



# Intel 64-bits arkitektur

Introduktion till assemblerprogrammering för x86\_64

2014-03-11

Blekinge Tekniska Högskola

Carina Nilsson

## Innehåll

1 Inledning.....	3
2 Introduktion till Intels arkitektur.....	3
2.1 Mera om Intels arkitektur .....	3
3 Intel assembler .....	4
3.1 Register.....	4
3.2 Assemblerinstruktioner .....	5
3.2.1 Minnesadresser .....	5
3.2.2 Flytta adress till register .....	5
3.2.3 Flytta data.....	6
3.2.4 Aritmetik.....	6
3.2.5 Speciella aritmetiska operationer .....	7
3.2.6 Villkorliga och ovillkorliga hopp .....	7
3.2.7 Subrutiner och stack.....	8
3.3 Anropskonvention .....	8
3.3.1 Parameteröverföring.....	8
3.3.2 Ansvar för att bevara registerinnehåll.....	8
3.3.3 Returvärden.....	8
3.4 Vanliga assemblerdirektiv .....	8
4 Några programexempel.....	9
4.1 Hello world! .....	9
4.2 Tvåpotenser.....	9
4.3 Fibonacci-talen .....	10
4.4 Maxvärdet av tre värden .....	11
5 Lista över figurer.....	12

## 1 Inledning

Eftersom kursen innehåller en grundlig genomgång av MIPS arkitektur kommer det här dokumentet fokusera på skillnader mellan den här arkitekturen och MIPS. Det är alltså ett dokument som förutsätter grundläggande kunskaper i datorarkitektur rent allmänt. Syftet med dokumentet är att ge grundläggande förståelse för hur Intels 64-bits processorer (och kompatibla från AMD) kan programmeras, och tillräckliga kunskaper för att kunna implementera assemblerrutiner i AT&T-syntax som kan länkas samman med högnivåmoduler skrivna i C och kompilerade med kompilatorn gcc.

## 2 Introduktion till Intels arkitektur

Intel är till skillnad från MIPS ingen RISC-arkitektur. Det finns en stor mängd mer eller mindre vanliga instruktioner i instruktionsrepertoaren. Instruktionsformatet är komplicerat. Den inre arkitekturen använder ändå pipeline och använder avancerad teknik för att exekvera instruktioner parallellt, med en efterföljande synkronisering. Det görs även avancerad gissning av väg genom programmet. Intel är en CISC-arkitektur, med delvis underliggande RISC-arkitektur, så man skulle kunna kalla den en hybridmaskin. Det gör att arkitekturen är synnerligen komplex.

Processorn har ett antal register och minnet är adresserbart bytevis.

### 2.1 Mera om Intels arkitektur

Intels arkitektur är mycket komplicerad och svår att ge en snabb översiktsbild av. Det finns en del beskrivningar på Intels hemsidor, men det är svårt att få en bra förståelse utifrån dem och det mesta liknar reklamtext som talar om hur effektivt det fungerar. Genom att göra mer och mer komplicerade lösningar och ta vissa delar som är bra ifrån RISC-arkitekturer har man lyckats utveckla en i grunden gammal arkitektur till att bli snabb. Arkitekturens komplexitet är dock slående. Instruktionsmässigt är det en CISC-maskin med en mängd komplicerade instruktioner. Under ytan är det dock en modern maskin som använder sig av en pipeline med ca 14 exekveringssteg. Man kan betrakta maskinens funktion som om någon satt och läste och avkodade komplicerade instruktioner och omvandlade till enkla RISC-betonade instruktioner och kastade iväg dem till pipelinen. På den låga nivån kan vissa instruktioner exekveras i godtycklig ordning, så kallad *"Out-of-Order"*-exekvering. Ett problem som uppstår med pipelinen är att man kan behöva resultat av en tidigare instruktion, som på grund av parallelliteten inte är färdigexekverad ännu. Det blir även problem med hämtning av kommande instruktioner vid förgreningar i programmet. Man kan ha hunnit långt förbi förgreningspunkten innan man vet vilken väg man ska ta när pipelinen har så många steg. Intel använder en komplicerad prediktion vid hopp (*Deep branch prediction*), som mer eller mindre smart gissar vilken väg genom programmet som är aktuell och hämtar och avkodar instruktioner längs den vägen och på så sätt håller pipelinen fylld. En flödesanalys utförs också för att på ett optimalt sätt utnyttja *"Out-of-Order"*-exekvering. Slutligen utförs även så kallad spekulativ exekvering, dvs. exekveringsväg gissas och exekveras, eventuellt exekveras till och med flera vägar samtidigt. När villkoren i programmet väl blivit kända (uppfyllda eller inte uppfyllda) tas resultaten om hand om de är korrekta eller kastas om de är fel. För att möjliggöra den här processen finns särskilda extra uppsättningar av temporära cpu-register (som inte programmeraren själv kan komma åt).

