

At Missing Link Electronics, a company working on reconfigurable platforms to link reliable automotive and aerospace electronics with the rapidly changing consumer and mobile-communications markets, we believe the PowerPC processor APU inside the Xilinx Virtex-5 FXT devices is a little gem. It provides embedded-systems designers with the same optimization powers traditionally available only to the “big guys” who build their own custom ASSP devices. Given what’s now available to Xilinx users, we think everyone should tap the APU to optimize their designs.

Basics of Extending Instruction Set via the APU

When designers want to optimize an embedded system, they typically do so by looking for ways to extend the instruction set of the microprocessor at the heart of their design. Traditionally, this is the best option when the complexity of the embedded system lies in the software portion of the design. You could also simply put new functionality in the design by adding dedicated hardware blocks.

However, you’ll likely find that increasing the instructions holds some great advantages that complement hardware changes, but is somewhat easier for designers to implement. For example, by extending the instructions, you can optimize your design in finer granularity. Also, extending the instruction set typically does not interfere with memory access; thus, it has the potential to optimize the system’s overall performance.

Even though individuals, companies and academic researchers have published papers on how to do it, extending an instruction set may seem like a “black art” to anyone new to this technique. But in reality, it isn’t that complex. Let’s examine how you can optimize your Virtex-5 FXT design by making some fairly simple additions to the PowerPC processor’s instruction set via the APU interface.

In general, to extend the instruction set of an embedded microprocessor, you need to understand you are modifying both the software and the hardware. First, you are adding hardware blocks to your system to perform specialized computations. These

computations execute in parallel in the FPGA fabric rather than sequentially in software. In Xilinx-speak, these hardware blocks are called Fabric Coprocessing Modules, or FCMs. You can write FCMs

out of our design without increasing the CPU clock frequency (which may cause other headaches).

The key reason to use the APU rather than connecting hardware blocks to the

The PowerPC processor APU inside the Xilinx Virtex-5 FXT devices is a little gem. It provides embedded-systems designers with the same optimization powers traditionally available only to the ‘big guys’ who build their own custom ASSP devices.

in VHDL or Verilog, and they will end up in the FPGA fabric of the Virtex-5 FXT device. You can connect one or more FCM to the PowerPC processor APU interface.

The next step is to adjust your software code to make use of those additional instructions. You have two options (assuming you are programming in C language). The first is to change the C compiler to automatically exploit cases where the use of the additional instructions would be beneficial. We’ll leave this option to the academics and certain folks working on ASSPs.

The second, and more elegant, option is not to touch the compiler but instead use so-called compiler-known functions. This means that manually, in our software code, we’ll call a C macro or a C function that makes use of these additional instructions.

Using either option, we must adjust the assembler so that it supports the new instructions. Fortunately, Xilinx includes a PowerPC compiler and assembler with the Embedded Development Kit (EDK) that already support these additional instructions.

When the PowerPC encounters these new instructions, it quickly detects that they are not part of its own original instruction set and defers the handling of them to the APU. Xilinx has configured the APU to decode these instructions, provide the appropriate FCM with the operand data and then let the FCM perform the computation.

If this is properly done, the software requires fewer instructions when running. Therefore, we can get more compute power

microprocessor via the PLB bus is the superior bandwidth and lower latency between the PowerPC processor and the APU/FCM. Another advantage lies in the fact that the APU is independent of the CPU-to-peripheral interface and therefore does not add an extra load to the PLB bus, which the system needs for fast peripheral access.

The APU provides various ways of interfacing between the PowerPC and the FCM. We can use a load-store method or the user-defined instruction (UDI) method. Chapter 12 of the Xilinx User Guide UG200 offers detailed descriptions of these techniques (http://www.xilinx.com/support/documentation/user_guides/ug200.pdf).

In our example we’ll deploy the UDI method, because it provides the most control over the system, enabling the highest performance. The example design is available for download from our Web site, at <http://www.missinglinkelectronics.com/support/>.

Example Design Description

By adding a UDI, we have extended the PowerPC processor’s instruction set to perform complex-number multiplications, a handy optimization for many multimedia decoding systems. The EDK diagram (see sidebar, Figure 1) shows the overall design, including how we connected the complex-number multiplier FCM to the PowerPC processor via the APU, and how software can make use of it.

We picked complex-number multiplication as an example because of its wide

applicability in decoding streaming-media data, and because it clearly demonstrates how to make use of the APU by adding a special-purpose instruction.

Complex-number multiplication is defined as multiplying two complex numbers, each having a real value and an imaginary value.


$(a_R + j a_I, \text{ where } j*j = -1):$

$$(a_R + j a_I) * (b_R + j b_I) = (a_R * b_R - a_I * b_I) + j (a_I * b_R + a_R * b_I)$$

For efficiency, the complex-number multiplication hardware block (`cmplxmul`), performs the multiplication in three stages. This saves hardware resources by using only two multipliers and two adders in this multicycle implementation. Figure 2 shows a block diagram (in sketch form) of the complex-number multiplication FCM.

