高階系統晶片設計實驗期末專題報告

Lightweight Cryptographic to the FPGA

Students

R12921068 吳凱濠

R12921A10 謝宗翰

F11921061 郭霖璟

June 2024

Contents

- **Chapter 1 Background Introduction**
- **Chapter 2 IP Structure Design**
- **Chapter 3 FPGA PS Side Validation**

Chapter 1 Background Introduction

1. Increasing Demand

With the development of the Internet of Things (IoT), the demand for security on embedded devices is getting higher and higher.

2. What is ASCON?

ASCON is a lightweight hash function and AEAD (authenticated encryption with associated data) family. It has recently been selected as the primary choice in the LWC project by NIST.

3. Lightweight Crypto Challenges

As the main aim of lightweight cryptography are resource-constrained embedded devices, one of the main concerns are efficient implementations and protection against physical attacks.

Chapter 2 IP Structure Design

1. The consideration of conversion C to HLS

Due to the non-blocking feature of Verilog, there is no need to specifically store variables before performing calculations during design. However, in C/HLS-C, because it executes line by line, variables must be stored first before calculations can be made.

Verilog

```
assign sl0 = (x4 & x1) ^ x3 ^ (x2 & x1) ^ x2 ^ (x1 & x0) ^ x1 ^ x0;
assign sl1 = x4 ^ (x3 & x2) ^ (x3 & x1) ^ x3 ^ x2 ^ x1 ^ x0 ^ (x2 & x1);
assign sl2 = (x4 & x3) ^ x4 ^ x2 ^ x1 ^ 64'hffffffffffffffff;
assign sl3 = (x4 & x0) ^ (x3 & x0) ^ x4 ^ x3 ^ x2 ^x1 ^ x0;
assign sl4 = (x4 & x1) ^ x4 ^ x3 ^ (x1 & x0) ^ x1;
```

C/HLS-C

```
t[0] = state[0]; t[1] = state[1]; t[2] = state[2]; t[3] = state[3]; t[4] = state[4];
state[0] = (t[4] & t[1]) ^ t[3] ^ (t[2] & t[1]) ^ t[2] ^ (t[1] & t[0]) ^ t[1] ^ t[0];
state[1] = t[4] ^ (t[2] & t[3]) ^ t[3] ^ (t[3] & t[1]) ^ t[2] ^ (t[1] & t[2]) ^ t[1] ^ t[0] ;
state[2] = (t[4] & t[3]) ^ t[4] ^ t[2] ^ t[1] ^ 0xffffffffffffffff;
state[3] = (t[4] & t[0]) ^ (t[3] & t[0]) ^ t[4] ^ t[3] ^ t[2] ^ t[1] ^ t[0];
state[4] = (t[4] & t[1]) ^ t[4] ^ t[3] ^ (t[1] & t[0]) ^ t[1];
```

2. Challenges when designing ASCON IP

- **Different input width between FSIC and IP**

  The original design of the HLS IP input port was 128 bits, but FSIC data input port is only 32 bits, therefore the input have to divide into 4 x 32bits.

- **Cosim hang out when RTL simulation**

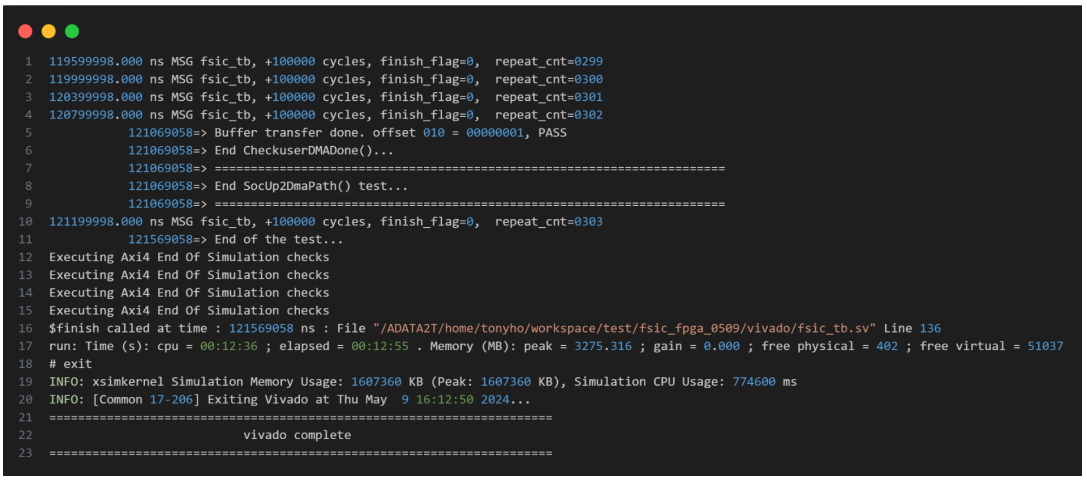
  Due to the aforementioned issue (different width). The numbers of adlen and plen were incorrect, leading to a cosim hang out.

