SIEMENS EDA

Algorithmic C (AC) Communication Library Reference Manual

Software Version v3.4.5
August 2022

Table of Contents

Chapter 1: Introduction to ac_comm

The Algorithmic C Communication Library (*ac_comm*) contains synthesizable C++ functions commonly used in Communication like layer mapping and precoding. The functions use a C++ class-based object design so that it is easy to instantiate multiple variations of objects into a more complex subsystem and utilize the AC Datatypes for true bit-accurate behavior.

The input and output arguments of these functions are parameterized so that arithmetic may be performed at the desired precision and provide a high degree of flexibility on the area/performance trade-off of hardware implementations obtained during Catapult synthesis.

The following sections provide a summary of the *ac_comm* library:

1.1. Using the ac_comm Library

In order to utilize any of the top-level *ac_comm* classes, locate the required *ac_** header file from the *ac_comm* directory, and include that in your code. A summary of the files and classes within the *ac_comm* directory is given in Summary of Classes. A line that includes the required file for the *ac_lte_map_precode* class is given below.

```
#include <ac_lte_map_precode.h>
```

Each class has a public *run()* interface function that is synthesized as the top-level block by catapult and is called by the user in their design/testbench, in accordance with the requirements of class-based hierarchy.

The following coding requirements must be kept in mind while using the *ac_comm* library. Failure to do so can result in erroneous functioning.

- The user must ensure that they write data inputs for a single set of inputs every time they call the *run()* function. The user must not pass any more or less data than that.
- If they design uses DirectInputs, the user must make sure that they do not change while the design is processing a frame in hardware.

1.2. Summary of Classes

The following tables summarize the classes currently supported in the *ac_comm* library.

Description	Class Name(s)	Header File(s)
LTE MIMO Layer Mapper and Precoder	ac_lte_map_precode *Sub-blocks:* ac_lte_map ac_lte_precode	ac_lte_map_precode.h
Bit-to-Symbol Mapping	ac_bit2symbol	ac_bit2symb_mapping.h
LTE Rate 1/3 Tail Biting Convolutional Encoder	ac_conv_encoder	ac_conv_encoder.h
Advanced Encryption Standard	ac_aes	ac_aes.h
LDPC Encoder/Decoder	ac_ldpc_5g_decoder ac_ldpc_5g_encoder	ac_ldpc_5g_decoder.h ac_ldpc_5g_encoder.h
LTE Rate Matching	ac_ratematch	ac_ratematch.h
LTE Rate 1/3 Turbo Encoder	ac_turbo_encoder	ac_turbo_encoder.h

1.3. Installing the ac_comm Library

The library consists of the directories and files shown here:

```
.
|-- include
|   |-- ac_comm
|   |   |-- ac_lte_map_precode.h
|   |   |-- ac_bit2symb_mapping.h
|   |   |-- ac_conv_encoder.h
|   |   |-- ac_aes.h
|   |   |-- ac_ldpc_5g_decoder.h
|   |   |-- ac_ldpc_5g_encoder.h
|   |   |-- ac_ratematch.h
|   |   `-- ac_turbo_encoder.h
|-- pdfdocs
|   `-- ac_comm_ref.pdf
`-- tests
    |-- rtest_ac_lte_map_precode.cpp
    |-- rtest_ac_ratematch.cpp
    `-- rtest_ac_turbo_encoder.cpp
```

In order to utilize this library you must have the AC Datatypes package and the AC Math package installed and configure your software environment to provide the path to the “include” directories of these packages as part of your C++ compilation arguments.

Chapter 2: Communication Library

The *ac_comm* package includes the following communication blocks:

- LTE MIMO Mapper and Precoder
- LDPC Encoder/Decoder
- Advanced Encryption Standard
- <placeholder, make sure you attach a cross-reference to your section here>

The communication libraries accept inputs and produce outputs through streaming ac_channel interfaces.

2.1. LTE MIMO Mapper and Precoder

The ac_lte_map_precode library provides an efficient hardware implementation for the LTE MIMO Layer Mapping and Precoding algorithm built according to LTE specifications listed in [1]. It can be pipelined with an II of 1 and works with ac_complex inputs/outputs.

