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 – asembler, gprof – profiler, ld ...

[2] „Using as", dokumentacja kompilatora, http://sourceware.org/binutils/docs-2.22/as/index.html
rozdział 3 – składnia pliku źródłowego programu asemblerowego, szczególną uwagę proszę zwrócić na wpływ białych znaków, sposób tworzenia komentarzy oraz stałych
rozdział 4 – segmenty programu, typy segmentów zawierających dane
rozdział 7 – dyrektywy asemblera, 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

Umiejętności:

a) kompilacja źródeł asemblerowych za pomocą kompilatora **as** i konsolidatora (linkera) **ld** oraz kompilatora **gcc**, znajomość podstawowych opcji kompilatora i linkera (o, g, e, l, v, c)

b) tworzenie prostych plików sterujących przetwarzaniem za pomocą programu **make**

c) podstawy tworzenia kodu asemblerowego – struktura programu, deklarowanie zmiennych, atrybuty zmiennych (adres, wartość i rozmiar zmiennej jednostkowej), tryby adresowania, użycie podstawowych funkcji systemowych (przekazywanie argumentów i zwracanie wyniku; przeczytać plik unistd.h – identyfikatory funkcji i podręcznik systemowy **man**).

Przykładowe zadania:

a) przepisać i skompilować kod zamieszczony we wzorcowym sprawozdaniu

b) zmodyfikować kod do postaci pozwalającej wykorzystać funkcję systemową read

c) przekształcić tekst za pomocą masek logicznych lub szyfrem Cezara

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

Przykładowe zadania

1. Przetwarzanie tekstu
   1. 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.
   2. Zaszyfrować szyfrem Cezara ciąg liter ASCII ze strumienia wejściowego
2. Jak działają instrukcje i co jest zapisane w pamięci
   1. Zbadać (gdb) dokładnie działanie instrukcji dzielenia (div, idiv) oraz mnożenia (mul, imul).
   2. Oglądnąć obraz programu, jego danych i stosu programowego w pamięci komputera
3. Przetwarzanie liczb
   1. Pobrać ze standardowego strumienia wejściowego ciąg znaków dowolnej długości, potraktować każdy znak jako zapis cyfry szesnastkowej (np. znak 'A' odpowiada wartości 10 dziesiętnie) i skonstruować liczbę w kodzie naturalnym binarnym i/lub U2. Wynik wypisać na standardowy strumień wyjściowy. Zakładamy, że wprowadzane są wyłącznie znaki 'A" F' i cyfry. Uzupełnić kodem wykrywającym niepoprawne znaki we wczytanym ciągu.
   2. Pobrać ze standardowego strumienia wejściowego ciąg bajtów o danej długości i wyprowadzić na standardowe wyjście ciąg znaków reprezentujący szesnastkowy zapis wartości tej liczby.
   3. Pobrać ze standardowego strumienia wejściowego dowolnej długości ciąg cyfr dziesiętnych reprezentujących zapis dziesiętny pewnej liczby i zapisać wartość tej liczby binarnie w pamięci.
   4. Binarną reprezentację liczby całkowitej zapisaną w kolejnych słowach pamięci wypisać na standardowe wyjście w postaci ciągu znaków reprezentujących jej wartość dziesiętną.

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

Przykładowe zadania – funkcje

1. Proste operacje arytmetyczne (dodawanie, odejmowanie, mnożenie) na wielkich liczbach (kilka tysięcy bitów) w kodzie naturalnym binarnym i/lub U2. Rozmiar liczby jest wielokrotnością rozmiaru słowa maszynowego.
2. Największy wspólny dzielnik zgodnie z algorytmem Euklidesa dla liczb o dowolnej długości.
3. Generator liczb pierwszych zgodnie z algorytmem sita Erastotenesa.
4. Sortowanie tablicy liczb 64/96/128-bitowych.
5. Liczby Fibonacciego wielokrotnej precyzji (krotność długości słowa maszynowego).

