LEON PCI Verification Study

GR-TECH-021
Issue 1.3
February 2003

Prepared for Estec contract 15102/01/NL/FM, CCN-5

Gaisler Research

Gaisler Research

jiri@gaisler.com

LEON PCI Verification Study

1 Introduction

1.1 Scope

This report contains the outcome of the LEON PCI verification study (work package 2), performed by Gaisler Reserch. The aim of the study was the following:

- Verify the functionality of the AMBA wrapper for the InSilicon PCI core
- Verify the compatibility of the InSilicon PCI interface with PCI buses in legacy PC ‘s
- Verify the conpatibility of the AMBA wrapper with the implementation of the LEON AMBA buses

1.2 Summary

The functionality of the InSilicon PCI core has been verified with focus on the LEON/AMBA interface and compatibility with desktop PCI buses. Three problems were found:

- PCI 64-bit addressing confused legacy BIOSes and made it difficult to use LEON in standard PC’s (1)
- the programing sequence of certain registers in the AHB/PCI interface could cause the operation of the PCI interface to fail (2)
- the LEON SDRAM controller could not handle a burst write from the PCI interface (3)

Problems 1 and 2 were solved by a design modification of the AMBA wrapper (Estec), while problem (3) was solved by a modification in the SDRAM controller (Gaisler Research). After the design modifications, all tested PCI transfer types were succefully verified both on a passive PCI backplane, and in a standard PC.

2 Test procedure and results

2.1 Verification approach

The verification of the PCI interface was carried out by implemeting the LEON processor with the AMBA wrapper and InSilicon PCI interface on a FPGA prototype board, connect the board to a PCI backplane, and then excersise all relevant PCI cycles. Two type of PCI buses were used; the expansion PCI bus in a legacy PC, and a passive PCI/ISA backplane. This tested the PCI interface in both satellite mode (legacy PC) and host mode (passive backplane).

2.2 FPGA test board

The prototype board hosting the Leon processor, AMBA wrapper and PCI core was an Avnet Virtex-E Development Board(figure 1), which has a PCI interface connected directly to the FPGA. The board has 32 Mbyte of SDRAM and one UART. The SDRAM was used as LEON main memory, while the UART was assigned to the DSU and used to download and run applications. The LEON processor was clocked at 24.6 MHz by an on-board oscillator, while the PCI interface was clocked by the PCI clock.

