University of Tromsø

INF-2200: Datamaskinarkitektur Obligatorist innlevering 1

Ahe103

12. February, 2013

1 Introduction

This assignment was meant as a training exercise in Assembly language the step between a high level language and the firmware. It's objective was to make a CPU intensive benchmark, profile said benchmark and then translate the most demanding part to Assembly by hand.

1.1 Requirements

A pre-greenlit algorithm that is proven to be CPU intensive

A profile showing the demanding part

A translation of the demanding part to Assembly

2 Technical Background

2.1 The Stack

The stack can be looked upon as a string with beads thread upon it. When something is added it's laid on top of what was previously there. In order to get at something lower down one need to either count down to what is needed or remove the top until you get at what you wanted. The way to manipulate the stack is by use of the operands PUSH(add) and POP(remove), one PUSHes something on the stack and POP's something off it. The stack is an extremely important structure and improper usage of POP, PUSH or direct manipulation of it will result in a skewed stack which in turn will result in fatal errors in the machine. The Stack is reference-able by the use of Registers which comes next.

2.2 Register

Registers are single entry memory locations within the CPU itself they are the fastest accessible memory locations in the entire computer, therefore they are essential for speedy operation. In the Assembly instruction set that was used in this assignment (x86 AT&T) there are 8 registers total, but only 6 that should ever be trifled with. In the past they were labeled for specific use, but in this day and age that convention has taken a backseat to usability. The Registers names are as follows EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. The 4 first are free to use as one sees fit except EBX as that is callee saved including ESI and EDI meaning they must be saved in some way before being free to use. The last two are the Base pointer and the Stack Pointer, these two are very sensitive and should never be edited unless one knows exactly what to do. The Base pointer points to the base(or bottom) of the stack frame. The Stack pointer points to the top of the stack in the stack fame.

2.3 Stack Frame

A stack frame is a framework that the OS and all other applications use. It needs to be used so that data does not get overwritten. When a stack frame is created, a new Base and Stack pointer is created, but the old ones need to be remembered so that when the stack frame is empty and not in use anymore the old frame can be reestablished.

2.4 Call order

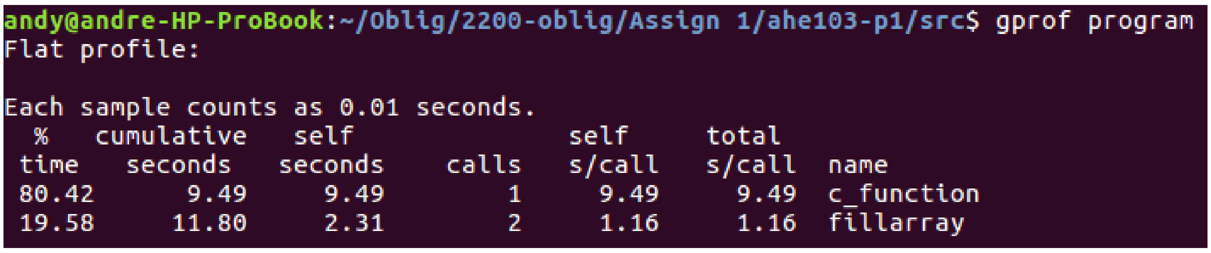
Call order in assembly is very important in Assembly (as with almost any other language). When retrieving variables from the other languages that stored these one wishes to store them in the correct place for use. If the app send in variables A, B and C, if not pulled in the correct order A can become C and B can become A. Therefore its crucial to know the correct order data is sent

3 Design

The assignment called for a CPU intensive algorithm and there were some picked out to be chosen. The choice fell on the Sieve of Eratosthenes as it seemed like quick-to-implement code, the sieve does prime number calculation.

