FPGA Peripheral Expansion & FPGA Co-Processing

with a TI TMS320C6000

July 2004, ver 1.0 Application Note 352

Introduction

This application note describes how peripherals and co-processors can be added to Texas Instrument's (TI's) TMS320C6000 family of digital signal processing (DSP) devices. The hardware interface is a connection between TI's external memory interface (EMIF) and a first-in first-out (FIFO) buffer. In the Altera® Ateme EMIF reference design, a fast Fourier transform (FFT) co-processor and an LED peripheral provide examples of how a system can be developed using Altera development tools and architecture elements. The architecture provides flexibility to allow many types of peripherals and co-processors. The concept is demonstrated on Ateme's DMDK642 development board, which features a TI DM642 digital media processor and an Altera EP1C20 Cyclone™ FPGA.

For more information on the DMDK642 development board, refer to the Ateme web site, **www.ateme.com**.

Altera supplies the reference design and example system design as Verilog HDL source code. Altera also supplies example software to demonstrate its use.

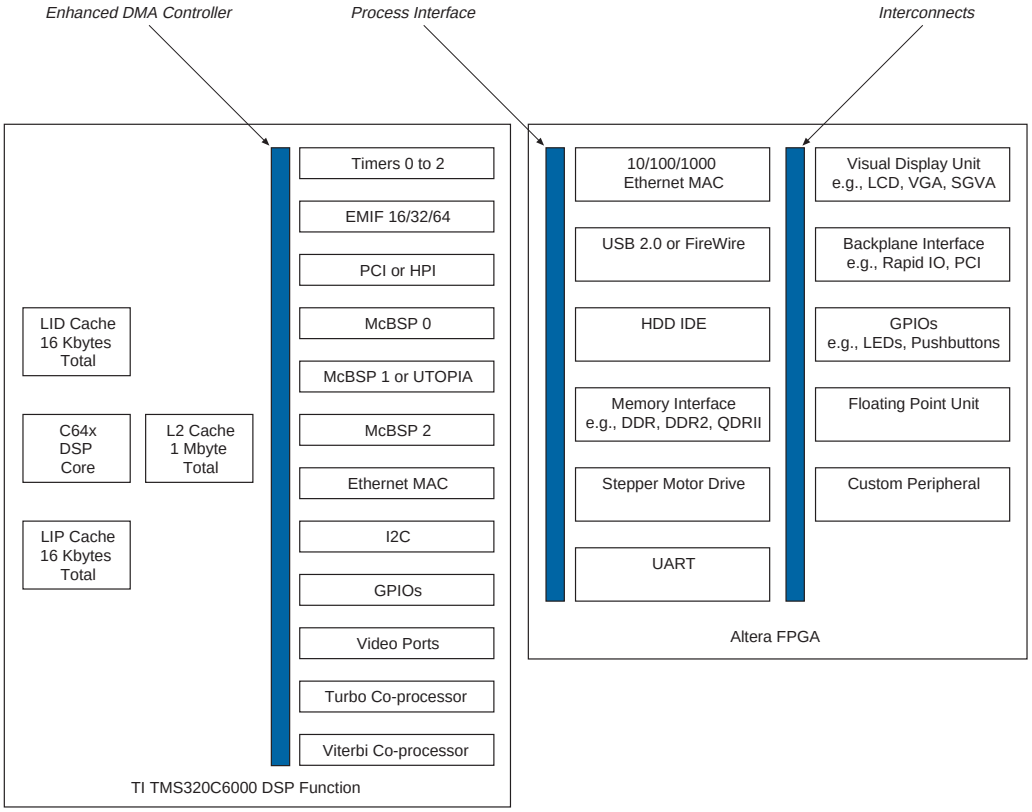
Background

The following sections discuss FPGA peripheral expansion and co-processing.

FPGA Peripheral Expansion

Although off-the-shelf processors like TI's C6000 family of digital signal processors offer a rich peripheral set, there are often additional peripherals that are required in a system. Figure 1 shows the peripherals currently offered in the TI TMS320C6000 family and other peripherals that may also be required in a system. Altera's SOPC Builder facilitates the integration and use of peripherals. Currently, there are over 50 components from Altera and its third parties that are SOPC Builder ready, and the list continues to grow.

Figure 1. Peripherals



SOPC Builder provides Altera customers with a new and easy-to-learn design method that significantly reduces menial "wire-connecting" tasks, while also enabling new levels of system integration. A once time-consuming and tedious task becomes push-button easy.

SOPC Builder is a hardware-generation engine, and embedded software designers require a complete software development environment to match this custom-built hardware. SOPC Builder automatically produces this software. SOPC Builder creates corresponding software components

to jump-start software development and provide a complete design environment when a system is generated. The software development environment can include some or all of the following features :

- Header files
- Generic peripheral drivers
- Custom software libraries
- OS/real-time operating system (RTOS) kernels

Besides convenience, this development environment also promotes better design coherency between hardware and software engineers. With SOPC Builder, changes to the hardware are immediately reflected in the software development environment. Software engineers can consequently design without any fear of hardware changes. As long as the software team has access to the latest header files, libraries, and drivers, both teams work to the same goal and the development cycle can continue smoothly.

SOPC Builder provides an immediate simulation of hardware and software environment. SOPC Builder generates all ModelSim® project files, including formatted bus-interface waveforms and a complete simulation testbench. Compiled software code is automatically placed into memory models and compiled with the rest of the project files.

