FPGA-based Accelerators for Cryptography

Isak Edo

Kogens Lyngby 2018

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 324
2800 Kongens Lyngby, Denmark
Phone +45 45 25 30 31
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

Cryptography involves mathematical theory and encryption methods. Cryptography algorithms are designed around computational hardness assumptions. This leads to heavy computational intensive algorithms. Sometimes a software approach could not be enough, but a hardware approach could be very complex.

In this project, we present a halfway between software and hardware approach using an FPGA. The intended outcome of the project is the design and development of two hardware-based accelerators for cryptography that can be dynamically loaded into the FPGA. Multiple approaches are presented during the project in order to design and test the accelerators.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark. This project was prepared with a fulfillment of requirements for working in the Bachelor Thesis.

The thesis was done as exchange student from the University of Zaragoza during the European mobility programme Erasmus Plus. All the work have been done in DTU Compute under the supervision of Alberto Nannarelli.

Lyngby 2. June 2018

Isak Edo Vivancos

Acknowledgements

First and foremost, I would like to give thanks to my supervisor Alberto Nannarelli for all the guidance during the work on the thesis and give me the opportunity to do my Bachelor Thesis as an exchange student in the DTU Compute department.

From the University of Zaragoza, I would also like to thank Dario Suarez for all the help provided during this academic year and giving me the opportunity to continue developing.

Special thanks to my parents Marga and Xavi for all their assistance and support during hard times, and finally my partner Desi for all her help and understanding while separated during this period.

Contents

Chapter 1

Introduction

In 1984, the Field Programmable Gate Arrays [1], FPGAs, showed up in the market by Xilinx. It is an evolution from the programmable logic devices, abbreviated as PLD. The FPGAs include the capacity of reconfiguration from PLD, but it also includes features from Application-specific integrated circuits, ASIC [2]. These circuits are designed to implement a unique task. It requires more designing time, but it reaches a better performance in time and power than general purpose circuits.

The FPGA allows the designer to implement application-specific designs in hardware, without creating a specific circuit for it. The architecture of the FPGA is based on logic blocks. These logic blocks are interconnected under the necessity of the programmer. In consequence, it is possible to create a custom design in hardware. Furthermore, it has the advantage that can be programmed several times. Even if the outcome is the same, there are differences between ASIC and FPGA. These differences summarised in the following table are mainly focused on cost.

	FPGA	**ASIC**
Time to market	Fast	Low
Development cost	Low	High
Design flow	Simple	Complex
Unit cost	High	Low
Performance	Medium	High
Power Consumption	High	Low
Unit size	Medium	Low

Table 1: FPGA and ASIC comparison

While the ASIC reaches better performance, the FPGA reduces the time and complexity necessary to create the design. Nowadays, the FPGA is integrated into System on Chips called all programmable SoC. These circuits are composed of a core and interconnected logic blocks that can be configured. As a result, these devices allow creating custom hardware in the programmable logic that can perform specific operations faster than the core. An overview can be seen in the following figure 1, this example corresponds to the Xilinx technology used in the project.

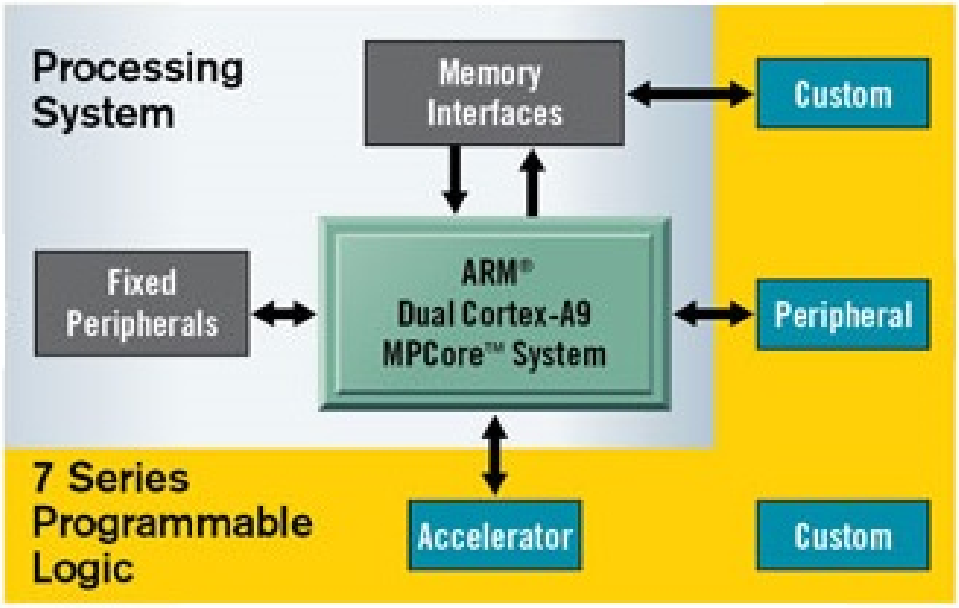

Figure 1: Zynq 7000 AP SoC

This is the concept of an accelerator, a specific design hosted in the programmable logic that performs an operation that requires a high cost in the core. This dedicated hardware can be used to provide much less latency and power consumption in specific tasks. The problem with the programmable logic is the limitation of resources. In consequence, it is possible to allocate a limited number of accelerators. The advantage of the FPGAs is based on the capacity of reconfiguration. It allows the device to re-program the implemented hardware into a different one. This feature can be used to program different accelerators under demand creating dynamic hardware libraries.

The aim of this project is to create an embedded design capable of changing the hardware configuration of the FPGA dynamically. This partial reconfiguration has to be invisible for the user. The idea of the project is to use this reconfigurable logic to create different accelerators depending on the task needed.

Specifically, our work is focused on cryptography algorithms and evaluating their performance, benefits, and possibilities. Cryptography is one of the most important topics while working in computer security. Moreover, cryptography algorithms are designed around computational hardness assumptions making some steps of these algorithms expensive in computational resources.

In Chapter 2 the board and the information necessary to understand the project will be described. In Chapter 3 and Chapter 4, two different algorithms implemented as accelerators will be presented, RSA and Blowfish respectively, while Chapter 5 will describe the embedded hardware platform designed to integrate these accelerators into the board. The following chapter, Chapter 6, will describe how to integrate the design into a Linux environment. In Chapter 7, the full process to perform the reconfiguration of the hardware will be shown, followed by Chapter 8 where the delay and power measurements for the design will be discussed. Finally, Chapter 9 will be focused on the final thoughts and future improvements.

The full process and the results obtained through this process is fully summarised in the following chapters. The order is the presented above, from the general concepts to the specific details of the implementation in order of execution, and finishing by the measurements and conclusions.

Chapter 2

Background

In this chapter, we will present the board used in the thesis focusing on which features provided by the board we have used for the project. Then, we will explain the AXI DMA used to communicate with the accelerators and how it works with them. Finally, the tools used in the project and workflow followed are presented.

2.1 Zedboard

The board used for the thesis is the Zedboard, a low-cost development board for the Xilinx Zynq®-7000 All Programmable SoC [3]. It is suitable for both hardware and software developers to create rapid prototypes. The ZedBoard mix onboard peripherals and expansion capabilities. It is a common project by Avnet, Xilinx, and Diligent. The FPGA where the accelerators are built is Artix-7 provided by Xilinx. This company is known for inventing the filed-programmable gate array (FPGAs).

The board is provided with different memories, connectivity, user inputs, video, etc. In the project, it is not necessary to use all of them. The DDR3 RAM memory and the SD card are required to launch the Linux on the board. The JTAG micro USB port will be used to program the FPGA in the bare metal test, and the UART micro USB port to connect the board. Finally, Ethernet RJ45 to provide internet connectivity to the board. The most important technical specifications are summarised below:

ZYNQ™-7000 SOC XC7Z020	Dual ARM® Cortex™-A9 MPCore™ Up to 667 MHz operation NEON™ Processing / FPU Engines
Memory	512 MB DDR3 256 Mb Quad-SPI Flash 4 GB SD card
CONNECTIVITY	10/100/1000 Ethernet USB OTG (Device/Host/OTG) USB UART
VIDEO/DISPLAY	HDMI output (1080p60 + audio) VGA connector 128 x 32 OLED
AUDIO	24-bit stereo audio CODEC Stereo line in/out Headphone Microphone input
ANALOG	Xilinx XADC header Supports 4 analog inputs 2 Differential / 4 Single-ended
DEBUG/PROGRAMMING	On-board USB JTAG programming port ARM Debug Access Port (DAP)
OTHERS	12V DC input @ 3.0 A (Max) Length: 6.3 inches Width: 5.3 inches CE and RoHS certified

Table 2: Zedboard components

The position of the components is shown in the next figure 2. The micro USB ports that we need to connect the board to the computer are located in the top left part of the board. While the SD card cage is located behind the board, on the right side. The switches, LEDs, and other ports that can be seen in the picture are not used for this project.

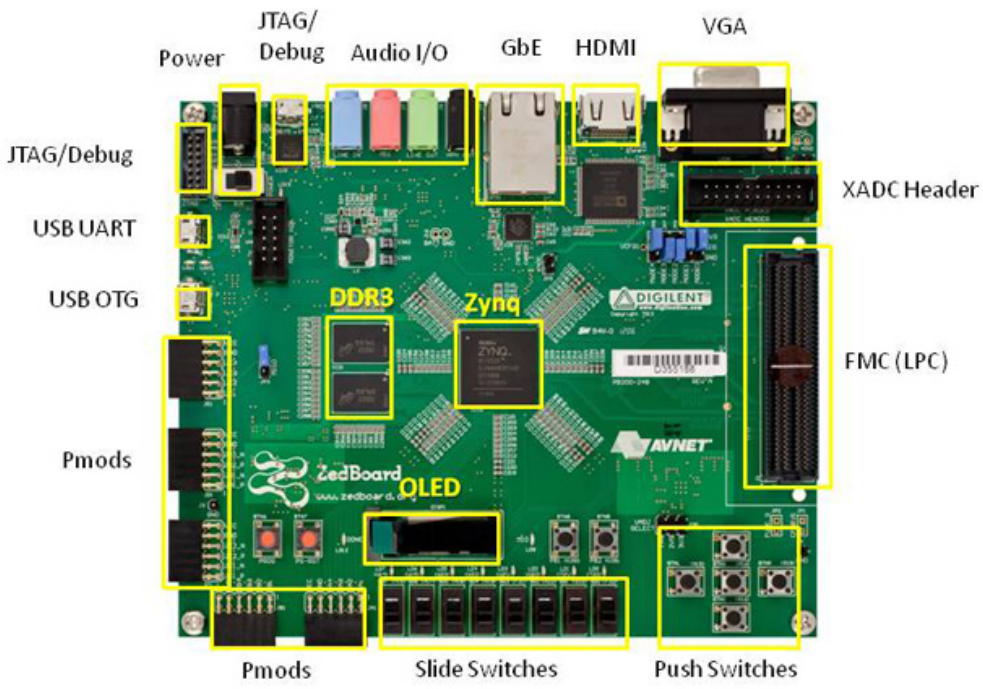

Figure 2: Zedboard components

This board [4] can be divided into two different parts, programmable system and programmable logic. The programmable system (PS) contains the ARM Cortex^{TM}-A9 which are capable to host a Linux environment. This PS is connected directly to 512Mbyte of DDR3 RAM memory. The programmable logic (PL) allows the programmer to implement custom designs on hardware, it is composed of 85k 7-series cells. Concretely, the cryptography accelerators are implemented in this part of the board. This PL is connected to GPIO, VGA, HDMI, etc. However, for the project, we only need to connect to the Ethernet controller. The connections are summarised in the figure 3 located on the next page.

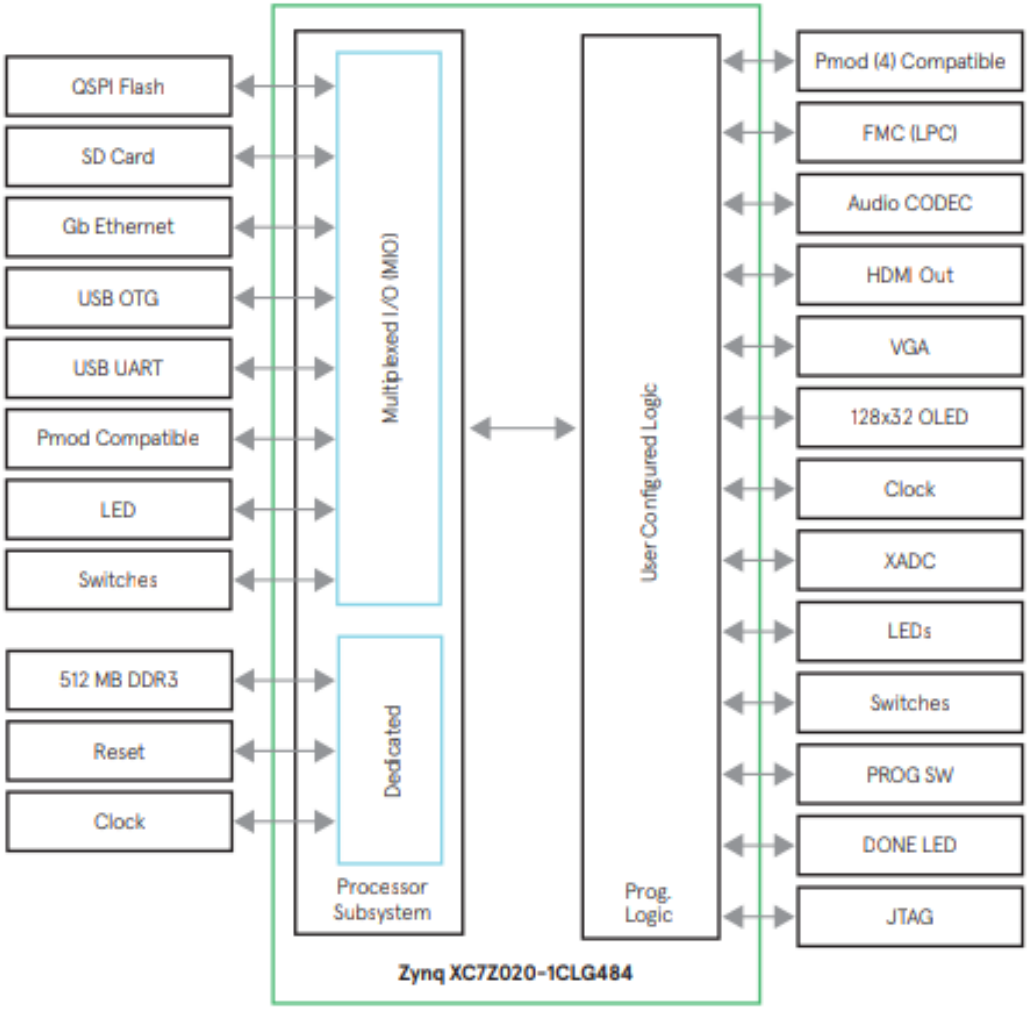

Figure 3: Zedboard connections

The programmable logic performed by the Artix-7 is composed of different components [5]. The board counts with 53,200 look-Up tables, which is a table that determines what the output is given an input. It includes also 220 DSP Slices [6], which are specialised cells for digital signal processing, but they can be used to speed up many applications, such as the cryptography algorithms implemented. Each DSP slice includes a 25 x 18 two complement multiplier (which can be bypassed), and a 48 bits accumulator. Finally, related to memory, the PL provides 106,400 flip-flops, and 140 RAM blocks of 36Kb each (Total of 4.9Mb). This board doesn't own a PCIe bus.

2.2 AXI DMA

The AXI Direct Memory Access IP [7] provides high bandwidth between memory and AXI-Stream interfaces. There are two independent channels, Axi Stream to Memory-Mapped (S2MM) and Memory Mapped to Axi Stream (MM2S). Both can work with different width from 32 bits to 1,024 bits. In our application, we can use the MM2S channel to send all the data to the accelerator in one burst, and the MM2S to get the result back into a memory map. The theoretical throughput for each channel considering a frequency of 100MHz and 10,000b transferred is:

MM2S Channel: 399.04Mb/s
S2MM Channel: 298.58Mb/s

An schematic overview of the AXI DMA could be the following 4:

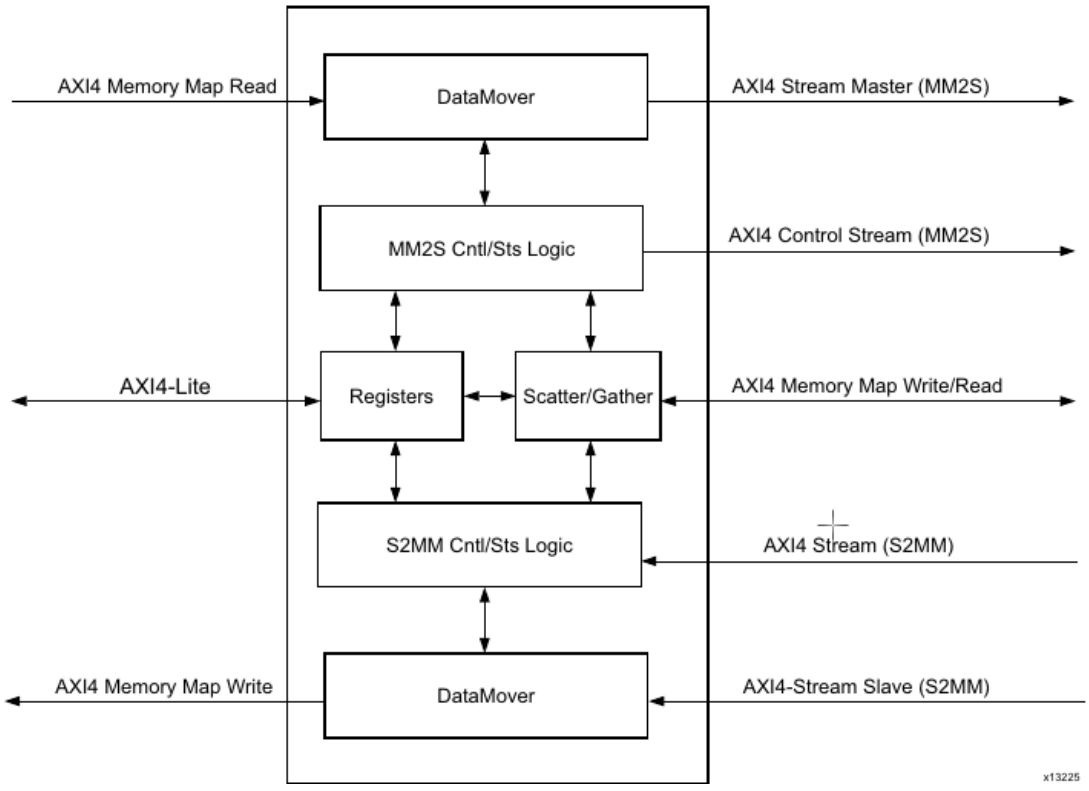

Figure 4: AXI DMA Block overview

The registers mapped in memory can be used to configure the Axi DMA, there are separate registers for each channel, but their functionality is equivalent. For the direct map mode that we need in our application, we use the following. The *Control register* to configure the DMA in read/write mode, and enable the interrupts after a transaction.

The *Status register* to check the status of the DMA, including if it is halted due to an error. The *Destination address* to specify where is located the memory mapped. Finally, the *Length* to specify the number of bytes in the transaction. The last one must match the exact number of bytes, otherwise, the DMA might halt.

The DMA configures the port of the channels as AXI4-Stream [8], so it is possible to send an unlimited length of data. It is based on a master-slave protocol. For the DMA specific case, MM2S channel port works as a master, and S2MM port works as a slave. This port specification is composed of the signals, TDATA, TREADY, TVALID, TKEEP, and TLAST.

