dComArk: Aflevering 5
Programmering i x86-64

Christoffer Müller Madsen, 201506991
Jacob Hartmann, 20094613
Casper Vestergaard Kristensen, 201509411
Datalogi

13. december 2015

Spørgsmål A

I opgavebeskrivelsen nævnes det at Fibonacci-talrækken for $n \geq 0$ kan beskrives rekursivt ved følgende:

$$\begin{aligned} \texttt{fib}(0) &= 0 \\ \texttt{fib}(1) &= 1 \\ \texttt{fib}(n) &= \texttt{fib}(n-1) + \texttt{fib}(n-2) \end{aligned}$$

Der er skrevet et program `fib.c` i sproget C, der tager argumentet n fra kommandolinjen og bergner det n'te fibonacci-tal ved brug af en rekursiv funktion `fib(n)`, der står i overensstemmelse med den ovenstående definition. Resultatet af denne udregning printes derefter ved brug af `printf`. Kildekoden til programmet er vist med kommentarer i Listing 1.

Listing 1: `fib.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  unsigned long long fib(unsigned long long n) {
5      unsigned long long r;           // Declare a local variable r
6      if (n == 0) {                   // If n is zero, return 0.
7          r = 0;
8      } else if (n == 1) {            // If n is one, return 1.
9          r = 1;
10     } else {                        // Else, calculate the number
11         r = fib(n-1) + fib(n-2);    //  to return by utilising
12     }                               //  recursion.
13     return r;
14 }
15
16 int main(int argc, char *argv[]) {
17     unsigned long long n, r;        // Define local variables
18     n = atoll(argv[1]);             // Convert the first argument
19                                     //  to a long long value and
20                                     //  load it into n.
21     r = fib(n);                     // Set the variable r to the
22                                     //  n'th Fibonacci number.
23                                     // Print the number found.
24     printf("fib(%lld) = %lld\n", n, r);
25     return 0;                       // Return 0 to indicate a
26                                     //  successful execution.
27 }
```

Først fortælles, at de to biblioteker `stdio.h` og `stdlib.h` skal inkluderes. Dette gøres for at have adgang til funktioner som `atoll` og `printf`, der skal bruges til henholdsvist at lave en `long long` værdi[1] ud fra argumentet til programmet og til at printe det endelige output. Herefter defineres funktionen fib ud fra den definition, der er givet i opgave formuleringen. I `main`-funktionen deklareres de to lokale variable `n` og `r` som `unsigned long long` værdier. `n` sættes til at være det første argument til programmet ved brug af `atoll`-funktionen og `r` sættes til det n'te fibonacci-tal ved brug af funktionen `fib(n)`. Dernæst printes strengen `fib(n)=r`, hvor `n` og `r` erstattes af deres respektive værdier. Til sidst returneres værdien 0 for at vise at programmet eksekverede succesfuldt. Der er udført en række tests på programmet for at sikre at det opfører sig efter hensigten. Resultaterne af de kørte tests ses i Tabel 1.

[1] I Linux er `long long` under x86-64 implementeret som en 64-bit værdi.

n	`fib(n)`	Forventet?
0	0	✓
1	1	✓
2	1	✓
3	2	✓
4	3	✓
5	5	✓
6	8	✓
10	55	✓
20	6765	✓
21	10946	✓
22	17711	✓

Tabel 1: Tests udført på `fib.c` . Returværdierne for funktionen er korrekte i forhold til de forventede tal.

Spørgsmål B

Til denne opgave er Fibonacci-funktionen implementeret i en funktion skrevet i x86-64 symbolsk maskinsprog. Koden til funktionen `fib` er placeret i filen med navnet `fib.S` og er vist i Listing 2 i bilagenes afsnit A. Der er skrevet et program i C til at kalde funktionen og printe dennes returværdi. Dette gøres i et C-program da det er nemmere at foretage disse handlinger i C fremfor symbolsk maskinkode. Koden til dette program findes i Listing 3 i bilagenes afsnit B samt i filen `fib-ext.c` der er vedlagt denne fil. Programmet er nærmere beskrevet i kommentarerne i filen `fib.S` samt i besvarelsen af Spørgsmål C, hvor proceduren angående de rekursive funktionskald, der udføres i funktionen, forklares nærmere.

Sammenligning med programmet skrevet i C Det nyligt skrevne program i symbolsk maskinkode testes ved at undersøges hvorvidt det giver samme resultater som programmet skrevet i C for det tidligere viste udvalg af værdier for n. Resultaterne af disse tests ses i Tabel 2.

n	`fib.c`	`fib-ext.c`	Overensstemmelse
0	0	0	✓
1	1	1	✓
2	1	1	✓
3	2	2	✓
4	3	3	✓
5	5	5	✓
6	8	8	✓
10	55	55	✓
20	6765	6765	✓
21	10946	10946	✓
22	17711	17711	✓

Tabel 2: Tests udført på `fib.c` og `fib-ext.c`. Det ses at returværdierne af de to funktioner stemmer overens med hinanden. `fib-ext.c` benytter funktionen `fib` implementeret i x86-64 symbolsk maskinkode fremfor C.

Spørgsmål C

I funktionen `fib` i filen `fib.S` gøres følgende for at være i overensstemmelse med de givne kaldkonventioner:

- Udsatte registre gemmes på stakken for at bevare deres værdi på tværs af funktionskald
- Parametre fyldes i registret `rdi`
- Funktionen `fib` kaldes ved brug af `call`-instruktionen
- Stakafsnit etableres gennem manipulation af `rbp` og `rsp`
- Returværdien af funktionen lægges i `rax`
- Stakafsnit nedlægges ved brug af `leave` instruktionen
- Der returneres fra kaldet ved brug af `ret` instruktionen.
- De gemte registerværdier genoprettes

Disse vil blive gennemgået med eksempler i de følgende afsnit. Der tages udgangspunkt i hvorledes funktionen `fib` opfører sig lige inden og efter den kalder sig selv med nye parametre.

Sikring af caller-save registre I kodeudsnittet nedenfor ses det hvorledes værdien i `rdi`-registret sænkes med 1 inden den lægges på stakken. Grunden til at denne værdi lægges på stakken er for at beskytte den fra hvad end funktionen kan finde på at gøre ved værdien i registret. Eftersom `rdi` *ikke* er et callee-save register er det ifølge kaldkonventionerne nødvendigt som caller selv at sørge for, at værdien af dette register gemmes på stakken inden kaldet af funktionen.

```
1 decq %rdi             # Decrement n by 1.
2 push %rdi             # Save the value n - 1 in stack to
3                       #  be able to recover it later.
```

Kodeudsnittet nedenfor viser det sted i koden, hvor returværdien af det første kald af `fib`, der er gemt i `rax`, gemmes på stakken inden det andet funktionskald. Dette gøres da `rax`, ligesom `rdi`, ikke er et callee-save register.

```
push %rax               # Save the result of fib(n-1)
```

Parameterindlæsning Parametren til det første rekursive kald af funktionen `fib` inde i kroppen af funktionen er allerede givet ved blot at sænke værdien af `rdi` med 1, da det første kald der ønskes er `fib(n-1)`. Dette er hvad der ses gjort i den første listing ovenfor. Et andet eksempel på indlæsning af parametre ses før det andet kald af `fib` hvor den gamle værdi af `rdi` indlæses fra stakken og igen sænkes med 1 før `fib` kaldes.

```
1 pop %rdi              # Restore rdi = n-1
2 decq %rdi             # Decrement rdi by 1.
```

Funktionskald I følgende kodeudsnit kaldes funktionen `fib` for første gang med parametren `n-1`, der blev placeret i `rdi` i sidste kodeeksempel.

```
1 call fib              # Call the function.
2                       # rdi contains the value (n-1).
```

Kaldet af funktionen medfører, at værdien af `rip`-registret, der indeholder instruktionspointeren, lægges på stakken således at maskinen kender den adresse i hukommelsen som den skal vende tilbage til, efter kørslen af den kaldte funktion er overstået. Værdien, der lægges på stakken, kaldes også for returadressen grundet dens funktion. Værdien af returværdiregistret `rax` vil være lig returværdien af `fib(n-1)` når den kaldte funktion returnerer.

Etablering af stakafsnit I starten af `fib`-funktionens udførsel eksekveres følgende ordrer:

```
1 push %rbp             # Save old rbp to stack and align
2                       #  the stack pointer to 16-bytes.
3 movq %rsp, %rbp       # Write new base pointer.
```

Disse to ordrer sørger for at gemme den gamle base pointer og dernæst flytte base-pointeren til det øverste element på stakken. Dette gøres for at skabe et nyt stakafsnit og for at opretholde den 16-byte alignment som er foreskrevet i konventionerne i ABI. [Matz et al., 2015] Den gamle værdi af `rip` er allerede lagt på stakken i forbindelse med funktionskaldet og optager derved de første 8 bytes i det nye stakafsnit. Når den gamle værdi af `rbp`-registret lægges på stakken resulterer det i at det nye stakafsnit nu har en størrelse på 16 bytes, da begge registre, hvis værdier lægges på stakken, har en bredde på 8 bytes. Denne alignment er til for at gøre det muligt at benytte visse instruktioner fra SSE instruktionssættene, der indeholder ordrer til blandt andet vektorregning. Disse instruktioner benyttes imidlertid ikke i funktionen `fib`, men 16-byte tilpasningen udføres alligevel af god skik.

Placering af returværdier i registre Værdien som ønskes returneret fra funktionen skal lægges i `rax`.[2] Dette ses gjort i følgende uddrag fra `fib.S` hvor returværdierne af de to kald summeres i `rax`.