FPGA Co-processor

Across a wide spectrum of applications, signal processing algorithm complexity is exceeding the processing capabilities of standalone digital signal processors. In some of these applications, software developers use hardware co-processors to off-load a variety of algorithms including Viterbi decoding, turbo encoding/decoding, butterfly processing, discrete cosine transforms (DCT), and 1D and 2D filters. In a few cases, digital signal processors include on-chip hardware co-processors where the end application supports the expense of designing such a market-specific solution. In third-generation wireless systems, the addition of the turbo forward-error-correction algorithm had a huge impact on the amount of processing required per user data channel in a channel element card. TI has successfully conquered this challenge of using coprocessors for turbo and Viterbi processing in existing third-generation basestations. Unfortunately, the high cost of implementation makes the availability of digital signal processors with end-market specific co-processors unattainable. In applications where no co-processors are available, design tools and methodologies enable companies to develop their own co-processors using the latest FPGAs that easily interface with a wide range of digital signal processors and general purpose processors (GPP), and provide increased system performance and lower system costs.

The following actions describe a typical flow to develop the co-processor:

- Profile applications to identify high-load software algorithms suitable for off-loading to coprocessors
- Parameterize off-the-shelf co-processors if available. See list of IP optimized for Altera FPGAs. If no suitable co-processor is available, develop custom coprocessor block using DSP Builder
- Consider viable-coprocessor system architectures
- Select processor interface
- Integrate hardware and software using SOPC Builder
- Verify in hardware

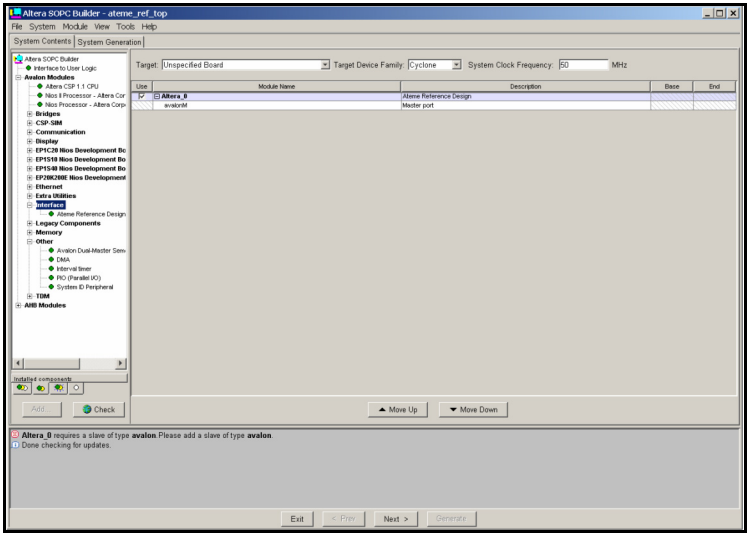
Ateme EMIF Reference Design

The Ateme EMIF reference design connects the synchronous FIFO buffer to two Atlantic™ interfaces (for transmitting data and receiving data, respectively). The asynchronous EMIF is used for control and status, and is also connected to an Avalon™ bus. In the reference design example system, the Avalon bus is connected via a parallel I/O (PIO) module to status LEDs present on the DMDK642.

For more information on the Atlantic interface, refer to *FS13: The Atlantic Interface Functional Specification*. For more information on the Avalon bus, refer to the *Avalon Bus Specification Reference Manual*.

Figure 2 shows SOPC builder with the Altera Ateme EMIF Reference Design listed under Interfaces.

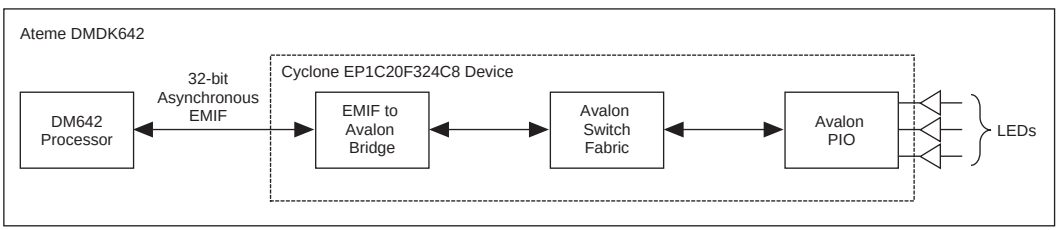
Figure 2. Ateme EMIF Reference Design in SOPC Builder



Peripheral Expansion Functional Description

Figure 3 shows a block diagram of the EMIF peripheral expansion example in the Altera Ateme EMIF Reference Design.

Figure 3. Peripheral Expansion Block Diagram



EMIF

The EMIF provides a glueless interface to a variety of external memory components including SDRAM, SRAM, and FIFO buffers. In the Altera Ateme EMIF Reference Design the 64-bit EMIF is used in 32-bit

asynchronous mode through the chip select two memory space. An EMIF-to-Avalon bridge module allows the connection of Avalon slave peripherals in Altera's SOPC builder.

Avalon Switch Fabric

The Avalon bus is a simple bus architecture designed for connecting on-chip processors and peripherals together into a system–on–a–programmable chip (SOPC). The Avalon bus is an interface that specifies the port connections between master and slave components, and specifies the timing by which these components communicate.

Basic Avalon bus transactions transfer a single byte, half-word, or word (8, 16, or 32 bits) between a master and slave peripheral. After a transfer completes, the bus is immediately available on the next clock for another transaction, either between the same master-slave pair, or between unrelated masters and slaves. The Avalon bus also supports advanced features, such as latency-aware peripherals, streaming peripherals and multiple bus masters. These advanced transfer modes allow multiple units of data to be transferred between peripherals during a single bus transaction.

The Avalon bus supports multiple bus masters. This multi-master architecture provides great flexibility in the construction of SOPC systems, and is amenable to high bandwidth peripherals. For example, a master peripheral may perform direct memory access (DMA) transfers, without requiring a processor in the data path to transfer data from the peripheral to memory.

