ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

Computer Architecture (CO2008)

Báo cáo bài tập lớn môn Kiến trúc máy tính
Sắp xếp chuỗi bằng giải thuật Quick Sort

GVHD: Vũ Trọng Thiên
SV thực hiện: Nguyễn Phúc Tiến – 2014725
Tô Dịu Quang – 2014251
Nguyễn Hữu Hiếu – 2013149

Ngày 3 tháng 6 năm 2022

Mục lục

1 Đề bài

Đề 5: Sắp xếp chuỗi.
Cho một chuỗi số nguyên 50 phần tử. Sử dụng hợp ngữ assembly MIPS, viết thủ tục sắp xếp chuỗi đó theo thứ tự tăng dần theo giải thuật Quick Sort. Yêu cầu xuất ra từng bước trong quá trình demo.

2 Thuật toán Quick Sort

2.1 Khái niệm Quick Sort

Thuật toán Quick Sort (Sắp xếp nhanh) là một quy trình có hệ thống để sắp xếp các phần tử của một mảng. QuickSort là một thuật toán sử dụng cách thức chia để trị (Divide and Conquer algorithm).

Tên gọi "Quick Sort" ám chỉ thuật toán này có khả năng sắp xếp dữ liệu nhanh hơn nhiều so với bất kỳ thuật toán sắp xếp truyền thống nào khác. Tuy nhiên, Quick Sort không được ổn định vì thứ tự tương đối của các phần tử bằng nhau không được đảm bảo.

2.2 Cách thức hoạt động của Quick Sort

Thuật toán sẽ chọn ra một phần tử trong mảng để làm điểm đánh dấu gọi là pivot. Sau khi chọn được điểm đánh dấu, nó sẽ chia mảng đó thành hai mảng con bằng cách so sánh với pivot đã chọn. Một mảng sẽ bao gồm các phần tử nhỏ hơn hoặc bằng pivot và mảng còn lại luôn lớn hơn hoặc bằng pivot.

Sau đó, quá trình này được lặp lại đủ số lần cho đến khi các mảng nhỏ có thể được sắp xếp một cách dễ dàng để tạo ra một tập dữ liệu được sắp xếp đầy đủ.

Các nhiều phiên bản Quick Sort khác nhau chọn pivot theo những cách khác nhau. Tốc độ sắp xếp của thuật toán phải phụ thuộc vào việc lựa chọn pivot, có một số cách để chọn như sau:

1. Luôn chọn phần tử đầu tiên của mảng.
2. Luôn chọn phần tử cuối cùng của mảng.
3. Chọn một phần tử random.
4. Chọn một phần tử có giá trị nằm giữa mảng (median element).

2.3 Độ phức tạp

2.3.1 Độ phức tạp về thời gian của Quick Short

Trong các trường hợp tốt nhất, trung bình và xấu nhất, thuật toán Quick Sort thực hiện với độ phức tạp O(n), O(nlog n) và O(n^2) tương ứng. Đây là một trong những thuật toán sắp xếp hiệu quả nhất khi nói đến độ phức tạp về thời gian.

2.3.2 Độ phức tạp về không gian của Quick Short

Độ phức tạp không gian trung bình của Quick Sort là O(log n) và độ phức tạp không gian trong trường hợp xấu nhất là O(n). Điều này ngang bằng với hầu hết các thuật toán sắp xếp phổ biến, nhưng bản chất của thuật toán đệ quy là chúng không tối ưu hóa việc sử dụng bộ nhớ.

2.4 Ưu điểm

Chạy nhanh (nhanh nhất trong các thuật toán sắp xếp dựa trên việc só sánh các phần tử). Do đó quicksort được sử dụng trong nhiều thư viện của các ngôn ngữ như Java, C++ (hàm sort của C++ dùng Intro sort, là kết hợp của Quicksort và Insertion Sort).

2.5 Nhược điểm

Tùy thuộc vào cách chia thành 2 phần, nếu chia không tốt, độ phức tạp trong trường hợp xấu nhất có thể là O(n^2). Nếu ta chọn pivot ngẫu nhiên, thuật toán chạy với độ phức tạp trung bình là O(n log n) (trong trường hợp xấu nhất vẫn là O(n^2), nhưng ta sẽ không bao giờ gặp phải trường hợp đó).

