XAPP717 (v1.1.1) Sept. 29, 2005

Accelerated System Performance with the APU Controller and XtremeDSP Slices

Author: Harn Hua Ng and Latha Pillai

Summary

Portions of certain software applications that are implemented in software can run faster by moving the implementation into hardware. For example, in a Virtex™-4 FX FPGA, the embedded PowerPC™ 405 (PPC405) processor can run software and offload computations to hardware modules in the FPGA. In such a system, a coprocessor interface known as the Auxiliary Processor Unit (APU) is used to transfer data between the processor and XtremeDSP™ (DSP48) slices in the FPGA. Because certain computations can be done more efficiently in software, and others in hardware, an APU-enhanced system results in a faster overall solution for many digital signal processing (DSP) applications.

Introduction

This application note introduces the APU and describes the main features of an APU-enhanced system. Included examples illustrate how the APU transfers data between the processor and the FPGA. The two examples are:

- A simple system that moves data from memory through the processor and APU, into registers in the FPGA, and back into memory
- A demo video application that performs the Inverse Discrete Cosine Transform (IDCT) portion of MPEG decoding using the APU to offload computations

The video application uses the XtremeDSP capabilities of Virtex-4 FPGAs to implement an IDCT component for image decoding. Combined with the APU, users can increase the performance of a video application by offloading IDCT computations that were previously done in software to an IDCT module in the FPGA. The numerous computations involved in IDCT make it a candidate for performance improvement with the APU and XtremeDSP modules.

Users have the option of either simulating or implementing the examples in real hardware. The examples that accompany this application note use the Xilinx ML403 evaluation platform. The xapp717.zip file includes Verilog and C source files as well as FPGA bitstreams and software executables.

Included Files

Table 1 lists some of the reference design files contained in the zip file located at http://www.xilinx.com/bvdocs/appnotes/xapp717.zip.

Table 1: **Reference Design Directory and Files**

```
xapp717/
        |--hw/
        |     |--apu_idct/ (XPS project for IDCT demo)
        |     |           |--system.xmp       (XPS project file)
        |     |           |--system.mhs       (XPS hardware description file)
        |     |           |--download.bit     (FPGA bitstream)
        |     |           |--apu_idct_sim.elf (Software executable for simulation)
        |     |           |--apu_idct.elf     (Software executable for hardware)
        |     |
        |     |--apu_loadstore/ (XPS project for FCM Register Load/Store Example)
        |     |                |--apu_loadstore.xmp     (XPS project file)
        |     |                |--apu_loadstore.mhs     (XPS hardware description file)
        |     |                |--download.bit          (FPGA bitstream)
        |     |                |--apu_loadstore_sim.elf (Software executable for simulation)
        |     |                |--apu_loadstore_hw.elf  (Software executable for hardware)
        |
        |--sw/ (C source code for XPS projects)
        |     |--standalone/
        |     |             |--apu_idct/ (IDCT demo)
        |     |             |           |--src/
        |     |             |           |     |--apu_idct.c
        |     |             |           |     |--apu_idct_sim.c
        |     |             |           |     |--bootload_basicgraphics.c
        |     |             |           |     |--bootload_basicgraphics.h
        |     |             |           |     |--xrom_lcd.c
        |     |             |           |     |--xrom_lcd.h
        |     |             |
        |     |             |--apu_loadstore/ (FCM Register Load/Store Example)
        |     |             |                |--src/
        |     |             |                |     |--apu_loadstore_sim.c
        |     |             |                |     |--apu_loadstore_hw.c
```

Table 2 lists the image files available on the demos and reference design area of the ML403 website, located at http://www.xilinx.com/products/boards/ml403/reference_designs.htm.

Table 2: **Pre-Built Image Files**

File	Description
apd.xdct.bin	Sample image for the IDCT demo
apu_idct.img.zip	Compressed 512 MB CompactFlash image with IDCT demo and image data

Required Hardware/Tools

- Xilinx ML403 embedded platform
- Xilinx ISE 7.1i (Service Pack 2 through Service Pack 4)
- Xilinx Platform Studio 7.1i (Service Pack 2)
- ModelSim (6.0a through 6.0e)
- VGA monitor

Overview of the Fundamentals

Introduction to the APU

The APU allows the designer to extend the native PowerPC 405 instruction set with custom instructions for execution by an FPGA Fabric Coprocessor Module (FCM). An APU-enhanced system enables tighter integration between an application-specific function and the processor pipeline, making the APU implementation superior to, for example, a bus peripheral.

When an instruction arrives, the processor and the APU decode it simultaneously. If the instruction is meant for the APU and the FCM, the APU relays it to the FCM. Different types of instructions affect whether the processor waits for the FCM to finish executing the instruction. For example, for an FCM write instruction, the processor requests data from the FCM and writes it into a processor register. Hence, the processor must wait for output data from the FCM before executing the next instruction. One important point is that the APU only decodes the instructions and does not execute them.

Another function of the APU is clock domain synchronization between the processor clock and the FCM clock. The maximum clock frequency of the logic in the FCM is typically less than that of the processor block. Hence, when the processor clock is faster than the FCM clock, the APU keeps signals in both clock domains synchronized with one another.

Figure 1 shows the pipeline flow between the PPC405 core, the APU controller, and the FCM.

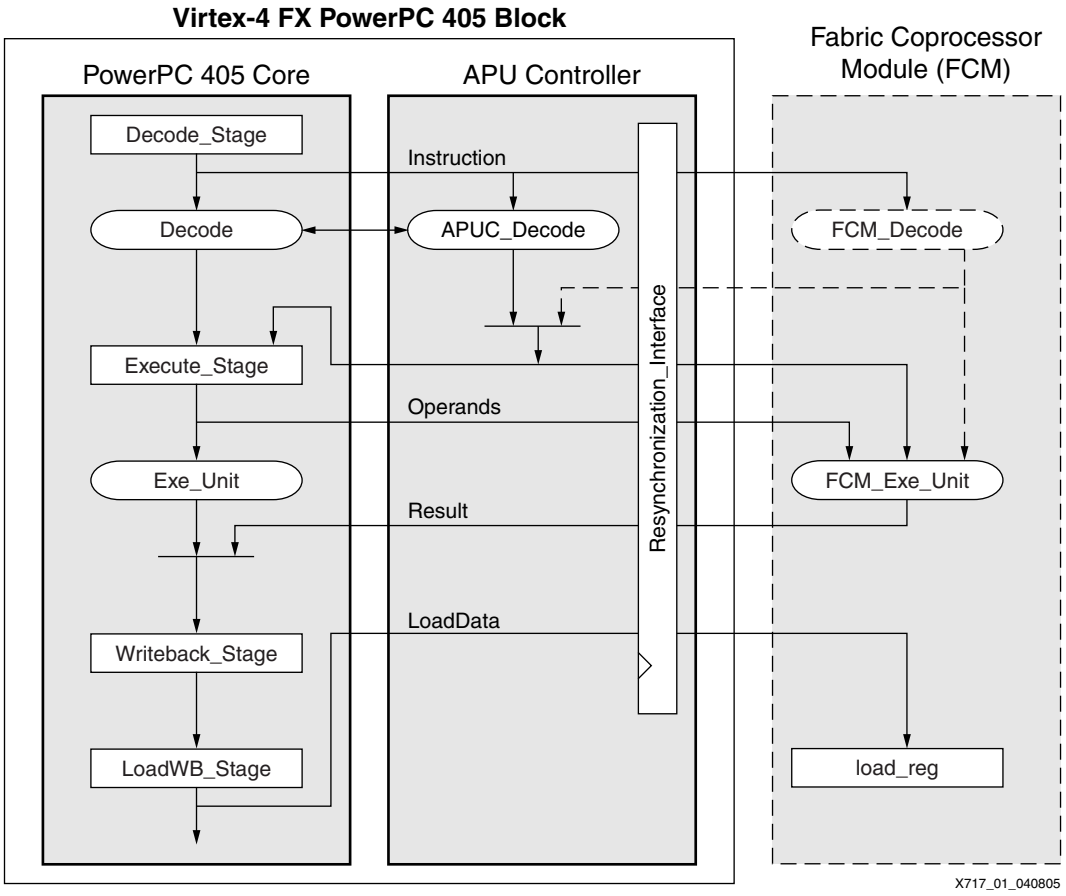

Figure 1: **Pipeline Flow Diagram**

Introduction to the FCM

The FCM represents all modules in the FPGA that interact with the APU controller. It typically runs at a lower clock frequency than that of the processor and APU. In general, the processor clock to FCM clock ratio can be in integer multiples such as 1:1, 2:1, 3:1, ..., 16:1. During operation, the FCM can also signal an exception to the processor.