2.1.1. Introduction

The design consists of two sub-blocks, ac_lte_map and ac_lte_precode. The former takes care of layer mapping while the latter takes care of precoding. Both these sub-blocks are implemented as separate C++ classes. The top-level design, i.e. ac_lte_map_precode, is a wrapper around both these sub-blocks and interconnects them appropriately. The input to ac_lte_map_precode is first passed to ac_lte_map, which carries out layer mapping. The layer-mapped output is then passed as input to ac_lte_precode, which carries out precoding. The precoder output is the final output of the design.

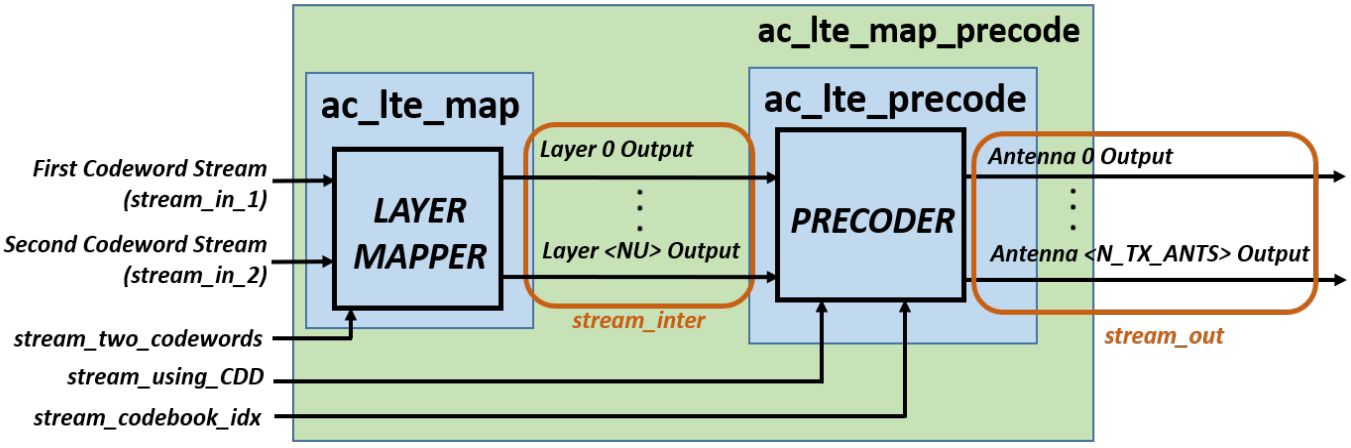

Illustration 1: Block Diagram of LTE MIMO Mapper and Precoder.

Refer to Illustration 1 for a generic block diagram of the mapper and precoder with NU number of layers and N_TX_ANTS number of antennas. The mapper and precoder receives two streaming inputs, stream_in_1 and stream_in_2. The former contains inputs from the first codeword and the latter contains inputs from the second codeword, if it is being used. If the design is configured to accept two codewords, inputs from

stream_in_1 and stream_in_2 are split across NU layer outputs. If the design is configured to accept only one codeword, the design only splits the input from stream_in_1 across NU layer outputs. The codeword-to-layer mapping scheme followed is specified in [1]. Whether the design accepts one codeword or two is signaled by a boolean flag sent on stream_two_codewords.

The output from the layer mapper is concatenated into an ac_matrix object and passed to the precoder via the stream_inter interconnect channel. The W(i) precoding matrix is extracted based on the codebook index value streamed via stream_codebook_idx. If the design is not using the Cyclic Delay Diversity (CDD) scheme, W(i) will be multiplied directly with the precoder input. If the design is using CDD, W(i) will be multiplied with the D(i) and U matrices, and the resulting product will be multiplied with the output from the layer mapper to form the precoder output. Similar to the stream_two_codewords channel, the stream_using_CDD channel also carries a flag which tells the design whether it should use CDD or not.

