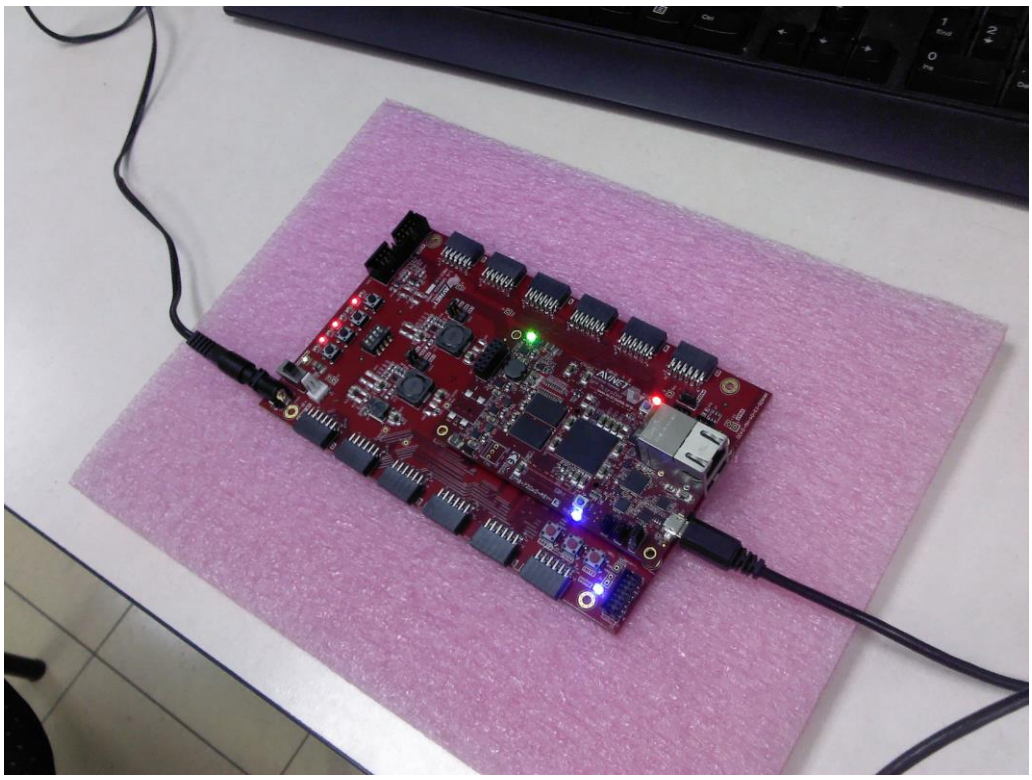
Zynq-7000 based Implementation of the KHAZAD Block Cipher



Xilinx Open Hardware Design Contest 2018

Efraim Wasserman and Yossef Shitzer

Supervisor: Uri Stroh

Jerusalem College of Technology - Lev Academic Center (JCT)

Team number: xohw18-116

YouTube link:

https://youtu.be/-ytT7xl_mq8

1. Introduction

"The KHAZAD Legacy-Level Block Cipher" is an involutional block cipher designed by Paulo S.L.M. Barreto and Vincent Rijmen. It has a substitution-permutation network (SPN) structure, and it uses a 128-bit key, operates on 64-bit data blocks, and comprises 8 rounds. An interesting feature of the algorithm is that the same round function is used for both the key schedule part and for encryption/decryption.
The algorithm has been submitted as a candidate for the first open NESSIE workshop in 2000. This first version now considered obsolete. For phase 2 of NESSIE, a modified version has been submitted, named "Khazad-tweak", and has been accepted as NESSIE finalist [1].
This version can be found here [2].

Due to their many advantages, FPGA devices are being used in many fields of digital signal processing systems, including cryptography, and in particular implementing various block cipher algorithms [3]. As in many cases, one of the designers' main challenges is to carefully maintain balance between Timing considerations and Area considerations. This implementation demonstrates a fair combination of these two factors.

Block ciphers uses what it is called "mode of operation" [4]. This project implements the basic Electronic Codebook (ECB) mode and the more complex Cipher Block Chaining (CBC) mode. The project contains two primary IP's, which represent two "stages" of the design: IP KHAZAD implements the basic ECB mode, and IP enveloped_KHAZAD implements the CBC mode.

This hardware implementation uses the MicroZed 7010 development board by Avnet Inc., which is based on a Xilinx Zynq-7010 All Programmable SoC. The Zynq Z-7010 device integrates a dual-core ARM Cortex A9 processor (PS, processing system) with an Artix-7 FPGA (PL, programmable logic). This new concept allows many interesting and exciting possibilities, and enables us to

design an all-at-one-device complete cryptographic system, from user-input to results-output.

As it reasonable to do, the PL in our design is used mainly for implementing the algorithm, and the PS is used mainly for dealing with user input & output operations.

The project had several motives:

1. To get familiar with different aspects of FPGA design.
2. To get familiar with the unique Zynq devices architecture.
3. To develop a full practical system, which can be used for safe and secure modern communication.

2. Design

Following is a brief description of the design. Additional details and explanation can be found in the corresponding source files.

Platform: The MicroZed 7010 development board. This board can be used as both a stand-alone board, or combined with an I/O carrier card as an embeddable system-on-module.
This implementation was designed to be fully operational even when using the stand alone mode. Plugging the board into the carrier card will activate more indicator LEDs.

2.1 Hardware

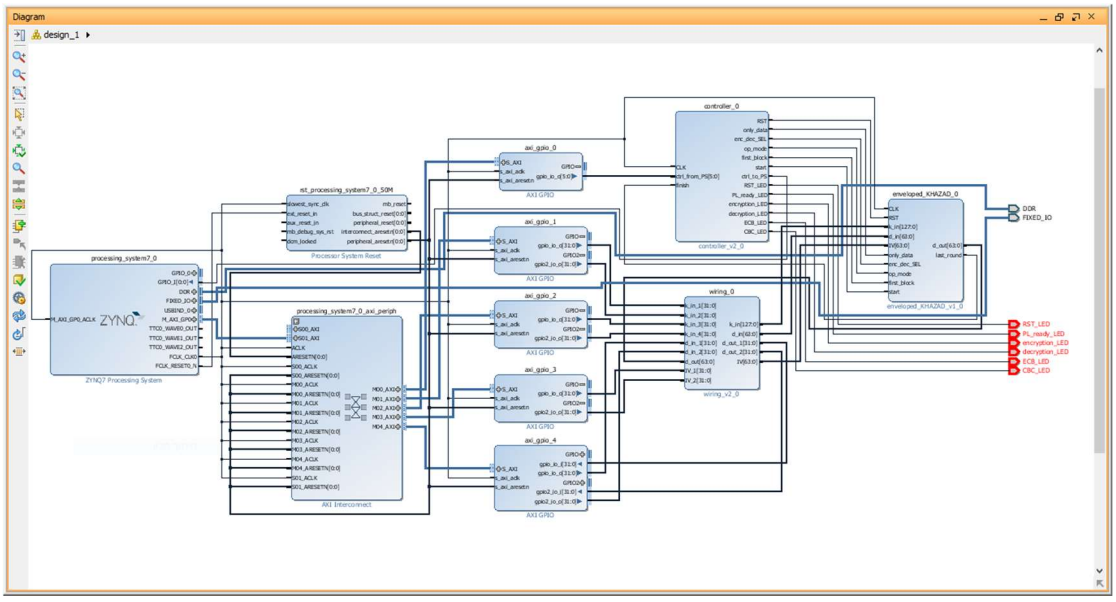


Figure 1. The project Block Design

The source files were written in Verilog. Their names, structure and logic tries to resemble the original algorithm description, as appears in the developers' article [2].

The PL design is fully synchronous, using the PS FCLK_CLK0.

The design's input and output operations are being done with the help of the Zynq **PS** part, the 32-bit ARM Cortex A9 (represented in the Vivado block design by **ZYNQ7 Processing System** block). The user input and the design's control signals are transferred from the PS to the **PL** part via AXI4 interface. To ensure simplicity and ease-of-use, **AXI4-Lite** was used (as recommended by Xilinx in this video [5]), with Xilinx LogiCORE IP **AXI_GPIO** modules, and each AXI_GPIO is communicating with a single fixed register.
These data and control signals enable the PL to execute the desired calculations, until a result is ready. In the PL part, encryption/decryption operations are calculated using parallel computing of the entire 64-bits data, for fast and simple operation, and as countermeasure against side-channel attacks [3].
A PL output flag indicating the result is ready is transferred to the PS via the Zynq's **EMIO** interface. Then, the result itself is transferred using AXI_GPIO again.

As described in later section, the design was able to fit into the Zynq device as is. Therefore, no BRAM blocks were used, to avoid unnecessary constrains during place & route and to ensure maximum portability.

Following a bottom-to-top approach, the design's modules can be arranged in this order:

1. Module **P_mini_box**, a 4-bit to 4-bit substitution box.

2. Module **Q_mini_box**, a 4-bit to 4-bit substitution box.

3. Three **P_mini_box**es instantiations and three **Q_mini_box**es instantiations form module **S_box**, an 8-bit to 8-bit substitution box.

4. Eight **S_box**es form module **gamma**, 64-bit to 64-bit nonlinear transformation layer.

5. Module **poly_mult**, modular polynomial multiplication over GF(2^8) with the Khazad algorithm reduction polynomial.

6. Eight **poly_mult**s form module **row_mult**, polynomial multiplication of 8-bit data with a 32-bit vector represents a row of the Khazad matrix H.

7. Eight **row_mult**s form module **theta**, 64-bit to 64-bit linear diffusion transformation layer.

8. Modules **gamma** and **theta**, along with **XOR** gates, form module **round_function_plus**, all possible versions of the Khazad round function, used for key scheduling and for encryption/decryption.

9. Module **round_function_plus**, with additional logic and an FSM implementation, form the **IP module KHAZAD**, which implements the complete Khazad algorithm, and is sufficient to implement the ECB mode. A full run of the algorithm takes 24 clock-cycles: 9 cycles to calculate nine 64-bit round keys, 7 cycles to calculate seven 64-bit inverse decryption keys, and 8 cycles to perform 8-rounds encryption/decryption operation. If the different keys were already calculated in previous runs, and the same key is used again, the only_data option allows to use the pre-calculated keys so the operation only takes 8 cycles, a fact that have significant influence on the design throughput. The implementation is using the multi-cycle iterative implementation architecture, not loop-unrolling architecture and not pipeline architecture in order to save FPGA area and to allow non-feedback block cipher modes of operation like CBC. (A test bench module for **KHAZAD** source code simulation using the developers' test vectors file can be found in the archive tests directory.)

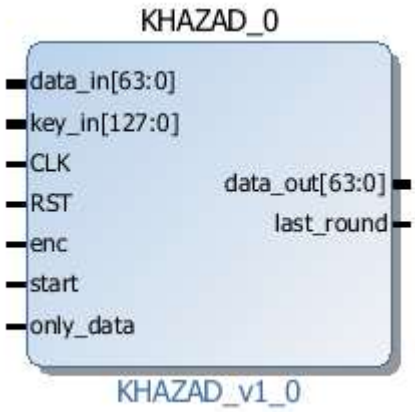


Figure 2. The KHAZAD IP

10-11. Modules **op_mode_enc** and **op_mode_dec** implement the pre-KHAZAD and post-KHAZAD calculations need to be done for CBC mode.

12. Module **CBC_dec_memory** save the previous cipher input necessary for CBC decryption.

13. Module **KHAZAD** together with **op_mode_enc**, **op_mode_dec** and **CBC_dec_memory** form **IP module enveloped KHAZAD**, which wraps KHAZAD with the above modules to fully implement the CBC mode.

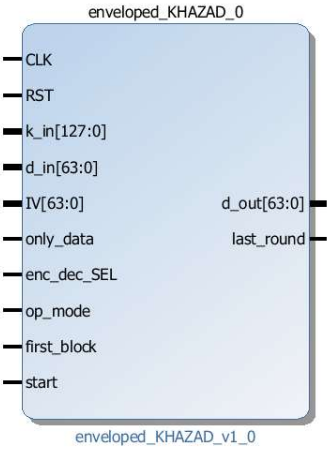


Figure 3. The enveloped_KHAZAD IP

14. **IP module controller** manages control signals between the PS program and the PL design, and some indicator LEDs output.

15. **IP module wiring**, an intermediate between the PS and the PL that concatenates and splits data vectors.

16. Xilinx IP **AXI_GPIO #0**, single channel, transfers the control signal ctrl_from_PS[5:0] from the PS to module **controller**.

17-19. Xilinx IP **AXI_GPIO #1**, **#2**, **#3**, dual channel, transfer the key and the CBC initialization vector (IV) from the PS to module **wiring**.

20. Xilinx IP **AXI_GPIO #4**, dual channel, works bidirectionally, and transfers 64-bit data_in from the PS to **wiring**, and 64-bit data_out from **wiring** to PS.

In addition, a **Processing System Reset** and **AXI Interconnect** are being added and configured automatically by Vivado.
External ports control some **user LEDs** in the MicroZed carrier card, as configured in an XDC file.

2.2 Software

The software uses many C standard libraries and Xilinx libraries. In addition, the source files included in this project are:

khazad-tweak32.h: A header file, identical to the khazad-tweak32.c file that can be found at [2]. This is a reference code by developers Paulo S.L.M. Barreto and Vincent Rijmen. We used this code in the test_vectors functions (see below) to compare the results of the hardware implementation to the reference code software implementation.

nessie_modified.h: A slightly modified version of the original nessie.h reference code header file that can be also found at [2]. Used mainly for data type and globals definitions.

main.c: This is our main program file. Mainly used to show the user an options menu, which calls functions defined in the next file.

KHAZAD_Zynq.h: This is our primary header file, written by the team, containing the functions and definitions to run the various applications of the design. The file defines several functions:

1. **board_configuration**: configures the PS, the peripherals, the bidirectional AXI_GPIO module and the interrupt handling.

2. **Zynq_crypt**: The primary function that communicates with the PL fabric and get it to execute a "crypt" operation: encryption or decryption. The function gets the input entered by the user (key, text, CBC Initialization Vector), and necessary boolean flags for the exact type of operation needed. (For further details and explanation regarding the function parameters see the source file comments.)
The function makes use of the static variable ctrl and configure it to give the appropriate instructions to the design. It sends the relevant data and the ctrl value via the AXI interface: The data goes to the wiring module, and the ctrl signal to the controller module.
An interesting feature is the start/finish system. For maximum portability (see below), the system works in that way: The LSB of ctrl is used to issue a start command, if it's different from a corresponding certain bit in the PL (comparison made by simple XOR). When an operation has ended, the PL bit is made equal to the PS ctrl bit, so the next start command is issued by toggling this PS bit, and so on.
After the function is notified that the PL operation has ended (by polling and making the bits comparison we have just described), the result is read through AXI.

3. **Zynq_crypt_simple**: Simplified version of the above function, with limited options.

4-5. **test_vectors_full, test_vectors_short**: full and short versions of tests used to compare the hardware implementation ECB results to reference code results. It was designed to imitate the original test vectors program, so most of the line codes were taken the original reference code bctestvectors.c file, which can be also found at [2]. It calls the Zynq_crypt_simple function.

6. **demonstration**: Designed to demonstrate the hardware implementation correctness in a simple friendly way. It prompts the user for key and for plaintext string data, then processes the data block-by-block. Each block is encrypted and then decrypted again, by calling the Zynq_crypt function. For each block the plaintext, the ciphertext, the decrypted text (all three in hexadecimal ASCII code) and the block number are printed on screen. Comparisons are made between the plaintext and decrypted text of each block, and between the original characters string and a decrypted re-calculated string. If a comparison fails, an error massage appears. Since the function works block-by-block and switches between encryption and decryption, only ECB mode can be used for this demonstration.

7. **application**: A practical application. The user can choose between encryption and decryption, then he's prompted for a key, for operation mode and for initialization vector (if CBC mode was chosen). For encryption, a plaintext string is entered, and the ciphertext is calculated and printed on screen as hexadecimal figures. For decryption, a ciphertext hexadecimal figures string is entered, and the plaintext is calculated and printed on screen as regular characters. It calls the Zynq_crypt function.

8. **USR_button_ISR**: The user button Interrupt Service Routine. The function communicates with the PL fabric, and configure the ctrl variable to give a reset

command to the design. The user can use this for resetting the PL at any time, for instance, in order to erase previous data and key.

9,10: **print_data, compare_blocks**: Auxiliary functions, taken from the bctestvectors.c file mentioned above.

2.3 Design Reuse

The PS program is endianness-neutral. It can be executed on any 32-bit or 64-bit processor with a C compiler. Many of the functions was written in a way that they can be reused as standalone programs.
The PL algorithm implementation doesn't depend on any special feature of the Zynq Artix-7 FPGA architecture, so it can be implemented in any FPGA device, given sufficient area and sufficient number of I/O's. Alternatively, one can use other component to handle the I/O, like the PS in the Zynq devices (as done in our design).

As said above, the project contains two primary IP's. If one wants to reuse the design for ECB only, or to wrap it differently to implement different block cipher modes of operation, one can use module KHAZAD. In the source files directory one can find all the Version 1.0 files that put together the ECB-only implementation.

To ensure maximum portability, no assumptions were made regarding the design's inner data transfer rates and AXI frequency, nor the design is relying on this implementation de-facto PL frequency, CPU frequency or AXI frequency. Signals and commands that are meant to reach their destination sequentially one-after-the-other, e.g. the key, data and start command, are being sent one-after-the-other, using the same interface (i.e., the AXI4-Lite interface. The EMIO interface is only used in the reversed direction). As described above, a special bistable flags system was implemented, to ensure portability and reuse while achieving three goals:
1. No overlapping of one start-command with the next one.
2. No missing the PL-finish-flag.

3. No overlapping of one PL-finish-flag raising with the next one.

All of the design's source files, IPs files and project files have been made available for reuse (Open Source), and can be found here:
https://github.com/efwasser/KHZAD_Zynq

3. Results

The main challenge while implementing the design was building accurate and robust modules, which implement the algorithm without wasting area, and without wasting any clock cycles within the main algorithm module KHAZAD or while transferring signals between KHAZAD and other modules.

Some of the results we got:

Module KHAZAD implements the complete algorithm using only one instantiation of the round_function_plus module. As mentioned previously, a full operation takes 24 clock-cycles, and an operation using pre-calculated keys takes 8 cycles. No dead cycles exists within this module, or while working with the controller module.

We used Vivado 2015.4 to implement this module on an Artix-7 FPGA device with sufficient number of I/O's (XC7A75TFGG484-1). The achieved clock speed is 123 MHz. This reflects a throughput of 328 Mbps when calculating new key schedule for each data block (unlikely scenario), and throughput of 985.22 Mbps when using pre-calculated keys.
The implementation utilizes 1356 Slice LUTs and 1287 Slice registers. The number of slices used is 447, so we can calculate a Throughput/Area ratio (T/A) of 2.20 Mbps/slices.

After that we used Vivado to implement the entire PL design on our Zynq device. The achieved clock speed is 111.11 MHz, i.e. 9 ns clock-cycles. For the PL part, this can reflect a theoretical throughput of 296 Mbps when calculating keys for each data block, and throughput of 888.89 Mbps when using pre-calculated keys. It utilizes 2803 Slice LUTs and 3630 Slice registers. The number of slices used is 1122, so we can calculate a Throughput/Area ratio (T/A) of 0.79 Mbps/slices.

(These results reflect the Vivado project submitted in this archive. Later we tried Vivado 2016.4 for synthesis and implementation, and achieved better utilization result of 2673 LUTs and 3622 registers. Later versions of Vivado may produce even better result.)

The actual throughput of the entire design depends heavily on the PS part, the different functions used, and the number of AXI transactions performed. For instance, each iterative of the loop in the test_vectors functions (see KHAZAD_Zynq.h source file), which includes many CPU operations and nine AXI transactions, has a latency of 3.34 μs.

4. Conclusion

As stated in NESSIE Security Report [6]: "A primary selling-point of Khazad is that all components of the algorithm use involutions. This involutional structure is important for implementations and to make encryption and decryption equivalent operations apart from the order of the key schedule." We have succeed in making use of that fact to implement the algorithm in a simple elegant way, and create a relatively-low-area design that can fit into many FPGA devices, and enable use of modern secure communication even in limited resources environments. That was done while "keeping an eye" on the design timing and achieving a fair clock speed.

Our results for implementing the algorithm module show significant improvement over previously reported results. In [7], the Khazad algorithm was implemented in two ways: iterative implementation architecture, and pipeline architecture, targeting a Xilinx VIRTEX-E FPGA device. The reported results are:

Architecture	Area (CLBs)	Frequency (MHz)	Throughput (Mbps)
Iterative	2250	65	462
Pipeline	9277	70	4480

Table 1. Some previously reported results for implementing Khazad

For the iterative architecture approach (which is identical to ours), we show significantly higher frequency, significantly higher throughput, and better T/A ratio (even after generous conversion calculations).

