AES Co-Processor on Terasic DE2-150

Moein Pahlavan Yali
Patrick Schaumont

10/20/2013

Abstract

This design uses the Terasic DE2i-150 kit to demonstrate how to use the FPGA (Altera Cyclone IV GX) as a co-processor for Intel Atom N2600 processor. In this example, we use an AES encryption module in Verilog. To get familiar with installation and setup of the hardware/software development environment, read the introduction project on this board (Hello PCIex). This document assumes that you have the setup ready, and it goes through the details of design and implementation. The design files are accessible at http://rijndael.ece.vt.edu/de2i150.

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. The hellopci design demonstrates the basics of installation and the setup of the development environment. That design also demonstrates FPGA configuration, a PCI device driver as a Linux kernel module, and finally a C program that used the device driver to communicate to FPGA through the PCIex. In this design, we will use the FPGA as a co-processor for AES encryption. The main difference of this design with hellopci is that it uses a different interfacing technique. Instead of using a kernel module, we use mmap function that maps a region of physical memory to an address range in user-space. This way we can directly map a memory space from FPGA into an application C program running on CPU. This design effort makes no particular effort to optimize performance, but rather emphasizes on design flow and connectivity. The document describes the following steps.

• Explain the design approach.

1 We acknowledge the generous support from the Intel Embedded University Program.
2 Verilog implementation for AES is downloaded from http://opencores.org/project,aes_core authored by Rudolf Usselmann.
3 The design files are for a DE2i-150 Rev C board, supported by V2.0.1 of Terasic’s DE2i-150 CDROM.
• Develop an FPGA configuration that implements the interface between PCIex and the AES hardware module.

• Develop an application program including a user-space PCI driver that uses mmap to access PCI. The program sends an AES key and plaintext over PCI and reads the ciphertext back.

There’s also Quickstart Section that shows how to run the precompiled design. Note that if you have issues with setup or environment you can refer to our hellopni example.

2 Quickstart

The design files expand to the following directory structure.

\AES
  \quick Quickstart files
  \app Atom application
  \AES.sof FPGA bitstream
  \source Design files
  \fpga
    \AES.qar Quartus Project
    \app
      \app.c Application source
      \aes.h AES header
      \aes_locl.h AES header
      \aes_core.c AES source
      \Makefile Makefile

If you have a DE2i-150 board with Yocto 8.0.2, you can run the design through the following steps.

1. Download the bitstream to the FPGA.

   DEVEL$ quartus_pgm -c USB-Blaster -m jtag -o "P;AES.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. Run the application. Run the application. It runs one AES operation on the CPU and FPGA both and prints out the results.

   BOARD$ app

   The output should look like this:
3 Design

The design is simple and generic; the main goal for this design is to be understandable and reusable, rather than optimized and efficient. A high-level representation of the design looks like this:

On the software side, we only have the C application that is communicating with hardware side through PCIex. Of course, there are other software components in Linux Kernel that take care of this communication and that support the `mmap` instruction in the C program, but we consider these layers transparent to the developer for simplicity. On the hardware side, there is PCI IP compiler provided by Altera that translates a PCI interface to an Avalon interface, making it possible to have this PCI IP with other Avalon components on an Avalon Bus. On-chip memory can be used as a shared memory (between C and hardware) for exchanging data. There are also two parallel I/O ports that can be used for direct memory mapped communication between C program and custom logic. These ports are used to implement a two-way handshake, such that access to the shared memory can be controlled. The C program demonstrates execution of the AES coprocessor.
4 Development of the FPGA Hardware

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

- A PCIex hard macro interface module, configured as Avalon Memory-Mapped Bus Master.
- Two 32-bit parallel I/O port with Avalon Memory-mapped Bus Slave interface. These ports are used as a two-way handshake between C and custom logic.
- A dual-port on-chip memory used as a shared memory for sending inputs and output back and forth.

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/AES.qar, and can be restored as a Quartus project. In Quartus, select Tools → QSYS, and load the platform configuration file de2i150_core.qsys. The platform inspector will show the design.

2. Double-click on the pcie_hard_ip_0 module to inspect the PCIex interface settings. This design uses a basic configuration with a single 32-KByte memory space that drives both parallel I/O ports and the on-chip memory. Close

3. Double-click on the pcie_hard_ip_0 module to inspect the PCIex interface settings. This design uses a basic configuration with a single 32-KByte memory space that drives both parallel I/O ports and the on-chip memory. Close this window to go back to platform inspector.

4. See the AES_core_0 component and its connections. This is our custom logic that is connected to the system as a memory-mapped component. It has an Avalon master interface that connects it to the on-chip memory. You can also see status_port, command_port and other I/O connections. If you want to learn how to make a component with Avalon Interface, you can refer to the Altera tutorial for making QSYS components. If you want to see the details of our component you can right click on AES_core on the left panel (component library under Project category) and click edit. This might help you in case you want to create a component of your own.

5. Go back to the platform inspector and examine the Avalon address map of the platform. The pcie_hard_ip_0 is a master on the Avalon bus and defines five regions: two for the parallel I/O ports, one for on-chip memory and two related to PCI configuration. These addresses are important for application development.

6. Go to the Generate tab and create Verilog code for the de2i150_core design. Close QSYS when the conversion is complete.

7. In Quartus, inspect the top-level entity AES. This module instantiates the QSYS design (de2i150_core) and connects it to the FPGA I/O pins and the custom logic. The pin assignment is stored in AES.qsf; Terasic provides a program SystemBuilder to generate these pin assignments automatically based on a selected subset of peripherals.
8. You can inspect the `custom_logic_top` module to see how it implements a simple state machine to use an instantiation of AES module, as explained in the Design Approach section. Note that the interface for this module is generic and not specific to AES, so it can be used for any custom logic.