```
1 addq (%rsp), %rax     # Add fib(n-1) to rax which contains
2                       #  the reuslt of fib(n-2). fib(n-1)
3                       #  is retrieved from the stack.
```

Returværdien fra det andet funktionskald ligger allerede i `rax`, mens returvædien fra det første kald ligger øverst på stakken. Af denne grund er det muligt blot at bruge værdien, der ligger ved den adresse som stak pointeren `rsp` peger på, som det første argument. Herved summeres værdien af det øverste element på stakken og værdien af `rax` og placeres igen i `rax`.

Nedlæggelse af stakafsnit Stakafsnittet nedlægges meget simpelt ved brug af `leave` instruktionen. Hvis dette skulle gøres manuelt ville det bestå af først at genskabe den gamle værdi af `rsp` ved at flytte pointeren til der hvor base pointeren `rbp` på nuværende tidspunkt peger på. Dernæst flyttes `rbp` til den

[2] Hvis det ønskes at returnere to værdier, så fortæller konventionerne i ABI [Matz et al., 2015], at det er muligt at benytte registret `rdx` til at give én returværdi yderligere.

tidligere værdi af denne, som er gemt netop der hvor både `rbp` og `rsp` nu peger, hvorefter dette element på stakken fjernes. Herved er `rsp` og `rbp`—og derved stakrammen—bragt tilbage til den oprindelige tilstand hvilket medfører at alle lokale variable samt elementer i den nedlagte stakramme er væk. De beskrevne handlinger kan opsummeres med følgende kode:

```
1 movq %rbp, %rsp
2 pop %rbp
```

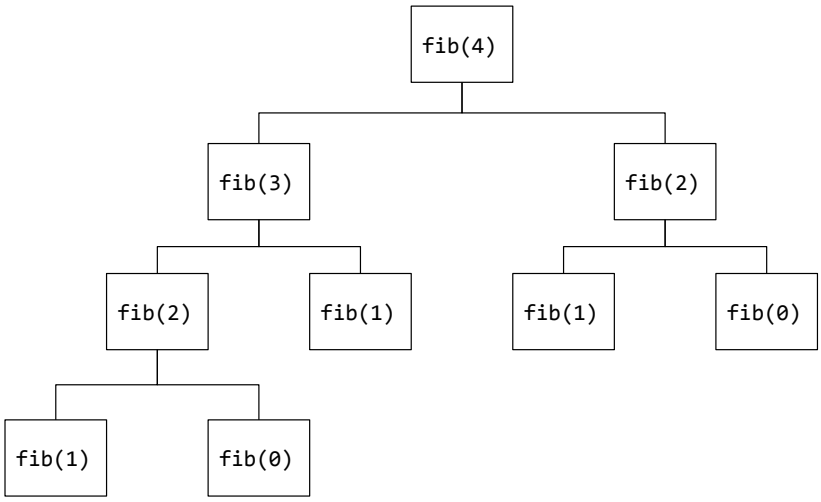
Returnering fra kaldet Returneringen fra kaldet klares også meget let ved brug af instruktionen `ret`. Denne instruktion sørger for at den næste instruktion der kaldes er den tidligere værdi af `rip`. Returadressen, der udgør den tidligere værdi af `rip`, ligger der hvor `rsp` nu peger på.

Genoprettelse af sikrede caller-save registre Efter kaldet er returneret genoprettes eventuelt gemte registre. I `fib.S` ses dette ved at den gemte værdi af `rdi` læses fra stakken ind i `rdi`. Denne handling ses i følgende instruktion:

```
pop %rdi                # Restore rdi = n-1
```

Ved at gemme værdien af registret `rdi` før funktionskaldet, var det altså muligt at bevare værdien af registret ved at gendanne denne efter funktionskaldet.

Eksempel med `fib(4)` For at vise forgreningen af rekursive kald simuleres et kald af funktionen `fib` fra filen `fib.s` gennem `fib-ext.c` med værdien af den første parameter sat til 4. På Figur 1 ses et diagram, der viser de kald som hvert kald af funktionen foretager sig. Denne figur antyder, at antallet stiger hurtigt med en forøgelse af funktionens parameter.



Figur 1: En repræsentation af hvilke kald, der udføres af de enkelte kald af funktionen `fib` for at færdiggøre det oprindelige kald af `fib(4)`.

For at vise de mange oprettelser og nedlæggelser af stakafsnit, der foretages i forbindelse med de mange kald, er der skabt en visualisering af stakken ved hver af de stak- og registerændrende ordre. Stakvisualiseringen kan ses i den vedlagte fil `Stakdiagram.pdf` samt i bilagets sidste del.

Da det kun er de stak- og registerændrende ordrer, der vises, er branch-instruktionerne i funktionen `fib` *ikke* afbildet, da disse ikke manipulerer med stakken. De individuelle stakrammers dybde i forhold til hinanden er vist ved brug af grå farvekodning. Endvidere benyttes der også farver til at vise typen af de forskellige ændringer, der foretages i forhold til stakken og registre. De farver og toner af grå, der benyttes, vises sammen med deres betydning i starten af afbildningen.

Spørgsmål D

Implementeringen af Fibonacci-tal-funktionen ved brug af iteration fremfor rekursion kan udføres i x86-64 symbolsk maskinkode på baggrund af C-programet `fib_iter.c`, der var inkluderet i opgavebeskrivelsen. Den nye implementering i symbolsk maskinkode findes i den vedlagte fil `fib_iter.S` samt i Listing 4, afsnit C i bilagene. Der defineres i denne fil en funktion `fib_iter(n)` til bestemmelse af det n'te Fibonacci-tal ved brug af iteration. Fortolkning af programargumenter, kald af funktionen samt print af resultatet foretages af C-programmet `fib_iter-ext.c` for at simplificere disse handlinger. Koden til dette program findes i den vedlagte fil `fib_iter-ext.c` samt i Listing 5, afsnit D i bilagene.

Koden og opbygningen i `fib_iter.S` er så vidt muligt oversat fra C-programmet `fib_iter.c` hvilket kommentarerne, der er skrevet til den symbolske maskinkode i filen `fib_iter.S` viser ved at forklare hvilke dele af C-programmet der er oversat til de enkelte ordrer i den symbolske maskinkode.

Spørgsmål E

Binær fil	Kildefil(er)
`fib`	`fib.c`
`fib-as`	`fib-ext.c` `fib.S`
`fib-pure-as`	`fib-pure-as.S`
`fib-java`	`Fib.java`
`fib_iter`	`fib_iter.c`
`fib_iter-as`	`fib_iter-ext.c` `fib_iter.S`
`fib_iter_java`	`FibIter.java`

Tabel 3: En oversigt over hvilke kildefiler, der er blevet brugt til at skabe de forskellige eksekverbare filer.

For at undersøge effektiviteten af de forskellige implementeringer af Fibonacci-funktionen benyttes programmet `time`, der måler de ressourcer et program bruger. Her er vi interesseret i `real`-ressourcen, der beskriver hvor lang tid programmet har kørt i realtid målt i sekunder. Dette vil svare til at tage tid på programmet ved brug af et stopur. Først undersøges det hvor hurtigt de forskellige implementeringer kan bestemme det 50. Fibonacci-tal. Resultatet af denne test er vist i Tabel 4. Bemærk at der også er lavet en implementering af både den rekursive og den iterative metode i Java. Dette er blot gjort for at skabe endnu et referencepunkt for effektiviteten af de forskellige implementeringer. De to Java-implementeringer vil derfor ikke blive drøftet i samme grad som implementeringerne i C og symbolsk maskinsprog.

Binær fil	Sprog	Kørselstid (s)
`fib`	C	120.82
`fib-as`	C og symbolsk maskinkode	109.18
`fib-pure-as`	Symbolsk maskinkode	110.60
`fib-java`	Java	70.91
`fib_iter`	C	0.00
`fib_iter-as`	C og symbolsk maskinkode	0.00
`fib_iter_java`	Java	0.19

Tabel 4: Tabel over kørselstiderne for de syv implementeringer af Fibonacci-funktionen kørt med argumentet 50. Tiderne er målt ved brug af programmet `time`.

Det er ret tydeligt at se ud fra disse resultater, at de tre iterative implementeringer er meget hurtigere end de rekursive implementeringer. Dette kan formodes at være på grund af de tre rekursive funktioners kraftige brug af ordrer, der manipulerer med stakken, da ordrer udført på stakken, er en del langsommere end tilsvarende ordrer udført på registrene. Som det blev vist i Spørgsmål C optræder disse ordrer eksempelvis når værdier fra registre før et funktionskald skal gemmes på stakken og efter funktionskaldet igen skal hentes fra stakken for at disse værdier ikke skal gå tabt. Den rekursive natur af de tre implementeringer øger antallet af funktionskald kraftigt, hvorved implementeringernes hastighed forringes grundet de mange stakmanipulationer der er nødvendige for at kalde og returnere fra funktioner.

Rekursive implementeringer De to rekursive implementeringer `fib-as` og `fib-pure-as` er nogenlunde lige effektive. Dog er programmet `fib`, der er skrevet udelukkende i C, ti sekunder langsommere end de to programmer `fib-as` og `fib-pure-as` hvor henholdsvis `fib`-funktionen og hele programmet er skrevet i x86-64 symbolsk maskinkode. Det kan tænkes at disse to implementeringer er hurtigere fordi de benytter færre stakordrer og i stedet benytter sig af maskinens registre. Java-implementeringen kører på næsten halvdelen af den tid af kørselstiden for de tre andre rekursive funktioner. Dette kan tænkes at være grundet optimeringen af Java når denne kompileres til bytekode og senere køres med JVM.

Iterative implementeringer For at opnå en mere sigende sammenligning af kørselstid for de tre iterativer implementeringer udføres denne gang en test udelukkende med de iterative implementeringer. Her udføres `fib`-funktionen med argumentet 5000000000. Resultaterne kan ses i Tabel 5.

Binær fil	Sprog	Kørselstid (s)
`fib_iter`	C	20.68
`fib_iter-as`	C og symbolsk maskinkode	2.87
`fib_iter_java`	Java	6.14

Tabel 5: Tabel over kørselstiderne for de tre iterative implementeringer af Fibonacci-funktionen kørt med argumentet 5000000000. Tiderne er igen målt ved brug af programmet `time`.

Her ses det, at der er en væsentlig forskel mellem de tre iterative implementeringer. `fib_iter`, der er skrevet i C er den langsommeste med en kørselstid, der er syv gange længere end kørselstiden for `fib_iter-as`, hvor funktionen `fib` er skrevet i symbolsk maskinsprog og kaldes fra et program skrevet i C. En stor kilde til denne hastighedsforskel kan igen tænkes at findes i brugen af registrene fremfor stakken. I den implementering af `fib`, der kaldes i `fib_iter-as`, benyttes stakken overhovedet ikke, mens det kan tænkes at den kompilerede udgave af C-koden i `fib_iter` benytter sig af stakken fremfor registrene. Compilerens implementering af C-koden kan dog undersøges nærmere ved at se på den maskinkode som compileren genererer ved brug af programmet `objdump`. Før dette kan gøres, er det dog nødvendigt at drøfte hvorledes der skabes en objektfil som `objdump` kan læse.

Til at kompilere og samle al kode i C og symbolsk maskinsprog har vi benyttet os af `gcc` (*GNU Compiler Collection*). Ved brug af kommandolinjeargumenterne `-c` og `-g` fås en ulinket objektfil, der indeholder information til *debugging*-brug—denne information bliver brugbar senere. Den genererede objektfil kan dernæst bruges til at finde ud af hvordan compileren har implementeret programmet i maskinsproget ved brug af `objdump`. For at gøre dette med `fib_iter.c` køres følgende kommandoer på en Linux-maskine med `gcc`, `objdump` og de nødvendige forudsætninger for disse to programmer installeret.