The reported pipeline architecture achieved much higher throughput (as expected), but only at the cost of using a lot more FPGA area, and the notable inability to implement non-feedback block cipher modes of operation like CBC, CFB, etc. In addition, we still show significantly higher frequency and better T/A ratio.

We have managed to build the design as a "two-stages-project": Stage 1 implements only the basic algorithm, i.e. ECB mode, and stage 2 adds more hardware modules to implement the more complex CBC mode. This modularity is important for clear comprehensible design, and for IPs reusability as described earlier.

As it turns out, the special Zynq-7 architecture is ideal for implementing block ciphers, for several reasons:

1. Block ciphers require a great number of I/O's, which can dictate using a "large" FPGA with high pin count. The presence of a processor near the PL suggests an alternative way.
2. As one can see in the C source files, encryption and decryption operations may require a lot of input & output data manipulation, e.g. from characters and ASCII code to hexadecimal figures, and vice versa. The flexibility of the PS is very helpful in those situations.
3. In cryptographic and communication systems, security is the leading factor. The unique combination of a PS and an FPGA together on the same device, allows more secured development, and of course – more secured communication.

Future work may include:

Implementing additional or alternative block ciphers modes of operation.

Integrating the entire design into larger communication system, or using the basic KHAZAD IP by itself with necessary modifications (e.g. replacing the AXI4-Lite interface with other one, using RAM, etc.).

5. References

[1] See:
https://web.archive.org/web/20171011071731/http://www.larc.usp.br/~pbarreto/KhazadPage.html

[2] P. Barreto and V. Rijmen, "The Khazad Legacy-Level Block Cipher", In First Open NESSIE Workshop, KU-Leuven, 2000. Submission to NESSIE. Available at:
https://www.cosic.esat.kuleuven.be/nessie/tweaks.html

[3] F. X. Standaert, "Secure and efficient symmetric encryption using FPGAs", in Cryptographic Engineering, Springer, 2009, Chapter 11, pp 295-320.

[4] For quick basic explanation see:
https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

[5]
https://www.xilinx.com/video/hardware/how-to-use-the-3-axi-configurations.html

[6] NESSIE Security Report D20-v2, p 37. Available at:
http://www.cosic.esat.kuleuven.be/nessie/deliverables/D20-v2.pdf

[7] P. Kitsos, N. Sklavos, M.D. Galanis and O. Koufopavlou, "64-bit Block ciphers: hardware implementations and comparison analysis", in Computers and Electrical Engineering 30 (2004), Elsevier, pp 593–604. Available at:
https://www.sciencedirect.com/science/article/pii/S0045790605000108

Appendix

Full source code for user generated hardware/software and constraints

1. Final verilog files

2. Version 1.0 verilog files

3. Final C files

4. Version 1.0 C files

5. Final constraints file (XDC)

6. Version 1.0 constraints file (XDC)

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
This module implements the KHAZAD algorithm (Khazad-tweak version) mini-substitution-box "P
mini box",
using logic functions (not memory), as described in:
P. Barreto and V. Rijmen, The Khazad Legacy-Level Block Cipher, appendix B
available at:
https://www.cosic.esat.kuleuven.be/nessie/tweaks.html
**********************************************************************************************
***********
**********************************************************************************************
***********/
module P_mini_box  // 4-bit to 4-bit substitution box
(
  input  [3:0] data_in,
  output [3:0] data_out
);

wire t0, t1, t1_2, t1_3, t1_4, t2, t2_2, t2_3, t2_4, t3, t3_2, t4, t4_2, t4_3;

assign t0 = data_in[0] ^ data_in[1];
assign t1 = data_in[0] ^ data_in[3];
assign t2 = data_in[2] & t1;
assign t3 = data_in[3] & t1;
assign t4 = t0 | t3;
assign data_out[3] = t2 ^ t4;
assign t1_2 = ~t1;
assign t2_2 = data_in[1] & data_in[2];
assign t4_2 = data_in[3] | data_out[3];
assign t1_3 = t1_2 ^ t2_2;
assign data_out[0] = t4_2 ^ t1_3;
assign t4_3 = data_in[2] & t1_3;
assign t2_3 = t2_2 ^ data_in[3];
assign t2_4 = t2_3 | t4_3;
assign data_out[2] = t0 ^ t2_4;
assign t3_2 = t3 ^ t4_3;
assign t1_4 = t1_3 | data_out[3];
assign data_out[1] = t3_2 ^ t1_4;

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
This module implements the KHAZAD algorithm (Khazad-tweak version) mini-substitution-box "Q
mini box",
using logic functions (not memory), as described in:
P. Barreto and V. Rijmen, The Khazad Legacy-Level Block Cipher, appendix B
available at:
https://www.cosic.esat.kuleuven.be/nessie/tweaks.html
**********************************************************************************************
***********
**********************************************************************************************
***********/
module Q_mini_box  // 4-bit to 4-bit substitution box
(
  input  [3:0] data_in,
  output [3:0] data_out
);

wire t0, t0_2, t0_3, t0_4, t1, t1_2, t1_3, t1_4, t1_5, t2, t3, t3_2, t3_3, t4;

assign t0 = ~data_in[0];
assign t1 = data_in[1] ^ data_in[2];
assign t2 = data_in[2] & t0;
assign t3 = data_in[3] ^ t2;
assign t4 = t1 & t3;
assign data_out[0] = t0 ^ t4;
assign t0_2 = data_in[0] ^ data_in[1];
assign t1_2 = t1 ^ t2;
assign t0_3 = t0_2 ^ t3;
assign t1_3 = t1_2 | t0_3;
assign data_out[2] = data_in[2] ^ t1_3;
assign t1_4 = t1_3 & data_in[0];
assign t3_2 = data_in[3] & t0_3;
assign t3_3 = t3_2 ^ t2;
assign data_out[1] = t1_4 ^ t3_3;
assign t1_5 = data_in[2] | data_out[0];
assign t0_4 = t0_3 ^ t3_3;
assign data_out[3] = t1_5 ^ t0_4;

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
This module implements the KHAZAD algorithm (Khazad-tweak version) substitution-box "S box",
using instantiations of "P" and "Q" mini-boxes, as described in:
P. Barreto and V. Rijmen, The Khazad Legacy-Level Block Cipher, section 6.2
available at:
https://www.cosic.esat.kuleuven.be/nessie/tweaks.html
**********************************************************************************************
***********
**********************************************************************************************
***********/
module S_box  // 8-bit to 8-bit substitution box
(
  input  [7:0] data_in,
  output [7:0] data_out
);

wire [3:0] p1, q1, p2, q2, p3, q3;

P_mini_box  P_mini_box_1  (data_in[7:4], p1);
Q_mini_box  Q_mini_box_1  (data_in[3:0], q1);
P_mini_box  P_mini_box_2  ({p1[1:0], q1[1:0]}, p2);
Q_mini_box  Q_mini_box_2  ({p1[3:2], q1[3:2]}, q2);
P_mini_box  P_mini_box_3  ({q2[3:2], p2[3:2]}, p3);
Q_mini_box  Q_mini_box_3  ({q2[1:0], p2[1:0]}, q3);

assign data_out = {p3, q3};

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
This module implements the KHAZAD algorithm (Khazad-tweak version) nonlinear layer "gamma",
using 8 instantiations of the "S" substitution box.
The 64-bit data_in (=state) is separated into eight 8-bit parts, and each one is going into
an S-box, as described in:
P. Barreto and V. Rijmen, The Khazad Legacy-Level Block Cipher, section 3.1
available at:
https://www.cosic.esat.kuleuven.be/nessie/tweaks.html
**********************************************************************************************
***********
**********************************************************************************************
***********/
module gamma  // 64-bit to 64-bit transformation
(
  input  [63:0] data_in,
  output [63:0] data_out
);

S_box   S0  (data_in[63:56], data_out[63:56]);
S_box   S1  (data_in[55:48], data_out[55:48]);
S_box   S2  (data_in[47:40], data_out[47:40]);
S_box   S3  (data_in[39:32], data_out[39:32]);
S_box   S4  (data_in[31:24], data_out[31:24]);
S_box   S5  (data_in[23:16], data_out[23:16]);
S_box   S6  (data_in[15: 8], data_out[15: 8]);
S_box   S7  (data_in[ 7: 0], data_out[ 7: 0]);

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
This module implements modular polynomial multiplication over GF(2^8),
with the KHAZAD reduction polynomial: x^8 + x^4 + x^3 + x^2 + 1 = 100011101 = 00011101
(upper bit exceeds limits and omitted).
The multiplications are calculated based upon multiplications by 2, 4 and 8.
Each additional multiplication by 2 is a one-place left shift, with conditional XOR if MSB =
1.
This condition is implemented by logical AND with "and_mask", which is replications of the
MSB.
**********************************************************************************************
***********
**********************************************************************************************
***********/
module poly_mult  // 8-bit to 8-bit transformation
(
  input      [7:0] data_in,
  input      [3:0] multiplier,
  output reg [7:0] data_out
);

wire [7:0] and_mask   = {8{data_in[7]}};                                    // mask for
multiplication by 2
wire [7:0] mult2      = ({data_in[6:0],1'h0}) ^ ((8'b00011101) & and_mask);  //
multiplication by 2
wire [7:0] and_mask_2 = {8{mult2[7]}};                                      // mask for
multiplication by 4
wire [7:0] mult4      = ({mult2[6:0],1'h0}) ^ ((8'b00011101) & and_mask_2);  //
multiplication by 4
wire [7:0] and_mask_3 = {8{mult4[7]}};                                      // mask for
multiplication by 8
wire [7:0] mult8      = ({mult4[6:0],1'h0}) ^ ((8'b00011101) & and_mask_3);  //
multiplication by 8

always @(*)
case (multiplier)  // only the cases possible in KHAZAD
    4'h1: data_out = data_in                    ;
    4'h3: data_out = mult2 ^ data_in            ;  // 3=2+1
    4'h4: data_out = mult4                      ;
    4'h5: data_out = mult4 ^ data_in            ;  // 5=4+1
    4'h6: data_out = mult4 ^ mult2              ;  // 6=4+2
    4'h7: data_out = mult4 ^ mult2 ^ data_in    ;  // 7=4+2+1
    4'h8: data_out = mult8                      ;
    4'hb: data_out = mult8 ^ mult2 ^ data_in    ;  // b=11=8+2+1
    default: data_out = 8'h0                    ;
endcase

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
This module implements polynomial multiplication of 8-bit data_in with a 32-bit vector
represents a row of the KHAZAD matrix H,
using 8 instantiations of "poly_mult".
The 32-bit row_coefficients vector is separated into eight 4-bit parts, each part represents
one coefficient of the matrix.
The result of each poly_mult instantiation is 8-bit wide,
and the final result of multiplication by the entire row is given by the 64-bit data_out
output vector.
**********************************************************************************************
***********
**********************************************************************************************
***********/
module row_mult  // 8-bit to 64-bit transformation
(
  input   [7:0]  data_in,
  input   [31:0] row_coefficients,
  output  [63:0] data_out
);

poly_mult  M0  (data_in, row_coefficients[31:28], data_out[63:56]);
poly_mult  M1  (data_in, row_coefficients[27:24], data_out[55:48]);
poly_mult  M2  (data_in, row_coefficients[23:20], data_out[47:40]);
poly_mult  M3  (data_in, row_coefficients[19:16], data_out[39:32]);
poly_mult  M4  (data_in, row_coefficients[15:12], data_out[31:24]);
poly_mult  M5  (data_in, row_coefficients[11: 8], data_out[23:16]);
poly_mult  M6  (data_in, row_coefficients[7 : 4], data_out[15 :8]);
poly_mult  M7  (data_in, row_coefficients[3 : 0], data_out[7 : 0]);

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
This module implements the KHAZAD algorithm linear diffusion layer "theta", using 8
instantiations of "row_mult".
The 64-bit data_in (=state) is separated into eight 8-bit parts, and each one is multiplied
by a 32-bit vector,
which represents a row of the KHAZAD matrix H (zeros not included), to yield a 64-bit result.
The final output data_out is the result of XOR between all the 8 results of row_mult,
similar to the method described in:
P. Barreto and V. Rijmen, The Khazad Legacy-Level Block Cipher, section 7.1
available at:
https://www.cosic.esat.kuleuven.be/nessie/tweaks.html
**********************************************************************************************
***********
**********************************************************************************************
***********/
module theta  // 64-bit to 64-bit transformation
(
  input  [63:0] data_in,
  output [63:0] data_out
);

wire [63:0] T0, T1, T2, T3, T4, T5, T6, T7;

row_mult  R0  (data_in[63:56], 32'h134568b7, T0);
row_mult  R1  (data_in[55:48], 32'h3154867b, T1);
row_mult  R2  (data_in[47:40], 32'h4513b768, T2);
row_mult  R3  (data_in[39:32], 32'h54317b86, T3);
row_mult  R4  (data_in[31:24], 32'h68b71345, T4);
row_mult  R5  (data_in[23:16], 32'h867b3154, T5);
row_mult  R6  (data_in[15: 8], 32'hb7684513, T6);
row_mult  R7  (data_in[ 7: 0], 32'h7b865431, T7);

assign data_out = T0 ^ T1 ^ T2 ^ T3 ^ T4 ^ T5 ^ T6 ^ T7;

endmodule
```

```
/*************************************************************************************************
***********
*************************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
*************************************************************************************************
***********
*************************************************************************************************
***********
This module implements the KHAZAD round function "rho", using instantiations of previous
modules.
The complete round function, used in most rounds of the cipher and in the key schedule, is
composed of the "gamma" function,
then "theta", then XOR with round_key (which is an actual round key for the cipher, or some
round constant for the key schedule).
In the last round of the cipher, only gamma and XOR are used.
Similarly, to calculate the "inverse" key rounds, required for decryption, only the theta
function is used, by turning on the "only_theta_req" signal.
So the module is used not only for the complete round function but also for the last round
function and for theta only.
The word "plus" in the module's name refers to that fact.
The above operations are all described in:
P. Barreto and V. Rijmen, The Khazad Legacy-Level Block Cipher, section 3
available at:
https://www.cosic.esat.kuleuven.be/nessie/tweaks.html
*************************************************************************************************
***********
*************************************************************************************************
***********/
module round_function_plus
(
  input  [63:0] data_in,
  input  [63:0] round_key,
  input  only_theta_req,
  output [63:0] theta_out,
  output [63:0] last_round_out,
  output [63:0] rho_out
);

wire [63:0] gamma_out, theta_in;

gamma  G  (data_in, gamma_out);

assign theta_in = (only_theta_req == 1) ? data_in : gamma_out;

theta  T  (theta_in, theta_out);               // theta_out is the output for the inverse
key rounds

assign last_round_out = gamma_out ^ round_key;  // last_round_out is the output for the
cipher's last round

assign rho_out = theta_out ^ round_key;         // rho_out is the output for the complete
round function

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
This module implements the complete KHAZAD algorithm, using only the round_function_plus
module and a finite-state machine (FSM).
A full operation takes 24 clock-cycles: 9 cycles to calculate nine 64-bit round keys, 7
cycles to calculate seven 64-bit inverse decryption keys,
and 8 cycles to perform 8-rounds encryption/decryption.
**********************************************************************************************
***********
**********************************************************************************************
***********
Inputs & outputs description:
Input data_in: 64-bit data, plaintext (for encryption) or ciphertext (for decryption).
Input key_in: 128-bit secret key.
Input CLK: clock pulses.
Input RST: design reset signal.
Input enc: the desired operation. enc = 1: encryption, enc = 0: decryption.
Input start: start operation signal. start = 1: begin operation, start = 0: on idle.
Input only_data: indicates the key period. only_data = 1: new data_in, same key. only_data =
0: new data_in and new key_in, need to recalculate round keys.
Output data_out: 64-bit data, ciphertext (for encryption) or plaintext (for decryption).
Output last_round: 1-clock-cycle-width pulse indicates the last round is being calculated.
When the flag turns on, last round has begun. When the flag
turns off, last round has ended, and data_out is valid.
**********************************************************************************************
***********
**********************************************************************************************
***********
The KHAZAD algorithm is described in:
P. Barreto and V. Rijmen, The Khazad Legacy-Level Block Cipher
available at:
https://www.cosic.esat.kuleuven.be/nessie/tweaks.html
**********************************************************************************************
***********
**********************************************************************************************
***********/
module KHAZAD
(
  input  [63:0] data_in,
  input  [127:0] key_in,
  input  CLK, RST, enc, start, only_data,
  output reg [63:0] data_out,
  output reg last_round
);

reg [4: 0] ctrl;  // controls the state of the FSM
reg [63:0] round_keys [0:8];
reg [63:0] inv_round_keys [1:7];
reg [63:0] Kminus2, input_1, input_2; // inputs for round_function_plus
reg only_theta_req;
wire [63:0] only_theta_out, last_round_out, rho_out;

round_function_plus R (input_1, input_2, only_theta_req, only_theta_out, last_round_out,
rho_out);

always @(posedge CLK)
begin

  if (RST)
    begin
        ctrl <= 5'd0;
        data_out <= 64'h0;
```

```
        only_theta_req <= 0;
        last_round <= 0;
    end

else
    case (ctrl)

        5'd0:
            begin
                if (last_round)                     // FSM got here after last round was
                completed
                    data_out <= last_round_out;     // output data_out only when it's valid
                last_round <= 0;                    // turn off the flag
                if (start)
                    if (only_data)                  // no need for key schedule. start
                    encryption/decryption operation to save time
                        begin
                            if (enc)                // encryption
                                begin
                                    input_1 <= data_in ^ round_keys[0];
                                    input_2 <= round_keys[1];
                                end
                            else                    // decryption
                                begin
                                    input_1 <= data_in ^ round_keys[8];
                                    input_2 <= inv_round_keys[7];
                                end
                            ctrl <= 5'd17;
                        end
                    else                            // key schedule needed. start
                    calculating the round keys
                        begin
                            Kminus2 <= key_in[127:64];
                            input_1 <= key_in[63:0];
                            input_2 <= 64'hba542f7453d3d24d;  // round constant
                            ctrl <= 5'd1;
                        end
                else
                    ctrl <= 5'd0;                   // if start==0, do nothing, stay in this
                    state and wait for start signal
            end

        5'd1:   // key schedule continuation
            begin
                round_keys[0] <= rho_out ^ Kminus2;
                Kminus2 <= input_1;
                input_1 <= rho_out ^ Kminus2;
                input_2 <= 64'h50ac8dbf70529a4c;
                ctrl <= 5'd2;
            end

        5'd2:
            begin
                round_keys[1] <= rho_out ^ Kminus2;
                Kminus2 <= input_1;
                input_1 <= rho_out ^ Kminus2;
                input_2 <= 64'head597d133515ba6;
                ctrl <= 5'd3;
            end

        5'd3:
            begin
                round_keys[2] <= rho_out ^ Kminus2;
                Kminus2 <= input_1;
                input_1 <= rho_out ^ Kminus2;
                input_2 <= 64'hde48a899db32b7fc;
                ctrl <= 5'd4;
            end

        5'd4:
            begin
                round_keys[3] <= rho_out ^ Kminus2;
                Kminus2 <= input_1;
```

```
            input_1 <= rho_out ^ Kminus2;
            input_2 <= 64'he39e919be2bb416e;
            ctrl <= 5'd5;
        end

    5'd5:
        begin
            round_keys[4] <= rho_out ^ Kminus2;
            Kminus2 <= input_1;
            input_1 <= rho_out ^ Kminus2;
            input_2 <= 64'ha5cb6b95a1f3b102;
            ctrl <= 5'd6;
        end

    5'd6:
        begin
            round_keys[5] <= rho_out ^ Kminus2;
            Kminus2 <= input_1;
            input_1 <= rho_out ^ Kminus2;
            input_2 <= 64'hccc41d14c363da5d;
            ctrl <= 5'd7;
        end

    5'd7:
        begin
            round_keys[6] <= rho_out ^ Kminus2;
            Kminus2 <= input_1;
            input_1 <= rho_out ^ Kminus2;
            input_2 <= 64'h5fdc7dcd7f5a6c5c;
            ctrl <= 5'd8;
        end

    5'd8:
        begin
            round_keys[7] <= rho_out ^ Kminus2;
            Kminus2 <= input_1;
            input_1 <= rho_out ^ Kminus2;
            input_2 <= 64'hf726ffede89d6f8e;
            ctrl <= 5'd9;
        end

    5'd9:   // key schedule: end of calculating the round keys, start calculating the
    inverse decryption keys
        begin
            round_keys[8] <= rho_out ^ Kminus2;
            only_theta_req <= 1;
            input_1 <= round_keys[1];
            ctrl <= 5'd10;
        end

