Linux Assembly

Uitwerkingen van de vragen en opdrachten

The choice of a GNU generation

Hoofdstuk 3

1. (a) Een system call is een functie geleverd door de kernel (het operating system, een interface tussen applicaties en de hardware). Via system calls kan software met de kernel communiceren.

   (b) Bij het programmeren met system calls maakt men direct gebruik van services van de kernel (het operating system).
   Als men programmeert met libraries maakt men gebruik van de functionaliteit van de libraries en niet de van kernel zelf. Hier ligt het verschil.

   (c) Ze zijn operating system afhankelijk.

   (d) System calls zijn sneller en het demonstreert de manier waarop in de kernel system calls zijn geïmplementeerd.

2. (a) Als er de mogelijkheid zou zijn om ze te overschrijven kan het programma een andere werking/verloop krijgen dan er oorspronkelijk bedoeld was.

   (b) Het operating system zal dit in veel gevallen niet accepteren.
   Er kunnen zich de volgende situaties voordoen met het programma:

   - Segmentatie fouten
   - Variabelen overschrijven
   - Bufferoverflows
   - Programma verloop verandert
   - enz. . .

3. Ja, dit is mogelijk. C programma's bestaan als executable ook uit instructies net als gecompileerde assembly code.
   Soms bieden C compilers extra mogelijkheden om assembly met C te combineren aan. Dit wordt 'inline assembly' genoemd. Maar ook zonder deze functionaliteit is het mogelijk om beide in één executable te laten samen werken.

4. (a) Omdat assembly code in het algemeen doelmatiger geschreven is. Er is minder zinloze code (bijvoorbeeld vanuit libraries).

(b) Het programmeren in assembly kan tijdrovend zijn, soms wordt code te snel ingewikkeld en bovendien is het machine afhankelijk.

5.

```
1  section .data
2      hello db "Hello", 0xa
3      lenhello equ $ - hello
4
5      world db "world", 0x0a
6      lenworld equ $ - world
7
8  section .text
9
10 global _start
11 _start:
12
13 ; druk de hello1 af
14     mov eax, 4
15     mov ebx, 1
16     mov ecx, hello
17     mov edx, lenhello
18     int 0x80
19
20 ; druk hello2 af
21     mov eax, 4
22     mov ebx, 1
23     mov ecx, world
24     mov edx, lenworld
25     int 0x80
26
27 ; exit
28     mov eax, 1
29     mov ebx, 0
30     int 0x80
```

6. (a) Om de code te structureren en leesbaar te houden. Elk veld heeft een ander doel, zo kan men gemakkelijk instructies van operanden scheiden.

(b)
- **Label veld**
  Het labelen, een entry point maken in het geheugen.
- **Mnemonic veld**
  Hier staan de instructies in menslogische woorden (mnemonic's).
- **Operand veld**
  Geeft aan waarop de instructie iets uitvoert (bijv. geheugen of registers).
- **Commentaar veld**
  Extra toelichting bij de code van de programmeur. Deze informatie wordt niet in de executable gezet.

(c) Mov is een instructie en komt uit het mnemonic veld waar de instructies staan.

7. Nee dat is niet verplicht en ook niet haalbaar. Omdat niet elke regel assembly een label, mnemonic of operand veld nodig heeft. Het commentaar veld is altijd optioneel omdat het niet in de executable wordt gezet.

8. (a) man 2 chmod

   (b) Op regel 22 van /usr/include/asm/unistd.h staat #define __NR_chmod 15. De system call voor chmod is 15.

9.

```
1  section .bss
2          input resb 1
3
4  section .text
5  global _start
6  _start:
7  ; ssize_t read(int fd, void *buf, size_t count);
8          mov eax, 3
9          mov ebx, 0
10         mov ecx, input
11         mov edx, 5
12         int 0x80
13
14 ; ssize_t write(int fd, const void *buf, size_t count);
15         mov eax, 4
16         mov ebx, 1
17         mov ecx, input
18         mov edx, 5
```

```
19          int 0x80
20
21  ; void _exit(int status);
22          mov eax, 1
23          mov ebx, 0
24          int 0x80
```

10. Verander de regel 6 in:

```
1       pathname db "/etc/inittab", 0
```

Hoofdstuk 4

1. (a) Het uitvoeren van instructies en hardware aansturen.
   (b) Bewerkingen van software worden uitgevoerd door de processor. De processor bepaald de werking/loop van de software en stuurt hardware aan.

2. (a) Een set (lijst/verzameling) van processor instructies die vastliggen in de processor.
   (b) Nee, een instructieset ligt vast in de processor.

