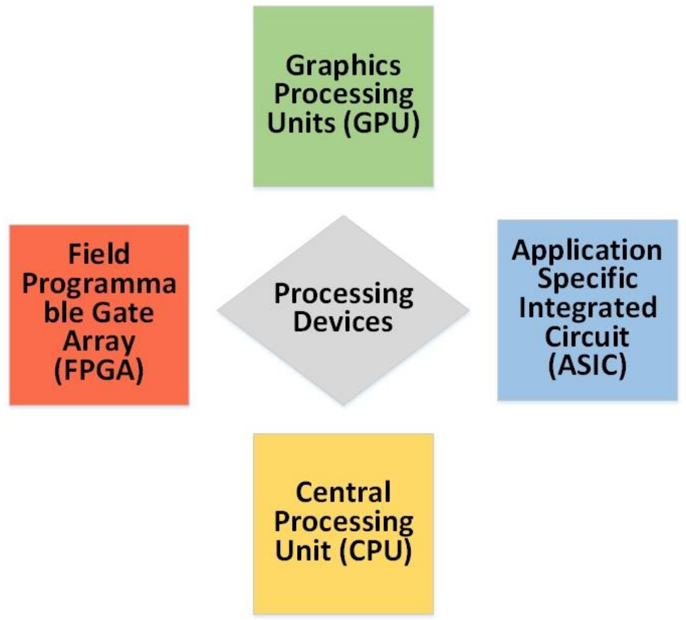
Hardware Security: Safeguarding Systems Against Attacks

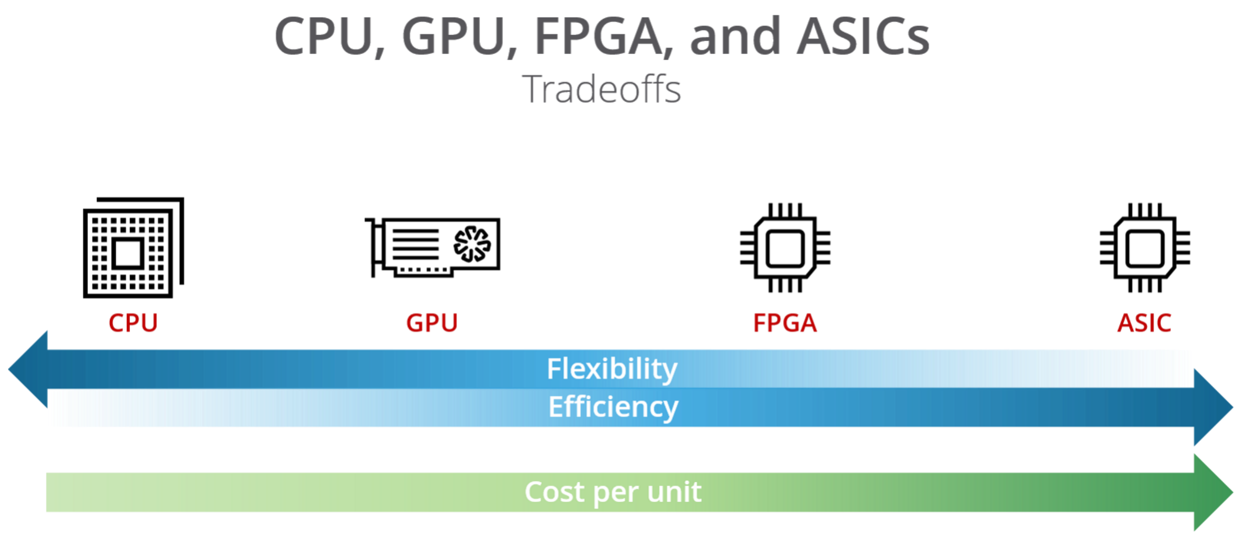
Dr. Manjith B.C.

IIIT Kottayam

Contents

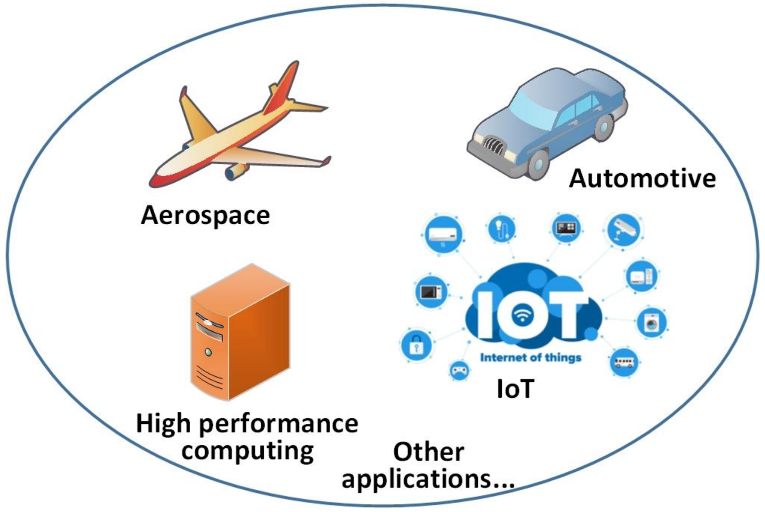
Processing Devices

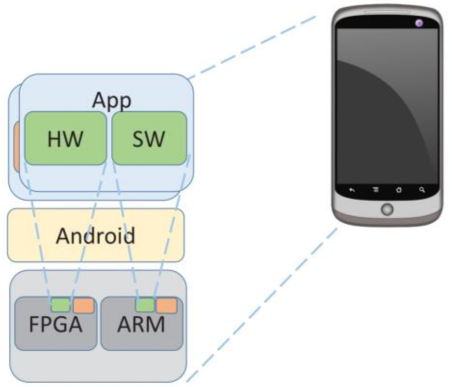

Graphics Processing Units (GPU)
Field Programmable Gate Array (FPGA)
Processing Devices
Application Specific Integrated Circuit (ASIC)
Central Processing Unit (CPU)



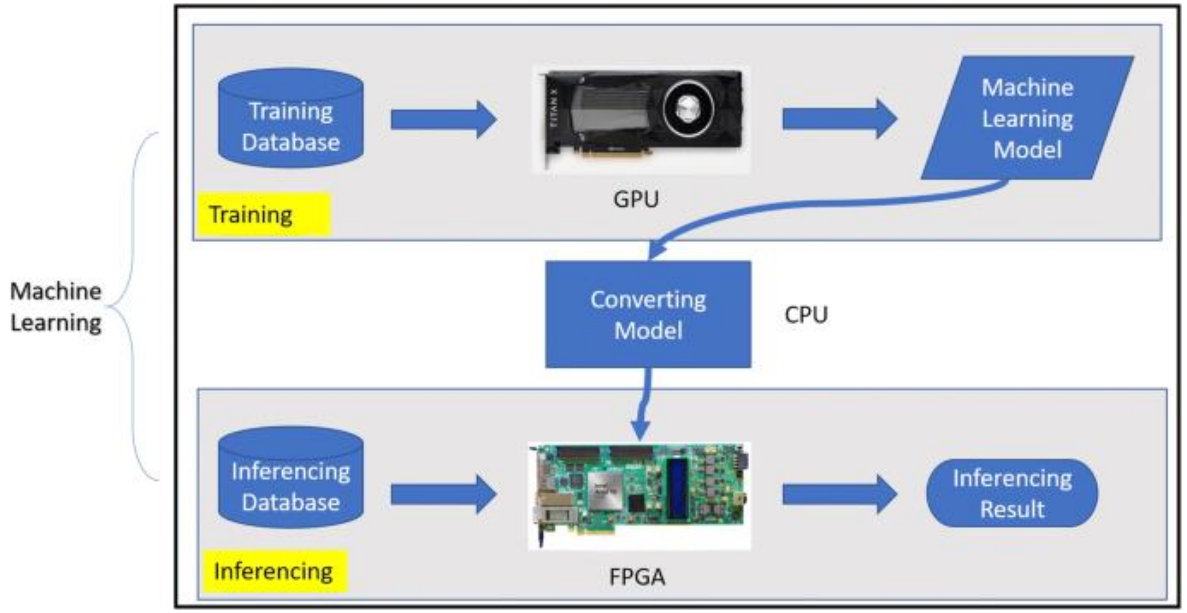
- Hardware Platforms
- Efficiency and flexibility

FPGA Applications





Smart phone with a processor (ARM) coupled with programmable hardware (FPGA).



BG27

SoC for IoT

Evolution of application areas of Field-Programmable Gate Arrays (FPGAs)

1980s -1990s
Glue Logic and Custom Circuitry

1990s - 2000s
Digital Signal Processing (DSP)

1990s - 2000s
Networking and Telecommunications

2000s - Present
Embedded Systems

2000s - Present
High-Performance Computing

2000s - Present
Software-Defined Networking (SDN)

2000s - Present
Artificial Intelligence (AI) and Machine Learning (ML)

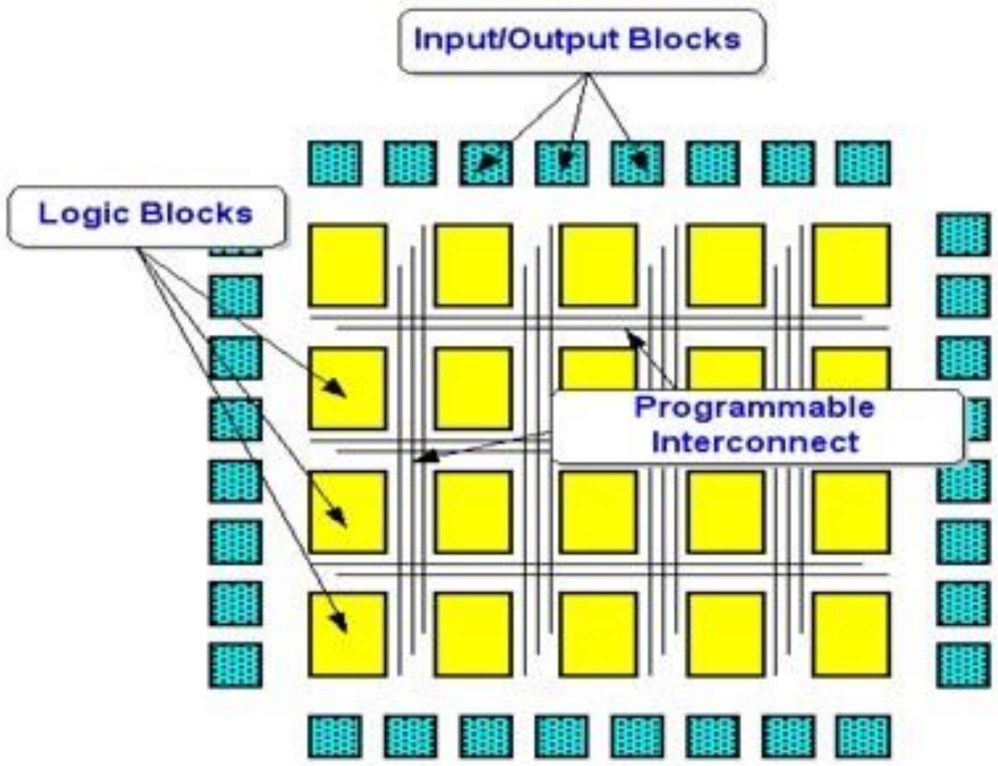
2000s - Present
Internet of Things (IoT)

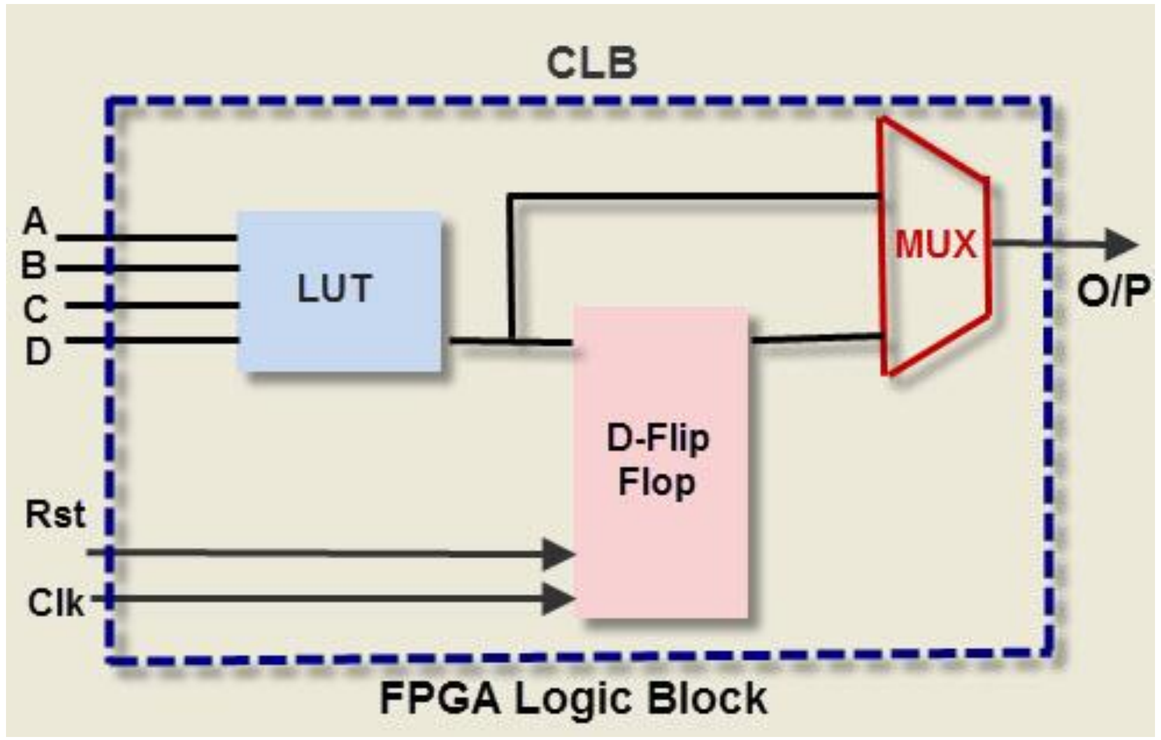
Overview of FPGA technology

Field Programmable Gate Array (FPGA) is an integrated circuit that can be programmed and reprogrammed to perform various digital logic functions.

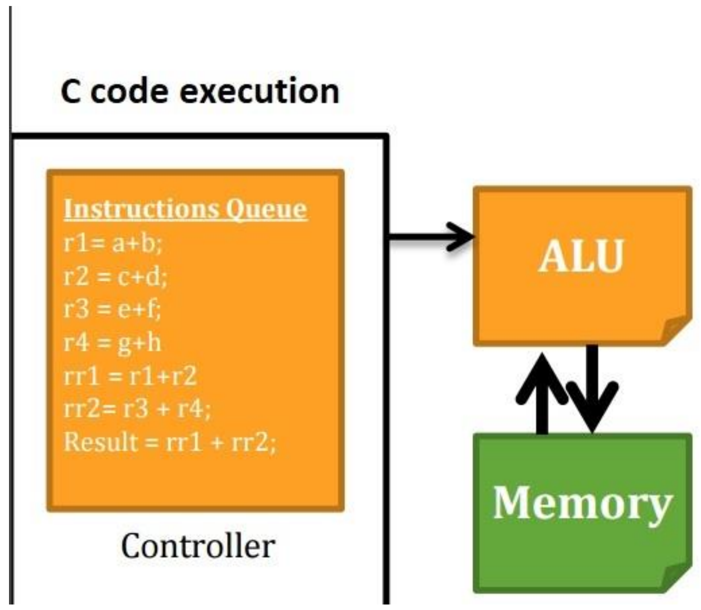
Consist of configurable logic blocks (CLBs), programmable interconnects, and input/output (I/O) blocks

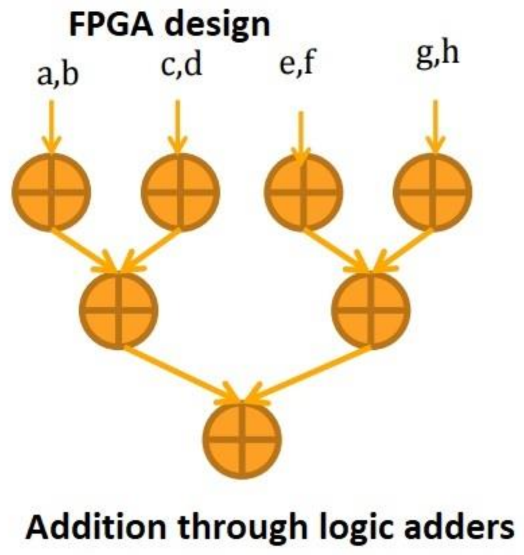
FPGA inside view





CPU and FPGA based instruction execution





How FPGAs Work

Configuration: FPGAs are programmed using hardware description languages (HDLs) such as VHDL or Verilog, or higher-level languages like C/C++ and OpenCL.

Mapping: The programmed logic is mapped onto the FPGA's CLBs and interconnects to create the desired digital circuit.

Execution: Once programmed, the FPGA performs the specified logic functions and interconnections, providing custom hardware acceleration.

