1. Asembler i wstawki asemblerowe w C

Asembler jest językiem programowania na poziomie sprzętowym. Do zalet korzystania z asemblera należy zaliczyć:

- większą szybkość działania programu napisanego w asemblerze,
- krótki kod wynikowy programu napisanego w asemblerze,
- możliwość z zaprogramowaniem operacji związanymi ze sprzętem (w języku wyższego poziomu może to być niemożliwe).

Znajomość programowania w asemblerze ma również duże znaczenie poznawcze, gdyż pozwala poznać działanie komputera od wnętrza. W rozdziale podano tylko wybrane zagadnienia programowania asemblera, czytelnik pragnący pogłębienia wiadomości może skorzystać z odpowiednich pozycji literatury [1, 3-4, 8-9].

1.1. Powiązanie instrukcji asemblera z budową wewnętrzną komputera

Program wynikowy (binarny) zapisany jest w systemie dwójkowym i składa się z rozkazów. Dla każdego procesora zdefiniowana jest ***lista podstawowych rozkazów***. Na bazie tych rozkazów powstaje całe oprogramowanie komputerowe. Zatem rozkaz jest pewną elementarną operacją, realizowaną przez dany komputer.

Każdemu typowi rozkazu procesora przypisano pewien symbol, mnemonik (zamiast kodu rozkazu podanego w postaci dwójkowej). Otrzymany w ten sposób język programowania nazywany jest ***językiem adresów symbolicznych***. Dla przykładu podano poniżej zapis rozkazu jednobajtowego w postaci dwójkowej i symbolicznej.

1110 1100	IN AL,DX

Rozkaz ten powoduje załadowanie bajtu z portu urządzenia zewnętrznego, o adresie podanym w rejestrze DX. Rozkazy procesora mogą mieć różną długość, poniżej podano przykład rozkazu o długości dwóch bajtów.

1110 0100 1010 0010	IN AX,C2H

Rozkaz ten, podobnie jak poprzedni, powoduje pobranie zawartości portu urządzenia zewnętrznego. Adres tego portu jest podany w drugim bajcie rozkazu i wynosi C2 w zapisie szesnastkowym. Można przyjąć, że rozkaz składa się z następujących pól:

- ***pola operacji***, w którym określony jest typ wykonywanej operacji.
Pole operacji zawiera kod operacji, który w sposób jednoznaczny identyfikuje operację, która ma być wykonana. Operacjami mogą być dodawanie, mnożenie, skok, itp.;

- ***pól operandów***, w których określany jest dostęp do wartości operandów.
Do wykonania operacji niezbędne są operandy, przy czym występują co najwyżej dwa operandy wejściowe. Wyniki operacji mogą być zapisane do operanda wyjściowego. Każda operacja wymaga pewnej liczby operandów, operacja dodawania i mnożenia - dwóch, operacja skoku - jednego, a instrukcja powrotu funkcji - żadnego. Pola operandów mogą wskazywać na rejestry, zawierać adresy pamięci, pod którymi przechowywane są operandy, lub bezpośrednio mogą zawierać wartości operandów. Identyfikacja ośmiu podstawowych rejestrów może być dokonana przy użyciu trzech bitów. W przypadku podania adresu operanda w pamięci konieczne są dwa bajty, co znacznie wydłuża rozkaz. Stanowi to przyczynę występowania w operacjach tylko jednego operanda z pamięci;

- ***pola trybu***, w którym określana jest interpretacja operacji i operandów.
Pola trybu są złożone z kilku pól. Jedno z tych pól podaje typ pamięci, w którym przechowywane są operandy. Dla operacji z dwoma operandami, obydwa operandy mogą się znajdować w rejestrach, albo jeden operand znajduje się w rejestrze a drugi w pamięci. Następne pole trybu zawiera kierunek przekazywania danych, to znaczy określa który z operandów jest operandem wejściowym a który wyjściowym. Kolejne pole wskazuje czy w rozkazie występują dane bezpośrednie, to znaczy czy rozkaz zawiera wartość operanda. W kolejnym polu trybu podawane jest, czy stosowane jest pośrednie adresowanie operandów w rejestrach. To znaczy określane jest czy zawartość rejestru zawiera wartość operanda, czy jego adres w pamięci;

- ***długość rozkazu***, w którym podawana jest liczba bajtów zajmowanych przez rozkaz.
W przypadku przeznaczenia dwóch bitów na długość rozkazu, możemy mieć rozkazy o długości do czterech bajtów. Często w rozkazach długość rozkazu podawana jest w sposób niejawny.

Język adresów symbolicznych nosi nazwę ***asembler*** i ma następujące trzy własności:

- każdemu rozkazowi z listy podstawowych rozkazów przypisana jest instrukcja symboliczna,
- kody operacji, nazwy rejestrów, argumenty rozkazu zapisywane są symbolicznie,
- adresy komórek pamięci podawane są za pomocą nazw symbolicznych.

Należy zwrócić uwagę, że program tłumaczący program napisany w asemblerze na program maszynowy (wewnętrzny) nosi nazwę również asembler. W literaturze polskiej spotyka się rozróżnienie nazw: program tłumaczący nazywany jest asemblerem (nazwa rozpoczyna się od małej litery) a język adresów symbolicznych Asemblerem (nazwa rozpoczyna się od dużej litery). W opracowaniu tym będziemy używać nazw rozpoczynających się od małych liter. Natomiast nazwa konkretnego asemblera będzie pisana zgodnie z nazwą przyjętą przez producenta asemblera, np. Turbo Assembler. W opracowaniu będą omawiane instrukcje asemblera firmy Borland o nazwie Turbo Assembler.

