Linux Assembly

The choice of a GNU generation

Linux Assembly

Some brighteyed and crazy, some frightened and lost.

Roger Waters

document versie: 1.0, 02-03-2002
auteur: H.P. Berends
layout: pdfLaTeX, Initworks - www.initworks.com

Voorwoord

Dit dictaat is geschreven voor studenten met enige programmeerervaring, maar met weinig tot geen ervaring met assembly. Aan de hand van voorbeelden worden de beginselen van de taal gedemonstreerd; er wordt dieper ingegaan op de werking van methodes achter de assembly taal.

In het bijzonder wil ik Remko de Vrijer <*devrijer@initworks.com*> en Marco Hazebroek <*m.hazebroek@hro.nl*> bedanken voor hun medewerking bij het maken van dit dictaat. Beiden hebben het complete manuscript verschillende malen doorgenomen. Hun reviews en visie zijn als uitgangspunt gebruikt bij het schrijven van dit document.

Ook wil ik de volgende mensen bedanken voor hun bijdrage aan dit dictaat.
A.J. Schrijver <*arjan@cimdroom.nl*>, J. van Iersel <*jurrian@cimdroom.nl*>, R.A.T.M.B.M. van Valkenburg <*robrecht@qualitysnooker.com*>, B. Mels <*bart.mels@cmg.nl*>, M.J. Bethlehem <*mjb@radon.nl*>, A. Veenstra <*veenstra@chello.nl*>, R. Huizer <*ronald@grafix.nl*>, M. Nuyten <*pnuyten@concepts.nl*>.

Inhoudsopgave

Hoofdstuk 1

Inleiding

Door een betere kennis van de werking van computers, kan er productiever geprogrammeerd worden in hogere programmeertalen zoals C en C++. Het leren programmeren in assembly is een goede manier om dit te bereiken.

De doelstelling van dit dictaat is een introductie geven in assembly en de lezer assembly te leren en het te laten schrijven. Assembly is een low-level taal en geeft een goed beeld van de taken van het operating system.
Als platform is gekozen voor het GNU/Linux operating system; dit is een opensource UNIX systeem met goede debug mogelijkheden en beschikbare kernel sourcecode. Zo kan Linux goed worden gebruikt bij de demonstratie van de werking van software.

Dit dictaat zal ook veel ingaan op de Intel processor architectuur omdat kennis van de werking van de processor van groot belang is bij het programmeren in assembly.

Dit document is bedoeld als een inleiding in assembly, het is namelijk niet mogelijk om alle aspecten van assembly te behandelen. Na het lezen van dit dictaat heeft men een completer beeld van wat assembly is en kan men technische documenten over processorarchitectuur in combinatie met assembly beter aan.

1.1 Opbouw dictaat

Hoofdstuk 2 '**Ontstaan van assembly**' beschrijft de historie van de computer en het ontstaan van assembly.

In hoofdstuk 3 '**Assembly**' wordt de basis van het assembly programmeren behandeld. Hierbij wordt een inleiding gegeven in het programmeren met system calls in assembly.

Hoofdstuk 4 '**Architectuur**' beschrijft de architectuur van een computer. Kennis hiervan is nodig, omdat assembly programma's direct omgaan met hardware.

Hoofdstuk 5 '**Registers**' gaat in op registers binnen de processor. Kennis hiervan is belangrijk omdat instructies en registers nauw verbonden zijn. Zonder gebruik te maken van registers is het onmogelijk om in assembly te programmeren.

Hoofdstuk 6 '**Taal niveaus**' licht hogere programmeertalen en assembly toe. Het beschrijft hoe er geprogrammeerd kan worden in verschillende talen en gaat in op de samenwerking.

Hoofdstuk 7 '**Programma structuur**' beschrijft hoe assembly programma's structuur krijgen.

In het dictaat wordt gebruik gemaakt van *NASM* en *ALD*, voor uitleg over de installatie hiervan zie bijlage A en B.

Hoofdstuk 2

Ontstaan van assembly

Het is 1935. Duitsland wordt geregeerd door de bende van Hitler, de wereld staat aan de vooravond van een ramp, als Konrad *Zuse* zijn ouders uit de woonkamer zet. Zuse, een jonge, wat wereldvreemde ingenieur met achterovergekamd zwart haar, die als statisticus werkzaam is bij de Henschel-Flugzeug-Werke, heeft de ruimte nodig voor 'de constructie van het eerste mechanische brein'. In die kamer in de Berlijnse wijk Kreuzberg wil Zuse een machine bouwen die in staat is om gecompliceerde statistische berekeningen uit te voeren. Dan is hij daar op zijn werk tenminste van verlost. "Ik ben te lui om te rekenen", had Zuse vaak bekend. Bovendien, vindt hij, moet zijn apparaat kunnen schaken en het liefst ook tekenen. [1]

2.1 De eerste computers

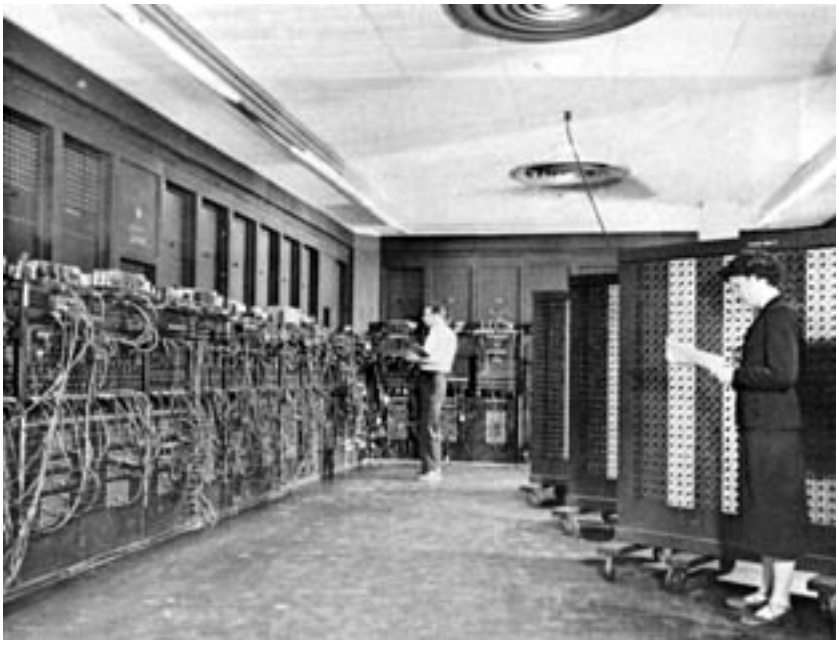
Zuse [2] was de bedenker van de eerste binaire computer. Het apparaat was een wangedrocht van tandraderen, uitsteeksels en zwengels. Hij ontwierp zijn machines vóór de oorlog; pas in 1946 bouwden Mauchly en Eckert, in opdracht van het Amerikaanse ministerie van Defensie, de *ENIAC*. De ENIAC was de eerste elektrische computer, een groot monster van 19.000 radiobuizen.

Tot die tijd waren de Amerikanen niet op het idee gekomen om te werken met binaire getallenstelsels, ze werkten met decimalen. Zuse was zijn tijd ver vooruit. Hij is er nooit rijk van geworden en zijn uitvinding heeft relatief weinig invloed gehad op de geschiedenis van de computer. Toch komt hem alle eer toe.

De onderdelen van computers en de gebruikte technieken komen niet alleen uit

[1] bron: Rotterdams Dagblad, Zaterdag 21 juli 2001

[2] Voor meer informatie over de uitvindingen van Zuse en de machines die hij heeft gebouwd zie http://irb.cs.tu-berlin.de/~zuse/hc.html.



Figuur 2.1: *De ENIAC; computers van vroeger vulden hele ruimtes*

deze tijd. Al vanaf de 17e eeuw zijn mensen bezig om rekenen te automatiseren. Pas in de 20e eeuw lukte het om een bruikbare logische computer te maken. De eerste computers voedden zich met ponskaarten. Dit zijn kaarten met gaten erin. De gaten in de kaarten werden omgezet naar enen en nullen; op deze manier konden elektrische machines lezen.
In 1935 liet John von Neumann zien dat de voorlopers van de computer instructies en data als één kon behandelen. Instructies en data konden op deze manier binnen de computer gebruikt worden, zo waren er geen ponskaarten meer nodig. De *Von Neumann Architectuur* is een standaard geworden voor alle moderne computersystemen. Von Neumann, die in die tijd net als Zuse in Berlijn woonde, heeft zijn tijdgenoot nooit ontmoet.

Von Neumann's systemen gingen om met machinetaalinstructies en dat was geen gemakkelijke taal om in te programmeren. Alle programma's bestonden uit bijna onleesbare codes. In de jaren '50 werd de '*assembly language*' ontwikkeld. Deze '*assembler talen*' gebruikten woordcodes om mee te programmeren, in plaats van getallen. Elke woordcode [3] kwam overeen met één machinetaalinstructie. Machinetaal en assemblytalen worden, omdat ze relatief diep op de technische details

[3] Deze woordcodes zijn mnemonics (zie hoofdstuk 3). Voor een programmeur is het woord '*mov*' eenvoudiger te onthouden dan de code 'B8'.

ingaan low-level languages genoemd.

2.2 Nu

In deze tijd wordt er veelal in hogere programmeertalen geprogrammeerd. Toch is assembly onontbeerlijk, ook voor hogere programmeertaalprogrammeurs. Omdat het programmeren in assembly inzicht geeft in de manier waarop computers werken krijgt een programmeur zicht op de mogelijkheden en de technische details achter software. Het is in assembly mogelijk om optimaal doelmatige software te schrijven.

In de volgende hoofdstukken wordt geprobeerd een zo goed mogelijk beeld te schetsen over assembly. Ondanks dat veel aspecten onbehandeld blijven, wordt er veel aandacht besteed aan de uitvoer en realisatie van hardware aangestuurd door software. Dit kan een goede ondergrond zijn om verdere aspecten van assembly te leren.

Hoofdstuk 3

Assembly

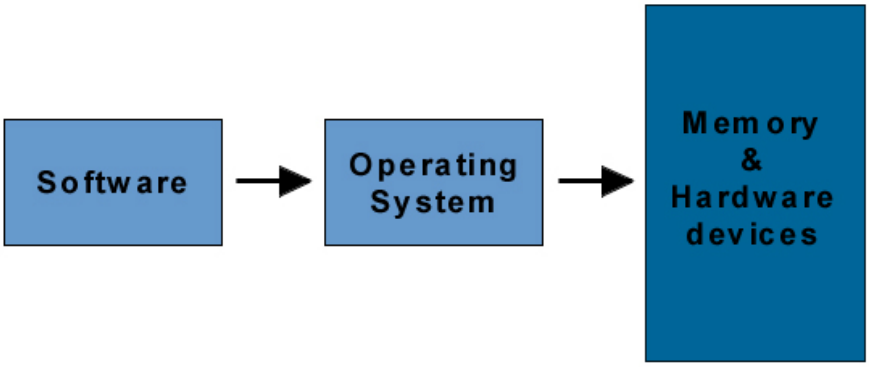
In dit hoofdstuk wordt de werking van assembly programma's uitgelegd. Het programmeren in assembly gebeurt in dit dictaat via het operating system, die vervolgens de hardware aanstuurt. Aan de hand van een 'hello world' voorbeeld leert men hoe system calls in Linux gebruikt kunnen worden.
Er wordt een beschrijving gegeven van de opbouw van assembly sourcecode, ook wordt er aandacht besteed aan de opbouw van assembly sourecode en de representatie van executables in het geheugen.

3.1 Assembly programmeren

Het schrijven van programma's in assembly kan op verschillende manieren. Er kan gebruik worden gemaakt van allerlei service's, bijvoorbeeld van libraries en system calls. Libraries zijn programmabibliotheken. Ze bestaan vaak uit voorgeprogrammeerde software routines die gebruikt kunnen worden voor generieke taken. System calls zijn aanspreekpunten om rechtstreeks services van de kernel te vragen.
In dit hoofdstuk wordt er geprogrammeerd aan de hand van system calls. Dit omdat system calls de werking van het Linux operating system laten zien, in tegenstelling tot libraries, die als een schil om system calls heen zitten.

Door assembly te programmeren wordt er achter de schermen van Linux gekeken. De Linux kernel is geschreven in C. De architectuur afhankelijke delen zijn in assembly geschreven.

De meeste softwaretools die gebruikt worden in deze handleiding staan standaard op een GNU/Linux systeem. De gebruikte tools die niet standaard geïnstalleerd zijn worden beschreven in de bijlages.



Figuur 3.1: *Protected operating system*

3.2 Hello world

Door gebruik te maken van de *Netwide Assembler* (*nasm*) en GNU *linker* (*ld*) kunnen we assembly programma's maken.
Als de assembly sourcecode gecompileerd is met NASM moet het worden gelinkt door een linker. De linker maakt van de objectcode een *ELF* binary die op het systeem gedraaid kan worden.

```
section .data
    hello db "Hello world", 0x0a
    len equ $ - hello    ; lengte hello (12)

section .text
global _start        ; entry point, label
_start:

; ssize_t write(int fd, const void *buf, size_t count);
    mov eax, 4       ; system call nr.  van write
    mov ebx, 1       ; 1e arg: fd van stdout
    mov ecx, hello   ; 2e arg: pointer naar hello
    mov edx, len     ; 3e arg: lengte hello
    int 0x80         ; call kernel
```

```
; void _exit(int status);
    mov eax, 1          ; system call nr. van exit
    mov ebx, 0          ; 1e arg: status
    int 0x80            ; call kernel
```

Objectcode is het resultaat van het compileren, een gecompileerde file bevat machinetaal in plaats van assembly instructies. Om er vervolgens een executable van te maken moet het gelinkt worden. Dit wordt gedaan met een linker (ld). Meestal heeft een file met objectcode de .o extensie.

NASM is niet de algemeen gebruikte *assembler* voor UNIX systemen, dat is '*as*'. Maar de syntax van *as* is niet gelijk aan de Intel syntax. Voor het installeren van NASM zie bijlage A.

Sla de gegeven assembly sourcecode op in het bestand 'hello.asm'. Er moet hierbij gebruik worden gemaakt van een teksteditor.
Door de volgende *commando's* te geven wordt het bestand gecompileerd en gelinkt.

nasm -f elf hello.asm
ld hello.o -o hello

Het commando 'nasm -f elf hello.asm' compileert het programma in objectcode en plaats de objectcode in 'hello.o'. De optie '-f elf' betekent dat de output ELF objectcode is.
Het linken gebeurt met 'ld hello.o -o hello'. De optie '-o' (output) geeft aan waar de gelinkte code (de executable) heen wordt geschreven.

Er bestaat nu een executable met de naam 'hello'. Dit bestand kan worden uitgevoerd door './hello'. Er verschijnt 'Hello world' op het scherm.

3.3 Assembly sourcecode

Universele assembly sourcecode bestaat uit vier velden.
De vier velden zijn: het *label* veld, het *mnemonic* [1] veld, het *operand veld* en het *commentaar veld*.

1. Het label veld kan worden gebruikt om het doel van een jump instructie aan

[1] Mnemonic is een Grieks woord en wordt uitgesproken als 'neh-MAHN-ik'. Mnemonics zijn afkortingen van woorden die voor instructies staan, ze zijn meer menslogisch dan numerieke waarden.

te geven. Een jump kan worden gezien als een *goto* instructie.
Bijvoorbeeld: '_start:' Labels eindigen op ':'.

2. Het mnemonic veld bevat de mnenonic die gekoppeld is aan een processorinstructie. Voorbeelden van mnemonic's zijn, add, sub, mov. Een mnemonic vervangt de moeilijk te onthouden processorinstructie of opcode.
Bijvoorbeeld: 'int' of 'mov'

3. Het operand veld bevat de operands waarop de instructies worden uitgevoerd. Heeft een mnemonic meer operands dan worden ze gescheiden door komma's.
Bijvoorbeeld: 'eax, [source]'

   Er zijn drie verschillende operands:

   - **Registers**
     Registers kunnen waarden bevatten. Meestal heeft het gebruik van registers de voorkeur boven het gebruik van geheugen omdat registers in de processor zelf liggen, waardoor ze sneller te adresseren zijn.
     Bijvoorbeeld: 'mov eax, edx' of 'inc ebx'
   - **Immediates**
     Dit zijn getallen zoals 99 of 0xc4fe.
     Bijvoorbeeld: 'mov eax, 0xb4dc0d3'
   - **Geheugen**
     Geheugen is in assembly gelabeld met een naam.
     Bijvoorbeeld: 'test eax, input'

     Wordt dit operand tussen vierkante haakjes gezet (*[]*) dan betekent dit dat het om de inhoud van het geheugen gaat, niet het adres (de pointer) zelf.
     Bijvoorbeeld 'mov esi, [demo]'
     Hier wordt de inhoud van het geheugen dat 'demo' heet in het ESI-register gezet.

4. Het laatste veld, het *commentaar veld*, wordt gescheiden door een ';', en is niet verplicht. Alles dat na de ';' staat wordt gezien als commentaar. Vooral bij low-level languages zoals assembly is commentaar zeer belangrijk. Goed commentaar kan sourcecode verduidelijken.

Een regel assembly sourcecode kan er bijvoorbeeld zo uitzien:

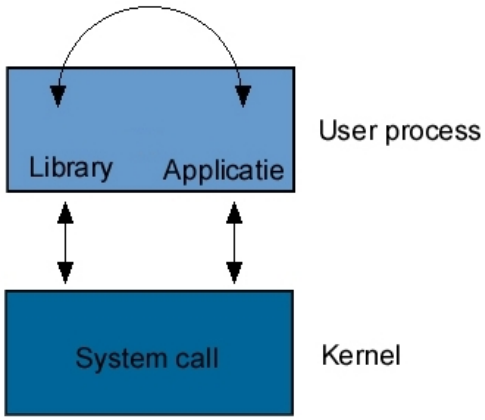
routine: mov eax, [memlocation]

Er wordt er een '*mov*'-instructie uitgevoerd die inhoud van memlocation (rechts)

naar eax (links) kopiëerd. In dit dictaat wordt er gebruik gemaakt van de Intel syntax die het source operand rechts neemt en het destination operand links.
Het statement 'routine:' geeft het label aan. Het gebruik van labels is niet verplicht.

3.4 Kernel, system calls en interrupts in Linux

Software bestaat vaak uit de volgende onderdelen: het vragen om input, het maken van berekeningen, het geven van output, en het sluiten. Om dit te realiseren, maakt het meestal gebruik van de services van het operating system. Assembly kan dus niet alleen processor afhankelijk zijn maar ook nog eens operating system afhankelijk.



Figuur 3.2: *System call en libraries*

Alle operating systemen leveren een bepaalde service aan de software; meestal gebeurt dit via en vanuit de kernel. UNIX en varianten, waaronder Linux, hebben een aantal entry points direct naar de kernel toe. Dit worden system calls genoemd.

System calls (figuur 3.2) kunnen vanuit C worden aangesproken met behulp van libraries, maar een applicatie kan ook direct gebruik maken van system calls. Meestal worden system calls via de C library (libc) wrapper aangesproken.

Het gebruik van libc in assembly is niet verplicht, het is een keuze van de programmeur.

