Jeroen/Jean Update README.md | Latest commit 112aea1 on Nov 14, 2022 | History

1 contributor

220 lines (108 sloc) | 12.3 KB | Raw | Blame

Intro in Assembly

Assembly is vrijwel de laatste verstaanbare programmeertaal die er bestaat. Alle programmeertalen worden immers eerst door hun eigen compiler omgezet naar assembly en vervolgens naar machine code. Hoewel het niet noodzakelijk is om volwaardig te kunnen programmeren in assembly kennen de opbedding, is het wel handig om een basiskennis van assembly te hebben. Het analyseren van viruses komt er immers vaak op neer dat we het analyseren van de assembly code van dat virus om zo te achterhalen hoe het virus in interactie gaat met je systeem. Daarnaast maakt een basiskennis van assembly ook meteen duidelijk hoe een CPU gebruik maakt van registers en ALU.

Omgeving opzetten

Voor het programmeren van onze assembly oefeningen, zijn er verschillende mogelijkheden. Er zijn vele online omgevingen waar je assembly kan schrijven en testen. Echter deze laten niet toe om gebruik te maken van hulpfuncties dewelke wij aanbieden. Om dit goed te faciliteren, gaan we gebruik maken van Unix. Dit is een server die je kan gebruiken om je oefeningen in te programmeren en te testen. Volgende stappen zorgen ervoor dat je assembly omgeving correct is opgezet.

- Log via SSH in op ssa.ucllabs.be op poort 22345 met je nummer.
- Maak een nieuwe directory aan
- Kopieer de inhoud van /tmp/Compsys_Assembler in de directory die je net hebt aangemaakt

Dit zou volgende bestanden moeten bevatten:

```
jeroen@Laptop-ux11:~/Documents/UCLL/Compsys/test/Compsys_Assembler$ ls -l
total 48
-rw-r--r-- 1 jeroen jeroen  1017 Nov  4 22:26 gt.asm
-rw-r--r-- 1 jeroen jeroen 12815 Nov  4 22:26 gtins.asm
-rw-r--r-- 1 jeroen jeroen   505 Nov  4 22:26 nsamm.asm
-rw-r--r-- 1 jeroen jeroen   495 Nov  4 22:26 test.asm
-rw-r--r-- 1 jeroen jeroen   505 Nov  4 22:26 test.in
-rw-r--r-- 1 jeroen jeroen   213 Nov  4 22:26 test.ut
-rwxrwxr-x 1 jeroen jeroen    38 Nov  4 22:26 vertaal
-rwxrwxr-x 1 jeroen jeroen    54 Nov  4 22:26 voerut
```

Zoals je kan zien zijn er reeds enkele x.asm bestanden. Hiervoor later in het labo meer uitleg. Het gemakkelijkste gaat zijn, als je al je oefeningen in deze folder maakt.

Assembleren en uitvoeren

Alvorens we een programma kunnen uitvoeren, moet de geschreven assembly code omgezet worden naar machine code. Doordat wij gebruik mogen maken mnemotechnische functie code en symbolische adressen, moeten deze eerst correct worden omgezet naar hexadecimale code. Dit laatste is immers het enigste wat een CPU verstaat (eigenlijk binair, maar hexadecimaal is gemakkelijker om te lezen). Omzetten van de mnemotechnische functie code naar machine code en symbolische adressen correct berekenen om gebruik te kunnen maken van het geheugen, gebeurt door een assembleerprogramma of assembler.

Om dit vlot te laten verlopen, hebben wij een bash script geschreven dat dit voor jullie zal doen:

```
$ ./vertaal vraag1
```

Het voltooit dus om `vertaal` op te roepen gevolgd door de filename zonder extensie. Dit zal een *.o bestand (object bestand) maken van je programma.

Dit bestand kunnen we echter nog niet uitvoeren. Omdat we ook gebruik maken van functie's die in een andere programma staan geschreven (gtins.asm en gt.asm) alsook subroutines van het OS, moeten deze nog op een correcte manier bij elkaar worden samengevoegd en gecompileerd worden tot 1 uitvoerbaar bestand. Ook hier hebben wij een bash script voor geschreven dat dit voor jullie doet:

```
$ ./voerut vraag1
graag een getal tussen -2147483648 en 2147483647 : 12
48
```

❗ maak gebruik van `vertaal` om object bestanden aan te maken van **gtins.asm** en **gt.asm**.

Invoer en uitvoer

Voor het oplossen van de oefeningen, ga je regelmatig input moeten vragen aan de eindgebruiker. Echter het vragen van input of output genereren in assembly is niet gedaan met 1 enkele print code. Daarnaast moet je ook al grondige kennis hebben van assembly om dit tot een goed einde te brengen. Omdat dit vooltij het doel van het CPO gaat, mag je gebruik maken van de wij hebben voorzien.

