

PROGRAM LABORATORIUM Architektury Komputerów (0-4)

Sprawozdanie:

1. W sprawozdaniu powinny znaleźć się odpowiedzi na trzy pytania:
 - zakres planowanych (wg programu) i wykonanych prac,
 - przebieg ćwiczenia – sposób wykorzystania narzędzi i opis opracowanych algorytmów
 - wnioski (rzeczowe!).
2. Jeśli sprawozdanie zawiera fragmenty innych prac (w tym także innych sprawozdań) bez wyraźnej adnotacji, traktowane jest to jako PLAGIAT z wszelkimi tego konsekwencjami.
3. Nieodzownym elementem każdego sprawozdania jest spis literatury i odwołania do literatury w tekście. Jeśli w sprawozdaniu znajduje się opis działań wykonanych poza laboratorium, należy to wyraźnie zaznaczyć i rozdzielić wyniki uzyskane w trakcie laboratorium od wyników późniejszych.
4. Szczególną uwagę należy zwrócić także na sposób prezentacji danych uzyskanych np.: wskutek pomiarów. Złym pomysłem jest zamieszczanie zrzutu ekranu zawierającego niezmodyfikowany zapis działania programu dla wszystkich możliwych przypadków. Znacznie lepszym rozwiązaniem jest zamieszczenie tabeli/wykresu lub innych form czytelnie prezentujących zaobserwowane zjawiska.
5. Kody źródłowe programów należy zamieścić w całości (wraz z komentarzami).

Literatura podstawowa (obowiązkowa)

- [1] Dokumentacja poleceń dostępna w manualach Linux-a (*man command*) a także w katalogach:
<http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium>
<http://sourceware.org/binutils/> (GNU binutils main page)
<http://sourceware.org/binutils/docs-2.22/> (ver. 2011-11-11)/gas – assembler, gprof – profiler, ld ...
- [2] „Using as”, dokumentacja kompilatora, <http://sourceware.org/binutils/docs-2.22/as/index.html>
rozdział 3 – składnia pliku źródłowego programu assemblerowego, szczególną uwagę proszę zwrócić na wpływ białych znaków, sposób tworzenia komentarzy oraz stałych
rozdział 4 – segmenty programu, typy segmentów zawierających dane
rozdział 7 – dyrektywy assemblera, w szczególności organizacja pamięci (deklaracja zmiennych rezerwacja bloków pamięci) oraz dyrektywy umożliwiające stosowanie makr.
rozdział 8.11 –informacje niezbędne do sprawnego tworzenia kodu („jak adresować kolejne elementy tablicy”, „jak przesłać pomiędzy pamięcią a rejestrem 1/2/4 bajty” itp.)
- [3] Dokumentacja dotycząca tworzenia programów:
<http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/...>
Najważniejsze informacje, powiązane z trybem pracy procesora to:
 - dostępne rejestry procesora i tryby adresowania,
 - dostępne instrukcje i operacje możliwe do wykonania za pomocą pojedynczej instrukcji
 - sposób wykonania instrukcji
 - operacje, których NIE MOŻNA wykonać za pomocą pojedynczej instrukcji.

Środowisko programistyczne: Linux (gcc /as /ld/...),

Podstawowe umiejętności:

- edycja tekstu -vim, praca z dokumentacją.
- użycie podstawowych poleceń systemu operacyjnego Linux (ls, cd, mkdir, rm,...)
- korzystanie z podręcznika systemowego **man** ze szczególnym uwzględnieniem metod zdobycia informacji o funkcjach systemowych i bibliotecznych

0. Środowisko programistyczne laboratorium Architektury komputerów

Cel: nabycie umiejętności takich jak:

- kompilacja źródeł assemblerowych za pomocą kompilatora **as** i konsolidatora (linkera) **ld** oraz kompilatora **gcc**, znajomość podstawowych opcji kompilatora i linkera (o, g, e, l, v, c)
- tworzenie prostych plików sterujących przetwarzaniem za pomocą programu **make**
- podstawy tworzenia kodu assemblerowego – struktura programu, deklarowanie zmiennych, atrybuty zmiennych (adres, wartość i rozmiar zmiennej jednostkowej), tryby adresowaniaużycie podstawowych funkcji systemowych (przekazywanie argumentów i zwracanie wyniku; przeczytać plik `unistd.h` – identyfikatory funkcji i podręcznik systemowy **man**).

Przykładowe zadania:

- przepisać i skompilować kod zamieszczony we wzorcowym sprawozdaniu
- zmodyfikować kod do postaci pozwalającej wykorzystać funkcję systemową `read`
- przekształcić tekst za pomocą masek logicznych lub szyfrem Cezara

1. Proste konstrukcje programowe (pętle, badanie warunków) z użyciem instrukcji assemblera.

Cel: nabycie umiejętności:

1. Organizacja pętli programowej:

- inicjalizacja wskaźników, nadanie wartości początkowych argumentom, (inicjalizacja licznika)
- aktualizacja wskaźników, dostęp do argumentów indeksowanych
- wytworzenie i bezpieczne przechowanie (zapis) wyniku działania na argumentach
- wytworzenie warunku ponowienia (porównanie) i/lub aktualizacja licznika
- rozgałęzienie – skok warunkowy do etykiety początku pętli

2. Struktury pętli: `for`, `while`, `until`

3. Zagnieżdżanie pętli – ochrona danych (argumentów, wskaźników i warunków)

Przykładowe zadania

1. Przetwarzanie tekstu

- Pobrać ze standardowego strumienia wejściowego procesu ciąg znaków i zamienić wszystkie małe litery na wielkie, wszystkie wielkie na małe, zmienić wielkość co drugiej litery itp.
- Zaszyfrować kluczem XOR ciąg liter ASCII ze strumienia wejściowego
- Zaszyfrować szyfrem Cezara ciąg liter ASCII ze strumienia wejściowego

2. Jak działają instrukcje i co jest zapisane w pamięci

- zbadac (gdb) dokładnie działanie instrukcji dzielenia (`div`, `idiv`) oraz mnożenia (`mul`, `imul`).
- ogłębnie obraz programu, jego danych i stosu programowego w pamięci komputera

3. Przetwarzanie liczb

- Pobrać z wejścia standardowego ciąg cyfr dziesiętnych reprezentujących liczbę całkowitą, zapisać jej wartość w pamięci i wyprowadzić na standardowe wyjście w zapisie szesnastkowym.
- Pobrać z wejścia standardowego ciąg znaków reprezentujący liczbę szesnastkową (znak +/-, cyfry 0...9, litery A...F), zapisać tę liczbę w kodzie naturalnym binarnym a liczbę przeciwną do niej w kodzie U2. Wyniki wyprowadzić na standardowe wyjście w zapisie szesnastkowym, binarnym i dziesiętnym. Uzupełnić kodem wykrywającym niepoprawne znaki.
- Binarną reprezentację liczby całkowitej zapisaną w kolejnych słowach pamięci wypisać na standardowe wyjście w postaci ciągu znaków reprezentujących jej wartość dziesiętną.
- Pobrać z wejścia standardowego ciąg znaków reprezentujący liczbę szesnastkową (znak +/-, cyfry 0...9, litery A...F, przecinek/kropka pozycyjna), i wyprowadzić na standardowe wyjście w zapisie dziesiętnym.

2. Tworzenie i użycie funkcji, funkcje rekurencyjne, obsługa plików w assemblerze procesora x86.

Cel: nabycie umiejętności:

1. Tworzenie i użycie funkcji
 - a) kontekst (blok aktywacji) funkcji
 - b) struktura kodu, zmienne lokalne i globalne,
 - c) przekazywanie argumentów przez wartość i przez wskaźnik
 - d) przekazywanie wyników działania funkcji i zakończenie funkcji
2. Zagnieżdżanie funkcji i funkcje rekurencyjne

Przykładowe zadania – funkcje obliczeniowe

1. Proste operacje arytmetyczne (dodawanie, odejmowanie, mnożenie) na liczbach rozszerzonej precyzji (rozmiar=wielokrotność słowa maszynowego) w kodzie NB i U2.
2. Generator liczb pierwszych zgodnie z algorytmem sita Erastotenesa.
3. Sortowanie tablicy liczb 64/96/128-bitowych.

Przykładowe zadania – funkcje rekurencyjne (wersja iteracyjna i rekurencyjna)

1. Odwrotność modulo na podstawie algorytmu Euklidesa dla liczb o dowolnej długości.
2. Liczby Fibonacciego wielokrotnej precyzji (krotność długości słowa maszynowego).
3. Obliczenie wartości wyrażeń kombinatorycznych

3. Technika łączenia różnych języków programowania, optymalizacja (analiza wydajności kodu).

Cel: nabycie umiejętności łączenia algorytmów assemblerowych i tworzonych w języku symbolicznym

Przykładowe zadania (! **wyłączyć opcje optymalizacji w kompilatorze gcc**)

1. Powiązania algorytmów, funkcji i zmiennych
 - a) użyć w assemblerze funkcji napisanych w języku C (funkcje biblioteczne i własne), sprawdzić (np. komendą **man**) działanie funkcji systemowych jeśli mają kilka wersji (np **exit** vs. **_exit**),
 - b) użyć w języku C funkcji napisanych w assemblerze (argumenty i wartości różnych typów),
 - c) użyć w assemblerze zmiennych i stałych zdefiniowanych w C i odwrotnie.
 - d) napisać w języku C i assemblerze funkcje działające na dużych tablicach dwuwymiarowych.
 - e) utworzyć funkcję w formie wstawki assemblerowej w języku C
2. Zmierzyć czas wykonania algorytmów za pomocą instrukcji **rdtsc** (ang. Time Stamp Counter) oraz za pomocą dostępnych narzędzi (np. **gprof**).
3. Na podstawie pomiarów czasu wykonania oraz analizy kodu assemblerowego powstałego podczas kompilacji zlokalizować najbardziej czasochłonne fragmenty programu i odpowiednio je modyfikując skrócić czas wykonania programu.

Literatura (<http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/...>)

[1] SYSTEM V APPLICATION BINARY INTERFACE ..., w tym:

- Function Calling Sequence (informacje o przekazywaniu parametrów do funkcji i zwrocie wartości).
- sposób działania stosu (instrukcje push, pop, enter, leave, call, ret) oraz rozmiary argumentów.

[2] IA32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture ,

[3] Antoni Myłka, Profilowanie i wydajność: gprof i strace (Krótkie wprowadzenie do profilowania kodu z użyciem gprof), <http://www.g2inf.one.pl/referaty/.mylka/gprof/index.html>

[4] Dokumentacja programu gprof, <http://sourceware.org/binutils/docs/gprof/index.html>

[5] Eric Youngdale, The ELF Object File Format: Introduction, (ELF F Format by Dissection) (Budowa plików z kodem wykonalnym i ich uruchamianie), <http://www.linuxjournal.com/article/1059>

4. Jednostka zmiennoprzecinkowa w architekturze IA-32e (x86/P#/i#).

Cel: nabycie umiejętności:

1. Rozpoznawanie statusu i ustanawianie trybu działania jednostki zmiennoprzecinkowej (FPU)
2. Rozpoznawanie i obsługa wyjątków
3. Kontrola dokładności obliczeń
4. Implementacja podstawowych metod numerycznych

Przykładowe zadania (interfejs do języka C, procedury assemblerowe)

- a) ustawienie i sprawdzenie statusu FPU (precyzja obliczeń, tryb zaokrąglania i reakcja na wyjątki)
- b) próby wykonania działań prowadzących do różnych błędów i obsługa tych błędów:
 - i) obsługa łagodna - korekta zakresu przez modyfikację wykładnika i zapamiętanie stosownej etykiety (np. nadmiar / niedomiar : $\pm 1/2$ amplitudy (2^{k-1})) np. znacznik wewnątrz funkcji
 - ii) ostrzeżenia: np. NaN, niedozwolony argument,
- c) obliczanie SQRT lub pola wg wzoru Herona dla krytycznych danych
- d) oprogramowanie w assemblerze lub C algorytmów obliczania wartości funkcji elementarnych (sin, cos, ln, exp) całek oznaczonych, przybliżeń szeregiem Taylora. Dobrać kolejność i liczbę operacji zmiennoprzecinkowych, aby uzyskać wynik mieszczący się w zadanej precyzji obliczeń.

Literatura (<http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/>)

[1] IA32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture ,

[2] D. Goldberg, What Every Computer Scientist Should Know About FloatingPoint Arithmetic, March 1991, issue of Computing Surveys., <http://dlc.sun.com/pdf/8007895/8007895.pdf>

5. Przetwarzanie sygnałów cyfrowych z użyciem jednostek MMX i SSE.

Cel: nabycie umiejętności:

1. Odczyt danych z plików sygnałowych .bmp, .jpg, .au, .avi?
2. Dostęp do informacji jednostkowej w pliku sygnałowym
3. Przetwarzanie informacji jednostkowych
4. Współbieżne przetwarzanie sygnałów obrazu i dźwięku

Przykładowe zadania:

1. Odczyt danych o rozdzielczości i strukturze obrazu z pliku .bmp, .jpg
2. Przekształcenia obrazu lub jego części bez zmiany jego treści: obrót, odwracanie (C lub asm)
3. Zamiana kolorów na stopnie szarości, tworzenie negatywu (C, asm, asm-mmx)
4. Rozjaśnianie i wygaszanie składowych koloru (C, asm, asm-mmx)
5. Rozmycie i konturowanie obrazu (C, asm, asm-mmx)

6. Odczyt atrybutów procesora i ich eksperymentalna weryfikacja

Cel: nabycie umiejętności:

1. Odczyt kodów identyfikujących (CPUID)
2. Dekodowanie – interpretacja kodów identyfikujących

Zadania

1. Odczyt informacji o strukturze buforów cache
2. Eksperymentalna weryfikacja parametrów bufora cache
3. Zbadanie skuteczności użycia buforów cache

LINUX – sesja

```

svn://lak.ict.pwr.wroc.pl/... albo
ssh lak.ict.pwr.wroc.pl      # (lak: 156.17.40.28)
    login: konto             # (L2.1 C16: "student"; 013 C3: "1student")
    passwd: hasło           # (L2.1 C16: "pwr-stud-013"; 013 C3: "stud013")

```

LINUX – użyteczne komendy ([.] – opcja)

```

ls [-al]                      # drukuj zawartość katalogu [all, long - szczegółowo]
ls [-al] ab*/ab              # lista plików, których nazwa zaczyna/kończy się od ab
mc                            # Midnight Commander (katalog okienkowy)
find filename                # wyszukaj plik o nazwie filename (dozwolone *)
pwd                          # wypisz nazwę katalogu roboczego
cd                            # zmiana katalogu (cd .. - zmień na katalog nadrzędny)
cat filename                 # drukuj zawartość pliku tekstowego filename
cp filename fcopy            # kopiuj zawartość pliku filename do pliku fcopy
rm filename                  # usuń plik filename
clear                        # wyczyść ekran
history [n]                  # wyświetl historię [n] ostatnich poleceń
history -c                   # wyczyść historię poleceń
↑/↓ (NumPad)                # nawigacja po historii komend

help command                 # pomoc dla komendy zewnętrznej (powłoki .bash)
man command                  # wywołanie instrukcji obsługi polecenia (manual-a)
    command --help           # wywołanie listy opcji polecenia
    „Ctrl”+D/„Ctrl”+C      # przerwanie procesu
vim                          # wierszowy edytor tekstowy
gvim                        # edytor vim z interfejsem graficznym
exit                        # zamknięcie sesji i wylogowanie

```

Tworzenie i uruchamianie programu – schemat podstawowy

```

vim plik.s                   # utwórz plik źródłowy (z tekstem programu) „plik.s”
                             # !! pamiętaj o kopii zapasowej pliku źródłowego
as plik.s [opcje]            # kompiluj źródłowy „plik.s” - wynik w pliku
ld plik.o [opcje]            # konsoliduj plik skompilowany - wynik: plik wykonalny
./prog                       # uruchom program „prog” z katalogu bieżącego (./)
./prog par1 par2 ...         # uruchom program z parametrami (dostępne na stosie,
                             # tak jak parametry wywołania →→ funkcji)

```

połączenie make (niezalecane, łatwo dostępna historia poleceń)

```

                             # przykład pliku wsadowego (do uruchamiania poleceniem make)
                             # zmiana opcji wymaga zmiany treści pliku wsadowego
plik: plik.o                 # reguła konsolidacji (linkowania)
    ld -o plik plik.o        # „Tab” konieczny na początku linii komendy, opcja -o plik wynikowy
plik.o: plik.s                # reguła kompilacji (asemblacji)
    as -o plik.o plik.s      # opcja -o : wskazanie nazwy pliku wynikowego

