KATEDRA INFORMATYKI,
TECHNICZNEJ
POLITECHNIKI WROCŁAWSKIEJ

Laboratorium architektury komputerów - materiały

Słowa kluczowe:
- Architektura IA-32
- Programowanie assembler

Wrocław 2020

1. Wstęp

Niniejszy skrypt ma stanowić pomoc dla studentów odbywających zajęcia w laboratorium z przedmiotu Architektura Komputerów 2. Zawiera on informacje zebrane z wielu źródeł i nie jest publikacją w sensie obowiązującego prawa. Jego celem jest dostarczenie studentom materiałów w celu ułatwienia osiągnięcia następujących celów dydaktycznych:

1. Zapoznanie się z podstawami uruchamiania programów w systemie Linux.
2. Zapoznanie się z podstawami architektury IA-32 (32 bitowa architektura procesorów Intel)
3. Podstawowe instrukcje maszynowe procesora architektury IA-32
4. Podstawy pisania prostych programów w języku assemblera Gnu (konwencja AT&T)
5. Uruchamianie programów za pomocą programu uruchomieniowego gdb (GNU debugger)
6. Konstrukcja makr i funkcji
7. Wykorzystanie biblioteki standardowej C w programach napisanych w assemblerze
8. Tworzenie programów w języku C i wykorzystanie w nich funkcji napisanych w assemblerze.
9. Zapoznanie się z zagadnieniami reprezentacji liczb zmiennoprzecinkowych w formacie IEEE 754
10. Zapoznanie się z konstrukcją i zasadami programowania jednostki zmiennoprzecinkowej x87 FPU

W zakresie architektury procesorów IA-32 podstawowym podręcznikiem jest pozycja [10] IA-32 Intel® Architecture Software Developers Manual Volume 1: Basic Architecture a w szczególności:

Rozdział	Tytuł	Zawartość
3	Basic Execution Enviroment	Opis trybów pracy, rejestrów, trybów adresowania, organizacji pamięci
4	Data Types	Opis typów danych
5.1	Instruction set summary General Purpose Instructions	Instrukcje ogólnego przeznaczenia
5.2	Instruction set summary FPU 87 Instructions	Instrukcje procesora zmiennoprzecinkowego X87
6	Procedure Calls, Interrupts, Exceptions	Stos, wywołania funkcji, przewania, wyjatki
7	Programming enviroment for the General Purpose Instructions	Podstawy programowania z użyciem instrukcji ogólnego przeznaczenia
8	Programming with the x87 FPU	Podstawy programowania z użyciem instrukcji procesora zmiennoprzecinkowego x87 FPU

Tab. 1-1 Obowiązujące rozdziały podręcznika IA-32 Intel® Architecture Software Developers Manual Volume 1

Przystępne wprowadzenie do programowania w GNU assemblerze dla architektury IA-32 zawarte jest w ksiażce [3] Jonathan Bartlet, Programming from Ground Up. Książka ta (z dodatkami) liczy tylko 230 stron. Doskonałym wprowadzeniem do architektury IA-32 i programowania w assemblerze jest książka [4] R. E. Bryant, D. R. O`Hallaron, Computer System, a Programmers Perspective Szczególnie ważne są rozdziały 2 i 3. Niezbędne szczegóły dotyczące języka GNU assemblera zawarte są w opracowaniu Dean Elsner, Jay Fenlason & friends ,Using As dostępnym pod adresem https://sourceware.org/binutils/docs/as/. Szczególnie ważne są rozdziały 3,4,5.

Autor bardzo dziękuje prof dr hab. Januszowi Biernatowi za przeczytanie pracy i wskazanie błędów co umożliwiło ich usunięcie.

2. Podstawy posługiwania się systemem Linux

2.1 Wstęp

Poniżej podane zostały podstawowe informacje umożliwiające posługiwanie się systemem w zakresie uruchamiana prostych programów napisanych w języku assemblera GAS i C.

2.2 Uzyskiwanie pomocy

Polecenie	Opis
`man polecenie/funkcja`	Uzyskanie informacji o poleceniu / funkcji – narzędzie `man`
`info polecenie/funkcja`	Uzyskanie informacji o poleceniu / funkcji – narzędzie `info`
`whatis słowo_kluczowe`	Uzyskanie krótkiej informacji o temacie danym w postaci słowa kluczowego
`apropos słowo_kluczowe`	Przeszukanie dokumentacji w poszukiwaniu słowa kluczowego
`file nazwa_pliku`	Uzyskanie informacji o typie podanego pliku
Katalog `/usr/share/doc`	W katalogu tym zawarta jest dokumentacja dla różnych programów, pakietów i modułów
Internet	Witryna http://kernel.org/doc/manpages Witryny dystrybucji Linuxa Debiana: http://www.debian.org/ Ubuntu : http://ubuntu.pl/

2.2.1 Narzędzie man

Standardowym systemem przeglądania dokumentacji jest narzędzie `man`. Uruchamiamy je wpisując w terminalu polecenie:

```
$man temat
```

gdzie `temat` jest tekstem określającym na temat który chcemy uzyskać informację. Przykładowo gdy chcemy uzyskać informację na temat funkcji `fork` piszemy:

```
$man fork
```

Dokumentacja pogrupowana jest tradycyjnie w działach które podane są w poniższym zestawieniu:

Dział	Zawartość
1	Polecenia
2	Wywołania systemowe
3	Funkcje biblioteczne
4	Pliki specjalne – katalog /dev
5	Formaty plików
6	Gry
7	Definicje, informacje różne
8	Administrowanie systemem
9	Wywołania jądra

Wiedza o działach bywa przydatna gdyż nieraz jedna nazwa występuje w kilku działach. Wtedy man wywołujemy podając jako drugi parametr numer sekcji.

```
$man numer_sekcji temat
```

Na przykład:

```
$man  3 open
```

Przydatnym poleceniem jest opcja –k

```
$man  -k słowo_kluczowe
```

```
Pozwala przeszukać manual i znaleźć tematy w których występuje dane słowo kluczowe.Np.:
```

```
$man  -k open
```

Do poruszania się w manualu stosujemy klawisze funkcyjne:

⇑	Linia do góry
⇓	Linia w dół
PgUp	Strona do góry
PgDn	Strona w dół
/ temat	Przeszukiwanie do przodu
? temat	Przeszukiwanie do tyłu

Strona podręcznika składa się z kilku sekcji: nazwa (NAME) , składnia (SYNOPSIS) , konfiguracja (CONFIGURATION), opis (DESCRIPTION), opcje (OPTIONS), kod zakończenia (EXIT STATUS), wartość zwracana (RETURN VALUE), błędy (ERRORS), środowisko (ENVIROMENT), pliki (FILES), wersje (VERSIONS), zgodne z (CONFORMING TO), uwagi, (NOTES), błędy (BUGS), przykład (EXAMPLE), autorzy (AUTHORS), zobacz także (SEE ALSO).
Dokumentacja man dostępna jest w postaci HTML pod adresem : http://www.kernel.org/doc/man-pages

Narzędzia do przeglądania man'a:

- `tkman` – przeglądanie w narzędziu Tkl
- `hman` – przegladanie w trybie HTML

2.2.2 Narzędzie apropos

Narzędzie apropos wykonuje przeszukanie stron podręcznika man w poszukiwaniu podanego jako parametr słowa kluczowego.

```
$apropos słowo_kluczowe
```

2.2.3 Narzędzie whatis

Narzędzie `whatis` wykonuje przeszukanie stron podręcznika man w poszukiwaniu podanego jako parametr słowa kluczowego. Następnie wyświetlana jest krótka informacja o danym poleceniu / funkcji.

```
$whatis słowo_kluczowe
```

Przykład:

```
$whatis open
open (1)- start a program on a new virtual terminal
open (2)- open and possibly create a file or device
open (3posix)- open a file
```

Uzyskane strony podręcznika można następnie wyświetlić za pomocą narzędzia `man`.

2.2.4 Narzędzie info

Dodatkowym systemem przeglądania dokumentacji jest narzędzie `info`. Uruchamiamy je wpisując w terminalu polecenie:

```
$info temat
```

2.2.5 Klucz --help

Większość poleceń GNU może być uruchomiona z opcją - - help. Użycie tej opcji pozwala na wyświetlenie informacji o danym poleceniu.

Dokumentacja systemu Linux dostępna jest w Internecie. Można ją oglądać za pomocą wchodzącej w skład systemu przeglądarki Firefox.

Ważniejsze źródła podane są poniżej:
- Dokumentacja man w postaci HTML: http://www.kernel.org/doc/man-pages
- Materiały Linux Documentation Project: http://tldp.org
- Machtelt Garrels, Introduction to Linux - http://tldp.org/LDP/intro-linux/intro-linux.pdf
- Dokumentacja na temat dystrybucji UBUNTU: - http://help.ubuntu.com
- Brian Ward, Jak działa Linux, Podręcznik administratora, Helion,

2.3 Operowanie plikami i katalogami

2.3.1 Pliki i katalogi

W systemie Linux prawie wszystkie zasoby są plikami. Dane i urządzenia są reprezentowane przez abstrakcję plików. Mechanizm plików pozwala na jednolity dostęp do zasobów tak lokalnych jak i zdalnych za pomocą poleceń i programów usługowych wydawanych z okienka terminala. Plik jest obiektem abstrakcyjnym z którego można czytać i do którego można pisać. Oprócz zwykłych plików i katalogów w systemie plików widoczne są pliki specjalne. Zaliczamy do nich łącza symboliczne, kolejki FIFO, bloki pamięci, urządzenia blokowe i znakowe.
System umożliwia dostęp do plików w trybie odczytu, zapisu lub wykonania. Symboliczne oznaczenia praw dostępu do pliku dane są poniżej:

`r` - Prawo odczytu (*ang. read*)
`w` - Prawo zapisu (*ang. write*)
`x` - Prawo wykonania (*ang. execute*)

Prawa te mogą być zdefiniowane dla właściciela pliku, grupy do której on należy i wszystkich innych użytkowników.

`u` - Właściciela pliku (*ang. user*)
`g` - Grupy (*ang. group*)
`o` - Innych użytkowników (*ang. other*)

2.3.2 Polecenia dotyczące katalogów

Pliki zorganizowane są w katalogi. Katalog ma postać drzewa z wierzchołkiem oznaczonym znakiem /. Położenie określonego pliku w drzewie katalogów określa się za pomocą ścieżki. Rozróżnia się ścieżki absolutne i relatywne. Ścieżka absolutna podaje drogę jaką trzeba przejść od wierzchołka drzewa do danego pliku. Przykład ścieżki absolutnej to `/home/juka/prog/hello.c`. Ścieżka absolutna zaczyna się od znaku /. Ścieżka relatywna zaczyna się od innego znaku niż /. Określa ona położenie pliku względem katalogu bieżącego. Po zarejestrowaniu się użytkownika w systemie katalogiem bieżącym jest jego katalog domowy. Może on być zmieniony na inny za pomocą polecenia `cwd`.

2.3.2.1 Uzyskiwanie nazwy katalogu bieżącego

Nazwę katalogu bieżącego uzyskuje się pisząc polecenie `pwd`. Na przykład:

```
$pwd
/home/juka
```

2.3.2.2 Listowanie zawartości katalogu

Zawartość katalogu uzyskuje się wydając polecenie `ls`. Składnia polecenia jest następująca:

```
ls [-l] [nazwa]
```

Gdzie:
`l` - Listowanie w „długim” formacie, wyświetlane są atrybuty pliku
`nazwa` - Nazwa katalogu lub pliku

Gdy nazwa określa pewien katalog to wyświetlona będzie jego zawartość. Gdy nazwa katalogu zostanie pominięta wyświetlana jest zawartość katalogu bieżącego. Listowane są prawa dostępu, liczba dowiązań,

właściciel pliku, grupa, wielkość, data utworzenia oraz nazwa. Wyświetlanie katalogu bieżącego ilustruje Przykład 2-1.

```
$ls -l
-rwxrwxr-x 1 root root  7322 Nov 14  2003 fork3
-rw-rw-rw- 1 root root   886 Mar 18  1994 fork3.c
```

typ, właściciel, grupa, inni, liczba dowiązań, właściciel, grupa, wielkość, data utworzenia, nazwa

Przykład 2-1 Listowanie zawartości katalogu bieżącego.

Typy plików:

oznaczenie	Opis
-	Regularny
d	Katalog (directory)
b	Plik specjalny – urządzenie blokowe
c	Plik specjalny – urządzenie znakowe
p	Plik specjalny – łącze lub plik FIFO
l	Plik specjalny – link symboliczny
s	Plik specjalny - gniazdko

Tabela 2-1 Typy plików w systemie Linux

2.3.2.3 Listowanie drzewa katalogów

```
tree –L poziom katalog
```

Przykład

```
$tree –L 2  /etc
```

2.3.2.4 Zmiana katalogu bieżącego

Katalog bieżący zmienia się na inny za pomocą polecenia `cd`. Składnia polecenia jest następująca: `cd nowy_katalog` .Gdy jako parametr podamy dwie kropki `..` to przejdziemy do katalogu położonego o jeden poziom wyżej. Zmianę katalogu bieżącego ilustruje Przykład 2-2.

```
$pwd
/home/juka
$cd   prog
$pwd /home/juka/prog
```

Przykład 2-2 Zmiana katalogu bieżącego

2.3.2.5 Tworzenie nowego katalogu

Nowy katalog tworzy się poleceniem `mkdir`. Polecenie to ma postać: `mkdir  nazwa_katalogu`. Tworzenie nowego katalogu ilustruje Przykład 2-3.

```
$ls
prog
$mkdir src
$ls
prog src
```

Przykład 2-3 Tworzenie nowego katalogu

2.3.2.6 Kasowanie katalogu

Katalog kasuje się poleceniem `rmdir`. Składnia polecenia `rmdir` jest następująca: `rmdir nazwa_katalogu`. Aby możliwe było usuniecie katalogu musi on być pusty. Kasowanie katalogu ilustruje Przykład 2-4.

```
$ls
prog src
$rmdir src
$ls
prog
```

Przykład 2-4 Kasowanie katalogu

2.3.3 Polecenia dotyczące plików

Kopiowanie pliku
Pliki kopiuje się za pomocą polecenia `cp`. Składnia polecenia `cp` jest następująca:

```
cp [-ifR] plik_źródłowy plik_docelowy
cp [-ifR] plik_źródłowy katalog_docelowy
```

Gdzie:
`i` - Żądanie potwierdzenia gdy plik docelowy może być nadpisany.
`f` - Bezwarunkowe skopiowanie pliku.
`R` - Gdy plik źródłowy jest katalogiem to będzie skopiowany z podkatalogami.

Kopiowanie plików ilustruje Przykład 2-5.

```
$ls
nowy.txt prog
$ls prog
$
$cp nowy.txt prog
$ls prog
nowy.txt
```

Przykład 2-5 Kopiowanie pliku `nowy.txt` z katalogu bieżącego do katalogu `prog`

Zmiana nazwy pliku
Nazwę pliku zmienia się za pomocą polecenia `mv`. Składnia polecenia `mv` dana jest poniżej:

```
mv [-if] stara_nazwa nowa_nazwa
mv [-if] nazwa_pliku katalog_docelowy
```

Gdzie:
`i` - Żądanie potwierdzenia gdy plik docelowy może być nadpisany.
`f` - Bezwarunkowe skopiowanie pliku.

Zmianę nazwy plików ilustruje Przykład 2-6.

```
$ls
stary.txt
$mv stary.txt nowy.txt
$ls
nowy.txt
```

Przykład 2-6 Zmiana nazwy pliku `stary.txt` na `nowy.txt`

2.3.3.1 Kasowanie pliku

Pliki kasuje się za pomocą polecenia `rm`. Składnia polecenia rm jest następująca:
`rm [-Rfi] nazwa`

Gdzie:
`i` - Żądanie potwierdzenia przed usunięciem pliku.
`f` - Bezwarunkowe kasowanie pliku.
`R` - Gdy nazwa jest katalogiem to kasowanie zawartości wraz z podkatalogami.

Kasowanie nazwy pliku ilustruje Przykład 2-7.

```
$ls
prog nowy.txt
$rm nowy.txt
$ls
prog
```

Przykład 2-7 Kasowanie pliku `nowy.txt`

2.3.3.2 Listowanie zawartości pliku

Zawartość pliku tekstowego listuje się za pomocą poleceń:

- `more nazwa_pliku`,
- `less nazwa_pliku`, `cat nazwa_pliku`.
- `cat nazwa_pliku`

Można do tego celu użyć też innych narzędzi jak edytor `vi`, edytor `gedit` lub wbudowany edytor programu `Midnight Commander`.

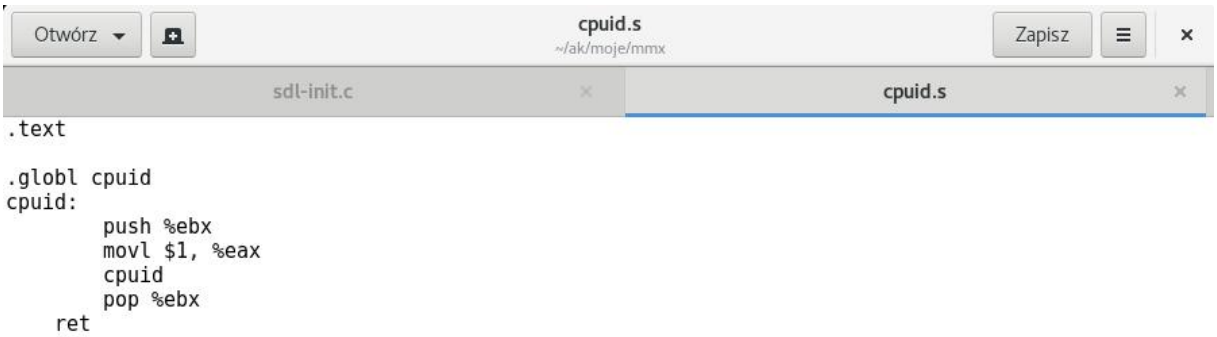
2.3.3.3 Szukanie pliku

- `locate wzorzec`
- `find ścieżka wzorzec`

Przykład
`$find /etc –name passwd`

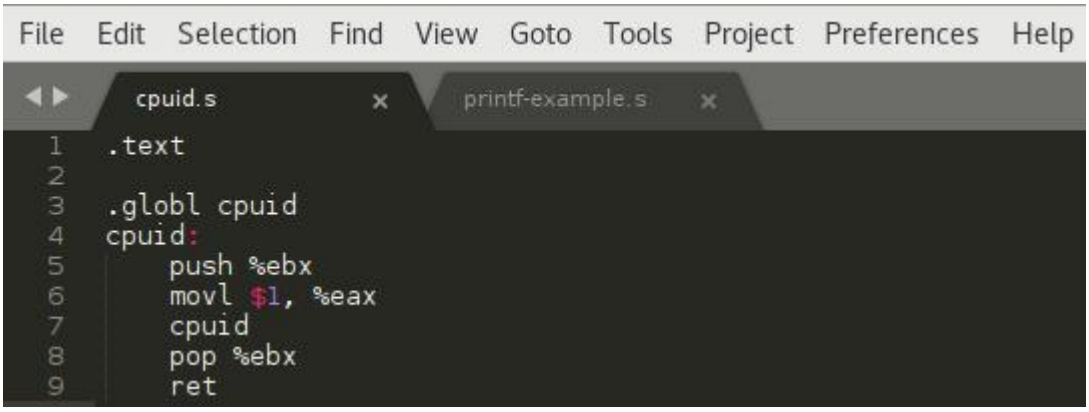
2.4 Edycja plików

Przy pisaniu programów niezbędna jest edycja plików. W systemie linux dostępnych jest wiele edytorów plików tekstowych. Wymienić można takie edytory jak `nano, vim, gedit, sublime_text 3`. Standardowym edytorem jest gedit który można uruchomić pisząc w konsoli polecenie: `gedit &`.



Ekran 2-1 Okno edytora gedit.

Innym godnym polecenia edytorem jest SublimeText 3 napisany w Pythonie. Sposób instalacji edytora został podany w https://vitux.com/how-to-install-sublime-text-3-code-editor-on-debian-10/



Ekran 2-2 Okno edytora Sublime Text 3.

2.5 Program Midnight Commander do zarządzania komputerem

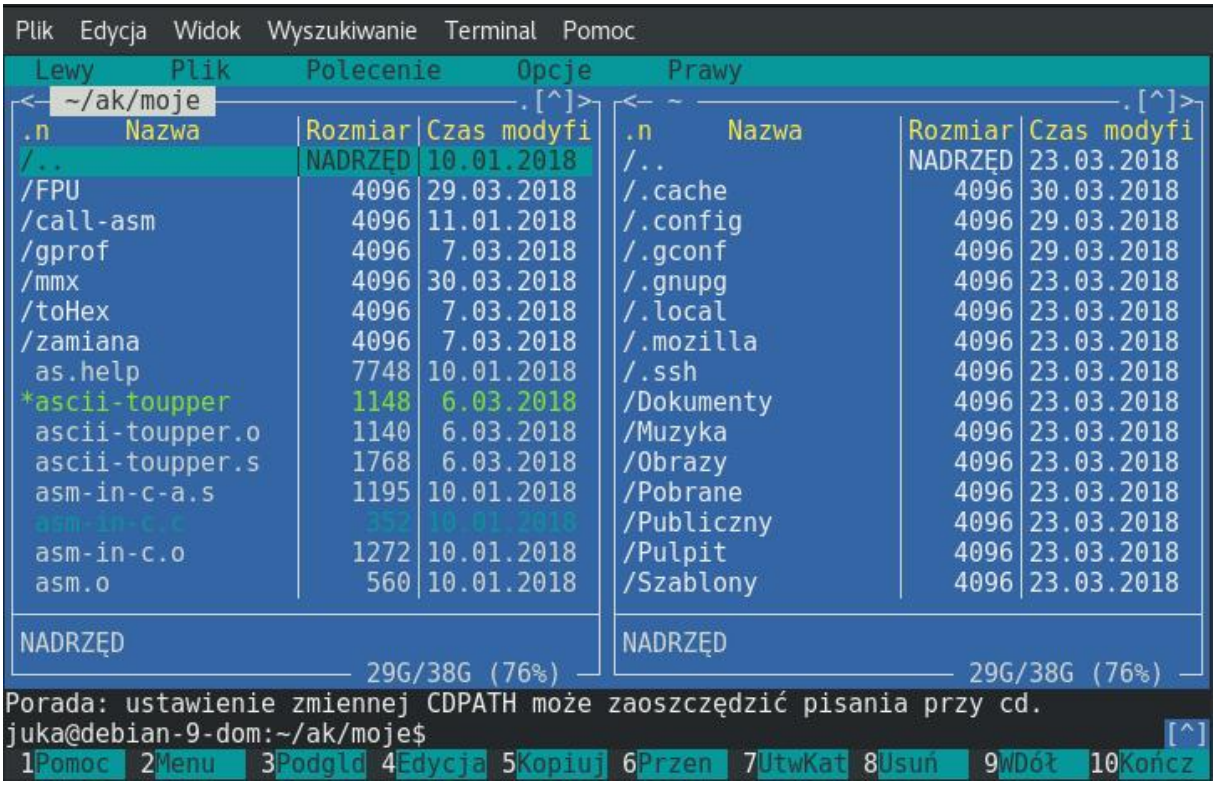
Większość czynności związanych z zarządzaniem komputerem wykonać można za pomocą programu Midnight Commander. Strona domowa projektu to https://midnight-commander.org/. Program instaluje się jak niżej.

```
$sudo apt-get install mc
```

Program uruchamia się pisząc:

```
$mc
```

Wygląd konsoli z uruchomionym programem mc pokazano poniżej



Ekran 2-3 Wygląd programu Midnight Commander

2.5.1 Konsola i standardowe wejście i wyjście

Podstawowym narzędziem komunikacji z systemem jest konsola. Umożliwia ona uruchamianie programów, wprowadzanie do nich danych wejściowych i obserwację wyników. Dla każdego wykonywanego programu system automatycznie otwiera trzy pliki:

- Standardowe wejście
- Standardowe wyjście
- Wyjście komunikatu o błędach

Opis	Wartość uchwytu	Przypisanie początkowe
Standardowe wejście	0	Klawiatura
Standardowe wyjście	1	Ekran
Wyjście komunikatów o błędach	2	Ekran

Tab. 2-1 Standardowe wejście / wyjście programu

Standardowe wejście wyjście posiada przyporządkowanie początkowe do urządzeń pokazane w Tab. 2-1. Może ono jednak zostać zmienione.

```
$ ./prog < plik_we
$ ./prog > plik_wy
$ ./prog > plik_wy > plik_err
$ ./prog < plik_we > plik_wy
$ ./prog1 | prog2
```

Przykład:

```
juka@debian-9-dom:~$ ls -l /etc | grep passwd
-rw-r--r--  1 root root     2102 mar 23  2018 passwd
-rw-------  1 root root     2102 mar 23  2018 passwd-
```

2.6 Operowanie procesami

2.6.1 Wyświetlanie uruchomionych procesów

2.6.1.1 Polecenie ps

Polecenie ps pozwala uzyskać informacje o uruchomionych procesach. Posiada ono wiele przełączników.

```
ps                      - wyświetlane są procesy o tym samym EUID co proces konsoli.
ps  - ef                - wyświetlanie wszystkich procesów w długim formacie.
ps  - ef | grep nazwa   - sprawdzanie czy wśród  procesów istnieje proces nazwa
```

2.6.1.2 Polecenie top

Pozwala uzyskać informacje o procesach sortując je według czasu zużycia procesora. Lista odświeżana jest c0 5 sekund. Poniżej podano przykład wywołania polecenia top.

```
$top
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1831 juka      20   0 83340  20m  16m S   37  0.5   1:07.64 gnome-system-
  951 root      20   0 76812  21m  10m S   10  0.5   0:41.70 Xorg
    1 root      20   0  2892 1684 1224 S    0  0.0   0:00.58 init
```

Przykład 2-8 Użycie polecenia top

Symbol	Opis
PID	Identyfikator procesu
USER	Nazwa efektywnego użytkownika
PR	Priorytet procesu
NI	Wartość parametru nice
VIRT	Całkowita wielkość pamięci wirtualnej użytej przez proces
RES	Wielkość pamięci rezydentnej (nie podlegającej wymianie) w kb
SHR	Wielkość obszaru pamięci dzielonej użytej przez proces
S	Stan procesu: R – running, D – uninteruptible running, S – sleeping, T – traced or stoped, Z - zombie
%CPU	Użycie czasu procesora w %
%MEM	Użycie pamięci fizycznej w %
TIME+	Skumulowany czas procesora zużyty od startu procesu
COMMAND	Nazwa procesu

Tab. 2-2 Znaczenie parametrów polecenia top

2.6.1.3 Polecenie pstree

Polecenie wyświetla drzewo procesów. Można obserwować zależność procesów typu macierzysty – potomny.

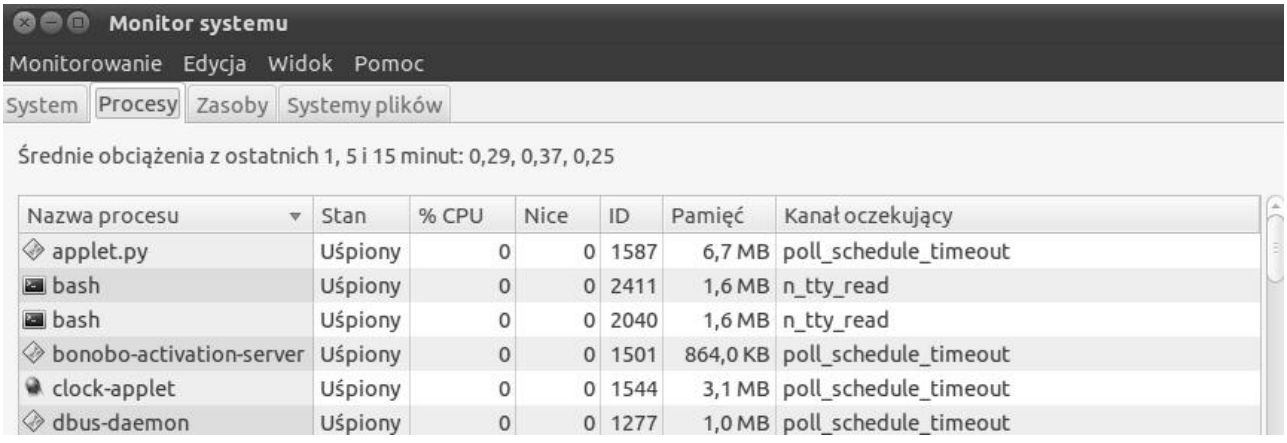
```
juka@debian-9-dom:~$ pstree
systemd─┬─ModemManager─┬─{gdbus}
        │              └─{gmain}
        ├─NetworkManager─┬─dhclient
        │                ├─{gdbus}
        │                └─{gmain}
        ├─accounts-daemon─┬─{gdbus}
        │                 └─{gmain}
```

Przykład 2-1 Wyświetlenie drzewa procesów (fragment)

2.6.1.4 Monitor systemu

W systemie Ubuntu dostępny jest monitor systemu który wyświetla informacje dotyczące aktywnych procesów. Uruchomienie następuje poprzez:

- Wybór opcji System / Administracja / Monitor systemu
- Wpisanie w terminalu polecenia: `gnome-system-monitor`



Przykład 2-9 Użycie polecenia monitora systemu

2.6.2 Kasowanie procesów

Procesy kasuje się poleceniem kill. Składnia polecenia jest nastepująca:

```
kill [-signal | -s signal] pid
```

Gdzie:
`signal` – numer lub nazwa sygnału
`pid` - pid procesu który należy skasować

Nazwa sygnału	Numer	Opis
SIGTERM	15	Zakończenie procesu w uporządkowany sposób
SIGINT	2	Przerwanie procesu, sygnał ten może być zignorowany
SIGKILL	9	Przerwanie procesu, sygnał ten nie może być zignorowany
SIGHUP	1	Używane w odniesieniu do demonów, powoduje powtórne wczytanie pliku konfiguracyjnego.

Tab. 2-3 Częściej używane sygnały

2.7 Zadania

2.7.1 Uzyskiwanie informacji o stanie systemu

Zobacz jakie procesy i wątki wykonywane są aktualnie w systemie.

2.7.2 Uzyskiwanie informacji o obciążeniu systemu

Używając polecenia `top` zbadaj który z procesów najbardziej obciąża procesor.

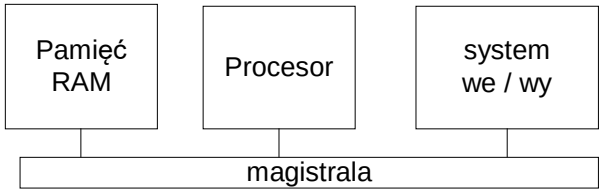
2.7.3 Archiwizacja i kopiowania plików

W systemie pomocy znajdź opis archiwizatora `tar`. Używając programu `tar` spakuj wszystkie pliki zawarte w katalogu bieżącym do pojedynczego archiwum o nazwie `programy.tar` (polecenie: `tar –cvf programy.tar *`). Następnie zamontuj dyskietkę typu MSDOS i skopiuj archiwum na dyskietkę. Dalej utwórz katalog `nowy` i skopiuj do niego archiwum z dyskietki i rozpakuj do postaci pojedynczych plików (polecenie: `tar –xvf programy.tar`).

3. Sprzęt w architekturze IA-32

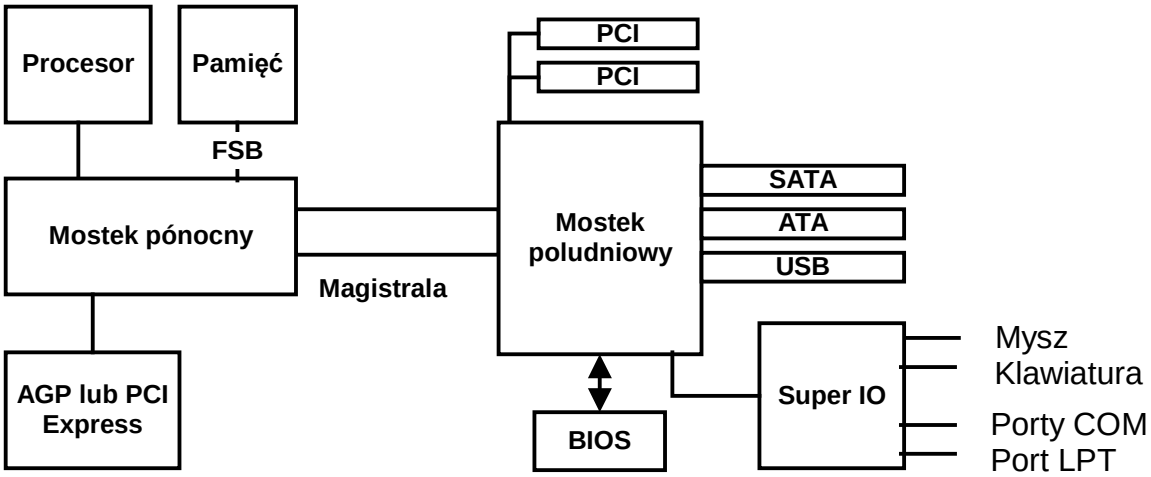
Aby program mógł się wykonywać i komunikować z otoczeniem niezbędny jest:

- pamięć
- procesor
- system wejścia wyjścia.



Rys. 3-1 Struktura systemu jednoprocesorowego

W rzeczywistości komputer jest bardziej skomplikowany. Uproszczony schemat płyty głównej komputera PC pokazuje poniższy rysunek.



Rys. 3-2 Architektura komputera PC

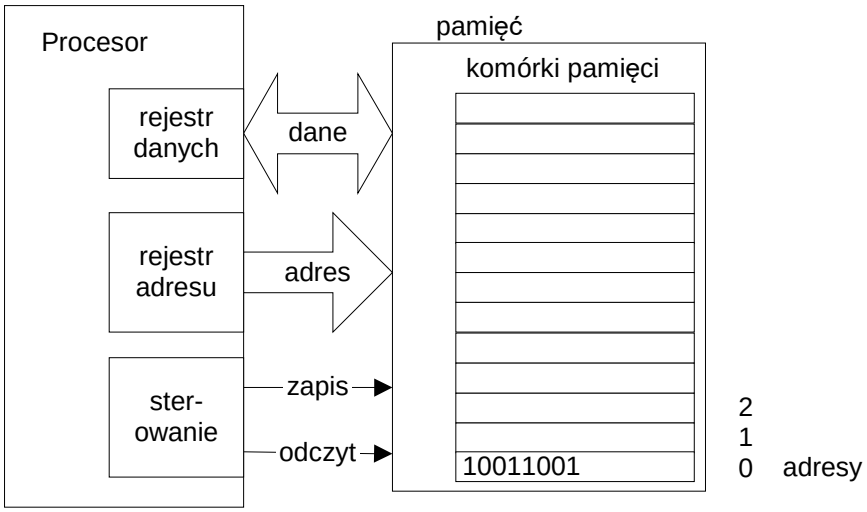
Mostek południowy jest układem sterującym magistralami:

- SATA – szeregowa magistrala do podłączania kontrolerów dyskowych, cdrom
- ATA – równoległa magistrala do podłączania kontrolerów dyskowych, cdrom
- USB - uniwersalna magistrala szeregowa do podłączania pamięci i innych urządzeń
- PCI - wewnetrzna magistrala do podłączania kontrolerów urzadzeń we/wy np. graficznych, pamięci itp.

Do magistral podłączone są różnorodne urządzenia zewnetrzne jak pamięci dyskowe, kontrolery graficzne, sieciowe itp.

3.1 Pamięć

Pamięć może być rozpatrywana jako liniowa tablica komórek (bajtów). Zawartość komórek to bajty, każda komórka posiada adres.



Rys. 3-3 Pamięć komputera

Pamięć może być postrzegana jako czarna skrzynka która realizuje operację odczytu i zapisu.

Odczyt:
Na wejście adresowe podaje się adres addr komórki która ma być odczytana i wystawia sygnał odczytu. Na linii danych pojawia się jej zawartość z = `M[addr].`

Zapis:
Na wejście adresowe podaje się adres `addr` komórki która ma być zapisana, na linii danych zawartość `z` i wystawia sygnał zapisu. Pod adresem `addr` zostaje zapisana zawartość `z, M[addr] = z.`
W procesorach IA-32 adresowanie pamięci jest 32 bitowe. Znaczy to że adres może posiadać najwyżej 4 bajty. W ten sposób można zaadresować 2^{32} = 4294967296 bajtów.

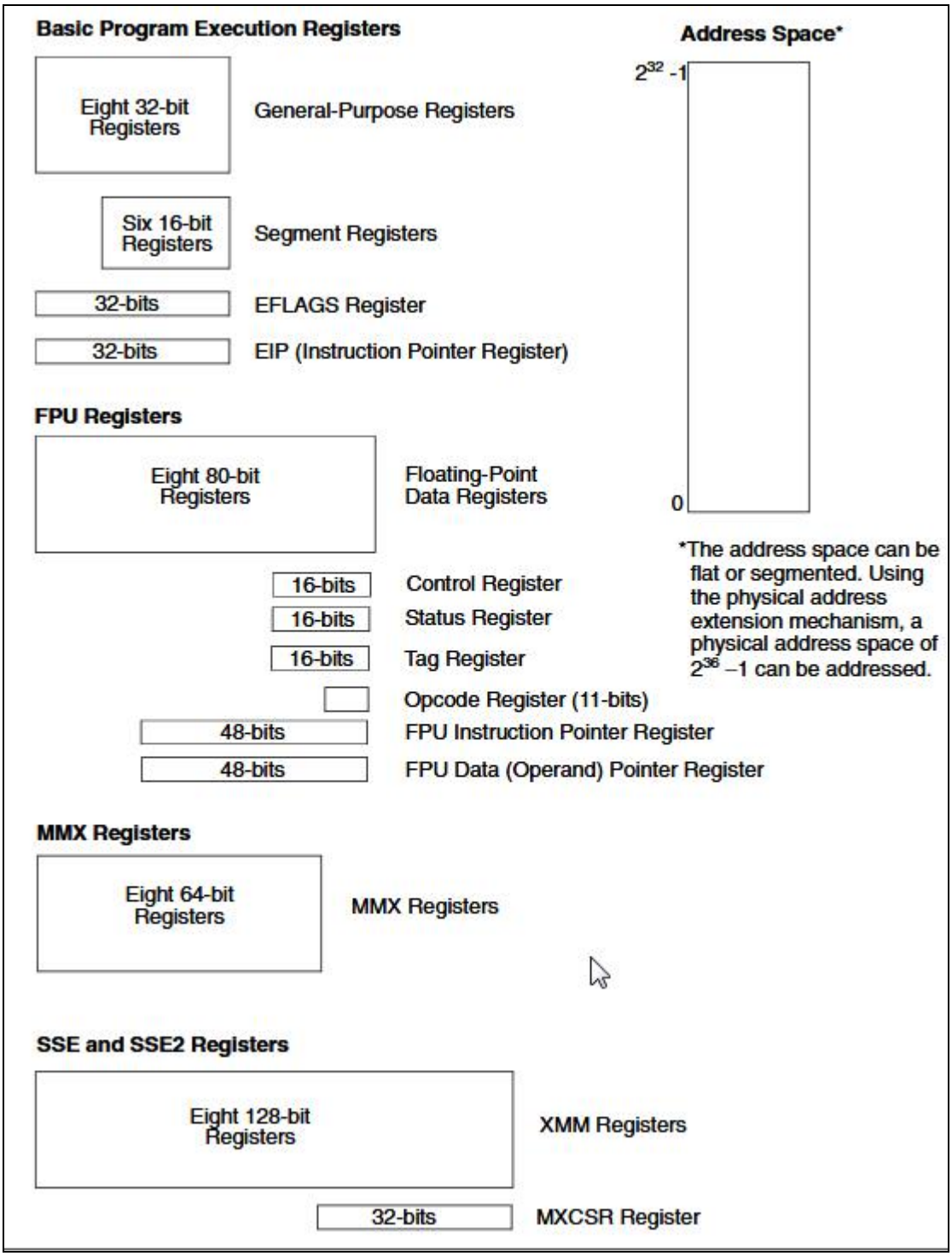
Powyższy opis nie specyfikuje ile bajtów jest przesyłane w jednej operacji odzczytu / zapisu. Zwykle może to być 1,2,4,8 bajtów.

3.2 Procesor

Procesor wykonuje instrukcje które zawarte są w pamięci. Procesor zawiera następujące elementy:

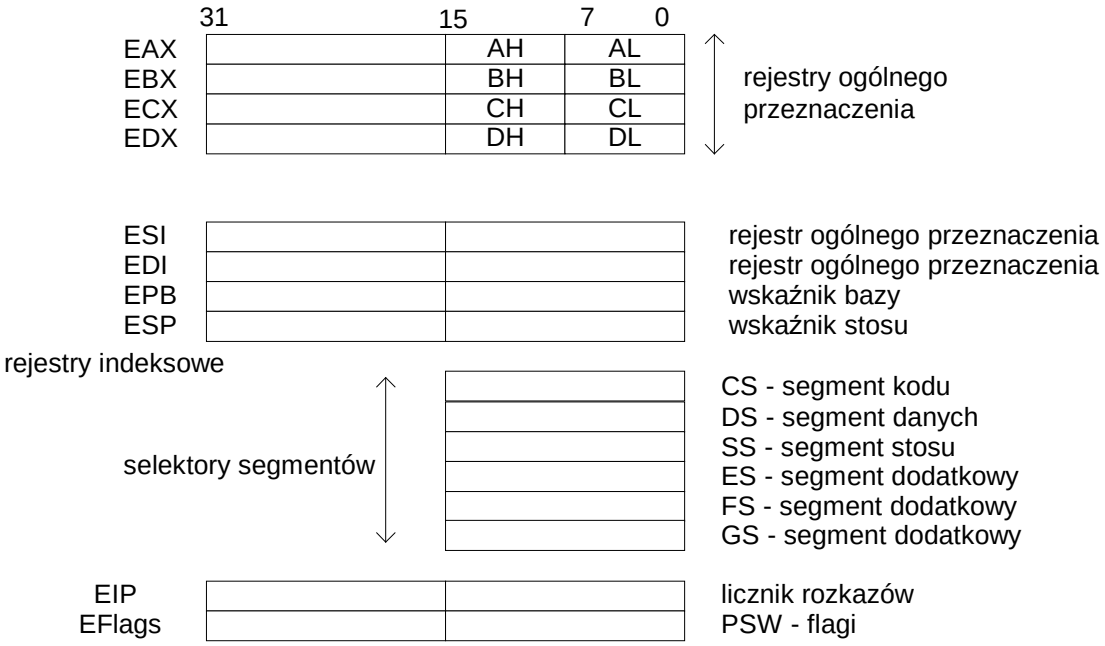
- Licznik rozkazów (instrukcji)
- Dekoder instrukcji
- Szynę danych
- Rejestry
- Jednostkę arytmetyczno logiczną

Rejestry procesora architektury IA-32 pokazuje poniższy rysunek pochodzący z [10].



Rys. 3-1 Podstawowe elementy środowiska IA-32 (według [10])

Rejestry ogólnego przeznaczenia mają swoje indywidualne nazwy co pokazuje poniższy rysunek.



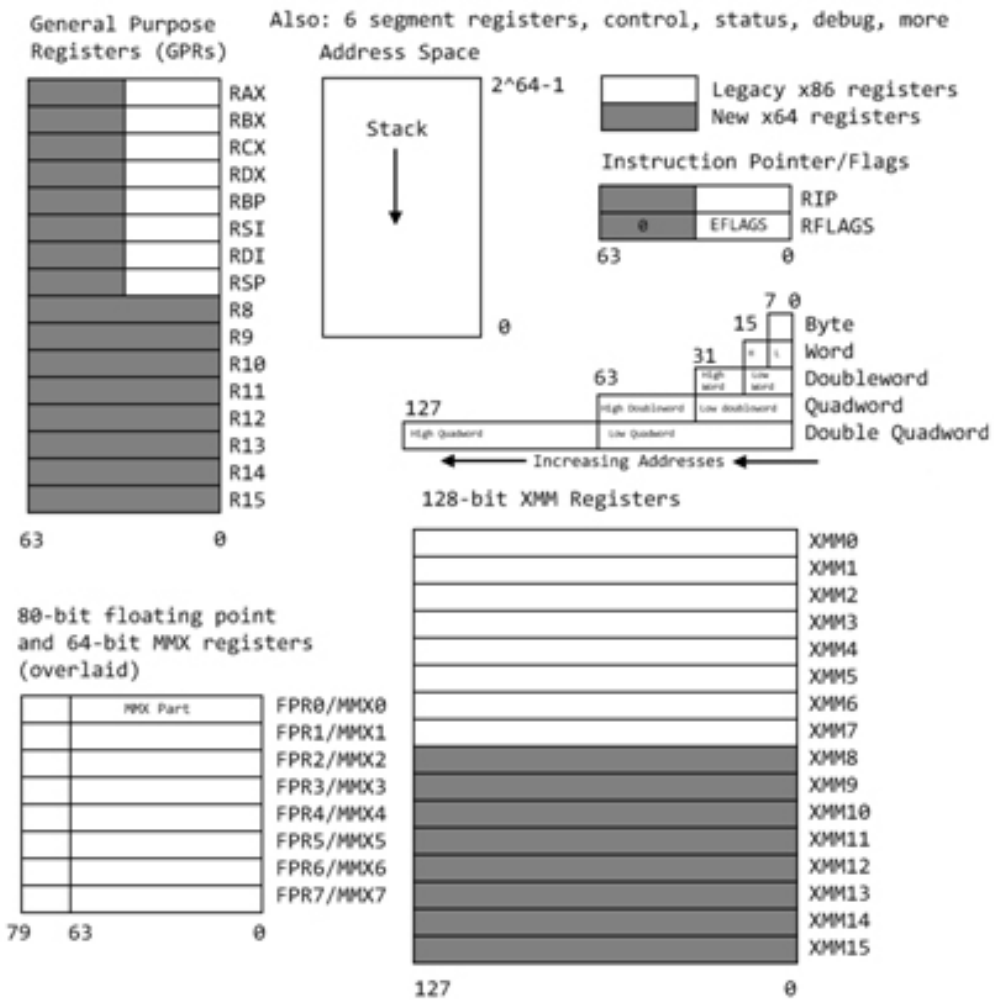
Rys. 3-4 Rejestry procesora w architekturze IA-32 dla trybu chronionego

Procesory rodziny IA-32 posiadają trzy tryby pracy:

- tryb rzeczywisty,
- tryb chroniony
- tryb zarządzania systemem.

Właściwym trybem normalnej pracy systemu jest tryb chroniony.

Wraz ze wzrostem wymagań na wielkość programów pamięć 4 GB jaka może być zaadresowana w procesorach 32 bitowych okazała się niewystarczająca. Aby zwiekszyć dostępną pamięć wprowadzono architekturę 64 bitową nazwaną X64. Rejestry procesora w tej architekturze pokazuje poniższy rysunek.



Rys. 3-5 Rejestry procesora w architekturze IA-64 dla trybu chronionego

Najważniejsze mechanizmy wspierające wieloprogramowość:

Wsparcie ochrony programów i systemu operacyjnego:

- segmentacja
- stronicowanie pamięci
- poziomy ochrony procesora.

Wsparcie przełączania procesów i współbieżnego systemu wejścia / wyjścia zapewniają:

- system przerwań i wyjątków
- autonomiczny system wejścia wyjścia

3.3 Zarządzanie pamięcią

W procesorze wykonuje się wiele procesów wśród których mogą być procesy nieprawidłowo napisane i złośliwe. Aby zapewnić bezpieczeństwo i efektywność w architekturze IA-32 stosowany jest złożony mechanizm zarządzania pamięcią. W jego skład wchodzą:

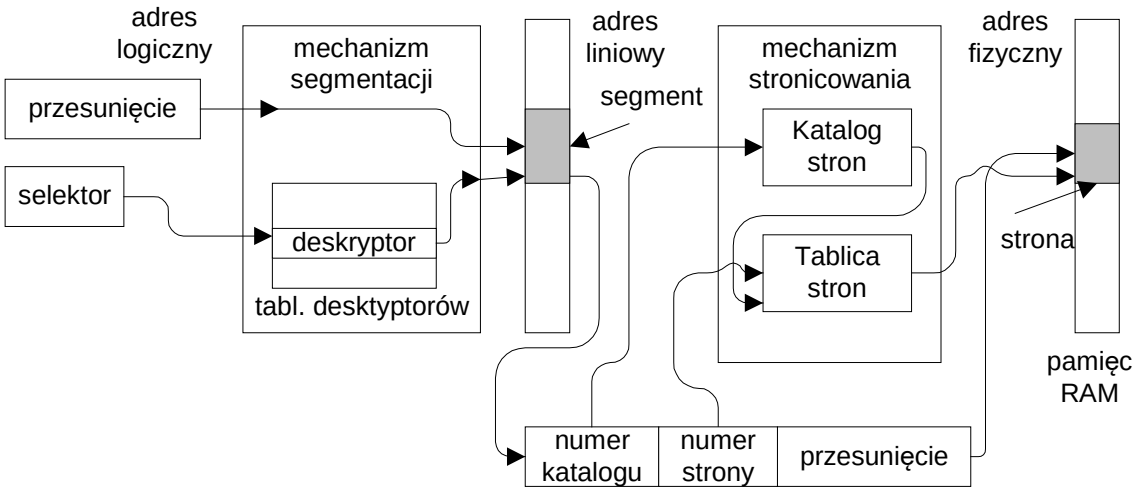
- Mechanizm segmentacji
- Mechanizm stronicowania

Segmentacja	Ochrona obszarów pamięci używanych przez procesy przed dostępem przez inny proces
	Zapewnienie przemieszczalności programów
Stronicowanie	Realizacja pamięci wirtualnej (większej niż fizyczna)
	Rozwiązanie problemu fragmentacji pamięci (pokawałkowania na niespójne obszary)
	Ochrona obszarów pamięci

Tab. 3-1 Mechanizmy sprzętowe zarządzania pamięcią

3.3.1 Segmentacja

Mechanizmem stosowanym do wzajemnego odizolowania procesów jest mechanizm segmentacji. Umożliwia on również relokację. Segmentacja jest mechanizmem sprzętowym polegającym na podziale pamięci operacyjnej na ciągłe bloki nazywane segmentami. Próba siegnięcia przez proces poza przydzielone mu segmenty kończy się tak zwanym wyjątkiem i dalej usunięciem procesu.



Rys. 3-6 Mechanizm zarządzania pamięcią w procesorach IA-32

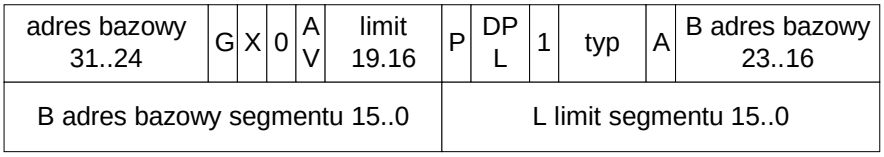
W procesorach IA-32 pamięć logiczna jest dwuwymiarowa. Adres składa się z:

- selektora (ang. *selector*) - określa segment pamięci
- przesunięcia (ang. *offset*) - wyznacza adres wewnątrz segmentu.

Każdy segment charakteryzuje się takimi parametrami jak:

- początek bloku,
- wielkość
- atrybuty

Parametry segmentu przechowywane są w 8 bajtowym rekordzie nazywanym deskryptorem segmentu.



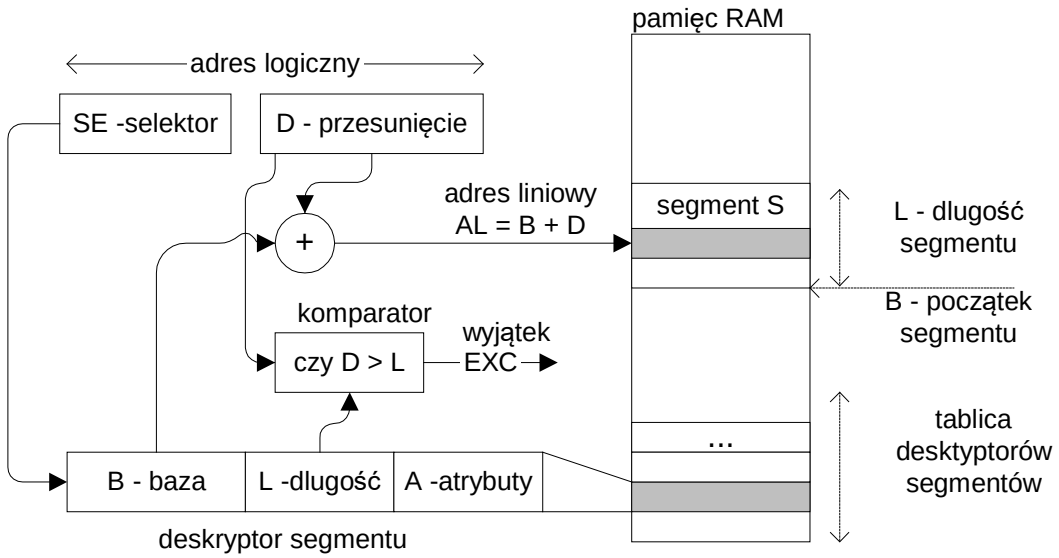
Rys. 3-7 Zawartość deskryptora segmentu pamięci w procesorach IA-32

B - adres bazowy segmentu
L - długość segmentu
G - sposób interpretacji limitu segmentu (0 – bajty, 1 – strony 4KB),
DPL - poziom uprzywilejowania segmentu,
P - bit obecności segmentu (używany w pamięci wirtualnej),
AV - nie używany,
A - mówi czy deskryptor jest używany.

Deskryptory są przechowywane w dwóch rodzajach tablic:

- globalnej tablicy deskryptorów GDT (ang. *Global Descriptor Table*)
- lokalnej tablicy deskryptorów LDT (ang. *Local Descriptor Table*).

W systemie istnieje:
- jedna tablica GDT - opisuje segmenty widoczne dla wszystkich procesów
- wiele lokalnych tablic deskryptorów LDT (ang. *Local Descriptor Table*), opisujących prywatne segmenty procesów

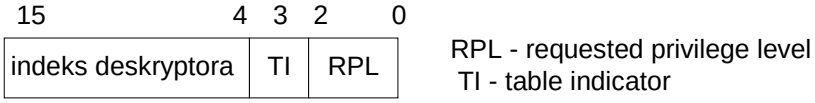


Rys. 3-8 Uproszczony schemat mechanizmu segmentacji

Adres logiczny składa się z:
- selektora segmentu SE
- przesunięcia D.