```
1 gcc -g -c fib_iter.c
2 objdump -d -M gas -S fib_iter.o
```

`objdump`-kommandoen printer oversættelsen af koden, kommenteret med den

tilsvarende kildekode fra `fib_iter.c`, til terminalen.[3] Kommentarerne skabes på baggrund af den debugging-information, der blev genereret sammen med maskinkoden af `gcc`. Ved brug af kommentarerne findes `for`-løkken fra `fib_iter.c`, hvor det ses at der fortrinsvist benyttes ordrer, der referer til stakken fremfor registrene. Dette indikeres ved brugen af base-pointerregistret `rbp` fremfor de almene registre. Ud fra dette kan det konkluderes at grunden til at `fib_iter` udføres langsommere end `fib_iter-as` er, at mens `fib_iter-as` kun benytter sig af registrene, bruger `fib_iter` stakken, hvilket medfører en kraftig forøgelse i programmets kørselstid.

Konklusion Gennem de udførte tests og beskrivelsen af funktionskald i Spørgsmål C er det vist, at de iterative implementeringer af `fib`-funktionen er meget hurtigere end de rekursive implementeringer på grund af de rekursive implementeringers mange funktionskald og dertilhørende stakoperationer, da disse er væsentligt langsommere end registeroperationerne som de iterative implementeringer gør brug af. Ved nærmere undersøgelse af maskinkoden som `gcc` genererede kunne det også vises at forskellen mellem kørselstiderne for `fib_iter` og `fib_iter-as` skyldes at `fib_iter-as` fortrinsvist bruger registrene til at gemme midlertidige værdier, mens `fib_iter-as` derimod bruger stakken til dette. Dette gør `fib_iter` meget langsommere end `fib_iter-as`.

Litteratur

[Matz et al., 2015] Matz, M., Hubička, J., Jaeger, A. og Mitchell, M. (2015). *System V Application Binary Interface (ABI) Draft Version 0.99.6.*

[3] Objektdumpet der bruges er vedlagt i filen `fib_iter.objdump.S`.

A fib.S

Listing 2: fib.S

```
1  .global fib
2  .section .text
3  fib:    push %rbp              # Save old rbp to stack and align
4                                 #  the stack pointer to 16-bytes.
5          movq %rsp, %rbp        # Write new base pointer.
6          cmpq $1, %rdi          # Compare the argument to 1.
7          je one                 # If equal to one go to 'one' and
8                                 #  return 1.
9          cmpq $0, %rdi          # Compare the argument to 0.
10         je zero                # If equal to zero go to 'zero'
11                                #  and return zero.
12
13         decq %rdi              # Decrement n by 1.
14         push %rdi              # Save the value n - 1 on the stack.
15         call fib               # Call the function.
16         pop  %rdi              # Restore rdi = n-1
17         decq %rdi              # Decrement rdi by 1.
18         push %rax              # Save the result of fib(n-1)
19         call fib               # Call the function.
20         addq (%rsp),%rax       # Add the top element (fib(n-1)) to
21                                #  rax which contains fib(n-2).
22         jmp end                # Jump to 'end' to leave and return
23                                #  from this call to the caller.
24
25 zero:   movq $0, %rax          # Set return value to 0.
26         jmp end
27 one:    movq $1, %rax          # Set return value to 1.
28 end:    leave                  # Return the state of the base and
29                                #  stack pointers to their
30                                #  original state.
31         ret                    # Return from the current function
32                                #  with the contents of rax as the
33                                #  return value.
```

B `fib-ext.c`

Listing 3: `fib-ext.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  extern unsigned long long fib(unsigned long long n);
5
6  int main(int argc, char *argv[]) {
7      unsigned long long n, r;
8      n = atoll(argv[1]);
9      r = fib(n);
10     printf("fib(%llu) = %llu\n", n, r);
11
12     return 0;
13 }
```

C fib_iter.S

Listing 4: fib_iter.S

```
1  .section .text
2  .global fib_iter
3
4  fib_iter:
5          push %rbp               # Save old base pointer to stack.
6                                  # This also makes sure that rsp is
7                                  #  16-byte aligned.
8          movq %rsp, %rbp         # Write new base pointer
9          cmpq $1, %rdi           # Compare the argument to 1
10         je one                  # If n = 1 go to 'one' label.
11         cmpq $0, %rdi
12         je zero                 # If n = 0, go to 'zero' label.
13
14
15         movq $1, %rcx           # Set minus_one (rcx) to 1
16         movq $0, %rdx           # Set minus_two (rdx) to 0
17         movq %rcx, %rax         # Set r (rax) to minus_one (rcx)
18
19 loop:   addq %rdx, %rax         # r (rax) = minus_one (rcx)
20                                 #           + minus_two (rdx)
21                                 # rax is already set to minus_one
22                                 #  due to the last line in the loop
23                                 # The first time the loop is run,
24                                 #  this is handled before the loop.
25         movq %rcx, %rdx         # minus_two (rdx) = minus_one (rcx)
26         movq %rax, %rcx         # minus_one (rcx) = r (rax)
27         decq %rdi               # Decrement the counter by 1.
28         cmpq $1, %rdi           # Loop if the counter is over 1:
29         jle end                 # If counter is 1:
30                                 #  End and return r (rax)
31         jmp loop                # If counter is 0: loop.
32
33 zero:   movq $0, %rax           # Return 0.
34         jmp end
35 one:    movq $1, %rax           # Return 1.
36 end:    leave                   # Return the state of the base and
37                                 #  stack pointers to their
38                                 #  original state.
39         ret                     # Return r (rax)
```

