



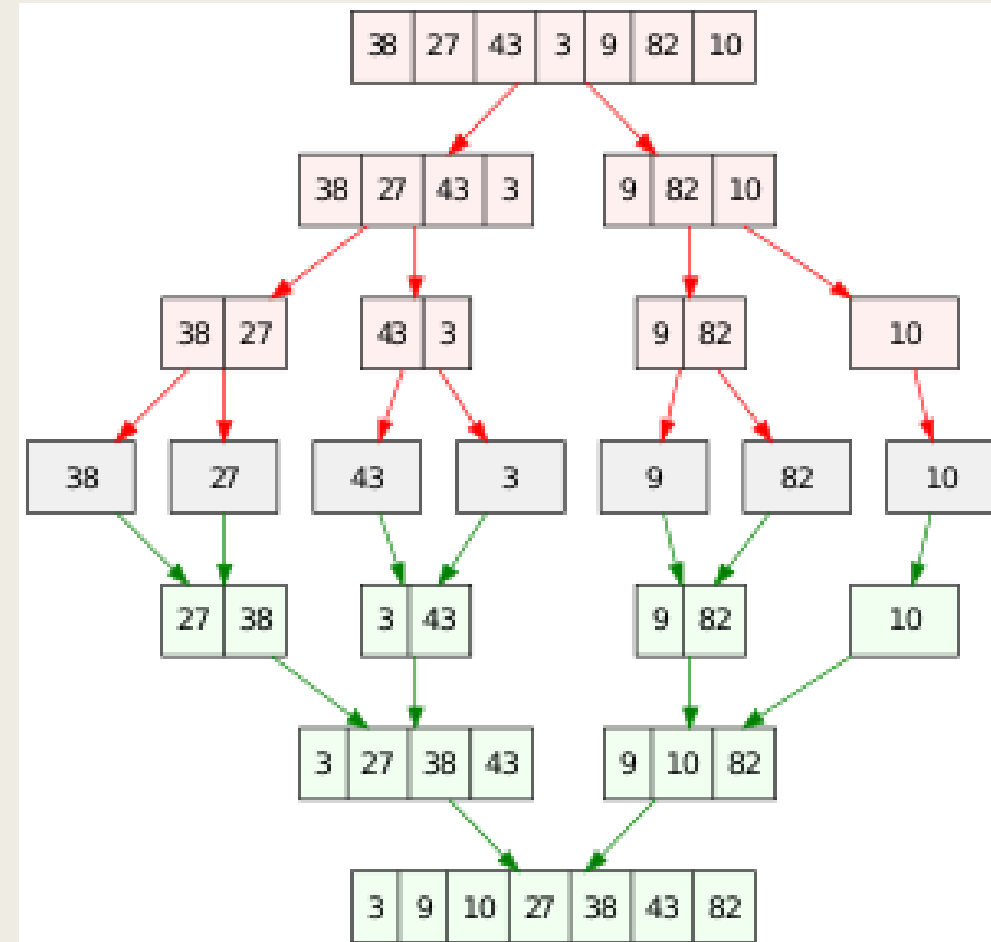
MERGESORT IN RISC-V ASSEMBLY

Stefano Vezzalini – 04/09/2019



II Mergesort

- È un algoritmo di ordinamento ricorsivo basato sul Divide Et Impera
- La sequenza viene divisa a metà in due sottosequenze
- Questa operazione si ripete ricorsivamente fino a quando la sottosequenza è ordinata
- Due sottosequenze ordinate vengono fuse insieme (merge)
- Ha complessità $\theta(n \log n)$



Funzioni principali

`mergesort ()`

- È la funzione principale
- Riceve come parametri due estremi di una sequenza
- Calcola il numero di elementi della sequenza decidendo se si è arrivati al caso base
- Chiama se stesso ricorsivamente due volte su due sottosequenze
- Chiama `merge ()` per unire le due sottosequenze

`merge ()`

- Riceve come parametri le seguenti informazioni sulla sequenza:
 - *Indirizzo del primo elemento*
 - *Indirizzo dell'elemento a metà*
 - *Indirizzo dell'ultimo elemento*
- Fonde due sottosequenze ordinate ritornando una sequenza ordinata

mergesort()

```
3  # MERGESORT(*testArray, first, last)
4  # param a0 -> first element address
5  # param a1 -> last element address
6  ##
7  mergesort:
8
9  # Stack management
10  addi sp, sp, -32      # Adjust stack pointer
11  sd ra, 0(sp)         # Load return address
12  sd a0, 8(sp)         # Load first element address
13  sd a1, 16(sp)        # Load last element address
14
15  # Base case
16  li t1, 1             # Size of one element
17  sub t0, a1, a0       # Calculate number of elements
18  ble t0, t1, mergesort_end # If only one element remains in the array, return
19
20  srli t0, t0, 1        # Divide array size to get half of the element
21  add a1, a0, t0        # Calculate array midpoint address
22  sd a1, 24(sp)        # Store it on the stack
23
24  jal mergesort         # Recursive call on first half of the array
25
26  ld a0, 24(sp)        # Load midpoint back from the stack
27  ld a1, 16(sp)        # Load last element address back from the stack
28
29  jal mergesort         # Recursive call on second half of the array
30
31  ld a0, 8(sp)         # Load first element address back from the stack
32  ld a1, 24(sp)        # Load midpoint address back from the stack
33  ld a2, 16(sp)        # Load last element address back from the stack
34
35  jal merge            # Merge two sorted sub-arrays
36
37  mergesort_end:
38  ld ra, 0(sp)
39  addi sp, sp, 32
40  ret
41
```

```
1  MERGESORT(A,i,j)
2  if i < j
3  then
4      k := floor((i+j)/2)
5  MERGESORT(A,i,k)
6  MERGESORT(A,k+1,j)
7  MERGE(A,i,k,j)
```