    5'd10:
        begin
            inv_round_keys[1] <= only_theta_out;
            input_1 <= round_keys[2];
            ctrl <= 5'd11;
        end

    5'd11:
        begin
            inv_round_keys[2] <= only_theta_out;
            input_1 <= round_keys[3];
            ctrl <= 5'd12;
        end

    5'd12:
        begin
            inv_round_keys[3] <= only_theta_out;
            input_1 <= round_keys[4];
            ctrl <= 5'd13;
        end

    5'd13:
        begin
```

```
            inv_round_keys[4] <= only_theta_out;
            input_1 <= round_keys[5];
            ctrl <= 5'd14;
        end

    5'd14:
        begin
            inv_round_keys[5] <= only_theta_out;
            input_1 <= round_keys[6];
            ctrl <= 5'd15;
        end

    5'd15:
        begin
            inv_round_keys[6] <= only_theta_out;
            input_1 <= round_keys[7];
            ctrl <= 5'd16;
        end

    5'd16:  // end of calculating the inverse keys, end of key schedule, start
    encryption/decryption
        begin
            inv_round_keys[7] <= only_theta_out;
            only_theta_req <= 0;
            if (enc)
              begin
                input_1 <= data_in ^ round_keys[0];
                input_2 <= round_keys[1];
              end
            else
              begin
                input_1 <= data_in ^ round_keys[8];
                input_2 <= only_theta_out;
              end
            ctrl <= 5'd17;
        end

    5'd17:  // encryption/decryption continuation
        begin
            input_1 <= rho_out;
            if (enc)
              input_2 <= round_keys[2];
            else
              input_2 <= inv_round_keys[6];
            ctrl <= 5'd18;
        end

    5'd18:
        begin
            input_1 <= rho_out;
            if (enc)
              input_2 <= round_keys[3];
            else
              input_2 <= inv_round_keys[5];
            ctrl <= 5'd19;
        end

    5'd19:
        begin
            input_1 <= rho_out;
            if (enc)
              input_2 <= round_keys[4];
            else
              input_2 <= inv_round_keys[4];
            ctrl <= 5'd20;
        end

    5'd20:
        begin
            input_1 <= rho_out;
            if (enc)
              input_2 <= round_keys[5];
            else
```

```
                input_2 <= inv_round_keys[3];
            ctrl <= 5'd21;
        end

    5'd21:
        begin
            input_1 <= rho_out;
            if (enc)
              input_2 <= round_keys[6];
            else
              input_2 <= inv_round_keys[2];
            ctrl <= 5'd22;
        end

    5'd22:
        begin
            input_1 <= rho_out;
            if (enc)
              input_2 <= round_keys[7];
            else
              input_2 <= inv_round_keys[1];
            ctrl <= 5'd23;
        end

    5'd23:
        begin
            input_1 <= rho_out;
            if (enc)
              input_2 <= round_keys[8];
            else
              input_2 <= round_keys[0];
            ctrl <= 5'd0;
            last_round <= 1;  // raise the flag
        end

    default:
        ;

  endcase

end

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
This module implements the pre-KHAZAD calculation needs to be done for CBC-mode encryption:
XOR with the
Initialization Vector (IV) for first_block = 1, or with the last KHAZAD cipher output
(Cminus1) for first_block = 0.
For CBC-mode decryption and for ECB-mode, this module does nothing.
**********************************************************************************************
***********
**********************************************************************************************
***********/
module op_mode_enc
(
  input              op_mode     ,   // 0: ECB  1:CBC
  input      [63:0] d_in        ,
  input      [63:0] Cminus1     ,
  input      [63:0] IV          ,
  input              enc_dec_SEL ,   // 0: dec  1: enc
  input              first_block ,
  output reg [63:0] d_out
);

always @(*)
begin
    if ((enc_dec_SEL == 0)||(op_mode == 0))  // for decryption and for ECB-mode do nothing
        d_out <= d_in;
    else                                     // for CBC-mode encryption
        begin
            if (first_block == 1)
                d_out <= d_in ^ IV;
            else
                d_out <= d_in ^ Cminus1;
        end
end

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
This module implements the post-KHAZAD calculation needs to be done for CBC-mode decryption:
XOR with the
Initialization Vector (IV) for first_block = 1, or with the last cipher input (Cminus1) for
first_block = 0.
For CBC-mode encryption and for ECB-mode, this module does nothing.
**********************************************************************************************
***********
**********************************************************************************************
***********/
module op_mode_dec
(
  input              op_mode     ,   // 0: ECB  1:CBC
  input      [63:0] d_in        ,
  input      [63:0] Cminus1     ,
  input      [63:0] IV          ,
  input              enc_dec_SEL ,   // 0: dec  1: enc
  input              first_block ,
  output reg [63:0] d_out
);

always @(*)
begin
    if ((enc_dec_SEL == 1)||(op_mode == 0))  // for encryption and for ECB-mode do nothing
        d_out <= d_in;
    else                                     // for CBC-mode decryption
        begin
            if (first_block == 1)
                d_out <= d_in ^ IV;
            else
                d_out <= d_in ^ Cminus1;
        end
end

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
For CBC encryption, XOR is made with the last cipher output, so d_out_KHAZAD is used.
But for CBC decryption XOR is made with previous cipher input, not the current cipher input,
so the previous cipher
needs to be saved in memory. We named it Cminus1[63:0].
The current cipher input may be needed for future XOR. In case this input will not remain
valid in the input ports,
we save it too as C[63:0].
**********************************************************************************************
***********
**********************************************************************************************
***********/
module CBC_dec_memory
(
  input              CLK     ,
  input              RST     ,
  input              start   ,
  input      [63:0] C        ,
  output reg [63:0] Cminus1
);

reg [63:0] mem_buffer;

always @(posedge CLK)
begin
    if (RST)
        begin
            Cminus1 <= 64'h0;
            mem_buffer <= 64'h0;
        end
    else
        if (start) // for each start pulse, advance the saved ciphers
            begin
                mem_buffer <= C;
                Cminus1 <= mem_buffer;
            end
end

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
Module KHAZAD by itself can only give results for ECB mode. This module envelops module
KHAZAD together with the
modules and wiring necessary to implement CBC mode: op_mode_enc, CBC_dec_memory, op_mode_dec.
The initial d_in is going into modules op_mode_enc and CBC_dec_memory, coming out of
op_mode_enc as d_in_KHAZAD,
then going into KHAZAD.
KHAZAD output is coming out as d_out_KHAZAD, then going into op_mode_enc (for possible
future XOR)
and into op_mode_dec. The final output is going out of op_mode_dec as d_out.
Cminus1 is the output of module CBC_dec_memory, and it is used in module op_mode_dec.
**********************************************************************************************
***********
**********************************************************************************************
***********/
module enveloped_KHAZAD
(
input           CLK             ,
input           RST             ,
input  [127:0]  k_in            ,
input  [63:0]   d_in            ,
input  [63:0]   IV              ,
input           only_data       ,
input           enc_dec_SEL     ,
input           op_mode         ,
input           first_block     ,
input           start           ,
output [63:0]   d_out           ,
output          last_round
);

wire [63:0] d_in_KHAZAD, d_out_KHAZAD, Cminus1;

op_mode_enc     op_mode_1  (op_mode, d_in, d_out_KHAZAD, IV, enc_dec_SEL, first_block,
d_in_KHAZAD);
KHAZAD          KHZD       (d_in_KHAZAD, k_in, CLK, RST, enc_dec_SEL, start, only_data,
d_out_KHAZAD, last_round);
CBC_dec_memory  CBC_mem    (CLK, RST, start, d_in, Cminus1);
op_mode_dec     op_mode_2  (op_mode, d_out_KHAZAD, Cminus1, IV, enc_dec_SEL, first_block,
d_out);

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
This module manages control signals between the PS program and the PL design, and some
indicator LEDs output.
Input ctrl_from_PS[5:0] is received from the PS via AXI:
      ctrl_from_PS[5] - RST                                                    1: reset
      command. 0: no reset command.
      ctrl_from_PS[4] - only_data                                              1: new data,
      same key. 0: new data and new key.
      ctrl_from_PS[3] - enc_dec_SEL: the desired operation                     1:
      encryption. 0: decryption.
      ctrl_from_PS[2] - op_mode: the desired cryptographic mode of operation   1: CBC. 0:
      ECB.
      ctrl_from_PS[1] - first_block: flag for the CBC mode                     1: first
      data block. 0: not first data block.
      ctrl_from_PS[0] - bistable input start/ready flag. To start an operation this bit must
      not be equal to the ctrl_to_PS flag (start_condition=1).
      When bits are equal, start_condition=0, and the PS will be notified that operation has
      ended.
Input finish: a pulse from KHAZAD when last_round has ended.
Output ctrl_to_PS: bistable output start/ready flag sent to the PS via AXI to notify PS that
operation has ended.
**********************************************************************************************
***********
**********************************************************************************************
***********
Version 2.0: ECB+CBC implementation
**********************************************************************************************
***********
**********************************************************************************************
***********/
module controller
(
input               CLK             ,
input       [5:0]   ctrl_from_PS    ,
input               finish          ,
output              RST             ,
output              only_data       ,
output              enc_dec_SEL     ,
output              op_mode         ,
output              first_block     ,
output              start           ,
output  reg         ctrl_to_PS      ,
output              RST_LED         ,
output              PL_ready_LED    ,
output              encryption_LED  ,
output              decryption_LED  ,
output              ECB_LED         ,
output              CBC_LED
);

assign RST              = ctrl_from_PS[5];
assign only_data        = ctrl_from_PS[4];
assign enc_dec_SEL      = ctrl_from_PS[3];
assign op_mode          = ctrl_from_PS[2];
assign first_block      = ctrl_from_PS[1];

wire start_condition;
reg start_EN;

assign start_condition = ctrl_from_PS[0] ^ ctrl_to_PS;
assign start = start_condition & start_EN; // start_EN makes start a one-cycle pulse
```

```
always @(posedge CLK)
begin
    if (RST)
        ctrl_to_PS <= 0;
    else if (finish)                    // a pulse from KHAZAD when last_round has ended
        ctrl_to_PS <= ctrl_from_PS[0]; // when bits are equal, start_condition=0, and PS
        notified that operation has ended
end

always @(posedge CLK)
begin
    if (RST)
        start_EN <= 1;
    else
        begin
            if (!start_condition)
                start_EN <= 1;
            else if (start_EN)
                start_EN <= 0;
        end
end

// Indicator LEDs output:
assign RST_LED = RST;
assign PL_ready_LED = (!RST) && (!start_condition); // =nor. LED ON when KHAZAD waiting, OFF
when busy
assign encryption_LED = enc_dec_SEL;
assign decryption_LED = !enc_dec_SEL;
assign ECB_LED = !op_mode;
assign CBC_LED = op_mode;

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
This module is merely an intermediate between the PS and the PL design.
It concatenates the four 32-bit key_in parts input into one 128-bit key for the PL,
the two 32-bit d_in parts input into one 64-bit d_in for the PL,
the two 32-bit IV parts input into one 64-bit IV for the PL,
and splits the 64-bit d_out output into two 32-bit d_out for the PS.
**********************************************************************************************
***********
**********************************************************************************************
***********
Version 2.0: ECB+CBC implementation
**********************************************************************************************
***********
**********************************************************************************************
***********/
module wiring
(
  input  [31:0]  k_in_1 ,
  input  [31:0]  k_in_2 ,
  input  [31:0]  k_in_3 ,
  input  [31:0]  k_in_4 ,
  input  [31:0]  d_in_1 ,
  input  [31:0]  d_in_2 ,
  input  [63:0]  d_out  ,
  input  [31:0]  IV_1   ,
  input  [31:0]  IV_2   ,
  output [127:0] k_in   ,
  output [63:0]  d_in   ,
  output [31:0]  d_out_1,
  output [31:0]  d_out_2,
  output [63:0]  IV
);

assign k_in = {k_in_1, k_in_2, k_in_3, k_in_4};
assign d_in = {d_in_1, d_in_2};
assign IV = {IV_1, IV_2};
assign d_out_1 = d_out[63:32];
assign d_out_2 = d_out[31:0];

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
This module manages control signals between the PS program and the PL design, and some
indicator LEDs output.
Input ctrl_from_PS[3:0] is received from the PS via AXI:
      ctrl_from_PS[3] - RST                                  1: reset command. 0: no reset
      command.
      ctrl_from_PS[2] - only_data                            1: new data, same key. 0: new
      data and new key.
      ctrl_from_PS[1] - enc_dec_SEL: the desired operation  1: encryption. 0: decryption.
      ctrl_from_PS[0] - bistable input start/ready flag. To start an operation this bit must
      not be equal to the ctrl_to_PS flag (start_condition=1).
      When bits are equal, start_condition=0, and the PS will be notified that operation has
      ended.
Input finish: a pulse from KHAZAD when last_round has ended.
Output ctrl_to_PS: bistable output start/ready flag sent to the PS via AXI to notify PS that
operation has ended.
**********************************************************************************************
***********
**********************************************************************************************
***********
Version 1.0: ECB implementation
**********************************************************************************************
***********
**********************************************************************************************
***********/
module controller
(
input               CLK             ,
input       [3:0]   ctrl_from_PS    ,
input               finish          ,
output              RST             ,
output              only_data       ,
output              enc_dec_SEL     ,
output              start           ,
output  reg         ctrl_to_PS      ,
output              RST_LED         ,
output              encryption_LED  ,
output              decryption_LED  ,
output              PL_ready_LED
);

assign RST              = ctrl_from_PS[3];
assign only_data        = ctrl_from_PS[2];
assign enc_dec_SEL      = ctrl_from_PS[1];

wire start_condition;
reg  start_EN;

assign start_condition = ctrl_from_PS[0] ^ ctrl_to_PS;
assign start = start_condition & start_EN; // start_EN makes start a one-cycle pulse

always @(posedge CLK)
begin
    if (RST)
        ctrl_to_PS <= 0;
    else if (finish)                    // a pulse from KHAZAD when last_round has ended
        ctrl_to_PS <= ctrl_from_PS[0]; // when bits are equal, start_condition=0, and PS
        notified that operation has ended
end

always @(posedge CLK)
```

```
begin
    if (RST)
        start_EN <= 1;
    else
        begin
            if (!start_condition)
                start_EN <= 1;
            else if (start_EN)
                start_EN <= 0;
        end
end

// Indicator LEDs output:
assign RST_LED = RST;
assign encryption_LED = enc_dec_SEL;
assign decryption_LED = !enc_dec_SEL;
assign PL_ready_LED = (!RST) && (!start_condition); // =nor. LED ON when KHAZAD waiting, OFF
when busy

endmodule
```

```
/**********************************************************************************************
***********
**********************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
**********************************************************************************************
***********
**********************************************************************************************
***********
This module is merely an intermediate between the PS and the PL design.
It concatenates the four 32-bit key_in parts input into one 128-bit key for the PL,
the two 32-bit d_in parts input into one 64-bit d_in for the PL,
and splits the 64-bit d_out output into two 32-bit d_out for the PS.
**********************************************************************************************
***********
**********************************************************************************************
***********
Version 1.0: ECB implementation
**********************************************************************************************
***********
**********************************************************************************************
***********/
module wiring
(
  input  [31:0] k_in_1  ,
  input  [31:0] k_in_2  ,
  input  [31:0] k_in_3  ,
  input  [31:0] k_in_4  ,
  input  [31:0] d_in_1  ,
  input  [31:0] d_in_2  ,
  input  [63:0] d_out   ,
  output [127:0] k_in   ,
  output [63:0] d_in    ,
  output [31:0] d_out_1 ,
  output [31:0] d_out_2
);

assign k_in = {k_in_1, k_in_2, k_in_3, k_in_4};
assign d_in = {d_in_1, d_in_2};
assign d_out_1 = d_out[63:32];
assign d_out_2 = d_out[31:0];

endmodule
```

```
#ifndef PORTABLE_C__
#define PORTABLE_C__
/*****************************************************************************
***********
*****************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
*****************************************************************************
***********
*****************************************************************************
***********
This file is essentially the original reference code header file "nessie.h",
slightly modified for data type compatibility with xilinx libraries to avoid conflict
definitions.
"xil_types.h" have been included in this file, and a few definitions have been commented.
The original file is available at:
https://www.cosic.esat.kuleuven.be/nessie/tweaks.html
*****************************************************************************
***********
*****************************************************************************
***********/

#include <limits.h>
#include "xil_types.h" // for data type compatibility

/* Definition of minimum-width integer types
 *
 * u8   -> unsigned integer type, at least 8 bits, equivalent to unsigned char
 * u16  -> unsigned integer type, at least 16 bits
 * u32  -> unsigned integer type, at least 32 bits
 *
 * s8, s16, s32  -> signed counterparts of u8, u16, u32
 *
 * Always use macro's T8(), T16() or T32() to obtain exact-width results,
 * i.e., to specify the size of the result of each expression.
 */

typedef signed char s8;
typedef unsigned char u8;

#if UINT_MAX >= 4294967295UL

typedef signed short s16;
//typedef signed int s32; -- declared in "xil_types.h"
typedef unsigned short u16;
//typedef unsigned int u32; -- declared in "xil_types.h"

#define ONE32   0xffffffffU

#else

typedef signed int s16;
typedef signed long s32;
typedef unsigned int u16;
typedef unsigned long u32;

#define ONE32   0xffffffffUL

#endif

#define ONE8    0xffU
#define ONE16   0xffffU

#define T8(x)   ((x) & ONE8)
#define T16(x)  ((x) & ONE16)
#define T32(x)  ((x) & ONE32)

/*
 * If you want 64-bit values, uncomment the following lines; this
```

```
 * reduces portability.
 */
#if ((1UL << 31) * 2UL) != 0UL
#ifdef _MSC_VER
typedef unsigned __int64 u64;
typedef signed __int64 s64;
#else  /* !_MSC_VER */
//typedef unsigned long u64; -- declared in "xil_types.h"
//typedef signed long s64; -- declared in "xil_types.h"
#define ONE64   0xffffffffffffffffUL
#endif /* ?_MSC_VER */
#else
#ifdef _MSC_VER
typedef unsigned __int64 u64;
typedef signed __int64 s64;
#else  /* !_MSC_VER */
typedef unsigned long long u64;
typedef signed long long s64;
#define ONE64   0xffffffffffffffffULL
#endif /* ?_MSC_VER */
#endif
#define T64(x)  ((x) & ONE64)
/*
 * Note: the test is used to detect native 64-bit architectures;
 * if the unsigned long is strictly greater than 32-bit, it is
 * assumed to be at least 64-bit. This will not work correctly
 * on (old) 36-bit architectures (PDP-11 for instance).
 *
 * On non-64-bit architectures, "long long" is used.
 */

/*
 * U8TO32_BIG(c) returns the 32-bit value stored in big-endian convention
 * in the unsigned char array pointed to by c.
 */
#define U8TO32_BIG(c)  (((u32)T8(*(c)) << 24) | ((u32)T8(*((c) + 1)) << 16) \
                        ((u32)T8(*((c) + 2)) << 8) | ((u32)T8(*((c) + 3))))

/*
 * U8TO32_LITTLE(c) returns the 32-bit value stored in little-endian convention
 * in the unsigned char array pointed to by c.
 */
#define U8TO32_LITTLE(c)  (((u32)T8(*(c))) | ((u32)T8(*((c) + 1)) << 8) \
                       (u32)T8(*((c) + 2)) << 16) | ((u32)T8(*((c) + 3)) << 24))

/*
 * U8TO32_BIG(c, v) stores the 32-bit-value v in big-endian convention
 * into the unsigned char array pointed to by c.
 */
#define U32TO8_BIG(c, v)    do { \
        u32 x = (v); \
        u8 *d = (c); \
        d[0] = T8(x >> 24); \
        d[1] = T8(x >> 16); \
        d[2] = T8(x >> 8); \
        d[3] = T8(x); \
    } while (0)

/*
 * U8TO32_LITTLE(c, v) stores the 32-bit-value v in little-endian convention
 * into the unsigned char array pointed to by c.
 */
#define U32TO8_LITTLE(c, v)    do { \
        u32 x = (v); \
        u8 *d = (c); \
        d[0] = T8(x); \
        d[1] = T8(x >> 8); \
        d[2] = T8(x >> 16); \
        d[3] = T8(x >> 24); \
    } while (0)

/*
 * ROTL32(v, n) returns the value of the 32-bit unsigned value v after
```

```
 * a rotation of n bits to the left. It might be replaced by the appropriate
 * architecture-specific macro.
 *
 * It evaluates v and n twice.
 *
 * The compiler might emit a warning if n is the constant 0. The result
 * is undefined if n is greater than 31.
 */
#define ROTL32(v, n)   (T32((v) << (n)) | ((v) >> (32 - (n))))

