



Assembler för INTEL/AMD 64 bitar (x64) del 1

Arkitektur x64

- 16 generella register
(som jämförelse har 64bits ARM 32 st) (32bits ARM 16 st)
- Ordlängd 64 bitar
- De flesta instruktioner kan jobba mot en operand i register och *en (1)* operand i minnet. (Instruktionerna kan givetvis arbeta med bara registerinnehåll också)
(jämför ARM: Bara register-register förutom LDR och STR)

Den lägsta halvan av
registren (32 bitar)
har eget namn

Register

64-bitsregister	32-bitsregister	16-bitsregister	8-bitsregister
<u>rax</u>	<u>eax</u>	<u>ax</u>	al (ah, hög byte i ax)
<u>rbx</u>	<u>ebx</u>	<u>bx</u>	bl (bh, hög byte i bx)
<u>rcx</u>	<u>ecx</u>	<u>cx</u>	cl (ch, hög byte i cx)
<u>rdx</u>	<u>edx</u>	<u>dx</u>	dl (dh, hög byte i dx)
<u>rsi</u>	<u>esi</u>	<u>si</u>	<u>sil</u>
<u>rdi</u>	<u>edi</u>	<u>di</u>	<u>dil</u>
<u>rbp</u>	<u>ebp</u>	<u>bp</u>	<u>bpl</u>
<u>rsp</u>	<u>esp</u>	<u>sp</u>	<u>spl</u>
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

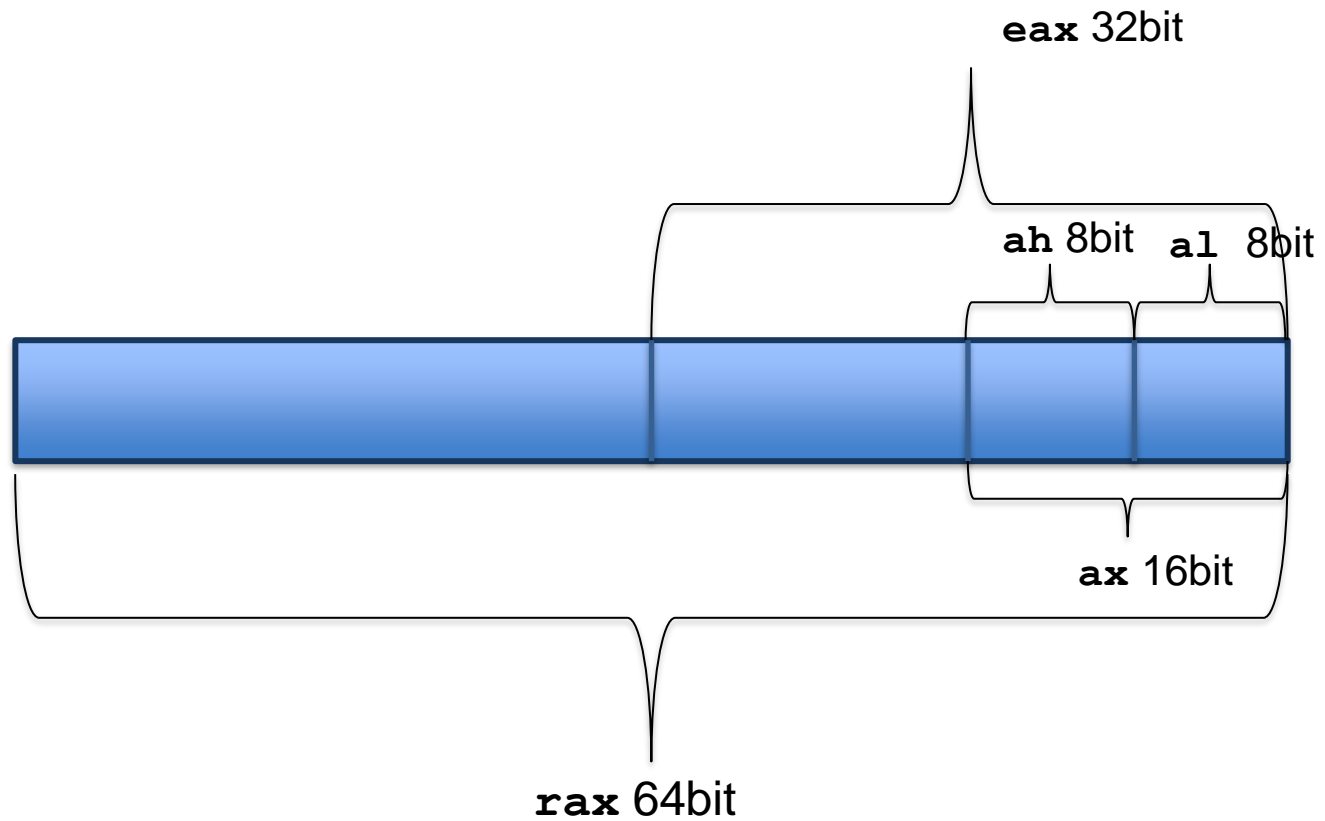
Obs! en rad i
tabellen är
olika delar av
ett och samma
register

Samtliga namn (utom specialfallen i parenteser) avser de lägsta bitarna

Arkitektur x64

- De 16 generella 64-bit registerna kan "styckas upp i"
 - 32-bitars register
 - 16-bitars register
 - 8-bitars register

