ECE532 - Digital Hardware
Group Project Report

Audio-to-MIDI Converter

James Shu-Hen Chen 991163904
Sang-Joon Lee 990908354

Table of Contents

TOPIC Pages

1 Overview of the Project

1.1 Objective

The initial objective of this project is to implement an Audio-to-MIDI (ATM) converter on the Xilinx Vertex-II Multimedia Board. The board would sample audio (music) signals as an input, and outputs MIDI sequences corresponding to the music in real time.

1.2 Background

Musical Instrument Digital Interface (MIDI) is a standard in transmitting musical audio information in digital format. The standard is supported by most musical synthesizers, where the musical notes are synthesized and/or manipulated.

An ATM Converter adds MIDI compatibility to non-MIDI instruments. It converts audio signals produced by conventional instruments into the MIDI standard, thus allowing digital manipulation of the musical notes. The audio information can be outputted to a MIDI synthesizer. Additionally, the ATM Converter can record music into compact MIDI data files.

A good introduction to the MIDI standard can be found at [1] and [2].

1.3 Organization

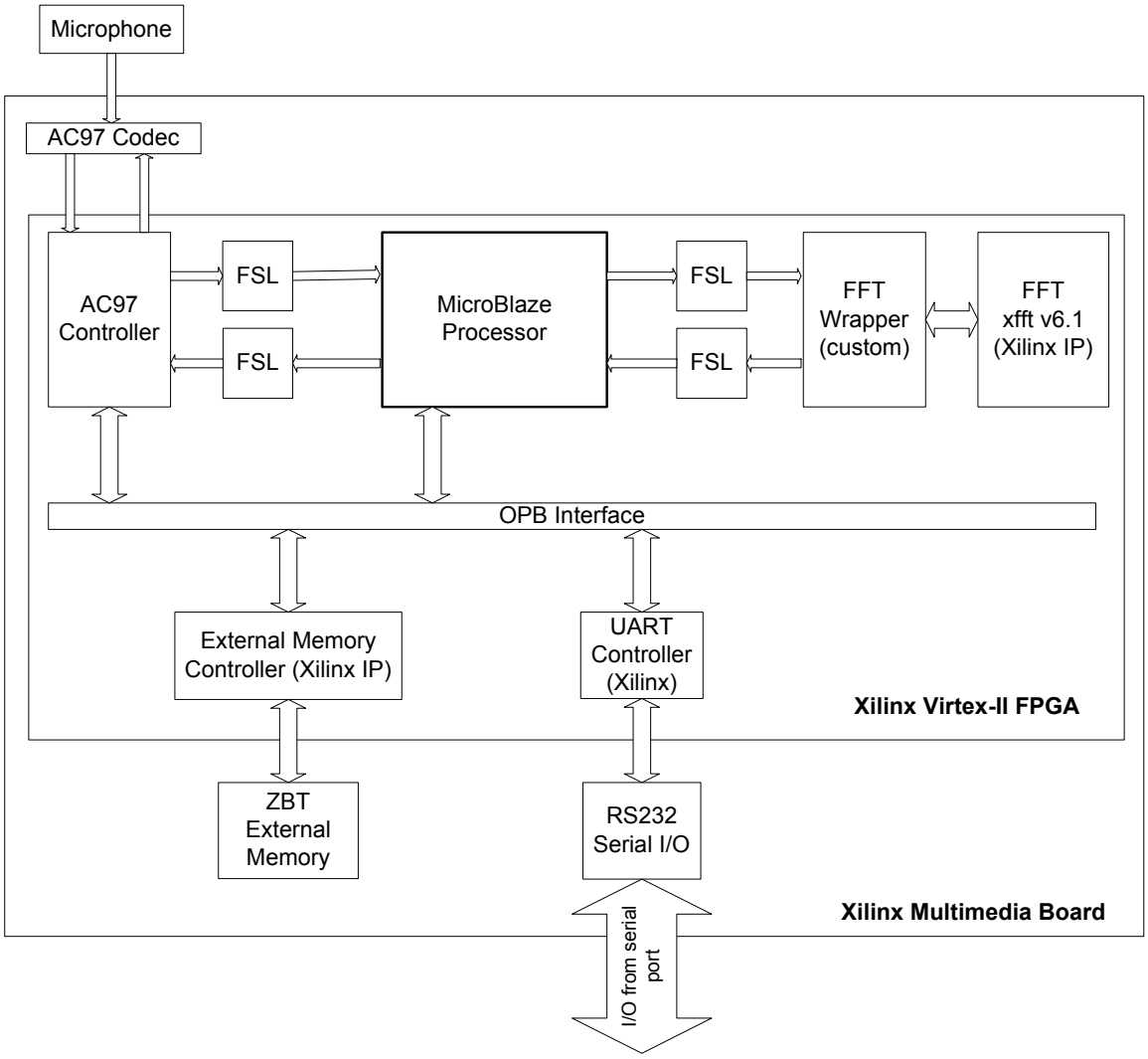


Figure 1 System Block Diagram

AC97 Controller

Responsible for capturing audio sample.

External Memory Controller

Required to access the ZBT memory

UART Controller

Allows system to output MIDI sequences through the RS232 Port.

FFT Core

Performs FFT operations for 1024 points.

FFT Wrapper

Allows the FFT core to interface with the FSL bus.

MicroBlaze Processor

Processes FFT data to determine whether or not a note has played or stopped, and sends MIDI sequences to the RS232 port corresponding to these events.

Fast Simplex Link (FSL)

Allows communication between the MicroBlaze Processor (software) and the various hardware blocks.

2 Outcome

Because of the various difficulties we encountered with the FFT, our system is unable to convert analog audio signals to MIDI sequences. However, the system is capable of capturing audio samples at 8000 Hz and determining which music note has been played from the hard-coded FFT data. Upon detecting a note has been played, series of bytes will be outputted from the system through the RS232 port representing the start of a note, the MIDI channel, index of the note, and the velocity (loudness).

We tested the rest of the system with a function that generate stub FFT outputs, and everything appears to be functioning correctly.

Since the MIDI interface is not directly compatible to the RS232 interface, a serial to MIDI converter must be attached between the RS232 port at the multimedia board and the MIDI synthesizer. A comparatively inexpensive solution can be ordered from http://www.ittymidi.com/converter_box_info.asp

Future Improvements

To reproducing the music more accurately, it is possible to monitor the time domain audio information to detect any significant change. We can then perform an FFT operation using the audio data follows immediately after the change. This not only minimizes the variance in the delay between a note is played and the MIDI sequence is sent, it also can give a better representation of the velocity (loudness) of the note.

Another possible improvement is, of course, to get the FFT wrapper working. Also, using a radix-4 or pipelined version of FFT may slightly improve the performance.

3 Description of IP Blocks

3.1 AC'97 Sound Controller

The AC97 Sound Controller IP block was provided by previous year's project. The core is responsible for controlling serial data flow between the external AC97 CODEC chip and the FPGA. The controller provides the user with capturing sound and playback audio signal through the FSL bus.[4] This IP block was used to capture audio signal and stored in the ZBT memory to perform FFT. Approximately 6 ms before disabling sound recording, the captured sound will contain gibberish data. The sound samples within this timeframe would be discarded.

Please refer to 'AC97 Controller' document from pervious year for more detail.

3.2 External Memory Controller (Xilinx)

The external memory core interfaces the FPGA to the external memory. The data is received from the AC97 Controller is written and stored in the external memory. These data will later be processed with a FFT module. Also, the software core is stored in the external memory. Please refer to ZBT memory data sheet provided by Xilinx.

3.3 UART (Xilinx)

This is the Uart peripheral for input/output used by the Xilinx Xilkernel. It is used to output the MIDI sequences to the RS232 port.

3.4 Fast Simplex Link (Xilinx)

The Fast Simplex Link (FSL) is a uni-directional point-to-point communication channel bus used to perform fast communication between any two design elements on the FPGA when implementing an interface with the FSL bus.

The Fast Simplex Link (FSL) Bus (v2.00a) was implemented in this project. To implement real-time audio processing capability, the FSL bus was used to directly read from the AC97 core as well feed the audio signal from Microblaze to FFT core and retrieve result from FFT core.

Please refer to *Fast Simplex Link (FSL) Bus (v2.00a)* data sheet provided by Xilinx for detailed information regarding this IP block.

3.5 FFT Wrapper

3.5.1 Description

The Fast Fourier Transform (FFT) wrapper responsible for communicating controls signals to the FFT core through the FSL buses. The wrapper handles data inputs from FSL FIFO and signal synchronization issue between the FFT core and FSL buses.

The FFT core used is xfft v3.1 provided by Xilinx. This FFT core was generated using the LogicCore tool provided by Xilinx.

Features:

- FFT Architecture option: Radix-2
- Uses minimum resources.
- Forward complex FFT only
- Transform size N = 1024
- Input data type: uint16
- Output data type: uint 32
- Magnitude of the frequency signal is returned as output
- Input and Output represented in natural order
- For use with Xilinx Platform Studio

3.5.2 Theory of Operation

The FFT wrapper reads in N number of input and passes the value to FFT core. When all input data are read, the FFT core computes a 1024-point forward DFT. Please refer to Fast Fourier Transform data sheet provided by Xilinx for more details on how FFT is implemented.

Input data is represented in natural order, and the output data is also represented in natural order.

The input data is accepted as real component only. The imaginary component of the input is set to “0000000000000000” by default.

The FFT wrapper returns magnitude of the result signal as an output. This magnitude is calculated by following equation.

Frequency Magnitude ^2 = Real Comp*Real Comp + Imaginary Comp *Imaginary Comp

3.5.3 Limitation

The FFT wrapper does not support external input signals to control forward or inverse FFT option nor option to change the size of number of point FFT to be performed. Hence, these parameters are fixed for this FFT wrapper implementation in the HDL code to meet the requirements of the Audio-to-MIDI converter project. For Audio-to-MIDI converter project, only 1024-point forward FFT is allowed.

3.5.4 Design Parameters and Signals

This section describes design parameters, input and output signals required by the FFT wrapper.

Design Parameters

C_DWIDTH
Specifies the width in bits of the master and slave connected to FSL bus.

C_INPUT_FSL_DEPTH
Specifies the depth of the input FIFO implemented by the FSL bus. The depth can be as low as 1 or high as 8192. Since this project is implemented with 1024 point FFT, the C_INPUT_FSL_DEPTH has been set to 1024 by default.

C_OUTPUT_FSL_DEPTH
Specifies the depth of the output FIFO implemented by the FSL bus. The depth can be as low as 1 or high as 8192. Since this project implements 1024 point FFT, the FFT core generates 1024 points of FFT output data. Therefore, the C_OUTPUT_FSL_DEPTH has been set to 1024 by default.

Port Name	*Port Width*	*Direction*	*Description*
CLK	1	Input	Synchronous Clock
RESET	1	Input	FFT wrapper reset (Active High)
FSL_S_CONTROL	1	Input	Slave FSL control signal
FSL_S_DATA	16	Input	Input data bus: Real component only. b=16 bits
FSL_S_EXISTS	1	Input	Slave FSL data exists signal
FSL_M_FULL	1	Input	Master FSL full signal
FSL_S_CLK	1	Output	Slave asynchronous FSL clock
FSL_S_READ	1	Output	Slave FSL read signal
FSL_M_CLK	1	Output	Master asynchronous FSL clock
FSL_M_CONTROL	1	Output	Master FSL control signal
FSL_M_DATA	32	Output	Output data bus: Magnitude of output FFT data.
FSL_M_WRITE	1	Output	Master FSL write signal

CLK

Synchronous clock is used for FFT wrapper.

RESET
This input signal resets all parameter required for FFT wrapper and FFT core.

FSL_S_CONTROL
Input signal from FSL core. A single bit control signal that is propagated along with the data at every clock edge.

FSL_S_DATA
The input data register to store raw data from FSL bus.

FSL_S_EXISTS
Input signal from FSL core indicating that FIFO contains valid data. Whenever this signal has gone high data is read from FIFO.

FSL_M_FULL
An input signal from FSL core which indicate that FIFO used by FSL is full. Hence data can not be written into FSL until this signal has gone low.

FSL_S_CLK
This project uses synchronous clock mode, hence FSL_M_CLK has been set to CLK.

FSL_S_READ
Output signal to FSL core that controls the read acknowledge signal of the FIFO. When set to '1', the value of FSL_S_DATA and FSL_S_CONTROL are popped from the FIFO on a rising clock edge.

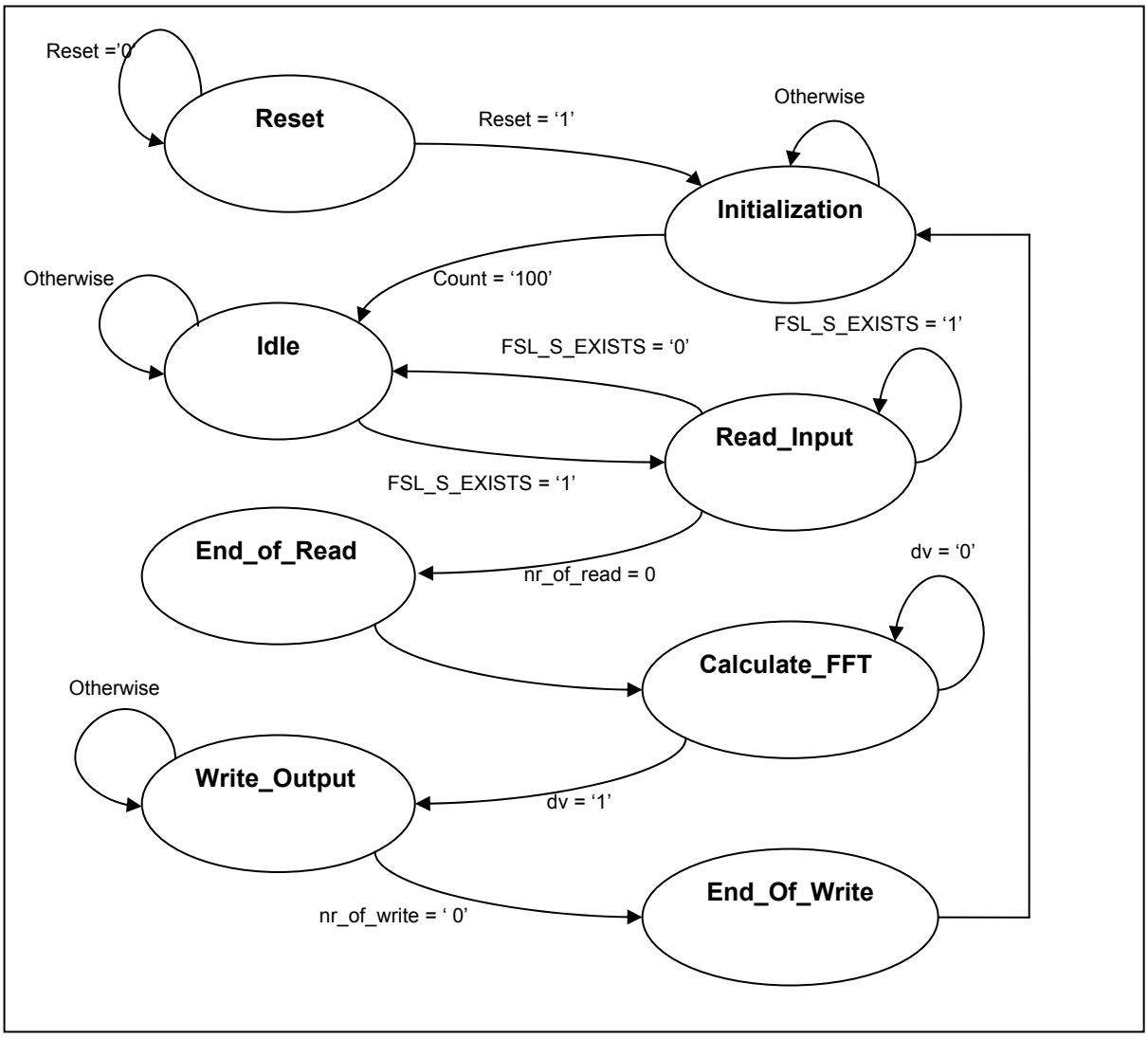
FSL_M_CLK
This project uses synchronous clock mode, hence FSL_M_CLK has been set to CLK.

FSL_M_CONTROL
Output signal to FSL core. A single bit control signal that is propagated along with the data at every clock edge.

FSL_M_DATA
The output data register to store magnitude of the FFT to FSL FIFO.

FSL_M_WRITE
An output signal that controls write enable signal of the FIFO. When set to '1', the value of FSL_M_DATA and FSL_M_CONTROL are pushed into the FIFO on the rising clock edge.

3.5.5 Finite State Machine Description



Description of each state:

Reset State:

In this state, the following internal and external signals are initialized.

- Internal signal, count, is initialized to "0000"
- Output signal to FFT core, ce (clock enable) is set to '1'.
- Master reset (sclr) for FFT core is set to '1'.
- Output signal to FFT core, unload signal is set to '0'.
- Data register for imaginary data is set to "00000000000000" by default.

Then, the state is set to '**Initialization**' state

Initialization State:

In this state, the following parameters are initialized for the FFT core.

- Load point size of the FFT transform. For this project, 1024 points have been implemented.
- Set FWD_INV signal to '1' to indicate forward FFT invocation.
- Start signal is set to '1' to begin reading input data to FFT.

After the Start signal is set to high, we must wait for 3 clock cycle before writing input data to FFT core. A counter has been implemented such that the state holds for 3 clock cycles before it is changed to 'Idle' state.

Idle State:

In this state,

- If FSL_S_EXISTS is '1', then state is changed to 'Read_Input' state.
- While FSL_S_EXISTS is '0', ce (clock enable) signal for FFT core is set to '0' so that no invalid data are written as an input.

Read_Input:

In this state,

- If FSL_S_EXISTS is '1', then read the data from FSL bus and write as input FFT with imaginary component set to "0000000000000000".
- If FSL_S_EXISTS is '0', then ce (clock enable) signal for FFT is disable so that no invalid data are written as an input.
- If all of 1024 points are read, then the state is change to 'End_Of_Read' state.

End_Of_Read:

This state reads the last input data and changes the state to 'Calculate_fft'

Calculate FFT:

In this state, the FFT is performed by the FFT core. When the signal 'done' is raised to '1',then an input signal 'unload' is set to '1' to unload FFT result in normal order. When the output signal, valid data (dv), is raised to '1' which indicates that output is ready to be read, then the state will change to 'Write_Output' state.

Write Output:

In this state, FSL_M_WRITE is set to '1' and output data from FFT core is read from FFT core and written to FSL FIFO. The state will not change until all of 1024 FFT output point has been written to FSL FIFO. When all values are written, the state is changed to 'End_of_Write' state.

End_of_Write:

In this state, the last FFT result is written to FSL FIFO and then the state is changed back to '**Initialization**' state.

3.5.6 Simulation

The FFT wrapper core was thoroughly simulated before it was downloaded to the FPGA. To do this, ModelSim 6.0 was used to simulate the FFT core wrapped with the FFT wrapper. A test bench was setup using ModelSim 6.0 environment by writing do scripts which defines the behaviour of FFT wrapper instance. A test bench had been setup under ISE environment, however, it was difficult and more time consuming to modify test vectors in ISE than ModelSim. Therefore, ModelSim was chosen as test bench environment.

3.6 Software Sound Processor

3.6.1 Description

The Software Sound Processor allows MicroBlaze processor to retrieve audio samples from the AC97 controller, pass these samples to the FFT core, retrieve FFT data, and analyzing the FFT data to determine if a MIDI sequence should be sent. Currently, the software only detects the presence of the loudest note that has been played.

The software first configures the AC97 controller to capture sound samples at 8000Hz. It reads the sound samples from the AC97 controller via the Fast Simplex Link (FSL) and send the sound samples to the Fast Fourier Transform (FFT) core in 128 ms chunks via FSL[1]. Then, the software retrieves the FFT data from the FFT core via FSL[2] and determine from this data if a note has been played or stopped playing. Finally, it determines the MIDI sequence to be sent and outputs the MIDI sequence to the RS232 port. The process is illustrated in the following figure:

[1] This function is currently commented out in the software, as the FFT wrapper is not yet functional.
[2] This function is currently replaced with another function that generates hard-coded FFT data.

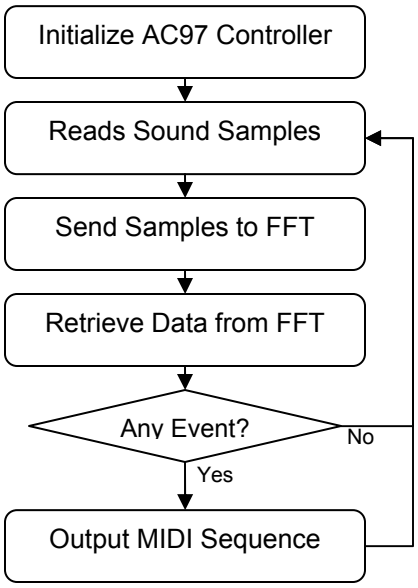


Figure 2 – Software Flow

3.6.2 Theory of Operation

Music notes can be represented by the type of note and the octave in which the note resides in. The first octave includes notes in the 32.7Hz to 61.74Hz, and the N^{th} octave includes notes from ($23.7 \times 2^{N-1}$) to ($61.74\ 7 \times 2^{N-1}$) Hertz. Thus, the octave N for each note with frequency F can be determined using the following inequality:

$$F / 2^{N-1} < 63 \text{ Hz}$$

The term *frequency / 2^{N-1}* represent the note's base frequency, which we can use to distinguish the type of music note within an octave. There are twelve notes within an octave, where each consecutive note has frequency of $\sqrt[12]{2}$ =1.0595 times greater than the previous note.

The software samples sound at 8000Hz, and pass the stored samples to the Fast Fourier Transform (FFT) core in128 ms chunks via the Fast Simplex Link (FSL). The FFT core would return the frequency domain representation of the captured sound data.

3.6.3 Limitations

At the sampling rate of 8000Hz and FFT vector length of 1024 values, we can only detect music notes in the third octave or higher. When the frequency of the

note falls below the third octave, two or more notes may appear in the same frequency range in the FFT output, thus making it difficult or impossible to distinguish which note is playing. The current implementation ignores any note that falls below the third octave.

There is approximately a 100ms delay between the start of the note to the corresponding MIDI sequences being outputted from the RS232 port. Majority of the delay comes from the time it takes to gather sound samples for the FFT.

3.6.4 Testing

Since the FFT core is not yet functional, a function that generates hard-coded FFT results was written for testing purposes. The following test cases were considered:

- First note starts playing
- A louder note starts playing while the first tone is still playing.
- Without releasing the note, the same note is played again.
- All frequencies are below the threshold value.

4 Design Tree

The structure of the project is as describe below:

AudioToMidi\ - Contains the audio to midi system.

system.mhs: A higher level description of the hardware modules in the system
system.mss: A higher level description of the software modules in the system

AudioToMidi\code\ - The source code for the software component of the audio-to-MIDI converter system.
system.c – software sound processor.
system_demo.c – software sound processor for demo.

AudioToMidi\data\: Contains the user constraint file (.ucf) which assigns external pins to ports, sets clock speed etc..

AudioToMidi\pcores\: Contains user designed peripherals

- *AudioFFT1024_v1_00_a (Custom IP v1.00)* is used to perform 1024 point FFT on audio signal.

Directory Structure	*Description*
…\audiofft1024_v1_00_a\	Contains 1024 point FFT core
…\audiofft1024_v1_00_a\data	Contains .mpd, .bbd and .pao files
…\audiofft1024_v1_00_a\hdl\vhdl\	Contains FFT wrapper file and FFT core wrapper file.
…\audiofft1024_v1_00_a\netlist\	Contains generated FFT netlist (.edn) and ngc file.
…\audiofft1024_v1_00_a\simulation	Contains ModelSim script used for simulation
...\audiofft1024_v1_00_a\simulation\ readMe.txt	This readMe file contains instructions to run .do scripts and briefly explains each test cases.

- *Opb_ac97_controller_v3_10_a* (User IP) is used to capture, record and playback audio signals.

Directory Structure	*Description*
…\opb_ac97_controller_v3_10_a \	Contains AC97 IP core
…\opb_ac97_controller_v3_10_a \data	Contains .mpd, .bbd and .pao files
…\ Opb_ac97_controller_v3_10_a \hdl\vhdl\	Contains AC97 controller file.
…\ Opb_ac97_controller_v3_10_a \netlist\	Contains generated netlist (.edn).

- *gen_zbt_addr_v1_00_a* is used to map the address lines of the ZBT connection to the EMC controller.

