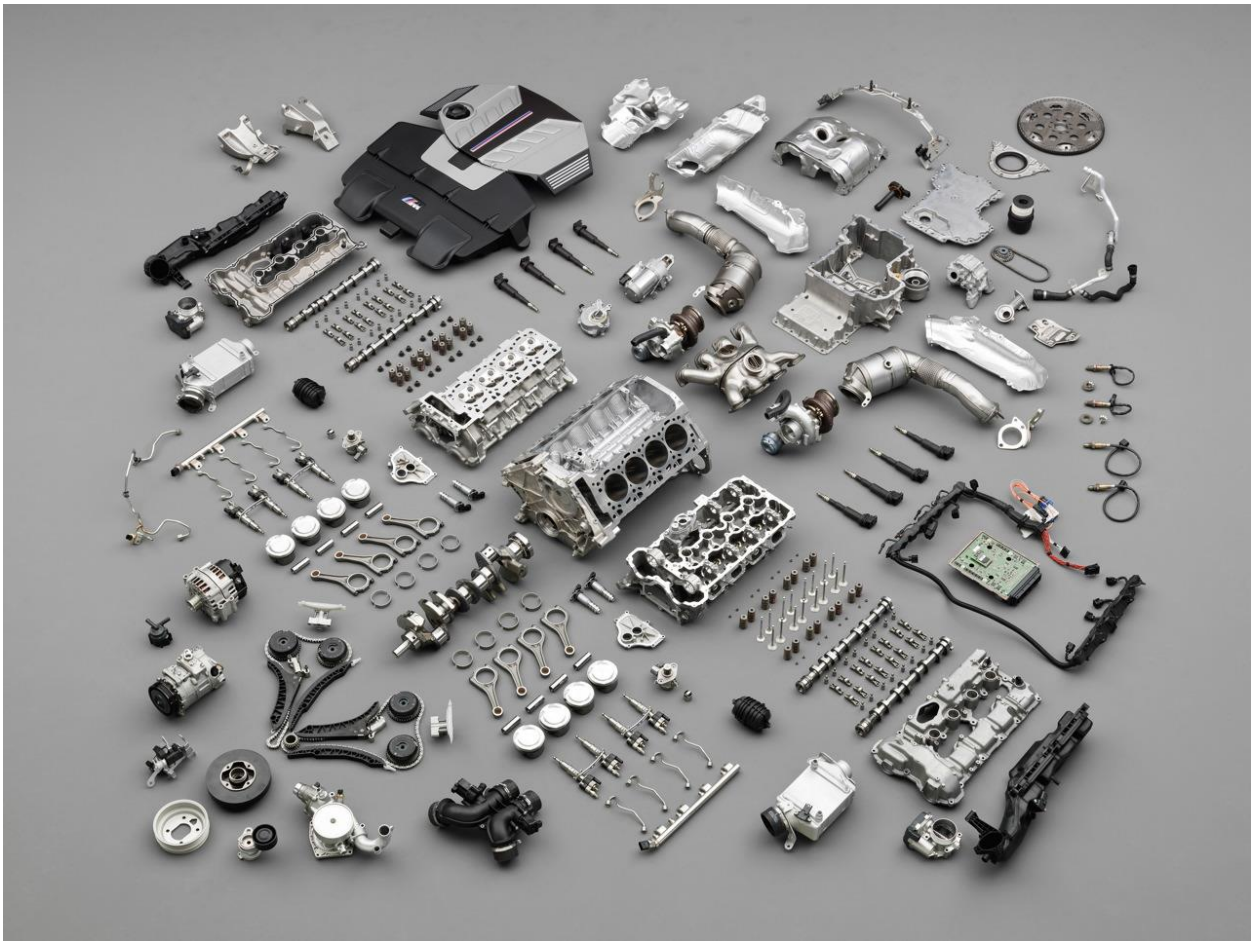
Kiedy i jak korzystać z asemblera. Podstawy programowania w asemblerze.

Autor *Bartosz Wójcik* / Opublikowane *Maj 2002* / Magazyn *Software 2.0*



Podstawy programowania w asemblerze, budowa procesora, rejestry, pamięć, instrukcje, wykorzystanie asemblera w C++ oraz Delphi.

1. Wprowadzenie

Asembler, czyli język programowania niskopoziomowego, umożliwiający wykorzystanie wszystkich możliwości procesora jest dziś już nieco zapomniany przez *„nowożytnych”* programistów.

Główną przyczyną takiego stanu jest fakt, iż pisanie w asemblerze nie należy do najprostszych czynności i jest bardzo czasochłonne (testowanie kodu, wyszukiwanie bug-ów etc.).

Jednak asembler w niektórych sytuacjach może okazać się idealnym rozwiązaniem,

przykładem mogą być wszelkiego rodzaju algorytmy wymagające szybkości działania, do których przykładowo można zaliczyć algorytmy kryptograficzne (szyfrowanie danych).

Mimo niesamowitego rozwoju kompilatorów w ostatnich latach, algorytmy takie jak np. *Blowfish, Rijndeal, Idea* napisane w asemblerze i *„ręcznie”* zoptymalizowane wykazują znaczną przewagę prędkości nad ich odpowiednikami napisanymi np. w C++ i skompilowanymi z opcjami maksymalnej optymalizacji.

Oprócz kryptografii asembler często wykorzystywany jest także przez twórców gier, czego najlepszym przykładem może być gra *QUAKE 2*, po opublikowaniu jej źródeł, okazało się, że wszelkie algorytmy wymagające szybkości działania, zostały napisane właśnie w asemblerze. Więc zaczynamy, jeszcze dla ścisłości powiem, że w tym artykule skupię się na omówieniu asemblera dla procesorów x86 i jego wykorzystaniu w środowisku Windows.

