaczego? Zobaczycie zaraz.
Najpierw bardzo ważna uwaga, która jest wnioskiem z powyższych rysunków.
Dane (które chcemy z powrotem odzyskać w niezmienionej postaci) położone na stosie instrukcją PUSH należy zdejmować kolejnymi instrukcjami POP W ODWROTNEJ KOLEJNOŚCI niż były kładzione.
Zrobienie czegoś takiego:

```
push    ax
push    dx
pop     ax
pop     dx
```

nie przywróci rejestrom ich dawnych wartości!

Przerwania i procedury a stos

Używaliśmy już instrukcji przerwania, czyli INT. Przy okazji omawiania stosu nadeszła pora, aby powiedzieć, co ta instrukcja w ogóle robi. Otóż, INT jest równoważne temu pseudo-kodowi:

```
pushf                   ; włóż na stos rejestr stanu procesora (flagi)
push    cs              ; segment, w którym aktualnie pracujemy
push    ip_next         ; adres instrukcji po INT
jmp     procedura_obslugi_przerwania
```

Każda procedura obsługi przerwania (Interrupt Service Routine, ISR) kończy się instrukcją IRET (interrupt return), która odwraca powyższy kod, czyli z ISR procesor wraca do dalszej obsługi naszego programu. Jednak oprócz instrukcji INT przerwania mogą być wywołana w inny sposób - przez sprzęt. Tutaj właśnie pojawiają się IRQ. Do urządzeń wywołujących przerwania IRQ należą między innymi karta dźwiękowa, modem, zegar, kontroler dysku twardego, itd...

Bardzo istotną rolę gra zegar, utrzymujący aktualny czas w systemie. Jak napisałem w jednym z artykułów, tyka on z częstotliwością ok. 18,2 Hz. Czyli ok. 18 razy na sekundę wykonywane są 3 PUSHe a po nich 3 POPy. Nie zapominajmy o push i pop wykonywanych w samej ISR tylko po to, aby zachować modyfikowane rejestry. Każdy PUSH zmieni to, co jest poniżej SP.

Dlatego właśnie żadne dane poniżej SP nie mogą być uznawane za wiarygodne.

Gdzie zaś znajdują się procedury obsługi przerwań?
W pamięci, pod adresami od 0000:0000 do 0000:03ff włącznie znajdują się 4-bajtowe adresy (pary CS oraz IP) odpowiednich procedur. Jest ich 256.
Pierwszy adres jest pod 0000:0000 - wskazuje on na procedurę obsługi przerwania int 0
Drugi adres jest pod 0000:0004 - int 1
Trzeci adres jest pod 0000:0008 - int 2
Czwarty adres jest pod 0000:000c - int 3
...
255-ty adres jest pod 0000:03fc - int 0FFh
W taki właśnie sposób działa mechanizm przerwań w DOSie.

Mniej skomplikowana jest instrukcja CALL, która służy do wywoływania zwykłych procedur. W zależności od rodzaju procedury (near - zwykle w tym samym pliku/programie, far - np. w innym pliku/segmencie), instrukcja CALL wykonuje takie coś:

```
push    cs              ; tylko jeśli FAR
push    ip_next         ; adres instrukcji po CALL
```

Procedura może zawierać dowolne (nawet różne ilości instrukcji PUSH i POP), ale pod koniec SP musi być taki sam, jak był na początku, czyli wskazywać na prawidłowy adres powrotu, który ze stosu jest zdejmowany instrukcją RET (lub RETF). Dlatego nieprawidłowe jest takie coś:

```
zla_procedura:
        push    ax
        push    bx
        add     ax, bx
        ret
```

gdyż w chwili wykonania instrukcji RET na wierzchu stosu jest BX, a nie adres powrotny! Błąd stosu jest przyczyną wielu trudnych do znalezienia usterek w programie.
Jak to poprawić bez zmiany sensu? Na przykład tak:

```
dobra_procedura:
        push    ax
        push    bx
        add     ax, bx
        add     sp, 4
        ret
```

Teraz już wszystko powinno być dobrze. SP wskazuje na dobry adres powrotny. Dopuszczalne jest też takie coś:
(przeskocz przykład innej dobrej procedury)

```
; TASM:
proc1   proc    near
        push    ax
        cmp     ax, 0           ; czy AX jest zerem?
        je      koniec1         ; jeśli tak, to koniec1

        pop     bx
        ret
koniec1:
        pop     cx
        ret
proc1   endp
```

(przeskocz ten sam przykład w wersji NASM i FASM)

```
; NASM/FASM:
proc1:                          ; bez PROC i NEAR
        push    ax
        cmp     ax, 0           ; czy AX jest zerem?
        je      koniec1         ; jeśli tak, to koniec1

        pop     bx
        ret
koniec1:
        pop     cx
        ret
; bez ENDP
```

SP ciągle jest dobrze ustawiony przy wyjściu z procedury mimo, iż jest 1 PUSH a 2 POP-y.
Po prostu ZAWSZE należy robić tak, aby SP wskazywał na poprawny adres powrotny, niezależnie od sposobu.

Alokacja zmiennych lokalnych procedury

Nie musi się to Wam od razu przydać, ale przy okazji stosu omówię, gdzie znajdują się zmienne lokalne funkcji (np. takich w języku C) oraz jak rezerwować na nie miejsce.

Gdy program wykonuje instrukcję CALL, na stosie umieszczany jest adres powrotny (o czym już wspomniałem). Jako że nad nim mogą być jakieś dane ważne dla programu (na przykład zachowane rejestry, inne adresy powrotne), nie wolno tam nic zapisywać. Ale pod adresem powrotnym jest dużo miejsca i to tam właśnie programy umieszczają swoje zmienne lokalne.
Samo rezerwowanie miejsca jest dość proste: liczymy, ile łącznie bajtów nam potrzeba na własne zmienne i tyle właśnie odejmujemy od rejestru SP, robiąc tym samym miejsce na stosie, które nie będzie zamazane przez instrukcje INT i CALL (gdyż one zamazują tylko to, co jest pod SP).
Na przykład, jeśli nasze zmienne zajmują 8 bajtów, to odejmujemy te 8 od SP i nasz nowy stos wygląda tak:

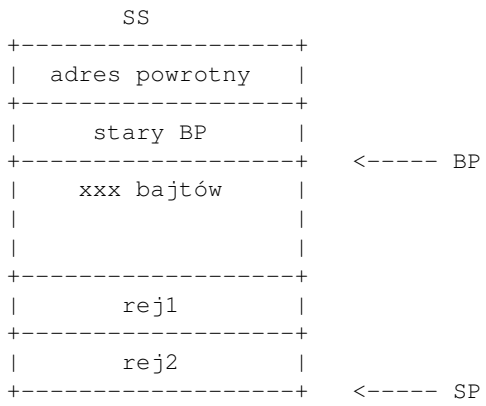
```
              SS
         +--------------------+
100h     |  adres powrotny    |
         +--------------------+   <----- stary SP = 100h
9eh      |       wolne        |
         +--------------------+
9ch      |       wolne        |
         +--------------------+
9ah      |       wolne        |
         +--------------------+
98h      |       wolne        |
         +--------------------+   <----- SP = 98h
```

SP wynosi 98h, nad nim jest 8 bajtów wolnego miejsca, po czym adres powrotny i inne stare dane.
Miejsce już mamy, korzystanie z niego jest proste - wystarczy odwoływać się do [SP], [SP+2], [SP+4], [SP+6]. Ale stanowi to pewien problem, bo po każdym wykonaniu instrukcji PUSH, te cyferki się zmieniają (bo przecież adresy się nie zmieniają, ale SP się zmienia). Dlatego właśnie do adresowania zmiennych lokalnych często używa się innego rejestru niż SP. Jako że domyślnym segmentem dla BP jest segment stosu, wybór padł właśnie na ten rejestr (oczywiście, można używać dowolnego innego, tylko trzeba dostawiać SS: z przodu, co kosztuje za każdym razem 1 bajt).

Aby móc najłatwiej dostać się do swoich zmiennych lokalnych, większość funkcji na początku zrównuje BP z SP, potem wykonuje rezerwację miejsca na zmienne lokalne, a dopiero potem - zachowywanie rejestrów itp. (czyli swoje PUSH-e). Wygląda to tak:

```
push    bp              ; zachowanie starego BP
mov     bp, sp          ; BP = SP

sub     sp, xxx         ; rezerwacja miejsca na zmienne lokalne
push    rej1            ; tu SP się zmienia, ale BP już nie
push    rej2
...

...
pop     rej2            ; tu SP znów się zmienia, a BP - nie
pop     rej1

mov     sp, bp          ; zwalnianie zmiennych lokalnych
                        ;   można też (ADD SP,xxx)
pop     bp

ret
```

Przy instrukcji `MOV SP, BP` napisałem, że zwalnia ona zmienne lokalne. Zmienne te oczywiście dalej są na stosie, ale teraz są już poniżej SP, a niedawno napisałem: żadne dane poniżej SP nie mogą być uznawane za wiarygodne.
Po pięciu pierwszych instrukcjach nasz stos wygląda tak:



Rejestr BP wskazuje na starą wartość BP, zaś SP - na ostatni element włożony na stos.
I widać teraz, że zamiast odwoływać się do zmiennych lokalnych poprzez [SP+liczba] przy ciągle zmieniającym się SP, o wiele wygodniej odwoływać się do nich przez [BP-liczba] (zauważcie: minus), bo BP pozostaje niezmienione.

Często np. w disasemblowanych programach widać instrukcje typu `AND SP, NOT 16` (lub `AND SP, ~16` w składni NASM). Jedynym celem takich instrukcji jest wyrównanie SP do pewnej pożądanej granicy, np. 16 bajtów (wtedy AND z wartością NOT 16, czyli FFFFFFF0h), żeby dostęp do zmiennych lokalnych trwał krócej. Gdy adres zmiennej np. czterobajtowej jest nieparzysty, to potrzeba dwóch dostępów do pamięci, żeby ją całą pobrać (bo można pobrać 32 bity z na raz w procesorze 32-bitowym i tylko z adresu podzielnego przez 4).

Ogół danych: adres powrotny, parametry funkcji, zmienne lokalne i zachowane rejestry nazywany jest czasem ramką stosu (ang. stack frame).
Rejestr BP jest czasem nazywany wskaźnikiem ramki, gdyż umożliwia od dostęp do wszystkich istotnych danych poprzez stałe przesunięcia (offsety, czyli te liczby dodawane i odejmowane od BP): zmienne lokalne są pod `[BP-liczba]`, parametry funkcji przekazane z zewnątrz - pod `[+liczba]`, zaś pod `[BP]` jest stara wartość EBP. Jeśli wszystkie funkcje w programie zaczynają się tym samym prologiem: `PUSH BP / MOV BP, SP`, to po wykonaniu instrukcji `MOV BP, [BP]` w BP znajdzie się wskaźnik ramki ... procedury wywołującej. Jeśli znamy jej strukturę, można w ten sposób dostać się do jej zmiennych lokalnych.

Zainteresowanych szczegółami adresowania lub instrukcjami odsyłam do Intela lub AMD

Następnym razem o podstawowych instrukcjach języka asembler.

- Ilu programistów potrzeba, aby wymienić żarówkę?
- Ani jednego. To wygląda na problem sprzętowy.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia:

1. Zadeklaruj tablicę 12 zmiennych mających po 10 bajtów:
    1. zainicjalizowaną na zera (pamiętaj o ograniczeniach kompilatora)
    2. niezainicjalizowaną

2. Zadeklaruj tablicę 12 słów (16-bitowych) o wartości BB (szesnastkowo), po czym do każdego z tych słów wpisz wartość FF szesnastkowo (bez żadnych pętli). Można (a nawet trzeba) użyć więcej niż 1 instrukcji. Pamiętaj o odległościach między poszczególnymi elementami tablicy. Naucz się różnych sposobów adresowania: liczba (nazwa zmiennej + numer), baza (rejestr bazowy + liczba), baza + indeks (rejestr bazowy + rejestr indeksowy).

3. Zadeklaruj dwuwymiarową tablicę bajtów o wartości 0 o wymiarach 13 wierszy na 5 kolumn, po czym do elementu numer 3 (przedostatni) w wierszu o numerze 12 (ostatni) wpisz wartość FF. Spróbuj użyć różnych sposobów adresowania.

Jak pisać programy w języku asembler?
Część 3 - Podstawowe instrukcje, czyli poznajemy dialekt procesora.

Poznaliśmy już rejestry, omówiliśmy pamięć. Pora zacząć na nich operować. Zanim zaczniemy, proszę Was o to, abyście tej listy też NIE uczyli się na pamięć. Instrukcji jest dużo, a próba zrozumienia ich wszystkich na raz może spowodować niezły chaos. Co najwyżej przejrzyjcie tą listę kilka razy, aby wiedzieć mniej-więcej, co każda instrukcja robi.

Instrukcje procesora można podzielić na kilka grup:

- instrukcje przemieszczania danych
- instrukcje arytmetyki binarnej
- instrukcje arytmetyki dziesiętnej
- instrukcje logiczne
- operacje na bitach i bajtach
- instrukcje przekazujące kontrolę innej części programu (sterujące wykonywaniem programu)
- instrukcje operujące na łańcuchach znaków
- instrukcje kontroli flag
- instrukcje rejestrów segmentowych
- inne

Zacznijmy je po kolei omawiać (nie omówię wszystkich).

1. instrukcje przemieszczania danych.
   Tutaj zaliczymy już wielokrotnie używane `MOV` oraz kilka innych: `XCHG` , `PUSH` i `POP`.

2. arytmetyka binarna.
   `add do_czego,co` - dodaj
   `sub od_czego,co` - odejmij
   `inc coś / dec coś` - zwiększ/zmniejsz coś o 1
   `cmp co, z_czym` - porównaj. Wykonuje działanie odejmowania co minus z_czym, ale nie zachowuje wyniku, tylko ustawia flagi.
   Wynikiem może być ustawienie lub wyzerowanie jednej lub więcej flag - zaznaczenie wystąpienia jednego z warunków. Główne warunki to:
   - A - above - ponad (dla liczb traktowanych jako liczby bez znaku): co > z_czym
     (przeskocz przykład użycia warunku A)

     ```
     cmp al,bl
     ja al_wieksze_od_bl     ; ja -  jump if above
     ```
   - B - below - poniżej (bez znaku): co < z_czym
   - G - greater - więcej niż (ze znakiem): co > z_czym
   - L - lower - mniej niż (ze znakiem): co < z_czym
   - O - overflow - przepełnienie (ze znakiem, np. przebicie 32767 w górę) ostatniej operacji. Niekoniecznie używane przy cmp.
   - C - carry - przepełnienie (bez znaku, czyli przebicie np. 65535 w górę)
     (przeskocz przykład użycia warunku C)

     ```
     add al,bl
     jc blad_przepelnienia   ; jc -  jump if carry
     ```
   - E lub Z - equal (równy) lub zero. Te 2 warunki są równoważne.

(przeskocz przykłady użycia warunków równości)

```
cmp ax,cx
je ax_rowne_cx
...
sub bx,dx
jz bx_rowne_dx
```

- NE/NZ - przeciwieństwo poprzedniego: not equal/not zero.
- NA - not above, czyli nie ponad - mniejsze lub równe (ale dla liczb bez znaku)
- NB - not below, czyli nie poniżej - większe lub równe (dla liczb bez znaku)
- NG - not greater, czyli nie więcej - mniejsze lub równe (ale dla liczb ze znakiem)
- NL - not lower, czyli nie mniej - większe lub równe (dla liczb ze znakiem)
- NC - no carry
- AE/BE - above or equal (ponad lub równe), below or equal (poniżej lub równe)
- NO - no overflow

3. arytmetyka dziesiętna
   - NEG - zmienia znak.
   - MUL, IMUL - mnożenie, mnożenie ze znakiem (czyli uwzględnia liczby ujemne)
     (przeskocz przykłady instrukcji mnożenia)

```
mul cl          ; AX = AL*CL
mul bx          ; DX:AX = AX*BX
mul esi         ; EDX:EAX = EAX*ESI
mul rdi         ; RDX:RAX = RAX*RDI

imul eax        ; EDX:EAX = EAX*EAX
imul ebx,ecx,2  ; EBX = ECX*2
imul ebx,ecx    ; EBX = EBX*ECX
imul si,5       ; SI = SI*5
```

   - DIV, IDIV - dzielenie, dzielenie ze znakiem.
     (przeskocz przykłady instrukcji dzielenia)

```
div cl  ; AL = (AX div CL), AH = (AX mod CL)
div bx  ; AX = (DX:AX div BX),
        ; DX = (DX:AX mod BX)
div edi ; EAX = (EDX:EAX div EDI),
        ; EDX = (EDX:EAX mod EDI)
div rsi ; RAX = (RDX:RAX div RSI),
        ; RDX = (RDX:RAX mod RSI)
```

4. Instrukcje bitowe (logiczne). AND, OR, XOR, NOT, TEST. Instrukcja TEST działa tak samo jak AND z tym, że nie zachowuje nigdzie wyniku, tylko ustawia flagi. Po krótce wytłumaczę te instrukcje:

   (przeskocz działanie instrukcji logicznych)

```
0 AND 0 = 0     0 OR 0 = 0      0 XOR 0 = 0
0 AND 1 = 0     0 OR 1 = 1      0 XOR 1 = 1
1 AND 0 = 0     1 OR 0 = 1      1 XOR 0 = 1
1 AND 1 = 1     1 OR 1 = 1      1 XOR 1 = 0
NOT 0 = 1
NOT 1 = 0
```

   Przykłady zastosowania:
   (przeskocz przykłady instrukcji logicznych)

```
and ax,1        ; wyzeruje wszystkie bity z
                ; wyjątkiem bitu numer 0.
or ebx,1111b    ; ustawia (włącza) 4 dolne bity.
                ; Reszta bez zmian.
xor cx,cx       ; CX = 0
not dh          ; DH ma 0 tam, gdzie miał 1
                ; i na odwrót
```

5. Instrukcje przesunięcia bitów.
    1. SAL, SHL - shift left.
       bit7 = bit6, bit6 = bit5, ... , bit1 = bit0, bit0 = 0.

    2. SHR - shift logical right
       bit0 = bit1, bit1 = bit2, ... , bit6 = bit7, bit7 = 0

    3. SAR - shift arithmetic right
       bit0 = bit1, bit1 = bit2, ... , bit6 = bit7, bit7 = bit7 (bit znaku zachowany!)
       Najstarszy bit w rejestrze nazywa się czasem właśnie bitem znaku.

    4. ROL - rotate left
       bit7 = bit6, ... , bit1 = bit0, bit0 = stary bit7

    5. RCL - rotate through carry left
       carry flag CF = bit7, bit7 = bit6, ... , bit1 = bit0, bit0 = stara CF

    6. ROR - rotate right
       bit0 = bit1, ... , bit6 = bit7, bit7 = stary bit0

    7. RCR - rotate through carry right
       CF = bit0, bit0 = bit1, ... , bit6 = bit7, bit7 = stara CF

   Przy użyciu SHL można przeprowadzać szybkie mnożenie, a dzięki SHR - szybkie dzielenie. Np. SHL AX,1 jest równoważne przemnożeniu AX przez 2, SHL AX,5 - przez 2^5 = 32. SHR BX,4 dzieli bx przez 16.

6. Instrukcje sterujące wykonywaniem programu.
    - Skoki warunkowe (patrz: warunki powyżej): JA=JNBE, JAE=JNB, JNA=JBE, JNAE=JB , JG=JNLE (jump if greater - dla liczb ze znakiem) = jump if not lower or equal, JNG=JLE, JGE=JNL, JNGE=JL, JO, JNO, JC, JNC, JS (jump if sign czyli bit7 wyniku jest równy 1), JNS, JP=JPE (jump if parity equal = liczba bitów równych jeden jest parzysta), JNP=JPO.
    - Skoki bezwarunkowe: JMP, JMP SHORT, JMP FAR
    - Uruchomienia procedur: CALL [NEAR/FAR]
    - Powrót z procedury: RET/RETF.
    - Przerwania: INT, INTO (wywołuje przerwanie INT4 w razie przepełnienia), BOUND (int 5)
    - Instrukcje pętli: LOOP. Składnia: LOOP gdzieś. Jeśli CX jest różny od 0, to skacz do gdzieś.

7. Operacje na łańcuchach znaków.
    1. LODS[B/W/D/Q] - Load Byte/Word/Dword/Qword
       MOV AL/AX/EAX/RAX , DS:[SI/ESI/RSI]
       ADD SI,1/2/4/8 ; ADD, gdy flaga kierunku DF = 0, SUB gdy DF = 1

    2. STOS[B/W/D/Q] - Store Byte/Word/Dword/Qword
       MOV ES:[DI/EDI/RDI], AL/AX/EAX/RAX

ADD DI,1/2/4/8 ; ADD/SUB jak wyżej

3. MOVS[B/W/D/Q] - Move Byte/Word/Dword/Qword
   MOV ES:[DI/EDI/RDI], DS:[SI/ESI/RSI] ; to nie jest instrukcja!
   ADD DI,1/2/4/8 ; ADD/SUB jak wyżej
   ADD SI,1/2/4/8

4. CMPS[B/W/D/Q] - Compare Byte/Word/Dword/Qword
   CMP DS:[SI/ESI/RSI], ES:[DI/EDI/RDI] ; to nie jest instrukcja!
   ADD SI,1/2/4/8 ; ADD/SUB jak wyżej
   ADD DI,1/2/4/8

5. SCAS[B/W/D/Q] - Scan Byte/Word/Dword/Qword
   skanuje łańcuch bajtów/słów/podwójnych słów/poczwórnych słów pod ES:[DI/EDI/RDI] w poszukiwaniu, czy jest tam wartość wskazana przez AL/AX/EAX/RAX.

Do każdej z powyższych instrukcji można z przodu dodać przedrostek REP (repeat), co spowoduje, że będzie ona wykonywana, aż CX stanie się zerem, lub REPE/REPZ lub REPNE/REPNZ co spowoduje, że będzie ona wykonywana, dopóty CX nie jest zerem i jednocześnie ZF (zero flag) =1 lub =0, odpowiednio.

8. Instrukcje wejścia/wyjścia do portów.
   Są one bardziej szczegółowo opisane w części poświęconej portom, ale podam tu skrót:
   - IN
     IN AL/AX/EAX, port/DX.
     Pobierz z portu 1/2/4 bajty i włóż do AL/AX/EAX (od najmłodszego). Jeśli numer portu jest mniejszy lub równy 255, można go podać bezpośrednio. Jeśli większy - trzeba użyć DX.

   - OUT
     OUT port/DX, AL/AX/EAX.
     Uwagi jak przy instrukcji IN.

9. Instrukcje flag
   - STC/CLC - set carry / clear carry. Do flagi CF wstaw 1 lub 0, odpowiednio.

   - STD/CLD. Ustaw DF = 1, DF = 0, odpowiednio.

   - STI/CLI. Interrupt Flag IF := 1, IF := 0, odpowiednio. Gdy IF=0, przerwania sprzętowe są blokowane.

   - Przenoszenie flag
     PUSHF / PUSHFD / PUSHFQ - umieść flagi na stosie (16, 32 i 64 bity flag, odpowiednio)
     POPF / POPFD / POPFQ - zdejmij flagi ze stosu (16/32/64 bity flag)
     SAHF / LAHF - zapisz AH w pierwszych 8 bitach flag / zapisz pierwsze 8 bitów flag w AH.

10. Instrukcja LEA - Load Effective Address.
    Wykonanie:

```
lea     rej, [pamięć]
```

jest równoważne:
(przeskocz pseudo-kod LEA)

```
mov     rej, offset pamięć      ; TASM
mov     rej, pamięć             ; NASM/FASM
```

Po co więc osobna instrukcja? Otóż, LEA przydaje sie w wielu sytuacjach do obliczania złożonych adresów. Kilka przykładów:

1. Jak w 1 instrukcji sprawić, że EAX = EBP-12 ?
   Odpowiedź: `lea eax, [ebp-12]`
2. Niech EBX wskazuje na tablicę o 20 elementach o rozmiarze 8 każdy. Jak do ECX zapisać adres 11-tego elementu, a do EDX elementu o numerze EDI?
   Odpowiedź: `lea ecx, [ebx + 11*8 ]` oraz `lea edx,[ebx+edi*8]`
3. Jak w 1 instrukcji sprawić, że ESI = EAX*9?
   Odpowiedź: `lea esi, [eax + eax*8 ]`

Pominąłem mniej ważne instrukcje operujące na rejestrach segmentowych i klika innych instrukcji. Te, które tu podałem, wystarczają absolutnie na napisanie większości programów, które można zrobić.

Wszystkie informacje przedstawione w tej części pochodzą z tego samego źródła: podręczniki Intela i podręczniki AMD

Byle głupiec potrafi napisać kod, który zrozumie komputer. Dobry programista pisze taki kod, który zrozumie człowiek.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia:

1. Zapisz instrukcje: do rejestru AX dodaj 5, o rejestru SI odejmij 178.

2. Nie używając cyfry jeden napisz jedną instrukcję, która zmniejszy rejestr DX o jeden.

3. Przemnóż wartość rejestru EDI przez 2 na przynajmniej dwa różne sposoby po jednej instrukcji. Postaraj się nie używać instrukcji (I)MUL.

4. W jednej instrukcji podziel wartość rejestru BP przez 8.

5. Nie używając instrukcji MOV spraw, by DX miał wartość 0 (na przynajmniej 3 sposoby, każdy po jednej instrukcji).

6. Nie używając instrukcji przesuwania bitów SH* ani mnożenia *MUL przemnóż EBX przez 8. Możesz użyć więcej niż 1 instrukcji.

7. W dwóch instrukcjach spraw, by EDI równał się 7*ECX. Postaraj się nie używać instrukcji (I)MUL.

Jak pisać programy w języku asembler?
Część 4 - Pierwsze programy, czyli przełamywanie pierwszych lodów.

Znamy już rejestry, trochę instrukcji i zasad. No ale teoria jest niczym bez praktyki. Dlatego w tej części przedstawię kilka względnie prostych programów, które powinny rozbudzić wyobraźnię tworzenia.

Ten program spyta się użytkownika o imię i przywita się z nim:
(przeskocz program pytający o imię)

```
; Program witający się z użytkownikiem po imieniu
;
; Autor: Bogdan D.
; kontakt: bogdandr (at) op (dot) pl
;
; kompilacja:
; nasm -f bin -o czesc.com czesc.asm
;
; kompilacja FASM:
; fasm czesc.asm czesc.com

org 100h

        mov     ah, 9           ; funkcja wyświetlania na ekran
        mov     dx, jak_masz    ; co wyświetlić
        int     21h             ; wyświetl

        mov     ah, 0ah ; funkcja pobierania danych z klawiatury
        mov     dx, imie        ; bufor na dane
        int     21h             ; pobierz dane

        mov     ah, 9
        mov     dx, czesc
        int     21h             ; wyświetl napis "Cześć"

        mov     ah, 9
        mov     dx, imie+2      ; adres wpisanych danych
        int     21h             ; wyświetl wpisane dane

        mov     ax, 4c00h
        int     21h

jak_masz        db      "Jak masz na imie? $"
imie    db 20           ; maksymalna ilość znaków do pobrania
        db 0            ; tu dostaniemy, ile znaków pobrano
        times 22 db "$" ; miejsce na dane

czesc           db      10, 13, 10, 13, "Czesc $"
```

Powyższy program korzysta z jeszcze nieomówionej funkcji numer 10 (0Ah) przerwania DOSa. Oto jej opis z listy przerwań Ralfa Brown'a - RBIL:
(przeskocz opis int 21h, ah=0ah)

```
INT 21 - DOS 1+ - BUFFERED INPUT
        AH = 0Ah
        DS:DX -> buffer (see #01344)
Return: buffer filled with user input

Format of DOS input buffer:
```

```
Offset  Size    Description     (Table 01344)
 00h    BYTE    maximum characters buffer can hold
 01h    BYTE    (call) number of chars from last input which
                        may be recalled
                (ret) number of characters actually read,
                        excluding CR
 02h  N BYTEs   actual characters read, including the
                        final carriage return
```

Jak widać, korzystanie z niej nie jest trudne. Wystarczy stworzyć tablicę bajtów na znaki czytane z klawiatury. Na początku tablicy podajemy, ile maksymalnie znaków chcemy wczytać. Drugi bajt ustawiamy na zero, by czytać tylko na bieżąco wprowadzane znaki, a nie to, co jeszcze może tkwić w DOS-owym buforze wejściowym.

Kolejny program wypisuje na ekranie rejestr flag w postaci dwójkowej. Zanim mu się przyjrzymy, potrzebna będzie nam informacja o funkcji 0Eh przerwania 10h (opis bierzemy oczywiście z RBIL):

(przeskocz opis int 10h, ah=0eh)

```
INT 10 - VIDEO - TELETYPE OUTPUT
        AH = 0Eh
        AL = character to write
        BH = page number
        BL = foreground color (graphics modes only)
Return: nothing
Desc:   display a character on the screen, advancing the
         cursor and scrolling the screen as necessary
Notes:  characters 07h (BEL), 08h (BS), 0Ah (LF), and 0Dh (CR)
         are interpreted and do the expected things
```

Dla nas zawartość BX nie będzie istotna. A ta funkcja po prostu wypisuje na ekran jakiś znak. No, teraz wreszcie możemy przejść do programu. Oto on (flagi.asm):
(przeskocz program wypisujący flagi)

```
; Program wypisujący flagi w postaci dwójkowej
;
; Autor: Bogdan D.
; kontakt:  bogdandr (at) op (dot) pl
;
; kompilacja:
; tasm flagi.asm
; tlink flagi.obj /t


.model tiny             ; to będzie mały program
.code                   ; tu zaczyna się segment kodu
.386                    ; będziemy tu używać rejestrów 32-bitowych.
                        ; .386 MUSI być po .code !

org 100h                ; to będzie program typu .com

main:                   ; etykieta jest dowolna, byleby zgadzała się
                        ; z tą na końcu

        pushfd          ; 32 bity flag idą na stos
```

```
        mov ax,0e30h    ; AH = 0eh, czyli funkcja wyświetlania,
                        ; AL = 30h = kod ASCII cyfry zero

        pop esi         ; flagi ze stosu do ESI

        mov cx,32       ; tyle bitów i tyle razy trzeba przejść
                        ; przez pętlę

petla:                  ; etykieta oznaczająca początek pętli.

        and al,30h      ; upewniamy się, że AL zawiera tylko 30h,
                        ; co zaraz się może zmienić. A dokładniej,
                        ; czyścimy bity 0-3, z których bit 0 może
                        ; się zaraz zmienić

        shl esi,1       ; Przesuwamy bity w ESI o 1 w lewo.
                        ; 31 bit ESI idzie
                        ; do Carry Flag (CF)

        adc al,0        ; ADC - add with carry. Do AL dodaj
                        ; 0 + wartość CF.

                        ; jeśli CF (czyli 31 bit ESI) = 1,
                        ; to AL := AL+1, inaczej AL bez zmian
        int 10h         ; funkcja 0e, wyświetl znak w AL,
                        ; czyli albo zero (30h) albo jedynkę (31h)

        loop petla      ; przejdź na początek pętli,
                        ; jeśli nie skończyliśmy

        mov ah,4ch      ; funkcja wyjścia do DOS
        int 21h         ; wychodzimy

end main                ; koniec programu. Ta sama etykieta, co na początku.
```

Wersje NASM i FASM:
(przeskocz wersję NASM/FASM programu)

```
; Program wypisujący flagi w postaci dwójkowej
;
; Autor: Bogdan D.
; kontakt:  bogdandr (at) op (dot) pl
;
; kompilacja NASM:
;   nasm -o flagi.com -f bin flagi.asm
;
; kompilacja FASM:
;   fasm flagi.asm flagi.com


org 100h                ; to będzie program typu .com

main:                   ; etykieta dowolna, nawet niepotrzebna

        pushfd          ; 32 bity flag idą na stos

        mov ax,0e30h    ; AH = 0eh, czyli funkcja wyświetlania,
                        ; AL = 30h = kod ASCII cyfry zero

        pop esi         ; flagi ze stosu do ESI
```

```
        mov cx,32       ; tyle bitów i tyle razy trzeba przejść
                        ; przez pętlę

petla:                  ; etykieta oznaczająca początek pętli.

        and al,30h      ; upewniamy się, że AL zawiera tylko 30h,
                        ; co zaraz się może zmienić. A dokładniej,
                        ; czyścimy bity 0-3, z których bit 0
                        ; może się zaraz zmienić

        shl esi,1       ; Przesuwamy bity w ESI o 1 w lewo.
                        ; 31 bit ESI idzie do flagi CF

        adc al,0        ; ADC - add with carry. Do AL dodaj
                        ; 0 + wartość CF.
                        ; jeśli CF (czyli 31 bit ESI) = 1,
                        ; to AL := AL+1, inaczej AL bez zmian

        int 10h         ; funkcja 0e, wyświetl znak w AL,
                        ; czyli albo zero (30h) albo jedynkę (31h)

        loop petla      ; przejdź na początek pętli,
                        ; jeśli nie skończyliśmy

        mov ah,4ch      ; funkcja wyjścia do DOS
        int 21h         ; wychodzimy
```

Kompilujemy go następująco (wszystkie programy będziemy tak kompilować, chyba że powiem inaczej):

```
        tasm flagi.asm
        tlink flagi.obj /t
```

Lub, dla NASMa:

```
        nasm -o flagi.com -f bin flagi.asm
```

Lub, dla FASMa:

```
        fasm flagi.asm flagi.com
```

Nie ma w tym programie wielkiej filozofii. Po prostu 25 bajtów radości...
Dociekliwy zapyta, z jakim kodem wyjścia wychodzi ten program. Odpowiedź brzmi oczywiście:
- Albo 30h albo 31h, w zależności od ostatniego bitu oryginalnych flag.

Teraz krótki programik, którego jedynym celem jest wyświetlenie na ekranie cyfr od 0 do 9, każda w osobnej linii:
(przeskocz program wyświetlający cyfry)

```
; tylko wersja NASM/FASM
;
; Program wypisuje na ekranie cyfry od 0 do 9
;
; kompilacja NASM:
;   nasm -O999 -o cyfry.com -f bin cyfry.asm
; kompilacja FASM:
;   fasm cyfry.asm cyfry.com
```

```
; definiujemy stałe:

%define         lf      10      ; Line Feed
%define         cr      13      ; Carriage Return

; stałe w wersji FASM:
;       lf = 10
;       cr = 13


org 100h                        ; robimy program typu .com

        mov     eax, 0          ; pierwsza wypisywana cyfra

wyswietlaj:
        call    _pisz_ld        ; uruchom procedurę wyświetlania
                                ; liczby w EAX
        call    _nwln           ; uruchom procedurę, która przechodzi
                                ; do nowej linii
        add     eax, 1          ; zwiększamy cyfrę
        cmp     eax, 10         ; sprawdzamy, czy ciągle EAX < 10
        jb      wyswietlaj      ; jeśli EAX < 10, to wyświetlamy
                                ; cyfrę na ekranie

        mov     ax, 4c00h       ; funkcja wyjścia z programu
        int     21h             ; wychodzimy



; następujące procedury (wyświetlanie liczby i przechodzenie
; do nowego wiersza) nie są aż tak istotne, aby omawiać je
; szczegółowo, gdyż w przyszłości będziemy używać tych samych
; procedur, ale z biblioteki, a te wstawiłem tutaj dla
; uproszczenia kompilacji programu.

; Ogólny schemat działania tej procedury wygląda tak:
; weźmy liczbę EAX=12345. Robimy tak:
; 1. dzielimy EAX przez 10. reszta = EDX = DL = 5.
; Zapisz do bufora. EAX = 1234 (iloraz)
; 2. dzielimy EAX przez 10. reszta = DL = 4.
; Zapisz do bufora. EAX=123 (iloraz)
; 3. dzielimy EAX przez 10. reszta = DL = 3.
; Zapisz do bufora. EAX=12
; 4. dziel EAX przez 10. DL = 2. zapisz. iloraz = EAX = 1
; 5. dziel EAX przez 10. DL = 1. zapisz. iloraz = EAX = 0.
; Przerywamy pętlę.
; Teraz w buforze są znaki: 54321. Wystarczy wypisać
; wspak i oryginalna liczba pojawia się na ekranie.


_pisz_ld:

;pisz32e
;we: EAX=liczba bez znaku do wypisania

        pushfd                  ; zachowujemy modyfikowane rejestry
        push    ecx
        push    edx
        push    eax
        push    esi

        xor si,si               ; SI będzie wskaźnikiem do miejsca,
```

```
                                ; gdzie przechowujemy cyfry.
                                ; Teraz SI=0.

        mov ecx,10              ; liczba, przez która będziemy dzielić

_pisz_ld_petla:
        xor edx,edx             ; wyzeruj EDX, bo instrukcja DIV
                                ; go używa
        div ecx                 ; dzielimy EAX przez 10

        mov [_pisz_bufor+si],dl ; do bufora idą reszty z dzielenia
                                ; przez 10, czyli cyfry wspak

        inc si                  ; zwiększ wskaźnik na wolne miejsce.
                                ; Przy okazji, SI jest też ilością
                                ; cyfr w buforze

        or eax,eax              ; sprawdzamy, czy liczba =0
        jnz _pisz_ld_petla      ; jeśli nie, to dalej ją dzielimy
                                ; przez 10

        mov ah,0eh              ; funkcja wypisywania
_pisz_ld_wypis:
        mov al,[_pisz_bufor+si-1]       ; wypisujemy reszty wspak
        or al,"0"       ; z wartości 0-9 zrobimy cyfrę "0"-"9"
        int 10h                 ; wypisujemy cyfrę

        dec si                  ; przechodzimy na wcześniejszą cyfrę
        jnz _pisz_ld_wypis      ; jeśli SI=0, to nie ma już cyfr

        pop esi                 ; przywracamy zmienione rejestry
        pop eax
        pop edx
        pop ecx
        popfd

        ret                     ; powrót z procedury

_pisz_bufor:    times   40      db 0    ; miejsce na 40 cyferek (bajtów)

_nwln:

;pisze znak końca linii (Windows)

        push ax
        push bp
        mov ax,(0eh << 8) | lf          ; AX = 0e0ah
        int 10h                         ; wyświetlamy znak LF
        mov al,cr
        int 10h                         ; wyświetlamy znak CR
        pop bp
        pop ax
        ret
```

Następny twór nie jest wolno stojącym programem, ale pewną procedurą. Pobiera ona informacje z rejestru AL i wypisuje, co trzeba. Oto ona:
(przeskocz procedurę _pisz_ch)

```
_pisz_ch:
```

```
;we: AL=cyfra heksadecymalna do wypisania 0...15
; CF=1 jeśli błąd

        push bp         ; zachowaj modyfikowane rejestry: BP, AX, Flagi
        push ax
        pushf

        cmp al,9                ; Sprawdzamy dane wejściowe : AL jest w
                                ; 0-9 czy w 10-15?
        ja _ch_hex              ; AL > 9. Skok do _ch_hex
        or al,30h               ; 0 < AL < 9. Or ustawia 2 bity,
                                ; czyniąc z AL liczbę z
                                ; przedziału 30h - 39h, czyli od "0"
                                ; do "9". Można było napisać
                                ; "ADD al,30h", ale zdecydowałem się
                                ; na OR, bo jest szybsze a efekt ten sam.

        jmp short _ch_pz        ; AL już poprawione. Skacz do miejsca,
                                ; gdzie wypisujemy znak.

_ch_hex:                        ; AL > 9. Może będzie to cyfra hex,
                                ; może nie.
        cmp al,15               ; AL > 15?
        ja _blad_ch             ; jeśli tak, to mamy błąd
        add al,"A"-10   ; Duży skok myślowy. Ale wystarczy to rozbić
                                ; na 2 kroki i wszystko staje się jasne.
                                ; Najpierw odejmujemy 10 od AL. Zamiast
                                ; liczby od 10 do 15 mamy już liczbę
                                ; od 0 do 5. Teraz tą liczbę dodajemy do
                                ; "A", czyli kodu ASCII litery A,
                                ; otrzymując znak od "A" do "F"

_ch_pz:                         ; miejsce wypisywania znaków.
        mov ah,0eh              ; numer funkcji: 0Eh
        int 10h                 ; wypisz znak

        popf                    ; zdejmij ze stosu flagi
        clc                     ; CF := 0 dla zaznaczenia braku błędu
                                ; (patrz opis procedury powyżej)
        jmp short _ch_ok        ; skok do wyjścia

_blad_ch:                       ; sekcja obsługi błędu (AL > 15)
        popf                    ; zdejmij ze stosu flagi
        stc                     ; CF := 1 na znak błędu

_ch_ok:                         ; miejsce wyjścia z procedury
        pop ax                  ; zdejmij modyfikowane rejestry
        pop bp

        ret                     ; return, powrót
```

To chyba nie było zbyt trudne, co?
Szczegóły dotyczące pisania procedur (i bibliotek) znajdują się w moim innym artykule.

Teraz pokażę pewien program, który wybrałem ze względu na dużą ilość różnych instrukcji i sztuczek. Niestety, nie jest on krótki. Ale wspólnie spróbujemy przez niego przejść. Jest to wersja dla TASMa, ale obok instrukcji postaram się zamieścić ich NASMowe odpowiedniki. Oto on:
(przeskocz program zliczajacy liczby pierwsze)

```
; Program liczy liczby pierwsze w przedziałach
; 2-10, 2-100, 2-1000,... 2-100.000
;
; Autor: Bogdan D.
; kontakt: bogdandr (at) op (dot) pl
;
; kompilacja TASM:
;   tasm ile_pier.asm
;   tlink ile_pier.obj /t
;
; kompilacja NASM:
;   nasm -f bin -o ile_pier.com ile_pier.asm
;
; kompilacja FASM:
;   fasm ile_pier.asm ile_pier.com


.model tiny             ; to będzie mały program. NASM/FASM: usunąć.
.code                   ; początek segmentu kodu. NASM:
                        ; "section .text" lub nic. FASM: nic
.386                    ; będziemy używać rejestrów 32-bitowych.
                        ; NASM: "CPU 386" lub nic, FASM: nic

org 100h                ; to będzie program typu .com

start:                  ; początek...

        xor ebx,ebx     ; EBX = liczba, którą sprawdzamy, czy jest
                        ; pierwsza. Zaczniemy od 3. Poniżej jest 3
                        ; razy INC (zwiększ o 1). Najpierw EBX = 0,
                        ; bo "XOR rej,rej" zeruje dany rejestr.

        xor edi,edi     ; EDI = bieżący licznik liczb pierwszych

        xor ecx,ecx     ; ECX = stary licznik liczb (z poprzedniego
                        ; przedziału).
                        ; Chwilowo, oczywiście 0.

        inc ebx         ; EBX = 1

        mov esi,10      ; ESI = bieżący koniec przedziału: 10, 100, ..

        inc edi         ; EDI = 1. uwzględniamy 2, która jest
                        ; liczbą pierwszą

        inc ebx         ; EBX = 2, pierwsza liczba będzie = 3

petla:                  ; pętla przedziału

        cmp ebx,esi     ; czy koniec przedziału? (ebx=liczba,
                        ; esi=koniec przedziału)
        jae pisz        ; EBX >= ESI - idź do sekcji wypisywania
                        ; wyników

        mov ebp,2       ; EBP - liczby, przez które będziemy dzielić.
                        ; pierwszy dzielnik = 2

        inc ebx         ; zwiększamy liczbę. EBX=3. Będzie to
                        ; pierwsza sprawdzana.

spr:                    ; pętla sprawdzania pojedynczej liczby
```

```
        mov eax,ebx     ; EAX = sprawdzana liczba
        xor edx,edx     ; EDX = 0
        div ebp         ; EAX = EAX/EBP (EDX było=0),
                        ; EDX=reszta z dzielenia

        or edx,edx      ; instrukcja OR tak jak wiele innych,
                        ; ustawi flagę zera ZF na 1, gdy jej wynik
                        ; był zerem. W tym przypadku pytamy:
                        ; czy EDX jest zerem?

        jz petla        ; jeżeli dzieli się bez reszty (reszta=EDX=0),
                        ; to nie jest liczbą pierwszą
                        ; i należy zwiększyć liczbę sprawdzaną
                        ; (inc ebx)

        inc ebp         ; zwiększamy dzielnik

        cmp ebp,ebx     ; dzielniki aż do liczby
        jb spr          ; liczba > dzielnik - sprawdzaj dalej tę
                        ; liczbę. Wiem, że można było sprawdzać tylko
                        ; do SQRT(liczba) lub LICZBA/2, ale
                        ; wydłużyłoby to program i brakowało mi już
                        ; rejestrów...

juz:                    ; przerobiliśmy wszystkie dzielniki,
                        ; zawsze wychodziła reszta,
                        ; więc liczba badana jest pierwsza

        inc edi         ; zwiększamy licznik liczb znalezionych
        jmp short petla ; sprawdzaj kolejną liczbę aż do końca
                        ; przedziału.

                        ; sekcja wypisywania informacji

pisz:
        mov edx,offset przedzial        ; NASM/FASM: bez "offset"
        mov ah,9
        int 21h         ; wypisujemy napis "Przedział 2-...."

        mov eax,esi     ; EAX=ESI=koniec przedziału
        call _pisz_ld   ; wypisz ten koniec (EAX)


; NASM: mov ax,(0eh << 8) | ":" ; << to shift left, | to logiczne OR

        mov ax,(0eh shl 8) or ":" ; to wygląda zbyt skomplikowanie,
                        ; ale jest o dziwo prawidłową instrukcją.
                        ; Jest tak dlatego, że wyrażenie z prawej
                        ; strony jest obliczane przez kompilator.
                        ; 0eh przesunięte w lewo o 8 miejsc daje
                        ; 0E00 w AX. Dalej, dopisujemy do tego
                        ; dwukropek, którego kod ASCII nas nie
                        ; interesuje a będzie obliczony przez
                        ; kompilator. Ostatecznie, to wyrażenie
                        ; zostanie skompilowane jako "mov ax,0e3a".
                        ; Chodzi o to po prostu, aby
                        ; nie uczyć się tabeli kodów ASCII na pamięć.

        int 10h         ; wypisujemy dwukropek

        add ecx,edi     ; dodajemy poprzednią ilość znalezionych
                        ; liczb pierwszych
```

```
        mov eax,ecx     ; EAX = ilość liczb pierwszych od 2 do
                        ; końca bieżącego przedziału

        call _pisz_ld   ; wypisujemy tą ilość.


        mov ah,1        ; int 16h, funkcja nr 1: czy w buforze
                        ; klawiatury jest znak?
        int 16h
        jz dalej        ; ZF = 1 oznacza brak znaku. Pracuj dalej.
        xor ah,ah
        int 16h         ; pobierz ten znak z bufora
                        ; (int 16h/ah=1 tego nie robi)
koniec:
        mov ax,4c00h
        int 21h         ; wyjdź z programu z kodem wyjścia = 0

dalej:                  ; nie naciśnięto klawisza
        cmp esi,100000  ; 10^5
        je koniec       ; ESI = 100.000? Tak - koniec, bo dalej
                        ; liczy zbyt długo.

        mov eax,esi     ; EAX=ESI
        shl eax,3       ; EAX = EAX*8
        shl esi,1       ; ESI=ESI*2
        add esi,eax     ; ESI = ESI*2 + EAX*8 =ESI*2+ESI*8= ESI*10.
                        ; Znacznie szybciej niż MUL

        xor edi,edi     ; bieżący licznik liczb

        jmp short petla ; robimy od początku...

przedzial       db      10,13,"Przedzial 2-$"


; NASM/FASM:
; _pisz_bufor: times 6 db 0
_pisz_bufor db 6 dup (0) ; miejsce na cyfry dla następującej procedury:


_pisz_ld:

;we: EAX=liczba bez znaku do wypisania

        push ecx                ; zachowujemy modyfikowane rejestry
        push edx
        push eax
        push esi

        xor si,si       ; SI=0. Będzie wskaźnikiem w powyższy bufor.

        mov ecx,10      ; będziemy dzielić przez 10, aby uzyskiwać
                        ; kolejne cyfry. Reszty z dzielenia pójdą
                        ; do bufora, potem będą wypisane wspak, bo
                        ; pierwsza reszta jest przecież cyfrą jedności

_pisz_ld_petla:
        xor edx,edx     ; EDX=0

        div ecx         ; EAX=EAX/ECX, EDX = reszta, która mieści się
                        ; w DL, bo to jest tylko 1 cyfra dziesiętna
```

```
        mov [_pisz_bufor+si],dl ; Cyfra do bufora.

        inc si          ; Zwiększ numer komórki w buforze, do której
                        ; będziemy teraz pisać

        or eax,eax      ; EAX = 0 ?

        jnz _pisz_ld_petla ; Jeśli nie (JNZ), to skok do początku pętli

        mov ah,0eh              ; funkcja wypisania
_pisz_ld_wypis:
        mov al,[_pisz_bufor+si-1] ; SI wskazuje poza ostatnią cyfrę,
                        ; dlatego jest -1. Teraz AL= ostatnia cyfra,
                        ; czyli ta najbardziej znacząca w liczbie

                ; Zamień liczbę 0-9 w AL na gotową do wypisania cyfrę:
        or al,"0" ; lub "OR al,30h" lub "ADD al,30h".

        int 10h                 ; wypisz AL

        dec si                  ; zmniejsz wskaźnik do bufora.

        jnz _pisz_ld_wypis      ; Jeśli ten wskaźnik (SI) nie jest zerem,
                                ; wypisuj dalej

        pop esi                 ; odzyskaj zachowane rejestry
        pop eax
        pop edx
        pop ecx

        ret                     ; powrót z procedury

end start                       ; NASM/FASM: usunąć tą linijkę
```

Kilka uwag o tym programie:

- Czemu nie zrobiłem `MOV EBX,2` a potem `INC EBX`, które musiało być w pętli?
  Bo `XOR EBX, EBX` jest krótsze i szybsze.

- Dobra. Więc czemu nie:

  ```
  xor ebx,ebx
  inc ebx
  inc ebx
  ```

  Te instrukcje operują na tym samym rejestrze i każda musi poczekać, aż poprzednia się zakończy. Współczesne procesory potrafią wykonywać niezależne czynności równolegle, dlatego wcisnąłem w środek jeszcze kilka niezależnych instrukcji.

- Ten program sprawdza za dużo dzielników. Nie można było sprawić, by sprawdzał tylko do np. połowy sprawdzanej liczby?
  Można było. Używając zmiennych w pamięci. Niechętnie to robię, bo w porównaniu z prędkością operacji procesora, pamięć jest wprost NIEWIARYGODNIE wolna. Zależało mi na szybkości.

- Czy zamiast

  ```
  mov ax,(0eh shl 8) or ":"
  ```

nie prościej byłoby zapisać

```
mov ah,0eh
mov al,":"      ; lub 3ah
```

Jasne, że byłoby prościej... zrozumieć. Ale nie wykonać dla procesora. Jedną instrukcję wykonuje się szybciej niż 2 i to jeszcze pośrednio operujące na tym samym rejestrze (AX).

- Czy nie prościej zamiast tych wszystkich SHL zapisać jedno MUL lub IMUL?
  Jasne, że prościej. Przy okazji dobre kilka[naście] razy wolniej.

- Dlaczego ciągle `XOR rej,rej`?
  Szybsze niż `MOV rej,0`, gdzie to zero musi być często zapisane 4 bajtami zerowymi. Tak więc i krótsze.

- Dlaczego na niektórych etykietach są jakieś znaki podkreślenia z przodu?
  Niektóre procedury są żywcem wyjęte z mojej biblioteki, pisząc którą musiałem zadbać, by przypadkowo nazwa jakieś mojej procedury nie była identyczna z nazwą jakiejś innej napisanej w programie korzystającym z biblioteki.
  Czy nie mogłem tego potem zmienić?
  Jasne, że mogłem. Ale nie było takiej potrzeby.

- Czemu `OR rej,rej` a nie `CMP rej,0`?
  OR jest krótsze i szybsze. Można też używać `TEST rej,rej`, które nie zmienia zawartości rejestru.

- Czemu `OR al, "0"`?
  Bardziej czytelne niż `ADD/OR al,30h`. Chodzi o to, aby dodać kod ASCII zera. I można to zrobić bardziej lub mniej czytelnie.

Wiem, że ten program nie jest doskonały. Ale taki już po prostu napisałem...
Nie martwcie się, jeśli czegoś od razu nie zrozumiecie. Naprawdę, z czasem samo przyjdzie. Ja też przecież nie umiałem wszystkiego od razu.

Inny program do liczb pierwszych znajdziecie tu: prime.txt.

Następnym razem coś o ułamkach i koprocesorze.

Podstawowe prawo logiki:
Jeżeli wiesz, że nic nie wiesz, to nic nie wiesz.
Jeżeli wiesz, że nic nie wiesz, to coś wiesz.
Więc nie wiesz, że nic nie wiesz.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia:
(można korzystać z zamieszczonych tu procedur)

1. Napisz program, który na ekranie wyświetli liczby 90-100.

2. Napisz program sprawdzający, czy dana liczba (umieścisz ją w kodzie, nie musi być wczytywana znikąd) jest liczbą pierwszą.

3. Napisz program wypisujący dzielniki danej liczby (liczba też w kodzie).

Jak pisać programy w języku asembler?
Część 5 - Koprocesor, czyli jak liczyć na ułamkach. Odwrotna Notacja Polska.

Jak zapewne większość wie, koprocesor (FPU = Floating Point Unit, NPX = Numerical Processor eXtension) służy do wykonywania działań matematycznych. Podstawowy procesor też oczywiście wykonuje działania matematyczne (dodawanie, mnożenie, ...) ale tylko na liczbach całkowitych. Z czasem jednak przyszła potrzeba wykonywania obliczeń na liczbach niecałkowitych, czyli ułamkach (liczbach zmiennoprzecinkowych). Dlatego firmy produkujące procesory zaczęły je wyposażać właśnie w układy wspomagające pracę na ułamkach. Do procesorów 8086, 80286, 80386 były dołączane jako osobne układy koprocesory: 8087, 80287, 80387 (80187 nie wprowadził żadnych istotnych nowości. Była to przeróbka 8087, a może nawet był to po prostu ten sam układ). Procesory 486SX miały jeszcze oddzielny koprocesor (80387) a od 486DX (w każdym razie u Intela) koprocesor był już na jednym chipie z procesorem. I tak jest do dziś.

Ale dość wstępu. Pora na szczegóły.

Typy danych

Zanim zaczniemy cokolwiek robić, trzeba wiedzieć, na czym ten cały koprocesor operuje.
Oprócz liczb całkowitych, FPU operuje na liczbach ułamkowych różnej precyzji:

- Pojedyncza precyzja. Liczby takie zajmują po 32 bity (4 bajty) i ich wartość maksymalna wynosi ok. 10^{39} (10^39). Znane są programistom języka C jako float.
- Podwójna precyzja. 64 bity (8 bajtów), max = ok. 10^{409} (10^409). W języku C są znane jako double
- Rozszerzona precyzja. 80 bitów (10 bajtów), max = ok. 10^{4930} (10^4930). W języku C są to long double

Jak widać, ilości bitów są oczywiście skończone. Więc nie każdą liczbę rzeczywistą da się dokładnie zapisać w postaci binarnej. Na przykład, jedna dziesiąta (0.1) zapisana dwójkowo jest ułamkiem nieskończonym okresowym! Poza tym, to, czego nas uczyli na matematyce, np. oczywista równość: a+(b-a)=b nie musi być prawdą w świecie ułamków w procesorze ze względu na brak precyzji!

Poza ułamkami, FPU umie operować na BCD (binary coded decimal). W takich liczbach 1 bajt dzieli się na 2 części po 4 bity, z których każda może mieć wartość 0-9. Cały bajt reprezentuje więc liczby od 0 do 99, w których cyfra jedności jest w młodszych 4 bitach a cyfra dziesiątek - w starszych 4 bitach.
Szczegółami zapisu liczb ułamkowych nie będziemy się tutaj zajmować. Polecam, tak jak poprzednio: strony Intela i strony AMD , gdzie znajduje się też kompletny opis wszystkich instrukcji procesora i koprocesora.

Rejestry koprocesora

Po zapoznaniu się z typami (a przede wszystkim z rozmiarami) liczb ułamkowych, powstaje pytanie: gdzie koprocesor przechowuje takie ilości danych?
FPU ma specjalnie do tego celu przeznaczonych 8 rejestrów, po 80 bitów każdy. W operacjach wewnętrznych (bez pobierania lub zapisywania danych do pamięci) FPU zawsze używa rozszerzonej precyzji.

Rejestry danych nazwano st(0), st(1), ... , st(7) (NASM: st0 ... st7). Nie działają jednak one tak, jak zwykłe rejestry, lecz jak ... stos! To znaczy, że dowolnie dostępna jest tylko wartość ostatnio położona na stosie czyli wierzchołek stosu, w tym przypadku: st(0). Znaczy to, że do pamięci (FPU operuje tylko na własnych

rejestrach lub pamięci - nie może używać rejestrów ogólnego przeznaczenia np. EAX itp.) może być zapisana tylko wartość z st(0), a każda wartość pobierana z pamięci idzie do st(0) a stare st(0) przesuwa się do st(1) itd. każdy rejestr przesuwa się o 1 dalej. Jeżeli brakuje na to miejsca, to FPU może wygenerować przerwanie (wyjątek) a rejestry danych będą prawdopodobnie zawierać śmieci.
Operowanie na rejestrach FPU będzie wymagało nieco więcej uwagi niż na zwykłych, ale i do tego można się przyzwyczaić.

Instrukcje koprocesora

Zacznijmy od omówienia kilku podstawowych instrukcji. Przez [mem] będę nazywał dane będące w pamięci (32-, 64- lub 80-bitowe, int oznacza liczbę całkowitą), st jest częstym skrótem od st(0). Jeżeli komenda kończy się na P to oznacza, że zdejmuje dane raz ze stosu, PP oznacza, że zdejmuje 2 razy: st(0) i st(1).

1. Instrukcje przemieszczenia danych:
   - FLD/FILD [mem] - załaduj liczbę rzeczywistą/całkowitą z pamięci. Dla liczby rzeczywistej jest to 32, 64 lub 80 bitów. Dla całkowitej - 16, 32 lub 64 bity.
   - FST [mem32/64/80] - do pamięci idzie liczba ze st(0).
   - FSTP [mem32/64/80] - zapisz st(0) w pamięci i zdejmij je ze stosu. Znaczy to tyle, że st(1) o ile istnieje, staje się st(0) itd. każdy rejestr cofa się o 1.
   - FIST [mem16/32] - ewentualnie obciętą do całkowitej liczbę z st(0) zapisz do pamięci.
   - FISTP [mem16/32/64] - jak wyżej, tylko ze zdjęciem ze stosu.
   - FXCH st(i) - zamień st(0) z st(i).

2. Instrukcje ładowania stałych
   - FLDZ - załaduj zero. st(0) = 0.0
   - FLD1 - załaduj 1. st(0) = 1.0
   - FLDPI - załaduj pi.
   - FLDL2T - załaduj log2(10)
   - FLDL2E - załaduj log2(e)
   - FLDLG2 - załaduj log(2)=log10(2)
   - FLDLN2 - załaduj ln(2)

3. Działania matematyczne:
   - dodawanie: FADD, składnia identyczna jak w odejmowaniu prostym

   - odejmowanie:
     FSUB [mem32/64] st := st-[mem] ,
     FSUB st(0),st(i) st := st-st(i),
     FSUB st(i),st(0) st(i) := st(i)-st(0),
     FSUBP st(i), st(0) st(i) := st(i)-st(0) i zdejmij,
     FSUBP (bez argumentów) = FSUBP st(1),st(0) ,
     FISUB [mem16/32int] st := st-[mem]

   - odejmowanie odwrotne:
     FSUBR [mem32/64] st := [mem]-st(0)
     FSUBR st(0),st(i) st := st(i)-st(0)
     FSUBR st(i),st(0) st(i) := st(0)-st(i)
     FSUBRP st(i),st(0) st(i) := st(0)-st(i) i zdejmij

FSUBRP (bez argumentów) = FSUBRP st(1),st(0)
FISUBR [mem16/32int] st := [mem]-st

- mnożenie: FMUL, składnia identyczna jak w odejmowaniu prostym.