- *RS232* is the OPB UART controller core. The serial connection is used to display printf statements mainly for debugging purposes.

- *lmb_bram* is the Block RAM (BRAM) connected by the Local Memory Bus (LMB) generated by the Base System Builder. This BRAM is used to store the xmdstub such that the executable can be downloaded to the FPGA.

References

[1] MIDI Specification 1.0
http://www.sfu.ca/sca/Manuals/247/midi/MIDISpec.html
[2] The MIDI Specification
http://www.ibiblio.org/emusic-l/info-docs-FAQs/MIDI-doc/
[3] Look RS232
http://www.lookrs232.com/
[4] AC97 Controller

Appendix A: FFT Wrapper Core

A1: audiofft1024.vhd code listing:

```
------------------------------------------------------------------------------
-- audiofft1024 - entity/architecture pair
------------------------------------------------------------------------------
--
------------------------------------------------------------------------------
-- Filename:          audiofft1024
-- Version:           1.00.a
-- Date:              Mon Mar 21 22:21:17 2005
-- VHDL-Standard:     VHDL'93
------------------------------------------------------------------------------
-- Naming Conventions:
--      active low signals:                  "*_n"
--      clock signals:                       "clk", "clk_div#", "clk_#x"
--      reset signals:                       "rst", "rst_n"
--      generics:                            "C_*"
--      user defined types:                      "*_TYPE"
--      state machine next state:                "*_ns"
--      state machine current state:             "*_cs"
--      combinatorial signals:                   "*_com"
--      pipelined or register delay signals:   "*_d#"
--      counter signals:                       "*cnt*"
--      clock enable signals:                    "*_ce"
--      internal version of output port:       "*_i"
--      device pins:                         "*_pin"
--      ports:                               "- Names begin with Uppercase"
--      processes:                             "*_PROCESS"
--      component instantiations:              "<ENTITY_>I_<#|FUNC>"
------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
--use ieee.std_logic_unsigned.all;


------------------------------------------------------------------------------
--
-- Definition of Parameters
-- C_WIDTH              : Signal width
--
-- C_INPUT_FSL_DEPTH       : Input FSL bus depth
-- C_OUTPUT_FSL_DEPTH    : Output FSL bus depth
--
-- Definition of Ports
-- CLK             : Synchronous clock
-- RESET           : System reset, should always come from FSL bus
-- FSL_S_CLK       : Slave asynchronous clock
-- FSL_S_READ      : Read signal, requiring next available input to be read
-- FSL_S_DATA      : Input data
-- FSL_S_CONTROL   : Control Bit, indicating the input data are control word
-- FSL_S_EXISTS    : Data Exist Bit, indicating data exist in the input FSL bus
-- FSL_M_CLK       : Master asynchronous clock
-- FSL_M_WRITE     : Write signal, enabling writing to output FSL bus
-- FSL_M_DATA      : Output data
-- FSL_M_CONTROL   : Control Bit, indicating the output data are contol word
-- FSL_M_FULL      : Full Bit, indicating output FSL bus is full
--
------------------------------------------------------------------------------

------------------------------------------------------------------------------
-- Entity Section
```

```
------------------------------------------------------------------------------

entity audiofft1024 is

	generic (
		C_INPUT_DWIDTH		: integer := 16;
		C_OUTPUT_DWIDTH		: integer := 32;

		-- These two parameters determine the default value of
		-- C_FSL_DEPTH for FSL bus.
		--
		-- Do not change the names of these parameters.
		--
		-- The default depths of the input and output FSL buses
		-- are set to the total number of input and output
		-- words respectively.
		--
		-- If you change the default values here, remember to update
		-- them in audiofft1024_v2_1_0.mpd file.

		C_INPUT_FSL_DEPTH  : natural := 1024;
		C_OUTPUT_FSL_DEPTH : natural := 1024
	);

	port
	(
		-- Bus protocol ports.
		CLK			: in  std_logic;
		RESET			: in  std_logic;
		FSL_S_CLK		: outstd_logic;
		FSL_S_READ		: outstd_logic;
		FSL_S_DATA		: in  std_logic_vector(0 to C_INPUT_DWIDTH-1);
		FSL_S_CONTROL		: in  std_logic;
		FSL_S_EXISTS		: in  std_logic;
		FSL_M_CLK		: outstd_logic;
		FSL_M_WRITE		: outstd_logic;
		FSL_M_DATA		: outstd_logic_vector(0 to C_OUTPUT_DWIDTH-1);
		FSL_M_CONTROL		: outstd_logic;
		FSL_M_FULL		: in  std_logic
	);
end audiofft1024;

------------------------------------------------------------------------------
-- Architecture Section
------------------------------------------------------------------------------
architecture arch_FFT of audiofft1024 is

-- Import external component:
--  1024 point FFT
--      * Radix-2 Architecture
--      * 16 bit signal width
--      * Clock Enable (ce) option
--      * Master Reset Option
--      * No scaling option
component fft1024radix2 is
	port (
	xn_re		: IN std_logic_VECTOR(15 downto 0);
	xn_im		: IN std_logic_VECTOR(15 downto 0);
	start		: IN std_logic;
	unload		: IN std_logic;
	nfft		: IN std_logic_VECTOR(4 downto 0);
	nfft_we		: IN std_logic;
	fwd_inv		: IN std_logic;
	fwd_inv_we	: IN std_logic;
	sclr		: IN std_logic;
	ce		: IN std_logic;
	clk		: IN std_logic;
	xk_re		: OUT std_logic_VECTOR(26 downto 0);
	xk_im		: OUT std_logic_VECTOR(26 downto 0);
	xn_index	: OUT std_logic_VECTOR(9 downto 0);
```

```
        xk_index              : OUT std_logic_VECTOR(9 downto 0);
        rfd                   : OUT std_logic;
        busy                  : OUT std_logic;
        dv                    : OUT std_logic;
        edone                 : OUT std_logic;
        done                  : OUT std_logic);
end component fft1024radix2;

        -- signals connected to fft1024radix2
        signal xn_re          : std_logic_VECTOR(15 downto 0);
        signal xn_im          : std_logic_VECTOR(15 downto 0);
        signal start          : std_logic;
        signal unload         : std_logic;
        signal nfft           : std_logic_VECTOR(4 downto 0);
        signal nfft_we        : std_logic;
        signal fwd_inv        : std_logic;
        signal fwd_inv_we     : std_logic;
        signal sclr           : std_logic;
        signal ce             : std_logic;
        --signal clk           : std_logic; --
        signal xk_re          : std_logic_VECTOR(26 downto 0);
        signal xk_im          : std_logic_VECTOR(26 downto 0);
        signal xn_index       : std_logic_VECTOR(9 downto 0);
        signal xk_index       : std_logic_VECTOR(9 downto 0);
        signal rfd            : std_logic;
        signal busy           : std_logic;
        signal dv             : std_logic;
        signal edone          : std_logic;
        signal done           : std_logic;

        -- states & bookeeping signals
        signal count          : std_logic_vector(4 downto 0);
        signal input_data     : std_logic_vector(15 downto 0);
        signal output_data    : std_logic_vector(15 downto 0);
        signal output_is_valid : std_logic;
        signal fsl_is_full    : std_logic;
        signal dataInBuf      :  std_logic_vector(15 downto 0);
        signal fftDataBuf_re  : std_logic_vector (15 downto 0);
        signal fftDataBuf_im  : std_logic_vector (15 downto 0);
       signal fftMagnitude    : std_logic_vector (31 downto 0);

-- In the Custom Function Wizard you specified the following
-- Number of input arrays: 1
-- Size of each input array: 512
-- This constant contain the total number of input words,
-- which is the product of the above
constant NUMBER_OF_INPUT_WORDS  : natural := 16; --1024;

-- In the Custom Function Wizard you specified the following
-- Number of output arrays: 1
-- Size of each output array: 512
-- This constant contain the total number of input words,
-- which is the product of the above
constant NUMBER_OF_OUTPUT_WORDS : natural := 16; --1024;

-- Finite States
type STATE_TYPE is (Initialization, Idle, Read_Inputs, Write_Outputs, Cal_fft, End_Of_Read,End_Of_Write);

signal state       : STATE_TYPE;

-- Counters to store the number inputs read & outputs written
signal nr_of_reads  : natural range 0 to NUMBER_OF_INPUT_WORDS - 1;
signal nr_of_writes : natural range 0 to NUMBER_OF_OUTPUT_WORDS - 1;

begin

-- Include FFT 1024 generated core component
fft1024radix2_I : fft1024radix2
        port map (
        xn_re       => xn_re,       -- : IN std_logic_VECTOR(15 downto 0);
```

```
      xn_im       => xn_im,       -- : IN std_logic_VECTOR(15 downto 0);
      start       => start,       -- : IN std_logic;
      unload      => unload,      -- : IN std_logic;
      nfft        => nfft,        -- : IN std_logic_VECTOR(4 downto 0);
      nfft_we     => nfft_we,     -- : IN std_logic;
      fwd_inv     => fwd_inv,     -- : IN std_logic;
      fwd_inv_we  => fwd_inv_we,  -- : IN std_logic;
      sclr        => sclr,        -- : IN std_logic;
      ce          => ce,          -- : IN std_logic;
      clk         => clk,         -- : IN std_logic;
      xk_re       => xk_re,       -- : OUT std_logic_VECTOR(15 downto 0);
      xk_im       => xk_im,       -- : OUT std_logic_VECTOR(15 downto 0);
      xn_index    => xn_index,    -- : OUT std_logic_VECTOR(9 downto 0);
      xk_index    => xk_index,    -- : OUT std_logic_VECTOR(9 downto 0);
      rfd         => rfd,         -- : OUT std_logic;
      busy        => busy,        -- : OUT std_logic;
      dv          => dv,          -- : OUT std_logic;
      edone       => edone,       -- : OUT std_logic;
      done        => done);       -- : OUT std_logic;

FSL_S_READ  <= FSL_S_EXISTS   when state = Read_Inputs   else '0';

The_FFT : process (CLK,RESET) is
begin  -- process The_SW_accelerator

  if RESET = '1' then                   -- Synchronous reset (active high)
    -- Initialize all parameters on RESET signal
    nr_of_reads       <= 0;
    nr_of_writes      <= 0;
    count             <= "00000";             -- set counter to zero
    ce                <= '1';                 -- set clock enable to '1'
    sclr              <= '1';                 -- set Master Reset to '1'
    unload            <= '0';                 -- set unload results to '0'
    start             <= '0';                 -- set start signal to '0'
    xn_im             <= "0000000000000000";  -- intialize imaginary component to 0
    state             <= Initialization;      -- goto ready state
  elsif CLK'event and CLK = '1' then      -- Rising clock edge

    case state is
      -- INITIALIZATION STATE: Set all require parameters for FFT
      when Initialization =>
                                  if count = 0 then
                                        sclr         <= '0';        -- Master Reset to '0'
                                        nfft_we      <= '1';        -- NFFT set to '1' to invoke signal
                                        nfft         <= "00100";      --"01010";     -- Set to 16 point NFFT
                                        count        <= count + 1;   -- Update Clock count
                                    FSL_M_WRITE    <= '0';        -- set FSL_M_WRITE to '0', no data written to FFT
                                  elsif count = 1 then
                                        nfft_we      <= '0';
                                        nfft         <= "00000";
                                        fwd_inv_we   <= '1';        -- Setup fwd or inv NFFT
                                        fwd_inv      <= '1';        -- Set to '1' for forward FFT
                                        start        <= '1';        -- Set 'start' signal to high
                                        count        <= count + 1;   -- update clock count

                                  elsif count = 2 then
                                        fwd_inv_we   <= '0';
                                        fwd_inv      <= '0';
                                        start        <= '0';        -- down 'start' signal
                                        count        <= count + 1;   -- update clock count

                                  elsif count = 4 then          -- wait for 3 clock cycle before starting to read input
                                        count        <= "00000";     -- reset count to zero
                        --              ce           <= '0';        -- clock is disabled
                                        nr_of_reads  <= NUMBER_OF_INPUT_WORDS - 1;
                                        state        <= Idle;       -- goto next state 'Idle'
                                  else
                                        count        <= count + 1;   -- otherwise waste clock cycle
                                  end if;
```

```
-- IDLE STATE: wait until there is an input to read
when Idle =>
                        if (FSL_S_EXISTS = '1') then       -- If data exists
                                dataInBuf           <= FSL_S_DATA;     -- store to a buffer
                                nr_of_reads         <= nr_of_reads - 1; -- subtract number of data read
                                ce                  <= '1';            -- enable clock
                                state               <= Read_Inputs;    -- goto Read_Input state
                        else                        -- If data does NOT exists
                                ce                  <= '0';            -- disable the clock
                        end if;

-- READ_INPUT STATE: Reads input and write to xn_re and xn_im port of FFT
when Read_Inputs =>
                        if (FSL_S_EXISTS = '1') then                    -- If data exists
                                xn_re               <= dataInBuf;                   -- write to FFT
                                xn_im               <= "0000000000000000";
                                dataInBuf           <= FSL_S_DATA;                  -- read another input

                                if (nr_of_reads = 0) then                   -- If all N points are read to FFT
                                        nr_of_writes <= NUMBER_OF_OUTPUT_WORDS - 1;
                                        state       <= End_Of_Read;         -- goto next state
                                else
                                        nr_of_reads  <= nr_of_reads - 1;    -- If not, continue reading
                                end if;

                        else                                        -- If data does NOT exist
                                xn_re       <= dataInBuf;     -- pass the previously read data to FFT
                                xn_im       <= "0000000000000000";
                                ce          <= '0';                         -- disable clock
                                state       <= Idle;                        -- Goto Idle state
                        end if;

-- END_OF_READ STATE: Wait for FFT to perform calculation
when End_Of_Read =>
                        xn_re       <= dataInBuf;                   -- Pass the previously read data to FFT
                        xn_im       <= "0000000000000000";
                        state       <= Cal_fft;                     -- Goto CAL_FFT state

-- CAL_FFT STATE: wait for FFT to perform calculation
when Cal_fft =>
                        ce      <= '1';

                        if edone = '1' then                     -- If calculation is complete
                                unload <= '1';                          -- Invoke unload result
                        else
                                unload <= '0';                          -- else continue wait
                        end if;

                        if dv = '1' then                        -- If valid data is found
                                fftDataBuf_re <= xk_re(26) & xk_re(14 downto 0);  -- read the data
                                fftDataBuf_im <= xk_im(26) & xk_im(14 downto 0);
                                state           <= Write_Outputs;               -- then goto 'Write_Outputs' state
                        end if;

-- WRITE_OUTPUTS STATE: Write the FFT result to FSL
when Write_Outputs =>

                        if (nr_of_writes = 0) then                  -- until all results are read
                                state <= End_Of_Write;                  -- goto next state
                        else
                                if (FSL_M_FULL = '0') then              -- while FIFO is not full
                                        FSL_M_WRITE   <= '1';                   -- write FFT result to FIFO
                                        nr_of_writes  <= nr_of_writes - 1;

                                -- calculate maginitude of the frequency
                                fftMagnitude  <= fftDataBuf_re*fftDataBuf_re + fftDataBuf_im*fftDataBuf_im;

                                FSL_M_DATA    <= std_logic_vector (fftMagnitude);    -- Write to FIFO
                                fftDataBuf_re <= xk_re(26) & xk_re(14 downto 0);
                                fftDataBuf_im <= xk_im(26) & xk_im(14 downto 0);
```

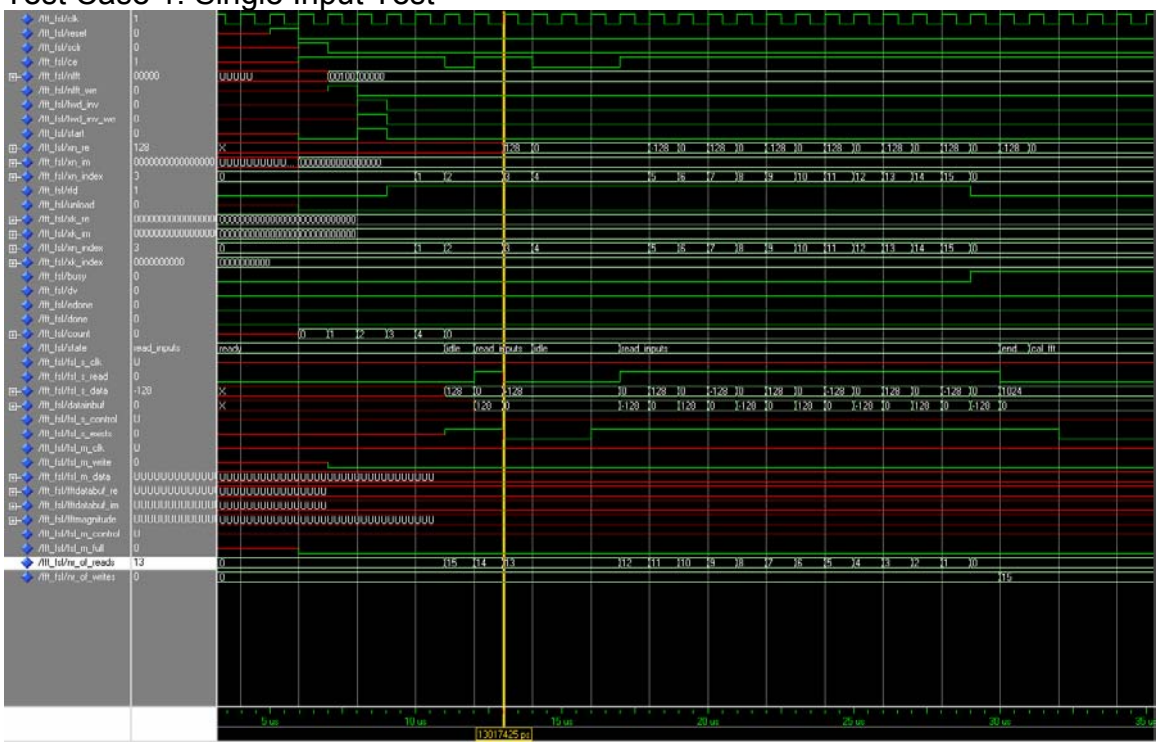
```
                                end if;
                        end if;

    -- END_OF_WRITE STATE: Completed one bash
    when End_Of_Write =>
                        fftMagnitude    <= fftDataBuf_re*fftDataBuf_re + fftDataBuf_im*fftDataBuf_im;
                        FSL_M_DATA      <= std_logic_vector (fftMagnitude);
                        FSL_M_WRITE     <= '1';                     -- Write to FSL FIFO
                        state           <= Initialization;              -- go back to ready state

    -- OTHERWISE: Reset and goto READY State
    when others =>
                        sclr            <= '0';
                        ce              <= '1';
                        nfft            <= "00000";
                        nfft_we         <= '0';
                        fwd_inv         <= '0';
                        fwd_inv_we      <= '0';
                        start           <= '0';
                        unload          <= '0';
                        state           <= Initialization;              -- goto 'Ready' State

        end case;
  end if;

end process The_FFT;
-- [End] of process

end architecture arch_FFT;
-- [end] of Architecture
```

A2: audiofft1024_v1_00_a.mpd listing:

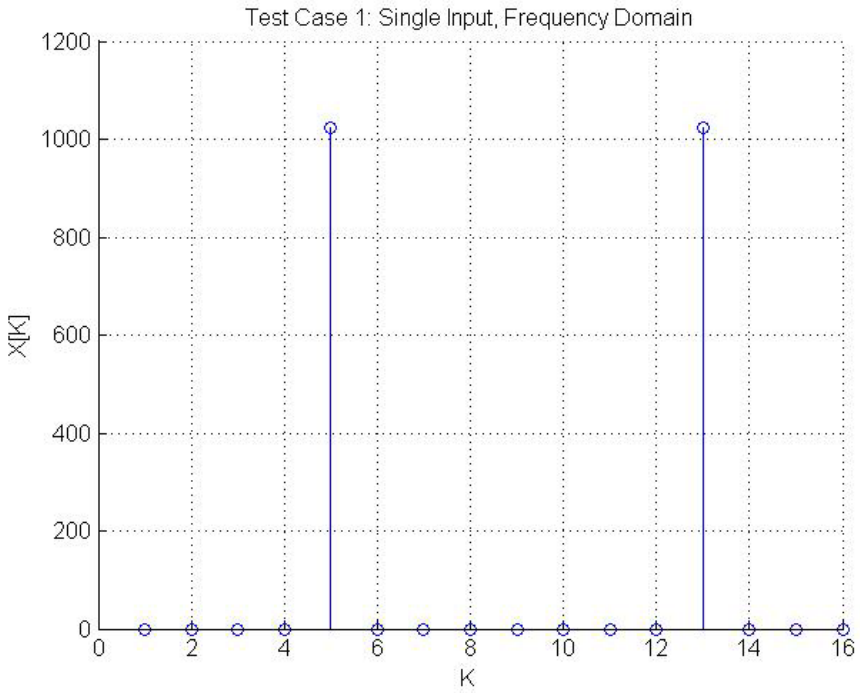
```
##############################################################################
##
## Microprocessor Peripheral Definition :
##
## Filename: audiofft1024_v2_1_0.mpd
## Description:       Microprocessor Peripheral Description
## Date:              Mar 2 20:42:20 2005
##############################################################################

BEGIN audiofft1024

##Peripheral Options
OPTION IPTYPE = PERIPHERAL
OPTION IMP_NETLIST = TRUE
OPTION HDL = VHDL

## GENERIC PARAMETERS:
PARAMETER C_INPUT_DWIDTH = 16, DT=integer
PARAMETER C_OUTPUT_DWIDTH = 32, DT=integer
PARAMETER C_INPUT_FSL_DEPTH = 1024, DT = NATURAL, BUS = MFSL:SFSL
PARAMETER C_OUTPUT_FSL_DEPTH = 1024, DT = NATURAL, BUS = MFSL:SFSL

## Bus Interfaces
BUS_INTERFACE BUS=SFSL, BUS_STD=FSL, BUS_TYPE=SLAVE
BUS_INTERFACE BUS=MFSL, BUS_STD=FSL, BUS_TYPE=MASTER

# Global ports
PORT Clk   = "", DIR=IN, SIGIS=CLK
# rosinger EDK 6.2i PORT Reset = "", DIR=IN, BUS = MFSL0:SFSL0
PORT Reset = "", DIR=IN

## proware signals
PORT FSL_S_CLK     = FSL_S_Clk,     DIR=out, SIGIS=CLOCK, BUS=SFSL,
PORT FSL_S_READ    = FSL_S_Read,    DIR=out, BUS=SFSL
PORT FSL_S_DATA    = FSL_S_Data,    DIR=in, VEC=[0:15], BUS=SFSL
PORT FSL_S_CONTROL = FSL_S_Control, DIR=in,BUS=SFSL
PORT FSL_S_EXISTS  = FSL_S_Exists,  DIR=in, BUS=SFSL

PORT FSL_M_CLK     = FSL_M_Clk,     DIR=out, SIGIS=CLOCK, BUS=MFSL
PORT FSL_M_WRITE   = FSL_M_Write,   DIR=out, BUS=MFSL
PORT FSL_M_DATA    = FSL_M_Data,    DIR=out, VEC=[0:31], BUS=MFSL
PORT FSL_M_CONTROL = FSL_M_Control, DIR=out, BUS=MFSL
PORT FSL_M_FULL    = FSL_M_Full,    DIR=in, BUS=MFSL

END
```