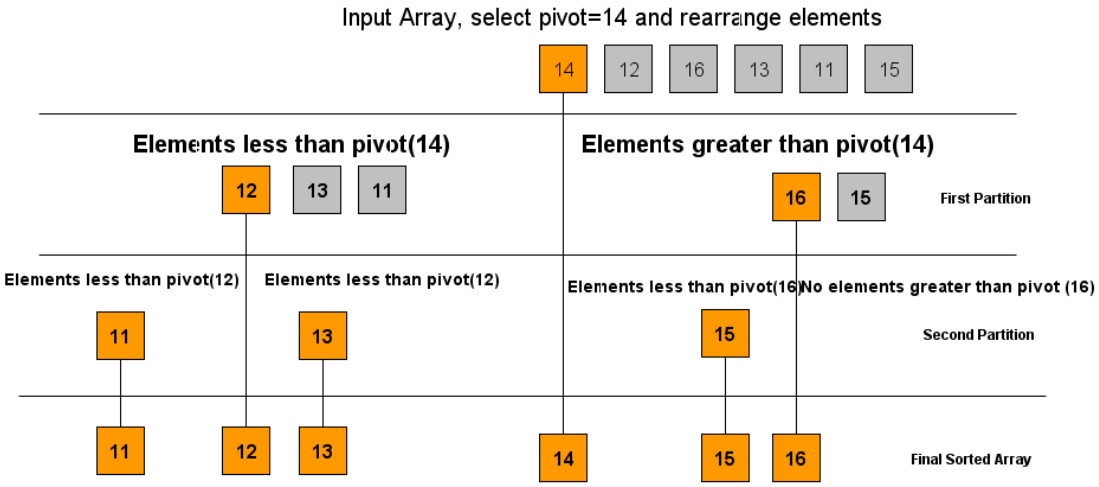
Do tốc độ sắp xếp thuật toán phải phụ thuộc vào việc lựu chọn pivot, do đó nó có tính không ổn đinh.

3 Giải pháp hiện thực

Như ta đã biết, Quick Sort có 4 cách chọn pivot. Lần thực hiện này nhóm đã chọn phần tử đầu tiên làm pivot.

Thuật toán sẽ có hai giai đoạn. Giai đoạn đầu là phân đoạn mảng (partition()) và giai đoạn sau là sắp xếp (quickSort()).

1. Giai đoạn Partition
   Ở giai đoạn này, nhiệm vụ của ta là tìm ra vị trí chính xác của pivot và chia mảng hiện tại thành 2 mảng con, 1 mảng có giá trị nhỏ hơn pivot và 1 mảng có giá trị lớn hơn pivot. Cuối cùng di chuyển pivot về đúng vị trí của nó. Cụ thể:
   - Đặt pivot là phần tử đầu tiên.
   - Ta bắt đầu bằng 2 index i, j. Index i nằm ở phần tử bên trái cùng của dãy chỉ số + 1 (bỏ qua pivot) và đảm nhận việc tìm phần tử có giá trị lớn hơn pivot. Index j là index nằm ở vị trí bên phải cùng và tìm phần tử có giá trị nhỏ hơn pivot.
   - Khi 2 điều kiện này thỏa mãn và i $\leq$ j thì đổi chỗ 2 giá trị cho nhau. Sau cùng ta đổi giá trị của phần tử nằm ở index j và pivot. Khi này pivot sẽ nằm đúng vị trí của mình và chia dãy số thành 2 dãy (nhỏ hơn và lớn hơn pivot).
2. Giai đoạn Quick Sort
   Ở giai đoạn này, nhiệm vụ ta đơn giản hơn, chỉ cần lấy trị trả về của hàm partition, đó cũng là vị trí của pivot. Sau đó ta sẽ gọi đệ quy để sắp xếp 2 mảng con trái và phải. Trường hợp dừng của đệ quy là khi các mảng con có kích thước bằng 0 hoặc 1, khi đó nó đã được sắp xếp theo định nghĩa, vì vậy ta không cần phải sắp xếp lại.



Hình 1: *Ảnh mô tả quá trình sắp xếp mảng bằng giải thuật Quick Sort*

Khi đã hoàn chỉnh được hướng đi, nhóm sẽ tiến hành code trên C++ trước, sau đó sẽ code tương tự bằng ngôn ngữ MIPS assembly. Dưới đây là 2 hàm partition() và quicksort() theo ngôn ngữ C++.

```
int Partition(int arr[], int start, int end) {
    // return the pointer which points to the pivot after rearrange the array.
    int pivot = arr[start];
    int i = start + 1;
    int j = end;
    do {
      while ((i <= j) && (arr[i] <= pivot)) {
        i++;
      }
      while ((i <= j) && (arr[j] > pivot)) {
        j--;
      }
      if (i < j) {
        swap(arr[i], arr[j]);
      }
    } while (i <= j);

    swap(arr[j], pivot)

    return j;
}

void QuickSort(int arr[], int start, int end) {
      // print out the index of pivot in subarray after everytime calling method
    Partition.
    if (start < end) {
        int j = Partition(arr, start, end);
        cout << j << " ";
        QuickSort(arr, start, j - 1);
        QuickSort(arr, j + 1, end);
    }
}
```

