Notater: INF2270

Assembler

Veronika Heimsbakk
veronahe@student.matnat.uio.no

11. juni 2014

Innhold

Introduksjon

Dette er mine eksamensnotater i kurset INF2270 – Datamaskinarkitektur som jeg tok våren 2014 ved Institutt for informatikk, Universitetet i Oslo. Notatene er basert på forelesningsfoiler, egne notater (hovedsakelig fra INF1400 – Digital design) og pensumbøkene. Disse inneholder sikkert en del feil og mangler, ikke stol på de. Men send meg gjerne en e-post om det er noen innvendinger.

1 Registere

	%AX	
%EAX	%AH	%AL
%EBX	%BH	%BL
%ECX	%CH	%CL
%EDX	%DH	%DL

Spesielle 16-bits registere: %CS, %DS, %ES, %FS, %GS, %SS, og spesielle 32-bits registere: %EFLAGS, %EIP.

Frie registere Konvensjonen er at %EAX, %ECX og %EDX er frie registere.

Bundne registere De andre registerne er bundne registere. Om de endres, må man ta vare på den opprinnelige verdien og sette denne tilbake før retur.

2 Assembler-programmering

En testfunksjon som returnerer verdien 19.

```
1     .globl nineteen
2 nineteen:
3     movl  $19, %eax
4     ret
```

Følgende C program kan teste funksjonen:

```
1 #include <stdio.h>
2
3 extern int nineteen(void);
4
```

```
5 int main (void)
6 {
7   printf("nineteen() = %d\n", nineteen());
8   return 0;
9 }
```

2.1 Instruksjoner

Hvor - tilsvarer `w` (word), `b` (byte), eller `l` (long).

mov- Flytt en verdi.

add- Adder en verdi til en annen.

sub- Subtraher.

inc- Øk en verdi med 1.

dec- Senk en verdi med 1.

ret Returner fra funksjon.

$17 Konstant.

%eax Registeret %EAX.

4(%esp) Parametre på stakken.

neg- Negasjon (med fortegn).

imul- Multiplikasjon (med fortegn).

div- Divisjon.

jmp Hopp.

cmp- Sammenligne.

lea- Fungerer som `mov`, men henter adressen istedet for verdien..

2.2 Variabler

```
1     .text
2 move:
3     movl  $3, %eax
4     movl  4(%esp), %eax
5     movl  %eax, var
6     ret
7
8     .data
9 var:    .long 17  # En long med verdi 17
10 arr:   .fill 8   # 8 byte uten initialverdi
```

Man kan sette av plass til variabler med spesifikasjonen `.long` eller `.fill`. De bør legges i `.data`.

Faste variabler Disse lever så lenge programmet kjører. De kan gis en initialverdi. Det vanlige er å legge slike variabler i `.data`-segmentet.

```
1         .globl   a, b, d, f
2         .text
3 f:      ret
4         .data
5 a:      .long 0
6 b:      .long 0
7 c:      .byte 0
8         .align   2
9 d:      .long 5
```

Tilsvarende i C:

```
1 int a, b;
2 static char c;
3 long d = 5;
4
5 void f (void) {}
```

2.3 Byte, ord og langord

`mov`- finnes for byte, word (2 byte) og long (4 byte):

```
1 movb      $0x12 , %al
2 movw      $0x1234 , %ax
3 movl      $0x12345678 , %eax
```

2.4 Multiplikasjon

I tillegg til `imulw`/`imull`, finnes en versjon som jobber med faste registere: `mulb` og `imulb` jobber med %AX, `mulw` og `imulw` jobber med %DX:%AX, og `mull` og `imull` jobber med %EDX:%EAX.

2.5 Divisjon

Divisjon gir to svar (kvotient og rest). Den er også litt rar når det gjelder registerbruk: `divl` og `idivl` får svar i %EAX og rest i %EDX. `divw` og `idivw` får svar i %AX og rest i %DX, `divb` og `idivb` får svar i %AL og rest i %AH.