Instrukcje języka asembler składają się z następujących członów:

PoleEtykiety PoleOperacji PoleArgumentów PoleKomentarza

W każdej instrukcji musi wystąpić pole operacji.
Na przykład instrukcja:

```
dodanie:    ADD DX,2     ; Dodanie 2 do rejestru DX
```

powoduje dodanie liczby 2 do zawartości rejestru DX. Komentarz umieszczony jest po średniku. Przed każdą instrukcją może wystąpić etykieta, w przykładzie jest nią nazwa „dodanie”. Etykietę stosuje się zwykle wtedy, gdy z innej części programu dokonywany jest skok do tej instrukcji.

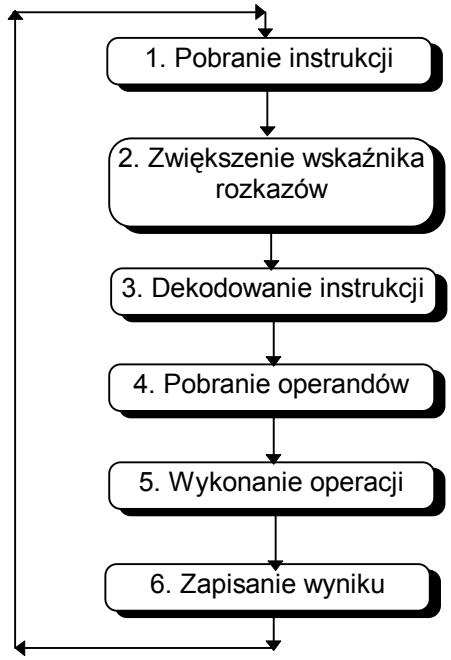
Fazy wykonywania instrukcji

Podstawowymi układami komputera są pamięć i procesor. Z pamięci pobierane są rozkazy do wykonania przez procesor oraz dane będące operandami operacji. Procesor nie wykonuje operacji bezpośrednio na danych zapamiętanych w pamięci komputera. Dane te są ładowane do szybkich pamięci procesora, zwanych rejestrami. Operacje wykonywane są na danych zawartych w rejestrach, wynik zapisywany jest w jednym z rejestrów. Wynik z rejestru przesyłany jest zwykle potem do pamięci komputera. Szerokie informacje dotyczące budowy i działania komputerów można znaleźć w [6, 7].

Wykonanie programu polega na powtarzającym się wykonaniu kolejnych faz instrukcji procesora przedstawionych na rysunku 1.1. Poniżej został przedstawiony opis poszczególnych faz.

1. Pobranie instrukcji

Instrukcje tworzące program zapisywane są do kolejnych komórek pamięci. Aktualna instrukcja pobierana jest z pamięci spod adresu wskazanego przez wskaźnik instrukcji (IP-Instruction Pointer). Instrukcja jest ładowana do rejestru, zwanego rejestrem instrukcji (IR - Instruction Register).



Rys. 1.1. Fazy wykonania instrukcji procesora

2. Zwiększenie wskaźnika rozkazów

Wskaźnik rozkazu (IP) wskazuje na instrukcję w pamięci, która ma być wykonywana. Po załadowaniu instrukcji do rejestru instrukcji znana jest długość instrukcji liczona w bajtach. Zatem wskaźnik rozkazu zostanie powiększony o długość pobranego rozkazu.

3. Dekodowanie instrukcji

Pole operacji instrukcji w sposób jednoznaczny identyfikuje instrukcję. Pole trybu uszczegóławia typ instrukcji. Poprzez analizę tych dwóch pól określone są akcje niezbędne do wykonania instrukcji.

4. Pobranie operandów

Do wykonania operacji muszą być dostępne operandy. Operandy, w zależności od trybu instrukcji, mogą się znajdować w rejestrach, w pamięci RAM lub jako dane umieszczane bezpośrednio w instrukcji. Operandy są kopiowane do dwóch rejestrów wejściowych arytmometru. Jeżeli operandy znajdują się w rejestrach ogólnego przeznaczenia, to są kopiowane poprzez magistralę procesora z tych rejestrów do rejestrów źródłowych arytmometru. W przypadku, gdy operand znajduje się w pamięci RAM, to jest ładowany z pamięci do rejestru źródłowego. W tym przypadku adres operanda jest częścią instrukcji i jest przechowywany w rejestrze instrukcji. Jeżeli operandy znajdują się w instrukcji, to oznacza, że są przechowywane w rejestrze rozkazów. Operand taki ładowany jest z rejestru rozkazów do rejestrów wejściowych arytmometru.

5. Wykonanie operacji

Wszystkie potrzebne operandy zostały skopiowane do rejestrów źródłowych. W arytmometrze wykonywana jest operacja, a wynik zapisywany jest do rejestru wynikowego.

6. Zapisanie wyników

Po wykonaniu operacji jej wynik zapisywany jest do rejestru wynikowego. Wynik ten będzie zapisany do miejsca wskazanego przez operand przeznaczenia, określonego w instrukcji.

7. Przejście do wykonania fazy 1.

Po wykonaniu wszystkich tych faz nastąpi przejście do wykonania fazy 1.

