Architecture: assembly and compiler optimisation

In this practical you will undertake various tasks related to understanding the assembly language of x86-64, in the AT&T syntax (also known as GNU Assembler Syntax, or GAS). You will be analysing both unoptimised and optimised assembly.

For the highest marks, you will also be required to analyse a piece of assembly of the ARMv8 architecture (or to be precise AArch64, which is the 64-bit execution state of ARMv8). This will involve independent study.

This practical is worth 25% of the continuous assessment portion of this module and is due 21:00 on Wednesday Feb 26th.

Background

You are supplied with C source code to do integer exponentiation, in `pow.c`, `main.c`, and `pow.h`. The provided `Makefile` compiles these to an executable `main`, as well as to several assembly files of x86-64, with varying degrees of optimisation, with `pow0.s` being unoptimised and `pow3.s` being most optimised.

(For the assembly files to be of the expected kind, you need to run the `Makefile` on a machine with a x86-64 architecture, preferably one of our lab machines running Fedora.[1])

This assignment is broken up into four parts, which are best done in the indicated order.

1 Analysing unoptimised assembly code

You are to provide a well commented version of the assembly compiled for function `power` in `pow.c`, to help a human reader understand it.

- After running `make`, make a copy of `pow0.s` called `pow0-commented.s` and add code commenting to the instructions. (Be very careful to enter your comments into the copy `pow0-commented.s`, rather than into `pow0.s`. The latter could be overwritten by a subsequent run of `make`.)

[1] If you want to do this practical on your personal laptop, and your laptop does not have (a recent version of) `clang`, then at least run `make` on the lab machines before completing the rest of the assignment on your laptop. A compiler other than `clang` may produce entirely different assembly.

- The comment character in assembly is `#`. From this character onward, the rest of the line is ignored by the assembler.
- Comments should come after instructions on the same line. If necessary, you can add additional lines with no instructions to extend a comment.
- Your commented file should not change the assembly code at all, so that it could still be used as input to the assembler.
- Each assembly language instruction for the function `power` should have some kind of comment. This comment needs to indicate succinctly what the purpose of the instruction is *in its context*, for example by referring to variables or other expressions or statements in the C program, or by relating the instructions to aspects of the calling convention. Some comments may be as short as for example `# %eax = x+y`, as pseudo-formal notation to mean that the purpose of the instruction is to put `x+y` in register `%eax`, where `x` and `y` would be variables from a source C program.
- You do *not* need to comment on assembler directives, whose names start with a period.

After studying the assembly, you should be able to determine the structure of the stack frames, and describe your findings in the report. By means of a table or list of bullet points, list all elements of the stack frames of function `power`, with their index relative to the base pointer. Which values are stored where? Where is the return address stored, and where the saved base pointer? Where are 64-bit values used and where 32-bit values? Are any bytes in the stack frames unused? If so, which, and why are these bytes unused?

2 Recursion to iteration

Now study `pow1.s`, which was obtained by optimisation level 1. You will see that recursion was turned into iteration. Write a new C file `pow-iter1.c` with an implementation of `power` that computes the same function as that in `pow.c`, but now using iteration, matching `pow1.s` as close as possible (but avoiding 'goto' statements); there may be more than one correct solution.

Add lines to the `Makefile` to compile `pow-iter1.c` to an assembly file `pow-iter1.s` and object file `pow-iter1.o` with optimisation level 1. Further add lines to the `Makefile` to link `pow-iter1.o` and `main.o` to create the executable `main-iter1`. Run this code; it should give the same output as `main`.

Is `pow-iter1.s` identical to `pow1.s` ? In the report, describe the differences, if there are any. You can also describe any other interesting observations you made about `pow1.s` and `pow-iter1.s`.

3 Loop unrolling

Now study `pow2.s`, which was obtained by optimisation level 2. You will see this again uses iteration, but the loop is furthermore 'unrolled'. As before, write a new C file `pow-iter2.c` with an implementation of `power` that computes the same function, but now matching `pow2.s` as close as possible (once more avoiding 'goto' statements).

Add lines to the `Makefile` to compile `pow-iter2.c` to object file `pow-iter2.o` (with any optimisation level). Further add lines to the `Makefile` to link `pow-iter2.o` and `main.o` to create the executable `main-iter2`. Run this code; it should give the same output as `main` and `main-iter1`.

In the report, explain the structure of control flow in `pow2.s`. Why does this compute the same function? Why has the compiler chosen to do loop unrolling? Do you see any disadvantages to doing loop unrolling? Did you find any other interesting use of addressing modes in `pow2.s` that you want to comment on?

Some hints:

- It might not be easy or possible to explain everything that compiler optimisations do. But you should be able to at least reflect on the nature of some of the optimisations and reason about impact they might have on performance.
- The structure of control flow of loops in assembly is often best described using do-while loops, whereas human programmers might normally prefer ordinary while or for loops, whose logic is generally easier to understand.
- You might try to compile `pow-iter2.c` to assembly with optimisation level 2, but you might find that this is not quite what you expected. It is not required to study the assembly thus obtained.

4 ARMv8

Now study the provided file `pow-arm.s`, which is assembly of a very different architecture. It was also compiled from `pow.c` (but not by the provided `Makefile`). In order to understand this, some independent study may be required. A good starting point is `https://modexp.wordpress.com/2018/10/30/arm64-assembly/`.

Describe your findings in the report. This may include, but is not limited to:

- What do the various instructions in `pow-arm.s` do? (You could start by code-commenting the assembly as in Part 1 above.)
- What addressing modes are used and what do they mean?
- How do you see the 64-bit ARM calling convention reflected in the assembly?
- Where are 64-bit values used and where 32-bit values?

- What is the layout of the stack frames of the `power` function? Do you notice any interesting differences with the stack frames you found in Part 1?

Submission

Submit a zip file containing your report as a PDF and a directory of code, including all the provided files, as well as `pow0-commented.s` with the code commenting that you created, and the extended `Makefile`.

Marking

0-9 Work that fails to demonstrate an understanding of the meaning and purpose of assembly.

10-14 Completion of Parts 1, 2 and 3, explaining the purpose of each instruction in its context, but failing to express good understanding of overarching principles such as stack frames, calling conventions, and compiler optimisations.

15-17 Completion of Parts 1, 2 and 3, demonstrating thorough understanding of the instructions, as well as the overarching principles.

18-20 Completion of all 4 parts to a high standard.

Rubric

There is no fixed weighting between the different parts of the practical. Your submission will be marked as a whole according to the standard mark descriptors published in the Student handbook at:

> `https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#General_Mark_Descriptors`

You should submit your work on MMS by the deadline. The standard lateness penalties apply to coursework submitted late, as indicated at:

> `https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`

I would remind you to ensure you are following the relevant guidelines on good academic practice as outlined at:

> `http://www.st-andrews.ac.uk/students/rules/academicpractice`