For more information on the Avalon bus, refer to the *Avalon Bus Specification Reference Manual*.

Avalon PIO Module

To choose the PIO module, expand **Other**, choose **PIO (Parallel I/O)** and click **Add** (see Figure 2). You can now configure the PIO module (see Figure 4).

Figure 4. Choose the PIO Module

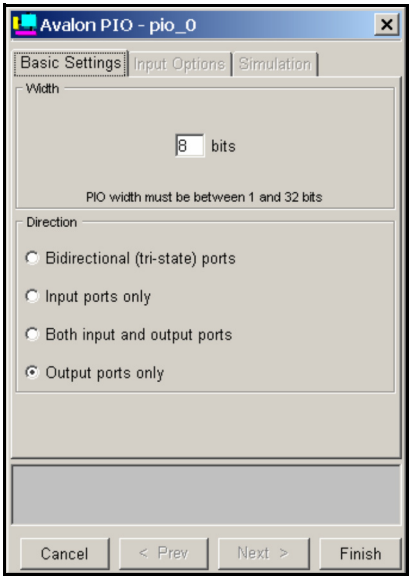
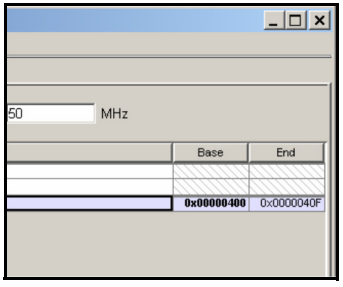


Figure 4 shows the reference design configuration, which has a PIO module a base address of `0x400` (see Figure 5).

Figure 5. PIO Base Address



LEDs

In the Altera Ateme EMIF reference design, the peripheral expansion capability is demonstrated by connecting an Avalon parallel I/O (PIO) module to drive the three user LEDs connected to the Cyclone EP1C20F324C8 device on the Ateme DMDK board.

TI Software Description

The TI software required to control the LEDs is simple, because the PIO appears directly in the DSP memory map. Various techniques are possible: one technique is illustrated using the C programming language as supported by TI's Code Composer Studio tool.

The EMIF base address of the FPGA in chip select two memory space is defined, e.g.,

```
#define FPGA_BASE 0xA0080000
```

The PIO module was assigned to a base address of `0x400`, so its EMIF base is defined as:

```
volatile unsigned int * led_port = ((volatile unsigned
int *)(FPGA_BASE + 0x400));
```

The LEDs are then controlled by simple assignments. The LEDs on the Ateme DMDK board are connected to bits 0 to 2 of the PIO module in the order red, green, and yellow. The LEDs illuminate when a `0` is written to the appropriate bit. For example, if the PIO is initially outputting `0`:

```
*led_port = 1;// Extinguish red LED
```

Or another example:

```
*led_port = 3;// Extinguish red & green LED
```

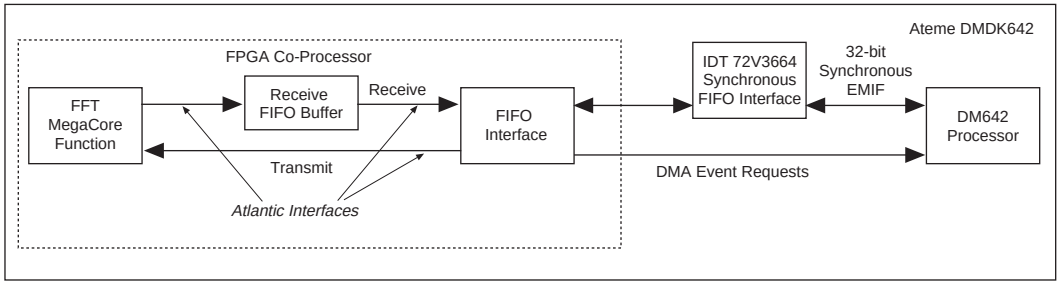
Summary

You can apply this concept to the other peripherals listed in the SOPC Builder peripheral table or you can create your own peripheral.

FPGA Co-Processor Functional Description

Figure 6 shows the Ateme EMIF reference design FPGA co-processor block diagram. The Atlantic interfaces allow you to insert one or more co-processor modules between the Atlantic slave source and the Atlantic slave sink.

Figure 6. Block Diagram



The DMA controller within the DM642 processor transmits packets of data to be processed via a synchronous EMIF to the DMDK642s on-board bi-directional synchronous FIFO buffer. The Ateme EMIF reference design monitors the fill level of the transmit FIFO buffer to determine when new data is available for processing, reads the data and passes it on to the co-processor function via the Atlantic transmit converter.

The output of the FFT MegaCore function is buffered by the receive FIFO buffer to allow for periods when the synchronous FIFO interface is busy. Data is transferred from the receive FIFO buffer to the synchronous FIFO buffer. A DMA transfer is requested when a whole packet of processed data is available to be read from the synchronous FIFO buffer by the processor

Table 1 shows how packets of data are scheduled through the hardware blocks of the Ateme EMIF reference design.

Table 1. Scheduling

Action	Step							
	1	2	3	4	5	6	7	8
EMIF	Write 0	–	Write 1	–	Write 2	Read 0	Write 3	Read 1
FIFO interface	–	In 0	–	In 1	Out 0	In 2	Out 1	In 3
FFT MegaCore function	–	–	FFT 0	–	FFT 1	–	FFT 2	–
Output FIFO buffer	–	–	–	Out 0	–	Out 1	–	Out 2

Looking at packet 0, it is first written over the EMIF, by the enhanced DMA (EDMA), into the bi-directional FIFO buffer from where it is read by the FIFO buffer interface (step 2) into the FFT MegaCore function. The FFT MegaCore function processes the packet (step 3) and writes it into the

receive FIFO buffer (step 4), from where it is read by the FIFO interface and written into the bi-directional FIFO buffer (step 5). Finally (step 6), the processed packet is read out of the bi-directional FIFO buffer by the EDMA. At this point the EDMA is now interleaving read and write packets from and to the bi-directional FIFO buffer.

