Hardware Software Codesign Using an Open-Source Authentication Algorithm

Sadoon Al-Bader, Haseeb Ur Rahman, Dr. Ashraf Zaher

Department of Electrical & Computer Engineering, American University of Kuwait
Salem Al-Mubarak Street, Salmiya, Kuwait

s00034652@auk.edu.kw

hjabbar@auk.edu.kw

azaher@auk.edu.kw

Abstract— **This report is the summary of an independent study course taken by the author in Hardware Software Codesign. It outlines all the accomplishments, as well as the challenges faced. The author recommends that any engineering student that is interested in this topic read this report to know what to expect from this field of engineering. The algorithm used for this study was the poly1305 message authentication code.**

Keywords— **Hardware Software Codesign, FPGA, Parallel Computing, Optimization, poly1305**

I. Introduction

Software design gives the programmer the flexibility and ease of modern software tools to create any project and implement any algorithm they desire. Hardware design gives the engineer the speed and parallelism needed in many algorithmic applications that software programmers can only dream of. Although both are perfectly viable fields of study that millions of students seek worldwide, it is less common to study and apply both in tandem. This has changed recently with big companies like Intel® (who recently acquired Altera®) and Xilinx® pushing for hardware software codesign with their latest FPGA offerings also including a full microprocessor that can work with the FPGA in any way the engineer sees fit. The purpose of this independent study and research is to learn the tools and methodologies for hardware software codesign and to try to implement an open source algorithm with them.

II. Background Information

FPGA boards – especially the student-oriented ones – usually come with the logic elements (LEs) that can be programmed using a Hardware Description Language (HDL) like VHDL or Verilog. The student designs his or her hardware in an IDE, programs the board, and finally he or she tests the design. The seemingly obvious drawback of such design method is that it is very rudimentary; the student is designing the hardware from scratch, and testing is very inconvenient and time consuming. Furthermore, even if a student or engineer designs a useful piece of hardware, there is no way to integrate it as part of a system, which is the typical way of designing hardware in this day and age. This is where hardware software codesign comes to play.

Intel® Cyclone™ IV chips were the first of the series to include a microprocessor. They came with very small Intel® Atom™ processors along with the FPGA component. This allowed running full operating systems on the boards. However, the processors were x86 based and very low power x86 chips such as the Atom™ tend to not fare very well in performance, and especially not in performance per watt. For the next generation, the Cyclone™ V, Intel® included an ARM® microprocessor with the FPGA. This has many benefits. One is that ARM® is the same ISA (Instruction Set Architecture) that modern smartphones use. In addition, ARM® processors use a RISC (Reduced Instruction Set Computer) architecture, which enables higher performance in the low-power applications of these single-board development kits, as well as the aforementioned smartphones.

A typical use of the advantages mentioned above is that a hardware designer can create a useful hardware block that is especially crucial for smartphones, such as hardware-accelerated authentication and encryption. The designer can then write a Linux™ kernel module (similar to a device driver in Windows™ vocabulary) that interfaces the FPGA with the SOC (System on a chip), and use the FPGA hardware block as if it were part of the system itself. If everything goes well, the designer can then run full Android™ on the board with a custom kernel that supports their design (remember that Android™ uses Linux as its kernel) and test the design before implementing it on a real phone. This can all be done without buying very expensive phone developer kits, signing contracts and paying hefty licensing premiums.

For these reasons, an Intel® Cyclone™ V board was chosen for this project. The specific board model was Terasic® DE10-Nano. It is a small board targeted at robotics projects. There are bigger boards with more features but this board was chosen to keep costs low, to make for a realistic comparison to phones and SBCs (Single Board Computers), and because the chip was more important than the I/O on the board.

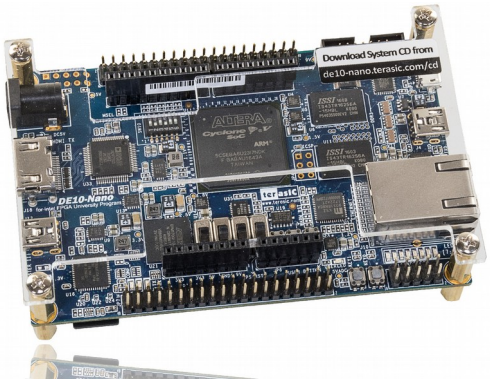

Fig. 1 The DE10-Nano Board

For the algorithm, poly1305 was chosen because it is a strong and versatile authentication code that is typically written in software. The goal was initially to design a hardware version of poly1305 and compare how many bytes per second can be authenticated in software versus the hardware implementation. Thanks to the recent surge of open-source projects though, it didn't take long to find a project on GitHub™ that did exactly that. Unfortunately for us living in the MENA region (Xilinx® does not allow their software to be used in certain countries), the project was designed with a Xilinx® board in mind, though the findings are useful regardless.

