To the Graduate Council:

I am submitting herewith a thesis written by Koay Teng Kuan entitled “Verification of Intellectual Property Blocks Using Reconfigurable Hardware.” I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Donald W. Bouldin
Major Professor

We have read this thesis
and recommend its acceptance:

Gregory Peterson

Michael Langston

Accepted for the Council:

Anne Mayhew
Vice Provost and
Dean of Graduate Studies

(Original signatures are on file with official student records.)

VERIFICATION OF INTELLECTUAL PROPERTY BLOCKS USING RECONFIGURABLE HARDWARE

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Koay Teng Kuan

December 2002

Dedication

This thesis is dedicated to my parents, Koay Cheoh Theam and Lim Mui Khim, great role models and friends, Uncle John, and the rest of my family for always believing in me, inspiring me, and encouraging me to reach higher to achieve my goals.

Acknowledgements

I wish to thank all those who helped me in completing my Master of Science in Electrical Engineering. I thank Dr. Bouldin for his patience and guidance in helping me to understand the different verification methodologies that can be apply to the Pilchard. I thank Dr. Peterson for his ideas and suggestions in how I should approach my research. I thank Dr. Langston for serving on my committee.

I thank Dr. Philip Leong and Tsoi Kuen Hung from Chinese University of Hong Kong for their assistance with the Pilchard hardware. I thank my fellow colleagues for their suggestions throughout my research.

I would like to also thank my family and friends, whose support and encouragement have made this work possible.

I would also like to acknowledge partial support for this project by the National Science Foundation under grants CCR-0075792 and EIA-9972889.

Abstract

The purpose of this thesis is to develop a procedure to verify intellectual property (IP) cores on the Pilchard platform which contains reconfigurable hardware. The hardware and tools used for the verification process are documented.

Two IP cores are used as examples of how the Pilchard design flow is to be applied. One core that does a simple logical function is implemented to serve as a demonstration of Pilchard read and write operations. To demonstrate the versatility of the hardware platform, a complex core that performs a Fast Fourier Transform operation was also implemented successfully.

Results from these IP implementations indicate that for high performance IP cores to be verified on the Pilchard, careful attention must be exercised to minimize the possible timing delay that occurs during place-and-route.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Previous Work

In recent years, advances in manufacturing have increased the gate capacity and performance of transistor-based chips. Integrated circuits are now capable of containing tens of millions of gates. This advancement has enabled complex designs to be integrated onto a single chip. However, chip designers are faced with a difficult challenge when trying to develop a ten million gate design from scratch.

Computer assisted design (CAD) tools have not evolved at the same pace as the silicon chip manufacturing industry. The available CAD tools are not able to provide a productivity gain as the gate count has increased. With the current design tools and methodologies, designers are not capable of effectively customizing a million gate without devoting additional time and resources. Hence, to take advantage of the large gate counts without sacrificing designer productivity, a new methodology has recently been introduced to the design community

Design reuse is a method in which pre-designed and pre-verified designs are reintegrated into a new design. Reusable designs are know as virtual components, cores, macros, or blocks. As most reusable cores are obtained from other sources, they also are

known as intellectual property (IP) cores. These IP cores, like a MPEG compression core or an on-chip memory, can be viewed as sub-components that could be used for development of a complex system design.

Reusable IP cores, when properly documented, can help increase design productivity. Statistics obtained by Dr. Barry Boehm of the University of Southern California [1], have shown that for software reuse, it takes about 50% more effort to prepare the code for reuse. However, the next designer using the code will benefit by a 70% reduction in development time. This shows that reuse still requires a significant amount of time to be integrated and verified in the new environment. However, the overall design period is shorter and the time-to-market is much better.

The general recognition of the advantage of IP blocks by the design community has prompted research into development of technical standards and methods to facilitate the integration of IP cores. The Virtual Socket Interface Alliance (VSIA) [2] was formed to develop the standards required for IP cores interfacing and design practices. CAD tool companies, like Synopsys and Mentor Graphics, have also combined their resources to help make IP reuse a reality by developing and demonstrating a design reuse methodology [3].

However, there is a challenge for designers trying to use reusable IP cores. Since most IP cores are considered “black boxes” by designers using them, the IP needs to be verified for its functionality and timing performance.

Testing and verification of the IP is an important part of the development tree because it ensures that the designs implemented are reliable and meet specification. The verification process becomes even more crucial when a few IP cores from different

sources are integrated together to function as a single entity. The different interfaces and requirements of the individual IP cores may vary from one to the other and therefore have to be validated.

The goal of verification is to achieve a very high level of test coverage such that a high confidence level in the IP's functional correctness can be achieved. The approach to achieve this confidence level is by performing verification on individual IP cores first and then design with multiple integrated IP cores. Most CAD tools allow IP verification to be performed by simulating the IP cores at register transfer level (RTL) or at the gate level [4]. However, the run-time of the simulator becomes a problem when a large number of tests are conducted.

Prototyping is another method of verifying the functional correctness of IP cores. A prototype allows for testing to be performed on the IP cores at real time speed. The run-time of testing done on a prototype may be 100 times faster than the run-time on a simulator when handling a large number of tests. In addition, the current Field Programmable Gate Array (FPGA) devices allow IP cores to be rapidly implemented. Hence, verification through prototyping using FPGA-equipped boards like the Wildforce and Firebird from Annapolis Microsystems [5] is often carried out.

However, for some actual hardware implementation and verification, some IP cores cause a problem to arise with the majority of FPGA platforms in existence. IP cores for applications like those for image processing and cryptography are designed to function as a coprocessor that operates in conjunction with a host microprocessor. With the majority of the FPGA platforms, like the Wildforce and Firebird from Annapolis

Microsystems, which use the PCI bus for data movement, the coprocessor IP cores that have been optimized to operate above 100MHz would be limited by the PCI bus speed.

In response to this data movement bottleneck issue, a dual in-line memory module (DIMM) based reconfigurable computing platform called “Pilchard” was developed by the Chinese University of Hong Kong (CUHK) [6]. By using the DIMM communication bus, the Pilchard manages to allow operation at either 100MHz or 133MHz, giving a maximum bandwidth of 1064 MB/s. Furthermore, since a VirtexE chip is used as the FPGA device of choice on Pilchard, the gate capacity allows Pilchard to do testing on larger designs. Initial integration of IP to take advantage of the Pilchard features was performed by a design group at CUHK [7].

Currently, the Electrical and Computing Engineering (ECE) department of the University of Tennessee (UT) is in the stage of finalizing the procurement of eight Pentium III Personal Computers (PCs) with a Pilchard board installed in each system. The individual Pentiums are part of a grid service cluster (GSC) implemented through the Scalable Intracampus Research Grid (SInRG) project. Using the SInRG middleware, NetSolve, users can access the Pilchard platform located in the GSC. A guideline on how to setup and access the Pilchard located on the GSC with NetSolve has been documented in a thesis [8].

1.2 Thesis Goal

The task to integrate and verify an IP design on a FPGA platform has never been easy. The main goal of this thesis is to develop a detailed procedure for IP verification using the Pilchard reconfigurable platform. As the Pilchard platform will soon become

available in the Electrical and Computing Engineering (ECE) department, it is necessary to develop a design development method to aid the first-time users of the platform. To achieve this goal, this thesis will provide documentation regarding the specific tools and methods used by the author in the preparation of an IP core to its implementation on the Pilchard hardware. This thesis will also provide a tutorial prepared by the author to show how a 2-radix fixed point Fast Fourier Transform (FFT) IP is integrated in the Pilchard board. This thesis is written based on following assumptions:

- Reader is able to code in VHDL.
- Reader is able to code in C/C++.
- Reader has basic knowledge in the Unix/Linux OS environment.

Chapter 2

Pilchard Platform and Design Package

2.1 Pilchard Overview

The Pilchard is a reconfigurable computing development platform with a memory slot interface developed by the Chinese University of Hong Kong. Pilchard's design goal was to address the data movement bottleneck issue by introducing the dual in-line memory module (DIMM) bus as the interconnecting bus between the host processor and the FPGA board. The Pilchard board is a 6-layer impedance-controlled FR4 board that is roughly twice the height of the standard dimension of a DIMM card. The Pilchard board is shown in Figure 2.1. Figure 2.2 shows the block diagram of the Pilchard board.

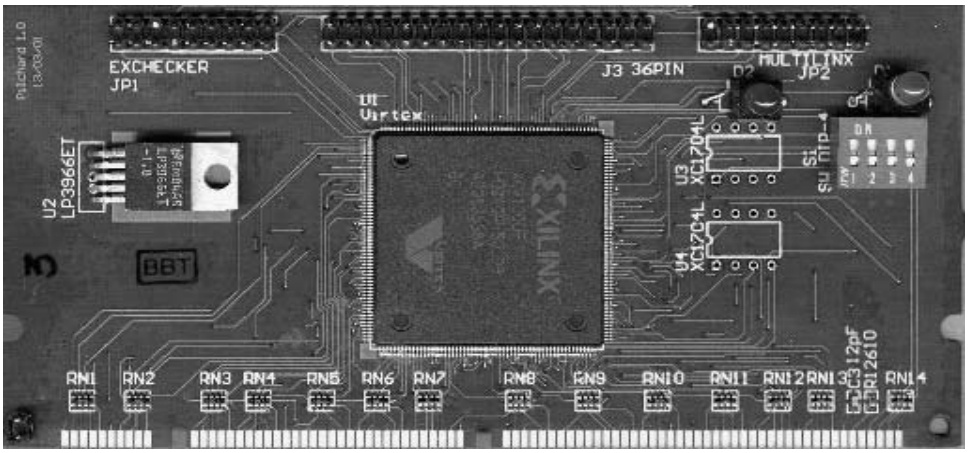

Figure 2.1: The Pilchard board. [6]

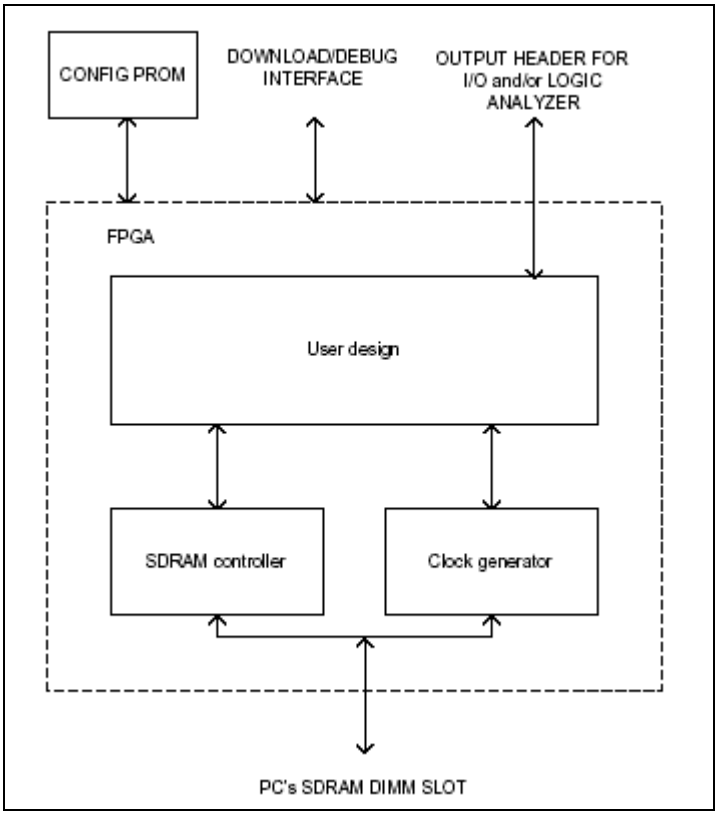

Figure 2.2: Block Diagram of Pilchard board [6]

As shown in the Figure 2.1, the Pilchard board has a single FPGA device component. The board FPGA component can be from any of the Xilinx Virtex or Virtex-E device family with the PQ240 or HQ240 packages. This gives the choice of FPGA devices range from the XCV150 to XCV1000E. Currently, this thesis work is based upon the use of a Pilchard board with a single Xilinx Virtex-E FPGA, XCV1000E-HQ240. The specifications of the Pilchard platform are shown in Table 1.

Pilchard is currently supported by an ASUS CUSL2-C motherboard and runs under the Linux operating system. Under the Linux OS environment, only the Parallel Cable III with the Xchecker interface can be used to download the configuration bit file into the Pilchard platform.

Table 1: Pilchard platform specifications. [12]

Features	**Description**
Host Interface	DIMM interface
	64-bit Data I/O
	12-bit Address bus
External (Debug) Interface	27 bits I/O
Configuration Interface	X-Checker, MultiLink and JTAG
Maximum System Clock Rate	133 MHz
Maximum External Clock rate	240 MHz
FPGA Device	XCV1000E-HQ240-6
Dimension	133mm x 65mm x 1mm
OS Supported	GNU / Linux

2.2 Xilinx VirtexE Chip

The Xilinx VirtexE FPGA chip is a SRAM FPGA. This means that the programmable connections in the FPGA are made using pass-transistors, transmission gates, or multiplexers that are controlled by Static RAM (SRAM) cells. SRAM technology allows for fast in-circuit reconfiguration but increases the physical size of the FPGA chip due to the space requirement of the RAM technology. The FPGA has three main configurable elements: the Configuration Logic Blocks (CLBs), Input/Output Blocks (IOBs), and interconnects. The CLBs provide the functional element for constructing the user's logic. The IOBs provide the interface between the package pins and the internal signal lines. Interconnects are the routing paths that connects the inputs and outputs of the CLBs and IOBs to the respective networks. Configuration of the FPGA is done by programming the internal static memory cells that determine the logic functions and the internal connections implemented in the FPGA. The XCV1000E-HQ240 used on the Pilchard board has the following features, summarized in Table 2.

Table 2: The Xilinx VirtexE XCV1000E-HQ240 features. [9]

Parameter	Features
System Gates	1,569,178
Logic Gates	331,776
CLB Arrays	64 x 96
Logic Cells	27,648
User I/Os	660
Differential I/Os	281
BlockRAM Bits	393,216
Distributed RAM Bits	393,216

The VirtexE FPGA also has eight digital Delay-Locked Loops (DLLs) to deal with clock distribution problems, four Global Clock Buffers (GCLKs) for global clock distribution and tristate buffers (BUFTs) for driving on-chip busses. One major feature of the VirtexE FPGA chip that is used extensively in this thesis is the Virtex™ Block SelectRAM+™ [9].

The Block SelectRAM+™ is a real synchronous memory block that is four CLBs high. In the chip selected, there are 96 blocks for a total of 393,216 Block SelectRAM bits. Each of the blocks can be seen in Figure 2.3, as a fully synchronous dual-ported 4096 bit RAM with independent control signals for each port. The block can be configured to different bit width as shown in Table 3, depending on the design requirements. The configuration of the block can be determined and changed by the designer using development software such as the Xilinx® Core Generator™ System V3.1i, which is used in this thesis. The SelectRAM+™ is a True Dual-Port™ RAM, meaning that every read/write request can be fulfilled in two clock cycles.

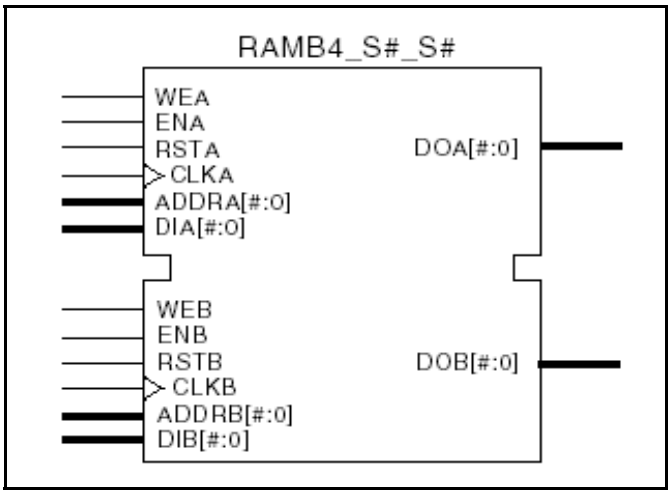

Figure 2.3: A fully synchronous dual port BlockRAM. [9]

Table 3: Configuration for the BlockRAM. [9]

Width	Depth	ADDR Bus	Data Bus
1	4096	ADDR<11:0>	DATA<0>
2	2048	ADDR<10:0>	DATA<1>
4	1024	ADDR<9:0>	DATA<3>
8	512	ADDR<8:0>	DATA<7>
16	256	ADDR<7:0>	DATA<15>

2.3 Pilchard Design Files - VHDL

The VHSIC Hardware Description Language (VHDL) is a language for describing a digital electronic system. The advantage of VHDL is that it is able execute concurrent statements. A concurrent statement is a statement that can be executed to perform its functions at the same time as another concurrent statement. This concurrency can be use to model actual circuit behavior.

VHDL also allows designs to be developed hierarchically. A large and complex design can be broken down into smaller modules or components. If required, these components can be further broken down into even smaller modules or sub-components. This helps the designer to concentrate on the specific functions and information of the design and to manage its complexity.

To aid designers in developing and integrating designs onto the Pilchard, a set of VHDL source and other necessary files are provided with the Pilchard platform. The VHDL files that are needed by designers in dealing with the VHDL codes of their design are: 1) pilchard.vhd, and 2) pcore.vhd.

The “pilchard.vhd” file is the top level VHDL code that interfaces the Pilchard board to the host DIMM slot directly. It configures the necessary global clock signal, I/O

buffer, and startup reset of the Pilchard FPGA VirtexE chip. Designers usually do not need to modify the code in this file. This file is modified only when new sources are added to the interface, such as a new clock sources. This VHDL file serves as a wrapper file to the “pcore.vhd” during design synthesis. The VHDL code of the “pilchard.vhd” is included in the Appendix.

The VHDL file “pcore.vhd” is where the designer’s chosen IP cores are added. This “wrapper” file contains the predefined ports that designers can use for accessing the host interface. The VHDL code template of the “pcore.vhd” is shown as Figure 2.4.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity pcore is
port (
      clk: in std_logic;
      clkdiv: in std_logic;
      rst: in std_logic;
      read: in std_logic;
      write: in std_logic;
      addr: in std_logic_vector(13 downto 0);
      din: in std_logic_vector(63 downto 0);
      dout: out std_logic_vector(63 downto 0);
      dmask: in std_logic_vector(63 downto 0);
      extin: in std_logic_vector(25 downto 0);
      extout: out std_logic_vector(25 downto 0);
      extctrl: out std_logic_vector(25 downto 0) );
end pcore;

architecture syn of pcore is

-- Users can modify this section to include their designs

end syn;
```

Figure 2.4: VHDL template for the wrapper file “pcore.vhd”

In addition to the two VHDL files, a Pilchard user constraint file (UCF) and library of pre-synthesized components in (EDIF) formats are also provided. The UCF file contains information about the pin locations and clock frequency of the FPGA VirtexE chip on the Pilchard board. The EDIF library contains the pre-synthesized components used by the “pilchard.vhd” code.

2.4 Pilchard Design Files - C Codes

The C Language is a programming language used widely by the computer science community. As a sequential programming language, the C code executes its process line by line from the beginning to the end of the code. In the Pilchard design process, the C codes are used to help the designers download their design into the Pilchard as well as provide a means in which the host computer can access the Pilchard board.

Three C code source files and a makefile are provided with the Pilchard platform. The “iflib.h” is the header file that provides the read and writes API function prototypes for Pilchard. The “iflib.c” is the implementations of the API functions. The “download.c” is the C codes to download the configuration bit file to the Pilchard board. All three C source files are included in the Appendix.

The four API functions defined in the “iflib.h” header file deals with the data transfer between the host and the Pilchard. The “write64” and “read64” APIs handle the 64-bit data transfer. The 32-bit data transfer is handle by the “write32” and “read32” APIs. Since the 32-bit data transfer is inefficient and slow, it is recommended that only the 64-bit data transfer APIs are used. The 64-bit APIs use a structure with two integers to store the 64-bit data as two 32-bit data. The 32-bit APIs uses just a single integer to

store its 32-bit data. All APIs uses character address pointers. In this addressing method, to increment an address in the actual hardware, the software address had to be incremented by eight.

The template for the main C code that is used to interface the Pilchard board is shown as Figure 2.5. The C code opens the Pilchard board as an address space memory mapped device. The “mmap” command allow access to the Pilchard board registers directly without incurring the overhead of a system call.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

#include "iflib.h"

int main (void)
{
   int fd;
   char *memp;

   fd = open(DEVICE, O_RDWR);
   memp = (char *)mmap(NULL, MTRRZ, PROT_READ, MAP_PRIVATE, fd, 0);
   if (memp  == MAP_FAILED) {
      perror(DEVICE);
      exit(1);
   }

   munmap(memp, MTRRZ);
   close(fd);
   return 0;
}
```

Figure 2.5: The main C code for the user program.

2.5 Pilchard Host Interface

The main function of the host interface is for host-Pilchard data transfer. Through the "pcore.vhd", the designer can access the memory bus for a 64-bit data transfer by using either the "din" or "dout" port. Designers that are integrating their designs into the Pilchard have to be familiar with the host read cycle and host write cycle through these ports. This is to ensure that they are able to synchronize the data transfer of their designs correctly.

In the "pcore.vhd", when the host read is initiated for a certain memory address in Pilchard, the address will appear in the "addr" port. The "read" port will change to high at the same time. The value at the memory address will be ready at the "dout" port at the next clock cycle. The host read cycle in "pcore.vhd" is shown as Figure 2.6.

When a host write is initiated to a memory space in Pilchard, the address will appear in the "addr" port. The "write" port will change to high. The data that is to be written to the memory address must also be ready at the same time. The cycle for host write in "pcore.vhd" is shown as Figure 2.7.

