

# Accelerated System Performance with APU-Enhanced Processing

The Auxiliary Processor Unit (APU) controller is a key embedded processing feature in the Virtex-4 FX family.

by Ahmad Ansari  
Senior Staff Systems Architect  
Xilinx, Inc.  
[ahmad.ansari@xilinx.com](mailto:ahmad.ansari@xilinx.com)

Peter Ryser  
Manager, Systems Engineering  
Xilinx, Inc.  
[peter.ryser@xilinx.com](mailto:peter.ryser@xilinx.com)

Dan Isaacs  
Director, APD Embedded Marketing  
Xilinx, Inc.  
[dan.isaacs@xilinx.com](mailto:dan.isaacs@xilinx.com)

The APU controller provides a flexible high-bandwidth interface between the re-configurable logic in the FPGA fabric and the pipeline of the integrated IBM™ PowerPC™ 405 CPU. Fabric co-processor modules (FCM) implemented in the FPGA fabric are connected to the embedded PowerPC processor through the APU controller interface to enable user-defined configurable hardware accelerators. These hardware accelerator functions operate as extensions to the PowerPC 405, thereby offloading the CPU from demanding computational tasks.

## APU Instructions

The APU controller allows you to extend the native PowerPC 405 instruction set with custom instructions that are executed by the soft

FCM; the primary capabilities are shown in Figure 1. This provides a more efficient integration between an application-specific function and the processor pipeline than is possible using a memory-mapped coprocessor and shared bus implementation.

The instructions supported by the APU are classified into three main categories:

- User-defined instructions (UDI)
- PowerPC floating-point instructions
- APU load/store instructions

The UDIs are programmed into the controller either dynamically through the PowerPC 405 device control register (DCR) or statically when the FPGA is configured through its bitstream. The APU controller allows you to optimize your system architecture by decoding instructions either internally or in the FCM.

The floating-point unit (FPU) is an example of an FCM. The PowerPC floating-point instruction set is decoded in the APU controller, whereas the computational functionality is implemented in the FPGA fabric. To support FPUs with different complexities, the APU controller allows you to select subgroups of the PowerPC floating-point instructions. These instructions are executed in the FCM while other subgroups of instructions are either computed through software FPU

emulation or ignored completely. This fine-tuning optimizes FPGA resources while accelerating the most critical calculations with dedicated logic.

The APU controller also decodes high-performance load and store instructions between the processor data cache or system memory and the FPGA fabric. A single instruction transfers up to 16 bytes of data – four times greater than a load or store instruction for one of the general purpose registers (GPR) in the processor itself. Thus, this capability creates a low-latency and high-bandwidth data path to and from the FCM.

## APU Controller Operation

Figure 2 identifies the key modules of the APU controller and the 405 CPU in relation to the FCM soft coprocessor module implemented in FPGA logic. To explain the operation of the APU controller and the processor interactions related to the execution units in soft logic, we can trace the step-by-step sequence of events that occur when an instruction is fetched from cache or memory.

Once the instruction reaches the decode stage, it is simultaneously presented to both the CPU and APU decode blocks. If the instruction is detected as a CPU instruction, the CPU will continue to execute the instruction as it would normally. Otherwise, within the same cycle, the CPU

- **Extends PPC 405 Instruction Set**
  - Floating Point Support (with soft auxiliary processor)
  - User-Defined Instructions
- **Offloads CPU-Intensive Operations**
  - Matrix Calculations
    - Video Processing
  - Floating-Point Mathematics
    - 3D Data Processing
- **Direct Interface to HW Accelerators**
  - High Bandwidth
  - Low Latency