1.2. Operacje na rejestrach procesora

Przy wykonywaniu różnych instrukcji szczególną rolę odgrywają rejestry procesora. Sposób korzystania z rejestrów nie jest ściśle zdefiniowany, istnieją jednak pewne umowne reguły, które są stosowane przez programistów.
Szesnastobitowe rejestry mikroprocesorów 8086/8088 (rys. 1.2), dostępne programowo, dzieli się na :

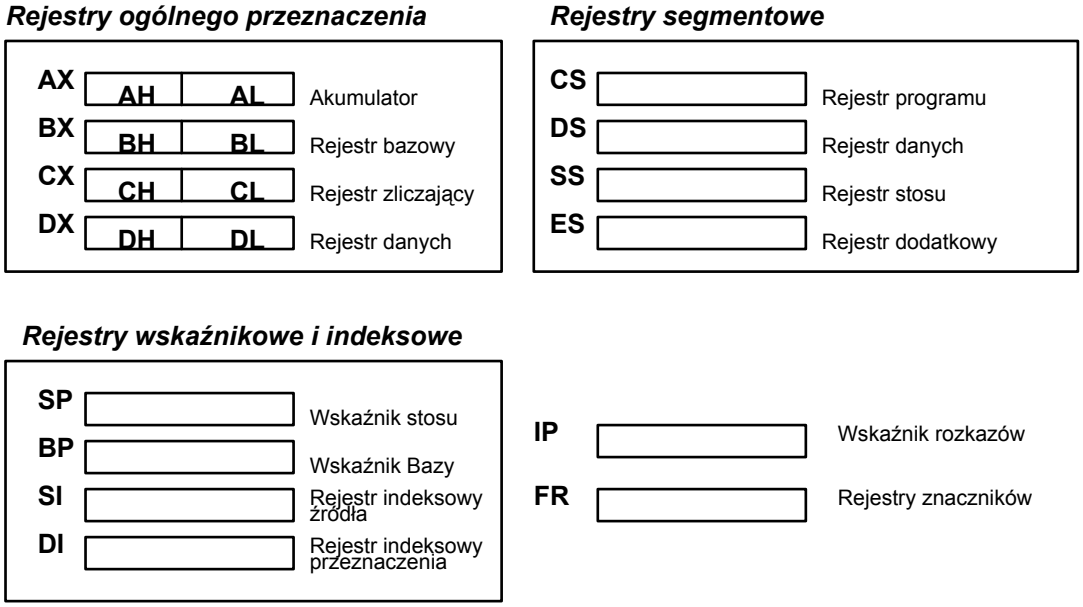
rejestry ogólnego przeznaczenia AX, BX, CX, DX,

rejestry wskaźnikowe i indeksowe SP, BP, SI, DI,

rejestry segmentowe CS, DS, ES, SS,

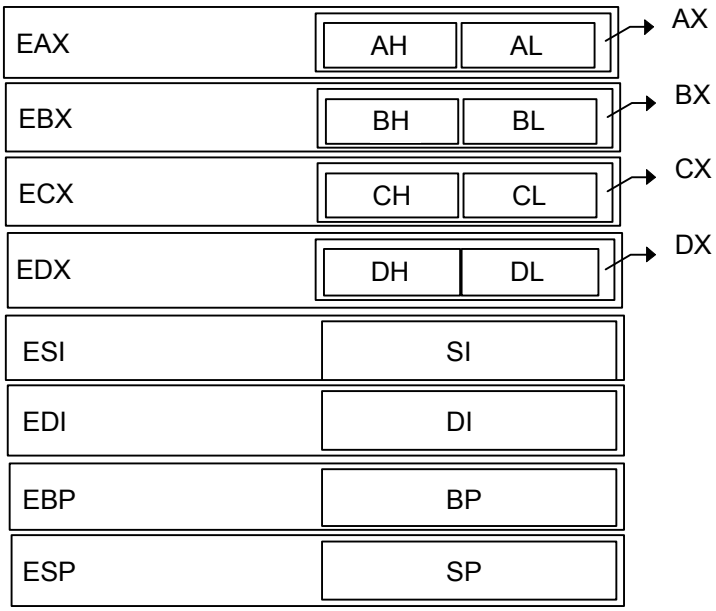
wskaźnik rozkazów IP,

rejestr znaczników FLAGS.



Rys. 1.2. Rejestry mikroprocesora 8086/88

W procesorach 32 bitowych pojawiły się nowe rejestry, niektóre rejestry pozostały bez zmian, inne zostały powiększone do 32 bitów. Rejestry ogólnego przeznaczenia używane w procesorach 16 bitowych zostały powiększone do 32 bitów. Aby zapewnić zgodność oprogramowania w tych rejestrach wyróżniono podzbiór rejestrów 16 bitowych (rys. 1.3).



Rys. 1.3. Przykład konstrukcji rejestrów w procesorze 32 bitowym

W celu umożliwienia działania na nowych rejestrach rozszerzono listę rozkazów procesora. Dokładniejsze informacje dotyczące różnic między procesorami stosowanymi w komputerach klasy PC zawarte są w [2, 5].

1.2.1. Rejestry ogólnego przeznaczenia

Rejestry ogólnego przeznaczenia (nazywane również rejestrami roboczymi) wykorzystywane są do przechowywania danych i wykonywania różnych operacji (np.: arytmetycznych, logicznych), ale pełnią też funkcje specjalne, odpowiadające ich nazwom.

Każdy z rejestrów ogólnego przeznaczenia może być również traktowany jako dwa rejestry 8-bitowe. Są to rejestr górny, zawierający bardziej znaczący bajt (High) i rejestr dolny zawierający mniej znaczący bajt (Low).

