Lab 2 – Intro to the SHARC EZ-KIT Lite Evaluation Board

"64K ought to be enough for anybody" – W. Gates, III

Overview

So you've written your program and simulated it, and thoroughly debugged it. Before you go burn 10,000 PROMs, it might be a good idea to try your program on some real hardware. Simulation is great, but… what if there's a bug in the simulator? Impossible? Well, no matter how good your simulator is, if you are in the "real world," your boss (who long ago stopped being nice, and started getting real) wil *not* let you blame 10,000 recalled cell phones on a "bug in the simulator."

That's where the Evaluation Board (EVB) comes in.

The EVB lets you run your actual program on the actual processor. You get to test how our program handles interrupts, how it interfaces with ports and peripherals.

The SHARC EZ-KIT Lite EVB is pretty nice, as testbed systems go. It has a built in analog interface, including an audio front end. It has a 3 push buttons and 4 "blinky lights," one other light is the power LED. And believe me, as someone who has put in a couple of years debugging programs for this board and others like it, you gotta have blinky lights.

SHARC EZ-KIT Lite Features

The SHARC EZ-KIT Lite has, of course, an ADSP-2106x DSP, operating at 40MHz. The processor itself has 2 blocks of internal memory of 512Kbits each. The memory could be divided in to blocks and configured to be either 48 bits or 32 bits wide. Although the ADSP-2106x allows the use of external memory, this board does not have the external memory to worry about. Here is a list of the SHARC EZ-KIT Lite's main selling points:

- Analog interface: 16-bit stereo audio interface
- 3 user input push-button switches and 4 output LEDs
- Expansion interface (Eurocard DIN-96)
- JTAG connection for In-Circuit Emulator (ICE)
- RS-232 host connection

Let's examine each in closer detail.

Analog Interface

For audio applications, this board is the thing. All the analog interfacing has been done for you through the AD1847 Stereo Codec. The audio interface has 2 1/8" jacks, one for input (A/D) and one for output (D/A). The sample rate for audio signals can be set by software to any of a fixed set of 16 rates from 8kHz to 48 kHz. So you can't have *any* rate you want; you have to pick one that Analog Devices offers you.

The ADSP-2106x DSP connects to the codec through a serial port on the chip called SPORT0. This means some initial setup has to be done in order to make the codec work. Luckily, this is already done for you. The file `init1847.asm` provides the basic setup for AD1847 codec to use with SHARC EZ-KIT Lite.

Let's look at some code from the file:

```
.GLOBAL setup_1847;
.GLOBAL spt0_asserted;
.GLOBAL rx_buf, tx_buf;
```

A `.GLOBAL` directive in front of a label tells that the label is expected to be used by other assembly program and the definition of the label could be found somewhere in this file. So, the other files, namely your own .asm files, do not have to define these symbols themselves. Instead, `.EXTERN` directive is used to tell the linker to look for these symbols somewhere else. So, you have to put

```
.EXTERN setup_1847;
.EXTERN spt0_asserted;
.EXTERN rx_buf, tx_buf;
```

in your program if you expect to use the initialization of the codec from this `init1847.asm`.

Well, what do these symbols mean, then? `setup_1847` is a subroutine you have to call in order to setup the codec. So you put the code like this

```
call  setup_1847;
```

somewhere in the very beginning of your main program. We will talk about subroutines more in Lab 3.

`spt0_asserted` is the subroutine that is called each time the processor transmits data to the codec. That is, the SPORT0 transmitting interrupt is asserted. A useful set of code is already put in the asm file, so there is nothing you have to do about this symbol. Now, the very important variables, `rx_buf` and `tx_buf`. `rx_buf` stores the information received by the codec. That is, we are talking about the input signal. And whatever our output is, we put in `tx_buf`. The two variables are 3 words long. The following table describes each word for each variable.

	rx_buf	**tx_buf**
word 0	Status data	Configuration data
word 1	Left channel A/D data	Left channel D/A data
word 2	Right channel A/D data	Right channel D/A data

Suppose you wanted to read some incoming audio data from the left channel. Here's all you have to do:

```
r0=dm(rx_buf + 1);
```

And do the similar thing to the output.

```
dm(tx_buf + 1) = r0;
```

A few more things to point out about audio interface. The audio data is only 16-bits. Incoming data is put in the 16 Most-Significant Bits (MSb) of a data word. Your program's outgoing data should also be put in the 16 MSb of the word. The format of the data is fixed-point two's complement fractional numbers.

