ZUC-128
3GPP LTE Stream Cipher

CSE-293 Project

Priyanka Dutta & Sakshi Garg

Outline

Application

- ZUC128 – What & Why?

Implementation

- Interface
- State Machine
- Parameters
- Algorithm

Workflow and Learning

Verification

- Scala verification
- Chisel verification

Future work

- Advice for future students of the course

Documentation

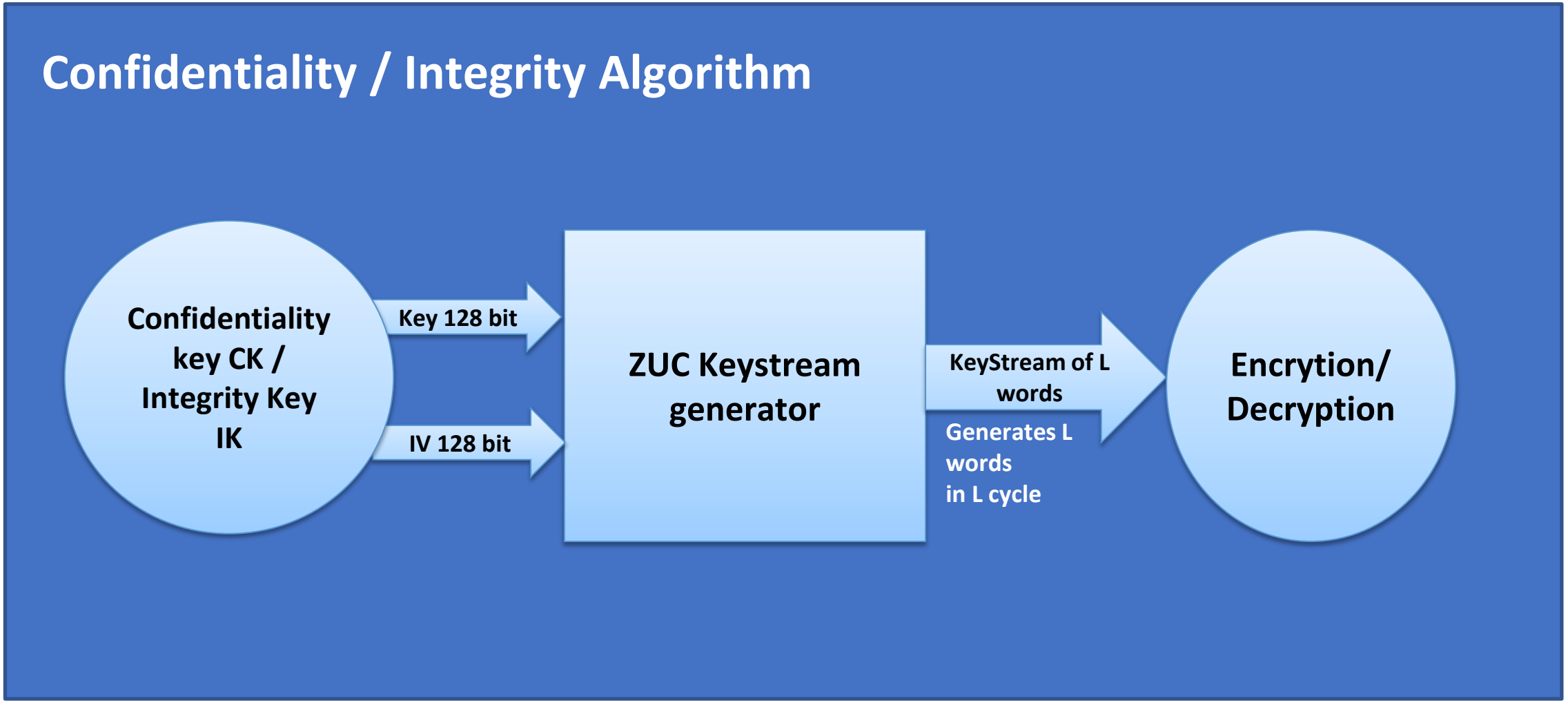
References & Links

ZUC128 – What & Why?

- A stream cipher used in 4G network
- Forms the heart of the 3GPP confidentiality algorithm 128-EEA3 and the 3GPP integrity algorithm 128-EIA3.
- Offers reliable security services in LTE networks.
- Used for data encryption and integrity protection of mobile communication systems
- Provide message encryption and ID authentication
- Robust enough to resist many existing cryptanalyses.