3. (a) Endian slaat op de volgorde waarop bytes worden opgeslagen in het geheugen.
   (b) Nee, big-endian en little-endian processoren hebben vaak complexe verschillen, snelheid is daarom moeilijk te testen.
   Toch heeft het gebruik van little-endian voordeel door waarden te verkorten.
   (c) Nee, klonen zijn Intel compatible dus verschillen in byte-ordering of in de instructieset zijn onmogelijk.

4. Nee, het EFLAGS register wordt op een andere manier gebruikt. Men kijkt naar bepaalde veranderingen van bits op het register (bijv. de carry-flag voor carries).

5. - Verdelen over meerdere registers
   - In het geheugen zetten
   - Op de stack plaatsen
   - In bestanden opslaan
   - enz. . .

6. Nee, een Macintosh computer heeft een andere processor. Met een niet Intel compatible instructieset. Ook verschilt de byte-ordering.

7. Omdat nieuwe processoren backwards compatible moeten zijn om oudere software te draaien.

8. Het getal 0x41424344 wordt in het ECX-register geladen.
   0x41 = A in ascii
   0x42 = B
   0x43 = C
   0x44 = D
   Vervolgens wordt de inhoud van ECX in [var] gezet door 'mov [var], ecx'. Het adres van var wordt dan in het ECX-register geplaats met 'mov ecx, var'.

   Om iets af te drukken op het scherm kan men gebruik maken van system calls in assembly. Door 'mov edx, 4' bevat het EDX-register het getal vier, wat aangeeft dat er een string wordt afgedrukt van 4 letters.
   Het adres van (of pointer naar) de string die in [var] staat werd door 'mov ecx, var' al in het ECX-register geladen.
   Nu moet nog worden aangegeven waarheen moet worden geschreven. Dit gebeurt door 1 (stdout) in het EBX-register te laden (mov ebx, 1).
   Door 'mov eax, 4' te doen wordt het EAX-register geladen met de system call voor write.
   Nu de registers de juiste waarden bevatten kan de kernel worden aangeroepen en kan de system call worden gemaakt. Dit gebeurt door 'int 0x80'.

   Als het assembly programma gecompileerd en gedraaid wordt verschijnt er 'DCBA' op het scherm, dit omdat de Intel een little-endian systeem is. Het slaat waarden van rechts naar links op.

Hoofdstuk 5

1. (a) Ja, sommige registers mogen direct worden gewijzigd (niet alle!). Dit worden general-purpose registers genoemd. Omdat software vertaald wordt naar instructies die registers gebruiken als operanden.

   (b) De E van extented staat voor een uitbreiding van 16-bit registers naar 32-bit registers. De E in EAX betekent dat EAX (32-bit) een uitbreiding is op AX (16-bit).

   (c) Byte $=$ 8 bit

   (d) Word $=$ 16 bit

   (e) Double word (dword) $=$ 32 bit

2. (a) Het bevat een segment selector naar het code segment. CS bevat dus het adres waar het code segment in het geheugen begint.

   (b) Het EIP-register (instructie pointer) bevat een offset in het huidige code segment naar de eerst volgende instructie die wordt uitgevoerd.

   (c) Het EIP-register kan niet op deze wijze gewijzigd worden. Het is namelijk geen general-purpose register, het kan beïnvloed worden door jumps, calls en returns.

   (d) Het CS-register bevat een selector naar het huidige code segment en EIP bevat een offset binnen het huidige code segment. Door beiden te combineren (CS:EIP) wijzen ze naar de eerst volgende instructie.

3. (a) Omdat programmatuur verschillende handelingen uitvoert moet het geheugen bepaalde eigenschappen hebben. Elke section/segment moet andere eigenschappen hebben. Zo moet er data en code geplaats worden, deze zijn nodig voor geldige executables met nuttige uitkomst. Het geheugen wordt via segmenten geädresseerd, er zijn dus meerdere adressen en offsets nodig om te adresseren.

   (b) De segment registers (de registers die naar segmenten wijzen) worden gebruikt om aan te geven waar de segmenten zich bevinden. Het hui-

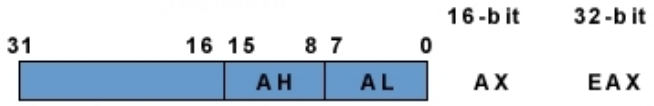
dige code segment kan gevonden worden via CS, het stack segment door SS.

(c) Nee, dit laat een assembler niet toe. De reden is dat het operating system dit niet zou accepteren omdat men dan in geheugen kan adresseren die niet van het proces zelf is. In dit geval handelt de assembler segmenten af voor de programmeur.

4. De intel 386 en compatible processoren (IA32) zijn 32-bit. Ze kunnen 2^{32} bytes of 4GB aan geheugen adresseren. Maar door segmentatie kan met het bereik uitbreiden.

5. Niet altijd wordt het volledige bereik van registers gebruikt. Door het op te delen kan men hier efficiënter mee omgaan.

