Migrating an Academic DSP Lab Course from the TMS320C30 to the TMS320C67 EVM

By Keith Hoover,
Professor of Electrical and Computer Engineering, Rose-Hulman Institute of Technology, Terre Haute, Indiana
(Keith.E.Hoover@Rose-Hulman.Edu)

Introduction

This paper describes the experiences of Professor Mark Yoder and myself in converting our TMS320C30 (C30) DSP projects course to the TMS320C67 (C67). The weekly laboratory projects assigned in our new DSP projects course will be described. . The pros and cons associated with C30 vs. C67 DSP-based instruction in an academic environment will be summarized.

The ECE department of Rose-Hulman Institute of Technology has regularly offered a graduate/senior-level one-quarter (10-week) DSP laboratory project class based on the Texas Instruments (TI) C30 DSP since 1992. This C30-based course was described in the *6th Annual TI TMS320 Educator's Conference Proceedings* ("TMS320C30 DSP Laboratory Course taken Concurrently with a DSP Theory Course", Keith Hoover, August 2, 1996). Last year, prompted by several DSP board failures, we decided to upgrade our laboratory project course from the C30 to the C67. We were still satisfied with our original C30 boards, but we were disappointed to find that the particular seven-year old DSP boards we had adopted (from a smaller DSP board manufacturer) were no longer made or supported. As we moved to the C67 DSP, we decided to "learn from past mistakes" and adopt the more "mainstream" TI C67 EVM DSP board that is bundled with the Code Composer Studio integrated development environment (IDE) software. We reasoned that these TI C67 boards might be supported for a longer time than C67 boards from a smaller company. In addition, we expected that most educational materials published for the C67 family would focus specifically on the TI C67 EVM DSP board.

More Power, but More to Learn!

As we began upgrading our course from the 60-Mips C30 board to the 1600-Mips C67 EVM we discovered that there is considerably more material for the student (and instructor) to learn. For example, the C67 board features a parallel, pipelined architecture, flexible stereo audio interface (CODEC), and more sophisticated support software (Code Composer Studio. One indication of the steeper learning curve is the substantial increase in the number of user manuals provided for the C67! Our original 10-week C30 course was long enough to acquaint students (who had already taken at least one general microprocessor or computer architecture course) with the C30 processor and its software development tools. This course was able to do a reasonable job of covering the salient features of C30 architecture, assembly-language programming, analog I/O, C programming, and PC/DSP communication. Only one TI manual had to be purchased by the students for the course (*TMS320C3x User's Guide*).

In contrast, the C67 EVM, along with its associated Code Composer Studio IDE, and its associated debugging and real-time support software, are supplemented by approximately 20 assorted user's manuals (so many that it was only practical to distribute them to the student in "soft form" as Adobe portable document ".pdf" files.) We felt that we could no longer adequately cover all aspects of C67 DSP programming applications in a single 10-week class as we did with the C30.

From One C30 Class to Three C67 Classes

Rather than trying to "teach it all" in one quarter, our C67 instruction was divided into three roughly independent 10-week courses. The C67's parallel, pipelined architecture, assembly-language, and C programming are covered in our Computer Architecture II course (EC332), taught by Professor Mark Yoder. This course is patterned after TI's *TMS320C62xx DSP Design Workshop* (Texas Instruments, DSP6-NOTES-3.2a, April 1999). Special emphasis is placed on assembly optimization, software pipelining, and techniques for writing optimized C code. In this course, students use the simulation capabilities of Code Composer Studio to write, debug, and analyze various assembly and C programs.

A second course, entitled "Real Time Systems Programming" (EC597), taught by Professor Mark Yoder, focuses on the use of Code Composer Studio's DSP/BIOS data analysis, data profiling, event scheduling, and real-time data exchange (RTDX) capabilities. This course emphasizes real-time programming issues. It does not dwell on the theory and implementation

behind the signal processing algorithms themselves. This course is patterned after Texas Instrument's *Real-time Software Design Workshop Using Code Composer Studio* (Texas Instruments, RTSD-NOTES 1.1, August 1999). In this course, the students work in teams to perform several assigned, real-time programming exercises that teach them about various aspects of Code Composer and DSP/BIOS. Then they choose a term project. Projects undertaken this year reflect the students' individual interests and area of specialization. They included a real-time, one-dimensional broom balancer, an MPEG-II video decoder, a phased audio microphone array, a video camera-based label recognizer, a voiceprint display, and a "voice over IP" internet telephone decoder.

