Politecnico di Torino - Electronic Engineering

Operating Systems

LAB Experience on the SHA1 accelerator

Authors:

Serra Jacopo (303510)
Emiliano Massimo (301114)
D'Elia Luca (296480)
Fagnani Pietro (303490)
Sineo Gioele (301398)
Massetti Marco (301163)

This work is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License.

Lab Experience

The purpose of this Lab is to use the SHA1 encrypting algorithm to verify the authenticity of a source file. It may happen that during the transmission of a file through a not protected channel, it can be altered by a malevolent attacker. To check that the file is compliant with the original one, you can encrypt it by using the SHA1 algorithm. As you know, whichever the length of a file is, if it is encrypted with this algorithm it returns a unique key of 160 bits. Then you can compare the returned key with the public key shared by the original owner of the file to check the authenticity. At this point you may think: "Nice, however...". That's why, to make the lab more challenging, since you aim to be an Electronic engineer in your bright future, you will implement the SHA1 accelerator phisically on a PYNQ-Z2 board kindly offered by the Politecnico of Torino. A further good news is that you are asked to do all this stuff in just one week. Now let's proceed together, step by step!

1.0.1 Directory organization

First of all, let's give a look at the files already available for you.

```
/
├── driver
│   ├── template_sha1_driver.h
│   └── template_sha1_driver.c
├── hdl
│   ├── sha1_v01
│   │   ├── dffhr.v
│   │   ├── sha1_exec.v
│   │   ├── sha1_readme_v01.txt
│   │   ├── sha1_round.v
│   │   └── sha1_testbench.v
│   ├── sequencer.v
│   ├── tb_top.v
│   └── top.v
├── vitis
│   └── template_HW_test.c
└── doc
    ├── tutorial.pdf
    └── Secure_Hash_Standard.pdf
```

You can find the hardware description of the SHA1 accelerator in the *hdl* directory. Refer to the files: *dffhr.v*, *sha1_exec.v*, *sha1_round.v*, *sequencer.v*, *top.v*. As you might expect, the top entity is inside the *top.v* file. Why don't you give a look to all of these files?
The *driver* directory and the *vitis* directory contain templates you will use later. The *doc* directory contains a tutorial for some of the software you will use and the description of the SHA1 algorithm.

1.0.2 Software Toolkit

Everything has been funny until now, but the first bad news is going to arrive. Are you ready? As the title suggests, you must download very large Software programs to interact with the board. If your laptop doesn't agree, you can still use the PC which you can find in the Lab when you have access to it. The programs to be used are:

- VIVADO: it is used to create the working hardware platform on the FPGA of the board.
- VITIS: it is used to connect the FPGA with the CPU of the board by using bare-metal code.

- PetaLinux: it offers everything necessary to customize, build and deploy Linux drivers on AMD processing systems. It is also used to compile a Lite version of Linux which contains your custom driver and the user application.

You can download the programs from `https://www.xilinx.com/support/download.html`

1.1 Synthesis of the Accelerator

The first step of this experience is to synthesize the provided accelerator on the FPGA mounted on board of the PYNQ-Z2. First of all, the VIVADO tool is required. Then, open it. At this point you can either RTFM (Read The *Friendly* Manual ;D), trying to figure out how to do your job, or you can follow the good-looking step-by-step tutorial already available on the folder *doc*. Yeah, I know what you are thinking of: "Damn, I wish I had more time to read the whole documentation!". Whichever solution you have chosen, the steps to be carried out are:

1. Create a new RTL project without including the HDL files
2. Select the PYNQ-Z2 board as a target board
3. Create a new Lite AXI4 interface peripheral
4. Add the provided HDL files as Design Sources
5. Instantiate and connect your SHA1 accelerator. It would be much easier if you use contiguous slave register to store a datum on more than 32 bits.
6. Elaborate the Design
7. Create your block Design and validate it
8. Generate the bit-stream and export it (.xsa file)

Now it's time to move to the VITIS software: you have to generate the interface between your FPGA and the CPU by using bare-metal code(you can again read the documentation or follow the tutorial). This time, the steps to be performed are:

1. Create a new platform from hardware to be interfaced with the ps7_cortexa9_0 processor
2. Write a test for your accelerator. Use the functions *xxx_mWriteReg* and *xxx_mReadReg* to write/read a reg of the platform *xxx*. Use the *xil_printf* function to print the output on the terminal. You can take inspiration from the template in the directory vitis.
3. Run the simulation in debug mode to check if the interface is correct and the accelerator is properly working.