2.1.2. C++ Code Overview

As mentioned earlier, the top-level design and the sub-blocks are all implemented as C++ classes. Given below is a snippet showing the template parameters of all these classes, along with the run() function and all the associated IO typedefs:

```
template <class IN_FXPT_TYPE, int NU, int M_0_SYMB_MAX>
class ac_lte_map
{
public:
  typedef ac_complex<IN_FXPT_TYPE> IN_TYPE;
  typedef ac_int<ac::nbits<M_0_SYMB_MAX>::val, false> M_0_SYMB_TYPE;
  typedef ac_matrix<ac_complex<IN_FXPT_TYPE>, NU, 1> INTER_TYPE;
  enum { M_LAYER_SYMB_MAX = M_0_SYMB_MAX/(NU/2) };
  typedef ac_int<ac::nbits<M_LAYER_SYMB_MAX>::val, false> M_L_S_TYPE;

  #pragma hls_pipeline_init_interval 1
  #pragma hls_design interface
  void CCS_BLOCK(run) (
    ac_channel<IN_TYPE> &stream_in_1,
    ac_channel<IN_TYPE> &stream_in_2,
    ac_channel<bool> &stream_two_codewords,
    ac_channel<INTER_TYPE> &stream_inter,
    const M_L_S_TYPE M_layer_symb
  )

[…]

template <class IN_FXPT_TYPE, class OUT_FXPT_TYPE, int N_TX_ANTS, int NU, int
M_0_SYMB_MAX>
class ac_lte_precode
{
public:
  typedef ac_complex<IN_FXPT_TYPE> IN_TYPE;
```

```
  typedef ac_matrix<ac_complex<OUT_FXPT_TYPE>, N_TX_ANTS, 1> OUT_TYPE;
  typedef ac_int<ac::nbits<M_0_SYMB_MAX>::val, false> M_0_SYMB_TYPE;
  typedef ac_matrix<ac_complex<IN_FXPT_TYPE>, NU, 1> INTER_TYPE;
  typedef ac_int<4, false> C_IDX_TYPE;
  enum { M_LAYER_SYMB_MAX = M_0_SYMB_MAX/(NU/2) };
  typedef ac_int<ac::nbits<M_LAYER_SYMB_MAX>::val, false> M_L_S_TYPE;

  #pragma hls_pipeline_init_interval 1
  #pragma hls_design interface
  void CCS_BLOCK(run) (
    ac_channel<INTER_TYPE> &stream_inter,
    ac_channel<OUT_TYPE> &stream_out,
    ac_channel<bool> &stream_using_CDD, // Is CDD used?
    ac_channel<C_IDX_TYPE> &stream_codebook_idx, // Input stream: Codebook index.
    const M_L_S_TYPE M_layer_symb // Length of the layer mapping output.
  )

[…]

#pragma hls_design top
template <class IN_FXPT_TYPE, class OUT_FXPT_TYPE, int N_TX_ANTS, int NU, int
M_0_SYMB_MAX>
class ac_lte_map_precode
{
public:
  typedef ac_complex<IN_FXPT_TYPE> IN_TYPE;
  typedef ac_matrix<ac_complex<OUT_FXPT_TYPE>, N_TX_ANTS, 1> OUT_TYPE;
  typedef ac_int<ac::nbits<M_0_SYMB_MAX>::val, false> M_0_SYMB_TYPE;
  typedef ac_int<4, false> C_IDX_TYPE;
  enum { M_LAYER_SYMB_MAX = M_0_SYMB_MAX/(NU/2) };
  typedef ac_int<ac::nbits<M_LAYER_SYMB_MAX>::val, false> M_L_S_TYPE;

  #pragma hls_pipeline_init_interval 1
  #pragma hls_design interface
  void CCS_BLOCK(run) (
    ac_channel<IN_TYPE> &stream_in_1, // Input stream: First codeword.
    ac_channel<IN_TYPE> &stream_in_2, // Input stream: Second codeword.
    ac_channel<bool> &stream_two_codewords, // Are two codewords used?
    ac_channel<OUT_TYPE> &stream_out, // Output stream.
    ac_channel<bool> &stream_using_CDD, // Is CDD used?
    ac_channel<C_IDX_TYPE> &stream_codebook_idx, // Input stream: Codebook index.
    const M_L_S_TYPE M_layer_symb // Length of the layer mapping output.
  )
  // Code
```

The definition of the template parameters shown in the snippet is given below:

Parameters	Description
IN_FXPT_TYPE	Fixed point type for real and imaginary part of input.
OUT_FXPT_TYPE	Fixed point type for real and imaginary part of output.
N_TX_ANTS	Number of antennas.
NU	Number of layers.
M_0_SYMB_MAX	Maximum possible size for first codeword.

