

POLITECNICO DI TORINO

Master of Science in Electronic Engineering

Master Degree Thesis

Implementation of a post-quantum cryptography algorithm on an FPGA board



Supervisor:
prof. Guido Masera

Candidate:
Giuseppe PULETTO

Co-Supervisor:
prof. Maurizio Martina

ACADEMIC YEAR 2019-2020

Abstract

Cryptography (from Ancient Greek words *kryptós* and *graphein*, which together mean "secret writing") is the set of theory and techniques used to guarantee a secure communication between sender and receiver. A communication is secure if there is no risk for the message to reach an unwanted receiver.

Cryptography originates in the military, from the need to secretly communicate with allies. Coding the message, in such a way that it is no longer understandable for who does not have the key to decrypt it, solves the problem. In cryptography literature, the encrypted message is called ciphertext. Coding and decoding are the operations to obtain the cyphertext from the plaintext and vice versa.

The encryption algorithm has to be as robust as possible. The more robust it is, the more difficult it is to understand the key to decrypt the cyphertext.

Today, cryptography finds application everywhere. Electronic commerce, chip-based payment cards, digital currencies, computer passwords, implanted medical devices make use of it. This means that even devices with limited storage and poor computational capabilities must be able to code and decode information in a robust and fast way.

Modern cryptosystems make use of public key algorithms, which use a public key and a private key. As the name suggests, the public key is known to all and is used to encode the message. Instead, the private key is secret. Only the recipient, which needs to decrypt the message, owns it. Although these two keys are linked one each other, it is impossible to recover the private key knowing only the public one.

RSA is an example of public key cryptography. The security of this cryptosystem is based on the so-called "factoring problem". This operation is computationally not sustainable by a classical computer.

In 1994, the mathematician Peter Shor invented an algorithm, which is able to break this kind of cryptosystems if executed by an ideal quantum computer.

Studying and developing cryptosystems able to resist to quantum computer attacks is urgent and inevitable.

A candidate for "post-quantum cryptography" is *McEliece* cryptosystem. It is based on the practical difficulties of decoding a generic linear code. The original version of this algorithm uses binary Goppa codes

and requires to store in memory large matrices. This makes problematic encrypting and decrypting in embedded systems with limited hardware resources.

Variants that use LDPC or QCLDPC codes solve this problem. They require less memory and computational effort. One of them is the *LEDAPkc* algorithm.

The decoding technique of LEDAPkc is called *Q-Decoder*. The aim of this thesis is the creation of a prototype on fpga which implements a Q-Decoder together with a client-server model communication.

An encrypted message is sent via the Internet. It is coded according LEDAPkc encryption algorithm. Once received, it is decrypted employing the Q-decoder decoding method. Therefore, both the sender and the recipient are hosts, that is, computers connected to a network.

The employed Q-decoder is similar to the one developed by Kristjane Koleci in the master thesis *VLSI QC-LDPC Decoder for Post-Quantum Cryptography*.

According to specifications, the prototype building requires an FPGA which has at least an Ethernet port. A built-in memory large enough to store information like message, private key, syndrome, correlation, etc. could maximize the decoder flexibility and minimize hardware resources usage and time consuming. The De1-Soc Board is a good compromise and the programmable logic is enough to implement the Q-Decoder.

The whole prototype is built using VHDL as a hardware description language and the standard edition of *Intel® Quartus® Prime Design Software 16.0*.

It is made up of two subsystems. The first is the HPS subsystem and includes the following IP components:

- an Hard Processor System;
- two FIFO memories to interface correctly the hard processor system with the decoder and vice versa;
- a phase locked loop for the clock inputs of the hard processor system and fifos input or output ports.

The second is in the FPGA block of the DE1-SoC and contains:

- the Q-decoder;
- the 64 MB SDRAM available in the fpga block;
- a sdram controller for the 64 MB SDRAM;
- a phase locked loop for the clock inputs of the sdram controller and fifos input or output ports;
- a parallel Input/Output component used by the HPS to start the decoding of the cyphertext.

The PLLs of the two subsystems share the reference clock of 50.0 MHz, which is the output of a clock source block.

First, the encrypted message is sent from client to server via the Ethernet. The communication between these two nodes of the network is established using the so called socket programming. This means that the two hosts run two different parts of a distributed application, that is written in C language. The server host is the hard processor system.

Once the encrypted message is received, the hps writes it, together with the private key and the transposed private key, on a fifo. Once written, they are read and stored in the SDRAM by one of the units which compose the Q-decoder.

Finally, the decoding process starts. The syndrome, that is the product of the message and the transposed private key, is computed, adjusted and written in memory. Also the correlation, which is obtained multiplying the syndrome by the private key, is stored in SDRAM. From the correlation and syndrome weight, the bits to flip in the message are found. Toggling them, the message is updated. The decoding process is an iterative algorithm. It ends when the syndrome weight is equal to zero or a maximum number of iterations is reached.

In terms of occupied resources, the Q-decoder uses the 7 % of ALMs, the 43% of pins, the 10 % of block memory bits and two out of six PLLs.

In terms of performances, the bottleneck is represented by the 64 MB SDRAM, which allows a clock signal of frequency equal or less than 143.0 MHz.

Desidero ringraziare innanzitutto il mio relatore, il Professore Guido Masera, per avermi dato la possibilità di avvicinarmi e appassionarmi a questo argomento.

Ringrazio i miei genitori, senza i quali nulla di tutto questo sarebbe mai stato possibile.

Ringrazio mia sorella, che da sempre per me è stata esempio di tenacia, forza e vita.

Ringrazio Carmelo, Davide, Giuseppe, Luca e Tommaso, che ogni giorno mi hanno regalato un sorriso diverso.

Infine ringrazio tutti quelli che in un modo o nell'altro hanno reso più leggeri questi anni di università, amici e colleghi, siete tanti e non riesco a nominarvi tutti, ma vi assicuro che vi porterò per sempre con me.

A mio padre e mia madre.

Contents

Abstract	i
1 Introduction	1
1.1 McEliece Cryptosystem	1
1.2 LEDApkc Cryptosystem	2
1.3 Prototype specifications	3
1.4 The choice of the FPGA	4
2 The prototype system IP blocks	6
2.1 A general overview of the whole system	6
2.2 About Qsys system integration tool	7
2.3 The HPS subsystem	8
2.3.1 Advanced Extensible Interface bridges	8
2.3.2 Avalon FIFO memories	9
2.3.3 SDRAM controller	11
2.3.4 Parallel Input\Ouput component	13
2.3.5 Clock Sources and phase locked loops	13
2.3.6 Reset signals	14
2.4 The Qsys design	15
3 Ethernet client-server model	16
3.1 Socket programming	16
3.1.1 Server steps	16
3.1.2 Client steps	17
3.2 The client-server computer program	18
3.2.1 server_fpga.c	18
3.2.2 client_server.c	19
3.2.3 common_functions.c	21
3.2.4 common_function.h	22
3.3 Makefile	22
4 Q-decoder architecture	24
4.1 QsysSubsystem	25
4.2 WritingSDRAM	26
4.2.1 WritingSDRAM_CU	26
4.2.2 WritingSDRAM_DP	28
4.3 SyndromeCMP	29

4.3.1	CU_VectorByCirculant	30
4.3.2	DP_VectorByCirculant	34
4.4	SyndromeAdjustandWeight	34
4.4.1	CU_SyndromeAdjust_and_Weight	34
4.4.2	DP_SyndromeAdjust_and_Weight	36
4.5	CorrelationCMP	36
4.5.1	CU_VectorByCirculantInteger	36
4.5.2	DP_VectorByCirculantInteger	41
4.6	ErrorPos	41
4.6.1	CU_ThresholdPos	41
4.6.2	DP_ThresholdPos	42
4.7	ThresholdEvaluation	42
4.8	MessageUpdate	42
4.8.1	CU_VectorPosFlip	42
4.8.2	DP_VectorPosFlip	44
4.9	WritingDecodedMsg	44
4.9.1	WritingDecodedMsg_CU	44
4.9.2	WritingDecodedMsg_DP	46
4.10	Reset signals management	47
4.11	SDRAM memory mapping	47
4.12	Synthesis and performances results	48
5	Conclusions	50

List of Figures

1.1	DE1-Soc block diagram	5
2.1	Prototype system	7
2.2	AXI interfaces	9
2.3	Avalon-MM write slave to Avalon-MM read slave	10
2.4	SDRAM controller block diagram	11
2.5	Avalon-MM read and write requests with <i>waitrequest</i> signal	13
2.6	Qsys design - FPGA IP components	15
2.7	Qsys design - HPS IP components	15
4.1	Q-decoder control unit	25
4.2	WritingSDRAM control unit flowchart	27
4.3	WritingSDRAM datapath schematic	29
4.4	VectorByCirculant control unit extra states	31
4.5	VectorByCirculant control unit	33
4.6	WritingDecodedMsg control unit flowchart	35
4.7	VectorByCirculant control unit extra states	38
4.8	VectorByCirculantInteger control unit	40
4.9	ThresholdPos control unit flowchart	42
4.10	VectorByPos control unit flowchart	43
4.11	WritingDecodedMsg control unit flowchart	45
4.12	WritingDecodedMsg datapath schematic	47
4.13	SDRAM memory mapping	48

Introduction

The LEDApkc Public Key Cryptosystem is the variant of McEliece Cryptosystem for QC-LDPC codes.

1.1 | McEliece Cryptosystem

The McEliece Cryptosystem takes its name from Robert J. McEliece, its inventor. The original version uses as secret codes irreducible Goppa codes, but the algorithm behind it can be generalized to other types of codes. It is an asymmetric encryption algorithm and this means that the message is coded employing a Public Key and decrypted through a Secret Key.

Let us suppose that the sender wish to transmit a message m through an insecure channel to the intended recipient. In cryptography literature, the sender is usually called *Alice*, the intended *Bob* and the adversary *Eve*. Alice and Bob derive their names from the first two letters of the alphabet, A and B, that are commonly used to mark the starting and ending point of a communication channel, a journey etc. Eve comes from eavesdropper.

Key generation and encryption. Let us imagine m as a vector $1 \times k$. First of all, in McEliece cryptosystem, Alice searches and finds Bob's Public Key G' , a matrix $k \times n$. G' is a public key, so it is accessible for everyone, including Eve. Then she encrypts m , multiplying it by the matrix G and adding to the result e , a random binary vector $1 \times n$ with weight t arbitrarily generated by Alice itself:

$$x = m \times G' + e$$

x is the ciphertext, the encrypted version of the message m . Once obtained x from m , Alice can send x to Bob through the channel.

Alice and Eve have the public key G' , but not the secret key. Only Bob has access to it. The private key is more than a simple matrix. The public key G' is linked to the private key, but it is impossible from G' going back to the private key. Bob generates the two keys, choosing randomly a secret binary block code $C(n,k)$, with n as codeword length, k

as information word length and a $k \times n$ Generator matrix G . G is able to correct a number of errors less or equal than t . The relationship between G and G' is the following:

$$G' = S \times G \times P$$

S is a dense $k \times k$ non singular binary scrambling matrix, P a $n \times n$ permutation matrix. G , S and P forms together the secret key.

Decryption. Once received x , Bob can recover first x' multiplying x by P and then m exploiting the relationship between x' and it m :

$$x' = m \times S \times G + e \times P^{-1}$$

1.2 | LEDApkc Cryptosystem

The LEDApkc algorithm differs from the McEliece one because uses as secret codes non-algebraic QC-LDPC codes, the public code and the private one are neither coincident or equivalent and employs non-bounded-distance decoding algorithms.

Key generation. Two matrices, H and Q , form the secret key. H is a parity check-matrix made of $r_0 \times n_0$, $p \times p$ circulant blocks. Q is a transformation matrix made of $n_0 \times n_0$, $p \times p$ circulant blocks. Starting from H and Q , the matrices L and M are computed:

$$L = H \times Q = [L_0|L_1|\dots|L_{n_0-1}]$$

$$M = L_{n_0-1}^{-1} \times L = [M_0|M_1|\dots|M_{n_0-2}|I_p] = [M_l|I_p]$$

where I_p is the $p \times p$ identity matrix, while M_l the public key.

Encryption function. Starting from the public key M_l , a matrix G' is computed as:

$$G' = [I_k|M_l^T]$$

where I_k is the $k \times k$ identity matrix. Also a binary error vector $1 \times pn_0$ with a Hamming weight t is generated. The message m is a binary vector $1 \times p(n_0 - 1)$. Once found G' and e , Alice can encrypt m as follows:

$$x = m \times G' + e$$

where x is the ciphertext.

Decryption function. Knowing H and Q , Bob computes the syndrome of the message, a binary $p \times 1$ vector from x :

$$s^T = (H \times Q) \times x^T$$

From s , it is possible to recover e remembering that:

$$(H \times Q) \times G'^T \times m^T = 0_{p \times p(n_0-1)} \times u^T = 0_{p \times 1}$$

As a consequence:

$$s^T = (H \times Q) \times x^T = (H \times Q) \times (m \times G' + e)^T = (H \times Q) \times e^T = H \times (e \times Q^T)^T$$

Found e , Bob recovers m from:

$$x + e = u \times [I_k | M_l^T]$$

Q decoder. There are several ways to recover the message m starting from the syndrome s .

The most efficient method is the one called *Q-decoder*. It is an iterative bit flipping algorithm with a lookup-table to reduce the number of iterations. Since it uses the transpose of matrix Q to estimate the error vector e , it is able to exploit the utmost correction power of QC-LDPC codes. This explains its remarkable efficiency and its name.

From the message m and private key \mathbb{L} , that contains n_0 circulant blocks, Bob finds the syndrome s . In the Kristjane Koleci's LEDA architecture, n_0 is equal to 2 and each circulant block is a $p \times p$ matrix, where p is 15013. Therefore, m is a $1 \times n_0 * p$, while L is a $p \times n_0 * p$. To compute the syndrome:

$$m \times L^T = [m_0 \quad m_1] \begin{bmatrix} L_0 \\ L_1 \end{bmatrix} = \begin{bmatrix} m_0 * L_0 \\ m_1 * L_1 \end{bmatrix} = \begin{bmatrix} s_0 \\ s_1 \end{bmatrix} = s^T$$

From s and L , the correlation *UPC* is evaluated:

$$s \times L = s \begin{bmatrix} L_0 & L_1 \end{bmatrix} = [s * L_0 \quad s * L_1] = UPC$$

From the syndrome weight and a proper lookup table, that derives from the chosen LEDA architecture parameters, the threshold is computed. Then, the errors positions are found subtracting each byte to the threshold. Indeed, if the correlation byte, which is related to a specific bit in the message, is higher than the threshold, the position of the bit is saved. The bit is marked as an error.

Finally, the message is updated flipping all its wrong bits.

The whole process is repeated until there are no more errors or the maximum number of iterations is reached. If no more errors remain, the decoding process ended successfully, otherwise, it failed.

1.3 | Prototype specifications

The prototype is built following the guidelines described below.