Appendix B: Simulation Results

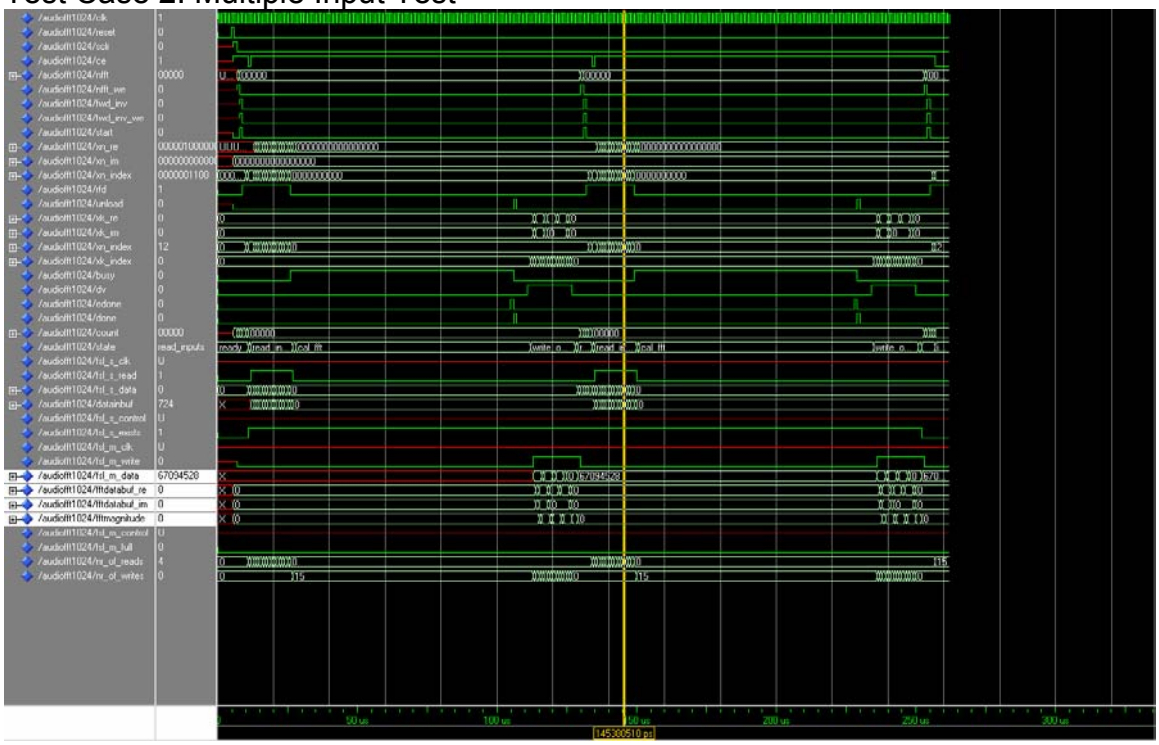
Test Case 1: Single Input Test



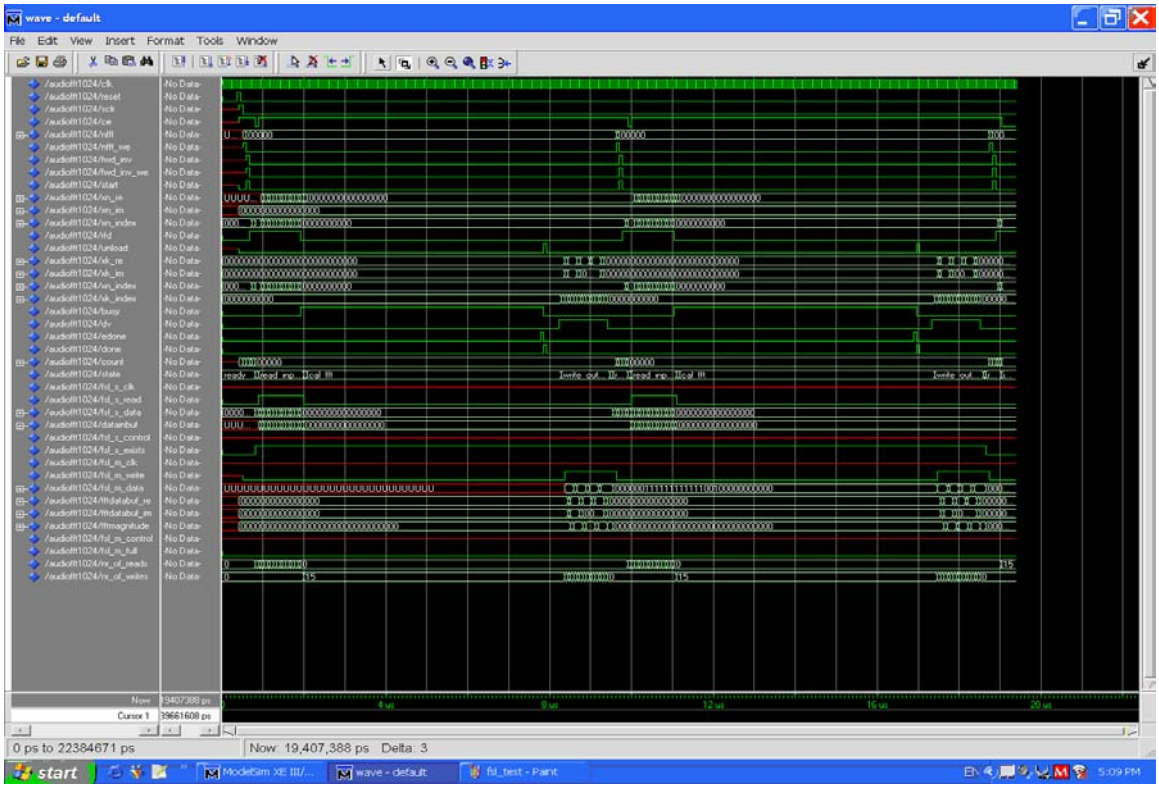
Test Case 1 - Verification: Matlab Simulation



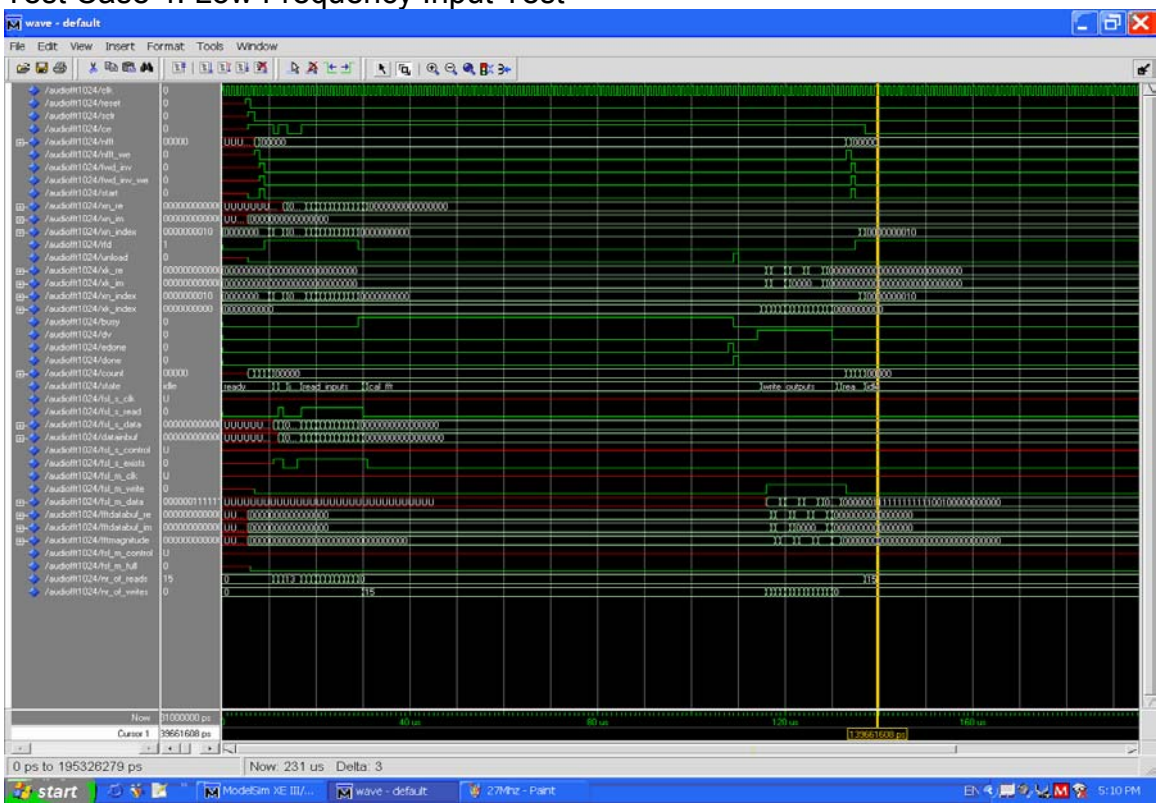
Test Case 2: Multiple Input Test



Test Case 3: 27 MHz Clock Test



Test Case 4: Low Frequency Input Test



Fast Fourier Transform v3.1

DS260 November 11, 2004 Product Specification

Introduction

The Fast Fourier Transform (FFT) is a computationally efficient algorithm for computing the Discrete Fourier Transform (DFT). The FFT core uses the Cooley-Tukey algorithm for computing the FFT[1].

Features

- Drop-in module for Virtex-II™, Virtex-II Pro™, Spartan-3™, and Virtex-4™ FPGAs
- Forward and inverse complex FFT
- Transform sizes $N = 2^m$, $m = 3 – 16$
- Data sample precision b_x = 8,12,16,20,24
- Phase factor precision b_w = 8,12,16,20,24
- Arithmetic types:
  - Unscaled (full-precision) fixed point
  - Scaled fixed point
  - Block floating point
- Rounding or truncation after the butterfly
- On-chip memory
- Block RAM or Distributed RAM for data or phase factor storage
- Run-time configurable forward or inverse operation
- Optional run-time configurable transform point size
- Run-time configurable scaling schedule for scaled fixed point
- Three architectures offer an exchange between core size and transform time
- For use with Xilinx CORE Generator™ system v6.3i and later

Overview

The FFT core computes an N-point forward DFT or inverse DFT (IDFT) where N can be 2^m, m = 3–16. The input data is a vector of N complex values represented as b_x-bit two's-complement numbers – b_x bits for each of the real and imaginary components of the data sample (b_x = 8,12,16,20,24). Similarly, the phase factors b_w can be 8, 12, 16, 20, or 24 bits wide.

All memory is on-chip using either Block RAM or Distributed RAM. The N element output vector is represented using b_y bits for each of the real and imaginary components of the output data. Input data is presented in natural order, and the output data can be in either natural or bit / digit reversed order.

Three arithmetic options are available for computing the FFT:

- Full-precision unscaled arithmetic
- Scaled fixed-point, where the user provides the scaling schedule
- Block-floating point

Note: For the scaled fixed-point and block-floating point options, superfluous LSBs can be either rounded or truncated after scaling.

Several parameters can be run-time configurable: the point size N, the choice of forward or inverse transform, and the scaling schedule. Both forward / inverse and scaling schedule can be changed frame by frame. Changing the point size resets the core.

Three architecture options are available:

- Pipelined, Streaming I/O. Allows continuous data processing.
- Radix-4, Burst I/O. Offers a load/unload phase and a processing phase; it is smaller in size but has a longer transform time.
- Radix-2, Minimum Resources. Uses a minimum of logic resources and is also a two-phase solution.

For detailed information about each option, see "Architecture Options" on page 4.

Theory of Operation

The FFT is a computationally efficient algorithm for computing a Discrete Fourier Transform (DFT). The DFT $X(k),\ k = 0,\ldots,N-1$ of a sequence $x(n),\ n = 0,\ldots,N-1$ is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-jnk2\pi/N} \quad k = 0,\ldots,N-1$$

Equation 1: **DFT**

where N is the transform size and $j = \sqrt{-1}$. The inverse DFT (IDFT) is

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{jnk2\pi/N} \quad n = 0,\ldots,N-1$$

Equation 2: **IDFT**

Algorithm

The FFT core uses the radix-4 and the radix-2 decomposition for computing the DFT. For two-phase solutions, the decimation-in-time (DIT) method is used, while the decimation-in-frequency (DIF) method is used for the streaming solution. When using radix-4, the FFT consists of $\log_4(N)$ stages, with each stage containing $N/4$ radix-4 butterflies. Point sizes that are not a power of 4 need an extra radix-2 stage for combining data.

An FFT using radix-2 has $\log_2(N)$ stages, with each stage containing $N/2$ radix-2 butterflies.

The inverse FFT (IFFT) is computed by conjugating the phase factors.

Finite Word Length Considerations

The radix-4 and radix-2 FFT algorithms process an array of data by successive passes over the input data array. On each pass, the algorithm performs radix-4 or radix-2 butterflies, where each butterfly picks up four or two complex numbers and returns four or two complex numbers to the same memory. The numbers returned to memory by the processor are potentially larger than the numbers picked up from memory. A strategy must be employed to accommodate this dynamic range expansion. Note that a full explanation of scaling strategies and their implications is beyond the scope of this document; for more information about this topic, see items 3 and 4 in the "References" section on page 36.

For a radix-4 DIT FFT, the values computed in a butterfly stage (except the second) can experience a growth to $4\sqrt{2} \approx 5.657$.

For radix-2, the growth can be up to $1+\sqrt{2} \approx 2.414$. This bit growth can be handled in three ways:

- Performing the calculations with no scaling and carry all significant bits to the end of the computation
- Scaling at each stage using a fixed-scaling schedule
- Scaling automatically using block-floating point

All significant bits are retained when doing full-precision unscaled arithmetic. The width of the data path increases to accommodate the bit growth through the butterfly.

When using scaling, a scaling schedule is used to scale by a factor of 1, 2, 4, or 8 in each stage. If scaling is insufficient, a butterfly output may grow beyond the dynamic range and cause an overflow. As a result of the scaling applied in the FFT implementation, the transform computed is a scaled transform. The scale factor *s* is defined as

$$s = 2^{\sum_{i=0}^{\log N-1} b_i}$$

Equation 3: **Scale Factor**

where b_i is the scaling (specified in bits) applied in stage *i*.

The scaling results in the final output sequence being modified by the factor *1/s*. For the forward FFT, the output sequence
X' (k), k = 0,...,N - 1 computed by the core is defined in Equation 4.

$$X'(k) = \frac{1}{s}X(k) = \frac{1}{s}\sum_{n=0}^{N-1} x(n)e^{-jnk2\pi/N} \qquad k = 0,\ldots,N-1$$

Equation 4: **Scaled FFT**

For the inverse FFT, the output sequence is

$$x(n) = \frac{1}{s}\sum_{k=0}^{N-1} X(k)e^{jnk2\pi/N} \qquad n = 0,\ldots,N-1$$

Equation 5: **Scaled IFFT**

If a radix-4 algorithm uses a scaling schedule of all 2's, the factor of *1/s* will be exactly equal to the factor of *1/N* in the inverse FFT equation (Equation 2). For radix-2, a scaling schedule of all 1's will provide the factor of *1/N*. Otherwise, additional scaling will be needed.

With block floating point, each data point in a frame is scaled by the same amount, and the scaling is kept track of by a block exponent. Scaling is performed only when necessary, which is detected by the core.

Architecture Options

The FFT core provides three architecture options to offer a trade-off between core size and transform time.

- **Pipelined, Streaming I/O**. Allows continuous data processing.
- **Radix-4, Burst I/O**. Offers a load / unload phase and a processing phase; it is smaller in size but has a longer transform time.
- **Radix-2, Minimum Resources**. Uses a minimum of logic resources and is also a two-phase solution.

Pipelined, Streaming I/O

This solution pipelines several radix-2 butterfly processing engines to offer continuous data processing. Each processing engine has its own memory banks to store the input and intermediate data (Figure 1). The core has the ability to simultaneously perform transform calculations on the current frame of data, load input data for the next frame of data, and unload the results of the previous frame of data. The user can stream in input data and, after the calculation latency, can continuously unload the results. If preferred, this design can also calculate one frame by itself or frames with gaps in between.

This architecture supports unscaled full-precision and scaled fixed point arithmetic methods. In the scaled fixed point mode, the data is scaled after every pair of radix-2 stages.

The unloaded output data can either be in bit reversed order or in natural order. By choosing the output data in natural order, additional memory resource will be utilized.

This architecture covers point sizes from 8 to 65536. The user has flexibility to select how many pipelined stages to use block RAM for data and phase factor storage.

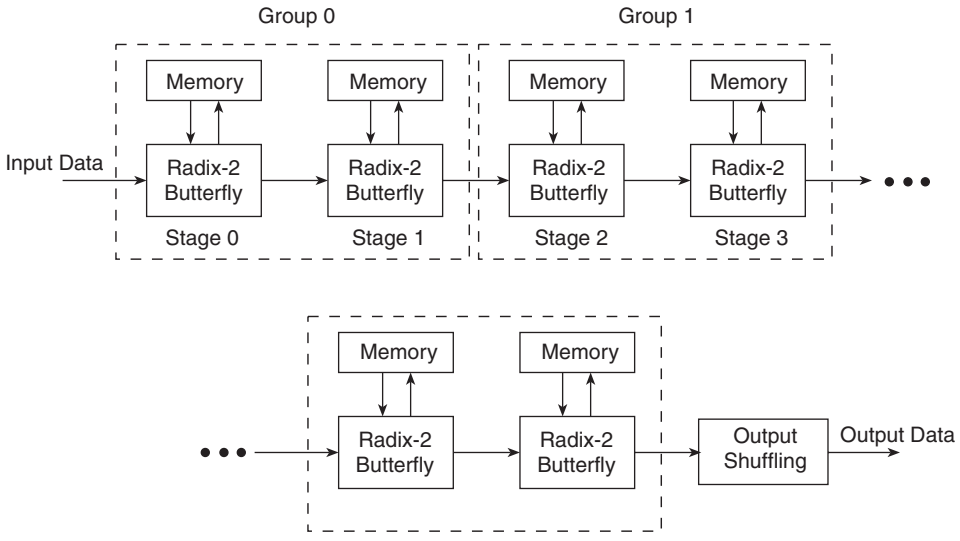


Figure 1: **Pipelined, Streaming I/O**

Radix-4, Burst I/O

With this solution, the FFT core uses one radix-4 butterfly processing engine and has two processes (Figure 2). One process is loading and/or unloading the data. The second process is calculating the transform. Data I/O and processing are not simultaneous. When the FFT is started, the data is loaded in. After a full frame has been loaded, the core will compute the FFT. When the computation has finished, the data can now be unloaded. During the calculation process, data loading and unloading cannot take place. The data loading and unloading processes can be overlapped if the data is unloaded in digit reversed order.

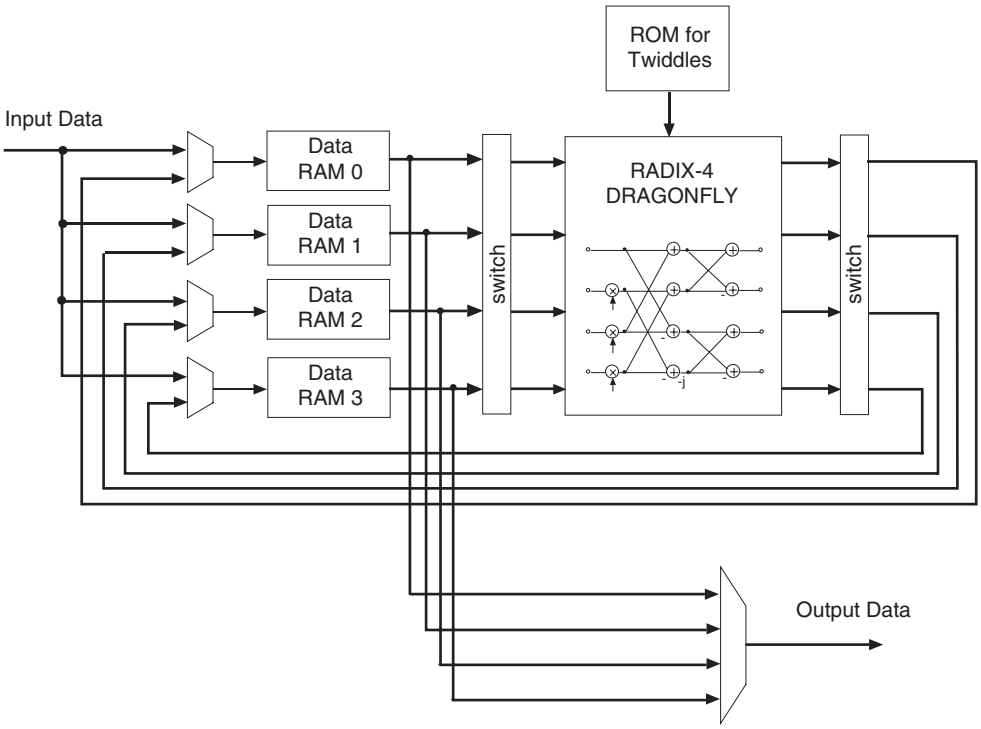


Figure 2: **Radix-4, Burst I/O**

This architecture has less resource usage than the Pipelined Streaming I/O architecture but a longer transform time and covers point sizes from 64 to 65536. All three arithmetic types are supported: unscaled, scaled, and block floating point. Phase factors can be stored in Block RAM or in Distributed RAM (for point sizes less than or equal to 1024).

Radix-2, Minimum Resources

This architecture uses one radix-2 butterfly processing engine (Figure 3) and has burst I/O like the radix-4 version. After a frame of data is loaded, the input data stream must halt until the transform calculation is completed. Then, the data can be unloaded. As with the Radix-4, Burst I/O architecture,

data can be simultaneously loaded and unloaded if the results are presented in bit-reversed order. This solution supports point sizes $N = 8 – 65536$ and uses a minimum of block memories.

All three arithmetic types are supported (unscaled, scaled, and block floating point). Both the data memories and phase factor memories can be in either block memory or distributed memory (for point sizes less than or equal to 1024).

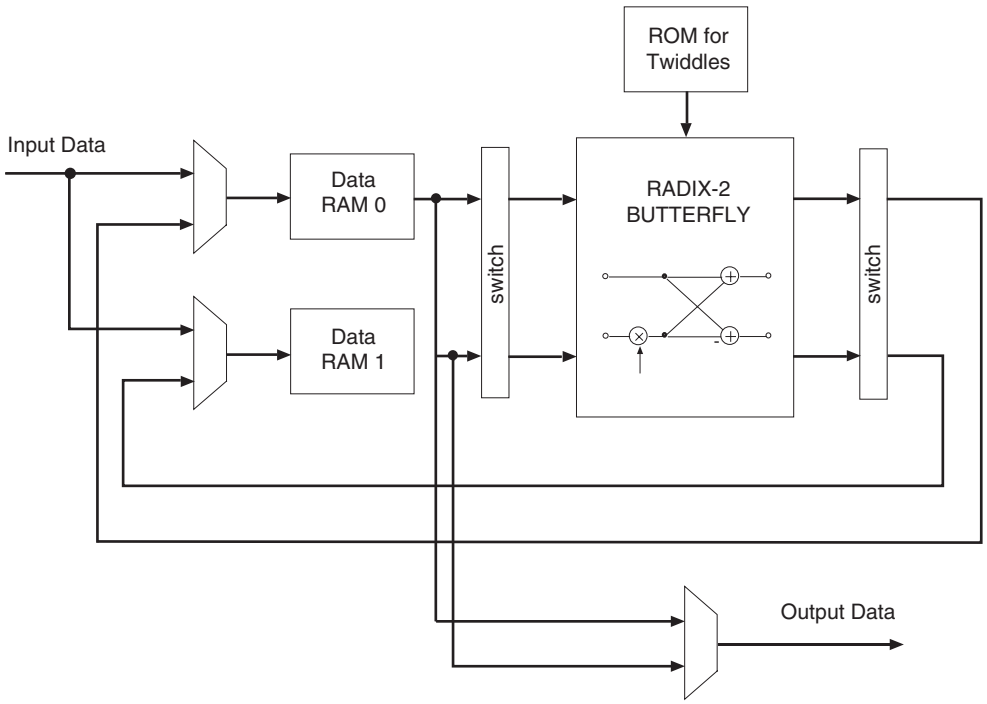


Figure 3: **Radix-2, Minimum Resources**

Core Symbol and Port Definitions

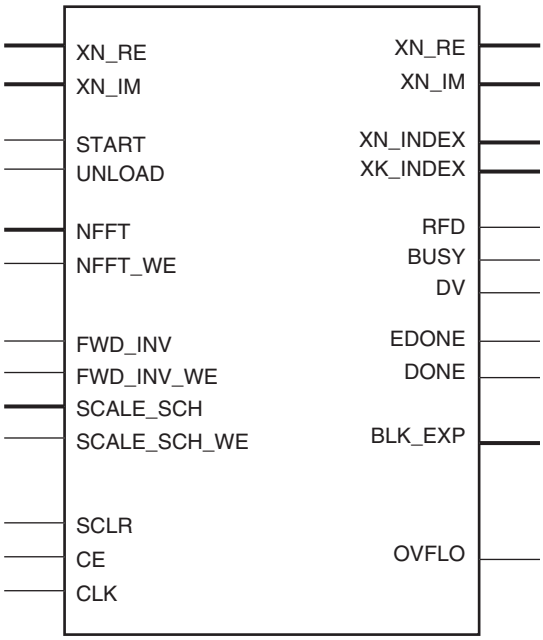


Figure 4: **Core Schematic Symbol**

Table 1: **Core Pinout**

Port Name	Port Width	Direction	Description
XN_RE	b_{xn}	Input	Input data bus: Real component (b_{xn} = 8, 12, 16, 20, 24)
XN_IM	b_{xn}	Input	Input data bus. Imaginary component (b_{xn} = 8, 12, 16, 20, 24)
START	1	Input	FFT start signal (Active High): START is asserted to begin the data loading and transform calculation (for the Burst I/O architectures). For continuous data processing, START will begin data loading, which proceeds directly to transform calculation and then data unloading.
UNLOAD	1	Input	Result unloading (Active High): For the Burst I/O architectures, UNLOAD will start the unloading of the results in normal order. The UNLOAD port is not necessary for the Pipelined, Streaming I/O architecture.
NFFT	5	Input	Point size of the transform: NFFT can be the size of the transform or any smaller point size. For example, a 1024-point FFT can compute point sizes 1024, 512, 256, and so on. The value of NFFT is $\log_2$ (point size). This port is optional for Pipelined, Streaming I/O architecture.

