DUBLIN CITY UNIVERSITY
SCHOOL OF ELECTRONIC ENGINEERING

ECE Paris
SCHOOL OF GENERAL ENGINEERING

Efficient SoC Implementation of the lightweight cipher Piccolo-80

Project Portfolio

Patrick BORE
ID Number: 17211519
August 2018

MASTER OF ENGINEERING

IN

NANOTECHNOLOGY ENGINEERING

Supervised by Dr. Xiaojun Wang

Acknowledgements

I would like to thank Dr. Xiaojun Wang, professor at Dublin City University, for taking on his time to explain me the goals of the project, for giving me some leads of research and most of all for giving me the opportunity to work on this subject. Thanks also to the Dublin City University and ECE Paris to provide free access to the IEEE database which contains very interesting scientific papers. Finally, I thank my friends for supporting me during my long writing days while they were in holidays.

Declaration

I hereby declare that, except where otherwise indicated, this document is entirely my own work and has not been submitted in whole or in part to any other university.

Signed: Date: 31/08/2018

Table of Contents

Acknowledgements ii

Declaration iii

SoC Implementation, Co-Design, of Piccolo-80 on the Zynq Chip through the AXI Bus 1

I. INTRODUCTION 1

II. THE PICCOLO PAPER 1

A. General remarks 1

B. Personal remarks 1

III. DIFFERENT IMPLEMENTATIONS OF PICCOLO 1

A. Speed Implementation 2

B. Pipeline Implementation 2

C. Resources Implementation 2

IV. Results 2

V. Co-design 2

VI. 3

Conclusion 3

Acknowledgment 3

Notation 3

References 3

Appendix A - Literature Review ..... A

SoC Implementation, Co-Design, of Piccolo-80 on the Zynq Chip through the AXI Bus

Patrick BORE, Master student DCU and ECE Paris

***Abstract*— In this paper I do a small review of the paper describing the Piccolo algorithm. I explain the ins and outs of 3 different implementation of the Piccolo-80 I did and made available. Then I compare them on different characteristics. And to finish I give few tips and trick to do a co-design on a processor and logic SoC chip.**

***Index Terms*— AXI, AXI Bus, Co-Design, Cryptography, Digilent, FPGA, Hardware Function, IP, Piccolo, Software and Hardware design, VHDL, Vivado, Xilinx, Zynq, Zynq Chip.**

I. Introduction

The aim of this project is to take advantage of the Zynq chip (SoC with programable logic) in the design of hardware function for co-design project. In this project we took a lightweight cryptography algorithm as a way to explore to possibilities of co-design. But this project is meant to be built on, probably by future master student or Phd student, that's why I made available everything I did with the needed explanation on my GitHub [34].

II. The Piccolo Paper

The paper describing the piccolo algorithm [1] is really good and did allow me to do my own implementation fairly easily. But there are few point that made me loose time that it could have been improved on. These proposed improvements are only from my personal experience with the paper.

A. General remarks

The 1st point I noticed is when they have a subdivision of a signal they don't precise, in the schematics, where are the most significant bits and less significant bits. Like we can see on the schematic for the S-Box for example. And on top of that they named them differently of what I'm used to. We can take the example of the Round Permutation when x_0 which is a subdivision of the 64 bits X signal is actually the 8 most significant bits and not the 8 less significant ones like it led me to suppose. There is too many times where the order of the bits are not specified on the schematic, they are explain on the written part but a more precise schematic would have be much appreciated.

The 2nd point is more aim for people not familiar with numerical electronics. In the different algorithm threw the paper the use of XOR logic is everywhere but I was surprise to not see it in the notation part as it should.

The 3rd point which might just be a point of view. The designer of the algorithm obviously did an implementation of it, but I couldn't find any record of their hardware implementation. It doesn't mean it doesn't exist, but it means at least that it's not easy to find.

At last but the most important is the size of the test vector … extremely small only 1 example for each version of piccolo. More of them and more random ones would allow a bigger confidence that the implementation you made is what the designer of piccolo had in mind. And even maybe on one of these examples a step by step of the different value threw the algorithm to allow easier debug if the intended result isn't there. You can find more value to test and a detailed step by step of the encryption on my larger report, available on my GitHub.

B. Personal remarks

In the F function we can find a matrix multiplication. 1st of all it took me a decent amount of time to figure out it wasn't normal multiplication but multiplication in a Galois Field. And secondly it does suppose that the reader knows about Galois Field which would only apply to people with a strong maths background or cryptography background. So an more detailed version of that part of the algorithm would definitely have been appreciated. No need for a master class on Galois Field but maybe a simple algorithm to do the needed calculation for the M matrix.

Figure 1. is the solution I design (low weigh and hardware oriented).
My knowledge in Galois Field operation is really limited so I'm pretty sure this could be improved to be even more efficient.

$$x'_{0(5)} = x_{0(4)}|\{0\}_2 \; xor \; (x_{1(4)}|\{0\}_2 \; xor \; \{0\}_2|x_{1(4)}) \; xor \; \{0\}_2|x_{2(4)} \; xor \; \{0\}_2|x_{3(4)}$$
$$x'_{1(5)} = \{0\}_2|x_{0(4)} \; xor \; x_{1(4)}|\{0\}_2 \; xor \; (x_{2(4)}|\{0\}_2 \; xor \; \{0\}_2|x_{2(4)}) \; xor \; \{0\}_2|x_{3(4)}$$
$$x'_{2(5)} = \{0\}_2|x_{0(4)} \; xor \; \{0\}_2|x_{1(4)} \; xor \; x_{2(4)}|\{0\}_2 \; xor \; (x_{3(4)}|\{0\}_2 \; xor \; \{0\}_2|x_{3(4)})$$
$$x'_{3(5)} = (x_{0(4)}|\{0\}_2 \; xor \; \{0\}_2|x_{0(4)}) \; xor \; \{0\}_2|x_{1(4)} \; xor \; \{0\}_2|x_{2(4)} \; xor \; x_{3(4)}|\{0\}_2$$

for $i \leftarrow 0$ *to* 3 *do*
  if $(x'_{i(5)}$ *MSB is* $\{1\}_2)$ *then*
    $x'_{i(4)} = x'_{i(5)} \; xor \; \{10011\}_2$ //and only keep the 4 LSB
  else then
    $x'_{i(4)} = x'_{i(5)}$ // only keep the 4 LSB

Fig. 1. Diffusion Matrix algorithm example

III. Different implementations of Piccolo

In all my implementation I added a bit to know if the implementation is supposed to encrypt the data or decrypt it. This have repercussion for the comparison of my algorithm size to others, but you need to keep in mind that my implementation are working implementation so design choice away from the pure theory implementation had to be made.

They also are available on my GitHub [34] for anybody to use and improve.

A. Speed implementation

The idea behind this implementation was to forget the size problem and just get the maximum speed by just

unfolding everything so the encryption would be done in one go. Because the Piccolo-80 algorithm is based on a 25 rounds system what I basically did is implement 25 versions of one round and just connected all of this together. It results of the algorithm being clock less as there only is combinatory logic. That doesn't mean it find the cipher instantly it just means that it is only limited by the logic and net latency.

B. Pipeline implementation

The idea behind the pipeline is to have a higher data rate. By using the speed implementation and adding register between steps. This allow to have less logic at once (roughly 1/25th) so less latency due to the 1 go idea of the speed implementation. This implies that you will have to wait 25 clock cycles before getting your first cipher out. But this is a small step back because of the memory between the steps you can start encrypting a new block on the 2 clock cycle so you will have to wait for 25 clock cycle for the 1st block but the next one will follow on the next clock cycle basically make the 25 clock latency inexistent as long as you encrypt a large number of packets. The main limitation of my implementation is the need of keeping the same key threw the block, if you want to change the key you will need to wait for all the block to be out of the encryption system so you basically come back to the beginning and will need to wait the 25 clock cycles again for the next block. But when your need to encrypt large size data you usually use the same key for the full document so that is not a problem for me. But it can be fixed by making a new implementation with dynamic key along the round, but it will increase the size needed for the algorithm largely so 50 clock cycles lost when you need to change key seems a good trade off.

C. Resources implementation

The resource implementation is more axed around using less logic. I didn't optimize the logic too much as I still wanted to keep a decent data rate in front of the other implementation. For example, I didn't use the proposed optimization given in the paper [1] because I find it too slow. This version looks more like a for loop that repeat itself 25 times. I made the round keys generated on a for loop to optimize also that part of the algorithm and not have a massive multiplexer with all the round keys waiting to be used.

The main speed restriction here is the need of 25 clock cycle to encrypt one block. And you'll have to wait as long for the 2nd block. As the logic is latency is reduced in front of the speed implementation.

IV. Results

In most papers you can see different implementations compared with Gate Equivalent (GE) but this doesn't apply to FPGA it's mainly aimed at ASIC. Because in a FPGA the chip itself will change the size of the final algorithm on the chip. So, to compare the different implementations I just used the different logic block needed to map each algorithm in the Zybo. All the value given here are either straight from Vivado or calculated from value from Vivado.

TABLE 1
COMPARATIVE TABLE BETWEEN THE IMPLEMENTATION

Implementations	Size (LUTs) Registers Muxes	Max Speed (MHz)	Throughput (Mbits/s)	Power (mW)	Energy (J/bits)
Speed	5916 212 32	9.7	620.8	1	1.61×10^{-15}
Pipeline	2602 1600 0	153	9792	8	0.82×10^{-15}
Resources	269 215 32	174.62	429.8	11	25.6×10^{-15}

The Size, Power and Max Speed are real value (applicable in the real world) the other are calculated from these values. So, the 9 Gbits/s for example would be hard to achieve and definitely unachievable with the AXI bus.

These values are coherent with the goals I had for every implementation.

In hindsight in front of this numbers the speed implementation is actually not that impressive specially compared to the resources version. The throughput which is probably the best way to compare them is really close between them and the difference in size is definitely worth it. And to be use on the Zybo this is definitely the best one as the AXI bus will never follow the pipeline implementation (at least not the 'slow' version of the AXI bus) as it already has trouble to follow the speed version. The cipher ready flag is rarely (to not say never) used, it shows that this implementation is fast enough for our usage.

But it worth to dig a bit more in the pipeline version as I already proposed multiple times because with this kind of speed it definitely could be interested, it could be use on every side of a descent speed bus without making it slower, so it could definitely make the chosen bus more secure to outside attacks.

V. Co-Design

In this part I'm going to give you an idea of my current thinking of co-design between logic and processor on specific SoC like the Zynq chip.

We are going to suppose you have some kind of project that could be implemented in software but for some reason you might want to make it faster. If you have access to an SoC that have ARM processor and Programable logic or the mean to make your own chip combining some kind of processor and logic. You know that logic is faster as it's specially design for the application, but it's is also expensive to design, and you will probably be more limited in memory and size of your application. So, one idea would be to use a combination of these 2 solutions: classical processor application and hardware application. That's what is reference in this document as Co-design.

The 1st thing your need to realize is that breaking a project between hardware and software will not automatically make it faster if it's done wrong, I would even say it will be more than likely slower if it's not perfectly implemented.

The 1st problem is the communication between the two entities. You will have some kind of bus to discuss between the 2 different systems and this will create latency, a lot of it. So, your 1st concern will be to make sure that the time won by implanting in hardware won't be lost by the time needed to get your data through the system.

Second thing you need to take care of is to know the strong part of each system. For example, because most

processor have hardware function for floating point arithmetic your processor should be attributed the part with the most floating points. The pros of the processor include but does not limit to: external communication (ethernet/network) most likely threw Linux, large memory management, file management, etc… In the other side logic is really efficient for hard algorithm with a lot of small steps and multi-channel (parallel) algorithm. The perfect example would be image processing, it usually includes so kind of multi-channel thinking to implement it in multiple pixel at once for example and image processing is usually just a long succession of small step like addition, subtraction, etc.…

And the last point to take care of is the size of each of the subdivision. When you do co-design you are tempted to break it into too many part, that will slow the project a lot due to the communication issue like stated earlier so when you make a subdivision don't forget to also take into account the lost time and the best way I found to make sure that the time lost from the communication isn't too high in front of the win is to keep the subdivision really big so you will optimize the data transfer. For example, if you do some kind of data processing on a 64-bit data packet if the processing is split too much you will send the same data but just at different state back and forward again and again. So, keep your subdivision block size coherent and the division need to be coherent for the data too!

In my case I had the Piccolo-80 algorithm. It applies on a 64 Bits block and have an 80 bits key. Furthermore, the Piccolo algorithm is lightweight meaning no complex operation, in fact there only is lookup table and Xor gates… So, it would have break common sense to break piccolo into multiple part because the time needed to send and receive the needed data is too large in front of the combinatory time. So, I went into a different approach and went to make what I call a Hardware Function. It's basically like a normal software function but instead of running the algorithm it only calls the hardware implementation of the algorithm. On the software part once, the function is implemented the use just look like the use of any function. Exception made of the fact you need a hardware pointer but once your final hardware system is designed you can abstract that thanks to the generated define that give you the needed address. I personally use this function on simple test, but it could be use into more complex application. And as mention before hopefully this function will be usable through Linux with a bit of effort.

VI. Conclusion

This project should only be the beginning a lot of improvement can be made.

The diffusion matrix (m_box.vhd) and different key scheduling component are pretty well design but I'm sure that with a bit more of thinking and testing there would be a way to use a bit less hardware space.

For the algorithm themselves I made decision with one particular propriety of the algorithm in mind. But there may be multiple way to do slightly different implementation that would me more useful in specific utilization of the algorithm.

For the rest, the S box the round permutation and the F function there is a proposition of implementation on the presentation paper [1] (the proposition is for the full algorithm) that I chose not to use because the earnings of this implementation are to me a bit low in front of the cost in time and complexity to implement.

And finally, the obvious way to make this project even better (and probably the goal for the next person to take this project up) is to use Linux on the board and use the hardware threw Linux but I only got the time to dream of it. This could turn this project into a really user-friendly Linux package.

Acknowledgment

I would like to thank Dr. Xiaojun Wang, professor at Dublin City University, for taking on his time to explain me the goals of the project, for giving me some leads of research and most of all for giving me the opportunity to work on this subject. Thanks also to the Dublin City University and ECE Paris to provide free access to the IEEE database which contains very interesting scientific papers. Finally, I thank my friends for supporting me during my long writing days while they were in holidays.

Notation

$a_{(b)}$: b denotes the bit length of a.
$a|b$ or $(a|b)$: Concatenation.
$a \leftarrow b$: Updating a value of a by a value of b.
${}^t\boldsymbol{a}$: Transposition of a vector or a matrix $\boldsymbol{a}$.
$\{a\}_b$: Representation in base b.

References

[1] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda,Toru Akishita, and Taizo Shirai ,"Piccolo: An Ultra-Lightweight Blockcipher", CHES 2011, pp. 342-357, 2011.

[2] Marine Minier, "On the Security of Piccolo Lightweight Block Cipher against Related-Key Impossible Differentials", Progress in Cryptology - INDOCRYPT 2013, pp. 308-318, December 2013.

[3] Seyyed Arash Azimi, Zahra Ahmadian, Javad Mohajeri, Mohammad Reza Aref, "Impossible Differential Cryptanalysis of Piccolo Lightweight Block Cipher", ISCISC 2014 11th, Sept. 2014.

[4] KhanAcademy, "Journey into cryptography", https://www.khanacademy.org/computing/computer-science/cryptography.

[5] Contel Dradford, "5 Common Encryption Algorithms and the Unbreakables of the Future", https://www.storagecraft.com/blog/5-common-encryption-algorithms/, July 2014.

[6] Ssh Communication security, "CRYPTOGRAPHY FOR PRACTITIONERS", https://www.ssh.com/cryptography/, August 2017.

[9] Digilent, "ZYBO Reference Manual", https://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPZYBO/documentation/ZYBO_RM_B_V6.pdf, February 2014.

[10] Digilent, "Installing Vivado and Digilent Board Files", https://reference.digilentinc.com/vivado/installing-vivado/start.

[11] Digilent, "Getting Started with Vivado", https://reference.digilentinc.com/vivado/getting_started/start.

[12]Digilent, “Getting Started with the Vivado IP Integrator”, https://reference.digilentinc.com/vivado/getting-started-with-ipi/start.
[13]Digilent, “Creating a Custom IP core using the IP Integrator”, https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-creating-custom-ip-cores/start.
[14] Xilinx, “All Programmable SoC with Hardware and Software Programmability”, https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#productTable.
[15] Ryad Benadjila, Jian Guo, Victor Lomné, Thomas Peyri, “lightweight-crypto-lib”, https://github.com/rb-anssi/lightweight-crypto-lib, February 2017.
[16] Louise H. Crockett Ross A. Elliot Martin A. Enderwitz Robert W. Stewart, “The Zynq Book”, Xilinx, July 2014.
[17] Zi Reviews Tech, “Why do some phone SoCs (System on a Chip) suck?”, https://www.youtube.com/watch?v=a7Sir9xVETo, Youtube, January 2018.
[18] Xiaojun Wang, “EE540 - HDL and High Level Logic Synthesis”, Loop, 2017.
[19] Wikipedia, “Field Programmable Gate Array”, https://en.wikipedia.org/wiki/Field-programmable_gate_array
[20] ARM, “Efficient Application Processors for Every Level of Performance”, https://www.arm.com/products/processors/cortex-a
[21] Dennis M. Ritchie, “The Development of the C Language”, Bell Labs / Second History of Programming Languages conference, Cambridge, April 1993.
[22] David Kahn, “The Codebreakers”, Signet Book, 1973.
[23] Eugen Antal, Pavol Zajac, “Key Space and Period of Fialka M-125 Cipher Machine”, Cryptologia 39:2, pages 126-144, 2015.
[24] Eli Biham, Adi Shamir, “Differential Cryptanalysis of Data Encryption Standard”, Springer Verlag.
[25] Yogesh Kumar, Rajiv Munjal, Harsh Sharma, “Comparison of Symmetric and Asymmetric Cryptography with Existing Vulnerabilities and Countermeasures”, IJCSMS International Journal of Computer Science and Management Studies, Vol. 11, Issue 03, Oct 2011.
[26] Aidan O’ Boyle, “Real-Time Audio Filtering on Raspberry Pi Literature Review”, DCU Master of Engineering, January 2016.
[27] Subariah Ibrahim and Mohd Aizaini Maarof, “Diffusion Analysis of a Scalable Fiestel Network”, World Academy of Science, Engineering and Technology International Journal of Computer, Electrical, Automation, Control and Information Engineering Vol:1, No:5, 2007.
[28] Preshing on Programming, “How to Build a GCC Cross-Compiler”, http://preshing.com/20141119/how-to-build-a-gcc-cross-compiler/, November 2014.
[29] GCC, “GCC, the GNU Compiler Collection”, http://gcc.gnu.org/
[30] Proxacutor, “Les FPGA”, http://proxacutor.free.fr/.
[31] Courtois N.T., Pieprzyk J. (2002) Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In: Zheng Y. (eds) Advances in Cryptology — ASIACRYPT 2002. ASIACRYPT 2002. Lecture Notes in Computer Science, vol 2501. Springer, Berlin, Heidelberg.
[18] ARM, “ARM Cortex-A Series Programmer’s Guide”, January 2014.
[32] ARM, “Cortex-A9 | Technical Reference Manual”, June 2012.
[33] ARM Developer, “Cortex-A9”, https://developer.arm.com/products/processors/cortex-a/cortex-a9.
[34] Patrick BORE, “Piccolo-80 Sources Code”, https://github.com/PatrickBORE/masterProject.git, GitHub, 31 August 2018.

Appendix A

Literature Review

DUBLIN CITY UNIVERSITY
SCHOOL OF ELECTRONIC ENGINEERING

ECE Paris
SCHOOL OF GENERAL ENGINEERING

Efficient SoC Implementation of the lightweight cipher Piccolo-80

Literature Review

Patrick BORE
ID Number: 17211519
January 2018

MASTER OF ENGINEERING

IN

NANOTECHNOLOGY ENGINEERING

Supervised by Dr. Xiaojun Wang

Acknowledgements

I would like to thank Dr. Xiaojun Wang, professor at Dublin City University, for taking on his time to explain me the goals of the project, for giving me some leads of research and most of all for giving me the opportunity to work on this subject. Thanks also to the Dublin City University and ECE Paris to provide free access to the IEEE database which contains very interesting scientific papers. Finally, I thank my friends for supporting me during my long writing days while they were in holidays.