When the APU passes an instruction to the FCM, the FCM must decode and execute it. For example, the FCM in the demo video application is composed of an IDCT module, a decoder, and two state machines. After decoding the instruction, the state machines handle data transfers between the APU and the IDCT module, using registers to store data for the IDCT operations. Data is transferred between the APU to the FCM via FCM *load* and FCM *store* instructions. The data is then transferred between the FCM to the IDCT module via a user-defined protocol.

Extended Instruction Set

The instructions that the APU and the FCM decode belong to an extension of the native PowerPC 405 instruction set. Two types of instructions, *pre-defined* and *user-defined*, are supported by the APU. A pre-defined instruction is hard-coded into the APU, and a user-defined instruction (UDI) is configurable by the user. For a UDI, the designer is able to customize the instruction's behavior and to configure the APU to decode it. This application note uses pre-defined FCM load/store instructions.

Based on the processor's behavior, pre-defined and user-defined instructions can be categorized into *autonomous* (do not stall processor's pipeline) and *non-autonomous* (stall processor's pipeline) instruction classes. For details on instruction classes, types, and formats, refer to *Chapter 4, PowerPC 405 APU Controller* of the *PowerPC 405 Block Reference Guide*[3].

APU Port List

The *PowerPC 405 Block Reference Guide*[3] also describes the APU ports through which the FCM transfers instructions and data between the processor, the APU, and the FPGA. Both the FCM register load/store example and the demo IDCT system use a subset of these ports. Table 3 shows the FCM output signals used in this application note.

Table 3: **FCM Output Signals**

Signal	Definition
FCMAPURESULT[0:31]	The FCM execution result that is passed to the processor through the APU controller.
FCMAPURESULTVALID	Indicates a valid FCMAPURESULT value. As a result, the APU receives the FCM's outputs and relays them to the processor to be stored into memory.
FCMAPUDONE	Indicates the completion of the instruction in the FCM to the APU controller. When the FCM completes an FCM load or an FCM store operation, it signals the APU that it is ready to receive the next instruction (if there is one).
FCMAPULOADWAIT	The FCM is not ready to receive the next load data from the APU. When each word of a multiple-word FCM load operation is sent from the APU to the FCM, the FCM cannot write each word into an FCM internal register file quickly enough to keep up with the APU. Hence, this signal is asserted to have the APU continue to assert the current word until the FCM is ready to read it.

Table 4 shows the FCM input signals used in this application note.

Table 4: **FCM Input Signals**

Signal	Definition
APUFCMINSTRVALID	This signal is asserted on two conditions: 1. A valid APU instruction is decoded by the APU controller. 2. An undecoded instruction is passed to the FCM for decoding. The FCM depends on this signal to detect an incoming instruction.
APUFCMDECODED	Asserted when the APU controller decoded the instruction before it is sent to the FCM. This signal tells the FCM that the APU is configured to recognize the incoming instruction.
APUFCMINSTRUCTION[0:31]	The instruction presented to the FCM. This signal is valid as long as APUFCMINSTRVALID is High.
APUFCMLOADDATA[0:31]	Data word loaded from storage to the APU register file. During an FCM load operation, data passes from the processor to the APU and finally to the FCM through this port.
APUFCMLOADDVALID	When asserted, the data word on the APUFCMLOADDATA[0:31] port is valid.

More complex transactions can be implemented using the entire set of ports and are not within the scope of this application note. See "APU-Enhanced IDCT," page 14 for additional details on the logic in the FCM module.

APU-to-FCM Interface

The FCM interacts with the APU through the APU's ports. When an FCM is first created, the designer must decide which interface to use between the APU and the FCM. The simplest option is a direct connection between the two, resulting in a one-to-one link. By observing the timing relationships of the input and output signals, the designer can create a customized direct connection to link one or multiple FCMs to the APU.

In a system with multiple FCMs, it can be helpful to use an interface provided by XPS called the Fabric Coprocessor Bus (FCB). The FCB is a multiplexer that allows the APU to be connected to more than one FCM. Using XPS, an FCB can be readily instantiated and deployed in a system, saving the need to create an APU-to-FCM interface from scratch. One restriction of the FCB core is that each FCM slave must decode a unique set of instructions. Figure 2 shows a direct connection interface, followed by an FCB interface.

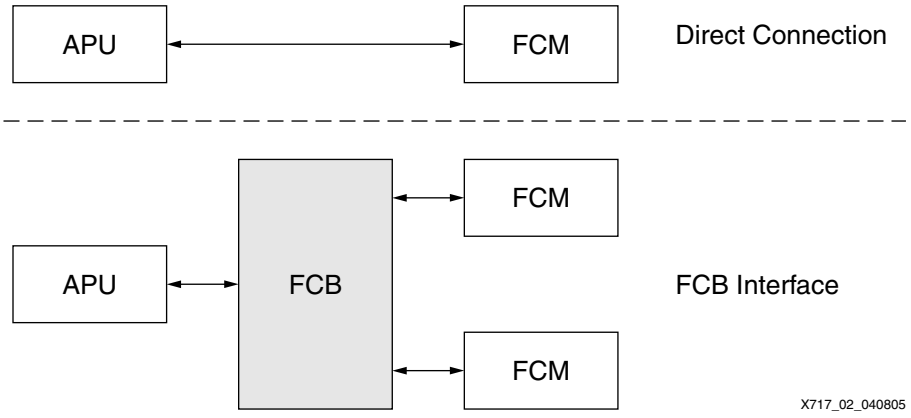

Figure 2: **APU-to-FCM Interfaces**

See "FCM Register Load/Store Example," page 6 for a description of a system that uses a direct connection. See "APU-Enhanced IDCT," page 14 for a description of a system that uses an FCB interface.

APU Initialization

To enable the APU, bit 6 of the processor's Machine State Register (MSR) must be set. This is done in software with a `mtmsr(XREG_MSR_APU_AVAILABLE)` function provided by XPS. See Table 4-1 in the *PowerPC 405 Block Reference Guide*[3] for details.

Next, the APU's decoding and instruction-handling behavior must be set using either of the following methods:

1. Statically through the TIEAPUCONTROL and TIEAPUUDIn (n = 1, 2, …, 8) ports of the processor module
2. Dynamically through the Device Control Register (DCR) -mapped APU Configuration and UDI Configuration Registers

   Note: UDI configuration is optional and only needed when UDIs are deployed in the system.

Refer to Tables 2-20, 4-4, 4-5, 4-9 and 4-10 in the *PowerPC 405 Block Reference Guide*[3] for details on APU initialization and configuration.

FCM Register Load/Store Example

The FCM load/store example shows how data is moved from memory through the processor and APU, into the FCM, and back into memory. Figure 3 shows an overview of the FCM register load/store system.

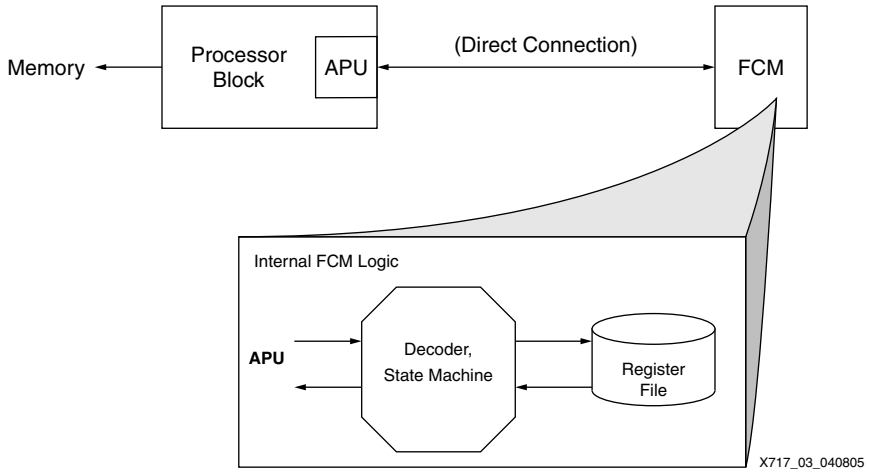

Figure 3: **FCM Register Load/Store System Overview**

In this system, the flow of data is as follows:

1. The processor forwards an FCM *load double-word* instruction to the APU.
2. The APU passes the instruction to the FCM, which decodes the FCM load and waits for data from memory to arrive via the APU. Figure 4, page 7 shows the timing relationships between signals in the APU-to-FCM interface for a multiple-word FCM load.