```

Edytor vim (wersja rozszerzona vi) [3]

vim [plik] # uruchomienie edytora [z nadaną nazwą pliku źródłowego]

tryby: (1) edycja tekstu; [Esc]+ZZ – zakończenie edycji z zapisem do pliku otwarcia
 (2) tryb komend (startowy) – przyjmowanie poleceń z klawiatury
 (3) polecenie edytowane (zapis, odczyt, wyszukanie, wyjście z vi) [Esc] - powrót do (2)

vim [plik]: → (2) → [a] (dołącz) / [i] (wstaw) /... / [p] (wkopiuj) → (1) ...(*edycja*)... (1) [Esc] → (2)
 (..) [Esc] → (2) ((2) [Esc] → „beep” → (2) – bez efektu, sygnał dźwiękowy błędnej sekwencji)
 (2) [:] / [/] / [?] → (3) – przejście do edycji wiersza poleceń ... [Esc] → (2)
 (3) :[w]q! [enter] = **w**rite and **q**uit ([zapisz zmiany,] zakończ i wróć do systemu)

(2) Tryb edycji tekstu

((2)[polecenie] → (1) edycja tekstu → [Esc] → (2))

i / I – insert, *wstaw przed kursorem / od początku wiersza (shift & insert)*

a / A – append, *wstaw za kursorem / na końcu wiersza (shift & append)*

x / X – extract, *usuń znak na pozycji kursora / przed kursorem*

p / P – paste *wstaw zawartość bufora za /przed kursorem lub do kolejnego /poprzedniego wiersza*

r / R – replace, *nadpisz znak na pozycji kursora / nadpisz tekst od pozycji kursora*

o / O – *wstaw wiersz poniżej / powyżej*

(polecenie → akcja (1) → (2))

dd/dw – delete, *usuń cały wiersz / słowo wskazane przez kursor*

d0 / d\$ – *usuń od początku wiersza do kursora / od kursora do końca wiersza (także D)*

yy/yw – yank, *kopiuj do bufora: aktualny wiersz / słowo wskazane przez kursor*

y0 / y\$ – *kopiuj od początku wiersza do kursora / od kursora do końca wiersza*

c.. – change, *zamień (usuń poprzedni tekst i wpisz nowy) (także d..i lub d..I)*

cc – *zamień aktualny wiersz (także [ddI]), cw* – *zamień słowo wskazane przez kursor (także dwi)*

c0 / c\$ – *zamień od początku wiersza do kursora (także d0i) / od kursora do końca wiersza (także d\$i)*

J – join, *przyłącz następny wiersz (usuń znak NewLine)...*

u – undo, *anuluj poprzednią akcję*

^r – redo, *powtórz poprzednią akcję*

Manewrowanie kursorem

Strzałkami: (znak) ← → ↑ ↓ (słowo) Ctrl+← Ctrl+→ albo z klawiatury

(3) Polecenia edytowane (z parametrami)

:r plik – wstaw (czytaj) plik

:w plik – write, zapisz do pliku (:w! *nazwa* – nadpisanie istniejącego pliku *nazwa*)

:q – quit, bez aktualizacji po ostatnim zapisie zmian (**:q!** – quit bezwarunkowo, bez zapisu zmian)

:wq [plik] – write & quit, zapisz do pliku otwarcia lub wskazanego (plik) i zakończ (także **:x**)

ZZ – bezwarunkowe zakończenie z zapisem zmian (jeśli plik ma nazwę) (= **:wq!**)

:help – wezwanie pomocy (**F1** – ekran pomocy) wyjście przez **:q**

/s/wzorzec/zamiennik[/g] – szukaj i zastąp (/g wszystkie wystąpienia w wierszu)

:1,\$s/wzorzec/zamiennik[/g] – szukaj i zastąp w całym pliku (od wiersza 1 do ostatniego)

:/j,k m t /:/j,k co t – przesun (move) /kopiuj (copy) wiersze od j do k począwszy od linii t

:/j,k d – usuń wiersze od j do k

/wzorzec | ?wzorzec – szukaj w pliku (tekście) wzorca od kursora do końca | początku

n / N – powtórz ostatnie wyszukiwanie w tym samym / przeciwnym kierunku

Opcje **vim** (w trybie 2), niedostępne w edytorze **vi**

– klawisze **Del** i **Backspace**

– Shift+Ins (w trybie wstawiania tekstu: **a/i** ...) – wstaw zawartość bufora

umożliwia kopiowanie tekstu ASCII z Win XP (Notepad) do edytora Ubuntu LINUX:

Notepad (zaznacz: Ctrl-A → wytnij do bufora: Ctrl-C) → vim / mcedit (otwórz plik → Shift-Ins)

– zaznaczenie myszą tekstu w oknie terminala (np. przez PuTTY) i skopiowanie do bufora:

umożliwia kopiowanie tekstu ASCII z okna LINUX (mcedit) do Win XP (Notepad):

vim / mcedit „plik” → okno terminala - mysz-R: „Copy All to Clipboard” → Notepad (Ctrl-V)

Uwaga: plik LINUX musi kończyć się pustą linią

Tryb poleceń – manewrowanie kursorem z klawiatury 80-klawiszowej (bez części NumPad)

[l] – →¹ (1 znak w prawo, także [space])

[h] – ←¹ (1 znak w lewo, także [backspace])

[j] / – ↓¹ (1 linia w dół)

[k] – ↑¹ (1 linia w górę)

[-] – ←←↑¹ (początek wiersza, linia w górę)

[+] – ←←↓¹ (początek wiersza, linia w dół = ⌵)

[0] – ←← (początek wiersza)

[\$] – →→ (koniec wiersza)

[w] – początek kolejnego słowa

[b] – najbliższy początek słowa

[Ctrl+f] / **[Ctrl+d]** – ekran / pół ekranu w dół

[Ctrl+b] / **[Ctrl+u]** – ekran / pół ekranu w górę

#[G] – początek wiersza o numerze #; **1[G]** – początek pliku, **[G]** – koniec pliku

[H] – początek bieżącego ekranu

Kompilacja (as), konsolidacja (ld) i uruchamianie programu

sourceware.org/binutils/docs2.17/as/index.html

!! Utwórz kopię zapasową pliku źródłowego (:w plik, :w! kopia) lub cp plik kopia
Tworzenie kodu dla środowiska 64-bitowego (IA-32e)

```
as plik.s -g -o plik.o      # plik wynikowy: plik.o, plik źródłowy: plik.s
                           # opcja -g: (generuj tablice powiązań dla debuggera)
      (as --help)          # pokaż listę opcji assemblera: -g, -o file, itp.)
      (as --gstabs+ ...)   # tablice debuggera dla kompilacji z optymalizacją
ld plik.o -o prog          # opcja -o: utwórz plik wykonalny o nazwie prog
      (ld --help)         # pokaż listę opcji konsolidatora
ld -m EMUL...             # wskazanie emulacji (wersji tworzonego kodu binarnego)
                           # (elf_i386/ i386linux/ elf_x86_64)
```

Użycie kompilatora C++ (etykieta startowa: _main zamiast _start)

```
gcc -g -o plik plik.s      # plik wykonywalny: plik, plik źródłowy: plik.s
                           # domniemane tworzenie kodu 64-bitowego (IA-32e)
```

Tworzenie kodu dla środowiska 32-bitowego (IA-32)

```
as --32 pr.s -g -o pr.o    # generuj kod 32 bitowy (na maszynie 64b AMD)
ld pr.o -m elf_i386 -o pr  # generuj kod 32-b na maszynie 64b w emulacji i386
```

Użycie kompilatora C++ (etykieta startowa: _main zamiast _start)

```
gcc -m32 -g -o pr pr.s     # plik wykonywalny: pr, plik źródłowy: pr.s
```

WYKONANIE: Konieczne podanie ścieżki dostępu do programu

```
./plik      # uruchomienie program z katalogu bieżącego (./)
echo $?     # 8-bitowy status zakończenia programu z %bl (to nie jest wynik!!)
```

Typowe błędy kompilacji:

- błędy składni (niepoprawne instrukcje, zła kolejność argumentów)
- brak symboli specjalnych (% przed nazwą rejestru, \$ przed wartością stałej)
- „unexpected end of file” – brak nowej linii w pliku źródłowym filename.s
- błędny kod znaku (podczas transferu z innego środowiska, np. windows)

Typowe błędy konsolidacji:

- wadliwa struktura programu (brak dyrektywy **glob(a)l**, brak otwarcia sekcji)
- brak definicji używanych symboli (domyślnie zewnętrzne dla kompilatora)

Typowe błędy wykonania:

„segmentation fault” (podczas wykonania)

- wskaźnik poza obszarem zmiennych (błąd indeksowania zmiennej)
- zmienne w sekcji **.text** (chroniona przed zapisem)
- brak dyrektywy **.data** przed deklaracją danych (deklaracja danych poza sekcją)
- **stos** programowy – przekroczenie dna stosu lub nadmierna rozbudowa
- błędne użycie funkcji systemowej (niepoprawne parametry)
- brak wywołania funkcji **exit (!)** (błędny nr lub argument **int \$syscall**)

„floating point error” – błąd obliczeń zmiennoprzecinkowych lub dzielenie przez 0

„arithmetic error” – błąd obliczeń stałoprzecinkowych (za duży iloraz)

Użycie debuggera gdb (linux-asm dla x86/IA-32/IA-32e)

```

## plik źródłowy i plik wykonalny powinny być w tym samym katalogu bieżącym/##
gdb ./plik                # program uruchomieniowy (debugger) z nazwą pliku
                           # wyświetlenie nagłówka: oczekiwanie na akcję: (gdb) ...

(gdb) break xyz            # ustawienie pułapki w miejscu xyz
                           # xyz = etykieta albo nazwa_funkcji albo nr linii

(gdb) clear xyz            # skasowanie pułapki w miejscu xyz
(gdb) disable x y ...     # zamaskowanie pułapki w miejscu x, y, ... (lub wszystkie)
(gdb) enable x y ...      # odmaskowanie pułapki w miejscu x, y, ... (lub wszystkie)

(gdb) run                  # uruchomienie załadowanego programu z poziomu (gdb)
                           # komunikat: „Starting program ...” – w razie zablokowania:
...                         # Ctrl-C; „Program received signal SIGINT, Interrupt.”#
(gdb) step                 # praca krokowa - wynik: (nr_linii   instrukcja)
                           (gdb) # oczekiwanie na akcję

(gdb) next                 # praca krokowa - funkcja jako pojedyncza instrukcja!
(gdb) finish               # wyjście z pracy krokowej
(gdb) continue             # wznow wykonanie po zatrzymaniu w pułapce
(gdb) kill                 # przerwij wykonanie programu

(gdb) print/w $reg         # wyświetl rejestr %reg (parametr $reg! w formacie
                           w: d – dziesiętnie, x – szesnastkowo (domyślny),
                           c – znakowo, o – ósemkowo, f – zmiennoprzecinkowo)

(gdb) print zmienna        # wyświetl wartość zmiennej
(gdb) x/rsf &addr          # wyświetl stan r jednostek pamięci o rozmiarze s (b –
                           bajt, h – półsłowo, w – słowo, g – dwusłowo) w formacie f
                           jak wyżej, a także i = instrukcja (r=1, s=w można pominąć)

(gdb) x &addr              # wyświetl stan r jednostek pamięci o rozmiarze s (b –
(gdb) display/w arg        # w – jak print, ale po każdej pułapce (stopie)
(gdb) display n &addr      # n komórek od adresu addr
(gdb) info                 # informacja o: rejestrach, pamięci,
(gdb) info reg              # informacja o rejestrach
(gdb) info breakpoints     # informacja o ustawionych pułapkach

(gdb) list                 # listing programu
(gdb) help [cmd]           # lista poleceń lub opis polecenia debugera [cmd]
(gdb) <ENTER>              # powtórz poprzednią akcję, jeśli jest to sensowne
(gdb) quit                 # wyjście z programu debuggera

```

symbole parametrów poleceń

\$reg – rejestr

*nazwa – wskaźnik

&zmienna – wartość zmiennej

Reprezentacja stałych w środowisku UNIX/Linux (as, gcc)**Zapis wartości liczbowych (znak liczby opcjonalny) i kodów**

```

[-]zc..c      # liczba dziesiętna, z=1,...,9, c=0,1,...,9, np. -32, 15
0d[-]c..c     # liczba dziesiętna (decimal), c=0,1,...,9, np. -32, 0d713
0x[-]h..h     # liczba szesnastkowa (hexadecimal), h=0,1,...,9,a,b,...,f
0q..q         # liczba ósemkowa, q=0,1,...,7, np. 031,
0bt..t        # liczba dwójkowa (binary), t=0,1, np. 0b1011
0f[-]rE[-]e   # liczba float, r,e dziesiętnie, e całkowite np. -314,59xE-2
'β'           # kod ASCII znaku alfanumerycznego β
"tekst"       # ciąg kodów ASCII znaków alfanumerycznych

```

Zapis znaków specjalnych ASCII w tekście (\="escape" – następny to specjalny) – konwencja LINUX

```

\ddd          # znak ASCII jako 3 cyfry ósemkowe
\xDD          # znak ASCII jako 2 cyfry szesnastkowe
\n            # LF,NL, zmiana wiersza (new line) (=x0A=x012)
\0            # NUL, koniec rekordu (=x00=x000)
\b            # BS, cofnięcie (backspace) (=x08=x010)
\t            # HT,TAB, znak tabulacji (tabulation) (=x09=x011)
\f            # FF, nowa strona (form feed) (=x0C=x014)
\r            # CR, powrót kursora (carriage return) (=x0D=x015)
\\ | \' | \?  # ukośnik, apostrof, znak zapytania (znaki specjalne)

```

UWAGA:

W pliku źródłowym assemblera **as** specyfikacja każdej stałej jako argumentu instrukcji musi być poprzedzona znakiem \$, np. \$05, \$\n, \$'a'

Obsługa kodów ASCII (przykład) – składnia assemblera as (AT&T/Linux)

a) obliczenie wartości znaku ASCII reprezentującego cyfrę szesnastkową (hex)

```

slot_c = 'A'-'9'-1    # odległość cyfr hex od dec = 7, więc lit_hex ma wartość 'lit_hex' – 0x37
stand = 'a'-'A'       # 'a'-'A' = 0x20=0b00100000, 'a'XOR'A' # standaryzacja kodu litery na małe
mask = 0xDF           # NOT 0x20 = 0xDF = 0b11011111 – maska „małe na wielkie”
                        # kopia znaku w rejestrze al
cmpb $'0',%al         # sprawdź, czy to kod cyfry dziesiętnej lub litery
jlt out               # wyjdź jeśli to nie jest kod cyfry
cmpb $'9',%al         # sprawdź, czy to kod cyfry dziesiętnej
jle hop               # jeśli to kod cyfry przejdź do obliczenia
andb $mask, %al       # nie-cyfra dziesiętna, zamiana litery na wielką
cmpb $'F',%al         # sprawdź, czy to kod cyfry hex
jgt out               # wyjdź jeśli to nie jest kod cyfry hex
subb $slot, %al,      # redukcja kodu wielkiej hex, 'A' ... 'F' uzyskuje kod 0x3A ... 0x3F
hop:                  #
andb $0xF, %al        # obliczenie wartości cyfry dziesiętnej lub hex w bajcie kodu

```

b) sprawdzenie, czy znak ASCII jest literą (kopia znaku w rejestrze al.)

```

orb $0x20, %al        # wszystkie małe (alternat. andb $0xDF, %al – wszystkie wielkie)
cmpb $'a',%al         # sprawdź, czy może być mała litera (%al ≥ 'a')
jbe not_letter        # jeśli (%al < 'a') to nie jest to kod litery
cmpb $'z',%al         # sprawdź, czy może być mała litera (%al ≤ 'z')
jbe letter            # jeśli (%al > 'z') to nie jest to kod litery
not_letter:           # to nie jest litera
    ...               # obsługa nie-liter
letter:               # to jest litera
    ...               # obsługa liter

```

Struktura programu w języku assemblera ATT – linux-asm dla IA-32 (80386+/Pentium)**Dyrektywy organizacyjne**

```
.globl etykieta      # dyrektywa udostępnienia etykiety poza plikiem
.globl _start|main   # etykieta startowa (_start – dla as, _main – dla gcc)
.data /.section data # sekcja danych programu, zawiera deklaracje zmiennych
.text /.section text  # sekcja algorytmu (tekst programu) – sekcja obowiązkowa
.bss /.section bss    # bufor nieinicjowanych danych globalnych (s.55/61 [1])

.type nf @function   # deklaracja funkcji nf udostępnianej poza plikiem
.include FILE        # wstawia specyfikowany plik jako część tekstu programu.
.section NAME        # deklaracja początku sekcji NAME
.code32              # wymuszenie wpasowanego kodu danych (32-b) w pliku
                    # wynikowym kompilatora, konsolidacja z opcją emulacji!
.size zmienna ile    # rozmiar zmiennej w bajtach
```

Stała i łańcuch znaków,

```
.equ VAL, stała      # lokalne przypisanie wartości (.data)
VAL = stała          # globalne przypisanie wartości (.data, .text)
napis: .ascii "Hi\n" # łańcuch znaków ASCII (txt: .ascii = txt: .ascii „\n”)
```

Zmienna, jej adres i rozmiar (inicjalizacja obowiązkowa)

```
dane:                # deklaracja zmiennej (bloku danych) o adresie „dane”:
.typ lista [,lista]  # inicjalizacja obowiązkowa, typ =.byte|.ascii|.long|.float
dane_size =.-dane     # bezpośrednio za deklaracją zmiennej: obliczany przez as
                    # rozmiar zmiennej w bajtach (nazwa zmiennej bez znaku $)
    dane              # wartość pierwszego elementu (bajtu/słowa) zmiennej „dane”
                    # ciąg cyfr/liter oznacza zmienną o wskazanym adresie, np.
    3145              # wartość zmiennej spod adresu 3145 (ósemkowo!)!!!
    $dane             # adres logiczny pierwszego bajtu zmiennej „dane” (wskaźnik)
inicjowana:
.rept (ile_razy)      # liczba powtórzeń wartości zmiennej inicjowanej
.(typ) value          # wartość powtarzana
.endr                 # koniec powtórzenia
```

Rezerwacja bufora w pamięci (inicjalizacja opcjonalna)

```
blok:                # deklaracja bufora pamięci o nazwie blok w sekcji danych
.space num [,fill]    # num – rozmiar [B], fill – wartość wypełnienia (dom. 0)
(.skip num)           # rezerwacja bloku bez inicjalizacji
```

Rezerwacja bufora danych nieinicjowanych – sekcja bss

```
.equ B_SIZE, num     # z wielu powodów num nie powinno przekraczać 16000.
.bss                  # początek sekcji bufora (bss) poza sekcją danych
.lcomm ADDR, B_SIZE   # alokuje w pamięci bufor ADDR o rozmiarze B_SIZE bajtów
.comm ADDR, B_SIZE    # alokuje w pamięci bufor ADDR o rozmiarze B_SIZE bajtów
```

Składnia AT&T/Linux

```

memento arg-s, arg-ak      # arg-s – argument swobodny (stała, rejestr, zmienna)
                           # arg-ak – argument akumulujący (rejestr, zmienna)
                           # arg-ak:= arg-ak działanie arg-s
                           # wymagana zgodność rozmiarów argumentów!