- The idea is to set up a client-server model communication. An encrypted message has to be sent via the Internet. It is coded according LEDApkc encryption algorithm. Once received, it has to be decrypted employing the Q-decoder decoding method. Therefore, both the sender and the recipient have to be hosts, that is, computers connected to a network.
- The main design efforts should be spent on the Q-Decoder implementation rather than the client-server Ethernet communication.
- The decoder has to be implemented on an fpga and be as similar as possible to the one developed by Kristjane Koleci in the master thesis *VLSI QC-LDPC Decoder for Post-Quantum Cryptography*.
- The prototype must be as small and fast as possible in terms of occupied hardware resources and time consuming.
- The system must be flexible to a further implementation based on the red/black concept. In cryptosystems, the red/black architecture segregates carefully the black signals from the red ones. The first are the ones who carry the cyphertext, the others who carry the plaintext.

1.4 | The choice of the FPGA

According to specifications, the prototype building requires an FPGA which has at least an Ethernet port. A built-in memory large enough to store information like message, private key, syndrome, correlation, etc. could maximize the decoder flexibility and minimize hardware resources usage and time consuming.

Keeping in mind the Q-decoder needs, the De1-Soc Board seems to be a good compromise. The programmable logic is enough to implement the Q-Decoder. This could be easily verified using tools like Intel Quartus Prime Design Software. These tools give the possibility to compile a VHDL project for a specific hardware programmable board. The VHDL project used is the one built by Kristjane Koleci and described in her master thesis. Furthermore The fpga integrates an ARM-based hard processor system consisting of processor, peripherals and memory interfaces. Its equipment includes an Ethernet networking and a 64MB SDRAM. 64 MB are more than enough to store the Q-decoder information. Having such a big memory also means space for further decoder improvements that could require more information to store.

The block diagram of the board is shown in figure 1.1. The picture is taken from the tutorial material offered by Altera.

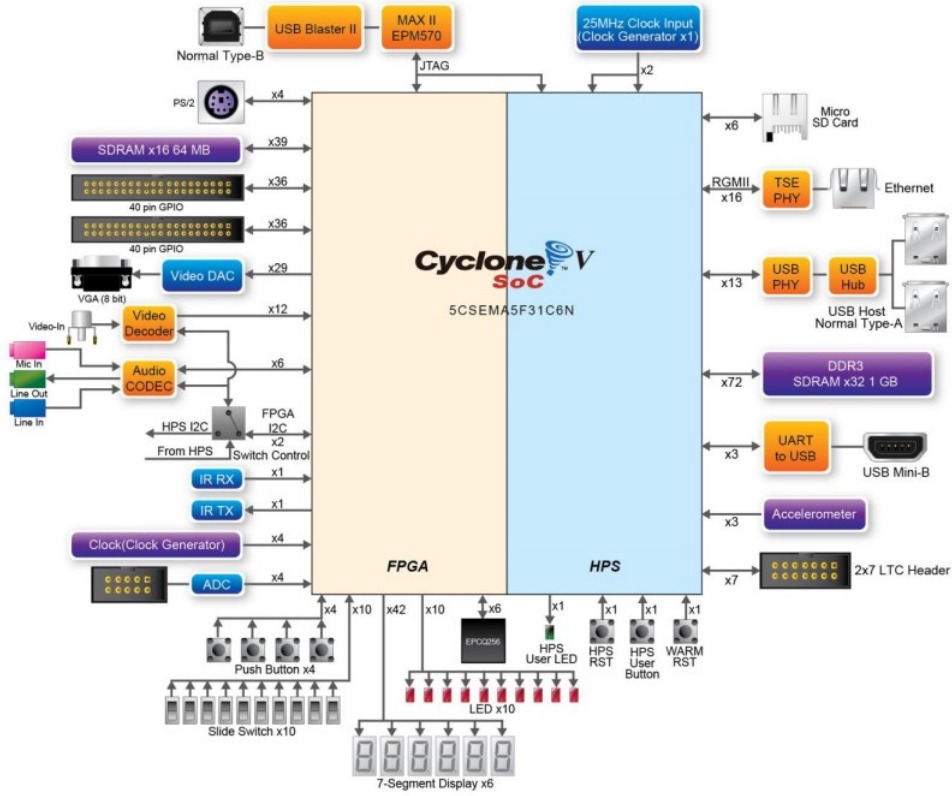


Figure 1.1: DE1-Soc block diagram

The Ethernet physical layer with RJ45 connector can transmit Ethernet frames at a rate of a gigabit per second. It is located in the Hard Processor System block of the DE1-Soc board. Unfortunately, there is no direct connection between the Ethernet connector and the FPGA block. This forces to instantiate in the prototype a minimal HPS which allows the programmable logic to receive and send information via Ethernet. However, not all evils come to harm. A hard processor system could significantly reduce the design efforts on implementing a well working and robust Ethernet communication. The idea is to exploit the power of the 800 MHz Dual-core ARM Cortex-A9 MPCore processor to run a simple C program which receives and send information via Ethernet. Thanks to HPS interfaces towards the FPGA region, the information can be transferred in the 64 MB SDRAM. Once stored in the SDRAM, they are accessible at occurrence by the Q-decoder. The hard processor system also guarantees flexibility and leaves space for further and powerful decoder improvements.

The prototype system IP blocks 2

2.1 | A general overview of the whole system

The prototype system is made up of two subsystems. The first is the HPS subsystem and includes the following IP components:

- the Arria V/Cyclone Hard Processor System, in the Qsys design called *ARM_A9_HPS*;
- two Avalon FIFO memories to interface correctly the hard processor system with the decoder and vice versa, in the Qsys design named as *FPGA_to_HPS_fifo* and *HPS_to_FPGA_fifo*;
- a phase locked loop for the clock inputs of the hard processor system, the HPS_TO_FPGA fifo input port and FPGA_TO_HPS fifo output port, in the Qsys design known as *HPS_pll*;

The second is in the FPGA block of the DE1-SoC and contains:

- the Q-decoder;
- the 64 MB SDRAM available in the fpga block;
- a sdram controller for the 64 MB SDRAM, in the Qsys design renamed as *FPGA_sdram_controller*;
- a phase locked loop for the clock inputs of the sdram controller, in the Qsys system called *FPGA_pll*, the HPS_TO_FPGA fifo output port and FPGA_TO_HPS fifo input port;
- a PIO, a parallel Input/Output component, in the Qsys design known as *LEDR_pio*, used for debug purposes and by the HPS to start the decoding of the cyphertext;

The PLLs of the two subsystems share the reference clock of 50.0 MHz, which is the output of

- the clock source block, in the Qsys design renamed as *ref_clock_block*.

Figure 2.1 shows a block diagram of the whole prototype system.

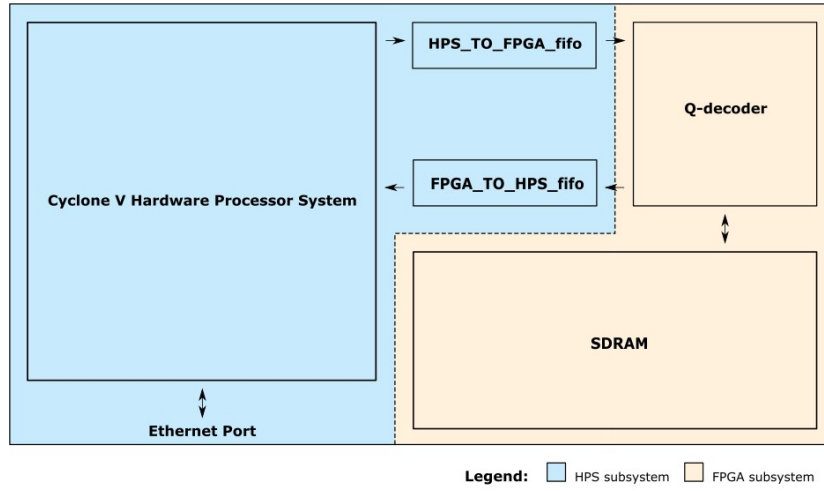


Figure 2.1: Prototype system

The whole prototype is built using VHDL as a hardware description language and the standard edition of *Intel® Quartus® Prime Design Software 16.0*.

2.2 | About Qsys system integration tool

Intel® Quartus® Prime Design Software Suite includes a system integration tool called *Qsys* or *Platform Designer*. *Qsys* is the best and fastest way to use intellectual (IP) functions and subsystems. Indeed, using this software component, the user can instantiate all the IP blocks available in the chosen FPGA board. The tool also generates automatically the interconnect logic between the used components, saving design time and improving productivity.

To use *Qsys*, it is enough following the *Qsys System Design Tutorial* offered by Intel. To summarize:

1. in the Quartus Prime Design Software Suite, click *Tools > Qsys* to create a new *Qsys* design; the *Qsys* window appears and in the tab called *System Contents* a first component is already instantiated; it is the clock source *clk_0*;
2. to add a new component, use the *IP catalog* tab to find a suitable IP function or subsystem and click *Add*;
3. to edit the properties of an IP block, double click on it to open the *Parameters* tab;

4. to rename or connect the IP blocks use the *System Contents* tab; this tab also shows the components HPS base and end addresses;
5. click *Generate HDL* to generate the Qsys system; it is possible choosing the HDL in which the system will be generated; the command *Generate > Show Instantiation Template* shows how to integrate the Qsys system with the rest of the HDL code;
6. click *Finish* to close the Qsys window.

2.3 | The HPS subsystem

The hard processor system available in the DE1-SoC board is the Cortex-A9 microprocessor unit (MPU) subsystem. It includes a 32-bit dual-core processor, a level 2 (L2) cache, an Accelerator Coherency Port ID mapper and debugging modules. The 32-bit dual-core processor is the Cortex-A9 MPCore. It is composed of two instantiation of the ARM Cortex-A9 processor. It also includes other modules: watchdog and private internal timers for each core, a global timer, a generic interrupt controller, a Snoop Control Unit and an Accelerator Coherency Port.

2.3.1 | Advanced Extensible Interface bridges

The HPS is connected to the FPGA fabric through a dedicated interface. It is the Advanced Extensible Interface which is also known as AXI bridge. There is a bridge to go from the HPS to the FPGA and another one for the vice versa. The first is the FPGA-TO-HPS bridge, the second the HPS-TO-FPGA bridge. These two interfaces are data width configurable: 32, 64 or 128 bits. They also manage data width conversion, clock crossing, buffering. In other words, they give all the tools to have access to HPS slaves from FPGA fabric and vice versa. There are also available lower performance AXI interfaces. They are called Lightweight FPGA-TO-HPS and HPS-TO-FPGA bridges. Their data width is fixed and equal to 32-bit and are useful for low-bandwidth traffic.

Picture [2.2](#) is taken from the embedded IP user guide offered by Intel and shows how AXI bridges connect FPGA fabric with HPS logic and vice versa. It also shows, in parenthesis, their clock domains.

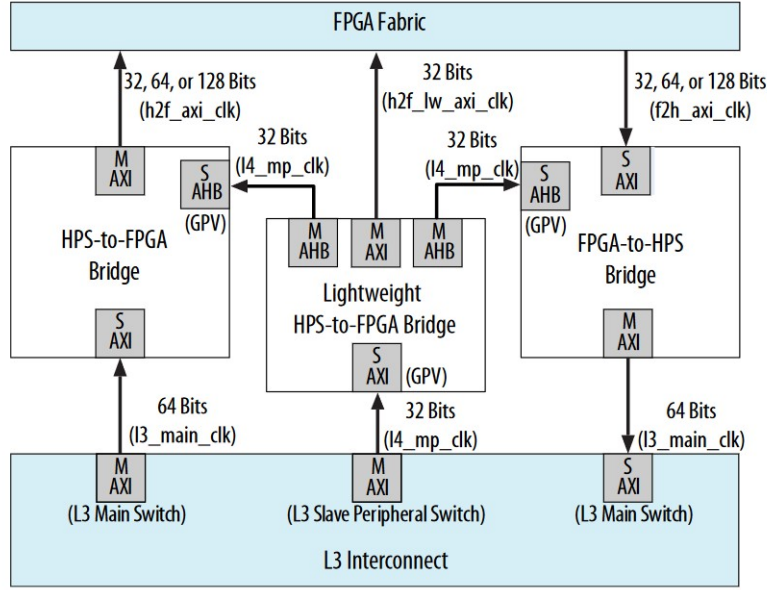


Figure 2.2: AXI interfaces

According to prototype specifications, the system must be as fast as possible. To achieve the best performances both lightweight and fast AXI interfaces are used. Indeed the former could be used to divert part of the traffic from the latter, improving the overall performance of the system. Therefore, the lightweight bridges carry the following signals:

- the control status register signals of the input port of HPS-TO-FPGA fifo;
- the control status register signals of the output port of FPGA-TO-HPS fifo;
- the signals at the input of the parallel input \output called LEDR_pio.

The high-performance AXI interfaces transport big data like message or secret key. These connections are specified through the *System Contents* tab of Qsys.

2.3.2 | Avalon FIFO memories

To buffer data and control the data flow in the Qsys system two on-chip FIFO memory core are instantiated. One, called HPS_TO_FPGA_fifo, manages the traffic between the HPS logic and the FPGA fabric. The other, called FPGA_TO_HPS_fifo does the same thing, but with the data from FPGA to HPS.

The two fifos help to guarantee design flexibility and an efficient data transport. Indeed, these memories can operate with a single clock, but also with separate clocks for input and output ports. The input interface

may be an Avalon Memory Mapped write slave or an Avalon Streaming sink, while the output one an Avalon Memory Mapped read slave or an Avalon Streaming source. Whatever configuration is chosen, the first data that arrives at the input is also the first data to be delivered to the output. Both in single and dual clock-mode, there is an optional interface called status interface which provides information about the state of the memory. For example, a *waitrequest* signal is asserted or not according to the fifo fill level. For write operations, it is asserted when the memory is full, while it is deasserted if there is enough space to store data. For read operations, it is asserted when the memory is empty, while it is deasserted when there is at least one data to read.

Since there is no need to use them for streaming data, both the two fifos are configured as Avalon Memory Mapped write slave to Avalon Memory Mapped read slave. In this mode, an Avalon-MM write master stores data into the memory by writing to the input port, and an Avalon-MM read master pops data out the memory by reading from its output port.

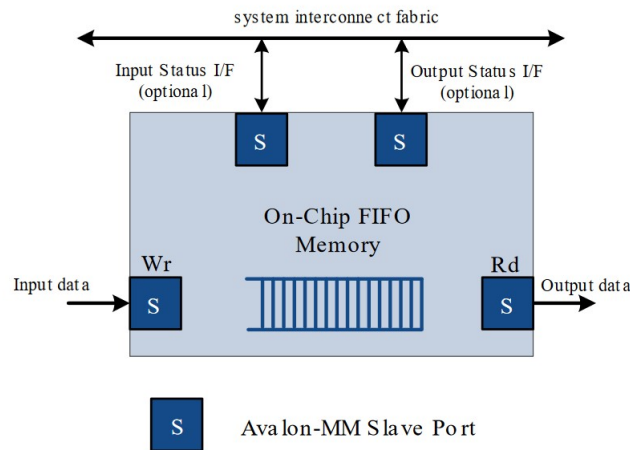


Figure 2.3: Avalon-MM write slave to Avalon-MM read slave

Picture 2.3 is taken from the embedded IP user guide offered by Intel and is a summary of a FIFO configured as Avalon-MM write slave to Avalon-MM read slave.

Since the HPS logic and the FPGA fabric may live in different clock domains, a wise choice is using the two memories with separate clocks for input and output. This choice also affects the status signals of the two ports.

