Tutorial: Implementing Unmasked AES with High Level Synthesis using Xilinx Vitis HLS

Phuc Mai and Boyang Wang

Department of ECE, University of Cincinnati
November, 2024

1 Introduction

What This Document is About? Given a software implementation of AES (Advanced Encryption Standard) written in C/C++ as the starting point, this document presents the workflow of establishing a hardware implementation of AES at the RTL (Register Transfer Level) level by using HLS (High Level Synthesis). This document specifically focuses on the case of using Xilinx Vitis HLS, which is a leading industry HLS tool. We first describe the process of generating the hardware implementation of AES with HLS given TinyAES [6], an unmasked AES software implementation written in C/C++. We simulate/verify the hardware implementation of unmasked AES at the RTL level and demonstrate the correctness and efficiency its bitstream file on a Digilent Arty A7-100T Artix-7 FPGA. In addition, we also extend the description of our pipeline to generate a hardware implementation of masked AES given an masked AES [3] software implementation written in C/C++. The detailed description of generating masked AES with HLS is presented in another document.

Why Is This Document Useful/Important? Applying HLS to a C program and verifying/running the obtained hardware implementation on a real device (e.g., a FPGA) is not trivial, especially given a complicated software implementation, such as AES. Multiple lines in the original software implementation need to be customized and modified, which requires reasonable amount of engineering time and deep understanding on both AES encryption and HLS process. The research and education community currently lacks existing documents describing the details of this comprehensive process. This document aims to fill this gap and help new students to strengthen their knowledge and skills on this topic.

Note 1. Due to the complexity of HLS process, there are different ways to modify the code to make the entire process successful. We present one way that works for us. We acknowledge that this may not be the best way.

Note 2. We generate the hardware implementations of AES at the RTL level mainly for pre-silicon side-channel analysis over simulated traces for our research projects. However,

the description of this document is general and can be used for educating/researching HLS on AES without considering side-channel analysis.

Pre-Requisite. To follow the content of this document, the readers are expected to have some basic understanding and background on AES and HLS. AES is the most popular symmetric-key encryption we use in almost every single application on every device in the real world. It is considered mathematically secure, even under attacks with quantum algorithms. Some useful references related to AES and HLS can be found at [6] and [5] respectively.

2 Background on High Level Synthesis

High Level Synthesis (HLS) is the process of synthesizing high level programming language code, e.g., C/C++ and SystemC, into Hardware Description Language (HDL), e.g., Verilog and VHDL, at the RTL level. It can ease the difficulty and complexity of writing HDL code directly and save design time. The role of HLS is similar as a compiler. Instead of compiling C code into assembly code or binaries in a compiler, HLS takes a C code as input and transforms the code into code at the RTL level through multiple intermediate code representations. In depth information about HLS can be found here [5].

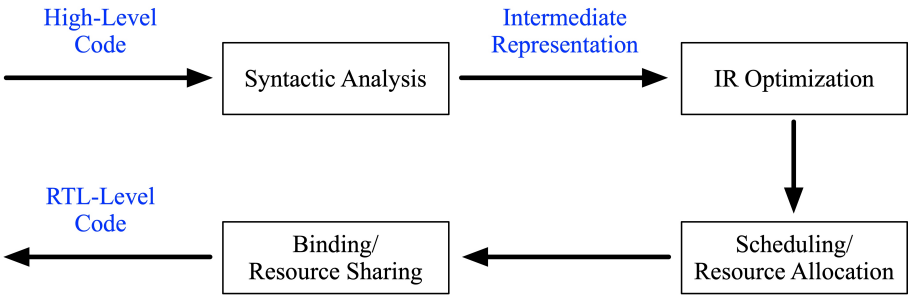


Fig. 1: High-Level Overview of High Level Synthesis (Input: High-level code; Output: RTL-Level code)

The high-level workflow of HLS is presented in Fig. 1. Specifically, given a software implementation (e.g. in C code) as input, HLS first performs lexical analysis and parsing to standardize input code into Intermediate Representation (IR). Independent from any programming language, IR can exist in form of abstract syntax tree, sequencing graph, control flow graph, or data flow graph. Next, HLS performs optimization on IR. Finally, HLS runs scheduling and binding to generate RTL-level code written in Hardware Description Language.

There are multiple commercial and open-source tools that can perform HLS. Commercial tool, such as Stratus HLS [2], Catapult [8], Vivado HLS [1], are considered to be robust and

efficient. These tools can support a wide range of high level code in terms of synthesizing. On the other hand, there is often a cost for license fee. Open-source tools, such as Bambu [4] and GAUT [7], are free to used. However, they may not be able to achieve the same efficiency and versatile level as the ones rendered by well-known commercial tools.

In this document, we investigate commercial HLS tools, specifically Xilinx Vitis HLS, in terms of designing AES hardware implementation using HLS. In addition, we use Xilinx Vivado to perform the verification of the obtained AES hardware implementation at the RTL level as well as on FPGAs.

3 Software and Hardware Settings

We use the following hardware and software in this tutorial.

- **Hardware**: A desktop with an Intel i5 CPU, 64 GB memory, and an Intel HD Graphics 630 GPU running Linux (Ubuntu 22.04); an Digilient Arty A7-100t Artix-7 FPGA board
- **Software** (all free or open-source): Xilinx Vivado 2023.2 ML Edition, Xilinx Vitis (version 2023.2), Xilinx Vitis HLS (version 2023.2), Python 3.10.12

4 Introduction to Vitis HLS

AMD/Xilinx Vitis is an Integrated Design Environment (IDE). Vitist HLS is one of the tools provided by Vitis. Vitis HLS, previously known as Vivado HLS, is highly integrated with Vivado. Vitis HLS takes C/C++ files as input and outputs Verilog or VHDL files and their IP for FPGA fabric.

4.1 Installation

The installation of Vitis is relatively straightforward. As Vitis is a commercial platform, one may need to register for an account, though no charge will be enforced during installation and day-to-day usage. Once registered, one can navigate to the following link to download the software [1]. In this document, we use Vitis version 2023.2. One may choose any supported version, however, it is suggested to choose among the latest versions for better support. When we install Vitis, Vivado and Vitis HLS will also be installed by default. The entire installation takes typically about 1∼2 hours to complete.

Default Installation Process. The default installation process is to download an AMD Unified Installer - Self Extracting Web Installer (as shown in Fig. 2) from the link above

[1] `https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis.html`

based on one's operating system (e.g., Linux), proceed to run the bin file, and follow its instructions to complete the installation. As a note, it is recommended to keep all options as default and ensure that the machine has at least 300 GB of remaining storage (for Vitis, Vitis HLS, Vivado and any other default packages).

During the installation process, the operating systems may miss some libraries needed for installing Vitis, which may not be raised until the last step of installation. If that is the case, one can install the missing libraries and then rerun the bin file until the installation of Vitis is successful.

Fig. 2: AMD Unified Installer Web Self Extracting

While the installation process is running, one can take a look at the terminal to watch out for any error along the way. If there's any file installation error as shown in 3, one can navigate to the link detailed with the error to install the required file manually. Once the file is installed, one should move it to `/tools/Xilinx/Downloads`; sudo privilege is required for this operation

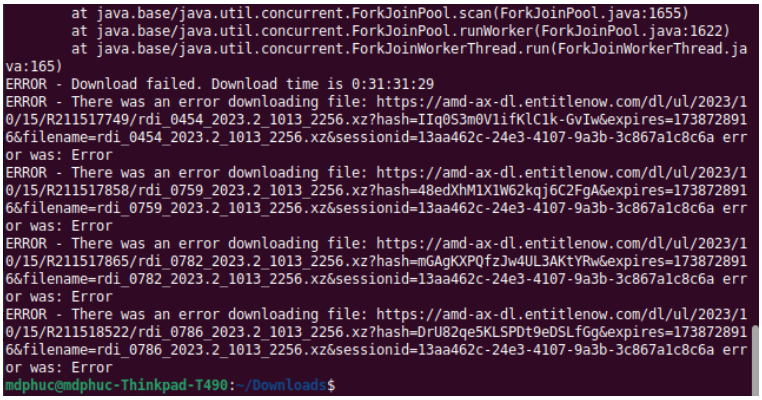


Fig. 3: AMD/Xilinx Vitis installation log

Alternative Installation Process. Adding the missing libraries and re-running the bin file could be time-consuming if the operating system misses a large number of libraries. An alternative installation process is to download the AMD Unified Installer SFD file (as shown in Fig. 4) of the same version, unzip it, and proceed to execute ***installLibs.sh*** to install any missing libraries first. Once it completes, one can download the AMD Unified Installer - Self Extracting Web Installer, proceed to run the bin file, and follow its instructions to complete the installation.

Fig. 4: AMD Unified Installer SFD

Usage. Once the installation process has been completed successfully, by default, Vitis, Vitis HLS, and Vivado file system can be found in `/tools/Xilinx`. One can follow the code snippet below to run Vitis. A successful Vitis run from the terminal can be found in 5

```
$ source /tools/Xilinx/Vitis/2023.2/settings64.sh
$ vitis # if one chooses to run Vitis
```

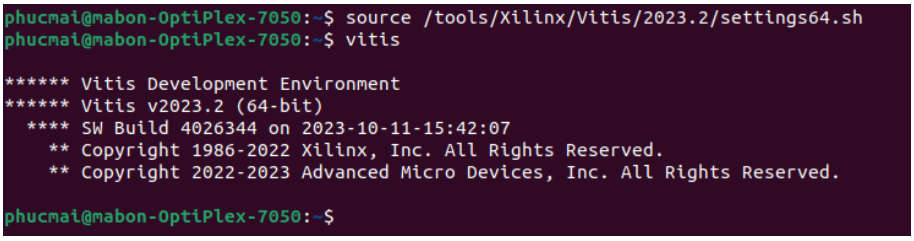


Fig. 5: Run Vitis from the terminal to launch Vitis GUI

5 HLS on TinyAES with Vitis HLS

In this section, we present the end-to-end steps of applying HLS on TinyAES [6], a software unmasked implementation of AES algorithm written in C/C++, with Vitis HLS

tool. *Unmasked* indicates the implementation does not have countermeasures against side-channel attacks. Given this software implementation as an input, the entire process outputs an associated hardware implementation of AES written in Verilog/VHDL. In addition, the hardware implementation is packed as an IP and this IP is integrated into a final hardware design using Vivado. The final hardware design is then compiled into a bitstream file, which is uploaded to a Digilent Arty A7-100T Artix-7 FPGA board for testing and verification. For the presentation of this section, we will first briefly describe the entire workflow of using Vitis HLS and Vivado in general. Next, we will introduce details of TinyAES and describe each step of the entire workflow given TinyAES.

5.1 The General Workflow of HLS with Vitis HLS

Given a software implementation in C/C++, the general workflow of HLS using Vitis HLS is shown in Fig. 6. First, after we launch Vitis, we create an HLS component by specifying a target board (similar as creating a new project by specifying a target board in Vivado). Next, we modify the C/C++ software implementation (if needed), create a main source file with a top function, write a testbench file for it in C/C++. Next, we add the top function, the source files, the testbench file into the HLS component by updating the configuration of the HLS component. Next, we run C simulations to verify the input and output of the modified software implementation to ensure its correctness. We then run HLS to generate the hardware implementation written in Verilog or VHDL. Once the hardware implementation is obtained, we run C/RTL co-simulation to ensure that the hardware implementation is correct. Next, we perform packaging to generate an IP of this hardware implementation.

This ends the typical HLS process (i.e., from a software implementation to a hardware implementation at the RTL-level). The remaining steps aim to further synthesize the design, generate the bitstream file of the final design, and run it on the target board. While these remaining steps are not parts of the typical HLS process, they are also relevant as the hardware implementation at the RTL-level should be able to generate the bitstream file and operate/verify correctly on the target board. It is worth mentioning that successfully obtaining the hardware design at the RTL-level with HLS does not necessarily suggest its compatibility with these remaining steps (e.g., some warnings may be ignored at the RTL-level but can prevent successfully generating the bitstream files for the real target). Therefore, we also present these remaining steps in the workflow to make the entire workflow self-contained.

Given the generated IP, we can load it in Vivado and create a final hardware design by providing function inputs to the IP and accessing function outputs from the IP. Next, we

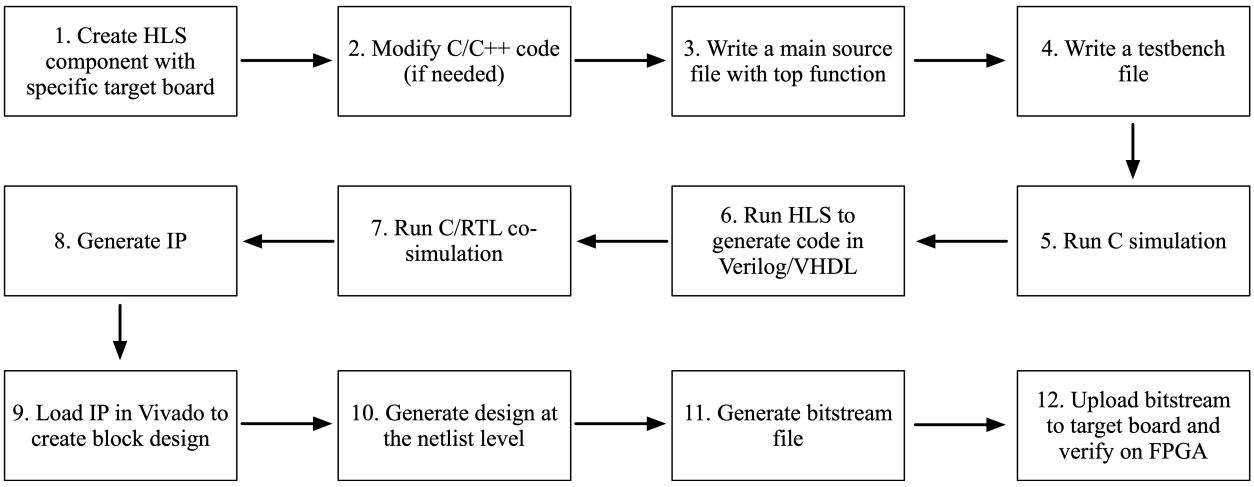


Fig. 6: The workflow of HLS for the final FPGA design using Vitis HLS and Vivado

transform the hardware design at the RTL-level into the netlist level and fix any warnings/errors. Then, we run the implementation in Vivado to create the bitstream file based on the design at the netlist level and fix any I/O port issues given the constraint of the target board. Finally, we program the FPGA target board with the bitstream file using Vivado hardware manager and verify the final design on the FPGA. If the design runs correctly on the FPGA, it completes the entire workflow. Otherwise, we fix the remaining issues/warnings until the design on the FGPA is correct.

Throughout this entire process, we find that two major aspects/steps are particularly challenging and significant for the case of HLS over TinyAES: (1) modifying the original software implementation to make it compatible with Vitis HLS and the downstream bitstream generation; and (2) running simulations/verification correctly at multiple levels (including C, RTL, and FPGA). Sometimes, it requires revisiting the software/hardware code through the process. We will describe the details of these steps later in this section.

5.2 Step 1: Create a New HLS Component

After we launch Vitis, we create a new HLS component by choosing "Create Component" under "HLS Development" on the welcome screen. We will need to provide a name of this HLS component and its location. After that, we need to create a configuration file (we select default option: Empty File and provide a name of the new empty configuration file). Next, we need to specify the top function name and source files. We leave them as empty (default) for now and choose to add them later. Next, we need to specify the target board part, which we choose the part as `xc7a100tcsg324-1` for Digilient Arty A7 Artix-7 FPGA board in our

example. Please note that the part for Digilient Arty A7 Artix-7 FPGA board may not show up in Vitis's default directory. To address this, one can add the target board part to Vitis's directory by following the guide in [2]. Once the target board's files are added, one can then continue with the process. Next, we need to specify the initial settings (e.g., clock, flow_target, etc.) for this HLS component. We leave them as default. This completes the step of creating the new HLS component. Figures of the above steps are presented from Fig. 7 to Fig. 12.

After we create this new HLS component, we will need to specify the top function, add source files, and create testbench files to this HLS component. Since we need to modify the source code of TinyAES and create testbench files in advance, we will discuss the details of our modifications and the testbench files first. Then, we will describe how to include these source files and testbench files in our HLS component.

5.3 Introduction to TinyAES

TinyAES is a small portable implementation of unmasked AES written in C, supporting Electronic Code Block (ECB), Counter mode (CTR), and Cipher Block Chaining (CBC) modes across all versions of AES: 128 bit, 192 bit, 256 bit. The source code for TinyAES can be found here [6]. In this document, we focus on the following three files only from the TinyAES repo.

- *aes.h*: Header file defines all AES encryption and decryption functions
- *aes.c*: Main file details all AES step functions and components
- *test.c*: Test file contains test scripts for verification

In this document, we focus on ***the source code associated with the encryption function of AES-128 using the ECB model only***, which is sufficient for us to analyze side-channel leakage of the hardware design. On the other hand, our discussion and methodology can be further extended to other modes of TinyAES.

After reviewing the original source code, the function `AES_ECB_encrypt()`, which performs AES-128 encryption with ECB mode, can be found in *aes.c* at line 470. We highlight the original code below as a reference for comparisons with later modified versions.

```
1 // Original version in TinyAES, at line 470 in aes.c
2 void AES_ECB_encrypt(const struct AES_ctx* ctx, uint8_t* buf){
3     // The next function call encrypts the PlainText with the Key using AES
          algorithm.
```

[2] https://digilent.com/reference/programmable-logic/guides/install-board-files

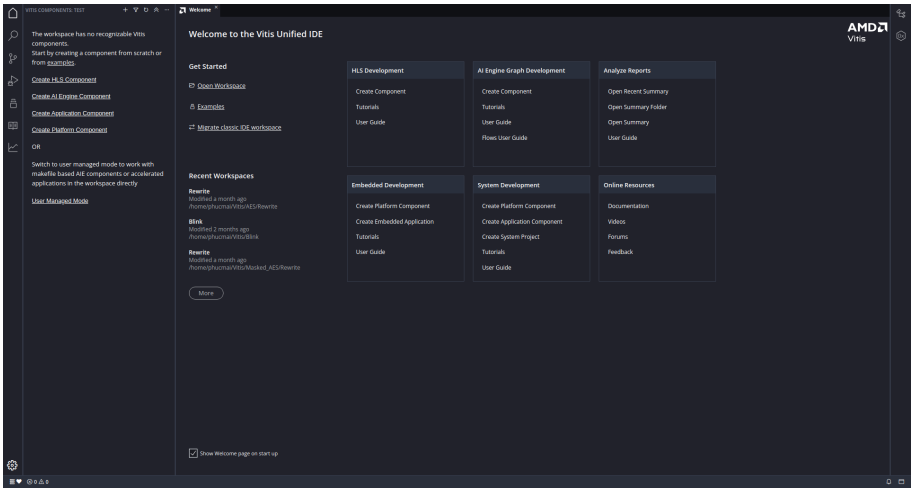


Fig. 7: Welcome Screen of Vitis

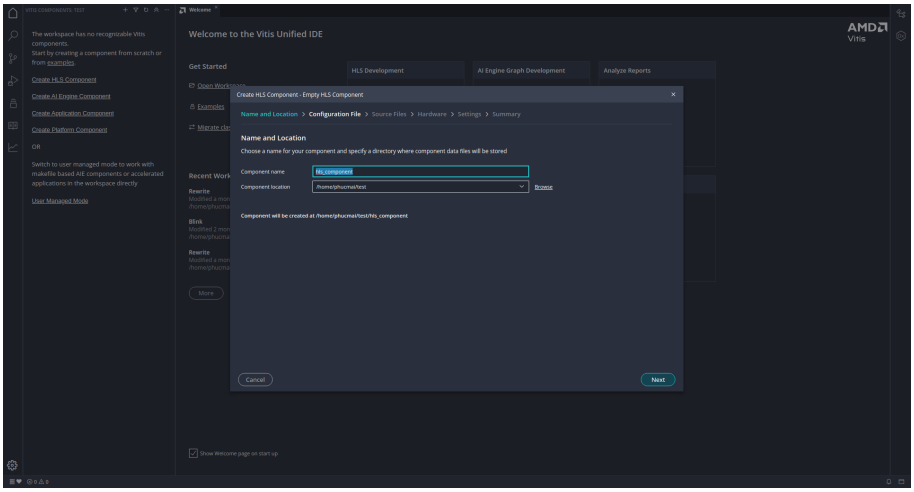


Fig. 8: Step 1: Provide a project name and location

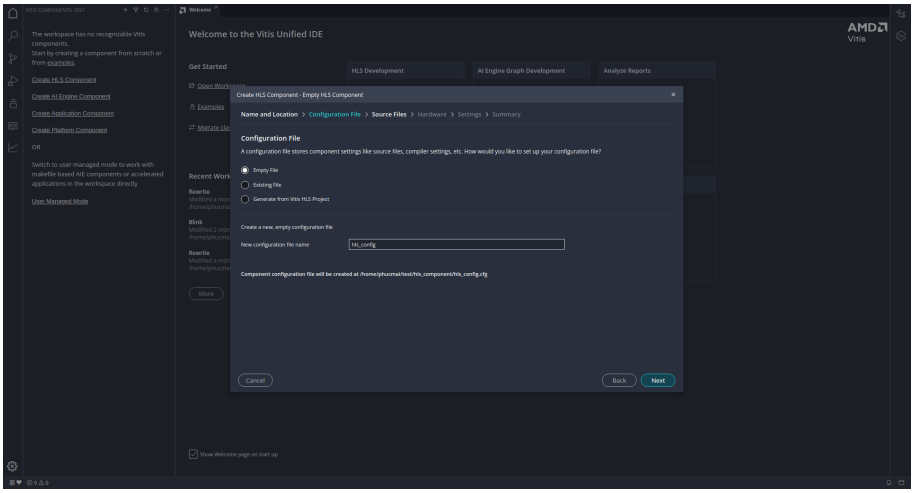


Fig. 9: Step 2: Create a new empty configuration file, named `hls_config`

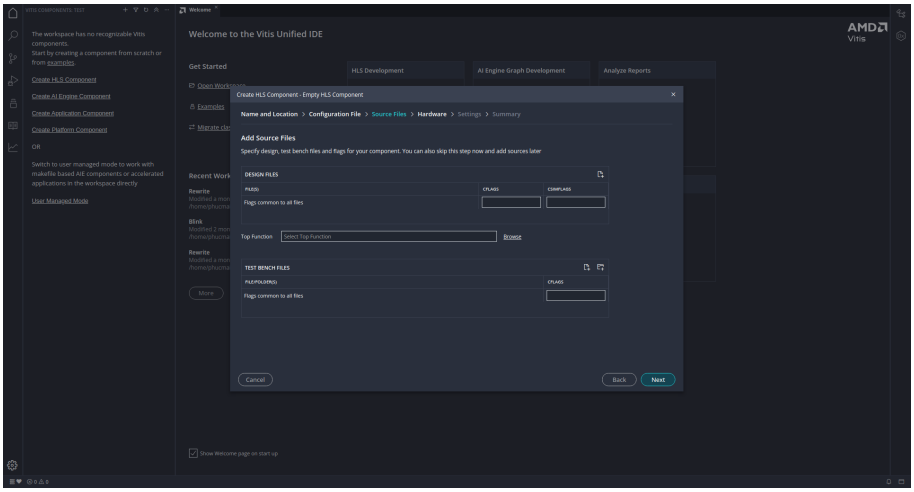


Fig. 10: Step 3: Specify top function and source files (we leave them as empty for now)