3. Technika łączenia różnych języków programowania, optymalizacja (analiza wydajności kodu).
Przykładowe zadania **(! wyłączyć opcje otymalizacji w kompilatorze gcc)**

a) napisać kod w asemblerze pozwalający używać funkcji napisanych w języku C i operujących na argumentach stałoprzecinkowych i zmiennoprzecinkowych (funkcje biblioteczne i własne).
b) napisać zestaw funkcji w asemblerze umożliwiających ich użycie z poziomu języka C. Funkcje te powinny operować na argumentach różnych typów i zwracać wartości różnych typów.
c) napisać kod pozwalający na używanie zmiennych i stałych zdefiniowanych w C z poziomu asemblera i zdefiniowanych w asemblerze z poziomu C.
d) napisać w języku C zestaw funkcji wykonujących operacje na dużych (o rozmiarze liczonym w megabajtach lub dziesiątkach megabajtów) tablicach jedno i dwuwymiarowych.

Na podstawie pomiarów czasu wykonania oraz analizy kodu asemblerowego powstałego podczas kompilacji zlokalizować najbardziej czasochłonne fragmenty programu i odpowiednio je modyfikując skrócić czas wykonania programu. Do pomiaru czasu wykonania wykorzystać zarówno dostępne narzędzia (np. gprof), jak i mechanizmy wbudowane w procesor (ang. Time Stamp Counter).

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 wykonywalnym oraz o ich ładowaniu do pamięci i uruchamianiu), http://www.linuxjournal.com/article/1059 (...article/1060)

4. Jednostka zmiennoprzecinkowa procesorów rodziny x86/Pentium/i#.
Przykładowe zadania

a) ustawić i sprawdzić status jednostki zmiennoprzecinkowej (np.: precyzję obliczeń, tryb zaokrąglania, wystąpienie wyjątków itp.). (funkcje/działania generujące poszczególne wyjątki Funkcje powinny mieć interfejs pozwalający na ich wywołanie z poziomu języka C.
b) obliczanie SQRT lub pola wg wzoru Herona dla krytycznych danych
c) opracowanie zestawu funkcji w języku C lub asembler obliczających wartości funkcji opisanych zadanym wzorem (rozwinięcia funkcji trygonometrycznych (sin, cos) w szereg Taylora, metody obliczania całek oznaczonych itp. Dobrać kolejność i liczbę operacji zmiennoprzecinkowych, aby uzyskać wynik mieszczący się w zadanej precyzji obliczeń .
d) 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 : +/- 1/2 amplitudy (2**(k-1)) np. znacznik wewnątrz funkcji
ii) ostrzeżenia: np. NaN, niedozwolony argument,

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

Przykładowe programy ("Programming from the ground up")

Dzielenie i mnożenie liczb 1024-bitowych (NB i/lub U2)

Generowanie liczb pierwszych $<2^{16}$ (sito Eratostenesa)

Generowanie m kolejnych liczb pierwszych z ustalonego zakresu ($>p$)

Znajdowanie NWD dwóch liczb 1024-b (algorytm Euklidesa)

Rozwiązywanie równania $a \cdot x \bmod p = 1$ metodą Euklidesa.

Potęgowanie modulo liczb 1024-b (tw. Fermata: $a^{p-1} \bmod p = 1$, redukcja reszty: $a^x \bmod p = a^{x \bmod (p-1)} \bmod p$,

redukcja argumentu: $a^x \bmod p = (a-p)^x \bmod p$)

Symulacja dodawania i odejmowania zmiennoprzecinkowego w dowolnym formacie

Symulacja mnożenia i dzielenia zmiennoprzecinkowego w dowolnym formacie

Rozkład wielomianu nad GF(2) na czynniki nierozkładalne

Generowanie permutacji wektora 32-pozycyjnego

Sprawdzanie czy liczba jest liczbą pierwszą (test Millera-Rabina, test Fermata)