When new audio data is ready, the audio subsystem asserts the SPORT0 receiving interrupt. If you enable this interrupt, you can set up an interrupt service routine which will read the data and store it in memory (or something). And actually, this interrupt is already enabled for you in `setup_1847` subroutine, so you do not have to enable it again. All you have to do is providing an interrupt service routine for this interrupt.

So, how do we include this init1847.asm in our program? First of all, assemble the file as usual. Then link the resulting object file with our object file obtained from assembling your own asm file. For example, suppose your asm file is called lab2.asm. The whole process is

`asm21k lab2` if no error, this should give lab2.obj which will be used when linking.
`asm21k init1847` This should give init1847.obj

Now link them together

```
ld21k –a ezlab2 lab2 init1847
```

If no error, this should give the executable. That's it!

Note that in order to yield `lab2.exe,` the file `lab2` should come before `init1847` when linking. Otherwise, you will get `init1847.exe` instead.

Buttons and Blinky Lights

The three buttons on the SHARC EZ-KIT Lite are connected to the following 2106x lines:

- RESET
- IRQ1
- FLAG1

No doubt you are familiar with reset buttons. The IRQ1 button can cause the 2106x to execute immediately a chosen subroutine. The FLAG1 button, when held, will set a bit in the 2106x ASTAT register (see the User's Manual). The two buttons work in similar ways: they both can send a signal to the 2106x. However, unless the program running on

the processor in looking for the FLAG1 signal, it won't notice it. The interrupt will always get the processor's attention, although your program must *enable* the interrupt, and you have to setup your *interrupt service routine*. More on that later.

The blinky lights are connected to flags 0, 1, 2, and 3. These are controlled by bits in the ASTAT register. Again, consult your user's manual for details on the ASTAT register.

These lines of code with configure flags 0, 2, and 3 as outputs, flag 1 as input and will turn off the LEDs 2 and 3:

```
bit set mode2 FLG0O|FLG2O|FLG3O;     /* configure flags as outputs */
bit clr mode2 FLG1O;                              /* flag1 as input */
bit clr astat FLG2|FLG3;       /* clear the flags => turn off LEDs */
```

It is left as an exercise for the students to figure out how to turn the LEDs on.

I suggest you become familiar with the blinky lights. I can't really stress how useful they are for debugging or diagnostic purposes. For example, suppose you are debugging a program. You want to make sure that a certain bit of code is being executed. Simply turn on an LED right after that bit of code. Another use is to toggle the LED's state inside a routine that is executed periodically. If the routine is running, the LED blinks.

Expansion Interface
I won't go into this too much, except to say that it is useful for connecting peripherals to the SHARC EZ-KIT Lite board.

JTAG Connection
You will discover that it is difficult to debug your programs when all you can do is blink some lights (not that that's not useful). But sometimes you want something that looks like the simulator. In that case , you use an *In-Circuit Emulator* or ICE.

The ICE lets you do all the things the simulator does, like single-stepping, setting breakpoints, viewing memory and registers, etc, but with the real hardware, with interrupts and analog interfaces and everything.

The disadvantage is that the ICE is expensive. And delicate. It's not something you let undergraduates play with much. But if you're good…

Using the SHARC EZ-KIT Lite

In the last lab, you learned how to use the assembler and linker to create an executable program from an assembly source file. You also learned how to use the simulator to test and (hopefully) to debug your programs. Now you are ready to use real hardware.

The first step is, of course, to create an executable. The executable file contains a *memory image* of what your program; i.e., it is a binary file whose contents would go straight into the memory of the DSP. As you know, the simulator uses the executable.

However, there is really no way to put the contents of this file directly into the memory of the DSP.

Here is where the *SHARC EZ-KIT Lite Host* program comes in. You can find this program under the SHARC EZ-KIT Lite program group under the Start menu. The program allows you to download your executable to the DSP's memory and execute it.

The SHARC EZ-KIT Lite EVB, just like other microprocessor-based circuits, starts with the power-on routine that stores in the PROM. The routine initializes the UART chip (which controls the serial port) and waits for the Host program to communicate with it. While waiting, it plays the Peter Gunn's theme through the output audio port. If you have speakers, you can here the song. The Host program allows you to download your executable to the memory and replace whatever that already in the memory at startup, then soft-reset the board, so your program can run.