In the AXI4-Stream protocol the signal TDATA is composed by the content we want to transmit, in blocks of the size of the channel. TVALID indicates if the data is valid, and TREADY informs the master if the salve is ready for a transmission. TKEEP indicates whether the content transferred by the signal TDATA is processed as part of the data stream. Finally, the TLAST signal informs the slave that it is the end of the transition. This TLAST signal is really important when transferring data from our accelerator to the DMA by the S2MM channel. The signal must be set to '1' just before or during the last block transmission, otherwise, the DMA will hang. A temporal diagram of how the signals works can be seen in the picture 5 below:

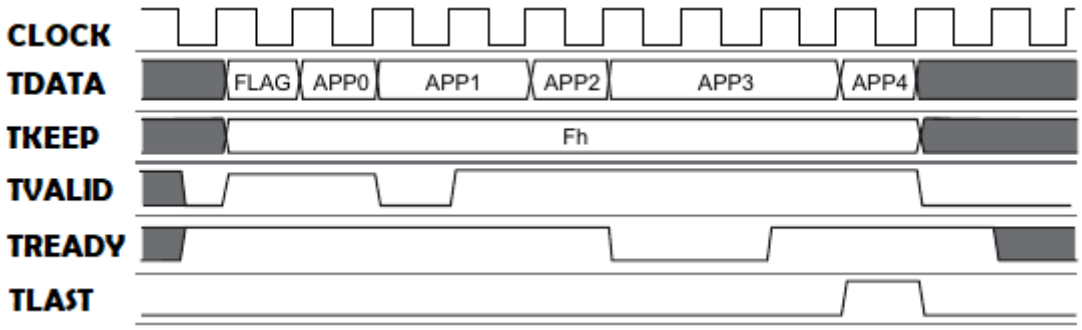

Figure 5: DMA transfer using AXI4-Stream

2.3 Tools and Workflow

This project involves hardware and software design, so we need advanced tools to improve the process performance. Since we are working with the Zedboard, we use the collection of tools provided by Xilinx in the version 2017.4, including Vivado HLS, Vivado, XSDK and Petalinux. Each one of this tools is focused on one different step of the workflow.

In the project, we follow the workflow shown in the figure 6. The main tool is Vivado [9]. This assists the user in the process of FPGA design, implementation, and verification. In addition to the traditional RTL-to-bitstream design flow, the environment offers the option to create graphically the design based on Intellectual Property blocks (IP blocks). These IPs can be instantiated, configured and connected using the graphic environment. The designer only needs to make sure that the IPs are properly connected, and the ports are specified properly. This is our approach followed in the project to design the hardware.

Once we have the design done, Vivado has support to do the synthesis and implementation of the design. The synthesis consists in transform the RTL design implementation into logic gates, while the implementation consists in place the synthesized design and make the physical connections. Finally, we can create the bitstream to program the FPGA.

We use Vivado HLS to implement the accelerator [10]. This tool transforms high-level code such as C++ into register transfer level (RTL) implementation that can be synthesized into the FPGA. This RTL can be packaged into a Custom IP that can be used later on by Vivado. For that purpose, it is necessary to specify the ports in the input and outputs of the package.

Once the bitstream is generated using Vivado, we can go through two different approaches to build the application, bare metal application or integrate the design into a Linux environment. For the bare metal approach, we use the Xilinx Software Development Kit (XSDK) [11] to implement the software. It is an Eclipse-based Integrated Development Environment (IDE) for the Zynq family microprocessors. It allows us to create a Board Support Package from the generated custom bitstream to boot the board, and run a C application.

Also, It is possible to integrate the hardware design in a Linux environment, in Zedboard case using Petalinux tools [12]. This group of tools provided by Xilinx allow us to build and configure a Linux based on the Suse distribution for the board. Using this tools we can create and add custom C applications to be used in the Linux as usual shell programs. In our project, we will follow both approaches to test the hardware design.

The whole process followed can be found in the figure 6 located in next page. It can be summarised as creating the custom accelerator with Vivado HLS. Then, the accelerator is added to the hardware design in Vivado. After the synthesis and implementation, we create the bitstream. Using the bitstream we can use the XSDK to create a bare metal application or integrate the design into Petalinux.

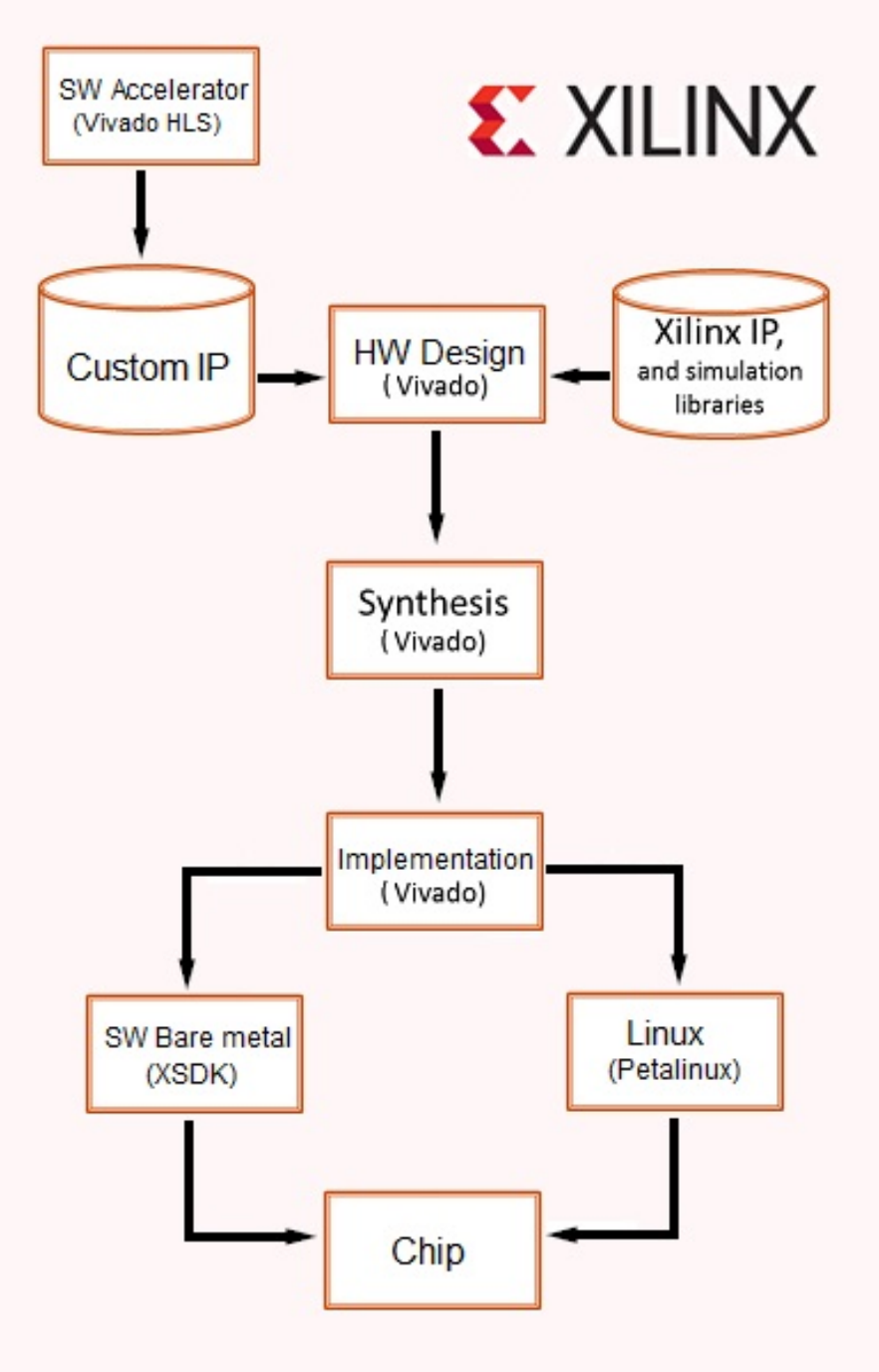

Figure 6: Project Workflow

Chapter 3

RSA algorithm

In this chapter, we will present the main algorithm developed in the thesis, the RSA algorithm. First, we will introduce the algorithm and why is computationally intense. Then, the actual limitations to implementing this algorithm. Finally, we will show how to implement the RSA in hardware using the Montgomery modular exponentiation approach.

3.1 Algorithm specification

The RSA [13] (Rivest, Shamir, Adleman) is the main cryptography algorithm used nowadays. It is an asymmetric algorithm because needs two different keys to encrypt and decrypt the message, one public and the other private. It is based on the huge computational cost of factorize the product of two large prime numbers. RSA includes four different steps: key generation, key distribution, encryption, and decryption.

To generate the key first it is necessary to choose two different large prime numbers p and q, these numbers must be secret. The prime numbers should be chosen randomly for security purpose, and similar in length but not the same. Then, the next step consists in calculate the product of both prime numbers called n. This number is part of the public and the private key, the length of n is considered as the length of the key. The following step consists in compute the Euler's totient function of n defined as:

$$\phi(n) = (p-1)(q-1).$$

Finally, the public exponent e is a number chosen between 1 and $\phi(n)$, while the private exponent d must be chosen being congruent with e modulus $\phi(n)$. This can be computed using the extended Ecluidean algorithm [14].

$$e \cdot d \equiv 1 \ mod \ \phi(n)$$
$$d = e^{-1} mod \ \phi(n)$$

In consequence, the public key would be the pair (e,n) and the private key would be the pair (d,n). Once both keys are generated it is possible to encrypt and decrypt messages. Given a plain text M, the encryption is performed as the exponentiation of M by the public exponent modulus n, being always M less than the modulus. The decryption is performed changing the public exponent by the private exponent when performing the exponentiation.

$$C = M^e mod \ n$$
$$M = C^d mod \ n$$

One example using simple numbers is the following. First we select to prime numbers $p = 29$ and $q = 31$. So the modulus n is computed as:

$$\text{n} = \text{pq} = 29 \cdot 31 = 899.$$

In consequence, the maximum value that could be encrypted using this modulus n is 899. And the Euler's totient function of n is calculated as:

$$\phi(n) = (p-1)(q-1) = 28 \cdot 30 = 840$$

Now, the public exponent e is chosen as a number between 1 and 840, in this case e will have the value 307. So, it is possible to compute the private exponent d using the extended Euclidean algorithm as follows:

$$d = e^{-1} mod \ \phi(n) = 307^{-1} mod \ 840 = 643$$

Being e and d congruent modulus $\phi(n)$:

$$307 \cdot 643 = 6033 = 1 \ mod \ 840$$

The public key is (307,899) and the private key is (643,899). Now given a plain text such as the letter "I", which is "49" in hexadecimal according to the ASCII table and "73" in decimal. It is possible to cipher this text using the public exponent e and return to the original using the private exponent d:

$$C = 73^{307} \ mod \ 899 \ = \ 292$$
$$M = 292^{643} \ mod \ 899 \ = \ 73 \ = \ 0x49 \ = \ "I"$$

3.2 Limitations in hardware

The hardware accelerator is focused on the encryption and decryption steps in the RSA algorithm. The key generator step is done only once, while the encryption and decryption are performed several times. Due to the characteristics of the RSA, the encryption and decryption accelerator are the same.

The problem of the encryption process lies in the exponentiation [15]. It is not possible to calculate first M^e and then the modulus, the space to store that number is enormous. For example, for a 256 bits exponent, the stored number is:

$$log_2(M^e) = e \cdot log_2(M) \approx 2^{256} \cdot 256 \approx 10^{80}$$

It is necessary to calculate the n modulus after each multiplication. We would need to perform a remainder operation every multiplication. The cost of dividing or doing the remainder operation by a number n is very big. Algorithms such as the binary exponentiation [14] can reduce the number of multiplications, but can not avoid the remainder operation.

3.3 Montgomery's method

In 1985, P. L. Montgomery [16] proposed a method to compute $a \cdot bmodn$ without dividing by n, only by powers of 2. This algorithm is based on the representation of the residue class module n. To explain this concept is necessary to introduce the idea of r.

Assuming that the modulus is a number of k-bits, r is defined as 2^r, being defined as $2^{r-1} \leq n \leq 2^r$. The reduction algorithm requires that r and n are primes between them. This is mathematically defined as their greater common divisor equal to 1, gcd(r,n) = 1. This is always true in RSA because n must be an odd number. The residue class for a number $a < n$ is described as:

$$\overline{a} = a \cdot r \ mod \ n$$

The Montgomery reduction exploits the property that there is one to one correspondence between the numbers in the range 0 and n-1, and the residue result of the above multiplication. It allows that given two residue integers $\overline{a}$ and $\overline{b}$ the Montgomery product is defined as the n-residue.

$$\overline{R} = \overline{a} \cdot \overline{b} \cdot r^{-1} \ mod \ n$$

Being r^{-1} the inverse of r modulo n.

$$r \cdot r^{-1} = 1 \ mod \ n$$

Additionally, to compute the product we need the integer n' described by the following equation. It can be calculated using the extended Euclidean algorithm [14].

$$r \cdot r^{-1} - n \cdot n' = 1$$

The resulting pseudo C algorithm to perform the Montgomery product is given below.

```
uint monPro(uint a, uint b) {
    uint t = a * b;
    uint m = t * c_n % r;   //Remainder by power of 2
    uint u = (t + m * n) / r; //Division by power of 2
    if (u >= n) return u - n;
    else return u;
}
```

The most important feature of the Montgomery multiplication is that the division and remainder operations are performed by r, not by n, being r a power of 2. In fact, this algorithm can be explicitly translated as the one below.

```
uint monPro(uint a, uint b) {
    uint t = a * b;
    uint m = t * c_n & (r - 1);
    uint u = (t + m * n) >> k;
    if (u >= n) return u - n;
    else return u;
}
```

Using a minus and a logical bitwise and operation instead of the remainder, and a logical right shift instead of a division.

3.4 Montgomery's exponentiation algorithm

Due to the necessity of precomputing some values, the Montgomery product presented in the previous section is especially suitable for a big number of multiplications, such as in the case of modular exponentiation. For the RSA case, the binary method [14] must be changed using Montgomery products instead. In consequence, the number of operations depends on the number of bits of the modulus, O(k), instead of the value of the exponentiation, O(exp). It is important to note that the value of the exponent is much bigger than the number of bits.

First, the complement of n must be precomputed using the extended Euclidean algorithm. For that, it is also necessary the number of bits k, and the number r. Then, the computation of the residue of the message and obtain its n-residue $\overline{M}$. Two remainders by n are needed, but these operations are performed only once. During the inner loop, all modular multiplications are performed using the Montgomery product.

The n-residue $\overline{x}$ of the quantity $x = M^e \ mod \ n$ is obtained after the loop execution. In consequence, it is necessary to obtain the value of x from its n-residue by executing the Montgomery product between $\overline{x}$ and 1. This is clear noting that:

$$\text{Given: } \overline{x} = x \cdot r \ mod \ n$$
$$\text{Implies: } x = \overline{x} \cdot r^{-1} \ mod \ n = \overline{x} \cdot 1 \cdot r^{-1} \ mod \ n = monPro(\overline{x}, 1)$$

The resulting pseudo C algorithm to perform the Montgomery exponentiation is given below.

```
uint monExp(uint32_t M, uint exp, uint n) {
    //Precompute n'. Extended euclidean algorithm
    uint mask = r >> 1;
    uint rem_m = M * r % n;
    uint rem_x = M % n;
    for(int i = k - 1; i >= 0; i--) {
        rem_x = monPro(rem_x,rem_x);     //First monPro
        //If the bit is 1 in that position
        if((exp & mask) != 0)
            rem_x = monPro(rem_m,rem_x,n); //Second monPro
        mask >>= 1;
    }
    return monPro(rem_x,1);
}
```

3.5 Algorithm trace

An example of the execution trace of this algorithm is the following. We use the public and private keys obtained in the previous section. The public key is (307,899), the private key is (643,899), and the message we want to cipher is the letter "I".

To cipher the text, first select the values for k and r.

$$\text{Given n} = 899:$$
$$2^9 \leq 899 \leq 2^{10}$$
$$k = 10, r = 2^{10} = 1024$$

Using the extended Eculidean algorithm we can obtain the values of r^{-1} and n'

$$r \cdot r^{-1} - n \cdot n' = 1$$
$$1024 \cdot 187 - 899 \cdot 213 = 1$$

Then we calculate $\overline{M}$, and the value of $\overline{x}$ when x is equal to 1.

$$\overline{M} = M \cdot r \ mod \ n = 73 \cdot 1024 \ mod \ 899 = 135$$
$$\overline{x} = r \ mod \ n = 1024 \ mod \ 899 = 125$$

Now we can calculate the loop for the number of bits in the key, in this case 10 times. For each iteration, depending on the i-est bit we perform another product between $\overline{M}$ and $\overline{x}$. To check this trace we need the binary value of the exponent 307 which is 0b0100110011.

e_i	**First**	**Second**
0	monPro(125,125) = 125	-
1	monPro(125,125) = 125	monPro(135,125) = 135
0	monPro(135,135) = 865	-
0	monPro(865,865) = 412	-
1	monPro(412,412) = 236	monPro(135,236) = 147
1	monPro(147,147) = 777	monPro(135,777) = 84
0	monPro(84,84) = 639	-
0	monPro(639,639) = 361	-
1	monPro(361,361) = 834	monPro(135,834) = 649
1	monPro(649,649) = 500	monPro(135,500) = 540

Table 3: RSA Encryption Montogomery

Finally we calculate the Montgomery product between $\overline{x}$ and 1 to get the value of the cipher text.

$$\text{C} = \text{monPro}(\overline{x}, 1) = \text{monPro}(540,1) = 292$$

The decryption process is symmetric as the one described above, but we need to change the public key to the private key. The value for k, r, r^{-1}, and n' are the same. So we can start computing $\overline{M}$ and $\overline{x}$, and then the loop. To check this trace we need the binary value of the exponent 643 which is 0b1010000011.

$$\overline{M} = M \cdot r \ mod \ n = 292 \cdot 1024 \ mod \ 899 = 540$$
$$\overline{x} = r \ mod \ n = 1024 \ mod \ 899 = 125$$

e_i	**First**	**Second**
1	monPro(125,125) = 125	monPro(540,125) = 540
0	monPro(540,540) = 355	-
1	monPro(355,355) = 289	monPro(540,289) = 781
0	monPro(781,781) = 284	-
0	monPro(284,284) = 149	-
0	monPro(149,149) = 5	-
0	monPro(5,5) = 180	-
0	monPro(180,180) = 439	-
1	monPro(439,439) = 614	monPro(540,614) = 387
1	monPro(387,387) = 256	monPro(540,256) = 135

Table 4: RSA Decryption Montogomery

Finally we calculate the Montgomery product between $\overline{x}$ and 1 to get the value of the cipher text.

$$\text{C} = \text{monPro}(\overline{x}, 1) = \text{monPro}(135,1) = 73 = 0\text{x}49 = \text{"I"}$$

In conclusion, Montgomery's exponentiation allows us to perform the modular exponentiation dividing by a power of 2, instead of n. Moreover, the execution time depends on the number of bits instead of the value of the exponent. This approach allows using keys up to 4096 bits. The number of operations is significantly less.

Chapter 4

Blowfish algorithm

In this chapter, we will present the second algorithm developed in the thesis, the Blowfish algorithm. The aim of this algorithm is to provide a demo application for partial reconfiguration, and an example of symmetric algorithm. We will show how to implement this algorithm in hardware.

4.1 Algorithm specification

The Blowfish [17] is a symmetric cryptography algorithm designed in 1993 by Bruce Schneier [18]. This algorithm uses the same key for encryption and decryption, in consequence, client and server must share the key. Blowfish [19] is a 64-bits block cipher with a variable length. It is divided into two steps: key expansion and data encryption. Even if this is a really fast algorithm, the precomputation of the key is very slow, it requires to operate with 4 Kilobytes of text. This drawback makes AES or Twofish more likely to be used in modern applications.

The algorithm uses two different structures to support the encryption and decryption process, the P-Array and four S-Boxes. P-Array consists of 18 32-bits subkeys, from P1 to P18, and four 32-bits S-Boxes with 256 entries. Both have to be initialized using hexadecimal values of pi [20].

The key expansion step consists in XOR the P-Arrays in order byte by byte with the secret key. If the key is not long enough, it is repeated again. Then, a 64-bits block initialized with zeros is encrypted using the algorithm. The resulting block substitutes P1 and P2. After that, this block is encrypted again and substitutes P3 and P4. This process is repeated until getting new values for the whole P-Array and the four S-Boxes.