D fib_iter-ext.c

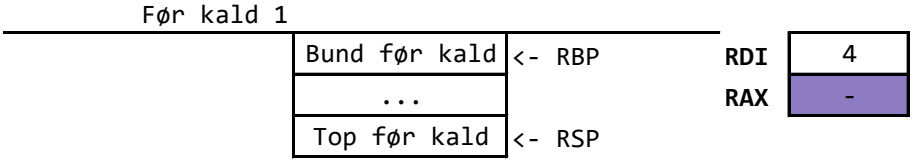
Listing 5: fib_iter-ext.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  extern unsigned long long fib_iter(unsigned long long n);
5
6  int main(int argc, char *argv[]) {
7      unsigned long long n, r;
8      n = atoll(argv[1]);
9      r = fib_iter(n);
10     printf("fib(%llu) = %llu\n", n, r);
11
12     return 0;
13 }
```

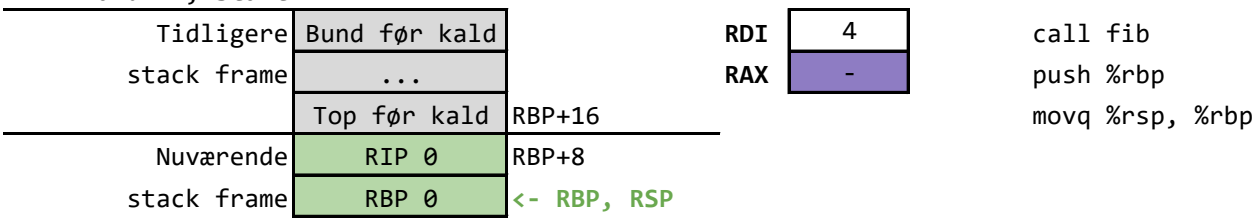
Farveforklaring

Nyt element eller movq til register	Rammedybde Dybde 4