Upon finding that the algorithm was already implemented in hardware as an open-source Master's degree project on GitHub™, the aim of this independent study shifted towards learning the advanced tools and methods used for embedded system design and hardware software codesign. The project was split into three parts:

1) Optimizing the algorithm in software for speed and efficiency.

2) Translating the algorithm to a modern multi-paradigm programming language.

3) Preparing the DE10-Nano board for hardware software codesign.

III. Software Optimization

The implementation was chosen based on this criteria:

A. Must be written in a language that is fast and can be compiled to machine code. Java and Python are not applicable here because the former requires a virtual machine and the latter is an interpreted programming language, while both also introduce large performance and memory overheads compared to C

B. Must be free and open-source so that the code can be modified, tested, and analyzed. (Note that "free" here corresponds to freedom, not the price tag of the code. Meaning developers are free to modify and republish it.)

C. Must be concise, clearly documented and/or easily readable.

D. Must have a built-in test code and optionally a way to measure performance. In this case performance is measured in bytes processed per second, which will be informally referred to as "speed" in this paper.

These four criteria make working with open-source software easier and more welcoming to someone who it trying to learn and/or improve software. With these criteria in mind, Monocypher was chosen because it satisfies all the four.

At the beginning, Monocypher was stripped down to only the code needed to run poly1305, and a small test program was written for it. This was done to quickly test if any changes broke the algorithm to a point where it does not work anymore. Once the changes have been tested with the small test program, the code would be copied and pasted to the original code from the GitHub™ repository and tested using the built-in test program. This is done by typing in "`make test`" in the command line. To test the speed after the changes, one can type "`make speed`", showing a formatted standard output of all the speeds for each algorithm.

One change was made that increased the speed of poly1305 by up to 10%[1] on most systems that have been tested. Figures 2 and 3 show a comparison of the original code and the modified code.

```
21 static uint32_t load32_le(const uint8_t s[4])
22 {
23     return (uint32_t)s[0]
24         | ((uint32_t)s[1] <<  8)
25         | ((uint32_t)s[2] << 16)
26         | ((uint32_t)s[3] << 24);
27 }
28
29 void crypto_poly1305_init(crypto_poly1305_ctx *ctx, const uint8_t key[32])
30 {
31     for (int i = 0; i < 5; i++) {
32         ctx->h[i] = 0;
33     }
34
35     ctx->c[4] = 1;
36     poly_clear_c(ctx);
37
38     for (int i = 0; i < 1; i++) { ctx->r  [0] = load32_le(key           ) & 0x0fffffff; }
39     for (int i = 0; i < 4; i++) { ctx->r  [i] = load32_le(key + i*4     ) & 0x0ffffffc; }
40     for (int i = 0; i < 4; i++) { ctx->pad[i] = load32_le(key + i*4 + 16);             }
41 }
```

Fig. 2 Original code

```
21 void crypto_poly1305_init(crypto_poly1305_ctx *ctx, const uint8_t key[32])
22 {
23   for (int i = 0; i < 5; i++) {
24     ctx->h[i] = 0;
25   }
26   ctx->c[4] = 1;
27   poly_clear_c(ctx);
28
29   ctx->r[0] = *(uint32_t*)(&key[0]) & 0x0fffffff;
30   ctx->r[1] = *(uint32_t*)(&key[4]) & 0x0ffffffc;
31   ctx->r[2] = *(uint32_t*)(&key[8]) & 0x0ffffffc;
32   ctx->r[3] = *(uint32_t*)(&key[12]) & 0x0ffffffc;
33   ctx->pad[0] = *(uint32_t*)(&key[16]);
34   ctx->pad[1] = *(uint32_t*)(&key[20]);
35   ctx->pad[2] = *(uint32_t*)(&key[24]);
36   ctx->pad[3] = *(uint32_t*)(&key[28]);
37 }
```

Fig. 3 Modified code

The for loops copying the data are unnecessarily slow in the original code. Line 38 did not even need to be a loop. The function `load32_le()` takes 4 bytes and combines them into one 4-byte element (unsigned 32-bit integer). It does so by shifting, which is very inefficient especially considering that C stores array elements next to each other in memory. Figure 4 shows how they are stored on a little endian machine (x86, ARM, modern POWER, RISC-V, and most modern architectures.)

