NAME: CARLOS

SURNAME: GUTIERREZ BERNAL

E-mail: cagut8951@uit.no

GitHub username: guti-1

GitHub repository: https://github.com/guti-1/assignemt-1_computerArchitecture.git or https://github.com/uit-inf-2200/assembly-x86-64-mik.git

INTRODUCTION

Description of benchmark

The benchmark used evaluates performance of the quicksort algorithm implemented in C and assembly. Quicksort is what his name says a sort algorithm employs a divide-and-conquer strategy to sort data efficiently. It complexity of O(n logn), which makes it suitable for handling large datasets.

The chosen benchmark involves sorting an array of 1,000,000 integers, which is a substantial size to stress-test the sorting algorithms. This benchmark is particularly relevant as it provides a practical scenario where sorting performance can significantly impact overall system efficiency, such as in databases, data analysis tools, and other software requiring rapid data manipulation.

IMPLEMENTATION

Assembly Implementation

The assembly implementation of quicksort is design to mirror the code in C while leveraging low-level optimization

- Stack management: *asm_function* uses the stack operation to save and restore registers and local variables.
- Recursive Calls: Recursive calls to asm_function handle the left and right partitions of the array. The use of assembly instructions such as mov, cmp, call, and dec/inc is crucial for performing arithmetic and managing recursion.
- Partition Function: The partitioning logic is invoked using a call instruction. The pivot index returned by the partition function determines the boundaries for recursive sorting.

C Implementation

The C implemntation:

- Partitioning: partition() is responsible for selecting a pivot and rearranging the array elements based on the pivot. This function utilizes a linear scan and swapping elements to place them in the correct position.
- Recursion; quicksort() handles the recursive calls for sorting the left and right partitions of the array.

METODOLOGY

Measuring hotspots

Hotspots were identified using a performance profiler to pinpoint the most time-consuming sections of the quicksort algorithm. The main hotspot is the recursive sorting loop, which is executed multiple times as the algorithm divides the array into smaller partitions.

Computer specs.

13th Gen Intel(R) Core(TM) i9-13900H 2.60 GHz // 16GB RAM // Windows 11 Home

Compiler for C: GCC for C and NASM for assemble using MYSYS2

Measuring Execution Time

Execution time was measured using high-resolution performance counters provided by the Windows API. The “QueryPerformanceFrequency” and “QueryPerformanceCounter” functions were used to obtain precise time measurements with microsecond resolution.

Experiment Parameters

- Input Data: Randomly generated integers ranging from 0 to 999,999.
- Array Size: 1,000,000 integers.
- Number of Iterations: 10 runs for each implementation (C and assembly).
- Validation: After sorting, arrays were validated for correctness to ensure that the sorting algorithms produced correctly ordered results.

RESULT

Theoretical Execution Time

The theoretical time complexity for quicksort is O(n logn).
For an array of size 1,000,000, the expected time complexity in terms of operations is approximately 1,000,000 * $\log_2$(1,000,000) which is roughly 20,000,000,000 operations. The actual time depends on implementation details and hardware efficiency.

```
cguti@Charly UCRT64 /c/Users/cguti/OneDrive/Escritorio/UiT/Computer Architecture/assembly-x86-64-mik
-main/assembly-x86-64-mik-main/src
$ make
gcc -Wall -O2 -m64 -g -c main.c
gcc -Wall -O2 -m64 -g -o quicksort_benchmark.exe main.o asm.o

cguti@Charly UCRT64 /c/Users/cguti/OneDrive/Escritorio/UiT/Computer Architecture/assembly-x86-64-mik
-main/assembly-x86-64-mik-main/src
$ ./quicksort_benchmark
Average C QuickSort time: 0.102092 seconds
Average Assembly QuickSort time: 0.106450 seconds
```

The C implementation is faster than the assembly could be due to the optimization of the compilers

DISCUSS

With all the results, is very rare to see C performing better than assembly, because assembly usually is better due to low-level optimizations.

Could be that my code probably has a lot of stacks operations and some other stuff leading to that extra time

To complete this assignment I had the help of ChatGPT by providing me:

- The skeleton of the code, the main structure.
- Some few examples of how should it work and what should do.
- On my report summarizing every aspect of the assignment and make it more easily to read and understand