9. In Quartus, compile the full design (Processing - Start Compilation). The result of the compilation is a bitstream AES.sof, which can be configured on the FPGA:

```
DEVEL$ quartus_pgm -c USB-Blaster -m jtag -o "P;AES.sof"
```

10. After the FPGA is configured, you will need to reboot the board so that the Yocto kernel can recognize the new PCI device. Use 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.

11. Check if the Yocto kernel has recognized the new PCI device by using `lspci` command.

```
BOARD$ lspci
```

Look for the following line in output:

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

## 5 Development of the Application Software

The C program is in `source/app/app.c`. It contains everything except for the C implementation of AES\footnote{We have used the AES code used from the OpenSSL Project (http://www.openssl.org/)}. Our application consists of the following components:

- **Reading the PCIe base address:** Every PCI device connected to a PC is identified by a vendor id and device id. You can find the vendor id and device id of our PCIe in QSYS by clicking on the `pcie_hard_ip_0` component. We have defined them in our C source code:

  ```c
  #define PCI_VENDOR_ID 0x1172
  #define PCI_DEVICE_ID 0x0004
  ```

  In linux, you can find the list of all PCI devices by their id and physical address under `/proc/bus/pci/devices`. We have a function `pci_read_base_address` that finds the base address of our PCIe device by reading this file and looking for the given vendor id and device id. The function can be used as following:

  ```c
  int pci_bar0 = pci_read_base_address(PCI_VENDOR_ID, PCI_DEVICE_ID);
  ```
• **mmap:** We use `mmap` function to map the physical location of our PCIe to a user-space pointer that can be used in C program as a regular pointer. You can see the details of `mmap` in its manual page. The location being mapped should be represented by a file descriptor. We use `/dev/mem` as the file descriptor, because it provides access to the system physical memory. We use `mmap` as follows:

```c
int fd = open("/dev/mem", O_RDWR|O_SYNC);
u8* ptr = mmap(0, MMAP_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd, pci_bar0);
```

As we need to have read and write permission, we should open the file descriptor by `O_RDWR` directive. `O_SYNC` guarantees synchronous I/O.

`mmap` needs 6 arguments: Setting the first argument to zero means we have no preference for returned pointer; the second one defines the mapping size which in our case `MMAP_SIZE` is set to 64K; the third one is requesting read and write access for this mapping by using `PROT_READ`|`PROT_WRITE`; the fourth one is set to `MAP_SHARED` to make sure writes to the device is accessible to all processes (this is not crucial for this application, but might be important depending on the application); and the last two ones are our file descriptor and PCI device base address.

• **Read/Write Operations:** Once we have the device pointer, everything is as easy as working with a regular pointer. We just have to make sure we are dealing with the correct location. This is where we use the address map we observed in QSYS. We have define those addresses in our C program as macros:

```c
#define PCI_OUTPORT 0XC040
#define PCI_INPORT 0XC060
#define PCI_MEMORY 0XC000
```

We have also defined functions for read/write operations for easy reusability. For example, look at `pci_mm_read` function:

```c
// read n bytes from an address
inline void pci_mm_read(u8* dst, u8* devptr, int offset, int n){
    int i;
    for(i=0; i<n; i++)
        *(u8*)((int)dst+i) = *(u8*)((int)devptr+offset+i);
}
```

`pci_mm_write`, `pci_get_status` and `pci_send_command` similarly use the device pointer. We use these functions in the last piece, `pci_aes128` function, as following:

```c
// run AES operation
inline void pci_aes128(u8* ptr, u8* cipher, u8* key, u8* txt){
    // Write key and plaintext
    pci_mm_write(key, ptr, PCI_MEMORY+AES_KEY_OFFSET, 16);
    // Encrypt
    pci_mm_write(cipher, ptr, PCI_MEMORY+AES_CIPHER_OFFSET, 16);
    // Decrypt
    pci_mm_write(cipher, ptr, PCI_MEMORY+AES_CIPHER_OFFSET, 16);
}
```
pci_mm_write(txt, ptr, PCI_MEMORY+AES_PLAINTEXT_OFFSET, 16);

    // Send start signal
    pci_send_command(ptr, 1);

    // Wait till it’s ready
    while(pci_get_status(ptr)!=1);

    // Read ciphertext
    pci_mm_read(cipher, ptr, PCI_MEMORY+AES_CIPHERTEXT_OFFSET, 16);
}

The following are the steps to run the program on Atom.

1. Note that you should do a cross-compilation i.e. you are not compiling the code to run on the development machine but on the Atom. So you need to set the environment by using `source` command:

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

2. Go to AES/source/app directory and use `make` to compile the program:

```
DEVEL$ make
```

3. Copy the executable file (app) to the board.

4. Run the executable. (You might need to set the permissions by `chmod` before you can run it.)

```
BOARD$ chmod 777 app
BOARD$ ./app
```

The output should look like this:

```
PCI BAR0 0x0000 = 0xb771e000
KEY: 0112233445566778899aabbccddeeff
PLAINTEXT: 0 1 2 3 4 5 6 7 8 9 a b c d e 0
CIPHERTEXT(CPU): 54f364e7ea88ed927f71d0b669976a
CIPHERTEXT(FPGA): 54f364e7ea88ed927f71d0b669976a
cpu_time: 15us
fpga_time: 97us
```

6 Conclusion

We demonstrated an AES coprocessor on the DE2i-150 board, based on memory-mapped communication between a C program executing on the ATOM processor and a hardware module configured in an FGPA. PCIex communications become transparent after use of the `mmap` primitive. This design can be generalized (and optimized) for other settings, as well.