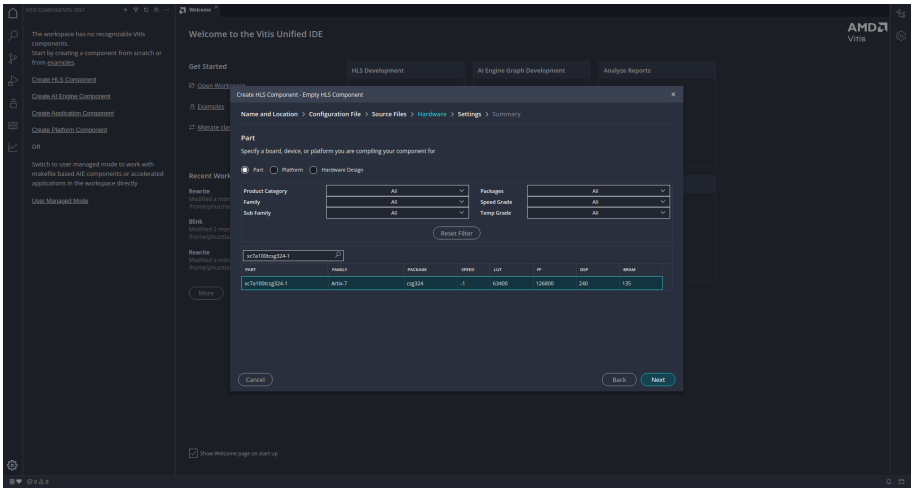


Fig. 11: Step 4: Specify target board part (choose/add `xc7a100tcsg324-1`)

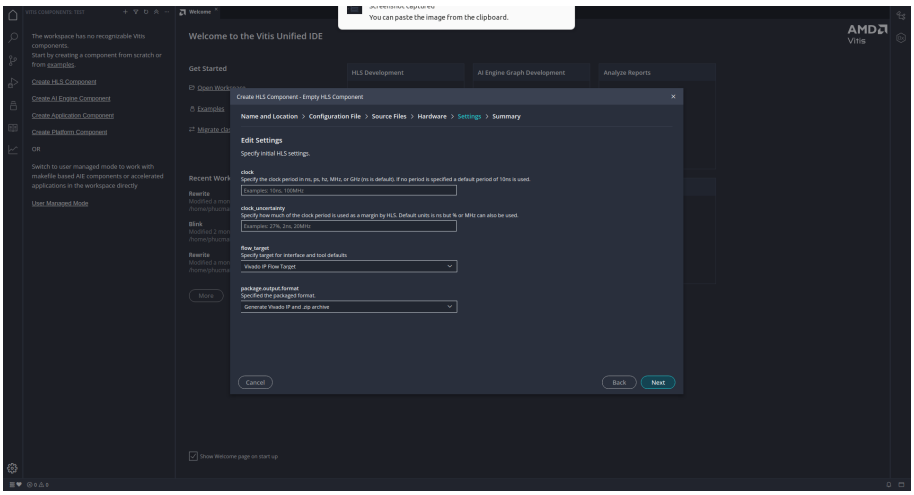


Fig. 12: Step 5: Specify settings (choose default settings)

```
4       Cipher((state_t*)buf, ctx->RoundKey);
5   }
```

The `Cipher()` function performs the 10 rounds of AES operations, including AddKeys, SubBytes, ShiftRows, and MixColumns, by following the standard AES algorithm. It can be found at line 413 in *aes.c*. It takes a `state_t*` (a 4 by 4 matrix) plaintext input converted from 16 `uint8_t` array provided by `AES_ECB_encrypt`. The another argument is `RoundKey`, which is the output of function `KeyExpansion()` (at line 146 in *aes.c*). We highlight the original code below as a reference for comparisons with later modified versions.

```
1   // Original version in TinyAES, at line 413 in aes.c
2   static void Cipher(state_t* state, const uint8_t* RoundKey){
3       uint8_t round = 0;
4
5       // Add the First round key to the state before starting the rounds.
6       AddRoundKey(0, state, RoundKey);
7
8       // There will be Nr rounds.
9       // The first Nr-1 rounds are identical.
10      // These Nr rounds are executed in the loop below.
11      // Last one without MixColumns()
12      for (round = 1; ; ++round) {
13        SubBytes(state);
14        ShiftRows(state);
15
16        if (round == Nr) {
17          break;
18        }
19        MixColumns(state);
20        AddRoundKey(round, state, RoundKey);
21      }
22
23      // Add round key to last round
24      AddRoundKey(Nr, state, RoundKey);
25  }
```

5.4 Step 2: Modify TinyAES C/C++ Code

Unfortunately, applying HLS directly on the original TinyAES does not generate the hardware implementation at the RTL level. There are many lines in the original C code of TinyAES that are not compatible with the process using Vitis HLS. Therefore, we need to modify the TinyAES C/C++ code without affecting the correctness of the AES encryption. These modifications are also specific for the HLS process with Vitis HLS. Put differently, if a different HLS tool is applied, the modifications may not be identical.

In the following, we highlight several main principles while we modify the source code for the compatibility with Vitis HLS.

1. Vitis HLS does not support pointers well (e.g., having a pointer and performing multiple operations on this pointer) and does not even support pointers to pointers at all. For instance, between an array (e.g. `uint8_t a[16]`) and a pointer to array (e.g., `uint8_t *a_ptr`), using the array in the C code is more compatible with the HLS process with Vitis HLS compared to using the pointer. We decide to avoid the use of the pointers as much as we can.
2. Although reading from an array and writing to the same array at the same time is permitted by the C/RTL Co-simulation, those arrays are translated into Single Port RAM by default when it comes to the actual hardware design on an FPGA. As a result, one can either read or write at one time, but not both (e.g., accessing two values from an array, calculating the sum of the two, and updating a value in the same array). While this can be potentially addressed by configuring an array as Dual Port RAM, we decide to keep the default setting with Single Port RAM and avoid reading from and writing to an array at the same time in the C code.
3. Although `struct` can be supported by Vitis HLS, we decide to avoid using `struct` for simplicity in this document.
4. Conversion between non-native C datatype is prohibited. For example, if one use `typedef uint8_t state_t[4][4]` and they have a variable `uint8_t a[16]`, if they later use `state_t* b = (state_t*)a`, `(state_t*)a` is a conversion between non-native C datatype as it tries convert `uint8_t` into `state_t*`

With the above four principles (referred to as the HLS principles) in mind, we make modifications in the following functions, `AES_ECB_encrypt()`, `Cipher()`, `AddRoundKey()`, `SubBytes()`, `ShiftRows()`, `MixColumns()`, `KeyExpansion()`, and `AES_init_ctx()`, in *aes.c*.

Modifications in `AES_ECB_encrypt()`. We first modify function `AES_ECB_encrypt()`. The first argument `AES_ctx*` in the original code is a `struct` accessed by a pointer, which allows one to access both IV (Initialization Vectors) and round keys. We update it as an array, `unit8_t RoundKey[AES_keyExpSize]`, for holding round keys only to avoid using pointers and `struct`. IV is not used in the ECB model anyway. `AES_keyExpSize` is a global variable, which was originally defined as 176 for AES-128 in *aes.h*. It indicates 176 bytes, which covers all the bytes for the 11 subkeys after performing key expansion. For the second argument `buf`, it is a pointer, we update it as an array `unit8_t buf[AES_BLOCKLEN]` instead to hold the plaintext. `AES_BLOCKLEN` is a global variable, which was originally defined as 16 for AES-128 in *aes.h*. It indicates 16 bytes in a block of a plaintext. In addition, we add an

additional argument, `unit8_t enc[AES_BLOCKLEN]`, to hold the ciphertext (i.e., output of the encryption) without writing it back to `buf[AES_BLOCKLEN]`. This addresses the potential issues of reading from and writing to the same array.

These three modifications update the declaration of the function. The single line of function call on `Cipher()` is also updated accordingly. We explain the details of the modifications in `Cipher()` next.

```
// Original version in TinyAES, at line 470 in aes.c
void AES_ECB_encrypt(const struct AES_ctx* ctx, uint8_t* buf){
     // Encrypt the PlainText with the Key using AES algorithm.
     Cipher((state_t*)buf, ctx->RoundKey);
}
```

```
// Modified version due to HLS
void AES_ECB_encrypt(const uint8_t RoundKey[AES_keyExpSize], const uint8_t buf[
    AES_BLOCKLEN], unit8_t enc[AES_BLOCKLEN]){
     // Encrypt the PlainText with the Key using AES algorithm.
     Cipher(buf, RoundKey, enc);
}
```

Modifications in `Cipher()`. Specifically, the first argument `state_t* state` is the state, which is typically a 4 by 4 array, representing all the 16 bytes initialized by the plaintext. The encryption keeps updating this state based on all the operations to derive the ciphertext after 10 rounds. Instead of using pointer to pointer to represent this state, we change it to a 1 dimensional array, `unit8_t buf_state[AES_BLOCKLEN]`. This does not affect the correctness of the encryption as show in Fig. 13. The second argument `unit8_t* RoundKey` is a pointer, and we modify it as an array `unit8_t RoundKey[AES_keyExpSize]`. In addition, the original code keeps updating the state by reading from and writing to it, which causes issues for the final hardware design. We add another array as an additional argument `unit8_t enc[AES_BLOCKLEN]` to hold the output without writing it back to the first array `unit8_t buf_state[AES_BLOCKLEN]`.

The original version and modified version of the function declaration are presented below.

```
// Original version in TinyAES, at line 413 in aes.c
static void Cipher(state_t* state, const uint8_t* RoundKey){
    ......
}
```

```
// Modified version due to HLS
static void Cipher(const uint8_t buf_state[AES_BLOCKLEN], const uint8_t RoundKey[
    AES_keyExpSize], uint8_t enc[AES_BLOCKLEN]){
    ......
```

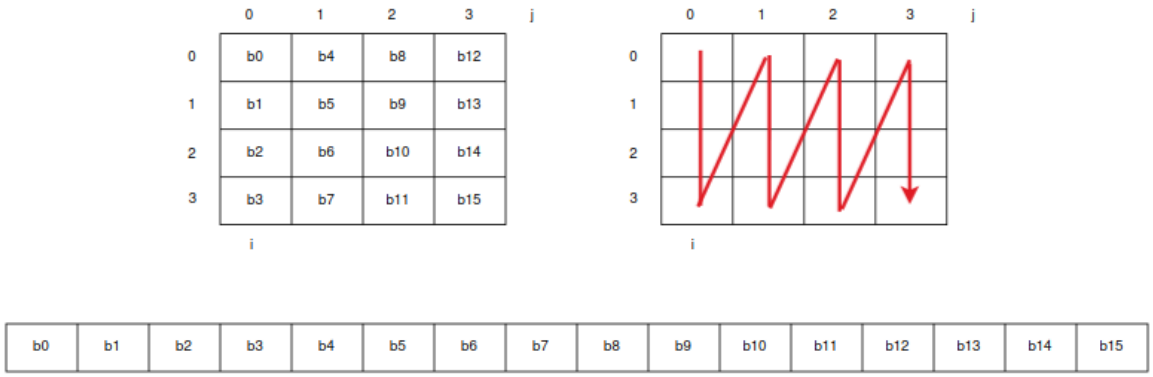


Fig. 13: A state representing as a 4×4 array of 16 bytes or a 1-dimensional array of 16 bytes.

```
}
```

For the code inside function `Cipher()`, we make the modifications accordingly by using arrays instead of points and introducing an additional array for each step, including AddRoundKey, SubBytes, ShiftRows, and MixColumns, to hold the output without writing back to the input array. The purpose of this is to transform single original array into a separate input array and a separate output array to avoid reading from and writing to the same array at the same time within the function. The original version and modified version can be found below.

```
// Original version in TinyAES, at line 413 in aes.c
static void Cipher(state_t* state, const uint8_t* RoundKey){
    uint8_t round = 0;

    // Add the First round key to the state before starting the rounds.
    AddRoundKey(0, state, RoundKey);

    // There will be Nr rounds.
    // The first Nr-1 rounds are identical.
    // These Nr rounds are executed in the loop below.
    // Last one without MixColumns()
    for (round = 1; ; ++round) {
      SubBytes(state);
      ShiftRows(state);

      if (round == Nr) {
        break;
      }
      MixColumns(state);
      AddRoundKey(round, state, RoundKey);
    }
    // Add round key to last round
```

```
    AddRoundKey(Nr, state, RoundKey);
}
```

```
// Modified version due to HLS
static void Cipher(const uint8_t buf_state[AES_BLOCKLEN], const uint8_t RoundKey[
    AES_keyExpSize], uint8_t enc[AES_BLOCKLEN]){

    // An additional array to hold output without writing back to the input array
        in the initial AddRoundKey
    uint8_t temp_new_buf_state1[AES_BLOCKLEN];

    // Add the First round key to the state before starting the rounds.
    AddRoundKey(0, temp_new_buf_state1, buf_state, RoundKey);

    // An additional array to hold output from each step without writing back to
        the input array in AddRoundKey, SubBytes, ShiftRows, or MixColumns
    uint8_t temp_new_buf_state_round_1[AES_BLOCKLEN];

    // The first 9 rounds
    for (int i = 1; i <= 9; i++){

        // A Round() function is introduced by us to simplify the code due to HLS
        Round(i, temp_new_buf_state_round_1, temp_new_buf_state1, RoundKey);

        // Copy array temp_new_buf_state_round_1 to  array temp_new_buf_state1,
            this avoids creating a new array to hold the output for every step
        for (int i = 0; i < AES_BLOCKLEN; i++){
            temp_new_buf_state1[i] = temp_new_buf_state_round_1[i];
        }
    }

    // The last round, i.e., 10-th round
    // An additional array to hold output from SubBytes in the 10-th round
    uint8_t temp_new_buf_state_round_10_1[AES_BLOCKLEN];
    SubBytes(temp_new_buf_state_round_10_1, temp_new_buf_state_round_1);

    // An additional array to hold output from ShiftRows in the 10-th round
    uint8_t temp_new_buf_state_round_10_2[AES_BLOCKLEN];
    ShiftRows(temp_new_buf_state_round_10_2, temp_new_buf_state_round_10_1);

    // An additional array to hold output from AddRoundKey in the 10-th round. It
        is also the final output, i.e., ciphertext
    uint8_t temp_new_buf_state_final[AES_BLOCKLEN];
    AddRoundKey(Nr, temp_new_buf_state_final, temp_new_buf_state_round_10_2,
        RoundKey);

    // Copy the ciphertext to array enc
```

```
    for (int i = 0; i < AES_BLOCKLEN; i++){
            enc[i] = temp_new_buf_state_final[i];
    }
}
```

As shown above, we introduce a function `Round()` to simplify the code of the first 9 rounds of AES inside the modified `Cipher()`. Each call of function `Round()` performs one round of AES based on the round key decided by the round number. The code of `Round()` is presented below.

```
// A new function in the modified version due to HLS
void Round(int round, uint8_t new_state[AES_BLOCKLEN], const uint8_t old_state[
    AES_BLOCKLEN], const uint8_t RoundKey[AES_keyExpSize]{

    // An additional array to hold output from SubBytes in round-th round.
    uint8_t temp_new_buf_state_round_1[AES_BLOCKLEN];
    SubBytes(temp_new_buf_state_round_1, old_state);

    // An additional array to hold output from ShiftRows in round-th round.
    uint8_t temp_new_buf_state_round_2[AES_BLOCKLEN];
    ShiftRows(temp_new_buf_state_round_2, temp_new_buf_state_round_1);

    // An additional array to hold output from MixColumns in round-th round.
    uint8_t temp_new_buf_state_round_3[AES_BLOCKLEN;
    MixColumns(temp_new_buf_state_round_3, temp_new_buf_state_round_2);

    AddRoundKey(round, new_state, temp_new_buf_state_round_3, RoundKey);
}
```

Due to the modifications in `Cipher()`, we also need to modify `AddRoundKey()`, `SubBytes()`, `ShiftRows()`, and `MixColumns()` accordingly. We present some selected details next.

Modifications in `AddRoundKey()`. We apply the same principles to modify `AddRoundKey()`. Specifically, we replace pointers with arrays, add an additional array to hold output to avoid reading from and writing to the same array at the same time within the function, and avoid conversions between non-native C datatype.

```
// Original version in TinyAES, at line 237 in aes.c
static void AddRoundKey(uint8_t round, state_t* state, const uint8_t* RoundKey){

    uint8_t i,j;
    for (i = 0; i < 4; ++i){
        for (j = 0; j < 4; ++j){
            (*state)[i][j] ^= RoundKey[(round * Nb * 4) + (i * Nb) + j];
        }
    }
}
```

```
// Modified version due to HLS
static void AddRoundKey(uint8_t round, uint8_t new_state[AES_BLOCKLEN], const
    uint8_t old_state[AES_BLOCKLEN], const uint8_t RoundKey[AES_keyExpSize]){

    uint8_t i,j;
    for (i = 0; i < 4; ++i){
        for (j = 0; j < 4; ++j){
            new_state[4*i + j] = old_state[4*i + j] ^ RoundKey[(round * Nb * 4) +
                (i * Nb) + j];
        }
    }
}
```

The content of `SubBytes()`, `ShiftRows()`, and `MixColumns()` can also be updated similarly with these principles. We only present the original and modified function definitions below. We skip the detailed modified code for each of them in this document. These details can be found in our repository [?].

```
// Original version in TinyAES at line 251 in aes.c
static void SubBytes(state_t* state) {
    ......
}

// Original version in TinyAES at line 266 in aes.c
static void ShiftRows(state_t* state) {
    ......
}

// Original version in TinyAES at line 300 in aes.c
static void MixColumns(state_t* state) {
    ......
}
```

```
// Modified version due to HLS
static void SubBytes(uint8_t new_state[AES_BLOCKLEN], const uint8_t old_state[
    AES_BLOCKLEN]) {
    ......
}

// Modified version due to HLS
static void ShiftRows(uint8_t new_state[AES_BLOCKLEN], const uint8_t old_state[
    AES_BLOCKLEN]) {
    ......
}

// Modified version due to HLS
```

```
static void MixColumns(uint8_t new_state[AES_BLOCKLEN], const uint8_t old_state[
    AES_BLOCKLEN]) {
    ......
}
```

Modifications in `KeyExpansion()`. After we make modifications to the above functions, we will need to make some minor changes to function `KeyExpansion()` and `AES_init_ctx()` to make the code consistent by using the same principles. We highlight the modifications on `KeyExpansion()` first. Specifically, we first update the arguments from pointers to arrays. Second, we use four separate variables, including, `tempa_0`, `tempa_1`, `tempa_2`, `tempa_3` rather than an array `tempa[4]` used in the original version. In other words, we replace `tempa[i]` with `tempa_i` in the code. This is because there are some lines associated with `tempa[i]` can lead to reading from and writing to the same array at the same time within the function. The original version and modified version are presented below. Only the lines that are associated with the modifications are presented.

```
// Original version in TinyAES, line 145 in aes.c
// This function produces Nb(Nr+1) round keys. The round keys are used in each
    round to decrypt the states.
static void KeyExpansion(uint8_t* RoundKey, const uint8_t* Key) {
  unsigned i, j, k;
  uint8_t tempa[4]; // Used for the column/row operations

  ......

  // All other round keys are found from the previous round keys.
  for (i = Nk; i < Nb * (Nr + 1); ++i) {
     k = (i - 1) * 4;
     tempa[0]=RoundKey[k + 0];
     tempa[1]=RoundKey[k + 1];
     tempa[2]=RoundKey[k + 2];
     tempa[3]=RoundKey[k + 3];

     if (i % Nk == 0) {

       const uint8_t u8tmp = tempa[0];
       tempa[0] = tempa[1];
       tempa[1] = tempa[2];
       tempa[2] = tempa[3];
       tempa[3] = u8tmp;

       tempa[0] = getSBoxValue(tempa[0]);
       tempa[1] = getSBoxValue(tempa[1]);
       tempa[2] = getSBoxValue(tempa[2]);
       tempa[3] = getSBoxValue(tempa[3]);
```

```
29
30        tempa[0] = tempa[0] ^ Rcon[i/Nk];
31      }
32      ......
33
34      j = i * 4; k=(i - Nk) * 4;
35      RoundKey[j + 0] = RoundKey[k + 0] ^ tempa[0];
36      RoundKey[j + 1] = RoundKey[k + 1] ^ tempa[1];
37      RoundKey[j + 2] = RoundKey[k + 2] ^ tempa[2];
38      RoundKey[j + 3] = RoundKey[k + 3] ^ tempa[3];
39    }
40  }
```

```
1  // Modified version due to HLS
2  static void KeyExpansion(uint8_t RoundKey[AES_keyExpSize], const uint8_t Key[
       AES_KEYLEN]) {
3
4    unsigned i, j, k;
5    uint8_t tempa_0, tempa_1, tempa_2, tempa_3;
6
7    ...... // remain the same
8
9    // All other round keys are found from the previous round keys.
10   for (i = Nk; i < Nb * (Nr + 1); ++i) {
11      k = (i - 1) * 4;
12      tempa_0=RoundKey[k + 0]; // replace tempa[0] with tempa_0
13      tempa_1=RoundKey[k + 1];
14      tempa_2=RoundKey[k + 2];
15      tempa_3=RoundKey[k + 3];
16
17      if (i % Nk == 0) {
18
19        const uint8_t u8tmp = tempa_0;
20        tempa_0 = tempa_1;
21        tempa_1 = tempa_2;
22        tempa_2 = tempa_3;
23        tempa_3 = u8tmp;
24
25        tempa_0 = getSBoxValue(tempa_0);
26        tempa_1 = getSBoxValue(tempa_1);
27        tempa_2 = getSBoxValue(tempa_2);
28        tempa_3 = getSBoxValue(tempa_3);
29
30        tempa_0 = tempa_0 ^ Rcon[i/Nk];
31      }
32
33      // remove the lines from #if to #endif as they are not related to AES-128.
```

```
34
35      j = i * 4; k=(i - Nk) * 4;
36      RoundKey[j + 0] = RoundKey[k + 0] ^ tempa_0;
37      RoundKey[j + 1] = RoundKey[k + 1] ^ tempa_1;
38      RoundKey[j + 2] = RoundKey[k + 2] ^ tempa_2;
39      RoundKey[j + 3] = RoundKey[k + 3] ^ tempa_3;
40    }
41  }
```

Modifications in `AES_init_ctx()`. Next, we make modifications on function `AES_init_ctx()` by replacing the pointers with arrays.

```
1  // Original version in TinyAES, line 219 in aes.c
2  void AES_init_ctx(struct AES_ctx* ctx, const uint8_t* key) {
3    KeyExpansion(ctx->RoundKey, key);
4  }
```

```
1  // Modified version due to HLS
2  void AES_init_ctx(uint8_t RoundKey[AES_keyExpSize], const uint8_t key[AES_KEYLEN])
3  {
4    KeyExpansion(RoundKey, key);
5  }
```

Removing lines that are not related to the ECB mode. One should also remove lines that are not associated with the ECB mode. These are typically contained in a `#if - #endif` block. For example, the following lines should be removed in the modified version.

```
1  // Original version in TinyAES, line 223 to line 233 in aes.c; Removing those
       lines in the modified version
2  #if (defined(CBC) && (CBC == 1)) || (defined(CTR) && (CTR == 1))
3  void AES_init_ctx_iv(struct AES_ctx* ctx, const uint8_t* key, const uint8_t* iv) {
4    KeyExpansion(ctx->RoundKey, key);
5    memcpy (ctx->Iv, iv, AES_BLOCKLEN);
6  }
7  void AES_ctx_set_iv(struct AES_ctx* ctx, const uint8_t* iv) {
8    memcpy (ctx->Iv, iv, AES_BLOCKLEN);
9  }
10 #endif
```

Modifications in *aes.h*. Once we complete all the above modifications, we need to make the associated changes in *aes.h*. Specifically, we remove all the function declarations that are not associated with the ECB mode.

```
1  // Modified version due to HLS
2
3  ..... // remain the same
4  #else
```

```
    #define AES_KEYLEN 16   // Key length in bytesW
    #define AES_keyExpSize 176
#endif

// remove struct AES_ctx as we avoid using struct in HLS

void AES_init_ctx(uint8_t RoundKey[AES_keyExpSize], const uint8_t key[AES_KEYLEN])
    ;

#if defined(ECB) && (ECB == 1)
void AES_ECB_encrypt(const uint8_t RoundKey[AES_keyExpSize], const uint8_t buf[
    AES_BLOCKLEN], uint8_t enc[AES_BLOCKLEN]);
#endif

// end of the file, remove the remaining lines
```

Turn Off Pipeline Optimization for `For` Loops. When we investigate our downstream hardware design, we find that `for` loops in C can create multiple time violations with the default control flow pipeline optimization in Vitis HLS. To address this issue, we decide to turn off the pipeline for every `for` loop. Specifically, we go through *aes.c* and add one additional line inside each `for` loop. One example code is listed below. As a tradeoff, it increases area and latency of our hardware design.

```
// Modified version due to HLS
...
for (i = 0; i < Nk; ++i) {
    #pragma HLS pipeline off  // add an additional line to turn off pipeline
    RoundKey[(i * 4) + 0] = Key[(i * 4) + 0];
    RoundKey[(i * 4) + 1] = Key[(i * 4) + 1];
    RoundKey[(i * 4) + 2] = Key[(i * 4) + 2];
    RoundKey[(i * 4) + 3] = Key[(i * 4) + 3];
}
...
```

If you reach this point, Congratulations! You have completed all the necessary modifications in the original TinyAES C code.