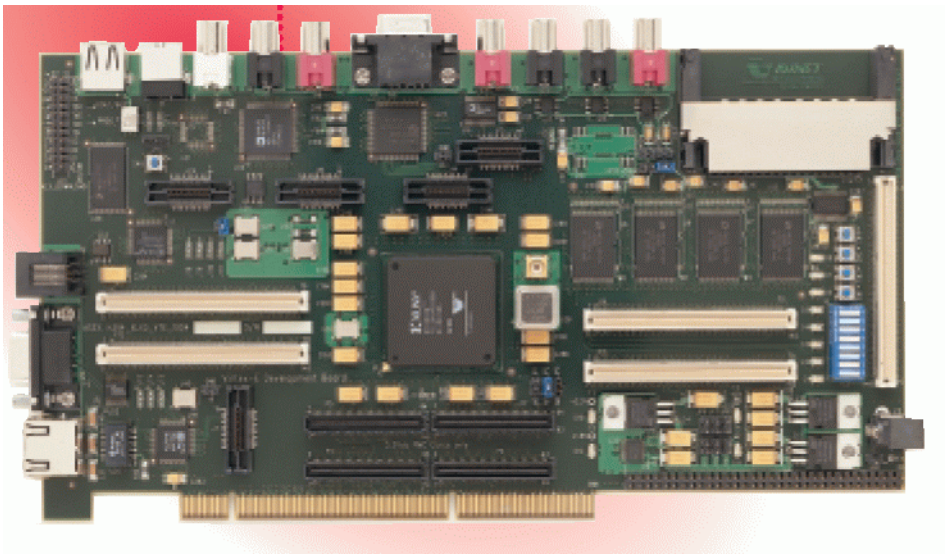

Fig 1: Avnet VirtxE Development Board

A problem encoutered during the place&route of the FPGA that the InSilicon core could not meet all timing requirements of a 33 MHz PCI bus when implemented in Virtex-1000E-6 device. In particular, the setup time of FRAME, IRDY and TRDY signals was around 12 ns instead of the required 7 ns. This meant that correct PCI operation was not guaranteed at 33 MHz, and would depend on the electrical characteristics of the PCI bus used.

2.3 Tests in legacy PC

The test board was inserted in a Captech PC (33 MHz PCI bus), with the PCI interface configured in ‘satellite’ mode. The board was recognized by the BIOS during booting, but disabled and could not be accessed from the operating system. The cause of this was found to be the use of 64-bit addressing; the InSilicon PCI core was initially configured to support 64-bit PCI addressing which most PC chipset do not support. After that the PCI core was modified to only use 32-bit addressing, the board was properly initialised and configured by the PC BIOS. A simple LINUX device

driver was developed which allowed reading and writing single (non-burst) data to both the I/O and memory BAR of the PCI interface. Reading and writing of the two areas worked without problems, and transfer rates of about 10 Mbyte/s were reached. Higher data rate would require DMA transfers but was not supported by the driver.

To test PCI target block transfers, a second FPGA PCI board also containing a LEON processor with the InSilicon PCI core was inserted into the PC. Block transfers were then performed between the two boards using the PCI DMA controller. The second board had no off-chip RAM, so the on-chip trace-buffer memory was used as buffer area for the DMA transfers. The netlist in the second FPGA board was then exchanged to a LEON configuration using the OpenCores PCI bridge and 64 Kbyte on-chip RAM. A test program was run from the on-chip RAM performing direct PCI access using LDD and STD instructions to the first board (containing the InSilicon PCI). The transfers were made while a simple LEON application was running on the first board to verify that arbitration and hand-over of the internal AHB bus worked properly. In all case, the test were successfull.

To verify that the DSU could be controlled from the PCI bus, the DSU monitor software was modified to use the PCI device driver rather than the DSU UART for communication. Several test applications were downloaded and debugged using the PCI interface, and no anomalies were found despite the non-conformance to the PCI timing (see “FPGA test board” on page 4). The PCI bus was kept lightly loaded during the tests (only one additional device) and the board was inserted as close as possible to the PCI bridge. Below is a DSU monitor log showing connection and execution of sofware:

```
jiri@mars:~/ibm/sources/dsumon$ dsumon -pci -i -u

 LEON DSU Monitor, version 1.0.7
 Copyright (C) 2001, Gaisler Research - all rights reserved.
 Comments or bug-reports to jiri@gaisler.com


 clock frequency       :  24.79 MHz
 register windows      :  8
 instruction cache     :  1 * 4 kbytes, 16 bytes/line (4 kbytes total)
 data cache            :  1 * 2 kbytes, 16 bytes/line (2 kbytes total)
 hardware breakpoints  :  4
 trace buffer          :  128 lines, mixed cpu/ahb tracing
 PCI core              :  insilicon (16e3:1e0f)
 sdram                 :  1 * 32 Mbyte @ 0x40000000
 sdram parameters      :  column bits: 9, cas delay: 2, refresh 15.5 us
 stack pointer         :  0x41fffff0
 UART 1 in DSU mode

dsu> lo stanford_leon
section: .text at 0x40000000, size 51024 bytes
section: .data at 0x4000c750, size 1904 bytes
total size: 52928 bytes (17858.5 kbit/s)
dsu> run
Starting
    Perm  Towers  Queens   Intmm      Mm  Puzzle   Quick  Bubble    Tree     FFT
      66     100      50     184    1650     567      67     100     550    1966

Nonfloating point composite is        266

Floating point composite is       1621

Program exited normally.
```

2.4 Test in passive backplane

To test the PCI interface in host mode, the FPGA board was modified to act as a system controller and inserted in a passive PCI backplane (figure 2). Two other boards were also inserted in the backplane; a 100 Mbit ethernet card from Realtek and a serial interface card. As system controller in a passive PCI backplane, configuration cycles could be tested and DMA transfers could be performed without disturbing the host operating system in a PC.