Libc wrappers zijn gemaakt om software af te schermen voor syntax verandering van system calls. Maar omdat de UNIX kernel, en dus ook Linux, in hoge mate

POSIX [2] compliant is (sterker nog, system calls hebben dezelfde syntax als kernel system calls en vice versa), heeft het vaak weinig nut om gebruik te maken van system call wrappers, zoals printf(), scanf() enz. Daarom is het vaak makkelijker om direct gebruik te maken van kernel calls, omdat dit de snelste manier is om gebruik te maken van de kernel services. Het gebruik van kernel calls heeft ook weer nadelen, omdat het per gebruikte kernel en dus operating system verschilt.

3.4.1 System calls in assembly

De werking van het assembly programma dat '*hello world*' demonstreert is niet geheel duidelijk. Het lijkt in de verste verte niet op andere programmeertalen. Toch heeft het er alles mee te maken.

In het voorbeeld wordt er 'Hello world' op het scherm geschreven aan de hand van system calls. System calls zijn diensten van het operating system die een bepaalde taak uitvoeren.
System calls worden besproken in section 2 van de man page. De manpage van *write* is op te vragen door 'man 2 write'.

Daar staat de C definitie van de write system call.

*ssize_t write(int fd, const void *buf, size_t count);*

In assembly ziet het er anders uit, de Linux kernel gaat als volgt om met system calls op assembly niveau.

- **ssize_write()** Het EAX-register bevat het system call nummer uit het bestand /usr/include/asm/unistd.h.
- **int fd** Het EBX-register bevat het eerste argument, in dit geval de file descriptor voor stdout die 1 is. stdout is meestal gekoppeld aan het beeldscherm, waardoor berichten die naar stdout worden geschreven automatisch op het scherm verschijnen.
- **const void* buf** Het ECX-register bevat een pointer (geheugenadres) naar de string die afgedrukt dient te worden.
- **size_t count** Het EDX-register bevat het aantal tekens dat de af te drukken string lang is.

[2] POSIX (Portable Operating System Interface) is een set standaarden voor UNIX operating systems. Er was behoefte aan standaarden binnen de verschillende UNIX stromingen.
Voor meer informatie zie de website van Open Group, www.opengroup.org, en de System V implementatie op http://stage.caldera.com/developer/devspecs/.

Door het programma *strace* te gebruiken kunnen we zien welke system calls een programma gebruikt.

strace hello

Het resultaat ziet er zo uit.

```
execve("./hello", ["hello"], [/* 39 vars */]) = 0
write(1, "Hello world\n", 12Hello world
                )            = 12
_exit(0)                                  = ?
```

Er staan nu codes op het scherm die erg op C programmatuur lijken. Om precies te zijn staan er 3 system calls op het scherm. De eerste is *execve*, de tweede is *write* en de derde is *exit* (*_exit* is gelijk aan *exit*).
De eerste is de system call die het programma laat runnen vanuit de shell, execve draait './hello'. Dit staat niet in de sourcecode omdat de shell input van de user neemt en execve uitvoert om het programma te starten. De andere system calls in het voorbeeld staan wel in de assembly sourcecode. *Write* en *exit* worden beiden gebruikt in het programma.
Met de opdracht 'int 0x80' wordt de *kernel* [3] aangeroepen. De kernel kijkt in het EAX register en voert aan de hand van de waarde in het EAX register een system call uit. Bij een *interrupt* ('*int*'-instructie) wordt de CPU aangeroepen en voert deze een bepaalde handeling uit, het programma heeft dan tijdelijk geen toegang tot de CPU.
De system calls van Linux zijn gedefiniëerd in '/usr/include/asm/unistd.h'. Zo komt de system call *write* overeen met met 4, en *exit* met 1.

Als we hetzelfde programma in C hadden geschreven had het er zo uitgezien.

```
#include <unistd.h>
int main(void) {
        write(1, "Hello world\n", 12);
        exit(0);
}
```

[3] De kernel is het operating system zonder applicaties (programma's). Het is de interface tussen hardware en de applicaties. Zo zorgt het ervoor dat er meerdere processen tegelijk kunnen draaien en dat deze niet elkaars geheugen kunnen overschrijven. (Walter de Jong bedankt voor je definitie.)

Sla de C sourcecode op in het bestand 'hello2.c' en compileer en link het met ' *gcc hello2.c -o hello2*'. Vergelijk de uitkomst van 'hello2' en 'hello' door beide uit te voeren.
Er is geen verschil in uitkomst. Betrek nu '*strace*', en ondervindt het verschil in de binaries. Al snel blijkt dat er bij de assembly versie van 'Hello world' twee system calls gebruikt worden in tegenstelling tot de C versie, waar er veel meer worden gebruikt.

Er zijn nog veel meer verschillen tussen de binaries, zoals de snelheid en grootte van de executable. De volgende commando's laten die verschillen zien.

ls -al hello{' ',2}
time hello
time hello2

Voor meer informatie over *time* raadpleeg de manpage.

3.5 Binary executables voor Linux

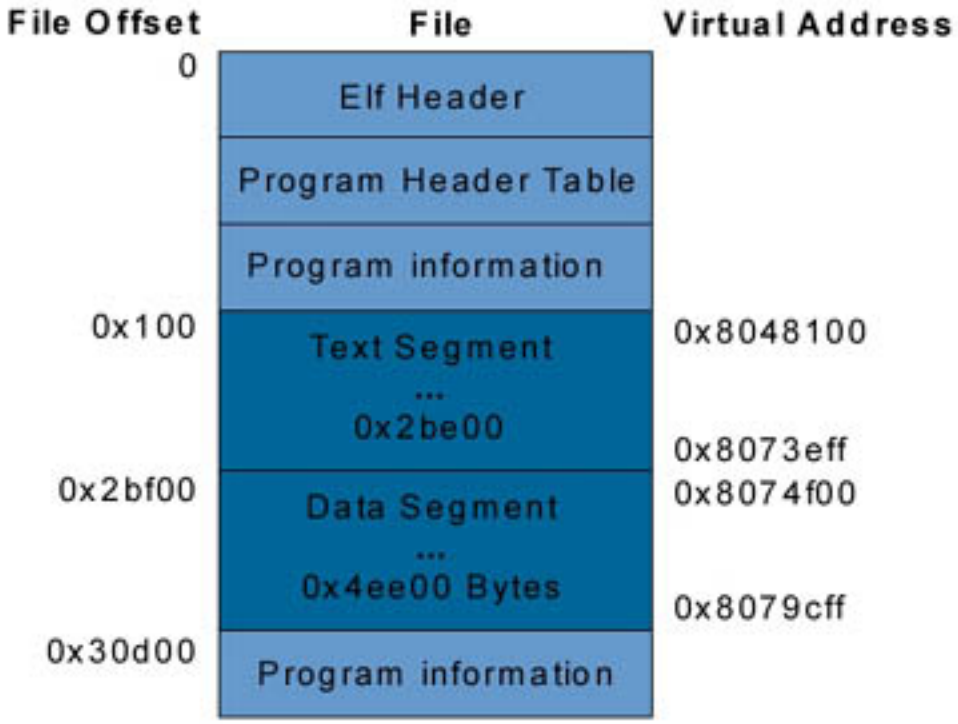
Moderne 32-bit *Unices* [4], zoals Linux, draaien in protected mode en maken gebruik van het '*Executable and Linkable Format*' (*ELF*) voor executable binaries. System calls in Linux worden gemaakt via 'int 0x80'. Het function nummer van de system call wordt uitgelezen uit EAX en de argumenten staan in de registers, niet op de stack. Er zijn maximaal zes argumenten mogelijk, EBX, ECX, EDX, ESI, EDI en EBP.

3.6 Sections

Assembly sourcecode bestaat uit één of meer delen, zogenaamde *sections* (figuur 3.3). Deze sections komen overeen met *segmenten* in het geheugen. Er zijn verschillende sections elk met een verschillend doel.

- .data section
  Deze section wordt gebruikt om constanten te declareren, zoals file names en buffer groottes. Het gaat hier om geïnitialiseerde data.
  Het data segment verandert niet tijdens het draaien van het programma.

[4] meervoud van UNIX



Figuur 3.3: *Een executable binary geladen in het geheugen*

In dit deel staan de *db*, *dd*, *dw* statements, het zijn *directives* en ze reserveren data in het geheugen. De 'd' uit deze opdrachten staat voor *define*. Een directive wordt alleen gebruikt door de assembler, na het vertalen van assembly naar machine code bestaan ze dan ook niet meer (alleen geheugenadressen worden gebruikt in de executable).

De opdracht *equ* (*equate*) maakt het mogelijk om symbolische namen te geven aan getallen. Deze worden door de assembler vervangen door getallen. Ook is het mogelijk om via een equate de lengte van reserveerd geheugen te bepalen. Met 'formatlen equ $ - format' neemt de symbolische naam 'formatlen' de lengte van de string 'format' aan.
Bijvoorbeeld:

```
string db "Dit is een string" , 0
format db "x=%d\n", 0
formatlen equ $ - format
error db "Error", 0xa
```

```
year equ 365
```

Bij strings wordt er geëindigd met een '0', omdat er anders niet bekend is waar de string eindigt. Zo is de code gelijk aan de strings of character arrays in C.

- .bss section
  In dit gedeelte worden de variabelen gedeclareerd. Het gaat hier om ongeïnitialiseerde data in het geheugen.
  De *geheugenvelden* hebben nog geen waarde; er wordt alleen nog maar geheugen gereserveerd. Pas als het programma draait zullen er bruikbare gegevens in komen te staan. Het .bss section verandert niets aan de executable als file. Omdat het .bss section nog geen data bevat en het wel om moet gaan met data is het write-enabled.

```
input resb 1
pi resq 1
```

- .text section
  Dit is het gedeelte waar de assembly code wordt geschreven. Op deze plek staan de instructies (weergeven als mnemonics).
  Er is een label gedeclareerd dat begint met een '_' (underscore). Dit betekent dat hier een entry is, zodat andere software het kan starten.
  De code in het code section van een programma hoeft niet lineair uitgevoerd te worden; er bestaan namelijk calls en jumps die de volgorde van executie veranderen. Het code section of .text section is als segment in het geheugen read-only. Tenzij er een wijziging is gemaakt in de program header (ELF header).

```
call re
mov [input], eax
```

De opdrachten die in het het .data en .bss section staan zijn geen instructies. Ze kunnen gezien worden als *pseudo-instructies*. Omdat bijvoorbeeld het reserveren van bytes en het labelen ervan op machine niveau niet bestaat. Het label wordt in de executable vervangen door een geheugenadres.

Zie hier een tabel met de datatypes en de grootte ervan. Deze worden gebruikt bij de res*x* en d*x* statements.

Intel datatype	C declaratie	Suffix	Grootte (Bytes)
Byte	char	b	1
Word	short	w	2
Double Word	int	d	4
Quad Word	double	q	8
Double Word	char *	d	4

Ter illustratie: Een pointer (dus ook een character pointer) is een double word (4Bytes, 32-bit). Dit omdat geheugenadressen 32-bits geadresseerd worden.

Om een indruk te krijgen van het .data en het .bss segment die in het geheugen liggen, kan men 'hello.asm' compileren en linken. Door het vervolgens in ALD te laden, kan men delen van het data segment bekijken. ALD staat voor Assembly Language Debugger, het is een assembly debugger voor de x86 die instructies in Intel syntax weergeeft. Zie ook bijlage B.

ald hello

Typ nu twee maal 'next' in tot dat de opdracht 'mov ecx, 0x80490a4' verschijnt. [5]
Bij deze opdracht wordt de pointer naar (of het adres van) de 'hello' string in het data segment naar het ECX-register geschreven. Dit betekent dat op 0x80490a4 *"Hello world", 0xa* staat.
Typ nu 'examine 0x80490a4' voor een dump van 20 bytes na 0x80490a4.

. . .
ald> examine 0x80490a4
Dumping 20 bytes of memory starting at 0x080490A4 in hex
080490A4: 48 65 6C 6C 6F 20 77 6F 72 6C 64 0A 00 00 00 00 Hello world.....
080490B4: 00 00 00 00 ....

[5] De waarde 0x80490a4 kan van systeem tot systeem verschillen.

3.7 Vragen en opdrachten

1. (a) Leg uit wat een system call is. Geef ook het verband aan met het operating system.
   (b) Wat is het verschil tussen een system call en een library functie?
   (c) Wat is een nadeel van system calls, vergeleken met andere routines?
   (d) Wat is een voordeel van system calls, ten opzichte van library calls?
2. (a) Assembly sourcecode bestaat uit sections, die overeen komen met delen in het geheugen. Waarom zijn sommige delen read-only?
   (b) Wat zou er gebeuren als die delen write enabled zijn? Geef een aantal voorbeelden en geef aan wat er zou gebeuren met het programma en het operating system.
3. Is het mogelijk om assembly en C in één programma te laten samenwerken. Waarom wel of waarom niet?
4. Assembly programmatuur is meestal kleiner en sneller dan software geschreven in hogere programmeertalen.
   (a) Waarom is dit?
   (b) Waarom wordt er dan niet massaal in assembly geschreven?
5. De sourcecode van het assembly programma 'hello.asm' zet 'Hello world' op het scherm. Verander de source zo dat het meerdere regels op het scherm zet door twee keer gebruik te maken van de system call write (let op: er moet twee keer een string worden gereserveerd).
6. (a) Waarom bestaat assembly code uit velden?
   (b) Geef de namen van de verschillende velden.
   (c) Bekijk de source code van 'hello.asm'. Uit welk veld komt 'mov'?
7. Is het verplicht op alle regels van assembly sourcecode 4 velden te hebben? Licht je antwoord toe.
8. (a) Vraag de manpage op van de system call 'chmod'. Er zijn namelijk verschillende manpages van 'chmod' omdat het ook een shell utility is.
   (b) Zoek het nummer van de system call die bij 'chmod' hoort.
   (hint: Gebruik hierbij de includes op het Linux systeem /usr/include/asm/unistd.h)

9. Schrijf een assembly programma dat input leest via de system call 'read'. Na het inlezen moet het programma via 'write' de de input van 'read' afdrukken. Maak gebruik van een variabele 'input' die als volgt wordt gedeclareerd in assembly.

```
section .bss
    input resd 1
```

   input kan bij de system call 'read' worden gezien als '*void *buf*' (het 2e argument) en wordt in het ECX-register gezet. Zie ook de man page van 'read'.

10. Het onderstaande programma, '*filexs.asm*' drukt 'file.txt' af. Verander het programma zo, zodat het '/etc/inittab' afdrukt.

    De inhoud van 'filexs.asm'.

```
section .bss
    buf resd 1
    count resd 1

section .data
    pathname db "file.txt", 0

section .text
global _start
_start:

; int open(const char *pathname, int flags);
    mov eax,5
    mov ebx, pathname
    mov ecx, 0 ; O_RDONLY uit /usr/include/asm/fcntl.h
    int 0x80
; eax kan error status bevatten. Dit wordt
; hier genegeerd.

; open returned een fd in eax
    mov ebx, eax

; ssize_t read(int fd, void *buf, size_t count);
```

```
; met read bepalen we het aantal uit te lezen bytes
    mov eax, 3          ; read syscall
                        ; ebx bevat de fd die open returnde
    mov ecx, buf        ; adres buf in ecx
    mov edx, count      ; count in edx
    int 0x80            ; lees 'count' aantal bytes

; read returned het aantal gelezen bytes in eax
; write gebruikt dit aantal als parameter
    mov edx, eax

; ssize_t write(int fd, const void *buf, size_t count);
    mov eax, 4          ; write syscall
    mov ebx, 1          ; fd van stdout
                        ; ecx bevat het adres van buf
                        ; edx bevat count
    int 0x80

; void _exit(int status);
    mov eax, 1
    mov ebx, 0
    int 0x80
```

De inhoud van 'file.txt'.

```
Q: What's a redneck fortune cookie?
A: A piece of cornbread with a food stamp baked inside.
```

Hoofdstuk 4

Architectuur

In dit hoofdstuk wordt de processor van een computer onder de loep genomen. Er wordt aandacht besteed aan de werking van de processor en aan de verschillende onderdelen in de processor. De grote verschillen tussen de processoren die er op de markt zijn en de kenmerken hiervan worden beschreven. De Intel processor wordt als uitgangspunt genomen.

4.1 Historie

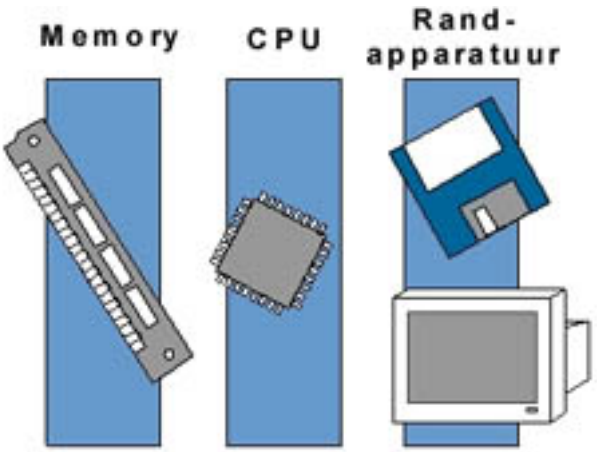
Tijdens de late jaren '60 werden geïntegreerde elektronische circuits geïntroduceerd. Honderden transistors, dioden, en resistoren pasten in één aparte chip. Het gevolg was een enorme snelheidswinst, een hogere capaciteit, meer betrouwbaarheid en het was velen malen goedkoper dan de traditionele hardware. Er was een nieuw fenomeen: de *microprocessor*.

4.2 Het Hart

Het hart van de computer is de *CPU*. Het staat voor *Central Processing Unit*, meestal is dit een andere term voor *processor* of *microprocessor*. De CPU beheerst alle interne en externe hardware en voert logische en rekenkundige opdrachten uit. Het is de meest verantwoordelijke chip in het systeem en bevat een logisch circuit dat de *instructies* uitvoert van de software. De instructies staan in een *instructieset* die vastligt in het *ROM geheugen* van de processor, dat niet herschrijfbaar is.

De instructieset bepaalt de *machinetaal* van de CPU. Is een instructieset te uitgebreid, dan heeft dat gevolgen voor de performance van het systeem. Processors

met verschillende instructiesets kunnen niet dezelfde software aan. Een gecompileerd programma voor een *Apple* computer zal niet draaien op een *Intel* systeem.



Figuur 4.1: *De CPU staat tussen het geheugen en de randapparatuur in.*

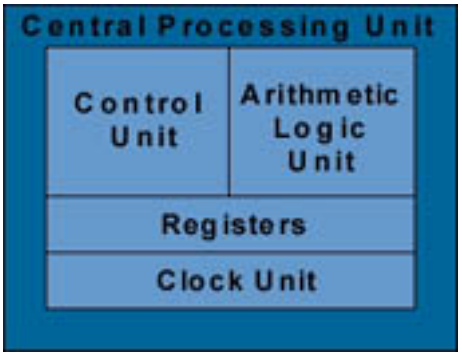
De CPU is de belangrijkste chip in een computer; toch is het mogelijk om er één of meer in een systeem te hebben. Deze techniek heet *SMP* (Symmetric Multi-Processing). Het zorgt er voor dat verschillende processoren het operating system en het geheugen delen. SMP systemen presteren beter dan *MMP* (Mono Multi-Processing) systemen. Het voordeel kan worden behaald door een balans van workload (uit te voeren werk) tussen verschillende processoren.