Declaration

I hereby declare that, except where otherwise indicated, this document is entirely my own work and has not been submitted in whole or in part to any other university.

Signed:  Date: 29/01/2018

Abstract

Cryptography has never been as important as it is nowadays. Most of numeric technology use cryptography at one point.
The main aim of this project is to implement the Piccolo 80 bits on a SoC[1] (Arm processor and Programmable logic). Piccolo is a recent lightweight block cipher with 64 bits block. In this lecture review is to get a deep understanding of the algorithm behind Piccolo and every technology that going to be useful to this project.
This new implementation could help, if the result follow, making Piccolo wildly use.

Table of Contents

Acknowledgements ii

Declaration ii

Abstract iii

Chapter 1: Introduction (max 3 pages) 2

Chapter 2: Technical Background 4

Piccolo 4

General Structure 5

Key generation algorithm 6

F function 6

Decryption 6

C Language 7

Cross compilation 7

Zynq Chip 7

Chapter 3: Methods 9

Piccolo Security 9

Performance of the C implementation of Piccolo 9

Software Hardware co-design 9

Chapter 4: Conclusion and Outline 10

Different steps of the project 10

Conclusion 10

References 11

Appendix 13

Notations 13

Table of Figures

Figure 1.1: The Enigma Machine…….....2
Figure 1.2: Diffie-Hellman Key Exchange Protocol……...3
Figure 2.1: Simplistic Fiestel Structure……..…....4
Figure 2.2: Piccolo algorithm……..5
Figure 2.3: Key generation algorithm……..6
Figure 2.4: F function organisation……..6
Figure 2.5: Decryption keys generation algorithm……..6
Figure 2.6: Zynq Board (ZyBo)……...7
Figure 2.7: FPGA operation diagram……..8
Figure 2.8: Cortex A9……..8
Figure A.1: Notation for the Algorithms……13

Chapter 1: Introduction (max 3 pages)

Cryptography is the practice and study of techniques to protect messages from unwanted third persons [7]. It could be by making the message unreadable (main use) or by signing the message These techniques can be traced to 1900 BCE. The main use of cryptography is during war which also is the main contributor to its evolution [8]. In the early days cryptography was trivial. There was for example simple substitution with a predefined set of equivalent symbols or a simple rule for the set of equivalent symbols like in Mulavediya for Hindi [22]. But with the critical importance of message in war time and the evolution of technology more complex algorithms were develop. With the help of the new electronics the encryption and decryption could be more complex and still be done in few seconds, the most known one must be the Enigma machine used by the Reich during World War 2 [23]. It was a well-designed system which didn't, like old system simply, replace on letter by another one, every letter could be any letter every time (except for itself which revealed to be a big flaw). With this flaw, their knowledge of the algorithm behind the code and the regularity of message with template like the weather forecaster Alan Turing and his team manage to decrypt the Enigma code in just minutes every morning. The process was boosted by the BOMB machine which could be describe as an ancestor of the computers [23].

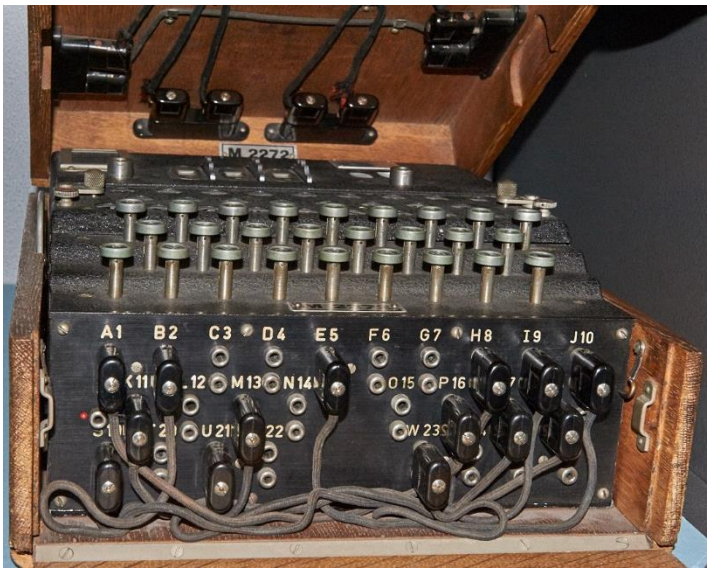


Figure 1.1: The Enigma Machine

Nowadays cryptography is mathematically analysed (Cryptanalysis) to make sure that to decrypt it even with current and future computers it would be too long [4]. This obviously doesn't consider some possible flaw in the algorithm that hasn't been found like the one found by Turing and his team or the one found in DES which was vastly used just years ago [24]. I can differentiate two kinds of cryptography symmetric and asymmetric. For symmetric method you need the same key to encrypt the message and to decrypt it. When you want to start a conversation with someone using symmetric method you need a protocol to exchange that common key, the most known and used is Diffie-Hellman [8]. Asymmetric methods are mostly reference as public key and private key. In opposition to the symmetric methods asymmetric use one key for encryption (public key) which could be seen as a padlock and another one for decryption (private key) which can be seen as the key for this specific padlock. It is generally use by companies so client can encrypt the data they send to them. All these asymmetric methods rely on maths and some proprieties of prime number [25].

Currently many methods are used, going from mail to web page threw instant message everything is encrypted in some degree [5][6]. RSA is the standard asymmetric algorithm, most of the secured webpage use it, as loop.dcu.ie for example as I can see in the specification of the HTTPS certificate. AES is the US standard for a symmetric encryption which is used by the NSA.

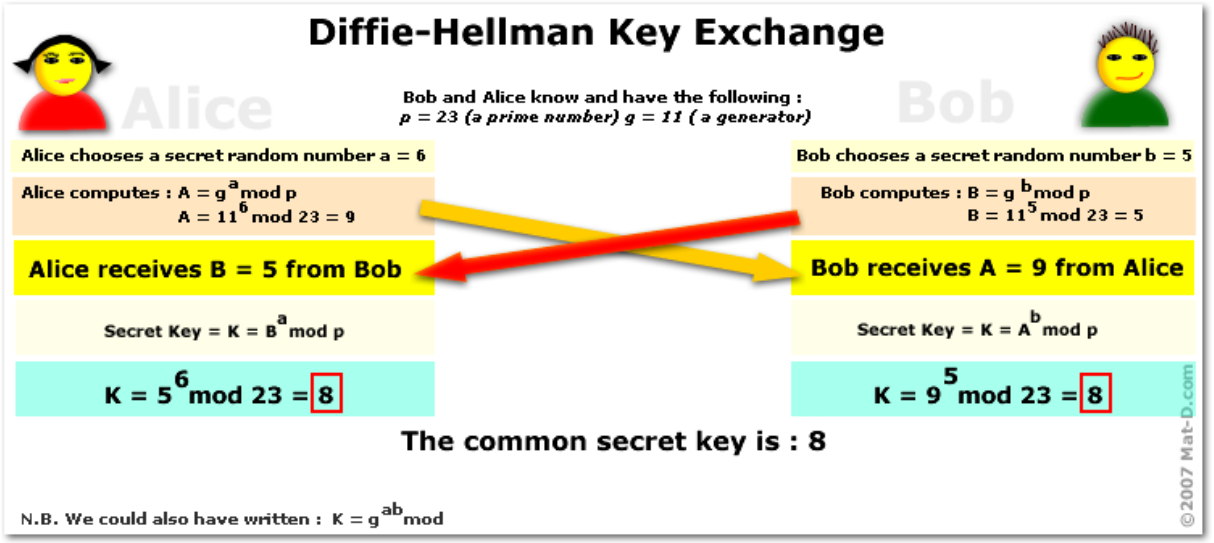


Figure 1.2: Diffie-Hellman Key Exchange Protocol

This project aims to implement a recent (2011) symmetric algorithm PICCOLO. Will see in more detail this algorithm in the next chapter but this algorithm is ultra-lightweight and optimise for very limited embedded system like RFID [1]. But this algorithm has already been implemented in VHDL by their creator and in C by a third party [11]. In this project in addition to test the performance of these two implementations a third implementation (a co-design between a C processor and some FPGA module on a SoC[1]) will be also tested to see improvement of the performances.

The main issue with this approach will be to design an efficient implementation of Piccolo that take advantage of the processor and the FPGA [14].

The remainder of this paper is structured as follow. The chapter two includes the description of current technology for the FPGA and processor on the SoC[1] used in this project with a detailed description of the Piccolo algorithm. The chapter three provides further explanations about the security of Piccolo. It also provides information about the different part this project will develop. The chapter four concludes by outlining the important information of this paper and the solution considered for the project [26].

Chapter 2: Technical Background

Piccolo

Piccolo is a block cipher [1]. Which mean that this algorithm will only encrypt a full block. So, if you want to encrypt a message you will first need to cut the message in a multitude of blocks, 64bits (which is equivalent to 8 ASCII character). There is 2 main types of algorithms: Substitution Permutation Network (SPN) and Fiestel Structure (Generalised Fiestel Network GFN). For example, AES follow a SPN and DES follow a GFN [1]. Fiestel type structure is the main type use for lightweight block cipher.

SPN is pretty much self-explanatory but what is a Fiestel Structure. The figure 2.1 just below show a simple implementation of a Fiestel structure. The signals go top to bottom and right to left.

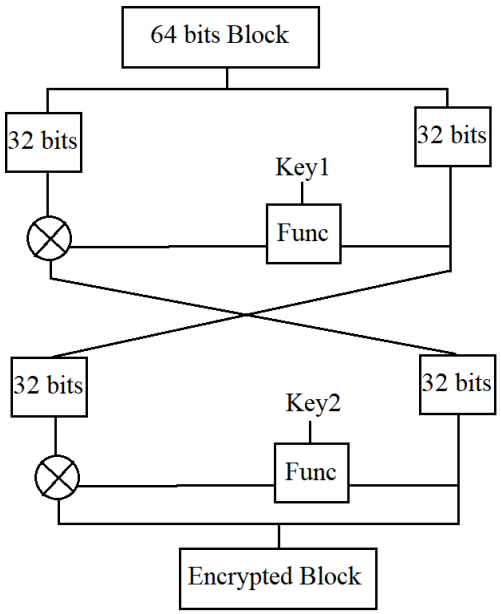


Figure 2.1: Simplistic Fiestel Structure

The keys are generated from the ‘main’ key. So, a Fiestel structure is the repetition of 2 steps. Firstly, you modify some part of the block and then you scramble the position of everything in the block [27].

Piccolo use either an 80 bits key or a 128 bits key. My project is mostly oriented on the 80 bits version. But most of the algorithm isn’t change between these 2 versions but every time this will come into account it will highlight it.

General Structure

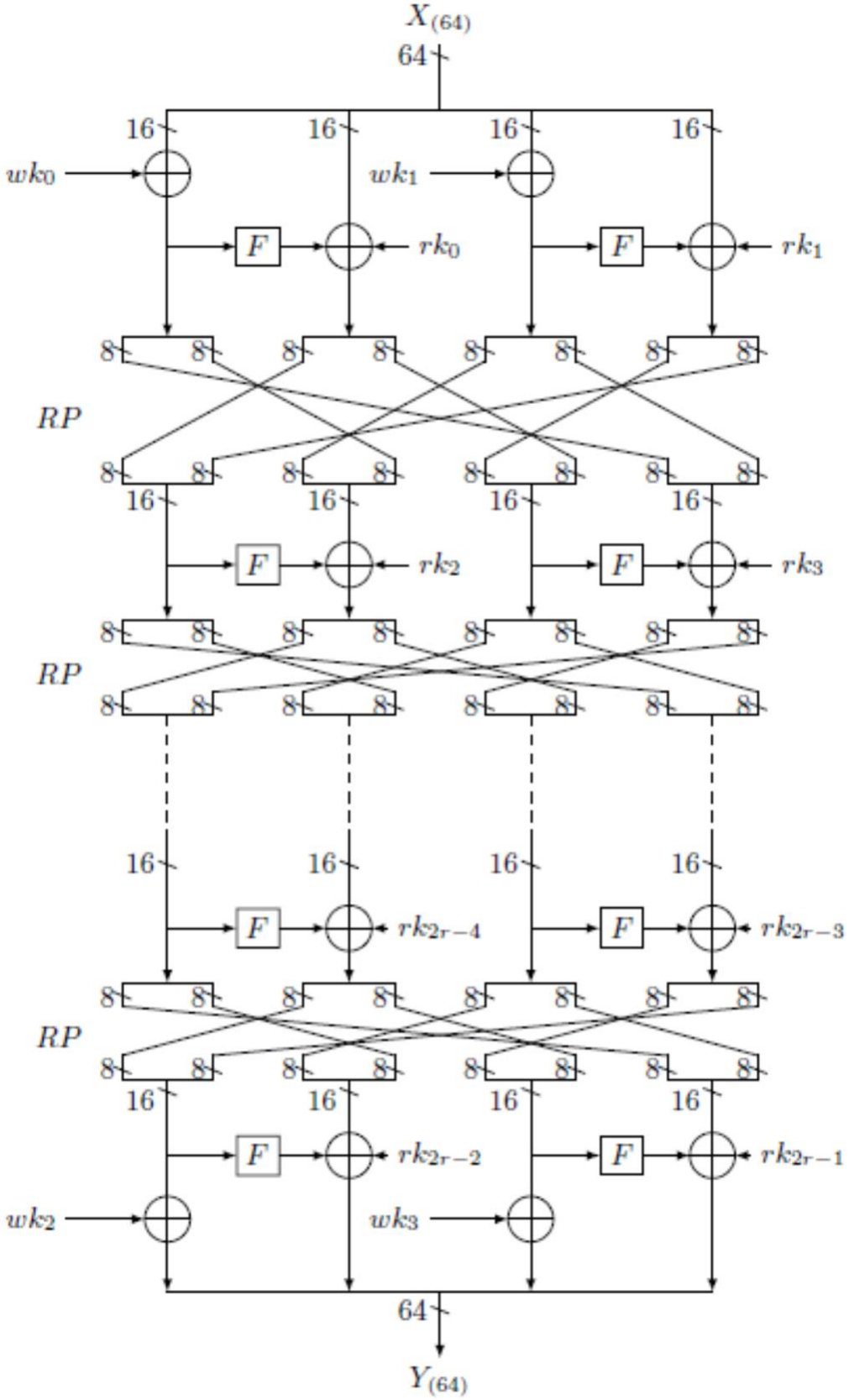


Figure 2.2: Piccolo algorithm [2]

Where wk_i $(0 \leq i < 4), rk_j$ $(0 \leq j < 2r)$ are generated keys by an algorithm on the 80 bits key. r is 25 for the 80 bits version.

Key generation algorithm

Constant Values:

$$con_{2i}^{80}|con_{2i+1}^{80} \leftarrow (c_{i+1}|c_0|c_{i+1}|\{00\}_2|c_{i+1}|c_0|c_{i+1})\ xor\ \{0F1E2D3C\}_{16}$$

With c_i the 5 bits representation of i.
For example:

$$con_0^{80}|con_1^{80} = 119286077$$

Algorithm $KS_r^{80}(K_{(80)})$:

$wk_0 \leftarrow k_0^L|k_1^R, wk_1 \leftarrow k_1^L|k_0^R, wk_2 \leftarrow k_4^L|k_3^R, wk_3 \leftarrow k_3^L|k_4^R$

for $i \leftarrow 0$ to $(r-1)$ do

$$(rk_{2i}, rk_{2i+1}) \leftarrow (con_{2i}^{80}, con_{2i+1}^{80}) \oplus \begin{cases} (k_2, k_3)\ (\text{if } i \bmod 5 = 0 \text{ or } 2) \\ (k_0, k_1)\ (\text{if } i \bmod 5 = 1 \text{ or } 4) \\ (k_4, k_4)\ (\text{if } i \bmod 5 = 3), \end{cases}$$

Figure 2.3: Key generation algorithm

Where k_i is the 16 bits i[th] part of the main 80 bits key. And the rest is the same as before.

F function

F is a 16 bits function.

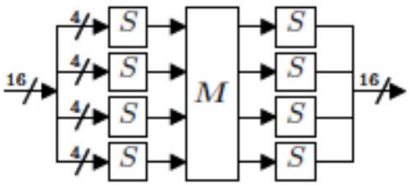


Figure 2.4: F function organisation [1]

Where S is a 4 bits substitution an M a 16 bits calculation [1].

x_i	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x_i)$	E	4	B	2	3	8	0	9	1	A	7	F	6	C	5	D

With $(0 \leq i < 4)$ and $X = x_0|x_1|x_2|x_3$

$$M(X): \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 3 & 1 & 2 & 3 \\ 3 & 3 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Decryption

To decrypt a block, you use the exact same system, you input your crypt block (still 64 bits) you just modify the different generated keys by following these rules:

$$wk_0' \leftarrow wk_2, wk_1' \leftarrow wk_3, wk_2' \leftarrow wk_0, wk_3' \leftarrow wk_1$$

for $i \leftarrow 0$ to $r-1$ do

$$rk_{2i}'|rk_{2i+1}' \leftarrow \begin{cases} rk_{2r-2i-2}|rk_{2r-2i-1}\ (\text{if } i \bmod 2 = 0) \\ rk_{2r-2i-1}|rk_{2r-2i-2}\ (\text{if } i \bmod 2 = 1) \end{cases}$$

Figure 2.5: Decryption keys generation algorithm [1]

Where everything with a prime (‘) is the decryption version of the key.

C Language

For this project I will need a software implementation of Piccolo. I am going to use C as it is the most supported language. For example, Piccolo has already been implemented in C [11]. C has been implemented in the 70's by Bells laboratory. It basically is a corrected version of B which was wildly use then [21].

Cross compilation

For the C code to be executed by any processor it need to be "translated" to machine language which is dependent on the processor. This is compilation, well actually it's a compilation chain as there is multiple steps [28]. Usually when you compile code you do it so it can be executed on your computer but in our case, we want, on our computer to compile C code for another system this is called Cross Compilation. There is multiple way to do that it usually depends on your target. But you either have a compilation chain designed specifically for your target (it happens when companies release new hardware they often develop their compilation chain with it) or you have to create it yourself. In this case GCC (GNU Compiler Collection) will be the biggest part of your system as it's support compilation for C for most hardware [29]. The reason why GCC is so well furnish in term of supported hardware is because it's a free software, as it's made and improved by the community (with some restriction).

Zynq Chip

Zynq chip are manufactured by Digilent / Xilinx and have the particularity against most of their product there isn't only Programmable logic (Field Programmable Gate Array FPGA) but there also is an ARM processor [12]. Which make it a SoC as there is two systems on the same chip FPGA and ARM [13]. The main use for SoC is to miniaturize system as it's getting harder and harder to miniaturize every component of the system.

One of the fundamental point of this project is that it should be done on a Zynq chip and in fact more precisely on a ZyBo which is a test board for their Z-7010 Zynq chip. One this board there is a dual core processor ARM Cortex-A9 and a Xilinx 7-series FPGA logic [9].

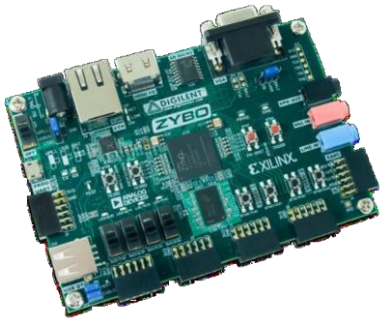


Figure 2.6: Zynq Board (ZyBo) [9]

To use the FPGA part of the ZyBo I will need to use an HDL (Hardware Description language). There is two language Verilog and VHDL. But even if I studied Verilog I've been practicing with VHDL for longer so in this project I will naturally use VHDL. As the ZyBo is from Xilinx the I will be able to use Vivado which is and IDE for Xilinx's FPGA. It also has a simulation tool that will be a big help for designing block in VHDL [15].

FPGA are basically a lot of logic component (flip-flop, logic gate , etc …) put in an array with no fixed connection. So with this technology you are able to physically connect different component even if you are doing it threw a language that look like C because VHDL is not a software language [16].

