# INTRODUCTION

### Knowing each other ...

As an electrical engineer, I am mainly interested in technical matters. During my student life and throughout my working life, it became clear to me that FPGAs are a real innovation that will help shape the future. As a result, I started to get involved in that technology through the Altera Cyclon series and the excellent kits delivered by Terasic (DE2, NEEK, DE0 Nano )

As a consequence, I decided to get my hands on a DE10 Standard kit to continue learning good skills on SoCs. It took just 4/5 days to receive the kit in pristine condition. Everything was carefully packaged and there was no damage whatsoever. As soon as I received the kit, I couldn't resist to open it right away and try it out. A quick look at the DE10 Standard CDRom documentation gave me a good idea of where to start. Moreover, there are plenty of very good websites and forums out there since Terasic has been a legacy dealer of Altera FPGAs training kit for quite a while.

* <http://www.terasic.com.tw> - the Terasic website, plenty of information available.
* <https://rocketboards.org/> - SoC Linux oriented.
* <http://www.alterawiki.com/wiki/Main_Page> - Plenty of informations on bootloader, fpga, NIOS, compiler ...
* <http://www.altera.com> - the Altera website.
* <http://www.fpga4fun.com> - very good to learn the basics.

#### Prerequisite

In order to keep that review as short and digest as possible, I will assum that the reader is already familiar with the following documentation present in the DE10 Standard CDRom : DE10-Standard\_Getting\_Started\_Guide.pdf (expecially chap. 5 : "Running Linux"). In case not, I strongly advise the reader to skim through that document beforehand.

#### What to do with the DE10 Standard kit ?

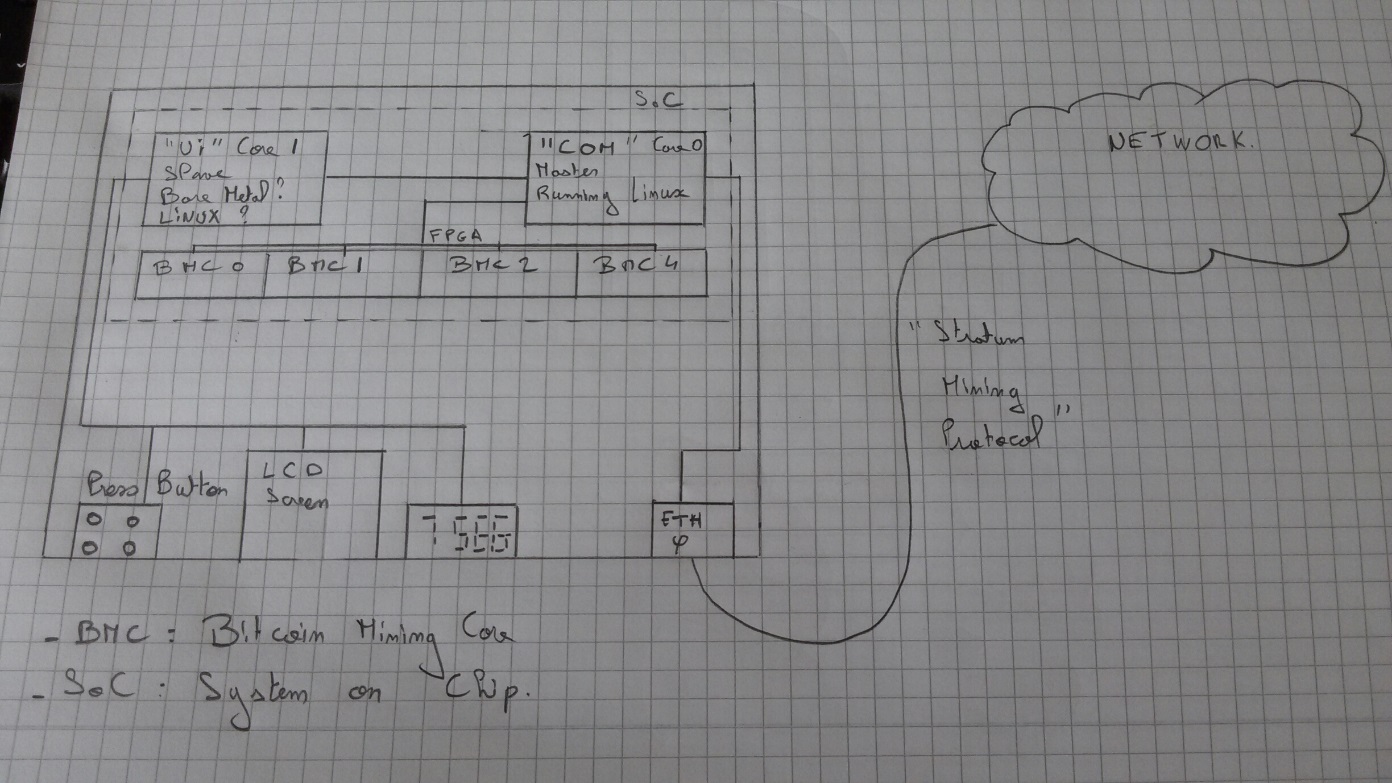
What could I use as a support for a good example ? An example that came to my mind and would fit the needs of an SoC, typically, is bitcoin mining. In fact, I already have an experience using FPGAs and Bitcoin that can be reused, see the (<https://github.com/BadissDjafar/BitcoinMining>).