Once the key is expanded, it is possible to perform the encryption and decryption. Both processes use the same expanded key, so it is necessary to expand it only the first time. The encryption process consists in the application of a function F, this function is iterated for sixteen rounds.

Each round begins by XOR the left half of the data with one P-array entry and use this data as input for the F function. Then, the same for the right half of the data. Finally, swap L and R. While in encryption the P-Array entries go from 0 to 18, during the decryption the P-Array entries are used in reverse order. The next figure 7 summarise the process:

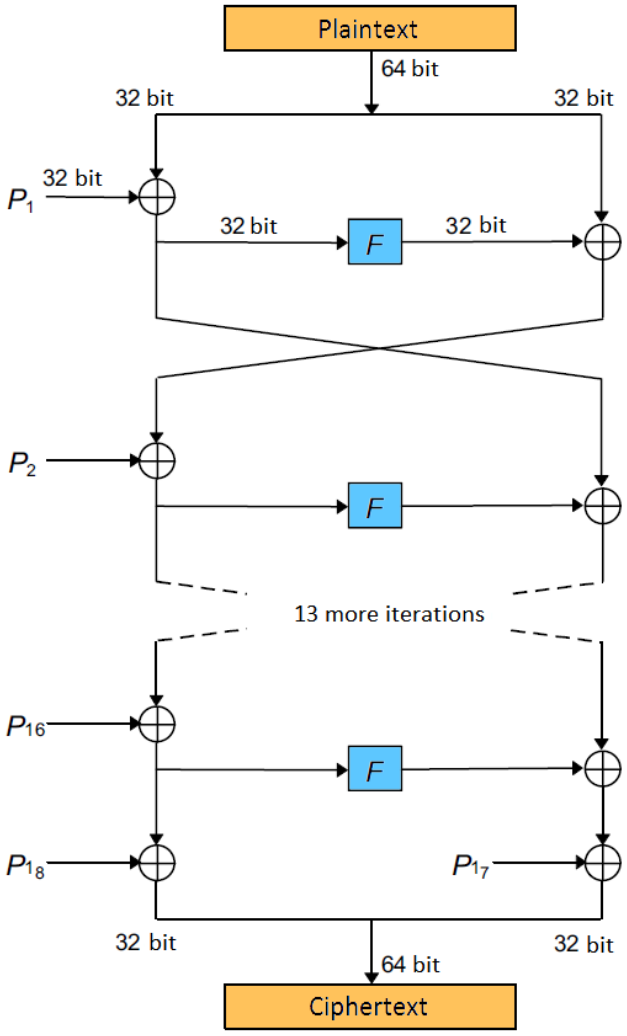

Figure 7: Blowfish worklow

The resulting pseudo C algorithm for the encryption process is shown below. The decryption is almost the same, but changing the indexes of the P-Arrays.

```
#define N 16

void Blowfish_Encrypt(bit32* left, bit32* right){
    //From P[1] to [16]
   for (uint i = 0; i < N; ++i) {
      left = left ^ P_ARRAY[i];
      right = f_function(left) ^ right;
      // Exchange left and right
   }

   // Exchange left and right
   right = right ^ P_ARRAY[N]; //P[17]
   left = left ^ P_ARRAY[N + 1]; //P[18]
}
```

Each iteration the data pass through the function F, this function splits the data into four 8-bit sub-data. Each of this 8-bit blocks is the output of one of the four S-Boxes, which returns 32-bits each. So the outputs are added module 32 and XORed to get a final 32-bit encrypted block. The process is summarised in the following figure 8:

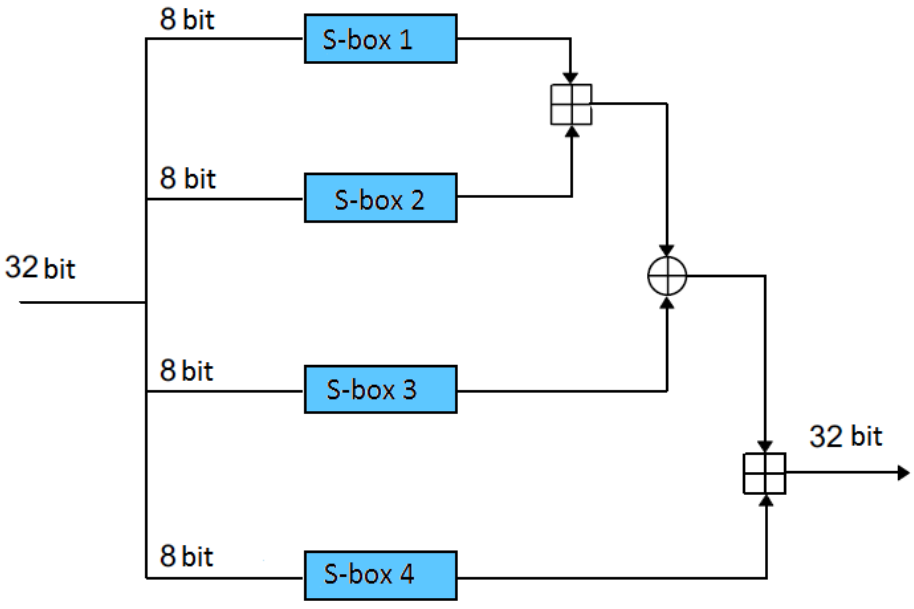

Figure 8: Blowfish F function

This function F is can be defined by the pseudo c algorithm below. Being the S-Boxes one array of two dimensions, the first one the number of boxes, and the second one the number of entries per box.

```
uint f_function(bit32 input) {
   //8-bit blocks
   bit8 a, b, c, d;
   d = (bit8)(input & 0xff); input >>= 8;
   c = (bit8)(input & 0xff); input >>= 8;
   b = (bit8)(input & 0xff); input >>= 8;
   a = (bit8)(input & 0xff);

   //Additions and XOR operations
   bit8 output;
   output = S_BOXES[0][a] + S_BOXES[1][b];
   output = output ^ S_BOXES[2][c];
   output = output + S_BOXES[3][d];
   return output;
}
```

4.2 Algorithm trace

We provide an example of the execution of the algorithm. Since the key expansion is too long, we will present the encryption and decryption process given a 64-bits key. The block we will encrypt is the chain "IsakEdo", which translated into hexadecimal is 0x004973616b45646f.

The encryption process begin by splitting the original block in two halves, left and right half. To follow the flow of the data the original left half will be represented in blue, and the right half will be represented in black.

Left = 0x00497361
Right = 0x6b45646f

Once we have the two halves we iterate through the F function. The following table summarises the output of the iterations after each iteration and permutation. Internal calculus of the F function is avoided for simplicity. There is no permutation after the last iteration.

Iteration	P-Array entry	**Left**	**Right**
1	**P[1]**	0xdab02e0c	0x51aa701a
2	**P[2]**	0x7d171ef0	0xc5e41231
3	**P[3]**	0xa3aed8f3	0x4ba3e0f4
4	**P[4]**	0x2a3a8cd9	0x39663fe8
5	**P[5]**	0x30242773	0x38414858
6	**P[6]**	0xc0025588	0x8a741901
7	**P[7]**	0xfd6223c2	0x902cdc6e
8	**P[8]**	0x6746d641	0x13be0747
9	**P[9]**	0xc82b7dcb	0x2ffc9fa
10	**P[10]**	0x9876b5c3	0xf8cb4507
11	**P[11]**	0x646f41c4	0x726034da
12	**P[12]**	0x1a5a75de	0x45d0392a
13	**P[13]**	0x95d40ab9	0xa3f18d67
14	**P[14]**	0x7416ebd7	0xd0efd6bf
15	**P[15]**	0x66f16f3b	0xb80b9ee9
16	**P[16]**	0x357cde77	0x4adcded8

Table 5: Blowfish Encryption

Finally, the last values are XORed per the P-Array entry 17 and 18 respectively. It returns the final encrypted text:

$$\text{Left} = 0\text{x}235892\text{be}$$
$$\text{Right} = 0\text{x}82991377$$

For the decryption process, we present the table of the iteration process. The left and right used are the encrypted halves obtained through the previous process. In this case, the P-Arrays entries are used in reverse order.

Iteration	P-Array entry	**Left**	**Right**
1	**P[18]**	0x704e5346	0x357cde77
2	**P[17]**	0x836267f3	0xb80b9ee9
3	**P[16]**	0x6fecf859	0xd0efd6bf
4	**P[15]**	0x00ebe52c	0xa3f18d67
5	**P[14]**	0xcbcbcc63	0x45d0392a
6	**P[13]**	0xd9743de9	0x726034da
7	**P[12]**	0xe8e948e3	0xf8cb4507
8	**P[11]**	0x235e3f8b	0x2ffc9fa
9	**P[10]**	0xf595c3d5	0x13be0747
10	**P[9]**	0x64a83d84	0x902cdc6e
11	**P[8]**	0x686fc1be	0x8a741901
12	**P[7]**	0x8336019a	0x38414858
13	**P[6]**	0x59d82475	0x39663fe8
14	**P[5]**	0x5f2cf52a	0x4ba3e0f4
15	**P[4]**	0x671e8e1e	0xc5e41231
16	**P[3]**	0x51aa701a	0x74115852

Table 6: Blowfish Decryption

Finally, the last values are XORed per the P-Array entry 2 and 1 respectively. It returns the original text:

Left = 0x00497361
Right = 0x6b45646f

Chapter 5

Hardware integration

In this chapter we will present the hardware implementation in Vivado HLS for both algorithms presented in the previous sections, RSA and Blowfish. Then we will explain how to embed this custom module into a Vivado hardware design using the IP blocks design feature. Finally, we will test the hardware using a bare metal application for the RSA algorithm.

5.1 Module implementation

After selecting the two algorithms, we can proceed to the hardware implementation. The implementation process is equal for both algorithms, but we will focus on the RSA first. Before implementing the accelerator into Vivado HLS, we need to know the specifications for the accelerator. We want to send all the data to the accelerator through the DMA, and get the result back using the DMA too. Moreover, we want the accelerator to interrupt the system when the operation is done.

As a result, two inputs ports are required. One of them to transmit the data using the DMA, and one to control the accelerator. In the case of RSA, the control specifies the number of blocks. Furthermore, one output port is necessary to send the result to the DMA. This can be specified in Vivado HLS using the following notation:

```
//Interface
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=control
#pragma HLS INTERFACE axis port=output_stream
#pragma HLS INTERFACE axis port=input_stream
#pragma HLS INTERFACE ap_none port=TLAST
```

In the first statement, the return port indicates that one output port is created with the returned value of the main function as raw data. This will produce an interrupt when set to '1'. The control port is read as raw data, while the input_stream is configured with the standard of Vivado AXI4-Stream used by the DMA. Finally, the output_stream is specified as Vivado AXI4-Stream, but when creating this from Vivado HLS, both TLAST and TKEEP signals are missing. In consequence, we need to manually create and manage TLAST. If we do not do that, the DMA will hang after the first output transaction.

When using AXI4-Stream as input in Vivado HLS, the accelerator stays idle until a new block of the data is received. We can use that in our favor to program the accelerator as a finite state machine. The process is the following. The first state reads the exponent from the input and set a flag saying that is read. Then, the second state reads the modulus, do the precomputation, and set another flag to done. The last state encrypts every block received from the input until reach the number of blocks specified by the control input. After the last block, the TLAST signal and the interrupt are set to '1'. Finally, it resets all the flags and come back to the first state. It is necessary to send one trash byte at the end of the data to awake the accelerator and reset everything. This is a state machine diagram of the RSA accelerator 9.

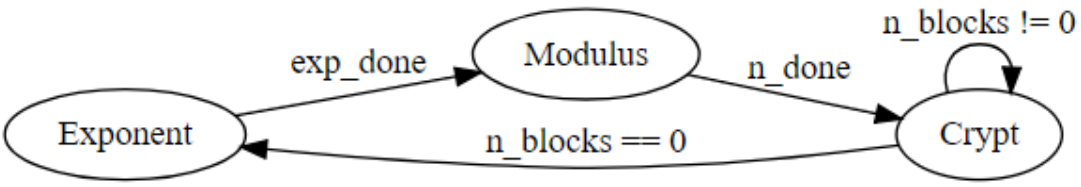

Figure 9: RSA accelerator state diagram

The approach followed for the Blowfish is similar to the RSA, but there are some differences. The accelerator has only two states, the first one performs the initialization of the key if needed, and the second states performs encryption and decryption. Since the key is not always initialized we use the control input to check if we need to do it. In the same way, since the encryption and decryption follow different approaches we need to use the control input to select the operation mode. For that purpose we need to steal two bits from the control sequence, the remaining bits specify the number of blocks as in the RSA.

All the code for the RSA hardware implementation can be found in the appendix A.1. While the code for the Blowfish hardware implementation can be found in the appendix A.2

After the implementation, we can proceed to synthesis the accelerator using Vivado HLS. It transforms the C++ code into RTL that can be used by Vivado later on. After that, we can export the RTL of the accelerator creating our own custom IP that can be used by Vivado. The result can be seen in the figure 10. In that figure, all the interfaces specified previously using the #pragma directive can be seen.

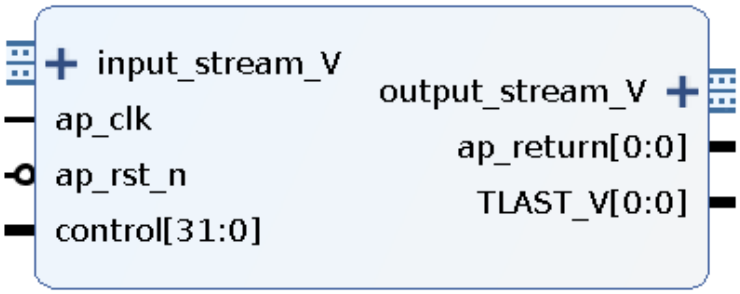

Figure 10: Accelerator interface block

Once the accelerator is packaged in a custom IP, we need to specify the return port as interrupt manually. Otherwise, Vivado won't detect it as an interrupt source when creating the bare metal application. This is done inside the Vivado project, after instantiating the block. Both accelerators works with sizes of 64 bits. This is done due to the lack of resources in the board when trying to increase the number of bits for the RSA.

5.2 Embedded platform

At this point, we have both accelerators implemented in hardware and packaged into a custom IP. Since the following process is exactly equal for both of them, we will present the hardware design for the RSA algorithm. The next step consists in design a wrapper for the accelerator allowing us to communicate with it. We can use the IP blocks provided by Vivado to create it. First of all, we need to add the processing system block [21].

This block works as logic connection between the programmable system and the programmable logic parts of the FPGA. It allows the programmable logic to interact with the SoC Zynq-7000. This block can be used to enable features such as I/O Peripherals, AXI I/O ports, PL Clock and interrupts, Multiplexed I/O (MIO), etc. An overview of the processing system block can be seen in the following figure 11:

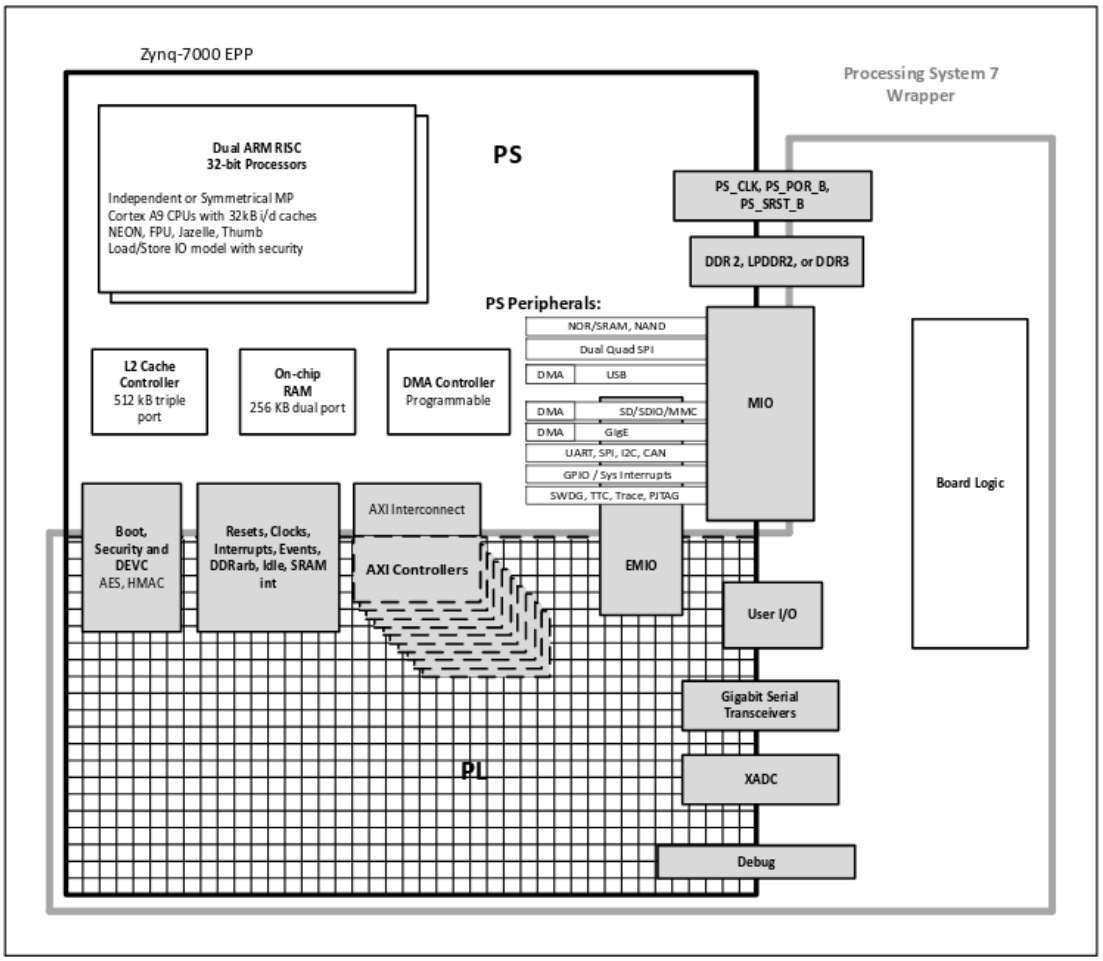

Figure 11: Processing system structure

Specifically for our project, we need to enable the PL to PS interrupts to allow the DMA and accelerator to interrupt the core. Also, a high performance AXI slave to get the data back from the DMA. Finally,by using the MIO we add support for SD card, Ethernet, GPIO, UART, and other peripherals.

Now we have the processing system to communicate with the PS. We can divide the remaining design into two pieces, the accelerator wrapper, and the system blocks. The system blocks are a set of blocks used as a support between the accelerator wrapper and the processing system. In our case, these blocks consist of an asynchronous reset generation, an interconnection network, and the concatenation block. The interconnection allows the processing system to send data to different slaves using the same master port. While the concatenation block merges the three different types of interruptions in one wire. We can see an overview of the design in the following picture 12.