As the VHDL code in `cmplxmul.vhd` demonstrates, we perform the complex-number multiplication in three clock cycles. In file `cmplxmul.vhd` we have implemented the FCM to perform this complex-number multiplication. File `fcmmul.vhd` provides the FCM/APU interface wrapper to connect our FCM to the APU. As we will show in our step-by-step procedure (see sidebar), you can use this wrapper as a template to connect your own FCM to the APU when using the UDI method (the load-store method requires a different interconnect).

We synthesized our design with Xilinx EDK/XPS 10.1.02 using Xilinx ISE® 10.1.02. We simulated and tested the design with ModelSim 6.3d SE.

The PowerPC processor APU that resides within the Xilinx Virtex-5 FXT devices allows embedded engineers to accelerate their systems in a very efficient way, by adding special-purpose user-defined instructions for hardware acceleration and coprocessing. Using the example design described here as a starting point will show you that mastering the APU is straightforward, and can give your designs a major performance boost without the use of special tools. 

Step-by-Step Guide to Using the APU

Here we present detailed information on how the engineers at Missing Link Electronics generated the necessary files for our example design, and how to use these files to reproduce the results on the Xilinx ML507 Evaluation Platform, which contains a Xilinx Virtex-5 XC5VFX70T device. We also show how to use this design as a starting point for your own APU-enhanced FPGA design.

Step 1: Build Your Coprocessor

Theoretically, you can build almost any coprocessor as long as it fits into your FPGA, but keep in mind that a user-defined instruction (UDI) can transport two 32-bit operands and one 32-bit result per cycle. Our coprocessor for complex-number multiplication is implemented in file `src/cmplxmul.vhd`.

Step 2: Build the FCM Wrapper

To be area-efficient, your coprocessor may need a multicycle behavior similar to ours. Therefore, you will need a state machine to implement a simple handshake protocol between the coprocessor and the Auxiliary Processing Unit (APU). In our example, we did this inside the wrapper “`fcmmul`,” which we implemented in file `src/fcmmul.vhd`.

Inside the wrapper `fcmmul`, we instantiated the complex-number multiplication hardware block `cmplxmul`, which becomes the Fabric Coprocessing Module (FCM). Thus, `fcmmul` provides the interface we need to connect it to the APU. You can find a detailed description of those interface signals in Xilinx document UG200, starting at page 188. The important detail is the timing diagram for “Non-Autonomous Instructions with Early Confirm Back-to-Back” on page 216, which shows the protocol between the APU and the FCM.

Step 3: Connect the FCM with the APU

In general, you can connect your FCM to the APU in two ways: by using the Xilinx Platform Studio (XPS) graphical user interface, or by editing the `.mhs` file. We have found that when cutting and pasting a portion of an existing design into a new one, it is easiest to edit the `.mhs` file. So for this example, we connect the FCM/wrapper and the APU in file `syn/apu/system.mhs`.

We suggest that you do the same. Just copy the section from “BEGIN `fcmmul`” to “END” from our example and paste it into your `.mhs` file.

To make it all work in XPS, you must also provide a set of files in a predefined file/directory structure. In our example, we have called the wrapper block `fcmmul`; therefore, the file/directory structure looks like this:

```
syn/apu/pcores/fcmmul/data/fcmmul_v2_1_0.mpd
syn/apu/pcores/fcmmul/data/fcmmul_v2_1_0.pao
syn/apu/pcores/fcmmul/hdl/vhdl/fcmmul.vhd
syn/apu/pcores/fcmmul/hdl/vhdl/cmplxmul.vhd
```

The `.mpd` file contains the port declarations of the FCM. The `.pao` file provides the names of the blocks and files associated with the FCM, and XPS finds the VHDL source files for the coprocessor and the wrapper in the `hdl/vhdl` directory.

You should replicate and adjust this tree as needed for your own APU-enhanced FPGA design.

Step 4: Hardware Simulation

We have provided the necessary files to test the APU example using ModelSim. As a prerequisite, and only if you have not done this yet, you must generate and compile the Xilinx

simulation library. You can do this from the XPS menu **XPS**→ **Simulation**→ **Compile Simulation Library**. Then generate all RTL simulation files for the entire design from the XPS menu **XPS**→ **Simulation**→ **Generate Simulation**.

Next, run the RTL simulation to verify your APU design—in particular, the handshake protocol between the APU, the wrapper and your coprocessor.

The simulation shows the two possibilities for the APU delivering the operands in either one or two cycles (as explained in UG200, on page 216). Look for the signals `FCMAPUDONE` and `FCMAPURESULTVALID`.

Step 5: Software Testing

For the complex-number multiplication, we have written a small standalone program, `syn/apu/aputest/aputest.c`, that demonstrates the use of the APU and our coprocessor from a software point of view.