The two fifos have depth equal to 8192, while their width is 32 bits. If necessary, both depth and width can be changed.

As mentioned above, the status interface for both input and output is optional: it can be included or not. In the Qsys system, they are present

for both input and output for both fifos.

2.3.3 | SDRAM controller

To connect easily the 64 MB SDRAM in the Qsys system a SDRAM controller is used. The SDRAM controller helps to reduce design time and efforts. Indeed, it handles all SDRAM protocol requirements. It has an Avalon Memory Mapped Interface to off-chip SDRAM and guarantees access to SDRAM with various sizes, widths and multiple chip selects. The Avalon-MM interface also allows pipelined read operations.

To handle correctly read and write operations, the SDRAM controller configuration has to match the one of the 64 MB SDRAM. Therefore, its clock frequency must match the SDRAM one. Also, address, data and control signals on SDRAM pins have to be stable when a clock edge arrives. A PLL may be useful to reduce clock skew between the controller and the SDRAM. If at low frequencies the PLL may be unnecessary, at high ones it is the only way to ensure stability of the signals at the SDRAM pins at the clock edge. For example, the PLL may add a phase shift to the SDRAM clock so that when the edge arrives the signals from the controller are already stable.

Figure 2.4 is taken from embedded IP user guide offered by Intel and shows the SDRAM controller and how it may be integrated with an SDRAM.

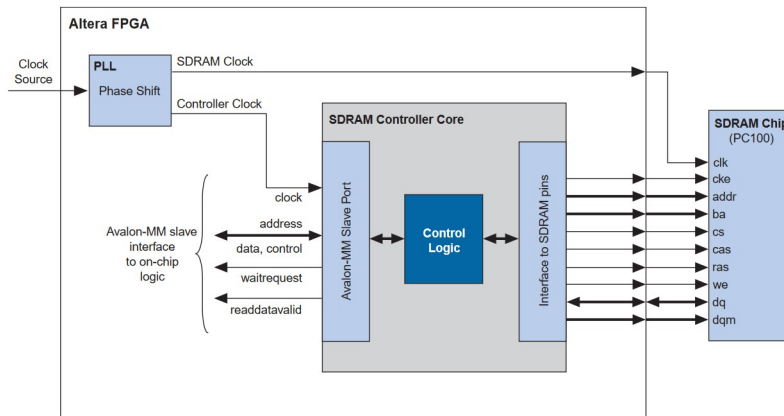


Figure 2.4: SDRAM controller block diagram

After opening the Qsys windows, an SDRAM controller can be chosen from the IP catalog and added to the system. Clicking on it, its configuration window appears. It has two tabs: the first is the *Memory Profile*, the second the *Timing*.

In the *Memory Profile* page it is possible configuring the data width, architecture and the address width of the memory to control. According

to the datasheet of 64 MB SDRAM, the data width is set to 16 bits, the number of chip selects to 1, the number of the banks to 4, the address row and column width respectively to 13 and 10.

In the *Timing* page it is possible choosing the right timing parameters for the controller. According to the SDRAM datasheet, the number of CAS latency cycles is set to 3, the initialization ones to 2. The CAS latency tells how many clock cycles there are from a read command to data out. The initialization refresh cycles are the refresh cycles the SDRAM controller performs as part of the initialization sequence after reset. The other SDRAM controller timing parameters are:

- how often the SDRAM controller refreshes the SDRAM (7.8 μ s);
- the delay after power up and before SDRAM initialization (100.0 μ s);
- how long is the refresh command (70.0 ns);
- how long is the precharge command (15.0 ns);
- ACTIVE to READ or WRITE delay (15.0 ns);
- access time, which depends on CAS latency (5.5 ns);
- write recovery time for precharge commands (14.0 ns).

The Avalon Memory Mapped interface simplifies a lot read and write operations to the 64 MB SDRAM. It also allows the slave to stall whatever operation as many cycles as required by asserting the *waitrequest* signal. There is only one rule: if the slave uses this signal for either read or write transfers, it has to use the signal for both.

Typically signals like *address*, *read* or *write*, *writedata*, *byteenable* arrive before the rising clock edge. At this point the slave raises the *waitrequest* signal to hold off the subsequent transfers if there are any and to hold constant the signals just received. The transfer ends at the first rising clock edge after the slave interface deasserts the *waitrequest* signal. The slave can stall the communication how long it needs. Picture 2.5, taken from Avalon Interface Specifications offered by Intel, shows a timing diagram about read and write transfers with the *waitrequest* signal.

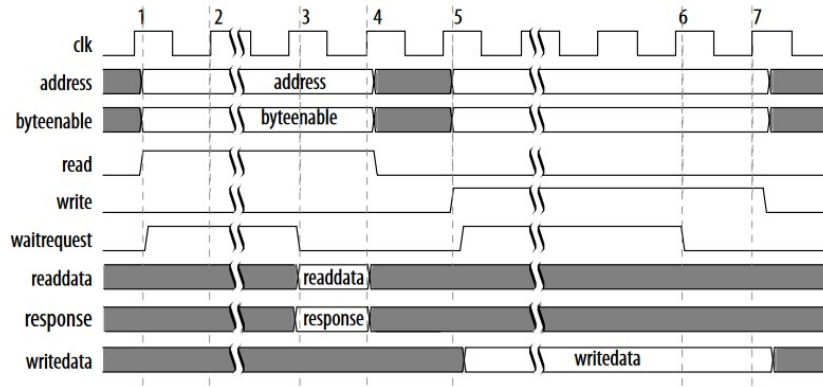


Figure 2.5: Avalon-MM read and write requests with *waitrequest* signal

2.3.4 | Parallel Input\Output component

The PIO core added to the Qsys design is used by the HPS to signal to the Q-decoder that the message and the private key are ready to be read from the `HPS_TO_FPGA_fifo`.

The core provides an Avalon Memory Mapped interface between an Avalon-MM SLAVE port and general purpose I/O ports. In other words, this IP component offers easy I/O access to user logic or external devices when a simple "bit banging" approach is enough.

In the prototype the PIO core is used to control the red leds of the DE1-SoC board. Checking their state, on or off, the Q-decoder knows when the message and the private key are ready to be read from the `HPS_TO_FPGA_fifo`.

Since, the red leds can be turned on or off also for debug purposes, the Qsys system controls all the ten red leds of the board. As a consequence, the width of the PIO core is set to 10 and its direction to Output. The reset value of the output ports is `0x000000000000003ff`.

2.3.5 | Clock Sources and phase locked loops

The Altera DE-series boards provides an on-board oscillator which generates a 50 MHz clock and this clock signal is connected to one or more pins usually called `CLOCK_50`, `CLOCK2_50` and so on.

The Qsys system requires more than one clock with a frequency higher than 50 MHz and different phase shifts. Specific clocks can be generated with one or more PLLs.

Two PLLs are used and two different IP components are chosen. The first is an IP component called *Altera PLL*. The second belongs to the class of *System and SDRAM PLL* cores. The Altera PLL is more complex than the other and generates more than one clock signal, all for the FPGA part of the prototype. The System PLL is used to generate one clock signal shared by different IP components in the HPS block of the

Qsys design. Both take a reference clock signal of 50.0 MHz.

The System PLL generates a clock signal of 143.0 MHz, which is used as the clock input of all AXI bridges, the PIO component, the HPS_TO_FPGA fifo input port and FPGA_TO_HPS fifo output port.

The Altera PLL generates three 120 MHz clock signals: the first for the 64 MB SDRAM, the second for the SDRAM controller, the HPS_TO_FPGA fifo output port and FPGA_TO_HPS fifo input port, the third for the Q-decoder. The SDRAM clock signal has a phase shift of -52.0 degrees. This shift helps to reduce the clock skew between the memory and its controller. The other two clocks are almost identical. They have the same frequency and no phase shift. The only difference between the two is that the Q-decoder clock signal is exported, the other not. In Qsys exporting a signal is the only way to make it visible from the outside. Qsys allows to export both input and output signals.

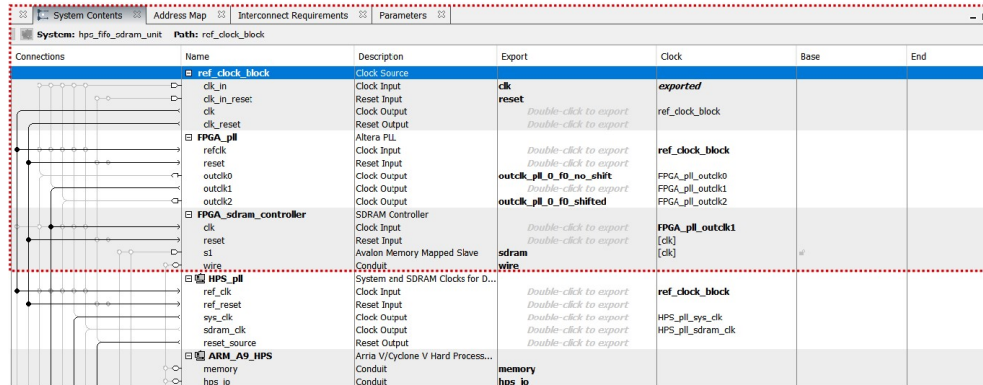
If necessary, the chosen frequencies for the HPS and FPGA blocks can be changed. There is only one constraint: they have to be compatible with the Qsys IP components. For example, the 64 MB SDRAM allows a clock signal of frequency equal or less than 143.0 MHz.

2.3.6 | Reset signals

All the IP components share the same reset signals except for the Hard Processor System, which has its own and the fifos. About these two memories, the reset signal of both input and output ports is the OR between two reset signals: the one of the FPGA subsystem and the one of the AXI bridges.

2.4 | The Qsys design

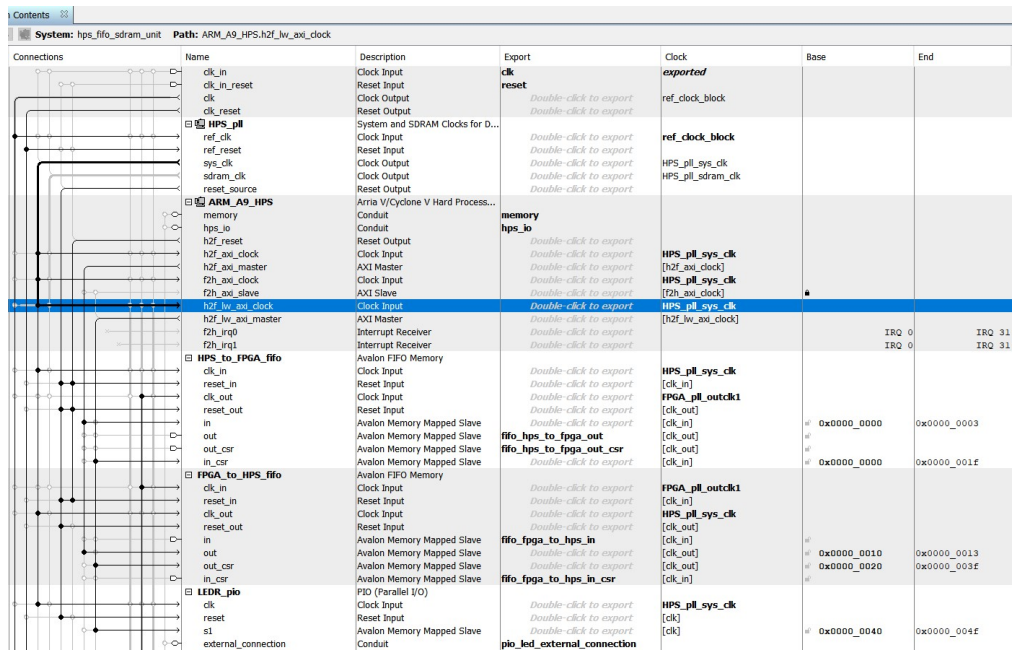
The final result is the Qsys design shown in figures 2.6 and 2.7. The first shows the FPGA subsystem, in the red square, the second the HPS one.



The image shows a Qsys design window with a red dashed box highlighting the FPGA subsystem components. The components listed are:

- ref_clock_block**: Clock Source. Exports: clk, reset. Clock: ref_clock_block.
- FPGA_pll**: Altera PLL. Exports: reset, outclk0, outclk1, outclk2. Clock: ref_clock_block.
- FPGA_sdrn_controller**: SDRAM Controller. Exports: reset, s1, w1r0. Clock: ref_clock_block.
- HPS_pll**: System and SDRAM Clocks for D... Exports: ref_clk, ref_reset, sys_clk, sdrn_clk, reset_source. Clock: ref_clock_block.
- ARM_A9_HPS**: Arria V/Cyclone V Hard Process... Exports: memory, hps_io. Clock: HPS_pll_sys_clk, HPS_pll_sdrn_clk.

Figure 2.6: Qsys design - FPGA IP components



The image shows a Qsys design window with a red dashed box highlighting the HPS subsystem components. The components listed are:

- ref_clock_block**: Clock Source. Exports: clk, reset. Clock: ref_clock_block.
- HPS_pll**: System and SDRAM Clocks for D... Exports: ref_clk, ref_reset, sys_clk, sdrn_clk, reset_source. Clock: ref_clock_block.
- ARM_A9_HPS**: Arria V/Cyclone V Hard Process... Exports: memory, hps_io. Clock: HPS_pll_sys_clk, HPS_pll_sdrn_clk.
- h2f_axi_clock**: Clock Input. Exports: h2f_axi_master, h2f_axi_slave. Clock: HPS_pll_sys_clk.
- HPS_to_FPGA_fifo**: Avalon FIFO Memory. Exports: clk_in, reset_in, clk_out, reset_out, in, out, in_csr, out_csr. Clock: HPS_pll_sys_clk.
- FPGA_to_HPS_fifo**: Avalon FIFO Memory. Exports: clk_in, reset_in, clk_out, reset_out, in, out, in_csr, out_csr. Clock: HPS_pll_sys_clk.
- LEDR_pio**: PIO (Parallel I/O). Exports: reset, s1, external_connection. Clock: HPS_pll_sys_clk.

Figure 2.7: Qsys design - HPS IP components

Ethernet client-server model

The *client-server model* is a network architecture in which a connection between a *client* and a *server* is established. The former requests a service, which is provided by the latter. There is also the possibility to have more than one client.

The words server and client have different meanings than *server-host* *client-host*. The first may refer either a computer or a computer program. The meaning of host is more specific and less ambiguous: it refers to a computer connected to a network. Therefore, server-host and client-host denotes computers connected to a network in which different parts of an application run.

The prototype implements a client-server model, in which a client-host sends an encrypted message via the Ethernet to the server-host, which decodes it. The two computer terminals run two different parts of a distributed application. This computer program uses what is called *socket programming*.

3.1 | Socket programming

In C language, there are a lot of ways to establish a communication between two nodes in a network. Socket programming is one of the simplest and effective. The nodes are also called sockets.

To establish the communication, there must be a listener, which listens on a particular port at a specific IP. Usually, it is the server. While it listens, another socket forms the connection reaching out the listener.

Two are the headers needed by socket programming: `<sys/socket.h>` and `<netinet/in.h>`.

