# Asembler 68k w przykładach - część 1

Asman



Nauka asemblera przeważnie zaczyna się od poznania rozkazów procesora albo od tego, w jaki sposób komputer przechowuje dane. Oczywiście jest to ważne, ale czasami trochę może być nużące dla czytelnika. Przydługie wywody potrafią skutecznie zniechęcić wiele osób, a brak choćby krótkich, oczywistych przykładów, potęguje to jeszcze bardziej. Dlatego też ja proponuję inne podejście - dużo przykładów przeplatanych teorią, mając nadzieję, że ten sposób zwiększy się liczbę osób znających asembler procesora 68000, a to z kolei przełoży się, mając cichą nadzieję, na liczbę nowych produkcji. Od czytelnika będę wymagał podstawowej znajomości Amigi, odrobiny matematyki i wytrwałości. Nie należy się zrażać, jeśli coś nie działa bądź czegoś nie rozumiemy. Zawsze można zapytać na forum PPA. Na początku dodam, że będziemy używać Asm-One. Nic nie stoi na przeszkodzie, aby używać AsmPro bądź innego asemblera należącego do tej rodziny (Asm-One, AsmPro, Trash'mOne). Jeśli ktoś czuje się na siłach, to może spróbować z innymi asemblerami, takimi jak Barfly, vasm, DevPac. W przypadku dwóch pierwszych musimy użyć następującej kombinacji: edytor tekstu zapisujący czysty ASCII i wspomniany asembler.

Zaopatrujemy się w Asm-One, pobierając go z Aminetu. Użytkownicy z kickstartem w wersji mniejszej niż 2.04 muszą się posiłkować starszą wersją 1.20, którą to bez problemu znaleźć można tutaj. Rozpakowujemy go i instalujemy. Przyda się też biblioteka ReqTools. Sam asembler uruchamiamy klikając w jego ikonę bądź wpisując w shellu jego nazwę. Na początku przywita nas komunikat o wyborze typu pamięci. Wpisujemy literkę P i naciskamy klawisz Enter. W ten sposób wybraliśmy pamięć typu public. Naszym oczom pokaże się komunikat o wyborze ilości pamięci, która to będzie przeznaczona jako robocza. Na początku naszej zabawy wystarczy około 100 kilobajtów. Użytkownicy z bardzo mała ilością pamięci w jaką wyposażony jest komputer powinni wybrać jeszcze mniej - od 20 do 50 kilobajtów. Jako pierwsze ćwiczenie proponuje powybierać bardzo małe ilości, włącznie z jednym kilobajtem i spróbować z przykładami z naszego kursu. Gdy pojawi się komunikat o niewystarczającej pamięci ("Workspace Memory Full"), to niechybny znak, aby zarezerwować więcej pamięci podczas kolejnego uruchomiania programu.

Ustawienia Asm-One proponuję skonfigurować tak, jak na podanym obok obrazku. Aby je ustawić, prawym przyciskiem myszy wybieramy w menu Assembler -> Preferences -> Environment. Zaznaczamy tylko następujące checkboxy:

w General Parameters: ReqTools Library, UpdateCheck, Safety, Clipboard Support W Monitor/Debugger zaznaczamy wszystkie z wyjątkiem Enable/Permit W Editor: Auto Indent, CTRL up/down, Keep x Nie będę się rozpisywał po co i dlaczego ustawiamy te opcje, bo zależy mi, aby jak najszybciej przejść do przykładów.