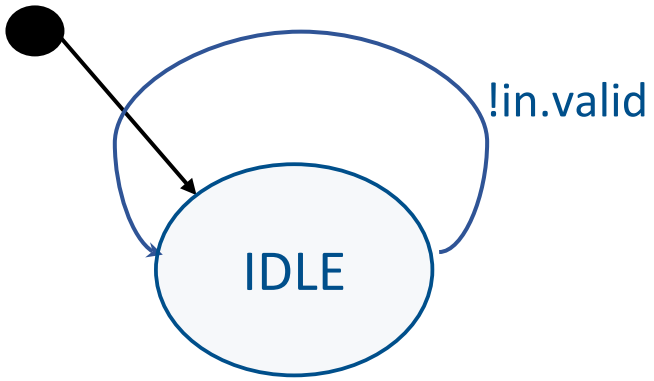
In September 2011, ZUC-128 was approved as the LTE international standard cipher for 3GPP at the 53rd 3GPP Meeting for System Architecture Group held at Fukuoka, Japan which is compatible with 4G.

ZUC Interface



ZUC-128: The heart of the 3GPP confidentiality algorithm 128-EEA3 and the 3GPP integrity algorithm 128-EIA3. Offers reliable security services in LTE networks.

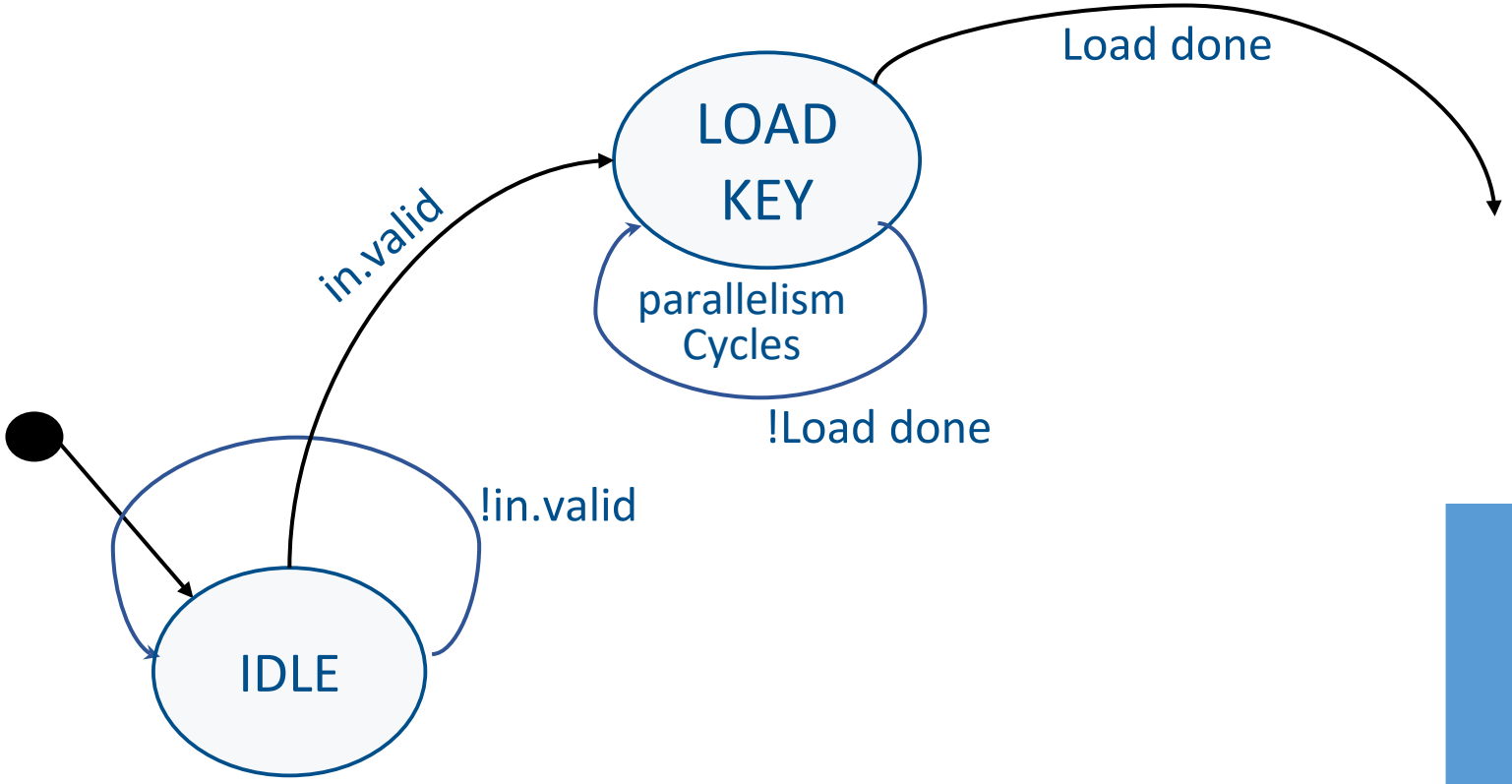
Implementation – State Machine



5 FSM states : IDLE :: LOADKEY :: INITMODE :: WORKMODE :: GEN KEYSTREAM

IDLE : In reset state or when the KeyStream generation is done.