Ændret ved addq incq eller decq	Dybde 3 Dybde 2
Udetermineret	Dybde 1
~~Slettet element~~	Dybde 0

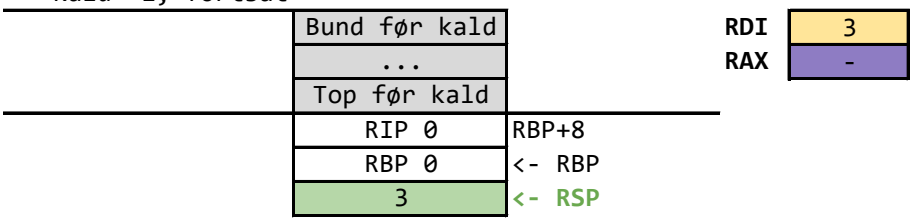
Før kald 1



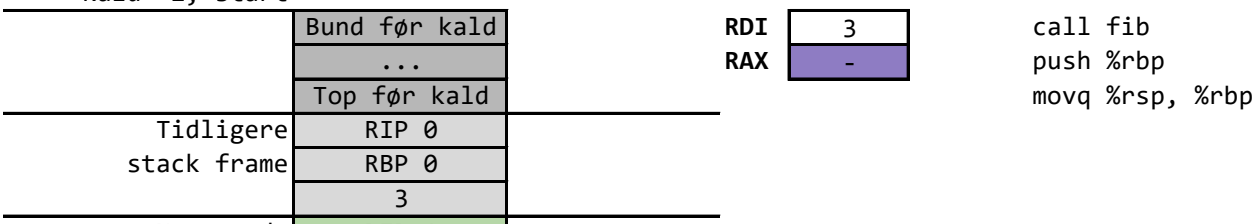
Kald 1, start



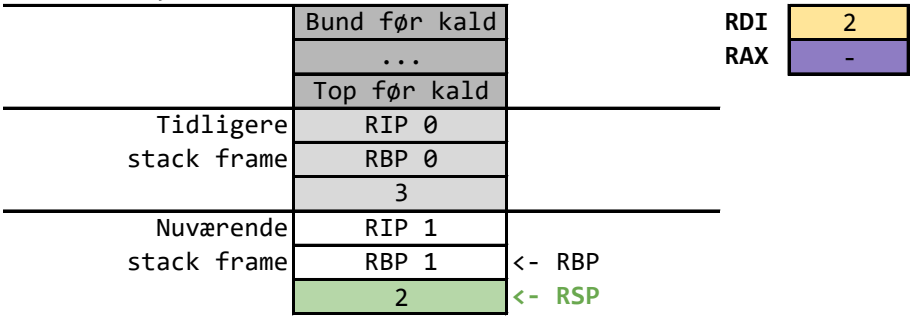
Kald 1, fortsat



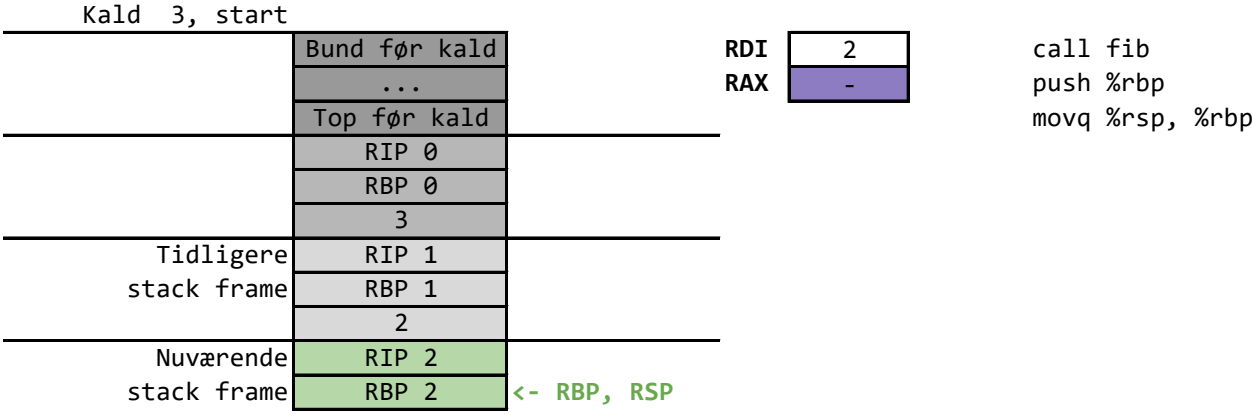
Kald 2, start



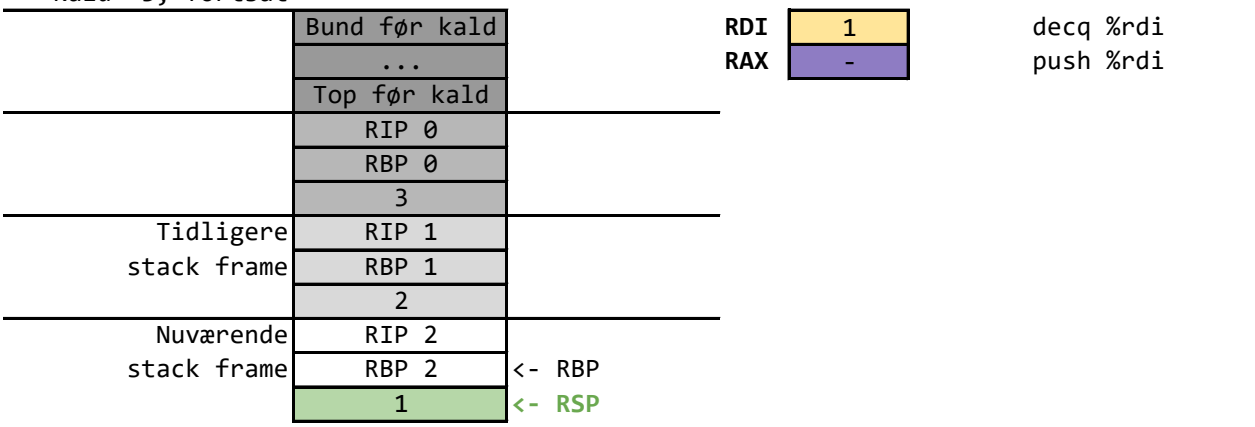
Kald 2, fortsat



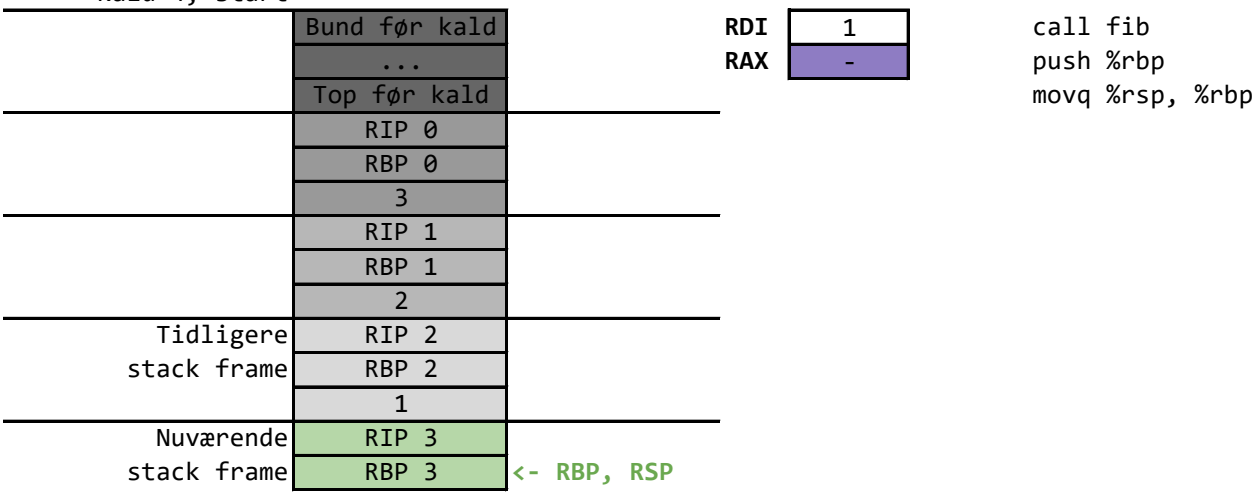
Kald 3, start



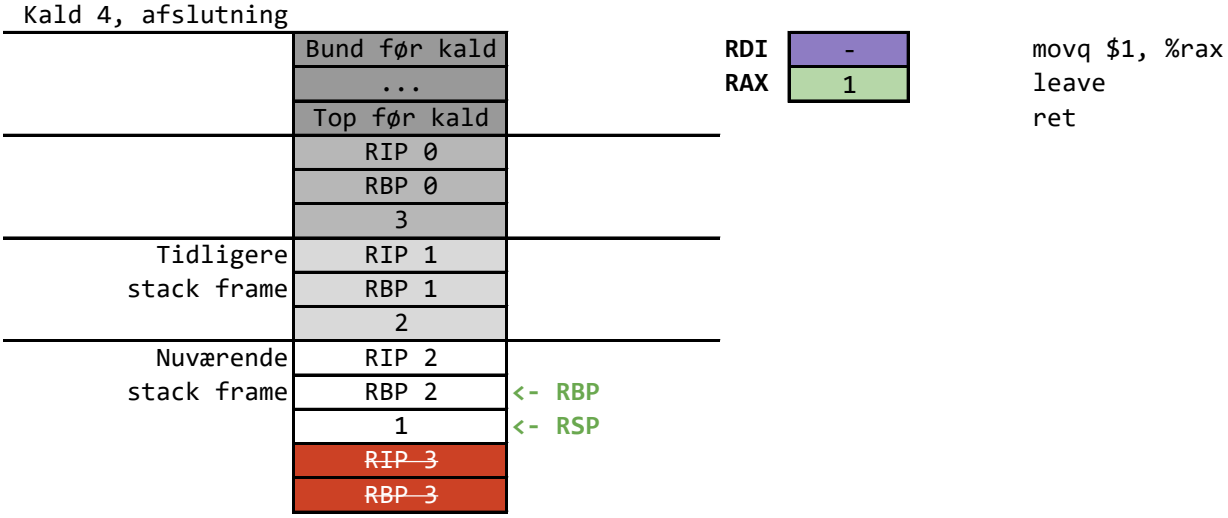
Kald 3, fortsat