Przechodzimy teraz do menu, wybieramy Assembler -> Preferences -> Assembler i tu wybieramy Rescue, AutoAlloc, Debug, Progress Indicator, Progress by line, DS Clear, Comment, Processor Warn a jako procesor wybieramy 68000. Zapisujemy ustawienia i jesteśmy prawie w domu, bo pozostaje nam z grubsza opisać jak się poruszać po Asm-One. Program posiada kilka trybów pracy. Jeden z nich to tryb linii poleceń i to tu powinniśmy być, o ile nie wciskaliśmy na przykład klawisza ESC. Drugi tryb to edycja kodu źródłowego. Między tymi dwoma trybami przechodzimy wciskając wspomniany już klawisz ESC bądź też używamy skrótów albo zwyczajnie wybierając odpowiednie pozycje w menu (Edytor/Commandline). Aby nie komplikować sprawy, poznamy tylko kilka podstawowych rzeczy związanych z Asm-One. Najważniejsze to kompilacja kodu źródłowego i do tego służy komenda A (i zatwierdzenie klawiszem Enter) w linii poleceń. Jeśli kompilacja przebiegła bez problemów, to uruchomienie wykonujemy za pomocą komendy J (i zatwierdzenie klawiszem Enter). W miarę nauki poznamy więcej przydatnych komend. Teraz proponuję, aby czytelnik zapoznał się z Asm-One i powciskał kilka razy ESC i pouzywał poznanych komend.

Przejdźmy do trybu poleceń i wpiszmy ?10. Po wciśnięciu klawisza Enter zobaczymy szereg niezrozumiałych znaczków. Są tam między innymi cyfry, kropki, ale także znak dolara i procenta. ? - to komenda Asm-One, która zamienia go w bardzo pożyteczny kalkulator. Z samej lewej strony mamy zapis wyniku w systemie szesnastkowym (zostanie to wyjaśnione nieco dalej), potem widzimy normalny wynik, czyli w systemie decymalnym, który to używamy

# Asembler 68k w przykładach - część 1

Asman



na co dzień. Dalej są cztery kropki w cudzysłowie a na końcu mamy reprezentację dwójkową. Wszystko to wygląda i brzmi bardzo skomplikowanie, dlatego proponuję, aby czytelnik sam popróbował swych sił i powpisywał różne liczby bądź inne konstrukcje matematyczne, takie jak dodawanie, odejmowanie, mnożenie bądź dzielenie. Wspomniana reprezentacja dwójkowa, to nic innego jak sposób, w jaki przedstawiamy liczbę. Na tę chwilę mam na myśli liczbę naturalną, a przedstawiamy ją za pomocą tylko dwóch cyfr: 0 i 1. Dlaczego dwóch cyfr a nie trzech? Dwójkowa reprezentacja w sposób naturalny związana jest z komputerami, które to są urządzeniami elektronicznymi służącymi do przetwarzania informacji. Najłatwiej jest przyjąć, że zero oznacza brak prądu a jedynka, że on jest. Analogia z żarówką też wydaje się być tutaj odpowiednia: zgaszona lub świecąca. W ten sposób właściwie zdefiniowaliśmy najmniejszą porcję informacji, jaką komputer może przetworzyć, a jest to bit przyjmujący dwa stany: 0 i 1. Jak najbardziej działa tu wciąż odniesienie do żarówki i mówimy, że bit jest zapalony, czyli ma wartość jeden. Zgaszając bit ustalamy jego wartość na zero. Grupując bity obok siebie tworzymy ciągi bitowe, na przykład 00001010. W nim występują tylko zera i jedynki (dygresja - najciekawsze w tym wszystkim jest to, że mówimy o systemie dwójkowym (bądź binarnym), a cyfra dwa nie pojawia się w nim). Skoro jeden bit może przyjąć dwie wartości 0 i 1, to dwa bity mogą przyjąć 4 wartości 00, 01, 10, 11. Trzy bity to osiem wartości 000, 001, 010, 011, 100, 101, 110, 111 i tak dalej. Jako proste ćwiczenie, niech czytelnik sam wypisze wszystkie możliwe 4 bitowe wartości. Widać, że ilość możliwych kombinacji danego ciągu bitowego to potęgi dwójki. W przypadku jednego bitu jest to 2^1, dla dwóch bitów to 2^2 i tak dalej.