The programmable thresholds in the bi-directional FIFO buffer are set to a value just less than the length of one packet, e.g., for a 1,024-point 16-bit FFT the value is 1,022, because each packet requires 1,024 32-bit words

In the transmit direction the bi-directional FIFO buffer almost full flag asserts just as the EDMA completes writing a packet. This action causes the FIFO interface to read the packet into the FFT MegaCore function at the next opportunity.

In the receive direction, the FIFO interface writes a packet into the bi-directional FIFO buffer. The FIFO buffer almost full flag signals a DMA event to the EDMA just as the complete packet is written.

FFT MegaCore Function

The FFT MegaCore function, version 2.0.0 is a high performance, highly parameterizable fast Fourier transform (FFT) processor, optimized for the Altera Stratix® II, Stratix GX, Stratix, and Cyclone™ device families. The FFT MegaCore function implements a complex FFT or inverse FFT (IFFT) for high-performance applications. The FFT MegaCore function has the following features:

- Support for Windows, Solaris, and Linux operating systems
- High throughput quad-output radix 4 FFT engine
- Support for multiple single-output and quad-output engines in parallel
- Multiple I/O data flow modes: streaming, buffered burst, and burst
- Atlantic compliant input and output interfaces
- Parameterization-specific VHDL and Verilog HDL testbench generation
- Transform direction (FFT/IFFT) specifiable on a per-block basis
- Easy-to-use IP Toolbench interface
- IP functional simulation models for use in Altera-supported VHDL and Verilog HDL simulators
- Radix-4 and mixed radix-4/2 implementations
- Block floating-point architecture to maintain the maximum dynamic range of data during processing
- Uses embedded memory
- Maximum system clock frequency >300 MHz

- Optimized to use Stratix II, Stratix GX, and Stratix DSP blocks and the TriMatrix™ memory architecture
- Support for OpenCore® Plus evaluation

Processor Interface

EMIF is chosen as the connection medium for the FPGA co-processor due to the data transfer rates available and the possibility of using the enhanced DMA (EDMA) controller integrated within the DM642 processor. Table 2 shows the peak data transfer rates based on the given clock rates. In practice, lower overall rates will be achieved, e.g., when transferring data between two resources that share the EMIF.

Table 2. Peak Data Rates

EMIF Mode	Peak Transfer (MBps)		
	@ 66 MHz	@ 100 MHz	@ 133 MHz
32-bit Asynchronous *(1)*	53	80	106
32-bit Synchronous	264	400	532
64-bit Synchronous	528	800	1,064

Note to Table 2:
(1) Assume each asynchronous access is 5 EMIF clocks.

In a generic design, the FPGA co-processor may be implemented as a memory mapped device using either a synchronous or asynchronous EMIF connection (determined by performance requirements). FIFO buffers may be used to allow EMIF burst accesses to proceed without wait states, independently of the rate at which the co-processor consumes or produces data.

On the ATEME DMDK board, the Altera Ateme EMIF Reference Design is connected via a 32 bit bi-directional synchronous FIFO buffer. This appears in the DSP memory map in chip select space 3. Transmit packets (digital signal processor to FPGA, or software to hardware) are written to the FIFO buffer and receive packets (FPGA to digital signal processor, or hardware to software) are read from the FIFO buffer. FIFO status signals are available to the DSP function (for the digital signal processor side of the FIFO buffer) and to the FPGA (for the FPGA side of the FIFO buffer).

Atlantic Interface

The Atlantic interface has the following features:

- Flexible interface for packet-oriented data of arbitrary length
- Interfaces all Altera cell and packet MegaCore functions
- Synchronous point-to-point connection
- High throughput with flexible flow control
- Master source/slave sink or master sink/slave source relationships
- Scalable clock frequency and data path width
- Fixed start of packet (SOP) alignment simplifies packet handling

For more information on the Atlantic interface, refer to *FS13: The Atlantic Interface Functional Specification*.

Benefits of using TI's EDMA

Using the EDMA to move data between the digital signal processor and the co-processor frees up the digital signal processor to perform other tasks while waiting for the co-processor to return data. The EDMA transfers are triggered by requests from the FPGA co-processor, either when it is ready for new data to be processed (transmit direction) or has processed data available (receive direction. An interrupt is requested to the DSP core after each complete EDMA transfer. The EDMA gives considerable flexibility in the way the data is transferred. In the Altera Ateme EMIF Reference Design the whole of an FFT block is transferred at once. The example case of a 1,024 point 16-bit FFT equates to a block size of 4,096 bytes (each complex sample being two bytes real and two bytes imaginary data).

Hardware Integration

Figure 7 shows the EMIF reference design pinouts.

