Practical Introduction to Hardware Security

Task 0: Lab Introduction

Lattice iCE40-HX8K Breakout Board Setup Guide

Chair of Dependable Nano Computing, Karlsruhe Institute of Technology

Instructors: Dennis Gnad, Jonas Krautter, Mehdi Tahoori

April 27, 2022

1 Generic

Just follow the next step by step guide (it refers to the following sections). Please read the full document, and experiment further as we propose in Section 1.6.

1.1 Step-by-Step

This is a guide for Linux. The best option is to use native Linux, alternatively you can run it in a VM, but then you should make sure that USB 2.0 forwarding is activated and works. It will principally work, but there can still be issues with the standard USB 1.1 forwarding (even with USB 2.0 there can be rare lock-up issues sometimes). Some students were also successful to work with Windows/Mac directly. Packages for the FPGA tools exist for Windows/Mac, so it is not impossible. But there you also need python (we suggest *Anaconda*).
Please follow:

1. Get Task-0-source_files.zip from Ilias.
2. Download the oss-cad-suite from 2021-11-01.[1]. Please stick to this version for now, as there are sometimes still problems.
3. Extract to a common directory, for instance /Downloads/oss-cad-suite
4. Read Section 1.2.
5. Setup your path variable as described in Section 1.3.
6. Try the project as explained in Section 1.4
7. Program your FPGA board Section 1.5, please let us know in case of any problems.
8. Make sure basic_uart.py works (in Windows, check that you use Python 3 and not 2)
9. Experiment and practice further according to Section 1.6.

10. Try to do the TODOs in basic_uart.py

[1] https://github.com/YosysHQ/oss-cad-suite-build/releases/tag/2021-11-01

1.2 Tools and Project Overview

We supply you a basic Makefile-based project in `Task-0-source_files`, please read that Makefile to get an idea about it. Here we explain the basic steps and in the following subsections explain the details.

After getting the respective tools from the *toolchain-ice40* and *toolchain-yosys* git, the Makefile will use the respective yosys and nextpnr commands to generate a bitstream for the FPGA — after you added the PATH variable as explained in the next section. If you don't add the PATH variables, you might need to adjust the Makefile for the respective path(s) in which the ice40 and nextpnr toolchains are installed, but we recommend using the PATH variable.

After that, the bitstream file (top_level.bin) can be generated and programmed to the FPGA's SRAM. Then, the design can be accessed through UART using a terminal emulator. It works because this specific design includes a UART module, and the board has a USB-UART chip.

1.3 PATH variable in Linux

To 'install' the oss-cad-suite, simply extract them to a permanent location (i.e. /home/< username >/Downloads/oss-cad-suite). After that, you just need to source (with '.' operator) the respective environment file whenever you want to use the toolchain. Most probably you are using BASH and want to add an alias to your .bashrc. That means, edit the .bashrc in your home director (/home/<username>/.bashrc). Files starting with a '.' are considered hidden files in UNIX-like systems, so you might want to enable an option to see it. If you don't have a .bashrc, please don't start with an empty file, but copy the one from /etc/skel/.bashrc to your home directory, and then edit that. Simply add the following line to the end of the file (adjusted to where you put the downloaded toolchain(s)):

```
alias oss-cad-suite=". ${HOME}/Downloads/oss-cad-suite/environment"
```

After you have changed the .bashrc, every shell that you open **after** the change will be able to run 'oss-cad-suite', and after that was run once, that shell can run the respective commands that the project *Makefile* is using. That means, what is executed when entering 'make' in the project subdirectory.

1.4 Compiling the example project with make and UART port permissions

There will be issues in programming if no respective udev rules file is set up yet. Thus, please add a new file as root in /etc/udev/rules.d/ (for example, call it '99-lattice-hx8k.rules'), with the following content, adding all users of the group 'plugdev' to have access to the respective UART:

```
ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6010", MODE="660", GROUP="plugdev"
```

Now either put your user into the group 'plugdev', or you can use another group (for `GROUP=".."`) in which your user is already a member of. You can check the groups you are member of in the shell, with the command 'groups'.