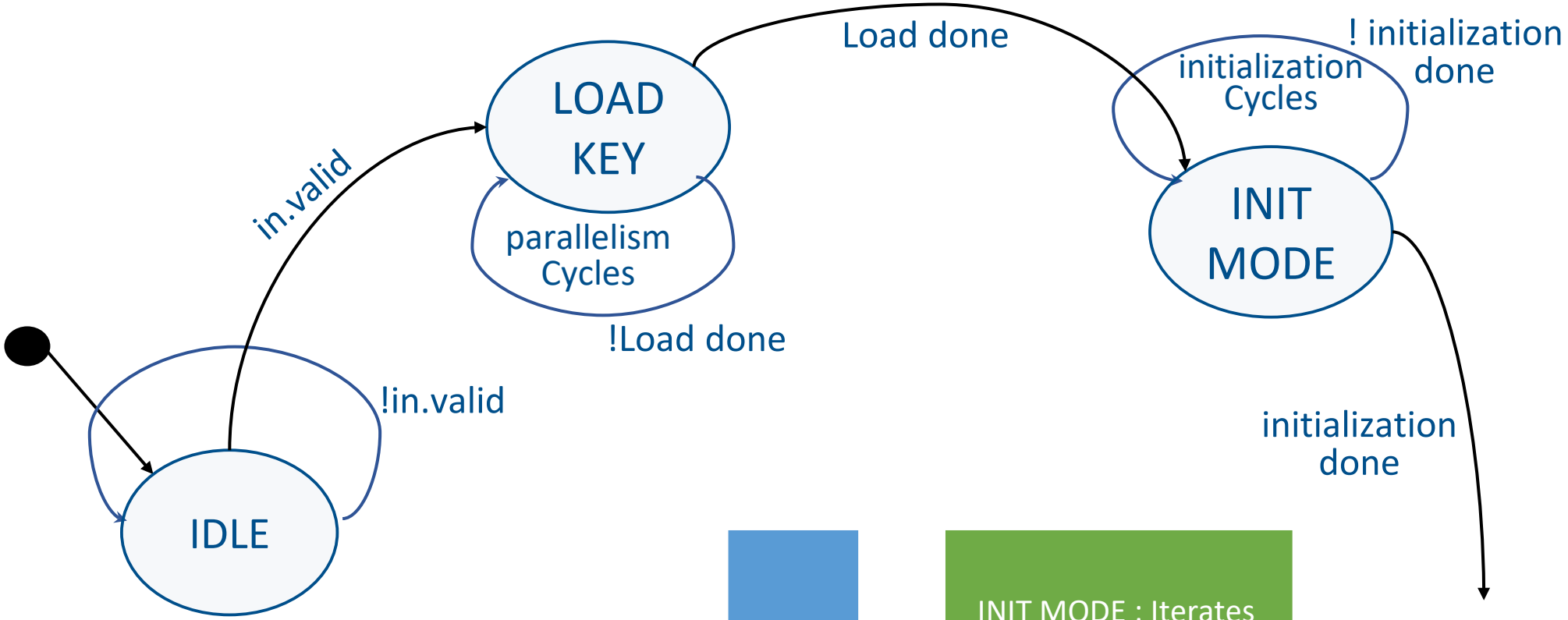
Implementation – State Machine



IDLE

LOAD KEY : System is ready and takes in all the inputs in parallelism number of cycles. Also, it initializes 16 LFSR registers.

