Architecture: x86 Assembly

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). This will include adding comments to some assembly, and using inline assembly in a C program, to let it print stack frames.

This practical is worth 28% of the continuous assessment portion of this module and is due 21:00 on Wednesday Feb 27th.

Background

You are supplied with C source code that converts a given integer to a string that is its decimal representation. The code is complicated by wanting to avoid any prior assumptions about the maximal length of the string, allocating space (by malloc) only after the length of the string has been determined through a sequence of recursive calls.

We first look at two provided C files, `itos.c` and `main.c`, and one header file `itos.h`. The provided `Makefile` compiles these to produce an executable program `main`, as well as several assembly files, with varying degrees of optimisation, with `itos0.s` being unoptimised and `itos3.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.) Two further provided C files will be explained later.

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

1 Commenting assembly code

Your task is to provide a well commented version of the assembly compiled from function `itos_recur` in `itos.c`, to help a human reader understand it.

- After running `make`, make a copy of `itos0.s` called `itos0-commented.s` and add code commenting to the instructions belonging to the `itos_recur` function. (Be very careful to enter your comments into the copy `itos0-commented.s`, rather than into `itos0.s`. The latter could be overwritten by another run of the Makefile.)

- Comments should come after instructions on the same line. If necessary, you can add additional lines with no instructions to extend a comment.
- The comment character in assembly is `#`. From this character onward, the rest of the line is ignored by the assembler.
- Your commented file should not change the assembly code at all, so that it could still be used as input to compilation of the program.
- Each assembly language instruction for the function `itos_recur` 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 = a+b`, as pseudo-formal notation to mean that the purpose of the instruction is to put `a+b` in register `%eax`, where `a` and `b` are variables in the original C program.
- You do *not* need to comment on assembler directives, whose names start with a period.

2 Stack frames

Your task is to use inline assembly to print stack frames corresponding to the `itos_recur` function, to confirm your analysis from Part 1, and to write up your findings in a report. To help you with implementation, a second version of `itos_recur` is provided, in `itos-stack.c`, which only differs in an added call of `print_stack`. Function `print_stack` is implemented in `stack.c`, using auxiliary function `print_recur`. What remains for you is to complete the implementation of `print_recur`. The code as provided can be executed by calling `main-stack`.

Inline assembly consists of snippets of assembly inserted into C code, typically with references to C variables to allow information to flow between the C code and the assembly. For example, if the C code contains:

```
long rbp;
asm("movq %%rbp, %0;" : "=r"(rbp));
```

then this is translated in the assembly as a `movq` from the value in the `%rbp` register to a location that corresponds to C variable `rbp`. The C program can subsequently use that value.

The above inline assembly has only 'output'. An example of 'input' combined with 'output' is:

```
long val1 = 1000;
long val2;
asm("movq (%1), %0;" : "=r"(val2) : "r"(val1));
```

The `%0` and `%1` can be seen as placeholders for arguments of the assembly instructions, with the indices `0` and `1` referring to `"=r"(val2)` and `"r"(val1)`, in this order. The round brackets in `(%1)` mean indirection as usual. This means that we copy the value in the address held in a register, and this register is tied to C variable `val1`. The destination of the copy is a register that is tied to C variable `val2`. As a result, this inline assembly instruction copies the quadword at the memory address in C variable `val1` to C variable `val2`. (Of course, trying to run the above verbatim, with constant 1000, will most likely lead to a segmentation fault.)

If you Google for `inline assembly` or `inline assembler`, then you will be able to find much relevant documentation. However, this practical does not need much more than the two kinds of statement with `asm` that were exemplified above.

You may now want to study `stack.c`. You will see that C variables that are to contain quadwords are declared to be of type `int64_t`. Even though this is typically the same as `long` on 64-bit machines, the former is safer. There are corresponding types for 32-bit and 16-bit values.

Now turn to extending `print_recur`, of which the purpose should be to print out the contents of the stack frames. You can print out all quadwords within a stack frame, but realise that some values stored in the stack may be doublewords, or even (single) words, for which slightly different inline assembly instructions are required from the ones above.

The formatting of the output is up to you, but an obvious choice would be to print next to each value in the frame also:

- its absolute address,
- its offset relative to the basepointer,

Write a report about your findings, describing the values you found in the stack frames. Do they correspond to your analysis in Part 1? Are there any values that you cannot explain? Of how many bytes does a stack frame consist, and which are not used, and why are they not used? Can you tell from the printed values whether x86-64 is little-endian or big-endian?

To the report you may also add general reflection on Part 1; perhaps there were some observations you made about the instructions that could not conveniently be put in the code commenting. You could also reflect on any conversion between words, doublewords and quadwords, and where and how this happens in the annotated assembly.

It is conceivable that the added call of `print_stack` in `itos_recur` would lead to global changes to the assembly implementing this function, possibly changing the layout of the stack frames. Confirm this or rule this out, by comparing `itos0.s` and `itos-stack.s`.

