Introduction to the Hackaday Supercon 2019 Badge Hacking

Piotr Esden-Tempski

Version 0.1, 12 Nov 2019

Table of Contents

Overview

Course Objectives

This lab manual is a workbook that accompanies Hackaday Supercon FPGA Badge workshop.

Our objective is for attendees to get as much hands-on time with hardware as possible during our classes. Lectures serve as introductions to topics, but the walk-throughs and challenges in this manual will lead you to deeper understanding and higher retention of important information. We recommend recording as much as possible in this manual, as it may become a resource for your future hardware hacking adventures.

This training is based on the SecuringHardware.com model developed by Joe FitzPatrick. If you enjoy this wokrshop make sure to check out the list of workshops SecuringHardware.org offer. 1BitSquared also adapts and develops their own workshops for the hardware they manufacture and distribute. For example the iCEBreaker FPGA development board. You can find the WTFPga and iCEBreaker workshops on GitHub at github.com/icebreaker-fpga.

Chapter 1. Introduction

The Supercon 2019 Badge is centered around the Lattice Semiconductor ECP5 FPGA. As far as we know this is the very first time that a conference decided to put an FPGA on their official badge! On top of that they chose an FPGA that is supported by the Open-Source FPGA flow that runs on any OS you might choose making it a very accessible FPGA platform. If you like those ideas make sure to hack on your badge, spred the word what you did and that you like that idea. Maybe other conferences will get inspired too.

This workshop is a self-guided hands-on crash-course of how to program the "soft core" and it’s peripherals on your badge. We will also briefly venture into building an alternative hardware design for our badge to see what amazing power is given to us by being able to modify the hardware without picking up a soldering iron or learning how to make your own semiconductors. ;)

1.1. Hardware Overview

Besides the FPGA part and if you grew up in the 90' you will recognize the shape. It is one of the most iconic handheld game consoles. That is still not enough to make the board interesting though. We will need some more chips and connectors to make it an interesting device.

The badge hardware consists of the following components:

- Lattice Semiconductor ECP5 FPGA: LFE5U-45F
- Battery Power supply
- 128MBit (16MByte) Serial Flash
- Two 64MBit (8MByte) Serial RAM
- 320 x 512 resolution LCD
- Mono Audio amplifier
- Eight Buttons
- Eleven RGB LEDs
- IrDA transceiver

And a bunch of connections to the outside world:

- Micro USB Connector
- HDMI Video Connector
- Cartridge Port
- Two SAO Connectors
- Pmod Connector
- Speaker Connector

- JTAG/UART Connector

insert hardware legend here

1.2. Gateware Overwiew

The badge hardware by itself is not very useful by itself. We need a hardware design loaded into the FPGA. You can either let the logic itself do things or you can load a "Soft core" system on chip. Essentially programming the FPGA to behave like a microcontroller.

Sprite_tm as the lead developer with contributions from the community put together a pretty cool retro gaming console inspired architecture. At it’s core are two Open Source processor cores called picorv32, which in turn are an implementation of the open RISC-V Instruction Set Architecture (ISA) specification.

The processor cores would not be very useful without any peripherals. So the official badge soc comes with the following peripherals:

- Linerenderer graphics subsystem (outputs video to the LCD and HDMI at the same time)
  - Indexed color map with up to 256 24bit colors
  - Five graphics layers with 8 bit alpha channel
    - Background
    - Framebuffer
    - Tilemap A
    - Tilemap B
    - Spritelayer
  - Amiga style Copper transformations
- Audio subsystem
  - PCM
    - 16 bit mono audio
    - configurable audio
    - FIFO based with interrupt support
  - Synthesizer
    - 16 voices
    - Sawtooth, Triangle, Pulse, Sub-Harmonic waveworms
    - Configurable waveforms with subdivisible 4k wavetable
    - Per-voice volume control
    - Configurable attack/release time

  - Queued / Timed mode available (FIFO based with interrupt support)
- UART (aka. serial)
- IrDA (aka. infrared wireless)
- Random Number Generator
- Delta-Sigma ADC to measure battery voltage
- USB, implements DFU bootloader and file system access to the flash

1.3. Software Overview

Your badge is coming with some software that is running on the Gateware. It consists of thee main components:

- DFU Bootloader (allows us to reflash the Gateware)
- IPL (Initial Program Loader) providing application file management over USB and loading your user applications
- User Applications

Chapter 2. Setting Up

2.1. Get the toolchain

If you did not already you will need to install the FPGA and RISC-V toolcahins first.