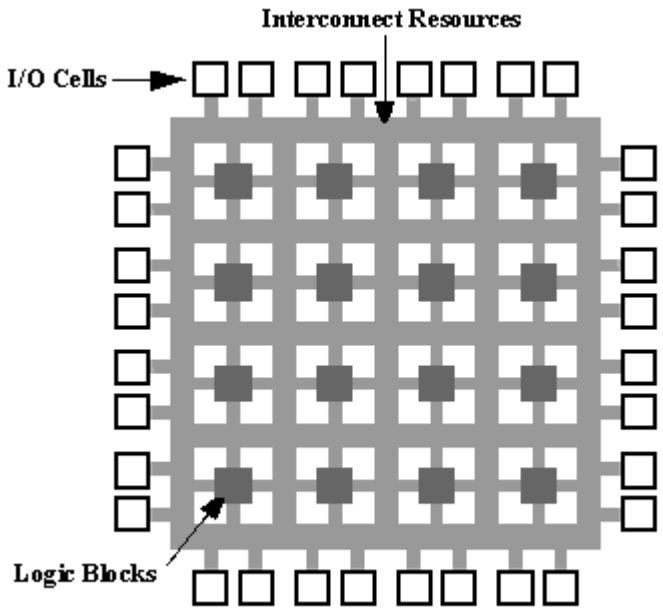


Figure 2.7: FPGA operation diagram [31]

ARM processor is the leader in embedded system. They have different processor that range from basic processor to eight core powerful processor [17]. In the Z-7010 there is a dual core Cortex-A9 processor. This processor is 32 bits. As there is two core I will be able to run 2 thread at the same time (A thread is like a new ,almost independent software, the best analogy is tabs on web navigator). It has an ArmV7 architecture and therefore the armv7 instruction set [19].

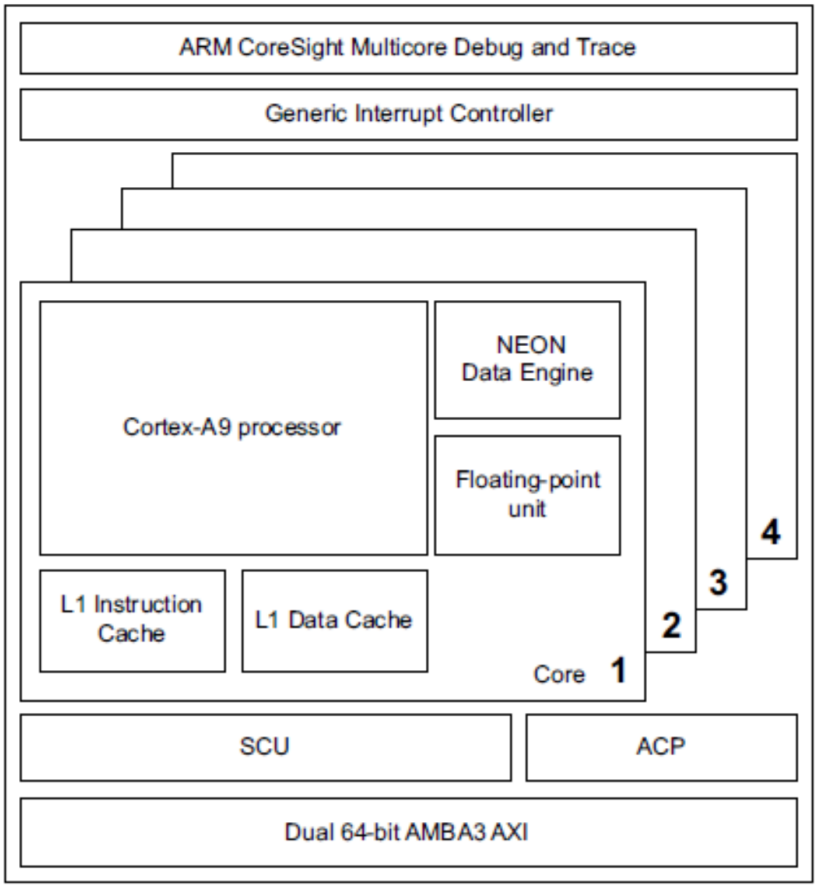


Figure 2.8: Cortex A9 [18]

Chapter 3: Methods

Piccolo Security

As any cipher to know it's good you need to try to break it. Currently in cryptanalysis there is multiple methods to break ciphers [32]. The most obvious one would be brute force which is simply testing all the key to see if any match the encrypted block you have. But even for Piccolo-80 there is still 2^{80} different key which is not realizable. Then you have more complex method that usually try to find some small mistake in the algorithm to reduce this number, that's what A. Turing and his team did. For Fiestel block cipher the main methods to crack it are Meet-in-the-middle and Impossible Differential attack. During the conception of Piccolo, the team tested them and determined that Piccolo is safe against these attacks [1]. But that could be expected as they just designed it because if they find a way to break their algorithm they would have modified it before publishing. So, the real test is with time and more people trying to break it. Someone from a university in Lyon (France) tried to break it and came down to a complexity of 2^{73} with a man in the middle attack [2]. Another team from Tehran (Iran) bring the complexity down to $2^{55.18}$ with an impossible differential attack [3] which is still a fair amount specially for a lightweight block cipher. So any new cipher is never really secure until enough people tried to break it and it usually only arrive when it start to be commonly use in some kind of technology.

Performance of the C implementation of Piccolo

As mention before in this report we already have an optimize implementation of Piccolo in C [11]. So first thing would be to test this against the Hardware implementation made by the team that developed Piccolo. For that it would have to be an implementation on an embedded system. For use this project I will simply use the Cortex A9 included in the Zynq chip [12]. To compare them I will most likely start by the speed how many clock cycles does it need. Them if possible compare the energy consumption and lastly the size of the device in our case between the Cortex A9 processor against the equivalent logic circuit.

Software Hardware co-design

The main aim of this project is to realize an optimize implementation of Piccolo-80 using hardware and software. So, for that a big part will be analysing the C and Logic implementation to see what part would be more efficient in which part of the system. Thanks to this co-design we will be able to move on to three fronts at the same time, the two cores of the ARM processor and the logic part (which could be divide in more concurrent part). But for all this part to work together communication will be key and might be the reason why this implementation won't improve on the already existing implementation [9].

At the end I will be testing its performance on the same parameter than for the C implementation.

Chapter 4: Conclusion and Outline

Different steps of the project

- Getting familiar with the Zybo and the Zynq chip:
  - Get all the design tools
  - Running a simple C soft
  - Make a simple logic application
  - Make a simple connection between the ARM and a logic circuit
- Running the C implementation of Piccolo on the ZyBo
- Testing its performances
- Running the VHDL implementation of Piccolo on the Zybo
- Designing the new C/VHDL implementation of Piccolo
- Testing it performance

Conclusion

This project is about a new implementation on a lightweight block cipher: Piccolo-80. The idea is to use a Zynq Chip which integrated an ARM processor and so Programmable logic to improve on the mono-technology implementation. As this new implementation would be just on one chip it could be the implementation if piccolo start to be wildly use.

References

[1] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda,Toru Akishita, and Taizo Shirai ,"Piccolo: An Ultra-Lightweight Blockcipher", CHES 2011, pp. 342-357, 2011.

[2] Marine Minier, "On the Security of *Piccolo* Lightweight Block Cipher against Related-Key Impossible Differentials", Progress in Cryptology - INDOCRYPT 2013, pp. 308-318, December 2013.

[3] Seyyed Arash Azimi, Zahra Ahmadian, Javad Mohajeri, Mohammad Reza Aref, "Impossible Differential Cryptanalysis of Piccolo Lightweight Block Cipher", ISCISC 2014 11th, Sept. 2014.

[4] KhanAcademy, "Journey into cryptography", https://www.khanacademy.org/computing/computer-science/cryptography.

[5] Contel Dradford, "5 Common Encryption Algorithms and the Unbreakables of the Future", https://www.storagecraft.com/blog/5-common-encryption-algorithms/, July 2014.

[6] Ssh Communication security, "CRYPTOGRAPHY FOR PRACTITIONERS", https://www.ssh.com/cryptography/, August 2017.

[7] Wikipedia, "Cryptography", https://en.wikipedia.org/wiki/Cryptography.

[8] Wikipedia, "Cryptographie", https://fr.wikipedia.org/wiki/Cryptographie.

[9] Digilent, "ZYBO Reference Manual", https://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPZYBO/documentation/ZYBO_RM_B_V6.pdf, February 2014.

[10] Xilinx, "All Programmable SoC with Hardware and Software Programmability", https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#productTable.

[11] Ryad Benadjila, Jian Guo, Victor Lomné, Thomas Peyri, "lightweight-crypto-lib", https://github.com/rb-anssi/lightweight-crypto-lib, February 2017.

[12] Louise H. Crockett Ross A. Elliot Martin A. Enderwitz Robert W. Stewart, "The Zynq Book", Xilinx, July 2014.

[13] Wikipedia, "System on a chip", https://en.wikipedia.org/wiki/System_on_a_chip.

[14] Zi Reviews Tech, "Why do some phone SoCs (System on a Chip) suck?", https://www.youtube.com/watch?v=a7Sir9xVETo, Youtube, January 2018.

[15] Xiaojun Wang, "EE540 - HDL and High Level Logic Synthesis", Loop, 2017.

[16] Wikipedia, "Field Programmable Gate Array", https://en.wikipedia.org/wiki/Field-programmable_gate_array

[17] ARM, "Efficient Application Processors for Every Level of Performance", https://www.arm.com/products/processors/cortex-a

[18] ARM, "ARM Cortex-A Series Programmer's Guide", January 2014.

[19] ARM, "Cortex-A9 | Technical Reference Manual", June 2012.

[20] ARM Developer, "Cortex-A9", https://developer.arm.com/products/processors/cortex-a/cortex-a9.

[21] Dennis M. Ritchie, "The Development of the C Language", Bell Labs / Second History of Programming Languages conference, Cambridge, April 1993.

[22] David Kahn, "The Codebreakers", Signet Book, 1973.

[23] Eugen Antal, Pavol Zajac, "Key Space and Period of Fialka M-125 Cipher Machine", Cryptologia 39:2, pages 126-144, 2015.

[24] Eli Biham, Adi Shamir, "Differential Cryptanalysis of Data Encryption Standard", Springer Verlag.

[25] Yogesh Kumar, Rajiv Munjal, Harsh Sharma, "Comparison of Symmetric and Asymmetric Cryptography with Existing Vulnerabilities and Countermeasures", IJCSMS International Journal of Computer Science and Management Studies, Vol. 11, Issue 03, Oct 2011.

[26] Aidan O' Boyle, "Real-Time Audio Filtering on Raspberry Pi Literature Review", DCU Master of Engineering, January 2016.

[27] Subariah Ibrahim and Mohd Aizaini Maarof, "Diffusion Analysis of a Scalable Fiestel Network", World Academy of Science, Engineering and Technology International Journal of Computer, Electrical, Automation, Control and Information Engineering Vol:1, No:5, 2007.

[28] Preshing on Programming, "How to Build a GCC Cross-Compiler", http://preshing.com/20141119/how-to-build-a-gcc-cross-compiler/, November 2014.

[29] GCC, "GCC, the GNU Compiler Collection", http://gcc.gnu.org/.

[30] Image,http://www.cryptomuseum.com/crypto/enigma/m3/img/m3_m2272_000_large.jpg

[31] Proxacutor, "Les FPGA", http://proxacutor.free.fr/.

[32] Courtois N.T., Pieprzyk J. (2002) Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In: Zheng Y. (eds) Advances in Cryptology — ASIACRYPT 2002. ASIACRYPT 2002. Lecture Notes in Computer Science, vol 2501. Springer, Berlin, Heidelberg.

Appendix

SoC[1]: System on a Chip

Notations

$a_{(b)}$: b denotes the bit length of a.
$a|b$ or $(a|b)$: Concatenation.
$a \leftarrow b$: Updating a value of a by a value of b.
${}^t\boldsymbol{a}$: Transposition of a vector or a matrix $\boldsymbol{a}$.
$\{a\}_b$: Representation in base b.

Figure A.1: Notation for the Algorithms

Appendix B

Project Design Plan

DUBLIN CITY UNIVERSITY
SCHOOL OF ELECTRONIC ENGINEERING

ECE Paris
SCHOOL OF GENERAL ENGINEERING

Efficient SoC Implementation of the lightweight cipher Piccolo-80

Project Design Plan

Patrick BORE
ID Number: 17211519

MASTER OF ENGINEERING

IN

NANOTECHNOLOGY ENGINEERING

Supervised by Dr. Xiaojun Wang

Table of Contents

Time Table 1

Project Goals 2

Project Scope 2

Annex 3

Time Table

Nom	Date de début	Date de fin
Literature Review	08/01/18	29/01/18
Oral Presentation	30/01/18	19/02/18
Normal Semester	20/02/18	21/05/18
Handling ZyBo Tools	22/05/18	22/06/18
3 VHDL implementations	25/06/18	13/07/18
Speed optimized implementation	25/06/18	29/06/18
Pipeline Implementation	02/07/18	06/07/18
Resource-optimized implementation	09/07/18	13/07/18
Test bench VHDL Implementation	16/07/18	20/07/18
Co-design implementation	23/07/18	17/08/18
Co-design testing	20/08/18	24/08/18
Protfolio Redaction	06/08/18	31/08/18
Presentation	03/09/18	07/09/18

This has been done with GanttProject. A graphic representation is in the annex but not really readable. Here is a link to the .gan of the project so you can open it yourself if needed. (https://www.dropbox.com/s/ovympbv9nfr52mb/Project%20design%20plan.gan?dl=0)

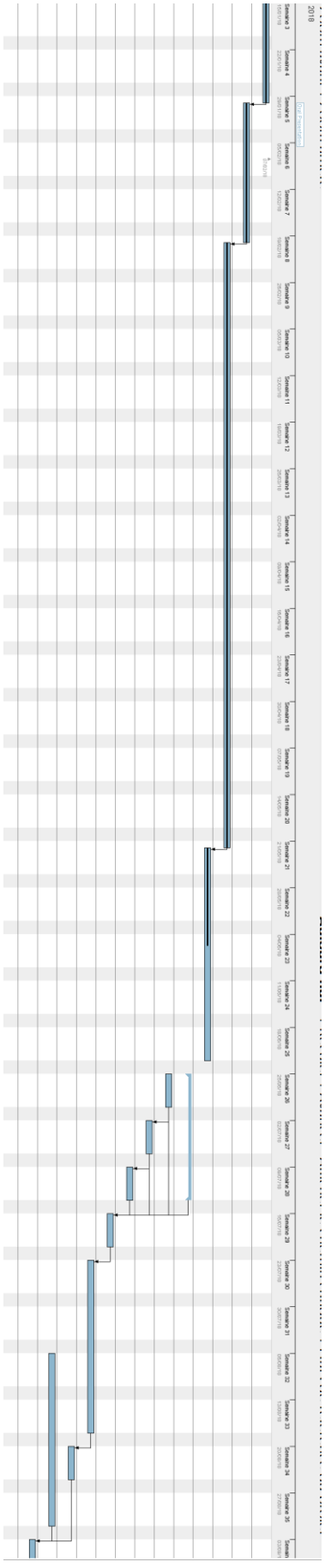
Project Goals

The main aim of the project is to implement an efficient implementation, a co-design of C and VHDL, of Piccolo-80 on a specific SoC: Zynq chip on the ZyBo board. We will also test the efficiency of 3 different pure hardware implementation against the co-design implementation.

Project Scope

- Bare Metal C programming
- Programable logic and VHDL
- Programmable logic ⇔ Processor communication
- Co-design programmable logic / processor
- Cryptography (Piccolo -80)
- Light weigh cryptography

Annex



Appendix C

Finished Project Report

DUBLIN CITY UNIVERSITY
SCHOOL OF ELECTRONIC ENGINEERING

ECE Paris
SCHOOL OF GENERAL ENGINEERING

Efficient SoC Implementation of the lightweight cipher Piccolo-80

Finished Project Report

Patrick BORE
ID Number: 17211519

MASTER OF ENGINEERING

IN

NANOTECHNOLOGY ENGINEERING

Supervised by Dr. Xiaojun Wang

I. Table of Contents

II. Introduction 1
A. Goals 1
B. Project Flow 1
C. Time management 2
III. Piccolo 3
A. Main Problems and the solution found 3
1. Piccolo Paper 3
2. Personal 3
B. Possible optimisation and amelioration 4
C. Test Vector 5
1. Round by Round test vector 5
2. First round extremely detailed 6
3. More test values 7
IV. Co-Design 8
A. Tutorials 8
1. Create an Slave AXI IP Block 8
2. Using the IP with the Processor 18
B. Co-Design flow 23
V. Sources and GitHub 25
A. VHDL Sources 25
1. Speed Implementation 25
2. Pipeline Implementation 25
3. Resources Implementation 26
B. IPs 26
1. Speed Implementation 26

2. Resources Implementation 27

C. How to use them 27

VI. Results 28

A. Comparative Table 28

B. Discussion 28

VII. Conclusion 29

VIII. References 30

II. Introduction

A. Goals

The aim of this project is to take advantage of the Zynq chip (SoC with programable logic) in the design of hardware function for co-design project. In this project we took a lightweight cryptography algorithm as a way to explore to possibilities of co-design. But this project is meant to be built on (probably by future master student or Phd student), that's why a big part of the report is tutorial based and that I made available everything I did with the needed explanation on my GitHub.

B. Project Flow

To do this project my organisation was quite not remarkable. The main goal was to achieve a co-design for the piccolo-80 algorithm.

After the lecture review and a discussion with my project mentor (the process will be described in more detail in the Co-design flow part in V. Co-Design) we decide to implement piccolo like a hardware function. Meaning the full algorithm will be hardware based and will be usable threw the processor C code with simple functions.

So the project did separate itself into two main goals. Design efficient hardware implementations of the algorithm (threw VHDL) and then use them threw the processor part of the Zynq chip. Two part that you can find in this report: the tutorial-based explanation for the co-design part (V. Co-Design) and comparison of the different implementations (VI. GitHub and sources and VII. Result).

For the implementation I already had the knowledge needed in HDL to do it straight away and except for the few problems explain in the next part everything went according to the plan and followed the predicted timing. So, I ended with 3 different implementations that were working on the behavioural simulation. I had a choice to improvise a way to test them in hardware with some kind of input/output system (uart / I2C) or go straight to the processor and C code of the chip to try to use the implementation that way. Because I was fairly confident that I will successfully have some kind of communication between the FPGA and the processor part of the Zynq chip I went straight to the processor / logic communication for having more time to figure out the hardware function part of the project.

So, my 1st step was to get familiar with the design tools: Vivado for the block design of hardware (IP based) and the SDK for the C code. I found two easy project/tutorial on the Zybo manufacturer website [X]. The 1st one was a led blinking [X] so to get used to the user interface and possibilities and the 2nd one was on how to make an IP block (for the block design) using AXI to communicate with the processor. My tutorial on How to create a slave AXI block is largely inspired by this project. Then I modified these two projects in order to see the possibilities and make some of the possible mistakes in order to avoid them when the time to apply that to my Piccolo-80 implementation.

I then had to rethink a bit my implementations to make them software friendly and AXI friendly. I for example didn't took the time to do that for the pipeline implementation as determine that I didn't had enough time for it (definitely a good choice).

And I finally made the IPs for both of my piccolo implementation and connected all of that in a test design to make some easy to use function for this hardware function.

C. Time management

I mainly followed the project design plan. When a task was risking to not fit inside I either simplified it of fully skip it. The perfect examples of that is the pipeline implementation that didn't got optimized for an AXI IP and the hardware test of the different algorithm before any co-design.

III. Piccolo

A. Main Problems and the solution found

1. Piccolo Paper

The paper describing the piccolo algorithm [1] is really good and did allow me to do my own implementation fairly easily. But there are few point that made me loose time that it could have been improved on. These proposed improvements are only from my personal experience with the paper.