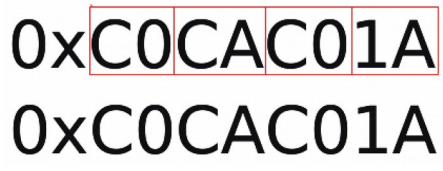

Fig. 4 4-byte array (top) versus 32-bit integer (bottom)

So instead of taking each byte on its own, one could simply treat the four-byte "slice" of an array as a full 32-bit integer, resulting in the exact same data. The counterargument was that this would not work on big endian systems where data is stored in the opposite way, so the array would be stored like `0x1AC0CAC0`.[1] However, by simply avoiding the shifting, the code can be 10% faster, suggesting that there is a bottleneck in the copying process and that it is slowing down the overall authentication. The resulting assembly code also shows the vast reduction in code size and shows that the new code has no jumps or branches, typically meaning it is faster.[2]

An alternative approach would be to use `memcpy()`, a standard C function that copies any type of data of any size from the source to the destination, specified by its arguments. This approach, while in theory should be faster due to its

optimization on such a low level, somehow caused the code to become slower on all tested machines. Further investigation is needed and would have been done if not for the shortage of time.

```
22  void crypto_poly1305_init(crypto_poly1305_ctx *ctx, const uint8_t key[32])
23  {
24    for (int i = 0; i < 5; i++) {
25      ctx->h[i] = 0;
26    }
27    ctx->c[4] = 1;
28    poly_clear_c(ctx);
29
30    memcpy(&ctx->r, &key[0], 16);
31    memcpy(&ctx->pad, &key[16], 16);
32    ctx->r[0] &= 0x0FFFFFFF;
33    ctx->r[1] &= 0x0FFFFFFC;
34    ctx->r[2] &= 0x0FFFFFFC;
35    ctx->r[3] &= 0x0FFFFFFC;
36  }
```

Fig. 5 Alternative copy method

The same logic of copying as shown in figure 4 was applied to other code in the program, and can be seen in other methods like `crypto_poly1305_update()`.

IV. Translation

The process of translating code from one language to another has three main benefits. The first is that it forces the programmer to learn the new language and sometimes learn a bit of the original language. The second is that the programmer can add his or her own touches to the code where he or she sees fit. For example, one can simplify some steps in a way that could not have been done in the original language. One can also write a built-in unit-test as will be shown shortly. The third benefit is that the programmer can test the efficiency of both languages, and while it is not a scientific approach especially considering the different features and paradigms of programming languages, it does give an idea, especially when analyzing both codes in assembly like was done in section III of this report.

"Translate" here, is not a formal word that is used to describe rewriting the code in a different language. But colloquially if one can call a programming language a "language", then it is arguable that rewriting the same logic in a different language can be informally called "translation", and unsurprisingly can come with its own version of "meanings lost in translation".

The chosen language was D. D is a very special language. It is multi-paradigm; it does not force a programmer to use any style of programming but gives the option of five different paradigms.[3] D is also fully compatible with C code; C code can be written inside a D program by denoting that it is C code by typing `extern(C){}`.[4] D is a compiled language with three officially supported compilers: Digital Mars D Compiler (DMD), GNU Compiler Collection (GDC, lately merged to official GCC), and an LLVM based compiler (LDC).[5] This report will only highlight the main features used in the D translation so as not to be repetitive; the code looks very similar to C, with many more quality-of-life enhancements. There is no benefit in speed in switching from C to any other programming language these days, quite the opposite in fact, so the choice of a language was simply personal as learning a new language was part of the goal of this study.

The first feature of D used was an optional feature (in fact most D features are optional to keep the programmer in control) called `dub`. `dub` is a command line package manager for the D language. One can create projects using it, and add dependencies to other libraries that will be automatically downloaded during compilation.[6] It is a very versatile tool that takes away the need for a separate build tool like `make` or `cmake`.