Funkcje selektora pełni jeden z rejestrów segmentowych:

- dla kodu selektorem jest rejestr CS.,
- dla danych rejestr DS.,
- dla stosu SS.



Rys. 3-9 Zawartość rejestru selektora segmentu

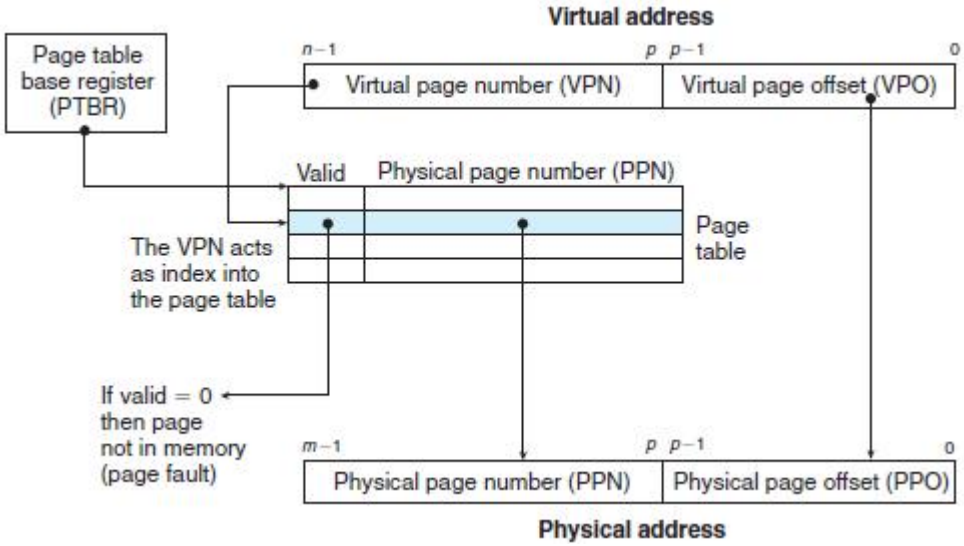
Selektor zawiera:
- indeks deskryptora - położenie segmentu znajdującego się w tablicy deskryptorów,
- TI - określa o którą tablicę chodzi (0 – GDT, 1 - LDT)
- RPL - żądany poziom uprzywilejowania – określa poziom uprzywilejowania procesu.

Adres liniowy jest to suma pobieranego z pola adresowego rozkazu przesunięcia D i adresu początku segmentu B pobieranego z deskryptora. Komparator sprawdza czy przesunięcie D nie wykracza poza długość segmentu L zapisanego w deskryptorze. Gdy tak się zdarzy generowany jest wyjątek EXC który powoduje wywołanie

systemu operacyjnego. System operacyjny podejmuje decyzję, co zrobić z naruszającym przydzielony segment procesem. Adres liniowy może być poddany dalszemu przetwarzaniu przez mechanizm stronicowania.

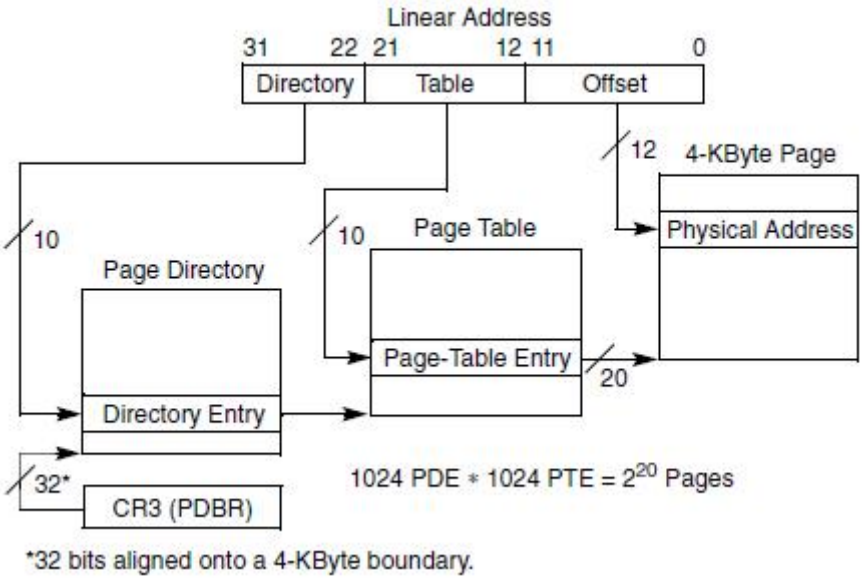
3.3.2 Stronicowanie

Mechanizm stronicowania polega na podzieleniu segmentów pamięci z adresu liniowego na części zwane stronami. Wielkość strony to zwykle 4 kB, 2 MB lub 8 MB. Podstawowy schemat stronicowania pokazuje poniższy rysunek.



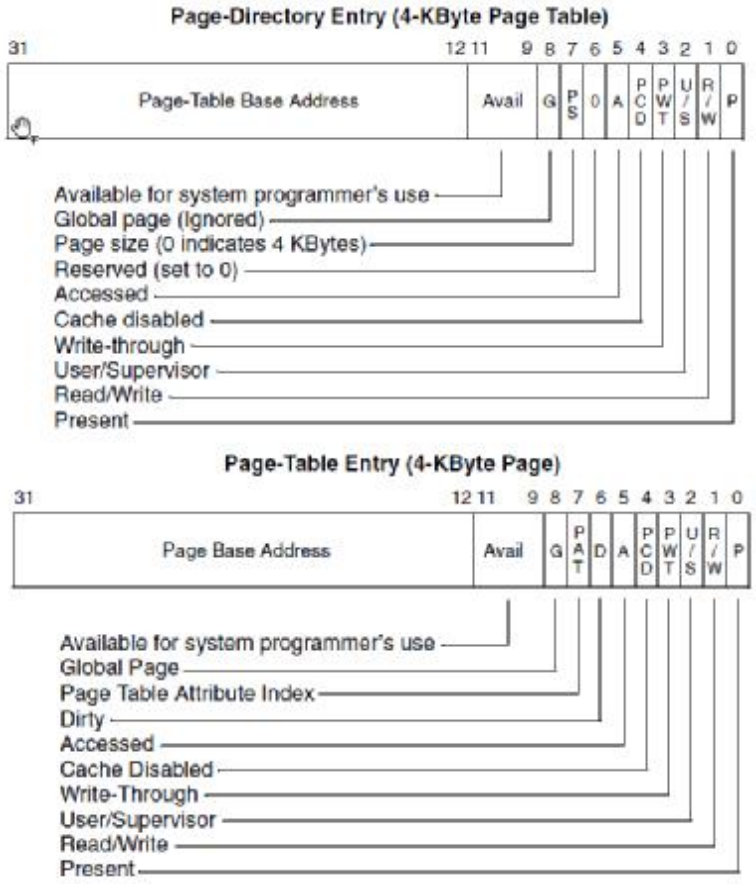
Rys. 3-10 Podstawowy schemat stronicowania

W architekturze IA-32 stosowany jest dwupoziomowy mechanizm stronicowania. Starsza część adresu wirtualnego (bity 31-22) identyfikuje tak zwany katalog, bity 21-12 określają tablicę stron a bity 11-0 adres na stronie co pokazuje poniższy rysunek.



Rys. 3-11 Dwupoziomowy schemat stronicowania

Strony adresu liniowego odwzorowywane są w strony pamięci fizycznej, przy czym mogą istnieć strony pamięci liniowej nie mające odpowiednika w pamięci fizycznej. Takie strony umieszczone są w pamięci dyskowej w tak zwanym obszarze wymiany. Próba dostępu do nieobecnej w pamięci fizycznej strony powoduje wyjątek „page fault”. W ramach obsługi tego wyjątku powoływany jest system operacyjny który sprowadza brakującą stronę z pamięci zewnętrznej do operacyjnej.



Rys. 3-12 Zawartość pozycji katalogu stron i tablicy stron dla wielkości strony 4 KB (według [12])

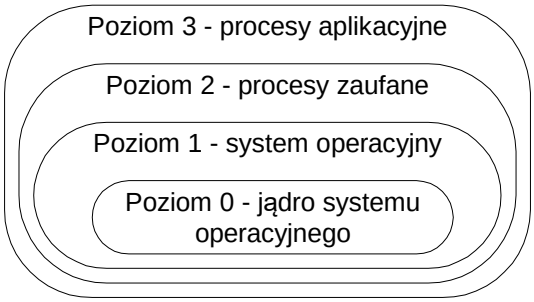
3.4 Ochrona procesora

Aby system operacyjny mógł wykonywać swe funkcje powinien mieć on dostęp do wszystkich istotnych zasobów procesora, mechanizmu zarządzania pamięcią, kontrolera przerwań i kontrolerów wejścia wyjścia. Procesy aplikacyjne nie mogą mieć dostępu do tego typu zasobów, gdyż czy to na skutek błędów czy intencjonalnie mogłyby zdestabilizować pracę systemu. We współczesnych mikroprocesorach wprowadza się dwa (lub więcej) tryby pracy procesora:

- tryb użytkownika (ang. *User Mode*)
- tryb systemowy (ang. *System Mode*).

- Tryb systemowy - proces może wykonywać wszystkie instrukcje procesora, sięgać do wszystkich obszarów pamięci i przestrzeni wejścia wyjścia.
- Tryb użytkownika - nie jest dozwolony dostęp do rejestrów: związanych z zarządzaniem pamięcią, obsługą przerwań zarządzaniem pracą procesora.

W mikroprocesorach o architekturze IA-32 ochrona procesora oparta jest o koncepcję poziomów ochrony (ang. *Privilege Level*).



Rys. 3-13 Poziomy ochrony w mikroprocesorach Intel

3.5 System operacyjny

3.5.1 Funkcje systemu operacyjnego

System operacyjny jest zbiorem programów sterujących pracą komputera. Będzie omówiony na przykładzie systemu Linux.

- Oddziela użytkownika od złożonego sprzętu i tworzy środowisko w którym wykonują się programy.
- Dostarcza interfejsu do komunikacji z użytkownikiem.

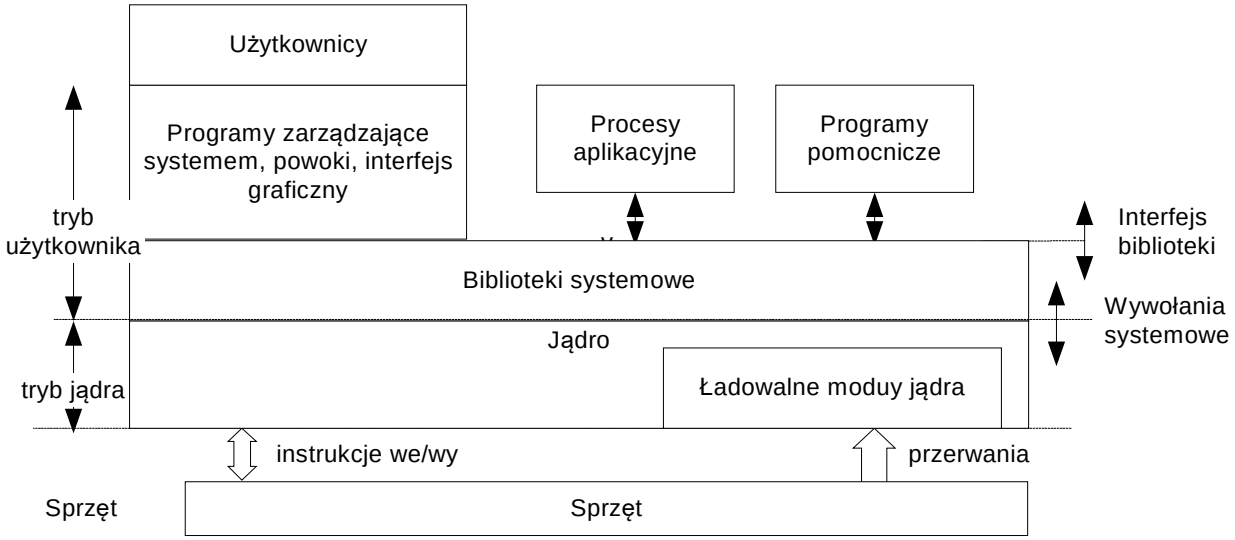
Najważniejsze cechy systemu Linux są następujące:

- Implementuje abstrakcję procesów.
- Jest w stanie obsługiwać wielu użytkowników. Zaimplementowane mechanizmy kontroli dostępu do zasobów pozwala na bezpieczne ich współistnienie.
- Implementuje abstrakcję plików. Plikami są tak pliki regularne, katalogi jak i urządzenia. Umożliwiają dostęp do bardzo nieraz różnych zasobów w jednolity sposób.
- Implementuje pamięć wirtualną. Pamięć podzielona jest na strony a dostęp do stron jest jednolity niezależnie czy są one umieszczone w pamięci operacyjnej czy zewnętrznej.
- Implementuje zaawansowane mechanizmy ochrony zasobów oparte na segmentacji pamięci i dwóch trybach pracy procesora - trybie systemowym (ang. *system*) i użytkownika (ang. *user*). Pamięć używana przez jeden proces jest chroniona przed innymi procesami. Umożliwia to implementację niezawodnie działającego oprogramowania.
- Implementuje stos protokołów TCP/IP. Zapewnia to możliwość budowy systemów rozproszonych i dostęp do sieci Internet.
- Umożliwia wykorzystanie wielu procesorów zapewniając przetwarzanie równoległe. Obsługiwany jest model SMP (ang. *Symetric Multiprocessing*).

3.5.2 Składowe systemu

System składa się z trzech głównych elementów

- Jądro – odpowiada za realizację wszystkich istotnych abstrakcji systemu: pamięci witrualnej, procesów i plików.
- Biblioteki systemowe – określają standardowy zestaw wywołań systemowych za pomocą których aplikacje mogą współdziałać z jądrem.
- Programy i demony systemowe – programy wykonujące funkcje systemowe i pomocnicze. W tym demony systemowe – programy pracujące w sposób ciągły i realizujące różne funkcje (ftp, demon sieciowy, itd.).



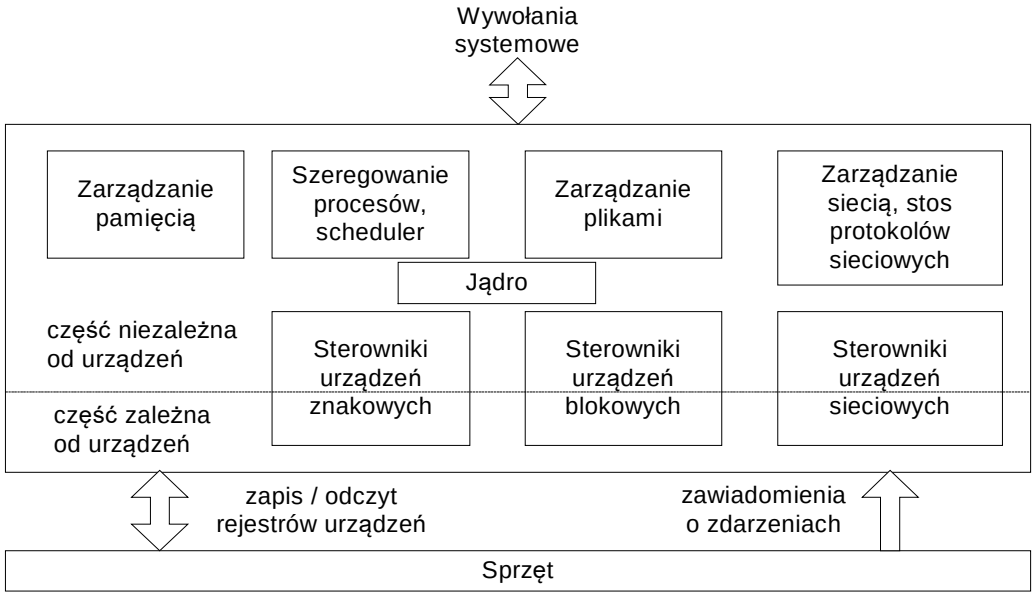
Rys. 3-14 Ogólny schemat systemu Linux

3.5.3 Jądro

Jądro dostarcza wszystkich podstawowych funkcji do działania komputera. Należą do nich następujące funkcje:

- Zarządzania pamięcią
- Tworzenie wątków i procesów
- Szeregowanie procesów
- Obsługa mechanizmów komunikacji międzyprocesowej IPC (ang. *Inter Process Communication*)
- Obsługa systemu wejścia wyjścia
- Obsługa systemu plików
- Obsługa sieci

Interfejs do usług jądra realizowany jest poprzez biblioteki systemowe. Dostarczają one bardziej złożonych wersji podstawowych usług systemowych.



Rys. 3-2 Koncepcyjny schemat jądra Linuksa

1.1.1 Tryb jądra i tryb użytkownika

Cały kod jądra i używane przez ten kod struktury danych utrzymywane są w jednej przestrzeni adresowej.

- Kod jądra wykonywany jest w uprzywilejowanym trybie procesora (ang. *Privileged Mode*),
- procesy wykonywane są w trybie użytkownika (ang. *User Mode*).

W trybie użytkownika pewne potencjalnie niebezpieczne instrukcje nie mogą być wykonywane. Do potencjalnie niebezpiecznych instrukcji zaliczamy:

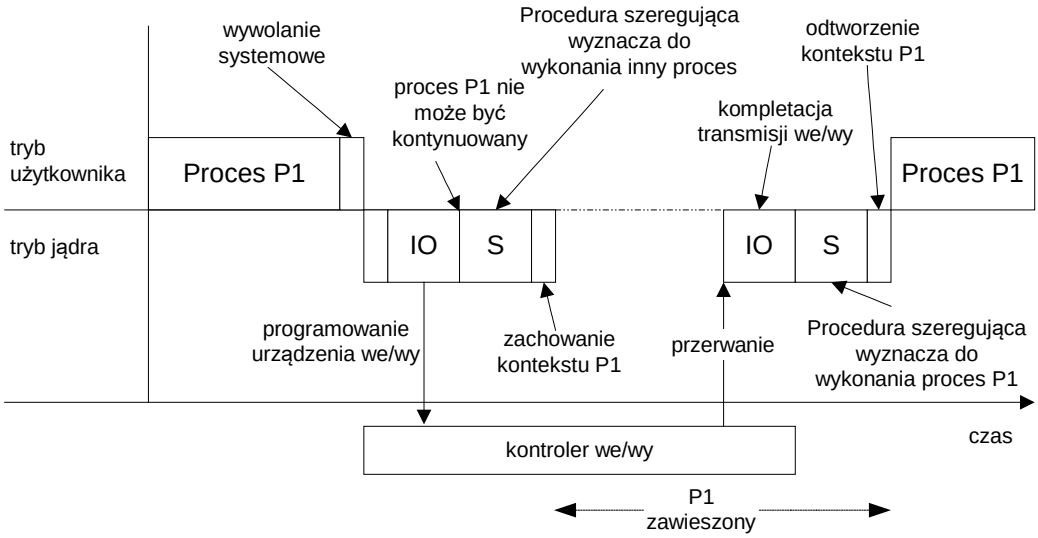
- Instrukcje wejścia / wyjścia, dokonujące manipulacji na urządzeniach zewnętrznych
- Instrukcje dostępu do konfiguracyjnych i sterujących rejestrów procesora
- Instrukcje zmiany niektórych flag procesora
- Instrukcje zatrzymywania procesora

Próba wykonania niebezpiecznych instrukcji w trybie użytkownika powoduje powstanie wyjątku (ang. *Exception*) i w konsekwencji zakończenie procesu. Gdy proces potrzebuje wykonania akcji której nie może sam przeprowadzić , na przykład kontaktu z urządzeniem zewnętrznym, formułuje odpowiednie zlecenie w postaci wywołania systemowego (ang. *System Call*) i przekazuje te zlecenie do jądra. Następuje przełączenie procesu w tryb jądra (ang. *Kernel Mode*).

Przykład:

```
read(int fh, void * bufor, int size).
```

Jądro może zapoczątkować żądaną przez proces akcję, np. zainicjować transmisję z/do urządzenia wejścia/wyjścia. Do czasu jej zakończenia proces nie może być kontynuowany. Dlatego też sterowanie może nie powrócić do wykonującego wywołanie systemowe procesu, gdyż ten i tak nie może być kontynuowany. Zamiast tego wykonywana jest procedura szeregująca procesy w wyniku której wyznaczany jest kolejny proces który może być kontynuowany. Po jakimś czasie żądane przez proces zasoby staną się dostępne, co zostanie zasygnalizowane przerwaniem od kontrolera urządzenia wejścia/wyjścia. Wtedy procedura szeregująca może ponownie przekazać sterowanie do zawieszonego procesu i będzie on w trybie użytkownika kontynuowany.

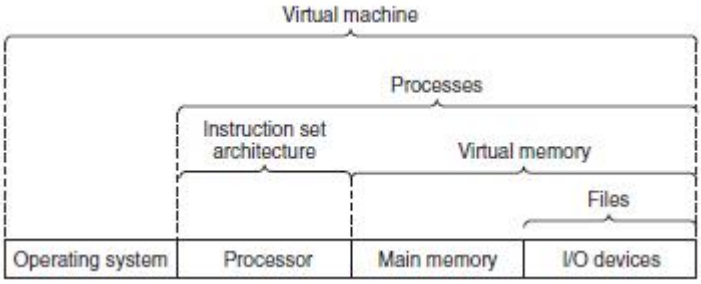


Rys. 3-3 Realizacja wywołania systemowego, przejście procesu P1 pomiędzy trybami jądra i użytkownika

3.6 Podstawowe abstrakcje informatyki

Współczesny system komputerowy jest bardzo konstrukcją bardzo złożoną. W opanowaniu tej złożoności pomagają różnorodne abstrakcje których cechą jest uproszczenie rzeczywistego komputera i sprowadzeniu go prostrzego modelu. Istotne abstrakcje to:

- ISA - poziom instrukcji maszynowych (ang. *Instruction Level Architecture*). Komputer opisywany jest jako zbiór rejestrów na których operują wykonywane sekwencyjnie instrukcje. Pomijany jest fakt że niektóre instrukcje mogą być wykonywane równolegle.
- Pamięć wirtualna (ang. *Virtual memory*) – pamięć rozpatrywana jest jako liniowa tablica komórek pamięci. Każdy z procesów widzi ją w jednakowy sposób.
- Pliki (ang. *Files*) – Podstawowe urządzenia wejścia wyjścia widziane są jako pliki czyli urządzenia abstrakcyjne na których można wykonywać kilka operacji: otwarcia, zapisu, odczytu, szukania, zamknięcia. Dostęp jest sekwencyjny i następuje począwszy od bieżącej pozycji pliku.
- Proces (ang. *Process*) – jest wirtualnym procesorem który ma iluzję że posiada na własność procesor. W rzeczywistości dzieli o z innymi procesami.
- System operacyjny (ang. *Operating system*). Zarządza komputerem, jego usługi dostępne są poprzez ustalony interfejs API (ang. *Application Program Interface*).
- Maszyna wirtualna (ang. *Virtual Machine*). Składa się z systemu operacyjnego, procesora, pamięci i urządzeń wejścia wyjścia. W rzeczywistym komputerze może być wykonywanych wiele maszyn wirtualnych.

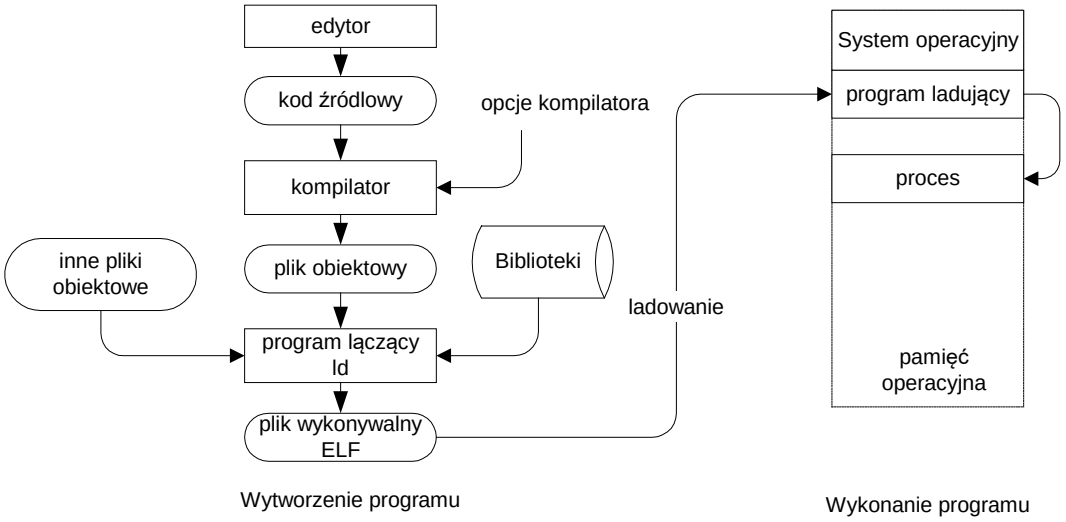


Rys. 3-15 Podstawowe abstrakcje i ich zależności (wg.)

4. Kompilacja, pierwsze programy

4.1 Jak kod przekształca się w proces.

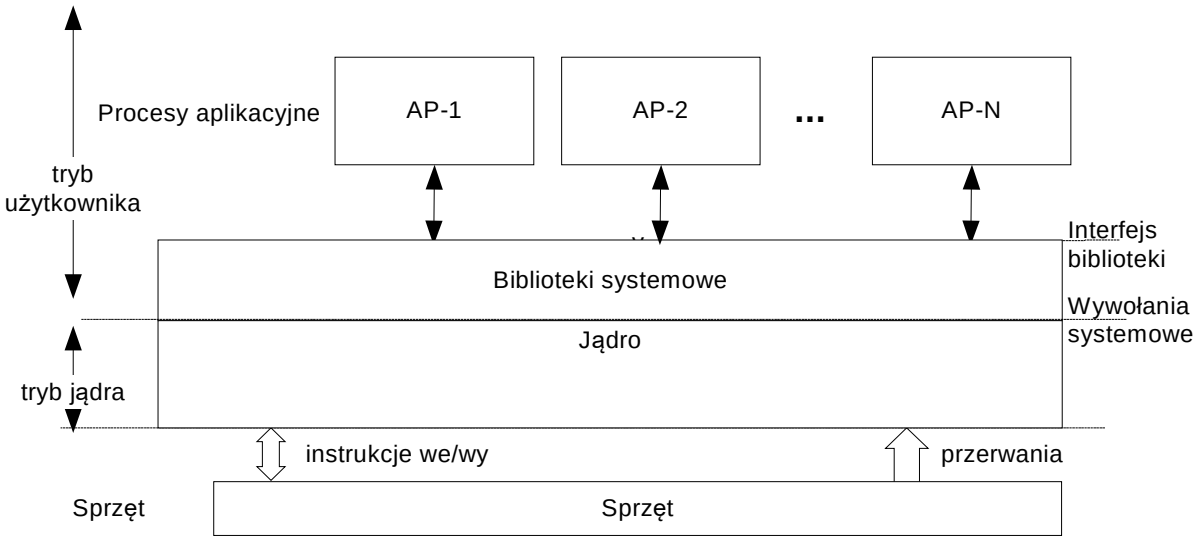
Kod aplikacji tworzony jest w określonym języku programowania w postaci pliku źródłowego. Plik źródłowy przetwarzany jest następnie przez kompilator (w przypadku programów w assemblerze jest program as) do tak zwanego pliku obiektowego. Dalej pliki obiektowe (może być ich więcej) łączone są przez program łączący `ld` (ang. *linker*) w program wykonywalny zapisany w formacie ELF (ang. *Executable and Linkable Format*). Następnie program wykonywalny przekształcany jest w wykonujący się proces, co wykonywane jest przez program ładujący systemu operacyjnego (ang. *loader*). Proces tworzenia i wykonywania programu pokazany jest na poniższym rysunku.



Rys. 4-1 Przebieg procesu wytworzenia i wykonania programu

4.2 Program w środowisku systemu operacyjnego

Programy aplikacyjne wykonują się w środowisku systemu operacyjnego. Mają one postać procesów. Proces jest wykonującym się programem korzystającym z wirtualnego procesora, wirtualnej pamięci i wirtualnych urządzeń wejścia wyjścia. Proces aplikacyjny może się komunikować z systemem operacyjnym za pomocą tak zwanych wywołań systemowych co pokazuje poniższy rysunek.



Rys. 4-1 Ogólny schemat systemu Linux

Wywołania systemowe są sposobem komunikacji pomiędzy procesem aplikacyjnym a systemem operacyjnym. Proces aplikacyjny może żądać od systemu wykonania pewnych usług, na przykład instrukcji wejścia/wyjścia. Przykładami usług systemu operacyjnego może być wyprowadzanie danych na konsolę, dostęp do plików, wykonywanie usług komunikacji sieciowej czy dostęp do interfejsu graficznego. Przykładem wywołania systemowego jest wykonanie

funkcji `exit(nr)` która powoduje zakończenie procesu i przekazanie do systemu operacyjnego kodu powrotu `nr`. Przykład użycia funkcji `exit` podany jest poniżej.

```
#include <stdlib.h>
int main(void) {
   exit(5);
}
```

Przykład 4-1 Program `exit.c` - użycie funkcji `exit`

Powyższy program napisany jest w języku C i używa funkcji bibliotecznej `exit` która jest opakowaniem wywołania systemowego. Powyższy program można skompilować z opcją –S i zobaczyć jak wywoływana jest funkcja `exit`.

```
$gcc exit.c –o exit.s -S
```

Wywołanie systemowe wykonywane jest za pomocą przerwania programowego INT 80. Poszczególne funkcje wykonywane jako wywołania systemowe są ponumerowane. Numerom odpowiadają poszczególne funkcje. Zestawienie wywołań systemowych można znaleźć w pliku `/usr/include/i386-linux-gnu/asm/unistd_32.h`

Ćwiczenie:

Obejrzyj plik zawierający numery wywołań systemowych

```
#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
…
```

Przykład 4-2 Fragment pliku nagłówkowego `/usr/include/i386-linux-gnu/asm/unistd_32.h` definiującego wywołania systemowe.

Wywołanie systemowe przebiega w następujący sposób:

1. W rejestrze EAX umieszcza się numer wywołania systemowego
2. W innych rejestrach umieszcza się pozostałe parametry wywołania
3. Wykonuje się przerwanie programowe INT80

Przykład wykonania wywołania systemowego o numerze 1 `exit` podany jest poniżej.

```
#Przeznaczenie: Program konczy się przekazując kod powrotu
#do procesu macierzystego
#Kod powrotu moze byc sprawdzony przez polecenie
# echo $?
#
# after running the program
#
#Zmienne:
# %eax number wywolania systemowego
# %ebx kod powrotu
#
.section .data
.section .text
.globl _start
_start:
movl $1, %eax # numer wywolania systemowego do EAX
# wywolanie 1 exit – zakonczenie procesu
movl $33, %ebx # to jest kod powrotu zwrocony do OS
# echo $?
int $0x80 # wywolanie jadra
```

Przykład 4-3 Program w assemblerze wykonujący wywołanie systemowe `exit`

W powyższym programie użyto prostego sposobu przekazywania danych z programu do jego procesu macierzystego którym w tym przypadku jest `shell`. Polecenie `echo $?` zwraca wartość zmiennej (shella) `$?` której wartością jest kod statusu (powrotu) procesu potomnego. Kod powrotu będzie równy zawartości rejestru `%bl` bezpośrednio prze wykonaniem funkcji `exit`. Jeśli ostatnią instrukcją w programie przed `exit` jest wpisanie czegoś do `%ebx/%rbx`, to echo $? zwróci kod z najniższych 8 bitów, czyli z `%bl`.

4.3 Kompilacja, łączenie i uruchomienie programu

Assembler wywołuje się poleceniem `as`. Opis opcji można uzyskać pisząc:

```
as –-help
```

Ogólna postać wywołania to:

```
as [opcje] nazwa_pliku
```

Typowe przypadki użycia dane są dalej. Kompilacja pliku źródłowego `plik.s` do pliku obiektowego `plik.o`.

```
as plik.s –o plik.o
```

Kompilacja pliku źródłowego `plik.s` do pliku obiektowego `plik.o` z informacją dla debuggera.

```
as plik.s –g –o plik.o
```

Program łączący uruchamia się poleceniem `ld`. Ogólna postać wywołania to:

```
ld [opcje] nazwa_pliku
```

W poniższym przykładzie łączymy plik obiektowy `plik.o` w kod wykonywalny `plik`.

```
ld plik.o –o plik
```

Powinien powstać plik wykonywalny o nazwie `plik.`

Dla przykładu kompilacja i uruchomienie wygląda następujące:

```
$ as exit.s -o exit.o
$ ld exit.o -o exit
$ ./exit; echo $?
33
```

Przykład 4-4 Kompilacja i uruchomienie programu exit

4.4 Informacje o pliku obiektowym

Ogólne informacje o pliku wykonywalnym można uzyskać za pomocą programu `file` co pokazano poniżej.

```
$ file exit
exit: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically
linked, not stripped
```

Kolejne informacje o pliku wykonywalnym można uzyskać za pomocą programu objdump. Program ten wywołuje się następująco:

```
objdump <option(s)> <file(s)>
```

Musi być podana przynajmniej jedna z opcji:

-a, --archive-headers	Display archive header information
-f, --file-headers	Display the contents of the overall file header
-p, --private-headers	Display object format specific file header contents
-P, --private=OPT,OPT...	Display object format specific contents
-h, --[section-]headers	Display the contents of the section headers
-x, --all-headers	Display the contents of all headers
-d, --disassemble	Display assembler contents of executable sections
-D, --disassemble-all	Display assembler contents of all sections
-S, --source	Intermix source code with disassembly
-s, --full-contents	Display the full contents of all sections requested

Przykłady użycia programu podano poniżej

```
$objdump -d exit.o
exit.o:     file format elf32-i386
Disassembly of section .text:
00000000 <_start>:
   0: b8 01 00 00 00          mov    $0x1,%eax
   5: bb 21 00 00 00          mov    $0x21,%ebx
   a: cd 80                   int    $0x80
```

Przykład 4-5 Disassemblacja programu exit – kod

```
objdump -x exit.o
exit.o:     file format elf32-i386
exit.o
architecture: i386, flags 0x00000010:
HAS_SYMS
start address 0x00000000
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000000c  00000000  00000000  00000034  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000040  2**0
                  ALLOC
SYMBOL TABLE:
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 g       .text  00000000 _start
```

Przykład 4-6 Disassemblacja programu exit – segmenty

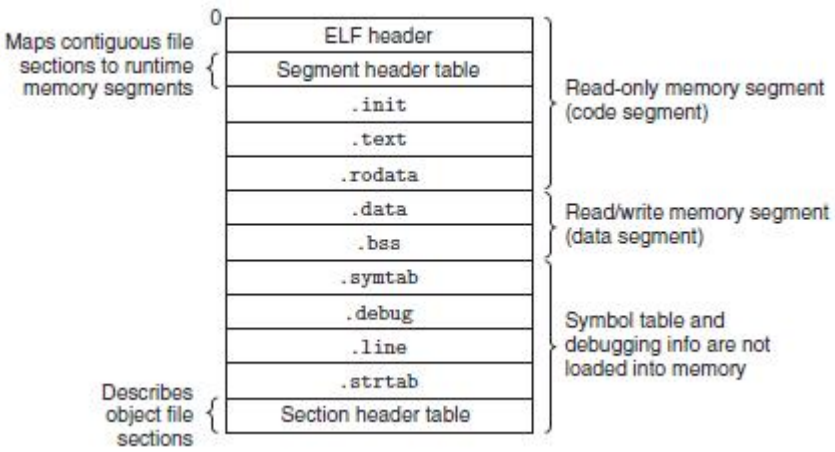
4.5 Pliki ELF

Pliki obiektowe, wykonywalne, biblioteki, obrazy pamięci) zapisywane są w formacie ELF (ang. *Executable and Linkable Format*). Format ich opisany jest w [4]. Pliki obiektowe mogą być w jednym z trzech formatów:

- Relokowalne pliki obiektowe (ang. R*elocable object file*)– zawierają binarny kod i dane które mogą być łączone z innymi plikami obiektowymi i finalnie dają wykonywalne pliki obiektowe.
- Wykonywalne pliki obiektowe (ang. *Executable object file)* – zawierają binarny kod i dane które mogą być bezpośrednio załadowane do pamięci.

- Współdzielone pliki obiektowe (ang. *Shared object file)* – specjalny typ pliku obiektowego który może być łączony dynamicznie używany do tworzenia bibliotek współdzielonych.

Format wykonywalnego pliku obiektowego pokazany jest poniżej.



Rys. 4-2 Format wykonywalnego pliku ELF

Opis sekcji pliku ELF podaje poniższa tabela.

`.init`	Kod inicjalizacyjny programu – funkcja `_init`
`.text`	Kod programu
`.rodata`	Dane tylko do odczytu (read only)
`.data`	Zainicjowane zmienne języka C
`.bss`	Niezaicjowane dane języka C. Podane są typy i wielkości obszarów
`.symtab`	Tablica symboli - Informacje o funkcjach i zmiennych globalnych
`.rel.text`	Relocable text – lista lokacji w segmencie kodu `.text` które muszą być zmodyfikowane gdy linker łączy ten plik z innymi.
`.rel.data`	Relocable data– lista lokacji w segmencie danych które muszą być zmodyfikowane gdy linker łączy ten plik z innymi. Chodzi o zmienne globalne zdefiniowane w innych modułach.
`.debug`	Informacje dla debugera. Zawiera kod źródłowy, listę zmiennych i ich typów. Otrzymywane gdy skompilujemy program z opcją –g.
`.line`	Informacje dla debugera. Zawiera informacje które linie kodu źródłowego odpowiadają liniom kodu maszynowego.

Tab. 4-1 Typy sekcji pliku ELF

Części składowe pliku w formacie ELF mogą być odczytane za pomocą programu `readelf`. Opcje programu uzyskujemy za pomocą polecenia `readelf –h`.

```
$readelf -h
readelf: Warning: Nothing to do.
Usage: readelf <option(s)> elf-file(s)
 Display information about the contents of ELF format files
 Options are:
  -a --all                   Equivalent to: -h -l -S -s -r -d -V -A -I
  -h --file-header           Display the ELF file header
  -l --program-headers       Display the program headers
  -S --section-headers       Display the sections' header
  -g --section-groups        Display the section groups
  -t --section-details       Display the section details
  -e --headers               Equivalent to: -h -l -S
  -s --syms                  Display the symbol table
  --dyn-syms                 Display the dynamic symbol table
  -n --notes                 Display the core notes (if present)
  -r --relocs                Display the relocations (if present)
  -u --unwind                Display the unwind info (if present)
  -d --dynamic               Display the dynamic section (if present)
```

Tab. 4-2 Opcje programu readelf

Plik ELF składa się z nagłówka oraz sekcji. Nagłówek zawiera ogólne informacje o pliku co pokazuje poniższy przykład.

```
$ readelf -h exit
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048054
  Start of program headers:          52 (bytes into file)
  Start of section headers:          252 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         1
  Size of section headers:           40 (bytes)
  Number of section headers:         5
```

Przykład 4-7 Dane nagłówka pliku exit

Pełną informację o pliku ELF uzyskujemy za pomocą polecenia `readelf –a nazwa_pliku` co pokazuje poniższy przykład.

```
$readelf -a exit
...
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        08048054 000054 00000c 00  AX  0   0  1
  [ 2] .shstrtab         STRTAB          00000000 0000d9 000021 00      0   0  1
  [ 3] .symtab           SYMTAB          00000000 000060 000060 10      4   2  4
  [ 4] .strtab           STRTAB          00000000 0000c0 000019 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
There are no section groups in this file.
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x00060 0x00060 R E 0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .text
...
```

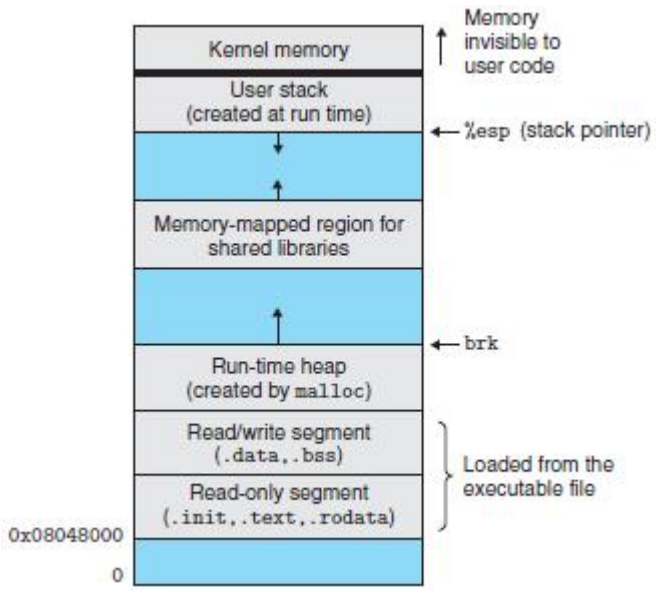
Przykład 4-8 Fragment pliku ELF programu exit

W systemie Linux program zaczyna się od adresu `08048054` co widoczne jest na powyższym przykładzie, patrz wiersz `[ 1] .text.` Koniec obszaru pamięci zajmowanej przez program zależy wielkości jego kodu i danych. Program wkonywalny w formacie ELF umieszczony jest w systemie plików. Aby stał się procesem należy wykonać jeszcze szereg czynności takich jak sprawdzenie praw dostępu, sprawdzenie formatu pliku, utworzenie potrzebnych segmentów pamięci, skopiowanie do nich kodu i danych, utworzenie nowego procesu i przekazanie sterowania do funkcji _init. Czynności te inicjuje shell gdy przekazujemy mu nazwę programu. Shell wykonuje funkcję systemu operacyjnego execv(...) która wykonuje dalsze czynności.

W ogólności znajdujący się już w pamięci operacyjnej program (czyli proces) zawiera nastepujące segmenty:

- Kod programu – obszary ./init, .text, .rodata (pamięć tylko do odczytu)
- Dane programu - obszary .data, .bss (pamięć do odczytu i zapisu)
- Sterta - obszar pamięci dynamicznej pobieranej przez funkcję malloc, kończy się adresem `brk`.
- Obszar na biblioteki współdzielone
- Stos użytkownika, zmieniający się w trakcie wykonywania programu
- Pamięć wirtualna jądra

Poszczególne obszary pamięci zajmowane przez pokazuje poniższy rysunek.



Rys. 4-3 Mapa pamięci procesu (według [4])

Chciaż jak widać program może potencjalnie zająć 2 GB pamięci prawie zawsze potrzebuje znacznie mniej. Nasz przykładowy program `exit` potrzebuje tylko 16 bajtów. W czasie normalnej pracy w systemie jest wiele programów. Można zadać sobie pytanie jak to jest że ich przestrzenie pamięci się nie nakładają, wszak każdy z nich zaczyna się od adresu 0x0848000. Należy zaznaczyć że na powyższym rysunku pokazane są liniowe adresy wirtualne. Adresy te są zamieniane na adresy rzeczywiste przez system zarządzania pamięcią. Na system ten składa się sprzętowa jednostka MMU (ang. *Memory Management Unit*) wraz z mechanizmami zarządzania pamięcią wchodzącymi w skład systemu operacyjnego. System zarządzania pamięcią zamienia adresy liniowe segmentów programu na strony pamięci wirtualnej, przydzielając ich tylko tyle ile w danej chwili jest potrzebne. Następnie strony zamieniane są na ramki pamięci fizycznej które oczywiście umieszczone są różnych lokacjach .

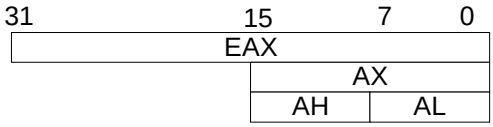
5. Rejestry pamięć i adresowanie w architekturze IA-32

5.1 Nazwy rejestrów

Nazwy rejestrów poprzedzone są znakiem %. Dla trybu 80386 określone są następujące rejestry:

Rys. 5-1 Rejestry procesora w architekturze IA-32 dla trybu chronionego

Rejestry ogólnego przeznaczenia są rejestrami 32 bitowymi. Do ich części 15 i 8 bitowych można sie odwołać poprzez wykorzystanie skrótów. Sposób odwołannia się do poszczególnych części rejestru EAX pokazuje poniższy rysunek.



Rys. 5-2 Odwołanie do różnych części rejestru EAX

Należy jednak zaznaczyć że np rejestr AH nie jest jakimś odrębnym od EAX rejestrem ale jego bitami od 8 do 15. Pokazany sposób odwoływania się do rejestrów ma korzenie historyczne. Architektura IA-32 ewoluowała do obecnej postaci z architektury 16 bitowej.

8 rejestrów 32-bit	%eax (akumulator), %ebx, %ecx, %edx, %edi, %esi, %ebp (rejestr ramki), i %esp (wskaźnik stosu)
8 rejestrów 16 bitowych	%ax, %bx, %cx, %dx, %di, %si, %bp, %sp
8 rejestrów 8 bitowych	%ah, %al, %bh, %bl, %ch, %cl, %dh, %dl
6 rejestrów segmentowych	%cs (segment kodu), %ds (segment danych), %ss (segment stosu), %es, %fs, %gs.
5 rejestrów kontrolnych procesora	%cr0, %cr2, %cr3, %cr4, and %cr8
6 rejestrów debuggera	%db0, %db1, %db2, %db3, %db6, and %db7
2 rejestry testowe	%tr6 and %tr7
8 rejestrów zmiennego przecinka, te rejestry nakładają się z 8 rejestrami MMX	%st(0), %st(1), %st(2), %st(3), %st(4), %st(5), %st(6), %st(7), %mm0, %mm1, %mm2, %mm3, %mm4, %mm5, %mm6 and %mm7
8 rejestrów 128 bitowych SSE	%xmm0, %xmm1, %xmm2, %xmm3, %xmm4, %xmm5, %xmm6 and %xmm7

Tab. 5-1 Nazwy rejestrów w assemblerze gnu

Flaga	Angielska nazwa	Opis
CF	Carry Flag	Ostatnia operacja arytmetyczna spowodowała nadmiar. Stosowana do liczb całkowitych bez znaku
ZF	Zero Flag	Ostatnia operacja arytmetyczna to zero.
SF	Sign Flag	W wyniku ostatniej operacji arytmetycznej otrzymano wynik ujemny.
OF	Overflow Flag	W wyniku ostatniej operacji arytmetycznej na liczbie uzupełnienie do 2 otrzymano nadmiar.

Tab. 5-2 Niektóre flagi procesora X86

5.2 Ogólna postać instrukcji

Instrukcje procesora IA-32 opisane są dokładnie w [11].
Ogólna postać instrukcji jest następująca:

```
label: mnemonic argument1, argument2, argument3
```

label:	Etykieta instrukcji zakończona dwukropkiem (opcjonalna)
mnemonic	Unikalna nazwa operacji
argument1, argument2, argument3	Argumenty (opcjonalne)

Gdy występują dwa argumenty lewy określa źródło a prawy przeznaczenie.

5.3 Stałe i odwołania się do rejestrów

Domyślnie assembler GNU używa tak zwanej konwencji AT&T (inna konwencja to Intel) do zapisu instrukcji. W konwencji tej:

- Argumenty natychmiastowe poprzedzone są znakiem $, np. `movl $4, %eax`
- Odwołania do rejestrów poprzedzone są znakiem %, np. `movl $4, %eax`
- W rozkazach najpierw występuje źródło a potem przeznaczenie, np. `movl $4, %eax` oznacza przesłanie wartości 4 do rejestru `%eax`. Podobnie `addl $4, %eax` oznacza dodanie 4 rejestru `%eax`. W konwencji Intel kolejność jest odwrotna.
- Wielkość operandu odnoszącego się do pamięci określona jest ostatnią literą instrukcji. I tak ‘b’, ‘w’, ‘l’ i ‘q’ specyfikują bajt (8-bit), słowo (16-bit), długie słowo (32-bit) i poczwórne słowo (64-bit) odniesienia do pamięci.
- Długie skoki i wywołania funkcji maja postać: `lcall/ljmp $`*`section`*`, $`*`offset`*
- Znak średnika ; separuje instrukcje w tej samej linii

5.4 Prefiksy instrukcji

Prefiksy modyfikują działanie niektórych instrukcji. Należą do nich instrukcje operujące na łańcuchach, blokady magistrali, zmiany domyślnego rejestru segmentowego, zmiany wielkości operandu i adresu.

- Zmiana domyślnego rejestru segmentowego może być wykonana przez dodanie przed instrukcją prefiksu `cs, ds, ss, es, fs, gs.`
- Prefiksy `data16` i `addr16` zmieniają dane/operand 32 bitowy na 16 bitowy. Podobnie `data32` i `addr32` zmieniają dane/operand 16 bitowy na 32 bitowy.
- Prefiks `lock` blokuje przerwania i dostęp do pamięci (działa z niektórymi instrukcjami).
- Prefiks `wait` powoduje oczekiwanie aż koprocesor nie zakończy poprzedniej operacji
- Prefiksy `rep, repe, repne` powodują powtórzenie instrukcji odnoszącej się do łańcucha tyle razy ile wynosi zawartość rejestru `%ecx` (lub `%cx`).

5.5 Odwołania do pamięci

5.5.1 Adresowanie i jego tryby

Ważną rolę odgrywają rozkazy odwołujące się do pamięci. Mogą to być:

- Pobranie zawartości określonej (przez adres) komórki pamięci i przesłanie jej do rejestru
- Przesłanie zawartości rejestru do okreslonej (przez adres) komórki

W pewnym uproszczeniu przesłania te realizowane są przez rozkazy:

```
mov źródło, przeznaczenie
```

Działanie polega na przesłaniu zawartości źrodła do określonego adresem przeznaczenia. Źródło może być:

- Wartością literalną określoną w rozkazie (stałą, argument natychmiastowy), np. 33
- Zawartością rejestru
- Zawartością komórki pamięci o danym adresie.

Przeznaczeniem może być:

- Rejestr
- Komórka o określonym adresie

Jeżeli rozkaz odwołuje się do pamięci, znaczy to że pobiera do rejestru zawartość komórki o określonym adresie lub też zawartość rejestru lub określona w rozkazie wartość ma być zapisana w komórce pamięci o danym adresie. Procesory IA-32 używają dwuwymiarowego adresowania pamięci. Adres składa się z selektora segmentu i przesunięcia.

Rys. 5-3 Składniki adresu pamięci w architekturze IA-32

Selektor segmentu jest wskaźnikem na tak zwany deskryptor segmentu który opisuje położenie i atrybuty danego segmentu pamięci.Selektor segmentu jest zawarty w jednym z rejestrów segmentowych co pokazane jest poniżej.

Type of Reference	Register Used	Segment Used	Default Selection Rule
Instructions	CS	Code Segment	All instruction fetches.
Stack	SS	Stack Segment	All stack pushes and pops. Any memory reference which uses the ESP or EBP register as a base register.
Local Data	DS	Data Segment	All data references, except when relative to stack or string destination.
Destination Strings	ES	Data Segment pointed to with the ES register	Destination of string instructions.

Tab. 5-3 Domyślne rejestry segmentowe

Rejestry segmentowe dobierane są domyślnie dla danego typu instrukcji co pokazuje powyższa tabela. Mogą jednak być nadpisane za pomocą operatora dwukropka **:** co pokazuje poniższy przykład.

```
mov %eax, es:(%ebx)
```

Sposób obliczania tego adresu określa konwencja użyta w programie GNU assembler. GNU assembler używa konwencji AT&T (inna konwencja to INTEL). Konwencja AT&T określa sposób odwołania się instrukcji do pamięci. Jest ona postaci:

segment:disp(base, index, scale) **Wzór 5-1**

Pole	Nazwa	Uwagi
base	rejestr bazowy	Koniecznie rejestr
index	rejestr indeksowy	Koniecznie rejestr
scale	mnożnik 1,2,4 lub 8	Gdy parametru brak, przyjmuje się 1
disp	przesunięcie	Jest to wartość stała, zwykle etykieta lub liczba

adres = base + index * scale + disp Wzór 5-2

Pamięć operacyjną możemy traktować jak liniową tablicę bajtów. Oznaczmy ją jako `M[MAXADDR]` gdzie `MAXADDR` jest adresem ostatniej komórki pamięci (komórek jest `MAXADDR+1`) a ich adresy są z przedziału od 0 do `MAXADDR`. Posługując się tym zapisem możemy wyrazić zapis i odczyt do/z komórki o adresie `addr` następujący sposób:

Zapis wielkości `x` pod adres `addres` `M[addres] = x`
Odczyt wielkości `x` z adresu `addres` `x = M[addres]`

Poprzez odwołanie się do pamięci rozumiemy zapis lub odczyt. Tak więc powyższe wzory okreslają skąd w pamieci pobieramy operand lub po jaki adres w pamięci zapisujemy wynik.

W powyższym wzorze nie wszystkie elementy muszą wystapić, w szczególności segment jest ustalany domyślnie . Może pewnych elementów brakować, ale przecinki muszą pozostać. Należy zauważyć że %`base` i %`indeks` musza być rejestrami a `disp` i `scale` stałymi.

disp(%base,%index,scale)	adres = disp + %base + %indeks * scale
disp(,%index,scale)	adres = disp + %indeks * scale
disp(%base)	adres = disp + %base
(indeks)	adres = %indeks

Tab. 5-4 Sposoby obliczania adresów przy odwołaniach do pamięci

Tryb adresowania indeksowanego (ang. *indexed addressing mode*)
Tryb ten wykorzystuje pola `disp`, `index` i `scale` wyrażenia ogólnego.

```
disp(,%index,scale)    adres = disp + %indeks * scale
```

Jako rejestr indeksowy można użyć dowolnego rejestru ogólnego przeznaczenia. Można także użyć mnożnika 1,2,4,8 dla zawartości rejestru indeksowego by sięgać do kolejnych bajtów, słów lub długich słów tablicy. Dla przykładu by sięgnąć do trzeciego bajtu łańcucha zaczynającego się od etykiety `string_start`, ładujemy 2 do %`ecx` i postępujemy jak dalej.

```
movl $2, %ecx
movl string_start(,%ecx,1), %eax
```

Tryb adresowania bazowo indeksowego (ang. *base pointer addressing mode*)
Tryb ten wykorzystuje pola `disp` i `%base` wyrażenia ogólnego.

```
disp(%base)          adres = disp + %base
```

Do przesunięcia określonego w polu `disp` dodawany jest adres zawarty w rejestrze `%base.` Przykład dany jest poniżej gdzie do adresu zawartego w `%eax` dodawane jest przesunięcie 4. Zawartość lokacji pamięci o adresie `%eax + 4` (4 bajty) przesyłane są do rejestru `%ebx`.

```
movl 4(%eax), %ebx
```

Tryb adresowania bezpośredniego (ang. *direct addressing mode*)
W adresowaniu bezpośrednim w instrukcji zawarty jest adres operandu . Tryb bezpośredni wykorzystuje część `disp` danego wyżej ogólnego wyrażenia.

```
disp     adres = disp
```

Instrukcja wykorzystująca ten tryb może mieć postać:

```
movl ADDRESS, %eax
```

W tym przypadku do rejestru `%eax` przesyłane jest słowo (4 bajty) **spod adresu** `ADDRESS`. Podobnie rozkaz

```
movl 0x1F22, %eax
```

oznacza przesłanie do akumulatora `%eax` czterech bajtów zaczynając od komórki o adresie `0x1F22` a nie przesłanie do `%eax` wartości `0x1F22`.