5.5 Step 3: Write the Main Source File

Next, we will need to write a main source file for our Vitis HLS component, which contains a top function. Given the inputs, the main source file will perform the program and provide an output. In our case, given two inputs, a plaintext and a key, this main source file will perform the function, i.e., AES128-ECB encryption (in our case), and outputs a ciphertext.

Although Vitis HLS supports both C (`.c`) and C++ (`.cpp`) as the format of a main source file, it is recommended to use C++. On the other hand, within a main source file in `.cpp` format, it is recommended to use the native C library and coding style as C++ functionalities may not be fully supported. Therefore, we write our main source file in C native code but save it as `.cpp`. Specifically, we name it as `test.cpp`.

Ideally, we should directly write the main source file with two inputs (plaintext and key) and one output (a ciphertext) for the final product. However, since this is still in the design phase, testing and verification is important. Our first version of this main source file is essentially a test case, which verifies the correctness of our design. Specifically, we will write a top function and we call our modified versions of function `AES_init_ctx()` and then function `AES_ECB_encrypt()` (from the last section) given a known plaintext and a known key inside the top function. Then, we compare the output with a known ciphertext to test the correctness of our modified C program inside this top function. To achieve this, we also rename our modified `aes.c` file as `aes_c.h` file such that we can leverage it as a header file in our main source file. Our first version of this main source file is listed below.

```
// Our first version of the main source file test.cpp for HLS
#include <cstdint>
#include <stdint.h>
#include <math.h>
#include <ap_int.h>
#include <stdio.h>
#include <ap_axi_sdata.h>
#include <sys/types.h>

#ifndef AES_H
  #define AES_H
  #include "aes.h"
#endif

#ifndef AES_C_H
  #define AES_C_H
  #include "aes_c.h"
#endif

static int test_encrypt_ecb(uint8_t key[16], uint8_t plaintext[16], uint8_t
    ciphertext[16], uint8_t RoundKey[AES_keyExpSize]){

    AES_init_ctx(RoundKey, key); // initialize the key with KeyExpansion()
    AES_ECB_encrypt(RoundKey, plaintext, ciphertext); // perform AES128-ECB

    return 0;
}
```

```
27
28 // Our top function
29 void run_test_input(){
30
31     uint8_t RoundKey[AES_keyExpSize];
32     // a known plaintext
33     uint8_t plaintext[16] = {0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9,
           0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a};
34     // a known key
35     uint8_t key[16] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
           0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
36     // a known ciphertext based on the plaintext and key
37     uint8_t expected_output[16] = {0x3a, 0xd7, 0x7b, 0xb4, 0x0d, 0x7a, 0x36, 0x60,
           0xa8, 0x9e, 0xca, 0xf3, 0x24, 0x66, 0xef, 0x97};
38
39     uint8_t ciphertext[16];
40
41     // call AES_init_ctx and AES_ECB_encrypt
42     test_encrypt_ecb(key, plaintext, ciphertext, RoundKey);
43
44     // compare each byte in the ciphertext one by one, if correct, pass remains as
           0x50, otherwise it is updated to 0x10
45     int pass = 0x50;
46     for (int i = 0; i < 16; i ++) {
47         if (enc[i] != expected_output[i]) {
48             pass = 0x10;
49             break;
50         }
51     }
52
53     // report our verification result
54     if (pass == 0x50) {
55         printf("Passes\n");
56     } else {
57         printf("Fails\n");
58     }
59 }
```

While the first version of the main file `test.cpp` is sufficient for us to verify the correctness of the C program, it is not sufficient for us to generate the hardware design in Verilog or VHDL using Vitis HLS. This is because (1) we need to pass plaintext and key as input and provide the ciphertext as output of an IP that we will generate based on the RTL-level design; (2) we need to modify the code to make it Vitis HLS compatible. Therefore, we need to make some modifications in the `test.cpp`.

First, we add multiple arguments to the top function. We add `ap_input<8> *p` to hold the result of the comparison check on the ciphertext. Vitis HLS uses `ap_int<N>` to define arbitrary precision integer data type, where N is the bit-size of the integer and N can be from 1 to 1024. We also add four arguments, `uint8_t plaintext[16]` to hold the plaintext as an input, `uint8_t key[16]` to hold the key as an input, `uint8_t expected_ciphertext[16]` to hold the expected ciphertext (pre-calculated) as an input, and `uint8_t ciphertext[16]` to hold the ciphertext as an output. Keeping the result of the comparison check will allow us to perform the on-device verification later. Note that since we only use `*p` once (assigning a value to it), there is only one operation for the pointer, which is fine. We do not need to replace this pointer with an array as before.

Besides adding these arguments, we also need to add HLS parameters in the main source file to (1) handle FPGA I/O ports, (2) optimize the RAM usage for S-box and reverse S-box, and (3) turn off optimizations on `for` loops. Specifically, Vitis HLS packages synthesized modules into an FPGA IP which can be loaded into Vivado for block design later. To configure I/O ports, we need to add HLS parameter ***#pragma HLS INTERFACE mode=<mode> port=<name> [OPTIONS]*** in the code. More information can be found in the following link [3]. By default, Vitis HLS configures every input/output of the top function with mode `ap_none` (only data port with no associated signal). In our investigation, we find that the best way to build input and output port is using mode `ap_ovld` (port with valid signal if data is ready). Although the return port is not entirely relevant in our example, one should configure return port as either `ap_ctrl_hs`, `ap_ctrl_none`, or `s_axilite`. For normal use (in our example), the return port is tied to `ap_ctrl_none`.

In our example, TinyAES makes use of two large arrays, one for `sbox` (S box) and one for `rsbox` (Reverse S box). This can consume much RAM usage for the synthesized model. To address this, we use ***#pragma HLS array_parition variable=... complete*** to split a large array into individual variables to reduce RAM usage. There is a `for` loop when we check the correctness of the output ciphertext. Similar as modifying the code in *aes.c*, we add a line with ***#pragma HLS pipeline off*** within this loop to turn optimization off. The `printf()` function is also removed as HLS (in general) does not support `printf()` function. We basically move this part of the code related to the `printf()` function to our testbench file later.

With the above updates, we now have our second version of our main source file below

```
// Our second version of the main source file test.cpp for HLS
... // remain the same
```

[3] `https://docs.amd.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas`

```
void run_test_input(ap_uint<8> *p, uint8_t ciphertext[16], uint8_t plaintext[16],
    uint8_t key[16], uint8_t expected_ciphertext[16]) {
    // set the FPGA I/O ports
    #pragma HLS INTERFACE mode=ap_ovld port=plaintext
    #pragma HLS INTERFACE mode=ap_ovld port=key
    #pragma HLS INTERFACE mode=ap_ovld port=expected_ciphertext
    #pragma HLS INTERFACE mode=ap_ovld port=ciphertext
    #pragma HLS INTERFACE mode=ap_ovld port=p
    #pragma HLS INTERFACE mode=ap_ctrl_none port=return  // this line needs to be
        disabled when run C sythesis and C/RTL-simulation but enabled when run C
        sythesis and generate the IP

    // reduce RAM usage for S box and reverse S box
    #pragma HLS array_partition variable=sbox type=complete
    #pragma HLS array_partition variable=rsbox type=complete

    uint8_t RoundKey[AES_keyExpSize];

    uint8_t ciphertext_temp[16];

    test_encrypt_ecb(key, plaintext, ciphertext_temp, RoundKey);

    int pass = 0x50;

    for (int i = 0; i < 16; i++){
        // turn optimization off for this for loop
        #pragma HLS pipeline off
        if (ciphertext_temp[i] != expected_ciphertext[i]) {
            pass = 0x10;
            break;
        }
    }

    // copy ciphertext
    for (int i = 0; i < 16; i++){
        ciphertext[i] = ciphertext_temp[i];
    }

    *p = pass;
}
```

5.6 Step 4: Write the Testbench file

After we create the main source file above, we will need to create the testbench file, which is for running the C simulation and (later) C/RTL co-simulation of the entire design in our HLS component. We name our testbench file as `testbench_simulation.cpp`. This

testbench file contains a `main()` function. In the `main()` function, we read key, plaintext, and expected ciphertext from `key.txt`, `plaintext.txt`, and `expected_ciphertext.txt`, call the top function in the main source file to run AES-128-ECB encryption, and print out the result of the verification. It is worth mentioning that, unlike the source files above (i.e., *aes.h*, *aes_c.h*, *test.cpp*), this testbench file is only for C simulation and C/RTL co-simulation, and will not be part of the hardware design on FPGA. In other words, we do not need to apply the previous HLS principles to modify the C code in the testbench file.

It is also worth mentioning that we particularly load key, plaintext, and expected ciphertext from files rather than hard-coding them in the testbench file. Although this leads to more lines of code in the testbench file to handle reading content from files, it scales well when we need to replace plaintexts or keys for generating different simulated traces for our pre-silicon side-channel analysis. The code of this testbench file is presented below.

```
// Our testbench file testbench_simulation.cpp for C simulation
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdbool.h>
#include <sys/types.h>
#include <ap_int.h>

// a function transforms characters from hex to decimal
int hex_to_decimal(char hex) {
    switch(hex) {
        case '0':
            return 0;
            break;
        case '1':
            return 1;
            break;
        case '2':
            return 2;
            break;
        case '3':
            return 3;
            break;
        case '4':
            return 4;
            break;
        case '5':
            return 5;
            break;
        case '6':
```

```
32              return 6;
33              break;
34          case '7':
35              return 7;
36              break;
37          case '8':
38              return 8;
39              break;
40          case '9':
41              return 9;
42              break;
43          case 'a':
44              return 10;
45              break;
46          case 'b':
47              return 11;
48              break;
49          case 'c':
50              return 12;
51              break;
52          case 'd':
53              return 13;
54              break;
55          case 'e':
56              return 14;
57              break;
58          case 'f':
59              return 15;
60              break;
61          default:
62              return 0;
63              break;
64      }
65  }
66
67  // a function transforms a string to a an array of integers.
68  void hexstring_to_uint8_tarray(char input[], int input_size, uint8_t *
        output_uint8_t) {
69      for (int i = 0; i < input_size; i += 2) {
70          output_uint8_t[i / 2] = (uint8_t)(16 * hex_to_decimal(input[i]) +
                hex_to_decimal(input[i + 1]));
71      }
72  }
73
74  // a function prints out an array of integers in hex for easier observation from
        the terminal.
75  static void phex(uint8_t* str) {
```

```
76      uint8_t len = 16;
77      unsigned char i;
78      for (int i = 0; i < len; ++i) {// int declariation was missing in the code, at
            least from the version I have, could you please double check?
79          printf("%.2x", str[i]);
80      }
81      printf("\n");
82  }
83
84  // the name of our top function defined in our main source file
85  void run_test_input(ap_uint<8> *p, uint8_t ciphertext[16], uint8_t plaintext[16],
        uint8_t key[16], uint8_t expected_ciphertext[16]);
86
87  // The main function
88  int main(){
89
90      int exit = 0;
91      FILE *file_key;
92      FILE *file_in;
93      FILE *file_out;
94      ap_uint<8> p;
95      uint8_t ciphertext[16];
96
97      int key_size = 16 * 2;
98      int plaintext_size = 16 * 2;
99      int expected_ciphertext_size = 16 * 2;
100
101     char key_str[key_size + 1];
102     char plaintext_str[plaintext_size + 1];
103     char expected_ciphertext_str[expected_ciphertext_size + 1];
104
105     uint8_t key_uint8_t[key_size];
106     uint8_t plaintext_uint8_t[plaintext_size];
107     uint8_t expected_ciphertext_uint8_t[expected_ciphertext_size];
108
109     // read the key
110     file_key = fopen("./key.txt", "r");
111     if (NULL == file_key) {
112         quick_exit(0);
113     }
114     fgets(key_str, sizeof(key_str), file_key);
115     fclose(file_key);
116     hexstring_to_uint8_tarray(key_str, strlen(key_str), key_uint8_t);
117
118     // read the plaintext
119     file_in = fopen("./plaintext.txt", "r");
120     if (NULL == file_in) {
```

```
121         quick_exit(0);
122     }
123     fgets(plaintext_str, sizeof(plaintext_str), file_in);
124     fclose(file_in);
125     hexstring_to_uint8_tarray(plaintext_str, strlen(plaintext_str),
            plaintext_uint8_t);
126
127     // read the expected ciphertext
128     file_out = fopen("./expected_ciphertext.txt", "r");
129     if (NULL == file_out) {
130         quick_exit(0);
131     }
132     fgets(expected_ciphertext_str, sizeof(expected_ciphertext_str), file_out);
133     fclose(file_out);
134     hexstring_to_uint8_tarray(expected_ciphertext_str, strlen(
            expected_ciphertext_str), expected_ciphertext_uint8_t);
135
136     printf("\nSimulation␣starts:␣\n");
137     printf("Key:␣");
138     phex(key_uint8_t);
139     printf("\n");
140     printf("Plaintext:␣");
141     phex(plaintext_uint8_t);
142     printf("\n");
143     printf("Expected␣output:␣");
144     phex(expected_ciphertext_uint8_t);
145     printf("\n");
146
147     // call our top function
148     run_test_input(&p, ciphertext, plaintext_uint8_t, key_uint8_t,
            expected_ciphertext_uint8_t);
149
150     // show the verification result
151     printf("p␣=␣%d\n", p);
152     if (p == 0x50) {
153        printf("Passes\n");
154     } else {
155        printf("Fails\n");
156     }
157     printf("\nSimulation␣ends:␣\n");
158
159     return 0;
160 }
```

We also create *key.txt*, *plaintext.txt*, and *expected_ciphertext.txt* by providing the following key, plaintext, and expected ciphertext, respectively. They are the same key, plaintext, and expected ciphertext we used in the first version of our main source file *test.cpp*.

```
// a known plaintext in plaintext.txt
6bc1bee22e409f96e93d7e117393172a
```

```
// a known key in key.txt
2b7e151628aed2a6abf7158809cf4f3c
```

```
// a known output ciphertext in expected_ciphertext.txt
3ad77bb40d7a3660a89ecaf32466ef97
```

5.7 Step 5: Specify the Top Function and Run C Simulation

Given all the modified source files and the testbench files, we will need to add them and specify the top function in our HLS component for TinyAES in order to run C simulation and downstrean tasks. To update these configurations, we just need to update the configuration file (named as `hls_config.cfg` in our example) using one of the two ways:

1. Manually modify the file `hls_config.cfg`, which can be found at `/hls_config.cfg`.
2. Leverage Vitis's GUI to modify the file `hls_config.cfg`, which can be found under the HLS component we created in Step 1 (as shown in Fig. 14).

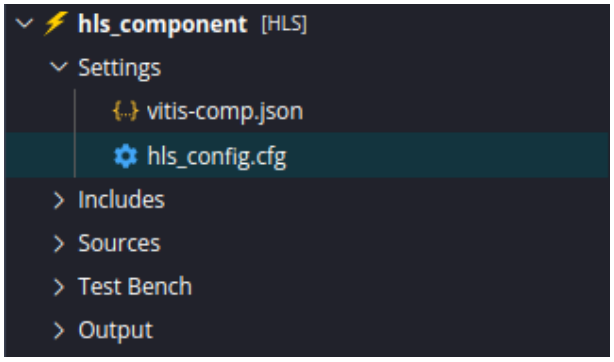


Fig. 14: The configuration file can be found under "Settings" of our HLS component.

In this example, we add `testbench_simulation.cpp`, `key.txt`, `plaintext.txt`, *expected_ciphertext.txt* as testbench files, `aes.h`, `aes_c.h`, and `test.cpp` as input files, and `run_top_input` as top function name. The content of the configuration file `hls_config.cfg` after these updates is presented below:

```
part=xc7a100tcsg324-1 # part for Digilient Arty A7 Artix-7 FPGA board

[hls]
syn.top=run_test_input   # set the top function
tb.file=key.txt  # added as a testbench file
tb.file=plaintext.txt  # added as a testbench file
tb.file=expected_ciphertext.txt  # added as a testbench file
tb.file=testbench_simulation.cpp  # added as a testbench file
cosim.trace_level=all # capture all signal when running co-simulation
csim.code_analyzer=0
syn.interface.clock_enable=0
syn.file=aes.h   # added as a source file
syn.file=aes_c.h  # added as a source file
syn.file=test.cpp  # added as a (main) source file containing the top function
```

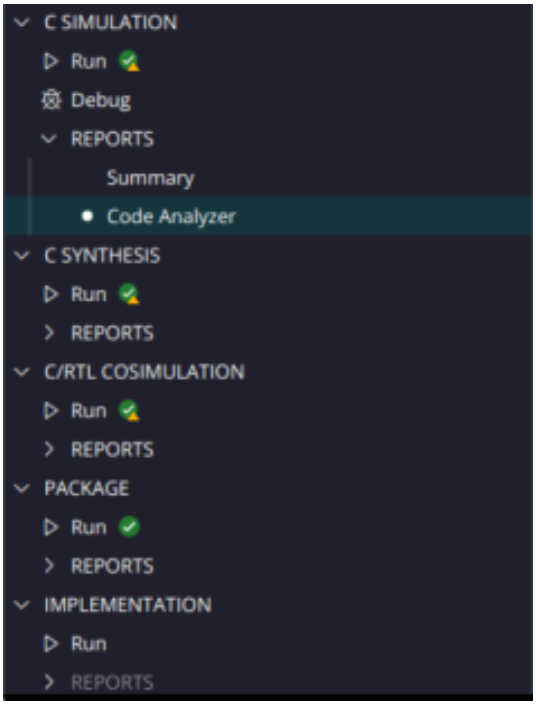


Fig. 15: Functionalities within a HLS component

Once we complete the above steps, we can run C simulation. Specifically, we can click "Run" under "C Simulation" of our HLS component as shown in Fig. 15. Vitis compiles the C code and generates binaries using `Clang` as the compiler. The results of a successful C simulation can be found below. Besides the information we intend to print out, Vitis HLS also reports CPU time, peak memory usage, and total time, which is 7 seconds in this example.

```
Simulation starts:
Key: 2b7e151628aed2a6abf7158809cf4f3c
Plaintext: 6bc1bee22e409f96e93d7e117393172a
Expected output: 3ad77bb40d7a3660a89ecaf32466ef97
p = 80
Passes

Simulation ends:
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
INFO: [HLS 200-111] Finished Command csim_design CPU user time: 1.29 seconds. CPU
    system time: 0.25 seconds. Elapsed time: 1.56 seconds; current allocated memory
    : 0.000 MB.
INFO: [HLS 200-1510] Running: close_project
INFO: [HLS 200-112] Total CPU user time: 3.42 seconds. Total CPU system time: 0.52
     seconds. Total elapsed time: 3.73 seconds; peak allocated memory: 274.531 MB.
INFO: [Common 17-206] Exiting vitis_hls at Mon Jan 13 17:00:58 2025...
INFO: [vitis-run 60-791] Total elapsed time: 0h 0m 7s
C-simulation finished successfully
```

If the C simulation is successful, one can move on to the next step. On the other hand, if there is any compilation issue, an error flag will be raised when running the C simulation and the simulation will break immediately. In that case, one will need to debug the testbench file and source files to ensure everything is correct.