/*
 *  Khazad-specific definitions
 */
#define R           8
#define KEYSIZE     128
#define KEYSIZEB    (KEYSIZE/8) // = 16
#define BLOCKSIZE   64
#define BLOCKSIZEB  (BLOCKSIZE/8) // = 8

typedef struct NESSIEstruct {
    u32 roundKeyEnc[R + 1][2];
    u32 roundKeyDec[R + 1][2];
} NESSIEstruct;

/**
 * Create the Khazad key schedule for a given cipher key.
 * Both encryption and decryption key schedules are generated.
 *
 * @param key           The 128-bit cipher key.
 * @param structpointer Pointer to the structure that will hold the expanded key.
 */
void NESSIEkeysetup(const unsigned char * const key,
                    struct NESSIEstruct * const structpointer);

/**
 * Encrypt a data block.
 *
 * @param   structpointer   the expanded key.
 * @param   plaintext       the data block to be encrypted.
 * @param   ciphertext      the encrypted data block.
 */
void NESSIEencrypt(const struct NESSIEstruct * const structpointer,
                   const unsigned char * const plaintext,
                         unsigned char * const ciphertext);

/**
 * Decrypt a data block.
 *
 * @param   structpointer   the expanded key.
 * @param   ciphertext      the data block to be decrypted.
 * @param   plaintext       the decrypted data block.
 */
void NESSIEdecrypt(const struct NESSIEstruct * const structpointer,
                   const unsigned char * const ciphertext,
                         unsigned char * const plaintext);


#endif   /* PORTABLE_C__ */
```

```
#ifndef KHAZAD_ZYNQ_H
#define KHAZAD_ZYNQ_H
/*****************************************************************************
***********
*****************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
*****************************************************************************
***********
*****************************************************************************
***********
This is the KHAZAD_Zynq library header file, containing the functions and definitions
to run the various applications of the design. The file defines these functions:
1.  USR_button_ISR
2.  board_configuration
3.  Zynq_crypt
4.  Zynq_crypt_simple
5.  test_vectors_full
6.  test_vectors_short
7.  demonstration
8.  application
9.  print_data
10. compare_blocks
11. about
Identical lines of code may appear in some of functions. This was done to ease the re-use of
the functions
as standalone programs.
*****************************************************************************
***********
*****************************************************************************
***********
Version 2.0: ECB+CBC implementation
*****************************************************************************
***********
*****************************************************************************
***********/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include "platform.h"
#include "xgpio.h"
#include "xgpiops.h"
#include "xparameters.h"
#include "xil_types.h"
#include "xstatus.h"
#include "Xscugic.h"
#include "xil_exception.h"
#include "nessie_modified.h"
#include "khazad-tweak32.h"

/*****************************************************************************
***********
*****************************************************************************
***********
  Globals definitions
*****************************************************************************
***********
*****************************************************************************
***********/

// AXI_GPIO modules registers adresses:
#define ADDR0 ((unsigned int)0x41200000)
#define ADDR1 ((unsigned int)0x41210000)
#define ADDR2 ((unsigned int)0x41220000)
#define ADDR3 ((unsigned int)0x41230000)
#define ADDR4 ((unsigned int)0x41240000)
#define ADDR_offset ((unsigned int)0x00000008)
```

```
// maximal length of input strings, in number of characters, including spaces:
#define MAX_LENGTH  400

/************************************************************************************************
************
*************************************************************************************************
************
  Global variables declarations
*************************************************************************************************
************
*************************************************************************************************
************/

static XGpioPs my_Gpio; // for the PS
static XGpio GPIO_4;    // for the bidirectional module AXI_GPIO_4
static XScuGic my_Gic;  // for GIC: general interrupt controller
static u16 ctrl;        // for sending instructions to the PL design. Further details in
main.c file.
static char *hex = "0123456789ABCDEF"; // for base conversion functions

/************************************************************************************************
************
*************************************************************************************************
************
  Functions definitions
*************************************************************************************************
************
*************************************************************************************************
************/

/************************************************************************************************
************
  1. USR_button_ISR: the user button Interrupt Service Routine.
  The function communicates with the PL fabric. It makes use of the static variable ctrl
  and configure it to give the appropriate instructions to the design.
*************************************************************************************************
************/
static void USR_button_ISR(void *CallBackReff)
{
  XGpioPs_IntrDisablePin(&my_Gpio, 51); // for debouncing the switch
  XGpioPs_IntrClearPin(&my_Gpio, 51);   // clear the interrupt flag
  XGpioPs_WritePin(&my_Gpio, 47, 0);    // turn off the PS-ready indicator LED
  ctrl = 0x0020;           // reset=1
  Xil_Out16(ADDR0,ctrl);  // send from PS to FPGA via AXI interface
  printf("\t\tReset was asserted!\n\r");
  sleep(1);                // for debouncing the switch
  ctrl = 0x0000;           // reset=0
  Xil_Out16(ADDR0,ctrl);
  XGpioPs_WritePin(&my_Gpio, 47, 1);    // turn on the LED
  XGpioPs_IntrEnablePin(&my_Gpio, 51);
}

/************************************************************************************************
************
  2. board_configuration: configures the PS, the peripherals, the bidirectional AXI_GPIO and
  the interrupt.
*************************************************************************************************
************/
void board_configuration()
{
  XGpioPs_Config *GPIO_Config;
  GPIO_Config = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
  u16 status;
  status = XGpioPs_CfgInitialize(&my_Gpio, GPIO_Config, GPIO_Config->BaseAddr);
  if (status == XST_SUCCESS)
    printf("XGpioPs configuration successful! \n\r");
  else
    printf("XGpioPs configuration failed! \n\r");
  XGpioPs_SetDirectionPin(&my_Gpio, 47, 1); // board LED, output
  XGpioPs_SetOutputEnablePin(&my_Gpio, 47, 1);
  XGpioPs_SetDirectionPin(&my_Gpio, 51, 0); // USR button, input
  XGpioPs_SetDirectionPin(&my_Gpio, 54, 0); // EMIO pin, input
```

```
  status = XGpio_Initialize(&GPIO_4, XPAR_AXI_GPIO_4_DEVICE_ID); // bidirectional AXI_GPIO
  module
  if (status == XST_SUCCESS)
    printf("AXI_GPIO configuration successful! \n\r");
  else
    printf("AXI_GPIO configuration failed! \n\r");
  // Interrupt configuration:
  XScuGic_Config *Gic_Config;
  Gic_Config = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
  status = XScuGic_CfgInitialize(&my_Gic, Gic_Config, Gic_Config->CpuBaseAddress);
  if (status == XST_SUCCESS)
    printf("GIC configuration successful! \n\r");
  else
    printf("GIC configuration failed! \n\r");
  Xil_ExceptionInit(); // initialize exception handlers on the processor
  Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
  (Xil_ExceptionHandler)XScuGic_InterruptHandler, &my_Gic);
  status = XScuGic_Connect(&my_Gic, XPS_GPIO_INT_ID, (Xil_ExceptionHandler)USR_button_ISR,
  (void *)&my_Gic);
  if (status == XST_SUCCESS)
    printf("GPIO interrupt handler connection successful! \n\r");
  else
    printf("GPIO interrupt handler connection failed! \n\r");
  XGpioPs_SetIntrTypePin(&my_Gpio, 51, XGPIOPS_IRQ_TYPE_EDGE_RISING); // interrupt on rising
  edge
  XGpioPs_IntrClearPin(&my_Gpio, 51);  // clear any pending residual USR_button interrupts
  XGpioPs_IntrEnablePin(&my_Gpio, 51); // enable USR_button interrupt
  XScuGic_Enable(&my_Gic, XPS_GPIO_INT_ID); // enable GPIO interrupt (SPI: Shared Peripheral
  Interrupt)
  Xil_ExceptionEnable(); // enable interrupt handling
  XGpioPs_WritePin(&my_Gpio, 47, 1); // turn on the PS-ready indicator LED
}

/*****************************************************************************************
***********
  3. Zynq_crypt: the main function that communicates with the PL fabric and get it to execute
  a "crypt" operation: encryption or decryption. The function makes use of the static
  variable ctrl
  and configure it to give the appropriate instructions to the design.
  Function input parameters:
  text: pointer to u8 data-in array.
  key1, key2, key3, key4: four u32 parts of the key.
  IV1, IV2: two u32 parts of the IV (Initialization Vector).
  only_data: indicates the key period.                          1: new data, same key. 0:
  new data and new key.
  enc_dec: flag to the desired operation.                       1: encryption. 0: decryption.
  op_mode: flag to the desired cryptographic mode of operation. 1: CBC. 0: ECB.
  first_block: flag for the CBC mode.                           1: first data block. 0: not
  first data block.
  new_IV: new IV flag.                                          1: new IV. 0: same IV.
  Function returns:
  result: pointer to u8 data out array.
*****************************************************************************************
***********/
void Zynq_crypt(const u8 * const text, const u32 key1, const u32 key2, const u32 key3, const
u32 key4, const u32 IV1, const u32 IV2, const bool only_data, const bool enc_dec, const bool
op_mode, const bool first_block, const bool new_IV, u8 * const result)
{
  XGpioPs_WritePin(&my_Gpio, 47, 0); // turn off the PS-ready indicator LED
  // map u8-array text to two u32 d_in parts:
  u32 d_in_1 =
        ((u32)text[0] << 24) ^
        ((u32)text[1] << 16) ^
        ((u32)text[2] <<  8) ^
        ((u32)text[3]      );

  u32 d_in_2 =
        ((u32)text[4] << 24) ^
        ((u32)text[5] << 16) ^
        ((u32)text[6] <<  8) ^
        ((u32)text[7]      );

  if (!only_data)
```

```
  {
      // send key:
      Xil_Out32(ADDR1,key1)               ;
      Xil_Out32(ADDR1 + ADDR_offset,key2);
      Xil_Out32(ADDR2,key3)               ;
      Xil_Out32(ADDR2 + ADDR_offset,key4);
  }

  if ((op_mode == 1) && (new_IV))
  {
      // send IV:
      Xil_Out32(ADDR3,IV1)                ;
      Xil_Out32(ADDR3 + ADDR_offset,IV2);
  }

  // set GPIO_4 to output and send data:
  XGpio_SetDataDirection(&GPIO_4,1,0x00000000);
  Xil_Out32(ADDR4               ,d_in_1);
  Xil_Out32(ADDR4 + ADDR_offset,d_in_2);

  // ctrl setting:
  ctrl = ctrl ^ 0x0001;   // toggle bit 0, to issue a start command
  if (first_block == 1)
    ctrl = ctrl | 0x0002; // = ctrl|000010, turn on bit 1
  else
    ctrl = ctrl & 0xFFFD; // = ctrl&111101, turn off bit 1
  if (op_mode == 1)
    ctrl = ctrl | 0x0004; // = ctrl|000100, turn on bit 2
  else
    ctrl = ctrl & 0xFFFB; // = ctrl&111011, turn off bit 2
  if (enc_dec == 1)
    ctrl = ctrl | 0x0008; // = ctrl|001000, turn on bit 3
  else
    ctrl = ctrl & 0xFFF7; // = ctrl&110111, turn off bit 3
  if (only_data == 1)
    ctrl = ctrl | 0x0010; // = ctrl|010000, turn on bit 4
  else
    ctrl = ctrl & 0xFFEF; // = ctrl&101111, turn off bit 4

  Xil_Out16(ADDR0,ctrl);
  // can't send the start command before both key & data sent because key schedule will end
  before data has arrived,
  // and additional flags, and time to send them, will be needed.

  u16 finish;
  // set GPIO_4 to input and wait for data:
  XGpio_SetDataDirection(&GPIO_4,1,0xFFFFFFFF);
  do {
    finish = XGpioPs_ReadPin(&my_Gpio, 54); // read from FPGA via EMIO interface (polling)
    } while ((finish ^ ctrl) & 0x0001); // while LSB of finish != LSB of ctrl

  // read data from PL via AXI bus:
  u32 d_out_1 = Xil_In32(ADDR4);
  u32 d_out_2 = Xil_In32(ADDR4 + ADDR_offset);

  // map two u32 d_out parts to u8-array text:
  result[0] = (u8)(d_out_1 >> 24);
  result[1] = (u8)(d_out_1 >> 16);
  result[2] = (u8)(d_out_1 >>  8);
  result[3] = (u8)(d_out_1      );
  result[4] = (u8)(d_out_2 >> 24);
  result[5] = (u8)(d_out_2 >> 16);
  result[6] = (u8)(d_out_2 >>  8);
  result[7] = (u8)(d_out_2      );

  XGpioPs_WritePin(&my_Gpio, 47, 1); // turn on the LED
}

/*****************************************************************************************
***********
  4. Zynq_crypt_simple: this function is very similar to Zynq_crypt, just more simple (less
  input parameters),
  for use in the test vectors operations.
```

```
  The key is given as one parameter, a pointer to u8 array - reflecting the way it's
  calculated in the test vectors files,
  and mapping to four u32 key parts is done inside the function. Also the function lacks the
  only_data option and the CBC option.
  One can easily create different versions of this function for other kinds of tests.
*****************************************************************************************
***********/
void Zynq_crypt_simple(const u8 * const text, const u8 * const key, const bool enc_dec, u8 *
const result)
{
  XGpioPs_WritePin(&my_Gpio, 47, 0); // turn off the PS-ready indicator LED
  // map u8-array text to two u32 d_in parts:
  u32 d_in_1 =
        ((u32)text[0] << 24) ^
        ((u32)text[1] << 16) ^
        ((u32)text[2] <<  8) ^
        ((u32)text[3]      );

  u32 d_in_2 =
        ((u32)text[4] << 24) ^
        ((u32)text[5] << 16) ^
        ((u32)text[6] <<  8) ^
        ((u32)text[7]      );

  // map u8-array key to four u32 key parts:
  u32 key1 =
        ((u32)key[ 0] << 24) ^
        ((u32)key[ 1] << 16) ^
        ((u32)key[ 2] <<  8) ^
        ((u32)key[ 3]      );

  u32 key2 =
        ((u32)key[ 4] << 24) ^
        ((u32)key[ 5] << 16) ^
        ((u32)key[ 6] <<  8) ^
        ((u32)key[ 7]      );

  u32 key3 =
        ((u32)key[ 8] << 24) ^
        ((u32)key[ 9] << 16) ^
        ((u32)key[10] <<  8) ^
        ((u32)key[11]      );

  u32 key4 =
        ((u32)key[12] << 24) ^
        ((u32)key[13] << 16) ^
        ((u32)key[14] <<  8) ^
        ((u32)key[15]      );

  // send key:
  Xil_Out32(ADDR1,key1)                   ;
  Xil_Out32(ADDR1 + ADDR_offset,key2)     ;
  Xil_Out32(ADDR2,key3)                   ;
  Xil_Out32(ADDR2 + ADDR_offset,key4)     ;
  // set GPIO_4 to output and send data:
  XGpio_SetDataDirection(&GPIO_4,1,0x00000000);
  Xil_Out32(ADDR4               ,d_in_1) ;
  Xil_Out32(ADDR4 + ADDR_offset,d_in_2) ;

  // ctrl setting:
  ctrl = ctrl ^ 0x0001;   // toggle bit 0, to issue a start command
  if (enc_dec == 1)
    ctrl = ctrl | 0x0008; // = ctrl|001000, turn on bit 3
  else
    ctrl = ctrl & 0xFFF7; // = ctrl&110111, turn off bit 3
  ctrl = ctrl & 0xFFE9;   // = ctrl&101001, turn off bits 4,2,1. only_data and CBC mode
  aren't active

  Xil_Out16(ADDR0,ctrl);
  // can't send the start command before both key & data sent because key schedule will end
  before data has arrived,
  // and additional flags, and time to send them, will be needed.
```

```
  u16 finish;
  // set GPIO_4 to input and wait for data:
  XGpio_SetDataDirection(&GPIO_4,1,0xFFFFFFFF);
  do {
    finish = XGpioPs_ReadPin(&my_Gpio, 54); // read from FPGA via EMIO interface (polling)
    } while ((finish ^ ctrl) & 0x0001); // while LSB of finish != LSB of ctrl

  // read data from PL via AXI bus:
  u32 d_out_1 = Xil_In32(ADDR4);
  u32 d_out_2 = Xil_In32(ADDR4 + ADDR_offset);

  // map two u32 d_out parts to u8-array text:
  result[0] = (u8)(d_out_1 >> 24);
  result[1] = (u8)(d_out_1 >> 16);
  result[2] = (u8)(d_out_1 >>  8);
  result[3] = (u8)(d_out_1      );
  result[4] = (u8)(d_out_2 >> 24);
  result[5] = (u8)(d_out_2 >> 16);
  result[6] = (u8)(d_out_2 >>  8);
  result[7] = (u8)(d_out_2      );

  XGpioPs_WritePin(&my_Gpio, 47, 1); // turn on the LED
}

/*****************************************************************************************
***********
  5. test_vectors_full: this function is essentially the original reference code
  bctestvectors.c code,
  combined with encryption and decryption on Zynq, to test the hardware implementation ECB
  results.
  If these results are different than the reference code results, an error massage appears.
  This test includes a stage of 10^8 iterations, so it may take some time to be completed.
*****************************************************************************************
***********/
void test_vectors_full()
{
  struct NESSIEstruct subkeys;
  u8 key[KEYSIZEB];
  u8 plain[BLOCKSIZEB];
  u8 cipher[BLOCKSIZEB];
  u8 decrypted[BLOCKSIZEB];
  u32 i;
  int v;
  u8 Zynq_cipher[BLOCKSIZEB];
  u8 Zynq_decrypted[BLOCKSIZEB];

  printf("Test vectors -- set 1\n");
  printf("=====================\n\n");

  /* If key size is not a multiple of 8, this tests too much (intentionally) */
  for(v=0; v<(KEYSIZEB*8); v++)
    {
      memset(plain, 0, BLOCKSIZEB);
      memset(key, 0, KEYSIZEB);
      key[v>>3] = 1<<(7-(v&7));

      NESSIEkeysetup(key, &subkeys);
      NESSIEencrypt(&subkeys, plain, cipher);
      NESSIEdecrypt(&subkeys, cipher, decrypted);

      printf("Set 1, vector#%3d:\n", v);
      print_data("key", key, KEYSIZEB);
      print_data("plain", plain, BLOCKSIZEB);
      print_data("cipher", cipher, BLOCKSIZEB);
      print_data("decrypted", decrypted, BLOCKSIZEB);

      if(compare_blocks(plain, decrypted, BLOCKSIZE) != 0)
        printf("** Decryption error: **\n"
               "   Decrypted ciphertext is different than the plaintext!\n");

      Zynq_crypt_simple(plain, key, 1, Zynq_cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 0, Zynq_decrypted);
      print_data("Zynq cipher", Zynq_cipher, BLOCKSIZEB);
```

```
    print_data("Zynq decrypted", Zynq_decrypted, BLOCKSIZEB);

    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    if(compare_blocks(plain, Zynq_decrypted, BLOCKSIZE) != 0)
      printf("** Zynq decryption error: **\n"
             "   Decrypted ciphertext on Zynq is different than the plaintext!\n");

    for(i=0; i<99; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 100 times", cipher, BLOCKSIZEB);
    print_data("Iterated 100 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");
    for(i=0; i<900; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 1000 times", cipher, BLOCKSIZEB);
    print_data("Iterated 1000 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    printf("\n");
  }

printf("Test vectors -- set 2\n");
printf("=====================\n\n");

/* If block size is not a multiple of 8, this tests too much (intentionally) */
for(v=0; v<(BLOCKSIZEB*8); v++)
  {
    memset(plain, 0, BLOCKSIZEB);
    memset(key, 0, KEYSIZEB);
    plain[v>>3] = 1<<(7-(v&7));

    NESSIEkeysetup(key, &subkeys);
    NESSIEencrypt(&subkeys, plain, cipher);
    NESSIEdecrypt(&subkeys, cipher, decrypted);

    printf("Set 2, vector#%3d:\n", v);
    print_data("key", key, KEYSIZEB);
    print_data("plain", plain, BLOCKSIZEB);
    print_data("cipher", cipher, BLOCKSIZEB);
    print_data("decrypted", decrypted, BLOCKSIZEB);

    if(compare_blocks(plain, decrypted, BLOCKSIZE) != 0)
      printf("** Decryption error: **\n"
             "   Decrypted ciphertext is different than the plaintext!\n");

    Zynq_crypt_simple(plain, key, 1, Zynq_cipher);
    Zynq_crypt_simple(Zynq_cipher, key, 0, Zynq_decrypted);
    print_data("Zynq cipher", Zynq_cipher, BLOCKSIZEB);
    print_data("Zynq decrypted", Zynq_decrypted, BLOCKSIZEB);