4 Hiện thực code

Trong phần .data ta sẽ khai báo những testcase cho Quick Sort cũng như các string để giao tiếp với người dùng.

```
 .data
# define testcases for quicksort
testcase1: .word 129, 10, 31, 44, 16, 3, 33, 34, 35, 44, 44, 25, 48, 16, 32, 37, 8, 33,
    30, 6, 18, 26, 0, 37, 40, 30, 50, 32, 5, 41, 0, 32, 12, 33, 22, 14, 34, 1, 0, 41, 45,
    8, 39, 27, 23, 45, 10, 50, 34, 47
testcase2: .word 39, 49, 2, 6, 46, 8, 18, 31, 3, 3, 21, 28, 24, 46, 16, 29, 9, 4, 8, 11,
    3, 49, 23, 11, 34, 30, 48, 2, 5, 45, 8, 30, 14, 14, 0, 6, 33, 31, 16, 12, 20, 36, 3,
    37, 8, 36, 44, 45, 9, 7
testcase3: .word 8, 10, 25, 11, 15, 41, 15, 4, 20, 0, 11, 17, 35, 17, 20, 16, 48, 29, 8,
    2, 28, 13, 17, 40, 3, 45, 40, 29, 40, 28, 12, 45, 46, 28, 5, 40, 24, 6, 42, 32, 29,
    33, 45, 27, 11, 26, 38, 29, 25, 21
testcase4: .word -349, 95, 395, -277, 164, 241, -110, -64, -158, -1010, 66, -197, -52,
    158, 2022, 88, -192, 344, -180, -201, -384, -422, 333, 488, -331, -7, 372, 37, 331,
    -106, 370, 482, -159, 265, -294, -495, 359, 448, 297, 53, -351, 212, 239, 449, 611,
    53, 37, -118, 249, 357
# message
aftersort: .asciiz "After sorting: "
indexOfpivots: .asciiz "Index of pivots: "
space: .asciiz " "
```

Tiếp đến phần .text, phần code chính của chương trình.

– Đầu tiên ta sẽ print các message và đặt các argument $a0, $a1, $a2 lần lượt là: địa chỉ mảng, giá trị start, giá trị end. Hai giá trị start và end là các chỉ số đầu và cuối của mảng.

```
# Print message
li      $v0, 4
la      $a0, indexOfpivots
```

```
syscall
# Store value to argument
la      $t0, testcase1  # Moves the address of array into register $t0

addi    $a0, $t0, 0     # Set argument 1 to the array.
addi    $a1, $zero, 0   # Set argument 2 to (start = 0)
addi    $a2, $zero, 49  # Set argument 3 to (end = 49, last index in array)
```

– Sau khi hoàn tất các argument, ta sẽ gọi hàm quicksort().

```
jal     quicksort # Call quicksort
```

– Bởi vì trong hàm quicksort() có gọi thêm hàm partition() cho nên ta phải sử dụng stack để lưu các argument, vị trí pivot cũng như là địa chỉ trả về $ra lên stack. Tổng cộng ta cần lưu 5 giá trị lên stack cho nên ta sẽ mở rộng stack lên 5.

```
quicksort:    # Implement quicksort method
      addi    $sp, $sp, -20     # Adjust stack for 5 items

      sw      $a0, 0($sp)       # Store array address
      sw      $a1, 4($sp)       # Store start
      sw      $a2, 8($sp)       # Store end
      sw      $ra, 12($sp)      # Store return address
      sw      $s0, 16($sp)      # Store pivot index
```

– Ta chỉ thực hiện đệ quy khi giá trị $start < end$. Thanh ghi $a1 chứa giá trị start, thanh ghi $a2 chứa giá trị end như ta đã đặt trước đó. Ta sử dụng lệnh beq để đặt điều kiện.

```
      bge     $a1, $a2, endif   # If start >= end, endif
```

– Nếu thỏa điệu kiện $start < end$ thì ta gọi hàm partition để tìm ra vị trí chính xác của pivot (hàm partition sẽ được mô tả cụ thể sau).

```
      jal     partition         # Call partition
```

