Testing performance of SPIFI interface and selected DSP library functions

George Romaniuk , February, 2012

My last embedded project required me to use an FPGA to handle interfaces to ADCs and DACs as well as some signal processing. The selection of an interface between the CPU and FPGA came down to these three options:

1. Map the FPGA into a non-multiplexed address/data bus;
2. Map the FPGA into a multiplexed address/data bus;
3. Or use SPI to communicate with the FPGA.

The SPI interface has the lowest pin count, but handling the data transfers between the SPI port and the application requires a lot of code and the latency may be a problem for some control applications. I was hoping to find a quad SPI interface and an intelligent SPI controller with FIFO and DMA but I couldn't find such thing until I got NXP slides showing the LPC4350.

My interest in LPC4350 was drawn by its 200 MHz ARM M4 core with DSP instructions, USB, Ethernet, and SPIFI.

The SPIFI was particularly interesting since the LPC4350 slides mentioned a high transfer rate of 50 MB/s and the ability to execute code from the memory-mapped SPIFI device. This triggered my attention and I signed up for the project.

The Hitex board arrived in a nice box. All I had to do was to attach four stand-offs (not supplied) to keep the board off the desk surface and connect my 5V power supply.

The letter found in the box contained all that was needed to move on with installation of Keil and getting access to the LPC4300 project community. Downloading and installing the Keil MDK went smoothly, and the license keys arrived quickly.

I powered the board and saw both demos working. I started my blog and downloaded data sheets for the parts and the updated LPC4350A_PDL.zip file with examples, posted by John Donovan.

Download the sample code here

The examples were installed in /Boards/Hitex directory. I opened multi-project and every example in this collection compiled except for the SPIFI one. I went back to data sheet and Preliminary users manual to learn more about the part and check how much functionality is covered by the examples, with focus on SPIFI.

In UM10503, section 22.3 I found the following statement:

- External flash is directly memory mapped for fast access.

This is truly amazing but how can I map my FPGA instead of flash? Further reading of Chapter 22 did not offer any additional insight. I searched the web and older example files and found two pieces of information. One is the LPC4350/Drivers/Lib/SPIFI_ROM_support.doc document, the other is a set of include files:

LPC4350/Tools/Flash/SPIFIdriver/LPC18xx.h with 8 SPIFI related registers at 0x40003000

LPC4350/Core/Device/NXP/LPC43xx/Include/LPC43xx.h, which has nothing defined at 0x40003000

From the above I concluded that for the time being the SPIFI device is pretty much dedicated to supporting external FLASH through the ROM-based routines.

I still don't understand why there is a cloak of mystery around the SPIFI interface; traditionally it was related to problems with the hardware. A quick look at the SPIFI header files and the code convinced me that understanding the operation of SPIFI by reverse engineering of the code would be a waste of time. I decided to verify the claims presented in the user manual.

The non-compiling SPIFI example was fixed by adding debug_frmwrk.c and lpc43xx_uart.c to the driver group. I also added code from the Fastboot example to optionally run the core at 204 MHz. To time the various SPIFI functions I used Repetitive Interrupt Timer (RIT) because the RTC has 1 sec resolution; the ATIMER is only 16 bit long and is running of 1024 Hz clock. The RIT is 32 bit long and runs off a higher speed selectable clock.

There is a driver for the RIT, though it is missing the user friendly GetRITCount function and you need to uncomment RIT in the lib_config.h file in order to use RIT.

With all these changes I was able to time the SPIFI functions and present my findings on serial console:

```
    Hello NXP Semiconductors

    SPIFI demo
      - MCU: LPC4300
      - Core: ARM CORTEX-M4
      - Communicate via: UART1 - 115200 bps
*********************************************************************************
    CPU Clock frequency is 0204000000 Hz

    Timer systematic error is 0000000014 RIT clock ticks or 0000000068 ns
    Initializing SPIFI driver...
    Initializing SPIFI driver takes 0000010911 ns
      FLASH manufacturer ID = 0x00EF
      FLASH device type =  0x0040
      FLASH device ID =  0x0014
    OK
    Erasing QSPI device...
    Erasing FLASH takes 1183522200 ns
    OK
    Programming + verifying QSPI device...
    Writing 256 byte page into FLASH takes 0000661039 ns

    Readback of 256 byte page takes 0000005700 ns
    OK!
```

I was impressed with data read, 256 bytes in 5700 ns equals to transfer rate of 44.91 MB/s. I verified the signals with an oscilloscope, and everything looked reasonable. The memory window showed all the data. I would like to mention that the nice looking red jumpers on the development board do not allow for easy access to signals. Jumpers with an open top would be helpful.

My attempt to write data to the memory-mapped FLASH failed with an access violation. I had to use SPIFI_ROM driver function spifi_program to write data into the FLASH.
I believe that NXP will update the user manual and expose SPIFI registers for those who would like to use SPIFI to interface with external devices other than serial FLASH. The SPIFI is very fast indeed.

I was tempted to time a few routines from the DSP library. I selected the FFT-based convolution example as a starting point. The example did not compile because of missing ARMCM4.h file. After some searching I found the missing file in the \Keil\ARM\Device\ARM\ARMCM4\Include directory. I compiled the example and wanted to add debug and timer functions. This turned out to be a problem due to conflicts within header files.
Eventually I decided to compile the convolution example into a library and use it within the SPIFI example.

Everything compiled and linked nicely but upon startup I got a hardware exception. Looking at the stack I found the exception code to be 0xFFFFFFF9 and realized that the FPU was not

initialized. First I noticed that the file lpc43xx.h has __FPU_PRESENT was set to 0 so I changed it to 1 but the problem was still there. The next step was to understand how to enable the FPU and look for this code in the startup file. Inspection of system_LPC43xx.c file uncovered the missing initialization.

After the FPU was enabled, the code executed and the SNR check failed. The length parameter in function arm_snr_f32 was incorrect given testRefOutput_f32[126]. Correct length is 1 shorter than used in the original example.

Eventually the program produced the following message on my terminal:

FFT based convolution takes 0000027839 RIT clock ticks or 0000136465 ns

I broke the total time down into individual function calls for 200 MHz M4 clock:

Function	CPU Clocks	Time in ns
status = arm_cfft_radix4_init_f32(cfft_instance_ptr, 64, 0, 1);	38	190
arm_cfft_radix4_f32(cfft_instance_ptr, Ak);	4723	23,615
arm_cfft_radix4_f32(cfft_instance_ptr, Bk);	4723	23,615
arm_cmplx_mult_cmplx_f32(Ak, Bk, AxB, MAX_BLOCKSIZE/2);	134	670
status = arm_cfft_radix4_init_f32(cfft_instance_ptr, 64, 1, 1);	218	1090
arm_cfft_radix4_f32(cfft_instance_ptr, AxB);	4708	23,540
Floating point SNR calculation	10,415	52,075

I also checked the floating point FIR arm_fir_example_f32.c to verify that such a filter can be run in real time on the M4 core. The input sampling rate is 48 KHz, the number of samples per block is 32, and this corresponds to 667 µs of real time. The FIR takes 4582 clocks (22,910 ns) to process one block so the M4 would be 3.4% utilized filtering one audio channel.

All these results make me think about using the LPC4350A in my next project, but it would be nice if all the DSP functions were given execution time in CPU clocks and the CMSIS examples for the LPC4350A were cleaned up a bit to consistently start using ARMCM4.h instead of occasionally requiring ARMCM3.h.

I want to thank NXP for giving me early access to the LPC4350A and software tools.

Date of release: February 2012
Published in the USA