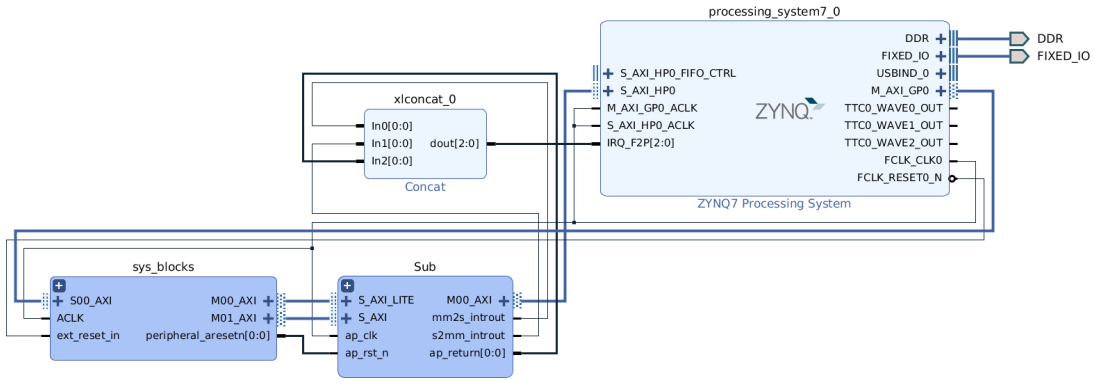

Figure 12: Global design in Vivado

In the figure above, the accelerator wrapper is defined as *Sub*. Thanks to the interconnect block inside the *sys_block* module, the processing system can send orders to the accelerator and the DMA using the same port. This is performed by mapping the IP blocks into memory, which is done by Vivado.

processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
Sub/axi_dma_0	S_AXI_LITE	Reg	0x4040_0000	64K	0x4040_FFFF
Sub/RSA/axi_gpio_0	S_AXI	Reg	0x43C0_0000	64K	0x43C0_FFFF
Sub/axi_dma_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF

Figure 13: Memory mapped addresses

By writing in this directions 13, we can control the behavior of the DMA and the accelerator. Even more, the DMA has different registers to program their behavior as explained in the AXI DMA section. We can access them adding an offset to the original address provided by Vivado. Also, we can specify the range of memory that can be accessed by each DMA channel, in this case 512 Mbytes, which is the whole DDR3 RAM.

Inside the accelerator wrapper, we have the accelerator and the DMA. As we have designed the accelerator to read the input data from the DMA and return the result using the DMA too, we need to make the logical connections between them. We connect the DMA MM2S channel into the input stream of the accelerator, and the output stream to the DMA S2MM channel. We can see the design in the following figure 14.

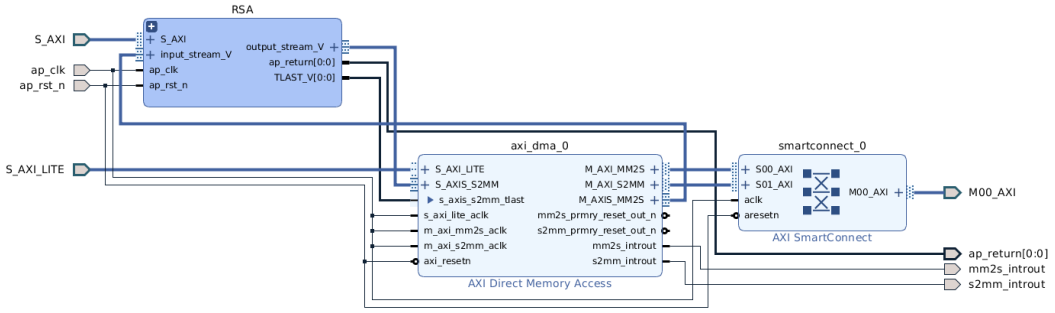

Figure 14: Accelerator wrapper design in Vivado

It is necessary to connect the manually created TLAST port to the DMA S2MM channel TLAST signal to make the DMA work properly. We also use a smartconnect IP block to send the status of the DMA to the processor when required. By using this IP we only need one slave port in the processing system.

Finally, it is necessary to add AXI GPIO as support to the accelerator. The memory mapped GPIO can be used to send control data to the accelerator by writing into memory, in the same way as the DMA. In consequence, the RSA module is internally defined as the following two blocks 15

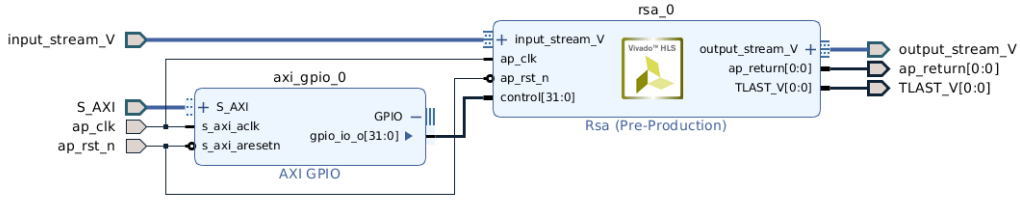

Figure 15: RSA design in Vivado

The *rsa_0* block, is the one exported by Vivado HLS created in the previous section. As it is shown, Vivado maintain the same ports with the same specification than the exported package. We only need to transform the *ap_return* signal into interrupt as explained before to make it work.

After the hardware design is fully specified, we can proceed to synthesise the design. Then, implement the design and write the bitstream. Using this bitstream we can program the FPGA with our custom hardware, including the accelerator. We also export the hardware into a .hdf file. Using this file we can create a support library to implement a test bare metal application.

5.3 RSA Bare metal application

Once we have the final hardware design, we can proceed to create an application to test the hardware implemented in the FPGA. We present a C application for the RSA algorithm. Since it is a bare metal application, first of all, we need to initialize the system. The XSDK environment provides some libraries to initialize the system, but for the registers of the DMA, we need to do it manually. Moreover, we need to link each interrupt to their handler. One for the S2MM interruption, one for the MM2S interruption, and one for the RSA interruption.

After the system is initialized, we can begin with the demo application for the RSA. This is a simple application that shows the encryption and decryption process. The user can choose between these two options, and see the interrupts, state of the memory before using the accelerator, and after using it. For the encryption process, first, we copy the two parts of the key to memory. Then, we divide the original text into blocks of 64 bits, and copy the hexadecimal representation in memory. This is the code to transform the exponent in text representation to their hexadecimal representation, and copy it to memory:

```
for(int i = size - 4; i >= 0; i -= 4) {
   memcpy(tmp,&exp[i*2],8);
   u32 ul = strtoul (tmp, NULL, 16);
   Xil_Out32(baseDDRmm2s + mem_offset, ul);
   mem_offset += 4;
}
```

Size is the number of bytes for the exponent. It must be done in blocks of 32 bits, because of the limitations of the function *strtoul*. Once the input memory is initialized, we can write the DMA registers of *Destination address* and *Length* for the memory mapped to stream channel (MM2S). After writing *Length* register, the transaction begins. Finally, writing the same registers for the stream to memory mapped (S2MM) begins the output transaction to get the result. The process is equivalent for the decryption, but changing the key. The encrypted code is saved as byte representation directly instead of text so we can use it as input for the decryption process. This is the code to enable the DMA transitions:

```
int read_size = size*(n_blocks + 2) + 1;
Xil_Out32(DMA+MM2S_START_ADDRESS, baseDDRmm2s);
Xil_Out32(DMA+MM2S_LENGTH, read_size);

int write_size = size*n_blocks;
Xil_Out32(DMA+S2MM_DESTINATION_ADDRESS, baseDDRs2mm);
Xil_Out32(DMA+S2MM_LENGTH, write_size);
```

All the code for the RSA bare metal application can be found in the appendix B.1, but the initialization of the system. We omit this part of the code because might be too long and it is an overhead that does not help to understand how the accelerator works. This is the example of one possible output for the encryption process and decryption process:

```
Message: Ejemplo
Hex message 0x456A656D706C6F
Public Key 0xE52BEB9D61E0DE7
Private Key 0xCFAB57EE0038D7
Modulus 0x12A231A4A56447F5
## Encrypt
MM2S: interrupt arrived (input)
RSA: interrupt arrived
S2MM: interrupt arrived (output)
Input in memory+0x0: 0xE52BEB9D61E0DE7
Input in memory+0x8: 0x1D1D96CC9FD4BEF
Input in memory+0x10: 0x456A656D706C6F
Output in memory+0x0: 0x12A231A4A56447F5
## Decrypt
MM2S: interrupt arrived (input)
RSA: interrupt arrived
S2MM: interrupt arrived (output)
Input in memory+0x0: 0xCFAB57EE0038D7
Input in memory+0x8: 0x1D1D96CC9FD4BEF
Input in memory+0x10: 0x12A231A4A56447F5
Output in memory+0x0: 0x456A656D706C6F
Message: Ejemplo
```

Chapter 6

Petalinux integration

In this chapter, we will present the Petalinux tools. We will go through the full process to install a Linux distribution in the Zedboard. Then, we will explain how to communicate with our custom hardware using a provided C program by Xilinx called *peek* and *poke*.

6.1 Installation

After the hardware specification and bitstream is generated following the steps in the previous chapter, we can proceed to install a Linux environment using both. First, it is important to check if the requirements to launch a Linux in the custom hardware are enabled in the ZYNQ processing system block. This requirements are [12]:

- Triple Time Controller
- External memory controller with at least 32 MB of memory
- Uart port
- Non-volatile memory (SD/MMC)
- Ethernet (Optional)

Even if the Ethernet option is optional we will need it to connect the board to the internet and deploy the web server in the next chapter. The Non-volatile memory is optional, but we want to boot from SD card, it is possible to use another type of Non-volatile memory such as QSPI Flash.

It is necessary to use the Petalinux tools [22] to build and launch the Linux. To access the tools, we must install Petalinux on our personal computer and indicate their location. This is done by the following command:

```
>$ source <petalinux−instalaltion>/settings.sh
```

After having access to the Petalinux tools. The first step is create a project where the Linux environment will be built. For that, Xilinx provide a command for that, *petalinux-create*. It is necessary to indicate the project which includes the hardware specification wanted on the board using the command *petalinux-config*. In our case, this is the custom hardware created in the previous chapter.

```
>$ petalinux−create −−type project −−template=ZYNQ −name=project
>$ petalinux−config −−get−hw−description=<route>/system.hdf
```

In the first command, the option template indicates the type of the processor where the Linux will run, in our case Zynq from the Zedboard. The option type indicates that it is a project, not an app. In the second command, we specify the hardware description we want to associate with our project. After that, we can build the Linux using *petalinux-build*. Finally, using the generated binaries, we can create a bootable image using the command *petalinux-package*. This is generated in the subfolder *./images/linux*.

```
>$ petalinux−build
>$ cd images/linux
>$ petalinux−package −−boot −−format BIN −−fsbl <FSBL image>
    −−fpga <FPGA bitstream> −−u−boot
```

The FSBL is the First Stage Bootloader. It configures the FPGA if any bitstream is provided and loads the operating system (OS) from non-volatile memory to memory DDR. The FPGA bitstream is the one generated in the previous step that we can use to program the FPGA.

Once the Linux is built we need to prepare the SD card to launch it. Two different partitions are required. The first one must be FAT32, with the boot flag enabled, and at least 60 MBytes. The second partition, must be Linux and formatted as ext4.

```
>$ fdisk /dev/mmcblk0
Device       Boot  Start       End Sectors  Size  Id  Type
/dev/mmcblk0p1 *      2048   133119  131072   64M e  W95 FAT16 (LBA)
/dev/mmcblk0p2      133120 7786495 7653376 3.7G 83 Linux
```

```
>$ sudo mkfs.vfat -F 16 /dev/mmcblk0p1 -n BOOT
>$ sudo mkfs.ext4 /dev/mmcblk0p2 -L rootfs
```

Finally, we just copy the files BOOT.BIN generated using *petalinux-package* and image.ub generated during the *petalinux-build* into the BOOT partition of the SD card. Now it is possible to boot the Linux from it.

6.2 Communication with the accelerator

After the Linux is ready, we need to communicate with the programmable logic. For that we can use the provided programs *peek* and *poke*. First of all, it is necessary to enable them before building the Petalinux using *petalinux-config -c rootfs*. We can see the interface in the next figure 16.

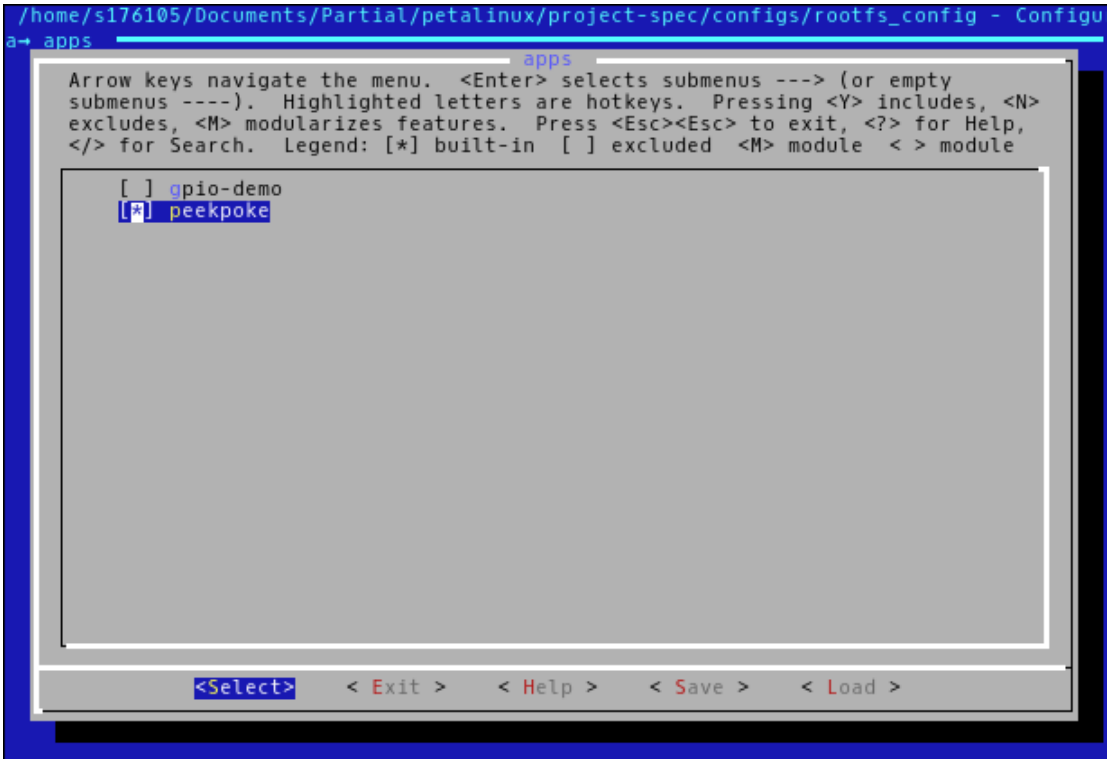

Figure 16: Enable the application peekpoke

These applications work as an interface to send an receive data from memory. They deal with possible situations while reading the memory such as page fault. Using *peek* we can read the content from memory, while using *poke* we can write values in memory. Since the accelerator and the DMA from the previous chapter are memory mapped, we can communicate with them.

An example script performing the encryption for the RSA as the previous chapter is shown below:

```
#/bin/bash

poke 0x43c00000 0x1 #Value 1 to GPIO, it means 1 block of data to encrypt

poke 0x40400000 0x10003 #Init DMA channel MM2S
poke 0x40400030 0x10003 #Init DMA channel S2MM

poke 0xA000000 0xD61E0DE7 #Exponent little endian
poke 0xA000004 0xE52BEB9
poke 0xA000008 0x9FD4BEF #Modulus
poke 0xA00000C 0x1D1D96CC
poke 0xA000010 0x6D706C6F #Message: Ejemplo
poke 0xA000014 0x456A65

poke 0x40400018 0xa000000 #MM2S channel memory mapped
poke 0x40400028 0x19  #MM2s channel length
poke 0x40400048 0xa001000 #S2MM channel memory mapped
poke 0x40400058 0X8   #S2MM channel length
```

After executing the script, we can read the result from memory designed as output using *peek*, and the output is the same than example in the previous:

```
>$ peek 0xa001000
0xa56447f5
>$ peek 0xa001004
0x12a231a4
```

Chapter 7

Partial reconfiguration

In this chapter, we will present how to change the hardware design to enable partial reconfiguration of the accelerators going through the full process. Then, we will explain how to perform the partial reconfiguration in Petalinux and how to install a web server into the board. We will load a web application into the web server that will provide a user-friendly interface to test the full design including the partial reconfiguration.

7.1 Partial reconfiguration workflow

Partial reconfiguration is the ability to modify blocks of logic while the FPGA is still operating [23]. It is possible to save partial bitstream of one region of the FPGA that can be loaded while the rest of the programmable logic is still operating. In our case, it allows us to create a partial reconfigurable region. This region will host either the RSA or the Blowfish algorithm depending on the demands of the system. The DMA, the system blocks, and the processing system are statically configured in hardware. All the partial reconfiguration process can be done in the layout window using the command line. Vivado console works with Tool Command Language (TCL)

Before doing the partial reconfiguration, we need to prepare the hardware design previously presented. The static part of the design remains equal, we only need to switch the module related to the cryptography algorithm to an empty module, called dummy module. This reconfigurable module only defines the interface specification to make the logical connections. Later on, we will be able to insert dynamically our accelerators into it. A high-level representation of the partial reconfiguration performed can be found in the next figure:

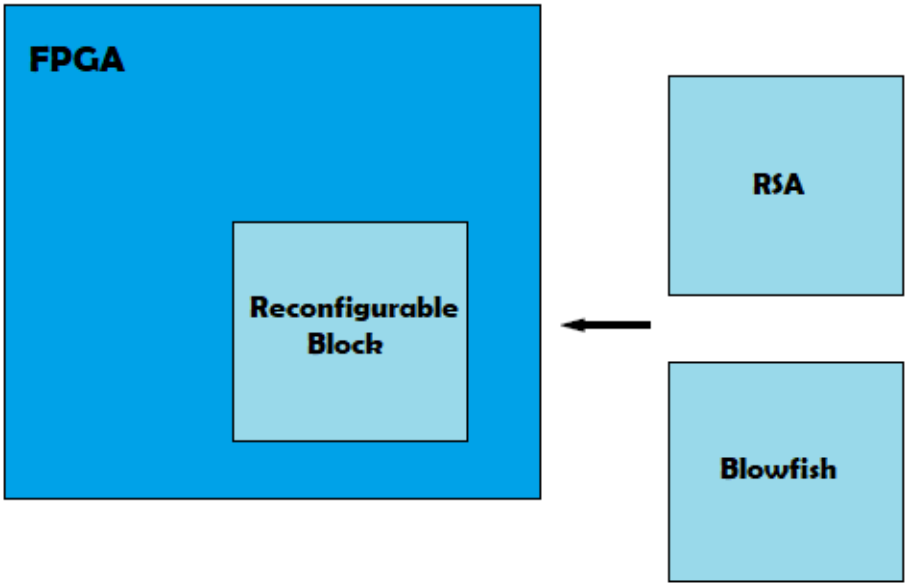

Figure 17: Partial reconfiguration overview

Once we have the new hardware design, we need to prepare the algorithm to fit into this module. The previous module removed from the system was composed of two IPs, AXI GPIO and the accelerator from Vivado HLS. We need to include exactly the same to the partial design. It is important that the ports between the empty module and the algorithm module are equal, including the name. Otherwise, we will have problems when trying to read the algorithm into the new module. The design is equal to the one presented 15 in the hardware integration section.

This partial design can be exported as a checkpoint. This checkpoint can be used in the main Vivado project to be loaded into the reconfigurable module. However, before writing the checkpoint it is necessary to synthesize the partial design using the option *-no_iobuf*. It avoids the default creation of buffers by Vivado when exporting a port.