```

% - wskazanie rejestru procesora jako argumentu instrukcji (np. %esi, %ah, %esp, %eax,...)

Wskazanie argumentu w pamięci – tryb adresowania

```

zmienna(rej_1, rej_2, skala)

```

Rejestry – tryb 64b

```

%r#b, %r#w, %r#d          # dolne bity r8-r15 (b-bajt, w-półsłowo, d-słowo) np. %r13d

```

Podstawowa lista instrukcji

```

mov                # kopiowanie
or                 # suma logiczna (bitowa)
xor                 # suma wykluczająca (bitowa)
and                 # iloczyn logiczny (bitowy)
test               # iloczyn logiczny (bitowy) bez zapisania wyniku
adc                # dodawanie z przeniesieniem
    add            # dodawanie pojedyncze (CF=0)
sbb                # odejmowanie z pożyczką
    sub            # odejmowanie pojedyncze (CF=0)
mul                # mnożenie naturalne
div                # dzielenie naturalne
imul               # mnożenie całkowite
idiv               # dzielenie całkowite
inc                # zwiększenie (wskaźnika)
dec                # zmniejszenie (wskaźnika)
lea                # obliczenie adresu na podstawie trybu adresowania
xchg               # kopiowanie wzajemne
jwar               # rozgałęzienie jeśli warunek „war” jest spełniony
call               # wywołanie funkcji
ret                # zwrot sterowania do poziomu wywołania funkcji
push               # skopiowanie argumentu na szczyt stosu
pop                # skopiowanie ze szczytu stosu do argumentu

loop               # powtarzaj dopóki licznik≠0
int $0x80          # wywołanie funkcji systemowej w trybie 32-b
syscall            # wywołanie funkcji systemowej w trybie 64-b

!(sysenter)        # przełączenie na wirtualny tryb 64-b (nowa instr. IA-32e)
!(sysexit)         # powrót z wirtualnego trybu 64-b i zwrot sterowania do SO

```

Najważniejsze funkcje systemu Linux, tryb 64-b (wynik w %rax)

nazwa	%rax	%rdi	%rsi	%rdx	Uwagi (wynik funkcji)
sys_exit	60	kod błędu int error_code			restart – zwrot sterowania do s.o.
sys_read	0	numer pliku unsigned int fd	adres bufora char *buf	rozmiar bufora size_t count	odczyt pliku; %rax: liczba bajtów
sys_write	1	numer pliku unsigned int fd	adres bufora const char *buf	rozmiar bufora size_t count	zapis do pliku; %rax: liczba bajtów
sys_open	2	nazwa pliku const char *file	lista opcji int flags	kod dostępu int mode	%rax: deskryptor pliku lub kod błędu
sys_close	6	deskryptor pliku unsigned int fd			zamyka plik o danym deskrypcorze.
sys_brk	12	unsigned long brk			
sys_chdir	80	const char *filename			
sys_mkdir	83	const char *pathname	int mode		
sys_rmdir	84	const char *pathname			
sys_getpid	39				

Lista funkcji: Linux 4.7 (pulled from github.com/torvalds/linux on Jul 20 2016), x86_64

```

                                # schemat użycia funkcji dla x86-64 (IA-32e)
movq $FILE_ID, %rdi           # identyfikator pliku (STDIN=0, STDOUT=1) (%ebx w IA-32)
movq $PAR1, %rsi              # parametr 1, np. adres bufora we/wy (%ecx w IA-32)
movq $PAR2, %rdx              # parametr 2, np. rozmiar bufora (%edx w IA-32)
movq $FUNC_ID, %rax           # numer funkcji do %rax (%eax w IA-32)
syscall                       # ogólne wywołanie funkcji (nowa instrukcja IA-32e)

```

Najważniejsze funkcje systemu Linux [1], tryb 32-b

nazwa	%eax	%ebx	%ecx	%edx	Uwagi
exit	1	status (int)			restart – zwrot sterowania do s.o.
read	3	numer pliku ¹⁾	adres bufora	rozmiar bufora	odczyt pliku – zwraca w %eax l.bajtów
write	4	numer pliku ¹⁾	adres bufora	rozmiar bufora	zapis do pliku – zwraca w %eax l.bajtów
open	5	nazwa pliku ²⁾	tryb dostępu (mode)	kod dostępu rwx (lub 0666)	zwraca w %eax deskryptor pliku o danej nazwie lub kod błędu.
close	6	deskryptor pliku			zamyka plik ¹⁾ o danym numerze.
chdir	12	nazwa katalogu ⁴⁾			przełącza do wskazanego katalogu.
getpid	20				identyfikator ID bieżącego procesu.
mkdir	39	nazwa katalogu ⁴⁾	kod dostępu		tworzy katalog podrzędny
rmdir	40	nazwa katalogu ⁴⁾			usuwa katalog.
brk	45	adres pułapki			jeśli %ebx=0, zwraca bieżący adres pułapki

¹⁾ nadawany przez LINUX deskryptor (numer) pliku (*file descriptor*), ²⁾ pierwsze litery nazwy pliku

³⁾ opcje dostępu – odczyt lub/i zapis, ⁴⁾ pierwsze litery nazwy katalogu, ⁵⁾ 0 – absolutny, 1 – względny

Prawie kompletna lista funkcji systemu jest na stronie <http://www.lxhp.in-berlin.de/lhpsyscall.html>.

```

                                # schemat użycia funkcji dla x86 (IA-32)
movl $FILE_ID, %ebx           # identyfikator pliku (STDIN=0, STDOUT=1)
movl $PAR1, %ecx              # parametr 1, np adres bufora, do %ecx
movl $PAR2, %edx              # parametr 2, np. rozmiar bufora, do %edx
movl $FUNC_ID, %eax           # nr funkcji do %eax
int $SYSCALL32                # wywołanie funkcji systemowej (SYSCALL32 = 0x80)

```

Pierwsze programy (środowisko IA-32, kod 32-b)**## wczytanie tekstu – przetworzenie – wyświetlenie tekstu przetworzonego ##**

```

SYSCALL32 = 0x80      # sysfun: nr funkcji w %eax, parametry: %ebx, %ecx, %edx
EXIT = 1              # nr funkcji restartu (=1) – zwrot sterowania do s.o.
STDIN = 0              # nr wejścia standardowego (klawiatura) do %ebx
READ = 3              # nr funkcji odczytu wejścia (=3)
STDOUT = 1            # nr wyjścia standardowego (ekran tekstowy) do %ebx
WRITE = 4             # nr funkcji wyjścia (=4)
BUF_SIZE = 254        # rozmiar bufora (w bajtach/znakach ASCII) – max 254
DISTANCE = 'z'-'a'+1
COMPL = 'z'+'a'

.data                # przygotowanie bufora wejścia
komunikat: .ascii "Hello\n"      # tekst komunikatu,
rozmiar = . - komunikat      # obliczenie liczby znaków komunikatu
TEXT_SIZE .long 0
BUF: .space BUF_SIZE      # deklaracja bufora wejścia
[BUF_SIZE =.- BUFOR]      # obliczony rozmiar bufor (w bajtach) jeśli trzeba

.globl _start
.text
_start:
movl $rozmiar, %edx      # rozmiar bufora w bajtach ($rozmiar) do %edx
movl $komunikat, %ecx     # adres startowy bufora ($komunikat) do %ecx
movl $STDOUT, %ebx       # identyfikator wyjścia (pliku) (STDOUT=1)
movl $WRITE, %eax        # wyprowadź tekst powitalny z bufora (pamięci)
int $SYSCALL32           #

movl $BUF_SIZE, %edx     # rozmiar bufora w bajtach ($BUF_SIZE) do %edx
movl $BUFOR, %ecx        # adres początku bufora ($BUFOR) do %ecx
movl $STDIN, %ebx        # identyfikator wejścia (pliku) (STDIN=0) do %ebx
movl $READ, %eax         # wczytaj plik wejściowy do bufora aż do znaku „ENTER”
int $SYSCALL32           # (do końca pliku - znaku 0)
movl %eax, TEXT_SIZE     # w %eax funkcja zwraca liczbę wczytanych znaków

    CALL ENCRYPT          # przetwórz znaki za pomocą procedury „ENCRYPT”

movl TEXT_SIZE, %edx     # faktyczna liczba znaków w bajtach do %edx
movl $BUFOR, %ecx        # adres początku bufora ($BUFOR) do %ecx
movl $STDOUT, %ebx       # identyfikator pliku (STDOUT=0) do %ebx
movl $WRITE, %eax        # wyprowadzenie zawartości bufora na wyjście
int $SYSCALL32           # wywołanie funkcji

[movl $num, %ebx]        # w %ebx kod stanu, „echo $” zwraca %bl (8 niższych bitów)
movl $EXIT, %eax         # restart – obowiązkowe zakończenie programu
int $SYSCALL32           # wywołanie funkcji

```

```
ERR_MSG .asci „Niepoprawny znak\n”
```

```
ER_LEN=.- ERR_MSG
```

```
ENCRYPT:                # wielka litera – szyfrowanie, mała litera – deszyfrowanie
movl $0, %edi           # pierwszy znak jest kluczem
movb BUF(,%edi,1), %bl  # klucz (pierwszy znak) do %al
test $0x20, %bl         # duże mają bit b5=0, małe b5=1
jz szyfruj              # jeśli duża (b5=0) szyfruj, klucz w %bl
subb $COMPL, %bl        # -klucz_deszyfr=klucz-(‘a’+‘z’) (mała litera)
negb %bl                # klucz_deszyfr=(‘a’+‘z’)-klucz
andb $0xDF, %bl         # klucz – duża litera (xorb $0x20, %bl)
szyfruj:
incl %edi               # indeksacja wskaźnika znaku w buforze
movb BUF(,%edi,1), %al  # ujednolicenie – wszystkie litery duże
andb $0xDF, %al         # czy znak jest literą
cmpb $‘Z’, %al          # czy znak jest literą
ja error
subb $‘A’, %al          # czy znak jest literą
jb error
addb %bl, %al           # szyfrowanie – dodanie klucza
cmpb $‘z’, %al
jbe szyfruj
subb $DISTANCE, %al     # korekta cyklu
cmpl TEXT_SIZE, %edi
jbe szyfruj
ret
error:                  # znak nie jest literą
movl $WRITE, %eax       # wyprowadzenie zawartości bufora na wyjście
movl $STDOUT, %ebx      # identyfikator pliku (STDOUT=0) do %ebx
movl $ERR_MSG, %ecx     # adres początku bufora ($BUFOR) do %ecx
movl $ER_LEN, %edx      # faktyczna liczba znaków w bajtach do %edx
int $SYSCALL32          # wywołanie funkcji
ret
```

Uwaga:

Najprostsze szyfrowanie /deszyfrowanie – XOR z użyciem wybranej maski

Koncepcja pliku w systemie UNIX / LINUX – obsługa plików z poziomu assemblera

Pliki UNIX /LINUX, niezależnie od rodzaju i sposobu wytworzenia, są dostępne jako łańcuch bajtów. Dostęp do pliku rozpoczyna jego otwarcie przez podanie nazwy. Wtedy system operacyjny podaje (tymczasowy) numer, zwany deskryptorem pliku (*file descriptor*), używany jako odsyłacz do pliku podczas jego użycia. Po zapisie lub odczycie plik należy zamknąć, co unieważnia deskryptor.

Postępowanie z plikami (dealing with files) – tryb 32-b

1. Podaj do systemu Linux nazwę pliku i żądany tryb otwarcia (odczyt, zapis, odczyt i zapis, utwórz go jeśli nie istnieje, itd.). Wykonuje to funkcja `open` (`%eax = 5`), która pobiera nazwę pliku, kod trybu oraz zbiór kodów dostępu (*permissions set*) jako parametry. Adres pierwszego znaku nazwy pliku powinien być w `%ebx`. W `%ecx` należy wpisać kod trybu użycia (0 dla plików, które będą tylko odczytywane, 03101 dla plików, które będą zapisywane (! zero wiodące jest konieczne). Zbiór dostępu ma być wpisany do `%edx`. Jeśli nie znasz kodów dostępu UNIX / LINUX, wpisz kod 0666 (! zero wiodące jest konieczne – patrz [1: rozdział 10, sekcja *Truth, Falsehood, and Binary Numbers*]).
2. LINUX zwróci w `%eax` deskryptor pliku (*file descriptor*), który jest odsyłaczem do tego pliku.
3. Teraz można wykonać funkcje `read` / `write` (`%eax = 3/4`), wpisując deskryptor pliku do `%ebx`, adres bufora danych do `%ecx`, rozmiar bufora do `%edx`. Funkcja (`read/write`) zwróci (w `%eax`) liczbę znaków przeczytanych z pliku, albo kod błędu (*error code*), który jest liczbą ujemną w systemie U2).
4. Po zakończeniu operacji plik należy zamknąć za pomocą funkcji `close` (`%eax = 6`), której jedynym parametrem jest deskryptor pliku (w `%ebx`). Deskryptor jest odtąd unieważniony.

Bity kodu opcji dla funkcji lub *system call* nazywa się *flags*. Parametrem funkcji systemowej `open` jest lista flag tworząca kod dostępu zapisany w rejestrze `%edx`. Niektóre z tych kodów (ósemkowo) to:

<code>O_RDONLY</code>	– 00	– tylko odczyt (read-only mode)
<code>O_WRONLY</code>	– 01	– tylko zapis (write-only mode)
<code>O_RDWR</code>	– 02	– zapis i odczyt (both reading and writing)
<code>O_CREAT</code>	– 0100	– utworzenie pliku, jeśli nie istnieje (create the file if it doesn't exist)
<code>O_TRUNC</code>	– 01000	– skasuj zawartość pliku, jeśli istnieje (erase the contents of the file)
<code>O_APPEND</code>	– 02000	– Dopisywanie na końcu pliku (start writing at the end of the file)

Flagi można logicznie sumować (OR), np. `O_WRONLY` (01) OR `O_CREAT` (0100) daje kod 0101.

Pliki standardowe i specjalne

W filozofii Linux/UNIX każde źródło danych (połączenie sieciowe, urządzenia) jest plikiem.

Komunikacja jest realizowana za pomocą specjalnych plików, zwanych rurami (*pipes*). Niektóre z nich wymagają specyficznych sposobów tworzenia i otwierania (nie używa się funkcji `open`), ale mogą być czytane i zapisywane normalnie (`read` / `write` z system calls).

W systemie Linux programy startują zwykle z trzema deskryptorami plików standardowych:

STDIN – file descriptor 0

Wejście standardowe (*standard input*), zawsze w trybie read-only, zwykle klawiatura.