Bitcoin mining consist in receiving a 512 bytes block of data in hex format and given that block, Find a golden "nonce" i.e a smaller 4 bytes block with enough leading 0s. Then once that "nonce" is found, it has to be packaged in a 512 bytes block and sent to the blockchain, wich is the mechanism ensuring the trustworthiness of the whole bitcoin scheme.

The challenge consist in being able to compute as much "nonce" as possible (based upon the informations received in the 512 bytes block) and find the so called golden "nonce". The computation of a "nonce" consist in a signature calculation based on the SHA256 algorithm. It is commonly refered to the output of that algorithm as a "Hash". Subsequently, the challenge can be summarised in the following sentence :

How many hashes can be computed in a given time frame (typically MHashes per second or GHashes per second) ?

That is why BMCs (bitcoin mining cores) are very good at it because of their big number crunching capabilities.

[](https://github.com/BadissDjafar/DE10_REVIEW/blob/master/SCREENSHOTS/Functional_design_new.jpg)

That will come later this year. See <https://github.com/BadissDjafar/DE10_Standard> for future developments.

But before we get there, we need to kick off a series of demo projects that would be used as building blocks and help us getting a good grasp at the kit.

Therefore, let us start with simple stuff ;) - first things first ...

# GETTING STARTED

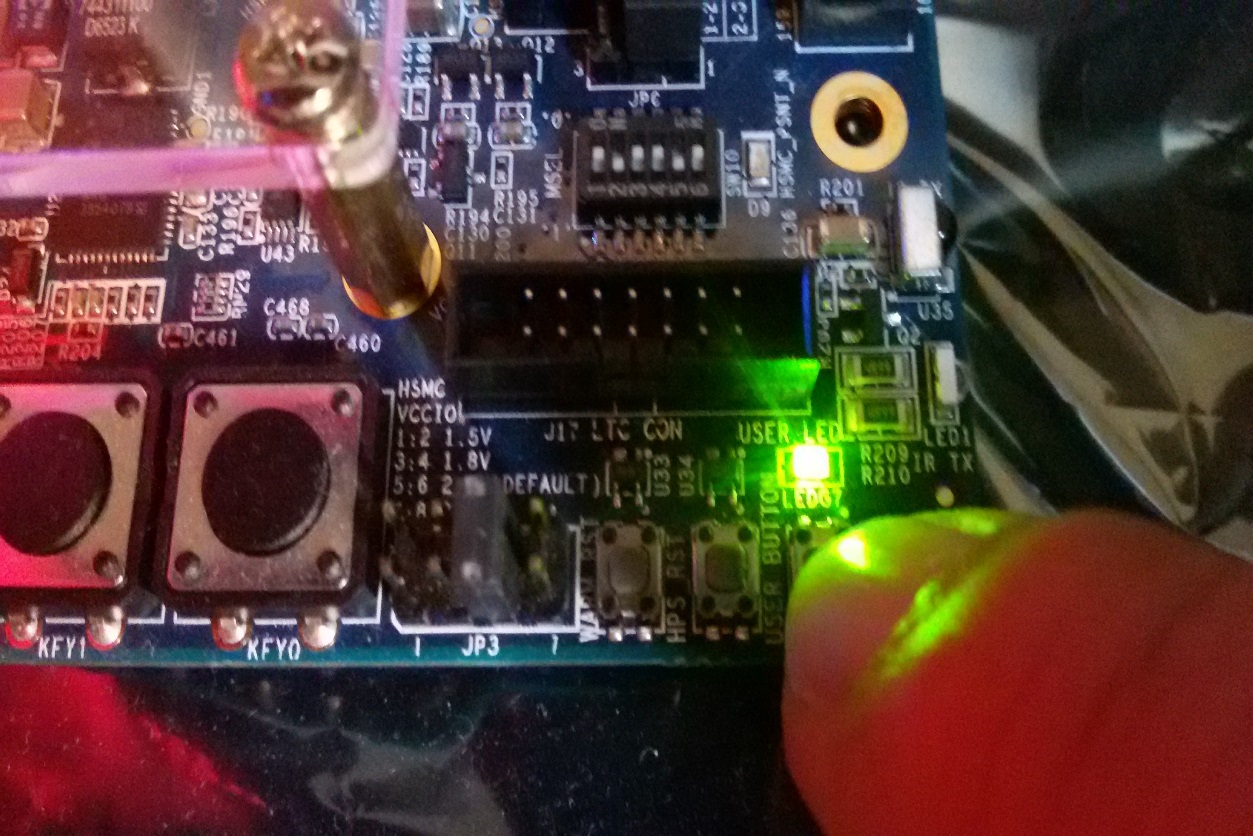
### Blinking LEDs ...