Instead of running the implementation directly succeeding the synthesis, it is necessary to do some previous tasks in the layout. First of all, the partial reconfigurable region must be specified using a physical block. This pBlock creates a boundary in the layout where the specified block is located. Vivado allows to assign a region graphically as in the following figure 18:

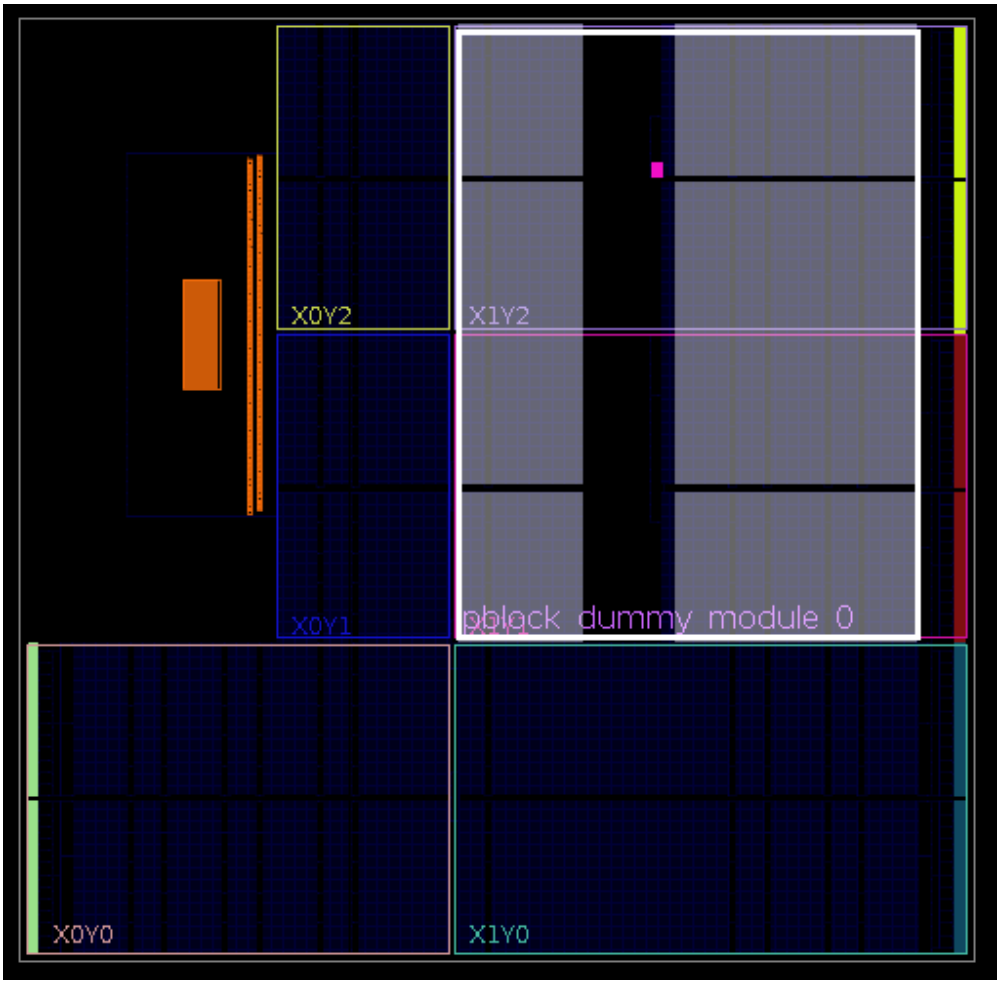

Figure 18: Partial pBlock layout

Apart from the initialization of the pBlock, it is necessary to configure it properly in order to perform the partial reconfiguration. The physical block needs to be specified as reset after configuration, which reset the partial hardware after doing the reconfiguration. Also, the snapping property of the pBlock must be asserted to ON, otherwise we can have problems after loading the bitstream. This is done with the following Vivado console commands.

```
set_property RESET_AFTER_RECONFIG true [get_pblocks pblock_dummy_module_0]
set_property SNAPPING_MODE ON [get_pblocks pblock_dummy_module_0]
```

Moreover, the created pBlock must allocate enough resources to host the accelerators which mainly include DSPs, LUT, and registers. Vivado shows an estimation of the needed resources after reading the checkpoint into the dummy module. We read the RSA algorithm first using the following command:

```
read_checkpoint -cell design_1_i/Sub/dummy_module_0 Partial/rsa.dcp
```

Now Vivado shows the resources consumption for the RSA algorithm. We present a comparison between the RSA and the Blowfish algorithm in the table below 19. We attach both algorithms to compare the resources needed, but the Blowfish will be loaded later on in the workflow.

Physical Resource Estimates		RSA		Blowfish	
Site Type	Available	Assigned	% Util	Assigned	% Util
Slice LUTs	20000	6780	33.90	1082	5.41
LUT as Logic	20000	6762	33.81	1082	5.41
LUT as Memory	6800	18	0.26	0	0.00
Slice Registers	40000	6773	16.93	1539	3.85
Register as Flip Flop	40000	6773	16.93	1539	3.85
Register as Latch	40000	0	0.00	0	0.00
F7 Muxes	10000	23	0.23	0	0.00
F8 Muxes	5000	0	0.00	0	0.00
Block RAM Tile	60	0	0.00	2	3.33
RAMB36/FIFO	60	0	0.00	2	3.33
RAMB18	120	0	0.00	0	0.00
DSPs	80	32	40.00	0	0.00

Figure 19: Resources estimation

The resources for the RSA algorithm are much bigger than for the Blowfish. The reason depends on the type of resource. For the LUT and registers, the RSA needs six times more blocks due to the specification of the for loops. In the Blowfish the number of iterations in the for loops are fixed for all possible inputs, but for the RSA algorithm, the number of iterations depends on the number of bits that compose the key. This uncertainty is translated into more resources in Vivado. On the other hand, Blowfish is based on XORs while the RSA needs to use DSPs to perform the multiplications. Since the area increases quadratically we cannot increase the number of bits for the RSA to 128b, because of the impossibility to draw a pBlock with enough resources.

We can proceed to place and route the design for the RSA algorithm. As a result, we can see the layout of the board with used logic blocks painted in light blue. The painted region inside the pBlock corresponds to the RSA algorithm. This figures 20 shows the post-implementation layout.

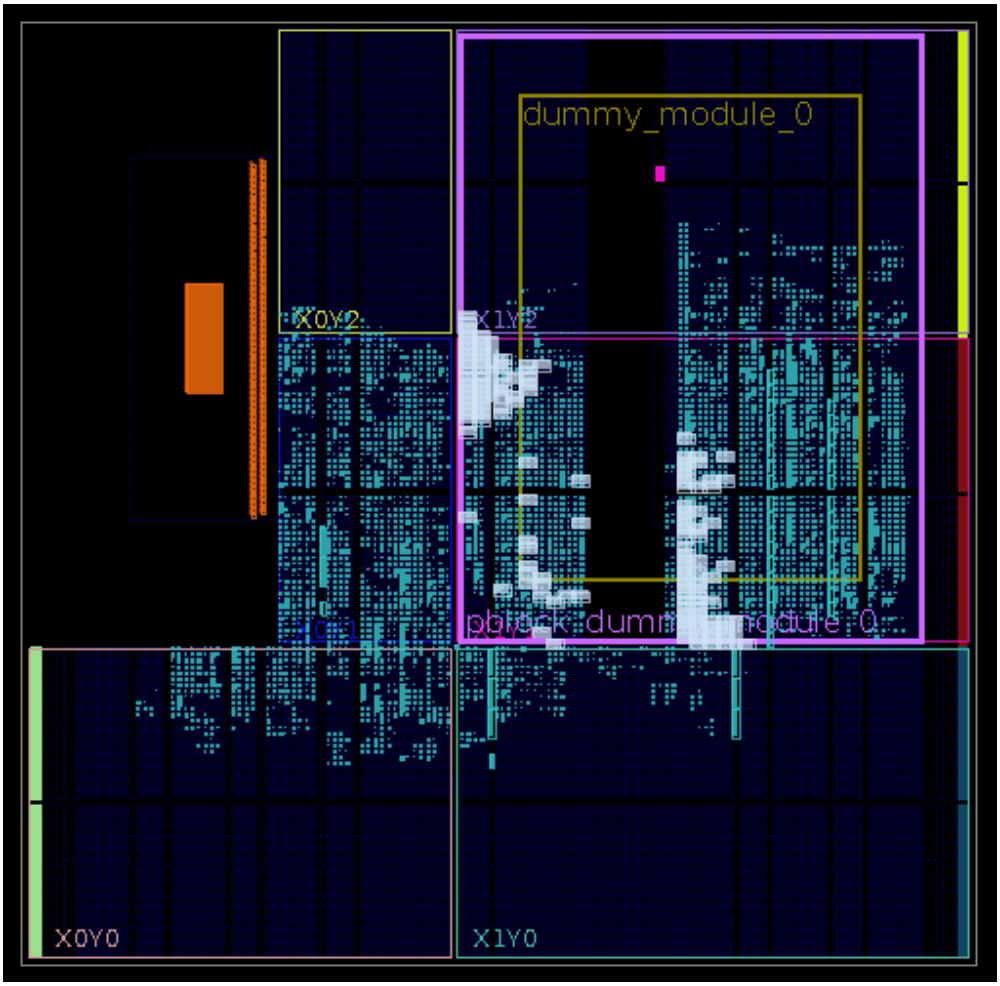

Figure 20: RSA implementation layout

After the implementation, we need to do two more tasks with the RSA. We need to write the full bitstream and create the partial bitstream using the full one. The commands are the following:

```
write_bitstream -file Partial/bitstreams/rsa.bit
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -loadbit "up 0
    Partial/bitstreams/rsa_pblock_dummy_module_0_partial.bit"
    Partial/bitstreams/rsa.bin
```

After that, it is time to proceed with the Blowfish algorithm, but it is important to lock the current static logic before removing the RSA. Otherwise, the new implementation for the second algorithm may destroy the one created by the previous one. These are the commands:

```
update_design -black_box -cell design_1_i/Sub/dummy_module_0
lock_design -level routing
```

The first command removes the RSA algorithm from the dummy module, while the second one locks the static part. Now we can proceed to load the Blowfish into the reconfigurable region, place, and route the design. The commands are analogous to the previous algorithm. The result can be seen in the next figure 21:

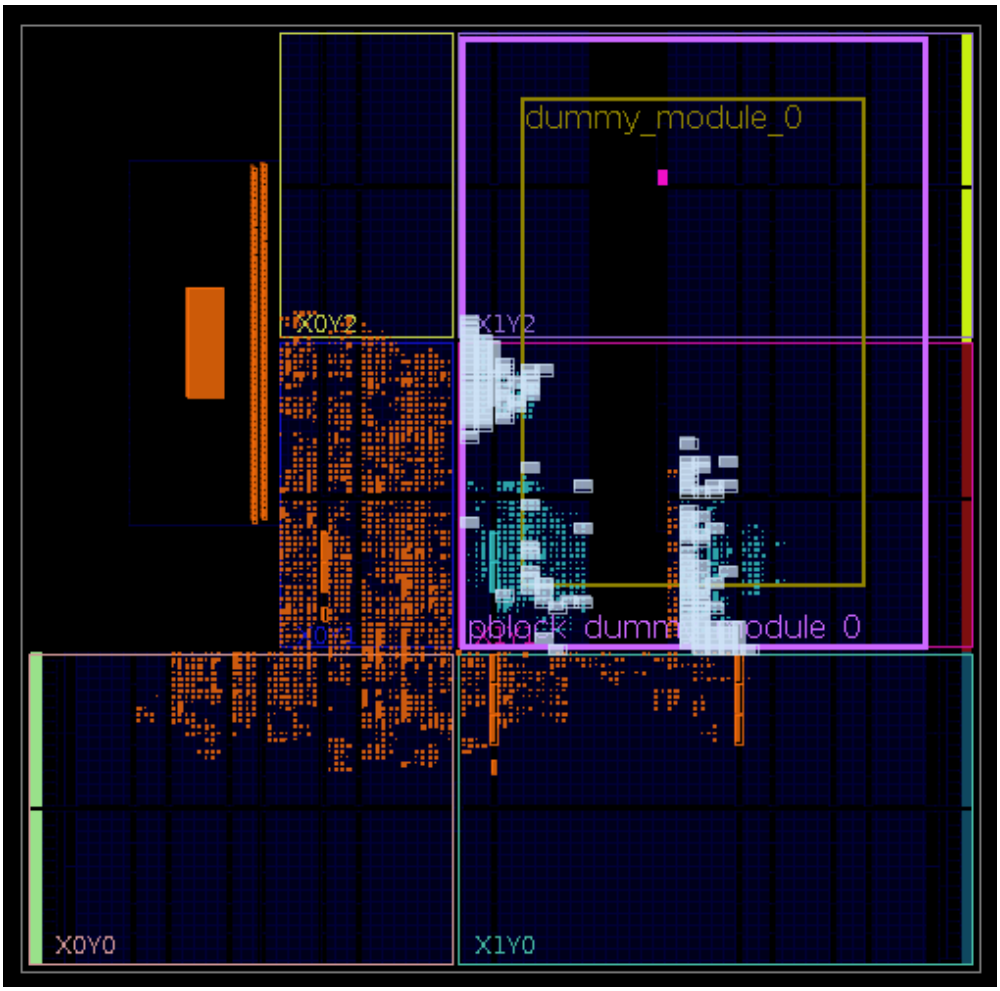

Figure 21: Blowfish implementation layout

The orange blocks that can be seen correspond to the locked static part, while the light blue blocks inside the dummy module correspond to the Blowfish algorithm. After writing the partial bitstream with the previous Tcl commands, it is possible to empty the reconfigurable module and save the bitstream as a blank configuration without algorithms. This blank bitstream can be used as initial configuration to program the FPGA. The problem with the blank bitstream is that some pins are let floating. Before the place and route design steps, we need to add buffers using the following command to avoid floating pins:

```
update_design -buffer_ports -cell design_1_i/Sub/dummy_module_0
```

The full workflow Tcl script can be found in the appendix B.2. It shows explicitly all the steps needed to perform the partial reconfiguration for both algorithms.

7.2 Petalinux

In the previous section, we had explained how to create the partial bitstream to perform the partial reconfiguration. In this section, we will show how to integrate their use in Petalinux. The Linux installed on the board does not need any change to allow this option, the Petalinux already provides a driver to partially configure the FPGA. We only need to copy the blank bitstream and the partial bitstream into the second partition created in the SD card in the previous section.

This driver, Xilinx Device Configuration, is located under the directory */dev* under the name *xdevcfg*. By providing a bitstream to this driver through the standard, it is possible to reconfigure the FPGA. Then, it provides a flag to indicate that the next bitstream provided is a partial bitstream. The full sequence of commands in Linux to program the FPGA with a blank bitstream, and configure the RSA algorithm in the partial block is presented below.

```
echo 0 > /sys/devices/soc0/amba/f8007000.devcfg/is_partial_bitstream
cat /run/media/mmcblk0p2/blank.bit > /dev/xdevcfg
echo 1 > /sys/devices/soc0/amba/f8007000.devcfg/is_partial_bitstream
cat /run/media/mmcblk0p2/rsa.bin > /dev/xdevcfg
```

The flag *is_partial_bitstream* is the one used to inform the driver that the next input is a partial bitstream. Also, the driver provides a flag called *prog_done* which can be checked to be sure that FPGA configuration has been done properly. In the user application, we will perform the partial configuration through the method presented in this section.

7.3 Web Server

The final test for the accelerators will be done using a web interface. In consequence, we need to host it into a web server. Petalinux provides a HTTP daemon which manages the HTTP petitions to the board. This daemon needs to be enabled explicitly using the command *>$ petalinux-config -c rootfs*, and surfing through the GUI until the Busybox options 22. It is located in Filesystem packages, base, Busybox.

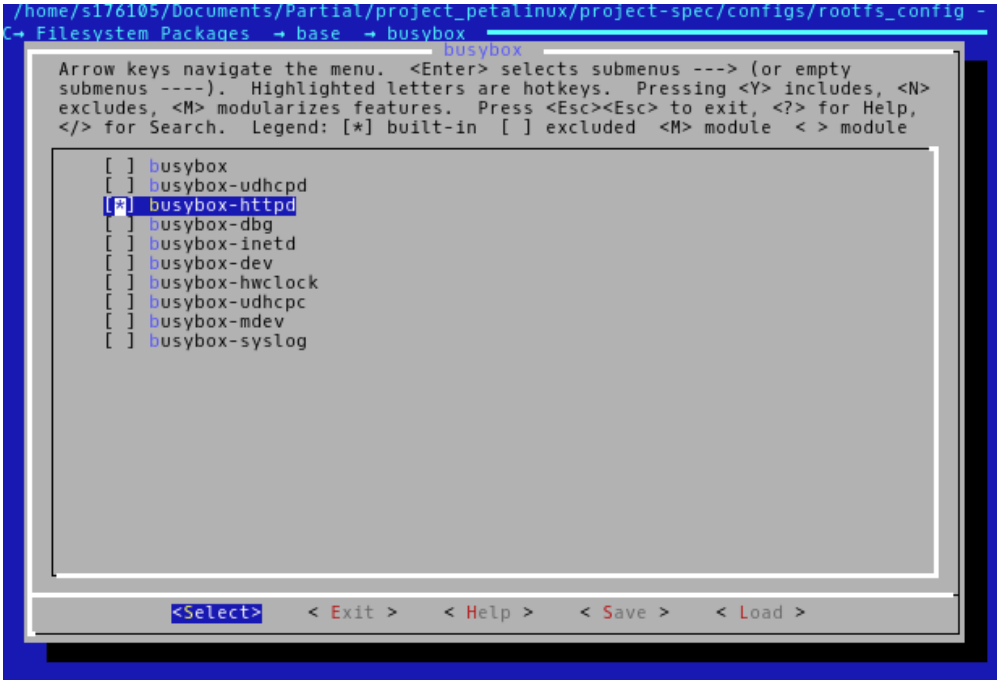

Figure 22: Busybox HTTP Daemon activation

Since the Petalinux is based on Suse, the default web server folder is /srv/www. All the files inside this folder can be accessed using the HTTP protocol and a web browser. In consequence, the future web application must be located there. Due to the temporary file system, If we want to have access to the web server after booting without copying the files manually every time, we need to do the following process. First, we create an application with the name *www* using the following command.

```
>$ petalinux-create -t apps --template install -n www --enable
```

It is not necessary to activate the application manually as with the peekpoke due to the option *enable*. The install template shows an example of how to copy files into the Petalinux file system. This application will be used to copy the applications files to the /srv/www folder in the temporary file system. For that, we modify the configuration file *www.bb*. The appended lines are shown below.

```
S = "${WORKDIR}"

do_install() {
        install -d ${D}/srv
        cp -r ${S}/www ${D}/srv

}

FILES_${PN} += "/srv/*"}
```

Before building the Linux, it is necessary to add the software packages required for the user application presented in the following subsection. These applications are *bc* and *xxd*. The command *bc* is used to do calculus in hexadecimal format, while *xxd* can be used to translate a hexadecimal literal string into ASCII text. The package that needs to be enabled for the first command is the one with the same name. In contrast, the package that hosts *xxd* is vim-common, the one used for the text editor also.

7.4 User application

Once we have the web server ready, and the bitstreams prepared into the SD card, we can proceed to test the user application. The demo application provides a user-friendly way to test the accelerators. By using a web interface, the user can change between the different configurations, RSA, Blowfish, or empty. Then, the user can write a raw text that is encrypted and decrypted using the configured accelerator. The result is presented in two boxes to check if the decrypted text is equal to the original raw text, it is possible to see the encrypted text also.

The user application needs the use of CGI scripts to communicate with the board. These cgi scripts are executable programmes that are executed on the server side. The CGI binaries must be installed inside a subfolder called cgi-bin with execution permission, and using the suffix .cgi. Otherwise, the HTTP daemon cannot find them. In our case, these CGI scripts are written in BASH shell. These scripts are used to do operations in the board and generate the HTML of the resulting web page explicitly.

The first web page when accessing the web server using a web browser is an introduction. In this page, a simple button is presented that starts the demo application on click. The introduction web page can be seen below 23:

Figure 23: Introduction web page

When the user clicks on the button, the server executes the initialisation script. This script programs all the board with an empty bitstream, and loads a blank partial bitstream into the reconfigutable blocks. Then, it sets the initial configuration to BLANK writing into a configuration file. This file will be used by the encryption script later on. Finally, it generates the HTML for the main page.

The HTML is created directly using the standard output, for that it is necessary to specify that the text provided is an HTML we page. In consequence, the first line of all the cgi scripts when generating the next web page must be the one presented below. After that, the following outputs from the script are treated as HTML code.

```
echo "Content-type: text/html"
echo ""
```

The web page generated is the main page of the demo application. This page provides a user-friendly web interface to change between the different algorithms, and cipher a text. First of all, the web page provides a paragraph explaining how it works, and a supporting picture. The main page can be divided into two sections, algorithm selection, and encryption. The web page can be seen in the following figure 25.