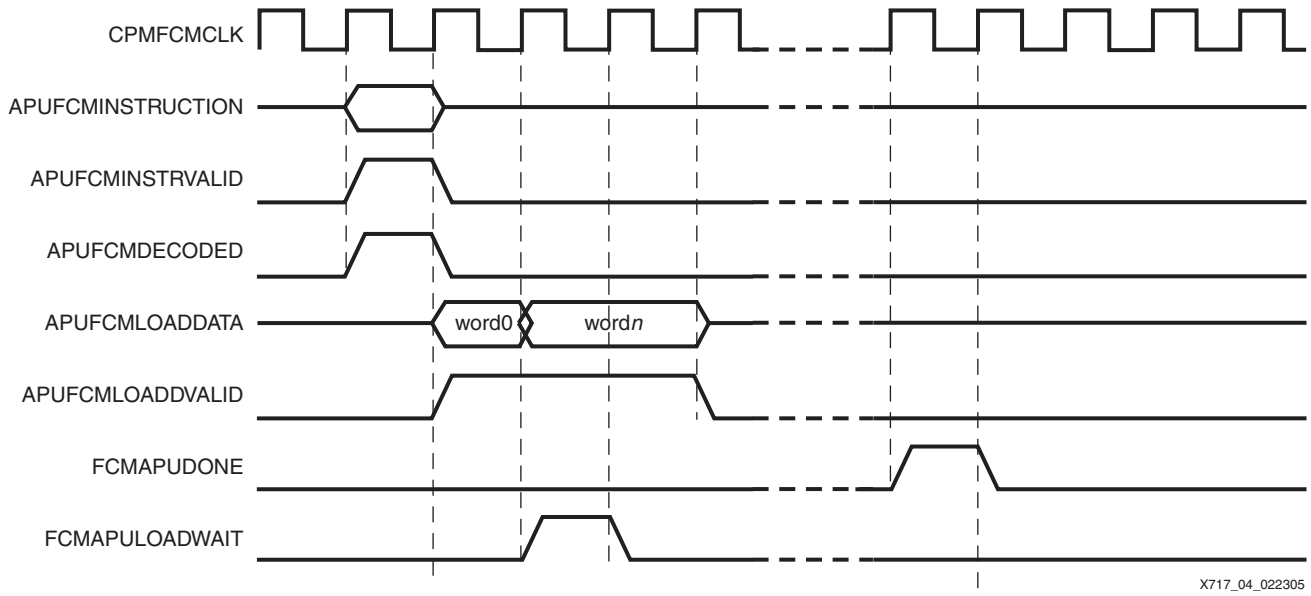

Figure 4: **Timing Diagram for Multiple-Word FCM Load**

3. As each word is received from the APU, the FCM writes it to a register file. Eventually, eight bytes of data are stored in this register file.
4. Next, the processor forwards an FCM *store double-word* instruction to the APU.
5. The FCM decodes the FCM store and reads eight bytes of data from the target register.
6. The FCM then returns the data to the APU which relays it to the processor.
7. Finally, the processor writes the data back into memory. Figure 5 shows the timing relationship between signals in the APU-to-FCM interface for a multiple-word FCM store operation.

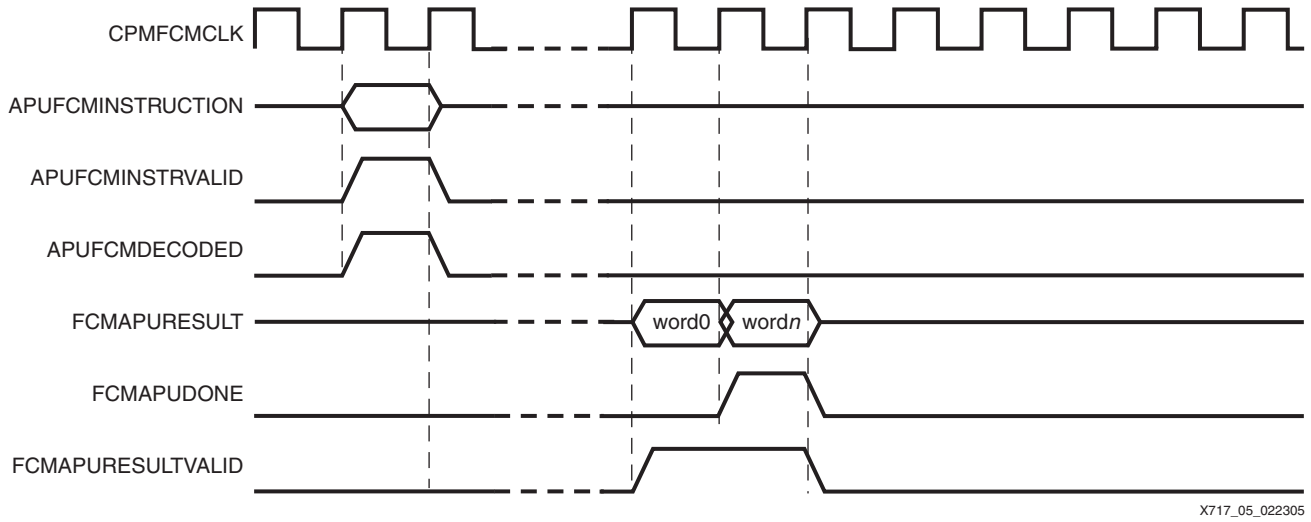

Figure 5: **Timing Diagram for Multiple-Word FCM Store**

FCM Register Load/Store Example State Machine

In the FCM, a state machine, shown in Figure 6, handles the FCM load/store operations and communicates with the APU.

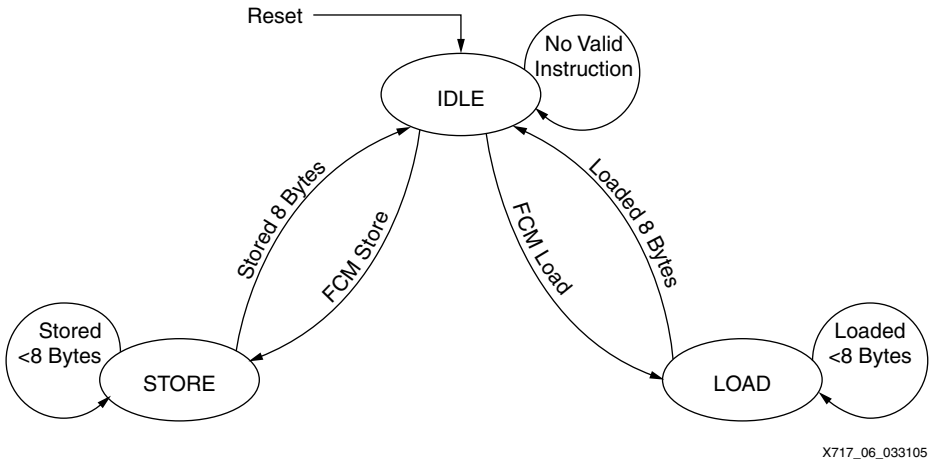

Figure 6: **FCM Register Load/Store State Machine**

Verilog Source Code for the FCM Register Load/Store Example

Table 5, page 9 shows excerpts of the Verilog code for this example. The decode_ldst module parses instructions from the APU and relays the instruction type, the data transfer size and other details to the state machine. Using the inputs from the APU and the decode_ldst module, the state machine decides if there is a valid load/store instruction to execute. Each state and input signal combination influences the output signals of the FCM module, asserting FCMAPUDONE, FCMAPURESULTVALID, FCMAPURESULT, and FCMAPULOADWAIT accordingly.

During the IDLE state, if there is a valid instruction, the state machine transitions to the appropriate LOAD or STORE state based on the decode_ldst module's output. The FCM remains in a LOAD or STORE state until all data has been transferred, after which the state machine returns to the IDLE state.

If the current instruction is a load instruction and load-data arrive in the same clock cycle as the APUFCMINSTRVALID and APUFCMDECODED signals, the FCM asserts the FCMAPULOADWAIT signal to ask the APU to hold the data for an additional clock cycle.

Table 5: **Verilog Code for the FCM Register Load/Store Example**

```
/************************ combinational blocks ************************/
// decoder
decode_ldst decode_ldst_0
  ( // outputs
    .update(ldst_update),
    .size(ldst_size),
    .store_or_loadn(store_or_loadn),
    .valid_ldst(ldst_valid),
    // inputs
    .APUFCMINSTRUCTION(APUFCMINSTRUCTION)
  );

// state machine logic
always @(curr_state or store_or_loadn or store_or_loadn_reg or
         ldst_size_counter or ldst_size_reg or
         APUFCMINSTRVALID or APUFCMLOADDVALID or APUFCMDECODED)
  begin
    case (curr_state)
      STATE_IDLE: // wait for valid instruction
        if (APUFCMINSTRVALID & APUFCMDECODED) // valid instruction
          if (store_or_loadn) // store instruction
            next_state = STATE_STORE;
          else // load instruction
            if (APUFCMLOADDVALID) // load data arrived at the same time
              if (ldst_size_counter < ldst_size_reg)
                next_state = STATE_LOAD;
              else
                next_state = STATE_IDLE;
            else
              next_state = STATE_LOAD;
          else
            next_state = STATE_IDLE;

      STATE_LOAD: // seen a valid load instruction, wait for valid data
        // keep track of how many words to access
        if (ldst_size_counter < ldst_size_reg)
          next_state = STATE_LOAD;
        else
          if (APUFCMLOADDVALID)
            next_state = STATE_IDLE;
          else
            next_state = STATE_LOAD;

      STATE_STORE: // seen a valid store instruction, output data
        // keep track of how many words to access
        if (ldst_size_counter < ldst_size_reg)
          next_state = STATE_STORE;
        else
          next_state = STATE_IDLE;

      default:
        next_state = STATE_IDLE;

    endcase // case(curr_state)
  end // always @ (curr_state or store_or_loadn or ...

// output assignments
// assert done when all data has been written/transferred
assign FCMAPUDONE = (((curr_state==STATE_LOAD) & APUFCMLOADDVALID &
                      (ldst_size_counter==ldst_size_reg)) |
                     ((curr_state==STATE_STORE) &
                      (ldst_size_counter==ldst_size_reg)) |
                     ((curr_state==STATE_IDLE) & APUFCMLOADDVALID &
                      (ldst_size_counter==ldst_size_reg))) ? 1'b1:1'b0;
// result to return to APU
assign FCMAPURESULT = regfile_rdata;
// return value is valid
assign FCMAPURESULTVALID = (curr_state==STATE_STORE);
// ask APU to hold load data because FCM is not ready to process it
assign FCMAPULOADWAIT = ((curr_state==STATE_IDLE) & APUFCMLOADDVALID);
```