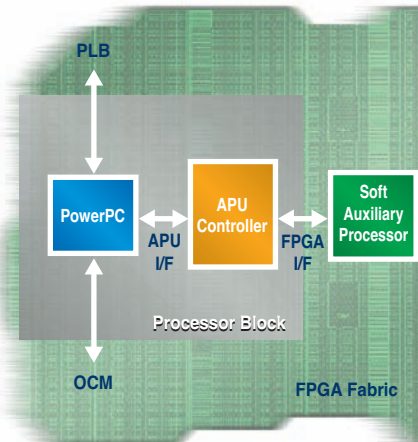


Figure 1 – APU expanded processing capabilities

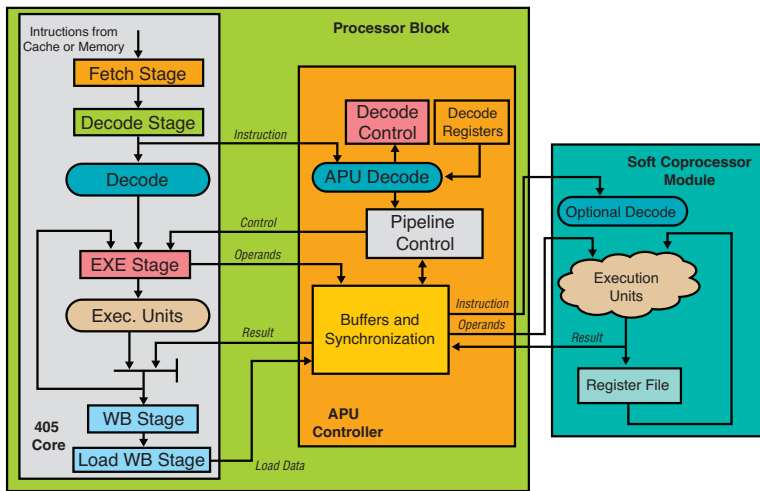


Figure 2 – APU controller processing operative block diagram

will look for a response from the APU controller. If the APU controller recognizes the instruction, it will provide the necessary information back to the CPU.

If the APU controller does not respond within that same cycle, an invalid instruction exception will be generated by the CPU. If the instruction is a valid and recognized instruction, the necessary operands are fetched from the processor and passed to the FCM for processing.

Because the PowerPC processor and the FCM reside in two separate clock domains, synchronization modules of the APU controller manage the clock frequency difference. This allows the FCM to operate at a slower frequency than the processor. In this instance, the APU controller would receive the resultant data from the coprocessor and

at the proper execution time send the data back to the processor. The APU controller knows in advance, based on instruction type, if or when it will get the result.

### Autonomous and Non-Autonomous Instructions

Two major categories of instructions exist: autonomous and non-autonomous. For autonomous instructions, the CPU continues issuing instructions and does not stall while the FCM is operating on an instruction. This overlap of execution allows you to achieve high performance through techniques such as software pipelining.

On the other hand, during the synchronized execution, the CPU pipeline stalls while the FCM is operating on an instruction. This feature allows you to

implement synchronization semantics to pace the software execution with the hardware FCM latency.

Non-autonomous instruction types are further divided into blocking and non-blocking. If blocking, asynchronous exceptions or interrupts are blocked until the FCM instruction completes. Otherwise, if non-blocking, the exception or interrupt is taken and the FCM is flushed.

### Software Description

Software engineers can access the FCM from within assembler or C code. On one side, Xilinx has enabled the GCC compiler (which is contained in the Embedded Development Kit) to generate code that uses an FCM floating-point unit to calculate floating-point operations. Furthermore, assembler mnemonics are available for UDIs and the pre-defined load/store instructions, enabling you to place hardware-accelerated functions into the regular program flow. For the ultimate level of flexibility, you can define your own instructions designed specifically for the hardware functionality of the FCM.

You can easily use the pre-defined load/store instructions through high-level C macros. For example, in an application where the FCM is used to convert pixel data into the frequency domain, 8 pixels of 16 bits are transferred from main memory to an FCM register with a simple program:

```
unsigned short pixel_row[8]; // 8 pixels,
                             // each pixel has a size of 16 bits
lqfcm(0, pixel_row); // transfer a row of
                     // pixels to FCM register zero
```

The quadword load operation maintains cache coherency as the data is moved through the cache, if caching is enabled for the corresponding address space.

The FCM operation on the pixel data can start on an explicit command; for example, a UDI. However, for many applications the operation starts immediately after the FCM hardware detects the completion of the load instruction.

The latter approach has many advantages:

- Simple software – A load operation moves the data from the memory to

the FCM and starts the operation. A subsequent store instruction retrieves the result of the operation and stores it back to main memory.

- High data transfer rates – Quadword load and store operations take just a few cycles to complete. A single operation moves 16 bytes within that timeframe.
- Low latency – FCM load operations are simple to use. The processor completes the operation in a single cycle.

The principle of the RISC architecture uses a number of simple instructions on data stored in general-purpose registers (GPR) to compute complex operations. User-defined instructions fall into this category but take the concept a step further in that the system architect defines the complexity of the operation on data stored in GPRs and FCM registers (FCR). Again, from a software point of view, the engineer codes user-defined instructions through C macros. GCC recognizes mnemonics such as `udi0fcm` as a user-defined operation of the general form:

```
udi0fcm<FCRT5/RT5>,<FCRA5/RA5/imm>,<FCRB5/RB5/imm>
```

The target of the operation is either a GPR or an FCR. The operands are either GPRs, FCRs, immediate values, or a combination. As you can see, the semantics are not defined by the instruction and depend on your intentions and the implementation in the FCM.

