Hello PCIex on Terasic DE2i-150

Patrick Schaumont

6/18/2013

Abstract

This design uses the Terasic DE2i-150 kit\(^1\) to demonstrate how to implement a PCIex communication link between an Atom processor and a Cyclone IV GX FPGA. This document covers installation and setup of the hardware/software development environment, development of the FPGA design, and development of the driver software. The design files are accessible at \(\text{http://rijndael.ece.vt.edu/de2i150}^2\).

1 Design Objectives and Tasks

The DE2i-150 board is a combined Atom/FPGA development platform with an Intel Atom N2600 with an Altera Cyclone IV GX 150 FPGA. The specifications of the board are available from Terasic’s DE2i-150 Development Kit pages. The FPGA is integrated on the Atom bus system through two 1x PCIex communication links. In the hellopci design, a program running on the Atom processor will read the position of the 16 switches on the DE2i-150, and display their setting as a hexadecimal value on the lower four hex displays of the board. Both of these peripherals (switches and hex displays) are controlled through the Cyclone FPGA, and accessing their value from the Atom processor requires Atom-FPGA communication over PCIex. This design effort makes no particular effort to optimize performance, but rather emphasizes design flow and connectivity. The document describes the following steps.

- Install and configure a development environment for Altera FPGA, and integrate the DE2i-150 FPGA hardware in this environment (Section 4.1).
- Install and configure a development environment for Atom processor, and integrate the DE2-150 Atom processor in this environment (Section 4.3).
- Develop an FPGA configuration that implements the interface between PCIex and the hex displays and switches (Section 5).
- Develop a PCI device driver to access the FPGA hardware from the OS kernel running on Atom (Section 6).

\(^1\)We acknowledge the generous support from the Intel Embedded University Program

\(^2\)The design files are for a DE2i-150 Rev C board, supported by V2.0.1 of Terasic’s DE2i-150 CDROM.
• Develop an application program for the Atom to read the switches and display their value on the hex display (Section 7).

There's also Quickstart Section that shows how to run the precompiled design, and a Section that explains setup of the development hardware.

# Quickstart

The design files expand to the following directory structure.

```plaintext
hellopci
  \quick Quickstart files
    \altera_driver.ko Kernel Module
    \app Atom Application
    \pcihello.sof FPGA bitstream
  \source Design Source Code
    \fpga Source Code for FPGA
      \pcihello.qar
    \driver Kernel Module
      \altera_driver.c
      \Makefile
    \app Atom Application
      \app.c
      \Makefile
```

If you have a configured and working DE2i-150 board running Yocto 8.0.2, you can run the design through the following steps.

1. Download the bitstream to the FPGA.
   ```bash
   DEVEL$ quartus_pgm -c USB-Blaster -m jtag -o "P;pcihello.sof"
   ```

2. Reboot the DE2i-150. Press the reset switch right next to the power plug on the board. Don't do a cold-reboot; you will lose the FPGA bitstream.

3. Copy the kernel module (`altera_driver.ko`) and application (`app`) to the board.

4. Install the kernel module.
   ```bash
   BOARD$ insmod altera_driver.ko
   ```

5. Create a device in the file system
   ```bash
   BOARD$ mknod /dev/de2i150_altera c 91 1
   ```

6. Run the application. While the application is running, toggling any of the lower 16 switches will change the lower 4 hex digits on the hex display.
   ```bash
   BOARD$ app
   ```
3 Development Hardware

I am using the following development hardware.

- A development workstation with Ubuntu 12.04 LTS. You will need a large disk to accommodate all development software (yocto + quartus take up close to 60G when a full build is performed).

- A USB cable between the development workstation and the USB Blaster port of the DE2i-150.

- An Ethernet switch between the development workstation and the Atom Ethernet port of the DE2i-150. The local network between the development workstation and the DE2i-150 simplifies file transfer (TFTP), and supports remote login to the board (SSH).