User Coding Requirements

In general, the following requirements must be kept in mind while using the design and calling the *run()* function. Failure to do so can result in erroneous functioning.

- The user must ensure that they write all data inputs for the first and second codewords (if using two codewords) every time they call the *run()* function. The user must not pass any more or less data than that. If the design has three layers and uses two codewords, the number of inputs in the second codeword must be twice that in the first codeword. For all other values of NU (number of layers), the second codeword has the same number of inputs as in the first.

- The user must also ensure that the value of M_layer_symb passed through the run() function is correctly initialized. The user can use the following line of code as a guideline for the correct initialization of M_layer_symb, where n_cw is the number of codewords and M_0_symb is the number of inputs in the first codeword:

```
int M_layer_symb = n_cw == 1 ? M_0_symb/NU : M_0_symb/(NU/2);
```

2.1.3. Architectural Overview

As can be seen in the snippet above, all the top-level run() functions are pipelined with an II of 1, which is signaled by the pragma hls_pipeline_init_interval 1. Due to this, all the loops internal to ac_lte_map are also pipelined with an II of 1 by default. The PRECODE_PC_IDX_LOOP in ac_lte_precode which loops over all the precoder inputs, is also pipelined with an II of 1, but all the loops internal to it—except for the ac_matrixmul loops located in ac_matrix.h and ac_matrixmul.h—are fully unrolled via pragmas. If synthesizing the design, it is recommended that the user unroll all loops in ac_matrix and ac_matrixmul.h as well, for maximum throughput.

2.1.4. Limitations

- Since MIMO/spatial multiplexing is being used, the number of antennas can only be 2 or 4.

- Since MIMO/spatial multiplexing is being used, the number of layers must be greater than one and must not exceed the number of antennas.

2.1.5. References

[1] Etsi.org. 3GPP TS 36.211 Version 14.2.0 Release 14. [online] Available at: https://www.etsi.org/deliver/etsi_ts/136200_136299/136211/14.02.00_60/ts_136211v140200p.pdf .

2.2. LDPC Encoder/Decoder

2.2.1. Introduction

The 5G mobile communications systems offer a far higher performance level beyond former generations of mobile communications systems. Wireless data traffic is estimated to increase by 1000-fold by the end of 2020 with more than 50 billion mobile devices connected to these wireless networks with peak data rates up to 10 Gbps. Forward error correction plays an extremely crucial role in high-speed communication systems. The search for an efficient trade-off between high performance, high throughput capabilities, low hardware complexity, low cost, and low power consumption makes the hardware implementation of an LDPC decoder still challenging. Designers has to deal with many possible options of algorithms, quantization parameters, parallelisms, code rates, and frame lengths. IP provides a reduced area and power are particularly compulsory for mobile devices. Therefore, designs of the area and energy-efficient FEC chips are excessively desirable. ac_5g_ldpc is low-complexity and high-throughput implementation LDPC Encoder and Decoder for emerging 5G wireless communications standards.

