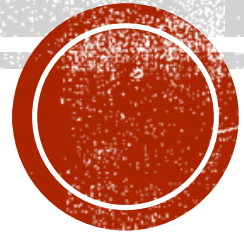
Radix Sort in RISC-V asm

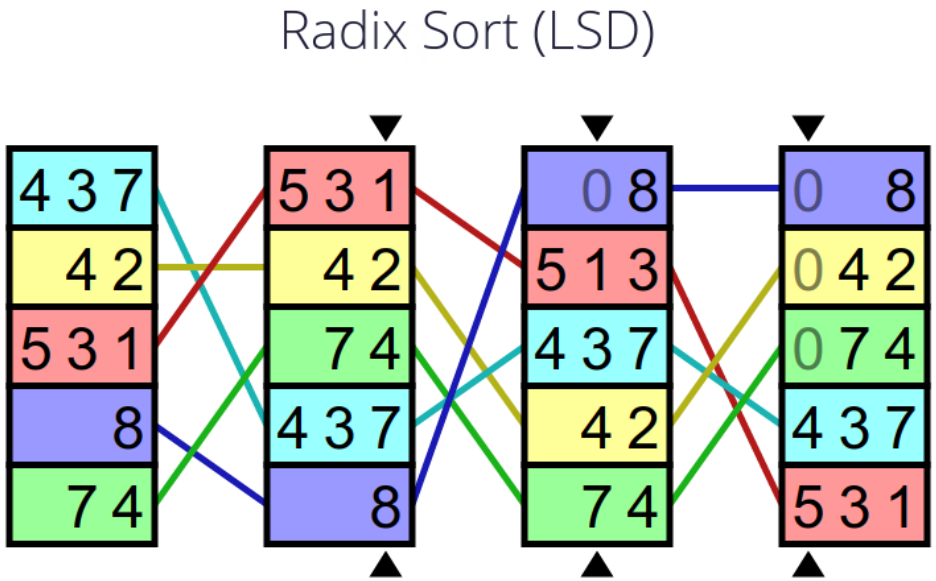
Blu Borghi – 30/8/2019



Radix Sort

Specifiche:

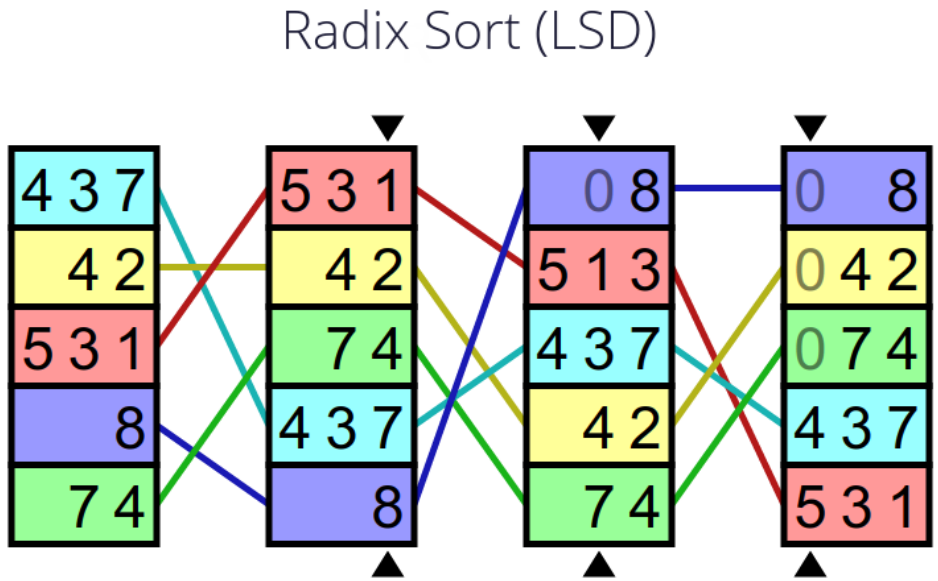
- Algoritmo di ordinamento non-comparativo, non è limitato al lower bound *o(n log n)*
- Lavora su numeri interi o naturali in una base qualsiasi, a seconda dell'implementazione
- Si appoggia a un algoritmo di ordinamento stabile, in questo caso al Counting Sort



Radix Sort

Funzionamento:

- Partendo dalla cifra meno significativa ordina gli elementi prendendo in considerazione uno spazio decimale per volta
- In caso di cife identiche mantiene l'ordine (relativo) del passaggio precedente (vedi nell'ultimo passaggio **0**08, **0**42, **0**74)
- Non effettua confronti per decidere in che ordine sistemare i numeri