We will focus primarely on the HPS (hard processing system). That is namely the two ARM cores present on the Cyclone V SoC. For each demo, you will hopefully learn something from the kit and can use it as a reference in your own design.

Having the kit up and running (see prerequisite), it was time for me to get on coding. Fortunatly, I did not start from scratch as some demonstration projects were available. In general, the first thing I like to see when I start a project is blinking LEDs - literally, that is the equivalent of checking the heartbeat and temperature of a patient for a doctor. Therefore I decided to start with a small blinking LED demo project.

The LED demo project (<https://github.com/BadissDjafar/DE10_DEMO/tree/master/DEMO_GPIO>) is essentialy a backport from one of the demonstration project that is provided with the kit. It has just been tidied up and the demo "shutdown" has been improved. The task consist in openning the shared memory device file and map the GPIO physical address to a virtual address usable in Linux user space. Then, thank to a signal controlled endless loop, one of the HPS LED is lit periodically in a heartbeat fashion. To be honest, thank to the demonstration CDRom and the documentation, that took about 30mins - For some reason, blinking LEDs always make embedded software hobbyist happy. But in some cases that can be non-trivial, fortunately, that was an easy one.

### LED demonstation :

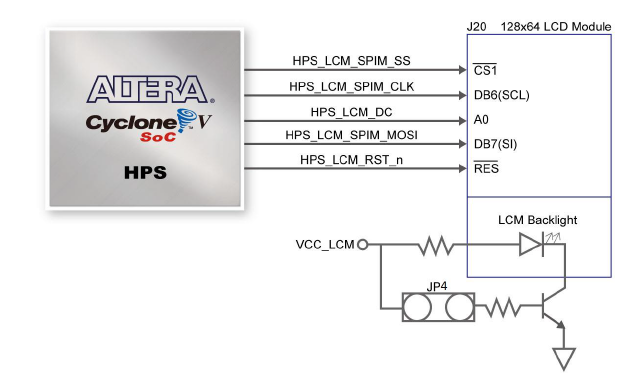
[](https://github.com/BadissDjafar/DE10_REVIEW/blob/master/SCREENSHOTS/LED.jpg)

Having done that, let us explore more extensively the exciting features of the DE10 Standard kit.

As I am planning on displaying network infos on the LCD screen in my bitcoin project. I was eager to know how it operates. Actually, the LCD screen on the kit sits on the SPI bus of the HPS GPIO bank.

Being already familiar with HPS GPIO usage, we know how to map the GPIOs bank to a usable Linux virtual address. But going beyond that, this little demo shows how the control the LCD screen.