- dzielenie: FDIV, składnia identyczna jak w odejmowaniu prostym.
- dzielenie odwrotne: FDIVR, składnia identyczna jak w odejmowaniu odwrotnym.

- wartość bezwzględna: FABS (bez argumentów) zastępuje st(0) jego wartością bezwzględną.
- zmiana znaku: FCHS: st(0) := -st(0).
- pierwiastek kwadratowy: FSQRT: st(0) := SQRT[st(0)]
- reszty z dzielenia: FPREM, FPREM1 st(0) := st(0) mod st(1).
- zaokrąglanie do liczby całkowitej: FRNDINT: st(0) := (int)st(0).

4. Komendy porównania:
   - FCOM/FCOMP/FCOMPP, FUCOM/FUCOMP/FUCOMPP, FICOM/FICOMP, FCOMI/FCOMIP, FUCOMI/FUCOMIP, FTST, FXAM.
   Tutaj trzeba trochę omówić sytuację. FPU oprócz rejestrów danych zawiera także rejestr kontrolny (16 bitów) i rejestr stanu (16 bitów).
   W rejestrze stanu są 4 bity nazwane C0, C1, C2 i C3. To one wskazują wynik ostatniego porównania, a układ ich jest taki sam, jak flag procesora, co pozwala na ich szybkie przeniesienie do flag procesora. Aby odczytać wynik porównania, należy zrobić takie coś:

```
fcom
fstsw   ax      ; tylko od 386. Inaczej:
                ; fstsw word ptr [zmienna] / mov ax,[zmienna]
sahf            ; AH -> flagi
```

   i używać normalnych komend JE, JB itp.
   FCOM st(n)/[mem] - porównaj st(0) z st(n) (lub zmienną w pamięci) bez zdejmowania st(0) ze stosu FPU
   FCOMP st(n)/[mem] - porównaj st(0) z st(n) (lub zmienną w pamięci) i zdejmij st(0)
   FCOMPP - porównaj st(0) z st(1) i zdejmij oba ze stosu
   FICOM [mem] - porównaj st(0) ze zmienną całkowitą 16- lub 32-bitową w pamięci
   FICOMP [mem] - porównaj st(0) ze zmienną całkowitą 16- lub 32-bitową w pamięci, zdejmij st(0)
   FCOMI st(0), st(n) - porównaj st(0) z st(n) i ustaw flagi procesora, nie tylko FPU
   FCOMIP st(0), st(n) - porównaj st(0) z st(n) i ustaw flagi procesora, nie tylko FPU, zdejmij st(0)

   Komendy kończące sie na I lub IP zapisują swój wynik bezpośrednio do flag procesora. Można tych flag od razu używać (JZ, JA, ...). Te komendy są dostępne tylko od 386.

   FTST porównuje st(0) z zerem.
   FXAM bada, co jest w st(0) - prawidłowa liczba, błąd (NaN = Not a Number) czy 0.

5. Instrukcje trygonometryczne:
   - FSIN, FCOS, FSINCOS, FPTAN, FPATAN
   st(0) := funkcja[st(0)].
   FPTAN = partial tangent = tangens,
   FPATAN = arcus tangens.

6. Logarytmiczne, wykładnicze:
   - FYL2X st(1) := st(1)*log2[st(0)] i zdejmij
   - FYL2XPI st(1) := st(1)*log2[st(0) + 1.0] i zdejmij
   - F2XM1 st(0) := 2^[st(0)] - 1

7. Instrukcje kontrolne:
   - FINIT/FNINIT - inicjalizacja FPU. Litera N po F oznacza, aby nie brać po uwagę potencjalnych niezałatwionych wyjątków.
   - FLDCW, FSTCW/FNSTCW - Load/Store control word - zapisuje 16 kontrolnych bitów do pamięci, gdzie można je zmieniać np. aby zmienić sposób zaokrąglania liczb.
   - FSTSW/FNSTSW - zapisz do pamięci (lub rejestru AX) słowo statusu, czyli stan FPU
   - FCLEX/FNCLEX - wyczyść wyjątki
   - FLDENV, FSTENV/FNSTENV - wczytaj/zapisz środowisko (rejestry stanu, kontrolny i kilka innych, bez rejestrów danych). Wymaga 14 albo 28 bajtów pamięci, w zależności od trybu pracy procesora (rzeczywisty - DOS lub chroniony - Windows/Linux).
   - FRSTOR, FSAVE/FNSAVE - jak wyżej, tylko że z rejestrami danych. Wymaga 94 lub 108 bajtów w pamięci, zależnie od trybu procesora.
   - FINCSTP, FDECSTP - zwiększ/zmniejsz wskaźnik stosu - przesuń st(0) na st(7), st(1) na st(0) itd. oraz w drugą stronę, odpowiednio.
   - FFREE - zwolnij podany rejestr danych
   - FNOP - no operation. Nic nie robi, ale zabiera czas.
   - WAIT/FWAIT - czekaj, aż FPU skończy pracę. Używane do synchronizacji z CPU.

Przykłady
Dość już teorii, pora na przykłady. Programiki te wymyśliłem pisząc ten kurs.

Przykład 1 (będzie to program typu .exe, bo dodamy moją bibliotekę do wyświetlania wyników):
(przeskocz program wyświetlający częstotliwość zegara)

```
; TASM:
; z wyświetlaniem:
;   tasm naszplik.asm
;   tlink naszplik.obj bibl\lib\bibldos.lib
; bez wyświetlania:
;   tasm naszplik.asm
;   tlink naszplik.obj

.model small
.stack 400h                     ; stos dla programu .exe
.data

dzielna         DQ 1234DDh      ; 4013 91a6 e800 0000 0000
dzielnik        DQ 10000h
iloraz          DT ?

.code

; jeśli nie chcesz wyświetlania, usuń tę linijkę niżej:
 include incl\std_bibl.inc

start:
```

```
        mov     ax, @data
        mov     ds, ax
        mov     es, ax          ; konieczne w programie typu .exe !
                                ; Życie przestaje być
                                ; wygodne. DOS już nam nie ustawi
                                ; DS=ES=CS. A nasze dane są w
                                ; segmencie kodu, stad ustawiamy DS=CS
                                ; W programie typu .com to na pewno
                                ; nie zaszkodzi.

        finit                   ; zawsze o tym pamiętaj !!!!

        fild    qword ptr [dzielna]     ; ładujemy dzielną. st(0) = 1234DD
        fild    qword ptr [dzielnik]    ; ładujemy dzielnik. st(0) = 10000h,
                                        ; st(1) = 1234DD
        fdivp                           ; dzielimy. st(1) := st(1)/st(0) i
                                        ; zdejmij. st(0) ~= 18.2
        fstp    tbyte ptr [iloraz]      ; zapisujemy st(0) do pamięci i
                                        ; zdejmujemy ze stosu

; jeśli nie chcesz wyświetlania, usuń te 3 linijki niżej:
        mov     di, offset iloraz ; DI=adres zmiennej zawierającej wynik
        piszd80                   ; wyświetl wynik
        nwln                      ; przejdź do nowej linii

        mov     ax, 4c00h
        int     21h

end start
```

Teraz wersja dla NASMa. O tym, jak NASMem zrobić program typu .exe napisane jest w jego dokumentacji. Wymaga to przede wszystkim stworzenia własnego segmentu stosu i nakierowanie na niego rejestrów SS:SP. Trzeba mieć też odpowiedni linker, np. VAL. Można jednak użyć jednego z plików dołączonych do mojej biblioteki i tak też zrobimy.
(przeskocz ten program w wersji NASMa)

```
; NASM:
; z wyświetlaniem:
;   nasm -f obj -o fpu1.obj fpu1.asm
;   val fpu1.obj,fpu1.exe,,bibl\lib\bibldos.lib,
; bez wyświetlania:
;   nasm -f obj -o fpu1.obj fpu1.asm
;   val fpu1.obj,fpu1.exe,,,

; częściowa zgodność z TASMem:
%include "bibl\incl\dosbios\nasm\do_nasma.inc"

; jeśli nie chcesz wyświetlania, usuń tą linijkę niżej:
%include "bibl\incl\dosbios\nasm\std_bibl.inc"

.model small
.stack 400h
.code

..start:
        mov     ax, cs
        mov     ds, ax
        mov     es, ax          ; konieczne w programie typu .exe !
                                ; Życie przestaje być
                                ; wygodne. DOS już nam nie ustawi DS=ES=CS.
                                ; A nasze dane są w
```

```
                                ; segmencie kodu, stąd ustawiamy DS=CS.
                                ; W programie typu .com to na pewno nie
                                ; zaszkodzi.

        finit                   ; zawsze o tym pamiętaj !!!!

        fild    dword [dzielna] ; ładujemy dzielną. st(0) = 1234DD
        fild    dword [dzielnik]; ładujemy dzielnik. st(0) = 10000h,
                                ; st(1) = 1234DD
        fdivp   st1, st0        ; dzielimy. st(1) := st(1)/st(0) i
                                ; zdejmij. st(0) ~= 18.2
                                ; FASM: fdivp st(1)

        fstp    tword [iloraz]  ; zapisujemy st(0) do pamięci i zdejmujemy
                                ; ze stosu

; jeśli nie chcesz wyświetlania, usuń te 3 linijki niżej:
        mov     di, iloraz
        piszd80                 ; wyświetl wynik
        nwln                    ; przejdź do nowego wiersza

        mov     ax, 4c00h
        int     21h

align 8                         ; NASM w tym miejscu dorobi kilka NOP-ów
                                ; (instrukcji nic nie robiących, ale
                                ; zabierających czas), aby adres dzielił się
                                ; przez 8 (patrz dalej).

dzielna         dd 1234ddh      ; 4013 91a6 e800 0000 0000
dzielnik        dd 10000h

iloraz          dt 0.0
```

Wersja dla FASMa:
(przeskocz ten sam program w wersji FASMa)

```
; FASM:
; z wyświetlaniem:
;   fasm fpu1.asm fpu1.obj
;   alink fpu1.obj  bibl\lib\bibldos.lib -c- -entry _start -oEXE -m-
; bez wyświetlania:
;   fasm fpu1.asm fpu1.exe

; jeśli chcesz wyświetlanie:
format coff
public _start
include "bibl\incl\dosbios\fasm\std_bibl.inc"
use16

; jeśli nie chcesz wyświetlania:
;format MZ
;entry kod:_start
;stack 400h
;segment kod

_start:
        mov     ax, cs
        mov     ds, ax
        mov     es, ax          ; konieczne w programie typu .exe !
                                ; Życie przestaje być
```

```
                                ; wygodne. DOS już nam nie ustawi DS=ES=CS.
                                ; A nasze dane są w
                                ; segmencie kodu, stąd ustawiamy DS=CS.
                                ; W programie typu .com to na pewno nie
                                ; zaszkodzi.

        finit                   ; zawsze o tym pamiętaj !!!!

        fild    dword [dzielna] ; ładujemy dzielną. st(0) = 1234DD
        fild    dword [dzielnik]; ładujemy dzielnik. st(0) = 10000h,
                                ; st(1) = 1234DD
        fdivp                   ; dzielimy. st(1) := st(1)/st(0) i
                                ; zdejmij. st(0) ~= 18.2

        fstp    tword [iloraz]  ; zapisujemy st(0) do pamięci i zdejmujemy
                                ; ze stosu

; jeśli nie chcesz wyświetlania, usuń te 3 linijki niżej:
        mov     edi, iloraz
        piszd80                 ; wyświetl wynik
        nwln                    ; przejdź do nowego wiersza

        mov     ax, 4c00h
        int     21h

dzielna         dd 1234ddh      ; 4013 91a6 e800 0000 0000
dzielnik        dd 10000h

iloraz          dt 0.0
```

Ten przykład do zmiennej iloraz wstawia częstotliwość zegara komputerowego (ok. 18,2 Hz). Należy zwrócić uwagę na zaznaczanie rozmiarów zmiennych (dword/qword/tbyte ptr).
Dyrektywa ALIGN ustawia kolejną zmienną/eytkietę tak, że jej adres dzieli się przez 8 (qword = 8 bajtów). Dzięki temu, operacje na pamięci są szybsze (np. dla zmiennej 8-bajtowej zamiast 3 razy pobierać po 4 bajty, bo akurat tak się zdarzyło, że miała jakiś nieparzysty adres, pobiera się 2x4 bajty). Rzecz jasna, skoro zmienna dzielna (i dzielnik) ma 4 bajty, to adresy zmiennych dzielnik i iloraz też będą podzielne przez 4.
Ciąg cyfr po średniku to ułamkowa reprezentacja dzielnej. Skomplikowane, prawda? Dlatego nie chciałem tego omawiać.

Przykład 2: czy sinus liczby pi rzeczywiście jest równy 0 (w komputerze)?
(przeskocz program z sinusem)

```
.model tiny
.code
;.386                           ; odkomentować, jeżeli .387 sprawia problemy
.387

org 100h
start:
        finit                   ; zawsze o tym pamiętaj !!!!

        fldpi                   ; wczytujemy PI
        fsin                    ; obliczamy sin(PI)
        ftst                    ; porównujemy st(0) z zerem.
        fstsw ax                ; zapisujemy rejestr stanu bezpośrednio w AX.
                                ; Dlatego było .387

        sahf                    ; AH idzie do flag
```

```
        mov ah,9                ; AH=9, flagi niezmienione
        je jest_zero            ; st(0) = 0? Jeśli tak, to wypisz, że jest
        mov dx,offset nie_zero  ; zmienić DX na EDX, jeżeli sprawia problemy
        jmp short pisz
jest_zero:
        mov dx,offset zero      ; DX/EDX jak wyżej
pisz:
        int 21h                 ; wypisz jedną z wiadomości.

        mov ax,4c00h
        int 21h

nie_zero        db      "Sin(PI) != 0.$"
zero            db      "Sin(PI) = 0$"

end start
```

Wersja dla NASMa i FASMa:
(przeskocz wersję NASM/FASM programu z sinusem)

```
org 100h

start:
        finit                   ; zawsze o tym pamiętaj !!!!

        fldpi                   ; wczytujemy PI
        fsin                    ; obliczamy sin(PI)
        ftst                    ; porównujemy st(0) z zerem.
        fstsw ax                ; zapisujemy rejestr stanu bezpośrednio w AX.
                                ; Dlatego było .387

        sahf                    ; AH idzie do flag
        mov ah,9                ; AH=9, flagi niezmienione
        je jest_zero            ; st(0) = 0? Jeśli tak, to wypisz, że jest
        mov dx,nie_zero
        jmp short pisz
jest_zero:
        mov dx,zero
pisz:
        int 21h                 ; wypisz jedną z wiadomości.

        mov ax,4c00h
        int 21h

nie_zero        db      "Sin(PI) != 0$"
zero            db      "Sin(PI) = 0$"
```

Przykład 3: czy pierwiastek z 256 rzeczywiście jest równy 16, czy 200 jest kwadratem liczby całkowitej (komentarze do .386/.387 jak wyżej)?
(przeskocz ten przykład)

```
.model tiny
.code
.386
.387

org 100h                        ; program typu .com

start:
        finit                   ; zawsze o tym pamiętaj !!!!
```

```
        mov ax,cs
        mov ds,ax               ; konieczne w programie typu .exe !
                                ; Życie przestaje być
                                ; wygodne. DOS już nam nie ustawi DS=ES=CS.
                                ; A nasze dane są w
                                ; segmencie kodu, stąd ustawiamy DS=CS.

        fild word ptr [dwa_pie_sze]     ; st(0) = 256
        fsqrt                           ; st(0) = sqrt(256)
        fild word ptr [szesnascie]      ; st(0) = 16, st(1) = sqrt(256)
        fcompp                          ; porównaj st(0) i st(1), zdejmij oba
                                        ; st: [pusty]
        fstsw ax
        sahf

        mov ah,9
        je tak256
        mov dx,offset nie_256
        jmp short pisz_256
tak256:
        mov dx,offset tak_256
pisz_256:
        int 21h                         ; wypisz stosowną wiadomość

                        ; do zapisu stanu stosu, czyli rejestrów danych FPU
                        ; można używać takiego schematu zapisu,
                        ; który jest krótszy:
                        ; st:  (0),  (1),  (2),  ... , (7)

        fild word ptr [dwiescie]        ; st: 200
        fsqrt                           ; st: sqrt(200)
        fld st(0)                       ; do st(0) wczytaj st(0).
                                        ; st: sqrt(200), sqrt(200)
        frndint                         ; zaokrąglij do liczby całkowitej.
                                        ; st:  (int)sqrt(200), sqrt(200)
        fcompp                          ; porównaj i zdejmij oba.
        fstsw ax
        sahf

        mov ah,9
        je tak200
        mov dx,offset nie_200
        jmp short pisz_200
tak200:
        mov dx,offset tak_200
pisz_200:
        int 21h                         ; wypisz stosowną wiadomość


        mov ax,4c00h
        int 21h

dwa_pie_sze     dw      256
dwiescie        dw      200
szesnascie      dw      16

nie_256 db      "SQRT(256) != 16$"
tak_256 db      "SQRT(256) = 16$"
nie_200 db      "Liczba 200 nie jest kwadratem liczby calkowitej$"
tak_200 db      "Liczba 200 jest kwadratem liczby calkowitej$"
end start
```

Teraz dla NASMa i FASMa:
(przeskocz ten sam przykład w wersji NASM/FASM)

```
org 100h                        ; program typu .com

start:
        finit                   ; zawsze o tym pamiętaj !!!!

        mov ax,cs
        mov ds,ax               ; konieczne w programie typu .exe !
                                ; Życie przestaje być
                                ; wygodne. DOS już nam nie ustawi DS=ES=CS.
                                ; A nasze dane są w
                                ; segmencie kodu, stąd ustawiamy DS=CS.
                                ; W programie typu .com to na pewno nie
                                ; zaszkodzi.

        fild word  [dwa_pie_sze]        ; st(0) = 256
        fsqrt                           ; st(0) = sqrt(256)
        fild word [szesnascie]  ; st(0) = 16, st(1) = sqrt(256)
        fcompp                  ; porównaj st(0) i st(1) i zdejmij oba
                                        ; st: [pusty]
        fstsw ax
        sahf

        mov ah,9
        je tak256
        mov dx, nie_256
        jmp short pisz_256
tak256:
        mov dx,tak_256
pisz_256:
        int 21h                         ; wypisz stosowną wiadomość

                        ; do zapisu stanu stosu, czyli rejestrów danych FPU
                        ; można używać takiego schematu zapisu,
                        ; który jest krótszy:
                        ; st:  (0),  (1),  (2),  ... , (7)

        fild word [dwiescie]            ; st: 200
        fsqrt                           ; st: sqrt(200)
        fld st0                         ; do st(0) wczytaj st(0).
                                        ; st: sqrt(200), sqrt(200)
        frndint                         ; zaokrąglij do liczby całkowitej.
                                        ; st:  (int)sqrt(200), sqrt(200)
        fcompp                          ; porównaj i zdejmij oba.
        fstsw ax
        sahf

        mov ah,9
        je tak200
        mov dx,nie_200
        jmp short pisz_200
tak200:
        mov dx,tak_200
pisz_200:
        int 21h                         ; wypisz stosowną wiadomość


        mov ax,4c00h
```

```
        int 21h

dwa_pie_sze     dw      256
dwiescie        dw      200
szesnascie      dw      16

nie_256 db      "SQRT(256) != 16$"
tak_256 db      "SQRT(256) = 16$"
nie_200 db      "Liczba 200 nie jest kwadratem liczby calkowitej$"
tak_200 db      "Liczba 200 jest kwadratem liczby calkowitej$"
```

Dwa ostatnie programiki zbiłem w jeden i przetestowałem. Wyszło, że sinus PI jest różny od zera, reszta była prawidłowa.

Oczywiście, w tych przykładach nie użyłem wszystkich instrukcji koprocesora (nawet spośród tych, które wymieniłem). Mam jednak nadzieję, że te proste programy rozjaśnią nieco sposób posługiwania się koprocesorem.

Odwrotna Notacja Polska (Reverse Polish Notation, RPN)

Ładnie brzmi, prawda? Ale co to takiego?
Otóż, bardzo dawno temu pewien polski matematyk, Jan Łukasiewicz, wymyślił taki sposób zapisywania działań, że nie trzeba w nim używać nawiasów. Była to notacja polska. Sposób ten został potem dopracowany przez Charlesa Hamblina na potrzeby informatyki - w ten sposób powstała Odwrotna Notacja Polska. W zapisie tym argumenty działania zapisuje przed symbolem tego działania. Dla jasności podam teraz kilka przykładów:
(przeskocz przykłady na ONP)

```
   Zapis tradycyjny                ONP
        a+b                     a b +
        a+b+c                   a b + c +  ; ab+ stanowi pierwszy argument
                                           ; drugiego dodawania
        c+b+a                   c b + a +
        (a+b)*c                 a b + c *
        c*(a+b)                 c a b + *
        (a+b)*c+d               a b + c * d +
        (a+b)*c+d*a             a b + c * d a * +
        (a+b)*c+d*(a+c)         a b + c * d a c + * +
        (a+b)*c+(a+c)*d         a b + c * a c + d * +
        (2+5)/7+3/5             2 5 + 7 / 3 5 / +
```

Ale po co to komu i dlaczego mówię o tym akurat w tej części?
Powód jest prosty: jak sie dobrze przyjrzeć zapisowi działania w ONP, to można zobaczyć, że mówi on o kolejności działań, jakie należy wykonać na koprocesorze. Omówimy to na przykładzie:
(przeskocz ilustrację relacji między ONP a koprocesorem)

```
        Zapis tradycyjny (jeden z powyższych przykładów):
                (a+b)*c+(a+c)*d

        Zapis w ONP:
                a b + c * a c + d * +

        Uproszczony kod programu:
```

```
        fld     [a]
        fld     [b]
        faddp                   ; NASM: faddp st1, st0
        fld     [c]
        fmulp                   ; NASM: fmulp st1, st0
        fld     [a]
        fld     [c]
        faddp                   ; NASM: faddp st1, st0
        fld     [d]
        fmulp                   ; NASM: fmulp st1, st0
        faddp                   ; NASM: faddp st1, st0

        Teraz st0 jest równe wartości całego wyrażenia.
```

Jak widać, ONP znacznie upraszcza przetłumaczenie wyrażenia na kod programu. Jednak, kod nie jest optymalny. Można byłoby na przykład zachować wartości zmiennych a i c na stosie i wtedy nie musielibyśmy ciągle pobierać ich z pamięci. Dlatego w krytycznych sekcjach kodu stosowanie zasad ONP nie jest zalecane. Ale w większości przypadków Odwrotna Notacja Polska sprawuje się dobrze i uwalnia programistów od obowiązku zgadywania kiedy i jakie działanie wykonać.
Pamiętajcie tylko, że stos koprocesora może pomieścić tylko 8 liczb!

Następnym razem o SIMD.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia:

1. Napisz program, który sprawdzi (wyświetli stosowną informację), czy liczba PI dzielona przez samą siebie daje dokładne 1.0

2. Napisz program obliczający (nie wyświetlający) wartość 10*PI. Potem sprawdź, czy sinus tej liczby jest zerem.

3. Napisz program mówiący, która z tych liczb jest większa: PI czy log2(10).

4. Napisz program sprawdzający, czy 10*PI - PI - PI - PI - PI - PI = 5*PI.

5. Zamień na ONP:
   a/c/d + b/c/d
   a/(c*d) + b/(c*d)
   (a+b)/c/d
   (a+b)/(c*d)

6. Zamień z ONP na zapis tradycyjny (daszek ^ oznacza potęgowanie):
   ab*cd*e/-
   a5/c7/ed-9/*+
   a3+b/de+6^-

```
dc-7b*2^/
```

Jak pisać programy w języku asembler?
Część 6 - SIMD, czyli jak działa MMX.

- A cóż to takiego to SIMD ?! - zapytacie.
Już odpowiadam.
SIMD = Single Instruction, Multiple Data = jedna instrukcja, wiele danych.
Jest to technologia umożliwiająca jednoczesne wykonywanie tej samej instrukcji na kilku wartościach. Na pewno znany jest wam co najmniej jeden przykład zastosowania technologii SIMD. Jest to MultiMedia Extensions, w skrócie MMX u Intela, a 3DNow! u AMD. Innym mniej znanym zastosowaniem jest SSE, które omówię później.
Zaczniemy od omówienia, jak właściwie działa to całe MMX.

MMX / 3DNow!
Technologia MMX operuje na 8 rejestrach danych, po 64 bity każdy, nazwanych mm0 ... mm7. Niestety, rejestry te nie są prawdziwymi (oddzielnymi) rejestrami - są częściami rejestrów koprocesora (które, jak pamiętamy, mają po 80 bitów każdy). Pamiętajcie więc, że nie można naraz wykonywać operacji na FPU i MMX/3DNow!.

Rejestry 64-bitowe służą do umieszczania w nich danych spakowanych. Na czym to polega? Zamiast mieć np. 32 bity w jednym rejestrze, można mieć dwa razy po 32. Tak więc rejestry mieszczą 2 podwójne słowa (dword, 32 bity) lub 4 słowa (word, 16 bitów) lub aż 8 spakowanych bajtów.

Zajmijmy się omówieniem instrukcji operujących na tych rejestrach.
Instrukcje MMX można podzielić na kilka grup (nie wszystkie instrukcje będą tu wymienione):

- instrukcje transferu danych:
  - MOVD mmi, rej32/mem32 (i=0,...,7)
  - MOVQ mmi, mmj/mem64 (i,j=0,...,7)

- instrukcje arytmetyczne:
  - dodawanie normalne: PADDB (bajty) / PADDW (słowa)/ PADDD (dword-y)

  - dodawanie z nasyceniem ze znakiem: PADDSB (bajty)/ PADDSW (słowa).
    Jeżeli wynik przekracza 127 lub 32767 (bajty/słowa), to jest do tej wartości zaokrąglany, a NIE jest tak, że nagle zmienia się na ujemny. Daje to lepszy efekt, np. w czasie słuchania muzyki czy oglądania filmu. Hipotetyczny przykład: 2 kolory szare z dadzą w sumie czarny a nie coś pośrodku skali kolorów.

  - dodawanie z nasyceniem bez znaku: PADDUSB / PADDUSW.
    Jeżeli wynik przekracza 255 lub 65535, to jest do tej wartości zaokrąglany.

  - odejmowanie normalne: PSUBB (bajty)/ PSUBW (słowa)/ PSUBD (dword-y)

  - odejmowanie z nasyceniem ze znakiem: PSUBSB (bajty)/ PSUBSW (słowa).
    Jeśli wynik jest mniejszy niż -128 lub -32768 to jest do tej wartości zaokrąglany.

- odejmowanie z nasyceniem bez znaku: PSUBUSB (bajty) / PSUBUSW (słowa)
  Jeśli wynik jest mniejszy niż 0, to staje się równy 0.

- mnożenie:
  - PMULHRWC, PMULHRIW, PMULHRWA- mnożenie spakowanych słów, zaokrąglanie, zapisanie tylko starszych 16 bitów wyniku (z 32).
  - PMULHUW - mnożenie spakowanych słów bez znaku, zachowanie starszych 16 bitów
  - PMULHW, PMULLW - mnożenie spakowanych słów bez znaku, zapisanie starszych/młodszych 16 bitów (odpowiednio).
  - PMULUDQ - mnożenie spakowanych dwordów bez znaku

- mnożenie i dodawanie: PMADDWD - do młodszego dworda rejestru docelowego idzie suma iloczynów 2 najmłodszych słów ze sobą i 2 starszych (bity 16-31) słów ze sobą. Do starszego dworda - suma iloczynów 2 słów 32-47 i 2 słów 48-63.

- instrukcje porównawcze:
  Zostawiają w odpowiednim bajcie/słowie/dwordzie same jedynki (FFh/FFFFh/FFFFFFFFh) gdy wynik porównania był prawdziwy, same zera - gdy fałszywy.
  - na równość PCMPEQB / PCMPEQW / PCMPEQD (EQ oznacza równość)
  - na większe niż: PCMPGTPB / PCMPGTPW / PCMPGTPD (GT oznacza greater than, czyli większy)

- instrukcje konwersji:
  - pakowanie: PACKSSWB / PACKSSDW, PACKUSWB - upychają słowa/dwordy do bajtów/słów i pozostawiają w rejestrze docelowym.
  - rozpakowania starszych części (unpack high): PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ - pobierają starsze części bajtów/słów/dwordów z jednego i drugiego rejestru, mieszają je i zostawiają w pierwszym.
  - rozpakowania młodszych części (unpack low): PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ - jak wyżej, tylko pobierane są młodsze części

- instrukcje logiczne:
  - PAND (bitowe AND)
  - PANDN (najpierw bitowe NOT pierwszego rejestru, potem jego bitowe AND z drugim rejestrem)
  - POR (bitowe OR)
  - PXOR (bitowe XOR)

- instrukcje przesunięcia (analogiczne do znanych SHL, SHR i SAR, odpowiednio):
  - w lewo: PSLLW (słowa) / PSLLD (dword-y), PSLLQ (qword)
  - w prawo, logiczne: PSRLW (słowa) / PSRLD (dword-y), PSRLQ (qword)
  - w prawo, arytmetyczne: PSRAW (słowa)/ PSRAD (dword-y)

- instrukcje stanu MMX:
  - EMMS - Empty MMX State - ustawia rejestry FPU jako wolne, umożliwiając ich użycie. Ta instrukcja musi być wykonana za każdym razem, gdy kończymy pracę z MMX i chcemy zacząć pracę z FPU.

Rzadko która z tych instrukcji traktuje rejestr jako całość, częściej operuje na poszczególnych wartościach osobno, równolegle.

Spróbuję teraz podać kilka przykładów zastosowania MMX. Ze względu na to, że TASM bez pomocy zewnętrznych plików z makrami nie obsługuje MMX, będę używał składni NASMa i FASMa.

Przykład 1. Dodawanie dwóch tablic bajtów w pamięci. Bez MMX mogłoby to wyglądać mniej-więcej tak:
(przeskocz dodawanie tablic)

```
; EDX - adres pierwszej tablicy bajtów
; ESI - adres drugiej tablicy bajtów
; EDI - adres docelowej tablicy bajtów
; ECX - liczba bajtów w tablicach. Przyjmiemy, że różna od zera...

        petla:
                mov al, [edx]   ; pobierz bajt z pierwszej
                add al, [esi]   ; dodaj bajt z drugiej
                mov [edi], al   ; zapisz bajt w docelowej
                inc edx         ; zwiększ o 1 indeksy do tablic
                inc esi
                inc edi
                loop petla      ; działaj, dopóki ECX różne od 0.
```

A z MMX:
(przeskocz dodawanie tablic z MMX)

```
        mov ebx, ecx    ; EBX = liczba bajtów
        and ebx, 7      ; będziemy brać po 8 bajtów - obliczamy
                        ; więc resztę z dzielenia przez 8

        shr ecx, 3      ; dzielimy ECX przez 8
petla:
        movq mm0, [edx] ; pobierz 8 bajtów z pierwszej tablicy
        paddb mm0, [esi]; dodaj 8 spakowanych bajtów z drugiej
        movq [edi], mm0 ; zapisz 8 bajtów w tablicy docelowej
        add edx, 8      ; zwiększ indeksy do tablic o 8
        add esi, 8
        add edi, 8
        loop petla      ; działaj, dopóki ECX różne od 0.

        test ebx, ebx   ; czy EBX = 0?
        jz koniec       ; jeśli tak, to już skończyliśmy

        mov ecx, ebx    ; ECX = resztka, co najwyżej 7 bajtów.
                        ; te kopiujemy tradycyjnie
petla2:
        mov al, [edx]   ; pobierz bajt z pierwszej
        add al, [esi]   ; dodaj bajt z drugiej
        mov [edi], al   ; zapisz bajt w docelowej
        inc edx         ; zwiększ o 1 indeksy do tablic
        inc esi
        inc edi
        loop petla2     ; działaj, dopóki ECX różne od 0
koniec:

        emms            ; wyczyść rejestry MMX, by FPU mogło z nich korzystać
```

Podobnie będą przebiegać operacje PAND, POR, PXOR, PANDN.

Przy dużych ilościach danych, sposób drugi będzie wykonywał około 8 razy mniej instrukcji niż pierwszych, bo dodaje na raz 8 bajtów. I o to właśnie chodziło.

Przykład 2. Kopiowanie pamięci.
Bez MMX:
(przeskocz kopiowanie pamięci)

```
; DS:SI - źródło
; ES:DI - cel
; ECX - ilość bajtów
        mov ebx, ecx    ; EBX = ilość bajtów
        and ebx, 3      ; EBX = reszta z dzielenia liczby bajtów przez 4
        shr ecx, 2      ; ECX = liczba bajtów dzielona przez 4

        cld             ; kierunek: do przodu
        rep movsd       ; dword z DS:SI idzie pod ES:DI, DI:=DI+4,
                        ; SI:=SI+4, dopóki CX jest różny od 0
        mov ecx, ebx    ; ECX = liczba pozostałych bajtów
        rep movsb       ; resztkę kopiujemy po bajcie
```

Z MMX:
(przeskocz kopiowanie pamięci z MMX)

```
                mov ebx, ecx    ; EBX = ilość bajtów
                and ebx, 7      ; EBX = reszta z dzielenia liczby bajtów
                                ; przez 8

                shr ecx, 3      ; ECX = liczba bajtów dzielona przez 8
        petla:
                movq mm0, [esi] ; MM0 = 8 bajtów z tablicy pierwszej
                movq [edi], mm0 ; kopiujemy zawartość MM0 pod [EDI]
                add esi, 8      ; zwiększamy indeksy tablic o 8
                add edi, 8
                loop petla      ; działaj, dopóki ECX różne od 0

                mov ecx, ebx    ; ECX = liczba pozostałych bajtów
                cld             ; kierunek: do przodu
                rep movsb       ; resztkę kopiujemy po bajcie

                emms            ; wyczyść rejestry MMX
```

lub, dla solidniejszych porcji danych:
(przeskocz kolejne kopiowanie pamięci)

```
                mov ebx, ecx    ; EBX = ilość bajtów
                and ebx, 63     ; EBX = reszta z dzielenia liczby bajtów
                                ; przez 64
                shr ecx, 6      ; ECX = liczba bajtów dzielona przez 64
        petla:
                ; kopiuj 64 bajty spod [ESI] do rejestrów MM0, ... MM7
                movq mm0, [esi]
                movq mm1, [esi+8]
                movq mm2, [esi+16]
                movq mm3, [esi+24]
                movq mm4, [esi+32]
```

```
                movq mm5, [esi+40]
                movq mm6, [esi+48]
                movq mm7, [esi+56]

                ; kopiuj 64 bajty z rejestrów MM0, ... MM7 do [EDI]
                movq [edi   ], mm0
                movq [edi+8 ], mm1
                movq [edi+16], mm2
                movq [edi+24], mm3
                movq [edi+32], mm4
                movq [edi+40], mm5
                movq [edi+48], mm6
                movq [edi+56], mm7

                add esi, 64     ; zwiększ indeksy do tablic o 64
                add edi, 64
                loop petla      ; działaj, dopóki ECX różne od 0

                mov ecx, ebx    ; ECX = liczba pozostałych bajtów
                cld             ; kierunek: do przodu
                rep movsb       ; resztkę kopiujemy po bajcie

                emms            ; wyczyść rejestry MMX
```

Przykład 3. Rozmnożenie jednego bajtu na cały rejestr MMX.
(przeskocz rozmnażanie bajtu)

```
org 100h

        movq mm0, qword [wart1] ; mm0 = 00 00 00 00 00 00 00 33
                                ;  (33h = kod ASCII cyfry 3)

        punpcklbw mm0, mm0      ; do najmłodszego słowa włóż najmłodszy bajt
                                ; mm0 i najmłodszy bajt mm0 (czyli ten sam)
                                ; mm0 = 00 00 00 00 00 00 33 33

        punpcklwd mm0, mm0      ; do najmłodszego dworda włóż dwa razy
                                ; najmłodsze słowo mm0
                                ; mm0 = 00 00 00 00 33 33 33 33

        punpckldq mm0, mm0      ; do najmłodszego (i jedynego) qworda włóż 2x
                                ; najmłodszy dword mm0 obok siebie
                                ; mm0 = 33 33 33 33 33 33 33 33

        movq [wart2], mm0

        emms                    ; wyczyść rejestry MMX

        mov dx, wart2
        mov ah, 9
        int 21h         ; wypisz ciąg znaków wart2 zakończony znakiem dolara

        mov ax, 4c00h
        int 21h

wart1   db      "3"                     ; cyfra 3
        times 7 db 0                    ; i 7 bajtów zerowych

wart2:  times   8       db      2       ; 8 bajtów o wartości 2
```

```
        db "$"                          ; dla int 21h/ah=9
```

Kompilujemy, uruchamiamy i ... rzeczywiście na ekranie pojawia się upragnione osiem trójek!

Technologia MMX może być używana w wielu celach, ale jej najbardziej korzystną cechą jest właśnie równoległość wykonywanych czynności, dzięki czemu można oszczędzić czas procesora.

Technologia SSE
Streaming SIMD Extensions (SSE), Pentium III lub lepszy oraz najnowsze procesory AMD
Streaming SIMD Extensions 2 (SSE 2), Pentium 4 lub lepszy oraz AMD64
Streaming SIMD Extensions 3 (SSE 3), Xeon lub lepszy oraz AMD64

Krótko mówiąc, SSE jest dla MMX tym, czym FPU jest dla CPU. To znaczy, SSE przeprowadza równoległe operacje na liczbach ułamkowych.
SSE operuje już na całkowicie osobnych rejestrach nazwanych xmm0, ..., xmm7 po 128 bitów każdy. W trybie 64-bitowym dostępne jest dodatkowych 8 rejestrów: xmm8, ..., xmm15.
Prawie każda operacja związana z danymi w pamięci musi mieć te dane ustawione na 16-bajtowej granicy, czyli jej adres musi się dzielić przez 16. Inaczej generowane jest przerwanie (wyjątek).

SSE 2 różni się od SSE kilkoma nowymi instrukcjami konwersji ułamek-liczba całkowita oraz tym, że może operować na liczbach ułamkowych rozszerzonej precyzji (64 bity).

U AMD częściowo 3DNow! operuje na ułamkach, ale co najwyżej na dwóch gdyż są to rejestry odpowiadające MMX, a więc 64-bitowe. 3DNow! Pro jest odpowiednikiem SSE w procesorach AMD. Odpowiedniki SSE2 i SSE3 pojawił się w AMD64.

Instrukcje SSE (nie wszystkie będą wymienione):

- Przemieszczanie danych:
  - MOVAPS - move aligned packed single precision floating point values - przemieść ułożone (na granicy 16 bajtów) spakowane ułamki pojedynczej precyzji (4 sztuki po 32 bity)
  - MOVUPS - move unaligned (nieułożone) packed single precision floating point values
  - MOVSS - move scalar (1 sztuka, najmłodsze 32 bity rejestru) single precision floating point value

- Arytmetyczne:
  - ADDPS - add packed single precision floating point values = dodawanie czterech ułamków do czterech
  - ADDSS - add scalar single precision floating point values = dodawanie jednego ułamka do innego
  - MULPS - mnożenie spakowanych ułamków, równolegle, 4 pary
  - MULSS - mnożenie jednego ułamka przez inny
  - DIVPS - dzielenie spakowanych ułamków, równolegle, 4 pary
  - DIVSS - dzielenie jednego ułamka przez inny
  - obliczanie odwrotności ułamków, ich pierwiastków, odwrotności pierwiastków, znajdowanie wartości największej i najmniejszej

- Logiczne:
  - ANDPS - logiczne AND spakowanych wartości (ale oczywiście tym bardziej zadziała dla jednego ułamka w rejestrze)
  - ANDNPS - AND NOT (najpierw bitowe NOT pierwszego rejestru, potem jego bitowe AND z drugim rejestrem) dla spakowanych
  - ORPS - OR dla spakowanych
  - XORPS - XOR dla spakowanych

- Instrukcje porównania: CMPPS, CMPSS, (U)COMISS
- Instrukcje tasowania i rozpakowywania. Podobne działanie jak odpowiadające instrukcje MMX.
- Instrukcje konwersji ułamek->liczba całkowita i na odwrót.
- Instrukcje operujące na liczbach całkowitych 64-bitowych (lub 128-bitowych w SSE 2)

W większości przypadków instrukcje dodane w SSE 2 różnią się od powyższych ostatnią literą, którą jest D, co oznacza double precision, np. MOVAPD.

No i krótki przykładzik. Inne wersja procedury do kopiowania pamięci. Tym razem z SSE.
(przeskocz kopiowanie pamięci z SSE)

```
; Tylko jeśli ESI i EDI dzieli się przez 16! Inaczej używać MOVUPS.
                mov ebx, ecx    ; EBX = ilość bajtów
                and ebx, 127    ; EBX = reszta z dzielenia liczby bajtów
                                ; przez 128
                shr ecx, 7      ; ECX = liczba bajtów dzielona przez 128
        petla:
                ; kopiuj 128 bajtów spod [ESI] do rejestrów XMM0, ... XMM7
                movaps xmm0, [esi]
                movaps xmm1, [esi+16]
                movaps xmm2, [esi+32]
                movaps xmm3, [esi+48]
                movaps xmm4, [esi+64]
                movaps xmm5, [esi+80]
                movaps xmm6, [esi+96]
                movaps xmm7, [esi+112]

                ; kopiuj 128 bajtów z rejestrów XMM0, ... XMM7 do [EDI]
                movaps [edi    ], xmm0
                movaps [edi+16 ], xmm1
                movaps [edi+32 ], xmm2
                movaps [edi+48 ], xmm3
                movaps [edi+64 ], xmm4
                movaps [edi+80 ], xmm5
                movaps [edi+96 ], xmm6
                movaps [edi+112], xmm7

                add esi, 128    ; zwiększ indeksy do tablic o 128
                add edi, 128
                loop petla      ; działaj, dopóki ECX różne od 0

                mov ecx, ebx    ; ECX = liczba pozostałych bajtów
                cld             ; kierunek: do przodu
                rep movsb       ; resztkę kopiujemy po bajcie
```

Nie jest to ideał, przyznaję. Można było np. użyć instrukcji wspierających pobieranie danych z pamięci: PREFETCH.

A teraz coś innego: rozdzielanie danych. Przypuśćmy, że z jakiegoś urządzenia (lub pliku) czytamy bajty w postaci XYXYXYXYXY..., a my chcemy je rozdzielić na 2 tablice, zawierające tylko XXX... i YYY... (oczywiście bajty mogą mieć różne wartości, ale idea jest taka, że co drugi chcemy mieć w drugiej tablicy). Oto, jak można tego dokonać z użyciem SSE (składnia NASM/FASM, bo TASM w ogóle nie zna SSE). To jest tylko fragment programu.
(przeskocz rozdzielanie bajtów)

```
        mov     ah, 9
        mov     dx, dane_pocz
        int     21h

        mov     ah, 9
        mov     dx, dane
        int     21h                     ; wypisz dane początkowe

; FASM: movaps          xmm0, dqword [dane]
        movaps          xmm0, [dane]
        movaps          xmm1, xmm0
                ; XMM1=XMM0 = X1Y1 X2Y2 X3Y3 X4Y4 X5Y5 X6Y6 X7Y7 X8Y8

        packuswb        xmm0, xmm0
                ; XMM0 = Y1Y2 Y3Y4 Y5Y6 Y7Y8 Y1Y2 Y3Y4 Y5Y6 Y7Y8

        psrlw           xmm1, 8
                ; XMM1 = 0 X1 0 X2 0 X3 0 X4 0 X5 0 X6 0 X7 0 X8
        packuswb        xmm1, xmm1
                ; XMM1 = X1X2 X3X4 X5X6 X7X8 X1X2 X3X4 X5X6 X7X8

; FASM: movq    qword [dane2], xmm0
        movq    [dane2], xmm0   ; dane2 ani dane1 już nie mają adresu
                                ; podzielnego przez 16, więc nie można
                                ; użyć MOVAPS a my i tak
                                ; chcemy tylko 8 bajtów
; FASM: movq    qword [dane1], xmm1
        movq    [dane1], xmm1

        mov     ah, 9
        mov     dx, dane_kon
        int     21h

        mov     ah, 9
        mov     dx, dane1
        int     21h                     ; wypisz pierwsze dane końcowe

        mov     ah, 9
        mov     dx, dane2
        int     21h                     ; wypisz drugie dane końcowe

        mov     ax, 4c00h
        int     21h



align   16                              ; dla SSE

dane            db      "ABCDEFGHIJKLMNOP", 10, 13, "$"
dane1           db      0, 0, 0, 0, 0, 0, 0, 0, 13, 10, 9, "$"
dane2           db      0, 0, 0, 0, 0, 0, 0, 0, 13, 10, "$"
```

```
dane_pocz db "Program demonstrujacy SSE. Dane na poczatku: ", 10, 13, 9, "$"
dane_kon        db      "Dane na koncu: ", 10, 13, 9, "$"
```

Po szczegółowy opis wszystkich instrukcji odsyłam, jak zwykle do Intela i AMD

Instrukcje typu SIMD wspomagają szybkie przetwarzanie multimediów: dźwięku, obrazu. Omówienie każdej instrukcji w detalu jest niemożliwe i niepotrzebne, gdyż szczegółowe opisy są zamieszczone w książkach Intela lub AMD.

Miłej zabawy.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia:

1. Z dwóch zmiennych typu qword wczytaj do dwóch dowolnych rejestrów MMX (które najlepiej od razu skopiuj do innych), po czym wykonaj wszystkie możliwe dodawania i odejmowania. Wynik każdego zapisz w oddzielnej zmiennej typu qword.

2. Wykonaj operacje logiczne OR, AND i XOR na 64 bitach na raz (wczytaj je do rejestru MMX, wynik zapisz do pamięci).

3. Wczytajcie do rejestru MMX wartość szesnastkową 30 31 30 31 30 31 30 31, po czym wykonajcie różne operacje rozpakowania i pakowania, zapiszcie i wyświetlcie wynik jak każdy normalny ciąg znaków.

4. Wczytajcie do rejestrów XMM po 4 liczby ułamkowe dword, wykonajcie dodawania i odejmowania, po czym sprawdźcie wynik koprocesorem.

Jak pisać programy w języku asembler?
Część 7 - Porty, czyli łączność między procesorem a innymi urządzeniami.

Nie zastanawialiście się kiedyś, jak procesor komunikuje się z tymi wszystkimi urządzeniami, które znajdują się w komputerze?
Teraz zajmiemy się właśnie sposobem, w jaki procesor uzyskuje dostęp do urządzeń zewnętrznych (zewnętrznych dla procesora, niekoniecznie tych znajdujących się poza obudową komputera).

Procesor może porozumiewać z urządzeniami przez wydzielone obszary RAM-u. Te informacje można zobaczyć w Windows we właściwościach (prawie) każdego urządzenia, na karcie Zasoby, pod hasłem Zakres pamięci.
Mimo iż niekiedy sporo urządzeń zajmuje po jakimś skrawku pamięci, to jednak nie wszystkie. Głównym sposobem komunikacji ciągle pozostają porty (Zasoby -> Zakres wejścia/wyjścia).

Porty są to specjalne adresy, pod które procesor może wysyłać dane. Stanowią oddzielną strefę adresową (16-bitową, jak dalej zobaczymy, więc najwyższy teoretyczny numer portu wynosi 65535), choć czasami do niektórych portów można dostać się przez pamięć RAM. Są to porty mapowane do pamięci (memory-mapped), którymi nie będziemy się zajmować.

Lista przerwań Ralfa Brown'a (RBIL) zawiera plik ports.lst (który czasami trzeba osobno utworzyć - szczegóły w dokumentacji). W pliku tym (wygodnie jest go przeglądać np. programem ii.exe, dołączanym do RBIL) znajdują się szczegóły dotyczące całkiem sporej liczby portów odpowiadającym różnym urządzeniom. I tak, mamy np.

- Kontrolery DMA
- Programowalny kontroler przerwań (Programmable Interrupt Controller, PIC)
- Programowalny czasomierz (Programmable Interval Timer, PIT)
- Kontroler klawiatury
- CMOS
- SoundBlaster i inne karty dźwiękowe
- Karty graficzne i inne karty rozszerzeń (np. modem)
- Porty COM, LPT
- Kontrolery dysków twardych
- i wiele, wiele innych...

No dobrze, wiemy co ma który port i tak dalej, ale jak z tego skorzystać?

Procesor posiada 2 instrukcje przeznaczone specjalnie do tego celu. Są to IN i OUT.
Ich podstawowa składnia wygląda tak:

```
in al/ax/eax, numer_portu
out numer_portu, al/ax/eax
```

Uwagi:

1. Jeśli numer_portu > 255, to w jego miejsce musimy użyć rejestru DX
2. Do operacji na portach nie można używać innych rejestrów niż AL, AX lub EAX.
3. Wczytane ilości bajtów zależą od rejestru, a ich pochodzenie - od rodzaju portu:

- jeśli port `num` jest 8-bitowy, to
  `IN AL, num` wczyta 1 bajt z portu o numerze `num`
  `IN AX, num` wczyta 1 bajt z portu `num` (do AL) i 1 bajt z portu `num+1` (do AH)
  `IN EAX, num` wczyta po 1 bajcie z portów `num`, `num+1`, `num+2` i `num+3` i umieści w odpowiednich częściach rejestru EAX (od najmłodszej)
- jeśli port `num` jest 16-bitowy, to
  `IN AX, num` wczyta 2 bajty z portu o numerze `num`
  `IN EAX, num` wczyta 2 bajty z portu o numerze `num` i 2 bajty z portu o numerze `num+1`
- jeśli port `num` jest 32-bitowy, to
  `IN EAX, num` wczyta 4 bajty z portu o numerze `num`

4. Podobne uwagi mają zastosowane dla instrukcji `OUT`

Teraz byłaby dobra pora na jakiś przykład (mając na uwadze dobro swojego komputera, NIE URUCHAMIAJ PONIŻSZYCH KOMEND):

```
in      al, 0   ; pobierz bajt z portu 0
out     60h, eax; wyślij 4 bajty na port 60h

mov     dx, 300 ; 300 > 255, więc musimy użyć DX
in      al, dx  ; wczytaj 1 bajt z portu 300
out     dx, ax  ; wyślij 2 bajty na port 300
```

Nie rozpisywałem się tutaj za bardzo, bo ciekawsze i bardziej użyteczne przykłady znajdują się w moich mini-kursach (programowanie diód na klawiaturze, programowanie głośniczka).

Jak już wspomniałem wcześniej, porty umożliwiają dostęp do wielu urządzeń. Jeśli więc chcesz poeksperymentować, nie wybieraj portów zajętych np. przez kontrolery dysków twardych, gdyż zabawa portami może prowadzić do utraty danych lub uszkodzenia sprzętu.
Dlatego właśnie w nowszych systemach operacyjnych (tych pracujących w trybie chronionym, jak np. Linux czy Windows) dostęp do portów jest zabroniony dla zwykłych aplikacji (o prawa dostępu do portów trzeba prosić system operacyjny).
Jak więc działają np. stare DOS-owe gry? Odpowiedź jest prosta: nie działają w trybie chronionym. System uruchamia je w trybie udającym tryb rzeczywisty (taki, w jakim pracuje DOS), co umożliwia im pełną kontrolę nad sprzętem.
Wszystkie programy, które dotąd pisaliśmy też uruchamiają się w tym samym trybie, więc mają swobodę w dostępie np. do głośniczka czy karty dźwiękowej. Co innego programy pisane w nowszych kompilatorach np. języka C - tutaj może już być problem. Ale na szczęście my nie musimy się tym martwić...

Jeszcze jeden ciekawy przykład - używanie CMOSu. CMOS ma 2 podstawowe porty: 70h, zwany portem adresu i 71h, zwany portem danych. Operacje są proste i składają się z 2 kroków:

1. Na port 70h wyślij numer komórki (1 bajt), którą chcesz odczytać lub zmienić. Polecam plik cmos.lst z RBIL, zawierający szczegółowy opis komórek CMOS-u
2. Na port 71h wyślij dane, jeśli chcesz zmienić komórkę lub z portu 71h odczytaj dane, jeśli chcesz odczytać komórkę

Oto przykład. Odczytamy tutaj godzinę w komputerze, a konkretnie - sekundy:

```
mov     al, 0
```

```
        out     70h, al
        out     0edh, al
        in      al, 71h
```

Wszystko jasne, oprócz tej dziwnej komendy: `OUT 0edh, al`. Jak spojrzycie w ports.lst, ten port jest (jako jeden z dwóch) opisany jako dummy port for delay. Czyli nic nie robi, poza opóźnianiem.
Po co to komu, pytacie?
Przy współczesnych częstotliwościach procesorów, CMOS (jak z resztą i inne układy) może po prostu nie zdążyć z odpowiedzią na naszą prośbę, gdyż od chwili wysłania numeru komórki do chwili odczytania danych mija za mało czasu. Dlatego robimy sobie przerwę na kilkanaście taktów zegara procesora.
Kiedyś między operacjami na CMOSie zwykło się pisać `JMP short $+2`, co też oczywiście nie robiło nic, poza zajmowaniem czasu (to jest po prostu skok o 2 bajty do przodu od miejsca, gdzie zaczyna się ta 2-bajtowa instrukcja, czyli skok do następnej instrukcji), ale ta operacja już nie trwa wystarczająco długo, aby ją dalej stosować.

W dzisiejszych czasach porty już nie są tak często używane, jak były kiedyś. Jest to spowodowane przede wszystkim wspomnianym trybem chronionym oraz tym, że wszystkie urządzenia mają już własne sterowniki (mające większe uprawnienia do manipulowania sprzętem), które zajmują się wszystkimi operacjami I/O. Programista musi jedynie uruchomić odpowiednią funkcję i niczym się nie przejmować.
Dawniej, portów używało się do sterowania grafiką czy wysyłania dźwięków przez głośniczek lub karty dźwiękowe. Teraz tym wszystkim zajmuje się za nas system operacyjny. Dzięki temu możemy się uchronić przed zniszczeniem sprzętu.

Mimo iż rola portów już nie jest taka duża, zdecydowałem się je omówić, gdyż po prostu czasami mogą się przydać. I nie będziecie zdziwieni, gdy ktoś pokaże wam kod z jakimiś dziwnymi instrukcjami IN i OUT...

Szczegóły dotyczące instrukcji dostępu do portów także znajdziecie, jak zwykle, u AMD i Intela

Miłej zabawy.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia:

1. Zapoznaj się z opisem CMOSu i napisz program, który wyświetli bieżący czas w postaci gg:mm:ss (z dwukropkami). Pamiętaj o umieszczeniu opóźnień w swoim programie.

Jak pisać programy w języku asembler?
Część 8 - Bardziej zaawansowane programy, czyli zobaczmy, co ten język naprawdę potrafi.

No cóż, nie jesteśmy już amatorami i przyszła pora, aby przyjrzeć się temu, w czym asembler wprost błyszczy: algorytmy intensywne obliczeniowo. Specjalnie na potrzeby tego kursu napisałem następujący programik. Zaprezentuję w nim kilka sztuczek i pokażę, do jakich rozmiarów (tutaj: 2 instrukcje) można ścisnąć główną pętlę programu.
Oto ten programik:
(przeskocz program obliczający sumę liczb)

```
; Program liczący sumę liczb od 1 do liczby wpisanej z klawiatury
;
; Autor: Bogdan D.
;
; kompilacja NASM:
;   nasm -O999 -o ciag_ar.obj -f obj ciag_ar.asm
;   alink ciag_ar.obj  bibl\lib\bibldos.lib -c- -oEXE -m-
; kompilacja FASM:
;   fasm ciag_ar.asm ciag_ar.obj
;   alink ciag_ar.obj  bibl\lib\bibldos.lib -c- -entry _start -oEXE -m-



%include "bibl\incl\dosbios\nasm\std_bibl.inc"
%include "bibl\incl\dosbios\nasm\do_nasma.inc"

.stack 400h

section .text

; FASM:
; format coff
; include "bibl\incl\dosbios\fasm\std_bibl.inc"
; use16
; public _start

_start:
..start:
        pisz
        db      "Program liczy sume liczb od 1 do podanej liczby.",cr,lf
        db      "Podaj liczbe calkowita: ",0

        we32e           ; pobieramy z klawiatury liczbę do rejestru EAX
        jnc     liczba_ok       ; flaga CF=1 oznacza błąd


blad:
        pisz
        db      cr, lf, "Zla liczba!",0

        wyjscie 1                       ; mov ax, 4c01h / int 21h

liczba_ok:
        test    eax, eax                ; jeśli EAX=0, to też błąd
        jz      blad

        mov     ebx, eax                ; zachowaj liczbę. EBX=n
        xor     edx, edx                ; EDX = nasza suma
        mov     ecx, 1
```

```
petla:
        add     edx, eax                ; dodaj liczbę do sumy
        sub     eax, ecx                ; odejmij 1 od liczby
        jnz     petla                   ; liczba różna od zera?
                                        ; to jeszcze raz dodajemy

        pisz
        db      cr, lf, "Wynik z sumowania 1+2+3+...+n= ",0

        mov     eax, edx                ; EAX = wynik
        pisz32e                         ; wypisz EAX

        mov     eax, ebx                ; przywrócenie liczby
        add     eax, 1                  ; EAX = n+1
        mul     ebx                     ; EDX:EAX = EAX*EBX = n*(n+1)

        shr     edx, 1
        rcr     eax, 1                  ; EDX:EAX = EDX:EAX/2

        pisz
        db      cr, lf, "Wynik ze wzoru: n(n+1)/2= ",0

        pisz64          ; wypisuje na ekranie 64-bitową liczbę całkowitą
                        ; z EDX:EAX


        wyjscie 0                       ; mov ax, 4c00h / int 21h
```

Jak widać, nie jest on ogromny, a jednak spełnia swoje zadanie. Teraz przeanalizujemy ten krótki programik:

- Komentarz nagłówkowy.
  Mówi, co program robi oraz kto jest jego autorem. Może zawierać informacje o wersji programu, o niestandardowym sposobie kompilacji/uruchomienia i wiele innych szczegółów.

- pisz, we32e, pisz32e oraz pisz64.
  To są makra uruchamiające procedury z mojej biblioteki. Używam ich, bo są sprawdzone i nie muszę ciągle umieszczać kodu tych procedur w programie.

- Makro wyjscie zawiera w sobie kod wyjścia z programu, napisany obok.

- `test rej, rej / jz ... / jnz ...`
  Instrukcja `TEST` jest szybsza niż `CMP rej, 0` i nie zmienia zawartości rejestru, w przeciwieństwie do `OR`. Jest to najszybszy sposób na sprawdzenie, czy wartość rejestru wynosi 0.

- Pętla główna.
  Jak widać, najpierw do sumy dodajemy n, potem n-1, potem n-2, i na końcu 1. Umożliwiło to znaczne skrócenie kodu pętli, a więc zwiększenie jej szybkości. Napisanie `SUB EAX, ECX` zamiast `SUB EAX, 1` skraca rozmiar instrukcji i powoduje jej przyspieszenie, gdyż dzięki temu w samej pętli procesor operuje już tylko na rejestrach.