Figure 7. Pinouts

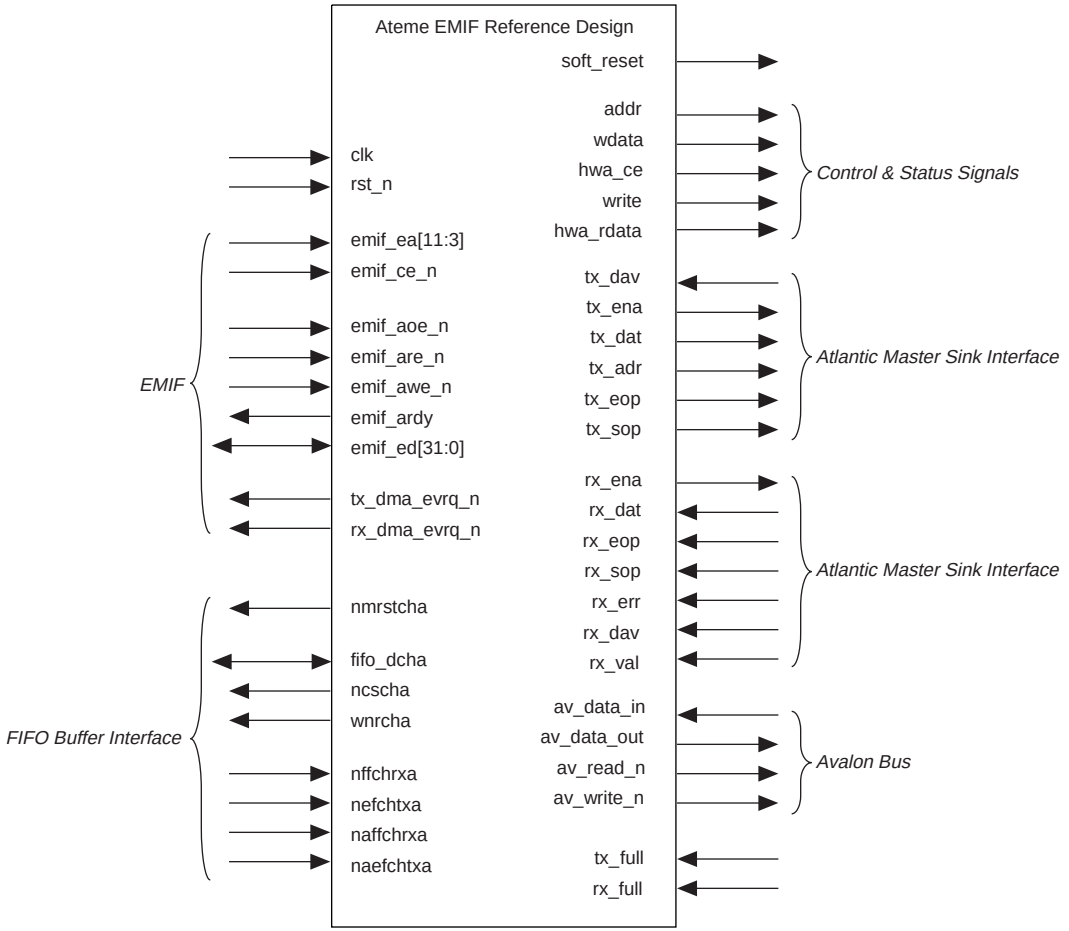


Table 3 shows the Ateme EMIF reference design signals.

Table 3. Signals (Part 1 of 2)

Signal Name	Width	Direction	Description
clk_in	1	I	Clock.
rst_n	1	I	Asynchronous reset, active low.
emif_ea[11:2]	10	I	EMIF address.
emif_ce_n	1	I	EMIF device enable. The device select must be set up as asynchronous.
emif_be_n	1	I	EMIF byte enable.
emif_aoe_n	1	I	EMIF asynchronous output enable.
emif_are_n	1	I	EMIF asynchronous read enable.
emif_awe_n	1	I	EMIF asynchronous write enable.
emif_ardy	1	O	
emif_ed[31:0]	32	I/O	EMIF data.
tx_dma_evrq_n	1	O	Transmit DMA event request. Asserted low to request a new block of data to be encoded.
rx_dma_evrq_n	1	O	Receive DMA event request. Asserted low to signal that a block of encoded data is available.
nmrstcha	1	O	Master reset to synchronous FIFO buffer.
fifo_dcha	32	I/O	Bidirectional data bus to/from FIFO buffer.
ncscha	1	O	Device select to FIFO buffer.
wnrcha	1	O	Active high write strobe to FIFO buffer.
nffchrxa	1	I	Receive FIFO buffer (FPGA to digital signal processor) full.
naffchrxa	1	I	Receive FIFO buffer (FPGA to digital signal processor) almost full.
nefchtxa	1	I	Transmit FIFO buffer (digital signal processor to FPGA) empty.
naefchtxa	1	I	Transmit FIFO buffer (digital signal processor to FPGA) almost empty.
soft_reset	1	O	Software controlled reset for co-processor.
addr	10	O	Control port address.
wdata	32	O	Control port write data.
hwa_ce	1	O	Control port chip enable.
write	1	O	Control port write strobe.
hwa_rdata	32	I	Control port read data.
tx_dav	1	I	Atlantic master source (transmit) data available.

Table 3. Signals (Part 2 of 2)

Signal Name	Width	Direction	Description
tx_ena	1	O	Atlantic master source (transmit) enable.
tx_dat	32	O	Atlantic master source (transmit) data.
tx_adr	4	O	Atlantic master source (transmit) address.
tx_eop	1	O	Atlantic master source (transmit) end of packet.
tx_sop	1	O	Atlantic master source (transmit) start of packet.
rx_ena	1	O	Atlantic master sink (receive) enable.
rx_dat	32	I	Atlantic master sink (receive) data.
rx_eop	1	I	Atlantic master sink (receive) end of packet.
rx_sop	1	I	Atlantic master sink (receive) start of packet.
rx_err	1	I	Atlantic master sink (receive) error.
rx_dav	1	I	Atlantic master sink (receive) data available.
rx_val	1	I	Atlantic master sink (receive) data valid.
av_address	22	O	Avalon address.
av_data_in	32	I	Avalon read data.
av_data_out	32	O	Avalon write data.
av_read_n	1	O	Avalon read strobe, active low.
av_write_n	1	O	Avalon write strobe, active low.
av_wait	1	I	Avalon wait request.
tx_fifo_full	1	I	Transmit (software to hardware) FIFO buffer status.
rx_fifo_full	1	I	Receive (hardware to software) FIFO buffer status.