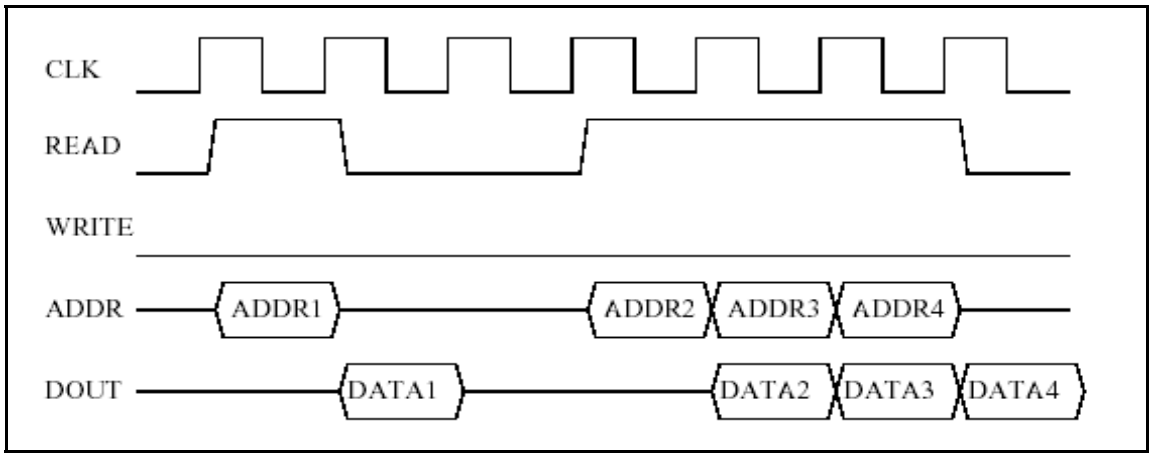

Figure 2.6: Pilchard host read cycle. [12]

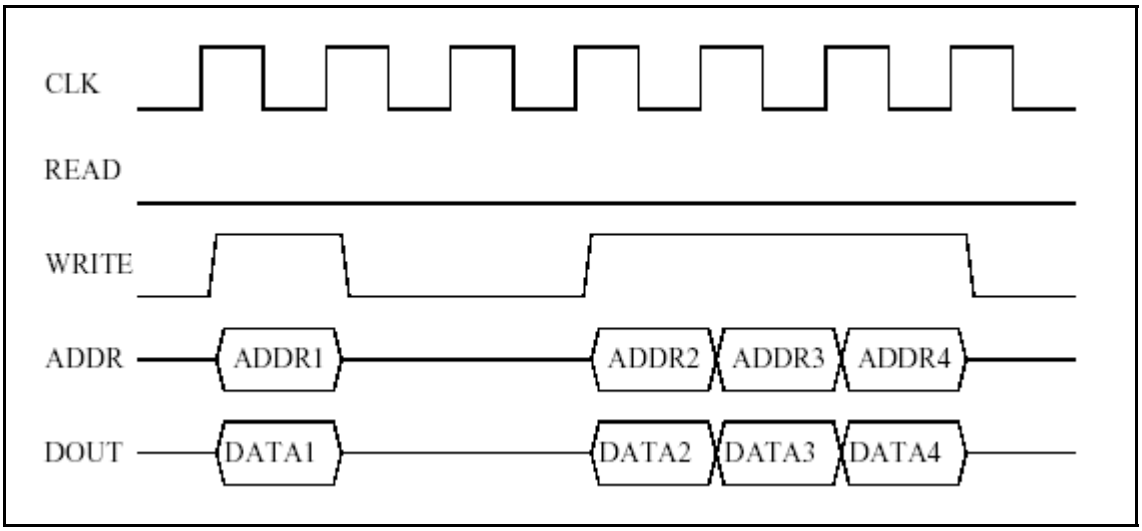

Figure 2.7: Pilchard host write cycle. [12]

2.6 Pilchard Access

For the work described in this thesis, the Pentium PC with the Pilchard board is located in the Chinese University of Hong Kong network. In order to access that Pilchard board, a UTK user would have to remotely connect to the CUHK network through the Internet. Users are required to use the Secure Shell (SSH) when trying to access the CUHK network through the Internet. The various gateways and UNIX OS machines connecting the Pilchard to the Internet in the CUHK network are shown as Figure 2.8.

The first gateway that a remote access user would encounter is the gateway that allows Internet connection to the CSE department of CUHK. Most UNIX commands do not function at this gateway. Beyond this gateway is a fully functional UNIX machine, sparc77, that users can use to store data. A second gateway, pc90017, allows users on the CSE department UNIX machines to access the machine, utk1, with the Pilchard board. A login name and password is required for each of the gateway and UNIX machines. The machine, utk1, is used to store the files and data that will be used by the Pilchard board.

Figure 2.8 also shows how data transfer is done using the UNIX File Transfer Protocol (FTP). The “infile” represent the data files needed to be transferred from the Internet to the utk1 machine. The “outfile” represent the data file needed to be transferred from the utk1 machine to the Internet. After accessing the sparc77 machine, the “infile” is transferred via FTP from the Internet and stored. The utk1 machine is then accessed and the “infile”, stored in the sparc77 machine, is transferred via FTP over to be stored. For file transfer from the utk1 machine to the Internet, the “outfile” is transferred via FTP to the sparc77 machine before being transferred again to the Internet.

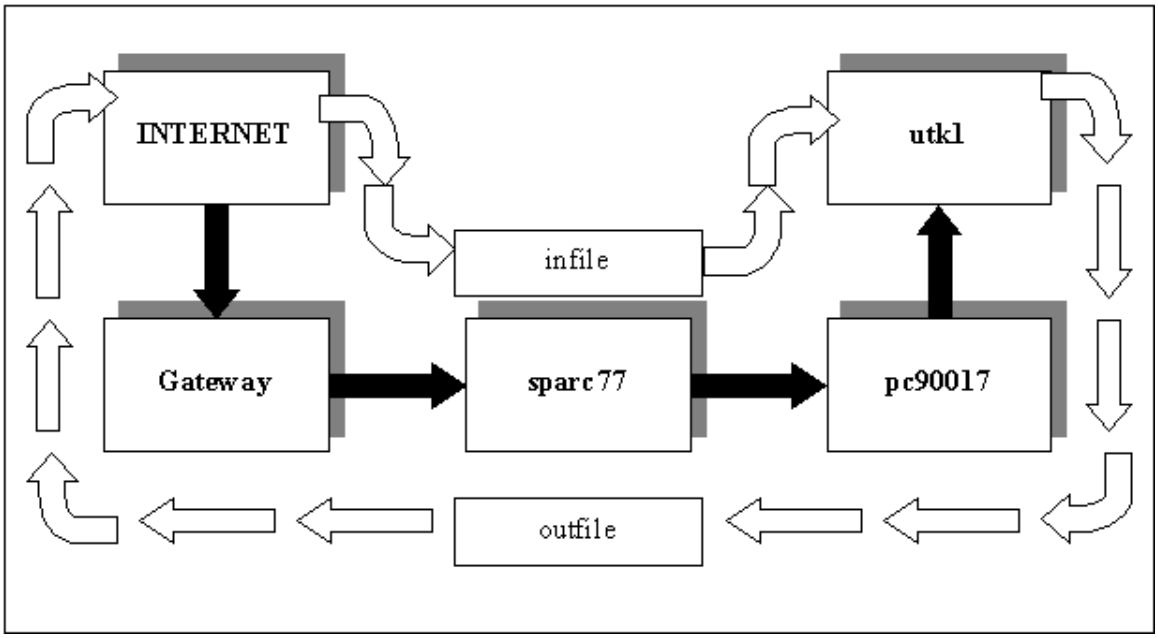

Figure 2.8: Gateways and UNIX machines connecting the Internet to Pilchard

Chapter 3

Software Tools

As the purpose of this thesis is to develop a detailed procedure to verify an IP core on the Pilchard platform, the tools employed play an important part in the development tree. Flexibility and simplicity of the development tools allow for shorter development time and resources in verifiying the IP.

3.1 Editing

For VHDL source editing, the UNIX “xedit” is initially used to develop and write the VHDL codes. Modifications and debugging of the VHDL codes are performed in Mentor Graphics® ModelSim SE VHDL5.5c GUI interface. The reason ModelSim was chosen is because it allows for modification and verification of the VHDL codes simultaneously. For C code editing, the UNIX “xedit” is also used.

3.2 Compilations and Simulation

For VHDL, Mentor Graphics® ModelSim SE VHDL 5.5c is used for the compilation and simulation of the pre-synthesis design. An executable script has been written to perform these functions. In the script, the VHDL files of a design and the behavioral models files

from the XilinxLibCore are loaded into ModelSim before the GUI is initiated. Under the ModelSim GUI interface, the designs can be modified, recompiled or simulated. The script written to simulate the Pilchard testbench of "fft4.vhd" IP core is shown as Figure 3.1. Mentor Graphics® HDL Designer™ Pro (version 2001.5) is used to generate the graphical representation of the state machines and modules of the design.

For C codes, the C compiler is used in conjunction with the make file to create the executable version of the respective C codes. The C compiler, GCC 2.95, installed on the Pilchard PC was chosen because the compiler has the necessary libraries required for codes compilation. Figure 3.2 shows the make-file for the "iftest" implementation.

```
#!/usr/bin/csh -f

source ~cad/.cshrc
mentor_tools

vlib XilinxCoreLib
vlib work
vmap XilinxCoreLib XilinxCoreLib

vcom -work XilinxCoreLib /sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/ul_utils.vhd
vcom -work XilinxCoreLib /sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/mem_init_file_pack_v3_1.vhd
vcom -work XilinxCoreLib /sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/blkmemdp_pkg_v3_1.vhd
vcom -work XilinxCoreLib /sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/blkmemdp_v3_1_comp.vhd
vcom -work XilinxCoreLib /sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/blkmemdp_v3_1.vhd
vcom -work XilinxCoreLib /sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/prims_constants_v4_0.vhd
vcom -work XilinxCoreLib /sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/prims_utils_v4_0.vhd
vcom -work XilinxCoreLib /sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/c_reg_fd_v4_0.vhd
vcom -work XilinxCoreLib /sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/c_reg_fd_v4_0_comp.vhd
vcom -work XilinxCoreLib /sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/c_mux_bus_v4_0_comp.vhd
vcom -work XilinxCoreLib /sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/c_mux_bus_v4_0.vhd

vcom -work work FFTPACK.VHD
vcom -work work CONTROL1.VHD
vcom -work work FFT1.VHD
vcom -work work STATE.VHD
vcom -work work FFT1_TOP.VHD
vcom -work work ram_c_4.vhd
vcom -work work ram_ir_4.vhd
vcom -work work pcore.vhd
vcom -work work pcoretb.vhd
vcom -work work tb.vhd
vsim tb
```

Figure 3.1: The ModelSim script for the Pilchard testbench of "fft4.vhd".

```
all: pilchard.o

# note modules outside of kernel source should properly
# do this, not work like this (this breaks SMP machines for
# one). See Olaf Titz' CPIE module for a good example (link
# from patches page).

# for compiling a kernel module
# note that -O2 is *necessary*. Kernel code assumes optimisations
# performed by gcc

CC= gcc
KCFLAGS=-DDEBUG -D__KERNEL__ -I/usr/src/linux/include -Wall -Wstrict-prototypes -
O2 -fomit-frame-pointer -pipe -fno-strength-reduce -malign-loops=2 -malign-jumps=2
-malign-functions=2 -DMODULE

# for user space
CFLAGS= -Wall -Wstrict-prototypes  -g -O2

pilchard.o: pilchard.c
      $(CC) $(KCFLAGS) -c -o $@ $<

install: pilchard.o
      -rmmod pilchard
      insmod pilchard.o
      /usr/bin/setpci -s 0:0.0 0x54=9
      cp pilchard.o /lib/modules/pilchard/.

iftest: iftest.c iflib.o iflib.h
      gcc $(CFLAGS) -o iftest iftest.c iflib.o

iflib.o: iflib.c iflib.h

download:   download.c
      gcc $(CFLAGS) -o download download.c

clean:
      rm -f *.o iftest download core
```

Figure 3.2: The make-file for the “iftest” implementation.

3.3 Synthesis

For synthesis, FPGA Compiler II from Synopsys tools version FC3.5 was used. The synthesis step is automated by a script that uses the fc2_shell command line. Included with the Pilchard design package is a script, “pilchard.fst”, which works with the fc2_shell scripting command to synthesize the interface of the Pilchard together with the design core.

The script is written to target the VirtexE chip, XCV1000E-HQ240. The design modules or IP cores, either in VHDL or in EDIF format, can be added to the script. FPGA Compiler II required that the lowest level design files be added first. Once the last file is added, the FPGA Compiler II will analyze the added files for any coding errors and violations. A chip of the targeted device will be synthesized based on the design files added. This chip will then be optimized by the synthesis tools. The script will complete and an output file in EDIF format is produced. In this thesis, the final output EDIF file will be always be “pilchard.edf.” The “pilchard.fst” script, modified to work with the “fft4.vhd” is shown in Figure 3.3.

If other synthesis tools are used, a few important considerations must be accounted for, such as:

1) The final output EDIF file, “pilchard.edf” must be synthesized from the top-level VHDL file, “pilchard.vhd.”
2) Unused connections detected in the “pilchard.vhd” file must not be removed.
3) Hierarchy of the designs should be preserved and not flattened.

```
# pilchard.fst
#
# This script creates an optimized chip for the Pilchard sample design.
# It is designed for the UNIX environment.
#
# To run the script:
#
#    fc2_shell -f pilchard.fst

# Define variables
#
set proj pilchard_proj
set top pilchard
set target VIRTEXE
set device V1000EHQ240
set speed -6
set chip pilchard
set export_dir .

# Remove any old version of the project,
# and create the new project
# comment out this section to work on
# an existing project

exec rm -rf $proj
create_project -dir . $proj

# to work on an existing project
# merely open it
open_project $proj

# Setup project variables
#
proj_export_timing_constraint = "yes"

# Setup default variables
#
default_clock_frequency = 133

# Identify the library source files
#create_library LIB
#add_file -library LIB -format VHDL vhdl/lib.vhd

# Identify the design source files

add_file -library WORK -format VHDL vhdl/FFTPACK.VHD
add_file -library WORK -format VHDL vhdl/STATE.VHD
add_file -library WORK -format VHDL vhdl/CONTROL1.VHD
add_file -library WORK -format VHDL vhdl/FFT1.VHD
```

Figure 3.3: The modified synthesis script for Pilchard implementation of “fft4.vhd”.

```
add_file -library WORK -format VHDL vhdl/FFT1_TOP.VHD
add_file -library WORK -format EDIF edif/ram_c_4.edn
add_file -library WORK -format EDIF edif/ram_ir_4.edn
add_file -library WORK -format VHDL vhdl/pcore.vhd
add_file -library WORK -format VHDL vhdl/pilchard.vhd

# Analyze all the source files and display the progress

analyze_file -progress

# Create a chip targetted for $TARGET with the default part and
# speed grade.  The chip will be named $chip.  $top indicates
# the top level design.

create_chip -target $target -device $device -speed $speed -module -name
$chip $top

# Set the current chip to add constraints

current_chip $chip

# Read the constraints file
#source constraints.fst

# Optimize the current chip
#
set opt_chip [format "%s-Optimized" $chip]
optimize_chip -name $opt_chip

# Show any error and warning messages for the chip
#
list_message

# Create a timing report
#
report_timing

# Write out the PPR netlist and constraints to the directory $export_dir
#
export_chip -dir $export_dir -no_timing_constraint

# Save and close the project
#
close_project

quit
```

Figure 3.3: Continued.

3.4 Place-and-Route

The place and route process can be a time consuming task as several steps and strategies have to be resolved. The place and route software chosen for this work is the Xilinx PAR Tools version 4.1i. A script was written and used to automate this step. In this script, the following commands are included to ensure that the overall process can be accomplished: ngdbuild, map, par, trce, and bitgen and fpga editor.

NGDBuild [10] is used to read the input netlist files and create NGD files that contain both the logical description of the design in Xilinx Native Generic Database (NGD) primitives and a description of the original hierarchy of the input netlist. MAP [10] is used to map the logical design to a Xilinx FPGA. TRACE [10] is used to perform static timing analysis of the design based on the specified input timing constraints. BITGEN [10] is used to produce a bitstream for configuring the targeted Xilinx device. FPGA_editor [10] is used to view the fully routed design on the FPGA device. The script used for the place and route is shown in Figure 3.4.

```
#!/bin/csh –f
ngdbuild –sd edif –uc ucf/pilchard.ucf –p XV1000EHQ240-6 pilchard.edf pilchard.ngd
map –p XV1000EHQ240-6 –cm speed –o map.ncd pilchard.ngd pilchard.pcf
par –w –ol 5 –d 2 map.ncd pilchard.ncd pilchard.pcf
trce pilchard.ncd pilchard.pcf –v 10 –o pilchard.twr
bitgen –w –l pilchard.ncd pilchard.bit pilchard.pcf
fpga_editor pilchard.ncd &
```

Figure 3.4: The place and route script.

Chapter 4

Design Stage – From IP to Hardware

4.1 Pilchard Design Flow

The design flow is a process in which a design is iteratively entered, implemented, and verified until it is correct and meets its specification. The Pilchard design flow shown in Figure 4.1 resemble to the commonly used HDL (Hardware Description Language) design flow. However, there is a minor difference in the flow, mainly the way verification is done on a HDL design. In the Pilchard design flow, the Pilchard designs are verified in two ways. The first verification is a functional verification which is done before the synthesis stage. The second verification is the in-circuit verification that is initiated after the download of the designs onto the Pilchard platform has been completed. One question that arises from Figure 4.1 is, "What about the post-layout simulation that is usually done in a HDL design flows?"

Post-layout simulation is a simulation that takes the back-annotation of a fully routed design and performs a functional simulation with timing information. The way the Pilchard design files were setup, this would be a formidable task. The details of this will be discussed in the rest of the thesis.

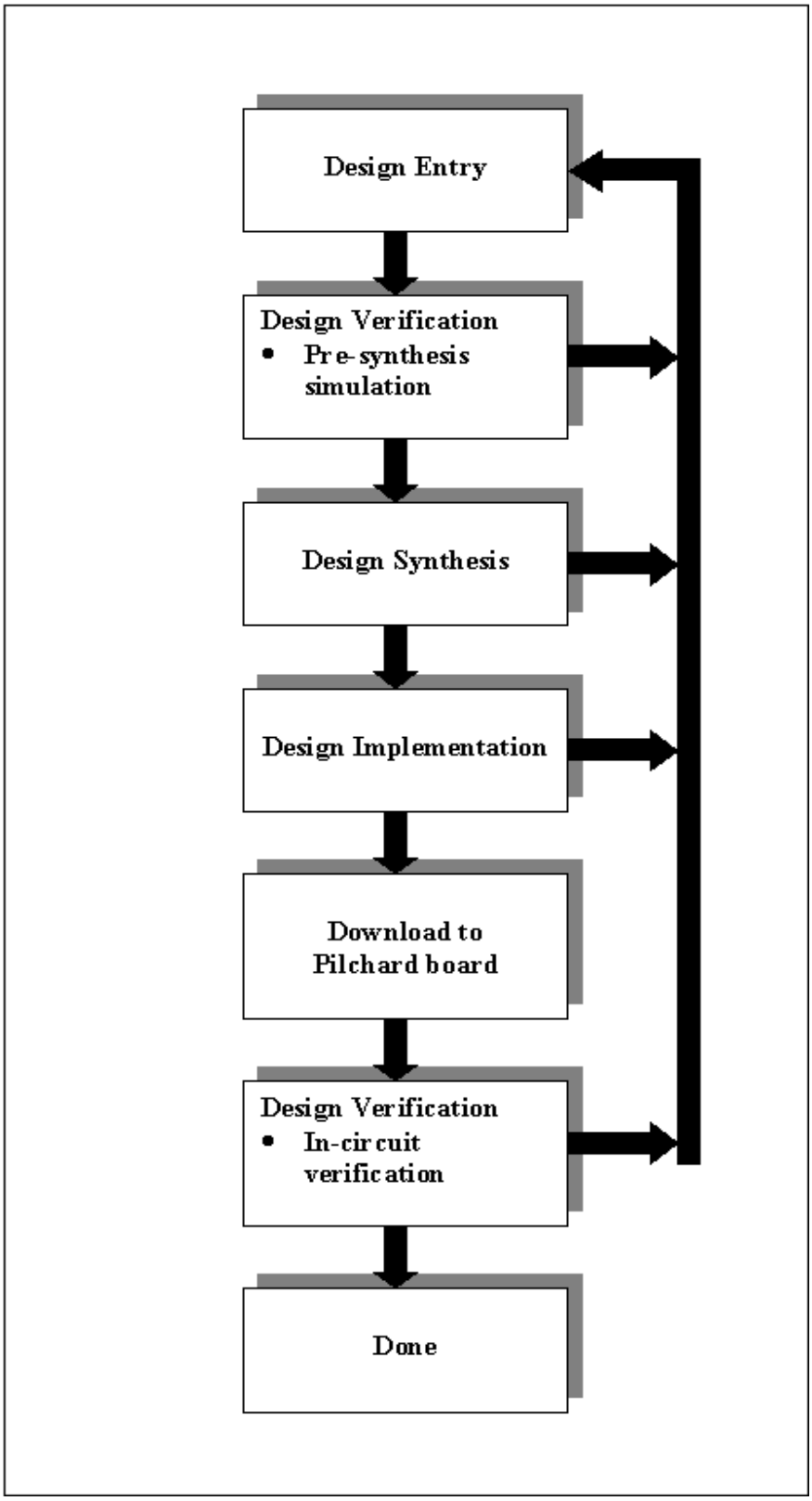

Figure 4.1: Pilchard Design Flow

4.2 Design Entry

Design entry is the first step in integrating an IP core to function in the Pilchard platform. For design entry, a designer can combine IP from several sources to work together. This is useful as it helps the designer in shortening the design time and resources used. For example, in this thesis the Fast Fourier Transform (FFT) IP needed both a dual-port RAM and a single-port RAM to be able to function. Designing and integrating these RAMs modules would have increased the design time. Instead, the RAM IP cores from Xilinx CORE Generator™ were used and the only design time involved was from integrating the IP.

As discussed in the previous chapter, the Pilchard design file, “pcore.vhd”, is used as a wrapper for the IP cores that are to be tested. Figure 4.2 shows the level in which IP cores are located when they are incorporated into the “pcore.vhd”.

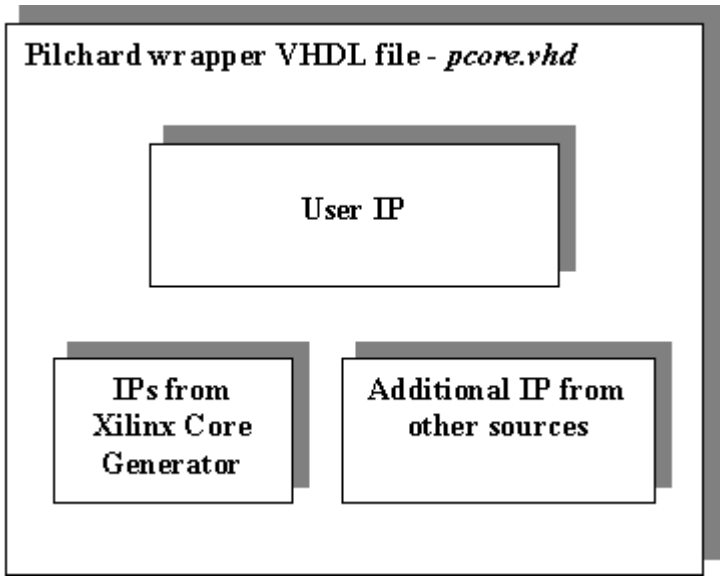

Figure 4.2: Pilchard wrapper file, “pcore.vhd”, with the individual IP cores.

As shown in Figure 4.2, the multiple IP cores are set in a hierarchical order with the top-level file being the “pcore.vhd” file. The design entry flow is shown as Figure 4.3.

During the design entry stage, the IP cores are modified to use the “pcore.vhd” ports. The author recommends the following guidelines to help with IP integration:

1. Design a FSM to control the data transfer and operations of all integrated IPs.
2. Data transfer is performed at the system clock speed. If an IP core needs to run at a lower speed, the “clkdiv” port in “pcore.vhd” can be used to provide half-clock speed. Extreme care must be taken since the user will then have to manage the two clocks.