## 3 Intel assembler

Här redovisas bara en delmängd av maskinens assemblerspråk. Språket är egentligen inte bara knutet till den maskin man använder, utan också till vilken assembler (översättarprogrammet) som används. Här kan en varning vara på plats, för olika assembler har olika syntax. Gnu som tillhandahåller C-kompilatorn `gcc` och assemblern `gas` använder så kallad AT&T-syntax (vanlig i Unix/Linux-världen), medan andra kan använda så kallad Intel-syntax (vanlig i Windows-världen). En väsentlig och störande skillnad mellan syntaxerna är att ordningen mellan operander är omkastad. I det här dokumentet beskrivs AT&T-syntax, eftersom det är det vi använder på våra maskiner i labbet.

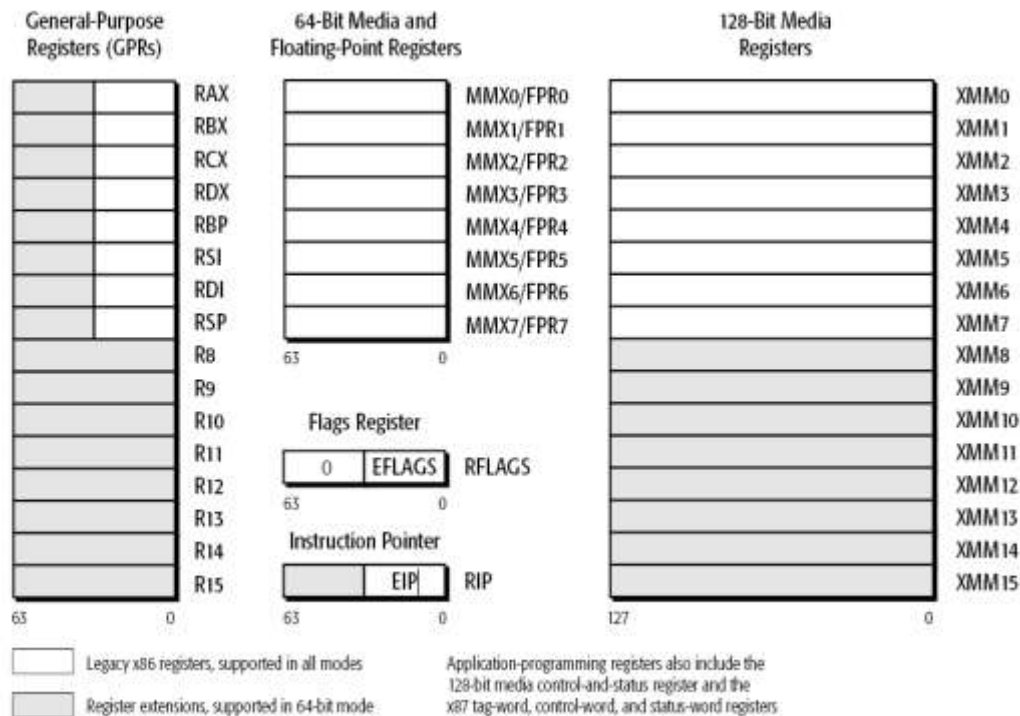
### 3.1 Register

Det finns i princip 16 generella register om 64 bitar, men de kan också användas som 32-, 16- eller 8-bitsregister med andra namn enligt *Figur 1*. En del register har en speciell användning. Exempelvis är ett av dem stackpekare (`rsp`). Det finns dessutom ett antal segmentregister, som används på lite olika sätt beroende på hur man hanterar minne, men för vår enkla programmering behöver vi egentligen inte bry oss om det. För vår del räcker det att använda registren vi ser i *Figur 1*.