Registers

Byte and half-word accesses to the FPGA co-processor registers have undefined results. Only word accesses are allowed on the DMD642 (EMIF byte enables are not connected to the FPGA). The Atlantic interface is always big-endian. The symbol size for the Atlantic converters is fixed at 8 bits. Symbol swapping would be required between the EMIF and the Atlantic interface to cope with a digital signal processor running in little endian mode.

Table 4 summarizes the registers present in the EMIF reference design.

Table 4. Registers

Mnemonic	Address (h) *(1)*	Access	Description
TX_CREDIT	–	–	Transmit credit register—not directly accessible.
VERSION	00	Read	Co-processor version.
TEST	24	Read/Write	Test register.
RXCONV_STATUS	80	Read	DMA event status register.
TX_CREDIT_INC	80	Write	Increment TX_CREDIT register.
CONTROL	C0	Write	Control register.

Note to Table 4:
(1) The addresses are offsets from the FPGA base address of A0080000h.

Transmit Credit Register (TX_CREDIT)

The FPGA co-processor must signal a DMA event to the EDMA to trigger the writing of packets in the transmit direction. This action is ultimately under control of the processor, which must write to the TX_CREDIT register within the FPGA co-processor to give the co-processor permission, or credit, to process a packet, e.g., once the associated transmit and receive buffers have been allocated in memory. While the FPGA co-processor has one, or more, transmit credits it requests a transmit DMA packet whenever the FIFO interface is idle. The TX_CREDIT register is decremented when a packet is moved from the bi-directional FIFO buffer to the FFT MegaCore function. The TX_CREDIT register is not directly accessible.

Version Number Register (VERSION 00h)

The version number register is a read only register that gives the FPGA co-processor version number.

Table 5 shows the version number register format.

Table 5. version Number Register Format

Data Bit	Mnemonic	Description
31:0	VERSION	Version number = 1d1d1d1dh.

Test Register (TEST 24h)

The test register is a read/write test register that is required by the DMDK642 boot firmware. Do not use this register.

Transmit Credit Increment Register (TX_CREDIT_INC 80h)

Table 6 shows the transmit credit register format.

Table 6. Transmit Credit Register Format

Data Bit	Mnemonic	Description
0	S	A write with the S bit asserted causes a soft reset of the FPGA co-processor.
1	I	A write with the I bit asserted causes the tx_credit register to be incremented.
31:2	0	Always write 0 for future compatibility.

DMA Event Status Register (RXCONV_STATUS 80h)

Table 7 shows the DMA event status register format.

Table 7. DMA Event Status Register Format

Data Bit	Mnemonic	Description
0	Tx	State of transmit DMA event request (active high).
1	Rx	State of receive DMA event request (active high).
31:2	0	Always reads 0.

Control Register (CONTROL C0h)

Table 8 shows the control register format.

Table 8. Control Register Format

Data Bit	Mnemonic	Description
0	INV	0 = MegaCore function performs FFT transform; 1 = MegaCore function performs inverse FFT transform.
31:1	0	Always write 0 for future compatibility.

Cost & Performance Improvements

This section discusses the cost and performance improvements for the following areas:

- The reference design
- Best-case scenario
- Enhancements

The Reference Design

Table 9 shows the FFT MegaCore function parameters in the reference design.

Table 9. FFT MegaCore Function Parameters

Parameter	Setting
Target device family	Cyclone
Transform length	1,024 points
Data precision	16 bits
Twiddle precision	16 bits
FFT engine architecture	Quad output
Number of parallel FFT engines	1
I/O data flow	Buffered burst
Structure	3 multipliers/5 adders
Implement multipliers in	Logic cells
Implement appropriate logic functions in RAM	Yes

For a full explanation of the parameters, refer to the *FFT MegaCore Function User Guide*.

The whole reference design uses approximately 8,600 logic elements (LEs) or 43% of the EP1C20 device. The EMIF interface is running at 66 MHz, which is limited by the external FIFO buffer. For simplicity, the FFT co-processor is also running at 66 MHz.

In the Altera Ateme EMIF Reference Design a 1,024 point 16-bit FFT can be completed in 31 µs. This compares to 10 µs if you using the entire DM642 processor running at 600 MHz. However, taking the 10k price of the Cyclone 1C20 and multiplying by the logic element usage (43%) gives an effective cost of $9.64. This compares against the 10k price of the DM642 digital signal processor. In this case, this FPGA co-processor shows a 1.5 times improvement in terms of price/performance.

Best Case Scenario

Table 10 shows the co-processor synchronous clock rates.

Table 10. Co-Processor Synchronous Clock Rates

Co-Processor	Synchronous Clock Rate (µs)		
	32-Bit 66-MHz	64-Bit 133-MHz	64-Bit 200-MHz
EMIF and transmit FIFO buffer	11.5	3.9	1.9
FFT MegaCore Function	19.5	9.7	6.4
EMIF and receive FIFO buffer	11.5	3.9	1.9
Average throughput	31	13.6	8.3

A solution based on a 64-bit synchronous EMIF running at 133 MHz and the FFT co-processor running at 200 MHz is estimated to produce an average throughput of 8.3 µs (see Table 10). The new Cyclone II device has multipliers that reduce the design size to under 5,000 LEs. Implementing this design in a EP2C8 device gives an effective cost of $5, yielding an price/performance improvement of 10.8 times.

Enhancements

If starting a new board-level FPGA co-processor design, the synchronous FIFO buffer on the ATEME DMDK board should be integrated into the FPGA. You should consider carefully the width and clock speed of the EMIF. Current C6x family digital signal processors support up to 64 bits EMIF at up to 133 MHz.