### SPI LCD connections :

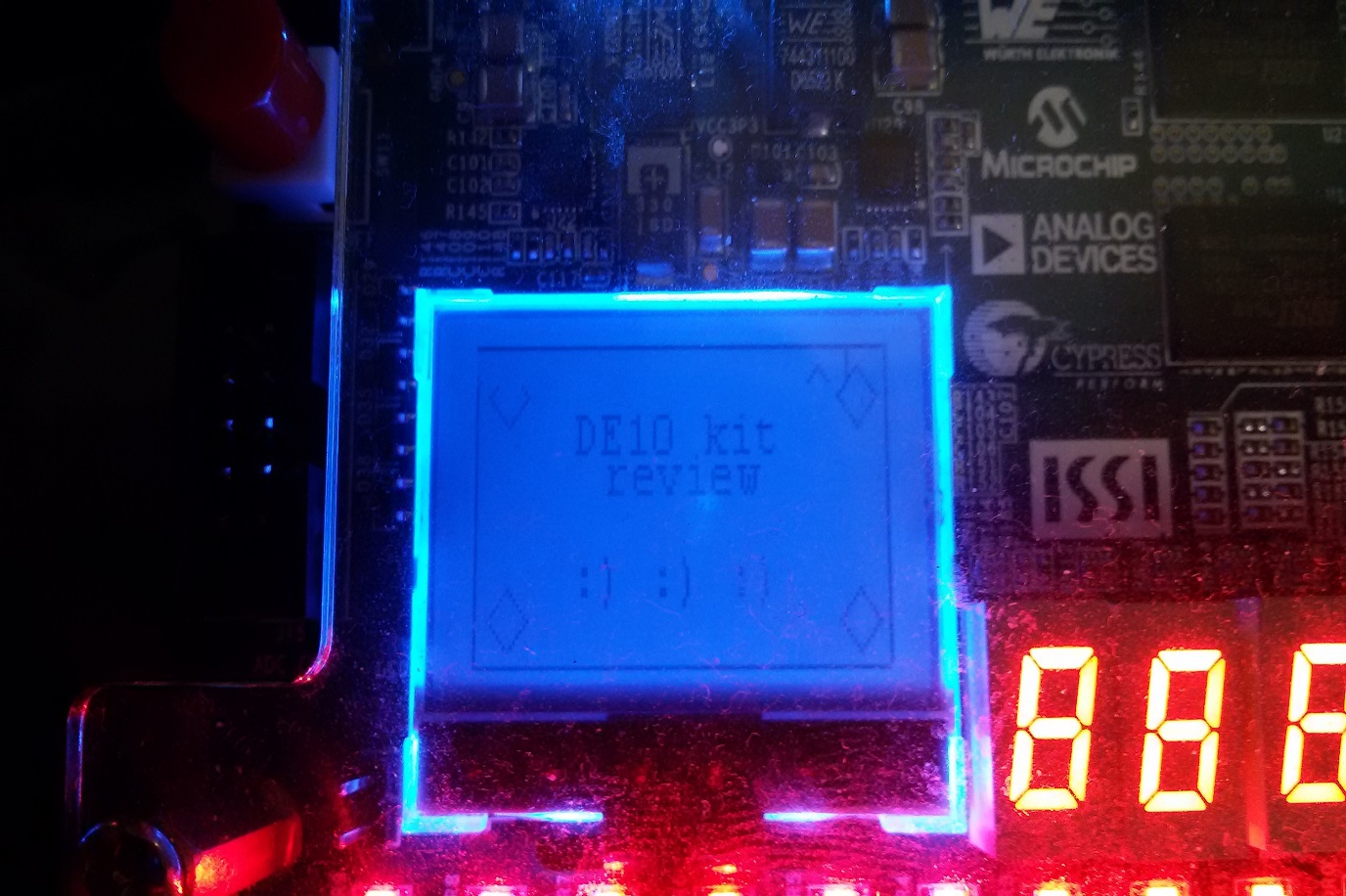
[](https://github.com/BadissDjafar/DE10_REVIEW/blob/master/SCREENSHOTS/LCD_CONNECTIONS.png)

The software design of the LCD Demo requires a bit more skills than the LED demo (see <https://github.com/BadissDjafar/DE10_DEMO/tree/master/DEMO_LCD>). As a result, it worth spending a moment to think of what we are dealing with and ask ourself the right questions :

What is an LCD screen ? What do I want to do with it ? How to deal with it ? Do I want to deal with SPI messages all the time ? Wouldn't it be nice to be able to use letters, draw shapes and all the rest ? Ultimately, how can I make LCD screen usage as simple as possible and make my life easier ? ...

For all these reasons, I have choosen to decouple the part that is driving the SPI bus and the screen management so to speak. the LCD management is split in a driver and a manager. The role of the driver is to deal with the internal mechanic of pixels , SPI meassages and addresses. The manager shall focus on making LCD screen usage as painless as possible. It consist essentialy in a graphic library and should be considered as such. If you wish to get details on how it is implemented, please do feel free to check the code. It is likely that this LCD graphic library will change in the future, but for now, that's it !

### LCD screen demonstration :

[](https://github.com/BadissDjafar/DE10_REVIEW/blob/master/SCREENSHOTS/LCD0.jpg)

Thank to the demo projects implemented by Terasic present in the documentation, yet again, I did not start from scratch. I could just compile the example and test it right away. I just had to focus on the design and that was really a big help. If I had to put myself in the shoes of somebody that is new to all this, that would definitely be an excellent start.

# HPS Application

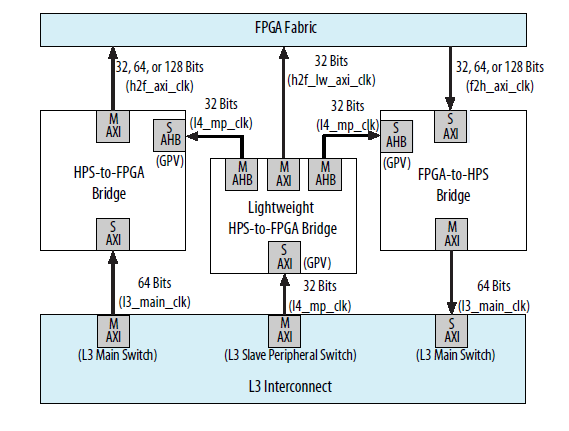
### ... and beyond

As we went through the basic examples, we saw the fundamentals and what can be achieved in a very short amount of time with the DE10 Standard kit. It is actually fairly straightforward to do rapid prototyping with it as far as the hard cores are concerned.