Tryb adresowania pośredniego (ang. *indirect addressing mode*)
W trybie adresowania pośredniego adres operandu zawarty jest w pewnym rejestrze indeksowym.

```
(base)      adres = base
```

Przykładowo rejestr `%eax` może zawierać adres operandu przesyłanego do `%ebx`

```
movl (%eax), %ebx
```

W powyższym przykładzie do rejestru `%ebx` przesyłamy zawartość komórki o adresie zawartym w rejestrze `%eax`. Ważny jest nawias, gdyby go nie było czyli instrukcja byłaby taka:

```
movl %eax, %ebx
```

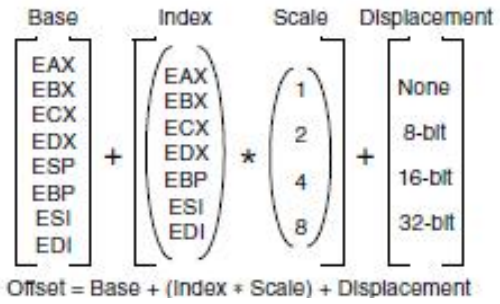
Zawartość rejestru `%eax` byłaby przesyłana do rejestru `%ebx.`

Tryb natychmiastowy (ang. *immediate mode*)
W adresowaniu natychmiastowym w instrukcji zawarta jest wprost wartość operandu.
Musi być ona poprzedzona znakiem `$` co pokazuje poniższy przykład.

```
movl $33, %eax
```

W przykładzie następuje wpisanie wartości 33 do rejestru `%ebx`.

Każdy z podanych wyżej trybów (z wyjątkiem natychmiastowego) może być użyty w instrukcji przesłań zarówno jako źródło jak i przeznaczenie.



Rys. 5-4 Sposób ustalania adresu efektywnego

Jeszcze jedno wyjaśnienie trybów adresowania podają poniższe tabele.

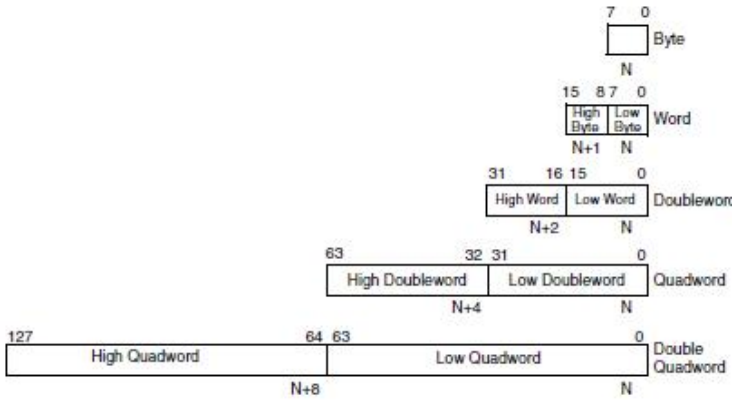
`$Imm`	wartość bezpośrednia (stała)
`%R`	Oznaczenie rejestru, %Rb – rejestr bazowy, %Ri – rejestr indeksowy
`Z[%R]`	zawartość rejestru %R
`M[addr]`	zawartość pamięci o adresie `addr`
`S`	współczynnik skali, stała 1,2,4,8,..

Typ	Forma	Operand	Nazwa
Natychmiastowy	`$Imm`	`Imm`	Natychmiastowy
Rejestr	`%R`	`Z[%R]`	Rejestr
Pamięć	`Imm`	`M[Imm]`	Absolutny
Pamięć	`(%R)`	`M[Z[%R]]`	Pośredni
Pamięć	`Imm(%R)`	`M[Imm + Z[%R]]`	Baza + przesunięcie
Pamięć	`(%Rb,%Ri)`	`M[Z[%Rb] + Z[%Ri]]`	Indeksowany
Pamięć	`Imm((%Rb,%Ri)`	`M[Imm + Z[%Rb] + Z[%Ri]]`	Indeksowany
Pamięć	`(,%Ri,S)`	`M[Z[%Ri] * S]`	Indeksowany ze skalą
Pamięć	`Imm(,%Ri,S)`	`M[Imm + Z[%Ri] * S]`	Indeksowany ze skalą
Pamięć	`(Rb,%Ri,S)`	`M[Z[%Rb] + Z[%Ri] * S]`	Indeksowany ze skalą
Pamięć	`Imm(Rb,%Ri,S)`	`M[Imm + Z[%Rb] + Z[%Ri] * S]`	Indeksowany ze skalą

Tab. 5-5 Postacie operandów, mogą to być wielkości natychmiastowe (stała), zawartość rejestru, zawartość komórki pamięci

5.5.2 Ile bajtów przesyłamy

Podstawowe typy danych dostępne w architekturze IA-32 pokazuje poniższy rysunek.



Rys. 5-5 Typy danych w architekturze IA-32

Każda instrukcja odnosząca się do przesłań z/do pamięci musi na końcu zawierać postfiks `b, w, l , q`, specyfikujący długość przesyłanych danych (1,2,4,8 bajtów). Przykładowo instrukcja:

```
movl tablica(,%edi,4), %eax
```

kończy się literą `l` co oznacza że przesyłamy liczbę long (4 bajty). Podobnie należy wyspecyfikować typ zmiennych.

skrót	długość	Opis ang.	Opis
`b`	1	`byte`	Znak
`w`	2	`word`	Słowo
`l`	4	`long`	Liczba całkowita / adres
`q`	8	`double`	Liczba float

Tab. 5-6 Sufiksy okreslające liczbę przesyłanych bajtów w gnu assembler

Deklaracja C	Typ danych Intel	Sufix assemblera	Wielkość (bajty)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	2
`long int`	Double word	`l`	4
`char *`	Double word	`l`	4
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8
`long double`	Extended precision	`t`	10/12

Tab. 5-7 rozmiary danych języka C w IA32

Wykonując przesłanie z/do pamięci należy użyć rejestru odpowiedniej długości. Tak na przykład gdy przesyłamy 4 bajty należy użyć rejestru 32 bitowego. Przykładowo

```
movl (%ecx), %eax
```

Gdy przesyłamy 2 bajty należy użyć rejestru 16 bitowego.

```
movw (%ecx), %ax
```

Gdy przesyłamy 1 bajt należy użyć rejestru 8 bitowego.

```
movb (%ecx), %al.
```

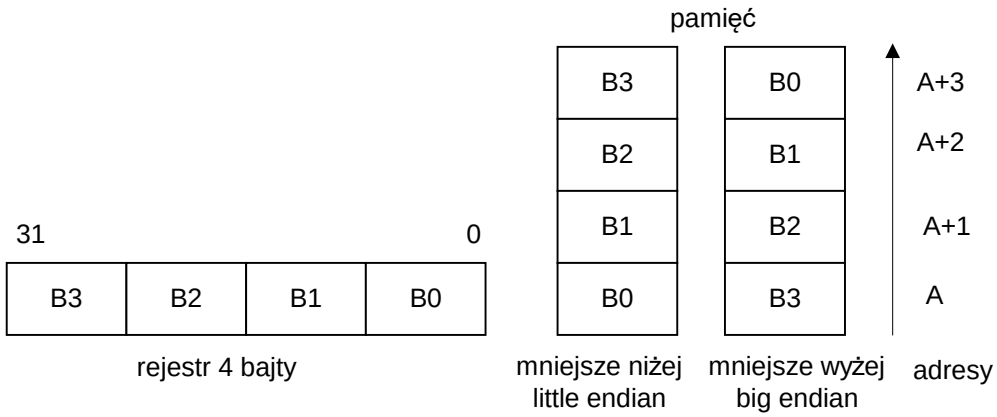
Przykład dostępu do tablicy o nazwie `tablica` poprzez adresowanie indeksowe pokazano poniżej.

```
tablica: #To są dane w tablicy zawierającej liczby long
.long 3,67,34,224,45,75,54,34,44,33,22,11,66,0
...
movl $0, %edi # przeslij 0 do rejestru indeksowego
movl tablica(,%edi,4), %eax # przeslij do eax kolejna liczbe z tablicy
...
```

Przykład 5-1 Dostęp do elementów tablicy poprzez rejestr indeksowy

5.5.3 Kolejność zapisu bajtów w pamięci.

Powiedzmy że za pomocą instrukcji `movl` przesyłamy do pamięci 4 bajty zawarte w rejestrze `%eax`. Powstaje pytanie jak będą one umieszczone w pamięci. Procesory IA32 stosują konwencję mniejszy niżej (ang. *little endian*). Polega ona na tym że najmniej znaczący bajt będzie zapisany w komórce o niższym adresie co pokazuje poniższy rysunek. Konwencja mniejszy wyżej (ang. *big endian*). Polega ona na tym że najmniej znaczący bajt będzie zapisany w komórce o wyższym adresie. Konwencję mniejszy wyżej stosują procesory Power PC, Motorola, SPARC.



Rys. 5-6 Sposoby wewnętrznej reprezentacji liczb

6. Instrukcje procesora IA-32

6.1 Ogólna postać instrukcji

Instrukcje procesora IA-32 opisane są dokładnie w [11].
Ogólna postać instrukcji jest następująca:

```
label: mnemonic argument1, argument2, argument3
```

`label:`	Etykieta instrukcji zakończona dwukropkiem (opcjonalna)
`mnemonic`	Unikalna nazwa operacji
`argument1, argument2, argument3`	Argumenty (opcjonalne)

```
Gdy występują dwa argumenty lewy określa źródło a prawy przeznaczenie.
```

Argumenty będą oznaczane następująco:

I – argument natychmiastowy (ang. *immediate*)
R – rejestr (ang. *register*)
M – pamięć (ang. *memory*)

Flagi są znacznikami aktualnego stanu procesora, zawarte są w 32 rejestrze EFLAGS. Flagi zmieniają się pod wpływem instrukcji, a także ich stan wpływa na działanie procesora.

Flaga	Angielska nazwa	Opis
`CF`	Carry Flag	Ostatnia operacja arytmetyczna spowodowała nadmiar. Stosowana do liczb całkowitych bez znaku
`ZF`	Zero Flag	Ostatnia operacja arytmetyczna to zero.
`SF`	Sign Flag	W wyniku ostatniej operacji arytmetycznej otrzymano wynik ujemny.
`OF`	Overflow Flag	W wyniku ostatniej operacji arytmetycznej na liczbie uzupełnienie do 2 otrzymano nadmiar.
`P`	Parity	Ustawiana na 1 gdy najmłodszy bajt ostatniej operacji zawiera parzystą liczbę jedynek.

Tab. 6-1 Niektóre flagi procesora X86

6.2 Instrukcje przesłań

Instrukcje przesłań tworzą bardzo istotną grupę rozkazów. Najważniejsze z nich podano w poniższej tabeli. W tabeli obowiązują nastepujące oznaczenia:

I – operand natychmiastowy
R – rejestr
M – pamięć

W poniższej tabeli rozkazy przesłań dotyczą długiego słowa (4 bajty) gdyż mają końcowkę l. Jeżeli końcówka będzie b,w,q przesłanie będzie obejmowało bajt, słowo (2 bajty) lub długie słowo (8 bajtów).

Instrukcja	Operandy	Flagi
`movl`	`I/R/M, I/R/M`	`O/S/Z/A/C`
Move – przesuń. Kopiuje słowo (4 bajty) z jednej lokacji do drugiej. Np. `movl %eax, %ebx` kopiuje zawartość rejestru `%eax` do `%ebx`		
`movb`	`I/R/M, I/R/M`	`O/S/Z/A/C`
Kopiuje 1 bajt z jednej lokacji do drugiej. Np. `movb %al, %bl` kopiuje zawartość rejestru `%al` do `%bl`		
`leal`	`M, R/M`	`O/S/Z/A/C`
Load effective address. Oblicza w standardowy sposób adres operandu M i zamiast jego zawartości ładuje jego adres do R/M. Na przykład: `leal 5(%ebp,%ecx,1),%eax` Ładuje adres obliczony jako `5 + %ebp + 1*%ecx` i zapamiętuje go w `%eax`		
`popl`	`R/M`	`O/S/Z/A/C`
Ściąga słowo ze stosu do lokacji określonej przez R/M. Jest równoważna instrukcji `movl (%esp), R/M` po której następuje `addl $4, %esp` czyli zwiększenie wskaźnika stosu o 4. Jej wariantem jest `popfl` która zdejmuje ze stosu flagi i umieszcza je w `%efl`.		
`pushl`	`R/M`	`O/S/Z/A/C`
Przesyła słowo określone przez R/M na stos. Jest równoważna instrukcji `subl $4, %esp` (zwiększenie wskaźnika stosu o 4) po której następuje `movl I/R/M, (%esp)`. Jej wariantem jest `pushfl` która zapisuje na stos flagi z `%efl`.		
`xchgl`	`R/M, R/M`	`O/S/Z/A/C`
Zamienia ze sobą wartości określone w argumentach		

Tab. 6-2 Instrukcje przesłań

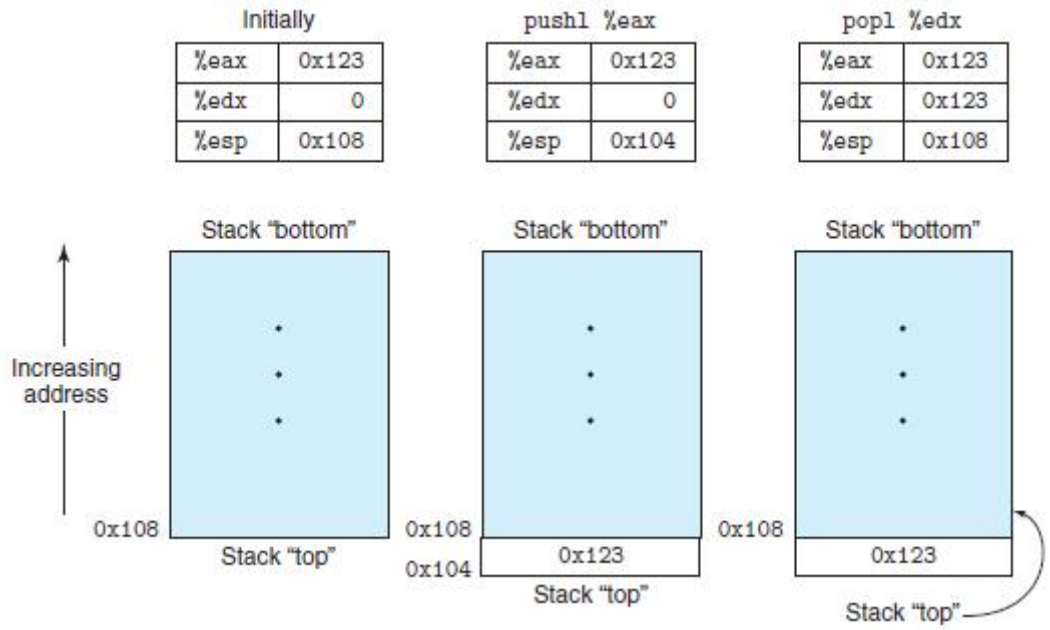
Instruction		Effect	Description
MOV	*S*, *D*	$D \leftarrow S$	Move
`movb`		Move byte	
`movw`		Move word	
`movl`		Move double word	
MOVS	*S*, *D*	$D \leftarrow$ SignExtend(S)	Move with sign extension
`movsbw`		Move sign-extended byte to word	
`movsbl`		Move sign-extended byte to double word	
`movswl`		Move sign-extended word to double word	
MOVZ	*S*, *D*	$D \leftarrow$ ZeroExtend(S)	Move with zero extension
`movzbw`		Move zero-extended byte to word	
`movzbl`		Move zero-extended byte to double word	
`movzwl`		Move zero-extended word to double word	
`pushl`	*S*	R[%esp] $\leftarrow$ R[%esp] $-$ 4; M[R[%esp]] $\leftarrow S$	Push double word
`popl`	*D*	$D \leftarrow$ M[R[%esp]]; R[%esp] $\leftarrow$ R[%esp] $+$ 4	Pop double word

Tab. 6-3 Instrukcje przesłań - podsumowania

```
1       movl $0x4050,%eax          Immediate--Register, 4 bytes
2       movw  %bp,%sp              Register--Register,  2 bytes
3       movb (%edi,%ecx),%ah       Memory--Register,    1 byte
4       movb $-17,(%esp)           Immediate--Memory,   1 byte
5       movl %eax,-12(%ebp)        Register--Memory,    4 bytes
```

Przykład 6-1 Operacje przesłań - przykłady

Z przesłań ważne są operacje `popl` i `pushl` które operują na stosie. Stos pełni ważną rolę przy implementacji funkcji i przerwaniach. Jest on implementacją kolejki LIFO (ang. *Last In First Out*). Stos może być rozumiany jako tablica wraz ze wskaźnikiem wierzchołka stosu. Operacje polegają na dopisywaniu bajtów do wierzchołka (wtedy on się zmniejsza) lub pobieraniu (wierzchołek się zwiększa). Pokazuje to przykład pochodzący z [4].



Przykład 6-2 Operacje ze stosem

Przykładowo operacje `pushl %eax` jest równoważna:

```
sub $4, %esp
movl %eax, (%esp)
```

Podobnie `popl %edx` jest równoważna:

```
movl (%esp), %edx
addl $4,%esp
```

6.3 Instrukcje arytmetyczne i logiczne

Ogólny zapis instrukcji arytmetycznych i logicznych, zgodnie z [4] podano poniżej.

6.3.1 Instrukcje operujące na liczbach całkowitych

Ogólny zapis instrukcji arytmetycznych podaje poniższa tabela. Podobnie jak poprzednio tabeli obowiązują nastepujące oznaczenia:
I – operand natychmiastowy
R – rejestr
M – pamięć

W poniższej tabeli rozkazy różnią się końcówką. Jeżeli końcówka będzie `b,w,l,q` dotyczy bajtu, słowa (2 bajty), podwójnego słowa (4 bajty) lub długiego słowa (8 bajtów). Dla przykładu instrukcje:

`addb` – dodaje bajty
`addw` – dodaje słowa
`addl` – dodaje podwójne słowa

Instrukcja	Operandy	Flagi
`addl`	`I/R/M, R/M`	`O/S/Z/A/P/C`
Dodaje operand określony w I/R/M do operandu określonego w R/M a wynik jest w drugim operandzie. Jeżeli wystąpił nadmiar lub przeniesienie to flagi O i C są ustawiane. Operuje na liczbach ze znakiem i bez znaku.		
`adcl`	`I/R/M, R/M`	`O/S/Z/A/P/C`
Dodaj z przeniesieniem (ang. *Add with carry long*). Dodaje bit przeniesienia, operand określony w I/R/M do operandu określonego w R/M. Jeżeli wystąpił nadmiar lub przeniesienie to flagi O i C są ustawiane. Używane do operacji na danych dłuższych niż słowo. Jeżeli dodawane są liczby dłuższe niż słowo, pierwsza powinna być dodana przy użyciu instrukcji `addl` a następne przy użyciu `adcl` by uwzględnić przeniesienia.		
`cdq`		`O/S/Z/A/P/C`
Konwertuje liczbę zawartą w `%eax` do liczby 8 bajtowej zawartej w rejestrach `%edx:%eax` uwzględniając znak. Instrukcja używana przed instrukcją dzielenia `idivl`.		
`cmpl`	I/R/M, R/M	O/S/Z/A/P/C
Porównaj dwie liczby całkowite. Odbywa się poprzez odjęcie pierwszego operandu od drugiego. Wynik jest odrzucany, ale flagi są ustawiane. Instrukcja zwykle używana przed skokiem warunkowym.		
`decl`	`R/M`	`O/S/Z/A/P`
Zmniejszenie o 1 rejestru bądź komórki pamięci. Gdy stosowane do bajtów używamy `decb`.		
`divl`	`R/M`	`O/S/Z/A/P`
Instrukcja przeprowadza dzielenie bez znaku. Dzielone jest podwójne słowo zawarte w `%edx:%eax` przez zawartość wskazaną w operandzie `R/M`. Rejestr `%eax` zawiera iloraz a rejestr `%edx` resztę. Gdy rejestr `%eax` jest za duży by pomieścić wynik generowane jest przerwanie 0.		
`idivl`	`R/M`	`O/S/Z/A/P`
Instrukcja przeprowadza dzielenie ze znakiem, działanie takie jak `divl`.		
`mull`	`R/M/I, R`	`O/S/Z/A/P/C`
Mnożenie liczb integer bez znaku, wynik pamiętany w drugim operandzie. Jeżeli brak drugiego operandu, ma on być w rejestrze `%eax` a wynik w postaci podwójnego słowa pamiętany w `%edx:%eax.`		
`incl`	`R/M`	`O/S/Z/A/P`
Zwiekszenie o 1 rejestru bądź komórki pamięci. Gdy stosowane do bajtów używamy `incb`.		
`imull`	`R/M/I, R`	`O/S/Z/A/P/C`
Mnożenie liczb integer ze znakiem, wynik pamiętany w drugim operandzie. Jeżeli brak drugiego operandu, ma on być w rejestrze `%eax` a wynik w postaci podwójnego słowa pamiętany w `%edx:%eax.`		
`negl`	`R/M`	`O/S/Z/A/P/C`
Negacja (uzupełnienie do 2) zawartości rejestru lub pamięci.		
`sbbl`	I/R/M, R/M	O/S/Z/A/P/C
Odejmowanie z pożyczką. Działa analogicznie jak adcl tyle że przeprowadzane jest odejmowanie. Zwykle używa się `subl`.		
`subl`	`I/R/M, R/M`	`O/S/Z/A/P/C`
Odejmuje operand określony w I/R/M od operandu określonego w R/M, wynik jest w drugim operandzie. Operuje na liczbach ze znakiem i bez znaku.		

Tab. 6-4 Instrukcje arytmetyczne operujące na liczbach całkowitych

Ogólny zapis instrukcji arytmetycznych i logicznych, zgodnie z [4] podano poniżej.

Instruction		Effect	Description
`leal`	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D + 1$	Increment
DEC	D	$D \leftarrow D - 1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \& S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	Logical right shift

Tab. 6-5 Podsumowanie instrukcji arytmetycznych

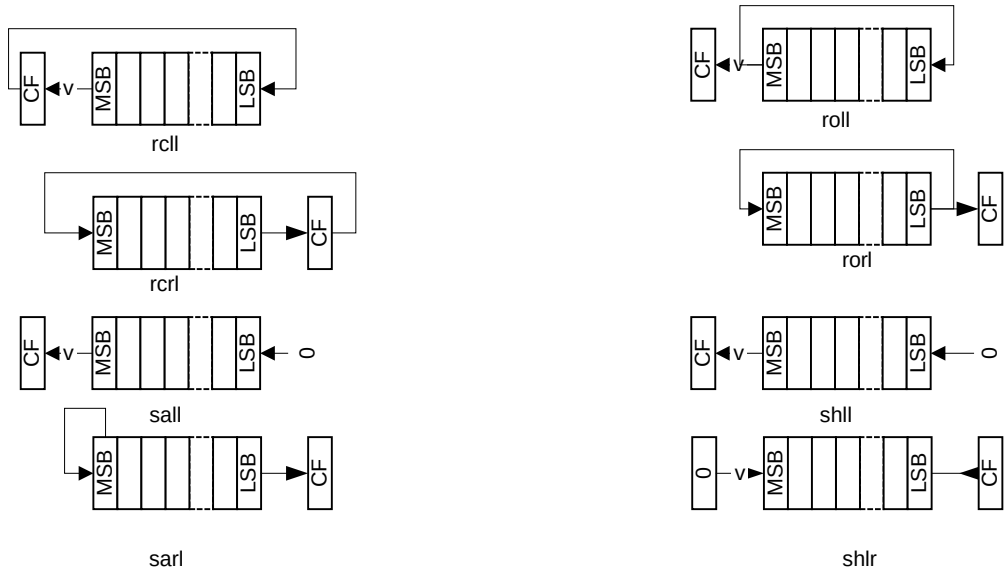
6.3.2 Instrukcje logiczne i przesunięć

Instrukcje logiczne operują na bitach a nie na bajtach. Ich podsumowanie zawiera poniższa tabela.

Instrukcja	Operandy	Flagi
`andl`	`I/R/M, R/M`	`O/S/Z/P/C`
Przeprowadzane jest logiczne AND na każdym bicie w operandach a wynik pamiętany jest w drugim.		
`notl`	`R/M`	
Przeprowadzana jest operacja negacji na każdym z bitów. Nazywane jest także dopełnieniem do jeden.		
`rcll`	`I/%cl, R/M`	`O/C`
Na drugim operandzie przeprowadzana jest rotacja w lewo, włączając flagę C. Liczba przesunięć określana jest przez pierwszy operand (wartość natychmiastowa lub rejestr `%cl`) . Ustawiana jest flaga O.		
`rcrl`	`I/%cl, R/M`	`O/C`
Tak samo jak wyżej ale rotacja w prawo.		
`roll`	`I/%cl, R/M`	`O/C`
Na drugim operandzie przeprowadzana jest rotacja w lewo. Liczba przesunięć określana jest przez pierwszy operand (wartość natychmiastowa lub rejestr `%cl`) . Ustawiana jest flaga O i C ale nie biorą udziału w rotacji.		
`rorl`	`I/%cl, R/M`	`O/C`
Tak samo jak wyżej ale rotacja w prawo.		
`sall`	`I/%cl, R/M`	`C`
Arytmetyczne przesunięcie w lewo. Bit znaku przechodzi do flagi C, na najmłodszą pozycję wchodzi 0, pozostałe bity przesuwane o 1 pozycję w lewo. Liczba przesunięć określana jest przez pierwszy operand (wartość natychmiastowa lub rejestr `%cl`) .		
`sarl`	`I/%cl, R/M`	`C`
Arytmetyczne przesunięcie w prawo. Najmniej znaczący bit przechodzi do flagi C. Bit znaku jest powielany i przesuwany w prawo. Pozostałe bity przesuwane o 1 pozycję w prawo. Liczba przesunięć określana jest przez pierwszy operand (wartość natychmiastowa lub rejestr `%cl`) .		
`shll`	`I/%cl, R/M`	`C`
Logiczne przesunięcie w lewo. Najbardziej znaczący bit przechodzi do flagi C, na najmłodszą pozycję wchodzi 0. Liczba przesunięć określana jest przez pierwszy operand (wartość natychmiastowa lub rejestr `%cl`) .		

shrl	I/%cl, R/M	C
Logiczne przesunięcie w prawo. Najmniej znaczący bit przechodzi do flagi C, na najstarszą pozycję wchodzi 0. Liczba przesunięć określana jest przez pierwszy operand (wartość natychmiastowa lub rejestr %cl) .		
testl	I/R/M, R/M	O/S/Z/A/P/C
Logiczne AND obydwu argumentów (czy w rejestrach 0 czy nie 0). Wynik jest odrzucany ale flagi ustawiane.		
xorl	I/R/M, R/M	O/S/Z/A/P/C
Wykonuje XOR operandów i umieszcza wynik w drugim. Ustawia flagi C i O na false.		

Tab. 6-6 Instrukcje logiczne



Rys. 6-1 Instrukcje przesunięć arytmetycznych, logicznych i rotacji. MSB – najstarszy bit (ang. *More Significant Bit*), LSB – najmłodszy bit (ang. *Less Significant Bit*)

6.4 Instrukcje sterujące zmieniające kolejność wykonania instrukcji programu

W sytuacji normalnej instrukcje programu wykonywane są sekwencyjnie. Jednak w pewnych sytuacjach kolejność wykonywanych instrukcji zależy od stanu procesora w szczególności danych. Zmiana sekwencji wykonywanych instrukcji może się zdarzyć w następujących sytuacjach:

- Skoki
- Wywołania funkcji
- Przerwania programowe
- Przerwania sprzętowe

6.4.1 Flagi

W większości przypadków sekwencja instrukcji zmieniana jest przez skoki warunkowe. Ich wykonanie zależy od stanu bitów w rejestrze flagowym.

Flaga		Opis
CF	Carry flag	Ostatnia operacja spowodowała nadmiar, wynik operacji nie mieści się w rejestrach. Stosowane do działań na liczbach bez znaku (unsigned).
ZF	Zero flag	Wynikiem ostatniej operacji było 0
SF	Sign flag	Ustawiona gdy wynik ostatniej operacji był ujemny
OF	Overflow flag	W wyniku ostatniej operacji na argumentach uzupełnienie do 2 nastapił nadmiar (dodatni lub ujemny)

Tab. 6-7 Flagi procesora używane w skokach warunkowych

Do ustawiania flag służą instrukcje `cmp` i `test` pokazane poniżej.

Instruction		Based on	Description
CMP	S_2, S_1	$S_1 - S_2$	Compare
cmpb		Compare byte	
cmpw		Compare word	
cmpl		Compare double word	
TEST	S_2, S_1	S_1 & S_2	Test
testb		Test byte	
testw		Test word	
testl		Test double word	

Tab. 6-8 Operacje porównania i testowania

Operacja `cmp` służy do porównywania arytmetycznego

`cmpl op1, op2`	I/R/M, R/M	O/S/Z/A/P/C
Porównaj dwie liczby całkowite. Odbywa się poprzez odjęcie pierwszego operandu od drugiego czyli wykonywana jest operacja `op2 – op1`. Wynik jest odrzucany, ale flagi są ustawiane. Instrukcja zwykle używana przed skokiem warunkowym.		

Tab. 6-9 Operacja porównania cmp

Operacja `test op1, op2` wykonuje bitowy AND operandów `op1 i op2`.

Powyższe instrukcje nie zmieniają stanu rejestrów a jedynie ustawiają flagi. Flagi są zmieniane także przez instrukcje arytmetyczne. Instrukcje operujące na adresach nie zmieniają flag. Zamiast czytać flagi stosuje się następujące sposoby postępowania:

- Ustawia się stan określonego bajtu na podstawie ustawienia flag.
- Wykonuje się skok warunkowy
- Wykonuje się warunkowego przesłania danych.

Ustawienia bajtu w zależności od flag dokonać można za pomocą instrukcji set opisanej poniżej.

Instruction		Synonym	Effect	Set condition
sete	D	setz	$D \leftarrow$ ZF	Equal / zero
setne	D	setnz	$D \leftarrow$ ~ZF	Not equal / not zero
sets	D		$D \leftarrow$ SF	Negative
setns	D		$D \leftarrow$ ~SF	Nonnegative
setg	D	setnle	$D \leftarrow$ ~(SF ^ OF) & ~ZF	Greater (signed >)
setge	D	setnl	$D \leftarrow$ ~(SF ^ OF)	Greater or equal (signed >=)
setl	D	setnge	$D \leftarrow$ SF ^ OF	Less (signed <)
setle	D	setng	$D \leftarrow$ (SF ^ OF) | ZF	Less or equal (signed <=)
seta	D	setnbe	$D \leftarrow$ ~CF & ~ZF	Above (unsigned >)
setae	D	setnb	$D \leftarrow$ ~CF	Above or equal (unsigned >=)
setb	D	setnae	$D \leftarrow$ CF	Below (unsigned <)
setbe	D	setna	$D \leftarrow$ CF | ZF	Below or equal (unsigned <=)

Tab. 6-10 Instrukcje ustawienia bajtu w zależności od flag

W assemblerze wyniki porównania liczb zależą od tego czy są ze znakiem czy bez znaku. Gdy rozważamy liczby ze znakiem `a` i `b` aby je porównać wykonujemy instrukcję `cmp b,a` która wykonuje operację odejmowania: `t = a - b`. Gdy `a - b < 0` ustawiona będzie CF. Gdy `a = b` ustawiona będzie ZF.

6.4.2 Instrukcje skoku

Instrukcje skoku powodują przejście do innej niż następna instrukcji. Określona jest ona zwykle przez etykietę.

`jcc`	`Adres przeznaczenia`	O/S/Z/A/C
Instrukcja skoku warunkowego, `cc` jest kodem warunku. Skok jest wykonywany gdy warunek ma wartość logiczną *true*. Warunek jest ustawiany zazwyczaj w poprzedniej instrukcji, zwykle jest to porównanie. Gdy warunek ma wartość *false*, wykonywana jest następna instrukcja. Warunki `cc` są dane poniżej:		
`[n]a[e]`	większy niż (ang. *above*), gdy na początku wystąpi `n` jest to negacja (ang. *not*), gdy na końcu wystąpi `e` (ang. *equal*) większy lub równy	`unsigned`
`[n]b[e]`	mniejszy (lub równy), (ang. *bellow*)	`unsigned`
`[n]e`	równy (ang. *equal*)	
`[n]z`	zero	
`[n]g[e]`	Większy (ang. *greater*), (lub równy)	`signed`
`[n]l[e]`	Mniejszy (ang. *less*), (lub równy)	`signed`
`[n]c`	Ustawiona flaga przeniesienia C – carry	
`[n]o`	Ustawiona flaga nadmiaru O - overflow	
`[n]p`	Ustawiona flaga parzystości P - parity	
`[n]s`	Ustawiona flaga znaku S - sign	
`ecxz`	Rejestr `%ecx` zawiera 0	
`jmp`	`Adres przeznaczenia`	
Skok bezwarunkowy do adresu przeznaczenia, czyli adres przeznaczenia kopiowany do rejestru `%eip`. Gdy adres przeznaczenia jest rejestrem poprzedzonym gwiazdką (np. `jmp *%eax)`następuje skok do adresu zawartego w rejestrze.		

Tab. 6-11 Instrukcje zmiany kolejności wykonywania kodu

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	ZF	Equal / zero
`jne`	*Label*	`jnz`	~ZF	Not equal / not zero
`js`	*Label*		SF	Negative
`jns`	*Label*		~SF	Nonnegative
`jg`	*Label*	`jnle`	~(SF ^ OF) & ~ZF	Greater (signed >)
`jge`	*Label*	`jnl`	~(SF ^ OF)	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	SF ^ OF	Less (signed <)
`jle`	*Label*	`jng`	(SF ^ OF) | ZF	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	~CF & ~ZF	Above (unsigned >)
`jae`	*Label*	`jnb`	~CF	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	CF	Below (unsigned <)
`jbe`	*Label*	`jna`	CF | ZF	Below or equal (unsigned <=)

Tab. 6-12 Instrukcje skoku

Skoki warunkowe mogą być bezpośrednie lub pośrednie.

- Skoki bezpośrednie – adres skoku jest zakodowany w instrukcji, zwykle jest to etykieta, np. `.L1`
- Skoki pośrednie – adres skoku zawarty jest w rejestrze lub pamięci. Wtedy adres poprzedzony jest *.

`jmp *%EAX` – skok do adresu który zawarty jest w rejestrze EAX.
`jmp *(%EAX)` – skok do adresu który zawarty jest w komórce której adres jest w rejestrze EAX.

Skoki mogą być też relatywne do zawartości licznika rozkazów PC (PC-relative). Działanie skoków warunkowych wyjaśnia poniższy przykład.

```
if(a == b) {
/* Gałąź True  */
} else {
/* Gałąź False */
}
/* Dalej */
```

Przykład 6-3 Porównanie a i b w języku C

```
# Wpisz do rejestrow a i b by je porownac
movl a, %eax
movl b, %ebx
# Porownaj – ustaw flagi
cmpl %eax, %ebx
# Gdy true idz do galaz_true
je galaz_true
galaz_false: #ta etykieta nie jest konieczna
# tylko dla dokumentacji
# Kod galezi false jest tutaj
# Skok do dalej
jmp dalej
galaz_true:
# Kod galezi true jest tutaj
dalej:
# Tutaj spotykaja sie obydwie galezie
```

Przykład 6-4 Porównanie `a = b` w assemblerze

Poniżej podany został inny przykład pochodzący z [17].

```
if ( EAX > 1 ) {
   // kod if
}  else {
  //kod else
}
```

Przykład 6-5 Porównanie EAX > 1 w C

```
cmpl $1, %eax     # porównaj EAX z 1
jg ifcode         # skok do  ifcode gdy EAX > 1
elsecode:
... # else-code
jmp end
ifcode:
... # IF-code
end:
```

Przykład 6-6 Porównanie EAX > 1 w assemblerze

6.4.3 Pętle

W języku C (jak i w innych) występują pętle postaci `do .. while(...), while(...) i for`.

6.4.3.1 Pętla do ... while(...)

Pętla `do{ ...} while(...)` ma postać:

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

Instrukcje wykonywane są tak długo jak `warunek` będzie miał wartość nie zerową. Może być ona przepisana w następujący sposób:

```
loop:
  instrukcje;
  t = warunek
  if(t) goto loop;
```

Przykład petli `do while` dany jest poniżej.

```
do {
  //loop code
} while( EAX > 1 );
```

Przykład 6-7 Kod pętli do while w języku C

```
loop:
...               # loop code
cmpl $1, %eax     # repeat the loop
jg loop           # if EAX > 1
```

Przykład 6-8 Kod petli `do ... while` w assemblerze

Budowę petli `do … while(…)` można przeanalizować na przykładzie funkcji obliczania silni liczby n czyli n!.

6.4.3.2 Pętla while(...)

Pętla ta ma postać:

```
while(warunek)
  instrukcje;
```

Gdy warunek ma wartość różną od zera blok instrukcji będzie wykonany. Pętlę tę można zapisać inaczej.

```
if(!warunek)
  goto koniec;
petla:
  instrukcje;
  if(warunek) goto petla;
koniec:
```

```
while(1 < EAX)
{
  //loop code
}
```

Przykład 6-9 Kod petli `while` w języku C

```
loop:
cmpl $1, %eax     # if EAX <= 1 jump to
jle end           # the end of the loop
...               # loop code
jmp loop          # repeat the loop
end:
```

Przykład 6-10 Kod pętli `while` w assemblerze

6.4.3.3 Pętla for

Pętla `for` ma postać:

```
for(inicjalizacja; warunek; aktualizacja) instrukcje;
```

Pętlę tę można przepisać za pomocą skoków warunkowych i instrukcji `goto`.

```
inicjalizacja;
if(!warunek) goto koniec;
petla:
  instrukcje;
  aktualizacja;
if(warunek) goto petla;
koniec:
```

Dla przykładu rozważmy petlę for:

```
for ( EAX = 0; EAX < 100; EAX++ )
{
      //loop code
}
```

Jest ona równoważna konstrukcji:

```
EAX = 0;
while ( EAX < 100 )
{
  //loop code
  EAX++;
}
```

Przykład 6-11 Kod petli `for` w C

```
movl $0, %eax
loop:
cmpl %100, %eax
ja end
 // loop code
incl %eax
jmp loop
end:
```

Przykład 6-12 Kod petli `for` w assemblerze

Język assemblera x86 posiada wsparcie dla wykonywania pętli. Używany jest instrukcja `loop` z etykietą i rejestr `%ecx` jako licznik. Gdy wartość licznika `%ecx` jest różna od zera następuje skok do etykiety. Po każdym wykonaniu wartość licznika zmienjsza się o 1. Gdy licznik osiągnie zero wykonywana jest następna instrukcja po `loop`.

```
#Wpisz do %ecx wartość licznika
etykieta:
#Zawartość pętli
loop etykieta
```

Przykład 6-13 Działanie pętli loop

Przykład wykonania instrukcji petli 100 razy pokazano poniżej.

```
for(i=0; i < 100; i++)
{
/* Zawartosc petli */
}
```

Przykład 6-14 Pętla 100x w języku C

```
# inicjalizacja petli – 100 do %ecx
movl $100, %ecx
loop_begin:
#
# Zawartosc petli
#
# Zmniejszenie zawartosci %ecx i skok do loop_begin gdy nie zero
loop loop_begin
# Kontynuacja programu
```

Przykład 6-15 Pętla 100x w języku assemblera

Poniżej dany jest przykład obliczania funkcji `n!`.

```
int fact_for_goto(int n)
{
 int i = 2;
 int result = 1;
 if (!(i <= n)) goto done;
 loop:
 result = result * i;
 i++;
 if (i <= n) goto loop;
 done:
 return result;
}
```

Przykład 6-16 Obliczanie funkcji `n!` w języku C

6.4.4 Wywoływanie funkcji

Powtarzalne fragmenty kodu umieszcza się w funkcjach. Funkcja zazwyczaj posiada argumenty. Funkcja operuje na swych argumentach i ew. zmiennych globalnych i kończy się rozkazem `ret`. Aby wywołać fukcję należy:

- Skopiować na stos jej parametry w odwrotnej kolejności
- Wykonać instrukcję `call adres_funkcji`

Instrukcja `call adres_funkcji` powoduje skopiowanie na stos adresu instrukcji następującej bezpośrednio po `call` (jest to adres powrotu) a następnie załadowanie do rejestru instrukcji `%eip` adresu nowej funkcji (`adres_funkcji`)bedącej argumentem wywołania. Spowoduje to rozpoczęcie wykonywania kodu nowej funkcji. Ostatnia instrukcja funkcji czyli `ret` powoduje skopiowanie do licznika rozkazów ze stosu adresu powrotu i wznowienie wykonywania kodu czyli instrukcji nastepnej po `call`.

Instrukcja	Operandy	Flagi
`call`	`Adres docelowy`	`O/S/Z/A/C`
Składuje na stosie adres następnej instrukcji wskazywanej przez licznik rozkazów `%eip` i wykonuje skok do instrukcji wskazanej w operandzie. Do określenia adresu skoku można też użyć poprzedzonego gwiazdką nazwy rejestru. Np. `call *%eax` spowoduje skok do adresu określonego w `%eax.`		
`int`	`I`	`O/S/Z/A/C`
Powoduje wykonanie przerwania programowego o numerze I. Instrukcja używana do komunikacji z jądrem i jako interfejs wywołań systemowych		
`jcc`	`Adres przeznaczenia`	`O/S/Z/A/C`
`ret`		`O/S/Z/A/C`
Powrót z funkcji. Zdejmuje ze stosu wartość i przesyła ja do licznika rozkazów `%eip`.		

Dla przykładu rozważymy wywołanie fukcji:

```
printf("The number is %d", 88);
```

W assemblerze wywołanie wygląda jak poniżej.

```
.section .data
text_string:
.ascii "The number is %d\0"
.section .text
pushl $88
pushl $text_string
call printf
popl %eax
popl %eax #%eax jest rejestrem nie wykorzystanym
```

Przykład 6-17 Wywołanie sekwencji `printf("The number is %d", 88);`

6.5 Inne instrukcje

Instrukcja	Operandy	Flagi
`in`	`port lub %dx`	
Przesłanie zawartości portu do rejestru AL, AX, EAX. Gdy użyte `%dx` adres portu w rejestrze `%dx` i zakres portów z przedziału 0-65535. Gdy argument natychmiastowy port z zakresu 0-255.		
`out`	`al/ax/eax/, I/%dx`	
Przesłanie zawartości rejestru `%al, %ax, %eax` do portu określonego jako argument natychmiastowy lub w rejestrze `%dx`		

6.6 Przykład programu – szukanie maksimum w tablicy

Prosty program szukania maksymalnego elementu w tablicy podany został poniżej.

```
# Program szukania maksimum w tablicy
// kompilacja: gcc maksimum_c.c -o maksimum_c
// uruchomienie; ./maksimum_c ; echo $?
#include <stdlib.h>

int tab[14] = {3,67,34,222,45,75,54,34,44,33,22,11,66,0};

int main(void) {
    int edi;  // Indeks tablicy
    int ebx;  // Aktualne maksimum
    int eax;  // Wartosc aktualna
    edi = 0;
    eax = tab[0];
    ebx = eax;
    start_loop:
    // Zero jest znacznikiem konca tablicy
    if(eax == 0) goto loop_exit;
    edi++;
    eax = tab[edi]; // w eax biezacy element
    // Gdy biezacy element wiekszy od ebx, do ebx wstawiamy eax
    if(eax > ebx) ebx = eax;
    goto start_loop;
    loop_exit:
    // Kod powrotu ebx
    exit(ebx);
}
```

Przykład 6-1 Szukanie maksimum w tablicy, program `maksimum_c.c`

Obecnie ten sam algorytm zostanie zapisany w języku assemblera. Prezentuje on dostęp do danych poprzez indeksowanie. Przykład pochodzi z [3]. W sekcji `.data` umieszczona jest tablica zawierająca liczby 4 bajtowe (`long`). Można się do niej odwołać poprzez etykietę `tablica:` która podaje przesunięcie względem początku segmentu `.data`. Licznik tablicy znajduje się w rejestrze `%ecx` który zwiększany jest w każdym kroku algorytmu.

```
#znajdowanie maksimum w tablicy tablica
# rejestr %edi  - indeks tablicy
# rejestr %ebx  - aktualnie najwiekszy element
# rejestr %eax  - biezacy element
.section .data
 tablica: #To są dane w tablicy zawierającej liczby long
.long 3,67,34,224,45,75,54,34,44,33,22,11,66,0
.section .text
.globl _start
_start:
movl $0, %edi # przeslij 0 do rejestru indeksowego
movl tablica(,%edi,4), %eax # przeslij do eax kolejna liczbe z tablicy
movl %eax, %ebx  # pierwszy element jest na razie najwiekszy
start_loop: # start petli
cmpl $0, %eax # sprawdz czy koniec tablicy (jest tam 0)
je loop_exit
incl %edi # zwieksz licznik
movl tablica(,%edi,4), %eax
cmpl %ebx, %eax # porownaj czy %eax  wiekszy od maksimum w %ebx
jle start_loop  # skocz do poczatku petli gdyz %eax jest mniejszy od max
movl %eax, %ebx # w %eax jest aktualne maksimum
jmp start_loop  # skocz na oczatek petli
loop_exit:
# %ebx zawiera kod powrotu funkcji exit i jest to maksimum
movl $1, %eax #1 jest kodem wywolania exit()
int $0x80
```

Przykład 6-18 Znajdowanie maksimum w tablicy – program `maximum.s`

Kompilacja programu:

```
as maximum.s –o maximum.o
ld maximum.o –o maximum
```

Wykonanie:

```
./maximum ; echo $?
```

Jako wynik będzie wyświetlona największa liczba czyli 224.

6.7 Zadania

6.7.1 Instrukcje przesłań - uzupełnienia

Dla każdej z poniższych linii uzupełnij instrukcję mov o właściwy sufiks (b, w l) właściwy dla operanda. Na przykład mov ma być przepisana jako `movb, movw movl`.

```
1    mov   %eax, (%esp)
2    mov   (%eax), %dx
3    mov   $0xFF, %bl
4    mov   (%esp,%edx,4), %dh
5    push  $0xFF
6    mov   %dx, (%eax)
7    pop   %edi
```

6.7.2 Tryby adresowania

Załóżmy że w rejestrach i komórkach pamięci mamy następujące wartości:

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Wypełnij poniższą tabelke pokazując wartości dla danych operandów.

Operand	Value
%eax	_____
0x104	_____
$0x108	_____
(%eax)	_____
4(%eax)	_____
9(%eax,%edx)	_____
260(%ecx,%edx)	_____
0xFC(,%ecx,4)	_____
(%eax,%edx,4)	_____

6.7.3 Instrukcje przesłań - błędy

Każda z poniższych instrukcji zawiera błąd. Wyjaśnij jaka jest jego przyczyna.

```
1    movb $0xF, (%bl)
2    movl %ax, (%esp)
3    movw (%eax),4(%esp)
4    movb %ah,%sh
5    movl %eax,$0x123
6    movl %eax,%dx
7    movb %si, 8(%ebp)
```

6.7.4 Instrukcja leal

Załóżmy że rejestr %eax zawiera wartość x a rejestr %ecx zawiera wartość y. Wypełnij poniższą tabelkę wpisując formułę pokazującą co zawiera rejestr %edx.

Instruction	Result
leal 6(%eax), %edx	_____
leal (%eax,%ecx), %edx	_____
leal (%eax,%ecx,4), %edx	_____
leal 7(%eax,%eax,8), %edx	_____
leal 0xA(,%ecx,4), %edx	_____
leal 9(%eax,%ecx,2), %edx	_____

6.7.5 Instrukcje arytmetyczne

Załóżmy że wartości zawarte w rejestrach i komórkach pamięci sa następujące:

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Wypełnij poniższą tabelę wpisując jaka będzie zawartość rejestrów i pamięci.

Instruction	Destination	Value
`addl %ecx,(%eax)`	_____	_____
`subl %edx,4(%eax)`	_____	_____
`imull $16,(%eax,%edx,4)`	_____	_____
`incl 8(%eax)`	_____	_____
`decl %ecx`	_____	_____
`subl %edx,%eax`	_____	_____

6.7.6 Przesunięcia

Załóżmy że chcemy napisac w assemblerze następującą funkcję:

```
int shift_left2_rightn(int x, int n)
{
    x <<= 2;
    x >>= n;
    return x;
}
```

Poniższy kod realizuje tę funkcję przesunięć i ma umieścić wynik końcowy x w rejestrze %eax. Dwie kluczowe instrukcje pominięto – należy je uzupełnić. Parametry x i n są umieszczone relatywnie względem %ebp z przesunięciem 8 i 12.

```
1     movl    8(%ebp), %eax    Get x
2     _________                x <<= 2
3     movl    12(%ebp), %ecx   Get n
4     _________                x >>= n
```

6.7.7 Porównania logiczne

Mamy następujący kod funkcji w C która porównuje argument a z 0 dla różnych typów danych. Operacja porównania TEST może być różna i może być zdefiniowana w #define TEST.

```
int test(data_t a) {
    return a TEST 0;
}
```

Dla poniższej tabelki podaj właściwy typ danych data_t i jaka jest operacja TEST.

A.
```
testl   %eax, %eax
setne   %al
```

B.
```
testw   %ax, %ax
sete    %al
```

C.
```
testb   %al, %al
setg    %al
```

D.
```
testw   %ax, %ax
seta    %al
```

7. Struktura programu

7.1 Program źródłowy w assemblerze

Program w języku assemblera (GNU assembler, GAS) jest plikiem tekstowym składającym się z elementów oddzielonych spacjami co opisano w [7]. Elementy oddzielone znakiem nowej linii lub średnikiem ; tworzą wyrażenia. Elementami pliku źródłowego są:

1. Dyrektywy dla kompilatora – określają sposób kompilacji
2. Symbole - są składnikami pozostałych elementów np. wyrażeń
3. Etykiety – określają symbolicznie adresy
4. Wyrażenia - określają „co robić”
5. Stałe - symboliczne określenie (powtarzających się) danych
6. Komentarze - wyjaśnienia, nie są przetwarzane

7.2 Symbole

Symbolem jest ciąg znaków zawierający tylko litery, cyfry i znaki **_ . $.** Symbol nie może zaczynać się od cyfry. Małe i duże litery są rozróżniane. Symbole mogą być także zamknięte w znakach cudzysłowu, wtedy wszystkie znaki są dozwolone.

7.3 Wyrażenia

Wyrażenie (ang. *statement*) kończy się znakiem nowej linii `\n` bądź innym separatorem linii. Wyrażenie opcjonalnie rozpoczyna się od etykiety po której następuje kluczowy symbol od którego zależy dalsza składnia.

- Jeżeli symbol zaczyna się od kropki jest dyrektywa dla kompilatora.
- Jeżeli symbol zaczyna się od litery jest to instrukcja assemblera.

Etykieta musi kończyć się znakiem dwukropka :.

Przykłady wyrażeń dane są poniżej.

```
etykieta1:     .dyrektywa parametry
etykieta2:     instrukcja   operand_1, operand_2, …
```

7.4 Komentarze

W kompilatorze GAS są dwa rodzaje komentarzy. Podano je poniżej:

```
/* To jest komentarz – rodzaj 1  */
#  To jest komentarz - rodzaj 2
```

Komentarze rodzaju 1 nie mogą być zagnieżdżane.

7.5 Stałe w kompilatorze GAS

W kodzie assemblera występują stałe które określają wartości początkowe innych symboli jak zmiennych, adresów i obszarów pamięci. Wyróżnia się tutaj:

- Znaki (ang. *character*)
- Łańcuchy (ang. *String*)
- Liczby (ang. *Number*)

7.5.1 Zapis znaków

Znaki zapisuje się podając odpowiadającą mu literę poprzedzoną apostrofem. Na przykład:
`mychar: .byte 'c` definiuje zmienną `mychar` jako znak `c`. Nie każdy znak ma drukowalny odpowiednik. W takim przypadku stosuje się konwencję ze znakiem odwrotnego ukośnika (ang. *backslash*) po którym następuje specyfikacja znaku.

\ddd	Kod znaku określono ósemkowo np. \012
\xDD	Kod znaku określono szesnastkowo np. \x0A
\n	Znak nowej linii szesnastkowo \0xA ósemkowo \012
\0	Koniec łańcucha, NULL szesnastkowo \x00 ósemkowo \000
\t	Znak tabulacji szesnastkowo \x09 ósemkowo \011
\f	Przejście do następnego wiersza (FF form feed) szesnastkowo \x0C ósemkowo \014
\r	Powrót karetki (CR Carriage Return) szesnastkowo \x0D ósemkowo \015

Tab. 7-1 Kody ASCII niektórych znaków specjalnych

7.5.2 Zapis łańcuchów

Łańcuchy zapisuje się podając ich zawartość w cudzysłowie. Gdy w łańcuchu występują znaki specjalne wpisuje się je zgodnie z powyższą tabelą. Przykład definicji łańcucha o nazwie `msg` podano poniżej.

```
.data
msg: .ascii "Hello, world!\n"
```

7.5.3 Zapis liczb

Liczby zapisywać można w kodzie szesnastkowym, dziesiętnym, ósemkowym lub binarnym.

