Ugeopgave 5

Computerarkitektur

Kristian Gausel[1], Rasmus Skovdal[2], og Steffan C. S. Jørgensen[3]

[1]201509079, 201509079@post.au.dk
[2]201509421, rasmus.skovdal@post.au.dk
[3]201505832, 201505832@post.au.dk

1. maj 2016

Resumé

I denne rapport vil vi implementere beregning af fibonacci-funktionen i *C* og *Assembly*.

Indledning

Fibonacci-funktionen er defineret som følger

$$
\begin{aligned}
f(0) &= 0 \\
f(1) &= 1 \\
f(n) &= f(n-1) + f(n-2), n > 1
\end{aligned}
$$

Dette vil vi implementere i C samt i $Assembly$.

Opgave A

Fibonacci-funktionen, der ses i kodeudsnit 1, er implementeret i *C*-kode. I koden indlæses argumenter fra kommandolinjen med *atoi* (**a**rgument **to** **i**nteger) i linje 13. I *fib*-metoden anvendes et switch-statement. Hvis inputtet til metoden er hhv. 0 eller 1 returneres værdien af inputtet. Hvis inputtet derimod har andre værdier, kalder metoden sig selv ud fra den rekursive definitionen af fibonacci-funktionen.

Kode 1: Metoden *fib(n)* i *C*

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int fib(int a) {
5    switch( a ) {
6      case 0:
7      case 1:
8        return a;          // If a = 0 or a = 1, return a
9      default:             // Else do recursive call
10       return fib(a-1) + fib(a-2); }}
11
12 int main(int argc, char *argv[]) {
13   int a = atoi(argv[1]); // Set a to the argument
14
15   // Print the result
16   printf("fib(%i) = %i\n", a, fib(a));
17   return 0; }            // Return some value
```

Opgave B

I denne opgave har vi lavet en ny udgave af programmet fra kodeudsnit 1, hvor det er implementeret i x86-64 symbolsk maskinsprog i en ekstern fil. I kodeudsnit 2 ses den del af programmet, der er kodet i *C*, og i kodeudsnit 3 ses *Assembly*-delen.

Kode 2: *C*-delen af koden

```c
#include <stdio.h>
#include <stdlib.h>

extern int fibB(int n); // Import the Assembly-code

int main(int argc, char *argv[]) {
  int n = atoi(argv[1]); // Set n to the argument

  // Print the result
  printf("fib(%i) = %i\n", n, fibB(n));
  }
```

Ved test af det nye program findes det, at vi når frem til de samme, rigtige resultater, og at programmet der består af en kombination af *C*- og *Assembly*-kode er marginalt hurtigere end programmet skrevet udelukkende i *C*-kode.

Kode 3: *Assembly*-delen af koden

```
.section .text
.global fibB

fibB:
    pushq  %rbp           # save old base pointer
    movq   %rsp,%rbp      # create new base pointer
    subq   $16, %rsp      # local variable space
                          #   for n and fib(n-1)

    cmpq   $1,%rdi        # compare n to 1
    je     is_one         # if n = 1, return 1
    jl     is_lte_zero    # if n < 1, return 0

    decq   %rdi           # n = n-1
    movq   %rdi, -8(%rbp) # save (n-1)
    call   fibB           # recursive call: fib(n-1)
        # n is read from the local variables.
        #   this is necessary, as no register
        #   is safe with a recursive call.
    movq   -8(%rbp), %rdi # load (n-1)
    decq   %rdi           # create n-2 with (n-1)-1
        # fib(n-1) is saved as a local variable
        #   this is also necessary, as no register
        #   is safe with a recursive call.
    movq   %rax, -16(%rbp) # save fib(n-1)
    call   fibB           # recursive call: fib(n-2)
    movq   -16(%rbp), %rdx # load fib(n-1)
    addq   %rdx,%rax      # fib(n-1) + fib(n-2)
    leave                 # clean up (remake stack)
    ret                   # return

is_lte_zero:
    movq $0, %rax         # set return value to 0
    leave                 # remake caller's stack
    ret                   # return

is_one:
    movq $1, %rax         # set return value to 1
    leave                 # remake caller's stack
    ret                   # return
```

Opgave C

Det bemærkes, at koden i kodeudsnit 3 overholder kaldkonventionen, da den etablerer nye stakafsnit ved at gemme den gamle *base pointer* og etablerer en ny ved hvert metodekald i linjerne *5-6*.