With that done, it is time now to get familiar with the SoC in its essence. That is, not focusing only on the ARM cores but also on the FPGA fabric and how both components can be used together.

This is not a very straightforward thing to do to be truely honest, espcially if you are new to all these things. Therefore, I suggest that if you don't have experience with SoCs, try an FPGA only demo project first (see previous projects https://github.com/BadissDjafar/IPs for example). Essentially, the communication between HPS and the FPGA fabric can be broken down in simple base principles.

### HPS/FPGA communication channels :

[](https://github.com/BadissDjafar/DE10_REVIEW/blob/master/SCREENSHOTS/AXI_BRIDGE.png)

But before we come to that, let's use a concrete case so we can establish our knowledge on a well defined example. As you already know, the HPS interfaces with features like LEDs, push buttons, LCD screen, DRAM and so one. But not everything is "visible" to the HPS module, some of the features are only "visible" by the FPGA fabric in the sense that the wires are connected to it. In the DE10 Standard case, the 7 segment displays are connected to the FPGA IO port only and not to HPS.

The problem is, so far we have run application code on an operating sytem running on one of the HPS cores.

But what if we'd like to display a message as simple as "0xDEADBEEF" for example ? How would we access the 7 segment displays then ?

As a system, the main difference sits in the communications channels used between the HPS and the FPGA and that has to be understood. In order to improve our understanding, we will have the speak about AXI bridges and communication channels.

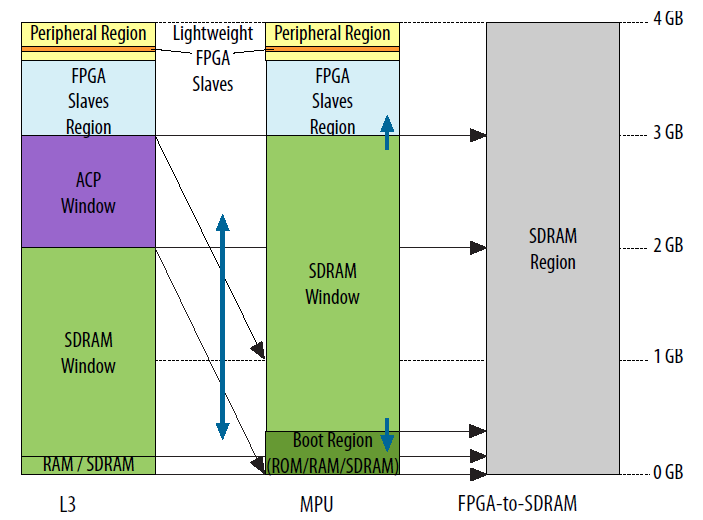
As shown previously, there are three communication channels between the HPS and the FPGA fabric. 64 bits HPS-to-FPGA, 64 bits FPGA-to-HPS and so called "lightweight" HPS-to-FPGA. The HPS-FPGA bridges allow masters in the FPGA fabric to communicate with slaves in the HPS logic and vice versa. For example, if you implement memories or peripherals in the FPGA fabric, HPS components such as the MPU can access them. Components implemented in the FPGA fabric, such as the Nios II soft CPU for example, can also access memories and peripherals in the HPS logic.

# FPGA & HPS

### the devil is in the details ...

As often in embedded systems, things are viewed in term of addresses since that is all the software can see.

### Communication channels and address spaces :

[](https://github.com/BadissDjafar/DE10_REVIEW/blob/master/SCREENSHOTS/ADDRESS_SPACES.png)

Inter-module communication within the Cyclon V SoC is no picnic - A dedicated module from ARM called "System Interconnect" has been implemented in order to deal with that. That module is composed of several submodules that handle communications with the peripherals, memory, buses and the FPGA fabric through specific interfaces (AXI,AHB,APB and so on). For more details, see the documentation from Altera "Cyclone V Hard Processor System Technical Reference Manual" (cv\_5v4.pdf) chap. 7.

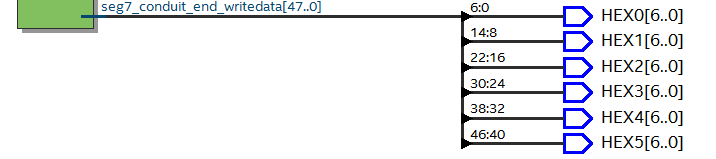
We will focus, in our example, on the "lightweight HPS-to-FPGA" channel. For the time being, let us ignore the different address spaces shown in the figure above and concentrate on L3. In it, you can observe that the memory region that we are interested in is intertwined with the "peripheral" memory region. That basicaly means that we will be able, if the FPGA fabric is configured to do so, to access the feature that we are interested in (namely the 7 segment displays) through that region in the exact same way that we access the LCD screen or the GPIO banks for example.

