Cryptography on Hardware Platforms (WS 2023/24)

Assignment 1

Code review deadline 23rd November.
Final deadline 7th December.

In Assignment-1, you will implement a public-key cryptography design following hardware/software codesign. The assignment consists of several tasks starting with Task-1.

The total points for Assignment-1 is 60 including 15 points for the individual oral defense.

Review class notes on modular arithmetic, polynomial arithmetic and the Vivado tutorial before starting the assignment.

Task-1: Hardware implementation of a 64-bit modular multiplier (10 points)

Soft deadline: 12th November

You are given the 64-bit prime p = 18434813901432784897. Implement a modular multiplier circuit entirely in hardware that multiplies two elements $a, b \in F_p$ and produces the result $c \in F_p$ such that $c = a*b*R^{-1} \mod p$ where $R=2^{64}$. The modular reduction must be performed using the *Montgomery method*.

The input/output ports of the module for this modular multiplier are defined as follows.

```
module modular_multiplier(clk, mul_ina, mul_inb, mul_start, mul_out, mul_done);
input clk;
input [63:0] mul_ina, mul_inb;  // Two 64 bit inputs a and b
input mul_start;                // When this signal is 1, the multiplication starts.
output [63:0] mul_out;          // Result of the modular multiplication is available in this port.
output mul_done;                // This signal becomes 1 when the multiplication ends.
```

Simulate the module with several inputs to verify the functional correctness of your modular reduction circuit.

Synthesize and implement the module using Vivado 2020.2 to obtain area and clock frequency reports. For the synthesis and implementation, the FPGA board must be set to Pynq-z2 in Vivado 2020.2.

The modular multiplier should obtain over 100 MHz clock frequency as the overall public-key processor will be clocked at 100 MHz in the FPGA.

Optimization hints: Generally it is not the case that a hardware design obtains high speed as well as very small low area. You can consider optimizing the modular reduction circuit for speed or area. Bit-parallel architectures offer high throughput e.g., one modular reduction per cycle. Whereas, sequential architectures offer small area by using resource sharing between components. In both cases, pipeline registers play an important role in obtaining good clock frequency. As the modular multiplier performs several multiplications, try to optimally use DSP multipliers for obtaining a good design.

About optimizations, there is a great quote by a great person: "*If I had eight hours to chop down a tree, I'd spend six hours sharpening my axe.*" – so, plan your design well before writing the code.

We provide a Python model of Montgomery reduction operation for prime *p* on course webpage.

Task-2: Hardware/Software co-design implementation of polynomial multiplier (30 points)

Intermediate deadline: 23rd November. We provide you feedback on your code from git.
Final Deadline: 7th December. You submit the project.

Before starting the hardware implementation of the polynomial multiplier, make sure that you have a software model (e.g., Python) of your polynomial multiplication algorithm (e.g., Karatsuba or NTT-based) working in the Montgomery domain.

Open the Vivado project and Vitis workspace of Assignment 1 for the implementation of this task (files are available in course website).

Implement a polynomial multiplier for multiplying two polynomials where each polynomial is in the polynomial ring $R_p = \mathbf{Z}_p/<x^{256} + 1>$ with the prime coefficient modulus p = 18434813901432784897. That means each polynomial has 256 coefficients and each coefficient is modulo p.

The polynomial multiplier architecture must use the Montgomery modular multiplier circuit from Task-1 to compute all modular multiplications. If you do not have a working Montgomery modular multiplier circuit, contact us as soon as possible.

The input/output ports of the module for the polynomial multiplier are defined as follows.

```
module polynomial_multiplication(clk, rst, read_poly_op_sel, read_address, write_address, wea,
                                 data64_in, done, data64_out);


input clk;
input rst;                      // Active high
output [9:0] read_address;      // BRAM address for reading a 64-bit word
output read_poly_op_sel;        // Select one of the two polynomial operands
input [63:0] data64_in;         // Word read from the BRAM

output [9:0] write_address;     // BRAM address where the result word will be written
output wea;                     // BRAM write enable signal. The data in the write-bus will get written when this is 1.
output [63:0] data64_out;       // Word to be written to BRAM in the memory address specified by write_address
output done;                    // This signal becomes 1 when a given polynomial multiplication gets computed.
                                // After that it should stay 1 until rst becomes 1.
```

This module reads operand polynomials from the BRAM memory of a cryptoprocessor and writes the resulting polynomial into the BRAM.

You may follow HW/SW co-design for implementing the polynomial multiplier. That means computationally expensive parts of the polynomial multiplier are executed in the hardware while the remaining cheaper parts are executed in the software. Note that communication between ARM and FPGA is slow. Generally, the best performance is achieved when almost all of the expensive computation steps are executed in the hardware.