Eksempel .

```
1       .globl div10
2 # C-signatur: unsigned int div10(unsigned int v)
3 div10:
4       movl   4(%esp), %eax # %eax = v
5       movl   $0, %edx      # %edx:
6       movl   $10, %ecx     # %ecx = 10
7       divl   %ecx          # (%eax, %edx) =
8                            #     (%edx:%eax/10, %edx:%eax%10)
9       ret
```

2.6 Flagg

De fleste operasjonene har en bieffekt: visse egenskaper ved resultatet blir lagret i *flaggene*.

Z Zero. Settes til 1 når svaret er 0 (og til 0 ellers).

S Sign. Settes lik øverste bit i svaret.

C Carry. Settes lik den menteoverføringen som skjedde øverst i resultatet.

O Overflow. Settes om svaret var for stort.

P Parity. Settes om laveste byte har et partall 1-bit.

2.7 Maskeoperasjoner

Maskeoperasjonene brukes til å sette eller nulle ut bit i henhold til et gitt mønster (*maske*).

Maske-AND (`andb`) Lik operasjonen & i C. Det er forskjell på & (maske-AND eller bit-AND) og && (logisk AND) i C. `1 & 4 == 0` og `1 && 4 == 1`.

Maske-OR (`orb`) Denne operasjonen setter de bit som er markert i masken. Denne operasjonen er tilgjengelig i C som `|`.

Maske-NOT (`notb`) Denne snur *alle* bit-ene. Finnes i C som $\sim$.

Maske-XOR (`xorb`) Denne operasjonen *snur* bare de bit som er markert i masken. Denne operasjonen kalles ofte «logisk addisjon». Den er tilgengelig i C og heter `^`.

2.8 Hopp

Instruksjonen for å hoppe hetet `jmp`.

```
1     jmp dit
2 dit:
```

Betinget hopp Man kan angi at flaggene skal avgjøre om man skal hoppe.

```
1  jz      dit    # Hopp om Z(ero)
2  jnz     dit    # Hopp om ikke Z
3  jc      dit    # Hopp om C(arry)
4  jnc     dit    # Hopp om ikke C
5  js      dit    # Hopp om S(ign)
6  jns     dit    # Hopp om ikke S
7  jo      dit    # Hopp om O(verflow)
8  jno     dit    # Hopp om ikke O
9  jp      dit    # Hopp om P(arity)
10 jnp     dit    # Hopp om ikke P
```

Testing Flaggene kan settes som følge av vanlige instruksjoner:

```
1     .globl  abs1
2 abs1: movl  4(%esp), %eax
3       addl  $0, %eax
4       jns   ret2
5       negl  %eax
6 ret2: ret
```

Alternativt kan vi eksplisitt sjekke to verdier mot hverandre med instruksjonen `cmp-`:

```
1      .globl   abs2
2 abs2: movl    4(%esp), %eax
3       cmpl    $0, %eax
4       jns     ret1
5       negl    %eax
6 ret1: ret
```

En tredje mulighet er `test-`.

```
1       .globl   s3
2 s3:     tesl   $0x80000000, 4(%esp)
3         jz   a3_pos
4         movl   $1,%eax
5         jmp  a3_x
6 a3_pos: movl   $0, %eax
7 a3_x:   ret
```

2.9 Alignment

Brukeren kan angi at variabler skal være *alignet*, dvs ikke krysse ordgrenser: `.align n`. Denne spesifikasjonen får assembleren til å legge inn 0 eller flere byte med ett eller annet inntil adressen har n 0-bit sist.

2.10 Bit-operasjoner

Det finnes fire operasjoner for å jobbe med enkelt-bit:

btl Gjør ingenting.

btcl Snur bit-et.

btrl Nuller bit-et.

btsl Setter bit-et.

Alle kopierer dessuten det opprinnelige bit-et til C-flagget.

```
btl    $2, %eax    # Sjekker bit 2 i EAX.
```

2.11 Funksjonskall

De seks første parameterne ligger i %RDI, %RSI, %RDX, %RCX, %R8, %R9. Øvrige parametere ligger på stakken. Parameterregisterne samt %R10 og %R11 er frie registere, de øvrige er bundne. Returverdien skal ligge i %RAX.