Notice that in my previous statement, I wrote "if the FPGA fabric is configured to do so". That is where the FPGA design part comes into play. The task there consist in implementing a design (in VHDL or Verilog for example) that will describe the behaviour of the FPGA fabric, allow access to the desired features and finally, program the SoC with it.

In order to do that, you'd use typically the Quartus and Qsys tools suite. For more detail on the design, see the following design <https://github.com/BadissDjafar/DE10_DEMO/tree/master/DEMO_FPGA_GHRD>.

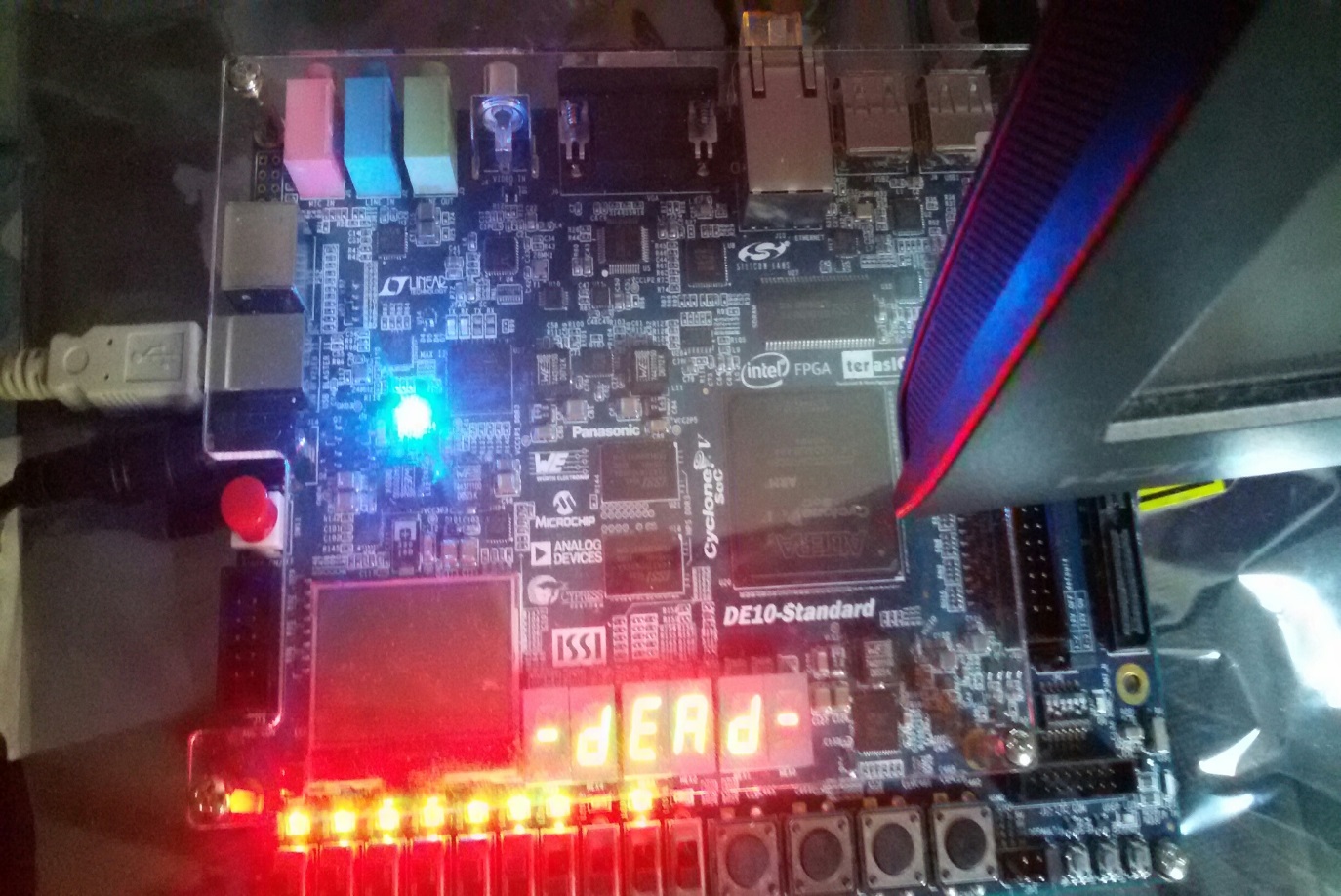
In the case of the DE10 Standard and the 7 segment displays "0xDEADBEEF" demo, here is a view of the netlist generated thank to Qsys and Quartus :