3.1.1 | Server steps

These are the steps the server has to follow to establish a working and efficient socket communication:

1. socket creation, whose descriptor is the return value of function `int socket(int communication_domain, int communication_type, int internet_protocol);` the communication domain is `AF_INET`,

the macro for Internet domain sockets, the `communication_type` `SOCK_STREAM`, which is a byte-stream communication, the Internet protocol 0;

2. socket binding to the correct port and address, done using the function `int bind(int socket_descriptor, const struct sockaddr *addr, socklen_t addrlen)`; `addr` is a structure which contains information like the socket family, the IP address and the port number; the return address is an integer and is used to check if the binding ends correctly;
3. listening for a connection, done using the function `int listen(int socket_descriptor, int backlog)`; the integer `backlog` defines how many connections can be queued and if the connection arrives when the queue is full, the client receives the error `ECONNREFUSED`;
4. accepting the first connection in the queue, using the function `int accept(int socket_descriptor, const struct sockaddr *addr, socklen_t *addrlen)`; the return value is the file descriptor of the connected socket corresponding to the listened socket `socket_descriptor`.

After accepting, the connection is established and client and server can exchange data.

An optional step can be done before the socket binding.

- Using the function `int setsockopt(int socket_descriptor, int level, int optname, const void *optval, socklen_t optlen)`, it is possible manipulating the options of then listened socket. The `level` is the level at which the option belongs; `optname` and `optval` are the name and the value of the option to set. This function allows to reuse IP addresses and ports.

Because it is optional and not necessary, it is not used in the prototype client-server model.

3.1.2 | Client steps

Before exchanging data, the client has to connect to the socket listened by the server. The function `int connect(int socket_descriptor, const struct sockaddr *addr, socklen_t addrlen)` establishes the link to the socket using its descriptor. The struct `const struct sockaddr *addr` contains the address and the port number of the server socket.

3.2 | The client-server computer program

The program to send a message to the server by the client is a distributed application. In other words, it is split in two parts, one run on the server-host, the other on the client-host.

The whole program is written in C language and it is organized into four files. The one called *server_fpga_com.c* implements the server. The second, named as *client_server.c*, runs on the client. The third, renamed as *common_functions.c*, contains all the functions used by both the server and the client. The fourth, called *common_functions.h*, is a header file. It contains macros and function prototypes.

This modularity guarantees efficiency, flexibility, higher readability.

3.2.1 | server_fpga.c

The *server_fpga.c* file contains the following functions:

- *int main(int argc, char *argv[]);*
- *void fromClient2READ_FIFO (int *newsockfd, int *message_length);*
- *void fromLtrFiles2READ_FIFO (int *message_length);*
- *void fromREAD_FIFOtoClient(int *newsockfd).*

Obviously, it also uses the ones in the *common_functions.c* file.

main(). The *main()* function returns an integer and receives as input argument the port number of the socket connection. It returns '0' if the whole process ended successfully, '1' if not.

After setting and accepting the socket connection, it gets the addresses of HPS_TO_FPGA fifo input port and status register, FPGA_TO_HPS output port and status register, and LEDR_pio. The functions *open()* and *mmap()* are used, the first to open the device */dev/map*, the second to map the address space of the process to the memory object */dev/map*. During this step, also the base and end address visible in pictures [2.6](#) and [2.7](#) are used.

Then, all the red leds of DE1-SoC are initialized to '1'. '1' means that the led is turned off. The main is ready to receive the message from the client and to write it into the HPS_TO_FPGA fifo. This is done by function *fromClient2READ_FIFO()*. Also the private key is written, so the function *fromLtrFiles2READ_FIFO()* is called.

After writing the private key, the LEDR are set to '0'. They turn on: the Q-decoder can read them from the fifo.

For debug purpose, after decoding the message, the Q-decoder writes it on the FPGA_TO_HPS fifo. The main waits for the message through a while loop, which checks if the fifo has at least a valid data. When

the decoder starts to write on the fifo, the main calls the function `from_READ_FIFOtoClient()`, which read the message and send it to the client. At the end, the main closes the connection and the listened sockets.

fromClient2READ_FIFO(). The function reads integers from the client and writes them on `READ_FIFO`. Its input arguments are the integer pointers `newsockfd` and `message_length`. The first points to the connection socket, the second to a variable which stores the length of the message to receive. It uses the function `error()` of the `common_functions.c` file. The function does not return any value.

The `read()` function is used to read from the connection socket. The first data received is the message length. In this way, the server knows how many packets it has to read from the socket. Every packet is read and immediately written on `HPS_TO_FPGA` fifo. The length is written as the address the last 16 bits of the message will have in the SDRAM.

fromLtrFiles2READ_FIFO(). The function reads integers from the Ltr files and writes them on `HPS_TO_FPGA` fifo. The Ltr files contain the private key. Its input arguments are the pointers to the connection socket and to the message length variable. It uses the function `error()` of the `common_functions.c` file. The function does not return any value.

Before reading the Ltr files, it analyses the lengths of their "binary strings": the two files contain strings of only '0' and '1' chairs. Their parallelisms have to be equal to each other and equal to 16 bits.

To speed up the whole process, before writing them on the fifo, the binary strings are packaged together in 32-bit integers. 32-bit is also the width of the fifo. The two most significant bytes of the packet are a binary string from the *Ltr2.txt* file. The least significant ones are a binary string from the *Ltr1.txt* file.

The first data written on fifo is the address the last 16 bits of the private key will have in the SDRAM. Every packet is read and immediately written on `HPS_TO_FPGA` fifo.

from_READ_FIFOtoClient(). The function reads integers from `FPGA_TO_HPS` fifo and send them to the client. The reading process ends when the fifo becomes empty. It has no return values and one input argument, which is the pointer to the connection socket. It uses the function `error()` of the `common_functions.c` file.

3.2.2 | client_server.c

The *client_server.c* file contains the following functions:

- `int main(int argc, char *argv[]);`

- *void msgTransmit (char *msg1_filename, char *msg2_filename, int msg1_parallelism, int msg2_parallelism, int msg1_packets_number, int *sockfd);*
- *void ReceivingDecodedMessage (int *sockfd, int msg1_packets_number).*

Obviously, it also uses the ones in the *common_functions.c* file.

main(). The *main()* function reads the message from the files *Msg1.txt* and *Msg2.txt* and send it to the server. It returns an integer and receives as input argument the host name and the port number of the socket connection. It returns '0' if the whole process ended successfully, '1' if not. The first step is to connect to the socket. Then, the parallelisms of the message files are computed. They have to be equal to each other and equal to 8 bits. The message is sent using the *msgTransmit()* function. For the debug purposes, after sending the encrypted message, the client reads from the socket the decoded message. The function *ReceivingDecodedMessage()* is used.

At the end the socket is closed and the *main()* function ends.

msgTransmit(). The function opens the two files containing the message, packages one or more rows of each file into 32-bit packets and sends them to the server. Its input parameters are:

- *msg1_filename*, the pointer to the string which stores the name of the file containing the first half of the message;
- *msg2_filename*, the pointer to the string which stores the name of the file containing the second half of the message;
- *msg1_parallelism*, the length of each binary string in the file containing the first half of the message;
- *msg2_parallelism*, the length of each binary string in the file containing the second half of the message;
- *msg1_packets_number*, the length of the file in 16-bit packets;
- *sockfd*, the pointer to socket.

The function does not return any value. It uses the function *error()* and *binstr2int()* of the *common_functions.c* file.

The message files contain strings of only '0' and '1' characters. Communication takes place transmitting 32-bit integers. Instead of transmitting eight bytes for each row of the two files, the data which is sent is the corresponding integer of one or more rows packaged together. In this way, four rows occupy only 4 bytes. The first and the third least significant bytes of the integer are two consecutive rows from *Msg1.txt* file, the second and the fourth two consecutive ones from *Msg2.txt* file. First, the

client sends to the server how many packets it has to receive. Then it starts to deliver the message.

ReceivingDecodedMessage(). The function receives from the server a '0' or a '1' according to how ended the decoding ('0' stand for failure, '1' for success) and then the decoded message. While it receives the message, it also writes it on a file called *DecMsg.txt*. This file will contain strings of only '1' and '0' chars. In case of failure, the first line of the file contains a warning message. Its input arguments are *msg1_packets_number* and *sockfd*, which are respectively the length of the file in 16-bit packets and the pointer to the socket. The function does not return any value. It uses the function *error()* of the *common_functions.c* file.

3.2.3 | common_functions.c

The header files called *common_functions.c* contains the following functions, which are common to both client and server .c files:

- *void error(const char *msg);*
- *int binstr2int(const char *buffer1, const int representation);*
- *void msgfileOverview (char *msg_filename, int *msg_parallelism, int *msg_packets_number, int mode).*

error(). The function invokes *perror()* to signal an error on *stderr*. Its input parameter, which is a pointer to a string, is passed to *perror()*. It returns the integer '1'.

binstr2int(). The function converts an unsigned or two's complement binary string into its corresponding integer value. A binary string is a string of only '0' and '1' characters. Its input arguments are *buffer1*, which is the pointer to the binary string, and an integer called *representation*, which can be '0' or '1' depending on how the binary string has to be interpreted. '0' stands for unsigned, '1' for signed. The return value is the corresponding integer of the string.

msgfileOverview(). The function opens a file each row of which is a binary string longer always the same. '0', '1' and '\n' are the only characters allowed. Then, it finds the length of binary strings and computes how long the file is in 16-bit packets. Its input arguments are:

- *msg_filename*, the pointer to the string containing the name of the file;
- *msg_parallelism*, the pointer to a variable which stores the parallelism of message files;

- *msg_packets_number*, the pointer to a variable whose value is the length of the message in 16-bit packets;
- *mode*, the integer variable according to which the user is asked for the file name or not; if mode is set to '0', the user is asked for the file name, if '1', the file name taken into account is the one stored in *msg_filename*.

It uses the function *error()* of the *common_functions.c* file. The function does not return any value.

3.2.4 | common_function.h

This header file contains the prototypes of the functions in *common_functions.c* file. There are also two macros:

- *packet_dimension*, which is the size in number of bits of each packet exchanged between client and server;
- *debug*, for debug purposes.

The *debug* macro is used for conditional compilation. If it is set to '1', it enables instructions which help to understand how the whole client server communication works. They are calls to *printf()* function, so they do not interfere with the program normal behaviour.

3.3 | Makefile

To compile the *.c* files, a makefile is used. A makefile is a file in which a set of directives is written. These are then used by make build automation tool to generate one or more targets. Writing a makefile makes the compilation faster, more intuitive and easier, since:

- it keeps track of compilation settings;
- it helps to detect user oversights and errors during the whole building process;
- there is no need to compile the entire program every time a change to a functionality or a class is needed: only changed files are compiled;
- it helps to present the whole project in a more systematic and efficient way.

Since the client and server programs have to run on different hosts, the compiler toolchain for *server_fpga.c* is different from the *client_server.c* one. However, there is the possibility to run both on the same host. This opportunity may be useful for debug purposes. The steps to run both client and server on the DE1-SoC HPS are:

1. compiling *server_fpga.c* and *client_server.c* with the compiler toolchain of Arria V/Cyclone Hard Processor System;
2. booting Linux on the DE1-SoC board through an SD card;
3. connecting the DE1-SoC to a PC establishing a serial communication;
4. opening on the PC the HPS Linux terminal;
5. putting the two compiled files into the SD card;
6. running the server and client programs writing on the HPS Linux terminal the commands `.\<server_compiled_file_name> <port number> &` and `.\<client_compiled_file_name> localhost <port number>`.

The second, third, fourth and fifth steps can be easily done according to the tutorial material of DE1-SoC. Step 4 requires applications like PuTTY.

The compiler toolchain of the first step can be downloaded installing the application called *Intel Soc FPGA Embedded Development Suite*, also known as *SOC EDS*.

Running a makefile is simple. First, the SOC EDS Command Shell is opened. Then, through the linux based *cd* command, the working folder is changed to the one where the makefile is located. At this point, it is enough typing and executing the command *make* on the shell. The makefile will execute the recipe to generate the target called *build*. There are also two other targets: *clean*, to remove the created object and executable files, and *init*, to create the *exe* and *obj* folders, which are the directories where are generated the executable and object files.

If the client and server have to run on different hosts, the client program has to be compiled according to the machine in which will be executed.

Q-decoder architecture

The Q-decoder architecture is similar, but not equal to the one developed by Kristjane Koleci in her master thesis. It cannot be exactly the same, because it would not be compatible to the developed Qsys design. Indeed, it would not have the right tools to read from/write to FIFOs and SDRAM.

The decoder is written in VHDL and hierarchical. The top entity, called *DECODER*, is organized in a control unit and a datapath. The datapath is made up of the following components:

- *QsysSubsystem*, the Qsys system;
- *WritingSDRAM*, to read message and private key from HPS_TO_FPGA fifo and write them on SDRAM;
- *SyndromeCMP*, to find the syndrome;
- *SyndromeAdjustAndWeight*, to adjust the syndrome and compute its weight;
- *CorrelationCMP*, to compute the correlation;
- *ThresholdEvaluation*, to evaluate the threshold;
- *ErrorPos*, to find which bits of the message have to be toggled;
- *MessageUpdate*, to update the message stored in SDRAM;
- *WritingDecodedMsg*, to write the decoded message on FPGA_TO_HPS fifo;
- *Counter*, to count the number of iterations of the whole process.

Each of them, except Counter and HPS_FIFO_SDRAM, has its own control unit and datapath.

The DECODER control unit is quite similar to the one of Kristjane's Q-decoder. The new state *SynAdjWeight* replaces the old states *SynAdj* and *SynWCmp*. The syndrome is adjusted and its weight is computed in one single step. A new state called *WAITING_HS* is added. In it, the

state machine waits for the Hard Processor System to write the needed information in the HPS_TO_FPGA_FIFO. Figure 4.1 shows the whole DECODER CU.

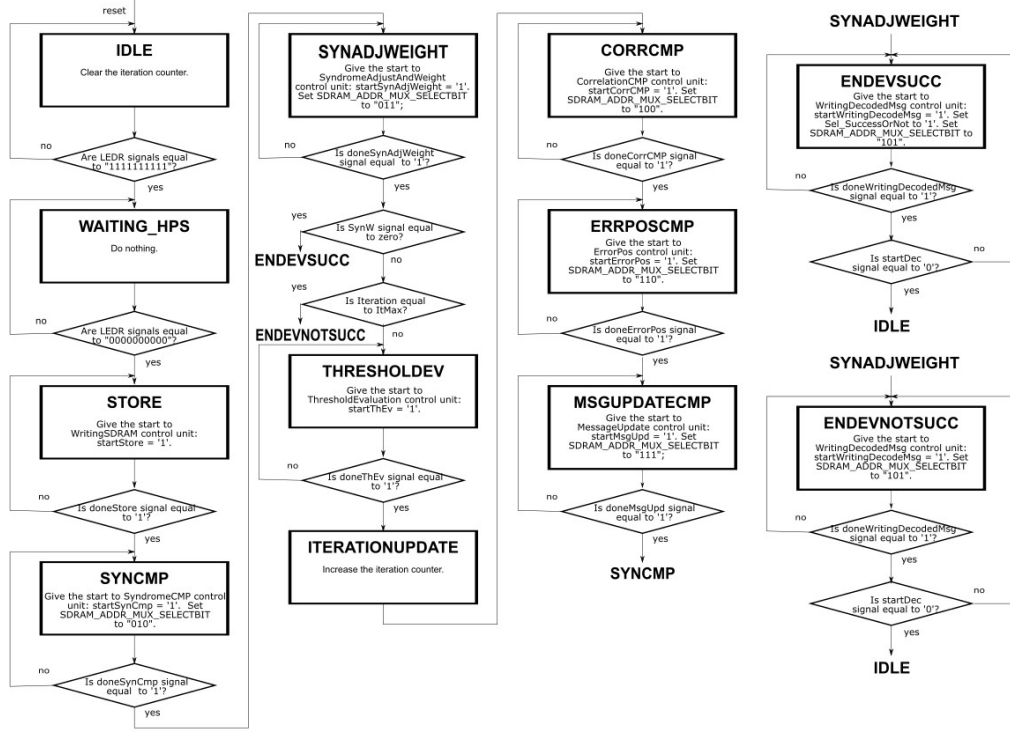


Figure 4.1: Q-decoder control unit

Each state is tracked by a signal called HEX1. It drives one of the 7-Segment displays of DE1-Soc board.