The 1st point I noticed is when they have a subdivision of a signal they don't precise where are the most significant bits and less significant bits. Like we can see on the schematic for the S-Box for example. And on top of that they named them differently of what I'm used to. We can take the example of the Round Permutation when $x_0$which is a subdivision of the 64 bits X signal is actually the 8 most significant bits and not the 8 less significant ones like the led me to suppose. So there is too many times where the order of the bits are not specified on the schematic, they are explain on the written part but a more precise schematic would have be much appreciated.

The 2nd point is more aim for people not familiar with numerical electronics. In the different algorithm threw the paper the use of XOR logic is everywhere but I was surprise to not see it in the notation part as it should.

The 3rd point which might just be a point of view. The designer of the algorithm obviously did an implementation of it, but I couldn't find any record of their hardware implementation. It doesn't mean it doesn't exist, but it means at least that it's not easy to find.

At last but the most important is the size of the test vector … extremely small only 1 example for each version of piccolo. More of them and more random ones would allow a bigger confidence that the implementation you made is what the designer of piccolo had in mind. And even maybe on one of these examples a step by step of the different value threw the algorithm to allow easier debug if the intended result isn't there. This will be fixed by me in the D section.

2. Personal

In the F function we can find a matrix multiplication. 1st of all it took me a decent amount of time to figure out it wasn't normal multiplication but multiplication in a Galois Field. And secondly it does suppose that the reader knows about Galois Field which would only apply to people with a strong maths background or cryptography background. So an more detailed version of that part of the algorithm would definitely have been appreciated. No need for a master class on Galois Field but maybe a simple algorithm to do the needed calculation for the M matrix. Figure I. is the solution I design (low weigh and hardware oriented).
My knowledge in Galois Field operation is really limited so I'm pretty sure this could be improved to be even more efficient.

$$x'_{0(5)} = x_{0(4)}|\{0\}_2 \; xor \left(x_{1(4)}|\{0\}_2 \; xor \{0\}_2|\, x_{1(4)}\right) xor \{0\}_2|\, x_{2(4)} \; xor \{0\}_2|\, x_{3(4)}$$
$$x'_{1(5)} = \{0\}_2|\, x_{0(4)} \; xor \; x_{1(4)}|\{0\}_2 \; xor \left(x_{2(4)}|\{0\}_2 \; xor \{0\}_2|\, x_{2(4)}\right) xor \{0\}_2|\, x_{3(4)}$$
$$x'_{2(5)} = \{0\}_2|\, x_{0(4)} \; xor \{0\}_2|x_{1(4)} \; xor \; x_{2(4)}|\{0\}_2 \; xor \left(x_{3(4)}|\{0\}_2 \; xor \{0\}_2|\, x_{3(4)}\right)$$
$$x'_{3(5)} = \left(x_{0(4)}|\{0\}_2 \; xor \{0\}_2|\, x_{0(4)}\right) xor \{0\}_2|\, x_{1(4)} \; xor \{0\}_2|x_{2(4)} \; xor \; x_{3(4)}|\{0\}_2$$

$for\ i \leftarrow 0\ to\ 3\ do$

$if\ \left(x'_{i(5)}\ MSB\ is\ \{1\}_2\right)\ then$

$x'_{i(4)} = x'_{i(5)}\ xor\ \{10011\}_2$ //and only keep the 4 LSB

$else\ then$

$x'_{i(4)} = x'_{i(5)}$ // only keep the 4 LSB

Figure I. Diffusion Matrix algorithm example

B. Possible optimisation and amelioration

The 1st amelioration possible is to finish the job for the pipeline implementation. This implementation is the one with the maximal throughput but to be able to use it properly through software a flag driven FIFO is needed. And if I had to do it I would do that on the hardware for most of it at least. For the flag I would go in the same direction that the resources implementation but with more flags.

The diffusion matrix (m_box.vhd) and different key scheduling component are pretty well design but I'm sure that with a bit more of thinking and testing there would be a way to use a bit less hardware space.

For the algorithm themselves I made decision with one particular propriety of the algorithm in mind. But there may be multiple way to do slightly different implementation that would me more useful in specific utilisation of the algorithm.

For the rest, the S box the round permutation and the F function there is a proposition of implementation on the presentation paper [1] (the proposition is for the full algorithm) that I chose not to use because the earnings of this implementations are to me a bit low in front of the cost in time and complexity to implement.

The block design could definitely be improving too. For example, I had to use a GPIO IP block to have access to the xil_printf function which I'm sure an easier way and more elegant way is available if you take the time to search for it.

The use of a different AXI bus might also be beneficial as the one I use isn't the high speed one but once again didn't had enough time to go look more into that. I was more looking into a proof of concept kind of project than a consumer ready project.

And finally, the obvious way to make this project even better (and probably the goal for the next person to take this project up) is to use linux on the board and use the hardware threw linux but I only got the time to dream of it. This could turn this project into a really user friendly linux package.

C. Test Vector

As stated earlier the low size of the test vector is ridiculous so I'm going to try to do a small fix. I'm going to go through the different steps for the existing test vector and give 3 other test values that I used. I don't think more than one detailed is needed but the vector definitely should be even larger than 4 values. I would say the perfect way would be a formatted text file (to be able to use software on it) with between 10 and 100 different random values would be nice. But for obvious reason I didn't had the time for that.

1. Round by Round test vector

These values are for the given test vector (the 1st one in the table above). I consider the end of a round just after the round permutation block. So the value will be the one from just after the round permutation for each round except for the last one which will be the cipher text as the last round is a bit different. The Round keys are to be used on the state on the same line to get to the next line state. The whitening keys are: $wk_0 = 0x0033$, $wk_1 = 0x2211$, $wk_2 = 0x8877$, $wk_3 = 0x6699$.

Round #	State Value	Round Key (2*r|2*r+1)	
Debut / Round 0	0x0123456789abcdef	0x4349	0x4f4a
Round 1	0xa5ccab10bf0b01ba	0x1f0b	0x070d
Round 2	0x3561bfcc17b4a50b	0x534d	0x4748
Round 3	0x14c51761818135b4	0xa78f	0xb5a1
Round 4	0x9ee081c575091481	0x2705	0x1b0a
Round 5	0x82a175e05bfb9e09	0x7b47	0x534d
Round 6	0xd4275ba1e8be82fb	0x3701	0x1308
Round 7	0x00fce827bf15d4be	0x0b5b	0x6b43
Round 8	0x61bcbffc34590015	0xcf95	0x81ac
Round 9	0xf66e34bca9406159	0x5f1b	0x2705
Round 10	0x0a21a96e1912f640	0x135d	0x6740
Round 11	0x1d191921e2db0a12	0x6f17	0x3f03
Round 12	0x3c89e219a4f41ddb	0x2351	0x7f46
Round 13	0x4889a489e2b83cf4	0xf79b	0x9dab
Round 14	0x02b5e289d3d948b8	0x7711	0x3300
Round 15	0x91add3b583b802d9	0xcb6b	0x0b5b
Round 16	0xf14483ad794691b8	0x872d	0x4b1e
Round 17	0x8622794405f8f146	0xdb6f	0x0359
Round 18	0x117f0522e35786f8	0x1fa1	0xe9b6
Round 19	0xf02de37fad021157	0xaf27	0x5f1b
Round 20	0x9239ad2d2e1ff002	0xe361	0x1f5e
Round 21	0x0cec2e39c15e921f	0xbf23	0x5719
Round 22	0x9e50c1ec0b0f0c5e	0xf365	0x175c
Round 23	0x6a780b5046449e0f	0x47b7	0xc5bd
Round 24	0x055c467853616a44	0xc73d	0x6b16
Fin / Round 25	0x8d2bff9935f84056		

2. First round extremely detailed

There are 3 different steps: the 1st one is after the whitening keys, the 2nd one is after the F function and the round keys and last one is after the round permutation. I'll detailed the F function just a bit after.

Step 0: 0x0123456789abcdef
0x0123 XOR 0x0033 = 0x0110 , 0x89ab XOR 0x2211 = 0xabba

Step 1: 0x01104567abbacdef
F(0x0110) = 0xa325 , F(0xabba) = 0x3d69
0xa325 XOR 0x4567 XOR 0x4349 = 0xa50b , 0x3d69 XOR 0x89ab XOR 0x4f4a = 0xbfcc

Step 2: 0x0110a50babbabfcc

Step 3: 0xa5ccab10bf0b01ba

For the F function we are going to do both of the F function used in the previously. There is also 3 step for the F function: the 1st one is after the 1st layer of S-Box the second is after the diffusion matrix M (detailed later) and after the last S-Box layer. No need for more detailed for the S-Box as it's a lookup table which is given in the paper [1].

Step	F(0x 0 1 1 0)	F(0x a b b a)
1	0x e 4 4 e	0x 7 f f 7
2	0x 9 4 3 e	0x 4 f c 7
3	0x a 3 2 5	0x 3 d 6 9

For the diffusion matrix M (once again we are going to do for both the M matrix used above.

M(0xe44e):

$$x_0 = 0b01110, x_1 = 0b00100, x_2 = 0b00100, x_3 = 0b01110$$

$2 \times x_i$ is simply x_i with a right shift of 1 bit

$$2 \times x_0 = 0b11100, 2 \times x_1 = 0b01000, 2 \times x_2 = 0b01000, 2 \times x_3 = 0b11100$$

$3 \times x_i = 2 \times x_i \; XOR \; x_i$ as we are in a Galois Field

$$3 \times x_0 = 0b10010, 3 \times x_1 = 0b01100, 3 \times x_2 = 0b01100, 3 \times x_3 = 0b10010$$

And if you follow the matrix and/or the Figure I. you get:

$$x'_{0(5)} = 0b11100 \; xor \; 0b01100 \; xor \; 0b00100 \; xor \; 0b01110 = 0b11010$$
$$x'_{1(5)} = 0b01110 \; xor \; 0b01000 \; xor \; 0b01100 \; xor \; 0b01110 = 0b00100$$
$$x'_{2(5)} = 0b01110 \; xor \; 0b00100 \; xor \; 0b01000 \; xor \; 0b10010 = 0b10000$$
$$x'_{3(5)} = 0b10010 \; xor \; 0b00100 \; xor \; 0b00100 \; xor 0b11100 = 0b01110$$

There is 2 of them that need to use the irreducible polynomial:

$$x'_{0(5)} = x'_{0(5)} \; XOR \; 0b10011 = 0b01001$$
$$x'_{2(5)} = x'_{2(5)} \; XOR \; 0b10011 = 0b00011$$

So if you take only the 4 last bits :

$$x'_0 = 0b1001, x'_1 = 0b0100, x'_2 = 0b0011, x'_3 = 0b1110$$
$$M(0xe44e) = 0x943e$$

M(0x7ff7):

$$x_0 = 0b00111, x_1 = 0b01111, x_2 = 0b01111, x_3 = 0b00111$$

$2 \times x_i$ is simply x_i with a right shift of 1 bit

$$2 \times x_0 = 0b01110, 2 \times x_1 = 0b11110, 2 \times x_2 = 0b11110, 2 \times x_3 = 0b01110$$

$3 \times x_i = 2 \times x_i \; XOR \; x_i$ as we are in a Galois Field

$$3 \times x_0 = 0b01001, 3 \times x_1 = 0b10001, 3 \times x_2 = 0b10001, 3 \times x_3 = 0b01001$$

And if you follow the matrix and/or the Figure I. you get:

$$x'_{0(5)} = 0b01110 xor\ 0b10001 xor\ 0b01111 xor\ 0b00111 = 0b10111$$
$$x'_{1(5)} = 0b00111 xor\ 0b11110 xor\ 0b10001 xor\ 0b00111 = 0b01111$$
$$x'_{2(5)} = 0b00111 xor\ 0b01111 xor\ 0b11110 xor\ 0b01001 = 0b11111$$
$$x'_{3(5)} = 0b01001 xor\ 0b01111 xor\ 0b01111 xor\ 0b01110 = 0b00111$$

There is 2 of them that need to use the irreducible polynomial:

$$x'_{0(5)} = x'_{0(5)} \; XOR \; 0b10011 = 0b00100$$
$$x'_{2(5)} = x'_{2(5)} \; XOR \; 0b10011 = 0b01100$$

So if you take only the 4 last bits:

$$x'_0 = 0b0100, x'_1 = 0b1111, x'_2 = 0b1100, x'_3 = 0b0111$$
$$M(0x7ff7) = 0x4fc7$$

3. More test values

I didn't change the key (didn't had enough time but there should be values with different keys!)

Key size	Key	Plain Text	Cipher Text
80 bits	0x00112233445566778899	0x0123456789abcdef	0x8d2bff9935f84056
80 bits	0x00112233445566778899	0x0a14c5d47383fe58	0xa4ad58e01b04d445
80 bits	0x00112233445566778899	0xaad97ba12299bd18	0xfe673fc79f94cdda
80 bits	0x00112233445566778899	0x82a707a9b1b5dcfb	0xa854a64a20bee09a

IV. Co-Design

A. Tutorials

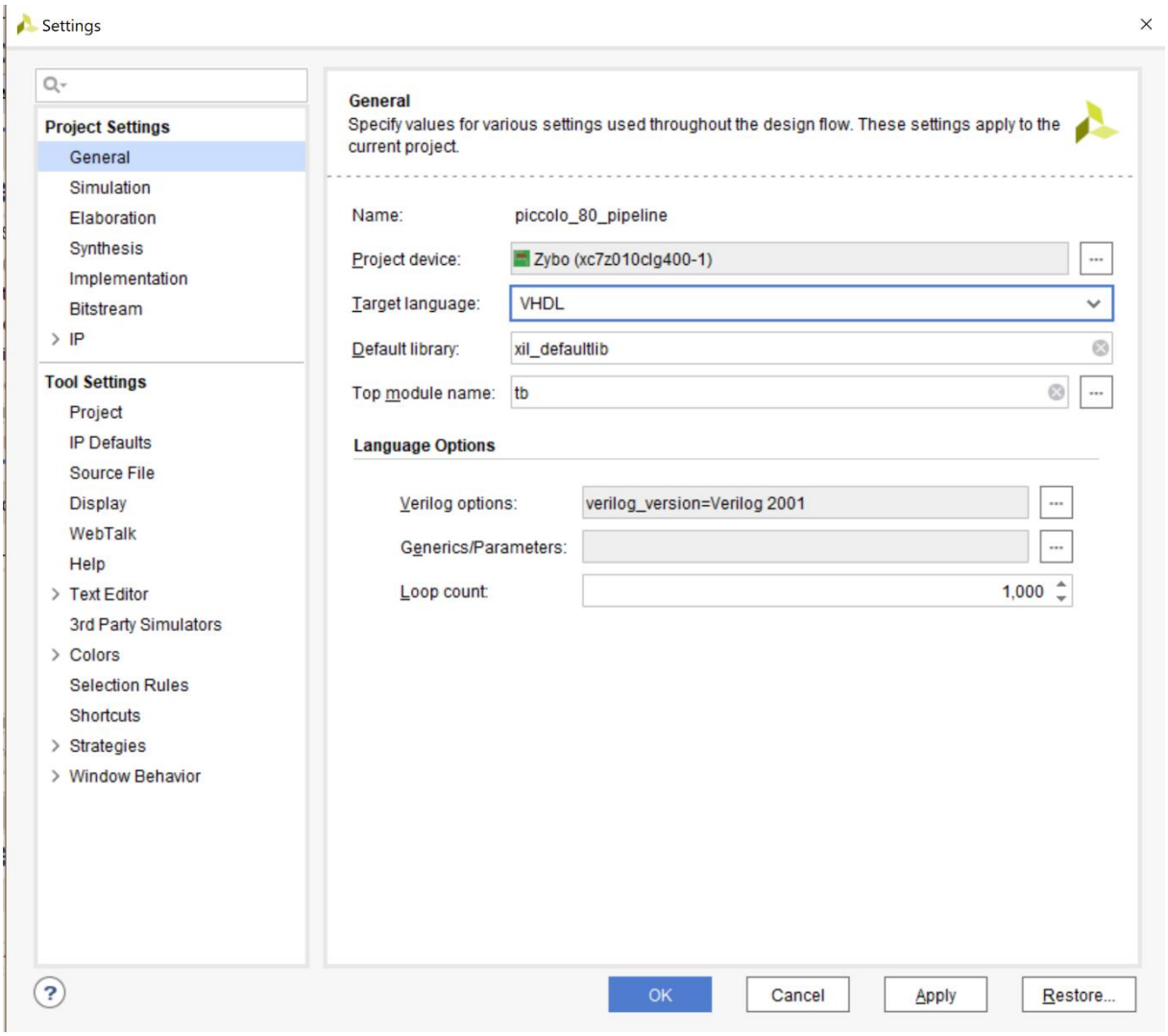
These tutorials are NOT STAND ALONE you need to do this tutorial [10] [11] [12] [13] before to first of all get familiar with Vivado and the SDK. A bit of general knowledge about FPGA, Xilinx hardware, VHDL and C code (pointer and memory and embedded oriented C coding) can't hurt neither. I realized this tutorial on a Windows 10 computer with Vivado 2018.2. But Vivado is fairly constant in its User Interface threw the version so this should work for all the version that are not to far of this one.

1. Create an Slave AXI IP Block

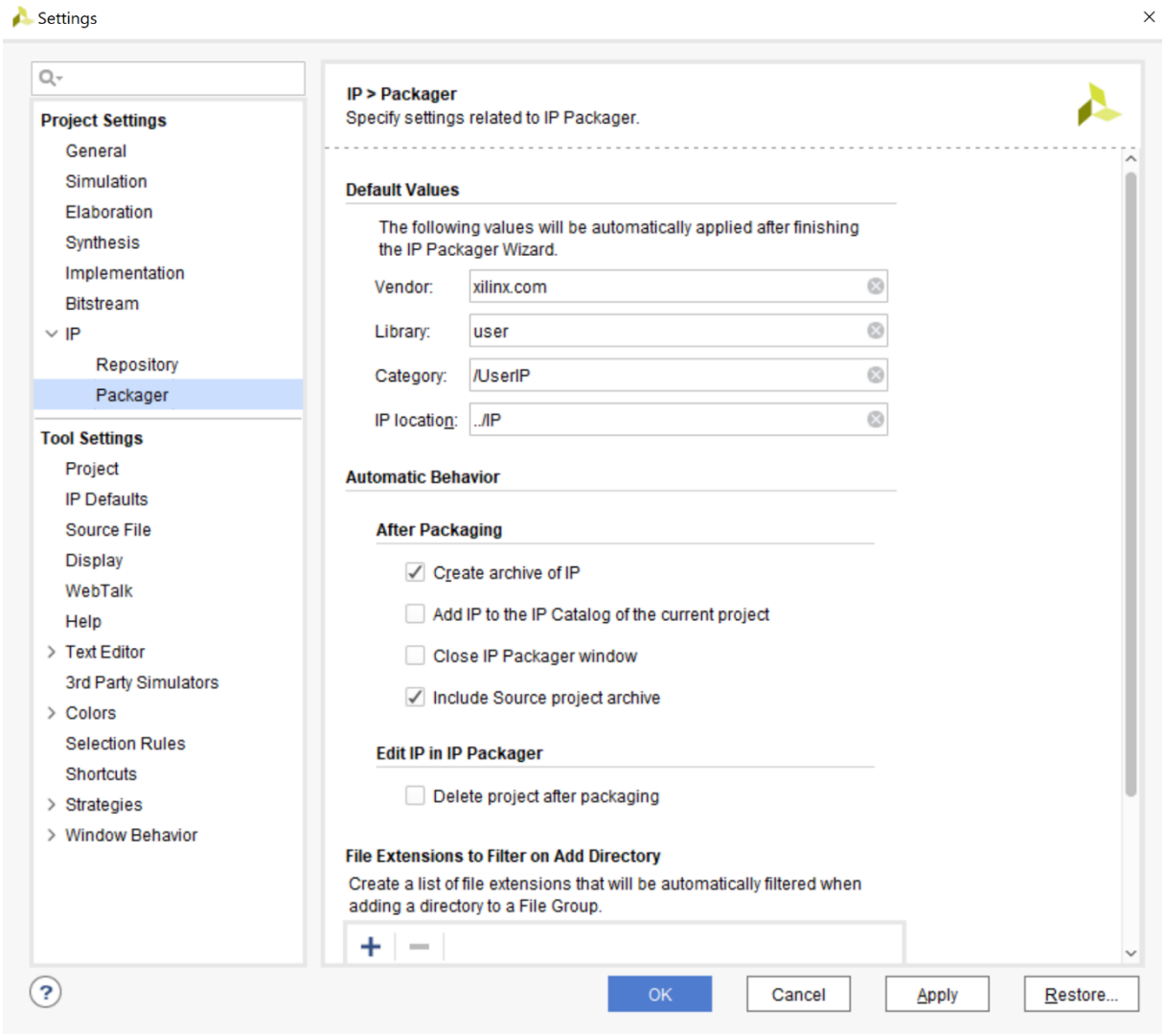
First of all, you will need to have some kind of working hardware description of the function you want to implement as a hardware function. Then you will need to rewrap it into a top module like this one:

```
entity piccolo_80_speed_axi_top is
    Port ( reg0 : in STD_LOGIC_VECTOR (31 downto 0); --data_in msb
           reg1 : in STD_LOGIC_VECTOR (31 downto 0); --data_in lsb
           reg2 : out STD_LOGIC_VECTOR (31 downto 0); --data out msb
           reg3 : out STD_LOGIC_VECTOR (31 downto 0); --data out lsb
           reg4 : in STD_LOGIC_VECTOR (31 downto 0); --encrypt flag + key msb (79 downto 64)
           reg5 : in STD_LOGIC_VECTOR (31 downto 0); --key (63 downto 32)
           reg6 : in STD_LOGIC_VECTOR (31 downto 0);
           clk : in std_logic); -- key lsb (31 downto 0)
end piccolo_80_speed_axi_top;

architecture Behavioral of piccolo_80_speed_axi_top is

component piccol_80_top is
    Port ( data : in STD_LOGIC_VECTOR (63 downto 0);
           key : in STD_LOGIC_VECTOR (79 downto 0);
           encrypt : in STD_LOGIC;
           clk : in std_logic;
           cipher : out STD_LOGIC_VECTOR (63 downto 0));
end component;
```