– Sau khi kết thúc việc gọi hàm và giá trị trả về index j (vị trí pivot), gọi giá trị đó là pi, ta sẽ thực hiện gọi đệ quy cho dãy số bên trái pivot. Bắt đầu từ vị trí start và kết thúc ở pi - 1. Ta truyền argument là 3 ô ở stack là start, end và pivot index. Vì hàm đệ quy sẽ gọi lại hàm quicksort() cho nên ta không cần dùng sw (store word) để lưu vào stack, ta cần cập nhật giá trị của thanh ghi vì lần gọi lại hàm sẽ cập nhật lên stack.

```
      add     $s0, $v0, $zero   # s0 pivot index = v0 (pi)
      lw      $a1, 4($sp)       # a1 = start
      addi    $a2, $s0, -1      # a2 = pi -1
      jal     quicksort         # Call quicksort
```

– Tương tự như dãy số bên trái, dãy số bên phải sẽ bắt đầu từ vị trí pi + 1 (trừ pivot) và kết thúc là vị trí end.

```
      addi    $a1, $s0, 1       # a1 = pi + 1
      lw      $a2, 8($sp)       # a2 = end
      jal     quicksort         # Call quicksort
```

– Nếu điều kiện $start < end$ trên không thỏa thì chương trình sẽ rẽ nhánh đến label $endif$. Tại đây ta tiến hành khôi phục lại các phần tử vào thanh ghi tương ứng và pop ra khỏi stack. Ta thêm 5 phần tử thì sẽ tiến hành pop ra 5 phần tử. Cuối cùng sẽ jump về chỗ gọi hàm.

```
endif:
      lw      $a0, 0($sp)       # Restore a0
      lw      $a1, 4($sp)       # Restore a1
      lw      $a2, 8($sp)       # Restore a2
      lw      $ra, 12($sp)      # Restore return address
      lw      $s0, 16($sp)      # Restore index of pivot
      addi    $sp, $sp, 20      # Pop 5 items from stack
      jr      $ra               # Junp back to the caller
```

Tiếp đến ta sẽ thực thi hàm partition mà trong hàm quicksort() đã gọi.

- Do ta đã mô tả hàm partition thì trong hàm này có gọi một hàm khác là hàm swap(). Cho nên ta cần phải lưu địa chỉ trả về cũng như 3 giá trị thanh ghi argument $a1, $a2, $a3. Tổng cộng sẽ là 4 phần tử cần lưu lên stack nên ta mở rộng stack thêm 4 phần tử.

```
partition:  # Implement partion function
        addi    $sp, $sp, -16   # Adjust stack for 4 items

        sw      $a0, 0($sp)     # Store a0
        sw      $a1, 4($sp)     # Store a1
        sw      $a2, 8($sp)     # Store a2
        sw      $ra, 12($sp)    # Store return address
```

- Trong hàm này, thanh ghi chứa các giá trị sau đây:
  + Thanh ghi $s1: Giá trị start (bên trái cùng của mảng) dạng word.
  + Thanh ghi $s2: Giá trị end (bên phải cùng của mảng) dạng word.
  + Thanh ghi $t3: Giá trị phần tử ở vị trí start, giá trị pivot.
  + Thanh ghi $t4: Chỉ số index i = start + 1.
  + Thanh ghi $t5: Giá trị index j = end.
  + Thanh ghi $t6: Giá trị phần tử nằm ở vị trí index i.
  + Thanh ghi $t7: Giá trị phần tử nằm ở vị trí index j.
- Đầu tiên ta khởi tạo các giá trị end, start, pivot, index i và j.

```
        move    $s1, $a1        # s1 = start
        move    $s2, $a2        # s2 = end

        sll     $t3, $s1, 2     # t3 = 4 * start
        add     $t3, $a0, $t3   # t3 = pivot = arr + start;
        lw      $t3, 0($t3)

        addi    $t4, $s1, 1     # t4 = index i = start + 1;
        add     $t5, $s2, $zero # t5 = index j = end;
```

- Tiếp đến ta sẽ tìm giá trị mà lớn hơn pivot bằng index i khi (i ≤ j).

```
        while:
                bgt     $t4, $t5, endwhile
                search_ge: # Search for element greater than or equal to pivot
```

  + Vì mỗi phần tử integer chiếm 4 bytes cho nên ta cần nhân index lên 4 lần hoặc shift left 2 rồi cộng với địa chỉ mới ra ô nhớ mà ta cần. Đầu tiên ta lấy giá trị ra khỏi ô nhớ để so sánh với pivot.

```
                # while ((i <= j) && (arr[i] <= pivot)) i++;
                sll     $t6, $t4, 2                     # t6 = 4i
                add     $t6, $a0, $t6                   # t6 = arr + i
                lw      $t6, 0($t6)                     # t6 = arr[i]
```

