Week 11 Lab

CC2511

Task Description

You'll write a fast implementation of an average function for 32-bit integers in assembly code. You will first implement the function in C, and then you will implement the same function in assembly code. The objective is to make the assembly code version as fast as possible.

The operation that you must implement is Eq. (1), in which there is repeated addition and a single division:

$$\text{(1)} \quad \text{average}(\mathbf{x}) = \left\lfloor \frac{\sum_i (\mathbf{x}_i)}{\boldsymbol{n}} \right\rfloor,$$

amount of numbers have to be a power of 2

$$n = 2^q, 0 \leq q < 16$$

where **x** is an input array containing **n** items and the index i loops over all the items in the array. Furthermore, all members of **x** are 32-bit integers and **n** is a power of 2 less than 2^16, i.e. 2, 4, 8, 16, ... , 32768. The result is an integer which is the floor of the real-number average, i.e. the largest integer that is less than or equal to the exact result. Examples are shown in the table below.

rounded down

x	q	result	real
[1,2,3,4]	2	2	2.5
[1,2,3,6]	2	3	3
[1,2,3,4,5,6,7,8]	3	4	4.5
[1,2,3,4,5,6,7,12]	3	5	5

Your task is to implement Equation (1) with 32-bit inputs and a 32-bit output, creating two functions whose C declarations are as follows:

```
extern uint32_t c_average(uint32_t Q, uint32_t* values);
extern uint32_t asm_average(uint32_t Q, uint32_t* values);
```

The input array contains 32 bit unsigned integers. To avoid overflows, your intermediate total must be more than 32 bits long. You can declare a 64 bit integer in C using

why?

```
unsigned long long int accumulator = 0;
```

and in assembly by splitting the 64 bit result over two registers.

A starter project is provided on LearnJCU that includes an accuracy and speed test for an array of 128 integers (i.e. Q = 7). Download

and unzip this zip file, then copy the Lab11 folder to your local Git repository. Do not modify the `main.c` file. Write your implementations in the `c_average.c` and `asm_average.s` files.

Motivation

Many processing systems require a statistic for a set of integers - for example in machine learning applications deployed to small embedded devices with no floating-point support.

The techniques used in this example are typical of use cases where developing in assembly code is a worthwhile investment.

Hints

- It's recommended that you implement the C version first.
- When you add two 32-bit unsigned numbers together, your result may overflow, setting the carry flag. You can then keep track of the number of times the result overflows.
- Consuder what happens when you shift a binary number to the right by n places. How does this compare to dividing by 2^n?
- For the bit shifts, use the "logical shift left" and "logical shift right" instructions (`lsl` and `lsr`). These work with *unsigned* integers, in contrast with "arithmetic shifts" that work with signed integers.
- You will need to think about how to shift a 2 x 32-bit (i.e. 64-bit) intermediate result into a 32-bit integer. You might find it helpful to draw a diagram showing the sequence of bits in the two 32-bit numbers and how these are split across two registers. On your diagram, you can show how the two parts of this number need to be combined to produce the final result.

Assessment

A completed version of this project might look like the screenshot below.

To complete this lab task, you must demonstrate:

- A handwritten assembly function that calculates the same result but attempts to outperform the C implementation.

Optional extension task

- Make your assembly code as fast as possible. What's the best time that you can achieve?
- Once you have both versions working, try adjusting the compiler optimisation level (F1 > CMake: Select Variant > Release). After changing the compiler settings, Clean the project (F1 > CMake: Clean Rebuild) to force a full re-build. Can you beat an optimising compiler?