The signal SDRAM_ADDR_MUX_SELECTBIT allows to multiplex the signals to read from and write to SDRAM of the Q-decoder datapath components.

4.1 | QsysSubsystem

The *QsysSubsystem* is the VHDL module which allows to integrate the design developed under Qsys tool with the rest of Q-decoder.

It is made up of only one component, called *hps_fifo_sdram*, which derives directly from the Qsys design. It is the so-called *Instantiation Template*. This template is the entity of the Qsys system in a specific HDL. It is accessible after generating the Qsys design. The available HDL are VHDL and Verilog. The chosen language is VHDL.

QsysSubsystem is just an extra layer between *hps_fifo_sdram* and the other VHDL modules. It is instantiated to enhance the whole prototype flexibility. Indeed, thanks to this unit, it is possible renaming the input and output signals of Qsys design as the user prefer. Furthermore, not

all the signals have to be controlled from outside or are useful. For example, the SDRAM chip select does not need to be controlled: it has to be always active. As a consequence, it remains confined inside the QsysSubsystem and set to '1'.

4.2 | WritingSDRAM

The WritingSDRAM unit reads the message and the private key written from HPS_TO_FPGA fifo and writes them on SDRAM. It has a control unit and a datapath.

4.2.1 | WritingSDRAM_CU

Three processes implement the state machine of WritingSDRAM control unit. For debug purpose, each state could be tracked by HEX0. It is an output signal long seven bits which could drive one of the 7-Segment displays of DE1-SoC board. The states of this control unit are:

- *RESET*;
- *SPILLING_LENGTH*;
- *WRITING_LENGTH*;
- *SPILLING_FIFO*;
- *WRITING_DATA_ON_REG*;
- *WRITING_DATA_ON_SDRAM_L*;
- *SDRAM_ADDR_UPD_1*;
- *WRITING_DATA_ON_SDRAM_H*;
- *SDRAM_ADDR_UPD_2*;
- *END_STORE*;

Figure 4.2 depicts the state machine flowchart.

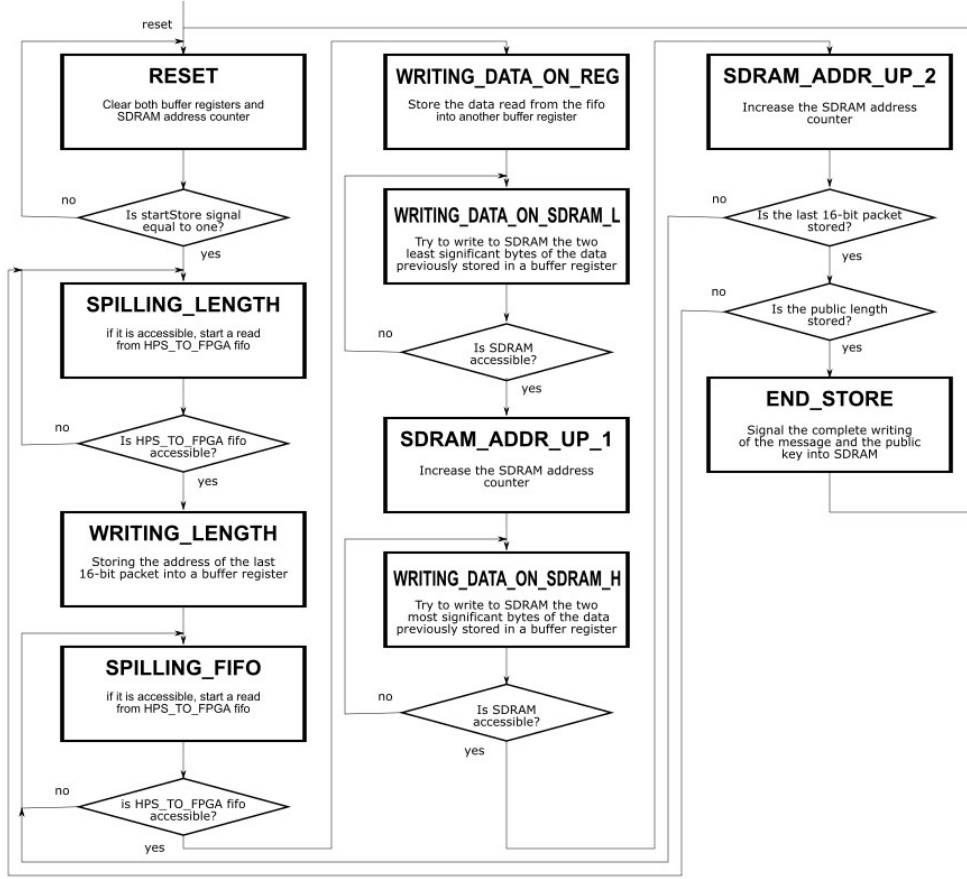


Figure 4.2: WritingSDRAM control unit flowchart

The RESET state sets to zero the content of *sdr_am_addr_increaser*, *H2F_readdata_buffer_register*, *message_length_buffer_register* and *control_bit_from_server*. The state machine evolves in the next state only if the signal called *startStore* is equal to '1'.

During the state called SPILLING_LENGTH, the control unit asks to fifo the last SDRAM address of the message or private key. It is a 32-bit data, but only the least significant 25-bits are stored. Twenty five bits are enough to address the whole 64 MB SDRAM. Asking a data is allowed only if the fifo is not busy. A flag, called *HPS_TO_FPGA_OUT_CSR_READDATA(1)*, tells if the FIFO is busy or not. It is the second less significant bit in the fifo control status register. When the FIFO is not accessible, it is equal to '1'. If it is set to '0', the fifo is readable. The read request is done setting to '1' the control bit called *HPS_TO_FPGA_READ*. If the fifo is busy, the control unit waits in the SPILLING_LENGTH state. Once the fifo becomes readable, the read request is performed and the state machine evolves in the next state.

In WRITING_LENGTH state, the last SDRAM address of the message or private key is stored in the *message_length_buffer_register*.

SPILLING_FIFO is the state in which the control unit asks a new 32-

bit data to the fifo. It is one of the 32-bit packets which compose the message, private key or the transposed private key. The read request is performed in the same way as in SPILLING_LENGTH state.

During WRITING_DATA_ON_REG state, the 32-bit packet read from fifo is stored into the register called H2F_readdata_buffer_register.

The least two significant bytes are written into SDRAM during WRITING_DATA_ON_SDRAM_L state, the most significant ones during WRITING_DATA_ON_SDRAM_H state. Before writing any data in SDRAM, a check if it is accessible and writable is done. If the SDRAM control bit, called SDRAM_WAIT, is equal to '0', writing is possible. If the bit is '1', the memory is busy. If the memory is busy, the control unit has to wait.

The SDRAM address is increased by one in SDRAM_ADDR_UPD_1 state or in SDRAM_ADDR_UPD_2 state.

When the control unit reaches END_STORE state, the signal called doneStore is set to '1'. This means that the the unit has finished its job.

For debug purposes, another state, called WRITING_DATA_ON_F2H_FIFO, could be introduced. It helps to check the correctness of data read from fifo. In this state, the information is stored in FPGA_TO_HPS fifo. Once stored, the hard processor could run a C code and read it.

4.2.2 | WritingSDRAM_DP

The datapath of the WritingSDRAM unit is made up of the following components:

- *H2F_readdata_buffer_register*, the 32-bit register which stores the message or private key packet read from HPS_TO_FPGA fifo;
- *message_length_buffer_register*, the 25-bit register which stores the last message or private key SDRAM address;
- *control_bit_from_server*, the flip-flop which stores '0' or '1' according to which kind of information is being processed;
- *sdram_addr_increaser*, the 25-bit counter needed to correctly address the SDRAM;
- *sdram_inputdata_lh_mux*, a 16-bit multiplexer that selects which of the four bytes read from fifo it is time to store in SDRAM;
- *hex_decoder_for_seeing_data_in_SDRAM*, a decoder for debug purposes.

Figure 4.3 depicts the datapath schematic.

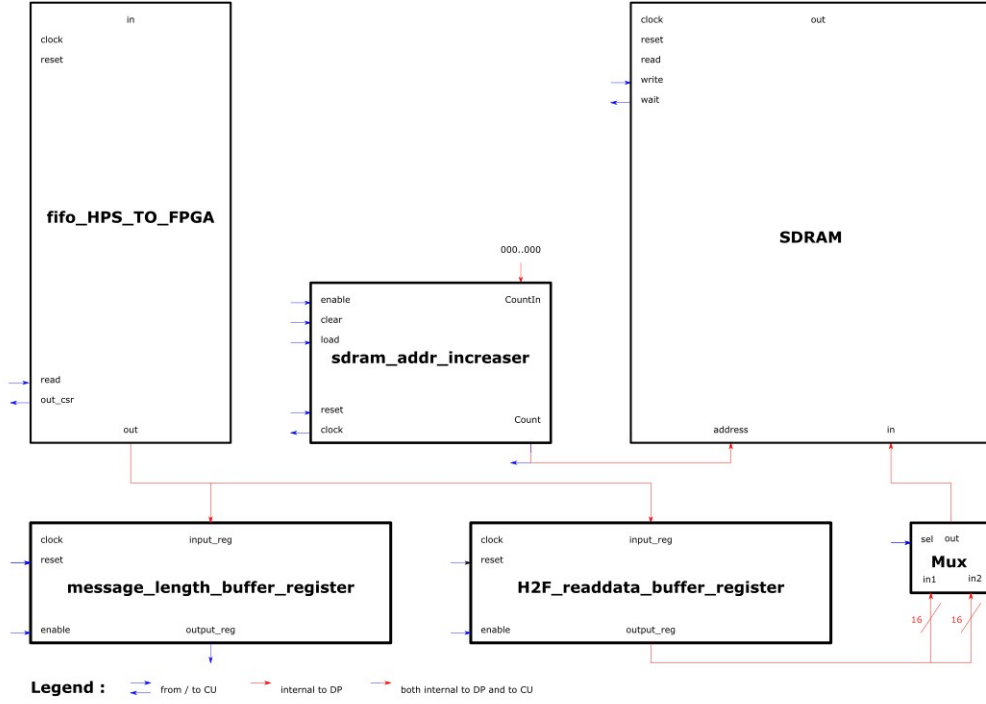


Figure 4.3: WritingSDRAM datapath schematic

The message length is stored as the SDRAM address of the most significant two bytes of its last 32-bit packet. The public length is saved in the same way.

If the control_bit_from_server output is equal to '0', the control unit is reading and writing the message, if it is '1', the control unit is processing the private key or the transposed private key.

Both private key and transposed private key are stored in SDRAM. Each of them is divided in two parts: LtrT1 and LtrT2 are the first and second parts of the private key, Ltr1 and Ltr2 are the first and second parts of the transposed private key. The nomenclature of these four blocks is the same of Kristjane Koleci's Q-decoder. The four blocks are read by the HPS from a single file and then stored in the fifo. Then, the WritingSDRAM unit transfers them in SDRAM. Since Ltr1 and Ltr2 are stored first and Ltr1T and Ltr2T last, the last SDRAM address to save at the beginning of WritingSDRAM state machine is the one of the last byte of Ltr2T, which is the last byte of the private key.

The hex_decoder_for_seeing_data_in_SDRAM output could drive a 7-segment display of DE1-SoC and help when debugging internal signals.

4.3 | SyndromeCMP

The SyndromeCMP unit computes the syndrome. It is almost identical to the unit developed by Kristjane Koleci in her Q-decoder. The differences

concern how the unit interfaces with SDRAM.

SyndromeCMP has a control unit and a datapath.

The control unit differs only for the output signal called *sdram_readdata_sel_bit*. This signal allows to select the right message byte during syndrome computation. The message is read from SDRAM sixteen bits at a time. Only the upper or the lower byte is useful for the computation, according to which part of syndrome is being processed. If the part to compute is the first one, the least significant byte is used. If the part to compute is the second one, the most significant is selected. For debug purpose, each control unit state could be tracked by HEX0. It is an output signal long seven bits which could drive one of the 7-Segment displays of DE1-SoC.

The datapath has different only the input and output signals to read from/write to SDRAM. The rest is exactly same. It includes a component called VectorByCirculant. It computes the multiplication between two circulant matrix. During the syndrome computation, the matrices, which have to be multiplied each other, are the message and the transposed private key.

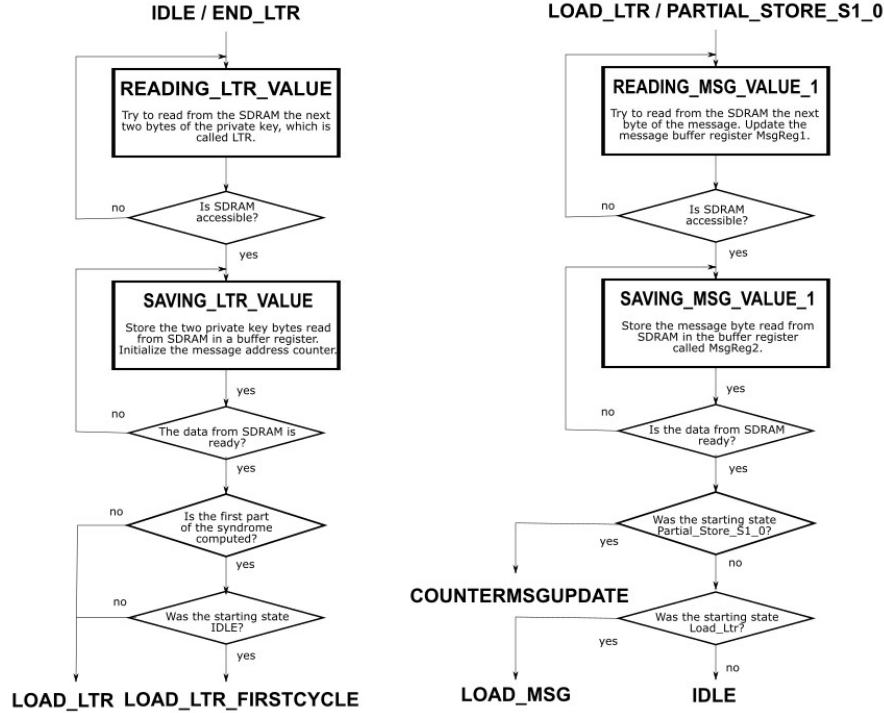
Also the VectorByCirculant component is made up of a control unit and a datapath.