2. Podstawy asemblera

Jeśli nigdy nie pisałeś w asemblerze, aby w ogóle można było zacząć coś zrobić, najpierw należy zapoznać się choćby z podstawowymi informacjami takimi jak rejestry procesora, stos, instrukcje.

Domyślny procesor, dla przykładu użyje procesora *Intel Pentium MMX* (jedynie taki posiadam :-), z punktu widzenia programisty, do wykorzystania mamy cały zestaw instrukcji począwszy od 8, 16 i 32 bitowych x86, poprzez instrukcje zmiennoprzecinkowe i MMX.

Procesor posiada osiem 32 bitowych rejestrów ogólnego przeznaczenia + rejestr flag, dodatkowo osiem 80 bitowych rejestrów kooprocesora (st0 - st7) i tyle samo 64 bitowych rejestrów MMX (mm0 - mm7), oprócz tego procesor posiada także masę rejestrów kontrolnych, których raczej nie będziemy używać.

Teraz wyjaśnienie, co to jest ten rejestr, otóż rejestr, to jakby komórka pamięci, do której możemy tymczasowo zapisywać dane, możemy wymieniać dane między rejestrami, wykonywać operacje logiczne i arytmetyczne. Procesor Pentium jest procesorem 32 bitowym, co oznacza, że każdy z rejestrów ogólnego przeznaczenia ma 32 bitów szerokości (typ *unsigned int* pod C).Wszystkie rejestry 32 bitowe posiadają 16 bitową połówkę (pamiątka po procesorze 286), z kolei 16 bitowe połówki w rejestrach *EAX, EBX, ECX i EDX* dzielą się na kolejne dwie 8 bitowe połówki:

Nazwa rejestru	16 bitowa połówka	8 bitowe połówki	Opis
EAX	AX	AH i AL	Akumulator
EBX	BX	BH i BL	Baza

ECX	CX	CH i CL	Licznik dla operacji wykonywanych na ciągach oraz dla pętli
EDX	DX	DH i DL	Dane
ESI	SI	n/a	Rejestr źródłowy dla instrukcji operujących na ciągach
EDI	DI	n/a	Rejestr docelowy dla instrukcji operujących na ciągach
EBP	BP	n/a	Wskaźnik danych na stosie, wykorzystywany w funkcjach jako wskaźnik parametrów zapamiętanych na stosie
ESP	SP	n/a	Wskaźnik stosu

Rejestry ogólnego przeznaczenia

Pisząc program, czy też wstawkę w asemblerze pod Windows, możemy korzystać z wszystkich rejestrów ogólnego przeznaczenia, ale często grozi to destabilizacją pracy programu, gdy użyje się specjalnych rejestrów, konkretnie ESP i EBP, np. wyzerowanie rejestru ESP w kodzie funkcji, najprawdopodobniej spowoduje zawieszenie się programu w jego dalszej części (jeśli program np. będzie chciał powrócić z funkcji do kodu programu).

Stos

Stos to mówiąc potocznie obszar pamięci zarezerwowany na potrzeby programu, jest wykorzystywany m.in. do przekazywania parametrów funkcjom (jako 32bitowe wartości), służy też do tymczasowego przechowywania danych, wszystkie zmienne lokalne również *„tworzone”* są na stosie. Po uruchomieniu programu rejestr ESP (Stack Pointer - wskaźnik stosu) wskazuje na koniec stosu, zapamiętując jakieś dane na stosie, rejestr ESP jest dekrementowany, i w obszar pamięci wskazywany przez ESP zapisywana jest wartość. Do zapisywania danych na stosie wykorzystuje się instrukcję *push*, np:

```
__asm {

push    5               // zapisz na stosie liczbę 5 (32bit)
push    eax             // zapamiętuje na stosie zawartość rejestru EAX
push    dword ptr[edx]  // zapamiętuje wartość wskazywaną przez wskaźnik zapisany
                        // w rejestrze EDX

sub     esp,4           // odpowiednik instrukcji 'push 5'
mov     dword ptr[esp],5

sub     esp,4           // odpowiednik instrukcji 'push eax'
mov     dword ptr[esp],eax
}
```

Aby zdjąć wartość ze stosu należy skorzystać z instrukcji *pop*, która działa w odwrotny sposób do *push*, najpierw odczytywana jest wartość spod adresu wskazywanego przez rejestr ESP, następnie rejestr ESP jest inkrementowany:

```
__asm {

push    5               // zapamiętaj na stosie 4 32bitowe wartości
```

```
push    eax
push    dword ptr[edx]
push    13B0C032h

pop     eax             // zdejmij ze stosu ostatnią zapamiętaną wartość,
                        // która w tym wypadku jest liczba 13B0C032h
pop     dword ptr[edx]  // operacja ta nic nie zmienia, ponieważ zapamiętana
                        // była wartość wskazywana przez rejestr EDX i została
                        // ona tylko przywrócona

pop     edx             // do rejestru EDX odczytaj wartość jaką miał rejestr EAX
pop     ecx             // do rejestru ECX zostanie wpisana liczba 5

push    5               // zapamiętaj na stosie liczbę 5
                        // instrukcje symulujące 'pop eax'
mov     eax,dword ptr[esp]
add     esp,4
}
```

Ograniczenia w systemie Windows

Jak pisałem wcześniej, z naszego punktu widzenia mamy dostęp do wszystkich instrukcji procesora, dla którego piszemy kod, ale ogranicza nas system, w naszym wypadku Windows. Oznacza to, że możemy np. używać instrukcji odwołujących się do portów, jak to miało miejsce w systemie *MS-DOS*, ale najpewniej skończy się to zawieszeniem programu.

Do instrukcji, których używanie kończy się potocznie zwanym zwisem, można zaliczyć ww. instrukcje operujące na portach, oprócz tego instrukcje odwołujące się do przerwań, modyfikujące rejestry kontrolne i segmentowe.