Algoritmi implementati

- Radix Sort
  - Funzione richiamata dal programma principale
  - Utilizza la funzione Max per trovare il numero maggiore
  - Calcola il numero di cifre del numero maggiore, ovvero il numero di passaggi da effettuare
  - Per ogni passaggio richiama il Counting Sort selettivo su uno spazio decimale diverso
- Counting Sort selettivo
  - Ordina l'array che gli viene passato considerando una sola cifra decisa dal chiamante
  - Gestisce le cifre uguali mantenendo l'ordine relativo preesistente tra i due elementi
- Max
  - Restituisce il numero massimo contenuto dell'array passato come parametro

Costo Computazionale

- Calcolo del valore massimo nell'array, costo O(n)
- Counting Sort, costo O(n+k), k=10 (sempre costante), supponendo che n>=10 possiamo approssimare a O(n)
- Radix Sort, ripete il Counting Sort d volte, dove d è il numero di cifre del valore massimo dell'array (d = | log10(max) | +1)
- Costo complessivo O(n + d*n), ma d ≥ 1 quindi il costo diventa O(d*n)
- Per competere con un algoritmo comparative con O(nlogn), la quantità di cifre del numero più alto deve essere al massimo log2(n)
  - ES: n = 1'000 log2(1'000) = 9,9 d ≤ 10 max ≤ 1'000'000'000
- Costo in termini di memoria: O(n)

radixsort()

C++ algorithm_cpp/radix.cpp

```cpp
void radixsort(int arr[], int n)
{
    // Find the maximum number
    // to know number of digits
    int max = getMax(arr, n);

    // Do counting sort for every digit.
    // Note that instead of passing digit
    // number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; max >= exp; exp *= 10)
        countSort(arr, n, exp);
}
```

RISC-V src/radixsort.s

```
radixsort:  #a2=n #a3=input array address
    addi sp,sp,-8
    sd ra,0(sp)
    jal ra,max
    ld ra,0(sp)
    addi sp,sp,+8

    addi sp,sp,-16
    sd s0,0(sp)
    sd s1,8(sp)

    #for (int exp = 1; max >= exp; exp *= 10)
    #   countSort(arr, n, exp);
    addi s0,a0,0    #s0 = max
    li s1,1         #s1 = exp
loop2:
    addi a4, s1, 0 #passing exp to countingsort

    addi sp,sp,-8
    sd ra,0(sp)
    jal ra,countingsort
    ld ra,0(sp)
    addi sp,sp,+8

    li t0,10
    mul s1,s1,t0
    bge s0,s1,loop2
endloop2:
    ld s0,0(sp)
    ld s1,8(sp)
    addi sp,sp,16
ret
```

countingsort() [1/3]

C++ algorithm_cpp/radix.cpp

```cpp
void countSort(int arr[], int n, int exp)
{
    int output[n]; // output array
    int i, count[10] = {0};
    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[ (arr[i]/exp)%10 ]++;
    // Change count[i] so that count[i] now contains actual
    // position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
    // Build the output array
    for (i = n - 1; i >= 0; i--) {
        count[ (arr[i]/exp)%10 ]--;
        output[ count[(arr[i]/exp)%10] ] = arr[i];
    }
    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

RISC-V src/countingsort.s

```
#   // Store count of occurrences in count[]
#   for (i = 0; i < n; i++)
#       count[ (arr[i]/exp)%10 ]++;
    li t0,0 #i=0
loop3:
    li t1,bytes
    mul t1,t0,t1 #offset from arr[0] to arr[i]
    add t2,a3,t1 #t2 = address of arr[i]
    lw t3,0(t2)  #t3 = arr[i]
    # count[ (arr[i]/exp)%10 ]++;
    div t4,t3,a4
    li t1,10
    rem t4,t4,t1 #t4 = (arr[i]/exp)%10 //index of count array
    la a5,count  #a5 = count array address
    li t1,bytes
    mul t1,t4,t1 #offset from count[0] to count[t4]
    add t5,a5,t1 #t5 = address of count[t4]
    lw t6,0(t5)  #t6 = count[t4]
    addi t6,t6,1 #t6 = t6 + 1
    sw t6,0(t5)  #count[t4] = count[t4] + 1

    addi t0,t0,1    #i++
    blt t0,a2,loop3 #i<n
endloop3:
```

countingsort() [2/3]

C++ algorithm_cpp/radix.cpp

```cpp
void countSort(int arr[], int n, int exp)
{
    int output[n]; // output array
    int i, count[10] = {0};
    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[ (arr[i]/exp)%10 ]++;
    // Change count[i] so that count[i] now contains actual
    // position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
    // Build the output array
    for (i = n - 1; i >= 0; i--) {
        count[ (arr[i]/exp)%10 ]--;
        output[ count[(arr[i]/exp)%10] ] = arr[i];
    }
    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