Voor het vragen vragen van input maak je gebruik van `[in(test)]`. Dit zorgt ervoor dat er aan de eindgebruiker een getal gevraagd wordt en opgeslagen wordt in de variabele `test`. Je mag `[in(variabele)]` ook enkel gebruiken met een variabelen. Rechtstreeks invoer naar een register is niet mogelijk.

Voor het tonen van een getal mag je gebruik maken van `uit(test)`. Dit zorgt ervoor dat de waarde van de variabele test aan de eindgebruiker wordt getoond. Je mag `uit(variabele)` ook enkel gebruiken met een variabelen. Rechtstreeks de inhoud van een register aan de eindgebruiker laten zien is niet mogelijk.

Code blocks

Het schrijven van een assembly programma moet steeds voldoen aan een vaste structuur alsook steeds de juist extensie hebben. Je bestandsnaam moet dus steeds onder de vorm zijn van `cnaam.ask`. De inhoud van je bestand bestaat uit 3 secties die al dan niet verplicht zijn:

- data sectie (niet verplicht)
- bss sectie (niet verplicht)
- text sectie (verplicht)

Een voorbeeld programma zou er zo uit kunnen zien:

```
%include "gt.asm"
section .data
vier: dd 1
miljard: dd 1000000000
viere: dd 2000000000

section .bss
help: resd 1

section .text
global _start
_start:
    mov ebp, esp; for correct debugging
    ;write your code here
    mov eax, [vier]
    add eax, [viere]
    mov eax, [viere]
    imul dword [vier]
    mov [help], edx
    uit(help)

    mov eax, 1
    int 0x80
```

Data sectie

De data sectie wordt gebruikt voor het declareren van geïnitialiseerde variabelen en constanten. Dit geven we aan door:

```
section .data
```

BSS sectie

De bss sectie wordt gebruik voor het declareren van niet geïnitialiseerde variabelen. Alle variabelen zoals in volgende paragraaf vermeld staat, worden aanvaard. Dit geven we aan door:

```
section .bss
```

Text sectie

In deze sectie gaan we onze eigenlijke code schrijven. Deze moet verplichtend starten met global CMAIN. Dit geeft aan de CPU weer waar het eigenlijke uitvoeren van het programma moet beginnen. Alhanktelijk van welke compiler je gebruik, kan dit ook global _start zijn. Voor de linux omgeving waar wij in werken zal dit steeds `_start` zijn.

```
section .text
global _start
_start:
```

Geheugen

In zowel de data als de .bss sectie is het mogelijk om bepaalde delen van het geheugen te reserveren om hier later gebruik van te maken in het programma. Dit noemen we doorgaans een variabele. De manier waarop we een variabele definiëren hangt of we deze reeds een waarde gaan meegeven bij het definiëren of niet.

Geïnitialiseerde data

Bij geïnitialiseerde data gaan wat we de hand van een variabele een stukje geheugen reserveren dat tijdens het uitvoeren van het programma kan gebruikt worden. Deze variabele gaat reeds een initiële waarde meekrijgen tijdens het declareren. Volgende mogelijkheden zijn ter onze beschikking.

Directive	Doel	Gebruikt geheugen
DB	Definiëer Byte	alloceer 1 byte
DW	Definiëer Word	alloceer 2 bytes
DD	Definiëer Dubbelwoord	alloceer 4 bytes
DQ	Definiëer Quadwoord	alloceer 8 bytes
DT	Definiëer Tien Bytes	alloceer 10 bytes

```
section .data
    choice: DB 'y'
    number: DW 12345
    neg_number: DW -12345
    number2: DD 2345abh
```

Niet geïnitialiseerde data

Bij niet geïnitialiseerde data gaan we aan de van een variabele een stukje geheugen reserveren zonder deze te initialiseren (gaan startwaarde geven). Dit doen we in de .bss sectie. Hiervoor kan men gebruik maken van ondenstaande mogelijkheden.

Directive	Doel
RESB	Reserveer 1 byte
RESW	Reserveer 1 woord
RESD	Reserveer 1 dubbelwoord
RESQ	Reserveer 1 quadwoord
REST	Reserveer 10 bytes

```
section .bss
    sign: RESB 1
    number: RESW 1
    big_number: RESD 1
```

Constanten

Constanten zijn een vorm van variabelen die we gaan initialiseren en nadien nooit aanpassen. Deze gaan we definiëren in de .data sectie. Hiervoor maken we gebruik van `equ`. Eigenlijk mogen we dit niet echt kaderen binnen geheugen. Bij assembly is het namelijk zo dat een constante die gedefinieerd is met EQU, geen vaste plaats in het geheugen zal innemen. Telkens wanneer de assembler deze constante tegenkomt in de code zoals het assemblieren, zal hij deze letterlijk vervangen door de waarde die aangegeven is in de .data sectie met EQU.

```
section .data
    star: EQU '*'
    number: EQU 5128
    big_number: EQU -2423523423
    ...
mov eax, star
mov eax, number
mov eax, big_number
```

Basisbewerkingen

Binnen deze cursus gaan we enkel gebruik maken van een beperkte instructieset binnen assembly: MOV, ADD, SUB, IMUL en IDIV. Deze volstaan om een duidelijk beeld te krijgen van een lagere programmeertaal en hoe hier mee om te gaan.