Table 1: **Core Pinout *(Continued)***

Port Name	Port Width	Direction	Description
NFFT_WE	1	Input	Write enable for NFFT (Active High): Asserting NFFT_WE will automatically cause the FFT core to stop all processes and to initialize the state of the core. This port is optional for Pipelined, Streaming I/O architecture.
FWD_INV	1	Input	Control signal that indicates if a forward FFT or an inverse FFT is performed. When FWD_INV=1, a forward transform is computed. If FWD_INV=0, an inverse transform is performed.
FWD_INV_WE	1	Input	Write enable for FWD_INV (Active High).
SCALE_SCH	$2 \times ceil\left(\frac{NFFT}{2}\right)$ for Streaming I/O and Radix-4 Burst I/O architectures or 2 x $NFFT$ for Radix-2 Minimum Resources	Input	Scaling schedule: For Radix-4 Burst I/O and Radix-2 minimum resources architecture, the scaling schedule is specified with two bits for each stage. The scaling can be specified as 3, 2, 1, or 0, which represents the number of bits to be shifted. An example scaling schedule for N =1024, Radix-4 burst I/O is [1 0 2 3 2]. For N=128, Radix-2 minimum resources, one possible scaling schedule is [1 1 1 1 0 1 2]. For Pipelined Streaming I/O architecture, the scaling schedule is specified with two bits for every pair of radix-2 stages. For example, a scaling schedule for N=256 could be [2 2 2 3]. When N is not a power of 4, the maximum bit growth for the last stage is one bit. For instance, [0 2 2 2 2] or [1 2 2 2 2] are valid scaling schedules for N=512, but [2 2 2 2 2] is invalid. The two MSBs of SCALE_SCH can only be 00 or 01. This port is only available with scaled arithmetic (not unscaled or block-floating point).
SCALE_SCH_WE	1	Input	Write enable for SCALE_SCH (Active High): This port is available only with scaled arithmetic.
SCLR	1	Input	Master reset (Active High): Optional port.
CE	1	Input	Clock enable (Active High): Optional port.
CLK	1	Input	Clock
XK_RE[(B-1):0]	b_{xk}	Output	Output data bus: Real component.
XK_IM[(B-1):0]	b_{xk}	Output	Output data bus: Imaginary component.
XN_INDEX	$\log_2$ (point size)	Output	Index of input data.
XK_INDEX	$\log_2$ (point size)	Output	Index of output data.
RFD	1	Output	Ready for data (Active High): RFD is High during the load operation.
BUSY	1	Output	Core activity indicator (Active High): This signal will go High while the core is computing the transform.
DV	1	Output	Data valid (Active High): This signal is High when valid data is presented at the output.

Table 1: **Core Pinout *(Continued)***

Port Name	Port Width	Direction	Description
EDONE	1	Output	Early done strobe (Active High): EDONE goes High one clock cycle immediately prior to DONE going active.
DONE	1	Output	FFT complete strobe (Active High): DONE will transition High for one clock cycle at the end of the transform calculation.
BLK_EXP	5	Output	Block exponent: Available only when block-floating point is used.
OVFLO	1	Output	Arithmetic overflow indicator (Active High): OVFLO will be High during result unloading if any value in the data frame overflowed. The OVFLO signal is reset at the beginning of a new frame of data. This port is optional and only available with scaled arithmetic.

Graphical User Interface

The FFT core graphical user interface (GUI) consists of three screens. Options for each screen are described below.

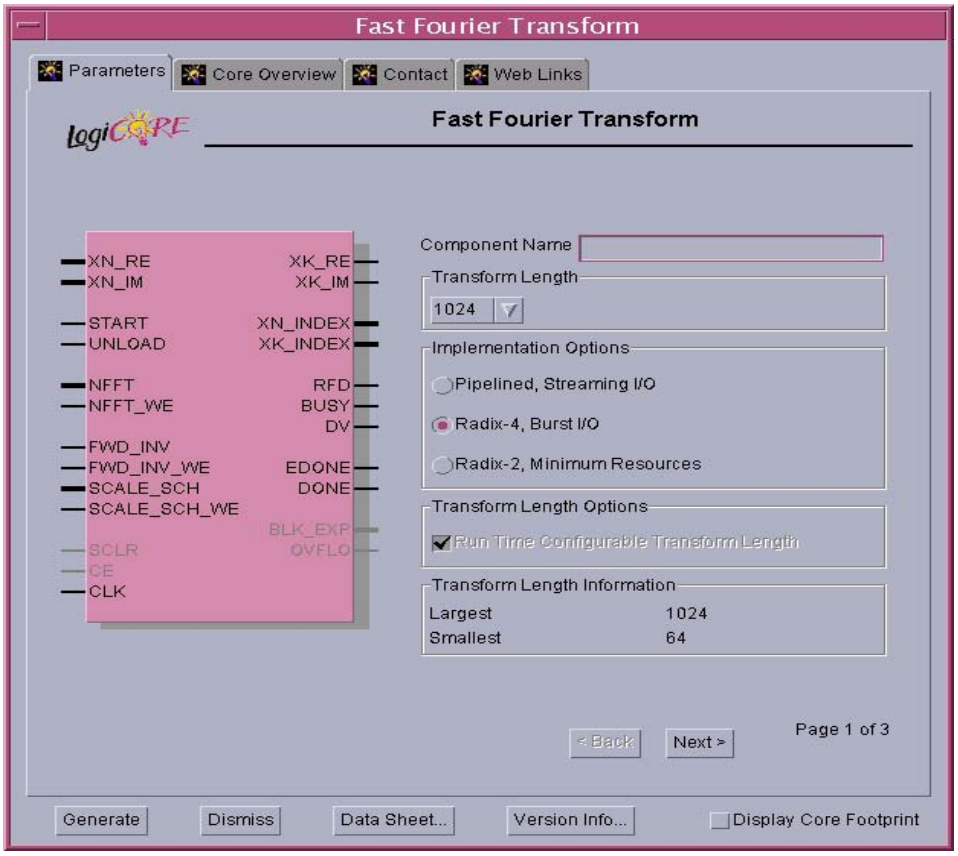


Figure 5: **XFFT Core GUI Main Window**

- **Component Name**: The name of the core component to be instantiated. The name must begin with a letter and be composed of the following characters: a to z, 0 to 9, and "_".
- **Transform Length**: Select the desired point size. All powers of two from 8 to 65536 are available.
- **Implementation Options**: Select an implementation option, as described in "Architecture Options" on page 4.
  - Radix-4, Streaming I/O, and Radix-2 Minimum Resources support point sizes 8 to 65536.
  - Radix-4 Burst I/O architecture supports point sizes 64 to 65536.
- **Transform Length Option**: Available only for the Radix-4, Streaming I/O architecture. Select the transform length to be run-time configurable or not. The core uses fewer logic resources when the transform length is not run-time configurable.
- **Transform Length Information**: When the transform length is run-time configurable, the core has the ability to reprogram the point size while the core is running; that is, the core can support the selected point size and any smaller point size. The GUI displays the supported point sizes based on

the Transform Length, Transform Length Option, and Implementation Option selected.

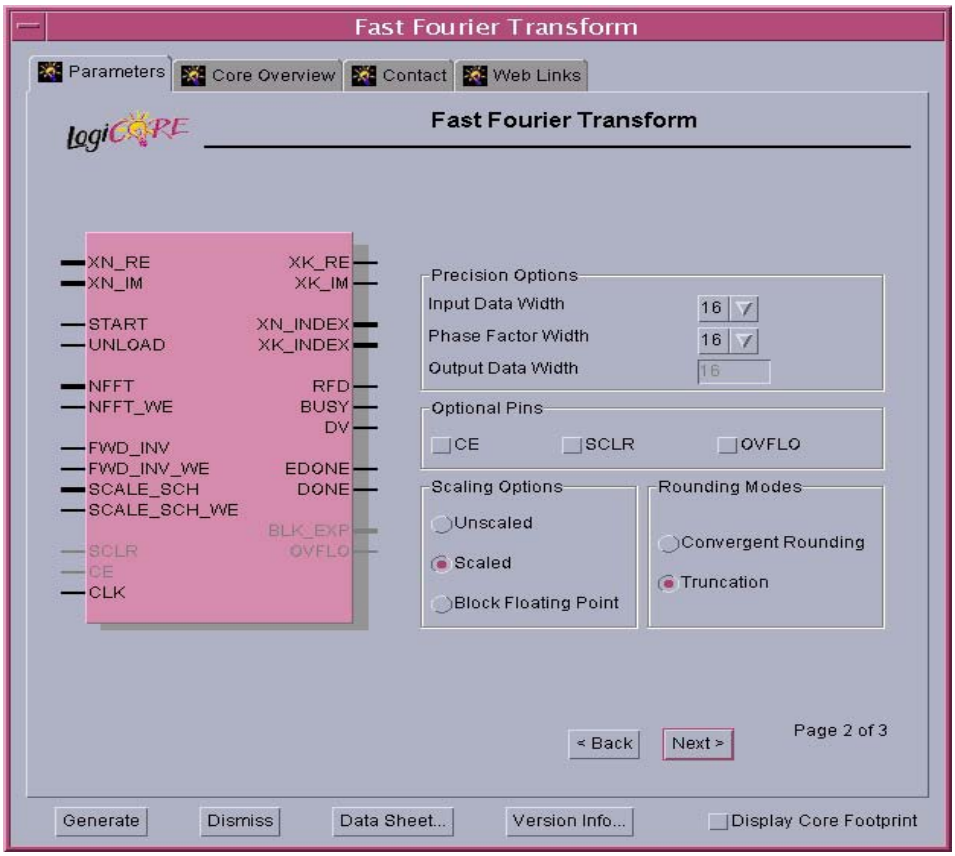


Figure 6: **XFFT Core GUI Precision and Scaling Option Window**

- **Precision Options**
  - Input data width and phase factor data width can be 8, 12, 16, 20, or 24.
  - Output data width (which may be affected by the Scaling Option) is also displayed.
- **Optional Pins**
  - Clock Enable (CE), Synchronous Clear (SCLR), and Overflow (OVFLO) are optional pins. If no option is selected, some logic resources are saved.
- **Scaling Options**
  - Unscaled
  - Scaled
  - Block Floating Point. Note that Block Floating Point is unavailable with the Pipelined Streaming I/O architecture.
- **Rounding Modes**: At the output of the butterfly, the LSBs in the datapath need to be trimmed. These bits can be truncated or rounded using convergent rounding, an unbiased rounding scheme also known as round-to-nearest (even) number. When the fractional part of a number is equal to exactly one-half, convergent rounding rounds down if the number is odd (resulting in LSB=0), and rounds up if the number is even (resulting in LSB=1). Convergent rounding can be used to avoid

the DC bias that would be introduced by truncation.

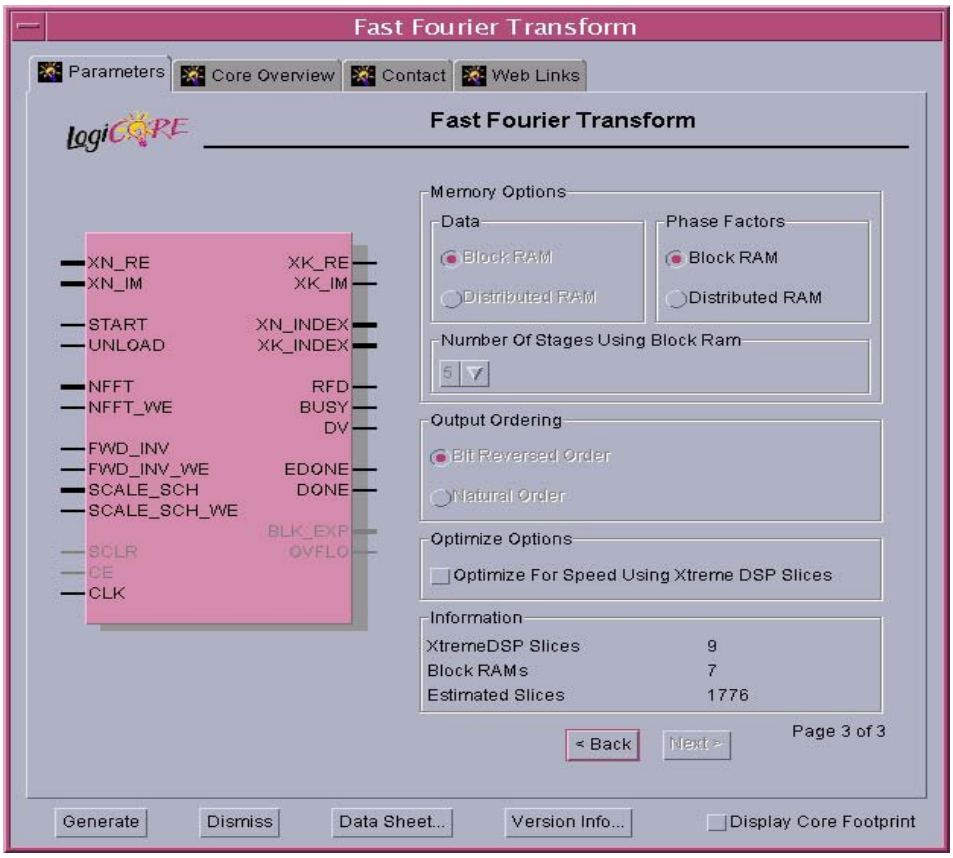


Figure 7: **XFFT Core GUI Memory Options Window**

- **Memory Options**
  - For Pipelined Streaming I/O solution, the data can be partially stored in Block RAM and partially in Distributed RAM. The user can select the number of pipelined stages using Block RAM for data and phase factor storage.
  - For Radix-4 and Radix-2 burst I/O architecture, either Block RAM or Distributed RAM can be used for data and phase factor storage. Data storage in Block RAM is always available, and Distributed RAM data storage is supported by the Radix-2 Minimum Resource architecture. Phase factor storage is also always available in Block RAM. Phase factor storage can be in distributed RAM for all point sizes 1024 or under.
- **Output Ordering**: This output data can either be in bit reversed order or in natural order, and is available only for Pipelined Streaming I/O architecture.
- **Optimize Options**
  - For Radix-4 Burst I/O architecture in Virtex-4, the entire dragonfly can be computed in XtremeDSP slices. Selecting *Optimize For Speed Using XtremeDSP Slices* will allow a faster maximum clock speed at the cost of using more XtremeDSP slices. This checkbox is only available when the CORE Generator target architecture is Virtex-4.
- **Information**: Based on the options selected, this area displays the XtremeDSP slice count/embedded multiplier usage, block RAM numbers, and estimated slice count.

XCO Parameters

The following table describes valid entries for the xco parameters. Note that parameters are not case sensitive.

Table 2: **XCO Parameters**

XCO Parameter	Valid Values
rounding_modes	convergent_rounding truncation
ce	false true
scaling_options	scaled unscaled block_floating_point
memory_options_phase_factors	block_ram distributed_ram
output_data_width	If scaling_option is scaled or block_floating_point, then: output_data_width = input_data_width If scaling option is unscaled, then: output_data_width = input_data_width + $\log_2$ (transform_length) + 1
input_width	8, 12, 16, 20, 24
transform_length	8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536
run_time_configurable_transform_length	false true
memory_options_data	block_ram distributed_ram
number_of_stages_using_block_ram_for_data_and_phase_factors	0 - 12
output_ordering	bit_reversed_order natural_order
implementation_options	radix_4_burst_io pipelined_streaming_io radix_2_minimum_resources
component_name	Name must begin with a letter and be composed of the following characters: a to z, 0 to 9, and "_".
phase_factor_width	8, 12, 16, 20, 24
sclr	false true
ovflo	false true
optimize_for_speed_using_xtreme_dsp_slices	false true

Control Signals and Timing

Synchronous Clear

Asserting the Synchronous Clear (SCLR) pin results in resetting all output pins, internal counters, and state variables to their initial values. All pending load processes, transform calculations, and unload processes stop and are reinitialized. However, internal frame buffers retain their contents. NFFT will be set to the largest FFT point size permitted (the Transform Length value set in the GUI). The scaling schedule will be set to *1/N*. For the Radix-4 Burst I/O and Pipelined Streaming I/O architectures with a non-power-of-four point size, the last stage will have a scaling of 1, and the rest will have a scaling of 2.

Table 3: **Synchronous Clear Reset Values**

Signal	Initial / Reset Value
NFFT	maximum point size = *N*
FWD_INV	Forward = 1
SCALE_SCH	1/*N* [10 10... 10] for Radix-4 or Pipelined architecture when *N* is a power of 4. [01 10... 10] for Radix-4 or Pipelined architecture when *N* is not a power of 4. [01 01... 01] for Radix-2

Transform Size

The transform point size can be set through the NFFT port if the run-time configurable transform length option is selected. Valid settings and the corresponding transform sizes are provided in Table 4. If the NFFT value entered is too large, the core sets itself to the largest available point size (selected in the GUI). If the value is too small, the core sets itself to the smallest available point size: 64 for the Radix-4 Burst I/O architectures and 8 for the Radix-2 Minimum Resources and Pipelined Streaming I/O architecture.

NFFT values are read in on the rising clock edge when NFFT_WE is High. A new transform size re-times all current processes within the core, so every time a transform size is latched in, regardless of whether or not the new point size differs from the current point size, the core is internally reset. (Note that FWD_INV and SCALE_SCH are not reset.) Holding NFFT_WE High continues to reset the core on every clock cycle.

Table 4: **Valid NFFT Settings**

NFFT[4:0]	Transform size (*N*)
00011	8
00100	16
00101	32
00110	64
00111	128
01000	256
01001	512

Table 4: **Valid NFFT Settings *(Continued)***

NFFT[4:0]	Transform size (*N*)
01010	1024
01011	2048
01100	4096
01101	8192
01110	16384
01111	32768
10000	65536

Forward/Inverse and Scaling Schedule

The transform type (forward or inverse) and the scaling schedule can be set frame-by-frame without interrupting frame processing. The transform type can be set using the FWD_INV pin. Setting FWD_INV to 0 produces an inverse FFT, and setting FWD_INV to 1 creates the forward transform.

The scaling performed during successive stages can be set via the SCALE_SCH pin. The value of the SCALE_SCH bus is used as pairs of bits [... N4, N3, N2, N1, N0]: each pair representing the scaling value for the corresponding stage (except for Pipelined Streaming I/O architecture). In each stage, the data can be shifted by 0, 1, 2, or 3 bits, which corresponds to SCALE_SCH values of 00, 01, 10, and 11. Stages are computed starting with stage 0 as the two LSBs. For example, when $N = 1024$, [11 10 00 01 10] translates to a right shift by 2 for stage 0, shift by 1 for stage 1, no shift for stage 3, a shift of 2 in stage 3, and a shift of 3 for stage 4. The conservative schedule SCALE_SCH = [10 10 10 11 10] will completely avoid overflows in the Radix-4 architecture. For the Radix-2 architecture, the conservative scaling schedule of [01 01 01 10 01] will prevent overflow.

For the pipelined streaming architecture, consider every pair of adjacent radix-2 stages as a group. That is, group 0 contains stage 0 and 1, group 1 contains stage 2 and 3, and so forth. The value of the SCALE_SCH bus is also used as pairs of bits [... N4, N3, N2, N1, N0]. Each pair represents the scaling value for the corresponding group of two stages. In each group, the data can be shifted by 0, 1, 2, or 3 bits which corresponds to SCALE_SCH values of 00, 01, 10, and 11. Groups are computed starting with group 0 as the two LSBs. For example, when $N = 1024$, [11 10 00 01 10] translates to a right shift by 2 for group 0, shift by 1 for group 1, no shift for group 3, a shift of 2 in group 3, and a shift of 3 for group 4. The conservative schedule SCALE_SCH = [10 10 10 10 11] will completely avoid overflows in the Pipelined Streaming architecture. Note that when N is not a power of 4, the last group only contains one stage, and the maximum bit growth for the last group is one bit. Therefore, the two MSBs of scaling schedule can only be 00 or 01. A conservative scaling schedule for N=512 is SCALE_SCH=[01 10 10 10 11]

The user is allowed great flexibility to set the transform type (Forward/Inverse) and the scaling schedule. The FWD_INV and SCALE_SCH values are latched into temporary registers whenever the corresponding WE pins are High. FWD_INV_WE and SCALE_SCH_WE can be asserted at any time before the frame of data is loaded in. The core will read these temporary registers at XN_RE/XN_IM(0). These are the values that will be used for that frame of data. There is no way to alter those values once the transform calculation phase has started. Any WE assertions after XN_RE/XN_IM(0) affect the frame that follows.

Both the scaling schedule and the transform type are registered internally, so there is no need to hold these values on the pins. Also, if the scaling and transform type are constant through multiple frames, (that is, no new values are latched in) registered values will apply for successive frames. The scaling schedule and transform type are not reset when NFFT_WE is asserted.

The initial value and reset value of FWD_INV is forward = 1. The scaling schedule is set to *1/N*. That translates to [10 10 10 10... 10] for the Radix-4 architectures, and [01 01...01] for the Radix-2 architecture. The core will read in (2*number of stages) bits for the scaling schedule. So, when the point size decreases, the leftover MSBs will be ignored.

Overflow

The Overflow (OVFLO) signal (used only with fixed-point scaling) will be High during unloading if any point in the data frame overflowed. For the Radix-4 Burst I/O and Radix-2 Minimum Resources architectures, The OVFLO signal will go High as soon as an overflow occurs during the computation and remain High during the entire time the frame is unloading.

Block Exponent

The Block Exponent (BLK_EXP) signal (used only with the block-floating point option) contains the block exponent. This signal will be valid during the unloading of the data frame. The value present on the port represents the total number of bits the data was scaled during the transform. For example, if BLK_EXP has a value of 00101b = 5, this means the output data (XK_RE, XK_IM) was scaled by 5 bits (shifted right by 5 bits), or in other words, was divided by 32, in order to fully utilize the available dynamic range of the output data path.

Timing for Pipelined Streaming I/O

Asserting START starts the data loading phase, which will immediately flow into the transform calculation phase and then the data unloading phase. Pulsing START once will allow the transform calculation for a single frame. Pulsing START every N clock cycles will allow continuous data processing. Alternatively, holding START High will also allow continuous data processing (Figure 8). If START is pulsed at any time during the loading of the current frame (when XN_INDEX = 0 to N-1), it signals to the core that another frame needs to be loaded after the current one. If no NFFT_WE, FWD_INV_WE, or SCALE_SCH_WE were asserted before the initial START, then the defaults will be used. This architecture can also support non-continuous data streams (Figure 9). Simply assert START at any time to begin data loading. After the data frame is loaded, the core will proceed to calculate the transform and then output the results.

Input data (XN_RE, XN_IM) corresponding to a certain XN_INDEX should arrive four clock cycles later than the XN_INDEX it matches (Figure 10). In this way, XN_INDEX can be used to address external memory or a frame buffer storing the input data. RFD will remain High with XN_INDEX during the loading phase when it is valid to input data.

BUSY will go High while the core is calculating the transform. DONE will go High during the last cycle of the calculation phase. EDONE will go High one cycle before that. The cycle after DONE goes High, the core begins unloading. During the unloading phase, while valid output results are present on XK_RE/ XK_IM, DV (Data Valid) will be High. During unloading, XK_INDEX will correspond to the XK_RE/XK_IM being presented.