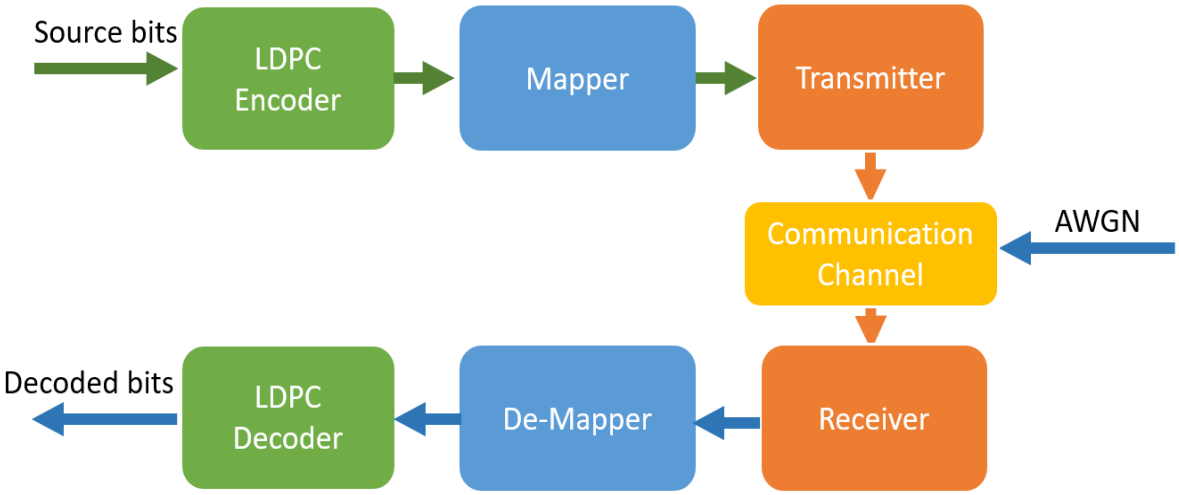

Illustration 2: Communication Block Diagram

2.2.2. C++ Code Overview

The top-level design and the sub-blocks are all implemented as C++ classes. Given below is a snippet showing the run() function of these classes. The typedefs, constants and a struct are defined in a file called ac_ldpc_5g_constants.h. Some of the structs are defined in a file ac_ldpc_5g_decoder.h

```
class ac_ldpc_5g_encoder{
```

```
  public:
  ac_ldpc_5g_encoder(){}
  #pragma hls_design interface
  void CCS_BLOCK(run)(ac_channel<ldpc_5g_config> &config_ch,
                      ac_channel<uint384>        &din_ch,
                      //ac_channel<ldpc_5g_config> &config_out_ch,
                      ac_channel<uint384>        &dout_ch)

[…]

class ac_ldpc_5g_decoder{
#pragma hls_design interface
   void CCS_BLOCK(run)(ac_channel<ldpc_5g_config> &config_ch,
         ac_channel<inpType>        &din_ch,
         ac_channel<ldpc_5g_config> &config_out_ch,
         ac_channel<uintb>        &dout_ch)
[…]
  // Code
```

User Coding Requirements

The following requirements must be kept in mind while using the design and calling the *run()* function. Failure to do so can result in erroneous functioning.

- The user must ensure that they write data inputs every time they call the *run()* function. Otherwise C++ execution will error out saying “empty channel read”.

2.2.3. Architectural Overview

LDPC 5G designs are expected to be high throughput and need to encode and decode bit streams on the go. ac_ldpc_5g_encoder/deocoder classes are pipelined at II=1 by default, provides high thoughput and low latency. Users can still customize the design from Catapult GUI as well

The architectural exploration options are detailed as follows:

- Interface shown in Illustration 3 is for ac_ldpc_5g implements interface configuration, input and output channels by default all channels are with ready valid interface.

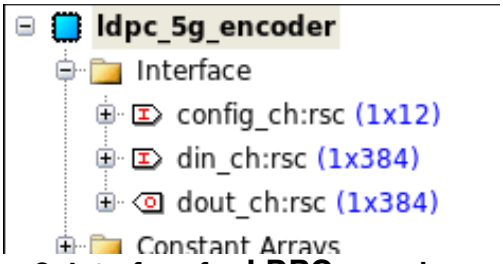

Illustration 3: Interface for LDPC encoder

- Design uses several constant arrays by default mapped to ROMs, user can map these array on registers.
- By default the design is architected to run at high throughput and thus having clock over Area penalty, as some loops shown in Illustration 3 are fully unrolled. User can partially unroll the loop to decrease both throughput and Area.

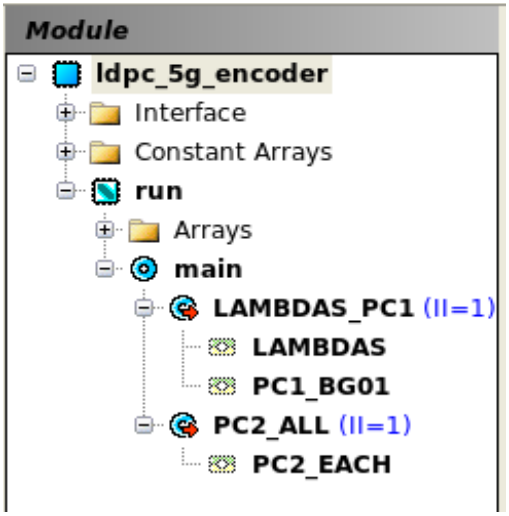

Illustration 4: Loop Options

2.2.4. Limitations

- LDPC Decoder is based on Min-Sum soft decoding only
- LDPC Decoder Still limited to BG 1. BG 2 Yet to be supported.
- Design is tested at 250MHz 45nm nangate lib

