

Sprawozdanie laboratorium OiAK

Maciej Myśków 272794

Termin oddania: 26.03.2024

Zadanie 1: Kalkulator dużych liczb

Politechnika Wrocławska Filia w Jeleniej Górze II

Rok grupa 3 godzina:

17:05

Spis treści:

1. Cel i zakres
2. Sposób wykonania
3. Opis struktury kodu oraz rejestrów
4. Przedstawienie algorytmów oraz błędów
5. Podsumowanie oraz to czego nie udało się osiągnąć

1. Cel i zakres

Celem ćwiczenia było napisanie w języku assembler kalkulatora, który wykonuje obliczenia (dodawanie, odejmowanie oraz mnożenie) na dużych liczbach szesnastkowych. Program ma dać użytkownikowi możliwość komunikacji przez terminal oraz otrzymanie wyniku wybranej operacji przy wybranych parametrach.

Ćwiczenie obejmuje setup środowiska, napisanie kodu oraz przetestowanie poprawności działania programu dla wybranych danych testowych. Każdy krok zostanie omówiony w niniejszym sprawozdaniu.

2. Sposób wykonania

Wybrane środowisko pracy to system Linux Ubuntu wraz z zainstalowanymi niezbędnymi dodatkami do pracy w języku Asembler. Posłużono się edytorem kodu Visual Studio Code.

Kod programu napisany jest w architekturze x86 z rejestrami w wersji 32-bitowej, zgodnie z zaleceniami w treści zadania laboratoryjnego. Wybrana składnia języka Asembler to składnia Intel. W kodzie programu nie wykorzystano żadnej z dozwolonych bibliotek C. Kompilacja programu jest wykonywana za pomocą narzędzia NASM oraz linkera ld. Składnia poleceń niezbędnych do kompilacji i uruchomienia programu to:

- `nasm -f elf32 kalkulator.asm`
- `ld -m elf_i386 -s -o kalkulator kalkulator.o`
- `./kalkulator`

Do weryfikacji pracy oraz jako sposób dostarczenia projektu posłużono się systemem kontroli wersji Git oraz platformą Github.

3. Opis struktury kodu oraz rejestrów

W kodzie występują następujące sekcje/etykiety:

- `bss`
- `text`
- `global _start`

W sekcji `bss` przetrzymywane są **nie**-zainicjalizowane zmienne, których wartość nie jest z góry określona, natomiast konieczne jest podanie rezerwacji miejsca w bajtach na każdą zmienną w pamięci programu. W kodzie wykorzystano tę sekcję do rezerwacji miejsca na dwa

operandy (100 bajtów), wynik (101 bajtów = 100 + jeden na przeniesienie/przepiętnienie) oraz operacje arytmetyczną (2 bajty)

Sekcja tekst służy do zakomunikowania kompilatorowi, że od tego miejsca w kodzie rozpoczyna się wykonywanie operacji / rozkazów procesora.

Sekcja global start wraz z etykietą _start, komunikuje linkerowi, że od tego miejsca ma powiązać główny plik wykonawczy kodu, który zostanie wygenerowany podczas kompilacji.

W kodzie zostały wykorzystane następujące rejestry ogólnego przeznaczenia: Rejestry

32-bitowe

- EAX
- EBX
- ECX
- EDX
- ESI
- EDI

Rejestry 8-bitowe:

- AL
- BL oraz jeden 16-bitowy: AX

4. Przedstawienie algorytmów oraz błędów

Wszystkie algorytmy w kodzie zostały napisane najpierw w języku wysokopoziomowym C++ a następnie została dokonana próba przepisania kodu na język maszynowy Asembler. Niestety kod posiada kilka błędów i nie działa w pełni z założeniami w zadaniu do wykonania. Algorytmy odzwierciedlają zwykłe działanie w słupku.

Dodawanie:

Pierwszym omawianym algorytmem będzie algorytm dodawania. W kodzie następuje przeniesienie wartości podanych przez użytkownika do rejestrów al oraz bl, co umożliwia wykonywanie operacji arytmetycznych. Wynik dodawania jest przetrzymywany w rejestrze al. Następnie następuje sprawdzenie, czy nie osiągnięto limitu sumy (dla systemu hex jest to 15), jeśli tak to dokonujemy przejścia do poprzedniego bitu (od końca) i uwzględniamy przeniesienie.

Na koniec zostaje wywołana funkcja zapisu wyniku częściowego do zarezerwowanej zmiennej wynik poprzez przeniesienie wartości z rejestru al do wyniku na stosownej pozycji. W trakcie dokonywana jest konwersja na zapis szesnastkowy w terminalu dodaniem do wyniku wartości 7h, co oznacza odpowiednie przekonwertowanie kodu ASCII.