- `SHR EDX, 1 / RCR EAX, 1`
  Wynik musimy podzielić przez 2, zgodnie ze wzorem. Niestety, nie ma instrukcji `SHR` dla 64 bitów. Więc trzeba ten brak jakoś obejść. Najpierw, `SHR EDX, 1` dzieli EDX przez 2, a bit 0 ląduje we fladze CF. Teraz, `RCR EAX, 1` (rotate THROUGH CARRY) wartość CF (czyli stary bit 0 EDX) umieści w bicie 31 EAX. I o to chodziło!

Poniższy programik też napisałem dla tego kursu. Ma on pokazać złożone sposoby adresowania oraz instrukcje warunkowego przesunięcia (CMOV..):
(przeskocz program z macierzą)

```
; Program wczytuje od użytkownika macierz 3x3, po czym znajduje
; element największy i najmniejszy
;
; Autor: Bogdan D.
;
; kompilacja:
;   nasm -O999 -o macierze.obj -f obj macierze.asm
;   alink macierze.obj  bibl\lib\bibldos.lib -c- -oEXE -m-
; kompilacja FASM:
;   fasm macierze.asm macierze.obj
;   alink macierze.obj  bibl\lib\bibldos.lib -c- -entry _start -oEXE -m-

%include "bibl\incl\dosbios\nasm\std_bibl.inc"
%include "bibl\incl\dosbios\nasm\do_nasma.inc"

%define rozmiar 3

.stack 400h

section .text

; FASM:
; format coff
; include "bibl\incl\dosbios\fasm\std_bibl.inc"
; use16
; rozmiar = 3
; public _start

_start:
..start:
        mov     ax, cs
        mov     ds, ax  ; DS musi być = CS, bo inaczej zapisywalibyśmy
                        ; nie tam, gdzie trzeba, a macierz jest w
                        ; segmencie kodu.

        pisz
        db      "Prosze podac 9 elementow macierzy,"
        db      cr,lf," a ja znajde najwiekszy i najmniejszy.",0

        xor     edx, edx                        ; ECX = 0
        mov     ebx, macierz

petla_wczyt:
        pisz
        db      cr, lf, "Prosze podac element nr ", 0
        mov     eax, edx
        add     eax, 1
        pisz32e                         ; wypisz numer elementu

        mov     ax, (0eh << 8) | ":"            ; wypisz dwukropek
; FASM:
```

```
;       mov     ax, (0eh shl 8) or ":"

        int     10h
        mov     al, spc                 ; wypisz spację
        int     10h

        we32e                           ; wczytaj element
        jc      blad
        mov     [ebx+4*edx], eax        ; umieść w macierzy

        add     edx, 1                  ; zwiększ licznik elementów
                                        ; i równocześnie pozycję w macierzy

        cmp     edx, rozmiar*rozmiar
        jb      petla_wczyt

        jmp     wczyt_gotowe

blad:
        pisz
        db      cr, lf, "Zla liczba!",0

        wyjscie 1


wczyt_gotowe:
                                        ; EBP = max, EDI = min

        mov     ebp, [ebx]
        mov     edi, [ebx]              ; pierwszy element
        mov     edx, 1
        mov     eax, 1
        mov     esi, rozmiar*rozmiar

znajdz_max_min:
        mov     ecx, [ ebx + 4*edx ]
        cmp     ebp, ecx        ; EBP < macierz[edx] ?
        cmovb   ebp, ecx        ; jeśli tak, to EBP = macierz[edx]

        cmp     edi, ecx        ; EDI > macierz[edx] ?
        cmova   edi, ecx        ; jeśli tak, to EDI = macierz[edx]

        add     edx, eax
        cmp     edx, esi
        jb      znajdz_max_min

        pisz
        db      cr, lf, "Najwiekszy element: ",0
        mov     eax, ebp
        pisz32e

        pisz
        db      cr, lf, "Najmniejszy element: ",0
        mov     eax, edi
        pisz32e


        wyjscie 0



macierz:        times   rozmiar*rozmiar         dd 0
```

Przypatrzmy się teraz miejscom, gdzie można zwątpić w swoje umiejętności:

- `mov ax, (0eh << 8) | ":"`
  Znaki << odpowiadają instrukcji `SHL`, a znak | odpowiada instrukcji `OR`. Mamy więc: `0eh shl 8`, czyli 0e00h. Robimy `OR` z dwukropkiem (3ah) i mamy AX=0e3ah. Uruchamiając przerwanie 10h, na ekranie otrzymujemy dwukropek.

- `mov [ebx+4*edx], eax`
  EBX = adres macierzy. EDX = 0, 1, 2, ..., rozmiar*rozmiar=9. Elementy macierzy mają rozmiar po 4 bajty każdy, stąd EDX mnożymy przez 4. Innymi słowy, pierwszy EAX idzie do [ebx+4*0]=[ebx], drugi do [ebx+4] (na 2 miejsce macierzy), trzeci do [ebx+8] itd.

- Fragment kodu:

```
mov     ecx, [ ebx + 4*edx ]
cmp     ebp, ecx        ; EBP < macierz[edx] ?
cmovb   ebp, ecx        ; jeśli tak, to EBP = macierz[edx]

cmp     edi, ecx        ; EDI > macierz[edx] ?
cmova   edi, ecx        ; jeśli tak, to EDI = macierz[edx]

add     edx, eax
cmp     edx, esi
jb      znajdz_max_min
```

  Najpierw, do ECX idzie aktualny element. Potem porównujemy EBX z tym elementem i, gdy EBP < ECX, kopiujemy ECX do EBP. Do tego właśnie służy instrukcja `CMOVB` (Conditional MOVe if Below). Instrukcje z rodziny `(F)CMOV` umożliwiają pozbywanie się skoków warunkowych, które obniżają wydajność kodu.
  Podobnie, porównujemy EDI=min z ECX.
  Potem zwiększamy EDX o 1 i sprawdzamy, czy nie przeszliśmy przez każdy element macierzy.

Powyższy program trudno nazwać intensywnym obliczeniowo, bo ograniczyłem rozmiar macierzy do 3x3. Ale to był tylko przykład. Prawdziwe programy mogą operować na macierzach zawierających miliony elementów. Podobny program napisany w HLLu jak C czy Pascal po prostu zaliczyłby się na śmierć.

Teraz pokażę program, który ewoluował od nieoptymalnej formy (zawierającej np. więcej skoków warunkowych w głównej pętli oraz inne nieoptymalne instrukcje) do czegoś takiego:
(przeskocz program znajdujący liczby magiczne)

```
; L_mag.asm
;
; Program wyszukuje liczby, które są sumą swoich dzielników
;
; Autor: Bogdan D.
; kontakt: bogdandr (at) op (dot) pl
;
; nasm -O999 -o l_mag.com -f bin l_mag.asm
; fasm l_mag.asm l_mag.com


org 100h
```

```
start:
        mov     ax,cs
        mov     ebx,1                   ; liczba początkowa

        mov     ebp,1
        mov     ds,ax

align 16
start2:
        mov     esi,ebx                 ; ESI = liczba

        mov     ecx,ebp                 ; EBP = 1
        shr     esi,1                   ; zachowanie połowy liczby

        xor     edi,edi                 ; suma dzielników=0

align 16
petla:
        xor     edx,edx                 ; dla dzielenia
        nop
        cmp     ecx,esi                 ; czy ECX (dzielnik)>liczba/2?
        mov     eax,ebx                 ; przywrócenie liczby do dzielenia
        nop
        ja      dalej2                  ; Jeśli ECX > ESI, to koniec
                                        ; dzielenia tej liczby

        nop
        div     ecx                     ; EAX = EDX:EAX / ECX, EDX=reszta

        nop
        nop
        add     ecx,ebp                 ; zwiększamy dzielnik o 1
        nop

        test    edx,edx                 ; czy ECX jest dzielnikiem?
                                        ; (czy EDX=reszta=0?)
        nop
        nop
        jnz     petla                   ; nie? Dzielimy przez następną liczbę

                                        ; tak?
        lea     edi,[edi+ecx-1]         ; dodajemy dzielnik do sumy, nie
                                        ; sprawdzamy na przepełnienie.
                                        ; ECX-1 bo dodaliśmy EBP=1 do
                                        ; ECX po DIV.

        jmp     short petla             ; dzielimy przez kolejną liczbę
        ud2


align 16
dalej2:
        cmp     ebx,edi                 ; czy to ta liczba?
                                        ; (czy liczba=suma dzielników)
        mov     ah,9
        mov     edx,jest
        jne     nie                     ; nie

        int     21h                     ; tak - napis  "znaleziono "

        mov     eax,ebx
        call    pl                      ; wypisujemy liczbe
```

```
align 16
nie:
        mov     ah,1
        int     16h
        nop
        jnz     klaw

        cmp     ebx,0ffffffffh          ; czy juz koniec zakresu?
        nop
        je      koniec                  ; tak

        add     ebx,ebp                 ; nie, zwiększamy liczbe badana o
                                        ; jeden i od początku
        nop
        jmp     short start2
        ud2


align 16
klaw:
        xor     ah,ah
        int     16h
koniec:
        mov     ah,9
        mov     edx,meta
        nop
        int     21h                     ; napis  "koniec "

        mov     eax,ebx
        call    pl              ; wypisujemy ostatnią sprawdzoną liczbę
spr:                                    ; czekamy na klawisz
        mov     ah,1
        nop
        int     16h
        jz      spr

        xor     ah,ah
        int     16h

        mov     ax,4c00h
        int     21h
        ud2

align 16
pc:                                     ; wypisuje cyfrę w AL
        mov     ah,0eh
        push    ebp
        or      al,30h
        int     10h
        pop     ebp
        ret
        ud2

align 16
pl:                             ; wypisuje liczbę dziesięciocyfrową w EAX
        mov     ecx,1000000000
        xor     edx,edx
        div     ecx
        call    pc

        mov     eax,edx
```

```
        mov     ecx,100000000
        xor     edx,edx
        div     ecx
        call    pc

        mov     eax,edx
        mov     ecx,10000000
        xor     edx,edx
        div     ecx
        call    pc

        mov     eax,edx
        mov     ecx,1000000
        xor     edx,edx
        div     ecx
        call    pc

        mov     eax,edx
        mov     ecx,100000
        xor     edx,edx
        div     ecx
        call    pc

        mov     eax,edx
        mov     ecx,10000
        xor     edx,edx
        div     ecx
        call    pc

        mov     eax,edx
        xor     edx,edx
        mov     ecx,1000
        div     ecx
        call    pc

        mov     eax,edx
        mov     cl,100
        div     cl
        mov     ch,ah
        call    pc

        mov     al,ch
        xor     ah,ah
        mov     cl,10
        div     cl
        mov     ch,ah
        call    pc

        mov     al,ch
        call    pc
        ret
        ud2

align 4
jest    db      10,13,"Znaleziono: $"
meta    db      10,13,"Koniec. ostatnia liczba: $"
```

A oto analiza:

- Pętla główna:
  Dziel EBX przez kolejne przypuszczalne dzielniki. Jeśli trafisz na prawdziwy dzielnik

(reszta=EDX=0), to dodaj go do sumy, która jest w EDI.
Unikałem ustawiania obok siebie takich instrukcji, które zależą od siebie, jak np. `CMP / JA` czy `DIV / ADD`

- Nie za dużo tych `NOP`'ów?
  Nie. Zamiast czekać na wynik poprzednich instrukcji, procesor zajmuje się... robieniem niczego. Ale jednak sie zajmuje. Współczesne procesory potrafią wykonywać wiele niezależnych instrukcji praktycznie równolegle. Więc w czasie, jak procesor czeka na wykonanie poprzednich instrukcji, może równolegle wykonywać NOPy. Zwiększa to przepustowość, utrzymuje układy dekodujące w ciągłej pracy, kolejka instrukcji oczekujących na wykonanie nie jest pusta.

- Co robi instrukcja `lea edi,[edi+ecx-1]` ?
  LEA - Load Effective Address. Do rejestru EDI załaduj ADRES (elementu, którego) ADRES wynosi EDI+ECX-1. Czyli, w paskalowej składni: EDI := EDI+ECX-1. Do EDI dodajemy znaleziony dzielnik. Musimy odjąć 1, bo wcześniej (po dzieleniu) zwiększyliśmy ECX o 1.

- Co robi instrukcja `UD2` i czemu jest umieszczona po instrukcjach JMP ?
  Ta instrukcja (UnDefined opcode 2) wywołuje wyjątek wykonania nieprawidłowej instrukcji przez procesor. Umieściłem ją w takich miejscach, żeby nigdy nie była wykonana.
  Po co ona w ogóle jest w tym programie w takich miejscach?
  Ma ona interesującą właściwość: powstrzymuje jednostki dekodujące instrukcje od dalszej pracy. Po co dekodować instrukcje, które i tak nie będą wykonane (bo były po skoku bezwarunkowym) ? Strata czasu.

- Po co ciągle `align 16` ?
  Te dyrektywy są tylko przed etykietami, które są celem skoku. Ustawianie kodu od adresu, który dzieli się przez 16 może ułatwić procesorowi umieszczenie go w całej jednej linii pamięci podręcznej (cache). Mniej instrukcji musi być pobieranych z pamięci (bo te, które są najczęściej wykonywane już są w cache), więc szybkość dekodowania wzrasta. Układania kodu i danych zwiększa ogólną wydajność programu

---

O tych wszystkich sztuczkach, które tu zastosowałem, można przeczytać w podręcznikach dotyczących optymalizacji programów, wydanych zarówno przez Intel, jak i AMD (u AMD są też wymienione sztuczki, których można użyć do optymalizacji programów napisanych w języku C).
Podaję adresy (te same co zwykle): AMD, Intel

Życzę ciekawej lektury i miłej zabawy.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia:

1. Napisz program obliczający Największy Wspólny Dzielnik i Najmniejszą Wspólną Wielokrotność

dwóch liczb wiedząc, że:
NWD(a,b) = NWD(b, reszta z dzielenia a przez b) i NWD(n,0)=n (algorytm Euklidesa)
NWW(a,b) = a*b / NWD(a,b)

2. Napisz program rozkładający daną liczbę na czynniki pierwsze (liczba może być umieszczona w kodzie programu).

Jak pisać programy w języku asembler?
Część 9 - Narzędzia programisty, czyli co może nam pomagać w programowaniu.

Debugery

(przeskocz debugery)

Wszystkim się może zdarzyć, że nieustanne, wielogodzinne gapienie się w kod programu nic nie daje i program ciągle nie chce nam działać. Wtedy z pomocą przychodzą debugery. W tej części zaprezentuję kilka wartych uwagi programów tego typu. Nie będę szczegółowo mówił, jak ich używać, bo zwykle posiadają albo menu, albo dokumentację czy inną pomoc.

Debuggery programów DOS-owych:
(przeskocz DOSowe debugery)

1. Watcom Debugger (WD).
   Rozpowszechniany z pakietem OpenWatcom, WD jest najlepszym z darmowych debugerów. Umożliwia wyświetlanie rejestrów procesora, flag, koprocesora, MMX i SSE, śledzenie wartosci zmiennych, stawianie pułapek (breakpoint, klawisz F9), podgląd wyjścia programu (klawisz F4), wykonywanie do kursora i wiele innych działań. Posiada obsługę myszy. Pozwala debugować wszystko, co może być wytworzone przez pakiet OpenWatcom - .com, .exe (MZ i LE) i wiele innych.

2. Turbo Debugger (TD) firmy Borland.
   Jeden z najlepszych dostępnych. Niestety, nie darmowy. Umożliwia wyświetlanie rejestrów 16/32-bit, rejestrów koprocesora, stosu i pewnych regionów pamięci (standardowo DS:0000) oraz flag i daje możliwość modyfikacji ich wszystkich. Można obserwować zmienne oraz to, co się dzieje na ekranie poza debuggerem. Gdy testowałem program działający w trybie graficznym, to po każdej komendzie monitor przełączał się w ten tryb, po czym wracał do debuggera, co umożliwia podglądanie naprawdę każdego kroku. Niestety, zrobienie czegokolwiek poza przejściem do kolejnej instrukcji wymagało przejścia w tryb pełnego ekranu, gdyż okienko w Windows nie za bardzo było potem odświeżane... Niestety, TD nie ma on pojęcia o technologiach takich, jak MMX czy SSE (jest po prostu za stary). Posiada wiele opcji debugowania: trace-over, step-into, execute-to, animate, ...
   Nadaje się do plików .com/.exe (najnowsza wersja 5.5 obsługuje tylko pliki 32-bitowe). Można go ściągnąć ze stron Borlanda po zarejestrowaniu się.

3. D86.
   Darmowy obsługuje tylko procesory 16-bitowe (brak rozszerzonych rejestrów), ale można podglądać rejestry, flagi, pamięć i koprocesor. D86 jest rozprowadzany razem z A86, darmowym kompilatorem języka asembler, i rozpoznaje symbole (nazwy zmiennych itp.), które A86 umieszcza w plikach .sym, co ułatwia proces debugowania.
   Posiada pełną dokumentację. Pomoc w sprawie klawiszy jest w każdej chwili dostępna pod kombinacją klawiszy Alt-F10.
   Niestety, miałem problem z przeglądaniem aplikacji graficznej: po uruchomieniu trybu graficznego nie mogłem wrócić do ekranu debuggera i musiałem poczekać, aż program się zakończy. D86 zna instrukcje koprocesora.
   Płatna wersja, D386, zna MMX, SSE i 3DNow!.

4. Codeview (CV) firmy Microsoft.
   Moje doświadczenie z tym debuggerem jest krótkie, gdyż nie spełnił on wszystkich moich oczekiwań.

Po uruchomieniu od razu trzeba otworzyć jakiś program (i z tego co pamiętam, aby otworzyć inny program, trzeba wyjść z całego debuggera. Te programy, które chciałem otworzyć, CV otwierał tak długo, że można było pomyśleć, że komputer się zawiesił...
Nawet chyba nie jest rozprowadzany osobno, tylko razem z MASMem (za darmo). Trzeba przejść długi proces instalacji, ustawiać zmienne środowiska, ...

5. Insight
   Natrafiłem na niego, gdy szukałem czegoś (nowszego), co mogło by zastąpić Turbo Debuggera. Wyglądem nawet przypomina TD, ale ma kilka wad. Pierwszą jest brak rozpoznawania instrukcji koprocesora (wszystkie dekoduje jako ESC + coś tam). O MMX nie ma co myśleć. Drugą wadą, którą znalazłem jest to, że po wejściu w tryb graficzny okienko Tryb MS-DOS z debuggerem przestaje się odświeżać i trzeba się przełączyć na pełny ekran.
   Ale jako-tako, działa. Posiada opcje step-over, trace-into, animate. Można zmieniać wartości rejestrów.

6. Advanced Fullscreen Debugger
   Nawet ładne narzędzie. Pozwala w jednej chwili oglądać kod, stos, rejestry i 2 bloki pamięci (standardowo ustawiane na DS:0000). Obsługa jest prosta: na dole ekranu jest pokazane, co robią klawisze funkcyjne, ale można też wpisywać komendy. Bardzo pomocne jest to, że po wpisaniu pierwszej literki pojawiają się dostępne komendy zaczynające się od niej. Niestety, ma te dwa problemy, które ma Insight: po uruchomieniu trybu graficznego okienku z debuggerem przestaje być odświeżane (trzeba się przełączyć na pełny ekran) i nie rozpoznaje instrukcji koprocesora.

7. TRW2000
   Umie debugować programy typu .com i .exe. Jednak coś jest nie tak z obsługą myszy a praca z nim nie jest zbyt wygodna. Strona domowa TRW: www.hktk.com/soft/soft_tools/trw_1.html

Debuggery programów dla Windows:
(przeskocz windowsowe debuggery)

1. GoBug
   Część pakietu GoDevTools (www.godevtool.com). Poza nim są m.in kompilator języka asembler oraz resource compiler. Wszystko to przydać się może przy pisaniu programów dla Windows. Ja osobiście używam FASMa, ale moim debuggerem jest właśnie GoBug. Ma miły dla oka wygląd, zna rejestry FPU, MMX. Wyświetla kod programu, stan rejestrów, oraz stos względem ESP oraz EBP. Obsługuje wieloprocesowość oraz symbole użyte w kodzie, o ile znajdzie odpowiedni plik z nimi. Po przytrzymaniu prawego klawisza myszki na instrukcji pojawiają się bajty zapisane szesnastkowo, które się na tą instrukcję składają.
   GoBug rozpoznaje uruchomienia procedur Windows-owych z bibliotek DLL.
   Dobre narzędzie.

2. Olly Debugger
   Można go za darmo ściągnąć z jego strony domowej: ollydbg.de. Wygląda bardziej profesjonalnie niż GoBug i podobnie jak on, rozpoznaje uruchomienia procedur systemowych. Stos jest wyświetlany tylko względem ESP. Wyświetla rejestry i flagi. Stara się łączyć umieszczanie parametrów na stosie z uruchomieniem procedury, ale nie zawsze to wychodzi. Przewijając okienko z kodem niektóre instrukcje mogą się nagle zmieniać. Obsługa jest według mnie trudniejsza. Czcionka instrukcji jest mniejsza, co jeszcze bardziej utrudnia ich rozczytanie. Bardziej nie wnikałem w jego obsługę.
   W tej chwili bardziej polecam GoBug niż OllyDbg.

Wiem, że nie wszyscy od razu z entuzjazmem rzucą się do ściągania i testowania przedstawionych wyżej programów i do debugowania własnych.
Niektórzy mogą uważać, że debugger nie jest im potrzebny. Może i tak być, ale nie zawsze i nie u wszystkich. Czasem (zwykle po długim sterczeniu przed ekranem) przychodzi chęć do użycia czegoś, co tak bardzo może ułatwić nam wszystkim życie.
Pomyślcie, że gdyby nie było debuggerów, znajdowanie błędów w programie musielibyście pozostawić swojej nie zawsze wyćwiczonej wyobraźni. Dlatego zachęcam Was do korzystania z programów tego typu (tylko tych posiadanych legalnie, oczywiście).

Środowiska programowania, edytory i disasemblery/hex-edytory

(przeskocz ten dział)

Środowisko programowania (Integrated Development Environment, IDE) to, jak wszyscy wiemy, program, który umożliwia edycję kodu, jego kompilację i uruchamianie programu wynikowego. Znanych jest wiele IDE dla języków wysokiego poziomu, ale język asembler też ma kilka swoich:
(przeskocz środowiska)

- RadASM - radasm.visualasembler.com - środowisko programistyczne obsługujące wiele kompilatorów (MASM, TASM, NASM, HLA).
- NasmIDE: uk.geocities.com/rob_anderton
- TasmIDE: creamelana.tripod.com/Tasm/TasmIDE.htm
- Środowisko dla FASMa (wbudowane w kompilator w wersji GUI): flatasembler.net oraz Fresh: fresh.flatasembler.net
- WinAsm Studio: code4u.net/winasm
- AsmEdit: asmedit.massmind.org (dla MASMa)

Jeśli mimo tego ktoś nie chce lub nie lubi używać IDE, zawsze może skorzystać z któregoś ze zwykłych edytorów. Przedstawione poniżej propozycje to co prawda nie muszą być edytorami napisanymi specjalnie do programowania w asemblerze, ale może coś Wam przypadnie do gustu:
(przeskocz edytory)

- Programmer's File Editor: www.movsd.com/tools.htm
- Quick Editor: www.movsd.com/qed.htm
- The Gun: www.movsd.com/thegun.htm
- HTE: hte.sf.net

Jeśli nie podoba się Wam żaden z wymienionych, to możecie wejść na stronę The Free Country.com - edytory, gdzie przedstawionych jest wiele edytorów dla programistów.

Kolejną przydatną rzeczą może okazać się disasembler lub hex-edytor. Jest to program, który podobnie jak debugger czyta plik i ewentualnie tłumaczy zawarte w nim bajty na instrukcje asemblera, jednak bez możliwości uruchomienia czytanego programu.
Disasemblery mogą być przydatne w wielu sytuacjach, np. gdy chcemy modyfikować pojedyncze bajty po kompilacji programu, zobaczyć adresy zmiennych, itp.
Oto kilka przykładów programów tego typu:
(przeskocz hex-edytory)

- XEdit: www.ircdb.org
- b2hedit: www.movsd.com/tools.htm
- Biew: biew.sf.net

I ponownie, jeśli nie spodoba się Wam żaden z wymienionych, to możecie wejść na stronę The Free Country.com - disasemblery, aby poszukać wśród pokazanych tam programów czegoś dla siebie.

Programy typu MAKE

Programy typu MAKE (np. GNU MAKE) służą do automatyzacji budowania dużych i małych projektów. Taki program działa dość prosto: uruchamiamy go, a on szuka pliku o nazwie Makefile w bieżącym katalogu i wykonuje komendy w nim zawarte. Teraz zajmiemy się omówieniem podstaw składni pliku Makefile.

W pliku takim są zadania do wykonania. Nazwa zadania zaczyna się w pierwszej kolumnie, kończy dwukropkiem. Po dwukropku są podane nazwy zadań (lub plików) , od wykonania których zależy wykonanie tego zadania. W kolejnych wierszach są komendy służące do wykonania danego zadania.
UWAGA: komendy NIE MOGĄ zaczynać się od pierwszej kolumny! Należy je pisać je po jednym tabulatorze (ale nie wolno zamiast tabulatora stawiać ośmiu spacji).
Aby wykonać dane zadanie, wydajemy komendę `make nazwa_zadania`. Jeśli nie podamy nazwy zadania (co jest często spotykane), wykonywane jest zadanie o nazwie all (wszystko).

A teraz krótki przykład:
(przeskocz przykład)

```
all:    kompilacja linkowanie
        echo "Wszystko zakonczone pomyslnie"

kompilacja:
        nasm -O999 -f obj -o plik1.obj plik1.asm
        nasm -O999 -f obj -o plik2.obj plik2.asm
        nasm -O999 -f obj -o plik3.obj plik3.asm

        tasm /z /m plik4.asm
        tasm /z /m plik5.asm
        tasm /z /m plik6.asm

linkowanie: plik1.obj plik2.obj plik3.obj plik4.obj plik5.obj plik6.obj
        alink -o wynik.exe plik1.obj plik2.obj plik3.obj plik4.obj \
                 plik5.obj plik6.obj -c- -oEXE -m-

help:
        echo "Wpisz make bez argumentow"
```

Ale MAKE jest mądrzejszy, niż może się to wydawać!
Mianowicie, jeśli stwierdzi, że wynik.exe został stworzony PÓŹNIEJ niż pliki .obj podane w linii zależności, to nie wykona bloku linkowanie, bo nie ma to sensu skoro program wynikowy i tak jest aktualny. MAKE robi tylko to, co trzeba. Oczywiście, niezależnie od wieku plików .obj, dział kompilacja i tak zostanie wykonany (bo nie ma zależności, więc MAKE nie będzie sprawdzał wieku plików).

Znak odwrotnego ukośnika \ powoduje zrozumienie, że następna linia jest kontynuacją bieżącej, znak krzyżyka # powoduje traktowanie reszty linijki jako komentarza.

Jeśli w czasie wykonywanie któregokolwiek z poleceń w bloku wystąpi błąd (ściśle mówiąc, to gdy błąd zwróci wykonywane polecenie, jak u nas TASM czy NASM), to MAKE natychmiast przerywa działanie z informacją o błędzie i nie wykona żadnych dalszych poleceń (pamiętajcie więc o umieszczeniu w zmiennej środowiskowej PATH ścieżki do kompilatorów).

W powyższym pliku widać jeszcze jedno: zmiana nazwy któregoś z plików lub jakieś opcji sprawi, że trzeba ją będzie zmieniać wielokrotnie, w wielu miejscach pliku. Bardzo niewygodne w utrzymaniu, prawda?
Na szczęście z pomocą przychodzą nam ... zmienne, które możemy deklarować w Makefile i które zrozumie program MAKE.
Składnia deklaracji zmiennej jest wyjątkowo prosta i wygląda tak:

```
NAZWA_ZMIENNEJ = wartosc
```

A użycie:

```
$(NAZWA_ZMIENNEJ)
```

Polecam nazwy zmiennych pisać wielkimi literami w celu odróżnienia ich od innych elementów. Pole wartości zmiennej może zawierać dowolny ciąg znaków.
Jeśli chcemy, aby treść polecenia NIE pojawiała się na ekranie, do nazwy tego polecenia dopisujemy z przodu znak małpki @, np.

```
@echo "Wszystko zakonczone pomyslnie"
```

Uzbrojeni w te informacje, przepisujemy nasz wcześniejszy Makefile:
(przeskocz drugi przykład)

```
# Mój pierwszy Makefile

NASM        = nasm     # ale można tu w przyszłości wpisać pełną ścieżkę
NASM_OPCJE  = -O999 -f obj

TASM        = tasm
TASM_OPCJE  = /z /m

ALINK       = alink
ALINK_OPCJE = -c- -oEXE -m-

PLIKI_OBJ   = plik1.obj plik2.obj plik3.obj plik4.obj plik5.obj plik6.obj
PROGRAM     = wynik.exe

all:    kompilacja linkowanie
        @echo "Wszystko zakonczone pomyslnie"

kompilacja:
        $(NASM) $(NASM_OPCJE) -o plik1.obj plik1.asm
        $(NASM) $(NASM_OPCJE) -o plik2.obj plik2.asm
        $(NASM) $(NASM_OPCJE) -o plik3.obj plik3.asm

        $(TASM) $(TASM_OPCJE) plik4.asm
        $(TASM) $(TASM_OPCJE) plik5.asm
        $(TASM) $(TASM_OPCJE) plik6.asm

linkowanie:     $(PLIKI_OBJ)
        $(ALINK) -o $(PROGRAM) $(PLIKI_OBJ) $(ALINK_OPCJE)

help:
```

```
        @echo "Wpisz make bez argumentow"
```

Oczywiście, w końcowym Makefile należy napisać takie regułki, które pozwolą na ewentualną kompilację pojedynczych plików, np.

```
plik1.obj:      plik1.asm plik1.inc
        $(NASM) $(NASM_OPCJE) -o plik1.obj plik1.asm
```

Choć na razie być może niepotrzebna, umiejętność pisania plików Makefile może się przydać już przy projektach zawierających tylko kilka modułów (bo nikt nigdy nie pamięta, które pliki są aktualne, a które nie). O tym, ile Makefile może zaoszczędzić czasu przekonałem się sam, pisząc swoją bibliotekę - kiedyś kompilowałem każdy moduł z osobna, teraz wydaję jedno jedyne polecenie `make` i wszystko się samo robi. Makefile z biblioteki jest spakowany razem z nią i możecie go sobie zobaczyć.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Jak pisać programy w języku asembler?
Część 10 - Nie jesteśmy sami, czyli jak łączyć asemblera z innymi językami.

Jak wiemy, w asemblerze można napisać wszystko. Jednak nie zawsze wszystko trzeba pisać w tym języku. W tej części pokażę, jak asemblera łączyć z innymi językami. Są na to 2 sposoby:

- Wstawki asemblerowe wpisywane bezpośrednio w kod programu
- Osobne moduły asemblerowe dołączane potem do modułów napisanych w innych językach

Postaram się z grubsza omówić te dwa sposoby na przykładzie języków Pascal, C i Fortran 77. Uprzedzam jednak, że moja znajomość języka Pascal i narzędzi związanych z tym językiem jest słaba.

Pascal

(przeskocz Pascal-a)

Wstawki asemblerowe realizuje się używając słowa asm. Oto przykład:

```
{ DOS/Windows }

program pas1;

begin
        asm mov eax,4
end;
end.
```

Można też stosować nieco inny sposób - deklarowanie zmiennej reprezentującej rejestry procesora. Poniższy wycinek kodu prezentuje to właśnie podejście (wywołuje przerwanie 13h z AH=48h, DL=80h, DS:DX wskazującymi na obiekt a):

```
uses crt,dos;

Var
   regs: Registers;

BEGIN
        clrscr();
        With regs DO
        Begin
          Ah:=$48;
          DL:=$80;
          DS:=seg(a);
          DX:=ofs(a);
        End;

        Intr($13,regs);
```

Teraz zajmiemy się bardziej skomplikowaną sprawą - łączenie modułów napisanych w Pascal-u i asemblerze. Pascal dekoruje nazwy zmiennych i procedur, dorabiając znak podkreślenia z przodu. Jakby tego było mało, do nazwy procedury dopisywana jest informacja o jej parametrach. Tak więc z kodu

```
var
```

```
 c:integer;
 d:char;

procedure aaa(a:integer;b:char);
```

otrzymujemy symbole: _C, _D oraz _AAA$INTEGER$CHAR.

Oprócz tego, zwykle w Pascal-u argumenty na stos szły od lewej do prawej, ale z tego co widzę teraz, to Free Pascal Compiler działa odwrotnie - argumenty idą na stos wspak. W naszym przykładzie najpierw na stos pójdzie zmienna typu char, a potem typu integer (obie rozszerzone do rozmiaru DWORDa).
Jedno jest pewne: jeżeli twoja procedura jest uruchamiana z programu napisanego w Pascal-u, to ty sprzątasz po sobie stos - należy przy wyjściu z procedury wykonać `RET liczba`, gdzie liczba jest rozmiarem wszystkich parametrów włożonych na stos (wszystkie parametry są rozmiaru co najmniej DWORD).
Jeśli to ty uruchamiasz procedury napisane w Pascal-u, to nie musisz się martwić o zdejmowanie parametrów ze stosu.

Samo dołączanie modułów odbywa się na linii poleceń, najlepiej w tym celu użyć linkera (po uprzednim skompilowaniu innych modułów na pliki obiektowe).

C i C++

(przeskocz C i C++)

Wstawki asemblerowe zaczynają się wyrażeniem asm { a kończą klamrą zamykającą } (ale NIE w gcc, o tym później). Przykład:

```
asm {
        mov eax, 1
}
```

Wszystkie nowe kompilatory produkują programy 32- lub 64-bitowe, przypominam więc, aby we wstawkach NIE używać przerwań (DOS-a i BIOS-u w Windows).

W C i C++ można, podobnie jak w Pascalu, deklarować zmienne reprezentujące rejestry procesora. Plik nagłówkowy BIOS.H oferuje nam kilka możliwości. Oto przykład:

```
#include <bios.h>
...

REGS rejestry;
...
        rejestry.x.ax = 0x13;
        rejestry.h.bl = 0xFF;
        int86 (0x10, rejestry, rejestry);
```

Łączenie modułów jest prostsze niż w Pascal-u. Język C dekoruje nazwy, dodając znak podkreślenia z przodu. UWAGA - w języku C++ sprawy są trudniejsze nawet niż w Pascal-u. Dlatego, jeśli chcemy, aby nazwa naszej funkcji była niezmieniona (poza tym, że ewentualnie dodamy podkreślenie z przodu) i jednocześnie działała w C++, zawsze przy deklaracji funkcji w pliku nagłówkowym, należy dodać `extern "C"`, np.

```
#ifdef __cplusplus
extern "C" {
```

```
#endif

extern void _naszafunkcja (int parametr, char* a);

#ifdef  __cplusplus
}
#endif
```

W systemach 32-bitowych parametry przekazywane są na stosie OD PRAWEJ DO LEWEJ, czyli pierwszy parametr (u nas powyżej: int) będzie włożony na stos jako ostatni, czyli będzie najpłycej, a ostatni (u nas: char*) będzie najgłębiej.
W systemach 64-bitowych sprawa wygląda trudniej: parametry, w zależności od klasy, są przekazywane (także od prawej do lewej):

- na stosie, jeśli ich rozmiar przekracza 8 bajtów lub zawiera pola niewyrównane co do adresu
- kolejno w rejestrach RDI, RSI, RDX, RCX, R8, R9, jeśli jest klasy całkowitej (mieści się w rejestrze ogólnego przeznaczenia)
- kolejno w rejestrach XMM0 ... XMM7 lub ich górnych częściach, jeśli jest klasy SSE lub SSEUP, odpowiednio
- w obszarze pamięci, jeśli jest klasy zmiennoprzecinkowej lub zespolonej

W C/C++ to funkcja uruchamiająca zdejmuje włożone parametry ze stosu, a NIE funkcja uruchamiana.

Na systemach 32-bitowych parametry całkowitoliczbowe do 32 bitów zwracane są w rejestrze EAX (lub jego częściach: AL, AX, w zależności od rozmiaru), 64-bitowe w EDX:EAX, zmiennoprzecinkowe w ST0. Wskaźniki w 32-bitowych kompilatorach są 32-bitowe i są zwracane w EAX (w 16-bitowych zapewne w AX).
Struktury są wkładane na stos od ostatnich pól, a jeśli funkcja zwraca strukturę przez wartość, np.
`struct xxx f ( struct xxx a )`
to tak naprawdę jest traktowana jak taka funkcja:
`void f ( struct xxx *tu_bedzie_wynik, struct xxx a )`
czyli jako ostatni na stos wkładany jest adres struktury, do której ta funkcja ma włożyć strukturę wynikową.

Na systemach 64-bitowych sprawa ponownie wygląda inaczej. Tu także klasyfikuje się typ zwracanych danych, które są wtedy przekazywane:

- w pamięci, której adres przekazano w RDI (tak, jakby był to pierwszy parametr) - tak na przykład można zwracać struktury. Po powrocie, RAX będzie zawierał przekazany adres
- w kolejnym wolnym rejestrze z grupy RAX, RDX, jeśli klasa jest całkowita
- w kolejnym wolnym rejestrze z grupy XMM0, XMM1, jeśli klasa to SSE
- w górnej części ostatniego używanego rejestru SSE, jeśli klasa to SSEUP
- w ST0, jeśli klasa jest zmiennoprzecinkowa
- razem z poprzednią wartością w ST0, jeśli klasa to X87UP
- część rzeczywista w ST0, a część urojona w ST1, jeśli klasa jest zespolona

Polecam do przeczytania x64 ABI (np. dokument x64-abi.pdf, do znalezienia w Internecie).

Dołączanie modułów (te napisane w asemblerze muszą być uprzednio skompilowane) odbywa się na linii poleceń, z tym że tym razem możemy użyć samego kompilatora, aby wykonał za nas łączenie (nie musimy uruchamiać linkera).

No to krótki 32-bitowy przykładzik (użyję NASMa i Borland C++ Builder):

```
; NASM casm1.asm

section .text use32

global  _suma

_suma:
; po wykonaniu push ebp i mov ebp, esp:
; w [ebp]    znajduje się stary EBP
; w [ebp+4]  znajduje się adres powrotny z procedury
; w [ebp+8]  znajduje się pierwszy parametr,
; w [ebp+12] znajduje się drugi parametr
; itd.

%idefine        a       [ebp+8]
%idefine        b       [ebp+12]

        push    ebp
        mov     ebp, esp

        mov     eax, a
        add     eax, b

; LEAVE = mov esp, ebp / pop ebp
        leave
        ret
```

oraz plik casm.c:

```
#include <stdio.h>

extern int _suma (int a, int b); /* deklaracja funkcji zewnętrznej */

int suma (int a, int b);         /* prototyp funkcji */

int c=1, d=2;

int main()
{
        printf("%d\n", suma(c,d));
        return 0;
}
```

Kompilacja odbywa się tak:

```
nasm -o casm1.obj -f obj casm1.asm
bcc32 casm.c casm1.obj
```

Uwaga: w kompilatorach GNU: DJGPP, Dev-C++, MinGW, CygWin format wyjściowy NASMa powinien być ustawiony na COFF. Możliwe, że format COFF trzeba będzie wybrać także w innych.

W wyniku otrzymujemy programik, który na ekranie elegancko wyświetla wynik równy 3.

Może się zdarzyć też, że chcemy tylko korzystać z funkcji języka C, ale główną część programu chcemy napisać w asemblerze. Nic trudnego: używane funkcje deklarujemy jako zewnętrzne (pamiętając o znaku podkreślenia), ale uwaga - swoją funkcję główną musimy nazwać _main. Jest tak dlatego, że teraz punkt startu programu nie jest w naszym kodzie, lecz w samej bibliotece języka C. Program zaczyna się między innymi ustawieniem tablic argumentów listy poleceń i zmiennych środowiska. Dopiero po tych operacjach

biblioteka C uruchamia funkcję _main instrukcją CALL.
Inną ważną sprawą jest to, że naszą funkcję główną powinniśmy zakończyć instrukcją RET (zamiast normalnych instrukcji wyjścia z programu), która pozwoli przekazać kontrolę z powrotem do biblioteki C, umożliwiając posprzątanie (np. wyrzucenie buforów z wyświetlonymi informacjami w końcu na ekran).
Krótki (także 32-bitowy) przykładzik:

```
section .text

global _main

extern _printf

_main:

        ; printf("Liczba jeden to: %d\n", 1);
        push    dword 1         ; drugi argument
        push    dword napis     ; pierwszy argument
        call    _printf         ; uruchomienie funkcji
        add     esp, 2*4        ; posprzątanie stosu

        ; return 0;
        xor     eax, eax
        ret                     ; wyjście z programu

section .data

napis: db "Liczba jeden to: %d", 10, 0
```

Kompilacja powinna odbyć się tak:

```
nasm -o casm2.obj -f obj casm2.asm
bcc32 casm2.obj
```

Jedna uwaga: funkcje biblioteki C mogą zamazać nam zawartość wszystkich rejestrów (poza EBX, EBP, ESI, EDI w systemach 32-bitowych, i RBX, RBP, R12, R13, R14, R15 na systemach 64-bitowych), więc nie wolno nam polegać na zawartości rejestrów po uruchomieniu jakiejkolwiek funkcji C.

Kompilator GNU gcc wymaga osobnego wytłumaczenia. Składnia wstawek asemblerowych różni się od powyższej dość znacznie, a jej opisy możecie znaleźć w podręczniku GCC (sekcje: 5.34 i 5.35), na stronach DJGPP oraz (w języku polskim) na stronie pana Danileckiego.
Jak zauważycie, różni się nawet sam wygląd instrukcji, gdyż domyślnie gcc używa składni AT&T języka asembler. U siebie mam krótkie porównanie tych składni.

Fortran 77

W tym języku nie wiem nic o wstawkach asemblerowych, więc przejdziemy od razu do łączenia modułów.
Fortran dekoruje nazwy, stawiając znak podkreślenia PO nazwie funkcji lub zmiennej (wyjątkiem jest funkcja główna - blok PROGRAM - która nazywa się `MAIN__`, z dwoma podkreśleniami).
Nie musimy pisać externów, ale jest kilka reguł przekazywania parametrów:

- parametry przekazywane są od prawej do lewej, czyli tak jak w C.

- jeśli to jest tylko możliwe, wszystkie parametry przekazywane są przez referencję, czyli przez wskaźnik. Gdy to jest niemożliwe, przekazywane są przez wartość.
- jeśli na liście parametrów pojawia się łańcuch znakowy, to na stosie przed innymi parametrami umieszczana jest jego długość.
- wyniki są zwracane w tych samych miejscach, co w języku C.

Na przykład, następujący kod:

```
        REAL FUNCTION aaa (a, b, c, i)

                CHARACTER a*(*)
                CHARACTER b*(*)
                REAL c
                INTEGER i

                aaa = c
        END

[...]
                CHARACTER x*8
                CHARACTER y*5
                REAL z,t
                INTEGER u

                t=aaa (x, y, z, u)
[...]
```

będzie przetłumaczony na asemblera tak (samo uruchomienie funkcji):

```
        push    5
        push    8
        push    u_      ; adres, czyli offset zmiennej "u"
        push    z_
        push    y_
        push    x_

        call    aaa_
```

(to niekoniecznie musi wyglądać tak ładnie, gdyż zmienne x, y, u i z są lokalne w funkcji MAIN__, czyli są na stosie, więc ich adresy mogą wyglądać jak [ebp-28h] lub podobnie).

Funkcja uruchamiająca sprząta stos po uruchomieniu (podobnie jak w C).
Dołączać moduły można bezpośrednio z linii poleceń (w każdym razie pod Linuksem z kompilatorem F77/G77).

Podam teraz przykład łączenia Fortrana 77 i asemblera. W oryginale użyłem narzędzi Linuksowych: NASMa i F77, ale po minimalnych przeróbkach powinno to też działać pod Windows. Oto pliki:

```
        ; NASM - asm1fl.asm
        section .text use32
        global  suma_

        suma_:

        ; po wykonaniu push ebp i mov ebp, esp:
        ; w [ebp]    znajduje się stary EBP
        ; w [ebp+4]  znajduje się adres powrotny z procedury
```

```
; w [ebp+8]  znajduje się pierwszy parametr,
; w [ebp+12] znajduje się drugi parametr
; itd.

%idefine        a       [ebp+8]
%idefine        b       [ebp+12]

        push    ebp
        mov     ebp, esp

; przypominam, że nasze parametry są w rzeczywistości
; wskaźnikami do prawdziwych parametrów

        mov     edx, a          ; EDX = wskaźnik do 1-szego parametru
        mov     eax, [edx]      ; EAX = 1-szy parametr
        mov     edx, b
        add     eax, [edx]

        leave
        ret
```

I teraz plik asmfl.f:

```
PROGRAM funkcja_zewnetrzna

INTEGER a,b,suma

a=1
b=2

WRITE (*,*) suma(a,b)

END
```

Po skompilowaniu (ewentualnie zmieniając opcję -f u NASMa):

```
nasm -f obj -o asm1fl.obj asm1fl.asm
f77 -o asmfl.exe asmfl.f asm1fl.obj
```

i uruchomieniu na ekranie powinna ponownie pojawić się cyfra 3.

Informacji podanych w tym dokumencie NIE należy traktować jako uniwersalnych, jedynie słusznych reguł działających w każdej sytuacji. Aby uzyskać kompletne informacje, należy zapoznać się z dokumentacją posiadanego kompilatora.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia:

1. Napisz plik asemblera, zawierający funkcję obliczania reszty z dzielenia dwóch liczb całkowitych. Następnie, połącz ten plik z programem napisanym w dowolnym innym języku (najlepiej w C/C++, gdyż jest najpopularniejszy) w taki sposób, by Twoją funkcję można było uruchamiać z tamtego programu. Jeśli planujesz łączyć asemblera z C, upewnij się że Twoja funkcja działa również z programami napisanymi w C++.

Jak pisać programy w języku asembler?
Część 11 - Pamięć jest nietrwała, czyli jak posługiwać się plikami.

Jak wiemy, wszystkich danych nie zmieścimy w pamięci. A nawet jeśli zmieścimy, to pozostaną tam tylko do najbliższego wyłączenia prądu. Dlatego trzeba je zapisywać do pliku, a potem umieć je z tego pliku odczytać. W tej części zajmiemy się właśnie operacjami na plikach.

Do operowania na plikach posłużymy się kilkoma funkcjami przerwania 21h:

- AH = 3Ch - utworzenie pliku (wymazanie, gdy już istnieje).
  W rejestrze CX podajemy atrybuty pliku (ustawiony bit 0 oznacza plik tylko do odczytu, bit 1 - czy plik ma być ukryty, bit 2 - plik systemowy, 3 - etykieta woluminu, 4 - zawsze zero, 5 - plik archiwalny), a DS:DX wskazuje na nazwę pliku.
  UWAGA: Nazwa musi być zakończona bajtem zerowym
  Niewypełnienie powyższego warunku jest przyczyną wielu błędów w programach.
  W rejestrze AX otrzymujemy uchwyt do pliku (file handle) - specjalną wartość przydzielaną nam przy otwieraniu pliku.

- AH = 3Dh - otworzenie istniejącego pliku.
  W rejestrze AL podajemy tryb dostępu, DS:DX wskazuje na nazwę pliku.
  Tryby dostępu określa się następującymi bitami w AL (pomijam nieistotne informacje, całość możecie znaleźć w RBIL):

```
        Bit     Opis
        2-0     tryb dostępu
                000 tylko do odczytu
                001 tylko do zapisu
                010 odczyt i zapis
        3       zarezerwowany, musi być 0
        6-4     tryb współdzielenia (DOS 3.0+)
                000 tryb zgodności
    DENYALL - zabroń innym odczytu i zapisu
    DENYWRITE - zabroń innym zapisu
    DENYREAD - zabroń innym odczytu
    DENYNONE - nikomu nie zabraniaj
        7       dziedziczenie
                Jeśli ten bit jest ustawiony, plik jest prywatny dla bieżącego procesu
                i nie będzie dziedziczony przez procesy potomne
```

  UWAGA: Nazwa musi być zakończona bajtem zerowym
  W rejestrze AX otrzymujemy uchwyt do pliku (file handle) - specjalną wartość przydzielaną nam przy otwieraniu pliku.

- AH = 3Eh - zamknięcie otwartego pliku.
  W rejestrze BX podajemy uchwyt do pliku.

- AH = 3Fh - odczyt z pliku.
  W rejestrze BX podajemy uchwyt do pliku, w CX - ilość bajtów do odczytania, DS:DX wskazuje na miejsce, dokąd będziemy zapisywać.

- AH = 40h - zapis do pliku.
  W rejestrze BX podajemy uchwyt do pliku, w CX - ilość bajtów do zapisania, DS:DX wskazuje na miejsce, z którego będziemy czytać dane do zapisania.

- AH = 42h - przechodzenie na określoną pozycję w pliku.
  Rejestr AL mówi DOSowi, skąd wyruszamy: 0 - z początku pliku, 1 - z bieżącej pozycji, 2 - z końca pliku. BX = uchwyt pliku, CX : DX - odległość, o którą się przesuwamy (może być ujemna).

- AH = 41h - usuwanie pliku.
  DS:DX wskazuje na nazwę pliku.
  UWAGA: Nazwa musi być zakończona bajtem zerowym

Wszystkie te funkcje ustawiają flagę carry (CF=1), gdy wystąpił jakiś błąd.
Po szczegóły (w tym kody błędów) odsyłam do Listy Przerwań Ralfa Brown'a.

Przykładowe użycie tych funkcji:
(przeskocz przykłady)
Utworzenie pliku i zapisanie czegoś do niego:

```
mov     ah, 3ch         ; numer funkcji - utworzenie
mov     dx, nazwa       ; adres nazwy pliku
xor     cx, cx          ; atrybuty. Zero oznacza normalny plik.
int     21h             ; utworzenie pliku
jc      blad            ; sprawdzamy, czy nie ma błędu.

mov     [uchwyt], ax
mov     bx, ax          ; zapisujemy uchwyt

mov     ah, 40h         ; numer funkcji - zapis
                        ; BX = uchwyt do pliku
mov     cx, 1024        ; ilość bajtów do zapisania
mov     dx, bufor       ; adres bufora, z którego bierzemy bajty
int     21h             ; zapis do pliku
jc      blad            ; sprawdzamy, czy nie ma błędu.

mov     ah, 3eh         ; numer funkcji - zamknięcie pliku
                        ; BX = uchwyt do pliku
int     21h             ; zamykamy plik
jc      blad            ; sprawdzamy, czy nie ma błędu.
```

Otwarcie istniejącego pliku, odczytanie i zapisanie czegoś do niego:

```
mov     ah, 3dh         ; numer funkcji - otwieranie pliku
mov     al, 00010010b   ; wyłączny dostęp do odczytu i zapisu
mov     dx, nazwa       ; adres nazwy pliku
int     21h             ; utworzenie pliku
jc      blad            ; sprawdzamy, czy nie ma błędu.

mov     [uchwyt], ax
mov     bx, ax          ; zapisujemy uchwyt

mov     ah, 3fh         ; numer funkcji - odczyt
                        ; BX = uchwyt do pliku
mov     cx, 1024        ; ilość bajtów do odczytania
mov     dx, bufor       ; adres bufora, do którego czytamy
int     21h             ; czytamy z pliku
jc      blad            ; sprawdzamy, czy nie ma błędu.
```

```
        ; .... operacje na bajtach z pliku, na przykład
        xor     byte [bufor], 0ffh

        mov     ah, 40h         ; numer funkcji - zapis
                                ; BX = uchwyt do pliku
        mov     cx, 1024        ; ilość bajtów do zapisania
        mov     dx, bufor       ; adres bufora, z którego bierzemy bajty
        int     21h             ; zapis do pliku
        jc      blad            ; sprawdzamy, czy nie ma błędu.

        ; Zauważcie, że zapisane bajty wylądowały po odczytanych, gdyż nie
        ; zmieniliśmy pozycji w pliku, a ostatnia operacja (odczyt) zostawiła
        ; ją tuż po odczytanych bajtach

        mov     ah, 3eh         ; numer funkcji - zamknięcie pliku
                                ; BX = uchwyt do pliku
        int     21h             ; zamykamy plik
        jc      blad            ; sprawdzamy, czy nie ma błędu.
```

A teraz prawdziwy przykład. Będzie to nieco uszczuplona (pominąłem wczytywanie nazwy pliku) wersja mojego programu na_male.asm. Program ten zamienia wszystkie wielkie litery w podanym pliku na ich małe odpowiedniki. Reszta znaków pozostaje bez zmian. Jedna rzecz jest warta uwagi - nigdzie nie zmieniam rejestru BX, więc ciągle w nim jest uchwyt do pliku i nie muszę tego uchwytu zapisywać do pamięci.
A teraz kod:
(przeskocz na_male.asm)

```
; Program zamienia wszystkie litery w podanym pliku z wielkich na male.
;
; Autor: Bogdan D.
; kontakt: bogdandr (at) op (dot) pl
;
; nasm -O999 -o na_male.com -f bin na_male.asm
; fasm na_male.asm na_male.com

org 100h

start:
        mov     dx, info
        mov     ah, 9
        int     21h

        mov     ax, 3d02h       ; otwórz do odczytu i zapisu,
                                ; zabroń wszystkim dostępu
        mov     dx, plik        ; adres nazwy pliku
        int     21h

        jnc     otw_ok
        call    plik_blad       ; uruchamiamy tą procedurę, gdy wystąpił błąd
        jmp     zamk_ok         ; jeśli nie udało się nam nawet otworzyć
                                ; pliku, to od razu wychodzimy z programu.


otw_ok:
        mov     bx, ax          ; zapisujemy uchwyt do pliku
        mov     bp, 400h        ; BP = rozmiar bufora.

czytaj:
        mov     ah, 3fh         ; funkcja czytania
```

```
                                ; BX = uchwyt
        mov     dx, bufor       ; adres bufora, dokąd czytamy
        mov     cx, bp          ; kilobajt - rozmiar bufora
        int     21h             ; odczyt

        jnc     czyt_ok
        call    plik_blad       ; uruchamiamy tą procedurę, gdy wystąpił błąd

czyt_ok:
        xor     di, di          ; DI będzie wskaźnikiem do bufora.
                                ; Na początku go zerujemy.
        cmp     ax, cx          ; Czy ilość bajtów odczytana (AX) =
                                ; = ilość żądana (CX) ?
        jne     przy_eof        ; jeśli nie, to plik się skończył

zamiana:
        mov     dl, [bufor+di]  ; wczytujemy znak z bufora do DL

        cmp     dl, "A"
        jb      znak_ok
        cmp     dl, "Z"
        ja      znak_ok

        or      dl, 20h         ; jeśli okazał się wielką literą, zamieniamy
                                ; go na małą
        mov     [bufor+di], dl  ; i zapisujemy w miejsce,
                                ; gdzie poprzednio był

znak_ok:
        inc     di              ; przechodzimy do innych znaków
        loop    zamiana         ; aż przejdziemy przez cały bufor
                                ; (CX=BP=400h)

        mov     dx, ax          ; DX = ilość przeczytanych bajtów
        mov     ax, 4201h       ; idź do ... od pozycji bieżącej.
                                ; aby zapisać zmienione litery,
                                ; musimy przejść
                                ; się w pliku o 1 kilobajt wstecz.
                                ; Do CX:DX wpisujemy odległość
        neg     dx              ; DX = -DX
;       dec     cx              ; CX po wyjściu z pętli jest zerem,
                                ; więc wykonanie DEC zrobi z niego -1.
        mov     cx, 0ffffh      ; CX = -1
                                ; CX:DX = -DX = -ilość przeczytanych bajtów

                                ; BX = uchwyt
        int     21h             ; wykonujemy przeskok w pliku

        jnc     idz_ok
        call    plik_blad

idz_ok:                         ; po udanym przeskoku

        mov     dx, bufor       ; DX = adres bufora, skąd będziemy brać dane
                                ; do zapisania
                                ; BX = uchwyt
        mov     ah, 40h         ; funkcja zapisz
        mov     cx, bp          ; CX = BP = 400h = długość bufora.
        int     21h             ; zapisujemy

        jmp     short czytaj    ; i idziemy czytać nową partię danych.
```

```
przy_eof:                       ; gdy jesteśmy już przy końcu pliku.

;       xor     di, di          ; DI już = 0 (wcześniej to zrobiliśmy)

        mov     bp, ax          ; BP = ilość przeczytanych znaków
        mov     cx, ax          ; CX = ilość przeczytanych znaków

zamiana2:
        mov     dl, [bufor+di]  ; pobieramy znak z bufora do DL

        cmp     dl, "A"
        jb      znak_ok2
        cmp     dl, "Z"
        ja      znak_ok2

        or      dl, 20h         ; jeśli okazał się wielką literą,
                                ; zamieniamy go na małą
        mov     [bufor+di], dl  ; i zapisujemy w miejsce,
                                ; gdzie poprzednio był

znak_ok2:
        inc     di              ; przechodzimy do innych znaków
        loop    zamiana2        ; aż przejdziemy przez cały bufor
                                ; (CX = BP = ilość bajtów)

        mov     dx, bp
;       dec     cx              ; CX po wyjściu z pętli jest zerem, więc
                                ; wykonanie DEC zrobi z niego -1.
        mov     cx, 0ffffh      ; CX = -1
                                ; CX:DX = -DX
        mov     ax, 4201h       ; idź do ... od pozycji bieżącej.
        neg     dx              ; DX = -DX.
                                ; CX:DX = -DX = -ilość przeczytanych bajtów

                                ; BX = uchwyt
        int     21h             ; wykonujemy przeskok w pliku

        jnc     idz_ok2
        call    plik_blad

idz_ok2:                        ; po udanym przeskoku

        mov     dx, bufor       ; zapiszemy do pliku resztę danych.
                                ; DX = adres bufora.
                                ; BX = uchwyt
        mov     cx, bp          ; CX = ilość bajtów uprzednio odczytancyh
        mov     ah, 40h         ; funckja zapisu do pliku
        int     21h             ; zapisujemy

        jnc     zamk            ; gdy nie ma błędu, to zamkniemy plik
        call    plik_blad

zamk:
        mov     ah, 3eh
                                ; BX = uchwyt
        int     21h             ; zamykamy nasz plik
        jnc     zamk_ok
        call    plik_blad

zamk_ok:
        mov     ax, 4c00h
        int     21h             ; wyjście...
```

```
plik_blad:                      ; procedura wyświetla informację o tym, że
                                ; wystąpił błąd i wypisuje numer tego błędu
        push    ax
        push    bx

        mov     dx, blad_plik
        mov     bx, ax
        mov     ah, 9
        int     21h

        mov     ax, bx
        call    pl

        pop     bx
        pop     ax

        ret

pl:                     ; procedura wypisuje liczbę (4 znaki szesnastkowe)

        mov     bx, ax
        shr     ax, 12
        call    pc2

        mov     al, bh
        and     al, 0fh
        call    pc2

        mov     al, bl
        shr     al, 4
        and     al, 0fh
        call    pc2

        mov     al, bl
        and     al, 0fh
        call    pc2

        ret

pc2:
;we: AL - cyfra hex
;wy: pisze cyfrę, niszczone ax

        cmp     al, 9
        mov     ah, 0eh
        ja      hex
        or      al, "0"
        jmp     short pz
hex:
        add     al, "A"-10
pz:
        int     10h

        ret

bufor:          times 400h db 0         ; bufor wielkosci 1 kilobajta
;plik:          times 80 db 0
plik            db "aaa.txt",0          ; nazwa pliku

info db "Program zamienia wielkie litery w pliku na male.",10,13,"$"
```

```
input1          db "Podaj nazwe pliku do przetworzenia: $"
zla_nazwa       db 10,13,"Zla nazwa pliku.$"
blad_plik       db 10,13,"Blad operacji na pliku. Kod: $"
```

Ten program chyba nie był za trudny, prawda? Cała treść skupia się na odczytaniu paczki bajtów, ewentualnej ich podmianie i zapisaniu ich w to samo miejsce, gdzie były wcześniej.