- A separate monitor and USB keyboard, attached to the Intel DE2i-150, can help to debug an unresponsive board.
4 Development Software and Configuration

This section enumerates the steps to install the development software.

4.1 Altera Quartus 12.1

The design uses the Altera Quartus 12.1 Web Edition development software, installed under /opt/altera/12.1. The Quartus software is available Altera. I modified my path variables for easy access to the Quartus tools by adding the following lines to /.bashrc.

```bash
alias n2cs="/opt/altera/12.1web/nios2eds/nios2_command_shell.sh"
export PATH=$PATH:/opt/altera/12.1web/quartus/bin
export PATH=$PATH:/opt/altera/12.1web/modelsim_ase/bin
export PATH=$PATH:/opt/altera/12.1web/quartus/sopc_builder/bin
```

4.2 FPGA configuration

After connecting the USB Blaster port to the development workstation, test if the device is visible with the lsusb command:

```bash
BOARD$ lsusb
```

When either of these steps fail, you need to debug the bitstream configuration link. It’s easy to find detailed explanations of this process online.

4.3 Yocto download and configuration

1. The embedded OS running on the Atom processor is Yocto: https://www.yoctoproject.org/. I used poky-danny-8.0.2 with an additional board support package. Download these two packages and install them under /opt/yocto:

```bash
DEVEL$ cd /opt/yocto
DEVEL$ wget http://downloads.yoctoproject.org/releases/yocto/yocto-1.3.2/poky-danny-8.0.2.tar.bz2
DEVEL$ tar jfxv poky-danny-8.0.2.tar.bz2
DEVEL$ wget http://downloads.yoctoproject.org/releases/yocto/yocto-1.3.2/meta-intel-danny-8.0.2.tar.gz
DEVEL$ tar xfvz meta-intel-danny-8.0.2.tar.gz
```
2. Next, configure Yocto to use the appropriate target (cedartrail).

```
DEVEL$ cd /opt/yocto/poky-danny-8.0.2
DEVEL$ source oe-init-build-end
```

3. In `conf/local.conf`, select `cedartrail-nopvr` as a target:

```
MACHINE ??= "cedartrail-nopvr"
```

4. In `conf/bblayers.conf`, add cedartrail support:

```
BBLAYERS ?= "\
 /opt/yocto/poky-danny-8.0.2/meta \
 /opt/yocto/poky-danny-8.0.2/meta-yocto \
 /opt/yocto/poky-danny-8.0.2/meta-yocto-bsb \
 /opt/yocto/meta-intel-danny-8.0.2 \
 /opt/yocto/meta-intel-danny-8.0.2/meta-cedartrail \
"
```

5. Compile yocto, as well as a cross-development environment for it:

```
DEVEL$ bitbake core-image-sato
DEVEL$ bitbake meta-toolchain-sdk
```

This will result in several packages in the `build/tmp/deploy` subdirectory from `poky-danny-8.0.2`. The yocto kernel is called `images/core-image-sato-cedartrail-nopvr.hddimg`. The toolkit is called `sdk/poky-eglibc-i686-i586-toolchain-gmae-1.3.2.sh`

6. Install the kernel on the board. The easiest method is to copy the image on a USB stick, boot the DE2i-150 board from the USB stick, and use the installation option of the live image to install Yocto on the SDD of the board. You can copy the image with `dd`. If your USB would be `/dev/sdd`, you would use:

```
DEVEL$ sudo dd \
    if=core-image-sato-cedartrail-nopvr.hddimg \
    of=/dev/sdd
```

A USB stick will not automatically be selected as boot target by the DE2i-150 board; you will have to interrupt the BIOS boot by pressing F10 and selecting the USB as boot target. You will need to attach a USB keyboard and a monitor to the board to achieve this. Furthermore, the live image on the USB disk will not install on the board by itself; you have to interrupt the USB boot process by pressing TAB and select the `install` boot option.
4.4 Local network configuration

The development workstation and DE2i-150 board are attached to a local network through their Ethernet connections. A configuration for a network 192.168.10.0/24 is as follows.