The third course, taught by myself, will be the focus of this paper. Entitled "DSP Projects" (EC581), it is the one that most directly corresponds to our original C30 DSP projects course. This course focuses on the C67 real-time implementation of common DSP algorithms using the C programming language. The only prerequisite for this course is our senior-level DSP theory course (which uses the popular *Discrete Time Signal Processing* by Oppenheim, Schafer, and Buck). The primary goal of this course is to reinforce the DSP theory course by providing students with an opportunity to implement and test algorithms in software. Only about half of the students taking this course will have had either of the other two C67 courses. For this reason, the emphasis in this class is on writing simple, working C code, rather than on optimizing the code after it is working. All I/O is to be done at the lowest level possible (keeping the student "close to the hardware"), rather than using the more sophisticated and abstract DSP/BIOS real-time I/O functions (which are covered in another course).

The DSP Projects course consists of three components: (1) Nine single-week projects, (2) Classroom demonstrations of more advanced topics, (3) A term project. All project work is to be done in two-person teams. Most of the projects involve some aspect of digital audio processing, which has been found to be an especially motivating topic for many of our students. Project reports are required to be in a prescribed, semi-formal, "memorandum-style" format, with all data, program listings, and detailed data analyses included in appendices. The course meets for two hours of "pre-lab lectures" and one 3-hour lab session per week. In addition, the lab is open on a walk-in basis throughout the week. Each student is expected to spend approximately six hours per week in the laboratory, outside of the scheduled class/lab times.

C67 DSP Projects Course Content

Each of the required C67-based experiments currently performed in the DSP Projects course (seven on the C67 EVM in real-time using C, and two using Hyperception's RIDE 4.2 "block diagram" DSP tool) will be briefly described.

Please note that the descriptions below are only very terse summaries of the lab handouts given to the students. Interested instructors are welcome to download the weekly C67 project assignments, as well as associated sample programs referred to in these project assignment handouts, from the following URL:

http://rose-hulman.edu/~hoover/.

(1) **Code Composer Familiarization, Audio Sampling, Reverberation, Comb Filter, Flanger**

The student is guided through the use of Code Composer Studio to perform editing, compilation, linking, downloading, debugging (single-stepping, setting breakpoints, setting up watch variables, etc.), and execution of a simple C-language program which contains *printf( )* and *scanf( )* functions that perform simple data processing operations. Next, a basic, interrupt-driven, sampling program is introduced. This basic program and its companion interrupt routine will serve as a "template" for many of the following DSP lab projects. The student is then asked to modify this basic sampling template to turn it into an audio reverberation program, and then later into a comb filter. Finally, for extra credit, the comb filter delay may be made continuously variable, to implement an audio flanger.

(2) Floating Point and Fixed Point FIR filter implementation

A MATLAB *M-file* (which calls the "*FIR1*" MATLAB FIR digital filter design function) is used to design and plot the frequency response of 15^{th} order bandstop, bandpass, and highpass filters. Then a real-time, C digital filtering program is written that uses

floating point filter coefficients to implement the various FIR filters on the C67 board. The resulting real-time filter is tested using a function generator and an oscilloscope, with the observed results plotted over the frequency response curve predicted by MATLAB. Next, the program is rewritten so that it uses only integer math (prescaling the floating point filter coefficients by multiplying them by a large power of 2 and then truncating them to integer form).

(3) IIR Filter Implementation and Digital Wah-Wah Effect
Use Momentum Software's QEDesign Digital Filter Design program to design an IIR bandpass filter to meet given specifications. Graphically interpret the resulting pole-zero plot (using ruler measurements) to verify the predicted magnitude response. Next, write a C program that implements the IIR filter in "Direct Form II", cascaded 2^{nd}-order biquadratic sections. Experimentally record observed, real-time filter performance using an oscilloscope and a function generator, and plot the experimental measurements over the predicted magnitude response. Finally, implement a series of 10 second-order bandpass digital filters whose passbands span the audio spectrum. Write a program that periodically (every 50 ms) switches filter coefficients, resulting in a filter of continuously varying passbands. Listen to the effect when a low frequency (100-Hz) square wave (spectrally rich) is played through the system. Note the classic "Wah-wah" effect.

(4) Audio AGC With Silence Threshold (Digital Audio Amplitude Compression)
Write a real-time C-language program for the C67 that stores a 30 ms audio "frame" in a circular buffer. This program must also calculate the average of the absolute values of the audio samples in the frame. Finally, the program must scale each value in the frame by this average magnitude to adjust the average magnitude of the frame to a constant, pre-specified value. Then the resulting adjusted audio frame should be sent out to the loudspeaker, while the next 30 ms frame is being recorded. Silent frames are detected by comparing the average frame magnitude against an experimentally determined "silence threshold" value. If the frame's average magnitude is below this threshold, the frame is zeroed rather than scaled. The program must be written efficiently enough that no (or very few) audio samples are missed between frames. The resulting speech heard in the loudspeaker should sound natural and continuous and be of constant average amplitude, even when the speaker moves several feet away from the microphone.