5.8 Step 6: Run HLS to Generate Verilog/VHDL Code

Once the above C simulation is successful, we can click "Run" under "C Synthesis" to generate our design at the RTL level. Vitis HLS is capable of generating the RTL level design in Verilog and VHDL separately but at the same time without any further specification or configuration. In other words, we obtain two versions of the RTL level design after we click "Run" under "C Synthesis", one in Verilog and one in VHDL. The version in Verilog can be found in `<path_to_hls_component>/hls/syn/verilog` and the version in VHDL can be found under `<path_to_hls_component>/hls/syn/vhdl`

Special Note. It is worth mentioning that before we run "C Synthesis", if we would like to run the later C/RTL co-simulation based on the design generated by C Synthesis, we need to comment out the return port tied to `ap_ctrl_none`, i.e., line 7, in function `run_test_input` in main source file *test.cpp*.

```
// comment out this line if one would like to generate the RTL code and run C/RTL
    co-simulation, however, still enable this line if one would like to generate
    the RTL code and other downstream steps.
```

```
#pragma HLS INTERFACE mode=ap_ctrl_none port=return  // line 7 in function
    run_test_input in test.cpp
```

Otherwise, the following errors will occur later on during the C/RTL co-simulation.

```
[ERROR] ERROR: [COSIM 212-345] Cosim only supports the following 'ap_ctrl_none'
    designs: (1) combinational designs; (2) pipelined design with II of 1; (3)
    designs with array streaming or hls_stream or AXI4 stream ports.
[ERROR] ERROR: [COSIM 212-5] *** C/RTL co-simulation file generation failed. ***
[ERROR] ERROR: [COSIM 212-4] *** C/RTL co-simulation finished: FAIL ***
 INFO: [HLS 200-111] Finished Command cosim_design CPU user time: 0.22 seconds.
     CPU system time: 0.06 seconds. Elapsed time: 0.29 seconds; current allocated
     memory: 3.336 MB.
 INFO: [HLS 200-1510] Running: close_project
 ERROR:
 INFO: [HLS 200-112] Total CPU user time: 2.31 seconds. Total CPU system time:
     0.35 seconds. Total elapsed time: 2.45 seconds; peak allocated memory: 277.801
      MB.
 INFO: [Common 17-206] Exiting vitis_hls at Tue Jan 14 11:37:17 2025...
[ERROR] Failed to run co-simulation
```

A successful example of C Synthesis is presented below as a reference.

```
INFO: [HLS 200-111] Finished Creating RTL model: CPU user time: 0.25 seconds. CPU
    system time: 0.02 seconds. Elapsed time: 0.28 seconds; current allocated memory
    : 354.008 MB.
INFO: [HLS 200-111] Finished Generating all RTL models: CPU user time: 0.34
    seconds. CPU system time: 0.02 seconds. Elapsed time: 0.37 seconds; current
    allocated memory: 362.719 MB.
INFO: [HLS 200-111] Finished Updating report files: CPU user time: 0.48 seconds.
    CPU system time: 0.02 seconds. Elapsed time: 0.5 seconds; current allocated
    memory: 366.617 MB.
INFO: [VHDL 208-304] Generating VHDL RTL for run_test_input.
INFO: [VLOG 209-307] Generating Verilog RTL for run_test_input.
INFO: [HLS 200-790] **** Loop Constraint Status: All loop constraints were
    satisfied.
INFO: [HLS 200-789] **** Estimated Fmax: 138.33 MHz
INFO: [HLS 200-111] Finished Command csynth_design CPU user time: 9.5 seconds. CPU
     system time: 1.32 seconds. Elapsed time: 15.19 seconds; current allocated
    memory: 92.266 MB.
INFO: [HLS 200-1510] Running: close_project
INFO: [HLS 200-112] Total CPU user time: 11.67 seconds. Total CPU system time: 1.6
     seconds. Total elapsed time: 17.45 seconds; peak allocated memory: 366.750 MB.
INFO: [Common 17-206] Exiting vitis_hls at Mon Jan 13 17:15:07 2025...
INFO: [v++ 60-791] Total elapsed time: 0h 0m 21s
Synthesis finished successfully
```

The Verilog version of our RTL level design contains 17 files, as shown in 16. The .dat files are used for storing ROM data (key, expected output, etc. in our source code). The remaining Verilog files contain top function module and other modules. The total number of lines of the entire design is about 12,000 across all the *.v* files. (`run_test_input.v` is generated from our top function, and serves as the top module for our RTL design. The first 25 lines of `run_test_input.v` are presented below as a reference.

```
// ==============================================================
// Generated by Vitis HLS v2023.2
// Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
// Copyright 2022-2023 Advanced Micro Devices, Inc. All Rights Reserved.
// ==============================================================

`timescale 1 ns / 1 ps

(* CORE_GENERATION_INFO="run_test_input_run_test_input,hls_ip_2023_2,{
    HLS_INPUT_TYPE=cxx,HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED=0,HLS_INPUT_PART=xc7a100t-
    csg324-1,HLS_INPUT_CLOCK=10.000000,HLS_INPUT_ARCH=others,HLS_SYN_CLOCK
    =7.229000,HLS_SYN_LAT=1918,HLS_SYN_TPT=none,HLS_SYN_MEM=1,HLS_SYN_DSP=0,
    HLS_SYN_FF=514,HLS_SYN_LUT=10375,HLS_VERSION=2023_2}" *)

module run_test_input (
    ap_clk,
    ap_rst,
    ap_start,
    ap_done,
    ap_idle,
    ap_ready,
    p,
    p_ap_vld,
    out1,
    out1_ap_vld,
    plaintext_address0,
    plaintext_ce0,
    plaintext_q0
);
......
```

At this point, we have successfully generated a design at the RTL level.

5.9 Step 7: Run C/RTL Co-Simulation (Optional)

After we generate the design at the RTL level, we can (optionally) choose to perform C/RTL co-simulation. Specifically, we can run the C/RTL co-simulation by click “Run”

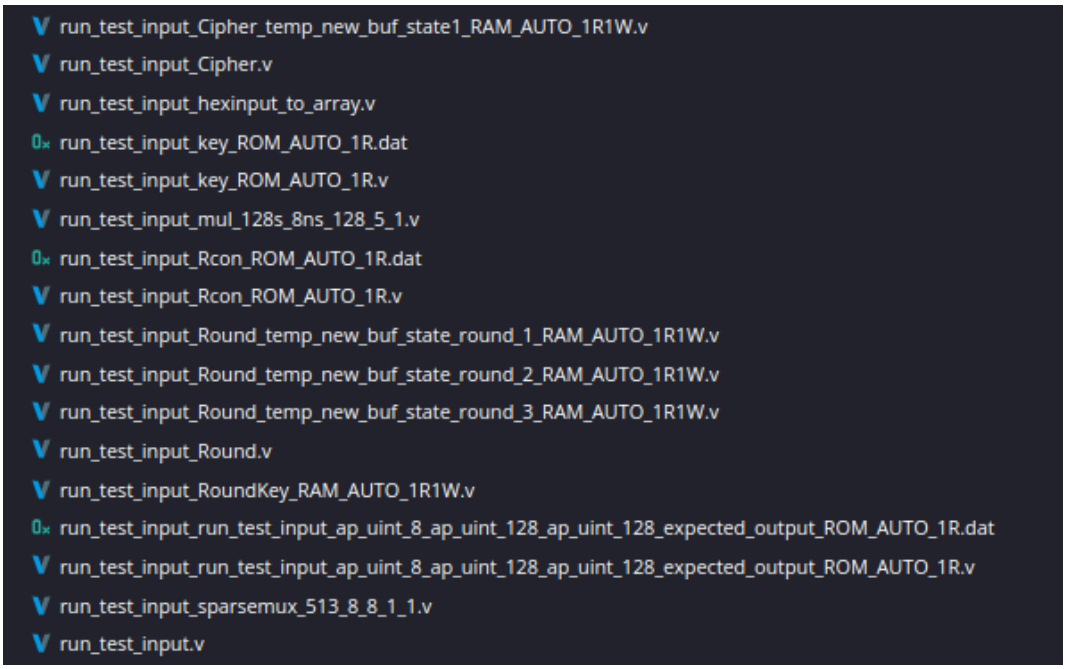


Fig. 16: File generated after running C Synthesis

under "C/RTL Co-simulation" The testbench file, i.e., *testbench_simulation.cpp*, is used for both C/RTL co-simulation. No additional files need to be created. The purpose of C/RTL Co-simulation is to ensures the results of the C program and the RTL level design are consistent given the same testbench. It is done by running the simulation with the stimuli from C testbench input, capturing the output data from the simulation waveform, and check the waveform against the results produced by the original C source. A successful example of C/RTL co-simulation is presented below as a reference. It took 31 seconds to complete the C/RTL co-simulation.

```
Simulation starts:
Key: 2b7e151628aed2a6abf7158809cf4f3c
Plaintext: 6bc1bee22e409f96e93d7e117393172a
Expected output: 3ad77bb40d7a3660a89ecaf32466ef97

p = 80
Passes

Simulation ends:
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
INFO: [COSIM 212-211] II is measurable only when transaction number is greater
    than 1 in RTL simulation. Otherwise, they will be marked as all NA. If user
    wants to calculate them, please make sure there are at least 2 transactions in
    RTL simulation.
```

```
INFO: [HLS 200-111] Finished Command cosim_design CPU user time: 25.2 seconds. CPU system time: 2.39 seconds. Elapsed time: 25.4 seconds; current allocated memory: 16.328 MB.
INFO: [HLS 200-1510] Running: close_project
INFO: [HLS 200-112] Total CPU user time: 27.36 seconds. Total CPU system time: 2.66 seconds. Total elapsed time: 27.62 seconds; peak allocated memory: 290.793 MB.
INFO: [Common 17-206] Exiting vitis_hls at Mon Jan 13 17:18:18 2025...
INFO: [vitis-run 60-791] Total elapsed time: 0h 0m 31s
Co-simulation finished successfully
```

5.10 Generate a Simulated Trace from C/RTL Co-Simulation (Optional)

Generating a VCD File for Pre-Silicon Side-Channel Analysis. For our research in pre-silicon side-channel analysis, we also would like to save the waveform from the RTL simulation into a VCD file, which can be used to generate a simulated trace of an AES execution based on our RTL design. There are two opportunities to generate a VCD file in our pipeline, one is from C/RTL co-simulation in Vitis HLS and one is from simulation of the final design in Vivado. While we prefer to obtain the VCD file from the final design in Vivado later as the one we run in C/RTL co-simulation still carry many verification components, we describe the process of generating a VCD file from C/RTL co-simulation below.

As Vitis HLS C/RTL co-simulation does not output a VCD file directly, we need to make some additional changes in the setting of the HLS component. In our example, we follow the instructions from [4] to dump the VCD file. Specifically, assuming that we have run the above C/RTL simulation at least once, we go to `<path_to_hls_component>/hls/sim/verilog` and find the following two files.

```
run_xsim.sh
run_test_input.tcl  # the name is based on the name of your top function
```

In file *run_xsim.sh*, add `-debug all` option to the `xelab` line (likely the first line) to enable trace logging. In file *run_test_input.tcl*, add the following lines at the beginning of the file to dump VCD

```
open_vcd
log_vcd [get_object /*]
run all
close_vcd
quit
```

[4] https://lyftfc.github.io/research/fpga/2022/01/23/vitis-hls-xsim-wvf.html

After the updates, we run the RTL simulation again (by running `sim.sh` in the same directory), and a *dump.vcd* file should be generated under `<path_to_hls_component>/hls/sim/verilog`. An example of the first 20 lines of *dump.vcd* is presented below as a reference.

```
$date
   Thu Jan 16 14:02:30 2025
$end

$version
  2023.2
$end

$timescale
  1ps
$end

$scope module apatb_run_test_input_top $end
$var reg 1 ! AESL_clock $end
$var reg 1 " rst $end
$var reg 1 # dut_rst $end
$var reg 1 $ start $end
$var reg 1 % ce $end
$var reg 1 & tb_continue $end
$var wire 1 ' AESL_start $end
```

Generating a Simulated Trace from a VCD File. Once we obtain a VCD file above, we need to generate a simulated trace for pre-silicon side-channel analysis. This simulated trace represents the estimated power consumption of the RTL design of AES ecnryption at each timestamp given a plaintext and a key. To generate a simulated trace from a VCD file, we utilize TOFU [9], which parses a VCD file and calculates/simulates power consumption based on toggle counts in the VCD file. Assuming that TOFU has been installed correctly, there are two factors we need to keep in mind when we generate a simulated trace from a VCD file.

1. TOFU cannot parse empty lines in a VCD file
2. TOFU cannot parse `integer` datatype in a VCD file.

In our VCD file *dump.vcd*, there are multiple empty lines (e.g., line 4, 8, 12, etc.) and multiple lines with `integer` datatype (e.g., line 54, 55, and 56). We write the following Python script to (1) remove empty lines and (2) replace `integer` with `reg` in a VCD file.

```
# Our fix_vcd.py file to process a VCD file for simulated trace generation
import os

vcd_file = "./dump.vcd"
```

```
5
6  with open("{0}".format(vcd_file), "r") as f:
7      lines = f.readlines() # read all lines
8      tmp_lines = []
9      for i in range(len(lines)):
10         tmp = None
11
12         if "integer" in lines[i]:
13             tmp = lines[i].replace("integer", "reg") # change integer into reg
14         else:
15             tmp = lines[i]
16
17         if tmp != "\n":
18             tmp_lines.append(tmp) # remove any empty line
19
20 with open("{0}".format(vcd_file), "w") as f:
21     for line in tmp_lines:
22         f.write(line) # update original file
```

After we execute the above script, we can run TOFU on (processed) *dump.vcd* to generate a simulated trace. Specifically, we go to TOFU's installation path, create a directory named "traces" and copy the (processed) VCD file *dump.vcd* to this directory. Next, we go to this directory and create/modify the TOFU setting file `settings_example.json` as below:

```
1  # settings_example.json
2  {
3      "vcdGlob": "dump.vcd",
4      "pickleGlob": "dump.pickle",
5      "signalsFileNameLiterals": "signals_name.json",
6      "signalsFileName": "signals.json",
7      "signalPropertiesFile": "signal_properties.pickle",
8      "leakageModel": "HammingWeight",
9      "window": false,
10     "windowFrom": null,
11     "windowTo": null,
12     "valueExtractFunction": "valueExtractIndex",
13     "writeTraces": true,
14     "writeTracesBatchSize": 10,
15     "traceFileName": "dump.h5",
16     "align": false,
17     "downsample": 1e5,
18     "format": "lascar"
19 }
```

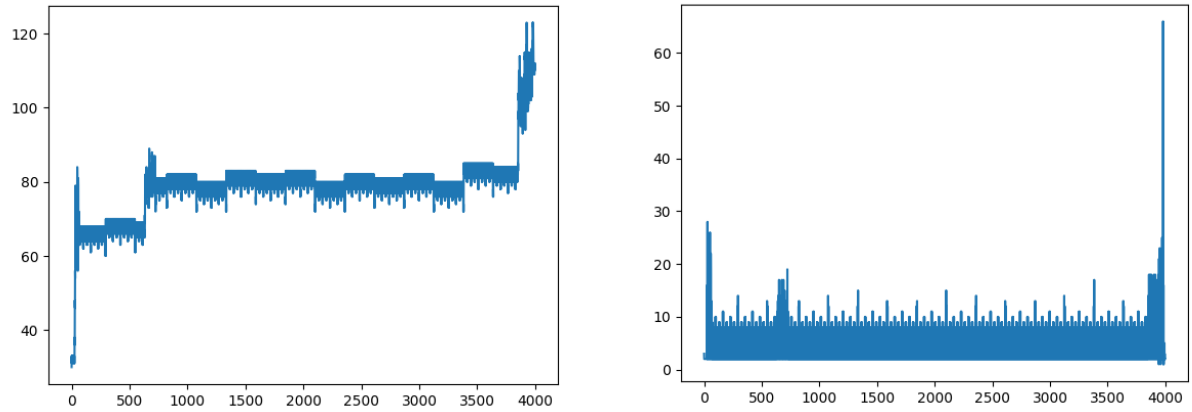
In this particular example, we need to specify four parameters in the setting file. `vcdGlob` (dump.vcd) is the name of the VCD file we use, `pickleGlob` (dump.pickle) is the name of

the pickle file that will be generated based on the given VCD file. `leakageModel` is the side-channel leakage model we would like to choose (either HammingWeight or HammingDistance). `traceFileName` (dump.h5) is the name of the output file, which saves the simulated trace. We leave other parameters as default.

We then go back to TOFU's installation path to run the following two Python scripts from TOFU:

```
parse.py --settings ./traces/settings_example.json
synthesize.py --settings ./traces/settings_example.json
```

After running the two scripts, one simulated trace is saved in `./traces/dump.h5`. A visual example of a simulated trace in Hamming Distance (or Hamming Weight) can be found in 17



(a) The trace using Hamming Weight

(b) The trace using Hamming Distance

Fig. 17: Two simulated traces generated by TOFU respectively from the same *dump.vcd*

5.11 Step 8: Generate IP

At this point, we have successfully created a RTL level design for AES that can pass the simulation at both C and RTL level given our testbench. The next step is to pack the design into an FPGA IP, which will be loaded into Vivado later to generate a bitstream for an FPGA (in our case, an Arty A7 FPGA board).

To make that happen, we will need to revisit our code to make some minor changes in our main source file (*test.cpp*) and our HLS component configuration file (*hls_config.cfg*), and then rerun the C Synthesis without running the C/RTL simulation. This is because that

some code in our design is compatible with C/RTL simulation but is not compatible with the generation of the IP/bitstream for FPGAs).

Special Note 1. We commented out the return port tied to `ap_ctrl_none`, i.e., line 7, in function `run_test_input` in main source file *test.cpp* previously to avoid errors in C/RTL simulation. Now, we need to enable this line again.

```
1 #pragma HLS INTERFACE mode=ap_ctrl_none port=return  // line 7 in function
      run_test_input in test.cpp
```

Special Note 2. In our current version of the top function (`run_test_input`) in our main source file, we have the following

```
1 // current version for C Synthesis and C/RTL co-simulation
2 void run_test_input(ap_uint<8> *p, uint8_t ciphertext[16], uint8_t plaintext[16],
      uint8_t key[16], uint8_t expected_ciphertext[16]) {
3     ......
4 }
```

When we synthesize a top function into an FPGA IP, each argument of the top function `run_test_input` is translated into an input/output port of the FPGA IP. Given arrays as arguments, i.e., (`ciphertext`, `plaintext`, `key`, and `expected_ciphertext` in our case, it is complicated to work with when it comes to block design and generate the final bitstream. To simplify the process, we decide to use a 128-bit variable to represent an array argument in our current version. For instance, we use `ap_uint<128> plaintext_128` to replace `uint8_t plaintext[16]` (`ap_uint<128>` indicates a 128-bit unsigned integer). The updated version of the declaration of the top function `run_test_input` is presented below as a reference.

```
1 // updated version for C Synthesis and IP Packaging
2 void run_test_input(ap_uint<8> *p, ap_uint<128> *ciphertext_128, ap_uint<128>
      plaintext_128, ap_uint<128> key_128, ap_uint<128> expected_ciphertext_128) {
3     ......
4 }
```

While we can pass plaintext, key, expected_ciphertext, ciphertext as input/output with the way above, each one is still a single 128-bit integer, which still needs to be further sliced into 16 bytes defined in `uint8_t` for operations within AES. Therefore, we write another additional `take_input.h` file to handle this transformation of a 128-bit integer `ap_uint<128>` to an array of 16 8-bit integers `uint8_t` (or visa versa for `ciphertext_128`). The logic of this transformation is shown in 18. The details of this file can be found in our repository.

As a result, some code in the top function also need to be updated. Version 3 of our top function, `run_test_input`, is presented below as a reference.

