

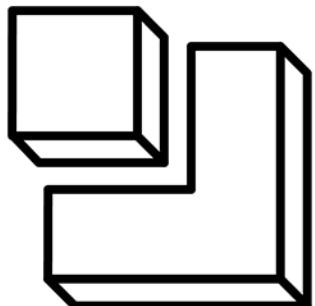


HACKADAY

Introduction to the Hackaday Supercon 2019 Badge Hacking

Piotr Esden-Tempski, Sophi Kravitz, Sylvain Munaut, Mike Walters and friends

Version 0.1, 12 Nov 2019



1 BIT SQUARED

Table of Contents

Overview	3
1. Introduction	5
1.1. Hardware Overview	5
1.2. Gateware Overwiew	6
1.3. Software Overview	7
2. Setting Up	9
2.1. Get the toolchain	9
2.2. Get the workshop repository	9
2.3. Build your first app	9
3. First FPGA Badge Demo	11
3.1. Play with the text console	12
3.2. Create a framebuffer	12
3.3. Create the color palette	13
3.4. Draw the palette on the screen	13
3.5. Draw random fire seeds	14
3.6. Render a frame of fire	14
3.7. Loop the animation	15
3.8. Optimize framerate	15
4. A new SOC hardware	17
4.1. Build and load the new SOC	17
4.2. Use newly added hardware	17
5. Final notes	19

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 workshop make sure to check out the list of workshops SecuringHardware.com offers. 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 <https://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, spread the word of what you did and that you like the 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 its 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. ;)

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 Supercon Chat](https://hackaday.io/messages/room/280647) [<https://hackaday.io/messages/room/280647>] or the [1BitSquared discord](https://1bitsquared.com/pages/chat) [<https://1bitsquared.com/pages/chat>].

1.1. Hardware Overview

If you grew up in the 90s you will certainly recognize the shape. It is one of the most iconic handheld game consoles. That alone is not enough to make the board interesting, though. We will need some more chips and connectors to make it an interesting device.

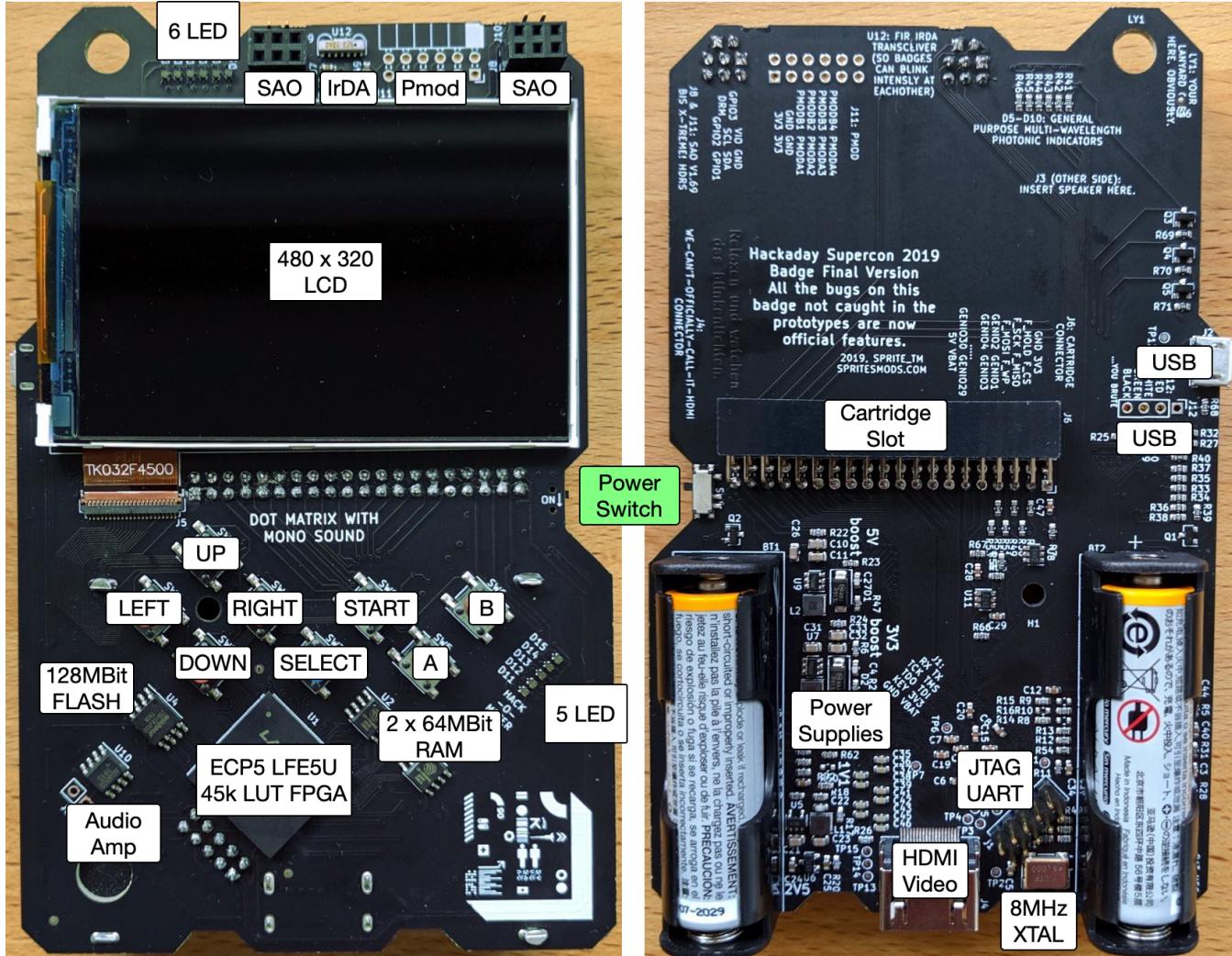
The badge hardware consists of the following components:

- Lattice Semiconductor ECP5 FPGA: LFE5U-45F
- Battery Power supply
- 128MBit (16MByte) Serial Flash
- Two 64MBit (8MByte) Serial RAM
- 320 x 480 Resolution LCD
- Mono Audio Amplifier
- Eight Buttons
- Eleven 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



1.2. Gateware Overview

The badge hardware by itself is not very useful. 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 its 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
 - Sprites
 - Amiga style Copper transformations
- Audio subsystem
 - PCM
 - 16 bit mono audio
 - Configurable sample rate
 - FIFO based with interrupt support
 - Synthesizer
 - 16 voices
 - Fixed waveforms: Sawtooth, Triangle, Pulse, Sub-Harmonic
 - Configurable waveforms: 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 comes with some software running on the Gateware. It consists of three main components:

- DFU Bootloader (allows us to reflash the Gateware)
- IPL (Initial Program Loader) providing application file management over USB and the ability to load and run 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 toolchains first.

Follow the instructions in the official badge [FPGA SOC Repository](https://github.com/spritetm/hadbadge2019_fpgasoc#how-to-use) [https://github.com/spritetm/hadbadge2019_fpgasoc#how-to-use].

If you have trouble with internet connectivity you can check the USB stick in your swag bag, the zip files containing the tools are on there. You will need to unzip them and add the **bin** directory to your **PATH** environment variable.

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/basic
```

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.

1. The skeleton program can be found in [hadbadge2019_workshops/basic/app-basic-workshop](#).
2. Run **make** and if everything goes well you should find a **basic-workshop.elf** in the same directory.
3. Power off your badge.
4. Plug in your badge into your computer over USB.
5. Power on the badge.
6. Your computer should automatically mount your badge and make it appear as a drive.
7. Copy over the **basic-workshop.elf** to the badge.
8. Power the badge off and on again.
9. The badge should show a list of files and one of them should be the newly uploaded **basic-workshop.elf** file.
10. Press the **A (SW6)** button.
 - Does it work?

- What does the application do?
- Look into the `main.c` file and check if what you see on the LCD is what you should expect.

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 all the needed tools are 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 80s and 90s 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 address 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 configuration 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 composed 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 placeable 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/basic/fpgasoc/soc/ipl/gloss/console_out.c](https://github.com/hadbadge2019/workshops/basic/fpgasoc/soc/ipl/gloss/console_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 converts the indexes into real 24bit colors on the fly which 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 felt like playing with, so here we go. On the other hand it will show you why you want hardware acceleration at the end, so bear with me. :)

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

- First thing we need to do is allocate some memory:

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

The nice side effect 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 palette from index number 17 to 255. This is essentially an offset added onto the color indexes stored in the framebuffer. We need to leave the first 16 color palette entries for the tiles used for the text console. Down the road if you don't want to use those tiles, or replace them you can use the full color palette.

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);
```

- Fourth we have to tell the graphics hardware to enable the framebuffer layer. You can do that by adding two more bits to the `GFX_LAYEREN_REG` line `GFX_LAYEREN_FB_8BIT|GFX_LAYEREN_FB`. These two bits tell the graphics hardware to enable the layer and that we are using 8 bit pixel values. The alternative would be 4 bit values but would limit us to 128 palette colors.

If you are interested in learning more about the graphics subsystem. The documentation is available here: https://github.com/Spritetm/hadbadge2019_fpgasoc/blob/master/doc/gfx.md

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 `create_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 then you only need to 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 fill the framebuffer memory with values from 0 to 238. Remember that we only have 239 indexes available as the first 16 are used by the text console tiles.

You can use the `FB_PIX` to make access to the frame buffer pixel entries bit easier.

Another important step is to flush the cache. By default to speed up accesses, the CPU doesn't write back all the data immediately into the memory chips but keeps them locally in a small memory on the FPGA itself. But if you want the pixel values to actually be seen on screen, they need to be

written out to the actual RAM chips where the GPU can see them. For this we need to "flush the cache":

```
cache_flush(fbmemp, fbmem+FB_WIDTH*FB_HEIGHT);
```

How does it look? :D

3.5. Draw random fire seeds

Now that we have the frame buffer set up and the color palette ready we can start working on our fire effect.