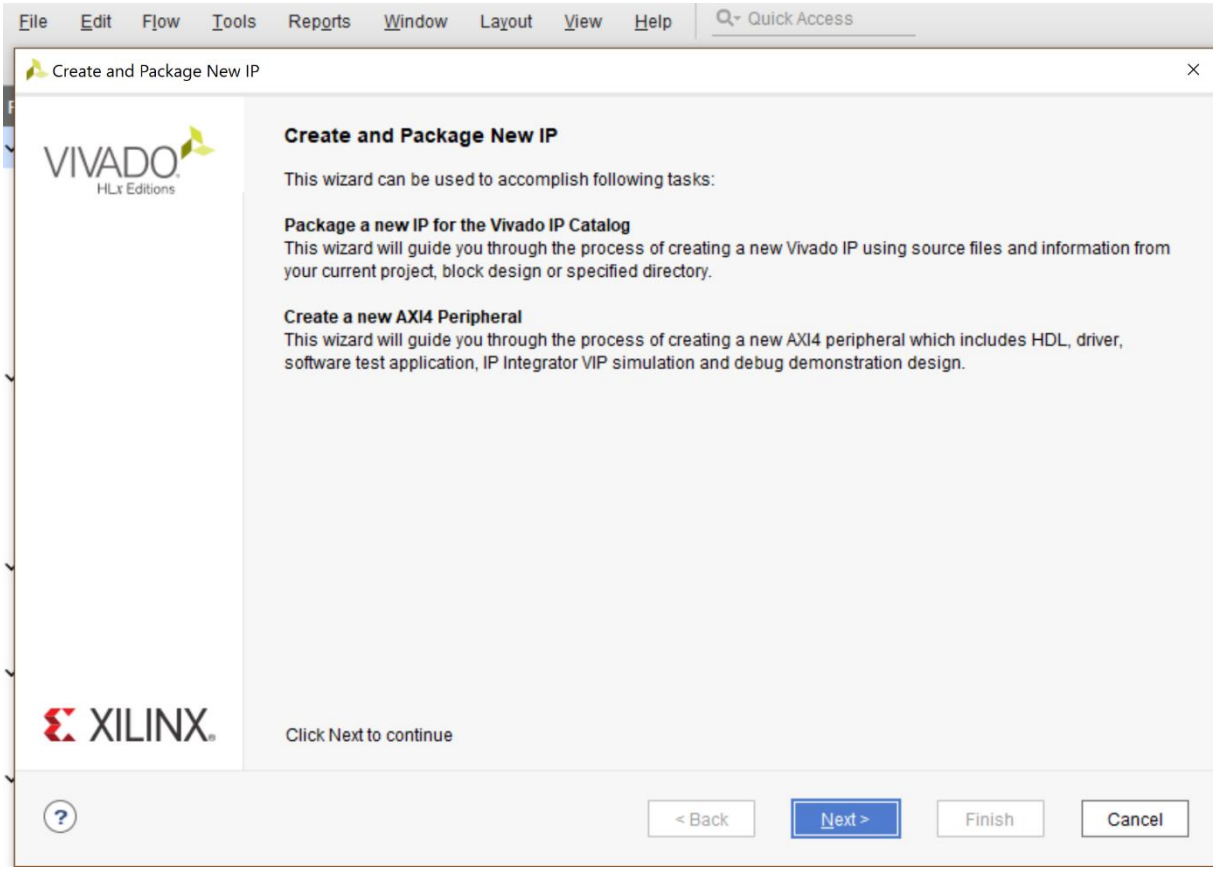
The idea behind the *_axi_top module is to make your function register ready. Because in an AXI slave the master simply has access to mapped register inside the slave. So the basic idea will be to connect these input/output to the registers in the AXI slave. This is the place to make any kind of hardware FIFO or any kind memory. Some of the flag can also be dealt with here depending on your function and mapping. Do not connect input and output in the same register!



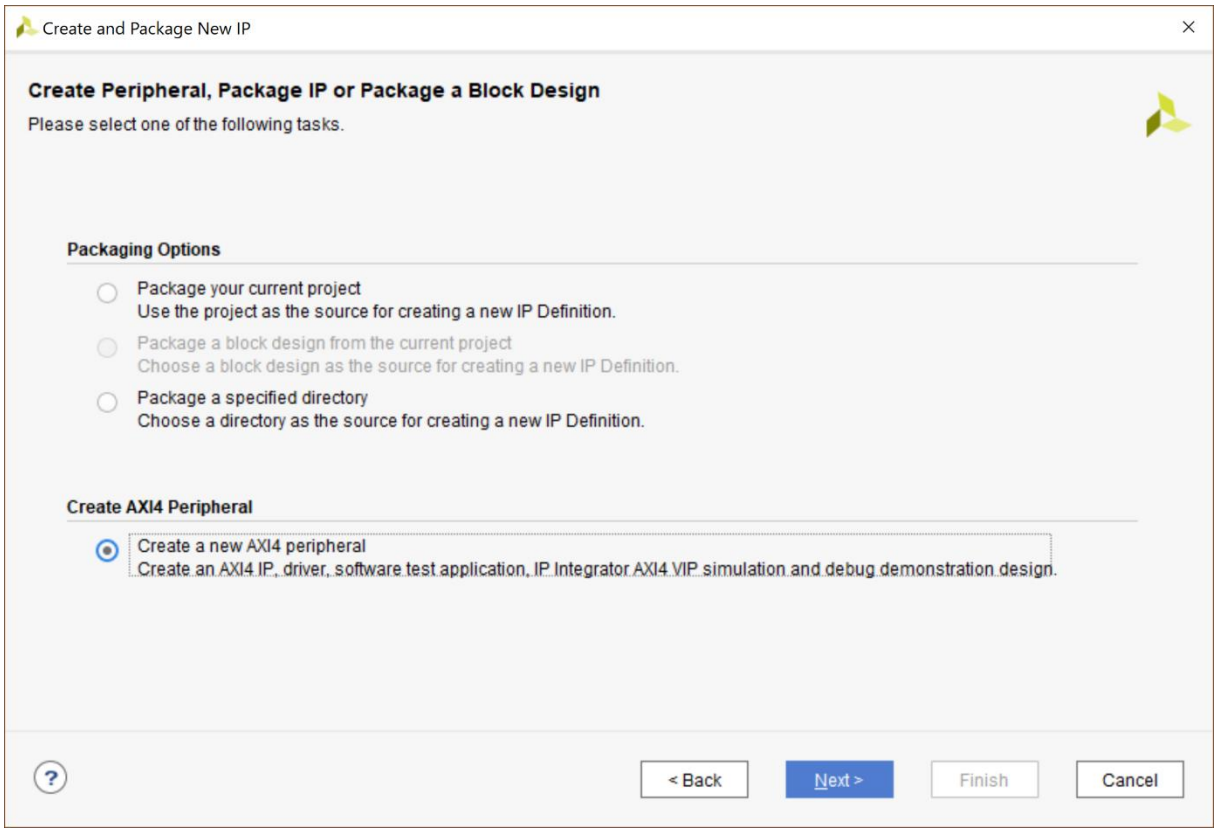
Just check in your project setting if they are coherent. Specially the target language as Vivado will use this language when it will generate HDL code. Which will be useful when we are going to connect our function to the AXI bus which is HDL code generate by Vivado.



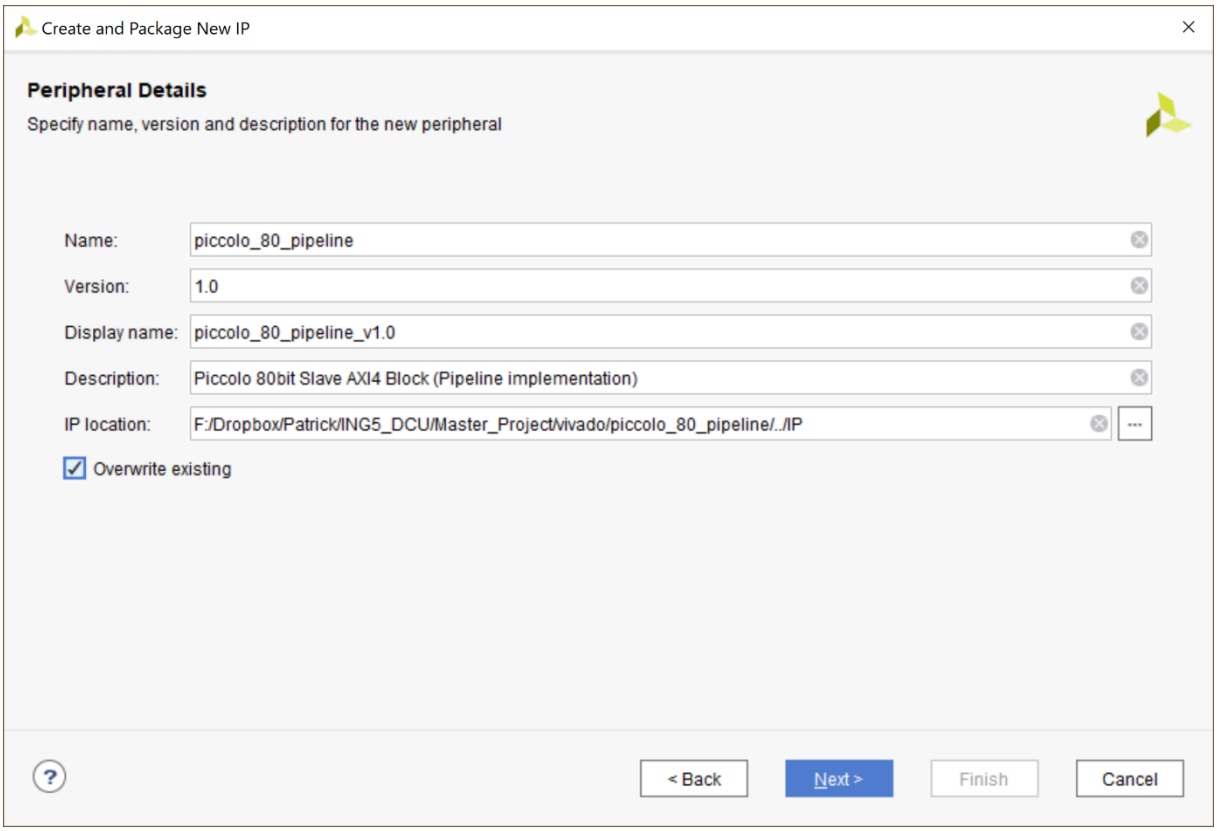
Just check that the info are coherent. You can see on the left part how to access this from the project settings. When you create an IP it doesn't wrap everything in a single folder like it would when you create a project so I can be a good idea to have an empty folder. This is that folder that you need to link in the "IP location".



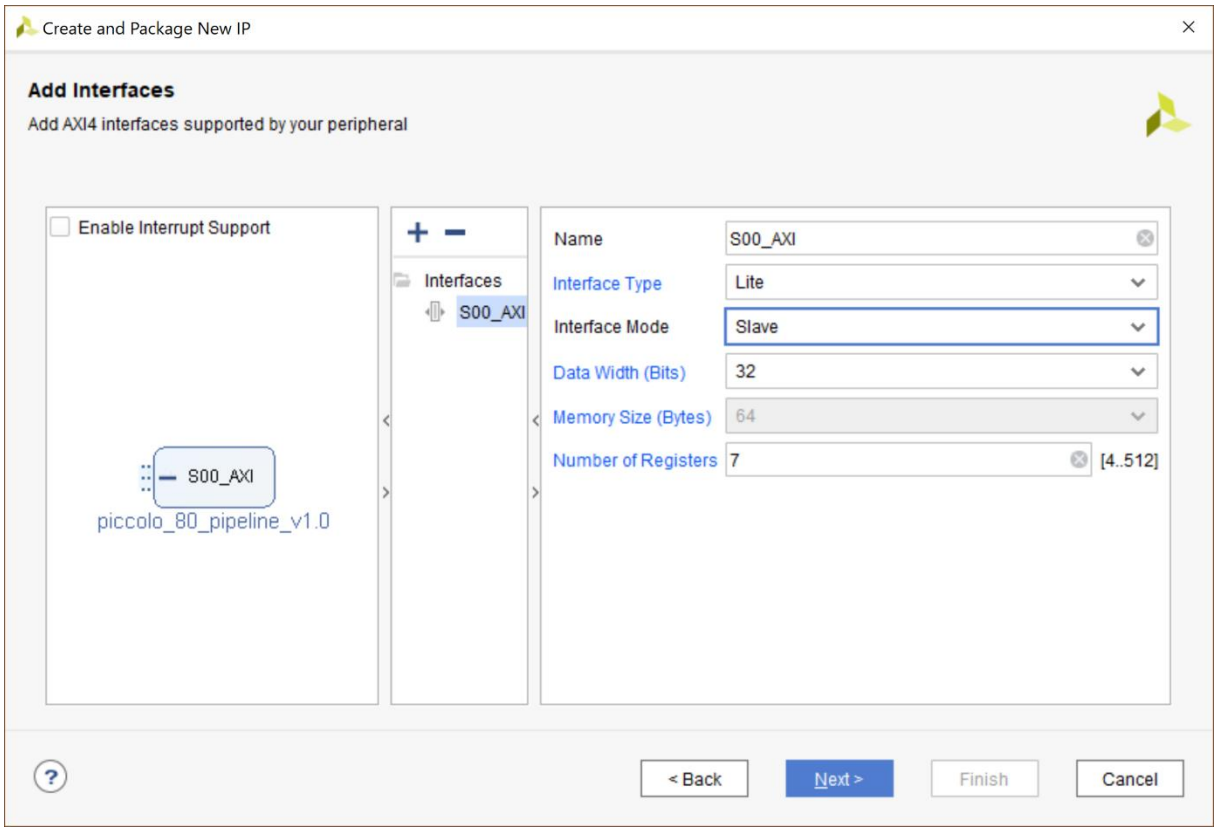
You can access the Create IP Wizard from the menu bar : Tools → Create and Package New IP …



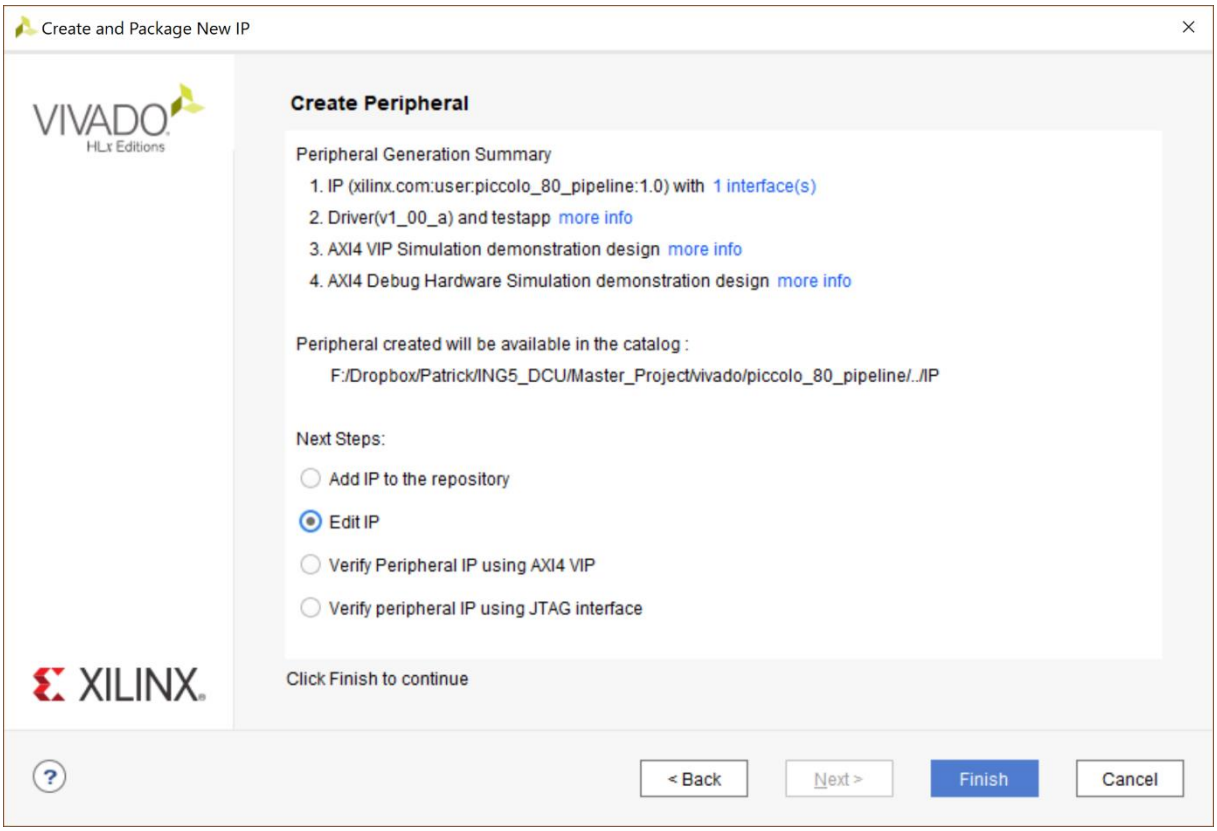
Select the AXI4 Peripheral



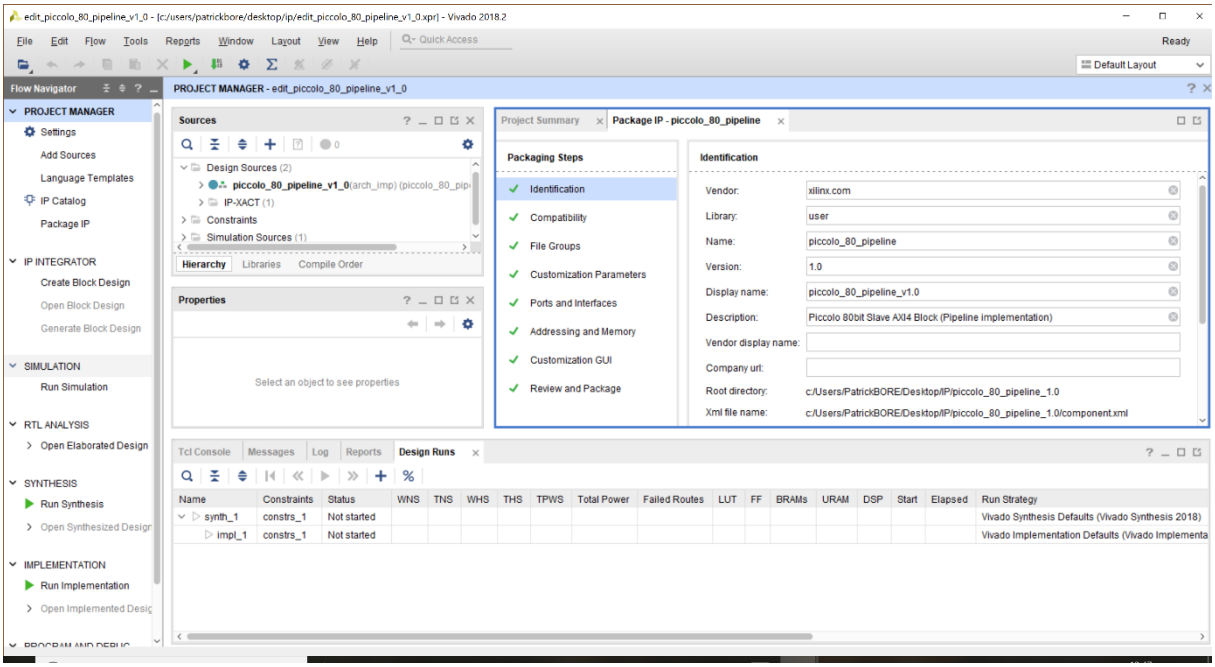
Just fill the fields with the information related to you hardware function.