Apropos rejestrów segmentowych, Windows używa modelu pamięci *FLAT*, co oznacza, że teoretycznie mamy dostęp do całej pamięci od adresu 0 po 0xFFFFFFFF (nie polecam jednak modyfikować obszarów zarezerwowanych przez system) i żeby uzyskać dostęp do danych wcale nie trzeba odwoływać się np. do rejestru *DS:*.

3.Użycie asemblera

Aby skorzystać z dobrodziejstw asemblera należy najpierw sprawdzić czy nasze narzędzie pracy umożliwia jego wykorzystanie. Produkty takie jak *Borland Delphi, Builder, Watcom C++*, czy *Microsoft Visual C++* umożliwiają użycie (kompilowanie) kodu asemblera, z popularnych pakietów RAD jedynie *Visual Basic* nie daje możliwości pisania kodu w asemblerze. Wyżej wspomniane produkty umożliwiają wykorzystanie kodu asemblera na dwa sposoby, pierwszym z nich są tzw. *wstawki*, czyli mówiąc prościej, kod asemblera dodawany jest pomiędzy regularnym kodem napisanym np. w C++. Drugim sposobem umożliwiającym wykorzystanie asm-a jest linkowanie (łączenie) modułów napisanych w asemblerze z modułami napisanymi np. w *Delphi*.

Wstawki asemblerowe

Zanim przystąpimy do pisania kodu asemblera, warto sprawdzić jak mamy go pisać, istnieją bowiem dwa typy składni, w której zapisuje się kod asemblera. Pierwszym typem jest tzw. składnia *„intel”* wykorzystywana w produktach takich jak *Delphi, Builder, MSVC, Borland TASM, Microsoft MASM* (kompilatory asemblera), składnia ta obecnie stanowi standard i 90% źródeł zapisanych jest zgodnie z nią. Drugim typem

zapisu kodu asemblera jest składnia *„at&t”* wykorzystywana np. w kompilatorach C takich jak *GCC* (platforma Linux), *DJGPP* i *LCC*.

Wstawki stanowią najprostszy sposób na pisanie kodu asm, aby zapisać kod asemblera w *Delphi* lub *Builderze* należy go ująć pomiędzy znacznikami *„asm”* co oznacza początek kodu asemblera, po zakończeniu kodu należy zamknąć znacznik *„end;”*, przykład:

```
// nasze pierwsze 'hello world' w asemblerze, wersja dla Delphi
asm                     // początek kodu asemblera

mov     eax,1           // przenieś wartość 0x00000001 do rejestru EAX
                        // w C++ odpowiednikiem tej instrukcji jest operator
                        // przypisania '=' np.
                        // x = 1;
                        // w Delphi odpowiednikiem także jest operator
                        // przypisania ':=' np.
                        // y := 1;

mov     ecx,eax         // przenieś do rejestru ECX zawartość rejestru EAX,
                        // czyli w ECX znajdzie się wartość 0x00000001

shl     ecx,2           // Shift Left, przesuwanie bitów rejestru ECX o 2 bity
                        // w lewo, przesuwanie bitów jak wiadomo (lub nie :),
                        // służy do mnożenia wartości przez kolejne potęgi
                        // liczby 2, przesuwając zawartość 0x00000001 o 2 bity
                        // w lewo w ECX znajdzie się wartość 0x00000001 * 4 = 0x00000004
                        // w C++ przesuwanie bitów zapisujemy operatorem '<<'
                        // np. x = y << 2;
                        // w Delphi przesuwanie bitów zapisuje się podobnie
                        // jak w kodzie asemblera operatorem 'shl' lub 'shr'
                        // np. x := y shl 2;

shr     eax,1           // Shift Right, przesuwanie bitów rejestru EAX o 1 bit
                        // w prawo

and     eax,0           // And, logiczne mnożenie bitów, tabelka funkcji And:
                        // 0 * 0 = 0
                        // 1 * 0 = 0
                        // 1 * 1 = 1
                        // dowolna wartość przemnożona przez 0 da 0, czyli w
                        // tym wypadku rejestr EAX zostanie wyzerowany
                        // w C++ odpowiednikiem tej instrukcji jest operator '&'
                        // np. x = y & 0;
                        // w Delphi
                        // np. x = y and 0;

or      eax,0FFFFFFFFh  // Or, logiczne dodawanie bitów, tabelka funkcji Or:
                        // 0 + 0 = 0
                        // 1 + 0 = 1
                        // 1 + 1 = 1
                        // w tym przypadku do EAX dodawana jest wartość
                        // 0xFFFFFFFF, pisząc w Delphi 32bitowe liczby hex
                        // zaczynające się od litery, należy poprzedzić je
                        // jednym zerem na początku i literą 'h' na końcu, co
                        // oznacza, że liczba zapisana jest w postaci
                        // heksadecymalnej
                        // w C++ odpowiednikiem funkcji Or jest operator '|'
                        // np. x = y | 0xFFFFFFFF;
                        // w Delphi
                        // np. x := y or $FFFFFFFF;

sub     edx,edx         // Subtract, odejmowanie, wyzeruj zawartość rejestru EDX
                        // w C++ odpowiednikiem tej instrukcji jest np.
                        // x = x - x;

xor     eax,eax         // eXclusive Or tabela dla funkcji XOR
                        // 0 ^ 0 = 0
                        // 1 ^ 0 = 1
                        // 1 ^ 1 = 0
                        // funkcja daje 1 dla 2 rożnych bitów, jeśli bity sa
                        // takie same to zostaną wyzerowane, więc instrukcja
                        // xor eax,eax wyzeruje zawartość rejestru eax
```

```
                        // odpowiednikiem w C++ funkcji XOR jest operator '^'
                        // np. x = x ^ y
                        // w Delphi
                        // np. x := x xor y;

end;                    // koniec kodu asemblera
```