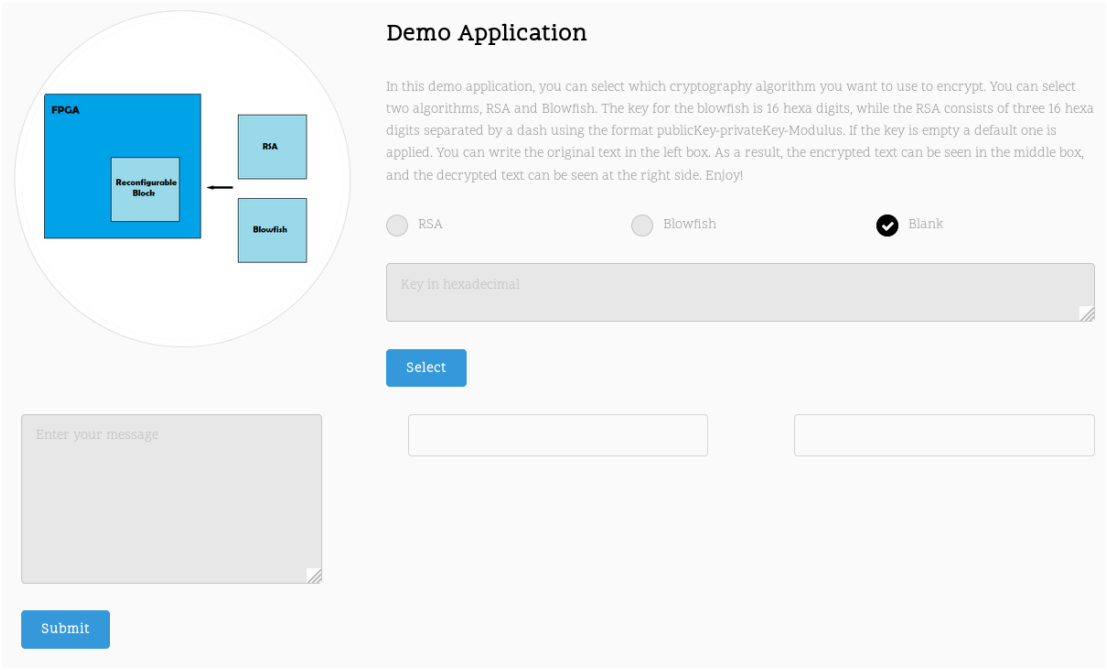

Figure 24: Main web page

In the first section, the user can change between RSA, Blowfish, and empty partial configuration. Besides the algorithm selection, the user can type a key for both algorithms. For the Blowfish the key is sixteen hexadecimal bits, while for the RSA the key consist in three strings of sixteen hexadecimal bits separated by a dash between them. The keys must be written in the following order: public exponent, private exponent, and modulus. If no key is provided, the server provides one.

When clicking on the button "Select", the server executes another CGI script to make the changes to the board. First, the script has to read the value from the form. The form uses the method GET, so the parameters can be read in bash using the *QUERY_STRING* environment variable. Using the following code, the input values are saved in a variable with the name: var_<parameter_name>.

```
#Read QUERY_STRING
saveIFS=$IFS
IFS='=&'
parm=($QUERY_STRING)
IFS=$saveIFS

for ((i=0; i<${#parm[@]}; i+=2))
do
    declare var_${parm[i]}=${parm[i+1]}
done
```

```
#Example:
#QUERY_STRING=algorithm=2&message=1234567890abcdef
#var_algorithm -> 2
#var_message -> 1234567890abcdef
```

The comments below the code show an example QUERY_STRING input, and the stored values in the variables. The script has two main functions, perform the partial configuration with the selected algorithm by the user, and sanitize the input key.

If the key is provided by the user, the server needs to ensure that the user key will not generate a problem. For that, first, we check if the if the length of the key is correct, and second if it is written using a literal hexadecimal string. The second check is done using regular expressions. It is possible to check the length and the characters in the same regular expressions, but two different checks are used in order to provide proper feedback to the user. When there is a problem with the key, it is presented to the user as a red message. The checks for the Blowfish can be seen in the next piece of code:

```
if [ "$var_algorithm" -eq "2" ] && [ "$size" -ne "0" ]; then
   if [ "$size" -ne "16" ]; then
      error="Blowfish key must be 16 digits"
   else
      if ! [[ "$var_message" =~ ^[0-9A-Fa-f]{16} ]]; then
         error="Blowfish key must be hexadecimal digits"
      fi
   fi
fi
```

The regular expression checks if all the characters are decimal digits, and letters from a to f including uppercase. Also, ensures that the length of the key is exactly sixteen. If the key is empty, or the one provided is correct, the script performs the partial reconfiguration. It writes the partial bitstream into the driver as presented before, and update the configuration file used by the encryption with the new configuration. The key is added to the configuration file only if the user provides one. An example for the Blowfish is presented below:

```
echo "BLOWFISH" > .config
cat /run/media/mmcblk0p2/blowfish.bin > /dev/xdevcfg
msg="Blowfish algorithm selected"
if [ "$size" -ne "0" ]; then
   echo "$var_message" >> .config
fi
```

If the partial reconfiguration is performed properly, a message is shown to the user. Before generating the web page, it is important to reset the DMA after the reconfiguration. If not done, the DMA will hang after the first transaction.

The second section of the main web page performs the encryption and decryption of a given text by the user. It reads the text from the form in the same way as the previous script and reads the configuration file to know the type of encryption performed. It is important because the RSA has three subkeys, while the Blowfish has one unique key. If the text is empty, or no algorithm is configured when clicking the "Submit" button, the boxes present a default text.

The same approach is followed to send the data to the accelerator than in the previous chapter. Using the command *poke* the script writes the data into a memory mapped region that the DMA will send to the accelerator. For that, first, it is necessary to transform the input text into hexadecimal format. Then, the script iterates the string dividing it into blocks that can be sent to the accelerator.

It is important to distinguish between three different cases when dividing into blocks. The general case when there are enough data to fill a full block. When there is data to fill only half of a block, so the other half must be set to zeroes. Finally, when there is data to fill half of a block and part of the other half. The last case is necessary because of the way that BASH works with substrings. At the same time, the number of blocks must be stored in order to configure the DMA and the accelerator. Part of the code is shown below:

```
hex_msg=$(echo -n "$var_message" | hexdump -v -e '/1 "%02X "' | tr -d ' ')
size=${#hex_msg}
it="$size"
while [ "$it" -gt "0" ]; do
   ...
   tmp=$(($it - 8))
   poke 0x$MM2S_addr 0x${hex_msg:$tmp:8}
   MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
   tmp=$(($tmp - $N))
   poke 0x$MM2S_addr 0x${hex_msg:$tmp:$N}
   poke 0x$MM2S_addr 0x$value
   MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
   it=$tmp
   ...
   n_blocks=$(echo "obase=ibase=16;$n_blocks+1" | bc)
done
```

For simplicity, it is shown only in the code related to the general case when dividing the blocks. The text transformation to hexadecimal is done using the command *hexdump* configuring the output as parts of one byte. This is done to avoid problems of endianness. Then, the loop iterates the hexadecimal string from the end to the beginning. It is important because when writing a 64 bits block, the second 32 bits half of the block must be in the lesser position. Finally, the memory offset is increased by four, because the data is written in blocks of four bytes, 32 bits. To perform hexadecimal operations we use the command *bc*. It was installed in the previous section. Finally, the number of characters copied in the upper 32 bits half change depending on the algorithm. For the Blowfish, we can fill the half block, but for the RSA, we preserve the first 8 bits as zeroes to avoid problems due to the length of the key. Because of the property of the RSA where the block must be lesser than the module.

After the data is ready, the DMA transactions are activated in the same way as the previous chapter. The resulting encrypted text is copied from the output memory to the input memory, printing in the web interface the encrypted text. Finally, the DMA transactions are activated again in order to get the decrypted text in the output memory. This decrypted text is printed in the web interface too. The exampling bash code that gets the decrypted from the output memory is shown below.

```
it=$(echo "ibase=16;$n_blocks*2" | bc)
tmp=$((($it - 1) * 4))
tmp=$(echo "obase=16; $tmp" | bc)
S2MM_addr=$(echo "obase=ibase=16;$S2MM_base_addr+$tmp" | bc)
while [ "$it" -gt "0" ]; do
   output=$(peek 0x$S2MM_addr)
   hex=$(echo "0x${output:2:32" | xxd -r)
   decrypted="$decrypted$hex"
   S2MM_addr=$(echo "obase=ibase=16;$S2MM_addr-4" | bc)
   it=$(($it - 1))
done
```

The number of iterations is twice the number of blocks due to each block of 64 bits is divided into two blocks of 32 bits. Since the data is previously stored in reverse order, we need to read it back in reverse order too. Moreover, the text is saved in hexadecimal format, so it needs to be converted to ASCII text again. For that purpose, the command *xxd* installed in the previous section can be used. Using the option *r*, the command gets a hexadecimal input and returns ASCII text.

Finally, a possible use for the web application is shown below. In the example, the user provides their own RSA key and writes a text in the left text area. The encrypted text can be seen in the box located in the middle, while the decrypted version from the middle text can be seen in the right box.

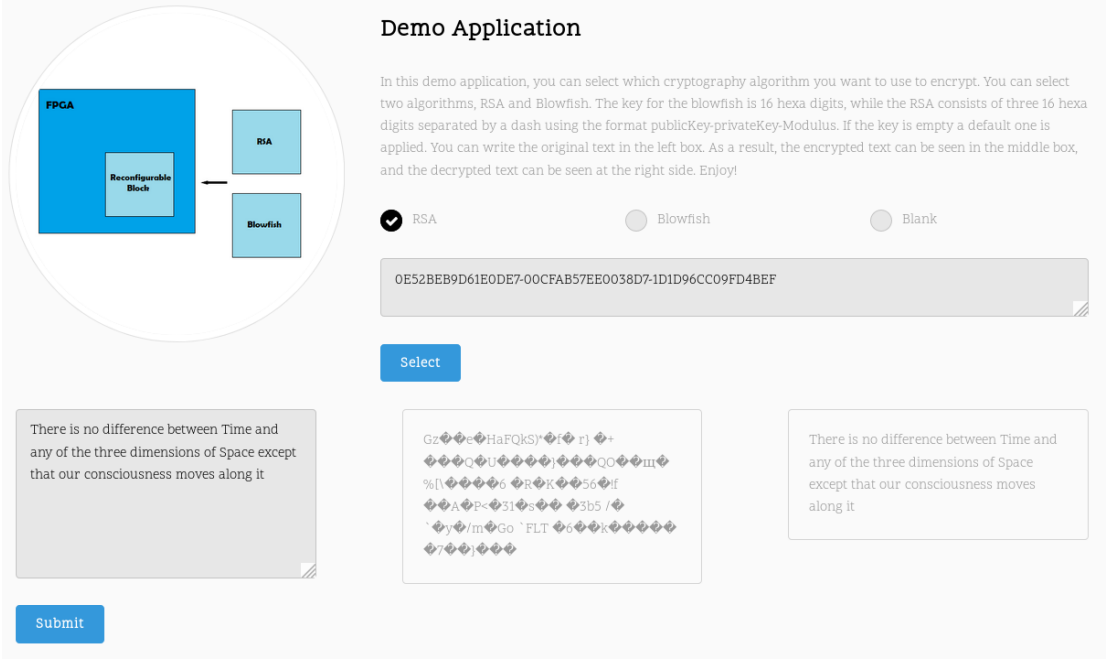

Figure 25: Main web page test

All the HTML that composes the web pages can be seen in the appendix B.3. While the three CGI scripts presented used to operate the FPGA can be found in the appendix B.4

Chapter 8

Delay and Power

In this chapter, we will present the measurements of delay and power for the RSA and the Blowfish algorithm. Moreover, we will explain how to take these measurements precisely using the previous bare metal and Linux application presented for the delay and power respectively.

8.1 Delay measurements

There are different ways to measure the delay of the application. It is possible to add an AXI Timer [24] to the hardware design in order to measure the clock in the programmable logic. Another approach measures the time in the Linux application using *time*, but it is really hard to be precise due to the overhead when calling this command. Finally, the approach followed in this project is based on a time library provided by Xilinx. Since in the Bare metal application there are not operative systems that support the software, it is not possible to use the "time.h" library in C. However, Xilinx provides a library [25] that uses the clock in the SoC Zynq 7000, called "xtime_l".

This library provides structures, macros, and functions to measure the time. The clock in the SoC works with a frequency of 667MHz, and the time counter provided by the library increment by one after two clock cycles. In consequence, the counter of ticks can be used to calculate the number of cycles. The number of cycles can be used to calculate the time using the macro "COUNTS_PER_SECOND", which is the ticks of cycles in a second. An example code showing how to measure is shown below. The code prints the number of cycles, and the measured time in microseconds:

```
#include "xtime_l.h"

...

XTime tStart, tEnd;

XTime_GetTime(&tStart);
//Code to measure
XTime_GetTime(&tEnd);

printf("Output took %llu clock cycles.\n", 2*(tEnd - tStart));
printf("Output took %.2f us.\n", 1.0 * (tEnd - tStart) /
    (COUNTS_PER_SECOND/1000000));

...
```

We can use this to measure the time of our accelerators. We want to measure the delay since we send the data until we receive the result. As a result, we need to start the clock when the DMA sends the data to the accelerator, and stop it when it returns the result back. For that purpose, we can use the interruption handlers. The delay for both accelerators can be seen in the following table. We duplicate the number of blocks sent to the accelerator in each test. We begin from 32 bytes of data until 1024 bytes of data.

Bytes	RSA Delay [us]	Blowfish Delay [us]
32	140.71	367.89
64	234.61	370.78
128	423.19	376.33
256	799.78	387.50
512	1553.34	409.76
1024	3059.85	454.81

Table 7: Accelerator delay

Since we know the delay, we can calculate the number of cycles that the FPGA needs to do the full calculus. In the Zedboard, the FPGA has a clock frequency of 100MHz. which is translated into a clock cycle of 10ns. The resulting clocks can be seen in the next table.

Bytes	RSA Clock cycles	Blowfish Clock cycles
32	14071	36789
64	23461	37078
128	42319	37633
256	79978	38750
512	155334	40976
1024	305985	45481

Table 8: Accelerator delay

These clock cycles include the time that the accelerator needs to perform the full operation plus the time spent by the DMA to send the data back. In the following graph, we can see the tendency of the delay for both algorithms 26.

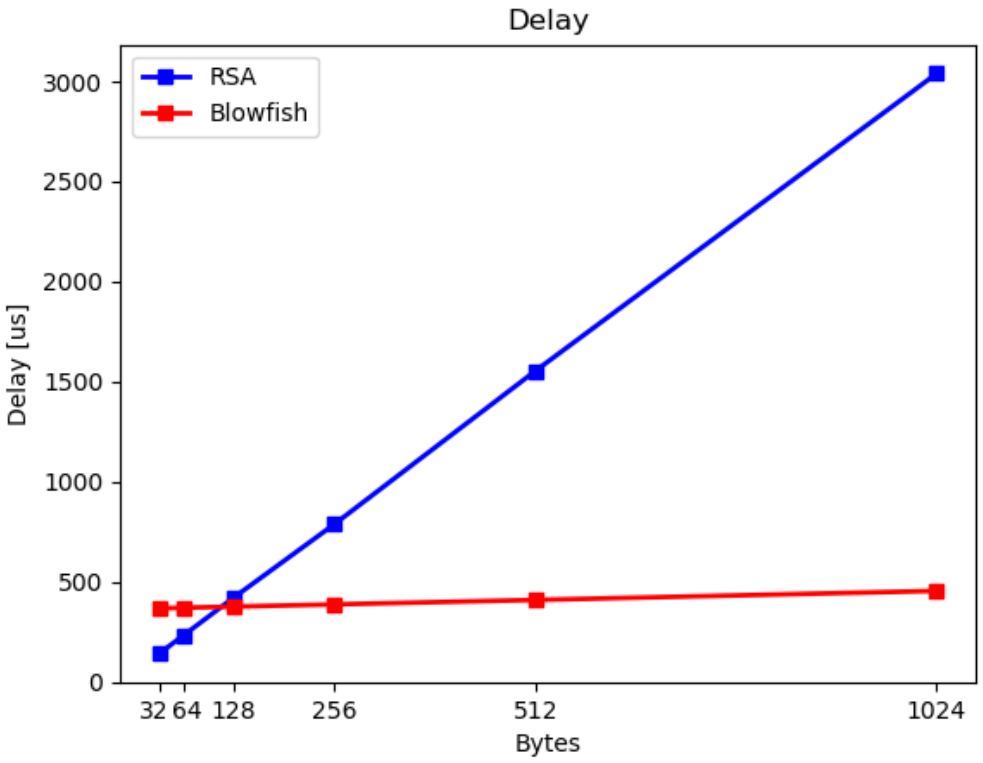

Figure 26: Delay tendency accelerator [us]

In the graph above, we can see that the RSA follows a constant slope while the Blowfish maintains almost the same delay. The reason for the RSA is that the precomputation has a really low impact on the delay. In consequence, the delay for the accelerator is linear to the number of blocks that it must encrypt. However, for the Blowfish, the main drawback of the algorithm is the initialization of the key. This graph shows that the difference when increasing the number of blocks is low. It means that the delay is lead by the initialization for the key.

The delay for the RSA is higher than for the Blowfish as expected, and this gap keeps increasing with the number of bytes. This is one of the reasons why asymmetric cryptography is used to perform the key exchange, but the symmetric cryptography is used during the communication.

After timing the accelerator, we want to compare it with the execution time in the SoC. For that, the same algorithm that the one in the accelerator is executed in the SoC. Moreover, it is done in the bare metal application following the previous approach in order to be as precise as possible. The results for both algorithms are the following.

Bytes	**RSA Delay [us]**	**Blowfish Delay [us]**
32	1133.8	23172.24
64	2140.41	23355.1
128	4194.15	23722.95
256	8299.6	24449.3
512	16514.91	25908.21
1024	32937.96	28831.16

Table 9: SoC delay

In general, the timing increases for all the sizes for both algorithms. The delay for the RSA depends on the number of blocks, while the delay for the Blowfish is dominated by the initialization of the key. Unlike the previous case, the Blowfish is slower than the RSA for sizes lower than 1024 bytes. The reason is how the initialization of the key is performed. It needs to work with four kilobytes of text which is allocated in memory. Accessing this memory from the SoC delays the execution time. This is the main drawback for the Blowfish, It can be seen graphically in the following graph 27.

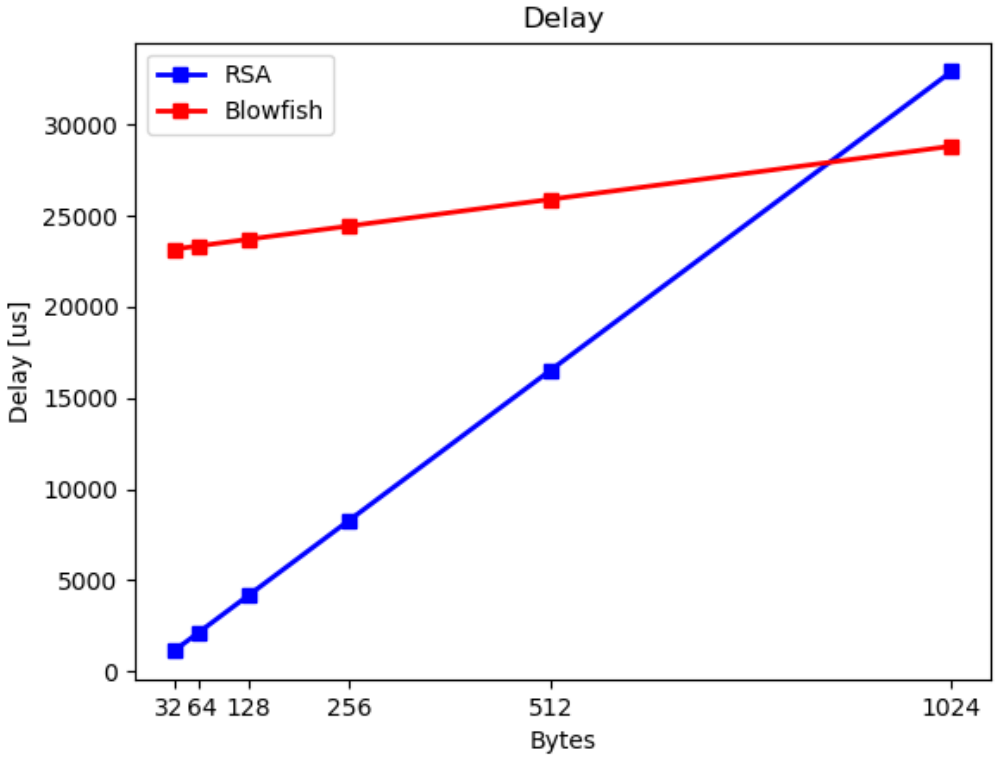

Figure 27: Delay tendency SoC [us]

The cost of initializing the key makes the Blowfish algorithm less likely to be used than other symmetric algorithms. We can see the speedup of the accelerator compared to the SoC in the following table. We use the timing values for 1024 bytes.