The use of FIFO buffers in the transmit and receive paths allows the FFT (or any other co-processor IP) to use a different clock than that used by the EMIF interface. If it is identified that the co-processor IP is a bottleneck then it may be run at a higher speed than the EMIF interface to decrease processing time. If other system issues mean that the co-processor IP is not a bottleneck, the clock speed can be reduced to reduce dynamic power consumption in the FPGA.

The FFT co-processor in this reference design is a relatively simple example. Once the decision is taken to implement a co-processor in an FPGA, it is important to maximize the processing of data within the FPGA. This minimizes the overhead of transferring data between the digital signal processor and the FPGA in relation to the time taken in processing the data and maximizes overall system performance.

Using a higher speed interface such as Serial Rapid I/O or Serial Lite allows higher speed data transfers between the digital signal processor and the co-processor, further increasing overall system performance.

Example Software

The reference design includes a software example, which is a simple demonstration that streams blocks of data through the FFT co-processor. The software project is built using the digital signal processor and BIOS libraries included with the Texas Instruments Code Composer Studio software to configure the EDMA controller and interrupts.

Two DSP general purpose I/O (GPIO) pins are dedicated for use as event triggers for the EDMA: one for transmit data (digital signal processor to co-processor) and one for receive data (co-processor to EDMA). The co-processor requests a new transmit DMA whenever the `TX_CREDIT` register is non-zero and the FFT MegaCore function is ready for more data. It requests a receive DMA whenever a packet of data is available from the FFT MegaCore function in the receive FIFO buffer.

Each time a DMA is completed, the EDMA sends an interrupt request to the digital signal processor. The software tracks the number of packets transmitted and received. When a pre-defined number of packets are complete, the software calculates the performance of the FFT co-processor.

Apart from a software reset and incrementing the `TX_CREDIT` register, all accesses to the FPGA co-processor are performed by the EDMA controller. In the reference design the EDMA is set up by calling the `initEdma()` function in the **ateme_dma.c** source file. GPIO 0 is linked to the receive DMA; GPIO3 to the transmit DMA.

The example software `main()` routine is in the **ateme_fft.c** source file included with the reference design. `Main()` performs the following tasks:

- Sets up `timer0` for performance measurement
- Initializes the memory buffers with the sine wave data for the EDMA
- Resets the FPGA co-processor and synchronous FIFO buffer
- Initializes the TI chip support library
- Calls `initEdma()` to initialize the EDMA controller
- Starts the timer
- Increments `TX_CREDIT`
- Waits until all blocks have been processed
- Calculates average time to process one FFT

Each time a transmit or receive DMA interrupt occurs `edmaHwi()` is called to handle the interrupt. This function is in **ateme_dma.c**. Counters are updated to track the number of transmit and receive interrupts

received until a pre-defined limit is reached, at which point the EDMA and interrupts are disabled. The transmit interrupt handler increments the `TX_CREDIT` register to allow further blocks to be processed.

No further processing of the data is performed by the example software.

Figure 8 shows real input data.

The sampled sine wave continues for 1,024 samples. The imaginary input samples are all zero.

Figure 8. Real Input Data

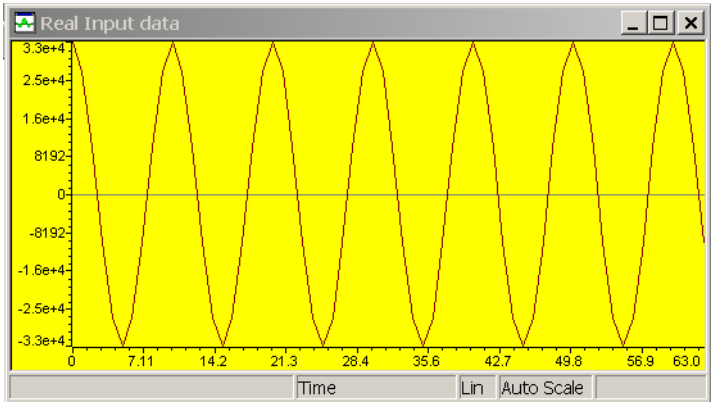


Figure 9 shows real output data.

Figure 9. Real Output Data

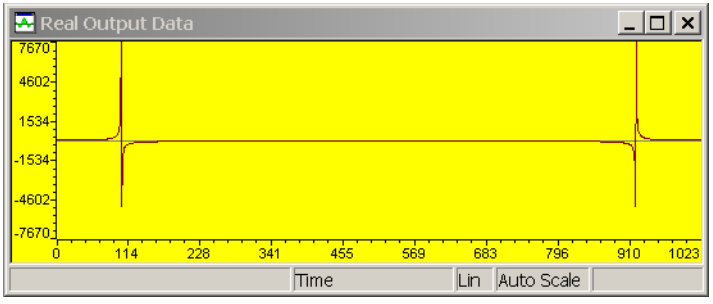


Figure 10 shows imaginary output data.