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
<b>rax</b>	<b>eax</b>	<b>ax</b>	<b>al</b>
<b>rbx</b>	<b>ebx</b>	<b>bx</b>	<b>bl</b>
<b>rcx</b>	<b>ecx</b>	<b>cx</b>	<b>cl</b>
<b>rdx</b>	<b>edx</b>	<b>dx</b>	<b>dl</b>
<b>rsi</b>	<b>esi</b>	<b>si</b>	<b>sil</b>
<b>rdi</b>	<b>edi</b>	<b>di</b>	<b>dil</b>
<b>rbp</b>	<b>ebp</b>	<b>bp</b>	<b>bpl</b>
<b>rsp</b>	<b>esp</b>	<b>sp</b>	<b>spl</b>
<b>r8</b>	<b>r8d</b>	<b>r8w</b>	<b>r8b</b>
<b>r9</b>	<b>r9d</b>	<b>r9w</b>	<b>r9b</b>
<b>r10</b>	<b>r10d</b>	<b>r10w</b>	<b>r10b</b>
<b>r11</b>	<b>r11d</b>	<b>r11w</b>	<b>r11b</b>
<b>r12</b>	<b>r12d</b>	<b>r12w</b>	<b>r12b</b>
<b>r13</b>	<b>r13d</b>	<b>r13w</b>	<b>r13b</b>
<b>r14</b>	<b>r14d</b>	<b>r14w</b>	<b>r14b</b>
<b>r15</b>	<b>r15d</b>	<b>r15w</b>	<b>r15b</b>

*Figur 1. Namngivning av register och delar av register i Intels x86-64-arkitektur*

Assemblern känner igen registernamn om man låter dem börja med `%`. Det betyder att `rax` kan adresseras genom att skriva `%rax` i koden. Man måste också komma ihåg att `rax`, `eax`, `ah` och `al` är samma register. Det gäller att vara medveten om att om man har ett värde i `rax` och flyttar något till `al`, så kommer enbart de 8 lägsta bitarna i `rax` att påverkas, resten ligger kvar som förut. Det betyder att man får hantera det på bästa sätt själv genom att exempelvis nollställa valda delar av registret.



Figur 2. Fullständig bild av registren hos x86-64-arkitekturen ur programmerarens synvinkel

## 3.2 Assemblerinstruktioner

Intel har ett mer liberalt instruktionsformat än RISC-maskiner. RISC-arkitekturer har normalt ett fåtal instruktioner som flyttar från minne till register (*load*) och från register till minne (*store*), medan alla andra instruktioner har operanderna i register eller möjligen direkt i instruktionen (*immediate*). Intel-assemblern är inte alls så restriktiv.

Man brukar ange formatet (storleken) på data som instruktionen ska arbeta på så att man tydligt vet vad som händer vid exekveringen. Det sker genom att man ger instruktionen ett suffix (en extra bokstav sist). Suffixet för 64 bitar är *q*, för 32 bitar är det *l*, för 16 bitar är det *w* och för 8 bitar är det *b*.

### 3.2.1 Minnesadresser

När minnesadresser används i instruktionerna skriver man normalt på något av följande sätt. Ett sätt är att man definierat en symbol i programmets datasektion med kolon (t ex *SUM:* ). Då kan man använda symbolen *SUM* som adress. Ett annat sätt är att ange adressen som en offset och ett register som innehåller basadressen t ex *4(%rax)* , vilket betyder att innehållet i *%rax* ökat med 4 används som adress. Data som inte ligger i en minnescell, utan direkt i instruktionen (*immediate*)anges med ett dollartecken framför, t ex *\$4*. Observera att *immediate data* representeras som 32-bits tal.

### 3.2.2 Flytta adress till register

Det finns en speciell instruktion som flyttar en adress till ett register, som heter *leaq* (*load effective address*).

Exempel:

```
leaq SUM, %r11
```

### 3.2.3 Flytta data

Man kan flytta data mellan register och minne på båda håll samt mellan register med samma instruktion. Man kan däremot inte normalt flytta data direkt mellan två minnesadresser.

Grundinstruktionen för att flytta data är `mov` och den kan förses med önskat suffix. Varianten `movl` är ett snällt specialfall som automatiskt fyller ut destinationsregistrets höga halva med nollor om flytten sker till ett register.