4.3.1 | CU_VectorByCirculant

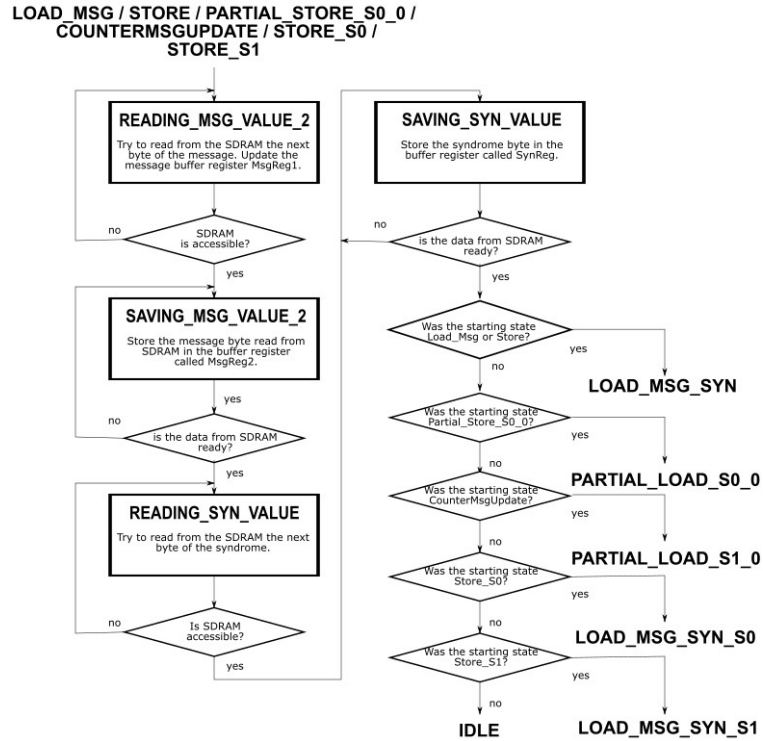
The new state machine of the VectorByCirculant control unit differs from the old one in the following extra states:

- *READING_LTR_VALUE*;
- *SAVING_LTR_VALUE*;
- *READING_MSG_VALUE_1*;
- *SAVING_MSG_VALUE_1*;
- *READING_MSG_VALUE_2*;
- *SAVING_MSG_VALUE_2*;
- *READING_SYN_VALUE*;
- *SAVING_SYN_VALUE*.

Figure 4.4 shows how these states are integrated in the already existing VectorByCirculant state machine. To understand the starting state, three flip flops are used. Each starting state sets them in a different configuration. According to how the flip flops are configured, the control unit evolves in a different state. This strategy helps to avoid dozens of states more.



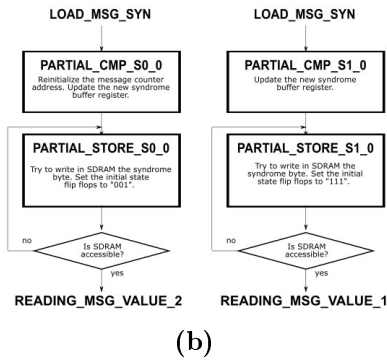
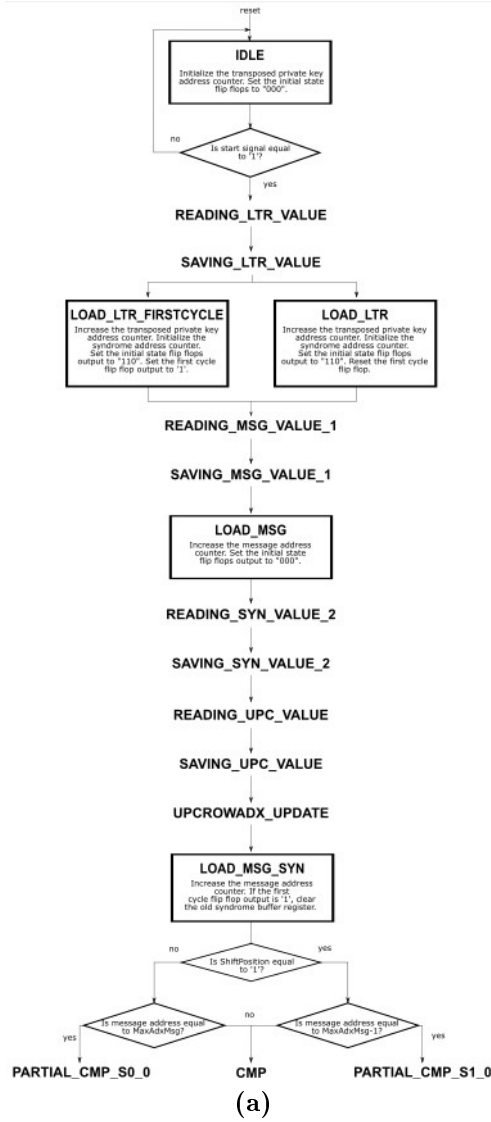
(a) Extra states to read two bytes of the transposed private key from SDRAM (b) Extra states to read a byte of the message from SDRAM



(c) Extra states to read a byte of the message and a byte of the syndrome from SDRAM

Figure 4.4: VectorByCirculant control unit extra states

Figure 4.5 show the other state machine states, already present in the older Q_decoder version.



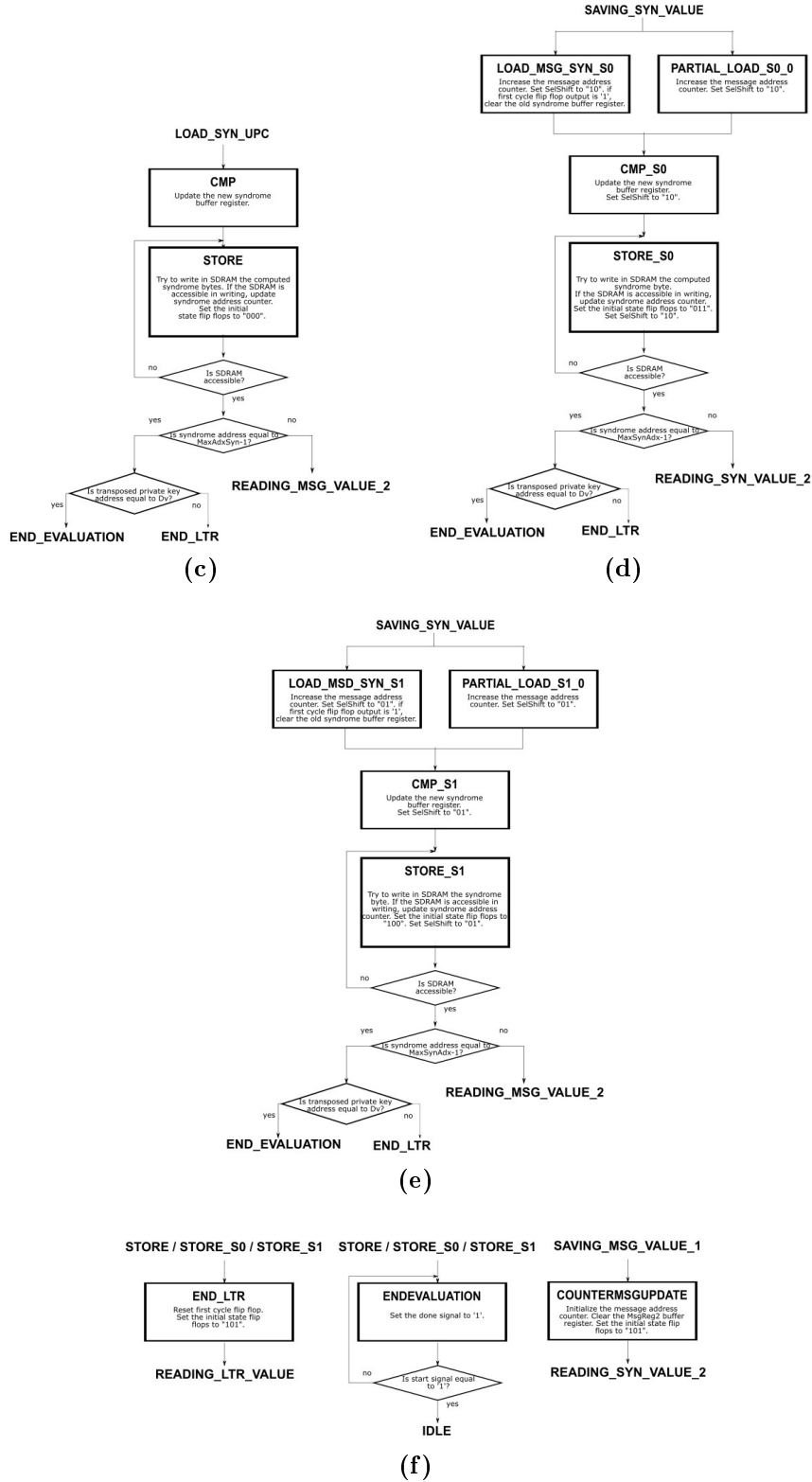


Figure 4.5: VectorByCirculant control unit

4.3.2 | DP_VectorByCirculant

The main changes in the new VectorByCirculant datapath concern how it interfaces with SDRAM. The input and output signals to read from/write to SDRAM are different. The message, transposed private key and syndrome addresses are multiplexed. Also, there are three more flip-flops, whose function is the one described in CU_VectorByCirculant section. The syndrome computation is an iterative process. During the first iteration, the behaviour of the unit is slightly different. The old syndrome register has to be cleared. An additional flip flop helps to discriminate the first iteration from the others. It replaces the latch present in the old Q-decoder version, that causes instability once the FPGA is programmed.

4.4 | SyndromeAdjustandWeight

The SyndromeAdjustandWeight unit adjusts the last syndrome byte if necessary and computes the syndrome weight. It merges two units called SyndromeAdjust and SyndromeWeight in Kristjane's Q-decoder to save control unit states and increase speed. It has a control unit and a datapath.

4.4.1 | CU_SyndromeAdjust_and_Weight

Two processes implements the state machine of WritingDecodedMsg control unit. For debug purpose, each state could be tracked by HEX0. It is an output signal long seven bits which could drive one of the 7-Segment displays of DE1-SoC. This state machine has the following states:

- *idle*;
- *reading_syn_row*;
- *storing_syn_row*;
- *CmpW_Count_incr*;
- *CmpA*;
- *CmpW*;
- *StoreSyn*;
- *EndEv*.

Figure 4.6 depicts the state machine flowchart.

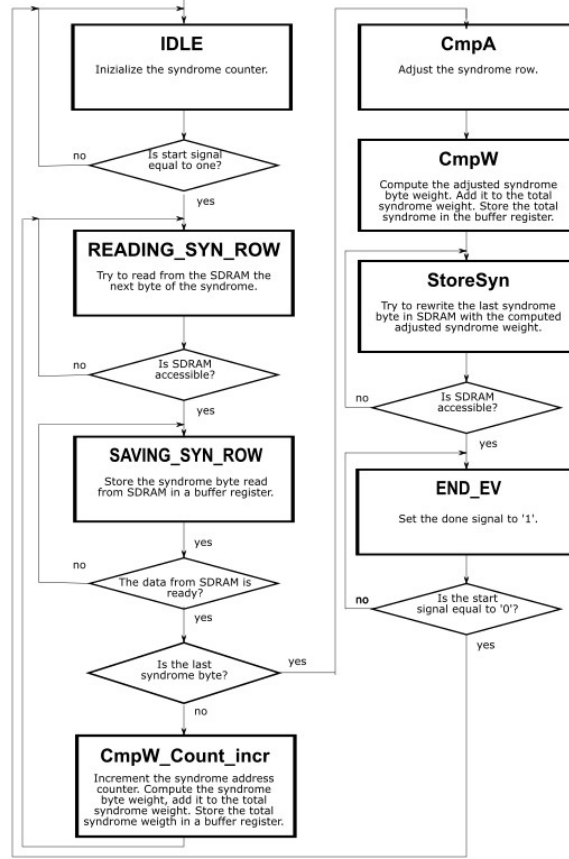


Figure 4.6: WritingDecodedMsg control unit flowchart

In the idle state, the syndrome SDRAM address counter is initialized to the SDRAM address of the first syndrome byte. If the start signal is equal to '1', the state machine evolves into the reading_syn_row state. In reading_syn_row and storing_syn_row states, the least significant byte of the SDRAM row at the syndrome address counter is read and stored in a buffer register.

If it is not the last syndrome byte, its weight is computed and added to the total syndrome weight, that is stored in a proper register and equal to zero if the byte is the first one. This is done in CmpW_Count_incr state. The control unit returns in reading_syn_row state.

If the byte read from SDRAM is the last syndrome byte, it is adjusted in CmpA state. Adjusting means setting to '0' all the bits of the byte which does not belong to the syndrome. Indeed, the syndrome length may not be an exact multiple of one byte. In CmpW state, the weight of the adjusted byte is computed and added to the total syndrome weight. In StoreSyn, the last syndrome byte in SDRAM is overwritten with the adjusted one.

In EndEv, a signal called done is set to '1'. The control unit returns in idle state if the start signal return to '0'.

4.4.2 | DP_SyndromeAdjust_and_Weight

The datapath of the SyndromeAdjustandWeight unit is made up of the following components:

- *SynInReg*, the register which stores the syndrom byte read from SDRAM;
- *SynOutReg*, the register which stores the adjusted syndrome byte;
- *SW_reg*, the register which stores the total syndrome weight;
- *SynCounter*, the counter whose output is the SDRAM address of the syndrome byte to read from or overwrite in SDRAM.

4.5 | CorrelationCMP

The CorrelationCMP unit computes the correlation. The differences with the homonym unit in Kristjane Koleci's Q-decoder concern how the unit interfaces with SDRAM. It has a control unit and a datapath.

The new control unit has no *UpMemCS* signals. The correlation is no longer stored in two memories, each with its own chip select. It is saved in SDRAM and there is no need to drive any chip select signal.

Each of the two correlation parts is computed separately multiplying the syndrome by one of the two private key parts.

For debug purpose, each control unit state could be tracked by HEX0, which could drive one of the 7-Segment displays of DE1-SoC.

The datapath differs on how it interfaces with the SDRAM. The rest does not change. It still includes the component called VectorByCirculantInteger. It computes the multiplication between two circulant matrix in 8-bit integers. The matrices which are multiplied each other are the syndrome and the private key.

Also the VectorByCirculantInteger component is made up of a control unit and a datapath.

4.5.1 | CU_VectorByCirculantInteger

The new state machine of the VectorByCirculantInteger control unit differs from the old one in the following extra states:

- *reading_ltrT_value*;
- *saving_ltrT_value*;
- *reading_syn_value_1*;
- *saving_syn_value_1*;
- *reading_syn_value_2*;

- *saving_syn_value_2*;
- *reading_UPC_value*;
- *saving_UPC_value*;
- *UPCRowAdx_Update*

Figure 4.7 shows how these states are integrated in the already existing VectorByCirculant state machine. To understand the starting state, three flip flops are used. The strategy is the one adopted in VectorByCirculant control unit.