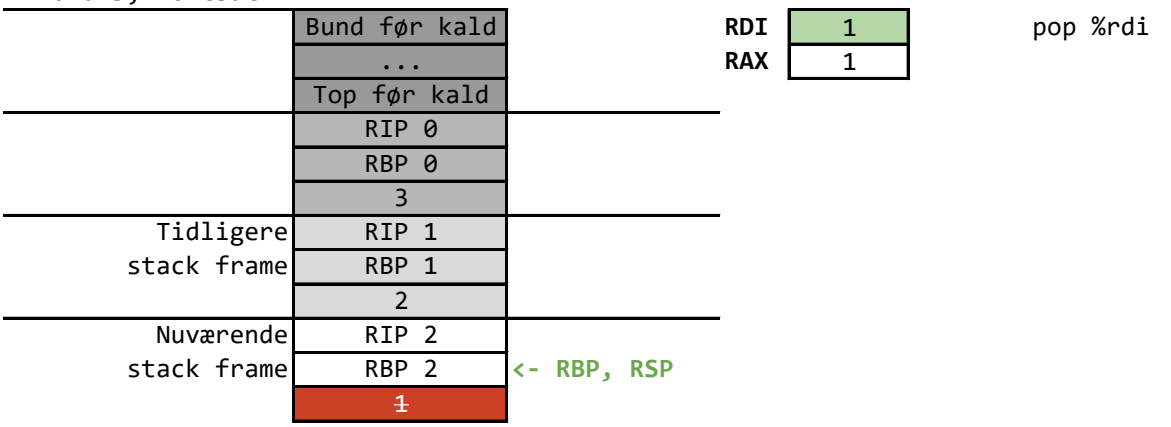
Kald 4, start



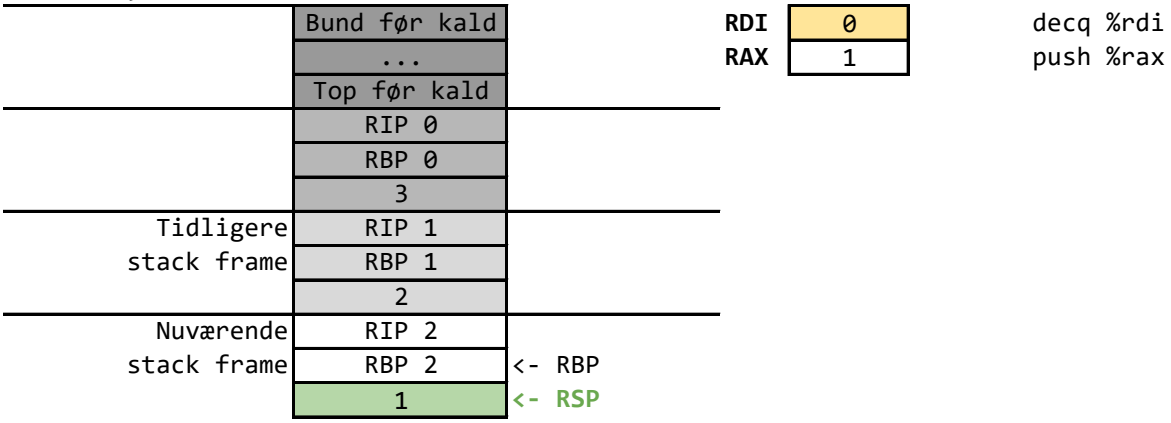
Kald 4, afslutning



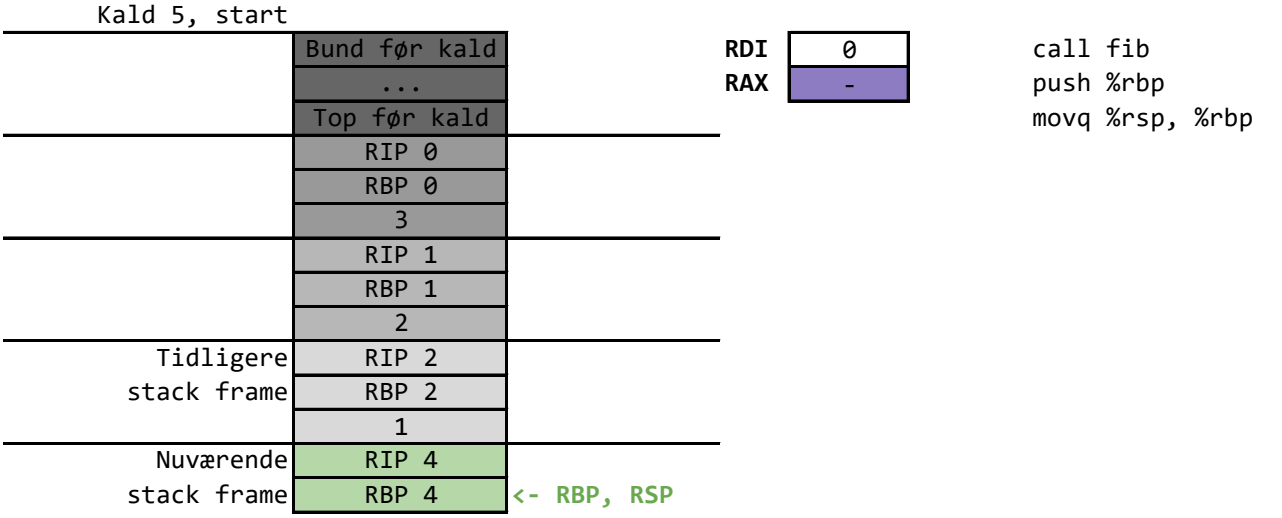
Kald 3, fortsat 2



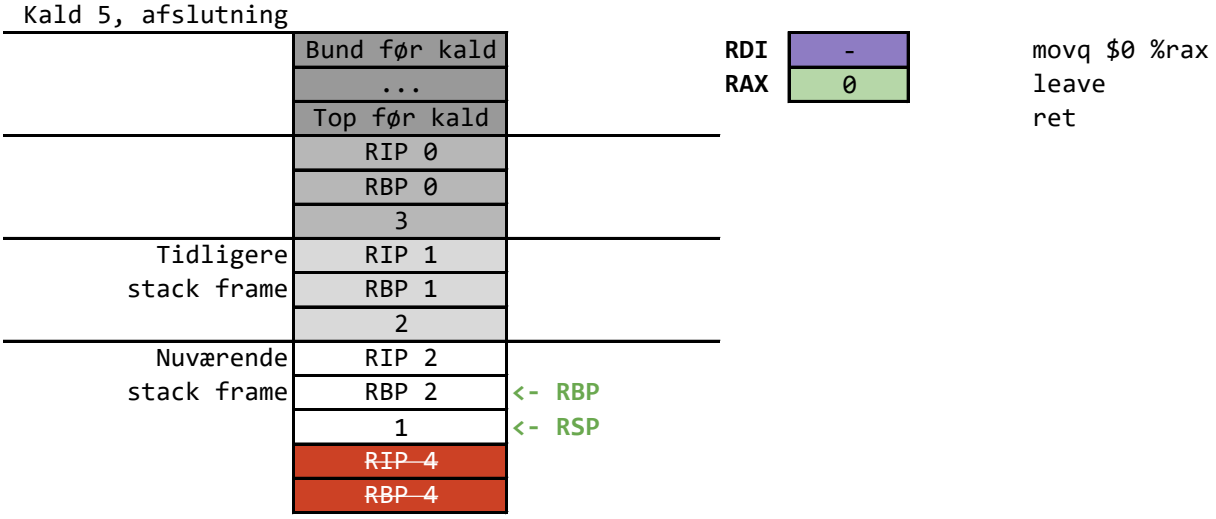
Kald 3, fortsat 3



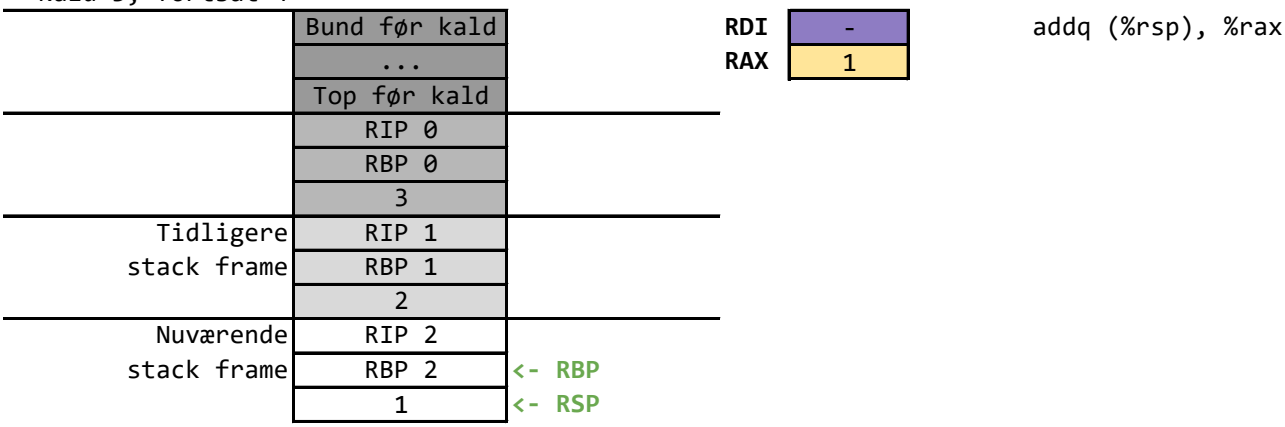
Kald 5, start



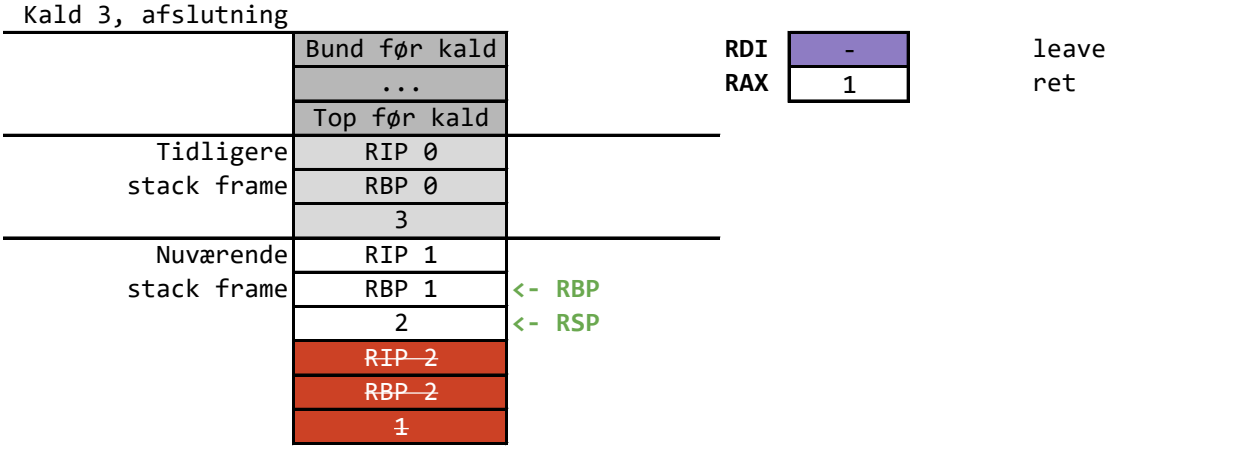
Kald 5, afslutning



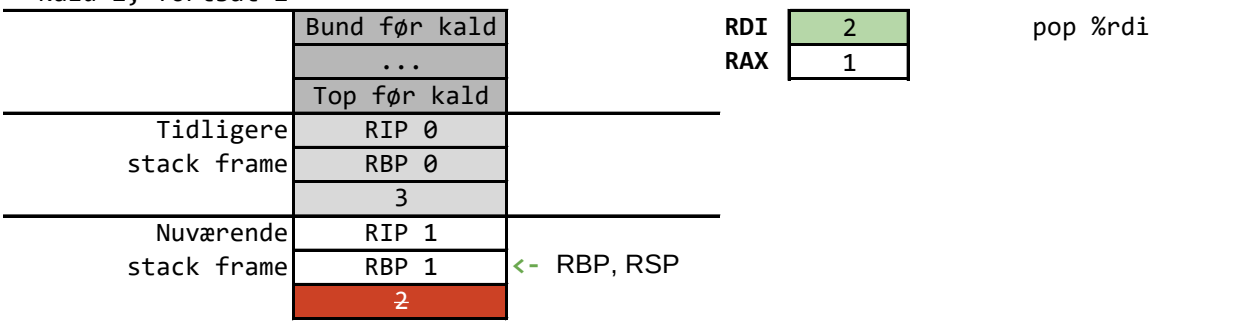
Kald 3, fortsat 4



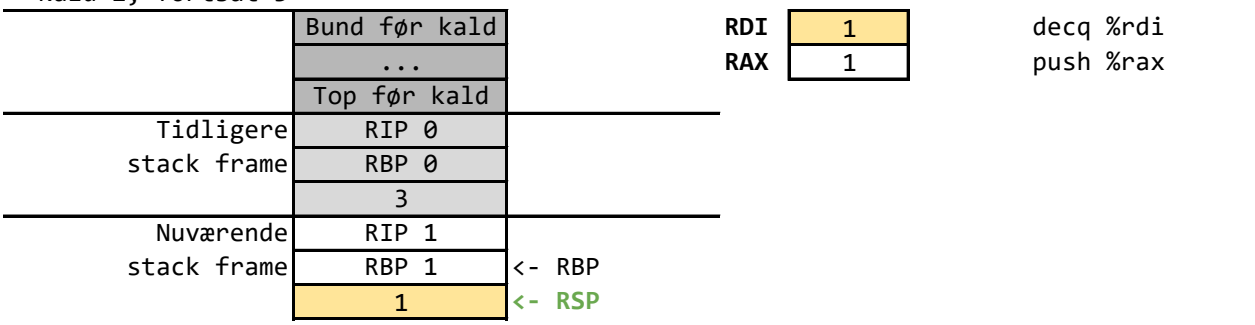
Kald 3, afslutning



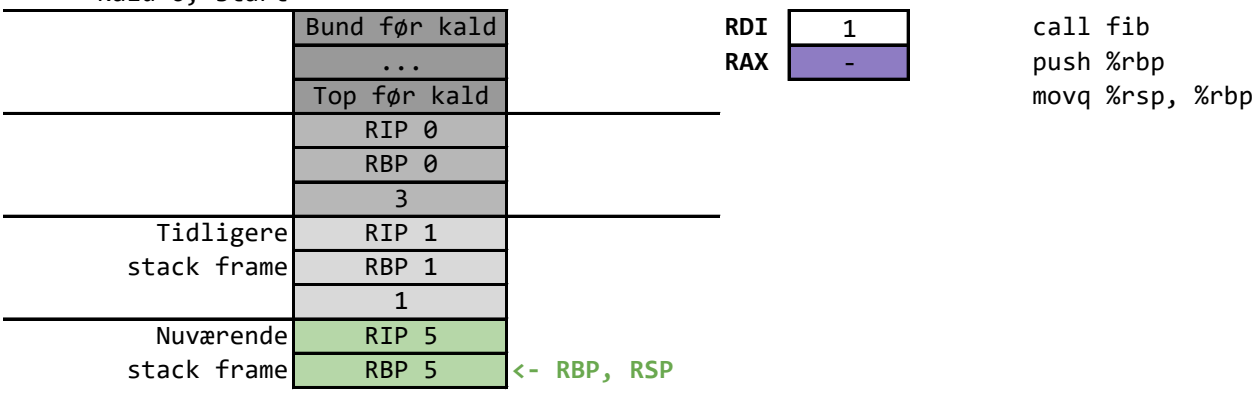