Instruktion	Resultat	Beskrivning
<code>movq S, D</code>	$D \leftarrow S$	flytta 64-bits ord
<code>movabsq I, R</code>	$R \leftarrow I$	flytta 64-bits ord
<code>movslq S, R</code>	$R \leftarrow \text{SignExtend}(S)$	flytta 32-bits ord med teckentillägg
<code>movsbq S, R</code>	$R \leftarrow \text{SignExtend}(S)$	flytta 8-bits ord med teckentillägg
<code>movzbq S, R</code>	$R \leftarrow \text{ZeroExtend}(S)$	flytta 8-bits ord utfyllt med nollor
<code>pushq S</code>	$R[\%rsp] \leftarrow R[\%rsp] - 8; M[R[\%rsp]] \leftarrow S$	lägg $S$ överst på stacken
<code>popq D</code>	$D \leftarrow M[R[\%rsp]]; R[\%rsp] \leftarrow R[\%rsp] + 8$	hämta till $D$ från överst på stacken

$S$  = källa (source),  $D$  = destination,  $I$  = immediate data,  $R$  = register

Figur 3. Speciella 64-bitsinstruktioner för att flytta data.

Det finns speciella instruktioner för att hantera stacken. Med stack menas ett helt vanligt datautrymme. Det som hårdvaran hanterar är instruktionerna `push` och `pop` som betraktar innehållet i `%rsp` som en stackpekare. Instruktionen `pushq S` dekrementerar innehållet i `%rsp` med 8 och sparar innehållet i  $S$  på den plats i minnet som `%rsp` nu pekar på. Instruktionen `popq D` överför innehållet från den minnesplats som `%rsp` pekar på till  $D$  och inkrementerar därefter `%rsp` med 8.

### 3.2.4 Aritmetik

Det finns instruktioner för heltalsaritmetik, exempelvis addition och subtraktion. Operanderna kan ligga i minnet eller i register, dock kan inte båda operanderna ligga i minnet. Det finns också logiska instruktioner som utför bitvisa logiska operationer på operanderna såsom `and`, `or` och `xor`.

Instruktion	Resultat	Beskrivning
<code>leaq S, D</code>	$D \leftarrow \&S$	ladda effektiv adress
<code>incq D</code>	$D \leftarrow D + 1$	inkrement (räkna upp med 1)
<code>decq D</code>	$D \leftarrow D - 1$	dekrement (räkna ned med 1)
<code>negq D</code>	$D \leftarrow -D$	negation (teckenväxling)
<code>notq D</code>	$D \leftarrow \sim D$	invertera alla bitar
<code>addq S, D</code>	$D \leftarrow D + S$	addition
<code>subq S, D</code>	$D \leftarrow D - S$	subtraktion
<code>imulq S, D</code>	$D \leftarrow D * S$	multiplikation
<code>xorq S, D</code>	$D \leftarrow D \wedge S$	bitvis xor
<code>orq S, D</code>	$D \leftarrow D \vee S$	bitvis eller
<code>andq S, D</code>	$D \leftarrow D \& S$	bitvis och
<code>salq k, D</code>	$D \leftarrow D \ll k$	bitvis vänsterskift $k$ positioner
<code>shlq k, D</code>	$D \leftarrow D \ll k$	samma som ovan
<code>sarq k, D</code>	$D \leftarrow D \gg k$	aritmetiskt högerskift $k$ positioner
<code>shrq k, D</code>	$D \leftarrow D \gg k$	logiskt högerskift $k$ positioner

Figur 4. Aritmetiska operationer för 64 bitar

### 3.2.5 Speciella aritmetiska operationer

När man multiplicerar kan lätt resultatet bli för stort för registret resultatet ska läggas i. När två tal med 64 bitar multipliceras kan resultatet bli upp till 128 bitar. För att kunna hantera en så stor produkt används registren `%rdx` och `%rax` för att tillsammans forma ett register med 128 bits kapacitet. Dessa multiplikationsinstruktioner tar en operand från `%rax` och en från `S`, och resultatet läggs i de 128 bitar som utgörs av `%rdx` och `%rax` tillsammans. Även divisionsinstruktionerna använder dessa båda register på ett lite speciellt sätt. Täljartalet läggs i `%rdx:%rax` som då betraktas som ett tal med 128 bitar, och nämnartalet i tas ur `S`. Heltalsdivision ger två resultat, en kvot som hamnar i `%rax` och en rest som hamnar i `%rdx`. Specialinstruktionen `cqto` kan användas för att lägga till teckenutfyllnad från `%rax` till `%rdx` när man ska dela ett teckensatt tal.