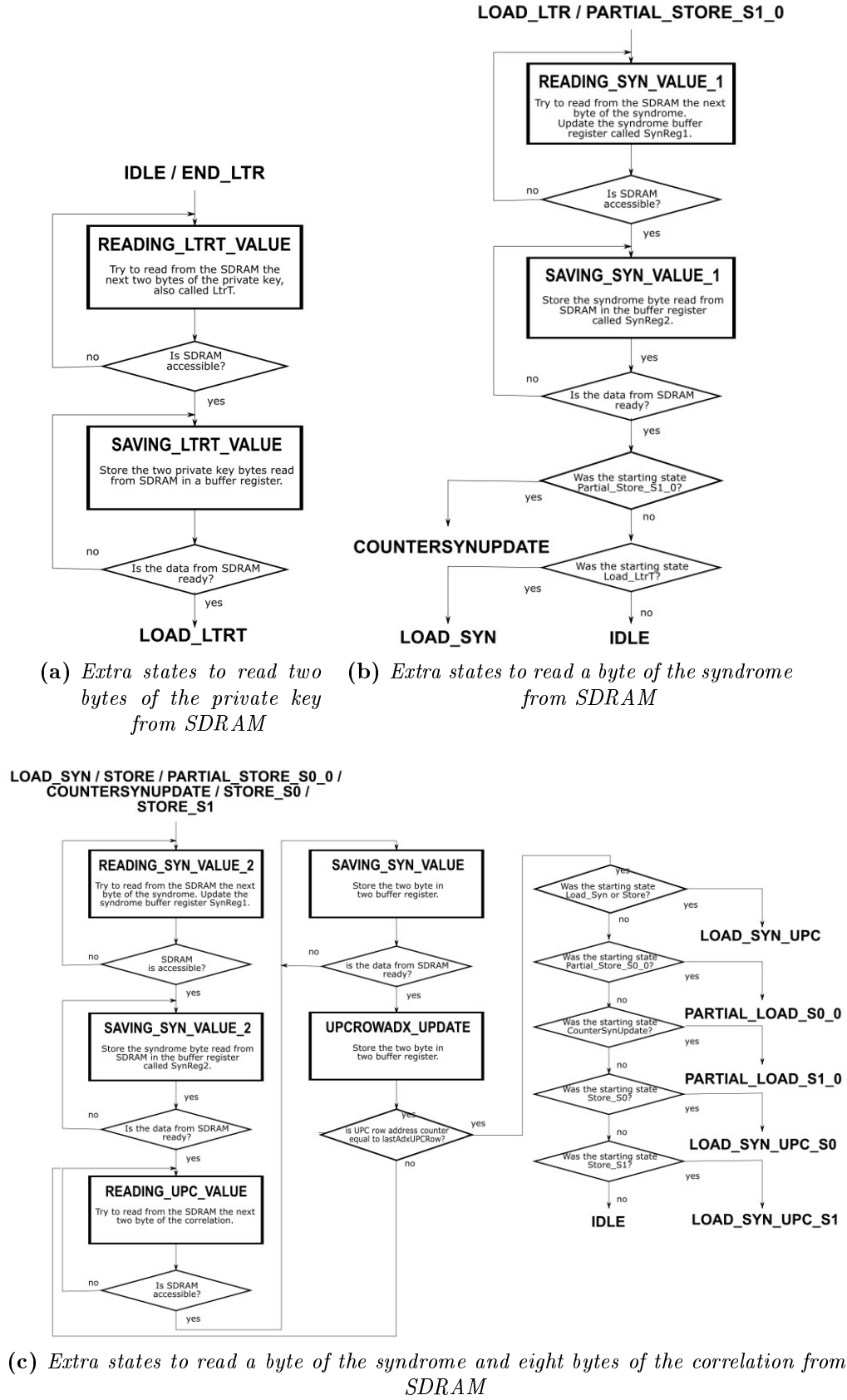


Figure 4.7: VectorByCirculant control unit extra states

Figure 4.8 show the other state machine states, already present in the older Q_decoder version.

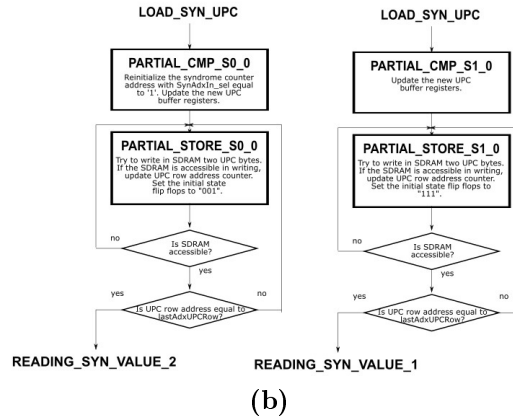
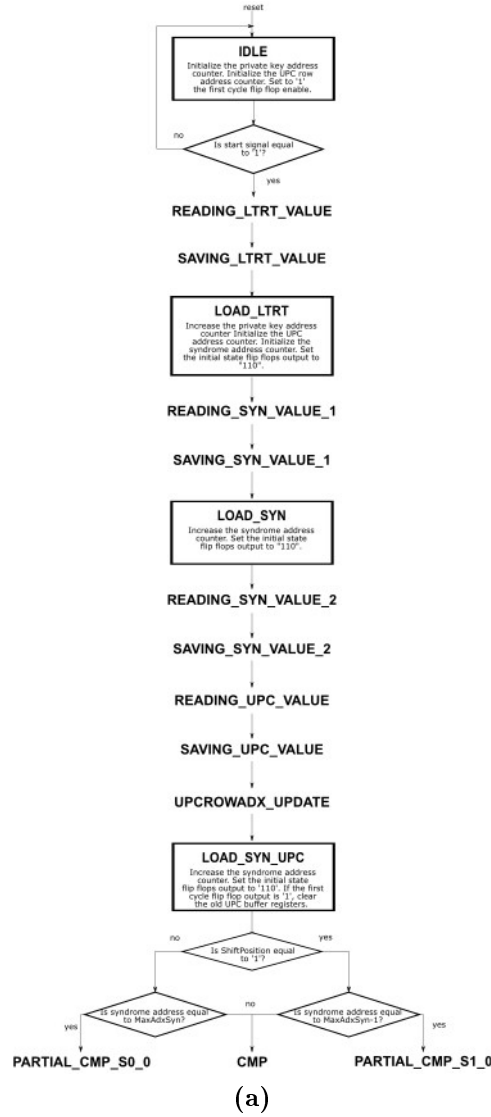




Figure 4.8: VectorByCirculantInteger control unit

4.5.2 | DP_VectorByCirculantInteger

The main changes in the new VectorByCirculant datapath concern how it interfaces with SDRAM. The input and output signals to read from/write to SDRAM are different. The private key, syndrome and correlation addresses are multiplexed.

Each correlation row is eight bytes long and it is read from / write to SDRAM two bytes at time. To write all the bytes, four write requests to SDRAM are needed. Two counters are used to address the correlation bytes in SDRAM. The first is a 23-bit counter and addresses a particular UPC row. The second is a 2-bit counter and addresses a specific couple of bytes in a UPC row.

Also, there are three more flip-flops, whose function is the one described in CU_VectorByCirculantInteger section.

An additional flip flop helps to discriminate the first iteration from the others. It replaces the latch present in the old Q-decoder version, that causes instability once the FPGA is programmed.

4.6 | ErrorPos

The ErrorPos unit finds the message bits to flip before repeating the whole process. These bits are also called error bits. Also, this unit derives from the homonym one in the Kristjane Koleci's Q-decoder version and differs on how it interfaces with SDRAM to get and write information. As an example, the chip select signals *NewPosCS* are no more needed. For debug purpose, each control unit state could be tracked by the HEX0 signal.

It is divided in a control unit and a datapath. The datapath includes the component called ThresholdPos.

Also the ThresholdPos has a control unit and a datapath.

4.6.1 | CU_ThresholdPos

The new ThresholdPos control unit includes the following extra states:

- *reading_UPC_value*;
- *saving_UPC_value*;
- *UPCRowAdx_Update*;
- *Store_first_half*;
- *Store_second_half*.

The *ReadUPC* and *Store* states have been removed.

Figure 4.9 depicts ThresholdPos control unit.

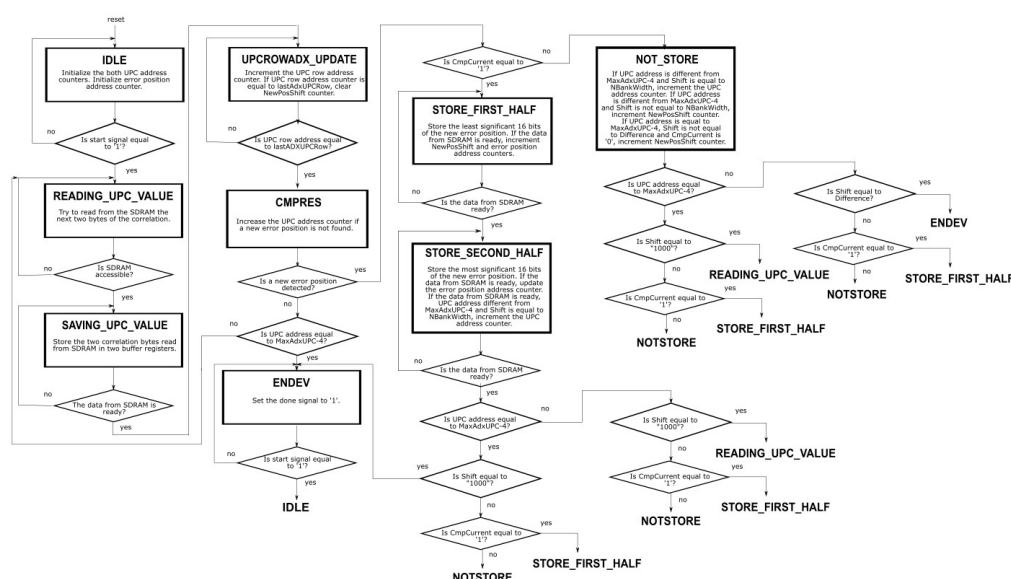


Figure 4.9: ThresholdPos control unit flowchart

4.6.2 DP ThresholdPos

The new ThresholdPos datapath differs in the input/output signals to read from/write to SDRAM. The correlation and error positions addresses are multiplexed.

As in VectorByCirculantInteger unit, the correlation is read two bytes at times and two counters are used to address its bytes in SDRAM.

There is no more need of old ShiftReg and AdxReg buffer registers.

4.7 | ThresholdEvaluation

The ThresholdEvaluation unit is identical to the one in the Kristjane Koleci's Q-decoder.

4.8 | MessageUpdate

The Message Update unit is quite similar to the one in the old Q-decoder. The changes concern the unit interface towards the memory. The two components of the unit are the control unit and the datapath. As in the other units, each control unit state could be tracked by the HEX0 signal. The datapath includes the VectorPosFlip unit.

4.8.1 CU VectorPosFlip

The new CU `VectorPosFlip` includes the following extra states:

- *reading NewPos 1;*

- *saving_NewPos_1*;
- *reading_NewPos_2*;
- *saving_NewPos_2*;
- *reading_Msg*;
- *saving_Msg*;
- *PosAdx_up_1*;
- *PosAdx_up_2*.

Figure 4.10 depicts the VectorByPos state machine.

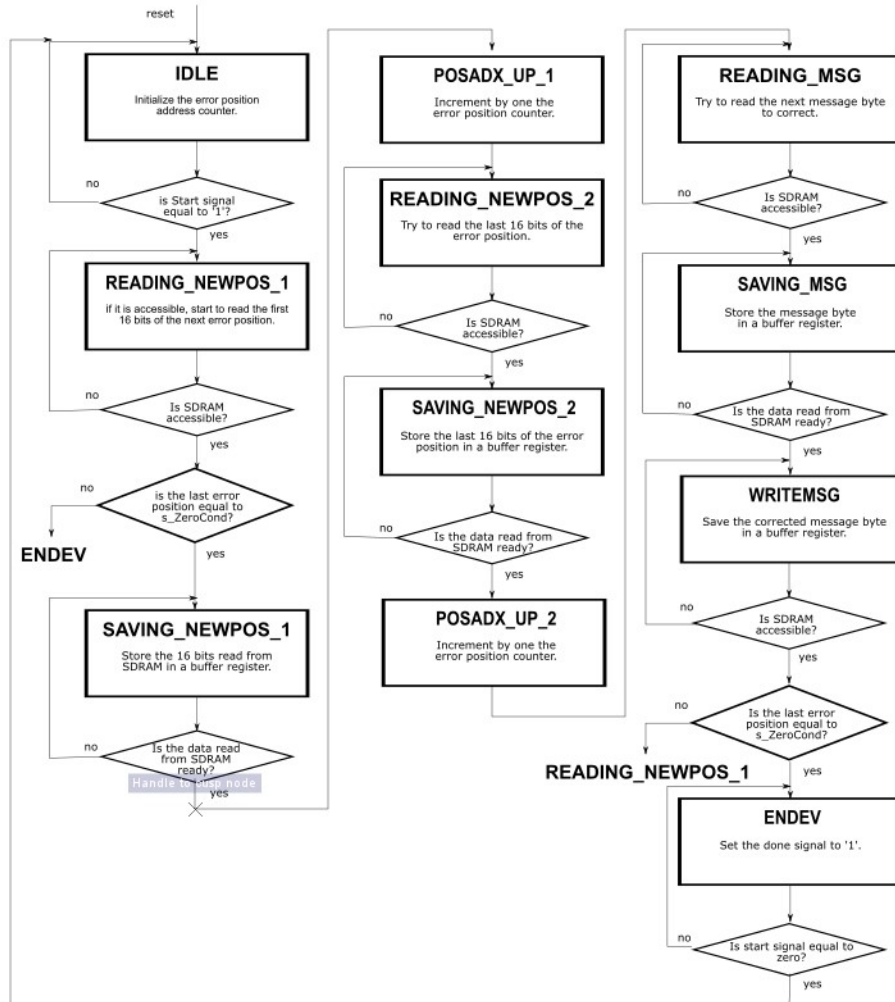


Figure 4.10: VectorByPos control unit flowchart

4.8.2 | DP_VectorPosFlip

The new VectorPosFlip datapath differs in its interface towards SDRAM. The error positions and message addresses are multiplexed. Since it is 32-bits long, each error position requires two read requests to be stored.

4.9 | WritingDecodedMsg

The function of WritingDecodedMsg unit is the one suggested by its name. First, it writes to the fifo called FPGA_TO_HPS fifo a zero or one according to how ended the decoding process. Then, it writes the whole decoded message. Zero means a failed decoding, one success. It has a control unit and a datapath.

4.9.1 | WritingDecodedMsg_CU

Two processes implements the state machine of WritingDecodedMsg control unit. For debug purpose, each state could be tracked by HEX0. It is an output signal long seven bits which could drive one of the 7-Segment displays of DE1-SoC. This state machine has the following states:

- *idle*;
- *WritingDecOutcome*;
- *ReadingSDRAM_1*;
- *BufferingFirst2B*;
- *BufferingFirst2B*;
- *AdxUpd_1*;
- *ReadingSDRAM_2*;
- *BufferingLast2B*;
- *AdxUpd_2*;
- *EndWrite*.

Figure 4.11 depicts the state machine flowchart.