First we need to populate the bottom most row of the framebuffer with random white/black pixels.

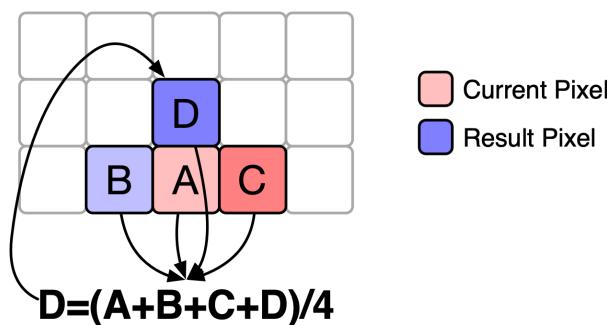
We can use another handy hardware feature. The random number generator. If you read a value from [MISC_REG\(MISC_RNG_REG\)](#) you will be provided with a word full of random bits.

Use those random bits to fill the last line of the framebuffer ($y = FB_HEIGHT - 1$) with 0 or 238.

3.6. Render a frame of fire

Now we can implement the actual fire effect.

- Iterate over all the pixels starting with the row that we just seeded with white pixels.
- Average (add together and then divide by the amount of values added: $\text{avg} = (a+b+c+d)/4$) the value of:
 - The pixel you are on
 - The pixel to your left
 - The pixel to your right
 - The pixel above you
- Decrement the value of the resulting average by 1 if the value is greater than 0.
- Store the resulting value to the pixel above your position.



NOTE

Don't forget to handle the first and last columns accordingly. For example don't add the pixel to your left when you are right at the left edge. ;)

You can put all that code into a function so it is easier to call over and over again and to make your code easier to read.

How does it look?

3.7. Loop the animation

Now that we have created one static frame of our fire effect. Try to write a loop function that keeps calling the draw function consisting of the random seed and render parts we wrote in the last two sections.

You can check for a button press in that loop to exit your program gracefully.

The current state of the buttons is stored in `MISC_REG(MISC_BTN_REG)` you can `&` its value with `BUTTON_START` to mask out the other buttons. You can find the other defines and register definitions in [hadbadge2019_workshops-git/basic/fpgasoc/soc/ipl/gloss/machDefines.h](#).

3.8. Optimize framerate

Now that you have implemented the fire effect and it is animating you are probably a bit disappointed about the framerate. Besides finding a faster processor we can also optimize our code a tiny bit.

One possibility is to exit the rendering loop when we realize that all the pixels on the row we are on are equal to 0. This will essentially mean that there is only black left all the way to the end of the screen and there is nothing else for us to do.

NOTE

Now that you arrived here there is less of a chance that we will spoil anything by telling you where to find our reference implementation of the effect. You can find it in the badge fpgasoc repository under the name `app-fire` and in the workshop repository in the `solution` directory. If you or your neighbor gets stuck someone will probably point you to this note so you can take a look at the reference implementation. :)

3.8. Optimize framerate

Chapter 4. A new SOC hardware

Now I know you are probably asking yourself why we went through all this. We could probably have done most of that more efficiently and with less trouble on a microcontroller with modern features.

Well here we go. I don't think many microcontrollers can grow new hardware features. :D

4.1. Build and load the new SOC

Go into the `fpgasoc` subdirectory and check out the `comb_vid_filter` branch by running `git checkout comb_vid_filter`.

In this branch we have added one more hardware feature to the graphics subsystem. Can you figure out what it is? You can see the diff by running `git diff master`.

To build and load the modified SOC make sure your badge is running the IPL and is connected to your computer over USB and run the following commands:

```
cd soc
make dfu_flash_all
```

This build process will take a while. It is synthesizing the new bitstream for your FPGA, building the new IPL (which should not be really necessary in this particular case) and finally using DFU to load the new code onto your badge.

In some cases the synthesis fails with a timing error. In such case just run the `make` command once again and it should finish the process.

The badge should now be behaving the same way as before.

4.2. Use newly added hardware

If you read through the diff you will see that there is a new definition in the `gloss` hardware register header. It is called `GFX_LAYEREN_FLT_GRAY` and can be set in the `GFX_LAYEREN_REG` register.

Change your demo code to toggle that bit.

What happens?

This is a very very simple demo of what you can do if you can actually modify the hardware you are using.

If you want to learn more about how to write some verilog continue to the advanced class where we

build some logic! :D

Chapter 5. Final notes

I hope you had fun with this workshop. There is so much more to learn about the badge itself as well as FPGA development.

Make sure to take part in the badge hacking that is happening the whole time during Supercon.

Shameless plug: If you want to have some more FPGA hardware to play with give the [iCEBreaker](#) [<https://1bitsquared.com/products/icebreaker>] a look. It is designed with FPGA beginners and developers in mind. It uses a smaller FPGA than the one on the badge but it's still plenty for a lot of applications.

See you around! :D