4.3 Processor onderdelen

Een CPU bestaat uit verschillende onderdelen. Zeer vereenvoudigd zijn er de volgende onderdelen, elk met een eigen taak (zie figuur 4.2).

1. **Control Unit**
   De Control Unit regelt de activiteiten van interne en externe apparaten. Het interpreteert de instructies, het bepaalt welke data er nodig zijn en de plaats ervan. Ook houdt het zich bezig met het wegschrijven van resultaten. Het stuurt signalen, die de hardware devices aanzetten tot het uitvoeren van de instructies.

2. **ALU**
   De ALU (Arithmetic and Logic Unit) is het deel van de CPU waar de rekenkundige bewerkingen plaatsvinden. Het bestaat uit circuits die rekenkundige



Figuur 4.2: *Schematische weergave van CPU*

operaties kunnen uitvoeren zoals optellen, aftrekken, vermenigvuldigen en delen op data die verkregen zijn uit het geheugen en de registers. Ook heeft het de mogelijkheid data te vergelijken en met bits te schuiven.

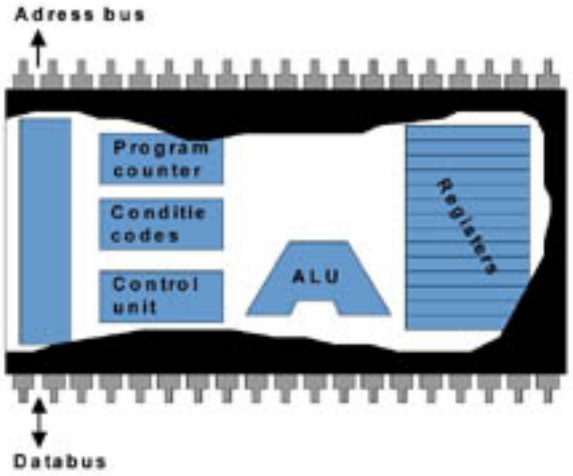
3. **Registers**
   Als de ALU taken uitvoert, heeft het tijdelijke opslagruimte nodig voor data. Dit gebeurt in de registers. Registers zijn speciale geheugencellen in de processor. Ze worden gebruikt om geheugen te adresseren, data te manipuleren en te verwerken. Sommige registers hebben een gereserveerd doel, terwijl andere *general-purpose registers* zijn. Omdat de data in de registers telkens worden gebruikt, bestaan ze uit zeer snel geheugen dat data direct bewerkt en de uitkomst weer kan opslaan na een bewerking. Mocht de data in een register niet nodig zijn voor de volgende instructie dan worden ze naar het *RAM* geschreven en houden de registers zich bezig met de volgende instructie. Dit als een programmeur efficiënte assembly code heeft geschreven.

4. **Clock Unit**
   De activiteiten van een processor bestaan uit verschillende instructies voorgesteld in duizenden individuele stappen. De stappen moeten in de juiste volgorde van vaste intervallen verlopen. Het deel van de processor dat zich met die intervallen bezig houdt, is de Clock Unit, ook wel Timer genoemd. Elke operatie in de processor vindt plaats op de puls van de Clock Unit. Geen enkele operatie, hoe simpel ook, kan worden gedaan in een tijd die korter is dan die tussen de intervallen. Maar sommige operaties kosten meer dan een puls. Geeft de Clock Unit kortere pulsen dan werkt de processor sneller. De *kloksnelheid* wordt aangeduid in *MegaHertz* (*MHz*), en tegenwoordig ook *GigaHertz* (*GHz*). Een processor van *n* MHz heeft een Clock Unit die *n*

miljoen pulsen per seconde produceert.

Processor informatie opvragen in een Linux systeem kan door het volgende commando [1].

cat /proc/cpuinfo



Figuur 4.3: *De CPU chip*

4.4 Kenmerken

De kenmerken van een processor zijn:

1. Het aantal uitgevoerde instructies per seconde
2. Het aantal bits van de woordlengte

De woordlengte is het aantal bits dat een CPU kan verwerken per *clockcycle*.

Er zijn twee marktleidende fabrikanten van processoren: Intel en *Motorola*. Andere processoren zijn meestal compatible met deze. Bijvoorbeeld *AMD* en *Cyrix*, deze processoren gebruiken dezelfde machinetaal en instructieset als de Intel, terwijl hun inwendige structuur verschilt.

[1] Zie voor extra informatie over de implementatie van de processor in het operating system de includes van de kernel: '/usr/include/asm-i386/processor.h'

4.5 Geheugen

Computergeheugen bestaat uit logische circuits, dit wordt statisch geheugen of *SRAM* genoemd. Geheugen circuits bestaan uit cellen; ze representeren nullen en enen. Een verzameling van cellen noemt men een chip.

In *DRAM* (Dynamic RAM) liggen de geheugencellen heel erg dicht tegen elkaar aan. Ze verliezen hun inhoud snel, daarom dient DRAM regelmatig ververst te worden.

Bij *phsysical memory*, het echte (tastbare) geheugen is de omvang makkelijk te berekenen. Dit gaat op dezelfde manier als bij rechthoeken: 2 x l x b, waar l staat voor de lengte, oftewel het aantal words, en b staat voor de breedte, oftewel de woordgrootte.

De volgende commando's geven informatie over het geheugen.

cat /proc/meminfo
free

4.6 CISC/RISC

Bij het ontwerp van processoren bedacht men een techniek die *CISC* (Complex Instruction Set Computer of Computing) heette. Een CISC processor is een processor met een zeer rijke instructieset. Dit zou een betere samenwerking met hogere programmeertalen hebben.
Dit voordeel is te weerleggen omdat hogere programmeertalen nogal verschillen en compilers die gebruik moeten maken van de instructieset ook. Zo wordt het onnodig ingewikkeld.
De Intel *80x86* serie maakt gebruik van deze CISC technologie.

Omdat er steeds meer in hogere programmeertalen werd geprogrammeerd en omdat geheugen steeds sneller werd, verdwenen de belangrijkste redenen van CISC. Maar nog steeds hielden ontwikkelaars zich bezig met het versnellen van processoren. Een van de technieken die werd geïntroduceerd was *RISC* (Reduced Instruction Set Computer). Een RISC processor bevatte een gereduceerd aantal simpele instructies.
Een lange reeks simpele instructies presteert uiteindelijk dan net zoveel of zelfs beter op een RISC processor dan een enkele complexe instructie op een CISC processor die hetzelfde doet. Het belangrijkste doel voor de ontwikkelaars was niet het ontwikkelen van de instructieset, maar de snelheid van de chip. Bij RISC

architecturen zijn de frequent gebruikte instructies snel. Eén van de designtechnieken was om meerdere instructies tegelijk uit te voeren (*pipelining*).

Dit heeft tot gevolg dat in het meest ideale geval de RISC processor één instructie per kloktik kan uitvoeren. Het aantal megahertz kan zo gelijk zijn aan het aantal instructies per seconde; dit wordt ook wel *MIPS* (Million Instructions Per Second) genoemd. Dit lukt niet altijd, maar toch is er een effectieve capaciteit van zo'n 60% tot 70% mogelijk, afhankelijk van de te verwerken taak. CISC zou zulke snelheden niet kunnen halen; toch zijn de instructies krachtiger dan die van de RISC. RISC is taakonafhankelijk, in tegenstelling tot CISC. Een RISC processor kan daardoor 2 tot 4 maal sneller zijn. De *PowerPC* van Apple en de huidige generatie van *Macintosh* systemen zijn gebaseerd op RISC.

De processor afhankelijke delen van de Linux kernel zijn voor een groot deel in assembly geschreven. Het biedt meerdere assembly sourcecodes aan voor de verschillende architecturen die het ondersteunt.
Geef het volgende commando in.

ls -l /usr/src/linux/include/ | grep asm

Dit geeft aan dat de kernel source symbolisch gelinkt is met de assembly sourcecode voor de i386. Verder geeft het een lijst van alle processor architecturen aan die ondersteund worden, zoals de de Sun Sparc, Dec Aplha en andere types. [2]

4.7 Endian

"It is allowed on all Hands, that the primitive Way of breaking Eggs before we eat them, was upon the larger End: But his present Majesty's Grand-father, while he was a Boy, going to eat an Egg, and breaking it according to the ancient Practice, happened to cut one of his Fingers. Whereupon the Emperor his Father, published an Edict, commanding all his Subjects, upon great Penalties, to break the smaller End of their Eggs. The People so highly resented this Law, that our Histories tell us, there have been six Rebellions raised on that Account;... It is computed that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy: But the Books of the Big-Endians have been long forbidden..."

Jonathan Swift's Gulliver's Travels

[2] Voor meer informatie over de Linux kernel en verschillende processor architecturen zie: http://www.kernel.org

De termen *big-endian* en *little-endian* zijn geleend uit Gulliver's Travels.
In een computer staan ze voor de volgorde waarin bytes worden opgeslagen in het geheugen (*byte ordering*).

Bij big-endian staat het 'big-end', de meest significante byte vooraan en wordt het bewaard op het laagste geheugenadres.
Little-endian is de volgorde waarbij de minst significante byte vooraan in het geheugen wordt bewaard.
Bijvoorbeeld, in een big-endian computer, worden de bytes voor het hexadecimale getal B9 62 bewaard als volgt. B9 wordt bijvoorbeeld op adres 1000 bewaard en 62 op 1001. Bij een little endian systeem gaat het net andersom. 62 zou worden bewaard op 1000, B9 op 1001.

Nog een voorbeeld:

- Bij *little-endian* wordt 0x65 66 67 68 opgeslagen als 0x68 0x67 0x66 0x65.
- Bij *big-endian* wordt 0x65 66 67 68 opgeslagen als 0x65 0x66 0x67 0x68.

De meeste RISC processoren zijn big-endian. Voor Nederlanders, die graag van links naar rechts lezen, is dit ideaal. Maar de Intel processoren en de software die ervoor wordt geschreven is little-endian.

Little endian betekent dat het 'minst significante byte' (LSB) wordt bewaard op het laagste geheugenadres (vooraan). [3]

Door de opdracht 'file <bestand>' te geven kan men zien wat voor executable het is en of het geschikt is voor het gebruikte operating system en architectuur.
Bijvoorbeeld:

file /bin/bash

Kort de voordelen van little-endian en big-endian:

- *Little-endian*: makelijk implementeerbaar in hardware
- *Big-endian*: begrijpelijker

4.7.1 Little-endian

Over de Intel wordt gezegd dat het een little-endian architectuur is. Dit kan worden getest door naar de opcodes (de uit te voeren instructies in hexadecimale waar-

[3] Bekijk de files in de directory '/usr/include/linux/byteorder/' voor meer informatie over big- en little-endian.

den) te kijken.

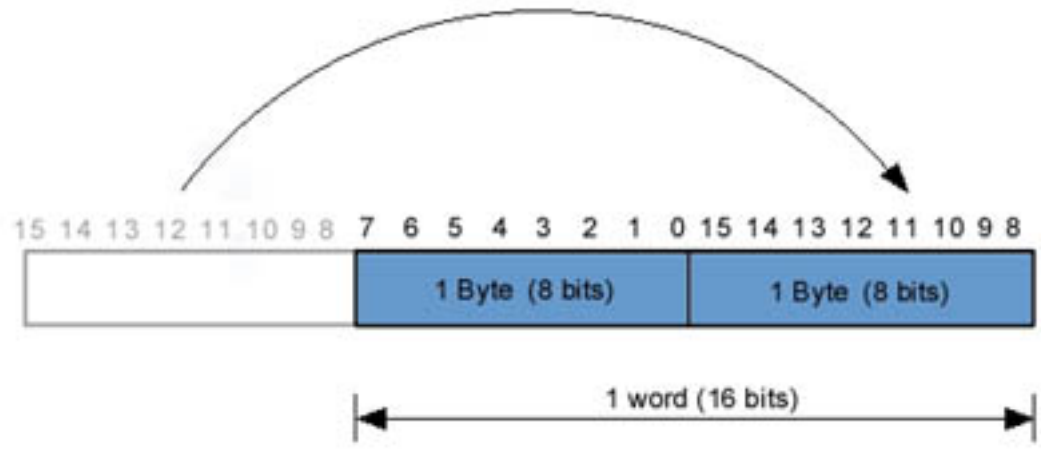
De volgende sourcecode, 'endian.asm', demonstreert de werking van little-endian, door gebruikt te maken van een *debugger*.

```
section .text

global _start
_start:

        mov     eax,    1
        mov     eax,    0x8001
        mov     eax,    0x00008001
        mov     eax,    0x8001abcd

;exit
        mov     ebx,    0
        mov     eax,    1
        int     0x80
```



Figuur 4.4: *De representatie van een word in het geheugen van een little-endian systeem.*

Compileer en link de sourcecode. Open het in de *Assembly Language Debugger* (ALD).

ald endian

Een snapshot van de uitkomst.

```
...
ald> next
eax = 0x00000001 ebx = 0x00000000 ecx = 0x00000000 edx = 0x00000000
esp = 0xBFFFF970 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds = 0x0000002B es = 0x0000002B fs = 0x00000000 gs = 0x00000000
ss = 0x0000002B cs = 0x00000023 eip = 0x08048085 eflags = 0x00200346

08048085 B801800000 mov eax, 0x8001
ald> next
eax = 0x00008001 ebx = 0x00000000 ecx = 0x00000000 edx = 0x00000000
esp = 0xBFFFF970 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds = 0x0000002B es = 0x0000002B fs = 0x00000000 gs = 0x00000000
ss = 0x0000002B cs = 0x00000023 eip = 0x0804808A eflags = 0x00200346

0804808A B801800000 mov eax, 0x8001
ald> next
eax = 0x00008001 ebx = 0x00000000 ecx = 0x00000000 edx = 0x00000000
esp = 0xBFFFF970 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds = 0x0000002B es = 0x0000002B fs = 0x00000000 gs = 0x00000000
ss = 0x0000002B cs = 0x00000023 eip = 0x0804808F eflags = 0x00200346

0804808F B8CDAB0180 mov eax, 0x8001abcd
ald> next
eax = 0x8001ABCD ebx = 0x00000000 ecx = 0x00000000 edx = 0x00000000
esp = 0xBFFFF970 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds = 0x0000002B es = 0x0000002B fs = 0x00000000 gs = 0x00000000
ss = 0x0000002B cs = 0x00000023 eip = 0x08048094 eflags = 0x00200346
```

De debugger laat door het commando 'next' de uitwerking zien van de instructies op de registers.
Bekijk de dikgedrukte regel. In het geheugen wordt op het adres 0804808F een instructie gelezen; het programma is namelijk geladen in het geheugen.

De opdracht 'mov eax, 0x8001' wordt vertaald in B801800000 in machinecode. Het getal 0x8001 wordt opgedeeld in delen van 2 delen van elk 1 byte, 01 en 80. B8 staat voor de opcode (instructie) van 'mov eax'.

De opdracht 'mov eax, 0x8001abcd' wordt vertaald naar B8CDAB0180. Het laatste gedeelte de 'abcd' staat nu vooraan in een andere volgorde.

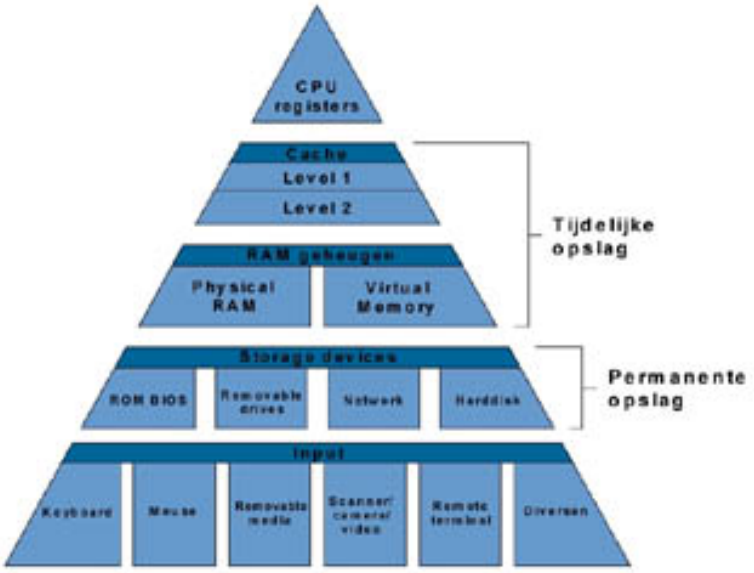
Ook kan de lengte van de instructies verschillen. Op een Intel systeem kan dat liggen tussen de 1 en 15 bytes. Instructies die vaak voorkomen zijn meestal klein en zijn daarom sneller te verwerken.

4.8 De modi van de 80x86

De Intel *80386* heeft mogelijkheden om de tekortkomingen van de oudere *8086* tegemoet te komen. De 8086 heeft bijna geen mogelijkheden op het gebied van het ondersteunen van geheugen protectie, virtual memory, multitasking en de manier om geheugen te adresseren boven de 640 KB.
Toch is de 80386 geheel backwards compatible met de 8086. De 80386, *80486* en verder worden hier als elkaars gelijken gezien. Het gaat erom dat de 80386 gebaseerd is op al zijn voorgangers en dat de meeste technieken gelijk zijn gebleven bij nieuwere types in de 80386+ serie.

De 8086 werkt met de *real mode*. Toen de ontwerpers bij Intel de 80286 ontwierpen wilden ze extra mogelijkheden toevoegen aan de 8086. Maar ze wilden ook 8086 compatibiliteit. Om aan beide eisen te voldoen verzonnen ze twee modi: Real Mode en *Protected Mode*. Real mode, de standaard op dat moment, maakte van de chip een 8086 met kleine veranderingen. Veranderingen waren er in overvloed als er gekeken wordt naar de *protected mode*. Bijna alle software die ontworpen is voor de 8086 zal niet werken in de protected mode zonder aanpassingen. *DOS* was in de tijd één van deze software.

De 80386 heeft meer capaciteiten dan de 8086 en de 286. De standaard mode, zoals die bij de voorgangers was de real mode. De 286 en 386 kunnen beide opereren in de protected mode. Toch is de interne structuur is totaal verschillend. De 386 heeft namelijk 32-bits registers en twee nieuwe 16-bit registers. De 386 support nog een derde mode, *Virtual 8086 mode* (V86). In de *V86 mode* handelt de 386 als in de protected mode maar behandelt het processen als in de real mode. V86 kan worden gezien als een *emulator*. [4]



Figuur 4.5: *Tijdelijke opslag en permanente opslag*

4.9 Voordelen van de protected mode

1. **Adressering**
   De 386 kan 4 GB aan geheugen adresseren, dit is het meest significante verschil tussen de protected mode en de real mode. Protected mode software kan zonder probleem 4 GB (2^{32} bytes) aan geheugen voor data, code en stack reserveren. Door gebruik te maken van omslachtige technieken kan de real mode ook adresseren boven de 1 MB gebruiken om data te plaatsen. Deze technieken, het lineair adresseren van de code en stack ruimte, is in het algemeen onhandig. Bovenal, het volledige 4 GB adresseren komt weinig voor bij computers van het PC kaliber.