3 Stakken

Stakken er veldig sentral i x86-arkitekturen til rutinekall, parameteroverføring, lagring av mellomresultater og plass til lokale variabler. Av historiske grunner vokser stakken mot *lavere* adresser.

Legge elementer på stakken Instruksjonene `pushw` og `pushl` legger verdier på stakken.

Fjerne elementer fra stakken Til dette brukes `popw` og `popl`.

4 Flyt-tall

Tall med desimalkomma kan skriver på flere måter:

$$8388708,0$$

$$8,388708 \cdot 10^6$$

$$8,39 \cdot 10^6$$

De to siste ($\pm M \cdot G^E$) er såkalte *flyt-tall* og består av:

- Mantisse (M).
- Grunntall (G).
- Eksponent (E).
- Fortegn.

Normalisert En normalisert mantisse er en binærbrøk med følgende egenskap:

$$1 \leq M < 2$$

4.1 Standarden IEEE 754 for 32-bits flyt-tall

S	Eksponent	Mantisse

S er fortegnet; 0 for positiv og 1 for negativ.

Grunntallet er 2.

Eksponenten er på 8 bit og lagres med fast tillegg 127.

Mantissen er helst normalisert og på 24 bit, men kun de 23 etter binærkommaet lagres.

Mantissen er på 24 bit, og $2^{24} \approx 1,7 \cdot 10^7$.

Eksempel $1,0_{10} = 1,0_2$ som er normalisert. Eksponenten er $0 + 127 = 127 = 1111111_2$. Fortegnet er 0.

0	01111111	00000000000000000000000

Som spesialkonvensjon er 0 representert av kun 0-bit

Eksempel 2 Hvordan lagres $-12,8125$?

$$12,8125_{10} = 1100,1101_2 = 1,1001101_2 \times 2^3$$

Eksponenten er $3 + 127 = 130 = 10000010_2$. Fortegnet er 1.

1	10000010	10011010000000000000000

Det største tallet er omtrent $2^{254-127} \times 2 \sim 3,4 \cdot 10^{38}$. Eksponenten 0 er reservert for unormaliserte tall og tallet 0, eksponenten 255 for ∞ og `NAN`.

0	11111110	11111111111111111111111

4.2 Regne med flyt-tall

x86 har en egen flyt-tallsprosessor: x87. Den har egne registere `ST(0)`–`ST(7)` som brukes som en stakk; de inneholder `double`-verider. `ST(0)` (ofte bare kalt `ST`) er toppen.

Konstanter

fldz Dytter 0.0 på stakken.

fld1 Dytter 1.0 på stakken.

Lese fra minnet/stakken

flds var Dytter float var på stakken.

fldl var Dytter double var på stakken.

fld %st(1) Dytter kopi av ST(x) på stakken.

Skrive til minnet/stakken

fsts var Skriver ST(0) til var som float.

fstl var Skriver ST(0) til var som double.

fst %st(4) Kopierer ST(0) til ST(x).

fstps var Som instruksjonene over, popper stakken etterpå.

fstpl var

fstp %st(5)

4.2.1 Aritmetiske operasjoner

```
1 fadds   var     # ST(0) += float var
2 faddl   var     # ST(0) += double var
3 fadd    %st(4)  # ST(0) += ST(x)
4 faddp           # ST(1) = ST(0)+ST(1); popp
5 fiadds  ivar    # ST(0) += short ivar
6 fiaddl  ivar    # ST(0) += long ivar
```

Tilsvarende operasjoner finnes for subtraksjon, multiplikasjon og divisjon:

```
1 fsubs   var   # ST(0) -= float var
2 fmuls   var   # ST(0) *= float var
3 fdivs   var   # ST(0) /= float var
```

4.3 Sammenligninger

```
1 ftst            # Sammenlign ST(0) med 0.0
2 fcoms   ivar    # Sammenlign ST(0) med short var
3 fcoml   ivar    # Sammenlign ST(0) med long var
4 fcom    %st(7)  # Sammenlign ST(0) med ST(x)
5 fcom            # Sammenlign ST(0) med ST(1)
```

Likedan som over, men popper etterpå: legg på en `p` etter `fcom`. `fcompp` popper to ganger.