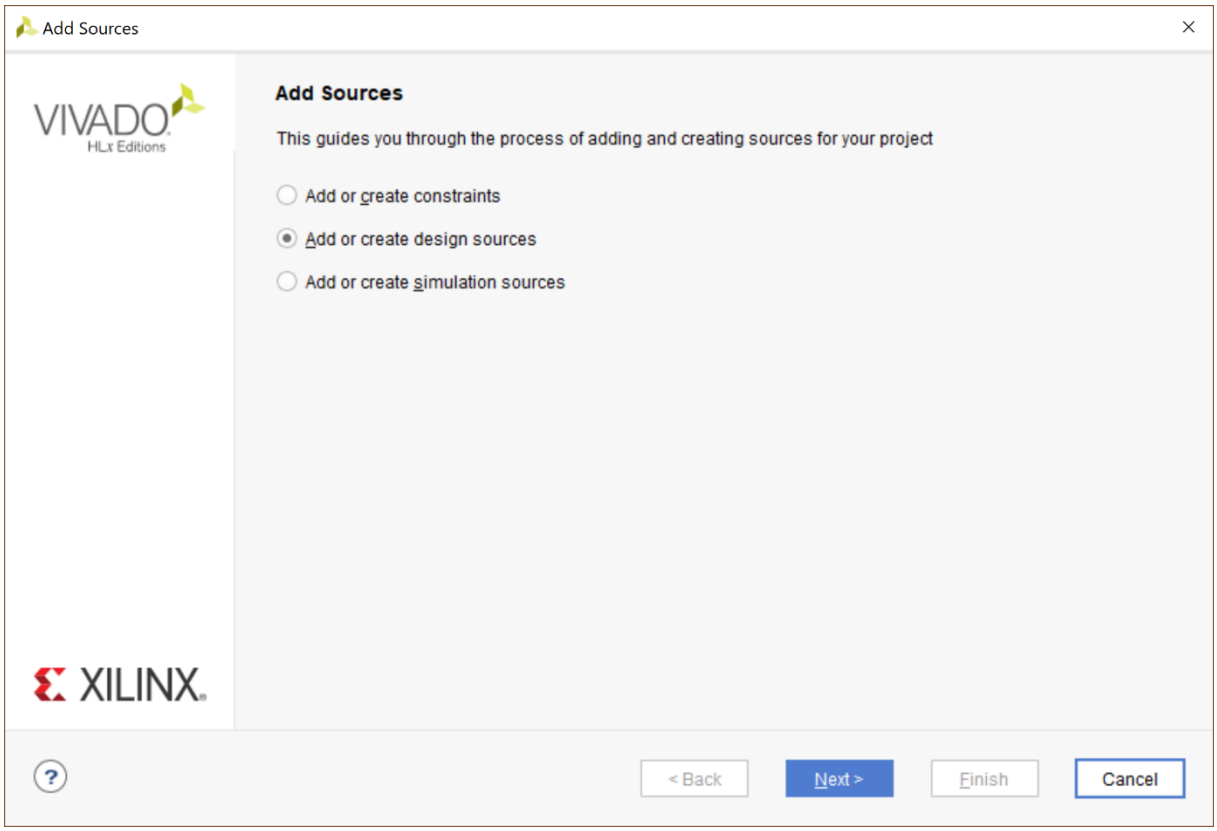
Be careful to put the correct number of registers. As many as you function need!



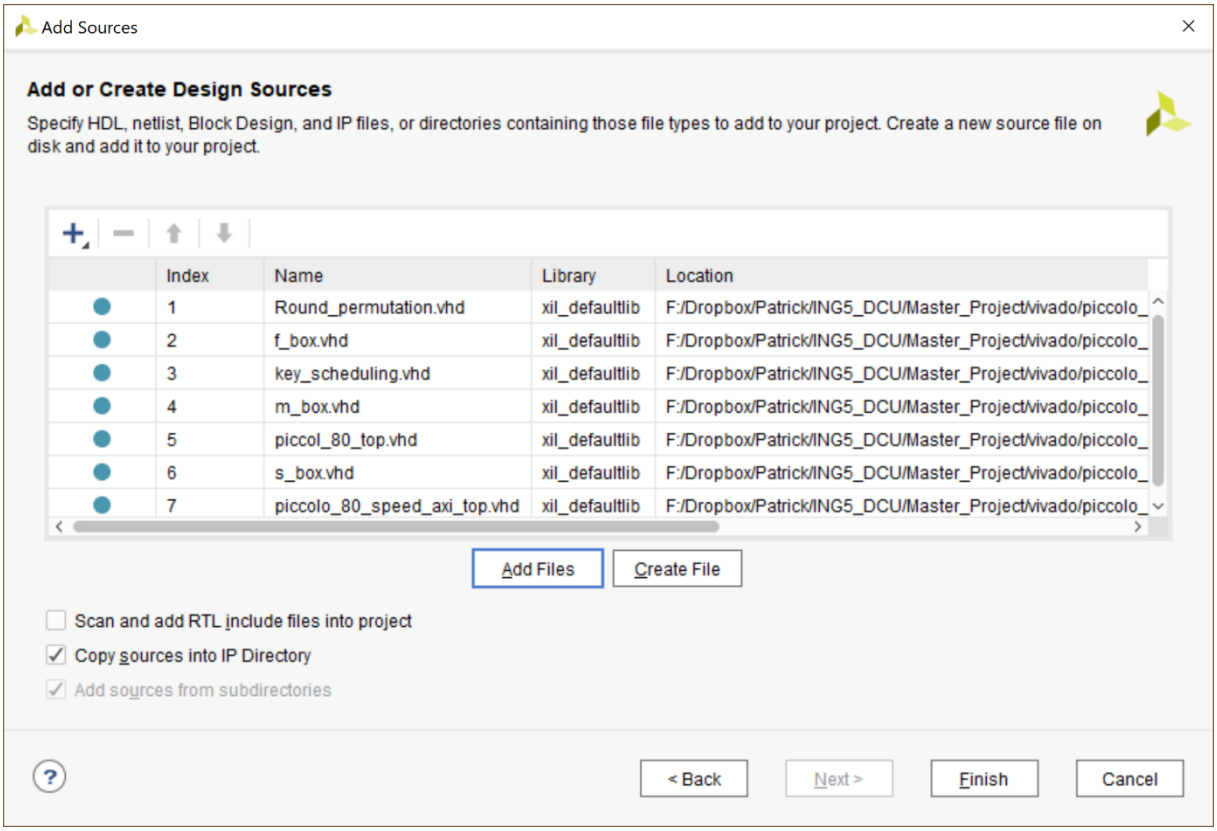
SELECT “Edit IP” and press Finish



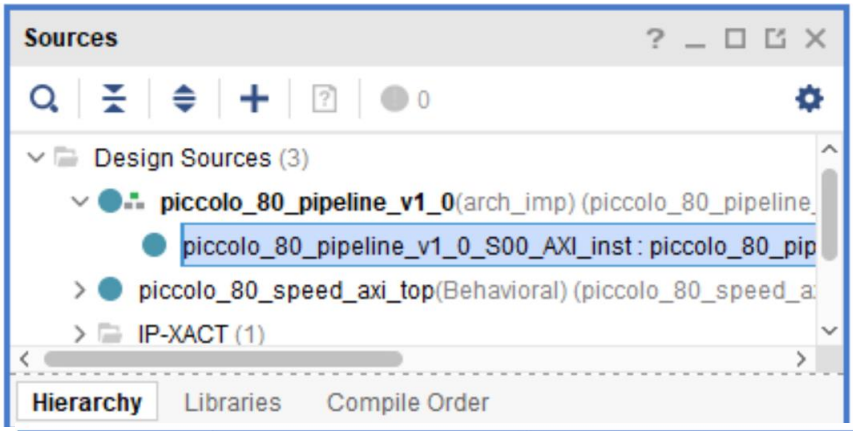
That what the Vivado should look like after few second (max few minutes if your computer is a bit slow). And this open into a new Vivado you still have access to your other project that was open.



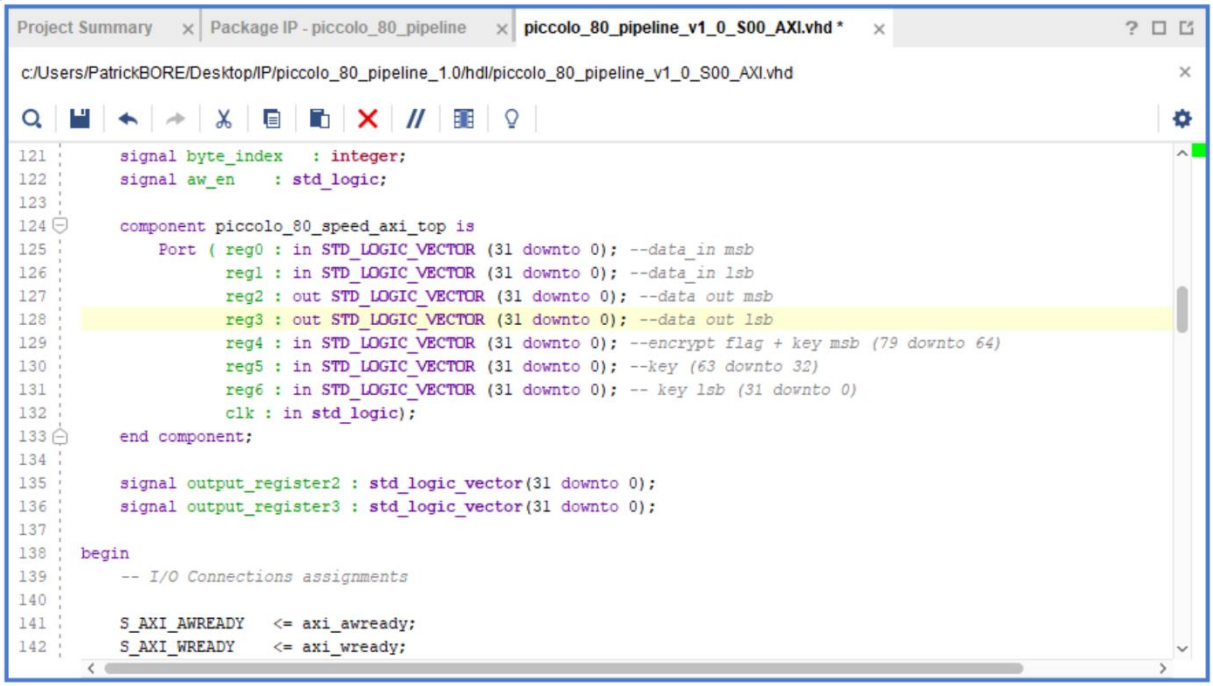
Go to add the sources of your hardware function.



Don’t forget to SELECT “Copy sources into IP Directory”



Once the sources are added your source tree should look like this. Now the idea is to link your *_axi_top to the AXI bus by modifying the select source (the one highlight in blue).



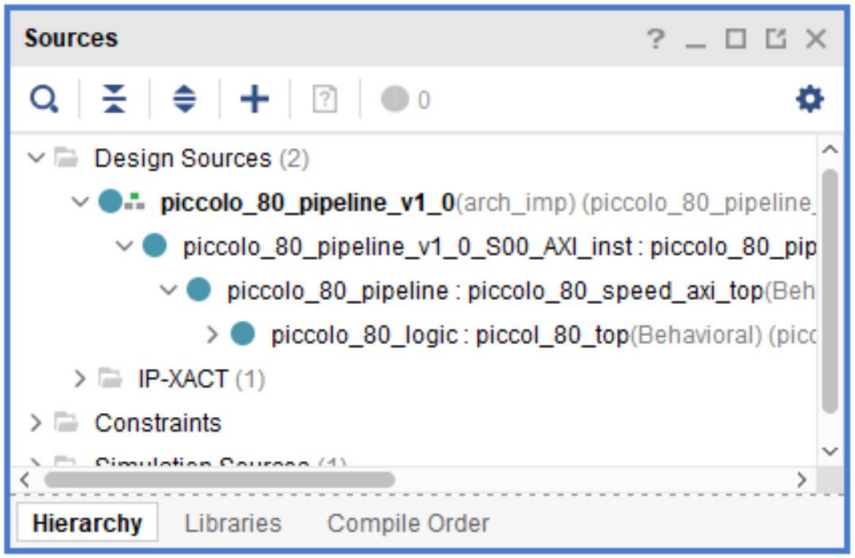
Add the component description just before the “begin” in the previously selected source. You will also need to create a signal for every output register.

```
439     -- Add user logic here
440     piccolo_80_pipeline : piccolo_80_speed_axi_top port map (
441             reg0 => slv_reg0,
442             reg1 => slv_reg1,
443             reg2 => output_register2,
444             reg3 => output_register3,
445             reg4 => slv_reg4,
446             reg5 => slv_reg5,
447             reg6 => slv_reg6,
448             clk => S_AXI_ACLK);
449     -- User logic ends
450
451 end arch_imp;
452
```

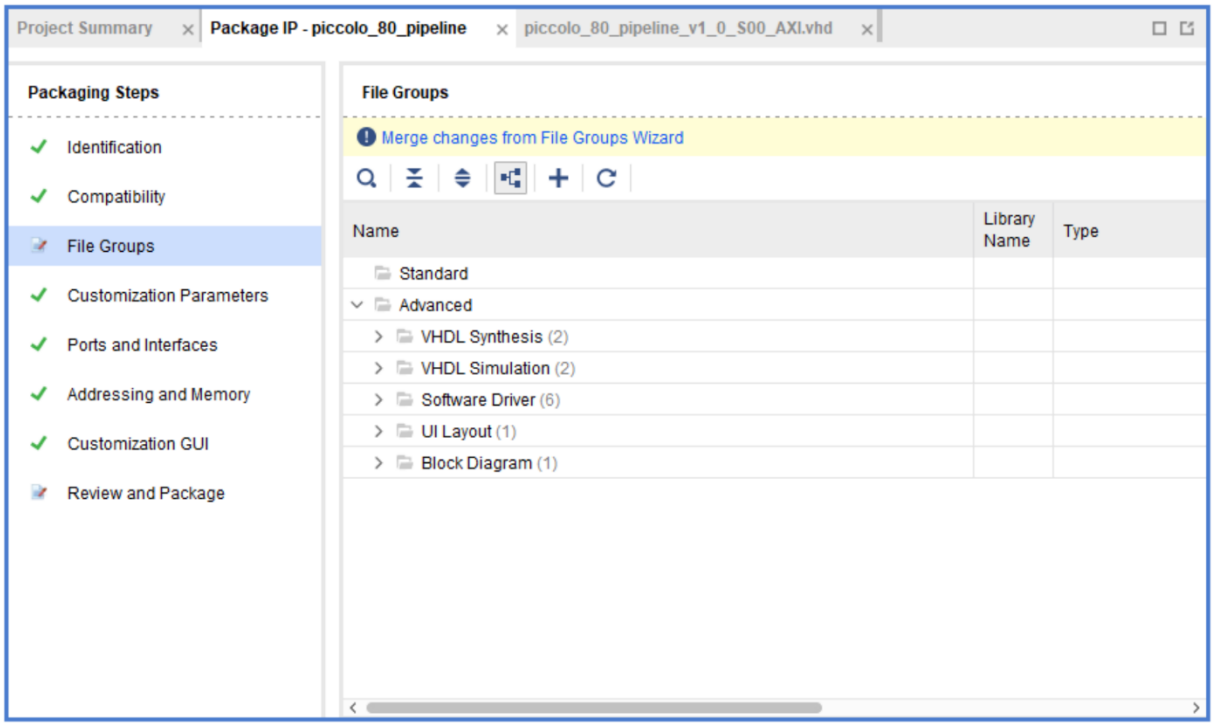
Now you need to map your component to the slave registers and your output registers signals. Double check that the names of the slave register are still coherent with the one I have as they have been generated by Vivado. This is meant to be done at the end of the source file in the user logic section.

```
395     process (slv_reg0, slv_reg1, output_register2, output_register3, slv_reg4, slv_reg5, slv_reg6, axi_araddr, S_A
396     variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
397     begin
398         -- Address decoding for reading registers
399         loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
400         case loc_addr is
401           when b"000" =>
402             reg_data_out <= slv_reg0;
403           when b"001" =>
404             reg_data_out <= slv_reg1;
405           when b"010" =>
406             reg_data_out <= output_register2;
407           when b"011" =>
408             reg_data_out <= output_register3;
409           when b"100" =>
410             reg_data_out <= slv_reg4;
411           when b"101" =>
412             reg_data_out <= slv_reg5;
413           when b"110" =>
414             reg_data_out <= slv_reg6;
415           when others =>
416             reg_data_out  <= (others => '0');
417         end case;
```

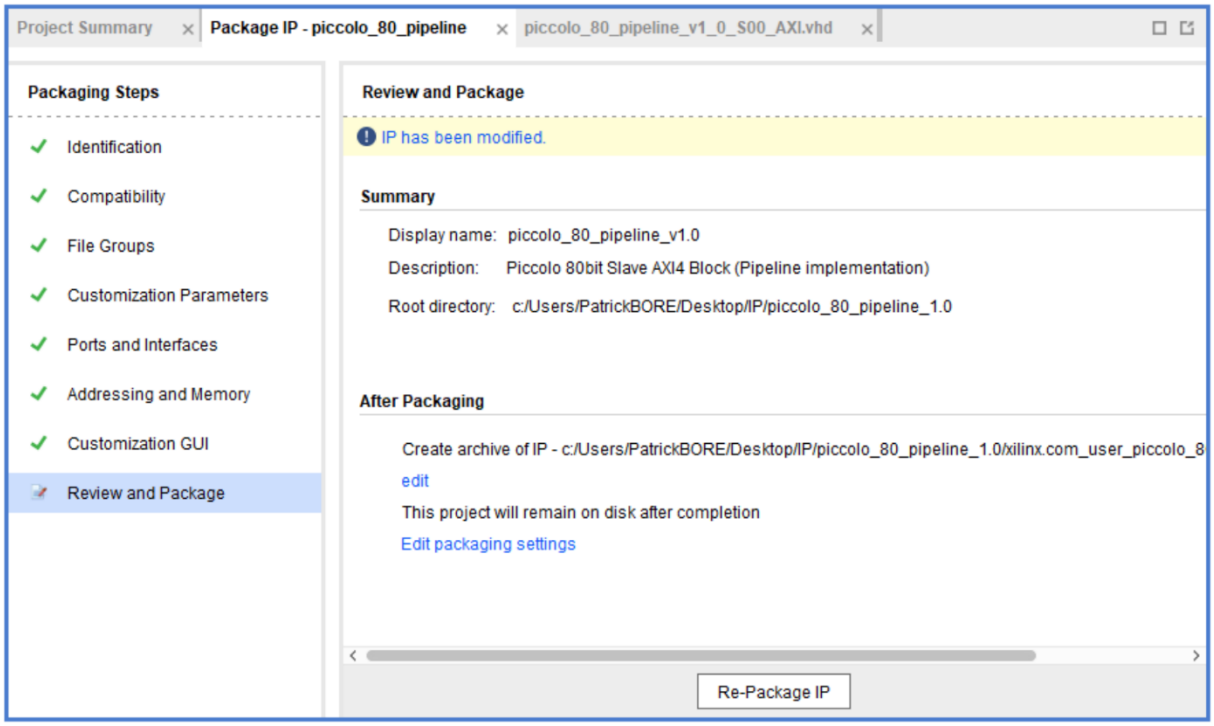
Now you need to go up few lines, so your output register will be output by the AXI bus when you try to read at the corresponding address. You'll need to replace the slv_reg* by output_register* in the switch case inside the process AND inside the sensitivity list of the same process. Now save the modified file.