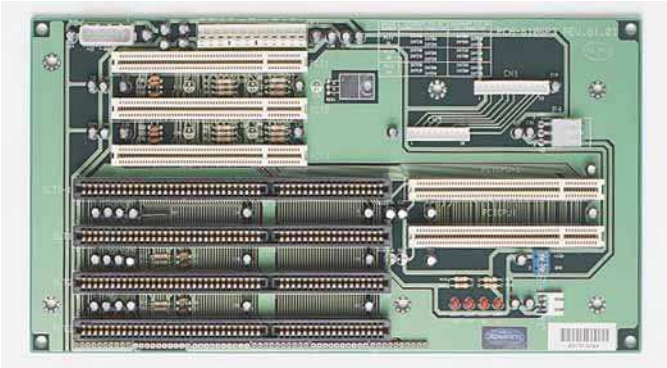

Fig 2: 3-slot passive PCI backplane

A simple PCI BIOS for LEON was developed; the BIOS scanned the PCI bus for attached devices and performed configuration cycles to map detected devices in an free part of the PCI address space. The PCI configuration cycles were made using the APB interface to the PCI core.

Several LEON test programs were developed to verify the different supported transfer modes. These were in the end put together into a single program testing the following operations for both read and write.

- APB Slave
- AHB Slave
- DMA - AHB Master

During the development of the test program, two problems were encountered:

- DMA read operation would fail if the dma_address was not written very closely after the mas_wc has been written
- DMA read operation would fail if the target address was in the SDRAM area and the word count was larger than 1

After a modification of the AMBA wrapper as well as the SDRAM controller, both problems were removed and the test program executed without errors. The log produced by the test program can be found below. The log also gives information on which tests actually where made. The passive backplane PCI clock was set to 10 MHz during the tests to avoid possible timing problems due to the non-conformant PCI timing.