If you use HW/SW codesign approach, the base multiplier architecture in the FPGA must be at least a 32 coefficient polynomial multiplier. Example: say, you are using the Karatsuba method and using a 32-coefficient or larger base polynomial multiplier in the FPGA.

Like Task-1, simulate your multiplier architecture to check functionality. Compile (run implementation option) the Cryptoprocessor project in Vivado and ensure that the project compiles and meets the timing constraint of 100 MHz.

Optimization hints: Optimize the polynomial multiplier very well for speed or area. There are several important design choices that you have to make. Algorithm has an important role in the performance as well as implementation complexity. For example, implementing a deep Karatsuba solely in HW will be quite challenging. Architectural design choices make an implementation efficient or inefficient. HW/SW task partitioning is also an important aspect to influence the performance and area.

Here are some HW/SW data transfer costs obtained from the board:

1. Sending one buffer of size 64x1024 bits from ARM to FPGA takes ~18K cpu cycles.
2. Receiving one buffer of size 64x1024 bits from FPGA to ARM takes ~9K cpu cycles.
3. Sending/receiving *m* times smaller data roughly takes 1/*m* times cpu cycles.

Here are speed figures for one of several good implementations. Consider NTT-based polynomial multiplication c(x) = a(x)*b(x) as an example.

1. One time transferring a(x) and b(x) to FPGA costs 2*4.5K cpu cycles.
2. The polynomial multiplier is solely in HW and takes 7K FPGA cycles which is equivalent to 5*7= 35K cpu cycles since the CPU clock is 5x faster than FPGA clock.
3. Receiving the result polynomial c(x) takes 2.5K cpu cycles.
4. Total cpu cycle = 2*4.5 + 5*7 + 2.5 ~ 50K cpu cycles

If you choose a HW/SW codesign approach, consider the overall scenario (data exchange + FPGA computation + SW computation) as the metric for obtaining indicative speed figures. For the software side computation cost, you may assume that one 64-bit arithmetic instruction costs 5 to 10 cpu cycles for integer addition or multiplication.

Metric for area consumption in FPGA: After implementation of the Vivado project, you can check the utilization report to obtain resource count for individual modules e.g, modular multiplier and the polynomial multiplier. As FPGA consists of heterogeneous logic and memory elements, use the following area metric.

$$\#\text{Area} = 1*LUT + 1*FF + 100*DSP + 1000*BRAM$$

For example, if your polynomial multiplier uses 500 LUTs, 230 FFs, 5 DSPs, and 1 BRAM then the total area = 2230.

Task-3: Testing the polynomial multiplier in FPGA (5 points)

Final Deadline: 6th December. You submit the project.

Use Vitis to use the FPGA's polynomial multiplier. Test the functional correctness and obtain real performance benchmark (i.e, cycle count) from Vitis for 1000 random test cases.

If you use standard hardware design practices, and your design is synthesizable and meets timing constraints, then it is likely that the real HW implementation will work correctly.

See course website for how to Vitis platform.

Submission guidelines

- The **final deadline** for submitting the entire project is 6th December midnight.
  Individual defence of the project will take place in the next week.
- Soft deadline for preliminary code review is 23rd November.
- Upload the Vivado project and Vitis workspace to the git repository of your team by the deadlines.
- Upload your short one-page report to the git repository of your team by the final deadline. The report should have:
    - Summary and explanation of your design choices strategy.
    - Optimization techniques that you used.
    - Implementation results of your polynomial multiplier (cycle count and area count).
        - Number of LUTs, FFs, DSPs, BRAMs and Area metric
        - Cycle count spend on data transfer, HW-side and SW-side

Marking scheme

15 points are reserved for the individual oral defense of Assignment 1.

Task 1 (10 points)

- You get 7 points if the modular multiplier is functionally correct and the code is synthesizable in Vivado.

- You get 0 to 3 points based on how well your implementation of Task 1 is optimized in terms of resource requirements or speed or both.

Task 2 (30 points)

- You get 22 points if the polynomial multiplier (or base polynomial multiplier in HW/SW codesign) is functionally correct in simulation and the project is synthesizable in Vivado.
- You get a point between 0 and 8 depending on various optimizations you perform and how close the polynomial multiplier is to the theoretical best performance with a given hardware area cost.
  No points for optimization will be awarded if the design is not correct or non-implementable.

Task 3 (5 points)

- You get 5 points if the implementation passes the tests in the FPGA board.

You will be working in teams of two. We expect each of you to contribute equally to the assignment. Individual defense of the assignment will take place a week after the submission deadline. The defense has 15 points. You will be asked questions related to the project of Assignment 1.