Składnia pliku źródłowego

Niniejszy dokument nie jest w żadnym wypadku kompletnym opisem środowiska i metodyki programowania w asemblerze. Stanowi on raczej przewodnik po dokumentacji, do której lektury gorąco zachęcam. Podczas lektury dokumentacji należy pamiętać, że GNU asembler jest narzędziem stosowanym na wielu różnych platformach sprzętowych. Z tego powodu należy zawsze upewnić się, że dany rozdział lub właściwość kompilatora dotyczy typu procesora, na którym aktualnie pracujemy.

Kod źródłowy

Kod źródłowy jest zapisany w pliku tekstowym. Plik musi być zakończony znakiem nowej linii. Poszczególne elementy pliku są rozdzielone dowolną liczbą spacji i/lub tabulatorów. Zbiór elementów zakończony znakiem końca linii (enter lub średnik) jest *wyrażeniem* (nie mylić z *wyrażeniem arytmetycznym*). W zależności od zawartości wyrażenia mogą być różnie interpretowane przez kompilator. Elementami wyrażeń są:

1. *Dyrektywy* – ciągi liter rozpoczynające się kropką, np.:

   ```
   .text
   ```

   Dyrektywy są używane do sterowania procesem kompilacji kodu źródłowego, np. definiowania granic segmentów, umieszczania danych w pamięci, wyboru procesora docelowego itd.

2. *Symbole* – ciągi znaków zawierające wyłącznie litery, cyfry i znaki „_.$”. Symbole nie mogą rozpoczynać się cyfrą. Symbol składa się z *nazwy* oraz *atrybutów.* Jednym z ważniejszych atrybutów jest *wartość* symbolu, którą można jawnie ustalić jako wynik *wyrażenia*. Dzięki możliwości jawnego ustalenia wartości symbole mogą być używane przez programistę np. w celu uniknięcia wielokrotnego wpisywania stałych liczbowych, co wpływa pozytywnie na łatwość pielęgnacji i czytelność kodu:

   ```
   symbol_1 = 1


   mov  $symbol_1, %eax
   mov  $symbol_1, %ebx
   mov  $symbol_1, %ecx
   mov  $symbol_1, %edx
   mov  $symbol_1, %esi
   ```

   Specjalnym przypadkiem symboli są *etykiety.* Często używanym symbolem jest także „.”.

3. *Etykiety* są symbolami zakończonymi znakiem „:”. Najprościej mówiąc wartością etykiety jest adres, pod którym ta etykieta została zdefiniowana. Etykiet najczęściej używa się, aby uniknąć ręcznego obliczania adresów w programie:

   ```
   etykieta_1:
   ```

```
.....

jmp etykieta_1
```

4. *Wyrażenia arytmetyczne* są używane do określania wartości liczbowych (np. adresów). Elementami wyrażeń są argumenty i operatory. Argumentami mogą być symbole, liczby lub inne wyrażenia otoczone nawiasami okrągłymi. Operatory określają działania arytmetyczne wykonywane na argumentach. Wartość wyrażenia arytmetycznego musi być możliwa do obliczenia podczas kompilacji.
5. *Komentarze* są ciągami znaków ignorowanymi podczas kompilacji kodu.
6. *Stałe* są liczbami o o wartościach zapisanych w taki sposób, aby można je było określić jednoznacznie podczas kompilacji. Rozróżnia się stałe liczbowe oraz znakowe.

Literatura obowiązkowa

[1] IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture , http://www.zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/Intel%20Penium%20IV/IA-32%20Intel%20Architecture%20Software%20Developers%20Manual%20vol.%201%20-%20Basic%20Architecture.pdf

rozdział 3 – zawiera informacje o dostępnych rejestrach procesora i trybach adresowania – wiedza niezbędna, należy rozumieć np. czym się różni tryb adresowania bezpośredni od natychmiastowego i pośredniego. Zrozumienie informacji zawartych w rozdz. 3.7 (z pominięciem 3.7.4) jest absolutnie niezbędne.
rozdział 4 – przejrzeć w powiązaniu z rozdziałem 3
rozdziały 5 i 7 – zapoznać się z dostępnymi grupami instrukcji i operacjami możliwymi do wykonania za pomocą pojedynczych instrukcji oraz operacjami, których NIE MOŻNA wykonać jedną instrukcją. Znajomość wszystkich mnemoników wraz z dopuszczalnymi trybami adresowania nie jest konieczna, ale należy umieć szybko znaleźć potrzebną instrukcję oraz wydobyć niezbędne informacje z [3]