Pliki są podstawowym sposobem przechowywania danych. Myślę więc, że się ze mną zgodzicie, iż opanowanie ich obsługi jest ważne i nie jest to aż tak trudne, jakby się mogło wydawać.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia:

1. Napisz program, który wykona po kolei następujące czynności:
    1. Utworzy nowy plik
    2. Zapisze do niego 256 bajtów o wartościach od 00 do FF (nie musicie zapisywać po 1 bajcie)
    3. Zamknie ten plik
    4. Otworzy ponownie ten sam plik
    5. Zapisze odczytane bajty w nowej tablicy 256 słów w taki sposób:

        ```
        00 00 00 01 00 02 00 03 00 04 .... 00 FD 00 FE 00 FF
        ```

        czyli każdy oddzielony bajtem zerowym (należy przeczytać wszystkie bajty, po czym ręcznie je przenieść gdzie indziej i wzbogacić)
    6. Zamknie otwarty plik
    7. Usunie ten plik

Jak pisać programy w języku asembler?
Część 12 - Czego od nas pragną, czyli linia poleceń programu. Zmienne środowiska.

Teraz zajmiemy się dość istotną sprawą z punktu widzenia programisty i użytkownika oprogramowania: linią poleceń. Nie wszyscy lubią podawać dane programowi w czasie jego pracy i odpowiadać na pytania o dane. Często (o ile jest to możliwe) można tego oszczędzić i zamiast bezustannie zadawać użytkownikowi pytania, przeczytać, co wpisano nam w linię poleceń. Umożliwia to pisanie programów, które raz uruchomione z prawidłową linią poleceń nie pytają już się o nic a tylko wykonują swoją pracę bez przeszkadzania użytkownikom.

Przejdźmy więc do szczegółów. Wszystkie operacje, które wykonamy, będą się opierać na założeniu, że w swoim programie nie zrobiliście absolutnie nic z rejestrem DS. Jeśli go zmieniliście, to użyjcie tej funkcji (opis oczywiście z Listy Przerwań Ralfa Brown'a):
(przeskocz int 21h, ah=62h)

```
INT 21 - DOS 3.0+ - GET CURRENT PSP ADDRESS
        AH = 62h
Return: BX = segment of PSP for current process
```

i otrzymaną w BX wartość wpiszcie do DS.

Mając oryginalny DS (wtedy pokazuje on na Program Segment Prefix - PSP), można w nim znaleźć wiele ciekawych informacji:

- bajt pod [ds:80h] mówi nam, ile znaków znajduje się na linii poleceń, bez kończącego znaku nowej linii (Enter = 13 ASCII).
- od [ds:81h] do [ds:0FFh] jest linia poleceń. Jak widać, ma ona długość 128 znaków i tylko tyle możemy wpisać, uruchamiając nasz program. Teraz również widać, dlaczego programy typu COM zaczynają się od adresu 100h - po prostu wcześniej nie mogą, bo CS=DS.
- pod [ds:2ch] znajduje się numer segmentu, w którym umieszczono kopię zmiennych środowiskowych (tych ustawianych komendą SET, np. w autoexec.bat) do wykorzystania przez nasz program. Zmienne środowiskowe zapisane są od początku segmentu i oddzielone od siebie bajtami zerowymi. Dwa bajty zerowe pod rząd oznaczają koniec zmiennych.

Wszystko ładnie wygląda w teorii, ale jak tego używać? Aby odpowiedzieć na to pytanie, napisałem ten oto krótki programik. Jedynym celem jego życia jest wyświetlenie długości jego linii poleceń, samej linii poleceń, numerów segmentów: kodu, danych i środowiska (dla porównania), oraz samych zmiennych środowiskowych (jeśli wyświetla się za dużo lub za mało, można zmienić liczbę na końcu programu - pokażę, którą).

Oto kod (NASM):
(przeskocz kod programu)

```
; Program wyświetla własną linię poleceń i zmienne środowiskowe.
;
; Autor: Bogdan D.
; kontakt: bogdandr (at) op (dot) pl
;
```

```
; kompilacja NASM:
;   nasm -O999 -o liniap.obj -f obj liniap.asm
;   alink liniap.obj  bibl\lib\bibldos.lib -c- -oEXE -m-
; kompilacja FASM:
;   fasm liniap.asm liniap.obj
;   alink liniap.obj  bibl\lib\bibldos.lib -c- -entry _start -oEXE -m-

; dołączamy moją biblioteczkę
%include "bibl\incl\dosbios\nasm\std_bibl.inc"
%include "bibl\incl\dosbios\nasm\do_nasma.inc"

.stack 400h                     ; program typu EXE musi mieć stos

; FASM:
; format coff
; include "bibl\incl\dosbios\fasm\std_bibl.inc"
; use16
; public _start
; i nic poza tym

; FASM:
; _start:
..start:                        ; miejsce startu programu
        mov     si, 80h ; [ds:80h] - długość linii poleceń bez Entera
        xor     eax, eax
        mov     al, [si]        ; AL = długość linii poleceń
        pisz
        db      "Dlugosc linii polecen: ", 0
        pisz8                   ; wypisujemy AL
        nwln                    ; przechodzimy do nowej linii

        mov     cx, ax          ; CX=długość linii poleceń,
                                ; abyśmy wiedzieli,
                                ; ile znaków należy wyświetlić
        inc     si              ; SI=81h. [ds:81h] to początek
                                ; linii poleceń
        pisz
        db      "Linia polecen=", 0
        pisz_dl                 ; wypisujemy CX znaków spod DS:SI,
                                ; czyli całą linię poleceń
        nwln

        mov     ax, cs
        pisz
        db      "Segment kodu programu CS=", 0
        pisz16                  ; wyświetlamy AX=CS
        nwln
        mov     ax, ds
        pisz
        db      "Segment danych DS=", 0
        pisz16                  ; wyświetlamy AX=DS
        nwln

        mov     ax, [ds:2ch]
        pisz
        db      "Segment zmiennych srodowiskowych: DS:[2ch]=",0
        pisz16                  ; wyświetlamy AX=segment środowiska
        nwln

        mov     ds, ax          ; DS = segment środowiska
        xor     si, si          ; SI = początek segmentu
        pisz
```

```
        db      "Zmienne srodowiskowe: ", 0

        mov     ah, 0eh         ; funckja wypisywania znaku
        dec     si              ; tylko po to, aby najbliższe INC SI
                                ; zadziałało prawidłowo i ustawiło nas z
                                ; powrotem na 0
wypisz_srod:
        nwln                    ; przejdź do nowej linii
wypisz:
        inc     si              ; SI teraz pokazuje na kolejny znak
        cmp     si, 400 ; żeby nie było za długo -
                        ; to tę liczbę MOŻNA ZMIENIĆ
        ja      koniec

        mov     al, [si]        ; pobierz znak spod [DS:SI]
        test    al, al          ; czy bajt zerowy?
        jz      sprawdz         ; jeśli tak, to sprawdzimy,
                                ; czy nie dwa pod rząd

        int     10h             ; wypisz znak
        jmp     short   wypisz  ; i w kółko od nowa

sprawdz:
        cmp     byte [si+1], 0
        jne     wypisz_srod

koniec:
        wyjscie
```

Jak widać, nie było to aż takie trudne jak się mogło zdawać na początku. Właśnie poznaliście kolejną rzecz, która jest łatwa w użyciu, a możliwości której są duże. Teraz będziecie mogli śmiało zacząć pisać programy, których jedynym kanałem komunikacyjnym z użytkownikiem będzie linia poleceń, co znacznie uprości ich obsługę.

Tylko pamiętajcie o dodaniu kodu wyświetlającego sposób użycia programu, gdy nie podano mu żadnych parametrów.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia:

1. Napisz program, który utworzy plik podany jako parametr. Jeśli podano drugi parametr (oddzielony od pierwszego spacją), zapisz jego wartość do tego pliku. Jeśli nie podano żadnych parametrów, niech program wypisze stosowną wiadomość.

2. Napisz program, który oblicza NWD (patrz część 8) dwóch liczb podanych na linii poleceń. Jeśli nie podano wystarczającej liczby parametrów, niech program wyświetli stosowną wiadomość.

Jak pisać programy w języku asembler?
Część 13 - Operacje na bitach, czyli to, w czym asembler błyszczy najbardziej.

W tej części poznamy ważną grupę instrukcji - operacje na bitach. Te właśnie instrukcje odróżniają asemblera od innych języków, gdzie rzadko pojawia się możliwość działania na tych najmniejszych jednostkach informacji (odpowiednie operatory istnieją w językach C i Pascal, ale inne języki, jak np. Fortran 77, są tego pozbawione).
Mimo iż o wszystkich instrukcjach opisanych w tej części już wspomniałem przy okazji omawiania podstawowych rozkazów procesora, to instrukcje bitowe są ważne i zasługują na oddzielny rozdział, poświęcony w całości tylko dla nich.

Zdawać by się mogło, że z takim jednym, maleńkim bitem niewiele da się zrobić: można go wyczyścić (wyzerować), ustawić (wstawić do niego 1) lub odwrócić jego bieżącą wartość. Ale te operacje mają duże zastosowania i dlatego ich poznanie jest niezbędne. Jeśli sobie przypomnicie, to używaliśmy już wielokrotnie takich instrukcji jak AND czy XOR. Teraz przyszedł czas, aby poznać je bliżej.

Instrukcja NOT

(przeskocz NOT)
Instrukcja NOT (logiczna negacja - to NIE jest to samo, co zmiana znaku liczby!) jest najprostszą z czterech podstawowych operacji logicznych i dlatego to od niej rozpocznę wstęp do instrukcji bitowych.
NOT jest instrukcją jednoargumentową, a jej działanie wygląda tak:

```
NOT 0 = 1
NOT 1 = 0
```

Używamy tej instrukcji wtedy, gdy chcemy naraz odwrócić wszystkie bity w zmiennej lub rejestrze. Na przykład, jeśli AX zawiera 0101 0011 0000 1111 (530Fh), to po wykonaniu NOT AX w rejestrze tym znajdzie się wartość 1010 1100 1111 0000 (ACF0h). Dodanie obu wartości powinno dać FFFFh.
NOT może mieć zastosowanie tam, gdzie wartość logiczna fałsz ma przyporządkowaną wartość zero, a prawda - wartość FFFFh, gdyż NOT w tym przypadku dokładnie przekłada prawdę na fałsz.

Instrukcja AND

(przeskocz AND)
Instrukcji AND (logicznej koniunkcji) najprościej używać do wyzerowania bitów. Tabelka działania AND wygląda tak:

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

No ale jakie to może mieć zastosowanie?
Powiedzmy teraz, że chcemy sprawdzić, czy bit numer 4 (numerację będę podawał od zera) rejestru AX jest równy 1, czy 0. Tutaj nie wystarczy proste porównanie CMP, gdyż reszta rejestru może zawierać nie wiadomo co. Z pomocą przychodzi nam właśnie instrukcja AND. Poniżej pseudo-przykład:

```
and     ax, 0000 0000 0001 0000b        ; (and ax, 16)
```

Teraz, jeśli bit numer 4 (odpowiadający wartości 2^4=16) był równy 1, to cały AX przyjmie wartość 16, jeśli zaś był równy zero, to cały AX będzie zerem. Na nasze szczęście, instrukcja AND ustawia odpowiednio flagi procesora, więc rozwiązaniem naszego problemiku będzie kod:

```
and     ax, 16
jz      bit_4_byl_zerem
;jnz    bit_4_nie_byl_zerem
```

A jakieś zastosowanie praktyczne?
Już podaję: zamiana małych liter na wielkie. W kodzie ASCII litery małe od wielkich różnią się tylko tym, że mają ustawiony bit numer 5. Tak więc po wykonaniu:

```
mov     al, "a"
and     al, 5fh         ; 5fh = 0101 1111 - czyścimy bit 5
                        ; (i 7 przy okazji)
```

w rejestrze AL będzie kod wielkiej litery A.
Inne zastosowanie znajdziecie w moim kursie programowania głośniczka:

```
in      al, 61h
and     al, not 3               ; zerujemy bity 0 i 1
                                ; NASM: and al,~3
out     61h, al
```

W tym kodzie instrukcja AND posłużyła nam do wyczyszczenia bitów 0 i 1 (NOT 3 = NOT 0000 0011 = 1111 1100).

Jak zauważyliście, instrukcja AND niszczy zawartość rejestru, oprócz interesujących nas bitów. Jeśli zależy Wam na zachowaniu rejestru, użyjcie instrukcji TEST. Działa ona identycznie jak AND, ale nie zapisuje wyniku działania. Po co nam więc taka instrukcja? Otóż, wynik nie jest zapisywany, ale TEST ustawia dla nas flagi identycznie jak AND. Pierwszy kod przepisany z instrukcją TEST będzie wyglądał tak:

```
test    ax, 16
jz      bit_4_byl_zerem
;jnz    bit_4_nie_byl_zerem
```

Teraz nasz program będzie ciągle działać prawidłowo, ale tym razem zawartość rejestru AX została zachowana.
Jest jeszcze jedno ciekawe zastosowanie instrukcji TEST:

```
test    ax, ax
```

I co to ma niby robić? Wykonuje `AND AX, AX` , nigdzie nie zapisuje wyniku i tylko ustawia flagi.
No właśnie! Ustawia flagi, w tym flagę zera ZF. To, co widzicie powyżej to najwydajniejszy sposób na to, aby sprawdzić czy wartość rejestru nie jest zerem.

Instrukcja OR

(przeskocz OR)
Instrukcja OR (logiczna alternatywa) w prosty sposób służy do ustawiania bitów (wpisywania do nich 1). Tabelka działania wygląda następująco:

```
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
```

Jeśli na przykład chcemy, aby 2 najmłodsze bity rejestru BX były się równe 1, a nie chcemy naruszać innych bitów (czyli MOV jest wykluczone), możemy to zrobić tak:

```
or      bx, 0000 0000 0000 0011         ; (or bx, 3)
```

Zastosowanie tego jest proste. Podam 2 przykłady. Pierwszy z nich jest wyjęty z mojej procedury wytwarzającej dźwięk w głośniczku (i kursu poświęconego temu zagadnieniu):

```
in      al, 61h
or      al, 3                           ; ustawiamy bity 0 i 1
out     61h, al
```

Przykład drugi jest odwróceniem operacji AND na znakach ASCII:

```
mov     al, "A"
or      al, 20h                 ; 20h = 0010 0000 - ustawiamy bit 5
```

teraz w AL powinien być kod małej literki a.
Instrukcja OR nie ma swojego odpowiednika, jakim jest TEST dla AND. Ale za to ma inne ciekawe zastosowanie - można nią sprawdzić, czy 2 rejestry naraz nie są zerami (to jest najlepszy sposób - bez żadnych CMP, JNZ/JZ itp.):

```
or      ax, bx
```

Podobnie, jak w instrukcji AND, flaga zera będzie ustawiona, gdy wynik operacji jest zerem - a to może się zdarzyć tylko wtedy, gdy AX i BX są jednocześnie zerami.
Zauważcie, że nie można do tego celu użyć instrukcji AND. Dlaczego? Podam przykład: niech AX=1 i BX = 8. AX i BX nie są oczywiście równe zero, ale:

```
        0000 0000 0000 0001     (=AX)
AND     0000 0000 0000 1000     (=BX)
        =
        0000 0000 0000 0000
```

Dlatego zawsze należy przemyśleć efekt działania instrukcji.

Instrukcja XOR

(przeskocz XOR)
Instrukcji XOR (eXclusive OR, logiczna alternatywa wykluczająca) używa się do zmiany stanu określonego bitu z 0 na 1 i odwrotnie.
Działanie XOR jest określone tak:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

Zauważmy także, że dla dowolnych a i b mamy:
(a XOR b) XOR b = a
a XOR 0 = a
a XOR -1 = NOT a (-1 = FF w bajcie, FFFF w słowie i FFFFFFFF w dwordzie)
a XOR a = 0
Z tej ostatniej równości wynika natychmiast, że wyXORorwanie rejestru z samym sobą zawsze go wyzeruje. W ten sposób otrzymujemy jeden z dwóch najwydajniejszych sposobów na wyzerowanie rejestru:

```
xor     rej, rej
```

Drugi sposób to SUB rej,rej.

Teraz przykład: chcemy, aby wartość rejestru AX stała się równa 1 gdy rejestr był wyzerowany, a zerem, gdy była w tym rejestrze jedynka. Oto, jak możemy to zrobić:

```
        cmp     ax, 1
        je      wyzeruj
        mov     ax, 1
        jmp     koniec
wyzeruj:
        mov     ax, 0
koniec:
```

Ale wersja optymalna wygląda tak:

```
xor     ax, 1
```

gdyż mamy:

```
wartość AX:     0000 0000 0000 0001             0000 0000 0000 0000
        XOR     0000 0000 0000 0001             0000 0000 0000 0001
        =
nowy AX:        0000 0000 0000 0000             0000 0000 0000 0001
```

Jak widać, jest to o wiele prostsze i wydajniejsze rozwiązanie. Dlatego właśnie dobrze jest, gdy pozna się instrukcje logiczne.

Instrukcje przesuwania bitów

(przeskocz instrukcje przesuwania)
Instrukcje przesuwania bitów (shift) przemieszczają bity, nie zmieniając ich wzajemnego położenia (przesuwają grupowo). To wyjaśnienie może się wydawać bardzo pokrętne, ale spokojnie - zaraz wszystko się wyjaśni.
Na początek powiem, że jest kilka takich instrukcji (które też były podane w rozdziale o podstawowych instrukcjach procesora):

- SHL - shift left (shift logical left) = przesunięcie (logicznie) w lewo
- SAL - shift arithmetic left = przesunięcie (arytmetycznie) w lewo
- SHR - shift logical right = przesunięcie (logiczne) w prawo
- SAR - shift arithmetic right = przesunięcie (arytmetyczne)
- SHLD/SHRD = przesunięcia logiczne w lewo/prawo o podwójnej precyzji

Działanie każdej z tych instrukcji pokażę na przykładzie.
Niech na początku AX = 1010 0101 1010 0101 (A5A5h).

SHL i równoważna SAL działa tak (zakładając, że przesuwamy o jeden): najstarszy bit jest we fladze CF, każdy inny bit wchodzi na miejsce bitu starszego o 1, a do bitu zerowego wkładane jest zero.
Po wykonaniu SHL AX,3 wartość AX będzie więc wynosić 0010 1101 0010 1000 (2D28h), gdyż wszystkie bity przesunęliśmy o 3 miejsca w lewo, oraz CF=1 (bo jako ostatnia z rejestru wyleciała jedynka).

Instrukcja SHR działa w drugą stronę niż SHL: bit zerowy jest umieszczany we fladze CF, każdy inny bit wchodzi na miejsce bitu młodszego o 1, a do najstarszego bitu wkładane jest zero.
Dlatego teraz po wykonaniu SHR AX,1 w rejestrze AX będzie 0001 0110 1001 0100 (1694h), bo poprzednie bity AX przesunęliśmy o 1 miejsce w prawo, oraz CF=0.
SAR różni się od SHR nie tylko nazwą, ale też działaniem. Słowo arytmetyczne w nazwie NIE jest tu bez znaczenia. Gdy SAR działa na liczbach ze znakiem, to zachowuje ich znak (bit7), tzn wykonuje to samo, co SHR, ale zamiast wkładać zero do najstarszego bitu, wstawia tam jego bieżącą wartość.
Z poprzedniego przykładu mamy, że AL = 94h = 1001 0100. Gdy teraz wykonamy SAR AL,2 to jako wynik otrzymamy 1110 0101 (E5h), bo wszystkie bity poszły o 2 miejsca w prawo o bit 7 został zachowany, i CF=0.

SHLD i SHRD wykonują to samo, co SHL i SHR ale na 2 rejestrach naraz (no niezupełnie). Na przykład wykonanie `SHLD EAX,EBX,3` spowoduje że 3 najstarsze bity EAX zostaną wyrzucone (i CF=ostatni z wyrzuconych) oraz 3 najstarsze bity EBX przejdą na nowo powstałe miejsca w 3 najmłodszych bitach EAX. Ale uwaga: EBX pozostaje niezmieniony ! I to jest właśnie przyczyna użycia słów no niezupełnie.

Ale nie sposób powiedzieć o SHL i SHR bez podania najbardziej popularnego zastosowania: szybkie mnożenie i dzielenie.
Jak można mnożyć i dzielić tylko przesuwając bity, pytacie?
Otóż, sprawa jest bardzo prosta. Wpiszcie do AX jedynkę i wykonajcie kilka razy SHL AX,1 za każdym

razem sprawdzając zawartość AX. Jak zauważycie, w AX będą kolejno 1,2,4,8,16,... Czyli za każdym razem zawartość AX się podwaja.
Ogólnie, `SHL rej,n` mnoży zawartość rejestru przez 2^n. Na przykład `SHL AX,4` przemnoży AX przez 2^4 = 16.
Ale co zrobić, gdy chcemy mnożyć przez coś innego niż 2^n?
Odpowiedź jest równie prosta, np. AX * 10 = (AX*8) + (AX*2) - z tym się chyba zgodzicie. A od tego już tylko 1 krok do

```
mov     bx, ax
shl     ax, 3           ; AX = AX*8
shl     bx, 1           ; BX = BX*2 = AX*2
add     ax, bx          ; AX = AX*10
```

Ale niekoniecznie musimy dodawać wyniki. Zauważcie, że AX * 15 = (AX*8) + (AX*4) + (AX*2) + AX. Trzeba byłoby wykonać 3 SHL i 3 ADD. Ale my skorzystamy z innego rozwiązania: AX * 15 = (AX*16) - AX. Już tylko 1 SHL i 1 SUB. Stąd mamy:

```
mov     bx, ax
shl     ax, 4           ; AX = AX*16
sub     ax, bx
```

Dokładnie w ten sam sposób działa dzielenie (tylko oczywiście przy dzieleniu używamy SHR/SAR i niestety szybko możemy dzielić tylko przez potęgi dwójki). Pilnujcie tylko, aby używać tej właściwej instrukcji! Jak wiemy, 65534 = 0FFFEh = -2 . Teraz, oczywiście FFFE SHR 1 = 7FFFh = 32767 (=65534/2) a FFFE SAR 1 = FFFF = -1 (= -2/2). Widać różnicę, prawda? Pamiętajcie, że SAR patrzy na znak i go zachowuje.

Używanie SHL dla mnożenia i (zwłaszcza) SHR dla dzielenia może znacznie przyśpieszyć nasze programy, gdyż instrukcje MUL i DIV są dość wolne.

Instrukcje rotacji bitów

(przeskocz instrukcje rotacji)
Teraz przedstawię kolejną grupę instrukcji bitowych - instrukcje rotacji bitów. W tej grupie są tylko 4 instrukcje:

- ROL - rotate left = obrót w lewo.
  Ta instrukcja robi tyle, co SHL, lecz zamiast do bitu zerowego wkładać zero, wkłada tam bieżącą wartość najstarszego bitu (przy okazji zachowując go także we fladze CF).
  bit7 = bit6, ... , bit1 = bit0, bit0 = stary bit7
- RCL - rotate through carry left = obrót w lewo z użyciem flagi CF. Ta instrukcja jest podobna do ROL z jedną różnicą: wartość wstawiana do najmłodszego bitu jest brana z flagi CF, a nie od razu z najstarszego bitu. Po wzięciu bieżącej wartości CF, najstarszy bit jest do niej zapisywany.
  carry flag CF = bit7, bit7 = bit6, ... , bit1 = bit0, bit0 = stara CF
- ROR - rotate right = obrót w prawo. Ta instrukcja robi tyle, co SHR, lecz zamiast do najstarszego bitu wkładać zero, wkłada tam bieżącą wartość najmłodszego bitu (przy okazji zachowując go także we fladze CF).
  bit0 = bit1, ... , bit6 = bit7, bit7 = stary bit0

- RCR - rotate through carry right = obrót w prawo z użyciem flagi CF. Ta instrukcja jest podobna do ROR z jedną różnicą: wartość wstawiana do najstarszego bitu jest brana z flagi CF, a nie od razu z najmłodszego bitu. Po wzięciu bieżącej wartości CF, najmłodszy bit jest do niej zapisywany.
  CF = bit0, bit0 = bit1, ... , bit6 = bit7, bit7 = stara CF

Schematyczne działanie tych instrukcji na bajtach widać na tych rysunkach:
(przeskocz rysunki)

```
ROL:
                +-->-------------->--------------><--+
                |                                    |
        CF <-   7 <- 6 <- 5 <- 4 <- 3 <- 2 <- 1 <- 0

RCL:
                +-->------------> CF >------------><--+
                |                                     |
                7 <- 6 <- 5 <- 4 <- 3 <- 2 <- 1 <- 0

ROR:
                +--<--------------<--------------<--+
                |                                    |
                7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0   -> CF

RCR:
                +--<------------< CF <------------<--+
                |                                     |
                7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0
```

W przypadku ROL i ROR, to ostatni wyjęty z jednej strony a włożony z drugiej strony bit zostaje też zapisany do flagi CF.
RCR i RCL działają tak, że bit, który ma zostać wstawiony, jest pobierany z CF, a wypchnięty bit ląduje w CF, a nie od razu na nowym miejscu.

No to kilka przykładów:

```
0011 1100  ROL  2  =  1111 0000  (tak samo jak SHL)
0011 1100  ROL  3  =  1110 0001

1111 0000  ROR  1  =  0111 1000  (tak samo jak SHR)
1010 0011  ROR  5  =  0001 1101
```

Zastosowanie tych instrukcji znalazłem jedno: generowanie chaosu w rejestrach...
Po co to mi? Na przykład generatory liczb pseudo-losowych z mojej biblioteki korzystają z tych właśnie instrukcji (a także z kilku poprzednich, np. XOR).

Instrukcje testowania i szukania bitów

(przeskocz instrukcje BT*)
Ostatnia już grupa rozkazów procesora to instrukcje testowania i szukania bitów. W tej grupie znajdują się:

- BT - Bit Test
- BTC - Bit Test and Complement
- BTR - Bit Test and Reset
- BTS - Bit Test and Set
- BSF - Bit Scan Forward
- BSR - Bit Scan Reverse

Teraz po kolei omówię działanie każdej z nich.
Instrukcje BT* przyjmują 2 argumenty: miejsce, gdzie mają znaleźć dany bit i numer tego bitu, a zwracają wartość tego bitu we fladze CF. Ponadto, BTS ustawia znaleziony bit na 1, BTR czyści znaleziony bit a BTC odwraca znaleziony bit.
Kilka przykładów:

```
bt      eax, 21         ; umieść 21. bit EAX w CF
jc      bit_jest_1
...
bts     cl, 2           ; umieść 2. bit CL w CF i ustaw go
jnc     bit_2_byl_zerem
...
btc     dh, 5           ; umieść 5. bit DH w CF i odwróć go
jc      bit_5_byl_jeden
```

Instrukcje Bit Scan przyjmują 2 argumenty: pierwszy z nich to rejestr, w którym będzie umieszczona pozycja (numer od zera począwszy) pierwszego bitu, którego wartość jest równa 1 znalezionego w drugim argumencie instrukcji. Dodatkowo, BSF szuka tego pierwszego bitu zaczynając od bitu numer 0, a BSR od najstarszego (numer 7, 15 lub 31 w zależnosci od rozmiaru drugiego argumentu).

Teraz szybki przykładzik:

```
mov     ax, 1010000b
bsf     bx, ax
bsr     cx, ax
```

Po wykonaniu powyższych instrukcji w BX powinno być 4, a w CX - 6 (bity liczymy od zera).

Jak pewnie zauważyliście, w kilku miejscach w tym tekście wyraźnie podkreśliłem słowa najwydajniejszy i im podobne. Chciałem w ten sposób uzmysłowić Wam, że operacje logiczne / binarne są bardzo ważną grupą instrukcji. Używanie ich, najlepiej wraz z instrukcją LEA służącą do szybkich rachunków, może kilkakrotnie (lub nawet kilkunastokrotnie) przyśpieszyć najważniejsze części Waszych programów (np. intensywne obliczeniowo pętle o milionach powtórzeń - patrz np. program L_mag.asm z 8. części tego kursu).

Dlatego zachęcam Was do dobrego opanowania instrukcji binarnych - po prostu umożliwia to pisanie programów o takiej wydajności, o której inni mogą tylko pomarzyć...

Po szczegółowy opis wszystkich instrukcji odsyłam, jak zwykle do : Intela i AMD

Ciekawe operacje na bitach (w języku C)

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)

Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia:

1. W jednej komendzie policz:
    1. iloraz z dzielenia EDI przez 4
    2. resztę z dzielenia EDI przez 4
    3. największą liczbę mniejszą lub równą EDI dzielącą sie przez 4

   Wskazówka: 4 = 2^2 oraz możliwe reszty z dzielenia przez 4 to 0, 1, 2 i 3 i zajmują one co najwyżej 2 bity.

2. W jednej komendzie:
    1. ustaw bity 0, 11, 4 i 7 rejestru CX, nie ruszając pozostałych
    2. wyczyść bity 9, 2, 7 i 25 rejestru ESI, nie ruszając pozostałych
    3. przełącz (zmień wartość na odwrotną) bity 16, 4, 21, 1 i 10 rejestru EAX, nie ruszając pozostałych
    4. spraw, by wartość rejestru AL=18h zmieniła się na 80h, bez instrukcji MOV
    5. spraw, by wartość rejestru AL=18h zmieniła się na 81h, bez instrukcji MOV
    6. przełącz bit 23 rejestru EDX nie ruszając pozostałych, a jego starą wartość umieść we fladze CF

Jak pisać programy w języku asembler?
Część 14 - Operacje o wielokrotnej precyzji, czyli co zrobić, gdy liczby nie mieszczą się w rejestrach.

Czasami w naszych programach zachodzi potrzeba, aby posługiwać się np. liczbami przekraczającymi 4 czy nawet 8 bajtów, a my mamy tylko rejestry 32-bitowe (lub czasem 16-bitowe).
Co wtedy zrobić?
Odpowiedzi na to właśnie pytanie postaram się udzielić w tej części kursu.

Do naszych celów posłuży coś, co się nazywa arytmetyką wielokrotnej precyzji (ang. Multiprecision Arithmetic). Generalną zasadą będzie zajmowanie się obliczeniami po kawałku (bo z resztą inaczej się nie da) i zapamiętywanie, czy z poprzedniego kawałka wynieśliśmy coś w pamięci (do tego celu w prosty sposób wykorzystamy flagę CF, która wskazuje właśnie, czy nie nastąpiło przepełnienie).
Najpierw kilka ustaleń:

1. Będę tutaj używał rejestrów 32-bitowych, ale w razie potrzeby dokładnie te same algorytmy działają także dla rejestrów innych rozmiarów.
2. Zmienne arg1 i arg2 mają po 16 bajtów (128 bitów) każda. Na potrzeby nauki wystarczy w sam raz.
3. Zmienna wynik ma tyle samo bajtów, co arg1 i arg2, z wyjątkiem mnożenia, gdzie oczywiście musi być dwa razy większa.
4. Zmienna wynik na początku zawiera zero.
5. Kod nie zawsze będzie optymalny, ale chodzi mi o to, aby był jak najbardziej jasny i przejrzysty.

A więc do dzieła.

Dodawanie

(przeskocz dodawanie)
Dodawanie, podobnie jak uczyli nas w szkole, zaczynamy od najmłodszych cyfr (cyfr jedności) - tyle że zamiast pojedynczych cyferek będziemy dodawać całe 32-bitowe kawałki naraz. Flaga CF powie nam, czy z poprzedniego dodawania wynosimy coś w pamięci (nawet z dodawania dużych liczb wyniesiemy co najwyżej 1 bit w pamięci). To coś trzeba oczywiście dodać potem do wyższej części wyniku.
No to dodajemy:
(przeskocz program do dodawania)

```
mov     eax, [arg1]
add     eax, [arg2]     ; dodajemy 2 pierwsze części liczb
mov     [wynik], eax    ; i ich sumę zapisujemy w pierwszej
                        ; części wyniku. Flaga CF mówi, czy
                        ; wynosimy coś w pamięci

mov     eax, [arg1+4]
adc     eax, [arg2+4]   ; dodajemy drugie części + to,
                        ; co wyszło z poprzedniego dodawania
                        ; [arg1] i [arg2] (a to jest w fladze
                        ; CF, stąd instrukcja ADC zamiast ADD)

mov     [wynik+4], eax  ; całość:[arg1+4]+[arg2+4]+"w pamięci"
                        ; z pierwszego dodawania zapisujemy tu
                        ; Flaga CF zawiera (lub nie) bit
                        ; "w pamięci", ale tym razem z ADC

                        ; podobnie reszta działania:
```

```
        mov     eax, [arg1+8]
        adc     eax, [arg2+8]
        mov     [wynik+8], eax

        mov     eax, [arg1+12]
        adc     eax, [arg2+12]
        mov     [wynik+12], eax

        jc      blad_przepelnienie
```

Odejmowanie

(przeskocz odejmowanie)
W szkole uczyli nas, że zaczynamy od najmłodszych cyfr i ewentualnie pożyczamy od starszych. Tutaj będziemy robić dokładnie tak samo! Wymaga to jednak poznania nowej instrukcji - SBB (Substract with Borrow). Działa ona tak samo, jak zwykła instrukcja odejmowania SUB, ale dodatkowo odejmuje wartość flagi CF, czyli 1 lub 0, w zależności od tego, czy w poprzednim kroku musieliśmy pożyczać czy też nie. Ewentualną pożyczkę trzeba oczywiście odjąć od wyższej części wyniku.
Piszmy więc (od arg1 będziemy odejmować arg2):
(przeskocz program do odejmowania)

```
        mov     eax, [arg1]
        sub     eax, [arg2]             ; odejmujemy 2 pierwsze części
        mov     [wynik], eax            ; i zapisujemy wynik
                                        ; flaga CF mówi, czy była pożyczka

        mov     eax, [arg1+4]
        sbb     eax, [arg2+4]           ; odejmujemy razem z pożyczką (CF),
                                        ; jeśli w poprzednim odejmowaniu
                                        ; musieliśmy coś pożyczać

        mov     [wynik+4], eax          ; wynik: [arg1+4]-[arg2+4]-pożyczka
                                        ; z pierwszego odejmowania
                                        ; CF teraz zawiera pożyczkę z SBB

                                        ; podobnie reszta działania:
        mov     eax, [arg1+8]
        sbb     eax, [arg2+8]
        mov     [wynik+8], eax

        mov     eax, [arg1+12]
        sbb     eax, [arg2+12]
        mov     [wynik+12], eax

        jc      arg1_mniejszy_od_arg2
```

Zmiana znaku liczby

(przeskocz NEG)
Teraz zajmiemy się negacją (zmianą znaku liczby). Ta operacja jest o tyle dziwna, że wykonujemy ją od góry (od najstarszych bajtów) i po negacji niższych trzeba zadbać o pożyczkę we wszystkich wyższych częściach.

Popatrzcie (będziemy negować arg1):
(przeskocz program do negacji)

```
neg     dword [arg1+12]         ; negujemy najstarszą część

neg     dword [arg1+8]          ; negujemy drugą od góry
sbb     dword [arg1+12], 0      ; jeśli była pożyczka od starszej
                                ; (a prawie zawsze będzie), to tę
                                ; pożyczkę odejmujemy od starszej

neg     dword [arg1+4]          ; negujemy kolejną część i odejmujemy
                                ; pożyczki od starszych części
sbb     dword [arg1+8], 0
sbb     dword [arg1+12], 0

neg     dword [arg1]            ; negujemy kolejną część i odejmujemy
                                ; pożyczki od starszych części
sbb     dword [arg1+4], 0
sbb     dword [arg1+8], 0
sbb     dword [arg1+12], 0
```

Dla większych liczb nie wygląda to za ciekawie. Dlatego najprostszym sposobem będzie po prostu odjęcie danej liczby od zera, do czego zastosujemy poprzedni algorytm odejmowania.

Mnożenie

(przeskocz mnożenie)
Mnożenie jest nieco bardziej skomplikowane, ale ciągle robione tak jak w szkole, czyli od prawej. Ustalmy dla wygody, że arg1 zawiera ABCD, a arg2 - PQRS (każda z liter oznacza 32 bajty). Ogólny schemat wygląda teraz tak:
(przeskocz schemat mnożenia)

```
                          A  B  C  D
                        * P  Q  R  S
                        =
                                 D*S
                              C*S
                           B*S
                        A*S
                                 D*R
                              C*R
                           B*R
                        A*R
                                 D*Q
                              C*Q
                           B*Q
                        A*Q
                                 D*P
                              C*P
                           B*P
                     + A*P
               =
                     F  G  H  I  J  K  L


[wynik]    = L = D*S
[wynik+4]  = K = C*S + D*R
[wynik+8]  = J = B*S + C*R + D*Q
```

```
        [wynik+12] = I = A*S + B*R + C*Q + D*P
        [wynik+16] = H = A*R + B*Q + C*P
        [wynik+20] = G = A*Q + B*P
        [wynik+24] = F = A*P
        (rzecz jasna, każdy iloczyn zajmuje 2 razy po 4 bajty, np. L zajmuje
         [wynik] i częściowo [wynik+4], ale tutaj podałem tylko miejsca,
         gdzie pójdą najmłodsze części każdego w iloczynów)
```

Obliczenia wyglądałyby tak (pamiętamy, że wynik operacji MUL jest w EDX:EAX):
(przeskocz program mnożenia)

```
        ; przez rozpoczęciem procedury zmienna "wynik" musi być wyzerowana!
;[wynik] = L = D*S

        mov  eax, dword [arg1]          ; EAX = D
        mul  dword [arg2]               ; EDX:EAX = D*S
        mov  dword [wynik], eax
        mov  dword [wynik+4], edx


;[wynik+4]  = K = C*S + D*R

        mov  eax, dword [arg1+4]        ; EAX = C
        mul  dword [arg2]               ; EDX:EAX = C*S
        add  dword [wynik+4], eax
        adc  dword [wynik+8], edx

        adc  dword [wynik+12], 0

        mov  eax, dword [arg1]          ; EAX = D
        mul  dword [arg2+4]             ; EDX:EAX = D*R
        add  dword [wynik+4], eax
        adc  dword [wynik+8], edx

        adc  dword [wynik+12], 0

;[wynik+8]  = J = B*S + C*R + D*Q

        mov  eax, dword [arg1+8]        ; EAX = B
        mul  dword [arg2]               ; EDX:EAX = B*S
        add  dword [wynik+8], eax
        adc  dword [wynik+12], edx

        adc  dword [wynik+16], 0

        mov  eax, dword [arg1+4]        ; EAX = C
        mul  dword [arg2+4]             ; EDX:EAX = C*R
        add  dword [wynik+8], eax
        adc  dword [wynik+12], edx

        adc  dword [wynik+16], 0

        mov  eax, dword [arg1]          ; EAX = D
        mul  dword [arg2+8]             ; EDX:EAX = D*Q
        add  dword [wynik+8], eax
        adc  dword [wynik+12], edx

        adc  dword [wynik+16], 0

;[wynik+12] = I = A*S + B*R + C*Q + D*P
```

```
        mov  eax, dword [arg1+12]       ; EAX = A
        mul  dword [arg2]               ; EDX:EAX = A*S
        add  dword [wynik+12], eax
        adc  dword [wynik+16], edx

        adc  dword [wynik+20], 0

        mov  eax, dword [arg1+8]        ; EAX = B
        mul  dword [arg2+4]             ; EDX:EAX = B*R
        add  dword [wynik+12], eax
        adc  dword [wynik+16], edx

        adc  dword [wynik+20], 0

        mov  eax, dword [arg1+4]        ; EAX = C
        mul  dword [arg2+8]             ; EDX:EAX = C*Q
        add  dword [wynik+12], eax
        adc  dword [wynik+16], edx

        adc  dword [wynik+20], 0

        mov  eax, dword [arg1]          ; EAX = D
        mul  dword [arg2+12]            ; EDX:EAX = D*P
        add  dword [wynik+12], eax
        adc  dword [wynik+16], edx

        adc  dword [wynik+20], 0

;[wynik+16] = H = A*R + B*Q + C*P

        mov  eax, dword [arg1+12]       ; EAX = A
        mul  dword [arg2+4]             ; EDX:EAX = A*R
        add  dword [wynik+16], eax
        adc  dword [wynik+20], edx

        adc  dword [wynik+24], 0

        mov  eax, dword [arg1+8]        ; EAX = B
        mul  dword [arg2+8]             ; EDX:EAX = B*Q
        add  dword [wynik+16], eax
        adc  dword [wynik+20], edx

        adc  dword [wynik+24], 0

        mov  eax, dword [arg1+4]        ; EAX = C
        mul  dword [arg2+12]            ; EDX:EAX = C*P
        add  dword [wynik+16], eax
        adc  dword [wynik+20], edx

        adc  dword [wynik+24], 0

;[wynik+20] = G = A*Q + B*P

        mov  eax, dword [arg1+12]       ; EAX = A
        mul  dword [arg2+8]             ; EDX:EAX = A*Q
        add  dword [wynik+20], eax
        adc  dword [wynik+24], edx

        adc  dword [wynik+28], 0

        mov  eax, dword [arg1+8]        ; EAX = B
        mul  dword [arg2+12]            ; EDX:EAX = B*P
```

```
        add  dword [wynik+20], eax
        adc  dword [wynik+24], edx

        adc  dword [wynik+28], 0

;[wynik+24] = F = A*P

        mov  eax, dword [arg1+12]       ; EAX = A
        mul  dword [arg2+12]            ; EDX:EAX = A*P
        add  dword [wynik+24], eax
        adc  dword [wynik+28], edx

        adc  dword [wynik+32], 0
```

Dzielenie

(przeskocz dzielenie)
Dzielenie dwóch liczb dowolnej długości może być kłopotliwe i dlatego zajmiemy się przypadkiem dzielenia dużych liczb przez liczbę, która mieści się w 32 bitach. Dzielić będziemy od najstarszych bajtów do najmłodszych. Jedna sprawa zasługuje na uwagę: między dzieleniami będziemy zachowywać resztę w EDX (nie będziemy go zerować), gdyż w taki sposób otrzymamy prawidłowe wyniki. Oto algorytm (dzielimy arg1 przez 32-bitowe arg2):
(przeskocz program dzielenia)

```
        mov     ebx, [arg2]     ; zachowujemy dzielnik w wygodnym miejscu

        xor     edx, edx        ; przed pierwszym dzieleniem zerujemy EDX
        mov     eax, [arg1+12]  ; najstarsze 32 bity
        div     ebx
        mov     [wynik+12], eax ; najstarsza część wyniku już jest policzona

                                ; EDX bez zmian! Zawiera teraz resztkę
                                ; z [wynik+12], która jest mniejsza od
                                ; EBX. Ta resztka będzie teraz starszą
                                ; częścią liczby, którą dzielimy.
        mov     eax, [arg1+8]
        div     ebx
        mov     [wynik+8], eax

                                ; EDX bez zmian!
        mov     eax, [arg1+4]
        div     ebx
        mov     [wynik+4], eax

                                ; EDX bez zmian!
        mov     eax, [arg1]
        div     ebx
        mov     [wynik], eax
                                ; EDX = reszta z dzielenia
```

Jeśli wasz dzielnik może mieć więcej niż 32 bity, to trzeba użyć algorytmu podobnego do tego, którego uczyliśmy się w szkole. Ale po takie szczegóły odsyłam do AoA (patrz ostatni akapit w tym tekście).

Operacje logiczne i bitowe

(przeskocz operacje bitowe)
Przerobiliśmy już operacje arytmetyczne, przyszła więc kolej na operacje logiczne.
Na szczęście operacje bitowe AND, OR, XOR i NOT nie zależą od wyników poprzednich działań, więc po prostu wykonujemy je na odpowiadających sobie częściach zmiennych i niczym innym się nie przejmujemy.
Oto przykład (obliczenie arg1 AND arg2):
(przeskocz AND)

```
mov     eax, [arg1]
and     eax, [arg2]
mov     [wynik], eax

mov     eax, [arg1+4]
and     eax, [arg2+4]
mov     [wynik+4], eax

mov     eax, [arg1+8]
and     eax, [arg2+8]
mov     [wynik+8], eax

mov     eax, [arg1+12]
and     eax, [arg2+12]
mov     [wynik+12], eax
```

Pozostałe trzy (OR, XOR i NOT) będą przebiegać dokładnie w ten sam sposób.

Sprawa z przesunięciami (SHL/SHR) i rotacjami jest nieco bardziej skomplikowana, gdyż bity wychodzące z jednej części zmiennej muszą jakoś znaleźć się w wyższej części. Ale spokojnie, nie jest to aż takie trudne, gdy przypomnimy sobie, że ostatni wyrzucony bit ląduje we fladze CF.
A co zrobić, gdy chcemy przesuwać o więcej niż jeden bit (wszystkie wyrzucone bity nie znajdą się przecież naraz w CF)?
Niestety, trzeba to robić po jednym bicie na raz. Ale ani SHL ani SHR nie pobiera niczego z flagi CF. Dlatego użyjemy operacji rotacji bitów przez flagę CF.
Pora na przykład (SHL arg1, 2):
(przeskocz SHL)

```
shl     dword [arg1], 1         ; wypychamy najstarszy bit do CF
rcl     dword [arg1+4], 1       ; wypchnięty bit wyląduje tutaj w
                                ; bicie numer 0, a najstarszy zostaje
                                ; wypchnięty do CF

rcl     dword [arg1+8], 1       ; najstarszy bit z [arg1+4] staje się
                                ; tutaj najmłodszym, a najstarszy z
                                ; tej części ląduje w CF
rcl     dword [arg1+12], 1      ; najstarszy bit z [arg1+8] staje się
                                ; tutaj najmłodszym, a najstarszy z
                                ; tej części ląduje w CF

                                ; mamy już SHL o 1 pozycję. Teraz
                                ; drugi raz (dokładnie tak samo):
shl     dword [arg1], 1
rcl     dword [arg1+4], 1
rcl     dword [arg1+8], 1
rcl     dword [arg1+12], 1
```

Podobnie będzie przebiegać operacja SHR (rzecz jasna, SHR wykonujemy OD GÓRY):
(przeskocz SHR)

```
; SHR arg1, 1

shr     dword [arg1+12], 1      ; wypychamy najmłodszy bit do CF
rcr     dword [arg1+8], 1       ;wypchnięty bit wyląduje tutaj w bicie
                                ; najstarszym, a najmłodszy zostaje
                                ; wypchnięty do CF
rcr     dword [arg1+4], 1       ; najmłodszy bit z [arg1+8] staje się
                                ; tutaj najstarszym, a najmłodszy z
                                ; tej części ląduje w CF
rcr     dword [arg1], 1         ; najmłodszy bit z [arg1+4] staje się
                                ; tutaj najstarszym, a najmłodszy z
                                ; tej części ląduje w CF
```

Gorzej jest z obrotami (ROL, ROR, RCL, RCR), gdyż ostatni wypchnięty bit musi się jakoś znaleźć na początku. Oto, jak możemy to zrobić (pokażę ROL arg1, 1):
(przeskocz ROL)

```
; najpierw normalny SHL:

shl     dword [arg1], 1
rcl     dword [arg+4], 1
rcl     dword [arg+8], 1
rcl     dword [arg1+12], 1

; teraz ostatni bit jest w CF. Przeniesiemy go do
; najmłodszego bitu EBX.

mov     ebx, 0                  ; tu nie używać XOR! (zmienia flagi)
rcl     ebx, 1                  ; teraz EBX = CF
          ADC ebx, 0)           ; (można też użyć

; i pozostaje nam już tylko dopisać najmłodszy bit w wyniku:

or      [arg1], ebx             ; lub ADD - bez różnicy
```

ROL o więcej niż 1 będzie przebiegać dokładnie tak samo (ten sam kod trzeba powtórzyć wielokrotnie). Sprawa z RCL różni się niewiele od tego, co pokazałem wyżej. Ściśle mówiąc, SHL zamieni się na RCL i nie musimy zajmować się bitem, który wychodzi do CF (bo zgodnie z tym, jak działa RCL ten bit musi tam pozostać). Cała operacja będzie więc wyglądać po prostu tak:
(przeskocz RCL)

```
rcl     dword [arg1], 1
rcl     dword [arg+4], 1
rcl     dword [arg+8], 1
rcl     dword [arg1+12], 1
```

Operacje ROR i RCR przebiegają podobnie:
(przeskocz ROR)

```
; ROR arg1, 1

; najpierw normalny SHR (pamiętajcie, że od góry):

shr     dword [arg1+12], 1
rcr     dword [arg1+8], 1
rcr     dword [arg1+4], 1
rcr     dword [arg1], 1         ; najmłodszy bit został wypchnięty
```

```
; teraz ostatni bit jest w CF. Przeniesiemy go do
; najstarszego bitu EBX.

mov     ebx, 0                  ; tu nie używać XOR! (zmienia flagi)
rcr     ebx, 1                  ; teraz EBX = 00000000 lub 80000000h

; i pozostaje nam już tylko dopisać najstarszy bit w wyniku:

or      [arg1+12], ebx
```

I już tylko prosty RCR:
(przeskocz RCR)

```
rcr     dword [arg1+12], 1
rcr     dword [arg1+8], 1
rcr     dword [arg1+4], 1
rcr     dword [arg1], 1
```

Porównywanie liczb

(przeskocz porównywanie)

Porównywanie należy oczywiście zacząć od najstarszej części i schodzić do coraz to niższych części. Pierwsza różniąca się para porównywanych elementów powie nam, jaka jest relacja między całymi liczbami. Porównywać można dowolną ilość bitów na raz, w tym przykładzie użyję podwójnych słów (32 bity) i będę sprawdzał na równość:
(przeskocz program do porównywania)

```
mov     eax, [arg1+12]
cmp     eax, [arg2+12]  ; porównujemy najstarsze części
jne     nie_rowne
mov     eax, [arg1+8]
cmp     eax, [arg2+8]
jne     nie_rowne
mov     eax, [arg1+4]
cmp     eax, [arg2+4]
jne     nie_rowne
mov     eax, [arg1]
cmp     eax, [arg2]     ; porównujemy najmłodsze części
jne     nie_rowne
jmp     rowne
```

To by było na tyle z rozszerzonej arytmetyki. Mam nadzieję, że algorytmy te wytłumaczyłem wystarczająco dobrze, abyście mogli je zrozumieć. Jeśli nawet coś nie jest od razu jasne, to należy przejrzeć rozdział o instrukcjach procesora i wrócić tutaj - to powinno rozjaśnić wiele ewentualnych wątpliwości.

Niektóre algorytmy zawarte tutaj wziąłem ze wspaniałej książki Art of asembler (Art of asembly Language Programming, AoA) autorstwa Randalla Hyde'a. Książkę tę zawsze i wszędzie polecam jako świetny materiał do nauki nie tylko samego asemblera, ale także architektury komputerów i logiki. Książka ta dostępna jest ZA DARMO.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia:

1. Napisz program, który zrobi, co następuje:
    1. Przemnoży EAX przez EBX (wartości dowolne, ale nie za małe) i zachowa wynik (najlepiej w rejestrach).
    2. Przemnoży ECX przez EBP.
    3. Jeśli dowolny wynik wyszedł zero (sprawdzić każdy co najwyżej 1 instrukcją), to niech przesunie te drugi w prawo o 4 miejsca. Jeśli nie, to niech doda je do siebie.

Jak pisać programy w języku asembler?
Część 15 - If/else if/else, do, while, for, switch/case - czyli o tym, jak używać bloków kontrolnych.

Wszystkie języki wysokiego poziomu mają pewne bloki kontrolne i pętle, co może w oczach niektórych osób stanowić przewagę nad asemblerem. Dlatego teraz pokażę, jak przepisać te wszystkie struktury z wykorzystaniem asemblera, często uzyskując kod lepszy niż ten wytworzony przez kompilatory języków wyższego poziomu.
Zanim zaczniemy, dodam, że nie każdy język wysokiego poziomu posiada opcje kompilacji warunkowej (coś w stylu #ifdef w języku C), ale za to KAŻDY dobry kompilator języka asembler ma taką opcję wbudowaną! Po szczegóły odsyłam do instrukcji posiadanego kompilatora.

Bloki decyzyjne (warunkowe) if/else if/else.

(przeskocz bloki warunkowe)

Przetłumaczenie czegoś takiego na asembler nie jest trudne i opiera się na instrukcjach CMP oraz odpowiednich skokach warunkowych. Pokażę to na przykładzie (będę używał składni języka C, gdyż posiada wszystkie struktury, które chciałbym omówić):
(przeskocz schemat bloku if/else)

```
if (a == b)             /* czy a jest równe b? */
{
        /* część 1 */
}
else if (a == c)
{
        /* część 2 */
}
else
{
        /* część 3 */
}
```

Powyższy kod można po prostu zastąpić czymś takim (zakładam zmienne 32-bitowe):
(przeskocz asemblerowy schemat bloku if/else)

```
        mov     eax, [a]
        cmp     eax, [b]
        jne     elseif1

        ; część 1

        jmp     po_if1
elseif1:
        cmp     eax, [c]        ; pamiętajmy, że [a] już jest w EAX
        jne     else1

        ; część 2

        jmp     po_if1
else1:

        ; część 3

po_if1:
```

Na szczególną uwagę zasługuje przypadek porównywania zmiennej do zera, gdzie zamiast instrukcji `CMP EAX, 0` użyjemy instrukcji `TEST EAX, EAX`.
Jeśli zaś trafi się Wam dość prosty kod w stylu:
(przeskocz przykład if/else)

```
if (a == b)             /* czy a jest równe b? */
{
        d = a;          /* wstaw wartość a do d */
}
else if (a == c)
{
        d = b;
}
else
{
        d = 0;
}
```

lub wyrażenie warunkowe, czyli coś postaci:

```
        d = (a == b)? a : 0;
```

To możecie (a nawet powinniście) użyć instrukcji warunkowego kopiowania danych CMOV*. Instrukcje te powodują znacznie wydajniejszą pracę procesora (który już nie musi co dwie instrukcje skakać i czytać nowych instrukcji). Pierwszy fragment kodu po przetłumaczeniu mógłby wyglądać tak (uwaga, TASM nie zna tych instrukcji. Niezbędne będą pewne makra, które można znaleźć w Internecie) :
(przeskocz tłumaczenie przykładu if/else)

```
xor     edx, edx        ; domyślna wartość, którą wstawimy
                        ; do zmiennej D wynosi zero

mov     eax, [a]
cmp     eax, [b]
cmove   edx, eax        ; gdy a == b, to do EDX wstaw
                        ; wartość A, czyli EAX

cmp     eax, [c]
cmove   edx, [b]        ; gdy a == c, to do EDX wstaw wartość B

mov     [d], edx        ; do D wstaw wynik operacji
                        ; (A, B lub domyślne 0)
```

A drugi:
(przeskocz tłumaczenie wyrażenia warunkowego)

```
xor     edx, edx        ; domyślna wartość to 0

mov     eax, [a]
cmp     eax, [b]        ; porównaj A z B

cmove   edx, eax        ; gdy równe, to EDX=[a]

mov     [d], edx        ; do D wstaw wynik operacji
```

Tylko nowoczesne kompilatory języka C potrafią wyczyniać takie sztuczki.
Podobne instrukcje istnieją także dla liczb i rejestrów zmiennoprzecinkowych: FCMOV*.

Pętle.

(przeskocz pętle)

Z pętlami jest trochę gorzej, gdyż jest ich więcej rodzajów.
Zacznijmy od pętli o znanej z góry ilości przejść (powtórzeń, iteracji), czy pętli typu

```
for (wyrażenia początkowe; warunek wykonywania; wyrażenie końcowe)
```

Na przykład:
(przeskocz przykład pętli for)

```
for (i=1; i<10; i=i+1)
{
        j=j+i;
}
```

zostałoby przetłumaczone na:
(przeskocz tłumaczenie tego przykładu)

```
        mov     ecx, 1          ; ECX to zmienna I. i=1
petla_for:
        cmp     ecx, 10
        jae     koniec_petli    ; wychodzimy, gdy i >= 10

        add     eax, ecx        ; EAX to zmienna J. j=j+i

        add     ecx, 1          ; i=i+1
        jmp     short petla_for
koniec_petli:
```

Jeśli warunkiem zakończenia pętli jest to, że pewna zmienna osiągnie zero, można stosować instrukcję LOOP.
Przykład:
(przeskocz drugą pętlę for)

```
for (i=10; i>0; i--)
{
        j=j+i;
}
```

może zostać przetłumaczony na 2 sposoby:
(przeskocz sposoby tłumaczenia)

```
; sposób 1:
        mov     ecx, 10         ; ECX to zmienna I. i=1
petla_for:
        cmp     ecx, 0          ; lub: test ecx, ecx
        jbe     koniec_petli    ; wychodzimy, gdy i <= 0

        add     eax, ecx        ; EAX to zmienna J. j=j+i

        sub     ecx, 1          ; i=i-1
        jmp     short petla_for
koniec_petli:
```

```
; sposób 2:
        mov     ecx, 10         ; ECX to zmienna I. i=1
petla_for:
        add     eax, ecx        ; EAX to zmienna J. j=j+i
        loop    petla_for       ; zmniejsz ECX o 1 i jeśli różny od
                                ;    zera, skocz do: petla_for
```

Pamiętajmy jednak, że instrukcja LOOP działa tylko na rejestrze (E)CX, więc jeśli chcemy mieć kilka zagnieżdżonych pętli, to przed każdą z nich (rozpoczynającą się zmianą rejestru ECX) musimy zachować ten rejestr (np. na stosie), a po zakończeniu pętli musimy przywrócić jego poprzednią wartość.

Sprawa z pętlami o nieznanej ilości powtórzeń nie jest o wiele trudniejsza. Pętla typu for jest całkowicie równoważna pętli while. Właśnie z tego skorzystamy, a kod niewiele będzie się różnić budową od poprzedniego przykładu.
Powiedzmy, że mamy taką pętlę:
(przeskocz ten przykład)

```
for (i=100; i+1<=n; i=i+2)
{
        j=j+i+4;
}
```

Możemy ją zastąpić równoważną konstrukcją:
(przeskocz zamianę for na while)

```
i=100;
while (i+1 <= n)
{
        j=j+i+4;
        i=1+2;
}
```

Otrzymujemy kod:
(przeskocz tłumaczenie while)

```
        mov     ecx, 100        ; ECX to zmienna I. i=100
nasza_petla:
        mov     ebx, ecx
        add     ebx, 1          ; EBX = i+1
        cmp     ebx, [n]        ; sprawdzamy, czy i+1 <= n
        ja      koniec_while    ; wychodzimy, gdy i+1 > n

        add     eax, ecx        ; EAX to zmienna J. j=j+i
        add     eax, 4          ; j=j+i+4

        add     ecx, 2          ; i=i+2
        jmp     short nasza_petla
koniec_while:
```

Ostatni rodzaj pętli to pętle typu do-while (repeat...until). Taka pętla różni się tym od poprzedniczek, że warunek jest sprawdzany po wykonaniu kodu pętli (czyli taka pętla zawsze będzie wykonana co najmniej raz). Daje to pewne możliwości optymalizacji kodu.
Popatrzmy na taki przykład:
(przeskocz przykład do-while)

```
do
```

```
{
        i=i+1;
        j=j-1;
} while ((i < n) && (j > 1));
```

Warunek wyjścia to: i >= n LUB j <= 1.
A teraz popatrzcie, co można z tym zrobić:
(przeskocz tłumaczenie do-while)

```
petla_do:
        add     ecx, 1          ; ECX to zmienna I. i=i+1
        add     edx, 1          ; EDX to zmienna J. j=j+1

        cmp     ecx, [n]
        jae     koniec          ; i >= n jest jednym z warunków
                                ; wyjścia. Drugiego nie musimy
                                ; sprawdzać, bo wynik i tak
                                ; będzie prawdą
        cmp     edx, 1
        jbe     koniec          ; j <= 1 to drugi warunek wyjścia

        jmp     petla_do
koniec:
```

Można przepisać to w lepszy sposób:
(przeskocz lepszy sposób)

```
petla_do:
        add     ecx, 1          ; ECX to zmienna I. i=i+1
        add     edx, 1          ; EDX to zmienna J. j=j+1

        cmp     ecx, [n]
        jae     koniec          ; i >= n jest jednym z warunków
                                ; wyjścia. Drugiego nie musimy
                                ; sprawdzać, bo wynik i tak
                                ; będzie prawdą

                                ; jeśli nadal tutaj jesteśmy,
                                ; to z pewnością i < n.
        cmp     edx, 1
        ja      petla_do        ; j <= 1 to drugi warunek
                                ; wyjścia. Jeśli j > 1,
                                ; to kontynuuj wykonywanie pętli.
                                ; Jeśli j < 1, to po prostu
                                ; opuszczamy pętlę:
koniec:
```

Jeśli warunek kontynuacji lub wyjścia z pętli jest wyrażeniem złożonym, to:

- jeśli składa się z alternatyw (działań typu OR, ||), to na pierwszym miejscu sprawdzajcie te warunki, które mają największą szansę być prawdziwe. Oszczędzicie w ten sposób czasu na bezsensowne sprawdzanie reszty warunków (wynik i tak będzie prawdą).
- jeśli składa się z koniunkcji (działań typu AND, &&), to na pierwszym miejscu sprawdzajcie te warunki, które mają największą szansę być fałszywe. Wynik całości i tak będzie fałszem.