Advantages of FPGAs

- Flexibility
- Performance
- Customization
- Rapid Prototyping
- Power Efficiency
- High-Speed Interfaces

Importance of hardware security in FPGA-based designs

Vulnerabilities in FPGA-Based Designs

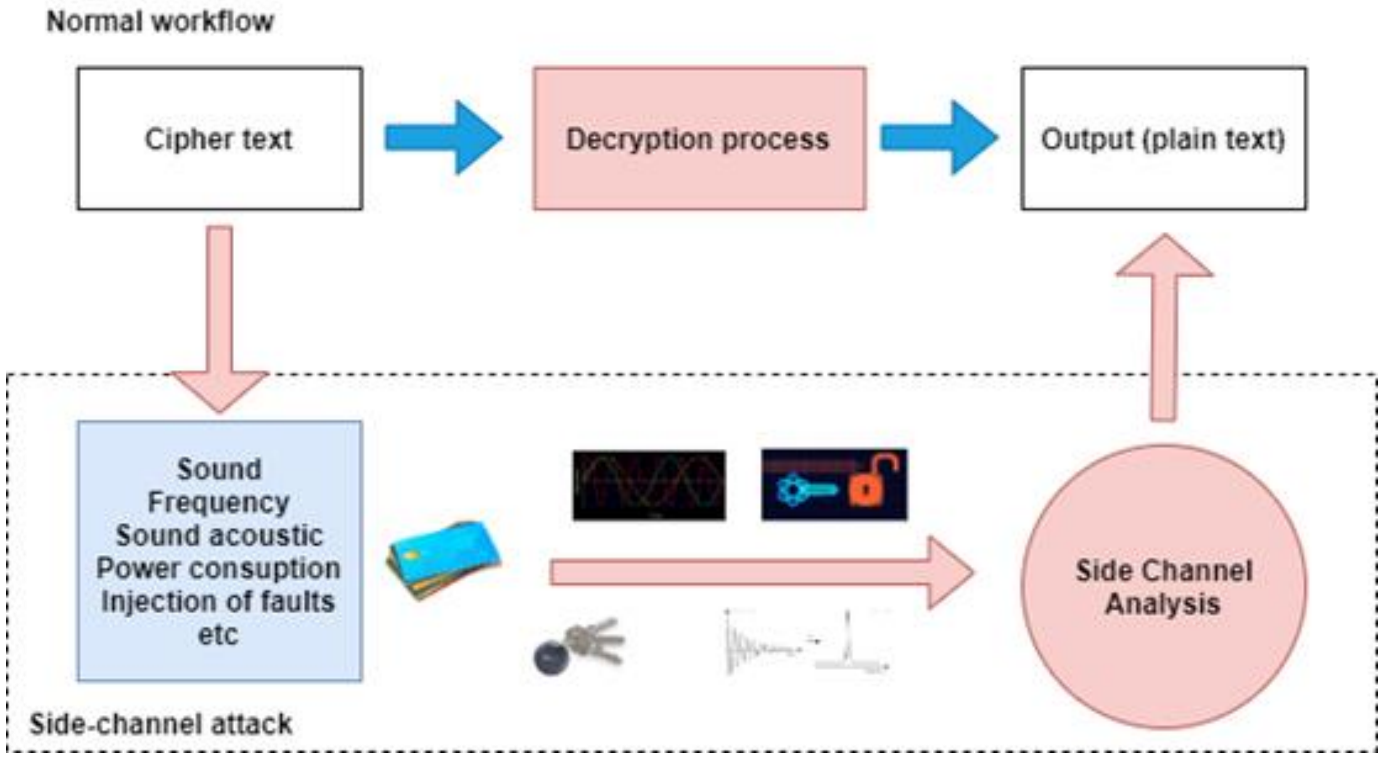
Configuration Bitstream: Unauthorized access or modification of the FPGA configuration bitstream

Side-Channel Attacks: Unintended information leakage through power consumption, electromagnetic radiation, or timing analysis

Design Flaws: Undetected design flaws or backdoors

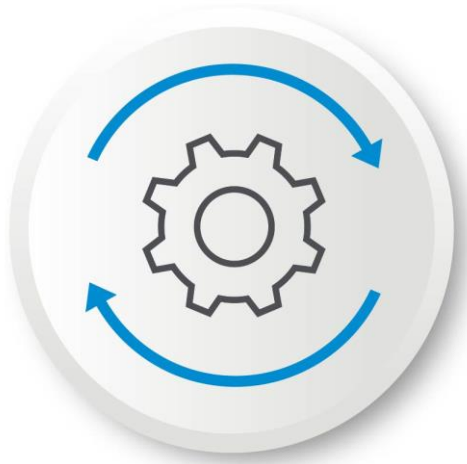
Trojans and Counterfeits: malicious functionality or unauthorized modifications

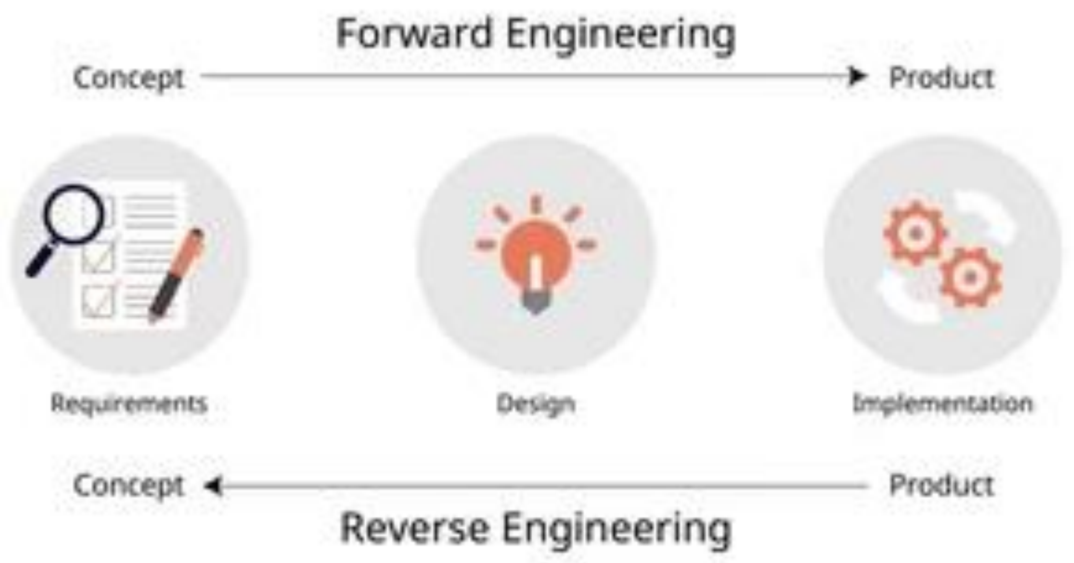
Threats and Attacks on FPGA-based Designs



Side-Channel Attacks

- Power Analysis Attacks: Analyzing power consumption patterns to extract cryptographic keys or sensitive data.
- Electromagnetic Analysis: Analyzing electromagnetic emissions to recover secret information or cryptographic keys.
- Timing Analysis: Exploiting timing variations to infer sensitive data or cryptographic keys.

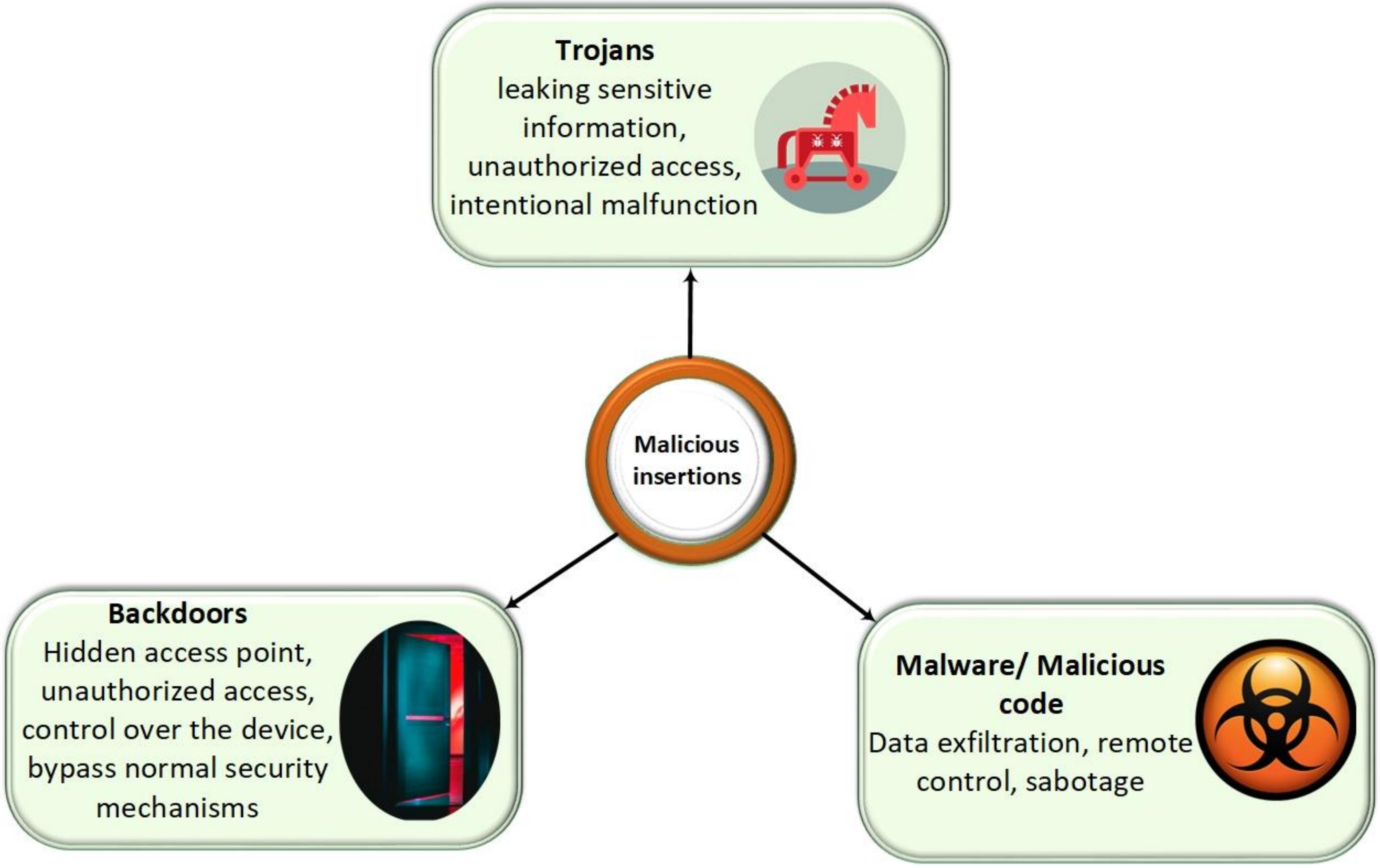




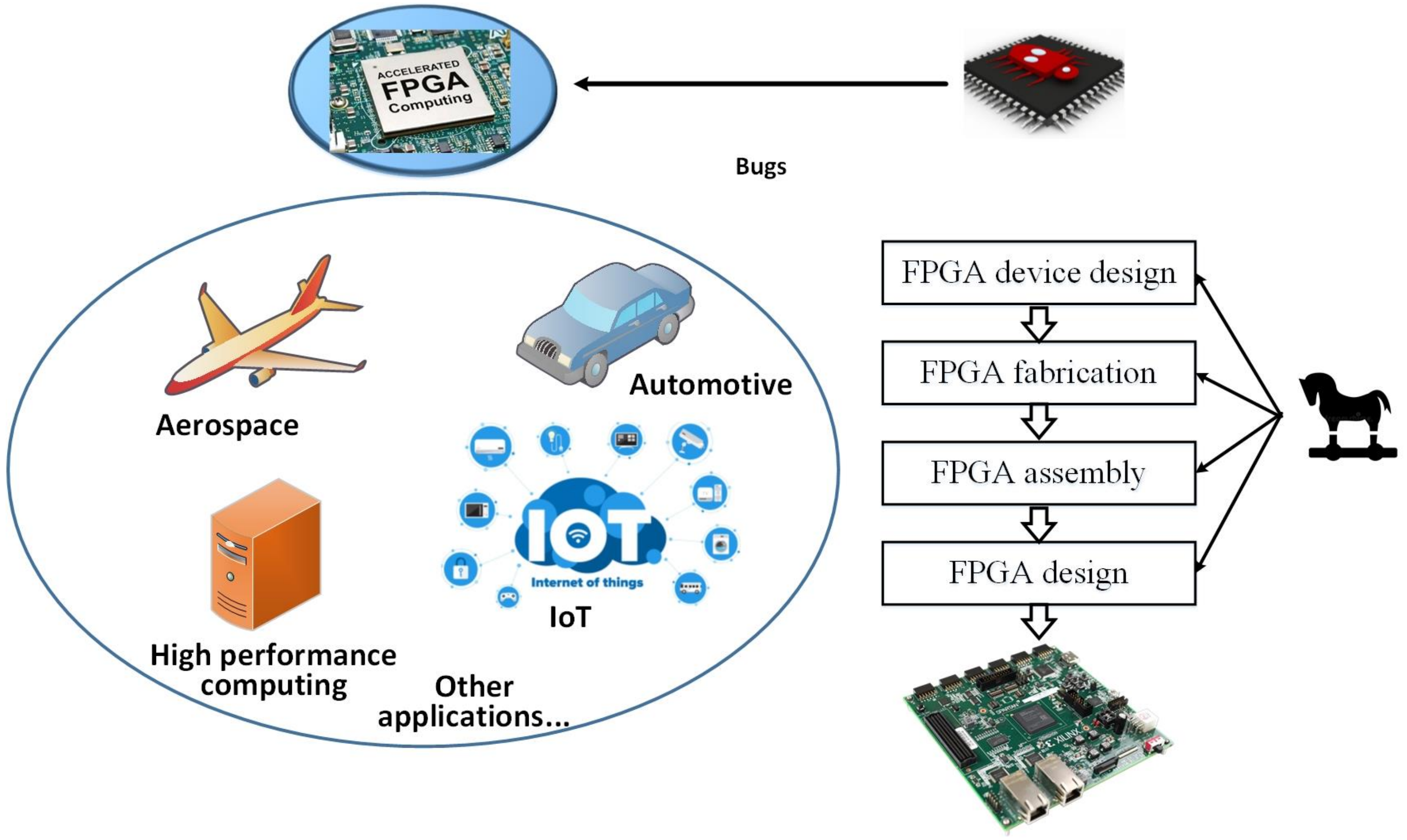
Reverse engineering attacks on FPGA designs

- Reverse engineering is the process of analyzing and understanding the design and functionality of a system by deconstructing it.
- FPGAs are susceptible to reverse engineering attacks, which can compromise intellectual property (IP) and expose sensitive information.

Malicious insertions and Trojans in FPGA designs

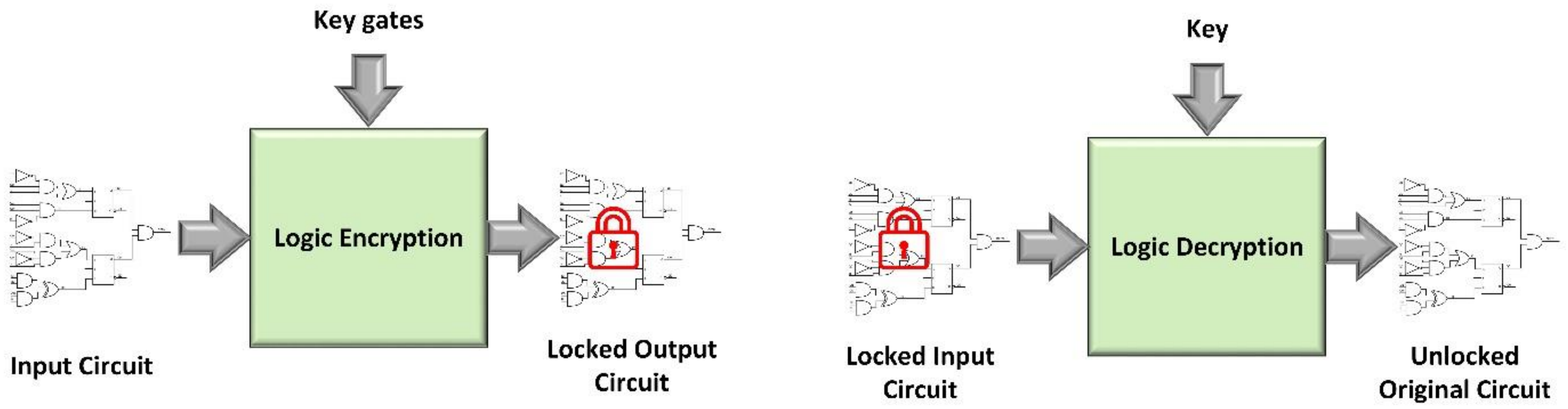

Trojans
leaking sensitive information, unauthorized access, intentional malfunction
Malicious insertions
Backdoors
Hidden access point, unauthorized access, control over the device, bypass normal security mechanisms
Malware/ Malicious code
Data exfiltration, remote control, sabotage

Malicious insertion during different phases of FPGA design



Protecting Intellectual Property (IP) in FPGA Designs

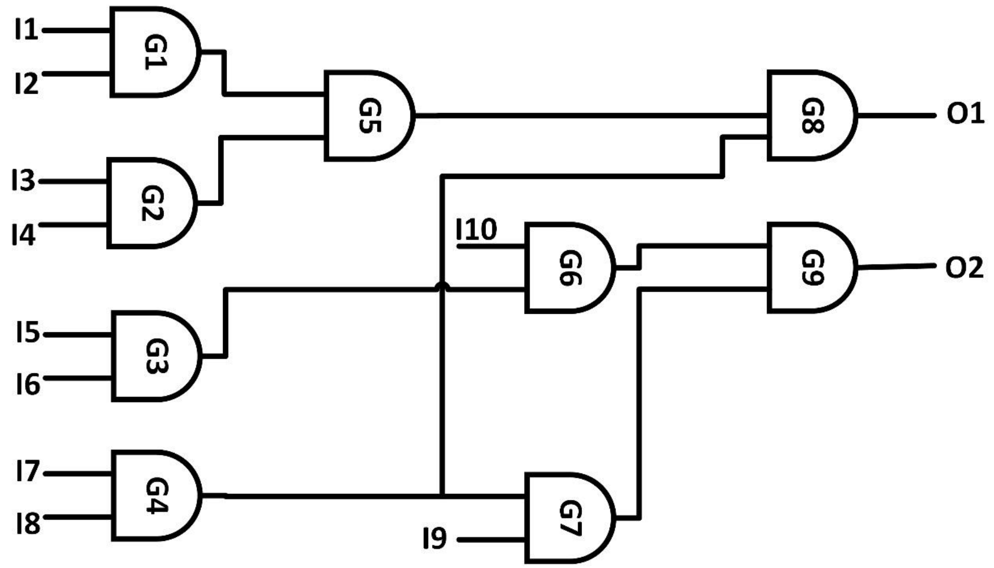
- Encryption - Encrypt the bitstream or the configuration files
- Secure Configuration - Secure boot mechanisms or authentication protocols to verify the integrity and authenticity of the configuration files
- IP Core Protection - Protect the IP cores by encrypting or obfuscating them
- Logic Locking - adding additional logic gates or obfuscating the design to make it harder for an attacker to understand or modify the design
- Watermarking - To track and identify the source of any unauthorized copies or infringements
- Secure Supply Chain - Ensure to obtain the hardware and IP from trusted sources to mitigate the risk of malicious modifications or tampering
- Access Control - Limit access to authorized personnel only