[2] „Using as", dokumentacja kompilatora as, http://sourceware.org/binutils/docs/as/

rozdział 3 – składnia pliku źródłowego programu asemblerowego, szczególną uwagę proszę zwrócić na wpływ białych znaków, sposób tworzenia komentarzy oraz stałych
rozdział 4 – sekcje programu, szczególną uwagę proszę zwrócić na różne typy sekcji zawierających dane
rozdział 7 – dyrektywy asemblera (nie mylić z instrukcjami, mnemonikami, poleceniami ani funkcjami), na początku najistotniejsze są dyrektywy pozwalające operować na obszarach pamięci (wypełniać je i/lub rezerwować) oraz dyrektywy umożliwiające stosowanie makr. Zalecam przejrzenie wszystkich dostępnych dyrektyw i korzystanie z nich w razie potrzeby.
Rozdział 9.15 – znajomość informacji zawartych w tym rozdziale jest NIEZBĘDNA do sprawnego tworzenia kodu. Zapoznając się z tym rozdziałem należy znaleźć odpowiedzi na pytania w rodzaju „jak adresować kolejne elementy tablicy", „jak przesłać pomiędzy pamięcią a rejestrem 1/2/4 bajty" itp. Szczególnie istotne w początkowym okresie nauki są rozdziały 9.15.3 i 9.15.8, choć zdecydowanie zalecam dokładne zapoznanie się ze wszystkimi rozdziałami z zakresu 9.15.2 – 9.15.8.

[3] IA-32 Intel® Architecture Software Developer's Manual Volume 2: Instruction Set Reference, http://www.zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/Intel%20Penium%20IV/IA-32%20Intel%20Architecture%20Software%20Developers%20Manual%20vol.%202%20-%20Instruction%20Set%20Reference.pdf

Tematy ćwiczeń

Ćwiczenie 0

Celem ćwiczenia jest zapoznanie uczestników kursu ze środowiskiem programistycznym używanym na laboratorium oraz z metodologią pracy z dokumentacją.

Umiejętności:

a) Użycie podstawowych poleceń systemu Linuks (ls, cd, mkdir, rm)
b) Korzystanie z podręcznika systemowego man ze szczególnym uwzględnieniem metod zdobycia informacji o funkcjach systemowych i bibliotecznych (strony 2 i 3 podręcznika)
a) Kompilacja źródeł asemblerowych za pomocą kompilatora as i linkera ld oraz kompilatora gcc, znajomość podstawowych opcji kompilatora i linkera (-o, -g, -e, -l, -v, -c)
b) Tworzenie prostych plików sterujących kompilacją z użyciem programu make
c) Podstawy tworzenia kodu asemblerowego – deklarowanie zmiennych, segmentów programu, wywoływanie funkcji systemowych, różnica pomiędzy adresem a wartością zmiennej (tryb adresowania natychmiastowy, bezpośredni i pośredni)

Przykładowe zadania:

a) przepisać i skompilować kod zamieszczony we wzorcowym sprawozdaniu
b) zmodyfikować kod do postaci pozwalającej wykorzystać funkcję systemową read.

Zwrócić szczególną uwagę na metody deklarowania zmiennych (obszarów pamięci używanych do przechowywania danych) oraz metody przekazywania argumentów i zwracania wyniku z funkcji systemowych. Zapoznać się z plikiem unistd.h zawierającym numery funkcji systemowych oraz ze stronami podręcznika systemowego man dotyczącymi używanych funkcji.

Sprawozdanie:

W sprawozdaniu powinny znaleźć się odpowiedzi na trzy pytania: jaki był zakres prac na laboratorium, w jaki sposób ćwiczenie zostało przeprowadzone oraz jakie wnioski wyciągnięto. Poprzez sposób przeprowadzenia ćwiczenia należy rozumieć zarówno sposób wykorzystania dostępnych narzędzi, jak i opis opracowanych algorytmów.