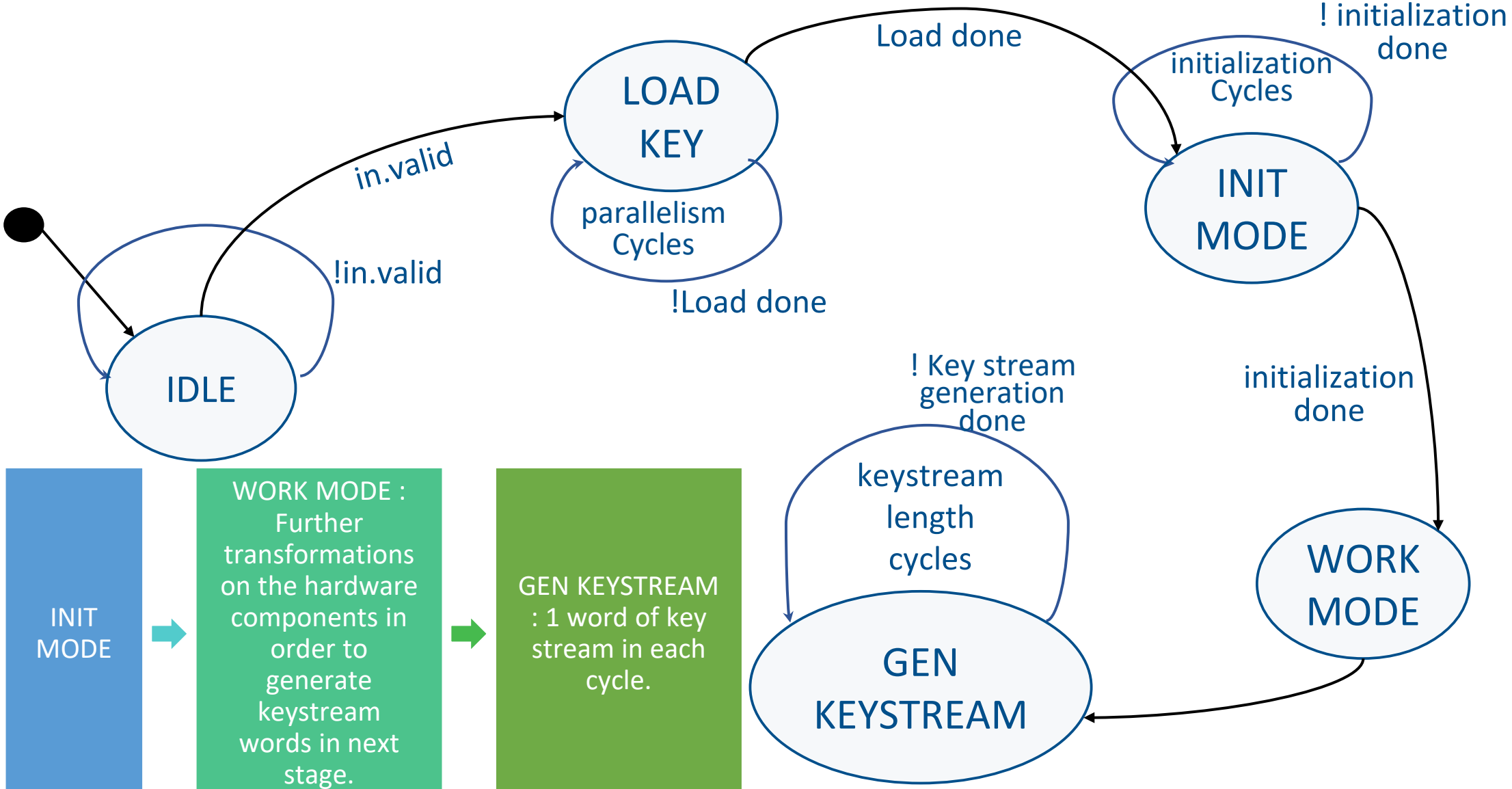
Implementation – State Machine



LOAD KEY

INIT MODE : Iterates through following stages 32 times: Bit-Reorganization, non-Linear function (F), and LFSR.