This code sequence demonstrates the use of a user-defined instruction as an example of a complex add operation:

```
struct complex {
    int r, i; // 32 bit integer for real and imaginary parts
};
complex a, b, r;
ldfcm(0, &a); // load complex number a into FCM register 0
ldfcm(1, &b); // load complex number b into FCM register 1
udi0fcm(2, 1, 0); // udi0fcm computes r = a + b, where r is stored in FCM register 2
stdfcm(&r, 2); // store complex result from FCM register 2 to variable r
```

To increase the readability of the code, you can redefine the user-defined instruction with regular C preprocessor constructs. Instead of using the `udi0fcm()` macro, you can redefine it to a more comprehensible `complex_add()` macro with `#define complex_add(r, a, b) udi0fcm(r, a, b)` and change the listing to call `complex_add(2, 1, 0)` instead of `udi0fcm(2, 1, 0)`.

Therefore, system architects can partition their tasks into hardware- and software-executed pieces that are efficiently and precisely interfaced to one another through the use of the APU controller. This partitioning can be done statically during the initial system configuration or dynamically during the program execution. Using the direct processor/FPGA coupling presented by the APU controller and its high throughput interfaces, hardware/software synchronization is greatly simplified and performance significantly improved.

### Accelerating System Performance

The following examples showcase key advantages the APU provides based on two different scenarios. The first scenario is essentially a benchmarking comparison of a finite impulse response (FIR) filter using a soft FPU core, implemented as an FCM attached directly to the APU controller (as compared to software emulation used to calculate the filter function). The second scenario implements a two-dimensional

inverse discrete cosine transform (2D-IDCT) typically used as one of the processing blocks in MPEG-2 video decompression, again compared to emulating the 2D-IDCT function in software.

The two use cases are different in that the FPU implements a set of registers in the FPGA fabric upon which the FPU instructions operate. The 2D-IDCT only requires load and store operations, while the functionality of the operation on the data stream is fixed. In either case the operations are complex enough to justify offloading into the FPGA fabric.

Thus, the combination of using the APU and FPGA hardware acceleration clearly provides a significant performance advantage over software emulation – or the conventional method involving the processor and processor local bus architecture with a soft co-processing function.

### FIR Filter

The implementation of floating-point calculations in hardware yields an improvement by a factor of 20 over software emulation. Connecting the FPU as an FCM to the APU controller provides performance improvement because the latency to access the floating-point registers is reduced and dedicated load and store instructions move the operands and results between the FPU registers and the system memory.

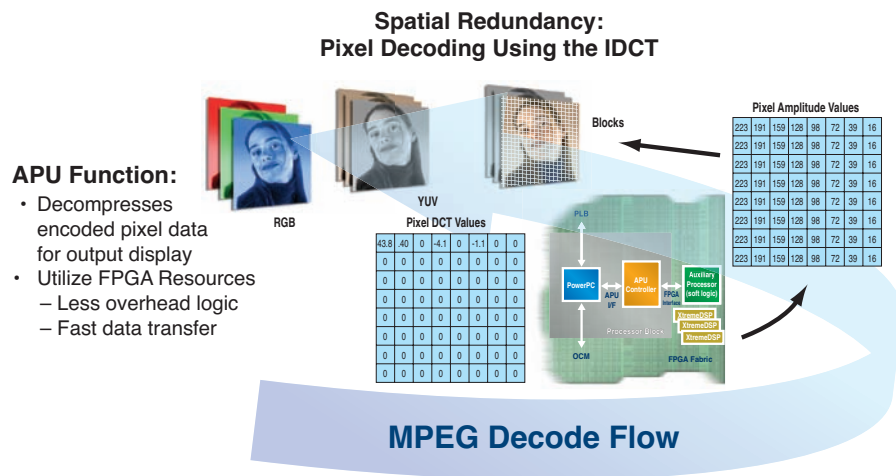
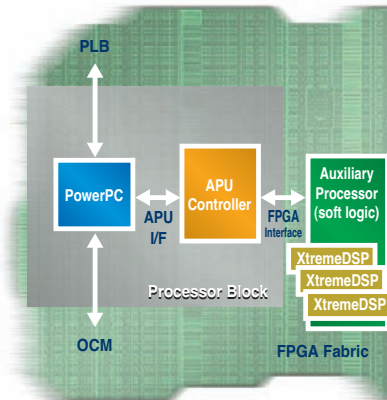