1. Assuming eth0 as the interface on the development workstation, you add the following in `/etc/network/interfaces`:

   ```
   auto eth0
   iface eth0 inet static
       address 192.168.10.1
       netmask 255.255.255.0
       network 192.168.10.0
       broadcast 192.168.10.255
   ```

2. On the DE2i-150 board, there’s a similar `/etc/network/interfaces` file, which is configured as follows. Note that this file marks the development workstation (192.168.10.1) as a gateway.

   ```
   auto eth0
   iface eth0 inet static
       address 192.168.10.10
       netmask 255.255.255.0
       network 192.168.10.255
       gateway 192.168.10.1
   ```

3. You can also add static DNS configuration by configuring `/etc/resolv.conf` on the board. This file can have the same contents as the `/etc/resolv.conf` on the development workstation.

4. You can access Internet on the DE2i-150 through the development workstation by using forwarding. Assume that you have two Ethernet cards in the development workstation: the eth0 interface is used for the local network, and the eth1 interface is connected to the Internet. Forwarding is configured as follows.

   ```
   DEVEL$ sudo iptables --flush
   DEVEL$ sudo iptables --table nat --flush
   DEVEL$ sudo iptables --delete-chain
   DEVEL$ sudo iptables --table nat --delete-chain
   DEVEL$ sudo iptables --table nat --append \
       POSTROUTING --out-interface eth1 -j MASQUERADE
   DEVEL$ sudo iptables --append FORWARD --in-interface eth1 -j ACCEPT
   DEVEL$ sudo echo 1 > /proc/sys/net/ipv4/ip_forward
   ```

5. After all of this is configured, reinitialize the network configuration on the development workstation with

   ```
   DEVEL$ sudo /etc/init.d/networking restart
   ```
and on the DE2i-150 board with

```bash
BOARD$ /etc/init.d/networking restart
```

Afterwards, test the configuration with a `ping www.google.com` on the DE2i-150.

6. Finally, you can also configure a TFTP daemon on the development workstation. This enables you to quickly transfer files from the development environment to the DE2i-150. Install TFTP on the development workstation using:

```bash
DEVEL$ sudo apt-get install xinetd tftp tftpd
```

We'll use the directory `/opt/tftp` on the development workstation as the transfer point. Add this configuration in the TFTP configuration file `/etc/xinet.d/tftp`:

```bash
service tftp
{
    protocol    = udp
    port         = 69
    socket_type  = dgram
    wait         = yes
    user         = nobody
    server       = /usr/sbin/in.tftpd
    server_args  = /opt/tftp
    disable      = no
}
```

TFTP is picky when it comes to permissions. The `tftp` directory in `/opt` needs to belong to user `nobody`, and have `777` permissions. Monitor `/var/log/syslog` for error messages pertaining to denied access.

7. As an example, to transfer a file `hello.ko` from the development workstation to DE2i-150, copy the file in `/opt/tftp` and use the following command on DE2i-150:

```bash
BOARD$ tftp -g -r hello.ko <ip_name_from_development_ws>
```

8. As an example, to transfer a file `data.txt` from DE2i-150 to the development workstation, first ensure that the file exists on the development workstation (use `touch /opt/tftp/data.txt`), and then transfer the file with the following command.

```bash
BOARD$ tftp -p -l data.txt <ip_name_from_development_ws>
```
5 Development of the HelloPCI FPGA Hardware

The FPGA platform for this design is created in QSYS, the platform design environment for Quartus. The design configuration includes the following components.

- A PICex hard macro interface module, configured as Avalon Memory-Mapped Bus Master.
- A 32-bit parallel I/O port with Avalon Memory-mapped Bus Slave interface. This port drives the hex display.
- A 16-bit parallel I/O port, with Avalon Memory-mapped Bus Slave interface. This port reads the switches.