Rejestrem szczególnym jest akumulator AX. Używany jest do operacji arytmetycznych, logicznych oraz do zapisywania wyników tych operacji.

Przykład:

```
        mov  ax,13      ; umieść w AX daną o wart. 13
        mov  al,0       ; wyzeruj dolny bajt rejestru AX
        mov  ax,[Fix]   ; załaduj daną spod adresu DS:Fix
```

BX (basis register) - rejestr bazowy może być uważany jako dwa 8-bitowe rejestry BH i BL. 16 bitów tego rejestru może być użytych jako część adresu dostępnego obszaru pamięci, tworząc z rejestrem segmentowym pełny adres (SEGMENT:OFFSET) DS:BX.

Przykład:

```
        mov  bx,13
        mov  ax,[bx] ; załaduj daną spod adresu DS:13
```

CX (count register) - rejestr zliczający wykorzystywany też do blokowego przesyłania danych. Może być używany jako dwa 8-bitowe rejestry CH i CL.

Przykład:

```
        xor bx,bx               ; wyzeruj rejestr BX
        mov cx,5                ; ustaw licznik
Znowu:
```

```
mov [word ptr bx], 13   ; traktuj BX jako adres słowa
inc bx                  ; zwiększ BX o 1
loop Znowu              ; wykonaj według licznika 5 razy
```

Instrukcja loop powoduje zmniejszenie wartości rejestru CX o 1 i jeżeli wartość z tego rejestru po wykonaniu tej operacji jest większa od zera, następuje przejście do instrukcji wskazanej etykietą.

DX (data register) - rejestr danych jest jedynym rejestrem, w którym można podać adres portu w rozkazach wejścia-wyjścia. Wykorzystuje się go jeszcze podczas dzielenia i mnożenia.

```
mov  ax,300
mul bx  ; umieść w DX, AX wartość iloczynu AX*BX
```

1.2.2. Rejestry wskaźnikowe i indeksowe

W skład tych rejestrów wchodzą dwa rejestry wskazujące na określony obszar pamięci (rejestry wskaźnikowe) i dwa rejestry indeksowe. Rejestr SP wskazuje na adres szczytowego słowa stosu. Stos jest to fragment pamięci przydzielanej dynamicznie przez programy, w której informacje są przechowywane i pobierane według reguły LIFO (Last In First Out - ostatni przyszedł pierwszy obsłużony). Najważniejszym przeznaczeniem stosu jest zapamiętanie zapisów przy wywołaniu podprogramów, jak: adresy wywołania, parametry, zmienne lokalne.

Stos może być również użyty jako chwilowa pamięć, chociaż nie jest to jego zasadnicze zadanie. Informację zorganizowaną w postaci stosu można zapisać tylko na wierzchołek stosu i odczytać również tylko z wierzchołka. Stos porównuje się do stosu talerzy leżących jeden na drugim. Zapis informacji na stos odpowiada położeniu nowego talerza, odczyt - zdjęciu talerza z wierzchołka stosu. Kompilatory języków programowania przyjmują standardowo stos o wielkości 512 B.

SP (stack pointer) - wskaźnik stosu. Tak jak rejestr BX, ten rejestr może być użyty jako wskaźnik pamięci. Stosowany jest często przy standardowych operacjach zapisu i odczytu stosu, tworząc wraz z rejestrem SS pełny adres danej, ostatnio odesłanej na stos w postaci SS:SP. Rejestr SP jest analogiczny do rejestru IP (który występuje z CS), wskazującego kolejny rozkaz do wykonania.

BP (base pointer - wskaźnik bazy). Używany jest podczas operacji niestandardowych, np. przy pobieraniu parametrów przekazywanych przez stos. Rejestr BP jest stosowany jak gdyby do robienia migawkowych zdjęć bieżącego położenia wierzchołka stosu, by znać dokładnie położenie na stosie określonej informacji.

SI (source index register) - rejestr indeksowy źródła. Podobnie jak BX, może być użyty jako wskaźnik pamięci i jest stosowany w instrukcjach przetwarzających łańcuchy znaków, tworząc wówczas pełny adres DS:SI.

DI (destination index register) - rejestr indeksowy przeznaczenia. Rejestr DI jest bardziej podobny do rejestru SI w tym, że może być użyty jako wskaźnik pamięci i ma specjalne właściwości, gdy jest użyty z instrukcjami przetwarzającymi łańcuchy znaków, tworząc wówczas pełny adres ES:DI.

Z rejestrami BP, SI i DI związane są niektóre tryby adresowania pamięci. Tryby adresowania określają sposób obliczania adresu danej (adresu efektywnego -EA) znajdującej się w pamięci. Dla procesorów 80x86 wyróżniono 12 trybów adresowania.

1. Tryb domyślny: niektóre instrukcje (CBW, MUL, ...) operują na ustalonych rejestrach, przeważnie na akumulatorze (AX, AL). Przykładowo instrukcja

```
CBW
```

przekształca bajt w AL na słowo w AX.