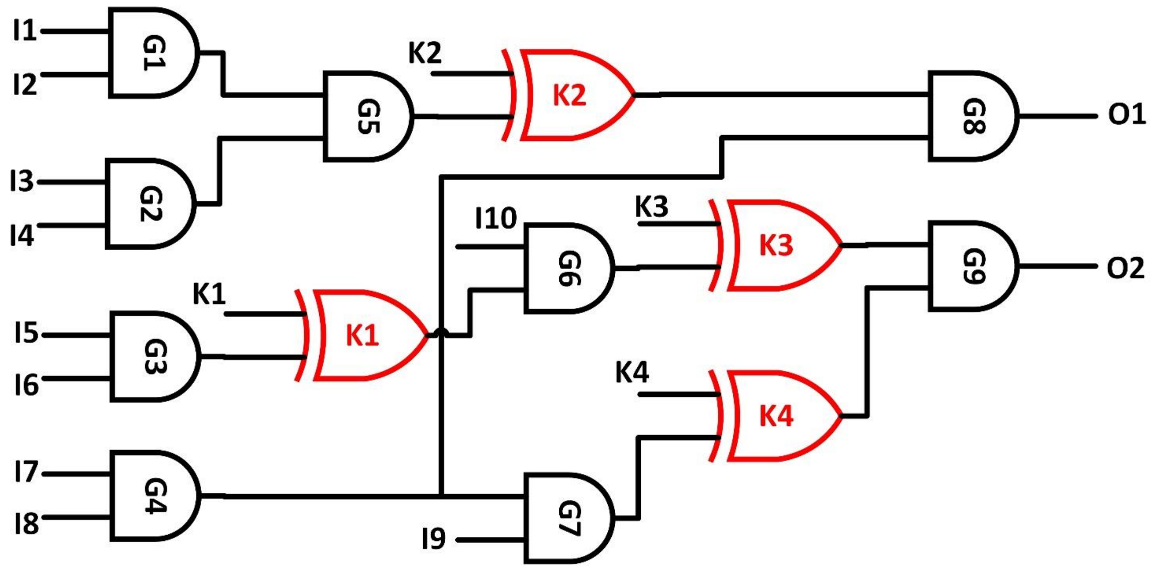
Logic Encryption

- IP owner encrypts/locks the netlist
- IP is activated by loading the correct key

Logic Encryption using XOR gates



Unencrypted Circuit



Encrypted Circuit

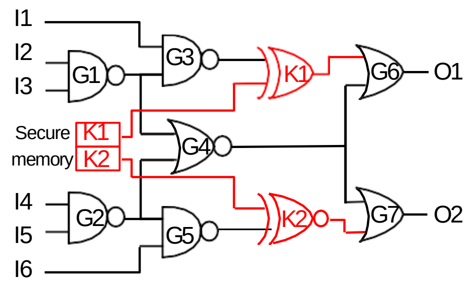
The circuit produces correct output only when the correct key is supplied

Logic Encryption Techniques

Random LE (RLE)[1]

Key-gates at random locations

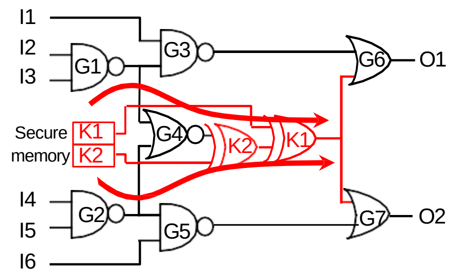
Key-gates uniformly distributed in the netlist



Fault analysis based LE (FLE)[2]

Key-gates at the most influential locations in the netlist

Key-gates tend to be localized and mostly back-to-back



1. J. Roy, et al. ,"EPIC: Ending Piracy of Integrated Circuits," DATE, 2008.
2. J. Rajendran "Fault-Analysis based Logic Encryption", TCOMP 2015

Cryptographic Accelerators in FPGA-based Hardware Security

Role of cryptographic accelerators in FPGA designs

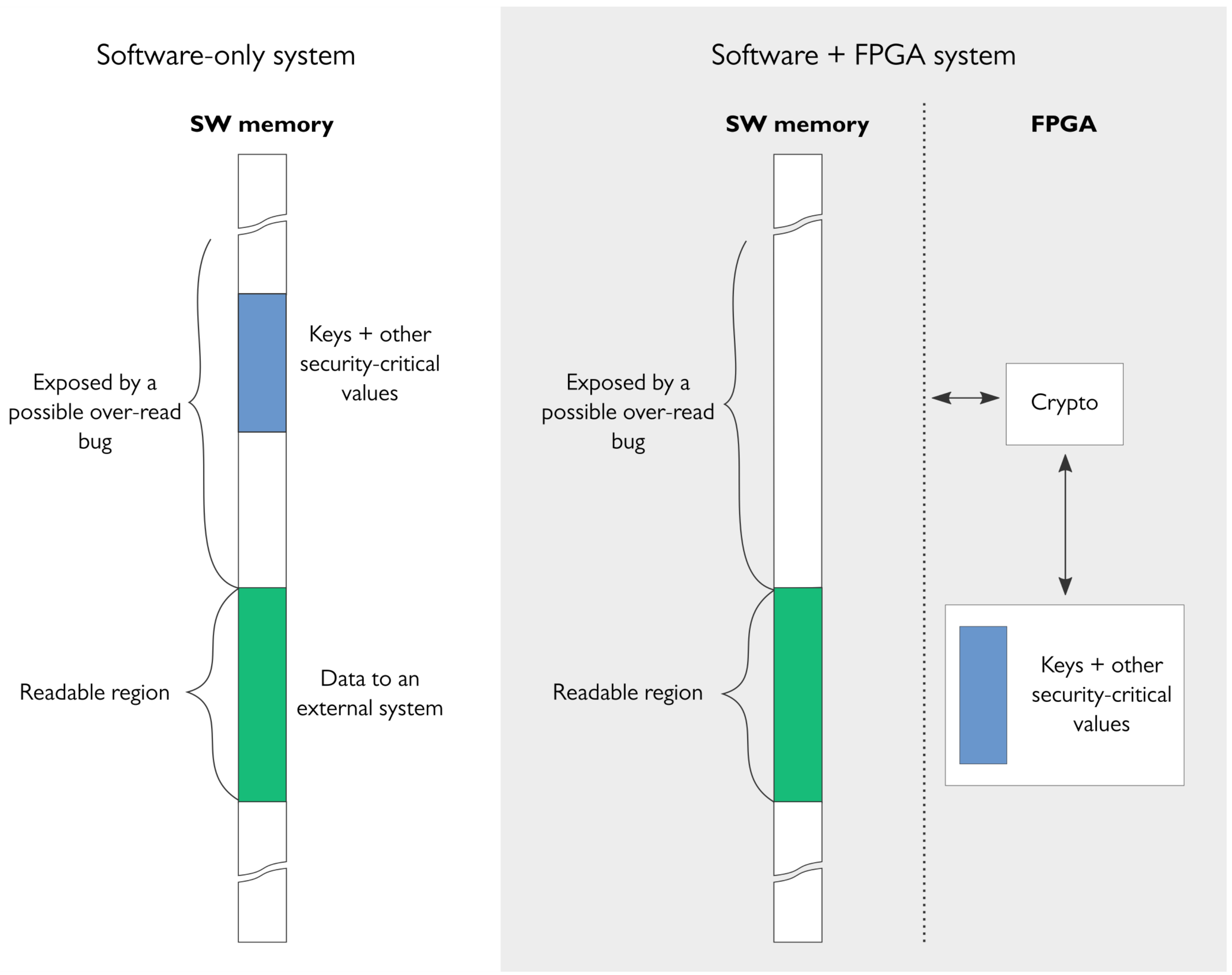
- Enhanced Performance- Parallel processing enabling efficient data processing in real-time or high-throughput scenarios
- Hardware-level Security-Protected key storage, tamper-resistant designs, and physical security mechanisms
- Low Latency and Deterministic Timing - Critical for time-critical applications
- Power Efficiency - FPGA hardware accelerators can achieve higher performance per watt
- Flexible Algorithm Support – Allows customization and adaptability to specific application requirements

Role of cryptographic accelerators in FPGA designs

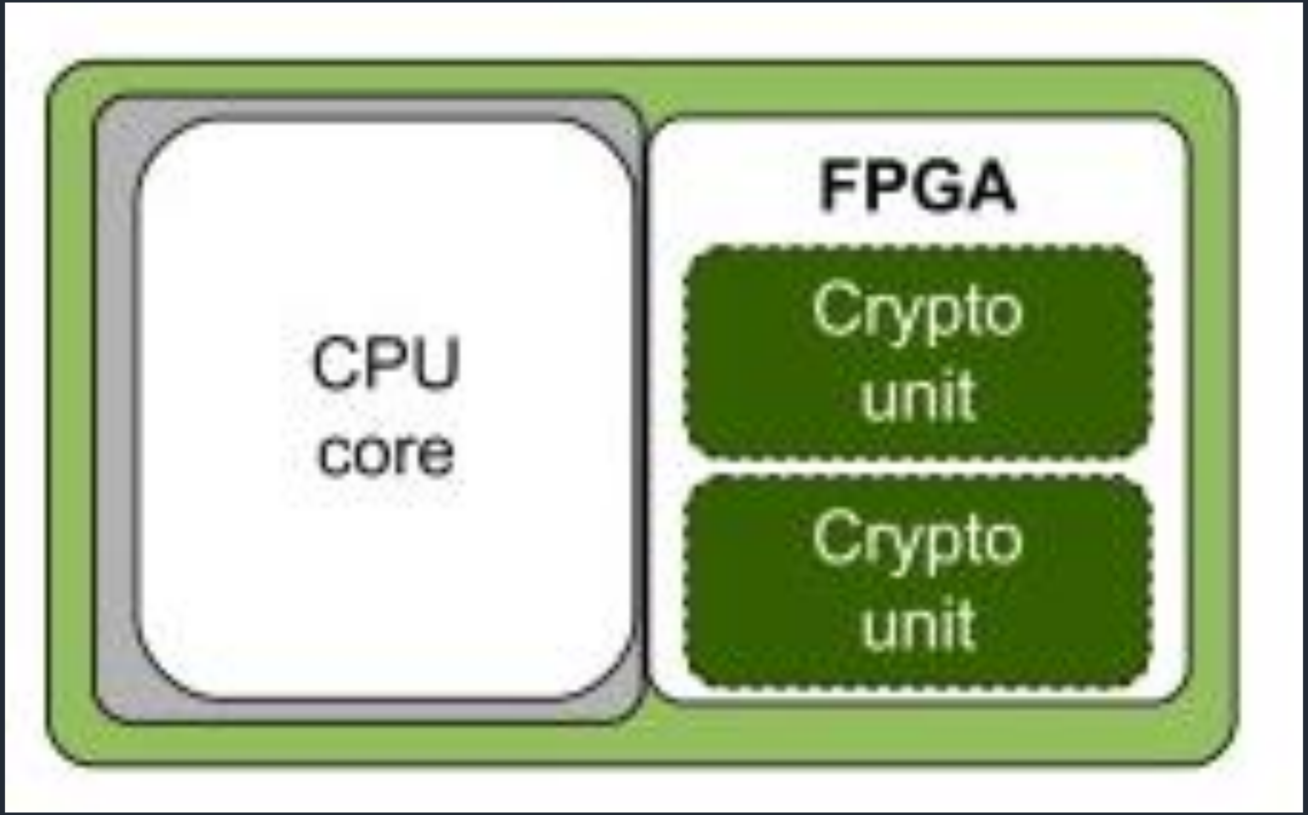
Design Flexibility and Reconfigurability - Accommodate new cryptographic standards, or incorporate custom cryptographic functions as needed

Resource Optimization - Careful design and optimization techniques, such as pipelining, parallel processing, and resource sharing

Isolation of critical cryptography and keys


Software-only system
SW memory
Keys + other security-critical values
Exposed by a possible over-read bug
Readable region
Data to an external system
Software + FPGA system
SW memory
FPGA
Exposed by a possible over-read bug
Crypto
Readable region
Keys + other security-critical values

Implementing encryption, decryption, and authentication in FPGA hardware



Case Study

128 bit AES encryption

Designed by Rijmen-Daemen in Belgium

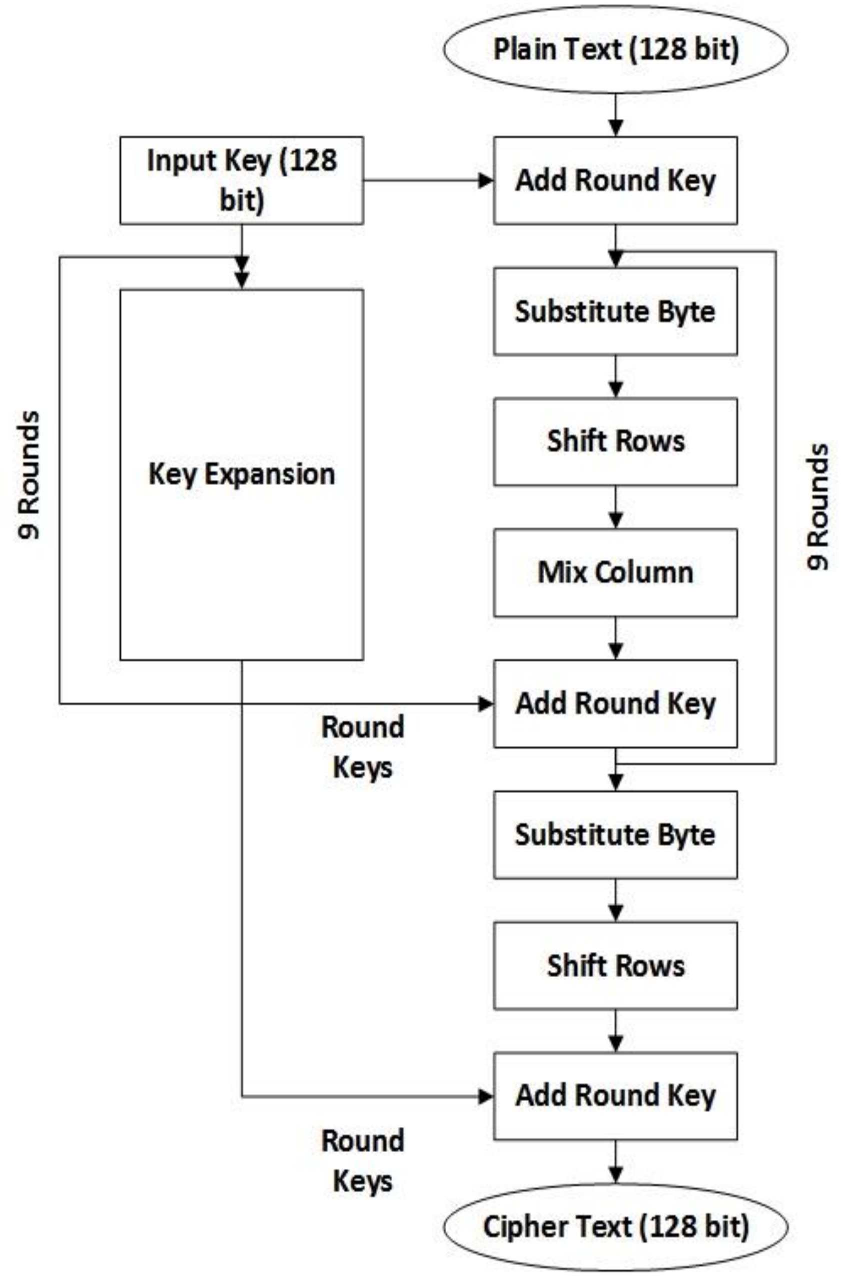
Has 128 bit keys, 128 bit data

An iterative rather than Feistel cipher

- processes data as block of 4 columns of 4 bytes
- operates on entire data block in every round

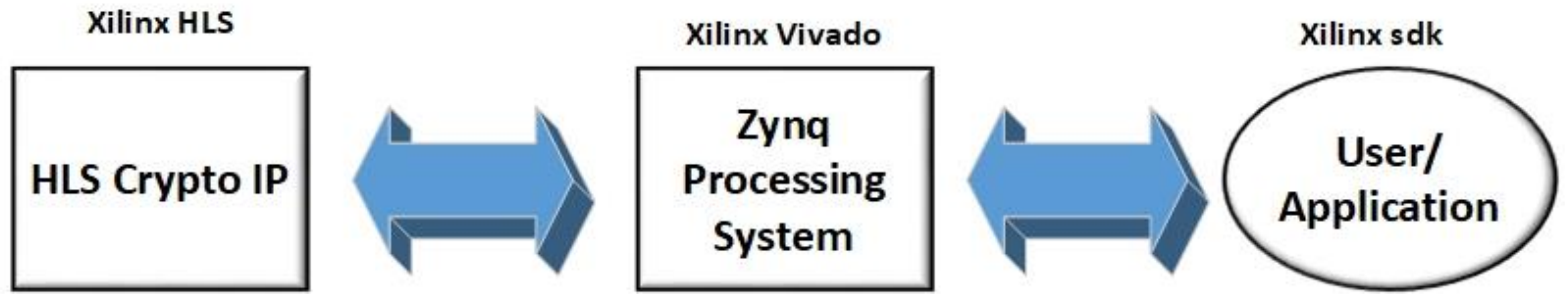
Designed to have:

