# Testing LPC4350 floating point performance

Jerry Durand, February, 2012

When I discovered that the M4 core in the LPC43xx series could be used as a hardware Floating Point Unit (FPU) while the actual code ran in the M0 core, I immediately thought of an application—a standalone 4-axis CNC control box for use with table-top milling machines. Currently these machines are almost exclusively controlled by software running on very old PC hardware that has a parallel port and either MS-DOS or Windows XP for an operating system. Control of these milling machines entails tasks such as receiving g-code commands in ASCII text and a large number of high-precision floating point operations, which calls for an FPU. This all has to occur rapidly and continuously as each of the four motors has to be updated thousands of times per second.

It seemed the M0 core would excel at the basic input/output (I/O) and timing functions while using the M4 as an FPU would speed the floating point operations.

I originally hoped to port the entire *RS274NGC* open source Enhanced Machine Controller (EMC) software to the LPC4350 and run it with and without the FPU enabled, comparing the time to process a sample g-code file. Problems getting the development software and examples set up caused me to run short of time even though NXP was fairly fast to fix problems as they were found. Instead I modified one of the example programs (GPIO_LedBlinky) to enable testing the performance and this turned out to have some interesting results I might have otherwise missed.

I ran the test code in eight different configurations:

1. FPU disabled, 32-bit float multiplication
2. FPU enabled, 32-bit float multiplication
3. FPUdisabled, 64-bit double multiplication
4. FPU enabled, 64-bit double multiplication
5. FPU disabled, 32-bit float division
6. FPU enabled, 32-bit float division
7. FPUdisabled, 64-bit double division
8. FPU enabled, 64-bit double division

Download the sample code here

All tests were run from internal SRAM so external memory access time would not distort the results. The optimization level was left at the default of zero (0) as I had problems with the compiler eliding parts of my code if I tried higher levels.

Test results (loops per second averaged over 10 seconds):

| | | |
|---|---|---|
| 1. | 990,550 | float, * |
| 2. | 2,768,000 | FPU, float, * |
| 3. | 284,455 | double, * |
| 4. | 276,796 | FPU, double, * |
| 5. | 282,316 | float, / |
| 6. | 1,894,000 | FPU, float, / |
| 7. | 85,762 | double, / |
| 8. | 85,053 | FPU, double, / |

As expected 32-bit floating point operations run a lot faster with the FPU, 2.79 times faster for multiply and 6.7 times faster for division. I would have expected somewhat better performance but this is still a significant improvement in speed.

The 64-bit floating point operations were a different story, for both multiplication and division the operations ran SLOWER with the FPU than they did without it. This points to an error in the library functions since it should be no worse than equal. I was very surprised by this result and expect that when the problem is fixed the 64-bit operations will improve significantly.

In conclusion the low cost of these parts and the simplicity of using the FPU allows performance improvement to an existing product that uses 32-bit floating point operations. There is no long development cycle or fear of breaking something in your code as the changes to use the FPU consist of adding a small initialization routine for the FPU and enabling it in your compiler options. This also leaves open the option of an even greater improvement with no hardware changes once you've had time to port your critical code to run directly on the M4 core instead of just using it as an FPU.

In the case of 64-bit floating point operations, it seems this may not be the best choice if you only use the M4 as an FPU. Running your critical code directly on the M4 may give a significant improvement but that requires porting the code to run on both cores and isn't something I tested.

**www.nxp.com**

Date of release: February 2012
Published in the USA