Design On Pilchard Platform

Technical Report

Version 0.1

Venkatesh Bhaskaran

vbhaskar@utk.edu

February 5, 2003

Table Of Contents

1. A Brief Introduction to Pilchard's High Performance Re-configurable Platform

Pilchard is a HPRC platform developed in the CSE department at the Chinese University of Hong Kong. Its efficient interface and low cost make it suitable for various applications including cryptosystems, image processing, speech processing, clustering and rapid prototyping. The unique features about pilchard are that it uses the DIMM slot as an interface as opposed to PCI thereby leveraging bandwidth.

The following files are needed to get a design up and running on the pilchard.

Host Side Files

pilchard.c, iflib.c, iflib.h, iftest.c [Its design specific], download.c, Makefile

FPGA Design Files

A. VHDL Files

1. pcore.vhd, pilchard.vhd

B. UCF Files

1. pilchard.ucf

C. EDIF Files

iob_fdc.edif [This is most important edif file. You don't have to use other file listed here but it doesn't hurt to use it, just in case. Details on 'how-to' are explained later in this technical report.

iob_fdc_1.edif, iob_fdce.edif, iob_fdce_1.edif, iob_fdp.edif, iob_fdp_1.edif, iob_fdpe.edif, iob_fdr.edif, iob_fdr_1.edif, iob_fdre.edif, iob_fdre_1.edif, iob_fds.edif, iob_fds_1.edif, iob_fdse.edif, iob_fdse_1.edif, one_shot.edif

Xilinx utility programs such as Coregen need to have XilinxCoreLib library set up to be able to use to the primitive components. Details on 'how to' and such is explained later in this technical report. In short, the following libraries need to be set up before running developer's design.

1. XilinxCoreLib
2. Designware Components
3. Simprim models
4. Unisim models
5. Work library components

Setup scripts for Pilchard designs

These are following scripts used in a design process. An attempt is made to walk a newbie developer through each of these scripts for a simple design that implements full adder on the pilchard.

1.sc_coregen_ram----------Used in Simulations
2.vsimscrpt----------Used in Simulations
3.pilchard.fst----------Used in Synthesis
4.V2make----------Used in Place and Route

Simulations

So far listings of design files that are needed in a generic design have been discussed. The next step is to walk a designer through the whole process from specification to generating bit file and burning it on pilchard.

The first step, would be to develop a solid simulation model of a design. Users might require to do functional verification as well as timing simulation to make sure the design works ok on the hardware. For timing verification, a post layout simulation is done with a standard delay format [.sdf] file.

The design files under consideration are

1.fulladd.vhd

2.adder.vhd

3.ram_pix.vhd

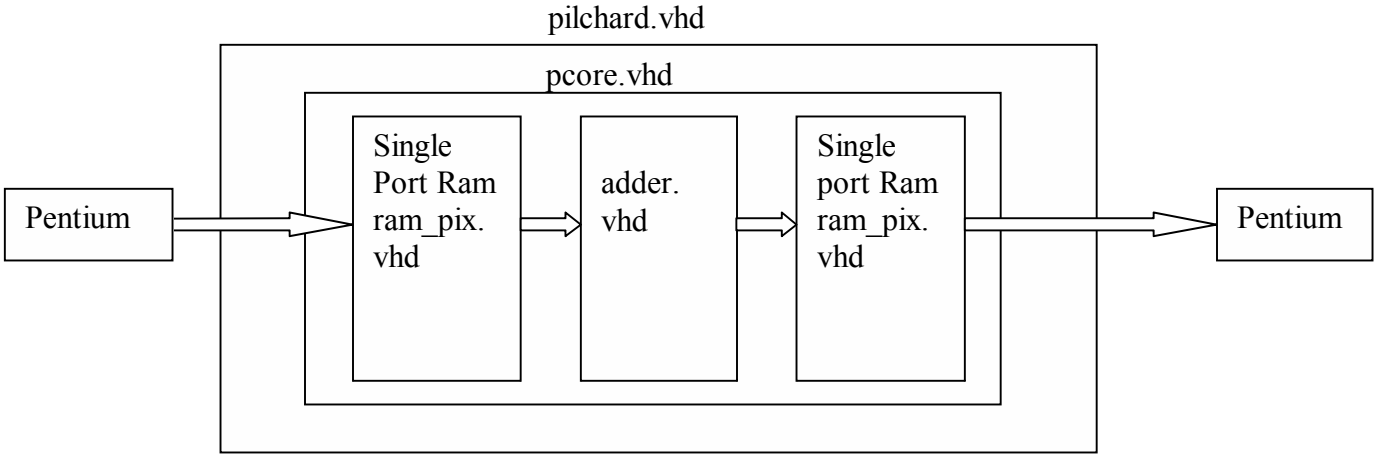
4.pcore_add.vhd

5.pcore_add_tb.vhd

6.pilchard.vhd

The entity pcore_add is a standard template that wraps developer's design. The pilchard.vhd can be thought of as a pad frame for the virtex part on the pilchard. The pilchard.vhd instanstiates various tri-state buffers and IOB's for improving the drive strengths of the signals from the outside world.