	RSA	**Blowfish**
Accelerator	3059.85	454.81
SoC	32937.96	28831.16
Speed-up	~10.7	~63.3

Table 10: SoC delay

The speedup for the Blowfish is approximately sixty-three times faster, because of the initialization of the key is performed much faster in the FPGA than in the SoC which needs to access to memory. On the other hand, the speed up for the RSA is approximately 10 times faster. The improvement is not as big as in the Blowfish due to the number of multiplication. The SoC can perform the multiplication in a functional unit while the FPGA needs to use a big amount of DSPs. Moreover, the small size of the key allows the processor to operate with the data directly. Even so, the accelerator is faster than the SoC.

8.2 Power measurements

The Zedboard is provided with a 10mΩ, 1W current sense resistor in series with the 12V input power supply that can be used to measure the voltage on the board. The shunt-register is connected in parallel with the jumper J21 located near to the audio I/O peripherals. It is possible to measure the voltage drop across the resistors using this header J21. Then, the power can be calculated using the following equation:

$$P = \frac{mV}{10m\Omega} 12V$$

The input voltage can be measured using a multimeter. In our case, we have used the 34461A Digital Multimeter [26]. Using this device, it is possible to measure the voltage. The installed firmware provides a web interface with a java application that can be used to control the multimeter. It allows the user to start and stop the measurement, and print the data read. Moreover, it is possible to change the number of measures per second to adjust the precision.

We use a BASH script to measure the power on the board during the activity. First, it copies all the data required to the input memory mapped and configures the whole FPGA reconfiguring the partial hardware as blank. Then, it performs the partial reconfiguration for the RSA, performs an encryption, and reconfigures the partial hardware as blank. Finally, it performs the partial reconfiguration for the Blowfish, performs an encryption, and reconfigures the partial hardware as blank.

The following graph 28 shows the behavior during the explained phases of the script. After the whole reconfiguration the power consumption descends, but after the first partial reconfiguration and encryption process, it maintains a bit higher, even if reconfiguring again to the blank partial bitstream. This is because the usage of the DDR3 memory for the first time in the encryption process.

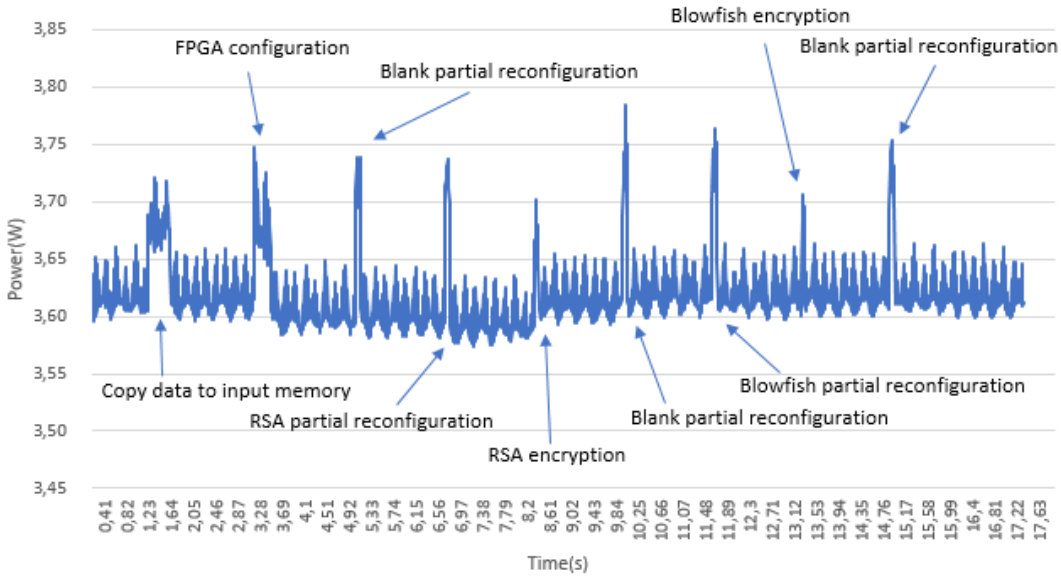

Figure 28: Power measuraments for the board

The graph shows that the power consumption maintains equal independently of the partial bitstream configured. It increases after the first encryption. The voltage measured and the power calculated can be seen in the next table. The table shows the average measurements depending on the status of the board. All the measurements are done without the Ethernet cable connected.

STATUS	v (mV)	Power (W)
IDLE	2.02	2.42
PS init before first encryption	2.98	3.58
PS init after first encryption	3.04	3.65
Copying data to memory	3.08	3.70
Encryption	3.08	3.70
FPGA configuration	3.13	3.76
Partial reconfiguration	3.12	3.74

Table 11: Voltage measurements

It can be inferred from the table, that most of power consumption depends on the PS and PL static part.

Chapter 9

Conclusion

In this thesis, we have presented a design halfway between software and hardware to implement cryptography algorithms that can be dynamically reconfigured. We explained all the steps to implement and integrate into the board one example for each type of cryptography, asymmetric and symmetric.

Furthermore, we have explained step by step the full process. Beginning from the software implementation of the accelerators, following the hardware design, and finishing in the possible outcomes to test the design. The two solutions presented to test the design are the bare metal application, which can be used to debug, and the web interface integrated in a Linux, which provides a user friendly interface.

We have demonstrated that this solution can be very interesting for communications. It is possible to perform the key exchange using the RSA algorithm in hardware. Then, once the key is agreed between the client and server, reconfigure to the Blowfish algorithm. If the connection is lost, it is possible to reconfigure to the RSA to perform the key exchange and recover as soon as possible.

The entire experience of this project leads to different results. First of all, the use of a FPGA includes most of the advantages of a Application-specific integrated circuit [2], but reducing the time and complexity related to the design and implementation of this circuits. Moreover, the use of advanced tools such as the provided by Xilinx reduce the knowledge needed by the programmer to implement the wanted design.

Secondly, when creating an accelerator two possible approaches can be followed, design the specification of the hardware or use a high level synthesis tool such as Vivado HLS. In our case, we have followed the second approach. The main advantage of this method is the possibility to use high level code such as C, but has some drawbacks. It is not possible to control the full process. For example, in the area problem, Vivado HLS allows to specify the allocation of some resources, and reduce the number of instances of a core. However, the final result can avoid these specifications.

This thesis is open to further improvements. Since this is a test application, the size of the key for the RSA is not the ideal. Furthermore, if we want the launch application into the actual market the first improvement is the size of the key. The problem of doing this is the area needed for the accelerator due to the quadratic increase in the area when increasing the number of bits. A possible solution comes from using a bigger FPGA with more resources, or try to serialize the execution using Vivado HLS.

Moreover, the execution time can be improved in bigger sizes of the key dividing each block into lesser words. In consequence, the encryption process can be performed in separated words in parallel, but it requires more resources. If the domain of the application executed is known, it is possible to reduce drastically the area needed by fixing the number of bits in the key. This removes the uncertainty in the number of iterations.

On the other hand, nowadays the most used symmetric algorithm is the Advanced Encryption Standard [27]. A possible improvement for the system could be adding AES in the place of the Blowfish. This algorithm requires that the data comes in blocks of 128 bits, which may differ with the number of bits for the RSA.

Accelerators C code

A.1 RSA accelerator

```
bit64 monPro(bit128 a, bit128 b, bit64 n, bit128 r, bit8 k, bit64 c_n) {
   bit128 t = a * b;
   bit64 r_1 = r - 1;
   bit128 tmp_m = t * c_n;
   bit128 m = tmp_m & r_1;
   bit128 tmp_u = m * n;
   bit128 tmp_u2 = t + tmp_u;
   bit128 u = tmp_u2 >> k;
   if(u >= n) return u - n;
   else return u;
}

bit64 modInverse(bit128 a, bit64 n) {
   sbit64 n0 = n;
   sbit64 y = 0, x = 1;
   if(n == 1) return 0;
   while(a > 1) {
      sbit64 q = a / n;
      sbit64 t = n;
      n = a % n, a = t; t = y;
      y = x - q * y; x = t;
   }
   if(x < 0) x+= n0;
   return x;
}

bit64 highestOneBit(bit64 k) {
   uint16_t count = 0;
   while (k>0) {count++; k>>=1;}
   return count;
}


bit rsa(uint32_t control, bit64 input_stream, bit64 * output_stream, bit *
    TLAST) {
```

```
//Interface
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS interface ap_none port=control
#pragma HLS INTERFACE axis port=output_stream
#pragma HLS INTERFACE axis port=input_stream
#pragma HLS interface ap_none port=TLAST

//Control
static bit exp_done = 0;
static bit n_done = 0;
static bit crypt_done = 0;
static uint32_t n_blocks = 0;

//Variables
static bit64 n = 0;
static bit64 exp = 0;
static uint8_t k;
static bit128 c_n;
static bit128 r;
*TLAST = 0b0;

if (control == 0x00000000) {return 0;}
else if (exp_done == 0) {
   //Expontent and initialise number of blocks
   exp = input_stream;
   n_blocks = control;
   exp_done = 1;
   return 0;
} else if (n_done == 0) {
   //Modulus and Precalculation
   n = input_stream;
   k = highestOneBit(n);
   r = 1; r = r << k;
   bit128 inv_r = modInverse(r,n);
   bit128 tmp_c_n = inv_r * r;
   c_n = (tmp_c_n - 1) / n;
   n_done = 1;
   return 0;
} else if (crypt_done == 0) {
   //Encrypt/Decrypt
   bit64 mask = r >> 1;
   bit128 rem_m = input_stream * r % n;
   bit128 rem_x = r % n;
   int i;
   for(i = k - 1; i >= 0; i--) {
      rem_x = monPro(rem_x,rem_x,n,r,k,c_n);
      if(exp & mask) rem_x = monPro(rem_m,rem_x,n,r,k,c_n);
      mask >>= 1;
   }
```

```
      //Output
      *output_stream = monPro(rem_x,1,n,r,k,c_n);
      n_blocks = n_blocks - 1;
      if(n_blocks == 0) {crypt_done = 1; *TLAST = 0b1; return 1;}
      return 0;
   } else {
      //Reset
      exp_done = 0;
      n_done = 0;
      crypt_done = 0;
      n_blocks = 0;
      return 0;
   }
   return 0;
}
```

A.2 Blowfish accelerator

```
#define N 16
bit32 P_ARRAY[16 + 2] = {..};
bit32 S_BOXES[4][256] = {..};

bit32 f_function(bit32 input) {
   bit8 a, b, c, d;
   bit32 output;
   d = (bit8)(input & 0xff); input >>= 8;
   c = (bit8)(input & 0xff); input >>= 8;
   b = (bit8)(input & 0xff); input >>= 8;
   a = (bit8)(input & 0xff);
   output = S_BOXES[0][a] + S_BOXES[1][b];
   output = output ^ S_BOXES[2][c];
   output = output + S_BOXES[3][d];
   return output;
}


void Blowfish_Encrypt(bit32 *left, bit32 *right){
   bit32 left_tmp, right_tmp, aux;
   left_tmp = *left;
   right_tmp = *right;

   for (bit8 i = 0; i < N; ++i) {
      left_tmp = left_tmp ^ P_ARRAY[i];
      right_tmp = f_function(left_tmp) ^ right_tmp;
      /* Exchange left_tmp and right_tmp */
      aux = left_tmp; left_tmp = right_tmp; right_tmp = aux;
   }
```

```
   /* Exchange left_tmp and right_tmp */
   aux = left_tmp; left_tmp = right_tmp; right_tmp = aux;

   right_tmp = right_tmp ^ P_ARRAY[N];
   left_tmp = left_tmp ^ P_ARRAY[N + 1];
   *left = left_tmp;
   *right = right_tmp;
}


void Blowfish_Decrypt(bit32 *left, bit32 *right){
   bit32 left_tmp, right_tmp, aux;
   left_tmp = *left;
   right_tmp = *right;

   for (bit8 i = N + 1; i > 1; --i) {
      left_tmp = left_tmp ^ P_ARRAY[i];
      right_tmp = f_function(left_tmp) ^ right_tmp;
      /* Exchange left_tmp and right_tmp */
      aux = left_tmp; left_tmp = right_tmp; right_tmp = aux;
   }

   /* Exchange left_tmp and right_tmp */
   aux = left_tmp; left_tmp = right_tmp; right_tmp = aux;

   right_tmp = right_tmp ^ P_ARRAY[1];
   left_tmp = left_tmp ^ P_ARRAY[0];
   *left = left_tmp;
   *right = right_tmp;
}


void Blowfish_Init(bit64 key) {
   bit32 data;

   bit64 aux = key;
   for (int i = 0; i < N + 2; ++i) {
      data = 0;
      for (int k = 0; k < 8; ++k) {
         data = (data << 4) | (aux & 0xf); aux >>= 4;
         if (aux == 0) aux = key;
      }
      P_ARRAY[i] = P_ARRAY[i] ^ data;
   }

   bit32 datal = 0, datar = 0;
   for (int i = 0; i < N + 2; i += 2) {
      Blowfish_Encrypt(&datal, &datar);
```

```
      P_ARRAY[i] = datal;
      P_ARRAY[i + 1] = datar;
   }

   for (int i = 0; i < 4; ++i) {
      for (int j = 0; j < 256; j += 2) {
         Blowfish_Encrypt(&datal, &datar);
         S_BOXES[i][j] = datal;
         S_BOXES[i][j + 1] = datar;
      }
   }
}

/**
 * Control:
 *  First bit: indicates if it is necessary to initialize the key, 0 yes, 1 no
 *  Second bit: indicates if need to perform encryption or decryption, 0 en, 1
     de
 *  Remainder bits: number of blocks
 */

bit blowfish(bit32 control, bit64 input_stream, bit64 * output_stream, bit *
    TLAST) {

   //Interface
   #pragma HLS INTERFACE ap_ctrl_none port=return
   #pragma HLS interface ap_none port=control
   #pragma HLS INTERFACE axis port=output_stream
   #pragma HLS INTERFACE axis port=input_stream
   #pragma HLS interface ap_none port=TLAST

   //Control
   static bit init_done = 0;
   static bit crypt_done = 0;
   static bit32 n_blocks = 0;
   static bit32 encrp_decrp = 0; //0 Encryption, else Decryption
   static bit32 perform_init = 0; //Do key init, else key already initialized

   //Variables
   *TLAST = 0b0;
   perform_init = control & 0x80000000;

   if (control == 0x00000000) {return 0;}
   else if (init_done == 0 && perform_init == 0) {
      //Init the key
      n_blocks = control & 0x3fffffff;
      encrp_decrp = control & 0x40000000;
      Blowfish_Init(input_stream);
      init_done = 1;
```

```
  } else if (init_done == 0 && perform_init != 0) {
     //Init the control variables, but the key
     n_blocks = control & 0x3fffffff;
     encrp_decrp = control & 0x40000000;
     init_done = 1;
  } else if (crypt_done == 0) {
     //Encrypt/Decrypt
     bit32 left = (input_stream & 0xffffffff00000000) >> 32;
     bit32 right = input_stream & 0xffffffff;
     if(encrp_decrp == 0)
        Blowfish_Encrypt(&left,&right);
     else
        Blowfish_Decrypt(&left,&right);
     bit64 result = left; result <<= 32;
     result |= right;
     *output_stream = result;
     n_blocks = n_blocks - 1;
     if(n_blocks == 0) {crypt_done = 1; *TLAST = 0b1; return 1;}
     return 0;
  } else {
     //Reset
     init_done = 0;
     crypt_done = 0;
     encrp_decrp = 0;
     perform_init = 0;
     n_blocks = 0;
     return 0;
  }
  return 0;
}
```

Applications

B.1 Bare metal application C

*Initialisation functions and auxiliary functions are avoided to simplify the reading

```
#define N_BYTES 8

//interrupt controller
XScuGic INTCInst;
static XScuGic_Config *IntcConfig;
static u32 baseDDRmm2s= 0xa000000;
static u32 baseDDRs2mm= 0xa001000;

// Auxiliar functions
u32 up32(u64 val) { return (val & 0xffffffff00000000) >> 32;}
u32 down32(u64 val) { return val & 0xffffffff;}


//#########################################################################
//########################## Interrupt handlers ###########################
//#########################################################################

void InterruptHandlerS2MM ( void ){
   u32 tmpValue = Xil_In32 (DMA+S2MM_STATUS_REGISTER);;
   tmpValue = tmpValue | 0x1000; // Reset interrupt
   Xil_Out32(DMA+S2MM_STATUS_REGISTER, tmpValue);
}

void InterruptHandlerMM2S ( void ){
   u32 tmpValue = Xil_In32 (DMA+MM2S_STATUS_REGISTER);
   tmpValue = tmpValue | 0x1000; // Reset interrupt
   Xil_Out32(DMA+MM2S_STATUS_REGISTER, tmpValue2);

}

void InterruptHandlerRSA ( void ){
   XScuGic_Disable(&INTCInst, rsa_intr);
   Xil_Out32(RSA, 0x00000000); // Reset interrupt
    XScuGic_Enable(&INTCInst, rsa_intr);
}
```

```
//########################################################################
//############################## Application ##############################
//########################################################################

void decrypt(char exp[], char n[], u32 blocks[], int n_blocks, int size) {

   u32 mem_offset = 0;
   char tmp[8];
   Xil_Out32(RSA, n_blocks);

   //Exponent
   for(int i = size - 4; i >= 0; i -= 4) {
      memcpy(tmp,&exp[i*2],8);
      u32 ul = strtoul (tmp, NULL, 16);
      Xil_Out32(baseDDRmm2s + mem_offset, ul);
      mem_offset += 4;
   }

   //Modulus
   for(int i = size - 4; i >= 0; i -= 4) {
      memcpy(tmp,&n[i*2],8);
      u32 ul = strtoul (tmp, NULL, 16);
      Xil_Out32(baseDDRmm2s + mem_offset, ul);
      mem_offset += 4;
   }

   //Mesages
   for(int j = 0; j < n_blocks*size/4; j++) {
      Xil_Out32(baseDDRmm2s + mem_offset, blocks[j]);
      mem_offset += 4;
   }

   int read_size = size*(n_blocks + 2) + 1;
   Xil_Out32(DMA+MM2S_START_ADDRESS, baseDDRmm2s);
   Xil_Out32(DMA+MM2S_LENGTH, read_size);

   int write_size = size*n_blocks;
   Xil_Out32(DMA+S2MM_DESTINATION_ADDRESS, baseDDRs2mm);
   Xil_Out32(DMA+S2MM_LENGTH, write_size);

}

void encrypt(char exp[], char n[], char blocks[][N_BYTES], int n_blocks, int
    size) {

   u32 mem_offset = 0;
   char tmp[8];
   Xil_Out32(RSA, n_blocks);
```

```
   //Exponent
   for(int i = size - 4; i >= 0; i -= 4) {
      memcpy(tmp,&exp[i*2],8);
      u32 ul = strtoul (tmp, NULL, 16);
      Xil_Out32(baseDDRmm2s + mem_offset, ul);
      mem_offset += 4;
   }

   //Modulus
   for(int i = size - 4; i >= 0; i -= 4) {
      memcpy(tmp,&n[i*2],8);
      u32 ul = strtoul (tmp, NULL, 16);
      Xil_Out32(baseDDRmm2s + mem_offset, ul);
      mem_offset += 4;
   }

   //Mesages
   for(int j = 0; j < n_blocks; j++) {
      for(int i = size - 1; i >= 0; i-=4) {
         u32 ul = (u32)(blocks[j][i-3]); ul <<= 8;
         ul = ul | (u32)(blocks[j][i-2]); ul <<= 8;
         ul = ul | (u32)(blocks[j][i-1]); ul <<= 8;
         ul = ul | (u32)(blocks[j][i]);
         Xil_Out32(baseDDRmm2s + mem_offset, ul);
         mem_offset += 4;
      }
   }

   int read_size = size*(n_blocks + 2) + 1;
   Xil_Out32(DMA+MM2S_START_ADDRESS, baseDDRmm2s);
   Xil_Out32(DMA+MM2S_LENGTH, read_size);

   int write_size = size*n_blocks;
   Xil_Out32(DMA+S2MM_DESTINATION_ADDRESS, baseDDRs2mm);
   Xil_Out32(DMA+S2MM_LENGTH, write_size);

}

int main() {
    init_platform();
   xil_printf("\r\n--- Entering main() --- \r\n");
    xil_printf("Initializing system... \r\n ");

    int status;
    status = IntcInitFunction(XPAR_PS7_SCUGIC_0_DEVICE_ID);
      if (status != XST_SUCCESS)
         return XST_FAILURE;
```

```
    //Disable cache
   Xil_DCacheDisable();
    initialize_system();
   xil_printf("done! \n\r");
   //Keys
   char exp[16] = "0E52BEB9D61E0DE7";
   char d[16] = "00CFAB57EE0038D7";
   char n[16] = "1D1D96CC09FD4BEF";

   //Max 140 characters, 20, 40
   char mesg[140] = "Ejemplo";
   char blocks[20][N_BYTES];
   for(int i = 0; i < 20; i++)
      for(int j = 0; j < N_BYTES; j++)
         blocks[i][j]='\0';
   u32 crypted[40];
   u32 uncrypted[40];

   xil_printf("Message: %s \n\r",mesg);
   int n_blocks = message_to_blocks(mesg,blocks,strlen(mesg));
   trace_hex_mesg(blocks, n_blocks);

   int loop=1;
   while(loop) {
      int control=0;
      xil_printf("## want a new cycle? 0 no, 1 encrypt, 2 decrypt \n\r");
      scanf("%d", &control);

      if (control==0){
         loop=0;}
      else if (control==1){

         encrypt(exp,n,blocks,n_blocks,N_BYTES);
         trace_input(n_blocks);
         trace_output(n_blocks);
         save_output(crypted,n_blocks);

      } else {

         decrypt(d,n,crypted,n_blocks,N_BYTES);
         trace_input(n_blocks);
         trace_output(n_blocks);
         save_output(uncrypted,n_blocks);
         trace_mesg(uncrypted,n_blocks*2);
      }
   }
   xil_printf("-- Bye Bye -- \n\r");
   return 0;
}
```