You can download precompiled toolchain from:

https://github.com/xobs/ecp5-toolchain/releases

Download the version for your OS and make sure the `bin` subdirectory is in your $PATH environment variable. When you are done with that test the installation by making sure you can run `yosys`, `nextpnr-ecp5` and `riscv64-unknown-elf-gcc` from your command line.

For additional information about tool installations you can also refer to the badge documentation on github [https://github.com/Spritetm/hadbadge2019_fpgasoc#how-to-use].

You still might need a few additional tools installed on your system. Here is the list of the tools you might need to install too:

- git
- make

let us know if we missed any other dependencies here :D

NOTE On Linux you might need to add a udev rule so that you get the needed permissions to access your badges USB interface. You will need to open `/etc/udev/rules.d/59-hadb19.rules` in a text editor and add the following rules in that file. When you change the rules make sure to unplug and plug in your badge again.

```
ATTR{idVendor}=="1d50", ATTR{idProduct}=="614a", MODE="660", GROUP="plugdev",
TAG+="uaccess"
ATTR{idVendor}=="1d50", ATTR{idProduct}=="614b", MODE="660", GROUP="plugdev",
TAG+="uaccess"
```

2.2. Get the workshop repository

You probably already have it on your drive as you are reading this, but for good measure here is how you obtain the workshop git repository:

```
git clone --recursive https://github.com/esden/hadbadge2019_workshops.git
cd hadbadge2019_workshops/base
```

2.3. Build your first app

Now that you have the toolchain and workshop repository installed we can try uploading your first app to the badge.

- The skeleton program can be found in `hadbadge2019_workshops/base/app-basic-workshop`.
- Run `make` and if everything goes well you should find a `basic-workshop.elf` in the same directory.
- Power off your badge.
- Plug in your badge into your computer over USB.
- Power on the badge.
- Your computer should automaticall mount your badge and appear as a drive.
- Copy over the `basic-workshop.elf` to the badge.
- Power the badge off and on again.
- The badge should show a list of files and one of them should be the newly uploaded `basic-workshop.elf` file.
- Press the `A` (`SW6`) button.

Does it work? If you have any questions or run into trouble let any of the helpers know, we are here to help. If you are working on this outside of the Supercon workshops join the hackaday.io chat or the 1BitSquared discord.

NOTE	Make sure that when you are copying the `.elf` file to the badge that the copy process is really finished. In some cases the operating system buffers write operations to disks. This is meant to speed things up in normal use. In such case, the copying process might indicate that it finished, but the file is not actually fully written to the badge. On linux you can make sure the write process is finished by calling `sync`. This command will flush all your drive buffers and make sure data is all written physically to your drives.

Chapter 3. First FPGA Badge Demo

Now that we learned a bit about the badge and confirmed that the all the needed tools installed correctly we can get down to business and write a small demo effect. The badge Gateware design is on purpose designed to resemble a retro game console drawing on computer architectures from the 80' and 90' remixed with modern open source tools and cpu design.

Our first program will be inspired by the Demoscene culture. We will be writing a small video effect that is rendering the graphics on the fly. No video file playback here! Demoscene is the art of showing your programming and artistic skills at the same time. But don’t worry this one is very easy.

First we have to learn a bit about the video hardware on the badge. We will gloss over most of it in sake of saving time and only talk about the hardware features we will need for our demo.

If you look at our existing example program it consists of one `main.c` file. First thing you can see at the beginning of the `main` function is that we are using some strange macros `GFX_REG(GFXBGNDCOL_REG)`. This macro is expanded by the compiler to a specific addrss in the memory of our microcontroller. That address is wired directly to the video graphics hardware implemented in our FPGA.