Jeśli sprawozdanie zawiera fragmenty innych prac (w tym także innych sprawozdań) bez wyraźnej adnotacji, traktowane jest to jako PLAGIAT z wszelkimi tego konsekwencjami. Nieodzownym elementem każdego sprawozdania jest spis literatury oraz odwołania do tejże 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.

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 jasno i czytelnie prezentujących zaobserwowane zjawiska. Wyjątkiem są kody źródłowe programów, które muszą być zamieszczone w całości.

Ćwiczenie 1

Celem ćwiczenia jest opanowanie umiejętności tworzenia prostych konstrukcji programowych (pętle, badanie warunków) z użyciem instrukcji asemblera.

Przykładowe zadania

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. 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.
3. To samo, co w punkcie 2, ale należy dodać kod wykrywający niepoprawne znaki we wczytanym ciągu.
4. Pobrać ze standardowego strumienia wejściowego ciąg bajtów o zadanej długości i wyprowadzić na standardowy strumień wyjściowy ciąg znaków reprezentujący szesnastkowy zapis wartości tej liczby.
5. Pobrać ze standardowego strumienia wejściowego ciąg znaków o dowolnej długości, potraktować ten ciąg jako zapis dziesiętny pewnej liczby i zapisać wartość tej liczby binarnie w pamięci.
6. Binarną reprezentację liczby z zadania 5 wypisać na standardowy strumień wyjściowy w postaci ciągu znaków reprezentujących wartość dziesiętną.

Ćwiczenie 2

Celem ćwiczenia jest utrwalenie umiejętności konstruowania prostych konstrukcji w asemblerze procesora x86.

Przykładowe zadania

1. Zestaw funkcji realizujących proste operacje arytmetyczne (dodawanie, odejmowanie, mnożenie) na wielkich liczbach w kodzie naturalnym binarnym i/lub U2. Przeprowadzane operacje powinny wykorzystywać wbudowane w procesor sprzętowe mechanizmy ułatwiające wykonywanie takich operacji, np. propagacja przeniesienia pomiędzy kolejnymi pozycjami. Poprzez wielkie liczby rozumie się liczby zawierające co najmniej kilka tysięcy bitów. Reprezentacja liczb powinna uwzględniać możliwe różnice w ich długości, tzn. należy unikać reprezentacji o stałej liczbie pozycji. Rozwiązaniem może być np. przechowywanie liczb w strukturze zawierającej liczbę zajmowanych pozycji i tablicę z wartościami poszczególnych cyfr.
2. Funkcja obliczająca największy wspólny dzielnik zgodnie z algorytmem Euklidesa dla liczb o dowolnej długości.
3. Funkcja wyznaczająca liczby pierwsze zgodnie z algorytmem sita Erastotenesa.

4. Funkcja sortująca tablicę liczb 64/96/128-bitowych.
5. Funkcja obliczająca kolejne liczby Fibonacciego dla argumentów o szerokości będącej całkowitą wielokrotnością szerokości słowa maszynowego.

Ćwiczenie 3

Celem ćwiczenia jest zapoznanie się z technikami pozwalającymi na użycie w tym samym projekcie różnych języków programowania oraz z podstawami optymalizacji (analiza wydajności kodu).

Przykładowe zadania

a) Napisać kod w asemblerze pozwalający używać funkcji napisanych w języku C i operujących na różnych argumentach stałoprzecinkowych i zmiennoprzecinkowych. Należy wziąć pod uwagę zarówno funkcje biblioteczne, jak i funkcje utworzone własnoręcznie.
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 zmniejszyć czas wykonania programu. Do pomiaru czasu wykonania programu należy wykorzystać zarówno odpowiednie narzędzia (np. gprof), jak i mechanizmy wbudowane w procesor (ang. Time Stamp Counter).
e) Wykonać pomiary wydajności kodu asemblerowego napisanego w trakcie poprzednich zajęć oraz jego nowej wersji przygotowanej w języku C. Porównać oba rozwiązania, zaproponować i zaimplementować optymalizacje pozwalające na zwiększenie wydajności.