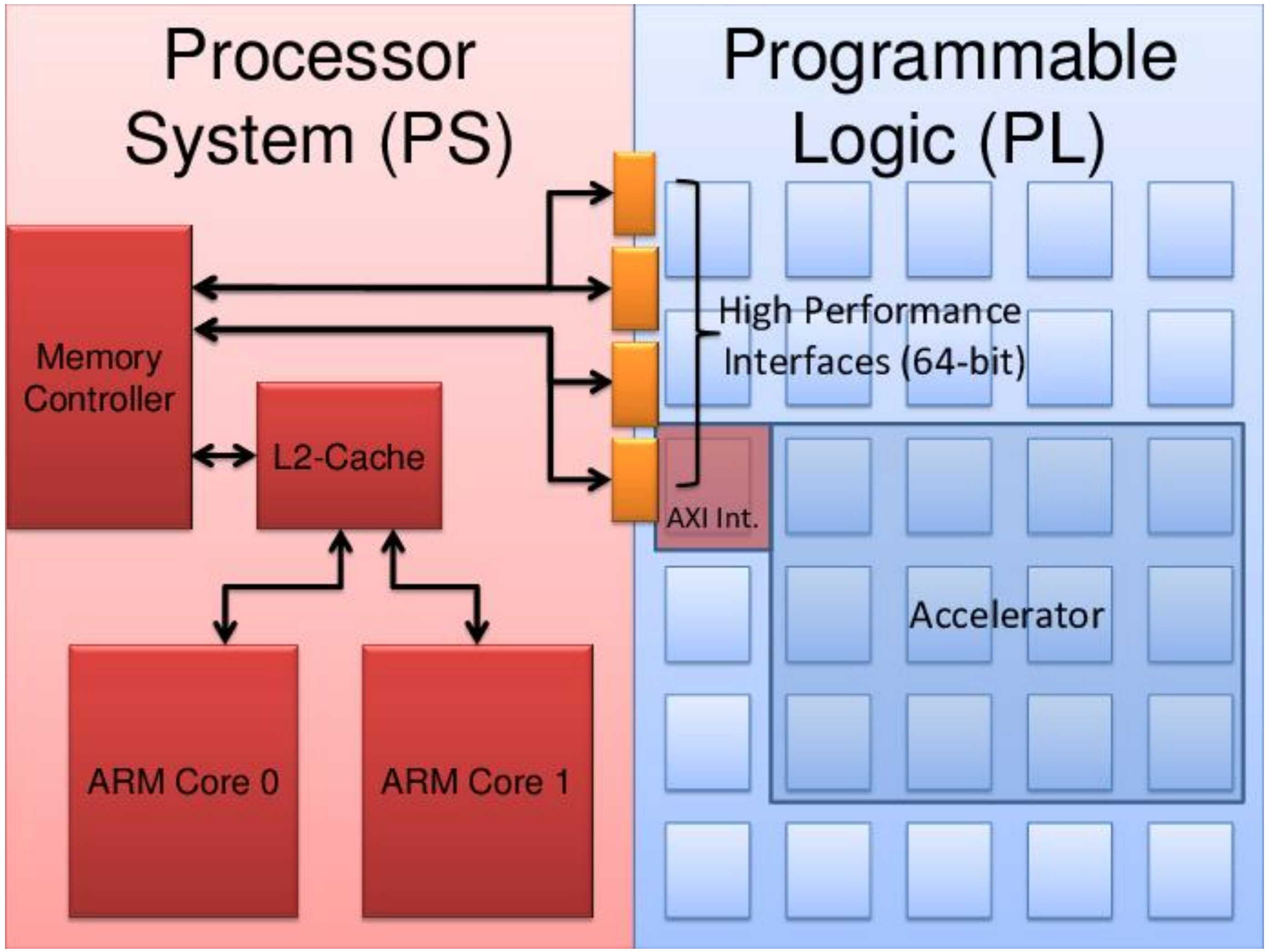
- resistance against known attacks
- speed and code compactness on many CPUs
- design simplicity



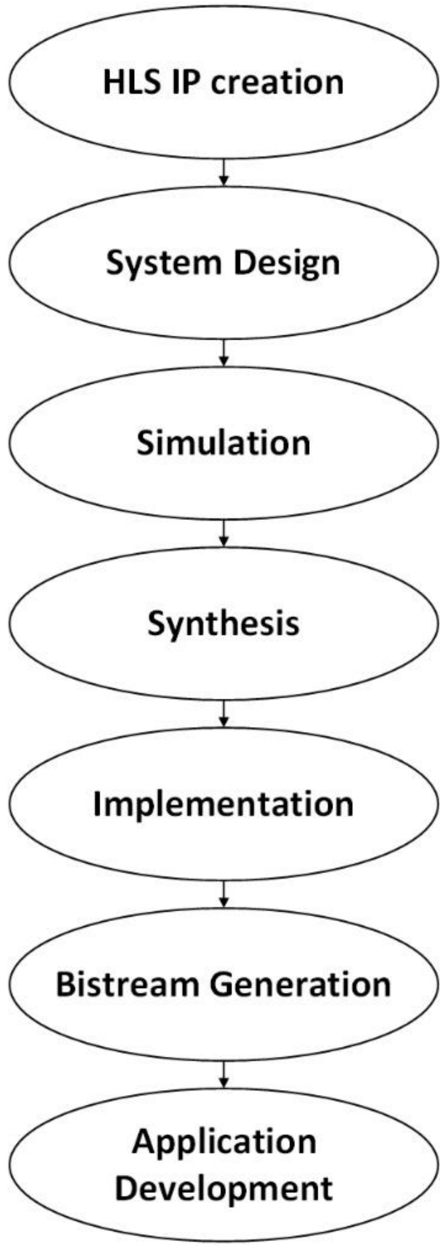
Hardware implementation on FPGA

- Tools
  - Xilinx HLS (high level synthesis of algorithm in C, C++ and System C)
  - Xilinx Vivado (synthesis and analysis of HDL designs, generating binaries for FPGA)
  - Xilinx sdk (development of embedded software applications for the design)

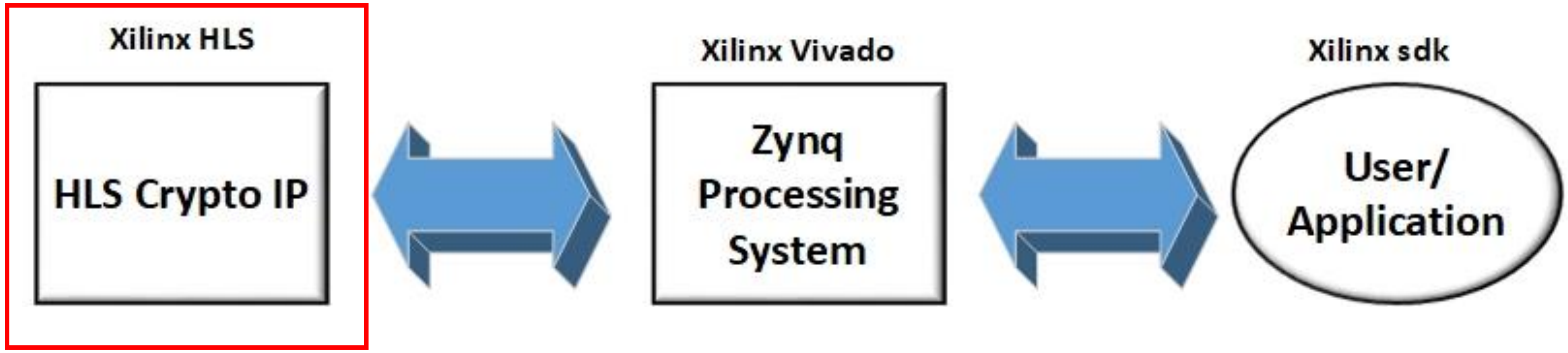



Processor System (PS)
Programmable Logic (PL)
Memory Controller
L2-Cache
ARM Core 0
ARM Core 1
High Performance Interfaces (64-bit)
AXI Int.
Accelerator

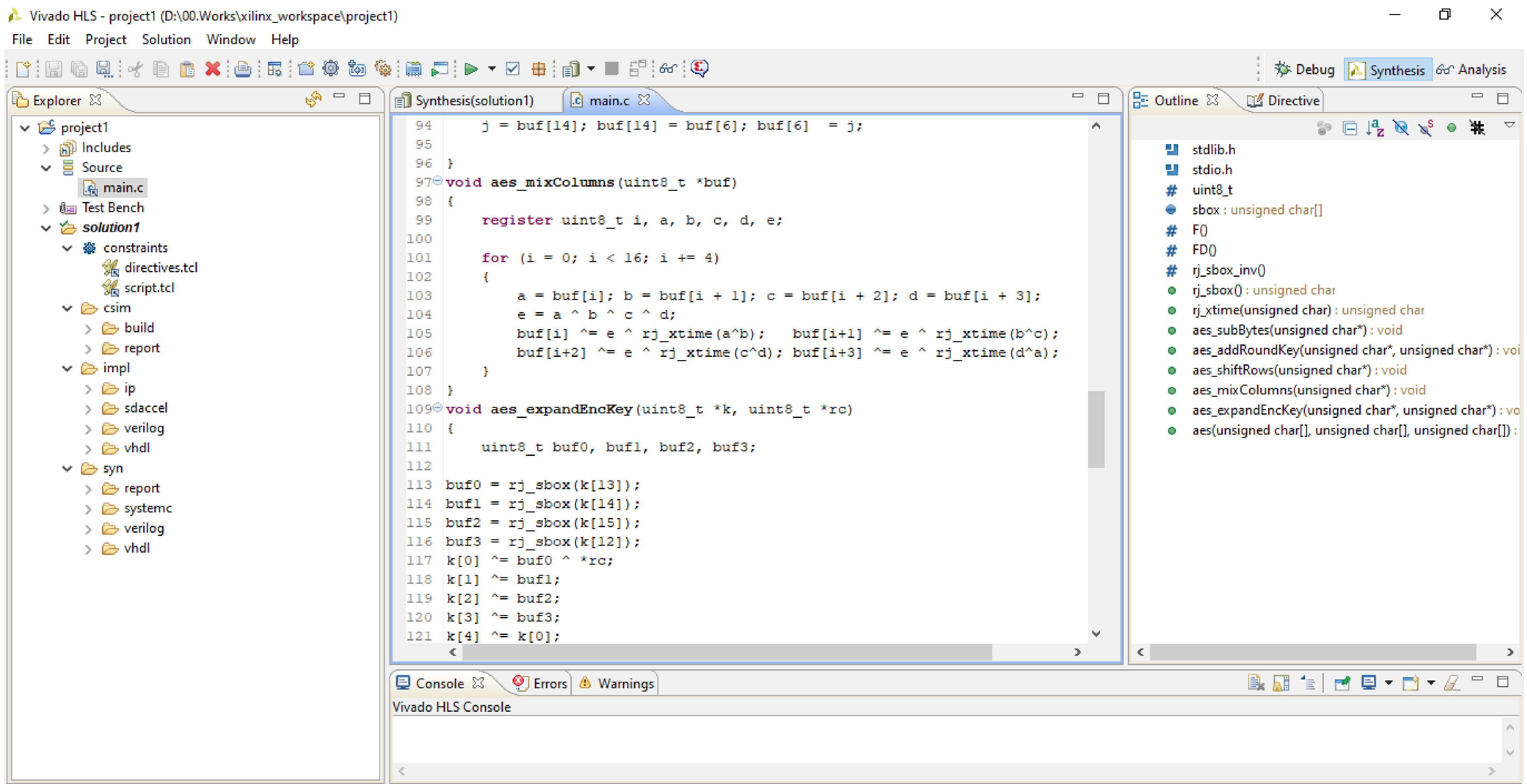
FPGA Design flow



Implementation



Xilinx HLS



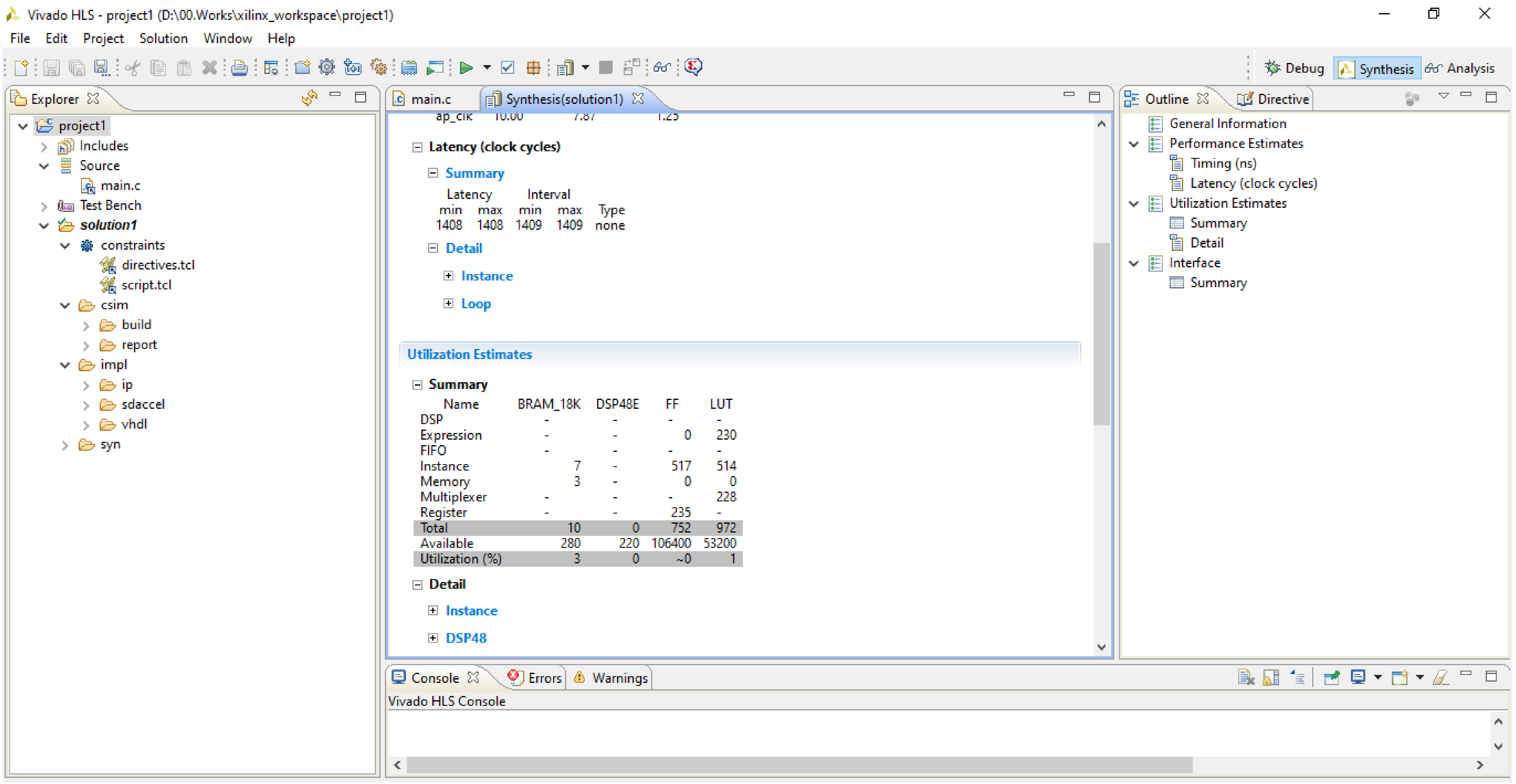
Vivado HLS - project1 (D:\00.Works\xilinx_workspace\project1)

```
project1
    Includes
    Source
        main.c
    Test Bench
    solution1
        constraints
            directives.tcl
            script.tcl
        csim
            build
            report
        impl
            ip
            sdaccel
            verilog
            vhdl
        syn
            report
            systemc
            verilog
            vhdl
```

```c
    j = buf[14]; buf[14] = buf[6]; buf[6]  = j;

}
void aes_mixColumns(uint8_t *buf)
{
    register uint8_t i, a, b, c, d, e;

    for (i = 0; i < 16; i += 4)
    {
        a = buf[i]; b = buf[i + 1]; c = buf[i + 2]; d = buf[i + 3];
        e = a ^ b ^ c ^ d;
        buf[i] ^= e ^ rj_xtime(a^b);   buf[i+1] ^= e ^ rj_xtime(b^c);
        buf[i+2] ^= e ^ rj_xtime(c^d); buf[i+3] ^= e ^ rj_xtime(d^a);
    }
}
void aes_expandEncKey(uint8_t *k, uint8_t *rc)
{
    uint8_t buf0, buf1, buf2, buf3;

buf0 = rj_sbox(k[13]);
buf1 = rj_sbox(k[14]);
buf2 = rj_sbox(k[15]);
buf3 = rj_sbox(k[12]);
k[0] ^= buf0 ^ *rc;
k[1] ^= buf1;
k[2] ^= buf2;
k[3] ^= buf3;
k[4] ^= k[0];
```

```
stdlib.h
stdio.h
uint8_t
sbox : unsigned char[]
F()
FD()
rj_sbox_inv()
rj_sbox() : unsigned char
rj_xtime(unsigned char) : unsigned char
aes_subBytes(unsigned char*) : void
aes_addRoundKey(unsigned char*, unsigned char*) : voi
aes_shiftRows(unsigned char*) : void
aes_mixColumns(unsigned char*) : void
aes_expandEncKey(unsigned char*, unsigned char*) : vo
aes(unsigned char[], unsigned char[], unsigned char[]) :
```

Vivado HLS Console



```
ap_clk  10.00  7.87  1.25
```

Latency (clock cycles)

Summary

```
 Latency     Interval
min   max   min   max   Type
1408  1408  1409  1409  none
```

Detail

Instance

Loop

Utilization Estimates

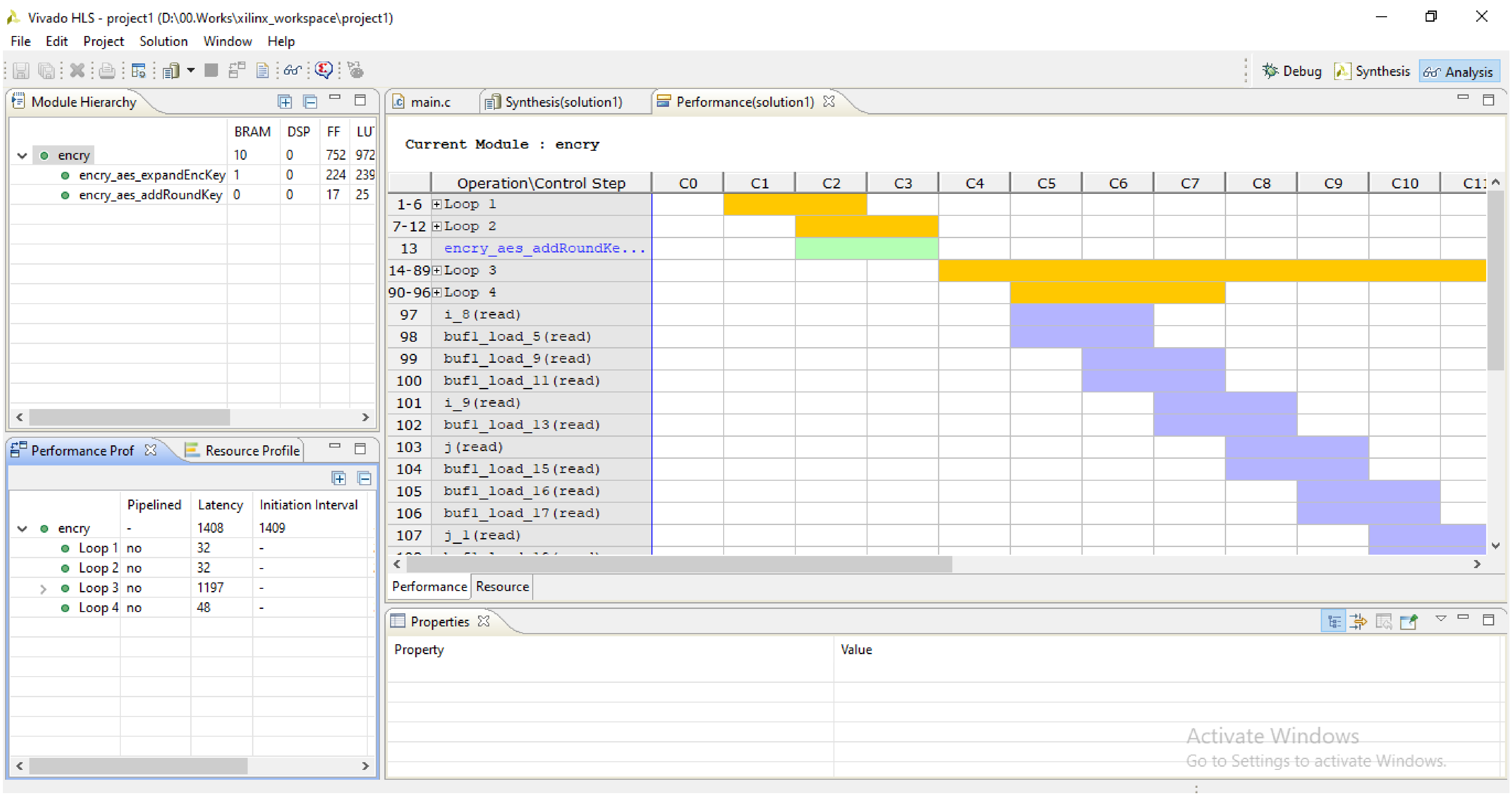
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	230
FIFO	-	-	-	-
Instance	7	-	517	514
Memory	3	-	0	0
Multiplexer	-	-	-	228
Register	-	-	235	-
Total	10	0	752	972
Available	280	220	106400	53200
Utilization (%)	3	0	~0	1