The following are step-by-step instructions to compile an FPGA bitstream and configure it on the DE2i-150.

1. The project design file is `source/fpga/pcihello.qar`, and can be restored as a Quartus project. In Quartus, select Tools → QSYS, and load the platform configuration file, `pcihellocore.qsys`. The platform inspector will show the following design.
2. Double-click on the \texttt{pcie\_hard\_ip\_0} module to inspect the PCI\textsubscript{ex} interface settings. This design uses a basic configuration with a single 32-KByte memory space that drives both parallel I/O ports.

3. Go back to the platform inspector and examine the Avalon address map of the platform. The \texttt{pcie\_hard\_ip\_0} is a master on the Avalon bus and defines four regions: two for the parallel I/O ports, and two related to PCI configuration.

4. Go to the Generate tab and create Verilog code for the \texttt{pcihello\_core} design. Close QSYS when the conversion is complete.

5. In Quartus, inspect the top-level entity \texttt{pcihello}. This module instantiates the QSYS design (\texttt{pcihello\_core}) and connects it to the FPGA I/O pins. The pin assignment is stored in \texttt{pcihello\_qsf}; Terasic provides a program SystemBuilder to generate these pin assignments automatically based on a selected subset of peripherals.

6. In Quartus, compile the full design (Processing - Start Compilation). The result of the compilation is a bitstream \texttt{pcihello\_sof}, which can be configured on the FPGA.

7. The FPGA can now be configured with Quartus. Alternately, you can program the FPGA from the command line.

\texttt{DEVEL\$\ \textit{quartus\_pgm} \ -c \ USB-Blaster \ -m \ jtag \ -o \ "P;pcihello\_sof"}

8. After the FPGA is configured, you will need to reboot the board so that the Yocto kernel can recognize the new PCI device. Use \texttt{reboot} on a command line prompt on the DE2i-150, or press the reset button right next to the power connector on the DE2i-150.
6 Development of the PCI Device Driver

We will develop two layers of software to control the parallel I/O ports in the FPGA design. The first layer is a PCI Device Driver. This software is written as a kernel module. It deals with low-level hardware access. The second layer is the application that accesses the driver.

The PCI device driver will handle two tasks. First, it will take care of proper PCI driver registration and initialization of the basic memory-mapped access between the FPGA and the kernel. Next, it will convert this low-level memory-mapped model into a simple character-based device interface. The idea is that, after the driver is initialized, the FPGA can be accessed as a device /dev/de2i150_altera. Sending data to this device will write bytes to the parallel I/O port that drives the hex display; reading data from this device will copy bytes from the parallel I/O port that samples the switches.

1. Before compiling code, check if the Yocto kernel can see the actual FPGA with lspci. You should see a device as follows. More verbose output can be obtained with lspci -vv.

   BOARD$ lspci
   ...
   01:00.0 Non-VGA unclassified device: Altera Corporation Device 0004 (rev 01)
   ...

2. First, initialize the cross-compilation design environment with

   DEVEL$ source /opt/yocto/poky/1.3.2/environment-setup-core2-poky-linux

3. The driver source code is available on source/driver, and is listed as well in the Appendix. Since this is a kernel module, it must be compiled in our cross-compiled yocto environment against a kernel image. The SDK does not seem to provide these files. However, they are available from the build directory of the Yocto kernel. Consult the Makefile under source/driver for configuration instructions. In particular, you have to initialize an environment variable KERNELDIR before compiling the kernel module. This can be done on a single command line with

   KERNELDIR=/opt/yocto/poky-danny-8.0.2/build/tmp/work/
   cedartrail_nopvr-poky-linux/linux-yocto-3.0.32+git1+b\
   f5ee4945ee6d7498e6a16356f2357f76b5e2f0_1+1e79e03d115\
   ed17782ab53909a4f3555e434833-r4.1/linux-cedartrail-n\
   opvr-standard-build make

   Note that calling make without defining KERNELDIR will yield a module for the kernel of the development workstation instead of for the kernel of the DE2i_150.