64-bits register

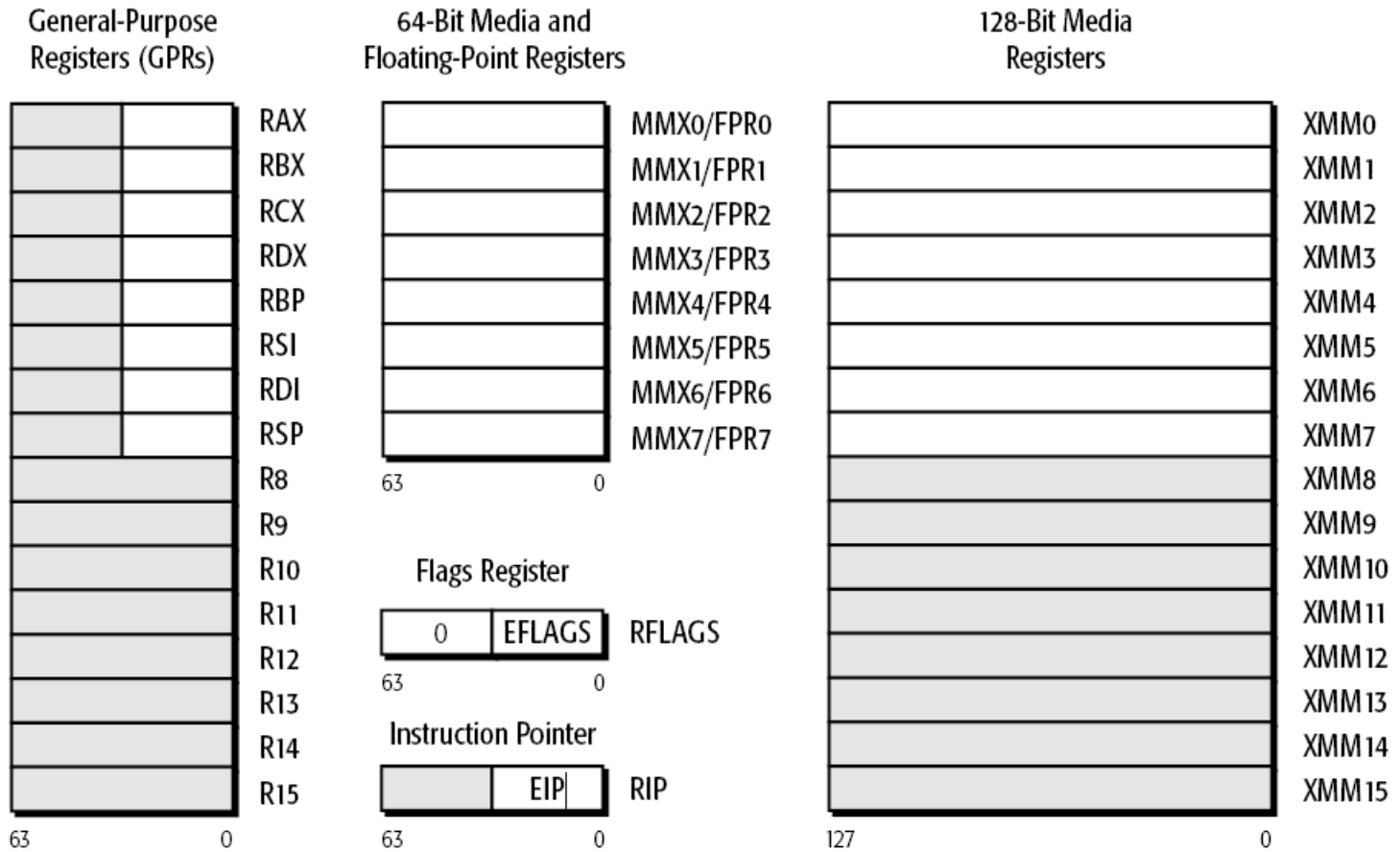



Fler register


- Det finns fler register utöver dessa 16 generella heltalsregister
- FPR – Ursprungliga flyttalsregister
- MMX – 64-bitars ”media”-register. Mappade ovanpå FPR.
- XMM – 128-bitars flyttal. *Streaming SIMD Extension (SSE)*
- RIP - Programräknaren

FPR=Floating point register för x87 flyttalprocessor (ursprunglig användning)
 MMX=Integer 64-bit (eller 2 st packed 32-bit, 4 st packed 16-bit, 8 st packed 8-bit) (mappat ovanpå FPR)
 Problem: MMX/FPR upptagen av x87 => kunde inte streama (SIMD). Man fick då skapa XMM-register

Fler register



 Legacy x86 registers, supported in all modes

 Register extensions, supported in 64-bit mode

Application-programming registers also include the 128-bit media control-and-status register and the x87 tag-word, control-word, and status-word registers

Ännu fler utökningar och register

- Utökningar (Extensions) med nya register och nya instruktioner
 - Instruktioner på vektorer av flyttal (SIMD)
- Advanced Vector Extension (AVX)
 - **YMM – 256-bitar.**
- AVX-512
 - **ZMM – 512 bitar.**

■ ... (MMX-) integer registers
 ■ ... floating-point registers (double-prec.)
 ■ ... floating-point registers (single-prec.)
■ ... only available in 64-bit (long) mode
 ■ ... not accessible at the same time as AH, BH, CH, DH (require REX prefix)
 (Mostly) managed by the OS:
■ ... segment, status- and control registers
■ ... system registers

Arkitektur x64

- VARNING: Olika versioner av assembler-översättare:
 - AT&T-syntax (Unix/Linux-världen)
(gnu-C-kompilator gcc och gnu-assembler gas)
 - Intel-syntax (Windows-världen)
- Ordningen mellan operanderna är omkastad!
- I dokumentet på Canvas ("Intel 64-bits arkitektur – Introduktion till assembler-programmering för x86_64") används AT&T (samt på labbar, dvs omkastat mot ARM).