XPS: FCM Register Load/Store Example MHS File

In an XPS project, the microprocessor hardware specification (MHS) file is where hardware modules are instantiated. Table 6 shows excerpts from the FCM register load/store project's apu_loadstore.mhs file, in particular the sections where the processor module and the FCM are declared.

Table 6: **Excerpts from hw/apu_loadstore/apu_loadstore.mhs**

```
# Virtex-4 PPC405 module
BEGIN ppc405_virtex4
  ...
  PARAMETER C_APU_CONTROL = 0x0001
  PORT CPMFCMCLK = sys_clk_s
  # apu, fcm signals
  PORT FCMAPURESULT = fcmapuresult
  PORT FCMAPUDONE = fcmapudone
  ...
  PORT APUFCMRADATA = apufcmradata
  PORT APUFCMRBDATA = apufcmrbdata
  ...
END
```

```
# FCM load/store module
BEGIN fcm_loadstore
  ...
  PARAMETER HW_VER = 1.00.a
  PORT clock = sys_clk_s
  PORT reset = sys_bus_reset
  # apu, fcm signals
  PORT FCMAPURESULT = fcmapuresult
  PORT FCMAPUDONE = fcmapudone
  ...
  PORT APUFCMRADATA = apufcmradata
  PORT APUFCMRBDATA = apufcmrbdata
  ...
END
```

In Table 6, PARAMETER C_APU_CONTROL corresponds to the TIEAPUCONTROL port of the processor module. Bit 15, the FCM enable bit, is set, implying that the resulting FPGA bitstream is statically configured to have the APU support FCM operations. Both the ppc405_virtex4 and fcm_loadstore modules share clock, reset, and APU port signals.

XPS: The FCM as a Pcore in the FCM Register Load/Store Example

Because the fcm_loadstore module is not a built-in peripheral core (pcore) in XPS, it is included in the design as a user pcore. Refer to *Chapter 4, Create and Import Peripheral Wizard* of the *Embedded System Tools Reference Manual*[7] for details on creating a user pcore. The fcm_loadstore pcores in the xapp717/hw/apu_loadstore/pcores directory can be used as references for creating a custom FCM pcore.

The following FCM pcores are included:

- Version 1.00.a that uses a direct connection between the APU and the FCM
- Version 1.00.b that uses an FCB interface

Both FCMs have identical functionality, and the FCM register load/store example uses version 1.00.a by default.

C Source Code for the FCM Register Load/Store Example

Table 7 shows the C source code for the FCM register load/store example.

Table 7: **FCM Register Load/Store Example C Code**

```
#include "xbasic_types.h"
#include "xcache_l.h"
#include "xparameters.h"
#include "xpseudo_asm.h"

#define ldfcmx(rn, base, adr)     __asm__ __volatile__(\
                                  "ldfcmx " #rn ",%0,%1\n"\
                                  : : "b" (base), "r" (adr)\
                                  )

#define stdfcmx(rn, base, adr)     __asm__ __volatile__(\
                                   "stdfcmx " #rn ",%0,%1\n"\
                                   : : "b" (base), "r" (adr)\
                                   )

// Data structures
volatile Xint32 __attribute__ ((aligned (32))) src[4] = {214,49,-3,20};
volatile Xint32 __attribute__ ((aligned (32))) dst[4] = {-1,-1,-1,-1};

int main(void)
{
  // initialize caches
  XCache_EnableDCache(0x80000001);
  XCache_EnableICache(0x80000001);
  // initialize APU
  mtmsr(XREG_MSR_APU_AVAILABLE);

  // load double-word from src[0] to FCM reg 2
  ldfcmx(2, src, 0);
  // store double-word from FCM reg 2 to dst[0]
  stdfcmx(2, dst, 0);

  // move data to GPR14 to see it in simulation
  mtgpr(14,((unsigned int *)dst)[0]);
  mtgpr(14,((unsigned int *)dst)[1]);
  return 0;
}
```

At this time, XPS does not provide software drivers for FCM load/store instructions, so in this example, they are defined as pseudo-assembly instructions. For information on how to declare inline assembly instructions, refer to *Section 5.35* of the *GNU GCC 3.4.3 Manual*[6]. These declarations allow the designer to invoke assembly instructions using a C-like syntax. In the **`main()`** function, the APU is enabled by setting bit 6 of the MSR, as covered in "APU Initialization," page 6. Then FCM load/store instructions are invoked to perform the data transfers. The final result is that elements 0 and 1 in the *src* array are copied to elements 0 and 1 in the *dst* array.

The subsequent sections explain how to set up the project files, tools, and hardware to simulate and implement the FCM register load/store example.

Setting Up the Tools/Hardware

1. Connect an ML403[3] board, PC, and download cable after verifying that the appropriate software is installed on the PC. See "Required Hardware/Tools," page 2.
   a. Connect power to the ML403 board.

b. Connect a Parallel Cable IV (PC4) cable from the board to a PC.
c. Connect a serial cable from the board to the PC.

2. Set up the serial terminal program.
   a. On the PC, open HyperTerminal or any other terminal application.
   b. Start a connection to the serial port that connects to the ML403 board.
   c. Set the terminal parameters as follows:
      - Baud rate: 9600
      - Data: 8-bit
      - Parity: none
      - Stop: 1 bit
      - Flow control: none
3. Extract the files from the xapp717.zip archive.

   The resulting directory, **xapp717/hw**, contains XPS projects for all of the examples and demos in this application note. The project directory for the FCM register load/store example is xapp717/hw/apu_loadstore.
4. Start Xilinx Platform Studio (XPS) and open the apu_loadstore project:

   File → **Open** → **apu_loadstore.xmp**
5. Generate the software executables.

   Tools → **Build All User Applications**

   Software executable for simulation:
   xapp717/hw/apu_loadstore/ppc405_0/code/apu_loadstore_sim.elf

   Software executable for hardware:
   xapp717/hw/apu_loadstore/ppc405_0/code/apu_loadstore_hw.elf
6. Set up the simulation environment.

   This process involves compiling simulation libraries for ISE and EDK. Choose a simulation tool; this application note uses ModelSim 5.8e.
   a. Compile the ISE simulation libraries.

      Refer to *Chapter 6, Simulating Your Design* in the *Synthesis and Verification Design Guide* for ISE 7.1i[1] for instructions on how to compile simulation libraries for ISE.

      For example, in a Linux environment:

      ```
      % cd /home/mti_58e/ISE_lib
      % compxlib -f all -l all -s mti_se .
      ```
   b. Compile the EDK simulation libraries.

      Refer to *Chapter 6, Simulation Model Generator* in the *Embedded System Tools Reference Manual* for EDK 7.1i[2] to learn how to compile simulation libraries for EDK.

      For example:

      ```
      % cd /home/mti_58e/EDK_lib
      % compedklib -s mti_se -o . -X ../ISE_lib
      ```
   c. Check that the paths to the simulation libraries correspond to the ISE/EDK libraries created in Steps 6a and 6b.
      - **Options** → **Project Options...** and click the HDL and Simulation tab.
      - Click **OK**.
   d. Ensure that the correct software application is used for simulation.
      - Click the Applications tab.

- Right-click on the software project, apu_loadstore_sim, and select **Mark to Initialize BRAMs**.
- Verify that all other software projects do not have the **Mark to Initialize BRAMs** option highlighted in green.