If we write to that register we are setting configuration of our graphics hardware that then runs in the background and displays things to our LCD and HDMI video outputs.

The graphics hardware on the badge is based on multiple layers that are being automatically blended together, this way we don’t have to have a very fast CPU that can quickly update the video memory to generate visual effects.

We have 5 layers available:

The background layer: it is a solid color that fills the screen. This is what is being set up first at the beginning of our program.

The framebuffer layer: It is a bitmap that we can address each pixel individually to draw graphics on.

Two tilemap layers: Maps of tiles composited together by the video hardware to draw graphics out of pre defined graphical blocks. They are always in a grid.

Sprite layer: On this layer we can draw freely placable graphic elements.

Currently the example program is only using the background and the first tile layer. It also uses a convenience API on top of the first tile layer that acts like a text console. You can write to it and manipulate it by using the `fprintf` function and escape codes.

3.1. Play with the text console

When we start our program IPL leaves us with a set of tiles that represent characters. Opening the `/dev/console` allows us to easily write text to the screen. As you see our code already uses an escape sequence. The console supports some more Escape sequences:

- `\033nX` set cursor x position at `n`
- `\033nY` set cursor y position at `n`
- **todo** add more but need to test them…

There is a bunch more, you can find their implementation in `hadbadge2019_workshops/base/fpgasoc/soc/ipl/gloss/consol_out.c`.

Try writing a nice greeting text in the center of the screen.

3.2. Create a framebuffer

Most graphics system except the background color is based on indexed color. Instead of storing 24bits of color information for each pixel of tiles sprites and the framebuffer, we are only storing an index that points to a color palette with 256 entries. The graphics hardware on the fly converts the indexes into real 24bit colors that are then sent to the LCD or HDMI interface.

For our actual demo effect we will setup the framebuffer and draw our effect there.

NOTE	I know it is bit hypocritical after raving about how the tile and sprite layers save us processing power but this is what I fealt like plaing with, so here we go. On the other hand it will show you why you want hardware acceleration at the end, so bare with me. :)

You can add the framebuffrer memory allocation and config code right after the background config.

- First we need to do is allocate some memory:

```
fbmem=calloc(FB_WIDTH, FB_HEIGHT);
```

The nice sideeffect of using calloc is that the memory will be initialized with 0 so we don't have to do that in a separate step.

- Second we have to tell the graphics hardware about the newly allocated framebuffer memory:

```
GFX_REG(GFX_FBPITCH_REG)=(17<<GFX_FBPITCH_PAL_OFF)|(FB_WIDTH<<GFX_FBPITCH_PITCH_OFF);
```


The first field in this register tells the graphics hardware that the framebuffer is using colors stored in the pallette from index number 17 to 255. This is essentially an offset added onto the color indecies stored in the framebuffer.

The second field tells the graphics hardware how wide the framebuffer is.

- Third we have to tell the graphics hardware where our framebuffer memory is located:

```
GFX_REG(GFX_FBADDR_REG)=((uint32_t)fbmem);
```

3.3. Create the color palette

As described in the previous section, the framebuffer is using indexed color. To save you some time we have left a function to configure the color palette in the codebase. All you need to do is call the `crate_fire_palette();` function.

This function steps through the palette entries and sets up the assigned colors.

> **TIP** To achieve some demo effects you can draw on the layers once and only manipulate the color palette entries to achieve interesting animated effects without having to manipulate a lot of memory. See for example "plasma" effect.

3.4. Draw the palette on the screen

Now you can try out the palette that you have created. Just draw

3.5. Draw random fire seeds

xxx

3.6. Render a frame af fire

xxx

3.7. Loop the animation

xxx

3.8. Optimize framerate

xxx

Chapter 4. A new SOC hardware

xxx

4.1. Build and load the new SOC

xxx

4.2. Use newly added hardware

xxx

Chapter 5. Conclusion

xxx