  + Nếu giá trị phần tử bé hơn pivot, ta sẽ cộng index lên 1 và jump về vòng lặp. Ngược lại (đã tìm được vị trí ≥ pivot) ta sẽ rẽ nhánh đến label end loop với điều kiện bắt buộc là (i ≤ j).

```
                bgt     $t4, $t5, end_search_ge         # if (i > j) break
                bgt     $t6, $t3, end_search_ge         # if (arr[i] > pivot) break
                addi    $t4, $t4, 1                     # else i++
                j       search_ge
            end_search_ge:
```

- Tương tự như việc tìm kiếm phần tử lớn hơn pivot, việc tìm kiếm phần tử nhỏ hơn pivot sẽ bắt đầu từ index j, tức phần tử cuối cùng bên phải mảng. Nếu giá trị của ô nhớ lớn hơn pivot, thì ta sẽ giảm index 1 đơn vị và jump về vòng lặp. Ngược lại (đã tìm được vị trí < pivot) ta sẽ rẽ nhánh đến label end loop với điều kiện bắt buộc là (i ≤ j).

```
            search_smaller: # Search for element smaller than pivot
            # while ((i <= j) && (arr[j] > pivot))
            sll     $t7, $t5, 2                 # t6 = 4j
            add     $t7, $a0, $t7               # t7 = arr + j
            lw      $t7, 0($t7)                 # t7 = arr[i]
            bgt     $t4, $t5, end_search_smaller  # if (i > j) break
            ble     $t7, $t3, end_search_smaller  # if (arr[j] <= pivot) break
            subi    $t5, $t5, 1                 # else j--
            j       search_smaller
        end_search_smaller:
```

– Sau khi 2 vòng lặp tìm kiếm vị trí mà phần tử lớn hơn hoặc bằng và bé hơn pivot với điều kiện (i ≤ j). Ta sẽ swap 2 phần tử đó với nhau và jump về label *while*. Phần tử bé hơn sẽ ở bên trái, phần tử lớn hơn hoặc bằng sẽ về bên phải. Lúc này dãy số của ta sẽ dần dần được phân ra 2 dãy con bởi giá trị pivot. Khi swap, ta cần truyền 2 index i và j vào 2 thanh ghi argument $a1 và $a2 tương ứng với i và j.

```
            # swap(arr[i], arr[j])
            bgt     $t4, $t5, endwhile          # if (i > j) break
            add     $a1, $t4, $zero             # a1 = t4 = index i
            add     $a2, $t5, $zero             # a0 = t4 = index j
            jal     swap

            j       while
    endwhile:
```

– Kết thúc vòng lặp while (i ≤ j) tức pivot đã tìm được vị trí chính xác của nó, vị trí của j. Ta sẽ swap pivot với phần tử index j. Lúc này dãy số của ta sẽ được chia nhỏ ra 2 dãy số với dãy bên trái nhỏ hơn pivot và dãy bên phải lớn hơn pivot.

```
    # swap(arr[j], pivot)
    lw      $a1, 4($sp)         # Load start from stack
    add     $a2, $t5, $zero     # a0 = t5 = index j
    jal     swap
```

– Sau đó ta in ra vị trí của pivot ngoài console để ta có thể kiểm soát được hành vi chương trình của mình và return giá trị hàm partition() qua thanh ghi $v0.

```
    # Print pivot index
    li      $v0, 1
    move    $a0, $a2
    syscall
    # Print space
    li      $v0, 4
    la      $a0, space
    syscall

    add     $v0, $a2, $zero     # Return j
```

– Trước khi thoát khỏi hàm, ta cần load lại thanh ghi $ra và restore lại thanh ghi $a0 vì ta có xài thanh ghi này khi print string và integer trên. Cuối cùng ta pop những phần tử như ta đã thêm, ta thêm 4 lúc đầu thì sẽ pop 4 phần tử ra khỏi stack.

```
    lw      $a0, 0($sp)         # Restore a0
    lw      $ra, 12($sp)        # Return address
    addi    sp, $sp, 16         # Pop 4 items from stack
    jr      $ra                 # Junp back to the caller
```

Tiếp theo ta sẽ thực thi hàm swap() mà trong hàm partition() đã gọi. Hàm swap() này sẽ sử dụng 3 tham số truyền vào là địa chỉ của mảng và 2 index cần swap tương ứng với 3 thanh ghi argument $a0, $a1, $a2.

– Các thanh ghi sử dụng trong hàm này:

  + Thanh ghi $t1: Địa chỉ phần tử nằm ở index i.

  + Thanh ghi $t2: Địa chỉ phần tử nằm ở index j.

+ Thanh ghi $s3: Giá trị phần tử nằm ở index i.
+ Thanh ghi $s4: Giá trị phần tử nằm ở index j.

