



PROGETTO 03

ARRAY QUICK SORT



OBIETTIVO

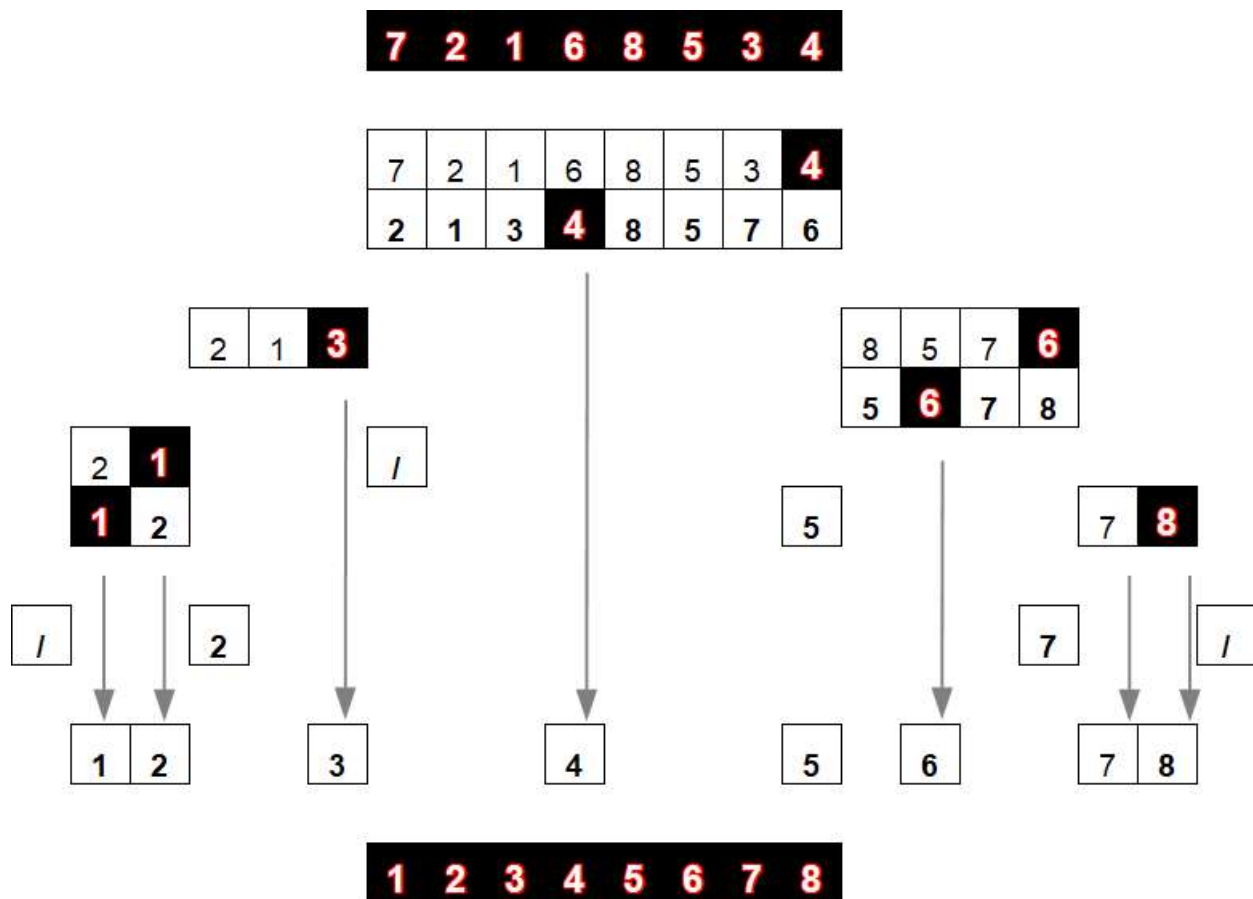
Consegna:

realizzare un programma assembly RISC-V in grado di riordinare un array di dati di tipo double word sfruttando l'algoritmo di ordinamento ricorsivo quick sort.

DESCRIZIONE

Il programma presentato è suddiviso su più file per rendere maggiormente chiaro il codice prodotto e per poter essere trasportabile su ogni altro file, potrà infatti essere comodamente importato e quindi incluso nel file permettendo all'utente di accedere alle proprie funzioni. Il programma riceve in ingresso, tramite una funzione personalizzata di scan terminale, 8 dati di tipo double word e tramite l'algoritmo di quick sort stampa con una print formattata personale, l'array ordinato. Il programma scritto in linguaggio assembly RISC-V dovrà essere compilato ed eseguito tramite l'emulatore qemu su terminale.