MOV

De MOV instructie kopieert de data van de tweede operand naar de locatie van de eerste operand. Volgende mov mogelijkheden zijn beschikbaar.

- `mov reg,reg`;
- `mov reg,[mem]`;
- `mov [mem],reg`;
- `mov reg,constt`;
- `mov [mem],[const]`;

Het is meteen duidelijk dat het niet mogelijk is om via het MOV bevel van de ene variabele naar de andere variabele rechtstreeks te kopiëren. Dit kan men enkel bekomen door de waarde van de eerste variabele naar een register te kopiëren en vervolgens de inhoud van het register naar de andere variabele te kopiëren.

ADD

De ADD instructie gaat twee operanden met elkaar optellen en het resultaat in de eerste operand opslaan. Volgende ADD instructies zijn mogelijk:

- `add reg,reg`;
- `add reg,[mem]`;
- `add reg,[const]`;

SUB

De SUB instructie gaat in de eerste operand het resultaat opslaan van de eerste operand - de tweede operand. Volgende SUB instructies zijn mogelijk:

- `sub reg,reg`;
- `sub reg,[mem]`;
- `sub reg,[const]`;

IMUL

Om te vermenigvuldigen hebben we een dubbele accumulator nodig. Dit komt doordat het resultaat van een vermenigvuldiging dubbel zo lang kan zijn als elk van de factoren. Vanaf de i386 kungneren registers EDX (ROX) en EAX (RAX) als dubbele accumulator. Zoals EAX is EDX een register in de CPU dat uit 32 bits bestaat. EDX kan ook gebruikt worden als accumulator.

```
getal1: DD 7
getal2: DD 3
...
mov eax,[getal1]
imul dword [getal2]
```

❗ Let op: met `imul dword` kan je geen gebruik maken van constanten en moet je met een geïnitialiseerde variabele werken. ❗

IDIV

De deling is de omgekeerde bewerking van de vermenigvuldiging, en verloopt ook via (EDX,EAX). Het deeltal moet zich bevinden in het registerpaar. De deling levert 2 resultaten op: rest en quotiënt. Het quotiënt komt in EAX, de rest in EDX.

```
...
mul: DD 8
deeltal: DD 85
deler: DD 3
...
mov edx,[mul]
mov eax,[deeltal]
idiv dword [deler]
```

❗ Let op: met `idiv dword` kan je geen gebruik maken van constanten en moet je met een geïnitialiseerde variabele werken. ❗

Oefeningen

Nu je omgeving correct is opgezet en je een programma kan assembleren en compileren, is het tijd om zelf aan de slag te gaan.

Maak volgende programma's in assembly:

1. Schrijf een programma dat aan de gebruiker een getal vraagt, dit getal leest en het vierdubbel ervan toont op het scherm. **Je mag enkel gebruik maken van ADD**
2. Schrijf een programma dat aan de gebruiker 3 getallen vraagt en de som ervan toont op het scherm.
3. Schrijf een programma dat aan de gebruiker een getal vraagt, dit getal leest en het 19-voud ervan toont op het scherm. Je kan het getal 18 keer bij zichzelf bijtellen met 18 add-bevelen. Even denken, het kan ook met veel minder bevelen! **Je mag enkel gebruik maken van ADD**
4. Schrijf een programma dat drie getallen vraagt (x, y en z) en dat dan de waarde van 2x + 4y + 8z toont. **Je mag enkel gebruik maken van ADD**
5. Schrijf een programma dat de waarde van B^2-4AC toont. Eerst worden de waarden van A, B, C ingelezen.
6. Schrijf een programma dat een getal leest, A, en de waarde van A^7 toont. (Hoe groot kunnen onze getallen zijn? Als A7 kan, hoe groot kan A dan zijn?)
7. Definieer constanten met de waarden 3, 12, 1583 en 420. Definieer geïnitialiseerde variabelen met volgende waarden: 274 en 11. Gebruik voor de opstelling en de afhandeling EDX als accumulator. Schrijf een programma dat:
   - 3 + 12 berekent en het resultaat toont;
   - 3 - 12 berekent en het resultaat toont;
   - Quotiënt en rest berekent van de deling 1583 / 274 en deze resultaten toont;
   - 420 * 11 berekent en het resultaat toont.

Give feedback