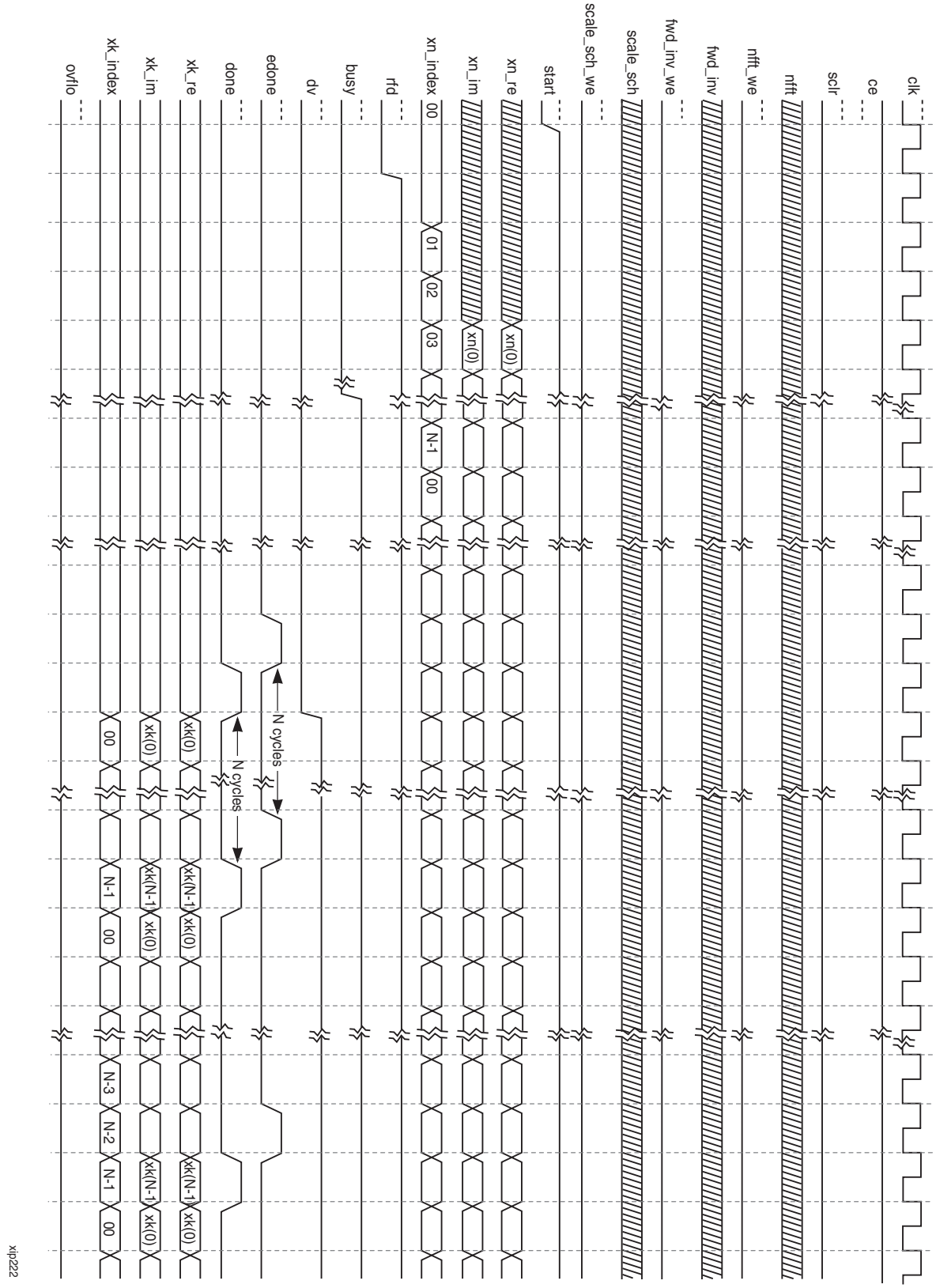
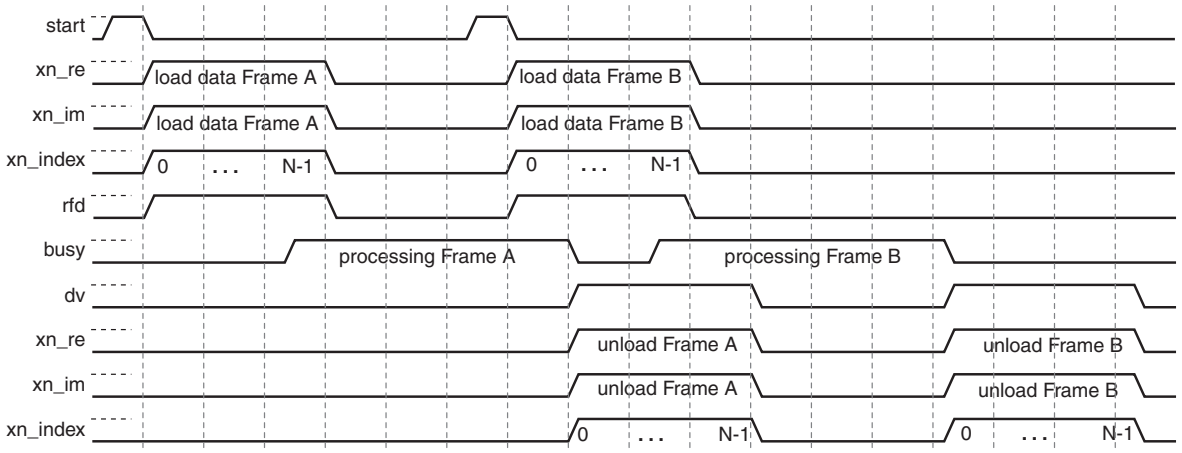


Figure 8: **Timing for Continuous Streaming Data**



Note:
All transitions are synchronous with the rising edge of the clock.

Figure 9: **Timing for Non-Continuous Data Stream**

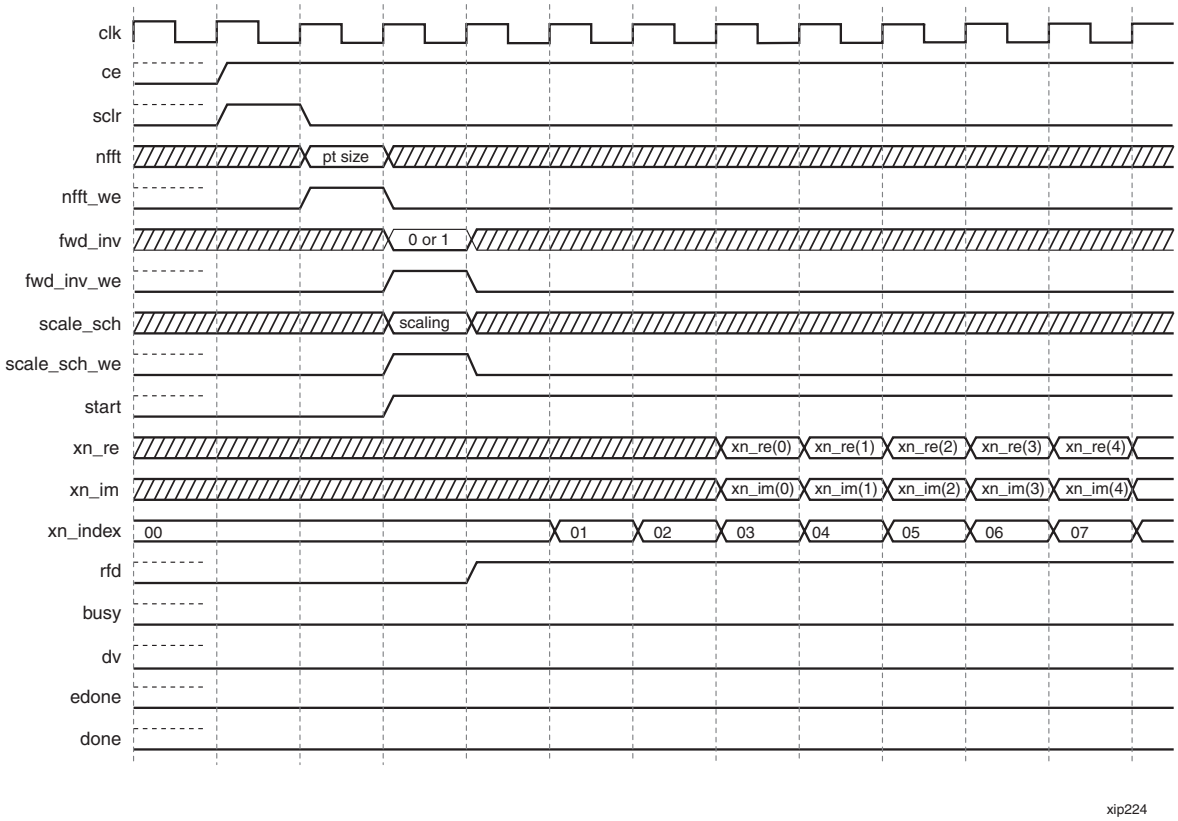


Figure 10: **Beginning of Data Frame**

Timing for Radix-4 Burst I/O and Radix-2 Minimum Resources

The START signal begins the data loading phase, which leads directly to the calculation phase. As with the Pipeline Streaming I/O architecture, START being pulsed at any time during the loading of the current frame (when XN_INDEX = 0 to N-1) signals the core that another frame will be processed after the current one.

Input data (XN_RE, XN_IM) corresponding to a certain XN_INDEX should arrive four clock cycles later than the XN_INDEX it matches (Figure 10). In this way, XN_INDEX can be used to address external memory or a frame buffer storing the input data. RFD will remain High with XN_INDEX during the loading phase when it is valid to input data.

BUSY will go High while the core is calculating the transform. DONE will go High during the last cycle of the calculation phase. EDONE will go High one cycle before that.

After START is asserted and the data is loaded and processed, two options are available to unload data:

- To output the data in natural order, UNLOAD should be asserted (Figure 11).
- To output data in bit/digit reversed order, the user can assert START again. While the next frame of data is loaded, the results will be presented in bit/digit reversed order at the same time (Figure 12).

DV remains High during data unloading in both cases.

START and UNLOAD can be tied High (Figure 13). In this case, the core will alternate between load/calculate and unload. The core will continuously load, process, and unload data.

There is a latency of *k* CLK cycles after triggering an unload before the output data XK_RE/XK_IM is presented (*k* = 7 for Radix-4 Burst I/O and *k* = 5 for Radix-2 Minimum Resources). If the unload process is triggered by pulsing UNLOAD after transform calculations are completed and DONE has gone High, then there is a delay of *k* clock cycles after UNLOAD before XK_RE(0) and XK_IM(0) appear on their ports. If UNLOAD is pulsed during transform calculation before DONE has gone High, then XK_RE and XK_IM will appear *k* clock cycles after DONE has gone High which means the calculation is complete. If bit/digit reversed unloading is triggered by pulsing START, then XK_RE and XK_IM will appear *k* clock cycles after START (Figure 14).

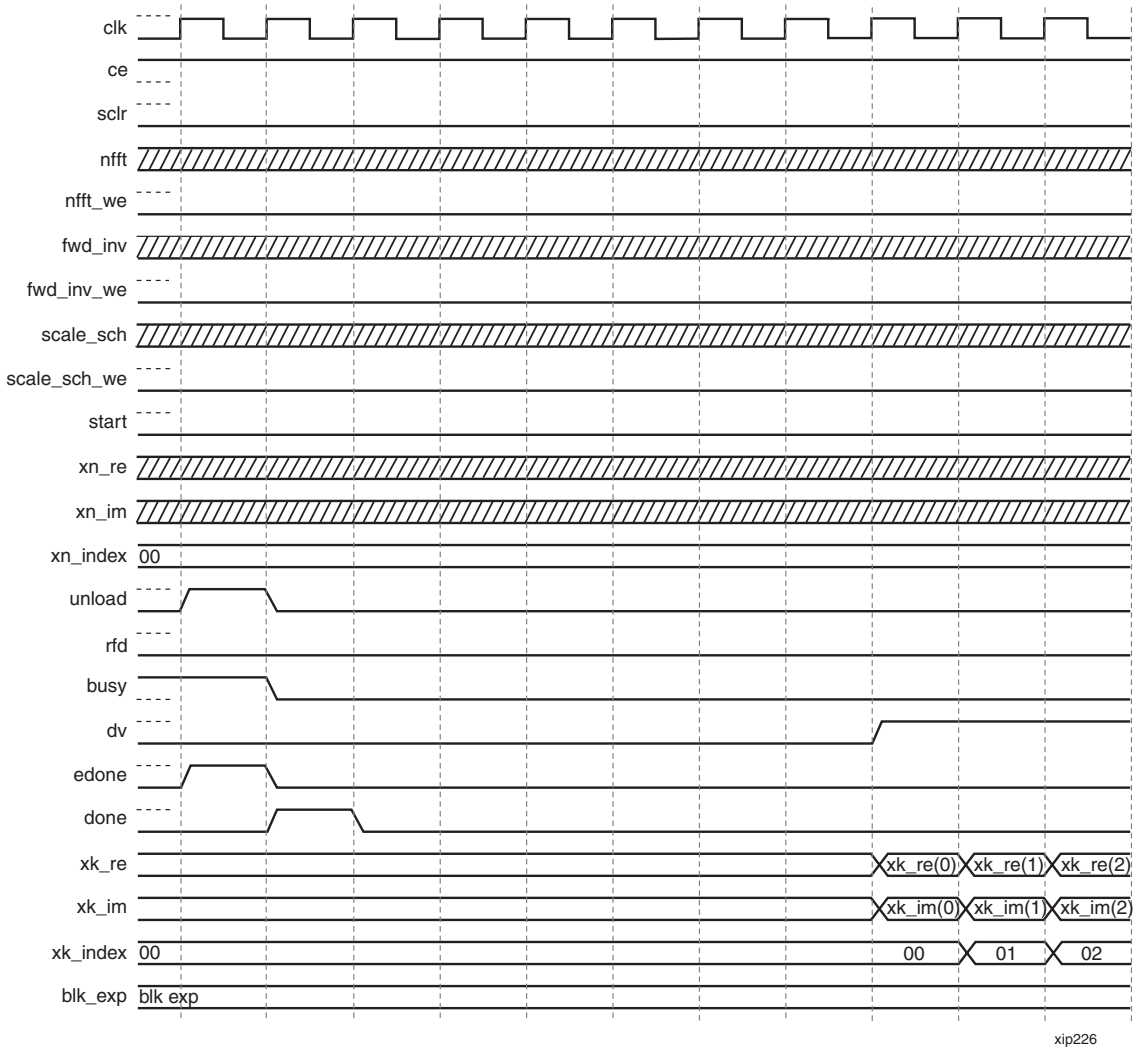


Figure 11: **Unload Output Results in Natural Order**

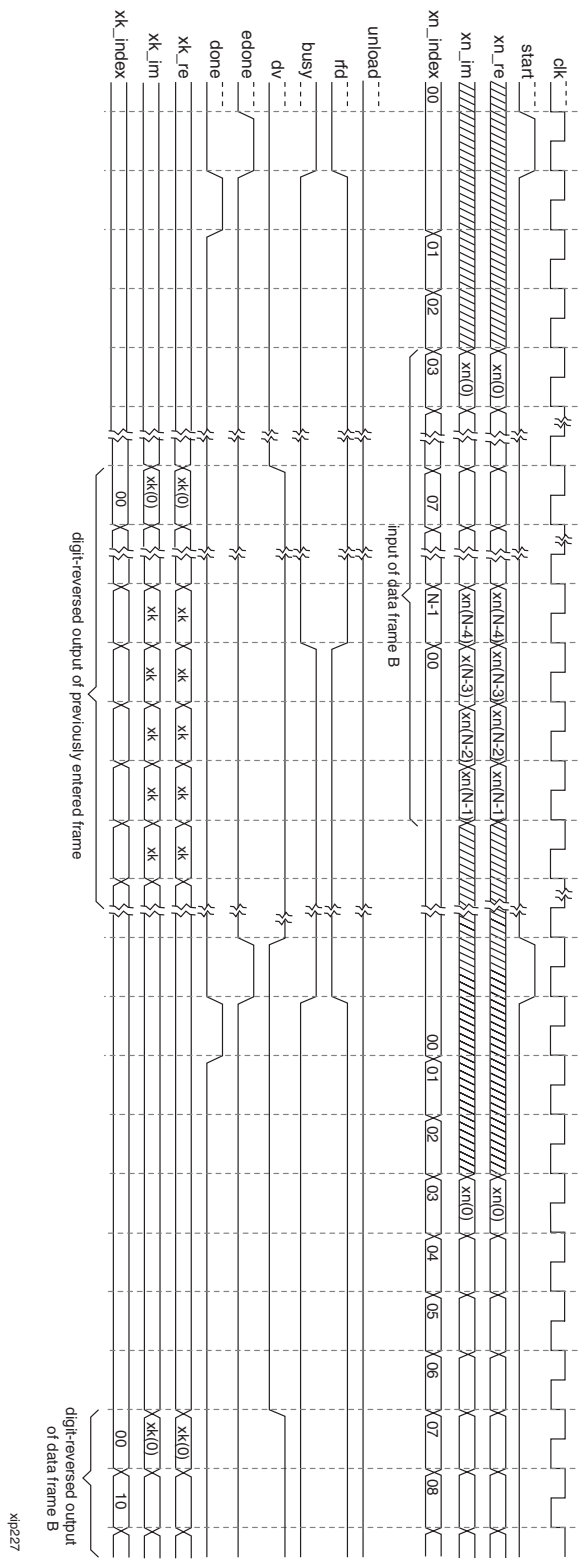


Figure 12: **Unload Results in Bit/Digit Reversed Order**

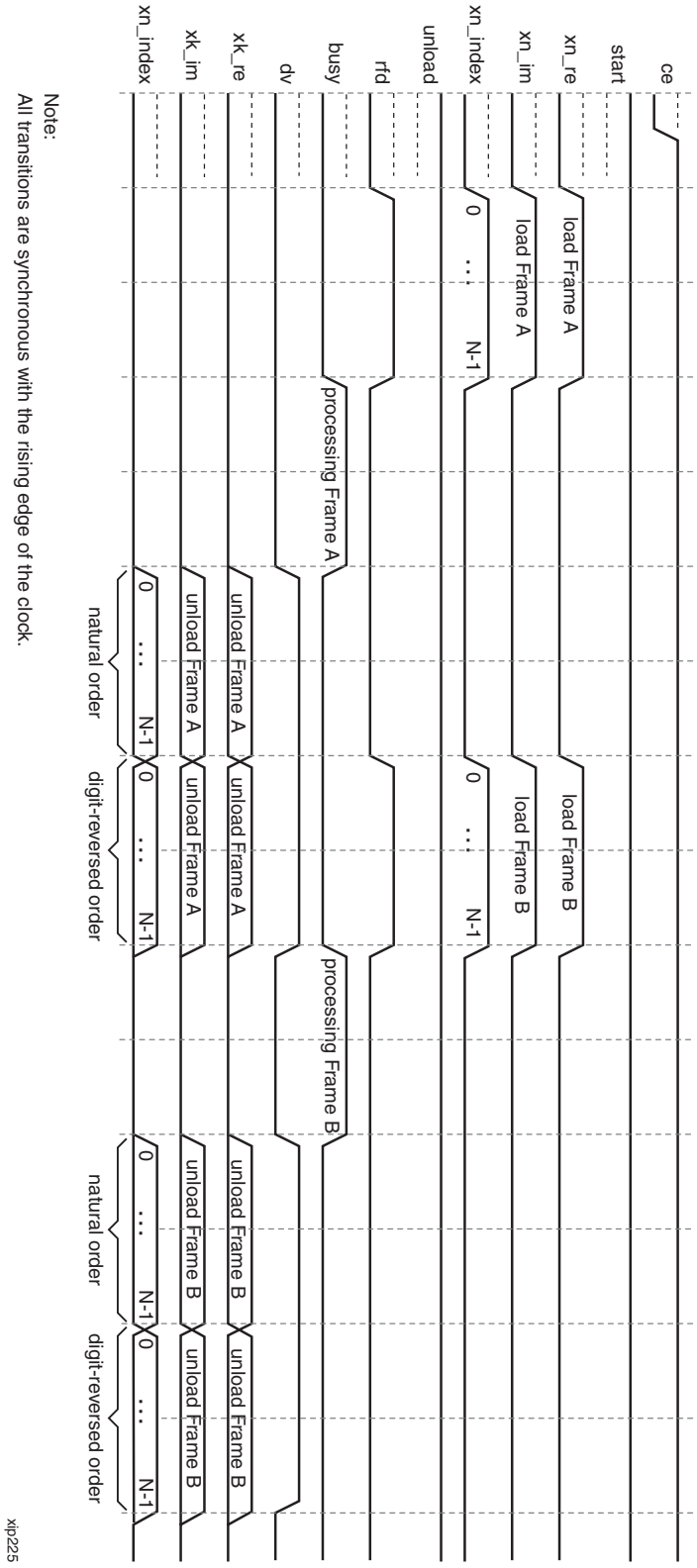


Figure 13: **Timing for Burst I/O Solutions**

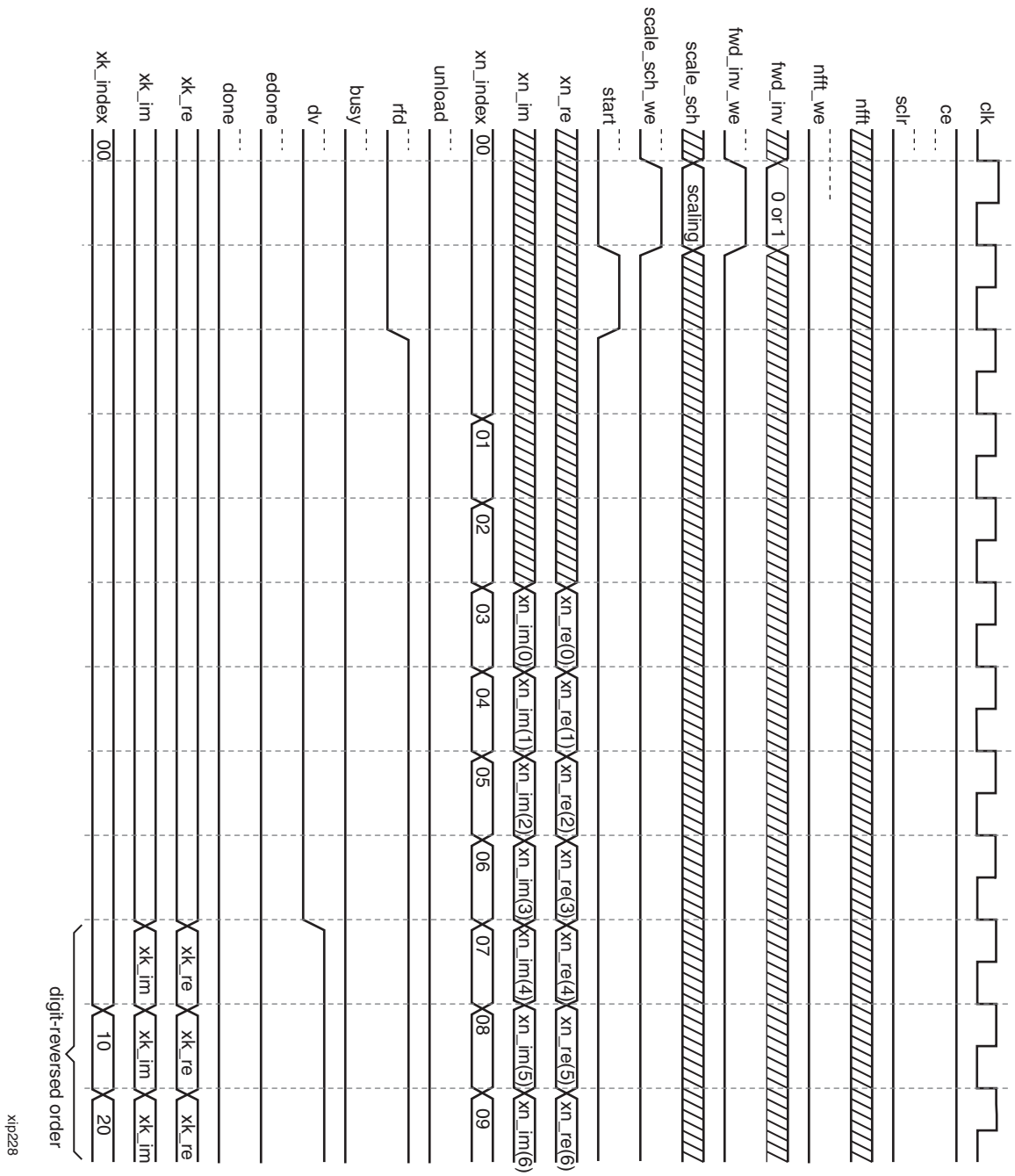


Figure 14: **Unloading Results in Bit/Digit Reversed Order**

Performance and Resource Usage

The following tables list the resource usage and transform time for a selected set of parameters. For a variety of point sizes and input data / phase factor bit widths, the slice count, Block RAM count, and XtremeDSP slice / embedded hardware multiplier count is listed. Also, the maximum clock frequency is listed next to the transform time. The transform time is described by the number of clock cycles or number of microseconds needed to perform the transform calculation. (Time for loading and unloading the data is not included in that value.) For the non-streaming I / O architectures, an additional column is provided, which contains the number of clock cycles and microseconds needed to load and calculate the transform.