0d[-]c...c	Zapis dziesiętny	c=0,1,2,...,9	0d221
0x[-]h...h	Zapis szesnastkowy	h=0,1,...,9,a,b,c,d,e,f	0x1c
0q...q	Zapis ósemkowy	q=0,1,...,7	0777
0bt...t	Zapis binarny	t=0,1	0b11001101
0f[-]i.uE[-]e	Zapis zmiennoprzecinkowy	i = część całkowita, u = część ułamkowa e = wykładnik	0.125 -3.14E-4

Tab. 7-2 Zapis liczb w różnych formatach

7.6 Sekcje

Sekcja jest ciągłym obszarem adresów traktowanym w jednolity sposób. Program może się składać z wielu plików kompilowanych do tak zwanych plików obiektowych. Podczas łączenia linker czyta pliki obiektowe i tworzy z nich program wykonywalny. Gdy assembler tworzy plik obiektowy, zakłada że jego adresy zaczynają się od zera. Jednak program wynikowy może składać się z wielu plików obiektowych. Linker `ld` łączy bloki kodu i danych z plików obiektowych w program wynikowy, przesuwając bloki pamięci w taki sposób by się nie nakładały. Takie bloki nazywane są sekcjami a proces relokacją. Plik wykonywalny składa się z co najmniej trzech sekcji:

`.text` – sekcja kodu
`.data` - sekcja danych zainicjowanych
`.bss` - sekcja danych nie zainicjowanych

Niektóre sekcje mogą być puste. W pliku obiektowym segment `.text` zaczyna się od adresu 0, po nim następuje sekcja `.data` o po niej `.bss`.

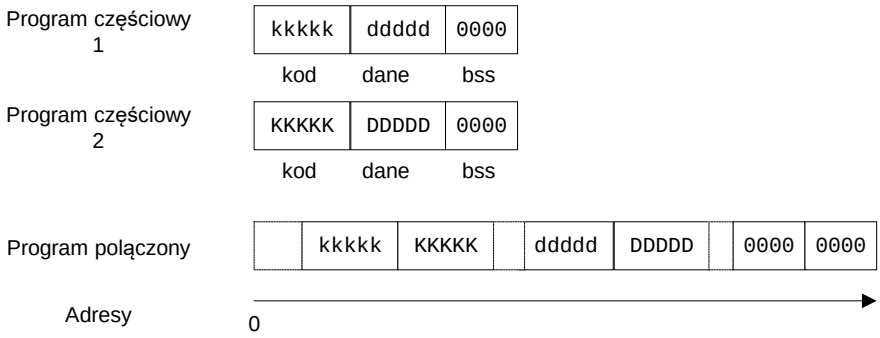
Linker ma poskładać pliki obiektowe tak by adresy się nie nakładały, w każdym pliku obiektowym dla każdego obszaru pamięci musi być informacja:

- Gdzie zaczyna się obszar pamięci
- Jak jest duży
- Do jakiego segmentu należy

W rzeczywistości każdy adres zapisywany jest w postaci:

```
(sekcja) + (przesunięcie względem początku sekcji)
```

Poniższy rysunek ilustruje sposób łączenia plików obiektowych „Program częściowy 1” i „Program częściowy 2” w program wynikowy



Rys. 7-1 Łączenie dwóch plików obiektowych

7.7 Dyrektywy kompilatora GAS

Dyrektywy dla kompilatora sterują procesem kompilacji a także określają segmenty pamięci i definiują dane. Dyrektywy kompilatora zaczynają się kropką. Ich pełny zestaw jest obszerny i pokazany jest w [8]. Tutaj opisane zostaną tylko najważniejsze podane w [9].

7.7.1 Dyrektywa .section

Jak już wspomniano program zbudowany jest z ciągłych obszarów pamięci nazywanych sekcjami. Dyrektywa określa postać tych sekcji. Najczęściej spotykane sekcje to:

- `.data` – sekcja danych zainicjowanych
- `.bss` – sekcja danych nie zainicjowanych
- `.tekst` - sekcja kodu, zawiera instrukcje programu

Ogólna postać dyrektywy opisana w [8] jest następująca:

`.section` *`name`* `[, "`*`flags`*`"[, @`*`type`*`[,`*`flag_specific_arguments`*`]]]`

7.7.2 Dyrektywa .data / .section .data

Sekcja danych, zawiera zainicjowane zmienne i stałe. Oznaczona jest jako:

`.section .data` lub `.data`

7.7.3 Dyrektywa .bss / .section .bss

Sekcji danych nie zainicjowanych Oznaczona jest jako:

`.section .bss` lub `.bss`

Jest to informacja dla kompilatora że trzeba zarezerwować pamięć na dane które będą ustalane potem

7.7.4 Dyrektywa ./text / .section .text

Sekcji kodu, zawiera instrukcje programu

```
.section .text
```

7.7.5 Dyrektywa .type

Dyrektywa `.type` określa typ symbolu. Ogólna postać dyrektywy jest następująca:

```
.type nazwa_typu, opis_typu
```

W dalszej części będzie używana dyrektywa określająca funkcje: np. funkcję o nazwie `power`

```
.type power, @function
```

7.7.6 Dyrektywa .globl

Dyrektywa `.globl` informuje kompilator że występujący po niej symbol ma być dostępny dla linkera `ld` poza plikiem w którym występuje. Przykład:

```
.globl _start
```

Dyrektywa informuje że etykieta `_start` ma być dostępna na zewnątrz tego pliku. Etykieta `_start` określa punkt wejścia do programu (pierwsza instrukcja). Inny przykład:

```
.globl _main
```

7.7.7 Dyrektywa .include

Dyrektywa `.include nazwa_pliku` nakazuje włączyć dany plik do pliku źródłowego

7.7.8 Dyrektywa .equ symbol, wyrażenie

Dyrektywa `.equ` służy do nadania wartości `wyrażenie` symbolowi `symbol,` służy więc definiowaniu stałych. Ma ona postać:

```
.equ symbol, wyrażenie
```

Dyrektywa nie powoduje rezerwacji pamięci. W poniższym przykładzie nadajemy wartości symbolom `SYS_READ, SYS_OPEN, SYS_WRITE`

```
.section .data
# Numery wywołań systemowych
.equ SYS_READ, 3
.equ SYS_OPEN, 5
.equ SYS_WRITE, 4
```

Przykład 7-1 Różne dyrektywy `.equ`

7.7.9 Dyrektywa `.lcomm symbol , length`

Dyrektywa `.lcomm` ma postać:

```
.lcomm symbol , length
```

Powoduje ona zarezerwowanie w segmencie `.bss` dla symbolu `symbol length` bajtów. Odpowiada tablicy w języku C. Przykład dany jest poniżej.

```
.section .bss
.lcomm base, 100
```

Dyrektywa powoduje zarezerwowanie 100 bajtów pamięci. Etkieta początku tego obszaru to `base`. W języku C byłoby to: `int x[25]; // Dlaczego 25 a nie 100`

7.7.10 Dyrektywa .ascii

Dyrektywa `.ascii "string"` rezerwuje pamięć na napis `"string"`. Ilość bajtów zależy od długości łańcucha. Dla przykładu poniższy fragment programu definiuje w pamięci pod lokacją nazwisko1: `napis "Kowalski".`

```
nazwisko:
.ascii "Kowalski\0"
```

W języku C byłoby to: `char nazwisko[] = "Kowalski";`

7.7.11 Dyrektywa .byte

Dyrektywa `.byte val1,val2,val3, ...` służy do nadawania wartości `val1,val2,val3` kolejnym bajtom. Dla przykładu poniżej zdefiniowana jest bajtowa zmienna o nazwie mychar i wartości początkowej `0x01`.

```
mychar:
.byte 'a
```

W języku C byłoby to: `char mychar = 'a'`

7.7.12 Dyrektywa .long

Dyrektywa `.long val1,val2,val3, ...` służy do nadawania wartości `val1,val2,val3` kolejnym obszarom 4 bajtowym. Wartość pojedynczej komórki jest z przedziału od 0 do 0 do 4294967295.

```
.section .data
bufor: #Dalej zdefiniowana jest zawartosc bufora
.long 3,67,34,222,45,75,54,34,44,33,22,11,66,0
```

W języku C byłoby to:

```
int bufor[] = {3,67,34,222,45,75,54,34,44,33,22,11,66,0}
```

7.7.13 Dyrektywa .float

Dyrektywa `.float val1,val2,val3, ...` służy do nadawania wartości `val1,val2,val3` kolejnym zmiennym 4 bajtowym w formacie `float`.

```
.section .data
pi:
.float 3.14
```

W języku C byłoby to:

```
float pi = 3.14
```

7.7.14 Dyrektywa .space

Dyrektywa `.space size, fill` służy do wypełnienia obszaru pamięci o rozmiarze `size` bajtami o zawartości `fill`. Gdy pominiemy przecinek i parametr obszar będzie wypełniony zerami.

7.7.15 Dyrektywa .rept

Dyrektywa `./rept count` powoduje powtórzenie `count` razy wyrażenia zawartego pomiędzy `./rept` a `./endr.` Przykładowo:

```
.rept 3
.long 0
.endr
```

Równoważne jest

```
.long   0
.long   0
.long   0
```

Dyrektywa	Opis	Przykład
`.data` `.section .data`	Sekcja danych zainicjowanych	`.data`
`.bss` `.section .bss`	Sekcja danych nie zainicjowanych	`.bss`
`.text` `.section .text`	Sekcja kodu	`.text`
`.type nazwa_typu, opis_typu`	Określenie typu symbolu	`.type power, @function`
`.globl`	Występujące dalej symbole są globalne	`.globl _main`
`.include nazwa_pliku`	Włącz plik zewnętrzny	`.include common.h`
`.lcomm symbol , length`	Rezerwacja obszaru o nazwie `symbol` długości `length`	`.lcomm base, 100`
`.ascii ”string”`	Rezerwacja miejsca na string znaków ASCII	`nazwisko:` `.ascii "Kowalski\0"`
`.byte val1,val2,val3,…`	Nadaje kolejnym bajtom wartości `val1,val2,val3,`	`mychar:` `.byte 0x01`
`.long val1,val2,val3,…`	Nadaje kolejnym obszarom 4 bajtowym wartości `val1,val2,val3,…`	`bufor:` `.long 3,67,34,222`
`.float val1,val2,val3,…`	Nadaje kolejnym liczbom float 4 bajtowym wartości `val1,val2,val3,…`	`Pi:` `.float 3.14`
`.space size, fill`	Wypełnia obszar długości size znakami fill	`Bufor:` `.space 200 \0x00`

`.rept count`
`...`
`.endr`
	Powtarza sekwencję pomiędzy `./rept` a `./endr count` razy	`.rept 3`
		`.long 0`
		`.endr`

Tab 7-1 Ważniejsze dyrektywy kompilatora `as`

7.8 Obliczanie długości danych

Język assemblera pozwala na obliczanie długości obszaru danych $dane_1$, $dane_2$, ...,$dane_n$ i przypisanie jej do pewnego symbolu `dlugosc_danych`. Pokazane to zostało na poniższym schemacie. Symbol `etykieta` jest w istocie adresem obszaru $dane_1$. W wyrażeniu `dlugosc_danych = .- etykieta` kropka oznacza adres bieżący (tuż po obszarze $dane_n$), a różnica tej wielkości – początku obszaru daje właśnie długość obszaru danych.

```
etykieta:
dane1
dane2
...
danen
dlugosc_danych = .- etykieta
```

Przykład obliczenia długości łańcucha `text` dany jest poniżej.

```
.text
msg: .ascii "Hello, world!\n"
msgLen = . - msg
```

7.9 Przypisanie wartości symbolom

W języku assemblera symbolom można przypisać wartość używając operatora =. Na przykład symbolowi `SYS_CALL = 0x80` nadajemy wartość `0x80`. Symbolu możemy użyć w innym miejscu, np. zamiast `int $0x80` możemy użyć `int $SYS_CAL.`

```
SYS_CALL = 0x80
int $SYS_CAL
```

W języku C odpowiada temu:

```
#define SYS_CALL  0x80
```

7.10 Uruchamianie kompilatora i programu łączącego

Assembler wywołuje się poleceniem `as`. Opis opcji można uzyskać pisząc:

```
as –help
```

Ogólna postać wywołania to:

```
as [opcje] nazwa_pliku
```

Typowe przypadki użycia dane są dalej. Kompilacja pliku źródłowego `plik.s` do pliku obiektowego `plik.o`.

```
as plik.s --32 –o plik.o
```

Kompilacja pliku źródłowego `plik.s` do pliku obiektowego `plik.o` z informacją dla debuggera.

```
as plik.s --32 –g –o plik.o
```

Program łączący uruchamia się poleceniem `ld`. W poniższym przykładzie łączymy plik obiektowy `plik.o` w kod wykonywalny `plik`.

```
ld plik.o –m elf_i386 –o plik
```

Kod prostego programu który wykonuje tylko wywołanie systemowe `exit(0)` podano poniżej.

```
# Program ktory wykonuje wywolanie systemowe exit(0)
# Zwraca kod powrotu ktory może być sprawdzony poleceniem
# echo $?
# ZMIENNE:
# %eax zawiera numer wywolanie systemowego
# %ebx zawiera kod powrotu
#
.section .data
.section .text
.globl _start
_start:
movl $1, %eax # Numer wywolania syst w EAX
# Liczba 1 to numer wywolania systemowego – zakoncz proces
movl $0, %ebx # Kod powrotu w EBX
# testowanie przez: echo $?
int $0x80 # Wywolanie systemu operacyjnego
```

Przykład 7-2 Kod programu `exit1.s` który wywołuje funkcję systemową `exit`

Dyrektywa `.globl _start` informuje że symbol `_start` ma być znany poza plikiem w którym występuje. Symbol `_start` jest etykietą która zastępuje adres instrukcji którą poprzedza czyli `movl $1, %eax.` Etykieta kończy się dwukropkiem i zastępuje adres komórki pamięci (programu lub danych).

Kompilacja, łączenie i wykonanie programu `exit1`

```
$as exit1.s --32 -o exit1.o
$ld exit1.o –m elf_i386 -o exit1
$./exit1
$echo $?
0
```

7.11 Mapa pamięci programu

Program ładujący umieszcza mający się wykonać program (zapisany w formacie ELF) w pamięci operacyjnej. Sekcja kodu `.text` zaczyna się od ustalonego adresu `0x08048000` bezpośrednio po której, w

kierunku wyższych adresów, nastepują sekcje danych `.data` i `.bss`. Z kolei, patrząc od góry, największy adres jaki może być użyty przez program w Linuksie to `0xbfffffff`. Od tego adresu zaczyna się stos, który jak wiemy, rośnie w kierunku niższych adresów. Pod adresem `0xbfffffff` znajduje się komórka o zawartości 0. Dalej znajdują się zmienne otoczenia i argumenty wywołania programu, nazwa programu a dalej liczba tych argumentów. Od tej komórki zaczyna się stos, czyli w momencie rozpoczęcia programu wskazuje na nią rejestr `%esp`. Do liczby argumentów, ich wartości i zmiennych otoczenia można odwołać się z funkcji `main` programu w języku C. Pokazuje to program `arg-env.c` dany poniżej.

```
// Program wyprowadza na konsole: argc, argv, otoczenie
#include <stdio.h>

int main(int argc, char * argv[],char * envp[]) {
    int i;
    printf("argc= %d\n",argc);
    for(i=0;i<argc;i++) {
      printf("argv[%d]=%s\n",i,argv[i]);
    }
    // Wyprowadzamy otoczenie ---
    while(*envp != NULL) {
           printf("%s\n",*envp);
           *envp++;
    }
}
```

Przykład 7-3 Program `arg-env.c` wypisujący liczbę argumentów i ich wartości

Uruchamiając program jak poniżej:

```
$./arg-env a1 a2 a3
```

otrzymamy wynik:

```
argc= 4
argv[0]= argumenty
argv[1]= a1
argv[2]= a2
argv[3]= a3
...
```

Pomiędzy szczytem stosu a końcem segmentu danych (ta lokacja pamięci oznaczana jest jako `brk`) znajduje się wolna przestrzeń. W tej może być umieszczony jest obszar na zmienne dynamiczne, tak zwana sterta (ang. *heap*). Sterta implementuje tak zwaną pamięć dynamiczną, w odróżnieniu od pamięci znajdującej się w segmencie danych `.data` i `bss` nazywanej pamięcią statyczną. Jej wielkość daje się obliczyć podczas kompilacji. Pamięć dynamiczna jest używana wtedy gdy z góry nie wiadomo ile pamięci będzie w programie potrzebne. Ma to miejsce na przykład przy użyciu list.
Jeżeli sterta zostanie użyta, zmienna `brk` odpowiednio się przesunie w kierunku wyższych adresów, tak aby wskazywać na koniec sterty a tym samym koniec segmentu danych (sterta umieszczona jest w segmencie danych). Segment danych można powiększać w trakcie działania programu za pomocą wywołania systemowego `brk`. Jest to wywołanie systemowe o numerze `45 (0x2d)`. Przed wywołaniem systemu w rejestrze `%eax` należy umieścić 45 a w rejestrze `%ebx` należy umieścić nowy adres końca segmentu danych. Gdy będzie tam 0 to zwrócony będzie aktualny koniec segmentu danych czyli wartość zmiennej `brk`.

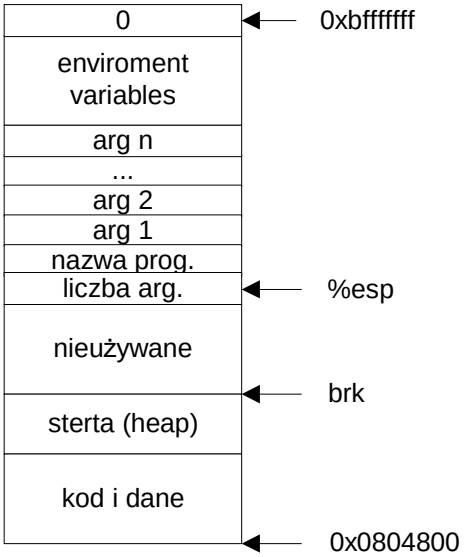
Wywołanie systemowe `brk` implementowane jest przez funkcję biblioteczną `brk`.

```
int brk(void *end_data_segment);
```

Jej argumentem jest nowy adres końca segmentu danych. Funkcja `sbrk` zwiększa segment danych o wartość argumentu `increment` i zwraca jego nową wartość.

```
void *sbrk(intptr_t increment)
```

Gdy jako `increment` podamy 0 , funkcja zwróci aktualną wartość zmiennej `brk`. Mapa pamięci programu pokazana jest na poniższym rysunku.



Rys. 7-2 Mapa pamięci programu w systemie Linux

Program może uzyskać określony obszar na stercie używając systemowej funkcji `malloc`.

```
void *malloc(size_t size);
```

Funkcja `malloc` rezerwuje na stercie `size` bajtów pamięci i zwraca wskaźnik na ten obszar. Pamięć ta może być zwolniona za pomocą funkcji `free`.

```
void free(void *ptr);
```

Przykład użycia funkcji `sbrk`, `malloc` i `free` podaje poniższy przykład.

```
// Demonstracja funkcji sbreak
// Kompilacja gcc sbreak3.c -o sbreak3
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>

void main() {
  int *p1, *p2;
  printf("sbrk(0) przed malloc(4): 0x%x\n", sbrk(0));
  p1 = (int *) malloc(4); // Rezerwacja 4 bajtow
  *p1 = 1;
  printf("sbrk(0) po `p1 = (int *) malloc(4)': 0x%x\n", sbrk(0));
  p2 = (int *) malloc(4);
  *p2 = 2;
  printf("sbrk(0) po `p2 = (int *) malloc(4)': 0x%x\n", sbrk(0));
  printf("p1 = 0x%x, *p1 = %d, p2 = 0x%x *p2 = %d\n", p1,*p1, p2, *p2);
}
```

Przykład 7-4 Demonstracja użycia funkcji `sbreak` – program `sbreak3.c`

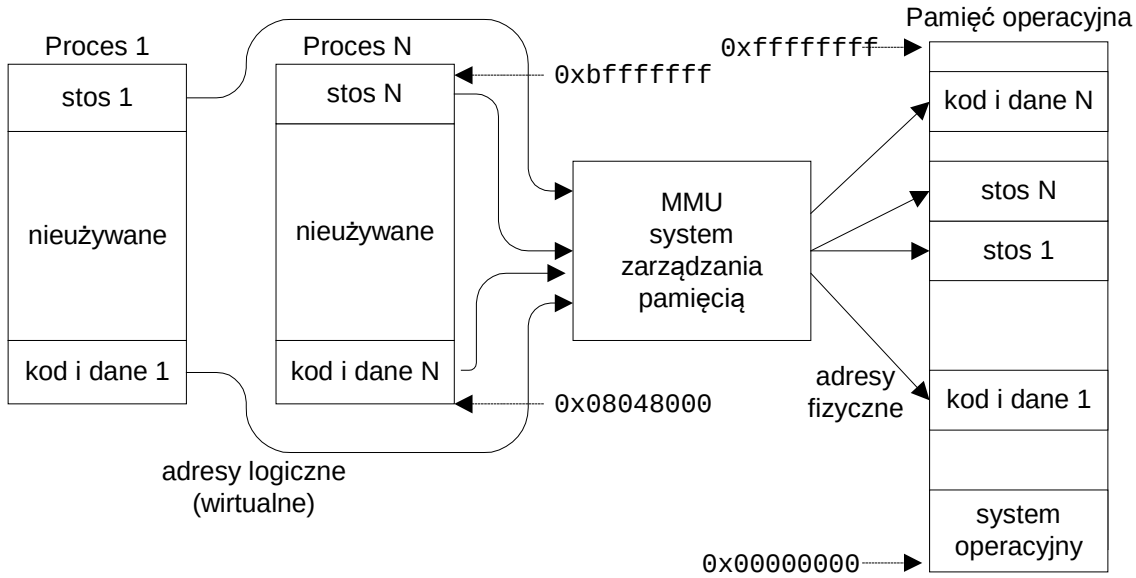
Wyniki uruchomienia programu dane są poniżej.

```
sbrk(0) przed malloc(4): 0x195d000
sbrk(0) po `p1 = (int *) malloc(4)': 0x197f000
sbrk(0) po `p2 = (int *) malloc(4)': 0x197f000
p1 = 0x195e010, *p1 = 1, p2 = 0x195e020 *p2 = 2
```

Analiza powyższych wyników pokazuje że w wyniku pierwszego działania funkcji `malloc(4)` wielkość `brk` przesunęła się znacznie więcej niż o 4 bajty (`0x197f000 - 0x195d000 = 0x22000 = 139264`) a w wyniku drugiego wywołania nie zmnieniła się w ogóle. Jest to wynik buforowania, najmniejszą jednostką przydziału pamięci jest strona.

Należy zauważyć że zmienna umieszczona na stercie (dynamiczna) jest dostępna tylko poprzez wskaźnik. W powyższym przykładzie jest to `*p1 i *p2`.

Powstaje pytanie jak to jest możliwe że w komputerze wykonuje się wiele procesów a wszytkie używają takich samych obszarów pamięci. Odpowiedź jest taka, że procesy widzą pamięć witualną z zakresu od `0x08048000` do `0xbfffffff` a nie pamięć fizyczną. Pamięć wirtualna jest transformowana do pamięci fizycznej przez mechanizm zarządzania pamięcią systemu operacyjnego, który z kolei wykorzystuje układ zarządzania pamięcią MMU procesora (ang. *Memory Management Unit*). Tak więc mimo że adresy logiczne poszczególnych procesów nakładają się, są one transformowane przez system zarządzania pamięcią w strony pamięci fizycznej które się nie nakładają. Tłumaczenie adresów wirtualne w fizyczne w pokazuje pokazuje poniższy rysunek.



Rys. 7-3 Tłumaczenie adresów logicznych (wirtualnych) na fizyczne.

7.12 Program Hello

Program Hello w języku C może wyglądać następująco:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
      char napis[] = "Hello word\n";
      write(1,napis,sizeof(napis));
      exit(33);
}
```

Przykład 7-1 Program `chello.c` w języku C

Odbiega on od zwyczajowej formy ale pokazuje użycie standardowego urządzenia wyjścia (oznaczanego jako `stdout`) które posiada zawsze numer 1. Jest więc ono widziane jako plik o uchwycie 1 i nie trzeba tego pliku otwierać. Z kolei funkcja `write` ma postać:

```
int write(int fdes, void *bufor, int nbytes)
```

`fdes`	Uchwyt do pliku zwracany przez funkcję open
`bufor`	Bufor w którym umieszczane są bajty przeznaczone do zapisu
`nbytes`	Liczba bajtów którą chcemy zapisać

Funkcja powoduje zapis do pliku identyfikowanego przez `fdes` `nbytes` bajtów znajdujących buforze `bufor`.

Poniżej pokazano program Hello napisany w języku assemblera GNU wykorzystuje ten sam mechanizm. Analiza tego programu pozwala na zorientowanie się w budowie programów w GNU assemblerze.

```
# program Hello w gnu assembler
# kompilacja: as hello2.s --32 -o hello2.o -g
# laczenie  : ld hello2.o –m elf_i386 -o hello2
# wykonanie : ./hello2
# Definicje ---------------------------------
STDIN       = 0         # standardowe wejscie
STDOUT      = 1         # standardowe wyjscie

SYSCALL = 0x80          # int 0x80 - wywolanie systemu operacyjnego

# Funkcje systemowe
SYSEXIT     = 1         # zakonczenie procesu
SYSWRITE    = 4         # numer funkcji write

EXIT_SUCCESS = 0  # kod powrotu

# segment danych ------------------------------------
 .data
 napis:
 .string "hello world!\n"
 len = . - napis

# segment tekstu (kodu) -----------------------------
.text
.global _start
 _start:
# Wywolanie funkcji pisz na stdout
        movl $SYSWRITE, %eax  # kod funkcji SYSWRITE
        movl $STDOUT, %ebx    # 1 arg. - syst. deskryptor stdout
        movl $napis, %ecx     # 2 arg. - adres początkowy napisu
        movl $len, %edx       # 3 arg. - długość łańcucha
        int $SYSCALL          # wywołanie przerwania programowego
                              # wywolanie systemu op
# Wywolanie funkcji exit (zakoncz proces)
movl $SYSEXIT, %eax      # kod funkcji SYSEXIT
        movl $EXIT_SUCCESS, %ebx # 1 arg. -- kod powrotu z programu
        int $SYSCALL           # wywołanie przerwania programowego
```

Przykład 7-5 Kod programu `hello2.s`

Program używa wyjaśnionych poniżej symboli:

`.text` - część pliku zawiera kod programu
`.global _start` – etykieta o zasięgu globalnym, nazywa się `start`
`_start:` - od tej etykiety zaczyna się program
`.data` - część programu zawierająca dane
`napis:` - etykieta dla napisu
`len = . – napis` - długość napisu

`len` - jest to zmienna, która zawiera długość napisu. Kropka oznacza "aktualny adres" w pamięci (w naszym przypadku koniec napisu), a "napis" - adres etykiety, pod którą zawarto początek napisu. Różnica koniec - początek daje długość napisu.

8. Uruchamianie programów za pomocą narzędzia gdb

Rzadko zdarza się by napisany przez nas program od razu działał poprawnie. Na ogół zawiera wiele błędów które trzeba pracowicie poprawiać. Typowy cykl uruchamiania programów polega na ich edycji, kompilacji i wykonaniu. Przy kompilacji mogą się ujawnić błędy kompilacji które wymagają poprawy a gdy program skompiluje się poprawnie przystępujemy do jego wykonania. Często zdarza się że uruchamiany program nie zachowuje się w przewidywany przez nas sposób. Wówczas należy uzyskać dodatkowe informacje na temat:

- Ścieżki wykonania programu
- Wartości zmiennych a ogólniej zawartości pamięci związanej z programem

Informacje takie uzyskać można na dwa sposoby:

- Umieścić w kodzie programu dodatkowe instrukcje wyprowadzania informacji o przebiegu wykonania i wartości zmiennych.
- Użyć programu uruchomieniowego (ang. *Debugger*)

Gdy używamy pierwszej metody (możliwe w C, trudno użyć w assemblerze), dodatkowe informacje o przebiegu wykonania programu są zwykle wypisywane na konsoli za pomocą instrukcji `printf` lub też zapisywane do pliku. Po uruchomieniu programu, instrukcje wypisywania dodatkowych informacji są z programu usuwane. Użycie pierwszej metody jest w wielu przypadkach wystarczające. Jednak w niektórych, bardziej skomplikowanych przypadkach, wygodniej jest użyć programu uruchomieniowego. Program taki daje następujące możliwości:

- Uruchomienie programu i ustawienie dowolnych warunków jego wykonania (np. argumentów, zmiennych otoczenia, itd)
- Doprowadzenie do zatrzymania programu w określonych warunkach.
- Sprawdzenie stanu zatrzymanego programu (np. wartości zmiennych, zawartość rejestrów, pamięci, stosu)
- Zmiana stanu programu (np. wartości zmiennych) i ponowne wznowienie programu.

W świecie systemów klasy POSIX , szeroko używanym programem uruchomieniowym jest `gdb` (ang. *gnu debugger*) który jest częścią projektu GNU Richarda Stallmana. Może on być użyty do uruchamiania programów napisanych w językach C, C++, assembler, Ada , Fortran, Modula-2 i częściowo OpenCL. Program działa w trybie tekstowym, jednak większość środowisk graficznych IDE takich jak Eclipse czy CodeBlocks potrafi się komunikować z `gdb` co umożliwia pracę w trybie okienkowym. Istnieją też środowiska graficzne specjalnie zaprojektowane do współpracy z `gdb` jak chociażby DDD (ang. *Data Display Debugger*). Program gdb posiada wiele możliwości i obszerną dokumentację podaną w a tutaj podane zostaną tylko najważniejsze polecenia.

Kompilacja programu

Aby możliwe było uruchamianie programu z użyciem `gdb` testowany program należy skompilować z kluczem: `-g`. Użycie tego klucza powoduje że do pliku obiektowego z programem dołączona zostanie informacja o typach zmiennych i funkcji oraz zależność pomiędzy numerami linii programu a fragmentami kodu binarnego. Rozważmy przykładowy program `maximum.s` podany w [3]. Aby skorzystać z debuggera program `maximum.s` należy skompilować następująco:

```
as maximum.s --32 -o maximum.o -g
ld maximum.o -m elf_i386 -o maximum
```

```
#znajdowanie maksimum w tablicy tablica
# rejestr %edi  - indeks tablicy
# rejestr %ebx  - aktualnie najwiekszy element
# rejestr %ebx  - biezacy element
.section .data
 tablica: #To są dane w tablicy zawierającej liczby long
.long 3,67,34,224,45,75,54,34,44,33,22,11,66,0
.section .text
.globl _start
_start:
movl $0, %edi # przeslij 0 do rejestru indeksowego
movl tablica(,%edi,4), %eax # przeslij do eax kolejna liczbe z tablicy
movl %eax, %ebx  # pierwszy element jest na razie najwiekszy
start_loop:   # start petli
cmpl $0, %eax # sprawdz czy koniec tablicy (jest tam 0)
je loop_exit
incl %edi # zwieksz licznik
movl tablica(,%edi,4), %eax
cmpl %ebx, %eax # porownaj czy %eax  wiekszy od maksimum
jle start_loop  # skocz do poczatku petli gdyz %eax jest mniejszy od max
movl %eax, %ebx # w %eax jest aktualne maksimum
jmp start_loop  # skocz na oczatek petli
loop_exit:
# %ebx zawiera kod powrotu funkcji exit i jest to maksimum
movl $1, %eax #1 jest kodem wywolania exit()
int $0x80
```

Przykład 8-1 Program `maximum.s`

8.1 Uruchomienie debuggera gdb i ustawianie środowiska

Program `gdb` uruchamia się w następujący sposób:

```
gdb [opcje] [prog [obraz-pam lub pid]]
gdb [opcje] --args prog [argumenty_prog ...]
```

gdzie:

`opcje`	Opcje programu które można uzyskać pisząc: `gdb --h`
`prog`	Nazwa pliku wykonywalnego
`obraz-pam`	Obraz pamięci utworzony przy awaryjnym zakończeniu programu
`pid`	Pid procesu do którego chcemy się dołączyć
`args`	Argumenty programu

Najprostszy sposób uruchomienia debuggera `gdb` w celu uruchamiania programu zawartego w pliku `prog` to napisanie na konsoli polecenia: `gdb prog`

Można też dołączyć się do już działającego programu. W tym celu po nazwie programu należy podać jego `pid` co pokazuje `Tabela 8-1`. Program `gdb` może też służyć do analizy przyczyny awaryjnego zakończenia programu. Gdy proces jest kończony, na skutek otrzymania jednego z pewnych sygnałów, system operacyjny tworzy plik zawierający obraz pamięci procesu. Obraz ten może być analizowany przez `gdb` w celu znalezienia przyczyny awaryjnego zakończenia procesu. Aby dokonać analizy procesu `prog` który został awaryjnie zakończony, a jego oraz pamięci został zapisany w pliku `core`, `gdb` uruchamia się jak następuje: `gdb prog core`. Jeszcze jeden wariant uruchomienia programu `gdb` pozwala na podanie argumentów uruchamianego programu. Gdy program `prog` należy uruchomić z argumentami `a1 a2 ... an` to wtedy `gdb` należy uruchomić z opcją `--arg` jak następuje: `gdb --arg prog a1 a2 ... an`. Typowe sposoby uruchomienia programu `gdb` podaje `Tabela 8-1`.

`gdb prog`	Zwykłe uruchomienie `gdb` dla programu zawartego w pliku `prog`
`gdb prog core`	Uruchomienie `gdb` dla programu z pliku `prog`. Obraz pamięci znajduje się w pliku core.
`gdb prog pid`	Dołączenie się do działającego programu, oprócz nazwy podajemy też jego `pid`
`gdb --args prog argumenty`	Uruchomienie `gdb` dla programu z argumentami
`gdb –help`	Uzyskiwanie pomocy

Tabela 8-1 Różne sposoby uruchomienia programu gdb

Program `gdb` kończy się wpisując polecenie `quit`, skrót `q` lub też kombinację klawiszy `Ctrl+d`.
Możemy uruchomić program `gdb` w celu testowania podanego w programu test. W tym celu piszemy polecenie:

```
$gdb maximum
```

Po wpisaniu tego polecenia zgłasza się program `gdb` i oczekuje na wprowadzenie poleceń.

8.2 Uzyskiwanie pomocy

Program `gdb` posiada znaczną liczbę poleceń. Wpisując polecenie `help` uzyskujemy zestawienie kategorii poleceń, wpisując polecenie `help all` uzyskujemy zestawienie wszystkich poleceń.

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

Ekran 8-1 Działanie polecenia help

8.3 Listowanie programu źródłowego

Uzyskanie fragmentu kodu źródłowego następuje przez użycie polecenia `list`. Polecenie to występować może w różnych wariantach co pokazuje `Tabela 8-2`.

`list`	Listowanie fragmentu kodu źródłowego począwszy od bieżącej pozycji
`list nr`	Listowanie fragmentu kodu źródłowego w pobliżu linii o numerze `nr`
`list pocz, kon`	Listowanie fragmentu kodu źródłowego od linii `pocz` do linii `kon`
`list +`	Listowanie nastepnego fragmentu kodu źródłowego (od pozycji bieżącej)
`list -`	Listowanie poprzedniego fragmentu kodu źródłowego (od pozycji bieżącej)

Tabela 8-2 Polecenia listowania fragmentu kodu źródłowego

Gdy mamy już uruchomiony gdb i testujemy program test możemy wylistować fragment kodu źródłowego pisząc polecenie list jak pokazuje `Ekran 8-2`.

```
root@kali:/home/juka/ak/barlet# gdb maksimum
GNU gdb (Debian 7.11.1-2) 7.11.1
…
Type "show configuration" for configuration details.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from maksimum...done.
(gdb) l
1     #znajdowanie maksimum w tablicy tablica
2     # rejestr %edi  - indeks tablicy
3     # rejestr %ebx  - aktualnie najwiekszy element
4     # rejestr %ebx  - biezacy element
5     .section .data
6      tablica: #To są dane w tablicy zawierającej liczby long
7     .long 3,67,34,224,45,75,54,34,44,33,22,11,66,0
8     .section .text
9     .globl _start
10    _start:
```

Ekran 8-2 Listowanie kodu źródłowego

8.4 Zatrzymywanie procesu, punkty zatrzymania

Testowanie programów polega zwykle na próbie ich wykonania i zatrzymania, po czym następuje zbadanie stanu programu. Momenty zatrzymania określane przez tak zwane punkty zatrzymania (ang. *breakpoint*). Są trzy metody zdefiniowania punktu zatrzymania:

- Wskazanie określonej linii w kodzie programu lub też nazwy fukcji. Jest to zwykły punkt zatrzymania.
- Zatrzymanie programu gdy zmieni się wartość zdefiniowanego przez nas wyrażenia (ang. *watchpoint*).
- Zatrzymanie programu gdy zajdzie określone zdarzenie (ang. *catchpoint*) jak wystąpienie wyjątku czy załadowanie określonej biblioteki.

Tworzonym punktom zatrzymania nadawane są kolejne numery począwszy od 1. Po utworzeniu mogą one być aktywowane (ang *enable*), dezaktywowane (ang. *disable*) i kasowane (ang. *delete*). Najprostszym sposobem ustawienia punktu zatrzymania jest użycie polecenia: `break nr_lini`. Następuje wtedy ustawienie punktu zatrzymania w danej linii programu. Polecenie: `info break` powoduje wypisanie ustawionych punktów wstrzymania. Możemy teraz, w naszym przykładowym programie, ustawić punkt zatrzymania na linii 10 co robimy poleceniem: `break 10` jak pokazuje Ekran 8-3.

```
(gdb) list 10
10    _start:
11    movl $0, %edi # przeslij 0 do rejestru indeksowego
12    movl tablica(,%edi,4), %eax # przeslij do eax kol liczbe z tablicy
13    movl %eax, %ebx  # pierwszy element jest na razie najwiekszy
14    start_loop: # start petli
(gdb) break 10
Breakpoint 1 at 0x8048074: file maksimum.s, line 10.
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x08048074 maksimum.s:10
(gdb)
```

Ekran 8-3 Ustawienie punktu zatrzymania

Można także zatrzymać program przy wejściu do danej funkcji. Używamy wtedy polecenia:

```
break nazwa_funkcji
```

Punkty wstrzymania można kasować za pomocą polecenia: `clear nr_linii`.

Polecenie `break` występuje w wielu wariantach co opisane jest w dokumentacji. Między innymi można ustawić zatrzymanie procesu gdy spełniony jest pewien warunek. Aby uzyskać informację o ustawionych punktach zatrzymania używamy polecenia: `info break` . Jego działanie pokazano powyżej.

Ustawienie punktu zatrzymania	`break nr_linii`
	`break nazwa_funkcji`
	`break nr_linii if warunek`
		`b`

Ustawienie pułapki, program zatrzyma się gdy zmieni się wartość obserwowanej zmiennej	`watch nazwa_zmiennej`	`w`
Listowanie punktów zatrzymania	`info break`	
Kasowanie punktu zatrzymania	`clear nr_linii`	

Tab. 8-1 Polecenia dotyczące punktów zatrzymania

8.5 Uruchamianie procesu

Jeżeli w programie ustawiono punkty zatrzymania to można go uruchomić. Wykonuje się to poprzez polecenie `run`.

```
 (gdb) run
Starting program: /home/juka/ak/barlet/maksimum
Breakpoint 1, _start () at maksimum.s:11
11    movl $0, %edi # przeslij 0 do rejestru indeksowego
(gdb)
```

Ekran 8-4 Wykonanie programu do punktu zatrzymania

W chwili obecnej program zatrzymał się na linii 11. Aby wykonać jeden krok (następną instrukcję) wpisujemy polecenie `step`. Program przechodzi do kolejnej instrukcji.

```
(gdb) step
12    movl tablica(,%edi,4), %eax # przeslij do eax kolejna liczbe z
tablicy
(gdb) step
13    movl %eax, %ebx  # pierwszy element jest na razie najwiekszy
```

Ekran 8-5 Praca krokowa przy użyciu polecenia `step`

Polecenie step może posiadać parametr określający liczbę kroków do wykonania. Będzie wtedy postaci:

```
step liczba_krokow
```

Innym poleceniem umożliwiającym pracę krokową jest polecenie `next` lub `next [liczba_krokow].` Różni się ono od polecenia `step` że nie wchodzi do funkcji o ile taka byłaby wywoływana.

Jeżeli chcemy by program wykonywał się do punktu zatrzymania lub do końca użyjemy polecenia: `continue`.

Listowanie programu	`list [numer linii]`	`skrót`
Uruchomienie programu	`run [argumenty]`	

Ustawienie punktu zatrzymania	`break nr_linii`
	`break nazwa_funkcji`
	`break nr_linii if warunek`
		`b`

Ustawienie pułapki, program zatrzyma się gdy zmieni się wartość obserwowanej zmiennej	`watch nazwa_zmiennej`	`w`
Listowanie punktów zatrzymania	`info break`	
Kasowanie punktu zatrzymania	`clear nr_linii`	
Wyprowadzenie wartości zmiennych	`print nazwa_zmienne`	`p`
Kontynuacja do następnego punktu wstrzymania	`continue`	`c`
Wykonanie następnej instrukcji lub przejście N instrukcji dalej, nie wchodzimy do funkcji	`next` `next N`	`n`
Wykonanie następnej instrukcji lub przejście N instrukcji dalej, wejście do funkcji	`step` `step N`	`s`
Wyjście z funkcji	`finish`	
Uzyskanie pomocy	`help`	`h`
Ustawienie wartości zmiennej `var` na `wartosc`	`set variable var=wartosc`	
Ustawienie wartości zmiennej `var` na `wartosc`	`print var=wartosc`	
Ustawienie argumentów	`set args arg1 arg2 …`	
Wyświetlenie argumentów	`show args`	
Zakończenie pracy programu	`quit, kill`	`q`

Tabela 8-3 Najczęściej używane polecenia kontroli wykonania programu gdb

8.6 Wyświetlanie rejestrów i zawartości pamięci

Zawartość rejestrów za pomocą polecenia: `info reg`

```
(gdb) info reg
eax            0x3      3
ecx            0x0      0
edx            0x0      0
ebx            0x3      3
esp            0xbffff460     0xbffff460
ebp            0x0      0x0
esi            0x0      0
edi            0x1      1
eip            0x8048088      0x8048088 <start_loop+6>
eflags         0x10202  [ IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
```

Ekran 8-6 Wyświetlenie rejestrów procesora za pomocą polecenia `info reg`

Rejestry można wyświetlać także indywidualnie używając polecenia `print $nazwa_rejestru`. W gdb nazwy rejestrów poprzedzone są znakiem $. Przykład wyświetlenia rejestru `eax` dany jest poniżej.

```
(gdb) print $eax
$1 = 3
```

Można także wyświetlić rejestr poleceniem:

`info r nazwa_rejestru`

```
gdb) info r eax
eax   0x3   3
```

Zawartość flag można wyświetlić za pomocą polecenia `print $eflags` jak pokazano poniżej.

```
(gdb) print $eflags
$15 = [ CF AF SF IF ]
```

Wartości zmiennych globalnych i lokalnych można wyświetlić za pomocą polecenia: `info variables` co pokazano poniżej.

```
(gdb) info variables
All defined variables:

Non-debugging symbols:
0x0804909e  tablica
0x080490d6  __bss_start
0x080490d6  _edata
0x080490d8  _end
```

Ekran 8-7 Wyświetlenie zmiennych programu za pomocą polecenia `info variables`

Wyświetlanie zawartości rejestrów	`info reg`	`i r`
Wyświetlanie zawartości danego rejestru	`info reg nazwa_rej`	`i r eax`
Wyświetlanie zawartości danego rejestru, `f` określa format: x - heksadecymalny o – oktalny d – dziesiętny u – dziesiętny bez znaku a – adres f – zmiennoprzecinkowy (float)	`print/f $nazwa_rej`	`i r eax`
Wyświetlanie zawartości wszystkich rejestrów	`info reg all`	`I r a`
Wyświetlanie zawartości zmiennych	`info variables`	`I r a`

Tab. 8-2 Wyświetlanie zawartości rejestrów

Ważną funkcją jest wyświetlanie zawartości pamięci danych w tym zmiennych. Do tego celu służy polecenie:

`x/rsf &adres`

Występujące po znaku ukośnika / atrybuty `rsf` specyfikują sposób wyświetlania pamięci co pokazuje poniższa tabela.

Wyświetlanie zawartości pamięci o danym adresie	`x/rsf &adres`
	`r – liczba powtórzeń (ang. repeat)`
	`s – długość wyświetlanej jednostki (ang. size)`
	`b(byte), h(halfword), w(word), g(giant, 8 bytes)`
	`f – format wyświetlanej jednostki`
	`x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char), s(string)`	

Wyświetlanie zmiennych lub rejestrów	`print/f $rejestr`
	`print/f zmienna`
	`f – format wyświetlanej jednostki`
	`x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char), s(string)`	

Tab. 8-3 Polecenia wyświetlania rejestrów i zawartości pamięci

Możemy wyświetlić zawartość zmiennej tablica: z Przykład 6-18 używając polecenia `x/10wd &tablica` co pokazano poniżej. W tym poleceniu `10` oznacza liczbę wyświetlanych jednostek, `w` że są to słowa a `d` specyfikuje dziesietny format wyświetlania

```
5     .section .data
6      tablica: #To są dane w tablicy zawierającej liczby long
7     .long 3,67,34,224,45,75,54,34,44,33,22,11,66,0
```

Kod 8-1 Zmienna tablica wyspecyfikowana w kodzie programu maksimum.s

```
(gdb) x/10wd &tablica
0x804909e:  3     67    34    224
0x80490ae:  45    75    54    34
0x80490be:  44    33
```

Przykład 8-1 Wyświetlanie zawartości zmiennej `tablica` w gdb

8.7 Zmiana zawartości zmiennych i rejestrów

Zawartość zmiennej lub rejestru można zmienić za pomocą polecenia:

```
print nazwa_zmiennej=wartość
set var nazwa_zmiennej=wartość
```

```
(gdb) print $eax=3
$8 = 3
(gdb) print $eax
$9 = 3
(gdb) set var $eax=4
(gdb) print $eax
$10 = 4
```

Przykład 8-2 Zmiana zawartości rejestru `%eax` w gdb

8.8 Ramki stosu

W językach programowania szeroko używane są funkcje. Gdy funkcja jest wywoływana na stosie zapamiętywany jest adres powrotu i dalej zmienne lokalne funkcji. Ramki posiadają kolejne numery i można się pomiędzy nimi przemieszczać. Wyświetlenie rozszerzonej informacji o aktualnej ramce może być uzyskane za pomocą polecenia `info frame` jak pokazano poniżej.