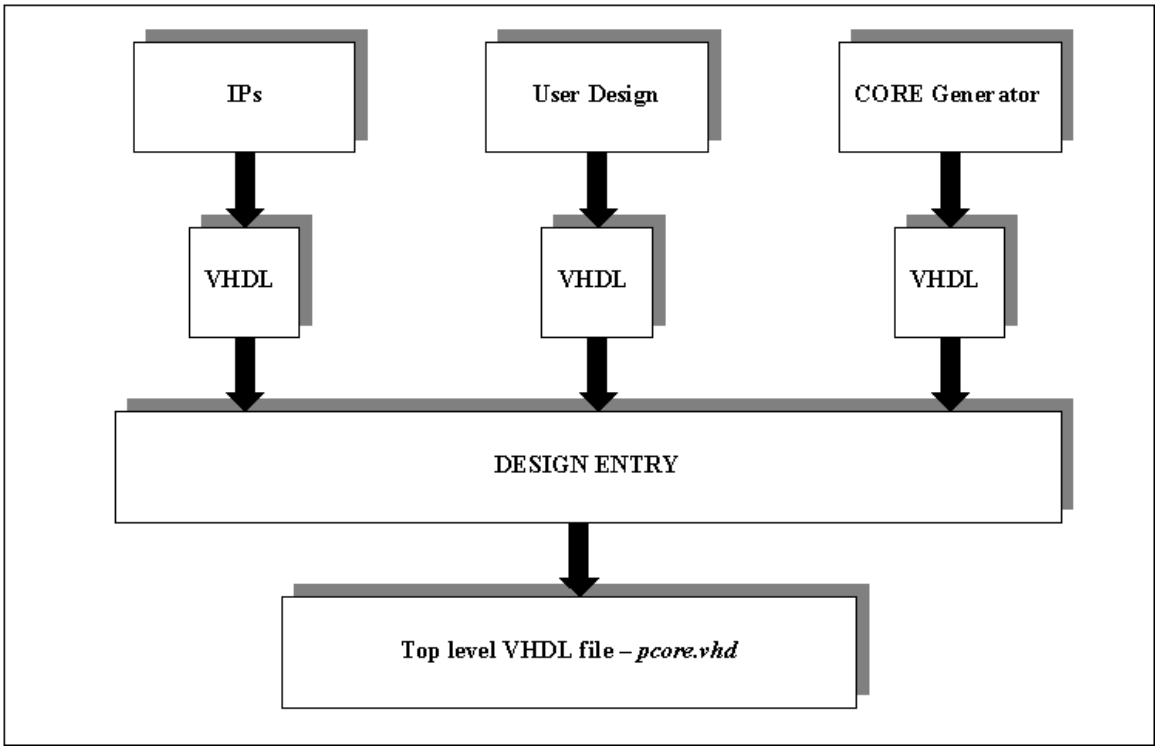

Figure 4.3: Pilchard Design Entry.

4.3 Design Verification – Pre-synthesis Simulation

Pre-synthesis simulation is the first of two design verifications in the Pilchard design process. In this stage, the functional simulation of the design is being tested to verify the correctness of the logic in the design. Since the design has not been implemented on any device, timing information is unavailable at this stage. The simulator will test the logic using unit delays. To perform the functional simulation, a testbench that simulates the behavior of the design core must be written. Designing a testbench can consume a considerable amount of time for an inexperienced designer. In response to this possible problem, a testbench that performs the basic function of testing has been written with the following functions and characteristics:

1) A Finite State Machine (FSM) with four states: Initialize, loading data to the core, start core and wait, and unloading data from the core.
2) Allows the reading of an input test vector file in TXT format.
3) Allows the writing of output files in TXT format.

Due to a port name that was used in “pcore.vhd” file, the read function that was implemented in the testbench was not working correctly when running the simulation. As this problem did not appear in synthesis and implementation of the design, an additional file labeled “pcoretb.vhd” was used to rename the ports of the “pcore.vhd”. Figure 4.4 shows the hierarchy for the input testbench files with “tb.vhd” being the top-level. Figure 4.5 shows the testbench file, and also other files and the library that are necessary for the simulator to function. This testbench is only used by the simulator and is not to be synthesized.

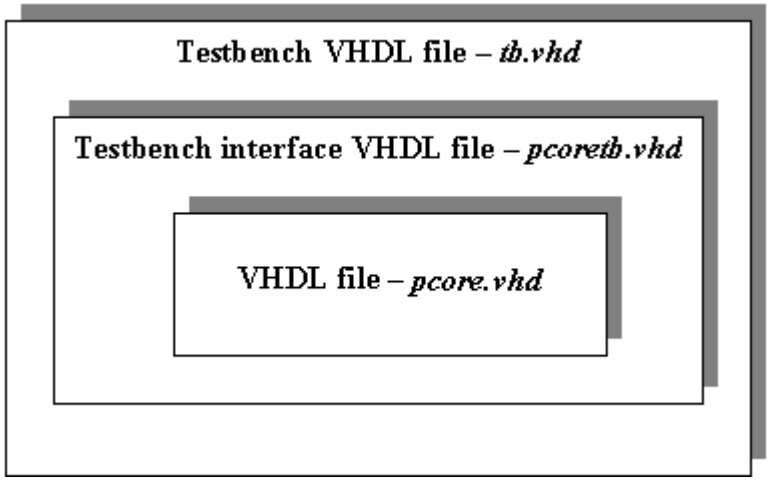

Figure 4.4: Pilchard testbench files.

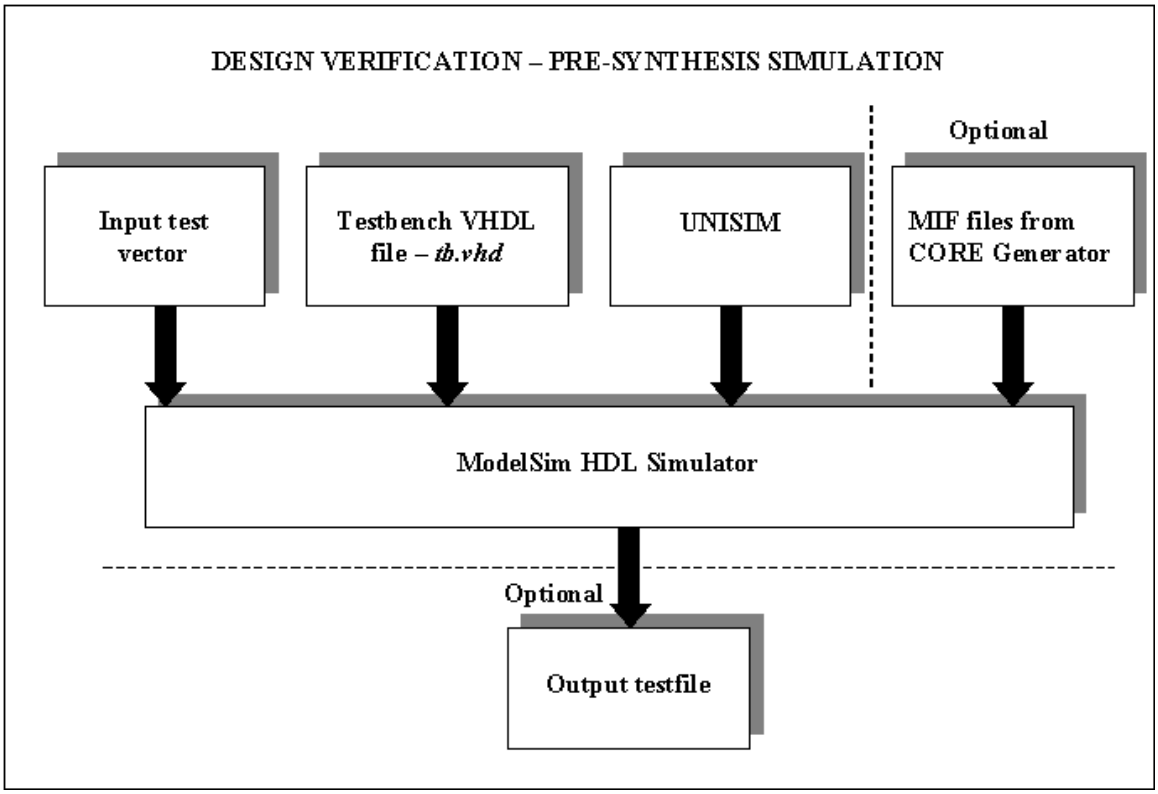

Figure 4.5: Pilchard Design Verification – Pre-Synthesis Simulation.

With the testbench file correctly integrated with the IP core, the simulator might require the XilinxCoreLib behavioral models to be included when simulating the functional behavior of certain IP. Certain IP cores may require the UNISIM library to be included. It is also required that all the VHDL designs files that were used in the IP be entered into the simulator. If the IP cores were generated by Xilinx Core Generator™, there will be a generated VHDL file of the IP and a possible Memory Initialization File (MIF). Both of these files are required to be entered into the simulator. The MIF file is used by the simulator to simulate modules, such as memories, which use arrays of values.

In the simulator, there are many ways for designers to test the functionality of their designs. In this thesis, one method is the use of an input test vector file and an output test result file. The input test vector file is a file with data vectors that causes the IP under test to produce a certain output result. These results are then recorded or viewed by the IP designer to determine if the IP is working as expected.

As an example, the FFT IP was initially simulated without incorporating it with any other IP or Pilchard wrappers. Using the testbench provided with the FFT IP, the input test vectors produced a certain output file. Using the same input test vector file, the FFT IP with the integrated RAM and Pilchard wrapper was then simulated. This Pilchard version of the FFT IP was verified for its validity as its output file corresponds to the output file generated previously from the IP standalone implementations.

4.4 Design Synthesis

After the functional verification of the design is completed, the “pcore.vhd” file, which contains the user design, is integrated into the “pilchard.vhd” file. This file is used in the design synthesis process shown in Figure 4.6. In the design synthesis stage, the “pilchard.vhd” file is translated into an EDIF file that contained the implementation netlist of the design. In addition to the “pilchard.vhd” file, all the VHDL files as well as EDIF files used in the design of the core are also required to be presented to the synthesis tool.

After the synthesis script generates the EDIF file of the synthesized design, a report with information about the synthesis process and the maximum clock frequency that the designs can be expected to run correctly is also produced.

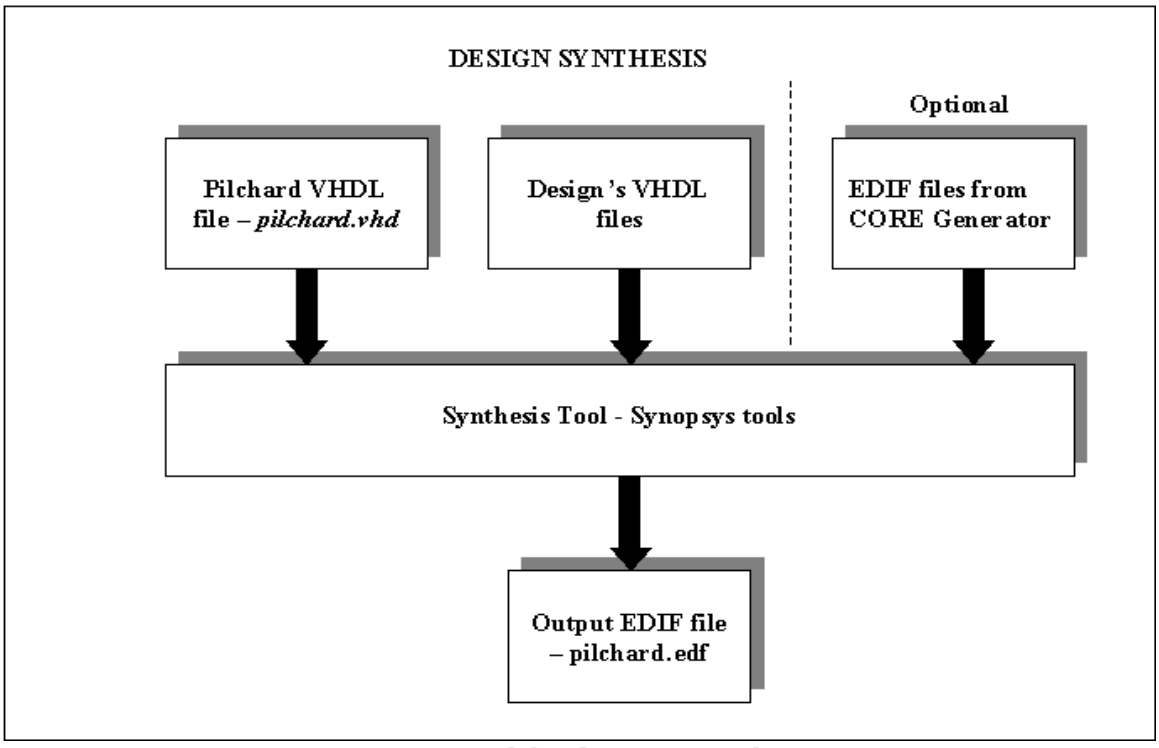

Figure 4.6: Pilchard Design Synthesis.

4.5 Design Implementation

In the design implementation stage shown in Figure 4.7, the synthesized design file, “pilchard.edf” is used for place and route (PAR). The PAR process also requires that the relevant EDIF files be included.

The Pilchard user constraint file (UCF) and EDIF library are also required to be in place at the beginning of the PAR process. The place and route script will run a series of commands to optimize, map, place, and route the design. The bit file, with the fully routed design, will be generated at the end of the script.

As the place and route process begins, the NGDBuild command line will read the input netlist files (EDIF) to create the corresponding Native Generic Database (NGD). These NGD files contained both the logical description of the design in Xilinx Native Generic Database (NGD) primitives and a description of the original hierarchy of the input netlist.

The MAP command line will read in an NGD file of the design and perform a design rule check (DRC) of the file before mapping the logic to the components in the targeted Xilinx FPGA. The user can choose to either map the design based on speed or area. In this thesis, mapping is done to optimize speed. The output file from this command is a Native Circuit Description (NCD) file and a Physical Constraints (PCF). The NCD file contains the physical representation of the design mapped to the components in the targeted Xilinx FPGA. The PCF file contains the constraints in terms of physical elements.

The PAR command line reads in the mapped NCD file and PCF file to place and route the design on the targeted Xilinx FPGA. The overall placer effort is set to the

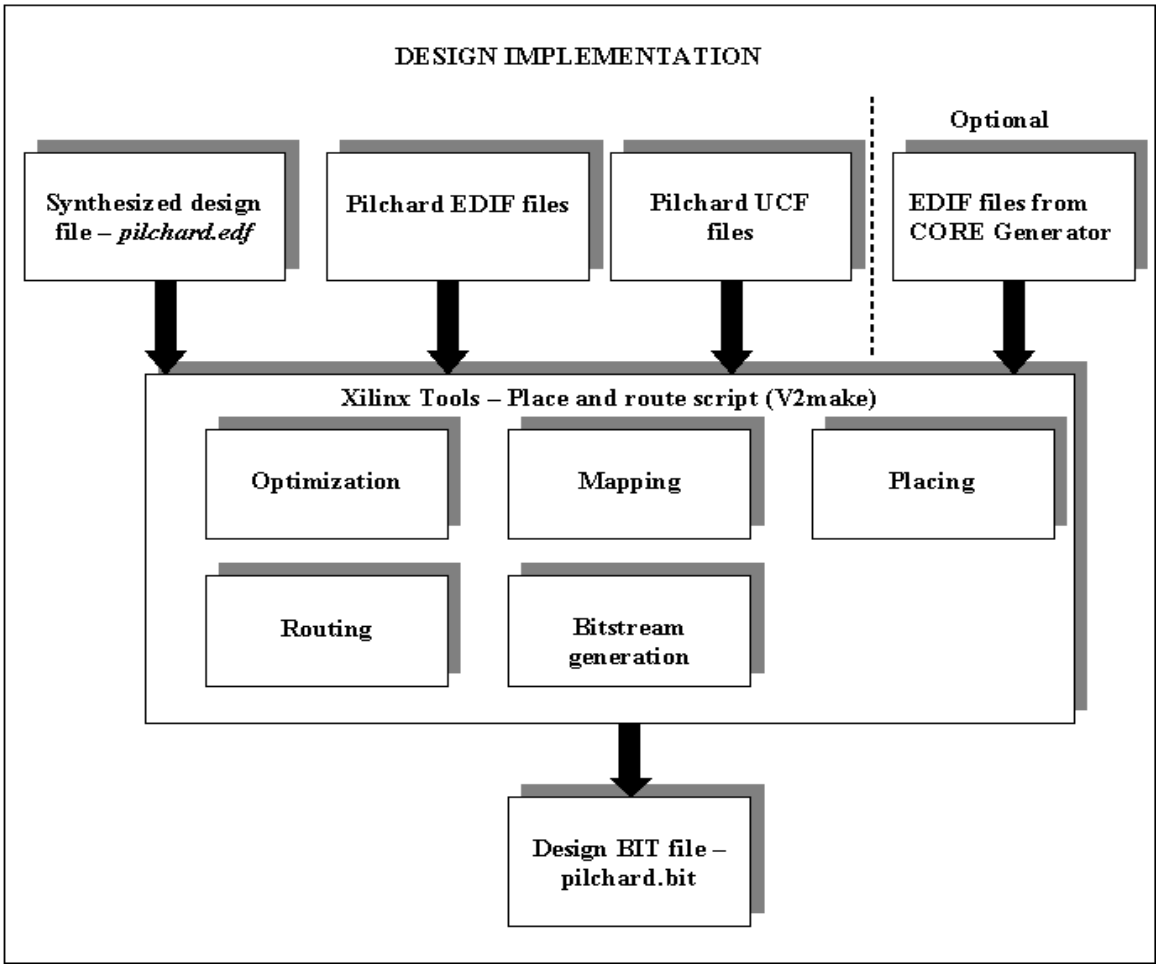

Figure 4.7: Pilchard Design Implementation.

highest level. The effort level determine the PAR effort in placing and routing of the design and also in achieving the specified timing constraints. The routing of the design is iterated twice in an attempt to reduce delays in the routed design. The output file from this command is a routed NCD file of the design.

The TRACE command line reads in a fully routed NCD file and a PCF file which contains the specified timing constraints. Static timing analysis is performed on the design. The output file from the TRACE command is a formatted timing report (TWR) file.

Finally, the BITGEN command line reads in a fully routed NCD file and generates a configuration binary file with the .bit extension. This BIT file contains the configuration information of the NCD file that is targeted to the Xilinx device. This BIT file can be downloaded into the FPGA memory cells to configure the device.

In most cases, for post simulation purposes, the command line NGDANNO is used to generate a generic timing simulation model of the fully routed design. NGDANNO produces a SDF file that must be back annotated with the FPGA netlist for post-synthesized gate-level simulation. Since the Pilchard DIMM interface in “pilchard.vhd” is also part of the synthesized EDIF file used for place and route, the resultant SDF file will have the DIMM interface “pilchard.vhd” as the top-level of the design. However, the functional simulation testbench uses the “pcore.vhd” as the top level of the design. So, the SDF file will be incompatible and a new testbench must be written. Since the timing results from the post-layout simulation can also be accomplished by in-circuit verification, the Pilchard post-layout verification is not performed. Hence, the NGDANNO command is not required in the design

implementation flow. However, compared to post-layout simulations, the in-circuit verification does not provide as much visibility of the design timing.

The place and route report generated by the PAR script provides information regarding the location of the IOBs placed, the placer effort specified, the FPGA device utilization summary, the initial timing analysis, and the total time for PAR to complete.

4.6 Downloading to Pilchard

In this stage of the Pilchard design flow, the bitstream with the fully routed design is transferred to the host computer with the Pilchard. The bitstream is then downloaded into the Pilchard board using the compiled and executable C code, "download.c". The user initiates the bitstream downloading in the Unix OS environment by typing the following command:

download pilchard.bit

In this thesis, the bitstream, input files, and the C codes were transferred via the File Transfer Protocol (FTP) to the computer with the Pilchard, which is located in the CUHK network. Figure 4.8 shows the downloading process to Pilchard.

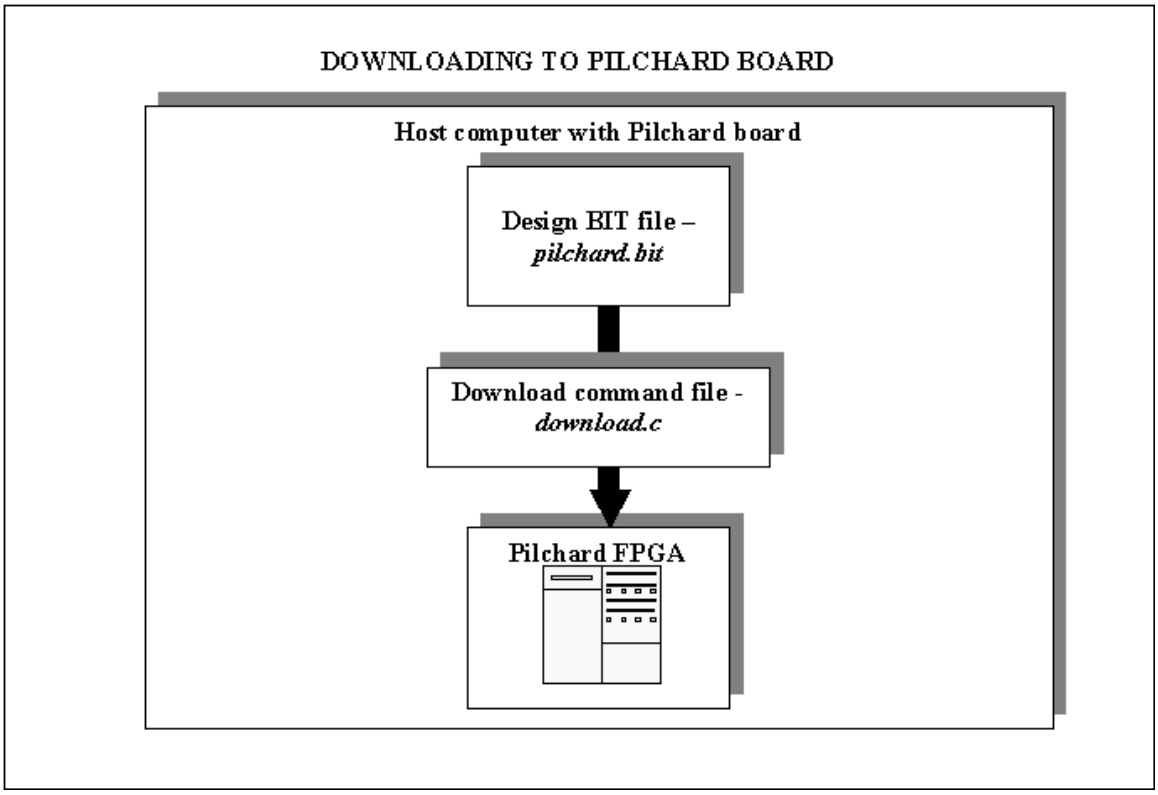

Figure 4.8: Downloading to Pilchard board.

4.7 In-Circuit Verification

With the BIT file downloaded successfully into the Pilchard, the user will need to write a C code that uses the APIs in the "iflib.h" to access their designs in the Pilchard FPGA. This C code should be able to perform exactly the same functions as the testbench used during pre-synthesis simulation. This is to enable a comparison of results between the two verification processes. As a starting point, the C code template shown in Figure 2.4 as a guideline in writing the code. Simplicity is stressed in the writing of this C codes as it will be difficult to debug any behavioral problems that might arise when the code is executed. Hence, the author suggests that the following guideline be used when writing the C code for accessing the downloaded design in the Pilchard:

1) The C code will only have three stages of operations: Loading, Computation and Unloading.
2) The input data should only be downloaded and stored in the FPGA during the loading stage. Input data must in integer format.
3) The computation stage will start the operation of the IP in the downloaded design. No data transfer should occur between the host and the Pilchard during this stage. The computation stage should be able to detect certain status flag signals from the IP to indicate that the final output data is ready.
4) The unloading stage will upload the output data from the FPGA.
5) If functional simulation were done using the proposed testbench during pre-synthesis simulation, the C code should be written to mimic the testbench operation.

Chapter 5

Implementation

In the implementation of the design flow proposed by the author, two IP cores are used as prototype examples. Each of the IP cores is obtained from a different source. Some requirements about the chosen IPs have to be known by the user of the Pilchard board. This is to help the user in integrating the IP as well as in verifying the results obtained after its implementation on the FPGA. The application intent, functional description, testbench and HDL models of each IP core used will be explained in this chapter of the thesis. This chapter of the thesis will also explain the specific preparation made to each IP core. The VHDL codes of the IP cores are included in the Appendices.