Kald 2, fortsat 2



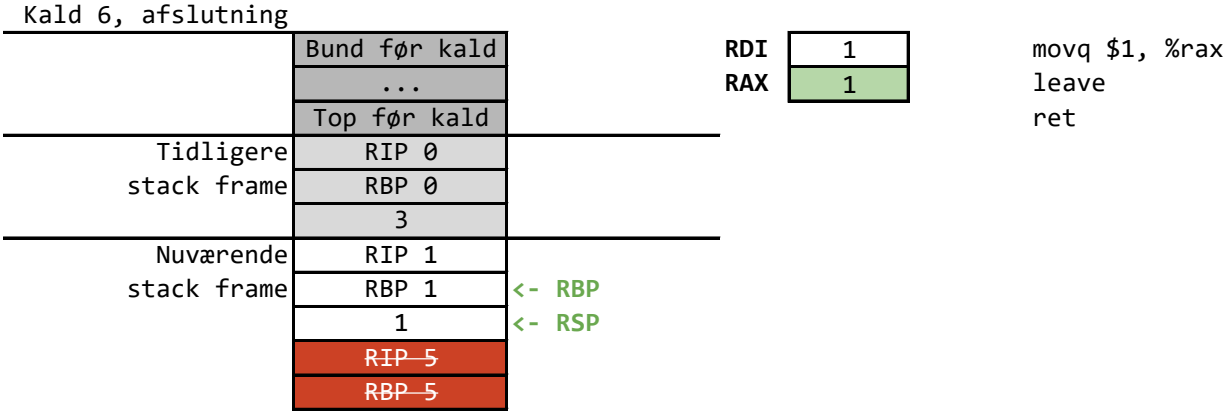
Kald 2, fortsat 3



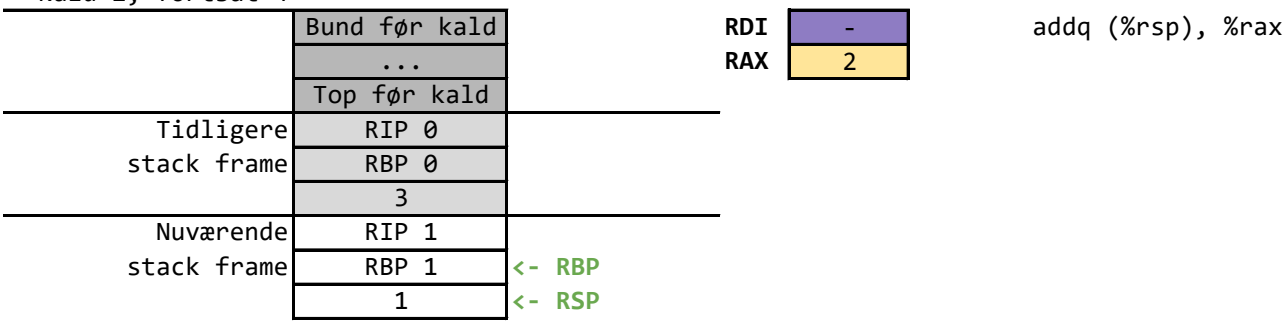
Kald 6, start



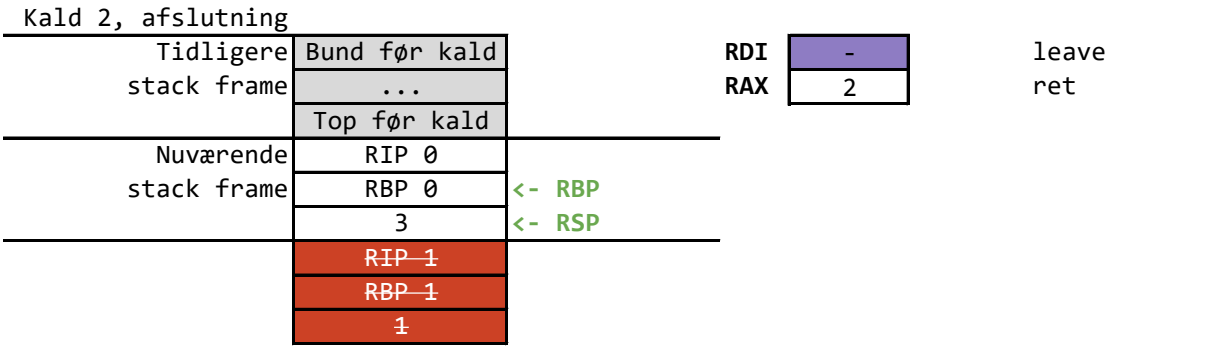
Kald 6, afslutning



Kald 2, fortsat 4

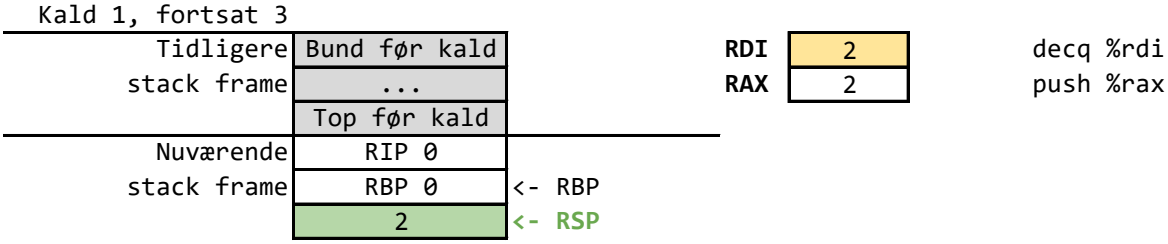


Kald 2, afslutning

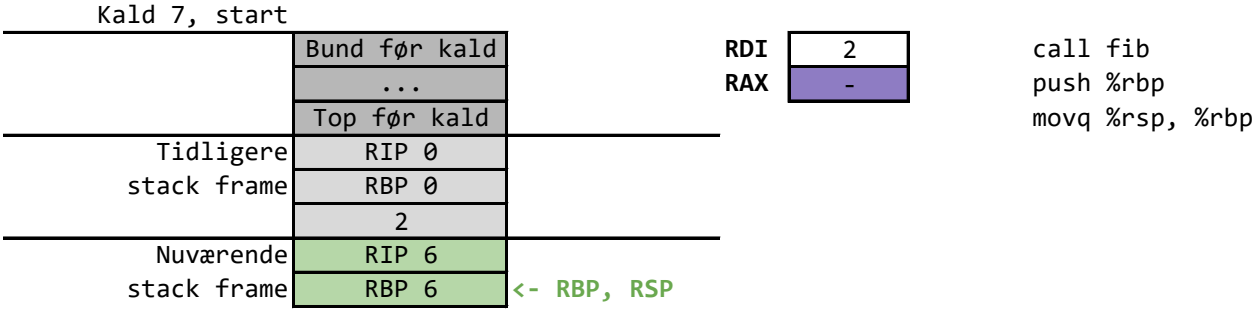


Kald 1, fortsat 2

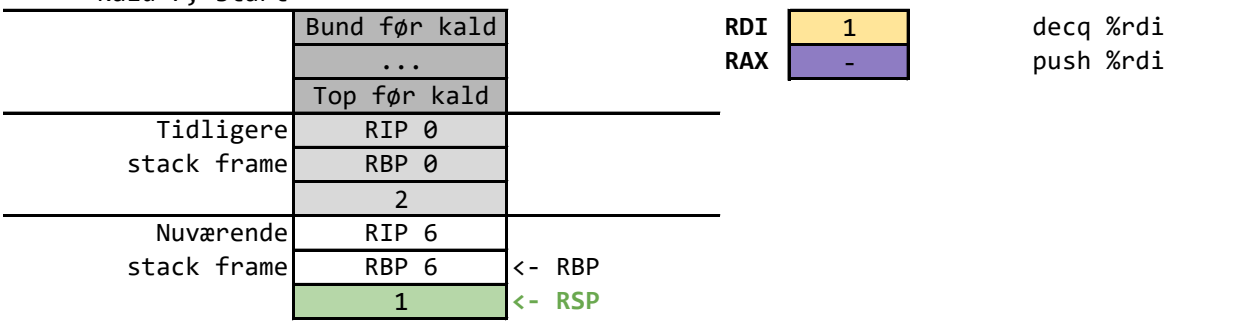
Kald 1, fortsat 3



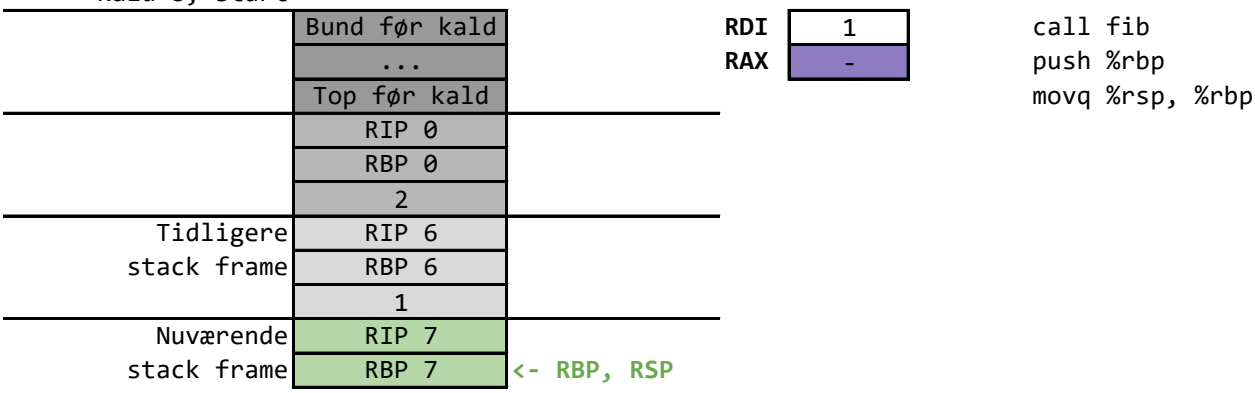
Kald 7, start



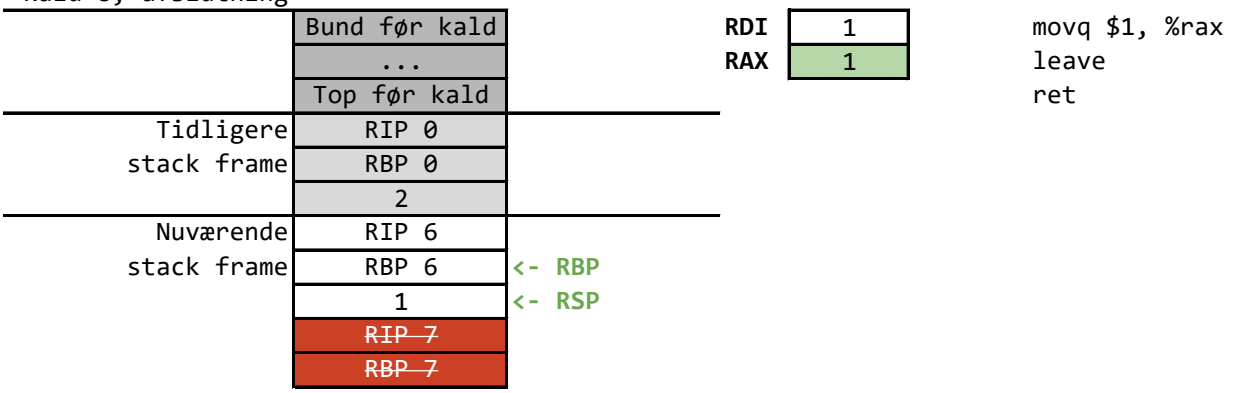