    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    if(compare_blocks(plain, Zynq_decrypted, BLOCKSIZE) != 0)
      printf("** Zynq decryption error: **\n"
             "   Decrypted ciphertext on Zynq is different than the plaintext!\n");

    for(i=0; i<99; i++)
    {
```

```
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 100 times", cipher, BLOCKSIZEB);
    print_data("Iterated 100 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");
    for(i=0; i<900; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 1000 times", cipher, BLOCKSIZEB);
    print_data("Iterated 1000 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    printf("\n");
  }

printf("Test vectors -- set 3\n");
printf("=====================\n\n");

for(v=0; v<256; v++)
  {
    memset(plain, v, BLOCKSIZEB);
    memset(key, v, KEYSIZEB);

    NESSIEkeysetup(key, &subkeys);
    NESSIEencrypt(&subkeys, plain, cipher);
    NESSIEdecrypt(&subkeys, cipher, decrypted);

    printf("Set 3, vector#%3d:\n", v);
    print_data("key", key, KEYSIZEB);
    print_data("plain", plain, BLOCKSIZEB);
    print_data("cipher", cipher, BLOCKSIZEB);
    print_data("decrypted", decrypted, BLOCKSIZEB);

    if(compare_blocks(plain, decrypted, BLOCKSIZE) != 0)
      printf("** Decryption error: **\n"
             "   Decrypted ciphertext is different than the plaintext!\n");

    Zynq_crypt_simple(plain, key, 1, Zynq_cipher);
    Zynq_crypt_simple(Zynq_cipher, key, 0, Zynq_decrypted);
    print_data("Zynq cipher", Zynq_cipher, BLOCKSIZEB);
    print_data("Zynq decrypted", Zynq_decrypted, BLOCKSIZEB);

    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    if(compare_blocks(plain, Zynq_decrypted, BLOCKSIZE) != 0)
      printf("** Zynq decryption error: **\n"
             "   Decrypted ciphertext on Zynq is different than the plaintext!\n");

    for(i=0; i<99; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 100 times", cipher, BLOCKSIZEB);
    print_data("Iterated 100 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");
    for(i=0; i<900; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 1000 times", cipher, BLOCKSIZEB);
```

```
      print_data("Iterated 1000 times on Zynq", Zynq_cipher, BLOCKSIZEB);
      if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
        printf("** Zynq encryption error: **\n"
               "   Ciphertext from Zynq is different than the reference code cipher!\n");

      printf("\n");
    }

  printf("Test vectors -- set 4\n");
  printf("=====================\n\n");

  for(v=0; v<4; v++)
    {
      memset(plain, v, BLOCKSIZEB);
      memset(key, v, KEYSIZEB);

      NESSIEkeysetup(key, &subkeys);
      NESSIEencrypt(&subkeys, plain, cipher);
      Zynq_crypt_simple(plain, key, 1, Zynq_cipher);

      printf("Set 4, vector#%3d:\n", v);
      print_data("key", key, KEYSIZEB);
      print_data("plain", plain, BLOCKSIZEB);

      for(i=0; i<99999999; i++)
        {
          memset(key, cipher[BLOCKSIZEB-1], KEYSIZEB);
          NESSIEkeysetup(key, &subkeys);
          NESSIEencrypt(&subkeys, cipher, cipher);
        }
      print_data("Iterated 10^8 times", cipher, BLOCKSIZEB);
      for(i=0; i<99999999; i++)
        {
          memset(key, Zynq_cipher[BLOCKSIZEB-1], KEYSIZEB);
          Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
        }
      print_data("Iterated 10^8 times on Zynq", Zynq_cipher, BLOCKSIZEB);
      if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
        printf("** Zynq encryption error: **\n"
               "   Ciphertext from Zynq is different than the reference code cipher!\n");

      printf("\n");
    }

  printf("\n\nEnd of test vectors\n");
}

/*****************************************************************************
***********
  6. test_vectors_short: this function is identical to the previous one,
  except the 10^8 iterations stage was shortened into 10^6 iterations.
  We recommend to use this test if time is short.
*****************************************************************************
***********/
void test_vectors_short()
{
  struct NESSIEstruct subkeys;
  u8 key[KEYSIZEB];
  u8 plain[BLOCKSIZEB];
  u8 cipher[BLOCKSIZEB];
  u8 decrypted[BLOCKSIZEB];
  u32 i;
  int v;
  u8 Zynq_cipher[BLOCKSIZEB];
  u8 Zynq_decrypted[BLOCKSIZEB];

  printf("Test vectors -- set 1\n");
  printf("=====================\n\n");

  /* If key size is not a multiple of 8, this tests too much (intentionally) */
  for(v=0; v<(KEYSIZEB*8); v++)
    {
      memset(plain, 0, BLOCKSIZEB);
```

```
    memset(key, 0, KEYSIZEB);
    key[v>>3] = 1<<(7-(v&7));

    NESSIEkeysetup(key, &subkeys);
    NESSIEencrypt(&subkeys, plain, cipher);
    NESSIEdecrypt(&subkeys, cipher, decrypted);

    printf("Set 1, vector#%3d:\n", v);
    print_data("key", key, KEYSIZEB);
    print_data("plain", plain, BLOCKSIZEB);
    print_data("cipher", cipher, BLOCKSIZEB);
    print_data("decrypted", decrypted, BLOCKSIZEB);

    if(compare_blocks(plain, decrypted, BLOCKSIZE) != 0)
      printf("** Decryption error: **\n"
             "   Decrypted ciphertext is different than the plaintext!\n");

    Zynq_crypt_simple(plain, key, 1, Zynq_cipher);
    Zynq_crypt_simple(Zynq_cipher, key, 0, Zynq_decrypted);
    print_data("Zynq cipher", Zynq_cipher, BLOCKSIZEB);
    print_data("Zynq decrypted", Zynq_decrypted, BLOCKSIZEB);

    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    if(compare_blocks(plain, Zynq_decrypted, BLOCKSIZE) != 0)
      printf("** Zynq decryption error: **\n"
             "   Decrypted ciphertext on Zynq is different than the plaintext!\n");

    for(i=0; i<99; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 100 times", cipher, BLOCKSIZEB);
    print_data("Iterated 100 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");
    for(i=0; i<900; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 1000 times", cipher, BLOCKSIZEB);
    print_data("Iterated 1000 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    printf("\n");
  }

printf("Test vectors -- set 2\n");
printf("=====================\n\n");

/* If block size is not a multiple of 8, this tests too much (intentionally) */
for(v=0; v<(BLOCKSIZEB*8); v++)
  {
    memset(plain, 0, BLOCKSIZEB);
    memset(key, 0, KEYSIZEB);
    plain[v>>3] = 1<<(7-(v&7));

    NESSIEkeysetup(key, &subkeys);
    NESSIEencrypt(&subkeys, plain, cipher);
    NESSIEdecrypt(&subkeys, cipher, decrypted);

    printf("Set 2, vector#%3d:\n", v);
    print_data("key", key, KEYSIZEB);
    print_data("plain", plain, BLOCKSIZEB);
    print_data("cipher", cipher, BLOCKSIZEB);
    print_data("decrypted", decrypted, BLOCKSIZEB);
```

```
    if(compare_blocks(plain, decrypted, BLOCKSIZE) != 0)
      printf("** Decryption error: **\n"
             "   Decrypted ciphertext is different than the plaintext!\n");

    Zynq_crypt_simple(plain, key, 1, Zynq_cipher);
    Zynq_crypt_simple(Zynq_cipher, key, 0, Zynq_decrypted);
    print_data("Zynq cipher", Zynq_cipher, BLOCKSIZEB);
    print_data("Zynq decrypted", Zynq_decrypted, BLOCKSIZEB);

    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    if(compare_blocks(plain, Zynq_decrypted, BLOCKSIZE) != 0)
      printf("** Zynq decryption error: **\n"
             "   Decrypted ciphertext on Zynq is different than the plaintext!\n");

    for(i=0; i<99; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 100 times", cipher, BLOCKSIZEB);
    print_data("Iterated 100 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");
    for(i=0; i<900; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 1000 times", cipher, BLOCKSIZEB);
    print_data("Iterated 1000 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    printf("\n");
  }

printf("Test vectors -- set 3\n");
printf("=====================\n\n");

for(v=0; v<256; v++)
  {
    memset(plain, v, BLOCKSIZEB);
    memset(key, v, KEYSIZEB);

    NESSIEkeysetup(key, &subkeys);
    NESSIEencrypt(&subkeys, plain, cipher);
    NESSIEdecrypt(&subkeys, cipher, decrypted);

    printf("Set 3, vector#%3d:\n", v);
    print_data("key", key, KEYSIZEB);
    print_data("plain", plain, BLOCKSIZEB);
    print_data("cipher", cipher, BLOCKSIZEB);
    print_data("decrypted", decrypted, BLOCKSIZEB);

    if(compare_blocks(plain, decrypted, BLOCKSIZE) != 0)
      printf("** Decryption error: **\n"
             "   Decrypted ciphertext is different than the plaintext!\n");

    Zynq_crypt_simple(plain, key, 1, Zynq_cipher);
    Zynq_crypt_simple(Zynq_cipher, key, 0, Zynq_decrypted);
    print_data("Zynq cipher", Zynq_cipher, BLOCKSIZEB);
    print_data("Zynq decrypted", Zynq_decrypted, BLOCKSIZEB);

    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");
```

```
      if(compare_blocks(plain, Zynq_decrypted, BLOCKSIZE) != 0)
        printf("** Zynq decryption error: **\n"
               "   Decrypted ciphertext on Zynq is different than the plaintext!\n");

      for(i=0; i<99; i++)
      {
        NESSIEencrypt(&subkeys, cipher, cipher);
        Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
      }
      print_data("Iterated 100 times", cipher, BLOCKSIZEB);
      print_data("Iterated 100 times on Zynq", Zynq_cipher, BLOCKSIZEB);
      if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
        printf("** Zynq encryption error: **\n"
               "   Ciphertext from Zynq is different than the reference code cipher!\n");
      for(i=0; i<900; i++)
      {
        NESSIEencrypt(&subkeys, cipher, cipher);
        Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
      }
      print_data("Iterated 1000 times", cipher, BLOCKSIZEB);
      print_data("Iterated 1000 times on Zynq", Zynq_cipher, BLOCKSIZEB);
      if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
        printf("** Zynq encryption error: **\n"
               "   Ciphertext from Zynq is different than the reference code cipher!\n");

      printf("\n");
    }

  printf("Test vectors -- set 4 (shortened)\n");
  printf("=================\n\n");

  for(v=0; v<4; v++)
    {
      memset(plain, v, BLOCKSIZEB);
      memset(key, v, KEYSIZEB);

      NESSIEkeysetup(key, &subkeys);
      NESSIEencrypt(&subkeys, plain, cipher);
      Zynq_crypt_simple(plain, key, 1, Zynq_cipher);

      printf("Set 4, vector#%3d:\n", v);
      print_data("key", key, KEYSIZEB);
      print_data("plain", plain, BLOCKSIZEB);

      for(i=0; i<999999; i++)
        {
          memset(key, cipher[BLOCKSIZEB-1], KEYSIZEB);
          NESSIEkeysetup(key, &subkeys);
          NESSIEencrypt(&subkeys, cipher, cipher);
        }
      print_data("Iterated 10^6 times", cipher, BLOCKSIZEB);
      for(i=0; i<999999; i++)
        {
          memset(key, Zynq_cipher[BLOCKSIZEB-1], KEYSIZEB);
          Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
        }
      print_data("Iterated 10^6 times on Zynq", Zynq_cipher, BLOCKSIZEB);
      if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
        printf("** Zynq encryption error: **\n"
               "   Ciphertext from Zynq is different than the reference code cipher!\n");

      printf("\n");
    }

  printf("\n\nEnd of test vectors (shortened)\n");
}

/*****************************************************************************************
***********
  7. demonstration: This function is designed to demonstrate the hardware implementation
  correctness in a simple way.
  It prompts the user for key and for plaintext string data, then processes the data
  block-by-block.
```

```
  Each block is encrypted and then decrypted again. For each block the plaintext, the
  ciphertext,
  the decrypted text (all three in hexadecimal ASCII code) and the block number are printed
  on screen.
  Comparisons are made between the plaintext and decrypted text of each block, and between
  the original characters string and a decrypted re-calculated string.
  If a comparison fails, an error massage appears.
  Since the function works block-by-block and switchs between encryption and decryption,
  only ECB mode can be used for this demonstration.
*****************************************************************************************
***********/
void demonstration()
{
  u32 key1, key2, key3, key4;
  bool valid_answer, go = 1;
  u8 answer, data_string[MAX_LENGTH+1], decrypted_string[MAX_LENGTH+1], plain[BLOCKSIZEB],
  cipher[BLOCKSIZEB], decrypted[BLOCKSIZEB];
  printf("************************************************************ \n\r");
  printf("Secret key configuration: \nKHAZAD is using a 128-bit key (32 figures in
  hexadecimal base). \n");
  printf("For your convenience, the key is entered in 4 parts, each one up to 8 figures in
  hexadecimal base \n(leading zeros will be added). \n\r");
  while (go)
  {
      printf("Please enter key part number 1 \n");
      scanf("%x", &key1);
      printf("Please enter key part number 2 \n");
      scanf("%x", &key2);
      printf("Please enter key part number 3 \n");
      scanf("%x", &key3);
      printf("Please enter key part number 4 \n\r");
      scanf("%x", &key4);
      printf("key= %08X%08X%08X%08X \n", key1, key2, key3, key4);
      printf("\nPlease enter data to encrypt (up to %d characters, including spaces) \n",
      MAX_LENGTH);
      scanf(" %[^\r]s", data_string); // scanf format to allow reading spaces

      u16 i = 0, j = 0, k = 0, block_num = 0;
      while (data_string[i])
      {
          if (j < BLOCKSIZEB)
          {
              memset(plain+j, data_string[i], 1);
              j++;
          }

          if ((j != BLOCKSIZEB) && !(data_string[i+1])) // residue exists
              memset(plain+j, 0, BLOCKSIZEB-j); // zero padding

          if (((j != BLOCKSIZEB) && !(data_string[i+1])) || (j == BLOCKSIZEB)) // need to
          encrypt
          {
              if (block_num == 0)
                Zynq_crypt(plain, key1, key2, key3, key4, 0, 0, 0, 1, 0, 0, 0, cipher);   //
                first operation: only_data = 0. enc_dec = 1.
              else
                Zynq_crypt(plain, key1, key2, key3, key4, 0, 0, 1, 1, 0, 0, 0, cipher);   //
                not first operation: only_data = 1. enc_dec = 1.
              Zynq_crypt(cipher, key1, key2, key3, key4, 0, 0, 1, 0, 0, 0, 0, decrypted); //
              not first operation: only_data = 1. enc_dec = 0.
              printf("\n Text block number %u: \n", block_num);
              print_data("plaintext", plain, BLOCKSIZEB);
              print_data("ciphertext", cipher, BLOCKSIZEB);
              print_data("decrypted text", decrypted, BLOCKSIZEB);
              if(compare_blocks(plain, decrypted, BLOCKSIZE) != 0)
                printf("** Decryption error: **\n    Decrypted ciphertext is different than
                the plaintext!\n\r");
              for (k = 0; k < j; k++)
                decrypted_string[block_num*BLOCKSIZEB + k] = decrypted[k];
              block_num++;
              j = 0;
          }
          i++;
```

```
      }

      decrypted_string[i] = 0; // NULL character to end decrypted_string
      printf("\nThe original string is: %s \n\r", data_string);
      printf("The decrypted string is: %s \n\r", decrypted_string);
      if (strcmp(data_string, decrypted_string) == 0)
        printf("Texts identical! \n\r");
      else
        printf("ERROR: texts not identical! \n\r");

      valid_answer = 0;
      do {
          printf("Do you want to try this demonstration again? Please answer y/n \n\r");
          scanf(" %c", &answer);
          if ((answer == 'y') || (answer == 'Y') || (answer == '1'))
            valid_answer = 1;
          else if ((answer == 'n') || (answer == 'N') || (answer == '0'))
          {
            valid_answer = 1;
            go = 0;
            printf("Going back to the main menu... \n\r");
          }
          else
            printf("Not a valid input. \n\r");
      } while (!valid_answer);
  }
  return;
}

/*****************************************************************************************
***********
  8. application: A simple practical application.
  The user can choose between encryption or decryption,
  then he's prompted for a key, for operation mode and for initialization vector (if CBC
  mode was chosen).
  For encryption, a plaintext string is entered,
  and the ciphertext is calculated and printed on screen as hexadecimal figures.
  For decryption, a ciphertext hexadecimal figures string is entered,
  and the plaintext is calculated and printed on screen as regular characters.
*****************************************************************************************
***********/
void application()
{
  u32 key1, key2, key3, key4, IV1, IV2;
  bool enc_dec, only_data, op_mode, first_block, new_IV, valid_answer, key_entered,
  IV_entered, first_run = 1, go = 1;
  u8 answer, data_in_string[MAX_LENGTH+1], cipher_in_string[MAX_LENGTH+1],
  block_in[BLOCKSIZEB], block_out[BLOCKSIZEB+1];
  printf("************************************************************ \n\r");
  while (go)
  {
    key_entered = 0;
    IV_entered = 0;
    valid_answer = 0;
    do {
          printf("For encryption, please press 'e'.  For decryption, please press 'd'. \n\r");
          scanf(" %c", &answer);
          if ((answer == 'e') || (answer == 'E'))
          {
            valid_answer = 1;
            enc_dec = 1;  // =encryption
            printf("Encryption mode \n\r");
          }
          else if ((answer == 'd') || (answer == 'D'))
          {
            valid_answer = 1;
            enc_dec = 0;  // =decryption
            printf("Decryption mode \n\r");
          }
          else
            printf("Not a valid input. \n\r");
          } while (!valid_answer);
```

```
  valid_answer = 0;
  do {
      printf("For ECB mode, please press '0'.  For CBC mode, please press '1'. \n\r");
      scanf(" %c", &answer);
      if (answer == '0')
      {
        valid_answer = 1;
        op_mode = 0;
        printf("ECB mode \n\r");
      }
      else if (answer == '1')
      {
        valid_answer = 1;
        op_mode = 1;
        printf("CBC mode \n\r");
      }
      else
        printf("Not a valid input. \n\r");
  } while (!valid_answer);

if (!first_run)
{
    valid_answer = 0;
    do {
        printf("Do you want to enter a new key? Please answer y/n \n\r");
        scanf(" %c", &answer);
        if ((answer == 'y') || (answer == 'Y') || (answer == '1'))
            valid_answer = 1;
        else if ((answer == 'n') || (answer == 'N') || (answer == '0'))
            valid_answer = 1;
        else
            printf("Not a valid input. \n\r");
    } while (!valid_answer);
}

if ((first_run) || ((!first_run) && ((answer == 'y') || (answer == 'Y') || (answer ==
'1'))))
{
    printf("Secret key configuration: \nKHAZAD is using a 128-bit key (32 figures in
    hexadecimal base). \n");
    printf("For your convenience, the key is entered in 4 parts, each one up to 8
    figures in hexadecimal base \n(leading zeros will be added). \n\r");
    printf("Please enter key part number 1 \n");
    scanf("%x", &key1);
    printf("Please enter key part number 2 \n");
    scanf("%x", &key2);
    printf("Please enter key part number 3 \n");
    scanf("%x", &key3);
    printf("Please enter key part number 4 \n\r");
    scanf("%x", &key4);
    key_entered = 1;
}
printf("key= %08X%08X%08X%08X \n\r", key1, key2, key3, key4);

if ((!first_run) && (op_mode == 1))
{
    valid_answer = 0;
    do {
        printf("Do you want to enter an IV? Please answer y/n \n\r");
        scanf(" %c", &answer);
        if ((answer == 'y') || (answer == 'Y') || (answer == '1'))
            valid_answer = 1;
        else if ((answer == 'n') || (answer == 'N') || (answer == '0'))
            valid_answer = 1;
        else
            printf("Not a valid input. \n\r");
    } while (!valid_answer);
}

if ((op_mode == 1) && ((first_run) || ((!first_run) && ((answer == 'y') || (answer ==
'Y') || (answer == '1')))))
{
    printf("Initialization Vector: \n");
```