2. **Virtual memory**
   De Memory Management Unit (*MMU*) van de 386 kan virtual memory implementeren, dat er voor kan zorgen dat processen denken dat er 4 GB aan geheugen is, terwijl er in werkelijk veel minder aanwezig is. In plaats daarvan gebruikt het delen van de harddisk om data op te slaan.

   Computers hebben een bepaald aantal MB's aan RAM beschikbaar voor de

[4] Een emulator is hardware of software dat zich voordoet als iets anders, zodat de software die gebruik maakt van de emulator op dezelfde manier werkt als in de verwachte omgeving. Deze techniek wordt veel gebruikt bij hardware die backwards compatible moet zijn.

CPU. Helaas is dat niet altijd voldoende om alle processen tegelijk te laten draaien. Als het *operating system* een groot aantal processen draait in het fysieke geheugen is het bijna niet mogelijk om alle data van de processen op te slaan. Kan het operating system geen *virtual memory* gebruiken dan zal het aangeven dat er niet genoeg *RAM* beschikbaar is en adviseren om bepaalde processen te beëindigen.
Virtual memory gaat hier anders mee om. Er wordt gekeken welke delen van het geheugen het minst recent zijn gebruikt en kopiëert die vervolgens naar de harddisk. Meestal gebeurt dit in de vorm van een swap file of *swap-partitie*.

3. **Adres vertaling**
   De *MMU* kan ook adressen vertalen voordat ze gebruikt worden. Software processen werken met logische adressen. De 386 converteert lineair de logische adressen naar 32-bit (zonder segment adres). De MMU converteert dan weer lineaire adressen naar fysieke adressen. fysieke adressen liggen op de RAM chips.

4. **Segmentatie**
   In real mode zijn alle *segmenten* 64 KB lang en zitten op vaste locaties. In protected mode kunnen segmenten verschillen in lengte van 1 byte to 4 GB. De segmenten kunnen onderverdeeld worden in 4 K pages. Pages vervangen de segmenten.

5. **Process protection**
   Verschillende processen kunnen beveiligd worden tegen elkaar. Het ene proces heeft geen toegang tot de data van het andere proces. Het operating systeem heeft ongelimiteerd toegang tot alle processen. Maar processen hebben dit zeker niet, ze kunnen geen data van andere processen dwars zitten of sturen. Dit maakt de loop van een proces onafhankelijk van andere processen. [5]

6. **32-bit registers**
   Registers op de 386 zijn 32-bits lang. Behalve de opgedeelde registers, zoals AX (16 bit) in het EAX (32 bit) register. Ook zijn er meer segment registers bijgekomen.

7. **Adresserings methodes**
   In de real mode konden constante waarden alleen worden gebruikt in het BX, BP en SI, DI register. In de protected mode kunnen alle registers gebruikt worden.

[5] Tenzij de kernel dit expliciet toelaat, bijvoorbeeld bij *ptrace*. Voor meer info 'man 2 ptrace'.

8. **Multitasking**
   Ook een zeer belangrijk voordeel van de 386 is dat het multitasking support. De 386 heeft veel snelle technieken om de huidige processor stand (alle data in de registers) tijdelijk op te slaan en verder te gaan met een ander proces. Later kan de stand van de processor weer worden opgepakt en kan er verder worden gegaan, net alsof er geen onderbreking heeft plaatsgevonden.

9. **Debuggen van hardware**
   De 386 heeft speciale hardware voor het debuggen van data breakpoints. Wat wil zeggen dat op hardware niveau de data in bepaalde registers in de gaten kan worden gehouden.

4.10 De 80x86 CPU familie

Figuur 4.6: *De Intel 386 processor*

1. **8088, 8086** Deze zijn als er zeer oppervlakkig wordt gekeken gelijk, intern verschillen ze wel degelijk. Beiden werden ze gebruikt in de eerste PC's. Ze bevatten ze de volgende 16-bit registers: AX, BX, CX, DX, SI, DI, BP, SP,

CS, DS, SS, ES, IP, FLAGS. Deze processoren ondersteunen real mode. In deze mode kan een proces al het geheugen adresseren, inclusief dat van andere processen. Dit is zeer gevaarlijk en maakt debugging extra moeilijk. Het geheugen wordt verdeeld in segmenten van 64 KB.

2. **80286** Deze CPU werd het eerst gebruikt in de AT-PC. Het voegt een paar extra instructies toe aan de 8086. Het heeft namelijk de mogelijkheid om in 16-bit protected mode te werken. Het kan nu 16 megabytes aan geheugen adresseren en het beschermt processen van elkaar. Helaas zijn de segmenten verdeeld in delen van 64 KB.

3. **80386** Deze CPU volgde de 80286 op en nam veel eigenschappen over inclusief de registers en voegde daar de volgende 32-bits registers aan toe: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP en twee nieuwe 16-bit segment registers FS en GS. Ook maakt deze CPU gebruik van een nieuwe 32-bit protected mode. In deze mode kan het tot 4 GB aan geheugen adresseren. Processen worden verdeeld in segmenten, die nu pages heten en kunnen een lengte hebben van 1 byte tot 4 GB.

4. **80486/Pentium/Pentium Pro** Deze CPU's voegen weinig nieuwe instructies toe, ze zorgen er voor dat de instructies sneller worden uitgevoerd.

5. **Pentium MMX** Deze processor voegt MMX (MultiMedia, eXentions) toe aan de instructieset van de Pentium. De instructies zorgen er voor dat veel gebruikte grafische operaties sneller worden uitgevoerd.

6. **Pentium II/Pentium III/Pentium 4** Dit is de Pentium Pro met MMX. De Pentium III/4 is eigenlijk alleen qua verwerkingssnelheid verschillend van de Pentium II.

7. **Itanium/IA-64** De Itanium is een 64-bits processor architectuur die gebaseerd is op EPIC (Explicitly Parallel Instruction Computing), het parallel uitvoeren van operaties binnen de processor. De Itanium, die in 2005 uitkomt

zal de eerste 64-bits processor zijn die Intel heeft ontworpen. Ook is de processor van het RISC type.

4.11 Vragen en opdrachten

1. (a) Wat is de taak van de processor (CPU)?
   (b) Waarom is de processor in een computer de belangrijkste chip?
2. (a) Wat is een instructieset?
   (b) Kan je een instructieset herschrijven?
3. (a) Leg het begrip endian uit.
   (b) Kan er worden gesteld dat little-endian systemen sneller zijn dan big-endian systemen? Licht toe.
   (c) Als de Intel processoren little-endian systemen zijn, kunnen klonen van die processor dan big-endian zijn?
   Intel is CISC, kan een AMD processor RISC zijn?
4. De waarde van het EFLAGS register kan veranderen door bepaalde instructies. Is het nuttig om de verandering van het gehele register in de gaten te houden? Licht toe.
5. Stel er moet met getallen worden gerekend die groter zijn dan 32-bit en daarom niet in de registers van de 386 passen, toch moeten de waarden worden opgeslagen. Noem enkele methodes om hiermee om te gaan.
6. Is het mogelijk om software die voor een Macintosh is gecompileerd te draaien op een Intel computer?
   Geef aan waarom dit (niet) kan.
7. Het produceren van hardware wordt steeds goedkoper, toch gebruiken de huidige processoren dezelfde technieken als oudere processoren. Waarom wordt er niet direct overgestapt op een totaal andere architectuur?
8. Bekijk het volgende programma.

```
section .bss
        var     resd    1

section .text

global _start
_start:
```

```
mov ecx, 0x41424344
mov [var], ecx  ; inhoud ecx in var
mov ecx, var    ; adres van var in ecx

mov eax, 4      ; sys call write
mov ebx, 1      ; 1e arg: fd van stdout
                ; ecx bevat adres var
mov edx, 4      ; lengte
int 0x80        ; call kernel

mov ebx, 0      ; 1e arg: exit code
mov eax, 1      ; sys call exit
int 0x80        ; call kernel
```

Compileer en link het bovenstaande voorbeeld, 'endian2.asm'. Run het programma en verklaar de output.
Waarom is de volgorde van het resultaat totaal anders dan verwacht?
(0x41 of 65 betekent een A)

Hoofdstuk 5

Registers

In dit hoofdstuk worden de verschillende typen registers behandeld en waarvoor ze dienen. Er wordt een inleiding gegeven in de werking van jumps, die het mogelijk maken om 'sprongen' binnen het code segment te maken, dit beïnvloedt de uitvoering van programma's.
In assembly zijn er twee representaties van gehele getallen, beide worden behandeld. Verder wordt er aandacht besteed aan rekenkundige bewerkingen en logische operatoren.

5.1 Geheugencellen

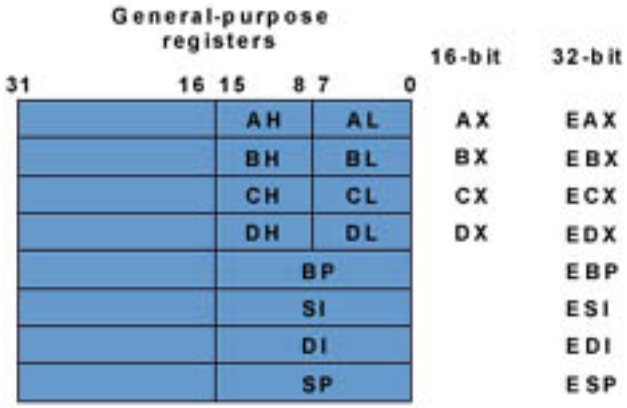
Registers zijn geheugencellen in een processor. Ze kunnen waarden aannemen of bevatten pointers naar data in het geheugen, deze data kunnen bijvoorbeeld instructies zijn voor de processor. Registers zijn meestal een veelvoud van een byte breed, bijvoorbeeld 16, 32, 64-bit. Zo ook bij de Intel 386 die registers van 32-bit heeft. Programmatuur krijgt een bepaalde werking door instructies die worden uitgevoerd door de processor. De instructies in assembly stellen mnemonic's voor en hun operands zijn registers, geheugenadressen of immediates.
Bijvoorbeeld:

```
push eax            ; eax is een register operand
mov ebx, 1          ; het getal 1 is een immediate operand
xor [memloc], edx ; memloc is een memory operand
```

Door instructies uit te voeren veranderen de waarden in de registers. Deze veranderingen kunnen worden getest, zo kan programmatuur een bepaalde volgorde van uitvoering krijgen.

5.2 Intel registers

Ondanks dat de 8088 8-bit registers had, waren er wel 16-bit mogelijkheden. De processor was zodanig ontworpen dat het om kon gaan met een 16-bit adresseringssysteem. Dit deed het door gebruik te maken van twee registers H en L. Beide registers waren 8-bit en vormden samen 16-bit. De 16-bit 8086 voegde een extra X aan de namen van de processor registers toe. De registers AX, BX, CX, en DX zijn paren van AH en AL, van BH en BL, van CH en CL, van DH en DL. De andere 16-bit registers, SP, BP, SI en DI, hebben geen X in hun namen en vormen daarom ook geen paren.



Figuur 5.1: *General-purpose registers van de Intel 386*

De *databus* van de 8086 was 16-bit en het kon 65.536 (2^{16}) locaties adresseren in een segment. De *adresbus* was 20-bit, dit was een truukje om meerdere segmenten te kunnen gebruiken op hetzelfde moment. Daarom was het geheugen niet helemaal gelimiteerd aan 64K, alleen is er geen manier om alle 2^{20} bytes tegelijk te adresseren.

Veel van de tekortkomingen aan de 8086 processor zijn verbeterd bij de 386. De acht 32-bit registers die beginnen met een E (extended) zijn uitbreidingen op de corresponderende 16-bit registers van de 8086. De databus en de adresbus van de 386 zijn beiden 32-bit dat betekent dat het 4.294.967.296 (2^{32}, 4GB) aan geheugenlokaties kan adresseren.

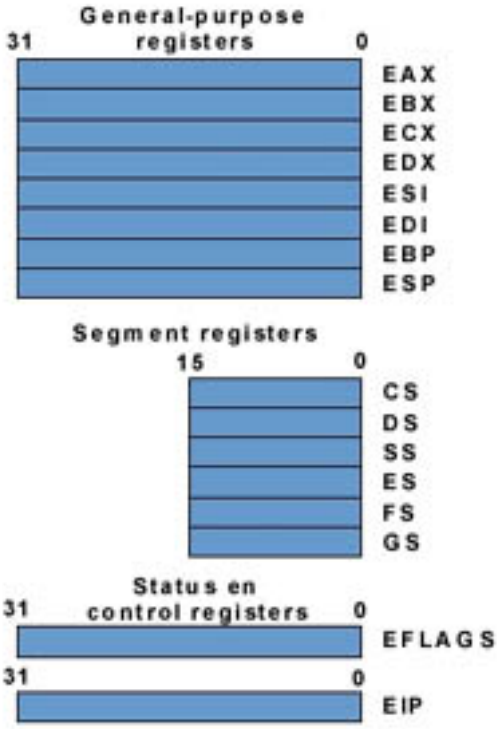
De Linux kernel biedt als operating systeem de mogelijkheid om 64GB aan geheu-

gen te adresseren, het maakt gebruik van 3-level paging dat deels wordt aangeboden door de *IA32* (= Intel Architecture, voor 32-processoren dus de 386 en later).

Eén van de redenen waardoor de Intel x86 lijn van processoren een groot succes werd, naast het feit dat een grote software fabrikant uit Redmond zijn software ervoor schrijft, is dat de processoren backwards compatible zijn. Software die draaide op voorgangers van de processoren, draait ook op nieuwere types.

De 32-bits registers van de 386 kunnen maximaal 2^{32} verschillende waarden aannemen (van 0x0000 0000 tot 0xFFFF FFFF). De 16-bit registers kunnen 2^{16} verschillende waarden aannemen (van 0x0000 tot 0xFFFF).

De 386 heeft 16 registers voor algemeen gebruik en voor het schrijven van software. Die zijn onder te verdelen in:



Figuur 5.2: *De registers van de Intel 386*

1. General-purpose registers. Deze acht registers worden gebruikt voor het opslaan van data en pointers. Ze dienen als operands voor instructies.

2. Segment registers. Deze registers bevatten de zes segment selectors.

3. Status en control registers. Deze registers veranderen van waarde bij bepaalde voorwaarden.

De 32-bit general-purpose data registers zijn EAX, EBX, ECX, EDX, ESI, EDI, EBP en ESP. Ze worden ondermeer gebruikt voor logische en rekenkundige bewerkingen, voor het berekenen van geheugen adressen en het opslaan van geheugen pointers. Omdat de 32-bit registers niet groot genoeg zijn om grote brokken data te bevatten worden er pointers (verwijzingen naar geheugen adressen) gebruikt naar data in het geheugen.

Ondanks dat alle general-purpose registers gebruikt kunnen worden, moet er toch erg voorzichtig worden omgegaan met het ESP-register. Het ESP-register bevat de stackpointer; het wijst naar het begin van de stack. Sommige instructies maken altijd gebruik van bepaalde registers. Zo maken string instructies gebruik van ECX, ESI en EDI.

Een kort overzicht van de general-purpose registers en het gebruik ervan:

1. **EAX** Accumulator voor operands en het opslaan van data.
2. **EBX** Pointer naar het DS segment (Base register)
3. **ECX** Counter voor string en lus (loop) operaties
4. **EDX** I/O (data) pointer
5. **ESI** Pointer naar de data in het segment waar DS naartoe wijst, de bron van string operaties (source index).
6. **EDI** Pointer naar data (of resultaat) in een segment waar ES naartoe wijst; wordt ook gebruikt als resultaat pointer voor string operaties (destination index).
7. **ESP** Stack pointer (in het SS segment)
8. **EBP** Pointer naar de data op de stack (in het SS segment) (Base pointer)

8-bit	16-bit	32-bit
AH	AX	EAX
AL	BX	EBX
BH	CX	ECX
BL	DX	EDX
CH	BP	EBP
CL	SP	ESP
DH	SI	ESI
DL	DI	EDI

5.3 Segment registers

De segment registers (CS, DS, SS, ES, FS en GS) bevatten 16-bit segment selectors. Een segment selector is een speciale pointer, die wijst naar een segment in het geheugen. In de protected mode wordt het geheugen verdeeld in segmenten. Om een bepaald geheugensegment te kunnen aanspreken moet het bijbehorende segment selector opgeslagen zijn in het register. Deze details zijn bij het programmeren niet van belang omdat de assembler [1] zelf met segmenten omgaat.

5.4 Instructie pointer

De *instructie pointer*, *EIP* bevat een *offset* in het huidige code segment. Met behulp van de selector uit CS (code segment) en de offset kan er naar een instructie worden gewezen. EIP is een register dat telkens wijst naar de volgende uit te voeren instructie. Wijzigingen aan het EIP register hebben dus direct invloed op de volgorde van uitvoering van het programma. Op sommige systemen wordt de *instructie pointer* aangeduid met *program counter* (*PC*).

5.5 Flags register

Het 32-bit *EFLAGS* register bevat een groep van *status flags*, een *control flag*, en *system flags*. Sommige van de flags kunnen direct worden gewijzigd, andere worden alleen door *special-purpose instructies* gewijzigd. Niet alle bits op het EFLAGS register worden gebruikt. Door het aan- of uitzetten van flags kan er gemakkelijk met voorwaarden worden omgegaan. Dit is het belangrijkste doel van het flags register. Het kan daarom ook niet als volledig 32-bits register worden gebruikt.

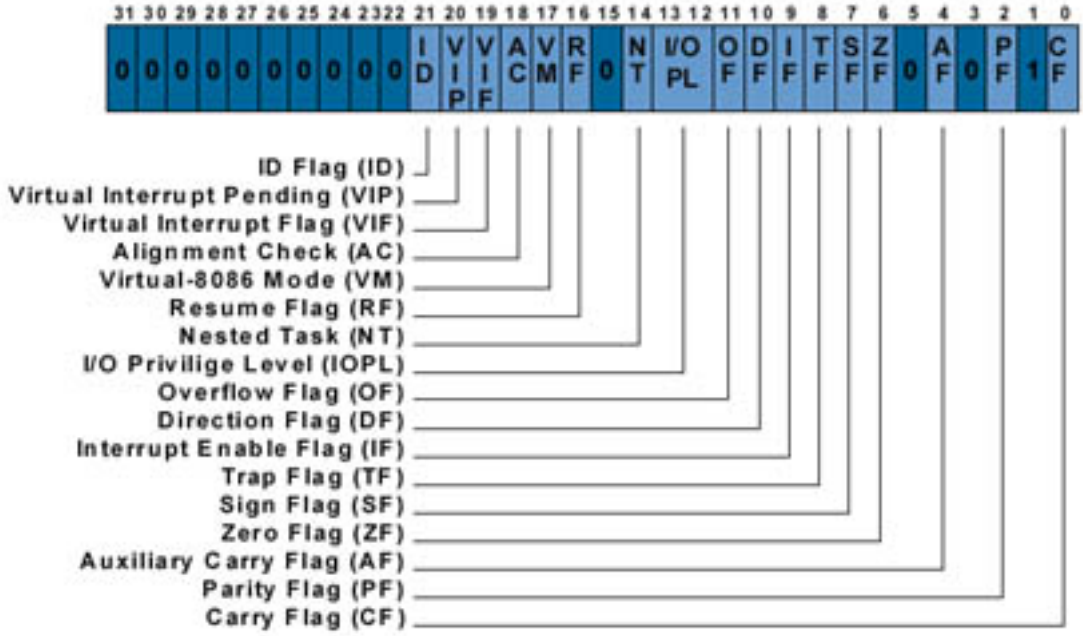
5.5.1 Carry flag

In assembly kan men voorwaarden testen aan de hand van het flags register. Toch heeft het flag register nog andere taken. Zoals het bijhouden van een eventuele carry. De mnenonic '*add*' en '*adc*' worden gebruikt bij het optellen van getallen in de registers.

