Intel 64-bits arkitektur

Introduktion till assemblerprogrammering för x86_64

2017-04-20
Blekinge Tekniska Högskola
Carina Nilsson

Reviderad 2023-04-17

Innehåll

1 Inledning

Eftersom kursen innehåller en grundlig genomgång av en ARM-arkitektur kommer det här dokumentet fokusera på skillnader mellan Intels arkitektur och ARM. 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 ARM ingen RISC-arkitektur. Det finns en stor mängd mer eller mindre vanliga instruktioner i instruktionsrepertoaren. Instruktionsformatet är komplicerat. Den inre arkitekturen använder ändå pipeliner och använder avancerad teknik för att exekvera instruktioner parallellt, med en efterföljande synkronisering. Det görs även avancerad gissning av exekveringsväg 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 bytevis adresserbart.

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 ladda in instruktioner långt förbi förgreningspunkten innan man vet vilken väg programmet ska följa 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 hålls 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.

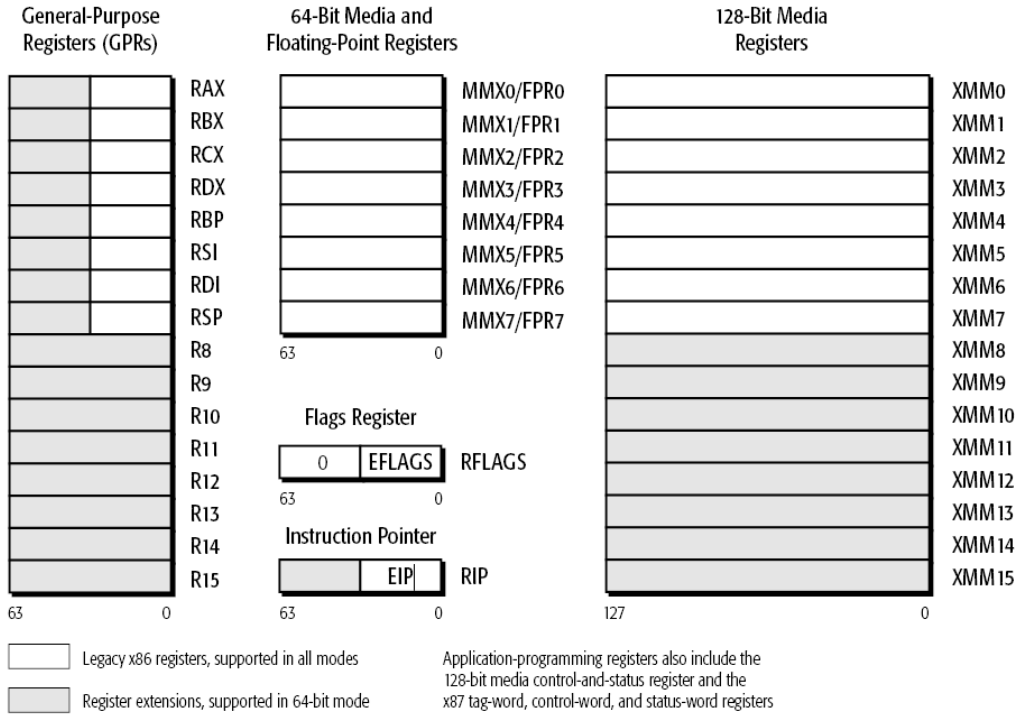
3.1 Register

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

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 som exempelvis ARM. 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 ange 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 `lea` (*load effective address*).
Exempel:

```
leaq sum, %r11
```

