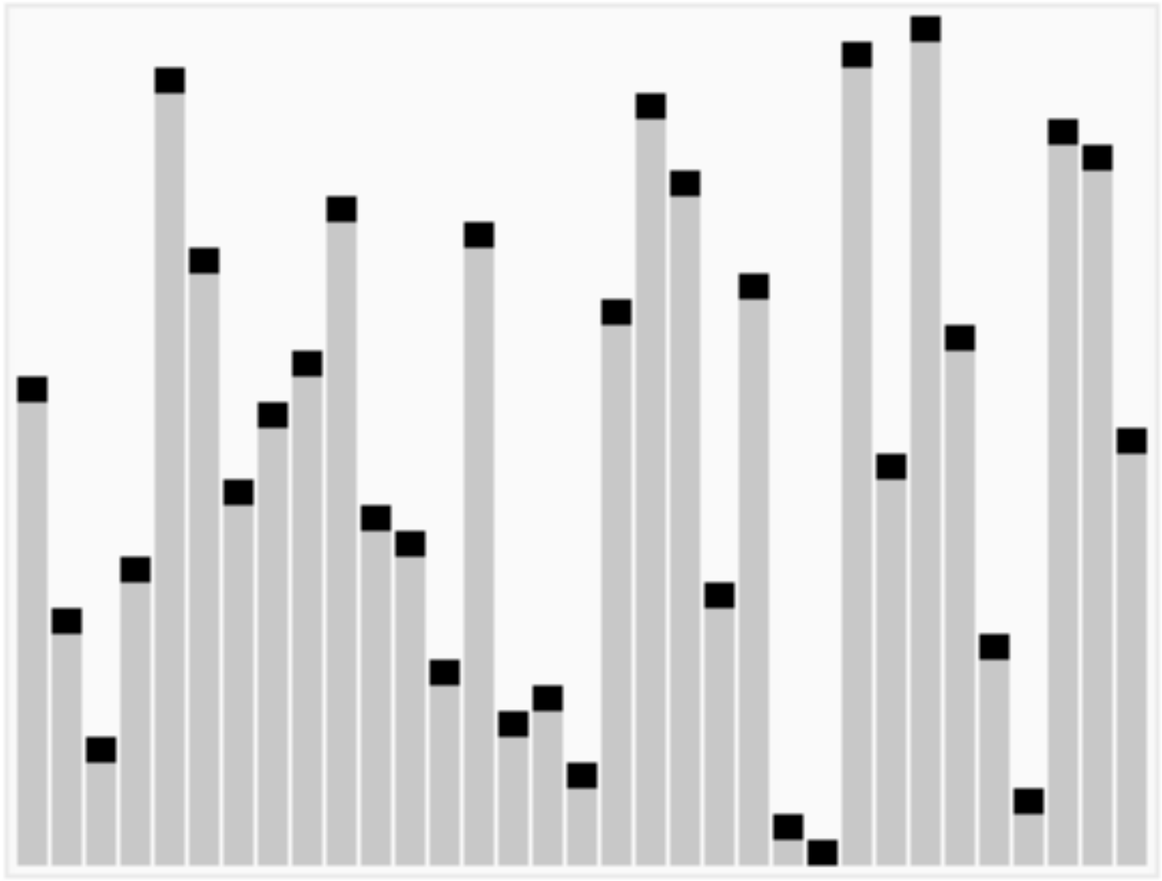
Quicksort in RISC-V Assembly

Marte Montipo' - 18/07/2019

Cos'è il Quicksort?

- È un algoritmo di ordinamento ricorsivo, basato sulla tecnica del divide-et-impera.
- Funziona mediante l'uso di un "pivot" che divide di volta in volta ogni parte dell'array in due parti (el. più piccoli e el. più grandi).
  - In questa implementazione, il pivot è sempre l'elemento più grande
- Lower bound: $o(n \log n)$
  Upper bound: $O(n^2)$
  Average: $O(n \log n)$



Che funzioni vanno implementate?