These numbers were obtained using scaled fixed-point arithmetic with truncation after the butterflies. Data and phase factor storage is in Block RAM for the burst I / O architectures. For the streaming I / O architecture, the default value in the GUI is used for distributing data and phase factor storage between Block RAM and Distributed RAM. The slice counts are approximately the same if using block floating point arithmetic.

Table 5: **Performance and Resource Utilization for the Virtex-II Family: Pipelined, Streaming I/O**

Point Size	Input Data Width	Phase Factor Width	Slices	Block RAM	MULT 18x18	Max. Clock Frequency (MHz)		Transform Time			Device
								Clock Cycles	Time (µs)		
						-6	-5		-6	-5	
64	16	16	1280	1	8	188	169	64	0.34	0.38	2v250
256	16	16	1769	4	12	188	143	256	1.36	1.79	2v1000
1024	16	16	2294	7	16	188	169	1024	5.45	6.06	2v1500
2048	16	16	2545	10	20	188	143	2048	10.89	14.32	2v3000
8192	16	16	3141	24	24	188	143	8192	43.57	57.29	2v4000
65536	16	16	4285	150	28	n/a	169	65536	n/a	387.79	2v8000
64	24	24	2463	2	24	148	134	64	0.43	0.48	2v2000
1024	24	24	4526	12	48	148	134	1024	6.92	7.64	2v6000
8192	24	24	6252	37	72	148	134	8192	55.35	61.13	2v6000

Note: ISE 6.3i speed file - Production 1.118 2004-08-11.

Table 6: **Performance and Resource Utilization for the Virtex-II Family: Radix-4, Burst I/O**

Point Size	Input Data Width	Phase Factor Width	Slices	Block RAM	MULT 18x18	Max. Clock Frequency (MHz)		Transform Time			Data Load + Transform Time			Device
								Clock Cycles	Time (µs)		Clock Cycles	Time (µs)		
						-6	-5		-6	-5		-6	-5	
64	16	16	1331	8	9	195	149	91	0.47	0.61	155	0.79	1.04	2v1000
256	16	16	1411	7	9	195	149	289	1.48	1.94	545	2.79	3.66	2v1000
1024	16	16	1498	7	9	195	149	1319	6.76	8.85	2343	12.02	15.72	2v1000
2048	16	16	1649	7	9	195	149	3117	15.98	20.92	5165	26.49	34.66	2v1000
8192	16	16	1753	22	9	195	149	14387	73.78	96.56	22579	115.79	151.54	2v3000
65536	16	16	2167	158	9	n/a	175	131129	n/a	749.31	196665	n/a	1123.80	2v8000
64	24	24	2669	12	36	152	138	97	0.64	0.70	161	1.06	1.17	2v3000

Table 6: **Performance and Resource Utilization for the Virtex-II Family: Radix-4, Burst I/O *(Continued)***

Point Size	Input Data Width	Phase Factor Width	Slices	Block RAM	MULT 18x18	Max. Clock Frequency (MHz)		Transform Time			Data Load + Transform Time			Device
								Clock Cycles	Time (µs)		Clock Cycles	Time (µs)		
						-6	-5		-6	-5		-6	-5	
1024	24	24	2968	11	36	152	138	1321	8.69	9.57	2345	15.43	16.99	2v3000
8192	24	24	3107	33	36	152	138	14389	94.66	104.27	22581	148.56	163.63	2v6000

Note: ISE 6.3i speed file - Production 1.118 2004-08-11.

Table 7: **Performance and Resource Utilization for the Virtex-II Family: Radix-2, Minimum Resources**

Point Size	Input Data Width	Phase Factor Width	Slices[2]	Block RAM [2]	MULT 18x18	Max. Clock Frequency (MHz)		Transform Time			Data Load + Transform Time			Device
								Clock Cycles	Time (µs)		Clock Cycle	Time (µs)		
						-6	-5		-6	-5		-6	-5	
64	16	16	530/709	3/0	3	188	169	265	1.41	1.57	329	1.75	1.95	2v250
256	16	16	567/1365	3/0	3	192	166	1079	5.62	6.50	1335	6.95	8.04	2v250
1024	16	16	630/3053	3/0	3	176	150	5187	29.47	34.58	6211	35.29	41.41	2v250
2048	16	16	717	5	3	179	156	11338	63.34	72.68	13386	74.78	85.81	2v250
8192	16	16	808	18	3	166	142	53334	321.29	375.59	61526	370.64	433.28	2v1500
65536	16	16	1062	130	3	168	147	524497	3122.01	3568.01	590033	3512.10	4013.83	2v6000
64	24	24	973	5	12	152	138	277	1.82	2.01	341	2.24	2.47	2v1000
1024	24	24	1175	5	12	152	138	5190	34.14	37.61	6214	40.88	45.03	2v1000
8192	24	24	1276	25	12	152	138	53336	350.89	386.49	61528	404.79	445.86	2v3000

Notes:
1. ISE 6.3i speed file - Production 1.118 2004-08-11.
2. Second number is if input data and phase factors are stored in distributed memory.

Table 8: **Performance and Resource Utilization for the Virtex-II Pro Family: Pipelined, Streaming I/O**

Point Size	Input Data Width	Phase Factor Width	Slices	Block RAM	MULT 18x18	Max. Clock Frequency (MHz)		Transform Time			Device
								Clock Cycles	Time (µs)		
						-7	-6		-7	-6	
64	16	16	1279	1	8	214	198	64	0.30	0.32	2vp4
256	16	16	1768	4	12	214	198	256	1.20	1.29	2vp7
1024	16	16	2284	7	16	214	198	1024	4.79	5.17	2vp20
2048	16	16	2545	10	20	214	198	2048	9.57	10.34	2vp20
8192	16	16	3120	24	24	214	198	8192	38.28	41.37	2vp30
65536	16	16	4252	150	28	214	198	65536	306.24	330.99	2vp70
64	24	24	2462	2	24	165	153	64	0.39	0.42	2vp20

Table 8: **Performance and Resource Utilization for the Virtex-II Pro Family: Pipelined, Streaming I/O *(Continued)***

Point Size	Input Data Width	Phase Factor Width	Slices	Block RAM	MULT 18x18	Max. Clock Frequency (MHz)		Transform Time			Device
								Clock Cycles	Time (µs)		
						-7	-6		-7	-6	
1024	24	24	4518	12	48	165	153	1024	6.21	6.69	2vp30
8192	24	24	6238	37	72	165	153	8192	49.65	53.54	2vp50

Note: ISE 6.3i speed file - Production 1.88 2004-08-11.

Table 9: **Performance and Resource Utilization for the Virtex-II Pro Family: Radix-4, Burst I/O**

Point Size	Input Data Width	Phase Factor Width	Slices	Block RAM	MULT 18x18	Max. Clock Frequency (MHz)		Transform Time			Data Load + Transform Time			Device
								Clock Cycles	Time (µs)		Clock Cycles	Time (µs)		
						-7	-6		-7	-6		-7	-6	
64	16	16	1330	8	9	223	205	91	0.41	0.44	155	0.70	0.76	2vp7
256	16	16	1410	7	9	223	205	289	1.30	1.41	545	2.44	2.66	2vp7
1024	16	16	1492	7	9	223	205	1319	5.91	6.43	2343	10.51	11.43	2vp7
2048	16	16	1637	7	9	223	205	3117	13.98	15.20	5165	23.16	25.20	2vp7
8192	16	16	1723	22	9	223	205	14387	64.52	70.18	22579	101.25	110.14	2vp20
65536	16	16	2132	158	9	223	205	131129	588.02	639.65	196665	881.91	959.34	2vp70
64	24	24	2665	12	36	170	158	97	0.57	0.61	161	0.95	1.02	2vp30
1024	24	24	2948	11	36	170	158	1321	7.77	8.36	2345	13.79	14.84	2vp30
8192	24	24	3076	33	36	170	158	14389	84.64	91.07	22581	132.83	142.92	2vp50

Note: ISE 6.3i speed file - Production 1.88 2004-08-11.

Table 10: **Performance and Resource Utilization for the Virtex-II Pro Family: Radix-2, Minimum Resources**

Point Size	Input Data Width	Phase Factor Width	Slices[2]	Block RAM[2]	MULT 18x18	Max. Clock Frequency (MHz)		Transform Time			Data Load + Transform Time			Device
								Clock Cycles	Time (µs)		Clock Cycles	Time (µs)		
						-7	-6		-7	-6		-7	-6	
64	16	16	531/709	3/0	3	223	200	265	1.19	1.33	329	1.48	1.65	2vp4
256	16	16	567/1365	3/0	3	223	205	1079	4.84	5.26	1335	5.99	6.51	2vp4
1024	16	16	630/3053	3/0	3	207	186	5187	25.06	27.89	6211	30.00	33.39	2vp4
2048	16	16	716	5	3	216	193	11338	52.49	58.75	13386	61.97	69.36	2vp4
8192	16	16	774	18	3	194	174	53334	274.92	306.52	61526	317.14	353.60	2vp7
65536	16	16	1061	130	3	201	181	524497	2609.44	2897.77	590033	2935.49	3259.85	2vp70
64	24	24	971	5	12	170	158	277	1.63	1.75	341	2.01	2.16	2vp7

Table 10: **Performance and Resource Utilization for the Virtex-II Pro Family: Radix-2, Minimum Resources *(Continued)***

Point Size	Input Data Width	Phase Factor Width	Slices[2]	Block RAM[2]	MULT 18x18	Max. Clock Frequency (MHz)		Transform Time			Data Load + Transform Time			Device
								Clock Cycles	Time (µs)		Clock Cycles	Time (µs)		
						-7	-6		-7	-6		-7	-6	
1024	24	24	1154	5	12	170	158	5190	30.53	32.85	6214	36.55	39.33	2vp7
8192	24	24	1245	25	12	170	158	53336	313.74	337.57	61528	361.93	389.42	2vp20

Notes:
1. ISE 6.3i speed file - Production 1.88 2004-08-11.
2. Second number is if input data and phase factors are stored in distributed memory.

Table 11: **Performance and Resource Utilization for the Virtex-4 Family: Pipelined, Streaming I/O**

Point Size	Input Data Width	Phase Factor Width	Slices	Block RAM	XtremeDSP Slices	Max. Clock Frequency (MHz)		Transform Time			Device
								Clock Cycles	Time (µs)		
						-11	-10		-11	-10	
64	16	16	1126	1	8	335	299	64	0.19	0.21	4vsx25
256	16	16	1539	4	12	335	299	256	0.76	0.86	4vsx25
1024	16	16	1975	7	16	335	299	1024	3.06	3.42	4vsx25
2048	16	16	2161	10	20	315	281	2048	6.50	7.29	4vsx25
8192	16	16	2664	24	24	315	281	8192	26.01	29.15	4vsx25
65536	16	16	3713	150	28	315	281	65536	208.05	233.22	4vsx35
64	24	24	2156	2	24	269	242	64	0.24	0.26	4vsx25
1024	24	24	3887	12	48	256	230	1024	4.00	4.45	4vsx25
8192	24	24	5320	37	72	256	230	8192	32.00	35.62	4vsx35

Note: ISE 6.3i speed file - Preview 1.47 2004-08-11.

Table 12: **Performance and Resource Utilization for the Virtex-4 Family: Radix-4, Burst I/O**

Point Size	Input Data Width	Phase Factor Width	Slices	Block RAM	Xtreme DSP Slices	Max. Clock Frequency (MHz)		Transform Time			Data Load + Transform Time			Device
								Clock Cycles	Time (µs)		Clock Cycles	Time (µs)		
						-11	-10		-11	-10		-11	-10	
64	16	16	1291	8	9	303	273	100	0.33	0.37	164	0.54	0.60	4vsx25
64	16	16	893	8	28	410	360	100	0.24	0.28	164	0.40	0.46	4vsx25
256	16	16	1367	7	9	303	273	292	0.96	1.07	548	1.81	2.01	4vsx25
256	16	16	974	7	28	421	370	292	0.69	0.79	548	1.30	1.48	4vsx25
1024	16	16	1445	7	9	303	273	1322	4.36	4.84	2346	7.74	8.59	4vsx25
1024	16	16	1052	7	28	421	370	1322	3.14	3.57	2346	5.57	6.34	4vsx25
2048	16	16	1555	7	9	303	273	3120	10.30	11.43	5168	17.06	18.93	4vsx25
2048	16	16	1162	7	28	315	281	3120	9.90	11.10	5168	16.41	18.39	4vsx25

Table 12: **Performance and Resource Utilization for the Virtex-4 Family: Radix-4, Burst I/O *(Continued)***

Point Size	Input Data Width	Phase Factor Width	Slices	Block RAM	Xtreme DSP Slices	Max. Clock Frequency (MHz)		Transform Time			Data Load + Transform Time			Device
								Clock Cycles	Time (µs)		Clock Cycles	Time (µs)		
						-11	-10		-11	-10		-11	-10	
8192	16	16	1643	22	9	303	273	14390	47.49	52.71	22582	74.53	82.72	4vsx25
8192	16	16	1251	22	28	315	281	14390	45.68	51.21	22582	71.69	80.36	4vsx25
65536	16	16	2083	158	9	303	273	131132	432.78	480.34	196668	649.07	720.40	4vsx35
65536	16	16	1771	158	28	342	305	131132	383.43	429.94	196668	575.05	644.81	4vsx35
64	24	24	2306	12	36	248	224	124	0.50	0.55	188	0.76	0.84	4vsx25
64	24	24	1597	12	64	410	360	118	0.29	0.33	182	0.44	0.51	4vsx35
1024	24	24	2528	11	36	248	224	1330	5.36	5.94	2354	9.49	10.51	4vsx25
1024	24	24	1824	11	64	256	230	1328	5.19	5.77	2352	9.19	10.23	4vsx35
8192	24	24	2677	33	36	248	224	14398	58.06	64.28	22590	91.09	100.85	4vsx35
8192	24	24	1972	33	64	256	230	14398	56.23	62.59	22588	88.23	98.21	4vsx35

Note: ISE 6.3i speed file - Preview 1.47 2004-08-11.

Table 13: **Performance and Resource Utilization for the Virtex-4 Family: Radix-2, Minimum Resources**

Point Size	Input Data Width	Phase Factor Width	Slices[2]	Block RAM[2]	Xtreme DSP Slices	Max. Clock Frequency (MHz)		Transform Time			Data Load + Transform Time			Device
								Clock Cycles	Time (µs)		Clock Cycle	Time (µs)		
						-11	-10		-11	-10		-11	-10	
64	16	16	515/728	3/0	3	330	297	283	0.86	0.95	347	1.05	1.17	4vsx25
256	16	16	564/1270	3/0	3	330	297	1082	3.28	3.64	1338	4.05	4.51	4vsx25
1024	16	16	634/3462	3/0	3	308	269	5190	16.85	19.29	6214	20.18	23.10	4vsx25
2048	16	16	671	5	3	308	269	11341	36.82	42.16	13389	43.47	49.77	4vsx25
8192	16	16	733	18	3	276	243	53337	193.25	219.49	61529	222.93	253.21	4vsx25
65536	16	16	1023	130	3	290	256	524545	1808.78	2049.00	590081	2034.76	2305.00	4vsx35
64	24	24	860	5	12	266	240	331	1.24	1.38	395	1.48	1.65	4vsx25
1024	24	24	994	5	12	256	230	5199	20.31	22.60	6223	24.31	27.06	4vsx25
8192	24	24	1128	25	12	256	230	53345	208.38	231.93	61537	240.38	267.55	4vsx25

Notes:
1. ISE 6.3i speed file - Production 1.118 2004-08-11.
2. Second number is if input data and phase factors are stored in distributed memory.

The dynamic range characteristics are shown by performing *slot noise* tests. First, a frame of complex Gaussian noise data samples is created. An FFT is taken to acquire the spectrum of the data. To create the slot, a range of frequencies in the spectra is set to zero. To create the input slot noise data frame, the inverse FFT is taken, then the data is quantized to use the full input dynamic range. Because of the quantization, if a perfect FFT is done on the frame, the noise floor on the bottom of the slot will be nonzero. The Input Data figures, which basically represent the dynamic range of the input format, display this.

This slot noise input data frame is fed to the FFT core to see how shallow the slot becomes due to the finite precision arithmetic. The depth of the slot shows the dynamic range of the FFT.

Figures 15 through 24 show the effect of input data width on the dynamic range. All FFTs have the same bit width for both data and phase factors. Block floating point arithmetic is used with rounding after the butterfly. The figures show the input data slot and the output data slot for bit widths of 24, 20, 16, 12, and 8.

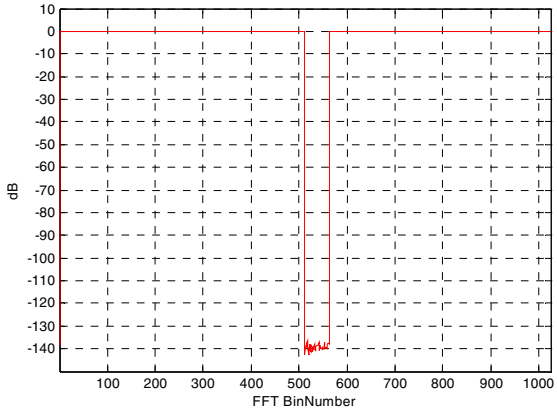


Figure 15: **Input Data: 24 Bits**

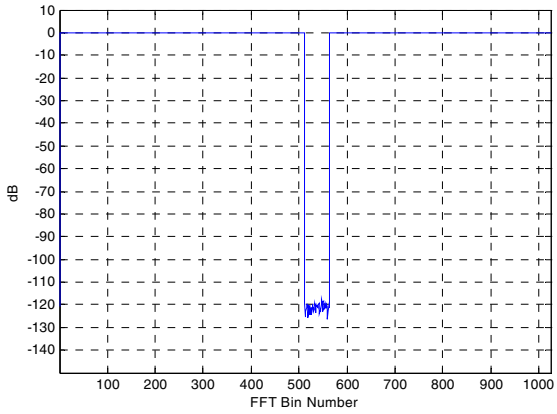


Figure 16: **FFT Core Results: 24 Bits**

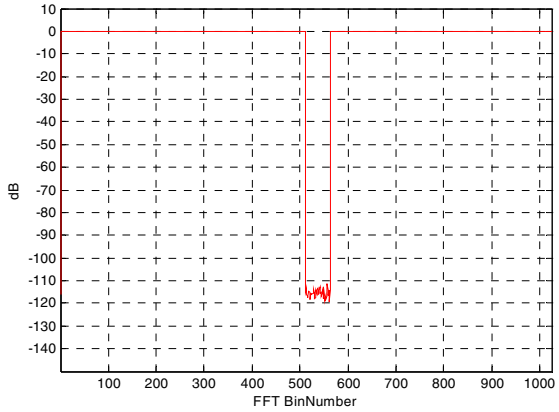


Figure 17: **Input Data: 20 Bits**

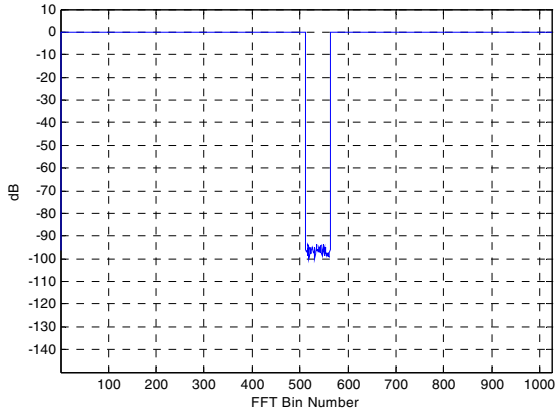


Figure 18: **FFT Core Results: 20 Bits**

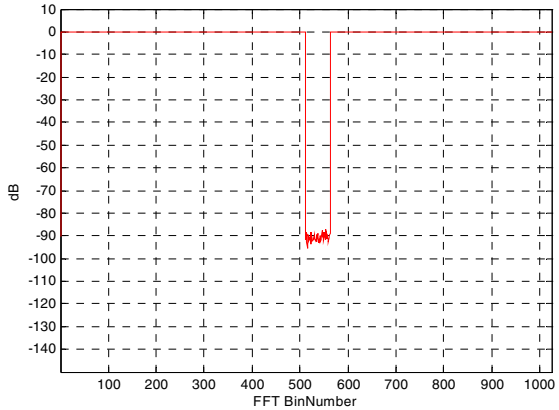


Figure 19: **Input Data: 16 Bits**

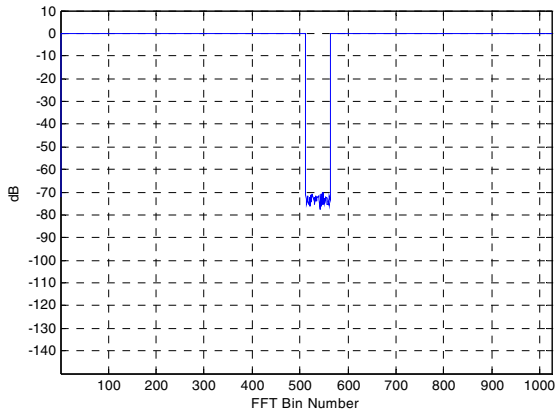


Figure 20: **FFT Core Results: 16 Bits**

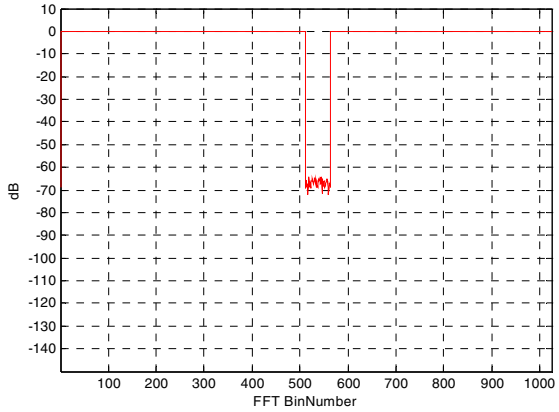


Figure 21: **Input Data: 12 Bits**

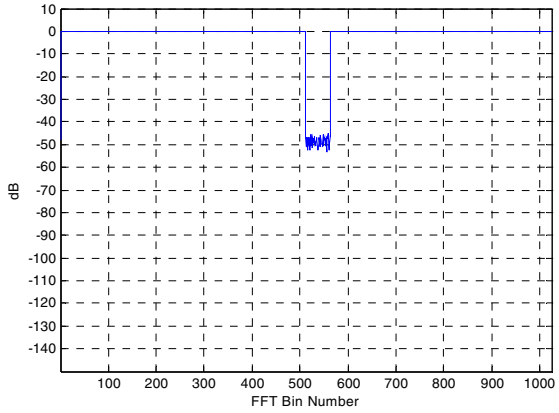


Figure 22: **FFT Core Results: 12 Bits**

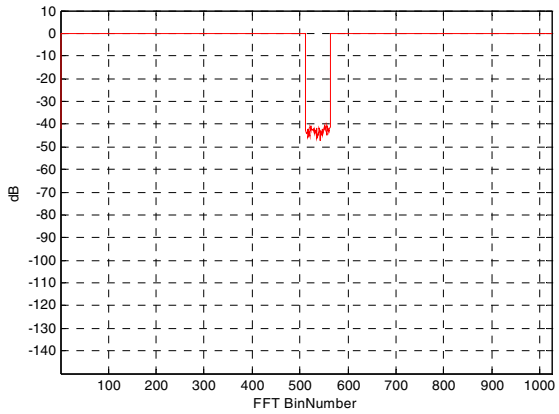


Figure 23: **Input Data: 8 Bits**

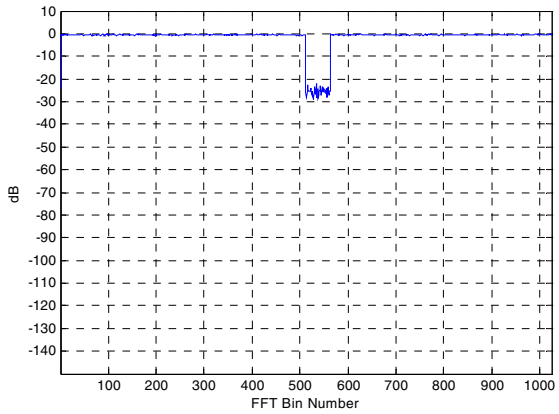


Figure 24: **FFT Core Results: 8 Bits**

There are several options available that also affect the dynamic range. Consider the arithmetic type used.

Figures 25, 26, and 27 display the results of using unscaled, scaled (scaling of 1/1024), and block floating point. All three FFTs are 1024 point, Radix-4 transforms with 16-bit input, 16-bit phase factors, and convergent rounding.