```
5   pushq  %rbp          # save old base pointer
6   movq   %rsp,%rbp     # create new base pointer
```

Desuden sørger den for at kalde *leave* og returnere i *rax*-registret i linjerne *28-30*, *33-35* og *38-40*.

```
28  addq   %rdx,%rax     # fib(n-1) + fib(n-2)
29  leave                # clean up (remake stack)
30  ret                  # return
```

Vores implementation af fib benytter sig desuden af lokale variable, hvilket der gøres plads til vha. flytning af *stakpointer*.

```
7   subq   $16, %rsp     # local variable space
8                        #   for n and fib(n-1)
```

```
14  decq   %rdi          # n = n-1
15  movq   %rdi, -8(%rbp) # save (n-1)
```

```
27  movq   -16(%rbp), %rdx # load fib(n-1)
```

Parametre placeres i det korrekte register (*rdi*) ved alene at holde n opdateret i dette register.

```
14  decq   %rdi          # n = n-1
15  movq   %rdi, -8(%rbp) # save (n-1)
16  call   fibB          # recursive call: fib(n-1)
```

```
20  movq   -8(%rbp), %rdi # load (n-1)
21  decq   %rdi          # create n-2 with (n-1)-1
```

I appendiks A har vi lavet en optegning af stakkens udseende for *fib(4)*.

Opgave D

Fibonacci-funktionen kan også implementeres iterativt, hvilket er mere kompliceret, men derimod meget mere effektivt. Det har vi gjort nedenfor i kodeudsnittene 4 og 5 i hhv. *C*- og *Assymbly*-kode.

Efter at x86-64 intel processoren har fået tilføjet mange flere registre i forhold til forgængerne, er det muligt at implementere den iterative metode helt uden brug af stakken udover det krævede for kaldkonventioner. Herved bliver dette ikke kun hurtigere ved en iterative implementation frem for en rekursiv, men også alene ved brug af registre, da disse er omtrent 100 gange hurtige at bruge end stakken.

Kode 4: *C*-delen af den iterative fibonacci bestemmelse

```c
#include <stdio.h>
#include <stdlib.h>

// Import the Assembly-code
extern int fib_iter(int n);

int main(int argc, char *argv[]) {
  int n = atoi(argv[1]); // Set n to the argument

  // Print the result
  printf("fib(%i) = %i\n", n, fib_iter(n)); }
```

Opgave E

I denne opgave vil vi se på de fire forskellige implementationer af fibonacci-funktionen. Vi vil se på implementationerne skrevet i *C* og i *Assembly*. Vi vil både se på de rekursive og iterative implementationer. Vi anvender UNIX-kommandoen *time* til sammenligningen af kørselstiderne, der findes som et gennemsnit af seks kørsler.

Tilgang	**Kode**	**Gennemsnitligt tidsforbrug for $fib(46)$**
Rekursiv	C	14,74
	Assembly	15,54
Iterativ	C	0,0023
	Assembly	0,0020

Tabel 1: Sammenligning af kørselstider

Som det ses i tabel 1 er de iterative implementationer langt hurtigere end de rekursive. Dette er helt forventeligt, da en iterativ tilgang typisk er hurtigere end en rekursiv tilgang fordi funktionskald bruger mere *regnekraft*, da der bl.a. skal laves en ny *stack frame*, hvilket ikke er nødvendigt i en iterativt, løkke-baseret implementation. Vi forventede, at *Assembly*-koden ville være hurtigere end *C*-koden, og det var også tilfældet i de iterative implementationer. Men når det kom til de rekursive implementationer, så må vi erkende, at *C*-compileren er bedre og mere effektiv end vi er til at skrive *Assembly*-kode.

Kode 5: *Assembly*-delen af den iterative fibonacci bestemmelse

```
.section .text
.global fib_iter

fib_iter:
    pushq   %rbp        # save old base pointer
    movq    %rsp,%rbp   # establish new base pointer
        #%rdi = parameter = n
        #%rax = return value
        #%r8 = fib_new
        #%r9 = fib_old

    cmpq    $1,%rdi     # compare n to 1
    je      is_one      # if n = 1, return 1
    jl      is_lte_zero #if n < 1 (n = 0), return 0

    movq    $1, %r8     # r8 = fib(1) = 1
    movq    $0, %r9     # r9 = fib(2) = 0

    decq    %rdi
    movq    %rdi, %rcx # loop n-1 times

fib_loop:
    movq    %r8, %rax   # %rax = fib_new
    addq    %r9, %rax   # %rax = fib_new + fib_old
    movq    %r8, %r9    # fib_old = fib_new
    movq    %rax, %r8   # fib_new = old+new
    loop    fib_loop