Suffix till instruktioner

- Suffix till instruktioner anger hur stort dataformat som ska användas
 - b = byte (8 bitar)
 - s = *short (16-bits heltal) eller single (32-bits flyttal)*
 - w = word (16 bitar, OBS!!)
 - l = long (32-bits heltal eller 64-bits flyttal)
 - q = quad (64 bitar)
 - t = *ten bytes (80 bits flyttal)*
- Om man utelämnar suffix används formatet hos destinationsregistret (osäker programmering, **rekommenderas inte**)

Add^q = 64-bit addition
Add^l = 32-bit addition
Add^w = 16-bit addition
Add^b = 8-bit addition

Några exempel

q = quad = 64 bitar

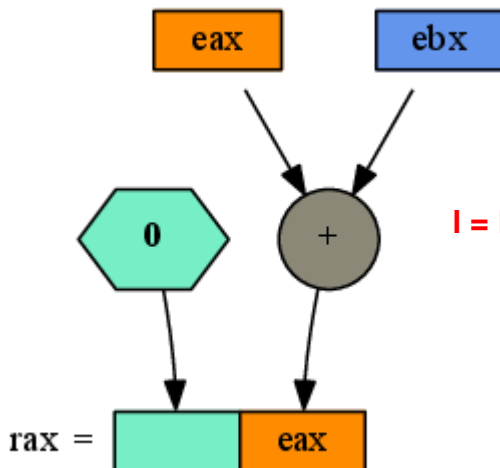
\$ anger
immediate
data

addq ^{64 bitar} %rbx, ^{64 bitar} %rax
 addq \$-1, %rax
 decq %rax

% anger register

OBS! ordning

rax <- rax+rbx
 # rax <- rax-1
 # rax <- rax-1



- Instruktioner som skriver över lägre halvan av ett register lägger nollor i övre halvan:

I = long = 32 bitar 32 bitar 32 bitar
 addl ^{32 bitar} %ebx, ^{32 bitar} %eax
 add ^{64 bitar} %rbx, ^{32 bitar} %eax

#eax <- eax+ebx

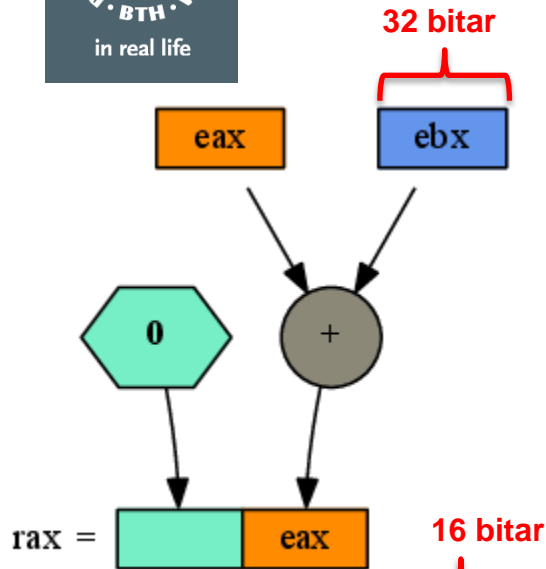
#fungerar inte!

(Formatet måste vara lika stort!)

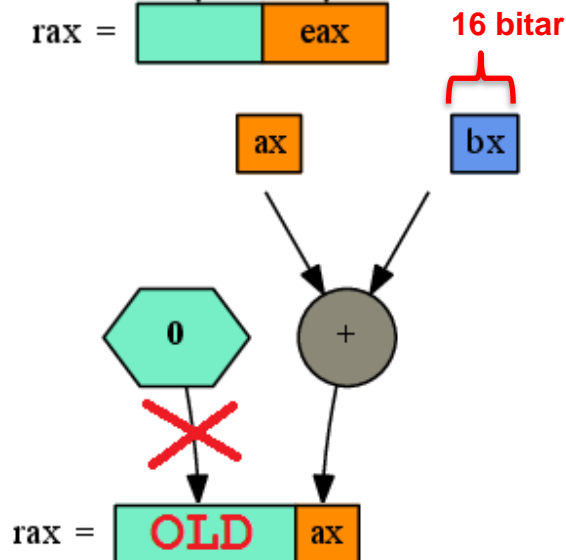
Addq = 64-bit addition
 Addl = 32-bit addition
 Addw = 16-bit addition
 Addb = 8-bit addition

Inget suffix => eax
 (destinationsregister)
 bestämmer (32 bitar)

OBS!!!



- Att den högre delen av registret fylls med nollor gäller ***bara instruktioner som skriver 32 bitar.***



- Om man skriver 16 eller 8 bitar i ett register kommer resten att vara oförändrat. (Nollställ själv!)

Notera att rax, eax, ax, ah och al
är samma register

Indirekt adressering till minne

Parentes anger att
registrets innehåll
tolkas som adress

l = long = 32 bitar
movl (**%rbx**), **%eax**

64 bitar
32 bitar

#laddar ett 32-bitstal från
#minnesadressen **rbx** pekar på till **eax**
(**%rbx**) = 64-bitars adress som pekar på ett 32-bitars tal

q = quad = 64 bitar
movq **%rdi**, (**%r12**)

64 bitar
64 bitar

#sparar 64 bitar från **rdi** till
#den minnesplats **r12** pekar på

l = long = 32 bitar
movl **%eax**, **-4(%rbp)**

32 bitar
64 bitar

#sparar 32 bitar från **eax** till
#adressen **rbp-4**

Minnesadressen i register
rbp ökas med (-4)

Add**q** = 64-bit addition
Add**l** = 32-bit addition
Add**w** = 16-bit addition
Add**b** = 8-bit addition

Alternativ adressering till minne

Istället för (**%rbx**) man använda:

- Deklarera en variabel (t.ex. **SUM:**) i programmets datasektion. Då kan **SUM** användas direkt i instruktionen.