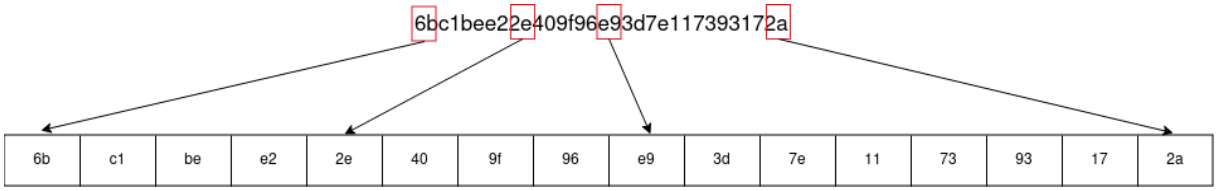


Fig. 18: Slicing a 128 bit integer into an array of 16 8- bit integers

```
// updated version of top function for running C Sythesis and IP Generation
    without running C/RTL-Simulation
void run_test_input(ap_uint<8> *p, ap_uint<128> *ciphertext_128, ap_uint<128>
    plaintext_128, ap_uint<128> key_128, ap_uint<128> expected_ciphertext_128){

uint8_t key[16], uint8_t expected_output[16]){
#pragma HLS INTERFACE mode=ap_ovld port=plaintext_128
#pragma HLS INTERFACE mode=ap_ovld port=ciphertext_128
#pragma HLS INTERFACE mode=ap_ovld port=key_128
#pragma HLS INTERFACE mode=ap_ovld port=expected_ciphertext_128
#pragma HLS INTERFACE mode=ap_ovld port=p
#pragma HLS INTERFACE mode=ap_ctrl_none port=return

#pragma HLS array_partition variable=sbox type=complete
#pragma HLS array_partition variable=rsbox type=complete

    uint8_t RoundKey[AES_keyExpSize];

    uint8_t ciphertext[16];
    uint8_t plaintext[16];
    uint8_t expected_ciphertext[16];
    uint8_t key[16];

    take_input(plaintext_128, plaintext);
    take_input(key_128, key);
    take_input(expected_ciphertext_128, expected_ciphertext);

    test_encrypt_ecb(key, plaintext, ciphertext, RoundKey);

    int pass = 0x50;

    for (int i = 0; i < 16; i ++){
        #pragma HLS pipeline off
        if (ciphertext[i] != expected_ciphertext[i]){
            pass = 0x10;
            break;
        }
    }

```

```
    ap_uint<128> ciphertext_temp = 0;

    for (int i = 0; i < 16; i++){
        #pragma HLS pipeline off
        ap_uint<128> power_16 = 1;
        for (int j = 0; j < (15 - i) * 2; j++){
            #pragma HLS pipeline off
            power_16 *= 16;
        }
        ciphertext_temp += power_16*ciphertext[i];
    }


    *p = pass;
    *ciphertext_128 = ciphertext_temp;
}
```

In addition, we also need to add `take_input.h` as an additional header file in our main source file *test.cpp* and also include it as an additional source file in our configuration file `hls_config.cfg` of HLS component.

```
    # updated version of hls\_config.cfg
    part=xc7a100tcsg324-1

    [hls]
    syn.top=run_test_input
    tb.file=key.txt
    tb.file=plaintext.txt
    tb.file=expected_ciphertext.txt
    tb.file=testbench_simulation.cpp
    cosim.trace_level=all
    csim.code_analyzer=0
    syn.interface.clock_enable=0
    syn.file=aes.h
    syn.file=aes_c.h
    syn.file=take_input.h  # added as a source file
    syn.file=test.cpp
```

With all the updates above, we rerun C Synthesis by clicking "Run" under "C Synthesis" to generate our design at the RTL level again. Then, we skip C/RTL-Co-simulation and we execute "Run" under "Package" in Fig. 15. The output of a successful Package is listed below for reference. The generated IP is included in a zip file (`run_test_input.zip`), which can be found under `<path to hls component>/<hls component name>`. The detailed structure of the zip file is shown in Fig. 19.

```
INFO: [IP_Flow 19-234] Refreshing IP repositories
```

```
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository '/tools/Xilinx/Vivado/2023.2/data/ip'.
INFO: [Common 17-206] Exiting Vivado at Mon Jan 20 12:29:07 2025...
INFO: [HLS 200-802] Generated output file hls_component2/run_test_input.zip
INFO: [HLS 200-111] Finished Command export_design CPU user time: 15.99 seconds. CPU system time: 0.66 seconds. Elapsed time: 26.11 seconds; current allocated memory: 6.840 MB.
INFO: [HLS 200-1510] Running: close_project
INFO: [HLS 200-112] Total CPU user time: 18.07 seconds. Total CPU system time: 0.92 seconds. Total elapsed time: 28.3 seconds; peak allocated memory: 281.336 MB.
INFO: [Common 17-206] Exiting vitis_hls at Mon Jan 20 12:29:17 2025...
INFO: [vitis-run 60-791] Total elapsed time: 0h 0m 32s
Package finished successfully
```

```
phucmai@mabon-OptiPlex-7050:~/Vitis/AES/Rewrite/hls_component2/hls_component2$ tree run_test_input
run_test_input
├── component.xml
├── constraints
│   └── run_test_input_ooc.xdc
├── doc
│   └── ReleaseNotes.txt
├── hdl
│   ├── verilog
│   │   ├── run_test_input_Cipher_temp_new_buf_state1_RAM_AUTO_1R1W.v
│   │   ├── run_test_input_Cipher.v
│   │   ├── run_test_input_hexinput_to_array.v
│   │   ├── run_test_input_mul_128s_8ns_128_5_1.v
│   │   ├── run_test_input_Rcon_ROM_AUTO_1R.dat
│   │   ├── run_test_input_Rcon_ROM_AUTO_1R.v
│   │   ├── run_test_input_RoundKey_RAM_AUTO_1R1W.v
│   │   ├── run_test_input_Round_temp_new_buf_state_round_1_RAM_AUTO_1R1W.v
│   │   ├── run_test_input_Round_temp_new_buf_state_round_2_RAM_AUTO_1R1W.v
│   │   ├── run_test_input_Round_temp_new_buf_state_round_3_RAM_AUTO_1R1W.v
│   │   ├── run_test_input_Round.v
│   │   ├── run_test_input_sparsemux_513_8_8_1_1.v
│   │   └── run_test_input.v
│   └── vhdl
│       ├── run_test_input_Cipher_temp_new_buf_state1_RAM_AUTO_1R1W.vhd
│       ├── run_test_input_Cipher.vhd
│       ├── run_test_input_hexinput_to_array.vhd
│       ├── run_test_input_mul_128s_8ns_128_5_1.vhd
│       ├── run_test_input_Rcon_ROM_AUTO_1R.vhd
│       ├── run_test_input_RoundKey_RAM_AUTO_1R1W.vhd
│       ├── run_test_input_Round_temp_new_buf_state_round_1_RAM_AUTO_1R1W.vhd
│       ├── run_test_input_Round_temp_new_buf_state_round_2_RAM_AUTO_1R1W.vhd
│       ├── run_test_input_Round_temp_new_buf_state_round_3_RAM_AUTO_1R1W.vhd
│       ├── run_test_input_Round.vhd
│       ├── run_test_input_sparsemux_513_8_8_1_1.vhd
│       └── run_test_input.vhd
├── misc
│   └── logo.png
└── xgui
    └── run_test_input_v1_0.tcl

7 directories, 30 files
```

Fig. 19: Structure of the Generated IP

5.12 Step 9: Generate a Bitstream and Deploy the Bitstream on FPGA

Once we generate the IP successfully, we can move on to the next step. Specifically, we will first need to start Vivado by following the commend below in the terminal. A successful Vivado run can be found in 20

```
$ source /tools/ Xilinx/Vitis/2023.2/settings64.sh
$ vivado # if one chooses to run Vivado
```

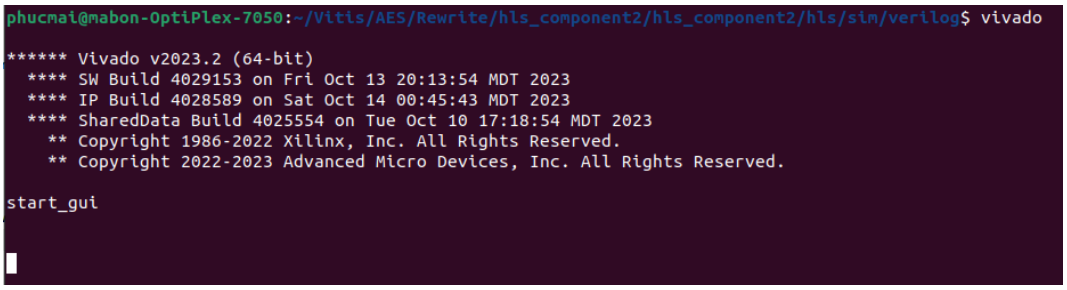


Fig. 20: Run Vivado from the terminal to launch Vivado GUI

Create a New Project in Vivado. After we launch Vivado, we create a new project by selecting "Create Project" under "Quick Start" section on the welcome screen. Next, we will need to provide the name of this project (we name it `AES_ECB_128` in our example) and its location. We then need to specify the type of the project ("RTL project" in our case); make sure the "Do not specify sources at this time" box is checked. Finally, we need to specify the target board (Arty A7-100 in our case). This completes the step of creating the new Vivado project. Figures of the above steps are presented from Fig. 21 to Fig. 25.

Add IP to Vivado Project. Next, we provide a top module name and add the our IP generated from the last step to the Vivado project IP repository. Specifically, we go to "Settings" under "PROJECT MANAGER" and navigate into "Repository" tab under "IP" drop down menu. We then select "+" under "IP Repositories" and select the path to folder `run_test_input`, which is extracted from `run_test_input.zip` obtained from the last step. Figure of this action in presented in Fig. 26.

Fig. 21: Welcome Screen of Vivado

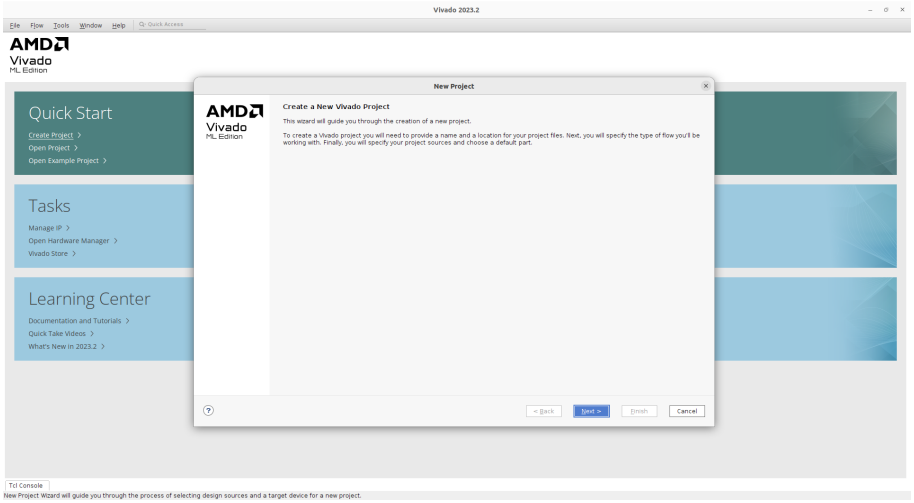


Fig. 22: Start of instruction guide

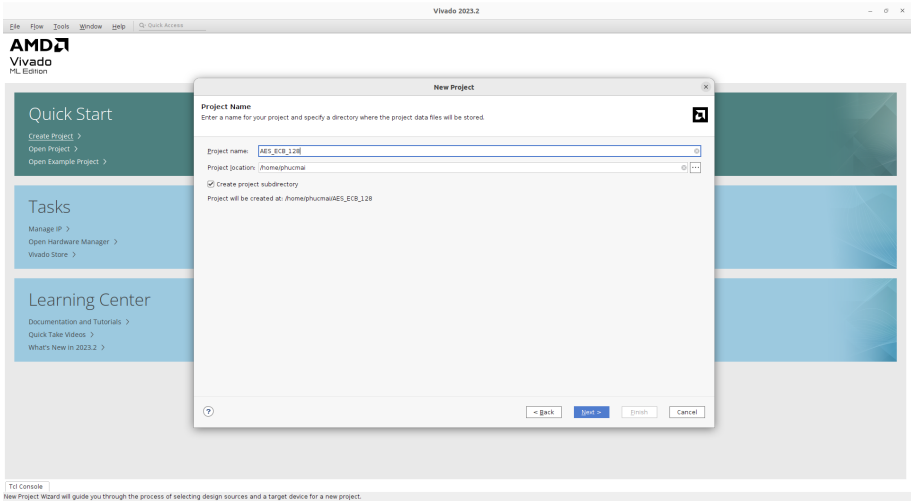


Fig. 23: Provide a project name and location

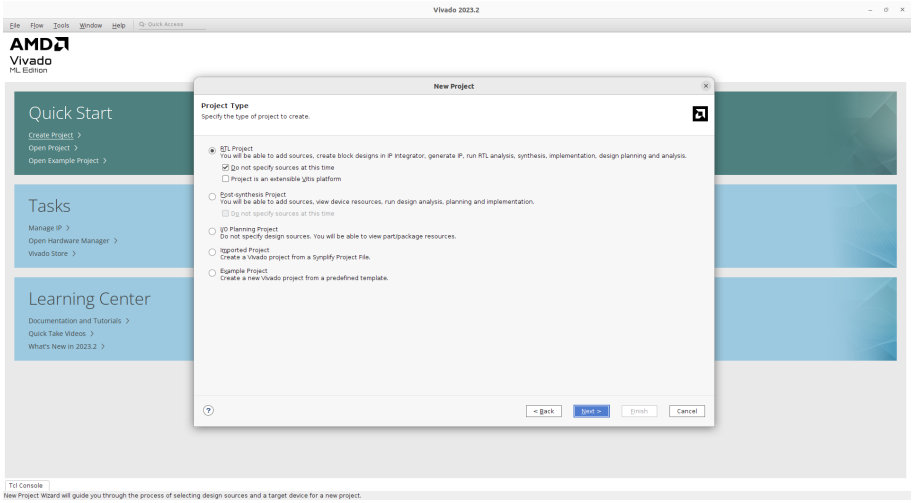


Fig. 24: Specify project type

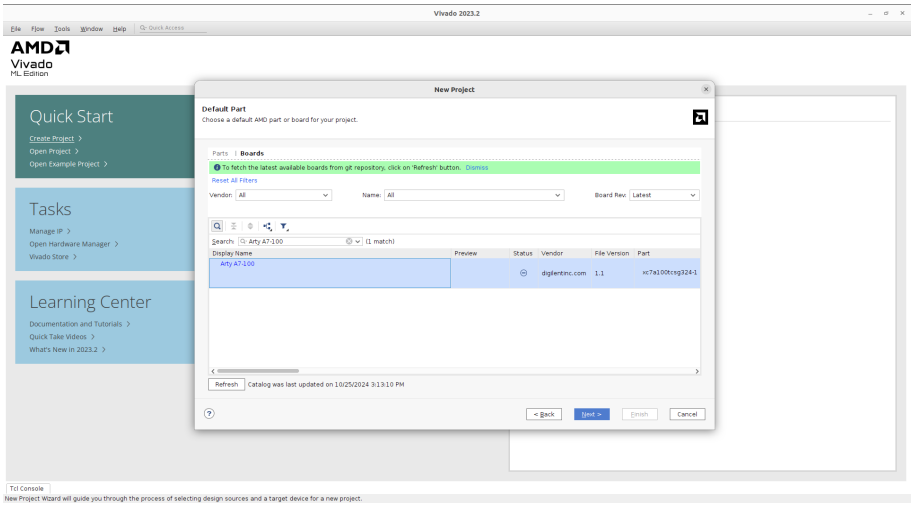


Fig. 25: Specify target board part (choose `xc7a100tcsg324-1`)

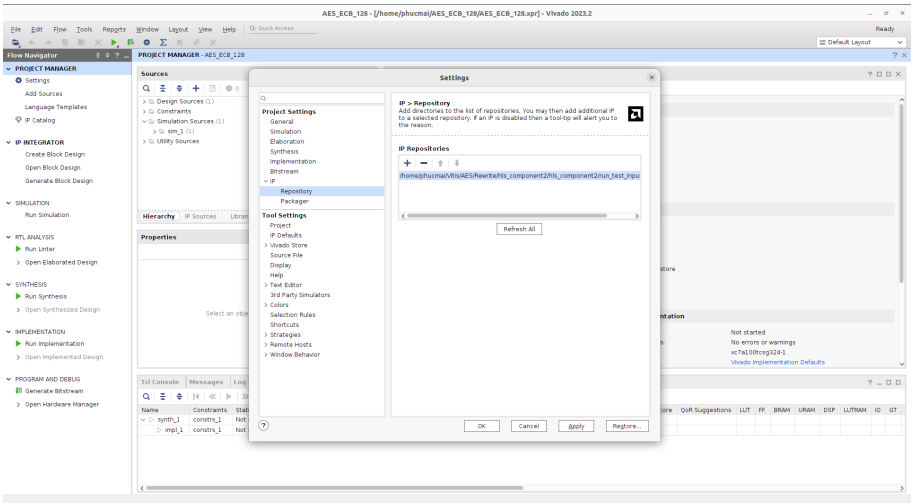


Fig. 26: Adding our generated IP to Vivado project's IP repository

Next, we will discuss how to create a block design and synthesize it into a FPGA bitstream. We will also show how to integrate hardware verification component into our design as well.

Create Block Design. First step is to create a new block design for our project. Specifically, we select "Create Block Design" under "IP INTEGRATOR". We then provide the name (`design_1` in our example), location, and source files of the design, which we left as default for now. Fig. 27 shows the default configurations for a new block design.

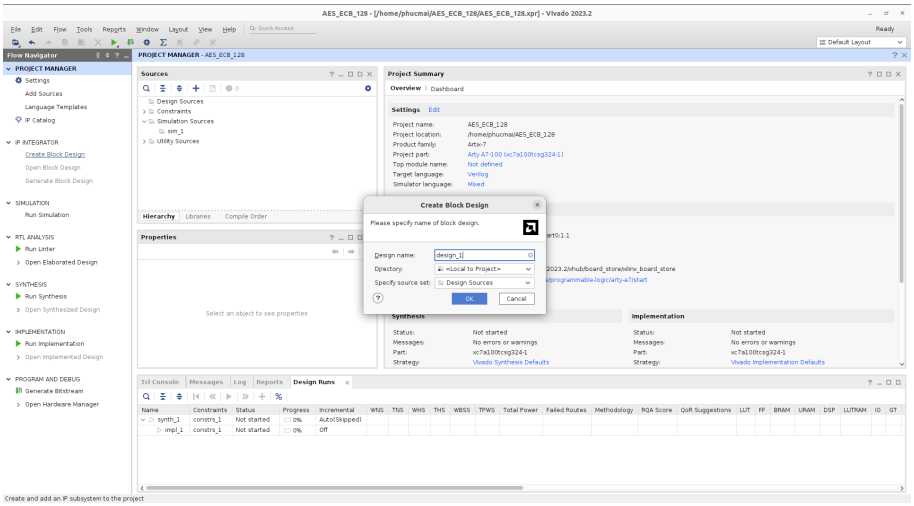


Fig. 27: Create a new block design for `AES_ECB_128` in Vivado

Next, we select the "+" button in "Diagram" window, search for "Run_test_input" and press "Enter" to add `run_test_input` IP into our block design (`design_1`). Fig. 28 and Fig. 29 show the process of adding IP component to `design_1` and the appearance of `design_1` after `run_test_input` IP has been added.

Next, we select "Run Connection Automation" on the green ribbon located at the top of "Diagram" window. A window, presented in Fig. 30, shows the options that the automation tool will follow to create support blocks and wires for our design. We can leave everything as default for now. After we click "OK", the appearance of `design_1` at this point can be found in Fig. 31.

As shown in Fig. 31, Vivado still suggests "Run Connection Automation", indicating some components and/or connections are needed for `design_1` to be a valid design. Therefore, we run "Run Connection Automation" again for **the second time**. This time we check all

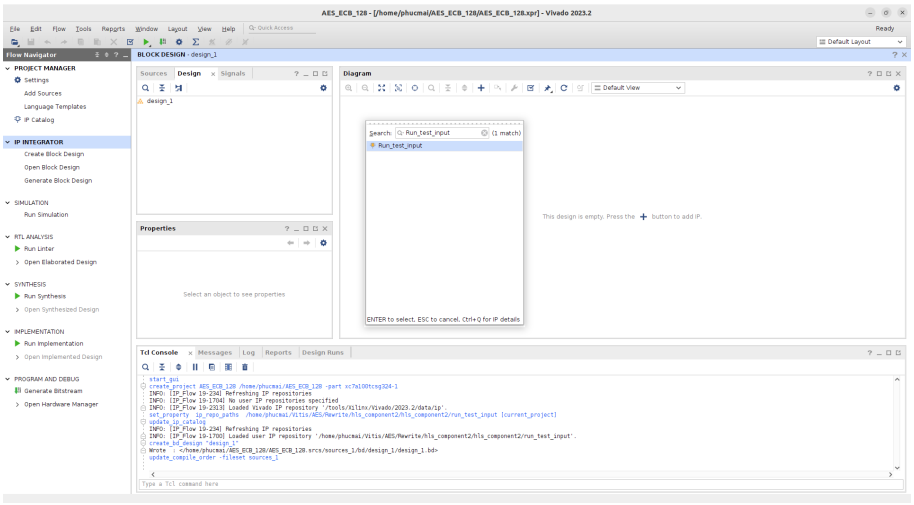


Fig. 28: Add `run_test_input` IP to `design_1` block design

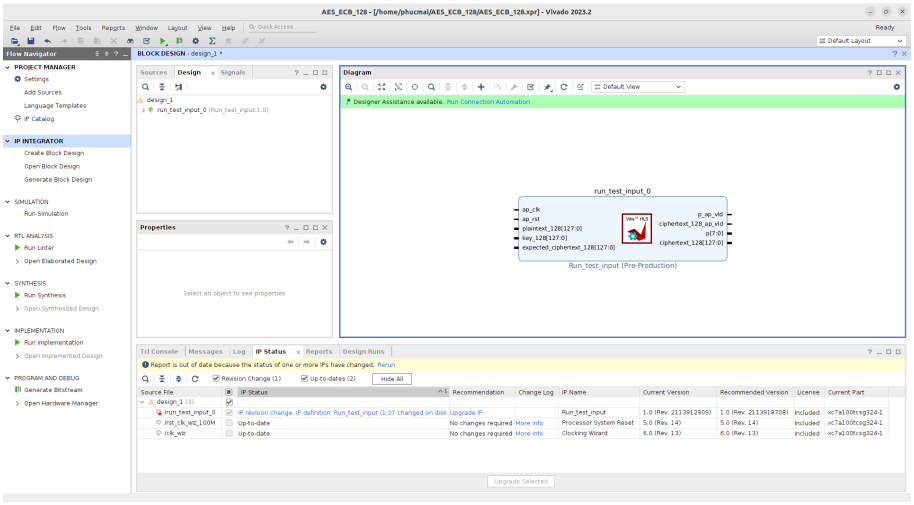


Fig. 29: Appearance of `design_1` after `run_test_input` IP has been added

the boxes (e.g., `clk_wiz` and `rst_clk_wiz_100M` as shown in Fig. 32). The appearance of `design_1` at this point can be found in Fig. 33.