Przykłady:

```
1)   a == 0 || (b > 1 && c < 2)
```

```
2)    (b < d || a == 1) && c > 0
```

W przypadku 1 najpierw sprawdzamy, czy a jest równe zero. Jeśli jest, to cały warunek jest prawdziwy. Jeśli nie jest, sprawdzamy najpierw ten z dwóch pozostałych, który ma największą szansę bycia fałszywym (jeśli któryś jest fałszywy, to wynik jest fałszem).
W przypadku 2 najpierw sprawdzamy, czy c jest większe od zera. Jeśli nie jest, to cały warunek jest fałszywy. Jeśli jest, to potem sprawdzamy ten z pozostałych warunków, który ma większą szansę bycia prawdziwym (jeśli któryś jest prawdziwy, to wynik jest prawdą).

Decyzje wielowariantowe (wyrażenia typu switch/case)

(przeskocz decyzje wielowariantowe)

Fragment kodu:
(przeskocz schemat switch/case)

```
switch (a)
{
        case 1: .....
        case 2: .....
        ....
        default: .....
}
```

w prosty sposób rozkłada się na serię wyrażeń if i else if (oraz else, o ile podano sekcję default). Te zaś już umiemy przedstawiać w asemblerze. Jest jednak jedna ciekawa sprawa: jeśli wartości poszczególnych przypadków case są zbliżone (coś w stylu 1, 2, 3 a nie 1, 20, 45), to możemy posłużyć się tablicą skoków (ang. jump table). W tej tablicy przechowywane są adresy fragmentów kodu, który ma zostać wykonany, gdy zajdzie odpowiedni warunek. Brzmi to trochę pokrętnie, dlatego szybko pokażę przykład.
(przeskocz przykład switch/case)

```
switch (a)
{
        case 1:
                j=j+1;
                break;
        case 2:
                j=j+4;
                break;
        case 4:
                j=j+23;
                break;
        default:
                j=j-1;
}
```

A teraz tłumaczenie:
(przeskocz tłumaczenie przykładu switch/case)

```
        mov     eax, [a]
        cmp     eax, 1                  ; jeśli a < 1 lub a > 5,
                                        ; to na pewno default
        jb      sekcja_default

        cmp     eax, 5
        ja      sekcja_default
```

```
        jmp     [przypadki + eax*2 - 2]

przyp1:
        add     dword ptr [j], 1        ; NASM/FASM: bez słowa PTR
        jmp     koniec

przyp2:
        add     dword ptr [j], 4        ; NASM/FASM: bez słowa PTR
        jmp     koniec

przyp4:
        add     dword ptr [j], 23       ; NASM/FASM: bez słowa PTR
        jmp     koniec

sekcja_default:
        sub     dword ptr [j], 1

koniec:

....
przypadki       dw offset przyp1, offset przyp2
                dw offset sekcja_default, offset przyp4
; NASM:
;przypadki      dw przyp1, przyp2, sekcja_default, przyp4
```

Kod najpierw sprawdza, czy a ma szansę być w którymś z przypadków (jeśli nie jest, to oczywiście wykonujemy sekcję default). Potem, jeśli a=1, to skacze pod etykietę w `zmienne [przypadki + 1*2 - 2 ] = [przypadki] = przyp1`.
Podobnie, jeśli a=2, skoczymy do przyp2. Jeśli a=3, skoczymy do sekcji default, a jeśli a=4, skoczymy do sekcji przyp4.

Od razu widać wielką zaletę takiego rozwiązania: w jednej jedynej instrukcji wiemy, gdzie musimy skoczyć. Jak liczba przypadków będzie wzrastać, zauważymy też wadę tego rozwiązania: rozmiar tablicy szybko rośnie (wynosi on różnicę między wartością najwyższą możliwą a najniższą możliwą pomnożoną przez 2 bajty). Dlatego to rozwiązanie jest nieprzydatne dla możliwych wartości: {1, 20, 45} (42 wartości z 45 byłyby nieużywane, czyli wskazujące na sekcję default - zdecydowane marnotrawienie pamięci). W takim przypadku lepiej użyć tradycyjnej metody if/else if/else.

Mam nadzieję, że wiedza zawarta w tej części kursu umożliwi Wam pisanie lepszych i bardziej złożonych programów niż to było do tej pory. Teraz będziecie wiedzieć, co tak właściwie robię kompilatory, tłumacząc niektóre wyrażenia kontrolne. Ta wiedza pomoże Wam pisać lepsze programy w językach wyższego poziomu (gdyż już teraz wiecie, jak zapisywać wyrażenia logiczne tak, by dostać najbardziej wydajny kod).

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia:

1. Zaimplementować zdanie:
   Jeśli EAX jest równy EBX lub ECX nie jest równy EBP, to do EDX wstaw EAX, inaczej do ECX wstaw 0.

2. Zaimplementować zdanie (użyć instrukcji warunkowego przesuwania):
   Jeśli EAX jest równy EBX lub ECX nie jest równy EBP, to do EDX wstaw EAX, inaczej do EDX wstaw 0.

3. Napisać program, który liczy sumę liczb od 10 do dowolnej liczby wpisanej w kodzie/czytanej z linii poleceń.

4. Zaimplementować zdanie:
   Dopóki ECX jest większe od 1, zmniejsz ECX o 2.

5. Zaimplementować zdanie:
   Zwiększaj EAX o 3, dopóki EAX jest mniejsze od 100.

Jak pisać programy w języku asembler?
Część 16 - Operacje na łańcuchach znaków. Wyrażenia regularne.

Jak wiemy, łańcuch znaków to nic innego jak jednowymiarowa tablica bajtów. Dlatego podane tutaj informacje tak samo działają dla tablic.

W zestawie instrukcji procesora przeznaczonych jest klika rozkazów przeznaczonych specjalnie do operacji na łańcuchach znaków: MOVS, CMPS, SCAS, LODS, STOS. To nimi właśnie teraz się zajmiemy.

Rozkazy te operują na łańcuchach spod DS:[SI/ESI/RSI] lub ES:[DI/EDI/RDI] lub obydwu. Rejestry segmentowe nie będą tutaj grać dużej roli bo pokazują zawsze na ten sam segment, więc będziemy je pomijać. Oraz, zajmiemy się omówieniem instrukcji tylko na ESI oraz EDI, pomijając rejestry 64-bitowe, dla których wszystko wygląda analogicznie.

Instrukcje występują w 4 formach: *B, *W, *D (dla 32-bitowych) i *Q (dla 64-bitowych). Operują one odpowiednio na bajtach, słowach, podwójnych słowach i danych 64-bitowych. Po każdym wykonaniu pojedynczej operacji zwiększają rejestry ESI i EDI o 1, 2, 4 lub 8, przechodząc tym samym na następne elementy.

UWAGA: Zwiększaniem rejestrów *SI i *DI steruje flaga kierunku DF: jeśli równa 0, oba rejestry są zwiększane, jeśli 1 - są zmniejszane o odpowiednią liczbę (co pozwala np. na przeszukiwanie łańcuchów wspak). Flagę DF można wyczyścić instrukcją CLD, a ustawić instrukcją STD.

MOVS

(przeskocz MOVS)
Zasadą działania tej instrukcji jest przeniesienie odpowiedniej ilości bajtów spod DS:[SI] i umieszczenie ich pod ES:[DI]. Ale przeniesienie co najwyżej 4 bajtów to przecież żaden wysiłek:

```
mov     eax, ds:[si]            ; NASM/FASM: mov eax, [ds:si]
mov     es:[di], eax
```

Dlatego wymyślono prefiks REP (powtarzaj). Jest on ważny tylko dla instrukcji operujących na łańcuchach znaków oraz instrukcji INS i OUTS. Powoduje on powtórzenie działania instrukcji (E)CX razy. Teraz już widać możliwości tej instrukcji. Chcemy przenieść 128 bajtów? Proszę bardzo:

```
mov     ax, seg zrodlo
mov     ds, ax
mov     si, offset zrodlo       ; NASM/FASM: mov si, zrodlo

mov     ax, seg cel
mov     es, ax
mov     di, offset cel          ; NASM/FASM: mov di, cel

cld                             ; sprawdzaj do przodu
mov     cx, 128
rep     movsb
```

Oczywiście, dwie ostatnie linijki można było zastąpić czymś takim i też by podziałało:

```
mov     cx, 32
rep     movsd
```

Sposób drugi oczywiście jest lepszy, bo ma mniej operacji (choć najwiecej czasu i tak zajmuje samo rozpędzenie się instrukcji REP).

Instrukcji REP MOVS* można używać do przenoszenia małej ilości danych. Gdy ilości danych rosną, lepiej sprawują się MMX i SSE (patrz: część 6.)

CMPS

(przeskocz CMPS)
Ta instrukcja porównuje odpowiednią ilość bajtów spod DS:[SI] i ES:[DI]. Ale nas oczywiście nie interesuje porównywanie pojedynczych ilości. Myślimy więc o prefiksie REP, ale po chwili zastanowienia dochodzimy do wniosku, że w ten sposób otrzymamy tylko wynik ostatniego porównania, wszystkie wcześniejsze zostaną zaniedbane. Dlatego wymyślono prefiksy REPE/REPZ (powtarzaj, dopóki równe/flaga ZF ustawiona) oraz REPNE/REPNZ (powtarzaj, dopóki nie równe/flaga ZF = 0).
Na przykład, aby sprawdzić równość dwóch łańcuchów, zrobimy tak:

```
mov     ax, seg lancuch1
mov     ds, ax
mov     si, offset lancuch1     ; NASM/FASM: mov si, lancuch1

mov     ax, seg lancuch2
mov     es, ax
mov     di, offset lancuch2     ; NASM/FASM mov di, lancuch2

mov     cx, 256         ; tyle bajtów maksymalnie chcemy porównać
cld
repe    cmpsb                   ; dopóki są równe, porównuj dalej
jnz     lancuchy_nie_rowne
```

REPE przestanie działać na pierwszych różniących się bajtach. W CX otrzymamy pewną liczbę. Różnica liczby 256 i tej liczby mówi o ilości identycznych znaków i jednocześnie o tym, na której pozycji znajdują się różniące się znaki.
Oczywiście, jeśli po ukończeniu REPE rejestr CX=0, to znaczy że sprawdzono wszystkie znaki (i wszystkie dotychczas były równe). Wtedy flagi mówią o ostatnim porównaniu.
REPE CMPS ustawia flagi jak normalna instrukcja CMP.

SCAS

(przeskocz SCAS)
Ta instrukcja przeszukuje łańcuch znaków pod ES:[DI] w poszukiwaniu bajtu z AL, słowa z AX lub podwójnego słowa z EAX. Służy to do szybkiego znalezienia pierwszego wystąpienia danego elementu w łańcuchu.
Przykład: znaleźć pozycję pierwszego wystąpienia litery Z w zmiennej lancuch1:

```
                mov     ax, seg lancuch1
                mov     es, ax

                mov     al, "Z"         ; poszukiwany element
                mov     di, lancuch1
                mov     cx, 256
                cld
                repne   scasb           ; dopóki sprawdzany znak różny
                                        ; od "Z", szukaj dalej

                je      znaleziono

                mov     di, -1          ; gdy nie znaleziono,
                                        ; zwracamy -1
                jmp     koniec

        znaleziono:
                sub     di, lancuch1    ; DI = pozycja znalezionego
                                        ; znaku w łańcuchu
```

REPNE przestanie działać w dwóch przypadkach: CX=0 (wtedy nie znaleziono szukanego elementu) oraz wtedy, gdy ZF=1 (gdy po drodze natrafiła na szukany element, wynik porównania ustawił flagę ZF).

LODS

(przeskocz LODS)
Instrukcje LODS* pobierają do AL/AX/EAX odpowiednią ilość bajtów spod DS:[SI]. Jak widać, prefiksy REP* nie mają tutaj sensu, bo w rejestrze docelowym i tak zawsze wyląduje ostatni element.
Ale za to tej instrukcji można używać do pobierania poszczególnych znaków do dalszego sprawdzania, np.

```
                cld
        petla:
                lodsb                           ; pobierz kolejny znak

                cmp     al, 13
                jne     nie_enter

                cmp     al, "0"
                je      al_to_zero

                ....

                loop    petla
```

STOS

(przeskocz STOS)
Instrukcja ta umieszcza AL/AX/EAX pod ES:[DI]. Poza oczywistym zastosowaniem, jakim jest np. zapisywanie kolejnych podawanych przez użytkownika znaków gdzieś do tablicy, STOS można też użyć do szybkiej inicjalizacji tablicy w czasie działania programu lub do wyzerowania pamięci:

```
            mov     ax, seg tablica
            mov     es, ax
            mov     di, offset tablica ; NASM/FASM: mov di, tablica

            mov     eax, 11223344h
            mov     cx, 1000
            cld
            rep     stosd
            ...

tablica  dd 1000 dup(0)         ; NASM/FASM:
                                ; tablica: TIMES 1000 dd 0
```

Wyrażenia regularne

Wyrażenia regularne (regular expressions, regex) to po prostu ciągi znaczków, przy użyciu których możemy opisywać dowolne łańcuchy znaków (adresy e-mail, WWW, nazwy plików z pełnymi ścieżkami, ...).
Na wyrażenie regularne składają się różne symbole. Postaram się je teraz po krótce omówić.

- aaa (dowolny ciąg znaków) - reprezentuje wszystkie łańcuchy, które go zawierają, np. laaaaaaaaaato.
- ^ - oznacza początek linii (wiersza). Na przykład wyrażenie ^asembler reprezentuje wszystkie linie, które zaczynają się od ciągu znaków asembler. Innymi słowy, każda linia zaczynająca się od asembler pasuje do tego wyrażenia.
- $ - oznacza koniec linii. Na przykład wyrażenie asm$ reprezentuje wszystkie linie, które kończą się na asm.
- . (kropka) - dowolny znak (z wyjątkiem znaku nowej linii). Na przykład wyrażenie ^a.m$ reprezentuje linie, które zawierają w sobie tylko a*m, gdzie gwiazdka to dowolny znak (w tym cyfry). Do tego wzorca pasują asm, aim, a0m i wiele innych.
- | (Shift+BackSlash)- oznacza alternatywę. Na przykład wyrażenie a|b|z reprezentuje dowolną z tych trzech liter i żadną inną.
- (,) - nawiasy służą do grupowania wyrazów. Na przykład ^(aa)|(bb)|(asm) reprezentuje linie, które zaczynają się od aa, bb lub asm.
- [,] - wyznaczają klasę znaków. Na przykład wszystkie wyrazy zaczynające się od k, a lub j pasują do wzorca [ajk].*. Można tutaj podawać przedziały znaków - wtedy 2 skrajne znaki oddzielamy myślnikiem, np. [a-z]. Umieszczenie w środku znaku daszka ^ oznacza przeciwieństwo, np. [^0-9] reprezentuje znaki, które nie są cyfrą (a tym samym wszystkie ciągi nie zawierające cyfr).
- ? - oznacza co najwyżej 1 wystąpienie poprzedzającego znaku lub grupy. Na przykład, ko?t reprezentuje wyrazy kot i kt, ale już nie koot.
- * - oznacza dowolną ilość wystąpień poprzedzającego znaku/grupy. Wyrażenie ko*t reprezentuje więc wyrazy kt, kot, koot, kooot, itd.
- + - oznacza co najmniej jedno wystąpienie poprzedzającego znaku/grupy. Na przykład al(fa)+ reprezentuje alfa, alfafa, alfafafa itd, ale nie al.
- {n} - oznacza dokładnie n wystąpień poprzedzającego znaku/grupy. Wyrażenie [0-9]{7} reprezentuje więc dowolny ciąg składający się dokładnie z 7 cyfr.
- {n,} - oznacza co najmniej n wystąpień poprzedzającego znaku/grupy. Wyrażenie [a-z]{2,} reprezentuje więc dowolny ciąg znaków składający się co najmniej z 2 małych liter.
- {n,m} - oznacza co najmniej n i co najwyżej m wystąpień poprzedzającego znaku/grupy. Więc wyrażenie [A-M]{3,7} reprezentuje dowolny ciąg znaków składający się z co najmniej 3 i co najwyżej 7 wielkich liter z przedziału od A do M.

- Jeśli w łańcuchu może wystąpić któryś ze znaków specjalnych, należy go w wyrażeniu poprzedzić odwrotnym ukośnikiem \.

Dalsze przykłady:

- ([a-zA-Z0-9]+\.?)+[a-zA-Z]+@([a-zA-Z0-9]+\.)+[a-zA-Z]{2,4} - adres e-mail (zapisany tak, by login ani domena nie kończyły się kropką)
- ([a-zA-Z]{3,6}://)?([a-zA-Z0-9/\-]+\.?)+[a-zA-Z0-9]+(#[a-zA-Z0-9\-]+)? - adres (z protokołem lub bez) zasobu na serwerze (zapisany tak, by nie kończył się kropką, może zawierać myślniki a w ostatnim członie także znak #)

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Napisać program zawierający 2 tablice DWORDów o wymiarach 17 na 31, po czym w trakcie działania programu wypełnić każde pole pierwszej wartością FFEEDDCCh. Potem, 8 pierwszych elementów skopiować do drugiej tablicy, a resztę drugiej wypełnić wartością BA098765h. Wtedy porównać zawartość obu tablic i wyliczyć pierwszą pozycję, na której się różnią (powinna oczywiście wynosić 9)

2. Napisać wyrażenie regularne, które opisze:
   - wszystkie wyrażenia deklaracji zmiennych: DB, DW, DP, DQ, DT
   - znacznik HTML bez atrybutów, czyli coś wyglądające tak: < PRE > lub tak: < /LI > (bez spacji).
   - liczbę szesnastkową dowolnej niezerowej długości z ewentualnym przedrostkiem 0x albo (do wyboru) przyrostkiem H lub h.

Jak pisać programy w języku asembler?
Część 17 - Pobieranie i wyświetlanie, czyli jak komunikować się ze światem.

O ile wyświetlanie i pobieranie od użytkownika tekstów jest łatwe do wykonania - wystarczy uruchomić tylko jedną funkcję systemową (ah=9 i ah=0A przerwania 21h) - to pobieranie i wyświetlanie na przykład liczb wcale nie jest takie proste i każdemu może przysporzyć problemów. W tej części podam parę algorytmów, dzięki którym każdy powinien sobie z tym poradzić.

Wyświetlanie tekstu

(przeskocz wyświetlanie tekstu)
Co prawda wszyscy już to umieją, ale dla porządku też o tym wspomnę.
Wszyscy znają funkcję ah=9 przerwania DOSa - wystraczy podać jej łańcuch znaków zakończony znakiem dolara, a ona wszystko sama wyświetli. Ale co, jeśli chcemy wyświetlić znak dolara? Albo nie mamy DOSa do dyspozycji?
Otóż, są jeszcze inne funkcje służące do wyświetlania tekstu na ekranie - na przykład funkcja ah=0E przerwania 10h (wyświetla po jednym znaku), funkcja ah=2 przerwania DOSa (wyświetla po jednym znaku), funkcja ah=13h przerwania 10h (wyświetla całe napisy, można podać pozycję napisu i kolor każdego znaku). Zawsze można też wyświetlać tekst ręcznie.

Pobieranie tekstu

(przeskocz pobieranie tekstu)
Do pobierania tekstów od użytkownika służyć może funkcja AH=0A przerwania DOSa. Wystarczy podać jej adres bufora takiej konstrukcji:

```
bufor   db 20           ; maksymalna ilość znaków do pobrania
        db 0            ; tu dostaniemy, ile znaków pobrano
dane:   times 22 db "$" ; miejsce na dane
```

DOS wczyta dane z klawiatury (co najwyżej tyle bajtów, ile podaliśmy), w drugim bajcie zwróci nam, ile faktycznie przeczytano (Enter kończy), a od trzeciego bajtu zaczynają się same dane. Można się do nich odwoływać albo poprzez `[dane]`, albo poprzez `[bufor+2]`.
Jeśli nie ma DOSa do dyspozycji, można korzystać z funkcji ah=0 przerwania klawiatury int 16h.

Wyświetlanie liczb

(przeskocz wyświetlanie liczb)
Są generalnie dwa podejścia do tego problemu:

1. dzielenie przez coraz mniejsze potęgi liczby 10 (zaczynając od najwyższej odpowiedniej) i wyświetlanie ilorazów
2. dzielenie przez 10 i wyświetlanie reszt wspak

Podejście pierwsze jest zilustrowane takim kodem dla liczb 16-bitowych (0-65535):

```
        mov     ax, [liczba]
        xor     dx, dx
        mov     cx, 10000
        div     cx
        or      al, '0'
        ; wyświetl AL jako znak
        mov     ax, dx
        xor     dx, dx
        mov     cx, 1000
        div     cx
        or      al, '0'
        ; wyświetl AL jako znak
        mov     ax, dx
        mov     cl, 100
        div     cl
        or      al, '0'
        ; wyświetl AL jako znak
        mov     al, ah
        xor     ah, ah
        mov     cl, 10
        div     cl
        or      ax, '00'
        ; wyświetl AL jako znak
        ; potem wyświetl AH jako znak
```

Jak widać, im więcej cyfr może mieć liczba, tym więcej będzie takich bloków. Trzeba zacząć od najwyższej możliwej potęgi liczby 10, bo inaczej może dojść do przepełnienia. W każdym kroku dzielnik musi mieć o jedno zero mniej, gdyż inaczej nie uda się wyświetlić prawidłowego wyniku (może być dwucyfrowy i wyświetli się tylko jakiś znaczek). Ponadto, jeśli liczba wynosi na przykład 9, to wyświetli się 00009, czyli wiodące zera nie będą skasowane. Można to oczywiście ominąć.

Podejście drugie jest o tyle wygodniejsze, że można je zapisać za pomocą pętli. Jest to zilustrowane procedurą _pisz_ld z części czwartej oraz kodem z mojej biblioteki:

```
        xor     si, si                  ; indeks do bufora
        mov     cx, 10                  ; dzielnik
_pisz_l_petla:                          ; wpisujemy do bufora reszty z
                                        ; dzielenia liczby przez 10,
        xor     dx, dx                  ; czyli cyfry wspak
        div     cx                      ; dziel przez 10
        or      dl, '0'                 ; dodaj kod ASCII cyfry zero
        mov     [_pisz_bufor+si], dl    ; zapisz cyfrę do bufora
        inc     si                      ; zwiększ indeks
        test    ax, ax                  ; dopóki liczba jest różna od 0
        jnz     _pisz_l_petla

_pisz_l_wypis:
        mov     al, [_pisz_bufor+si-1]  ; pobierz znak z bufora
        call    far _pisz_z             ; wyświetla znak
        dec     si                      ; przejdź na poprzedni znak
        jnz     _pisz_l_wypis
```

Zmienna _pisz_bufor to bufor odpowiedniej liczby bajtów.

Pobieranie liczb

(przeskocz pobieranie liczb)

Do tego zagadnienia algorytm jest następujący:

1. wczytaj łańcuch znaków od razu w całości lub wczytuj znak po znaku w kroku 3
2. wstępnie ustaw wynik na 0
3. weź kolejny znak z wczytanego łańcucha znaków (jeśli już nie ma, to koniec)
4. zamień go na jego wartość binarną. Jeśli znak wczytałeś do AL, to wystarczy:
   `sub al, '0'`
5. przemnóż bieżący wynik przez 10
6. dodaj do niego wartość AL otrzymaną z kroku 4
7. skacz do 3

Przykładową ilustrację można znaleźć także w mojej bibliotece:

```
        xor     bx, bx          ; miejsce na liczbę
l_petla:
        call    far _we_z       ; pobierz znak z klawiatury

        cmp     al, lf          ; czy Enter?
        je      l_juz           ; jeśli tak, to wychodzimy
        cmp     al, cr
        je      l_juz
                                ; przepuszczamy Spacje:
        cmp     al, spc
        je      l_petla

        cmp     al, '0'         ; jeśli nie cyfra, to błąd
        jb      l_blad
        cmp     al, '9'
        ja      l_blad

        and     al, 0fh         ; izolujemy wartość (sub al, '0')
        mov     cl, al
        mov     ax, bx

        shl     bx, 1           ; zrobimy miejsce na nową cyfrę
        jc      l_blad

        shl     ax, 1
        jc      l_blad
        shl     ax, 1
        jc      l_blad
        shl     ax, 1
        jc      l_blad

        add     bx, ax          ; BX=BX*10 - bieżącą liczbę mnożymy przez 10
        jc      l_blad

        add     bl, cl          ; dodajemy cyfrę
        adc     bh, 0
        jc      l_blad          ; jeśli przekroczony limit, to błąd

        jmp     short l_petla
```

Sprawdzanie rodzaju znaku

Powiedzmy, że użytkownik naszego programu wpisał nam jakieś znaki (tekst, liczby). Jak teraz sprawdzić, co dokładnie otrzymaliśmy? Sprawa nie jest trudna, lecz wymaga czasem zastanowienia i tablicy ASCII pod ręką.

1. Cyfry.
   Cyfry w kodzie ASCII zajmują miejsca od 30h (zero) do 39h (dziewiątka). Wystarczy więc sprawdzić, czy wczytany znak mieści się w tym zakresie:

```
cmp     al, '0'
jb      nie_cyfra
cmp     al, '9'
ja      nie_cyfra
; tu wiemy, że AL reprezentuje cyfrę.
; Pobranie wartości tej cyfry:
and     al, 0fh ; skasuj wysokie 4 bity, zostaw 0-9
```

2. Litery.
   Litery, podobnie jak cyfry, są uporządkowane w kolejności w dwóch osobnych grupach (najpierw wielkie, potem małe). Aby sprawdzić, czy znak w AL jest literą, wystarczy kod

```
        cmp     al, 'A'
        jb      nie_litera      ; na pewno nie litera
        cmp     al, 'Z'
        ja      sprawdz_male    ; na pewno nie wielka,
                                ; sprawdź małe
        ; tu wiemy, że AL reprezentuje wielką literę.
        ; ...
sprawdz_male:
        cmp     al, 'a'
        jb      nie_litera      ; na pewno nie litera
        cmp     al, 'z'
        ja      nie_litera
        ; tu wiemy, że AL reprezentuje małą literę.
```

3. Cyfry szesnastkowe.
   Tu sprawa jest łatwa: należy najpierw sprawdzić, czy dany znak jest cyfrą. Jeśli nie, to sprawdzamy, czy jest wielką literą z zakresu od A do F. Jeśli nie, to sprawdzamy, czy jest małą literą z zakresu od a do f. Wystarczy połączyć powyższe fragmenty kodu. Wyciągnięcie wartości wymaga jednak więcej kroków:

```
; jeśli AL to cyfra '0'-'9'
and     al, 0fh
; jeśli AL to litera 'A'-'F'
sub     al, 'A' - 10
; jeśli AL to litera 'a'-'f'
sub     al, 'a' - 10
```

   Jeśli AL jest literą, to najpierw odejmujemy od niego kod odpowiedniej (małej lub wielkiej) litery A. Dostajemy wtedy wartość od 0 do 5. Aby dostać realną wartość danej litery w kodzie szesnastkowym, wystarczy teraz dodać 10. A skoro AL-'A'+10 to to samo, co AL-('A'-10), to już wiecie, skąd się wzięły powyższe instrukcje.

4. Przerabianie wielkich liter na małe i odwrotnie.
   Oczywistym sposobem jest odjęcie od litery kodu odpowiedniej litery A (małej lub wielkiej), po czym dodanie kodu tej drugiej, czyli:

```
; z małej na wielką
```

```
sub     al, 'a'
add     al, 'A'
; z wielkiej na małą
sub     al, 'A'
add     al, 'a'
```

lub nieco szybciej:

```
; z małej na wielką
sub     al, 'a' - 'A'
; z wielkiej na małą
sub     al, 'A' - 'a'
```

Ale jest lepszy sposób: patrząc w tabelę kodów ASCII widać, że litery małe od wielkich różnią się tylko jednym bitem - bitem numer 5. Teraz widać, że wystarczy

```
; z małej na wielką
and     al, 5fh
; z wielkiej na małą
or      al, 20h
```

Poprzednia część kursu (Alt+3)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Korzystając z przedstawionych tu algorytmów, napisz algorytmy wczytujące i wyświetlające liczby dziesiętne 8-bitowe.
2. Korzystając z przedstawionych tu algorytmów, napisz algorytmy wczytujące i wyświetlające liczby szesnastkowe 16-bitowe (wystarczy zmienić liczby, przez które mnożysz i dzielisz oraz to, jakie znaki są dozwolone i wyświetlane - dochodzą litery od A do F).

Dynamiczna alokacja pamięci

Już w średnio zaawansowanych programach pojawia się potrzeba dynamicznego rezerwowania pamięci. Na przykład, użytkownik podaje nam rozmiar tablicy a my musimy taką tablicę utworzyć i na niej operować (nie znając wcześniej nawet maksymalnego jej rozmiaru). Rozwiązaniem takich problemów jest właśnie dynamiczna alokacja pamięci.
Pod DOSem pamięć alokuje się funkcją AH=48h przerwania 21h, w BX podając liczbę paragrafów do zaalokowania (1 paragraf = 16 bajtów). Jeśli alokacja pamięci się powiedzie, w AX otrzymujemy numer segmentu z zarezerwowaną dla nas pamięcią. Programy typu .com z założenia zajmują całą dostępną pamięć, więc aby coś zaalokować, należy najpierw trochę pamięci zwolnić.
Zwalnianie pamięci wykonuje się funkcją 49h, w ES podając numer segmentu do zwolnienia.

Jak widać, teoria nie jest skomplikowana. Przejdźmy więc może do przykładu. Ten krótki programik ma za zadanie zaalokować 160 bajtów, wyzerować je i na końcu zwolnić.
(przeskocz program)

```
; Dynamiczna alokacja pamięci pod DOSem
;
; Autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
;
; nasm -f obj -o allocdos.obj allocdos.asm
; val allocdos.obj,allocdos.exe,,,

section .text

..start:
        mov     ah, 49h
        mov     es, [ds:2ch]    ; ES=segment naszych zmiennych środowiskowych
        int     21h             ; zwalniamy

        mov     ax, seg info
        mov     ds, ax          ; DS = nasz segment danych (w razie czego)

        mov     ah, 48h         ; rezerwuj pamięć
        mov     bx, 10          ; 10 paragrafów
        int     21h
        jc      problem         ; CF=1 oznacza błąd

        mov     es, ax          ; ES = przydzielony segment

        mov     ah, 9
        mov     dx, info
        int     21h             ; wyświetl pierwszy napis

        mov     cx, 160         ; tyle bajtów wyzerujemy
        xor     di, di          ; poczynając od adresu 0 w nowym segmencie
        xor     al, al          ; AL = 0
        cld                     ; kierunek: do przodu
        rep     stosb           ; zerujemy obszar

        mov     ah, 49h
        int     21h             ; zwalniamy pamięć
        jc      problem
```

```
        mov     ah, 9
        mov     dx, info2
        int     21h

problem:
        mov     ax, 4c00h
        int     21h

koniec:

section .data

info    db      "Udana alokacja pamieci.",10,13,"$"
info2   db      "Udane zwolnienie pamieci.",10,13,"$"

; program typu .exe musi mieć zadeklarowany stos
section stack stack
        resb 400h
```

Zwalnianie pamięci w programach typu .com polega na zmianie rozmiaru segmentu kodu. Wykonuje się to funkcją AH=4Ah przerwania 21h, w ES podając segment, którego rozmiar chcemy zmienić (nasz segment kodu - CS), a w BX - nowy rozmiar w paragrafach.
Typowy kod wygląda więc tak:

```
        mov     ax, cs
        mov     es, ax          ; będziemy zmieniać rozmiar segmentu kodu
        mov     bx, koniec      ; BX = rozmiar segmentu kodu
        shr     bx, 4           ; BX /= 16 - rozmiar w paragrafach
        inc     bx              ; +1, żeby nie obciąć naszego programu
        mov     ah, 4ah         ; funkcja zmiany rozmiaru
        int     21h
```

UWAGA: Etykieta koniec musi być ostatnim elementem w kodzie programu.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Pisanie własnych bibliotek w języku asembler

Pewnie zdarzyło się już wam usłyszeć o kimś innym:
Ależ on(a) jest świetnym(ą) programistą(ką)! Nawet pisze własne biblioteki!
Pokażę teraz, że nie jest to trudne, nie ważne jak przerażającym się to może wydawać. Osoby, które przeczytają ten artykuł i zdobędą troszkę wprawy będą mogły mówić:
Phi, a co to za filozofia pisać własne biblioteki!

Zacznijmy więc od pytania: co powinno się znaleźć w takiej bibliotece?
Mogą to być:

- Funkcje wejścia i wyjścia, podobnie jak np. w języku C
- Funkcje, które już przepisywaliśmy ze 20 razy w różnych programach
- Sprawdzone funkcje, napisane przez kogoś innego, których nie umielibyśmy sami napisać, lub które po prostu mogą się przydać

Co to zaś jest to owa biblioteka?
Jest to plik (najczęściej z rozszerzeniem .lib), na który składa się skompilowany kod, a więc np. pliki .obj. Biblioteka eksportuje na zewnątrz nazwy procedur w niej zawartych, aby linker wiedział, jaki adres podać programowi, który chce skorzystać z takiej procedury.
Będę w tym artykule używał składni i linii poleceń Turbo Assemblera (TASMa) firmy Borland z linkerem TLink i bibliotekarzem TLib oraz NASMa (Netwide Assembler) i FASMa (Flat Assembler z linkerem ALink i darmowym bibliotekarzem znalezionym w Internecie (patrz linki na dole strony).

Napiszmy więc jakiś prosty kod źródłowy. Oto on:
(przeskocz przykładowy moduł biblioteki)

```
; wersja TASM

public _graj_dzwiek

biblioteka_dzwiek       segment byte    public "bibl"
assume cs:biblioteka_dzwiek

_graj_dzwiek            proc    far



; wejście:      BX = żądana częstotliwość dźwięku w Hz, co najmniej 19
;               CX:DX = czas trwania dźwięku w mikrosekundach
;
; wyjście:      CF = 0 - wykonane bez błędów
;               CF = 1 - błąd: BX za mały



czasomierz      equ     40h     ;numer portu programowalnego czasomierza
klawiatura      equ     60h     ;numer portu kontrolera klawiatury

        pushf                   ; zachowujemy modyfikowane rejestry
        push ax
        push dx
        push si
```

```
        cmp bx,19               ;najniższa możliwa częstotliwość to ok. 18Hz
        jb _graj_blad


        in   al,klawiatura+1    ; port B kontrolera klawiatury
        or   al,3               ; ustawiamy bity: 0 i 1 - włączamy głośnik i
                                ; bramkę od licznika nr. 2 czasomierza
                                ; do głośnika
        out  klawiatura+1,al



        mov si,dx               ;zachowujemy DX

        mov dx,12h
        mov ax,34ddh
        div bx                  ;AX = 1193181 / częstotliwość, DX=reszta

        mov dl,al               ;zachowujemy młodszy bajt dzielnika
                                ; częstotliwości



        mov al,0b6h

        out czasomierz+3,al     ;wysyłamy komendę:
                                ; (bity 7-6) wybieramy licznik nr. 2,
                                ; (bity 5-4) będziemy pisać najpierw bity 0-7
                                ;                potem bity 8-15
                                ;(bity 3-1) tryb 3:generator fali kwadratowej
                                ; (bit 0)    licznik binarny 16-bitowy

        mov al,dl               ; odzyskujemy młodszy bajt
        out czasomierz+2,al     ; port licznika nr. 2 i bity 0-7 dzielnika
                                ; częstotliwości
        mov al,ah
        out czasomierz+2,al     ; bity 8-15

        mov dx,si               ;odzyskujemy DX


_graj_pauza:
        mov ah,86h
        int 15h                 ; pauza o długości CX:DX mikrosekund

        jnc _graj_juz
        dec dx
        sbb cx,0                ; w razie błędu zmniejszamy CX:DX
        jmp short _graj_pauza

_graj_juz:

        in   al,klawiatura+1
        and  al,not 3           ; zerujemy bity: 0 i 1 - wyłączamy głośnik
                                ; i bramkę
        out  klawiatura+1,al

        pop si                  ; przywracamy rejestry
        pop dx
        pop ax
        popf
        clc                     ; brak błędu
```

```
        retf


_graj_blad:
        pop si                  ; przywracamy rejestry
        pop dx
        pop ax
        popf
        stc                     ; błąd

        retf

_graj_dzwiek            endp

biblioteka_dzwiek       ends
end
```

Teraz ten sam kod w składni NASMa/FASMa:
(przeskocz moduł w składni NASMa/FASMa)

```
; wersja NASM

global  _graj_dzwiek
; w FASMie:
;       format COFF
;       use16
;       PUBLIC _graj_dzwiek

segment biblioteka_dzwiek       ; FASM: section ".text" code

_graj_dzwiek:



; wejście:      BX = żądana częstotliwość dźwięku w Hz, co najmniej 19
;               CX:DX = czas trwania dźwięku w mikrosekundach
;
; wyjście:      CF = 0 - wykonane bez błędów
;               CF = 1 - błąd: BX za mały



czasomierz      equ     40h     ;numer portu programowalnego czasomierza
klawiatura      equ     60h     ;numer portu kontrolera klawiatury

        pushf
        push ax
        push dx
        push si


        cmp bx,19               ;najniższa możliwa częstotliwość to ok. 18Hz
        jb _graj_blad


        in   al,klawiatura+1    ; port B kontrolera klawiatury
        or   al,3               ; ustawiamy bity: 0 i 1 - włączamy głośnik i
                                ; bramkę od licznika nr. 2 czasomierza
                                ; do głośnika
        out  klawiatura+1,al
```

```
        mov si,dx               ;zachowujemy DX

        mov dx,12h
        mov ax,34ddh
        div bx                  ;AX = 1193181 / częstotliwość, DX=reszta

        mov dl,al               ;zachowujemy młodszy bajt dzielnika
                                ; częstotliwości



        mov al,0b6h

        out czasomierz+3,al     ;wysyłamy komendę:
                                ; (bity 7-6) wybieramy licznik nr. 2,
                                ; (bity 5-4) będziemy pisać najpierw bity 0-7
                                ;                potem bity 8-15
                                ;(bity 3-1) tryb 3:generator fali kwadratowej
                                ; (bit 0)    licznik binarny 16-bitowy

        mov al,dl               ; odzyskujemy młodszy bajt
        out czasomierz+2,al     ; port licznika nr. 2 i bity 0-7 dzielnika
                                ; częstotliwości
        mov al,ah
        out czasomierz+2,al     ; bity 8-15

        mov dx,si               ;odzyskujemy DX


_graj_pauza:
        mov ah,86h
        int 15h                 ; pauza o długości CX:DX mikrosekund

        jnc _graj_juz
        dec dx
        sbb cx,0                ; w razie błędu zmniejszamy CX:DX
        jmp short _graj_pauza

_graj_juz:

        in   al,klawiatura+1
        and  al,~3              ; zerujemy bity: 0 i 1 - wyłączamy głośnik
                                ; i bramkę
            AND AL, not 3       ; w FASMie:
        out  klawiatura+1,al

        pop si
        pop dx
        pop ax
        popf
        clc

        retf


_graj_blad:
        pop si
        pop dx
        pop ax
```

```
        popf
        stc

        retf
```

Jest to moja procedura wytwarzająca dźwięk w głośniczku (patrz mój inny artykuł). Trochę tego jest, co? Ale jest tu dużo spraw, które można omówić. Zacznijmy więc po kolei:

1. `public... / global...`
   Funkcje, które mają być widoczne na zewnątrz tego pliku, a więc możliwe do użycia przez inne programy, muszą być zadeklarowane jako public (TASM/FASM) (w NASMie: global). To jest na wszelki wypadek. Niektóre kompilatory domyślnie traktują wszystkie symbole jako publiczne, inne nie. Jeśli w programie będziemy chcieli korzystać z takiej funkcji, należy ją zadeklarować jako extrn (TASM/FASM) lub extern (NASM).

2. Deklaracja segmentu
   Żaden przyzwoity kompilator nie pozwoli na pisanie kodu poza jakimkolwiek segmentem (no chyba, że domyślnie zakłada segment kodu, jak NASM). Normalnie, w zwykłych programach, np typu .com, rolę tę pełni dyrektywa .code.

3. `assume`
   Mówimy kompilatorowi, że rejestr CS będzie wskazywał na ten segment

4. Gwiazdki lub inne elementy oddzielające (tu usunięte)
   Mogą się wydawać śmieszne lub niepotrzebne, ale gdy liczba procedur w pliku zaczyna sięgać 10-20, to NAPRAWDĘ zwiększają czytelność kodu, oddzielając procedury, dane itd.

5. Deklaracja procedury (wcześniej zadeklarowanej jako publiczna)
   Znak podkreślenia z przodu jest tylko po to, by w razie czego nie był identyczny z jakąś etykietą w programie korzystającym z biblioteki. Deklaracja jest typu far, żeby zmienić CS na bieżący segment i uniknąć kłopotów z 64kB limitem długości skoku (konkretnie to są to +/- 32kB).

6. To, czego procedura oczekuje i to, co zwraca.
   Jedną procedurę łatwo zapamiętać. Ale co zrobić, gdy jest ich już 100? Analizować kod każdej, aby sprawdzić, co robi, bo akurat szukamy takiej jednej....? No przecież nie.

7. Dobrą techniką programowania jest deklaracja stałych w stylu equ (lub #define w C). Zamiast nic nie znaczącej liczby można użyć wiele znaczącego zwrotu, co przyda się dalej w kodzie. I nie kosztuje to ani jednego bajtu. Oczywiście, ukrywa to część kodu (tutaj: numery portów), ale w razie potrzeby zmienia się tą wielkość tylko w 1 miejscu, a nie w 20.

8. `push...`
   Poza wartościami zwracanymi nic nie może być zmienione! Nieprzyjemnym uczuciem byłoby spędzenie kilku godzin przy odpluskwianiu (debugowaniu) programu tylko dlatego, że ktoś zapomniał zachować jakiegoś rejestru, prawda?

9. Sprawdzanie warunków wejścia, czy są prawidłowe. Zawsze należy wszystko przewidzieć.

10. Kod procedury. Z punktu widzenia tego artykułu jego treść jest dla nas nieistotna.

11. Punkt(y) wyjścia

Procedura może mieć dowolnie wiele punktów wyjścia. Tutaj zastosowano dwa, dla dwóch różnych sytuacji:

1. parametr był dobry, procedura zakończyła się bez błędów
2. parametr był zły, zwróć informację o błędzie

12. Koniec procedury, segmentu i pliku źródłowego. Słowo end nie zawsze jest konieczne, ale nie zaszkodzi. Wskazuje, gdzie należy skończyć przetwarzanie pliku.

Mamy więc już plik źródłowy. Co z nim zrobić? Skompilować, oczywiście!

```
tasm naszplik.asm /z /m
```

(/z - wypisz linię, w której wystąpił błąd
/m - pozwól na wielokrotne przejścia przez plik)
lub, dla NASMa:

```
nasm -f obj -o naszplik.obj naszplik.asm
```

(-f - określ format pliku wyjściowego
-o - określ nazwę pliku wyjściowego)
lub, dla FASMa:

```
fasm naszplik.asm naszplik.obj
```

Mamy już plik naszplik.obj. W pewnym sensie on już jest biblioteką! I można go używać w innych programach, np. w pliku program2.asm mamy:

```
...
extrn _graj_dzwiek:far          ; NASM: extern _graj_dzwiek
                                ; FASM: extrn _graj_dzwiek

        ...
        ...
        mov bx,440
        mov cx,0fh
        mov dx,4240h
        call far ptr _graj_dzwiek ; NASM: call far _graj_dzwiek
                                  ; FASM: call _graj_dzwiek
        ...
```

I możemy teraz zrobić:

```
tasm program2.asm /z /m
tlink program2.obj naszplik.obj
```

lub, dla NASMa:

```
nasm -f obj -o program2.obj program2.asm
alink program2.obj naszplik.obj -c- -oEXE -m-
```

lub, dla FASMa:

```
fasm program2.asm program2.obj
alink program2.obj naszplik.obj -c- -oEXE -m-
```

a linker zajmie się wszystkim za nas - utworzy plik program2.exe, zawierający także naszplik.obj. Jaka z tego korzyść? Plik program2.asm może będzie zmieniany w przyszłości wiele razy, ale naszplik.asm/.obj będzie ciągle taki sam. A w razie chęci zmiany procedury _graj_dzwiek wystarczy ją zmienić w 1 pliku i tylko jego ponownie skompilować, bez potrzeby wprowadzania tej samej zmiany w kilkunastu innych programach. Te programy wystarczy tylko ponownie skompilować z nową biblioteką, bez jakichkolwiek zmian kodu. No dobra, ale co z plikami .lib?
Otóż są one odpowiednio połączonymi plikami .obj. I wszystko działa tak samo.

No ale jak to zrobić?
Służą do tego specjalne programy, nazywane librarian (bibliotekarz). W pakiecie TASMa znajduje się program tlib.exe. Jego właśnie użyjemy (działa jak LLIB i wszystko robimy tak samo). Pliki .obj, które chcemy połączyć w bibliotekę można podawać na linii poleceń, ale jest to męczące, nawet jeśli napisze się plik wsadowy tlib.bat uruchamiający tlib. My skorzystamy z innego rozwiązania.
Programowi można na linii poleceń podać, aby komendy czytał z jakiegoś pliku. I to właśnie zrobimy.
Piszemy plik tlib.bat:

```
tlib.exe naszabibl.lib @lib.txt
```

i plik lib.txt (zwykłym notatnikiem):

```
+- ..\obj\pisz.obj &
+- ..\obj\wej.obj &
+- ..\obj\procesor.obj &
+- ..\obj\losowe.obj &
+- ..\obj\f_pisz.obj &

+- ..\obj\dzwiek.obj &
+- ..\obj\f_wej.obj &
+- ..\obj\fn_pisz.obj &
+- ..\obj\fn_wej.obj
```

(użyłem tutaj nazw modułów, które składają się na moją bibliotekę).
+- oznacza zamień w pliku dany moduł
& oznacza sprawdzaj jeszcze w kolejnej linijce
Przy pierwszym tworzeniu można użyć + zamiast +-, aby uniknąć ostrzeżeń o uprzedniej nieobecności danego modułu w bibliotece.
Teraz uruchamiamy już tylko tlib.bat a w razie potrzeby zmieniamy tylko lib.txt.

Gdzie zdobyć narzędzia:

1. NASM
2. Alink
3. Lib (LLIB, a nie ten z pakietu Borlanda czy Microsoft-u):
   www.dunfield.com/downloads.htm (szukaj SKLIB31.ZIP)
   www2.inf.fh-rhein-sieg.de/~skaise2a/ska/sources.html
   Jeśli tam go nie ma, to poszukajcie na stronach FreeDOS-a

Kopia mojej biblioteki powinna znajdować się na stronach, gdzie znaleźliście ten kurs.

Miłej zabawy.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Wyświetlanie obrazków BMP

Jeśli przejrzeliście mój poprzedni kurs związany z grafiką, to umiecie już coś samodzielnie narysować. Ale przecież w Internecie (i nie tylko) jest tyle ciekawych rysunków, nie mówiąc już o tych, które moglibyście stworzyć dla jakiegoś specjalnego celu, np. swojej własnej gry. Dlatego teraz pokażę, jak takie rysunki wyświetlać. Ze względu na prostotę formatu, wybrałem pliki typu BMP. Plik, który chcecie wyświetlić powinien mieć rozmiar 320x200 pikseli w 256 kolorach (jak pamiętamy, taki rysunek pasuje jak ulał do trybu graficznego 13h).
Wszystkie operacje na pliku zostały już przez mnie szczegółowo opisane w jednej z części mojego kursu, więc tutaj nie będziemy poświęcać im zbyt wiele uwagi.
Ale przejdźmy wreszcie do interesujących nas szczegółów.

Powinniście zaopatrzyć się w cokolwiek, co opisuje format BMP. Informacje, z których będę tutaj korzystał, znalazłem w Internecie (niestety, nie pamiętam już gdzie, ale możecie poszukać na Wotsit.org).
A oto nagłówek pliku BMP (składnia języka Pascal niestety, info: Piotr Sokolowski, 6 maja 1998r):
(przeskocz opis nagłówka)

```
Type
 TBitMapHeader =
  Record
        bfType :             Word; (dwa bajty)
        bfSize :             LongInt; (cztery bajty)
        bfReserved :         LongInt;
        bfOffBits :          LongInt;
        biSize :             LongInt;
        biWidth :            LongInt;
        biHeight :           LongInt;
        biPlanes :           Word;
        biBitCount :         Word;
        biCompression :      LongInt;
        biSizeImage :        LongInt;
        biXPelsPerMeter :    LongInt;
        biYPelsPerMeter :    LongInt;
        biClrUsed :          LongInt;
        biClrImportant :     LongInt;
  End;
```

Gdzie:

- bftype - jest to dwubajtowa sygnatura BM
- bfsize - czterobajtowy rozmiar pliku
- bfreserved - pole zarezerwowane (0)
- bfoffbits - przesunięcie (adres) początku danych graficznych
- bisize - podaje rozmiar nagłówka
- biwidth - wysokość bitmapy w pikselach
- biheight - szerokość bitmapy w pikselach
- biplanes - liczba planów (prawie zawsze ma wartość 1)
- bibitcound - ilość bitów na piksel. Przyjmuje wartość 1,4,8 lub 24.
- bicompression - sposób kompresji
- bisizeimag - rozmiar obrazka w bajtach. W przypadku bitmapy nieskompresowanej równe 0.
- biXpelspermeter, biYpelspermeter - ilość pikseli na metr
- biclrused - ilość kolorów istniejącej palety, a używanych właśnie przez bitmapę
- biclrimporant - określa, który kolor bitmapy jest najważniejszy, gdy równy 0 to wszystkie są tak samo istotne.

Ale spokojnie - nie musicie znać tych wszystkich pól, bo my nie będziemy wszystkich używać. Ściśle mówiąc, nie będziemy używać ani jednego z tych pól!
No to po co to wszystko?
Po to, aby znać długość nagłówka pliku (54 bajty), który ominiemy przy analizie pliku.

Po nagłówku idzie paleta 256 kolorów * 4 bajty/kolor = kolejny 1kB. Jeśli macie jakieś wątpliwości co do tego 1 kilobajta, to słusznie. Oczywiście, do opisu koloru wystarczą 3 bajty (odpowiadające kolorom czerwonemu, zielonemu i niebieskiemu - RGB), co daje razem 768 bajtów. Co czwarty bajt nie zawiera żadnej istotnej informacji i będziemy go pomijać (zmienna z).

Zaraz po palecie idzie obraz, piksel po pikselu. Niestety, nie jest to tak logiczne ustawienie, jak byśmy sobie tego życzyli. Otóż, pierwsze 320 bajtów to ostatni wiersz obrazka, drugie 320 - przedostatni, itd. Dlatego trzeba będzie troszkę pokombinować.
Zanim jeszcze zaczniemy, należy się przyjrzeć, których portów (choć to samo można uzyskać wywołując odpowiednie przerwanie) i dlaczego będziemy używać (patrzymy do pliku ports.lst w Spisie Przerwań Ralfa Brown'a):
(przeskocz opis portów)

```
        03C8  RW  (VGA,MCGA) PEL address register (write mode)
                  Sets DAC in write mode and assign start of color register
                  index (0..255) for following write accesses to 3C9h.
                  Don't read from 3C9h while in write mode. Next access to
                  03C8h will stop pending mode immediatly.
        03C9  RW  (VGA,MCGA) PEL data register
                  Three consequtive reads (in read mode) or writes (in write
                  mode) in the order: red, green, blue. The internal DAC index
                  is incremented each 3rd access.
                   bit7-6: HiColor VGA DACs only: color-value bit7-6
                   bit5-0:                        color-value bit5-0
```

Czyli najpierw na 3C8h idzie numer rejestru dla danego koloru (rejestrów jest 256 i kolorów też), a potem na 3C9h idą trzy wartości kolorów: czerwonego, zielonego i niebieskiego, których połączenie daje nam żądany kolor.

Ale dobierzmy się wreszcie do kodu:
(przeskocz program)

```
; Program wyświetla na ekranie kolorowy rysunek o rozmiarze
; 320x200 w 256 kolorach, umieszczony w pliku.
;
; nasm -O999 -o bmp1.com -f bin bmp1.asm
;
; Autor: Bogdan D., bogdandr (at) op (kropka) pl
;
; na podstawie kodu podpisanego "Piotr Sokolowski",
; napisanego w języku Pascal

org 100h

start:
        mov     ax, 13h
        int     10h     ; uruchamiamy tryb graficzny 13h - 320x200x256

        mov     ax, 3d00h               ; otwieramy plik tylko do odczytu
        mov     dx, nazwa_pliku
        int     21h
```

```
        jnc     otw_ok

        mov     ah, 9
        mov     dx, blad_plik           ; wyświetlane, gdy wystąpił błąd
        int     21h

err:
        mov     ax, 4c01h               ; wyjście z kodem błędu=1
        int     21h

otw_ok:
        mov     bx, ax                  ; bx = uchwyt do pliku
        mov     ah, 3fh                 ; czytanie z pliku
        mov     cx, 54                  ; wyrzucamy 54 bajty nagłówka
        mov     dx, smieci
        int     21h
        jc      err

; wczytywanie palety z pliku:


        xor     si, si                  ; indeks do tablicy "paleta"

czytaj_pal:
        mov     ah, 3fh                 ; czytanie z pliku
        mov     cx, 4                   ; czytam po 4 bajty - do b,g,r i z.
                                        ; ("z" nas nie interesuje)
        mov     dx, b
        int     21h
        jc      err

                                        ; ustawiamy paletę:
        mov     al, [r]
        shr     al, 2
        mov     [paleta+si], al         ; paleta[si] = [r] / 4

        mov     al, [g]
        shr     al, 2
        mov     [paleta+si+1], al       ; paleta[si] = [g] / 4

        mov     al, [b]
        shr     al, 2
        mov     [paleta+si+2], al       ; paleta[si] = [b] / 4

        add     si, 3                   ; przejdź o 3 miejsca dalej -
                                        ; na kolejne wartości RGB

        cmp     si, 256*3               ; sprawdź, czy nie zapisaliśmy
                                        ; już wszystkich kolorów
        jb      czytaj_pal

; wysyłanie palety do karty graficznej:

        xor     ax, ax
        xor     si, si                  ; SI = indeks do palety

        mov     dx, 3c8h                ; port karty graficznej
wyslij_palete:
        out     dx, al                  ; wysyłamy numer rejestru,
                                        ; wszystkie od 0 do 255
```

```
        inc     dx                      ; DX = port 3C9h

        push    ax
                                        ; zapisujemy kolorki:
                                        ; czerwony, zielony, niebieski.

        mov     al, [paleta+si]         ; AL = czerwony
                czytaj_pal)             ; (patrz: pętla
        out     dx, al

        mov     al, [paleta+si+1]       ; AL = zielony
        out     dx, al

        mov     al, [paleta+si+2]       ; AL = niebieski
        out     dx, al

        pop     ax

        add     si, 3                   ; przejdź do następnych kolorów

        dec     dx                      ; DX z powrotem 3C8h

        inc     ax                      ; wybierzemy kolejny rejestr koloru
                                        ; w karcie graficznej

        cmp     ax, 256                 ; sprawdź, czy już koniec pracy
        jb      wyslij_palete

; wczytywanie obrazka:

        mov     ax, 0a000h
        mov     ds, ax          ; czytaj bezpośrednio do pamięci ekranu

        mov     dx, 64000-320           ; DX = adres ostatniego wiersza

        mov     cx, 320                 ; z pliku czytamy po 320 bajtów

obrazek:
        mov     ah, 3fh
        int     21h                     ; czytaj 320 bajtów prosto na ekran
        jc      err

        sub     dx, 320                 ; przejdź do wcześniejszego wiersza
        jnc     obrazek                 ; dopóki nie musimy pożyczać
                                        ; do odejmowania. Pożyczymy dopiero
                                        ; wtedy, gdy DX < 320 - a to się
                                        ; zdarzy dopiero, gdy DX = 0, czyli
                                        ; przerobiliśmy cały obrazek i ekran.
                                        ; Wtedy kończymy pracę.


; koniec programu:

        mov     ah, 3eh
        int     21h                     ; zamknij plik
        jc      err

        xor     ah, ah
        int     16h                     ; czekamy na klawisz

        mov     ax, 3
        int     10h                     ; powrót do trybu tekstowego
```

```
        mov     ax, 4c00h
        int     21h

; dane:


nazwa_pliku     db      "rys1.bmp",0
blad_plik       db      "Blad operacji na pliku!$"

smieci:         times   54      db 0
paleta:         times   768     db 0
b               db 0
g               db 0
r               db 0
z               db 0
kolor           db 0
```

Mam nadzieję, że kod jest dość jasny. Nawet jeśli znacie asemblera tylko w takim stopniu, w jakim to jest możliwe po przeczytaniu mojego kursu, zrozumienie tego programu nie powinno sprawić Wam więcej kłopotów niż mnie sprawiło przetłumaczenie go z Pascala.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Pisanie boot-sektorów

Gdy już choć średnio znacie asemblera, to po pewnym czasie pojawiają się pytania (mogą one być spowodowane tym, co usłyszeliście lub Waszą własną ciekawością):

1. Co się dzieje, gdy ma zostać uruchomiony jest system operacyjny?
2. Skąd BIOS ma wiedzieć, którą część systemu uruchomić?
3. Jak BIOS odróżnia systemy operacyjne, aby móc je uruchomić?

Odpowiedź na pytanie 2 brzmi: nie wie. Odpowiedź na pytanie 3 brzmi: wcale. Wszystkie Wasze wątpliwości rozwieje odpowiedź na pytanie 1.