Sposób wstawiania kodu asemblera do kodu *MSVC* rożni się praktycznie tylko sposobem otwarcia znaczników informujących kompilator, że jest to kod asemblera:

```
// nasze drugie 'hello world' w asemblerze
__asm {                 // początek kodu asemblera

push    5               // zapamiętaj na stosie wartość 0x00000005
pop     eax             // zdejmij ze stosu wartość 0x00000005 i zapisz ją do
                        // rejestru eax

push    eax             // zapamiętaj na stosie zawartość rejestru EAX (czyli
                        // w tym przypadku 5)
pop     edx             // zdejmij ze stosu wartość 5 i zapisz ja do rejestru
                        // EDX

mov     ax,0FFFFh       // zapisz do 16bitowej połówki rejestru EAX wartość 0FFFFh
mov     dx,ax           // zapisz do 16bitowej połówki rejestru EDX wartość z
                        // rejestru AX
mov     al,11           // zapisz do dolnej (LO) 8 bitowej połówki
                        // rejestru AX wartość 11 (decymalnie)
mov     ah,11h          // zapisz do górnej (HI) 8 bitowej połówki rejestru AX
                        // wartość 11h (hex) co decymalnie równa się 17
}                       // koniec kodu asemblera
```

Korzystanie ze zmiennych

Pisząc w asemblerze ma się dostęp do wszystkich zmiennych globalnych oraz jeśli kod znajduje się w procedurze, to także dostęp do zmiennych lokalnych oraz parametrów procedury/funkcji, czyli praktycznie ma się takie same możliwości jak zwykły kod. Przykład wykorzystania zmiennych globalnych i lokalnych:

```
// zmienne globalne
var
ByteVar: Byte;          // bajt, 8 bitów
WordVar: Word;          // słowo, 16 bitów
IntVar: Integer;        // podwójne słowo, 32 bity
...

procedure noop;

// zmienne lokalne funkcji noop
var
LocalByte: Byte;
LocalWord: Word;
LocalInt: Integer;

begin

// inicjalizuj zmienne globalne
ByteVar := $FF;         // 8 bitowa wartość
WordVar := $FFFF;       // 16 bitowa
IntVar  := $FFFFFFFF;   // 32 bitowa

asm
mov     al,ByteVar      // do 8 bitowego rejestru wpisz 8 bitową wartość
mov     LocalByte,al    // zapisz do zmiennej lokalnej 8 bitową wartość

mov     ax,WordVar      // do 16 bitowego 16 bitową
mov     LocalWord,ax

mov     eax,IntVar      // do 32 bitowego 32 bitową
mov     LocalInt,eax

end;
```

```
end;
```

Przykład dla *MSVC* niewiele różni się od tego z *Delphi*:

```
// zmienne globalne
char ByteVar;
short WordVar;
int IntVar;
...

void noop()
{
// zmienne lokalne
char LocalByte;
short LocalWord;
int LocalInt;

// inicjalizuj zmienne globalne
ByteVar = 0xFF;         // 8 bitowa wartość
WordVar = 0xFFFF;       // 16 bitowa
IntVar  = 0xFFFFFFFF;   // 32 bitowa

__asm{

mov     al,ByteVar      // do 8 bitowego rejestru wpisz 8 bitową wartość
mov     LocalByte,al    // zapisz do zmiennej lokalnej 8 bitową wartość

mov     ax,WordVar      // do 16 bitowego 16 bitową
mov     LocalWord,ax

mov     eax,IntVar      // do 32 bitowego 32 bitową
mov     LocalInt,eax

}

}
```

Oprócz samych wstawek, całe funkcje mogą być zapisane w asemblerze, tutaj kilka ważnych uwag, funkcje z założenia zwracają wartości, jeśli piszemy funkcję, musimy zatroszczyć się żeby zwracana wartość przed wyjściem z funkcji była zapisana w rejestrze EAX, najpierw prosty przykład:

```
// wersja dla Delphi
function dodaj(x,y:integer):integer;
asm
mov     edx,x           // wpisz do rejestru EDX pierwszy parametr funkcji
mov     ecx,y           // do ECX wpisz drugi parametr funkcji
add     edx,ecx         // dodaj do siebie x i y
mov     eax,edx         // wynik dodawania zapisz w rejestrze EAX
                        // taką wartość zwraca funkcja
end;

// wersja dla C++
int mnoz(int x,int y)
{
__asm{
mov     edx,x           // wpisz do rejestru EDX pierwszy parametr funkcji
mov     ecx,y           // do ECX wpisz drugi parametr funkcji
imul    edx,ecx         // wymnóż x * y
mov     eax,edx         // wynik mnożenia zapisz w rejestrze EAX
                        // taką wartość zwraca funkcja
}

}
```

Wiemy już, że funkcje pisane w asemblerze musza zwracać wartości w rejestrze EAX, a co z innymi rejestrami? Mówiąc krótko, rejestry EAX, EDX i ECX po wyjściu z funkcji mogą ulec zmianie, rejestry EDI, ESI, EBX i EBP domyślnie nie mają prawa być zmieniane (ich stan ma być taki sam jak przed wywołaniem funkcji). Spytacie,