Chwileczkę, ale skąd mamy wiedzieć że 111 to 7 dwójkowo? Przecież to liczba 111 dziesiętnie. Trzeba oznaczyć w jakiś sposób, że mamy do czynienia z reprezentacją dwójkową. Przyjęło się, że liczbę dwójkową podajemy z przedrostkiem w postaci znaku procenta. I tak %11, to nic innego jak liczba 3. Warto się przekonać w AsmOne, że komenda ? daje sobie radę także z dwójkowymi liczbami, wystarczy wpisać ?%100. Powracając do wyniku pokazywanego przez AsmOne po użyciu komendy ? zauważymy, że właśnie ostatnia czwarta kolumna zaczyna się od znaku procenta, czyli mamy do czynienia z reprezentacją dwójkową, tylko ciąg cyfr jest dość długi i dla ułatwienia, co jakiś czas (co 8 bitów) mamy kropkę. Jak łatwo policzyć ciąg ten jest 32-bitowy, bo takie też jest ograniczenie tej komendy, a możemy się o tym przekonać wpisując tajemniczy, jak na razie, przykładzik ?$fffffffff i powinien ukazać nam się komunikat '** Out of Range 32 bit'. Do binarnego sposobu zapisu liczb jeszcze wrócimy, a teraz opiszemy pierwszą kolumnę w wyniku wyżej wspomnianej komendy, służącej za kalkulator. Jest to zapis w systemie szesnastkowym (heksadecymalnym). Od razu nasuwa się pytanie, dlaczego aż szesnastkowy a nie jakiś inny. Już spieszę z odpowiedzią - ten format zapisu bardzo ułatwia zapis liczb binarnych. Trzeba przyznać, że wpisywanie ciągów już 8 bitowych to sporo pisaniny, nie wspominając już od 32 bitach i większej ilości. Przyjrzyjmy się tabelce wszystkich wartości dla ciągu 4-bitowego.

0 - 0000 - 0 1 - 0001 - 1 2 - 0010 - 2 3 - 0011 - 3 4 - 0100 - 4 5 - 0101 - 5 6 - 0110 - 6 7 - 0111 - 7 8 - 1000 - 8 9 - 1001 - 9 10 - 1010 - A 11 - 1011 - B 12 - 1100 - C 13 - 1101 - D 14 - 1110 - E 15 - 1111 - F

Z lewej strony są wartości dziesiętne, potem reprezentacja bitowa (czyli dwójkowa) i na końcu kody szesnastkowe, bo warto dodać, że w systemach powyżej dziesiętnego musimy sobie poradzić z cyframi od 10 w górę, Poradzono sobie w ten sposób, że zamiast tworzyć jakieś nowy cyfry, użyto liter alfabetu. Jak widać w tabelce liczba dziesięć to litera A (nie ma znaczenia czy duża, czy mała). Zamiast pamiętać cztery cyfry, to wystarczy pamiętać jeden symbol. Na początku może to się wydawać trudne i proponuje po prostu nauczyć się na pamięć takiej tabelki. Świetnie! Potrafimy skracać zapis ciągu 4-bitowego, a co z większymi? Tniemy dany ciąg na podciągi 4 bitowe licząc od prawej i zamieniamy zgodnie z tabelką na symbole szesnastkowe. Niestety nie można w ten sposób przekonwertować ciągu binarnego do postaci dziesiętnej. Jako ćwiczenie zamienimy ciąg 16-bitowy 1011101010111010. Dzielimy go na podciągi mamy 1011.1010.1011.1010 i podpierając się tabelką ostatecznie mamy $BABA.

Po cichu nic nie mówiąc dodałem przedrostek $ przed liczbą szesnastkową, bo tak zwyczajowo się robi. W tym miejscu chciałbym napomknąć, że używając różne narzędzia dla koderów (i nie tylko dla nich), można się spotkać z taką sytuacją, że szesnastkowy zapis traktowany jest jako naturalny i nie poprzedzamy go znakiem dolara. Na przykład w

# Asembler 68k w przykładach - część 1

Asman



HrtMon jest taka możliwość po użyciu odpowiedniej komendy (hexlock). Wtedy to można się, delikatnie mówiąc, zdziwić dlaczego dostajemy "nieprawidłowe" wyniki przy użyciu kalkulatora w HrtMonie.