De mnenonic '*add*' doet het volgende.

operand1 = operand1 + operand2

[1] Een assembler kan worden gezien als een assembly compiler, in dit geval NASM.



Figuur 5.3: *Het EFLAGS register*

De mnenonic '*adc*' werkt op een iets andere wijze. Het neemt namelijk de carry flag van een eventuele vorige opdracht mee.

operand1 = operand1 + carry flag + operand2

Stel er moet een 64 bits getal opgeteld worden. Dat bestaat uit EDX:EAX (het eerste deel staat in EDX, het tweede in EAX) en EBX:ECX.
De volgende regels assembly tellen EDX:EAX op bij EBX:ECX en schrijft het terug in EDX:EAX.
(EDX:EAX = EDX:EAX + EBX:ECX)

```
add eax, ecx
adc edx, ebx
```

Bij aftrekken werkt het principe hetzelfde, alleen wordt er gebruik gemaakt van '*sub*' en '*sbb*' voor de carry. De volgende code trekt EBX:ECX af van EDX:EAX.

De carry wordt dan gebruikt als 'borrow' flag door sub en sbb.

```
sub eax, ecx
sbb edx, ebx
```

Beschouw het volgende assembly programma, 'carry.asm'. Het demonstreert de werking van de CLear Carry (*CLC*) instructie.

```
section .text

global _start

_start:
    mov     eax,    0xffffffff
    mov     ebx,    1
    add     eax,    ebx     ; eax = eax + ebx, CF==1
    adc     eax,    2       ; eax = eax + 2 + CF

    clc

    mov     eax,    0xffffffff
    mov     ebx,    1
    add     eax,    ebx     ; eax = eax + ebx, CF==1
    add     eax,    2       ; eax = eax + 2

; exit
    mov     ebx,    0
    mov     eax,    1
    int     0x80
```

Om de werking van het programma te begrijpen dient het gecompileerd en gelinkt te worden en vervolgens in ALD te worden geladen.

5.6 Sections en segment registers

Assembly code bestaat uit één of meer onderdelen, die segmenten worden genoemd. De segmenten kunnen benaderd worden via de segment registers (CS, DS, SS, ES, FS en GS).

Hoe de segmentregisters precies worden gebruikt ligt aan het type *memory management model* dat het *operating system* gebruikt. In het geval dat er geen gebruik wordt gemaakt van *flat* of *unsegmented memory model* beginnen de adressen bij 0 en kan er lineair door worden gerekend. Een modern operating system zoals *Linux* maakt gebruik van een segmented memory model. Elk segment register bevat een andere segment selector dat verwijst naar een ander lineair geheugenadres. Omdat er zes segment registers zijn kunnen er zes segmenten worden geadresseerd.

5.6.1 Taken

Elk van de segment registers hoort bij code, data of stack. Bijvoorbeeld het CS-register, dat de *segment selector* voor het code segment bevat. In het codesegment staan de instructies van het programma. De processor haalt telkens een instructie uit het code segment. Dit doet de processor door het logische geheugenadres te vinden met behulp van de waarde in CS en EIP (het instructieregister). Het EIP-register bevat het lineaire adres van de uit te voeren instructie in het code segment van CS. Het is niet mogelijk het CS-register te wijzigen via de assembler; dit zou de loop van de software beïnvloeden. (Het operating system laat dit niet toe, anders zouden processen in elkaars geheugen adresseren en vervolgens toegang krijgen.) Het wordt intern door de processor gewijzigd (bijvoorbeeld door interrupt handling en taak verwisselingen).

De registers DS, ES, FS en GS wijzen naar de vier data segmenten. Er zijn vier data segment registers door deze te gebruiken kan er op een veilige manier met datastructuren worden omgegaan. Dit is een tegemoetkoming van de processor naar protected mode.
Zo kan software bijvoorbeeld tegelijk datastructuren van de huidige module gebruiken in het eerste segment. Data uit een ander segment, dat door een hogere programmeertaal gekoppeld is aan het assembly programma, kan gebruikt worden in het tweede segment. Een dynamische datastructuur kan als derde, en als vierde segment gebruikt worden.

Het SS register bevat de segment selector voor het stack segment, waar de stack wordt opgeslagen. Alle stack operaties maken gebruik van het SS register om data te vinden in het stack segment. Het SS register mag, in tegenstelling tot CS, wel worden gewijzigd in assembly. Dit maakt het mogelijk meerdere stacks te gebruiken en daartussen te switchen. (Deze techniek wordt zeer weinig gebruikt.)

5.7 Jumps

Jumps zijn assembly instructies die de loop van het programma veranderen. Jumps veranderen de *instructie pointer* (*EIP*) die naar code in het code segment wijst. Er zijn verschillende typen jumps, meestal vindt er voor een jump een test plaats.

Labels zijn punten in de code section (.text). In de vorige voorbeelden is al gebruik gemaakt van labels, namelijk '*_start*', dat het begin van het programma aangaf.
Een jump beïnvloedt het EIP door het te laten wijzen naar een punt ergens in het code segment. Met zogeheten *far jumps* kan overigens ook naar een ander adres in een ander code segment worden gesprongen.
Labels labelen punten; ze zijn dus niets meer dan een vervanging van labels in geheugenadressen binnen het code segment.

Het is de taak van de assembler (NASM) om de adressen van een label waarheen gejumpt wordt te berekenen. Labels worden vervangen door geheugenadressen in het code segment. Door executables in ALD te laden is dit goed te zien.

Het onderstaande voorbeeld springt meerdere malen naar het 'print_eax'-label. Eerst wordt de ascii waarde van het hexadecimale getal 0x30 [2] in het eax register geladen waarnaar het wordt vergeleken met 0x3a. Verschilt het, dan wordt er een jump gemaakt naar print_eax. Zo niet dan loopt het programma door en wordt exit uitgevoerd.

```
section .bss
        buffer resd 1           ; herschrijfbare buffer

section .text
global _start

_start:
mov     eax, 0x30               ; eax=0x30, ascii waarde 0

print_eax:
        mov [buffer], eax       ; waarde van eax in buffer

        mov eax, 4              ; write syscall
        mov ebx, 1              ; fd, stdout
        mov ecx, buffer         ; adres van buffer in ecx
        mov edx, 1              ; lengte
```

[2] 0x30 komt overeen met getal 0, zie ascii tabel in bijlage D

```
        int 0x80                ; kernel call, maak interrupt

        mov eax, [buffer]       ; inhoud v. buffer in eax
        inc eax                 ; increment eax

        cmp eax, 0x3a           ; vergelijk eax, met 0x3a
        jnz print_eax           ; als eax!=0x3a jmp print_eax

exit:
        xor ebx, ebx
        mov eax, 1
        int 0x80
```

De werking van de '*cmp*'-instructie komt in hoofdstuk 7 'Programma structuur' aan de orde. Het wordt gebruikt om te vergelijken en heeft dezelfde werking als *sub*, alleen brengt 'cmp' geen wijzigingen aan in de operands maar in het ELFAGS register.

5.8 Signed en unsigned integers

In het vorige voorbeeld werd er gebruik gemaakt van een jump. De jump werd gemaakt aan de hand van een getal in het EAX-register, dat overeen kwam met een waarde uit de ascii-tabel.
In assembly wordt vaak gebruik gemaakt van jumps aan de hand van voorwaarden met getallen, maar wat als die getallen een negatieve waarde bevatten? Dit kan vervelende gevolgen met zich meebrengen en daarom worden er twee soorten integers (gehele getallen) gebruikt: *signed integers* en *unsigned integers*.

Signed integers	Unsigned integers
zowel pos. als neg.	altijd positief
two's complement	normale representatie

5.8.1 Unsigned integers

Unsigned integers zijn integers die altijd positief of 0 zijn. Ze worden gebruikt in de normale representatie. Dat wil zeggen dat het decimale getal 171 in hexadecimaal 0xab voorstelt, en binair 1010 1011.

Een 16-bits register zoals het AX-register bevat dus getallen van 0 tot 65535. Het heeft een bereik van 2^{16} of 65536.

In C code worden unsigned integers gedeclareerd met 'unsigned int . . . '. Deze mogen, net als in assembly, alleen positieve waarden bevatten.

5.8.2 Signed integers

Om gebruik te maken van negatieve getallen moeten we kunnen aangeven dat getallen positief of negatief zijn. Er worden hier 3 *coderingsmethoden* uitgelegd; de laatste, *two's complement*, wordt in de huidige computers gebruikt, maar kan gemakkelijker begrepen worden als de twee voorgangers ook duidelijk zijn.

Bij **sign magnitude** wordt het eerste bit, het *sign bit*, gebruikt om aan te geven of een getal positief of negatief is.
Een 0 representeert een positief getal, een 1 een negatief getal. Op deze manier kunnen er met 8 bits (1 Byte) getallen tussen de -127 en +127 gevormd worden.

-127	**1**111 1111
+127	**0**111 1111

en

-88	**1**101 1000
+88	**0**101 1000

Er is bij deze methode een nadeel omdat 0 op twee manieren gevormd kan worden. Ook is het hardwarematig complex te implementeren.

Een andere methode is **one's complement**.
Bij deze methode wordt er ook gebruik gemaakt van een sign bit. Alle bits worden bij een negatief getal geïnverteerd. [3]

-88	**1**010 0111
+88	**0**101 1000

Het blijkt dat computers, gezien als elektronische rekenapparaten erg efficiënt kunnen omgaan met one's complement.
Toch is er nog steeds een probleem met de 0, er zijn nog steeds twee manieren om 0 te vormen, namelijk 1111 1111 en 0000 0000.

Deze problemen kunnen worden opgelost door gebruik te maken van **two's complement**.
Bij negatieve getallen werkt dit als volgt: inverteer alle tekens en tel er vervolgens 1 bij op.

[3] Inverteren kan worden gezien als het gebruik van de NOT operator.

Het grootste getal	+127	**0**111 1111
Het kleinste getal	-128	**1**000 0000

-88	**1**010 1111
+88	**0**101 0001

Het voordeel van deze coderingsmethode is dat er maar één manier is om 0 te vormen. Ook is er een nadeel omdat het het bereik van -127 tot +128 asymmetrisch is, maar computers hebben hier geen moeite mee. Dit is dan ook de reden dat het in de huidige computers wordt gebruikt.

In programma's waar gebruik wordt gemaakt van integers [4] moet worden uitgegaan van *signed integers* die dus zowel positief als negatief kunnen zijn en die gebruik maken van two's complement.

5.8.3 Two's complement omzetten

Er zijn gevallen dat men een normale representatie van integers als unsigned integers om wil zetten naar signed integers die gebruik maken van two's complement. Dit kan worden gedaan met de instructie '*neg*'. Neg zet de inhoud van de meegegeven operand om naar een two's complement getal. Het inverteert alle bits en telt er daarna één bij op.

In assembly zou '*neg*' zo gebruik kunnen worden.

```
neg ax                  ; ax wordt omgezet naar tc.

neg [inputfield]        ; de inhoud wordt omgezet naar tc.

neg edx                 ; edx wordt omgezet naar tc.
```

5.9 Rekenen

Computers zijn niets meer of minder dan complexe tel- en rekenmachines. Rekenen is nog steeds de hoofdtaak. De rekenkundige instructies hebben direct invloed op de waarden in de registers.

[4] In C worden hoeven signed integers niet worden aangeven, in tegenstelling tot unsigned integers.

5.9.1 Increment en decrement

Increment en *decrement* zijn beide rekenkundige instructies, die men gebruikt bij tellers en loops. [5]

- De increment instructie, '*inc*', telt één op bij de huidige waarde van het register dat als operand is meegegeven.
  Bijvoorbeeld:

```
mov ah, 12          ; ah=12
inc ah              ; ah=13          eigenlijk ah=ah+1
inc ah              ; ah=ah+1 of ah++

mov ebx, 99998
inc ebx             ; ebx++  -> ebx=99999
```

- De tegenhanger van increment is decrement, '*dec*', deze heeft een tegenovergestelde werking en trekt één af van de huidige waarde in het register.

```
mov dl, 88          ; dl=88
dec dl              ; dl--      dl=87
```

5.9.2 Vermenigvuldigen en delen

Voor het rekenen met gehele getallen (integers) moet er in assembly rekening worden gehouden met de twee verschillende representaties van integers. Daarom zijn er bepaalde rekenkundige instructies in tweevoud.

Vermenigvuldigingen worden gemaakt met de '*mul*'-instructie. De 'mul'-instructie voert unsigned integer vermenigvuldigingen uit. Het wordt gebruikt als 'mul operand', waarbij operand een geheugenadres of register kan zijn. Er zijn meerdere mogelijkheden omdat de operands van grootte kunnen verschillen.

- **mul r/m8, AL** Hier wordt de meegeven operand van 8 bits vermenigvuldigd met AL (8 bit). Het resultaat wordt in AX (16 bit) opgeslagen.

[5] Meer over loops in het hoofdstuk 7 Programma structuur.

- **mul r/m16, AX** De operand van 16 bit wordt vermenigvuldigd met AX (16 bit). Het resultaat wordt opgeslagen in DX:AX.
- **mul r/m32, EAX** Hierbij wordt de operand van 32 bit vermenigvuldigd met EAX (32 bit). Het resultaat wordt in EDX:EAX opgeslagen.

Compileer en link het onderstaande assembly voorbeeld.
Laadt het programma in ALD en bekijk het resultaat. Ga na wat er gebeurt bij de vermenigvuldigingen en welke registers erbij betrokken zijn.

```
section .data
        getal1 db 0x8a
        getal2 db 0x6b
        getal3 dw 0x4141
        getal4 dw 0x4545
        getal5 dd 0x8a8a8a8a
        getal6 dd 0x9c9c9c9c

section .text
global _start
_start:

        nop
        mov al, [getal1]
        mul byte [getal2]       ; AX=getal2*getal1

        mov ax, [getal3]
        mul word [getal4]       ; DX:AX=getal4*getal3

        mov eax, [getal5]
        mul dword [getal6]      ; EDX:EAX=getal6*getal5

exit:
        mov eax, 1
        xor ebx, ebx
        int 0x80
```

Het voorbeeld demonstreert hoe de vermenigvuldiging van unsigned integers gaat in assembly. Het vermenigvuldigen van signed integers gaat op een andere wijze. Hiervoor wordt de '*imul*'-instructie gebruikt (NASM doc. A.76).

Het delen in assembly gebeurt op bijna dezelfde wijze als vermenigvuldigen. Voor meer informatie over delen en de manier waarop dit gebeurt zie de documentatie bij NASM section A.25 over de '*div*'-instructie voor unsigned integers en de '*idiv*'-instructie voor signed integers.

5.10 Logische operatoren en maskers

Bij het vorige voorbeeld, dat een jump demonstreerde, staat op regel 26 '*xor* ebx, ebx', terwijl in de andere voorbeelden '*mov* ebx, 0' stond. In dit geval is de uitwerking van de instructies gelijk, namelijk het laden van ebx met het getal '0'.

Assembly kent de volgende logische operatoren: *XOR, OR, AND en NOT*. Ze worden meestal gebruikt bij het toepassen van maskers.

5.10.1 AND

Het resultaat van de *AND operator* bij twee bits is alleen een 1 als beide 1 zijn.

Hieronder staat de waarheidstabel van de AND operator.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

- Met AND maskers kan er getest worden op bepaalde bits. Zo kan de AND operator worden gebruikt bij het isoleren van bits.

  1111 0001
  1000 0000 AND

  1000 0000

  Dit kan als volgt in assembly worden gebruikt:

```
mov     al,     0xf1    ; al is een 8bits register
and     al,     0x80    ; al is nu 0x80
```

  Het resultaat is dat het AL-register de waarde 0x80 bevat. Om dit beter te testen kan men er een assembly programma van maken en het laden in een debugger.

- Bij dit voorbeeld wordt hetzelfde AND masker gebruikt en het geeft aan dat het MSB bit een 0 is.

```
0111 0001
1000 0000 AND
-------------
0000 0000
```

In assembly had dit er er zo uit kunnen zien.

```
mov     bl,     0x71    ; bl bevat 0x71
and     bl,     0x80    ; pas een masker van 0x80 toe
```

Na het draaien van het voorbeeld bevat het BL-register de waarde 0000 0000.

5.10.2 OR

De logische *OR operator* wordt vaak gebruikt om individuele *bits* te veranderen zonder een bewerking uit te voeren over de andere bits.

De logische OR heeft altijd als uitkomst een 1 als één van de te testen bits 1 is. Zijn beide een 0 dan is de uitkomst een 0, anders geeft de OR altijd een 1.

Zie hier de waarheidstabel van de logische OR.

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

- Het volgende voorbeeld demonstreert het gebruik van de OR operator, die gezien kan worden als het tegengestelde van de AND operator.

```
0010 1011
1001 0010 OR
-------------
1011 1011
```

Als we dit naar assembly omzetten wordt het gelijkwaardig met.

```
mov     cl,     0x2b    ; cl=0x2b
or      cl,     0x92    ; cl is hierna 0xbb
```

Door een logische OR van 0x92 op het CL-register toe te passen dat 0x2b bevat, verandert de waarde in 0xbb.

5.10.3 XOR, exclusive OR

De logische *XOR operator* is gelijk aan de OR, maar is er één verschil. Een XOR van 2 bits is alleen 0 als beide bits gelijk zijn, anders is het resultaat 1.

Zie hier de waarheidstabel van de exclusive OR.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

- In het volgende voorbeeld wordt een demonstratie gegeven van het gebruik van de XOR.

```
0010 1010
1001 1111 XOR
-----------
1011 0101
```

In assembly zou dit er zo uitzien.

```
mov     dl,     0x2a    ; dl=0x2a
xor     dl,     0x9f    ; xorren van dl met 0x9f
```

Na het draaien van het stukje assembly code bevat het DL-register de waarde 0xb5. Dit omdat 0x2a xor 0x9f gelijk is aan 0xb5.

- De XOR kan ook gebruikt worden om bits om te draaien, 0 naar 1 en 1 naar 0.

```
1010 1010
1111 1111 XOR
-----------
0101 0101
```

- In de sourcecode die een jump demonstreert, wordt gebruik gemaakt van een de XOR in plaats van de 'mov'-instructie om een register de waarde 0 te geven. Een XOR masker van twee dezelfde waarden heeft als gevolg 0 (zie de waarheidstabel).

  Dit is een veel voorkomend iets in assembly, het volgende voorbeeld demonstreert dit. Het is een logisch gevolg als je bedenkt dat de XOR altijd een 0 geeft als beide bits gelijk zijn.