(5) **Audio VU Meter, with Separate Target and Host C Programs Communicating Through PCI Interface**.
Modify the AGC program from preceding assignment to pass the average frame magnitude (a new one every 30 ms) to a simple companion Microsoft Visual C++ "terminal application" program. Use the C67 EVM board's PCI interface hardware, where the average frame magnitude will be displayed with ASCII graphics. Use the "*dma_*" function call in the C program that runs on the C67 EVM board, and use the "*evm6x_read*" Windows API function in the companion C++ program that runs on the PC. First study the example C67 C program and the companion example C++ PC program. These illustrate the proper method for DMA transfer through the C67 EVM's PCI interface.

(6) **Radix-4 FFT Spectrum Analyzer**
Study the 64-point radix-4 FFT algorithm explained in the handout. Draw the simpler 16-point radix-4 FFT butterfly pattern and indicate the value of each node in the butterfly for the specific test pattern given. Compare the final results with those from MATLAB. (The results should agree.) Next run the C67 radix-4 FFT program that has already been coded using the same test data. Note that it is already set up to calculate the same 16-point FFT. Verify the proper results. Now modify this FFT program for 64 points and test it with two test cases (verified against MATLAB). Now integrate this 64-point FFT routine into your interrupt-driven, sampling program template. In the main program, a global index variable (incremented in the interrupt routine) is used to keep track of when 64 points have been stored. Then the FFT is called, and the frequency corresponding to the position of the peak magnitude value is

printed. Proper operation can be verified using a sine-wave function generator whose frequency is slowly varied between 0 Hz and half of the sampling rate.

(7) Real-Time, Narrow Band Noise Reduction via Adaptive Filtering
Implement an LMS adaptive filter, and then pass an audio signal (consisting of speech corrupted by a strong, additive, periodic interference signal) through a delay line. The delayed speech + noise signal is delivered to the reference input, while the non-delayed version of the speech + noise is delivered to the signal input of the LMS adaptive filter. The error signal becomes the de-noised output. First implement this filter using MATLAB and imported WAV files. Then demonstrate significant noise reduction by playing back the processed WAV file. Experiment with the adaption coefficient value and the necessary "decorrelation delay time". Finally, convert your MATLAB program to a real time C67 program.

(8) **Use of Hyperception Ride 4.2 to Perform Digital Filtering and Audio Scrambling/Descrambling**
Study and run the various block diagram DSP examples. Also study the instructions for using Hyperception's companion "Hypersignal Digital Filter Design Program". Now construct your own block diagram system that uses the concept of mixing and filtering to invert an audio spectrum. Cascade two of these spectral inversion systems to realize a simple audio scrambler and unscrambler. Demonstrate by first playing the scrambled audio from the first spectral inverter and then playing the unscrambled audio from the second spectral inverter.

(9) DSP Scavenger Hunt
Use Hyperception RIDE 4.2 digital filter blocks and/or your adaptive filter program to remove noise from a series of noise-polluted digital audio (WAV file) clips which indicate the location of various $5.00 bills hidden around campus. Since the nature of the noise varies from one clip to another, you will have to apply different filters to each clip. You may want to observe the spectrum of various frames within the clip before deciding how it should be filtered.

(10) Term Project (to be chosen by students who are taking the class for 4 credit hours) Example projects: Real-time audio spectrum analyzer, ultrasonic chirp sonar, touch-tone DTMF telephone monitor, guitar tuner, underwater ultrasonic receiver/transmitter, voice-operated security lock.

Pros and Cons of Upgrading an Academic DSP Lab Course from C30 to C67

Our experiences in migrating our DSP projects course from the C30 DSP board to the TI C67 EVM DSP board have been largely rewarding, but not completely without frustration. Making the transition has been much more challenging than we first imagined. Many of the real-time DSP operations that took too long on the old boards are not as "time-critical" on the new boards. Because we kept our DSP lab course at the C-language level, most of the lab projects we were doing with the C30 carried over without significant changes to the C67. However, the details of how analog I/O is performed have significantly changed and are significantly more involved than they were on the C30 boards. Fortunately, many of these I/O details are "buried" in the initial, interrupt-driven, sampling template program that was given to the students in the first lab assignment. A complete understanding of the details of this template requires that the students study the 76-page CODEC user manual and also the various C67 EVM API function calls that are described in the EVM user manual.

We are still struggling with frequent instabilities in our Windows installation of Code Composer. In our particular installation, Code Composer frequently "freezes", and the only solution that we have found is to exit Code Composer, run the DOS EVM67 board reset program, and then re-enter Code Composer. Having worked with the C67 EVM for 6 months, I still feel that my students and I still have a lot more to learn about the use of the high-level DSP/BIOS function of the C67 EVM.