Test program log

```
Performing configuration cycles...

Slot 1 -Device and Vendor ID = 0x71681409
BAR     Type    Base            Size
0       0x1     0xec000000      32 bytes
Slot 3 -Device and Vendor ID = 0x813910ec  <----Realtek
BAR     Type    Base            Size
0       0x1     0xeb000000      256 bytes   <---memory area
1       0x0     0xea000000      256 bytes   <---I/O area

**CPU MEM_BAR1**
Type    |Base Address   |Size
0x8     |0xd0000000     |16777216 bytes

**CPU MEM_BAR2**
Type    |Base Address   |Size
0x8     |0xc0000000     |16777216 bytes

**CPU IO_BAR**
Type    |Base Address   |Size
0x1     |0xb0000000     |1024 bytes

Configuration cycles done!
Testing AHB slave mode...
Writing 12 words to 0xea00008c...
Written!
Reading back the 12 words from 0xea00008c...
Offset 0: 0xa0000000 - Checked out identical!
Offset 1: 0xa0000001 - Checked out identical!
Offset 2: 0xa0000002 - Checked out identical!
Offset 3: 0xa0000003 - Checked out identical!
Offset 4: 0xa0000004 - Checked out identical!
Offset 5: 0xa0000005 - Checked out identical!
Offset 6: 0xa0000006 - Checked out identical!
Offset 7: 0xa0000007 - Checked out identical!
Offset 8: 0xa0000008 - Checked out identical!
Offset 9: 0xa0000009 - Checked out identical!
Offset 10: 0xa000000a - Checked out identical!
Offset 11: 0xa000000b - Checked out identical!
Read done!
int_st=0x00000000
Testing APB slave mode...
Writing 12 words to 0xea00008c...
Written!
Reading back the 12 words from 0xea00008c...
Offset 0: 0xb0000000 - Checked out identical!
Offset 1: 0xb0000001 - Checked out identical!
Offset 2: 0xb0000002 - Checked out identical!
Offset 3: 0xb0000003 - Checked out identical!
Offset 4: 0xb0000004 - Checked out identical!
Offset 5: 0xb0000005 - Checked out identical!
Offset 6: 0xb0000006 - Checked out identical!
Offset 7: 0xb0000007 - Checked out identical!
Offset 8: 0xb0000008 - Checked out identical!
Offset 9: 0xb0000009 - Checked out identical!
Offset 10: 0xb000000a - Checked out identical!
Offset 11: 0xb000000b - Checked out identical!
Read done!
int_st=0x00000000
Testing DMA mode...
Writing 8 words to 0x40009cf8...
Written!
Initiating transfer of 8 words from 0x40009cf8 to 0xea00008c...
Complete!
Dumping 0xea00008c with AHB direct addressing...
Offset 0: 0xc0000000 - Checked out identical!
Offset 1: 0xc0000001 - Checked out identical!
Offset 2: 0xc0000002 - Checked out identical!
Offset 3: 0xc0000003 - Checked out identical!
```

```
Offset 4: 0xc0000004 - Checked out identical!
Offset 5: 0xc0000005 - Checked out identical!
Offset 6: 0xc0000006 - Checked out identical!
Offset 7: 0xc0000007 - Checked out identical!
Read done!
int_st=0x00000080
Initiating transfer of 1 word from 0xea00008c to 0x40009cf8...
Complete!
Dumping 0x40009cf8...
Offset 0: 0xc0000004 - Checked out identical!
Read done!
int_st=0x00000080
Destroying data at 0x40009cf8...
Offset 0: 0xffffffff
Offset 1: 0xffffffff
Offset 2: 0xffffffff
Offset 3: 0xffffffff
Offset 4: 0xffffffff
Offset 5: 0xffffffff
Offset 6: 0xffffffff
Offset 7: 0xffffffff
Initiating transfer of 8 word from 0xea00008c to 0x40009cf8...
Complete!
Dumping 0x40009cf8...
Offset 0: 0xc0000000 - Checked out identical!
Offset 1: 0xc0000001 - Checked out identical!
Offset 2: 0xc0000002 - Checked out identical!
Offset 3: 0xc0000003 - Checked out identical!
Offset 4: 0xc0000004 - Checked out identical!
Offset 5: 0xc0000005 - Checked out identical!
Offset 6: 0xc0000006 - Checked out identical!
Offset 7: 0xc0000007 - Checked out identical!
Read done!
int_st=0x00000080
```