Instruktion	Resultat	Beskrivning
<code>imulq S</code>	$R[\%rdx]: R[\%rax] \leftarrow S \times R[\%rax]$	Full multiplikation med teckensatta tal
<code>mulq S</code>	$R[\%rdx]: R[\%rax] \leftarrow S \times R[\%rax]$	Full multiplikation med teckenlösa tal
<code>cltq</code>	$R[\%rax] \leftarrow \text{SignExtend}(R[\%eax])$	Konvertera <code>%eax</code> till 64 bitar
<code>cqto</code>	$R[\%rdx]: R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	Konvertera <code>%rax</code> till 128 bitar
<code>idivq S</code>	$R[\%rdx] \leftarrow R[\%rdx]: R[\%rax] \bmod S ;$ $R[\%rax] \leftarrow R[\%rdx]: R[\%rax] / S$	Division med teckensatta tal
<code>divq S</code>	$R[\%rdx] \leftarrow R[\%rdx]: R[\%rax] \bmod S ;$ $R[\%rax] \leftarrow R[\%rdx]: R[\%rax] / S$	Division med teckenlösa tal

Figur 5. Speciella aritmetiska operationer. Registerparet `%rdx` och `%rax` ses som ett 128-bits register.

### 3.2.6 Villkorliga och ovillkorliga hopp

För att styra exekveringsordningen och införa konstruktioner såsom selektion och iteration måste vi använda instruktioner för hopp, villkorliga eller ovillkorliga. Vid de villkorliga hoppen kan man behöva utföra en "test"-instruktion först som uppdaterar flaggregistren och gör ett villkorligt hopp med rätt villkor möjligt. Här visas bara ett litet urval av de mest användbara och nödvändiga villkorliga hoppinstruktionerna. Det finns ca 40! olika.

Instruktion	Jämförelse baserad på	Beskrivning
<code>cmpq S<sub>2</sub>, S<sub>1</sub></code>	$S_1 - S_2$	Jämför 64-bits dataord som teckensatta tal. OBS! Ordningföljden!
<code>testq S<sub>2</sub>, S<sub>1</sub></code>	$S_1 \& S_2$	Testar 64-bits dataord

Figur 6. Dessa jämförande operationer kan användas för godtycklig datastorlek med önskat suffix i stället för `q`.

Instruktion	Beskrivning
<code>jmp label</code>	ovillkorligt hopp till <code>label</code>

Figur 7. Ovillkorligt hopp

Instruktion	Beskrivning
<code>je label</code>	hoppa om föregående jämförelse är lika med noll
<code>jne label</code>	hoppa om föregående jämförelse inte är lika med noll
<code>jg label</code>	hoppa om resultat av jämförelse större än noll
<code>jl label</code>	hoppa om resultat av jämförelse mindre än noll
<code>jge label</code>	hoppa om resultat av jämförelse större än eller lika med noll
<code>jle label</code>	hoppa om resultat av jämförelse mindre än eller lika med noll

Figur 8. Villkorliga hopp att göra efter jämförelse.

### 3.2.7 Subrutiner och stack

Det finns ytterligare en hoppvariant som är mycket speciell, och det är instruktionen för hopp till subrutin. Det heter `call` och lägger automatiskt en återhopsadress (programräknarens läge) om 64 bitar på stacken. För att göra återhopp från subrutin används instruktionen `ret`, som automatiskt hämtar återhopsadressen från stacken till programräknaren.

## 3.3 Anropskonvention

För att assemblerrutiner ska vara användbara från högnivåspråk och vice versa är det viktigt att man håller sig till samma anropskonvention (*calling convention*) som den omgivning man ska samarbeta med använder. I vårt system använder vi Linux Ubuntu och GNU-kompilatorn `gcc`. I andra omgivningar kan andra konventioner gälla, så det som beskrivs här fungerar till exempel inte vid programmering med verktyget Visual Studio i Windows-miljö. De plattformsspecifika anropskonventionerna kallas ibland ABI (*Application Binary Interface*). Observera också att anropskonventionen för 32-bits Linux ser helt annorlunda ut.