Gdy zakończył się POST (Power-On Self Test), wykrywanie dysków i innych urządzeń, BIOS przystępuje do czytania pierwszych sektorów tych urządzeń, na których ma być szukany system operacyjny (u mnie jest ustawiona kolejność: CD-ROM, stacja dyskietek, dysk twardy).
Gdy znajdzie sektor odpowiednio zaznaczony: bajt nr 510 = 55h i bajt 511 = AAh (pamiętajmy, że 1 sektor ma 512 bajtów, a liczymy od zera), to wczytuje go pod adres bezwzględny 07C00h i uruchamia kod w nim zawarty (po prostu wykonuje skok pod ten adres). Nie należy jednak polegać na tym, że segment kodu CS = 0, a adres instrukcji IP=7C00h (choć najczęściej tak jest).
To właśnie boot-sektor jest odpowiedzialny za ładowanie odpowiednich części właściwego systemu operacyjnego. Na komputerach z wieloma systemami operacyjnymi sprawa też nie jest tak bardzo skomplikowana. Pierwszy sektor dysku twardego, zwany Master Boot Record (MBR), zawiera program ładujący (Boot Manager, jak LILO czy GRUB), który z kolei uruchamia boot-sektor wybranego systemu operacyjnego.

My oczywiście nie będziemy operować na dyskach twardych, gdyż byłoby to niebezpieczne. Z dyskietkami zaś można eksperymentować do woli...
A instrukcja jest prosta: umieszczamy nasz programik w pierwszym sektorze dyskietki, zaznaczamy go odpowiednimi ostatnimi bajtami i tyle. No właśnie... niby proste, ale jak o tym pomyśleć to ani to pierwsze, ani to drugie nie jest sprawą banalną.

Do zapisania naszego bootsektorka na dyskietkę możemy oczywiście użyć gotowców - programów typu rawwrite itp. Ma to pewne zalety - program był już używany przez dużą liczbę osób, jest sprawdzony i działa. Ale coś by było nie tak, gdybym w kursie programowania w asemblerze kazał Wam używać cudzych programów. Do napisania swojego własnego programu zapisującego dany plik w pierwszym sektorze dyskietki w zupełności wystarczy Wam wiedza uzyskana po przeczytaniu części mojego kursu poświęconej operacjom na plikach wraz z tą krótką informacją ze Spisu Przerwań Ralfa Brown'a:
(przeskocz opis int 13h, ah=3)

```
INT 13 - DISK - WRITE DISK SECTOR(S)
        AH = 03h
        AL = number of sectors to write (must be nonzero)
        CH = low eight bits of cylinder number
        CL = sector number 1-63 (bits 0-5)
             high two bits of cylinder (bits 6-7, hard disk only)
        DH = head number
        DL = drive number (bit 7 set for hard disk)
        ES:BX -> data buffer
Return: CF set on error
        CF clear if successful
```

Jak widać, sprawa już staje się prosta. Oczywiście, AL=1 (bo zapisujemy 1 sektor), DX=0 (bo stacja ma 2 głowice, a pierwsza ma numer 0, zaś numer dysku 0 wskazuje stację A:), CX=1 (bo numery sektorów zaczynają się od 1, a zapisujemy w pierwszym cylindrze, który ma numer 0).
Schemat działania jest taki:

- Otwórz plik zawierający skompilowany bootsektor
- Przeczytaj z niego 512 bajtów (do zadeklarowanej tablicy w pamięci)
- Zamknij plik
- Zapisz odczytane dane na dyskietce, korzystając z int 13h

Sprawa jest tak prosta, że tym razem nie podam gotowca. Gdy już mamy program zapisujący bootsektor na dyskietkę, trzeba się postarać o to, aby nasz programik (który ma stać się tym bootsektorem) miał dokładnie 512 bajtów i aby 2 ostatnie jego bajty to 55h, AAh.
Oczywiście, nie będziemy ręcznie dokładać tylu bajtów, ile trzeba, aby dopełnić nasz program do tych 512. Zrobi to za nas kompilator. Wystarczy po całym kodzie i wszystkich danych, na szarym końcu, umieścić takie coś (NASM/FASM):
(przeskocz tworzenie sygnatury)

```
        times 510 - ($ - start) db 0
        dw 0aa55h
```

Dla TASMa powinno to wyglądać mniej-więcej tak:

```
        db 510 - ($ - offset start) dup (0)
        dw 0aa55h
end start
```

To wyrażenie mówi tyle: od bieżącej pozycji w kodzie odejmij pozycję początku kodu (tym samym obliczając długość całego kodu), otrzymaną liczbę odejmij od 510 - i dołóż tyle właśnie bajtów zerowych. Gdy już mamy program długości 510 bajtów, to dokładamy jeszcze znacznik i wszystko jest dobrze.

Jest jednak jeszcze jedna sprawa, o której nie wspomniałem - ustawienie DS i wartości org dla naszego kodu. Otóż, jeśli stwierdzimy, że nasz kod powinien zaczynać się od offsetu 0 w naszym segmencie, to ustawmy sobie org 0 i DS=07C0h (tak, ilość zer się zgadza), ale możemy też mieć org 7C00h i DS=0. Żadne z tych nie wpływa w żaden sposób na długość otrzymanego programu, a należy o to zadbać, gdyż nie mamy gwarancji, że DS będzie pokazywał na nasze dane po uruchomieniu bootsektora.

Teraz, uzbrojeni w niezbędną wiedzę, zasiadamy do pisania kodu naszego bootsektora. Nie musi to być coś wielkiego - tutaj pokażę coś, co w lewym górnym rogu ekranu pokaże cyfrę jeden (o bezpośredniej manipulacji ekranem możecie przeczytać w moim innym artykule) i po naciśnięciu dowolnego klawisza zresetuje komputer (na jeden ze sposobów podanych w jeszcze innym artykule...).
Oto nasz kod (NASM):
(przeskocz przykładowy bootsektor)

```
; nasm -o boot.bin -f bin boot.asm

org 7c00h                       ; lub     "org 0"

start:
        mov     ax, 0b800h
        mov     es, ax          ; ES = segment pamięci ekranu
```

```
        mov     byte [es:0], "1" ; piszemy "1"

        xor     ah, ah
        int     16h             ; czekamy na klawisz

        mov     bx, 40h
        mov     ds, bx
        mov     word [ds:72h], 1234h    ; 40h:72h = 1234h -
                                        ; wybieramy gorący reset

        jmp     0ffffh:0000h            ; reset

times 510 - ($ - start) db 0            ; dopełnienie do 510 bajtów
dw 0aa55h                               ; znacznik
```

Nie było to długie ani trudne, prawda? Rzecz jasna, nie można w bootsektorach używać żadnych przerwań systemowych, np. DOS-owego int 21h, bo żaden system po prostu nie jest uruchomiony i załadowany. Tak napisany programik kompilujemy do formatu binarnego. W TASM-ie kompilacja wyglądałaby jakoś tak (po dodaniu w programie dyrektyw .model tiny, .code, .8086 i end start):

```
tasm bootsec1.asm
tlink bootsec1.obj,bootsec1.bin /t
```

Po kompilacji umieszczamy go na dyskietce przy użyciu programu napisanego już przez nas wcześniej. Resetujemy komputer (i upewniamy się, że BIOS spróbuje uruchomić system z dyskietki), wkładamy dyskietkę i.... cieszymy się swoim dziełem (co prawda ta jedynka będzie mało widoczna, ale rzeczywiście znajduje się na ekranie).

Zauważcie też, że ani DOS ani Windows nie rozpoznaje już naszej dyskietki, mimo iż przedtem była sformatowana. Dzieje się tak dlatego, że w bootsektorze umieszczane są informacje o dysku. Bootsektor typu FAT12 (DOSowy/Windowsowy) powinien się zaczynać mniej-więcej tak: (przeskocz systemowy obszar bootsektora)

```
org 7c00h                       ; lub org 0, oczywiście

start:
        jmp short kod
        nop

        db "        "   ; nazwa OS i wersja OEM (8B)
        dw 512          ; bajtów/sektor (2B)
        db 1            ; sektory/jednostkę alokacji (1B)
        dw 1            ; zarezerwowane sektory (2B)
        db 2            ; liczba tablic alokacji (1B)
        dw 224          ; liczba pozycji w katalogu głównym (2B)
                        ; 224 to typowa wartość
        dw 2880         ; liczba sektorów (2B)
        db 0f0h         ; Media Descriptor Byte (1B)
        dw 9            ; sektory/FAT (2B)
        dw 18           ; sektory/ścieżkę (2B)
        dw 2            ; liczba głowic (2B)
        dd 0            ; liczba ukrytych sektorów (4B)
        dd 0            ; liczba sektorów (część 2),
                        ; jeśli wcześniej było 0 (4B)
        db 0            ; numer dysku (1B)
        db 0            ; zarezerwowane (1B)
        db 0            ; rozszerzona sygnatura bloku ładującego
        dd 0bbbbddddh   ; numer seryjny dysku (4B)
```

```
        db "           "; etykieta (11B)
        db "FAT 12  "   ; typ FAT (8B), zwykle  "FAT 12  "

kod:
        ; tutaj dopiero kod bootsektora
```

Ta porcja danych oczywiście uszczupla ilość kodu, którą można umieścić w bootsektorze. Nie jest to jednak duży problem, gdyż i tak jedyną rolą większości bootsektorów jest uruchomienie innych programów (second stage bootloaders), które dopiero zajmują się ładowaniem właściwego systemu.

Jeszcze ciekawostka: co wypisuje BIOS, gdy dysk jest niewłaściwy (bez systemu)?
Otóż - nic! BIOS bardzo chętnie przeszedłby do kolejnego urządzenia.
Dlaczego więc tego nie robi i skąd ten napis o niewłaściwym dysku systemowym??
Odpowiedź jest prosta - sformatowana dyskietka posiada bootsektor!
Dla BIOSu jest wszystko OK, uruchamia więc ten bootsektor. Dopiero ten wypisuje informację o niewłaściwym dysku, czeka na naciśnięcie klawisza, po czym uruchamia int 19h. O tym, co robi przerwanie 19h możecie przeczytać w artykule o resetowaniu.

Miłego bootowania systemu!
P.S. Jeśli nie chcecie przy najdrobniejszej zmianie kodu resetować komputera, możecie poszukać w Internecie programów, które symulują procesor (w tym fazę ładowania systemu). Jednym z takich programów jest Bochs.

Co dalej?

O ile bootsektor jest ograniczony do 512 bajtów, to może w dość łatwy sposób posłużyć do wczytania do pamięci o wiele większych programów. Wystarczy użyć funkcji czytania sektorów:
(przeskocz opis int 13h, ah=2)

```
INT 13 - DISK - READ SECTOR(S) INTO MEMORY
        AH = 02h
        AL = number of sectors to read (must be nonzero)
        CH = low eight bits of cylinder number
        CL = sector number 1-63 (bits 0-5)
             high two bits of cylinder (bits 6-7, hard disk only)
        DH = head number
        DL = drive number (bit 7 set for hard disk)
        ES:BX -> data buffer
Return: CF set on error
        CF clear if successful
```

Jak widać, poza wartością rejestru AH, jej parametry nie różnią się od parametrów funkcji zapisu sektorów. Wystarczy więc wybrać nieużywany segment pamięci, np. ES=8000h i począwszy od offsetu BX=0, czytać sektory zawierające nasz kod, zwiększając BX o 512 za każdym razem. Kod do załadowania nie musi być oczywiście w postaci pliku na dyskietce, to by tylko utrudniło pracę (gdyż trzeba wtedy czytać tablicę plików FAT). Najłatwiej załadować kod tym samym sposobem, co bootsektor, ale oczywiście do innych sektorów. Polecam zacząć od sektora dziesiątego lub wyżej, gdyż zapisanie tam danych nie zamaże tablicy FAT i przy próbie odczytu zawartości dyskietki przez system nie pojawią się żadne dziwne obiekty.

Po załadowaniu całego potrzebnego kodu do pamięci przez bootsektor, wystarczy wykonać skok:

```
jmp     8000h:0000h
```

Wtedy kontrolę przejmuje kod wczytany z dyskietki.

Ale jest jeden kruczek - trzeba wiedzieć, jakie numery cylindra, głowicy i sektora podać do funkcji czytające sektory, żeby rzeczywiście odczytała te właściwe.
Struktura standardowej dyskietki jest następująca: 512 bajtów na sektor, 18 sektorów na ścieżkę, 2 ścieżki na cylinder (bo są dwie strony dyskietki, co daje 36 sektorów na cylinder), 80 cylindrów na głowicę. Razem 2880 sektorów po 512 bajtów, czyli 1.474.560 bajtów.
Mając numer sektora (bo wiemy, pod jakimi sektorami zapisaliśmy swój kod na dyskietce), odejmujemy od niego 1 (tak by zawsze wszystkie numery sektorów zaczynały się od zera), po czym dzielimy go przez 36. Uzyskany iloraz to numer cylindra (rejestr CH), reszta zaś oznacza numer sektora w tymże cylindrze (rejestr CL). Jeśli ta reszta jest większa bądź równa 18, należy wybrać głowicę numer 1 (rejestr DH), zaś od numeru sektora (rejestr CL) odjąć 18. W przeciwnym przypadku należy wybrać głowicę numer 0 i nie robić nic z numerem sektora.
W ten sposób otrzymujemy wszystkie niezbędne dane i możemy bez przeszkód w pętli czytać kolejne sektory zawierające nasz kod.
Całą tę procedurę ilustruje ten przykładowy kod:
(przeskocz procedurę czytania sektorów)

```
secrd:
;wejście: ax=sektor, es:bx -> dane

        dec ax          ; z numerów 1-36 na 0-35
        mov cl,36       ; liczba sektorów na cylinder = 36
        xor dx,dx       ; zakładamy na początek: głowica 0, dysk 0 (a:)
        div cl          ; AX (numer sektora) dzielimy przez 36
        mov ch,al       ; AL=cylinder, AH=przesunięcie względem
                        ;       początku cylindra, czyli sektor
        cmp ah,18       ; czy numer sektora mniejszy od 18?
        jb .sec_ok      ; jeśli tak, to nie robimy nic
        sub ah,18       ; jeśli nie, to odejmujemy 18
        inc dh          ; i zmieniamy głowicę
.sec_ok:
        mov cl, ah      ; CL = numer sektora
        mov ax,0201h    ; odczytaj 1 sektor
        inc cl          ; zwiększ z powrotem z zakresu 0-17 do 1-18

        push dx         ; niektóre biosy niszczą DX, nie ustawiają
                        ;       flagi CF, lub zerują flagę IF
        stc
        int 13h         ; wykonaj czytanie
        sti
        pop dx
```

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Rozpoznawanie typu procesora

(przeskocz wykrywanie procesora)

Jak zapewne wiecie, wiele programów (systemy operacyjne, gry, ...) potrafi jakoś dowiedzieć się, na jakim procesorze zostały uruchomione. Rozpoznanie typu procesora umożliwia np. uruchomienie dodatkowych optymalizacji w programie lub odmowę dalszego działania, jeśli program musi korzystać z instrukcji niedostępnych na danym procesorze.
Wykrywanie rodzaju CPU i FPU nie jest trudne i pokażę teraz, jak po kolei sprawdzać typ procesora (nie można przecież zacząć sprawdzania od najwyższych).
Informacje, które tutaj podam, są oczywiście słuszne dla wszystkich procesorów rodziny x86 (AMD, Cyrix, ...), a nie tylko Intela.
Generalnie sposoby wykrywania są dwa: poprzez rejestr FLAG lub poprzez zakodowanie w kodzie instrukcji, które wykonają się tylko na danym modelu (i późniejszych). Drugi sposób jest trochę trudniejszy: należy przejąć przerwanie INT6 (nieprawidłowa instrukcja) i sprawdzać, kiedy zostało wywołane.

1. odróżnienie 8088 od reszty
   (przeskocz 8088)
   Procesor 8088 od pozostałych odróżnia to, że zmniejsza on rejestr SP przed umieszczeniem go na stosie. Reszta robi to po umieszczeniu SP na stosie. Kod wyglądałby więc na przykład tak:
   (przeskocz kod dla 8088)

```
mov     ax, sp
push    sp
pop     cx
xor     ax, cx          ; lub cmp ax, cx
jz      nie_8088
```

2. 8086
   (przeskocz 8086)
   Na tym procesorze w rejestrze flag bity 12-15 zawsze mają wartość 1.
   (przeskocz kod dla 8086)

```
pushf                   ; flagi na stos
pop     ax              ; AX = flagi
and     ax, 0fffh       ; czyścimy bity 12-15
push    ax              ; AX na stos
popf                    ; flagi = AX
pushf                   ; z powrotem na stos
pop     ax              ; AX = flagi
and     ax, 0f000h      ; zerujemy bity poza bitami 12-15
cmp     ax, 0f000h      ; jeśli ustawione, to 8086
jz      jest_8086
```

3. 80186
   (przeskocz 80186)
   Test polega na próbie wykonania instrukcji smsw dx, nieprawidłowej na procesorach wcześniejszych niż 80286. Przerwanie nieprawidłowej instrukcji przejmujemy tak:
   (przeskocz kod dla 80186)

```
xor     ax, ax
mov     es, ax

les     bx, [es:6 << 2] ; FASM: les bx, [es:(6 shl 2)]

mov     [_stare06+2], es
mov     [_stare06], bx
```

```
        mov     es, ax
        mov     word [es:(6 << 2)], moje06
                ; FASM: mov word [es:(6 shl 2)], moje06
        mov     word [es:(6 << 2) + 2], seg moje06
                ; FASM: mov word [es:(6 shl 2)], seg moje06
```

Sama procedura obsługi przerwania wyglądać będzie tak:

```
moje06:
        pop     ax
        add     ax, 3
        push    ax
        xor     ax, ax
        iret
```

Proste: zwiększamy adres powrotny o 3 (długość instrukcji smsw dx) i zerujemy AX (potem w kodzie sprawdzimy jego wartość). Sam kod sprawdzający wygląda tak:

```
        mov     ax, 1
        db      0fh, 1, 0e2h            ; smsw dx
        or      ax, ax
        jz      jest_286
```

Przywrócenie oryginalnej procedury wygląda tak:

```
        xor     ax, ax
        les     cx, [_stare06]
        mov     ds, ax
        mov     [ds:(6 << 2)], cx
                ; FASM: mov [ds:(6 shl 2)], cx
        mov     [ds:(6 << 2) + 2], es
                ; FASM: mov [ds:(6 shl 2) + 2], es
```

4. 80286
   (przeskocz 80286)
   Na tym procesorze bity 12-15 flag zawsze mają wartość 0. Przykładowy kod wygląda więc tak:
   (przeskocz kod dla 80286)

```
        pushf                           ; flagi na stos
        pop     ax                      ; AX = flagi
        or      ax, 0f000h              ; ustawiamy bity 12-15
        push    ax                      ; AX na stos
        popf                            ; flagi = AX
        pushf                           ; flagi na stos
        pop     ax                      ; AX = flagi
        and     ax, 0f000h      ; jeśli wyczyszczone, to 286
        jnz     nie_286
```

5. 80386
   (przeskocz 80386)
   Na tym procesorze nie można zmienić bitu numer 18 we flagach (wiemy, że rejestr flag ma 32 bity). Bit ten odpowiada za Alignment Check i spowoduje przerwanie m.in wtedy, gdy SP nie będzie podzielne przez 4. Dlatego, zanim będziemy testować ten bit, musimy zachować SP i wyzerować jego najmłodsze 2 bity.
   (przeskocz kod dla 80386)

```
        mov     dx, sp
```

```
and     sp, ~3          ; aby uniknąć AC fault.
                        ; FASM: and sp, not 3
pushfd                  ; flagi na stos
pop     eax             ; EAX = E-flagi
mov     ecx, eax        ; zachowanie EAX
xor     eax, 40000h     ; zmiana bitu 18
push    eax             ; EAX na stos
popfd                   ; E-flagi = EAX
pushfd                  ; flagi na stos
pop     eax             ; EAX = flagi
xor     eax, ecx  ; czy takie same? jeśli tak, to 386
mov     sp, dx          ; przywrócenie SP
jz      jest_386
```

6. 80486
   (przeskocz 80486)
   Na tym procesorze nie można zmienić bitu 21 we flagach. Jeśli ten bit można zmienić, to procesor obsługuje instrukcję CPUID, której będziemy używać do dalszego rozpoznania. Kod:
   (przeskocz kod dla 80486)

```
pushfd                  ; flagi na stos
pop     eax             ; EAX = E-flagi
mov     ecx, eax        ; zachowanie EAX
xor     eax, 200000h    ; zmiana bitu 21
push    eax             ; EAX na stos
popfd                   ; E-flagi = EAX
pushfd                  ; flagi na stos
pop     eax             ; EAX = flagi
xor     eax, ecx  ; czy takie same? jeśli tak, to 486
jz      jest_486
jmp     jest_586
```

Zanim omówię sposób korzystania z instrukcji CPUID, zajmijmy się sposobem rozpoznania typu koprocesora.

Koprocesor

(przeskocz wykrywanie koprocesora)

Tutaj możliwości są tylko 4: brak koprocesora, 8087, 80287, 80387. No to do roboty.

1. czy w ogóle jest jakiś koprocesor?
   (przeskocz test na istnienie FPU)
   To sprawdzamy bardzo łatwo. Jeśli nie ma koprocesora, to w chwili wykonania instrukcji FPU może wystąpić przerwanie INT6 (nieprawidłowa instrukcja), ale nie o tym sposobie chciałem powiedzieć. Koprocesor można wykryć, jeśli słowo stanu zostanie zapisane prawidłowo. Oto kod:
   (przeskocz test na istnienie FPU)

```
fninit                  ; inicjalizacja zeruje rejestry

; wpisujemy jakąś niezerowa wartość:
mov     word [_fpu_status], 5a5ah

; zapisz słowo statusowe do pamięci:
fnstsw  [_fpu_status]
mov     ax, [_fpu_status]
or      al, al  ; jeśli zapisało dobrze (zera oznaczają
```

```
                        ; puste rejestry), to jest FPU

                jz      jest_FPU
```

2. 8087

   (przeskocz 8087)

   Sztuczka polega na wykorzystaniu instrukcji FDISI (wyłączenie przerwań), która rzeczywiście coś robi tylko na 8087. Po wyłączeniu przerwań w słowie kontrolnym zostaje włączony bit numer 7.

   (przeskocz kod dla 8087)

```
                ; zachowaj słowo kontrolne do pamięci:
                fnstcw  [_fpu_status]

                ; wyłączamy wszystkie
                ; przerwania (poprzez słowo kontrolne):
                and     word [_fpu_status], 0ff7fh

                ; załaduj słowo kontrolne z pamięci:
                fldcw   [_fpu_status]

                fdisi           ; wyłączamy wszystkie przerwania
                                ; (jako instrukcja)

                ; zachowaj słowo kontrolne do pamięci:
                fstcw   [_fpu_status]
                test    byte [_fpu_status], 80h ; bit 7 ustawiony?

                jz      nie_8087        ; jeśli nie, to nie jest to 8087
```

3. 80287

   (przeskocz 80287)

   Koprocesor ten nie odróżnia minus nieskończoności od plus nieskończoności. Kod na sprawdzenie tego wygląda tak:

   (przeskocz kod dla 80287)

```
                finit

                fld1            ; st(0)=1
                fldz            ; st(0)=0,st(1)=1
                fdivp   st1     ; tworzymy nieskończoność,
                                ; dzieląc przez 0
                fld     st0     ; st(1):=st(0)=niesk.
                fchs            ; st(0)= -niesk.

                                ; porównanie st0 z st1 i
                                ; zdjęcie obu ze stosu
                fcompp          ; 8087/287: -niesk. = +niesk.,
                                ; 387: -niesk. != +niesk.

                fstsw   [_fpu_status]   ; zapisz status do pamięci
                mov     ax, [_fpu_status]       ; AX = status

                sahf            ; zapisz AH we flagach. tak sie składa,
                                ; że tutaj również flaga ZF wskazuje na
                                ; równość argumentów.
                jz      jest_287
                jmp     jest_387
```

Dalsze informacje o procesorze - instrukcja CPUID

Od procesorów 586 (choć niektóre 486 też podobno ją obsługiwały), Intel i inni wprowadzili instrukcję CPUID. Pozwala ona odczytać wiele różnych informacji o procesorze (konkretny typ, rozmiary pamięci podręcznych, dodatkowe rozszerzenia, ...).
Korzystanie z tej instrukcji jest bardzo proste: do EAX wpisujemy numer (0-3) i wywołujemy instrukcję, np.

```
mov     eax, 1
cpuid
```

Teraz omówię, co można dostać przy różnych wartościach EAX.

1. EAX=0
   (przeskocz EAX=0)
   EAX = maksymalny numer funkcji dla CPUID.
   EBX:EDX:ECX = marka procesora (12 znaków ASCII).
   Intel - GenuineIntel
   AMD - AuthenticAMD
   NexGen - NexGenDriven
   Cyrix, VIA - CyrixInstead
   RISE - RiseRiseRise,
   Centaur Technology/IDT - CentaurHauls (programowalne, może być inne)
   United Microelectronics Corporation - UMC UMC UMC
   Transmeta Corporation - GenuineTMx86
   SiS - SiS SiS SiS
   National Semiconductor - Geode by NSC.

2. EAX=1
   (przeskocz EAX=1)
   EAX = informacje o wersji:
   - bity 0-3: stepping ID
   - bity 4-7: model
   - bity 8-11: rodzina. Wartości mogą być od 4 (80486) do 7 (Itanium) oraz 15 (co znaczy sprawdź rozszerzone informacje o rodzinie)
   - bity 12-13: typ procesora (0=Original OEM Processor, 1=Intel Overdrive, 2=Dual)
   - bity 16-19 (jeśli jest taka możliwość): rozszerzona informacja o modelu.
   - bity 20-27 (jeśli jest taka możliwość): rozszerzona informacja o rodzinie.

   EDX = cechy procesora (tutaj akurat z procesorów Intela; najpierw numery bitów):
   - 0: procesor zawiera FPU
   - 1: Virtual 8086 Mode Enchancements
   - 2: Debugging Extensions
   - 3: Page Size Extension
   - 4: Time Stamp Counter
   - 5: Model Specific Registers
   - 6: Physical Address Extensions
   - 7: Machine Check Exception
   - 8: instrukcja CMPXCHG8B
   - 9: procesor zawiera Zaawansowany Programowalny Kontroler Przerwań (APIC)
   - 11: instrukcje SYSENTER i SYSEXIT
   - 12: Memory Type Range Registers

- 13: Page Table Entries Global Bit
- 14: Machine Check Architecture
- 15: instrukcje CMOV*
- 16: Page Attribute Table
- 17: 32-bit Page Size Extensions
- 18: numer seryjny procesora
- 19: instrukcja CLFLUSH
- 21: Debug Store
- 22: monitorowanie temperatury i możliwość modyfikacji wydajności procesora
- 23: technologia MMX
- 24: instrukcje FXSAVE i FXRSTOR
- 25: technologia SSE
- 26: technologia SSE2
- 27: Self-Snoop
- 28: technologia Hyper-Threading
- 29: monitorowanie temperatury, układy kontroli temperatury
- 31: Pending Break Enable

3. EAX=2
   EBX, ECX, EDX = informacje o pamięci podręcznej cache i TLB

Nawet te informacje, które tu przedstawiłem są już bardzo szczegółowe i z pewnością nie będą takie same na wszystkich procesorach. To jest tylko wstęp. Dalsze informacje można znaleźć na stronach producentów procesorów, np. AMD, Intel, ale także tutaj: Sandpile, Lista przerwań Ralfa Brown'a (plik opcodes.lst)

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Pobieranie i ustawianie daty oraz godziny

W DOSie do pobierania bieżącej daty służy bezargumentowa funkcja numer 2Ah przerwania 21h. Po jej wywołaniu, w rejestrze CX dostajemy bieżący rok, w DH - miesiąc, a w DL - dzień miesiąca. Ponadto, w AL dostajemy numer dnia tygodnia (0 oznacza niedzielę)
Datę ustawia się, podając te same dane (z wyjątkiem dnia tygodnia) w tych samych rejestrach i wywołując funkcję 2Bh przerwania 21h.

Czas pobiera się bezargumentową funkcją 2Ch przerwania 21h. Po jej wywołaniu, w rejestrze CH dostajemy bieżącą godzinę, w CL - minutę, a w DH - sekundę. Aby zmienić bieżący czas systemowy, te same argumenty w tych samych rejestrach podajemy funkcji 2Dh przerwania 21h.

Oto krótki program dla NASMa, ilustrujący omówione funkcje:

```
; Program pobierający bieżącą datę i godzinę. Program NIC NIE WYŚWIETLA.
;
; Autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
;   nasm -f bin -o dataczas.com dataczas.asm

org 100h

        mov     ah, 2ah         ; 2B = ustaw
        int     21h

        mov     [rok], cx
        mov     [mies], dh
        mov     [dzien], dl
        mov     [dzient], al

        mov     ah, 2ch         ; 2D = ustaw
        int     21h

        mov     [godz], ch
        mov     [min], cl
        mov     [sek], dh

        mov     ax, 4c00h
        int     21h

rok     dw      0
mies    db      0
dzien   db      0
dzient  db      0

godz    db      0
min     db      0
sek     db      0
```

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Zabawa diodami na klawiaturze

Aby uczynić swój program bardziej atrakcyjnym wzrokowo i pochwalić się swoimi umiejętnościami, można sprawić, aby diody na klawiaturze wskazujące stan Num Lock, Caps Lock, Scroll Lock zaczęły migotać w jakimś rytmie.
Teraz pokażę, jak to zrobić.

Najpierw, tradycyjnie, spojrzymy w spis portów dołączony do Listy Przerwań Ralfa Brown'a. Potrzebny nam będzie podstawowy port kontrolera klawiatury, port 60h:

(przeskocz port 60h)

```
0060 RW KB controller data port or keyboard input buffer (ISA, EISA)
        should only be read from after status port bit0 = 1
        should only be written to if status port bit1 = 0
```

Jak widać, trzeba też znaleźć jakiś port statusu. Jest to port 64h:
(przeskocz port 64h)

```
Bitfields for keyboard controller read status (ISA, EISA):
Bit(s)  Description     (Table P0398)
 7      parity error on transmission from keyboard
 6      receive timeout
 5      transmit timeout
 4      keyboard interface inhibited by keyboard lock
        or by password server mode
 3      =1 data written to input register is command (PORT 0064h)
        =0 data written to input register is data (PORT 0060h)
 2      system flag status: 0=power up or reset  1=selftest OK
 1      input buffer full (input 60/64 has data for 8042)
        no write access allowed until bit clears
 0      output buffer full (output 60 has data for system)
        bit is cleared after read access
```

Tak więc, potrzebna nam będzie procedura sprawdzająca, czy można pisać do portu klawiatury. Spróbujmy ją napisać:
(przeskocz procedurę sprawdzającą zajętość portu)

```
; wersja TASM
czy_mozna_pisac         proc    near
        push eax
sprawdzaj:
        in al,64h
        and al,2                ; sprawdzamy bit nr. 1
        jnz sprawdzaj           ; jeśli różny od zera, to
                                ; sprawdzaj do skutku

        pop eax
        ret
czy_mozna_pisac         endp
```

Teraz wersja NASM/FASM:
(przeskocz wersję NASM/FASM tej procedury)

```
; wersja NASM
```

```
czy_mozna_pisac:
        push eax
sprawdzaj:
        in al,64h
        and al,2                ; sprawdzamy bit nr. 1
        jnz sprawdzaj           ; jeśli różny od zera, to
                                ;sprawdzaj do skutku

        pop eax
        ret
```

Ta powinna wystarczyć.
Trzeba jeszcze znaleźć polecenie kontrolera klawiatury, które kontroluje stan diód. Jest to bajt EDh:
(przeskocz komendę ustawiania diód)

```
EDh             set/reset mode indicators Caps Num Scrl
                bit 2 = CapsLk, bit 1 = NumLk, bit 0 = ScrlLk
                 all other bits must be zero.
```

Możemy już zacząć coś pisać:
(przeskocz pierwszy program)

```
call czy_mozna_pisac
MOV AL,0EDh
OUT 60h,AL
XOR AL,AL               ;żadna dioda się nie pali
OUT 60h,AL
call czy_mozna_pisac
MOV AL,0EDh
OUT 60h,AL
MOV AL,2                ;Num Lock
OUT 60h,AL
call czy_mozna_pisac
MOV AL,0EDh
OUT 60h,AL
MOV AL,1                ;Scroll Lock
OUT 60h,AL
call czy_mozna_pisac
MOV AL,0EDh
OUT 60h,AL
MOV AL,6                ;Caps+Num
OUT 60h,AL
```

To był tylko przykład. No więc uruchamiamy go i co? Bzyk! I już nasz program się zakończył. Może komuś udało się zaobserwować efekty (z wyjątkiem ostatniego, który jest trwały). To stawia 2 pytania:

1. Jak sprawić, żeby trwało to dłużej?
2. Jak powrócić do stanu pierwotnego, zgodnego z prawdą?

Odpowiedzią na pierwsze pytanie jest już użyta raz przeze mnie w innym artykule funkcja 86h przerwania 15h. Przypomnę: CX:DX = liczba mikrosekund przerwy, którą chcemy uzyskać.
Po dodaniu niezbędnych linijek program może wyglądać tak:
(przeskocz program z opóźnieniami)

```
MOV AH,86h
```

```
MOV CX,0Fh
MOV DX,4240h

call czy_mozna_pisac
MOV AL,0EDh
OUT 60h,AL
XOR AL,AL               ;żadna dioda się nie pali
OUT 60h,AL
INT 15h
;MOV AH,86h
;INT 15h
call czy_mozna_pisac
MOV AL,0EDh
OUT 60h,AL
MOV AL,2                ;Num Lock
OUT 60h,AL
MOV AH,86h
INT 15h
;MOV AH,86h
;INT 15h
```

i tak dalej...
Jeśli zauważycie, że to nic nie daje, to odkomentujcie drugie wywołania przerwania. Rejestr AH musi być przed każdym wywołaniem przywracany, gdyż przerwanie go modyfikuje.
A co z drugim pytaniem?
Z pomocą tym razem przychodzi spis przerwań. Patrzymy:

(przeskocz opis funkcji 2 przerwania 16h)

```
INT 16 - KEYBOARD - GET SHIFT FLAGS
        AH = 02h
Return: AL = shift flags (see #00582)
        AH destroyed by many BIOSes

Bitfields for keyboard shift flags:
Bit(s)  Description     (Table 00582)
 7      Insert active
 6      CapsLock active
 5      NumLock active
 4      ScrollLock active
 3      Alt key pressed
 2      Ctrl key pressed
 1      left shift key pressed
 0      right shift key pressed
```

Nasz programik będzie więc wyglądał mniej-więcej tak:
(przeskocz program z opóźnieniami i z przywracaniem stanu)

```
        MOV AH,2
        INT 16h
        MOV BH,AL               ; zachowujemy stary stan klawiatury

        MOV AH,86h
        MOV CX,0Fh
        MOV DX,4240h


        call czy_mozna_pisac
```

```
        MOV AL,0EDh
        OUT 60h,AL
        XOR AL,AL               ;żadna dioda się nie pali
        OUT 60h,AL
        INT 15h
        ;MOV AH,86h
        ;INT 15h
        ...
        ...

        XOR AL,AL
        TEST BH,01000000b       ; czy Caps był włączony?
        JZ nie_caps
        OR AL,4                 ; tak, ustaw bit 2
nie_caps:
        TEST BH,00100000b       ; czy Num?
        JZ nie_num
        OR AL,2
nie_num:
        TEST BH,00010000b       ; czy Scroll?
        JZ koniec
        OR AL,1
koniec:
        MOV BL,AL
        MOV AL,0EDh
        OUT 60h,AL
        MOV AL,BL
        OUT 60h,AL
        ...
```

Dalsze eksperymenty pozostawiam czytelnikom. Pamiętajcie, że istnieje aż 8 różnych kombinacji stanów diód i można przecież robić różne odstępy czasowe między zmianą stanu.

Miłej zabawy.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Opis najczęściej używanych funkcji podstawowych przerwań

Najlepszy opis wszystkich funkcji wszystkich przerwań można znaleźć w Ralf Brown's Interrupt List: (RBIL)

Spis treści:

---

int 10h (przerwanie karty graficznej)

- funkcja 0 - ustaw tryb graficzny:
  Argumenty:
  - AH = 0
  - AL = żądany tryb graficzny (patrz niżej)

  Podstawowe tryby graficzne i ich rozdzielczości:
  0 - tekstowy, 40x25, segment 0B800
  1 - tekstowy, 40x25, segment 0B800
  2 - tekstowy, 80x25, segment 0B800
  3 - tekstowy (tradycyjny), 80x25, segment 0B800
  12h - graficzny, 640x480 w 16/256tys. kolorach, segment 0A000
  13h - graficzny, 320x200 w 256 kolorach, segment 0A000

- funkcja 2 - ustaw pozycję kursora tekstowego:
  Argumenty:
  - AH = 2
  - BH = numer strony, zazwyczaj 0
  - DH = wiersz (0 oznacza górny)
  - DL = kolumna (0 oznacza lewą)

- funkcja 3 - pobierz pozycję kursora tekstowego i jego rozmiar:
  Argumenty:
  - AH = 3
  - BH = numer strony, zazwyczaj 0

  Zwraca:
  - CH = początkowa linia skanowania (górna granica kursora)
  - CL = końcowa linia skanowania (dolna granica kursora)
  - DH = wiersz (0 oznacza górny)
  - DL = kolumna (0 oznacza lewą)

- funkcja 0Eh - wypisz znak na ekran:
  Argumenty:
  - AH = 0Eh
  - AL = kod ASCII znaku do wypisania

- BH = numer strony, zazwyczaj 0
- BL = kolor (tylko w trybach graficznych)

- funkcja 0Fh - pobierz tryb graficzny:
  Argumenty:
  - AH = 0Fh

  Zwraca:
  - AH = liczba kolumn znakowych
  - AL = bieżący tryb graficzny
  - BH = aktywna strona

---

int 13h (obsługa dysków)

- funkcja 2 - czytaj sektory dysku do pamięci:
  Argumenty:
  - AH = 2
  - AL = liczba sektorów do odczytania (musi być niezerowa)
  - CH = najmłodsze 8 bitów numeru cylindra
  - CL =
    bity 0-5: numer sektora (1-63)
    bity 6-7: najstarsze 2 bity numeru cylindra (tylko dla twardych dysków)
  - DH = numer głowicy
  - DL = numer dysku, dla dysków twardych bit7=1 (0=dysk A:, 1=B:, 80h=C:, 81h=D:, ...)
  - ES:BX = adres miejsca, gdzie będą zapisane dane odczytane z dysku

  Zwraca:
  - flaga CF=1, jeśli wystąpił błąd; CF=0, gdy nie było błędu
  - AH=status (patrz niżej)
  - AL=liczba przeczytanych sektorów (nie zawsze prawidłowy)

  Podstawowe wartości statusu:
  - 0 - operacja zakończyła się bez błędów
  - 3 - dysk jest chroniony przed zapisem
  - 4 - sektor nie znaleziony / błąd odczytu
  - 6 - zmiana dyskietki. Najczęściej spowodowany tym, że napęd nie zdążył się rozpędzić. Ponowić próbę.
  - 80h - przekroczony limit czasu operacji. Dysk nie jest gotowy.

  Przykład (czytanie bootsektora):

```
mov     ax, 0201h       ; funkcja czytania sektorów
xor     dx, dx          ; głowica 0, dysk 0 = A:
mov     cx, 1           ; numer sektora
mov     bx, bufor       ; dokąd czytać
int     13h             ; czytaj
jnc     czyt_ok         ; sprawdź, czy błąd
```

- funkcja 3 - zapisz dane z pamięci na sektorach dysku:
  Argumenty:
  - AH = 3
  - AL = liczba sektorów do zapisania (musi być niezerowa)
  - CH = najmłodsze 8 bitów numeru cylindra
  - CL =
    bity 0-5: numer sektora (1-63)
    bity 6-7: najstarsze 2 bity numeru cylindra (tylko dla twardych dysków)
  - DH = numer głowicy
  - DL = numer dysku, dla dysków twardych bit7=1 (0=dysk A:, 1=B:, 80h=C:, 81h=D:, ...)
  - ES:BX = adres miejsca, skąd będą pobierane dane do zapisania na dysk

  Zwraca:
  - flaga CF=1, jeśli wystąpił błąd; CF=0, gdy nie było błędu
  - AH=status (patrz wyżej)
  - AL=liczba zapisanych sektorów (nie zawsze prawidłowy)

  Przykład (zapisywanie bootsektora):

```
mov     ax, 0301h       ; funkcja zapisu sektorów
xor     dx, dx          ; głowica 0, dysk 0 = A:
mov     cx, 1           ; numer sektora
mov     bx, bufor       ; skąd brać dane do zapisu
int     13h             ; zapisz
jnc     blad            ; sprawdź, czy błąd
```

---

int 15h (część BIOS-u)

- funkcja 86h - czekaj określoną liczbę milisekund:
  Argumenty:
  - AH = 86h
  - CX:DX = czas w milisekundach

  Zwraca:
  - flaga CF=0, gdy nie wystąpił błąd; CF=1 po błędzie
  - AH = status:
    80h nieprawidłowa komenda (PC,PCjr)
    83h funkcja już trwa
    86h funkcja nie jest obsługiwana (XT)

---

int 16h (obsługa klawiatury)

- funkcja 0 - pobierz kod naciśniętego klawisza (lub czekaj na naciśnięcie):
  Argumenty:

- AH = 0

Zwraca:

- AH = BIOSowy kod klawisza (skankod)
- AL = kod klawisza ASCII

- funkcja 1 - sprawdź, czy naciśnięto klawisz:
  Argumenty:
  - AH = 1
  Zwraca:
  - flaga ZF=1, gdy nie naciśnięto klawisza
  - flaga ZF=0, gdy naciśnięto klawisz:
    AH = BIOSowy kod klawisza (skankod)
    AL = kod klawisza ASCII

- funkcja 2 - pobierz stan klawiszów przełączających:
  Argumenty:
  - AH = 2
  Zwraca:
  - AL = flagi:
    bit7 = klawisz Insert jest aktywny
    bit6 = CapsLock aktywny
    bit5 = NumLock aktywny
    bit4 = Scroll Lock aktywny
    bit3 = naciśnięty klawisz ALT
    bit2 = naciśnięty klawisz CTRL
    bit1 = naciśnięty lewy klawisz SHIFT
    bit0 = naciśnięty prawy klawisz SHIFT

- funkcja 4 (Tandy 2000, ale chyba nie tylko) - opróżnij bufor klawiatury:
  Argumenty:
  - AH = 4

---

int 21h (DOS)

- funkcja 1 - czytaj klawisz:
  Argumenty:
  - AH = 1
  Zwraca:
  - AL = kod klawisza ASCII

- funkcja 2 - wyświetl znak:
  Argumenty:
  - AH = 2
  - DL = kod ASCII znaku do wypisania

- funkcja 9 - wyświetl napis:
  Argumenty:
  - AH = 9
  - DS:DX = adres łańcucha znaków zakończonego znakiem dolara $

- funkcja 2A - pobierz datę systemową:
  Argumenty:
  - AH = 2Ah

  Zwraca:
  - CX = rok (1980-2099)
  - DH = miesiąc
  - DL = dzień
  - podobno AL = dzień tygodnia (0=niedziela)

- funkcja 2B - ustaw datę systemową:
  Argumenty:
  - AH = 2Ah
  - CX = rok (1980-2099)
  - DH = miesiąc (1-12)
  - DL = dzień (1-31)

  Zwraca:
  - AL = status (0=sukces, FF=błąd)

- funkcja 2C - pobierz czas systemowy:
  Argumenty:
  - AH = 2Ch

  Zwraca:
  - CH = godzina
  - CL = minuta
  - DH = sekunda
  - DL = setne sekundy (nie zawsze)

- funkcja 2D - ustaw czas systemowy:
  Argumenty:
  - AH = 2Dh
  - CH = godzina
  - CL = minuta
  - DH = sekunda
  - DL = setne sekundy

  Zwraca:
  - AL = status (0=sukces, FF=błąd)

- funkcja 3C - utwórz plik (jeśli istnieje, skróć do zerowej długości):
  Argumenty:

- AH = 3Ch
- CX = atrybuty (patrz niżej)
- DS:DX = wskaźnik na nazwę pliku, zakończoną bajtem zerowym

Zwraca:

- gdy brak błędu: flaga CF=0 i AX = uchwyt do pliku
- gdy błąd: flaga CF=1 i AX = numer błędu: 3, 4 lub 5 (patrz niżej)

Atrybuty pliku:

- bit0 = plik tylko do odczytu
- bit1 = ukryty
- bit2 = systemowy
- bit3 = etykieta dysku (ignorowane)
- bit4 = zarezerwowany, musi być równy 0 (katalog)
- bit5 = bit archiwalny
- bit7 = udostępnialność w Novell NetWare

Najczęstsze kody błędów:

- 0 = brak błędu
- 1 = nieprawidłowy numer funkcji
- 2 / 3 = plik / ścieżka nie znaleziona
- 4 = za dużo otwartych plików
- 5 = brak dostępu
- 6 = niewłaściwy uchwyt do pliku
- 8 = za mało pamięci
- A = nieprawidłowe środowisko
- B = nieprawidłowy format
- C = nieprawidłowy kod dostępu
- 56h = nieprawidłowe hasło

Przykład:

```
mov     ah, 3ch                 ; utwórz plik
xor     cx, cx                  ; żadnych atrybutów
mov     dx, plik                ; DS:DX = adres nazwy pliku
int     21h
jnc     plik_ok                 ; sprawdź, czy wystąpił błąd
```

- funkcja 3D - otwórz istniejący plik:

Argumenty:

- AH = 3Dh
- AL = tryb dostępu (patrz niżej)
- DS:DX = adres nazwy pliku zakończonej bajtem zerowym
- CL = maska atrybutów pliku do wyszukiwania (tylko serwery)

Zwraca:

- gdy brak błędu: flaga CF=0 i AX = uchwyt do pliku
- gdy błąd: flaga CF=1 i AX = numer błędu: 1, 2, 3, 4, 5, C, 56h (patrz wyżej)

Tryb dostępu do pliku:

- bit0-2 = tryb dostępu:
  000 = tylko do odczytu

001 = tylko do zapisu
010 = odczyt/zapis
- bit3 = 0
- bit4-6 = tryb współdzielenia:
  000 = tryb kompatybilności
  001 = zabroń innym odczytu i zapisu
  010 = zabroń innym zapisu
  011 = zabroń innym odczytu
  100 = nie zabraniaj nikomu niczego
- bit7 = prywatność. Plik nie będzie dziedziczony przez procesy potomne

Przykład:

```
mov     ax, 3d02h               ; otwórz plik R/W, tryb zgodności
mov     dx, plik                ; DS:DX = adres nazwy pliku
int     21h
jnc     otw_ok
```

- funkcja 3E - zamknij plik:
  Argumenty:
  - AH = 3Eh
  - BX = uchwyt do pliku

  Zwraca:
  - gdy brak błędu: flaga CF=0
  - gdy błąd: flaga CF=1 i AX = numer błędu: 6 (patrz wyżej)

- funkcja 3F - czytaj z pliku:
  Argumenty:
  - AH = 3Fh
  - BX = uchwyt do pliku
  - CX = liczba bajtów do odczytania
  - DS:DX = adres bufora, który ma przyjąć dane

  Zwraca:
  - gdy brak błędu: flaga CF=0, AX = liczba przeczytanych bajtów
  - gdy błąd: flaga CF=1 i AX = numer błędu: 5, 6 (patrz wyżej)

- funkcja 40h - zapisz do pliku:
  Argumenty:
  - AH = 40h
  - BX = uchwyt do pliku
  - CX = liczba bajtów do zapisania
  - DS:DX = adres bufora zawierającego dane do zapisania

  Zwraca:
  - gdy brak błędu: flaga CF=0, AX = liczba zapisanych bajtów
  - gdy błąd: flaga CF=1 i AX = numer błędu: 5, 6 (patrz wyżej)

- funkcja 41h - skasuj plik:
  Argumenty:
  - AH = 41h
  - DS:DX = adres nazwy pliku, zakończonej bajtem zerowym
  - maska atrybutów (tylko serwery)

  Zwraca:
  - gdy brak błędu: flaga CF=0
  - gdy błąd: flaga CF=1 i AX = numer błędu: 2, 3, 5 (patrz wyżej)

- funkcja 42h - ustaw bieżącą pozycję w pliku:
  Argumenty:
  - AH = 42h
  - AL = skąd odliczamy pozycję:
    0 = początek pliku
    1 = bieżąca pozycja w pliku
    2 = koniec pliku
  - BX = uchwyt do pliku
  - CX:DX = liczba bajtów, o które chcemy się przesunąć (może być ujemna)

  Zwraca:
  - gdy brak błędu: flaga CF=0, DX:AX = nowa pozycja, w bajtach od początku pliku
  - gdy błąd: flaga CF=1 i AX = numer błędu: 1, 6 (patrz wyżej)

- funkcja 4B - załaduj i/lub uruchom inny program:
  Argumenty:
  - AH = 4Bh
  - AL = rodzaj działania:
    0 = załaduj i uruchom
    1 = załaduj, ale nie uruchamiaj
    3 = załaduj, nakładając na aktualny program
    4 = załaduj i uruchom w tle (tylko European MS-DOS 4.0)
  - DS:DX = adres nazwy programu, zakończonej bajtem zerowym. Nazwa musi uwzględniać rozszerzenie pliku.
  - ES:BX = adres bloku parametrów (patrz niżej)
  - CX = tryb (tylko dla AL=4):
    0 = proces potomny po zakończeniu zostaje umieszczony w stanie zombie
    1 = kod zakończenia procesu potomnego jest odrzucany

  Zwraca:
  - gdy brak błędu: flaga CF=0
  - gdy błąd: flaga CF=1 i AX = numer błędu: 1, 2, 5, 8, A, B (patrz wyżej)

  Blok parametrów (AL=0,1,4):
  - WORD: segment zmiennych środowiska dla procesu potomnego (0 = użyć środowiska rodzica)
  - DWORD: wskaźnik na linię poleceń dla programu uruchamianego
  - DWORD: wskaźnik na pierwszy FCB dla procesu potomnego (nieaktualne)
  - DWORD: wskaźnik na drugi FCB dla procesu potomnego (nieaktualne)
  - (AL=1) DWORD: po zakończeniu będzie zawierać początkowe SS:SP
  - (AL=1) DWORD: po zakończeniu będzie zawierać początkowe CS:IP

Przykład:

```
mov     [kom_ln+2], cs  ;uzupełniamy pola potrzebnych struktur
mov     [fcb1+2], cs
mov     [fcb2+2], cs

mov     ax, 4b00h       ; funkcja uruchomienia programu
mov     dx, program     ; adres nazwy programu
mov     bx, srod        ; środowisko i cała reszta
int     21h             ; uruchamiamy
....

fcb             db 3, " ", 0, 0, 0, 0, 0
linia_kom       db 0
                times 7fh db 0dh

srod            dw      0
kom_ln          dw      linia_kom, 0
fcb1            dw      fcb, 0
fcb2            dw      fcb, 0
```

- funkcja 4Ch - zakończ działanie bieżącego programu:
  Argumenty:
  - AH = 4Ch
  - AL = kod wyjścia (errorlevel) zwracany systemowi operacyjnemu (przyjmuje się, że AL=0 oznacza zakończenie bez błędów)

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Tryb graficzny w języku asembler

Na początek uprzedzam: jeśli myślicie o wysokich rozdzielczościach, to się zawiedziecie, gdyż ten kurs nie będzie takich omawiał. Jeśli naprawdę wolicie wysokie rozdzielczości, to poszukajcie w Internecie opisu standardu VESA lub DirectX API.

A my, zamiast bawić się w te wszystkie skomplikowane sprawy, zajmiemy się trybem 13h. Ten tryb oferuje rozdzielczość 320x200 w 256 kolorach (co widać też w Liście Przerwań Ralfa Brown'a - RBIL).
Ale najważniejszą jego cechą jest to, że 320x200 = 64000 < 64kB, więc cały ekran mieści się w jednym segmencie, co znacznie ułatwia pracę.

Ekran w trybie graficznym mieści się w segmencie 0A000h oraz:
0A000:0000 - pierwszy piksel (bajt, 256 możliwości)
0A000:0001 - drugi piksel
0A000:0002 - trzeci piksel
...

Do zmiany trybu graficznego używa się przerwania 10h, funkcji 0 (opis wyjęty z Listy przerwań Ralfa Brown'a):
przeskocz opis int 10h, ah=0

```
INT 10 - VIDEO - SET VIDEO MODE
        AH = 00h
        AL = desired video mode (see #00010)
Return: AL = video mode flag (Phoenix, AMI BIOS)
            20h mode > 7
            30h modes 0-5 and 7
            3Fh mode 6
        AL = CRT controller mode byte (Phoenix 386 BIOS v1.10)
Desc:   specify the display mode for the currently active display
         adapter
```

Jak widać, zmiana trybu graficznego na omawiany tryb 13h nie jest trudniejsza niż:

```
mov ax, 13h
int 10h
```

Powrót do tradycyjnego trybu tekstowego 80x25 wygląda tak:

```
mov ax, 3
int 10h
```

Pytanie brzmi: jak cokolwiek narysować?
Nic prostszego! Po prostu pod adres:
wiersz*320 + kolumna
zapisujemy odpowiedni bajt, np. tak (składnia TASM):

```
mov ax, 0a000h
mov es, ax
xor di,di
mov byte ptr es:[di],15         ; NASM/FASM:   mov byte [es:di],15
```

No ale 1 piksel to za mało na nasze ambicje, prawda?
Spróbujmy narysować poziomą linię (NASM):
przeskocz program rysujący linię poziomą

```
; nasm -O999 -o liniapoz.com -f bin liniapoz.asm

org 100h

        mov ax, 13h
        int 10h                 ; uruchom tryb graficzny 13h

        mov ax, 0a000h
        mov es, ax
        xor di, di

        mov al, 15
        mov cx, 10

        rep stosb               ; przenieś 10 bajtów wartości 15 pod
                                ; es:di = 0a000:0000

        xor ah, ah
        int 16h

        mov ax, 3
        int 10h                 ; powrót do trybu tekstowego

        mov ax, 4c00h
        int 21h
```

To chyba nie było zbyt trudne, prawda? No to spróbujmy coś trudniejszego: linia pionowa.

Cała filozofia w tym przypadku polega na tym, aby po narysowaniu piksela przejść o 1 wiersz niżej (czyli o 320 bajtów dalej). Piszmy więc (NASM):
przeskocz program rysujący linię pionową

```
; nasm -O999 -o liniapio.com -f bin liniapio.asm

org 100h

        mov     ax, 13h
        int     10h

        mov     ax, 0a000h
        mov     es, ax
        xor     di, di

        mov     al, 15
        mov     cx, 100

rysuj:
        mov     [es:di], al
        add     di, 320
        loop    rysuj

        xor     ah, ah
        int     16h

        mov     ax, 3
        int     10h
```

```
        mov     ax, 4c00h
        int     21h
```

Na razie było łatwo: rysować zaczynaliśmy w lewym górny rogu, więc DI był równy 0. A co, jeśli chcemy wyświetlać piksele gdzieś indziej?
Cóż, są dwie możliwości:

1. W czasie pisania programu (czyli przed kompilacją) znasz dokładną pozycję, gdzie będziesz rysować. W takim przypadku kompilator policzy DI za ciebie, wystarczy wpisać coś takiego:

   ```
   mov di, wiersz*320 + kolumna
   ```

   wstawiając w miejsce słów wiersz i kolumna znane przez siebie wartości.

2. Pozycja, gdzie będziesz rysować jest zmienna i zależy np. od tego, co wpisze użytkownik.
   Tutaj jest gorzej. Trzeba wpisać do programu instrukcje, które przemnożą wiersz przez 320 i dodadzą kolumnę. Należy raczej unikać powolnej instrukcji (I)MUL. Ten problem rozwiążemy tak (wiersz i kolumna to 2 zmienne po 16 bitów):

   ```
   mov ax, [wiersz]
   mov bx, [wiersz]        ; BX = AX
   shl ax, 8               ; AX = AX*256
   shl bx, 6               ; BX = BX*64 = AX*64
   add ax, bx              ; AX = AX*256 + AX*64 = AX*320 =
                           ;     = wiersz*320
   add ax, [kolumna]       ; AX = wiersz*320 + kolumna

   mov di,ax
   ```

Ostatni przykład: rysowanie okręgu (no, w każdym razie czegoś co miało być okręgiem a ma kształt bardziej przypominający elipsę...). Program ten wykorzystuje koprocesor do policzenia sinusów i kosinusów dla kątów od 0 do 360 stopni, przerobionych na radiany. Komentarze obok instrukcji FPU oznaczają stan stosu, od st(0) z lewej.
przeskocz program rysujący koło

```
; nasm -O999 -o kolo.com -f bin kolo.asm

org 100h

        mov ax, 13h
        int 10h

        mov ax, 0a000h
        mov es, ax

        mov cx, 360

        finit
        fldpi
        fild word [sto80]
```

```
        fdivp st1, st0          ; pi/180

        fld1
        fild word [r]           ; r, 1, pi/180
        fldz                    ; kąt=0, r, 1, pi/180

        mov al, 15

rysuj:
        fld st0                 ; kąt, kąt, r, 1, pi/180

        fmul st4                ; kąt w radianach
        mov di, 100*320 + 160   ; środek ekranu

        fsin                    ; sin(kąt), kąt, r, 1, pi/180
        fmul st2                ; sin(kąt)*r, kąt, r, 1, pi/180

        fistp word [wys]        ; kąt, r, 1, pi/180

        fld st0                 ; kąt, kąt, r, 1, pi/180
        fmul st4                ; kąt w radianach
        fcos                    ; cos(kąt), kąt, r, 1, pi/180
        fmul st2                ; r*cos(kąt), kąt, r, 1, pi/180

        fistp word [szer]       ; kąt, r, 1, pi/180

        add di, [szer]          ; dodajemy odległość poziomą

        mov dx, [wys]
        mov bx, dx
        shl dx, 8
        shl bx, 6
        add dx, bx              ; dx = wys*320

        sub di, dx              ; odejmujemy odległość pionową

        mov [es:di], al         ; wyświetlamy piksel

        fadd st0, st2           ; kat += 1

        dec cx
        jnz rysuj

        finit

        xor ah, ah
        int 16h

        mov ax, 3
        int 10h

        mov ax, 4c00h
        int 21h

r       dw      50
szer    dw      0
wys     dw      0
sto80   dw      180
```

Podobnie, używając FSIN i FCOS, można rysować np. linie ukośne, które pominąłem w tym kursie. Mam nadzieję, że po lekturze tego odcinka każdy bez problemów będzie rysował w tym dość prostym

(zwłaszcza do nauki) trybie graficznym.

Miłego eksperymentowania!

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Wykrywanie sprzętu

Niektóre programy nie tylko zajmują się przetwarzaniem danych, ale muszą też współpracować ze sprzętem, na przykład wykorzystać port szeregowy czy równoległy do przesyłania danych (czy to na drukarkę, czy do innego urządzenia). W tym artykule pokażę, jak wykrywać część urządzeń zainstalowanych w komputerze. Dobrze mieć spis przerwań Ralfa Brown'a pod ręką.

Wykrywanie ilości zainstalowanej pamięci RAM

(przeskocz wykrywanie pamięci)

UWAGA: NIE należy badać pamięci RAM, zapisując do niej określone bity pod każdy możliwy adres i sprawdzając, czy uda się odczytać te same bity (brak pamięci sygnalizowany jest odczytaniem FF). Część urządzeń w komputerze (zwłaszcza PCI) jest mapowana do pamięci i zapisywanie do pewnych obszarów jest równoznaczne z zapisywaniem do tych urządzeń, co może je poważnie uszkodzić!

Do odkrycia zainstalowanej ilości pamięci RAM można skorzystać z następujących funkcji BIOSu: int 15h z EAX=0e820h, int 15h z EAX=0000E820h oraz int 12h (najlepiej w tej kolejności).