LEON PCI test program

```
#include <stdio.h>
#define GETBIT(data, bit) ((data >> bit) & 0x1)
#define COLD
//#define DMAONLY
#define DMAWC 8
int main()
{

  int slot, bar, status, defvalue, bartype, ID_data, temp_data, baseaddr, i, masa;
  int topaddr = 0xec000000;

  volatile unsigned int *device_id,
    *status_command,
    *class_revision,
    *mem_base1_address,
    *mem_base2_address,
    *io_base_address,
    *config_ben,
    *mas_address,
    *mas_ben,
    *mas_wc,
    *mas_st,
    *mas_rs,
    *tar_pa,
    *tar_sc,
    *int_en,
    *int_st,
    *int_test,
    *mas_data,
    *mas_data_last,
    *dma_address,
    *ram;
volatile int *EthernetBaseAddr;

  /*****************************************************************/
  /* write single to PCI bus through APB */
  /* the function repeats the transaction, if errors occur */
  /*****************************************************************/
  int pci_apb_wr(int slot, int reg, int data)

  {
    while (GETBIT(*mas_st,7) == 1); /* check if no request pending (bit 7) */
    *mas_address = masa = (1 << slot) | (reg << 2);
    *mas_ben = 0;
    *mas_wc = (0xb << 8) | 1; /* start memory write burst for 1 word */
    while (GETBIT(*mas_st,6) == 1); /* if queue not full */

    *mas_data = data;
    while(GETBIT(*mas_st,7) == 1 && *int_st == 0); /* ==> wait for termination on PCI bus */
    //  printf("DMAin:addr:0x%08x,data::0x%08x\n", masa, data);
    *mas_ben = 0; /* reset byte enables */
    if (*int_st != 0) {
      *int_st = -1;
      return 1;
    }
    else return 0;
  }

  /*****************************************************************/
  /* read single from PCI bus through APB */
  /*****************************************************************/
  unsigned int pci_apb_rd(int slot, int reg, int *rd_data)

  {

    while (GETBIT(*mas_st,7) == 1); /* check if no request pending (bit 7) */
    *mas_address  = (1 << slot) | (reg << 2);
    *mas_ben = 0;
    *mas_wc = (0xa << 8) | 1; /* start memory read burst for 1 word */
    while (GETBIT(*mas_st,4) == 1 && *int_st == 0); /* wait f. !mrcv_fifo_empty */
```

```
  if (*int_st != 0) {
    *int_st = -1;
    return 1;
  }
  else
    *rd_data = *mas_data;
  while(GETBIT(*mas_st,7) == 1 && *int_st == 0); /* ==> wait for termination on PCI bus */

  *mas_ben = 0; /* reset byte enables */
  if (*int_st != 0) {
    *int_st = -1;
    return 1;
  }
  else return 0;
}

/*********************************************************************/
/* read single from PCI bus through APB */
/*********************************************************************/
unsigned int pci_apbany_rd(int addr, int *rd_data)

{

  while (GETBIT(*mas_st,7) == 1); /* check if no request pending (bit 7) */
  *mas_address  = addr;
  *mas_ben = 0;
  *mas_wc = (0x6 << 8) | 1; /* start memory read burst for 1 word */
  while (GETBIT(*mas_st,4) == 1 && *int_st == 0); /* wait f. !mrcv_fifo_empty */
  if (*int_st != 0) {
    *int_st = -1;
    return 1;
  }
  else
    *rd_data = *mas_data;
  while(GETBIT(*mas_st,7) == 1 && *int_st == 0); /* ==> wait for termination on PCI bus */

  *mas_ben = 0; /* reset byte enables */
  if (*int_st != 0) {
    *int_st = -1;
    return 1;
  }
  else return 0;
}

/*********************************************************************/
/* write single to PCI bus through APB */
/* the function repeats the transaction, if errors occur */