dlaczego tak jest? Otóż w kodzie *„wyprodukowanym”* przez kompilatory HLL (High Level Language - język wysokiego poziomu) ww. rejestry używane są w głównych pętlach programu do trzymania np. adresów jakichś funkcji, stałych wartości itp. i ich zmiana w kodzie funkcji może spowodować nieprawidłową lub niestabilną pracę całej aplikacji. Jak ustrzec się przed takimi błędami, otóż bardzo prosto:

```
// wersja dla Delphi
function licz(w,x,y,z:integer):integer;
asm
push    edi             // zapamiętaj na stosie kolejno wartości rejestrów
push    esi             // EDI, ESI i EBX
push    ebx

mov     edi,w           // odczytaj parametry funkcji do kolejnych rejestrów
mov     esi,x
mov     edx,y
mov     ebx,z

add     edi,esi         // w + x
add     edx,ebx         // y + z

imul    edi,edx         // (w+x) * (y+z)

xchg    eax,edi         // eXCHanGe, zamień wartość rejestru EAX z wartością
                        // rejestru EDI, mówiąc prościej wartość z rejestru
                        // EAX zostanie przepisana do rejestru EDI, a wartość
                        // rejestru EDI zostanie przepisana do rejestru EAX, w
                        // którym zwracamy wynik funkcji



pop     ebx             // zdejmij ze stosu wartości zapamiętanych rejestrów
pop     esi             // wartości zdejmujemy od końca (patrząc na kod
pop     edi             // można powiedzieć, że symetrycznie), czyli
                        // jeżeli rejestry były zapisane w kolejności
                        // EDI, ESI, EBX to zdejmujemy je ze stosu w kolejności
                        // EBX, ESI, EDI

end;
```

Oprócz tego, że rejestry EDI, ESI, EBX i EBP nie mogą być zmieniane, dodatkowo flaga kierunku DF (Direction Flag) przed wywołaniem funkcji domyślnie jest wyzerowana (takie jest założenie) i po wyjściu z funkcji oczekiwane jest, że nadal będzie wyzerowana. Wystarczy użyć instrukcji *„CLD”* jeśli jej stan jest zmieniany wewnątrz funkcji.

Pisząc kod w asemblerze wykorzystujący stos, należy szczególną uwagę zwrócić także na to, żeby wskaźnik stosu ESP był za każdym razem korygowany, np. jeśli w procedurze czy funkcji zapamiętamy coś na stosie, to przed wyjściem z funkcji należy tą wartość zdjąć ze stosu, tym razem przykład dla *MSVC*:

```
// przykład funkcji szyfrującej
void crypt(unsigned char *string)
{
__asm{
push    edx             // zapamiętaj na stosie rejestr EDX
mov     edx,string      // pobierz parametr ze stosu, czyli w tym wypadku
                        // wskaźnik do stringa, który ma by zaszyfrowany

cmp     edx,0           // sprawdź czy parametr podany funkcji jest poprawny
je      _exit_encrypt   // jeśli parametr jest niepoprawny wyjdź z funkcji

_encrypt_loop:
mov     al,byte ptr[edx]// pobierz kolejny bajt ze stringa podanego jako parametr
cmp     al,0            // sprawdź czy to koniec stringa, stringi są zapisane
                        // jako asciiz (bajt 00h oznacza koniec stringa)
```

```
je      _exit_encrypt

xor     al,7            // zaszyfruj bajt prostym xor-em
mov     byte ptr[edx],al// zapisz zaszyfrowany bajt
inc     edx             // ustaw wskaźnik stringa na kolejny bajt
jmp     _encrypt_loop   // wykonuj pętle szyfrowania aż do momentu napotkania
                        // bajta 00h
_exit_encrypt:
pop     edx             // ważne, koryguj stos, przywróć oryginalną wartość
                        // rejestru EDX

}
}
```

Wywoływanie funkcji z poziomu asemblera

Czasami w kodzie asemblera trzeba będzie wywołać jakąś funkcję, jak to zrobić? Bardzo prosto, funkcje wywołuje się instrukcją *call nazwa_funkcji*, warto zauważyć, że istnieje kilka sposobów wywoływania i *„sprzątania”* po funkcjach:

Nazwa	W kodzie C	Parametry	Zwracane wartości	Modyfikowane rejestry	Info
cdecl	cdecl	zapisywane na stosie, stos nie jest korygowany przez funkcję	eax, 8 bajtów: eax:edx	eax, ecx, edx, st(0), st(7), mm0, mm7, xmm0, xmm7	Sposób wywoływania funkcji z bibliotek C, wprowadzony przez Microsoft, wszystkie funkcje systemowe na platformie Linux również używają tego standardu
fastcall	__fastcall	ecx,edx, reszta parametrów przekazywana przez stos	eax, 8 bajtów: eax:edx	eax, ecx, edx, st(0), st(7), mm0, mm7, xmm0, xmm7	Microsoft wprowadził ten standard, ale potem w swoich produktach zmienił go na standard cdecl
watcom	__declspec (wcall)	eax, ebx, ecx, edx	eax, 8 bajtów: eax:edx	eax	Standard wywoływania funkcji wprowadzony przez firmę Watcom w ich kompilatorze C++

stdcall	__stdcall	zapisywane na stosie, stos korygowany przez samą funkcję	eax, 8 bajtów: eax:edx	eax, ecx, edx, st(0)st(7), mm0, mm7, xmm0, xmm7	Domyślny typ wywoływania funkcji API w Windows, w bibliotekach DLL
register	n/a	eax, edx, ecx, reszta na stosie	eax	eax, ecx, edx, st(0)st(7), mm0, mm7, xmm0, xmm7	Metoda wywoływania funkcji w Delphi firmy Borland

Sposób wywoływania funkcji w naszych programach (nie mówię o WinApi) często zależy od opcji, z jakimi program został skompilowany, dla *Delphi* standardem jest typ *„register”*, dla większości programów napisanych w C standardem jest *„cdecl”*.