Oszacowanie pierwiastka kwadratowego wg wzoru $n^2 = m^2 + \sum_{m+1}^{n}(2i-1)$ z liczby całkowitej

z wykorzystaniem wstepnego oszacowania: jeśli $2^{2k-2} \le X < 2^{2k}$, to $2^{k-1} < sqrt(X) < 2^k$

Oszacowanie stałoprzecinkowego pierwiastka kwadratowego z ustaloną dokładnością

Oszacowanie pierwiastka 3 stopnia wg wzoru $n^3 - n = 3\sum_{i=1}^{n}(i \cdot (i-1)) = 6\sum_{j=1}^{n-1}\sum_{i=1}^{j} i$ z wykorzystaniem

wstepnego oszacowania: jeśli $2^{3k-3} \le X < 2^{3k}$, to $2^{k-1} \le \sqrt[3]{X} < 2^k$

Obliczanie prawdopodobieństwa wg zmodyfikowanego wzoru Bernoulliego ($p^2 \sum_{i=0}^{n-k}\binom{k+i-1}{i}(1-p)^i$)

Sortowanie bąbelkowe zbioru liczb całkowitych – wynik w tablicy adresów liczb

Sortowanie quik-sort zbioru liczb całkowitych – wynik w tablicy adresów liczb

Sortowanie zbioru liczb całkowitych – wynik w tablicy adresów liczb

Odczytanie nagłówka odczytanego pliku

Wyszukanie tekstu w odczytanym pliku

Szyfrowanie i deszyfrowanie wczytanej sekwencji cyfr uproszczonym szyfrem ElGamala

Zliczanie bitów „1" w podanym zakresie pamięci.

Wyszukiwanie najdłuższej sekwencji bitów „1" w podanym zakresie pamięci.

Kompresja i dekompresja ciągu cyfr metodą Huffmanna (np. $P(i) = 2^{-i-1}$ dla $i=0,1,2,\ldots,9$)

Rekurencja - obliczanie silni, kombinacji C(k,n), wariacji, liczb Fibonacciego

Obliczenie skalowanej odwrotności liczby 1024-b (metoda Newtona)

Obliczenie skalowanej odwrotności pierwiastka z liczby 1024-b (metoda Newtona)

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: "lstudent")
    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)
```

polecenie 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] - powrór 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** – **i**nsert, *wstaw przed kursorem* / *od początku wiersza* (*shift & insert*)

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

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

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

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

o / **O** – *wstaw wiersz poniżej* / *powyżej*

(polecenie → akcja (1) → (2))

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

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

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

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

c.. – **c**hange, *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 – **j**oin, *przyłącz następny wiersze* (usuń znak NewLine)…

u – **u**ndo, *anuluj* poprzednią akcję

^r – **r**edo, *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* – **w**rite, zapisz do pliku (**:w!** *nazwa* – nadpisanie istniejącego pliku *nazwa*)

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

:wq [*plik*] – **w**rite & **q**uit, 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 – przesuń (**m**ove) /kopiuj (**co**py) 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-kluczowej (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 asemblera: -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 maszynę 64b AMD)
ld pr.o -m elf_i386 -o pr  # generuj kod 32-b na maszynę 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 bieżacym/##
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          # wznów 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
```

Konwencje asemblera

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 (bin[ary]), t=0,1, np. 0b1011
0f[-]i,fE[-]e            # liczba float, i,f,e dziesiętnie np. -314,59 x10-2 –
   '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  # 3 cyfry ósemkowe (kod ASCII)   \xDD # 2 cyfry szesnastkowe
   \n    # (\x0A=\012) LF,NL, new line    \0   # (\x00=\000) NUL, koniec rekordu
   \b    # (\x08=\010) BS, backspace      \t   # (\x09=\011) HT,TAB, tabulation
   \f    # (\x0C=\014) FF, form feed      \r   # (\x0D=\015) CR, carriage return
   \\, \", \?              # ukośnik, apostrof, znak zapytania (znaki specjalne)
```

UWAGA: specyfikacja każdej stałej jako argumentu instrukcji musi być poprzedzona znakiem $

Obsługa kodów ASCII (przykład) – składnia asemblera 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 = 0b1101111 – 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 asemblera 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, stala       # lokalne przypisanie wartości (.data)
VAL = stala           # 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