STDOUT – file descriptor 1

Wejście standardowe (*standard output*), zawsze w trybie write-only file, zwykle monitor.

STDERR – file descriptor 2

Błąd standardowy (*standard error*), plik write-only, zwykle monitor. Wyjścia przetwarzania wędrują do STDOUT komunikaty o błędach do STDERR. Można je rozdzielić na 2 osobne miejsca.

Każdy standardowy strumień danych można w Linux-ie przekierować do innego pliku.

Obsługa plików

```

## Nazwy plików przekazywane przez stos przed wywołaniem procedury ##
SYSCALL32 = 0x80          # syscall - parametry: %eax, %ebx, %ecx, %edx
EXIT = 1                 # nr funkcji restartu (=1) - zwrot sterowania do s.o.
STDIN = 0                 # nr wejścia standardowego (klawiatuara) do %ebx
READ = 3                  # nr funkcji odczytu wejścia (=3)
STDOUT = 1                # nr wyjścia standardowego (ekran tekstowy) do %ebx
WRITE = 4                 # nr funkcji wyjścia (=4)
OPEN = 5                  # (opcje otwarcia: /usr/include/asm/fcntl.h)
CLOSE = 6                 # nr funkcji zamknięcia pliku
CR_WRONLY_TR = 03101     # flaga: tylko zapis (notacja ósemkowa!)
RDONLY = 0                # flaga: tylko odczyt, składanie opcji - OR

.section .bss              # bufor danych
.lcomm BUFFER, 500         # (rozmiar musi być <16000)

.text                      # wywołanie z nazwami plików: ./prog file-in file-out
.equ FD_IN, -4             # lokalizacja deskryptora pliku we (par1)
.equ FD_OUT, -8            # lokalizacja deskryptora pliku wy (par1)
.equ ARG_1, 8              # lokalizacja nazwy pliku we (par1)
.equ ARG_2, 12             # lokalizacja nazwy pliku wy (par1)

.globl _start              ##START PROGRAMU##
start:
movl %esp, %ebp            # przechowanie wskaźnika stosu
subl $8, %esp              # miejsce na stosie na deskryptory plików (2*4 bajty)

movl $OPEN, %eax           # otwarcie pliku wejściowego
movl ARG_1(%ebp), %ebx      # nazwa pliku we (file-in) do %ebx
movl $RDONLY, %ecx         # flaga: tylko do odczytu
movl $0666, %edx           # bez znaczenia podczas otwierania
int $SYSCALL32
movl %eax, FD_IN(%ebp)     # zwrócony w %eax deskryptor pliku we do ramki stosu

movl $OPEN, %eax           # otwarcie pliku wyjściowego
movl ARG_2(%ebp), %ebx      # nazwa pliku wy (file-out) do %ebx
movl $CR_WRONLY_TR, %ecx   # flaga: tylko do zapisu
movl $0666, %edx           # tryb dla tworzonego pliku (jeśli nowy)
int $SYSCALL32
movl %eax, FD_OUT(%ebp)    # zwrócony w %eax deskryptor pliku wy do ramki stosu

read_loop_begin:          ##ODCZYT BLOKU Z PLIKU WEJŚCIOWEGO##
movl $READ, %eax
movl FD_IN(%ebp), %ebx     # pobranie z ramki deskryptora pliku odczytywanego
movl $BUFFER, %ecx         # adres bufora odczytu
movl $B_SIZE, %edx         # rozmiar bufora odczytu
int $SYSCALL32             # rozmiar bufora odczytu zwracany w %eax

```

```

    cml $0, %eax          # sprawdzenie, czy osiągnięto koniec pliku EOF
    jle end_loop          # jeśli wykryto EOF lub w razie błędu koniec

                                # argumenty wywołania przekazywane przez stos
    pushl $BUFFER         # adres bufora na stos
    pushl %eax            # rozmiar bufora (zwrócony w %eax) na stos
    call convert
    popl %edx             # rozmiar bufora ze stosu do %edx
    addl $4, %esp         # przywrócenie wskaźnika stosu %esp

    movl $WRITE, %eax     # blok po konwersji do pliku wyjściowego
    movl FD_OUT(%ebp), %ebx # pobranie z ramki deskryptora pliku wynikowego
    movl $BUFFER, %ecx    # adres bufora (pliku zapisywanego)
    int $SYSCALL32
    jmp read_loop_begin   # kontynuacja – następna porcja pliku

end_loop:
    movl $CLOSE, %eax     ###ZAMYKANIE PLIKÓW – nie ma potrzeby kontroli
    movl FD_OUT(%ebp), %ebx # błędu, nie ma to tutaj istotnego znaczenia
    int $SYSCALL32        # deskryptor pliku wynikowego
    movl $CLOSE, %eax
    movl FD_IN(%ebp), %ebx
    int $SYSCALL32
    movl $EXIT, %eax      ##EXIT##
    movl $0, %ebx
    int $SYSCALL32

convert:
    movl 12(%ebp), %eax    # jeśli bufor ma długość zero, funkcja nie jest użyta
    movl 8(%ebp), %ecx    # adres bufora
    decl %eax             # rozmiar bufora
                                #
convert_loop:
    movb -1(%eax,%ecx,1), %dl # element N jest pod adresem (%eax)+N-1
    cmpb $'a', %dl        # kolejny bajt (znak ASCII)
    jb next_byte
    cmpb $'z', %dl
    ja next_byte          # jeśli znak poza ('a' do 'z') weź kolejny
    andb $0xDF, %dl       # zamień na dużą i zapisz zwrótnie do bufora
    movb %dl, -1(%eax,%ecx,1)
next_byte:
    loop convert_loop     # kontynuuj jeśli nie osiągnięto końca bufora (ecx--)
    movl %ebp, %esp
    pop %ebp
    ret

```

Makra

Makro tworzy makrodefinicja (*macrodefinition*) i makrowywołanie (*macrocall*).

Makrodefinicja jest tekstowym opisem treści podobnych fragmentów kodu źródłowego w formie, która umożliwia automatyczne tworzenie podobnych fragmentów kodu, na przykład sekwencji instrukcji, które różnią się tylko argumentami. Makrodefinicję rozpoczyna dyrektywa **.macro** po której następuje **nazwa makra** i lista parametrów formalnych, a kończy dyrektywa **.endm**.

Makrowywołanie to użycie **nazwy makra**, po której następuje specyfikacja parametrów w kolejności podanej w makrodefinicji.

```
# makrodefinicje #
.macro write str, str_size, dest    # makro o nazwie „write”
    movl $WRITE, %eax
    movl \dest, %ebx
    movl \str, %ecx
    movl \str_size, %edx
    int $0x80
.endm                                # zakończenie makrodefinicji - dyrektywa .endm

.macro read buf, buf_size, source  # makro o nazwie „read”
    movl $READ, %eax
    movl \source, %ebx
    movl \buf, %ecx
    movl \buf_size, %edx
    int $0x80
.endm

#      wywołanie      #
write $msg1, $msg1_size, $STDOUT    # printf("%s", txt_msg1)
                                     # wyświetla txt_msg1 na monitorze (STDOUT)
read $in-buff, $in-buff_size, $STDIN # scanf("%s", bufor_wej)
                                     # odczytuje znaki z klawiatury (STDIN)
```

Funkcje

stos programowy – dynamiczna struktura danych wspomagająca użycie funkcji

adres powrotu (*return address*) – parametr automatyczny, tworzony na stosie przez wywołanie (call)

zmienne globalne – dostępne i zarządzane na zewnątrz funkcji

zmienne lokalne – używane tylko wewnątrz funkcji, ignorowane po zakończeniu

zmienne statyczne – dostępne tylko wewnątrz funkcji, pamiętane do kolejnego jej wywołania

Konwencje wywołania funkcji w języku C (*calling convention*)

Wskaźnik stosu %esp wskazuje lokalizację bajtu zajmującego szczyt stosu (konwencja Little Endian):

- operacja pushl wykonywana jako przesłanie słowa pod adres $-4(\%esp)$, czyli $esp-4$
- operacja popl wykonywana jako przesłanie słowa spod adresu (%esp)
- dostęp do słów poniżej szczytu stosu – adres $N*4(\%esp)$ – N to numer kolejny parametru

Wartość wskaźnika stosu może się zmienić podczas wykonania funkcji (np. skutek przerwania), więc dostęp do struktur danych funkcji wymaga użycia wskaźnika powiązania dynamicznego (%ebp w konwencji C dla IA-32). Pierwszą instrukcją funkcji musi więc być zachowanie „starego” wskaźnika i załadowanie %ebp nową wartością (którą jest aktualna wartość wskaźnika stosu %esp). Przed zakończeniem funkcji trzeba odtworzyć „stary” %ebp.

Jeśli kod assemblerowy jest wstawiany do programu w języku C należy także pamiętać, że kompilator gcc dla IA-32 [5] przypisuje rejestry %ebp, %esp, %ebx, %edi, %esi funkcji wywołującej, więc funkcja wywoływana powinna chronić nie tylko %ebp, %esp ale też %ebx, %edi i %esi (zachować i odtworzyć przy zakończeniu). Rejestry %eax, %ecx, %edx są przypisane funkcji wywoływanej, więc funkcja wywołująca powinna je przechować (na stosie), jeśli ich używa w chwili wywołania [5].

Funkcja w programie - schemat wywołania

```
.text
.global _start
_start:
.....          # wcześniejsza część programu
pushl PAR-N      # przekazanie parametru nr N
...
pushl PAR-1      # przekazanie parametru nr 1
call funkcja     # wywołanie, adres powrotu na szczyt stosu
    (addl $N*4, %esp) # oczyszczenie stosu (jedna z możliwości)
.....          # obsługa wyników funkcji
```

Struktura funkcji (konwencja C/C++)

```
.type funkcja @function # deklaracja funkcji
funkcja:
[„push ...”]          # rejestry f. wywołującej na stos (gcc)
pushl %ebp            # wskaźnik kontekstu poziomego wywołania (powiąz. dynam.)
movl %esp, %ebp       # wskaźnik kontekstu funkcji wywołanej: szczyt stosu
...                   # „ciało funkcji” (dostęp swobodny do kontekstu)
movl %ebp, %esp       # przywrócenie wskaźnika szczytu stosu
popl %ebp             # odtworzenie „starego” wskaźnika powiązania
[„pop ...”]           # rejestry f. wywołującej ze stosu (gcc)
ret                  # zwrot sterowania do miejsca wywołania, odtąd zmienne
                    # lokalne są niedostępne, na stosie pozostały parametry
```

Funkcja „wyświetl dziesiętnie zawartość rejestru 32-b”

```

SYSCALL32 = 0x80          # nr wywołania systemowego
WRITE = 4                 # nr funkcji „pisz”
STDOUT = 1                # nr wejścia standardowego
.data
BUFVY: .ascii „\n”      # miejsce na 10 cyfr
BWY_LEN=. -BUFVY          # rozmiar bufora
.equ PODSTAWA, 10
.equ NUMB_LEN, 10         # 10**9<2**32<10*10 (liczba max 10-cyfrowa w rej. 32-b)

.type p_reg_dec @function # wyświetlenie zawartości rejestru dziesiętnie
p_reg_dec:
[„push ...”]             # rejestry f. wywołującej na stos (gcc)
pushl %ebp                # „stary” wskaźnik powiązania dynamicznego na stos
movl %esp, %ebp           # „nowy” wskaźnik (%esp)
movl $PODSTAWA, %ebx
movl $NUMB_LEN-1, %ecx    #
konwert:
movl $0, %edx             # w %edx jest „stara” reszta, trzeba ją wyzerować
div %ebx                  # (%edx) %dl – kolejna cyfra, %eax=iloraz
orb '0', %dl              # kod ASCII cyfry 0 (albo $ZERO jeśli .equ ZERO, 0x30)
movb %dl, BUFVY(%ecx)
dec %ecx
andl %eax, %eax           # iloraz=0 – koniec konwersji
jnz konwert
end_konw:                 # teraz wyprowadź liczbę (kody ASCII z bufora)
movl $BWY_LEN, %edx       # rozmiar bufora w bajtach ($BWY_LEN) do %edx
movl $BUFVY, %ecx         # adres startowy bufora ($BUFVY) do %ecx
movl $STDOUT, %ebx        # nr wejścia do %ebx (STDOUT=1)
movl $WRITE, %eax         # nr funkcji do %eax (WRITE=4)
int $SYSCALL32            # syscall
movl %ebp, %esp           # przywrócenie wskaźnika szczytu stosu (i oczyszczenie)
popl %ebp                 # odtworzenie „starego” wskaźnik powiązania
[„pop ...”]              # rejestry f. wywołującej ze stosu (gcc)
ret                      # zwrot sterowania do miejsca wywołania

```

Funkcje rekurencyjne

Ciało funkcji rekurencyjnej zawiera:

- przekazywanie parametrów na kolejny poziom zagnieżdżenia (przez stos)
- sprawdzenie warunku zakończenia rekurencji
- wywołanie funkcji lub ominięcie jeśli koniec rekurencji
- wykonanie kolejnych obliczeń
- odtworzenie stosu poprzedniego poziomu
- przekazanie wyniku do poziomu poprzedniego wywołania

```

...                # wywołanie funkcji
[pushl $PAR]        # parametry funkcji na stos (zwykle statyczne)
pushl $ARG_REK      # argument funkcji na stos
call frecursive     # wywołanie
addl $ST_SIZE, %esp # oczyszczenie stosu (%esp poniżej parametru wywołania)
...                # dalszy ciąg programu

.type frecursive @function # deklaracja funkcji
frecursive:
pushl %ebp          # "stary" wskaźnik powiązania dynamicznego na stos
movl %esp, %ebp     # nowy wskaźnik: szczyt stosu - tu się rozpoczyna nowy
movl 8(%ebp), [%eax] # bieżący argument funkcji (z wnętrza stosu) do %eax
                    # ((%ebp):,,stary ebp, 4(%ebp): adres powrotu, 8(%ebp):ARG)
                    # (w IA-32 słowo=4bajty; w IA-32e słowo=8bajtów)

[cmpl $LAST, %eax]  # warunek końca [w %eax jest bieżący argument funkcji]
[je end_fact]       # koniec sekwencji wywołań rekurencyjnych

[decl %eax]         # obliczenie kolejnego argumentu rekurencji
[pushl ARG]         # kolejny argument rekurencji na stos [ARG=%eax]

call frecursive

[movl 8(%ebp), %ebx] # obliczenie etapowe
[mull %ebx]          # kolejne obliczenie (ew. imull %ebx, %eax)

end_fact:           # powrót na poprzedni poziom wywołania
movl %ebp, %esp     # przywrócenie wskaźnika szczytu stosu (i oczyszczenie)
popl %ebp           # odtworzenie "starego" wskaźnik powiązania
ret                 # zwrot sterowania do miejsca wywołania

```

Funkcja rekurencyjna (obliczanie silni)

```
.type factorial @function # deklaracja funkcji
factorial:
pushl %ebp                # "stary" wskaźnik powiązania dynamicznego na stos
movl %esp, %ebp           # nowy wskaźnik: szczyt stosu – tu się rozpoczyna nowy
movl 8(%ebp), %eax        # parametr (z wnętrza stosu) do %eax
cmpl $1, %eax             # warunek końca (1!=1), 1 to jednocześnie wartość pocz
je end_factorial          # koniec sekwencji wywołań rekurencyjnych
decl %eax                 # obliczenie kolejnego argumentu rekurencji
pushl %eax                # kolejny argument rekurencji na stos
call factorial
ret_adr:
movl 8(%ebp), %ebx        # parametr rekurencji do %ebx, bo %eax zawiera wynik
mull %ebx                 # kolejne obliczenie (ew. imull %ebx, %eax)
end_factorial:
movl %ebp, %esp           # przywrócenie wskaźnika szczytu stosu (i oczyszczenie)
popl %ebp                 # odtworzenie "starego" wskaźnik powiązania
ret                       # zwrot sterowania do miejsca wywołania
```