/*********************************************************************/
int pci_apbany_wr(int addr, int data)

{
  while (GETBIT(*mas_st,7) == 1); /* check if no request pending (bit 7) */
  *mas_address = masa = addr;
  *mas_ben = 0;
  *mas_wc = (0x7 << 8) | 1; /* start memory write burst for 1 word */
  while (GETBIT(*mas_st,6) == 1); /* if queue not full */

  *mas_data = data;
  while(GETBIT(*mas_st,7) == 1 && *int_st == 0); /* ==> wait for termination on PCI bus */
  //  printf("DMAin:addr:0x%08x,data::0x%08x\n", masa, data);
  *mas_ben = 0; /* reset byte enables */
  if (*int_st != 0) {
    *int_st = -1;
    return 1;
  }
  else return 0;
}

device_id    = (int *) 0x80000100;
status_command = (int *) 0x80000104;
class_revision = (int *) 0x80000108;
mem_base1_address = (int *) 0x80000110;
```

```
  mem_base2_address = (int *) 0x80000114;
  io_base_address  = (int *) 0x80000118;
  config_ben    = (int *) 0x80000144;
  mas_address   = (int *) 0x80000148;
  mas_ben       = (int *) 0x8000014c;
  mas_wc        = (int *) 0x80000150;
  mas_st        = (int *) 0x80000154;
  mas_rs        = (int *) 0x80000158;
  tar_pa        = (int *) 0x8000015c;
  tar_sc        = (int *) 0x80000160;
  int_en        = (int *) 0x80000164;
  int_st        = (int *) 0x80000168;
  int_test      = (int *) 0x8000016c;
  mas_data      = (int *) 0x80000170;
  mas_data_last = (int *) 0x80000174;
  dma_address   = (int *) 0x80000178;
  ram           = (int *) malloc(256);
#ifdef COLD
  printf("Performing configuratio cycles...\n");
  *status_command = 0x4;
  *mas_rs = 0x40; /* APB mode and mem rd/wr */
  *int_st = -1; /* reset interrupt status register */

  for (slot=31; slot>=11; slot--) {
    //printf("Slot %d ", slot);
    while (GETBIT(*mas_st, 7) != 0);

    status = pci_apb_rd(slot, 0, &ID_data);
    if (status == 1)
      {}//printf("-No device in slot\n");
    else {
      printf("Slot %d ", slot);
      printf("-Device and Vendor ID =\t0x%08x\n", ID_data);
      printf("BAR\tType\tBase\t\tSize\n");

      for (bar=0; bar<=5; bar++) {
      /*
      status = pci_apb_rd(slot, bar + 4, &temp_data);
      if (status == 1) {
        printf("-Error reading slot\n");
        break;
      }
      printf("test:0x%08x\n", temp_data);

      */

      status = pci_apb_wr(slot, bar + 4, -1);
      if (status == 1) {
        printf("-Error writing slot\n");
        break;
      }

      status = pci_apb_rd(slot, bar + 4, &temp_data);
      if (status == 1) {
        printf("-Error reading slot\n");
        break;
      }
      //     printf("tempdata=0x%08x\n", temp_data);
      defvalue = temp_data;
      if (GETBIT(defvalue,0) == 1) { /* I/O */
        bartype = defvalue & 0x3;
        defvalue = defvalue & ~(0x3);
        if (defvalue == 0)
          continue;
      }
      else {
        bartype = defvalue & 0xf;
        defvalue = defvalue & ~(0xf);
        if (defvalue == 0)
          continue;
      }
```

```
      baseaddr = (topaddr); // - 1) - ~(defvalue);
      topaddr -= 0x1000000;
      if ( (ID_data == 0x813910ec) && (bar == 1) )
      {
        EthernetBaseAddr = (int *) baseaddr;
        status = pci_apb_wr(slot, 1, 0x02000003);
        if (status == 1) {
          printf("-Error writing slot\n");
          break;
        }
      }

      printf("%d\t0x%x\t0x%08x\t%d bytes\n", bar, bartype, baseaddr, (~defvalue)+1);
      /*
      baseaddr = (baseaddr - 1) - ~(*mas_data - defvalue);