6. Ja, een wijziging aan AX heeft invloed op EAX.



7. (a) Door instructies uit te voeren die de flags op het EFLAGS register aan of uit te zetten. De flags kunnen als voorwaarde dienen bij de instructies 'cmp' en 'test'. Door gebruik te maken van jumps die aan de hand van de flags op het EFLAGS register naar bepaalde delen in de code springen (jumpen) krijgt een programma een bepaalde volgorde van uitvoering.

   (b) Mocht er een carry gegenereerd worden door een rekenkundige instructie dan wordt deze flag gezet.

   (c) De 'clc'-instructie betekent CLear Carry flag, het zet de carry flag op 0.

```
1 section .text
2
3 global _start
4
5 _start:
6         mov eax, 0xffffff
7         mov ecx, 0xaaaaaa
8         mul dword ecx
```

```
9
10          nop     ; Deze instructie heeft
11          ; geen uitwerking, het
12                  ; wordt alleen gebruikt
13                  ; als demonstratie.
14          clc
15
16          ;exit
17          mov     ebx,    0
18          mov     eax,    1
19          int     0x80
```

Voor de instructie 'mul dword ecx' heeft het EFLAGS de waarde 0x00200346. Na 'mul dword ecx' heeft het de waarde 0x00200B47, dit betekent dat onder andere het eerste bit (of bit 0) van het EFLAGS verandert is. Dat klopt omdat er een carry gegenereerd is bij de vermenigvuldiging. Bit 0 of het eerste bit van van het EFLAGS is namelijk de carry flag.
Na de instructie 'clc' bevat het EFLAGS de waarde 0x00200B46 ipv 0x00200B47. De carry flag is nu niet meer gezet.

(d) Omdat het volledig binair uitschrijven niet altijd zinnig is. Bij het weergeven als hexadecimaal getal kan met ook zien of een instructie wijzigingen aanbrengt op het EFLAGS reigster. Alleen welke flag(s) gewijzigt worden is niet meteen duidelijk.

8. De 386 is compatible met de 8086. Zo ondersteund de 386 real mode en heeft het ook 16-bit registers heeft net als de 8086.

9. Door waarden in registers te wijzigen en deze weer aan voorwaarden te testen kan software bepaalde handelingen uitvoeren.

10. (a) General-purpose registers worden gebruikt om data of pointers naar data (geheugen adressen) in te zetten. Ze worden vervolgens gebruikt als operanden van instructies.

    (b) Ja, de functies van de registers liggen niet vast. Toch maken sommige instructies gebruik van bepaalde registers. Bijvoorbeeld de 'mul'-instructie die gebruik maakt van AL/AX/EAX en DX/EDX aan de hand van de grootte van de operands.

11. (a) AX $=$ 16-bit

    (b) AL $=$ 8-bit, EAX $=$ 32-bit

(c) Ja, ze zijn volledig compatible.

12. NOP staat voor No OPeration. Deze instructie heeft geen uitwerking op registers of geheugen. Er zijn beplaalde gevallen waarin NOP's goed gebruikt kunnen worden. Bijvoorbeeld: als men een jumps per 4 bytes berekent kan het gemakkelijk zijn om de ontbreekende code met NOP's te vullen. Dit wordt veelal gedaan door compilers.

13. (a) Regel 7, 'mov eax, bx' is ongeldig. Omdat het verplaatsen van data vanuit een 16-bit register naar een 32-bit register niet mogelijk is (de processor laat dit niet toe).

    (b) -

    (c) -

    (d) De 'movsx'-instructie past de operanden aan elkaar aan. Op deze wijze kan men data tussen operanden van ongelijke breedte uitwisselen.

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

    - 1 OR $0 = 0$
      OR kan gebruikt worden bij het inverteren van bits.
    - 1 AND $1 = 1$
      AND kan gebruikt worden bij het testen van bepaalde bits.
    - 1 XOR $1 = 0$
      XOR kan gebruikt worden bij het zetten van bepaalde bits zonder dat de origele toestand van de bits bekend zijn.
    - NOT $1 = 0$
      NOT wordt gebruikt om bits te inverteren.

    (b) Het laden van het EDX-register met 0. Een XOR masker van twee dezelfde waarden levert altijd 0.

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

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

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

(d) 1100 1101 NOT
0011 0010

16. (a) Om het bereik aan te duiden.
Signed integers gebruiken two's complement.
Unsigned integers gebruiken een normale representatie.

(b) Signed integers maken gebruik van two's complement. Hierbij wordt het eerste bit (de sign bit) gebruikt om aan te geven of het een positief of negatief getal is.