L'algoritmo di quick sort è un algoritmo capace di ordinare un array, in un caso medio, in $O(n \log n)$. Il quick sort è un algoritmo di ordinamento ricorsivo che rientra nella categoria di divide-and-conquer per la propria risoluzione, inoltre, l'ordinamento è di tipo place-in, ovvero non richiede memoria extra nella sua risoluzione perché tutte le operazioni sono effettuate direttamente sull'array originale con solo l'utilizzo di memoria per alcune variabili temporanee. L'algoritmo, su un vettore in ingresso, richiede di scegliere un elemento su cui basare le varie comparazioni chiamato pivot (spesso ultimo elemento dell'array), e porre alla sua sinistra tutti gli elementi minori, mentre alla sua destra saranno situati tutti i restanti numeri maggiori. Questo processo di ri-arrangiamento di un vettore è definito come partizione, l'array risulta quasi spaccato in parti secondo il pivot. Una volta ottenuta la partizione del vettore, l'unico valore dell'array al posto esatto sarà il pivot, si otterranno infatti, rispetto al pivot, due nuovi sotto-array non necessariamente ordinati. Per risolvere dunque il loro ordinamento basterà riapplicare (concetto di ricorsione) l'algoritmo di partizione ad ognuno dei sotto-array, continuando così ad ordinare l'intero vettore ad ogni richiamo della funzione. Il processo ricorsivo di partizionamento dovrà terminare solo quando il sotto-array da ordinare sarà per definizione già ordinato, dunque quando il sotto-array avrà 0 od 1 elementi. A questo punto ogni funzione chiamata ritornerà alla sua chiamante e quindi l'algoritmo di quick sort si chiuderà avendo già ordinato il vettore.



SVILUPPO

Analisi algoritmi:

```
# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low --> Starting index,
# high --> Ending index
```

```
# Function to do Quick sort
```

```
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)
```

```
# Driver code to test above
```

```
arr = [10, 7, 8, 9, 1, 5]
```

```
n = len(arr)
```

```
quickSort(arr, 0, n-1)
```

```
print ("Sorted array is:")
```

```
for i in range(n):
```

```
    print ("%d" %arr[i]),
```

```
# Python program for Quicksort
```

```
# This function takes last element as pivot, places
# the pivot element at its correct position in sorted
# array, and places all smaller (smaller than pivot)
# to left of pivot and all greater elements to right
# of pivot
```

```
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 )
```

Quick Sort:

l'algoritmo di quick sort si compone di 4 parti fondamentali:

- condizione di uscita dal processo ricorsivo tramite il controllo della lunghezza del vettore passato come parametro in ingresso. Vengono confrontati gli indici di start e di end dell'array, se lo start è minore dell'end allora la condizione è corretta poiché il vettore avrebbe almeno due elementi;
- partizione del vettore rispetto all'elemento pivot (scelto come elemento di indice end), gli elementi minori del pivot sono posti a sinistra, mentre quelli maggiori a destra. La funzione ritorna il valore dell'indice associato al pivot (pIndex), questo infatti sarà l'unico elemento ordinato all'interno del vettore totale;
- chiamata di tipo ricorsivo alla procedura di quick sort sul sotto-array di valori minori, alla funzione vengono passati pertanto gli indici start e pIndex - 1.
- chiamata di tipo ricorsivo alla procedura di quick sort sul sotto-array di valori maggiori, alla funzione vengono passati pertanto gli indici pIndex + 1 ed end.

Partition:

la funzione di partizione del vettore accetta in ingresso come argomenti il vettore da ordinare e gli indici di start ed end. La funzione utilizza come variabili temporanee il pivot, associato come l'elemento di indice end dell'array, e un indice pIndex per memorizzare la posizione corrente del pivot. Con un ciclo for vengono scansionati tutti i valori del vettore tranne l'ultimo, poiché considerato come pivot. All'interno del ciclo se l'elemento i-esimo risulta minore o uguale al pivot, viene effettuato uno swap (scambio), tra il valore alla posizione i-esima e quello alla posizione pIndex-esima. Successivamente il valore del pIndex viene incrementato e si passa alla iterazione seguente. Tramite questo codice gli elementi minori del pivot vengono pertanto spostati alla sinistra del pIndex. Una volta completato il ciclo for si ottiene per cui un vettore circa ordinato rispetto al pivot, il quale però, si trova ancora alla sua posizione di origine end, bisognerà per cui effettuare un ultimo swap tra il valore alla posizione end-esima (pivot) e quello alla posizione pIndex-esima. Da cui la partizione completa, non resta altro che uscire ritornando come valore il pIndex che identifica la corretta posizione del pivot all'interno del vettore originale.