merge () 1/2

```
merge:
# Stack management
addi sp, sp, -32      # Adjust stack pointer
sd ra, 0(sp)          # Load return address
sd a0, 8(sp)          # Load first element of first array address
sd a1, 16(sp)         # Load first element of second array address
sd a2, 24(sp)         # Load last element of second array address

mv s0, a0              # First half address copy
mv s1, a1              # Second half address copy

merge_loop:
mv t0, s0              # copy first half position address
mv t1, s1              # copy second half position address
lb t0, 0(t0)           # Load first half position value
lb t1, 0(t1)           # Load second half position value

bgt t1, t0, shift_skip # If lower value is first, no need to perform operations

mv a0, s1              # a0 -> element to move
mv a1, s0              # a1 -> address to move it to
jal shift              # jump to shift

addi s1, s1, 1         # Increment second half index and point to the next element

shift_skip:
addi s0, s0, 1         # Increment first half index and point to the next element
ld a2, 24(sp)          # Load back last element address

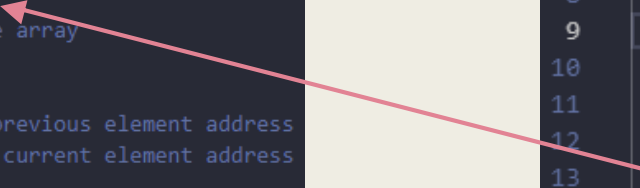
bge s0, a2, merge_loop_end
bge s1, a2, merge_loop_end
beq x0, x0, merge_loop
```

```
1  MERGE(A,i,k,j)
2    n1 := k-i+1
3    n2 := j-k
4    crea L[0..n1] e R[0..n2]
5    for t = 0 to n1-1
6      L[t] := A[i+t]
7    for t = 0 to n2-1
8      R[t] := A[k+1+t]
9    L[n1] := ∞
10   R[n2] := ∞
11   l := 0
12   r := 0
13   for t = i to j
14     if L[l] ≤ R[r]
15       then
16         A[t] := L[l]
17         l := l + 1
18     else
19       A[t] := R[r]
20       r := r + 1
```

merge () 2/2

```
85  ##
86  # Shift array element to a lower address
87  # param a0 -> address of element to shift
88  # param a1 -> address of where to move a0
89  ##
90  shift:
91
92      ble a0, a1, shift_end    # Location reached, stop shifting
93      addi t3, a0, -1          # Go to the previous element in the array
94      lb t4, 0(a0)             # Get current element pointer
95      lb t5, 0(t3)             # Get previous element pointer
96      sb t4, 0(t3)             # Copy current element pointer to previous element address
97      sb t5, 0(a0)             # Copy previous element pointer to current element address
98      mv a0, t3                # Shift current position back
99      beq x0, x0, shift        # Loop again
100
101  shift_end:
102
103      ret
104
```

```
1  MERGE(A,i,k,j)
2      n1 := k-i+1
3      n2 := j-k
4      crea L[0..n1] e R[0..n2]
5      for t = 0 to n1-1
6          L[t] := A[i+t]
7      for t = 0 to n2-1
8          R[t] := A[k+1+t]
9      L[n1] := ∞
10     R[n2] := ∞
11     l := 0
12     r := 0
13     for t = i to j
14         if L[l] ≤ R[r]
15             then
16                 A[t] := L[l]
17                 l := l + 1
18             else
19                 A[t] := R[r]
20                 r := r + 1
```



Modus operandi

- L'approccio più semplice e intuitivo è stato scrivere il codice assembly «traducendolo» da codice scritto ad alto livello (pseudocodifica)
- Questo, unito alla semplicità dell'ISA RISC-V ha semplificato molte cose ma in alcuni casi è stato necessario aguzzarsi di più di così
- Nella funzione `shift()` ad esempio non sono stati creati array di supporto per andare ad ordinare le sottosequenze

Debug e test

- **Compilazione:** spostiamoci nella cartella contenente i sorgenti e eseguiamo nel terminale `./compile.sh`
- **Avviamo il programma** lanciando il comando `qemu-riscv64 -g 2233 mergesort`
- In un altro terminale avviamo il **debug** tramite il comando `riscv64-unknown-elf-gdb -ex "tar rem:2233" mergesort`
- Per verificare il **funzionamento** verifichiamo lo stato dell'array prima del termine del programma lanciando nel terminale gdb il comando `x/5xb &testArray`

```
calcolatori@calcolatori-VirtualBox: /media/sf_riscv-mergesort 7
(gdb)
shift_skip () at mergesort.s:78
78          addi s0, s0, 1          # Increment first half
(gdb)
79          ld a2, 24(sp)          # Load back last element
(gdb)
81          bge s0, a2, merge_loop_end
(gdb)
merge_loop_end () at mergesort.s:107
107         ld ra, 0(sp)
(gdb)
108         addi sp, sp, 32
(gdb)
merge_loop_end () at mergesort.s:109
109         ret
(gdb)
mergesort_end () at mergesort.s:40
40         addi sp, sp, 32
(gdb)
mergesort_end () at mergesort.s:41
41         ret
(gdb)
_start () at main.s:23
23         ecall
(gdb) x/5xb &testArray
0x11281:      0x02      0x03      0x05      0x08      0x14
(gdb)
```


Domande?

Grazie per l'attenzione