7. Generate simulation files by choosing **Tools** → **Generate Simulation HDL files**.
8. Generate the FPGA bitstream.
   a. Ensure that the correct software application is used for the bitstream.
      - Click the Applications tab.
      - Right-click on the software project, ppc405_0_bootloop, and select **Mark to Initialize BRAMs**.
      - Verify that all other software projects do not have the **Mark to Initialize BRAMs** option checked (or highlighted in green).
   b. **Tools** → **Update Bitstream**

      This process takes approximately 5 minutes on a 3.6 GHz machine.

Running the FCM Register Load/Store Example Simulation

1. Launch ModelSim 5.8e.
2. At the prompt, enter the following to compile the simulation:

```
cd xapp717/hw/apu_loadstore
cd simulation/behavioral
do ../../data/testbench.do
```

3. When the waveform viewer pops up, enter **run -all** to run the simulation.
4. In the waveform viewer, observe the signals in the following section:
   - APU-to-FCM Interface

   The signals match up to those in the timing diagrams in Figure 4 and Figure 5, page 7.

Running the FCM Register Load/Store Example on the ML403 Board

1. Use ISE iMPACT to program the FPGA on the ML403 board with xapp717/hw/apu_loadstore/implementation/download.bit by way of the PC4 cable.
2. Connect to the processor core using the Xilinx Microprocessor Debugger (XMD).
   a. In XPS, select **Tools** → **XMD** to start an XMD session.
   b. Connect to the PPC405:

```
XMD% connect ppc hw
```

   c. Reset the system:

```
XMD% rst
```

3. Download and run the apu_loadstore_hw.elf software executable:

```
XMD% dow ppc405_0/code/apu_loadstore_hw.elf
XMD% run
```

Figure 7 shows the output on the serial terminal.

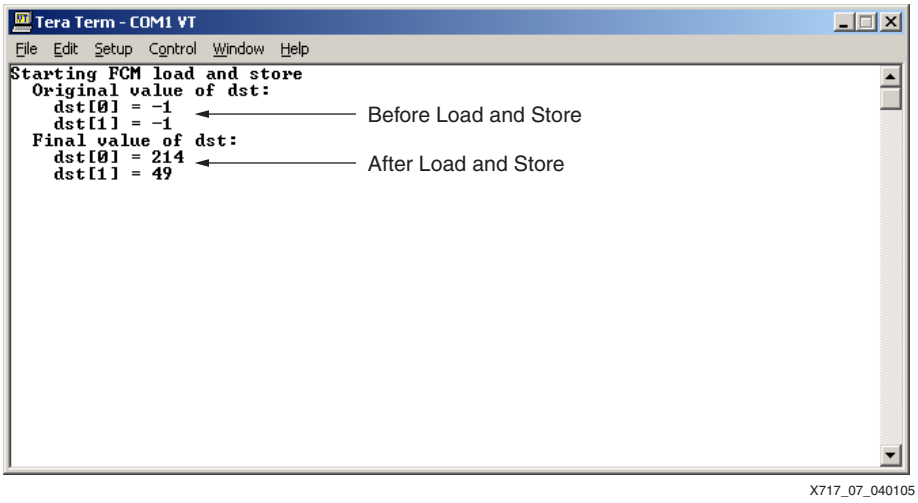

Figure 7: **Serial Terminal Output**

Summary of the FCM Register Load/Store Example

This example helps designers get started with using the APU to transfer data between the processor and the FPGA logic in a system. Although a direct connection is used between the APU and the FCM, another XPS project, apu_loadstore_fcb.xmp, is included in the same xapp717/hw/apu_loadstore directory to illustrate how the designer can replace a direct connection with an FCB interface.

APU-Enhanced IDCT

The "FCM Register Load/Store Example," page 6 shows how to create, debug, and implement a simple APU-enhanced system. The remainder of this application note covers a more complex system: a video application with APU-accelerated IDCT.

Figure 8 shows an overview of the APU-enhanced IDCT system.

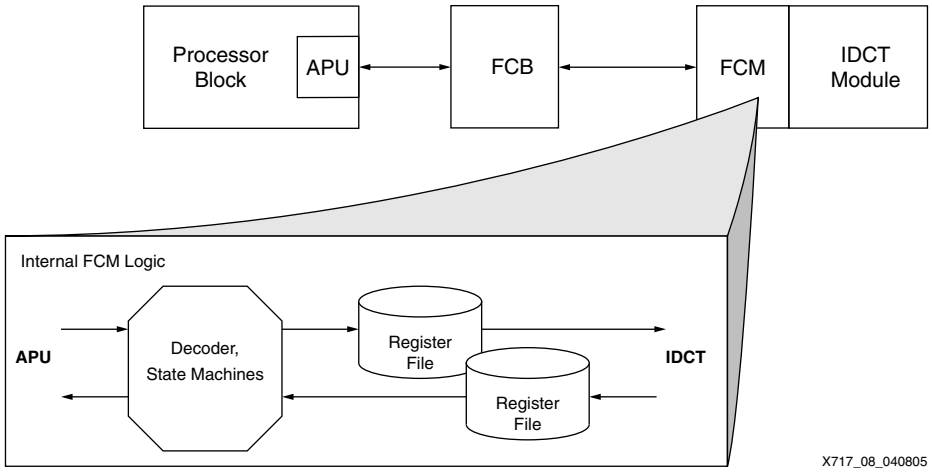

Figure 8: **APU-Enhanced IDCT System Overview**

IDCT Implementation

In digital signal processing, compression reduces the size of the data sent to a computation module, thereby reducing the bandwidth required for the digital representation of a signal. Compression can reduce transmission times as well as storage requirements. The Discrete

Cosine Transform (DCT) is a step in image compression that is reversed by the IDCT upon reception of DCT-transformed data. DCT and IDCT are two of the most computation-intensive functions in image encoding and decoding. Therefore, a fast and optimized DCT/IDCT implementation is essential in improving the performance of both the video encoder and decoder.

The basic operation of a two-dimensional (2D) IDCT is as follows:

If X is the input 8 x 8 DCT matrix, and Y is the output IDCT matrix, then $Y = C^t \cdot X \cdot C$.

The matrix multiplication is split into two parts. $C^t \cdot X$ (or $X \cdot C$) is calculated first, resulting in a one-dimensional (1D) IDCT. The results are obtained one column at a time and stored in memory. The output of the memory is usually read out one row at a time, a method referred to as corner-turn memory. This output is then multiplied by C (or C^t) to obtain the final 2D IDCT values. Corner-turn memory uses up resources and clock latency. The hardware IDCT implementation in this application note uses no corner-turn memory, thus making it more efficient in both speed and resource usage.

Software IDCT

In the software IDCT implementation, all IDCT operations are performed in software using C code from a Linux open source video player known as *xine*. This demo video application utilizes the xine IDCT routines in a stand-alone program for demonstrating system performance in software. (See "Running the Demo Video Application on the ML403 Board," page 24.)

Hardware IDCT

In the hardware IDCT implementation, all IDCT operations are implemented in hardware using the DSP48 slices. Xilinx System Generator for DSP[8] tool was used to implement the design. The tool generates a Verilog (or VHDL) file that was imported into the apu_idct XPS project.

Figure 9 shows the 8 x 8 IDCT macro block that contains the 1D IDCT and 2D IDCT modules. (See "Running the Demo Video Application on the ML403 Board," page 24.)