Quick sort (ricorsione sotto-array sinistro):

terminata l'esecuzione della partizione ad una variabile temporanea bisogna associarne il valore di ritorno. Da cui è possibile richiamare, tramite la tecnica della ricorsione, la funzione di quick sort sulla partizione sinistra del pivot, passando in questo caso come indici start e pIndex - 1. Il processo verrà dunque eseguito completamente sul vettore limitato dai due indici e si fermerà solo alla condizione di uscita dell'if.

Quick sort (ricorsione sotto-array destro):

terminata l'esecuzione dell'ordinamento sinistro, da cui è possibile richiamare, tramite la tecnica della ricorsione, la funzione di quick sort sulla partizione destra del pivot, passando in questo caso come indici pIndex + 1 ed end. Il processo verrà dunque eseguito completamente sul vettore limitato dai due indici e si fermerà solo alla condizione di uscita dell'if.

REALIZZAZIONE

```
ASM ts_quickSort.s X ASM tsQuickSort.s ASM tsScan.s ASM tsPrint.s
c: > Users > Semmy17 > Desktop > Università > Architettura Calcolatori > Progetti Esame > Prog_003 > ASM ts_quickSort.s
1  #-----PROGRAM TO ORDER WITH QUICK SORT ALGORITHM, NUMBERS FROM TERMINAL MONITOR-----#
2
3  #-----NUMBER AND MNEMONIC NAME REGISTER RISC-V-----#
4  # x0 -x4 # zero, ra, sp, gp, tp
5  # x5 -x7 # t0, t1, t2
6  # x8 -x9 # fp/s0, s1
7  # x10-x17 # a0, a1, a2, a3, a4, a5, a6, a7
8  # x18-x27 # s2, s3, s4, s5, s6, s7, s8, s9, s10, s11
9  # x28-x31 # t3, t4, t5, t6
10
11 #-----MODULE-----#
12 .include "tsScan.s"
13 .include "tsPrint.s"
14 .include "tsQuickSort.s"
15
16 #-----constant-----#
17 .global _start
18     # system code interrupt
19     .equ _SYS_EX, 93
20     # constant
21
22 #-----global symbol-----#
23 .section .data
24     vector: .space 64
25
26 .section .rodata
27     # message for terminal
28     inVect: .string "Insert 8 numbers:\n"
29     outVect: .string "Quick sort on vector:\n"
```

Il programma assembly RISC-V si compone di 4 files per rendere il codice chiaro e trasportabile, le funzioni importate infatti nel programma principale sono funzioni usate e compatibili con altri programmi assembly scritti.

Il codice riportato compone l'header del codice, sono infatti inclusi i file di appoggio per le funzioni necessarie da importare, la dichiarazione di costanti (interpretati dal programma come immediati), le dichiarazioni di dati globali.

La sezione data raccoglie infatti i dati disponibili per lettura e scrittura all'interno del programma, il dato vector indica infatti uno spazio in memoria di 64 celle, ovvero 64 byte, che nel caso specifico saranno assegnati ad un vettore di 8 dati di tipo double word. Al contrario la sezione rodata rende disponibili dati leggibili ma non scrivibili (section Read Only Data), sono infatti inserite alcuni dati di tipo string che raccolgono messaggi da mostrare su terminale per "comunicare" con l'utente.

```

31 #-----code-----#
32 .section .text
33
34 #-----main program-----#
35 _start:
36     # call to print function for message
37     la a1, inVect
38     li a2, 18
39     jal print
40
41     # scan number input and allocate in data vector
42     la s1, vector      # address of vector
43     #-----forscan-----#
44     li s0, 0           # index for cycle
45     forscan:
46         # user code
47         # prepare calling to function
48         mv a0, s1
49         jal scanSignedNumbersTilCarriage
50
51         addi s1, s1, 8 # increment byte pointer
52     # condition exit
53     li t0, 8
54     addi s0, s0, 1
55     blt s0, t0, forscan
56
57     jal printCarriage # print one line empty
60
61     # call to quick sort function to order vector
62     la a0, vector
63     li a1, 0
64     li a2, 7
65     jal quickSort
66
67     # call to print function for message
68     la a1, outVect
69     li a2, 22
70     jal print
71
72     # print signed numbers and carriage function
73     la s1, vector      # address of data vector
74     #-----forprint-----#
75     li s0, 0           # index for cycle
76     forprint:
77         # user code
78         # prepare calling to function
79         lb a0, 0(s1)
80         jal ra, printSignedNumbersAndCarriage
81
82         addi s1, s1, 8 # increment double word pointer
83     # condition exit
84     li t0, 8
85     addi s0, s0, 1
86     blt s0, t0, forprint
87
88     _end:
89     li a7, _SYS_EX
90     ecall
91