Figure 3 – Utilizing APU to decode pixel data for display output

## Example: Video Application – MPEG De-Compression Algorithm

- **Leverages Integrated Features**
  - PowerPC, APU, XtremeDSP Blocks
- **HW Acceleration Over Software**
  - Lower Latency and High Bandwidth
- **Efficient HW/SW Design Partitioning**
  - Optimized Implementation
- **Significant Performance Increase**



Over 20X Performance Improvement Compared to Software Emulation

Figure 4 – Accelerated system performance with APU

### 2D-IDCT

The 2D-IDCT transforms a block of 8 x 8 data points from the frequency domain into pixel information. A high-level diagram depicting the pixel decode by the APU controller, along with advantages, is shown in Figure 3. In this example, each data point has a resolution of 12 bits and is represented as a 16-bit integer value. The data structure is defined where each row of 8 pixels consumes 16 bytes. This is an ideal size that allows optimal use of the FCM load and store instructions described earlier. In other words, eight FCM quadword load instructions are needed to load a data block into the 2D-IDCT hardware. Eight FCM quadword store instructions are sufficient to copy the pixel data back into the system memory.

The calculation of the 2D-IDCT in the FCM starts immediately after the first load, and the pixel data is available shortly after the last load operation. As shown in Figure 4, the 2D-IDCT makes use of the new XtremeDSP™ slices in the Virtex-4 architecture that offer multiply-and-accumulate functionality.

A software-only implementation of a 2D-IDCT takes 11 multiplies and 29 additions together with a number of 32-bit load and store operations, while the hardware-accelerated version takes 8 load and 8 store operations. The reduced number of operations results in a speed-up of 20X in favor of a 2D-IDCT FCM attached through the APU controller.

By comparison, if you connect the 2D-IDCT hardware block to the processor local bus, as it is done conventionally, the system performance will be reduced. This increased latency is mainly caused by the bus arbitration overhead and the large number of 32-bit load and store instructions. This is illustrated schematically in Figure 5.

### Conclusion

The low-latency and high-bandwidth fabric coprocessor module interface of the APU controller enables you to accelerate algorithms through the use of dedicated hardware. Where operations are complex enough to justify the offloading into the FPGA fabric, or when acceleration of a

specific algorithm is desired to achieve optimal performance, the combination of the APU controller and FPGA hardware acceleration provides a definitive performance advantage over software emulation or the conventional method of attaching coprocessors to the processor memory bus.

Generating the accelerated functions called by user-defined instructions is easily performed through GUI-based wizards. This functionality will be included in subsequent releases of the powerful Embedded Development Kit or Platform Studio.

If you are more comfortable working at the source code or assembly level, the APU controller allows you to define your own instructions written specifically for the hardware functionality of the FCM, or you can easily use the pre-defined load/store instructions through high-level C macros.

The APU controller provides a close coupling between the PowerPC processor and the FPGA fabric. This opens up an entire range of applications that can immediately benefit customers by achieving increases in system performance that were previously unattainable.

For additional details on the APU controller in Virtex-4-FX devices, including detailed descriptions and timing waveforms, refer to the Virtex-4 PowerPC 405 Processor Block Reference Guide at [www.xilinx.com/bvdocs/userguides/ug018.pdf](http://www.xilinx.com/bvdocs/userguides/ug018.pdf).

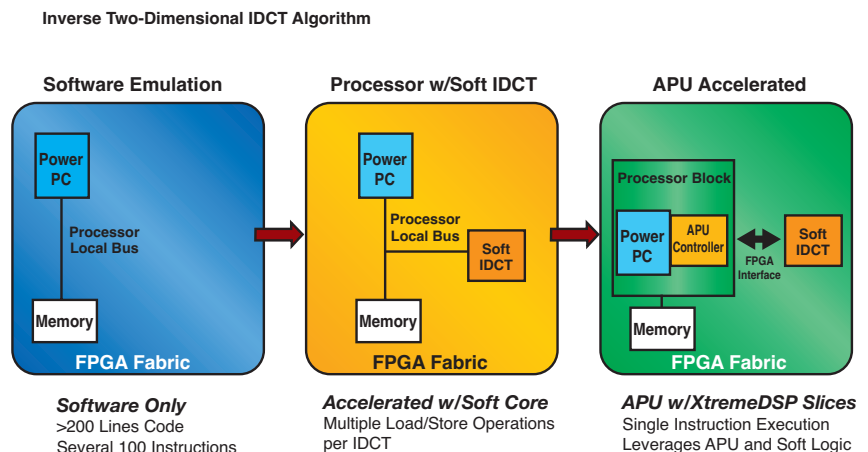


Figure 5 – Comparison of implementation models for 2D-IDCT