– zagnieżdżanie (budowa stosu)

	stos ↓		komentarz ↓
pushl %eax	→	„old” esp	
call factorial	→	ARG esp:=esp-4	# argument (ARG=eax) na stos
nxt_adr: ...	→	nxt_adr esp:=esp-4	# adresu powrotu na stos
		„old”	#
		ARG [ebp ⁽¹⁾ +8]	# argument: 8-11 bajt od szczytu
factorial:		nxt_adr [ebp ⁽¹⁾ +4]	# wykonanie 1 kroku rekurencji
pushl %ebp	→	%ebp esp:=esp-4	#
movl %esp, %ebp		ebp ⁽¹⁾ :=esp	# nowy wskaźnik ramki stosu
			#
movl 8(%ebp), %eax		eax:=[ebp ⁽¹⁾ +8]	# argument ARG z ramki do eax
cmpl \$1, %eax			#
je end_factorial		„old”	# jeśli eax=1, koniec rekurencji
		ARG [ebp ⁽¹⁾ +8]	#
		nxt_adr [ebp ⁽¹⁾ +4]	#
decl %eax		%ebp	# dopóki eax≠1 zagnieżdżaj
pushl %eax	→	ARG⁽¹⁾ [ebp ⁽²⁾ +8]	# parametr rekurencji na stos
call factorial	→	ret_adr [ebp ⁽²⁾ +4]	# adres powrotu
pushl %ebp	→	%ebp⁽¹⁾ esp:=esp-4	#
movl %esp, %ebp		ebp ⁽²⁾ :=esp	# nowy wskaźnik ramki stosu
			#
movl 8(%ebp), %eax		eax:=[ebp ⁽²⁾ +8]	# argument ARG ⁽¹⁾ z ramki do eax
cmpl \$1, %eax		„old”	#
je end_factorial		ARG	#
		nxt_adr	#
		%ebp	#
		ARG⁽¹⁾ [ebp ⁽²⁾ +8]	#
		ret_adr [ebp ⁽²⁾ +4]	#
decl %eax		%ebp⁽¹⁾	# dopóki eax≠1 zagnieżdżaj
pushl %eax	→	ARG⁽²⁾ [ebp ⁽³⁾ +8]	# parametr rekurencji na stos
call factorial	→	ret_adr [ebp ⁽³⁾ +4]	# adres powrotu
pushl %ebp	→	%ebp⁽²⁾ esp:=esp-4	#
movl %esp, %ebp		ebp ⁽³⁾ :=esp	# nowy wskaźnik ramki stosu
movl 8(%ebp), %eax		eax:=[ebp ⁽³⁾ +8]	# argument ARG ⁽²⁾ z ramki do eax
... ..			# kolejne wywołania dopóki eax≠1

– powroty (zwalnianie stosu)

		↑ <i>stos</i>		stos odwrócony
pushl %ebp			esp:=esp-4	#
movl %esp, %ebp			ebp ⁽ⁿ⁾ :=esp	# nowy wskaźnik ramki stosu
movl 8(%ebp), %eax			(ebp=ebp ⁽ⁿ⁾)	# ARG ⁽ⁿ⁻¹⁾ =[ebp ⁽ⁿ⁾ +8]=1
cmpl \$1, %eax				# teraz już eax=1
je end_factorial				# omiń „call”, bo eax = 1
end_factorial:			(esp=ebp ⁽ⁿ⁾)	# sekwencja powrotów:
movl %ebp, %esp				#
popl %ebp	←	%ebp⁽ⁿ⁻¹⁾	esp:=esp+4	# poprzednia ramka (ebp:=ebp ⁽ⁿ⁻¹⁾)
ret	←	ret adr	esp:=esp+4	#
		ARG ⁽ⁿ⁻¹⁾	[ebp ⁽ⁿ⁾ +8]	# argument ARG ⁽ⁿ⁻¹⁾ =1
		%ebp ⁽ⁿ⁻²⁾		#
		ret adr		#
		ARG ⁽ⁿ⁻²⁾	[ebp ⁽ⁿ⁻¹⁾ +8]	# argument ARG ⁽ⁿ⁻²⁾ =2
		%ebp ⁽ⁿ⁻³⁾		#
		ret adr		#
		ARG ⁽ⁿ⁻³⁾	[ebp ⁽ⁿ⁻²⁾ +8]	# argument ARG ⁽ⁿ⁻³⁾ =3
		#
		„old”		#
ret_adr:				#
movl 8(%ebp), %ebx				# [ebp ⁽ⁿ⁻¹⁾ +8]=ARG ⁽ⁿ⁻²⁾ =2 →ebx
mull %ebx, %eax				#
end_factorial:				#
movl %ebp, %esp		ARG⁽ⁿ⁻¹⁾	esp=ebp ⁽ⁿ⁻¹⁾	#
popl %ebp	←	%ebp⁽ⁿ⁻²⁾	esp:=esp+4	# poprzednia ramka (ebp:=ebp ⁽ⁿ⁻²⁾)
ret	←	ret adr	esp:=esp+4	#
		ARG ⁽ⁿ⁻²⁾	[ebp ⁽ⁿ⁻¹⁾ +8]	#
		%ebp ⁽ⁿ⁻³⁾		#
		ret adr		#
		ARG ⁽ⁿ⁻³⁾		# [ebp ⁽ⁿ⁻²⁾ +8]=ARG ⁽ⁿ⁻³⁾ =3
		...		#
		„old”		#
...				#
ret_adr:				#
movl 8(%ebp), %ebx				# [ebp ⁽²⁾ +8]=ARG ⁽¹⁾ =n-1 →ebx
mull %ebx, %eax				# wymnóż przez poprzedni iloczyn
end_factorial:				#
movl %ebp, %esp		(ARG⁽²⁾)	esp=ebp ⁽²⁾	#
popl %ebp	←	%ebp⁽¹⁾	esp:=esp+4	# poprzednia ramka (ebp:=ebp ⁽¹⁾)
ret	←	ret adr	esp:=esp+4	#
		ARG ⁽¹⁾	[ebp ⁽²⁾ +8]	# ARG ⁽¹⁾ =[ebp ⁽²⁾ +8]=n-1
		%ebp		#
		ret adr		#
		ARG		# ARG=[ebp ⁽¹⁾ +8]=n →ebx
		„old”		#
ret_adr:				#
movl 8(%ebp), %ebx				# wymnóż przez poprzedni iloczyn
mull %ebx, %eax				#
end_factorial:				#
movl %ebp, %esp		(ARG⁽¹⁾)		#
popl %ebp	←	%ebp		# przywrócenie ramki wywołania
ret	←	ret adr	esp:=esp+4	# zagnieżdżeń nie trzeba liczyć!
		ARG	[ebp ⁽¹⁾ +8]	# ARG=[ebp+8]=n
		„old”		#
next_adr: ...				#
				# przywrócony stos początkowy
addl \$4, %esp		ARG	esp:=esp+4	# argument funkcji na stosie
		„old”	esp	#

Funkcja wariacje (ang. *variations*) – iteracyjna i rekurencyjna

```

# argumenty: zmienne nn i kk;  $V(n,k)=V(n,k)=n!/(n-k)!$ 
#  $V(n,k+1)=(n-k)V(n,k)$   $V(n,n)=P(n)=n!$ 
movl nn, %ebx      # argument nn do %ebx
movl kk, %ecx      # argument kk do %ecx, także licznik iteracji
movl $1, %eax      #  $V(n,0)=1$ ,  $V(n,1)=n$ 
cmpl $0, %ecx      # sprawdzenie, czy k=0
jz end

    variter:      # iteracyjne obliczenie wariacji
    mull %ebx      # wymnóż przez kolejny czynnik
    decl %ebx      # kolejny mnożnik k-1
    loop variter   # koniec obliczeń gdy ecx=[kk]

# rekurencyjne obliczanie wariacji
pushl kk           # albo „pushl %ecx - argument kk na stos
pushl nn           # albo „pushl %ebx - argument nn na stos
call var           # wywołanie (w rejestrze %eax jest 1)
addl $8, %esp      # zwolnienie stosu, %esp poniżej parametru wywołania
end:
movl %eax, wynik   # wynik do zmiennej wynik

.type var @function
var:
[„push ...”]      # rejestry f. wywołującej na stos (gcc)
pushl %ebp        # „stary” wskaźnik powiązania dynamicznego na stos
movl %esp, %ebx   # nowy wskaźnik: szczyt stosu) do %ebx
movl 12(%ebp), %ecx # parametr k* (z wnętrza stosu) do %eax
movl 8(%ebp), %eax # parametr n* (z wnętrza to jednocześnie wartość #)
# *) movl 8(%ebp), %eax
# warunek końca #)
# *) cmpl $0, %ecx
je end_var        # koniec sekwencji wywołań rekurencyjnych
decl %ecx         # obliczenie kolejnego argumentu rekurencji
decl %eax         # obliczenie kolejnego argumentu rekurencji
# *) decl %ebx
pushl %ecx        # kolejny argument rekurencji (k*) na stos
pushl %eax        # kolejny argument rekurencji (n*) na stos
# *) pushl %ebx

call var          # parametr rekurencji do %ebx, bo %eax zawiera wynik
movl 8(%ebp), %ebx # kolejne obliczenie (ew. imull %ebx, %eax)
mull %ebx         # pierwsze wejście: %eax=n-k+1
end_var:         # przywrócenie wskaźnika szczytu stosu (i oczyszczenie)
movl %ebp, %esp   # odtworzenie „starego” wskaźnik powiązania
popl %ebp         # rejestry f. wywołującej ze stosu (gcc)
[„pop ...”]
ret              # zwrot sterowania do miejsca wywołania

```

Funkcja kombinacje (ang. *variations*) – iteracyjna i rekurencyjna

```
/* C(n,k)=n!/(k!(n-k)!)= C(n-1,k-1)*n/k*/
/* C(n,k)= {...([n·(n-1)/2]·(n-2)/3) ...·(n-k+1)} / k , albo w odwrotnej kolejności*/
/* C(n,k)= {...([(n-k+1)(n-k+2)/2]·(n-k+3)/3) ...· n} / k */
/* max C(n,k)=C(n,n/2). Także  $2^k < C(2k+1,k) < k^k$ , co pozwala oszacować zakres */
```

```
...                # argumenty: zmienne nn i kk; v(nn,kk)=v(n,k)=n!/(n-k)!
movl $0, %eax      # C(n<k,k)=df 0
movl kk, %ecx
movl nn, %ebx
cmpl %ebx, %ecx    #
j1 end_combine    # jeśli n<k, to (def) C(n<k,k)=0
subl %ecx, %ebx    # %ebx:= %ebx - %ecx (n:n-k)
cmpl %ebx, %ecx    # n-k>k?
jg hop:           # jeśli n<k, to (def) C(n<k,k)=0
movl %ebx, kk      # n-k zamiast k
hop:
movl $1, %eax      # C(n,0)=1 (pierwszy iloczyn)
cmpl $0, kk        #
je end_combine     # jeśli k=0 (lub k=n), to C(n,0)=1
movl nn, %ebx      # przywrócenie n (nie zmienia flag!)
movl $1, %ecx      # dzielnik (1,2,...,k)

    combine:       # iteracyjne obliczenie kombinacji
    mull %ebx      # wymnóż przez kolejny czynnik
    divl %ecx      # wynik w %edx:%eax
    decl %ebx      # kolejny mnożnik (nn--)
    incl %ecx      # kolejny dzielnik (licznik++ until kk)
    cmpl %ecx, kk  # powtarzaj, dopóki licznik (%ecx) nie przekroczy k
    j1 combine

                # rekurencyjne obliczanie kombinacji
pushl kk        # argument k na stos
pushl nn        # argument n na stos
call combine    # wywołanie (combine = newton lub pascal)
addl $8, %esp   # zwolnienie stosu, %esp poniżej parametru wywołania
end_combine:
movl %eax, wynik # wynik

.type pascal @function
pascal:
pushl %ebp      # "stary" wskaźnik powiązania dynamicznego na stos
movl %esp, %ebp # nowy wskaźnik: szczyt stosu do %ebx
...
call pascal
movl %ebp, %esp # przywrócenie wskaźnika szczytu stosu (i oczyszczenie)
popl %ebp       # odtworzenie "starego" wskaźnika powiązania
ret            # zwrot sterowania do miejsca wywołania
```

```

.type newton @function
newton:
[„push ...”]
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %ecx
movl 8(%ebp), %eax
cmpl $1, %ecx
je end_comb
decl %ecx
decl %eax
pushl %ecx
pushl %eax
call newton
movl 8(%ebp), %ebx
movl 12(%ebp), %ecx
mull %ebx
div %ecx
end_comb:
movl %ebp, %esp
popl %ebp
[„pop ...”]
ret

```

rekurencyjne obliczanie kombinacji
 # $C(n,k)=n!/(k!(n-k)!)=C(n-1,k-1)*n/k$
 # rejestry f. wywołującej na stos (gcc)
 # “stary” wskaźnik powiązania dynamicznego na stos
 # nowy wskaźnik: szczyt stosu do %ebx
 # parametr k^* (z wnętrza stosu) do %ecx
 # parametr n^* (z wnętrza stosu) do %ebx
 # warunek końca
 # koniec sekwencji wywołań rekurencyjnych
 # obliczenie kolejnego argumentu rekurencji
 # obliczenie kolejnego argumentu rekurencji
 # kolejny argument rekurencji (k^*) na stos
 # kolejny argument rekurencji (n^*) na stos
 # parametr rekurencji do %ebx, bo %eax zawiera wynik
 # parametr rekurencji do %ecx
 # kolejne obliczenie (ew. `imull %ebx, %eax`)
 # pierwsze wejście: $\%eax=n-k+1$, $\%ecx=1$
 # przywrócenie wskaźnika szczytu stosu (i oczyszczenie)
 # odtworzenie “starego” wskaźnik powiązania
 # rejestry f. wywołującej ze stosu (gcc)
 # zwrot sterowania do miejsca wywołania

Łączenie as z C**Użycie funkcji bibliotecznych C z poziomu assemblera – środowisko 32b (IA-32),**

Przed użyciem sprawdzić np. za pomocą komendy `man`, działanie funkcji systemowych jeśli występują w różnych wersjach (np. `exit` oraz `_exit`).

Dostęp do funkcji uzyskamy kompilując program za pomocą `gcc` a nie `as`, co wymaga użycia etykiety startowej `main` zamiast `_start` – kompilator *GNU gcc* inicjuje procedury ładujące biblioteki systemu. Alternatywą jest (z etykietą `_start`) wymuszone dołączenie bibliotek przez linker:

```
ld prog.o -o prog -lc -dynamic-linker /lib/ld-linux.so.2
```

int puts(const char *s)

```
# wysyła na STDOUT znaki od adresu złożonego na stosie i dodaje znak NL (newline)#
                                # kompilator gcc dołącza potrzebną bibliotekę
.data                            # dyrektywa .align 32 zbędna, dane bajtowe/ascii
napis: .ascii "Napis\0" # napis musi być zakończony terminatorem '\0' (koniec)
.text
.globl main                    # dyrektywa eksportowania nazwy do linkera
main:                          # etykieta startowa jak dla C, kompilacja gcc
push $napis                    # adres łańcucha na stos (parametr funkcji puts)
call puts                      # wywołanie funkcji języka C
call _exit                    # lub jak w as przez syscall albo int syscall32
```

printf(format-a, arg1, arg2,...)

```
.data                            # łańcuch formatujący jest pierwszym argumentem printf
format_string:                  # – określa liczbę i rodzaj parametrów – tu są 3 param.
.ascii "Hello! %s is a %s who loves the number %d\n\0"
text1: .ascii "Tekst1\0"        # pierwszy parameter %s (łańcuch znaków zakończony \0)
text2: .ascii "Tekst2\0"        # drugi parameter %s (łańcuch znaków zakończony \0)
number: .long 3                  # trzeci parameter %d (liczba dziesiętna)
.text
.globl _start
_start:                          # konieczny linkera w opcji dynamicznej lub main
                                # parametry przez stos w odwróconej kolejności
pushl number                    # trzeci – liczba dziesiętna (%d)
pushl $text2                    # drugi – łańcuch (%s)
pushl $text1                    # pierwszy – łańcuch (%s)
pushl $format_string            # łańcuch formatujący
call printf
#pushl $0                        .. zwracany kod błędu #
call _exit                      # funkcja zakończenia programu (por. z call exit)
```

Funkcja zlicza ilość znaków % w łańcuchu formatującym, co pozwala określić liczbę parametrów na stosie. Dodatkowo musi wykonywać konwersję liczb do ciągów znaków odpowiadających zapisowi pozycjonemu. Funkcja ma też wiele innych opcji formatowania (%o, %x, %e, %f, %g).

Wstawki assemblerowe w języku C (inline assembly) [6]

W języku C dostępna jest funkcja (konstrukcja składniowa), umożliwiająca wstawkę assemblerową. Ma ona postać łańcucha znaków przekazywanego do assemblera po wstępnym przetworzeniu przez kompilator C:

```
__asm__ (
    "instrukcja assemblera\n"
    "następna instrukcja\n"
    ...
    "ostatnia instrukcja\n"
    : zmienne wyjściowe (opcjonalne)
    : wartości wejściowe (opcjonalne)
    : niszczone rejestry (opcjonalne)
);
```

Instrukcje można też zapisać jako jeden łańcuch używając symbolu kontynuacji ‘\’ w kolejnej linii. Symbole poprzedzone znakiem % są traktowane jako argumenty instrukcji zapisanych w kolejnych łańcuchach, albo nazwy rejestrów (%eax, %ebx,...) albo numery porządkowe zmiennych (0, 1, 2, ...):

```
/* zamiana wartości zmiennych x i y (typu int) przy użyciu rejestrów */
__asm__ (
    "movl %2, %%eax\n"           // "x we" do eax
    "movl %3, %0\n"             // "y we" do "x wy"
    "movl %%eax, %1\n"          // eax do "y wy"
    : "=r"(x), "=r"(y)          // zmienne wyjściowe (nr 0 i 1)
    : "r"(x), "r"(y)            // wartości wejściowe (nr 2 i 3)
    : "%eax"                    // rejestr niszczone (eax)
);
```

Elementy list oddzielają przecinki. Każdy element ma określony sposób przekazywania.