```
    printf("For your convenience, the IV is entered in 2 parts, each one up to 8 figures
    in hexadecimal base \n(leading zeros will be added). \n\r");
    printf("Please enter IV part number 1 \n");
    scanf("%x", &IV1);
    printf("Please enter IV part number 2 \n");
    scanf("%x", &IV2);
    IV_entered = 1;
}

if (op_mode == 1)
    printf("IV= %08X%08X \n", IV1, IV2);

u16 i = 0;
if (enc_dec == 1)  // plaintext input: use the input as is
{
    printf("\nPlease enter data to encrypt (up to %d characters, including spaces) \n",
    MAX_LENGTH);
    scanf(" %[^\r]s", data_in_string);  // scanf format to allow reading spaces
    printf("\n\t the ciphertext: \n");
}
else               // ciphertext input: convert the input from hexadecimal figures
string to characters string
{
    printf("\nPlease enter data to decrypt - a sequence of hexadecimal figures pairs (up
    to %d characters) \n", MAX_LENGTH);
    scanf("%s", cipher_in_string);
    while (cipher_in_string[i])
    {
        char temp[] = {cipher_in_string[i],cipher_in_string[i+1]};  // each hexadecimal
        figures pair is the ASCII code of one character
        data_in_string[i/2] = (u8)strtol(temp, NULL, 16);           // convert
        hexadecimal string to long int to u8
        i+=2;
    }
    data_in_string[i/2] = 0;  // NULL character to end the string
    printf("\n\t the plaintext: \n");
}

block_out[BLOCKSIZEB] = 0;    // NULL character to end the string
i = 0;
u16 j = 0, k = 0, block_num = 0;
while (data_in_string[i])
{
    if (j < BLOCKSIZEB)
    {
        memset(block_in+j, data_in_string[i], 1);
        j++;
    }

    if ((j != BLOCKSIZEB) && !(data_in_string[i+1])) // residue exists
        memset(block_in+j, 0, BLOCKSIZEB-j); // zero padding

    if (((j != BLOCKSIZEB) && !(data_in_string[i+1])) || (j == BLOCKSIZEB)) // need to
    encrypt/decrypt
    {
        if (block_num == 0)
            first_block = 1;
        else
            first_block = 0;

        if ((block_num == 0) && (key_entered == 1))
            only_data = 0;
        else
            only_data = 1;

        if ((block_num == 0) && (IV_entered == 1))
            new_IV = 1;
        else
            new_IV = 0;

        Zynq_crypt(block_in, key1, key2, key3, key4, IV1, IV2, only_data, enc_dec,
        op_mode, first_block, new_IV, block_out);
```

```
            if (enc_dec == 1)
                for (k=0; k < BLOCKSIZEB; k++) // ciphertext output: print as hexadecimal
                figures
                {
                    putchar(hex[(block_out[k]>>4)&0xF]);
                    putchar(hex[(block_out[k]   )&0xF]);
                }
            else
                printf("%s", block_out);      // plaintext output: print as characters

            block_num++;
            j = 0;
        }
        i++;
    }

    first_run = 0;
    valid_answer = 0;
    do {
        printf("\n\nDo you want to use this application again? Please answer y/n \n\r");
        scanf(" %c", &answer);
        if ((answer == 'y') || (answer == 'Y') || (answer == '1'))
            valid_answer = 1;
        else if ((answer == 'n') || (answer == 'N') || (answer == '0'))
        {
            valid_answer = 1;
            go = 0;
            printf("Going back to the main menu... \n\r");
        }
        else
            printf("Not a valid input. \n\r");
       } while (!valid_answer);
  }
  return;
}

/*****************************************************************************************
***********
  9. print_data: print a given string "str", then "=", then the ASCII code in hexadecimal
  figures of
  each element in a given u8 array.
  From the reference code bctestvectors.c file.
*****************************************************************************************
***********/
void print_data(char *str, u8 *val, int len) // from reference code bctestvectors.c file
{
  int i;

  printf("%25s=", str);
  for(i=0; i<len; i++)
    {
      putchar(hex[(val[i]>>4)&0xF]);
      putchar(hex[(val[i]   )&0xF]);
    }
  putchar('\n');
}

/*****************************************************************************************
***********
  10. compare_blocks: compare two u8 strings with the same given length.
  From the reference code bctestvectors.c file.
*****************************************************************************************
***********/
int compare_blocks(u8 *m1, u8 *m2, int len_bits)
{
  int i;
  int lenb=(len_bits+7)>>3;
  int mask0 = (1<<(((len_bits-1)&7)+1))-1;

  if((m1[0]&mask0) != (m2[0]&mask0))
    return 1;

  for(i=1; i<lenb; i++)
```

```
    if(m1[i] != m2[i])
        return 1;

  return 0;
}

/*****************************************************************************************
***********
  11. about: print information about this project.
  When the design operates on bare-metal, there is no filesystem, so no text file can be used.
*****************************************************************************************
***********/
void about()
{

    printf("*****************************************************************************
    ***********************\n");

    printf("*****************************************************************************
    ***********************\n");
    printf("Zynq-7000 based Implementation of the KHAZAD Block Cipher\n");
    printf("Yossef Shitzer & Efraim Wasserman\n");
    printf("Jerusalem College of Technology - Lev Academic Center (JCT)\n");
    printf("Department of electrical and electronic engineering\n");
    printf("2018\n\r");

    printf("''The KHAZAD Legacy-Level Block Cipher'' is a block cipher designed by Paulo
    S.L.M. Barreto and Vincent Rijmen.\n");
    printf("It uses a 128-bit key, operates on 64-bit data blocks, and comprises 8
    rounds.\n");
    printf("The algorithm has been submitted as a candidate for the first open NESSIE
    workshop in 2000.\n");
    printf("This first version now considered obsolete. For phase 2 of NESSIE, a modified
    version has been submitted, \n");
    printf("named ''Khazad-tweak'', and has been accepted as NESSIE finalist.\n");
    printf("This version can be found here:\n");
    printf("https://www.cosic.esat.kuleuven.be/nessie/tweaks.html\n\r");

    printf("The algorithm developers wrote: \n");
    printf("''Khazad is named after Khazad-dum, ''the Mansion of the Khazad'', which in the
    tongue of the Dwarves is \n");
    printf("the name of the great realm and city of Dwarrowdelf, of the haunted mithril
    mines in Moria, the Black Chasm.\n");
    printf("But all this should be quite obvious - unless you haven't read J.R.R. Tolkien's
    ''The Lord of the Rings'', of course :-)  ''\n\r");

    printf("This hardware implementation of KHAZAD uses the MicroZed 7010 development board
    by Avnet Inc., \n");
    printf("which is based on a Xilinx Zynq-7010 All Programmable SoC.\n");
    printf("The Zynq Z-7010 device integrates a dual-core ARM Cortex A9 processor with an
    Artix-7 FPGA.\n");
    printf("This new concept allows many interesting and exciting possibilities.\n");
    printf("In our design, the programmable logic (PL) is used for implementing the
    algorithm, in ECB mode or CBC mode, \n");
    printf("and the processing system (PS) is used mainly for dealing with user input &
    output operations.\n");
    printf("The PL design files were written in Verilog. Synthesis and Implementation were
    done using Xilinx Vivado.\n");
    printf("The PS program was written in C, and compiled using Xilinx SDK.\n\r");

    printf("The MicroZed development board can be used as both a stand-alone board, \n");
    printf("or combined with a carrier card as an embeddable system-on-module.\n");
    printf("This implementation was designed to be fully operational even when using the
    stand-alone mode.\n");
    printf("Plugging the board into the carrier card will activate more indicator LEDs.\n\r");

    printf("This project was created as a final year project, with the guidance of Mr. Uri
    Stroh.\n");
    printf("We want to thank Mr. Stroh for his guidance and help, \n");
    printf("the Lev Academic Center (JCT) staff for supplying equipment and technical
    support, \n");
    printf("and the Xilinx and Avnet companies for their fine products, useful documentation
    and helpful websites.\n\r");
```

```
}

#endif
```

```
/*****************************************************************************************************
***********
*****************************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
*****************************************************************************************************
***********
*****************************************************************************************************
***********
Main program file
*****************************************************************************************************
***********
*****************************************************************************************************
***********
Version 2.0: ECB+CBC implementation
*****************************************************************************************************
***********
*****************************************************************************************************
***********/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h> // for usleep
#include "platform.h"
#include "xgpio.h"
#include "xgpiops.h"
#include "Xscugic.h"
#include "xil_exception.h"
#include "nessie_modified.h"
#include "KHAZAD_Zynq.h"

int main()
{
  init_platform(); // register initialization and UART setup
  printf("************************************************************ \n");
  printf("************************************************************ \n\r");
  printf("KHAZAD block cipher hardware implementation using the Zynq-7000 device and the
  MicroZed development board \n\r");
  printf("Yossef Shitzer & Efraim Wasserman \n\r");
  printf("Jerusalem College of Technology - Lev Academic Center (JCT) \n\r");
  printf("Department of electrical and electronic engineering \n\r");
  printf("2018 \n\r");
  printf("************************************************************ \n");
  printf("************************************************************ \n\r");

  // Peripherals and interrupt configuration:
  board_configuration();

  printf("AXI_GPIO0 address = 0x%x \n\r", ADDR0);
  printf("AXI_GPIO1 address = 0x%x \n\r", ADDR1);
  printf("AXI_GPIO2 address = 0x%x \n\r", ADDR2);
  printf("AXI_GPIO3 address = 0x%x \n\r", ADDR3);
  printf("AXI_GPIO4 address = 0x%x \n\r", ADDR4);
  printf("AXI_GPIO address offset = 0x%08x \n\r", ADDR_offset);

  // PL design initialization:
  /* ctrl is received in the PL as signal ctrl_from_PS[5:0]:
     bit 5 - RST
     bit 4 - only_data.                                          1: new data, same key.
     0: new data and new key.
     bit 3 - enc_dec: the desired operation.                     1: encryption. 0:
     decryption.
     bit 2 - op_mode: the desired cryptographic mode of operation.  1: CBC. 0: ECB.
     bit 1 - first_block: flag for the CBC mode.                 1: first data block. 0:
     not first data block.
     bit 0 - bistable start/ready semaphore flag. To run an operation this bit must not be
     equal to a matching flag in the PL. */
  XGpioPs_WritePin(&my_Gpio, 47, 0); // turn off the PS-ready indicator LED
```

```
  ctrl = 0x0020; // reset=1
  Xil_Out16(ADDR0,ctrl); // send from PS to FPGA via AXI interface
  usleep(50000);
  ctrl = 0x0000; // reset=0
  Xil_Out16(ADDR0,ctrl);
  XGpioPs_WritePin(&my_Gpio, 47, 1); // turn on the LED

  printf("****************************************************************** \n\r");
  printf("Welcome to the KHAZAD block cipher hardware implementation! \n\r");
  bool go = 1;
  u8 option;
  while (go) {
      printf("****************************************************************** \n\n\r");
      printf("Please choose an option: \n\n\r");
      printf("--1-- \t User application \n\r");
      printf("--2-- \t Simple ECB correctness demonstration \n\r");
      printf("--3-- \t Test vectors - full version. Warning: this test may take a long time
      \n\r");
      printf("--4-- \t Test vectors - short version \n\r");
      printf("--5-- \t About \n\r");
      printf("--6-- \t Exit \n\r");
      printf("To reset the FPGA design, you may press the MicroZed user button at any time.
      \n");
      printf("(Resetting the design while mid-operation may lead to wrong results.) \n\r");
      scanf(" %c", &option);

      switch(option) {
      case '1':
          application();
          break;
      case '2':
          demonstration();
          break;
      case '3':
          test_vectors_full();
          break;
      case '4':
          test_vectors_short();
          break;
      case '5':
          about();
          break;
      case '6':
          go = 0;
          printf("Thanks for using this design. Goodbye! \n\r");
          break;
      default:
          printf("Not a valid input. \n\r");
      }
  }

  cleanup_platform(); // cleanup, disable cache
  return 0;
}
```

```
#ifndef KHAZAD_ZYNQ_H
#define KHAZAD_ZYNQ_H
/*****************************************************************************
***********
*****************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
*****************************************************************************
***********
*****************************************************************************
***********
This is the KHAZAD_Zynq library header file, containing the functions and definitions
to run the various applications of the design. The file defines these functions:
1.  USR_button_ISR
2.  board_configuration
3.  Zynq_crypt
4.  Zynq_crypt_simple
5.  test_vectors_full
6.  test_vectors_short
7.  demonstration
8.  hardware_implementation
9.  print_data
10. compare_blocks
11. about
Identical lines of code may appear in some of functions. This was done to ease the re-use of
the functions
as standalone programs.
*****************************************************************************
***********
*****************************************************************************
***********
Version 1.0: ECB implementation
*****************************************************************************
***********
*****************************************************************************
***********/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include "platform.h"
#include "xgpio.h"
#include "xgpiops.h"
#include "xparameters.h"
#include "xil_types.h"
#include "xstatus.h"
#include "Xscugic.h"
#include "xil_exception.h"
#include "nessie_modified.h"
#include "khazad-tweak32.h"

/*****************************************************************************
***********
*****************************************************************************
***********
  Globals definitions
*****************************************************************************
***********
*****************************************************************************
***********/

// AXI_GPIO modules registers adresses:
#define ADDR0 ((unsigned int)0x41200000)
#define ADDR1 ((unsigned int)0x41210000)
#define ADDR2 ((unsigned int)0x41220000)
#define ADDR3 ((unsigned int)0x41230000)
#define ADDR_offset ((unsigned int)0x00000008)
```

```
// maximal length of input strings, in number of characters, including spaces:
#define MAX_LENGTH  400

/************************************************************************************************
************
************************************************************************************************
************
  Global variables declarations
************************************************************************************************
************
************************************************************************************************
************/

static XGpioPs my_Gpio; // for the PS
static XGpio GPIO_3;    // for the bidirectional module AXI_GPIO_3
static XScuGic my_Gic;  // for GIC: general interrupt controller
static u16 ctrl;        // for sending instructions to the PL design. Further details in
main.c file.
static char *hex = "0123456789ABCDEF"; // for base conversion functions

/************************************************************************************************
************
************************************************************************************************
************
  Functions definitions
************************************************************************************************
************
************************************************************************************************
************/

/************************************************************************************************
************
  1. USR_button_ISR: the user button Interrupt Service Routine.
  The function communicates with the PL fabric. It makes use of the static variable ctrl
  and configure it to give the appropriate instructions to the design.
************************************************************************************************
************/
static void USR_button_ISR(void *CallBackReff)
{
  XGpioPs_IntrDisablePin(&my_Gpio, 51); // for debouncing the switch
  XGpioPs_IntrClearPin(&my_Gpio, 51);   // clear the interrupt flag
  XGpioPs_WritePin(&my_Gpio, 47, 0);    // turn off the PS-ready indicator LED
  ctrl = 0x0008;           // reset=1
  Xil_Out16(ADDR0,ctrl);  // send from PS to FPGA via AXI interface
  printf("\t\tReset was asserted!\n\r");
  sleep(1);                // for debouncing the switch
  ctrl = 0x0000;           // reset=0
  Xil_Out16(ADDR0,ctrl);
  XGpioPs_WritePin(&my_Gpio, 47, 1);    // turn on the LED
  XGpioPs_IntrEnablePin(&my_Gpio, 51);
}

/************************************************************************************************
************
  2. board_configuration: configures the PS, the peripherals, the bidirectional AXI_GPIO and
  the interrupt.
************************************************************************************************
************/
void board_configuration()
{
  XGpioPs_Config *GPIO_Config;
  GPIO_Config = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
  u16 status;
  status = XGpioPs_CfgInitialize(&my_Gpio, GPIO_Config, GPIO_Config->BaseAddr);
  if (status == XST_SUCCESS)
    printf("XGpioPs configuration successful! \n\r");
  else
    printf("XGpioPs configuration failed! \n\r");
  XGpioPs_SetDirectionPin(&my_Gpio, 47, 1); // board LED, output
  XGpioPs_SetOutputEnablePin(&my_Gpio, 47, 1);
  XGpioPs_SetDirectionPin(&my_Gpio, 51, 0); // USR button, input
  XGpioPs_SetDirectionPin(&my_Gpio, 54, 0); // EMIO pin, input
  status = XGpio_Initialize(&GPIO_3, XPAR_AXI_GPIO_3_DEVICE_ID); // bidirectional AXI_GPIO
```

```
  module
  if (status == XST_SUCCESS)
    printf("AXI_GPIO configuration successful! \n\r");
  else
    printf("AXI_GPIO configuration failed! \n\r");
  // Interrupt configuration:
  XScuGic_Config *Gic_Config;
  Gic_Config = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
  status = XScuGic_CfgInitialize(&my_Gic, Gic_Config, Gic_Config->CpuBaseAddress);
  if (status == XST_SUCCESS)
    printf("GIC configuration successful! \n\r");
  else
    printf("GIC configuration failed! \n\r");
  Xil_ExceptionInit(); // initialize exception handlers on the processor
  Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
  (Xil_ExceptionHandler)XScuGic_InterruptHandler, &my_Gic);
  status = XScuGic_Connect(&my_Gic, XPS_GPIO_INT_ID, (Xil_ExceptionHandler)USR_button_ISR,
  (void *)&my_Gic);
  if (status == XST_SUCCESS)
    printf("GPIO interrupt handler connection successful! \n\r");
  else
    printf("GPIO interrupt handler connection failed! \n\r");
  XGpioPs_SetIntrTypePin(&my_Gpio, 51, XGPIOPS_IRQ_TYPE_EDGE_RISING); // interrupt on rising
  edge
  XGpioPs_IntrClearPin(&my_Gpio, 51);  // clear any pending residual USR_button interrupts
  XGpioPs_IntrEnablePin(&my_Gpio, 51); // enable USR_button interrupt
  XScuGic_Enable(&my_Gic, XPS_GPIO_INT_ID); // enable GPIO interrupt (SPI: Shared Peripheral
  Interrupt)
  Xil_ExceptionEnable(); // enable interrupt handling
  XGpioPs_WritePin(&my_Gpio, 47, 1); // turn on the PS-ready indicator LED
}

/*****************************************************************************************
***********
  3. Zynq_crypt: the main function that communicates with the PL fabric and get it to execute
  a "crypt" operation: encryption or decryption. The function makes use of the static
  variable ctrl
  and configure it to give the appropriate instructions to the design.
  Function parameters:
  text: pointer to u8 data in array.
  key1, key2, key3, key4: four u32 parts of the key.
  enc_dec: flag to the desired operation. enc_dec = 1: encryption, enc_dec = 0: decryption.
  only_data: indicates the key period. only_data = 1: new data, same key. only_data = 0: new
  data and new key.
  Function returns:
  result: pointer to u8 data out array.
*****************************************************************************************
***********/
void Zynq_crypt(const u8 * const text, const u32 key1, const u32 key2, const u32 key3, const
u32 key4, const bool enc_dec, const bool only_data, u8 * const result)
{
  XGpioPs_WritePin(&my_Gpio, 47, 0); // turn off the PS-ready indicator LED
  // map u8-array text to two u32 d_in parts:
  u32 d_in_1 =
        ((u32)text[0] << 24) ^
        ((u32)text[1] << 16) ^
        ((u32)text[2] <<  8) ^
        ((u32)text[3]      );

  u32 d_in_2 =
        ((u32)text[4] << 24) ^
        ((u32)text[5] << 16) ^
        ((u32)text[6] <<  8) ^
        ((u32)text[7]      );

  if (!only_data)
  {
      // send key:
      Xil_Out32(ADDR1,key1)              ;
      Xil_Out32(ADDR1 + ADDR_offset,key2);
      Xil_Out32(ADDR2,key3)              ;
      Xil_Out32(ADDR2 + ADDR_offset,key4);
  }
```

```
  // set GPIO_3 to output and send data:
  XGpio_SetDataDirection(&GPIO_3,1,0x00000000);
  Xil_Out32(ADDR3              ,d_in_1);
  Xil_Out32(ADDR3 + ADDR_offset,d_in_2);

  // ctrl setting:
  ctrl = ctrl ^ 0x0001;   // toggle bit 0, to issue a start command
  if (enc_dec == 1)
    ctrl = ctrl | 0x0002; // = ctrl|0010, turn on bit 1
  else
    ctrl = ctrl & 0xFFFD; // = ctrl&1101, turn off bit 1
  if (only_data == 1)
    ctrl = ctrl | 0x0004; // = ctrl|0100, turn on bit 2
  else
    ctrl = ctrl & 0xFFFB; // = ctrl&1011, turn off bit 2

  Xil_Out16(ADDR0,ctrl);
  // can't send the start command before both key & data sent because key schedule will end
  before data has arrived,
  // and additional flags, and time to send them, will be needed.

  u16 finish;
  // set GPIO_3 to input and wait for data:
  XGpio_SetDataDirection(&GPIO_3,1,0xFFFFFFFF);
  do {
    finish = XGpioPs_ReadPin(&my_Gpio, 54); // read from FPGA via EMIO interface (polling)
    } while ((finish ^ ctrl) & 0x0001); // while LSB of finish != LSB of ctrl

