Architecture: assembly and compiler optimisation

Due **21:00 on Wednesday Feb 16th, 2022**.
(As always MMS is definitive on deadline)

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

Background

There's a famous problem called the 'Collatz Conjecture' – often called the '3n+1 problem' – concerning simple arithmetic steps which lead to surprisingly deep maths. Take an integer: if it's even, divide by 2; and if it's odd, then multiply by 3 and add 1. Repeat until you end up with the number 1 (because a few steps previously you had a power of 2) or perhaps keep going forever. The Collatz Conjecture is that this process always ends with 1 for all positive integers. For more information see `https://en.wikipedia.org/wiki/Collatz_conjecture`, but understanding of the problem is not required for this practical.

You are supplied with recursive C source code which computes a sequence of steps in the generalised problem where each of the constants 2, 3 and 1 can be replaced by arbitrary integers. The code is supplied in `collatz.c`, `main.c`, and `collatz.h`.

The provided `Makefile` compiles these to an executable `main`, as well as to several assembly files for both x86_64 and ARM64 architectures, with degrees of optimisation. We will focus on optimisation levels 0 and 2 (for x86) and 1 (for ARM). The `Makefile` (run on lab machines) creates an executable program `main` as well as compiled assembly code for x86_64 and arm64. Because not all versions of `clang` produce the same assembly, I have provided three files[1] which are copies of the relevant files created by by running the `Makefile` on `palain`. Please use these files in what follows, to ensure everyone does the same work and marking is consistent.

This assignment is in three parts, which are best done in the indicated order.

[1] `collatz0-x86-commented.s`, `collatz2-x86-commented.s`, and `collatz1-arm-commented.s`

1 Analysing unoptimised assembly code

For this part you are required to comment the assembly code for function `collatz_recurse` in the file `collatz0-commented.s`, to help a human reader understand it. There is no need to add comments for assembly code for the function `collatz` in the same file or for any code in `main`.

The key goal of this practical is to demonstrate understanding of how C code relates the the assembly that it compiles to, so bear this in mind throughout your commenting.

- 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.
- Do not change the assembly code at all, please just add comments to what is there.
- Each assembly language instruction for the function `collatz_recurse` 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.
- You may see instructions you haven't seen in the lectures. A web search may help to find more information.
- In unoptimised code, beware that some instructions may not do anything useful in their context. Such instructions are generally removed by compiler optimisations (as you will see in Parts 2 and 3 below).

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, diagram, or list of bullet points, list all elements of the stack frame of function `collatz_recurse`, 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 or 16-bit or 8-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 `collatz2-commented.s`, which was obtained by optimisation level 2. You will see that recursion has turned into iteration. You are to add comments for the assembly code for function `collatz`.

Include in your report a diagram/table/list of the stack frame for function `collatz` in this file, answering the same questions about the stack frame as asked about the stack frame in Part 1. You can also further explain some of the observed compiler optimisations in the report. Explain, for example, why the function `collatz_recurse` has completely disappeared from the assembly version. The code commenting and the accompanying text in the report should convince the marker that you understand why the code does the same computation as before.

3 ARMv8

Now study the provided file `collatz1-arm-commented.s`, which is assembly of a very different architecture. It was also compiled from `collatz.c`, with optimisation level 1. 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/`. A quick reference is `https://courses.cs.washington.edu/courses/cse469/18wi/Materials/arm64.pdf`.

Add comments for the assembly code for function `collatz-recurse`. Comments now start with `//`. Describe in more detail in the report what you have learned about the ARMv8 architecture as it relates to this code. This may include, but is not limited to:

- What do the various instructions in `collatz1-arm-commented.s` do?
- 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 or 16-bit or 8-bit values?
- What is the layout of the stack frames of the `collatz_recurse` function? Do you notice any interesting differences with the stack frames you found in Part 1?

Submission

Submit a zip file containing:

- versions of `collatz0-x86-commented.s` and `collatz2-x86-commented.s` extended with your code commenting,
- if you completed Part 3, a version of `collatz1-arm-commented.s` extended with your code commenting, and
- your report as a PDF file.

Marking

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

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

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

18-20 Completion of all three 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`