- "r" - za pomocą dowolnego rejestru
- "m" - poprzez adres w pamięci
- "a", "b", "c", "d", "S", "D" - w rejestrach eax, ebx, ecx, edx, esi lub edi

Znak = bezpośrednio przed symbolem (r,m,...) oznacza, że lokalizacja dotyczy zmiennej wyjściowej. Znaki =& przed symbolem (r,m,...) oznaczają użycie innego rejestru dla zmiennej na wyjściu.

W instrukcji assemblera można bezpośrednio wskazać statyczną zmienną globalną, pisząc jej nazwę poprzedzoną znakiem \$. Na przykład: `__asm__("movl $xxx, %%eax"::"%eax");`

Aby zablokować optymalizację kodu assemblerowego przydatne jest pisanie `__volatile__` po `__asm__`. Bez tego `gcc` może uznać, że nasz assembler tutaj nic istotnego nie wnosi (czytaj: nie zmienia wartości zmiennej, ani nie wywołuje funkcji) i usunąć go z końcowego kodu.

Kompilacja z opcjami `-s` oraz `-fverbose-asm` pozwala wygenerować kod z kompilatora wyższego poziomu do assemblera. Kod generowany przez kompilator `gcc` będzie wtedy wyraźnie oddzielony komentarzami od kodu wstawki `asm`.

Przekazywanie nazw (zmiennych, funkcji) i danych pomiędzy modułami C i as

- korzystanie ze zmiennych zdefiniowanych w assemblerze z poziomu C i odwrotnie
- wywołanie własnej funkcji napisanej w C z poziomu assemblera
- wywołanie własnej funkcji argumentów zmiennoprzecinkowych
- korzystanie z funkcji napisanych w assemblerze z poziomu języka C

Funkcja `main()` w pliku z kodem w C zawiera wywoływanie funkcji assemblerowej, która wywołuje kolejno funkcje napisane w C:

- funkcja `suma`, dodająca wartość lokalną z as oraz zmienną globalną zdefiniowaną w C; wynik wypisuje funkcja biblioteczna `printf`;
- funkcja `iloraz` argumentów zmiennoprzecinkowych (wskazanych za pomocą dyrektywy `.float`) przesyłanych przez stos do FPU – wynik jest na szczycie stosu `st(0)` koprocessora (FPU)
- wypisanie wyniku z FPU (`st(0)`) za pomocą `printf` – należy go przesłać go ze stosu FPU (`st(.)`) na stos programowy po uprzedniej konwersji przez stos FPU na format `double` (standard w C).

```
#include <stdio.h>
extern void funkcja_asm(); / nazwa funkcji zewnętrznej (z innego pliku)
extern int globalna_z_asm; / nazwa zmiennej zewnętrznej (z innego pliku)
int globalna_z_C = 777;
void moja_fun(char *arg) / deklaracja własnej funkcji w C
{
printf("Wywołanie z C: %s", arg);
}
int suma(int a, int b) / deklaracja własnej funkcji w C
{
return a+b;
}
float iloraz(float a, float b) / deklaracja własnej funkcji w C
{
if(b==0.0) return 0.0;
return a/b;
}
int main() / funkcja główna w C
{
funkcja_asm(); / wywołanie funkcji zewnętrznej
printf("Zmienna z assemblera: %d\n", globalna_z_asm);
return 0;
}
.globl globalna_z_asm # deklaracja nazwy zmiennej jako globalnej
[.extern globalna_z_C] # zbędne, symbol niezdefiniowany jest uznany za extern
.data
napis: .ascii "Argument z assemblera, wynik funkcji z C = %d\n\0"
napis2: .ascii "Argument z assemblera, wynik float z C = %f\n\0"
liczba1: .float 3
liczba2: .float 4
.type globalna_z_asm, @object # zadeklaruj zmienna z C jako obiekt
```

```

.size globalna_z_asm, 4
globalna_z_asm: .long 444
.text
.globl funkcja_asm          # deklaracja nazwy funkcji jako globalnej
.type funkcja_asm, @function # definicja funkcji
funkcja_asm:
push %ebp
mov %esp, %ebp              # utworz ramke stosu
push $4                     # na stos liczba 4
push globalna_z_C           # na stos wartosc zmiennej globalne zdefiniowanej w C
call suma                   # wywolaj funkcje z C
add $8, %esp                # przesun stos
push %eax                   # wynik sumowania na stos
push $napis                 # adres napisu jako ciag formatujacy dla printf
call printf
add $4, %esp
mov liczba1, %eax           # zaladuj na stos zmienne float
mov %eax, 4(%esp)
mov liczba2, %eax
mov %eax, (%esp)
call iloraz                 # wywolaj funkcje z C operujaca na zmiennych float
fstps -8(%ebp)              # zapisz wynik z pamieci (ze stosu FPU st(0) )
flds -8(%ebp)               # zaladuj go ponownie do stosu FPU jako double
fstpl (%esp)                # ze szczytu stosu FPU na szczyt stosu programowego
push $napis2                # adres napisu jako ciag formatujacy dla printf
call printf                 # wypisz informacje
add $4, %esp
leave                       # usun ramke stosu (mov %ebp, %esp / pop %ebp)
ret

```

Użycie profilera do optymalizacji kodu – program w języku C działający na dużych zbiorach danych.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 5000000
void fun(int *dane)
{
    int i;
    for(i=0; i<SIZE; ++i)
        dane[i] = rand(); //wypelnij tablice losowymi danymi
    for(i=0; i<SIZE; ++i)
    {
        //dane[i] = 5*dane[i] + dane[i]/3 + dane[i]*3; // waskie gardlo zastapione przez:
        __asm(                                     / wstawka asm zamiast poprzedniej linii/
            "movl (%edi,%ecx,4), %%eax \n\

```



```

mov %%eax, %%ebx \n\
mov %%eax, %%edx \n\
shl $3, %%eax \n\
shr $1, %%ebx \n\
sub %%edx, %%ebx \n\
add %%ebx, %%eax \n\
movl %%eax, (%%edi,%%ecx,4)"
: // no output
:"c"(i), "D"(dane)          / modyfikatory _asm – przypisanie do rejestrów
);                          / koniec wstawki
}
}
int main()
{
int *dane = (int*) malloc(SIZE * sizeof(int)); //rezerwuj pamiec
srand(time(NULL)); //zainicjuj generator pseudolosowy
fun(dane); //wywołaj badana funkcje
free(dane); //zwolnij pamiec
return 0;
}

```

Wywołanie funkcji `__asm()` pozwala na wstawienie kodu assemblerowego w zadane miejsce w programie w C. Ma ono specyficzną budowę. Wewnątrz wstawki można dostać się do zmiennej iteracyjnej pętli „i” za pomocą rejestru `ecx` oraz adresu początku tablicy przez rejestr `edi`. Aby to było możliwe należy dodać modyfikatory do parametru wywołania instrukcji `__asm()`, a mianowicie `:"c"(i), "D"(dane)`. Informuje to kompilator, że do rejestru `ecx` ma wstawić wartość zmiennej „i” a do rejestru `edi` – adres tablicy (wart. zmiennej `dane`). Wyniki pomiaru czasu wykonania programu niezoptymalizowanego (bez wstawki) uzyskujemy za pomocą aplikacji `gprof` (do kompilacji należy użyć modyfikatora `-pg`). Przykładowy program skompilowano z domyślnymi ustawieniami optymalizacji automatycznej kompilatora.

(przykładowy komunikat)

Each sample counts as 0.01 seconds.

% cumulative self self total

time	seconds	seconds	calls	ms/call	ms/call	name	
100.00	0.22	0.22	1	220.00	220.00	fun	# przed optymalizacją
100.00	0.18	0.18	1	180.00	180.00	fun	# po optymalizacji (asm)

Do pomiarów można też wykorzystać instrukcję `rdtsc`, która zapisuje stan licznika cykli procesora (liczonych od restartu procesora – zwykle włączenia komputera) do pary rejestrów `edx:eax`.

Aby uzyskać dostęp do stanu licznika TSC (*Time Stamp Counter*) użyto makra preprocesora:

```
#define rdtsc() ({int64_t x; asm volatile("rdtsc" : "=A" (x)); x; })
```

Makro to jest wywoływane w kodzie jak funkcja i zwraca wartość typu `int64_t` – jest to rozszerzona do 64-b zmienna `int`. Aby jej użyć należy dołączyć plik nagłówkowy: `#include <inttypes.h>`.

Czas działania funkcji jest różnicą stanów licznika przed wywołaniem, podzieloną przez częstotliwość procesora (jeśli w kHz – wtedy czas jest w ms).

Odwzorowanie programu, danych i stosu w pamięci [1]

Każda sekcja jest ładowana do osobnego obszaru w pamięci (początek bloku: adres 0xBFFF FFFF). Kody instrukcji (.section .text) są ładowane począwszy od adresu 0x08048000, kody danych (.section .data) bezpośrednio po nich, a następnie bufor dynamiczny (.section .bss). Ostatni bajt nie może być wyżej niż w lokacji 0xBFFFFFFF. W tym miejscu Linux zaczyna tworzyć swój stos, który jest rozbudowywany w kierunku adresów rosnących, aż do swej kolejnej sekcji.

Na dnie stosu (ang. *the bottom of the stack*), którego adres jest najwyższy (ang. *the top address of memory*) jest początkowo umieszczone słowo zerowe (wszystkie bity są zerami). Po nim następuje zakończona zerem (ang. *the null-terminated*) nazwa programu w kodzie ASCII, a po niej zmienne środowiskowe programu (ang. *program's environment variables*). Dalej są się argumenty wywołania (ang. *command-line arguments*), czyli parametry (wartości) wpisane do linii polecenia podczas wywołania programu. Na przykład, uruchamiając `as`, podajemy jako argumenty: `as, sourcefile.s, -o i objectfile.o`. Po nich następują używane argumenty, umieszczone na początku bloku stosu wskazanego przez wskaźnik stosu (ang. *stack pointer*) `%esp`. Kolejne operacje na stosie zmieniają ten wskaźnik – złożenie danych na stos powoduje zmniejszenie `%esp`.

Obszar danych programu rozpoczyna się na dnie pamięci (ang. *the bottom of memory*) i jest budowany wzwyż (kolejne lokacje mają coraz wyższe adresy). Stos rozpoczyna się na szczycie pamięci (ang. *the top of memory*) i jest rozbudowywany w dół (w kierunku adresów malejących) po każdym wykonaniu instrukcji `push`. Obszar pomiędzy stosem a obszarem danych jest z zasady niedostępny z poziomu programu. Próba dostępu (użycie adresu z tego obszaru) kończy się sygnalizacją błędu, zwykle jako "segmentation fault". To samo zdarzy się podczas próby zaadresowania obszaru poniżej adresu 0x08048000. Ten najniższy dostępny adres jest nazywany *system break* (albo [*current*] *break*).

Zmienne środowiskowe (environment variables)		0x BFFF FFFF
...		
Arg 2		
Arg 1		
Nazwa programu (program name)		
Liczba argumentów (# of arguments)	%esp	
Pamięć dostępna dla programu (Unmapped memory)		
Kod i dane programu (Program Code and Data)	break	0x 0804 8000

Literatura

- [1] Jonathan Bartlett, Programming from the Ground Up, 2003.
- [2] J. Biernat, Architektura komputerów, Oficyna Wyd. PWr, Wrocław, 2005 (wyd. IV)
- [3] L. Madeja, Ćwiczenia z systemu Linux, MIKOM, 1999 (rozdz. 4).
- [4] D. Elsner, J. Fenlason & friends, Using as. The GNU assembler, Free Software Foundation, 1994
- [5] https://refspecs.linuxfoundation.org/LSB_3.2.0/LSB-Core-IA32/LSB-Core-IA32.pdf
- [6] <http://students.mimuw.edu.pl/SO/Projekt03-04/temat2-q6/book1.html>
- [7] J.Spolsky, "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets" (<http://www.joelonsoftware.com/articles/Unicode.html>.)

Jednostka zmiennoprzecinkowa x87 (IA-32)

Ma architekturę stosową. Stos tworzy 8 rejestrów ST(0) ... ST(7), bieżący numer rejestru szczytu zapisany w polu TOP rejestru stanu FPSR (ang. Floating Point Status Register). Operacja „load” („push”) powoduje dekrementację wskaźnika TOP, „store” („pop”) powoduje inkrementację TOP. Jeśli TOP=0, operacja „store” („pop”) powoduje cykliczne „zawinięcie” (ang. *wraparound*) i ustawiane jest TOP=7, a jeśli może to spowodować nadpisanie nieprzechowanej danej, generowany jest wyjątek „przepełnienie stosu” (*stack overflow* - # IS). Są też dostępne instrukcje które nie zmieniają wskaźnika TOP (zapis/odczyt rejestru ST()).

Argumentem instrukcji może być każdy rejestr stosu, adresowany względem szczytu stosu, wynik działania, jeśli nie wskazano inaczej, jest umieszczany na szczycie stosu (ST(0)).

Rejestr stanu FPSR (ang. Floating Point Status Register). Zawiera informacje o stanie FPU po wykonaniu ostatniej instrukcji. Tworzą ją kody wyjątków oraz kody warunkowe.

Rejestr sterujący FPCR (ang. Floating Point Control Register). Służy do określenia trybu pracy FPU: sposób reakcji na wyjątki, schemat zaokrąglania wyników.

Rejestr etykiet stosu (tag word) – 8x2b (00-ważny, 01-zero, 10-specjal: NaN, ∞ , denormal, 11-pusty) – **emms** (*empty mmx state*) po serii instrukcji MMX ustawia **TAG** na „pusty” (11)

Rejestry pomocnicze:

adres ostatnio wykonanej instrukcji (ang. last instruction pointer reg.) – (seg:offset = 16+32 b)

adres ostatnio użytego operandu (ang. last data pointer reg.) – (seg:offset = 16+32b)

kod ostatniej instrukcji (ang. opcode reg.) – 11b (pełny kod 16b: **11011**xx...xxx, – **11011** - „ESC”)

Dostęp do tych rejestrów realizują instrukcje FSTENV/FNSTENV, FLDENV, FRSTOR, FXSAVE, FXRSTOR. Instrukcje FINIT/FNINIT, FSAVE/FNSAVE zerują te rejestry.

Rejestr sterujący

b12 – sterowanie nieskończonościami (X – *Infinity Control*)

b11-10 – tryb zaokrąglania (RC – *Rounding Control*) – (00 – (dom.) – r. to nearest (do najbliższej parzystej), 01 – r. down – (w dół), 10 – r. up – (w górę), 11 – r. toward zero – (do najmniejszej bezwzględnie)

b9-8 – sterowanie dokładnością (PC – *Precision Control*) – (00-23b, 10-53b, 11-64+b (domyślne)0
maski wyjątków (bitów rejestru stanu):

b5 – dokładność (PM) – dotyczy tylko instrukcji FADD(P), FIADD, FSUB(P), FISUB, FSUBR(P), FISUBR, FMUL(P), FIMUL, FDIV(P), FIDIV, FDIVR(P), FIDIVR, FSQRT.

b4 – niedomiar (UM)

b3 – nadmiar (OM)

b2 – dzielenie przez 0 (ZM)

b1 – operand zdenormalizowany (DM) – (zamaskowanie bitu DE powoduje automatyczną konwersję operandu do formatu wewnętrznego (15b wykł.) jako znormalizowanego)

b0 – niedozwolone działanie (IM)

f1dcw – ładowanie zmiennych sterowania z pamięci lub rejestru (ax)

fstcw/fnstcw – przechowanie zmiennych sterowania

Po **finit/fninit** lub **fsave/fnsave** słowo sterujące ma kod 0x37F, co oznacza maskowanie wszystkich wyjątków, zaokrąglanie do najbliższej oraz dokładność 64 b

Rejestr stanu (FPU Status Word) – 16-bitowy: - odczyt: `fstsw`Wyjątki zmiennoprzecinkowe – maskowane przez bity rejestru sterującego (*FPU Control Word*)

- 0 – IE, *Invalid Operation* – niedozwolone działanie
- 1 – DE, *Denormalized Operand* – operand zdenormalizowany
- 2 – ZE, *Zero Divide* – dzielenie przez 0
- 3 – OE, *Overflow* – nadmiar (przepełnienie) wykładnika
- 4 – UE, *Underflow* – niedomiar (wynik zdenormalizowany – zbyt mały)
- 5 – PE, *Precision* – utrata dokładności

- 6 – SF, *Stack Fault* – błąd stosu (*register wraparound*) (0 – OK, 1 – błąd)
- 7 – ES, *Exception Status* – sygnalizacja błędu (0 – OK, 1 – błąd, jeśli niezamaskowany w FPCR)

Bit obliczany po zakończeniu każdej instrukcji i ustawiany na 1 jeśli wystąpi jakikolwiek niezamaskowany wyjątek. Jeśli był ustawiony przez wcześniejszą instrukcję, procesor wywołuje procedurę #MF (multiple fault) kiedy jest wykonana następna instrukcja inna niż sterująca lub MMX. Bitów ES oraz SF nie można zamaskować.
- 8, 9, 10, 14 – *Condition Code* – kody warunkowe C0, C1, C2, C3, znaczenie zależne od działania
- 11-13 – *TOP of Stack* – wskaźnik szczytu stosu
- 15 – B, *Busy* – FPU zajęte (używany tylko przez koprocesor 8087; B=ES)

Bity/flagi wyjątków są "lepkie" tzn. pozostają ustawione dopóki nie zostaną bezpośrednio (*explicitly*) wyzerowane za pomocą instrukcji **fclex** lub **fnclx**, mogą być reinicjowane przez **finit** / **fninit** lub nadpisane poprzez użycie **frstor** lub **fldenv** (odtworzenie środowiska)

Priorytety Wyjątków (moment obsługi)

- 1 – (IE) niedozwolone działanie i błąd stosu (SF) z powodu niedomiaru (przed wykonaniem)
- 2 – (IE) niedozwolone działanie i błąd stosu (SF) z powodu nadmiaru (przed wykonaniem)
- 3 – (IE) niedozwolone działanie wskutek próby użycia niepoprawnych danych (przed wykonaniem)
- 4 – (IE) niedozwolone działanie wskutek próby użycia SNaN (przed wykonaniem)
- 5 – Działanie używające QNaN (*Operations involving QNaN operands do not, in themselves, cause exceptions but they are handled with this priority relative to the handling of exceptions*)
- 6 – (IE) inne niedozwolone działanie lub próba dzielenia przez zero (ZE) (przed wykonaniem)
- 7 – (DE) działanie powodujące denormalizację (przed wykonaniem)
- 8 – (OE) nadmiar lub (UE) niedomiar (po wykonaniu)
- 9 – (PE) utrata dokładności (po wykonaniu)