```
mnemo 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 adresowanie
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 deskryptorze.
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/lhpsyscal.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
andb $0xDF, %al         # ujednolicenie – wszystkie litery duże
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 asemblera

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:

`O_RDONLY`	– 00	– tylko odczyt (read-only mode)
`O_WRONLY`	– 01	– tylko zapis (write-only mode)
`O_RDWR`	– 02	– zapis i odczyt (both reading and writing)
`O_CREAT`	– 0100	– utworzenie pliku, jeśli nie istnieje (create the file if it doesn't exist)
`O_TRUNC`	– 01000	– skasuj zawartość pliku, jeśli istnieje (erase the contents of the file)
`O_APPEND`	– 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 (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)
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
```

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

                           # argumenty wywołania przekazywaneprzez 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:                   # jeśli bufor ma długość zero, funkcja nie jest użyta
movl 12(%ebp), %eax        # adres bufora
movl 8(%ebp), %ecx         # rozmiar bufora
decl %eax                  #
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 zwrotnie 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      # prinf("%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. wskutek 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 asemblerowy 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
.globl _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 poziomu 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" wskaźnik 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
BUFWY: .ascii „           \n”     # miejsce na 10 cyfr
BWY_LEN=.-BUFWY             # 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, BUFWY(%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 $BUFWY, %ecx           # adres startowy bufora ($BUFWY) 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:

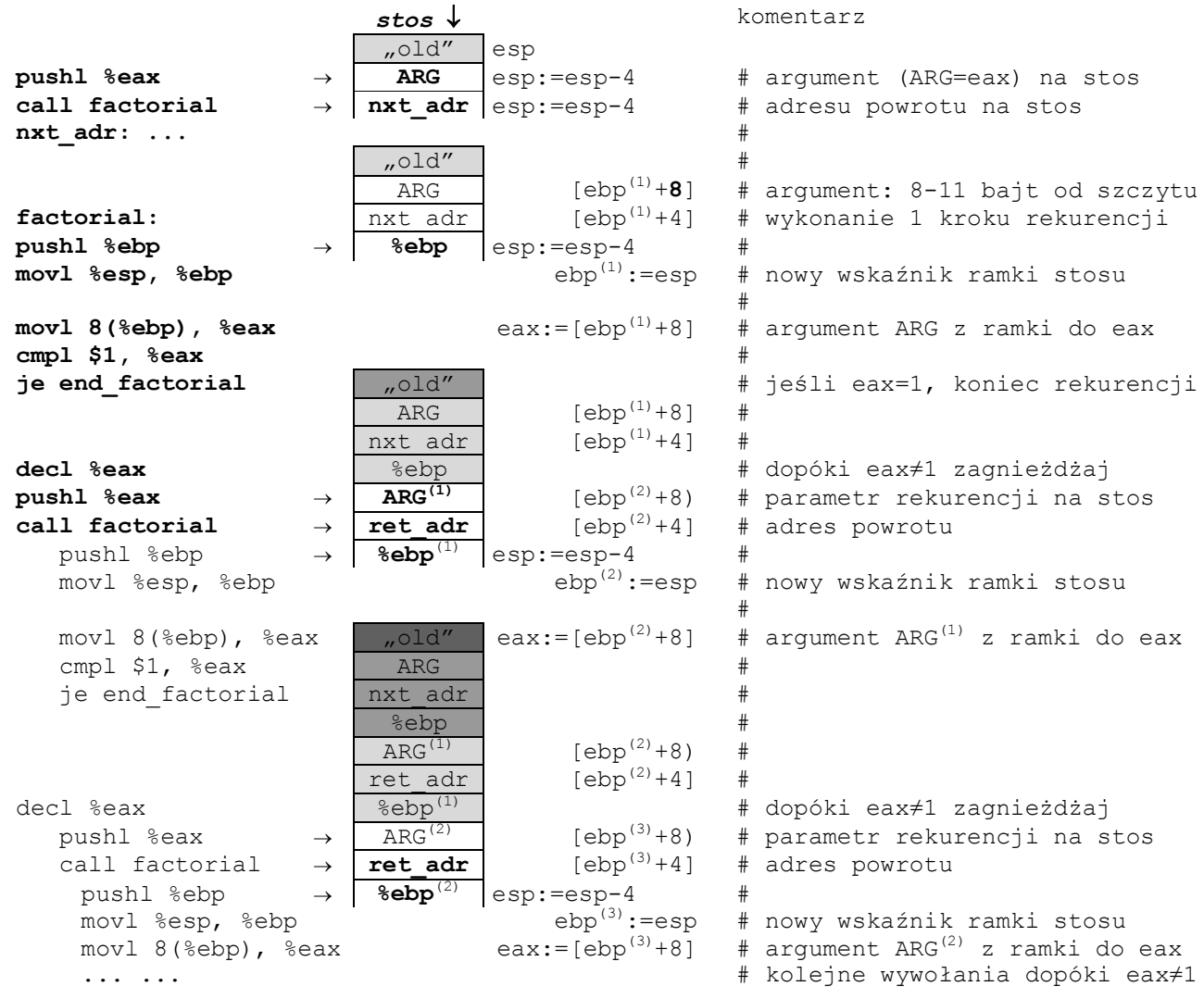
- przekazywanie parametrów na kolejny poziom zagniedżenia (przez stos)
- sprawdzenie warunku zakończenia rekurencji
- wywołanie funkcji lub ominięcie jeśli koniec rekurencji
- wykonanie kolejnych obliczeń
- odtworzenie stosu poprzedniego poziomu
- przekazanie wyniku do poziomu poprzedniego wywołania