```

_start:

identificato come il programma principale di ogni programma assembly RISC-V, la label `_start` identifica l'inizio del classico main all'interno della sezione `text`. Il main in sostanza svolge il compito di funzione principale e chiamante di tutte le altre funzioni del programma per gestirne l'esecuzione ed interfacciamento. All'interno del suo corpo sono riconoscibili i richiami alle funzioni di “scan”, “print” e `quickSort`. Mentre le prime due sono state approfondite e trattate in un'altra relazione, la funzione di `quickSort` rimane la funzione chiave per il progetto (basti sapere che “scan” e “print” lavorano in maniera simile, passando come argomento un dato od indirizzo; scansionano da terminale o stampano uno alla volta il dato in formato numerico). In ingresso alla funzione vengono infatti passati tre argomenti, il puntatore al dato `vector`, indice di inizio ed indice di fine. Il primo argomento identifica l'indirizzo in memoria del vettore composto da 8 dati di tipo double word. Il secondo e terzo argomento indicano rispettivamente, l'indice da cui partire per riordinare e l'indice a cui fermarsi durante l'ordinamento.

Per quanto invece concerne il richiamo delle funzioni di `scan` e `print`, i metodi sono molto simili. Viene infatti definita una struttura `for` per elaborare gli 8 dati da inserire o scrivere su terminale, ed il ciclo itera su un registro di tipo “saved” di appoggio poiché le funzioni chiamate “scan” o “print”, sovrascriverebbe un normale registro “temporary”. Ad ogni iterazione, rispettivamente, alla funzione di “scan” viene passato un indirizzo di locazione della memoria su cui salvare il dato letto, mentre alla funzione “print” un valore da stampare su terminale. I richiami aggiuntivi alla funzione `print` fuori dalla struttura dei cicli `for`, interessano la scrittura su monitor di messaggi di tipo stringa e quindi semplici notazioni riportate dal programma per l'utente.

```

1  #-----TS QUICK SORT-----#
2
3  #-----quick sort vector n element function-----#
4  # the function uses:
5  #   - argument register:
6  #     - a0: vector pointer
7  #     - a1: start index vector
8  #     - a2: end index vector
9  #   - temporary register t0, t1, t2, t3, t4, t5, t6
10 #   - saved register s0, s1, restored before main call
11
12 # function quick sort on array
13 quickSort:
14     # condition to end recursion call
15     bge a1, a2, pass
16     # store on stack return address, and saved register
17     addi sp, sp, -24
18     sd ra, 0(sp)
19     sd s0, 8(sp)
20     sd s1, 16(sp)
21
22     # saved value register a0, a2
23     mv s0, a0
24     mv s1, a2
25
26     jal partition      # call to partition function
27     mv t6, a0          # copy return register into app
28
29     mv a0, s0          # restore a0 argument
30     addi a2, t6, -1    # change end index
31     jal quickSort      # call to partition function
32
33     mv a0, s0          # restore a0 argument
34     addi a1, t6, 1     # change start index
35     mv a2, s1          # restore a2 argument
36     jal quickSort      # call to partition function
37
38     # load from stack return address, and saved register
39     ld ra, 0(sp)
40     ld s0, 8(sp)
41     ld s1, 16(sp)
42     addi sp, sp, 24
43
44     pass:
45     jr ra

```

tsQuickSort.s:

identificato come il file per l'implementazione dell'ordinamento tramite quick sort, il codice è suddiviso in due funzioni ovvero quickSort e partition.

Come visto in precedenza l'algoritmo di quick sort si basa sulla chiamata ricorsiva passando come argomenti gli indici calcolati tramite una funzione di partizionamento del vettore preso in considerazione.

La prima funzione è dunque quella di quick sort. Il corpo della funzione è abbastanza lineare, subito vengono infatti salvati sullo stack il registro ra (return address) ed i registri s0 ed s1. Il primo registro contiene il valore del program counter di ritorno, ovvero l'indirizzo dell'istruzione memory al quale la funzione farà riferimento per tornare alla funzione chiamata. Questo indirizzo viene calcolato al momento della chiamata della funzione di quick sort e viene calcolato come l'indirizzo dell'istruzione di chiamata più 4 (PC + 4), ovvero i 4 byte che identificano la prossima istruzione (in RISC-V ogni istruzione è codificata a 4 byte, 32 bit). Il registro return address deve essere salvato poiché la funzione quick Sort non essendo necessariamente una leaf-procedure, ovvero una funzione che non richiama alcuna altra funzione al suo interno, ad ogni salto ad un'altra funzione andrebbe a sovrascrivere il return address corrente, non riuscendo per cui a ritornare alla chiamata di origine.