– Vì mỗi phần tử integer chiếm 4 bytes nên ta cũng cần nhân index lên 4 lần rồi mới cộng với địa chỉ ô nhớ.

```
swap: # Implement swap function
        sll     $t1, $a1, 2     # t1 = 4 * a1 = 4 * i
        sll     $t2, $a2, 2     # t2 = 4 * a2 = 4 * j
        add     $t1, $a0, $t1   # t1 = arr + 4i
        add     $t2, $a0, $t2   # t2 = arr + 4j

```

– Load giá trị tại 2 index vào 2 thanh ghi $s3, $s4.

```
        lw      $s3, 0($t1)     # s3 = arr[i]
        lw      $s4, 0($t2)     # s4 = arr[j] = temp

```

– Cuối cùng là swap 2 giá trị đó cho nhau và jump về chỗ gọi hàm.

```
        # swap two elements
        sw      $s3, 0($t2)     # arr[j] = arr[i]
        sw      $s4, 0($t1)     # arr[i] = temp
        jr      $ra             # Jump back
endswap:

```

Như vậy quá trình thực hiện hàm quicksort đã kết thúc đồng nghĩa với việc dãy số của ta đã được sắp xếp. Chương trình của ta đã jump về sau lệnh gọi jal quicksort (lần gọi đầu tiên). Ta sẽ print các message và dãy số ra kiểm tra tính chính xác. Địa chỉ của mảng nằm ở thanh ghi argument $a0.

– Print các string ra console và gán lại các giá trị cần thiết.

```
li      $t1, 0      # Index use for print array loop
move    $t0, $a0    # Move array address to t0
# Print endline
li      $v0, 11     # Code 11 to print character
li      $a0, 10     # Ascii: 10 = newline
syscall
# Print message "After sorting: "
li      $v0, 4
la      $a0, aftersort
syscall

```

– Print dãy số ra console. Với dãy số gồm 50 số nguyên thì ta cần 50 * 4 = 200 bytes. Đó cũng là điều kiện để kết thúc vòng lặp (index = 200).

```
print_sort:
        beq     $t1, 200, exit_print
        ld      $s0, 0($t0)     # Load arr[i] into s0
        # Print integer
        li      $v0, 1          # Code = 1 to print int
        addi    $a0, $s0, 0     # Argument a0 = s0
        syscall
        addi    $t1, $t1, 4     # i += 4
        addi    $t0, $t0, 4     # arr = arr + 4
        # Print space
        li      $v0, 4
        la      $a0, space
        syscall
        j       print_sort
exit_print:

```

– Cuối cùng sẽ load giá trị 10 cho thanh ghi $v0 để kết thúc chương trình.

```
li      $v0, 10     # exit()
syscall

```

4.1 Kết quả

4.1.1 Testcase 1

```
testcase1: .word 129, 10, 31, 44, 16, 3, 33, 34, 35, 44, 44, 25, 48, 16, 32, 37, 8, 33,
                 30, 6, 18, 26, 0, 37, 40, 30,50, 32, 5, 41, 0, 32, 12, 33, 22, 14, 34,
                 1, 0, 41, 45, 8, 39, 27, 23, 45, 10, 50, 34, 47
```

Kết quả :

```
Index of pivots: 49 45 44 17 6 3 2 1 5 12 10 9 8 14 15 22 21 19 35 32 31 26 25 24 30 29 28 34 42 36 41 39 38 48 46
After sorting: 0 0 0 1 3 5 6 8 8 10 10 12 14 16 16 18 22 23 25 26 27 30 30 31 32 32 32 33 33 33 34 34 34 35 37 37 39 40 41 41 44 44 44 45 45 47 48 50 50 129
-- program is finished running --
```

4.1.2 Testcase 2

```
testcase2: .word 39, 49, 2, 6, 46, 8, 18, 31, 3, 3, 21, 28, 24, 46, 16, 29, 9, 4, 8, 11,
                 3, 49, 23, 11, 34, 30, 48, 2, 5, 45, 8, 30, 14, 14, 0, 6, 33, 31, 16,
                 12, 20, 36, 3, 37, 8, 36, 44, 45, 9, 7
```

Kết quả :

```
Index of pivots: 41 39 20 6 0 2 5 4 10 9 8 15 14 13 11 17 19 31 22 24 25 26 28 30 33 35 36 38 44 42 46 47 49
After sorting: 0 2 2 3 3 3 3 4 5 6 6 7 8 8 8 8 9 9 11 11 12 14 14 16 16 18 20 21 23 24 28 29 30 30 31 31 33 34 36 36 37 39 44 45 45 46 46 48 49 49
-- program is finished running --
```