Detail

Instance

DSP48



Vivado HLS - project1 (D:\00.Works\xilinx_workspace\project1)
File Edit Project Solution Window Help
Debug Synthesis Analysis
Module Hierarchy
main.c Synthesis(solution1) Performance(solution1)
Performance Prof Resource Profile
Performance Resource
Properties

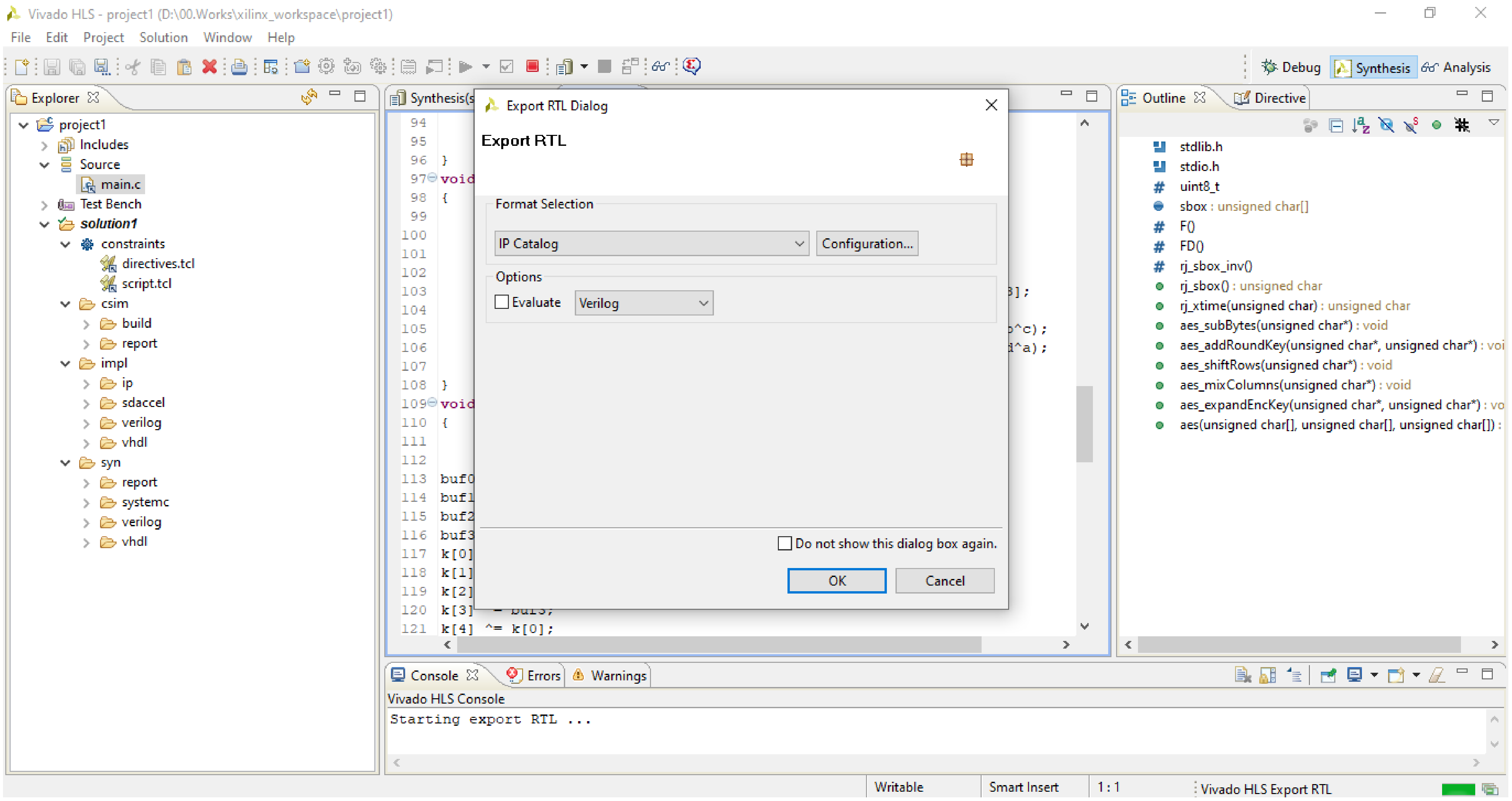
	BRAM	DSP	FF	LU
encry	10	0	752	972
encry_aes_expandEncKey	1	0	224	239
encry_aes_addRoundKey	0	0	17	25

	Pipelined	Latency	Initiation Interval
encry	-	1408	1409
Loop 1	no	32	-
Loop 2	no	32	-
Loop 3	no	1197	-
Loop 4	no	48	-

Current Module : encry

	Operation\Control Step	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C1
1-6	Loop 1		🟨	🟨									
7-12	Loop 2			🟨	🟨								
13	encry_aes_addRoundKe...			🟩	🟩								
14-89	Loop 3					🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨
90-96	Loop 4						🟨	🟨	🟨				
97	i_8(read)						🟦	🟦					
98	bufl_load_5(read)						🟦	🟦					
99	bufl_load_9(read)							🟦	🟦				
100	bufl_load_11(read)							🟦	🟦				
101	i_9(read)								🟦	🟦			
102	bufl_load_13(read)								🟦	🟦			
103	j(read)									🟦	🟦		
104	bufl_load_15(read)									🟦	🟦		
105	bufl_load_16(read)										🟦	🟦	
106	bufl_load_17(read)										🟦	🟦	
107	j_1(read)											🟦	🟦

Property	Value


Vivado HLS - project1 (D:\00.Works\xilinx_workspace\project1)
File Edit Project Solution Window Help
Debug
Synthesis
Analysis
Explorer
project1
Includes
Source
main.c
Test Bench
solution1
constraints
directives.tcl
script.tcl
csim
build
report
impl
ip
sdaccel
verilog
vhdl
syn
report
systemc
verilog
vhdl
Synthesis(s
Outline
Directive
stdlib.h
stdio.h
uint8_t
sbox : unsigned char[]
F()
FD()
rj_sbox_inv()
rj_sbox() : unsigned char
rj_xtime(unsigned char) : unsigned char
aes_subBytes(unsigned char*) : void
aes_addRoundKey(unsigned char*, unsigned char*) : voi
aes_shiftRows(unsigned char*) : void
aes_mixColumns(unsigned char*) : void
aes_expandEncKey(unsigned char*, unsigned char*) : vo
aes(unsigned char[], unsigned char[], unsigned char[]) :
Console
Errors
Warnings
Writable
Smart Insert
1 : 1
Vivado HLS Export RTL

Export RTL Dialog

Export RTL

Format Selection

IP Catalog | Configuration...

Options

☐ Evaluate | Verilog

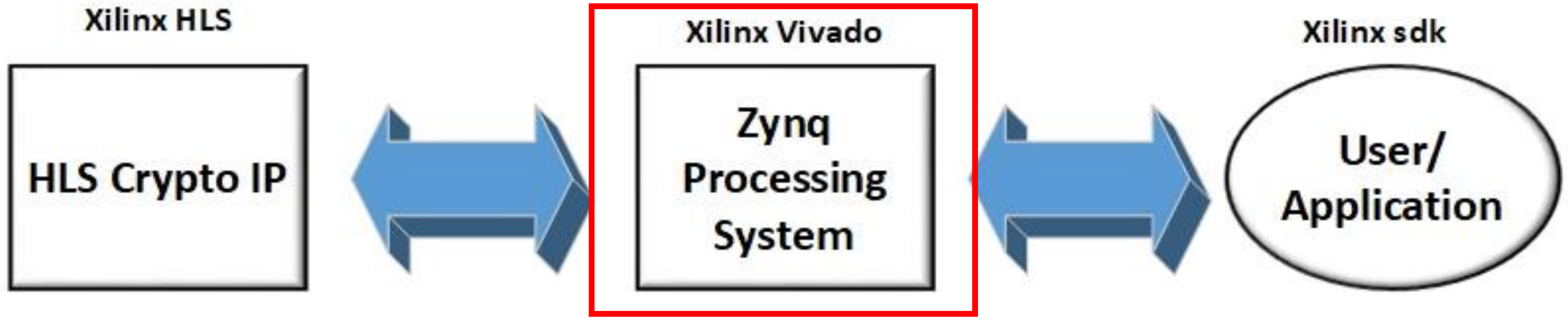
☐ Do not show this dialog box again.

OK | Cancel

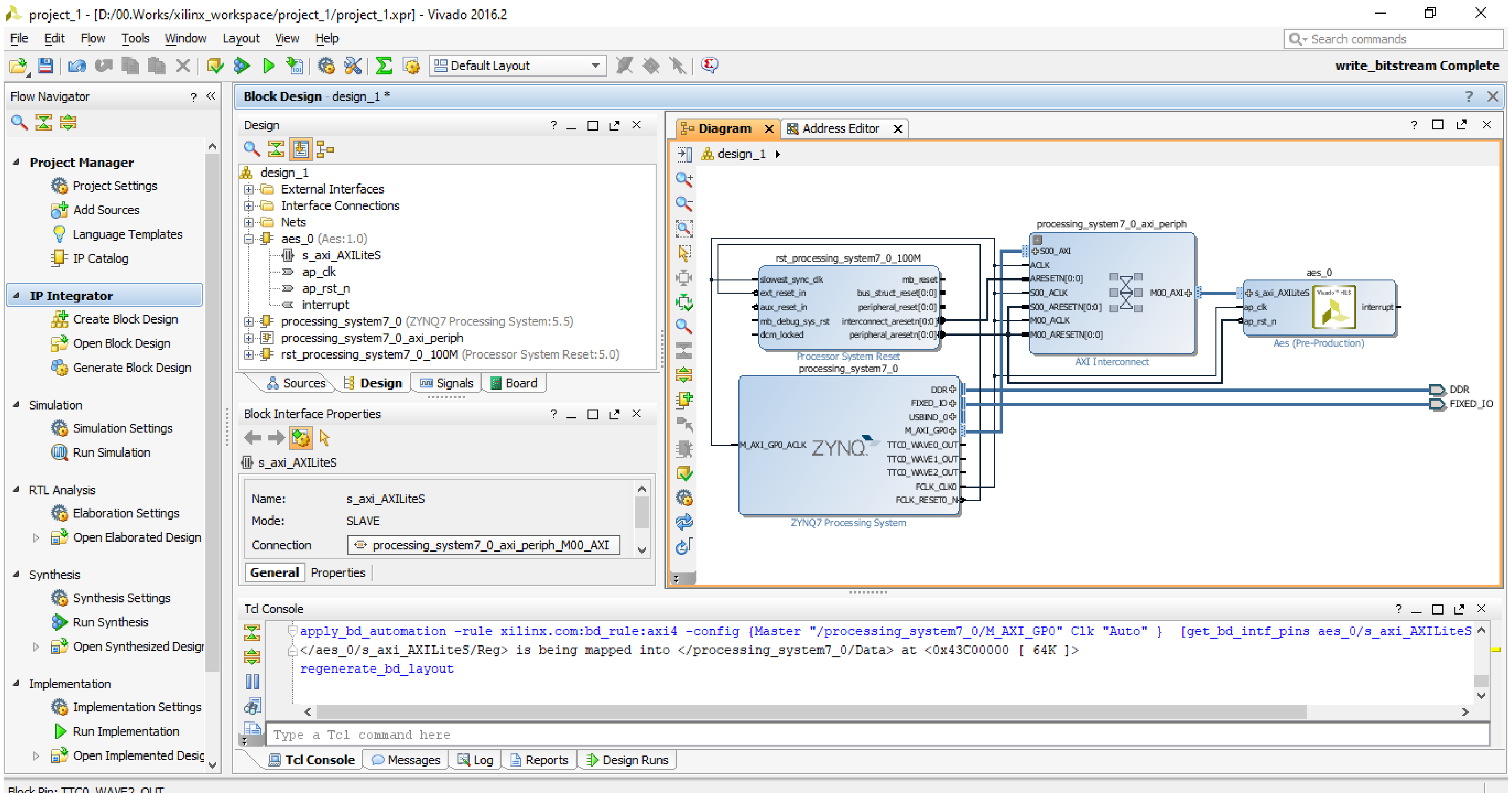
Vivado HLS Console

```
Starting export RTL ...
```

Implementation



Xilinx Vivado

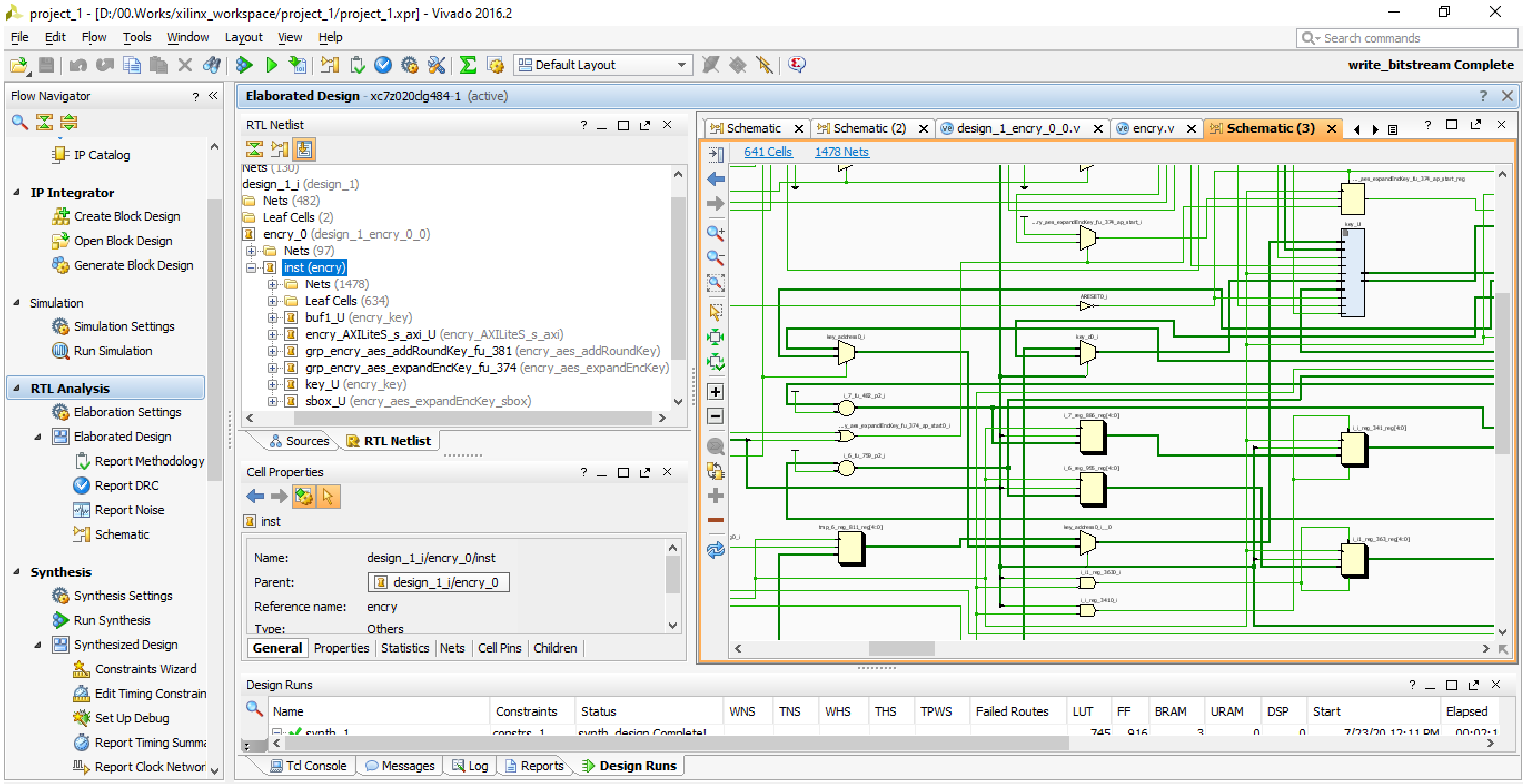


Name: s_axi_AXILiteS
Mode: SLAVE
Connection: processing_system7_0_axi_periph_M00_AXI

```
apply_bd_automation -rule xilinx.com:bd_rule:axi4 -config {Master "/processing_system7_0/M_AXI_GP0" Clk "Auto" }  [get_bd_intf_pins aes_0/s_axi_AXILiteS
</aes_0/s_axi_AXILiteS/Reg> is being mapped into </processing_system7_0/Data> at <0x43C00000 [ 64K ]>
regenerate_bd_layout
```

Type a Tcl command here

Schematic



project_1 - [D:/00.Works/xilinx_workspace/project_1/project_1.xpr] - Vivado 2016.2

write_bitstream Complete

Elaborated Design - xc7z020clg484-1 (active)