The second feature was technically two features: Unit-testing and conditional compilation. Unit-testing is built into the language, by just typing `unittest{}` and typing the unit-test code within the brackets, `dub` will automatically compile and run the unit test when `dub test` is sent to the command line, or through a supported IDE.

```
225  unittest{
226
227    write("###WARNING###: Using unittest config!\n\n");
228
229    const ubyte[32] key = [
230      0x85,0xd6,0xbe,0x78,0x57,0x55,0x6d,0x33,
Run:  unittest
c: 0x00000000_00000000_00000000_00000000_01000000
r: 0x85D6BE08_54556D03_7C44520E_40D50608
pad: 0x0103808A_FB0DB2FD_4ABFF6AF_4149F51B
c_idx: 0
mac:     : 0xA8061DC1_305136C6_C22B8BAF_0C0127A9
realmac: : 0xA8061DC1_305136C6_C22B8BAF_0C0127A9
ALL TESTS PASSED!!!
Done!
```

Fig. 6 Snippet of unittest code and output

The second feature, conditional compilation, is similar to the C and C++ preprocessor `#if, #elif`, etc. in a much simpler fashion[7][8]. One can specify inside the program whether portions of code should be compiled or not, depending on certain conditions. In this case, a `static if` can be used but the `version(){}` syntax was chosen for the simpler style. `version(unittest){}` compiles the code between the curly braces only if the the code is being tested. This is of course used outside the `unittest{}` section, in the main code. The main use of this is to print debug messages, in this project at least. It can however be used to control which code gets compiled depending on the operating system or architecture on the machine, with `version(Win64){}` or `version(X86_64){}` and can even compile based on the compiler used like in `version(DMD){}`.

```
114  void poly1305Init(Poly1305Data *ctx, const ubyte[] key)
115  {
116    ctx.c[4] = 1;
117
118    memcpy(&ctx.r, &key[0], 16);
119    memcpy(&ctx.pad, &key[16], 16);
120    ctx.r[0] &= 0x0FFFFFFF;
121    ctx.r[1] &= 0x0FFFFFFC;
122    ctx.r[2] &= 0x0FFFFFFC;
123    ctx.r[3] &= 0x0FFFFFFC;
124
125    //only print test values in unittest environment.
126    version(unittest) printCtx(ctx);
127  }
```

Fig. 7 The same function in figure 5 written in D

Figure 7 shows multiple features used. Firstly, D automatically assigns integers to 0, just like Java™. Secondly, the `memcpy` command used here is actually from the standard C library. This is because any standard C function can be used in D, not just user-written C code. Finally, it shows the `version(){}` syntax as discussed earlier.

V. Hardware Preparation

Hardware design is the proverbial wild west of Computer Engineering. While there are many resources that teach students how to design hardware in an HDL, one would be hard-pressed to find good documentation on how to practically use an FPGA board. All the Intel® documentation assumes previous knowledge and experience in hardware design, which makes it not helpful in this case. The scarcity of resources on how to configure FPGA boards is further proven when the main page – written by a hobbyist – that was used as a reference for using the DE10-Nano was taken down for no known reason.[9] This is where the American University of Kuwait's engineering labs provided the much-needed head start into working with FPGA boards. In Embedded System Design Lab (CPEG 340L), students learn how to configure the FPGA portion of a Cyclone™ III or IV board. The main focus of the course was to interface with I/O using the hardware pins in Quartus™. It would have taken months to get started with FPGA design if not for that laboratory portion of Embedded System Design.

The basic principle is simple: The FPGA+SOC board comes with a storage device, typically an SD card, with an operating system loaded on it. One can use whatever operating system provided – which is generally not recommended because they are usually very outdated and have security vulnerabilities that were never fixed – or use a Linux based operating system downloaded from the internet. There is no specific version of Linux that is up-to-date and made for the DE10-Nano, so the choice was once again personal and Arch Linux was chosen because of familiarity and simplicity. The SOC in the Cyclone™ V runs on the same ARMv7 architecture that is used in the Raspberry Pi™ 2. So the first step was to get an Arch Linux image for the Raspberry Pi. The only thing that makes an image specific to a board is that some boards require closed-source drivers to function properly, like the Raspberry Pi™ 2's video adapter. This means that they have specific Linux kernel packages for the board. Armed with this information, the provided Linux kernel was not used, as we needed our own kernel with changes from Altera® and Intel® to function properly.