Gli altri due registri “saved” s0 ed s1, devono essere memorizzati poiché, volendo rispettare la convenzione RISC-V riguardo i registri, tutti i registri “saved” non possono essere modificati da alcuna funzione chiamata. Nel caso di un loro necessario utilizzo quindi, devono essere prima salvati e poi ripristinati prima del ritorno alla funzione chiamante. Questi registri servono solo d'appoggio agli indici passati agli argomenti a0, a1, a2. Questi tre dati essendo argomenti di entrata per la funzione di partizione, vengono prima utilizzati e poi, solo il registro a0, sovrascritto con il valore di ritorno della funzione. I registri “saved” rimanendo intatti ad ogni eventuale chiamata preservano il contenuto passato precedentemente ed inoltre sono più veloci rispetto ad un salvataggio sullo stack. Nel dettaglio, i registri a0, a2 essendo utili sia alla prima chiamata ricorsiva quickSort che alla seconda, vengono modificati perdendo il loro valore originale, la prima chiamata verrebbe dunque processata correttamente mentre la seconda si baserebbe su dati sfalsati. Bisogna ripristinare dunque i registri a0 ed a2 prima del loro nuovo utilizzo. Come visto prima si fa affidamento ai registri “saved”. Si potrebbe infatti, salvare tutto il necessario sullo stack, questo vorrebbe dire però che ad ogni utilizzo dei dati bisognerebbe accedere allo stack in lettura o scrittura, sprecando pertanto molti cicli per accedere in memoria. Utilizzando però i “saved register” questo accesso allo stack si limita all'inizio ed alla fine della funzione, una volta in scrittura ed una in lettura per il ripristino, si risparmia qualche istruzione ed accesso aggiuntivo in memoria. La condizione if start < end è contenuta nell'istruzione di branch bge (branch greater equal). L'istruzione lavora in modo opposto, infatti by-passa il codice interno solo se la condizione è vera, svolge quindi il codice solo se l'indice end è ancora strettamente maggiore dell'indice start.

```

47 # function partition on array
48 partition:
49     slli t0, a1, 3
50     add t0, t0, a0 # pointer index
51     mv t1, t0      # partition index
52
53     addi t5, a2, -1 # end index -1
54     slli t5, t5, 3
55     add t5, t5, a0 # length pointer vector
56
57     ld t2, 8(t5)   # pivot
58
59     #-----forpartition-----#
60     forPartition:
61         # user code
62         ld t3, 0(t0) # element [*]
63         bgt t3, t2, next
64         # swap
65         ld t4, 0(t1) # app = [partition index]
66         sd t3, 0(t1) # [partition index] = element [*]
67         sd t4, 0(t0) # [pointer index] = app
68         addi t1, t1, 8
69     next:
70     # condition exit
71     addi t0, t0, 8 # increment dword pointer
72     ble t0, t5, forPartition
73
74     # swap
75     ld t4, 0(t1) # app = [partition index]
76     sd t2, 0(t1) # [partition index] = pivot
77     sd t4, 8(t5) # [pivot pointer index] = app
78
79     sub t1, t1, a0 # partition index - base pointer = offset pointer
80     srli a0, t1, 3 # partition index / 8 = return index
81
82     jr ra
83