Została nam do opisania trzecia kolumna (drugiej nie będziemy opisywać, bo jest to zapis dziesiętny). Widzimy tam szereg kropek, które są umieszczone w cudzysłowie. Jest to reprezentacja tekstowa. Jeśli użyjemy tabelki kodów ASCII dla ciągu 8-bitowego, to każdy taki ciąg będzie przedstawiony za pomocą jednego znaku, a że mamy 32 bity to tych znaków będzie równo 4. Wpisując na przykład ?$66616465 otrzymamy "fade". Co ciekawe w kalkulatorze można bez problemu wpisać ?"pies" i otrzymamy także odpowiedni wynik. Przy próbie wpisania dłuższego ciągu Asm-One pokaże ostatnie 4 znaki z użytego napisu, co przyznam osobiście mnie trochę zaskoczyło, bo spodziewałem się błędu. Namawiam gorąco czytelnika, aby sam spędził kilka ładnych chwil i pobawił się kalkulatorem w Asm-One.

Zanim przejdziemy do przykładu, który będziemy mogli zasemblować i uruchomić, bez wnikania czym jest proces asemblacji, musimy jeszcze powrócić do bitów. W danym ciągu bitowym rozróżniamy bit najmłodszy, to pierwszy licząc od prawej. Jest on także bitem najmniej znaczącym, który to samej wartości wnosi mało, bo dzięki niemu wiemy czy liczba jest parzysta, czy nie. Mamy także najstarszy bit - ten najbardziej z lewej strony - zwany też bitem najbardziej znaczącym, bo i wartość jego jest największa w ciągu bitowym. Mając dany ciąg bitowy możemy przy odrobinie wprawy przekształcić go w znaną nam liczbę dziesiętną. Konwersja jest bardzo prosta - każdy bit ma wagę 2^numer bitu, przy czym liczymy od prawej do lewej. I tak przykładowo ciąg 1101 należy skonwertować w następujący sposób. Na pierwszym miejscu z prawej jest 1 i mnożymy ją przez 2^0, potem mamy 0*2^1, dalej 1*2^2 i na końcu 1*2^3, sumując otrzymamy 1+0+4+8 = 13. Warto trochę poćwiczyć na przykładach przed przystąpieniem do konwersji odwrotnej, czyli jak z normalnej, czyli dziesiętnej liczby, zrobić jej postać binarną. Dla krótkich ciągów, najlepiej jest popatrzeć w naszą tabelkę. Dla dłuższych, trzeba użyć prostej matematyki i zacząć dzielić liczbę przez 2. Jeśli dzieli się przez dwa to resztę równą zero zapisujemy i powtarzamy proces. Na przykład mamy 14 i zamienimy ją na binarną.

14/2 = 7 R 0 7/2 = 3 R 1 3/2 = 1 R 1 1/2 = 0 R 1

I licząc od dołu do góry tylko reszty otrzymujemy liczbę dwójkową, czyli 1110. Warto teraz przed dalszym czytaniem potrenować trochę z różnymi liczbami. Zamiana szesnastkowego na dziesiętny wymaga zapamiętania kolejnych potęg liczby 16. Biorąc jeszcze raz F0, mamy F * 16^1 + 0 * 16^0 = 15 * 16 + 0 = 240. Jak widać nie jest to trudne. Aby zamienić dziesiętną na szesnastkową (heksadecymalną), to dzielimy liczbę przez maksymalną potęgę 16, zapisujemy iloraz i resztę i powtarzamy ten proces. Na przykład konwersja 194 na hex.

194/16 = 12 R 2 2/1 = 2

Patrzymy do tabelki, zamieniamy 12 na C i oto wynik - C2. Jak widać konwersja z dziesiętnego na szesnastkowy jest bardziej skomplikowana. Powróćmy do bitów i naszej tabelki. Mamy w niej ciągi 4-bitowe i jeden taki ciąg dla ułatwienia nazywamy nibble. Wszystko po to, aby nie pisać o 4-bitowym ciągu. Możemy ustawić obok siebie dwa nibble, najpierw starszy a potem młodszy, tworząc ciąg 8 bitowy, który to tworzy bajt. Uff, tyle musieliśmy powiedzieć, aby dotrzeć do niego. Bajt ma 256 możliwych kombinacji, jego reprezentacja heksadecymalna zajmuje dwa znaki (na przykład A0 to 160 dziesiętnie).