2. Tryb natychmiastowy: wartość danej do załadowania występuje jawnie w instrukcji.

```
MOV AX, 1213H       ; Załaduj liczbę 1213H do rejestru AX
```

3. Tryb bezpośredni: adres danej podany jest w rozkazie.

```
MOV AX, Wynik      ; Załaduj daną spod adresu Wynik do rejestru AX
```

4. Tryb indeksowy: adres danej znajduje się w rejestrze DI lub SI.

```
MOV [SI], AX       ; Prześlij zawartość AX pod adres podany w SI
```

5. Tryb bazowy: adres danej znajduje się w rejestrze BX lub BP. Użycie rejestru BP powoduje przyjęcie rejestru segmentowego SS.

```
MOV AX, [BX]       ; Prześlij daną spod adresu podanego w BX, do AX
```

6. Tryb indeksowy z przemieszczeniem. adres danej jest sumą zawartości rejestru SI lub DI oraz przemieszczenia.

```
MOV AX, [SI+4]          ; Prześlij daną spod adresu SI+4 do AX
```

7. Tryb bazowy z przemieszczeniem:

```
MOV [BP-2], AX     ; Prześlij zawartość AX pod adres BP-2
```

8. Tryb bazowy indeksowany:

```
MOV AX, [BX+DI]    ; Prześlij zawartość komórki spod adresu BX+DI do AX.
```

9. Bazowy indeksowany z przemieszczeniem.

10. Skalowany indeksowy z przemieszczeniem. Skalowanie polega na możliwości pomnożenia zawartości rejestru przez współczynnik 2, 4 lub 8.

11. Skalowany bazowy z przemieszczeniem.

12. Skalowany bazowy indeksowany z przemieszczeniem.

1.2.3. Rejestry segmentowe

Rejestry segmentowe stosowane są do adresowania pamięci operacyjnej. Pamięć o maksymalnej wielkości 1 MB jest dzielona na logiczne segmenty, nie większe niż 64 KB. Każde odwołanie się do pamięci zawiera adres segmentu i przemieszczenia (offset-u) danej w segmencie. Stosowana jest notacja zapisu adresu w postaci adres_segmentu:adres_przesunięcia. Każdy segment zaczyna się od adresu będącego wielokrotnością 16 bajtów. Umożliwia to zapisanie 20 bitowego adresu w 16 bitowym słowie. Przyjmowane jest, że najmłodsze 4 bity adresu początku segmentu równe są 0. System operacyjny MS DOS automatycznie określa zawartość rejestrów CS, ES i SS.

CS (code segment register) - rejestr wskazuje początek 64-KBajtowego bloku pamięci, segmentu kodu, w którym znajdują się kolejne instrukcje do wykonania. Rejestr CS tworzy pełny adres logiczny wraz z rejestrem IP o postaci CS:IP.

DS (data segment register) - rejestr segmentowy danych, wskazuje na początek segmentu danych.

SS (stack segment register) - rejestr stosu, wskazuje na początek segmentu pamięci, w którym jest zdefiniowany stos.

ES (extra segment register) - rejestr dodatkowy, wskazuje na dodatkowy segment danych.

1.2.4. Wskaźnik rozkazów i rejestr znaczników

IP (instruction pointer) - rejestr ten jest wskaźnikiem rozkazów i wraz z rejestrem CS adresuje kolejne rozkazy do wykonania. IP wskazuje adres względem początku segmentu programu (offset).

Znaczniki stanu z rejestru znaczników (Flags Register) informują o różnych cechach związanych z wynikami wykonywanych operacji arytmetycznych lub logicznych. Mikroprocesory Intel 8086/8088 mają 9 znaczników 1-bitowych w 16-bitowym rejestrze flagowym. Każdy z tych znaczników może być ustawiony na 0 lub 1. Siedem bitów tego rejestru nie jest wykorzystane i ich wartość jest nieokreślona.

Wyróżnia się dwie grupy tych znaczników : znaczniki stanu i znaczniki kontrolne.

CF (carry flag) - znacznik przeniesienia, przyjmuje wartość 1, gdy podczas działania nastąpiło przeniesienie z bitu najbardziej znaczącego na zewnątrz lub gdy wystąpiła pożyczka z zewnątrz do bitu najbardziej znaczącego.

PF (parity flag) - znacznik parzystości,

AX (auxillary flag) - wskaźnik przeniesienia pomocniczego,
Z (zero flag) - znacznik zera, przyjmuje wartość 1, gdy wynik działania równy jest zero,
SF (sign flag) - znacznik znaku,
OF (overflow flag) - znacznik nadmiaru, przyjmuje 1, gdy w danym działaniu wystąpił nadmiar,
TF (trap flag) - znacznik pracy krokowej,
IF (interrupt enable flag) - znacznik zezwalający na przerwanie,
DF (direction flag) - znacznik kierunku.

1.3. Wstawki Turbo Assemblera w programie napisanym w języku C, C++

W programie źródłowym napisanym w C można wstawiać instrukcje asemblera. Ciąg instrukcji asemblera, umieszczony w programie źródłowym języka C, nazwiemy wstawką asemblerową. Każdą wstawkę assemblerową rozpoczyna słowo kluczowe asm. Wstawka asemblerowa jest traktowana jako instrukcja programu źródłowego. We wcześniejszych wersjach w pliku zawierającym wstawki asemblerowe należało umieścić wiersz

```
#pragma inline
```

Umieszczenie takiego wiersza stanowiło informację dla kompilatora, że w programie znajdują się instrukcje napisane w asemblerze.

Przykład użycia wstawki asemblerowej:

```
if ( a>5 )
        { asm mov  ah,1;  asm  mov  al,2 }
else
        { asm mov  ah,3;  asm mov  al,4 }
```

Wstawka może mieć postać instrukcji albo deklaracji danych. Jeżeli wstawki mają postać instrukcji, to są umieszczane w segmencie kodu a wstawki w postaci deklaracji są umieszczane w segmencie danych. Wstawki asemblerowe mogą zawierać odwołania do wszystkich zmiennych i etykiet programu, ale nie mogą zawierać własnych etykiet.