  // read data from PL via AXI bus:
  u32 d_out_1 = Xil_In32(ADDR3);
  u32 d_out_2 = Xil_In32(ADDR3 + ADDR_offset);

  // map two u32 d_out parts to u8-array text:
  result[0] = (u8)(d_out_1 >> 24);
  result[1] = (u8)(d_out_1 >> 16);
  result[2] = (u8)(d_out_1 >>  8);
  result[3] = (u8)(d_out_1      );
  result[4] = (u8)(d_out_2 >> 24);
  result[5] = (u8)(d_out_2 >> 16);
  result[6] = (u8)(d_out_2 >>  8);
  result[7] = (u8)(d_out_2      );

  XGpioPs_WritePin(&my_Gpio, 47, 1); // turn on the LED
}

/*****************************************************************************************
***********
  4. Zynq_crypt_simple: this function is very similar to Zynq_crypt, just a little more
  simple,
  for use in the test vectors operations.
  The key is given as one parameter, a pointer to u8 array - reflecting the way it's
  calculated in the test vectors files,
  and mapping to four u32 key parts is done inside the function. Also the function lacks the
  only_data option.
  One can easily create different versions of this function for other kinds of tests.
*****************************************************************************************
***********/
void Zynq_crypt_simple(const u8 * const text, const u8 * const key, const bool enc_dec, u8 *
const result)
{
  XGpioPs_WritePin(&my_Gpio, 47, 0); // turn off the PS-ready indicator LED
  // map u8-array text to two u32 d_in parts:
  u32 d_in_1 =
        ((u32)text[0] << 24) ^
        ((u32)text[1] << 16) ^
        ((u32)text[2] <<  8) ^
        ((u32)text[3]      );

  u32 d_in_2 =
        ((u32)text[4] << 24) ^
        ((u32)text[5] << 16) ^
        ((u32)text[6] <<  8) ^
        ((u32)text[7]      );
```

```
  // map u8-array key to four u32 key parts:
  u32 key1 =
        ((u32)key[ 0] << 24) ^
        ((u32)key[ 1] << 16) ^
        ((u32)key[ 2] <<  8) ^
        ((u32)key[ 3]      );

  u32 key2 =
        ((u32)key[ 4] << 24) ^
        ((u32)key[ 5] << 16) ^
        ((u32)key[ 6] <<  8) ^
        ((u32)key[ 7]      );

  u32 key3 =
        ((u32)key[ 8] << 24) ^
        ((u32)key[ 9] << 16) ^
        ((u32)key[10] <<  8) ^
        ((u32)key[11]      );

  u32 key4 =
        ((u32)key[12] << 24) ^
        ((u32)key[13] << 16) ^
        ((u32)key[14] <<  8) ^
        ((u32)key[15]      );

  // send key:
  Xil_Out32(ADDR1,key1)                  ;
  Xil_Out32(ADDR1 + ADDR_offset,key2)    ;
  Xil_Out32(ADDR2,key3)                  ;
  Xil_Out32(ADDR2 + ADDR_offset,key4)    ;
  // set GPIO_3 to output and send data:
  XGpio_SetDataDirection(&GPIO_3,1,0x00000000);
  Xil_Out32(ADDR3              ,d_in_1) ;
  Xil_Out32(ADDR3 + ADDR_offset,d_in_2) ;

  // ctrl setting:
  ctrl = ctrl ^ 0x0001;   // toggle bit 0, to issue a start command
  if (enc_dec == 1)
    ctrl = ctrl | 0x0002; // = ctrl|0010, turn on bit 1
  else
    ctrl = ctrl & 0xFFFD; // = ctrl&1101, turn off bit 1
  ctrl = ctrl & 0xFFFB;   // = ctrl&1011, turn off bit 2, only_data isn't active

  Xil_Out16(ADDR0,ctrl);
  // can't send the start command before both key & data sent because key schedule will end
  before data has arrived,
  // and additional flags, and time to send them, will be needed.

  u16 finish;
  // set GPIO_3 to input and wait for data:
  XGpio_SetDataDirection(&GPIO_3,1,0xFFFFFFFF);
  do {
    finish = XGpioPs_ReadPin(&my_Gpio, 54); // read from FPGA via EMIO interface (polling)
    } while ((finish ^ ctrl) & 0x0001); // while LSB of finish != LSB of ctrl

  // read data from PL via AXI bus:
  u32 d_out_1 = Xil_In32(ADDR3);
  u32 d_out_2 = Xil_In32(ADDR3 + ADDR_offset);

  // map two u32 d_out parts to u8-array text:
  result[0] = (u8)(d_out_1 >> 24);
  result[1] = (u8)(d_out_1 >> 16);
  result[2] = (u8)(d_out_1 >>  8);
  result[3] = (u8)(d_out_1      );
  result[4] = (u8)(d_out_2 >> 24);
  result[5] = (u8)(d_out_2 >> 16);
  result[6] = (u8)(d_out_2 >>  8);
  result[7] = (u8)(d_out_2      );

  XGpioPs_WritePin(&my_Gpio, 47, 1); // turn on the LED
}
```

```
/*****************************************************************************************
***********
  5. test_vectors_full: this function is essentially the original reference code
  bctestvectors.c code,
  combined with encryption and decryption on Zynq, to test the hardware implementation
  results.
  If these results are different than the reference code results, an error massage appears.
  This test includes a stage of 10^8 iterations, so it may take some time to be completed.
*****************************************************************************************
***********/
void test_vectors_full()
{
  struct NESSIEstruct subkeys;
  u8 key[KEYSIZEB];
  u8 plain[BLOCKSIZEB];
  u8 cipher[BLOCKSIZEB];
  u8 decrypted[BLOCKSIZEB];
  u32 i;
  int v;
  u8 Zynq_cipher[BLOCKSIZEB];
  u8 Zynq_decrypted[BLOCKSIZEB];

  printf("Test vectors -- set 1\n");
  printf("=====================\n\n");

  /* If key size is not a multiple of 8, this tests too much (intentionally) */
  for(v=0; v<(KEYSIZEB*8); v++)
    {
      memset(plain, 0, BLOCKSIZEB);
      memset(key, 0, KEYSIZEB);
      key[v>>3] = 1<<(7-(v&7));

      NESSIEkeysetup(key, &subkeys);
      NESSIEencrypt(&subkeys, plain, cipher);
      NESSIEdecrypt(&subkeys, cipher, decrypted);

      printf("Set 1, vector#%3d:\n", v);
      print_data("key", key, KEYSIZEB);
      print_data("plain", plain, BLOCKSIZEB);
      print_data("cipher", cipher, BLOCKSIZEB);
      print_data("decrypted", decrypted, BLOCKSIZEB);

      if(compare_blocks(plain, decrypted, BLOCKSIZE) != 0)
        printf("** Decryption error: **\n"
               "   Decrypted ciphertext is different than the plaintext!\n");

      Zynq_crypt_simple(plain, key, 1, Zynq_cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 0, Zynq_decrypted);
      print_data("Zynq cipher", Zynq_cipher, BLOCKSIZEB);
      print_data("Zynq decrypted", Zynq_decrypted, BLOCKSIZEB);

      if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
        printf("** Zynq encryption error: **\n"
               "   Ciphertext from Zynq is different than the reference code cipher!\n");

      if(compare_blocks(plain, Zynq_decrypted, BLOCKSIZE) != 0)
        printf("** Zynq decryption error: **\n"
               "   Decrypted ciphertext on Zynq is different than the plaintext!\n");

      for(i=0; i<99; i++)
      {
        NESSIEencrypt(&subkeys, cipher, cipher);
        Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
      }
      print_data("Iterated 100 times", cipher, BLOCKSIZEB);
      print_data("Iterated 100 times on Zynq", Zynq_cipher, BLOCKSIZEB);
      if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
        printf("** Zynq encryption error: **\n"
               "   Ciphertext from Zynq is different than the reference code cipher!\n");
      for(i=0; i<900; i++)
      {
        NESSIEencrypt(&subkeys, cipher, cipher);
        Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
```

```
    }
    print_data("Iterated 1000 times", cipher, BLOCKSIZEB);
    print_data("Iterated 1000 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    printf("\n");
  }

printf("Test vectors -- set 2\n");
printf("=====================\n\n");

/* If block size is not a multiple of 8, this tests too much (intentionally) */
for(v=0; v<(BLOCKSIZEB*8); v++)
  {
    memset(plain, 0, BLOCKSIZEB);
    memset(key, 0, KEYSIZEB);
    plain[v>>3] = 1<<(7-(v&7));

    NESSIEkeysetup(key, &subkeys);
    NESSIEencrypt(&subkeys, plain, cipher);
    NESSIEdecrypt(&subkeys, cipher, decrypted);

    printf("Set 2, vector#%3d:\n", v);
    print_data("key", key, KEYSIZEB);
    print_data("plain", plain, BLOCKSIZEB);
    print_data("cipher", cipher, BLOCKSIZEB);
    print_data("decrypted", decrypted, BLOCKSIZEB);

    if(compare_blocks(plain, decrypted, BLOCKSIZE) != 0)
      printf("** Decryption error: **\n"
             "   Decrypted ciphertext is different than the plaintext!\n");

    Zynq_crypt_simple(plain, key, 1, Zynq_cipher);
    Zynq_crypt_simple(Zynq_cipher, key, 0, Zynq_decrypted);
    print_data("Zynq cipher", Zynq_cipher, BLOCKSIZEB);
    print_data("Zynq decrypted", Zynq_decrypted, BLOCKSIZEB);

    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    if(compare_blocks(plain, Zynq_decrypted, BLOCKSIZE) != 0)
      printf("** Zynq decryption error: **\n"
             "   Decrypted ciphertext on Zynq is different than the plaintext!\n");

    for(i=0; i<99; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 100 times", cipher, BLOCKSIZEB);
    print_data("Iterated 100 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");
    for(i=0; i<900; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 1000 times", cipher, BLOCKSIZEB);
    print_data("Iterated 1000 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    printf("\n");
  }

printf("Test vectors -- set 3\n");
printf("=====================\n\n");
```

```
for(v=0; v<256; v++)
  {
    memset(plain, v, BLOCKSIZEB);
    memset(key, v, KEYSIZEB);

    NESSIEkeysetup(key, &subkeys);
    NESSIEencrypt(&subkeys, plain, cipher);
    NESSIEdecrypt(&subkeys, cipher, decrypted);

    printf("Set 3, vector#%3d:\n", v);
    print_data("key", key, KEYSIZEB);
    print_data("plain", plain, BLOCKSIZEB);
    print_data("cipher", cipher, BLOCKSIZEB);
    print_data("decrypted", decrypted, BLOCKSIZEB);

    if(compare_blocks(plain, decrypted, BLOCKSIZE) != 0)
      printf("** Decryption error: **\n"
             "   Decrypted ciphertext is different than the plaintext!\n");

    Zynq_crypt_simple(plain, key, 1, Zynq_cipher);
    Zynq_crypt_simple(Zynq_cipher, key, 0, Zynq_decrypted);
    print_data("Zynq cipher", Zynq_cipher, BLOCKSIZEB);
    print_data("Zynq decrypted", Zynq_decrypted, BLOCKSIZEB);

    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    if(compare_blocks(plain, Zynq_decrypted, BLOCKSIZE) != 0)
      printf("** Zynq decryption error: **\n"
             "   Decrypted ciphertext on Zynq is different than the plaintext!\n");

    for(i=0; i<99; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 100 times", cipher, BLOCKSIZEB);
    print_data("Iterated 100 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");
    for(i=0; i<900; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 1000 times", cipher, BLOCKSIZEB);
    print_data("Iterated 1000 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    printf("\n");
  }

printf("Test vectors -- set 4\n");
printf("=====================\n\n");

for(v=0; v<4; v++)
  {
    memset(plain, v, BLOCKSIZEB);
    memset(key, v, KEYSIZEB);

    NESSIEkeysetup(key, &subkeys);
    NESSIEencrypt(&subkeys, plain, cipher);
    Zynq_crypt_simple(plain, key, 1, Zynq_cipher);

    printf("Set 4, vector#%3d:\n", v);
    print_data("key", key, KEYSIZEB);
    print_data("plain", plain, BLOCKSIZEB);
```

```
      for(i=0; i<99999999; i++)
        {
          memset(key, cipher[BLOCKSIZEB-1], KEYSIZEB);
          NESSIEkeysetup(key, &subkeys);
          NESSIEencrypt(&subkeys, cipher, cipher);
        }
      print_data("Iterated 10^8 times", cipher, BLOCKSIZEB);
      for(i=0; i<99999999; i++)
        {
          memset(key, Zynq_cipher[BLOCKSIZEB-1], KEYSIZEB);
          Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
        }
      print_data("Iterated 10^8 times on Zynq", Zynq_cipher, BLOCKSIZEB);
      if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
        printf("** Zynq encryption error: **\n"
               "   Ciphertext from Zynq is different than the reference code cipher!\n");

      printf("\n");
    }

  printf("\n\nEnd of test vectors\n");
}

/*****************************************************************************************
***********
  6. test_vectors_short: this function is identical to the previous one,
  except the 10^8 iterations stage was shortened into 10^6 iterations.
  We recommend to use this test if time is short.
*****************************************************************************************
***********/
void test_vectors_short()
{
  struct NESSIEstruct subkeys;
  u8 key[KEYSIZEB];
  u8 plain[BLOCKSIZEB];
  u8 cipher[BLOCKSIZEB];
  u8 decrypted[BLOCKSIZEB];
  u32 i;
  int v;
  u8 Zynq_cipher[BLOCKSIZEB];
  u8 Zynq_decrypted[BLOCKSIZEB];

  printf("Test vectors -- set 1\n");
  printf("=====================\n\n");

  /* If key size is not a multiple of 8, this tests too much (intentionally) */
  for(v=0; v<(KEYSIZEB*8); v++)
    {
      memset(plain, 0, BLOCKSIZEB);
      memset(key, 0, KEYSIZEB);
      key[v>>3] = 1<<(7-(v&7));

      NESSIEkeysetup(key, &subkeys);
      NESSIEencrypt(&subkeys, plain, cipher);
      NESSIEdecrypt(&subkeys, cipher, decrypted);

      printf("Set 1, vector#%3d:\n", v);
      print_data("key", key, KEYSIZEB);
      print_data("plain", plain, BLOCKSIZEB);
      print_data("cipher", cipher, BLOCKSIZEB);
      print_data("decrypted", decrypted, BLOCKSIZEB);

      if(compare_blocks(plain, decrypted, BLOCKSIZE) != 0)
        printf("** Decryption error: **\n"
               "   Decrypted ciphertext is different than the plaintext!\n");

      Zynq_crypt_simple(plain, key, 1, Zynq_cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 0, Zynq_decrypted);
      print_data("Zynq cipher", Zynq_cipher, BLOCKSIZEB);
      print_data("Zynq decrypted", Zynq_decrypted, BLOCKSIZEB);

      if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
        printf("** Zynq encryption error: **\n"
```

```
            "   Ciphertext from Zynq is different than the reference code cipher!\n");

    if(compare_blocks(plain, Zynq_decrypted, BLOCKSIZE) != 0)
      printf("** Zynq decryption error: **\n"
             "   Decrypted ciphertext on Zynq is different than the plaintext!\n");

    for(i=0; i<99; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 100 times", cipher, BLOCKSIZEB);
    print_data("Iterated 100 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");
    for(i=0; i<900; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 1000 times", cipher, BLOCKSIZEB);
    print_data("Iterated 1000 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    printf("\n");
  }

printf("Test vectors -- set 2\n");
printf("=====================\n\n");

/* If block size is not a multiple of 8, this tests too much (intentionally) */
for(v=0; v<(BLOCKSIZEB*8); v++)
  {
    memset(plain, 0, BLOCKSIZEB);
    memset(key, 0, KEYSIZEB);
    plain[v>>3] = 1<<(7-(v&7));

    NESSIEkeysetup(key, &subkeys);
    NESSIEencrypt(&subkeys, plain, cipher);
    NESSIEdecrypt(&subkeys, cipher, decrypted);

    printf("Set 2, vector#%3d:\n", v);
    print_data("key", key, KEYSIZEB);
    print_data("plain", plain, BLOCKSIZEB);
    print_data("cipher", cipher, BLOCKSIZEB);
    print_data("decrypted", decrypted, BLOCKSIZEB);

    if(compare_blocks(plain, decrypted, BLOCKSIZE) != 0)
      printf("** Decryption error: **\n"
             "   Decrypted ciphertext is different than the plaintext!\n");

    Zynq_crypt_simple(plain, key, 1, Zynq_cipher);
    Zynq_crypt_simple(Zynq_cipher, key, 0, Zynq_decrypted);
    print_data("Zynq cipher", Zynq_cipher, BLOCKSIZEB);
    print_data("Zynq decrypted", Zynq_decrypted, BLOCKSIZEB);

    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    if(compare_blocks(plain, Zynq_decrypted, BLOCKSIZE) != 0)
      printf("** Zynq decryption error: **\n"
             "   Decrypted ciphertext on Zynq is different than the plaintext!\n");

    for(i=0; i<99; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 100 times", cipher, BLOCKSIZEB);
```

```
    print_data("Iterated 100 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");
    for(i=0; i<900; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 1000 times", cipher, BLOCKSIZEB);
    print_data("Iterated 1000 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    printf("\n");
  }

printf("Test vectors -- set 3\n");
printf("=====================\n\n");

for(v=0; v<256; v++)
  {
    memset(plain, v, BLOCKSIZEB);
    memset(key, v, KEYSIZEB);

    NESSIEkeysetup(key, &subkeys);
    NESSIEencrypt(&subkeys, plain, cipher);
    NESSIEdecrypt(&subkeys, cipher, decrypted);

    printf("Set 3, vector#%3d:\n", v);
    print_data("key", key, KEYSIZEB);
    print_data("plain", plain, BLOCKSIZEB);
    print_data("cipher", cipher, BLOCKSIZEB);
    print_data("decrypted", decrypted, BLOCKSIZEB);

    if(compare_blocks(plain, decrypted, BLOCKSIZE) != 0)
      printf("** Decryption error: **\n"
             "   Decrypted ciphertext is different than the plaintext!\n");

    Zynq_crypt_simple(plain, key, 1, Zynq_cipher);
    Zynq_crypt_simple(Zynq_cipher, key, 0, Zynq_decrypted);
    print_data("Zynq cipher", Zynq_cipher, BLOCKSIZEB);
    print_data("Zynq decrypted", Zynq_decrypted, BLOCKSIZEB);

    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");

    if(compare_blocks(plain, Zynq_decrypted, BLOCKSIZE) != 0)
      printf("** Zynq decryption error: **\n"
             "   Decrypted ciphertext on Zynq is different than the plaintext!\n");

    for(i=0; i<99; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 100 times", cipher, BLOCKSIZEB);
    print_data("Iterated 100 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");
    for(i=0; i<900; i++)
    {
      NESSIEencrypt(&subkeys, cipher, cipher);
      Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
    }
    print_data("Iterated 1000 times", cipher, BLOCKSIZEB);
    print_data("Iterated 1000 times on Zynq", Zynq_cipher, BLOCKSIZEB);
    if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
      printf("** Zynq encryption error: **\n"
             "   Ciphertext from Zynq is different than the reference code cipher!\n");
```

```
      printf("\n");
    }

  printf("Test vectors -- set 4 (shortened)\n");
  printf("=====================\n\n");

  for(v=0; v<4; v++)
    {
      memset(plain, v, BLOCKSIZEB);
      memset(key, v, KEYSIZEB);

      NESSIEkeysetup(key, &subkeys);
      NESSIEencrypt(&subkeys, plain, cipher);
      Zynq_crypt_simple(plain, key, 1, Zynq_cipher);

      printf("Set 4, vector#%3d:\n", v);
      print_data("key", key, KEYSIZEB);
      print_data("plain", plain, BLOCKSIZEB);

      for(i=0; i<999999; i++)
        {
          memset(key, cipher[BLOCKSIZEB-1], KEYSIZEB);
          NESSIEkeysetup(key, &subkeys);
          NESSIEencrypt(&subkeys, cipher, cipher);
        }
      print_data("Iterated 10^6 times", cipher, BLOCKSIZEB);
      for(i=0; i<999999; i++)
        {
          memset(key, Zynq_cipher[BLOCKSIZEB-1], KEYSIZEB);
          Zynq_crypt_simple(Zynq_cipher, key, 1, Zynq_cipher);
        }
      print_data("Iterated 10^6 times on Zynq", Zynq_cipher, BLOCKSIZEB);
      if(compare_blocks(cipher, Zynq_cipher, BLOCKSIZE) != 0)
        printf("** Zynq encryption error: **\n"
               "   Ciphertext from Zynq is different than the reference code cipher!\n");

      printf("\n");
    }

  printf("\n\nEnd of test vectors (shortened)\n");
}

/*****************************************************************************************
***********
  7. demonstration: This function is designed to demonstrate the hardware implementation
  correctness.
  It prompts the user for key and for plaintext string data, then processes the data block
  by block.
  Each block is encrypted and then decrypted again. For each block the plaintext, the
  ciphertext,
  the decrypted text (all three in hexadecimal ASCII code) and the block number are printed
  on screen.
  Comparisons are made between the plaintext and decrypted text of each block, and between
  the original characters string and a decrypted re-calculated string.
  If a comparison fails, an error massage appears.
*****************************************************************************************
***********/
void demonstration()
{
  u32 key1, key2, key3, key4;
  bool go = 1;
  u8 data_string[MAX_LENGTH+1], decrypted_string[MAX_LENGTH+1], plain[BLOCKSIZEB],
  cipher[BLOCKSIZEB], decrypted[BLOCKSIZEB];
  printf("************************************************************ \n\r");
  printf("Secret key configuration: \nKHAZAD is using a 128-bit key (32 figures in
  hexadecimal base). \n");
  printf("For your convenience, the key is entered in 4 parts, each one up to 8 figures in
  hexadecimal base \n(leading zeros will be added). \n\r");
  while (go)
  {
      printf("Please enter key part number 1 \n");
      scanf("%x", &key1);
```