Actually this is not a real driver: it's only a test for your accelerator, but it's very useful to become familiar with the device. If you have arrived up to here, take a break to admire your little child. If everything is correct, your baby SHA1 accelerator should have printed its first words. How cute it is! By the way, it's time to proceed to the second step. So, close VITIS, turn your board off and kill your baby because we're going to move to the fantastic world of the drivers!

Device Functionality	VFS function	Notes
reset	open	open() is used once to establish the connection with the device.
power-off	release	Terminate the connection with the device.
set_DIN	write	Send data to the DIN_register.
set_CV	write	Send data to the CV_register.
set_prev_CV	write	Modify the use_prev_cv bit in control_register
set_start	write	Modify the start bit in control_register
get_DIN	read	Read the content of the DIN_register.
get_CV	read	Read the content of CV_register.
get_control_reg	read	Read the content of control_register.
get_status_reg	read	Read the content of status_register.
get_DOUT	read	Read the content of DOUT_register.
none	ioctl	ioctl() is used to select the target for read/write operations

Table 1.1: Mapping of the device functionalities

1.2 Driver implementation

The second step is to generate the driver for your device. First of all you must outline the device functionalities of your accelerator and associate each of them to a VFS function. To do so, you can follow the association described in the table 1.1.
Starting from the templates for the driver, you can describe the implementation of each VFS function. Hereinafter some useful tips for the code:

- open: reset the device, set start bit to zero and set the CV to its default value
- ioctl: if you've followed the previous suggestion, you should have contiguous Slave registers for the storing of a big datum. At this point store on a variable the physical address of the Slave_Register_0 and the offset of the other significant slave_regs to ease the implementation of the read and write
- read: use the *copy_to_user* function; if you've used contiguous slave registers for the same datum, you can simply ask this function to read the correct number of bytes starting from the first slave register: it will move to the subsequent slave register automatically.
- write: use the *copy_from_user* function; if you've used contiguous slave registers for the same datum, you can simply ask this function to write the correct number of bytes starting from the first slave register: it will move to the subsequent slave register automatically.

1.3 Generation of the LKM in PetaLinux

Guess what? You should use a machine running your preferred distribution of GNU/Linux, or as I've recently taken to calling it, GNU plus Linux, to proceed with the next steps.

1.3.1 Installation steps

Before installing PetaLinux you have to check and possibly install all the Software dependencies. A software dependency is a relationship between software components where one component relies on the other to work properly. At the link https://support.xilinx.com/s/article/73296?language=en_US you can find a script which download all the necessary packages. Launch it by issuing the command

```
$ chmod +x ./plnx-env-setup.sh
$ sudo ./plnx-env-setup.sh
```

Now you can finally install PetaLinux by issuing the command (the version of the software may change):

```
$ chmod +x ./Xilinx_Unified_2022.2_0507_1903_Lin64.bin
$ sudo ./Xilinx_Unified_2022.2_0507_1903_Lin64.bin
```

1.3.2 SD partition

In order to boot the OS, you need to partition the SD into two sections:

- Boot partition: it stores the programs useful for the boot of the operating system
- Root partition: it stores the file system

To do it, follow the tutorial on the link https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842385/How+to+format+SD+card+for+SD+boot

1.3.3 Loadable Kernel Module

Add the installation path to the environment variable:

```
$ export PATH=/tools/Xilinx/PetaLinux/2022.2/tool/tools
/common/petalinux/bin:$PATH
```