Literatura obowiązkowa

[1] SYSTEM V APPLICATION BINARY INTERFACE Intel386 Architecture Processor Supplement Fourth Edition
http://www.zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/Application%20Binary%20Interface/System%20V%20Application%20Binary%20Interface%20Intel386%20Architecture%20Processor%20Supplement%20Fourth%20Edition.pdf

rozdział 3 – najistotniejsza z punktu widzenia ćwiczenia jest część zatytułowana Function Calling Sequence, gdzie należy odszukać informacje o przekazywaniu parametrów do funkcji i zwracaniu wartości. Proszę także zwrócić uwagę na tab. 3.1 (str. 28) zawierającą rozmiary podstawowych typów danych. Pomocne może być także sprawdzenie w dokumentacji procesora działania stosu (instrukcje push, pop, enter, leave, call, ret).

[2] IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture
http://www.zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/Intel

%20Penium%20IV/IA-32%20Intel%20Architecture%20Software%20Developers%20Manual%20vol.%201%20-%20Basic%20Architecture.pdf

rozdział 6 – najistotniejsze są części 6.1, 6.2 i 6.3 (bez wywoływania funkcji pomiędzy różnymi poziomami uprzywilejowania)

Literatura uzupełniająca

[1] Eric Youngdale, The ELF Object File Format: Introduction, http://www.linuxjournal.com/article/1059
[2] Eric Youngdale, The ELF Object File Format by Dissection, http://www.linuxjournal.com/article/1060

Kilka informacji o budowie plików z kodem wykonywalnym oraz o ich ładowaniu do pamięci i uruchamianiu

[3] Antoni Myłka, Profilowanie i wydajność: gprof i strace
http://www.g2inf.one.pl/referaty/.mylka/gprof/index.html

Krótkie wprowadzenie do profilowania kodu z użyciem gprof. Polecam także dokumentację programu gprof ze strony http://sourceware.org/binutils/docs/gprof/index.html

Ćwiczenie 4

Celem ćwiczenia jest zapoznanie się z jednostką zmiennoprzecinkową procesorów rodziny x86.

Przykładowe zadania

a) Zakodowanie zestawu funkcji pozwalających ustawić i sprawdzić status jednostki zmiennoprzecinkowej (np.: precyzję obliczeń, tryb zaokrąglania, wystąpienie wyjątków itp.). Funkcje powinny mieć interfejs pozwalający na ich wywołanie z poziomu języka C.
b) Opracowanie zestawu funkcji w języku C lub asembler obliczających wartości funkcji opisanych zadanym wzorem. Przykładami wzorów mogą być rozwinięcia funkcji trygonometrycznych (sin, cos) w szereg Taylora, metody obliczania całek oznaczonych itp. Ważnym elementem ćwiczenia jest takie dobranie kolejności i liczby wykonywanych operacji zmiennoprzecinkowych, aby uzyskać wynik mieszczący się w zadanej precyzji obliczeń .

Literatura obowiązkowa

[1] IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture , http://www.zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/Intel%20Penium%20IV/IA-32%20Intel%20Architecture%20Software%20Developers%20Manual%20vol.%201%20-%20Basic%20Architecture.pdf

rozdział 8 – szczególną uwagę proszę zwrócić na typy danych, zakresy reprezentowanych wartości, możliwości zmiany parametrów jednostki i dostępne operacje możliwe do wykonania jednym rozkazem.

[2] What Every Computer Scientist Should Know About Floating-Point Arithmetic, by David Goldberg, published in the March, 1991 issue of Computing Surveys., https://ece.uwaterloo.ca/~dwharder/NumericalAnalysis/02Numerics/Double/paper.pdf lub http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Można traktować jako przypomnienie wiadomości z wykładu. Oczywiście wszelkie notatki i pozostałe materiały podane na wykładzie są także bardzo pomocne. Należy przeczytać, jeśli np. macie Państwo problemy z określeniem dopuszczalnych zakresów liczb reprezentowalnych w postaci znormalizowanej i zdenormalizowanej dla zadanej precyzji.