Odejmowanie:

Algorytm odejmowania jest analogiczny z tym, że wymagana jest dodatkowa funkcja o nazwie pożyczka, która dodaje do wyniku wartość 16 (szesnastkowy) aby uniknąć ujemnych liczb a następnie jest to uwzględniane w poprzedniej starszej pozycji bajtu. Funkcja pożyczka jest wywoływana przy sprawdzeniu warunku, czy pierwsza cyfra jest mniejsza od drugiej.

```
cmp al, bl
jl pozyczka ;jesli cyfra 1 mniejsza - pozyczka

sub al, bl
add bl, 7h
add al, '0'
jmp funkcja_zapisz_wynik_odejmowanie
```

Niestety w obydwu przypadkach (dodawania i odejmowania) mimo usilnych prób algorytm nie działa poprawnie. Na wyjście w terminalu zwracane są znaki ASCII zamiast wartości liczbowej. **Prawdopodobnie spowodowane jest to złą konwersją na system szesnastkowy, gdyż przy testowaniu tego samego algorytmu dla systemu dziesiętnego program zwracał poprawne wartości liczbowe.** Za konwersję ASCII

odpowiada np. ta linijka: `add al, 7h`. To co zostaje zwrócone po wykonaniu programu:

```
maciej@DESKTOP-05KKE8:~/laby$ nasm -f elf32 kalkulator.s
maciej@DESKTOP-05KKE8:~/laby$ ld -m elf_i386 -s -o kalkulator kalkulator.o
maciej@DESKTOP-05KKE8:~/laby$ ./kalkulator
5c55bc538d3af4b6436e2c7841277a8ef4a1bd6d
+
4945aa6db0bf9edee897a2b9b860ca0621e7189b
@@@A@+B++++$+e+BF+++++o@?>+?+<?+@F+++++maciej@DESKTOP-05KKE8:
~/laby$
```

Mnożenie:

Mnożenie sprawiło mi najwięcej problemów szczególnie tych od strony algorytmicznej, jednakże po wielu podejściach udało się rozwiązać problem.

W algorytmie mnożenia zastosowano dwie pętle – dla liczby 1 oraz liczby 2. Na początku pętli dla liczby 1 ładowany jest najmłodszy bit liczby 1 do rejestru bl. Następnie mnożymy bit liczby1, który znajduje się w rejestrze al przez bit liczby2, który ładowany jest z rejestru bl. Dokonujemy tego za pomocą dyrektywy mul bl, którą akceptuje NASM a wynik (bit) trzymany jest w rejestrze ax – specyfika działania mnożenia w NASM. W kolejnym kroku przesuwamy odpowiednio zmienną wynik, tak aby dodać wynik częściowy na odpowiednią pozycję. Sprawdzam czy po dodaniu trzeba będzie uwzględnić przeniesienie (wartość graniczna dla hex: 15). Jeśli tak, to odejmujemy od wyniku 16 i zapisujemy nowy wynik inkrementując następną pozycję o 1 – tak jak w przypadku algorytmu na dodawanie. Następnie inkrementujemy liczbę 1 i sprawdzamy czy osiągnięto koniec liczby 1 – jeśli nie to powtarzamy powyższe kroki cyklicznie do momentu zakończenia liczby 1. Analogicznie postępujemy z liczbą 2. Przed przystąpieniem do kolejnego cyklu zerujemy rejestr z przeniesieniem oraz wczytujemy kolejny bit liczby do rejestrów esi oraz edi.

Niestety ten algorytm również nie podaje poprawnego wyniku hexadecymalnego po

uwzględnieniu korekty:

```
add al, 7h  
add al, '0'
```

Również nie udało się pobierać danych od użytkownika w sposób zdefiniowany w treści zadania. Mianowicie użyto wczytania danych jedno pod drugim każde pobranie i wczytanie do pamięci zakończone enterem.

```
_start:
    mov eax, SYS_READ
    mov ebx, STDIN
    mov ecx, liczba1
    mov edx, 100
    int 0x80

    mov eax, SYS_READ
    mov ebx, STDIN
    mov ecx, operacja
    mov edx, 2
    int 0x80

    mov eax, SYS_READ
    mov ebx, STDIN
    mov ecx, liczba2
    mov edx, 100
    int 0x80
```

5. Podsumowanie oraz to czego nie udało się osiągnąć

Mimo kilku niedogodności oraz błędów związanych z działaniem programu udało się zrealizować większą część sprecyzowanych w poleceniu zadań programu. Program kompiluje się, nie zwraca żadnych błędów przy wywołaniu oraz udało się zaimplementować przynajmniej w jakiejś części wszystkie algorytmy.

Natomiast to czego się nie udało zrobić lub działa nie poprawnie, to chociażby sposób komunikacji programu z użytkownikiem. Program miał pobierać parametry w ten sposób: `./kalkulator [liczba1] [operacja] [liczba2]`, natomiast mój program pobiera parametry przez wpisanie ich w terminal w odpowiedniej kolejności. Rozwiązaniem mogłoby być zastosowanie pewnych przesunięć rejestrów esi oraz edi (po 4 na każdy parametr) natomiast to również nie zadziałało.

Kolejnym problemem, który został wspomniany wcześniej przy omawianiu algorytmów, jest sposób konwersji wyników na format szesnastkowy. W programie zastosowano dodanie do rejestru wyniku wartości 7h oraz odpowiednie działanie w słupkach charakterystyczne dla systemu hex. Natomiast powyższy zabieg generuje znaki ASCII na terminalu zamiast poprawnego wyniku. Podczas testu problem ten nie występuje, gdy obliczenia były nie konwertowane na system szesnastkowy (w dziesiętnym liczył poprawnie).

Ostatnim aspektem było zaimplementowanie logiki mnożenia. Spowodowało to wiele problemów ale udało się zaimplementować znaczną część logiki. Mianowicie dokonano mnożenia w pętli bit po bicie oraz zapisu do wyniku iloczynu częściowego. Udało się również po wielu próbach napisać algorytm dodawania iloczynów częściowych – tak jak robimy to w słupku.