```

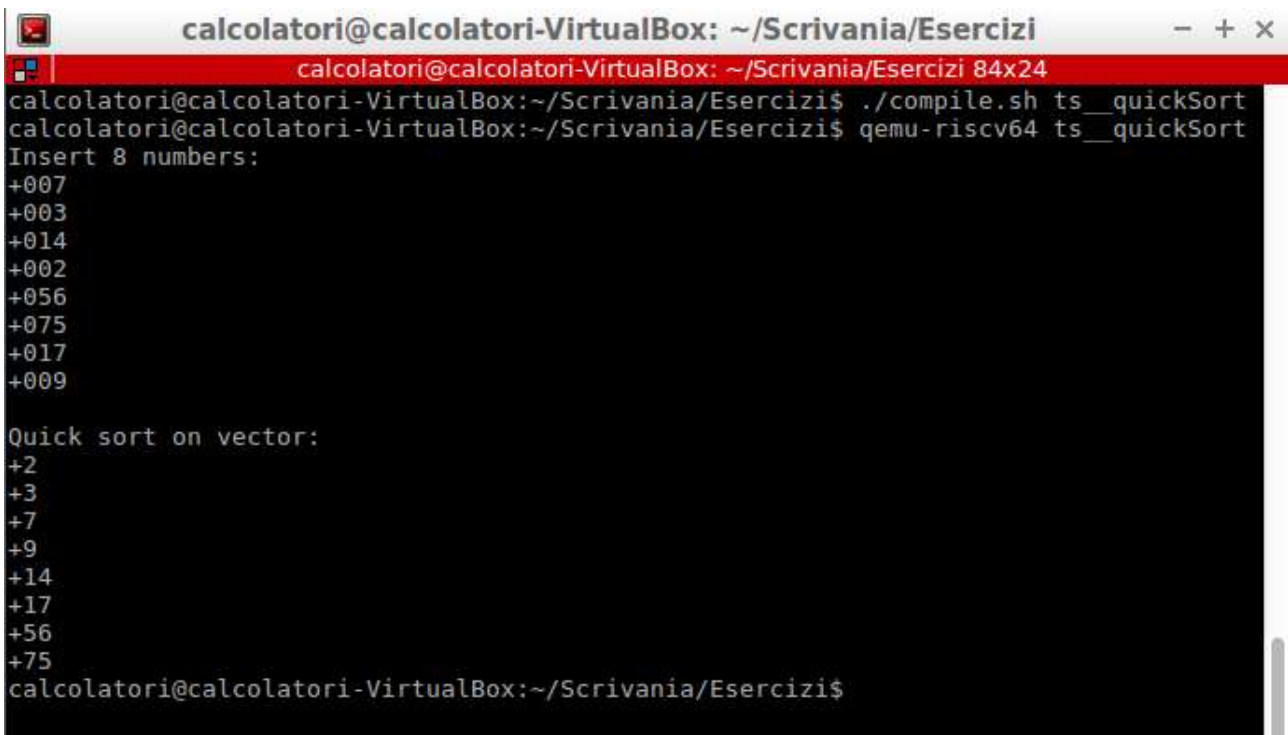
Passando alla funzione partition, da subito si nota come i registri argomento siano i medesimi della funzione quick sort. La funzione infatti basa il proprio codice sui dati del

vettore ristretto dagli indici di inizio e fine. La funzione è stata ottimizzata, gestendo infatti il vettore tramite puntatori piuttosto che indici. Questo metodo è possibile poiché, tramite l'indice di end, si conosce la lunghezza del vettore da prendere in considerazione e tutti i cicli si basano su un confronto sulla lunghezza piuttosto che un confronto sull'indice massimo raggiunto. Questo metodo riesce infatti a limitare il numero di istruzioni contenuto nel corpo del ciclo for poiché l'“indice” del for è già un offset adeguato al tipo di elementi del vettore e quindi non necessita calcoli aggiuntivi per accedere al corretto i-esimo elemento del vettore. Nella funzione le prime istruzioni calcolano infatti l'indirizzo del primo elemento del vettore tramite uno shift left 3 al quale viene sommato l'indirizzo base del vettore (shiftare a sinistra di 3 posizioni in binario è come moltiplicare per 8, 8 è corretto poiché si sta trattando un'array con dati di tipo double word). Quest'operazione è associata prima a t0, che sarà il puntatore indice del ciclo for, poi a t1, indice che sarà ritornato come partition index (pIndex). Le successive tre righe di codice calcolano la lunghezza del vettore meno l'ultimo elemento (elemento end-esimo è posto come pivot e non considerato nel ciclo), viene poi applicata la moltiplicazione per 8 (effettuare uno shift left è molto più veloce di moltiplicare, slli = 1 cpi, mul = 64 cpi), sommato infine l'indirizzo base per calcolare l'indirizzo fine - 1 del vettore in memoria. Ultimato questo processo il registro t5 avrà il risultato dell'operazione, manca solo associare l'elemento end-esimo al registro t2 (pivot), tramite un load double carichiamo il contenuto dell'indirizzo del registro t5 con offset 8, ovvero elemento successivo in termini di double word in memoria.