The block diagram of the design shown below gives a clear idea as to how the design works.



The ram_pix.vhd is a single port RAM module that has been generated by the 'coregen' utility. To simulate this design use the script sc_coregen_ram shown below.

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

source ~cad/.cshrc
mentor_tools

vlib XilinxCoreLib
vlib work
vmap XilinxCoreLib XilinxCoreLib

vcom -work XilinxCoreLib
/sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/mem_init_file_pack_v4_0.vhd
vcom -work XilinxCoreLib
/sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/blkmemdp_pkg_v4_0.vhd
vcom -work XilinxCoreLib
/sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/blkmemsp_pkg_v4_0.vhd
vcom -work XilinxCoreLib
/sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/blkmemdp_v4_0_comp.vhd
vcom -work XilinxCoreLib
/sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/blkmemdp_v4_0.vhd
vcom -work XilinxCoreLib
/sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/blkmemsp_v4_0_comp.vhd
vcom -work XilinxCoreLib
/sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/blkmemsp_v4_0.vhd
vcom -work XilinxCoreLib
/sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/prims_constants_v4_0.vhd
vcom -work XilinxCoreLib
/sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/prims_utils_v4_0.vhd
vcom -work XilinxCoreLib
/sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/c_reg_fd_v4_0.vhd
vcom -work XilinxCoreLib
/sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/c_reg_fd_v4_0_comp.vhd
vcom -work XilinxCoreLib
/sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/c_mux_bus_v4_0_comp.vhd
vcom -work XilinxCoreLib
/sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/c_mux_bus_v4_0.vhd
vcom -work XilinxCoreLib /sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/dividervht.vhd
vcom -work XilinxCoreLib
/sw/Xilinx5.1i/vhdl/src/XilinxCoreLib/dividervht_comp.vhd
```

Essentially what it does is compiles single port and dual port ram primitives into XilinxCoreLib library that has just been created.

Now that the primitives for compiling ram_pix.vhd are compiled, users can go ahead and compile the rest of the design.

The next step would be to run the script vsimscrpt shown below.

```
#!
source ~cad/.cshrc
mentor_tools
vlib work
vmap dware /home/chandra/ModelSim/synopsys/dware
vmap dw01  /home/chandra/ModelSim/synopsys/dw01
vmap dw02  /home/chandra/ModelSim/synopsys/dw02

vcom -work work fulladd.vhd
vcom -work work adder.vhd
vcom -work work pcore_add.vhd
vcom -work work pilchard.vhd
vcom -work work pcore_add_tb.vhd
vsim  work.pcore_add_tb &
```

The first 3 lines are not necessary but it doesn't hurt to say it again. If the design uses DesignWare IP blocks, map respective libraries to the path shown above. This time, the blocks need not be compiled, since it is mapped to Dr. Chandra's directory that has already been compiled. Again, it doesn't hurt to map designware blocks even if the design doesn't require it.

The last line invokes the simulator and designers can check prelayout simulations over here. This is just a functional check and doesn't reveal any information about timing closures and constraint. Once placed and routed, a standard delay format file is created that has timing information and the last line could be replaced to simulate an .sdf instead of a .vhd file. That would be a post layout simulation. Any glitches or setup/hold time violations could be detected here. Remember to generate an .sdf, there are 2 more steps involved. 1) Synthesis 2) Place and Route.

Let me now walk you through Synthesis step.

Synthesis

For synthesizing the design into gate level netlist, look at the following pilchard.fst script. It is pretty much self-explanatory. Run the script by invoking synopsys's Fpga Compiler

fc2_shell –f pilchard.fst

```
#
# 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_pro
set top pilchard
set target VIRTEXE
set device V1000EHQ240
set speed -6
set chip pilchard
set export_dir 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 = 50
```

```
#
# 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 fulladd.vhd
add_file -library WORK -format VHDL adder.vhd
add_file -library WORK -format EDIF ../ram_pix.edn
add_file -library WORK -format VHDL pcore_add.vhd
add_file -library WORK -format 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 -progress -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
#
```

```
exec rm -rf $export_dir
exec mkdir -p $export_dir
export_chip -progress -dir $export_dir -no_timing_constraint

#
# Save and close the project
#
close_project

quit
```

There are couple of things you might want to notice.

1. We add an edif file as opposed to vhd file for analyzing the ram_pix block. This is because of the fact that the coregen utility has already synthesized the single port ram block into gate level netlist with an extension '. edn'. There is no necessity to re-analyze and re-elaborate it. When you run this it might give you a series of warning messages that the ram_pix block is not linked to the design but just ignore it. The reason is the synthesis tool sees it as a black box and hence cannot resolve it but during the next step of place and route, it will find the necessary information it needs.

   The same kind of messages would appear for 'edif' IOB netlist primitives that have already been provided. So just ignore that too.

2. Since pilchard.vhd adds pads to the pcore_add wrapper, we have to ask the synthesis tool specifically not to add pads. This is done during the create_chip process where an additional option –module is added. It means to say that the entire core be synthesized in modularity and that no pads be added.

Once the synthesis process is done, a pilchard.edf file would have been created in the export_dir directory.

With this I will now walk you through the place and route step.

Place and Route/Generation Of Bitstream

To run the xilinx place and route tools type V2make script. The script is shown below.