4. The resulting module, altera_driver.ko, can now be copied to the DE2i_150 board with tftp. On the DE2i_150, open a command prompt and execute

   BOARD$ insmod altera_driver.ko
5. Verify that the driver is loaded with `lsmod`:

```
BOARD$ lsmod
Module       Size  Used by
altera_driver  2158  0
iptable_nat    3685  0
nf_nat         14774 1 iptable_nat
...
```

6. The driver prints debug messages to the kernel. Inspect those with `dmesg`:

```
BOARD$ dmesg
...
altera_driver: Found Vendor id: 41172
altera_driver: Resource start at bar 0: 80200000
altera_driver: char+pci drivers registered.
```

7. After loading the PCI driver, you need to create a file handle through which an application can access it. Do

```
mknod /dev/de2i150\_altera c 91 1
```

We can now complete the last step: writing and compiling the application.
7 Development of the Application Software

Thanks to the device driver, the application is very simple. The application will open
the FPGA device for read/write and execute a loop that repeatedly reads the switch
positions and renders them on the hex display. A simple recoding ensures that the 16
bits read from the switches occupy 4 hex digits on the hex display. The application
source is as follows.

```c
#include <unistd.h>
#include <fcntl.h>

unsigned char hexdigit[] = {0x3F, 0x06, 0x5B, 0x4F,
                        0x66, 0x6D, 0x7D, 0x07,
                        0x7F, 0x6F, 0x77, 0x7C,
                        0x39, 0x5E, 0x79, 0x71};

int main() {
    int i, j, k;

    int dev = open("/dev/de2i150_altera", O_RDWR);

    for (i=0; i>-1; i++) {
        read(dev, &j, 4);
        k = hexdigit[j & 0xF] |
            (hexdigit[(j >> 4) & 0xF] << 8) |
            (hexdigit[(j >> 8) & 0xF] << 16) |
            (hexdigit[(j >> 12) & 0xF] << 24);
        k = ~k;
        write(dev, &k, 4);
    }

    close(dev);
    return 0;
}
```

1. Compiling this application requires, just as for the kernel module, using the cross-
   compilation environment. Make sure you do at least once:

   ```bash
   DEVEL$ source /opt/yocto/poky/1.3.2/environment-setup-core2-poky-linux
   ```

2. Then you can compile with `make`:

   ```bash
   DEVEL$ make
   ```

3. After copying the application to the DE2i-150, you can run it. While it is running,
you can toggle the switches and observe the hex display render the hexadecimal
   representation of the 16-bit number programmed in the switches.

4. Press Ctrl-C to abort the application. The driver writes messages in the kernel
   log, and you can display those with `dmesg`:
BOARD$ dmesg
...
altera_driver: Found Vendor id: 41172
altera_driver: Resource start at bar 0: 80200000
altera_driver: char+pci drivers registered.
altera_driver: opened 1 time(s)
altera_driver: device closed.

5. If you change the kernel module and wish to reload it, you first have to unload
   the previous copy with rmmod:

   BOARD$ rmmod altera_driver

   The kernel log will respond with

   BOARD$ dmesg
   ...
   Goodbye from de2i150_altera.
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

MODULE_LICENSE("Public\nDomain");
MODULE_DESCRIPTION("Basic\nDriver\nPCIHello");

//-- Hardware Handles
static void *hexport;  // handle to 32-bit output PIO
static void *inport;   // handle to 16-bit input PIO