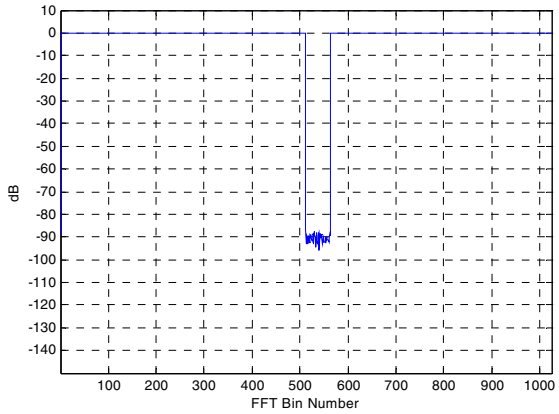


Figure 25: **Full-Precision Unscaled Arithmetic**

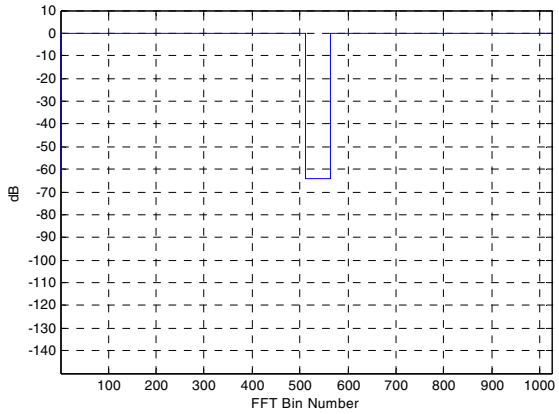


Figure 26: **Scaled (scaling of 1/N) Arithmetic**

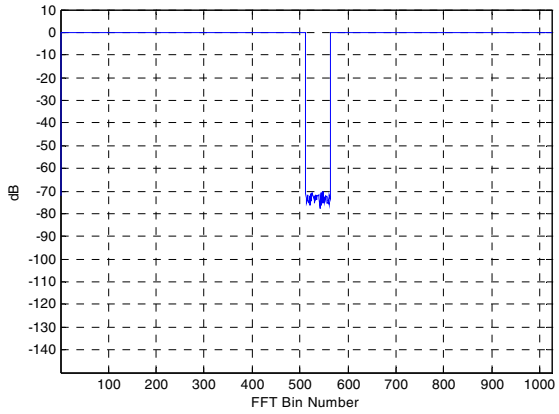


Figure 27: **Block Floating Point Arithmetic**

After the butterfly computation, the LSBs of the data path can be truncated or rounded. The effects of these options are shown below in Figures 28 and 29. Both transforms are 1024 points with 16-bit data and phase factors using block floating point arithmetic.

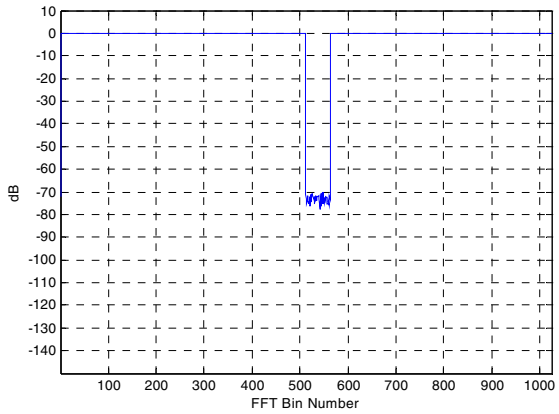


Figure 28: **Convergent Rounding**

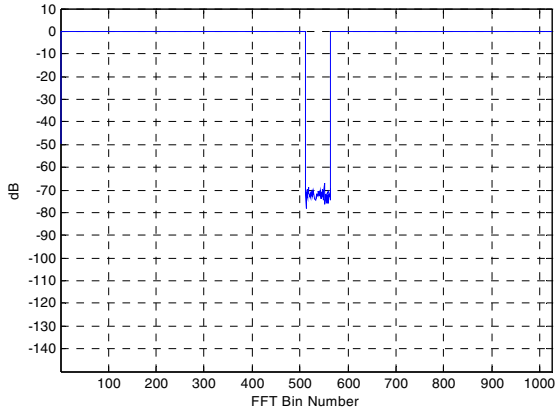


Figure 29: **Truncation**

For illustration purposes, the effect of point size on dynamic range is displayed Figures 30-32. The FFTs in these figures use 16-bit input and phase factors along with convergent rounding and block floating point arithmetic.

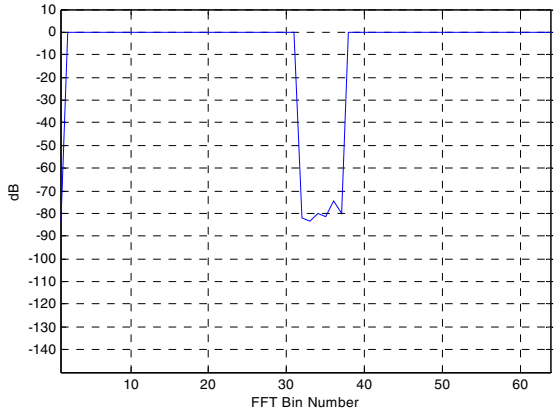


Figure 30: **64-point Transform**

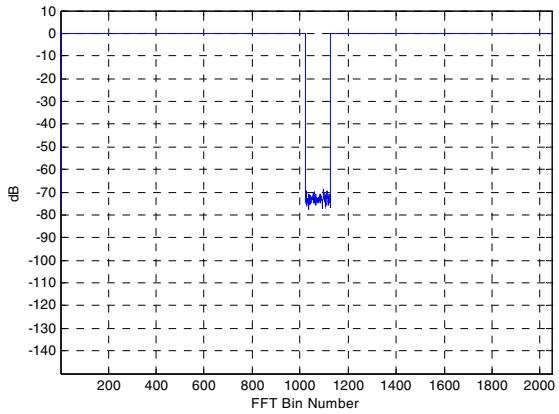


Figure 31: **2048-point Transform**

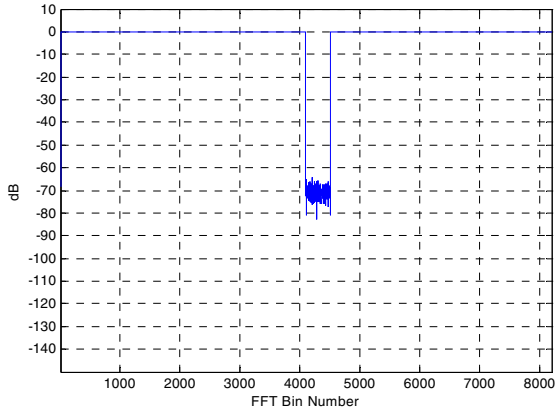


Figure 32: **8192-point Transform**

All of the above dynamic range plots show the result of a Radix-4 architecture. Figures 33-34 show two plots for the Radix-2 architecture. Both use 16-bit input and phase factors along with convergent rounding and block floating point.

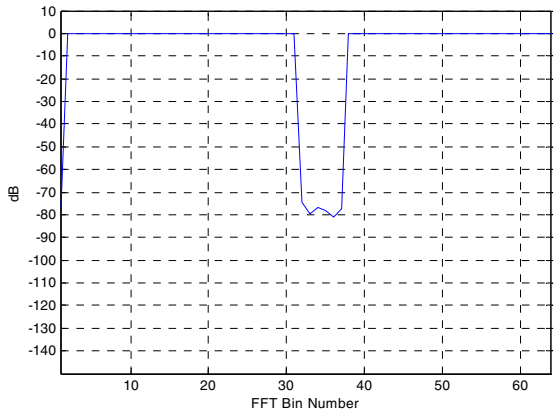


Figure 33: **64-point Radix-2 Transform**

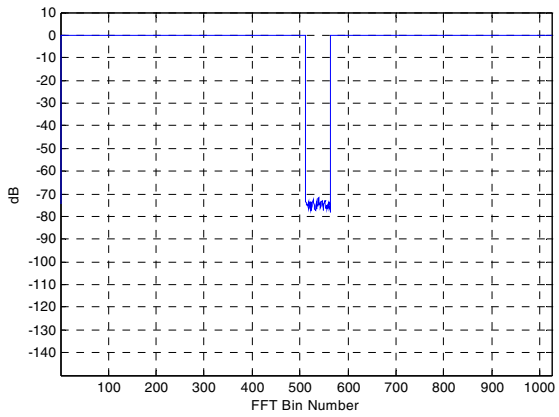


Figure 34: **1024-point Radix-2 Transform**

References

1. J. W. Cooley and J. W. Tukey, *An Algorithm for the Machine Computation of Complex Fourier Series*, *Mathematics of Computation*, Vol. 19, pp. 297-301, April 1965.
2. J. G. Proakis and D. G. Manolakis, *Digital Signal Processing Principles, Algorithms and Applications Second Edition*, Maxwell Macmillan International, New York, 1992.
3. W. R. Knight and R. Kaiser, *A Simple Fixed-Point Error Bound for the Fast Fourier Transform*, *IEEE Trans. Acoustics, Speech and Signal Proc.*, Vol. 27, No. 6, pp. 615-620, December 1979.
4. L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1975.

Ordering Information

The XFFT core may be downloaded from the Xilinx IP Center for use with the Xilinx CORE Generator v6.3i and later. The Xilinx CORE Generator system is bundled with all Alliance Series Software packages at no additional charge. Information about additional Xilinx LogiCORE modules is available on the Xilinx IP Center.

To order Xilinx software, please visit the Xilinx Silicon Xpresso Cafe or contact your local Xilinx sales representative.

Revision History

Date	Version	Revision
03/28/03	1.0	Xilinx release in new template.
07/14/03	2.0	Modified Figures 8 through 14, inclusive.
12/11/03	2.1	Updated to v2.1 release.
05/21/04	3.0	Updated to v3.0 release.
11/11/04	3.1	Updated document to support core v3.1 release - updated performance and resource utilization tables for Virtex-II and Virtex-II Pro. Also added performance and resource utilization tables for Virtex-4.

XAPP529 (v1.3) May 12, 2004

Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel

Author: Hans-Peter Rosinger

Summary

MicroBlaze™ has the ability to use its dedicated FSL bus interface to integrate a customized IP core into a MicroBlaze™ soft processor-based system. This document describes possible methods to include customized IP cores into a Soft Core Processor (SCP)-based design.

The FSL interface is described in great detail, and a reference application involving a 1-dimensional Inverse Direct Cosine Transform (IDCT) is used to show how the implementation of a customized core can be done in software and hardware. The first part of this document deals with the different methods of integrating user IP cores into a soft processor-based system. The second part contains a short overview on MicroBlaze and the FSL interface. After that, the reference design, which can be downloaded from the Xilinx web site, is explained. The last point of this document contains the conclusion regarding the use of the FSL interface

Introduction

One advantage of a Soft Core Processor (SCP) is its flexibility: it uses only the processor features required for a specific application. Another advantage is its ability to integrate customized user Intellectual Property (IP) cores, which can result in a dramatic acceleration in software execution time due to algorithms being executed in parallel in hardware and not sequentially in software. MicroBlaze is a powerful and inexpensive SCP solution for the Virtex™ and Spartan™-II/3-based FPGA series. MicroBlaze combines all the flexibility advantages of SCP.

Generally, there are two ways to integrate a customized IP core into a MicroBlaze-based embedded soft processor system. One way is to connect the IP on the On-chip Peripheral Bus (OPB). The OPB is part of the IBM Core Connect™ on-chip bus standard. The second way is to connect the user IP to the MicroBlaze dedicated Fast Simplex Link (FSL) bus system. If the application is time-critical, the user IP should be connected to the FSL bus system; otherwise, it can be connected as a slave or master on the OPB. If the customized core is connected to the dedicated FSL interface, it is then possible to use predefined C functions to use the user core in the application software. This document deals primarily with the connection of a user IP on the MicroBlaze FSL interface. For more information regarding the connection of a user IP on the OPB, please refer to the user core template document:

www.xilinx.com/ise/embedded/edk_docs.htm

Integration of a User IP into a Soft Processor-Based System

There are different ways to connect a user IP into a soft microprocessor-based system. In general, every application can be realized and implemented either as software algorithm or as structural hardware. It is important to use the hardware implementation advantage (parallel execution), which allows the realization of strict timing-driven applications and the ability to control the user IP in software (e.g., C or C++). Figure 1 demonstrates how the parallel execution advantage can be used. The software routine needs 12 clock cycles to calculate the result G; however, in hardware it takes only 2 clock cycles to compute the same result.

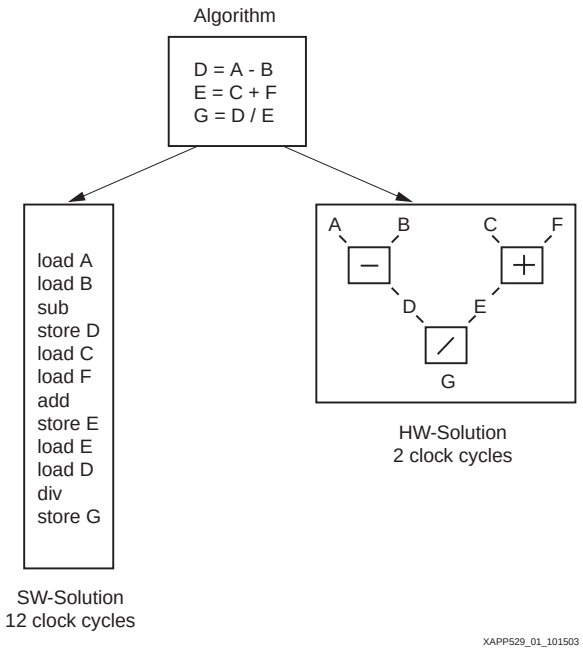


Figure 1: **Software versus hardware**

It is important to be able to use the customized IP core in the application program. Without an easy way to control the user IP, it doesn't make sense to include the core. One possible way is to integrate the user IP as an on-chip bus system (like the core connect bus). During the design stage of the core, the designer has to take the bus standard into account; that is, the designer has to make sure that he or she conforms to the bus specification if the core gets connected to the bus. This can be very time-consuming, a liability which no engineer can afford today. In some cases, templates, for example, the Xilinx IPIF user core template, exist. These templates make it easy and fast to connect an IP core on the Core connect bus. The next and more critical drawback to connecting the user IP to an on-chip bus is that most of the time the bus protocol overhead takes too much time and the speed advantage gets lost. Therefore, other different ways to include a customized user IP core are possible. One is to integrate the user IP as co-processor (if the processor has such a co-processor interface). If the soft processor has a special dedicated interface like the ARC Tangent, Tensilica, NIOS, or MicroBlaze soft processor, it is also possible to integrate a user IP. A soft processor is available as HDL source code or as a structural netlist. Therefore, it can be integrated into an ASIC or a FPGA.

Soft Processors Targeting ASIC Versus FPGA

It is important to understand the advantage of the flexibility made possible by using an FPGA design instead of an ASIC design. After the ASIC is manufactured, there is no way to reconfigure the logic inside the ASIC device. It would even be too costly to change the mask and manufacture a new ASIC. For a SCP, which targets the ASIC market, it is essential to be flexible in software. After the processor-based system with the customized instruction is mapped in an ASIC, it is possible to change the application only in software (changing the C-Code). Figure 2 shows an example on this. The first bar shows the execution time of a whole software program. Using a SCP with customized instructions reduces the overall execution time of the software program dramatically. If, for instance, the ASIC already exists but an aspect of the application changes for one customized instruction, then some modifications must be done.

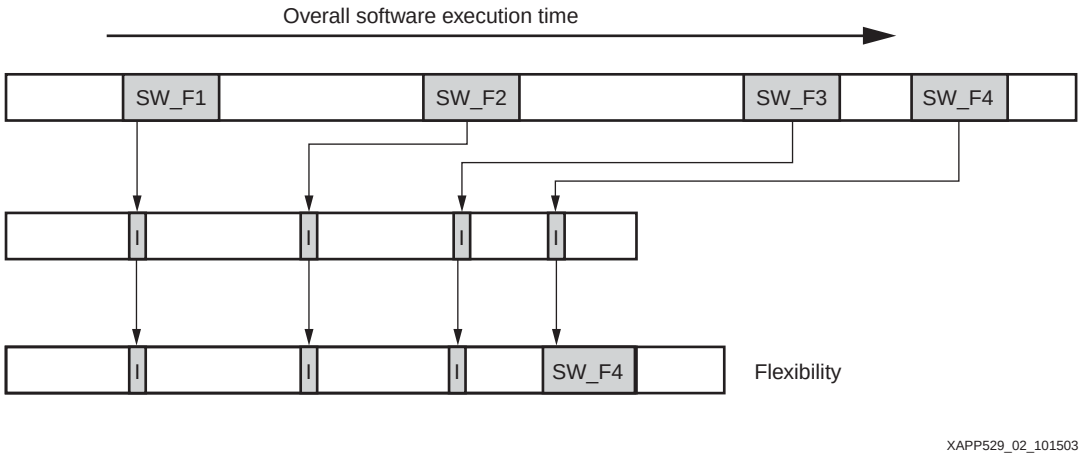


Figure 2: **Increasing software execution speed**

In an ASIC design, the only way is to not use the customized instruction but to address the new application requirement in software. Now, the overall software execution time will increase again; however, for an ASIC solution, it is the only way. Soft processors such as the ARC Tangent, the Tensilica, and the NIOS soft processor integrate the customized instruction completely in their execution unit and are useful if the target hardware is not changeable (ASIC).

FPGA devices instead allow for the reconfiguration of the internal logic very easily, quickly, and cheaply. Even in the last stages of the design, it is still possible to easily change the hardware inside the chip. If we look at the above example again, it is possible to change the new application requirement at any stage of the design in hardware, and it is not necessary to do the change in software. It is not that important to have the flexibility in software, because the flexibility in hardware has not been lost. For FPGA designs, it is not necessary or useful to include the customized user IP in the instruction set and inside the processor core, the RISC architecture. The next section details problems encountered when the customized IP core is included in the RISC architecture.

MicroBlaze FSL Interface Versus Customized Instruction

The integration of a customized IP core within the execution unit is very restrictive. One of the biggest restrictions is due to the nature of RISC processor architecture itself. Figure 3 shows a usual RISC processor architecture. Modern RISC architectures have a two-input and a one-output execution unit (ALU). Applications that require more than two input values and more than one output value are not optimal for these architectures, and several instructions have to be generated. Custom packet processing applications, for instance, require a lot of different dynamically changeable inputs (mask bits) and outputs. Those applications are not suitable for customized instructions because it is possible to use only two inputs (usually 64 bits) and one output (usually 32 bits).

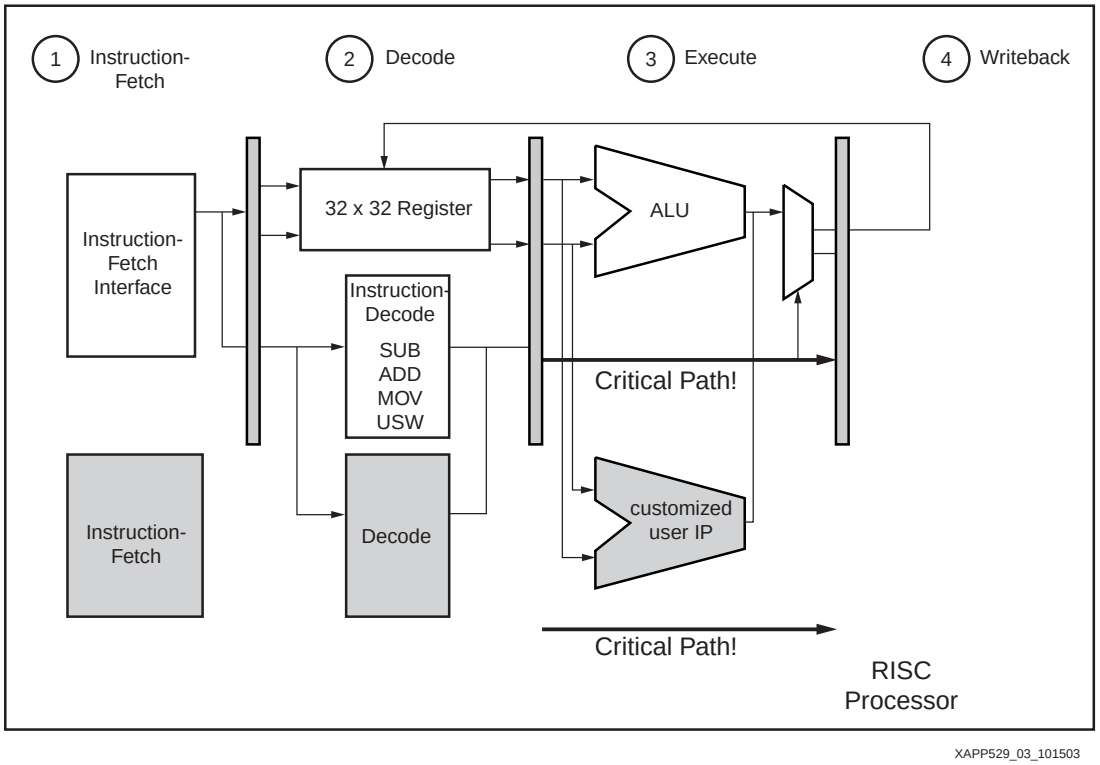


Figure 3: **Including a customized IP within the RISC architecture (Customized Instruction)**

Another bottleneck is the customized instruction itself. If the critical path of the whole system is through the user IP, the whole soft processor will decrease in performance (processor frequency), because the user IP is included within the soft processor architecture itself. If the RISC architecture doesn't allow the designer to stall the pipeline, the processor can't run at a higher frequency than the critical path would allow. The bigger the customized IP is, the more the designer must be careful not to decrease the whole processor performance. It is even not acceptable to cascade logic within one customized instruction, and several customized instructions have to be built. The software integration of customized instruction can't be handled directly from the compiler, thus the user has to use inline assembly to work with them. The customized instructions have to be implemented in software as inline assembler code and inline pragmas. This could produce a C application code, which is neither very clean nor portable.

Xilinx provides, with the MicroBlaze soft processor and the dedicated FSL interface, a very powerful, easy and flexible way to implement a customized user IP. Regarding the I/Os of the core, it is possible to use more than 2 dynamic inputs and more than 1 output because up to 16 FSL interface busses are provided. The user can use 8 inputs to the customized IP core and 8 outputs. Figure 4 shows the basic idea of connecting the customized user IP via the FSL interface onto the MicroBlaze. It is possible to provide the customized user IP core with many more inputs/outputs from another processor or external logic, and the big advantage is it is not necessary to change or extend the MicroBlaze core or the RISC architecture itself.

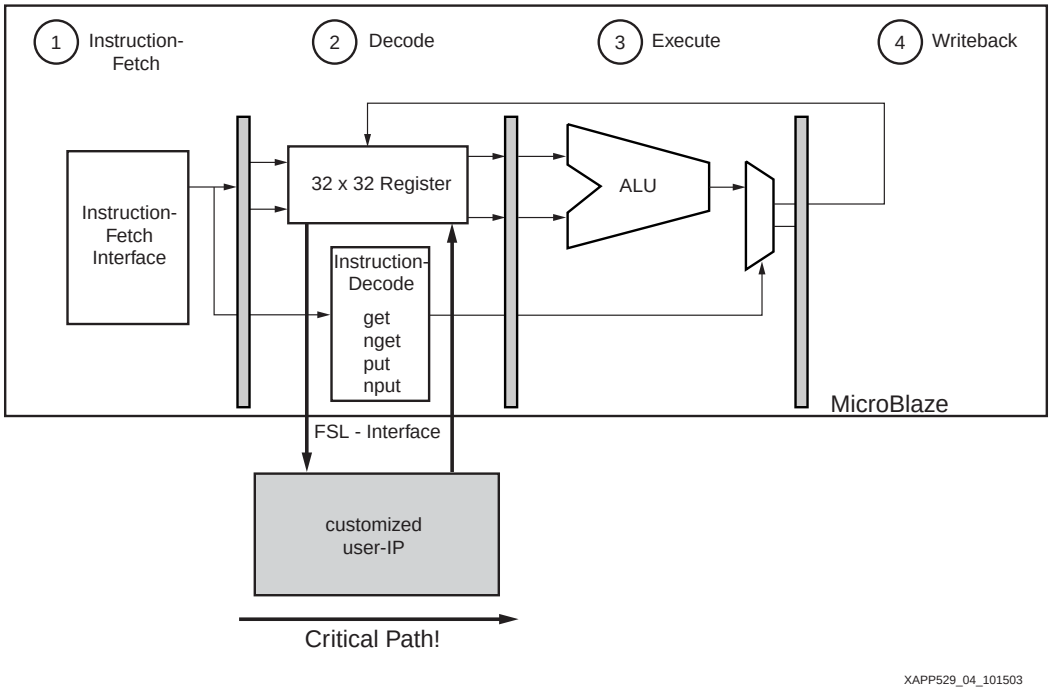


Figure 4: **Including a customized IP via the FSLinterface onto MicroBlaze**

Regarding the maximum frequency of the customized user IP, it won't decrease the clock frequency of MicroBlaze, because it is independent and the user core doesn't affect the internal MicroBlaze RISC architecture. It is no problem that the customized user IP is a subset of several IP cores that are cascaded together. The fact that the customized user core is implemented outside the processor architecture itself brings another advantage. If, for instance, the core takes 100 clock cycles to calculate a complex result, MicroBlaze can execute in the meantime a different application code and doesn't have to wait for the 100 clock cycles. The integration of the hardware in software doesn't require inline assembled code because the FSL interface has predefined C-macros that can be used for sending parameters to the hardware unit and to receive the result. Another powerful usage of the FSL is inter-processor communication. Two MicroBlaze processors have a very fast and clean way to communicate with each other. In the following sections, first MicroBlaze and then the FSL interface are discussed in greater detail. One example shows how a 1-dimension IDCT core gets connected in hardware within the EDK – XPS system builder and how to integrate the core in software.