The modified kernel source code was downloaded from GitHub™[10] and cross-compiled for the ARMv7 architecture. Cross-compilation is the process of compiling code on one architecture to produce binary code for another architecture. In this case, the full Linux kernel was compiled on x86_64 for the ARMv7 architecture, but many issues plagued the process. Video output through HDMI, networking, and USB never all worked together. One configuration had HDMI and networking, but not USB, another had USB and networking only, etc. The first expected culprit was the Linux kernel config file. This file decides what parts of the kernel are compiled and how drivers are set-up, among other things. A very useful feature of the Linux kernel is that the compiled kernel image can reproduce the config file and theoretically with the correct config file and source code, the same kernel image can be reproduced. Altera® ships the DE10-Nano with an outdated version of Angstrom Linux, but by acquiring the config file one can compile newer kernels with the same features enabled, instead of relying on guesswork. With the config, the kernel asks for many new features to be configured before it is compiled, and for the sake of potential compatibility, most were enabled, especially the ones that Altera® included in their custom source code. Linux 5.1 was compiled and ran perfectly on the DE10-Nano, with HDMI working as well.

By following the aforementioned and now-deleted guide, a simple hardware device was programmed for the board: an LED controller. This was done by adding the LED logic to the reference Altera® GHRD (Golden Hardware Reference Design). The GHRD is a full skeleton project that configures everything on the board to its defaults. The next step was to write the kernel module (driver) to run the LEDs from within Linux using C. Thankfully, the guide had the full source code for the userspace and kernel driver, but the userspace driver was used for simplicity. The LEDs worked, with a caveat: HDMI was not working. This was with the correctly configured kernel – everything would work perfectly until a custom hardware program (like the LED controller) was added. Fixing this bug was the most time consuming step in this study, as it took about a month. None of the Intel® or Altera® documentation mentioned anything about the HDMI port, except a small clue on the board diagram that led to the reason of this odd behavior. The HDMI controller, as shown in figure 8, is the only controller connected to the FPGA instead of the HPS (Hard Processor System, Altera®'s marketing term for the ARM SOC on the board). This means that in order to use the HDMI port, both the previously configured kernel modules and an FPGA configuration file is needed. But as usual, there is no documentation on this subject and the binary file provided on the SD card – which does support HDMI – cannot be decompiled. With not many options left, the next step was to try different example configurations that were provided on the board's CD to see if any have HDMI configured. After many hours of work, it was found that the sample named "DE10_NANO_SoC_FB" was the one, it seems that FB corresponds to "Framebuffer" here. With the correct hardware configuration and the correct kernel, everything worked flawlessly, and a very specific process to program the board was created despite the scarcity of documentation.

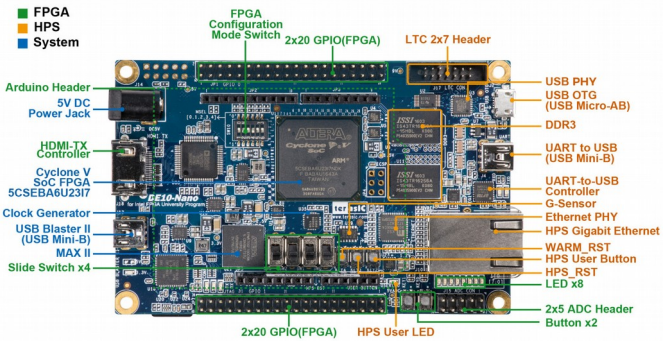

Fig. 8 The board layout from Altera®'s website

The initial plan was to convert Furkan Turan[11]'s implementation of poly1305 in VHDL from a Xilinx® board to this Altera® board, but with there being scarce documentation as stated before, and with Xilinx® not allowing use of their software in some countries, in addition to all the time wasted on configuring the board without the proper documentation, there was no time left to complete the main project.

VI. Other Milestones

As part of this major independent study course, some new skills were learned either due to the need, or because of the quality-of-life features that these skills provide.

The first was learning Verilog and SystemVerilog. This was because the syntax and features of SystemVerilog were much more advanced than that of VHDL or Verilog. However, SystemVerilog was not used in the project after finding Furkan's code and starting to focus on configuring the FPGA, and going with the more familiar VHDL.

The second skill was learning `git`. It is the most well-known version control software, and popular websites like GitHub™ and GitLab™ use it. The latter website was the one used to host this project because GitLab has a more user-friendly interface and is open source – it can even be deployed on a local server like a university or work server, which was useful during this project.