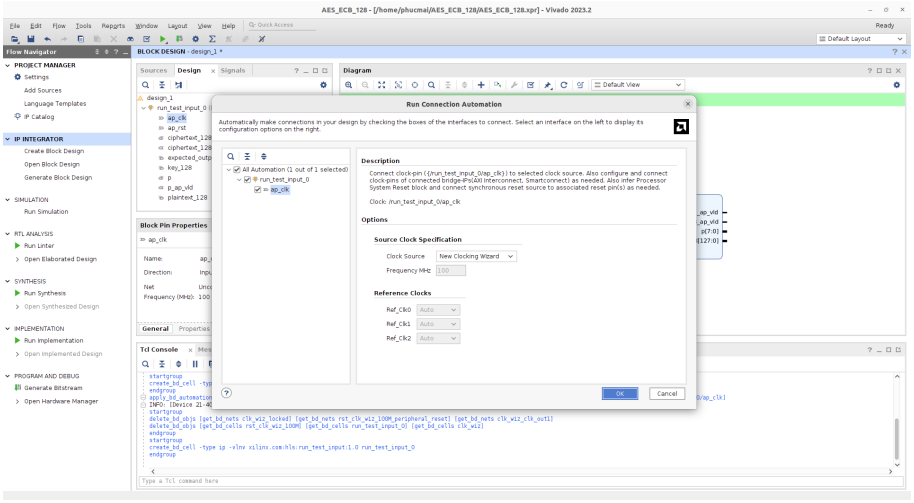


Fig. 30: Run Connection Automation the first time

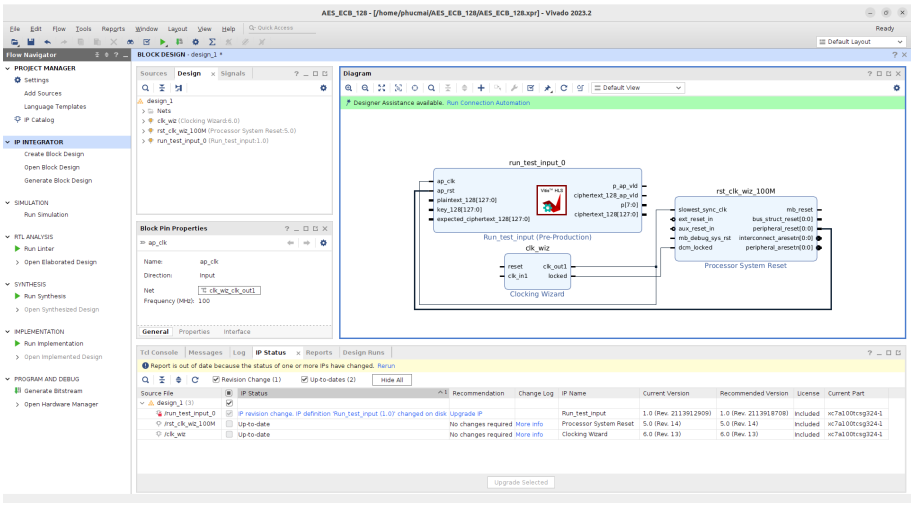


Fig. 31: `design_1` block design after running "Run Connection Automation" the first time

Next, we create additional components to supply plaintext, key, and expected_ciphertext into `run_test_input` IP. To do this, we can add one Constant IP component into `design_1` for plaintext, key, and expected_ciphertext, respectively. Specifically, we navigate to "+" in "Design" window, search for "Constant" and press "Enter"; one Constant IP, which can be found in Fig. 34, will then be added to `design_1` block design. We configure this Constant IP to be 128 bit in size to hold the value of our plaintext. To do this, we double click onto

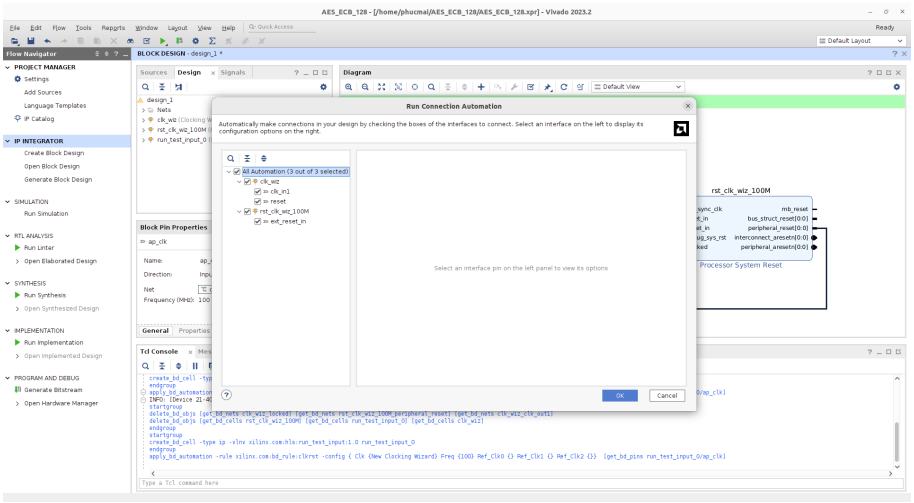


Fig. 32: Run Connection Automation the second time

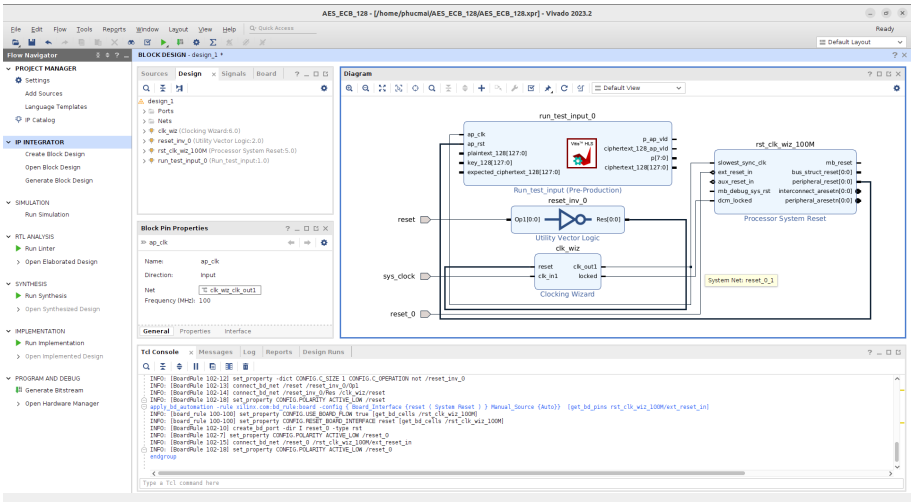


Fig. 33: `design_1` block design after running "Run Connection Automation" the second time

the Constant IP component; a "Re-customize IP" window will then be launched as shown in Fig. 35. We change "Const Width" into 128 and "Const Val" into the input value (we provide `0x6bc1bee22e409f96e93d7e117393172a` to "Const Val" as the plaintext, which was used in our C simulation and C/RTL simulation). An example Constant IP block for plaintext input after being configured is presented in Fig. 34 and Fig. 35 as a reference. Similarly, we add one Constant IP block for key (`0x2b7e151628aed2a6abf7158809cf4f3c`) and one for expected_ciphertext (`0x3ad77bb40d7a3660a89ecaf32466ef97`).

(a) Original Constant IP block

(b) Configured Constant IP block

Fig. 34: Appearance of Constant IP block

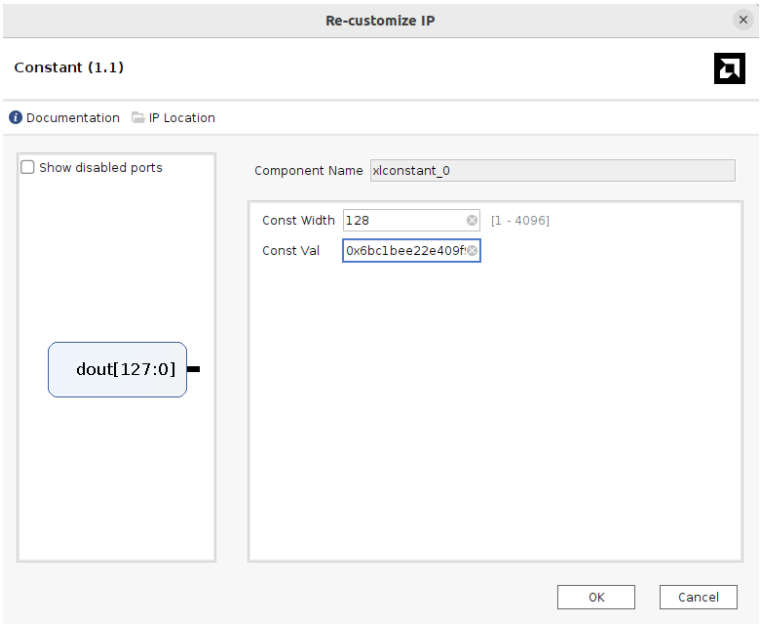


Fig. 35: Detail configuration of Constant IP block for plaintext input

Next step is to connect the three Constant IP blocks to `run_test_input`. Specifically, we click on the pin of each Constant IP block and drag it over to the corresponding input pin on `run_test_input` IP block. The final complete appearance of `design_1` block design

is presented in Fig. 36. After this action, we have successfully created a block design, which can be synthesized into a bitstream and run on FPGA.

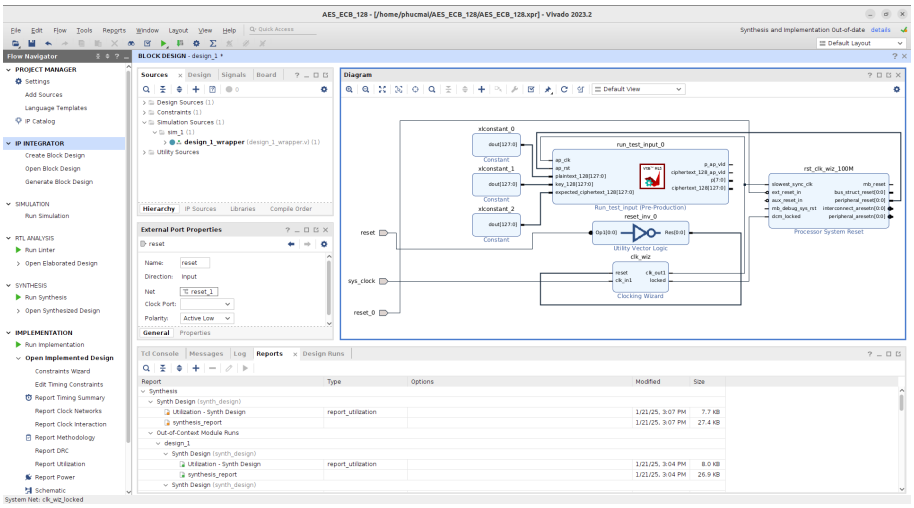


Fig. 36: Final complete appearance of `design_1` block design

Add Hardware Verification Component (Recommended but Optional). Vivado provides Integrated Logic Analyzer (ILA) IP block, which is able to capture signal for a given port during the run time on FPGA. With this, we can show whether the synthesized bitstream runs correctly on a FPGA board. Note that this step is optional but recommended for design verification.

To add ILA to our block design, we select "+" in "Diagram" window, search for "ILA", and press "Enter". An ILA IP block with default configuration, as shown in Fig. 37, is then added to our block design. The default configuration for ILA block is presented in Fig. 38 as a reference.

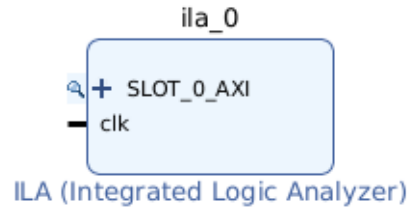


Fig. 37: Integrated Logic Analyzer (ILA) IP block

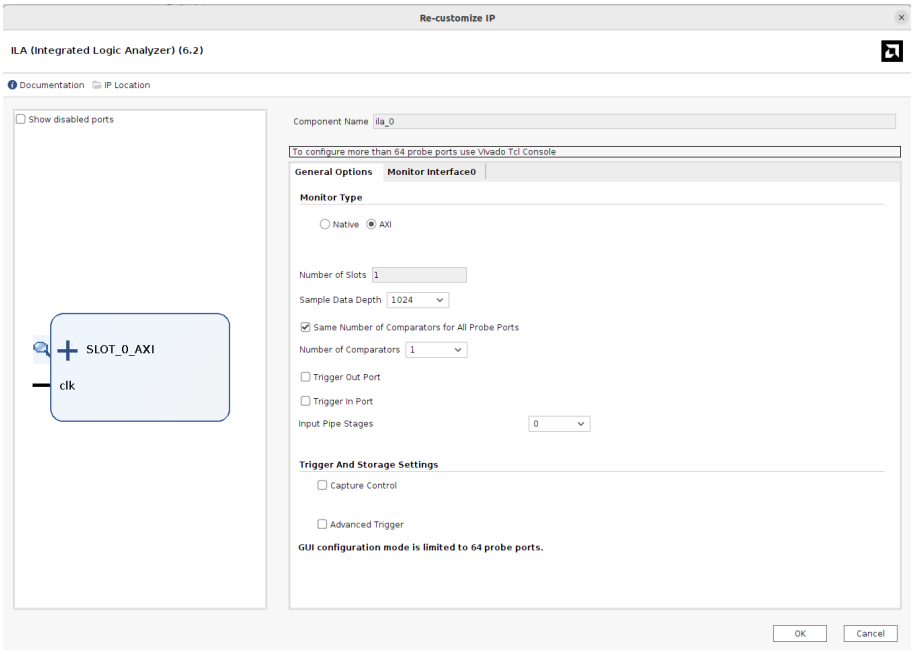
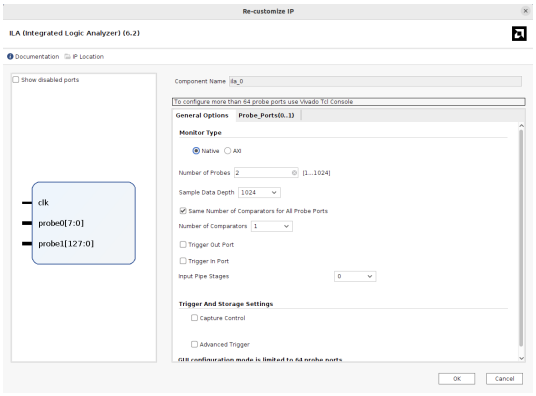
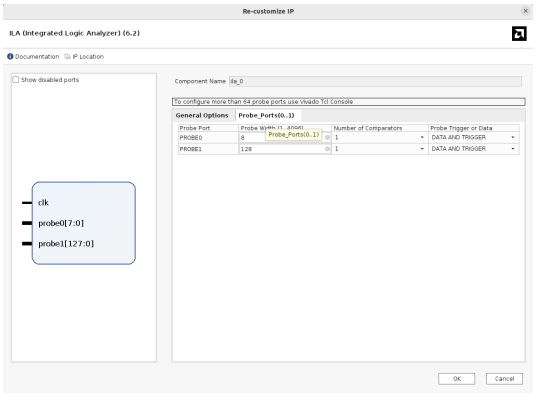


Fig. 38: Integrated Logic Analyzer (ILA)'s default configuration

We need to reconfigure ILA block to support our purpose. Specifically, as shown in Fig. 38, in "General Options" tab, we change "Monitor Type" into "Native", then set "Number of Probes" to 2. Then, in the "Probe_Ports" tab, we change "Probe Width" for "PROBE0" and "PROBE1" to be 8 and 128 respectively. The final configuration and appearance of configured ILA block is presented in Fig. 39 and Fig. 40 respectively.



(a) ILA's General Options configuration



(b) ILA's Probe Ports configuration

Fig. 39: ILA's configuration

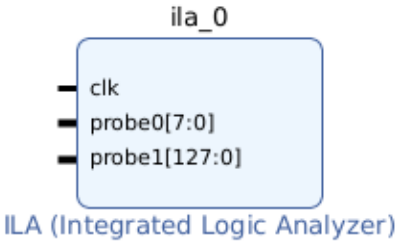


Fig. 40: Appearance of configured ILA block

Next, we wire `ila_0`'s `clk` pin to `clk_wiz`'s `clk_out1` pin. We then wire `ila_0`'s `probe0` and `probe1` pin to `run_test_input`'s `p` and `ciphertext_128` pin respectively. The final block design with ILA can be found in Fig. 41

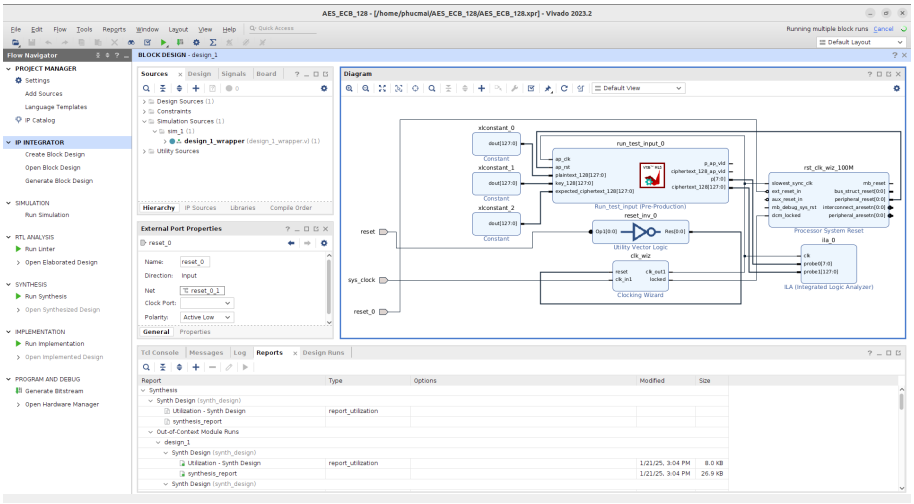


Fig. 41: Final block design with ILA integrated

As a preparation for generating bitstream, we need to create a HDL Wrapper for our block design. To do this, we right click on "design_1 (design_1.bd)" under "Simulation Sources/sim_1" in "Sources" window Fig. 42 and select "Create HDL Wrapper"; make sure "Let Vivado manage wrapper and auto-update" option is selected. The final HDL wrapper can be found in Fig. 43. Accordingly, we need to update "Settings/General/Top module name" to `design_1_wrapper` as seen in Fig. 44.

Synthesize the Block Design into a FPGA bitstream. To generate a bitstream for FPGA, we run "Synthesis", "Implementation", and "Generate Bitstream". Specifically, in

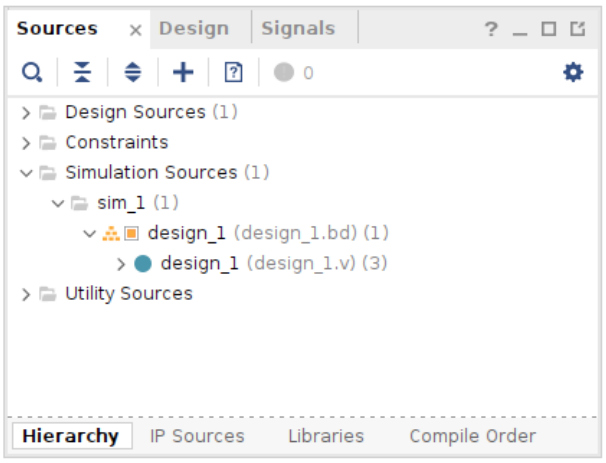


Fig. 42: Block design sources tab

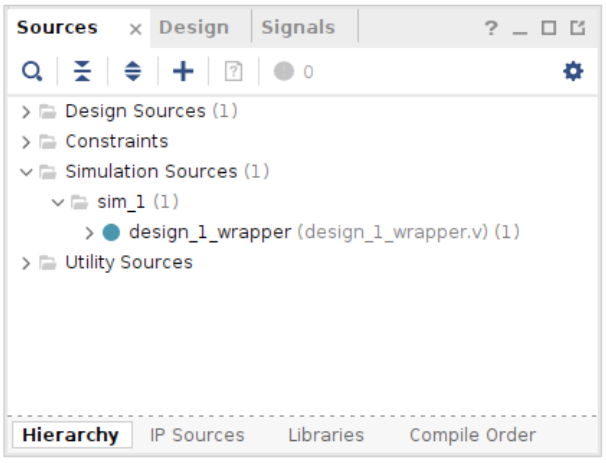


Fig. 43: Block design with HDL wrapper

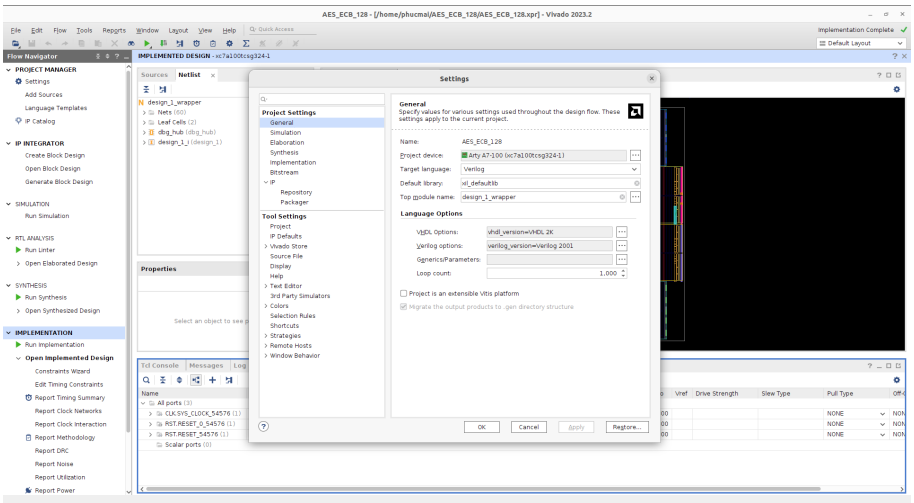


Fig. 44: Block design top module name updated to `design_1_wrapper`

"Flow Navigator" tab, we click on "Run Synthesis" under "SYNTHESIS" tab, then proceed to select "Run Implementation" under "IMPLEMENTATION" tab, and finally proceed to run "Generate Bitstream" under "PROGRAM AND DEBUG". A successful run of each step is indicated by the green tick symbol next to "design_1", "synth_1", and "impl_1" in "Design Runs" window as shown in Fig. 45

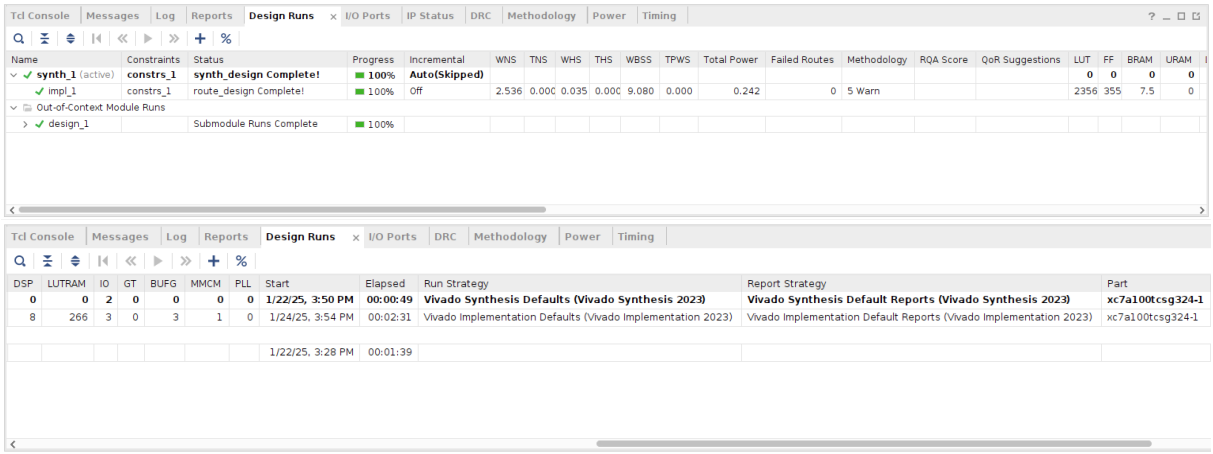


Fig. 45: Successful run as shown in "Design Runs" window

After running "Implementation", we need to update I/O ports configuration. Specifically, we navigate to "Window" tab and select "I/O Ports"; an example of default I/O ports configuration is presented below as a reference. We make sure the "I/O Std" field is LVCMOS33 (3.3V) according to the details of our FPGA board [5]. In addition, we update the RST.RESET_0_54576 port as fixed. The updated I/O ports configuration can be found in Fig. 47. We then press "Ctrl + S" on "I/O Ports" tab to save I/O ports configuration as a constraint file for our design.

After updating the I/O Ports, we re-run "Synthesis", "Implementation", and "Generate Bitstreams". The final schemetic post-synthesis is shown in Fig. 48 and the device layout post-implementation of our design is presented in Fig. 49. A successful "Generate Bitstream" run can be found in 50

At this point, we have successfully generated a bitstream from our design for Digilent Arty A7-100t FPGA board. We can locate it at `<path to Vivado project>/<project name>.runs/<top module name>.bit`. Next step is to upload the bitstream to the FPGA.