Anm: Med **lea** (*load effective address*) flyttar man en adress till ett register.

Exempel **leaq SUM, %r11**

q = quad = 64 bitar

- Ange ett tal direkt i instruktionen med hjälp av \$
Detta kallas *immediate adressering*.

Exempel: \$4 (Notera att det tolkas som 32 bitar.)

Hantering av stacken

64 bitar



- Registret **rsp** används normalt som stackpekare
- OBS! Stacken växer mot lägre adresser
- Instruktionerna **push** och **pop** sparar respektive hämtar data på stacken och uppdaterar stackpekaren automatiskt
- instruktionen **call** (som används för hopp till subrutin) "pushar" automatiskt återhoppsadressen på stacken
- instruktionen **ret** används vid återhopp från subrutin och "poppar" automatiskt återhoppsadressen från stacken till programräknaren

Instruktioner för att flytta data

Instruktion	Resultat	Beskrivning
mov _q S, D	$D \leftarrow S$	flytta 64-bits ord
movabs _q I, R	$R \leftarrow I$	flytta 64-bits ord imm.
movs _{lq} S, R	$R \leftarrow \text{SignExtend}(S)$	flytta 32-bits ord med teckentillägg (till 64-bit dest.)
movsb _q S, R	$R \leftarrow \text{SignExtend}(S)$	flytta 8-bits ord med teckentillägg (till 64-bit dest.)
movzb _q S, R	$R \leftarrow \text{ZeroExtend}(S)$	flytta 8-bits ord utfyllt med nollor (till 64-bit dest.)
push _q S	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$	lägg S överst på stacken
pop _q D	$D \leftarrow M[R[\%rsp]];$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	hämta till D från överst på stacken

%rsp =
stackpekare

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

Suffix: q = 64, l = 32, w = 16 och b = 8 bitar, t.ex. movq, movl, movw och movb

Komihåg: Endast movl fyller de översta 32-bitarna med nollor (ej movw eller movb).

Data kan flyttas till/från minne samt mellan register (normalt ej minne-minne)

Instruktioner för aritmetik och logik

q = 64-bit
l = 32-bit
w = 16-bit
b = 8-bit

OBS! Helt annan funktion vid endast en operand, se två sidor fram

.data
SUM:

Se exempel ovan

(Heltal)

se
nästa
sida

Instruktion	Resultat	Beskrivning
leaq S, D	$D \leftarrow \&S$ Ex: leaq SUM,%r11	ladda effektiv adress
incq D	$D \leftarrow D + 1$	inkrement (räkna upp med 1)
decq D	$D \leftarrow D - 1$	dekrement (räkna ned med 1)
negq D	$D \leftarrow -D$	negation (teckenväxling)
notq D	$D \leftarrow \sim D$	invertera alla bitar
addq S, D	$D \leftarrow D + S$	addition
subq S, D	$D \leftarrow D - S$	subtraktion
imulq S, D	$D \leftarrow D * S$	multiplikation
xorq S, D	$D \leftarrow D \wedge S$	bitvis xor
orq S, D	$D \leftarrow D \vee S$	bitvis eller
andq S, D	$D \leftarrow D \& S$	bitvis och
salq k, D	$D \leftarrow D \ll k$	bitvis vänsterskift k positioner
shlq k, D	$D \leftarrow D \ll k$	samma som ovan
sarq k, D	$D \leftarrow D \gg k$	aritmetiskt högerskift k positioner
shrq k, D	$D \leftarrow D \gg k$	logiskt högerskift k positioner

S = källa (source), D = destination

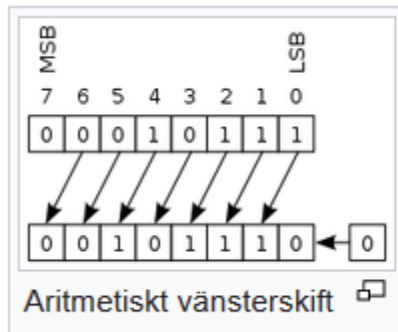
Operanderna kan ligga i minnet eller i register, dock kan inte båda operanderna ligga i minnet.

Instruktioner för aritmetik och logik

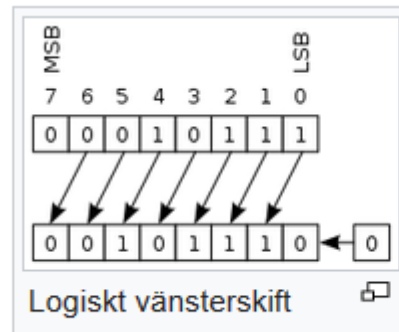
(forts.)