You should now have a source tree looking like that.



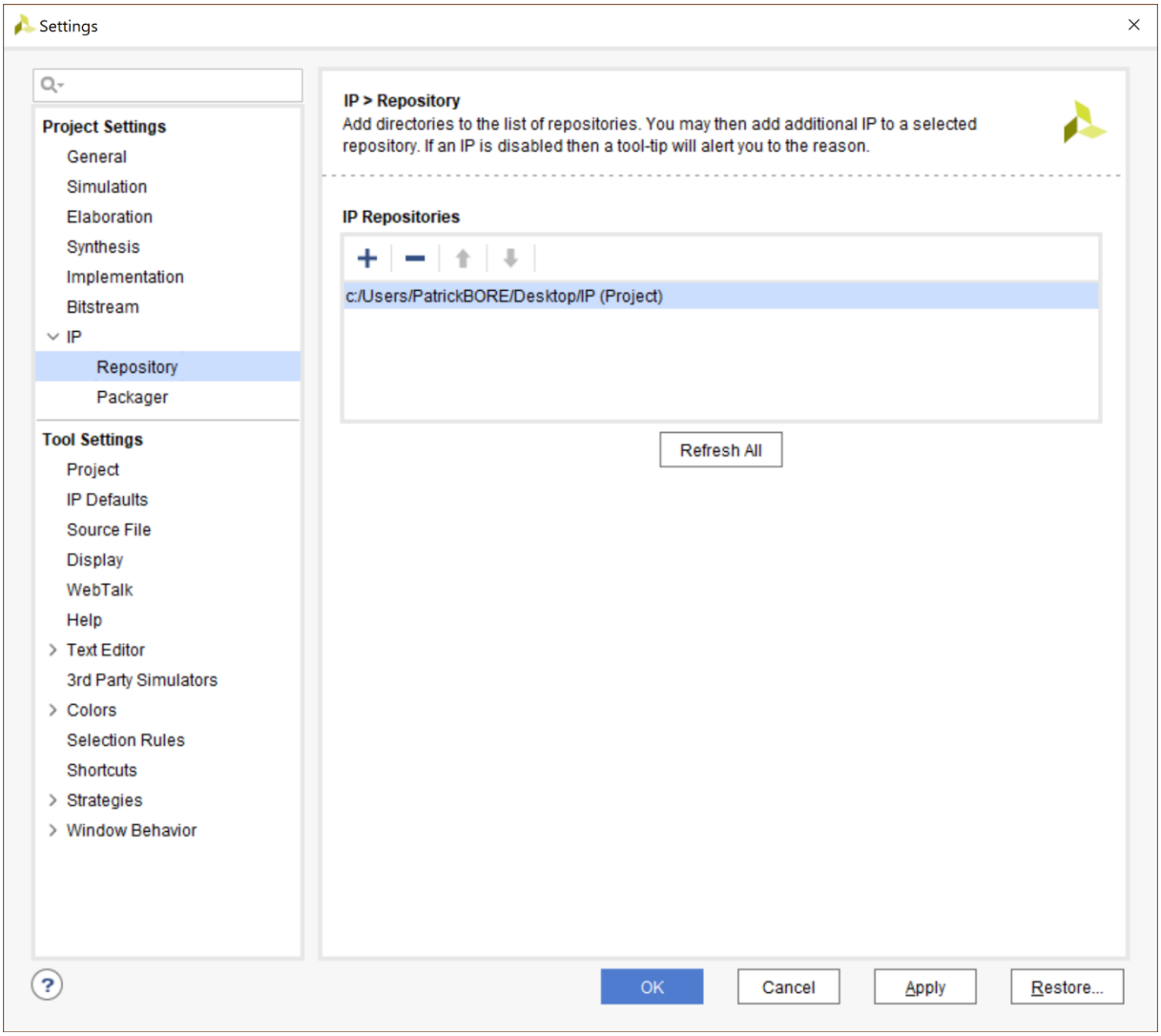
If you go back to the 1 file you arrive on now some of the green check mark are gone. Go on “File Groups” and do the “Merge changes from File Groupe Wizard”.



Now go to “Review and Package”. Click on Re-Package IP”. And here you are you just created an IP of your hardware function with an AXI4 bus slave interface.

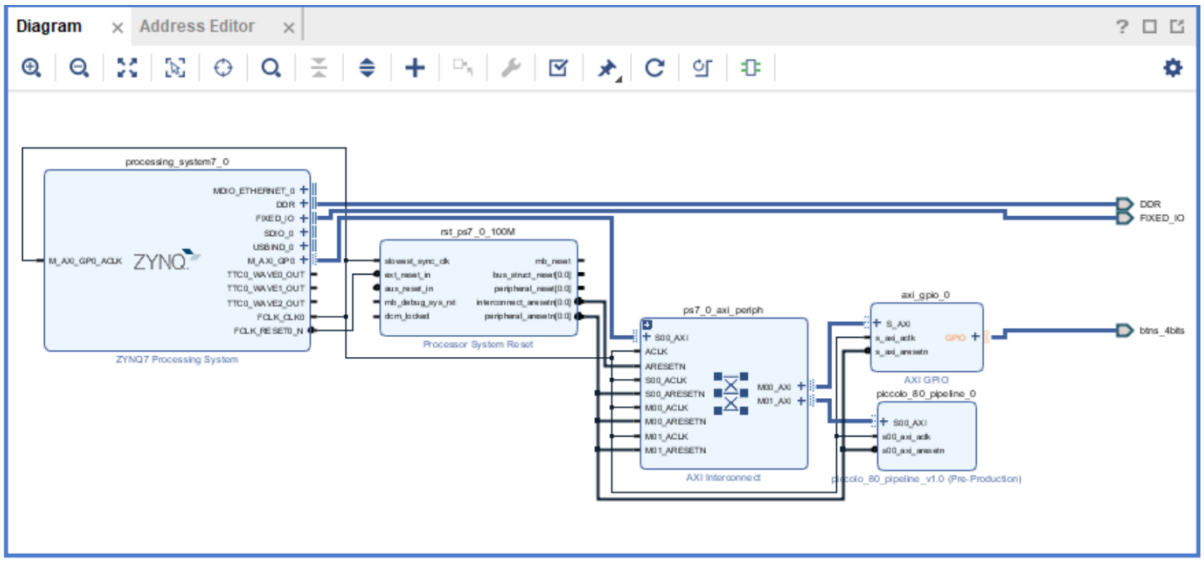
2. Using the IP with the Processor

Now that you have your Slave AXI block with your hardware function it's time to create a block design for the Zynq chip with your IP Block integrated to it.

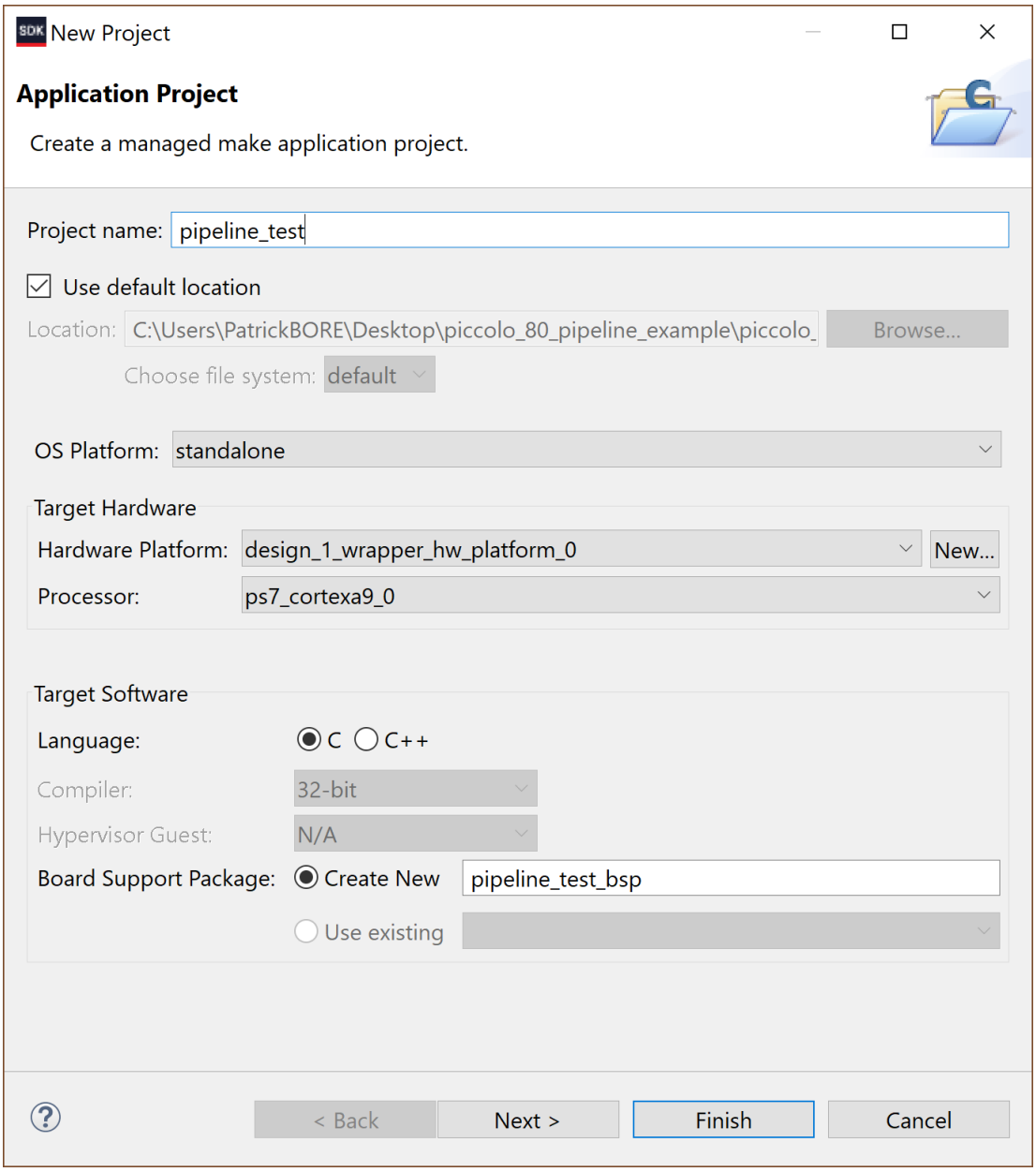


Well 1st of all you need to create a Vivado project and select your board/Chip (Zybo in my case). Then go on your "Project Settings" once again to link the folder with your brand new IP.

Then create a block design and follow the same kind of procedure than in IP Integrator tutorial [12] to create you system. You'll will obviously have to add your new IP to the design. Don't worry the auto-connect will also work on your personal IP. I personally also added the GPIO block for a simple reason: it automatically put all the library when you will go to the SKD. Specially the xil_printf library which is extremely useful during the project debugging part or simply if you want an easy way to get data back from the chip. (Don't forget the HDL Wrapper 😉)



You should have a block design looking a bit like this. Generate you Bitstream and go to the SDK. Leave it the needed time to start and deal with the hardware parameters. Don't forget to export the Hardware and to include the Bitstream before going to the SDK



Create a New Application project in C and select the “Hello World” Template.

Go to helloworld.c and delete everything under #include “xil_printf.h”

You have two examples of code for different implementation of Piccolo in my GitHub. I’m giving you the rough code I made to use inefficiently my pipeline implementation as I didn’t had time to modify the hardware function to rethink it for a software use as explained before in this report.

```
#include "xbasic_types.h"
#include "xparameters.h"

#define PICCOLO_80_PIPELINE_FLAG_ENCRYPT (1<<16)

Xuint64 piccolo_80_pipeline(Xuint32 *hardware, Xuint32 dataMSB,Xuint32 dataLSB,
Xuint32 Flag_keyUpper,Xuint32 keyMiddle, Xuint32 keyLower)
{
```

```
//data in
hardware[0]= dataMSB;
hardware[1]= dataLSB;

//Key + FLAG
hardware[4] = Flag_keyUpper;
hardware[5] = keyMiddle;
hardware[6] = keyLower;

__asm__("nop");
__asm__("nop");

return *(((Xuint64 *)hardware)+1); // return the value of the encrypted data}

int main()
{
init_platform();

Xuint64 val;

xil_printf("Piccolo_80_speed Test\n\r");

xil_printf("Decryption of 0x8D2BFF9935F84056 with the key 0x0011223344556677899\n\r");
val=piccolo_80_pipeline((Xuint32
*)XPAR_PICCOLO_80_PIPELINE_0_S00_AXI_BASEADDR,0x8D2BFF99,0x35F84056,
(~PICCOLO_80_PIPELINE_FLAG_ENCRYPT) & 0x0011,0x22334455,0x66778899);
xil_printf("Decrypted data is: 0x%08x%08x \n\r", val.Upper,val.Lower);

xil_printf("Encryption of 0x0123456789abcdef with the key 0x0011223344556677899\n\r");
val=piccolo_80_pipeline((Xuint32
*)XPAR_PICCOLO_80_PIPELINE_0_S00_AXI_BASEADDR,0x01234567,0x89abcdef,
PICCOLO_80_PIPELINE_FLAG_ENCRYPT | 0x0011,0x22334455,0x66778899);
xil_printf("Encrypted data is: 0x%08x%08x \n\r", val.Upper,val.Lower);

xil_printf("End of test\n\n\r");


   return 0;
}
```

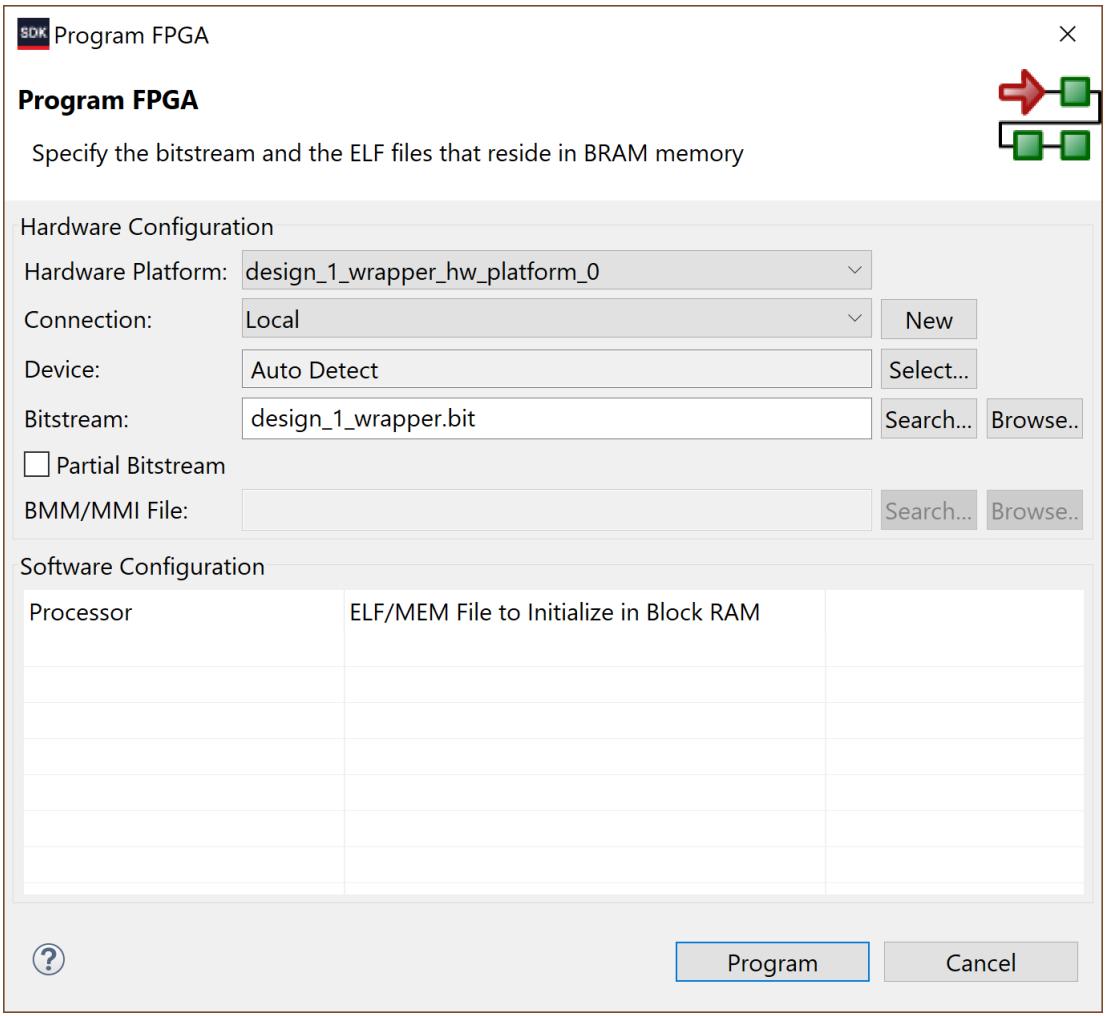
This should give you some direction to make your own C function for your specific Hardware function. Let's review the most interesting parts.

((Xuint32 *)XPAR_PICCOLO_80_PIPELINE_0_S00_AXI_BASEADDR is the define with the address of your registers inside the slave AXI block. You can find your define in xparameters.h but it should be formatted like this one. You can then write and read in the registers like you would do with any normal memory address. To me there is 3 possibilities :

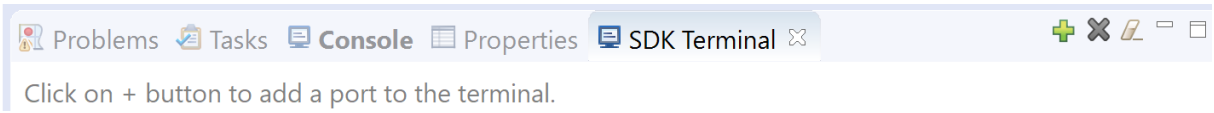
1)
```
#define PICCOLO_80_PIPELINE_mWriteReg(BaseAddress, RegOffset, Data) \
Xil_Out32((BaseAddress) + (RegOffset), (u32)(Data))
#define PICCOLO_80_PIPELINE_mReadReg(BaseAddress, RegOffset) \
 Xil_In32((BaseAddress) + (RegOffset))
```

You can use this premade function (already defined in your IP specific .h in the BSP).
2) You can dereference the pointer and access directly to the memory
3) Or you can use the base address like a fixed size table and access your different register like I did in the small function I wrote

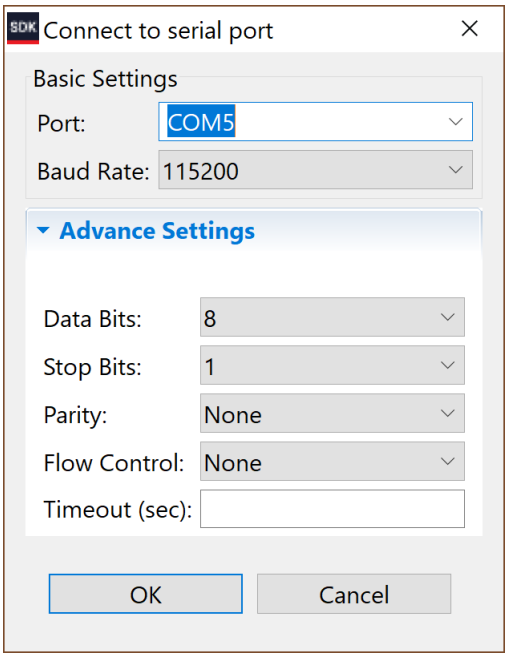
Then save your test code.