2.3. Advanced Encryption Standard

2.3.1. Introduction

This document specifies implementing the Rijndael algorithm , a symmetric block cipher that can process data blocks of 128 bits, using cipher keys with lengths of 128, 192, and 256 bits. Rijndael was designed to handle additional block sizes and key lengths, however they are not adopted in this standard. Throughout the remainder of this standard, the algorithm specified herein will be referred to as “the AES algorithm.” The algorithm may be used with the three different key lengths indicated above, and therefore these different “flavors” may be referred to as “AES-128”, “AES-192”, and “AES-256”.Illustration 5 shows the block diagram for AES Encryption and Decryption.

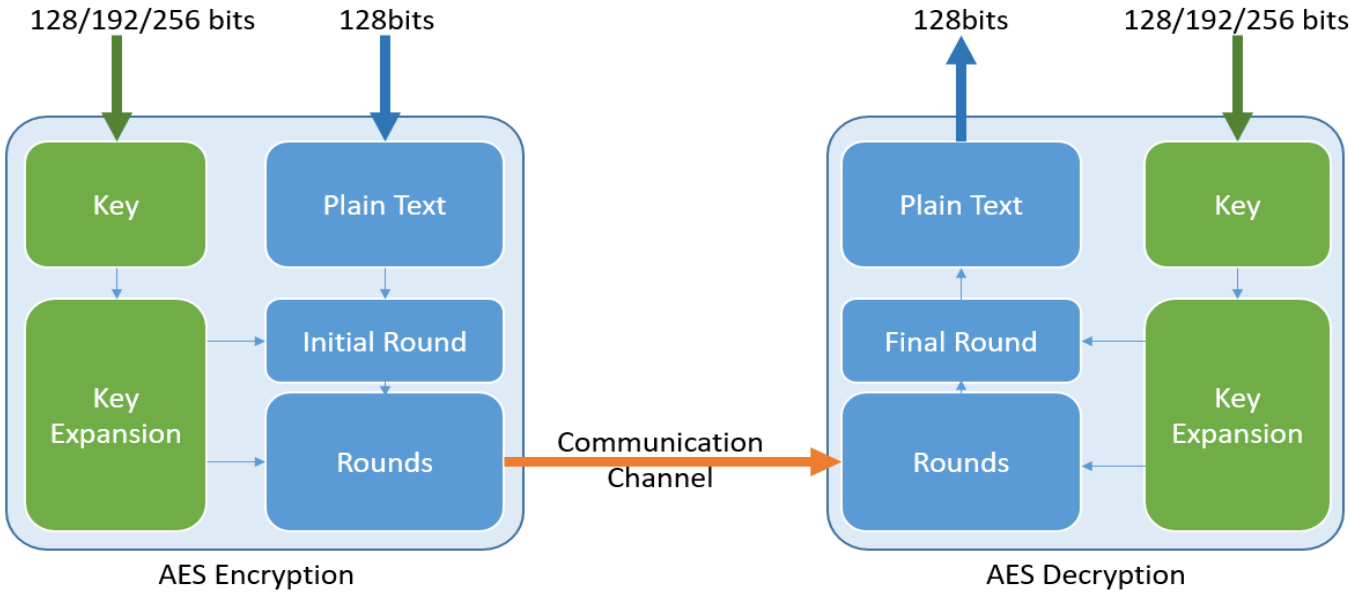

Illustration 5: AES Encryption and Decryption block diagram

2.3.2. C++ Code Overview

The top-level design and the sub-blocks are all implemented as C++ classes. Given below is a snippet showing the run() function of these classes. The typedefs, constants and a struct are defined in a file called ac_ldpc_5g_constants.h. Some of the structs are defined in a file ac_ldpc_5g_decoder.h

```
template <int keyLength = 128>
class ac_aes_encrypt {
#pragma hls_design interface
#pragma hls_pipeline_init_interval 1
```

```
  void CCS_BLOCK(run)(ac_channel< io_type > &inp_chan, ac_channel< ky_type > &key_chan,
ac_channel< io_type > &out_chan)

[…]

template <int keyLength = 128>
class ac_aes_decrypt {
#pragma hls_design interface
#pragma hls_pipeline_init_interval 1
  void CCS_BLOCK(run)(ac_channel< io_type > &inp_chan, ac_channel< ky_type > &key_chan,