4.1.3 Testcase 3

```
testcase3: .word 8, 10, 25, 11, 15, 41, 15, 4, 20, 0, 11, 17, 35, 17, 20, 16, 48, 29, 8,
                 2, 28, 13, 17, 40, 3, 45, 40, 29, 40, 28, 12, 45, 46, 28, 5, 40, 24, 6,
                 42, 32, 29, 33, 45, 27, 11, 26, 38, 29, 25, 21
```

Kết quả:

```
Index of pivots: 7 3 2 1 4 6 21 13 8 11 10 20 19 15 18 16 30 29 28 25 23 27 47 34 33 32 42 41 36 38 40 46 43 44 49
After sorting: 0 2 3 4 5 6 8 8 10 11 11 11 12 13 15 15 16 17 17 17 20 20 21 24 25 25 26 27 28 28 28 29 29 29 29 32 33 35 38 40 40 40 40 41 42 45 45 45 46 48
-- program is finished running --
```

4.1.4 Testcase 4

```
testcase4: .word -349, 95, 395, -277, 164, 241, -110, -64, -158, -1010, 66, -197, -52,
                 158, 2022, 88, -192, 344, -180, -201, -384, -422, 333, 488, -331, -7,
                 372, 37, 331, -106, 370, 482, -159, 265, -294, -495, 359, 448, 297, 53,
                 -351, 212, 239, 449, 611, 53, 37, -118, 249, 357
```

Kết quả:

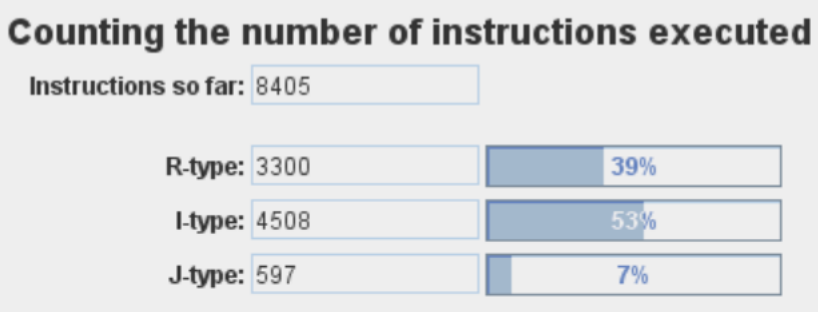
```
Index of pivots: 5 0 4 3 2 16 11 10 6 9 7 13 15 38 35 30 24 19 18 22 21 28 27 25 31 33 36 44 41 39 43 48 46
After sorting: -1010 -495 -422 -384 -351 -349 -331 -294 -277 -201 -197 -192 -180 -159 -158 -118 -110 -106 -64 -52 -7 37 37 53 53 66 88 95 158 164
 212 239 241 249 265 297 331 333 344 357 359 370 372 395 448 449 482 488 611 2022
-- program is finished running --
```

5 Thống kê số lệnh, loại lệnh và thời gian chạy của chương trình.

5.1 Thống kê số lệnh, loại lệnh (instruction type) của chương trình.

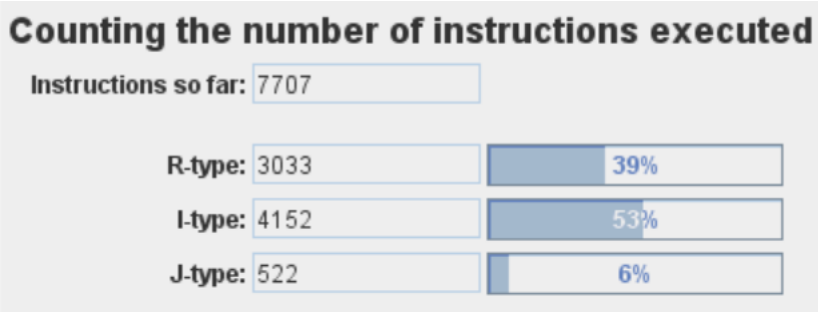
5.1.1 Testcase 1

Số lệnh thực thi đối với testcase1 có tổng cộng 8405 lệnh, trong đó R-type có 3300 lệnh(chiếm 39,2% tổng số lệnh), I-type có 4408 lệnh(chiếm 53,5% tổng số lệnh), J-type có 597 lệnh (chiếm 7,3% tổng số lệnh).