- **Put our HLS result into FSIC**

  When integrated into FSIC, we are unable to transmit data. Ultimately, it was discovered that when reg_rst set to 1, IP does not receive input or generate output.

  Only after all key, nonce, adlen, plen are programmed, and reg_rst is then set to 0, IP begin to receive input and produce output.

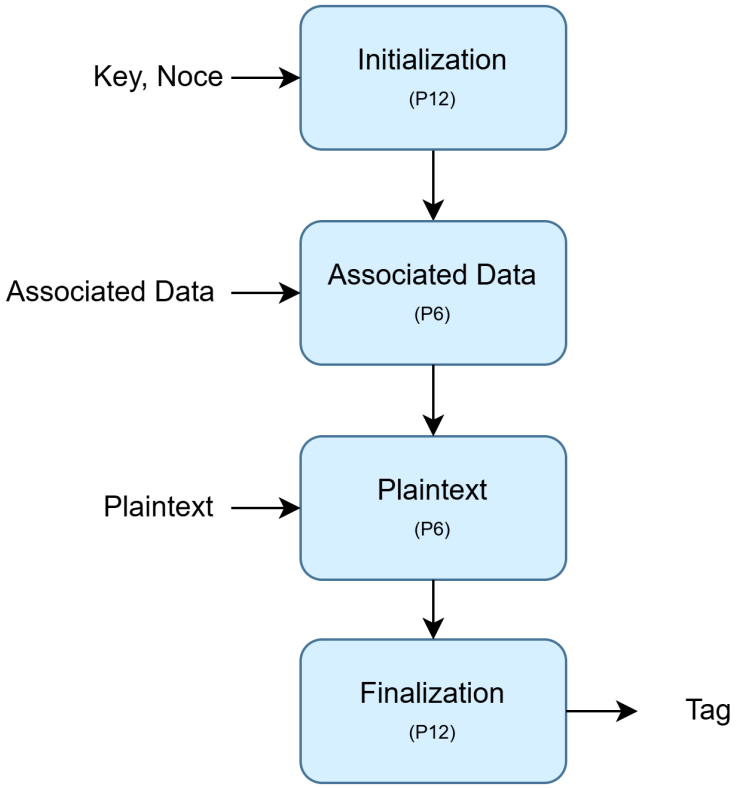
3. Simulation Result



```
119599998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0,  repeat_cnt=0299
119999998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0,  repeat_cnt=0300
120399998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0,  repeat_cnt=0301
120799998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0,  repeat_cnt=0302
          121069058=> Buffer transfer done. offset 010 = 00000001, PASS
          121069058=> End CheckuserDMADone()...
          121069058=> ======================================================================
          121069058=> End SocUp2DmaPath() test...
          121069058=> ======================================================================
121199998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0,  repeat_cnt=0303
          121569058=> End of the test...
Executing Axi4 End Of Simulation checks
Executing Axi4 End Of Simulation checks
Executing Axi4 End Of Simulation checks
Executing Axi4 End Of Simulation checks
$finish called at time : 121569058 ns : File "/ADATA2T/home/tonyho/workspace/test/fsic_fpga_0509/vivado/fsic_tb.sv" Line 136
run: Time (s): cpu = 00:12:36 ; elapsed = 00:12:55 . Memory (MB): peak = 3275.316 ; gain = 0.000 ; free physical = 402 ; free virtual = 51037
# exit
INFO: xsimkernel Simulation Memory Usage: 1607360 KB (Peak: 1607360 KB), Simulation CPU Usage: 774600 ms
INFO: [Common 17-206] Exiting Vivado at Thu May  9 16:12:50 2024...
======================================================================
                        vivado complete
======================================================================
```

