Cryptographic Engineering 2021-22

Exercise 3

In this exercise, you will implement a cryptoprocessor by 1) integrating the previously designed TRNG and AES instructions, and 2) designing new instruction(s) for supporting ring-LWE public-key encryption (PKE).

The PKE protocol will run from the SDK and it will use the newly created instructions for performing polynomial arithmetic.

The exercise has a total of 40 points. See 'Marking Scheme' for more information.

The deadline for code submission is 26th January 23:59.
See 'Code Submission' for more information.

You add the new HDL source files incrementally with respect to your Vivado project from Exercise 1 & 2. That will give you a full cryptosystem.

Download the latest SDK project 'SDK_cryptosystem.zip' from https://www.iaik.tugraz.at/ce
and replace the old directory \project_1\project_1.sdk with the new 'project_1.sdk' that is present inside the downloaded zip file.
(You may keep a backup of the old SDK project by renaming the old project_1.sdk directory)

There are three tasks.

Task 1: **Preliminaries** (0 points)

Open the file ComputeCore.v, review it carefully, and try to understand how different instruction blocks are instantiated in this module. See how the memory read/write signals are multiplexed based on the reset signals of the instructions.

You will be required to instantiate new instructions inside ComputeCore().
Instruction op-codes available for the new instructions are from 1-to-17 and 21-to-29. Note that 18, 19, and 20 are currently in use for TRNG, AES-enc, AES-dec respectively.

Also, see how a program word is decoded into an instruction-opcode and read/write base addresses.

Note that *OP2_sel* is currently set to 0 in ComputeCore(). Hence, a physical read address is generated with OP1 as the base address. You will need to change OP2_sel accordingly to enable operations between two operands OP1 and OP2.

Please review the 'Cryptoprocessor Overview' document on the course website for how to add new instructions: https://www.iaik.tugraz.at/wp-content/uploads/2021/08/practical-session-2021-11-29.pdf

In the SDK project, see the two files *instruction.h/.c* to understand how the SW commands the HW to perform a certain task. You need to add your new instruction intrinsics in these two files. Also, have a look at aes/test_aes.c and trng/test_trng.c to see how they use HW-instructions. You will need to do similar interfacing for the PKE protocol.

Task 2: Implementation of Polynomial Arithmetic (Total 30 points)

Open the HDL source directory 'Verilog_src' where the two previous subdirectories for '*trng*' and '*aes*' are present, and create the new subdirectory
'*pke/poly_arithmetic*'.

Keep all HDL source files associated with polynomial arithmetic inside '*pke/poly_arithmetic*'.

Choice of modulus: You are free to choose any coefficient modulus. Currently PKE_q = 2^12 in pke_parameters.h. You may update PKE_q (the modulus q) with a new 12-bit coefficient modulus value.

2.1 Polynomial adder/subtracter (10 points)

Implement a polynomial adder/subtracter unit and make it a new instruction block of the cryptoprocessor by instantiating it inside ComputeCore(). Note that there will be two instructions: one for addition and the other for subtraction.

This unit reads its operand words from the BRAM, then computes the resulting coefficients by performing coefficient-wise modular addition/subtraction, and writes them to the BRAM.

Update the SDK project with the new instruction intrinsics in instruction.c/.h. Test the functionality of the polynomial adder/subtracter unit from the SDK.

(*Performance hints*: For the given parameter set of the ring-LWE PKE, a coefficient is at most 12 bits long. Hence, you can put 4 coefficients of type uint16_t in one word of the BRAM.)

2.2 Polynomial multiplier (20 points)

Implement a polynomial multiplier unit and make it a new instruction block of the cryptoprocessor by instantiating it inside ComputeCore().

This unit reads operand polynomials from the BRAM and writes the resulting polynomial into the BRAM. You may follow HW/SW co-design for implementing the polynomial multiplier.

Note that polynomial multiplication is the most expensive operation in ring-LWE. Hence, ***optimize the polynomial multiplier very well*** towards your desired optimization goal.

Update the SDK project with the new instruction intrinsics in instruction.c/.h. Test the functionality of the polynomial multiplier from the SDK.

Task 3: HW/SW implementation (Total 10 points)

In this task you use the HW-based instructions from the SDK-based SW project to execute cryptographic operations.

3.1 In the SDK project, re-implement the PKE protocol such it uses the following HW-based instructions:

- TRNG instruction for generating quality random seeds.
- Polynomial arithmetic instructions for performing the polynomial operations.

3.2 Optimize the entire PKE protocol such that it minimizes the number of SW–HW polynomial-data exchanges. Such large data exchanges are very slow and hamper performance. You will report the cycle count for the entire protocol.

(*Hints:* You may use the BRAM to store intermediate values and perform the next operations in HW.)

3.3 In this task you check if the overall cryptoprocessor system (SW and HW) is functioning properly. In the SDK, using the AES and PKE functionalities, implement the following proof-of-concept secure communication protocol and then test it.

- First generate a public-private key-pair.
- Then use the public-key to encrypt a 256-bit random message. Low 128 bits of the message is the first AES key.
- Decrypt the encrypted message using the PKE-decryption and use the low 128 bits of the decrypted message as the other AES-key. The two AES keys should match.
- Now perform AES-encryption using the first key and AES-decryption using the other key. Compare the messages and the decrypted messages.

Check this protocol by running it at least 1000 times. (You may comment out unnecessary printf statements during this test).

Submission guidelines

- The deadline for submitting your project is 26th January 23:59.
- You will upload your project to the git repository of your team as ‘assignment3’.
- More information will be provided in the coming weeks on the structure of the report.

Marking scheme

Task 1 (0 points)

You get no points in Task 1.

Task 2 (30 points)

- You get 7 points in Task 2.1 if you have a working implementation of the polynomial adder/subtracter unit and you defend your work successfully.

  Otherwise you do not get any points for Task 2.1 overall.

  The remaining 3 marks in Task 2.1 are for the optimizations (e.g., cycle count, number of memory access, area, etc.).

- You get 12 points in Task 2.2 if you have a working implementation of polynomial multiplier unit and you defend your work successfully.

  You get additional 0 to 8 points based on how well your implementation of Task 2.2 performs (in terms of resource requirements, speed, etc.) compared to the implementations of other teams and our own implementation.

Task 3 (10 points)

- You get 3 points in Task 3.1 when it works correctly.
- You get 1 - 3 points depending on how well your implementation of Task 3.2 is compared to the implementations of the other teams and our own implementation.
- You get 4 points in Task 3.3 if your implementation passes all the 1000 tests. Otherwise you obtain a lower point.