RTL Netlist

- Nets (130)
- design_1_i (design_1)
  - Nets (482)
  - Leaf Cells (2)
  - encry_0 (design_1_encry_0_0)
    - Nets (97)
    - inst (encry)
      - Nets (1478)
      - Leaf Cells (634)
      - buf1_U (encry_key)
      - encry_AXILiteS_s_axi_U (encry_AXILiteS_s_axi)
      - grp_encry_aes_addRoundKey_fu_381 (encry_aes_addRoundKey)
      - grp_encry_aes_expandEncKey_fu_374 (encry_aes_expandEncKey)
      - key_U (encry_key)
      - sbox_U (encry_aes_expandEncKey_sbox)

Cell Properties

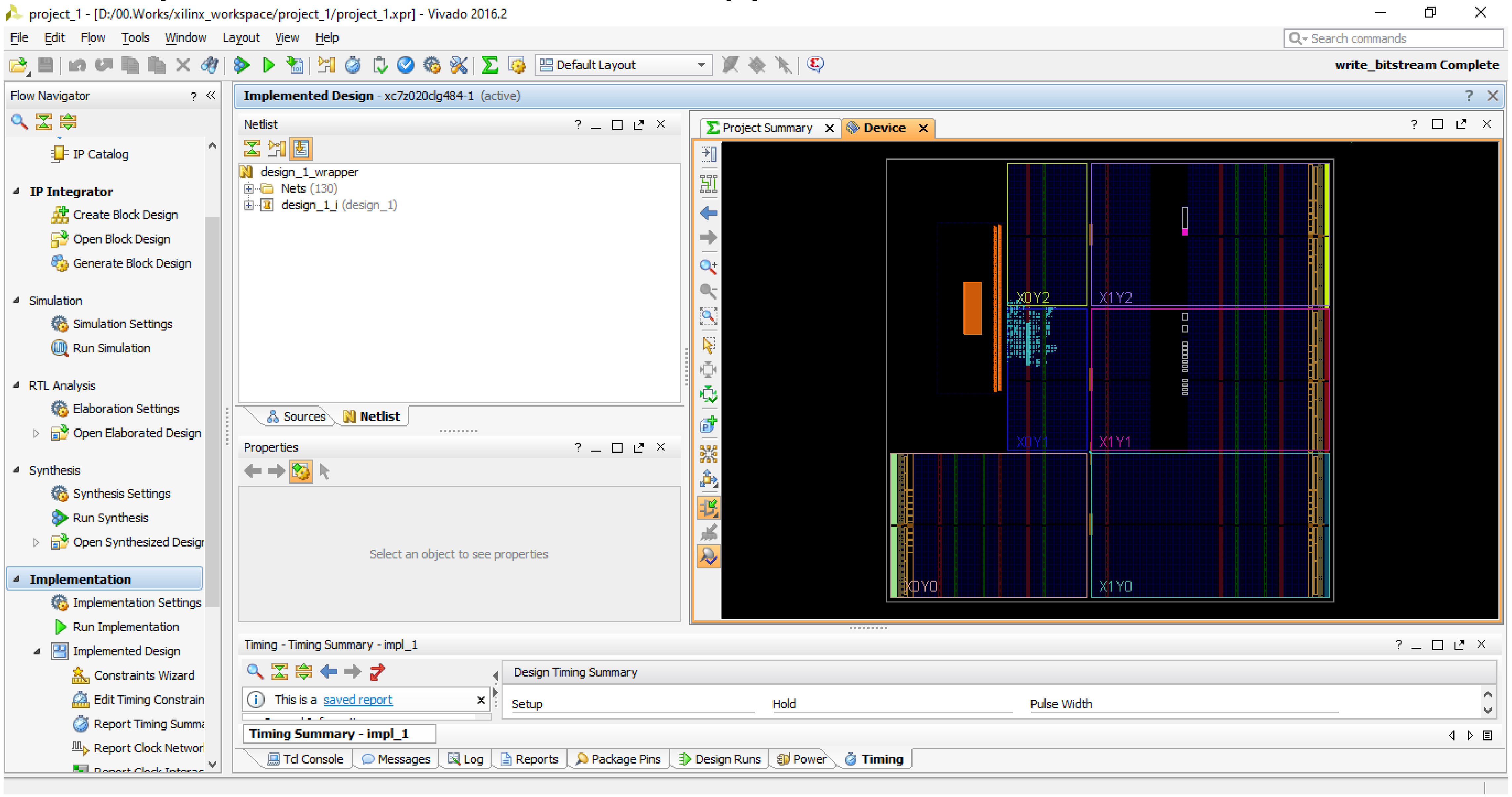
inst

Name:	design_1_i/encry_0/inst
Parent:	design_1_i/encry_0
Reference name:	encry
Type:	Others

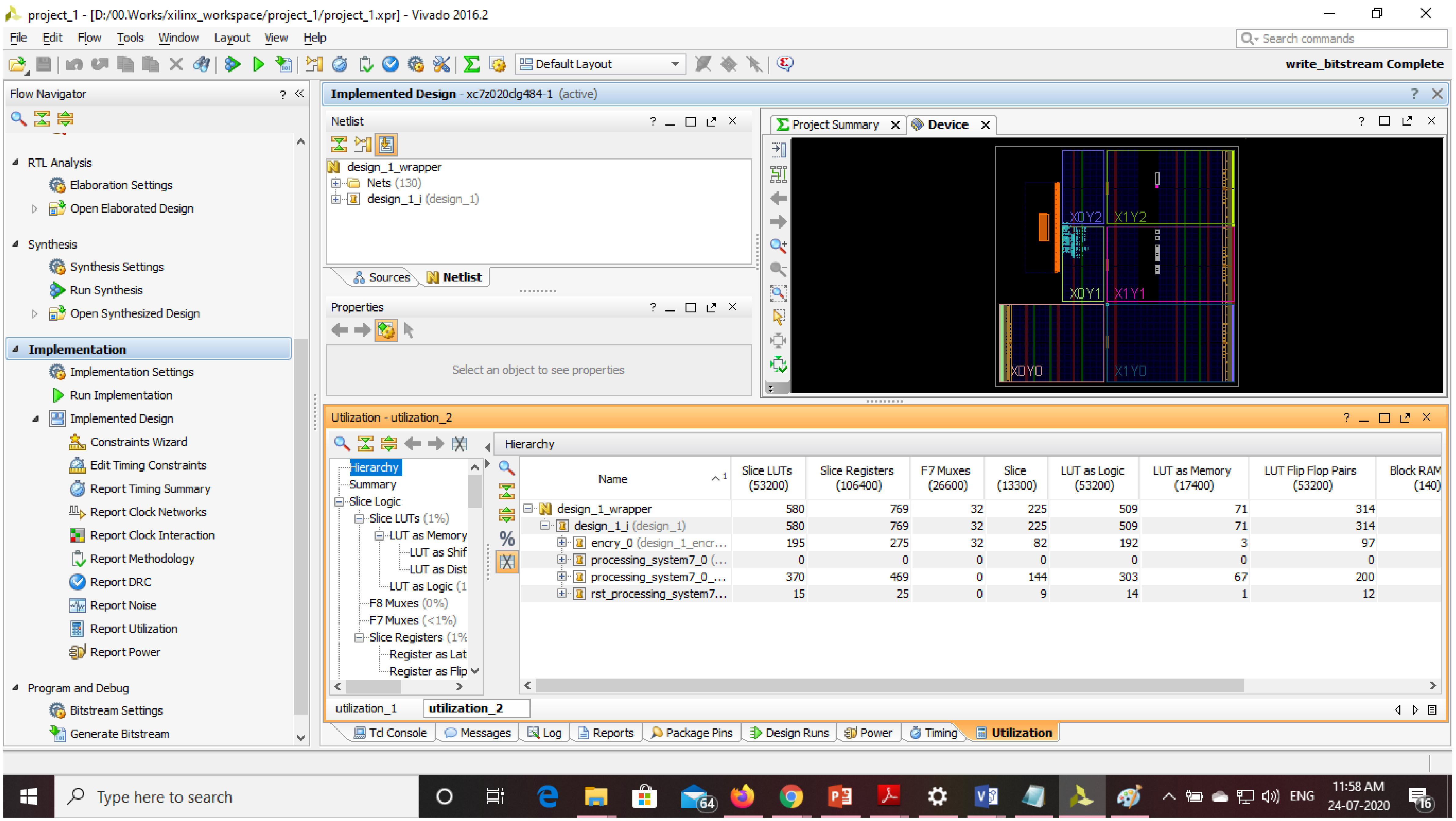
Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed
synth_1	constrs_1	synth_design Complete!							745	916	3	0	0	7/23/20 12:11 PM	00:02:1

Implementation Design



project_1 - [D:/00.Works/xilinx_workspace/project_1/project_1.xpr] - Vivado 2016.2



project_1 - [D:/00.Works/xilinx_workspace/project_1/project_1.xpr] - Vivado 2016.2

File Edit Flow Tools Window Layout View Help

Default Layout

write_bitstream Complete

Flow Navigator

RTL Analysis
- Elaboration Settings
- Open Elaborated Design

Synthesis
- Synthesis Settings
- Run Synthesis
- Open Synthesized Design

Implementation
- Implementation Settings
- Run Implementation
- Implemented Design
  - Constraints Wizard
  - Edit Timing Constraints
  - Report Timing Summary
  - Report Clock Networks
  - Report Clock Interaction
  - Report Methodology
  - Report DRC
  - Report Noise
  - Report Utilization
  - Report Power

Program and Debug
- Bitstream Settings
- Generate Bitstream

Implemented Design - xc7z020clg484-1 (active)

Netlist
- design_1_wrapper
  - Nets (130)
  - design_1_i (design_1)

Properties

Select an object to see properties

Project Summary | Device

Utilization - utilization_2

Hierarchy
- Summary
- Slice Logic
  - Slice LUTs (1%)
    - LUT as Memory
      - LUT as Shif
      - LUT as Dist
    - LUT as Logic (1
  - F8 Muxes (0%)
  - F7 Muxes (<1%)
  - Slice Registers (1%
    - Register as Lat
    - Register as Flip

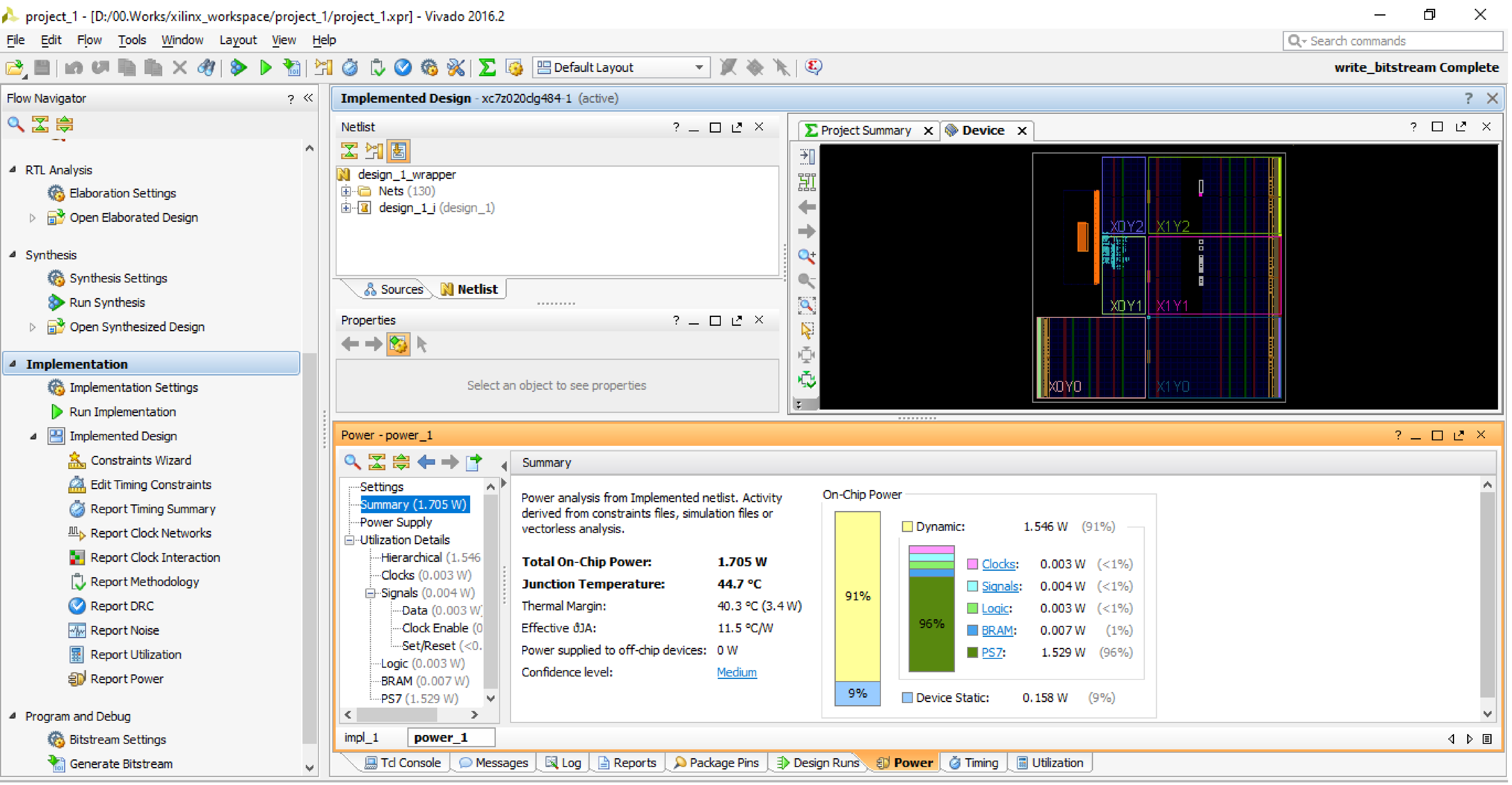
Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	LUT Flip Flop Pairs (53200)	Block RAM (140)
design_1_wrapper	580	769	32	225	509	71	314	
design_1_i (design_1)	580	769	32	225	509	71	314	
encry_0 (design_1_encr…	195	275	32	82	192	3	97	
processing_system7_0 (…	0	0	0	0	0	0	0	
processing_system7_0_…	370	469	0	144	303	67	200	
rst_processing_system7…	15	25	0	9	14	1	12	

utilization_1 | utilization_2

Tcl Console | Messages | Log | Reports | Package Pins | Design Runs | Power | Timing | Utilization

Type here to search

11:58 AM
24-07-2020



Summary

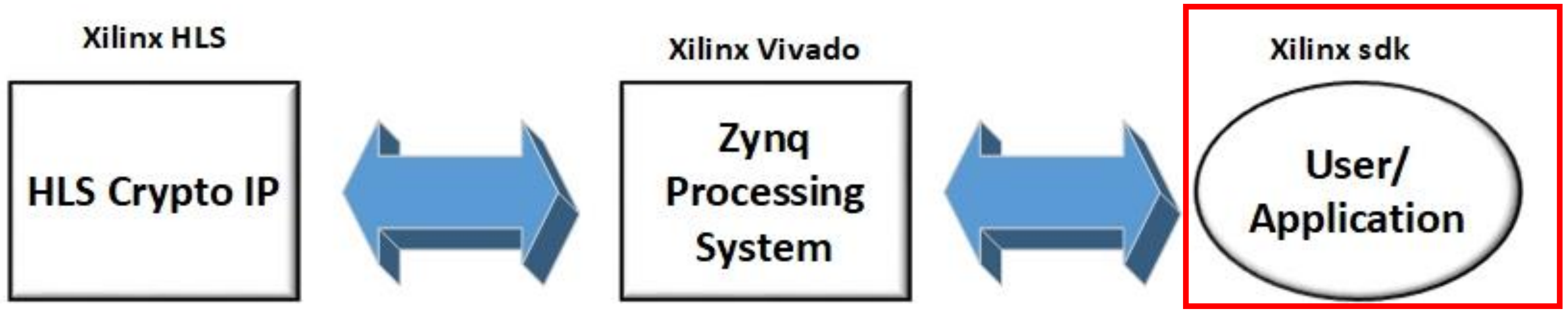
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**1.705 W**
Junction Temperature:	**44.7 °C**
Thermal Margin:	40.3 °C (3.4 W)
Effective ϑJA:	11.5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

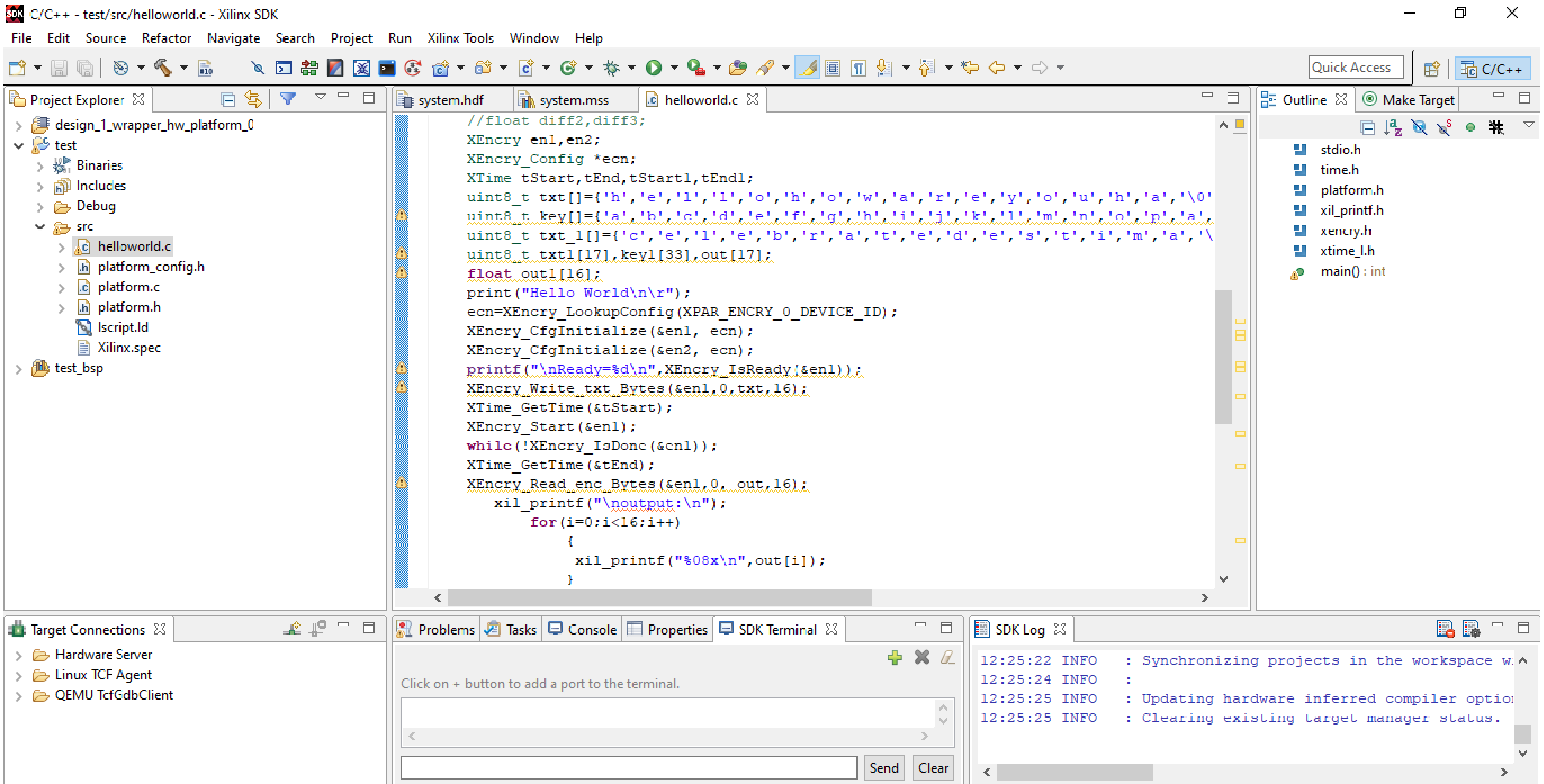
On-Chip Power

🟨	Dynamic:	1.546 W	(91%)
🟪	Clocks:	0.003 W	(<1%)
🟦	Signals:	0.004 W	(<1%)
🟩	Logic:	0.003 W	(<1%)
🟦	BRAM:	0.007 W	(1%)
🟩	PS7:	1.529 W	(96%)
🟦	Device Static:	0.158 W	(9%)