B.2 Partial reconfiguration workflow

```
#First rsa
update_design -black_box -cell design_1_i/Sub/dummy_module_0

startgroup
create_pblock pblock_dummy_module_0
resize_pblock pblock_dummy_module_0 -add {SLICE_X50Y51:SLICE_X113Y148
    DSP48_X3Y22:DSP48_X4Y57 RAMB18_X3Y22:RAMB18_X5Y57
    RAMB36_X3Y11:RAMB36_X5Y28}
add_cells_to_pblock pblock_dummy_module_0 [get_cells [list
    design_1_i/Sub/dummy_module_0]] -clear_locs
endgroup
set_property RESET_AFTER_RECONFIG true [get_pblocks pblock_dummy_module_0]
set_property SNAPPING_MODE ON [get_pblocks pblock_dummy_module_0]

read_checkpoint -cell design_1_i/Sub/dummy_module_0 Partial/rsa.dcp

set_property HD.RECONFIGURABLE TRUE [get_cells design_1_i/Sub/dummy_module_0]

opt_design
place_design
route_design

write_bitstream -file Partial/bitstreams/rsa.bit
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -loadbit "up 0
    Partial/bitstreams/rsa_pblock_dummy_module_0_partial.bit"
    Partial/bitstreams/rsa.bin

#Second blowfish
update_design -black_box -cell design_1_i/Sub/dummy_module_0
lock_design -level routing

read_checkpoint -cell design_1_i/Sub/dummy_module_0 Partial/blowfish.dcp

opt_design
place_design
route_design

write_bitstream -file Partial/bitstreams/blowfish.bit
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -loadbit "up 0
    Partial/bitstreams/blowfish_pblock_dummy_module_0_partial.bit"
    Partial/bitstreams/blowfish.bin

#Finally blank block
update_design -black_box -cell design_1_i/Sub/dummy_module_0
update_design -buffer_ports -cell design_1_i/Sub/dummy_module_0
```

```
place_design
route_design

write_bitstream -file Partial/bitstreams/blank.bit
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -loadbit "up 0
    Partial/bitstreams/blank_pblock_dummy_module_0_partial.bit"
    Partial/bitstreams/blank.bin
```

B.3 Web page

*Index web page

```
<!DOCTYPE HTML>
<html>
   <head>
      <title>FPGA Accelerators</title>
      <meta charset="utf-8" />
      <meta name="viewport" content="width=device-width, initial-scale=1" />
      <link rel="stylesheet" href="assets/css/main.css" />
   </head>
   <body>

      <!-- Header -->
         <header id="header" class="alt">
         </header>

      <!-- Banner -->
         <section id="banner">
            <div class="inner">
               <header>
                  <h1>FPGA-based Accelerators for Cryptography</h1>
               </header>
               <form method="post" action="./cgi-bin/init.cgi">
                  <div class="12u$">
                     <ul class="actions">
                        <li><input type="submit" value="START DEMO" /></li>
                     </ul>
                  </div>
               </form>
            </div>
         </section>
    </body>
</html>
```

*Demo application web page

```
<!DOCTYPE HTML>
<html>
   <head>
      <title>FPGA Accelerators</title>
      <meta charset="utf-8" />
      <meta name="viewport" content="width=device-width, initial-scale=1" />
      <link rel="stylesheet" href="assets/css/main.css" />
   </head>
   <body>
      <!-- Header -->
      <header id="header" class="alt">
      </header>
      <!-- Main -->
      <div id="main">
      <!-- Section -->
      <section class="wrapper style1">
         <div class="inner">
            <!-- 2 Columns -->
            <div class="flex flex-2">
            <div class="col col1">
               <div class="image round fit">
                  <img src="images/Init_pict.png" alt="" />
               </div>
            </div>
            <div class="col col2">
            <h3>Demo Application</h3>
            <p>In this demo application...</p>
            <form method="post" action="./set_algorithm.cgi">
            <div class="row uniform">
               <div class="4u 12u$(small)">
                  <input type="radio" id="rsa" name="algorithm" value="1">
                   <label for="rsa">RSA</label>
               </div>
               <div class="4u 12u$(small)">
                  <input type="radio" id="blowfish" name="algorithm" value="2">
                  <label for="blowfish">Blowfish</label>
               </div>
               <div class="4u 12u$(small)">
                  <input type="radio" id="blank" name="algorithm" value="3">
                  <label for="blank">Blank</label>
               </div>
               <div class="12u$">
                  <textarea name="message" id="message" placeholder="Key in
                      hexadecimal" rows="1"></textarea>
               </div>
```

```
            <div class="12u$">
               <ul class="actions">
               <li><input type="submit" value="Select" /></li>
               </ul>
            </div>
         </div>
         </form>
         </div>
         </div>

         <div class="flex flex-3">
         <div class="col col1">
         <form method="post" action="#">
         <div class="row uniform">
            <div class="12u$">
               <textarea name="message" id="message" placeholder="Enter your
                   message" rows="6"></textarea>
            </div>
            <div class="12u$">
               <ul class="actions">
               <li><input type="submit" value="Submit" /></li>
               </ul>
            </div>
         </div>
         </form>
         </div>
         <div class="col col2">
            <div class="box"></div>
         </div>
         <div class="col col3">
            <div class="box"></div>
         </div>
         </div>
      </div>
   </section>
   </div>
</body>
</html>
```

B.4 User application

*Initialisation script

```
#!/bin/bash

#Init system
echo 0 > /sys/devices/soc0/amba/f8007000.devcfg/is_partial_bitstream
cat /run/media/mmcblk0p2/blank.bit > /dev/xdevcfg
echo 1 > /sys/devices/soc0/amba/f8007000.devcfg/is_partial_bitstream
cat /run/media/mmcblk0p2/blank.bin > /dev/xdevcfg

echo "BLANK" > .config

#Generate web page
...
```

*Select algorithm script

```
#!/bin/bash

#Read QUERY_STRING
saveIFS=$IFS
IFS='=&'
parm=($QUERY_STRING)
IFS=$saveIFS

for ((i=0; i<${#parm[@]}; i+=2))
do
    declare var_${parm[i]}=${parm[i+1]}
done

#Check errors
error=""
msg=""
size=${#var_message}

if [ "$var_algorithm" -eq "1" ] && [ "$size" -ne "0" ]; then
   if [ "$size" -ne "50" ]; then
      error="RSA key must be three 16 digits dash separated"
   else
      if ! [[ "$var_message" =~
          ^[0-9A-Fa-f]{16}\-[0-9A-Fa-f]{16}\-[0-9A-Fa-f]{16} ]]; then
         error="RSA key must be hexadecimal digits"
      fi
   fi
fi
```

```
if [ "$var_algorithm" -eq "2" ] && [ "$size" -ne "0" ]; then
   if [ "$size" -ne "16" ]; then
      error="Blowfish key must be 16 digits"
   else
      if ! [[ "$var_message" =~ ^[0-9A-Fa-f]{16} ]]; then
         error="Blowfish key must be hexadecimal digits"
      fi
   fi
fi

#Check algorithm
if [ "$error" = "" ]; then
   if [ "$var_algorithm" -eq "1" ]; then
      echo "RSA" > .config
      cat /run/media/mmcblk0p2/rsa.bin > /dev/xdevcfg
      msg="RSA algorithm selected"
      if [ "$size" -ne "0" ]; then
         echo "$var_message" >> .config
      fi
   elif [ "$var_algorithm" -eq "2" ]; then
      echo "BLOWFISH" > .config
      cat /run/media/mmcblk0p2/blowfish.bin > /dev/xdevcfg
      msg="Blowfish algorithm selected"
      if [ "$size" -ne "0" ]; then
         echo "$var_message" >> .config
      fi
   elif [ "$var_algorithm" -eq "3" ]; then
      echo "BLANK" > .config
      cat /run/media/mmcblk0p2/blank.bin > /dev/xdevcfg
      msg="No algorithm selected"
   fi
fi


#Reset DMA
poke 0x40400000 0x4
poke 0x40400030 0x4

poke 0x40400000 0x10003
poke 0x40400030 0x10003

#Generate web page
..
```

*Encryption script

```
#!/bin/bash

#Read QUERY_STRING
saveIFS=$IFS
IFS='=&'
parm=($QUERY_STRING)
IFS=$saveIFS

for ((i=0; i<${#parm[@]}; i+=2))
do
    declare var_${parm[i]}=${parm[i+1]}
done

#Check configuration
var_algorithm=$(cat .config | head -n 1)
n_lines=$(cat .config | wc -l)
if [ $n_lines -eq "2" ]; then
   key=$(cat .config | tail -n 1)
else
   key=$(echo "NULL")
fi

#Encrypt
encrypted=""
decrypted=""
MM2S_addr="A000000"
S2MM_addr="A001000"
MM2S_base_addr="A000000"
S2MM_base_addr="A001000"
n_blocks=0
var_message=$(echo "$var_message" | tr '+' ' ')

if [ "$var_algorithm" = "RSA" ]; then

   N=6
   N2=14
   Overhead=11 #Exponent + Modulus + trash byte
   if [ "$key" = "NULL" ]; then
      poke 0x$MM2S_addr 0xD61E0DE7
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
      poke 0x$MM2S_addr 0xE52BEB9
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
      poke 0x$MM2S_addr 0x9FD4BEF
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
      poke 0x$MM2S_addr 0x1D1D96CC
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
```

```
   else
      poke 0x$MM2S_addr 0x${key:8:8}
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
      poke 0x$MM2S_addr 0x${key:0:8}
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
      poke 0x$MM2S_addr 0x${key:42:8}
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
      poke 0x$MM2S_addr 0x${key:34:8}
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
   fi

elif [ "$var_algorithm" = "BLOWFISH" ]; then
   N=8
   N2=16
   if [ "$key" = "NULL" ]; then
      poke 0x$MM2S_addr 0xD61E0DE7
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
      poke 0x$MM2S_addr 0xE52BEB9
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
   else
      poke 0x$MM2S_addr 0x${key:8:8}
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
      poke 0x$MM2S_addr 0x${key:0:8}
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
   fi
elif [ "$var_algorithm" = "BLANK" ]; then
   encrypted="No algorithm selected"
   decrypted="No algorithm selected"
fi

if [ "$var_message" = "" ]; then
   encrypted="Empty text"
   decrypted="Empty text"

elif [ "$var_algorithm" = "RSA" ] || [ "$var_algorithm" = "BLOWFISH" ]; then
   #Encryption
   hex_msg=$(echo -n "$var_message" | hexdump -v -e '/1 "%02X "' | tr -d ' ')
   size=${#hex_msg}
   it="$size"

   while [ "$it" -gt "0" ]; do
      if [ "$it" -lt "8" ]; then
         value=$(echo ${hex_msg:0:$it})
         poke 0x$MM2S_addr 0x$value
         MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
         poke 0x$MM2S_addr 0x0
         MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
         it=-1
```

```
    elif [ "$it" -lt "$N2" ]; then
        tmp=$(($tmp - $N))
        value=$(echo ${hex_msg:$tmp:$N})
        poke 0x$MM2S_addr 0x$value
        MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
        value=$(echo ${hex_msg:0:$tmp})
        poke 0x$MM2S_addr 0x$value
        MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
        it=-1
    else
        tmp=$(($it - 8))
        value=$(echo ${hex_msg:$tmp:8})
        poke 0x$MM2S_addr 0x$value
        MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
        tmp=$(($tmp - $N))
        value=$(echo ${hex_msg:$tmp:$N})
        poke 0x$MM2S_addr 0x$value
        MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
        it=$tmp
    fi

    n_blocks=$(echo "obase=ibase=16;$n_blocks+1" | bc)
done

poke 0x43c00000 0x$n_blocks
otpt_counter=$(echo "obase=ibase=16;$n_blocks*8" | bc)
if [ "$var_algorithm" = "RSA" ]; then
    inpt_counter=$(echo "obase=ibase=16;$otpt_counter+11" | bc)
else
    inpt_counter=$(echo "obase=ibase=16;$otpt_counter+9" | bc)
fi

poke 0x40400018 0x$MM2S_base_addr
poke 0x40400028 0x$inpt_counter
poke 0x40400048 0x$S2MM_base_addr
poke 0x40400058 0x$otpt_counter

it=0
end=$(echo "ibase=16;$n_blocks*2" | bc)
if [ "$var_algorithm" = "RSA" ]; then

    gpio_decrypt=$n_blocks
    MM2S_addr=$MM2S_base_addr
    if [ "$key" = "NULL" ]; then
        poke 0x$MM2S_addr 0xEE0038D7
        MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
        poke 0x$MM2S_addr 0xCFAB57
        MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
```

```
      else
         poke 0x$MM2S_addr 0x${key:25:8}
         MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
         poke 0x$MM2S_addr 0x${key:17:8}
         MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
      fi
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+8" | bc)
   else
      gpio_decrypt=$((0xC0000000 | 0x$n_blocks))
      gpio_decrypt=$(echo "obase=16; $gpio_decrypt" | bc)
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_base_addr+8" | bc)
   fi

   while [ "$it" -lt "$end" ]; do
      output=$(peek 0x$S2MM_addr)
      poke 0x$MM2S_addr $output
      hex="${output:2:32}"
      hex=$(echo "0x$hex" | xxd -r)
      encrypted="$encrypted$hex"
      S2MM_addr=$(echo "obase=ibase=16;$S2MM_addr+4" | bc)
      MM2S_addr=$(echo "obase=ibase=16;$MM2S_addr+4" | bc)
      it=$(($it + 1))
   done

   #Decrypt
   poke 0x43c00000 0x$gpio_decrypt
   poke 0x40400018 0x$MM2S_base_addr
   poke 0x40400028 0x$inpt_counter
   poke 0x40400048 0x$S2MM_base_addr
   poke 0x40400058 0x$otpt_counter

   it=$end
   tmp=$((($end - 1) * 4))
   tmp=$(echo "obase=16; $tmp" | bc)
   S2MM_addr=$(echo "obase=ibase=16;$S2MM_base_addr+$tmp" | bc)
   while [ "$it" -gt "0" ]; do
      output=$(peek 0x$S2MM_addr)
      hex="${output:2:32}"
      hex=$(echo "0x$hex" | xxd -r)
      decrypted="$decrypted$hex"
      S2MM_addr=$(echo "obase=ibase=16;$S2MM_addr-4" | bc)
      it=$(($it - 1))
   done

fi

#Generate web page
...
```

References

[1] Field-programmable gate array Wikipedia https://es.wikipedia.org/wiki/Field-programmable_gate_array

[2] Application-specific integrated circuit Wikipedia https://en.wikipedia.org/wiki/Application-specific_integrated_circuit

[3] Zedboard Xilinx Products https://www.xilinx.com/products/boards-and-kits/1-elhabt.html

[4] ZedBoard Getting Started Guide. Version 7.0 2017 AVNET

[5] Zynq-7000 All Programmable SoC Data Sheet: Overview DS190, June, 2017

[6] 7 Series DSP48E1 Slice User Guide UG479 v1.10, March, 2018

[7] AXI DMA v7.1 logiCORE IP Product Guide PG021 2016

[8] AXI4-Stream Infrastructure IP Suite v2.2 LogicCORE IP Product Guide PG085

[9] Vivado Design Suite User Guide: Design Flows Overview UG892 v2017.4

[10] Vivado Design Suite User Guide: High-Level Synthesis UG902 v2017.1

[11] Vivado Design Suite User Guide: Embedded Processor Hardware Design UG898 v2018.1

[12] Petalinux Tools Documentation: Reference guide UG1144 v2017.1

[13] RSA Article, Wikipedia https://en.wikipedia.org/wiki/RSA_(cryptosystem).

[14] D. E. Knuth. The Art of Computer Programming: Seminumerical Algorithms, volume 2. Reading, MA: Addison-Wesley, Second edition, 1981.

[15] RSA Hardware implementation, RSA security //ftp://ftp.rsasecurity.com/pub/pdfs/tr201.pdf.

[16] P. L. Montgomery. Modular multiplication without trial division. Mathematics of Computation, 44(170):519-521, April 1985.

[17] Blowfish Article, Wikipedia https://en.wikipedia.org/wiki/Blowfish_(cipher)

[18] Bruce Schneier security blog https://www.schneier.com/academic/blowfish/

[19] Blowfish specification cryptowiki http://cryptowiki.net/index.php?title=Cryptographic_algorithm_Blowfish

[20] Hexadecimal values of pi https://www.schneier.com/code/constants.txt

[21] Processing System 7 v5.5 LogiCORE IP Product Guide PG082 2017

[22] Petalinux Tools Documentation: Workflow Tutorial UG1156 v2017.1

[23] Vivado Design Suite User Guide: Partial Reconfiguration UG909 v2018

[24] AXI Timer 2.0 Logic CORE IP Product guide PG079 2016

[25] 0xStubs Measuring time Xilinx https://0xstubs.org/measuring-time-in-a-bare-metal-zynq-application/

[26] Keysight Technologies 34461A Digital Multimeter https://www.keysight.com/en/pdx-2891615-pn-34461A/digital-multimeter-6-digit-truevolt-dmm

[27] Advanced Encryption Standard Wikipedia https://es.wikipedia.org/wiki/Advanced_Encryption_Standard