```

# Instrukcje odczytujące rejestr stanu...
FSTSW, FNSTSW      # argument: zmienna lub %ax
FSTENW, FNSTENW    # argument: zmienna środowiskowa
FSAVE, FNSAVE      # argument: zmienna
FXSAVE             # argument: zmienna

Przykładowy program:
# Iloczyn skalarny = (5.6 x 2.4) + (3.8 x 10.3)
FLD arg1           # arg1=5.6; TOP--, ST(0)=arg1
FMUL arg2          # arg2=2.4; ST(0)=ST(0)*arg2
FLD arg3           # arg3=3.8; TOP--, ST(0)=arg3, ST(1)="old ST(0)"
FMUL arg4          # arg4=10.3; ST(0)=ST(0)*arg4
FADD ST(1)         # ST(0)=ST(0)+ST(1)
```

Ustawianie kodów warunkowych C0, C1, C2, C3

Wskazują wynik porównania zmiennoprzecinkowego i niektórych działań arytmetycznych. Instrukcje FINIT/FNINIT, FSAVE/FNSAVE zerują bity C#, FRSTORE/FLDENV kopiują bity C# ze zmiennej (pamięci) Bit C1 wskazuje rodzaj wyjątku #IS – błąd stosu (IE=1, SF=1) (C1=0 – próba odczytu z pustego stosu – ang. *stack overflow or underflow*, C1=1 – przepełnienie) – uwaga: wystąpienie wyjątku IE nie zeruje SF! W działaniach arytmetycznych w razie ustawienia PE (błąd dokładności) C1=1 wskazuje, że ostatnio wykonane zaokrąglenie było w górę. FXAM kopiuje do C1 bit znaku argumentu. C2=1 wskazuje, że w wyniku działania FPREM lub FPREM1 nastąpiła niekompletna redukcja (reszta częściowa jest niezerowa), jeśli C2=0, to (C0, C3, C1) zawierają 3 najniższe bity ilorazu (Q2, Q1, Q0) FPTAN, FSIN, FCOS, FSINCOS ustawiają C2=1 jeśli argument jest ponad ± 263 poza zakresem. W innych przypadkach stan bitów C0 ... C3 jest nieokreślony.

Rozgałęzienia

„nowy mechanizm” (P6 +)

FCOM, FCOMP, FCOMPP, FTST, FUCOMPP, FICOM, FICOMP ustawiają CF=C0, ZF=C3, PF=C2 bezpośrednio, instrukcja FCMOVcc wykonuje kopiowanie zmiennoprzecinkowe stosownie do CF, ZF, PF.

(„stary mechanizm” (8087 ... P5)

FCOM, FCOMP, FCOMPP, FTST, FUCOMPP, FICOM, and FICOMP ustawiają flagi C0 ...C3, instrukcja FSTSW AX kopiuje bity stanu do ax, instrukcja SAHF kopiuje kody C0 ...C3 do EFLAGS tak, że CF=C0, ZF=C3, PF=C2)

Porównania zmiennoprzecinkowe

fcom, fcomp, fcompp, - ustawiają warunki w FPSR, ale można je przekopiować do EFLAGS

wykonując **fstsw ax** i potem **sahf** (CF=C0, PF=C2, ZF=C3)

fucom, fucomp/fucompp - porównaj nieuporządkowane (QNaN) i ustaw flagi FPU (w FPSR)

ficom, fcomp - porównaj i ustaw flagi FPU (FPSR)

fcomi, fcomip - porównaj i ustaw flagi IU (EFLAGS: ZF, CF, PF)

fucomi, fucomip - porównaj nieuporządkowane (QNaN) i ustaw flagi IU (EFLAGS: PF)

fcmovcc - warunkowe kopiowanie (po **f(u)comi**) zależne od EFLAGS, warunki: **b/nb** (CF), **be/nbe** (CF,ZF), **e/ne** (ZF), **u/nu** (PF=1 – unordered, jeden z operandów jest QNaN))

ftst - porównaj z 0,0

fxam - sprawdź rodzaj argumentu w ST(0) (0, denormalizowany, normalizowany, NaN, nieskończoność, nieokreślony format) i ustaw bity FPCR

Transfery danych

fld - operand zmiennoprzecinkowy z pamięci na stos /

fst/fstp - operand zmiennoprzecinkowy ze stosu do pamięci /... i „pop”

fild - operand całkowity (integer) z pamięci po konwersji na 64b FP na stos

fist/fistp - operand całkowity (integer) ze stosu do pamięci /... i „pop”

fbld - operand z pamięci traktowany jako upakowany BCD (18 cyfr 4b) na stos

fbstp - operand: upakowany BCD (18 cyfr 4b) ze stosu i „pop”

fxch - wymiana argumentu z ST(i) ze szczytem stosu (ST(0))

Tworzenie stałych w ST(0)

FLDZ/FLD1 – wpis 0.0 / 1.0

FLDPI – wpis π (3,14159...)

FLDL2T / FLDL2E / FLDLG2 / FLDLN2 – wpis logarytmu $\log_2 10$ / $\log_2 e$ / $\log_{10} 2$ / $\ln 2$ /

Arytmetyka

fadd/faddp -

fiadd -

fsub/fsubp -

fisub -

fsubr/fsubrp -

fisubr -

fmul/fmulp -

fimul -

fdiv/fdivp -

fidiv -

fdivr/fdivrp -

fidivrr -

frem - reszta częściowa

frem1 - reszta częściowa IEEE

fabs -

fchs - change sign

fsqrt -

frndint - zaokrąglenie o całkowitej

fxtract - rozdziel kody wykładnik i mnożnik i zapisz je w formacie FP

Funkcje trygonometryczne – argumenty w radianach na szczycie stosu

fsin -

fcos -

fsincos -

fptan - tg

fpatan - arctg – argumenty w układzie kartezjańskim (arctg ST(1)/ST(0))

Typy danych

– zmiennoprzecinkowe wg standardu IEEE754 – format wewnętrzny rozszerzony 80b (15b wykładnika, bez bitu ukrytego)

– całkowite w kodzie znak-moduł (16/32/64b)

– BCD znak + 18 cyfr (bity 72...78 nieokreślone)

liczby zdenormalizowane są konwertowane na znormalizowane w formacie wewnętrznym

W formacie rozszerzonym mogą wystąpić kodowania liczb, które nie są „wspierane przez standard zmiennoprzecinkowy:

– bardzo duże liczby o 15-b kodzie wykładnika 11...11 (tzw. *pseudo-NaN* lub *pseudo-nieskończoności*), w razie ich użycia generowany jest wyjątek niedozwolonego działania (*invalid operation*)

– liczby nieznormalizowane (*unnormals*) z bitem wiodącym 0 oraz bardzo małe o 15-bitowym kodzie wykładnika 00...00 z bitem wiodącym 1 (tzw. *pseudo-zdenormalizowane*). Kody takie nie są generowane przez jednostki zmiennoprzecinkowe IA-32 (począwszy od 80387), ale są obsługiwane poprawnie jeśli zostaną użyte jako operandy, jednakże generowany jest wyjątek „argument zdenormalizowany”.

Działania

Mnemoniki z dopisaną literą P oznaczają zdjęcie argumentu ze stosu (pop) po zakończeniu działania) oraz zwiększenie wskaźnika stosu.

Brak specyfikacji argumentu potrzebnego oznacza ST(0)

kopiowanie (load/store/move/exchange)

FLD/FST/FSTP – argumenty zmiennoprzecinkowe, domyślny operand ST(0), argumenty mogą też być wyspecyfikowane, w FLD argumenty są konwertowane na format wewnętrzny)

FILD/FIST/FISTP – argumenty całkowite, domyślny operand ST(0)

FBLD/FBSTP – argumenty dziesiętne BCD, domyślny operand ST(0)

FXCH – argumenty ST(i), ST(0)

FCMOVcc – warunkowe kopiowanie ST(i) do ST(0)

Literatura FPU

IA-32 Intel Architecture Software Developer's Manual, Ch. 3&4. Instruction Set Reference, (sec. 4.2.2: "Floating-Point Data Types", sec. 4.2.1: "Signed Integers", sec. 4.7. "BCD and Packed BCD Integers", sec. 4.9. "Overview of Floating-Point Exceptions".

Przetwarzanie obrazów**Struktura pliku BMP** (ang. *bitmap image file* lub *device independent bitmap, dib*) (**obraz 2D, grafika rastrowa**)

fileheader	14 B	nagłówek pliku (do weryfikacji poprawności pliku)
DIB type	4 B	typ DIB (<i>Device Independent Bitmap</i>)→ rozmiar bitmapy
DIB header	type-var:	format pixela np. (40) BITMAPINFOHEADER , (124) BITMAPV5HEADER (Windows 98/2000 i nowsze)
maska	opcj 12/16 B	format pixela (Jeśli DIB w formacie BITMAPINFOHEADER)
Gap1	opcj var	artefakty (OS2)
pixel table	var	każdy wiersz tablicy pikseli wyrównany do wielokrotności 4 B.
Gap2	opcj var	artefakty (OS2)
profil kolorów	opcj var	może zawierać ścieżkę do pliku przechowującego profil kolorów.

Nagłówek pliku

offset	rozmiar	nazwa	opis (wartości całkowite w formacie <i>little endian</i>)
0x00	2 ASCII		najczęściej 'B' 'M' (0x42 0x4D), m.in Winows XP
0x02	4 B	filesize	rozmiar całego pliku w bajtach 0x42 0x4D = 'B' 'M' – najczęściej
0x06	4 B		zależy od aplikacji, która tworzy obraz
0x0A	4 B		offset (początkowy adres bitów w tablicy pikseli)
BITMAPINFOHEADER			
0x0E	4 B	biSize	rozmiar nagłówka informacyjnego ($\leq 40 = 0x28$)
0x12	4 B	biWidth	szerokość obrazu w pikselach
0x16	4 B	biHeight	wysokość obrazu w pikselach
0x1A	2 B	biPlanes	liczba warstw kolorów, zwykle 1
0x1C	2 B	biBitCount	liczba bitów na piksel
0x1E	4 B	biCompression	algorytm kompresji
0x22	4 B	biSizeImage	rozmiar rysunku
0x26	4 B	biXPelsPerMeter	rozdzielczość pozioma
0x2A	4 B	biYPelsPerMeter	rozdzielczość pionowa
0x2E	4 B	biClrUsed	liczba kolorów w palecie
0x32	1 B	biClrImportant	liczba ważnych kolorów w palecie (0 - wszystkie).
0x33	1 B	biClrRotation	flaga rotacja palety (0 - brak rotacji, 1 - podmiana palety)
0x34	2 B	biReserved	

Paleta kolorów (opcjonalna, po części nagłówkowej pliku .bmp)

- w trybach 8-bitowych każdy kolor opisany jest za pomocą 4 bitów (B, G, R, nieużywany lub alpha). Każdą ze składowych koloru należy podzielić przez 4.
- składowe RGB koloru piksela w obrazie zapisywane są w kolejności B,G,R, numer koloru to:

$$\text{Nr Koloru} = 65536 * \text{B} + 256 * \text{G} + \text{R} \quad (1 \text{ bajt na kolor, tryb True Color}),$$

Wiersz danych pliku BMP ma zawsze rozmiar **wielokrotności 4 bajtów**, w razie potrzeby dopisuje się bajty o wartości 0. Linie obrazu zapisywane są od dołu do góry.

W obrazach TrueColor piksel opisują 3 bajty w kolejności B, G, R; w trybie High Color 2 bajty (B – 6b, G – 5b, R – 5b). Jeśli liczba kolorów <256, numer koloru jest zapisany w jednym bajcie.

Animowane bitmapy powstają z cyklicznej podmiany kolorów w palecie, przykładem logo startowe systemu Windows. Animowane bitmapy można tworzyć dla obrazów 8-bitowych.

Program „showimage.c” (<http://www.zak.ict.pwr.wroc.pl/>)

```
gcc showimage.c -lmingw32 -lSDLmain -lSDL -lSDL_image -o showimage.exe
/*-lSDLmain - pr.gł., -lSDL - bibl. SDL, -lSDL_image - obrazek*/
/* -lmingw32 -???*/ ... (MinGW - port GCC z systemu Linux na windows)

/* Program showimage.c -- wyświetla obraz z pliku .bmp, używa bibliotek SDL*/
/* Wprowadzenie do SDL Video (http://sdl.beuc.net/sdl.wiki/FrontPage) */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <SDL/SDL.h>          /*SDL - Simple DirectMedia Layer*/
#include <SDL/SDL_image.h>

void Filter(unsigned char * buf, int width,int height,int size,char bpp) ;
SDL_Surface* Load_image(char *file_name)      /*dekl. zm. wskaźnikowej*/
{
    SDL_Surface* tmp = IMG_Load(file_name); /* Open the image file*/
    if ( tmp == NULL ) {
        fprintf(stderr, "Couldn't load %s: %s\n", file_name, SDL_GetError());
        exit(0);
    }
    return tmp;
}

void Paint(SDL_Surface* image, SDL_Surface* screen)
{
    /*BLIT (Bell Labs Intelligent Terminal) - programowalny terminal graficzny bmp*/
    SDL_BlitSurface(image, NULL, screen, NULL);
    if(SDL_BlitSurface(image, NULL, screen, NULL) < 0)
        fprintf(stderr, "BlitSurface error: %s\n", SDL_GetError());
    SDL_UpdateRect(screen, 0, 0, 0, 0);
};

int main(int argc, char *argv[])
{
    Uint32 flags;
    SDL_Surface *screen, *image;
    int depth, done;
    SDL_Event event;
    if ( ! argv[1] ) {          /* check command line usage
        fprintf(stderr, "Usage: %s <image_file>, (int) size\n", argv[0]);
        return(1);
    }
    if (! argv[2]) {
        fprintf(stderr, "Usage: %s <image_file>, (int) size\n", argv[0]);
        return(1);
    }
}
```