Instruktion	Resultat	Beskrivning
$\text{sal}_q\ k, D$	$D \leftarrow D \ll k$	bitvis vänsterskift k positioner
$\text{shl}_q\ k, D$	$D \leftarrow D \ll k$	samma som ovan
$\text{sar}_q\ k, D$	$D \leftarrow D \gg k$	aritmetiskt högerskift k positioner
$\text{shr}_q\ k, D$	$D \leftarrow D \gg k$	logiskt högerskift k positioner

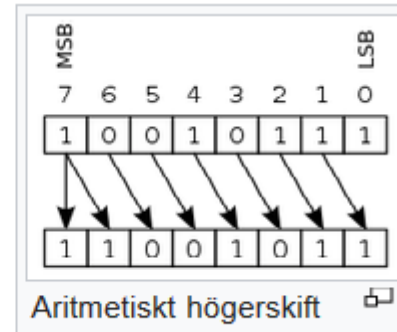
sal_q



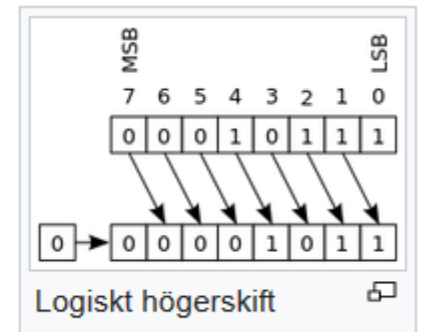
shl_q



sar_q



shr_q



Speciella aritmetiska instruktioner

OBS! Helt annan funktion vid två operander, se två sidor bak

OBS Heltal!

Instruktion	Resultat	Beskrivning
imul _q S	$R[\%rdx]: R[\%rax] \leftarrow S \times R[\%rax]$	Full multiplikation med teckensatta tal
mul _q S	$R[\%rdx]: R[\%rax] \leftarrow S \times R[\%rax]$	Full multiplikation med teckenlösa tal
clt _q	$R[\%rax] \leftarrow \text{SignExtend}(R[\%eax])$	Konvertera %eax till 64 bitar
cqto	$R[\%rdx]: R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	Konvertera %rax till 128 bitar
idiv _q S	$R[\%rdx] \leftarrow R[\%rdx]: R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]: R[\%rax] / S$	Division med teckensatta tal
div _q S	$R[\%rdx] \leftarrow R[\%rdx]: R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]: R[\%rax] / S$	Division med teckenlösa tal

o = 128-bit
(octoword)

$$\begin{array}{c}
 \begin{array}{c} S \\ \boxed{} \\ 63 \quad 0 \end{array} \times \begin{array}{c} \%rax \\ \boxed{} \\ 63 \quad 0 \end{array} = \begin{array}{c} \%rdx \\ \boxed{} \\ 127 \end{array} : \begin{array}{c} \%rax \\ \boxed{} \\ 64 \quad 63 \quad 0 \end{array}
 \end{array}
 \quad
 \begin{array}{c}
 \begin{array}{c} \%rdx \\ \boxed{} \\ 127 \end{array} : \begin{array}{c} \%rax \\ \boxed{} \\ 64 \quad 63 \quad 0 \end{array} / \begin{array}{c} S \\ \boxed{} \\ 63 \quad 0 \end{array} = \begin{array}{c} \%rax \\ \boxed{} \\ 63 \quad 0 \end{array} \text{ kvot} \quad \text{och} \quad \begin{array}{c} \%rdx \\ \boxed{} \\ 63 \quad 0 \end{array} \text{ rest}
 \end{array}$$

(Produkt av två st 64-bit tal kan bli 128-bit lång)

$$\begin{array}{r}
 1111 \\
 \times 1111 \\
 \hline
 1111 \\
 1111 \\
 1111 \\
 + 1111 \\
 \hline
 1110001
 \end{array}$$

Hoppinstruktioner (urval)

Tester att basera villkorliga hopp på

Instruktion	Jämförelse baserad på	Beskrivning
<code>cmp_q S₂, S₁</code>	$S_1 - S_2$	Jämför 64-bits dataord som teckensatta tal. OBS! Ordningsföljden!
<code>test_q S₂, S₁</code>	$S_1 \& S_2$	Testar 64-bits dataord

Ovillkorligt hopp

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

Villkorliga hopp

Instruktion	Beskrivning
<code>j_e label</code> (<i>e = equal</i>)	hoppa om föregående jämförelse är lika med noll
<code>j_{ne} label</code> (<i>ne = not equal</i>)	hoppa om föregående jämförelse inte är lika med noll
<code>j_g label</code> (<i>g = greater than</i>)	hoppa om resultat av jämförelse större än noll
<code>j_l label</code> (<i>l = less than</i>)	hoppa om resultat av jämförelse mindre än noll
<code>j_{ge} label</code> (<i>ge = greater or equal than</i>)	hoppa om resultat av jämförelse större än eller lika med noll
<code>j_{le} label</code> (<i>le = less or equal than</i>)	hoppa om resultat av jämförelse mindre än eller lika med noll

Programmering med subrutiner

Special-
instruktioner:

Instruktion	Beskrivning
call <i>function</i>	Subrutinanrop till <i>function (label)</i> , återhopsadress läggs automatiskt på stacken
ret	Retur från subrutin, återhopsadress hämtas automatiskt från stacken till pc

- **Stack alignment**

- Vid anrop av funktion måste stacken vara **16 bytes aligned**
- Det innebär att stackpekarens värde måste vara en adress jämnt delbar med 16 (annars blir det **segmentation fault**)
- Speciellt viktigt vid anrop till externa funktioner/bibliotek, t.ex. printf() från C standardbibliotek (stdio.h)
- Ett funktionsanrop lägger återhopsadressen på stacken (8 bytes) => stacken direkt unaligned
 - Pushq \$0 kan användas för att få stacken 16 bytes aligned igen

Anropskonventioner vid subrutin

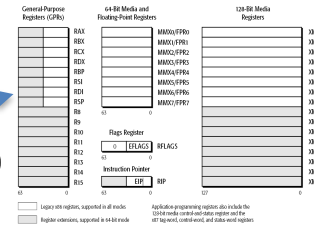
Gäller AT&T (inte Windows-miljö)

