INF-2200 Datamaskinarkitektur og -organisering INF-2200

Alexander Saaby Einshøj

September 23, 2015

1 Introduction

In this assignment, we were to select a micro-benchmark and implement it as a function in x86 assembly.

2 Technical Background

This assignment requires some basic knowledge of assembly and its basic operations for getting values from memory, moving between registers and pushing and popping to/from the stack. Also, some knowledge about the GAS AT&T x86 instruction set syntax is required.

2.1 Assembly

The assembly languages are low-level programming languages where the code written is translated through an assembler to machine code. The assembler is what allows us programmers to write assembly language with letters instead of having to write entire programs in pure numbers to be directly understood by the computer. When code is read by the assembler, each line is converted to CPU-level instructions.

2.2 x86

The x86 instruction set is an ISA(Instruction Set Architecture) which defines what operations we are allowed to use.

X86 is a 32-bit architecture, which means that we have 2^{32} different memory locations to store data. Programming in the assembly language is mainly about moving numbers around. To do this efficiently, we have access to registers. Registers are temporary storing units, which are extremely useful when programming in assembly. Using the registers to store/retrieve data only requires a couple of clock cycles by the CPU, as opposed to using memory instead, as it uses hundreds of cycles to complete the equivalent task.

In x86, there are eight general purpose registers, where I will only be mentioning the 32-bit versions of these. These are: EAX, ECX, EDX, EBX,

ESP, EBP, ESI and EDI.

Two worth describing are EBP and ESP. EBP is the stack base pointer used for hilding the address of the current stack frame. This means we can use the base pointer as a point of reference when pushing values on the stack. ESP (stack pointer) refers to the top address(address with the lowest value) of the stack as is.

The rest of the registers do have purposes of their own, but most of them can be used to what you want, and are not necessarily to be used as their respective names immediately indicate.

2.3 cdecl

The cdecl is a calling convention used a lot for the x86 architecture. It is specified that values and memory addresses are returned in the EAX register. In other words, any value you want to return in a function should be stored in EAX. The registers EAX, ECX, and EDX are caller-saved, that is they are free to be used without pushing anything onto the stack, while the rest of the registers are callee-saved. If a register is callee-saved, their old values need to be stored before you yourself can store values in them.

3 Implementation

The implementation of the heap sort was implemented in C, then a micro-benchmark was implemented in assembly. The syntax used was GAS *AT*&T x86 assembly.

4 Discussion

This assembly implementation could have been done in a number of different ways, and I chose to translate the C function rather directly into assembly code, line by line. In my first attempt, I used one register more than I do now, which I found to be unnecessary. After achieving this, I tried reducing the amount of callee registers used even more, by only pushing one onto the

stack. This turned out to be too difficult using the time given for the assignment, but the end result still did well.

The amount of memory used in my implementation is minimal, and I strongly believe it would be quite hard to improve at this point, as the benchmark implemented isn't all that big, leaving little room for doing it any other way.

As the chosen micro-benchmark is the sift-down function in the heapsort algorithm, it's a quite commonly used benchmark throughout the algorithm that uses alot of memory, therefore it's interesting to see how much improvement can be made with these simple lines of code. The implementation has been tested against a corresponding C-function at O0, O1, O2 and O3. The goal from the beginning was ultimately to beat the O3 optimalization of gcc, although this was harder than prospected. I believe that it could have been done, if I could manage with one register less, and i'm sure this is quite possible, although it requires alot of thinking and planning.

Perhaps the biggest issue I encountered while trying not to push EDI, was the very last bit of the benchmark, where in the C-code it does the SWAP. It is indeed possible to do the swap in this situation with only 3 registers other than the index of the array, EAX. The problem arises when I need the values of two of these registers from before the SWAP. At this point, I think as a general rule, one should have 4 available registers for performing this operation. This is because if you try to ommit one register, the difficulty of the operation multiplies manyfold what it previously was.

Another thing to notice, is the references I use for EBP, which are the three arguments passed onto the called function. The last argument, 'end', is put inside the loop, which means that it may be used multiple times in a single function call, as the other arguments are not. This could have been improved by omitting the need of resetting the register every time. The 'end' argument doesnt change throughout the loop of the implementation in the C code, and therefore it is needed to have a reference to this at all times.

The table below shows the results when sorting 9 elements 1 billion times, and the numbers below are the average time after doing this ten times for each level of optimalization.

Looking at table 1, we see that the assembly implementation is faster than the C implementation in the case of O0, O1 and O2. The O3 optimalization beats the assembly implementation by a small margin, possible because of my use of memory in the implementation. Since these tests were done with only 9 elements to sort, the numbers might have been different if I upped the amount of elements and lowered the run amount.

When running the program 1 billion times, it will go in and out of the function a billion times, which might take some time. This means that the numbers below might not be 100% accurate, but achieving anything that accurate would be very difficult.

Table 1: asmfunction vs Cfunction at different gcc optimalization levels

Optimalization	asmfunction	Cfunction	Difference
O0	93.736 s	169.838 s	44.81%
O1	68.168 s	75.64 s	9.88%
O2	67.888119 s	72.950607 s	6.94%
O3	61.960330 s	59.708731 s	-3.77%

5 References

[1] www.rossetacodes.org/wiki/Sorting_algorithms/Heapsort#C