```
...                     # 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
```

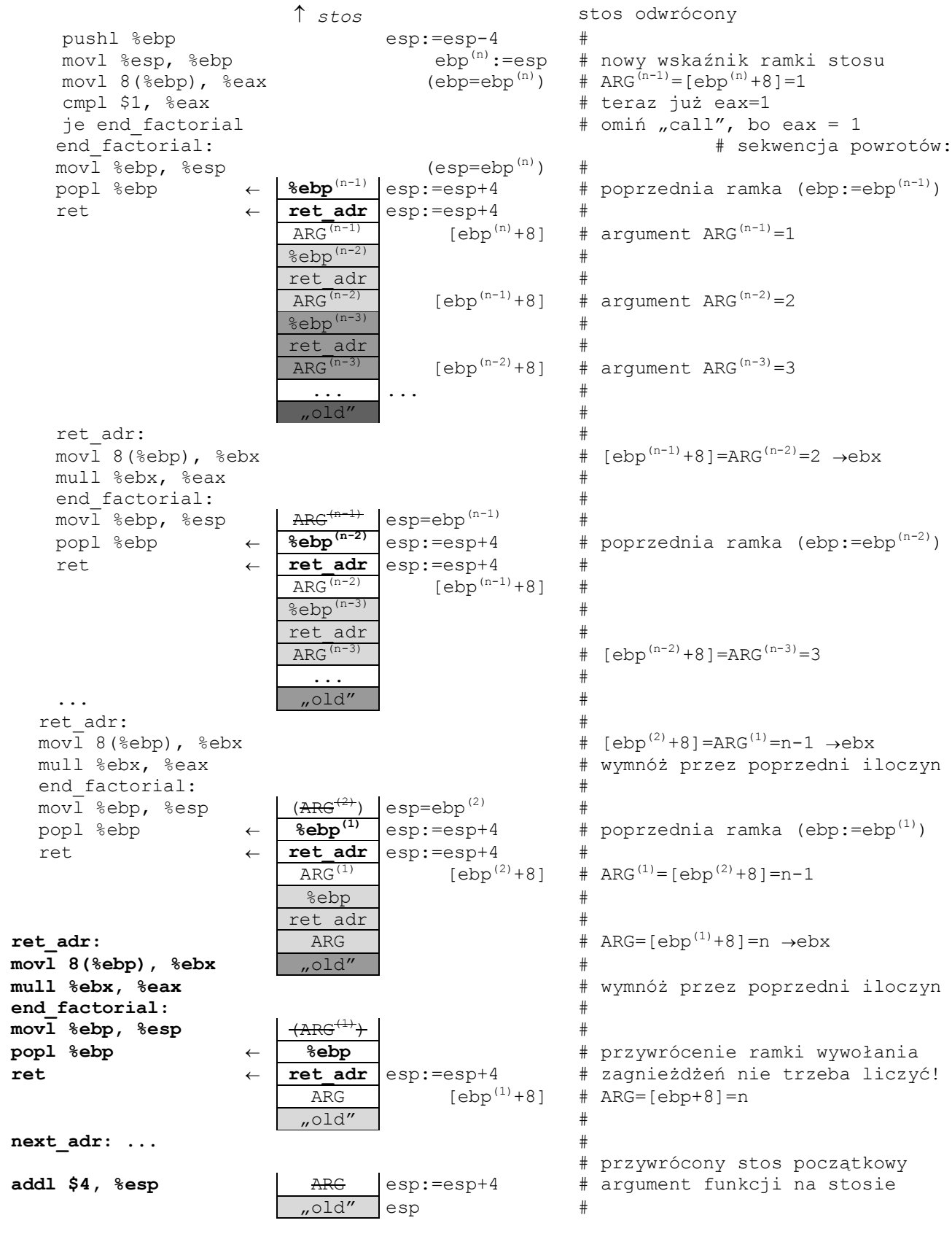
Funkcja rekurencyjna (obliczanie silni)