`quicksort`

- Questa funzione è quella che viene chiamata dal programma.
- Chiama `partition()` usando i bounds che vengono passati alla funzione, e si salva il pivot che partition restituisce;
- Chiama poi se stessa ricorsivamente due volte:
  - Una sull'insieme dei valori tra lower bound e pivot;
  - Una sull'insieme dei valori tra pivot e upper bound;

`partition`

- Questa funzione viene chiamata esclusivamente da `quicksort()`
- Si occupa di dividere una sezione dell'array in due parti, una con gli elementi più grandi del pivot, una con gli elementi più piccoli;
- Modifica l'array in place e ritorna la posizione finale del pivot.

quicksort()

```
quicksort:
    bltu a3, a2, quicksort_exit      # if (lo >= hi) we just return

    # save stuff in the stack
    addi sp, sp, -32
    sd ra, 0(sp)
    sd s10, 8(sp)       # s10 is going to hold lo
    sd s11, 16(sp)      # s11 is going to hold hi
    sd s9, 24(sp)       # s9 is going to hold the pivot

    # hold lo and hi
    mv s10, a2          # s10 <- lo
    mv s11, a3          # s11 <- hi

    # call partition
    jal ra, partition

    # save the pivot on s9
    mv s9, a0

    # s9 = pivot
    # s10 = lo
    # s11 = hi
    # recursively call quicksort on both subarrays
    addi a3, s9, -1         # hi = pivot (-1)
    mv a2, s10              # lo = lo
    jal ra, quicksort       # quicksort(a1, lo, pivot-1)

    addi a2, s9, 1          # lo = pivot (+1)
    mv a3, s11              # hi = hi
    jal ra, quicksort       # quicksort(a1, pivot+1, hi)

    # load stuff back from the stack
    ld ra, 0(sp)
    ld s10, 8(sp)
    ld s11, 16(sp)
    ld s9, 24(sp)
    addi sp, sp, 32
quicksort_exit:
    ret
```

```
def quickSort(arr,low,high):
    if low < high:
        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr,low,high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)
```

partition()

```
partition:
    # save stuff in the stack
    addi sp, sp, -24
    sd ra, 0(sp)
    sd s10, 8(sp)
    sd s11, 16(sp)

    # init pivot to high (a3)
    add t0, a1, a3
    lbu t0, 0(t0)

    addi t2, a2, -1     # (i) index of the smaller element => t2 = low - 1
    mv t6, a2           # t6 = j = low
    addi t5, a3, -1     # t5 = high-1

    partition_forloop:
    bgt t6, t5, partition_forloop_end    # if t6 > t5 then partition_forloop_end
        add s11, a1, t6         # s11 = *arr[j]
        lbu t1, 0(s11)           # t1 = *(arr[j])

        bgtu t1, t0, partition_forloop_inner_skip    # if t1>t0 skip (if arr[j]>pivot)
            addi t2, t2, 1      # i++
            add s10, a1, t2     # s10 = *arr[t2] = *arr[i]
            lbu t3, 0(s10)       # t3 = *(arr[i])
            sb t3, 0(s11)       # arr[j] = t3
            sb t1, 0(s10)       # arr[i] = t1
        partition_forloop_inner_skip:

        addi t6, t6, 1  # j++
    j partition_forloop
    partition_forloop_end:

    addi a0, t2, 1      # write return value as i+1

    # swap(&arr[i+1], &arr[high])
    add s10, a1, a0         # s10 = *arr[i+1]
    add s11, a1, a3         # s11 = *arr[high]
    lbu t2, 0(s10)           # t2 = *s10
    lbu t3, 0(s11)           # t3 = *s11
    sb t2, 0(s11)
    sb t3, 0(s10)

    # load stuff back from the stack
    ld ra, 0(sp)
    ld s10, 8(sp)
    ld s11, 16(sp)
    addi sp, sp, 24
partition_bail:
    ret
```