Funkcje WinApi (systemowe Windows) korzystają z mechanizmu *stdcall*, czyli parametry funkcji najpierw zapamiętywane są na stosie, po czym następuje wywołanie funkcji, po wywołaniu funkcji, nie trzeba korygować stosu (zdejmować wcześniej zapamiętanych parametrów), ponieważ funkcja zrobi to za nas, ciekawostką jest, że kilka funkcji WinApi nie korzysta ze sposobu wywoływania *stdcall*, ale z *cdecl*, czyli parametry zapamiętywane są na stosie, po czym wywoływana jest funkcja, ale samo korygowanie stosu musi być wykonane ręcznie. Przykładem takiej funkcji jest *„wsprintfA”* z biblioteki systemowej Windows user32.dll (jej odpowiednikiem w bibliotekach C jest *sprintf*), ten sposób został najprawdopodobniej wprowadzony, ponieważ ww. funkcje nie posiadają stałej ilości parametrów:

```
// string globalny
unsigned char tytul[] = "Liczby x i y";
...

// funkcja zamienia liczby x i y na ciąg asciiz, po czym wyświetlane
// jest okienko informacyjne pokazujące liczby x i y zapisane już
// w formie stringa
unsigned int int2str(unsigned char *bufor, unsigned int x, unsigned int y)
{
// string lokalny, dostępny tylko dla funkcji int2str
unsigned char format[] = "x = %lu\ny = 0x%X\n";

__asm{

// zwróć uwagę, w jaki sposób zapamiętywane są parametry funkcji,
// w C++ wywołanie tej funkcji wyglądałoby następująco:
// wsprintf(bufor, "x = %lu\ny = 0x%X\n", x, y);
// w asemblerze parametry przed wywołaniem funkcji zapamiętywane
// są na stosie w odwrotnej kolejności
```

```
push    y               // liczba y
push    x               // zapamiętaj na stosie liczbę x
lea     eax,format      // do EAX załaduj adres lokalnego stringa
push    eax             // zapisz wskaźnik do stringa formatującego
push    bufor           // zapamiętaj wskaźnik bufora wyjściowego, gdzie znajdzie
                        // się sformatowany tekst
call    wsprintfA       // wywołaj funkcję WinApi
add     esp,4*4         // koryguj stos, 4*4 = 16 bajtów, tyle zajmują zapamiętane
                        // na stosie parametry przed wywołaniem funkcji, pisząc kod
                        // np. w C++ kompilator sam troszczy się o to żeby wskaźnik
                        // stosu był korygowany, ale pisząc w asemblerze trzeba
                        // samemu zatroszczyć się o wszystko

push    MB_ICONINFORMATION // typ ikony jaka pojawi się obok tekstu w oknie
push    offset tytul    // tytuł okienka (zmienna globalna, używamy słowa
                        // kluczowego 'offset' ponieważ zapisujemy na stosie adres
                        // stringa)
push    bufor           // tekst jaki pojawi się w okienku
push    0               // uchwyt okna 'matki'
call    MessageBoxA     // pokaż okienko informacyjne
}
}
```

4. Jednostka MMX

MMX to nazwa rozszerzeń, jakie wprowadziła firma Intel do serii procesorów Pentium, skrót podobno pochodzi od *„MultiMedia eXtensions”*, ale sam Intel temu zaprzecza, a także nigdy nie wyjaśnił tej kwestii. Rozszerzenia MMX w procesorach Pentium obejmują nową listę rozkazów (konkretnie 57) oraz 8 nowych, 64 bitowych rejestrów.

Rejestry MMX współdzielone są z rejestrami FPU, oznacza to, że nie mogą być jednocześnie wykonywane instrukcje operujące na jednostce zmiennoprzecinkowej FPU (Floating Point Unit) oraz na jednostce MMX. Instrukcje MMX potrafią operować na danych w trybie *SIMD* (Single Instruction Multiple Data), tryb ten oznacza, że jedna instrukcja potrafi przetwarzać jednocześnie wiele danych, co nie jest możliwe korzystając ze standardowych instrukcji x86.

Instrukcje MMX znakomicie nadają się do przetwarzania multimedialnych danych, jak np. wideo, grafika, dźwięk, czego żywym przykładem mogą być takie programy jak DivX czy Winamp, intensywnie wykorzystujące kod MMX. Obecnie większość procesorów począwszy od tych firmy Intel poprzez AMD i Cyrix wspiera MMX.

Mimo, że MMX od ładnych paru lat stanowi praktycznie standard, kompilatory HLL domyślnie nie generują kodu MMX (oprócz specjalizowanych takich jak np. *VectorC*), tutaj przychodzi z odsieczą idea programowania MMX w asemblerze.

Pisząc procedury w MMX można niejednokrotnie uzyskać 100% przyrost prędkości w stosunku do oryginalnego kodu, wpływ na to, ma fakt wykorzystania ww. trybu SIMD. Wyobraźmy sobie np. taką sytuację, mamy dwie tablice 8 bajtowe i chcemy dodać kolejne bajty z obu tablic do siebie, przykładowo w C++ robimy to tak:

```
unsigned char tablica1[] = { 0x0A,0x1A,0x2A,0x3A,0x4A,0x5A,0x6A,0x7A };
unsigned char tablica2[] = { 0xA7,0xA6,0xA5,0xA4,0xA3,0xA2,0xA1,0xA0 };
...

for (int i = 0; i < 8; i++)
{
          tablica1[i] += tablica2[i];
```

```
}
```

Wszystko jest ok, ale tak czy tak operacja dodawania bajtów zostanie powtórzona 8 razy, a teraz spójrzmy jak można to zrobić o wiele wydajniej z wykorzystaniem MMX:

```
__asm {

movq    mm0,qword ptr[tablica1] // załaduj 8 bajtów z pierwszej tablicy
                                // do rejestru MM0

movq    mm1,qword ptr[tablica2] // 8 bajtów z drugiej tablicy do rejestru MM1
paddb   mm0,mm1                 // dodaj do siebie bajty z rejestrów MM0 i MM1
movq    qword ptr[tablica1],mm0 // zapisz wynik
}
```