When you first open the Host program, the program will try to locate and communicate with the EVB board. If everything went okay, it will show "Communications with board are OK". Otherwise, it will complain something and we have to fix the error. The error may be because the wrong Com port on the PC is being used. You can change this in the program under the menu Settings -> Select Com Port. To download your program to the memory, just go to File -> Open, and then select the program to run. The Host program will then download that program and show "Downloading to board" dialog. When finished, the Host program soft-resets the board and your program will run. If your program doesn't run, try to adjust some settings on external hardware, or maybe your code doesn't work.

Under the Settings menu, you will find a bunch of reset commands, which have different uses. After you run program a while and you might want to restart your program, you have to *soft-reset* the board. Your program will start over. If you *hard-reset*, your program will be replaced by the default kernel in PROM and you have to download it to the board again. The *manual reset* tells the Host program that you want to reset the board by yourself. When you select this menu, press the reset button on the board first, then click OK on the dialog that appears.

The Host program also allows us to download any data directly to the memory, or upload contents of the memory and dump to a file, or view contents of the memory. Just enter the range of the memory in hexadecimal format when asked. Choose floating-point if it's floating point data. And click the button. Everything is very straightforward.

Lab Assignment

In this lab we will be using a function generator to provide an analog input signal. We will display our analog outputs using an oscilloscope. The programming techniques we will use are:

- Interrupt handling
- Circular buffers

As mention earlier, the SPORT0 will generate an interrupt for us when there is data ready to be retrieved. So, we will use this interrupt as a hint. You must do the following things in your program:

1. Provide an *interrupt vector* and an *interrupt service routine (ISR)*. The vector is placed in the `spr0_svc` segment. So put these lines of code in your program:

   ```
   .SEGMENT/PM spr0_svc;
          jump spr0_asserted;
   .ENDSEG;
   ```

   Then in the `pm_sram` segment, create a subroutine called `spr0_asserted`. I suppose I should point out here that you don't have to call it `spr0_asserted`. You can call it whatever you want, *as long as it matches* the destination of the jump in the `spr0_svc segment`. Don't forget to end your service routine with the `rti` instruction:

   ```
   spr0_asserted:
     /* do whatever */
          :
     /* finished */
     rti;
   ```

2. Enable interrupts. There are two steps to this. First of all, there is a *global interrupt enable* bit in the `MODE1` register. This bit has to be set before any interrupts can occur. Then each interrupt source has its own mask bit in the `IMASK` register that has to be set before that source can cause interrupts. Think of interrupts as lights in a house. Then think of the global enable bit as the main breaker, and the interrupt masks as the light switches. The main breaker has to be closed before any of the lights can come on. After that, each light can be switched on or off independently of the others.

   In your case, you want to enable the SPORT0 receiving interrupt. You put some code like this:

   ```
     bit set imask SPR0I;     /* enable sport0 rcv irqs */
   ```

   Fortunately, this is already done for you in the `init1847.asm`. (Still remember this file?) You do NOT have to enable this interrupt again.

   Anyway, you still need to put these lines of code somewhere in your program to enable the global interrupt (preferably near the beginning):

   ```
     bit set mode1 IRPTEN;   /* enable global interrupts */
   ```

Programs

Your first program is simple: read data from ADC, then copy that data out to DAC. We do this at a sample rate of 48kHz. So, you just enable your interrupt and set up the interrupt service routine. In your `spr0` service routine, read from ADC into a register, then copy that data to DAC. Use a function generator and an O-scope to verify your program. Measure the time delay between the input and output signals.

Your second program will expand on the first. Keep track of the last input. When a new input arrives, subtract the last input from it, and output the difference. Hint: don't forget the format of the data read from the ADC is unsigned. Try this with square, triangle, and sine wave inputs. Sketch the results.

For the third program, keep track of the last N inputs. Output the average of the input and the last N inputs. Allow N to be specified at assembly time. Hint: a circular buffer will be useful here. Sketch the results for inputs of square, triangle, and sine waves.

For the Report

1. What kind of filter did you implement in program 2? In program 3?
2. What happens if your input signal has a frequency higher than 24kHz? Using program 1, input a sine wave at 30kHz, 40khz, 48kHz, and 60kHz. Measure the frequency of the output signal for each of these inputs.