5.1 IP Core – iftest.vhd

The first IP core that will be tested is the “iftest” IP core that was included in the Pilchard design package. The purpose of this IP core is to help first-time users in familiarizing themselves with IP integration into the Pilchard system The IP core is coded in VHDL and is written by the Pilchard board design team [7]. Since the IP was designed using the “pcore.vhd” template shown as Figure 2.4, the filename for the core is “pcore.vhd”. A testbench for functional simulation and a C code to access the design after it is downloaded into the Pilchard were provided with the “iftest” IP core. For verification, a

text file of the valid output from the implemented design on the Pilchard is also provided.

5.1.1 Application Intent

The application intent of this IP core was to test the interface of the Pilchard board and the host. The core performs a series of write operations from the host to the Pilchard and a series of read operations from the Pilchard to the host. This core also shows how to alter the state of the FSM in an IP core by the host.

5.1.2 HDL Models

The modules used in the IP are shown in Figure 5.1. Two memory modules are initiated in the IP. The memory modules are: 1) a single port BlockRAM that function as a 16-bit width memory with 256 addresses, and 2) a dual port BlockRAM that function as a 16-bit width memory with 256 addresses for each port. A FSM that controls the functional behavior of the IP core is embedded in the IP itself.

5.1.3 Functional Behavior

The functional behavior of the “iftest” IP is determined by the three states of its FSM. In the VHDL code, the three states are labeled as INI, ST1 and ST2. When the IP starts, the first state, INI, is setup to advance to the second state, ST1. While in ST1 state, if a write operation in memory address 255 or 0xFF is detected, the state will alter from ST1 to ST2. While in ST2 state, if a write operation is detected in memory address 255 or 0xFF, the state will alter from ST2 to ST1. The FSM is shown as Figure 5.2.

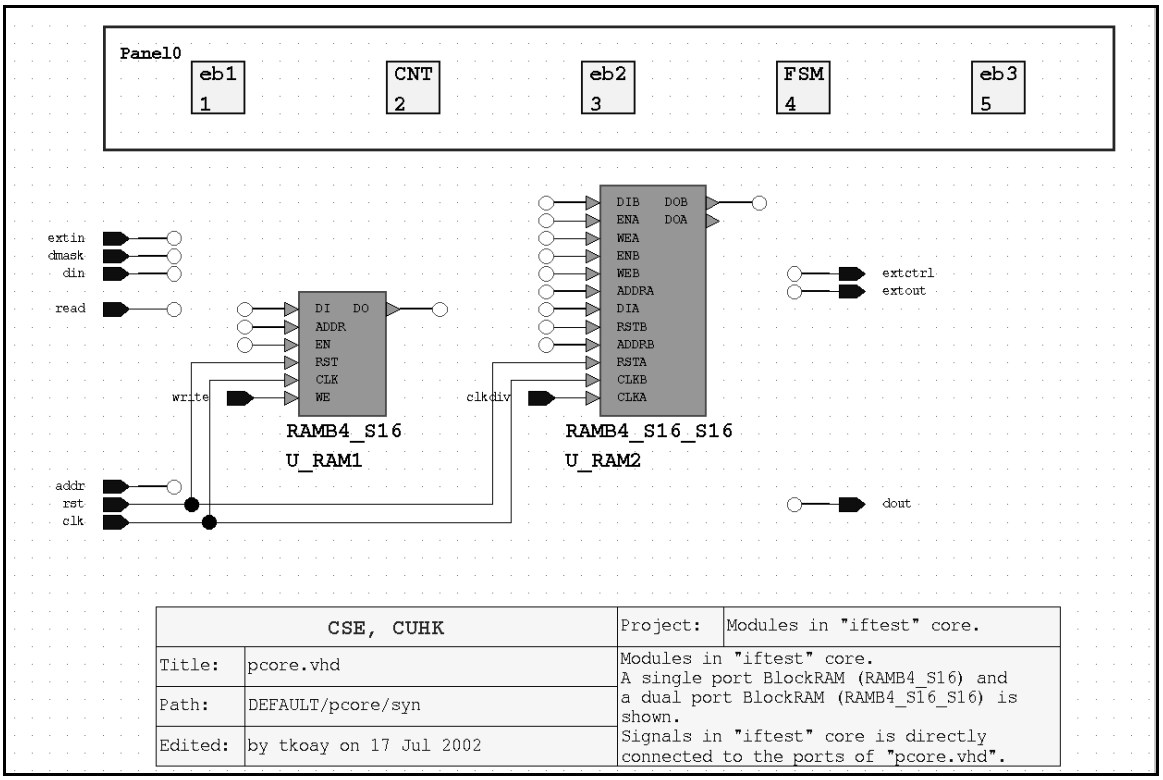

Figure 5.1: The modules used for the “iftest” core.

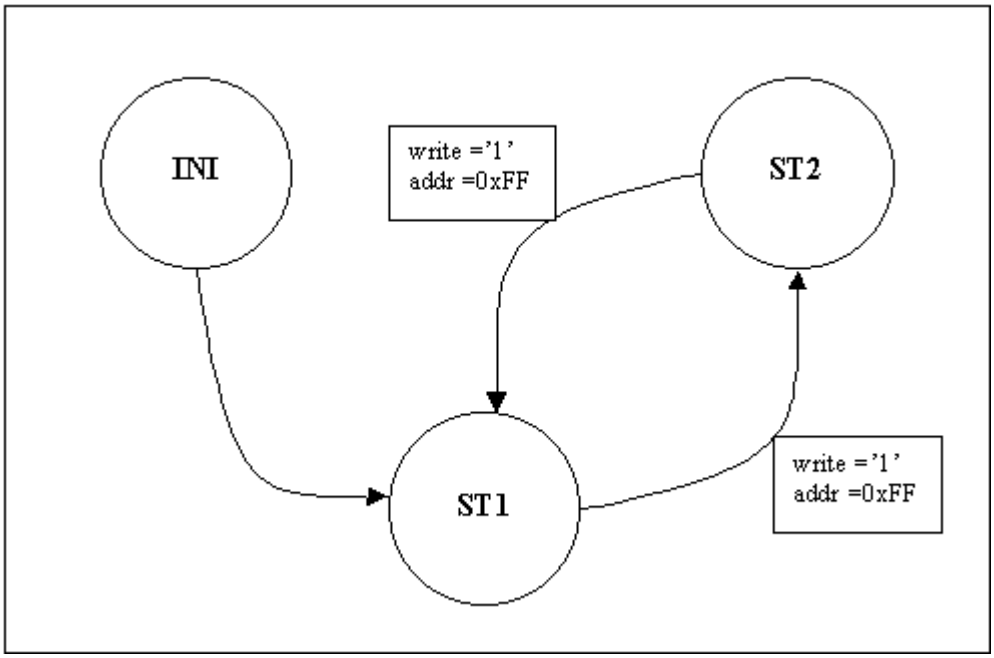

Figure 5.2: The FSM of the “iftest” core.

5.1.4 Implementation Using Pilchard Design Flow

The IP core is design specifically to be implemented on the Pilchard. As most of the interface as well as the timing of the states are already in place, the IP core is ready for pre-synthesis simulation. System clock is 100MHz.

For pre-synthesis simulation, a simple testbench was used to check if the FSM in the "iftest" core was actually working correctly. The input test vectors for the simulator are generated by a counter in the testbench. The UNISIM library was required as the "iftest" module uses two BlockRAMs. The waveforms acquired from the ModelSim are shown in Chapter 6 as Figure 6.1 and Figure 6.2.

During synthesis, the synthesis script was modified to read in only the "pcore.vhd" file and the pilchard.vhd" file. The routed design is downloaded into the Pilchard board.

For verification of the "iftest" design on the Pilchard, the following parameters must be tested:

1) The data transfer from the host computer to Pilchard is correct.
2) The counter in the "iftest" core is enabling at the correct state.
3) The states in the FSM must alter when a write operation is executed at memory address 255.

The provided C code was written to test all the mentioned parameters. The results obtained from the execution of this C code are verified against the valid output in the text file provided. The C code is included in Appendix.

5.2 IP Core – FFT4

The second IP core that was tested is the Fast Fourier Transform IP core from Honeywell Inc [11]. The FFT4 IP core is a one-dimensional radix-2 fixed point FFT that uses 16-bit complex data. The core is actually a parameterized FFT that can be scalable to compute 2^n data points, with n being an integer from zero. In this thesis, the IP core is scaled to use 4 data points. The IP core is coded in VHDL and a software implementation of the FFT is coded in "C". The VHDL filename for the core is fft4.vhd. Included also with this IP core, are three sets of data points in text format and a "C" file that generate the required twiddle factor that will be used in the FFT algorithm.

5.2.1 Application Intent

The application intent of this IP core was to test the scalability of FPGA platform. The IP core has three sets of data points that serve to measure the performance of the FPGA platform when dealing with small data (512), medium data (4096), and large data (16384). The chosen data points are initially input into a RAM that is accessible by the FFT core before the core is started. After the core starts, the Cooley Turkey FFT algorithm [11] is performed and the results of the computation is written back to the same RAM, overwriting the initial inputs. Once the core completes, the results of the FFT computation can be read out from the RAM for verification. However, the bit reversal is not implemented by this IP core.

5.2.2 HDL Models

The modules used in the IP are shown in Figure 5.3. The IP consist of three main modules: “control1”, “state”, and “fft1”. The “state” module is the FSM of the IP core. The “control1” module determines the appropriate data and memory address based on the state of the FSM. The “fft1” module performs the FFT computation based on the data presented and the state of the FSM.

5.2.3 Functional Behavior

The functional behavior of the FFT4 core is controlled by its FSM. The FSM is shown as Figure 5.4. The FSM consist of twelve states. The first three states passed the correct memory contents to the “fft1” module. The “read_data0” and “read_data1” states pass the input data while the “read_weight” state passes the current twiddle factor being used.

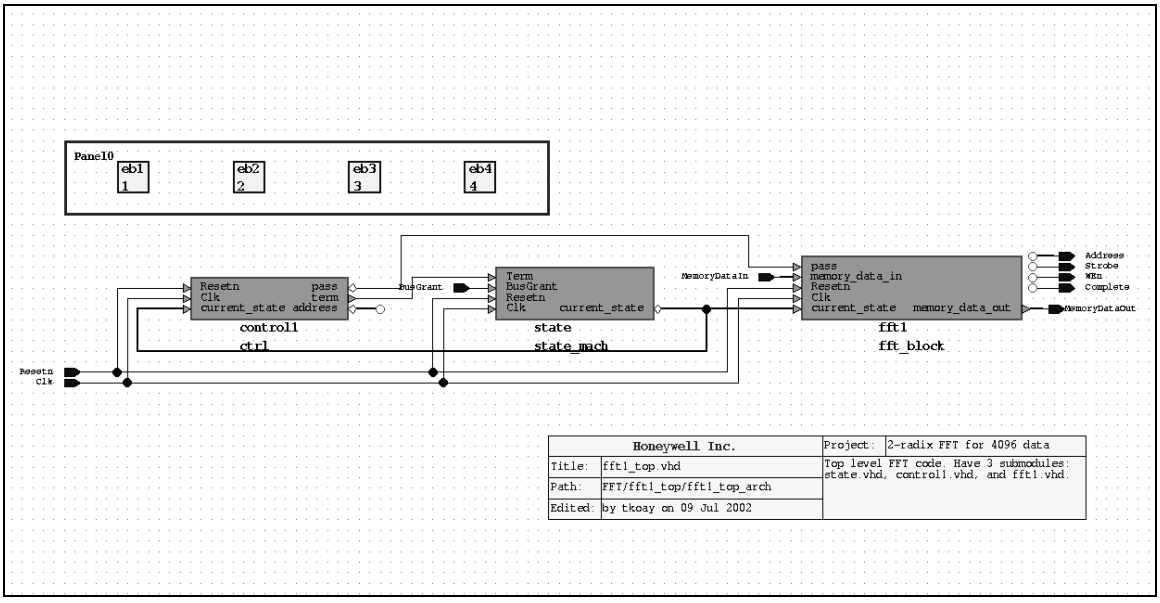

Figure 5.3: The modules in the FFT core.

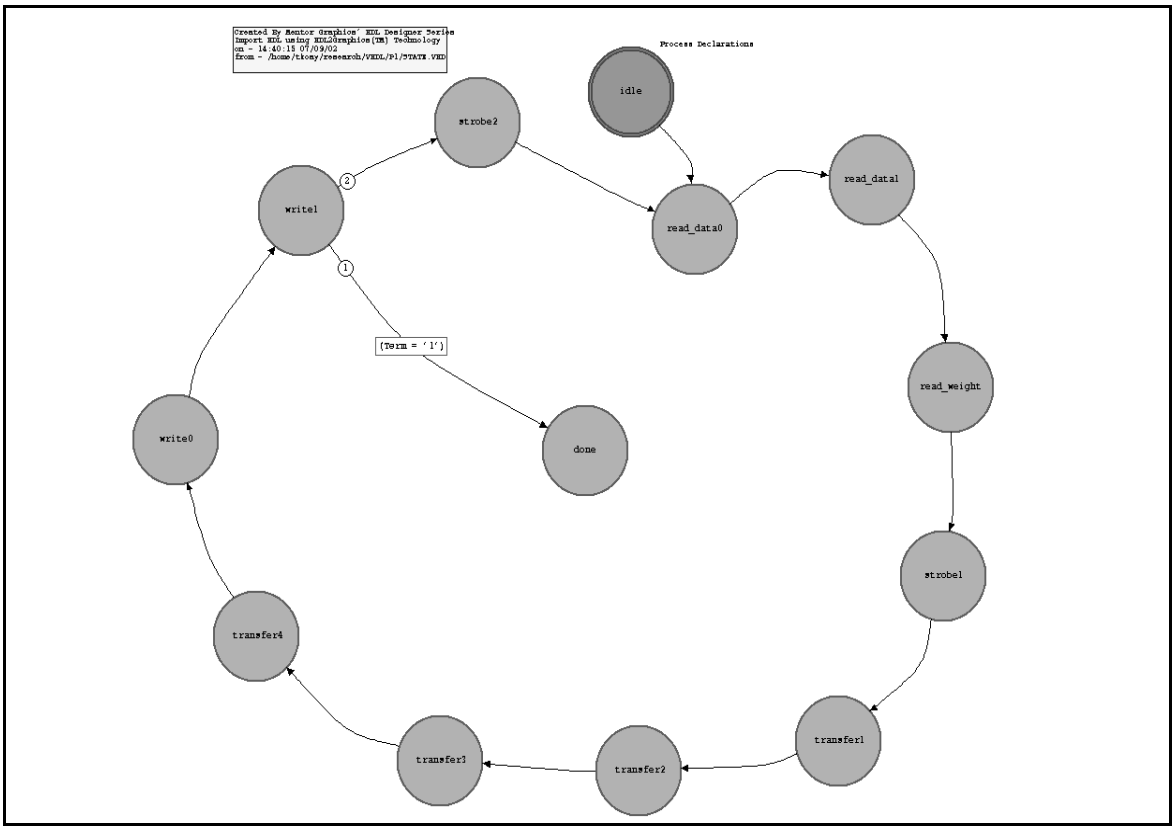

Figure 5.4: The FSM of the FFT core.

The data passed by these three read states, are expected to be ready at the input of the "fft1" module after two clock cycles. The four transfer states are the FFT "butterfly" arithmetic operation of the individual complex data using the twiddle factor. Once completed, the two write states will write the results of the FFT into the RAM.

5.2.4 Testbench

A testbench file, "dut.vhd", was also provided to test the FFT IP core. The testbench file uses a non-synthesizable RAM module to store the selected data points and appropriate twiddle factors. This testbench is used solely by the simulator and is not synthesized. In this thesis, the output generated by this testbench is used for verification of the results

that are obtained from the functional simulation and in-circuit verification stages. The graphical representation of the testbench file is shown as Figure 5.5.

Using the testbench and the four input data provided, an output file is generated. The waveform showing the first eleven states of the IP core after it was started is shown as Figure 5.6. The waveform showing the last eleven states of the IP core is shown as Figure 5.7. The significance of these two figures is that it shows the exact input and output data at the specific state of the FFT state machine. This information can assist the designer in integrating the FFT core into the Pilchard wrapper file, "pcore.vhd".

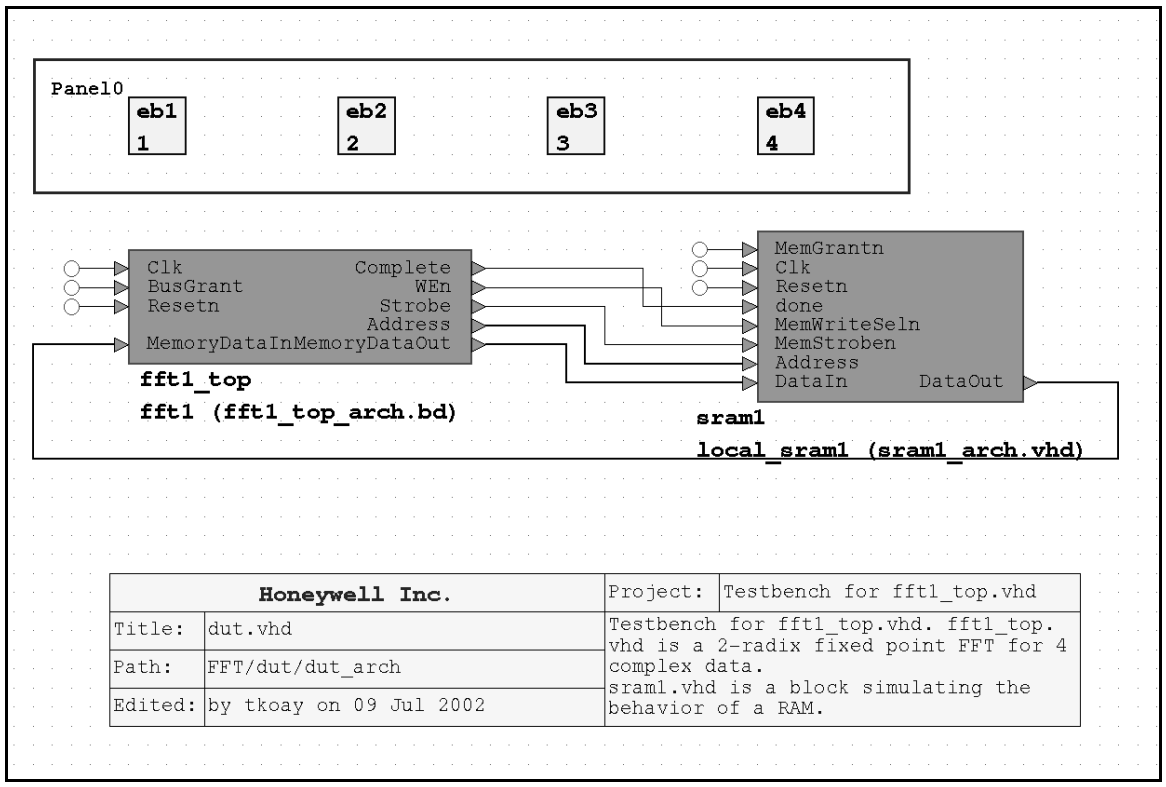

Figure 5.5: The testbench, "dut.vhd", provided with the FFT core.

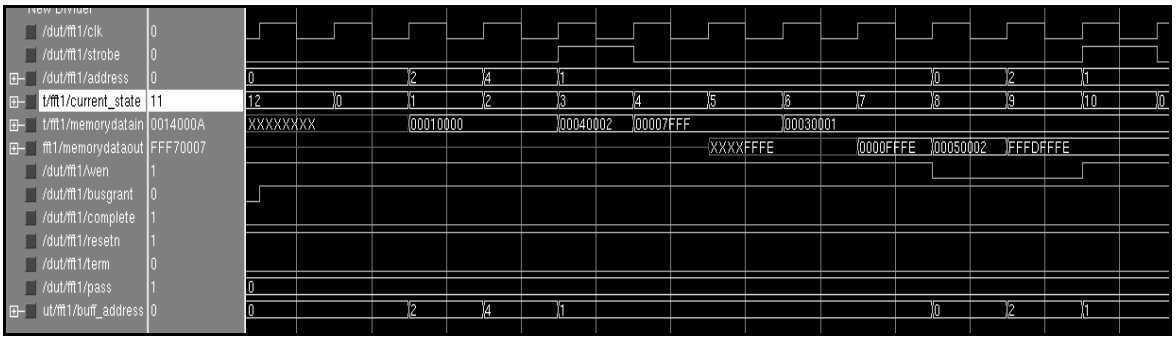

Figure 5.6: The first eleven states of the FSM in the FFT core, using “dut.vhd”.

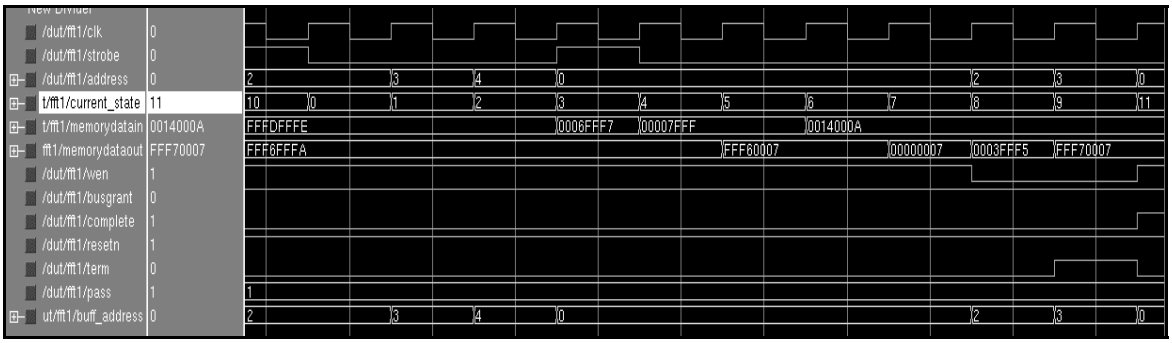

Figure 5.7: The last eleven states of the FSM in the FFT core, using “dut.vhd”.

5.2.5 Implementation using Pilchard Design Flow

During design entry, the FFT core is scaled to operate on four data points. The FFT core is also set to operate at half the system clock. System clock is 100MHz. As the FFT algorithm is intended to work with data points from a RAM, the Xilinx CoreGen™ is used to generate the necessary RAM IPs. Two RAM IPs are generated: 1) a 32 bit-width dual-port synchronous RAM with four address, and 2) a 32 bit-width single-port synchronous RAM with two memory address.

The dual-port RAM is used to store the input four data and the output four data that will be produced by the FFT4 core. Since both ports of a dual-port RAM access the same memory space, port B is designated to be used solely for data transfer between the host and RAM, while port A is used for data transfer between the RAM and the FFT core. Port B is set to operate at the system clock and port A is set to run at half the system clock.

The single-port RAM is used to store the two twiddle factors used by the FFT. Since the twiddle factors do not change through the entire FFT process, the two twiddle factors are initialized as the RAM content during its generation by CoreGen™. This causes the RAM module to function like a ROM. This RAM is set to operate at half the system clock.

Data transfer of both RAMs is arranged with bit 31 to 15 as the real numbers and bit 16 to 0 as the imaginary number. When connecting to the FFT core, the data buses of the RAMs are rewired as the FFT core deals with the most significant 16 bits as imaginary number and the least significant 16 bits as real numbers.

After the FFT core and RAMs are wrapped with the “pcore.vhd”, a FSM was written to control the operation of the IPs. This FSM, which operates at half the system clock, has four states: INI, ST1, ST2 and ST3.

During the INI state, the IP is initialized before the proceeding to state ST1. In this state, port A of the dual-port RAM is disabled, as there is no data transfer between the RAM and FFT core. Port B of the dual port RAM is enabled and is set for the write operation. The FSM of the FFT core is idle.

The ST1 state is for data loading. In this state, the input data points are written into the dual-port RAM from the host computer. For this purpose, only port B is enabled for the write operation. The FSM of the FFT core is idle. When the last data have been written into the RAM, the state will change to ST2.

