# DD1339 Introduktion till datalogi 2013/2014

# Uppgift nummer: 9

# Namn: Marcus Larsson

# Grupp nummer: 5

# Övningsledare: Marcus Dicander

# Betyg: ..... Datum: .............. Rättad av: .......................................

# Exercise Assembler

### multiplicerar varje element i en vektor med samma tal,

// This program multiply every number in a vector with the number 3 and overwrites the values with //the new.

word aSize 8 // length of array a

word a -1 2 -3 4 -5 6 -7 8 //This is the array

// Compute start address

loadc r1 a // r1 = &a[0] (the address of a[0])

// Compute where to stop

loadc r0 a // r0 = &a[0]

load r2 aSize // r2 = number of elements in a

add r2 r2 r2 // r2 = 2\*r2 = number of bytes in a

add r0 r0 r2 // r0 = &a[aLen] (first address after array)

loadc r5 3 // r5 the number we multiply with

Loop: loadr r2 r1 // r2 = a[i]

mul r3 r5 r2 // r5 = r5 \* r2

storer r3 r1 // &a[i]=r3

addc r1 2 // i++

jumpn r1 Loop // if &a[i] != &a[8] goto Loop

### flyttar runt ordet 0xff i ett intressant mönster i minnet på adresserna från 0x80 till 0xff,

//This program will move the word 0xff in an pattern in the RAM memory.

word a -1 //this is the bitpattern we will be moving around (ff)

load re a //re=a

loadc r0 0x10 //r0= hex number 10 This is 16, and stands for that I want to loop 16/2 times , 8 times.

loadc r2 0x80 //r2= hex number 80, The starting point for the word.

Loop: storer re r2 //Place word a in location found in r2

noop //Do nothing. Like a pause.

storer rf r2 //Place 00 in location found in r2 (overwrite the word a)

addc r2 0x12 //Increase r2 with 18. This will set the next location to be //dioganally down to the right.

addc r1 2 // i++

jumpn r1 Loop // if r1!=r0 go to loop. (if number 16 is not reached, loop //has not been done 8 times yet)

### beräknar och skriver ut (i RAM) de tretton första Fibonacci-talen (1, 1, 2, 3, 5, 8, ...).

//Calculates 13 first fibonacci numbers and saves them in order in RAM

word aSize 13 // length of array a

word a //This is an array

// Compute start address

loadc r1 a // r1 = &a[0] (the address of a[0])

// Compute where to stop

loadc r0 a // r0 = &a[0]

load r2 aSize // r2 = number of elements in a

add r2 r2 r2 // r2 = 2\*r2 = number of bytes in a

add r0 r0 r2 // r0 = &a[aLen] (first address after array)

//Load start values of fibonacci 0 and 1 in register 3 and 4

loadc r3 0

loadc r4 1

//Make sure loop starts at 2, first number calculated should be a[2].

addc r1 4 // i=i+2, the start position in the array.

Loop: add r5 r3 r4 //r5=r3+r4

storer r5 r1 //&a[i]=r5

move r4 r3 //r3=r4

move r5 r4 //r4=r5

addc r1 2 // i++

jumpn r1 Loop // if &a[i] != &a[13] goto Loop

# Exercise Ordouppskattning

**Algorithm** Loop1(*n*):

a = 0

**for** *i* = 1 **to** *n*

a += i

Det är endast en loop från 1 till n. Detta blir O(n).

**Algorithm** Loop2(*n*):

b = 1

**for** *i* = 1 **to** *4n*

b++

Endast 1 loop från 1 till 4n. Det är också O(n).

**Algorithm** Loop3(*n*):

c = 1

**for** *i* = 1 **to** *n2*

c--

Vi väljer att räkna c-- som den karaktersitiska operationen. Den utförs 1 gång varje varv i loopen som kör n2 gånger. Därför blir det O(n2)

**Algorithm** Loop4(*n*):

d = 5

**for** *i* = 1 **to** *3n*

**for** *j* = 1 **to** *i*

d = d + j

Väljer att räkna tilldelningen d=d+j. Den inre loopen räkna utförs första gången endast 1 gång, nästa 2 gånger osv till 3n gånger. Det blir en aritmetisk summa som bildas. En aritmetisk summa är av O(n2). (n(n+1)/2)

**Algorithm** Loop5(*n*):

e = 5

**for** *i* = 1 **to** *n2*

**for** *j* = 1 **to** *i*

e = e + j

Här har vi en yttre loop som utförs n2 gånger. Innanför den har vi en loop som kör från 0 till i. Det är en aritmetisk summa som inre loop, en operation utförs inne i loopen så det räcker att räkna antalet gånger looparna körs. Jag sa tidigare att aritmetisk summa innebär o(n2). Nu blir det O(n2)\*O(n2) eftersom dem körs i varandra. Detta blir O(n4).

# Exercise (n+1)3=O(n3)

 Definition: f(*n*) är *O*(g(*n*)) om det finns positiva konstanter *c* och *n*0 så att *f*(*n*) ≤ cg(*n*) för alla *n* ≥ *n*0.

Vi har f(n) =(n+1)3=n3+3n2+3n+1 och vi har g(n)=n3. För väldigt stora n så påverkar inte 3n2+3n+1 värdet av f(n). Därför är g(n) nästan lika med f(n) vid stora n.

Om vi lägger till en konstant framför g(n) alltså cg(n) så kommer cg(n) att växa snabbare än f(n). Alltså kommer cg(n) att passera f(n) och bli större vid något n=n0, och efter det fortsätta att växa snabbare vilket innebär att för alla *n* ≥ *n*0 kommer cg(n) vara större än f(n). Enligt definition är då f(n)=O(g(n)).

# Exercise Körtid på Reversealgoritm

Jag väljer att räkna en operation som alltid körs. Eftersom swap bara körs om elementen inte är lika så är det inte alltid som den operationen kommer att utföras. Däremot utförs alltid kollen om elementen är lika. Alltså väljer jag att räkna: **if** a[i] != a[n-1-i]

Loopen går från 0 till n/2, varje varv i loopen utförs if-satsen en gång. Detta innebär att den utförs (n/2) gånger. För udda tal blir det inte ett heltal, dessa tal avrundas nedåt. Så n=5 utförs if-satsen 2 gånger. Ordo för detta är O(n). Eftersom man kör igenom halva listan med n-element och funktionen blir n/2, en konstant 1/2 framför n.

Om man inte skulle kolla om elementen är lika, så kan man tänka sig att exempelvis java skulle behöva 3 operationer för varje swap, en där man sparar ena vädet, sedan 2 tilldelningar till platserna i arrayen. Detta skulle ge funktion 3n/2.

Om man tittar på dessa 2 modeller bredvid varandra, alltså med och utan if-satsen så kan man se att extremfallen blir ganska annorlunda. Om alla element i vektorn skulle vara lika så skulle if-satsen göra att färre operationer utförs. Endast n/2 gånger. Men om inga element skulle vara lika skulle det innebära 1 extra operation varje varv i loopen, alltså 4n/2. Utan if-satsen blir funktionen alltid 3n/2 oavsett om elementen är lika eller inte. Båda är O(n) med endast olika konstanter som skiljer dem åt.