```
(gdb) info frame
Stack level 0, frame at 0x0:
 eip = 0x8048087 in start_loop (maximum.s:34); saved eip = <unavailable>
 Outermost frame: outermost
 source language asm.
 Arglist at unknown address.
 Locals at unknown address, Previous frame's sp in esp
```

Przykład 8-3 Wyświetlenie informacji o ramce stosu

Ramkę można zmienić za pomocą polecenia

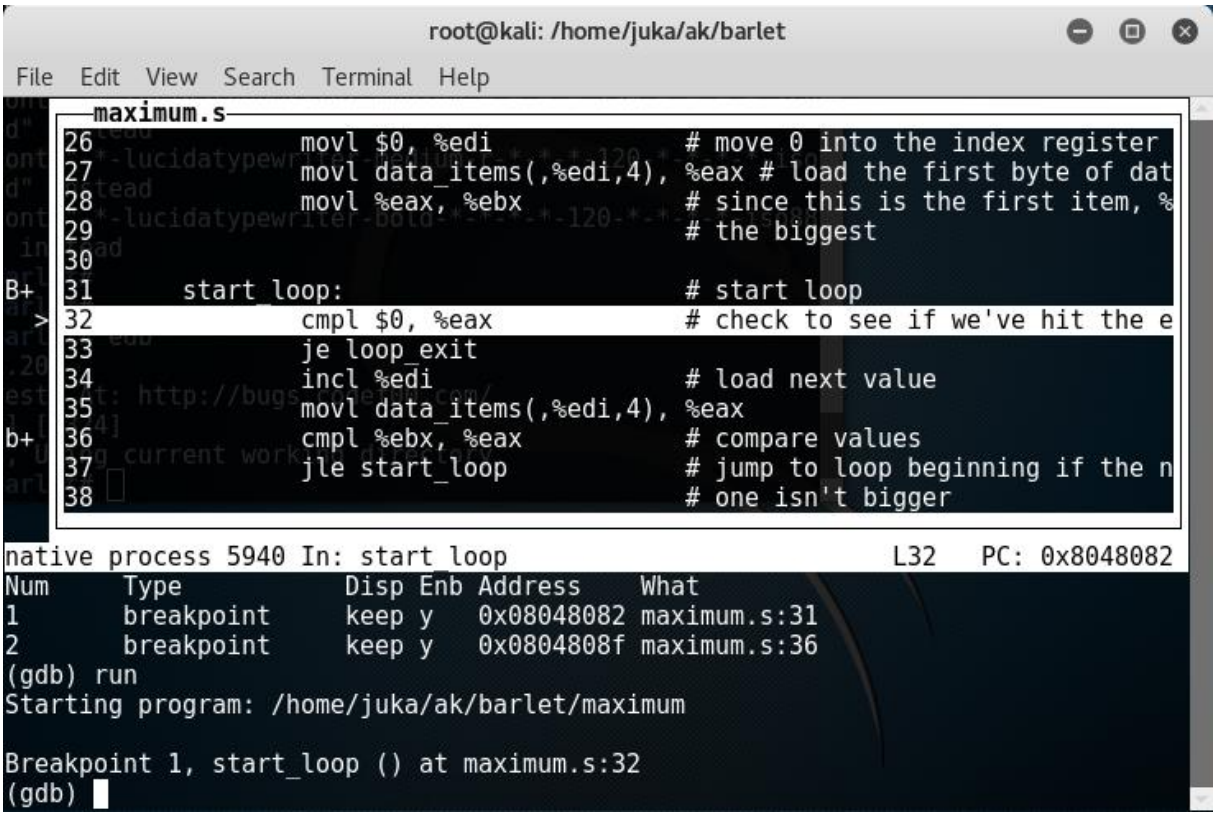
```
frame numer_ramki
```

a także poleceń: `up` i `down`. Polecenie: backtrace pozwala na zbadanie jak program dotarł do aktualnego miejsca.

8.9 Praca w trybie semigraficznym

Debugger gdb może pracować w trybie semigraficznym. Należy go uruchomić z opcją `–tui` , przykładowo

```
gdb –tui maximum
```



Ekran 8-8 Debugger gdb w postaci semigraficznej (opcja –tui)

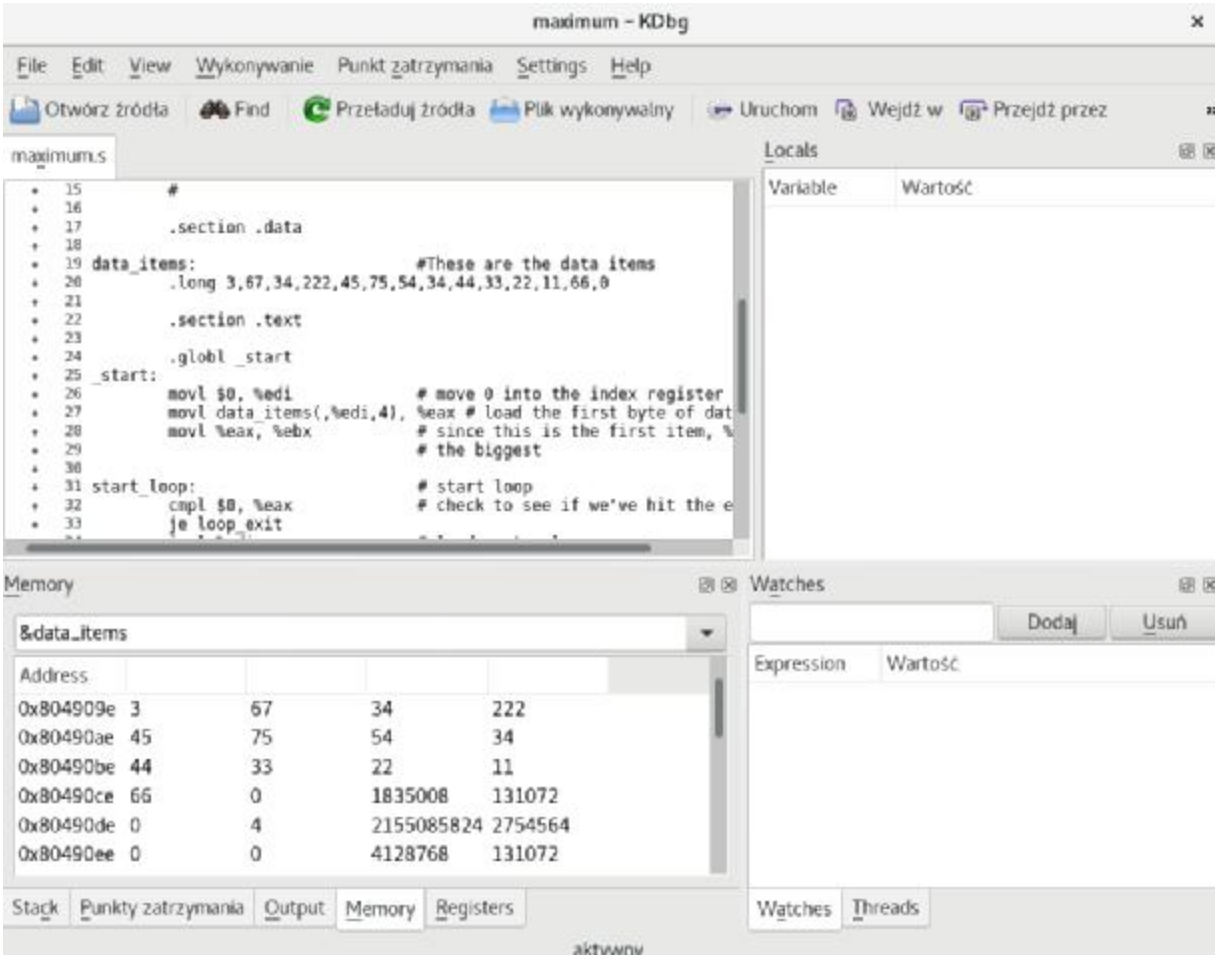
Istnieją także graficzne interfejsy do debuggera gdb. Wymienić tu można programy ddd, xgdb, kgdb.

8.10 Nakładka graficzna kdbg

Debugger gdb pracuje w trybie tekstowym. Ma to swoje zalety, np. takie że można pracować przez terminal szeregowy co ma znaczenie w systemach wbudowanych. Nie jest jednak wygodne, gdyż pamiętanie wielu skrótów poleceń nie jest zbyt twórcze i brak możliwości jednoczesnej obserwacji działania programu i jego skutków (rejestry, pamięć). Użytecznym narzędziem są nakładki graficzne na program gdb, jedną z nich jest kdbg. (patrz http://www.kdbg.org/manual/index.html). Program mający być uruchamiany, podobnie jak poprzednio kompilujemy z opcją -g. Debugger kdbg uruchamiamy z konsoli: `kdbg nazwa_programu` np.:

```
$kdbg maximum
```

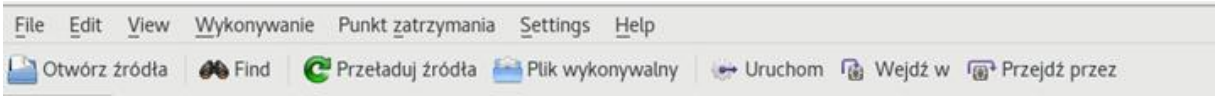
Po uruchomieniu pokazuje się formularz programu.



Ekran 8-9 Formularz startowy programu kdbg, uruchamiamy program maximum.s

8.10.1 Polecenia programu

Podstawowy opis programu znajduje się w http://www.kdbg.org/manual/index.html. Górna belka programu zawiera podstawowe opcje pokazane poniżej.



File - operowanie na plikach, otwarcie programu wykonywalnego, źródłowego, zakończenie programu.

Edit - odnajdowanie poleceń i fragmentów pliku źródłowego

View - można oznaczyć co ma być obserwowane: stos, zmienne lokalne, rejestry, punkty zatrzymania, pamięć, pułapki (ang. Watchpoints), wyjście standardowe, watki.

Wykonywanie - sterowanie przebiegiem programu

Punkt zatrzymania – ustawianie punktów zatrzymania

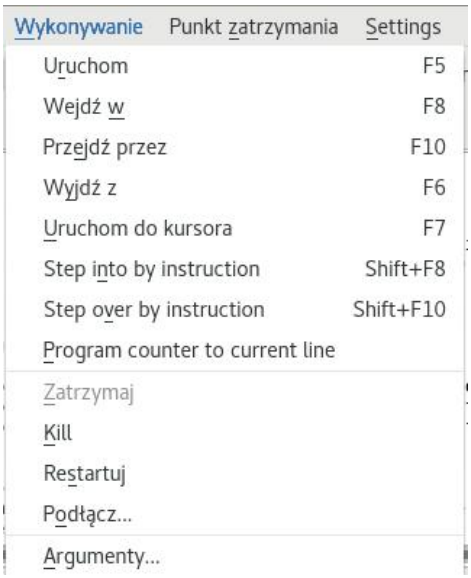
Settings - ustawienia opcji

Help - pomoc

Można korzystać także z przycisków dolnej części belki pokazanych poniżej. Ważny jest przycisk Uruchom.

Jego naciśniecie spowoduje uruchomienie programu lub przejście do kolejnego punktu zatrzymania.

Do ważniejszych menu należy Wykonywanie którego zawartość pokazana jest poniżej.



Ekran 8-10 Menu Wykonywanie

Ważniejsze pozycje to:

Uruchom F5 - Wybór opcji spowoduje uruchomienie programu. Należy wcześniej ustawić punkty zatrzymania gdyż inaczej program wykona się do końca i nic nie zobaczymy. Kolejne naciśnięcie spowoduje przejście do kolejnego punktu zatrzymania.

Wejdź w F8 - Wybór tej opcji spowoduje wykonanie następnej instrukcji. Gdy będzie nią wywołanie funkcji, nastąpi przejście do kodu funkcji.

Przejdź przez F10- Wybór tej opcji spowoduje wykonanie następnej instrukcji. Gdy będzie nią wywołanie funkcji, nastąpi przejście do kodu funkcji i wyjście z niej bez zatrzymania wewnątrz.

Wyjdź F6 - Opuszczenie bieżącej funkcji

Uruchom do kursora F7 - Wybór tej opcji spowoduje wykonanie instrukcji wskazywanej kursorem.

8.10.2 Okna programu

Program kdbg posiada następujące okna:

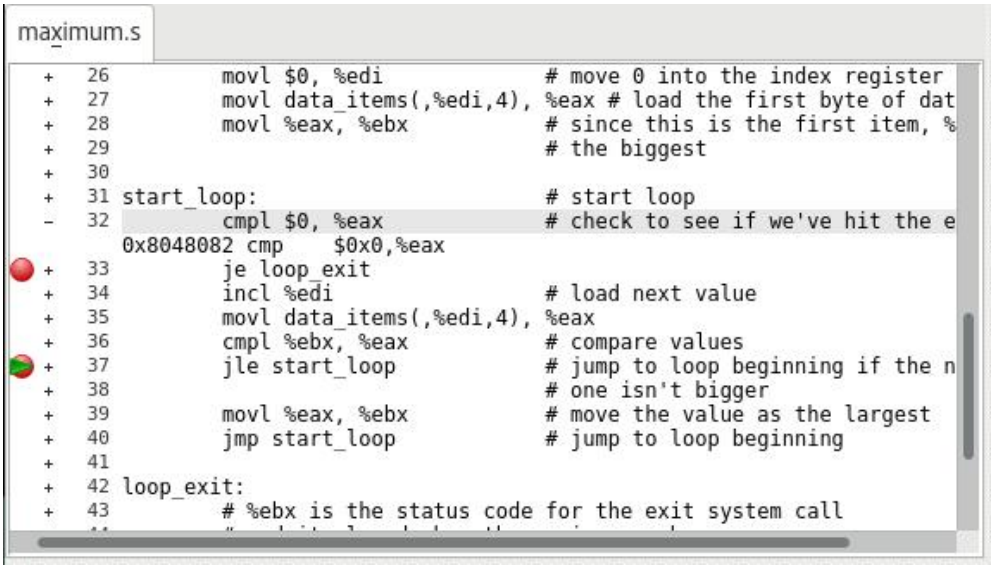
Okno kodu - wyświetlany jest wykonywany kod

Okno danych - Wyświetlane są rejestry, stos, pamięc, wyjście, punkty zatrzymania

Okno zmiennych lokalnych - Wyświetlane są zmienne lokalne

Okno pułapek - Wyświetlane są pułapki (ang. Watchpoints)

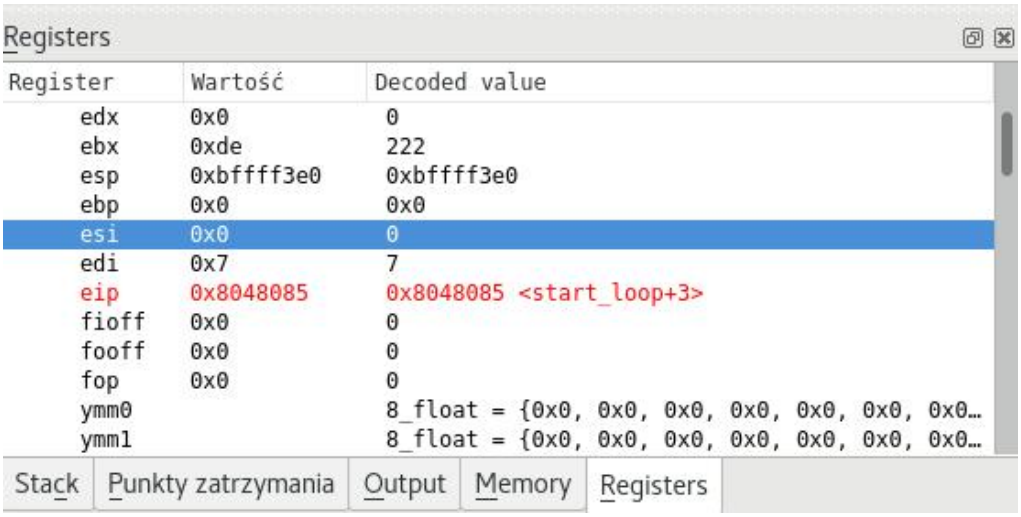
Okno kodu
W oknie kodu wyświetlany jest kod programu.



Ekran 8-11 Okno kodu

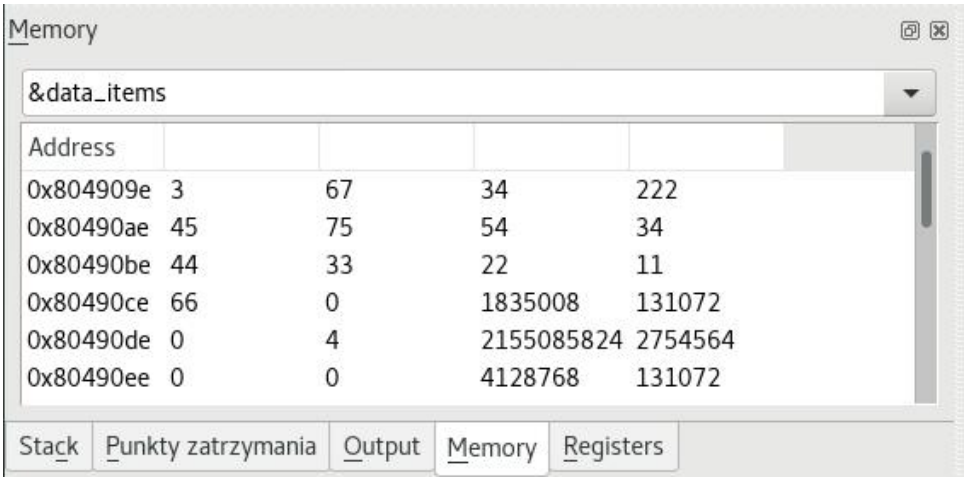
Klikając w początek danej linii można ustawić tam punkt zatrzymania. W powyższym oknie punkty zatrzymania są w liniach 33 i 37. Powtórne kliknięcie kasuje punkt zatrzymania. Kliknięcie w znak + spowoduje wyświetlanie adresu rozkazu. Zielony podświetlenie linii wskazuje bieżący wykonywany rozkaz.

Okno danych
Okno danych może pokazywać: stos, Punkty zatrzymania, rejestry, pamięc, ekran wyjściowy.



Ekran 8-12 Okno danych, wyświetlane sa rejestry

Wybór tego co ma być wyświetlane może być dokonany po kliknięciu w odpowiednią zakładkę. Na powyższym rysunku widać zawartość rejestrów w tym licznika rozkazów `eip`, widać na jaką instrukcję licznik ten wskazuje (`<start_loop + 3>`). Kliknięcie prawym klawiszem myszy na pole okna pozwala na zmianę formatu wyświetlania. Okno danych pozwala także na wyświetlenie obszarów pamięci o podanym adresie. Pokazuje to poniższy przykład. W polu adresu należy podać adres obszaru pamięci. W przykładzie podano wyrażenie `&data_items` co jest adresem tej zmiennej.



Ekran 8-13 Okno danych, wyświetlane są zmienne

8.11 Proste programy

Poniżej podane zostaną proste przykłady ilustrujące wykonywanie instrukcji w assemblerze

8.11.1 Dodawanie dwóch liczb

Poniżej podany został kod programu dodawania dwóch liczb.

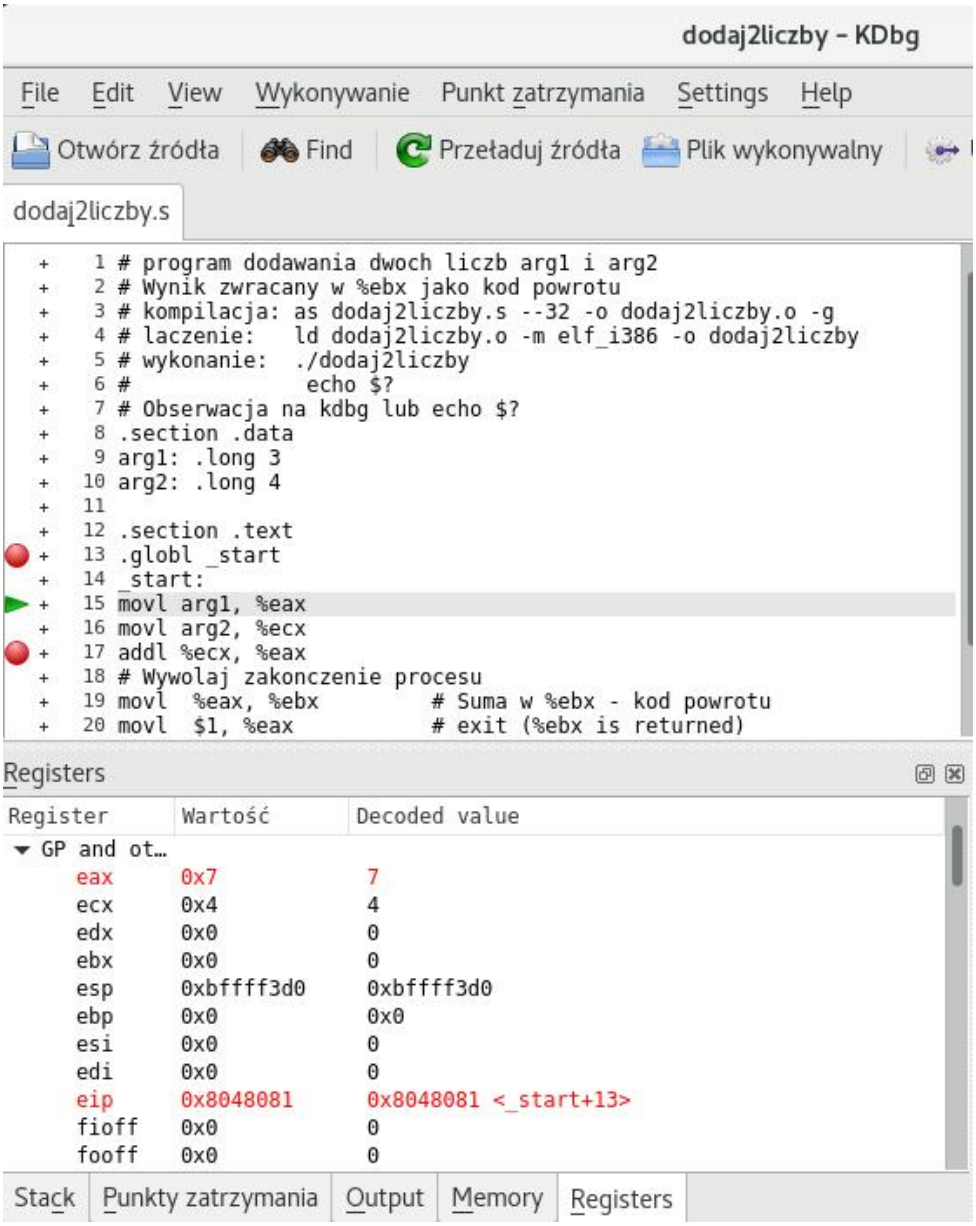
```
# program dodawania dwoch liczb arg1 i arg2
# Wynik zwracany w %ebx jako kod powrotu
# kompilacja: as dodaj2liczby.s --32 -o dodaj2liczby.o -g
# laczenie:   ld dodaj2liczby.o -m elf_i386 -o dodaj2liczby
# wykonanie:  ./dodaj2liczby
#              echo $?
# Obserwacja na kdbg lub echo $?
.section .data
arg1: .long 3
arg2: .long 4

.section .text
.globl _start
_start:
movl arg1, %eax
movl arg2, %ecx
addl %ecx, %eax
# Wywolaj zakonczenie procesu
movl  %eax, %ebx         # Suma w %ebx - kod powrotu
movl  $1, %eax           # exit (%ebx is returned)
int   $0x80
```

Przykład 8-2 Dodawanie dwóch liczb, program `dodaj2liczby.s`

Liczby zadane są jako zmienne arg1 i arg2 w postaci liczb int. Program kompilujemyi wykonujemy jak niżej:

```
$as dodaj2liczby.s --32 -o dodaj2liczby.o -g
$ld dodaj2liczby.o -m elf_i386 -o dodaj2liczby
$./dodaj2liczby
$ echo $?
```

Należy też zaobserwować jego działanie w debuggerze `kdbg` wstawiając punkt zatrzymania na instrukcji dodawania.



Ekran 8-14 Działanie programu `dodaj2liczby.s` obserwowane na debuggerze kdbg

8.11.2 Demonstracja skoku warunkowego

Kolejny program demonstruje skok warunkowy. Oblicza on wartość bezwzględną różnicy dwóch argumentów co pokazuje poniższy kod w języku C.

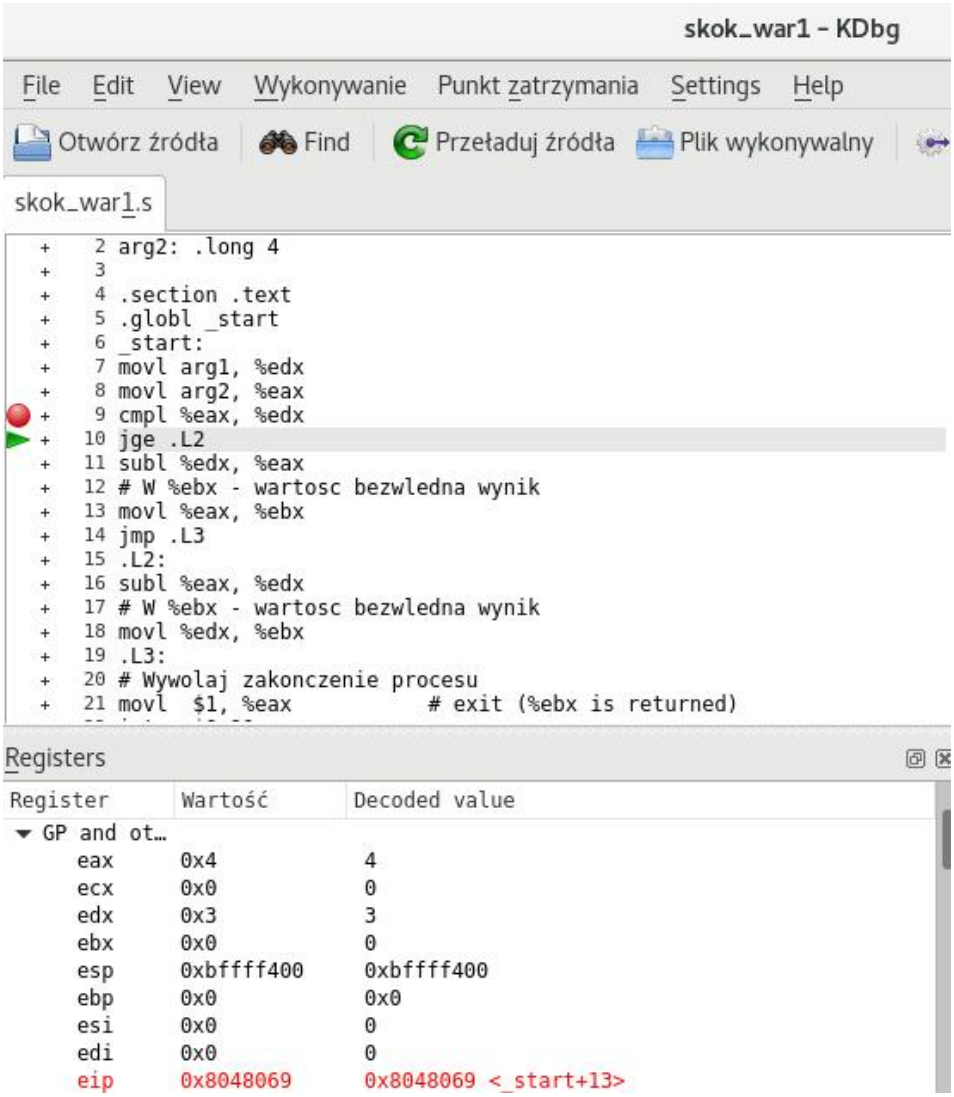
```
int absdiff(int arg1, int arg2) {
   if(arg1 < arg2)
      return (arg2 – arg1)
    else
      return (arg1 – arg2)
}
```

Poniżej pokazano ten kod w assemblerze.

```
# Demonstracja skoku warunkowego
# program oblicza wartosc bezwzgledna abs(arg1 - arg2)
# wynik w %ebx
# kompilacja: as skok_war1.s --32 -o skok_war1.o -g
# laczenie:   ld skok_war.o -m elf_i386 -o skok_war
# obserwacja w kdbg
arg1: .long 3
arg2: .long 4
.section .text
.globl _start
_start:
movl arg1, %edx
movl arg2, %eax
cmpl %eax, %edx
jge .L2
subl %edx, %eax
# W %ebx - wartosc bezwledna wynik
movl %eax, %ebx
jmp .L3
.L2:
subl %eax, %edx
# W %ebx - wartosc bezwledna wynik
movl %edx, %ebx
.L3:
# Wywolaj zakonczenie procesu
movl  $1, %eax           # exit (%ebx is returned)
int   $0x80
```

Przykład 8-3 Program demonstracji skoku warunkowego `skok_war1.s`

Działanie programu możemy obserwować w debuggerze kdbg jak pokazano poniżej.



Ekran 8-15 Program `skok_war1.s` demonstracji w kdbg skoku warunkowego

8.12 Zadania

8.12.1 Mnożenie dwóch liczb

Wzorujac się na poprzednim przykładzie z dodawania liczb napisz program mnożenia dwóch liczb. Wynik zaobserwuj na debuggerze `kdbg`.

8.12.2 Skok warunkowy

Wzorujac się na poprzednim przykładzie dotyczącym skoku warunkowego zmień wartość zmiennych arg1 = 4, arg2 = 3 . Wynik zaobserwuj na debuggerze kdbg.

8.12.3 Szukanie minimum w tablicy

Wzorując się na podanym wcześniej przykładzie szukania maksimum w tablicy (Przykład 8-1) napisz program szukania minimum w tablicy (końcowe zero nie wchodzi w zakres poszukiwań).

```
tablica: .long 63,67,34,224,45,75,54,34,44,33,22,11,66,0
```

Posłuż się debuggerem kdbg do prześledzenia przebiegu programu.

8.12.4 Sumowanie elementów tablicy

Wzorując się na podanym wcześniej przykładzie szukania maksimum w tablicy (Przykład 8-1) napisz program sumowania elementów tablicy

```
tablica: .long 63,67,34,224,45,75,54,34,44,33,22,11,66,0
```

Wynik sumowania ma być przekazany jako kod powrotu z programu. Należy go umieścić w rejestrze %ebx i wykonać wywołanie nr 1 – zakończenie programu podobnie jak poniżej.

```
movl $1, %eax # Numer wywolania syst w EAX
# Liczba 1 to numer wywolania systemowego – zakoncz proces
movl $0, %ebx # Kod powrotu w EBX
# testowanie przez: echo $?
int $0x80 # Wywolanie systemu operacyjnego
```

Posłuż się debuggerem `kdbg` do prześledzenia przebiegu programu.

8.12.5 Sumowanie dwóch tablic

Wzorując się na podanym wcześniej przykładzie szukania maksimum w tablicy (Przykład 8-1) napisz program sumowania dwóch tablic

```
tablica1: .long 63,67,34,224,45,75,54,34,44,33,22,11,66,0
tablica2: .long  1, 2, 3,  4, 5, 6, 7, 4, 4, 3, 2, 1, 6,0
```

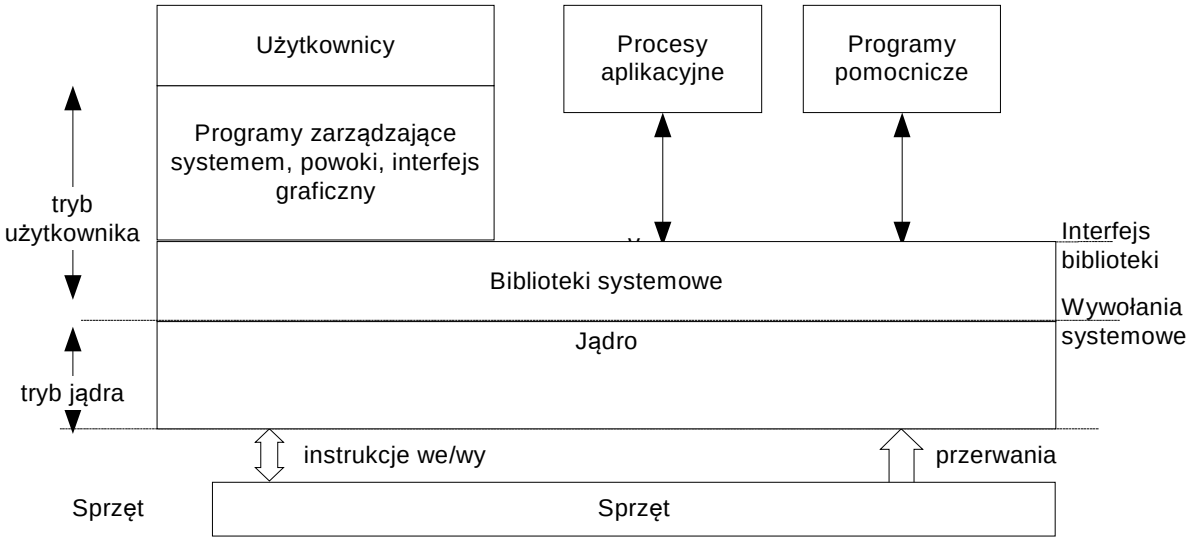
Wynik ma być w tablica2 znaczy to że `tablica2[i] = tablica1[i] + tablica[i} dla i=0,1,...,12`

Posłuż się debuggerem `kdbg` do prześledzenia przebiegu programu. Wyniki pokaż w oknie data/memory.

9. Wywołania systemowe

9.1 Funkcja wywołań systemowych

Programy aplikacyjne wykonują się w środowisku systemu operacyjnego. Mają one postać procesów. Proces jest wykonującym się programem korzystającym z wirtualnego procesora, wirtualnej pamięci i wirtualnych urządzeń wejścia wyjścia. Proces aplikacyjny może się komunikować z systemem operacyjnym za pomocą tak zwanych wywołań systemowych co pokazuje poniższy rysunek.



Rys. 9-1 Ogólny schemat systemu Linux

Wywołania systemowe są sposobem komunikacji pomiędzy procesem aplikacyjnym a systemem operacyjnym. Proces aplikacyjny może żądać od systemu wykonania pewnych usług, na przykład instrukcji wejścia/wyjścia. Przykładami usług systemu operacyjnego może być wyprowadzanie danych na konsolę, dostęp do plików, wykonywanie usług komunikacji sieciowej czy dostęp do interfejsu graficznego. Przykładem wywołania systemowego jest wykonanie funkcji `exit(nr)` która powoduje zakończenie procesu i przekazanie do systemu operacyjnego kodu powrotu `nr`. Przykład użycia funkcji `exit` podany jest poniżej.

```
#include <stdlib.h>
int main(void) {
   exit(5);
}
```

Przykład 9-1 Program exit.c - użycie funkcji exit

Powyższy program napisany jest w języku C i używa funkcji bibliotecznej `exit` która jest opakowaniem wywołania systemowego. Powyższy program można skompilować z opcją –S i zobaczyć jak wywoływana jest funkcja `exit`.

```
$gcc exit.c –o exit.s -S
```

Wywołanie systemowe wykonywane jest za pomocą przerwania programowego `INT 80`. Poszczególne funkcje wykonywane jako wywołania systemowe są ponumerowane. Numerom odpowiadają poszczególne funkcje. Zestawienie wywołań systemowych można znaleźć w pliku `/usr/include/i386-linux-gnu/asm/unistd_32.h`

```
#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
…
```

Przykład 9-2 Fragment pliku nagłówkowego `/usr/include/i386-linux-gnu/asm/unistd_32.h`

Wywołanie systemowe w architekturze IA-32 przebiega w następujący sposób:

1. W rejestrze EAX umieszcza się numer wywołania systemowego
2. W innych rejestrach umieszcza się pozostałe parametry wywołania
3. Wykonuje się przerwanie programowe INT80

Przykład wykonania wywołania systemowego o numerze 1 `exit` podany jest poniżej.

```
#PURPOSE: Simple program that exits and returns a
# status code back to the Linux kernel
#
#OUTPUT: returns a status code. This can be viewed
# by typing
#
# echo $?
#
# after running the program
#
#VARIABLES:
# %eax holds the system call number
# %ebx holds the return status
#
.section .data
.section .text
.globl _start
_start:
movl $1, %eax # this is the linux kernel command
# number (system call) for exiting a program
movl $0, %ebx # this is the status number we will
# return to the operating system.
# Change this around and it will
# return different things to
# echo $?
int $0x80 # this wakes up the kernel to run
# the exit command
```

Przykład 9-3 Wykonanie wywołania systemowego `exit(nr)` – zakończenie procesu

9.2 Niektóre wywołania systemowe

W programowaniu powszechnie wykorzystywana jest abstrakcja pliku. Plik jest abstrakcyjnym urządzeniem pamięciowym do którego można pisać bajty i bajty z niego czytać. Podstawowe funkcje języka C dotyczące plików opisane są w [3] i dane są poniżej.

`int open(char *path,int oflag,[mode_t mode])`

`path`	Nazwa pliku lub urządzenia
`oflag`	Tryb dostępu do pliku – składa się z bitów – opis w pliku nagłówkowym <fcntl.h>
`mode`	Atrybuty tworzonego pliku (prawa dostepu)

`int read(int fdes, void *bufor, int nbytes)`

`fdes` Uchwyt do pliku zwracany przez funkcję open
`bufor` Bufor w którym umieszczane są przeczytane bajty
`nbytes` Liczba bajtów którą chcemy przeczytać.

`int write(int fdes, void *bufor, int nbytes)`

`fdes` Uchwyt do pliku zwracany przez funkcję open
`bufor` Bufor w którym umieszczane są bajty przeznaczone do zapisu
`nbytes` Liczba bajtów którą chcemy zapisać

`int close(int fdes)`

`fdes` Uchwyt do pliku zwracany przez funkcję open

Powyższe funkcje stanowią opakowanie wywołań systemowych co pokazuje poniższa tabela.

Nazwa	%eax	%ebx	%ecx	%edx	Uwagi
exit	1	Status	-	-	Zakończenie procesu
fork	2	-	-	-	Utworzenie procesu
read	3	Uchwyt pliku	Adres bufora	Długość bufora	Odczyt z pliku
write	4	Uchwyt pliku	Adres bufora	Długość bufora	Zapis do pliku
open	5	Nazwa pliku	Flagi	Prawa dostępu	Otwarcie pliku. Uchwyt zwracany w %eax
close	6	Uchwyt pliku	-	-	Zamknięcie pliku

Tab. 9-1 Niektóre wywołania systemowe i ich parametry

Wynik wywołania systemowego lub kod błędu znajduje się w rejestrze %eax.

9.3 Standardowe wejście wyjście

Spośród plików wyróżnia się trzy pliki szczególne:

Nazwa symboliczna	Numer	Opis
STDIN	0	Standardowe wejście - klawiatura
STDOUT	1	Standardowe wyjście - konsola
STDERR	2	Standardowe wyjście błędów - konsola

Tab. 9-2 Pliki standardowego wejścia wyjścia

Pliki standardowego we/wy nie wymagają otwierania i można z nich korzystać w każdej chwili co pokazuje poniższy program.

```
# demonstracja pisania i czytania na/z standardowego wyjścia/wejścia
# przy uzyciu wywolan systemowych
# kompilacja: as read_write.s --32 -o read_write.o -g
# laczenie:   ld read_write.o -m elf_i386 -o read_write
# obserwacja w kdbg

SYSEXIT  = 1
SYSREAD  = 3
SYSWRITE = 4
SYSOPEN  = 5
STDIN = 0
STDOUT = 1
EXIT_SUCCESS = 0
.data
msg_echo: .ascii "                                                "
msg_echo_len = . - msg_echo

newline: .ascii "\n"
newline_len = . - newline
```

```
msg_hello: .ascii "Hello, world!\n"
msg_hello_len = . - msg_hello

.text
.global _start                # wskazanie punktu wejścia do programu
_start:
# piszemy msg_hello
mov $SYSWRITE, %eax           # funkcja do wywołania - SYSWRITE
mov $STDOUT, %ebx             # 1 arg. - syst. deskryptor stdout
mov $msg_hello, %ecx          # 2 arg. - adres początkowy napisu
mov $msg_hello_len, %edx      # 3 arg. - długość łańcucha
int $0x80               # wywołanie przerwania programowego -
                        # wykonanie funcji systemowej.
# czytamy string do msg_echo
movl $SYSREAD, %eax           # funkcja do wywołania - SYSREAD
movl $STDIN, %ebx             # 1 arg. - syst. deskryptor stdin
movl $msg_echo, %ecx          # 2 arg. - adres początkowy napisu
movl $msg_echo_len, %edx      # 3 arg. - długość łańcucha
int $0x80
# w %eax będzie dlugosc lancucha

# wypisujemy na STDOUT msg_echo
movl $SYSWRITE, %eax
movl $STDOUT, %ebx
movl $msg_echo, %ecx          # 2 arg. - adres początkowy napisu
movl $msg_echo_len, %edx      # 3 arg. - długość łańcucha
int $0x80

# wypisujemy na STDOUT znak \n
movl $SYSWRITE, %eax
movl $STDOUT, %ebx
movl $newline, %ecx           # 2 arg. - adres początkowy napisu
movl $newline_len, %edx       # 3 arg. - długość łańcucha
int $0x80

# zakonczenie programu
movl $SYSEXIT, %eax           # funkcja do wywołania - SYSEXIT
movl $EXIT_SUCCESS, %ebx      # 1 arg. -- kod wyjścia z programu
int $0x80                     # wywołanie przerwania programowego -
                              # wykonanie funcji systemowej.
```

Przykład 9-4 Program `read_write.s` wykorzystujący standardowe wejście i wyjście

W powyższym programie należy zauważyć wykorzystanie buforów. Bufory umieszczone są w segmencie `.data` Bufor ma swoją nazwę w postaci etykiety. Jego długość obliczana jest przez wyrażenie wykorzystujące kropkę . symbolizującą adres bieżący.

```
msg_hello: .ascii "Hello, world!\n"
msg_hello_len = . - msg_hello
```

Powyższe bufory są umieszczone w segmencie `.data` i zainicjowane. Mogą być również umieszczone w segmencie danych niezainicjowanych `.bss` , jak pokazano poniżej.

```
.section .bss
.lcomm my_buffer, 512
buf_size = 512
```

Kompilator tworzy w segmencie `.bss` bufor o nazwie `my_buffer` i rezerwuje na niego 512 bajtów. Wykorzystanie bufora dane jest poniżej.

```
movl $my_buffer, %ecx
movl $buf_size, %edx
movl 3, %eax
```

```
int $0x80
```

Program czyta do bufora 512 bajtów. Należy zauważyć że jako adres bufora podano `$my_buffer` , symbol poprzedzony znakiem dolara co wskazuje na adresowanie natychmiastowe. Do rejestru `%ecx` ładowany jest adres początku obszaru `my_buffer`.

9.4 Operacje na bitach w języku C

W języku C zdefiniowane są operacje na bitach. Operacje dwuargumentowe:

- koniunkcja bitowa ("&"),
- alternatywa bitowa ("|") i
- alternatywa rozłączna XOR (”^”)

Operacje jednoargumentowe:

- negacja bitowa ("~"),
- przesunięcie bitowe

bit a	bit b	a&b
0	0	0
0	1	0
1	0	0
1	1	1

bitowe AND &

bit a	bit b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

bitowe OR |

bit a	bit b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

bitowe XOR ^

Przykład bitowego AND

```
  11001110
 &10011000
 = 10001000
```

Przykład bitowego OR

```
  11001110
 |10011000
 =11011110
```

Przykład bitowego XOR

```
  11001110
 ^10011000
 =01010110
```

bit a	~a
0	1
1	0

Negacja bitowa

```
~11001110
 00110001
```

Przesunięcie w prawo >>
Przesuniecie w prawo [zmienna] >> [liczba miejsc] jest operacją dwuargumentową A >> n. Przesuwa ona bity operandu A o n pozycji w prawo. Na miejsce najstarszych bitów wchodzi 0
Gdy x=11100101 to x >> 1 daje w wyniku 01110010, x >> 2 daje w wyniku 00111001

Przesunięcie w lewo <<
Przesuniecie w lewo [zmienna] << [liczba miejsc] jest operacją dwuargumentową A << n. Przesuwa ona bity operandu A o n pozycji w lewo. Na miejsce najmłodszych bitów wchodzi 0
Gdy x=11100101 to x << 1 daje w wyniku 11001010, x << 2 daje w wyniku 10010100

```
#include <stdio.h>
 void pokaz_bity(unsigned int x){
        int i;
        for(i=(sizeof(int)*8)-1; i>=0; i--)
            (x&(1<<i))?putchar('1'):putchar('0');
        printf("\n");
 }

 int main(int argc,char *argv[]) {
        int j, m, n;
        j = atoi(argv[1]);
        printf("Dziesietne: %d hex: %08X binarnie - ",j, j);
        pokaz_bity(j);
        return 0;
 }
```

Przykład 9-1 Wypisywanie liczby w postaci HEX i binarnej

9.5 Zadania

9.5.1 Zamiana liter na duże

Program `read_write.s` pokazuje jak wczytywać łańcuch z standardowego wejścia i jak wyprowadzać napis na standardowe wyjście. Na jego podstawie napisz program który we wprowadzonym z klawiatury tekscie zamienia małe litery na duże pozostawiając duże bez zniany. Wykorzystaj poniszą tabelę kodów ASCII.

znak	dec	hex	bin
A	65	41	01000001
B	66	42	01000010
...			
Z	90	5a	01011010
...			
a	97	61	01100001
b	98	62	01100010
...			
z	122	7a	01111010

Przesunięcie `shift` pomiędzy kodem litery małe `‘a’` a duże `’A’` wynosi `shift = (int) ‘a’ – ‘A’= 32`. Zauważ też że kod małej litery może być uzyskany z kodu dużej litery poprzez wstawienia 1 na pozycji 5 czyli OR z 00100000. Poniżej dany jest kod w języku C realizujący tę zamianę.

```
// Zamiana malych liter na duże, lancuch wczytywany ze stdin
// kompilacja gcc zamiana_na_duze.c -o zamiana_na_duze
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#define STDIN  0
#define STDOUT 1
#define BSIZE 80
#define SHIFT 32 // rozniaca pomiedzy ord('a') i ord('A')

char bufor[] = "Podaj napis\n";
int main() {
      int ile,i;
      char c;
      write(STDOUT,bufor,strlen(bufor));
      // Odczyt napisu ze stdin
      ile = read(STDIN,bufor,BSIZE);
```

```
      printf("Bufor:%s znakow: %d\n",bufor,ile);
      for(i=0;i<ile;i++) {
        c = bufor[i];
        if( (c <= 'z') && (c >= 'a')) {
             c = c - SHIFT;
             bufor[i] = c;
        }
    }
    write(1,bufor,strlen(bufor));
    exit(ile);
}
```

Przykład 9-2 Program `zamien_na_duze.c` zamiany małych liter na duże we wprowadzanym z konsoli napisie

9.5.2 Szyfr Cezara

Szyfr Cezara jest prostym szyfrem podstawieniowym. Przy szyfrowaniu polega on zastąpieniu kodu danego znaku, kodem otrzymanym z kodu znaku jawnego do którego dodano klucz `key`. Operacja deszyfracji polega na odjęciu od kodu znaku zaszyfrowanego klucza `key`. Należy zabezpieczyć się przed wyjściem poza zakres kodowanych znaków, stąd w przykładzie operacja modulo %. Znaczy to że jak wyjdziemy poza zakres kodów, należy dokonać korekty. Przykład kodowania znaków podano poniżej.

Tekst jawny	a	b	c		y	z
Tekst zaszyfrowany	b	c	d		z	a

Przykład 9-5 Szyfr Cezara przy kluczu 1

Poniżej podano program kodujący w języku C. Funkcja `tolower( c )` zamienia kod znaku `c` na kod małej litery.

```
// Szyfr Cezara ----------------------------
// Szyfrowane sa tylko duze litery A-Z
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define SIZE 256

void main(int argc, char *argv[]) {
    char p[SIZE];
    int key,i,enc;
    printf("szyfr Cezara kodowanie\n");
    printf("Podaj tekst=");
    gets(p);
    printf("podaj klucz=");
    scanf("%d",&key);

    for(i=0;i<strlen(p);i++)
    {
           if((p[i] >= 'A') && (p[i] <= 'Z')) {
             // Dodajemy klucz ----
             enc=p[i]+key;
             // Czy klucz poza zakresem A-Z gdy tak korekcja
             if(enc > 'Z') { enc = enc - ('Z'-'A' + 1); }
             p[i] = enc;
           }
           printf("%c",p[i]);
    }
    printf("\n");
}
```

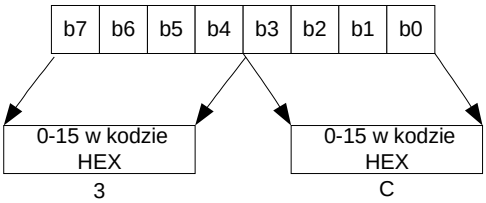
Przykład 9-6 Program kodujący według szyfru Cezara

Napisz w assemblerze program kodowania znaków według szyfru Cezara a następnie program dekodowania. Jako wzór weź program `read_write.s`. W czasie pisania programu uwzględnij następujące kroki.

1. Przyjmij że przesunięcie jest stałe i wynosi 1
2. Najpierw załóż że długość łańcucha jest zadeklarowana jak w programie `read_write.s` o potem wykorzystaj wynik funkcji systemowej `read`.
3. Zaobserwój w `gdb` zawartość wprowadzonej tablicy
4. Napisz program dekodujący

9.5.3 Zamiana kodów znaków ASCII na HEX

Znaki ASCII kodowane są w jednym bajcie. Tylko niektóre mają swoją widoczną reprezentację, powstaje pytanie jak je przedstawić? Można zawartość jednego bajtu – 8 bitów, podzielić na dwie części po 4 bity, i przedstawić je jako kolejne liczby HEX które daje się wypisać za pomocą znaków 0-9 i A-F.



Program powinien wykonać nastepujące kroki:

1. Zadeklarować zmienne: `threeBytes: .ascii "   ", .lcomm oneByte, 1`
2. Za pomocą wywołania systemowego READ wczytać do bufora `oneByte` 1 znak ze `stdin`.
3. Gdy kod powrotu funkcji READ != 1 zakończyć proces.
4. Skopiować z `oneByte` do `%al` odczytany bajt i wyodrębnić 4 młodsze bity.

5. Przekształcić je w zalezności od wartości na znak 0-9 (dodać do `%al` liczbę odpowiadającą znakowi ‘0) lub (dodać do `%al` liczbę odpowiadającą znakowi ‘A) znak A-F i zapisać do bufora `threeBytes` na pozycji 1
6. Skopiować z `oneByte` do `%al` odczytany bajt i wyodrębnić 4 starsze 4 bity.
7. Przekształcić je w zalezności od wartości na znak 0-9 lub A-F i zapisać do bufora threeBytes na pozycji 0
8. Bufor `threeBytes` zapisać na `stdout` za pomocą funkcji WRITE
9. Przejść do kroku 2.

Po kompilacji uruchamiamy program kierując do niego dane z pliku `ascii.txt` a wyniki kierując do pliku `wynik.txt`.

```
./toHex < ascii.txt > wynik.txt
```

Plik `ascii.txt` zawiera kody kolejnych znaków od 0 do 255. Tylko niektóre z nich są widoczne. Kod programu w języku C podany jest poniżej.

```
// Wyprowadzanie na stdout kodów HEX znaków
// wprowadzanych ze STDIN
// kompilacja gcc toHexmyc.c -o toHexmyc
// Uruchomienie: toHexmyc < ascii.txt
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#define STDIN  0
#define STDOUT 1

char threeBytes[3] = "   ";
char oneByte;
int main() {
      int ile;
      char c;
      do {
        ile = read(STDIN,&oneByte,1);
        // Mlodszy kwartet - starszy maskujemy
        c = oneByte & 0x0F;
        c = c + '0';
        if(c > '9') c = c + 'A' -'9' - 1;
        threeBytes[1] = c;
        // Starszy kwartet - przesuwamy
        c = oneByte;
        c = c >> 4;
        c = c & 0x0F;
        c = c + '0';
        if(c > '9') c = c + 'A' -'9' - 1;
        threeBytes[0] = c;
        write(STDOUT,threeBytes,3);
    } while(ile >0);
    exit(0);
}
```

Przykład 9-3 Program `toHexmyc.c` zamiany znaków wczytywanych ze STDIN na ich kody HEX

9.5.4 Ciąg Fibonacciego

W ciągu Fibonacciego n-ty element f(n) dany jest wzorem:

`f(n) = f(n-1) + f(n-2)` dla `n>1` przy czym `f(0)=0 a f(1)=1`

Poniżej dany jest kod w języku C.

```
int main() {
   int n, first = 0, second = 1, next, i;
   printf("Podaj liczbe n\n");
   scanf("%d",&n);
   for ( i = 0 ; i < n ; i++ ){
      if ( i <= 1 ) next = i;
      else {
         next = first + second;
         first = second;
         second = next;
      }
   }
   printf("Wynik: %d\n",next);
   return 0;
}
```

Przykład 9-7 Program w języku C do obliczania ciągu Fibonacciego

Napisz program obliczania liczb fibonacciego w assemblerze dla liczb int.

10. Tablice i wskaźniki

Tablice są mechanizmem integracji danych (np. liczb) w większe struktury. Dostęp do tablic odbywa się poprzez indeksy które można obliczać, przez co przetwarzanie tablic można automatyzować.

10.1 Przykład – obliczanie maksimum tablicy

```
// Kompilacja by uzyskac kod assemblerowy
// gcc maximum.c -S -fno-asynchronous-unwind-tables

#include <stdlib.h>
#define DIM 8

int tab[DIM] = {4,1,6,89,12,6,33,7};

int main() {
 int i;
 int max = 0;
 for(i=0;i<DIM;i++) {
    if(max < tab[i]) max = tab[i];
 }
 return max;
}
```

Przykład 10-1 Przykład użycia tablicy w języku C – program maximum.c

Możemy program skompilować by uzyskać kod assemblerowy jak niżej.

```
$gcc maximum.c -S -fno-asynchronous-unwind-tables
```

```
      .file "maximum.c"
      .globl      tab
      .data
      .align 32
      .type tab, @object
      .size tab, 32
tab:
      .long 4
      .long 1
      ...
      .long 7
      .globl      max
      .bss
      .align 4
      .type max, @object
      .size max, 4
max:
      .zero 4
      .text
      .globl      main
      .type main, @function
main:
      pushl %ebp
      movl  %esp, %ebp
      subl  $16, %esp
      movl  $0, -4(%ebp)
      jmp   .L2
.L4:
      movl  -4(%ebp), %eax
      movl  tab(,%eax,4), %edx
      movl  max, %eax
      cmpl  %eax, %edx
      jle   .L3
      movl  -4(%ebp), %eax
```

```
      movl  tab(,%eax,4), %eax # dostep do elementu tablicy tab
      movl  %eax, max
.L3:
      addl  $1, -4(%ebp)
.L2:
      cmpl  $7, -4(%ebp)
      jle   .L4
      movl  max, %eax
      leave
      ret
      .size main, .-main
      .ident      "GCC: (Debian 6.1.1-11) 6.1.1 20160802"
      .section    .note.GNU-stack,"",@progbits
```

Przykład 10-2 Przykład użycia tablicy – program `maximum.s`

10.2 Dostęp do elementów tablicy

Jeżeli zdefiniujemy typ `T` (typ elementów tworzących tablicę) i stałą całkowitą `N` (wymiar tablicy), tablicę E deklarujemy jak niżej.

```
T E[N];
```

Kompilator rezerwuje ciągły obszar `L*N` bajtów gdzie `L` jest liczbą bajtów zajmowaną przez obiekt T czyli `L=sizeof(T)`. Oznaczymy adres tego obszaru jako x_A. Kompilator wprowadza identyfikator A który jest wskaźnikiem na początek tej tablicy. Elementy tablicy są dostępne przez indeks `i` (liczbę całkowitą) z przedziału od `0` do `N-1`. Elementy tablicy są pamiętane pod adresami:

x_E + L*i

Dla przykładu rozważmy tablice.

```
char        A[12];
char        *B[8];
double      C[6];
double      *D[5];
```

Rozmiary i dostęp do poszczególnych elementów podano poniżej.

Array	Element size	Total size	Start address	Element *i*
A	1	12	x_A	$x_A + i$
B	4	32	x_B	$x_B + 4i$
C	8	48	x_C	$x_C + 8i$
D	4	20	x_D	$x_D + 4i$

Instrukcje architektury IA32 zawierają instrukcje ułatwiające dostęp do tablic. Dla przykładu założymy że E jest tablicą `int` i chcemy obliczyć E`[i]` gdzie adres E jest pamiętany w `%edx` a indeks `i` w `%ecx` a wynik kopiujemy do `%eax`. Instrukcja dana poniżej:

```
movl (%edx,%ecx,4), %eax
```

kopiuje element x_E do rejestru `%eax`. Dopuszczalne współczynniki skali to 1,2,4,8 odpowiadają rozmiarom typom danych.

10.3 Arytmetyka wskaźników

Język C implementuje arytmetykę wskaźników gdzie obliczana wartość zależy od typu danych na które wskazuje wskaźnik. Tak więc wskaźnik zawiera następujące informacje:

- Adres obiektu
- Jego typ, w tym rozmiar

Jeżeli `p` jest wskaźnikiem na obiekt typu T który zajmuje L bajtów (`L=sizeof(T)`) i wartość `p` jest x_p to wyrażenie `p+1` ma wartość x_p `+ L` czyli wskazuje on na następny element. W związku ze wskaźnikami zdefiniowano operacje `& i *`:

Operacja: `p = &Expr` - zwraca adres `p` obiektu `Expr`.

operacja: `*p` - zwraca to na co wskazuje `p` czyli `Expr`

czyli zachodzi równość `Expr = *&Expr`

Dostęp do elementów tablicy A może być realizowany tak przez indeks jak przez wskaźnik. Wartość `A[i]` jest identyczna jak `*(A + i)` – obliczana jest jako element i-ty tablicy.

Nawiązując do poprzedniego przykładu tablicy E zawierającej wartości `int` założymy że adres E jest w `%edx` a indeks `i` w `%ecx` możemy podać poniższe przykłady przesłania wartości elementu i do rejestru `%eax`. Zapis `M[addr]` oznacza zawartość pamięci o adresie `addr`. Należy zauważyć że rozmiar elementu `int` z których zbudowana jest tablica to 4 bajty.

Expression	Type	Value	Assembly code
E	int *	x_E	movl %edx,%eax
E[0]	int	$M[x_E]$	movl (%edx),%eax
E[i]	int	$M[x_E + 4i]$	movl (%edx,%ecx,4),%eax
&E[2]	int *	$x_E + 8$	leal 8(%edx),%eax
E+i-1	int *	$x_E + 4i - 4$	leal -4(%edx,%ecx,4),%eax
*(E+i-3)	int *	$M[x_E + 4i - 12]$	movl -12(%edx,%ecx,4),%eax
&E[i]-E	int	i	movl %ecx,%eax

Przykład 10-3 Różne zapisy dostępu do tablicy `int E[N]`

Poniżej podano program liczenia maksimum w tablicy z wykorzystaniem wskaźników.

```
#include <stdlib.h>
#define DIM 8

 int * tab;
 int max = 0;

int main() {
 int i;
 tab = malloc(DIM*4);
 *(tab + 0) = 4;
 *(tab + 1) = 1;
 *(tab + 2) = 6;
 *(tab + 3) = 89;
 *(tab + 4) = 12;
 *(tab + 5) = 6;
 *(tab + 6) = 33;
 *(tab + 7) = 7;
 for(i=0;i<DIM;i++) {
    if(max < *(tab+i)) max = *(tab + i);
 }
 return max;
}
```

Przykład 10-4 Program `maximum-p.c` obliczania maksimum z użyciem wskaźników

10.4 Przykład – obliczanie maksimum w tablicy, assembler

Poniżej zamieszczony jest kod programu maksimum.s szukania maksimum liczb w tablicy.

```
#znajdowanie maksimum w tablicy tablica
# rejestr %edi  - indeks tablicy
# rejestr %ebx  - aktualnie najwiekszy element
# rejestr %ebx  - biezacy element
.section .data
 tablica: #To są dane w tablicy zawierającej liczby long
.long 3,67,34,224,45,75,54,34,44,33,22,11,66,0
.section .text
.globl _start
_start:
movl $0, %edi # przeslij 0 do rejestru indeksowego
movl tablica(,%edi,4), %eax # przeslij do eax kolejna liczbe z tablicy
movl %eax, %ebx  # pierwszy element jest na razie najwiekszy
start_loop: # start petli
cmpl $0, %eax # sprawdz czy koniec tablicy (jest tam 0)
je loop_exit
incl %edi # zwieksz licznik
movl tablica(,%edi,4), %eax
cmpl %ebx, %eax # porownaj czy %eax  wiekszy od maksimum
jle start_loop  # skocz do poczatku petli gdyz %eax jest mniejszy od max
movl %eax, %ebx # w %eax jest aktualne maksimum
jmp start_loop  # skocz na oczatek petli
loop_exit:
# %ebx zawiera kod powrotu funkcji exit i jest to maksimum
movl $1, %eax #1 jest kodem wywolania exit()
int $0x80
```

Przykład 10-5 Obliczanie maksimum tablicy

10.5 Zadania

10.5.1 Sumowanie liczb z tablicy

Dany poprzednio program `maksimum.s` pokazuje jak przetwarzać elementy tablicy zdefiniowanej w segmencie danych. Na jego podstawie napisz program który oblicza sumę liczb. Wyświetl wyniki używając wywołania systemowego WRITE.

10.5.2 Szukanie liczb pierwszych metodą sita Eratostenesa

Prawdopodobnie najstarszy algorytm szukania liczb pierwszych został przypisany Eratostenesowi z Cyreny. Żył on w latach od 276 pne. do 194 pne. Algorytm opisano go między innymi w Wkipedii https://pl.wikipedia.org/wiki/Sito_Eratostenesa. Algorytm w podanej niżej implementacji polega na utworzeniu tablicy typu `int primes[limit]` zawierającej liczby 1. Następnie począwszy od liczby 2 usuwane są kolejne wielokrotności tej liczby przez wpis 0 na pozycji 2*2, 2*4, 2*8,.... Podobnie postepujemy z kolejnymi liczbami. Indeksy tablicy `primes` tam gdzie jest jeden, są indeksami liczb pierwszych. Poniżej podano algorytm w języku C.