```
printf("Please enter key part number 2 \n");
scanf("%x", &key2);
printf("Please enter key part number 3 \n");
scanf("%x", &key3);
printf("Please enter key part number 4 \n\r");
scanf("%x", &key4);
printf("key= %08X%08X%08X%08X \n", key1, key2, key3, key4);
printf("\nPlease enter data to encrypt (up to %d characters, including spaces) \n",
MAX_LENGTH);
scanf(" %[^\r]s", data_string); // scanf format to allow reading spaces

u16 i = 0, j = 0, k = 0, block_num = 0;
while (data_string[i])
{
    if (j < BLOCKSIZEB)
    {
        memset(plain+j, data_string[i], 1);
        j++;
    }

    if ((j != BLOCKSIZEB) && !(data_string[i+1])) // residue exists
        memset(plain+j, 0, BLOCKSIZEB-j); // zero padding

    if (((j != BLOCKSIZEB) && !(data_string[i+1])) || (j == BLOCKSIZEB)) // need to
    encrypt
    {
        if (block_num == 0)
          Zynq_crypt(plain, key1, key2, key3, key4, 1, 0, cipher); // encryption,
          first operation: only_data = 0
        else
          Zynq_crypt(plain, key1, key2, key3, key4, 1, 1, cipher); // encryption, not
          first operation: only_data = 1
        Zynq_crypt(cipher, key1, key2, key3, key4, 0, 1, decrypted); // decryption,
        only_data = 1
        printf("\n Text block number %u: \n", block_num);
        print_data("plaintext", plain, BLOCKSIZEB);
        print_data("ciphertext", cipher, BLOCKSIZEB);
        print_data("decrypted text", decrypted, BLOCKSIZEB);
        if(compare_blocks(plain, decrypted, BLOCKSIZE) != 0)
          printf("** Decryption error: **\n    Decrypted ciphertext is different than
          the plaintext!\n\r");
        for (k = 0; k < j; k++)
          decrypted_string[block_num*BLOCKSIZEB + k] = decrypted[k];
        block_num++;
        j = 0;
    }
    i++;
}

decrypted_string[i] = 0; // NULL character to end decrypted_string
printf("\nThe original string is: %s \n\r", data_string);
printf("The decrypted string is: %s \n\r", decrypted_string);
if (strcmp(data_string, decrypted_string) == 0)
  printf("Texts identical! \n\r");
else
  printf("ERROR: texts not identical! \n\r");

u8 answer;
bool valid_answer = 0;
do {
    printf("Do you want to try this demonstration again? Please answer y/n \n\r");
    scanf(" %c", &answer);
    if ((answer == 'y') || (answer == 'Y') || (answer == '1'))
      valid_answer = 1;
    else if ((answer == 'n') || (answer == 'N') || (answer == '0'))
    {
      valid_answer = 1;
      go = 0;
      printf("Going back to the main menu... \n\r");
    }
    else
      printf("Not a valid input. \n\r");
} while (!valid_answer);
```

```
  }
  return;
}

/*****************************************************************************************
***********
  8. hardware_implementation: A simple practical application.
  The user is prompted to choose between encryption mode or decryption mode.
  For encryption, a key and plaintext string are entered,
  and the ciphertext is calculated and printed on screen as hexadecimal figures.
  For decryption, a key and ciphertext hexadecimal figures string are entered,
  and the plaintext is calculated and printed on screen as regular characters.
*****************************************************************************************
***********/
void hardware_implementation()
{
  u32 key1, key2, key3, key4;
  bool enc_dec, valid_answer, first_run = 1, go = 1;
  u8 answer, data_in_string[MAX_LENGTH+1], cipher_in_string[MAX_LENGTH+1],
  block_in[BLOCKSIZEB], block_out[BLOCKSIZEB+1];
  printf("************************************************************** \n\r");
  while (go)
  {
    do {
          valid_answer = 0;
          printf("For encryption, please press 'e'.  For decryption, please press 'd'. \n\r");
          scanf(" %c", &answer);
          if ((answer == 'e') || (answer == 'E'))
          {
            valid_answer = 1;
            enc_dec = 1;  // =encryption
            printf("Encryption mode \n\r");
          }
          else if ((answer == 'd') || (answer == 'D'))
          {
            valid_answer = 1;
            enc_dec = 0;  // =decryption
            printf("Decryption mode \n\r");
          }
          else
            printf("Not a valid input. \n\r");
          } while (!valid_answer);

    if (!first_run)
    {
        valid_answer = 0;
        do {
            printf("Do you want to enter a new key? Please answer y/n \n\r");
            scanf(" %c", &answer);
            if ((answer == 'y') || (answer == 'Y') || (answer == '1'))
                valid_answer = 1;
            else if ((answer == 'n') || (answer == 'N') || (answer == '0'))
                valid_answer = 1;
            else
                printf("Not a valid input. \n\r");
        } while (!valid_answer);
    }

    if ((first_run) || ((!first_run) && ((answer == 'y') || (answer == 'Y') || (answer ==
    '1'))))
    {
        printf("Secret key configuration: \nKHAZAD is using a 128-bit key (32 figures in
        hexadecimal base). \n");
        printf("For your convenience, the key is entered in 4 parts, each one up to 8
        figures in hexadecimal base \n(leading zeros will be added). \n\r");
        printf("Please enter key part number 1 \n");
        scanf("%x", &key1);
        printf("Please enter key part number 2 \n");
        scanf("%x", &key2);
        printf("Please enter key part number 3 \n");
        scanf("%x", &key3);
        printf("Please enter key part number 4 \n\r");
        scanf("%x", &key4);
```

```
}
printf("key= %08X%08X%08X%08X \n", key1, key2, key3, key4);

u16 i = 0;
if (enc_dec == 1)  // plaintext input: use the input as is
{
    printf("\nPlease enter data to encrypt (up to %d characters, including spaces) \n",
    MAX_LENGTH);
    scanf(" %[^\r]s", data_in_string);  // scanf format to allow reading spaces
    printf("\n\t the ciphertext: \n");
}
else                // ciphertext input: convert the input from hexadecimal figures
string to characters string
{
    printf("\nPlease enter data to decrypt - a sequence of hexadecimal figures pairs (up
    to %d characters) \n", MAX_LENGTH);
    scanf("%s", cipher_in_string);
    while (cipher_in_string[i])
    {
        char temp[] = {cipher_in_string[i],cipher_in_string[i+1]};  // each hexadecimal
        figures pair is the ASCII code of one character
        data_in_string[i/2] = (u8)strtol(temp, NULL, 16);           // convert
        hexadecimal string to long int to u8
        i+=2;
    }
    data_in_string[i/2] = 0;  // NULL character to end the string
    printf("\n\t the plaintext: \n");
}

block_out[BLOCKSIZEB] = 0;    // NULL character to end the string
i = 0;
u16 j = 0, k = 0, block_num = 0;
while (data_in_string[i])
{
    if (j < BLOCKSIZEB)
    {
        memset(block_in+j, data_in_string[i], 1);
        j++;
    }

    if ((j != BLOCKSIZEB) && !(data_in_string[i+1])) // residue exists
        memset(block_in+j, 0, BLOCKSIZEB-j); // zero padding

    if (((j != BLOCKSIZEB) && !(data_in_string[i+1])) || (j == BLOCKSIZEB)) // need to
    encrypt/decrypt
    {
        if ((block_num == 0) && ((first_run) || ((!first_run) && ((answer == 'y') ||
        (answer == 'Y') || (answer == '1')))))
            Zynq_crypt(block_in, key1, key2, key3, key4, enc_dec, 0, block_out); //
            first operation: only_data = 0
        else
            Zynq_crypt(block_in, key1, key2, key3, key4, enc_dec, 1, block_out); // not
            first operation: only_data = 1
        if (enc_dec == 1)
            for (k=0; k < BLOCKSIZEB; k++) // ciphertext output: print as hexadecimal
            figures
            {
                putchar(hex[(block_out[k]>>4)&0xF]);
                putchar(hex[(block_out[k]   )&0xF]);
            }
        else
            printf("%s", block_out);      // plaintext output: print as characters

        block_num++;
        j = 0;
    }
    i++;
}

first_run = 0;
valid_answer = 0;
do {
    printf("\n\nDo you want to use this application again? Please answer y/n \n\r");
```

```
        scanf(" %c", &answer);
        if ((answer == 'y') || (answer == 'Y') || (answer == '1'))
            valid_answer = 1;
        else if ((answer == 'n') || (answer == 'N') || (answer == '0'))
        {
            valid_answer = 1;
            go = 0;
            printf("Going back to the main menu... \n\r");
        }
        else
            printf("Not a valid input. \n\r");
       } while (!valid_answer);
  }
  return;
}

/*************************************************************************************************
***********
  9. print_data: print a given string "str", then "=", then the ASCII code in hexadecimal
  figures of
  each element in a given u8 array.
  From the reference code bctestvectors.c file.
*************************************************************************************************
***********/
void print_data(char *str, u8 *val, int len) // from reference code bctestvectors.c file
{
  int i;

  printf("%25s=", str);
  for(i=0; i<len; i++)
    {
      putchar(hex[(val[i]>>4)&0xF]);
      putchar(hex[(val[i]   )&0xF]);
    }
  putchar('\n');
}

/*************************************************************************************************
***********
  10. compare_blocks: compare two u8 strings with the same given length.
  From the reference code bctestvectors.c file.
*************************************************************************************************
***********/
int compare_blocks(u8 *m1, u8 *m2, int len_bits)
{
  int i;
  int lenb=(len_bits+7)>>3;
  int mask0 = (1<<(((len_bits-1)&7)+1))-1;

  if((m1[0]&mask0) != (m2[0]&mask0))
    return 1;

  for(i=1; i<lenb; i++)
    if(m1[i] != m2[i])
        return 1;

  return 0;
}

/*************************************************************************************************
***********
  11. about: print information about this project.
  When the design operates on bare-metal, there is no filesystem, so no text file can be used.
*************************************************************************************************
***********/
void about()
{

    printf("*************************************************************************************
    **********************\n");

    printf("*************************************************************************************
    **********************\n");
```

```
    printf("Zynq-7000 based Implementation of the KHAZAD Block Cipher\n");
    printf("Yossef Shitzer & Efraim Wasserman\n");
    printf("Jerusalem College of Technology - Lev Academic Center (JCT)\n");
    printf("Department of electrical and electronic engineering\n");
    printf("2018\n\r");

    printf("''The KHAZAD Legacy-Level Block Cipher'' is a block cipher designed by Paulo
    S.L.M. Barreto and Vincent Rijmen.\n");
    printf("It uses a 128-bit key, operates on 64-bit data blocks, and comprises 8
    rounds.\n");
    printf("The algorithm has been submitted as a candidate for the first open NESSIE
    workshop in 2000.\n");
    printf("This first version now considered obsolete. For phase 2 of NESSIE, a modified
    version has been submitted, \n");
    printf("named ''Khazad-tweak'', and has been accepted as NESSIE finalist.\n");
    printf("This version can be found here:\n");
    printf("https://www.cosic.esat.kuleuven.be/nessie/tweaks.html\n\r");

    printf("The algorithm developers wrote: \n");
    printf("''Khazad is named after Khazad-dum, ''the Mansion of the Khazad'', which in the
    tongue of the Dwarves is \n");
    printf("the name of the great realm and city of Dwarrowdelf, of the haunted mithril
    mines in Moria, the Black Chasm.\n");
    printf("But all this should be quite obvious – unless you haven’t read J.R.R. Tolkien's
    ''The Lord of the Rings'', of course :-)  ''\n\r");

    printf("This hardware implementation of KHAZAD uses the MicroZed 7010 development board
    by Avnet Inc., \n");
    printf("which is based on a Xilinx Zynq-7010 All Programmable SoC.\n");
    printf("The Zynq Z-7010 device integrates a dual-core ARM Cortex A9 processor with an
    Artix-7 FPGA.\n");
    printf("This new concept allows many interesting and exciting possibilities.\n");
    printf("In our design, the programmable logic (PL) is used for implementing the
    algorithm, \n");
    printf("and the processing system (PS) is used mainly for dealing with user input &
    output operations.\n");
    printf("The PL design files were written in Verilog. Synthesis and Implementation were
    done using Xilinx Vivado.\n");
    printf("The PS program was written in C, and compiled using Xilinx SDK.\n\r");

    printf("The MicroZed development board can be used as both a stand-alone board, \n");
    printf("or combined with a carrier card as an embeddable system-on-module.\n");
    printf("This implementation was designed to be fully operational even when using the
    stand-alone mode.\n");
    printf("Plugging the board into the carrier card will activate more indicator LEDs.\n\r");

    printf("This project was created as a final year project, with the guidance of Mr. Uri
    Stroh.\n");
    printf("We want to thank Mr. Stroh for his guidance and help, \n");
    printf("the Lev Academic Center (JCT) staff for supplying equipment and technical
    support, \n");
    printf("and the Xilinx and Avnet companies for their fine products, useful documentation
    and helpful websites.\n\r");
}


#endif
```

```
/*****************************************************************************************************
***********
*****************************************************************************************************
***********
Zynq-7000 based Implementation of the KHAZAD Block Cipher
Yossef Shitzer & Efraim Wasserman
Jerusalem College of Technology - Lev Academic Center (JCT)
Department of electrical and electronic engineering
2018
*****************************************************************************************************
***********
*****************************************************************************************************
***********
Main program file
*****************************************************************************************************
***********
*****************************************************************************************************
***********
Version 1.0: ECB implementation
*****************************************************************************************************
***********
*****************************************************************************************************
***********/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h> // for usleep
#include "platform.h"
#include "xgpio.h"
#include "xgpiops.h"
#include "Xscugic.h"
#include "xil_exception.h"
#include "nessie_modified.h"
#include "KHAZAD_Zynq.h"

int main()
{
  init_platform(); // register initialization and UART setup
  printf("************************************************************ \n");
  printf("************************************************************ \n\r");
  printf("KHAZAD block cipher hardware implementation using the Zynq-7000 device and the
  MicroZed development board \n\r");
  printf("Yossef Shitzer & Efraim Wasserman \n\r");
  printf("Jerusalem College of Technology - Lev Academic Center (JCT) \n\r");
  printf("Department of electrical and electronic engineering \n\r");
  printf("2018 \n\r");
  printf("************************************************************ \n");
  printf("************************************************************ \n\r");

  // Peripherals and interrupt configuration:
  board_configuration();

  printf("AXI_GPIO0 address = 0x%x \n\r", ADDR0);
  printf("AXI_GPIO1 address = 0x%x \n\r", ADDR1);
  printf("AXI_GPIO2 address = 0x%x \n\r", ADDR2);
  printf("AXI_GPIO3 address = 0x%x \n\r", ADDR3);
  printf("AXI_GPIO address offset = 0x%08x \n\r", ADDR_offset);

  // PL design initialization:
  /* ctrl is received in the PL as signal ctrl_from_PS[3:0]:
     bit 3 - RST
     bit 2 - only_data. only_data = 1: new data, same key. only_data = 0: new data and new
     key.
     bit 1 - enc_dec: the desired operation. enc_dec = 1: encryption, enc_dec = 0: decryption.
     bit 0 - bistable start/ready semaphore flag. To run an operation this bit must not be
     equal to a matching flag in the PL. */
  XGpioPs_WritePin(&my_Gpio, 47, 0); // turn off the PS-ready indicator LED
  ctrl = 0x0008; // reset=1
  Xil_Out16(ADDR0,ctrl); // send from PS to FPGA via AXI interface
  usleep(50000);
  ctrl = 0x0000; // reset=0
  Xil_Out16(ADDR0,ctrl);
```

```
  XGpioPs_WritePin(&my_Gpio, 47, 1); // turn on the LED

  printf("****************************************************************** \n\r");
  printf("Welcome to the KHAZAD block cipher hardware implementation! \n\r");
  bool go = 1;
  u8 option;
  while (go) {
      printf("****************************************************************** \n\n\r");
      printf("Please choose an option: \n\n\r");
      printf("--1-- \t Hardware implementation application \n\r");
      printf("--2-- \t Hardware implementation correctness demonstration \n\r");
      printf("--3-- \t Test vectors - full version. Warning: this test may take a long time
      \n\r");
      printf("--4-- \t Test vectors - short version \n\r");
      printf("--5-- \t About \n\r");
      printf("--6-- \t Exit \n\r");
      printf("To reset the FPGA design, you may press the MicroZed user button at any time.
      \n");
      printf("(Resetting the design while mid-operation may lead to wrong results.) \n\r");
      scanf(" %c", &option);

      switch(option) {
      case '1':
          hardware_implementation();
          break;
      case '2':
          demonstration();
          break;
      case '3':
          test_vectors_full();
          break;
      case '4':
          test_vectors_short();
          break;
      case '5':
          about();
          break;
      case '6':
          go = 0;
          printf("Thanks for using this design. Goodbye! \n\r");
          break;
      default:
          printf("Not a valid input. \n\r");
      }
  }

  cleanup_platform(); // cleanup, disable cache
  return 0;
}
```

```
##############################################################################################
############
##############################################################################################
############
# Zynq-7000 based Implementation of the KHAZAD Block Cipher
# Yossef Shitzer & Efraim Wasserman
# Jerusalem College of Technology - Lev Academic Center (JCT)
# Department of electrical and electronic engineering
# 2018
##############################################################################################
############
##############################################################################################
############
# This is an XDC file for the outputs going to the MicroZed I/O Carrier Card.
##############################################################################################
############
##############################################################################################
############
# Version 2.0: ECB+CBC implementation
##############################################################################################
############
##############################################################################################
############
# user LED 0:
set_property -dict {PACKAGE_PIN U14 IOSTANDARD LVCMOS33} [get_ports RST_LED]
# user LED 1:
set_property -dict {PACKAGE_PIN U15 IOSTANDARD LVCMOS33} [get_ports PL_ready_LED]
# user LED 3:
set_property -dict {PACKAGE_PIN U19 IOSTANDARD LVCMOS33} [get_ports encryption_LED]
# user LED 4:
set_property -dict {PACKAGE_PIN R19 IOSTANDARD LVCMOS33} [get_ports decryption_LED]
# user LED 6:
set_property -dict {PACKAGE_PIN P14 IOSTANDARD LVCMOS33} [get_ports ECB_LED]
# user LED 7:
set_property -dict {PACKAGE_PIN R14 IOSTANDARD LVCMOS33} [get_ports CBC_LED]
```

```
##############################################################################################
############
##############################################################################################
############
# Zynq-7000 based Implementation of the KHAZAD Block Cipher
# Yossef Shitzer & Efraim Wasserman
# Jerusalem College of Technology - Lev Academic Center (JCT)
# Department of electrical and electronic engineering
# 2018
##############################################################################################
############
##############################################################################################
############
# This is an XDC file for the outputs going to the MicroZed I/O Carrier Card.
##############################################################################################
############
##############################################################################################
############
# Version 1.0: ECB implementation
##############################################################################################
############
##############################################################################################
############
# user LED 0:
set_property -dict {PACKAGE_PIN U14 IOSTANDARD LVCMOS33} [get_ports PL_ready_LED]
# user LED 3:
set_property -dict {PACKAGE_PIN U19 IOSTANDARD LVCMOS33} [get_ports encryption_LED]
# user LED 4:
set_property -dict {PACKAGE_PIN R19 IOSTANDARD LVCMOS33} [get_ports decryption_LED]
# user LED 7:
set_property -dict {PACKAGE_PIN R14 IOSTANDARD LVCMOS33} [get_ports RST_LED]
```