### 7 segment displays netlist :

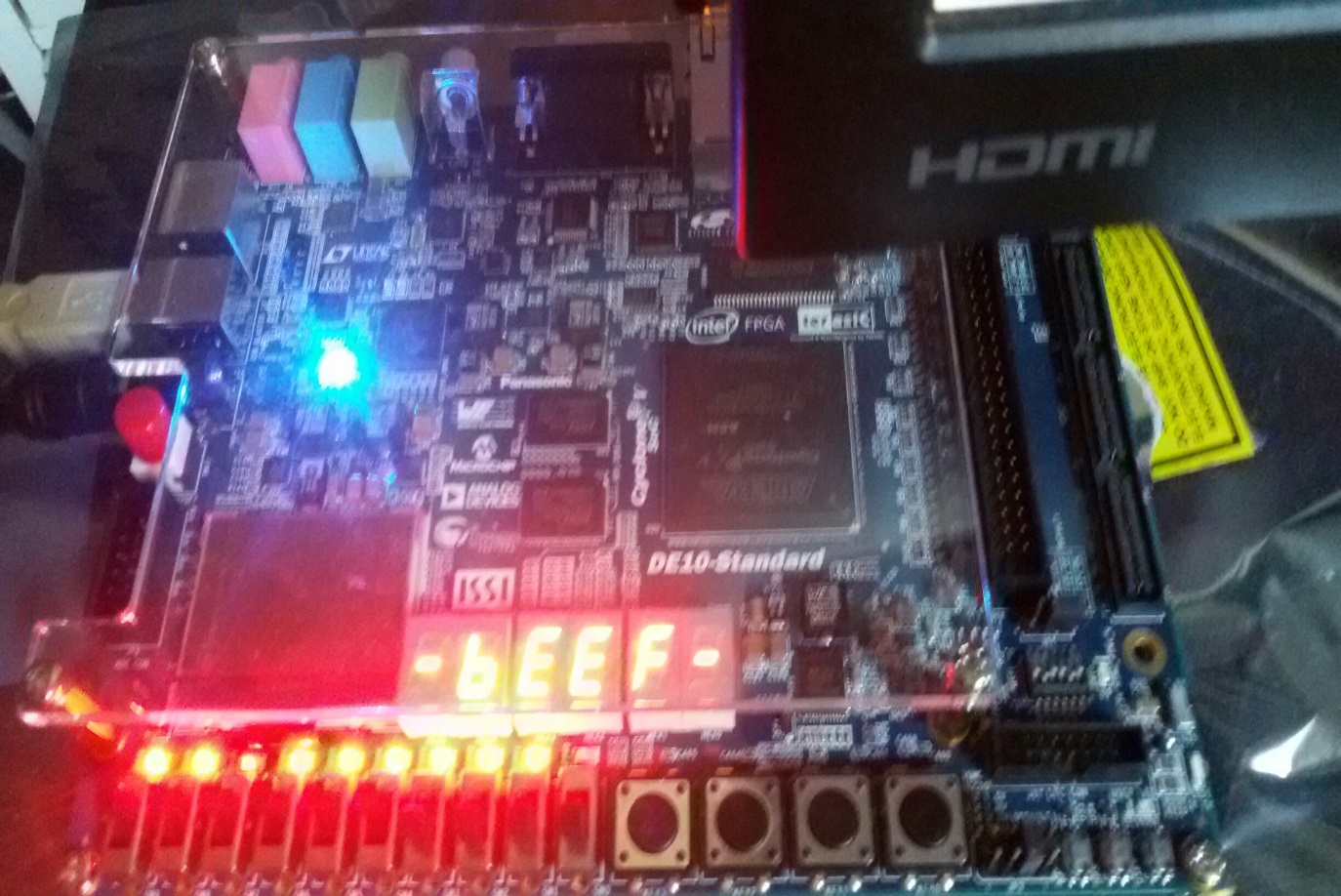
[](https://github.com/BadissDjafar/DE10_REVIEW/blob/master/SCREENSHOTS/7SEG_NETLIST.png)

From the software point of view, since we use the "lightweight HPS-to-FPGA" interface, there are no big differences with the HPS only demos actually. What you'd do is open the shared memory file "/dev/mem", map the physical addresses to virtual addresses and use it in your application. The real difference lays in the way you access the hardware, depending on how the address range that you map is layed out and that is what is important to remember from software perspective. See the code for more details (<https://github.com/BadissDjafar/DE10_DEMO/tree/master/DEMO_HPS_GHRD>)