### 3.3.1 Parameteröverföring

Heltalsparametrar och parametrar som är pekare (adresser) placeras i följande ordning: Första parametern i `%rdi`, andra parametern i `%rsi`, tredje parametern i `%rdx`, fjärde parametern i `%rcx`, femte parametern i `%r8`, sjätte parametern i `%r9`. Har man fler än sex parametrar överförs de resterande via stacken. Flyttalsparametrar överförs via de speciella flyttalsregistren benämnda `%xmm` (se fig. 2). Vid anrop till funktioner som tar ett variabelt antal parametrar ska `%rax` innehålla information om hur många vektorregister som används (`%xmm`-register).

### 3.3.2 Ansvar för att bevara registerinnehåll

Registren `%rbp`, `%rbx` och `%r12` – `%r14` "ägs" av den som anropar en funktion (*caller owned*). Det betyder att det är den anropade funktionens ansvar att bevara registerinnehållet, dvs. återställa dem till samma värden som de hade när funktionen anropades innan återhopp. I praktiken betyder det att om funktionen använder dessa register behöver de sparas först och återställas sist i funktionen exempelvis genom att använda stacken. Alla andra register är det upp till den som anropar en funktion att spara undan, eftersom funktionen får förändra deras innehåll.

### 3.3.3 Returvärden

Returvärden som är heltal eller pekare ska placeras i `%rax`. Har man returvärde av flyttalstyp ska det placeras i `%xmm0`.

## 3.4 Vanliga assemblerdirektiv

Assemblerdirektiv är inte programinstruktioner, utan berättar för översättarprogrammet hur olika delar av koden ska tolkas och placeras i minnet. Här följer en lista på de vanligaste direktiven:

<code>.align <i>n</i></code>	gör adressen som nästa instruktion eller datastruktur lagras på jämt delbar med $2^n$ (dvs slutar på <i>n</i> antal nollor binärt)
<code>.ascii <i>str</i></code>	teckensträngen <i>str</i> lagras i minnet
<code>.asciz <i>str</i></code>	teckensträngen <i>str</i> lagras i minnet följd av <code>NULL</code> -tecknet (ascii-kod 0)
<code>.byte <i>b1</i>, ..., <i>bn</i></code>	värdena <i>b1</i> , ..., <i>bn</i> lagras i minnet i en byte (8 bitar)vardera
<code>.data</code>	följande avsnitt specificerar data



- `.global sym`      medför att symbolen **sym** är global och kan användas i andra filer
- `.long l1, ..., ln`      värdena **l1, ..., ln** lagras i minnet i 32 bitar vardera
- `.quad q1, ..., qn`      värdena **q1, ..., qn** lagras i minnet i 64 bitar vardera
- `.space n`      reserverar ett **n** byte stort minnesutrymme
- `.text`      följande avsnitt innehåller programkod (instruktioner)
- `.word w1, ..., wn`      värdena **w1, ..., wn** lagras i minnet i 16 bitar vardera

## 4 Några programexempel

Här följer några enkla programexempel skrivna i 64-bits Intel-assembler för Linux.

### 4.1 Hello world!

Programmet skriver ut texten Hello world! i terminalfönstret.

```
.text                    # deklaration av text-sektion (kodavsnitt) .global
main    # startpunkt som länkaren känner igen
main:  movq $message, %rdi    # adress till strängen
       xorq %rax, %rax        # printf tar ett variabelt antal argument
       # så %rax behöver sättas till noll i vårt fall
       call printf
       call exit

.data    # deklaration av data-sektion
message: .asciz "Hello world!\n"      # definition av sträng
```

### 4.2 Tvåpotenser

Programmet nedan visar alla tal som är tvåpotenser mellan  $2^0$  och  $2^{31}$ .

```
.text
.global            main
main:  movq  $1, %rsi            # aktuellt värde
       movq  $32, %rdi          # räknare
L1:    pushq  %rsi               # lägg registervärde på stacken
       pushq  %rdi               # lägg registervärde på stacken
       movq  $format, %rdi      # formatsträngens adress
       # andra argumentet (värdet) ligger redan i %rsi
       xorq  %rax, %rax            # nollställ %rax
       call  printf
       popq  %rdi               # återhämta registervärde (OBS! ordningen)
       popq  %rsi               # återhämta registervärde
       addq  %rsi, %rsi            # dubblera värdet
       dec   %rdi               # räkna ned räknare
       jne   L1                # Hoppa till L1 om det inte blev noll
       call  exit

.data
format: .asciz "%ld\n"
```