W asemblerze różne operacje są zapisywane w bardzo podobny sposób. Rodzaj operacji zależy od kontekstu. Rozkazy języka asembler zmieniają zawartość rejestrów, dlatego przy opisie rozkazów podano, jakie zmiany wywoła rozkaz w rejestrach.

Przykład :

```
MOV    AH, DL        spowoduje załadowanie zawartości rejestru DL do AH,
MOV    AH, 0CH       spowoduje załadowanie rejestru wartością bezpośrednią.
```

Widzimy, że dwie w gruncie rzeczy bardzo odmienne operacje zapisane są przy użyciu identycznego mnemoniku i wedle takiej samej syntaktyki.
Wykaz wybranych instrukcji asemblera;
MOV - ładowanie wartości,
ADD - dodanie wartości,
SUB - odjęcie wartości,
INC - zwiększenie wartości o 1,
INT - wywołanie przerwania o podanym numerze,
JMP - skok do instrukcji z o podanej etykiecie,
CALL - wywołanie funkcji,
RET - powrót do funkcji wywołującej,
PUSH - położenie na stos,
POP - pobranie ze stosu,
LOOP - zmniejszenie o 1 zawartości rejestru CX i skok w zależności od wartości rejestru CX.

Programowanie wyłącznie przy użyciu instrukcji mikroprocesora wymagałoby znajomości adresów oraz zasad działania układów scalonych na płycie głównej. Niemal każdy program musi w pewnym momencie skomunikować się ze światem zewnętrznym, co wymaga znajomości otoczenia mikroprocesora. System operacyjny dostarcza procedury ułatwiające wykonanie określonych

operacji o charakterze czysto sprzętowym (np. wyświetlenie znaku na ekranie, sterowanie dyskiem itp.), poprzez wygenerowanie przerwania (którego parametry należy wcześniej ustawić w rejestrach). Tak więc typowy program, aby w ogóle coś mógł zrobić, będzie musiał bez przerwy korzystać z zasobów komputera za pomocą mechanizmów dostarczonych przez `BIOS` (Basic Input Output System) i `DOS` (Disk Operating System).

Przykład 1.1.

Napisać wstawkę asemblerową, której zadaniem jest wyprowadzenie znaków od a do z. Poniżej podano dwa rozwiązania tego zadania.

```
#pragma inline
void main(void)
{
        int i;

        for( i='a'; i<='z'; i++ )
        {
                asm mov DX, i
                asm mov AH,2
                asm int 21H
        }
        return;
}
```

Jeżeli występuje kilka instrukcji asemblera, to można je zapisać w postaci instrukcji złożonej, poprzedzone symbolem asm:

```
asm {
        /* instrukcje asemblera bez poprzedzającego asm */
}
```

1.4. Instrukcje asemblera sterujące przebiegiem programu

Sterowanie przebiegiem programu realizowane jest w oparciu o instrukcje skoków i instrukcję pętli (loop). Instrukcje skoku dzielimy na instrukcje skoku bezwarunkowego oraz instrukcje skoku warunkowego.

Instrukcja skoku bezwarunkowego

Instrukcja skoku bezwarunkowego jest postaci

```
jmp etykieta
```

Wyróżnia się następujące odmiany instrukcji skoku:

```
jmp SHORT etykieta
```

w tym przypadku adres skoku powinien się znajdować w przedziale od -128 do 127 (zakres liczby jednobajtowej) od instrukcji skoku. Instrukcja ta ma długość dwóch bajtów:

1 bajt	jmp
2 bajt	adres etykiety

```
jmp NEAR etykieta
```

adres skoku powinien być zawarty w przedziale przedstawionym za pomocą liczby dwubajtowej. W tym przypadku instrukcja będzie się składać z trzech bajtów:

1 bajt	jmp

2 bajt	1 bajt adresu
3 bajt	2 bajt adresu

```
jmp FAR etykieta
```

adres skoku może być zawarty w dowolnym miejscu programu. Adres podawany jest jako adres segmentu i adres przemieszczenia. Instrukcja będzie się składać z pięciu bajtów:

1 bajt	jmp
2 bajt	1 bajt segmentu
3 bajt	2 bajt segmentu
4 bajt	1 bajt przemieszczenia
5 bajt	2 bajt przemieszczenia

Instrukcja skoku warunkowego

Instrukcja skoku warunkowego podaje warunek, przy którym skok ma być wykonany. Przy kontroli warunku sprawdzane są przeważnie ustawienia znaczników. Najczęściej stosowane są instrukcje skoku sprawdzające, czy wynik poprzedzającej operacji jest równy zeru, czy nie. Instrukcje

```
je etykieta
jz etykieta
```

są równoważne i powodują skok do instrukcji z podaną etykietą, gdy znacznik zera jest ustawiony. Jeżeli skok ma być wykonany przy niezerowym wyniku, to należy posłużyć się jedną z dwóch równoważnych instrukcji:

```
jne etykieta
jne etykieta
```

W asemblerze nie występują instrukcje będące odpowiednikami instrukcji while, if else, for języka C. Poniżej podano zasady konstrukcji odpowiedników tych instrukcji w asemblerze.