  1110 0101
  1110 0101 XOR

  0000 0000

  Omgezet in assembly ziet het er zo uit.

```
mov al, 0xe5     ; al=0xe5
xor al, 0xe5     ; xorren van al met 0xe5
```

  Of direct via de registers met.

```
xor al, al       ; al=0
```

  Het AL-register is na de instructie 'xor al, al' altijd 0.

5.10.4 NOT

De *NOT operator* verschilt van de andere operators. Het maakt gebruik van één operand en wordt daarom ook wel een *unary operator* genoemd.

De NOT van een bit is de tegenovergestelde waarde van de huidige bit.

De waarheidstabel.

A	NOT A
0	1
1	0

- Een voorbeeld van het gebruik van de NOT operator.

  1000 1000 NOT

  0111 0111

In assembly code ziet het er zo uit.

```
mov     cl,     0x88    ; cl=0x88
not     cl              ; cl is nu 0x77
```

Het CL-register bevat na de instructie 'not cl' de waarde 0x77 in plaats van 0x88.

De 'not'-instructie wordt ook gebruikt bij het omzetten van getallen naar one's complement.

5.10.5 TEST

De instructie '*test*' wordt op dezelfde manier gebruikt als de AND operator en heeft dezelfde waarheidstabel, maar het wijzigt de registers die als operands worden gegeven niet. Het verandert alleen het EFLAGS register.

De 'test'-instructie heeft meerdere toepassingen, hier een korte demonstratie ervan. In dit voorbeeld wordt *test* gebruikt om te zien of het eerste bit in het CL-register gelijk is aan 1. Is het gelijk aan 0 dan wordt het ZF (zero flag) op 1 gezet. Omdat 0xf0 AND 0x80 een 1 oplevert en ZF daarom 0 is, wordt er niet gejumpt naar 'end'. De '*jz*' [6] jump wordt alleen gemaakt als JZ (Zero Flag op het EFLAGS register) een 1 is.

```
section .data
    foo db "test-test", 0xa
    len equ $ - foo

section .text
global _start
_start:
    mov cl, 0xf0    ; 11110000
    test cl, 0x80   ; 10000000

; 1111 0000
; 1000 0000 AND
; ---------
```

[6] In hoofdstuk 6 wordt er meer informatie gegeven over jumps.

```
; 1000 0000
;
; Dit is ongelijk aan 0
; Zero Flag (ZF) = 0

; jz jumpt als ZF een 1 is.
; ZF = 0 dus geen jump
    jz end

; druk foo af
    mov eax, 4
    mov ebx, 1
    mov ecx, foo
    mov edx, len
    int 0x80

end:
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

5.11 Vragen en opdrachten

1. (a) Is het mogelijk om registers in een processor direct te wijzigen?
   (b) Leg uit wat extended registers zijn.
   (c) Hoeveel bits zitten er in een byte, een word en een double word?

2. (a) Waar dient het CS-register voor?
   (b) Wat is de instructie pointer en waar wijst het naartoe?
   (c) Kan het EIP-register door bijvoorbeeld de 'mov'-instructie worden gewijzigd? Waarom wel/niet?
   (d) Op welke manier werkt de instructie pointer?

3. (a) Waarom bestaat assembly sourcecode uit meerdere segmenten/sections?
   Maak een link naar het geheugenmodel.
   (b) Wat is het nut van segment registers?
   (c) Kan een assembly programmeur de waarden in bijvoorbeeld FS of CS wijzigen? Laat de assembler, in dit geval NASM, dit toe?
   Probeer dit uit.

4. Waarom kan linux 64GB aan geheugen adresseren, terwijl dit niet mogelijk is als er naar de breedte van de registers wordt gekeken?

5. Leg uit waarom het handig is om registers in twee delen te splitsen, een 'low', L deel en een 'high', H deel?

6. Stel dat de waarde van AX wordt veranderd. Heeft dit gevolgen voor EAX en voor AH?

7. (a) Om een bepaalde structuur te krijgen binnen een programma moeten er voorwaarden worden getest. Hoe wordt dit gedaan in assembly?
   (b) Beschrijf de werking van de carry flag op het EFLAGS register.
   (c) Wat doet de 'clc'-instructie? Wat voor nut heeft het? Geef een sourcecode waarin een demonstratie staat met de werking ervan.
   (d) ALD laat de hexadecimale waarde zien van het EFLAGS register. Waarom is dit?

8. Software die geschreven is voor de 8086 draait ook op de 386. Leg uit waarom dit mogelijk is.

9. De registers in een processor zijn direct verantwoordelijk voor wat er in de computer gebeurt. Leg uit waarom het wijzigen direct invloed heeft op de handelingen van de computer.

10. (a) Wat is de functie van de general-purpose registers?
    (b) Mogen de general-purpose registers voor andere taken worden gebruikt dan waarvoor ze zijn bedoeld?

11. (a) Hoeveel bit is het AX-register?
    (b) Hoeveel bit is het AL-register; en het EAX-register?
    (c) Is het BX-register in de 8086 even lang als het BX-register in de 386?

12. Wat is het nut en de uitwerking van de nop instructie? Gebruik hierbij de NASM manual.

13. De volgende sourcecode is ongeldig. Er wordt de volgende foutmelding gegenereerd: 'invalid combination of opcode and operands'.

```
section .text

global _start
_start:

    mov bx, 0x555
    mov eax, bx

end:
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

    (a) Geef de reden van de ongeldigheid.
    (b) Vervang regel 6, 'mov eax, bx' door 'movzx eax, bx'.
    (c) Compileer en debug het programma.
    (d) Geef een toelichting over de mnemonic 'movzx' [7] .

[7] Voor meer informatie over over 'movzx' zie de NASM handleiding.

14. (a) Geef van alle vier de verschillende operatoren een voorbeeld en beschrijf het nut er van.

    (b) Wat is gevolg van 'xor edx, edx'? Welke waarde bevat het EDX-register na deze instructie.

15. Vul de ontbrekende waarden in.

    (a)
    ```
      1010 1101
    1110 0101 AND
    ---------
    ...
    ```

    (b)
    ```
      1111 1101
    1010 0111 OR
    ---------
    ...
    ```

    (c)
    ```
      1010 1101
    1010 0000 XOR
    ---------
    ...
    ```

    (d)
    ```
    1100 1101 NOT
    ---------
    ...
    ```

16. (a) Waarom zijn er twee type integers? Geef ook aan welke coderingsmethode beide gebruiken.

    (b) Leg uit hoe signed integers in moderne computers worden gebruikt. Licht kort het sign bit toe.

    (c) Zet het two's complement getal 1100 1101 om naar een decimaal getal.

    (d) De 'neg'-instructie zet getallen unsigned integers om naar signed integers in two's complement. Bestudeer de werking van 'neg' en schrijf zelf een programma dat een normale gerepresenteerde waarde van een register of geheugenadres omzet naar two's complement. (hint: Gebruik ALD om het programma te testen en lees de bondige documentatie over 'neg' in de NASM handleiding.)

17. Het delen van getallen gaat op bijna dezelfde wijze als het vermenigvuldigen in assembly. Schrijf een programma dat deelt aan de hand van 8, 16 en 32-bit getallen. Deze getallen zijn unsigned integers en er moet gebruik gemaakt worden van de '*div*'-instructie. Voor meer informatie zie de NASM handleiding over 'div'.

Hoofdstuk 6

Taal niveaus

Dit hoofdstuk beschrijft de verhouding van assembly ten opzichte van High Level languages. Er wordt beschreven hoe assembly is ontstaan en waarom men nu vooral in hogere programmeertalen programmeert. HLL en assembly zijn te combineren en hier wordt dan ook een kort voorbeeld van gegeven.
Ook wordt er aandacht besteed aan assemblers, in dit geval NASM en de mogelijkheden ervan, zoals het schrijven van macro's.

6.1 Assembly language

Een processor 'verstaat' strikt gezien alleen maar machinetaal. Deze taal is niet bruikbaar om in te programmeren. Er zou dan met geheugenadressen en instructies als numerieke waarden gewerkt moeten worden. Om toch op low-level gebied te kunnen programmeren is er een *assembly language*. Assembly zorgt ervoor dat er direct met de processor gecommuniceerd kan worden, maar wel via een taal die voor mensen leesbaar is. Elke instructie kan naar mnemonics worden omgezet. Iedere assembly *mnemonic* komt overeen met een *instructie*. Dit is de reden dat assembly machinegericht is. Er zijn veel verschillende processor architecturen met elk een eigen assembly taal.

6.2 Low-level languages

In de jaren '50 werd de assembly language ontwikkeld, omdat het niet meer mogelijk was om in machinetaal te programmeren; de software werd te complex en bevatte te veel regels code. Alle codes werden geprogrammeerd in numerieke waarden. Assembly loste dit op door het gebruik van mnemonic's; dit zijn woorden die overeenkomen met machinetaalinstructies.

Mnemonic's zijn veel gemakkelijker te onthouden dan numerieke waarden. Zo is er de mnemonic voor optellen, 'add' en voor aftrekken 'sub'. Als bijvoorbeeld de waarde in het register EAX en de waarde in EBX opgeteld moeten worden en weer opgeslagen worden in EAX, dan zou dit in machinecode er bijvoorbeeld als volgt uit zien: 03 03. In assembly kan het zo worden geschreven: 'add eax, ebx'.

Assembly languages maken gebruik van assemblers. Dit is software die assembly sourcecode omzet naar machinetaal. Een processor heeft een unieke set aan machinetaalinstructies, dit is de reden dat het ook een eigen assembly language heeft. Dit heeft tot gevolg dat assemblyprogrammatuur alleen geschreven kan worden voor een bepaald type processor.

Assembly komt programmeurs tegemoet door mnemonic's te gebruiken voor instructies in plaats van numerieke representaties. Maar nog steeds bestaat een zeer simpele handeling uit vele instructies. Dit zorgt ervoor dat het programmeren in assembly veel tijd kost. Ondanks dat sommige programmeurs gebruik maken van vele bibliotheken met voorgeprogrammeerde routines, neemt dit niet het basisprincipe van assembly weg. Er moet nog steeds veel tijd aan de werking van de routines worden besteed.

6.3 High Level Languages

Op een hoger niveau staan de hogere programmeertalen, HLL (High Level Languages), die probleem gericht werken. Deze talen zijn niet machine afhankelijk. De operaties uit de hogere programmeertalen zijn gericht op problemen die vaak voorkomen. Denk hierbij aan tellers en andere routines voor voorwaarden en programma flow. De problemen worden opgelost zonder specifieke machinetaal instructies te gebruiken.

Als er code is geschreven in een hogere programmeertaal is het de taak van een

compiler om die om te zetten naar machinecode. Er wordt op deze manier een uitvoerbaar binaire file gevormd met machinetaal. De compiler voor de hogere programmeertaal leest de sourcecode en vertaalt het naar assembly die precies doet wat er in de hogere programmeertaal was beschreven. De assembly code bestaat nu uit veel meer regels code dan de oorspronkelijk HLL sourcecode.

High Level Languages zijn ontwikkeld in de jaren rond 1955. De eerste taal was FORTRAN, gevolgd door Algol, Cobol en Lisp. Bekende moderne high-level languages zijn Pascal, C, C++. Deze high-level languages maken gebruik van compilers. Dit is software die de sourcecode van HLL omzet in machinetaal afhankelijke objectcode. Scripting en Virtual Machine worden hier buiten beschouwing gelaten, omdat ze indirect naar machinecode worden omgezet.

Strikt gezien is C geen HLL omdat het in C ook mogelijk is om low-level taken uit te voeren. Zo ondersteunt C indirect het gebruik van registers, low-level geheugenmanipulatie, goto-statements en het schuiven van bits en dergelijke. In dit dictaat wordt C gebruikt als HLL om bijvoorbeeld while- en for-statements te demonstreren in assembly.

6.4 Samenwerking tussen C en assembly

Het is mogelijk om programmatuur te schrijven die deels is geschreven in een HLL zoals *C of C++* en in assembly. Omdat *C* het gebruik van *functies* aanbiedt kunnen we sommige functies (*routines*) in assembly schrijven. De functies geschreven in assembly kunnen sneller en efficiënter zijn dan die in een HLL geschreven zijn.

In de *Linux kernel* gebeurt hetzelfde, de machineafhankelijke taken worden uitgevoerd door assembly functies die vanuit de C code worden aangeroepen. Omdat een groot deel van de kernel op deze manier in C is geschreven is het in zeer hoge mate te porten naar andere architecturen. (Alleen de machine afhankelijke assembly functies hoeven herschreven te worden.)

Het volgende (zeer korte) voorbeeld demonstreert een functie geschreven in assembly die samenwerkt met C. Omdat er mnemonics worden gebruikt die nadere uitleg nodig hebben van begrippen die pas in de volgende hoofdstukken worden uitgelegd is het voorbeeld beperkt.

De inhoud van de de file '*asm_code.asm*'.

```
section .text
global asmcode
```

```
asmcode:
        enter   0,0

        mov     eax,    0x12

        leave
        ret
```

De sourcecode van '*c_code.c*'

```
#include <stdio.h>
/*
        prototypen van de asm functie
        omdat anders de compiler de
        functie niet kent.
*/
extern int asmcode(void);

int main(void) {
int getal=0;

        getal=asmcode();
        printf("%d\n",getal);

        return 0;
}
```

Compileer de assembly functie met NASM met 'nasm -f elf asm_code.asm'. Dit zorgt ervoor dat er objectcode in asm_code.o komt te staan. Deze objectcode kan door de C compiler (GCC) worden gelinkt met een C programma.
Dit gebeurt door 'gcc asm_code.o c_code.c -o c_asm' te geven.
GCC kan zowel objectcode (asm_code.o) als C code (c_code.c) omzetten naar een executable. Het resultaat, een executable, wordt door de optie '-o' naar de file 'c_asm' geschreven.
Deze kan uitgevoerd worden en het resultaat is 18, omdat 0x12 in decimaal 18 is.

Alle commando's nog eens in het kort:

nasm -f elf asm_code.asm
gcc asm_code.o c_code.c -o c_asm

./c_asm

Dit voorbeeld demonstreert dat het mogelijk is om vanuit de ene taal functies aan te roepen uit de andere taal.
De assembly code wordt in dit voorbeeld aangeroepen vanuit de de 'main' functie van het C programma. Het resultaat van het assembly programma, de return waarde, wordt in het EAX-register gezet en wordt in de C code in de variabele 'getal' gezet. Vervolgens wordt deze naar het scherm geschreven.
In printf staat %d om het getal af te drukken; dit zorgt ervoor dat de decimale waarde van het getal wordt gebruikt. (Er wordt geen gebruikt gemaakt van de ascii waarden.)

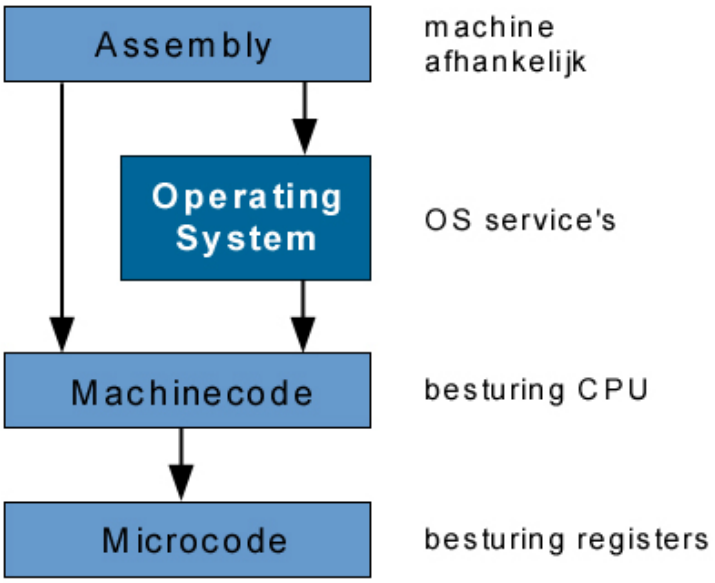
6.5 Low-level power!

Assembly geeft de programmeur directe controle over de computer. Dit is de voornaamste reden dat mensen geïnteresseerd zijn in assembly.
Voor de mensen die computers alleen maar gebruiken om software op te draaien is software zeer abstract. Programmeurs willen het gedrag van een computer kunnen beïnvloeden en omdat assembly platform afhankelijk is, is dit geen gemakkelijke taak.

De 386 is historisch gezien van redelijk groot belang. De huidige generatie processoren van Intel zijn allemaal gericht op die 32-bits processor. De assembly language van de 386 is een standaard geworden op assembly gebied. Dit omdat de Intel 386 vastgeroest zit aan assembly programmeren. Omdat de 386 gebruik maakt van *protected mode* kunnen er krachtigere applicaties worden geschreven die ook nog eens 32-bit zijn.

Assembly kent geen variabelen of andere HLL principes. Toch biedt het de beste middenweg tussen een machinetaal en een HLL. Door het gebruik van macro's en subroutines kan 1 regel veranderen in duizenden bytes aan machinetaal.

Er zijn legio voordelen die pleiten voor assembly. Maar het meest voor de hand liggend is het toegang hebben tot het lage niveau van de computer, en de totale controle over de CPU. Software die is geschreven in HLL is meestal trager dan assembly software, omdat een compiler standaard methoden gebruikt voor lezen en schrijven, het gebruik van variabelen en het aanroepen van routines uit eerder gecompileerde software. Een goed voorbeeld hiervan is het gebruik van de stack. Functies in HLL maken veelal gebruik van het stack principe. Code geschreven in een HLL, kan er wiskundig correct en efficiënt uitzien, maar kan op een totaal onhandige wijze omgaan met een stack. Dit kan software onnodig langzaam maken.



Assembly biedt de mogelijkheid om absoluut optimaal doelmatige software te schrijven. Het is bij assembly namelijk mogelijk om software te maken die minder instructies gebruikt dan HLL, daarom zijn assembly programma's meestal kleiner en sneller.

De nadelen van assembly zijn: een verhoogde kans op fouten in de software (bugs), het is processor afhankelijk, er zijn geen library-routines voor veel voorkomende taken, zoals het afdrukken van tekenreeksen of het lezen van gegevens. De bugs in software zijn een gevolg van onzorgvuldigheid; dit staat los van het gebruik van de programmeertaal. Het is zeker mogelijk om bugvrije software te schrijven in assembly. Toch is het zo dat een bug in assembly sourcecode meer invloed heeft dan bij HLL. Om dit te voorkomen is het belangrijk assembly code goed te ontwerpen en te debuggen.
Assembly wordt direct omgezet naar machinetaal en het is daarom vanzelfsprekend dat het afhangelijk is van de gebruikte processor. Dit maakt het porten van software een zware tot onmogelijke taak.
Ook biedt assembly geen mogelijkheden zoals voorgeschreven functies voor berekeningen en strings. Dit kan vervelend lijken, maar in werkelijkheid heeft het echter grote voordelen. Dit heeft tot gevolg dat assembly code transparanter en doelmatiger kan worden geschreven.