3.2.3 Flytta data

Man kan flytta data mellan register och minne på båda håll. Man kan också flytta data mellan olika register. 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
`movq` *S, D*	*D* ← *S*	flytta 64-bits ord
`movabsq` *I, R*	*R* ← *I*	flytta 64-bits ord
`movslq` *S, R*	*R* ← **SignExtend**(*S*)	flytta 32-bits ord med teckentillägg
`movsbq` *S, R*	*R* ← **SignExtend**(*S*)	flytta 8-bits ord med teckentillägg
`movzbq` *S, R*	*R* ← **ZeroExtend**(*S*)	flytta 8-bits ord utfyllt med nollor
`pushq` *S*	**R**[%rsp] ← **R**[%rsp] - 8; **M**[**R**[%rsp]] ← *S*	lägg *S* överst på stacken
`popq` *D*	*D* ← **M**[**R**[%rsp]]; **R**[%rsp] ← **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 innehåller 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, t ex addition och subtraktion. Operanderna kan då 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
`leaq` *S, D*	*D* ← *&S*	ladda effektiv adress
`incq` *D*	*D* ← *D* + 1	inkrement (räkna upp med 1)
`decq` *D*	*D* ← *D* - 1	dekrement (räkna ned med 1)
`negq` *D*	*D* ← -D	negation (teckenväxling)
`notq` *D*	*D* ← ~D	invertera alla bitar
`addq` *S, D*	*D* ← *D* + *S*	addition
`subq` *S, D*	*D* ← *D* - *S*	subtraktion
`imulq` *S, D*	*D* ← *D* * *S*	multiplikation
`xorq` *S, D*	*D* ← *D* ^ *S*	bitvis xor
`orq` *S, D*	*D* ← *D* | *S*	bitvis eller
`andq` *S, D*	*D* ← *D* & *S*	bitvis och
`salq` *k, D*	*D* ← *D* << *k*	bitvis vänsterskift *k* positioner
`shlq` *k, D*	*D* ← *D* << *k*	samma som ovan
`sarq` *k, D*	*D* ← *D* >> *k*	aritmetiskt högerskift *k* positioner
`shrq` *k, D*	*D* ← *D* >> *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
`imulq` *S*	**R**[%rdx]: **R**[%rax] ← *S* × **R**[%rax]	Full multiplikation med teckensatta tal
`mulq` *S*	**R**[%rdx]: **R**[%rax] ← *S* × **R**[%rax]	Full multiplikation med teckenlösa tal
`cltq`	**R**[%rax]← **SignExtend**(**R**[%eax])	Konvertera %eax till 64 bitar
`cqto`	**R**[%rdx]: **R**[%rax] ← **SignExtend**(**R**[%rax])	Konvertera %rax till 128 bitar
`idivq` *S*	**R**[%rdx] ← **R**[%rdx]: **R**[%rax] mod *S* **;** **R**[%rax] ← **R**[%rdx]: **R**[%rax] / *S*	Division med teckensatta tal
`divq` *S*	**R**[%rdx] ← **R**[%rdx]: **R**[%rax] mod *S* **;** **R**[%rax] ← **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
`cmpq` S_2, S_1	S_1 - S_2	Jämför 64-bits dataord som teckensatta tal. OBS! Ordningsföljden!
`testq` S_2, S_1	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
`jmp` *label*	ovillkorligt hopp till *label*

Figur 7. Ovillkorligt hopp

Instruktion	Beskrivning
`je` *label*	hoppa om föregående jämförelse är lika med noll
`jne` *label*	hoppa om föregående jämförelse inte är lika med noll
`jg` *label*	hoppa om resultat av jämförelse större än noll
`jl` *label*	hoppa om resultat av jämförelse mindre än noll
`jge` *label*	hoppa om resultat av jämförelse större än eller lika med noll
`jle` *label*	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 hoppinstruktion som är mycket speciell, och det är instruktionen för hopp till subrutin. Det heter `call` och lägger automatiskt en återhoppsadress (programräknarens läge) om 64 bitar på stacken. För att göra återhopp från subrutin används instruktionen `ret`, som automatiskt hämtar återhoppsadressen från stacken till programräknaren.

3.3 Adresseringsmoder

Adresseringsmod avgör för en instruktion hur data till instruktionen blir åtkomlig. Om instruktionen gör en minnesaccess innefattar det hur adressen till minnet genereras.

3.3.1 Omedelbar adressering (*Immediate*)

Här står datavärdet explicit i instruktionen. Moden markeras med att datavärdet föregås av `$`. Observera att det angivna talet inte kan vara större än 32 bitar.