Zapis odpowiednika instrukcji while w asemblerze

Instrukcja while jest postaci

```
while( warunek)
{
      instrukcje;
}
```

Odpowiedni ciąg instrukcji asemblera będzie postaci:

```
znowu:
      or al,al          ; ustawienie znaczników po instrukcji
      jz zakoncz
      ; instrukcje
      jmp znowu
zakoncz:
```

Zapis odpowiednika instrukcji if else w asemblerze

Instrukcja if else w języku C jest postaci:

```
if( warunek )
      instrukcje1;
else
      instrukcje2;
```

Odpowiedni zapis w asemblerze będzie postaci:

```
        ; obliczenie warunku
        or al,al
        jz do_else
        ; instrukcje1
        jmp zakoncz
do_else:
        ; instrukcje 2;
zakoncz:
```

Zapis odpowiednika instrukcji for w asemblerze

Iteracje w asemblerze wygodnie jest realizować przy użyciu instrukcji loop. Instrukcje języka C o postaci:

```
for( i= 0; i<10: i++)
{
        instrukcje;
}
```

można zrealizować stosując następujące instrukcje asemblera:

```
        mov cx,10
znowu:
        ; instrukcje
        loop znowu
```

Instrukcja loop powoduje zmniejszenie wartości liczby w rejestrze CX o jeden. Jeżeli wynik dekrementacji nie będzie równy zero, to nastąpi skok do etykiety.

Program z przykładu 1.1. można zmienić tak, aby również instrukcje sterujące pętlą napisane były w asemblerze.

```
#pragma inline
void main(void)
{
        int n= 'z'-'a' + 1, litera= 'a';
        asm {
                mov CX, n
                mov DX, litera
 petla:
                mov AH, 2
                int 21H
                inc DX
                loop petla

                mov DX, 13
                mov AH, 2
                int 21H
        }
        return;
}
```

W podanych przykładach wykorzystywano przerwanie numer 21H. Przerwanie to powoduje wywołanie funkcji MS-DOS (DOS Service Interrupt). Typ wywołanej funkcji zależy od numeru funkcji podanego w rejestrze AH. Dla przykładu zawartość rejestru

AH równe 2 - powoduje wyprowadzenie znaku zawartego w rejestrze DX na ekran,
AH równe 9 - powoduje wyprowadzenie łańcucha znaków zakończonego znakiem $. Łańcuch znaków zapisany jest pod adresem podanym w rejestrze DX.

Przykład 1.2.

Napisać wstawkę asemblerową, której zadaniem jest dodanie dwóch liczb całkowitych n i m oraz wypisanie wyniku.

```
#pragma inline
/* Dodaj dwie liczby n i m, wynik wyprowadź n na ekran */

void main(void)
{
        int     n= 12321, m=777;
        char    *info= "Wynik: n+m= $";
        asm {
                mov AX, n       /* Załaduj wartość n do rejestru AX          */
                add AX, m       /* Dodaj wartość m do rejestru AX            */
                mov DI, 10      /* Załaduj do rejestru DI liczbę 10          */
                sub CX, CX      /* Wyzeruj rejestr CX                        */
        };
 petla:
        asm  {
                /* zapisanie cyfr wyniku na stosie                           */
                sub  DX, DX     /* Wyzeruj rejestr DX                        */
                div  DI         /* Podziel liczbę z rejestru AX przez 10
        */
                push DX         /* Połóż resztę z dzielenia na stosie         */
                inc  CX         /* Zliczanie liczby bajtów w rejestrze CX
        */
                or AX, AX       /* Określ wartość znacznika zera             */
                jnz petla       /* Skok jeżeli znacznik zera ustawiony
        */

                /* wyprowadzenie nagłówka obliczeń                           */
                mov  DX, 0AH  /* Wprowadź znak nowej linii LF do DX
        */
                mov AH, 2       /*  Wprowadź kod funkcji systemu DOS do AH
        */
                int 21H         /* Wykonaj przerwanie 21H systemu DOS        */
                mov DX, 0DH   /* Wprowadź znak CR powrotu do pocz. linii */
                mov AH, 2       /*  Wprowadź kod funkcji systemu DOS do AH
        */
                int 21H         /* Wykonaj przerwanie 21H systemu DOS        */
                mov DX, info    /* Wprowadź adres łańcucha znaków do DX      */
                mov AH, 9       /*  Wprowadź kod funkcji systemu DOS do AH
        */
                int 21H         /* Wykonaj przerwanie 21H systemu DOS        */
        };
wyprowadz:
        asm  {
                /* wyprowadzenie znaków cyfr odłożonych na stosie
        */
                pop AX          /* Pobierz słowo ze stosu i załaduj do rej. AX*/
                add AL, '0'     /* Dodaj do zawartości AX wartość znaku '0' */
                mov DX, AX      /* Załaduj zawartość z rejestru AX do DX     */
                mov AH, 2       /*  Wprowadź kod funkcji systemu DOS do AH
        */
                int 21H         /* Wykonaj przerwanie 21H systemu DOS        */
                loop wyprowadz        /* Dekrementacja  CX, skok gdy CX różny od
0       */
        }
        return;
}
```

Przykład 1.3.
Napisać wstawkę asemblerową, której zadaniem jest wyprowadzenie liter od a do z na stos, a następnie wypisanie ich na ekranie w odwrotnej kolejności (poprzez pobieranie ich z stosu).

```
#pragma inline
void main(void)
{
        int i;
        asm sub CX,CX
        for(i='a';i<='z';i++)
        {
                asm mov DX,i
                asm push DX          /* Zapisanie na stos a, b, ... z          */
                asm mov AH,2
                asm int 21H
        }
        asm mov DX,0AH               /* Przejście na początek następnej linii
        */
        asm mov AH,2
        asm int 21H
        asm mov DX,0DH
        asm mov AH,2
        asm int 21H
        for(i=1;i<=26;i++)
        {
                asm pop DX           /* Pobranie znaku ze stosu                */
                asm mov AH,2         /* Wyprowadzenie pobranego znaku  */
        asm int 21H
        }
        asm mov DX,0AH               /* Przejście na początek następnej linii
        */
        asm mov AH,2
        asm int 21H
        asm mov DX,0DH
        asm mov AH,2
        asm int 21H

        return;
}
```