## 4.3 Fibonacci-talen

Programmet nedan skriver ut de 40 första Fibonacci-talen

```
.text
.global main
main: movq    $40, %rcx      # initiera %rcx till 40 (räknare)
      xorq    %rax, %rax     # nollställ %rax (aktuellt tal)
      movq    $1, %rbx      # initiera %rbx (nästa tal)
print: pushq  %rax          # lägg registervärde på stacken
      pushq  %rcx          # lägg registervärde på stacken
      movq    $format, %rdi # formatsträngens adress
      movq    %rax, %rsi    # andra argumentet till %rsi
      xorq    %rax, %rax     # nollställ %rax
      call    printf
      popq    %rcx         # återhämta registervärde
      popq    %rax         # återhämta registervärde
      movq    %rax,%rdx     # spara tal
      movq    %rbx, %rax    # och skifta så nästa tal blir aktuellt
      addq    %rdx, %rbx    # beräkna nästa tal
      decq    %rcx         #räkna ned räknaren
      jne     print        # om det inte blev noll beräkna ett nytt tal
      call    exit

.data
format: .asciz "%d\n"
```

## 4.4 Maxvärdet av tre värden

Assemblerfunktion som returnerar det största av tre argument.

Ny instruktion `cmovl` – *Conditional move if less*.

```
.text
.global MaxOfThree
MaxOfThree:
    cmpl    %esi, %edi    #jämför argument 1 och 2
    cmovl   %esi, %edi    #flytta %esi-värdet till %edi om det var större
    cmpl    %edx, %edi    #jämför med argument 3
    cmovl   %edx, %edi    #flytta %edx-värdet till %edi om det var större
    movl    %edi, %eax    #lägg returvärdet i %eax
    ret
```

Användning av ovanstående funktion i ett C-program som länkats med assemblerfilen vid kompilering till körbart program

```
#include <stdio.h>

extern int MaxOfThree(int, int, int);

int main()
{
    printf("Maxvärdet av talen 1, -4 och -7 är %d\n", MaxOfThree(1,-4,-7));
    printf("Maxvärdet av talen 2, -6 och 1 är %d\n", MaxOfThree(2, -6, 1));
    printf("Maxvärdet av talen 2, 3 och 1 är %d\n", MaxOfThree(2, 3, 1));
    printf("Maxvärdet av talen -2, 4 och 3 är %d\n", MaxOfThree(-2, 4, 3));
    printf("Maxvärdet av talen 2, -6 och 5 är %d\n", MaxOfThree(2, -6, 5));
    printf("Maxvärdet av talen 2, 4 och 6 är %d\n", MaxOfThree(2, 4, 6));
    return 0;
}
```

## 5 Lista över figurer

*Figur 1. Namngivning av register och delar av register i Intels x86-64-arkitektur*

Källa: "x64 Architecture", Microsoft Developer Network, januari 2014. [Online] Tillgänglig: <http://msdn.microsoft.com/en-us/library/ff561499.aspx>. [Hämtad: 21 februari, 2014].

*Figur 2. Fullständig bild av registren hos x86-64-arkitekturen ur programmerarens synvinkel*

Källa: A Karpov, E. Ryzhkov, "AMD64 (EMT64) architecture", oktober 2008. [Online] Tillgänglig: <http://www.viva64.com/en/a/0029/>. [Hämtad: 21 februari, 2014].

*Figur 3. Speciella 64-bitsinstruktioner för att flytta data.*

*Figur 4. Aritmetiska operationer för 64 bitar.*

*Figur 5. Speciella aritmetiska operationer. Registerparet %rdx och %rax ses som ett 128-bits register.*

*Figur 6. Dessa jämförande operationer kan användas för godtycklig datastorlek med önskat suffix i stället för q.*

*Figur 7. Ovillkorligt hopp*

*Figur 8. Villkorliga hopp att göra efter jämförelse.*