[3] Software Optimization Guide for AMD64 Processors, http://www.zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/AMD64/Software%20Optimization%20Guide%20for%20AMD64%20Processors.pdf

Istotny jest rozdz. 10.2 zatytułowany Achieving Two Floating-Point Operations per Clock Cycle.

Ćwiczenie 5

Celem ćwiczenia jest zapoznanie się z jednostkami wektorowymi rodziny x86.

Przykładowe zadania

a) Zaimplementować operator splotowy lub całkowy (mnożenie macierzy, filtry cyfrowe, numeryczne całkowanie/różniczkowanie itp.) z użyciem rozszerzeń wektorowych oraz bez ich użycia. Porównać wydajność implementacji oraz ich cechy – łatwość implementacji, narzuty wersji wektorowej, wymagania odnośnie rozłożenia danych w pamięci

Literatura obowiązkowa

[1] IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture , http://www.zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/Intel%20Penium%20IV/IA-32%20Intel%20Architecture%20Software%20Developers%20Manual%20vol.%201%20-%20Basic%20Architecture.pdf

rozdziały 9, 10 i 11 zawierające opis jednostek MMX, SSE i SSE2.

Ćwiczenie 6

Celem ćwiczenia jest zapoznanie się ze sposobem działania pamięci podręcznej.

Przykładowe zadania

a) Napisać program pokazujący trafienia/chybienia w pamięci podręcznej na podstawie obserwacji czasu wykonania poszczególnych instrukcji (np. MOV).
b) Napisać program pokazujący brak chybień podczas dostępu do bloków danych o rozmiarze mniejszym od rozmiaru pamięci podręcznej.
c) Napisać program pokazujący brak chybień podczas dostępu do danych z tej samej linii pamięci podręcznej.
d) Napisać program pokazujący migotanie podczas dostępu do wielodrożnej pamięci podręcznej.

Literatura obowiązkowa

Uwaga – poniższe pozycje zawierają mnóstwo szczegółów technicznych, które należy potraktować jedynie jako uzupełnienie materiału zaprezentowanego na wykładzie. Nie należy próbować zapamiętać wszystkich niuansów, wystarczy ogólne zrozumienie idei działania i ogólna wiedza o dodatkowych mechanizmach.

Uwaga – wiele przydatnych, praktycznych informacji dotyczących pisania wydajnego kodu można znaleźć na stronie Agnera Foga http://www.agner.org/optimize/

[1] IA-32 Intel® Architecture Software Developer's Manual Volume 3: System Programming Guide, http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/Intel%20Penium%20IV/IA-32%20Intel%20Architecture%20Software%20Developers%20Manual%20vol.%203%20-%20System%20Programming%20Guide.pdf

rozdział 10.

Podczas lektury rozdziału należy się skupić jedynie na zrozumieniu ogólnej idei działania pamięci podręcznej. Większości opisanych mechanizmów nie można skonfigurować z poziomu programu na niskim poziomie uprzywilejowania (z poziomu programu użytkownika).

[2] AMD64 Architecture Programmers Manual vol. 1 - Application Programming, http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/AMD64/AMD64%20Architecture%20Programmers%20Manual%20vol.%201%20-%20Application%20Programming.pdf

rozdział 3.9 zawiera uproszczony opis działania pamięci podręcznej.

[3] AMD64 Architecture Programmers Manual vol. 2 - System Programming, http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/AMD64/AMD64%20Architecture%20Programmers%20Manual%20vol.%202%20-%20System%20Programming.pdf

rozdział 7 zawiera opis mechanizmu pamięci podręcznej dostępny w procesorach używanych w laboratorium. Rozdział ten należy czytać podobnie jak [1].

[4] AMD Athlon 64 X2 Dual-Core Processor Data Sheet, http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/AMD64/AMD%20Athlon%2064%20X2%20Dual-Core%20Processor%20Data%20Sheet.pdf

dokument zawiera dane techniczne procesorów używanych w laboratorium.

Literatura uzupełniająca

[1] Software Optimization Guide for AMD64 Processors, http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/AMD64/Software%20Optimization%20Guide%20for%20AMD64%20Processors.pdf

rozdział 5 zawiera zalecane techniki pisania wydajnych kodów dla procesorów używanych w laboratorium.