```
#!/bin/csh -f
ngdbuild -sd ./export_dir -sd . -sd ../ramtest/edif -uc
../ramtest/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 &
```

NGDBuild 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 is used to map the logical design to a Xilinx FPGA.

TRACE is used to perform static timing analysis of the design based on the specified input timing constraints.

BITGEN is used to produce a bitstream for configuring the targeted Xilinx device.

FPGA_editor is used to view the fully routed design on the FPGA device.

Running the Pilchard

We can now download the bitfile to the pilchard to run the program on the board. To be able to do this we need to set up host side interface programs. These programs should be compiled on the pilchard machines. Hence, it is suggested that you simulate/synthesize/create bit file on the either faster or vlsi1 through vlsi6 machines, then do an ssh/rsh to pilchard1 to compile and run the following

make all

make iflib.o

make download

gcc –o iftest iftest.c iflib.o

The above 4 lines compile all the 'c' files and generates executables.

download pilchard.bit downloads the bitstream via the parallel port.

A "Done!" message indicates that it has been successfully downloaded.

./iftest runs the executable

We can take a quick look at the makefile and 'c' file and try to understand.

iftest.c

```
#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"
FILE *FileIn = NULL;
FILE *FileOut;

int main (void)
{
      int fd;
      int64 data;
```

```
int i;
int iChar = 0;
long dat[100];
int iSrcNode = -1;
char *memp;
char s[256] = "";
char *sString1 = NULL;

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

  if((FileIn = fopen("addtest_in_mod.txt","r"))== NULL)
  {
          fprintf(stderr, "Error Opening file");
          exit;
  }

  while(!feof(FileIn))
  {
          // Get each line into a buffer sLine
          fgets(s, 256, FileIn);

          // Check for commented lines, line feed and carriage return
 for (iChar = 0 ; iChar < 255 ; iChar++)
 if( (s[iChar] == '\n')||(s[iChar]=='\r')||(s[iChar] == '#'))
                          s[iChar] = '\0';

          if(!s[0]) continue;

          //Display the lines being read
          iSrcNode++;

          // Parsing of line starts here, first token
          sString1 = strtok(s," \t\n\r");
          if(!sString1)
                  continue;

          dat[iSrcNode] =  (long)atol(sString1);
      sString1 = NULL;
          strcpy(s,"");
  } // end of while loop
  fclose(FileIn);
for(i=0; i<=iSrcNode; i++) {
  printf("datax: %d\n",dat[i]); }

read64(&data, memp);
printf("before ini state = %d\n", data.w[1]&3);
/* write */
for(i=0; i<=iSrcNode; i++) {
      printf("Just Checking--1\n");
          //fscanf(FileIn,"\n", data.w[0]);
      data.w[0]= dat[i];
      printf("Just Checking--2\n");
```

```
            //data.w[0]=(i<<4)+i;
            data.w[1]=0;
            printf(" data probe: %d\n", data.w[0]);
            write64(data, memp+i*8);
            //printf(" %08X\n",memp+i*8);
      }
      read64(&data, memp);
      printf("after ini state = %d\n", data.w[1]&3);

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

The iftest.c file first maps the device file to a physical location using the mmap() function call. Once it allocates buffer space for pilchard, the program read data to be processed from a data file and writes it to the 64 bit data line on the pilchard using the write64() api. The 64 bit data is represented in a 2 32-bit integer type data structure. data.w[1] represent higher order bits while data.w[0] is for lower order bits. We can read stuff out using the read64() api but need to be a little bit careful since there is only one single bi-directional 64 bit bus for data transfer in and out. So bus arbitration should be taken care of especially when the application requires to do alternate reads and writes. For simple design like in the present case, it shouldn’t pose much problem.

Makefile

```
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
```

This makefile shouldn't be hard to figure out. It's very much self-explanatory. You don't have to tweak any other given code for most of the designs.

Summary of the commands to run a simple adder design on pilchard.

cd $current_dir

source ~cad/.cshrc

cp ~vbhaskar/pilchard_tut/vhd ./

cp ~vbhaskar/pilchard_tut/src ./

cp ~vbhaskar/pilchard_tut/edif ./

cp ~vbhaskar/pilchard_tut/pilchard.ucf .

cd vhd

mentor_tools

sc_coregen_ram

vsimscrpt

Manually add signals to the wave and run the simulation.

synopsys_new_tools

fc2_shell –f pilchard.fst

cd export_dir

check if pilchard.edf is generated

cd ..

V2make

cd ../src

make all

make iflib.o

make download

gcc –o iftest iftest.c iflib.o

download pilchard.bit

./iftest