Implementation



Xilinx sdk



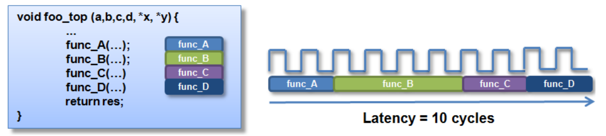
```
//float diff2,diff3;
XEncry en1,en2;
XEncry_Config *ecn;
XTime tStart,tEnd,tStart1,tEnd1;
uint8_t txt[]={'h','e','l','l','o','h','o','w','a','r','e','y','o','u','h','a','\0'
uint8_t key[]={'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','a',
uint8_t txt_1[]={'c','e','l','e','b','r','a','t','e','d','e','s','t','i','m','a','\
uint8_t txt1[17],key1[33],out[17];
float out1[16];
print("Hello World\n\r");
ecn=XEncry_LookupConfig(XPAR_ENCRY_0_DEVICE_ID);
XEncry_CfgInitialize(&en1, ecn);
XEncry_CfgInitialize(&en2, ecn);
printf("\nReady=%d\n",XEncry_IsReady(&en1));
XEncry_Write_txt_Bytes(&en1,0,txt,16);
XTime_GetTime(&tStart);
XEncry_Start(&en1);
while(!XEncry_IsDone(&en1));
XTime_GetTime(&tEnd);
XEncry_Read_enc_Bytes(&en1,0, out,16);
   xil_printf("\noutput:\n");
       for(i=0;i<16;i++)
           {
            xil_printf("%08x\n",out[i]);
           }
```

Click on + button to add a port to the terminal.

```
12:25:22 INFO    : Synchronizing projects in the workspace w
12:25:24 INFO    :
12:25:25 INFO    : Updating hardware inferred compiler optio
12:25:25 INFO    : Clearing existing target manager status.
```

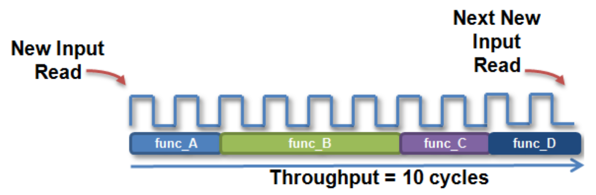
Latency and Throughput – The Performance Factors

- Design Latency
  - The latency of the design is the number of cycle it takes to output the result



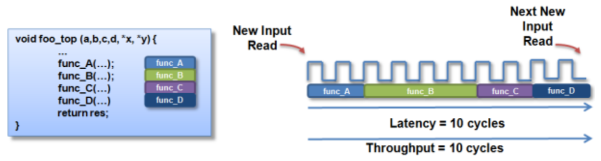
```
void foo_top (a,b,c,d, *x, *y) {
        ...
        func_A(...);
        func_B(...);
        func_C(...)
        func_D(...)
        return res;
}
```

- Design Throughput
  - The throughput of the design depends on the number of cycles between new inputs



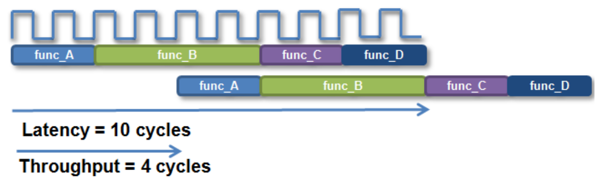
Latency and Throughput – The Performance Factors

- In the absence of any concurrency
  - Latency is the same as throughput



```
void foo_top (a,b,c,d, *x, *y) {
        ...
        func_A(...);
        func_B(...);
        func_C(...)
        func_D(...)
        return res;
}
```

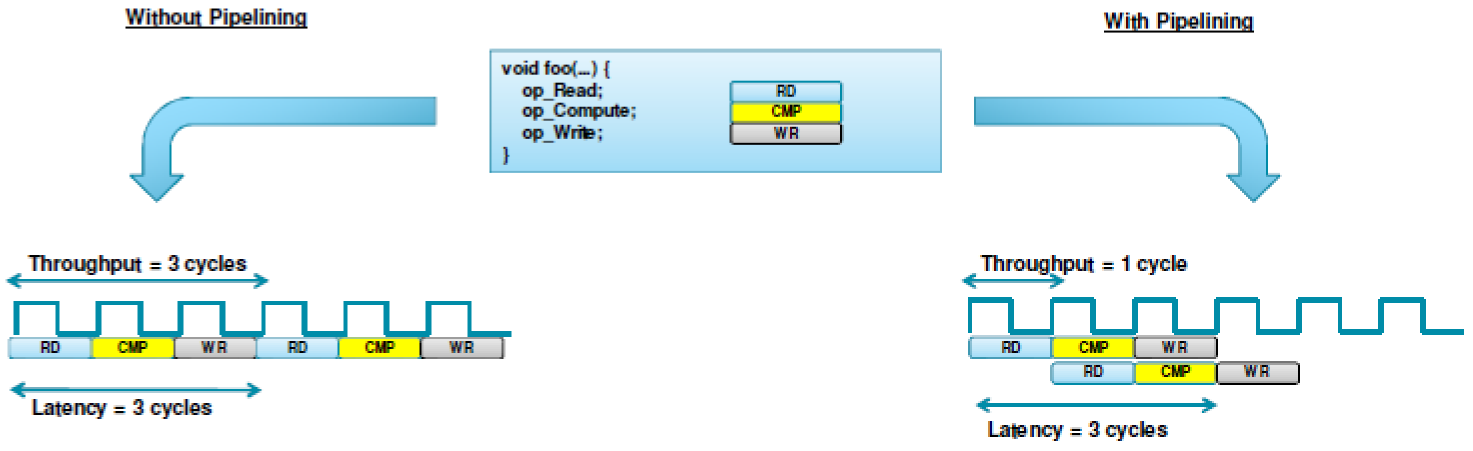
- Pipelining for higher throughput
  - Can pipeline functions and loops to improve throughput



Improving Throughput

Functions	by allowing functions to operate in parallel
Loops	schedule loops to operate in parallel
Operations	by allowing the operations to occur in parallel

Function Pipelining



Without Pipelining

There are 3 clock cycles before operation RD can occur again

- Throughput = 3 cycles

There are 3 cycles before the 1st output is written

- Latency = 3 cycles

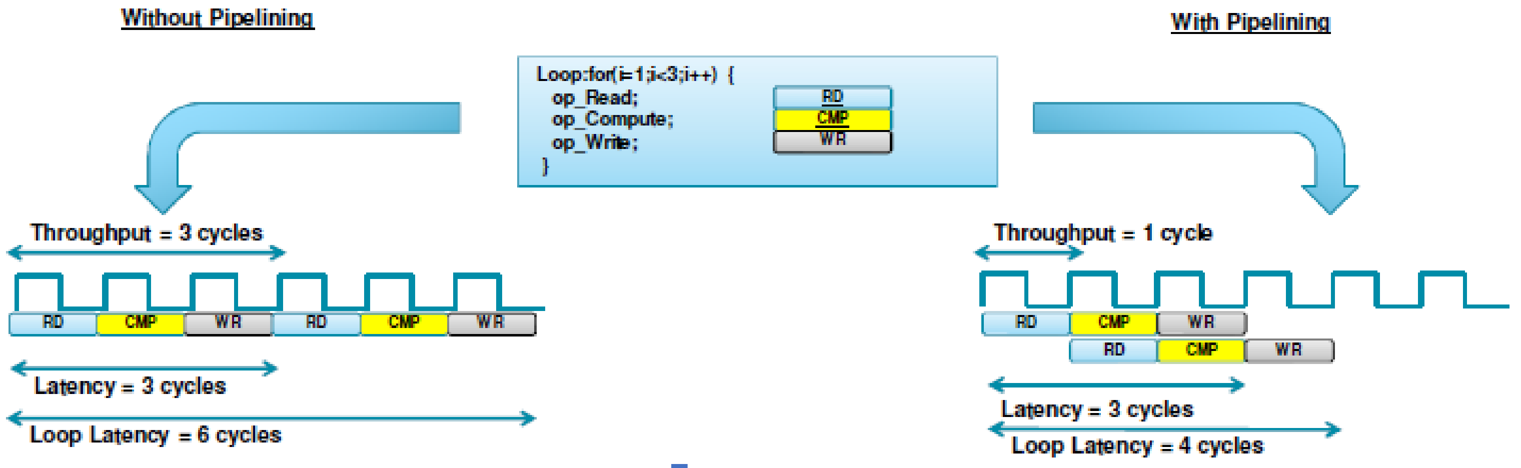
With Pipelining

The latency is the same

Throughput is better

Less cycles, higher throughput

Loop Pipelining



```
Loop:for(i=1;i<3;i++) {
  op_Read;
  op_Compute;
  op_Write;
}
```

Without Pipelining

There are 3 clock cycles before operation RD can occur again

Throughput = 3 cycles

There are 3 cycles before the 1st output is written

Latency = 3 cycles

For the loop, 6 cycles

With Pipelining

The latency is the same

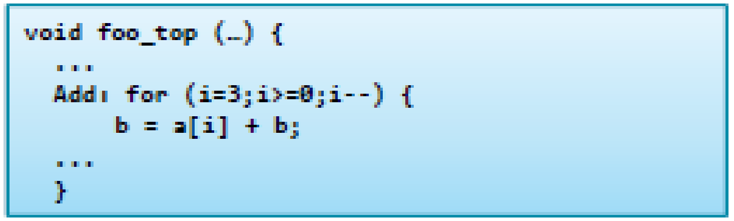
The throughput is better

Less cycles, higher throughput

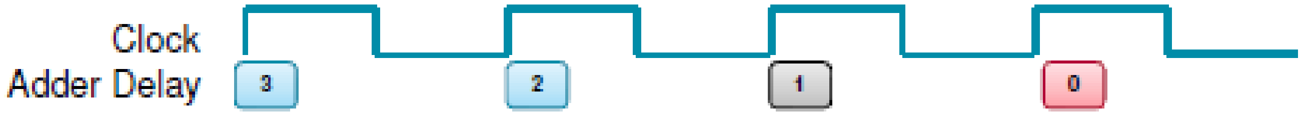
The latency for all iterations, the loop latency, has been improved

Rolled Loops Enforce Latency

- A rolled loop can only be optimized so much
  - Given this example, where the delay of the adder is small compared to the clock frequency

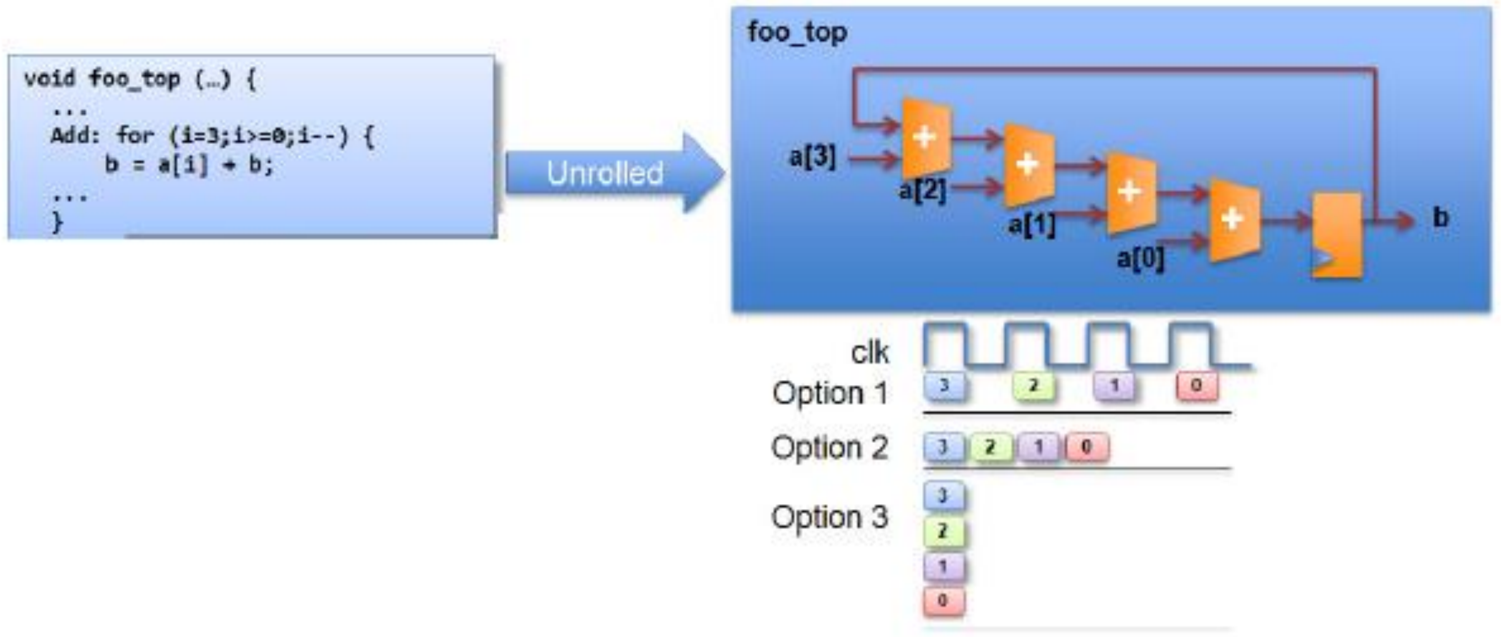


```
void foo_top (…) {
  ...
  Add: for (i=3;i>=0;i--) {
       b = a[i] + b;
  ...
  }
```



- This rolled loop will never take less than 4 cycles
  - No matter what kind of optimization is tried
  - This minimum latency is a function of the loop iteration count

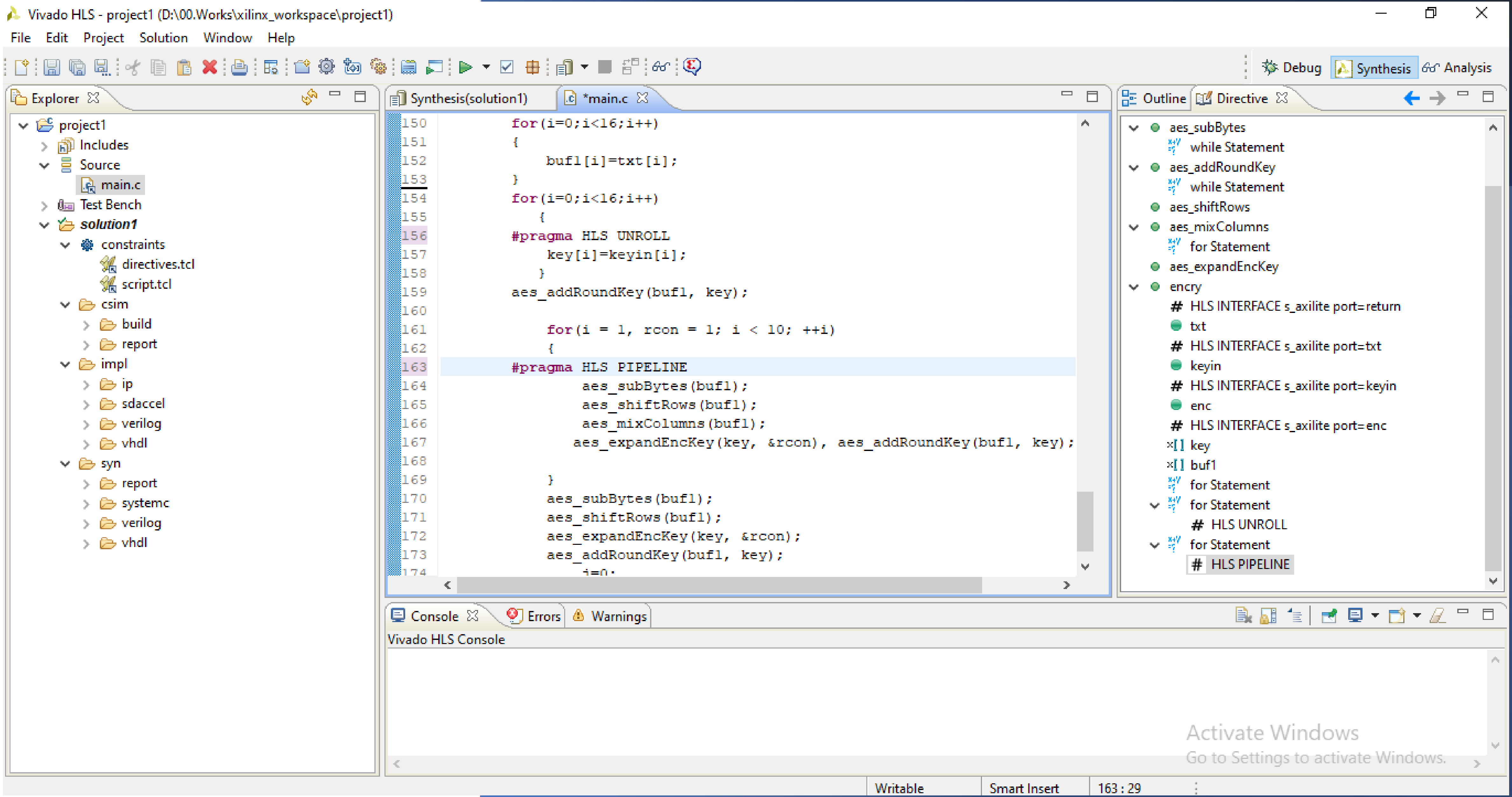
Unrolled Loops can Reduce Latency



```
void foo_top (…) {
  ...
  Add: for (i=3;i>=0;i--) {
      b = a[i] + b;
  ...
}
```