    leave
    ret

is_lte_zero:
    movq $0, %rax       # set return value in rax to 0
    leave               # remake caller's stack
    ret                 # return

is_one:
    movq $1, %rax       # set return value in rax to 1
    leave               # remake caller's stack
    ret                 # return
```

A Opgave C: Stacktrace

I dette appendiks har vi lavet en manuel, fuld optegning af stakkens udseende, når programmet køres med argumentet 4, hvilket kan findes i 6.

Herunder ses det *call tree*, der gennemløbes ved kald af *fib(4)*. Dette er inkluderet for at give et bedre overblik over rekursionens forløb.

```
fib(4)
    n-1 = 3
    fib(3)
        n-1 = 2
        fib(2)
            n-1 = 1
            fib(1)
                return 1
            n-1 = 0
            fib(0)
                return 0
            return 1 + 0
        n-1 = 1
        fib(1)
            return 1
        return 1 + 1
    n-1 = 2
    fib(2)
        n-1 = 1
        fib(1)
            return 1
        n-1 = 0
        fib(0)
            return 0
        return 1 + 0
    return 2 + 1

result: 3
```

Kode 6: Stacktrace af fib(4) | X: Stackpointer til caller af *fibB* | Y: Stackpointer til roden af *fibB*

```
5. pushq  %rbp                   | X |
6. subq   $16, %rsp              | ? | X |
15. movq  %rdi, -8(%rbp)         | n: 3 | X |
16. call   fibB
    5. pushq  %rbp               | Y || n: 3 | X |
    7. subq   $16, %rsp          | ? | Y || n: 3 | X |
    15. movq  %rdi, -8(%rbp)     | n: 2 | Y || n: 3 | X |
    16. call   fibB
        5. pushq  %rbp           | Y + 2 || n: 2 | Y || n: 3 | X |
        7. subq   $16, %rsp      | ? | Y + 2 || n: 2 | Y || n: 3 | X |
        15. movq  %rdi, -8(%rbp) | n: 1 | Y + 2 || n: 2 | Y || n: 3 | X |
        16. call   fibB
            5. pushq  %rbp       | Y + 4 || n: 1 | Y + 2 || n: 2 | Y || n: 3 | X |
            7. subq   $16, %rsp  | ? | Y + 4 || n: 1 | Y + 2 || n: 2 | Y || n: 3 | X |
            11. je     is_one
            40. ret
        25. movq %rax, -16(%rbp) | fib(1): 1 | Y + 2 || n: 2 | Y || n: 3 | X |
        26 call fibB
            5. pushq  %rbp       | Y + 4 || fib(1): 1 | Y + 2 || n: 2 | Y || n: 3 | X |
            7. subq   $16, %rsp  | ? | Y + 4 || fib(1): 1 | Y + 2 || n: 2 | Y || n: 3 | X |
            12. jl     is_lte_zero
            35. ret
        30. ret
    25. movq  %rax, -16(%rbp)    | fib(2): 1 | Y || n: 3 | X |
    26. call   fibB
        5. pushq  %rbp           | Y + 2 || fib(2): 1 | Y || n: 3 | X |
        7. subq   $16, %rsp      | ? | Y + 2 || fib(2): 1 | Y || n: 3 | X |
        11. je     is_one
        40. ret
    30. ret
25. movq %rax, -16(%rbp)         | fib(3): 2 | X |
26. call fibB
    5. pushq  %rbp               | Y || fib(3): 2 | X |
    7. subq   $16, %rsp          | ? | Y || fib(3): 2 | X |
    15. movq  %rdi, -8(%rbp)     | n: 1 | y || fib(3): 2 | X |
    16. call   fibB
        5. pushq  %rbp           | Y + 2 || n: 1 | y || fib(3): 2 | X |
        7. subq   $16, %rsp      | ? | Y + 2 || n: 1 | y || fib(3): 2 | X |
        11. je     is_one
        40. ret
    25. movq  %rax, -16(%rsp)    | fib(1): 1 | Y || fib(3): 2 | X |
    26. call   fibB
        5. pushq  %rbp           | Y + 2 || fib(1): 1 | Y || fib(3): 2 | X |
        7. subq   $16, %rsp      | ? || Y + 2 || fib(1): 1 | Y || fib(3): 2 | X |
        12. jl     is_lte_zero
        35. ret
30. ret
```