[5] `https://github.com/Digilent/digilent-xdc/blob/master/Arty-A7-100-Master.xdc`

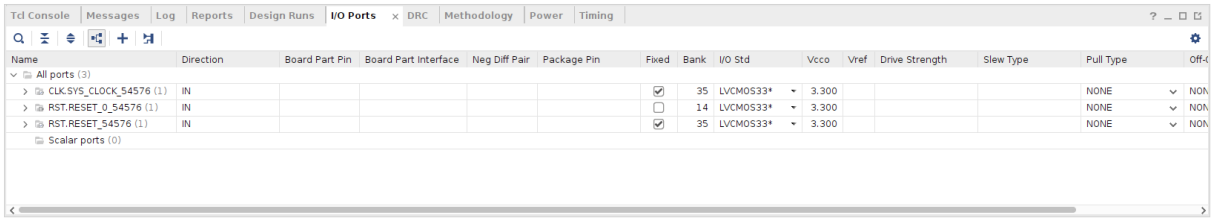


Fig. 46: Block design default I/O ports configuration

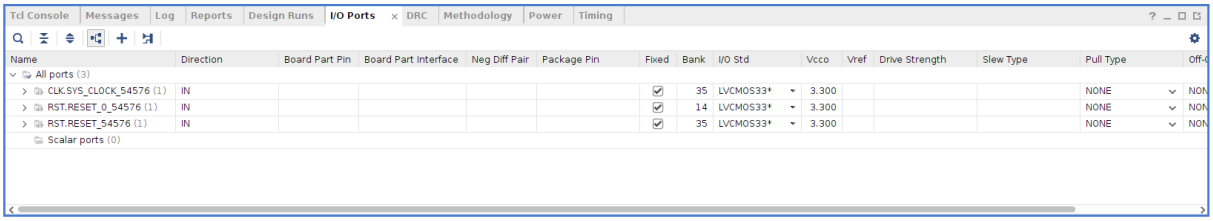


Fig. 47: Block design final I/O ports configuration

Deploy a Bitstream on FPGA. To deploy the bitstream (`<top module name>.bit`) obtained from above step on Digilient Arty A7-100t FPGA board, we first need to connect the board to our machine and select "Open Hardware Manager" under "PROGRAM AND DEBUG" to open Vivado's hardware manager as shown in Fig. 51.

Next step is to click on "Open target" then "Auto connect" to connect the target FPGA board to Vivado's hardware manager. Once the board is connected, we need to click on "Program device" to program the board with the bitstream obtained previously.

If we add the ILA component in the bitstream and would like to capture signal from the board when hardware is running, we can click on the "Play" symbol; then any signal captured by ILA is displayed on the screen, which in our case is `p` (pass flag) and `ciphertext_128` (output ciphertext). A successful run is presented below as a reference.

If one can reach this point, it means that they have successfully generated a working bitstream running AES ECB for Digilient Arty A7-100t FPGA board!

5.13 Step 10: Obtain the Final Design

At this point, we have successfully translated an AES ECB 128 software implementation (based on TinyAES [6]) into a hardware version using Vitis HLS and Vivado for Digilient Arty A7-100t FPGA board. As discussed above, the design is proved to operate successfully and produce correct ciphertext when running on physical FPGA board. While this is successful, the current hardware design contains many verification components, which are not needed

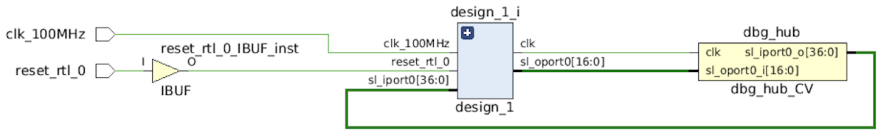


Fig. 48: Block design I/O ports configuration

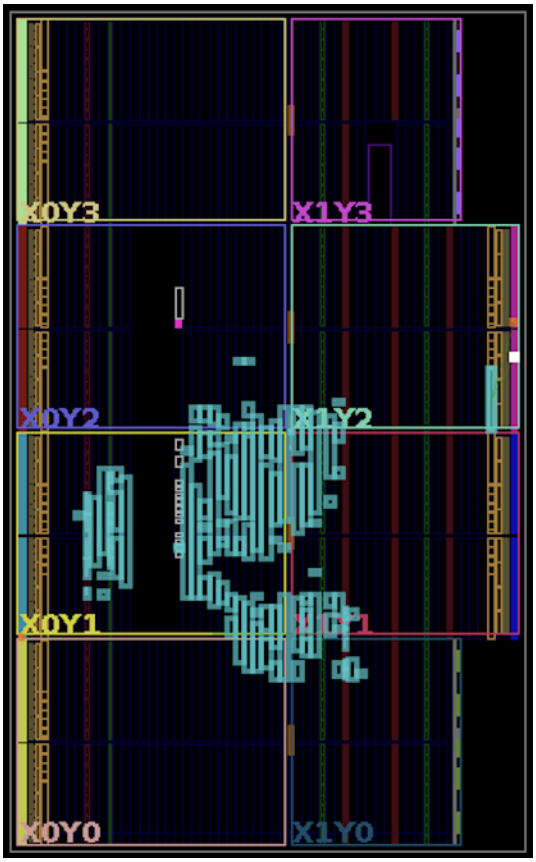


Fig. 49: Block design device layout

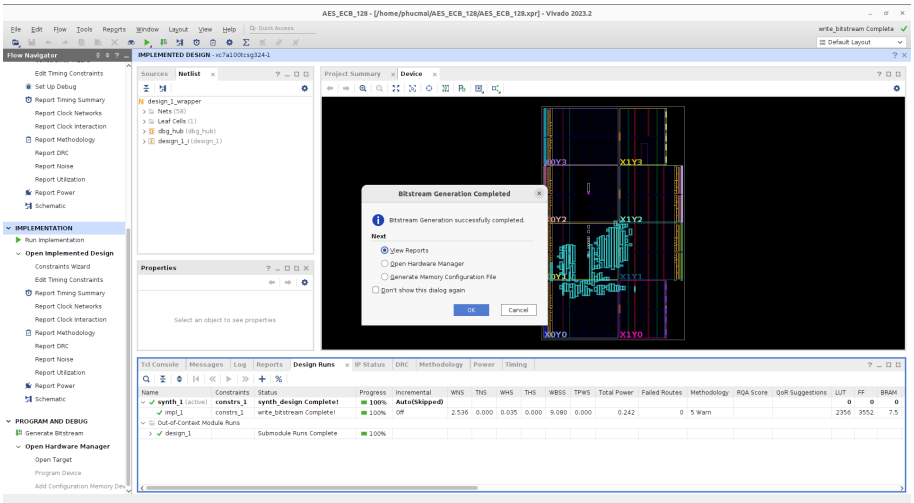


Fig. 50: An Example of Successful “Generate Bitstream” Run

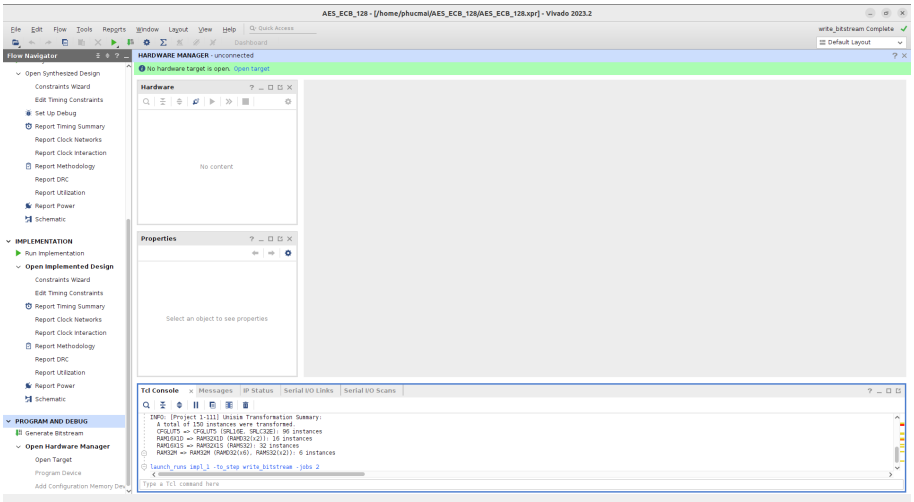


Fig. 51: Vivado’s hardware manager

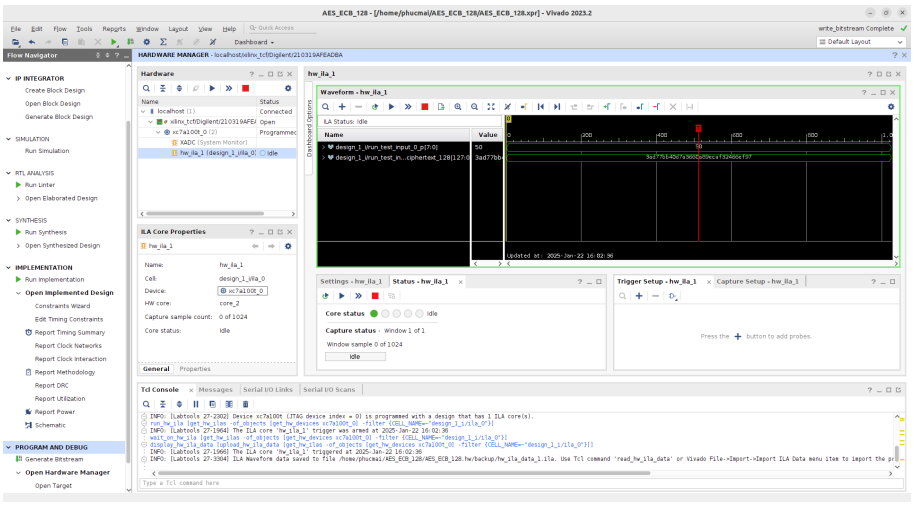


Fig. 52: On-device verification result, which shows the encryption runs correctly.

Table 1: Results of the prototype and utilization of the current hardware design (with ILA and verification component, referred as verifiable design) on Digilient Arty A7-100t (`xc7a100tcsg324-1`) and Spartan-7 SP701 Evaluation Platform (`xc7s100fgga676-2`)BOYANG SAYS: Phuc: please update the table

	Verifiable Design (Arty A7)	Verifiable Design (Spartan-7)
Power (W)	0.243	??
LUTs (util.)	2354 (14.85%)	??
Flip-flops (util.)	3552 (2.80%)	??
BRAM Blocks (util.)	7.5 (3.33%)	??
DSP slices (util.)	8 (3.33%)	??
Frequency (MHz)	100	??
Latency (μs)	25.66	??

for the final design. Ideally, the final design should take one key and one plaintext as input, and output a ciphertext. To obtain the final design, we will need to remove all the verification components in our current design (e.g., ILA block, `p` flag, `expected_ciphertext`, and the comparison mechanism between calculated ciphertext and expected ciphertext)[6].

Update Top Function for Final Design. First, we will need to go back to our HLS component in Vitis and make changes in our top function `run_test_input`. Specifically, in the top function `run_test_input`, we remove argument `expected_ciphertext` and `p` and their corresponding port configuration (line 7 and 8 respectively). The updated version of `run_test_input`'s function definition and port configurations can be found below

[6] Although obtaining the final design (without verification components) directly is feasible, we highly recommend to obtain the version with verification components first to ensure the correctness of the design as what we did in this tutorial.

```
// updated version of top function in test.cpp for the final design
......
void run_test_input(ap_uint<128> *ciphertext_128, ap_uint<128> plaintext_128,
    ap_uint<128> key_128){
#pragma HLS INTERFACE mode=ap_ovld port=plaintext_128
#pragma HLS INTERFACE mode=ap_ovld port=ciphertext_128
#pragma HLS INTERFACE mode=ap_ovld port=key_128
#pragma HLS INTERFACE mode=ap_ctrl_none port=return

#pragma HLS array_partition variable=sbox type=complete
#pragma HLS array_partition variable=rsbox type=complete
......
}
```

In addition, we remove the ciphertext result verification component of the program, which is associated with line 27, line 29 to 35, and line 50 in `run_test_input` function. We highlight these lines below as a reference.

```
// updated version of top function in test.cpp for the final design
// remove line 27 below
// int pass = 0x50;

// remove line 29 to 35 below
// for (int i = 0; i < 16; i ++){
//    #pragma HLS pipeline off
//    if (ciphertext[i] != expected_ciphertext[i]){
//        pass = 0x10;
//        break;
//    }
// }
......
// remove line 50 below
//*p = pass;
......
```

The updated version of `run_test_input` for the final design can be found below

```
// updated version of top function in test.cpp for the final design
void run_test_input(ap_uint<128> *ciphertext_128, ap_uint<128> plaintext_128,
    ap_uint<128> key_128){

uint8_t key[16], uint8_t expected_output[16]){
#pragma HLS INTERFACE mode=ap_ovld port=plaintext_128
#pragma HLS INTERFACE mode=ap_ovld port=ciphertext_128
#pragma HLS INTERFACE mode=ap_ovld port=key_128
#pragma HLS INTERFACE mode=ap_ctrl_none port=return

```

```
#pragma HLS array_partition variable=sbox type=complete
#pragma HLS array_partition variable=rsbox type=complete

    uint8_t RoundKey[AES_keyExpSize];

    uint8_t ciphertext[16];
    uint8_t plaintext[16];
    uint8_t expected_ciphertext[16];
    uint8_t key[16];

    take_input(plaintext_128, plaintext);
    take_input(key_128, key);
    take_input(expected_ciphertext_128, expected_ciphertext);

    test_encrypt_ecb(key, plaintext, ciphertext, RoundKey);

    ap_uint<128> ciphertext_temp = 0;

    for (int i = 0; i < 16; i++){
        #pragma HLS pipeline off
        ap_uint<128> power_16 = 1;
        for (int j = 0; j < (15 - i) * 2; j++){
            #pragma HLS pipeline off
            power_16 *= 16;
        }
        ciphertext_temp += power_16*ciphertext[i];
    }

    *ciphertext_128 = ciphertext_temp;
}
```

Obtain the Updated IP. With the updated version of the top function, we go through the same process as in Step 6 and Step 8 (Step 7 can be skipped for the final design) to obtain the updated IP. Specifically, we run "C Synthesis" and "Package" in Vitis HLS.

Obtain the Updated Block Design. Once we obtain the updated IP, we follow the guidelines as detailed in Step 9 to create an updated block design in Vivado, run synthesis and implementation, and then generate the bitstream. Specifically, instead of adding the ILA component (for verification) and Constant block (for supplying plaintext) as we did before, we expose the corresponding ports (`plaintext_128`, `key_128`, and `ciphertext_128`) such that I/O ports can be used to pass inputs to the design and record outputs from the design. To do this, we right click on the target port pin, for example `plaintext_128`, and

Fig. 53: Exposing port `plaintext_128`

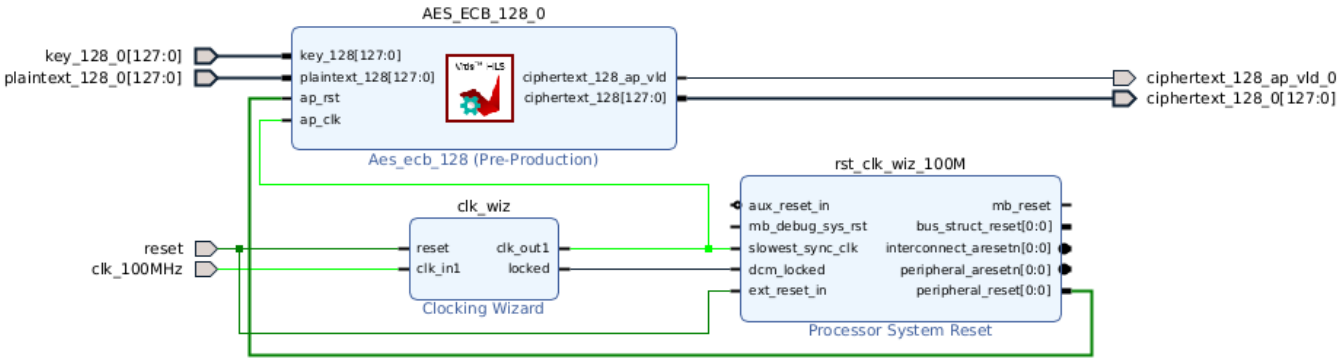


Fig. 54: Block design of final design

select make external. Fig. 53 shows the result of this operation. We apply the same method to `key_128` and `ciphertext_128`. Fig. 54 shows the updated block design in Vivado.

It's worth mentioning that exposed ports are actually mapped into I/O ports of the board. As one I/O port can only carry 1 bit (0 or 1), it requires 128 I/O ports to handle each of `plaintext_128`, `ciphertext_128` and `key_128` respectively, making the total of 384 I/O ports required for the final design, which exceeds the number of I/O ports available on Digilient Arty A7-100t FPGA board. In order to streamline further analysis on the model, we switch the target board to Spartan-7 SP701 Evaluation Platform (`xc7s100fgga676-2`). The measurement and data we present from this point is based on Spartan-7 SP701 Evaluation Platform. The Netlist schematice and device layout of the final design can be found in 55, 56, and 57. Table 2 shows the utilization of final design.

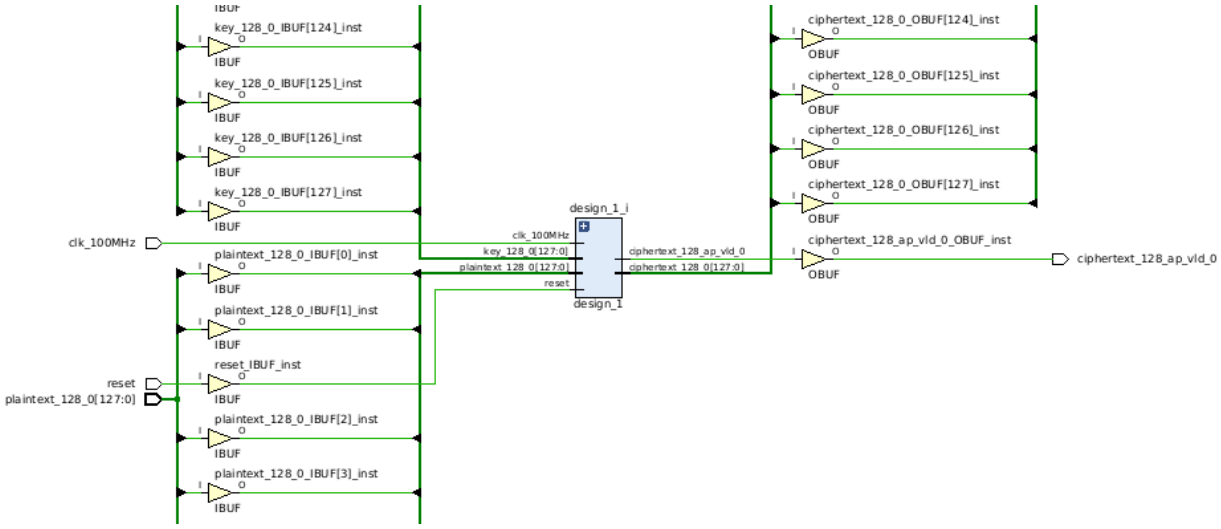


Fig. 55: Zoomed in the netlist schematic (final design)

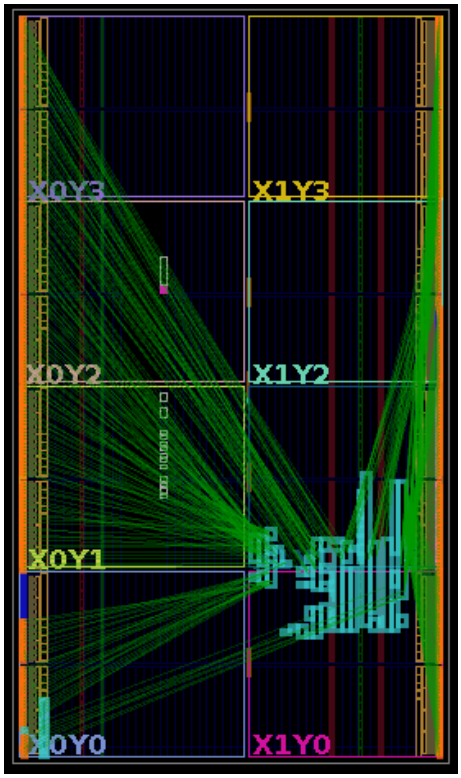


Fig. 56: Design device layout (final design)

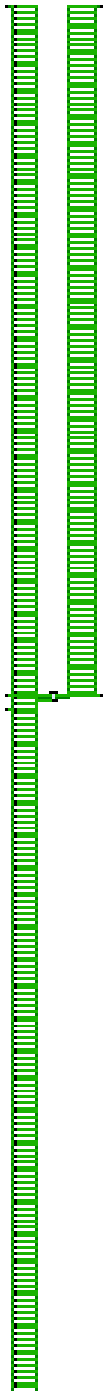


Fig. 57: Overall netlist schematic (Final design)

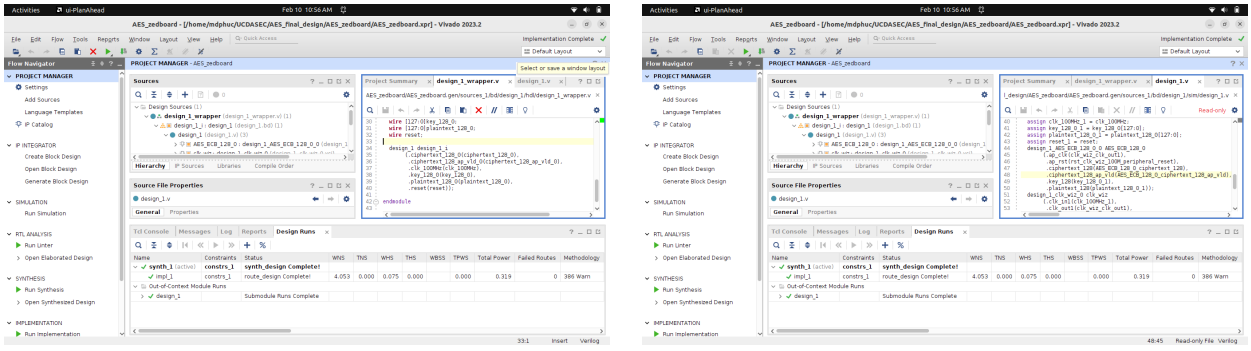
Table 2: Results of the prototype and utilization of the verifiable design and final design on Spartan-7 SP701 Evaluation Platform (`xc7s100fgga676-2`)

	Verifiable Design	Final Design
Power (W)	??	0.319
LUTs (util.)	??	975 (1.52%)
Flip-flops (util.)	??	1180 (0.92%)
BRAM Blocks (util.)	??	3(2.50%)
DSP slices (util.)	??	8 (5%)
Frequency (MHz)	??	100
Latency (μs)	??	21.93

5.14 Step 11: Generate a Simulated Trace of Final Design in Vivado

Given the final design, our next step is to obtain simulated traces of this design for side-channel analysis. Specifically, we run the simulation in Vivado with the final design by providing an additional testbench. This testbench passes a key and a plaintext as input to the final design, and the final design outputs the ciphertext. This is another way to obtain simulated traces in addition to the ones mentioned through C/RTL co-simulation.