- Heltalsparametrar skickas in (med `call`) i `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
 - första parametern i `%rdi`
 - andra parametern i `%rsi`
 - ...
 - Vid fler än 6 parametrar skickas resten via stacken (s)

- Flyttalsparametrar skickas in via `%xmm`, se  (`%rax` innehåller antal `%xmm` som används)

- Returvärde skickas i `%rax` som heltal (eller pekare)
Vid flyttal: `%xmm0`

- **%rbp, %rbx** och **%r12 - %r14** måste sparas undan och återställas av en subrutin om deras värden förändras i rutinen



OUT: rax

PUSH/POP: **rbx**

IN4: rCX

IN3: rdx

IN2 : rsi

IN1: rdi

PUSH/POP: **r**bp****

(Stackpekare): **rsp**

IN5: r8

IN6: r9

```
printf "förstör" %rdi
```

r10

r11

PUSH/POP: r12

PUSH/POP: r13

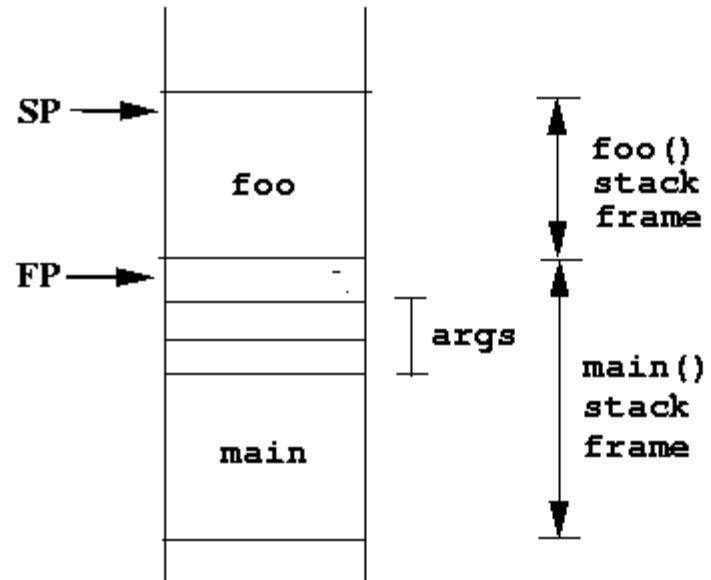
PUSH/POP: r14

r15

Att använda en "frame-pointer"

Om en funktion använder stacken kan det vara bekvämt att kopiera stackpekaren vid rutinens start till en "frame-pekare".

När man lämnar rutinen är det bara att återställa stackpekarens värde till denna.



Exempelprogram

Datadefinition

```
.data  
str:    .asciz "Fak=%d\n"  
buf:    .asciz "xxxxxxxxxx"  
endTxt: .asciz "slut\n"
```

Exempelprogram

```
.data
str:      .asciz "Fak=%d\n"
buf:      .asciz "xxxxxxxx"
endTxt:   .asciz "slut\n"
```

```
.text
.global main

main:
    pushq $0                # Stacken ska vara 16 bytes "aligned"
    movq  $5, %rdi          # Beräkna 5!
    call  fac               # IN=%rdi, OUT=%rax (dvs %rax=120)
    movq  %rax, %rsi        # Flytta returvärdet till argumentregistret %rsi
    movq  $str, %rdi        # skriv ut Fak= "resultat"
    call  printf            # %rdi pekar på formatsträng "Fak=%d\n", %rsi=värde
# läs med fgets(buf,5,stdin) (C-anropsformat) fgets(%rdi,%rsi,%rdx)
    movq  $buf, %rdi        # lägg i buf
    movq  $5,%rsi           # högst 5-1=4 tecken
    movq  stdin, %rdx       # från standard input
    call  fgets
    movq  $buf, %rdi        # fgets(%rdi,%rsi,%rdx) -> $buf -> %rdi
    call  printf            # skriv ut buffert
    movq  $endTxt,%rdi      # följd av slut
    call  printf
    popq  %rax              # avsluta programmet
    ret
```

```
OUT: rax
PUSH/POP: rbx
IN4: rcx
IN3: rdx
IN2: rsi
IN1: rdi
PUSH/POP: rbp
(Stackpekare): rsp
IN5: r8
IN6: r9
r10
r11
PUSH/POP: r12
PUSH/POP: r13
PUSH/POP: r14
r15
```

Exempelprogram forts

Här finns funktionen n! (rekursiv) **Exempel 3! = 3*2*1 = 6**

Inparameter: %rdi

Utparameter: %rax

fac:

```

    cmpq    $1,%rdi    # if n>1
    jle     lBase      # hoppa till lBase om klar ( *1 )
    pushq   %rdi        # lägg anropsvärde på stacken
    decq    %rdi        # räkna ned värdet med 1
    call    fac         # temp = fakultet av (n-1)
    popq    %rdi        # hämta från stack
    imul    %rdi,%rax    # return n*temp (%rax=%rax*%rdi)
    ret     # Återvänd

```

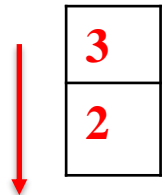
lBase:

```

    movq    $1,%rax    # else return 1
    ret     # Återvänd

```

Stack:



OUT: rax
 PUSH/POP: rbx
 IN4: rcx
 IN3: rdx
 IN2: rsi
 IN1: rdi
 PUSH/POP: rbp
 (Stackpekare): rsp
 IN5: r8
 IN6: r9
r10
r11
 PUSH/POP: r12
 PUSH/POP: r13
 PUSH/POP: r14
r15

Adresseringsmoder

Omedelbar adressering (Immediate):

Exempel:

```
addl $14,%eax      # eax <- eax+14
```

(Addering mellan ett tal och ett register.)