```
.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)



– powroty (zwalnianie stosu)



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

```
                        # argumenty: zmienne nn i kk; V(nn,kk)=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 zmniennej wynik

.type var @function     /* *) alternatywne zestawy instrukcji, wymaganie początkowe %eax=1 */
var:
[„push …”]              # rejestry f. wywołującej na stos (gcc)
pushl %ebp              # “stary” wskaźnik powiązania dynamicznego na stos
movl %esp, %ebp         # 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
cmpl $1, %ecx           # 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
movl 8(%ebp), %ebx      # parametr rekurencji do %ebx, bo %eax zawiera wynik
mull %ebx               # kolejne obliczenie (ew. imull %ebx, %eax)
end_var:                # pierwsze wejście: %eax=n-k+1
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
```

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              #
jl 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
   jl 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       # rekurencyjne obliczanie kombinacji wg trójkata Pascala
pascal:                      # C(n,k)= C(n-1,k)+ C(n-1,k-1)
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" wskaźnik powiązania
ret                          # zwrot sterowania do miejsca wywołania
```

```
.type newton @function      # rekurencyjne obliczanie kombinacji
newton:                     # C(n,k)=n!/(k!(n-k)!)= C(n-1,k-1)*n/k
[„push …”]                  # rejestry f. wywołującej na stos (gcc)
pushl %ebp                  # “stary” wskaźnik powiązania dynamicznego na stos
movl %esp, %ebp             # nowy wskaźnik: szczyt stosu stosu) do %ebx
movl 12(%ebp), %ecx         # parametr k* (z wnętrza stosu) do %ecx
movl 8(%ebp), %eax          # parametr n* (z wnętrza stosu) do %ebx
cmpl $1, %ecx               # warunek końca
je end_comb                 # koniec sekwencji wywołań rekurencyjnych
decl %ecx                   # obliczenie kolejnego argumentu rekurencji
decl %eax                   # obliczenie kolejnego argumentu rekurencji
pushl %ecx                  # kolejny argument rekurencji (k*) na stos
pushl %eax                  # kolejny argument rekurencji (n*) na stos
call newton
movl 8(%ebp), %ebx          # parametr rekurencji do %ebx, bo %eax zawiera wynik
movl 12(%ebp), %ecx         # parametr rekurencji do %ecx
mull %ebx                   # kolejne obliczenie (ew. imull %ebx, %eax)
div %ecx
end_comb:                   # pierwsze wejście: %eax=n-k+1, %ecx=1
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
```

Łączenie as z C/C++

Główne problemy przy łączeniu asemblera z kodem w języku wyższego poziomu to przekazanie danych i wyników oraz zapobieganie wzajemnemu niszczeniu zawartości rejestrów.

Użycie funkcji bibliotecznych C z poziomu asemblera,

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               # albo jak w as: mov $1, %eax - mov $0, %ebx - int $0x80
```

```
# 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
```