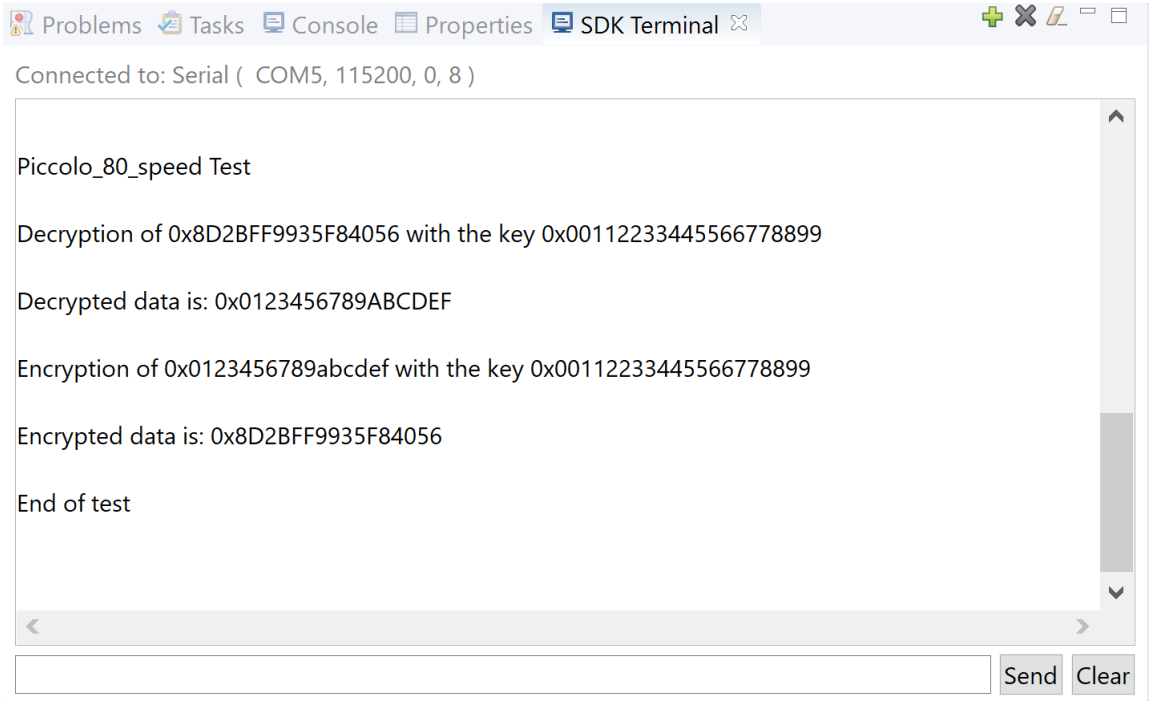
Connect your board and programme the FPGA.



If like me you use xil_printf go to the “SDK Terminal” and click on the green + button to open a serial port to you card. If you are using a Zybo board, you parameter should be the same as mine exception made from the “Port:” which will depend on the port your computer mounted the serial port for the board.



If like me you use xil_printf go to the "SDK Terminal" and click on the green + button to open a serial port to you card. If you are using a Zybo board you parameter should be the same as mine exception made from the "Port:" which will depend on the port your computer mounted the serial port for the board.



Now run you C project and you should have the xil_printf arriving on your serial port connection. Here you are if everything went as it should and your hardware function is working properly you now have a hardware coded function accessible from your C code aimed for the ARM Processor of the SoC.

B. Co-Design flow

In this part I'm going to give you an idea of my current thinking of co-design between logic and processor on specific SoC like the Zynq chip.

We are going to suppose you have some kind of project that could be implemented in software but for some reason you might want to make it faster. If you have access to an SoC that have ARM processor and Programable logic or the mean to make your own chip combining some kind of processor and logic. You know that logic is faster as it's specially design for the application, but it's is also expensive to design, and you will probably be more limited in memory and size of your application. So, one idea would be to use a combination of these 2 solutions: classical processor application and hardware application. That's what is reference in this document as Co-design.

The 1st thing your need to realize is that breaking a project between hardware and software will not automatically make it faster if it's done wrong, I would even say it will be more than likely slower if it's not perfectly implemented.

The 1st problem is the communication between the two entities. You will have some kind of bus to discuss between the 2 different systems and this will create latency, a lot of it. So, your 1st concern will be to make sure that the time won by implanting in hardware won't be lost by the time needed to get your data through the system.

Second thing you need to take care of is to know the strong part of each system. For example, because most processor have hardware function for floating point arithmetic your processor should be attributed the part with the most floating points. The pros of the processor include but does not limit to: external communication (ethernet/network) most likely threw linux, large memory management, file management, etc… In the other side logic is really efficient for hard algorithm with a lot of small steps and multi-channel (parallel) algorithm. The perfect example would be image processing, it usually includes so kind of multi-channel thinking to implement it in multiple pixel at once for example and image processing is usually just a long succession of small step like addition, subtraction, etc…

And the last point to take care of is the size of each of the subdivision. When you do co-design you are tempted to break it into too many part, that will slow the project a lot due to the communication issue like stated earlier so when you make a subdivision don't forget to also take into account the lost time and the best way I found to make sure that the time lost from the communication isn't too high in front of the win is to keep the subdivision really big so you will optimise the data transfer. For example, if you do some kind of data processing on a 64-bit data packet if the processing is split too much you will send the same data but just at different state back and forward again and again. So, keep your subdivision block size coherent and the division need to be coherent for the data too!

In my case I had the Piccolo-80 algorithm. It applies on a 64 Bits block and have an 80 bits key. Furthermore, the Piccolo algorithm is lightweight meaning no complex operation, in fact there only is lookup table and Xor gates… So, it would have break common sense to break piccolo into multiple part because the time needed to send and receive the needed data is too large in front of the combinatory time. So, I went into a different approach and went to make what I call a Hardware Function. It's basically like a normal software function but instead of running the algorithm it only calls the hardware implementation of the algorithm. On the software part once, the function is implemented the use just look like the use of any function. Exception made of the fact you need a hardware pointer but once your final hardware system is designed you can abstract that thanks to the generated define that give you the needed address. I personally use this function on simple test, but it could be use into more complex application. And as mention before hopefully this function will be usable through Linux with a bit of effort.

V. Sources and GitHub

Link : https://github.com/PatrickBORE/masterProject.git

All the useful source code of my project are at that link. This will prevent the use of paper to print the code in the annex … So if you want to read my code they are in the git.

There is the VHDL code for the different implementations in Piccolo/ Piccolo_80_VHDL_source. All the code is commented and I'm going to give a bit more information in this part.

I also made some IP of the implementation that are interfaced with the AXI bus. They are design for the Zybo. I also included an example project that use these IPs. I will give a bit more information about it in this part too.

A. VHDL Sources

In all my implementation I added a bit to know if the implementation is supposed to encrypt the data or decrypt it. This have repercussion for the comparison of my algorithm size to others but you need to keep in mind that my implementation are working implementation so design choice away from the pure theory implementation had to be made.

1. Speed Implementation

The idea behind this implementation was to forget the size problem and just get the maximum speed by just unfolding everything so the encryption would be done in one go. Because the Piccolo-80 algorithm is based on a 25 rounds system what I basically did is implement 25 versions of one round and just connected all of this together. It results of the algorithm being clock less as there only is combinatory logic. That doesn't mean it find the cipher instantly it just mean that it is only limited by the logic and net latency. Vivado gave me a latency of 103.079 ns. Meaning you could encrypt a block at a frequency of 9.7MHz.

2. Pipeline Implementation

The idea behind the pipeline is to have a higher data rate. By using the speed implementation and adding register between steps. This allow to have less logic at once (roughly $1/25^{th}$) so less latency due to the 1 go idea of the speed implementation. This implies that you will have to wait 25 clock cycles before getting your first cipher out. But this is a small step back because of the memory between the steps you can start encrypting a new block on the 2 clock cycle so you will have to wait for 25 clock cycle for the 1^{st} block but the next one will follow on the next clock cycle basically make the 25 clock latency inexistent as long as you encrypt a large number of packets. The main limitation of my implementation is the need of keeping the same key threw the block, if you want to change the key you will need to wait for all the block to be out of the encryption system so you basically come back to the beginning and will need to wait the 25 clock cycles again for the next block. But when your need to encrypt large size data you usually use the same key for the full document so that is not a problem for me. But it can be fixed by making a new implementation with dynamic key along the round, but it will increase the size needed for the algorithm largely so 50 clock cycles lost when you need to change key seems a good trade off.

You will not find the pipeline implementation as to me it's not finished as I didn't optimize it to be use from the processor part yet. The version I have at the minute is simply the speed implementation with minor modification that you can do if you want somewhere to start:

- Add a clock to the input of the piccol_80_speed component.
- On every line where I assign a value to X(i) you will have to make it a memory.
  So just add a when rising_edge(clk) to all of them.
- Do the same as you did for the X(i) but for the cipher output on line 383

And here you go you have a pipeline implementation.

3. Resources Implementation

The resource implementation is more axed around using less logic. I didn't optimise the logic too much as I still wanted to keep a decent data rate in front of the other implementation. For example, I didn't use the proposed optimisation given in the paper [1] because I find it too slow. This version looks more like a for loop that repeat itself 25 times. I made the round keys generated on a for loop to optimise also that part of the algorithm and not have a massive multiplexer with all the round keys waiting to be used.

The main speed restriction here is the need of 25 clock cycle to encrypt one block. And you'll have to wait as long for the 2nd block. As the logic is latency is reduced in front of the speed implementation.

B. IPs

For testing purposed I implemented IP of 2 of the implementations (didn't do the pipeline for previously stated reason). This IP are AXI4 slave to be able to communicate with the processing system. So, the way to use them is through mapped registers (through the AXI bus). Each IP will have a personal base address which is the address of the 1st register. In this part I'm going to explain the eventual modification made to the implementation to make it software use ready and give the detailed information about the memory mapping inside the block (the function of each register)

1. Speed Implementation

Because there isn't much timing involve in this implementation the IP is simply a mapping of the input/output to the registers.

- Reg0: MSB of the input block/data (bits 63 downto 32)
- Reg1: LSB of the input block/data (bits 31 downto 0)
- Reg2: MSB of the output block/cipher (bits 63 downto 32)
- Reg3: LSB of the output block/ cipher (bits 31 downto 0)
- Reg4: Flag + Key MSB
  - Reg4: bits 31 downto 17 → Not Connected (N/C)
  - Reg4: bit 16 → Encrypt flag: '1' encrypt and '0' decrypt
  - Reg4: bits 15 downto 0 → key MSB (bits 79 downto 64)
- Reg5: middle bits of the Key (bits 63 downto 32)
- Reg6: LSB of the Key (bits 31 downto 0)

s

2. Resources Implementation

Because the AXI bus is not a parallel but I had to implement data synchronisation as the different part of the data will be set at different timing. I also had to implement a flag to signify that the cipher is ready to be read and to keep it to start a new encryption. The synchronisation is done by the dataReady flag which in practice is mainly a reset high and the output flag is cipherReady which is at low when the data is ready.

To have an example on how to use it the easiest would be to go into the small vivado project I did the function for the resource IP take full advantage of the flags.

To avoid confusion, I mapped the register to the piccolo algorithm mainly the same way but be careful you do need a specific function for each different implementation!

- Reg0: MSB of the input block/data (bits 63 downto 32)
- Reg1: LSB of the input block/data (bits 31 downto 0)
- Reg2: MSB of the output block/cipher (bits 63 downto 32)
- Reg3: LSB of the output block/ cipher (bits 31 downto 0)
- Reg4: Input Flags + Key MSB
  - Reg4: bits 31 downto 18 → Not Connected (N/C)
  - Reg4: bit 17 → DataReady flag: '1' → data not ready/ reset and '0' → data ready
  - Reg4: bit 16 → Encrypt flag: '1' encrypt and '0' decrypt
  - Reg4: bits 15 downto 0 → key MSB (bits 79 downto 64)
- Reg5: middle bits of the Key (bits 63 downto 32)
- Reg6: LSB of the Key (bits 31 downto 0)
- Reg7: Output Flag
  - Reg7: bits 31 downto 1→ Not Connected (N/C)
  - Reg7: bit 0 → cypherReady flag : '1'→cypher not ready and '0'→ cypher ready

C. How to use them

I created a small project that use the 2 IP I created. You can find it in Piccolo/IP/. It's in the form of o zip file due to the high number of different files when you use Vivado. This project has been done with Vivado 2018.2. If you did the previous tutorial you shouldn't have much trouble to use the project. Few noticeable point:

- The repository for the IP isn't dynamic so it would be a good idea to change it to your personal file tree. The premade IP are also in the same zip in ./ressources and ./speed folders.
- The project has been made for a Zybo board.
- The bitstream is already created and the hardware already exported so if you just want you can go straight to the SDK to look at the code or to run the code if you have your own Zybo.

VI. Results

In most papers you can see different implementations compared with Gate Equivalent (GE) but this doesn't apply to FPGA it's mainly aimed at ASIC. Because in a FPGA the chip itself will change the size of the final algorithm on the chip. So, to compare the different implementations I just used the different logic block needed to map each algorithm in the Zybo. All the value given here are either straight from Vivado or calculated from value from Vivado.

A. Comparative Table

Implementations	**Size (LUTs)** Registers Muxes	**Max Speed (MHz)**	**Throughput (Mbits/s)**	**Power (mW)**	**Energy (J/bits)**
Speed	5916 212 32	9.7	620.8	1	1.61×10^{-15}
Pipeline	2602 1600 0	153	9792	8	0.82×10^{-15}
Resources	269 215 32	174.62	429.8	11	25.6×10^{-15}

The Size, Power and Max Speed are real value (applicable in the real world) the other are calculated from these values. So the 9 Gbits/s for example would be hard to achieve and definitely unachievable with the AXI bus.

B. Discussion

These values are coherent with the goals I had for every implementation.

In hindsight in front of this numbers the speed implementation is actually not that impressive specially compared to the resources version. The throughput which is probably the best way to compare them is really close between them and the difference in size is definitely worth it. And to be use on the Zybo this is definitely the best one as the AXI bus will never follow the pipeline implementation (at least not the 'slow' version of the AXI bus) as it already has trouble to follow the speed version. The cipher ready flag is rarely (to not say never) used, it shows that this implementation is fast enough for our usage.

But it worth to dig a bit more in the pipeline version as I already proposed multiple times because with this kind of speed it definitely could be interested, it could be use on every side of a descent speed bus without making it slower, so it could definitely make the chosen bus more secure to outside attacks.

VII. Conclusion

This project taught me a lot about the problem around Co-design. It made me for the 1st time do multiple implementation of the same algorithm for research purposes. But most importantly it made me improve a lot on my technical knowledge around FPGA bare-metal processor, Cryptography and the design tools around Xilinx Hardware.

VIII. References

[1] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda,Toru Akishita, and Taizo Shirai ,"Piccolo: An Ultra-Lightweight Blockcipher", CHES 2011, pp. 342-357, 2011.
[2] Marine Minier, "On the Security of Piccolo Lightweight Block Cipher against Related-Key Impossible Differentials", Progress in Cryptology - INDOCRYPT 2013, pp. 308-318, December 2013.
[3] Seyyed Arash Azimi, Zahra Ahmadian, Javad Mohajeri, Mohammad Reza Aref, "Impossible Differential Cryptanalysis of Piccolo Lightweight Block Cipher", ISCISC 2014 11th, Sept. 2014.
[4] KhanAcademy, "Journey into cryptography", https://www.khanacademy.org/computing/computer-science/cryptography.
[5] Contel Dradford, "5 Common Encryption Algorithms and the Unbreakables of the Future", https://www.storagecraft.com/blog/5-common-encryption-algorithms/, July 2014.
[6] Ssh Communication security, "CRYPTOGRAPHY FOR PRACTITIONERS", https://www.ssh.com/cryptography/, August 2017.
[9] Digilent, "ZYBO Reference Manual", https://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPZYBO/documentation/ZYBO_RM_B_V6.pdf, February 2014.
[10] Digilent, "Installing Vivado and Digilent Board Files", https://reference.digilentinc.com/vivado/installing-vivado/start.
[11] Digilent, "Getting Started with Vivado", https://reference.digilentinc.com/vivado/getting_started/start.
[12]Digilent, "Getting Started with the Vivado IP Integrator", https://reference.digilentinc.com/vivado/getting-started-with-ipi/start.
[13]Digilent, "Creating a Custom IP core using the IP Integrator", https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-creating-custom-ip-cores/start.
[14] Xilinx, "All Programmable SoC with Hardware and Software Programmability", https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#productTable.
[15] Ryad Benadjila, Jian Guo, Victor Lomné, Thomas Peyri, "lightweight-crypto-lib", https://github.com/rb-anssi/lightweight-crypto-lib, February 2017.
[16] Louise H. Crockett Ross A. Elliot Martin A. Enderwitz Robert W. Stewart, "The Zynq Book", Xilinx, July 2014.
[17] Zi Reviews Tech, "Why do some phone SoCs (System on a Chip) suck?", https://www.youtube.com/watch?v=a7Sir9xVETo, Youtube, January 2018.
[18] Xiaojun Wang, "EE540 - HDL and High Level Logic Synthesis", Loop, 2017.
[19] Wikipedia, "Field Programmable Gate Array", https://en.wikipedia.org/wiki/Field-programmable_gate_array
[20] ARM, "Efficient Application Processors for Every Level of Performance", https://www.arm.com/products/processors/cortex-a
[21] Dennis M. Ritchie, "The Development of the C Language", Bell Labs / Second History of Programming Languages conference, Cambridge, April 1993.
[22] David Kahn, "The Codebreakers", Signet Book, 1973.
[23] Eugen Antal, Pavol Zajac, "Key Space and Period of Fialka M-125 Cipher Machine", Cryptologia 39:2, pages 126-144, 2015.
[24] Eli Biham, Adi Shamir, "Differential Cryptanalysis of Data Encryption Standard", Springer Verlag.

[25] Yogesh Kumar, Rajiv Munjal, Harsh Sharma, "Comparison of Symmetric and Asymmetric Cryptography with Existing Vulnerabilities and Countermeasures", IJCSMS International Journal of Computer Science and Management Studies, Vol. 11, Issue 03, Oct 2011.
[26] Aidan O' Boyle, "Real-Time Audio Filtering on Raspberry Pi Literature Review", DCU Master of Engineering, January 2016.
[27] Subariah Ibrahim and Mohd Aizaini Maarof, "Diffusion Analysis of a Scalable Fiestel Network", World Academy of Science, Engineering and Technology International Journal of Computer, Electrical, Automation, Control and Information Engineering Vol:1, No:5, 2007.
[28] Preshing on Programming, "How to Build a GCC Cross-Compiler", http://preshing.com/20141119/how-to-build-a-gcc-cross-compiler/, November 2014.
[29] GCC, "GCC, the GNU Compiler Collection", http://gcc.gnu.org/
[30] Proxacutor, "Les FPGA", http://proxacutor.free.fr/.
[31] Courtois N.T., Pieprzyk J. (2002) Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In: Zheng Y. (eds) Advances in Cryptology — ASIACRYPT 2002. ASIACRYPT 2002. Lecture Notes in Computer Science, vol 2501. Springer, Berlin, Heidelberg.
[18] ARM, "ARM Cortex-A Series Programmer's Guide", January 2014.
[32] ARM, "Cortex-A9 | Technical Reference Manual", June 2012.
[33] ARM Developer, "Cortex-A9", https://developer.arm.com/products/processors/cortex-a/cortex-a9.
[34] Patrick BORE, "Piccolo-80 Sources Code", https://github.com/PatrickBORE/masterProject.git, GitHub,
31 August 2018.