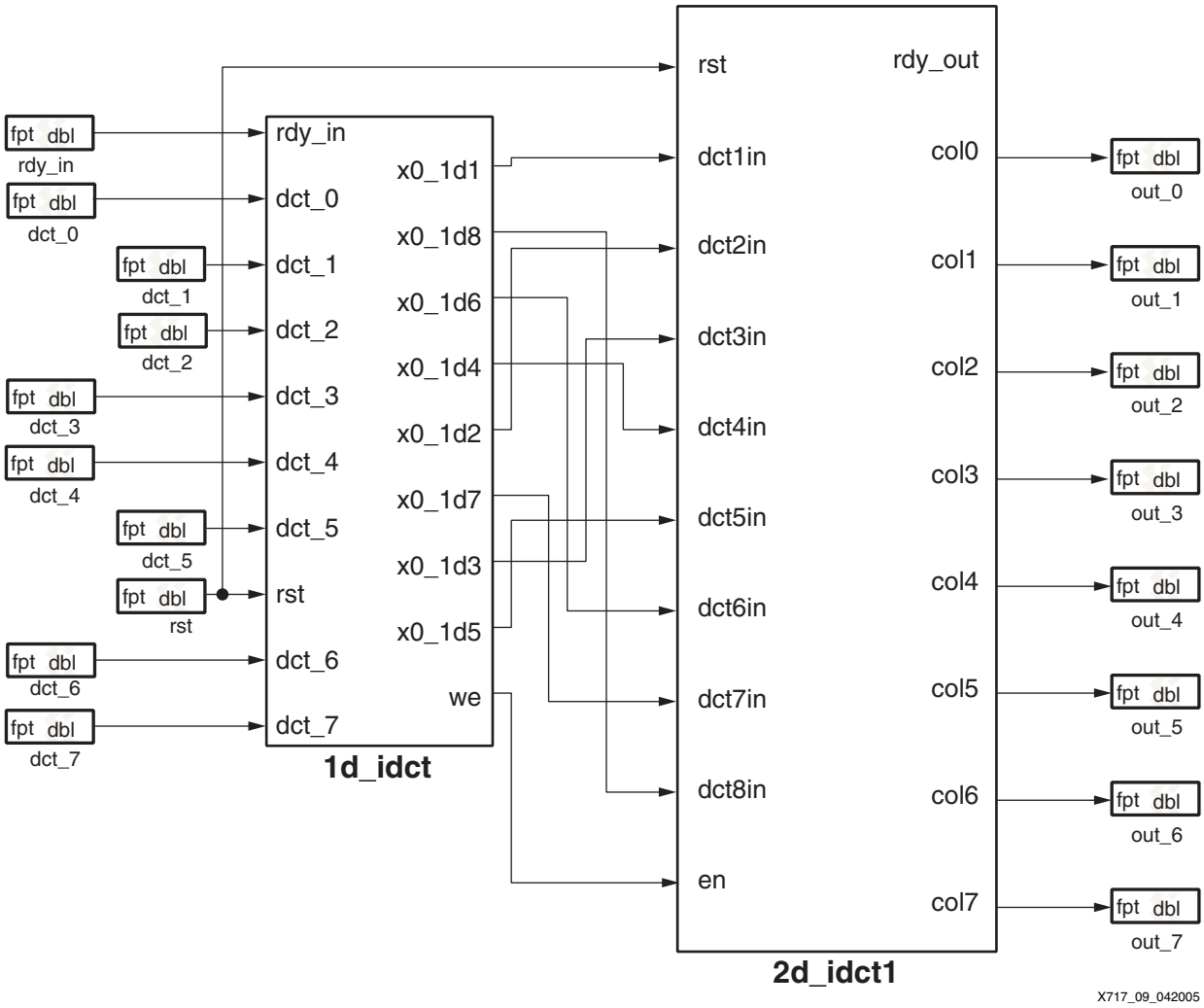

Figure 9: **IDCT Module from System Generator**

One row of input values enters the IDCT module from the FCM. A subsequent row of input data can arrive at a minimum of four clock cycles after the previous row of values. Coefficient values for computation are stored in internal look-up table (LUT) RAMs, and the IDCT module calculates the output value in two steps.

In the first step, the 1D IDCT is calculated with eight DSP48 slices. Figure 10, page 17 shows the detail of one DSP48 slice. Each DSP48 slice multiplies one input data value with four coefficient values, and the resulting four products are added together. The DSP48s work in a multiply-accumulate mode for four clock cycles. At the end of the fourth clock cycle, the outputs from two DSP48s are both added and subtracted to produce two of the 1D IDCT output values. Together, the eight DSP48s produce one column of eight 1D IDCT values. The overall latency is 13 clock cycles for the 1D IDCT.

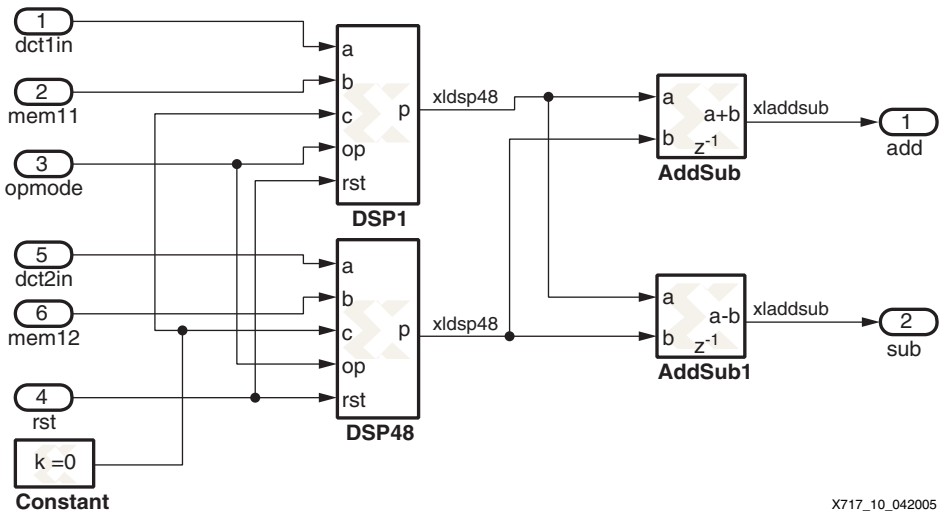

Figure 10: **Detail of DSP48 Slice in 1D IDCT**

In the second step, the output from the 1D IDCT enters the 2D IDCT block. The column-wise output from the 1D IDCT is fed into 16 DSP48s, where two DSP48 slices receive the same 1D value. These values are multiplied with 2D coefficients, and the DSP48s are again configured in a multiply-accumulate mode. Here, two DSP48 slices produce two output values: the sum of the two outputs and the difference of the two outputs. The 16 DSP48s produce 16 values at a time, corresponding to two rows of 2D IDCT values. Eight of these values are delayed so that only one row of outputs is valid at a given clock. The total latency for the 2D IDCT is 47 clocks. This latency includes the delay in reading the eight rows of input data at one row per clock, with subsequent rows applied every four clocks.

Verification of IDCT Module

Automated test programs verified that the software and hardware IDCT modules functioned correctly. To make sure that the IDCT modules received valid input data, a software DCT program produced the IDCT inputs, which consisted of random 8-bit RGB values ranging between –512 and 511, and sent these values to the IDCT modules. After DCT and IDCT operations, the input of the DCT matched the output of the IDCT, proving that the output of the IDCT modules in the APU-Enhanced IDCT system was correct.

Data Flow in the IDCT System

The APU-enhanced IDCT system uses pre-defined load/store instructions. In addition, the APU-to-FCM interface uses an FCB instead of a direct connection. This is done to illustrate how the designer can make use of the FCB interface provided by XPS.

In this system, the flow of data is as follows:

1. An IDCT operation begins with the processor forwarding an FCM *load quadruple-word* instruction for IDCT input data to the APU.
2. The APU passes the instruction to the FCM, which decodes the FCM load and waits for data from memory to arrive via the APU.

   Figure 4, page 7 shows the timing relationship between signals in the APU-to-FCM interface for a multiple-word FCM load. As each word is received from the APU, the FCM writes it to a 4 x 32-bit register file. Eventually, 16 bytes (eight pixels, two bytes per pixel) of IDCT input data are stored in this register file.
3. The FCM sends the IDCT input data to the IDCT module.
4. After eight FCM load quadruple-word instructions, the processor forwards an FCM *store double-word* instruction to the APU in anticipation of the IDCT output.

5. The FCM decodes the FCM *store* instruction and waits for data from the IDCT module.
6. After processing, the IDCT module returns IDCT results to the FCM.

   The IDCT module returns eight bytes (eight pixels, one byte per pixel) in a single clock cycle, and the FCM writes this value into a 16 x 32-bit register file. Then the FCM returns the IDCT results to the APU in a series of FCM store double-word operations.
7. Finally, the FCM returns the IDCT output data to the processor via the APU. This data is written back to memory.

   Figure 5, page 7 shows the timing relationship between signals in the APU-to-FCM interface for a multiple-word FCM store.

IDCT Demo State Machines

In the FCM, two state machines manage data transfers. Figure 11 shows the main state machine that is responsible for FCM load and FCM store operations. This state machine communicates with the processor using the APU.

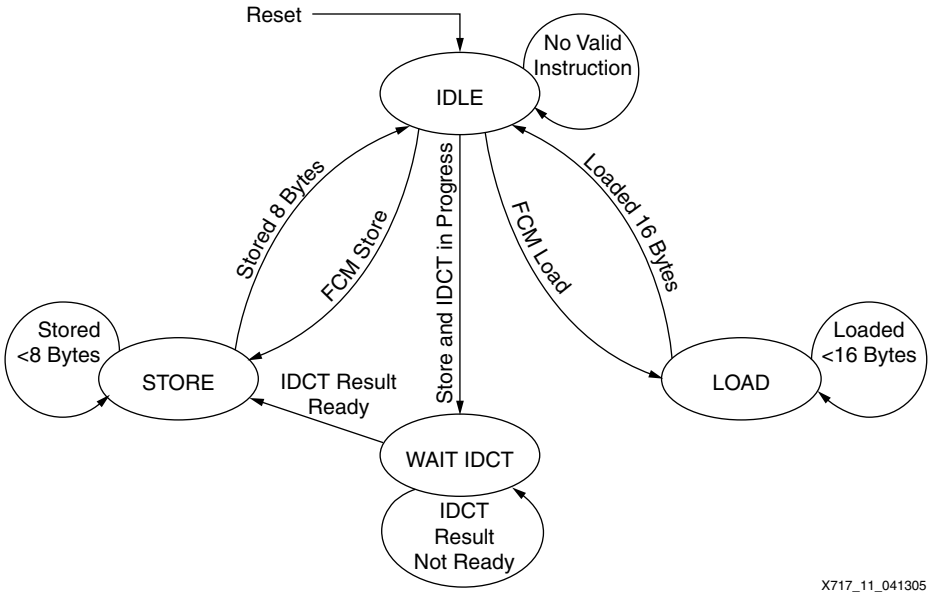

Figure 11: **Main State Machine**

Figure 12 shows the IDCT state machine that communicates with the IDCT module.