1.5. Możliwość programowania w języku C na poziomie asemblera

Zadeklarowane w module języka C zmienne i funkcje klasy extern mogą stać się dostępne przez ich identyfikatory w module asemblerowym. Warunkiem dostępności jest posłużenie się w module asemblerowym dyrektywą EXTERN, w której identyfikator zmiennej albo funkcji w języku C jest poprzedzony znakiem "_".
W języku C możliwe jest bezpośrednie odwołanie się do rejestrów procesora, można się tu posłużyć nazwami pseudozmiennych np.: _AX , _AL , _BX itp..
Pewne funkcje C wykorzystują struktury rejestrów, podane w plikach nagłówkowych bios.h i dos.h. Struktury te są postaci:

```
struct WORDREGS {
   unsigned int    ax, bx, cx, dx, si, di, cflag, flags;
};
struct BYTEREGS {
   unsigned char   al, ah, bl, bh, cl, ch, dl, dh;
};
union   REGS    {
```

```
   struct  WORDREGS x;
   struct  BYTEREGS h;
};
struct  SREGS  {
   unsigned int   es;
   unsigned int   cs;
   unsigned int   ss;
   unsigned int   ds;
};
struct  REGPACK {
   unsigned   r_ax, r_bx, r_cx, r_dx;
   unsigned   r_bp, r_si, r_di, r_ds, r_es, r_flags;
};
```

W języku C możliwe jest korzystanie z rejestrów oraz obsługa przerwań systemów DOS i BIOS, co ilustruje przykład 1.4.

Przykład 1.4.
Poniżej podano część programu obsługi myszy w języku C i języku asemblera:

```
union  REGS  rej, out ;
struct  SREGS  seg;
```

```
inicjalizuj_m()
{
        rej.h.ah = 0x0;
        rej.x.ax = 0;
        int86x ( 0x33,&rej,&out,&seg ) ;
        rej.h.ah = 0x1;
        rej.x.ax =1 ;
        int86x( 0x33,&rej,&out,&seg ) ;
}
```

```
inicjalizuj_m()
{
        asm mov ah, 0
        asm mov ax, 0
        asm int  33h   przerwanie nr 33 - obsługa myszki
        asm mov  ah, 1
        asm mov  ax, 1
        asm int  33h
}
```

1.6. Kompilacja programów z wstawkami asemblera

Jeżeli w programie napisanym w języku C wystąpią wstawki napisane w języku asembler, to należy o tym powiadomić kompilator języka C. Dokonywane jest to na dwa sposoby:

1) W przypadku, gdy program jest kompilowany poza systemem zintegrowanym języka C należy posłużyć się kompilatorem języka C wywoływanym za pomocą dyrektywy :

```
bcc   opcje   zbiory
```

Jeśli tłumaczony program zawiera wstawki asemblerowe, to kompilator powinien zostać uruchomiony z opcją -B, a w katalogu bieżącym powinien się znajdować makroasembler TASM i konsolidator TLINK.

2) Jeżeli zostanie w programie użyty wiersz

```
# pragma inline
```

to, gdy kompilator napotka ten wiersz, zostanie utworzony plik z instrukcjami asemblera, a następnie plik zostanie skompilowany przy użyciu asemblera TASM i dołączony jako plik o rozszerzeniu obj do programu.

Przykład :

```
bcc  -B  -I include  -L lib  source.c
```

- skompilowanie i skonsolidowanie z bibliotekami pliku SOURCE.C
- pliki nagłówkowe będą poszukiwane w podkatalogu INCLUDE, a biblioteki w podkatalogu LIB.

PROGRAM wykonywalny nosić będzie nazwę SOURCE.EXE.

Literatura :

[1] Bielecki J.: Turbo Assembler Wersja 2.0. Warszawa, Wydawnictwo PLJ 1991
[2] Goczyński R., Tuszyński M.: Mikroprocesory 80286, 80386 i i486. Warszawa, Komputerowa Oficyna Wydawnicza „Help" 1991
[3] Holzner S.: Programowanie w Borland C++, Warszawa, Intersoftland 1992
[4] Kruk S.: Turbo Assembler Wprowadzenie. Warszawa, Wydawnictwo PLJ 1992
[5] Mikroprocesory Firmy Intel. Praca zbiorowa pod redakcją Cezarego Stępnia. Warszawa, Wydawnictwo Naukowe PWN 1992
[6] Scragg G. W.: Computer Organization, A Top-Down approach. McGraw-Hill, Inc. 1992
[7] Skorupski A.: Podstawy budowy i działania komputerów. Warszawa, WKŁ 1996
[8] Syck G.: Turbo Assembler - Biblia użytkownika. Warszawa, LT&P1994
[9] Wróbel E.: Asembler 8086/88. Warszawa, WNT 1992

Ćwiczenia:

1. Napisz program w asemblerze wyprowadzający zadany ciąg liczb w odwrotnej kolejności.
2. Napisz program w asemblerze wyprowadzający tekst "Automatyka i Robotyka" w podanej i odwrotnej kolejności, stosując operacje na stosie.