After adding the file, reload the rules as root with:

```
sudo udevadm control --reload
```

Reconnect the board's USB if it was already connected. Then you can proceed to compile and program the bitstream, by first compiling it:

```
# cd Task-0-source_files/
# make
```

The output should look similar to the following:

```
yosys -q \
-L "top_level_yosys.log" \
-p "hierarchy -top top_level" \
-p "synth_ice40  -json top_level.json" \
top_level.v uart.v


... and so on ...
```

Ending with:

```
...

icetime -d hx8k -mtr top_level.rpt top_level.asc
// Reading input .asc file..
// Reading 8k chipdb file..
// Creating timing netlist..
// Timing estimate: 6.23 ns (160.52 MHz)
icepack top_level.asc top_level.bin
#
```

Now, please make sure the jumpers on the board are set correctly for SRAM programming, as shown in Figure 1(b). Then, please program the bitstream to the SRAM of the FPGA (if there is a problem, more details are in Section 2.1). In Linux, you type the following:

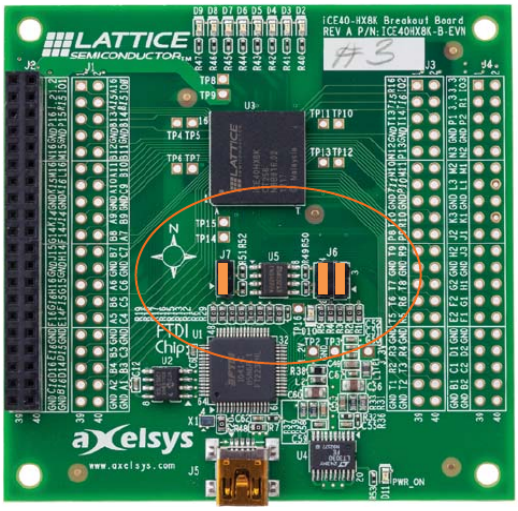
```
iceprog -S top_level.bin
```

Which can also be run from the Makefile (this also compiles the bitstream if it doesn't exist):

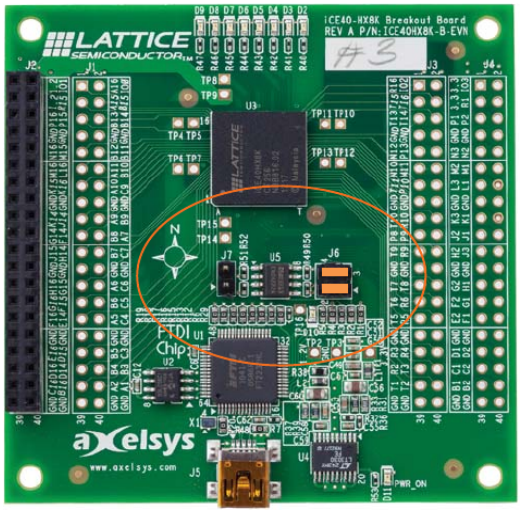
```
make prog
```

You should get an output like the following:

```
# make prog
iceprog -S top_level.bin
init..
cdone: high
reset..
cdone: low
programming..
cdone: high
Bye.
```



(a) Programming the flash memory on the board, which then gets transferred to the FPGA internal SRAM during power-up. (with 'iceprog' command or 'make prog-flash')



(b) Programming the SRAM memory on the FPGA chip directly. (with 'iceprog -S' command or 'make flash')

Figure 1: Jumper Settings to Program the Flash (slower but non-volatile) or SRAM (faster but volatile).

1.5 Communicating with the board through UART

After you managed to generate the bitstream file (top_level.bin) in Section 1.4, and successfully programmed it, we will now communicate with the FPGA through UART.

First, make sure you have the python 'serial' or sometimes 'pyserial' package installed in your system (for python version 3). In Ubuntu 20.04 this is done by the following (it is in some way also available in Windows in Anaconda, or in other Linux Distributions):

```
sudo apt-get install python3-serial
```

Then, please try the basic_uart.py python script. If that doesn't work, try out if the board responds by using a terminal emulator (see below in Section 2.2). There send an ascii 's' or 'S' (without any line ending like CR/LF). You will need to set up 115200 as the baud rate and probably select something like /dev/ttyUSB1 (in Linux, Windows would be COM3, COM4, or similar). In Linux you can watch the kernel log with 'dmesg -w'. Run that before you plug-in the board, and then check the output when you plug it in. You should see two ttyUSB# devices, where the second number # is the one you have to use in your terminal emulator and python script. Please consider that these numbers might also change again when something changes in your system.