The ST2 state is a computation state. In this state, the data from the dual-port RAM are accessible by the FFT core. No data transfer between the RAM and the host computer occur in this state. Hence, port B is disabled and port A is enabled. The FFT core will determine the read and write operation of port A. The single port RAM is enabled at this state. The FSM of the FFT core is started. When the “Complete” port of the FFT core goes high, the state will change to ST3.

The ST3 state is for data unloading. The data in the dual-port RAM is read out by the host computer. No data transfer between the RAM and FFT core occur during this state. Therefore, port B is enabled for read operation and port A is disabled. The FSM of the FFT core is idle. If a reset is detected, the state will reset back to INI.

The “pcore” module with the FFT core, RAMs and FSM is shown as Figure 5.8. This “pcore.vhd” file is included in the Appendix.

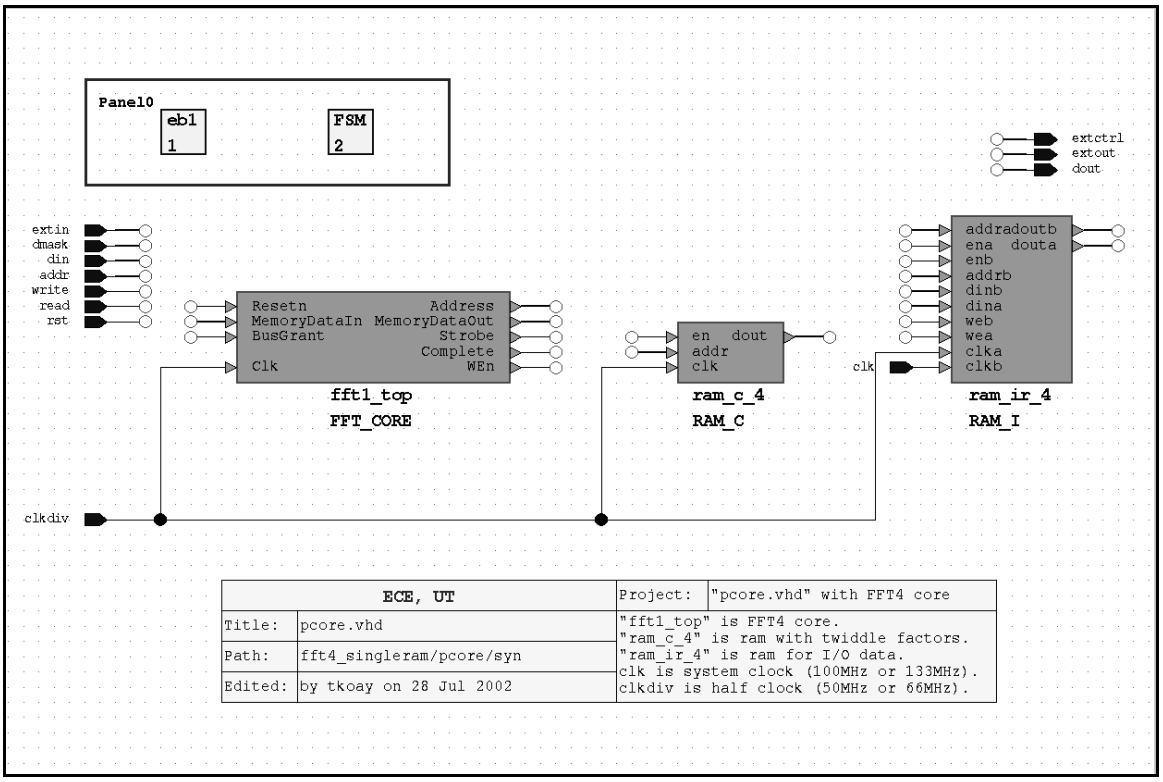

Figure 5.8: The IP cores integrated into the “pcore” module

For functional simulation, a testbench that uses the guideline suggested by the author in Chapter 4 was used. The input data is read into the testbench as an input file and the output data is recorded in an output file. This output file is then compared to the output data obtained using the “dut.vhd” testbench provided with the FFT core.

For in-circuit verification of the downloaded design, a C code that emulates the behavior of the testbench used during the functional simulation was created. As was suggested in Chapter 4, this C code will only perform three functions after it accesses the Pilchard board. The C code will load the input data into the Pilchard board. Then it will wait for the Pilchard to send a signal indicating that the output data is ready. After receiving this signal, the C code will proceed to unload the data from the Pilchard to the host computer. The C code will also perform a timestamp when the FFT core starts and

when the output data is ready to be unloaded. This is to determine the amount of time that is required for the Pilchard-implemented FFT to complete its operation.

During in-circuit verification, it was discovered that the downloaded design was not working correctly. Since the functional simulations of the IP cores were successful, it is hypothesized that the timing errors in the PAR might possibly be the reason that the implemented design was not working. To address the timing errors caused by the delays in the data paths of the routed designs, a designer has the choice of either minimizing the data path delays or increasing the clock period use by the FFT core.

The method of increasing the clock period was chosen, as it was easier to implement. As the FFT core was already using the slowest clock signal in the "pcore.vhd" file, a new clock port had to be created. This new clock port supplies a clock period that is four times longer than the system clock. The original Pilchard VHDL wrapper files, "pilchard" and "pcore", had to be modified to add this additional clock port. The modified VHDL codes are shown in the Appendix. The modified "pcore" module with the FFT core, and RAMs running on the new clock source is shown as Figure 5.9.

Using this new clock source as the clock for the IP cores, the FFT4 is reintegrated through the entire Pilchard design flow. When the design was place and routed at the design implementation stage, it is found that all timing constraints were fulfilled. Downloading the design into Pilchard was successful and the in-circuit verification shows that the downloaded FFT4 was running correctly in Pilchard.

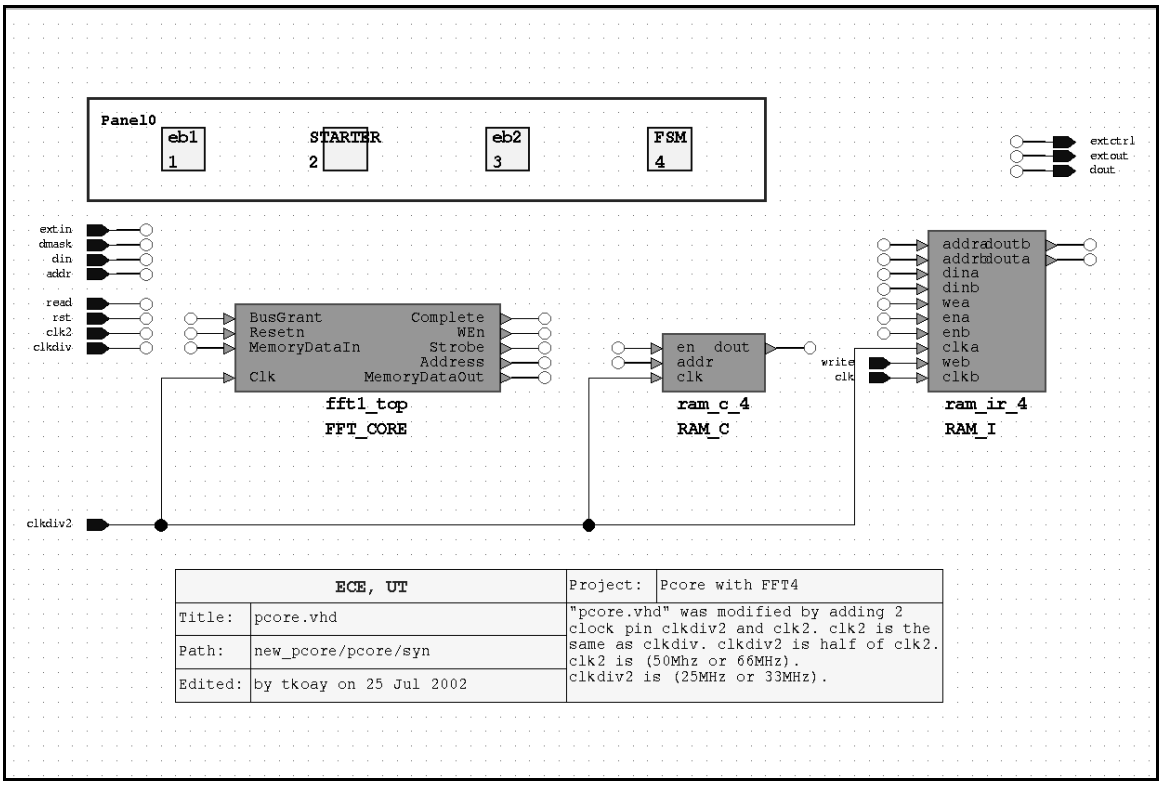

Figure 5.9: The IP cores in “pcore” modules running on a new clock source.

Chapter 6

Results and Discussion

6.1 Implemented IP core – iftest.vhd

The functional simulation of “pcore” module with the “iftest” IP core is shown as Figure 6.1 and Figure 6.2. From both Figure 6.1 and 6.2, it is clearly shown that the state of the core changes from ST1 to ST2 a clock cycle after a write occurred in memory address 255. It can also be observed in the figures that the counter in the core starts counting at ST2 and stops when the state changes back to ST1.

For the in-circuit verification, the downloaded design was successfully accessed in the Pilchard. In-circuit verification indicated that the design was operating correctly in Pilchard. Part of the output data obtained can be seen as Figure 6.3.

Timing errors were not a major issue in this implementation as the logic involved in this “iftest” core are minimal due to the simplicity of the core function. This “iftest” core implementation demonstrated that a simple IP core could be integrated into the Pilchard by a user with basic knowledge of VHDL codes and Pilchard functionality. However, if a complex IP core, like the FFT, is used, the timing issue can become a major problem. This problem is made difficult if the IP core is not optimized to run at a sufficiently high clock frequency. This was shown in the FFT4 implementation.

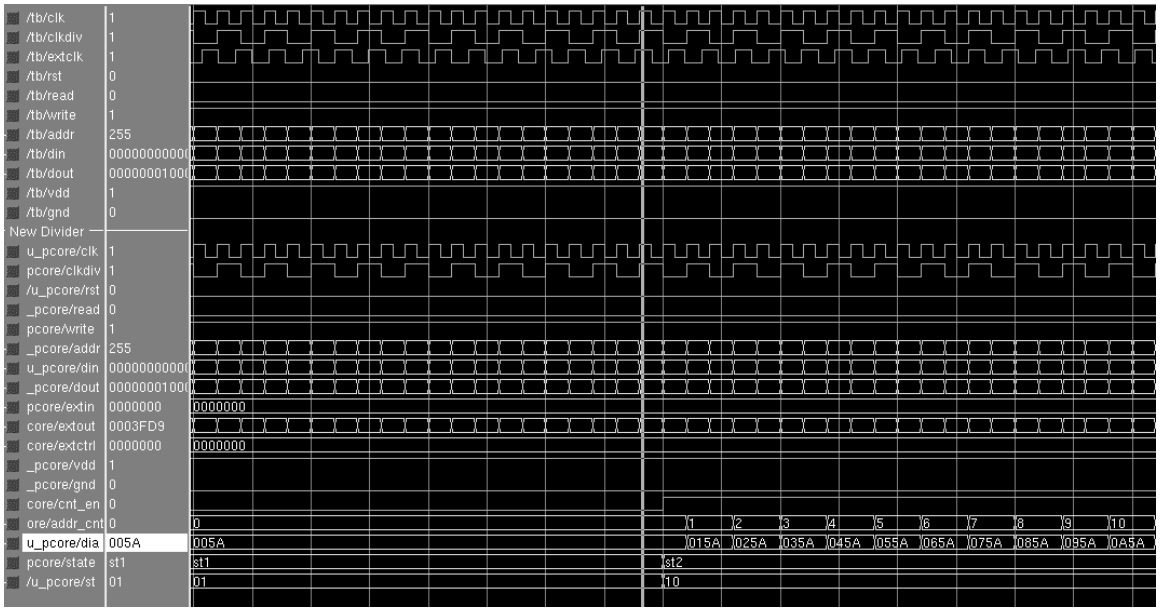

Figure 6.1: Functional simulation of “iftest” showing state ST1 change to ST2.

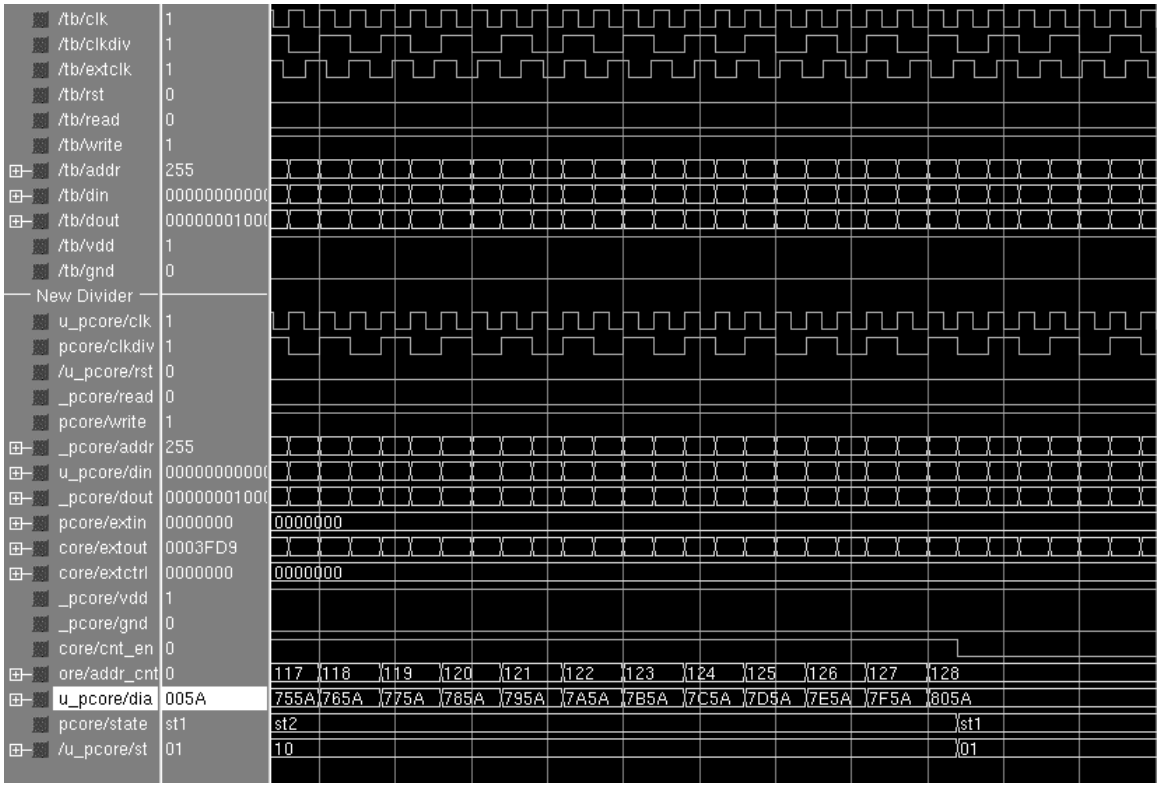

Figure 6.2: Functional simulation of “iftest” showing state ST2 change to ST1.

```
before ini state = 1
after ini state = 1
005A0000 015A0101 025A0202 035A0303 045A0404 055A0505 065A0606 075A0707
085A0808 095A0909 0A5A0A0A 0B5A0B0B 0C5A0C0C 0D5A0D0D 0E5A0E0E 0F5A0F0F
105A1010 115A1111 125A1212 135A1313 145A1414 155A1515 165A1616 175A1717
185A1818 195A1919 1A5A1A1A 1B5A1B1B 1C5A1C1C 1D5A1D1D 1E5A1E1E 1F5A1F1F
205A2020 215A2121 225A2222 235A2323 245A2424 255A2525 265A2626 275A2727
285A2828 295A2929 2A5A2A2A 2B5A2B2B 2C5A2C2C 2D5A2D2D 2E5A2E2E 2F5A2F2F
305A3030 315A3131 325A3232 335A3333 345A3434 355A3535 365A3636 375A3737
385A3838 395A3939 3A5A3A3A 3B5A3B3B 3C5A3C3C 3D5A3D3D 3E5A3E3E 3F5A3F3F
405A4040 415A4141 425A4242 435A4343 445A4444 455A4545 465A4646 475A4747
485A4848 495A4949 4A5A4A4A 4B5A4B4B 4C5A4C4C 4D5A4D4D 4E5A4E4E 4F5A4F4F
505A5050 515A5151 525A5252 535A5353 545A5454 555A5555 565A5656 575A5757
585A5858 595A5959 5A5A5A5A 5B5A5B5B 5C5A5C5C 5D5A5D5D 5E5A5E5E 5F5A5F5F
605A6060 615A6161 625A6262 635A6363 645A6464 655A6565 665A6666 675A6767
685A6868 695A6969 6A5A6A6A 6B5A6B6B 6C5A6C6C 6D5A6D6D 6E5A6E6E 6F5A6F6F
705A7070 715A7171 725A7272 735A7373 745A7474 755A7575 765A7676 775A7777
785A7878 795A7979 7A5A7A7A 7B5A7B7B 7C5A7C7C 7D5A7D7D 7E5A7E7E 7F5A7F7F
805A8080 815A8181 825A8282 835A8383 845A8484 855A8585 865A8686 875A8787
```

Figure 6.3: Part of the Pilchard results of “iftest”.

6.2 Implemented IP Core – FFT4

The functional simulation of the implemented “fft4.vhd” core is shown as Figure 6.4 and Figure 6.5. Figure 6.4 shows the first 11 states of the FFT core at the beginning of the “pcore” computation stage. Figure 6.5 shows the 11 states of the FFT before the “pcore” complete its computation stage. The format of the input file is a two-array, signed integer, four data of 16-bit complex numbers. The output data generated from the functional simulation is written to an output file as two-array, signed integer, four data of 16-bit complex numbers.

Verification of this output file is done by comparing the data in this output file to the output data from the “dut.vhd” testbench. By using the same input data in both

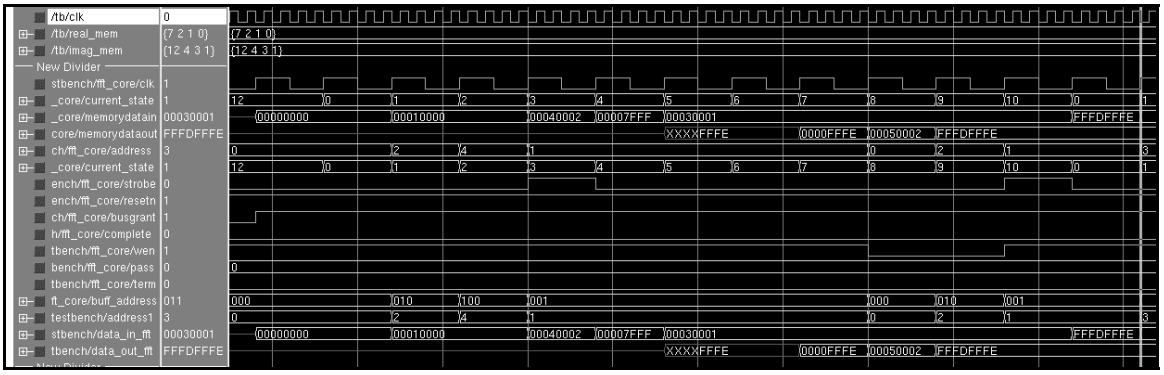

Figure 6.4: Functional simulation of “pcore” showing first 11 states of FFT4 core.

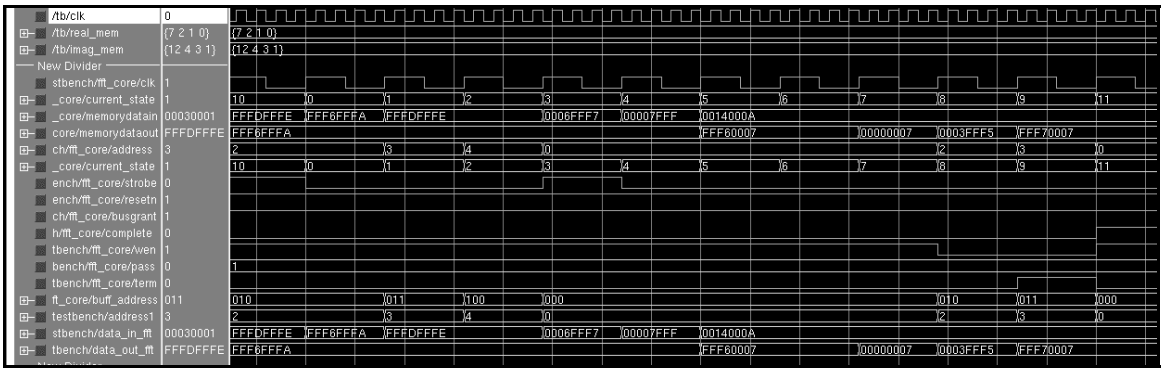

Figure 6.5: Functional simulation of “pcore” showing last 11 states of FFT4 core.

testbench, all four data points in the two output files are found to be identical. Hence, the functional behavior of the “pcore” module is verified to be correct.

For in-circuit verification, the downloaded design was successfully access in the Pilchard. The output data was verified to be valid. The time for the implemented FFT4 to complete its operation is found to be 39 microseconds.

The FFT core was scaled to different number of input data to determine the time the core complete its operation in the Pilchard. The FFT core was scaled to operate on 8, 64, 512, and 4096 data points. All the FFT cores were implemented using the Pilchard design flow. During place and route, the router tools found that the FFT512 and FFT4096 cores might have data paths that are not meeting the timing constraints.

All the FFT cores are verified during pre-synthesis and in-circuit verification. Since the functional simulation uses unit time delay instead of actual time delay, the execution time of FFT core during pre-synthesis simulation is ignored and only the validity of the outputs data is important. However, both execution time and validity of output data are important during in-circuit verification. The validity of the output data and the FFT core execution time are recorded in Table 4. The graph of the execution time is shown as Figure 6.6.

Table 4: The output results from functional and in-circuit verification

FFT Core	**Functional simulation**	**In-Circuit Verification**	
Input Data Points	**Output Data**	**Output Data**	**Time (microsecond)**
4	Valid	Valid	39
8	Valid	Valid	39
64	Valid	Valid	77
512	Valid	Invalid	1004
4096	Valid	Invalid	10779

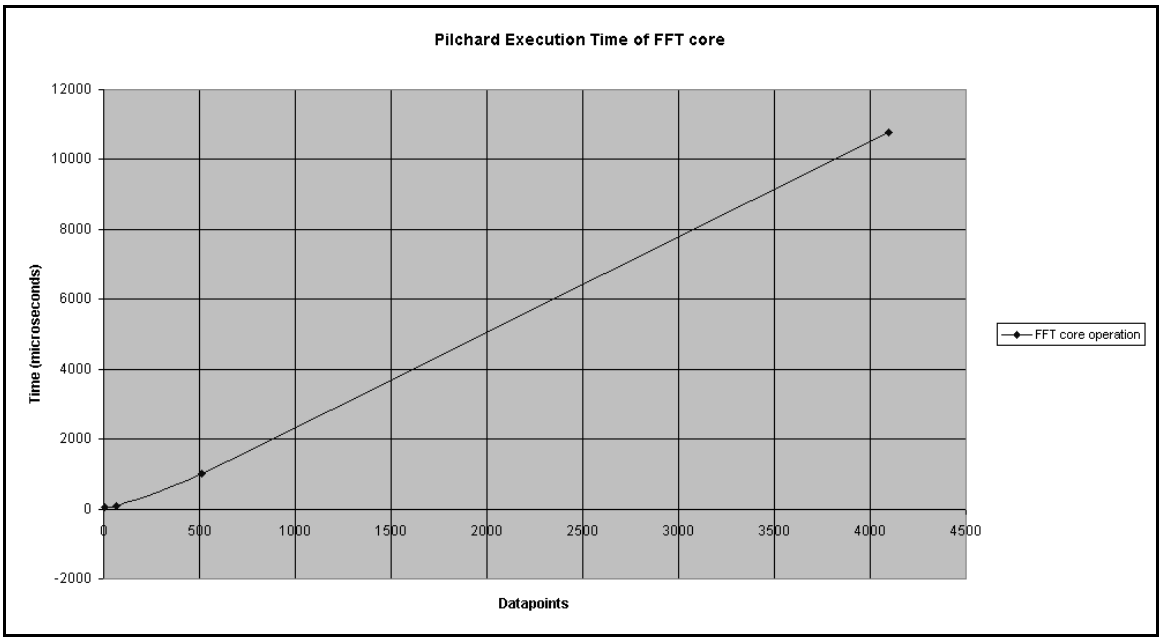

Figure 6.6: Graph of the Pilchard execution time of the FFT core.