4. Our IP Specification and Structure

Port Name	Port Address	Width (bit)	Port Functionality
reg_rst	0x00	1	reset IP when rest_rst = 1
reg_key1 - 4	0x04 - 0x10	32	128 bit key for encrypt plaintext
reg_nonce1 - 4	0x14 - 0x20	32	input with key
adlen	0x24	32	associate data length
plen	0x28	32	number of plaintext


Key, Noce
Initialization
(P12)
Associated Data
Associated Data
(P6)
Plaintext
Plaintext
(P6)
Finalization
(P12)
Tag

Chapter 3 FPGA PS Side Validation

1. User project R/W test

In the Jupyter Notebook, any data we write to the IP returns 0. Eventually, we changed the default value of rdata_tmp in user_prj.v to a non-zero value, and then we got a value on the notebook.

```
always @* begin
  if      (rd_addr[11:2] == 10'h000) rdata_tmp = reg_rst;
  else if (rd_addr[11:2] == 10'h001) rdata_tmp = reg_key1;
  else if (rd_addr[11:2] == 10'h002) rdata_tmp = reg_key2;
  else if (rd_addr[11:2] == 10'h003) rdata_tmp = reg_key3;
  else if (rd_addr[11:2] == 10'h004) rdata_tmp = reg_key4;
  else if (rd_addr[11:2] == 10'h005) rdata_tmp = reg_nonce1;
  else if (rd_addr[11:2] == 10'h006) rdata_tmp = reg_nonce2;
  else if (rd_addr[11:2] == 10'h007) rdata_tmp = reg_nonce3;
  else if (rd_addr[11:2] == 10'h008) rdata_tmp = reg_nonce4;
  else if (rd_addr[11:2] == 10'h009) rdata_tmp = reg_adlen;
  else if (rd_addr[11:2] == 10'h00A) rdata_tmp = reg_plen;
  else                               rdata_tmp = 32'h4455;
end
```