```
// Szukanie liczb pierwszych metoda sita Erastotenesa
// gcc primesc2.c -o primesc2
#include <stdio.h>

int main() {
 int numbertable[1000];
 int i;
 int number;
 int multiple;
 //Inicjalizacja tablicy numbertable
 for(i = 0; i < 1000; i++ ) numbertable[i] = 1;

 //algorytm sita
 for(number = 2; number < 1000; number++ )  {
      // Jesli liczba jest na liscie to jest pierwsza
      if ( numbertable[number] ) {
       // Wypisz liczbe pierwsza
       printf("%d ", number );
       // Pomin wszystkie wielokrotnosci liczby number
       multiple = 2 * number;
       while ( multiple < 1000 ) {
                numbertable[multiple] = 0;
                multiple += number;
       }
      }
  }
 return 0;
}
```

Przykład 10-6 Program `primesc2.c` szukania liczb pierwszych metodą sita Eratostenesa

Wyniki działania programu podano poniżej

```
$./primesc2
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101
103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197
199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311
313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431
433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557
563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661
673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809
811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937
941 947 953 967 971 977 983 991 997
```

Napisz program znajdowania liczb pierwszych metodą sita Eratostenesa. Można wykorzystać fragment danego niżej programu który inicjalizuje zmienne.

```
# Program znajduje liczby pierwsze metoda sita Eratostenesa
# Kompilacja:
# as primesa2.s -o primesa2.o
# gcc primesa2.o -o primesa
#
.bss
# NUMBERS:    .skip 1000       # Pamiec na tablice numbers
.lcomm NUMBERS, 1000           # Pamiec na tablice numbers

.text
formatstr:  .asciz "%d "           # format string for number printing
komunikat:  .asciz "Start\n"
.global main

# ************************************************************************
# Funkcja main - tu zaczyna sie program
main:   movl    %esp, %ebp      # initialize the base pointer

pushl   $komunikat          # push the format string for printing
call    printf              # print the number
addl    $4, %esp            # pop the format string

# Inicjalizacja tablicy NUMBERS
movl    $0, %eax            # initialize 'i' to 0.
loop1:  movb    $1, NUMBERS(%eax)   # set number table entry 'i' to 'true'
incl    %eax                # increment 'i'
cmpl    $1000, %eax         # while 'i' < 1000
jl      loop1               # go to start of loop1

# Tutaj ma byc program

call    exit                # exit the program
```

Przykład 10-7 Fragment programu `primesa2.s` szukania liczb pierwszych metodą sita Eratostenesa

11. Makroinstrukcje

Powtarzające się sekwencje programu, które różnią się tylko parametrami można zapisać w postaci makroinstrukcji. Makroinstrukcja może być wywołana, z różnymi parametrami poprzez tak zwane makrowywołanie. W czasie wywołania makroinstrukcji podawane argumenty które są aktualizowane. Makroinstrukcja rozpoczyna się dyrektywą `.macro` po której następuje jej nazwa i lista argumenty a kończy się dyrektywą `.endm`.

```
.macro nazwa arg1, arg2,
instrukcje
...
.endm
```

Występujące wewnątrz makroinstrukcji argumenty poprzedzone są znakiem ukośnika \ i występują jako `\arg1, \arg2,...`

Poniżej podany jest przykład utworzenia makra write.

```
# Numbers of kernel functions.
EXIT_NR  = 1
READ_NR  = 3
WRITE_NR = 4

STDOUT = 1
EXIT_CODE_SUCCESS = 0

.text
msg: .ascii "Hello, world!\n"
msgLen = . - msg

.global _start
_start:

mov $WRITE_NR, %eax
mov $STDOUT  , %ebx
mov $msg     , %ecx
mov $msgLen  , %edx
int $0x80

mov $EXIT_NR          , %eax
mov $EXIT_CODE_SUCCESS, %ebx
int $0x80
```

Przykład 11-1 Program `hello_m.s` piszący „Hello world”

```
# Program hello1_m, ilustracja budowy i dzialania makr
# kompilacja: as hello1_m.s --32 -o hello1_m.o -g
# laczenie:   ld hello1_m.o -m elf_i386 -o hello1_m
EXIT_NR  = 1
READ_NR  = 3
WRITE_NR = 4

STDOUT = 1
EXIT_CODE_SUCCESS = 0

.text
msg: .ascii "Hello, world!\n"
msgLen = . - msg

.global _start
_start:
# wypisz string na stdout
# arg1 - adres lancucha
# arg2 - dlugosc lancucha
.macro   write    str,  str_size
 mov $WRITE_NR, %eax    # funkcja do wywołania - SYSWRITE w %eax
 mov $STDOUT, %ebx      # 1 arg. - syst. deskryptor stdout
 mov \str, %ecx         # 2 arg. - adres początkowy napisu
 mov \str_size, %edx    # 3 arg. - długość łańcucha
 int $0x80        # wykonanie funcji systemowej.
.endm

# piszemy msg_hello
# wywolanie makra write
write $msg, $msgLen

mov $EXIT_NR          , %eax
mov $EXIT_CODE_SUCCESS, %ebx
int $0x80
```

Przykład 11-2 Program `hello1_m.s` piszący „Hello world” z wykorzystaniem makra

Przykład wywołania w celu wypisania łańcucha `msg_hello`

```
write $msg_hello, $msg_hello_len
```

11.1 Zadania

11.1.1 Makra w wywołaniach systemowych

Zmodyfikuj program `read_write.s` z rozdziału 9 wprowadzając makra `czytaj, pisz, exit` do czytania łańcuchów, pisania łańcuchów i zakończenia procesu.

12. Funkcje

12.1 Elementy funkcji

Funkcje są wydzielonymi fragmentami kodu, które mogą być niezależnie rozwijane i testowane. Program dzieli się na funkcje by nadać mu strukturę, wyeliminować powtarzające się fragmenty kodu i ułatwić uruchamianie. Funkcje mogą mieć parametry, czyli przekazywane jej wartości od których uzależnione jest jej działanie. Funkcja operuje na kilku strukturach które będą teraz wymienione.

Nazwa funkcji – symboliczne oznaczenie początkowego adresu kodu gdzie funkcja się rozpoczyna.

Parametry funkcji – jednostki danych przekazywane do funkcji od których uzależnione jest jej działanie.

Zmienne lokalne – zmienne pomocnicze używane przez funkcję. Miejsce na nie jest rezerwowane (na stosie) gdy funkcja jest wywoływana. Gdy funkcja się kończy, pamięć używana przez zmienne lokalne jest zwalniana.

Zmienne statyczne – zmienne pomocnicze które są dostępne pomiędzy kolejnymi wywołaniami funkcji. Nie są one dostępne dla innych jednostek programu

Zmienne globalne – zmienne zdefiniowane na zewnątrz funkcji

Adres powrotu – adres instrukcji od której należy wznowić program gdy funkcja się zakończy. Jest to potrzebne gdyż funkcja może być wywoływana z różnych miejsc programu. Adres powrotu jest składowany na stosie automatycznie przy wykonaniu funkcji `call`. Adres powrotu jest kopiowany do licznika instrukcji przy wykonaniu rozkazu `ret`.

Zwracana wartość – jest to metoda przekazywania wyniku działania funkcji do funkcji wywołującej.

Sposób przekazywania parametrów do funkcji i pobieranie wyników jest nazywane konwencją wywoływania (ang. *calling conwention*). Są one specyficzne dla języków programowania. Dla języka C opisano je w rozdziale 3 dokumentacji Intel® Architecture Software Developers Manual Volume 1: Basic Architecture. Należy się z tym rozdziałem zapoznać.

12.2 Stos

Operowanie na funkcjach intensywnie wykorzystuje stos programowy. Podstawowe instrukcje operujące na stosie podane są poniżej.

`popl`	`R/M`	`O/S/Z/A/C`

Ściąga słowo ze stosu do lokacji określonej przez R/M. Jest równoważna instrukcji:
`movl(%esp), R/M`
po której następuje:
`addl $4, %esp`
czyli zwiększenie wskaźnika stosu o 4.		

`pushl`	`R/M`	`O/S/Z/A/C`
Przesyła słowo określone przez R/M na stos. Jest równoważna instrukcji: `subl $4, %esp` (zmniejszenie wskaźnika stosu o 4) po której następuje `movl I/R/M, (%esp).`		

Instrukcja `pushl %eax` przesyła na wierzchołek stosu zawartość rejestru `%eax`. Jest ona równoważna instrukcjom:

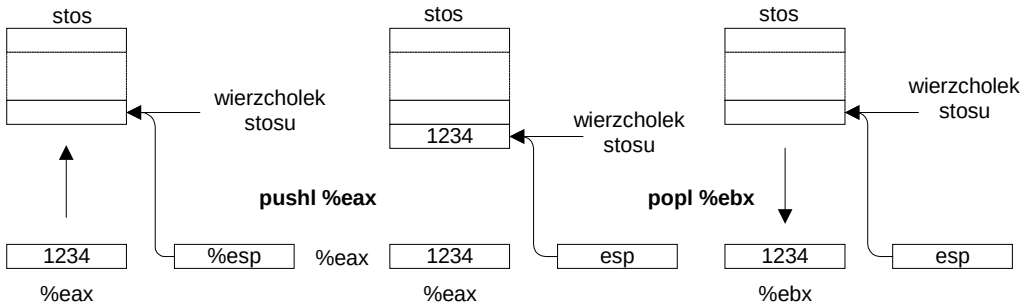
```
subl $4, %esp
movl %eax,(%esp)
```

Pierwsza instrukcja zmniejsza wskaźnik stosu `%esp` o 4 (gdyż będziemy umieszczać na stosie słowo 4 bajtowe). Druga przesyła do komórki wskazywanej przez adres zawarty w rejestrze `%esp` (adresowanie indeksowe) zawartość rejestru `%eax`. Ważny jest tu nawias obejmujący rejestr `%esp` co wskazuje na adresowanie indeksowane.

Instrukcja `popl %ebx` pobiera 4 bajty z wierzchołka stosu i przesyła je do rejestru `%ebx`. Jest ona równoważna instrukcjom:

```
movl(%esp), %ebx
addl $4, %esp
```

Działania te pokazane zostały poniżej.



Rys. 12-1 Ilustracja działania instrukcji `pushl %eax` i `popl %ebx` operujących na stosie

12.3 Wywoływanie funkcji

Instrukcje związane z wykonywanie funkcji podano poniżej.

Instruction		Description
`call`	*Label*	Procedure call
`call`	**Operand*	Procedure call
`leave`		Prepare stack for return
`ret`		Return from call

Tab. 12-1 Instrukcje związane z wywoływaniem funkcji

Powtarzalne fragmenty kodu umieszcza się w funkcjach. Funkcja zazwyczaj posiada argumenty. Funkcja operuje na swych argumentach i ewentualnie na zmiennych globalnych i kończy się rozkazem `ret`. Aby wywołać fukcję należy:

- Skopiować na stos jej parametry w odwrotnej kolejności
- Wykonać instrukcję `call adres_funkcji`

Instrukcja `call adres_funkcji` powoduje skopiowanie na stos adresu instrukcji następującej bezpośrednio po `call` (jest to adres powrotu z funkcji) a następnie załadowanie do rejestru instrukcji `%eip` adresu nowej funkcji (`adres_funkcji`)bedącej argumentem wywołania. Spowoduje to rozpoczęcie wykonywania kodu nowej funkcji. Ostatnia instrukcja funkcji czyli `ret` powoduje skopiowanie do licznika rozkazów ze stosu adresu powrotu i wznowienie wykonywania kodu czyli instrukcji następnej po `call`.

Instrukcja	Operandy	Flagi
`call`	`Adres docelowy`	`O/S/Z/A/C`
Składuje na stosie adres następnej instrukcji wskazywanej przez licznik rozkazów `%eip` i wykonuje skok do instrukcji wskazanej w operandzie. Do określenia adresu skoku można też użyć poprzedzonego gwiazdką nazwy rejestru. Np. `call *%eax` spowoduje skok do adresu określonego w `%eax.`		
`ret`		`O/S/Z/A/C`
Powrót z funkcji. Zdejmuje ze stosu wartość adresu powrotu i przesyła ją do licznika rozkazów `%eip`.		

Stos jest intensywnie wykorzystywany w języku C do organizacji wykonywania funkcji. Pełni on nastepujące funkcje:

- Przekazanie z funkcji wywołującej do wywoływanej parametrów funkcji
- Przechowanie adresu powrotu z funkcji
- Zapewnienie miejsca na zmienne lokalne funkcji

Istnieje dobrze ugruntowana konwencja tworzenia funkcji w języku C (a także w innych językach) która będzie dalej opisana. Przed wywołaniem funkcji, należy przesłać na stos argumenty funkcji w odwrotnej kolejności względem występowania na liście argumentów.

```
Parametr #N
...
Parametr 2
Parametr 1
Adres powrotu <--- (%esp)
```

Rys. 12-2 Obraz stosu po wykonaniu rozkazu `call`

Przy wywoływaniu funkcji obowiązują następujące reguły:

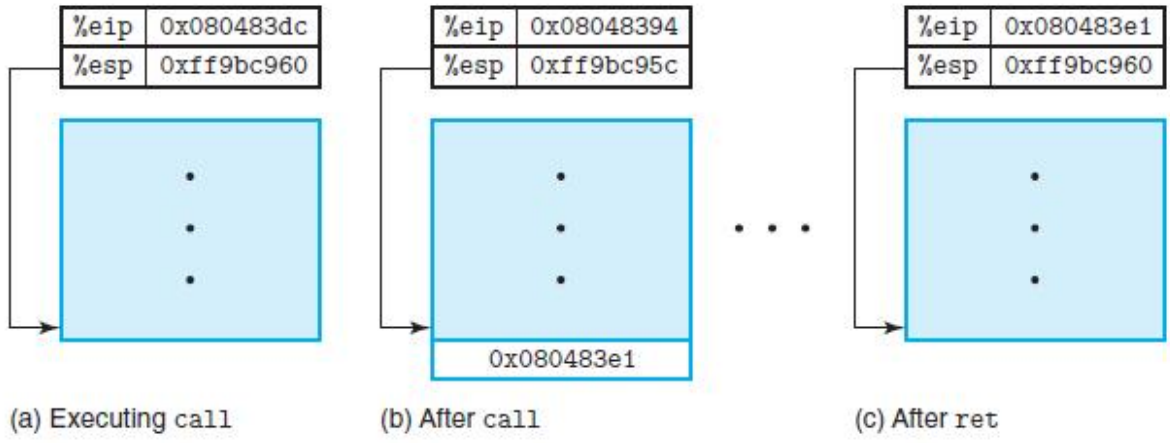
- Początek stosu jest wyrównany co do wielokrotności słowa 4 bajtowego (ang. Word Alligned)
- Funkcja wywołująca (ang. *Caller*) umieszcza na stosie argumenty w odwrotnej kolejności niż występują na liście argumentów funkcji.
- Wielkość argumentu ma być zwiększona do 4 bajtów gdy jest ona mniejsza.

Zakłada się że rejestry `%ebp`, `%ebx`, `%edi`, `%esi`, `%esp` należą do funkcji wywołującej i powinny być zachowane przez funkcję wywoływaną. W standardzie wywoływania funkcji dla języka C pewne rejestry pełnią ustaloną rolę:

`%esp`	Wskaźnik stosu, wyznacza dolną granicę stosu
`%ebp`	Wskaźnik bieżącej ramki stosu. Przechowuje wartość wskaźnika stosu z chwili tuż po wejściu do nowej funkcji. Nie dzieje się to jednak automatycznie. Zawartość wskaźnika stosu kopiowana jest do `%ebp` czyli `movl %esp, %ebp` w ramach prologu wykonywanego w funkcji wywoływanej. Względem tego wskaźnika można adresować argumenty `%ebp + 8, %ebp + 12,...` Bądź zmienne lokalne nowej funkcji: `%ebp – 4, %ebp – 8,...` Rejestr `%ebp` musi być przechowany przez funkcję wywołującą.
`%eax`	Przechowuje wartość int zwracaną przez funkcję. Gdy funkcja zwraca strukturę, unię lub inną złożoną strukturę jest to adres tej struktury

Tab. 12-2 Rejestry związanie z wywoływaniem funkcji

Wywołanie funkcji `call adres_funkcji` powoduje skopiowanie na stos adresu następnej instrukcji po `call`, a następnie przesłanie do licznika rozkazów `%eip` adresu nowej funkcji.



Rys. 12-3 Wywołanie funkcji i powrót z funkcji

Pierwszą rzeczą którą robi nowa funkcja to przesłanie na stos zawartości rejestru %ebp czyli wykonanie instrukcji `pushl %ebp`. Następnie kopiuje się do rejestru %ebp wskaźnik stosu %esp przez wykonanie instrukcji `movl %esp, %ebp.` Rejestr %ebp nazywany jest wskaźnikiem ramki stosu i pełni on ważną rolę. Pozwala on na dostęp do parametrów funkcji i do zmiennych lokalnych funkcji jako indeksów względem rejestru %ebp.

```
call adres_funkcji
...
# w nowej funkcji
pushl %ebp
movl %esp, %ebp

...
```

Obraz stosu po wykonaiu tego kodu pokazano poniżej.

```
Zawartość stosu   Adres względem %ebp
---------------------------------------
Parametr #N       <--- N*4+4(%ebp)
...
Parametr 2        <--- 12(%ebp)
Parametr 1        <--- 8(%ebp)
Adres powrotu     <--- 4(%ebp)
Stary %ebp        <--- (%esp) i (%ebp)
```

Rys. 12-4 Obraz stosu po wykonaniu rozkazu `call i pushl %ebp` oraz `movl %esp, %ebp`

Dla przykładu rozważymy wywołanie funkcji:

```
printf("The number is %d", 88);
```

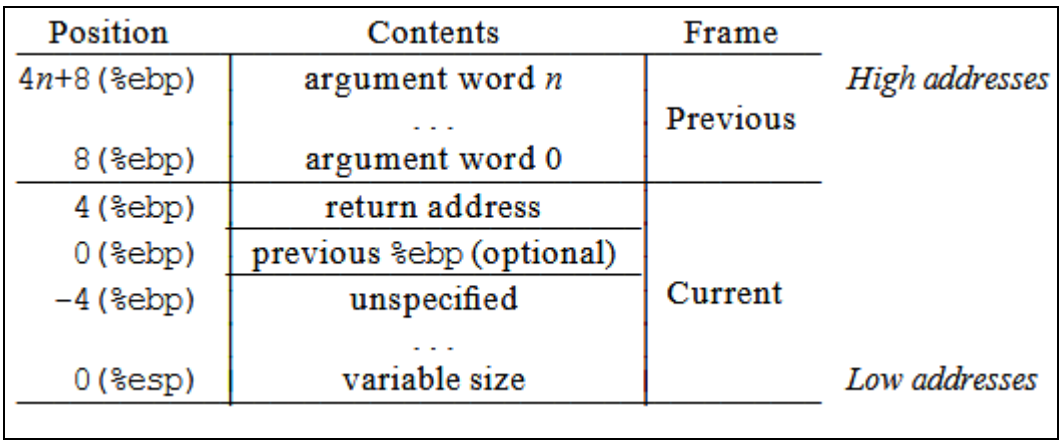
W assemblerze wywołanie wygląda jak poniżej.

```
.section .data
text_string:
.ascii "The number is %d\0"
.section .text
pushl $88
pushl $text_string
call printf
popl %eax
popl %eax #%eax jest rejestrem nie wykorzystanym
```

Przykład 12-1 Wywołanie sekwencji `printf("The number is %d", 88);`

12.4 Ramka stosu

Fragment stosu zaalokowany dla określonej funkcji nazywa się ramką stosu tej funkcji (ang. *stack frame*). Każdy parametr funkcji może być odczytany jako indeksowany względem `%ebp` jak pokazuje poniższy diagram.



Rys. 12-5 Umieszczanie argumentów funkcji na stosie względem rejestru `%ebp`

Jeżeli założymy że każdy argument funkcji zajmuje 4 bajty to adresem N-tego argumentu jest `8+4*N (N=0,1,2,3,...)` względem rejestru `%ebp`. Zmienne lokalne także trzymane są na stosie gdyż nie można zagwarantować że zmieszczą się w rejestrach. Drugim powodem lokalizacji zmiennych na stosie jest brak gwarancji że wywoływane w funkcji inne funkcje nie nadpiszą używanych w funkcji rejestrów.
Kolejnym krokiem (opcjonalnym), jest zachowanie rejestrów które mogą być nadpisane w wywoływanej funkcji a mogą być potrzebne w funkcji wywołującej. Następnym krokiem jaki może być wykonany, jest rezerwacja miejsca na stosie dla zmiennych lokalnych. W podanym niżej przykładzie jest to 80 bajtów.

```
prologue:
        pushl   %ebp            / save frame pointer
        movl    %esp, %ebp      / set new frame pointer
        subl    $80, %esp       / allocate stack space
        pushl   %edi            / save local register
        pushl   %esi            / save local register
        pushl   %ebx            / save local register
```

Kod 12-1 Prolog wykonania nowej funkcji

Przykładowo jeżeli chcemy zarezerwować na stosie miejsce dla dwóch zmiennych 4 bajtowych zmniejszamy wskaźnik stosu o 8.

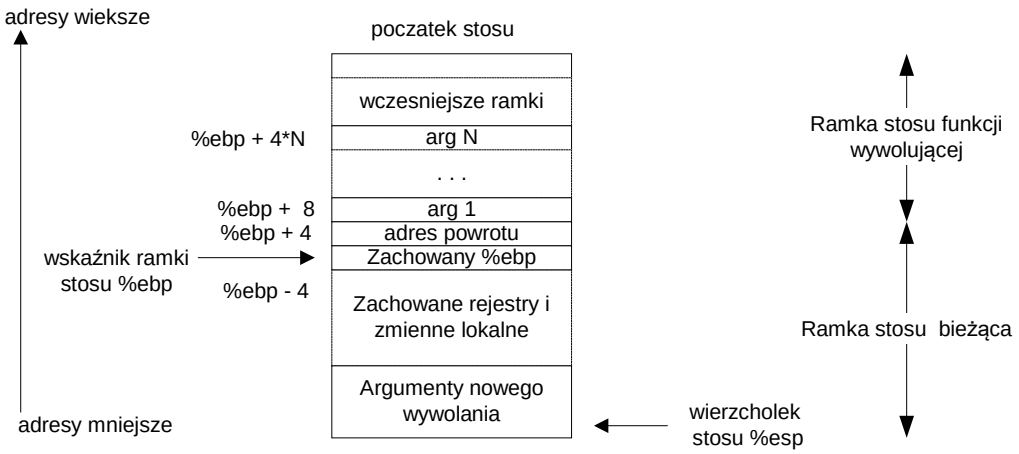
```
subl $8, %esp
```

Stos będzie wtedy miał postać:

```
Parametr #N <--- N*4+4(%ebp)
...
Parametr 2 <--- 12(%ebp)
Parametr 1 <--- 8(%ebp)
Adres powrotu <--- 4(%ebp)
Stary %ebp <--- (%ebp)
Zmienna lokalna 1 <--- -4(%ebp)
Zmienna lokalna 2 <--- -8(%ebp)i(%esp)
```

Rys. 12-6 Obraz stosu po rezerwacji miejsca na 2 zmienne lokalne

Jak widać zarówno parametry funkcji jak i zmienne lokalne są dostępne jako przesunięcie względem rejestru `%ebp`. Ramka stosu pokazana jest poniżej.



Rys. 12-7 Ramka stosu

Aktualna ramka stosu ograniczona jest przez dwa rejestry:

- Wskaźnik ramki stosu (ang. *frame pointer*) `%ebp` który wskazuje na początek ramki stosu funkcji.
- Wskaźnik stosu (ang. *stack pointer*) `%esp` który wskazuje na koniec ramki stosu funkcji.

Kiedy funkcja się kończy wykonywane są następujące kroki.

1. Wartość zwracana przez funkcję przesyłana jest do rejestru `%eax`.
2. Przywrócenie stosu do postaci takiej jak w chwili gdy funkcja była wywoływana. Zwolnienie miejsca na zmienne lokalne i przywrócenie poprzedniej wartości ramki stosu `%ebp`.
3. Przesłanie ze stosu do licznika instrukcji adresu następnej instrukcji po `call`. Wykonuje to rozkaz `ret`.

Zanim nastąpi powrót do funkcji wywołującej, należy przywrócić postać stosu taką jaka była bezpośrednio po wejściu do funkcji. Stąd musimy przywrócić wartość wskaźnika stosu `%esp` i rejestru bazowego `%ebp` z chwili wejścia do funkcji. Odbywa się to przez skopiowanie do `%esp` zawartości `%ebp` (`movl %ebp, %esp`) i przywrócenie poprzedniej wartości rejestru `%ebp` (`popl %ebp`).

Czynności te można wykonać jedną instrukcją `leave`. Tak więc zakończenie funkcji ma postać:

```
movl %ebp, %esp  # odtwarzamy stary stos
popl %ebp        # odtwarzamy stary %ebp
ret              # powrot z funkcji
```

Rys. 12-8 Sekwencja zakończenia funkcji

Gdy nastąpi powrót z funkcji, to stos się wycofa do poprzedniej postaci i pamięć na zmienne lokalne zostanie zwolniona.

```
# wejscie do funkcji
pushl %ebp        # zabezpieczenie starego %ebp
movl %esp, %ebp   # nowy %ebp zawiera aktualny wsk. stosu

# rozkazy funcji

#opuszczenie funkji
movl %ebp, %esp   # odtwarzamy stary stos
popl %ebp         # odtwarzamy stary %ebp
ret               # powrot z funkcji
```

Kod 12-1 Wejście i wyjście z funkcji

Trochę inny wzorzec zakończenia funkcji podano poniżej. Przywraca on zachowane na stosie rejestry `%ebx,%esi, %edi` i umieszcza w `%eax` z zwracaną przez funkcję wartośc.

```
        movl   %edi, %eax    / set up return value
epilogue:
        popl   %ebx          / restore local register
        popl   %esi          / restore local register
        popl   %edi          / restore local register
        leave                / restore frame pointer
        ret                  / pop return address
```

Kod 12-2 Inny wzorzec epilogu zakończenia funkcji

12.5 Zmienne globalne i lokalne

Zmienne globalne deklarowane są w segmentach `.data` i `.bss`. Zmienne lokalne utrzymywane są na stosie na poczatku ramki stosu funkcji. Dostęp do zmiennych globalnych i lokalnych jest różny. Zmienne globalne dostępne są przez adresowanie pośrednie (nazwa zmiennej jest adresem pod którym ona występuje), zmienne lokalne pośrednio względem rejestru indeksowego `%ebp`. Przykłady dane poniżej pochodzą z [3].

```
int my_global_var;

int func(){
  int my_local_var;
  my_local_var = 1;
  my_global_var = 2;
  return 0;
}
```

Przykład 12-2 Funkcja `func` w języku C

```
.section .data
.lcomm my_global_var, 4
.type func, @function
func:
pushl %ebp        # zachowaj stary base pointer
movl %esp, %ebp   # skopiuj wskaznik stosu do base pointer
subl $4, %esp     # zarezerwoj miejsce na my_local_var
.equ my_local_var, -4
# mozna uzywac stalej my_local_var by uzyskac  dostep do zmiennej
# my_local_var
movl $1, my_local_var(%ebp)
movl $2, my_global_var
movl %ebp, %esp   # zakonczenie funkcji i return
popl %ebp
ret
```

Przykład 12-3 Funkcja func w języku assemblera

Rezerwacja miejsca na stosie na zmienną `my_local_var` następuje przez wykonanie odejmowania:

```
subl $4, %esp
```

Dostęp jest względem rejestru `%ebp` z przesunięciem –4.

12.6 Przykład 1 funkcji – wykorzystanie języka C

Wykorzystajmy bardzo prostą funkcję napisaną w języku C która zwraca sumę dwóch liczb.

```
int sum(int x, int y)
{
      return (x+y);
}
```

Poniżej podany jest program dodaj.s w assemblerze który definiuje tę funkcję i następnie ją wywołuje. Program dany jest poniżej.

```
# program wywolujacy funkcje dodawania dwoch liczb
# Wynik zwracany jako kod powrotu
# kompilacja:
# as dodaj.s --32 -g -o dodaj.o
# ld dodaj.o -m elf_i386 -o dodaj

.section .data
arg1: .long 3
arg2: .long 4

.section .text
.globl _start
_start:
    pushl arg2                # push second argument
    pushl arg1                # push first argument
    call  dodaj               #call the function
    addl  $8, %esp            # Wyrownaj stos

    # Wywolaj zakonczenie procesu
    movl  %eax, %ebx         # Suma w %ebx - kod powrotu
    movl  $1, %eax           # exit (%ebx is returned)
    int   $0x80

# funkcja int dodaj(int a, int b) -------------------------------
.type dodaj, @function
dodaj:
    pushl %ebp           # save old base pointer
    movl  %esp, %ebp     # make stack pointer the base pointer
    movl  8(%ebp), %eax  # put first argument in %eax
    movl  12(%ebp), %ecx # put second argument in %ecx
    addl  %ecx, %eax     # dodaj zawartosc %ecx do %eax
    # wynik w %eax
    movl %ebp, %esp      # restore the stack pointer
    popl %ebp            # restore the base pointer
    ret
```

Przykład 12-1 Program `dodaj.s` – ilustracja wywoływania funkcji

Kompilujemy i łączymy program jak pokazano poniżej.

```
as dodaj.s --32 -g -o dodaj.o
ld dodaj.o -m elf_i386 -o dodaj
```

Następnie możemy zaobserwować jego działania używając debuggera gdb lub kdbg. Ustawiamy breakpoint na instrukcji `call dodaj` a następnie uruchamiamy program dochodząc do instrukcji `call`. Widzimy że adres następnej po `call` instrucji to `0x8048085` (klikamy w plusa przed instrukcją). Następnie obserwujemy obraz stosu.

```
+    7 .section .data
+    8 arg1: .long 3
+    9 arg2: .long 4
+   10
+   11 .section .text
+   12 .globl _start
+   13 _start:
+   14     pushl arg2               # push second argument
+   15     pushl arg1               # push first argument
+   16     call  dodaj              #call the function
-   17     addl  $8, %esp           # Wyrownaj stos
       0x8048085 add    $0x8,%esp
```

Memory

($esp)

Address				
0xbffff3f8	0x00000003	0x00000004	0x00000001	0xbffff58a

Przykład 12-2 Obraz stosu programu dodaj.s przed wywołaniem funkcji dodaj.

Należy zauważyć że stos wyświetlamy w oknie pamięci jako `($esp)`. Na stosie widzimy argumenty funkcji 3 i 4. Następnie wchodzimy do funkcji (klawisz F8) i obserwyjemy obraz stosu.

```
-   17     addl  $8, %esp           # Wyrownaj stos
       0x8048085 add    $0x8,%esp
+   18
+   19     # Wywolaj zakonczenie procesu
+   20     movl  %eax, %ebx         # Suma w %ebx - kod powrotu
+   21         movl  $1, %eax           # exit (%ebx is returned)
+   22         int   $0x80
+   23
+   24 # funkcja int dodaj(int a, int b) ---------------------------
+   25 .type dodaj, @function
+   26 dodaj:
+   27         pushl %ebp           # save old base pointer
```

Memory

($esp)

Address				
0xbffff3f4	0x08048085	0x00000003	0x00000004	0x00000001

Przykład 12-3 Obraz stosu programu `dodaj.s` wewnątrz funkcji `dodaj`.

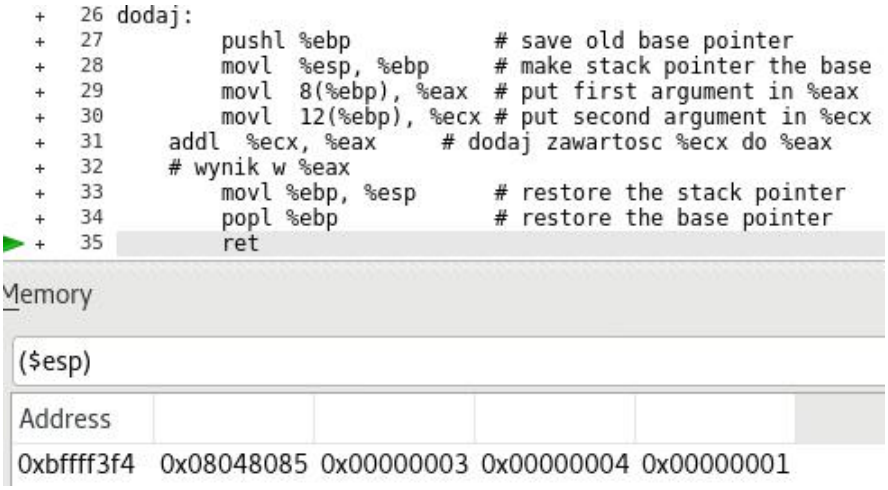
Widzimy na stosie (na powyższym ekranie) że na stosie jest adres powrotu `0x8048085` a następnie dwa argumenty 3 i 4. Przechodzimy następnie do instrukcji `addl %ecx, %eax` i przełączamy się na wyświetlanie rejestrów. Poniżej widać że są tam wartości argumentów 3 i 4.

Registers

Register	Wartość	Decoded value
▼ GP and ot…		
eax	0x3	3
ecx	0x4	4
edx	0x0	0
ebx	0x0	0
esp	0xbffff3f0	0xbffff3f0
ebp	0xbffff3f0	0xbffff3f0

Przykład 12-4 rejestry programu `dodaj.s` wewnątrz funkcji `dodaj`.

Wykonujemy jeszcze kilka kroków dochodząc do instrukcji `ret` co pokazano poniżej.



Przykład 12-5 Obraz stosu przed wykonaniem instrukcji `ret`.

Widać że na szczycie stosu jest adres powrotu czyli `0x8048085.` Wykonajmy jeszcze kilka kroków do zakończenia programu.

12.7 Przykład funkcji – potęgowanie w C

Rozważmy daną niżej funkcję `cpower` podnoszącą liczbę do potęgi pot.

```
// Funkcja cpower - podnoszenie do potegi
// kompilacja gcc -O1 -S cpowerfun.c
int cpower(int liczba, int pot)
{
      int i = 1;
      int wynik = liczba;
      // if(pot == 0) return 1;
      // if(pot == 1) return wynik;
      do {
       wynik = wynik * liczba;
       i++;
      } while(i<pot);
      return wynik;
}
```

Przykład 12-6 Plik `cpower.c` zawierający funkcję `cpower` podnoszenia do potęgi
Podobnie jak poprzednio skompilujmy ten program z opcją -S.

```
$gcc -O1 -S cpowerfun.c
```

Możemy przeanalizować wygenerowany kod assemblerowy.

```
      .file "cpowerfun.c"
      .text
      .globl      cpower
      .type cpower, @function
cpower:
.LFB0:
      .cfi_startproc
      pushl %ebx
      .cfi_def_cfa_offset 8
      .cfi_offset 3, -8
      movl  8(%esp), %ecx
      movl  12(%esp), %ebx
      movl  %ecx, %eax
      movl  $1, %edx
.L2:
      imull %ecx, %eax
      addl  $1, %edx
      cmpl  %ebx, %edx
      jl    .L2
      popl  %ebx
      .cfi_restore 3
      .cfi_def_cfa_offset 4
      ret
      .cfi_endproc
```

Przykład 12-7 Kod assemblerowy funkcji cpower.s

12.8 Przykład funkcji – potęgowanie

Jako przykład rozważymy funkcję podnoszenia do potęgi o nazwie `power.s` w przykładzie pochodzącym z [3].

```
# kompilacja:
# as power.s -m32 -g -o power.o
# ld power.p -m elf_i386 -o power
#
#PURPOSE:  Program to illustrate how functions work
      #          This program will compute the value of
      #          2^3 + 5^2
      #

      #Everything in the main program is stored in registers,
      #so the data section doesn't have anything.
      .section .data
      .section .text
      .globl _start
_start:
      pushl $3                  #push second argument
      pushl $2                  #push first argument
      call  power               #call the function
      addl  $8, %esp            # usuniecie argumentow ze stosu

      pushl %eax                #save the first answer before
                                #calling the next function
      pushl $2                  #push second argument
      pushl $5                  #push first argument
      call  power               #call the function
      addl  $8, %esp            # usuniecie argumentow ze stosu

      popl  %ebx                #The second answer is already
                                #in %eax.  We saved the
```

```
                                #first answer onto the stack,
                                #so now we can just pop it
                                #out into %ebx

      addl  %eax, %ebx          #add them together
                                #the result is in %ebx

      movl  $1, %eax            #exit (%ebx is returned)
      int   $0x80

      #PURPOSE:  This function is used to compute
      #          the value of a number raised to
      #          a power.
      #
      #INPUT:    First argument - the base number
      #          Second argument - the power to
      #                            raise it to
      #
      #OUTPUT:   Will give the result as a return value
      #
      #NOTES:    The power must be 1 or greater
      #
      #VARIABLES:
      #          %ebx - holds the base number
      #          %ecx - holds the power
      #
      #          -4(%ebp) - holds the current result
      #
      #          %eax is used for temporary storage
      #
      .type power, @function
power:
      pushl %ebp            #save old base pointer
      movl  %esp, %ebp      #make stack pointer the base pointer
      subl  $4, %esp        #get room for our local storage
      movl  8(%ebp), %ebx   #put first argument in %ebx
      movl  12(%ebp), %ecx #put second argument in %ecx
      movl  %ebx, -4(%ebp) #store current result

power_loop_start:
      cmpl  $1, %ecx        #if the power is 1, we are done
      je    end_power
      movl  -4(%ebp), %eax #move the current result into %eax
      imull %ebx, %eax      #multiply the current result by
                            #the base number
      movl  %eax, -4(%ebp) #store the current result
      decl  %ecx            #decrease the power
      jmp   power_loop_start #run for the next power

end_power:
      movl -4(%ebp), %eax   #return value goes in %eax
      movl %ebp, %esp       #restore the stack pointer
      popl %ebp             #restore the base pointer
      ret
```

Przykład 12-4 Program wykorzystujący funkcję podnoszenia do potęgi

W programie tym utworzono funkcję `power` która posiada dwa argumenty. Pierwszym argumentem jest liczba podnoszona do potęgi a drugim potęga. Funkcja zwraca wynik potęgowania w rejestrze `%eax`. Argumenty funkcji przekazywane są przez stos (w odwrotnej kolejności). Najpierw drugi argument a potem pierwszy.

```
pushl $3                #push second argument
pushl $2                #push first argument
call  power             #call the function
addl  $8, %esp          #move the stack pointer back
```

Po wykonaniu funkcji stos przywracany jest do poprzedniej postaci (instrukcja `addl  $8, %esp`) poprzez dodanie liczby 8 co odpowiada 2 argumentom typu `int`. Po wejściu do funkcji wykonywane są instrukcje:

```
pushl %ebp          #save old base pointer
movl  %esp, %ebp    #make stack pointer the base pointer
subl  $4, %esp      #get room for our local storage
```

Pierwsza zabezpiecza na stosie stary rejestr bazowy `%ebp`, druga tworzy nowy rejestr bazowy, trzecia zabezpiecza na stosie miejsce na wynik tymczasowy. Obraz stosu po wykonaniu tych instrukcji jest pokazany poniżej.

```
Arg 1 - liczba  <--- 12(%ebp)
Arg 2 - potęga  <--- 8(%ebp)
Adres powrotu   <--- 4(%ebp)
Stary %ebp      <--- (%ebp)
Wartość bieżąca <--- -4(%ebp)
```

Przykład 12-5 Stan stosu po wejściu do funkcji

Ważnym elementem funkcji jest pobranie argumentów funkcji. Można uzyskać do nich dostep za pomocą rejestru bazowego co widać z poprzedniego diagramu.

```
movl  8(%ebp), %ebx  # prześlij pierwszy argument do %ebx
movl  12(%ebp), %ecx # prześlij drugi argument do %ecx
```

Ważne jest też zakończenie funkcji.

```
movl -4(%ebp), %eax  #return value goes in %eax
movl %ebp, %esp      #restore the stack pointer
popl %ebp            #restore the base pointer
ret
```

Funkcja przekazuje wynik przez rejestr `%eax`, przywraca stan stosu i rejestru bazowego i poprzez `ret` wykonuje skok do instrukcji następnej po `call`.

12.9 Zabezpieczenie rejestrów

Jeżeli funkcja P wywołuje funcję Q to być może używa ona rejestrów które używa funkcja P. Tak więc Q powinna na wstepie zabezpieczyć pewne rejestry aby odtworzyć je przy zakończeniu.

```
int P(int x)
{
 int y = x*x;
 int z = Q(y);
 return y + z;
}
```

Funkcja P używa zmiennej y także po zakończeniu funkcji Q i musi być ona zabezpieczona (o ile y było w rejestrze). Są tu dwie możliwości:

- P zachowa rejestry, wywoła Q a potem je odtworzy
- Q na wstępie zachowa rejestry a potem je odtworzy

12.10 Zadania

12.10.1 Funkcja - dodawanie liczb

Uruchom program `dodaj.s` który definiuje i wywołuje funkcję int `dodaj(int a, int b)` która dodaje dwie liczby a i b i zwraca jej wynik. Przykładowe argumenty należy umieścić na stosie a następnie wywołać funkcję `dodaj`. Po jej wykonaniu należy przekazać wynik jako kod powrotu i zakończyć proces wywołaniem

systemowym `exit`. Przebieg wywołania funkcji należy zaobserwować za pomocą debuggera gdb lub kdbg. Wyświetlanie zawartości stosu może być wykonane za pomocą polecenia:

```
x/rsf &adres
r – liczba powtórzeń
s – długość wyświetlanej jednostki: b(byte), h(halfword), w(word),
    g(giant, 8 bytes)
f – format wyświetlanej jednostki: x(hex), d(decimal), u(unsigned decimal),
    t(binary), f(float), a(address), i(instruction), c(char), s(string)
```

Przykładowo:

```
x/6wd $esp
```

wyświetla 6 pozycji stosu długości 4 bajtów w formacie dziesietnym.

12.10.2 Zamiana liczb

Dany jest program w języku C wywołujący funkcję swap_add.

```
#include <stdio.h>

int main(void) {
 int arg1 = 534;
 int arg2 = 1057;
 int diff,sum;
 printf("arg1: %d arg2: %d\n",arg1,arg2);
 sum = swap_add(&arg1, &arg2);
 printf("arg1: %d arg2: %d\n",arg1,arg2);
 diff = arg1 - arg2;
 printf("sum: %d diff: %d\n",sum,diff);
 return sum * diff;
}

int swap_add(int *xp, int *yp)
{
 int x = *xp;
 int y = *yp;

 *xp = y;
 *yp = x;
 return x + y;
}
```

Przykład 12-6 Program `swap_add.c` w języku C

Poniżej dany jest napisany w assemblerze program zamien-as2.s zawierający funkcję swap_add i jej wywołanie..

```
# Program wywoluje funkcje zamiany miejscami argumenty
# int swap_add(int *xp, int *yp)
# zwraca sume tych argumentow
# kompilacja:
# as zamien-as2.s --32 -g -o zamien-as2.o
# ld zamien-as2.o -m elf_i386 -o zamien-as2
.section .data
arg1:
.long 3
arg2:
.long 4

.section .text
.globl _start
_start:
```

```
 leal arg2, %eax        # Oblicz adres &arg2
 pushl %eax             # Adres arg2 na stos
 leal arg1, %eax        # Oblicz adres &arg1
 pushl %eax             # Adres arg1 na stos

 call swap_add               # Wywolaj funkcje swap_add

 movl  %eax, %ebx           # kod powrotu do %ebx
 movl  $1, %eax              # kod funkcji exit do %eax
 int   $0x80                 # wywolaj system

# Kod funkcji swap_add tutaj
.type swap_add, @function
swap_add:
 pushl %ebp                  # Save old %ebp
 movl %esp, %ebp             # Set %ebp as frame pointer
 pushl %ebx                  # Save %ebx
 movl 8(%ebp), %edx          # Get xp
 movl 12(%ebp), %ecx         # Get yp
 movl (%edx), %ebx           # Get x
 movl (%ecx), %eax           # Get y
 movl %eax, (%edx)           # Store y at xp
 movl %ebx, (%ecx)           # Store x at yp
 addl %ebx, %eax        # Return value = x+y
 popl %ebx              # Restore %ebx
 movl %ebp, %esp         # Restore %esp from %ebp
 popl %ebp              # Restore %ebp
 ret
```

Przykład 12-7 Program `zamien-as2.s` zawierający kod funkcji `swap_add` w assemblerze

Napisz w assemblerze program `zamien-asf.s` zawierający tylko funkcję `swap_add` oraz w oddzielnym pliku w języku C program `zamien-asf.c` wywołujący tę funkcje. Obydwa programy kompilujemy jak niżej:

```
gcc zamien-asf.c zamien-asf.s -g -m32 -o zamien-asf
```

Uwaga!
W programie `zamien-asf.s` należy dodać dyrektywę `./global swap_add` aby symbol `swap_add` był widoczny poza plikiem `zamien-asf.s`. Zaobserwuj działanie programów w debuggerze kdbg.

12.10.3 Funkcje read, write, exit

Podany wczesniej przykład programu `read_write.s` wykorzystuje wywołania systemowe `read`, `write` i `exit`. Zdefiniowano je w Tab. 9-1. Napisz je w postaci funkcji których prototypy dano poniżej.

Prototyp	Opis	Numer wyw. syst.
`int read(int fh, char * buf, int ile)`	Odczyt z pliku fh do bufora buf znaków ile. Zwraca liczbę odczytanych znaków	3
`int write(int fh, char * buf, int ile)`	Zapis do pliku fh z bufora buf znaków ile. Zwraca liczbę zaisanych znaków	4
`int exit(int kod)`	Zakończenie procesu z kodem powrotu kod	1

Przekazanie parametrów funkcji ma być przez stos a wynik ma być w rejestrze `%eax`. Następnie wywołaj te funkcje podobnie jak w programie `read_write.s`. Program ma wykonać następujące kroki:

- Wyprowadzenie komunikatu „`Wprowadz napis`” na konsolę za pomocą fukcji `write`
- Wczytanie do bufora napisu z konsoli za pomocą funkcji `read`.
- Wyprowadzenie na konsolę wprowadzonego uprzednio napisu
- Zakonczenie procesu za pomocą funkcji `exit(kod_powrotu)`. Kod powrotu = długość wprowadzonego napisu.

12.10.4 Funkcje read, write, exit – dwa pliki

W poprzednim przykładzie zarówno funkcje `read, write, exit` jak ich wywołania były w jednym pliku. W drugim etapie umieść te funkcje w oddzielnym pliku `mylib.s` Wywołaj funkcje `read, write, exit` z programu umieszczonego w pliku `read_write_callmylib.s`
Kompilacja:

```
as mylib.s –32 –g –o mylib.o
gcc read_write_callmylib.s mylib.o –m32 –g –o read_write_callmylib
```

12.10.5 Funkcje open, read, write, close, exit

Uzupełnij poprzedni program o funkcje `open i close`.

Prototyp	Opis	Numer wyw. syst.
`int open(char *path, int flags, int mode)`	Funkcja powoduje otwarcie pliku lub urządzenia o nazwie wyspecyfikowanej w parametrze **`path`**. Otwarcie następuje zgodnie z trybem flags. Funkcja zwraca deskryptor pliku (uchwyt). Uchwyt pliku służy do identyfikacji pliku w innych funkcjach systemowych.	5
`int close(int fh)`	Zamknięcie pliku o uchwycie fh	6

`int open(char *path,int oflag,[mode_t mode])`
gdzie:

`path`	Nazwa pliku lub urządzenia
`oflag`	Tryb dostępu do pliku – składa się z bitów – opis w pliku nagłówkowym <fcntl.h>
`mode`	Atrybuty tworzonego pliku (prawa dostepu)

`int close(int fdes)`

gdzie:

`fdes`	Uchwyt do pliku zwracany przez funkcję open

Funkcja powoduje zamknięcie pliku identyfikowanego przez fdes. Należy ją wykonać gdy nie będą już wykonywane operacje na danym pliku .

Nazwa	`%eax`	`%ebx`	`%ecx`	`%edx`	`Uwagi`
`open`	`5`	Nazwa pliku	Flagi	Prawa dostępu	Otwarcie pliku. Uchwyt zwracany w %eax
`close`	`6`	Uchwyt pliku	-	-	Zamknięcie pliku

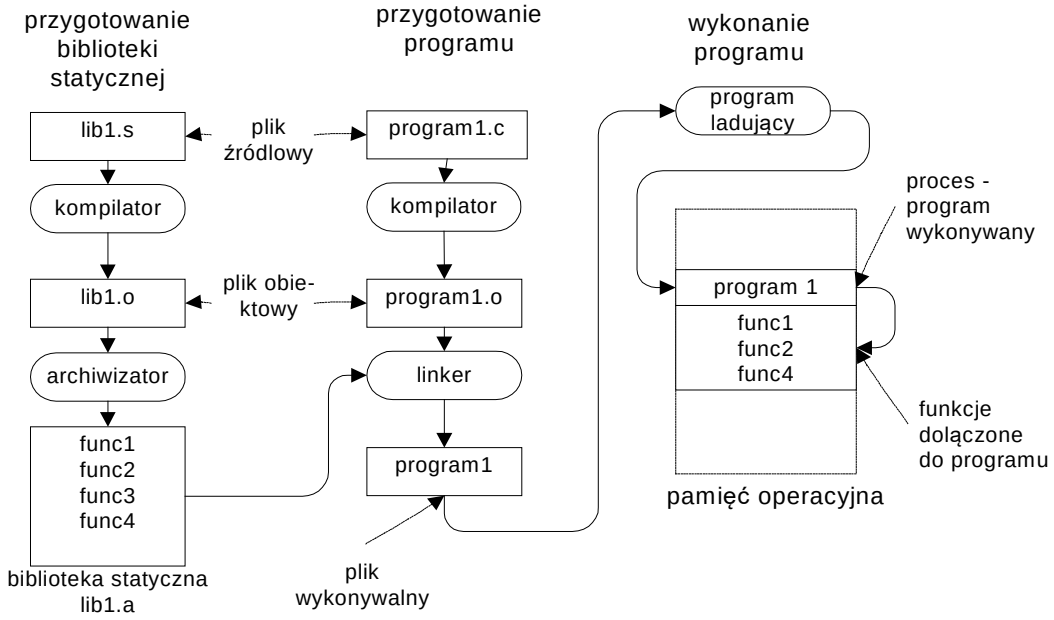
Funkcje: open, read, write, close, exit maja być umieszczone w oddzielnym pliku.

13. Biblioteki

Programy wykonują wiele typowych czynności których samodzielne programowanie byłoby niecelowe. Czynności te, typowo realizowane są przez napisane wcześniej funkcje. Funkcje te zgrupowane są w bibliotekach, które w dużej części są nieodłącznym elementem systemu. Wyróżniamy dwa rodzaje bibliotek:

- Biblioteki statyczne
- Biblioteki dynamiczne (współdzielone)

Biblioteka statyczna składa się z funkcji i danych na których funkcje te operują. Bibliotekę tworzy się kompilując zestawy funkcji, zawarte w jednym lub wielu plikach źródłowych, do postaci plików obiektowych. Pliki te następnie łączone są w archiwum za pomocą polecenia `ar`. Gdy program łączący buduje plik wykonywalny i gdy jest wskazanie by korzystał z bibliotek statycznych, to przeszukuje on bibliotekę i dołącza do pliku wykonywalnego moduły zawierające potrzebne funkcje. Schemat tworzenia biblioteki statycznej pokazuje Rys. 13-1.

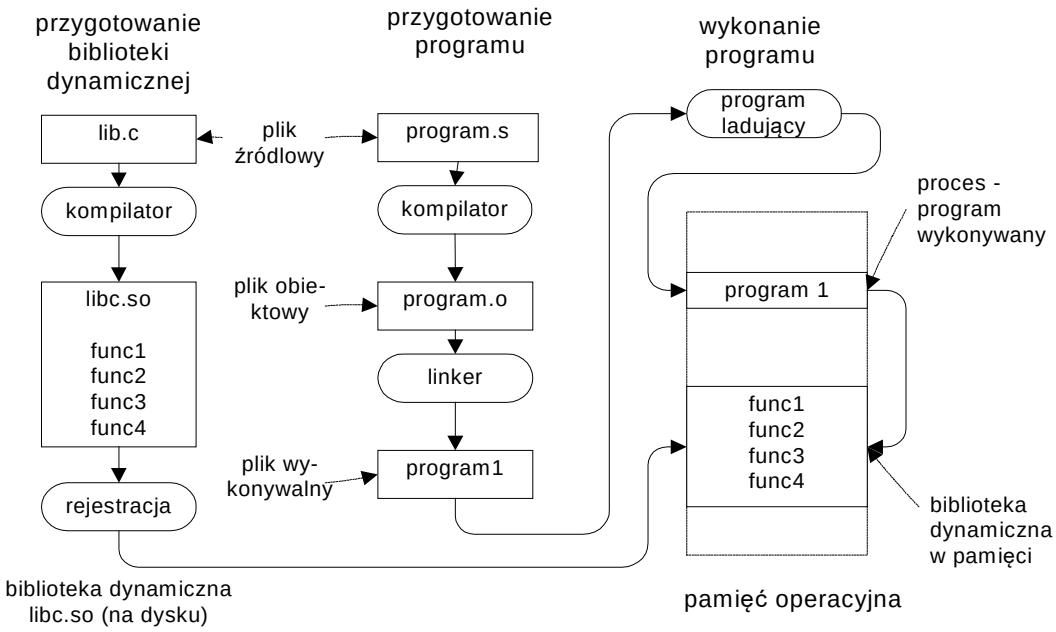


Rys. 13-1 Tworzenie i wykorzystanie biblioteki statycznej

Innym rodzajem bibliotek są biblioteki współdzielone. Zastosowanie bibliotek statycznych ma tak skutek że:

- Kod występujących w nich funkcji występuje wielokrotnie w różnych programach co powoduje niepotrzebne straty przestrzeni systemu plików
- W przypadku znalezienia w bibliotece błędu, należy przekompilować wszystkie programy korzystające z tej biblioteki.

Powstał więc pomysł by biblioteki umieścić w ogólnie znanym miejscu, a wtedy wiele programów mogłoby z nich korzystać. Jest to koncepcja biblioteki współdzielonej.



Rys. 13-2 Tworzenie i wykorzystanie biblioteki współdzielonej

Rozważmy poniższy program który wypisuje komunikat "hello world".