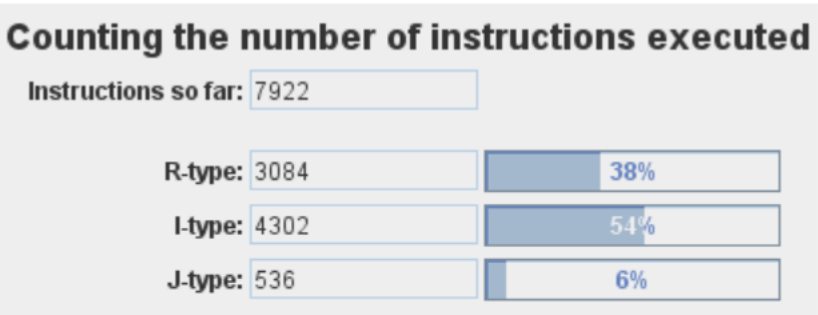
5.1.2 Testcase 2

Số lệnh thực thi đối với testcase2 có tổng cộng 7707 lệnh, trong đó R-type có 3033 lệnh(chiếm 39,4% tổng số lệnh), I-type có 4152 lệnh(chiếm 53,8% tổng số lệnh), J-type có 522 lệnh (chiếm 6,8% tổng số lệnh).



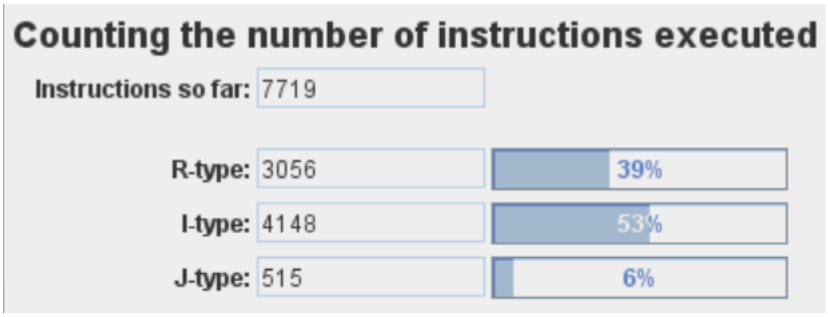
5.1.3 Testcase 3

Số lệnh thực thi đối với testcase3 có tổng cộng 7922 lệnh, trong đó R-type có 3084 lệnh(chiếm 38,9% tổng số lệnh), I-type có 4302 lệnh(chiếm 54,3% tổng số lệnh), J-type có 536 lệnh (chiếm 6,8% tổng số lệnh).



5.1.4 Testcase 4

Số lệnh thực thi đối với testcase4 có tổng cộng 7719 lệnh, trong đó R-type có 3056 lệnh(chiếm 39,6% tổng số lệnh), I-type có 4148 lệnh(chiếm 53,7% tổng số lệnh), J-type có 515 lệnh (chiếm 6,7% tổng số lệnh).



5.2 Thời gian chạy (execution time) trên máy tính kiến trúc MIPS

Ta có công thức tính thời gian chạy (execution time) là:

$$Execution\ time = \frac{CPI \times Total\ Instructions}{Clock\ rate}$$

Với CPI = 1 (đối với hệ thống single cycle) và Clock rate = 3.4 GHz, ta có :

Test case	Execution Time
testcase1	2.472 μs
testcase2	2.2668 μs
testcase3	2.33 μs
testcase4	2.2703 μs

6 Kết luận và Đánh giá

6.1 Kết luận

Bài báo cáo dựa trên kết quả của nhóm sau quá trình nghiên cứu và tìm hiểu. Qua đó, các thành viên trong nhóm đã củng cố thêm được kiến thức về kiến trúc tập lệnh (MIPS), thực thi chương tình sắp xếp theo giải thuật QuickSort, sử dụng các công cụ thống kê của MARS, các kiến thức liên quan đến bộ môn Kiến trúc Máy tính... Ngoài ra, nhóm đã phát triển thêm các kỹ năng ngoài lề như kỹ năng giải quyết vấn đề, kỹ năng làm việc nhóm, kỹ năng tự đánh giá, soạn thảo bằng Latex,...phục vụ cho các môn học sau.

6.2 Đánh giá

Name	ID	Task	Result
Nguyễn Phúc Tiến	2014725	Lên ý tưởng Viết code Làm báo cáo	100%
Tô Dịu Quang	2014251	Lên ý tưởng Debug Code Làm báo cáo	100%
Nguyễn Hữu Hiếu	2013149	Debug Code Làm báo cáo	100%

Tài liệu

[1] David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Fifth edition,Morgan Kaufmann Publishers, 2017.

[2] Dinh Duc Anh Vu, *Lectures slide of Computer Architecture*, Ho Chi Minh City University of Technology, 2021.

[3] Dong Tung, *Quick Sort là gì? Tìm hiểu chi tiết về Quick Sort*, https://wiki.tino.org/quick-sort-la-gi/.