(c) 0011 0001

(d) *mov al, 0xff*
neg al

17. Dit voorbeeld demonstreerd het delen van twee 8 bits getallen.

```
1  section .data
2  getal db 0x6
3
4  section .text
5  global _start
6  _start:
7
8          mov ax, 0xc
9          div byte [getal]      ; AL:AH=ax/getal
10
11         mov eax, 1
12         xor ebx, ebx
13         int 0x80
```

Hoofdstuk 6

1. Ja, het bss segment bevat ongeïnitialiseerde data die bij het draaien van de executable nog niet ingevuld is. Het bss segment bevat vaak 'variabelen' die afhankelijk zijn van de user.

2. Assembly is het programmeren met instructies aan de hand van mnemonic's en operanden gezien als registers, geheugen en immediates. Programmeren in machinetaal is het programmeren met numerieke representaties en geheugenadressen in plaats van menslogische constructies zoals 'xor edx, [mask]'.

3. Beiden programmeren direct de processor. Assembly is gelijk aan machinetaal alleen maakt het geen gebruik van nietszeggende numerieke waarden maar van woorden.

4. Omdat software geschreven in assembly machineafhankelijk is.

5. Omdat HLL standaard oplossingen heeft voor problemen om bijvoorbeeld te rekenen, met programma stuctuur en andere zaken om te gaan. Dit heeft als voordeel dat programmeurs zich niet in de specificaties van de processor hoeven te verdiepen en zich alleen met het programmeren op een hogere laag kunnen bezig houden.

6. Bijvoorbeeld: Het schrijven van routines die machineafhankelijke taken binnen een operating system uitvoeren zouden in assembly geschreven kunnen worden.

   Bijvoorbeeld: Het schrijven van GUI's kan het beste in HLL worden gedaan. Omdat hier directe controle over de processor niet expliciet nodig is.

7. De sourcecodes van programmeertalen (zowel HLL als assembly) worden eerst omgezet naar objectcode en daarna gelinkt tot een executable. De linker heeft de mogelijkheid om meerde objectcode files te combineren tot één executable waardoor functies vanuit het ene deel door de andere aangesproken kunnen worden.

8. De 0 geeft aan waar de string eindigd. Als er op assembly niveau naar wordt gekeken zien we geheugen met daarin een tekenreeks die eindigt met een 0 om het einde van de string aan te geven.

Hoofdstuk 7

1. (a) Conditionele jumps jumpen bij bepaalde voorwaarden. Onconditionele jumps jumpen altijd, er zijn geen voorwaarden.

   (b) Ja, er wordt geen voorwaarde getest.

2. - Het 10x afdrukken van een bepaalde string.
   - Bij de invoer van een Q sluit het programma af.

3. Nee, in principe niet, tenzij er juist wordt heen gejumpt.

4. (a) Bij het maken van jumps als statement hoeven er geen registers gebruikt te worden (bijvoorbeeld: 'jmp sub_code').
   Maar omdat de executie verandert moeten er wel registers gewijzigd worden. Vanuit dit oogpunt gekeken verandert het EIP, de instructie pointer. EIP wijst nu naar de instructie.
   Bij conditionele jumps wordt er ook naar het EFLAGS register gekeken. (Bij far jumps wordt ook het CS-register gewijzigd.)

   (b) EIP bevat het adres van de volgende instructie. Een jump wijzigt het EIP, deze bevat nu een ander adres. Dit heeft tot gevolg dat er naar andere code wordt gewezen, deze wordt vervolgens uitgevoerd.

5. (a) Bij deze code wordt de waarde van het ecx register niet in het geheugen opgeslagen maar in het esi register. Dit is een betere oplossing omdat registers sneller zijn dan RAM.

```
1 label:
2
3     ; de waarde in het ecx register wordt
4     ; wordt opgeslagen in esi, omdat ecx
5     ; bij het maken van de write syscall
6     ; gewijzigd wordt.
7     mov esi, ecx
```

```
8
9       mov eax, 4
10      mov ebx, 1
11      mov ecx, string
12      mov edx, len
13      int 0x80
14
15      ; haal de 'oude' waarde van ecx terug
16      ; uit esi
17      mov ecx, esi
18
19          loop label
```

(b) EFLAGS blijft ongewijzigd door de 'loop'-instructie.

6. Verander de JNZ jump naar een JNE jump.

7.

```
1  section .text
2  global _start
3  _start:
4
5      mov ecx, 10
6      xor eax, eax
7
8  label:
9      inc eax
10
11     loop label
12
13 end:
14     mov eax, 1
15     xor ebx, ebx
16     int 0x80
```

8. Door gebruik te maken van PUSH kan men elementen op de stack plaatsen. Deze kunnen er in LIFO (Last In First Out) volgorde weer uit worden gehaalt met POP.

9. Het wordt gebruikt bij het wegschrijven van data dat niet in de processor registers bewaard wordt zoals variabelen en functie parameters ed.