W sumie jedna instrukcja zamiast 8 powtórzeń dodawania, czyż nie jest to piękne, ale przede wszystkim o ile wydajniejsze. Kilka przykładowych funkcji operujących na grafice:

```
#define IMG_WIDTH 640
#define IMG_HEIGHT 320

...

//
// funkcja inicjalizuje jednostkę MMX, należy ją wywoływać
// przed operacjami MMX oraz po tym jak operowaliśmy na FPU
// i ponownie chcemy używać MMX, oraz jeśli chcemy operować na
// FPU po operacjach na MMX
//
void InitMMX()
{
__asm emms;             // Empty MultiMedia State, inicjalizuję
}                       // jednostkę MMX


//
// efekt zanikania ekranu (tzw. fadeout), fullscreen
//
void fadeout(DWORD *lpScreen,DWORD iRounds)
{
__asm {

mov     edx,iRounds     // pobierz liczbę powtórzeń

mov     eax,03030303h   // maska dla kolejnych składowych pikseli,
                        // zmniejszając wartość każdej składowej (RGB)
                        // kolejnych pikseli uzyskujemy efekt zanikania
                        // obrazu

movd    mm0,eax         // przenieś maskę do młodszej (lo) połówki
                        // rejestru MM0

punpckldq mm0,mm0       // kopiuj maskę na pozycje starszej połówki rejestru
                        // MMX tak, że cały rejestr, będzie zawierał wartość
                        // 0x0303030303030303

pxor    mm1,mm1         // wyzeruj rejestr MM1

_fadeout_max:

paddb   mm1,mm0         // pomnóż maskę, która będzie odejmowana od składowych
                        // pikseli, razy ilość powtórzeń (parametr iRounds)
dec     edx             //
jne     _fadeout_max


mov     eax,lpScreen    // wskaźnik mapy obrazu załaduj do rejestru EAX

                        // liczba pikseli obrazu /2 (piksel na mapie obrazu
                        // ma 4 bajty), liczba pikseli podzielona jest przez
                        // 2 ponieważ z użyciem MMX przetwarzać będziemy 2
                        // piksele jednocześnie
mov     ecx,(IMG_WIDTH*IMG_HEIGHT) / 2
```

```
_clear_screen_2_mmx:
                        // odczytaj do rejestru MM0 2 piksele z mapy obrazu
movq    mm0,qword ptr[eax]
psubusb mm0,mm1         // odejmuj od składowych 2 pikseli maskę, składowe
                        // i maska traktowane są, jako tablica 8 osobnych
                        // bajtów (SIMD)

                        // zapisz zmodyfikowane 2 piksele do mapy obrazu
movq    qword ptr[eax],mm0

add     eax,8           // koryguj wskaźnik mapy obrazu, ustaw go na
                        // następne 2 piksele

dec     ecx             // liczba powtórzeń pętli (ilość pikseli w buforze
                        // ekranu / 2)
jne     _clear_screen_2_mmx
}

}


//
// negatyw obrazu
//
void negatyw(DWORD *lpScreen)
{
__asm {

mov     eax,lpScreen    // do EAX wpisz wskaźnik mapy obrazu

                        // ECX liczba pikseli / 4, ponieważ przetwarzamy
                        // 4 piksele jednocześnie
mov     ecx,(IMG_WIDTH*IMG_HEIGHT) / 4

pcmpeqb mm7,mm7         // ustaw rejestr MM7 na 0xFFFFFFFFFFFFFFFF

_neg_mmx:
                        // odczytaj 2 piksele obrazu do MM0
movq    mm0,qword ptr[eax]
pxor    mm0,mm7         // XOR -1 działa jak logiczna funkcja NOT
movq    qword ptr[eax],mm0

movq    mm0,qword ptr[eax+8]
pxor    mm0,mm7
movq    qword ptr[eax+8],mm0

add     eax,16          // ustaw wskaźnik mapy obrazu na kolejne 4 piksele
dec     ecx
jne     _neg_mmx
}

}


//
// rozmazywanie obrazu (tzw. blur)
//
void blur(DWORD *lpScreen)
{
__asm {

push    esi             // zapamiętaj rejestry ESI i EDI
push    edi

mov     esi,lpScreen    // wskaźnik mapy obrazu załaduj do ESI


mov     ecx,( (IMG_WIDTH*IMG_HEIGHT) - (IMG_WIDTH*8) + 4 )
mov     eax,IMG_WIDTH*4 // szerokość wiersza w mapie obrazu
mov     edx,IMG_WIDTH*8 // szerokość dwóch wierszy

lea     esi,[esi+eax+4] // ustaw mapę obrazu na pierwszy piksel
                        // drugiego wiersza ekranu

pxor    mm7,mm7         // ustaw mm7 na 0
movd    mm0,[esi-4]     // czytaj 1 piksel
```

```
_blur_more:

movd    mm1,[esi+4]

mov     edx,esi
sub     edx,eax
movd    mm2,[edx]

movd    mm3,[esi+eax]
punpcklbw mm0,mm7       // rozpakuj składowe kolejnych 4 pikseli
punpcklbw mm1,mm7       // do WORDów
punpcklbw mm2,mm7
punpcklbw mm3,mm7
paddusw mm0,mm1         // dodaj kolejno do siebie wartości składowe
paddusw mm0,mm2         // kolorów 4 pikseli
paddusw mm0,mm3
psrlw   mm0,2           // suma składowych kolorów / 4, w ten sposób
                        // obliczamy
packuswb mm0,mm7        // spakuj składowe piksela zapisane jako WORDy
                        // w rejestrze MM0 do DWORDa
movd    [esi],mm0       // zapisz piksel
add     esi,4

dec     ecx
jne     _blur_more

pop     edi
pop     esi

}
}
```

5. Kiedy korzystać z asemblera

Jak wspomniałem na początku artykułu, asembler wykorzystywany jest głownie tam gdzie liczy się prędkość i tam znajduje zastosowanie. Pisząc jakieś algorytmy warto się czasami zatrzymać i pomyśleć czy czasem nie szybciej będzie, jeśli w jakichś krytycznych punktach programu (jak pętle itp.) użyjemy np. MMX.