Czas na przykład. Przejdźmy do edytora i wpiszmy bajt o wartości 15.

```
    dc.b %00001111   ; (bin) = 15 dziesiętnie (dec), $0f szesnastkowo (hex)
```

Ważne, aby przed dc.b umieścić tabulację (wciskając klawisz TAB raz bądź dwa razy) bądź kilka spacji (4 spacje to jeden znak tabulacji). dc.b to dyrektywa asemblera i nakazuje mu umieszczenie bajtu o wartości 15 (gdzie 15 zapisaliśmy w systemie dwójkowym). Dalej po tabulacji mamy znak średnika mówiący asemblerowi, że zaczynamy komentarz, ciągnący się do końca linii. Niewiele napisaliśmy a bardzo dużo trzeba wyjaśnić.

# Asembler 68k w przykładach - część 1

Asman



Używamy Asm-One jako asemblera, ale cóż to za stwór ten cały asembler. Należy to rozumieć jako tłumacz, który jedne rzeczy (w tym przypadku kod źródłowy) zamienia na inną rzecz (kod maszynowy, zrozumiały dla procesora). Kod źródłowy to wszystko to, co wpiszemy w trybie edycji, czyli innymi słowy jest to dowolny plik tekstowy, który możemy rozumieć jako bardzo długi ciąg bitowy (o ile powrócimy do reprezentacji tekstowej i wyobrazimy sobie dostatecznie długi napis). Kod maszynowy to także ciąg bitowy tylko zrozumiały przez procesor, który zawsze porównuje do wioskowego głupka, siedzącego i czekającego na żarcie, czyli ciągi bitów, aby zrobić z nimi niesamowite rzeczy. Jasne jest, że taki język (kod) maszynowy nie może być przypadkowym ciągiem. Jest on ściśle zależny od danego typu procesora, w naszym przypadku mamy Motorolę 680x0. Ten kawałek krzemu sam z siebie za dużo nie zdziała, bo te ciągi bitowe i to nie byle jakie, musi z jakiegoś magicznego miejsca pobrać i ewentualnie po przeróbce (nie wiem dlaczego, ale to zawsze kojarzy mi się z wydalaniem ciągów) gdzieś oddać. To pamięć komputera jest właśnie takim obszarem, gdzie leżą niesamowite ilości ciągów bitowych, gotowych do schrupania przez procesor. Sama pamięć to także jeden bardzo długi ciąg bitowy, Oczywiście dla ułatwienia pogrupowano bity i mówimy, że składa się on z bajtów. Dodatkowo, aby się nie pogubić owe bajty ponumerowano, tak zwyczajnie od zera, mówimy wtedy o adresach tychże bajtów. Wszystko to wydaje się może trochę skomplikowane, ale po czasie stykania się i używania nowych nazw i pojęć, wyda nam się to oczywiste. Proces tłumaczenia nazywamy translacją bądź kompilacją, zatem zróbmy to dla tej jednej prostej wpisanej linijki, używając do tego celu komendy A. Jeśli wszystko poszło bez problemów, to powinniśmy ujrzeć

```
Pass 1 Pass 2 No Errors Assembly Time: 00:00:00
```

A gdy zrobilibyśmy błąd, to czego powinniśmy oczekiwać i jak temu zaradzić? Pierwszy z nich to błąd oznaczający nieznaną instrukcję (**Ilegal Operator), który ukaże się nam, gdy nie będzie spacji przed dyrektywą dc.b, bądź wpiszemy niepoprawną instrukcję. Drugi błąd to gdy pomylimy się i wpiszemy zamiast ośmiu bitów, dziewięć i więcej (** Out of Range 8 Bit), na przykład dc.b %010100101. Kolejnym błędem, który może się pojawić to (** No Operand Space Allowed), czyli niedozwolona spacja, a uzyskałem go wstawiając spację między zerem a jedynką w ciągu bitowym.