Da questo punto una label sancisce l'inizio del ciclo for. Serve dunque confrontare l'elemento i-esimo del vettore con il pivot, si carica tramite un ld il contenuto della locazione di memoria associata all'indirizzo t0 (notare come il processo sia diretto lavorando a puntatori e non indici), e si effettua il confronto tramite un branch greater than. Sempre adottando la logica di prima, il codice interno al branch è ciclato solo se la condizione falsa, dunque solo se il pivot è minore od uguale dell'elemento i-esimo. In questo caso si svolge per cui lo swap tra i due elementi, appoggiandosi al registro t4 per prima memorizzare il contenuto alla pIndex-esima posizione, caricare il valore dell'elemento i-esimo alla pIndex-esima posizione ed infine utilizzare t4 per recuperare e caricare l'elemento pIndex-esima posizione alla i-esima posizione. Effettuato lo scambio tra i dati interessati bisogna aggiornare lo stato del pIndex incrementandolo di un'unità. Da cui si procede con l'operazione di confronto per uscire dal ciclo for, sommando 8 all'indirizzo salvato in t0 si accede all'elemento successivo di un array di elementi di tipo double word. Ripetuto il ciclo n volte bisogna infine effettuare lo swap tra l'elemento end-esimo (pivot) e l'elemento pIndex-esimo. La funzione, prima di uscire, ritorna come valore l'indice pIndex per basare le future chiamate ricorsive del quick sort. Bisogna pertanto reperire l'indice dell'ormai pivot. Avendo acceduto agli elementi come puntatori il registro t1 non contiene un indice bensì un puntatore in memoria, bisogna dunque ricavare l'offset secondo il vettore in memoria, sottraendo pertanto l'indirizzo base del vettore, ed infine fare uno shift right sull'offset dividendo dunque per 3 lo stato del registro. Il tutto va salvato nel registro a0 per essere conforme alla convenzione RISC-V riguardo registri argomenti e ritorno per funzioni.

Nella funzione quick sort si procede con le chiamate ricorsive. Semplicemente stavolta bisogna solo curarsi di sottrarre ed aggiungere un'unità agli indici di end e di start per basare le operazioni delle chiamate ricorsive sui corretti sotto-array del vettore originale. Da notare l'utilizzo dei registri “saved” per appoggiarsi ai corretti indici d'origine. L'unico registro a non dover essere ripristinato è l'a1. Questo registro è infatti “sporcato” all'ultima chiamata ricorsiva dunque non influenza alcuna altra operazione, non è necessario preoccuparsi di memorizzarlo su stack o in altri registri.

TEST



```
calcolatori@calcolatori-VirtualBox: ~/Scrivania/Esercizi
calcolatori@calcolatori-VirtualBox: ~/Scrivania/Esercizi 84x24
calcolatori@calcolatori-VirtualBox:~/Scrivania/Esercizi$ ./compile.sh ts__quickSort
calcolatori@calcolatori-VirtualBox:~/Scrivania/Esercizi$ qemu-riscv64 ts__quickSort
Insert 8 numbers:
+007
+003
+014
+002
+056
+075
+017
+009

Quick sort on vector:
+2
+3
+7
+9
+14
+17
+56
+75
calcolatori@calcolatori-VirtualBox:~/Scrivania/Esercizi$
```

Impostazioni emulazione:

per realizzare un test pratico del programma scritto in assembly ci si avvale degli strumenti di compilazione ed emulazione installati su una macchina virtuale Linux, capace di simulare l'architettura RISC-V ed interpretare dunque il codice assemblato. I comandi da impartire tramite monitor terminale sono quelli di compilazione, utile per creare l'eseguibile del file basandosi sul codice scritto tramite Visual Studio Code, e lo strumento di esecuzione qemu in grado di emulare l'architettura RISC-V ed eseguire il file assemblato in precedenza. Non sarà necessario in questo caso appoggiarsi al sistema di debugging gdb poiché è possibile tramite terminale, ricorrendo alle funzioni di "scan" e "print", inserire e leggere i dati ciclanti dal programma. Senza questi strumenti sarebbe stato necessario l'appoggio del debugger per vedere in tempo reale lo stato dei registri e quello della memoria del programma.

Emulazione:

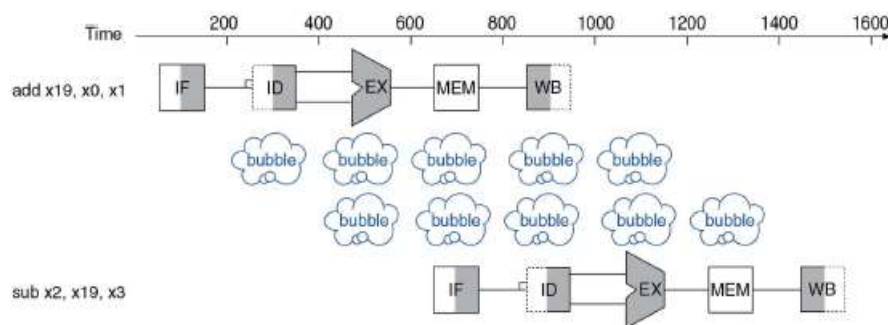
In fase di esecuzione il programma dunque presenta subito all'utente la richiesta di inserire 8 valori di tipo double word. La funzione "scan" permette l'input degli ingressi formattati però in segno e cifre lette (sxxx), riempiendo con degli 0 le cifre non significative. La funzione "scan" accetta in ingresso solo 3 cifre significative (settata per lettura di dati byte 0-255), il programma comunque, tramite il quick sort, elaborerà i dati byte come double word. Una volta inseriti gli 8 valori l'output mostrerà i dati ordinati secondo l'algoritmo.