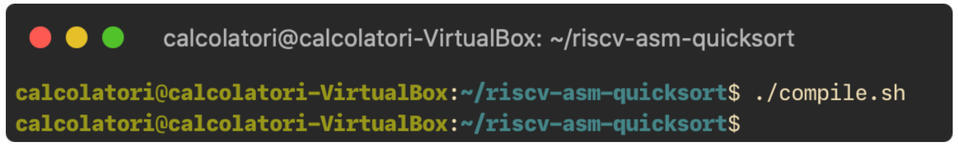
```
def partition(arr,low,high):
    i = (low-1)          # index of smaller element
    pivot = arr[high]      # pivot

    for j in range(low, high):
        # If current element is smaller than or
        # equal to pivot
        if arr[j] <= pivot:

            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[high] = arr[high],arr[i+1]
    return(i+1)
```

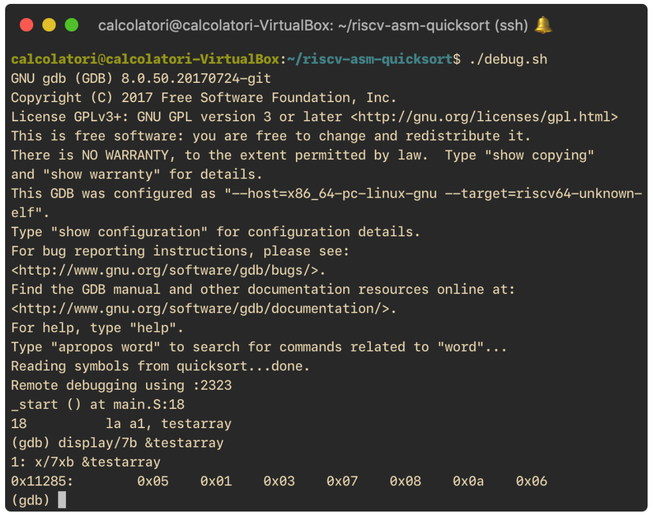
Debug e verifica delle funzionalità

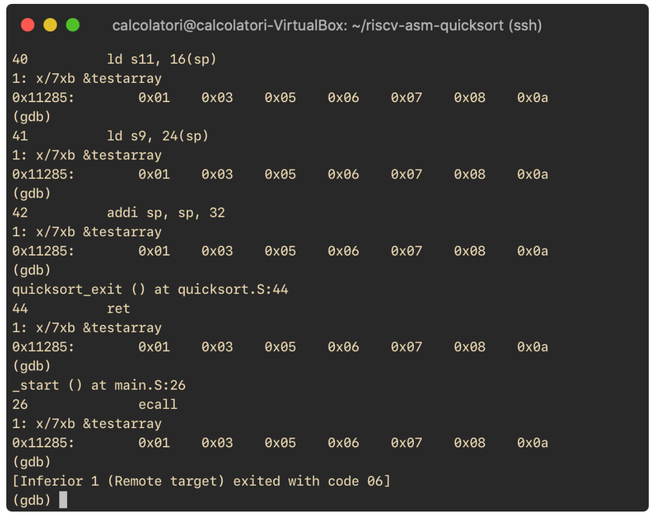
- **Compilazione**
  È necessario eseguire **`./compile.sh`**

- **Esecuzione**
  È obbligatorio eseguirlo con il debugger attivo (altrimenti non sarà possibile vedere il risultato) scrivendo **`qemu-riscv64 -g 2323 ./quicksort`**

- **Debug**
  Per debuggare basterà avviare **`./debug.sh`**;
  Basterà poi scrivere **`display/7b &nomevar`**; per visualizzare **`&nomevar`** a ogni step.
  Ad esempio: **`display/7b &testarray`**







Take-away principale

L'ISA RISC-V è semplice. Questo la rende molto veloce e permette ai compilatori di ottimizzare particolarmente il codice. Se però si vuole scrivere assembly direttamente, diventa necessario porre particolare attenzione a ogni istruzione che si scrive, sia per scrivere il proprio codice in modo che "sfrutti" la pipeline, sia per evitare che vi siano problemi logici.

Leggere e "tradurre" codice di riferimento scritto in un linguaggio di programmazione di alto livello semplifica decisamente questo compito.

Grazie per l'attenzione