```
#PURPOSE: This program writes the message "hello world" and
# exits
#
.include "linux.s"
.section .data
helloworld:
.ascii "hello world\n"
helloworld_end:
.equ helloworld_len, helloworld_end - helloworld
.section .text
.globl _start
_start:
movl $STDOUT, %ebx
movl $helloworld, %ecx
movl $helloworld_len, %edx
movl $SYS_WRITE, %eax
int $LINUX_SYSCALL
movl $0, %ebx
movl $SYS_EXIT, %eax
int $LINUX_SYSCALL
```

Przykład 13-1 Program `hello_nolib.s` piszący komunikat `"hello world"` bez użycia bibliotek

Program ten używa pliku `linux.s` który jest włączony za pośrednictwem dyrektywy `.include "linux.s"`

```
#Common Linux Definitions
#System Call Numbers
.equ SYS_EXIT, 1
.equ SYS_READ, 3
.equ SYS_WRITE, 4
.equ SYS_OPEN, 5
.equ SYS_CLOSE, 6
.equ SYS_BRK, 45

#System Call Interrupt Number
.equ LINUX_SYSCALL, 0x80

#Standard File Descriptors
.equ STDIN, 0
.equ STDOUT, 1
.equ STDERR, 2

#Common Status Codes
.equ END_OF_FILE, 0
```

Przykład 13-2 Plik "`linux.s`" zawierający typowe definicje

Kompilujemy powyższy program jak poniżej.

```
as hello_nolib.s -o hello_nolib.o –g
ld hello_nolib.o –o hello_nolib
```

Następnie wykonujemy uzyskując komunikat jak poniżej.

```
$./hello_nolib
hello world
```

W powyższym programie do wyprowadzenia na konsolę napisu użyliśmy wywołania systemowego `$SYS_WRITE`.

Za pomocą programu `file` możemy uzyskać informację o pliku wykonywalnym co pokazano poniżej.

```
$file hello_nolib
hello_nolib: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
statically linked, not stripped
```

Zamiast używać wprost wywołania systemowego możemy skorzystać z funkcji `printf i exit` zawartych w bibliotece `libc`. Biblioteka `libc` jest standardową biblioteką dla języka C w systemie Linux. Jej zawartość i sposób użycia zawarty jest w dokumentacji https://www.gnu.org/software/libc/manual/ . Proszę zapoznać się z tą stroną. Biblioteka ta zawiera dużą liczbę funkcji ogólnie wykorzystywanych w systemie Linux. W szczególności możemy odnaleźć tam znane już funkcje `open, read, write, exit`. Obecnie napiszemy program o takiej funkcjonalności jak poprzednio, ale użyjemy bibliotecznej funkcji `printf` i funkcji `exit`.

```
# Program pisze komunikat "hello world" i sie konczy
# kompilacja: gcc hello_lib.s -o hello_lib
.section .data
helloworld:
.ascii "hello world\n\0"
.section .text
.globl main
main:
#kopiujemy adres na stos
pushl $helloworld
# wywolanie printf
call printf
# wyrownanie stosu
addl $4, %esp
pushl $0
call exit
```

Przykład 13-3 Program `hello_lib.s` piszący komunikat "`hello world`" z użyciem biblioteki glibc

W programie użyta została biblioteczna funkcja `printf` i funkcja `exit`. Należy wiedzieć, że jako argument funkcji `printf` należy podać adres łańcucha "`hello world\n\0`" który to musi być umieszczony na stosie. Podobnie argumentem funkcji `exit` jest kod powrotu 0 który także umieszczamy na stosie. Program kompilujemy jak poniżej:

```
gcc hello_lib.s –o hello_lib
```

W związku z tym że do wytworzenia kodu wynikowego używamy kompilatora `gcc`, poczatek programu głównego musi być oznaczony etykietą globalną `main` a nie `_start` jak w poprzednich przypadkach. Program będzie używać bibliotek dynamicznych i powoduje dołączenie do programu wynikowego interfejsu do bibliotek dynamicznych. Możemy się o ty przekonać przy użyciu programu `file`, co pokazano poniżej.

```
$file hello_lib
hello_lib: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.2, not stripped
```

Uruchomienie programu daje łatwe do przewidzenia wyniki które podano poniżej.

```
$./hello_lib
hello world
```

W programie występują nazwy funkcji `printf` i `exit` W czasie linkowania linker nie zna adresów tych funkcji, dołącza jednak tak zwany linker dynamiczny. Gdy program jest uruchamiany, najpierw startuje program `/lib/ld-linux.so.2` który jest właśnie linkerem dynamicznym. Gdy program ten startuje, analizuje plik `hello_lib` i znajduje tam informację że potrzebna jest biblioteka `libc.so` . Linker dynamiczny musi teraz odnaleźć tę bibliotekę. Informacja o jej położeniu znajduje się w plikach konfiguracyjnych systemu (między innymi w pliku `/etc/ld.so.conf` i zmiennej otoczenia `LD_LIBRARY_PATH`). Na podstawie tych informacji linker dynamiczny odnajduje bibliotekę `libc.so`. Dalej adresy funkcji `printf` i `exit` są odnajdywane w bibliotece dynamicznej `libc.so` i wstawiane do kodu. Informacje o bibliotekach zawartych w programie wykonywalnym można uzyskać za pomocą programu `ldd` co pokazano poniżej.

```
$ldd ./hello_lib
linux-gate.so.1 (0xb7789000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75b3000)
/lib/ld-linux.so.2 (0x80036000)
```

Posługując się bibliotekami należy posiadać następujące informacje:

1. Jakie funkcje zawarte są w bibliotece
2. Jakie te funkcje mają argumenty (kolejność, typ znaczenie)

Większość programów systemu Linux napisana jest w języku C. Stąd i specyfikacja postaci tych funkcji jest dostosowana do języka C i nosi nazwę prototypu. Przykładowo:

```
int printf(char *string, ...);
int exit(int kod_powrotu);
```

W pierwszym przypadku parametr `char * string` jest adresem pierwszego bajtu tego łańcucha. Stąd na stos kopiujemy adres łańcucha `helloworld` a nie pierwszy znak co pokazano poniżej.

```
pushl $helloworld
call printf
```

Funkcja `printf` może posiadać zmienną liczbę argumentów. Sposób postepowania demonstruje poniższy przykład.

```
#Program pokazuje jak wywoływać funkcję printf
#
.section .data
#Ponizszy lancuch jest lancuchem formatujacym. Jest to pierwszy parametr i
funkcja printf uzywa go do ustalenie ile jest argumentow i jakiego sa typu
firststring:
.ascii "Hello! %s is a %s who loves the number %d\n\0"
name:
.ascii "Jonathan\0"
personstring:
.ascii "person\0"
numberloved:
.long 3
.section .text
.globl main
main:
# Parametry sa przekazywane w odwrotnej kolejnosci niż w prototypie
pushl numberloved   #To jest trzeci arg %d
pushl $personstring #Drugi arg %s
pushl $name         #Pierwszy arg %s
pushl $firststring  #Lancuch formatujacy
#in the prototype
call printf
pushl $0
call exit
```

Przykład 13-4 Program `printf-example.s` demonstrujący wywołanie funkcji `printf`

Program kompilujemy jak niżej:

```
$gcc printf-example.s –m32 -o printf-example
```

```
$./printf-example
Hello! Jonathan is a person who loves the number 3
```

Aby posługiwać się funkcjami biblioteki `libc` należy znać długości używanych tam typów danych. Pokazuje to poniższa tabela.

Typ	Długość	Uwagi
`int`	4	Liczba całkowita
`long`	4	Liczba całkowita
`long long`	8	Liczba całkowita podwójna
`short`	4	Liczba całkowita krótka
`char`	1	Znak
`float`	4	Liczba zmiennoprzecinkowa
`double`	8	Liczba zmiennoprzecinkowa podwójnej precyzji
*	4	Wskazuje że jest to adres zmiennej

Tab. 13-1 Długości argumentów różnych typów w języku C

Aby sprawdzić jakie funkcje zawarte są w danej bibliotece można użyć programu:

```
nm –D --defined-only nazwa_pliku_biblioteki.so
```

```
objdump –T nazwa_pliku_biblioteki.so | grep text
```

13.1 Zadania

13.1.1 Zamiana liter łańcucha na duże

Napisz program który wczytuje ze standardowego wejścia łańcuch za pomocą funkcji:

```
char *gets(char * string)
```

Następnie zamień małe litery na duże i wypisz tak przekształcony napis na konsoli uzywając funkcji bibliotecznej `printf`. Wykorzystaj wzorzec dany w `hello_lib2.s`.

13.1.2 Użycie funkcji scanf i printf

Napisz w assemblerze program `.s` który wykorzystuje pochodzące z biblioteki `glibc` funkcje `scanf` i `printf`. Program ma wczytywać: `int x,  char c, char tab[80]`

```
scanf("%d %c %s" ,x,c,tab)
```

Następnie należy te dane wypisać za pomocą funkcji:

```
printf("%d %c %s" ,x,c,tab);
```

13.1.3 Funkcja - dodawanie liczb z użyciem funkcji bibliotecznych

Na podstawie wcześniejszego programu `dodaj.s` napisz program `dodaj-funk.s` który definiuje i wywołuje funkcję int `dodaj(int a, int b)` która dodaje dwie liczby a i b i zwraca jej wynik. Należy wypisać komunikat: `printf("Podaj a i b\n\0")` następnie wczytać dane za pomocą wywołania funkcji `scanf("Podaj a i b\n\0",&arg1,&arg2)`, wywołać funkcję dodaj. Następnie nastepnie wypisać wynik za pomocą `printf("wynik %d\n”,wynik)` i zakończyć proces przez `exit(wynik)`. Kompilacja:

```
gcc dodaj-funk.s –o dodaj-funk
```

Zamiast etykiety `_start` należy użyć etykiety startowej `main`.

13.1.4 Tworzenie własnej biblioteki

W ćwiczeniu z poprzedniego rozdziału implementowano funkcje:

Prototyp	Opis	Numer wyw. syst.
`int open(char *path, int flags, int mode)`	Funkcja powoduje otwarcie pliku lub urządzenia o nazwie wyspecyfikowanej w parametrze `path`.	5
`int read(int fh, char * buf, int ile)`	Odczyt z pliku `fh` do bufora `buf` znaków `ile`. Zwraca liczbę odczytanych znaków	3
`int write(int fh, char * buf, int ile)`	Zapis do pliku `fh` z bufora `buf` znaków `ile`. Zwraca liczbę zapisanych znaków	4
`int close(int fh)`	Zamknięcie pliku o uchwycie `fh`	6
`int exit(int kod`	Zakończenie procesu z kodem powrotu `kod`	1

Umieść te funkcje w pliku `mylib.s`. Następnie utwórz bibliotekę wykonując polecenia/
Kompilacja:

```
as mylib.s –32 –g –o mylib.o
```

Archiwizacja, tworzenie biblioteki:

```
ar rcsv lmylib.a mylib.o
```

Sprawdzenie zawartości biblioteki:

```
nm lmylib.a
```

Wywołaj zawarte w bibliotece funkcje z programu `read_write_callmylib.s`. Kompilacja:

```
gcc read_write_callmylib.s -m32 -L. -lmylib -g -o read_write_callmylib
```

Spróbuj użyć standardowych funkcji `open, read, write, close, exit` zawartych w bibliotece standardowej `glibc`. Czy widzisz jakieś różnice względem utworzonej przez Ciebie biblioteki.

14. Wywoływanie funkcji w assemblera z programu w języku C

Wywołanie funkcji w assemblerze z programu w języku C nie jest trudne o ile rozumie się konwencję wywoływania funkcji języka C. Zgodnie z tą konwencją parametry funkcji są przesyłane na stos a zwracana wartość umieszczana jest w rejestrze `%eax`. Znaczy to że gdy wywołujemy funkcję z C kompilator umieści na stosie parametry funkcji a nasz program w assemblerze musi je ze stosu odczytać. Trzeba wiedzieć jednak dokładnie gdzie te parametry są przechowywane. Aby to stwierdzić trzeba wiedzieć jak wygląda stos gdy wywołujemy funkcję w assemblerze. Kiedy wywołujemy taką funkcję kompilator C wykonuje nastepujące kroki:

1. Wysyła na stos wszystkie parametry funkcji w odwrotnej kolejności do tej która pojawia się w prototypie funkcji. Dlaczego w odwrotnej kolejności ? Wydaje się, że dlatego, że gdy tak zrobimy pierwszy parametr będzie najbliższy wierzchołka stosu.
2. Wysyła na stos adres powrotu z wywoływanej funkcji. Będzie to adres instrukcji nastepnej po wywołaniu funkcji (call funkcja)

Wewnątrz funkcji assemblerowej powinny być wykonane nastepujące kroki:

1. Rejestr ramki stosu `%ebp` (ang. *base pointer/ frame pointer*) ma być przesłany na stos: `push %ebp`
2. Wskaźnik stosu `%esp` ma być przesłany do rejestru ramki stosu `%ebp`: `movl %esp, %ebp`

Operacje związane ze stosem to przesłanie tam parametrów funkcji, adresu powrotu funkcji i „starego” rejestru ramki stosu. Po tych operacjach stos będzie wyglądał jak poniżej.

```
Parametr #N       <--- N*4+4(%ebp)
...
Parametr 2        <--- 12(%ebp)
Parametr 1        <--- 8(%ebp)
Adres powrotu     <--- 4(%ebp)
Stary %ebp        <--- (%ebp)
--- wierzchołek stosu ----
```

Rys. 14-1 Obraz stosu po wejściu do funkcji

Z rysunku widać gdzie na stosie pamiętane są parametry funkcji i jak można do nich dotrzeć. Przy założeniu że parametry mają długość 4 bajtów położenie pierwszego parametru uzyskujemy dodając 8 do rejestru ramki `%ebp`. Dlaczego właśnie 8 ? Dlatego że trzeba przeskoczyć stary `%ebp` (4 bajty) i adres powrotu z funkcji (także 4 bajty). Wyrażając to w notacji assemblera AT&T przesłanie pierwszego parametru do rejestru `%eax` ma nastepujacą postać:

```
movl 8(%ebp), %eax
```

Adres drugiego parametru może być uzyskany przez dodanie 12 do rejestru bazowego `%ebp` (przy założeniu że długość pierwszego parametru to 4 bajty).

14.1 Przykład – modyfikacja tablicy

W podanym poniżej przykładzie z programu w języku C przekazujemy do funkcji w assemblerze dwa parametry. Pierwszy to wskaźnik na poczatek tablicy liczb liczb całkowitych `fren` a drugi to długość tej tablicy. Kody programów podane są poniżej.

```
// Program casmc.c wywolywanie funkcji w assemblerze z programu w C
// Program w assemblerze w pliku casma.s
// kompilacja:
// gcc casmc.c casma.s –m32 –o casm
// wykonaie:
// ./casm

#include <stdio.h>

/* prototype for asm function */
int * asm_mod_array(int *ptr,int size);
```

```
int main() {
 int fren[5]={ 1, 2, 3, 4, 5 };
 printf("Tablica przed wywolaniem: %d %d %d %d
        %d\n",fren[0],fren[1],fren[2],fren[3],fren[4]);

 /* call the asm function */
 asm_mod_array(fren, 5);

 printf("Po wywolaniu: %d %d %d %d
       %d\n",fren[0],fren[1],fren[2],fren[3],fren[4]);
 return 0;
}
```

Przykład 14-1 Wywołanie funkcji w asemblerze z programu w języku C program `casmc.c`

```
# Funkcja pobiera kolejne elementy tablicy asm_mod_array
# dodaje do kazdego elementu 1
# Znaczenie rejestrow:
#
# %edi - indeks kolejnego elementu tablicy: 0,1,2,...,4
# %ecx - ilosc elementow tablicy
# %eax - wskaznik na poczatek tablicy
# %edx - pamiec tymczasowa
#
.section .text
.globl asm_mod_array
.type asm_mod_array, @function
asm_mod_array:
pushl %ebp                  # zabezpieczamy stary %ebp
movl %esp, %ebp             # nowy base pointer = %esp (szczyt stosu)
movl 8(%ebp),%eax           # pierwszy par. – wsk. na pocz. tablicy do %eax
movl 12(%ebp),%ecx          # rozmiar tablicy do %ecx
xorl %edi, %edi             # zerujemy indeks biezacy w %edi

start_loop:                 # start petli
cmpl %edi, %ecx             # czy koniec tablicy
je loop_exit                # skok do koniec
movl (%eax,%edi,4), %edx    # przesylamy biezacy element tablicy do %edx
addl $1, %edx               # dodaj do elementu 1
movl %edx, (%eax,%edi,4)    # nadpisz nowy element nową wartoscia
incl %edi                   # zwieksz indeks, przesuwajac się po tablicy
jmp start_loop              # skocz na poczatek petli

loop_exit:                  # zakonczenie funkcji
movl %ebp, %esp
popl %ebp
ret
```

Przykład 14-2 Plik `casma.s` zawierający funkcję `asm_mod_array`

Kompilacja:

```
$ gcc casmc.c casma.s –m32 –o casm
$./casm
./casmc
Tablica przed wywolaniem: 1 2 3 4 5
Po wywolaniu: 2 3 4 5 6
```

Przykład 14-3 Wywołanie funkcji w asm z C – wynik

14.2 Pobieranie licznika cykli procesora

Procesory Pentium posiadają licznik cykli procesora w postaci rejestru 64 bitowego. Rejestr ten może być odczytany przez instrukcję `rdtsc`. Jego zawartość będzie skopiowana do rejestrów `%edx:%eax`. Poniżej podano program napisany w assemblerze realizujący funkcję `long long rdtsc().`

```
# Funkcja pobiera licznik cykli procesora do rejestrow %edx:%eax
# unsigned long long rdtsc()
# Kompilacja: gcc program.c rdtsc32.s –m32 –o program
.text
.globl rdtsc
rdtsc:
      push %ebx
      rdtsc
      pop %ebx
ret
```

Przykład 14-4 Pobieranie zawartości licznika cykli procesora – funkcja `rdtsc`, plik `rdtsc32.s`

Gdy funkcja ma zwrócić liczbę 8 bajtową (w języku C99 typ `long long`) młodsza część powinna być w rejestrze `%eax` a starsza w rejestrze `%edx`. Aby wykorzystać funkcję w programie napisanym w C należy w linii wywołania kompilatora `gcc` podać nazwę pliku z tą funkcją czyli pliku `rdtsc32.s`.

```
#include <stdio.h>
unsigned long long rdtsc() ;
int main()
{
      printf ("%llu\n", rdtsc()) ;
      printf ("%llu\n", rdtsc()) ;
      printf ("%llu\n", rdtsc()) ;
      printf ("%llu\n", rdtsc()) ;
      return 0 ;
}
```

Przykład 14-5 Wykorzystanie funkcji `rdtsc` do pomiaru czasu wykonania instrukcji

Program kompilujemy za pomocą wywołania `gcc` jak poniżej:

```
gcc rdtsc32.s rdtsc.c –m32 -o rdtsc
```

Wykonanie programu pokazane jest poniżej:

```
./rdtsc
564236366881441
564236387985118
564236387999167
564236388005936
```

Przykład 14-6 Wykonanie programu liczącego cykle procesora

14.3 Zadania

14.3.1 Funkcja sumowania liczb – wywołanie

Poprzednio rozważano program `dodaj.s` w którym implementowano funkcję `int dodaj(int a, int b)` zwracającą sumę dwóch liczb a i b. Dokonaj implementacji tej funkcji w assemblerze jako `dodaj-asm.s`. Następnie napisz w C program `dodaj-c.c` wywołujący tę funkcję.

14.3.2 Wywoływanie funkcji z assemblerowej biblioteki z programu w języju C

W zadaniu 13.1.4 z poprzedniego rozdzału utworzylismy bibliotekę o nazwie mylib.a zawierającą napisane w assemblerze wywołania funkcji open, read, write, close i exit. Napisz program `read_write_callmylib.c` w języku c wywołujący funkcje z tej biblioteki. Biblioteka ma być przygotowana z pliku `mylib.s` jak niżej.

```
as mylib.s --32 -g -o mylib.o
ar rcsv lmylib.a mylib.o
```

Program ma wykonać następujące kroki:

- Wyprowadzić na STDOUT komunikat: `Podaj napis`
- Wczytać ze STDIN tekst
- Wyprowadzić na STDOUT wprowadzony uprzednio tekst
- Zakończyć się przez wykonanie funkcji exit(kod_zakonczenia), gdzie kod zakończenia jest długością wprowadzonego napisu.

Program ma być skompilowany przy pomocy kompilatora `gcc` jak niżej:

```
gcc read_write_callmylib.c -m32 -L. -lmylib -g -o read_write_callmylib
```

14.3.3 Wywołanie funcji power z programu w C

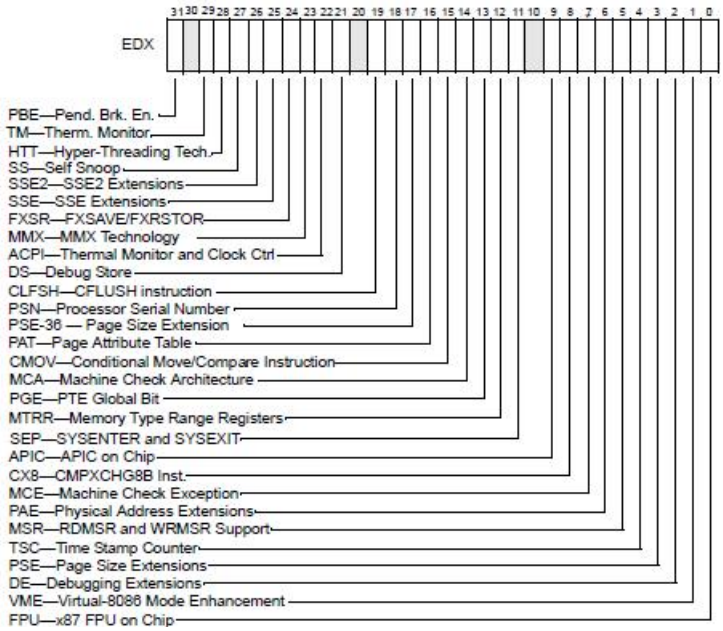
Napisz w C program wywołujący funkcję `power(int liczba, int potega)`. Funkcja na ma być napisana w assemblerze. Wykorzystaj poprzenio podany przykład.

14.3.4 Identyfikacja własności procesora

Rozkaz `cpuid` służy do identyfikacji własności procesora. Jego działanie opisane zostało w rozdziale 3, strona 317 dokumentacji [11]. W zależności od zawartości rejestru `%eax` (liczby od 0 do 3) umieszczane są w rejestrach `%eax, %ebx, %ecx, %edx` różne informacje o procesorze. Gdy `%eax = 1` zwracane są informacje dane w poniższej tabeli.

Rejestr	Opis	Parametr rejestr
`%eax`	Informacje o wersji procesora	1
`%ebx`	Informacje o wersji procesora cd, liczba rdzeni, identyfikacja kontrolera przewań	2
`%ecx`	Wyposażenie procesora rozszerzenia	3
`%edx`	Wyposażenie procesora	4

Tab. 14-1 Informacje o procesorze zwracane przez instrukcję `cpuid` dla `%eax=1`



Rys. 14-2 Informacje o wyposażeniu procesora zwracane przez instrukcję cpuid dla %eax=1 w rejestrze %edx

Napisz w assemblerze funkcję `int cpuid(int rejestr)` która w zależności od parametru `rejestr` (patrz Tab. 14-1) zwraca odpowiednie informacje o procesorze. Wywołaj tę funkcję z programu w C i wypisz wybrane parametry. Porównaj z informacjami zawartymi w pliku wirtualnym `/proc/cpuinfo`.

14.3.5 Funkcja pomiaru liczby cykli procesora rdtsc

Procesory Pentium posiadają 64 bitowy licznik cykli procesora TSC (ang. Time Stamp Counter). Rozkaz rdtsc powoduje odczytanie tego licznika i umieszczenie go w rejestrach `EDX:EAX`
Dokonaj implementacji funkcji `int rdtsc(void)` która czyta licznik cykli procesora. Przetestuj jej działania w programie w języku C. Funkcję należy wywołać przed i po pewnej operacji i obliczyć różnicę.

```
.text
.globl rdtsc
rdtsc:
      push %ebx
      cpuid
      rdtsc
      pop %ebx
ret
```

Typ `long long` w języku C99 zajmuje 8 bajtów. Specyfikacja w funkcji `printf` to `%llu` lub `%lli`.
Zbadaj dokładność pomiaru czasu w systemie Linux wykorzystując funkcję `clock_getres(...)`. Dokonaj oszacowania długości cyklu procesora wykorzystując funkcje pomiaru czasu systemu Linux. Wykorzystaj funkcje `clock_gettime(CLOCK_PROCESS_CPUTIME_ID, struct timespec *tp)`gdzie:

```
struct timespec {
        time_t   tv_sec;        /* seconds */
        long     tv_nsec;       /* nanoseconds */
};
```

Podaj jakie czynniki wpływają na dokładność pomiaru?

14.3.6 Podnoszenie do potęgi

Mamy dany program testującą funkcję podnoszenia do potęgi `int power(int base, int power).`

```
#include <stdio.h>

int power(int base, int exponent);

int main(void) {
      int result;
      result= power(2, 3);
      printf("%d\n", result);
      return result;
}
```

Przykład 14-7 Program `c_power_test.c` testowania funkcji podnoszenia do potęgi

Funkcja power napisana w C dana jest w poniższym pliku.

```
int power(int base, int exponent)
{
      int result= 1;
      int i;
      for(i= 0; i < exponent; i++) {
            result = result*base;
      }
      return result;
}
```

Przykład 14-8 Program c_power.c z funkcją power podnoszenia do potęgi

Aby otrzymać program wykonywalny, kompilujemy go jak poniżej:

```
gcc c_power_test.c c_power.c –m32 -o power
```

Wzorując się na przykładzie funkcji `asm_sum.s` Napisz funkcję `asm_power.s` który implementuje funkcję power w asemblerze. Kopilacja będzie przebiegała jak niżej:

```
gcc c_power_test.c asm_power.s –m32 -o power
```

14.3.7 Sumowanie elementów tablicy

Poniżej dano program `sum-tab.c w C` testujący szybkość operacji dodawania.

```
// Kompilacja:
// gcc sum-tab.c rdtsc32.s –m32 –O 0 –o sum-tab
#include <stdio.h>
#include <stdlib.h>

#define MAX 100000
int tab[MAX];
unsigned long long rdtsc() ;

int main(void) {
  int i,rd;
  int suma = 0;
  long long t1,t2;
  printf("test szybkosci C tablica %d el\n",MAX);
  for(i=0;i<MAX;i++) tab[i] = (int) (10.0*rand()/(RAND_MAX+1.0));
  // Test szybkosci
  t1 = rdtsc();
  for(i=0;i<MAX;i++) suma = suma + tab[i];
  t2 = rdtsc();
  t2 = t2 -t1;
  printf("suma: %d czas: %llu\n",suma,t2);
  return 0;

}
```

Przykład 14-9 Program `sum-tab.c` w C testujący szybkość operacji dodawania

Wykorzystując powyższy przykład i podany wczesniej program `casmc.c` napisz program `sum-tab-asm.c i sum-tab-asm.a` sumowania elementów tablicy. Funkcja sumowania `int sum_tab(int tab[],int size)` wywoływana w programie `sum-tab-asm.c w` C ma być napisana w assemblerze i umieszczona w pliku `sum-tab-asm.a`. Wykorzystując funkcję `rdtsc` porównaj czasy wykonania programów sumowania elementów tablicy. W jednym funkcja sumowania ma być napisana w języku C w drugim w assemblerze. Zwiększ istotnie wymiar tablicy (np. do 100000).

15. Liczby zmiennoprzecinkowe

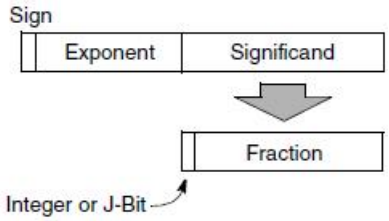
15.1 Informacje ogólne

Zapis liczb w formacie stałoprzecinkowym posiada istotną wadę. Polega ona na tym, że zapis bardzo małych i badzo dużych liczb wymaga dużej liczby znaków. Przykładowo jeżeli chcemy zapisać liczbę 123 000 000 000 potrzebujemy aż 12 znaków. W formacie zmiennoprzecinkowym liczbę tę można zapisać jako $123 * 10^9$ co w notacji inżynierskiej ma postać 123e9 i zajmuje tylko 5 znaków. Podobnie wyglądałby zapis bardzo małej liczby np. 0.000 000 000 123 którą zapisać można jako 123e-12. Należy zauważyć że liczby odzwierciedlają zazwyczaj różne wielkości fizyczne, a te mierzone są ze skończoną i niezbyt dużą dokładnością, np. trzech cyfr znaczących. Zapis liczby L w formacie zmiennoprzecinkowym ma postać: $L = c * p^w$ gdzie c jest tak zwaną mantysą, p jest podstawą systemu liczenia (zwykle 10 lub 2) a w jest wykładnikiem zwanym też cechą.

Podane tu informacje dotyczące liczb zmiennoprzecinkowych w architekturze IA-32 pochodzą z pracy [10]. Liczby zmiennoprzecinkowe pokrywają zakres od -∞ do +∞. Jako że każda reprezentacja maszynowa tych liczb zajmuje tylko skończoną liczbę bitów, nie wszystkie liczby rzeczywiste mogą mieć swą reprezentację maszynową. Sposób zapisu liczb rzeczywistych opisany jest w standardzie IEEE 754 a także w [10]. W tym standardzie liczba zmiennoprzecinkowa zapisana jest w postaci trzech składników: znaku, mantysy i wykładnika.

znak	*sign*	0 – gdy liczba dodatnia lub zero 1 – gdy liczba ujemna
mantysa	*significant*	1 bit części całkowitej (J-bit) zazwyczaj pomijany ułamek (ang. *fraction*)
wykładnik lub cecha	*exponent*	Liczba binarna reprezentująca potęgę 2 przez którą to liczbę należy pomnożyć mantysę

Tab. 15-1 Składniki liczby zmiennoprzecinkowej

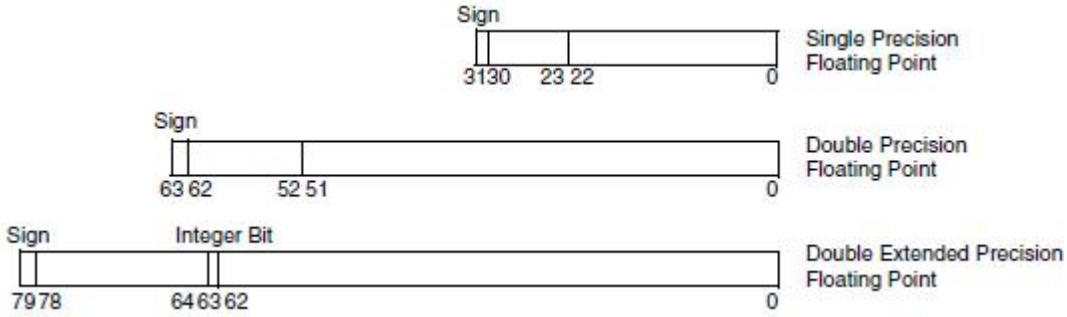


Rys. 15-1 Składniki liczby zmiennoprzecinkowej

Przykład zapisu liczby 178.125 podaje poniższa tabela.

Notation	Value		
Ordinary Decimal	178.125		
Scientific Decimal	$1.78125E_{10}2$		
Scientific Binary	$1.0110010001E_2111$		
Scientific Binary (Biased Exponent)	$1.0110010001E_210000110$		
IEEE Single-Precision Format	Sign	Biased Exponent	Normalized Significand
	0	10000110	01100100010000000000000 1. (Implied)

Tab. 15-2 Przykład zapisu liczby 178.125 w formacie IEEE 754 pojedynczej precyzji



Rys. 15-2 Formaty liczb zmiennoprzecinkowych w IA-32

15.2 Wykładnik

W architekturze IA-32 liczby zmiennoprzecinkowe zapisywane są z przesunietym (ang. *biased*) wykładnikiem (eksponentem). Znaczy to tyle że do wykładnika dodawana jest pewna stała, tak że ma on zawsze wartość dodatnią. Wielkość dodawana zależy od tego ile miejsc przeznaczono na wykładnik co pokazuje poniższa tabela.

Typ	Znak	Wykładnik	Mantysa	Długość całkowita	Przesunięcie wykładnika	Liczba cyfr dziesiętnych
Single IEEE 754	1	8	23	32	127	~7.2
Double IEEE 754	1	11	52	64	1023	~15.9
Double extended (x87 FPU)	1	15	64	80	16383	~19.2

Tab. 15-3 Formaty liczb zmiennoprzecinkowych w architekturze IA-32

Przesunięty format wykładnika stosuje się po to by wykluczyć operowanie na liczbach ze znakiem. Dla liczb pojedynczej precyzji nieprzesuniety wykładnik jest z zakresu –126...127. Po dodaniu przesunięcia (ang. *bias*) 127, wykładnik jest w zakresie 1..254.

15.2.1 Zera ze znakiem

Zero może być ze znakiem –0 lub +0 jak pokazuje poniższy rysunek. Obydwa zakodowania posiadają jednakową wartość.

15.3 Normalizacja liczb

Naturalna jest tendencja do zapisywania liczb w optymalny sposób. Znaczy to tyle że dysponując określoną liczbą miejsc na zapis mantysy i wykładnika należy dążyć do jak największej dokładności. W większości przypadków mantysa występuje w postaci znormalizowanej. Znaczy to, że z wyjątkiem zera, liczba ułamkowa przedstawiana jest w postaci: `1.ffff...f` gdzie f jest 0 albo 1. Dla liczb mniejszych od 1 wiodące zera są eliminowane (dla każdego wyeliminowanego zera zmiejsza się wykładnik o 1). Pomijane jest wiodące 1. Znormalizowana liczba rzeczywista składa się więc ze znaku, znormalizowanej mantysy z zakresu od 1 do 2 i wykładnika który specyfikuje położenie przecinka.

15.3.1 Nieznormalizowane i znormalizowane liczby skończone

Liczby skończone dzieli się na znormalizowane i nie znormalizowane. Kiedy liczba zbliża się do zera, wykładnik dochodzi do kresu i może już nie odzwierciedlić tak bliskich zeru liczb. Jest tak dlatego, że wykładnik ma skończoną liczbę cyfr i nie może odzwierciedlić kolejnych przesunięć (by w mantysie nie było na początku zer). Wtedy w mantysie z lewej strony pojawiają się zera. W liczbie znormalizowanej wykładnik jest z zakresu 1...254. W liczbie nie znormalizowanej wykładnik wynosi zero, Gdy w wyniku działań powstaje liczba nieznormalizowana mamy do czynienia z niedomiarem (ang. *underflow*) co może powodować wyjątek.

Operation	Sign	Exponent*	Significand
True Result	0	–129	1.01011100000...00
Denormalize	0	–128	0.10101110000...00
Denormalize	0	–127	0.01010111000...00
Denormalize	0	–126	0.00101011100...00
Denormal Result	0	–126	0.00101011100...00

NOTE:
* Expressed as an unbiased, decimal number.

Tab. 15-4 Proces denormalizacji

15.3.2 Nieskończoności ze znakiem

Wielkości -∞ i +∞ reprezentują najmniejszą i największą liczbę rzeczywistą. W formacie IEEE 754 nieskończoności reprezentowane są jako zapisy w których wykładnik wynosi 255 a mantysa 1.00...0 (wiodąca jedynka jest pomijana.

Arytmetyka liczb nieskończonych jest zawsze dokładna. Wyjątki są generowane gdy użycie nieskończoności jako argumentu daje nieprawidłową operację.

15.3.3 Symbole NaNs

Symbole NaNs pojawiają się gdy trzeba reprezentować wynik który nie jest liczbą (ang. *Not a Number*). Architektura IA-32 definiuje dwa rodzaje nie-liczb NaNs:

- SNaNs – nie liczby sygnalizujące (ang. Signalling NaNs)
- QNaNs – nie liczby nie sygnalizujące (ang. Quiet NaNs)

QNaNs mają jedynkę jako najbardziej znaczącą cyfrę mantysy, w SNaNs najbardziej znacząca cyfra mantysy wynosi zero.

NOTES:
1. Integer bit of fraction implied for single-precision floating-point format.
2. Fraction must be non-zero.
3. Sign bit ignored.

QNaNs może się pojawiać i propagować jako wynik operacji arytmetycznych. Pojawienie się SNaNs generuje wyjątek. SNaNs nie może być wynikiem operacji arytmetycznej i musi być generowane przez oprogramowanie by spowodować wyjątek. Generowanie wyjątków opisane jest poniższą regułą:

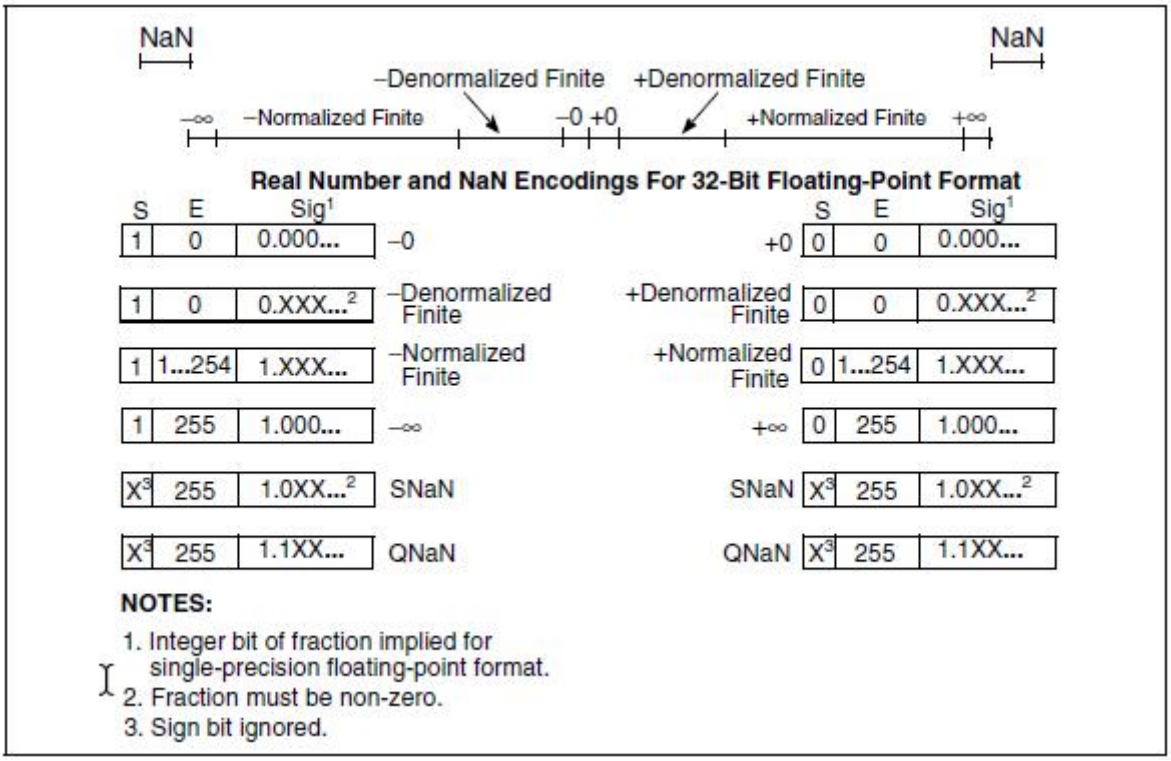
- Jeżeli jednym z operandów jest SNaNs i wyjątek nie jest zamaskowany wynik nie powstaje i generowany jest wyjątek.
- Jeżeli jednym z operandów jest SNaNs i wyjątek jest zamaskowany wynikiem jest QNaNs i wyjatek nie jest generowany.

15.4 Kodowanie nie liczb

W normie IEEE 754 definiuje się dodatkowo kodowanie wielkości specjalnych i nie będących liczbami. Tak więc kodowanie liczb zmiennoprzecinkowych łącznie obejmuje:

- Znormalizowne liczby skończone - normalized finite numbers
- Nieznormalizowne liczby skończone - denormalized finite numbers
- Zera ze znakiem – signed zeros
- Nieskończoności ze znakiem - signed infinities

- NaNs - not a numbers



Rys. 15-3 Liczby rzeczywiste 32 bitowe i NaNs

15.5 Zaokrąglanie

Wynik operacji zmiennoprzecinkowej może być precyzyjny, ale z powodu ograniczonej liczby bitów przeznaczonych na reprezentację liczby, nie może on być dokładnie zapisany. Wtedy musza być stosowane zaokrąglenia (opisane między innymi w https://pl.wikipedia.org/wiki/Zaokrąglanie). Przykładowo liczba a nie może być zakodowana jako liczba float pojedynczej precyzji gdyż dla mantysy przewidziano 23 bity a liczba ma 24 bity.

$(a)\ 1.0001\ 0000\ 1000\ 0011\ 1001\ 011\underline{1}E_2\ 101$

Aby zakodować tę liczbę procesor wytworzy dwie liczby b i c które są najbliższe liczbie a (b< a < c).

$(b)\ 1.0001\ 0000\ 1000\ 0011\ 1001\ 011E_2\ 101$

$(c)\ 1.0001\ 0000\ 1000\ 0011\ 1001\ 100E_2\ 101$

Zaokrąglenie wprowadzi błąd który jest mniejszy niż wartość ostatniej cyfry jej zapisu. Standard IEEE 754 wprowadza cztery rodzaje zaokragleń:

- Zaokrąglenie do najbliższej liczby
- Zaokrąglenie w dół
- Zaokrąglenie w górę
- Zaokrąglenie w kierunku 0

Rounding Mode	RC Field Setting	Description
Round to nearest (even)	00B	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (that is, the one with the least-significant bit of zero). Default
Round down (toward −∞)	01B	Rounded result is closest to but no greater than the infinitely precise result.
Round up (toward +∞)	10B	Rounded result is closest to but no less than the infinitely precise result.
Round toward zero (Truncate)	11B	Rounded result is closest to but no greater in absolute value than the infinitely precise result.

Tab. 15-5 Rodzaje zaokrągleń w IA-32 i ich zakodowanie w polu RC (Roudnig Control) rejestru sterującego x87

Sposób zaokrąglenia ustalany jest przez wpis do pola RC (ang. *Roudnig Control*) które jest częścią rejestru sterującego koprocesora x87 FPU lub występuje w jednostce SSE lub SSE2.

15.6 Wyjątki zmiennoprzecinkowe

Nie wszystkie operacje na argumentach zmiennoprzecinkowych dadzą się wykonać. Przykładem jest chociażby próba dzielenia przez zero. Sytuacja taka prowadzi do generowania tak zwanego wyjątku. Skutkiem powstania wyjatku może być:

- Uruchomienie kodu nazywanego przerwaniem wewnetrznym procesora (ang. *Exception handler*)
- Ustawienie określonych znaczników w rejestrze procesora i zwrócenie charakterystycznej wielkości jako wynik operacji.

Jak w takiej sytuacji może zareagować procesor i programista? Znowu dostepne są dwie możliwości:

- Napisanie specjanej funkcji uruchamianej gdy wystapi wyjatek (ang. *signal handler*).
- Po kazdej operacji zmiennoprzecinkowej testowanie wyniku i ewentualnie rejestru statusowego jednostki FPU.

W architekturze IA-32 wyróżnia się sześć rodzajów wyjątków zmiennoprzecinkowych:

Opis	Description	Kod wyjątku
Nieprawidłowa operacja	*Invalid operation*	#I
Dzielenie przez zero	*Divide-by-zero*	#Z
Nieznormalizowany operand	*Denormalized operand*	#D
Nadmiar numeryczny	*Numeric overflow*	#O
Niedomiar numeryczny	*Numeric underflow*	#U
Niedokładny wynik	*Inexact result*	#P

Tab. 15-6 Rodzaje wyjątków zmiennoprzecinkowych w architekturze IA-32

W architekturze IA-32 operacje zmiennoprzecinkowe wykonywane są przez koprocesor zmiennoprzecinkowy x87 FPU. Programista ma wpływ na funkcjonowanie tej jednostki poprzez możliwość wpisu do rejestru sterujacego i możliwość odczytu z rejestru statusowego. Wpisując określone wartości do rejestru sterującego można wpływać na:

- Rodzaj zaokragleń
- Maskowanie wyjątków

Z rejestru statusowego odczytać można informację o błędach jednostki w szczególności o tym czy i który wyjątek wystapił.

Każdy z wymienionych wyjątków ma swoją flagę IE, ZE, DE, OE, UE, PE i swoja maskę IM, ZM, DM, OM, UM, PM. Maski sa elementem rejestru sterującego jednostki x87 FPU, a flagi sa elementem rejestru jej statusu.

15.6.1 Wyjątek nieprawidłowa operacja (#I)

Wyjątek nieprawidłowa operacja (ang. *Invalid operation*) powstaje gdy jeden lub więcej operand jest nieprawidłowy. Gdy wyjątek jest zamaskowany (IM) procesor ustawia flagę IE a jako wynik podana będzie wartość QNaN i nadpisze rejestr przeznaczenia. Gdy wyjątek nie jest zamaskowany, w rejestrze statusu ustawiana jest flaga IE i wywoływany jest handler wyjątku. Wyjątek powstaje gdy operand jest nieprawidłowy np. dzielimy ∞ przez ∞.

15.6.2 Wyjątek nieznormalizowany operand (#D)

Wyjątek nieznormalizowany operand powstaje gdy instrukcja próbuje operować na nieznormalizowanym operandzie. Gdy wyjatek jest zamaskowany (DM) procesor ustawia flagę DE i instrukcja będzie wykonana. Gdy wyjątek nie jest zamaskowany, ustawiana jest flaga DE i wywoływany jest handler wyjątku.

15.6.3 Wyjątek dzielenie przez zero (#Z)

Wyjątek ten postawnie gdy procesor ma podzielić skończoną wartość przez zero. Gdy wyjątek jest zamaskowany (ustawiona flaga ZM) ustawiana jest flaga ZE i zwracana jest nieskończoność ze znakiem będącym XOR znaków operandów. Gdy wyjątek nie jest zamaskowany ustawiana jest flaga ZE i wywoływany jest handler wyjątku.

15.6.4 Wyjątek numeryczny nadmiar (#O)

Wyjątek numeryczny nadmiar powstaje gdy zaokraglenie wyniku operacji przekracza maksymalną wartość dającą się zapisać w polu wyniku. Wielkości maksymalne dla poszczególnych typów pokazuje poniższa tabela.

Floating-Point Format	Overflow Thresholds
Single Precision	$\lvert x \rvert \geq 1.0 * 2^{128}$
Double Precision	$\lvert x \rvert \geq 1.0 * 2^{1024}$
Double Extended Precision	$\lvert x \rvert \geq 1.0 * 2^{16384}$

Tab. 15-7 Maksymalne wielkości typów zmiennoprzecinkowych w IA-32

Gdy wyjątek jest zamaskowany (ustawiona flaga OM) ustawiana jest flaga OE i zwracany jest wynik zależny od przyjetego trybu zaokrąglania zgodnie z poniższą tabelą.

Gdy wyjątek nie jest zamaskowany ustawiana jest flaga OE i wywoływany jest handler wyjątku a wynik pozostaje bez zmiany.

15.6.5 Wyjątek numeryczny niedomiar (#U)

Wyjątek numeryczny niedomiar powstaje gdy wyniku operacji jest mniejszy niż dająca się zapisać znormalizowana wartość operandu wynikowego. Wartości te pokazuje poniższa tabela.

Floating-Point Format	Underflow Thresholds
Single Precision	$\lvert x \rvert < 1.0 * 2^{-126}$
Double Precision	$\lvert x \rvert < 1.0 * 2^{-1022}$
Double Extended Precision	$\lvert x \rvert < 1.0 * 2^{-16382}$

Tab. 15-8 Minimalne wartości typów zmiennoprzecinkowych

Sposób traktowania wyjatku niedomiaru zależy także od:

- Jak mała jest liczba
- Czy liczba jest niedokładna

Gdy wyjątek #U jest zamaskowany jest on zgłaszany przez ustawienie flagi UE tylko wtedy gdy wynik jest jednocześnie mały i niedokładny. Procesor zwraca nieznormalizowany wynik do operandu docelowego niezależnie od niedokładności.

Gdy wyjątek #U jest nie jest zamaskowany wyjatek jest zgłaszany gdy wynik jest mały niezależnie od niedokładności. Operandy pozostaja niezmienione i wywoływany jest handler wyjątku.

15.7 Typowe akcje podejmowane przez handler wyjątku

Wyjątki jednostki zmiennoprzecinkowej są obsługiwane w taki sam sposób jak inne wyjatki. Handler wyjątku jest elementem systemu operacyjnego i może także wywoływać handler dostarczony przez programiste jako handler obsługi sygnału.

15.8 Obsługa wyjątków w języku C

Użycie liczb zmiennoprzecinkowych w języku C może stwarzać problemy. Jeżeli program generuje wyjatek nadmiar, niedomiar czy dzielenie przez 0 wynik może być nieskończony bądź nie liczbowy. Jeżeli te niezdefiniowane wartości będą użyte w innych obliczeniach powstaną kolejne nieokreslone wyniki. Trudno wtedy znaleźć błąd.

Pomocne jest użycie funkcji `feenableexcept`. Wtedy gdy powstanie wyjątek będzie wygenerowany sygnał SIGFPE. Demonstruje to ponizszy przykład.

```
#define _GNU_SOURCE
#include <fenv.h>

int main(void) {
    feenableexcept(FE_INVALID   |
                   FE_DIVBYZERO |
                   FE_OVERFLOW  |
                   FE_UNDERFLOW);
    float a = 1., b = 0.;
    float c = a/b;
    return 0;
}
```

Przykład 15-1 Program wyjatek1.c
Program kompiluje się jak poniżej, zwracając uwagę na dołączenie biblioteki matematycznej (-lm).

```
gcc wyjatek1.c -lm -o wyjatek1
```

Gdy program zostanie uruchomiony spowoduje on wygenerowanie sygnału SIGFPE i dalej zakończenie procesu. Można także zainstalować własny handler obsługi sygnału SIGFPE co pokazano poniżej.