Figure 10. Imaginary Output Data

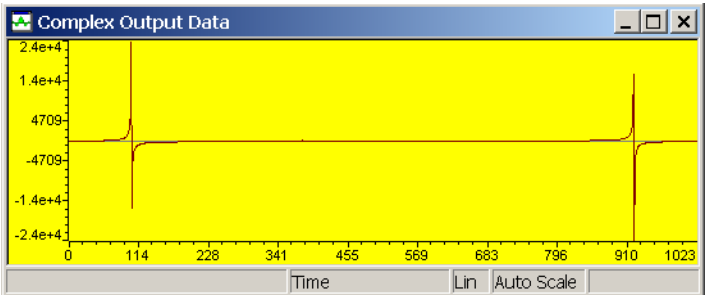
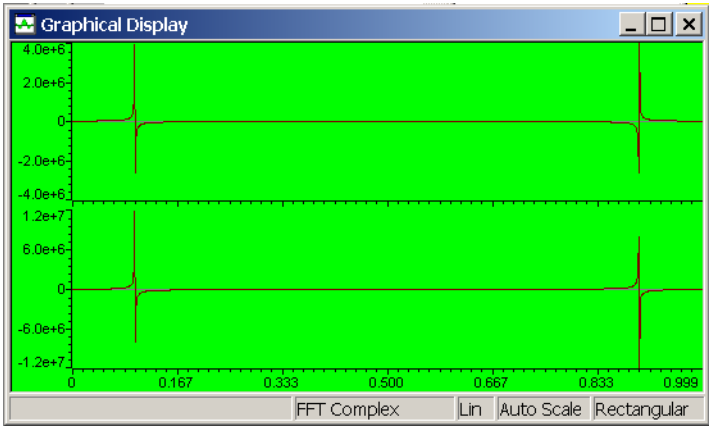


Figure 11 shows the output of Code Composer Studio with a complex FFT applied to the input data.

Figure 11. Code Composer Studio Output



Getting Started

This section describes the following steps:

1. Software Requirements.
2. Hardware Requirements
3. Install the Example System

4. Simulate the Example System

5. Run the Example Software

You must generate and synthesize the example design to generate the Quartus II Verilog HDL netlist, before you simulate the design.

Software Requirements

The reference design requires the following software:

- Quartus® II software version 4.0 or higher
- Altera FFT MegaCore function version 2.1.0
- Atlantic FIFO module
- ModelSim® simulator version 5.7e or higher
- Texas Instruments Code Composer Studio version 2.21 or higher

To use the Ateme EMIF reference design, you must first install the FFT MegaCore function and obtain a license. You can obtain a full license for the FFT MegaCore function, or you can use the function with an OpenCore Plus license. Either license allows the FFT MegaCore function to be instantiated from the example design.

For more inforrmation on OpenCore Plus hardware evaluation, refer to *AN 320: OpenCore Plus Hardware Evaluation*.

Hardware Requirements

To run the example software, you must have the Ateme DMDK642 development board with the reference design programmed into on-board flash for download to the FPGA at power on.

The DMDK642 development board must be connected to your PC via a suitable emulator that is supported by Code Composer Studio.

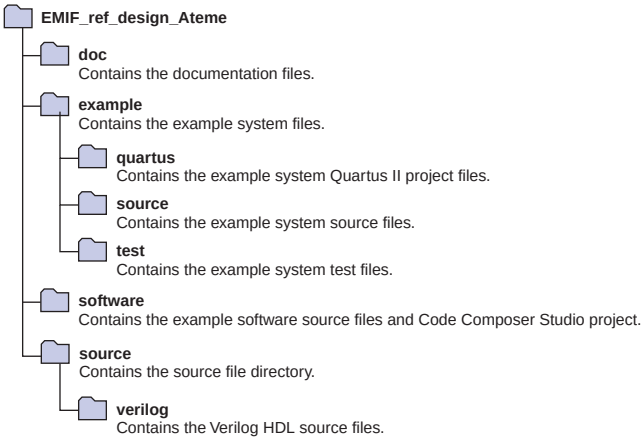
For information on how to load a new FPGA image, see section 4.8 in the *Ateme DMEK 642 Programmer's Guide*.

The **.rpd** file for the FPGA programming data is in the **example\Quartus** directory.

Install the Example System

To install the example system and EMIF reference design, run the **.exe** and follow the installation instructions. Figure 12 shows the directory structure.

Figure 12. Directory Structure



Simulate the Example System

The **example\test** directory contains a ModelSim project. A ModelSim **.do** file, **ateme_ref_ex.do**, contains all the commands necessary to compile and run the EMIF reference design example system testbench.

You may need to edit the **.do** file, to specify the correct installation directories and versions for the FFT MegaCore function and the Quartus II simulation libraries.

The **.do** file performs the following steps:

1. Refreshes the FFT MegaCore simulation libraries (may not be required depending upon the version of the ModelSim simulator).
2. Compiles the testbench source files.
3. Compiles the Verilog HDL simulation netlist output from the Quartus II software.
4. Compiles the Quartus II simulation library.

5. Opens the waveform window.

6. Runs the simulation.

If your ModelSim license supports mixed language simulation, you can use **ateme_ref_ex_real.do** to simulate the source files instead of the Quartus II output netlist.

To run the **.do** file, follow these steps:

1. Start the ModelSim simulator.

2. Choose **Open > Project** (File menu) and browse to the **EMIF_ref_design_ateme\example\test** directory.

3. Choose **ateme_ref_ex.mpf** and click **Open**.

4. Choose **Execute Macro** (Tools menu).

5. Choose either **ateme_ref_ex.do**, or **ateme_ref_ex_real.do** (see hand paragraph above) and click **Open**.

Run the Example Software

To run the example software, follow these steps:

1. Connect the emulator to the PC and the DMDK642 development board and power up.

2. Start Code Composer Studio.

3. Ensure the correct **GEL** file is loaded for the DMDK642 development board (see Ateme documentation).

4. Choose **Open** (Project menu).

You may be prompted to browse to the library file **bsl-x_x-dmek642-bios-d.lib**, which is in the **DMEK 642\bin** directory where you previously installed the Ateme DMDK642 software.

5. Browse to the **\software** directory and open **ateme_fft.pjt**.

6. Choose **Build** (Project menu), to compile the project.

7. Choose **Load Program** (File menu).

8. Browse to the **\software\debug** directory and open **ateme_fft.out**, to load the software into the DMDK642 development board.

9. Choose **Run** (Debug menu), to run the software on the DMDK642 development board.