Pierwsza z nich korzysta z 32-bitowych rejestrów, więc dopiero od procesora 386 można sprawdzać, czy jest dostępna. Kolejne uruchomienia tej funkcji zwracają informacje o kolejnych obszarach pamięci i ich typie, tworząc tym samym BIOSową mapę pamięci. Ta funkcja przyjmuje następujące argumenty:

- EAX = 0000E820h
- EDX = 534D4150h (stała)
- ES:DI - adres bufora o następującej strukturze:
  - 8 bajtów na adres obszaru pamięci
  - 8 bajtów na długość tego obszaru pamięci
  - 4 bajty na typ obszaru pamięci
- ECX - długość bufora spod ES:DI w bajtach (minimum to 20)
- EBX = adres początkowy, od którego BIOS ma zacząć sprawdzanie. Na początku jest to zero.

Jeśli wywołanie się powiedzie, funkcja zwraca, co następuje:

- flaga CF=0
- wskazany bufor zostaje wypełniony danymi
- EBX = następny adres, skąd kopiować (podajemy go w EBX jako początkowy do kolejnego wywołania) lub 00000000h jeśli koniec
- ECX = długość zwróconych informacji w bajtach

W przypadku niepowodzenia flaga CF=1. Przykładowe wwywołanie wygląda tak:

```
mov     ax, cs
mov     es, ax          ; jeśli bufor jest w sekcji kodu
mov     eax, 0e820h
mov     edx, 534D4150h
xor     ebx, ebx
mov     ecx, 20
mov     di, bufor
int     15h
```

```
        jc      blad

        ; tu operujemy na danych

blad:
...

bufor:
b_adres         dd 0, 0
b_dlugosc       dd 0, 0
b_typ           dd 0
```

Druga funkcja nie przyjmuje żadnych argumentów (poza numerem funkcji w AX) i zwraca ilość pamięci rozszerzonej od 1 MB do 16 MB, w kilobajtach, w AX. Jeśli wywołanie się nie powiedzie, flaga CF=1. Przykładowe wwywołanie wygląda tak:

```
        mov     ax, 0E801h
        int     15h
        jc      blad

        ; tu operujemy na danych

blad:
```

Trzecia funkcja (przerwanie int 12h) w ogóle nie przyjmuje żadnych argumentów, a zwraca liczbę kilobajtów ciągłej pamięci od bezwzględnego adresu 00000h.

Wykrywanie portów szeregowych i równoległych

(przeskocz wykrywanie portów)

Wykrywanie portów, o których wie BIOS, jest bardzo łatwe. Wystarczy zajrzeć do BDA (BIOS Data Area), czyli segmentu numer 40h, zawierajacego dane BIOSu.
Adresy kolejnych portów szeregowych (maksymalnie czterech) jako 16-bitowe słowa można znaleźć pod adresami 0040h:0000h, 0040h:0002h, 0040h:0004h, 0040h:0006h (choć ten ostatni adres może służyć do innych celów na nowszych komputerach), zaś adresy kolejnych portów równoległych (maksymalnie czterech) jako 16-bitowe słowa znajdują się pod adresami 0040h:0008h, 0040h:000Ah, 0040h:000Ch, 0040h:000Eh.

Jeśli dodatkowo chcecie wykryć rodzaj portu szeregowego, polecam kod darmowego sterownika myszy dla DOSa - CuteMouse (a szczególnie plik comtest.asm). Sterownik jest napisany w asemblerze i można go pobrać oraz obejrzeć jego kod źródłowy za darmo.

Wykryć rodzaj portów równoległych można za pomocą układów nimi sterujących, na przykład Intel 82091AA Advanced Integrated Peripheral (porty 22h-23h). Kod dla tego układu może wyglądać następująco:

```
        mov     al, 20h         ; numer rejestru, który chcemy odczytać
        out     22h, al         ; wysyłamy go na port adresu
        out     0edh, al        ; opóźnienie
        in      al, 23h         ; odczytujemy dane z portu danych
```

Informacje o portach równoległych znajdują się w bitach 5 i 6 odczytanego bajtu. Jeśli bity te mają wartość 0, to porty równoległe pracują w trybie zgodności z ISA, jeśli 1 - w trybie zgodności z PS/2, jeśli 2 - w trybie EPP, jeśli 3 - w trybie ECP.

Wykrywanie karty dźwiękowej AdLib

(przeskocz wykrywanie AdLib)

Karta ta ma dwa podstawowe porty: port adresu i stanu - 388h (do odczytu i zapisu) oraz port danych - 389h (tylko do zapisu). By zapisać coś do jednego z 244 rejestru karty, wysyłamy jego numer na port 388h, po czym wysyłamy dane na port 389h. Algorytm wykrywania karty składa się z następujących kroków (źródło: *Programming the AdLib/Sound Blaster FM Music Chips, Version 2.0 (24 Feb 1992), Copyright © 1991, 1992 by Jeffrey S. Lee*):

1. wyzerowanie obu czasomierzy poprzez zapisanie 60h do rejestru 4.
2. włączenie przerwań, zapisując 80h do rejestru 4. UWAGA: to musi być krok oddzielny od pierwszego
3. odczytanie stanu karty (port 388h) i zachowanie wyniku
4. zapisanie FFh do rejestru 2 (czasomierz 1)
5. uruchomienie czasomierza 1 poprzez zapisanie 21h do rejestru 4. Czasomierz 1 będzie zwiększał wartość zapisaną do rejestru 2 o 1 co każde 80 mikrosekund.
6. odczekanie co najmniej 80 mikrosekund
7. odczytanie stanu karty (port 388h) i zachowanie wyniku
8. wyzerowanie czasomierzy i przerwań (krok 1 i 2)
9. wyniki kroków 3 i 7 ANDować bitowo z wartością E0h. Wynikiem z kroku 3 powinna być wartość 0, a z kroku 7 - C0h. Jeśli obie się zgadzają, w komputerze zainstalowana jest karta AdLib.

Między każdym zapisem do portu adresu i wysłaniem danych należy odczekać 12 cykli karty. Po zapisaniu danych należy odczekać 84 cykle karty, zanim jakakolwiek kolejna operacja będzie mogła zostać wykonana. Ale że wygodniej jest operować w języku operacji niż cykli procesora karty, te czasy oczekiwania wynoszą odpowiednio: 6 i 35 razy czas potrzebny na odczytanie portu adresu. Ja w razie czego użyję odpowiednio: 10 i 40 operacji.

Do wykrywania karty AdLib może posłużyć więc następujący kod:

```
pisz_adlib 4, 60h
pisz_adlib 4, 80h

mov     dx, 388h
in      al, dx
mov     bl, al          ; zachowanie stanu w kroku 3

pisz_adlib 2, 0FFh
pisz_adlib 4, 21h

mov     ah, 86h
xor     cx, cx
mov     dx, 100
int     15h             ; wykonanie pauzy na 100 mikrosekund
jc      blad

mov     dx, 388h
in      al, dx
```

```
        mov     bh, al          ; zachowanie stanu w kroku 7

        pisz_adlib 4, 60h
        pisz_adlib 4, 80h

        and     bx, 0E0E0h
        cmp     bx, 0C000h      ; sprawdzenie obu wyników (kroki 3 i 7) na raz
        je      jest_adlib

        ; tu nie ma AdLib
```

gdzie makro `pisz_adlib` wygląda tak:

```
%imacro pisz_adlib 2    ; %1 - numer rejestru, %2 - dane do wysłania
        mov     dx, 388h
        mov     al, %1
        out     dx, al
        mov     cx, 10
%%loop1:                ; opóźnienie pierwsze
        in      al, dx
        loop    %%loop1
        inc     dx      ; port 389h
        mov     al, %2
        out     dx, al
        dec     dx
        mov     cx, 40
%%loop2:                ; opóźnienie drugie
        in      al, dx
        loop    %%loop2
%endm
```

Wykrywanie karty dźwiękowej SoundBlaster

(przeskocz wykrywanie SB)

Karta SoundBlaster może być zaprogramowana do korzystania z różnych portów podstawowych. Najczęściej spotykana wartość to 220h, ale możliwe są też między innymi 210h, 230h, 240h, 250h, 260h i 280h. Struktura jest podobna, jak w karcie AdLib: zakładając, że port bazowy to 220h, to dla lewego kanału portem adresu jest 220h, a portem danych - 221h, zaś dla prawego - odpowiednio 222h i 223h. Porty karty AdLib - 388h i 389h - służą do operacji na obu kanałach.
Wykrywanie tej karty przebiega tak samo, jak dla karty AdLib (procedura 9 kroków powyżej), ale skoro porty bazowe mogą być różne, proponuję następującą modyfikację makra do wysyłania danych:

```
%imacro pisz_sb 3       ; %1 - port bazowy, %2 - numer rejestru, %3 - dane
        mov     dx, %1
        mov     al, %2
        out     dx, al
        mov     cx, 6
%%loop1:                ; opóźnienie pierwsze
        in      al, dx
        loop    %%loop1
        inc     dx      ; port danych
        mov     al, %3
        out     dx, al
        dec     dx
```

```
        mov     cx, 35
%%loop2:                ; opóźnienie drugie
        in      al, dx
        loop    %%loop2

%endm
```

Wykrywanie zainstalowanych dysków twardych

(przeskocz wykrywanie dysków)

Jeśli BIOS wykryje jakieś dyski twarde, ich liczbę wpisuje do komórki pamięci pod adresem 0040h:007Eh (1 bajt).
Zakresy portów kontrolerów dysków twardych to: 01F0h-01F7h (pierwszy kontroler), 0170h-0177h (drugi). Są jeszcze 2 kontrolery, opisane jako EIDE: 01E8h-01EFh (trzeci kontroler) i 168h-016Fh (czwarty).

Każdy kontroler może obsłużyć dwa dyski - Master i Slave. Wyboru dysku, na którym wykonywane są operacje, dokonuje się, zapisując do portu baza+6 (gdzie baza to 01F0h, 0170h, 01E8h lub 168h). Bity 7 i 5 muszą być równe 1, a bitem czwartym wybiera się dysk (0=pierwszy, 1=drugi).
Komendy wysyła się do portu baza+7, a dane (po 512 bajtów) odczytuje się z portu bazowego. Przed wysłaniem komend należy sprawdzić, czy kontroler lub dysk nie są zajęte. Robi się to odczytując port stanu, będący zarazem portem komend (czyli baza+7). Bit 7 mówi, czy kontroler jest zajęty (powinien być równy zero), bit 6 - czy dysk jest gotowy do operacji (powinien być równy 1), bit 4 - czy dysk przeszedł na właściwą pozycję (powinien być równy 1). Reszta bitów jest nieistotna, jeśli chodzi o wysyłanie komend.
Portu statusu można użyć też, obok portu baza+1, do wykrywania błędów.

Możemy już więc napisać taki oto kod:

```
        mov     dx, 1f7h
spr_dysk:
        in      al, dx
        cmp     al, 50h         ; dysk gotowy, kontroler niezajęty
        jnz     spr_dysk
```

Gdy dysk jest gotów na przyjmowanie komend, można zacząć wysyłać nasze żądania. Najpierw ustawiamy, do którego dysku będziemy chcieli wysyłać dane:

```
        mov     dx, 1f6h
        mov     al, 10100000b   ; bit 4 = 0, wiec pierwszy dysk
        out     dx, al
```

Po tym, w razie czego, sprawdzamy ponownie gotowość dysku poprzednim kodem. Jeśli dysk jest gotów, wysyłamy komendę:

```
        mov     dx, 1f7h
        mov     al, 0ech        ; kod rozkazu identyfikacji
        out     dx, al
```

Przed odczytaniem danych musimy jednak sprawdzić nie tylko, czy dysk już jest gotów (czy skończył przetwarzać żądanie), ale też to, czy dane już są gotowe do odebrania. Sprawdzamy to podobnie, jak poprzednio, zamieniając tylko 50h na 58h (co dodatkowo sprawdza, czy bufor sektorów dysku wymaga

obsługi - czyli czy są już dla nas dane):

```
        mov     dx, 1f7h
spr_dysk:
        in      al, dx
        cmp     al, 58h         ; dysk gotowy, kontroler niezajęty, są dane
        jnz     spr_dysk
```

Po sprawdzeniu, że dane są dostępne, odbieramy je, lecz w nietypowy sposób: zamiast odbierać po jednym bajcie, odbieramy pod dwa na raz, do rejestru AX, po czym zamieniamy jego połówki miejscami. Jest to związane ze sposobem wysyłania danych przez dysk. Kod wygląda tak:

```
        mov     cx, 512/2       ; tyle słów do przeczytania
        mov     dx, 1f0h        ; stąd czytać
        xor     di, di          ; wskaźnik do bufora
czytaj:
        in      ax, dx          ; wczytaj 2 bajty
        xchg    al, ah          ; zamień połówki miejscami
        mov     [bufor+di], ax  ; zapisz wynik do bufora
        add     di, 2           ; przejdź na kolejną pozycję w buforze
        loop    czytaj
        ...
bufor:          times 513 db 0  ; dość, by pomieścić 1 sektor
```

Dysk zwraca nam 512 bajtów. Model dysku znajdziecie pod adresem 14h w buforze, ma on długość 10 słów (20 bajtów). Numer seryjny jest pod adresem 36h w buforze, ma on długość 20 słów (40 bajtów). W obu tych przypadkach, jeśli pierwszym słowem pod wskazanym adresem jest zero, to dysk nie podał tych informacji.

Pozyskanie tych informacji od napędów optycznych (CD, DVD) różni się tylko kodem operacji - zamiast ECh jest to A1h.

Wykrywanie napędów dyskietek

(przeskocz wykrywanie napędów dyskietek)

Wykrywanie typów napędów dyskietek jest znacznie prostsze niż w przypadku dysków twardych. W czasie uruchamiania komputera, BIOS wyszukuje napędy dyskietek i wpisuje je do CMOSu, skąd można je łatwo odczytać. Ze te informacje odpowiada bajt numer 10h. Odczytanie go wygląda tak:

```
        mov     al, 10h         ; numer bajtu do odczytania
        out     70h, al         ; port adresu CMOSu
        out     0edh, al        ; opóźnienie
        in      al, 71h         ; odczytanie wartości z portu danych CMOSu
```

Starsze 4 bity odczytanego bajtu odpowiadają pierwszemu napędowi, młodsze - drugiemu. I tak: wartość 0 oznacza brak danego napędu, 01h - 5,25 cala 360 kB, 02h - 5,25 cala 1,2 MB, 03h - 3,5 cala 720 kB, 04h - 3,5 cala 1,44 MB, 05h - 3,5 cala 2,88 MB.

Wykrywanie myszy

(przeskocz wykrywanie myszy)

Ogólnie wykrywanie myszy jako urządzenia może być dość skomplikowane, nie tylko ze względu na różnorodność złączy (szeregowa, PS/2, USB), ale także ze względu na różnorodność protokołów komunikacji z myszami. Wszystko to na szczęście jest zawarte w otwartym sterowniku myszy dla DOSa - CuteMouse. Sterownik jest napisany w asemblerze i można go pobrać oraz obejrzeć jego kod źródłowy za darmo.

Jeśli wystarczy Wam wiedzieć, czy jest załadowany jakikolwiek sterownik do myszy (co wskazywałoby na istnienie myszy), wystarczy taki oto kod:

```
xor     ax, ax
mov     es, ax
les     di, [es:33h << 2]       ; sprawdź, czy wektor przerwania
                                ; sterownika myszy nie jest zerem
mov     ax, es
or      ax, di
jz      brak_myszy

mov     al, [es:di]
cmp     al, 0cfh                ; sprawdź, czy procedura obsługi
                                ; przerwania myszy nie składa się
                                ; wyłącznie z instrukcji iret
je      brak_myszy

xor     ax, ax
int     33h                     ; sprawdź, czy sterownik zgłasza mysz
test    ax, ax
jz      brak_myszy
```

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Obsługa myszy w asemblerze

W tej części dowiemy się, jak dodać w naszych programach obsługę myszy.
Do naszych celów potrzebne nam będą załadowane sterowniki myszy w pamięci oraz przerwanie int 33h.
Zapoznajmy się z kilkoma podstawowymi funkcjami tegoż przerwania (patrzymy oczywiście w listę przerwań Ralfa Brown'a - RBIL):
(przeskocz opis funkcji int 33h)

```
INT 33 - MS MOUSE - RESET DRIVER AND READ STATUS
        AX = 0000h
Return: AX = status
                0000h hardware/driver not installed
                FFFFh hardware/driver installed
        BX = number of buttons
                0000h other than two
                0002h two buttons (many drivers)
                0003h Mouse Systems/Logitech three-button mouse
                FFFFh two buttons

INT 33 - MS MOUSE v1.0+ - SHOW MOUSE CURSOR
        AX = 0001h

INT 33 - MS MOUSE v1.0+ - HIDE MOUSE CURSOR
        AX = 0002h

INT 33 - MS MOUSE v1.0+ - POSITION MOUSE CURSOR
        AX = 0004h
        CX = column
        DX = row

INT 33 - MS MOUSE v1.0+ - RETURN BUTTON RELEASE DATA
        AX = 0006h
        BX = button number (see #03169)
Return: AX = button states (see #03168)
        BX = number of times specified button has been released since
                last call
        CX = column at time specified button was last released
        DX = row at time specified button was last released

(Table 03169)
Values for mouse button number:
0000h  left
0001h  right
0002h  middle
```

Tyle powinno nam wystarczyć. Są też funkcje, które np. definiują zasięg poziomy i pionowy kursora (można ustawić tak, że kursor będzie się mógł poruszać tylko w wyznaczonym oknie), ale tymi nie będziemy się zajmować.

Na dobry początek resetujemy sterownik i sprawdzamy, czy w ogóle jest jakaś mysz zainstalowana (jeszcze wcześniej można byłoby sprawdzić, czy sam sterownik nie jest procedurą, która nic nie robi, ale pominę to). Kod jest wyjątkowo krótki:

```
xor     ax, ax
int     33h                     ; sprawdzamy, czy jest mysz
or      ax, ax
jz      brak_myszy
```

Jak było widać wcześniej, pokazanie kursora nie jest trudne i sprowadza się do:

```
        mov     ax, 1
        int     33h                     ; pokaż kursor
```

Do szczęścia brakuje nam już tylko sprawdzenie, czy i gdzie naciśnięto jakiś przycisk. Do tego posłuży nam funkcja numer 6. Wystarczy w BX podać interesujący nas przycisk, a w CX i DX powinniśmy otrzymać współrzędne (dobrze jest przed rozpoczęciem pracy wywołać raz tą funkcję dla wszystkich przycisków, aby wyzerować liczniki naciśnięć). Przykład:

```
        mov     ax, 6
        xor     bx, bx
        int     33h
        or      bx, bx
        jz      nie_wcisnieto_lewego

        mov     [kolumna], cx
        mov     [wiersz], dx
```

Nie ma w tym dużo wysiłku, na szczęście. Wszystko za nas robi sterownik, a my tylko potem sprawdzamy rejestry.
Dlatego też od razu przejdę do finału tego artykułu i zaprezentuję program, w którym zawarłem wszystko to, o czym mówiłem. Zadaniem programu jest nic innego, jak tylko wyświetlenie odpowiedniego napisu, gdy użytkownik naciśnie jakiś klawisz myszki (naciśnięcie czegoś na klawiaturze spowoduje wyjście z programu). Napis zostanie wyświetlony w miejscu, gdzie naciśnięto przycisk.

Oto kod:
(przeskocz kod programu)

```
; Program wyświetlający napis w miejscu, gdzie został naciśnięty
; klawisz myszki.
;
; POD WINDOWS URUCHAMIAĆ W TRYBIE PEŁNOEKRANOWYM
;
;
; Autor: Bogdan D.
; kontakt: bogdandr (at) op (dot) pl
;
; nasm -O999 -o mysz.com -f bin mysz.asm


org 100h

        xor     ax, ax                  ; AX = 0
        int     33h                     ; sprawdzamy, czy jest mysz
        or      ax, ax
        jnz     mysz_ok

        mov     dx, nie_ma_myszy
        mov     ah, 9
        int     21h

        mov     ax, 4c01h               ; zwracamy kod błędu=1
        int     21h
```

```
mysz_ok:
        mov     ax, 3
        int     10h                     ; czyścimy ekran

        mov     ax, 1
        int     33h                     ; pokazujemy kursor myszki

                                        ; zerujemy liczniki naciśnięć:
        mov     ax, 6
        xor     bx, bx                  ; lewy przycisk
        int     33h

        mov     ax, 6
        mov     bx, 1                   ; prawy przycisk
        int     33h

petla:
        mov     ah, 1
        int     16h             ; czy naciśnięto klawisz na klawiaturze?
        jnz     koniec          ; jeśli tak, to wychodzimy z programu

        mov     ax, 6
        xor     bx, bx                  ; sprawdzamy lewy przycisk
        int     33h

        or      bx, bx          ; jeśli naciśnięto, to idziemy coś wyświetlić
        jnz     pokaz_l

        mov     ax, 6
        mov     bx, 1                   ; sprawdzamy prawy przycisk
        int     33h

        or      bx, bx          ; jeśli naciśnięto, to idziemy coś wyświetlić
        jnz     pokaz_p

        jmp     short petla

pokaz_l:
                                ; wiemy, ze CX=kolumna, DX=wiersz, gdzie
                                ;naciśnięto klawisz, ale to są numery pikseli
                                ; Aby otrzymać kolumnę i wiersz, dzielimy je
                                ; przez 8 (bo jest 8 pikseli/znak):
        shr     dx, 3
        shr     cx, 3

        mov     ah, 2                   ; funkcja ustawiania kursora
        mov     dh, dl                  ; DH = wiersz
        mov     dl, cl                  ; DL = kolumna
        int     10h                     ; ustaw nasz kursor tam

        mov     dx, lewy_p
        mov     ah, 9
        int     21h

        jmp     short petla

pokaz_p:
                                ; wiemy, ze CX=kolumna, DX=wiersz, gdzie
                                ;naciśnięto klawisz, ale to są numery pikseli
                                ; Aby otrzymać kolumnę i wiersz, dzielimy je
                                ; przez 8 (bo jest 8 pikseli/znak):
        shr     dx, 3
```

```
        shr     cx, 3

        mov     ah, 2                   ; funkcja ustawiania kursora
        mov     dh, dl                  ; DH = wiersz
        mov     dl, cl                  ; DL = kolumna
        int     10h                     ; ustaw nasz kursor tam

        mov     dx, prawy_p
        mov     ah, 9
        int     21h

        jmp     short petla


koniec:
        mov     ax, 4c00h
        int     21h

nie_ma_myszy    db      "Sterowniki myszy nie sa zainstalowane.$"
lewy_p          db      "Lewy$"
prawy_p         db      "Prawy$"
```

Jak widać, korzystanie z myszy niekoniecznie musi być tak trudne, jak to się mogło wydawać. Po bardziej zaawansowane funkcje radzę sięgnąć do RBIL, gdzie zawsze znajdziecie więcej informacji o danym przerwaniu niż w moim kursie, który skupia się przecież na przedstawianiu algorytmów a nie na zapamiętywaniu każdego szczególiku.

Miłego eksperymentowania.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Porty szeregowe i równoległe

Niektórym programom nie wystarcza działanie na samym procesorze czy sprzęcie znajdującym się w komputerze. Czasem trzeba połączyć się z jakimś urządzeniem zewnętrznym, takim jak modem zewnętrzny czy drukarka. Celem tego artykułu jest właśnie pokazanie, jak to zrobić. Przydatny będzie spis przerwań Ralfa Brown'a (RBIL)
Informacje tu zgromadzone pochodzą z książki Art of Assembly autorstwa Randalla Hyde'a i z informacji pochodzących z RBIL. Nie miałem możliwości ich zweryfikowania.

Dostęp przez BIOS

(przeskocz BIOS)

BIOS oferuje nam dostęp tylko do portów szeregowych, za pośrednictwem przerwania int 14h.

1. AH=0 - inicjalizacja portu.
   Rejestr AL ma zawierać parametry portu:
   - bity 5-7: szybkość, w bitach na sekundę: 000 - 110bps, 001 - 150bps, 010 - 300bps, 011 - 600bps, 100 - 1200bps, 101 - 2400bps, 110 - 4800bps, 111 - 9600bps.
   - bity 3-4: tryb parzystości: 00 - brak, 01 - nieparzysta, 10 - brak, 11 - parzysta.
   - bit 2: liczba bitów stopu: 0 - 1 bit stopu, 1 - 2 bity stopu.
   - bity 0-1: bity danych: 10 - 7 bitów, 11 - 8 bitów danych

   Rejestr DX ma zawierać numer portu, od 0 dla COM1 do 3 dla COM4.
   Po wykonaniu tej operacji, w AX zwracany jest stan portu (patrz AH=3 poniżej).
   Przykład:

```
mov     ah, 0           ; numer funkcji
mov     al, 11100011b   ; 9600,8,N (brak parzystości),1
mov     dx, 0           ; COM1
int     14h
```

2. AH=1 - wyślij znak do portu.
   DX ma zawierać numer portu, jak powyżej. AL ma zawierać wysyłany znak.
   Jeśli po wykonaniu tej operacji najstarszy bit AH jest jedynką, to wystąpił błąd.
   Przykład:

```
mov     ah, 1           ; numer funkcji
mov     al, "a"         ; znak do wysłania
mov     dx, 0           ; COM1
int     14h
```

3. AH=2 - odbierz znak z portu.
   DX ma zawierać numer portu, jak powyżej.
   Po wykonaniu przerwania int 14h, AL będzie zawierać odebrany znak.
   Jeśli po wykonaniu tej operacji najstarszy bit AH jest jedynką, to wystąpił błąd.
   Przykład:

```
mov     ah, 2           ; numer funkcji
mov     dx, 0           ; COM1
int     14h             ; znak w AL, jeśli nie ma błędu
```

4. AH=3 - odczytaj stan portu portu.
   DX ma zawierać numer portu, jak powyżej.
   Po wykonaniu przerwania int 14h, AX będzie zawierać stan portu. Znaczenie kolejnych bitów przedstawia tabela:

Bity statusu portu

numer	co znaczy
15	Przekroczenie czasu oczekiwania
14	Rejestr przesunięcia transmisji (Transmitter shift register) pusty
13	Rejestr przechowania transmisji (Transmitter holding register) pusty
12	Błąd wykrywania przerwy
11	Błąd ramki
10	Błąd parzystości
9	Błąd przepełnienia
8	Dane są dostępne
7	Wykryto sygnał linii odbiorczej
6	Wskaźnik dzwonienia
5	Dane są gotowe (DSR)
4	Można wysyłać (CTS)
3	Wykryto sygnał linii odbiorczej delta
2	Wykryto dzwonek na krawędzi opadającej
1	Dane delta są gotowe
0	Można wysyłać dane delta

Dostęp poprzez instrukcje IN i OUT

(przeskocz dostęp przez porty)

1. Porty szeregowe.
   Dla portów szeregowych przeznaczone są porty sprzętowe: 03F8h-03FFh (COM1), 02E8h-02EFh (COM2), 02F8h-02FFh (COM3), 03E8h-03EFh (COM4).
   Pierwszy z każdej grupy portów (port bazowy) jest portem danych - na niego wysyła się bajty do transmisji i z niego odczytuje się bajty odebrane przez port.
   Port baza+3 jest portem kontroli linii. Ustawia się w nim parametry portu:
   - bity 3-5 to kontrola parzystości: xx0 - brak, 001 - nieparzysta, 011 - parzysta, 101 - wysoka, 111 - niska, xx1 - programowa (x oznacza dowolną wartość).
   - bit 2 określa liczbę bitów stopu: 0 - 1 bit stopu, 1 - 2 bity (jeśli bity danych to 6, 7 lub 8) lub półtora bitu stopu (jeśli bitów danych jest 5).
   - bity 0-1 mówią o ilości bitów danych: 00 - 5 bitów, 01 - 6 bitów, 10 - 7 bitów, 11 - 8 bitów

   Port baza+5 to port stanu linii (tylko do odczytu). Najważniejsze jego bity to:
   - bit 5 - jeśli równy 1, to kontroler może przyjąć kolejny znak do wysłania.
   - bit 2 - błąd parzystości
   - bit 1 - przepełnienie. Poprzedni znak w buforze został stracony.
   - bit 0 - odebrano cały bajt i jest on gotowy do przeczytania.

   Przykład:

```
        mov     dx, 3f8h + 5    ; status COM1
spr_gotowy:
        in      al, dx
        test    al, 20h         ; sprawdź bit 5
        jz      spr_gotowy      ; czekaj na gotowość
```

```
mov     dx, 3f8h        ; bazowy port COM1
mov     al, 55h         ; bajt do wysłania
out     dx, al          ; wyślij bajt
```

2. Porty równoległe.
   Dla portów równoległych przeznaczone są porty sprzętowe: 0278h-027Ah (LPT1), 0378h-037Ah (LPT2).
   Pierwszy z każdej grupy portów (port bazowy) jest portem danych - na niego wysyła się bajty do transmisji i z niego odczytuje się bajty odebrane przez port (w przypadku portów dwukierunkowych).
   Port baza+1 jest portem stanu. Jego najważniejsze bity to:
   - bit 7 - jeśli równy 1, to kontroler jest zajęty.
   - bit 6 - brak potwierdzenia
   - bit 5 - koniec papieru (drukarka)
   - bit 3 - żaden błąd nie wystąpił

   Port baza+2 jest portem kontroli. Nie zawiera żadnych interesujących nas bitów.

Dostęp przez DOS

W DOSie można oczywiście używać przedstawionych powyżej sposobów na dostęp do portów, ale jest też jeszcze jedna możliwość - zapis do plików specjalnych. DOS powinien utworzyć specjalne "urządzenia", których można używać tak samo jak plików, a same dane lądują nie w plikach, a w portach.
Te specjalne urządzenia mają nazwy "COM1", "COM2", "COM3", "COM4", "LPT1", "LPT2" (oczywiście nie wszystkie muszą być obecne na każdym systemie). Otwierając plik o nazwie "COM1" i zapisując do niego dane, tak naprawdę nie zapisujemy ich do żadnego pliku, a są one wysyłane do urządzenia przyłączonego do portu COM1. Dlatego czasem może być problem z utworzeniem realnego pliku o nazwie "COM1", a nawet jeśli da się go stworzyć, to zapisywane do niego dane mogą nie trafiać tam, gdzie byśmy tego oczekiwali.
Tych urządzeń można używać jak normalnych plików, a to opisałem w swoim kursie.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Zarządzanie zasilaniem komputera.

Jeśli zastanawialiście się kiedyś, jak wyłączać dyski twarde lub resetować komputer używając tylko oprogramowania (nie naciskając żadnych przycisków), to w tym artykule powinniście znaleźć odpowiedź na wszystkie wasze pytania.

Wyłączanie monitora

(przeskocz wyłączanie monitora)

Zajmijmy się najpierw teorią. Przerwanie, którym najpierw się zajmiemy, to.... int 10h, przerwanie sterowników karty graficznej. Co one mają wspólnego z energią? Otóż, można z użyciem int 10h wyłączyć monitor. Zajrzyjmy do RBIL, w opis funkcji numer 4F10h. Uruchomienie jej z BL=0 powinno nas upewnić, że odpowiednie funkcje są zainstalowane (zwróci AL=4Fh):
(przeskocz sprawdzanie funkcji)

```
mov     ax, 4f10h
mov     bl, 0
xor     di, di
mov     es, di          ; przerwanie żąda ES:DI = 0
int     10h

cmp     al, 4fh
jne     wychodzimy      ; gdy klęska....
```

Teraz, jeśli wiemy, że ta funkcja działa, to patrzymy na kolejną. Wpisując 1 do BL możemy zmienić aktualny stan zasilania. W BH podajemy, co chcemy zrobić: 0-włączyć, 1-przełączyć w stan oczekiwania, 2-zawiesić, 4-wyłączyć monitor. Zanim jednak zaczniecie ochoczo pisać, dam wam radę: program należy napisać tak, aby po jakimś czasie monitor wracał jednak do stanu włączonego (bez resetowania komputera...). Wiem, że potrafilibyście coś takiego sami napisać, ale podam tutaj gotowe (i sprawdzone - działa nawet pod Win98) rozwiązanie:
(przeskocz program wyłączający monitor)

```
; Program wyłącza monitor
;
; Autor: Bogdan D.
; kontakt: bogdandr (at) op (dot) pl
;
; nasm -O999 -o mon_off.com -f bin mon_off.asm

org 100h

start:
        mov ax,4f10h            ; wybranie numeru funkcji
        mov bx,0401h            ; BL=1 - zmień stan. BH=4 - wyłącz
        int 10h

        xor ah,ah
        int 16h                 ; poczekaj na naciśnięcie klawisza

        mov ax,4f10h
        mov bx,1                ; BL=1 - zmień stan. BH=0 - włącz
```

```
        int 10h

        mov ax,4c00h
        int 21h
```

Wyłączanie twardych dysków

(przeskocz wyłączanie twardych dysków)
Z dyskami twardymi jest nieco gorzej. Tutaj trzeba się znać na kontrolerze HDD - na jego portach i wysyłanych tam komendach. Dlatego posłużę się gotowcem. Ma on wyłączyć 2 pierwsze dyski twarde. Dla dokładniejszych opisów portów i ich komend spójrzcie do pliku ports.lst dołączonego do RBIL. Gotowiec ten jest częścią doskonałego programu FDAPM (FreeDOS Advanced Power Management), który wraz z kodem źródłowym można znaleźć na stronach projektu FreeDOS.
(przeskocz program wyłączający dyski)

```
        mov dx,1f6h     ; pierwszy kontroler IDE (drugi: 176)

        mov al,0a0h     ; bez LBA, dysk nadrzędny (Master)
        out dx,al
        inc dx          ; DX = 1F7
        call miniWait   ; chwila przerwy
        mov al,0e0h     ; e0 = standby, e1 = włączony/idle
        out dx,al
        dec dx          ; DX = 1F6
        call miniWait
        mov al,0b0h     ; bez LBA, dysk podrzędny (Slave)
        out dx,al
        inc dx          ; DX = 1F7
        call miniWait
        mov al,0e0h     ; e0 = standby, e1 = włączony/idle
        out dx,al

        mov ax,4c00h
        int 21h

        miniWait:       ; bardzo krótki okres przerwy
        xchg ax, bx
        xchg bx, ax
        xchg ax, bx
        xchg bx, ax
        ret
```

Po zatrzymaniu twardego dysku można go uruchomić wykonując dowolną operację na systemie plików (na przykład wyświetlić zawartość bieżącego katalogu).

Parkowanie głowic twardego dysku jest sprawą prostszą, gdyż w tym przypadku pomaga nam BIOS. Aby zaparkować głowice pierwszego dysku twardego, użyj następującego kodu:

```
        mov     ah, 19h
        mov     dl, 80h
        int     13h

        jc      blad            ; nie pokazuje błędów pod Windows 98
```

Jeśli chcecie zaparkować głowice drugiego dysku, zamiast 80h wpiszcie 81h, jeśli trzeciego - 82h itd.

Resetowanie i wyłączanie komputera

Teraz ciekawsze sprawy - resetowanie komputera lub wyłączanie go. Na początek grzecznie posłużymy się przerwaniem - będzie to int15h, numery funkcji 5300h i 5307h (po szczegółowe opisy tych funkcji posyłam oczywiście do RBIL). Najpierw sprawdźmy w ogóle, czy Advanced Power Management (APM) - bo o nim mowa - jest zainstalowane:
(przeskocz sprawdzanie APM)

```
mov     ax, 5300h
xor     bx, bx          ; numer urządzenia = 0 = BIOS
int     15h

jc      niestety   ; gdy coś poszło nie tak (np. brak APM), to CF=1
```

Teraz spróbujmy wyłączyć system:
(przeskocz wyłączanie zasilania)

```
mov     ax, 5307h       ; funkcja APM
mov     cx, 3           ; wyłącz system. CX=2 - zawieś system, CX=1 -
                        ; przełącz system w stan oczekiwania stand-by
mov     bx, 1           ; wszystkie urządzenia
int     15h             ; spróbujemy wyłączyć...
```

Jeśli istnieje możliwość wyłączenia prądu w systemie, to powyższy kod powinien to załatwić.

Teraz przejdziemy do innych grzecznych sposobów na zresetowanie komputera. W RBIL znalazłem:
(przeskocz opis przerwań do resetowania)

```
INT 16 - AMI BIOS - BIOS-FLASH Interface - GENERATE CPU RESET
        AX = E0FFh

INT 14 - FOSSIL - REBOOT SYSTEM
AH = 17h
AL = method
    00h = cold boot
    01h = warm boot
```

Jak widać, nie wygląda to skomplikowanie. Niestety, żaden z powyższych sposobów nie działa u mnie pod czystym DOS-em, a pod Windows98 działa jedynie sposób z APM (int15h).

A teraz pokażę kilka niegrzecznych (ale za to sprawdzonych przeze mnie i działających bez pudła) sposobów na zresetowanie komputera.
Pierwszym takim sposobem jest długi skok pod adres FFFF:0000 (tam znajduje się część BIOSu odpowiedzialna za operacje wykonywane przy starcie komputera). Wcześniej do segmentu danych BIOSu (segment 40h), pod adres 72h należy wpisać 0, gdy chcemy zimny reset (taki, co obejmuje testy pamięci i wszystko inne), a 1234h, gdy chcemy gorący reset.
Odpowiednie kawałki kodu wyglądają tak (przypominam, że adres 0040h:0072h = 0000:0472h - patrz część 2 mojego kursu):
(przeskocz kod do ręcznego resetowania)

```
; zimny reset:

mov     ax, 40h
mov     ds, ax                  ; DS = 40h
mov     word [ds:72h], 0        ; zimny reset

; niektóre kompilatory (np. TASM) nie lubią instrukcji w stylu
; jmp 0FFFFh:0000h, więc zakoduję ją ręcznie
db      0eah            ; kod instrukcji wzięty z podręczników Intela
dw      0               ; offset
dw      0ffffh          ; segment


; gorący reset:

xor     ax, ax
mov     ds, ax                  ; DS = 0
mov     word [ds:472h], 1234h   ; gorący reset

db      0eah            ; kod instrukcji wzięty z podręczników Intela
dw      0               ; offset
dw      0ffffh          ; segment
```

Drugim (i prostszym) sposobem jest zapisanie do jednego z portów klawiatury (64h) jednego z bajtów F0-FE, który ma bit0 = 0 (jest takich oczywiście kilka, najczęściej stosuje się FEh), chociaż ten sposób nie jest zalecany.
Kod jest wyjątkowo prosty i wygląda tak:

```
mov     al, 0feh
out     64h, al
```

Celowo nie wspominam tutaj o jednym: o przerwaniu int19h, które służy do ponownego przeczytania bootsektorów i przeładowania systemu od nowa. Gdy wkładacie niesystemową dyskietkę do stacji i resetujecie komputer, to (o ile macie możliwość uruchomienia systemu z dyskietki) pojawia się napis informujący o nieprawidłowym dysku systemowym. Po naciśnięciu Entera uruchamiane jest właśnie int19h, które nie wykonuje żadnych resetów, tylko czyta bootsektory od nowa.
Nie wspominałem o int19h, gdyż jest ono niebezpieczne. Jeżeli jakikolwiek program przejął przerwanie np. zegara, to int19h nie przywróci poprzedniej procedury, co jest nieprzewidywalne w skutkach!

Sposoby na wyłączanie urządzeń mogą się Wam przydać, gdy np. będziecie pisać własny wygaszacz ekranu, a możliwość zresetowania komputera przyda się, gdy Wasze oprogramowanie zostanie zainstalowane i musi zmienić np. zawartość pliku autoexec.bat.
Informacje, które tutaj podałem mogą się Wam też przydać przy pisaniu boot-sektorów do własnych mini-systemów operacyjnych.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Programowanie głośniczka w asemblerze

Czy nie myślicie czasem, jakby to było, gdyby można było wzbogacić swój program oprócz efektu wizualnego, także o efekt dźwiękowy?
Programowanie kart dźwiękowych (zwłaszcza tych nowoczesnych) może sprawiać niemałe kłopoty. Stary, poczciwy PC-Speaker jest jednak urządzeniem względnie prostym w programowaniu.
I to właśnie tutaj udowodnię. Najpierw troszkę teorii, potem - do dzieła!

Sporo urządzeń w komputerze ma własne porty, przez które można się z nimi komunikować. Jednak głośniczek komputerowy nie ma własnego portu.
Jest tak przede wszystkim ze względu na oszczędności w budowie pierwszych PC-tów. Zamiast dać osobny port na głośnik, firmy produkujące komputery wcisnęły go pod opiekę dwóch innych urządzeń:

- czasomierza systemowego, który posłuży nam do wytworzenia impulsów odpowiedniej częstotliwości
- kontrolera klawiatury, który kontroluje, czy jest otwarty kanał z czasomierza do głośniczka, czyli czy można będzie wysyłać informacje.

Podstawowe porty czasomierza to porty od 40h do 43h (cały zakres to 40h - 5fh, h oznacza szesnastkowo), kontrolera klawiatury zaś - 60h do 64h (cały zakres: 60h - 6fh).
Nie będziemy ich jednak wszystkich używać. Będą na interesować tylko porty 42h, 43h i 61h.

Zacznijmy więc coś pisać:

```
in al,61h
or al,3
out 61h,al
```

Co zrobiliśmy? W spisie portów Listy Przerwań Ralfa Brown'a czytamy:

(przeskocz port 61h)

```
0061  R-  KB controller port B control register (ISA, EISA)
0061  -W  KB controller port B (ISA, EISA)

(R - czytanie (read) , W - pisanie (write))
```

oraz:
(przeskocz opis portu 61h)

```
Bitfields for KB controller port B (system control port) [output]:
Bit(s)  Description     (Table P0392)
 7      pulse to 1 for IRQ1 reset (PC,XT)
 6-4    reserved
 3      I/O channel parity check disable
 2      RAM parity check disable
 1      speaker data enable
 0      timer 2 gate to speaker enable
```

Komenda `IN AL,61h` czyta bieżący status kontrolera, `OR AL,3` ustawia (włącza) bity 0 (włączenie bramki do głośniczka) oraz 1 (włączenie możliwości wysyłania danych do głośniczka), `OUT 61h,AL` zapisuje nowy status do kontrolera.

Głośniczek jest włączony. Trzeba mu podać jakiś sygnał. Do tego posłuży nam czasomierz. W spisie portów czytamy:

(przeskocz opis portów 42h i 43h)

```
0042  RW  PIT  counter 2, cassette & speaker

0043  RW  PIT  mode port, control word register for counters 0-2
    Once a control word has been written (43h), it must be followed
    immediately by performing the corresponding action to the counter
    registers (40h-42h), else the system may hang!!
```

Do portów tych nie będziemy wysyłać jednak częstotliwości, którą chcemy uzyskać. Czasomierz pracuje na częstotliwości 1193181 (1234DDh) Hz i to tę wartość dzielimy przez żądaną częstotliwość, a wynik wysyłamy do odpowiednich portów.

Piszmy więc:
(przeskocz włączanie głośniczka)

```
mov bx,440h     ; Standardowy dźwięk A, 440 Hz
mov dx,12h      ; górna część liczby 1234dd
mov ax,34ddh    ; dolna część liczby 1234dd
div bx          ; ax = wartość do wysłania

push ax
mov al,0b6h
out 43h,al

pop ax
out 42h,al
mov al,ah
out 42,al
```

No i co my tutaj znowu zrobiliśmy?
4 pierwsze komendy to oczywiście uzyskanie wartości do wysłania na port, ale reszta?
Najpierw: 0b6h = 1011 0110

Bity 7 i 6 = 10 = wybierz (standardowo niezajęty) czasomierz nr 2 (łącznie są 3: zegar czasu rzeczywistego, czasomierz odświeżania pamięci RAM i ten trzeci, nieużywany)
Bity 5 i 4 = 11 = zapisujemy do czasomierza najpierw młodsze bity (0-7) wartości, potem starsze (8-15)
Bity 3-1 = 011 = wybierz tryb nr 3, czyli generator fali kwadratowej
Bit 0 = 0 = licznik binarny 16-bitowy.

Zgodnie z tym, najpierw wysyłamy młodszy bajt, AL a potem starszy, AH.

Skoro na port można wysłać największą wartość 0ffffh (teoretycznie największa jest 10000h, obcinana do 0000h), to jakiej odpowiada to częstotliwości?
1234dd / 10000h to ok. 12h, czyli 18. A dokładniej jest to coś około 18,2 Hz - standardowa częstotliwość zegara w komputerze (aby odmierzyć 1 sekundę trzeba ok 18 tyknięć tego zegara)

Nasz głośniczek już gra. Teraz trzeba sprawić, bo to troszkę potrwało. Pomocne będzie przerwanie 15h, funkcja 86h:

```
mov cx,0fh
mov dx,4240h
mov ah,86h
int 15h                 ; pauza o długości CX:DX mikrosekund
```

I dźwięk trwa 1 sekundę (F4240h = 1.000.000). Teraz trzeba go wyłączyć. Nic prostszego. Po prostu zamkniemy przejście między czasomierzem a głośniczkiem:

```
in al,61h
and al,not 3            ; zerujemy bity 0 i 1
                        ; NASM:  "and al,~3"
out 61h,al
```

Mam nadzieję, że podałem wystarczająco informacji, abyście samodzielnie zaczęli programować głośniczek. Jeśli mi się nie udało, to zawsze możecie skorzystać z gotowej procedury z mojej biblioteki.

To już koniec. Miłej zabawy!

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Pisanie plików .SYS

Sterowniki w postaci plików .SYS dzielą się na 2 rodzaje:

- Nie-DOS-owe pliki .SYS (ładowane z config.sys poleceniem DEVICE=...) zazwyczaj zawierają sterowniki takich urządzeń zewnętrznych jak np. CD-ROM.
- Pliki .SYS systemu DOS - np. MSDOS.SYS czy IO.SYS. Te pliki zawierają sterowniki urządzeń standardowych, jak konsola CON czy drukarka PRN.

Wszystkie te pliki łączy wspólna struktura, którą postaram się tutaj przedstawić. Informacje podane przeze mnie są wycinkiem z dokumentu Programmer's Technical Reference for MSDOS and the IBM PC, którego kopię można znaleźć na stronach systemu O3one (720 kB).

Pliki .SYS zaczynają się od adresu 0 (org 0), a nagłówek takiego pliku składa się z pięciu elementów:
(przeskocz elementy nagłówka)

- DWORD - pełny adres (najpierw offset, potem segment) do następnego takiego nagłówka jak ten, jeśli nasz plik .SYS obsługuje więcej niż jedno urządzenie. Jeśli mamy tylko jeden sterownik w naszym pliku, wpisujemy tutaj wartość -1, czyli FFFF:FFFF.
- WORD - atrybut urządzenia (opisany dalej)
- WORD - offset procedury strategii danego sterownika (opisane dalej)
- WORD - offset procedury przerwania danego sterownika (opisane dalej)
- 8 bajtów - nazwa (urządzenie znakowe) dopełniana w razie potrzeby spacjami do ośmiu znaków lub ilość jednostek (urządzenie blokowe)

Urządzenie znakowe to takie, które może wysyłać/odbierać pojedyncze bajty, np. CON, PRN, AUX. Można je otwierać jak normalne pliki.
Urządzenie blokowe to takie, które operują na blokach danych i są to zazwyczaj dyski.

(przeskocz tabelę atrybutów urządzenia)

Bity atrybutu i ich znaczenie

Numer bitu	Znaczenie
0	=0 - to urządzenie nie jest standardowym urządzeniem wejścia =1 - to urządzenie jest standardowym urządzeniem wejścia
1	=0 - to urządzenie nie jest standardowym urządzeniem wyjścia =1 - to urządzenie jest standardowym urządzeniem wyjścia
2	=0 - to urządzenie nie jest urządzeniem NUL =1 - to urządzenie jest urządzeniem NUL
3	=0 - to urządzenie nie jest urządzeniem CLOCK =1 - to urządzenie jest urządzeniem CLOCK
4	=0 - należy używać standardowych procedur we/wy CON =1 - należy używać szybkich procedur we/wy ekranu (int 29h)
5-10	zarezerwowane, muszą być równe 0
11	

	=0 - to urządzenie nie obsługuje wymiennych nośników (domyślne dla DOS 2.x) =1 - to urządzenie obsługuje wymienne nośniki (tylko dla DOS 3.0+)
12	zarezerwowane, musi być równy 0
13	=0 - format IBM (urządzenia blokowe) =1 - format nie-IBM (urządzenia blokowe) =1 - obsługuje funkcję zapisywania danych aż do stanu zajętości (output till busy, urządzenia znakowe)
14	=0 - nie obsługuje IOCTL =1 - obsługuje IOCTL
15	=0 - urządzenie blokowe =1 - urządzenie znakowe

Ostatnie pole w nagłówku to nazwa urządzenia (w przypadku urządzeń znakowych) lub ilość jednostek/dysków obsługiwanych przez ten sterownik (urządzenia blokowe).

Procedura strategii (strategy routine).

(przeskocz procedurę strategii)
Za każdym razem, jak DOS chce coś od naszego sterownika, uruchamia procedurę strategii, podając w parze rejestrów ES:BX adres nagłówka żądania (request header). Zawiera on informacje o tym, co mamy zrobić. Jedynym obowiązkowym zadaniem tej procedury jest zachowanie adresu z ES:BX w zmiennej lokalnej, aby można było potem odczytywać żądania w procedurze przerwania, która uruchamiana jest zaraz po procedurze strategii. Jeśli chcemy zrobić coś więcej, musimy zachować wszystkie rejestry (łącznie z flagami), które zmieniamy.
Procedura kończy się wywołaniem RETF, gdyż DOS uruchamia nasz sterownik wykonując CALL FAR.
Tak więc najprostszy przykład sprowadza się do:

```
mov     word cs:[nagl_zad], bx          ; NASM : [cs:nagl_zad]
mov     word cs:[nagl_zad+2], es        ; NASM : [cs:nagl_zad+2]
retf
```

Procedura przerwania (interrupt routine).

(przeskocz procedurę przerwania)
Ta procedura jest odpowiedzialna za wykonywanie poleceń od systemu. Polecenia te są zawarte w nagłówku żądania, który teraz omówię.
W procedurze przerwania również należy zachować wszystkie modyfikowane rejestry i wrócić do DOSa poleceniem RETF. Procedura przerwania jest uruchamiana przez DOS tuż po powrocie z procedury strategii, która musi zachować bieżący adres nagłówka żądania.

(przeskocz opis nagłówka żądania)
Nagłówek żądania

Odległość od początku	Długość	Zawartość

0	1 bajt	Długość w bajtach całego nagłówka i ewentualnych danych
1	1	Kod podjednostki w urządzeniach blokowych. Nieistotne dla urządzeń znakowych
2	1	Kod rozkazu
3	2	Status wykonania
5	8	zarezerwowane dla DOSa
0Ch	różna	Dane odpowiednie dla operacji

Kod podjednostki w urządzeniach blokowych jest istotny, gdy nasz sterownik obsługuje więcej niż 1 urządzenie.

(przeskocz listę rozkazów)

Kod rozkazu

Kod	Nazwa	Funkcja
0	- INIT	- Inicjalizacja sterownika. Używane tylko raz.
1	- MEDIA CHECK	- Sprawdzanie, czy zmieniono dysk od ostatniego sprawdzenia. Używane tylko w urządzeniach blokowych. Urządzenia znakowe nic nie robią.
2	- BUILD BPB	- Stworzenie nowego BIOS Parameter Block (BPB). Używane tylko w urządzeniach blokowych. Urządzenia znakowe nic nie robią.
3	- IOCTL INPUT	- Odczyt IOCTL. Uruchamiane tylko wtedy, gdy urządzenie ma ustawiony bit IOCTL.
4	- INPUT	- Odczyt danych.
5	- NONDESTRUCTIVE INPUT NO WAIT	- Odczyt danych.
6	- INPUT STATUS	- Stan odczytu
7	- INPUT FLUSH	- Opróżnienie kolejki wejściowej
8	- OUTPUT	- Zapis danych.
9	- OUTPUT	- Zapis danych z weryfikacją.
10	- OUTPUT STATUS	- Stan zapisu
11	- OUTPUT FLUSH	- Opróżnienie kolejki wyjściowej
12	- IOCTL OUTPUT	- Zapis IOCTL. Uruchamiane tylko wtedy, gdy urządzenie ma ustawiony bit IOCTL.
13	- DEVICE OPEN	- Uruchamiane tylko wtedy, gdy urządzenie ma ustawiony bit OPEN/CLOSE/RM.
14	- DEVICE CLOSE	- Uruchamiane tylko wtedy, gdy urządzenie ma ustawiony bit OPEN/CLOSE/RM.
15	- REMOVEABLE MEDIA	- Uruchamiane tylko wtedy, gdy urządzenie blokowe ma ustawiony bit OPEN/CLOSE/RM.
16	- OUTPUT UNTIL BUSY	- Uruchamiane tylko wtedy, gdy urządzenie znakowe ma ustawiony bit 13.

Najważniejsze rozkazy są opisane dalej.

(przeskocz listę wyników działania)
Status wykonania zadania

bit	Znaczenie
0-7	Kod błędu, gdy bit15 = 1
8	=1 oznacza Operacja zakończona
9	=1 oznacza Urządzenie zajęte
10-14	Zarezerwowane dla DOSa
15	=1 oznacza błąd

(przeskocz listę błędów sterownika)
Znaczenie numerów błędów

numer	Typ błędu
0	naruszenie ochrony przed zapisem
1	nieznana jednostka
2	urządzenie nie jest gotowe
3	nieznana komenda
4	błąd CRC
5	nieprawidłowa długość struktury żądania dostępu do dysku
6	błąd wyszukania (seek error)
7	nieznany nośnik
8	sektor nie znaleziony
9	koniec papieru w drukarce
10	błąd zapisu
11	błąd odczytu
12	błąd ogólny
13	zarezerwowane
14	zarezerwowane
15	nieprawidłowa zmiana dysku

Rozkazy

(przeskocz listę rozkazów sterownika)

- INIT.
  (przeskocz rozkaz init)
  ES:BX -> struktura zawierająca nagłówek żądania i dane. Ta struktura wygląda tak:
  Nagłówek żądania

Odległość od początku	Długość	Zawartość

0	13 bajtów	Nagłówek żądania
0Dh	1	Liczba jednostek w urządzeniach blokowych. Nieistotne dla urządzeń znakowych
0Eh?	4	Offset i segment końca kodu naszego sterownika. Mówi DOSowi, ile pamięci można zwolnić (wymieniony wcześniej dokument podaje tutaj offset 11h, który nie jest prawidłowy).
12h?	4	Wskaźnik na tablicę BPB (nieistotne dla urządzeń znakowych) / wskaźnik na resztę argumentów (wymieniony wcześniej dokument podaje tutaj offset 15h).
16h?	1	numer dysku (DOS 3.0+) (wymieniony wcześniej dokument podaje tutaj offset 19h).

W czasie inicjalizacji należy:

1. ustawić liczbę jednostek (tylko w urządzeniach blokowych). Wpisać 0, jeśli nie można uruchomić urządzenia.
2. ustawić wskaźnik na tablicę BPB (tylko w urządzeniach blokowych)
3. wykonać czynności inicjalizacyjne (np. modemów, drukarek)
4. ustawić adres końca rezydentnego kodu. Wstawić CS:0, jeśli nie można uruchomić urządzenia.
5. ustawić odpowiedni status w nagłówku żądania

- Odczyt/Zapis (funkcje: 3, 4, 8, 9, 12, 16).
(przeskocz rozkazy odczytu i zapisu)
ES:BX -> struktura zawierająca nagłówek żądania i dane. Ta struktura wygląda tak:
Nagłówek żądania

Odległość od początku	Długość	Zawartość
0	13 bajtów	Nagłówek żądania
0Dh	1	Bajt deskryptora nośnika z BPB (Media Descriptor Byte)
0Eh	4	Offset i segment bufora, z którego dane będą odczytywane/ do którego dane będą zapisywane.
12h	2	Ilość bajtów/sektorów do zapisania/odczytania.
14h	1	Początkowy numer sektora (tylko urządzenia blokowe). Nie ma znaczenia dla urządzeń znakowych.
16h	4	Offset i segment identyfikatora napędu (volume ID), gdy zwrócono kod błędu 0Fh.

W czasie tej operacji należy:

1. ustawić odpowiedni status w nagłówku żądania
2. wykonać zadanie
3. ustawić rzeczywistą liczbę przeniesionych bajtów/sektorów

- NONDESTRUCTIVE INPUT NO WAIT.
  (przeskocz rozkaz NONDESTRUCTIVE INPUT NO WAIT)
  Ten odczyt różni się od innych tym, że nie usuwa odczytanych danych z bufora.

  ES:BX -> struktura zawierająca nagłówek żądania i dane. Ta struktura wygląda tak:
  Nagłówek żądania

Odległość od początku	Długość	Zawartość
0	13 bajtów	Nagłówek żądania
0Dh	1	Bajt odczytany z urządzenia

  W czasie tej operacji należy:
  1. zwrócić bajt odczytany z urządzenia
  2. ustawić odpowiedni status w nagłówku żądania

- INPUT FLUSH
  (przeskocz rozkaz INPUT FLUSH)
  Wymuszenie wykonania wszystkich operacji odczytu, o których wie sterownik.
  ES:BX -> nagłówek żądania.

  W czasie tej operacji należy:
  1. ustawić odpowiedni status w nagłówku żądania

- OUTPUT FLUSH
  (przeskocz rozkaz OUTPUT FLUSH)
  Wymuszenie wykonania wszystkich operacji zapisu, o których wie sterownik.
  ES:BX -> nagłówek żądania.