If there are any issues, it can help to try installing the respective ftdi packages as mentioned in Section 2.1. But most of the time, something with your user group or udev rules might be still wrong.

1.6 First practices with Verilog and the FPGA board

In order to get some practice before you will get the first real task soon, we offer you some simple practices that you can try to fulfill. For that, please take a look into the Verilog code

in top_level.v and try to understand it. The next steps are to experiment and practice with it to get some basic 'feeling' for how it works. For a first try, just leave it as-is and run the basic_uart.py python script on the commandline, if not done already.

Next, please start to experiment/learn by trying to fulfill all the "`# TODO-Exercise:...`" written inside basic_uart.py. After that, do the following practices:

- Let LEDs blink differently,
- Try making an LED blink every second by counting up 12 million times, based on the 12MHz clock.
- Try answering with more than 1 character messages on the UART (as suggested inside basic_uart.py, try 'AB' instead of 'A') — here something needs to happen from one to the next clock cycle, by saving something across clock cycles.
- Try to go through multiple inputs on the UART (like a 'password entry' after which you light up a LED on success).
- Please also try other things that come to your mind! You can also ask us how you could implement a certain idea you have in mind.

Refer to the Verilog explanations from the introductory presentation, if required (03-Lab FPGA Intro.pdf) — also feel free to search for Verilog examples online. A good idea is to search for state-machines in Verilog, since performing logic that happens step-by-step over multiple clock cycles is pretty simple with a state machine.

2 Additional Software and Hardware Information

2.1 Programmer software troubleshooting

In some Linux distributions you might need to first install the *libftdi* package. For instance, in Ubuntu, do:

```
sudo apt-get install libftdi1 libftdi1-2
```

Also make sure the board has the respective jumper settings for either programming to the SRAM or Flash (usually we use SRAM, which is faster but will be gone after the FPGA is disconnected from USB). Jumper settings are shown in Figure 1 as is also explained in the iCE40HX-8K Breakout Board User Guide. Please check that guide if you need more information (cf. Section 3).

For programming, you can try the open-source software iceprog{.exe} directly instead of the Makefile. Use SRAM mode on the board (jumpers) and software (-S):

`iceprog -S <projectname>/top_level.bin` (should be also launched through 'make prog')

If you are using Windows, and if there are any problems, you can try two things:

- Run 'icestudio' in windows, mainly to guide you through the ftdi setup: https://github.com/FPGAwars/icestudio/releases — after that you don't need to use it anymore, and can try the Makefile / iceprog again
- Please ask us first before you do this, but it can be an option: Use the standalone Lattice Programmer software:
  http://www.latticesemi.com/latticediamond

2.2 Terminal emulation

This might only be needed if you encounter problems with your python scripts interfacing through the serial port.

Use any terminal emulator and make sure that you set the baud rate to "115200", without parity (typically the default). That is the default speed the initial project uses.
In **Windows**, you can, for example, use hterm or teraterm. Search and Download for them. For example:
https://www.heise.de/download/product/hterm-53283

In **Linux**, you can install cutecom (which has a GUI):

```
sudo apt-get install cutecom
```

screen (if you know how to handle it) also works, for example like this:

```
screen /dev/ttyUSB1 115200
```

3 Links

Lattice HX8K Breakout Board Main Website:
https://www.latticesemi.com/Products/DevelopmentBoardsAndKits/iCE40HX8KBreakoutBoard.aspx

Lattice HX8K Breakout Board Documentation / User Guide:
https://www.latticesemi.com/view_document?document_id=50373

Main open source toolchain to be used, which requires everything you need. Its commandline tools are used inside the Makefile of the example project and Makefiles of future tasks:
https://github.com/YosysHQ/oss-cad-suite-build/releases/tag/2021-11-01
yosys is a logic synthesis tool and nextpnr maps it to specific FPGA primitives

(so far not required) iCEcube2 vendor-specific proprietary software:
http://www.latticesemi.com/iCEcube2
This software also needs a (free) license that you can get by signing up at the Lattice website and request a license file bound to the MAC address of your ethernet adapter.