6.6 NASM

Assembly sourcecode wordt door een assembler omgezet naar machinecode. NASM is een assembler en staat voor Netwide Assembler. Het is een assembler voor de Intel 80x86 en die zo is ontworpen dat het gemakkelijk te porten is naar meerdere operating systems. Ook ondersteunt het verschillende soorten formaten, zoals bin, aout, coff, elf, as86, obj (DOS), PE (win32), rdf (het eigen formaat van NASM). NASM maakt gebruik van de Intel syntax, net als TASM en MASM. TASM en MASM zijn beiden geen opensource en draaien niet op Unices, tenzij in een DOS-emulator. [1]

De andere assembly syntax is de AT&T syntax, die de statements net andersom interpreteert als Intel syntax. De gebruikte syntax is niet belangrijk voor het leren van de principes van assembly; het is alleen van belang bij het programmeren. GAS is een AT&T syntax assembler van GNU. Het is eigenlijk een back-end van GCC en is daarom niet echt geschikt als assembler. Toch maakt GNU gebruik van de AT&T syntax. Dit kan nogal vervelend zijn bij het gebruik van andere GNU tools, zoals het debugging programma GDB, waarbij alles in AT&T stijl staat. Er bestaan software tools die de ene syntax omzetten de andere. Meestal werken deze tools niet optimaal en is kennis van beide vereist.

6.6.1 NASM macro's

Met NASM is het mogelijk om gebruik te maken van macro's. Macro's maken het mogelijk om bijvoorbeeld files te includen en mechanismen te programmeren voor taken die vaak voorkomen. Macro's kunnen worden gezien als preprocessor opdrachten.

In het onderstaande voorbeeld bevat 'syscall.mac' alle assembly instructies die een write en een exit system call mogelijk maken.
Het bestand 'macro.asm' maakt gebruik van de macro's door de file 'syscall.mac' te includen en de code ervan te gebruiken.

De inhoud van het bestand 'syscall.mac'.

```
%macro write 3
; gebruik: write fd message msglength
        mov eax, 4
        mov ebx, %1
        mov ecx, %2
```

[1] http://www.dosemu.org

```
        mov edx, %3
        int 0x80
%endmacro

%macro exit 1
; gebruik: exit status
        mov ebx, %1     ;error status, 0 is success
        mov eax, 1
        int 0x80
%endmacro
```

Zie hieronder de inhoud van 'macro.asm'. Dit bestand kan net als andere assemblyprogramma's worden gecompileerd en gelinkt worden. Macro's worden door NASM geïnterpreteerd, ze hebben geen uitwerking op de executable die door compileren en linken wordt gemaakt.

```
%include "syscall.mac"

section .data
msg db "Write␣syscall␣using␣macro's", 0xa
len equ $ - msg

section .text
global _start
_start:
    write 1, msg, len
    exit 0
```

Voor meer informatie over NASM macro's zie chapter 4 van de manual.

6.7 Vragen en opdrachten

1. Machinecode bevat de instructies voor de processor. Bevat de door de assembler gegenereerde machinecode nog afhankelijke delen? Delen die nog niet berekend en ingevuld konden worden zonder het programma te draaien? Licht je antwoord toe.

2. Wat is het verschil tussen machinetaal en assembly?

3. Wat is het verband tussen machinetaal en assembly?

4. Waarom kan in assembly geprogrammeerde software moeilijk worden geport?

5. Geef twee redenen waarom er in HLL wordt geprogrammeerd door de meeste programmeurs.

6. Geef een voorbeeld van een programmeerproject dat het best in assembly kan worden uitgevoerd en een voorbeeld van een project dat beter in een HLL kan worden uitgevoerd. Onderstreep hierbij de voordelen en nadelen van assembly. Zou men hierbij ook een combinatie tussen HLL en assembly kunnen kiezen?

7. Leg uit waarom het mogelijk is om vanuit een assembly programma een C functie aan te roepen of vise versa. Gebruik hierbij de term objectcode.

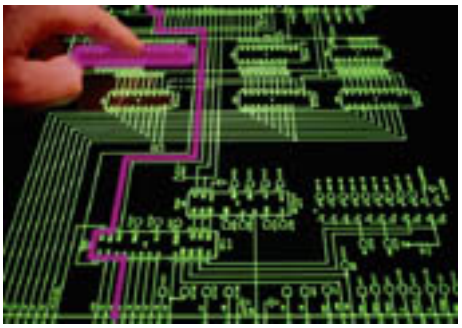
8. Tekenreeksen zijn in C character arrays. Waarom bevatten ze een 0 aan het einde van de character array (string)? Maak de terugkoppeling naar assembly niveau.

Hoofdstuk 7

Programma structuur

In de vorige hoofdstukken is er een korte uitleg gegeven over de manier waarop de processor omgaat met instructies en de manier waarop deze worden uitgelezen uit het code (.text) segment. Ook is er aandacht besteed aan jumps om de volgorde van executie te beïnvloeden. In dit hoofdstuk wordt hier dieper op ingegaan.

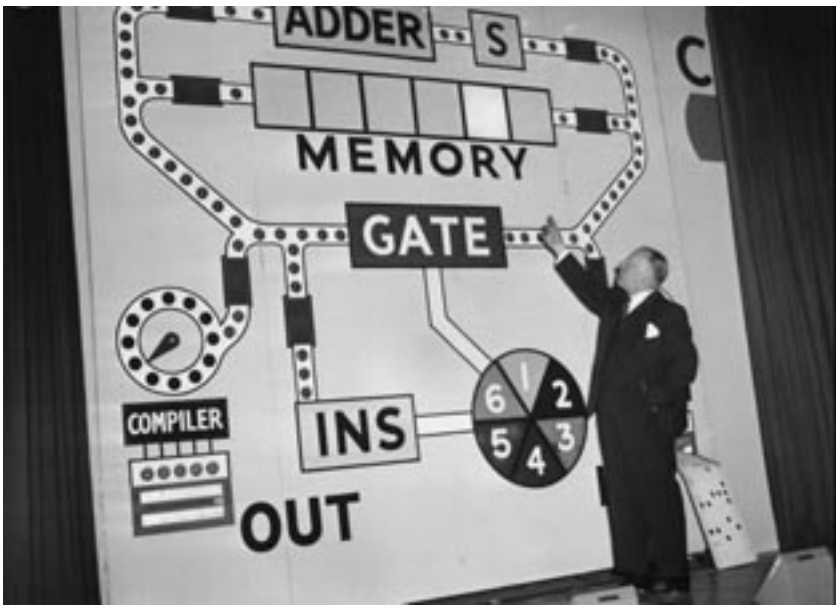
In *HLL* (*High Level Language*) zijn er statements zoals bijvoorbeeld *if*, *else* en *while*. Deze worden toegepast om op een gemakkelijke manier om te gaan met voorwaarden en programma structuur. In assembly ligt dit anders; er wordt gebruik gemaakt van jumps en het testen van bepaalde bits op het *EFLAGS register*. Dit gebeurt aan de hand van vergelijkingen door de '*cmp*'-instructie.



Een HLL zoals C maakt gebruik van functies die gezien kunnen worden als jumps, maar die na het uitvoeren van de code binnen de functie weer de oude programmagegevens kan bevatten. Dit gebeurt met gebruik van een speciaal deel binnen het geheugen, de stack. In dit hoofdstuk wordt hiervan een korte beschrijving gegeven.

7.1 Instructie executie

Computers kunnen met de uitvinding van *Von Neumann* instructies intern opslaan. De instructies worden bewaard in het geheugen (RAM) en uitgevoerd door de processor. Op deze manier kan men programma's zien als een lijst van instructies, die door de processor worden uitgevoerd. Het uitvoeren en uitlezen van instructies



Figuur 7.1: *Een simpel uitgevoerde computer*

wordt '*fetch-execute cycle*' genoemd en bestaat uit drie stappen. [1]

1. De processor haalt een instructie op uit het geheugen.
2. De processor voert de instructie uit.
3. Ga naar 1

Bijna alle computers zijn gebaseerd op deze methode met uitzondering van *DNA computers* en *quantum computers* die alleen nog maar in onderzoekscentra worden gebruikt.

[1] De werkelijke cyclus is de 'Von Neumann-cyclus', die uit vijf stappen bestaat, maar omdat niet alle details direct van belang zijn bij het schrijven en begrijpen van assembly worden ze hier buiten beschouwing gelaten.

7.2 Structuur door vergelijking

Assembly kent geen geavanceerde methoden om de volgorde van executie te veranderen zoals bijvoorbeeld C met if- en for-statements.
In assembly wordt er gebruik gemaakt van een elementaire manier om met *structuur* om te gaan. Deze manier kan worden gezien als het *goto* [2] statement.

Om *structuur* aan te brengen in programma's moeten er data vergeleken worden en moet aan de hand hiervan een beslissing worden gemaakt. Het vergelijken in assembly gebeurt aan de hand van de '*cmp*'-instructie. Als resultaat, het verschil tussen de twee operands van 'cmp', zet het bepaalde flaggen op het *EFLAGS* register aan of uit.

Het gebruik van de 'cmp'-instructie is als volgt:

cmp links, rechts

Links en rechts zijn operands en kunnen worden vervangen door een getal en/of een register. [3]
Het verschil van links - rechts wordt berekend en de flaggen worden gezet.

$$\begin{array}{lll} \text{links} = \text{rechts} & \text{ZF} = 1 & \text{CF} = 0 \\ \text{links} > \text{rechts} & \text{ZF} = 0 & \text{CF} = 0 \\ \text{links} < \text{rechts} & \text{ZF} = 0 & \text{CF} = 1 \end{array}$$

Omdat er twee manieren zijn om getallen (integers) te gebruiken ligt het bij het gebruik van signed integers (die positief en negatief kunnen zijn) anders. Hierbij wordt gebruik gemaakt van drie flags:

- Zero flag (ZF)
- Overflow flag (OF)
- Sign flag (SF)

De overflow flag wordt gezet als het resultaat *overflowt* of *underflowt*. De *sign flag* wordt gezet als het resultaat van de operatie negatief is.

$$\begin{array}{lll} \text{links} = \text{rechts} & \text{ZF} = 1 & \text{n.v.t.} \\ \text{links} > \text{rechts} & \text{ZF} = 0 & \text{SF} = \text{OF} \\ \text{links} < \text{rechts} & \text{ZF} = 0 & \text{SF} \neq \text{OF} \end{array}$$

[2] Het goto statement kan sourcecode onleesbaar maken en resulteren in grote onduidelijkheden. Toch is het onwaar om te zeggen dat het gebruik van goto geen gestructureerde code kan opleveren. Sterker nog, een C compiler zet statements zoals if en else ook om in assembly code!

[3] Het gebruik van twee getallen is onmogelijk en tegelijkertijd zinloos omdat het niet aangeeft waar het resultaat geplaats wordt.

De '*cmp*'-instructie brengt wijzigingen aan op het *EFLAGS* register. Maar er zijn meerdere instructies die het EFLAGS register aanpassen, enige aandacht hierbij is dus gepast.

7.3 Jumps & flags

Er zijn twee verschillende soorten jumps te onderscheiden.

- *Onvoorwaardelijke jump* (zonder condities)
- *Voorwaardelijke jumps* (met condities)

De onvoorwaardelijke jump gebeurt met de '*jmp*'-instructie. Er vindt geen test of andere manier om voorwaarden te testen plaats.

De voorwaardelijke jumps jumpen pas na een bepaalde conditie.
Hieronder een lijst van voorwaardelijke (*conditionele*) jumps.

Voorwaardelijke jump	jumps als ...
JZ	ZF=1
JN	ZF=0
JS	SF=1
JNS	SF=0
JO	OF=1
JNO	OF=0
JP	PF=1
JNP	PF=0
JC	CF=1
JNC	CF=0

Het is belangrijk om te begrijpen dat de code na de jump niet wordt uitgevoerd, tenzij hier naartoe wordt gejumpt.

Het volgende voorbeeld demonstreert het gebruik van een *onvoorwaardelijke* of *onconditionele* jump. De tweede string wordt door het gebruik van de jump niet op het scherm gezet.

```
section .data
string1 db "Dit␣wordt␣afgedrukt", 0x0a
len1    equ $ - string1
```

```
string2 db  "Dit␣wordt␣NIET␣afgedrukt", 0x0a
len2    equ $  -   string2

section .text
global _start
    _start:

    mov eax, 4        ; system call nr. van write
    mov ebx, 1        ; 1e arg: file descriptor (stdout)
    mov ecx, string1  ; 2e arg: pointer naar len1
    mov edx, len1     ; 3e arg: lengte van string1
    int 0x80

; de assembler vervangt end door een geheugenadres
    jmp end

; deze string wordt nooit afgedrukt
    mov eax, 4
    mov ebx, 1
    mov ecx, string2
    mov edx, len2
    int 0x80

end:
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

7.3.1 Het if-statement

Met het gebruik van voorwaardelijke jumps kunnen we C statements zoals if en else of for nabootsen.

De sourcecode '*jnz.asm*' heeft de volgende werking: het vergelijkt het EAX-register met 0, bevat EAX een 0 dan voert het '*if* uit anders '*else*'.

```
section .data
if  db  "IF", 0x0a
len equ $  -   if
```

```
section .text
global _start
_start:

    xor eax, eax

    cmp eax, 0
    jnz else

; het if deel
    mov eax, 4      ; system call nr. van write
    mov ebx, 1      ; 1e argument: file descriptor (stdout)
    mov ecx, if     ; 2e argument: pointer naar len
    mov edx, len    ; 3e argument: lengte van if
    int 0x80

    mov eax, 3
    jmp end         ; jmp naar end, else niet uitvoeren

; het else deel
else:
    mov eax, 2

end:
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

In assembly gebeurt er dit: de inhoud van EAX wordt vergeleken met 0, zijn ze gelijk dan wordt ZF=1 en CF=0. Dus de jump, 'jnz else', vindt niet plaats, omdat jnz alleen jumpt als ZF=0 is.

Met 'nasm -f elf jnz.asm' wordt het gecompileerd en met 'ld jnz.o -o jzn' wordt het gelinkt. Als resultaat wordt er 'IF' op het scherm gezet.

Het assemblyvoorbeeld kan worden gezien als de volgende *pseudo C-code*.

```
if(eax==0) {
        write(3, "IF\n");
```

```
        eax=3;
}
else {
        eax=2;
}
```

- Verander op regel 8 'xor eax, eax' in 'mov eax, 2' en compileer en link het opnieuw. Verklaar de uitkomst, maak hierbij gebruik van ALD.

7.4 Jumps & loops

Eerder dit hoofdstuk was er gesteld dat er verschillende jumps waren. De reden hiervan is dat het mogelijk is om getallen als signed en unsigned integers te gebruiken. Beiden hebben een andere representatie en hebben voor jumps daarom andere *opcodes*.

De in de vorige paragraaf gegeven lijst van jumps is niet makkelijk bruikbaar en bij geavanceerde structuren is het niet toereikend. Daarom biedt de 80x86 nog een set van *jump instructies* voor *signed* en *unsigned integers*.

Signed integers	jumps als ...	Unsigned integers	jumps als ...
JE, JZ	links $=$ rechts	JE, JZ	links $=$ rechts
JNE, JNZ	links $\neq$ rechts	JNE, JNZ	links $\neq$ rechts
JL, JNGE	links $<$ rechts	JB, JNAE	links $<$ recht
JLE, JNG	links $\leq$ rechts	JBE, JNA	links $\leq$ rechts
JG, JNLE	links $>$ rechts	JA, JNBE	links $>$ rechts
JGE, JNL	links $\geq$ rechts	JAE, JNA	links $\geq$ rechts

De 'jump equal' (*JE*) en 'jump not equal' (*JNE*) zijn voor signed en unsigned integers hetzelfde. Sterker nog, ze zijn alleen maar een andere naam voor *JZ* en *JNZ*. Elke *jump* heeft meerdere *mnemonics* voor dezelfde instructie. Dit kan vervelend lijken, maar is in de praktijk gemakkelijk. Zo is 'jump less equal' (*JLE*) hetzelfde als 'jump not greater' (*JNG*).

7.4.1 Het while-statement

De volgende sourcecode is een while-statement in assembly. Er wordt gebruik gemaakt van een *JA* [4] jump naar end.

[4] JA wordt gebruikt bij unsigned integers, zie de tabel van jumps voor signed en unsigned integers.

Sla de source op in het bestand '*while.asm*' en compileer en link het. Nu het een executable is kan het geladen worden in ALD om de werking te laten zien.

```
section .bss
buffer  resd 1

section .text
global _start
_start:
    xor eax, eax        ; eax=0
    inc eax             ; eax=1

while:
    cmp eax, 10
    ja end

    mov [buffer], eax   ; eax opslaan in buffer

    ; De code die hier zou staan wordt 10x herhaald.
    ; eax wordt opgeslagen in buffer zodat er bijvoorbeeld
    ; een system call kan worden gemaakt met gebruik van
    ; eax. Het opslaan van eax en het het aanmaken en
    ; gebruiken van buffer is niet noodzakelijk.

    mov eax, [buffer]   ; eax terughalen

    inc eax
    jmp while

end:
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

Het lijkt misschien vreemd dat er een JA jump wordt gebruik die jumpt als links $>$ rechts terwijl als het vertaald wordt naar pseudo C-code men een $<$ zou gebruiken. Dit komt omdat er normaal de 'while' in wordt gegaan nadat de voorwaarde geldig is. In dit voorbeeld wordt de 'while' juist verlaten.
Wees ervan bewust dat er duizenden manieren zijn om lussen of loops te maken!

Hieronder staat de pseudo C-code van 'while.asm', die dezelfde uitwerking heeft maar niet op dezelfde manier werkt.

```
unsigned int eax=1;
while(eax<10) {
        /* de code die 10x herhaald wordt */
        eax++;
}
```

7.4.2 Het for-statement

De Intel processor heeft instructies geïmplementeerd voor het maken van loops. Deze loop-instructies kunnen onder andere worden gebruikt bij het maken van for-statements. De loop-instructies hebben standaard één operand, dit is het label dat herhaald dient te worden (waarheen gejumpt wordt). Het EFLAGS register wordt bij de *loop-instructies* niet gewijzigd.

- **loop** Verlaagt ECX met 1, als ECX $\neq$ 0 wordt er naar het label gejumpt.
- **loope, loopz** Verlaagt ECX met 1, als ECX $\neq$ 0 en ZF $=$ 1 wordt er naar het label gejumpt.
- **loopne, loopnz** Verlaagt ECX met 1, als ECX $\neq$ 0 en ZF $=$ 0 wordt er naar het label gejumpt.

Voor meer informatie over de verschillen tussen de *loop*-instructies zie Section A.99 van de NASM handleiding.

Het onderstaande assembly 'snippet' demonstreert een for-loop.

```
        mov ecx, 10
enter_loop:

        ; de code die 10x wordt herhaald

loop enter_loop
```

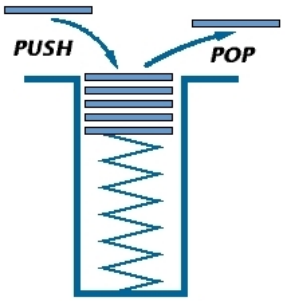
In pseudo C-code zou het er zo uit kunnen zien.