  W czasie tej operacji należy:
  1. ustawić odpowiedni status w nagłówku żądania

Przykład

Składając razem powyższe informacje, napisałem taki oto przykładowy plik .SYS.
Jest to sterownik wymyślonego urządzenia znakowego MYSZKA1, który obsługuje tylko funkcję INIT (oczywiście) i pobieranie danych z urządzenia, które sprowadza się do zwrócenia starego znacznika EOF (1Ah).
Aby było widać, że mój sterownik się ładuje (dzięki linii DEVICE=... w config.sys), dorobiłem kod wyświetlający na ekranie informację o ładowaniu.
Resztę zobaczcie sami:
(przeskocz przykładowy kod)

```
; Przykład sterownika typu .SYS
; Autor: Bogdan D.
; kontakt: bogdandr (małpka) op (kropka) pl
```

```
;
; kompilacja:
; nasm -O999 -w+orphan-labels -o protosys.sys -f bin protosys.asm

dd      0FFFFFFFFh              ; wskaźnik na następny sterownik
                                ; -1, bo mamy tylko 1 urządzenie
dw      08000h                  ; atrybuty (urz. znakowe), output till busy (A000)
dw      strategia               ; adres procedury strategii
dw      przerwanie              ; adres procedury przerwania
db      "MYSZKA1 "      ; nazwa urządzenia (8 znaków, dopełniane spacjami)

przerwanie:
        pushf
        push    es
        push    bx
        push    ax

        les     bx, [cs:request_header] ; ES:BX -> nagłówek żądania
        mov     al, [es:bx + 2]         ; kod rozkazu

        test    al, al                  ; 0 = INIT
        jz      .init

        cmp     al, 4                   ; czy ktoś chce czytać dane?
        je      .czytanie

        cmp     al, 5
        je      .czytanie2

                                        ; innych żądań nie obsługujemy

.koniec_przer:
                                        ; słowo wyniku w [es:bx+3]

        mov     word [es:bx + 3], 100h  ; mówimy, że wszystko zrobione

        pop     ax
        pop     bx
        pop     es
        popf

        retf

.init:
                                ; podajemy adres końca kodu, który ma
                                ; zostać w pamięci
                                ; można usunąć niepotrzebny już kod
        mov     word [es:bx + 0eh], koniec
        mov     [es:bx + 10h], cs
        pusha
        push    es

        mov     ah, 3                   ; pobranie aktualnej pozycji kursora
        xor     bx, bx
        int     10h                     ; DH, DL - wiersz, kolumna kursora

        inc     dh
        xor     dl, dl                  ; idziemy o 1 wiersz niżej,
                                        ; od lewej krawędzi
        push    cs
        mov     ax, 1301h               ; AH=funkcja pisania na ekran.
                                        ; AL=przesuwaj kursor
```

```
        mov     bx, 7                   ; normalne znaki (szary na czarnym)
        mov     cx, init1_dl            ; długość napisu
        mov     bp, init1               ; adres napisu
        pop     es                      ; segment napisu = CS
        int     10h                     ; napis na ekran.
                                        ; DH, DL wskazują pozycję.
        pop     es
        popa

        jmp     short .koniec_przer

.czytanie:                      ; jak ktoś chce czytać, zwracamy mu EOF
        push    es
        push    ax
        push    cx
        push    di

        mov     cx, [es:bx + 12h]       ; liczba żądanych bajtów
        les     di, [es:bx + 0Eh]       ; adres czytania/zapisywania
        mov     al, 1Ah                 ; 1ah = EOF
        rep     stosb                   ; zapisujemy

        pop     di
        pop     cx
        pop     ax
        pop     es
        jmp     short .koniec_przer

.czytanie2:                     ; jak ktoś chce czytać, zwracamy mu EOF
        mov     byte [es:bx+0Dh], 1Ah
        jmp     short .koniec_przer

request_header  dd      0               ; wskaźnik na nagłówek żądania

strategia:

        pushf
        mov     [cs:request_header], bx ; zapisujemy adres nagłówka żądania
        mov     [cs:request_header+2], es

        cmp     byte [cs:pierwsze], 1
        jne     .nie_pisz

        mov     byte [cs:pierwsze], 0
        pusha
        push    es

        mov     ah, 3                   ; pobranie aktualnej pozycji kursora
        xor     bx, bx
        int     10h                     ; DH, DL - wiersz, kolumna kursora

        inc     dh
        xor     dl, dl                  ; idziemy o 1 wiersz niżej,
                                        ; od lewej krawędzi

        push    cs
        mov     ax, 1301h               ; AH=funkcja pisania na ekran.
                                        ; AL=przesuwaj kursor
        mov     bx, 7                   ; normalne znaki (szary na czarnym)
        mov     cx, info1_dl            ; długość napisu
        mov     bp, info1               ; adres napisu
        pop     es                      ; segment napisu = CS
```

```
        int     10h                     ; napis na ekran.
                                        ; DH, DL wskazują pozycję.
        pop     es
        popa

.nie_pisz:
        popf
        retf

info1           db      "*** Uruchamianie sterownika MYSZKA1...",10,13,10,13
info1_dl        equ     $ - info1
init1           db      "*** INIT", 13, 10, 13, 10
init1_dl        equ     $ - init1
pierwsze        db      1

; wszystko od tego miejsca zostanie wyrzucone z pamięci
koniec:
```

Jak widać, było tu o wiele więcej opisu niż samej roboty i wcale nie okazało się to takie straszne.

Aby zobaczyć, czy nasz sterownik rzeczywiście został załadowany i ile zajmuje miejsca w pamięci, należy wydać polecenie `mem /c/p`.

Miłej zabawy.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Bezpośredni dostęp do ekranu

Jeśli myślicie, że odpowiednie funkcje przerwań 10h i 21h są jedynym sposobem na to, aby napisać coś na ekranie, to ten kurs pokaże Wam, jak bardzo się mylicie.

Na ekran w trybie tekstowym składa się 80x25 = 2000 znaków. Nie oznacza to jednak 2000 bajtów, gdyż każdy znak zaopatrzony jest w pewną wartość (1 bajt) mówiącą o jego wyglądzie. Łącznie jest więc 2000 słów (word, 16 bitów = 2 bajty), czyli 4000 bajtów. Mało, w porównaniu z wielkością 1 segmentu (64kB). Te 4000 bajtów żyje sobie w pewnym segmencie pamięci - 0B800h (kolorowe karty graficzne) lub 0B000h (mono).

Struktura tego bloku nie jest skomplikowana i wygląda następująco:

b800:0000 - znak 1, w lewym górnym rogu
b800:0001 - atrybut znaku 1
b800:0002 - znak 2, znajdujący się o 1 pozycję w prawo od znaku 1
b800:0003 - atrybut znaku 2
....

Czym zaś jest atrybut?
Jest to bajt mówiący o kolorze danego znaku i kolorze tła dla tego znaku. Bity w tym bajcie oznaczają:
3-0 - kolor znaku (16 możliwości)
6-4 - kolor tła (8 możliwości)
7 - miganie znaku (jeśli nie działa, to oznacza, że mamy 16 kolorów tła zamiast 8)

Jeszcze tylko wystarczy omówić kolory odpowiadające poszczególnym bitom i możemy coś pisać.
Oto te kolory:
Czarny - 0, niebieski - 1, zielony - 2, błękitny - 3, czerwony - 4, różowy - 5, brązowy - 6, jasnoszary (ten standardowy) - 7, ciemnoszary - 8, jasnoniebieski - 9, jasnozielony - 10, jasnobłękitny - 11, jasnoczerwony - 12, jasnoróżowy - 13, żółty - 14, biały - 15.

To powinno mówić samo za siebie: chcemy biały znak na czarnym tle? Odpowiedni bajt = 0fh.
A może żółty znak na niebieskim tle? Bajt = 1eh.

Poniżej zamieszczam także programik, który szybko napisałem w celu przetestowania teorii tu przedstawionej (składnia NASM):
przeskocz przykładowy program

```
; nasm -O999 -o test.com -f bin test.asm

org 100h

        mov ax, 0b800h
        mov bx, cs
        mov es, ax              ; es = 0b800 = segment pamięci ekranu
        mov ds, bx              ; ds = cs
        xor di, di              ; pozycja docelowa = di = 0
        mov si, tekst           ; skąd brać bajty
        mov cx, dlugosc         ; ile bajtów brać
```

```
        rep movsb               ; przesuń CX bajtów z DS:SI do ES:DI

        xor ah, ah
        int 16h

        mov ax, 4c00h
        int 21h

tekst   db "T",1,"e",2,"k",3,"s",4,"t",5
        db " ",6,"w",7,"i",8,"e",9,"l",10,"o",11,"k",12,"o",13
        db "l",14,"o",15,"r",16,"o",27h,"w",38h,"y",49h

dlugosc equ $-tekst
```

Zastosowałem w nim stałą typu `equ`, aby nie zmieniać CX po każdorazowej nawet najdrobniejszej zmianie tekstu.

Jak widać, wpisywanie każdorazowo znaku z jego argumentem niekoniecznie sprawia przyjemność. Na szczęście z pomocą przychodzi nam BIOS, ale nie funkcja 0e przerwania 10h, lecz funkcja 13h tegoż przerwania (opis wycięty z Ralf Brown's Intterrupt List):

(przeskocz opis int 10h, ah=13h)

```
INT 10 - VIDEO - WRITE STRING (AT and later,EGA)
        AH = 13h
        AL = write mode
           bit 0: update cursor after writing
           bit 1: string contains alternating characters
                  and attributes
           bits 2-7: reserved (0)
        BH = page number
        BL = attribute if string contains only characters
        CX = number of characters in string
        DH,DL = row,column at which to start writing
        ES:BP -> string to write
```

I krótki przykładzik zastosowania (fragment kodu dla TASMa):
(przeskocz przykład zastosowania int 10h, ah=13h)

```
mov cx,cs
mov ax,1301h                    ; funkcja pisania ciągu znaków
mov es,cx                       ; es = cs
mov bx,j_czer                   ; atrybut (kolor)
mov cx,info1_dl                 ; długość ciągu
mov bp,offset info1             ; adres ciągu
mov dx,(11 shl 8) or (40 - (info1_dl shr 1))    ;wiersz+kolumna
int 10h                         ; piszemy napis

info1           db      "Informacja"
info1_dl        equ     $ - info1
```

Najwięcej wątpliwości może wzbudzać linia kodu, która zapisuje wartość do DX (wiersz i kolumnę ekranu). Do DH idzie oczywiście 11 (bo do DX idzie b=11 shl 8, czyli 0b00h). Napis `(info1_dl shr 1)` dzieli długość tekstu na 2, po czym tę wartość odejmujemy od 40. Po co?
Jak wiemy, ekran ma 80 znaków szerokości. A tutaj od 40 odejmujemy połowę długości tekstu, który chcemy wyświetlić. Uzyskamy więc w taki sposób efekt wyśrodkowania tekstu na ekranie. I to wszystko.

No dobrze, a co jeśli nie chcemy używać przerwań a i tak chcemy mieć tekst w wyznaczonej przez nas pozycji?
Trzeba wyliczyć odległość naszego miejsca od lewego górnego rogu ekranu. Jak nietrudno zgadnąć, wyraża się ona wzorem (gdy znamy współrzędne przed kompilacją):
wiersz*80 + kolumna
i to tę wartość umieszczamy w DI i wykonujemy rep movsb.
Gdy zaś współrzędne mogą się zmieniać lub zależą od użytkownika, to użyjemy następującej sztuczki (kolumna i wiersz to 2 zmienne po 16 bitów):
(przeskocz obliczanie adresu w pamięci ze współrzędnych)

```
        mov ax, [wiersz]
        mov bx, ax              ; BX = AX
        shl ax, 6               ; AX = AX*64
        shl bx, 4               ; BX = BX*16 = AX*16
        add ax, bx              ; AX = AX*64 + AX*16 = AX*80
        add ax, [kolumna]       ; AX = 80*wiersz + kolumna

        mov di, ax
        shl di, 1       ; DI mnożymy przez 2, bo są 2 bajty na pozycję
```

i też uzyskamy prawidłowy wynik. Odradzam stosowanie instrukcji (I)MUL, gdyż jest dość powolna.

Zajmiemy się teraz czymś jeszcze ciekawszym: rysowanie ramek na ekranie. Oto programik, który na ekranie narysuje 2 wypełnione prostokąty (jeden będzie wypełniony kolorem czarnym). Korzysta on z procedury, która napisałem specjalnie w tym celu. Oto ten programik:
(przeskocz program rysujący okienka z ramką)

```
; Rysowanie okienek z ramką
;
; Autor: Bogdan D.
;
; nasm -O999 -o ramki.com -f bin ramki.asm


org 100h

; ramki podwójne:

        mov     ah, 7
        xor     bx, bx
        xor     cx, cx
        mov     dx, 9
        mov     bp, 9
        call    rysuj_okienko

        mov     ah, 42h
        mov     bx, 10
        mov     cx, 10
        mov     dx, 20
        mov     bp, 16
        call    rysuj_okienko

        xor     ah, ah
        int     16h

        mov     ax, 4c00h
        int     21h
```

```
rysuj_okienko:

; wejście:
;
;  AH = atrybut znaku (kolor)
;  BX = kolumna lewego górnego rogu
;  CX = wiersz lewego górnego rogu
;  DX = kolumna prawego dolnego rogu
;  BP = wiersz prawego dolnego rogu
;
; wyjście:
;  nic


r_p     equ     0bah            ; prawa boczna
r_pg    equ     0bbh            ; prawa górna (narożnik)
r_pd    equ     0bch            ; prawa dolna

r_g     equ     0cdh            ; górna
r_d     equ     r_g             ; dolna

r_l     equ     r_p             ; lewa boczna
r_lg    equ     0c9h            ; lewa górna
r_ld    equ     0c8h            ; lewa dolna

spacja  equ     20h


        push    di
        push    si
        push    es
        push    ax

        mov     di, cx
        mov     si, cx
        shl     di, 6
        shl     si, 4
        add     di, si          ; DI = DI*80 = numer pierwszego wiersza * 80


        mov     si, 0b800h
        mov     es, si          ; ES = segment ekranu

        mov     si, di
        add     di, bx          ; DI = pozycja początku
        add     si, dx          ; SI = pozycja końca

        shl     di, 1           ; 2 bajty/element
        shl     si, 1

        mov     al, r_lg
        mov     [es:di], ax     ; rysujemy lewy górny narożnik

        add     di, 2

        mov     al, r_g         ; będziemy rysować górny brzeg

.rysuj_gore:

        cmp     di, si          ; dopóki DI < pozycja końcowa
        jae     .koniec_gora
```

```
        mov     [es:di], ax
        add     di, 2
        jmp     short .rysuj_gore

.koniec_gora:
        mov     al, r_pg
        mov     [es:di], ax     ; rysujemy prawy górny narożnik


.wnetrze:
        shr     di, 1

        add     di, 80          ; kolejny wiersz
        sub     di, dx          ; początek wiersza

        push    di

        mov     di, bp
        mov     si, bp
        shl     di, 6
        shl     si, 4
        add     si, di          ; SI = SI*80 = numer ostatniego wiersza * 80

        pop     di

        cmp     di, si          ; czy skończyliśmy?
        je      .koniec_wnetrze

        mov     si, di
        add     di, bx          ; DI = pozycja początku
        add     si, dx          ; SI = pozycja końca

        shl     di, 1           ; 2 bajty / element
        shl     si, 1

        mov     al, r_l
        mov     [es:di], ax     ; rysujemy lewy brzeg
        add     di, 2

        mov     al, spacja      ; wnętrze okienka wypełniamy spacjami
.rysuj_srodek:

        cmp     di, si          ; dopóki DI < pozycja końcowa
        jae     .koniec_srodek

        mov     [es:di], ax
        add     di, 2
        jmp     short .rysuj_srodek

.koniec_srodek:

        mov     al, r_p
        mov     [es:di], ax     ; rysujemy prawy brzeg

        jmp     short .wnetrze

.koniec_wnetrze:


        mov     di, bp
        mov     si, bp
```

```
        shl     di, 6
        shl     si, 4
        add     di, si          ; DI = DI*80


        mov     si, di
        add     di, bx          ; DI = pozycja początku w ostatnim wierszu
        add     si, dx          ; SI = pozycja końca w ostatnim wierszu

        shl     di, 1           ; 2 bajty / element
        shl     si, 1

        mov     al, r_ld
        mov     [es:di], ax     ; rysujemy lewy dolny narożnik

        add     di, 2

        mov     al, r_d         ; będziemy rysować dolny brzeg

.rysuj_dol:

        cmp     di, si          ; dopóki DI < pozycja końcowa
        jae     .koniec_dol

        mov     [es:di], ax
        add     di, 2
        jmp     short .rysuj_dol

.koniec_dol:
        mov     al, r_pd
        mov     [es:di], ax     ; rysujemy prawy dolny narożnik


        pop     ax
        pop     es
        pop     si
        pop     di

        ret
```

Program nie jest skomplikowany, a komentarze powinny rozwiać wszystkie wątpliwości. Nie będę więc szczegółowo omawiał, co każda linijka robi, skupię się jednak na kilku sprawach:

- Oddzielanie instrukcji od jej argumentów tabulatorem
  Poprawia to nieco czytelność kodu.

- Kropki przed etykietami
  Sprawiają, że te etykiety są lokalne dla tej procedury. Nie będą się mylić z takimi samymi etykietami umieszczonymi po innej etykiecie globalnej.

- Stosowanie `equ`
  Wygodniejsze niż wpisywanie ciągle tych samych bajtów w kilkunastu miejscach. Szybko umożliwiają przełączenie się np. na ramki pojedynczej długości.

- Nie używam MUL, gdyż jest za wolne (co prawda tutaj nie zrobiłoby to może ogromnej różnicy, ale gdzie indziej mogłoby).

- Umieszczenie w programie sposobu kompilacji
  Może oszczędzić innym dużego bólu głowy, którego by się nabawili, szukając kompilatora dla tego kodu.

- Napisanie, co procedura przyjmuje i co zwraca
  Bardzo ważne! Dzięki temu użytkownik wie, co ma wpisać do jakich rejestrów, co procedura zwraca i (ewentualnie) które rejestry modyfikuje (tego raczej należy unikać).

Jak widać, ręczne manipulowanie ekranem wcale nie musi być trudne, a jest wprost idealnym rozwiązaniem, jeśli zależy nam na szybkości i nie chcemy używać powolnych przerwań.

Miłego eksperymentowania!

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Pisanie programów rezydentnych (TSR-ów)

W tym mini-kursie zajmiemy się sposobem pisania TSR-ów, czyli programów, które po uruchomieniu i zakończeniu pozostają w pamięci (TSR = Terminate and Stay Residend).

Pierwsze pytanie, które się nasuwa, brzmi: Po co to komu?
Główną przyczyną jest to, że chcemy coś robić w tle, czyli pozwalając użytkownikowi uruchamianie innych programów.
A co chcielibyśmy robić w tle?
No cóż, DOS-owe sterowniki (które też są TSR-ami) zajmują się wieloma sprawami, np. zarządzają pamięcią (jak EMM386.EXE), kontrolują CD-ROMy czy karty dźwiękowe.

Skoro już wiemy po co, to przyszła pora, aby dowiedzieć się, jak pisać takie programy.
Otóż, jak się okazuje, nie jest to wcale takie trudne. Spójrzmy, co oferuje nam Lista Przerwań Ralfa Brown'a (RBIL):
(przeskocz opis int 21h, ah=31h)

```
INT 21 - DOS 2+ - TERMINATE AND STAY RESIDENT
        AH = 31h
        AL = return code
        DX = number of paragraphs to keep resident
Return: never
Notes:  the value in DX only affects the memory block containing the
         PSP; additional memory allocated via AH=48h is not affected
        the minimum number of paragraphs which will remain resident
         is 11h for DOS 2.x and 06h for DOS 3.0+
        most TSRs can save some memory by releasing their environment
         block before terminating (see #01378 at AH=26h,AH=49h)
        any open files remain open, so one should close any files
         which will not be used before going resident; to access a
         file which is left open from the TSR, one must switch PSP
         segments first (see AH=50h)
```

(1 paragraf = 16 bajtów).
Jak widać, trzeba będzie zadbać o kilka spraw:

1. zamknięcie ewentualnych otwartych plików.
2. zwolnienie nieużywanej pamięci W zwolnieniu pamięci pomoże nam funkcja:
   (przeskocz opis int 21h, ah=49h)

   ```
   INT 21 - DOS 2+ - FREE MEMORY
           AH = 49h
           ES = segment of block to free
   Return: CF clear if successful
           CF set on error
           AX = error code (07h,09h)
   ```

   Jeśli uruchamiamy program typu .com, to DOS domyślnie przydziela mu całą dostępną pamięć. Będziemy zwalniać segment środowiska, adres którego znajdziemy pod ds:[2ch]. DOS sam zwolni pamięć przydzieloną naszemu programowi po jego zakończeniu. Jak wiemy, programy typu .com wczytywane są pod adres 100h w danym segmencie, a wcześniej jest PSP (Program Segment Prefix),

który zawiera między innymi linię poleceń (od offsetu 80h).
W programach typu .exe (wczytywanych zwykle pod adresem 0), DS pokazuje po prostu wcześniej niż CS (zazwyczaj DS = CS - 10h, czyli dodatkowe 10h*10h = 100h bajtów jest przed kodem).

3. jeśli nasz TSR przejmuje jakieś przerwanie (zazwyczaj tak właśnie będzie, bo po co pisać TSR, którego nie będzie można w żaden sposób uruchomić?), należy w swojej procedurze obsługi przerwania (Interrupt Service Routine - ISR) uruchomić starą ISR. Oprócz tego, po odinstalowaniu naszego TSR trzeba przywrócić adres starej ISR. Nie muszę chyba mówić, co by się stało, gdyby procesor chciał wykonać instrukcje pod adresem, pod którym nie wiadomo co się znajduje.

4. należy sprawdzić linię poleceń, z jaką uruchomiono nasz program (powiedzmy, że jeśli nic tam nie ma, to użytkownik chce zainstalować nasz program w pamięci, zaś jeśli jest tam literka u lub U, to użytkownik chce odinstalować nasz program).

Niestety, nie mam pod ręką lepszych własnych przykładów niż ten oto programik (też mój, oczywiście). Teoretycznie, w czasie dostępu do dysku twardego powinien włączyć diodę Scroll Lock na klawiaturze. Uruchamiać należy go oczywiście pod czystym DOSem. Może nie zawsze działać, ale są w nim elementy, które chciałbym omówić. Składnia dla kompilatora NASM.
(przeskocz przykładowy program)

```
; Pomysł polega na tym, aby w czasie dostępu do dysku twardego zapalać diodę
; Scroll Lock na klawiaturze.
;
; Autor: Bogdan D.
;
; nasm -O999 -o scrlck.com -f bin scrlck.asm
;
; z użyciem int 13h

; TASM:
; .model tiny
; .code

org 100h

start:
        jmp     kod


; to jest kod naszej procedury int 13h.
; Zostanie on w pamięci.

znacznik        db      "ECA135"
flagi           db      0

moje13h:
        pushf
        or      dl,dl   ; jeśli nie dysk twardy (bit7 = 0) to nie ma nas tu
        js      dysk_ok

to_nie_my:
        popf
        db 0eah                         ; długi skok do stare13h
        stare13h dd 4ch
```

```
dysk_ok:                ; sprawdzamy, którą komendę chce wykonać użytkownik

        test    al,al                   ; reset
        je      to_my

        cmp     ah,2                    ; czytaj
        je      to_my

        cmp     ah,3                    ; pisz
        je      to_my

;       cmp     ah,5                    ; formatuj
;       je      to_my

;       cmp     ah,6                    ; formatuj
;       je      to_my

;       cmp     ah,7                    ; formatuj
;       je      to_my

        cmp     ah,0ah                  ; czytaj
        je      to_my

        cmp     ah,0bh                  ; pisz
        je      to_my

        cmp     ah,0ch                  ; szukaj
        je      to_my

        cmp     ah,0dh                  ; reset
        je      to_my

        cmp     ah,0eh                  ; czytaj bufor sektora
        je      to_my

        cmp     ah,0fh                  ; pisz bufor
        je      to_my

        cmp     ah,21h                  ; PS/1+ czytaj sektory
        je      to_my

        cmp     ah,22h                  ; PS/1+ zapisuj sektory
        jne     to_nie_my

to_my:
        push    ax

                ;bit 2 = CapsLk, bit 1 = NumLk, bit 0 = ScrlLk,
                ; reszta bitów musi być równa 0


        push    es
        xor     ax, ax
        mov     es, ax
; TASM:  mov    al, byte ptr es:[0417h]
        mov     al, [es:0417h]          ; 0040:0017 - BIOS Data Area,
                                        ; bajt stanu klawiatury
; TASM:  mov    cs:[flagi], al
        mov     [cs:flagi], al          ; zachowujemy w bajcie  flagi
        pop     es
```

```
        mov     al, 0edh
        out     60h, al
        mov     al, 1                   ; zapalamy ScrLck
        out     60h, al

        pop     ax

; TASM:  call dword ptr cs:[stare13h]
        call    dword [cs:stare13h]     ; pozwól, żeby stara procedura
                                        ; int 13h też zrobiła swoje
                                        ; flagi już są na stosie

        pushf
        push    ax

                                        ; sprawdzamy, które diody były
                                        ; wcześniej zapalone
                                        ; i zapalamy je ponownie

        xor     al, al
; TASM:  test   byte ptr cs:[flagi], 01000000b
        test    byte [cs:flagi], 01000000b
        jz      nie_caps
        or      al, 4

nie_caps:
; TASM:  test   byte ptr cs:[flagi], 00100000b
        test    byte [cs:flagi], 00100000b
        jz      nie_num
        or      al, 2

nie_num:
; TASM:  test   byte ptr cs:[flagi], 00010000b
        test    byte [cs:flagi], 00010000b
        jz      koniec
        or      al, 1

koniec:

; TASM:  mov    cs:[flagi], al
        mov     [cs:flagi], al
        mov     al, 0edh
        out     60h, al
; TASM:  mov    al, cs:[flagi]
        mov     al, [cs:flagi]
        out     60h, al                 ; zapalamy diody

        pop     ax
        popf

        iret                            ; Interrupt RETurn - wychodzimy


; początek właściwego kodu

kod:
        mov     ax, cs
        mov     ds, ax          ; DS = CS, na wszelki wypadek

        xor     bx, bx

        mov     si, 80h         ; ds:[80h] - ilość znaków w linii poleceń
```

```
        mov     al, [si]

        mov     es, bx          ; ES = 0

        or      al, al          ; ilość znaków=0? To idziemy się zainstalować
        jz      instaluj

petla:
        inc     si              ; SI = 81h, 82h, ...

        mov     al, [si]        ; sprawdzamy kolejny znak w linii poleceń

        cmp     al, 0dh
        jz      instaluj        ; Enter = koniec linii, więc instaluj

                                ; u lub  U oznacza, że trzeba odinstalować
        cmp     al, "u"
        je      dezinst

        cmp     al, "U"
        jne     petla

; odinstalowanie

dezinst:
; TASM:  mov    es, word ptr es:[13h*4 + 2]
        mov     es, [es:13h*4 + 2]      ; ES = segment procedury obsługi
                                        ; int 13h (może naszej)
; TASM:  mov    di, offset znacznik
        mov     di, znacznik
        mov     cx, 6
        mov     si, di
        repe    cmpsb                   ; sprawdzamy, czy nasz znacznik jest
                                        ; na swoim miejscu
        jne     niema                   ; jeśli nie ma, to nie możemy się
                                        ; odinstalować

        mov     es, bx                  ; ES = 0
; TASM:  mov    es, word ptr es:[13h*4]
        mov     bx, [es:13h*4]
; TASM:  cmp    bx, offset moje13h
        cmp     bx, moje13h             ; sprawdzamy, czy offsety aktualnego
                                        ; int13h i naszego sie zgadzają

        jnz     niema                   ; jeśli nie, to nie nasza procedura
                                        ; obsługuje int13h i nie możemy się
                                        ; odinstalować


; TASM:  mov    es, word ptr es:[13h*4 + 2]
        mov     es, [es:13h*4 + 2]      ; segment naszego TSRa
        mov     ah, 49h

        cli                             ; wyłączamy przerwania, bo coś przez
                                        ; przypadek mogłoby uruchomić int 13h,
                                        ; którego adres właśnie zmieniamy

        int     21h                     ; zwalniamy segment naszego rezydenta

        cli
                                        ; kopiujemy adres starej procedury
                                        ; int13h z powrotem do
```

```
                                        ; Tablicy Wektorów Przerwań
                                        ; (Interrupt Vector Table - IVT)

; TASM:  mov    ax, word ptr [stare13h]
        mov     ax, [stare13h]          ; AX=offset starej procedury int 13h
; TASM:  mov    bx, word ptr [stare13h+2]
        mov     bx, [stare13h+2]        ; BX=segment starej procedury int 13h

; TASM:  mov    word ptr es:[13h*4], ax
        mov     [es:13h*4], ax
; TASM:  mov    word ptr es:[13h*4+2], bx
        mov     [es:13h*4+2], bx
        sti

; TASM:  mov    dx, offset juz_niema
        mov     dx, juz_niema           ; informujemy użytkownika, że
                                        ; odinstalowaliśmy program
        mov     ah, 9
        int     21h

        mov     ax, 4c00h
        int     21h                     ; wyjście bez błędu

niema:                                  ; jeśli adresy procedur int13h się
                                        ; nie zgadzają lub nie ma naszego
                                        ; znacznika, to poinformuj, że nie
                                        ; można odinstalować

; TASM:  mov    dx, offset nie_ma
        mov     dx, nie_ma
        mov     ah, 9
        int     21h

        mov     ax, 4c01h
        int     21h                     ; wyjście z kodem błędu = 1

; zainstalowanie

instaluj:
; TASM:  mov    es, word ptr es:[13h*4 + 2]
        mov     es, [es:13h*4 + 2]      ; ES = segment procedury obsługi
                                        ; int 13h (może naszej)
; TASM:  mov    di, offset znacznik
        mov     di, znacznik
        mov     cx, 6
        mov     si, di
        repe    cmpsb                   ; sprawdzamy, czy nasz znacznik
                                        ; już jest w pamięci
        je      juzjest                 ; jeśli tak, to drugi raz nie
                                        ; będziemy się instalować

; TASM:  mov    es, word ptr cs:[2ch]
        mov     es, [cs:2ch]            ; segment środowiska
        mov     ah, 49h
        int     21h                     ; zwalniamy


        mov     es, bx                  ; ES = 0
; TASM:  mov    ax, word ptr es:[13h*4]
        mov     ax, [es:13h*4]          ; AX=offset starej procedury int 13h
; TASM:  mov    bx, word ptr es:[13h*4+2]
        mov     bx, [es:13h*4 + 2]      ; BX=segment starej procedury int 13h
```

```
                                        ; zachowujemy adres i segment:
; TASM:  mov    word ptr [stare13h], ax
        mov     [stare13h], ax
; TASM:  mov    word ptr [stare13h+2], bx
        mov     [stare13h+2], bx

                                        ; zapisujemy nowy adres i
                                        ; segment do IVT
        cli
; TASM:  mov    word ptr es:[13h*4], offset moje13h
        mov     word [es:13h*4], moje13h
; TASM:  mov    word ptr es:[13h*4 + 2], cs
        mov     [es:13h*4 + 2], cs
        sti

; TASM: mov     dx, offset zainst
        mov     dx, zainst              ; informujemy, że zainstalowano
        mov     ah, 9
        int     21h

; TASM:  mov    dx, offset kod
        mov     dx, kod
        mov     ax, 3100h
        shr     dx, 4                   ; DX=kod/16=ilość paragrafów do
                                        ; zachowania w pamięci
        inc     dx
        int     21h                     ; int 21h, AX = 3100h - TSR

juzjest:                                ; jeśli nasz program już jest w
                                        ; pamięci, to drugi raz się nie
                                        ; zainstalujemy
; TASM:  mov    dx, offset juz_jest
        mov     dx, juz_jest
        mov     ah, 9
        int     21h

        mov     ax, 4c02h
        int     21h                     ; wyjście z kodem błędu = 2


nie_ma          db "Programu nie ma w pamieci.$"
juz_niema       db "Program odinstalowano.$"
juz_jest        db "Program juz zainstalowany.$"
zainst          db "Program zainstalowano.$"

; TASM: end start
```

Teraz omówię kilka spraw, o które moglibyście zapytać:

- Zaraz po starcie jest skok do kodu. Dlaczego?
  Funkcja 31h przerwania 21h musi dostać informację, ile paragrafów (od miejsca, gdzie zaczyna się program) ma zachować w pamięci. Dlatego więc najpierw w programie zapisujemy kod rezydentny a potem resztę (instalacja / dezinstalacja), która nie będzie potem potrzebna w pamięci.

- Po co ten znacznik?
  Aby upewnić się przy próbie odinstalowania, że to rzeczywiście naszą procedurę chcemy odinstalować. Niedobrze byłoby, gdyby jakiś inny program potem przejął to przerwanie, a my byśmy go wyrzucili z pamięci...

Treść znacznika może oczywiście być dowolna.

- Czemu uruchomienie starej procedury jest w środku naszej (a nie na początku czy na końcu) i czemu jest postaci `call dword ...` ?
  Chodzi o to, aby najpierw zapalić Scroll Lock, potem wykonać operację na dysku (do czego posłuży nam prawdziwa procedura int13h) i na końcu przywrócić stan diód na klawiaturze. Użycie CALL a nie JMP spowoduje, że odzyskamy kontrolę po tym, jak uruchomimy stare przerwanie. Zaś adres starego przerwania to segment i offset, czyli razem 4 bajty (stąd: DWORD).

- Czemu wszędzie jest `CS:` ?
  Gdy jesteśmy w naszej procedurze, nie wiemy, ile wynosi DS. Wiemy, że CS pokazuje na naszą procedurę. Są więc 2 wyjścia:
  - ♦ Zachować DS na stosie, po czym zmienić go na nasz segment
  - ♦ Zamiast nieznanego DS, używać znanego CS

  Wybrałem to drugie.

- Gdzie się dowiedzieć, jak zapalać diody na klawiaturze?
  Instrukcje znajdują się w moim innym kursie. Polecam.

- Co robi instrukcja IRET ?
  Interrupt Return robi tyle, co zwykły RET, ale jeszcze zdejmuje flagi ze stosu. Polecam opis instrukcji INT z drugiej części mojego kursu.

- Co znajduje się pod `ds:[80h]` ?
  Liczba bajtów linii poleceń programu.

- Gdzie znajduje się linia poleceń programu?
  Od ds:[81h] maksymalnie do ds:[0ffh] (od ds:[100h] zwykle zaczyna się kod programu). Napotkanie Carriage Return (13 = 0Dh) po drodze oznacza koniec linii poleceń.

- Czemu w kodzie jest `[es:13h*4]` zamiast `[es:4ch]` ?
  Czytelniejsze, bo oznacza, że chcemy adres przerwania 13h.

- Czemu int 21h jest otoczone przez CLI ?
  Nie chciałem ryzykować, że w chwili zmiany adresu lub zwalniania pamięci rezydenta trafi się jakieś przerwanie, które mogłoby chcieć uruchomić int13h (którego już nie ma po tym int21h lub którego adres jest niespójny - zmieniliśmy już segment, ale jeszcze nie offset itp.).

- Czemu program sprawdza znacznik itp. przy dezinstalacji ?
  Głupio byłoby odinstalować nie swoją procedurę...
  Tym bardziej, że najbliższe int13h spowodowałoby nieprzewidywalne skutki.

- Czemu program sprawdza znacznik przy instalacji ?
  Nie chcę, aby program instalował się wielokrotnie, gdyż potem odzyskanie adresu starej procedury zajęłoby tyle samo dezinstalacji, co instalacji.

- Co znajduje się w `DS:[2ch]` ?
  Numer segmentu pamięci, w którym trzymane są zmienne środowiskowe (jak PATH, BLASTER, i wszystkie inne ustawiane komendą SET, np. w pliku autoexec.bat). Możemy go zwolnić, bo dla każdego programu tworzona jest oddzielna kopia.

- Paragraf to 16 bajtów, więc dzielimy DX przez 16. Ale czemu dodajemy 1?
  Jeżeli kod wystaje ponad adres podzielny przez 16, to część jego zostanie utracona. Procesor będzie wykonywał nieznane instrukcje z nieprzewidywalnym skutkiem.

Chociaż DOS jest już rzadko używany, to jednak umiejętność pisania TSR-ów może się przydać, np. jeśli chcemy oszukać jakiś program i podać mu np. większy/mniejszy rozmiar dysku lub coś innego. Można też napisać DOS-owy wygaszacz ekranu jako TSR, program który będzie wydawał dźwięki po naciśnięciu klawisza, wyświetlał czas w narożniku ekranu i wiele, wiele innych ciekawych programów. Nawet jeśli nikomu oprócz nas się nie przydadzą lub nie spodobają, to zawsze i tak zysk jest dla nas - nabieramy bezcennego doświadczenia i pisaniu i znajdowaniu błędów w programach rezydentnych. Takie umiejętności mogą naprawdę się przydać, a z pewnością nikomu nie zaszkodzą.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Uruchamianie innych programów

Czasem zdarza się, że z poziomu naszego własnego programu musimy uruchomić jakiś inny program lub polecenie systemowe. Służy do tego funkcja systemowa AH=4B przerwania DOS-a 21h. Jej argumenty to kolejno:

- w AL - typ uruchomienia. Najczęściej AL=0, czyli załaduj i uruchom
- w ES:BX - adres struktury dotyczącej środowiska uruchamianego programu. Pola struktury to kolejno:
  - (WORD) segment zawierający zmienne środowiska. Można wpisać 0 (wtedy będzie skopiowany nasz segment środowiska).
  - (DWORD) adres linii poleceń uruchamianego programu
  - (DWORD) adres pierwszego File Control Block (FCB) uruchamianego programu (nieużywane)
  - (DWORD) adres drugiego FCB uruchamianego programu (nieużywane)
- w DS:DX - adres nazwy uruchamianego programu

Po więcej szczegółów odsyłam do listy przerwań Ralfa Brown'a (RBIL)

Spróbujmy teraz napisać jakiś prosty przykład - uruchomienie samego NASMa (powinien się wyświetlić błąd, że nie podano plików wejściowych). Program jest w składni NASM.
(przeskocz przykładowy program)

```
; Program uruchamiający inny program.
;
; Autor: Bogdan D.
; kontakt: bogdandr (at) op (dot) pl
;
; nasm -O999 -o exec_dos.com -f bin exec_dos.asm

section .text
org     100h

start:
        mov     ax, cs
        mov     es, ax          ; będziemy zmieniać rozmiar segmentu kodu
        mov     bx, koniec      ; BX = rozmiar segmentu kodu
        shr     bx, 4           ; BX /= 16 - rozmiar w paragrafach
        inc     bx              ; żeby nie obciąć naszego programu
        mov     ah, 4ah         ; funkcja zmiany rozmiaru
        int     21h
                                ; zwolnienie miejsca na ładowany program

        mov     [kom_ln+2], cs  ; uzupełniamy pola potrzebnych struktur
        mov     [fcb1+2], cs
        mov     [fcb2+2], cs

        mov     [sssp], sp      ; zachowujemy nasz stos
        mov     [sssp+2], ss

        mov     ax, 4b00h       ; funkcja uruchomienia programu
        mov     dx, program     ; adres nazwy programu
        mov     bx, srod        ; adres struktury środowiska
        int     21h             ; uruchamiamy

        cli                     ; przywracamy nasz stos
```

```
        mov     sp, [sssp]
        mov     ss, [sssp+2]
        sti


        mov     ax, 4c00h
        int     21h

sssp            dd      0       ; miejsce na SS i SP

; linia poleceń uruchamianego programu
linia_kom       db      0, "    ", 0dh

; File Control Block - juz nieużywana przez DOS
; struktura, tu ustawiona na jakieś bezpieczne domyślne wartości
; (zgodnie z książką Art of Assembler)
fcb             db      3," ",0,0,0,0,0

; nazwa programu do uruchomienia
program         db      "nasm.exe", 0

; struktura środowiska
srod            dw      0               ; segment środowiska. Nasz własny
                                        ; jest pod DS:[2ch]
kom_ln          dw      linia_kom, 0    ; offset i segment linii poleceń
fcb1            dw      fcb, 0          ; offset i segment pierwszego FCB
fcb2            dw      fcb, 0          ; offset i segment drugiego FCB

koniec:
```

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Skankody i kody ASCII klawiszy

Informacje te pochodzą z Ralf Brown's Interrupt List oraz ze znakomitej książki Art of Assembly Language Programming (wersja dla DOS-a) autorstwa Randalla Hyde'a. Książkę można za darmo ściągnąć z Webstera

(przeskocz skankody)

Skankody (scan codes) wysyłane przez klawiaturę

Klawisz	Naciśnięcie	Zwolnienie	Kl	Nac	Zwol	Kl	Nac	Zwol	Kl	Nac	Zwol
Esc	01	81	] }	1B	9B	. >	34	B4	END	4F	CF
1 !	02	82	ENTER	1C	9C	/ ?	35	B5	DÓŁ	50	D0
2 @	03	83	Ctrl	1D	9D	PShift	36	B6	PGDN	51	D1
3 #	04	84	A	1E	9E	* (num)	37	B7	INS	52	D2
4 $	05	85	S	1F	9F	alt	38	B8	DEL	53	D3
5 %	06	86	D	20	A0	spacja	39	B9	SysRq	54	D4
6 ^	07	87	F	21	A1	CAPS	3A	BA	/ (num)	E0 35	B5
7 &	08	88	G	22	A2	F1	3B	BB	enter (num)	E0 1C	9C
8 *	09	89	H	23	A3	F2	3C	BC	F11	57	D7
9 (	0A	8A	J	24	A4	F3	3D	BD	F12	58	D8
0)	0B	8B	K	25	A5	F4	3E	BE	LWin	5B	DB
- _	0C	8C	L	26	A6	F5	3F	BF	PWin	5C	DC
+ =	0D	8D	; :	27	A7	F6	40	C0	Menu	5D	DD
BkSp	0E	8E	‘ "	28	A8	F7	41	C1	ins (num)	E0 52	D2
Tab	0F	8F	~ `	29	A9	F8	42	C2	del (num)	E0 53	D3
Q	10	90	LShift	2A	AA	F9	43	C3	home (num)	E0 47	C7
W	11	91	\ |	2B	AB	F10	44	C4	end (num)	E0 4F	CF
E	12	92	Z	2C	AC	NUM	45	C5	pgup (num)	E0 49	C9
R	13	93	X	2D	AD	SCRLCK	46	C6	pgdn (num)	E0 51	D1
T	14	94	C	2E	AE	HOME	47	47	lewo (num)	E0 4B	CB
Y	15	95	V	2F	AF	GÓRA	48	C8	prawo (num)	E0 4D	CD
U	16	96	B	30	B0	PGUP	49	C9	góra (num)	E0 48	C8
I	17	97	N	31	B1	- (num)	4A	CA	dół (num)	E0 50	D0
O	18	98	M	32	B2	5 (num)	4C	CC	Palt	E0 38	B8
P	19	99	LEWO	4B	CB	PRAWO	4D	CD	Pctrl	E0 1D	9D
[{	1A	9A	, <	33	B3	+ (num)	4E	CE	Pauza	E1 1D 45 E1 9D C5	(brak)

Na żółto, małymi literami i napisem num oznaczyłem klawisze znajdujące się (moim zdaniem) na klawiaturze numerycznej.

Kody ASCII klawiszy z modyfikatorami

Klawisz	Skankod	kod ASCII	z Shift	z Control	z Alt	z NumLock	z CapsLock	z Shift+CapsLock	z Shift+NumLock
Esc	01	1B	1B	1B	(brak)	1B	1B	1B	1B
1 !	02	31	21	(brak)	7800	31	31	31	31
2 @	03	32	40	0300	7900	32	32	32	32
3 #	04	33	23	(brak)	7A00	33	33	33	33
4 $	05	34	24	(brak)	7B00	34	34	34	34
5 %	06	35	25	(brak)	7C00	35	35	35	35
6 ^	07	36	5E	1E	7D00	36	36	36	36
7 &	08	37	26	(brak)	7E00	37	37	37	37
8 *	09	38	2a	(brak)	7F00	38	38	38	38
9 (	0A	39	28	(brak)	8000	39	39	39	39
0)	0B	30	29	(brak)	8100	30	30	30	30
- _	0C	2D	5F	1F	8200	2D	2D	5F	5F
+ =	0D	3D	2B	(brak)	8300	3D	3D	2B	2B
BkSp	0E	08	08	7F	(brak)	08	08	08	08
Tab	0F	09	0F00	(brak)	(brak)	09	09	0F00	0F00
Q	10	71	51	11	1000	71	51	71	51
W	11	77	57	17	1100	77	57	77	57
E	12	65	45	05	1200	65	45	65	45
R	13	72	52	12	1300	72	52	72	52
T	14	74	54	14	1400	74	54	74	54
Y	15	79	59	19	1500	79	59	79	59
U	16	75	55	15	1600	75	55	75	55
I	17	69	49	09	1700	69	49	69	49
O	18	6F	4F	0F	1800	6F	4F	6F	4F
P	19	70	50	10	1900	70	50	70	50
[{	1A	5B	7B	1B	(brak)	5B	5B	7B	7B
] }	1B	5D	7D	1D	(brak)	5D	5D	7D	7D
ENTER	1C	0D	0D	0A	(brak)	0D	0D	0A	0A
CTRL	1D	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)
A	1E	61	41	01	1E00	61	41	61	41
S	1F	73	53	13	1F00	73	53	73	53
D	20	64	44	04	2000	64	44	64	44
F	21	66	46	06	2100	66	46	66	46
G	22	67	47	07	2200	67	47	67	47
H	23	68	48	08	2300	68	48	68	48
J	24	6A	4A	0A	2400	6A	4A	6A	4A
K	25	6B	4B	0B	2500	6B	4B	6B	4B
L	26	6C	4C	0C	2600	6C	4C	6C	4C
; :	27	3B	3A	(brak)	(brak)	3B	3B	3A	3A
‘ "	28	27	22	(brak)	(brak)	27	27	22	22

~ `	29	60	7E	(brak)	(brak)	60	60	7E	7E
LShift	2A	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)
\ |	2B	5C	7C	1C	(brak)	5C	5C	7C	7C
Z	2C	7A	5A	1A	2C00	7A	5A	7A	5A
X	2D	78	58	18	2D00	78	58	78	58
C	2E	63	43	03	2E00	63	43	63	43
V	2F	76	56	16	2F00	76	56	76	56
B	30	62	42	02	3000	62	42	62	42
N	31	6E	4E	0E	3100	6E	4E	6E	4E
M	32	6D	4D	0D	3200	6D	4D	6D	4D
, <	33	2C	3C	(brak)	(brak)	2C	2C	3C	3C
. >	34	2E	3E	(brak)	(brak)	2E	2E	3E	3E
/ ?	35	2F	3F	(brak)	(brak)	2F	2F	3F	3F
PShift	36	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)
* (num)	37	2A	(brak?)	10	(brak)	2A	2A	(brak?)	(brak?)
alt	38	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)
spacja	39	20	20	20	(brak)	20	20	20	20
caps lock	3A	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)
F1	3B	3B00	5400	5E00	6800	3B00	3B00	5400	5400
F2	3C	3C00	5500	5F00	6900	3C00	3C00	5500	5500
F3	3D	3D00	5600	6000	6A00	3D00	3D00	5600	5600
F4	3E	3E00	5700	6100	6B00	3E00	3E00	5700	5700
F5	3F	3F00	5800	6200	6C00	3F00	3F00	5800	5800
F6	40	4000	5900	6300	6D00	4000	4000	5900	5900
F7	41	4100	5A00	6400	6E00	4100	4100	5A00	5A00
F8	42	4200	5B00	6500	6F00	4200	4200	5B00	5B00
F9	43	4300	5C00	6600	7000	4300	4300	5C00	5C00
F10	44	4400	5D00	6700	6100	4400	4400	5D00	5D00
num lock	45	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)
scroll lock	46	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)
home	47	4700	37	7700	(brak)	37	4700	37	4700
góra	48	4800	38	(brak)	(brak)	38	4800	38	4800
pgup	49	4900	39	8400	(brak)	39	4900	39	4900
- (num)	4A	2D	2D	(brak)	(brak)	2D	2D	2D	2D
lewo	4B	4B00	34	7300	(brak)	34	4B00	34	4B00
5 (num)	4C	4C00	35	(brak)	(brak)	35	4C00	35	4C00
prawo	4D	4D00	36	7400	(brak)	36	4D00	36	4D00
+ (num)	4E	2B	2B	(brak)	(brak)	2B	2B	2B	2B
end	4F	4F00	31	7500	(brak)	31	4F00	31	4F00

dół	50	5000	32	(brak)	(brak)	32	5000	32	5000
pgdn	51	5100	33	7600	(brak)	33	5100	33	5100
ins	52	5200	30	(brak)	(brak)	30	5200	30	5200
del	53	5300	2E	(brak)	(brak)	2E	5300	2E	5300

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

(przeskocz różnice składni)

Najważniejsze różnice między dyrektywami TASMa, FASMa, NASMa, MASMa i Watcom Asemblera

typ dyrektywy	NASM	FASM	TASM / MASM / WA
deklaracje danych	NASM: db, dw, dd, dq, dt	FASM: db, dw/du, dd, dp/df, dq, dt	TASM/MASM/WA: db, dw, dd, dp/df, dq, dt
rezerwacja pamięci	NASM: resb, resw, resd, resq, rest	FASM: rb, rw, rd, rp/rf, rq, rt	TASM/MASM/WA: db/dw/dd/dp/dq/dt ilość DUP(?)
operacje liczbowe i bitowe	NASM: +, -, *, /, %, |, ^, &, <<, >>, ~	FASM: +, -, *, /, mod, or, xor, and, shl, shr, not	TASM/MASM/WA: +, -, *, /, mod, or, xor, and, shl, shr, not
deklaracje stałych	NASM: %define, %idefine, %xdefine, %xidefine, equ	FASM: =, equ	TASM/MASM/WA: =, equ
etykiety anonimowe	NASM: tylko w trybie zgodności z TASMem	FASM: @@, @b/@r, @f	TASM/MASM/WA: @@, @b, @f
makra	NASM: %macro, %imacro nazwa ilość_arg ... %endm	FASM: macro nazwa arg {...}	TASM/MASM/WA: nazwa macro arg ... endm
kompilacja warunkowa	NASM: %if, %if(n)def, %elif, %else, %endif	FASM: if, else if, else, end if	TASM/MASM/WA: if***, elseif, else, endif
struktury	NASM: struc nazwa ... endstruc	FASM: struc nazwa {...}	TASM/MASM/WA: nazwa struc ... ends
symbole zewnętrzne	NASM: extern, global	FASM: extrn, public	TASM/MASM/WA: extrn, public
segmenty	NASM: segment nazwa	FASM: segment nazwa ; (format MZ)	TASM/MASM/WA: nazwa segment
dostępność instrukcji	NASM: wszystkie domyślnie dostępne dyrektywa CPU	FASM: wszystkie zawsze dostępne	TASM/MASM/WA: .8086, .186, .286, .386, .486, .586, .686, .mmx, .xmm
typowy początek programu .com	NASM: org 100h	FASM: format binary org 100h	TASM/MASM/WA: .model tiny .code

			org 100h start:
typowy początek programu .exe	NASM: ..start:	FASM: format MZ stack 400h entry kod:start segment kod start:	TASM/MASM/WA: .model small .stack 400h .code start:

Sposoby kompilacji w TASM, FASM, NASM, MASM i Watcom Asemblerze

typ programu	**NASM**	**FASM**	**TASM**	**MASM (16-bitowy)**	**WA**
.com	nasm -f bin -o prog.com prog.asm	fasm prog.asm prog.com	tasm prog.asm tlink /t prog.obj	ml prog.asm	wasm -fpi87 prog.asm wlink system dos com f prog.obj name prog.com
.exe	nasm -f obj -o prog.obj prog.asm val prog.obj,prog.exe,,,	fasm prog.asm prog.exe	tasm prog.asm tlink prog.obj	ml prog.asm	wasm -fpi87 prog.asm wlink system dos f prog.obj name prog.exe

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Odnośniki do innych stron:

1. (DOBRY) Spis przerwań Ralfa Browna (Ralf Brown's Interrupt List, RBIL)
   (przeskocz RBIL)
   Jeśli zaczynasz programować dla DOS-a (i nie chcesz na razie pisać aplikacji okienkowych dla Windowsa), to nie pożałujesz, jeśli ściągniesz! Zawiera opis wszystkich funkcji DOSa, BIOS-u, i wiele innych informacji. Bez tego ani rusz! Do ściągnięcia tu: RBIL

2. Kompilatory języka asembler:
   (przeskocz kompilatory)
   - (DOBRY) NASM (The Netwide Assembler - DOS, Windows, Linux, 16-bit, 32-bit, 64-bit) - prosty w obsłudze kompilator języka asembler z pełną dokumentacją: strona NASMa.

     W sam raz do pisania programów typu .COM. Do pisania programów .EXE potrzebować będziesz linkera. Polecam Alink (darmowy program służący za DPMI znajdziecie na stronach, z których można pobrać NASMa - nazywa się CWSDPMI) lub VAL

   - Napisany przez Polaka FASM (The Flat Assembler - DOS, Windows, Linux, 16-bit, 32-bit, 64-bit): strona FASMa
     Absolutnie fantastyczne narzędzie do pisania programów okienkowych! Żadnych zbędnych śmieci, nie potrzebujesz zewnętrznych linkerów, bibliotek, niczego. FASM ma to wszystko w załącznikach, a wersja GUI dla Windows to kompilator ze środowiskiem, całość tylko w 1 pliku .exe!
     Całkiem nieźle radzi sobie też w Linuksie.

   - YASM (DOS, Linux, Windows, 16-bit, 32-bit, 64-bit): strona YASMa
     Prawie całkowicie zgodny ze składniami NASMa i GNU asemblera.

   - Napisany przez Polaka SB86 (dawniej SASM) - DOS, Windows, Linux, 16-bit, 32-bit: sb86.way.to
     Składnia różni się nieco od innych - przypomina nieco język C, ale z instrukcji wynika, że kompilator ten ma całkiem duże możliwości.

   - LZASM (Lazy Assembler - DOS/Windows, zgodny z TASMem): lzasm.hotbox.ru

   - JWasm (DOS/Windows, 16-bit, 32-bit, zgodny z MASMem w wersji 6): japheth.de/JWasm.html

   - A86 (DOS, darmowy tylko 16-bit + debugger 16-bit): eji.com

   - MASM (Microsoft Macro Assembler - DOS/Windows, 16-bit, 32-bit): www.masm32.com oraz webster.cs.ucr.edu
     16-bitowy linker znajduje się na stronach Microsoft

   - HLA (High-Level Assembler - Windows/Linux, 32-bit): webster.cs.ucr.edu

   - Jeremy Gordon's GoAsm + dobry debugger 32-bit GoBug i wiele innych (tylko Windows): www.godevtool.com

   - Odnośniki do innych kompilatorów: Forever Young Software - linki

3. Kursy, książki:
   (przeskocz kursy)
   - (DOBRY) The Art of Assembly Language Programmnig (Art of Assembler, AoA): webster.cs.ucr.edu
     (PL) Książka została przetłumaczona na język polski przez Kremika: www.rag.kgb.pl/aoapl.php
   - PC-Asm

   - Kursy programowania w trybie chronionym
   - Assembler Programming

   - Tutorial dla początkujących - Ready to start!
   - Atrevida PC Game Programming Tutorials: atrevida.comprenica.com
   - (PL) Kurs asemblera by Skowik: www.republika.pl/skowi_magik
   - (PL) Kursy asemblera: www.pieciuk.terramail.pl/assembler.htm
   - (PL) Asembler - szybkie wprowadzenie: www.asembler.host.sk

   - (PL) Jeszcze jeden kurs asemblera w połączeniu z Pascalem: www.zsme.tarnow.pl/killer/asm/asm.htm
   - (PL) Kopia kursu Grzegorza Złotowicza: www.shitsoft.net/programowanie/asm/index2.htm oraz kilka innych arytkułów: www.shitsoft.net/biblioteka/bib_prog.htm

   - (PL) Trochę artykułów o różnej tematyce: http://coders.shnet.pl/legacy/coders/main/assembler.html

   - (PL) Asembler Programowanie

4. Polskie fora o programowaniu:
   (przeskocz fora)
   - Vademecum Programisty
   - Forum koder.org

5. Dokumentacja procesorów (ich wszystkie instrukcje, rejestry, technologie):
   (przeskocz dokumentację)
   - AMD
   - Intel
   - DDJ Microprocessor Center: www.x86.org
   - Transmeta

   - Ogólna, wiele firm, wiele procesorów (ale tylko te zgodne z Intel/AMD): Sandpile

   - Spis instrukcji według kodu rozkazu: X86Asm
   - Kolejny spis instrukcji

6. Pisanie w asemblerze pod Linuksa:
   (przeskocz asm w Linuksie)
   - Kursy, porady, dużo różnych informacji - Linux Assembly: linuxassembly.org (alternatywny adres: asm.sourceforge.net)

- Kursy dla FreeBSD - int80h.org: www.int80h.org
- Debugger pob Linuksa: PrivateICE
- Linux Assembly Tutorial
- inny tutorial
- Przykładowe maleńkie programiki
- (PL) Wstawki asemblerowe w GCC - krótki kurs w języku polskim
- Porównanie składni AT&T ze składnią Intela oraz wstęp do wstawek asemblerowych (w GCC)
- Opis wstawek asemblerowych w GCC prosto z podręcznika GCC (sekcje: 5.34 i 5.35)
- Program przerabiający składnię AT&T na składnię NASMa
- (PL) RAG
- Kopia mojego opisu przerwania int 80h

- Książka Programming from the Ground Up
- Desktop Linux Asm
- kolejny opis wstawek asemblerowych
- Różne narzędzia
- Debuger do asemblera
- Pisanie pierwszych programów w NASMie
- Specyfikacja funkcji systemowych

7. Pisanie w asemblerze pod Windowsa:
(przeskocz asm w Windowsie)
   - (PL)(DOBRY) Asembler dla Windows (kopia kursu Iczeliona)
   - (DOBRY) Programowanie pod Windows'a: kurs Iczeliona
   - Tom Cat's Win32 Asm page

   - Olly Debugger
   - NaGoA - Nasm + GoRC (Go Resource Compiler) + Alink
   - GoAsm (+dobry debugger 32-bit GoBug, GoRC i wiele innych): www.godevtool.com
   - strona Hutch'a: www.movsd.com

   - (PL) RAG

8. Portale programistyczne:
(przeskocz portale)
   - (PL) 4programmers.net

   - (PL) Programik.com
   - Programmers' Heaven

   - The Free Country
   - Free Programming Resources

   - CodeWiki - wiki z różnymi wycinkami kodu

9. Strony poświęcone pisaniu systemów operacyjnych:
(przeskocz OS)
   - (DOBRY) Bona Fide OS Development
   - (DOBRY) Operating System Resource Center

- Kursy programowania w trybie chronionym

- Dokumentacja na różne tematy: strona systemu O3one
- OSDev.org
- Zakątek Boba
- OSDev.pl
- alt.os.development - najczęściej zadawane pytania

10. Środowiska programistyczne:
(przeskocz IDE)
    - RadASM - środowisko programistyczne obsługujące wiele kompilatorów (MASM, TASM, NASM, FASM, GoAsm, HLA)
    - NasmIDE
    - TasmIDE
    - Środowisko dla FASMa (wbudowane w kompilator w wersji GUI): flatassembler.net oraz Fresh
    - WinAsm Studio
    - AsmEdit (dla MASMa)
    - Lizard NASM IDE

11. Edytory i hex-edytory/disassemblery:
(przeskocz edytory)
    - (DOBRY)Programmer's File Editor
    - Quick Editor
    - The Gun
    - HTE
    - Dużo więcej na stronach The Free Country - edytory
    - (DOBRY) XEdit
    - b2hedit
    - Biew
    - Dużo więcej na stronach The Free Country - disassemblery

12. Inne:
(przeskocz inne linki)
    - (PL)(DOBRY) Mnóstwo różnych dokumentacji: mediaworks.w.interia.pl/docs.html
    - (PL) Kursy, linki, sporo o FASMie: Decard.net

    - (PL) Architektura procesorów firmy Intel: domaslawski.fm.interia.pl
    - Forever Young Software
    - Spis instrukcji procesora i koprocesora, czasy ich wykonywania, sztuczki optymalizacyjne: www.emboss.co.nz/pentopt/freeinfo.html
    - Strona poświęcona opisom foramtów plików różnego typu (graficzne, dźwiękowe): www.wotsit.org
    - Optymalizacja, dużo linków, makra dla kompilatorów: www.agner.org/assem

    - (PL) RAG

    - (PL) Wojciech Muła
    - (PL) Programowanie - KODER
    - Tabela kodów ASCII

- Informacje o dyskach twardych itp.: www.ata-atapi.com
- Brylanty asemblera
- Linki, źródła, informacje: grail.cba.csuohio.edu/~somos/asmx86.html
- Christopher Giese
- Laura Fairhead
- Jim Webster
- LadSoft
- Paul Hsieh

- Whiz Kid Technomagic

- Koms Bomb Assembly World: http://www.mujweb.cz/www/komsbomb/

- Comrade's homepage: comrade64.cjb.net, comrade.win32asm.com, comrade.ownz.com
- Ciekawe operacje na bitach (w C)

- Sztuczki optymalizacyjne: www.mark.masmcode.com.
- FASMLIB - biblioteka procedur, nie tylko dla FASMa: fasmlib.x86asm.net
- Strona domowa Franka Kotlera
- Projekt NASMX - zestaw makr, plików nagłówkowych i przykładów dla NASMa
- Biblioteka FXT - www.jjj.de/fxt - funkcje różnego typu

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)