This program configures the APU and defines the UDI. Then it runs a loop to compute the complex-number multiplication using our hardware coprocessor, compares it against the result of a software-only complex-number multiplication and provides a performance analysis.

You must configure the PowerPC APU before it can function properly, in one of two ways: You can either click in XPS and enter the initialization values for certain control registers of the APU, or you can configure the APU directly from the software program that uses the APU. We feel the latter option is more explicit and robust.

In our C source code file, you will find descriptive C macros and function calls to properly initialize the APU. Feel free to copy-and-paste into your program as needed.

Within the loop, we do complex-number multiplication first using the UDI and then using the software macro `ComplexMult`. We use our routines `Start_Time` and `Stop_Time` for performance analysis. The three calls `UDI1FCM_GPR_GPR_GPR` implement the three-cycle hardware complex-number multiplication. We define the C macro `UDI1FCM_GPR_GPR_GPR` in file `syn/apu/ppc440_0/include/xpseudo_asm_gcc.h`, which is a Xilinx EDK-generated file. We implement the C macro `UDI1FCM_GPR_GPR_GPR` via assembler mnemonic `udi1fcm`. Because Xilinx has patched the assembler, this `udi1fcm` mnemonic—though obviously not part of the original PowerPC 440 processor instruction set—is already a proper instruction the APU can handle.

In our test case, `aputest` is the XPS software project that we compiled, assembled, linked and downloaded into the Virtex-5 FXT block RAM for execution by the PowerPC processor.

Step 6: Generate the FPGA Configuration

You can generate the FPGA configuration bit file from the XPS menu **XPS**→ **Hardware**→ **Generate Bit Stream**. To save you some time, we have included a bit file for the Xilinx ML507 Development Platform. You can find it in `Syn/apu/implementation/download.bit`.

Step 7: Execute the Example Design

Download the FPGA configuration bit file, start the XPS debugger XMD (UART settings are 115200-8-N-1) and view the example design.

The run-time it reports is 4,717 cycles for the software-only design and 1,936 cycles for the UDI hardware-accelerated complex-number multiplication. Thus, the acceleration is approximately 2.4 times, with the complex-number multiplication running with no timing optimizations at 50 MHz. Of course, if we were to use pipelining and increase parallelism, the coprocessor could run much faster, increasing the overall acceleration to five to ten times.

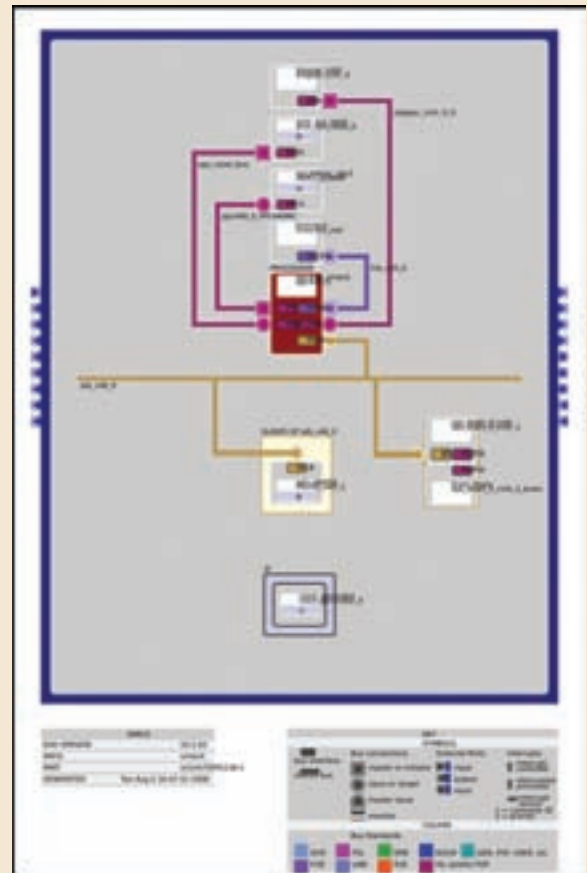


Figure 1 – EDK processor system block diagram

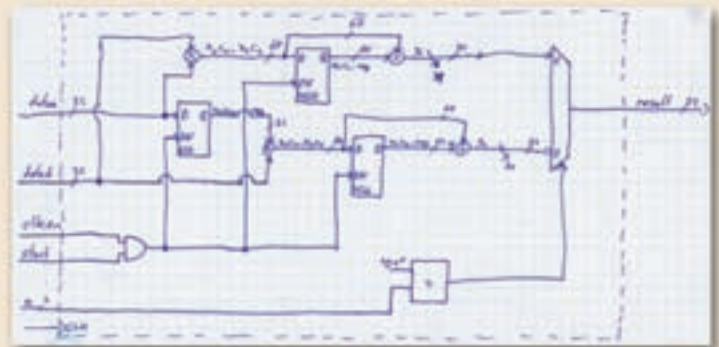


Figure 2 – Complex-number multiplication coprocessor was so easy to design, you can do it on graph paper.