OTTIMIZZAZIONI

Il programma presenta in totale due ottimizzazioni: la prima già spiegato legata al metodo di ciclo tramite puntatori, la seconda sulla funzione di swap (già implementata), ed una accessoria riguardo la gestione della sequenzialità delle istruzioni (non implementata in questo caso ma ugualmente approfondita).

Ricapitolando, la prima concerne dunque l'accesso agli elementi di un vettore tramite puntatori. Questo metodo permette di risparmiare istruzioni nel ciclo for per calcolare l'offset di un dato ponendo queste istruzioni prima dell'inizio del ciclo. Le istruzioni (spesso slli, add) sono ciclizzate pertanto solo una volta prima del for e non n volte al suo interno (la complessità / velocità $O(x)$ di un programma è infatti spesso calcolato sul numero di iterazioni dei cicli, la maggior parte di tempo di esecuzione un programma lo si impiega sui cicli, ottimizzare un'istruzione sola comporta un minimo vantaggio, ottimizzarla per n volte comporta un incremento delle prestazioni a lungo tempo).

La seconda ottimizzazione sta invece nel fatto di inglobare la funzione di swap (scambio posizione elementi) all'interno della partizione. Nel caso si dovesse chiamare una funzione esterna di swap, bisognerebbe nuovamente salvare sullo stack n registri per passare i corretti argomenti alla funzione e poi tornare all'esecuzione della funzione chiamante. Inglobando invece lo swap all'interno della partizione basta invece appoggiarsi ai registri attuali senza dover accedere ulteriormente alla memoria per operazioni accessorie. La scelta è stata presa poiché la funzione tsQuickSort.s lavora solo sull'algoritmo di quick sort, non sarebbe "logico" inserire un ulteriore richiamo ad una funzione di swap (si potrebbe creare una funzione esterna nuova swap.s). Inoltre un programma con meno basic blocks (blocchi di istruzioni sequenziali tra loro), sarà più veloce in fase di esecuzione poiché non dovrà "prevedere" salti e salvare eventualmente in memoria come nel caso specifico.



Ultima "ottimizzazione" potrebbe stare nel pipelining di alcune istruzioni della partition. Nelle prime righe di codice infatti la partition basa diverse istruzioni sul risultato di alcune istruzioni immediatamente precedenti. Questo tipo di procedimento porterebbe la CPU a creare tante linee di bubble quanti sono gli stage da aspettare perché il dato dell'istruzione antecedente sia aggiornato. Un'ottimizzazione che si potrebbe apportare sarebbe dunque quella di intervallare le varie istruzioni al fine che ad ogni passo non si debba aspettare il dato aggiornato ed ovviamente che le istruzioni siano slegate, al fine di ottenere il medesimo risultato di un programma non ottimizzato. Questo dettaglio è comunque già stato risolto grazie al supporto architetturale RISC-V, il contenuto del registro scritto nell'istruzione precedente è infatti riportato all'ingresso dell'ALU per l'istruzione successiva, evitando così la formazione di bubble all'interno della pipeline. Questo forwarding, tuttavia, non riesce ad ottimizzare istruzioni nel caso di load / use, in questo caso infatti bisognerebbe prevedere un back forwarding impossibile però a livello temporale. L'ottimizzazione proposta sarebbe dunque inefficace, poiché già ottimizzata nel caso specifico, ma efficace nel caso di una serie di istruzioni load / use.

CONCLUSIONI

Il progetto approfondito dimostra come a basso livello architettura e software siano strettamente collegati. In linguaggi assembly è infatti possibile apportare ottimizzazioni importanti anche solo evitando poche istruzioni all'interno di un ciclo o prevedere situazioni di disagio nel pipelining di alcune istruzioni.

Si dimostra infine, come comunque a fare la grande differenza, al di là di ottimizzazioni a livello architetturale e software, sia l'algoritmo proposto. In questo caso l'algoritmo di quick sort dimostra come lo stesso risultato può essere raggiunto tramite un programma nettamente superiore e migliore in termini di velocità e risorse (es. quick sort vs selection sort).