For ease of presentation, we first provide more detailed information regarding our block design. The block design we have in Vivado (`design_1`) is wrapped inside a Verilog file (`design_1_wrapper.v`). `design_1_wrapper.v` contains calls to the control module of the block design (`design_1.v`). `design_1.v` then calls main modules of the program to run AES encryption. The hierarchy can be found in 58



(a) HDL wrapper for our block design

(b) Main control module for our block design

Fig. 58: HDL-wrapped Vivado block design hierarchy

The content of `design_1_wrapper` is presented below as a reference.

```
`timescale 1 ps / 1 ps

module design_1_wrapper
    (ciphertext_128_0,
    ciphertext_128_ap_vld_0,
    clk_100MHz,
    key_128_0,
    plaintext_128_0,
    reset);
    output [127:0]ciphertext_128_0;
    output ciphertext_128_ap_vld_0;
    input clk_100MHz;
    input [127:0]key_128_0;
    input [127:0]plaintext_128_0;
    input reset;

    wire [127:0]ciphertext_128_0;
    wire ciphertext_128_ap_vld_0;
    wire clk_100MHz;
    wire [127:0]key_128_0;
    wire [127:0]plaintext_128_0;
    wire reset;

    design_1 design_1_i
        (.ciphertext_128_0(ciphertext_128_0),
        .ciphertext_128_ap_vld_0(ciphertext_128_ap_vld_0),
        .clk_100MHz(clk_100MHz),
        .key_128_0(key_128_0),
        .plaintext_128_0(plaintext_128_0),
        .reset(reset));

    endmodule
```

We create a testbench (`testbench.v`), which contains a `design_1_wrapper` **module initiation**, **clock generation**, and a **stimuli** (to update signals to reflect on `design_1_wrapper` module). This testbench provides a key and a plaintext, and also specific the name of the VCD file that will be generated from the simulation. The code of the testbench we used is listed below

```
`timescale 1ns/1ps

module main_tb();
    reg ap_clk;
    reg ap_rst;
    wire [127:0] ciphertext_128;
```

```
    wire ciphertext_128_ap_vld;
    reg [127:0] plaintext_128;
    reg [127:0] key_128;

    design_1_wrapper dut(
        .clk_100MHz(ap_clk),
        .reset(ap_rst),
        .ciphertext_128_0(ciphertext_128),
        .ciphertext_128_ap_vld_0(ciphertext_128_ap_vld),
        .plaintext_128_0(plaintext_128),
        .key_128_0(key_128)
    );

    always #5 ap_clk = ~ap_clk;
    initial begin
        ap_clk = 0;
        ap_rst = 1;
        plaintext_128 = 128'h0;
        key_128 = 128'h0;
        #20
        ap_rst = 0;
        #10
        plaintext_128 = 128'h6bc1bee22e409f96e93d7e117393172a;
        key_128 = 128'h2b7e151628aed2a6abf7158809cf4f3c;
        wait(ciphertext_128_ap_vld);
        #50 $finish;
    end

    initial begin
        $dumpfile("/home/mdphuc/UCDASEC/AES_final_design/aes_trace_behavioral.vcd")
        ;
        $dumpvars(0, dut); // dump all signals inside the program
    end

    initial begin
        $monitor("Time = %0t, ciphertext = %h", $time, ciphertext_128);
    end
endmodule
```

Usage. To run the simulation inside Vivado, first, we need to add our testbench program to the "Simulation Sources". To do this, we can navigate to "Simulation Sources" under "Sources" window, right click on "sim_1" and select "add simulation file" to add our testbench file to the Vivado project. The final look of "Sources" window can be found in 59

To run simulation, we can click on "Run Simulation" under "Simulation" tab in "Project Manager" section. The supported options for simulation are referenced in 60. We primarily

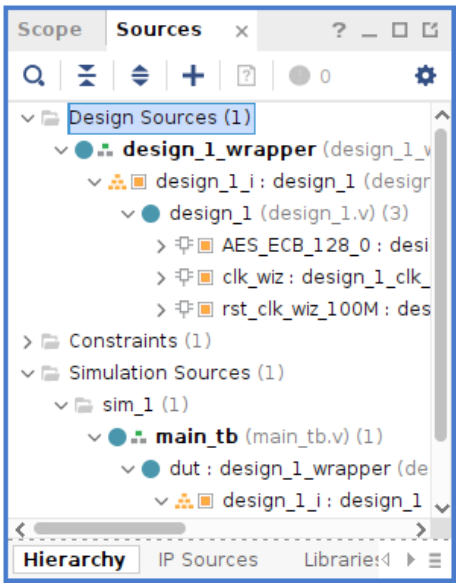


Fig. 59: "Sources" window after testbench file `main_tb.v` has been added

run Behavioral Simulation in this tutorial. Other simulations, such as Post-Synthesis and Post-Implementation, can also be performed with longer simulation time.

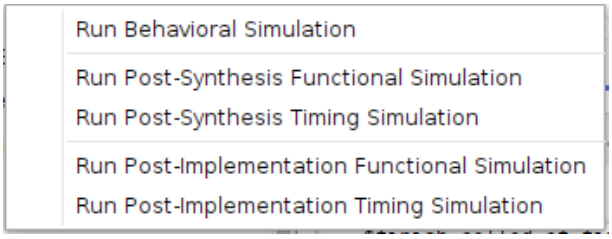


Fig. 60: Supported simulation options

It is worth mentioning that the default simulation time in Vivado is 1000ns. If one would like to run Post-implementation timing simulation, it is recommended to update the simulation time to 25000s. This can be done inside Vivado by navigate to "Settings" under "Project Manager", then go to "Simulation" tab under "Project Settings", and click on "Simulation" in the same tab. We then need to change `xsim.simulate.runtime` to be `25000ns`. Fig. 61 visualizes this process as a reference.

The simulation's waveform result can be found in 62, which is the same across behavioral simulation, post-synthesis simulation, and post-implementation simulation.

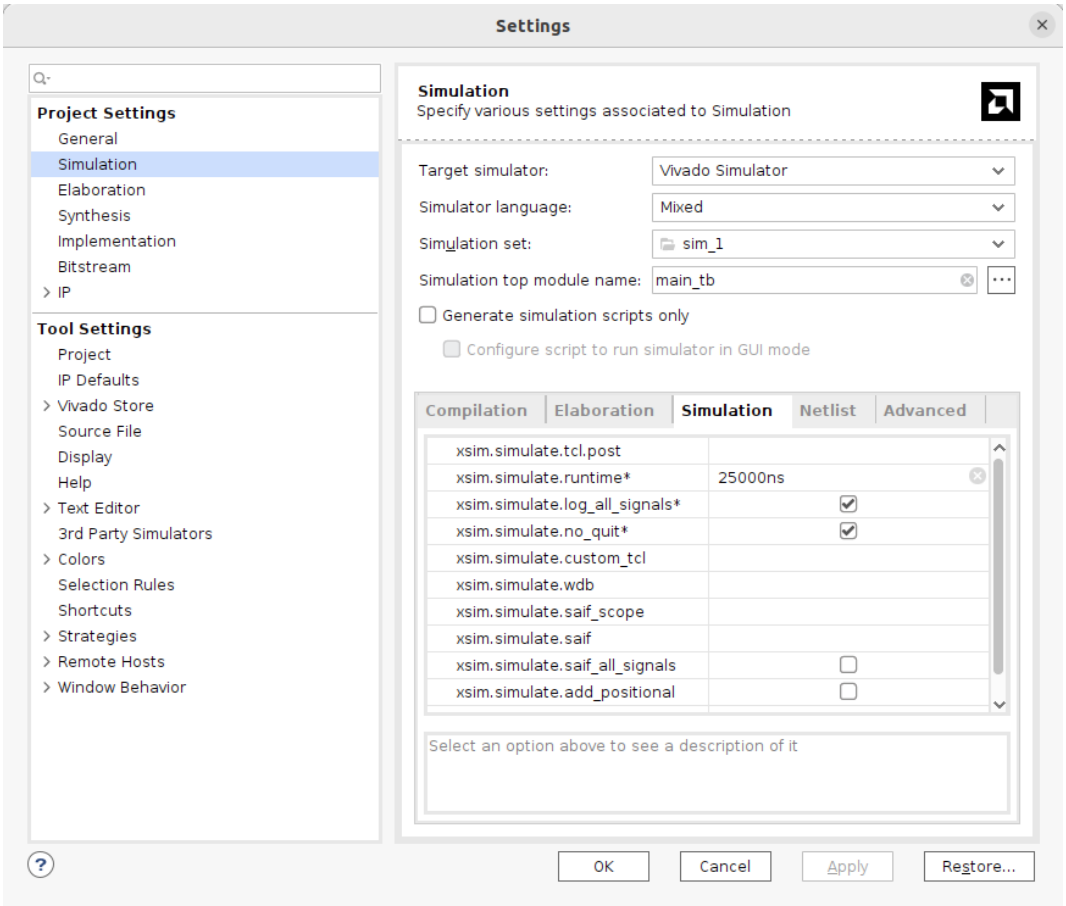


Fig. 61: Simulation time setting

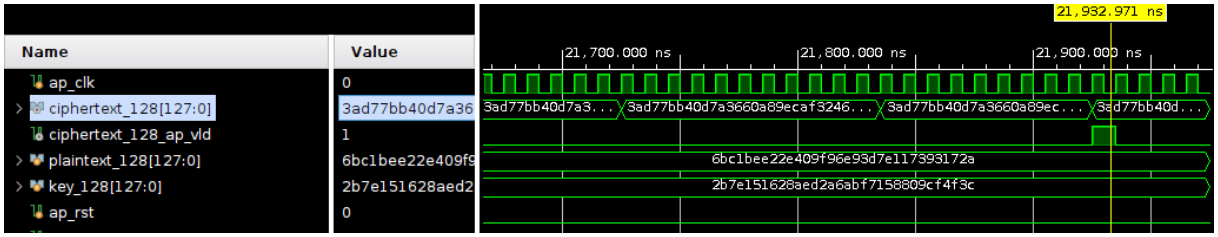


Fig. 62: Simulation's waveform result for final design

Once a VCD file is generated from the simulation, we will follow the same process as mentioned in Sec. 5.10 to parse it with TOFU and obtain a simulated trace. Fig. 63 shows example traces generated by TOFU from obtained VCD files for different stages of simulation: behvioral, post synthesis funcation and timing simulation, and post implementation functional and timing simulation.

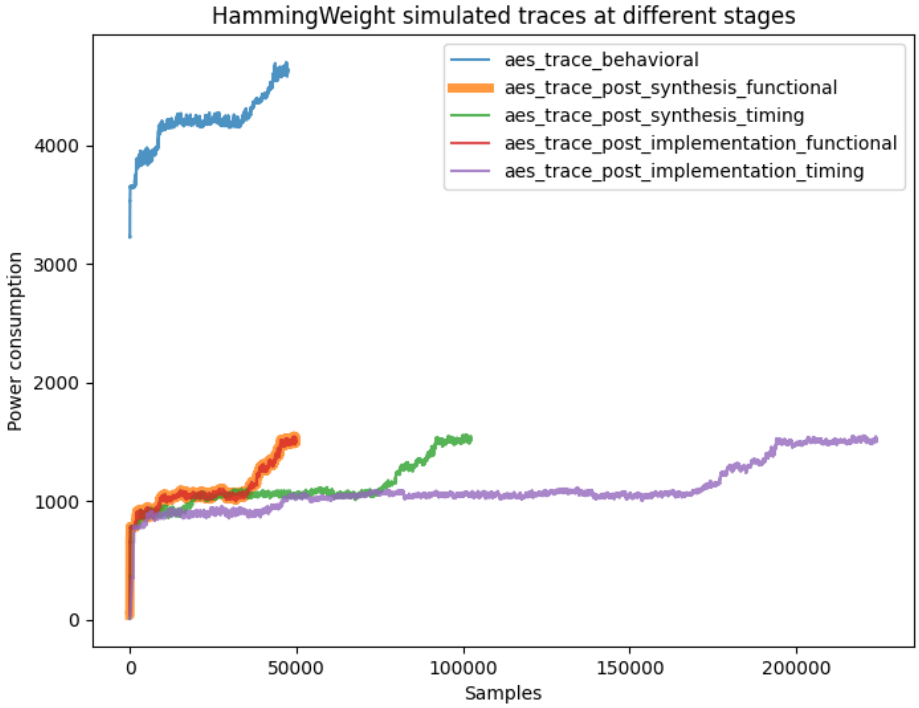


Fig. 63: Simulated traces (Hamming Weight in TOFU) across different levels of simulation in Vivado

5.15 Step 12: Establish A Dataset of Simulated Traces

Once we are able to generate a simulated trace from one VCD file, the next step is to generate multiple simulated traces automatically in a large scale given different plaintexts. Then we can merge these traces, plaintexts and the key into a single dataset saved in npz format for the downstream pre-silicon side-channel analysis.

BOYANG SAYS: Phuc: you can provide the details regarding how to generate multiple VCD files and traces automatically, and how to merge them into a single npz file

The process of generating multiple traces can be performed cleverly by leveraging Vivado's tcl scripting. In general, one can execute a tcl file in Vivado by running the following command:

```
vivado -mode tcl -source "<path␣to␣tcl␣file>"
```

In this document, we will not discuss in detail how to write a tcl file for Vivado, but rather we only go over the tcl file (`launch_simulation.tcl`) we will use to generate multiple simulated traces, which is presented below as a reference; one is highly encouraged to check Xilinx/AMD Technical Information Portal for more information about Vivado tcl commands [7].

```
# launch_simulation.tcl

open_project /home/mdphuc/UCDASEC/AES_Implementation/AESHardware/SMAesH-1.1.0/
    SMAesH_Vivado/SMAesH_Vivado.xpr
launch_simulation
set plaintexts "/home/mdphuc/UCDASEC/AES_Implementation/AESHardware/SMAesH
    -1.1.0/50000_plaintexts.txt"
set config_file "/home/mdphuc/UCDASEC/AES_Implementation/AESHardware/SMAesH-1.1.0/
    config.txt"

set fh [open $plaintexts r]
set plaintexts [split [read $fh] "\n"]
close $fh

set iteration 1

                # restart
                # run all
# Loop over each plaintext

foreach plaintext $plaintexts {
        if {$iteration <= 5000} {
                set cfg_fh [open $config_file w]
                puts $cfg_fh [format "%s,%d" $plaintext $iteration]
                close $cfg_fh

                restart
                run all

                puts "Iteration␣$iteration"
        }
        incr iteration
}
```

[7] https://docs.amd.com/r/en-US/ug835-vivado-tcl-commands/Introduction

```
31 exit
```

launch_simulation.tcl first runs open_project, which takes an absolute path to the Vivado project .xpr file, to open the Vivado project we created for the final design in previous steps. Next, it runs launch_simulation to put the design into simulation mode at RTL behavioral level. It's worth mentioning that, if one wants to run simulation at other levels, for example post synthesis functional or timing, they can simply replace launch_simulation with the corresponding line in the following code block:

```
1 launch_simulation -mode post-synthesis -type functional # functional simulation at
     post synthesis level
2 launch_simulation -mode post-synthesis -type timing # timing simulation at post
    synthesis level
3 launch_simulation -mode post-implementation -type functional # functional
    simulation at post synthesis level
4 launch_simulation -mode post-implementation -type functional # timing simulation
    at post synthesis level
```

Next, launch_simulation.tcl read from plaintext file (50000_plaintexts.txt, a file that contains 50000 128 bits plaintext we use in the experiment; any plaintext file is acceptable here, but one is advised to keep each plaintext on its own line) and store its data in variable plaintexts. Next, launch_simulation.tcl starts a for loop, which iterates over every plaintext, write that plaintext and its corresponding index into the config file (config.txt), and finally restart the simulation. Once simulation has been restarted, a new set of plaintext is read from config file, and a new trace will be generated for our dataset.

The version of launch_simulation.tcl we are showing here is capable of generating 5000 simulated traces based on plaintext number 1 to 5000 in 50000_plaintexts.txt. If one wants to generate a larger number of traces, they can replace number 5000 in line 19 with any desire number. For example, if one wants to target 10000 traces, they can modify line 19 as followed:

```
1 # Line 19 of launch_simulation.tcl
2     if ($iteration <= 10000) {
```

As discussed above, new plaintext and its corresponding index are written into config.txt; with this set up, we may need to update testbench.v such that it's able to read from config.txt to update the value of input plaintext and also set the name for output vcd file. To achieve this, we make use of Vivado's task; essentially, task is similar to a function with no return and can be called at any time. We create a task named loadConfig, which is presented below as a reference, and place loadConfig() in the **stimulis**.

```
1     task loadConfig;
```

```
        begin
            fileHandler = $fopen("/home/mdphuc/UCDASEC/AES_final_design/config.txt
                ","r");
            if (fileHandler) begin
                $fscanf(fileHandler, "%x,%d", plaintext_128, plaintextNum);
                $fclose(fileHandler);
            end
//              if (seedHandler) begin
//                  $fscanf(seedHandler, "%x", seed);
//                  $fclose(seedHandler);
//              end
            $sformat(vcdFilename, "/home/mdphuc/UCDASEC/AES_final_design/
                TRACE_EXAMINATION/plaintext%d.vcd", plaintextNum);
            $display(vcdFilename);
            $dumpfile(vcdFilename);
            $dumpvars(0, dut);
            $display("Dumping to: %s", vcdFilename);
        end
    endtask
```

Once `launch_simulation.tcl` has finished, 5000 simulated traces can be found under path stored in `vcdFilename`, which in our case, it is `/home/mdphuc/UCDASEC/AES_final_design/TRACE_EXAMINATION/`.

At this point, the process is the same as one used when generating one single trace. Specifically, we need to remove any empty line and replace `integer` with `reg` in each of the vcd files. Then we follow TOFU pipeline to generate h5 trace for each vcd file. This process can easily be automated using a simple Python script.

Once we have all the h5 trace in place, we can go ahead and pack them into a `.npz` dataset. The dataset contains 3 parts:

1. "plain_text": list of all plaintext
2. "power_trace": list of all h5 trace
3. "key": key used in encryption in the form of array of 16 bytes

It's worth mentioning that the order of "plain_text" should be aligned with the order of "power_trace", meaning at the same index i, power_trace[i] should be the trace got from running the design with plain_text[i]. To ease the process, we propose a Python program (`pack_trace.py`) for automation, which is presented below as a reference:

```
import os
import json
import argparse
import sys
import h5py
```

```
import numpy as np

def parseArgs(argv):
    parser = argparse.ArgumentParser()
    parser.add_argument('-i', '--input_dir', help='Input vcd trace dir')
    parser.add_argument('-t', '--trace', help="Trace (num start)_(num end)")
    parser.add_argument('-p', '--plaintext', help="Plaintext file")
    parser.add_argument("-o", "--output", help="Output")
    opts = parser.parse_args()
    return opts

def hex_str_to_array(input):
    hex_array = []

    for i in range(0, len(input), 2):
        hex_array.append(16 * int(input[i], 16) + int(input[i+1], 16))

    return hex_array

def hex_array_to_hex_str(input_arr):
    hex_str = ""
    for i in range(len(input_arr)):
        if len(str(hex(input_arr[i]))[2:]) == 1:
            hex_str += "0" + str(hex(input_arr[i]))[2:]
        else:
            hex_str += str(hex(input_arr[i]))[2:]

    return hex_str

if __name__ == "__main__":
    opts = parseArgs(sys.argv)

    power_traces = []
    plain_texts = []
    key_str = "2b7e151628aed2a6abf7158809cf4f3c"

    key = hex_str_to_array(key_str)

    trace_start_index = int(opts.trace.split("_")[0])
    trace_end_index = int(opts.trace.split("_")[1])

    vcd_traces = os.listdir(opts.input_dir)

    vcd_traces_core = []

    for vcd_trace in vcd_traces:
        if "plaintext" in vcd_trace:
```

```
            vcd_traces_core.append(vcd_trace)

    vcd_traces = vcd_traces_core

    tmp = sorted(vcd_traces, key=lambda x: int(x.split(".")[0][9:]))

    vcd_traces = tmp


    with open("{0}".format(opts.plaintext), "r") as f:
        plaintexts = f.readlines()

    size_trace = []

    for i, vcd_trace in enumerate(vcd_traces):
        if "plaintext" in vcd_trace and "h5" in vcd_trace:
            index = int(vcd_trace[9:].split(".")[0])

            if index >= trace_start_index and index <= trace_end_index - 1:
                with h5py.File('{0}/plaintext{1}.h5'.format(opts.input_dir, index)
                    , 'r') as file:
                    dataset = file['/leakages']
                    data = dataset[:, :]

                data = data.reshape(-1)

                size_trace.append(len(data))

    for i, vcd_trace in enumerate(vcd_traces):
        if "plaintext" in vcd_trace and "h5" in vcd_trace:
            index = int(vcd_trace[9:].split(".")[0])

            if index >= trace_start_index and index <= trace_end_index - 1:
                with h5py.File('{0}/plaintext{1}.h5'.format(opts.input_dir, index)
                    , 'r') as file:
                    dataset = file['/leakages']
                    data = dataset[:, :]

                data = data.reshape(-1)

                power_traces.append(data[:min(size_trace)])
                plain_texts.append(hex_str_to_array(plaintexts[index - 1].split("\
                    n")[0]))

                print("Done␣{0}/{1}".format(i + 1, trace_end_index -
                    trace_start_index), hex_array_to_hex_str(plain_texts[i]))

```

```
96      np.savez("{0}".format(opts.output), plain_text = plain_texts, power_trace =
            power_traces, key = key)
```

Once `pack_trace.py` has finished, a `npz` dataset can be found under `opts.output`.

References

1. AMD/Xilinx. AMD Design Suite. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html, 2013.
2. Cadence. Stratus High-Level Synthesis. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html, 2015.
3. CENSUS. Masked AES. https://github.com/CENSUS/masked-aes-c, 2020.
4. Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. Invited: Bambu: an open-source research framework for the high-level synthesis of complex applications, Dec 2021.
5. Hao Zheng, University of South Florida. High-Level Synthesis, Creating Custom Circuits from High-Level Code. https://cse.usf.edu/~haozheng/teach/cda4253/slides/hls-intro.pdf.
6. kokke. TinyAES: Small portable AES128/192/256 in C. https://github.com/kokke/tiny-AES-c, 2019.
7. Lab-STICC, Université de Bretagne-Sud. GAUT - A Free and Open-Source High-Level Synthesis tool. https://wiki.f-si.org/index.php/GAUT_-_A_Free_and_Open-Source_High-Level_Synthesis_tool, 2010.
8. Siemens. Catapult High-Level Synthesis and Verification. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/, 2004.
9. tueisec. TOFU - Toggle Count Analysis made simple. https://gitlab.lrz.de/tueisec/tofu, 2022.