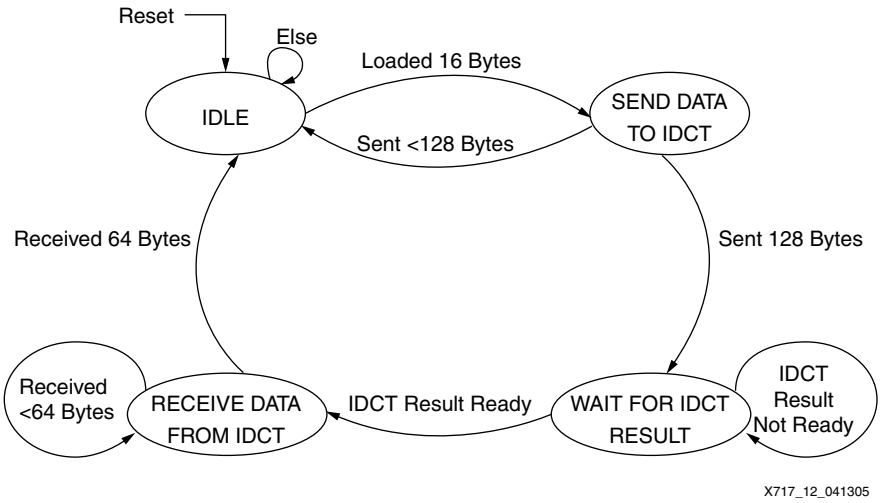

Figure 12: **IDCT State Machine**

Video Application

The demo video application displays a series of image frames to demonstrate an APU-enhanced IDCT application. This program is similar to a video player in that image frames are refreshed at a constant rate to give an illusion of motion.

For a video player, there are generally three steps between receiving video data (for instance, an MPEG movie file) to displaying moving pictures on a monitor:

1. Convert data into frames and have these frames undergo a Discrete Cosine Transform (DCT) compression.
2. Run the DCT output through IDCT decompression. (This is where the APU-enhanced IDCT module can be utilized.)
3. Send the IDCT output to a display controller and on to a monitor.

To illustrate the difference between IDCT in software and IDCT in hardware, the demo video application alternates between the software and the hardware IDCT modules (Step 2 above) to display image data from memory to a VGA monitor.

Initially, the image frames are rendered using the software IDCT module, and a rotating image is displayed. After five seconds, the video application automatically switches over to using the hardware IDCT module, and the rotation is noticeably faster. Every five seconds, the IDCT module switches between software and hardware implementations, and a vertical bar on the right-hand side of the screen updates to show the relative performance difference. Users also have the option of pressing any one of the pushbuttons on the ML403 board to initiate a switch.

Figure 13 shows the system-level block diagram.

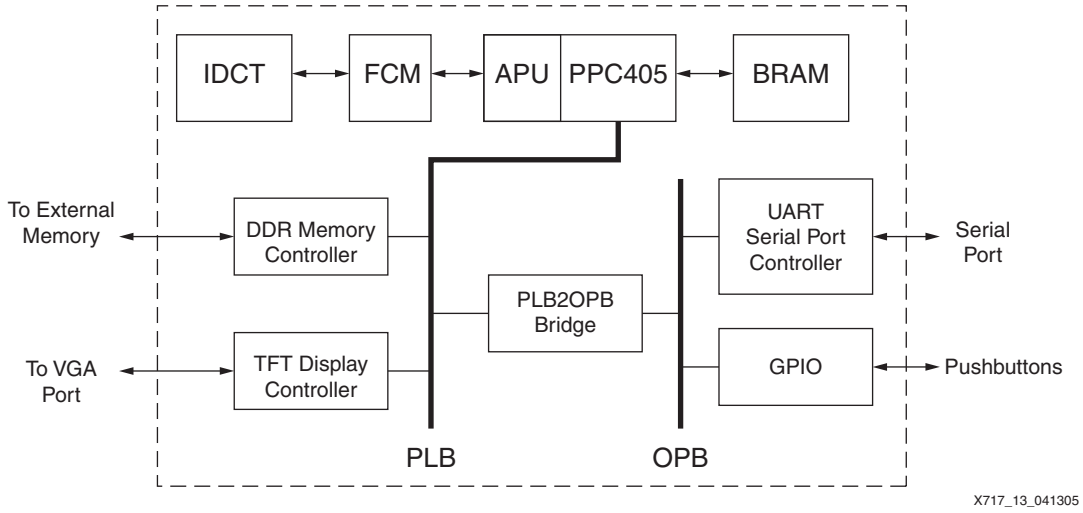

Figure 13: **Demo Video Application System-Level Block Diagram**

First, image data is loaded into DDR memory. Then DCT and IDCT operations are performed on the image data and the output is stored in memory locations reserved for TFT display buffers. Finally, the TFT display controller reads output image data from DDR memory and sends it through the VGA port for display on a monitor.

XPS: Close-up of the IDCT Demo MHS File

Like the "FCM Register Load/Store Example," page 6, the FCM is implemented as a user pcore in the XPS project. One difference in the video application is that an FCB interface sits between the APU and the FCM. Excerpts from system.mhs in Table 8 show how an FCB is instantiated in the design, as well as how declarations in the processor and FCM modules change to accommodate the FCB.

Table 8: **Excerpts from hw/apu_idct/system.mhs**

```
BEGIN ppc405_virtex4
  ...
  PARAMETER C_APU_CONTROL = 0x0001
  BUS_INTERFACE MFCB = fcb_v10_0
  ...
END

# FCB EDK pcore
BEGIN fcb_v10
  PARAMETER INSTANCE = fcb_v10_0
  PARAMETER HW_VER = 1.00.a
  PORT FCB_CLK = plb_clk
  PORT SYS_RST = bus_rst
END
```

```
# custom FCM IDCT module
BEGIN fcm
  PARAMETER INSTANCE = fcm_0
  PARAMETER HW_VER = 1.00.a
  BUS_INTERFACE SFCB = fcb_v10_0
  PORT clock = plb_clk
  PORT reset = bus_rst
END
```

XPS: The FCM as a Pcore in the IDCT Demo

Compared to the FCM defined in the "FCM Register Load/Store Example," page 6, the FCM IDCT demo's pcore contains state machine and decoder logic as well as an IDCT module. In the microprocessor peripheral description (MPD) file, the IDCT demo's FCM pcore contains port declarations that are not present in the MPD file of the FCM register load/store example's pcore.

Table 9 highlights the differences in the MPD files of an FCM pcore with a directly-connected APU-to-FCM, and an FCM pcore with an FCB interface.

Table 9: **Comparison of MPD Files**

Without FCB[1]	With FCB[2]
BEGIN fcm_loadstore ... PORT FCMAPUDONE = "", DIR = OUT PORT APUFCMFLUSH = "", DIR = IN ... END	BEGIN fcm ... BUS_INTERFACE BUS = SFCB, BUS_STD = FCB, BUS_TYPE = SLAVE ... PORT FCMAPUDONE = Sl_DONE, DIR = OUT, BUS = SFCB PORT APUFCMFLUSH = FCB_FLUSH, DIR = IN, BUS = SFCB ... END

Notes:

1. apu_loadstore/pcores/fcm_loadstore_v1_00_a/data/fcm_loadstore_v2_1_0.mpd
2. apu_idct/pcores/fcm_v1_00_a/data/fcm_v2_1_0.mpd

C Source Code for the IDCT Demo

Table 10 shows excerpts from the C source code for the IDCT portion of the demo video application.

Table 10: **IDCT C Code**

```
void mpeg2_idct_c_hw (int16_t *src, uint8_t *dst)
{
  // load idct input data from memory
  lqfcmx(0, block, 0*16);
  lqfcmx(0, block, 1*16);
  lqfcmx(0, block, 2*16);
  lqfcmx(0, block, 3*16);
  lqfcmx(0, block, 4*16);
  lqfcmx(0, block, 5*16);
  lqfcmx(0, block, 6*16);
  lqfcmx(0, block, 7*16);

  // store result into memory
  stdfcmx(0, resultblock, 0*32);
  stdfcmx(2, resultblock, 1*32);
  stdfcmx(4, resultblock, 2*32);
  stdfcmx(6, resultblock, 3*32);
  stdfcmx(8, resultblock, 4*32);
  stdfcmx(10, resultblock, 5*32);
  stdfcmx(12, resultblock, 6*32);
  stdfcmx(14, resultblock, 7*32);
}
```

In total, there are eight FCM loads and eight FCM stores. After eight loads, the IDCT module computes the values and the FCM waits to execute eight stores. When the IDCT outputs are ready, the FCM relays them to the APU, which in turn passes them to the processor and finally into memory.