```

/* Initialize the SDL library */

if ( SDL_Init(SDL_INIT_VIDEO) < 0 ) {
    fprintf(stderr, "Couldn't initialize SDL: %s\n",SDL_GetError());
    return(255);
}
flags = SDL_SWSURFACE;
image = Load_image( argv[1] );
printf( "\n\nImage proports:\n" );
printf( "BitsPerPixel = %i \n", image->format->BitsPerPixel );
printf( "BytesPerPixel = %i \n", image->format->BytesPerPixel );
printf( "width %d ,height %d \n\n", image->w, image->h );
SDL_WM_SetCaption(argv[1], "showimage");
/* Create a display for the image, except that we emulate 32bpp*/
depth = SDL_VideoModeOK(image->w, image->h, 32, flags);
if (depth==0) {
    if (image->format->BytesPerPixel>1) {
        depth=32;
    } else {
        depth=8;
    }
} else
if ((image->format->BytesPerPixel>1) && (depth==8) ) {
    depth = 32;
}
if(depth == 8) /*Use the deepest native mode for non-indexed images on 8bpp scr
    flags |= SDL_HWPALLETTE;
screen = SDL_SetVideoMode(image->w, image->h, depth, flags);
if ( screen == NULL ) {
    fprintf(stderr,"Couldn't set %dx%dx%d video mode: %s\n",
        image->w, image->h, depth, SDL_GetError());
    exit(1); /* Set
}
printf("Set 640x480 at %d bits-per-pixel mode\n",screen->format->BitsPerPixel);
/* Set the palette, if one exists */
if (image->format->palette && screen->format->palette) {
    SDL_SetColors(screen, image->format->palette->colors, 0,
        image->format->palette->ncolors);
}
Paint(image, screen); /* Display the image
done = 0;
int size =atoi( argv[2] );
printf("Actual size is: %d\n", size);
while ( ! done ) {
    if ( SDL_PollEvent(&event) ) {
        switch (event.type) {

```

```

case SDL_KEYUP:
switch (event.key.keysym.sym) {
    case SDLK_ESCAPE:
    case SDLK_TAB:
    case SDLK_q:
        done = 1;
        break;
    case SDLK_SPACE:
    case SDLK_f:
        SDL_LockSurface(image);
        printf("Start filtering... ");
Filter(image->pixels,image->w,image->h, size, image->format->BytesPerPixel );
        printf("Done.\n");
        SDL_UnlockSurface(image);
        printf("Repainting after filtered... ");
        Paint(image, screen);
        printf("Done.\n");
        break;
    case SDLK_r:
        printf("Reloading image... ");
        image = Load_image( argv[1] );
        Paint(image,screen);
        printf("Done.\n");
        break;
    case SDLK_PAGEDOWN:
    case SDLK_DOWN:
    case SDLK_KP_MINUS:
        size--;
        if (size==0) size--;
        printf("Actual size is: %d\n", size);
        break;
    case SDLK_PAGEUP:
    case SDLK_UP:
    case SDLK_KP_PLUS:
        size++;
        if (size==0) size++;
        printf("Actual size is: %d\n", size);
        break;
    case SDLK_s:
        printf("Saving surface at nowy.bmp ...");
        SDL_SaveBMP(image, "nowy.bmp" );
        printf("Done.\n");
    default:
        break;
}

```

```

        break;
//      case  SDL_MOUSEBUTTONDOWN:
//          done = 1;
//          break;
    case SDL_QUIT:
        done = 1;
        break;
    default:
        break;
    }
    } else {
        SDL_Delay(10);
    }
}
SDL_FreeSurface(image);
/* We're done! */
SDL_Quit();
return(0);
}

```

Pobranie piksela obrazu – getpixel()

/*Return the (x,y) pixel value. The surface must be locked before calling this!*/
 Uint32 getpixel(SDL_Surface *surface, int x, int y)

```

{
    int bpp = surface->format->BytesPerPixel;
    /* Here p is the address to the pixel we want to retrieve */
    Uint8 *p = (Uint8 *)surface->pixels + y * surface->pitch + x * bpp;
    switch(bpp) {
    case 1:
        return *p;
    case 2:
        return *(Uint16 *)p;
    case 3:
        if(SDL_BYTEORDER == SDL_BIG_ENDIAN)
            return p[0] << 16 | p[1] << 8 | p[2];
        else
            return p[0] | p[1] << 8 | p[2] << 16;
    case 4:
        return *(Uint32 *)p;
    default:
        return 0; /* shouldn't happen, but avoids warnings */
    }
}

```

Wstawianie piksela – putpixel()

```

void putpixel(SDL_Surface *surface, int x, int y, Uint32 pixel){
    int bpp = surface->format->BytesPerPixel;

```

```

uint8 *p = (uint8 *)surface->pixels + y * surface->pitch + x * bpp;
switch(bpp) {
case 1:
    *p = pixel;          /* p - adres pixela
    break;
case 2:
    *(uint16 *)p = pixel;
    break;
case 3:
    if(SDL_BYTEORDER == SDL_BIG_ENDIAN) {
        p[0] = (pixel >> 16) & 0xff;
        p[1] = (pixel >> 8) & 0xff;
        p[2] = pixel & 0xff;
    } else {
        p[0] = pixel & 0xff;
        p[1] = (pixel >> 8) & 0xff;
        p[2] = (pixel >> 16) & 0xff;
    }
    break;
case 4:
    *(uint32 *)p = pixel;
    break;
}
}
/* ustawienie żółtej plamki ((R=0xff, G=0xff, B=0x00)) na środku ekranu */
int x, y;
uint32 yellow;
/* Note: If the display is palettized, you must set the palette first. */
yellow = SDL_MapRGB(screen->format, 0xff, 0xff, 0x00);
x = screen->w / 2;
y = screen->h / 2;
/* Lock the screen for direct access to the pixels */
if ( SDL_MUSTLOCK(screen) ) {
    if ( SDL_LockSurface(screen) < 0 ) {
        fprintf(stderr, "Can't lock screen: %s\n", SDL_GetError());
        return;
    }
}
putpixel(screen, x, y, yellow);
if ( SDL_MUSTLOCK(screen) ) {
    SDL_UnlockSurface(screen);
}
}

```

Przykładowe rozwiązania**Rozmycie obrazu kolorowego (przekształcanie grup 9/16 sąsiednich pixeli)**

```

void blur(unsigned char *buf, int width, int height, char bpp){
int rowsize=(bpp*8*width+31)/32;          /*poprawka na długość kolumny
rowsize*=4;                               /* kolejne wiersze pikseli są upakowane tak, aby ich
int i=0;                                  /* szerokość była wielokrotnością 4 bajtów
int j=0, k=0;
int col=0;
int * sr= malloc(3*sizeof(int)); /*średnie trzech kolorow
for (i=0;i<height;i+=4){                  /*kolejne 4 wiersze
    for(j=0;j<rowsize;j+=12){              /*kolejne 4 piksele (12B)
        for (col=0; col<3;col++){          /*kolejne kolory
            sr[col]=0;
            for(k=0;k<4;k++){              /*kolejne 4 kolumny (po 3B)
                sr[col]+=(buf[rowsize*i+j+k*3+col]/=16); /* srednia: kazda wartosc /16
                sr[col]+=(buf[rowsize*i+j+rowsize+k*3+col]/=16);
                sr[col]+=(buf[rowsize*i+j+2*rowsize+k*3+col]/=16);
                sr[col]+=(buf[rowsize*i+j+3*rowsize+k*3+col]/=16);
            }
            for(k=0;k<4;k++){              /*kolejne 4 kolumny
                buf[rowsize*i+j+k*3+col]=sr[col];          /* zmiana wartosci wierszy
                buf[rowsize*i+j+rowsize+k*3+col]=sr[col];
                buf[rowsize*i+j+2*rowsize+k*3+col]=sr[col];
                buf[rowsize*i+j+3*rowsize+k*3+col]=sr[col];
            }
        }
    }
};

```

Obrót obrazu o 90 stopni (wymiaru obrazu zostają zamienione)

```

void Filter(SDL_Surface *screen, int width,int height,int size,char bpp){
    SDL_Surface new_screen; /*nowa tymczasowa powierzchnia
    new_screen = SDL_CreateRGBSurface(0,height,width,32,0,0,0,0); /* nowe wymiary
    Uint32 old_pixel;        /* obecnie przetwarzany piksel
    int x, y;
    for (x = 0; x < width; x++) {
        for (y = 0; y < height; y++) {
            old_pixel = getpixel(screen, width - x, height - y); /*pobranie
            putpixel(new_screen, y, x, old_pixel); /*wstawienie w innym miejscu
        }
    }
    screen = SDL_SetVideoMode(height, width, 32, SDL_SWSURFACE);
    SDL_BlitSurface(new_screen,NULL,screen,NULL); /* stara powierzchnia nową
    SDL_FreeSurface(new_screen); /* zwolnienie pamięci
    SDL_UpdateRect(screen, 0, 0, 0, 0); /* odświeżenie obrazu
}

```

Obrót w poziomie (horizontal mirror)

```
void HorizMirror (unsigned char *buf, int width, int height, int size, char bpp){
char temp = 0 ;
for (int w = 0; w < height; w++) { /* row
    for ( int k = 0; k < width /2; k++) { /*column
        temp = buf [w*width + k];
        buf [w*width + k] = buf [w*width+width - k] ;
        buf [w*width + width - k] = temp ;
    }
}
}
```

Obrót w pionie (vertical mirror)

```
void VertMirror (unsigned char *buf, int width, int height, int size, char bpp){
char temp = 0 ;
for (int w = 0; w < height/2; w++) { /* row
    for ( int k = 0; k < width; k++) { /* column
        temp = buf [w*width + k];
        buf [w*width + k] = buf [(height - w - 1)*width + k] ;
        buf [(height - w - 1) + k] = temp ;
    }
}
}
```

Obrót w pionie (vertical mirror)

```
void VertReflect (unsigned char *buf, int width, int height, int size, char bpp){
char temp = 0 ;
for (int i = 0; i < height; ++i) {
    for (int j = 0; j < width / 2; ++j) {
        for (int k = 0; k < bpp; ++k) {
            char tmp = *(buf+i*pitch+(width-j-1)*bpp+k);
            *(buf+i*pitch+(width-j-1)*bpp+k) = *(buf+i*pitch+j*bpp+k);
            *(buf+i*pitch+j*bpp+k) = tmp;
        }
    }
}

void Reflect (SDL_Surface *reflected, SDL_Surface *image){
uint32 pixel;
for(int y=0; y<image->h; y++){
for(int x=0; x<image->w; x++){
    [pixel = getpixel(image, image->w-x, y);] /* horizontal (w poziomie)
    [pixel = getpixel(image, x, image->h-y);] /* vertical (w pionie)
putpixel(reflected, x, y, pixel);
}
}
}
```

negatyw

```
__asm volatile("\n
    for (int i = 0; i < height * pitch; i += 8){
        pcmpeqd %%mm1, %%mm1 \    /* -1 do mm1
        movq (%0), %%mm0; \
        pandn %%mm1, %%mm0; \    /* (not mm0) & mm1
        [pxor %%mm1, %%mm0; \] /* (not mm0)
        movq %%mm0, (%0) \
        " :: "r"(buf + i) : "memory", "%mm0");
    }
```

negatyw

```
void Negative(SDL_Surface *filtered, SDL_Surface *image){
    Uint32 pixel;
    int mask = -1;
    __asm__ ("movq %0, %%mm0;"
    :: "ym" (mask));
    for(int y=0; y<image->h; y++){
        for(int x=0; x<image->w; x++){
            pixel = getpixel(image, x, y);
            __asm__ ("movq %1, %%mm1;"          /* -1 do mm1
            "pxor %%mm0, %%mm1;"              /* not
            "movq %%mm1, %0;"
            : "=r" (pixel)
            : "r" (pixel));
            putpixel(filtered, x, y, pixel);
        }
    }
}
```

rozjaśnianie obrazu (asm+mmx)

```
.type filtruj, @function    # rozjaśnianie obrazu
.global filtruj
filtruj:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax    # wskaźnik do wektora rozjaśniającego pozostałe piksele
    movl 12(%ebp), %edx    # wskaźnik na pierwszy piksel bitmapy
    movl 16(%ebp), %ecx    # liczba iteracji
    movq %ebx, %mm0    # załadowanie wektora??? do rejestru MMX
    xor %eax, %eax    # wyzerowanie eax, indeks pikseli bitmapy
petla:
    movq (%edx,%eax,8), %mm1    # załadowanie kolejnych 8 bajtów do rejestru MMX
    paddusb %mm0, %mm1    # dodanie z nasyceniem (rozjaśnienie)
    movq %mm1, (%edx,%eax,8)    # zwrot wyniku
    incl %eax    # przesuwamy indeks pikseli bitmapy
```



```

decl %ecx          # zmniejszamy liczbę iteracji
cmp $0, %ecx
jnz petla          # powtarzamy dopóki ecx != 0
emms               # czyścimy status rejestrów MMX
movl %ebp, %esp
popl %ebp
ret

```

przyciemnienie i rozjaśnienie

```

for (int i = 0; i < height * pitch; i += 8){
    __asm volatile("\
        movq (%0), %%mm0; \
        movq %%mm0, %%mm2; \
        pxor %%mm1, %%mm1; \
        punpckhbw %%mm1, %%mm2; \ /*rozpakowanie
        punpcklbw %%mm1, %%mm0; \
        movq %%mm2, %%mm4; \
        movq %%mm0, %%mm5; \
        psrlw $4, %%mm4; \      /*przesunięcie 4 bitów
        psrlw $4, %%mm5; \
        paddw %%mm4, %%mm2; \ /* psubw ...
        paddw %%mm5, %%mm0; \ /* psubw ...
        packuswb %%mm2, %%mm0; \ /* spakowanie
        movq %%mm0, (%0); \
    " :: "r"(buf + i) : "memory", "%mm0", "%mm1", "%mm2");
}

```

rozmycie obrazu

Średnia z piksela i przyległych do niego. Maski wycinają poszczególne kolory. W celu skalowania kolorów w MMX, każdy bajt jest rozszerzany na 2 bajty i mnożony przez czynnik skalujący. Suma tworzona w rejestrze %mm0, zredukowana do 1 bajtu a piksel jest zapisywany w obrazie wynikowym.

```

void Blur(SDL_Surface *filtered, SDL_Surface *image){
    uint64 pix;
    uint32 pixel;
    uint64 m1=0x00ff0000, m2=0x0000ff00, m3=0x000000ff;
    uint64 m4=0x000000ff00000000, m5=0x0000000000ff0000, m6=0x00000000000000ff;
    uint64 p[9], coefficient=0x1c711c711c711c71, result;
    for(int y=1; y<image->h-1; y++){
        for(int x=1; x<image->w-1; x++){
            for(int dx=0; dx<=2; dx++){
                for(int dy=0; dy<=2; dy++){
                    pixel=getpixel(image,x+dx-1, y+dy-1);
                    pix=(uint64)pixel;
                    p[2*dx+dy]=((pix & m1) << 16) | ((pix & m2) << 8) | (pix & m3);
                }
            }
        }
    }
}

```

```

__asm__("movd %1, %%mm0;"
"movd %10, %%mm2;"
"pmulhw %%mm2, %%mm0;"
"movd %2, %%mm1;"
"pmulhw %%mm2, %%mm1;"
"padd %mm1, %%mm0;"
"movd %3, %%mm1;"
"pmulhw %%mm2, %%mm1;"
"padd %mm1, %%mm0;"
"movd %4, %%mm1;"
"pmulhw %%mm2, %%mm1;"
"padd %mm1, %%mm0;"
"movd %5, %%mm1;"
"pmulhw %%mm2, %%mm1;"
"padd %mm1, %%mm0;"
"movd %6, %%mm1;"
"pmulhw %%mm2, %%mm1;"
"padd %mm1, %%mm0;"
"movd %7, %%mm1;"
"pmulhw %%mm2, %%mm1;"
"padd %mm1, %%mm0;"
"movd %8, %%mm1;"
"pmulhw %%mm2, %%mm1;"
"padd %mm1, %%mm0;"
"movd %9, %%mm1;"
"pmulhw %%mm2, %%mm1;"
"padd %mm1, %%mm0;"
"movd %%mm0, %0;"
:"=r" (result)
:"r" (p[0]), "r" (p[1]), "r" (p[2]), "r" (p[3]), "r" (p[4]),
"r" (p[5]), "r" (p[6]), "r" (p[7]), "r" (p[8]), "r" (coefficient));
pixel = ((result & m4) >> 16) | ((result & m5) >> 8) | (result & m6);
putpixel(filtered, x, y, pixel);
}
}
}

```

filtrowanie (użycie maski)

```

movq (%rdi),%mm0      # adres pierwszego bajtu w %rdi - kopia 8B do mm0
movq $mask,%mm1      #
    [pxor %mm1,%mm0]   # negatyw (maska =-1)
    [paddusb %mm1,%mm0] # rozjaśnianie (maska=byte-const, np 0x0A0A0A0A0A0A0A0A)
    [psubusb %mm1,%mm0] # zaciemnianie (maska=byte-const, np 0x0A0A0A0A0A0A0A0A)

```

Konwersja wartości piksela na komponenty RGB: `SDL_GetRGB()`, odwrotna: `SDL_MapRGB()`.