Note: the path to the PetaLinux executable file may be different. Always use yours.
Initialize PetaLinux by launching the initilization script:

```
$ source /tools/Xilinx/PetaLinux/2022.2/tool/settings.sh
```

The last two commands must be repeated every time you restart your PC. Move to your working directory where you can create your project by issuing:

```
$ petalinux-create -t project --template zynq -n sha1
```

Copy the bitstream file (.xsa file) generated by VIVADO in the current directory. Enter the project directory

```
$ cd sha1/
```

Insert the hardware description of your accelerator in your project:

```
$ petalinux-config --get-hw-description=../sha1_design_wrapper.xsa
```

Configure PetaLinux in order to accept the partition scheme of the SD :

```
$ petalinux−config
```

Now modify a flag following the path:
image packing configuration → root filesystem type → ext4 (SD)
Enable the root user required to be able to mount the kernel module:

```
$ petalinux−config −c rootfs
```

Enable the debug tweaks following the path:
Image features → debug tweaks
Create your kernel module with:

```
$ petalinux−create −t modules −−name sha1_driver −−enable
```

At this point, PetaLinux generate some default files including an empty .c file. You can fill the .c file with the description of your driver or you can replace it with your source code and then you have to compile everything. If you have replaced it, remember to adjust the Makefile by updating the first line with the name of your source file. To do it you can follow the mini-tutorial at the link
https://xilinx.github.io/Embedded-Design-Tutorials/docs/2021.1/build/html/docs/Introduction/Zynq7000-EDT/8-custom-ip-driver-linux.html#example-12-device-driver-development
Now sit back and relax because you're going to compile everything:

```
$ petalinux−build
```

Probably few months later, PetaLinux will automatically places the kernel object in the file system partition. At this point you need to create the files required to boot the OS and to configure the FPGA:

```
$ cd images/linux
$ petalinux−package −−boot −−fsbl zynq_fsbl.elf
    −−fpga system.bit −−u−boot u−boot.elf −−force
```

1.4 User Application

Now it's time to try your device and the device driver: the next task is to write a user application. In a C file write the code to verify the authenticity of either a generic file or a generic string. The application outcome is a 160 bits key which can be used to verify the file/string authenticity with a given public key (i.e.g use an online tool to obtain this key starting from your test file). The main steps of your algorithm are:

- parse the file into N blocks of 64 bytes to be sent sequentially to D_IN register of the accelerator
- Apply the sha1 algorithm to each block, writing the DIN_registers, issuing the start command and waiting for the end of the encryption (remember to use the prev_cv from the second block on).
- Read the last generated CV, this is the file key

.
NOTE: For the parsing operation you can refer to the document *.doc/Secure_Hash_Standard.pdf*. Remember the padding and the final adjustments on the last block!
TIP: You might have some endiannes problem, use the __builtin_bswap32 built in function to reverse bytes inside a variable.

1.5 Load on the board

1.5.1 Compile the User application

Launch PetaLinux. Issue the command:

```
$ petalinux-create -t apps -n userapp --enable
```

Copy the source file inside */sha1/project-spec/meta-user/recipes-apps/userapp/files/* At this point you have to adjust the makefile and the .bb file, similarly to what you've already done in the paragraph 1.3.3.
Compile everything:

```
$ petalinux-build
```

1.5.2 Boot of the OS and execution of the user application

Now you have to copy the files inside the SD. Copy the following three files in the boot partition:

- BOOT.BIN
- boot.scr
- image.ub

Extract the archive *rootfs.cpio* in the root partition. You can find all these files in *./sha1/images/linux*. You can finally connect your board and turn it on. Use a serial console to interface with it. Login as a root user:

```
username : root
password : root
```

Mount the driver by issuing:

```
$ insmod /lib/modules/$(uname -r)/extra/sha1_driver.ko
```

Finally you can launch the user application: you can find it in */usr/bin/userapp*. Choose a file, encrypt it with an online SHA1 tool and take note of the key. Re-encrypt it with your board. Do they match?