Setting Up the Demo Video Application

1. Follow the instructions in "Setting Up the Tools/Hardware," page 11 to prepare the necessary equipment and the simulation environment.
2. Connect a VGA monitor to the VGA port of the ML403 board, then power on the monitor.
3. Browse to the xapp717/hw/apu_idct directory.
4. Start Xilinx Platform Studio (XPS), and open the system project:

   File → **Open** → **system.xmp**
5. Generate the software executables.

   Tools → **Build All User Applications**

   Software executable for simulation:
   xapp717/hw/apu_idct/ppc405_0/code/apu_idct_sim.elf

   Software executable for hardware:
   xapp717/hw/apu_idct/ppc405_0/code/apu_idct.elf
6. Ensure that the correct software application is used for simulation.
   a. Click the Applications tab.
   b. Right-click on the software project, apu_idct_sim, and select **Mark to Initialize BRAMs**.

      Verify that all other software projects do not have the **Mark to Initialize BRAMs** option checked (or highlighted in green).
7. Generate simulation files by choosing **Tools** → **Generate Simulation HDL files**.
8. Generate the FPGA bitstream.
   a. Ensure that the correct software application is used for the bitstream.
      - Click the Applications tab.
      - Right-click on the software project, ppc405_0_bootloop, and select **Mark to Initialize BRAMs**.
      - Verify that all other software projects do not have the **Mark to Initialize BRAMs** option checked (or highlighted in green).
   b. **Tools** → **Update Bitstream**.

      This process takes approximately 20 minutes on a 3.6 GHz machine.

Running the APU IDCT Simulation

The apu_idct project contains a simulation example in which an IDCT is performed on a single 8 x 8 block of data. Besides verifying the FCM load/store operations, the simulation waveforms also show the timing relationships between the FCM state machines and the IDCT module. The simulation example allows the designer to test the IDCT module as well as the FCM's internal logic.

Table 11 shows the sample data set used in the IDCT simulation.

Table 11: **IDCT Simulation Data**

IDCT Input Block							
214	34	–6	8	–12	5	2	–1
49	–25	–4	–10	5	9	–2	1
–3	11	8	4	–1	–8	3	0
20	13	–9	4	–2	3	–1	2
–10	5	3	–15	–15	4	1	3
–1	–3	–3	10	9	–7	3	–2
1	15	5	6	–5	–14	–3	–4
–6	–6	10	6	–1	2	–4	–2

IDCT Output Block							
40	42	43	35	33	36	33	26
33	26	40	43	33	33	40	33
33	32	19	26	29	33	29	29
21	22	36	33	26	27	26	22
26	29	36	29	16	22	33	13
40	43	33	30	33	29	12	4
35	22	16	19	26	25	12	8
26	36	26	4	3	12	3	1

1. Launch ModelSim 5.8e.
2. At the prompt, enter the following to compile the simulation:

```
cd xapp717/hw/apu_idct
cd simulation/behavioral
do ../../data/testbench.do
```

3. When the waveform viewer pops up, enter **run -all** to run the simulation.
4. In the waveform viewer, observe the signals in the following sections:
   - IDCT Interface
   - APU-to-FCM to IDCT Interface
   - Processor Registers

   The signals match up to those in the timing diagrams in Figure 4, page 7 and Figure 5, page 7.

Running the Demo Video Application on the ML403 Board

The demo video application plays 30 image frames onto the display in a continuous loop. It can be loaded onto the ML403 board using either the XMD tool or the System ACE™ CompactFlash. Because of the volume of image data that must be loaded into memory, it takes about one hour to initialize the contents of memory through XMD. This initialization is reduced to a fraction of the time using the System ACE CompactFlash.

Loading the Demo Using XMD

1. Program the FPGA on the ML403 board with the generated bitstream.

   Use iMPACT to program the FPGA by way of the JTAG connections on the ML403 board.

2. Connect to the processor core using XMD.
   a. In XPS, select **Tools** → **XMD** to start an XMD session.
   b. Connect to the PPC405:

      ```
      XMD% connect ppc hw
      ```

   c. Reset the system:

      ```
      XMD% rst
      ```

   d. Load sample image data into memory:

      ```
      XMD% dow -data xdct/apd_logo/apd.xdct.bin 0x0
      ```

   Note: This process takes about one hour using the PC4 cable. See "Loading the Demo Video Application Using System ACE CompactFlash" for an alternative method.

3. Download and run the apu_idct.elf software executable:

   ```
   XMD% dow ppc405_0/code/apu_idct.elf
   XMD% run
   ```

Loading the Demo Video Application Using System ACE CompactFlash

1. Create an ACE file containing the FPGA bitstream, the image data, and the software executable.
   a. Copy apd.xdct.bin to xapp717/hw/apu_idct/xdct/apd_logo/
   b. **Tools** → **Xilinx Command Shell** and enter the following command:

      ```
      xapp717/hw/apu_idct % xmd -tcl genace.tcl -jprog -board ml403 -hw
      implementation/download.bit -data xdct/apd_logo/apd.xdct.bin 0x0 -elf
      ppc405_0/code/apu_idct.elf -ace apu.ace
      ```

   Note: This process takes about 10 minutes on a 3.6 GHz machine.

   c. Copy the resulting apu.ace file into a suitable CompactFlash card and load the contents into an ML403 board.

      After apu.ace is completely loaded, the demo video application starts.

Using the Pre-Built Demo Video Application

A pre-built demo video application is available from:
http://www.xilinx.com/products/boards/ml403/reference_designs.htm.

To use the pre-built demo, instead of building the bitstream and software executable, extract the contents of the apu_idct.img.zip file and re-image a 512 MB CompactFlash card with the extracted apu_idct.img file.

The tools and instructions for re-imaging a CompactFlash card can be downloaded from:
http://www.xilinx.com/products/boards/ml310/current/utilities/cf_image_tools.zip
http://www.xilinx.com/products/boards/ml310/current/utilities/cf_reimage.pdf

Conclusion

Before the existence of the APU, the typical approach to offloading computations to the FPGA was through a processor bus peripheral, for example, a processor local bus (PLB) floating point unit. Two disadvantages of a bus peripheral are the costs in execution time and logic incurred by the need for bus arbitration. However, the APU is not the best solution for every system. For an application whose execution time is mainly taken up by computation latency instead of data transfer latency, a bus peripheral is likely to have similar performance to an APU-enhanced solution.

This application note introduces the APU as a flexible and high-bandwidth coprocessor interface in the Xilinx Virtex-4 FX family of FPGAs. Along with the FCM and its interfaces to the APU, the APU can be used to offload computations into the FPGA, accelerating system performance. Video applications can make use of the new embedded DSP48 blocks to implement DSP algorithms. An example of how XPS is used to create, debug, and implement an APU-enhanced system is described to help the designer get started with using the APU.

Finally, an APU-enhanced video application that utilizes DSP48 blocks is demonstrated to have better performance than a software solution. This performance is limited only by memory bandwidth in the PLB, and can be improved by running the FCM at a higher clock frequency. While a hardware bus peripheral for IDCT can be used instead of the APU, the bus peripheral is constrained by the interface, bandwidth, and bus arbitration delays of the bus architecture.

References

1. ISE documentation
   http://www.xilinx.com/support/sw_manuals/xilinx7/index.htm
2. EDK documentation
   http://www.xilinx.com/ise/embedded/edk_docs.htm
3. ML403 embedded platform website
   http://www.xilinx.com/ml403
4. Xilinx, Inc., UG018: PowerPC™ 405 Processor Block Reference Guide
   http://www.xilinx.com/bvdocs/userguides/ug018.pdf
5. Xilinx, Inc., UG082: ML40x Reference Design User Guide
   http://www.xilinx.com/bvdocs/userguides/ug082.pdf
6. Free Software Foundation: GNU GCC 3.4.3 Manual, Section 5.35
   http://gcc.gnu.org/onlinedocs/gcc-3.4.3/gcc/Extended-Asm.html#Extended-Asm
7. Xilinx, Inc., UG111: Embedded System Tools Reference Manual (EDK 7.1)
   http://www.xilinx.com/ise/embedded/est_rm.pdf
8. Xilinx, Inc., UG073: XtremeDSP Design Considerations User Guide
   http://www.xilinx.com/bvdocs/userguides/ug073.pdf

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
04/29/05	1.0	Initial Xilinx release.
09/20/05	1.1	Updated Required Hardware/Tools section.
09/29/05	1.1.1	Removed unnecessary figure.