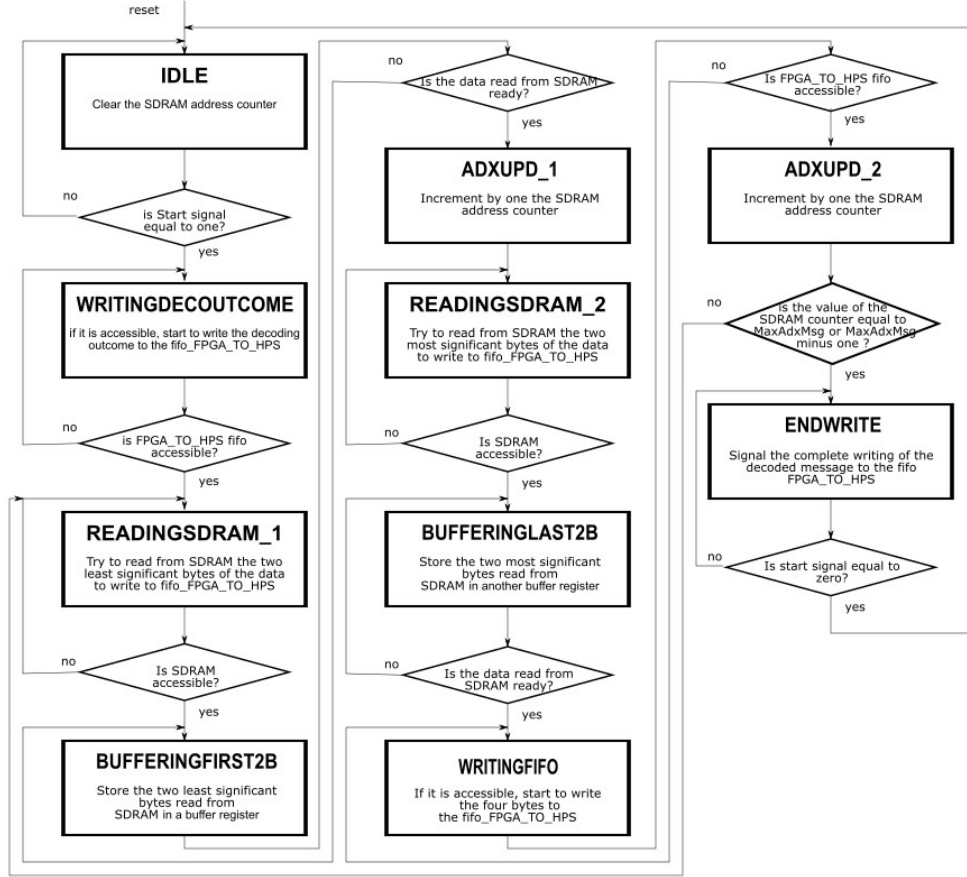


Figure 4.11: WritingDecodedMsg control unit flowchart

The idle state zeroes the counter `MsgAddress`. The writing process starts if the start signal is equal to '1'.

`WritingDecOutcome` is the state during the which a zero or one is written in fifo, according to the whole decoding process outcome. If the decoding ended successfully, the unit writes "00000000000000000000000000000001", otherwise "00000000000000000000000000000000". If the fifo is busy, the state machine waits in this state until the fifo becomes accessible again. A bit in the control status register called `FPGA_TO_HPS_IN_CSR_READDATA_0` tells if writing is possible or not.

In `ReadingSDRAM_1` state, the control unit reads from SDRAM the two least significant bytes of the data to send to fifo. In `BufferingFirst2B` state, these bytes are stored in the buffer register called `DataToWrOnFIFO_register_1`.

In `ReadingSDRAM_2` state, the control unit reads from SDRAM the two most significant bytes of the data to send to fifo. In `BufferingLast2B` state, these bytes are stored in the buffer register called `DataToWrOnFIFO_register_2`.

The sdram address is increased by one in `AdxUpd_1` and `AdxUpd_2` states.

The writing of the decoded message in the fifo ends when the SDRAM address counter reaches the message last SDRAM address. Since the size of the packets written in fifo is equal to 32 bits and the SDRAM has a width of 16 bits, an even number of SDRAM rows is always read. The size of the message in 16-bit packets is called MaxAdxMsg. If MaxAdxMsg is odd, the last SDRAM address read by the control unit will be MaxAdxMsg, otherwise MaxAdxMsg minus one.

The state machine reaches the EndWrite state when the writing process is finished. In this state, the done signal is set to '1'. The control unit returns to the IDLE state only if the start signal returns to '0'.

For debug purposes, the SDRAM address of the first bytes to write in fifo can be changed. In this way, the WritingDecodedMsg unit could be employed to print the SDRAM content at any range of addresses.

4.9.2 | WritingDecodedMsg_DP

The datapath of the WritingSDRAM unit is made up of the following components:

- *DataToWrOnFIFO_register_1*, the 16-bit register which stores the two least significant bytes of the data to write in fifo;
- *DataToWrOnFIFO_register_2*, the 16-bit register which stores the two most significant bytes of the data to write in fifo;
- *MsgAdress*, the counter whose output is the SDRAM address of the two least or most significant bytes of the data to write in fifo.

Figure 4.12 depicts the datapath schematic.

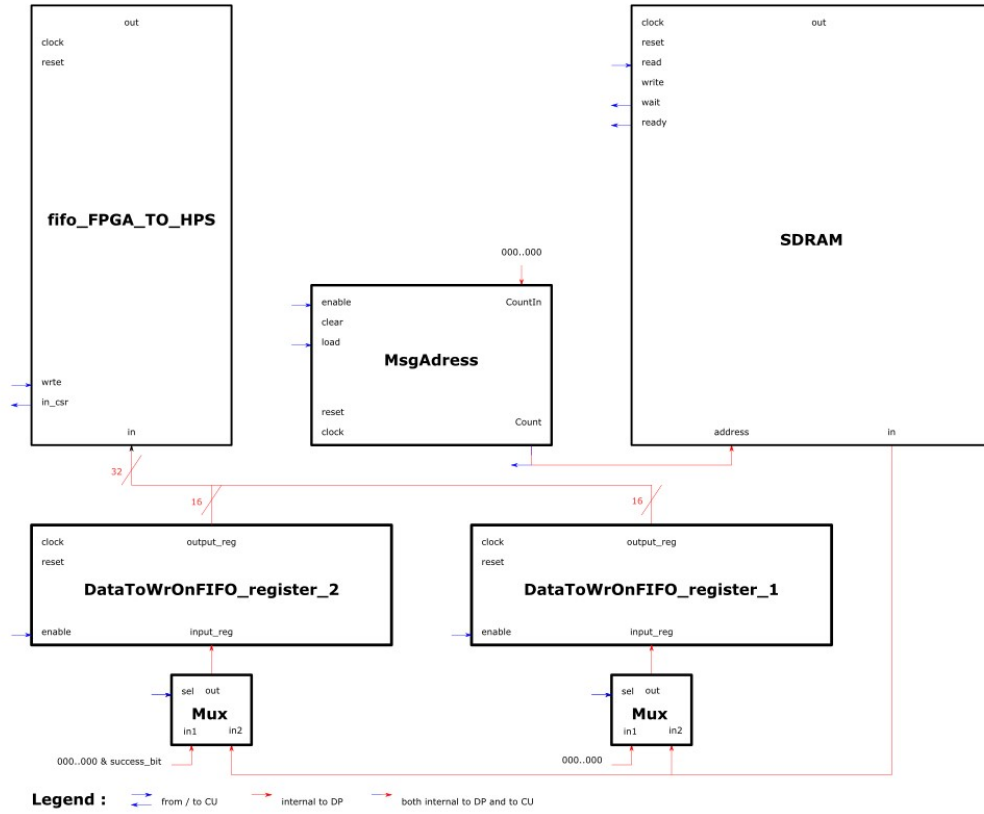


Figure 4.12: WritingDecodedMsg datapath schematic

4.10 | Reset signals management

All the DECODER datapath components have the same asynchronous reset signal, mapped to the KEY(0) button.

4.11 | SDRAM memory mapping

Figure 4.13 clarifies how information are stored in SDRAM.

Maintaining the same architecture parameters of Kristjane Koleci's LEDA architecture, the SDRAM is organized as follows:

- the least significant part of the message occupies the first byte of each of the first 1877 rows;
- the most significant part of the message occupies the second and last byte of each of the first 1877 rows;
- the syndrome occupies the first byte of each of the 1877 rows following those of the message;

- each of the private key and transposed private key parts occupies 81 rows of SDRAM and they are stored immediately after the syndrome;
- each correlation part occupies the following $1877 * 8 / 2 = 7508$ rows;
- the two errors positions parts are stored after the correlation most significant part, their sizes changes iteration by iteration.

The sizes and first addresses of each section can be found in the file named *resources.vhd*

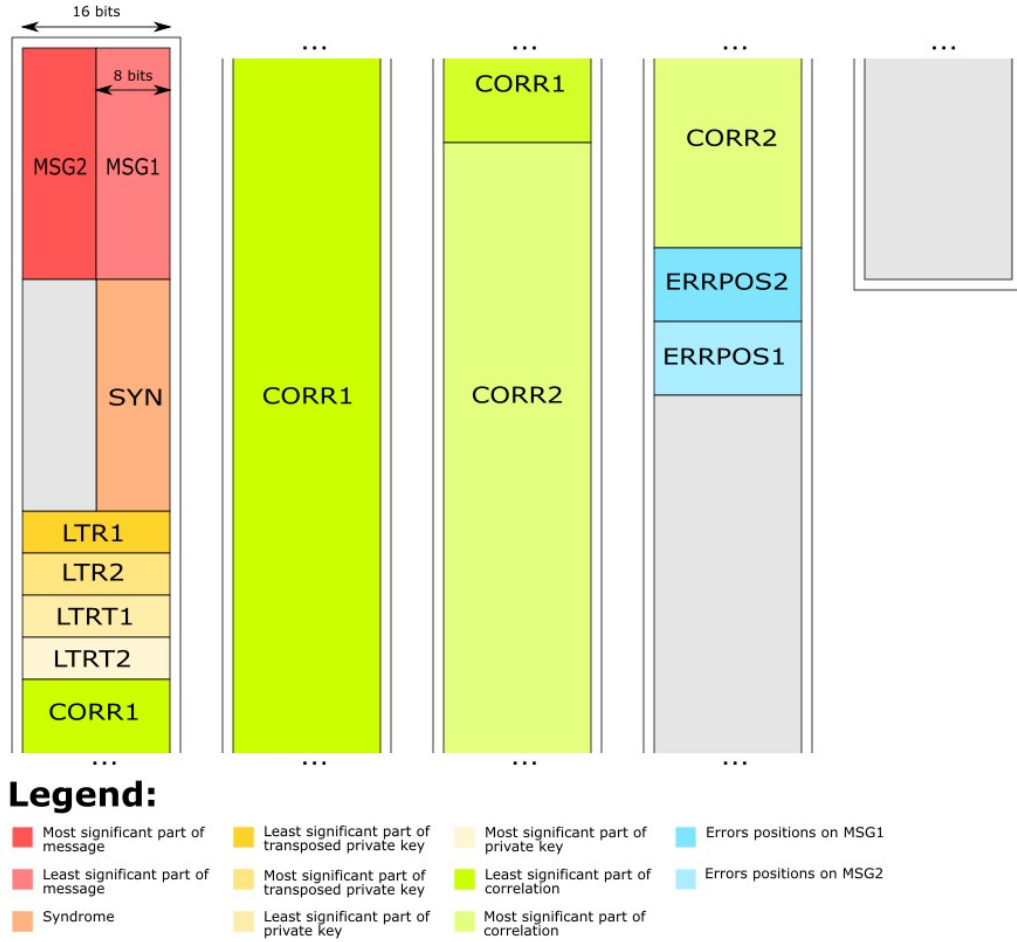


Figure 4.13: SDRAM memory mapping

4.12 | Synthesis and performances results

To find the performances and how many resources the Q-decoder occupies, the Intel Quartus Prime synthesis results are taken into account. Table 4.1 shows the occupied resources.

Resource name	Used	Available	percentage of use
Logic utilization (ALMs)	2,246	32,070	7%
Total pins	197	457	43 %
Total block memory bits	393,216	4,065,280	10 %
Total PLLs	2	6	33 %

Table 4.1: Q-decoder resources occupation

In terms of performances, the bottleneck is represented by the 64 MB SDRAM, which allows a clock signal of frequency equal or less than 143.0 MHz.

Conclusions

The whole system is a successful first attempt to build a working Q-decoder prototype on a DE1-SoC board. Unfortunately, this board does not provide to the FPGA part a direct access to the Ethernet port. Using the Hard Processor System is the right way to set up an Ethernet communication.

Further improvements are possible, since, for example, Avalon Memory Mapped Interface offers several strategies to read from/write to SDRAM. To write or read a large amount of data, the write burst or read burst modes are the fastest

Choosing a different board could provide to the FPGA fabric part a direct access to the Ethernet communication. In this way, it would no longer be necessary to use a hard processor system and socket programming.

Bibliography

- [1] Daniel J. Bernstein, Tanja Lange, "*Post-quantum cryptography*", NATURE 549, 188–194 (2017), September 14, 2017
- [2] F. C.G.P.P. S. M. Baldi A. Barengni, "*Low-Density Parity-Check Code-based publickey cryptosystems*", p. 58, 2017.[Online]. Available: https://www.ledacrypt.org/documents/LEDAcrypt_spec_latest.pdf
- [3] Kristjane Koleci, "*VLSI QC-LDPC Decoder for Post-Quantum Cryptography*", Politecnico di Torino, 2019
- [4] Altera Corporation, "*Qsys System Design Tutorial*", May 5, 2015 [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/tt/tt_qsys_intro.pdf
- [5] Intel, "*Avalon Interface Specification*", January 16, 2019 [Online]. Available: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an-cv-av-soc-ddg.pdf>
- [6] [Online]. Available: http://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/cv_5_HPS_tech_ref.pdf
- [7] Intel, "*Avalon Interface Specification*", January 3, 2020 [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf
- [8] Intel, "*Embedded Peripherals IP User Guide*", January 22, 2020 [Online]. Available: <https://www.intel.com/content/www/us/en/programmable/documentation/sfo1400787952932.html#iga1401394825911>
- [9] [Online]. Available: https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/HPS_peripherals/FPGA_addr_index.html, March 14, 2020
- [10] Altera, "*Cyclone V Hard Processor System Technical Reference Manual*", December 15, 2014 [Online]. Available:

- [http://people.ece.cornell.edu/land/courses/ece5760/
DE1_S0C/cv_5_HPS_tech_ref.pdf](http://people.ece.cornell.edu/land/courses/ece5760/DE1_S0C/cv_5_HPS_tech_ref.pdf)
- [11] [Online]. Available: [http://www.linuxhowtos.org/C_C++/
socket.htm](http://www.linuxhowtos.org/C_C++/socket.htm)
- [12] Altera Corporation - University Program, *Altera_UP_Clocks.pdf*,
May 2013
- [13] Terasic Technologies Inc., *"DE1-SoC User Manual.pdf"*,
January 28, 2019
- [14] Terasic Technologies Inc., *"My_First_HPS.pdf"*,
September 7, 2016
- [15] Terasic Technologies Inc., *"My_First_HPS-Fpga.pdf"*,
September 7, 2016