ECE4703 B Term 2009 Laboratory Assignment 4

Project Code and Report Due by 3:00pm 03-Dec-2009

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 single-section IIR filter code written for Laboratory Assignment 3 and compare the performance of unoptimized and optimized C code to your hand-optimized assembly language code. In the first part of this assignment, you will modify your DF-II single-section IIR filter code from Laboratory Assignment 3 to use 16-bit fixed-point coefficients. In the second part of this assignment, you will use a C-callable assembly language function to decrease the number of cycles needed to compute the output of your IIR filter.

Specifically, this assignment requires you to write a C function `iir_df2_c` and a C-callable assembly language function, `iir_df2_asm`, both of which perform the same exact function of calculating the output of a general m^{th} order IIR filter realized as a DF-II single section using global arrays for the filter coefficients and intermediate values. To get a head start on this assignment, it is recommended that you read Chapters 3 and 7 of your textbook.

Part 1: Fixed-point DF-II single-section IIR filter C function

Modify your DF-II single-section IIR filter code from Laboratory Assignment 3 to use 16-bit fixed-point filter coefficients and 32-bit fixed-point intermediate values. You will need to re-quantize your

filter coefficients using `fdatool` and note the number of fractional bits. The filter coefficients and intermediate values should all be declared as global arrays (shorts and ints, respectively).

To facilitate accurate comparisons between your C and ASM code, put all of the DF-II single-section IIR filter processing into a function called `iir_df2_c` that performs the task of calculating the output of an m^{th} order single-section Direct Form II IIR filter and updating the intermediate variable memory buffer. This function should be written generally in the sense that it should work for any filter order up to 255 (it may not run in real-time up to this order, but it should function correctly otherwise). Call this function from your ISR after getting a new sample from the ADC. The `iir_df2_c` function should have the following prototype

```
int iir_df2_c(char m, short x);
```

where `m` is an unsigned 8-bit integer containing the filter order (0 to 255) and `x` is a signed 16-bit integer containing the current input sample from the ADC. The output of the `iir_df2_c` function is a signed 32-bit integer containing filter output. You will probably need to scale this result in your ISR prior to sending it to the DAC. Do not pass in pointers to the filter coefficients or the intermediate values since these arrays are global. Your function can just read these arrays directly and modify the intermediate value array directly.

Confirm that your filter is working correctly and that it satisfies the frequency response requirements stated in Laboratory Assignment 2. Profile your `iir_df2_c` function with and without various levels of optimization and note the inclusive average and inclusive maximum values in your report. These values establish a baseline by which you will compare your hand-coded assembly language DF-II single-section IIR filter function.

Part 2: Fixed-point DF-II single-section IIR filter ASM function

Create a new project with the same main code and ISR code as in Part 1, but replace the IIR filter function `iir_df2_c` with a C-callable assembly language function `iir_df2_asm` that has the same prototype and performs the same task of calculating the output of an m^{th} order single-section Direct Form II IIR filter and updating the intermediate variable memory buffer. 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. Also recall that, 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 || (see Section 3.8 of your textbook). 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 the `iir_df2_asm` 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 fix the intermediate values memory buffer between calls 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.
- 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 the 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.

All bonuses will be based on profiling results on the C and ASM functions as measured by the grader using an m^{th} order DF-II single-section IIR filter with $2 < m < 255$. You will not be given m in advance, so please confirm that your function works for different values of m in this range. Your C and ASM functions have to work correctly and have to exactly match the input/output requirements 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, but you can ignore all of the floating point instructions for the purposes of this assignment. Your textbook is a good place to start but you will probably need to refer to the TMS320C6000 Programmer's Guide 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 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?

To help with optimization, it is highly recommended that you draw some flowcharts and dependency 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 single-section 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.