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

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

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

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

Aby zablokować optymalizację kodu asemblerowego przydatne jest pisanie __volatile__ po __asm__. Bez tego **gcc** może uznać, że nasz asembler 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 asemblera. 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 asemblerze z poziomu C i odwrotnie
- wywołanie własnej funkcji napisanej w C z poziomu asemblera
- wywołanie własnej funkcji argumentów zmiennoprzecinkowych
- korzystanie z funkcji napisanych w asemblerze z poziomu języka C

Funkcja `main()` w pliku z kodem w C zawiera wywoływanie funkcji asemblerowej, 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)` koprocesora (FPU)
– wypisanie wyniku z FPU (`st(0)`) za pomoca `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("Wywolanie 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 zewnetrznej
printf("Zmienna z asemblera: %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 asemblera, wynik funkcji z C = %d\n\0"
napis2:.ascii "Argument z asemblera, 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 sotsu programowego
push $napis2              # adres napisu jako ciag formatujący dla printf
call printf               # wypisz informacje
add $4, %esp
leave                     # usuń 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 zastąpione 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); //wywolaj badana funkcje
free(dane); //zwolnij pamiec
return 0;
}
```

Wywołanie funkcji `__asm()` pozwala na wstawienie kodu asemblerowego 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 0x BFFF 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`.

UWAGA: podobne, ale nierównoważne działanie (pushl i popl są nierozdzielne wobec przerwania):

```
movl %eax, (%esp)              # „pushl %eax”
subl $4, %esp
movl (%esp), %eax              # „popl %eax”
addl $4, %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

Dostęp do obszaru zakazanego, z ograniczeniami wynikającymi ze struktury systemu operacyjnego) można uzyskać za pomocą komunikatu przekazanego funkcjom jądra (kernel) systemu operacyjnego.

System plików UNIX/Linux

/ – katalog główny (root) –

/root – pliki prywatne superużytkownika

/usr – udostępnione dane i programy wspólnego użytku (**use**r)

/dev – sterowniki urządzeń (**dev**ices)

/bin – podstawowe polecenia systemowe (**bin**ary)

/etc – pliki konfiguracyjne systemu (**et c**etera)

/tmp – pliki tymczasowe (**tmp**orary)

/home – pliki prywatne (domowe) użytkowników

Procesor poleceń / powłoka (ang. shell) oraz pliki konfiguracyjne i znak zachęty (ang. prompt)

($) **Bourne shell** (oficjalny dostarczany z systemem UNIX) – autoexec: **.profile**

(%) **C shell** (bardziej elestyczny, wolniejszy) – autoexec: **.profile** oraz **.login**

($) **Korn shell** (to co najlepsze z Bourne & C shells)

plik **.profile** zawiera zmienne środowiskowe i polecenia, np.

HOME=/usr/john nazwa katalogu użytkownika

PATH=/bin:usr/bin:$HOME/bin ścieżki wyszukiwania programów

MAIL=/usr/spool/mail/'basename $HOME' ścieżka poczty

TERM tv950 typ terminala

export HOME PATH MAIL TERM polecenie wyeksportowania do systemu

w **C shell** zamiast export jest polecenie **setenv** dla każdej zmiennej np. setenv HOME ..

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-g6/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.)