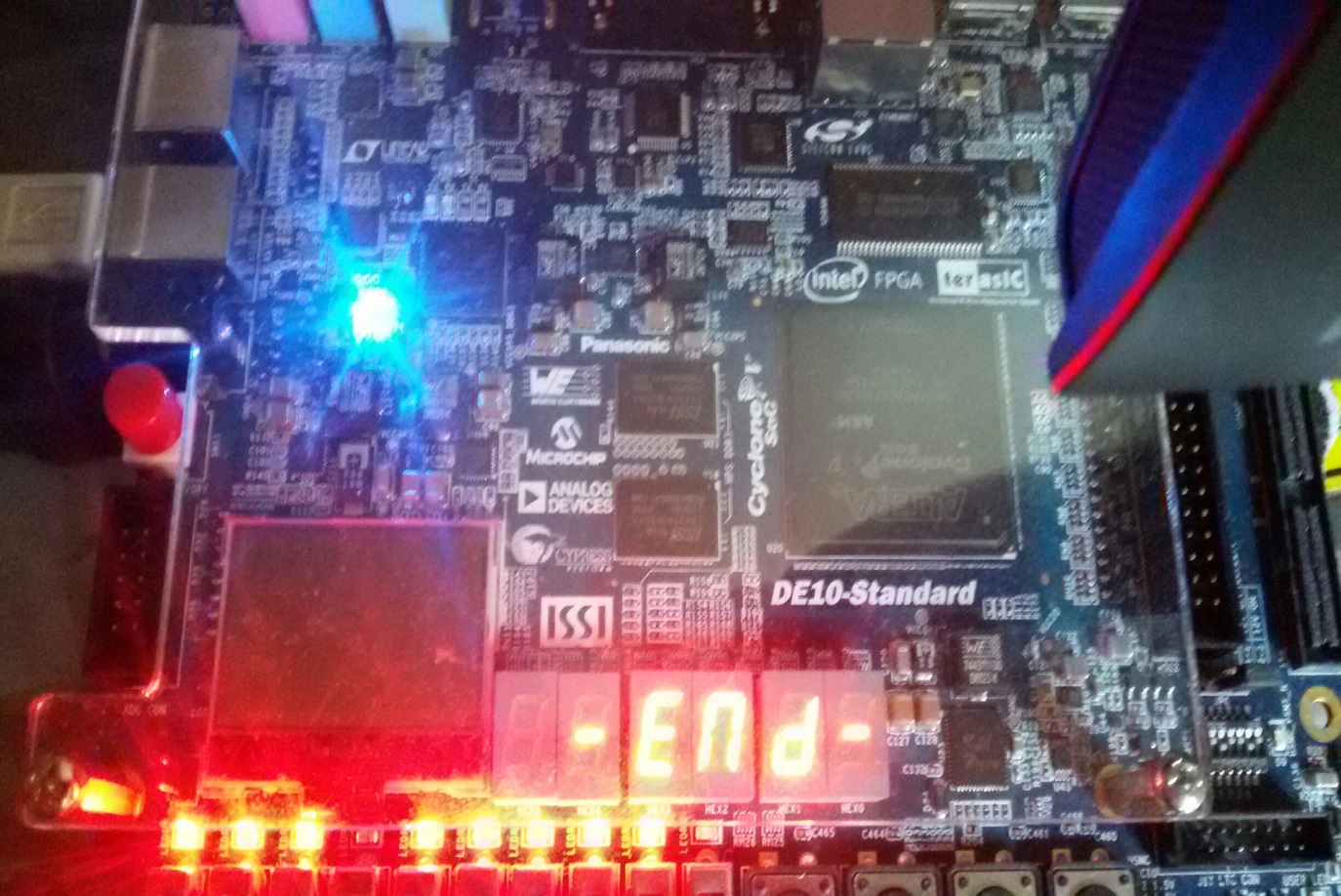
### 7 segment display demo :

[](https://github.com/BadissDjafar/DE10_REVIEW/blob/master/SCREENSHOTS/DEAD_1.jpg)

0xDEAD

[](https://github.com/BadissDjafar/DE10_REVIEW/blob/master/SCREENSHOTS/BEEF.jpg)

0xBEEF

[](https://github.com/BadissDjafar/DE10_REVIEW/blob/master/SCREENSHOTS/END.jpg)

-END-

This is only a quick introduction of the demo and a complete explanation would require more. But that is enough to give a good grasp at what is possible to do with the kit with little knowledge and a lot of interest in the technology.

# CONCLUSION

### "That's all folks !"

It is fair to say that the DE10 Standard kit is excellent value for money. It is probably one of the best platform out there to do rapid prototyping if you wish to predevelop a product. If you are more interested in just learning how to use SoCs or if you wish to use it for a university project, that is the best kit you can find, given the amount of documentation and the level of configurability. I had a lot of fun working on these demos and everything went smoothly. It was also a great oppurtunity to learn a lot and develop building bricks for people who might be interested but do not know where to start.

As a result, I strongly recommend the DE10 standard kit whether you are a hobbyist, a student. a teacher or an R&D engineer.