ECE4703 Laboratory Assignment 4

The goals of this laboratory assignment are:

- to familiarize you with assembly language programming and code optimization on the TMS320C6713,
- to allow you to experimentally try various coding strategies to best use the pipeline and functional units available on the TMS320C6713,
- to reinforce your understanding of the profiling capabilities of CCS.

Problem Statement

There are often cases where C code is unable to achieve the required performance for a timing-critical application. In these cases, assuming you have already done your best to optimize the data types and the memory usage, you have a couple options. You could use the optimization capabilities of the CCS C compiler to improve your code performance or you could optimize your code by writing critical functions in *hand-optimized assembly language*. While writing your own assembly code can be difficult for complicated algorithms, it is a reasonable approach for simple algorithms. Writing assembly language code also allows you to directly control the operation of the DSP and to get a much better feeling for the sensitivity of its performance to various factors including pipeline efficiency and functional unit efficiency.

In this lab assignment, you will modify your DF-II IIR filter code written for Laboratory Assignment 2 to improve its efficiency. Specifically, you will modify and functionalize your DF-II single-section (not SOS) C code from Laboratory Assignment 2 to calculate the output of a Direct Form II IIR filter with arbitrary order and arbitrary single-precision floating point coefficients stored in global arrays. You will then also write a C-callable assembly language function to perform the same function, hopefully performing the calculations with less cycles. You will then compare the performance of this C-callable assembly function to the performance of your C function for a 24th order IIR filter.

Part 1: Single-Precision Floating-point DF-II IIR filter C function

In Laboratory Assignment 2, you wrote C code to realize a real-time floating point DF-II IIR filter. In this assignment, you will functionalize your DF-II IIR filter code from Lab 2 and modify the code (if necessary) to allow for any filter order. In part 1 of this assignment, you will create a new project (probably based closely on a project from Laboratory Assignment 2) to realize a DF-II IIR filter function with prototype

```
short iirdfii(short,short)
```

that you will call from your ISR. Your function should accept the following inputs:

- Current input, 16-bit short
- Filter order, 16-bit short

The feedback coefficients, feedforward coefficients, intermediate values, and intermediate value index are all assumed to be globals (the first three are arrays of floats, the last is any integer datatype you wish). Your function should cast the input to an appropriate datatype, update the intermediate value buffer (by this point, your code should be “updating an index” and not “moving the contents of the buffer”), compute the new 32-bit single-precision float intermediate value

$$u[n] = x[n] - \sum_{i=1}^{N} a[i]u[n-i]$$

compute the output

$$y[n] = \sum_{i=0}^{N} b[i]u[n-i]$$

where N is the filter order, cast this result as a short, and then return this value.

Using your filter coefficients from Lab 2, confirm that your filter is working correctly and that it satisfies all of the requirements stated above. Profile your function (using the C6713 Cycle-Accurate Simulator) for a 24th order IIR filter (there will be 25 coefficients each in the numerator and denominator) with and without various levels of optimization and note the values in your report. These values establish a baseline by which you will compare your hand-coded assembly language DF-II IIR filter function.

Part 2: Single-Precision Floating-point DF-II IIR filter ASM function

Modify your project from Part 1 by adding a C-callable assembly language function that has the same prototype (with a different function name, of course) and performs the same task as your C function. Do not use linear assembly or inline assembly language. Your function should be written entirely in standard TMS320C6x assembly language and you are permitted to use any valid commands and directives in the programming guide. Please comment your ASM code liberally, both to aid debugging and to help the grader understand what you are doing.

Note that the inputs/output will be passed into/from your ASM function via registers as described in your textbook and the lecture notes. Your code will need to load the feedforward coefficients, feedback coefficients, intermediate values, and the intermediate value index from memory, e.g. using LDW or LDDW commands, into appropriate registers. In the process of computing the output $y[n]$, your our code will also compute the new intermediate value. This means that your code will need to write the newest intermediate value ($u[n]$) to memory from a register, e.g. using a STW or STDW command, and update the intermediate value index.

As part of your assembly language programming, you are allowed to specify which instructions are to be grouped into one *execution packet* via the parallel bars ||. You are also allowed to specify which functional unit should execute each command (recall the 8 functional units available in the TMS320C6713). You are encouraged to try various approaches to this problem to minimize the number of execution packets and the number of clock cycles required to execute your function function.

You can confirm that your ASM function is working correctly by calling both the C function and the ASM function with the same inputs (you may need to have two intermediate value buffers since each function modifies this buffer) and confirming that they produce exactly the same outputs. There should be no difference between these functions except execution speed.

Bonus points

To encourage you to put some time into ASM code optimization, bonus points will be available for this assignment as follows:

- 10 bonus points will be given in this assignment to each team that writes a C-callable assembly language function that executes in 90% or less of the cycles required by the unoptimized C function. In other words, if your C function takes 1000 cycles to run, you will get the bonus points if your ASM function runs in 900 or less cycles (and works correctly).
- 20 bonus points will be given in this assignment to each team that writes a C-callable assembly language function that executes in less cycles than their *fully-optimized* C function.
- An additional 20 bonus points will be given to the team that successfully implements their ASM function in the least number of cycles and meets all of the requirements of the assignment.

All bonuses will be based on profiling results on the C and ASM functions as measured by the grader. Your C and ASM functions have to work correctly and have to satisfy all of the requirements of the assignment to be eligible for bonus points.

In Lab

You will work with the same lab partner as in the prior laboratory assignments. Please contact the instructor if your lab partner has dropped the course or if you have concerns about your lab partner's performance on the prior assignments.

Suggested Procedure for Software Design

1. Begin by familiarizing yourself with assembly language programming for the TMS320C6x. There are many new instructions to learn and you should be sure to keep in mind the different functions for floating point operations versus integer/fixed-point operations. You will probably need to refer to the TMS320C6000 Programmer's Guide and/or Instruction Reference for the full details on certain instructions.
2. You may want to look at the assembly language produced by CCS for your C function in Part 1. This may give you some ideas on the types of instructions you will need to realize your ASM function.

3. Get your C function working first. It will probably be very difficult to troubleshoot your ASM function if you do not have the C function working first.

4. Don't worry about parallelizing/optimizing your ASM code in the beginning. Just get your ASM function working correctly. You can set breakpoints in your assembly code and also view the contents of registers (via the "View" menu) to facilitate troubleshooting.

5. Once you have your assembly language function working and you've fully tested it, think carefully about how to put instructions in parallel to maximize parallel processing (decrease the number of execution packets per fetch packet). Can you reorder instructions to avoid resource conflicts as well as avoid data and branch hazards? Can you perform useful tasks rather than inserting NOPs?

   To help with optimization, it is highly recommended that you draw some flowcharts and dependency/pipeline diagrams in this step. These should be included in your report.

Specific Items to Discuss in Your Report

Your report should focus primarily on your approach to developing an efficient ASM function for DF-II IIR filtering. There is a lot of room for analysis (pipeline usage, functional unit usage, data hazards, ...) and discussion here. You should discuss the profiling gains you were able to make with respect to the C function you wrote in Part 1 and use graphics appropriately to make key points.