ac_channel< io_type > &out_chan)
[…]
  // Code
```

User Coding Requirements

The following requirements must be kept in mind while using the design and calling the *run()* function. Failure to do so can result in erroneous functioning.

- The user must ensure that they write data inputs every time they call the *run()* function. Otherwise C++ execution will error out saying “empty channel read”.

2.3.3. Architectural Overview

AES designs by default are expected to be high throughput and need to encrypt and decrypt data on the go. ac_aes_encrypt/_decrypt classes are pipelined at II=1 by default providing high thoughput and low latency user can still customize the design from Catapult GUI

The architectural exploration options for each loop are detailed as follows:

- Interface shown in Illustration 6 is for AES-128 user can use multiple interface by default all channels are with ready valid interface.

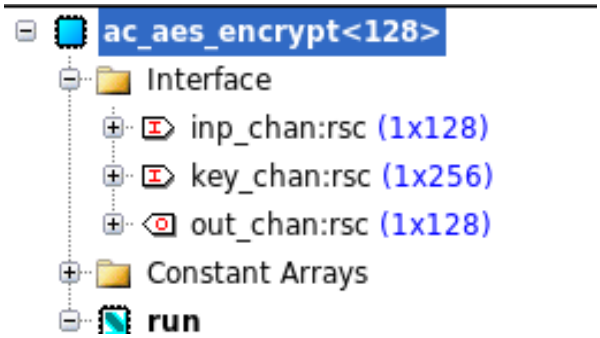

Illustration 6: Interface for AES encryption

- Design uses several constant arrays by default mapped to ROMs, user can map these array on registers.

- By default design run at a maximum throughput of 128bits/clock over Area penalty, as ROUND_LOOP shown in Illustration 7 is fully unrolled, user can partially unroll the loop to decrease both throughput and Area.

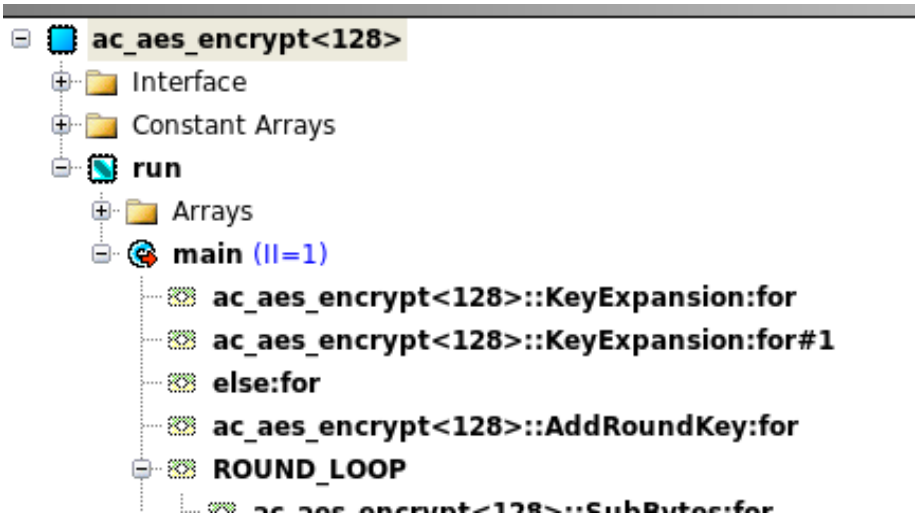

Illustration 7: Loop Options

2.3.4.

2.3.5. Limitations

- AES_ECB has general have limitations over other AES schema like AES_CBC and AES_CFB.
- Design is tested at 250MHz 45nm nangate lib

2.3.6.