//-- Char Driver Interface
static int access_count = 0;
static int MAJOR_NUMBER = 91;

static int char_device_open ( struct inode * , struct file *);
static int char_device_release ( struct inode * , struct file *);
static ssize_t char_device_read ( struct file * , char *,
      size_t , loff_t *);
static ssize_t char_device_write ( struct file * , const char *,
      size_t , loff_t *);

static struct file_operations file_opts = {
      .read = char_device_read,
      .open = char_device_open,
      .write = char_device_write,
      .release = char_device_release
};

static int char_device_open(struct inode *inodep, struct file *filep) {
    access_count++;
    printk(KERN_ALERT "altera_driver: opened%dt ime(s)\n", access_count);
    return 0;
}

static int char_device_release(struct inode *inodep, struct file *filep) {
    printk(KERN_ALERT "altera_driver: device closed.\n");
    return 0;
}

static ssize_t char_device_read(struct file *filep, char *buf,
      size_t len, loff_t *off) {
    short switches;
    size_t count = len;
    // printk(KERN_ALERT "altera_driver: read %d bytes\n", len);

while (len > 0) {
    switches = ioread16(inport);
    put_user(switches & 0xFF, buf++);
    put_user((switches >> 8) & 0xFF, buf++);
    len -= 2;
}
return count;
}

static ssize_t char_device_write(struct file *filep,
        const char *buf,
        size_t len,
        loff_t *off) {
char *ptr = (char *) buf;
size_t count = len;
short b = 0;
// printk(KERN_ALERT "altera_driver: write \%d bytes\n", len);
while (b < len) {
    unsigned k = *((int *) ptr);
    ptr += 4;
    b += 4;
iowrite32(k, hexport);
}
return count;
}

//-- PCI Device Interface

static struct pci_device_id pci_ids[] = {
    { PCI_DEVICE(0x1172, 0x0004), },
    { 0, }
};
MODULE_DEVICE_TABLE(pci, pci_ids);

static int pci_probe(struct pci_dev *dev,
        const struct pci_device_id *id);
static void pci_remove(struct pci_dev *dev);

static struct pci_driver pci_driver = {
    .name = "alterahello",
    .id_table = pci_ids,
    .probe = pci_probe,
    .remove = pci_remove,
};

static unsigned char pci_get_revision(struct pci_dev *dev) {
    u8 revision;

    pci_read_config_byte(dev, PCI_REVISION_ID, &revision);
    return revision;
}
static int pci_probe(struct pci_dev *dev, const struct pci_device_id *id) {
    int vendor;
    int retval;
    unsigned long resource;

    retval = pci_enable_device(dev);

    if (pci_get_revision(dev) != 0x01) {
        printk(KERN_ALERT "altera_driver: cannot find pci device\n");
        return -ENODEV;
    }

    pci_read_config_dword(dev, 0, &vendor);
    printk(KERN_ALERT "altera_driver: Found Vendor id: %x\n", vendor);

    resource = pci_resource_start(dev, 0);
    printk(KERN_ALERT "altera_driver: Resource start at bar 0: %lx\n", resource);

    hexport = ioremap_nocache(resource + 0XC000, 0x20);
    inport = ioremap_nocache(resource + 0XC020, 0x20);

    return 0;
}

static void pci_remove(struct pci_dev *dev) {
    iounmap(hexport);
    iounmap(inport);
}

//-- Global module registration

static int __init altera_driver_init(void) {
    int t = register_chrdev(MAJOR_NUMBER, "de2i150_altera", &file_opts);
    t = t | pci_register_driver(&pci_driver);

    if(t<0)
        printk(KERN_ALERT "altera_driver: error: cannot register char or pci.\n");
    else
        printk(KERN_ALERT "altera_driver: char+pci drivers registered.\n");

    return t;
}

static void __exit altera_driver_exit(void) {
    printk(KERN_ALERT "Goodbye from de2i150_altera.\n");
}
unregister_chrdev(MAJOR_NUMBER, "de2i150_altera");
pci_unregister_driver(&pci_driver);
}

module_init(altera_driver_init);
module_exit(altera_driver_exit);