Co zyskaliśmy po asemblacji naszego przykładziku? W zasadzie to nic, poza tym, że umiemy zapisać bajt gdzieś w pamięci, choć nawet jeszcze nie wiemy gdzie to dokładnie jest. Utarło się w asemblerach, że dane wpisujemy za pomocą dyrektywy dc (defince constant) i po kropce dajemy jej rozmiar. W następnym przykładzie umieścimy sobie trzy bajty o wartości 16.

```
dc.b 16    dc.b %1000    dc.b $f
```

Ale co, gdy chcemy umieścić tych danych trochę więcej? Przecież to tyle pisania. Na szczęście można to zapisać oszczędniej:

```
dc.b   16,%1000,$f
```

Poznaliśmy bajt, a teraz nadszedł czas na poznanie słowa (word), reprezentującego ciąg 16-bitowy bądź, jak komu wygodnie, dwa bajty. Tu minimalną wartością będzie 0 a maksymalną 2^16=65535. Najbardziej znaczącym bitem będzie bit 15. A teraz przykład jak zapisać słowo w Asm-One.

```
dc.w   $0012
```

Podobnie jak w przypadku bajta, tylko że zamiast dc.b piszemy dc.w. Poza wyżej wymienionymi błędami kompilacji, dla słowa będzie jeszcze **Out of Range 16 bit oraz błąd bardzo trudny do wychwycenia, bo kompilacja przebiega bez problemu. Zastanówmy się przez chwilę, co może się stać, gdy będziemy mieli na myśli bajt, a wpiszemy dc.w? Kompilacja przecież się powiedzie. Pozostaje tylko liczyć na ostrożne programowanie. Ostatni typ danych, jaki poznamy

# Asembler 68k w przykładach - część 1

Asman



to długie słowo (long), składające się z dwóch słów. Aby je napisać, używamy dyrektywy dc.l. Na przykład:

```
dc.l   $12345678
```

Co ciekawe, ten sam rezultat możemy osiągnąć używając różnych rozmiarów danych, na przykład dc.b $50,$51 oznacza to samo co dc.w $5051, ale należy pamiętać że dc.b 10,11 jest równe dc.w 2571. Proszę zwrócić uwagę, że tu nie ma dolara mówiącego, że mamy do czynienia z heksadecymalną reprezentacją, tylko normalne liczby. Aby to sprawdzić, polecam użyć kalkulatora.

Jak wspominałem wyżej, procesor operuje tylko na ciągach binarnych. Są one dosyć specyficzne, a taki zjadliwy ciąg nazywamy kodem maszynowym. Aby ułatwić sobie życie, kod maszynowy zastąpiono przez skróty słów, bardziej zrozumiałych dla człowieka. Przykładem niech będzie instrukcja move, która przenosi coś z jednego miejsce w drugie. Zatem, aby się porozumieć z procesorem, niezbędne jest teraz nauczenie się mnemoników - tak nazywamy te tajemnicze skróty słów, czyli instrukcji danego procesora. A sam asembler tłumaczy (kompiluje) mnemoniki na odpowiednie ciągi binarne. Nic nie stoi na przeszkodzie, aby pominąć mnemoniki i posługiwać się samymi ciągami binarnymi, używając do tego dyrektywy dc. Prześledzimy to na najprostszym przykładzie, który nic nie robi. W edytorze wpiszmy.

```
rts
```

Przejdźmy do linii poleceń i wykonajmy kompilację. Użyjemy teraz polecenia J, które uruchamia skompilowany przykład. Naszym oczom powinno ukazać się na pierwszy rzut oka dosyć dziwna tabelka - jest to status procesora i pokazuje on co po wyjściu z naszego przykładu zostało zmienione. Na razie nie zaprzątajmy sobie tym głowy, dodamy tylko, że są tam między innymi rejestry procesora. Zapiszmy nasz przykład za pomocą komendy W. Ten sam przykład zapiszmy za pomocą dyrektywy dc.

```
dc.w   $4e75   ;rts
```

To wszystko w pierwszym odcinku, zachęcam do eksperymentowania, zadawania pytań na forum PPA w razie wątpliwości.