Wyobraźcie sobie, że napisaliście właśnie encoder mp3, konkurencja zrobiła to samo, ale wasz korzysta z ręcznie napisanego kodu MMX, który jest trzykrotnie szybszy od tego konkurencji. Teraz kogo wybierze użytkownik, który zamiast czekać 30min na wykonanie zadania, będzie musiał jedynie poczekać 10min? Odpowiedź nasuwa się sama.

Asembler, oprócz tego, że idealnie nadaje się do pisania algorytmów wymagających prędkości, także jest wykorzystywany do pisania specyficznych programów jak np. exe-kompresory. Założę się, że większość osób kojarzy programy takie jak *UPX* czy *Aspack*, programy te służą do kompresji plików wykonywalnych, mówiąc prościej, jeśli napiszemy jakiś program i będzie zajmował dajmy na to 700kb, to po skompresowaniu go *UPXem* jego rozmiar zmniejszy się do ok. 300kb, ale program w formie pliku *EXE* będzie tak samo funkcjonalny jak przed kompresją. W tym wypadku asembler jest wykorzystany do napisania kodu tzw. *loadera*, czyli fragmentu kodu, który zapisany jest w pliku EXE (prawie jak wirus) i po uruchomieniu takiego programu, kod loadera dekompresuje dane pliku EXE i powoduje jego uruchomienie. Napisanie kodu loadera w języku HLL, obojętnie czy jest to C++, Delphi czy nawet Power Basic, jest praktycznie niemożliwe.

Można powiedzieć, że asembler ma specyficzne przeznaczenie, prędkość i nietypowe aplikacje, ale nie do końca, pisanie w asemblerze to nie tylko wstawki i pojedyncze

procedury. Całe programy mogą być pisane w asemblerze, czasami słyszę, jak ludzie mówią, że to niemożliwe, że nie można napisać dużej aplikacji w asemblerze od podstaw. Najczęściej mówią tak ci, którzy obcowali z asemblerem pięć minut, jeśli już nabierze się wprawy w kodowaniu, to nic nie stoi na przeszkodzie żeby budować profesjonalne aplikacje. Pisząc cały program w asemblerze mamy nad nim totalną kontrolę, wszystko zależy od nas, program jest wykonywany zgodnie z naszą wolą, nie jesteśmy zdani na *„łaskę”* kompilatora.

Ostatnimi czasy pisanie w asemblerze jest bardzo proste i wygodne, dużo ludzi na całym świecie zaczyna widzieć w tym języku prawdziwą magię, powstaje dużo projektów, można znaleźć całą masę przykładowych tutoriali i kodów źródłowych, dzięki którym każdy problem przestaje być problemem. Pisanie całych aplikacji w asemblerze ma też tą zaletę, iż nawet pięciomegabajtowy kod źródłowy, zostanie skompilowany do ok. 90kb pliku wykonywalnego. Przykładowo, aplikacja napisana w Delphi 6, zawierająca 1 okno, po skompilowaniu zajmuje ok. 300kb, a program napisany w asemblerze, który robi dokładnie to samo i bez problemów uruchamia się na całej serii systemów operacyjnych Windows począwszy od 95 po XP, zajmuje 4 kB. Dlaczego tak jest, przyczyna jest prosta, kompilator sam dodaje wszystko to, co nie jest potrzebne, ale mogłoby być użyte, a dlaczego tak to jest zrobione, to wypadałoby już zapytać firmy produkujące kompilatory.

Pomimo tego, że asembler może być wykorzystywany do wielu pożytecznych rzeczy, wykorzystywany jest też do pisania destrukcyjnych programów, jakimi niezaprzeczalnie są wirusy albo exploity, ale jak mawiał Kubuś Puchatek, to jest już całkiem inna historia...

6. Podsumowanie

Powyższe przykłady prezentują tylko mały zakres tego, co umożliwia asembler, jeszcze dużo jest do odkrycia, zarówno przed Wami jak i przede mną, bo asembler wbrew temu, co mówią, nie jest martwy, ciągle się zmienia, ewoluuje, dając nam możliwości, jakich nie da nam żaden język wysokiego poziomu. SSE, SSE2, 3DNow, terminy wymieniane w prasie, to nie fikcja, to wszystko stoi otworem, wystarczy tylko po to sięgnąć.

Z mojej strony mogę powiedzieć, że pisanie w asemblerze daje mi uczucie wolności, jakiego nie zaznałem pisząc w żadnym innym języku, i życzę Wam, aby podróż z asemblerem nie skończyła się na tym artykule.

7. Literatura

- www.win32asm.cjb.net - strona dla programistów ASM, źródła, tutoriale, forum
- www.int80h.org - programowanie w asemblerze pod FreeBSD
- www.rbthomas.freeserve.co.uk - grafika w Windows, algorytmy, fraktale
- www.chrisdragan.org - strona Chrisa Dragana, wiele przykładowych programów w asemblerze (MMX)

- www.azillionmonkeys.com/qed/index.html - znakomite artykuły nt. optymalizacji kodu (MMX, Pentium)
- asmjournal.freeservers.com - Assembly Programming Journal, magazyn traktujący o różnych aspektach programowania w asm, optymalizacja kodu z bibliotek C, pisanie w asm pod graficznymi powłokami systemów Unix, programowanie gier z wykorzystaniem DirectX, oraz wiele ciekawostek
- nasm.2y.net - oficjalna strona darmowego asemblera NASM (Windows, Unix)
- www.numega.com - SoftIce, debugger pozwalający dokładnie analizować kod dowolnego programu, zarówno na poziomie HLL oraz na poziomie kodu asemblera, niezastąpiony w wykrywaniu bugów

8. O autorze

Bartosz Wójcik – autor zabezpieczenia oprogramowania PELock dla programistów, wykorzystującego szyfrowania pliku aplikacji w połączeniu z systemem kluczy licencyjnych