As was shown Table 4, the output results from the functional simulation of all the FFT cores were verified to be correct. However, during in-circuit verification the possibility of the output data being incorrect was increased as the FFT cores were scaled to handle a larger number of input data. This was shown by the FFT512 and FFT4096 implementation.

As there were timing errors detected during the routing of FFT512 and FFT4096 cores, it is believed that these timing errors causes the output data of the both cores to be invalid when implemented on Pilchard. Since a slow clock was already used to fix this problem, any further improvement to reduce these timing errors in the data paths would have to be done by manually floorplanning the specific data paths to achieve a better timing delay. The effect of manual floorplanning on the data path delay is left to be determined by those who will continue this work in the future.

Chapter 7

Conclusion

7.1 Objectives Achieved

The goal of this thesis was to provide a procedure in which a user can use the Pilchard platform to verify IP cores. This goal was achieved by the successful Pilchard implementations of the “iftest” core and the FFT core. The successful implementation of the “iftest” core and the FFT core also demonstrated that the Pilchard could be use to implement a single IP from a single source or multiple IPs from different sources. Implementation of these cores also shows the possible problems that a user using the Pilchard platform would encounter. As the Pilchard platform used in the thesis was located at CUHK, this thesis also shows the methods in which IP verification is done remotely. Once, the Pilchard platform is available in UT, this thesis and the tutorial included will help facilitate the development cycle of first time user to the Pilchard platform.

7.2 Future Work

This thesis only provided the basic methods in which Pilchard can be used to verify the correctness of a certain IP core. With this thesis as the foundation for future work, more advanced verification and implementation of IP cores can be done.

In terms of testing, another type of verification testing can be applied using the Pilchard platform. So far, the thesis has only used the real code testing method in both functional and in-circuit verification stages. This method of testing helps in uncovering possible errors that might occur when the IP core is implemented in a real application. Other tests such as regression testing and random testing could further be used to uncover errors that might not have been detected in real code testing. This would further increase the confidence level in the functional correctness of the implemented IP cores.

In addition to including different method of testing, the verification method in the Pilchard could be further automated. Currently, the output data obtained from the Pilchard platform has to be manually compared by the user. This process becomes increasingly difficult as the number of output data increases. Hence, the user can implement a synthesizable testbench that can be downloaded into Pilchard to do automatic comparison of the output data. The testbench would only need to send a signal indicating whether the output data of implemented IP were correct or incorrect.

As was discussed in Chapter 5, the timing delay of a certain implemented IP cores can be improved with manual floorplanning. The effect of floorplanning can be further examined to determine the extent of its effect on the functionality of the IP core.

As was stated that the Pilchard platform is going to be part of the GSC network, this thesis can be expanded to include a verification work that is based on the usage of the GSC network grid.

As for the range of IP cores that can be tested, currently this thesis only deals with IP cores that are setup up to work in a single Pilchard board. Verification of IP cores that are setup to operate in parallel on multiple PC with Pilchard boards can be explored.

Bibliography

Bibliography

[1] NIST, http://aemp.eeel.nist.gov/reuse/

[2] Virtual Socket Interface Alliance. http://www.vsi.org

[3] Michael Keating and Pierre Bricaud, *Reuse Methodology Manual For System-On-A-Chip Designs*, KAP, 1999.

[4] Kelly, M and D. Bouldin, "Verification of Portable Intellectual Property Blocks For FPGAs", *Proceedings of 2000 IEEE Southeastern Conference (SECON)*, pp. 531-534, Nashville, TN, April 9, 2000.

[5] Annapolis Microsystems, http://www.annapmicro.com

[6] P.H.W. Leong, M.P. Leong, O.Y.H. Cheung, T. Tung, C.M. Kwok, and K.H. Lee, "Pilchard – A Reconfigurable Computing Platform with Memory Slot Interface", *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Rohnert Park, CA, April, 2001.

[7] K.H. Tsoi, K.H. Lee, and P.H.W. Leong, "A Massively Parallel RC4 Encryption Engine", *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa Valley, CA, April, 2002.

[8] Jeanne Marie Lehrter, "On a Grid-Based Interface to a Special-Purpose Hardware Cluster", M.S Thesis, Department of Computer Science, University of Tennessee, May 2002.

[9] Xilinx, "Virtex™-E 1.8 V Field Programmable Gate Arrays", *Datasheet (DS022)*, July 2002. http://www.xilinx.com/partinfo/ds022.htm

[10] "Development System Reference Guide", http://toolbox.xilinx.com/xilinx4/docs/dev/dev.html

[11] Ralph Kohler and Richard C. Metzger, "Benchmark Specification Document: Scalability Stressmark", *Benchmarking Tools and Assessment Environment for Configurable Computing V1.0*, Honeywell Inc., August 1998.

[12] K.H. Tsoi, *Pilchard User Reference (V0.1)*, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, NT Hong Kong, January 2002.

Appendix

A. Pilchard Design Files – VHDL and C Codes

```
--------------------------------------------------------------------------
--
-- Filename : pilchard.vhd
-- Description : VHDL code of the host to Pilchard interface.
--
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity pilchard is
port (
      PADS_exchecker_reset: in std_logic;
      PADS_dimm_ck: in std_logic;
      PADS_dimm_cke: in std_logic_vector(1 downto 0);
      PADS_dimm_ras: in std_logic;
      PADS_dimm_cas: in std_logic;
      PADS_dimm_we: in std_logic;
      PADS_dimm_s: std_logic_vector(3 downto 0);
      PADS_dimm_a: in std_logic_vector(13 downto 0);
      PADS_dimm_ba: in std_logic_vector(1 downto 0);
      PADS_dimm_rege: in std_logic;
      PADS_dimm_d: inout std_logic_vector(63 downto 0);
      PADS_dimm_cb: inout std_logic_vector(7 downto 0);
      PADS_dimm_dqmb: in std_logic_vector(7 downto 0);
      PADS_dimm_scl: in std_logic;
      PADS_dimm_sda: inout std_logic;
      PADS_dimm_sa: in std_logic_vector(2 downto 0);
      PADS_dimm_wp: in std_logic;
      PADS_io_conn: inout std_logic_vector(27 downto 0) );
end pilchard;

architecture syn of pilchard is

      component INV
      port (
            O: out std_logic;
            I: in std_logic );
      end component;

      component BUF
      port (
            I: in std_logic;
            O: out std_logic );
      end component;
```

```
component BUFG
port (
      I: in std_logic;
      O: out std_logic );
end component;

component CLKDLLHF is
port (
      CLKIN: in std_logic;
      CLKFB: in std_logic;
      RST: in std_logic;
      CLK0: out std_logic;
      CLK180: out std_logic;
      CLKDV: out std_logic;
      LOCKED: out std_logic );
end component;

component FDC is
port (
      C: in std_logic;
      CLR: in std_logic;
      D: in std_logic;
      Q: out std_logic );
end component;

component IBUF
port (
      I: in std_logic;
      O: out std_logic );
end component;

component IBUFG
port (
      I: in std_logic;
      O: out std_logic );
end component;

component IOB_FDC is
port (
      C: in std_logic;
      CLR: in std_logic;
      D: in std_logic;
      Q: out std_logic );
end component;

component IOBUF
port (
      I: in std_logic;
      O: out std_logic;
      T: in std_logic;
      IO: inout std_logic );
end component;

component OBUF
port (
```

```
      I: in std_logic;
      O: out std_logic );
end component;

component STARTUP_VIRTEX
port (
      GSR: in std_logic;
      GTS: in std_logic;
      CLK: in std_logic );
end component;

component pcore
port (
      clk: in std_logic;
      clkdiv: in std_logic;
      rst: in std_logic;
      read: in std_logic;
      write: in std_logic;
      addr: in std_logic_vector(13 downto 0);
      din: in std_logic_vector(63 downto 0);
      dout: out std_logic_vector(63 downto 0);
      dmask: in std_logic_vector(63 downto 0);
      extin: in std_logic_vector(25 downto 0);
      extout: out std_logic_vector(25 downto 0);
      extctrl: out std_logic_vector(25 downto 0) );
end component;

signal clkdllhf_clk0: std_logic;
signal clkdllhf_clkdiv: std_logic;
signal dimm_ck_bufg: std_logic;
signal dimm_s_ibuf: std_logic;
signal dimm_ras_ibuf: std_logic;
signal dimm_cas_ibuf: std_logic;
signal dimm_we_ibuf: std_logic;
signal dimm_s_ibuf_d: std_logic;
signal dimm_ras_ibuf_d: std_logic;
signal dimm_cas_ibuf_d: std_logic;
signal dimm_we_ibuf_d: std_logic;
signal dimm_d_iobuf_i: std_logic_vector(63 downto 0);
signal dimm_d_iobuf_o: std_logic_vector(63 downto 0);
signal dimm_d_iobuf_t: std_logic_vector(63 downto 0);
signal dimm_a_ibuf: std_logic_vector(14 downto 0);
signal dimm_dqmb_ibuf: std_logic_vector(7 downto 0);
signal io_conn_iobuf_i: std_logic_vector(27 downto 0);
signal io_conn_iobuf_o: std_logic_vector(27 downto 0);
signal io_conn_iobuf_t: std_logic_vector(27 downto 0);

signal s,ras,cas,we : std_logic;

signal VDD: std_logic;
signal GND: std_logic;

signal CLK: std_logic;
signal CLKDIV: std_logic;
signal RESET: std_logic;
```

```
	signal READ: std_logic;
	signal WRITE: std_logic;
	signal READ_p: std_logic;
	signal WRITE_p: std_logic;
	signal READ_n: std_logic;
	signal READ_buf: std_logic;
	signal WRITE_buf: std_logic;
	signal READ_d: std_logic;
	signal WRITE_d: std_logic;
	signal READ_d_n: std_logic;
	signal READ_d_n_buf: std_logic;

	signal pcore_addr_raw: std_logic_vector(13 downto 0);
	signal pcore_addr: std_logic_vector(13 downto 0);
	signal pcore_din: std_logic_vector(63 downto 0);
	signal pcore_dout: std_logic_vector(63 downto 0);
	signal pcore_dmask: std_logic_vector(63 downto 0);
	signal pcore_extin: std_logic_vector(25 downto 0);
	signal pcore_extout: std_logic_vector(25 downto 0);
	signal pcore_extctrl: std_logic_vector(25 downto 0);
	signal pcore_dqmb: std_logic_vector(7 downto 0);

begin

	VDD <= '1';
	GND <= '0';

	U_ck_bufg: IBUFG port map (
		I => PADS_dimm_ck,
		O => dimm_ck_bufg );

	U_reset_ibuf: IBUF port map (
		I => PADS_exchecker_reset,
		O => RESET );

	U_clkdllhf: CLKDLLHF port map (
		CLKIN => dimm_ck_bufg,
		CLKFB => CLK,
		RST => RESET,
		CLK0 => clkdllhf_clk0,
		CLK180 => open,
		CLKDV => clkdllhf_clkdiv,
		LOCKED => open );

	U_clkdllhf_clk0_bufg: BUFG port map (
		I => clkdllhf_clk0,
		O => CLK );

	U_clkdllhf_clkdiv_bufg: BUFG port map (
		I => clkdllhf_clkdiv,
		O => CLKDIV );

	U_startup: STARTUP_VIRTEX port map (
		GSR => RESET,
		GTS => GND,
```

```
      CLK => CLK );

U_dimm_s_ibuf: IBUF port map (
      I => PADS_dimm_s(0),
      O => dimm_s_ibuf );

U_dimm_ras_ibuf: IBUF port map (
      I => PADS_dimm_ras,
      O => dimm_ras_ibuf );

U_dimm_cas_ibuf: IBUF port map (
      I => PADS_dimm_cas,
      O => dimm_cas_ibuf );

U_dimm_we_ibuf: IBUF port map (
      I => PADS_dimm_we,
      O => dimm_we_ibuf );

G_dimm_d: for i in integer range 0 to 63 generate

      U_dimm_d_iobuf: IOBUF port map (
            I => dimm_d_iobuf_i(i),
            O => dimm_d_iobuf_o(i),
            T => dimm_d_iobuf_t(i),
            IO => PADS_dimm_d(i) );

      U_dimm_d_iobuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => dimm_d_iobuf_o(i),
            Q => pcore_din(i) );

      U_dimm_d_iobuf_i: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => pcore_dout(i),
            Q => dimm_d_iobuf_i(i) );

      U_dimm_d_iobuf_t: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => READ_d_n_buf,
            Q => dimm_d_iobuf_t(i) );

end generate;

G_dimm_a: for i in integer range 0 to 13 generate

      U_dimm_a_ibuf: IBUF port map (
            I => PADS_dimm_a(i),
            O => dimm_a_ibuf(i) );

      U_dimm_a_ibuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
```

```
            D => dimm_a_ibuf(i),
            Q => pcore_addr_raw(i) );

end generate;

pcore_addr(3 downto 0) <= pcore_addr_raw(3 downto 0);
addr_correct: for i in integer range 4 to 7 generate
      ADDR_INV: INV port map (
            O => pcore_addr(i),
            I => pcore_addr_raw(i) );
end generate;
pcore_addr(13 downto 8) <= pcore_addr_raw(13 downto 8);

G_dimm_dqmb: for i in integer range 0 to 7 generate

      U_dimm_dqmb_ibuf: IBUF port map (
            I => PADS_dimm_dqmb(i),
            O => dimm_dqmb_ibuf(i) );

      U_dimm_dqmb_ibuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => dimm_dqmb_ibuf(i),
            Q => pcore_dqmb(i) );

end generate;

pcore_dmask(7 downto 0) <= (others => (not pcore_dqmb(0)));
pcore_dmask(15 downto 8) <= (others => (not pcore_dqmb(1)));
pcore_dmask(23 downto 16) <= (others => (not pcore_dqmb(2)));
pcore_dmask(31 downto 24) <= (others => (not pcore_dqmb(3)));
pcore_dmask(39 downto 32) <= (others => (not pcore_dqmb(4)));
pcore_dmask(47 downto 40) <= (others => (not pcore_dqmb(5)));
pcore_dmask(55 downto 48) <= (others => (not pcore_dqmb(6)));
pcore_dmask(63 downto 56) <= (others => (not pcore_dqmb(7)));

G_io_conn: for i in integer range 2 to 27 generate

      U_io_conn_iobuf: IOBUF port map (
            I => io_conn_iobuf_i(i),
            O => io_conn_iobuf_o(i),
            T => io_conn_iobuf_t(i),
            IO => PADS_io_conn(i) );

      U_io_conn_iobuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => io_conn_iobuf_o(i),
            Q => pcore_extin(i - 2) );

      U_io_conn_iobuf_i: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => pcore_extout(i - 2),
            Q => io_conn_iobuf_i(i) );
```

```
      U_io_conn_iobuf_t: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => pcore_extctrl(i - 2),
            Q => io_conn_iobuf_t(i) );

end generate;

U_io_conn_0_iobuf: IOBUF port map (
      I => dimm_ck_bufg,
      O => open,
      T => GND,
      IO => PADS_io_conn(0) );

U_io_conn_1_iobuf: IOBUF port map (
      I => GND,
      O => open,
      T => VDD,
      IO => PADS_io_conn(1) );

READ_p <=
      (not dimm_s_ibuf) and
      (dimm_ras_ibuf) and
      (not dimm_cas_ibuf) and
      (dimm_we_ibuf);

U_read: FDC port map (
      C => CLK,
      CLR => RESET,
      D => READ_p,
      Q => READ );

U_buf_read: BUF port map (
      I => READ,
      O => READ_buf );

U_read_d: FDC port map (
      C => CLK,
      CLR => RESET,
      D => READ,
      Q => READ_d );

WRITE_p <=
      (not dimm_s_ibuf) and
      (dimm_ras_ibuf) and
      (not dimm_cas_ibuf) and
      (not dimm_we_ibuf);

U_write: FDC port map (
      C => CLK,
      CLR => RESET,
      D => WRITE_p,
      Q => WRITE );
```

```
      U_buf_write: BUF port map (
            I => WRITE,
            O => WRITE_buf );

      U_write_d: FDC port map (
            C => CLK,
            CLR => RESET,
            D => WRITE,
            Q => WRITE_d );

      READ_n <= not READ;

      U_read_d_n: FDC port map (
            C => CLK,
            CLR => RESET,
            D => READ_n,
            Q => READ_d_n );

      U_buf_read_d_n: BUF port map (
            I => READ_d_n,
            O => READ_d_n_buf );

      -- User logic should be placed inside pcore
      U_pcore: pcore port map (
            clk => CLK,
            clkdiv => CLKDIV,
            rst => RESET,
            read => READ,
            write => WRITE,
            addr => pcore_addr,
            din => pcore_din,
            dout => pcore_dout,
            dmask => pcore_dmask,
            extin => pcore_extin,
            extout => pcore_extout,
            extctrl => pcore_extctrl );

end syn;
```

```
------------------------------------------------------------------------
--
-- Filename: iflib.h
-- Written by : Brittle Tsoi Kuen Hung ( 8/13/2001 )
-- Description: C code that provide the write and read APIs.
--
------------------------------------------------------------------------

#define     DEVICE            "/dev/pilchard"
#define MTRRZ           0x100000
```

```
typedef struct
{
      int w[2];
} int64;

extern int pilchard(int, char **);
extern void write64(int64, char *);
extern void read64(int64 *, char *);
extern void write32(int, char *);
extern void read32(int *, char *);
```

```
------------------------------------------------------------------------
--
-- Filename: iflib.c
-- Description: C codes implementation of the write and read APIs in
--              iflib.h.
--
------------------------------------------------------------------------

#include "iflib.h"

#define     USEMOVQ

void
write64(int64 twrite, char *addr)
{
#ifdef      USEMOVQ
      __asm__ __volatile__(
      "
            movl %1,%%ecx\n
            movq %0,%%mm0\n
            movq %%mm0,(%%ecx)\n
      "
      :
      : "m" (twrite), "g" (addr)
      );
#else
      *((int *)(addr + 4)) = twrite.w[1];
      *((int *)addr) = twrite.w[0];
#endif
}

void
read64(int64 *data, char *addr)
{
#ifdef      USEMOVQ
      int64 tread;

      __asm__ __volatile__(
      "
            movl %1,%%ecx\n
```

```
            movq (%%ecx),%%mm1\n
            movq %%mm1,%0\n
      "
      : "=m" (tread)
      : "g" (addr)
      );
      data->w[0] = tread.w[0];
      data->w[1] = tread.w[1];
#else
      data->w[0] = *((int *)addr);
      data->w[1] = *((int *)(addr + 4));
#endif
}

void
write32(int twrite, char *addr)
{
      *((int *)addr) = twrite;
}

void
read32(int *data, char *addr)
{
      *data = *((int *)addr);
}
```

```
------------------------------------------------------------------------
--
-- Filename: download.c
-- Description: C codes to download load bit file to Pilchard.
--
------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* needed for ioperm() */
#include <sys/io.h>

#define DATA 0x378
#define STATUS DATA+1
#define CONTROL DATA+2

int main(int argc, char *argv[]) {
      FILE *bitfile;
      union {
            unsigned int i;
            unsigned short s[2];
            char c[4];
      } head_len;
      unsigned char head_key;
      char buf[2035464];
```

```
unsigned int i;
int j;
unsigned char tmp;

if (argc != 2) {
            printf("usage: %s <bitfile>\n", argv[0]);
            exit(0);
}

bitfile = fopen(argv[1], "r");
if (bitfile == NULL)
            goto Err_file;

fread(&(head_len.c[1]), 1, 1, bitfile);
fread(&(head_len.c[0]), 1, 1, bitfile);
fread(buf, head_len.s[0], 1, bitfile);
fread(&(head_len.c[1]), 1, 1, bitfile);
fread(&(head_len.c[0]), 1, 1, bitfile);

head_key = 0;
while (head_key != 0x65) {
      fread(&head_key, 1, 1, bitfile);
      fread(&(head_len.c[3]), 1, 1, bitfile);
      fread(&(head_len.c[2]), 1, 1, bitfile);
      if (head_key == 0x65) {
            fread(&(head_len.c[1]), 1, 1, bitfile);
            fread(&(head_len.c[0]), 1, 1, bitfile);
            fread(buf, head_len.i, 1, bitfile);
      }
      else {
            fread(buf, head_len.s[1], 1, bitfile);
            printf("%s\n", buf);
      }
}

fclose(bitfile);

if (iopl(3))
            goto Err_permission;

outb(0x04, CONTROL);

// sense VCC
outb(0x10, DATA);
tmp = inb(STATUS) & 0xA0;
if (tmp != 0x80)
            goto Err_cable;
outb(0x50, DATA);
tmp = inb(STATUS) & 0xA0;
if (tmp != 0x20)
            goto Err_cable;

printf("cabel detected\n");

// clear config
```

```
	outb(0x10, DATA);
	printf("configuration memory cleared\n");
	outb(0x14, DATA);
	tmp = inb(STATUS);

	printf("start loading %d bytes\n", head_len.i);
	for (i=0; i<head_len.i; i++) {
		for (j=7; j>=0; j--) {
			tmp = (buf[i]>>j) & 1;
			outb(tmp|0x14, DATA);
			outb(tmp|0x16, DATA);
		}
	}
	printf("finish loading\n");

	// Done
	tmp = inb(STATUS);
	tmp = (tmp>>4)&1;
	if (tmp)
		printf("DONE!\n");
	else
		goto Err_failed;

	return (0);

Err_file:
	printf("Err: bit file not found\n");
	return(1);

Err_permission:
	printf("Err: cannot access port\n");
	return(1);

Err_cable:
	printf("Err: cabel not found\n");
	return(1);

Err_failed:
	printf("Err: program failed\n");
	return(1);

}
```