Unrolled loops allow greater option & exploration

Unrolled loops are likely to result in more hardware resources and higher area

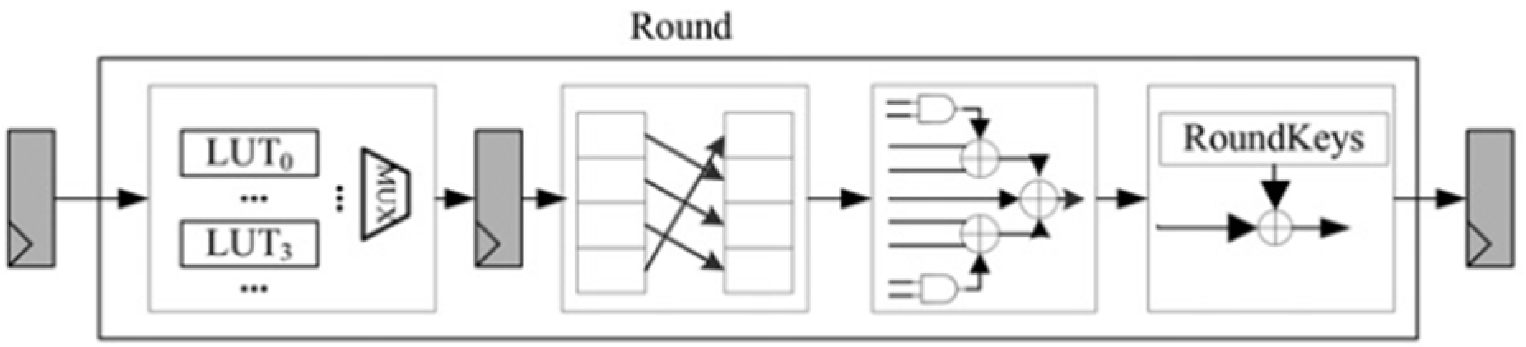

Vivado HLS - project1 (D:\00.Works\xilinx_workspace\project1)
File Edit Project Solution Window Help
Debug Synthesis Analysis
Explorer
Synthesis(solution1)
*main.c
Outline Directive
Console Errors Warnings
Vivado HLS Console
Writable Smart Insert 163 : 29

```
project1
  Includes
  Source
    main.c
  Test Bench
  solution1
    constraints
      directives.tcl
      script.tcl
    csim
      build
      report
    impl
      ip
      sdaccel
      verilog
      vhdl
    syn
      report
      systemc
      verilog
      vhdl
```

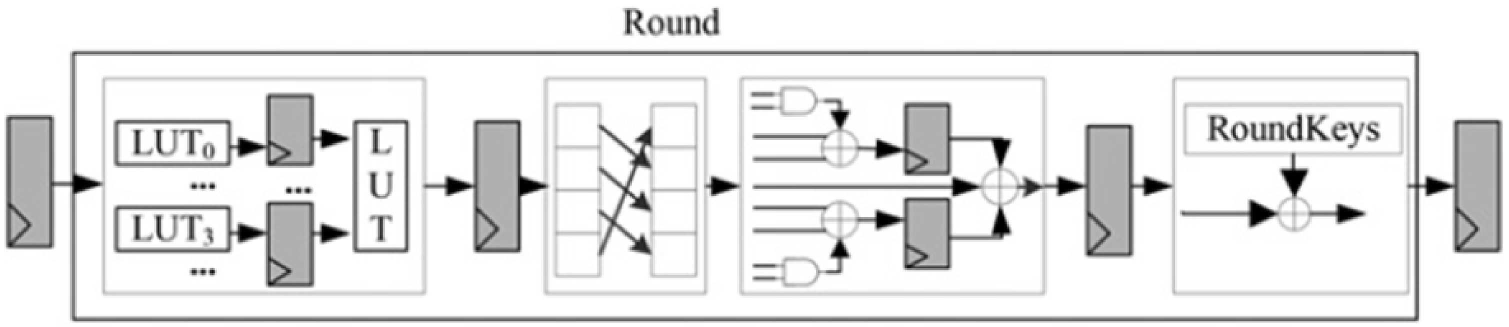
```
        for(i=0;i<16;i++)
        {
            bufl[i]=txt[i];
        }
        for(i=0;i<16;i++)
           {
        #pragma HLS UNROLL
            key[i]=keyin[i];
           }
        aes_addRoundKey(bufl, key);

            for(i = 1, rcon = 1; i < 10; ++i)
            {
        #pragma HLS PIPELINE
                    aes_subBytes(bufl);
                    aes_shiftRows(bufl);
                    aes_mixColumns(bufl);
                  aes_expandEncKey(key, &rcon), aes_addRoundKey(bufl, key);

            }
            aes_subBytes(bufl);
            aes_shiftRows(bufl);
            aes_expandEncKey(key, &rcon);
            aes_addRoundKey(bufl, key);
                i=0;
```

```
aes_subBytes
  while Statement
aes_addRoundKey
  while Statement
aes_shiftRows
aes_mixColumns
  for Statement
aes_expandEncKey
encry
  # HLS INTERFACE s_axilite port=return
  txt
  # HLS INTERFACE s_axilite port=txt
  keyin
  # HLS INTERFACE s_axilite port=keyin
  enc
  # HLS INTERFACE s_axilite port=enc
  key
  buf1
  for Statement
  for Statement
    # HLS UNROLL
  for Statement
    # HLS PIPELINE
```

Pipelined designs of AES operations on FPGAs



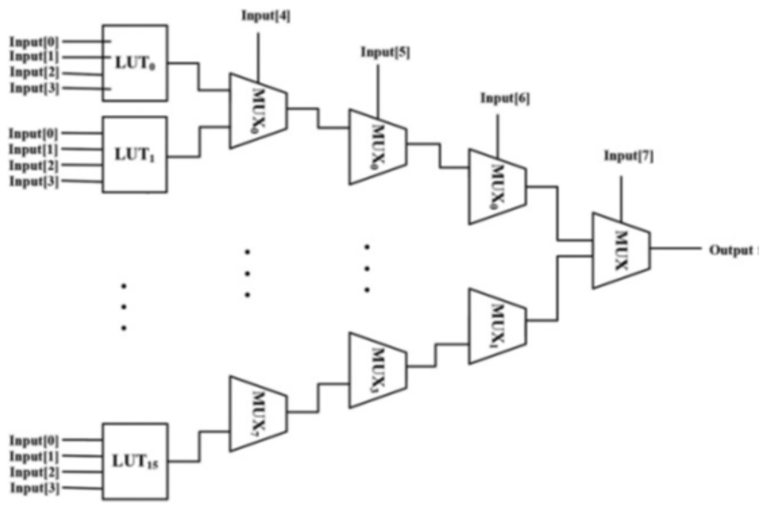
(a) Two-stage pipeline for each AES round



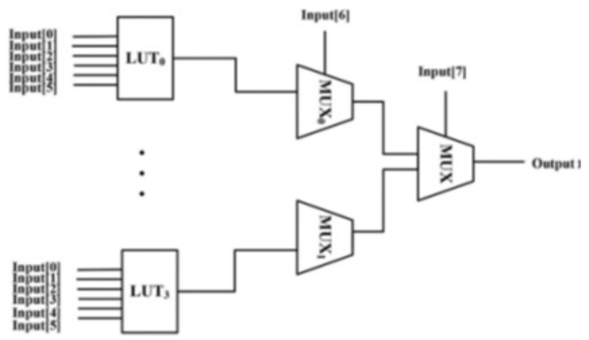
(b) Five-stage pipeline for each AES round

Implementation of s-box using LUTs

SubBytes operation implemented on FPGAs with 4-input LUTs, leading to logic depth 5

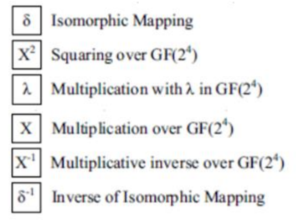


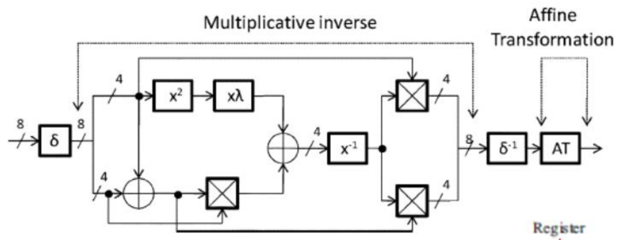
SubBytes operation implemented on FPGAs with 6 - input LUTs, leading to logic depth 3



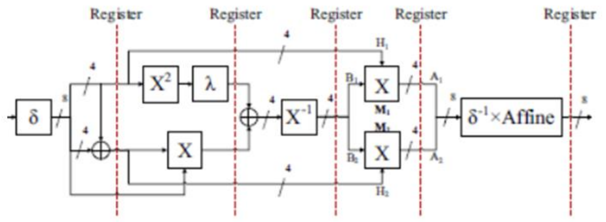
Implementation of s-box using combinational logic

Implementation of s-box in GF (2^4)

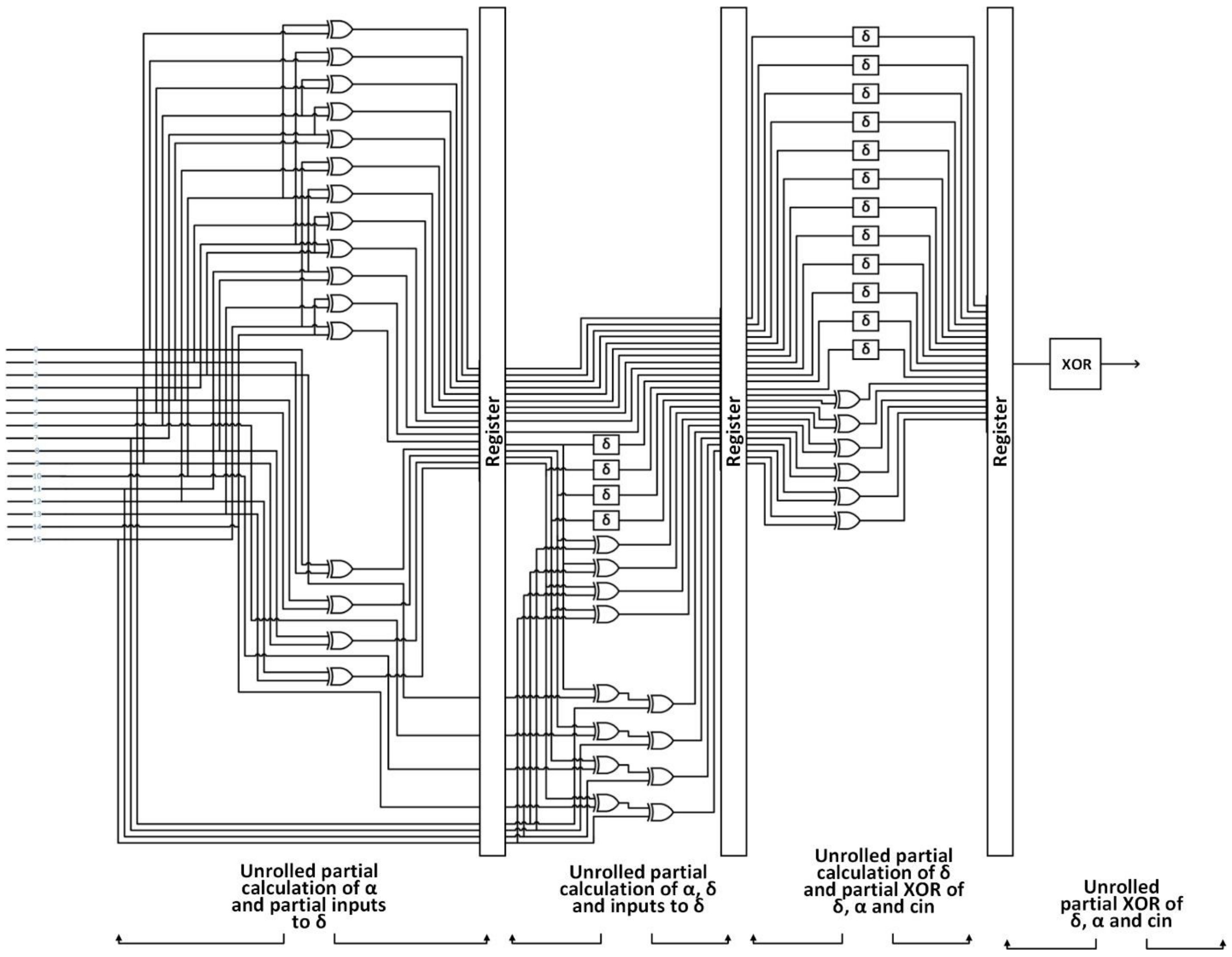

δ Isomorphic Mapping
X² Squaring over GF(2⁴)
λ Multiplication with λ in GF(2⁴)
X Multiplication over GF(2⁴)
X⁻¹ Multiplicative inverse over GF(2⁴)
δ⁻¹ Inverse of Isomorphic Mapping



Pipelined S-box

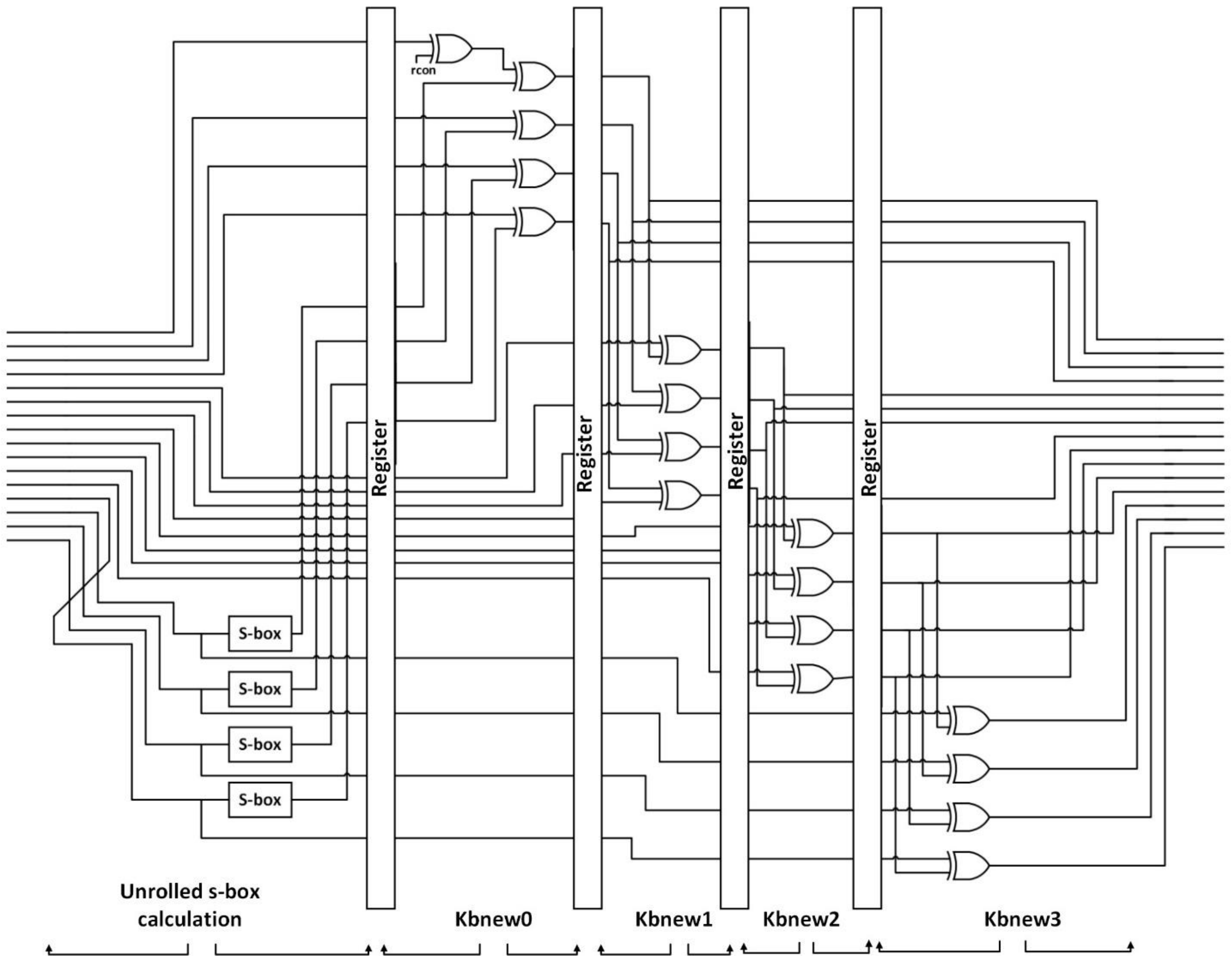


Four-stage pipelined design of Mix Column



Register
Register
Register
δ
XOR
Unrolled partial calculation of α and partial inputs to δ
Unrolled partial calculation of α, δ and inputs to δ
Unrolled partial calculation of δ and partial XOR of δ, α and cin
Unrolled partial XOR of δ, α and cin

Five-stage pipelined design of Key Expansion



Conclusion

- FPGA security is a critical concern in modern design practices.
- Protecting FPGA designs from malicious activities is essential to ensure the integrity and trustworthiness of systems.
- By adopting robust security measures, FPGA designers can mitigate risks and enhance the security posture of their designs.
- Collaboration between industry, academia, and regulatory bodies is essential to establish best practices, standards, and guidelines for FPGA security.

Queries??

Thank You