Cross-compilation was mentioned before in the configuration section, but it is a topic that is significant enough to warrant a full report on its own. Cross-compilation on its own can be very simple, but the process of configuring a compiler or collection of compilers like the GNU Compiler Collection can be very time-consuming. Translating code to a machine code of a different architecture is quite simple, where things become complicated is when one compiles software that uses libraries other than standard system libraries that are included with most compilers. These libraries that reside in system folders are specifically compiled for the ISA being used, and cannot be used for cross-compilation. One would need the full system image of the other system anytime he or she decided to compile software for that device's ISA.

Another option that was not heavily studied here is `distcc`, it is a distributed compile application that allows many client machines to compile one application, while linking only happens on the server machine. Meaning no need to copy the files over to another machine and waste time with `chroot` (more on that later).

In the same veins of cross-compilation comes emulation. Emulation is running a program that is written for another architecture on a computer. QEMU is a free and open-source project dedicated to doing that and more, in fact, QEMU is now used for Amazon® AWS™ instances because of its power and efficiency.[12] QEMU has a specific feature called "static emulation". By downloading the static binaries of QEMU for a given architecture, one can use these binaries to run any program written for any architecture on any other architecture, with a few exceptions. This was useful to test some simpler code and to be able to configure the board while in university where it is inconvenient to use by just `chroot`ing into the SD card. `chroot` is a Linux program that enables access to another Linux system as if it was the main system, and by using both `chroot` and QEMU, one can use the DE10-Nano's system on any computer and configure things easily.

VII. Conclusion

Hardware and software, when developed and designed together, make for a much more complete product that can outmatch other designs that focus only on one field. Likewise, engineers who are experienced in both software and hardware design are more likely to develop higher performing and more efficient products. There is hope that hardware manufacturers will improve product documentation and make hardware software codesign a more viable field of design. In a world where Moore's law is starting to show its age, efficient use of hardware resources is a must.

References

[1] S. Al-Bader "Made Poly1305 10% faster by Sadoon-AlBader · Pull Request #118 · LoupVaillant/Monocypher", GitHub, 2019. [Online]. Available: https://github.com/LoupVaillant/Monocypher/pull/118 [Accessed: 08- Sep- 2019]

[2] S. Al-Bader, "Compiler Explorer - C (x86-64 gcc 9.2)", Godbolt.org, 2019. [Online]. Available: https://godbolt.org/z/iVUHCW [Accessed: 08- Sep- 2019]

[3] "Overview - D Programming Language", Dlang.org, [Online]. Available: https://dlang.org/overview.html [Accessed: 08- Sep- 2019]

[4] "Interfacing to C - D Programming Language", Dlang.org, [Online]. Available: https://dlang.org/spec/interfaceToC.html [Accessed: 08- Sep- 2019]

[5] "Compilers - D Wiki", Wiki.dlang.org, [Online]. Available: https://wiki.dlang.org/Compilers [Accessed: 08- Sep- 2019]

[6] "Getting Started With DUB - DUB - The D package registry", Dub.pm, [Online]. Available: https://dub.pm/ [Accessed: 08- Sep- 2019]

[7] "The C Preprocessor vs D - D Programming Language", Dlang.org, [Online]. Available: https://dlang.org/articles/pretod.html [Accessed: 08- Sep- 2019]

[8] "Conditional Compilation - D Programming Language", Dlang.org, [Online]. Available: https://dlang.org/spec/version.html [Accessed: 08- Sep- 2019]

[9] O. guztech, "Building embedded Linux for the Terasic DE10-Nano (and other Cyclone V SoC FPGAs)", Bitlog, 2017. [Online]. Available: https://web.archive.org/web/20181117104243/https://bitlog.it/hardwar

e/building-embedded-linux-for-the-terasic-de10-nano-and-other-cyclone-v-soc-fpgas/ [Accessed: 08- Sep- 2019]

[10] Altera, "linux-socfpga", GitHub, 2019. [Online]. Available: https://github.com/altera-opensource/linux-socfpga. [Accessed: 08- Sep- 2019]

[11] F. Turan, "NaCl-Hardware", GitHub, 2016. [Online]. Available: https://github.com/furkanturan/NaCl-Hardware. [Accessed: 08- Sep- 2019]

[12] S. Sharwood, "AWS adopts home-brewed KVM as new hypervisor", Theregister.co.uk, 2017. [Online]. Available: https://www.theregister.co.uk/2017/11/07/aws_writes_new_kvm_based_hypervisor_to_make_its_cloud_go_faster/ [Accessed: 08- Sep- 2019]