3 Division by invariant integers using multiplication

After systematically commenting each line of `itos_recur` in `itos0.s`, there is not much merit in doing more of the same and commenting each line of `itos1.s`, `itos2.s`, or

`itos3.s`. However, we can investigate a few optimisations in these files. Here we consider so-called "division by invariant integers using multiplication".[1]

The idea is as follows. Integer division generally requires more clock cycles than integer multiplication. So instead of dividing a 64-bit number n by 10, we want to multiply n by some 64-bit number X, which results in a 128-bit number, part of which is the sequence of 64 bits that is the desired quotient. To simplify the discussion, we initially assume n is non-negative.

First some notation. For a real number z, the *floor* $\lfloor z \rfloor$ is the greatest integer smaller than or equal to z, and the *ceiling* $\lceil z \rceil$ is the smallest integer greater than or equal to z. In other words, these are the operations of rounding down and rounding up, respectively. Our goal is to compute $\lfloor n/10 \rfloor$ with a multiplication instruction. Our first attempt is to evaluate the formula:

$$\lfloor \lceil 2^{64}/10 \rceil * n/2^{64} \rfloor \tag{1}$$

We might hope that the number $\lceil 2^{64}/10 \rceil$ could perhaps serve as the integer X referred to above. Further note that the operation $\lfloor \ldots /2^{64} \rfloor$ can be done very efficiently, taking the 64 most significant bits.

Regrettably, this does not quite work. In taking the ceiling, the rounding-off error becomes too big and the difference with $\lfloor n/10 \rfloor$ may be 1 or more for some $n < 2^{63}$. To see this, consider the error:

$$\begin{aligned}
&\lceil 2^{64}/10 \rceil * n/2^{64} - \lfloor n/10 \rfloor = \\
&\lceil 2^{64}/10 \rceil * n/2^{64} - n/10 + n/10 - \lfloor n/10 \rfloor = \\
&(\lceil 2^{64}/10 \rceil - 2^{64}/10) * n/2^{64} + (n/10 - \lfloor n/10 \rfloor)
\end{aligned}$$

We have $2^{64} \equiv 6 \pmod{10}$, hence $\lceil 2^{64}/10 \rceil - 2^{64}/10 = 4/10$. Moreover, $n/10 - \lfloor n/10 \rfloor$ can be up to $9/10$, for $n \equiv 9 \pmod{10}$. For $n = 2^{63} - 1$ this gives us an upper bound on the error, which is $4/10 * (2^{63} - 1)/2^{64} + 9/10 \approx 1.1 \geq 1$.

On a second attempt and third attempt, we might try instead to evaluate:

$$\lfloor \lceil 2^{65}/10 \rceil * n/2^{65} \rfloor \tag{2}$$

or:

$$\lfloor \lceil 2^{66}/10 \rceil * n/2^{66} \rfloor \tag{3}$$

In your report, explain why formula (2) has the same problem as formula (1), and why formula (3) does give us $\lfloor n/10 \rfloor$ as desired. Further explain which instructions are used to realise formula (3) in `itos1.s` for non-negative n. Then explain what is done in the assembly to handle negative n.

[1] `https://gmplib.org/~tege/divcnst-pldi94.pdf`

4 Extensions

Further analyses of optimisations observed in `itos1.s` are seen as extensions. (There is little else of interest in `itos2.s` and `itos3.s`.)

You may also find out how to compile to the assembly of other architectures using `clang`. Add the necessary lines to the Makefile. Interesting architectures are for example MIPS and ARM (which come in a number of variations). Study their assembly for `itos.c`, and describe the differences with the x86-64 assembly. You might find it interesting to consider in particular the different calling conventions for these architectures and how they affect the produced assembly for `itos.c`.

(Naturally, the assembly for other architectures than x86 cannot be compiled into a form that is directly executable on the lab machines. But you might be able to use simulators for some architectures.)

Submission

Submit a zip file containing your report as a PDF and a directory of code, including all the provided files, with the completed implementation of `print_recur` in `stack.c`, and with the file `itos0-commented.s` with the code commenting that you created. Extra lines may have been added to the Makefile if you did any extensions.

In your report, the customary sections on testing, design and implementation would likely be short, as only Part 2 involves a few lines of code of your own making. Instead, the emphasis should be on demonstrating your understanding of assembly, of calling conventions, and of compiler optimisation.

Marking

0-9 Work that fails to demonstrate good understanding of the relation between the assembly and the C code from which it was obtained.

10-15 Informative code commenting explaining the purpose of each instruction in its context (Part 1), a working implementation of `print_stack` (Part 2) that was tested on the lab machines, and a report discussing the relevant issues in a clear and informative manner, demonstrating thorough understanding.

16-17 Completion of Part 3, in addition to the above, demonstrating thorough understanding of the principle of "division by invariant integers using multiplication" and its realisation in assembly.

18-20 At least some extensions from Part 4, in addition to all of the above.

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`