      printf("0x%08x\t|\n", baseaddr);
      */

      status = pci_apb_wr(slot, bar + 4, baseaddr);
      if (status == 1) {
        printf("-Error writing slot\n");
        break;
      }
      /*
      status = pci_apb_rd(slot, bar + 4, &temp_data);
      if (status == 1) {
        printf("-Error reading slot\n");
        break;
      }
      printf("tempdata=0x%08x\n", temp_data);
      */
      }/*bar*/
    }
  } /*slot*/
  /*probing Host Bars*/
  topaddr = 0xd0000000;
  printf("\n**CPU MEM_BAR1**\nType\t|Base Address\t|Size\n");
  *mem_base1_address = 0;
  defvalue = *mem_base1_address;
  printf("0x%x\t|", defvalue);
  *mem_base1_address = -1;
  defvalue = *mem_base1_address & ~0xf;
  baseaddr = (topaddr);// - 1) - ~(*mem_base1_address - defvalue);
  topaddr -= 0x10000000;
  *mem_base1_address = baseaddr;
  printf("0x%x\t|%d bytes\n", baseaddr, (~defvalue)+1);

  printf("\n**CPU MEM_BAR2**\nType\t|Base Address\t|Size\n");
  *mem_base2_address = 0;
  defvalue = *mem_base2_address;
  printf("0x%x\t|", defvalue);
  *mem_base2_address = -1;
  defvalue = *mem_base2_address & ~0xf;
  baseaddr = (topaddr);// - 1) - ~(*mem_base2_address - defvalue);
  topaddr -= 0x10000000;
  *mem_base2_address = baseaddr;
  printf("0x%x\t|%d bytes\n", baseaddr, (~defvalue)+1);

  printf("\n**CPU IO_BAR**\nType\t|Base Address\t|Size\n");
  *io_base_address = 0;
  defvalue = *io_base_address;
  printf("0x%x\t|", defvalue);
  *io_base_address = -1;
  defvalue = *io_base_address & ~0x3;
  baseaddr = (topaddr);// - 1) - ~(*io_base_address - defvalue);
  topaddr -= 0x10000000;
  *io_base_address = baseaddr;
  printf("0x%x\t|%d bytes\n\n", baseaddr, (~defvalue)+1);

  printf("Configuration cycles done!\n");
#endif
```

```
  EthernetBaseAddr = (int *) 0xea00008c;

  //////////
#ifndef DMAONLY
  printf("Testing AHB slave mode...\n");

  *mas_rs = 0x41; /* AHB mode and mem rd/wr */
  *int_st = -1; /* reset interrupt status register */

  printf("Writing 12 words to 0x%08x...\n", (int) EthernetBaseAddr);
  for (i = 0; i < 12; i++)
    *((int *) EthernetBaseAddr + i) = 0xa0000000 + i;

  printf("Written!\n");

  printf("Reading back the 12 words from 0x%08x...\n", (int) EthernetBaseAddr);
  for (i = 0; i < 12; i++)
  {
    temp_data = *((int *) EthernetBaseAddr + i);
    printf("Offset %d: 0x%08x - ", i, temp_data);
    if (temp_data == 0xa0000000 + i)
      printf("Checked out identical!\n");
    else
      printf("Not correct! Expected 0x%08x\n", 0xa0000000 + i);
  }
  printf("Read done!\n");
  printf("int_st=0x%08x\n", *int_st);

  //////////

  printf("Testing APB slave mode...\n");

  *mas_rs = 0x0; /* APB mode */
  *int_st = -1; /* reset interrupt status register */

  printf("Writing 12 words to 0x%08x...\n", (int) EthernetBaseAddr);
  for (i = 0; i < 12; i++)
    pci_apbany_wr((int) (EthernetBaseAddr + i), 0xb0000000 + i);

  printf("Written!\n");

  printf("Reading back the 12 words from 0x%08x...\n", (int) EthernetBaseAddr);
  for (i = 0; i < 12; i++)
  {
    pci_apbany_rd((int) (EthernetBaseAddr + i), &temp_data);
    printf("Offset %d: 0x%08x - ", i, temp_data);
    if (temp_data == 0xb0000000 + i)
      printf("Checked out identical!\n");
    else
      printf("Not correct! Expected 0x%08x\n", 0xb0000000 + i);
  }
  printf("Read done!\n");
  printf("int_st=0x%08x\n", *int_st);
#endif
  //////////
  printf("Testing DMA mode...\n");

  *mas_rs = 0x41; /* AHB mode and mem rd/wr */