General Description of the MicroBlaze Soft Processor

MicroBlaze is a standard 32-bit RISC Harvard-style Soft Processor, which is especially developed for the Virtex and Spartan-II/3-based FPGA architecture. The 32 by 32-bit registers are lookup table (LUT) RAM based. It guarantees a very short register access time. For memory, either the on-chip block RAM or off-chip memory can be used. The access time to the on-chip block RAM is minimal because there are dedicated routing resources to access them. Due to the fact that MicroBlaze is using the available FPGA resources very efficiently, it is possible to clock MicroBlaze up to 150 MHz. Thus, up to 125 Dhrystone MIPS can be reached. It is consequently the industry's fastest SCP for FPGAs. The MicroBlaze SCP can be customized for any application. Its barrel shifter, divide unit, data cache, instruction cache, and the FSL bus system are optional. The sizes of the caches are configurable from 2 to 64 Kbytes. Standard peripherals are provided as well and are Core Connect compatible. Consequently, they can be integrated in an embedded design very easy. These peripherals are either free, such as the memory controller, UART, interrupt controller, and timer, or commercial cores such as the Ethernet controller, gigabit Ethernet controller, PCI, HDLC, etc. All commercial IP Cores can be evaluated. For all free cores, the VHDL and the C-Code (TCP/IP stack) are readable. MicroBlaze is used in different areas such as

network applications, telecommunication applications, control, and consumer markets. Figure 5 shows a typical MicroBlaze SCP with its peripherals.

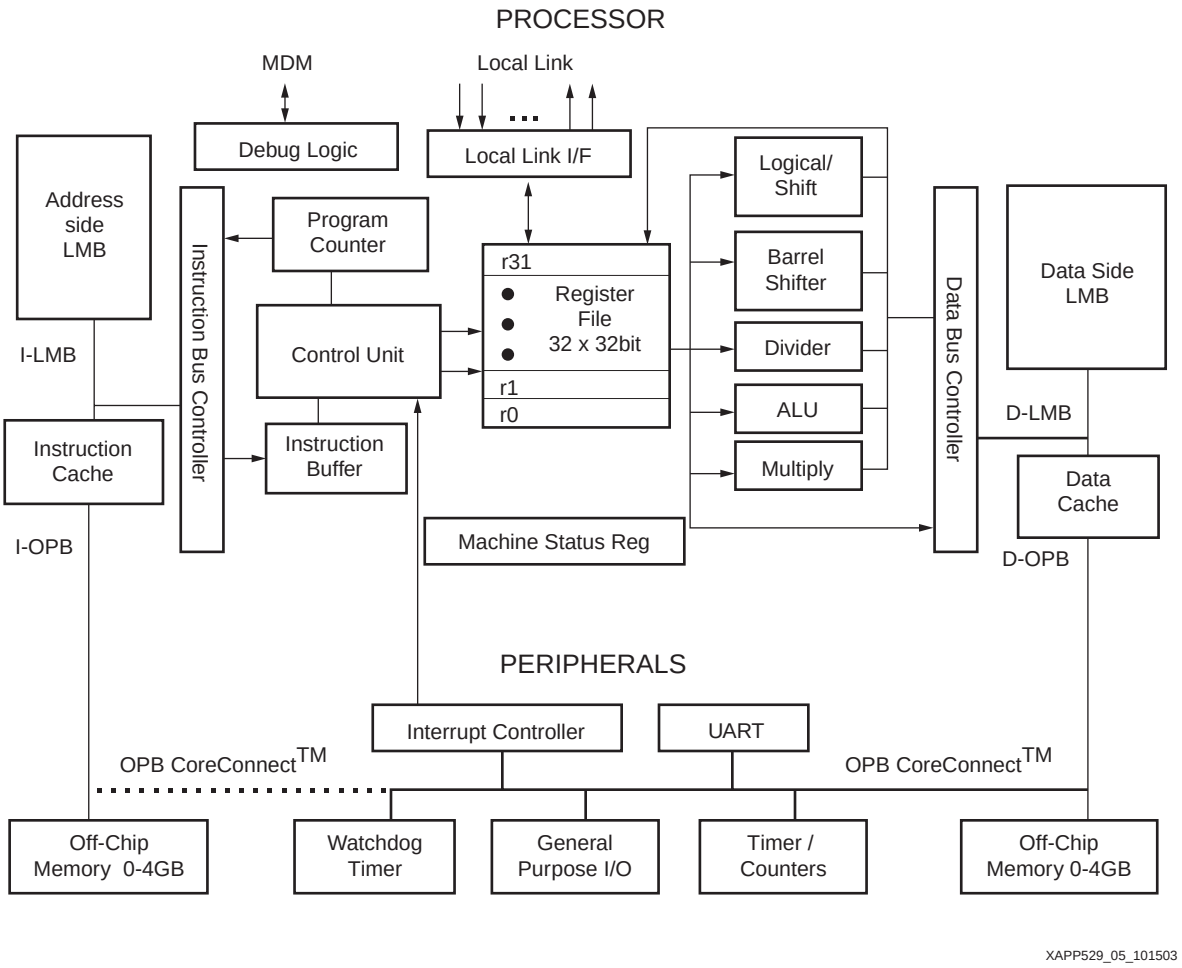


Figure 5: **MicroBlaze-based embedded processor system**

The Embedded Development Kit (EDK) includes the soft processor core and a standard set of peripherals and is available from Xilinx and its distribution partners. The kit includes a complete set of GNU-based software tools including the compiler, assembler, debugger, and linker. Variations of the kit include development boards that support the Virtex-E, Virtex-II, Virtex-II Pro, Spartan-II, Spartan-IIE, and Spartan-3 series of FPGAs.

For more information regarding MicroBlaze, please refer to the following link:

http://www.xilinx.com/ipcenter/processor_central/microblaze/index.htm

Detailed Description of the FSL Interface

This section describes the special FSL interface in detail. MicroBlaze contains eight input and eight output FSL interfaces. The FSL channels are dedicated unidirectional point-to-point data streaming interfaces. The FSL interfaces on MicroBlaze are 32 bits wide. Further, the same FSL channels can be used to transmit or receive either control or data words. A separate bit indicates whether the transmitted (received) word is control or data information. The performance of the FSL interface can reach up to 300 MB/sec. This throughput depends on the target device itself. The FSL bus system is ideal for MicroBlaze-to-MicroBlaze or streaming I/O communications.

The main features of the FSL interface are:

- Unidirectional point-to-point communication

- Unshared non-arbitrated communication mechanism
- Control and Data communication support
- FIFO-based communication
- Configurable data size
- 600 MHz standalone operation

The FSL bus is driven by one Master and drives one Slave. Figure 6 shows the principle of the FSL bus system and the available signals.

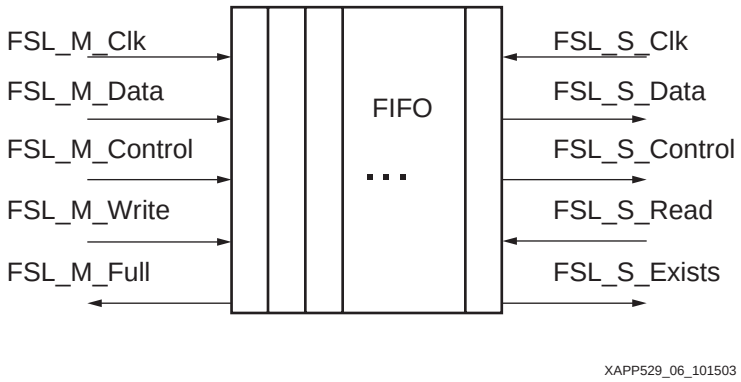


Figure 6: **FSL interface**

FSL peripherals may be created as a Master or a Slave to the FSL bus. A peripheral connected to the master ports of the FSL bus pushes data and control signals onto the FSL. All peripherals that act as a master to the FSL bus should create a bus interface of the type MASTER for the bus standard FSL in the Microprocessor Peripheral Description (MPD) file. A peripheral connected to the slave ports of the FSL bus reads and pops data and control signals from the FSL. All peripherals that are a slave to the FSL bus should create a bus interface of the type SLAVE for the bus standard FSL in the MPD file. The put and get instructions of MicroBlaze can be used to transfer the contents of a MicroBlaze register onto the FSL bus and vice-versa. The FSL bus configuration of MicroBlaze can be used in conjunction with any of the other bus configurations. Below is a brief overview of the FSL-related predefined C-functions available in EDK.

```
// Blocking Data Read and Write to Local Link no. id
microblaze_bread_datafsl(val, id)
microblaze_bwrite_datafsl(val, id)

// Non-blocking Data Read and Write to Local Link no. id
microblaze_nbread_datafsl(val, id)
microblaze_nbwrite_datafsl(val, id)

// Blocking Control Read and Write to Local Link no. id
microblaze_bread_cntlfsl(val, id)
microblaze_bwrite_cntlfsl(val, id)

// Non-blocking Control Read and Write to Local Link no. id
microblaze_nbread_cntlfsl(val, id)
microblaze_nbwrite_cntlfsl(val, id)
```

For more detailed information regarding the FSL bus information, please refer to the FSL bus data sheet (containing timing diagrams) and to the MicroBlaze user guide.

Description of the Application

As an application to demonstrate the use of the FSL interface, a 1-dimension IDCT is used. This DSP application highlights very well the performance win that could be reached. A 1-dimension IDCT realized in software would require a high execution time because the C- program would consist mainly of loops which get executed sequentially by the processor. If the application is implemented as its own hardware module, the execution time requires much fewer clock cycles. The used 1-IDCT core on the FSL interface is an example and needs approximately 150 LUTs and the latency of 64 clock cycles. Please note this IDCT core is used to show how to implement a user core on the FSL interface. The software application writes 8 values from memory to the FSL. The IDCT core gets the data and calculates the result. When the result is available, MicroBlaze reads the data (8 words) back from the FSL. The IDCT core is connected to the FSL interface as shown in Figure 7.

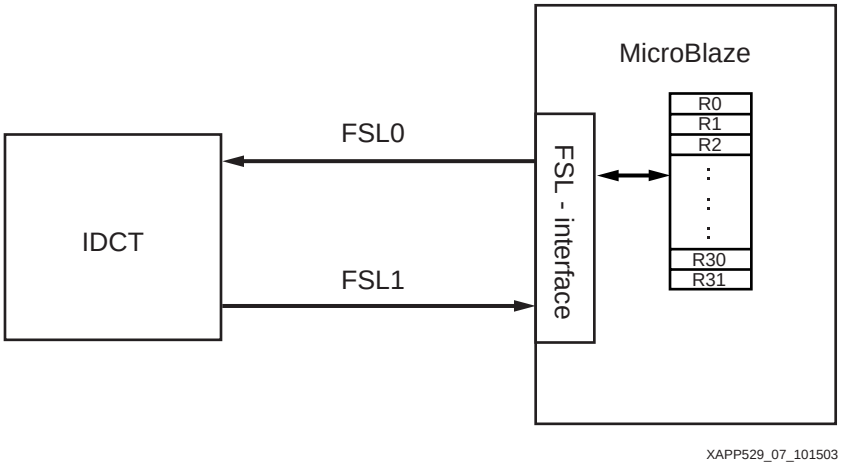


Figure 7: **Including the 1-dimensional IDCT IP via the FSL interface onto MicroBlaze**

For the FSL0 connection, the MicroBlaze is the Master on the FSL bus and the IDCT core is the Slave. Thus, MicroBlaze controls the data sent on the FSL0 bus to the IDCT core. For the FSL1 bus, it is vice versa, and the IDCT core is the Master and the MicroBlaze the Slave. The IDCT controls the data on the FSL1 bus.

By cascading the 1-dimensional IDCT core, it is possible to integrate a 2-dimensional IDCT core (Figure 8). The 1-D IDCT block will read from the FSL0 input and put the data out on the FSL1 bus. The corner-turn module also reads from FSL1 and puts out on FSL2. The last 1-D IDCT is also reading from the FSL2 and puts out the data on FSL3, which transfers the result back to MicroBlaze.

By doing this, the current 1-IDCT block can be used without any modification as a part in a 2-dimensional IDCT core. It also gives the user much more flexibility since it is possible to decide for another connection scheme at anytime.

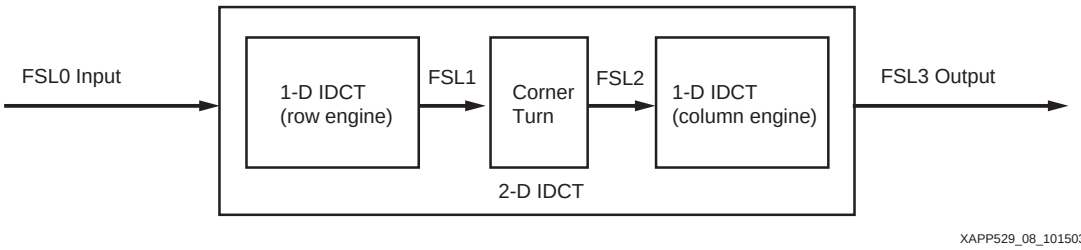


Figure 8: **Block Diagram for using a 1-D IDCT to implement a 2-D IDCT**

Integration in Hardware

This section describes how to integrate the user IP in a MicroBlaze-based embedded processor design. For this integration, the EDK 6.1 software tool is used. Figure 9 shows the embedded MicroBlaze design with the customized IDCT core and some OPB standard peripherals.

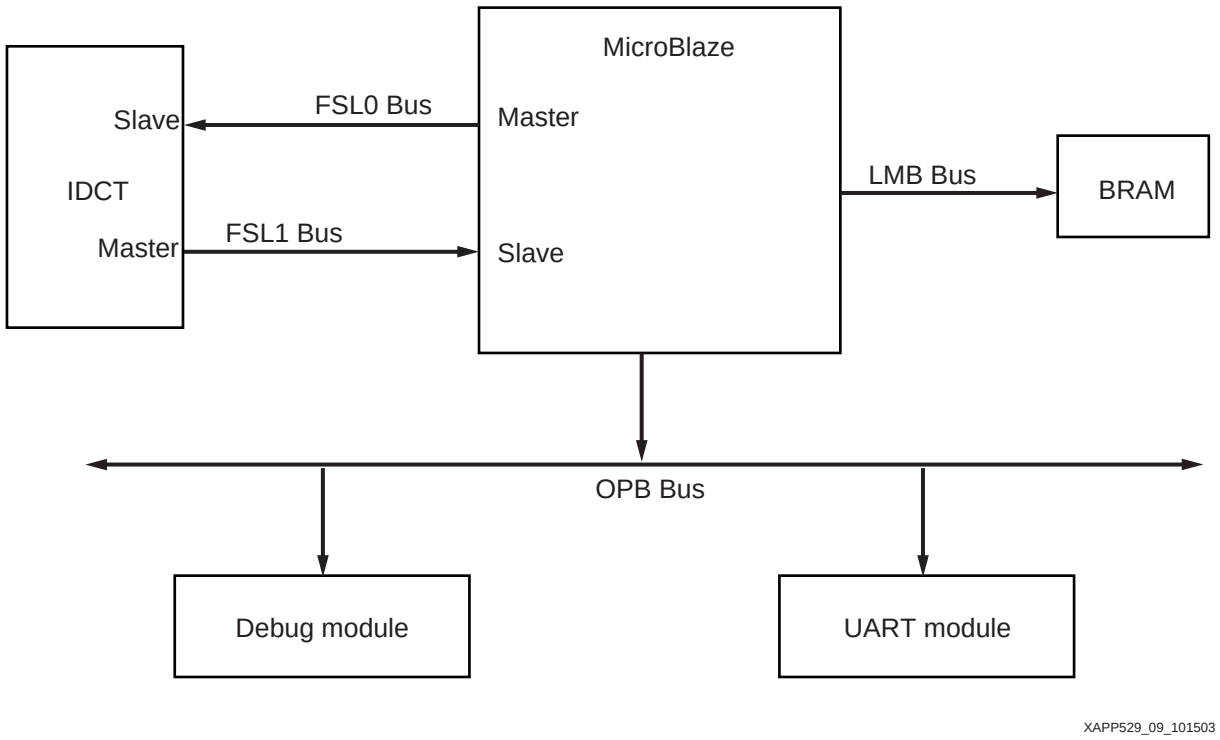


Figure 9: **Embedded processor system - hardware**

The whole embedded system consists of the MicroBlaze itself, two FSL bus systems, the user core, an OPB on-chip bus, two OPB peripherals (UART lite and the MicroBlaze Debug module), and the on-chip block RAM. The application program is stored in the on-chip block RAM.

Figure 10 shows in detail how the IDCT core is connected onto the MicroBlaze FSL interface. The IDCT is available in VHDL code. It is also possible to use a netlist instead.

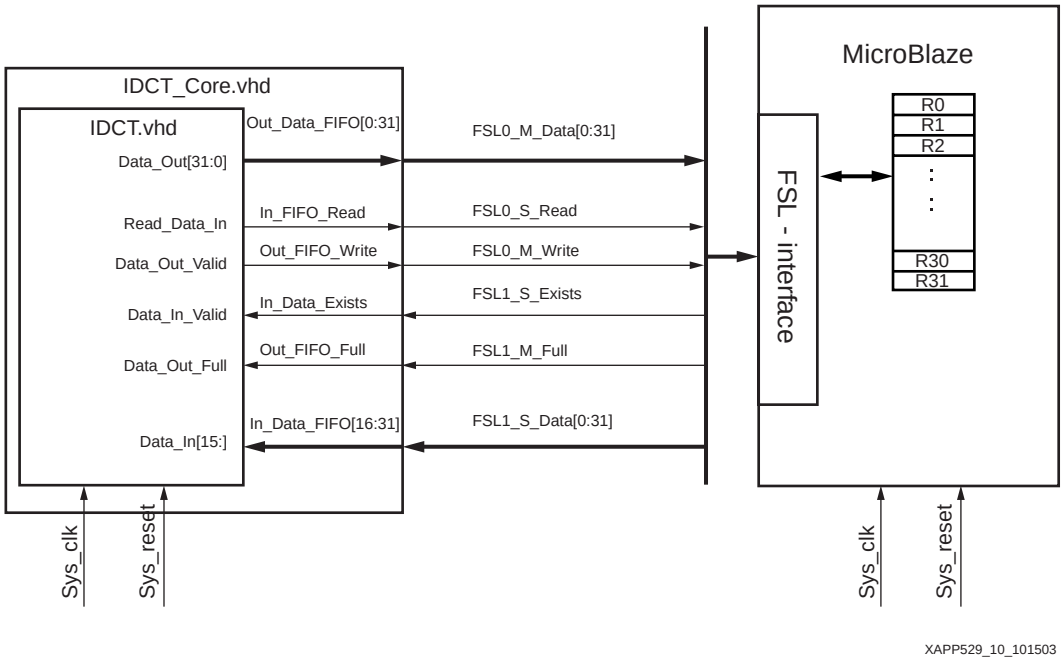
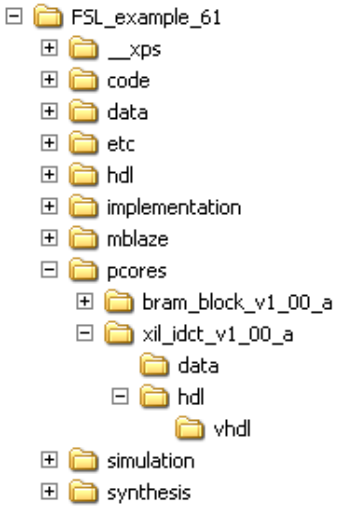


Figure 10: **Figure 10: Detailed connection of the 1-dimensional IDCT IP to MicroBlaze**

The control signals are provided from the FSL interface. External signals like the global system clock or the system reset can be integrated easily. In addition to the VHDL source code, a Microprocessor Peripheral Definition (MPD) file and a Peripheral Analyze Order (PAO) file are necessary. The MPD file defines the interface of the peripheral. The PAO file contains a list of HDL files that are needed for synthesis, and defines the analyze order for compilation. It is necessary to save all the files in a dedicated directory. The file structure in the XPS project should look like the following:



where the *xil_idct_v1_00_a/data* folder contains the MPD and the PAO file. The *vhdl* folder contains the VHDL source code of the user IP. If all the files are implemented correctly, the customized user core can be integrated in the Xilinx Platform Studio (XPS) and the bitstream of the hardware system can be generated.

Integration in Software

The next step is to integrate the user core into the software, the C application program. The application program is very simple and writes some data to the core and reads it back. The data block which will be written to the core consists of 8 input values. Before the next data block is written to the IDCT core, MicroBlaze waits for the resulting data block. Obviously, the resulting data block contains the 8 output values from the IDCT core. For writing into the user core, the predefined functions are used. For the example, the non-blocking write and read commands are used. The predefined functions are defined in the *mb_interface.h* file.

Verification of the Hardware

The verification of the hardware can be done in very different ways. The aim is to verify the FSL bus system and to be sure the data is transferred to the IP core and read back from the IP core correctly. It will be assumed that the customized IP core, in this case the IDCT core, already has the correct functionality. For the reference design, the verification was done with ModelSim 5.7e, and do script files are provided with the reference design. It can be seen from the output wave window that both the write to the core and the read from the core are successful.

Verification of the Software

To verify the software, the GNU debugger is used. The debugger can be started from XPS and is included in EDK. The *opb_mdm* debug module is used for the communication between MicroBlaze and the Xilinx Microprocessor Debugger (XMD) interface. On top of XMD, the GNU debugger GUI can be used.

Reference Design

The reference design targets the Memec Insight 2vp7 demo board (XC2VP7 -4, FG456 package). It has been implemented with the Xilinx EDK / ISE 6.2i software. The utilization values are completely device and implementation tool dependent. The total design requires 4 IOBs, 4 MULT18x18 elements, 4 RAMB16s and about 1300 slices, and the embedded soft processor design runs at a frequency of 100 MHz.

The MicroBlaze – FSL 1 dimension IDCT reference design can be downloaded from:

http://www.xilinx.com/bvdocs/appnotes/xapp529_6_1.zip

http://www.xilinx.com/bvdocs/appnotes/xapp529_6_2.zip

Conclusion

The MicroBlaze SCP with its powerful FSL interface can improve the performance of a whole application dramatically by outsourcing time-critical tasks into hardware. Besides the tremendous performance win, the solution is changeable until the last stage of the project by taking advantage of the flexibility of SCP and the FPGA architecture. By using customized instructions, the user is bounded to only two inputs and one output from the customized logic. With the FSL interface it is possible to have up to 8 inputs and 8 outputs, which allows much more flexibility, and cascaded logic within the customized core doesn't affect or lock the MicroBlaze RISC unit. The RISC architecture doesn't get manipulated and stays self-contained because it is not necessary to extend the processor RISC core. Predefined C functions are provided in EDK for integrating the customized user IP in a very easy and clean way in the C/C++ application program. If the target FPGA architecture is a Spartan-II, Spartan-IIE, or Spartan-3, it is even better, as these families are the most cost-effective solution that is available for high-performance embedded processor designs.

Revision History

The following table shows the revision history of this document.

Date	Version	Revision
11/30/03	1.0	Initial Xilinx release.
12/19/03	1.1	Corrected broken link.
3/16/04	1.2	Edit SCP to Soft Core Processor (1st use) and fix ref design link.
5/12/04	1.3	Edited links to reference designs.