In [39]:
```
ADDRESS_OFFSET = SOC_UP # 0x0000
mmio.write(ADDRESS_OFFSET, 0x00000001) # axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
print("mmio.read(ADDRESS_OFFSET): ", hex(mmio.read(ADDRESS_OFFSET))) # axil_cycles_gen(ReadCyc, SOC_UP, offset, data, 1);

mmio.write(ADDRESS_OFFSET + 0x04, 0x0000) # axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
print("mmio.read(ADDRESS_OFFSET + 0x04): ", hex(mmio.read(ADDRESS_OFFSET+ 0x04))) # axil_cycles_gen(ReadCyc, SOC_UP, offset, data
mmio.write(ADDRESS_OFFSET + 0x08, 0x0000) # axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
print("mmio.read(ADDRESS_OFFSET + 0x08): ", hex(mmio.read(ADDRESS_OFFSET+ 0x08))) # axil_cycles_gen(ReadCyc, SOC_UP, offset, data
mmio.write(ADDRESS_OFFSET + 0x0C, 0x0000) # axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
print("mmio.read(ADDRESS_OFFSET + 0x0C): ", hex(mmio.read(ADDRESS_OFFSET+ 0x0C))) # axil_cycles_gen(ReadCyc, SOC_UP, offset, data
mmio.write(ADDRESS_OFFSET + 0x10, 0x0000) # axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
print("mmio.read(ADDRESS_OFFSET + 0x10): ", hex(mmio.read(ADDRESS_OFFSET+ 0x10))) # axil_cycles_gen(ReadCyc, SOC_UP, offset, data

mmio.write(ADDRESS_OFFSET + 0x14, 0x0000) # axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
print("mmio.read(ADDRESS_OFFSET + 0x14): ", hex(mmio.read(ADDRESS_OFFSET+ 0x14))) # axil_cycles_gen(ReadCyc, SOC_UP, offset, data
mmio.write(ADDRESS_OFFSET + 0x18, 0x0001) # axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
print("mmio.read(ADDRESS_OFFSET + 0x18): ", hex(mmio.read(ADDRESS_OFFSET+ 0x18))) # axil_cycles_gen(ReadCyc, SOC_UP, offset, data
mmio.write(ADDRESS_OFFSET + 0x1C, 0x0000) # axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
print("mmio.read(ADDRESS_OFFSET + 0x1C): ", hex(mmio.read(ADDRESS_OFFSET+ 0x1C))) # axil_cycles_gen(ReadCyc, SOC_UP, offset, data
mmio.write(ADDRESS_OFFSET + 0x20, 0x0002) # axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
print("mmio.read(ADDRESS_OFFSET + 0x20): ", hex(mmio.read(ADDRESS_OFFSET+ 0x20))) # axil_cycles_gen(ReadCyc, SOC_UP, offset, data

mmio.write(ADDRESS_OFFSET + 0x24, 0x0002) # axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
print("mmio.read(ADDRESS_OFFSET + 0x24): ", hex(mmio.read(ADDRESS_OFFSET+ 0x24))) # axil_cycles_gen(ReadCyc, SOC_UP, offset, data
mmio.write(ADDRESS_OFFSET + 0x28, 0x001E) # axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
print("mmio.read(ADDRESS_OFFSET + 0x28): ", hex(mmio.read(ADDRESS_OFFSET+ 0x28))) # axil_cycles_gen(ReadCyc, SOC_UP, offset, data

# ADDRESS_OFFSET = SOC_CC # 0x5000
# mmio.write(ADDRESS_OFFSET, 0x0000)
# print("mmio.read(ADDRESS_OFFSET): ", hex(mmio.read(ADDRESS_OFFSET)))
```

```
mmio.read(ADDRESS_OFFSET):  0x1
mmio.read(ADDRESS_OFFSET + 0x04):  0x0
mmio.read(ADDRESS_OFFSET + 0x08):  0x0
mmio.read(ADDRESS_OFFSET + 0x0C):  0x0
mmio.read(ADDRESS_OFFSET + 0x10):  0x0
mmio.read(ADDRESS_OFFSET + 0x14):  0x0
mmio.read(ADDRESS_OFFSET + 0x18):  0x1
mmio.read(ADDRESS_OFFSET + 0x1C):  0x0
mmio.read(ADDRESS_OFFSET + 0x20):  0x2
mmio.read(ADDRESS_OFFSET + 0x24):  0x2
mmio.read(ADDRESS_OFFSET + 0x28):  0x1e
```

2. LADMA Ouput Test

In the Jupyter Notebook, the output of our LADMA is the value of the counter. Upon inspecting the code, we found that in LogicAnalyzer.v, after la_up_data is input, it must be under the condition that la_enable = 1 for la_up_data to be output; otherwise, it will maintain its original value.

When we try to program la_enable in Jupyter Notebook, we find that PYNQ crashes. After attempting to solve this issue for a week, we still cannot resolve the problem

```
assign la_changed = |( la_enable) & ( (up_la_data & la_enable) != r_la_data );
   // assign la_changed = up_la_data != r_la_data;

   ////////////////////////////////////////////
   // Always for latch up_la_data if changed //
   ////////////////////////////////////////////
   always @(posedge axi_clk or negedge axi_reset_n) begin
       if(!axi_reset_n) begin
           r_la_data <= 23'h0;
           r_la_data_available <= 1'b0;
       end else begin
           if( la_changed ) begin
               r_la_data <= up_la_data & la_enable;
               // r_la_data <= up_la_data;
               r_la_data_available <= 1'b1;
           end else begin
               r_la_data_available <= 1'b0;
           end
       end
   end
```