  *int_st = -1; /* reset interrupt status register */
  printf("Writing 8 words to 0x%08x...\n", (int) ram);
  for (i = 0; i < DMAWC; i++)
    *((int *) (ram + i)) = 0xc0000000 + i;

  printf("Written!\n");

  printf("Initiating transfer of 8 words from 0x%08x to 0x%08x...\n", (int) ram, (int) Ether-
netBaseAddr);

  *mas_address = (int) EthernetBaseAddr;
  *mas_wc      = (7 << 8) | DMAWC;
  *dma_address = (int) ram;
```

```
  while (GETBIT(*int_st, 7) == 0);

  printf("Complete!\n");

  printf("Dumping 0x%08x with AHB direct addressing...\n", (int) EthernetBaseAddr);
  for (i = 0; i < DMAWC; i++)
  {
    temp_data = *((int *) EthernetBaseAddr + i);
    printf("Offset %d: 0x%08x - ", i, temp_data);
    if (temp_data == 0xc0000000 + i)
      printf("Checked out identical!\n");
    else
      printf("Not correct! Expected 0x%08x\n", 0xc0000000 + i);
  }
  printf("Read done!\n");
  printf("int_st=0x%08x\n", *int_st);
  ///////////
  *int_st = -1; /* reset interrupt status register */

  printf("Initiating transfer of 1 word from 0x%08x to 0x%08x...\n", (int) EthernetBaseAddr,
(int) ram);

  *mas_address =  (int) (EthernetBaseAddr + 4);
  *mas_wc      = (6 << 8) | 1;
  *dma_address = (int) ram;

  while (GETBIT(*int_st, 7) == 0);

  printf("Complete!\n");

  printf("Dumping 0x%08x...\n", (int) ram);
  printf("Offset %d: 0x%08x - ", 0, *ram);

  if (*ram == 0xc0000000 + 4)
    printf("Checked out identical!\n");
  else
    printf("Not correct! Expected 0x%08x\n", 0xc0000000 + 4);
  printf("Read done!\n");
  printf("int_st=0x%08x\n", *int_st);

  ///////////
  *int_st = -1; /* reset interrupt status register */
  printf("Destroying data at 0x%08x...\n", (int) ram);
  for (i = 0; i < DMAWC; i++)
  {
    *((int *) (ram + i)) = -1;
    printf("Offset %d: 0x%08x\n", i, *((int *) (ram + i)) );
  }
  //  *((int *) 0x80000000) = 0x00000b33; /* write enable for prom */
  //  ram = (int *) 0x10000000; /* ie. prom */
  printf("Initiating transfer of 8 word from 0x%08x to 0x%08x...\n", (int) EthernetBaseAddr,
(int) ram);

  *mas_address =  (int) (EthernetBaseAddr);
  *mas_wc      = (6 << 8) | DMAWC;
  *dma_address = (int) ram;

  while (GETBIT(*int_st, 7) == 0);

  printf("Complete!\n");

  printf("Dumping 0x%08x...\n", (int) ram);
  for (i = 0; i < DMAWC; i++)
  {
    printf("Offset %d: 0x%08x - ", i, *((int *) (ram + i)) );
    if (*((int *) (ram + i)) == 0xc0000000 + i)
      printf("Checked out identical!\n");
    else
      printf("Not correct! Expected 0x%08x\n", 0xc0000000 + i);
  }
  printf("Read done!\n");
```

```
  printf("int_st=0x%08x\n", *int_st);

  return 0;
}
```

```
/* ========================================================================== */

// Program to perform direct PCI access using LDD/STD (OpenCores PCI core)

#include <stdio.h>

int main()
{

  double *dummyaddr;
  int data[100], indata[100];
  long long *longdata = data;
  long long *longindata = indata;
  volatile long long *targetaddr;
  int i;

  targetaddr = (long long *) 0xee010000;

  for (i=0; i < 100; i++) {
    data[i] = i;
  }

  for (i=0; i < 50; i++) {
    *( (long long *) (targetaddr + i) ) = *( (long long *) (longdata + i) );
  }

  for (i=0; i < 50; i++) {
    *( (long long *) (longindata + i) ) = *( (long long *) (targetaddr + i) );

  }
  for (i=0; i < 100; i++) {
    printf("read data %d\n", indata[i]);
  }
  /*
  while (1==1) {
    printf("is processor writing to UART\n");
  }
  */
}
```