3.1 Finding the demanding part

The sieve itself is a very short snippet of code, just 4 lines of the actual calculation of it. Therefore when taken out of the main function gprof (a profiling program for determining bottlenecks) marked that snippet as the most demanding part.



```
andy@andre-HP-ProBook:~/Oblig/2200-oblig/Assign 1/ahe103-p1/src$ gprof program
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 80.42      9.49     9.49        1     9.49     9.49  c_function
 19.58     11.80     2.31        2     1.16     1.16  fillarray
```

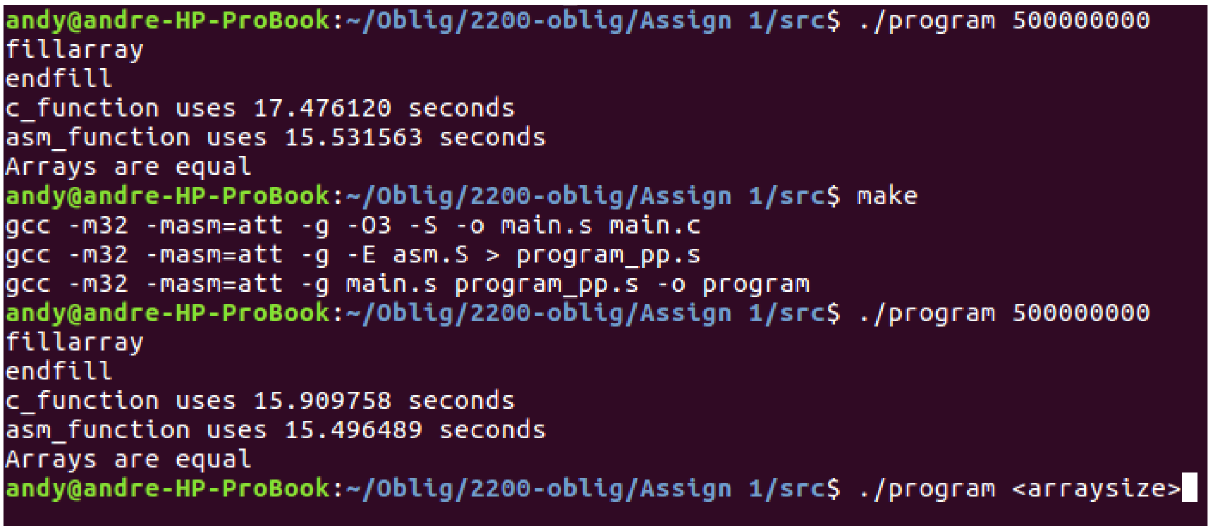
4 Implementation

Since the code is so short the implementation is a rather straight forward process. The 6 registers are more than enough to execute the assembly. No need for accessing memory with pushes and pop's. The first is to decompose the first for loop, then the If test, but a memory access is needed here as the array entry needs to be read. Decompose the second for loop, and lastly another memory access to write something to the array entry. There is only a need for 3 labels for jumps:

1. at the beginning of the first for loop
2. at the beginning of the second for loop
3. after the end of the second for loop (this is used if the If test fails, therefore the second for loop should not be performed for that integer)

5 Discussion

Seeing as a compiler uses defined algorithms to complete certain functions it might be very underoptimized and therefore rather slow in code execution. As that is the case writing the assembly tailored for this program it's able to beat even the strictest of optimization rules the GNU compiler had to offer. It might be a fluke but several runs have coloured the opinion that it is in fact the case



```
andy@andre-HP-ProBook:~/Oblig/2200-oblig/Assign 1/src$ ./program 500000000
fillarray
endfill
c_function uses 17.476120 seconds
asm_function uses 15.531563 seconds
Arrays are equal
andy@andre-HP-ProBook:~/Oblig/2200-oblig/Assign 1/src$ make
gcc -m32 -masm=att -g -O3 -S -o main.s main.c
gcc -m32 -masm=att -g -E asm.S > program_pp.s
gcc -m32 -masm=att -g main.s program_pp.s -o program
andy@andre-HP-ProBook:~/Oblig/2200-oblig/Assign 1/src$ ./program 500000000
fillarray
endfill
c_function uses 15.909758 seconds
asm_function uses 15.496489 seconds
Arrays are equal
andy@andre-HP-ProBook:~/Oblig/2200-oblig/Assign 1/src$ ./program <arraysize>
```

6 Conclusion

The assignment requested a CPU intensive algorithm that taxed the CPU to 100% capacity, doing so the Sieve of Eratosthenes was chosen for ease and suitability. Profiling found the hotspot, the hotspot C code was decomposed to assembly. The assembly code ran without hitch giving a sleeker execution than the GNU C compiler

In conclusion the assignment was completed and the assembly code runs without hitch. Giving knowledge of how to correctly parse C code into assembly oneself and giving experience in doing so. Had a different algorithm than the sieve been chosen the assignment might not have been completed in time.