RISC-V src/countingsort.s

```
#for (i = 1; i < 10; i++)
#       count[i] += count[i - 1];
    li t0,1 #i=1
loop4:
    la a5,count  #a5 = count array address
    li t1,bytes
    addi t0,t0,-1   #t0 = i-1
    mul t1,t0,t1    #offset from count[0] to count[i-1]
    add t2,a5,t1    #t2 = address of count[i-1]
    lw t4,0(t2)     #t4 = count[i-1]
    addi t0,t0,+1   #t0 = i
    li t1,bytes
    mul t1,t0,t1    #offset from count[0] to count[i]
    add t2,a5,t1    #t2 = address of count[i]
    lw t3,0(t2)     #t3 = count[i]
    add t3,t3,t4   #t3 = count[i] + count[i - 1];
    sw t3,0(t2)     #count[i] = t3

    addi t0,t0,1
    li t1,10
    blt t0,t1,loop4
endloop4:
```

countingsort() [3/3]

C++ algorithm_cpp/radix.cpp

```cpp
void countSort(int arr[], int n, int exp)
{
    int output[n]; // output array
    int i, count[10] = {0};
    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[ (arr[i]/exp)%10 ]++;
    // Change count[i] so that count[i] now contains actual
    // position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
    // Build the output array
    for (i = n - 1; i >= 0; i--) {
        count[ (arr[i]/exp)%10 ]--;
        output[ count[(arr[i]/exp)%10] ] = arr[i];
    }
    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

RISC-V src/countingsort.s

```
    addi t0,a2,-1 #i=n-1
loop6:
    li t1,bytes
    mul t1,t0,t1 #offset from arr[0] to arr[i]
    add t2,a3,t1 #t2 = address of arr[i]
    lw t3,0(t2)  #t3 = arr[i]

    # count[ (arr[i]/exp)%10 ]--;
    div t4,t3,a4
    li t1,10
    rem t4,t4,t1 #t4 = (arr[i]/exp)%10 //index of count array
    la a5,count  #a5 = count array address
    li t1,bytes
    mul t1,t4,t1 #offset from count[0] to count[t4]
    add t5,a5,t1 #t5 = address of count[t4]
    lw t6,0(t5)  #t6 = count[t4]
    addi t6,t6,-1 #t6 = t6 - 1
    sw t6,0(t5)  #count[t4] = count[t4] - 1

    #keeping t3=arr[i] and t6=count[ (arr[i]/exp)%10 ]
    #free tmp registers: t1,t2,t4,t5

    li t1,bytes
    mul t1,t6,t1 #offset from output[0] to output[t6]
    add t2,s3,t1 #t2 = address of output[t6]
    sw t3,0(t2)  #output[count[ (arr[i]/exp)%10 ] ] = arr[i];

    addi t0,t0,-1 #i--
    bge t0,zero,loop6 #i>=0
endloop6:
```

max()

C++ algorithm_cpp/radix.cpp

```cpp
// A utility function to get maximum value in arr[]
int getMax(int arr[], int n)
{
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}
```

RISC-V src/arrayutils.s

```
max:
    #a2=n #a3=array address #a0=max #a1=max address
    li t3,0     #i=0
    lw t2,0(a3)  #arr[i]
    #set first element as max
    addi a0,t2,0
    addi a1,a3,0
    li t3,1 #i=1
    bge t3,a2,endloop1 #i>=n
loop1:
    li t0,bytes
    mul t1,t3,t0 #offset
    add t0,a3,t1 #address of arr[i]
    lw t2,0(t0)  #arr[i]

    bgt t2,a0,then1 #if arr[i]>max
    j endif1 #do nothing
then1:
    #set arr[i] as max
    addi a0,t2,0
    addi a1,t0,0
endif1:
    addi t3,t3,1 #i++
    blt t3,a2,loop1 #i<n
endloop1:
ret
```

Debug e Risultati

- Compile & Run
  - Terminale 1: $ ./build.sh
  - Terminale 2: $ ./debug.sh

  La variabile di I/O appare nel debugger in automatico

- Risultati ottenuti
  - {170, 45, 75, 90, 802, 69, 4, 20} {4, 20, 45, 69, 75, 90, 170, 802}
  - {423, 65, 1004, 53, 5} {5, 53, 65, 423, 1004}
  - {2, 90, 20, 1} {1, 2, 20, 90}
  - {24985,399824,298342} {24985, 298342, 399824}

Conclusioni

- Ogni operazione semplice scritta in codice di alto livello può diventare lunga ed error-prone in RISC-V assembly, quindi è necessario suddividere il problema in più fasi:
  - Ideazione dell'algoritmo ad alto livello
  - Scomposizione top-down del codice in parti
  - Sviluppo (per ogni parte)
    - Traduzione del codice hi-level in assembly
    - Documentazione del codice tramite commenti
    - Debug e Testing
- Una volta superato questo ostacolo e completato il programma, RISC-V permette di ottenere una velocità di esecuzione molto alta e ottimizzata.

GRAZIE PER L'ATTENZIONE