Kald 7, start



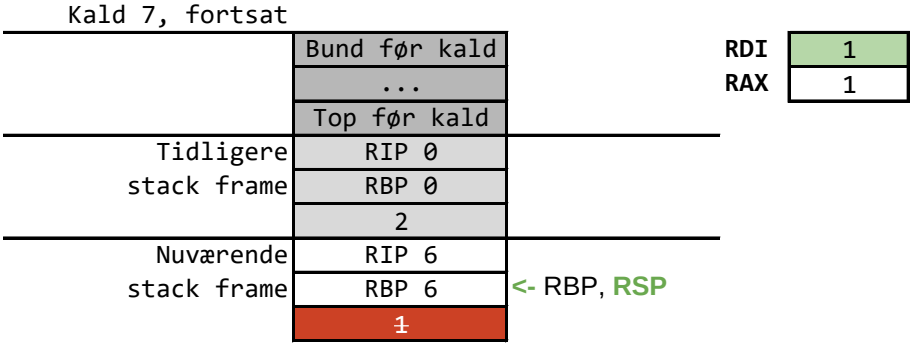
Kald 8, start



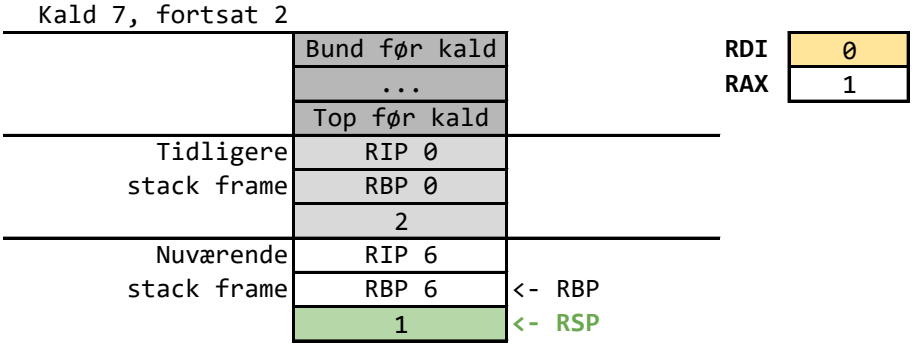
Kald 8, afslutning



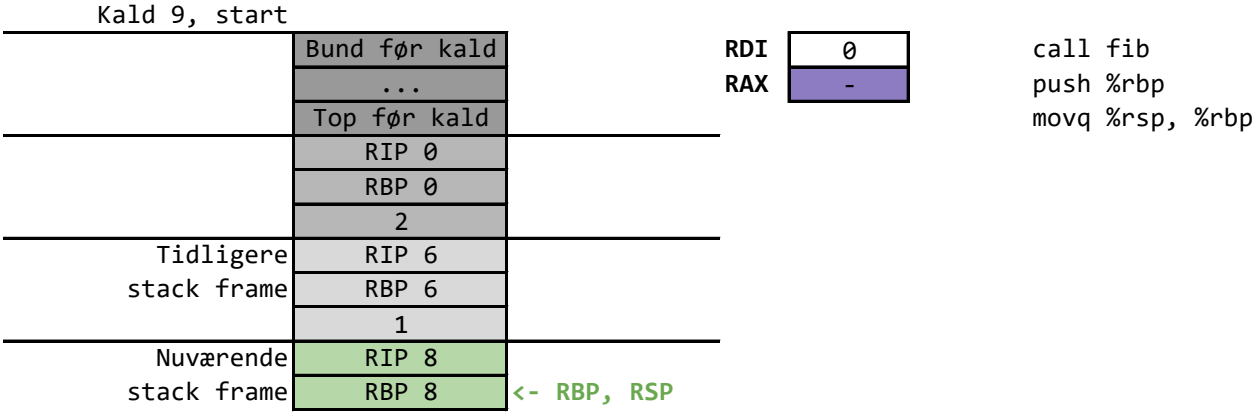
Kald 7, fortsat



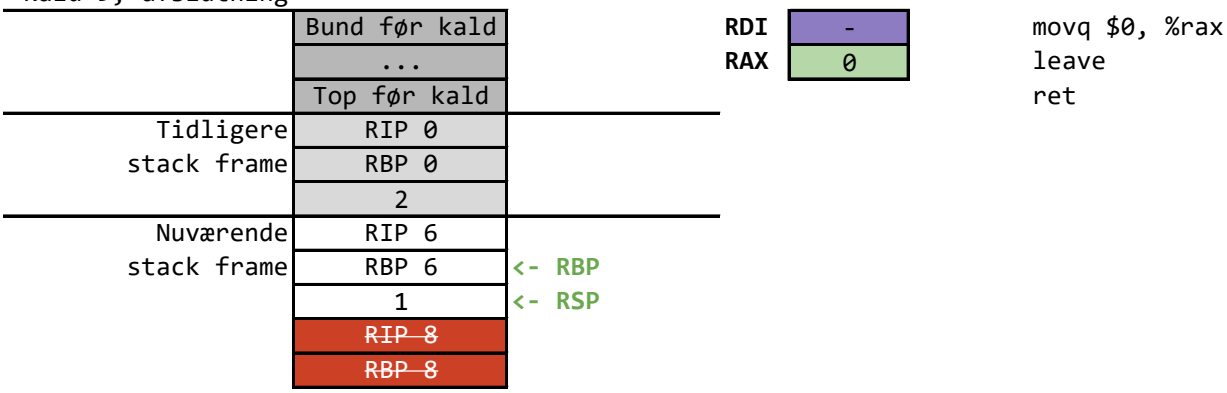
Kald 7, fortsat 2



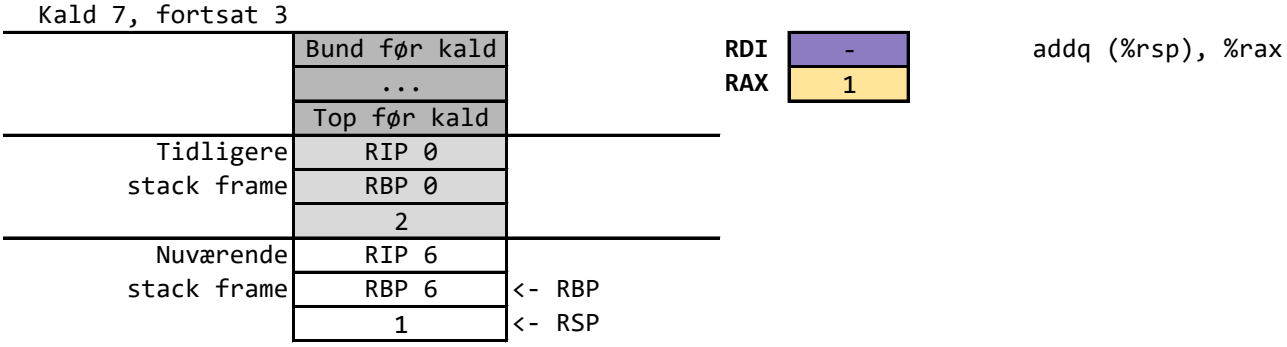
Kald 9, start



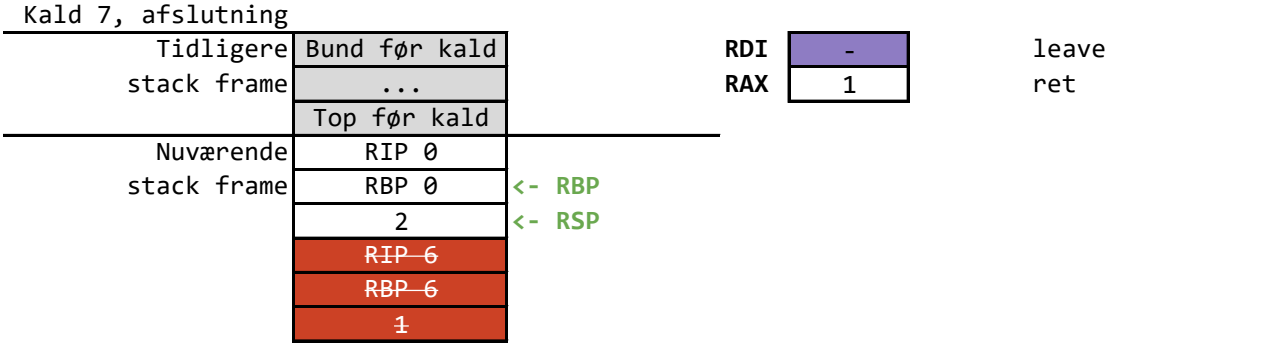
Kald 9, afslutning



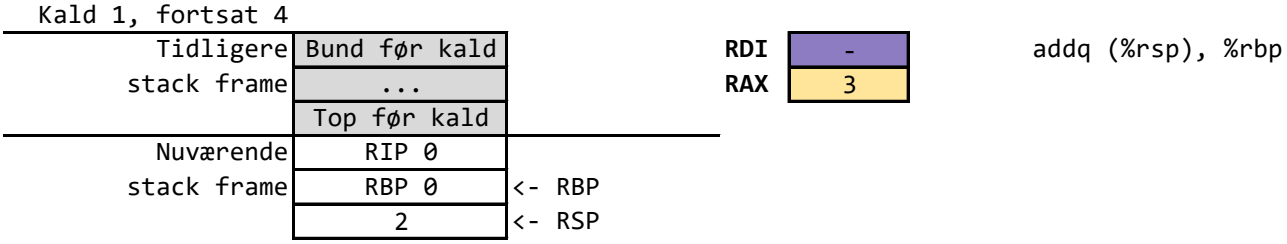
Kald 7, fortsat 3



Kald 7, afslutning



Kald 1, fortsat 4



Kald 1, afslutning

Efter kald 1