OBS: Talet efter \$ kan max vara 32 bitar. (jämför med 12 bitar av de 32 bitarna för ARM)

(**addl** = 32-bit)

(**%eax** = 32-bit)

Addq = 64-bit addition

Addl = 32-bit addition

Addw = 16-bit addition

Addb = 8-bit addition

Adresseringsmoder

Registerdirekt adressering:

Exempel:

addb %a1,%r8b # r8b <- r8b + a1

(Addering mellan två register.)

(**addb** = 8-bit)

(%**a1** = 8-bit)

(%**r8b** = 8-bit)

Del av registertabellen:

64-bitsregister	32-bitsregister	16-bitsregister	8-bitsregister
<u>rax</u>	<u>eax</u>	<u>ax</u>	<u>al</u> (ah, hög byte i ax)
r8	r8d	r8w	<u>r8b</u>

Addq = 64-bit addition

Addl = 32-bit addition

Addw = 16-bit addition

Addb = 8-bit addition

Adresseringsmoder

Absolut adressering:

Exempel:

.data

symbol: .long 20 # Talet 20 som 32-bit int

addl symbol,%eax # eax <- eax + 20

(Addering mellan variabel i minne och register.)

(**addl** = 32-bit)

(**symbol** = talet 20 lagrat som 32-bit integer)

(**%eax** = 32-bit)

Addq = 64-bit addition

Addl = 32-bit addition

Addw = 16-bit addition

Addb = 8-bit addition

Adresseringsmoder

Registerindirekt adressering med offset:

Exempel:

`.data`

```
string: .asciz "Hello world!"    # string <- "Hello world!"
```

```
leaq string,%rcx    #nu pekar rcx på "H" i "Hello world!"
```

```
movb 4(%rcx),%r8b    #hämtar innehållet från den adress rcx  
                    #anger plus offset 4 (dvs %r8b ← tecknet 'o')
```

(**leaq** = 64-bit)

(**string** = 12 st ASCII-tecken)

(**%rcx** = 64-bit)

(**%r8b** = 8-bit = 1 st ASCII-tecken)

lea (*load effective address*) flyttar adress till ett register.

Addq = 64-bit addition

Addl = 32-bit addition

Addw = 16-bit addition

Addb = 8-bit addition

Adresseringsmoder

Bas+index-adressering:

Exempel:

`.data`

```
string: .asciz "Hello world!" # string <- "Hello world!"
```

```
leaq string,%rcx #nu pekar rcx på "H" i "Hello world!"
```

```
movq $6,%rax #rax <- 6
```

```
movzbq 4(%rcx,%rax,1), %r8
```

#hämtar innehållet från adress (string+6*1+4)

#dvs %r8 ← tecknet 'd')

movzbq S, R (finns bara som 64-bit) gör $R \leftarrow \text{ZeroExtend}(S)$, dvs fyller ut med nollor i de högsta 64-8=56 bitarna till vänster om ASCII-tecknet 'o' innan det hamnar i r8, 64-bit

Format: offset (bas, index, antalbyte) och den effektiva adressen blir $\text{bas} + \text{index} * \text{antalbyte} + \text{offset}$

(**leaq** = 64-bit)

(**movq** = 64-bit)

(**string** = 12 st ASCII-tecken)

(%**rcx** = 64-bit)

(%**rax** = 64-bit)

(%**r8** = 64-bit)

Addq = 64-bit addition

Addl = 32-bit addition

Addw = 16-bit addition

Addb = 8-bit addition

Vanliga assemblerdirektiv

- .**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

Vanliga assemblerdirektiv, forts

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

Exempelprogram "Hello World!"

("Hello world!" skrivs ut i terminalfönstret.)

```
.text                # deklaration av text-sektion (kodavsnitt)
.global main         # startpunkt som länkaren känner igen

main:
    pushq $0          # för stack alignment 16 bytes

    { movq $message, %rdi
      call printf      # %rdi pekar på "H" i "Hello world!\n"

    call exit

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

- Vid anrop av funktion måste stacken vara 16 bytes *aligned*
- Stackpekarens värde måste vara en adress jämnt delbar med 16

Exempelprogram: Tvåpotenser

Skriv ut alla tvåpotenser mellan 2^0 och 2^{31} .

Inledning:

```
.text
.global main

main:
    pushq $0          # för stack alignment 16 bytes
    movq $1, %rsi     # aktuellt värde %rsi = 1
    movq $32, %rdi    # räknare %rdi = 32
```

Nu gäller:

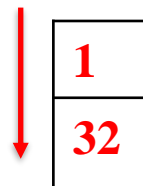
aktuell tvåpotens: $\%rsi = 1, 2, 4, 8, 16, \dots, 2^{31}$
räknare: $\%rdi = 32 \rightarrow 31 \rightarrow 30 \rightarrow \dots \rightarrow 0$

(forts.)

Exempelprogram: Tvåpotenser, forts

aktuell tvåpotens: %rsi = 1,2,4,8,16,...,2³¹
räknare: %rdi = 32->31->30->...->0

Stack:



lLoop:

```

pushq %rsi          # lägg registervärde på stacken (printf förstör)
pushq %rdi          # lägg registervärde på stacken (printf förstör)
movq $format, %rdi  # formatsträngens adress
                    # andra argumentet (värdet) ligger redan i %rsi
xorq %rax, %rax      # nollställ %rax före printf (ty X EXOR X = 0)
call printf         # %rdi pekar på formatsträng "%ld\n", %rsi=tvåpotens
                    # OBS! printf "förstör" %rdi

popq %rdi           # återhämta registervärde (OBS! ordningen)
popq %rsi           # återhämta registervärde
addq %rsi, %rsi      # dubblera tvåpotensen 1+1=2, 2+2=4, etc
decq %rdi           # räkna ned räknare 32->31->30->...->0 (dec)
jne lLoop          # Hoppa till lLoop om det inte blev noll
call exit

```

.data

format: .asciz "%ld\n"

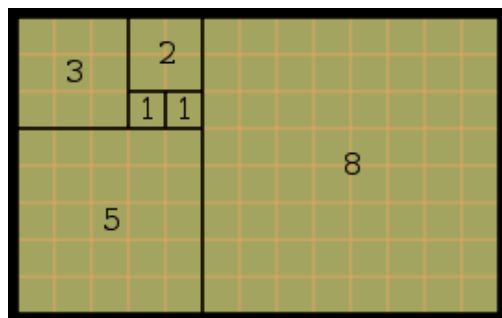
OUT: [rax](#)
PUSH/POP: [rbx](#)
IN4: [rcx](#)
IN3: [rdx](#)
IN2: [rsi](#)
IN1: [rdi](#)
PUSH/POP: [rbp](#)
(Stackpekare): [rsp](#)
IN5: [r8](#)
IN6: [r9](#)
[r10](#)
[r11](#)
PUSH/POP: [r12](#)
PUSH/POP: [r13](#)
PUSH/POP: [r14](#)
[r15](#)

Exempelprogram: Fibonacci-tal

Skriv ut de 40 första Fibonacci-talen.

varje tal är summan av de två föregående Fibonaccitalen
(start: 0, 1)

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393,
196418, 317811, 514229, 832040, 1346269, 2178309, 3524578,
5702887, 9227465, 14930352, 24157817, 39088169, 63245986



Varje kvadrat har Fibonaccital
som sidlängd.

Tillämpning:
Tillväxt hos kaniner



Exempelprogram: Fibonacci-tal

Skriv ut de 40 första Fibonacci-talen.

Inledning:

```
.text
.global main
main:
    pushq $0                # för stack alignment 16 bytes
    movq $40, %rcx          # initiera %rcx till 40 (räknare)
    xorq %rax, %rax         # nollställ %rax (aktuellt tal)
    movq $1, %rbx           # initiera %rbx (nästa tal)
```

Nu gäller: räknare: %rcx = 40->39->38->...->0
 aktuellt tal: %rax = 0
 nästa tal: %rbx = 1

Exempelprogram: Fibonacci-tal

Nu gäller: räknare: $\%rcx = 40 \rightarrow 39 \rightarrow 38 \rightarrow \dots \rightarrow 0$
 aktuellt tal: $\%rax = 0$
 nästa tal: $\%rbx = 1$

Stack:

0
40

lPrint:

```

pushq %rax           # lägg registervärde på stacken (printf förstör)
pushq %rcx           # lägg registervärde på stacken (printf förstör)
movq $format, %rdi   # formatsträngens adress
movq %rax, %rsi      # andra argumentet till %rsi (aktuellt tal)
xorq %rax, %rax      # nollställ %rax före printf (ty X EXOR X = 0)
call printf          # %rdi pekar på formatsträng "%ld\n", %rsi=tal
popq %rcx            # återhämta registervärde (OBS! ordningen)
popq %rax            # återhämta registervärde
movq %rax, %rdx      # spara undan aktuellt tal i %rdx
movq %rbx, %rax      # skifta så nästa tal blir aktuellt
                     # %rax: 0->1, 1->1, 1->2, 2->3, 3->5,...)
addq %rdx, %rbx      # beräkna nästa tal
decq %rcx            # räkna ned räknaren (40->39->38->...->0)
jne lPrint           # om det inte blev noll beräkna ett nytt tal
call exit
    
```

.data

format: .asciz "%d\n"

Exempelprogram: Max(IN1,IN2,IN3)

Jämför parameterordning till höger:

Här skall det vara 32 bitar, dvs

IN1 = %edi (jfr %rdi)

IN2 = %esi (jfr %rsi)

IN3 = %edx (jfr %rdx)

Men returvärde i %rax som standard → här %eax

Dvs MaxOfThree(%edi,%esi,%edx) → %eax

```
.text
.global MaxOfThree
MaxOfThree:
    cmp1 %esi, %edi #jämför argument 1 och 2
    cmov1 %esi, %edi #flytta %esi-värdet till %edi om större
    cmp1 %edx, %edi #jämför med argument 3
    cmov1 %edx, %edi #flytta %edx-värdet till %edi om större
    mov1 %edi, %eax #lägg returvärdet i %eax
ret
```

Ny instruktion **cmov¹** – *Conditional move if less.*

OUT:	<u>rax</u>
PUSH/POP:	<u>rbx</u>
IN4:	<u>rcx</u>
IN3:	<u>rdx</u>
IN2:	<u>rsi</u>
IN1:	<u>rdi</u>
PUSH/POP:	<u>rbp</u>
(Stackpekare):	<u>rsp</u>
IN5:	r8
IN6:	r9
	r10
	r11
PUSH/POP:	r12
PUSH/POP:	r13
PUSH/POP:	r14
	r15

Exempelprogram: Max(IN1,IN2,IN3)

Ovanstående assemblerfunktion `Max(%edi,%esi,%edx) -> %eax` skall länkas med C-programmet nedan för körbar fil

```
#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;
}
```