B. IFTEST Implementation – VHDL and C Codes

```
----------------------------------------------------------------------
--
-- Filename : pcore.vhd
-- Title : Pilchard implementation of “iftest” core.
--
--
----------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
--synopsys translate_off
library UNISIM;
--synopsys translate_on

entity pcore is
port (
      clk: in std_logic;
      clkdiv: in std_logic;
      rst: in std_logic;
      read: in std_logic;
      write: in std_logic;
      addr: in std_logic_vector(13 downto 0);
      din: in std_logic_vector(63 downto 0);
      dout: out std_logic_vector(63 downto 0);
      dmask: in std_logic_vector(63 downto 0);
      extin: in std_logic_vector(25 downto 0);
      extout: out std_logic_vector(25 downto 0);
      extctrl: out std_logic_vector(25 downto 0) );
end pcore;

architecture syn of pcore is

      component RAMB4_S16
      port (
            WE: in std_logic;
            EN: in std_logic;
            RST: in std_logic;
            CLK: in std_logic;
            ADDR: in std_logic_vector(7 downto 0);
            DI: in std_logic_vector(15 downto 0);
            DO: out std_logic_vector(15 downto 0) );
      end component;

      component RAMB4_S16_S16
```

```
      port (
            WEA: in std_logic;
            ENA: in std_logic;
            RSTA: in std_logic;
            CLKA: in std_logic;
            ADDRA: in std_logic_vector(7 downto 0);
            DIA: in std_logic_vector(15 downto 0);
            DOA: out std_logic_vector(15 downto 0);
            WEB: in std_logic;
            ENB: in std_logic;
            RSTB: in std_logic;
            CLKB: in std_logic;
            ADDRB: in std_logic_vector(7 downto 0);
            DIB: in std_logic_vector(15 downto 0);
            DOB: out std_logic_vector(15 downto 0) );
      end component;

      signal VDD: std_logic;
      signal GND: std_logic;

      signal cnt_en: std_logic;
      signal addr_cnt: std_logic_vector(7 downto 0);
      signal dia: std_logic_vector(15 downto 0);

      type state_type is (INI, ST1, ST2);
      signal state: state_type;
      signal st: std_logic_vector(1 downto 0);

begin

      VDD <= '1';
      GND <= '0';

-- single port BlockRAM with 16-bit databus and 8-bit address
-- the address is connected to the lower 8 bits of the systerm memory
-- address, the data lines are connected to the lower 16 bits of the
-- system memory data bus

      U_RAM1: RAMB4_S16 port map (
            WE => write,
            EN => VDD,
            RST => rst,
            CLK => clk,
            ADDR => addr(7 downto 0),
            DI => din(15 downto 0),
            DO => dout(15 downto 0) );

-- counter enable control, if state is ST2 then counter start, else
-- counter stop

      cnt_en <= '1' when state = ST2 else '0';

-- 8-bit counter controlled by state and clock by clkdiv, half of the
-- system clock rate
      CNT: process(clkdiv, rst)
```

```
      begin
            if rst = '1' then
                  addr_cnt <= (others => '0');
            elsif clkdiv'event and clkdiv = '1' then
                  if cnt_en = '1' then
                        addr_cnt <= addr_cnt + 1;
                  end if;
            end if;
      end process CNT;

-- dual port BlockRAM with 16-bit databus and 8-bit address
-- one port is written by the counter and the other is read by host
-- the address counter initialize the contents and then the host read
-- them
      dia <= addr_cnt&"01011010";
      U_RAM2: RAMB4_S16_S16 port map (
            WEA => cnt_en,
            ENA => VDD,
            RSTA => rst,
            CLKA => clkdiv,
            ADDRA => addr_cnt,
            DIA => dia,
            DOA => open,
            WEB => GND,
            ENB => VDD,
            RSTB => rst,
            CLKB => clk,
            ADDRB => addr(7 downto 0),
            DIB => dia,
            DOB => dout(31 downto 16) );

-- Finte State Machine with 3 states
-- when system start up, stat is INI
-- after that, the state will advance to ST1
-- if a write in address 0xFF (*8 in software), the state altered
      FSM: process(clk, rst)
      begin
            if rst = '1' then
                        state <= INI;
            elsif clk'event and clk = '1' then
                  case state is
                  when INI =>
                        state <= ST1;
                  when ST1 =>
                  if write = '1' and addr(7 downto 0) = "11111111" then
                              state <= ST2;
                        else
                              state <= ST1;
                        end if;
                  when ST2 =>
                  if write = '1' and addr(7 downto 0) = "11111111" then
                              state <= ST1;
                        else
                              state <= ST2;
                        end if;
```

```
                when others =>
                      state <= INI;
                end case;
          end if;
      end process FSM;

-- output the state to databus
      st <= "00" when state = INI else
            "01" when state = ST1 else
            "10";
      dout(33 downto 32) <= st;
      dout(63 downto 34) <= din(63 downto 34);

-- output debug signals to external header
      extout(0) <= clkdiv;
      extout(1) <= rst;
      extout(2) <= read;
      extout(3) <= write;
      extout(5 downto 4) <= st;
      extout(13 downto 6) <= addr(7 downto 0);
      extout(14) <= cnt_en;
      extout(22 downto 15) <= addr_cnt;
      extout(25 downto 23) <= (others => GND);
      -- enable external header for output
      extctrl <= (others => GND);

end syn;
```

```
--------------------------------------------------------------------------
--
-- Filename : tb.vhd
-- Title : Testbench for Pilchard implementation of “iftest” core.
--
--
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity tb is
end tb;

architecture sim of tb is

component pcore
port (
      clk: in std_logic;
      clkdiv: in std_logic;
      rst: in std_logic;
      read: in std_logic;
      write: in std_logic;
      addr: in std_logic_vector(13 downto 0);
```

```
      din: in std_logic_vector(63 downto 0);
      dout: out std_logic_vector(63 downto 0);
      dmask: in std_logic_vector(63 downto 0);
      extin: in std_logic_vector(25 downto 0);
      extout: out std_logic_vector(25 downto 0);
      extctrl: out std_logic_vector(25 downto 0) );
end component;

signal clk, clkdiv, extclk, rst, read, write:   std_logic;
signal addr         : std_logic_vector(13 downto 0);
signal din, dout, dmask : std_logic_vector(63 downto 0);
signal extin, extout, extctrl : std_logic_vector(25 downto 0);
signal VDD, GND : std_logic;

begin

VDD <= '1';
GND <= '0';

CLK_UNIT: process
begin
      loop
            clk <= '0';
            wait for 50 ns;
            clk <= '1';
            wait for 50 ns;
      end loop;
end process;

CLKDIV_UNIT: process
begin
      clkdiv <= '0';
      wait for 50 ns;
      loop
            clkdiv <= '1';
            wait for 100 ns;
            clkdiv <= '0';
            wait for 100 ns;
      end loop;
end process;

EXTCLK_UNIT: process
begin
      loop
            extclk <= '0';
            wait for 71 ns;
            extclk <= '1';
            wait for 71 ns;
      end loop;
end process;

RST_UNIT: process
begin
      rst <= '0';
      wait for 25 ns;
```

```
      rst <= '1';
      wait for 175 ns;
      rst <= '0';
      wait;
end process;

U_PCORE: pcore port map (
      clk => clk,
      clkdiv => clkdiv,
      rst => rst,
      read => read,
      write => write,
      addr => addr,
      din => din,
      dout => dout,
      dmask => dmask,
      extin => extin,
      extout => extout,
      extctrl => extctrl );

read <= GND;
write <= VDD;
process (clk, rst)
begin
      if rst = '1' then
            addr <= (others => GND);
      elsif clk'event and clk = '1' then
            addr <= addr + 1;
      end if;
end process;
din(11 downto 0) <= addr(11 downto 0);

din(63 downto 12) <= (others => '0');
dmask <= (others => '0');
extin <= (others => '0');

end sim;
```

```
--------------------------------------------------------------------------
--
-- Filename : iftest.c
-- Title : C code to access implemented “iftest” core in Pilchard.
--
--
--------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
```

```
#include "iflib.h"

int main (void)
{
      int fd;
      int64 data;
      int i;
      char *memp;

      fd = open(DEVICE, O_RDWR);
      memp = (char *)mmap(NULL, MTRRZ, PROT_READ, MAP_PRIVATE, fd, 0);
      if (memp  == MAP_FAILED) {
            perror(DEVICE);
            exit(1);
      }

      read64(&data, memp);
      printf("before ini state = %d\n", data.w[1]&3);
      /* write */
      for(i=0; i<256; i++) {
            data.w[0]=(i<<8)+i;
            data.w[1]=0;
            write64(data, memp+i*8);
      }
      read64(&data, memp);
      printf("after ini state = %d\n", data.w[1]&3);

      /* read */
      for(i=0; i<256; i++) {
            read64(&data, memp+i*8);
            printf(" %08X ", data.w[0]);
            if (!((i+1)%8))
                  printf("\n");
      }

      printf("after process\n");
      for (i=0; i<9; i++) {
            read64(&data, memp+2040);
            printf("%d: state = %d\n", i, data.w[1]&3);
            write64(data, memp+2040);
      }

      munmap(memp, MTRRZ);
      close(fd);
      return 0;
}
```