```
for(ecx=10; ecx>0; ecx--) {
        /* code die 10x wordt herhaald */
}
```

7.5 Stack

De *stack* is een abstract datatype dat veel wordt gebruikt in de informatica. De stack heeft de eigenschap dat het element wat er als laatste wordt opgezet er als eerste weer wordt afgehaald. Dit wordt ook wel *LIFO*, Last In First Out genoemd.

Om met elementen op de stack om te gaan zijn er twee instructie; *PUSH* en *POP*. PUSH plaats een element op de stack en POP haalt er een element af.



Met de *program stack*, als er over assembly wordt gesproken, bedoelen we een aaneenschakeling van geheugenadressen die data bevatten.
Op deze stack worden vooral variabelen, return adressen van functies en parameters van functies bewaard. Dit zijn vooral dingen waar hogere programmeertalen van afhankelijk zijn, toch wordt de stack direct gebruikt bij het programmeren in assembly. Meestal gebeurt dit als er tijdelijk data opgeslagen moet worden en er geen registers meer voor handen zijn omdat deze gebruikt (moeten) worden.

7.5.1 POP en PUSH

POP en PUSH zijn de meest voorkomende instructies met betrekking tot de stack. Beiden hebben als operand een register dat 16-bit (word) of 32-bit (dword) moet zijn. Dit betekent dat bytes pushen op de stack niet mogelijk is, ook wordt het afgeraden om words te pushen om de verwarring te voorkomen.

Het onderstaande voorbeeld *pusht/popt* de inhoud van het EAX-register op de

stack. Daarna plaats het de inhoud van EBX in EAX en popt het de 'oude' waarde van EAX in EBX die van de stack afkomt.

De werking van het programma is hetzelfde als de xchg-instructie, namelijk het omwisselen van de inhoud van twee registers. Om de werking van de sourcecode beter te begrijpen kan het in ALD geladen worden.

```
section .text
global _start
_start:
; wisselen met gebruik vd stack
    mov eax, 0xffeedd
    mov ebx, 0xaabbcc
    push eax            ; push inhoud eax op de stack
    mov eax, ebx        ; inhoud ebx naar eax
    pop ebx             ; oude inhoud eax in ebx via stack

; wisselen met xchg
    mov ecx, 0x1
    mov edx, 0x2
    xchg ecx, edx

end:
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

De werking van PUSH en POP zijn afhankelijk van het *ESP-register* [5] (de *stack-pointer*). *ESP* bevat een offset in het *stacksegment* (*SS*). [6]
PUSH plaats een dword in het geheugen door 4 van de inhoud van ESP af te trekken en het op dat geheugenadres te plaatsen. Er wordt 4 van de pointer afgetrokken omdat ESP een geheugenadres bevat in bytes en er een dword (4 Bytes) wordt gepushed. [7]

[5] Het ESP-register en het EBP-register worden ook gebruikt bij het maken van stackframes, details hierover staan niet in het dictaat. In hoofdstuk 6 Taalniveaus werden hiervoor de *enter* en de *leave* instructies gebruikt.

[6] Vaak is het *data segment* hetzelfde segment als het *stack segment*. Dit is te zien door de inhoud van *DS* en *SS* te bekijken in ALD .

[7] Eerder is gesteld dat men voorzichtig moet omgaan met het handmatig veranderen van het ESP-register. De reden daarvan is dat na het handmatig veranderen van de stack deze niet meer functioneert zoals men zou verwachten zonder aanpassingen.

Het volgende voorbeeld '*pushpop.asm*' demonstreert de werking van het ESP-register. Om het beter te begrijpen kan men het voorbeeld inladen in ALD. Wees ervan bewust dat de inhoud van het ESP-register van systeem tot systeem kan verschillen.

```
section .text
global _start
_start:

    mov eax, 0xa    ; eax=0xa                    esp=0xBFFFF910
    push eax        ; plaats 0xa op 0xBFFFF90C   esp=0xBFFFF90C
    push dword 0xb  ; plaats 0xb op 0xBFFFF908   esp=0xBFFFF908
    push dword 0xc  ; plaats 0xc op 0xBFFFF904   esp=0xBFFFF904

    pop eax      ; eax=0xc                       esp=0xBFFFF908
    pop ebx      ; ebx=0xb                       esp=0xBFFFF90C
    pop ecx      ; ecx=0xa                       esp=0xBFFFF910

end:
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

Als 'pushpop' in ALD geladen is, kan men de stack na de instructie 'push 0xb' bekijken door de inhoud van het geheugen te bekijken (waar push de waarden neerzet), oftewel de inhoud van het ESP-register. Let wel op dat alles in het geheugen via de little-endian manier is opgeslagen.

Hier een kort snapshot van de uitkomst in ALD.

```
...
...
08048086 680B000000 push 0xb
ald> next
eax = 0x0000000A ebx = 0x00000000 ecx = 0x00000000 edx = 0x00000000
esp = 0xBFFFF908 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds = 0x0000002B es = 0x0000002B fs = 0x00000000 gs = 0x00000000
ss = 0x0000002B cs = 0x00000023 eip = 0x0804808B eflags = 0x00200346

0804808B 680C000000 push 0xc
```

ald> examine 0xBFFFF908
Dumping 20 bytes of memory starting at 0xBFFFF908 in hex
BFFFF908: **0B 00 00 00 0A 00 00 00** 01 00 00 00 1A FA FF BF .....
BFFFF918: 00 00 00 00 ....
ald>
...
...

7.5.2 PUSHA/POPA

Het komt vaak voor dat men tijdelijk de data in de registers wil opslaan om bijvoorbeeld een routine in te gaan waar de registers veranderen. Het is dan gemakkelijk als de registers weer in de oude staat terug gezet kunnen worden.

Met de instructies *pusha* en *popa* pusht men de inhoud van de registers op de stack. Er zijn twee typen te onderscheiden; word registers en double word registers.

- **Pushaw** en **popaw** worden gebruikt bij word (16-bit) registers. Ze pushen/poppen AX, CX, DX, BX, SP, BP, SI en DI op/van de stack (niet in deze volgorde).
- **Pushad** en **popad** worden gebruikt bij double word (32-bit) registers. Deze pushen/poppen EAX, ECX, EDX, EBX, ESP, EBP, ESI en EDI op/van de stack (niet in deze volgorde).

Bijvoorbeeld:

```
...
pusha

; code die general-purpose registers wijzigt.

popa
...
```

7.5.3 PUSHF en POPF

PUSHF en *POPF* werken hetzelfde als PUSHA en POPA alleen pushen of poppen ze niet alle general-purpose registers maar het EFLAGS register. Op deze manier

kan men altijd de inhoud van het EFLAGS register behouden en het later weer gebruiken. Het is ook mogelijk om het EFLAGS register te poppen, te wijzigen en daarna weer te pushen.

7.6 Vragen en opdrachten

1. (a) Wat is het verschil tussen conditionele en onconditionele jumps?
   (b) Wordt een onvoorwaardelijke jump altijd uitgevoerd?
2. Noem twee een toepassingen waarbij jumps gebruikt kunnen worden.
3. De code na een jump wordt die nog uitgevoerd? Leg uit wanneer wel en wanneer niet.
4. (a) Welke registers zijn betrokken bij het maken van een jump?
   (b) Wat is de invloed van een jump op het instructieregister (EIP)?
5. (a) Schrijf een for-loop achtige constructie die 100 keer een bepaalde string op het scherm schrijft. Probeer hierbij zo min mogelijk instructies te gebruiken.
   (b) Test of het loop programma van de vorige opdracht inderdaad het EFLAGS register ongewijzigd laat. Doe dit door het programma in ALD te laden.
6. Verander in de sourcecode van 'jnz.asm' de jump naar een andere jump, maar doe het zo dat de werking exact hetzelfde blijft.
7. Schrijf een assembly programma met een lus die van 1 tot 10 telt. Deze getallen worden opgeteld in een general-purpose register naar keuze. Toon de werking ervan aan in ALD.
8. Leg de wijze waarop het datatype stack werkt uit.
9. Waar dient de stack als deel van het geheugen voor?

Bijlage A

NASM installeren

NASM (Netwide Assembler) is een opensource assembler voor Intel 80x86 processoren. Omdat het opensource is kunnen we een tarball downloaden met daarin de source, die we vervolgens gaan compileren en installeren.

A.1 Downloaden van de source

Download de sourcecode van laatste versie van NASM via http van:

http://www.kernel.org/pub/software/devel/nasm/source/

De tarball met de de sourcecode heeft een naam die als volgt is opgebouwd nasm-X.XX.tar.bz2, X.XX staat voor de versie.
Sla de tarball op in de directory ”/tmp”.
Nu het bestand ”/tmp/nasm-X.XX.tar.bz2” bestaat, kan het uit worden gepakt door ”tar xvyf nasm-X.XX.tar.bz2” in te tikken.
Door de tarball uit te pakken verschijnt er een directory ”nasm-X.XX/”.

A.2 ./configure && make && make install

Verander de de huidige directory in ”/tmp/nasm-X.XX/”.
Om de sourcecode juist in te stellen voor het gebruikte systeem moet er een configuratie worden ingesteld. Doe dit door het volgende script te runnen.

./configure

Aan de hand van de instelling gevonden door ”./configure” wordt de Makefile ingesteld. De Makefile dient als hulpmiddel bij het compileren. Compileer de sourcecode door:

make

Als er geen fouten zijn gevonden, is de sourcecode succesvol gecompileerd. Het kan nu worden geïnstalleerd met (dit commando vereist root rechten):

su
make install

Nu de source gecompileerd en geïnstalleerd is kan het programma NASM worden gestart door:

nasm -h

Er verschijnt nu een korte uitleg van de opties die mee kunnen worden gegeven aan NASM. Voor een meer gedetailleerde uitleg over NASM kan er worden gekeken naar de man page van NASM.

A.3 De NASM documentatie

NASM beschikt over een zeer uitgebreide documentatie die handig is bij het programmeren. Zo geeft het uitleg over alle mnemonic's die gebruikt kunnen worden en macro's ed. Ga naar de directory "/tmp/nasm-X.XX/doc/" en tik daar:

make install

De Makefile is zo ingesteld dat het de documentatie in verschillende formaten genereert. Dit is handig, er is nu documentatie in html, plain-text of in Postscript. De documentatie in html vorm staat in "/tmp/nasm-X.XX/doc/html/".

Bijlage B

ALD installeren

Compile the program and run it. Make sure you include the symbol table for the debugger or not... depending upon how macho you feel today.

Mudge, l0pht advisories

ALD (Assembly Language Debugger) is een debugging tool voor debuggen van x86 executables op assembly niveau.

B.1 Downloaden van de source

Download de sourcecode van laatste versie van ALD via http van:

http://ellipse.mcs.drexel.edu/source/

Sla de gedownloade tarball op in de directory ”/tmp/”.
Ga naar de ”/tmp” directory door ”cd /tmp” in te geven. Het bestand ”ald-X.X.XX.tar.gz” in de /tmp directory kan het uit worden gepakt door ”tar xvzf ald-X.XX.tar.gz”
Er verschijnt nu een directory met de naam ”ald-X.X.XX/”.

B.2 Configureren, compileren en installeren

Verander de de huidige directory in ”/tmp/ald-X.X.XX/” door ”cd ald-X.X.XX/”
Configureer de software met:

./configure

De configuratie is opgeslagen in de Makefile, er kan nu gecompileerd worden met het commando:

make

Als de sourcecode succesvol is gecompileerd kan het nu worden geïnstalleerd. Het installeren (het kopiëren van de binairies naar ”/usr/local/bin/”) vereist root-rechten op het systeem. Daarom moet er van gebruiker worden gewisseld (aannemende dat er niet als root werd gewerkt).

su
make install

ALD is nu geïnstalleerd en kan worden gestart door ”ald”.

Bijlage C

Linux Kernel Modules in assembly

Het schrijven van kernel modules in assembly is wel degelijk mogelijk,
ondanks dat het een tijdrovend karwei kan zijn.
Zie voor meer informatie over het schrijven van kernel modules De Vrijer 2001 - C Coding in the Linux Kernel Environment.
Het onderstaande voorbeeld demonstreert het gebruik van kernel modules in assembly.

Het kan gecompileerd worden door de volgende opdrachten. Verander eerst het versienummer op regel 5 naar de huidige kernel versie. [1]

nasm -f elf module.asm -o module.tmp
ld -r -o module.o module.tmp

Het kan worden ingeladen en uit het geheugen worden geladen met de onderstaande opdrachten. Hiervoor zijn root rechten op het systeem nodig.

insmod module.o
rmmod module

```
section .data
init db "Real␣programmers␣code␣in␣assembly", 0xa, 0
clean db "Said␣the␣kernel", 0xa, 0
kernel_version db "2.4.3", 0

global init_module
global cleanup_module
```

[1] Het commando '*uname -r*' toont de draaiende kernel versie.

```
global kernel_version

extern printk

section .text

init_module:
    push dword init   ; zet 'init' op de stack
    call printk       ; roep printk aan,
                      ; m.o.w. druk 'init' af

    pop eax           ; return waarde printk

    xor eax, eax      ; return moet 0 zijn
                      ; anders kan de module
                      ; niet worden geladen.

    ret               ; einde init_module()

cleanup_module:
    push dword clean  ; zet 'clean' op de stack
    call printk       ; roep printk aan
    pop eax           ; return waarde printk
    ret               ; einde cleanup_module()
                      ; en einde lkm
```

Bijlage D

Ascii Tabel

Decimaal	*Octaal*	*Hexadecimaal*	*Binair*	*Waarde*
000	000	000	00000000	NUL
001	001	001	00000001	SOH
002	002	002	00000010	STX
003	003	003	00000011	ETX
004	004	004	00000100	EOT
005	005	005	00000101	ENQ
006	006	006	00000110	ACK
007	007	007	00000111	BEL
008	010	008	00001000	BS
009	011	009	00001001	HT
010	012	00A	00001010	LF
011	013	00B	00001011	VT
012	014	00C	00001100	FF
013	015	00D	00001101	CR
014	016	00E	00001110	SO
015	017	00F	00001111	SI
016	020	010	00010000	DLE
017	021	011	00010001	DC1
018	022	012	00010010	DC2
019	023	013	00010011	DC3
020	024	014	00010100	DC4
021	025	015	00010101	NAK
022	026	016	00010110	SYN
023	027	017	00010111	ETB
024	030	018	00011000	CAN
025	031	019	00011001	EM

Decimaal	*Octaal*	*Hexadecimaal*	*Binair*	*Waarde*
026	032	01A	00011010	SUB
027	033	01B	00011011	ESC
028	034	01C	00011100	FS
029	035	01D	00011101	GS
030	036	01E	00011110	RS
031	037	01F	00011111	US
032	040	020	00100000	SP
033	041	021	00100001	!
034	042	022	00100010	"
035	043	023	00100011	#
036	044	024	00100100	$
037	045	025	00100101	%
038	046	026	00100110	&
039	047	027	00100111	’
040	050	028	00101000	(
041	051	029	00101001	)
042	052	02A	00101010	*
043	053	02B	00101011	+
044	054	02C	00101100	,
045	055	02D	00101101	-
046	056	02E	00101110	.
047	057	02F	00101111	/
048	060	030	00110000	0
049	061	031	00110001	1
050	062	032	00110010	2
051	063	033	00110011	3
052	064	034	00110100	4
053	065	035	00110101	5
054	066	036	00110110	6
055	067	037	00110111	7
056	070	038	00111000	8
057	071	039	00111001	9
058	072	03A	00111010	:
059	073	03B	00111011	;
060	074	03C	00111100	$<$
061	075	03D	00111101	=
062	076	03E	00111110	$>$
063	077	03F	00111111	?
064	100	040	01000000	@
065	101	041	01000001	A

Decimaal	*Octaal*	*Hexadecimaal*	*Binair*	*Waarde*
066	102	042	01000010	B
067	103	043	01000011	C
068	104	044	01000100	D
069	105	045	01000101	E
070	106	046	01000110	F
071	107	047	01000111	G
072	110	048	01001000	H
073	111	049	01001001	I
074	112	04A	01001010	J
075	113	04B	01001011	K
076	114	04C	01001100	L
077	115	04D	01001101	M
078	116	04E	01001110	N
079	117	04F	01001111	O
080	120	050	01010000	P
081	121	051	01010001	Q
082	122	052	01010010	R
083	123	053	01010011	S
084	124	054	01010100	T
085	125	055	01010101	U
086	126	056	01010110	V
087	127	057	01010111	W
088	130	058	01011000	X
089	131	059	01011001	Y
090	132	05A	01011010	Z
091	133	05B	01011011	[
092	134	05C	01011100	\
093	135	05D	01011101	]
094	136	05E	01011110	^
095	137	05F	01011111	_
096	140	060	01100000	‘
097	141	061	01100001	a
098	142	062	01100010	b
099	143	063	01100011	c
100	144	064	01100100	d
101	145	065	01100101	e
102	146	066	01100110	f
103	147	067	01100111	g
104	150	068	01101000	h
105	151	069	01101001	i

Decimaal	*Octaal*	*Hexadecimaal*	*Binair*	*Waarde*
106	152	06A	01101010	j
107	153	06B	01101011	k
108	154	06C	01101100	l
109	155	06D	01101101	m
110	156	06E	01101110	n
111	157	06F	01101111	o
112	160	070	01110000	p
113	161	071	01110001	q
114	162	072	01110010	r
115	163	073	01110011	s
116	164	074	01110100	t
117	165	075	01110101	u
118	166	076	01110110	v
119	167	077	01110111	w
120	170	078	01111000	x
121	171	079	01111001	y
122	172	07A	01111010	z
123	173	07B	01111011	{
124	174	07C	01111100	|
125	175	07D	01111101	}
126	176	07E	01111110	~
127	177	07F	01111111	DEL

Bibliografie

[1] ing. R. de Vrijer.
C coding in the linux kernel environment.
Initworks B.V.

[2] ir. F.J. Dijkstra.
Computers: organisatie - architectuur - communicatie.
EPN, 1998.

[3] Assembly Programming Journal.
http://asmjournal.freeservers.com.

[4] Linux Newsgroups.
alt.os.linux, comp.os.linux.

[5] Linux Documentation Project.
http://www.linuxdoc.org.

[6] Microprocessor Resources.
http://x86.org/.

[7] Roelf Sluman.
Toolbox voor machinetaalprogrammeurs.
Kluwer PC Boeken, 1989.

[8] Richard M. Stallman and Roland H. Pesch.
Debugging with gdb - the gnu source-level debugger.

[9] W. Richard Stevens.
Advanced programming in the unix environment.
Addison Wesley, 1993.

[10] Tom Swam.
Werken met turbo assembler.
Acedemic Service, 1991.

[11] Andrew S. Tanenbaum and Albert S. Woodhull.
Operaring systems: design and implementation - second edition.
Prentice-Hall, 1997.

[12] Tools Interface Standards (TIS).
Executable and linkable format.
Portable Format Specification.

[13] Little Endian vs. Big Endian.
http://www.noveltheory.com/techpapers/endian.asp.

[14] The x86 Assembly Language FAQ.
http://www2.dgsys.com/ raymoon/x86faqs.html.

[15] The x86 Assembly Language Newsgroup.
comp.lang.asm.x86.

[16] Technical x86 processor information.
http://www.sandpile.org/.

linux.org

Index