Exempel:

```
        .global main
        .text

main:
        addl    $14, %eax       #Adderar talet 14 till innehåller i eax (32bit)
```

3.3.2 Registerdirekt adressering

Här arbetar instruktionen enbart mot data i processorns interna register.

Exempel:

```
        .global main
        .text

main:
        addb    %al, %r8b       #Adderar al till r8b (8bit)
```

3.3.3 Absolut adressering

Här anges adressen direkt i instruktionen som ett tal eller en symbol (som också i praktiken är ett tal, som är adressen till den minnesplats den lagras på).

Exempel:

```
.data
symbol: .long 20

        .global main
        .text

main:
        movl    symbol, %eax    #hämtar innehållet från adressen symbol i
                                 #minnet till eax, dvs talet 20
```

3.3.4 Registerindirekt adressering

Här håller ett register adressen till data i minnet till vilken access ska ske.

Exempel:

```
.data
symbol: .word 20

        .global main
        .text

main:
        leaq    symbol, %rcx   #hämtar adressen symbol till rcx
        movw    (%rcx), %r8w   #hämtar innehållet från den adress rcx anger
```

3.3.5 Registerindirekt adressering med offset

Här håller ett register en adress. Offseten anger vilket tal som ska adderas till den adressen för att få effektiv adress.

Exempel:

```
.data
string: .asciz        ”Hello world!”

        .global main
        .text

main:
        leaq    string, %rcx   #hämtar adressen symbol till rcx
        movb    4(%rcx), %r8b  #hämtar innehållet från den adress rcx anger
                               # plus offseten 4 (dvs tecknet ’o’)
```

3.3.6 Bas+index-adressering

Man kan generera adress genom att låta ett register innehålla en basadress och ett annat register index, där den effektiva adressen blir summan av de båda. I den här moden kan även adressen regleras med offset samt hur många byte ett indexsteg ska ta. Adressen genereras ur formatet offset(bas, index, antal byte) och den effektiva adressen blir bas + index *antalbyte+ offset.
Om man önskar att utelämna ett av registren bas och offset i formeln kan man lämna den positionen i uttrycket blank.

```
.data
string: .asciz        ”Hello world!”

        .global main
        .text

main:
        leaq    string, %rcx   #hämtar adressen symbol till rcx
        movq    $4, %rax
        movzbq 4(%rcx,%rax,1), %r8   #hämtar innehållet adress (string+4*1+4)
                               # dvs tecknet ’r’
```

3.4 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 omgivningen man ska samarbeta i använder. I vårt system använder vi Linux 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.

3.4.1 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).

3.4.2 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 innan återhopp till samma värden som de hade när funktionen anropades. 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.

3.4.3 Returvärden

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

3.5 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:

`.align` ***n***	gör adressen som nästa instruktion eller datastruktur lagras på jämt delbar med 2^n (dvs slutar på ***n*** antal nollor binärt)
`.ascii` ***str***	teckensträngen ***str*** lagras i minnet
`.asciz` ***str***	teckensträngen ***str*** lagras i minnet avslutad med `NULL`-tecknet (ascii-kod 0)
`.byte` ***b1, …, bn***	värdena ***b1, .., bn*** lagras i minnet i en byte (8 bitar) vardera
`.data`	följande avsnitt specificerar data
`.global` ***sym***	medför att symbolen ***sym*** är global och kan refereras från 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.

```
        .global main # startpunkt som länkaren känner igen
        .text        # deklaration av text-sektion (kodavsnitt)
main:
        pushq  $0     #för stack alignment 16 bytes
        movq   $message, %rdi
        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:   pushq  $0
        movq   $1, %rsi       # aktuellt värde
        movq   $32, %rdi      # räknare
lLoop:
        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    lLoop  # 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:
        pushq  $0
        movq   $40, %rcx      # initiera %rcx till 40 (räknare)
        xorq   %rax, %rax     # nollställ %rax (aktuellt tal)
        movq   $1, %rbx       # initiera %rbx (nästa tal)
lPrint:
        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    lPrint    # 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