C. FFT4 Implementation – VHDL and C Codes

```
--------------------------------------------------------------------------
--
-- Filename : pcore.vhd
-- Written by: Koay Teng Kuan ( 7/22/2002 )
-- Title : Pilchard implementation of FFT modules by Honeywell Inc.
-- Revised : 7/25/2002
-- Description: Pilchard implementation of the 2-radix butterfly FFT.
--          Twiddle factors are instantiated in a RAM (acting as a ROM)
--          C codes ( FFT4.c ) will be use to load the 4 data of
--          16-bits (real/imaginary) inputs into a RAM before the core
--          is started. After completing the FFT process, a signal is
--          send to the host to prompt the host to read the processed
--          inputs from the RAM
--
-- Revision: Uses clockdiv2 for cores operation. Also parameterized
--           codes.
--
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_arith.all;


library work;
use work.fftpack.all;


entity pcore is
port (
      clk: in std_logic;
      clkdiv: in std_logic;
      clk2: in std_logic;
      clkdiv2: in std_logic;
      rst: in std_logic;
      read: in std_logic;
      write: in std_logic;
      addr: in std_logic_vector(13 downto 0);
      din: in std_logic_vector(63 downto 0);
      dout: out std_logic_vector(63 downto 0);
      dmask: in std_logic_vector(63 downto 0);
      extin: in std_logic_vector(25 downto 0);
      extout: out std_logic_vector(25 downto 0);
```

```
      extctrl: out std_logic_vector(25 downto 0) );
end pcore;

architecture syn of pcore is

constant Pass : integer := 2;  --  is n, which is 2^n. For 4 data, n=2.

COMPONENT fft1_top
GENERIC (Passes : INTEGER := 2);  -- is also n.
   PORT (
     Clk             :      IN       std_logic;
     Resetn          :      IN       std_logic;
     BusGrant        :      IN       std_logic;
     Complete        :      OUT      std_logic;
     WEn             :      OUT      std_logic;
     Strobe          :      OUT      std_logic;
     Address         :      OUT      std_logic_vector(Passes downto 0);
     MemoryDataIn    :      IN       std_logic_vector(31 downto 0);
     MemoryDataOut   :      OUT      std_logic_vector(31 downto 0)
   );
END COMPONENT;

component ram_ir_4
      port (
      addra: IN std_logic_VECTOR(1 downto 0);
      addrb: IN std_logic_VECTOR(1 downto 0);
      clka: IN std_logic;
      clkb: IN std_logic;
      dina: IN std_logic_VECTOR(31 downto 0);
      dinb: IN std_logic_VECTOR(31 downto 0);
      douta: OUT std_logic_VECTOR(31 downto 0);
      doutb: OUT std_logic_VECTOR(31 downto 0);
      ena: IN std_logic;
      enb: IN std_logic;
      wea: IN std_logic;
      web: IN std_logic);
end component;

component ram_c_4
      port (
      addr: IN std_logic_VECTOR(0 downto 0);
      clk: IN std_logic;
      dout: OUT std_logic_VECTOR(31 downto 0);
      en: IN std_logic);
end component;

signal bus_grant, web, ena, enb: std_logic;
signal VDD: std_logic;
signal GND: std_logic;
signal address1, address0, address11, address12, previous:
std_logic_vector(Pass downto 0);
signal data_in_fft, data_out_fft, data_temp, data_in_ram, data_out_ram:
std_logic_vector(31 downto 0);
signal data_out_c, data_out_mux, data_out_mux1, din_in_ram,
dout_out_ram: std_logic_vector(31 downto 0);
```

```
signal switch, done: std_logic;
signal resetn: std_logic;
signal strobe, we, Wen: std_logic;
signal ones, zeros, regcount, counter: std_logic_vector(Pass-1 downto
0);
signal flag : std_logic_vector(0 downto 0);
type state_type is (INI, ST1, ST2, ST3);
signal state: state_type;

begin

VDD <= '1';
GND <= '0';
ones <= (others => VDD);
zeros <= (others => GND);

resetn <= NOT(rst);
we <= NOT(Wen);

FFT_CORE: fft1_top port map (
           Clk             => clkdiv2,
           Resetn          => resetn,
           BusGrant        => bus_grant,
           Complete        => done,
           WEn             => Wen,
           Strobe          => strobe,
           Address         => address1(Pass downto 0),
           MemoryDataIn    => data_in_fft(31 downto 0),
           MemoryDataOut   => data_out_fft(31 downto 0)
   );

RAM_I:  ram_ir_4 port map (
      addra => address11 (1 downto 0),
      addrb => addr(1 downto 0),
      clka => clkdiv2,
      clkb => clk,
      dina => data_in_ram(31 downto 0),
      dinb => din_in_ram(31 downto 0),
      douta => data_out_ram(31 downto 0),
      doutb => dout_out_ram(31 downto 0),
      ena => ena,
      enb => enb,
      wea => we,
      web => write
      );

RAM_C:  ram_c_4 port map (
      addr => address12(0 downto 0),
      dout => data_out_c(31 downto 0),
      en => VDD,
      clk => clkdiv2
      );
```

```
data_in_fft(31 downto 16) <= data_out_mux(15 downto 0) when strobe =
'0' else
                        data_in_fft(31 downto 16);
data_in_fft(15 downto 0) <= data_out_mux(31 downto 16) when strobe =
'0' else
                        data_in_fft(15 downto 0);


address11(Pass-1 downto 0) <= address1(Pass-1 downto 0) when
address1(Pass downto Pass) = "0" else
                    address11(Pass-1 downto 0);
address12(Pass-2 downto 0) <= address1(Pass-2 downto 0) when
address1(Pass downto Pass) = "1" else
                    address12(Pass-2 downto 0);


data_in_ram(15 downto 0) <= data_out_fft(31 downto 16);
data_in_ram(31 downto 16) <= data_out_fft(15 downto 0);

din_in_ram(31 downto 16) <= din(47 downto 32);
din_in_ram(15 downto 0) <= din(15 downto 0);
dout(47 downto 32) <= dout_out_ram(31 downto 16);
dout(15 downto 0) <= dout_out_ram(15 downto 0);


STARTER: process(clk, rst)
begin
      if rst = '1' then
            counter <= (others => '0');
      elsif clk'event and clk ='1' then
            if write = '1' then
            regcount <= counter;
            counter <= counter + 1;
            end if;
      end if;
end process STARTER;


enb <= '0' when state = ST2 else
            '1';
ena <= '1' when state = ST2 else
            '0';


FSM: process(clkdiv2, rst)
begin
      if rst = '1' then
            bus_grant <= '0';
            dout(31 downto 31) <= "0";
            state <= INI;
      elsif clkdiv2'event and clkdiv2 ='1' then
      flag(0 downto 0) <= address1(Pass downto Pass);
      if flag(0 downto 0) = "0" then
            if we = '1' then
            data_out_mux(31 downto 0) <= data_out_mux(31 downto 0);
```

```
            else
            data_out_mux(31 downto 0) <= data_out_ram(31 downto 0);
            end if;
            else
            data_out_mux(31 downto 0) <= data_out_c(31 downto 0);
      end if;
            case state is
            when INI =>
                  dout(31 downto 31) <= "0";
                  bus_grant <= '0';
                  state <= ST1;
            when ST1 =>
                  dout(31 downto 31) <= "0";
                  if regcount(Pass-1 downto 0) = ones then
                        bus_grant <= '1';
                        state <= ST2;
                  else
                        bus_grant <= '0';
                        state <= ST1;
                  end if;
            when ST2 =>
                  if done ='1' then
                        bus_grant <= '0';
                        dout(31 downto 31) <= "1";
                        state <= ST3;
                  else
                        bus_grant <= '1';
                        dout(31 downto 31) <= "0";
                        state <= ST2;
                  end if;
            when ST3 =>
                  bus_grant <= '0';
                  dout(31 downto 31) <= "1";
                  state <= ST3;
            end case;
      end if;
end process FSM;

end syn;
```

```
--------------------------------------------------------------------
--
-- Filename : tb.vhd
-- Written by: Koay Teng Kuan ( 7/22/2002 )
-- Title : Testbench for Pilchard implementation of FFT modules by
--         Honeywell Inc.
-- Revised : 7/25/2002
-- Description: Testbench for functional simulation of Pilchard
--              implementation of the radix-2 fixed-point butterfly
--              FFT. Twiddle factors instantiated in a RAM is simulated
--              by the including the MIF file of the RAM core.
--              “DataIn.dat” is the text file with the 4 input data of
```

```
--               16-bits (real/imaginary). “DataOut1.txt” is the text
--               file with 4 output data.
--
-- Revision: Uses clockdiv2 for cores operation. Also parameterized
--           codes.
--
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_textio.hread;

library Std_DevelopersKit;
use Std_DevelopersKit.Std_Regpak.all;
use Std_DevelopersKit.Std_IOpak.all;

use STD.TEXTIO.all;

entity tb is
end tb;

architecture sim of tb is

constant Pass : integer := 2;   -- is n, which is 2^n. For 4 data, n=2.
constant mem_size : integer := 2**(2);

component pcoretb
port (
      clk: in std_logic;
      clkdiv: in std_logic;
      clk2: in std_logic;
      clkdiv2: in std_logic;
      rst: in std_logic;
      readtb: in std_logic;
      writetb: in std_logic;
      addr: in std_logic_vector(13 downto 0);
      din: in std_logic_vector(63 downto 0);
      dout: out std_logic_vector(63 downto 0);
      dmask: in std_logic_vector(63 downto 0);
      extin: in std_logic_vector(25 downto 0);
      extout: out std_logic_vector(25 downto 0);
      extctrl: out std_logic_vector(25 downto 0) );
end component;

type mem_type is array (mem_size-1 downto 0) of integer;
signal imag_mem, real_mem : mem_type;

signal clk, clkdiv, clk2, clkdiv2, extclk  : std_logic;
signal rst, readtb, write  : std_logic;
signal addr                : std_logic_vector(13 downto 0);
signal din, dout, dmask    : std_logic_vector(63 downto 0);
signal extin, extout, extctrl : std_logic_vector(25 downto 0);
signal dout1, dout0, d_out    : std_logic_vector(31 downto 0);
```

```
signal d_outI, d_outR         : std_logic_vector(15 downto 0);
signal Dflag, DflagOut, VDD, GND, print : std_logic;
type state_type is (INI, ST1, ST2, ST3, Done1);
signal state: state_type;
signal ONES, j: std_logic_vector(Pass-1 downto 0);
signal k: integer;

begin

VDD <= '1';
GND <= '0';
ONES <= (others => VDD);

CLK_UNIT: process
begin
      loop
            clk <= '0';
            wait for 50 ns;
            clk <= '1';
            wait for 50 ns;
      end loop;
end process;

CLKDIV_UNIT: process
begin
      clkdiv <= '0';
      clk2 <= '0';
      wait for 50 ns;
      loop
            clkdiv <= '1';
            clk2 <= '1';
            wait for 100 ns;
            clkdiv <= '0';
            clk2 <= '0';
            wait for 100 ns;
      end loop;
end process;

CLKDIV2_UNIT: process
begin
      clkdiv2 <= '0';
      wait for 100 ns;
      loop
            clkdiv2 <= '1';
            wait for 200 ns;
            clkdiv2 <= '0';
            wait for 200 ns;
      end loop;
end process;

EXTCLK_UNIT: process
begin
      loop
            extclk <= '0';
            wait for 71 ns;
```

```
            extclk <= '1';
            wait for 71 ns;
      end loop;
end process;

RST_UNIT: process
begin
      rst <= '0';
      wait for 25 ns;
      rst <= '1';
      wait for 175 ns;
      rst <= '0';
      wait;
end process;

U_PCORE: pcoretb port map (
      clk => clk,
      clkdiv => clkdiv,
      clk2 => clk2,
      clkdiv2 => clkdiv2,
      rst => rst,
      readtb => readtb,
      writetb => write,
      addr => addr,
      din => din,
      dout => dout,
      dmask => dmask,
      extin => extin,
      extout => extout,
      extctrl => extctrl );

dout1 <= dout(63 downto 32);
dout0 <= dout(31 downto 0);

process (clk, rst)

VARIABLE dptr, optr : line;
VARIABLE datin : integer;
VARIABLE i : natural;

FILE infile : text IS IN "DataIn.dat";
FILE outfile : TEXT IS OUT "DataOut1.txt";

begin
      if rst = '1' then
            state <= INI;
      i := 0;
             addr(13 downto 0) <= (others => GND);
      while not endfile(infile) loop
       readline(infile, dptr);
          read (dptr, datin);
          real_mem(i) <= datin;
          read (dptr, datin);
          imag_mem(i) <= datin;
            i := i + 1;
```

```
            END LOOP;
                  k <= 0;
                  j <= (others => GND);
elsif clk'event and clk ='1' then
            case state is

            when INI =>
            write <= '1';
            addr(13 downto Pass) <= (others => GND);
            addr(Pass-1 downto 0) <= j(Pass-1 downto 0);
            din(63 downto 48) <= (others => GND);
            din(31 downto 16) <= (others => GND);
            din(47 downto 32) <= To_StdLogicVector(real_mem(k), 16);
            din(15 downto 0) <= To_StdLogicVector(imag_mem(k), 16);
                        k <= k + 1;
                        j <= j + 1;
            state <= ST1;

            when ST1 =>
            write <= '1';
            if (j = ONES ) then
            addr(13 downto Pass) <= (others => GND);
            addr(Pass-1 downto 0) <= j(Pass-1 downto 0);
            din(63 downto 48) <= (others => GND);
            din(31 downto 16) <= (others => GND);
            din(47 downto 32) <= To_StdLogicVector(real_mem(k), 16);
            din(15 downto 0) <= To_StdLogicVector(imag_mem(k), 16);
            DflagOut <= '0';
            state <= ST2;
                  else
            addr(13 downto Pass) <= (others => GND);
            addr(Pass-1 downto 0) <= j(Pass-1 downto 0);
            din(63 downto 48) <= (others => GND);
            din(31 downto 16) <= (others => GND);
            din(47 downto 32) <= To_StdLogicVector(real_mem(k), 16);
            din(15 downto 0) <= To_StdLogicVector(imag_mem(k), 16);
                        k <= k + 1;
                        j <= j + 1;
            end if;

            when ST2 =>
            write <= '0';
            j <= (others => GND);
            i := 0;
            if DflagOut = '0' then
            if dout(31 downto 31) = "1" then
               addr(13 downto Pass) <= (others => GND);
               addr(Pass-1 downto 0) <= j(Pass-1 downto 0);
                        j <= j + 1;
                        Dflag <= '0';
                        DflagOut <= '1';
                  state <= ST2;
            else
                  DflagOut <= '0';
                  state <= ST2;
```

```
            end if;
            else
                  addr(13 downto Pass) <= (others => GND);
                  addr(Pass-1 downto 0) <= j(Pass-1 downto 0);
                  j <= j + 1;
                  state <= ST3;
            end if;

            when ST3 =>
            write <= '0';
            if Dflag = '0' then
            if i = (mem_size-1) then
                  addr(13 downto Pass) <= (others => GND);
                  addr(Pass-1 downto 0) <= j(Pass-1 downto 0);
                  print <= '1';
                  Dflag <= '1';
            else
                  addr(13 downto Pass) <= (others => GND);
                  addr(Pass-1 downto 0) <= j(Pass-1 downto 0);
                  Dflag <= '0';
            end if;
            real_mem(i) <= To_Integer(dout(47 downto 32), TwosComp);
            imag_mem(i) <= To_Integer(dout(15 downto 0), TwosComp);
                  j <= j + 1;
            else
                  state <= Done1;
            end if;
                  i := i + 1;

      when Done1 =>
      if print = '1' then
      print <= '0';
      FOR y in 0 to (mem_size-1) LOOP
      fprint(outfile, optr, "%s ", To_String(real_mem(y), "%10d"));
      fprint(outfile, optr, "%s \n", To_String(imag_mem(y), "%10d"));
      END LOOP;
      else
      state <= Done1;
      end if;
      end case;
      end if;
END process;

dmask <= (others => '0');
extin <= (others => '0');

end sim;
```

```
--------------------------------------------------------------------
--
-- Filename : tb.vhd
-- Written by: Koay Teng Kuan ( 7/22/2002 )
```

```
-- Title : Testbench wrapper for “pcore.vhd”
-- Revised : 7/25/2002
-- Revision: Added ports “clk2” and “clkdiv2”.
--
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_arith.all;

entity pcoretb is
port (
      clk: in std_logic;
      clkdiv: in std_logic;
      clk2: in std_logic;
      clkdiv2: in std_logic;
      rst: in std_logic;
      readtb: in std_logic;
      writetb: in std_logic;
      addr: in std_logic_vector(13 downto 0);
      din: in std_logic_vector(63 downto 0);
      dout: out std_logic_vector(63 downto 0);
      dmask: in std_logic_vector(63 downto 0);
      extin: in std_logic_vector(25 downto 0);
      extout: out std_logic_vector(25 downto 0);
      extctrl: out std_logic_vector(25 downto 0) );
end pcoretb;

architecture syn of pcoretb is

COMPONENT pcore port (
      clk: in std_logic;
      clkdiv: in std_logic;
      clk2: in std_logic;
      clkdiv2: in std_logic;
      rst: in std_logic;
      read: in std_logic;
      write: in std_logic;
      addr: in std_logic_vector(13 downto 0);
      din: in std_logic_vector(63 downto 0);
      dout: out std_logic_vector(63 downto 0);
      dmask: in std_logic_vector(63 downto 0);
      extin: in std_logic_vector(25 downto 0);
      extout: out std_logic_vector(25 downto 0);
      extctrl: out std_logic_vector(25 downto 0) );
END COMPONENT;

BEGIN
testbench: pcore port map (
      clk => clk,
      clkdiv => clkdiv,
      clk2 => clk2,
      clkdiv2 => clkdiv2,
```

```
      rst => rst,
      read => readtb,
      write => writetb,
      addr => addr,
      din => din,
      dout => dout,
      dmask => dmask,
      extin => extin,
      extout => extout,
      extctrl => extctrl );

end syn;
```

```
------------------------------------------------------------------------
--
-- Filename : pilchard.vhd
-- Modified by: Koay Teng Kuan ( 7/24/2002 )
--
-- Modification: Additional DDL and BUFG added. CLK2 and CLKDIV2 added.
--
------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity pilchard is
port (
      PADS_exchecker_reset: in std_logic;
      PADS_dimm_ck: in std_logic;
      PADS_dimm_cke: in std_logic_vector(1 downto 0);
      PADS_dimm_ras: in std_logic;
      PADS_dimm_cas: in std_logic;
      PADS_dimm_we: in std_logic;
      PADS_dimm_s: std_logic_vector(3 downto 0);
      PADS_dimm_a: in std_logic_vector(13 downto 0);
      PADS_dimm_ba: in std_logic_vector(1 downto 0);
      PADS_dimm_rege: in std_logic;
      PADS_dimm_d: inout std_logic_vector(63 downto 0);
      PADS_dimm_cb: inout std_logic_vector(7 downto 0);
      PADS_dimm_dqmb: in std_logic_vector(7 downto 0);
      PADS_dimm_scl: in std_logic;
      PADS_dimm_sda: inout std_logic;
      PADS_dimm_sa: in std_logic_vector(2 downto 0);
      PADS_dimm_wp: in std_logic;
      PADS_io_conn: inout std_logic_vector(27 downto 0) );
end pilchard;

architecture syn of pilchard is

      component INV
      port (
            O: out std_logic;
            I: in std_logic );
      end component;
```

```
component BUF
port (
      I: in std_logic;
      O: out std_logic );
end component;

component BUFG
port (
      I: in std_logic;
      O: out std_logic );
end component;

component CLKDLLHF is
port (
      CLKIN: in std_logic;
      CLKFB: in std_logic;
      RST: in std_logic;
      CLK0: out std_logic;
      CLK180: out std_logic;
      CLKDV: out std_logic;
      LOCKED: out std_logic );
end component;

component FDC is
port (
      C: in std_logic;
      CLR: in std_logic;
      D: in std_logic;
      Q: out std_logic );
end component;

component IBUF
port (
      I: in std_logic;
      O: out std_logic );
end component;

component IBUFG
port (
      I: in std_logic;
      O: out std_logic );
end component;

component IOB_FDC is
port (
      C: in std_logic;
      CLR: in std_logic;
      D: in std_logic;
      Q: out std_logic );
end component;

component IOBUF
port (
      I: in std_logic;
      O: out std_logic;
```

```
      T: in std_logic;
      IO: inout std_logic );
end component;

component OBUF
port (
      I: in std_logic;
      O: out std_logic );
end component;

component STARTUP_VIRTEX
port (
      GSR: in std_logic;
      GTS: in std_logic;
      CLK: in std_logic );
end component;

component pcore
port (
      clk: in std_logic;
      clkdiv: in std_logic;
      clk2: in std_logic;
      clkdiv2: in std_logic;
      rst: in std_logic;
      read: in std_logic;
      write: in std_logic;
      addr: in std_logic_vector(13 downto 0);
      din: in std_logic_vector(63 downto 0);
      dout: out std_logic_vector(63 downto 0);
      dmask: in std_logic_vector(63 downto 0);
      extin: in std_logic_vector(25 downto 0);
      extout: out std_logic_vector(25 downto 0);
      extctrl: out std_logic_vector(25 downto 0) );
end component;

signal clkdllhf_clk0: std_logic;
signal clkdllhf_clkdiv: std_logic;
signal clkdllhf_clk0_2: std_logic;
signal clkdllhf_clkdiv_2: std_logic;
signal dimm_ck_bufg: std_logic;
signal dimm_s_ibuf: std_logic;
signal dimm_ras_ibuf: std_logic;
signal dimm_cas_ibuf: std_logic;
signal dimm_we_ibuf: std_logic;
signal dimm_s_ibuf_d: std_logic;
signal dimm_ras_ibuf_d: std_logic;
signal dimm_cas_ibuf_d: std_logic;
signal dimm_we_ibuf_d: std_logic;
signal dimm_d_iobuf_i: std_logic_vector(63 downto 0);
signal dimm_d_iobuf_o: std_logic_vector(63 downto 0);
signal dimm_d_iobuf_t: std_logic_vector(63 downto 0);
signal dimm_a_ibuf: std_logic_vector(14 downto 0);
signal dimm_dqmb_ibuf: std_logic_vector(7 downto 0);
signal io_conn_iobuf_i: std_logic_vector(27 downto 0);
signal io_conn_iobuf_o: std_logic_vector(27 downto 0);
```

```
	signal io_conn_iobuf_t: std_logic_vector(27 downto 0);

	signal s,ras,cas,we : std_logic;

	signal VDD: std_logic;
	signal GND: std_logic;

	signal CLK: std_logic;
	signal CLKDIV: std_logic;
	signal CLK2: std_logic;
	signal CLKDIV2: std_logic;
	signal RESET: std_logic;
	signal READ: std_logic;
	signal WRITE: std_logic;
	signal READ_p: std_logic;
	signal WRITE_p: std_logic;
	signal READ_n: std_logic;
	signal READ_buf: std_logic;
	signal WRITE_buf: std_logic;
	signal READ_d: std_logic;
	signal WRITE_d: std_logic;
	signal READ_d_n: std_logic;
	signal READ_d_n_buf: std_logic;

	signal pcore_addr_raw: std_logic_vector(13 downto 0);
	signal pcore_addr: std_logic_vector(13 downto 0);
	signal pcore_din: std_logic_vector(63 downto 0);
	signal pcore_dout: std_logic_vector(63 downto 0);
	signal pcore_dmask: std_logic_vector(63 downto 0);
	signal pcore_extin: std_logic_vector(25 downto 0);
	signal pcore_extout: std_logic_vector(25 downto 0);
	signal pcore_extctrl: std_logic_vector(25 downto 0);
	signal pcore_dqmb: std_logic_vector(7 downto 0);

begin

	VDD <= '1';
	GND <= '0';

	U_ck_bufg: IBUFG port map (
		I => PADS_dimm_ck,
		O => dimm_ck_bufg );

	U_reset_ibuf: IBUF port map (
		I => PADS_exchecker_reset,
		O => RESET );

	U_clkdllhf: CLKDLLHF port map (
		CLKIN => dimm_ck_bufg,
		CLKFB => CLK,
		RST => RESET,
		CLK0 => clkdllhf_clk0,
		CLK180 => open,
		CLKDV => clkdllhf_clkdiv,
		LOCKED => open );
```

```
U_clkdllhf_clk0_bufg: BUFG port map (
      I => clkdllhf_clk0,
      O => CLK );

U_clkdllhf_clkdiv_bufg: BUFG port map (
      I => clkdllhf_clkdiv,
      O => CLKDIV );

U_clkdllhf2: CLKDLLHF port map (
      CLKIN => CLKDIV,
      CLKFB => CLK2,
      RST => RESET,
      CLK0 => clkdllhf_clk0_2,
      CLK180 => open,
      CLKDV => clkdllhf_clkdiv_2,
      LOCKED => open );

U_clkdllhf_clk0_2_bufg: BUFG port map (
      I => clkdllhf_clk0_2,
      O => CLK2 );

U_clkdllhf_clkdiv_2_bufg: BUFG port map (
      I => clkdllhf_clkdiv_2,
      O => CLKDIV2 );

U_startup: STARTUP_VIRTEX port map (
      GSR => RESET,
      GTS => GND,
      CLK => CLK );

U_dimm_s_ibuf: IBUF port map (
      I => PADS_dimm_s(0),
      O => dimm_s_ibuf );

U_dimm_ras_ibuf: IBUF port map (
      I => PADS_dimm_ras,
      O => dimm_ras_ibuf );

U_dimm_cas_ibuf: IBUF port map (
      I => PADS_dimm_cas,
      O => dimm_cas_ibuf );

U_dimm_we_ibuf: IBUF port map (
      I => PADS_dimm_we,
      O => dimm_we_ibuf );

G_dimm_d: for i in integer range 0 to 63 generate

      U_dimm_d_iobuf: IOBUF port map (
            I => dimm_d_iobuf_i(i),
            O => dimm_d_iobuf_o(i),
            T => dimm_d_iobuf_t(i),
            IO => PADS_dimm_d(i) );
```

```
      U_dimm_d_iobuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => dimm_d_iobuf_o(i),
            Q => pcore_din(i) );

      U_dimm_d_iobuf_i: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => pcore_dout(i),
            Q => dimm_d_iobuf_i(i) );

      U_dimm_d_iobuf_t: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => READ_d_n_buf,
            Q => dimm_d_iobuf_t(i) );

end generate;

G_dimm_a: for i in integer range 0 to 13 generate

      U_dimm_a_ibuf: IBUF port map (
            I => PADS_dimm_a(i),
            O => dimm_a_ibuf(i) );

      U_dimm_a_ibuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => dimm_a_ibuf(i),
            Q => pcore_addr_raw(i) );

end generate;

pcore_addr(3 downto 0) <= pcore_addr_raw(3 downto 0);
addr_correct: for i in integer range 4 to 7 generate
      ADDR_INV: INV port map (
            O => pcore_addr(i),
            I => pcore_addr_raw(i) );
end generate;
pcore_addr(13 downto 8) <= pcore_addr_raw(13 downto 8);

G_dimm_dqmb: for i in integer range 0 to 7 generate

      U_dimm_dqmb_ibuf: IBUF port map (
            I => PADS_dimm_dqmb(i),
            O => dimm_dqmb_ibuf(i) );

      U_dimm_dqmb_ibuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => dimm_dqmb_ibuf(i),
            Q => pcore_dqmb(i) );

end generate;
```

```
pcore_dmask(7 downto 0) <= (others => (not pcore_dqmb(0)));
pcore_dmask(15 downto 8) <= (others => (not pcore_dqmb(1)));
pcore_dmask(23 downto 16) <= (others => (not pcore_dqmb(2)));
pcore_dmask(31 downto 24) <= (others => (not pcore_dqmb(3)));
pcore_dmask(39 downto 32) <= (others => (not pcore_dqmb(4)));
pcore_dmask(47 downto 40) <= (others => (not pcore_dqmb(5)));
pcore_dmask(55 downto 48) <= (others => (not pcore_dqmb(6)));
pcore_dmask(63 downto 56) <= (others => (not pcore_dqmb(7)));

G_io_conn: for i in integer range 2 to 27 generate

      U_io_conn_iobuf: IOBUF port map (
            I => io_conn_iobuf_i(i),
            O => io_conn_iobuf_o(i),
            T => io_conn_iobuf_t(i),
            IO => PADS_io_conn(i) );

      U_io_conn_iobuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => io_conn_iobuf_o(i),
            Q => pcore_extin(i - 2) );

      U_io_conn_iobuf_i: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => pcore_extout(i - 2),
            Q => io_conn_iobuf_i(i) );

      U_io_conn_iobuf_t: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => pcore_extctrl(i - 2),
            Q => io_conn_iobuf_t(i) );

end generate;

U_io_conn_0_iobuf: IOBUF port map (
      I => dimm_ck_bufg,
      O => open,
      T => GND,
      IO => PADS_io_conn(0) );

U_io_conn_1_iobuf: IOBUF port map (
      I => GND,
      O => open,
      T => VDD,
      IO => PADS_io_conn(1) );

READ_p <=
      (not dimm_s_ibuf) and
      (dimm_ras_ibuf) and
      (not dimm_cas_ibuf) and
      (dimm_we_ibuf);
```

```
U_read: FDC port map (
      C => CLK,
      CLR => RESET,
      D => READ_p,
      Q => READ );

U_buf_read: BUF port map (
      I => READ,
      O => READ_buf );

U_read_d: FDC port map (
      C => CLK,
      CLR => RESET,
      D => READ,
      Q => READ_d );

WRITE_p <=
      (not dimm_s_ibuf) and
      (dimm_ras_ibuf) and
      (not dimm_cas_ibuf) and
      (not dimm_we_ibuf);

U_write: FDC port map (
      C => CLK,
      CLR => RESET,
      D => WRITE_p,
      Q => WRITE );

U_buf_write: BUF port map (
      I => WRITE,
      O => WRITE_buf );

U_write_d: FDC port map (
      C => CLK,
      CLR => RESET,
      D => WRITE,
      Q => WRITE_d );

READ_n <= not READ;

U_read_d_n: FDC port map (
      C => CLK,
      CLR => RESET,
      D => READ_n,
      Q => READ_d_n );

U_buf_read_d_n: BUF port map (
      I => READ_d_n,
      O => READ_d_n_buf );

-- User logic should be placed inside pcore
U_pcore: pcore port map (
      clk => CLK,
      clkdiv => CLKDIV,
```

```
            clk2 => CLK2,
            clkdiv2 => CLKDIV2,
            rst => RESET,
            read => READ,
            write => WRITE,
            addr => pcore_addr,
            din => pcore_din,
            dout => pcore_dout,
            dmask => pcore_dmask,
            extin => pcore_extin,
            extout => pcore_extout,
            extctrl => pcore_extctrl );

end syn;
```

```
--------------------------------------------------------------------------
--
-- Filename : fft4.c
-- Written by: Koay Teng Kuan ( 7/18/2002 )
-- Title : C codes to access the FFT4 core implemented in Pilchard.
--
-- Description : “fft4.c” is use to test the radix-2 FFT with 4 data of
--               16-bit complex values in Pilchard. Assume data in file
--               is in integer. “datain4.dat” is input data file.
--               “dataout4.txt” is output data file.
--
--------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/time.h>

#include "iflib.h"

FILE *dataInFile;

FILE *dataOutFile;


int main (void)
{
      int fd;
      int64 data[4];
      int64   result, compare;
      int i,j, real, imag;
      char *memp;
      long hw_ttime;
      struct timeval t1,t2;
```

```
      fd = open(DEVICE, O_RDWR);
      memp = (char *)mmap(NULL, MTRRZ, PROT_READ, MAP_PRIVATE, fd, 0);
      if (memp  == MAP_FAILED) {
            perror(DEVICE);
            exit(1);
      }

      dataInFile = fopen("datain4.dat","r");
      dataOutFile = fopen("dataout4.txt","w");

    for (i=0; i<4; i++)
    {
    fscanf(dataInFile,"%d %d\n", &real, &imag);
    data[i].w[1] = real;
    data[i].w[0] = imag;
    printf("i = %d real = %d imag = %d\n",i,data[i].w[1],data[i].w[0]);
    }



      for (i=0;i<4;i++)
      {
            write64(data[i], memp+i*8);
      }

      j=1;

      gettimeofday(&t1,NULL);

      printf("j = %d, Start Computation Stage. \n", j);

      while (j==1)
      {
            read64(&result, memp);
            compare.w[0] = (result.w[0] & 0x80000000);
            if (compare.w[0] == 0x80000000)
            { j=0; }
      }

      gettimeofday(&t2,NULL);
      hw_ttime=(t2.tv_sec-t1.tv_sec)*1000000+(t2.tv_usec-t1.tv_usec);

for (i=0;i<4;i++)
{
      read64(&result, memp+i*8);
      fprintf(dataOutFile,"%08x %08x\n",result.w[1], result.w[0]);
      printf("%08x %08x\n",result.w[1], result.w[0]);
}

printf("Hardware : Pilchard perform FFT4 computation in %d usec\n",
hw_ttime);

printf("FFT4 completed. \n");

fclose(dataInFile);
```

```
fclose(dataOutFile);

munmap(memp, MTRRZ);
close(fd);
return 0;
}
```

D. Pilchard Tutorial

This tutorial is to familiarize the user with the way the design files are organized, and the download process. From here onwards, the lines in bold are command lines to be entered by users. The FFT4 core discuss in the thesis is use in this tutorial.

Part I: Setup folders and files

Step1: The following commands have to be executed in UNIX

mkdir pilchard

cd pilchard

Step 2: Copy the compress FFT4 example files to the fft folder

cp /home/tkoay/pilchard/fft.tar.gz .

Step 3: Uncompress the FFT4 example files.

gunzip –c fft.tar.gz | tar xvf-

After uncompressing, the following folders and file will be located in the pilchard directory:

$pilchard/fft/src

$pilchard/fft/ucf

$pilchard/fft/vhdl

$pilchard/fft/edif

$pilchard/fft/pilchard.fst

$pilchard/fft/V2make

PART II: Simulate the FFT4 core with the "dut.vhd"

Step 1: Simulate the fft4 IP core using its "dut.vhd" file to obtain the expected output data that should be obtained given the current input data.

cd pilchard/fft/vhdl

Step 2: Ensure that input data file and twiddle factor files are presented.

$pilchard/fft/vhd/DataIn1.dat

$pilchard/fft/vhd/Coeff.dat.dat

Step 3: Run the script to start the simulation in the /pilchard/vhdl directory:

mentor_tools

softsim4

Step 4: After ModelSim complete loading, type the following command in the ModelSim command prompt:

run 5000

Step 5: Exit ModelSim and locate the following output file generated in the pilchard/vhdl folder:

$pilchard/fft/vhdl/DataOut1.dat

Step 6: This is the expected output file that will be use when verifying the other output file obtained through Pilchard implementation.

PART III: Simulate the "pcore.vhd" with FFT4 core

The current "pcore.vhd" in the pilchard/fft directory have already been retimed to operate with the FFT4 core. User will not required to modify the "pcore.vhd" that will be used in this example. The required RAM modules have also been created and integrated into the "pcore.vhd".

Step 1: User must be in the pilchard/fft directory while performing the simulation.

cd pilchard/fft/vhdl

Step 2: Ensure that input data file and twiddle factor files are presented.

$pilchard/fft/vhd/DataIn.dat

$pilchard/fft/vhd/ram_c_4.mif

Step 3: Run the script to start the functional simulation of the "pcore" module with the FFT4 core.

mentor_tools

sim4

Step 4: After ModelSim complete loading, type the following command in the ModelSim command prompt:

run 8000

Step 5: Exit ModelSim and locate the following output file generated in the pilchard/vhdl folder:

$pilchard/fft/vhdl/DataOut1.txt

Step 6: Open and view this output file. Comparison of this output file with the output file from the “dut.vhd” simulation should be the same. This shows that the functional behavior of the FFT4 core in “pcore.vhd” is correct.

Part IV: Synthesize and PAR the “pcore”

Step 1: User must have complete PART III of this tutorial before proceeding with this part. User must then be in the pilchard/fft directory.

$pilchard/fft

Step 2: Synthesize the “pcore” module using the script “pilchard.fst”. In this tutorial, the synthesis script have already been modify to include all the relevant VHDL and EDIF files needed.

synopsys_tools

fc2_shell –f pilchard.fst

Step 3: After synthesis is complete, the following file should be presented in the following directory:

$pilchard/fft/pilchard.edf

Step 4: Run the place and route script.

xilinx_tools

V2make

Step 5: After the PAR script completed, the following file should be presented in following folder:

$pilchard/fft/pilchard.bit

PART V: Transfer files to Pilchard

Step 1: Access the Pilchard machine in CUHK network:

ssh gw.cse.cuhk.edu.hk (Enter login name and password)

ssh sparc77 (Enter login name and password)

Step 2: Use FTP to transfer files to a temporary folder in sparc77:

mkdir pilchard_temp

cd pilchard_temp

ftp vlsi1.engr.utk.edu (Enter login name and password)

Step 3: At the FTP command prompt, type the following command:

cd pilchard/fft

get pilchard.bit

cd pilchard/fft/src

get (filename) *Note: Get all the files in the pilchard/fft/src folder*

bye

Step 5: From sparc77, access Pilchard board using the following commands

ssh utk@pc90017 (Enter login name and password)

ssh utk1 (Enter login name and password)

Step 6: Transfer files in the temporary folder in sparc77 to a directory in utk1

mkdir pil_test

cd pil_test

ftp sparc77 (Enter login name and password)

cd pilchard_temp

ls

get (filename) *Note: Get all the files in the pilchard_temp folder*

bye

Part VI: In-Circuit Verifcation on Pilchard.

Step 1: Compile the C codes in utk1

make all

make iflib.o

make download

gcc –o fft4 fft4.c iflib.o

Step 2: Download bit file to Pilchard board

download pilchard.bit

Note: A "Done!" message will be displayed once the downloading is complete.

Step 3: Run executable "fft4" to access the downloaded design in Pilchard:

./fft4

Note: The current FFT4 implementation is one-time use. To restart FFT4 core, download the bit file again before running the "fft4" executable.

Step 4: After completion of the "fft4" executable, an output file is generated in the following directory:

$utk1/pil_test/dataout4.txt

Step 5: Open and compare the content of this file with the output file generated in PART III. Important to note that the output file is in hexadecimal and only the last 4 hex value in each array are the valid outputs. Verify that the values of these outputs are the same as the signed decimal value from the output file from PART III.

Tutorial for FFT4 Complete

VITA

Koay Teng Kuan, also known as Jason Koay, was born in Kuala Lumpur, Malaysia on 9th June 1978. He enrolled in the American Twinning Program at Metropolitan College, Subang in 1996. Upon completion of the first part of the program, he transferred to the University of Tennessee in Knoxville where he graduated with a Bachelor of Science degree in Electrical Engineering in May 2000. In the August 2000, he entered the Graduate School of the University of Tennessee, Knoxville. As a graduate research assistant in the Electrical Engineering department, he will be completing the requirements for his Master of Science degree in Electrical Engineering in December 2002.