```
// Demonstracja przechwycenia sygnału SIGFPE
// kompilacja:
// gcc fpe.c -m32 -lm -o fpe
//
#define _GNU_SOURCE
#include <fenv.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int fperr = 0;
```

```
void handler(int sig) {
       printf("Sygnal %d\n",sig);
       fperr = 1;
       exit(0);
       return;
}

int main(void) {
    float a = 1., b = 0.;
    float c;
    feenableexcept(FE_INVALID   |  FE_DIVBYZERO |
                   FE_OVERFLOW  |  FE_UNDERFLOW);
    signal(SIGFPE,handler);
    c = a/b;
    sleep(1);
    printf("fperr: %d\n",fperr);
    return fperr;
}
```

Przykład 15-2 Przechwycenie sygnału SIGFPE

Funkcja `feenableexcept` jest zawarta w bibliotece glibc 2.2 i nowszych. Aby jej użyć należy dołączyć makro `_GNU_SOURCE.` Standard c99 definiuje także inne funkcje obsługi zmiennego przecinka.

Inne funkcje dotyczące obsługi zmiennego przecinka w C zdefiniowane są w pliku nagłówkowym <fenv.h>

```
int feclearexcept(int excepts);
int fegetexceptflag(fexcept_t *flagp, int excepts);
int feraiseexcept(int excepts);
int fesetexceptflag(const fexcept_t *flagp, int excepts);
int fetestexcept(int excepts);
int fegetround(void);
int fesetround(int rounding_mode);
int fegetenv(fenv_t *envp);
int feholdexcept(fenv_t *envp);
int fesetenv(const fenv_t *envp);
int feupdateenv(const fenv_t *envp);
```

16. Jednostka zmiennoprzecinkowa X87 FPU

16.1 Środowisko wykonawcze X87 FPU

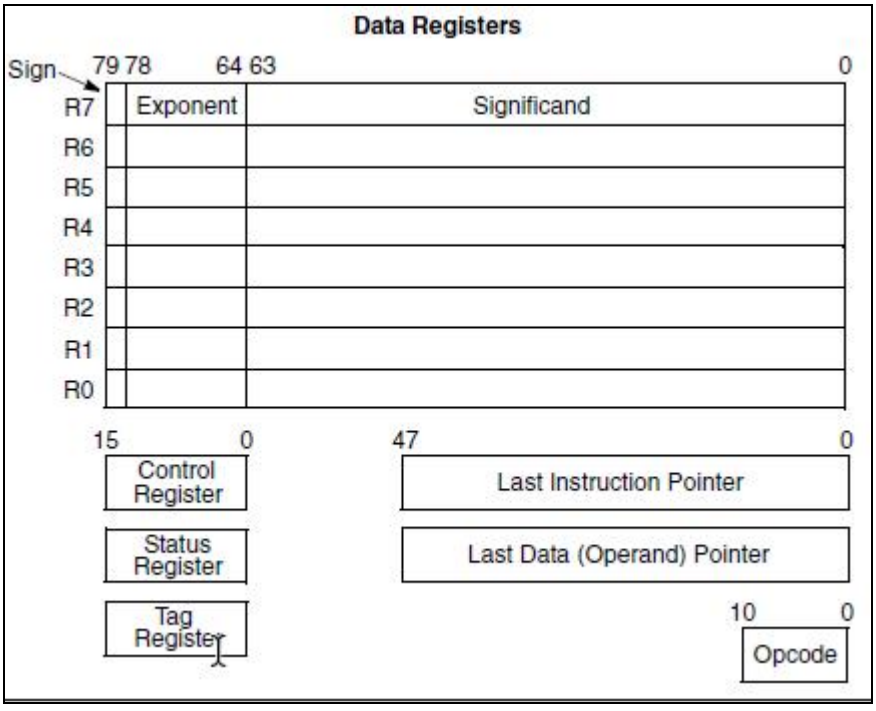
Jednostka zmiennoprzecinkowa X87 FPU jest elementem architektury IA-32 i procesorów Pentium. Poczatkowe procesory firmy Intel posiadały koprocesor matematyczny jako oddzielny układ (koprocesor) z własną listą rozkazów. Były to układy 8087, 80287, i387 dla procesorów 8086, 80286, i386. Począwszy od układu i486 koprocesor zmiennoprzecinkowy został zintegrowany z koprocesorem i stał się elementem architektury IA-32. Definiuje się tam zbiór instrukcji oznaczany jako x87. Zbiór instrukcji x87 implementuje standard liczb zmiennoprzecinkowych IEEE 754. Z racji tego że poczatkowo była to oddzielna jednostka nie ma bezposredniej komunikacji z rejestrami procesora x86 a komunikacja zachodzi przez pamięć. Wraz ze wprowadzeniem procesora Pentium 4 wprowadzono jednostkę SSE2 która może wykonywać operacje na liczbach zmiennoprzecinkowych pojedynczej i podwójnej precyzji (ale nie rozszerzonej). Niemniej jednak jednostka x87 jest domyslną jednostką przetwarzania liczb zmiennoprzecinkowych dla kompilatora gcc.

Jednostka x87 FPU opisana jest w dokumentacji [10] . Zawiera ona następujące rejestry:

- Rejestr statusowy
- Rejestr sterujący
- Rejestr znacznikowy słowa (ang. *The tag word register*)
- Rejestr wskaźnika ostatniej instrukcji
- Rejestr wskaźnika ostatniego operandu
- Rejestr kodu ostatniej instrukcji

16.1.1 Stos i rejestry danych

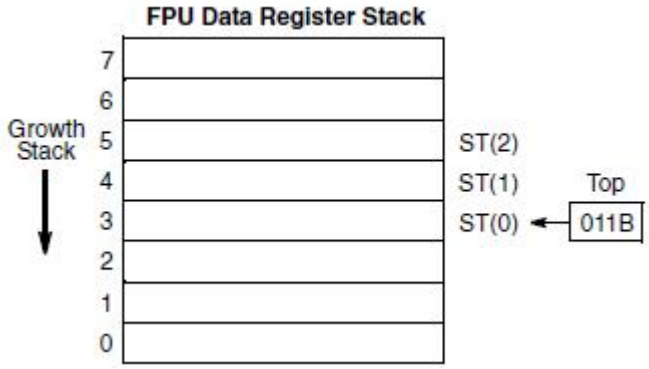
Dodatkowo jednostka zawiera osiem 80 bitowych rejestrów danych tworzących stos. Rejestry te nakładają się z rejestrami jednostki MMX, stąd jednostka MMX i procesor FPU nie mogą być używane jednocześnie. Rejestry jednostki pokazuje poniższy rysunek.



Rys. 16-1 Rejestry jednostki zmiennoprzecinkowej X87 FPU

Jednostka domyślnie przetwarza liczby zmiennoprzecinkowe w 80 bitowym formacie double extended. Gdy do rejestrów ładowane są liczby zmiennoprzecikowe w innych formatach, są one automatycznie konwertowane na format double extended. Kiedy wyniki są przesyłane z powrotem do pamieci z rejestrów X87 FPU mogą pozostać w formacie double extended lub być konwertowane na formaty zmiennoprzecinkowe mniej dokładne jak 32 bitowy lub 64 bitowy lub też format całkowity integer 32 bitowy lub upakowany BCD.

Rejestry danych są uformowane w stos od ST(0) do ST(7). Jego szczyt wskazuje 3 bitowe pole TOP zawarte w rejestrze statusu. W terminologii FPU operacja `load` równoważna jest operacji `push`, zmiejsza wskaźnik stosu o 1 i ładuje tam nową wartość. Operacja `store` równoważna jest operacji `pop` i pobiera ze szczytu stosu ST(0) wartość, przesyła ją do miejsca przeznaczenia i zwiększa wskaźnik TOP o jeden.



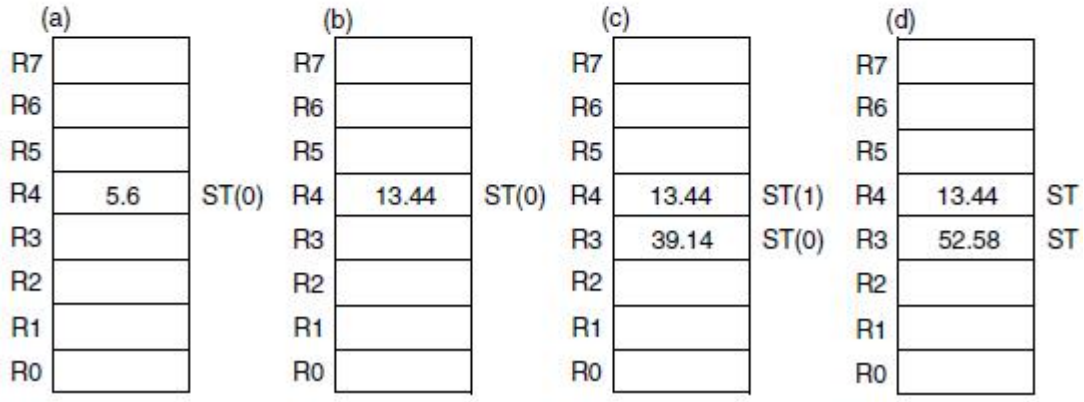
Rys. 16-2 Stos jednostko X87 FPU

Gdy operacja `load` wykonywana jest gdy TOP=0, to „przekręca” się on na wartość 7 i generowany jest wyjątek „przepełnienie stosu zmiennoprzecinkowego” (ang. *floating-point stack overflow*). Podobna sytuacja występuje gdy dla TOP=7 wykonujemy operację `store`. Poszczególne rejestry stosu oznaczane są jako ST(i) gdzie 0<i<7. Oznaczenie ST odnosi się do szczytu stosu ST(0). Przykładowo gdy TOP=3 rozkaz `FADD ST, ST(2)` powoduje dodanie rejestrów R3 i R5. Przykład dodania dwóch iloczynów 5.6 * 2.4 + 3.8 * 10.3 podaje praca [10].

Computation
Dot Product = (5.6 x 2.4) + (3.8 x 10.3)

Code:

```
FLD  value1 ;(a) value1=5.6
FMUL value2 ;(b) value2=2.4
FLD  value3 ; value3=3.8
FMUL value4 ;(c)value4=10.3
FADD ST(1)  ;(d)
```



Rys. 16-3 Przykład wykonania operacji 5.6 * 2.4 + 3.8 * 10.3

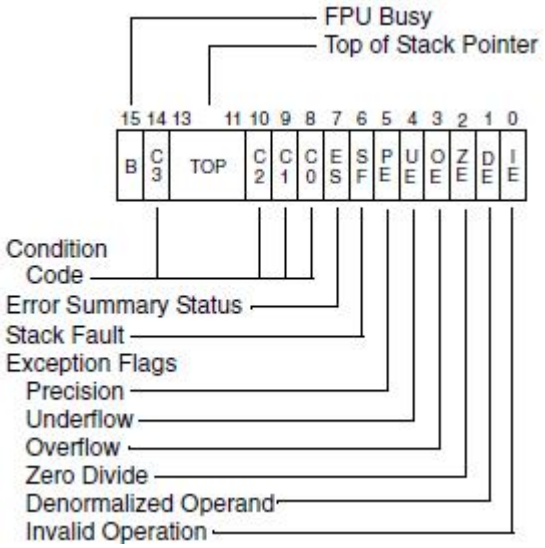
Operacja przebiega jak następuje:

rozkaz	opis	Wariant z Rys. 16-4
`FLD value1`	TOP=TOP-1; załaduj value1 do ST(0)	a
`FMUL value2`	Pomnóż ST(0) przez value2 (z pamięci), wynik w ST(0)	b
`FLD value3`	TOP=TOP-1; załaduj value3 do ST(0), poprzednia wartość w ST(1)	c
`FMUL value4`	Pomnóż ST(0) przez value4 (z pamięci), wynik w ST(0)	d
`FADD ST(1)`	Dodaj ST(0) i ST(1), wynik w ST(0)	

Przykład 16-1 Operacja 5.6 * 2.4 + 3.8 * 10.3

16.1.2 Rejestr statusowy

Jednostka X87 FPU posiada 16 bitowy rejestr statusowy co pokazano poniżej.



Rys. 16-5 Rejestr statusowy jednostko X87 FPU

Rejestr ten odzwierciedla status jednostki a w szczególności:

- Położenie TOP wierzchołka stosu FPU, bity 11,12,13
- Flagi warunku C0-C3 (ang. *condition code flag*)
- Flaga podsumowania błędu (ang. *error summary status flag*)
- Flagi wyjątków
- Flaga zajętości jednostki FPU

Flagi warunku C0-C3 zawierają informacje o wyniku ostatniej operacji i używane są przy instrukcjach warunkowych co opisano w dokumentacji [10].

Flagi wyjątków IE, DE, ZE, OE, UE, PE są ustawiane, gdy wystąpi jeden z wyjątków zmiennoprzecinkowych opisanych wczesniej. Wyjątki te mogą być maskowane flagami IM, ZM, DM, OM, UM, PM. Maski są elementem słowa sterującego jednostki x87 FPU.

Flaga SF jest ustawiana gdy wystąpi przepełnienie (w górę lub dół) stosu FPU.

Ogólna flaga wyjątku ES ustawiana jest gdy wystapi jakikolwiek wyjątek.

Bit 15, flaga B odpowiada fladze ES i jest używana dla zachowania kompatybilności.

Opis	Description	Flaga wyjątku
Ogólna flaga wyjątku	Error summary flag	ES
Przepełnienie stosu	Stack overflow / underflow	SF
Nieprawidłowa operacja	Invalid operation	IE
Dzielenie przez zero	Divide-by-zero	ZE
Nieznormalizowany operand	Denormalized operand	DE
Nadmiar numeryczny	Numeric overflow	OE
Niedomiar numeryczny	Numeric underflow	UE
Niedokładny wynik	Inexact result	PE

Tab. 16-1 Flagi wyjątków jednostki FPU w architekturze IA-32

16.1.3 Instrukcje rozgałęzień i warunkowe

Procesory architektury IA-32 wspierają dwa mechanizmy instrukcji skoków i warunkowych nazywane mechanizmem starym i nowym. Mechanizm stary użyty jest w procesorach starszych niż P6 Pentium. Polega on na wykonaniu dwóch kroków:

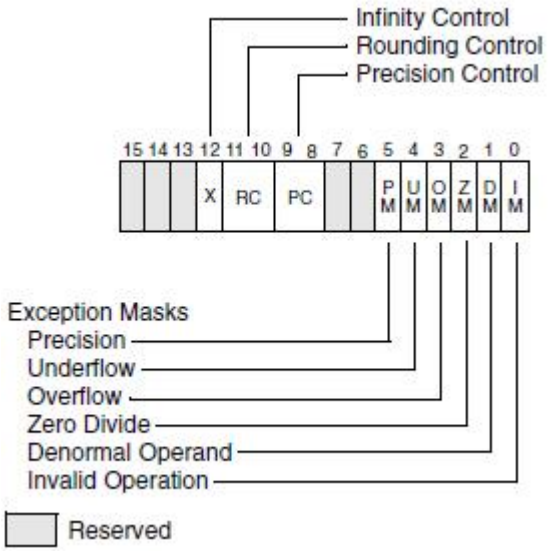
- Instrukcja FSTSW AX przenosi zawartość rejestru statusu FPU do rejestru EAX
- Instrukcja SAHF kopiuje 8 młodszych bitów z rejestru EAX do rejestru flag EFL

Kiedy flagi rejestru EFL są prawidłowe, mogą być wykonywane zwykłe instrukcje skoku.

W mechanizmie nowym instrukcje FCOMI, FCOMIP, FUCOMI, FOCOMIP ustawiają bezpośrednio flagi rejestru EFL i mogą być wykonywane zwykłe instrukcje warunkowe i skoku.

16.1.4 Słowo sterujące

Słowo sterujące jednostki X87 FPU ma 16 bitów. Ustala ono precyzję obliczeń, tryb zaokrągleń i maski wyjątków.



Rys. 16-6 Słowo sterujące jednostki X87 FPU

Maski wyjątków PM, UM, OM, ZM, DM, IM mogą być ustawione na 1, wtedy odpowiednie wyjątki wymienione w Tab. 15-6 nie są zgłaszane.

Flagi PC kontroli precyzji obliczeń służą do ustawiania precyzji obliczeń. Domyślnie ustawiana jest precyzja 80 bitowa double extended czyli flagi PC na 11B co pokazano poniżej.

Precision	PC Field
Single Precision (24-Bits)	00B
Reserved	01B
Double Precision (53-Bits)	10B
Double Extended Precision (64-Bits)	11B

Tab. 16-2 Flagi kontroli precyzji obliczeń

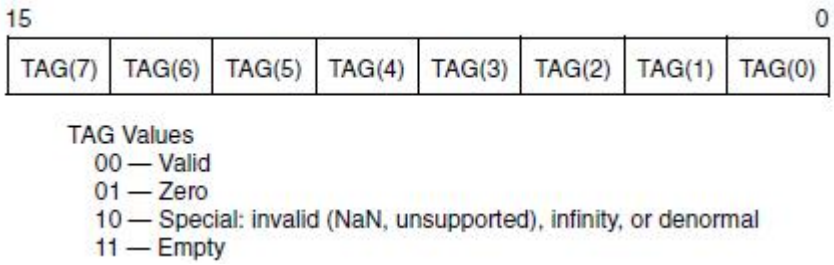
Flagi RC ustalają sposób wykonywania zaokragleń co pokazane jest w Tab. 15-5.

Flaga X kontroli nieskończoności (ang. Infinity Control) stosowana jest do zapewnienia zgodności z wcześniejszymi rozwiązaniami.

16.1.5 Rejestr znaczników tag register

16 bitowy rejestr znaczników (ang. *tag register*) opisuje zawartość 8 rejestów danych stosu. Zawartość rejestru może być następująca:

- 00 - prawidłowa,
- 01 – zero
- 10 – specjalna : NaN, nieskończoność, nieznormalizowana
- 11 - pusty



Rys. 16-7 Rejestr znaczników jednostki FPU

16.1.6 Wskaźniki ostatniej instrukcji i danych

Jednostka X87 FPU zawiera dwa 48 bitowe rejestry:

- Wskaźnik ostatniej instrukcji
- Wskaźnik ostatnich danych (operandu)

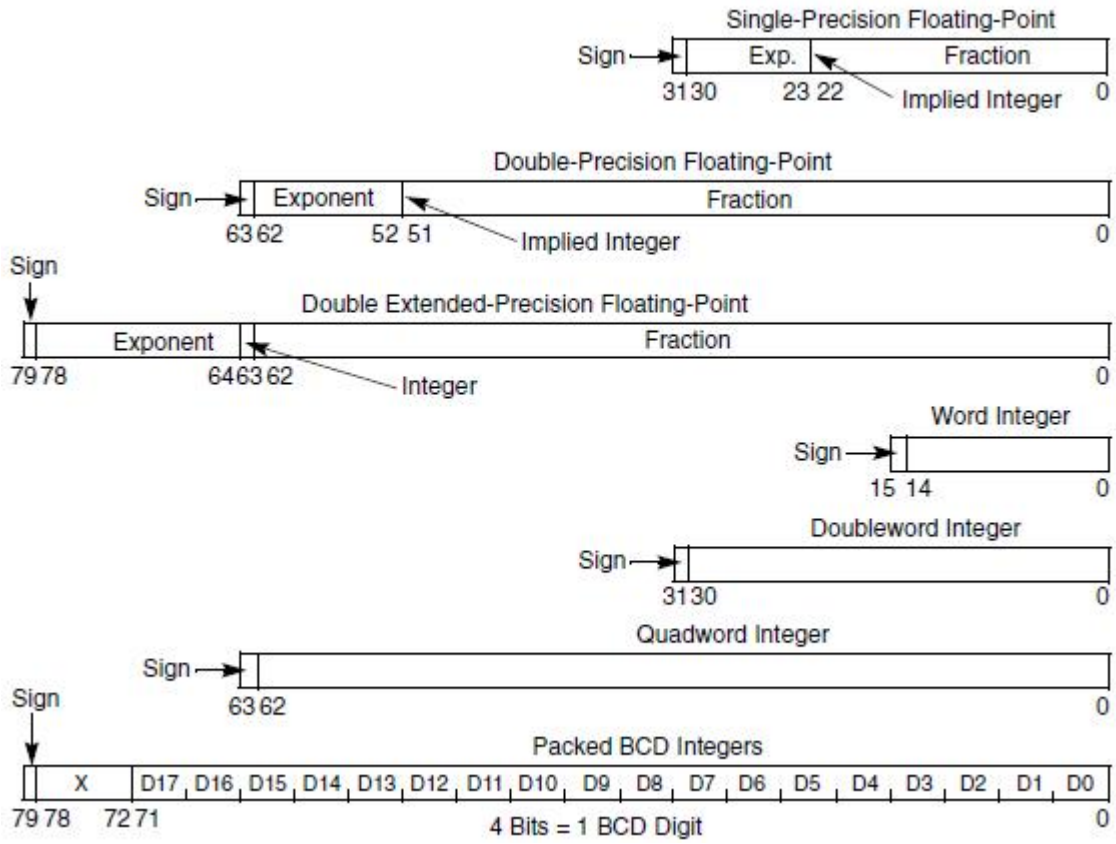
Informacje te wykorzystywane są przez handlery wyjątków.

16.2 Typy danych używane przez jednostkę X87 FPU

Jednostka X87 FPU używa nastepujące typy danych:

Nazwa	Angielska	Deklaracja w GAS	Bitów
Zmiennoprzecinkowe pojedynczej precyzji	Single precision float	`.float` `.single`	32
Zmiennoprzecinkowe podwójnej precyzji	Double precision float	`.double`	64
Zmiennoprzecinkowe podwójnej rozszerzonej precyzji	Extended double precision float	`.tfloat`	80
Słowo ze znakiem	Signed word	`.word`	16
Podwójne słowo ze znakiem	Signed doubleword integer	`.long` `.int`	32
Poczwórne słowo ze znakiem	Signed quadword integer	`.quad`	64
Spakowana w BCD liczba całkowita	BCD packed integer	-	80

Tab. 16-3 Typy danych używane w X87 FPU



Rys. 16-8 Typy danych używane w X87 FPU

16.3 Instrukcje jednostki FPU

Instrukcje jednostki X87 FPU opisane są w dokumentacji [10], [11] i w [25]. Instrukcje procesora FPU podzielone sa na sześć grup:

- Przesyłania danych
- Instrukcje arytmetyczne
- Porównania
- Trygonometryczne
- Ładowania stałych
- Sterujące jednostką FPU

16.3.1 Instrukcje przesyłania danych

Instrukcje jednostki FPU dotyczące przesyłania danych są nastepujące:

- Przesłanie liczby zmiennorzecinkowej, całkowitej albo spakowanej BCD z pamieci do rejestru ST(0).
- Zapamietanie w pamięci wierzchołka stosu ST(0) w formacie liczby zmiennorzecinkowej, całkowitej albo spakowanej BCD.
- Przesłanie zawartości pomiędzy komórkami stosu.

Instrukcja	Znaczenie
Operandy całkowite	
`fild`	Załaduj na stos liczbę integer
`fist`	Prześlij liczbę integer ze stosu do pamięci (nie usuwając jej ze stosu)
`fistp`	Prześlij liczbę integer ze stosu do pamięci i usuń ją ze stosu (operacja pop)
Operandy zmiennoprzecinowe	
`fld`	Załaduj na stos liczbę float
`fst`	Prześlij liczbę float ze stosu do pamięci (nie usuwając jej ze stosu)
`fstp`	Prześlij liczbę float ze stosu do pamięci i usuń ją ze stosu (operacja pop)
`fxch`	Zamień rejestry

Tab. 16-4 Ważniejsze instrukcje przesłań pamięć - stos jednostki FPU

Podane wyżej instrukcje mają wersje, różniące się sufiksem, w zależności od typu operandu przesyłanego z pamięci czy do pamięci. Gdy przesyłamy liczbę pojedynczej precyzji .float lub .single rozkaz ma sufiks (końcówkę) s (single), gdy podwójnej precyzji .double rozkaz ma sufiks l (long), gdy rozszerzonej .tfloat rozkaz ma sufiks t (ten bytes). W przypadku przesyłania liczb całkowitych sufiksy dla operandów .word', '.long' lub '.int', i '.quad' są 's' (single), 'l' (long), and 'q' (quad).

Instruction		Source format	Source location
`flds`	*Addr*	Single	$M_4[Addr]$
`fldl`	*Addr*	Double	$M_8[Addr]$
`fldt`	*Addr*	Extended	$M_{10}[Addr]$
`fildl`	*Addr*	Integer	$M_4[Addr]$
`fld`	`%st(`*i*`)`	Extended	`%st(`*i*`)`

Tab. 16-5 Ważniejsze instrukcje przesłań pamięć - stos jednostki FPU w zależności od typu operandu

Instrukcja `fldl qword` (load) zmniejsza wskaźnik TOP o 1 i przesyła zawartość 8 bajtów z adresu `qword` na nowy szczyt stosu do rejestru ST(0). Jeśli operand jest pojedyczej lub podwójnej precyzji, jest on zamieniany do formatu double extended. Instrukcja może być też użyta do przesłania innego rejestru ST(i) na wierzcołek stosu ST(0). Przykładowo:

```
fldl avg
fld (%esp)
```

Instrukcja FILD robi to samo ale przeprowadza konwersję z integer do formatu double extended

```
fild count
fild (%esp)
```

Przesyłanie typu wierzchołek stosu – pamięć ma różną postać w zależności od typu operandu w pamięci i od tego czy po wykonaniu przesłania wielkość zdejmowana jest ze stosu (operacja `pop`). Pokazuje to poniższa tabela.

Instruction		Pop (Y/N)	Destination format	Destination location
`fsts`	*Addr*	N	Single	$M_4[Addr]$
`fstps`	*Addr*	Y	Single	$M_4[Addr]$
`fstl`	*Addr*	N	Double	$M_8[Addr]$
`fstpl`	*Addr*	Y	Double	$M_8[Addr]$
`fstt`	*Addr*	N	Extended	$M_{10}[Addr]$
`fstpt`	*Addr*	Y	Extended	$M_{10}[Addr]$
`fistl`	*Addr*	N	Integer	$M_4[Addr]$
`fistpl`	*Addr*	Y	Integer	$M_4[Addr]$
`fst`	`%st(`*i*`)`	N	Extended	`%st(`*i*`)`
`fstp`	`%st(`*i*`)`	Y	Extended	`%st(`*i*`)`

Tab. 16-6 Ważniejsze instrukcje przesłań pamięć - stos jednostki FPU w zależności od typu operandu docelowego i wykonania operacji pop

Instrukcja `fstl qword` (store) przesyła zawartość szczytu stosu czyli rejestru ST(0) w formacie double do komórki o adresie `qword`.

```
fstl qword
```

Instrukcja `fstpl qword` (store and pop) przesyła zawartość szczytu stosu czyli rejestru ST(0) do komórki o adresie `qword` dokonując konwersji na typ double. Następnie wykonywana jest operacja `pop` czyli wskaźnik wierzchołka stosu TOP zwiększany jest o 1.

W poprzednich instrukcjach operand może być zawartością pamięci lub innym rejestrem ze stosu FPU. Nie jest możliwe przesłanie pomiędzy innym rejestrem x86 a rejestrem FPU.

Instrukcja `fxch st(i)` (exchange) wymienia zawartość rejestru ST(0) z rejestrem ST(i).

16.3.2 Ładowanie stałych

Procesor X87 FPU posiada instrukcje ładowania stałych. Pokazano je poniżej.

Instrukcja	Opis
`fldz`	`push 0`
`fld1`	`push 1`
`fldpi`	`push pi`
`fldl2e`	`push log2(e)`
`fldl2t`	`push log2(10)`
`fldlg2`	`push log10(2)`
`fldln2`	`push ln(2)`

Tab. 16-7 Instrukcje ładowania stałych w jednostce X87 FPU

16.3.3 Instrukcje arytmetyczne

Aby dodać, odjąć, pomnożyć lub podzielić dwie liczby w dyspozycji są dwie możliwości:

- Użyć dwóch rejestrów FPU
- Użyć jednego z rejestrów FPU (zwykle ST(0)) i operandu w pamięci.

Operandem może być liczba zmiennoprzecinkowe pojedynczej precyzji, podwójnej precyzji, rozszerzonej precyzji, słowo integer, podwójne słowo integer. Podczas wykonywania instrukcji są one tłumaczone na format rozszerzony podwójnej precyzji. Wynik operacji przechowywany jest zwykle w ST(0), ale możliwe są również wersje odwrotne (ang. *reverse*), kodowane z literą r na końcu.

Instrukcja	Opis
`fadd`	dodaj st(0) do st(1) i wykonaj pop (wynik w st(0))
`fadd st(n), st`	dodaj st(1) - st(7) do st(0)
`fadd st, st(n)`	dodaj st(0) do st(1-7)
`faddp st, st(n)`	dodaj st do st(n) i pop
`fadd x`	dodaj real x do st
`fiadd x`	dodaj integer word lub dword x do st, operandy są jak `fadd, faddp, fiadd`
`fsub, fisub`	odejmij float, integer
`fsubr, fisubr`	odejmij odwrotnie: st(1) = st-st(1), pop
`fmul, fimul`	pomnóż
`fdiv, fidiv`	podziel
`fdivr, fidivr`	podziel odwrotnie
`fsubp, fsubrp, fmulp, fdivp, fdivrp`	Po działaniu wykonaj pop podobnie jak w `faddp`

Tab. 16-8 Podstawowe operacje arytmetyczne jednostki FPU (bez suffiksów)

Gdy jeden z operandów instrukcji arytmetycznych znajduje się w pamięci, instrukcja posiada dodatkowy suffiks `s,l,t` zależny od typu danych operandu w pamięci. Przykłady dla instrukcji odejmowania pokazuje poniższa tabela.

Instruction		Operand 1	Operand 2	Format	Destination	Pop `%st(0)` (Y/N)
`fsubs`	*Addr*	`%st(0)`	$M_4[Addr]$	Single	`%st(0)`	N
`fsubl`	*Addr*	`%st(0)`	$M_8[Addr]$	Double	`%st(0)`	N
`fsubt`	*Addr*	`%st(0)`	$M_{10}[Addr]$	Extended	`%st(0)`	N
`fisubl`	*Addr*	`%st(0)`	$M_4[Addr]$	Integer	`%st(0)`	N
`fsub`	`%st(`*i*`),%st`	`%st(`*i*`)`	`%st(0)`	Extended	`%st(0)`	N
`fsub`	`%st,%st(`*i*`)`	`%st(0)`	`%st(`*i*`)`	Extended	`%st(`*i*`)`	N
`fsubp`	`%st,%st(`*i*`)`	`%st(0)`	`%st(`*i*`)`	Extended	`%st(`*i*`)`	Y
`fsubp`		`%st(0)`	`%st(1)`	Extended	`%st(1)`	Y

Tab. 16-9 Postacie instrukcji odejmowania w zależności od typu operandu, położenia wyniku i wykonania operacji pop.

Przykładowo instrukcja `faddp` wykonuje następujące działania:

- Dodaje st(0) do st(1) wynik pamiętany w st(1)
- Wykonuje operację pop, wynik będzie w st(0)

Instrukcja `fsubp` wykonuje następujące działania:

- Wykonuje odejmowanie st(1) – st(0), wynik w st(1)
- Wykonuje operację pop, wynik będzie w st(0)

Instrukcja `fsubpr` wykonuje następujące działania:

- Wykonuje odejmowanie st(0) – st(1), wynik w st(1)
- Wykonuje operację pop, wynik będzie w st(0)

Instrukcja `fdivp` wykonuje następujące działania:

- Wykonuje dzielenie st(1) = st(1)/st(0), wynik w st(1)
- Wykonuje operację pop, wynik będzie w st(0)

Instrukcja `fdivrp` wykonuje następujące działania:

- Wykonuje dzielenie st(1) = st(0)/st(1), wynik w st(1)
- Wykonuje operację pop, wynik będzie w st(0)

Jednostka X87 FPU wykonuje także operacje jednoargumentowe. Operandem jest domyslnie ST(0). Przedstawia je poniższa tabela.

Instrukcja	Opis
`fldz`	Ładuj zero
`fld1`	Ładuj 1
`fabs`	st = abs(st)
`fchs`	st = -st
`frndint`	Zaokrąglenie do liczby całkowitej (zależnie od trybu zaokraglania)
`fsqrt`	Pierwiastek kwadratowy
`fcos`	Kosinus (argument w radianach)
`fsin`	Sinus
`fsincos`	Sinus a następnie, push kosinus
`fptan`	Tangens
`fpatan`	st(1) = arctan(st(1)/st(0)), pop

Tab. 16-10 Jednoargumentowe operacje arytmetyczne jednostki FPU

16.3.4 Instrukcje porównania

Instrukcje porównania dwóch liczb zmiennoprzecinkowych różnią się od porównania liczb całkowitych gdyż trzeba rozpatrzyć cztery a nie trzy przypadki. Wyniki porównania mogą być nastepujące:

- Mniejszy niż
- Równy
- Większy niż
- Nieuporządkowany

Ostatni przypadek ma miejsce gdy co najmniej jeden z operandów przyjmuje wartość NaN (nie jest liczbą).

Instrukcja	Znaczenie
Operand całkowity	
FICOM	Porównaj z liczbą całkowitą
FICOMP	Porównaj z liczbą całkowitą i wykonaj pop
Operand zmiennoprzecinkowy	
FCOM	Porównaj liczby floating-point
FCOMP	Porównaj liczby floating-point i wykonaj pop
FCOMPP	Porównaj liczby floating-point i wykonaj pop dwókrotnie
FCOMI	Porównaj liczby floating-point i ustaw EFLAGS
FCOMIP	Porównaj liczby floating-point, ustaw EFLAGS, i wykonaj pop
FUCOM	Porównaj nie gen. wyjatku floating-point
FUCOMP	Porównaj nie gen. wyjatku floating-point and pop
FUCOMPP	Porównaj nie gen. wyjatku floating-point and pop twice
FUCOMI	Porównaj nie gen. wyjatku floating-point and set EFLAGS
FUCOMIP	Porównaj nie gen. wyjatku floating-point, set EFLAGS, and pop
FTST	Testuj floating-point (compare with 0.0)
FXAM	Examine floating-point

Tab. 16-11 Instrukcje porównania w jednostce X87 FPU

Instrukcje FCOM / FCOMP / FCOMPP dokonują porównania zawartości rejestru ST(0) z operandem zmiennoprzecinkowym w pamięci i ustawiają flagi C0, C2 i C3 rejestru statusowego zgodnie z poniższą tabelą.

Condition	C3	C2	C0
ST(0) > Source Operand	0	0	0
ST(0) < Source Operand	0	0	1
ST(0) = Source Operand	1	0	0
Unordered	1	1	1

Tab. 16-12 Ustawienie flag w wyniku porównania za pomoca instrukcji FCOM / FCOMP / FCOMPP

Gdy wynik operacji jest nieuporządkowany, generowany jest wyjątek nieprawidłowa operacja. Wersje FCOMP i FCOMPP dokonują po porównaiu pojedynczej lub podwójnej operacji pop.
Instrukcje FUCOM / FUCOMP / FUCOMPP działają tak jak FCOM / FCOMP / FCOMPP z tą różnicą że w przypadku otrzymania wyniku „nieuporządkowana" nie jest generowany wyjatek, a wynik jest postaci QNaN. Rozkazy FICOM i FICOMP działają podobnie jak FCOM / FCOMP tyle że operand w pamięci może być liczbą integer.

Instrukcje FCOMI i FCOMIP zostały wprowadzone w wersji P6. Działają one tak jak FCOM / FCOMP z ta różnicą że ustawiają flagi ZF, CF i PR rejestru flagowego EFL i mogą być bezpośrednio stosowane instrukcje skoków warunkowych.

Comparison Results	ZF	PF	CF
ST0 > ST(*i*)	0	0	0
ST0 < ST(*i*)	0	0	1
ST0 = ST(*i*)	1	0	0
Unordered	1	1	1

Tab. 16-13 Ustawienie flag w rejestrze EFLw wyniku porównania za pomoca instrukcji FCOMI / FCOMIP

16.3.5 Instrukcje sterujące

Instrukcje sterujące sterujące jednostki FPU pokazuje poniższa tabela.

Instrukcja	Opis
FINIT	Inicializacja FPU , ustawienie rejestrów na domyślne wartości
FNINIT	Inicializacja FPU , ustawienie rejestrów na domyślne wartości (bez sprawdzania błędów)
FFREE	Zwolnij rejestry
FINCSTP	Zwiększ wskaźnik stosu FPU
FDECSTP	Zmniejsz wskaźnik stosu FPU
FSTSW	Zapisz do pamięci rejestr statusowy FPU
FNSTSW	Zapisz do pamięci rejestr statusowy FPU bez sprawdzania warunku błędu
FCLEX	Skasuj flagę błędu (po sprawdzeniu warunków błędu)
FNCLEX	Skasuj flagę błędu (bez sprawdzeniu warunków błędu)
FSTCW	Zapisz do pamięci rejestr sterujący FPU
FNSTCW	Zapisz do pamięci rejestr sterujący FPU bez sprawdzania warunków błędu
FLDCW	Załaduj słowo sterujące FPU z pamięci (ang. *Load FPU control word*)
FSAVE	Zapamiętaj stan FPU po sprawdzeniu warunków błędu, inicjalizuj stan FPU (we wskazanej lokacji pamięci)
FNSAVE	Zapamiętaj stan FPU bez sprawdzeniu warunków błędu (we wskazanej lokacji pamięci)
FRSTOR	Odtwórz stan FPU
FSTENV	Zachowaj otoczenie FPU (po sprawdzeniu warunków błędu). Otoczenie to: control word, status word, tag word, instruction pointer, data pointer, and last opcode.
FNSTENV	Zachowaj otoczenie FPU (bez sprawdzeniu warunków błędu)
FLDENV	Załaduj z pamięci stan otoczenia FPU
FNOP	Instrukcja pusta (FPU no operation)
FWAIT	Czekaj na gotowość FPU (aż zakończy się obsługa wyjatku)

Tab. 16-14 Instrukcje sterujące jednostki FPU

Instrukcje sterujące służą do:

- Inicjalizacji rejestrów danych sterującego, statusu i innych na domyślne wartości.
- Zachowania i przywrócenia stanu FPU (konieczne przy przełączaniu kontekstu).
- Sterowania obsługą wyjątków.
- Innych celów.

16.4 Obsługa wyjątków w jednostce X87 FPU

Procesor zmiennoprzecinkowy X87 FPU generuje 8 wyjątków pokazanych w poniższej tabeli. Każdy z wyjatków ma swoją flagę w rejestrze statusu i maskę w rejestrze sterującym co pokazuje poniższa tabela.

Opis	Description	Kod wyjatku	Maska	Flaga wyjątku
Nieprawidłowa operacja arytmetyczna	Invalid arythmetic operation	IA#		ES
Przepełnienie stosu	Stack overflow / underflow	IS#		SF
Nieprawidłowa operacja	Invalid operation	I#	IM	IE
Dzielenie przez zero	Divide-by-zero	Z#	ZM	ZE
Nieznormalizowany operand	Denormalized operand	D#	DM	DE
Nadmiar numeryczny	Numeric overflow	O#	OM	OE
Niedomiar numeryczny	Numeric underflow	U#	UM	UE
Niedokładny wynik	Inexact result	P#	PM	PE

Tab. 16-15 Flagi wyjątków jednostki FPU w architekturze IA-32

Flaga ES jest ustawiana gdy wystapi jakikolwiek wyjątek. Maska wyjątku może być ustawiana instrukcjami FLDCW, FRSTOR, FXRSTOR a odczytana instrukcjami FSTCW, FNSTCW i innymi.

Gdy wystąpi wyjatek błąd reakcja zależy od ustawienia maski błędu.

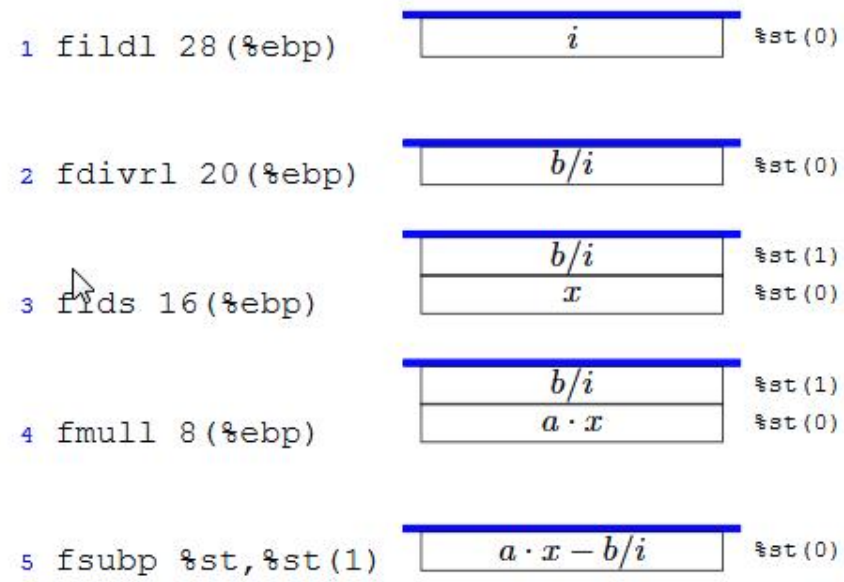
- Gdy maska nie jest ustawiona, wywoływany jest handler wyjątku.
- Gdy wyjatek jest zamaskowany, w wyniku działania operacji zwracana jest wartość niezdefiniowana QNaN.

16.5 Wykorzystanie liczb zmiennoprzecinkowych w funkcjach

Argumenty zmiennoprzecinkowe są do funkcji przekazywane przez stos podobnie do argumentów całkowitych. Argument typu `float` zajmuje 4 bajty, typu `double` wymaga 8 bajtów. Dla funkcji zwracających wynik typu `float` lub `double` wykorzystywany jest szczyt stosu zmiennoprzecinkowego w formacie `extended double`. Jako przykład rozważona zostanie poniższa funkcja .

```
double funct(double a, float x, double b, int i)
{
      return(a*x – b/i);
}
```

Argumenty przekazane są przez stos. Argumenty a, x, b, i będą miały przesunięcia 8, 16, 20 i 28 względem rejestru bazowego `%ebp`. Przykładowy kod funkcji podano poniżej.



Przykład 16-2 Kod funkcji func i przebieg obliczeń

Przekazywanie parametrów double do funkcji `printf` z biblioteki standardowej pokazuje poniższy przykład. Należy wiedzieć że do funkcji `printf` należy przekazać argumenty float skonwertowane do postaci `double`.

```
# Przyklad uzycia funkcji printf z programu w assemblerze x86/x87
# parametr float zamieniany na double
# kompilacja:
# gcc printftest.s -o printftest –m32 -g
.section .data
formatstr: .ascii "float: %f \n\0"
liczba1:   .float  6.148
.text
.global main

main:
# Ladujemy liczba1 do st(0)
fld liczba1
# rezerwacja 8 bajtow na argument double
push %eax
push %eax
# przesylamy na stos zwykly liczba1 z st(0)
fstpl (%esp)
# przesylamy na stos adres formatstr
pushl $formatstr
# wywolujemy printf("float: %f\n",liczba1);
call printf
# cofamy stos
popl %ebx
popl %ebx
popl %ebx
ret
```

Przykład 16-3 Wywołanie funkcji `printf(float: %f \n\0",liczba1)`

16.6 Przykład – obliczanie wyrażenia

Poniżej podano przykład obliczenia wyrazenia `z = z + x*y` z uzyciem FPU. Program nazywa się `add-test.s`.

```
# Obliczenie wyrazenia z = z + x*y z uzyciem FPU
#
# kompilacja: gcc -m32 -nostdlib -g add-test.s -o add-test
# obserwacja pod gdb:
# gdb -tui add-test
SYSEXIT=1
EXIT_SUCCESS=0
.att_syntax noprefix
.data
x: .double 2
y: .double 3
z: .double 4
.text
.globl _start
_start:
fldl    x
fldl    y
fmulp   st, st(1)
fldl    z
faddp   st, st (1)
fstpl   z
mov $SYSEXIT, %eax
mov $EXIT_SUCCESS, %ebx
int $0x80
```

Przykład 16-4 Obliczanie wyrażenia z = z + x*y za pomocą FPU

Program kompilujemy jak poniżej:

```
gcc -m32 -nostdlib -g add-test.s -o add-test
```

Następnie obserwujemy jego działanie pod kontrolą debuggera pisząc polecenie:

```
gdb –tui add-test
```

Ustawiamy punkt zatrzymania na instrukcji `fmulp st,st(1)` i możemy zaobserwować stan rejestrów pisząc polecenie:

```
info float
```

co pokazuje poniższy ekran.

```
(gdb) info float
  R7: Valid   0x40008000000000000000 +2
=>R6: Valid   0x4000c000000000000000 +3
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
  R0: Empty   0x00000000000000000000
Status Word:         0x3000
                       TOP: 6
Control Word:        0x037f   IM DM ZM OM UM PM
                       PC: Extended Precision (64-bits)
                       RC: Round to nearest
Tag Word:            0x0fff
Instruction Pointer: 0x73:0x0040019e
Operand Pointer:     0x7b:0x00402014
Opcode:              0xdd05
```

Przykład 16-5 Obliczanie wyrażenia z = z + x*y za pomocą FPU, testowanie z gdb, po instrukcji `fldl y`

```
 (gdb) info float
=>R7: Valid   0x4001c000000000000000 +6
  R6: Empty   0x4000c000000000000000
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
  R0: Empty   0x00000000000000000000
```

Przykład 16-6 Obliczanie wyrażenia z = z + x*y za pomocą FPU, testowanie z gdb, po instrukcji fmulp st, st(1)

16.7 Przykład – suma liczb z uzyciem FPU

Poniżej podano przykład programu w C który wywołuje funkcję suma napisaną w assemblerze.

```
// Kalkulator - program korzysta z kalk1.s
// kompilacja:  gcc kalk1.c kalk1.s -m32 -g -o kalk1
//
#include <stdio.h>
#include <stdlib.h>

// Funkcja zwraca sume a+b
double suma(double a, double b);

double a = 1.0;
double b = 2.1;
double wynik;
// unsigned long long rdtsc() ;

int main() {
  printf("Start\n") ;
  wynik = suma(a,b);
  printf(" %f + %f = %f\n",a,b,wynik);
}
```

Przykład 16-7 Program `kalk1.c` w C wywołujący funkcję suma z pliku kalk1.s

```
# Kalkulator - funkcje dla programu kalk1.c
# kompilacja:  gcc kalk1.c kalk1.s -m32 -g -o kalk1
#
.section .text
.globl suma
suma:
# prolog - nowa ramka stosu
push %ebp
movl %esp, %ebp
# laduj a do st(0)
fldl 8(%ebp)
#laduj b do st(0), w st(1) jest b
fldl 16(%ebp)
# dodaj st(0) + st(1) -> st(0)
faddp
# wynik w st(0)
# przywracamy stos
mov %ebp, %esp
pop %ebp
# powrot
ret
```

Przykład 16-8 Program `kalk1.s` zawierający funkcję suma

Programy się kompiluje za pomocą polecenia:

```
gcc kalk1.c kalk1.s -m32 -g -o kalk1
```

16.8 Zadania

16.8.1 Dodawanie liczb, wynik wyprowadzany funkcją printf

Zmodyfikuj przykład dodawanie liczb Przykład 16-4 tak aby wynik wyprowadzony był za pomocą funkcji `printf` jak w Przykład 16-3 .

16.8.2 Obliczanie formuły arytmetycznej

Wzorujac się na poprzednim programie napisz program obliczania formuły: `x = (a-b)* (-b + c)` gdzie: `a: .double 3.0, b: .double 1.0, c: .double 4.0, d: .double 2.0,  x: .float 0.0`. Obserwuj działanie pod gdb. Wynik wyprowadź za pomocą funkcji printf.

16.8.3 Kalkulator zmiennoprzecinkowy

Korzystając z Przykład 16-8 napisz program kalkulator który realizuje funkcje kalkulatora zmiennoprzecinkowego. Mają być wykonywane operacje:

+	dodawanie liczb	wynik = a+b
-	Odejmowanie liczb	Wynik = a-b
*	Mnożenie liczb	wynik = a*b
/	dzielenie liczb	Wynik = a/b
P	Pierwiastek kwadratowy	Wynik = sqrt(a)
s	Sinus	Wynik = sin(a)

17. Znaki i kody ASCII

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040	&#032;	Space	64	40	100	&#064;	@	96	60	140	&#096;	`
1	1	001	Start of Header	33	21	041	&#033;	!	65	41	101	&#065;	A	97	61	141	&#097;	a
2	2	002	Start of Text	34	22	042	&#034;	"	66	42	102	&#066;	B	98	62	142	&#098;	b
3	3	003	End of Text	35	23	043	&#035;	#	67	43	103	&#067;	C	99	63	143	&#099;	c
4	4	004	End of Transmission	36	24	044	&#036;	$	68	44	104	&#068;	D	100	64	144	&#100;	d
5	5	005	Enquiry	37	25	045	&#037;	%	69	45	105	&#069;	E	101	65	145	&#101;	e
6	6	006	Acknowledgment	38	26	046	&#038;	&	70	46	106	&#070;	F	102	66	146	&#102;	f
7	7	007	Bell	39	27	047	&#039;	'	71	47	107	&#071;	G	103	67	147	&#103;	g
8	8	010	Backspace	40	28	050	&#040;	(	72	48	110	&#072;	H	104	68	150	&#104;	h
9	9	011	Horizontal Tab	41	29	051	&#041;	)	73	49	111	&#073;	I	105	69	151	&#105;	i
10	A	012	Line feed	42	2A	052	&#042;	*	74	4A	112	&#074;	J	106	6A	152	&#106;	j
11	B	013	Vertical Tab	43	2B	053	&#043;	+	75	4B	113	&#075;	K	107	6B	153	&#107;	k
12	C	014	Form feed	44	2C	054	&#044;	,	76	4C	114	&#076;	L	108	6C	154	&#108;	l
13	D	015	Carriage return	45	2D	055	&#045;	-	77	4D	115	&#077;	M	109	6D	155	&#109;	m
14	E	016	Shift Out	46	2E	056	&#046;	.	78	4E	116	&#078;	N	110	6E	156	&#110;	n
15	F	017	Shift In	47	2F	057	&#047;	/	79	4F	117	&#079;	O	111	6F	157	&#111;	o
16	10	020	Data Link Escape	48	30	060	&#048;	0	80	50	120	&#080;	P	112	70	160	&#112;	p
17	11	021	Device Control 1	49	31	061	&#049;	1	81	51	121	&#081;	Q	113	71	161	&#113;	q
18	12	022	Device Control 2	50	32	062	&#050;	2	82	52	122	&#082;	R	114	72	162	&#114;	r
19	13	023	Device Control 3	51	33	063	&#051;	3	83	53	123	&#083;	S	115	73	163	&#115;	s
20	14	024	Device Control 4	52	34	064	&#052;	4	84	54	124	&#084;	T	116	74	164	&#116;	t
21	15	025	Negative Ack.	53	35	065	&#053;	5	85	55	125	&#085;	U	117	75	165	&#117;	u
22	16	026	Synchronous idle	54	36	066	&#054;	6	86	56	126	&#086;	V	118	76	166	&#118;	v
23	17	027	End of Trans. Block	55	37	067	&#055;	7	87	57	127	&#087;	W	119	77	167	&#119;	w
24	18	030	Cancel	56	38	070	&#056;	8	88	58	130	&#088;	X	120	78	170	&#120;	x
25	19	031	End of Medium	57	39	071	&#057;	9	89	59	131	&#089;	Y	121	79	171	&#121;	y
26	1A	032	Substitute	58	3A	072	&#058;	:	90	5A	132	&#090;	Z	122	7A	172	&#122;	z
27	1B	033	Escape	59	3B	073	&#059;	;	91	5B	133	&#091;	[	123	7B	173	&#123;	{
28	1C	034	File Separator	60	3C	074	&#060;	<	92	5C	134	&#092;	\	124	7C	174	&#124;	|
29	1D	035	Group Separator	61	3D	075	&#061;	=	93	5D	135	&#093;	]	125	7D	175	&#125;	}
30	1E	036	Record Separator	62	3E	076	&#062;	>	94	5E	136	&#094;	^	126	7E	176	&#126;	~
31	1F	037	Unit Separator	63	3F	077	&#063;	?	95	5F	137	&#095;	_	127	7F	177	&#127;	Del

18. Dodatek 2 - Operacje na bitach w języku C

W języku C zdefiniowane są operacje na bitach. Operacje dwuargumentowe:

- koniunkcja bitowa ("&"),
- alternatywa bitowa ("|") i
- alternatywa rozłączna (XOR)

Operacje jednoargumentowe:

- negacja bitowa ("~"),
- przesunięcie bitowe

bit a	bit b	a&b
0	0	0
0	1	0
1	0	0
1	1	1

bitowe AND &

bit a	bit b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

bitowe OR |

bit a	bit b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

bitowe XOR ^

Przykład bitowego AND

```
 11001110
&10011000
= 10001000
```

Przykład bitowego OR

```
 11001110
|10011000
=10001000
```

Przykład bitowego XOR

```
 11001110
^10011000
=01010110
```

bit a	~a
0	1
1	0

Negacja bitowa

```
~11001110
 00110001
```

Przesunięcie w prawo >>

Przesuniecie w prawo [zmienna] >> [liczba miejsc] jest operacją dwuargumentową A >> n. Przesuwa ona bity operandu A o n pozycji w prawo. Na miejsce najstarszych bitów wchodzi 0
Gdy x=11100101 to x >> 1 daje w wyniku 01110010, x >> 2 daje w wyniku 00111001

Przesunięcie w lewo <<

Przesuniecie w lewo [zmienna] << [liczba miejsc] jest operacją dwuargumentową A << n. Przesuwa ona bity operandu A o n pozycji w lewo. Na miejsce najmłodszych bitów wchodzi 0
Gdy x=11100101 to x << 1 daje w wyniku 11001010, x << 2 daje w wyniku 10010100

```
#include <stdio.h>
 void pokaz_bity(unsigned int x){
        int i;
        for(i=(sizeof(int)*8)-1; i>=0; i--)
            (x&(1<<i))?putchar('1'):putchar('0');
      printf("\n");
 }

 int main(int argc,char *argv[]) {
        int j, m, n;
        j = atoi(argv[1]);
        printf("Dziesietne: %d hex: %08X binarnie - ",j, j);
        pokaz_bity(j);
        return 0;
 }
```

Przykład 18-1 Wypisywanie liczby w postaci HEX i binarnej

19. Literatura

[1] Strona internetowa ZAK, http://www.zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/

[2] Janusz Biernat wykłady z Architektury komputerów, http://www.zak.ict.pwr.wroc.pl/materials/architektura/wyklad%20AK2/

[3] Jonathan Bartlet, Programming from Ground Up, http://cvs.savannah.nongnu.org/viewvc/pgubook/pgubook/ProgrammingGroundUp/

[4] Bryant O'Hallaron, Computer Systems, a programmer's perspective. http://www.codeman.net/wp-content/uploads/2011/12/Computer_Systems-A_Programmers_Perspective-2e.pdf.

[5] Dean Elsner, Jay Fenlason & friends, Using as The GNU Assembler, ftp://ftp.gnu.org/old-gnu/Manuals/gas/html_chapter/as_toc.html

[6] BIERNAT J., Architektura komputerów , Wrocław, OW PWr, 2005 (wyd. IV)

[7] Wprowadzenie do Laboratorium http://www.zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Wprowadzenie%20do%20laboratorium.pdf

[8] Dean Elsner, Jay Fenlason & friends, Using As, https://sourceware.org/binutils/docs/as/

[9] Laboratorium AK –ATT asembler (LINUX) , http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Linux-asm-lab-2015.pdf

[10] IA-32 Intel® Architecture Software Developers Manual Volume 1: Basic Architecture , https://www.intel.pl/content/www/pl/pl/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html

[11] IA-32 Intel® Architecture Software Developers Manual Volume 2: Instruction Set Reference, http://www.zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/Intel%20Pentium%20IV/IA-32%20Intel%20Architecture%20Software%20Developers%20Manual%20vol.%202%20-%20Instruction%20Set%20Reference.pdf

[12] IA-32 Intel® Architecture Software Developers Manual Volume 3: System programmers guide.

[13] The GNU project Debugger https://www.gnu.org/software/gdb/documentation/

[14] System V Application Binary Interface (x86_64) - rozdział 3, http://www.sco.com/developers/devspecs/abi386-4.pdf

[15] Brian Ward, Jak działa Linux, Helion 2015.

[16] The GNU C Library https://www.gnu.org/software/libc/manual/html_mono/libc.html

[17] Denis de Leeuw Duarte, TI1400 Lab Course Manual, http://www.ds.ewi.tudelft.nl/~mihai/ti1400-files/ti1400-manual.pdf

[18] Himanshu Arora, GPROF Tutorial – How to use Linux GNU GCC Profiling Tool, https://www.thegeekstuff.com/2012/08/gprof-tutorial/

[19] http://tommesani.com/index.php/component/content/article/2-simd/39-mmx-conversion.html

[20] MATLAB toolbox https://documents.epfl.ch/users/l/le/leuteneg/www/MATLABToolbox/FPUasmHints.html

[21] Alex Peleg, Uri Weiser, MMX Technology Extension to the Intel Architecture, Israel Design Center, IEEE Micro 1996, http://www.engr.uconn.edu/~zshi/course/cse5302/ref/peleg96mmx.pdf

[22] Opis biblioteki SDL, https://www.libsdl.org/release/SDL-1.2.15/docs/html/guidevideo.html#AEN83

[23] Kurs CSE3101 Assembly Language Programming Notes, https://cs.fit.edu/~mmahoney/cse3101/notes.html

[24] Instrukcje IA-32 https://c9x.me/x86/

[25] David R. Hallaron, Randal E. Bryant, CS:APP2e Aside ASM:87, X87-Based Support for Floating Point http://csapp.cs.cmu.edu/2e/waside/waside-x87.pdf

[26] Lista instrukcji x86/x87 https://www.felixcloutier.com/x86/index.html

[27] David A Rusling, A Linux kernel, https://www.tldp.org/LDP/tlk/mm/memory.htm