Implementation – State Machine

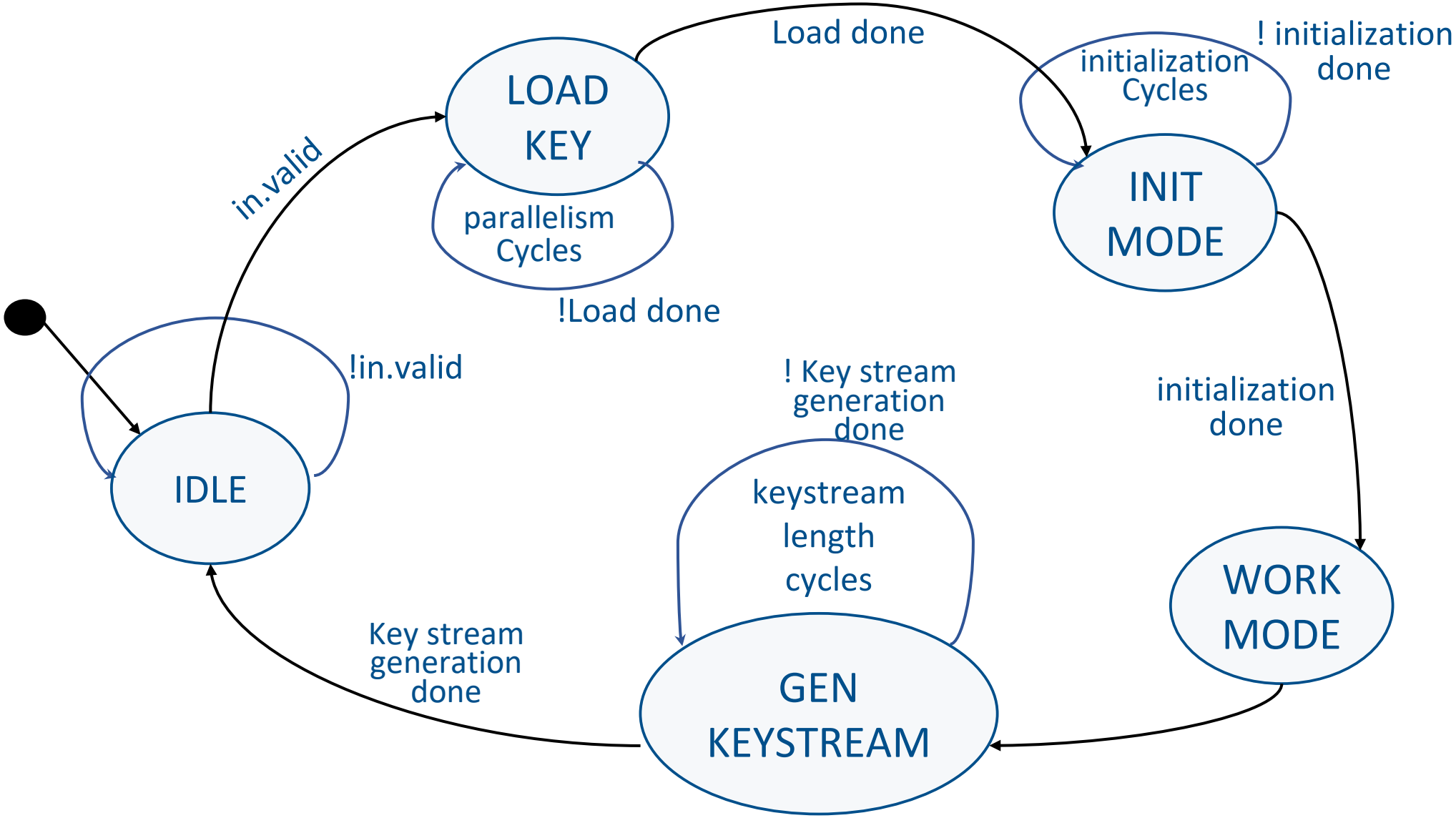


INIT MODE

WORK MODE : Further transformations on the hardware components in order to generate keystream words in next stage.

GEN KEYSTREAM : 1 word of key stream in each cycle.

Implementation – State Machine



ZUC Parameters

KeyStreamLength

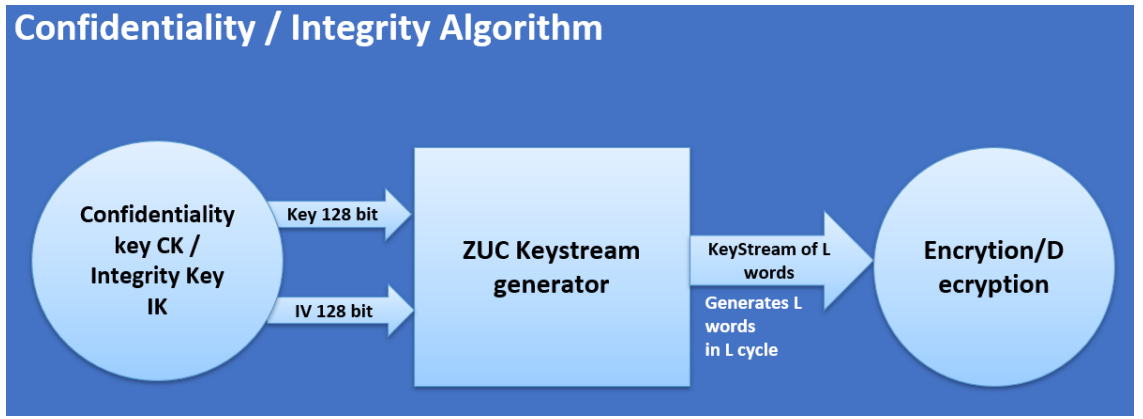
- ZUC will generate as many number of words of KeyStream as provided by this parameter
- Can be programed different values based upon encryption / decryption algorithm which is using this output to encrypt / decrypt data

Key Length

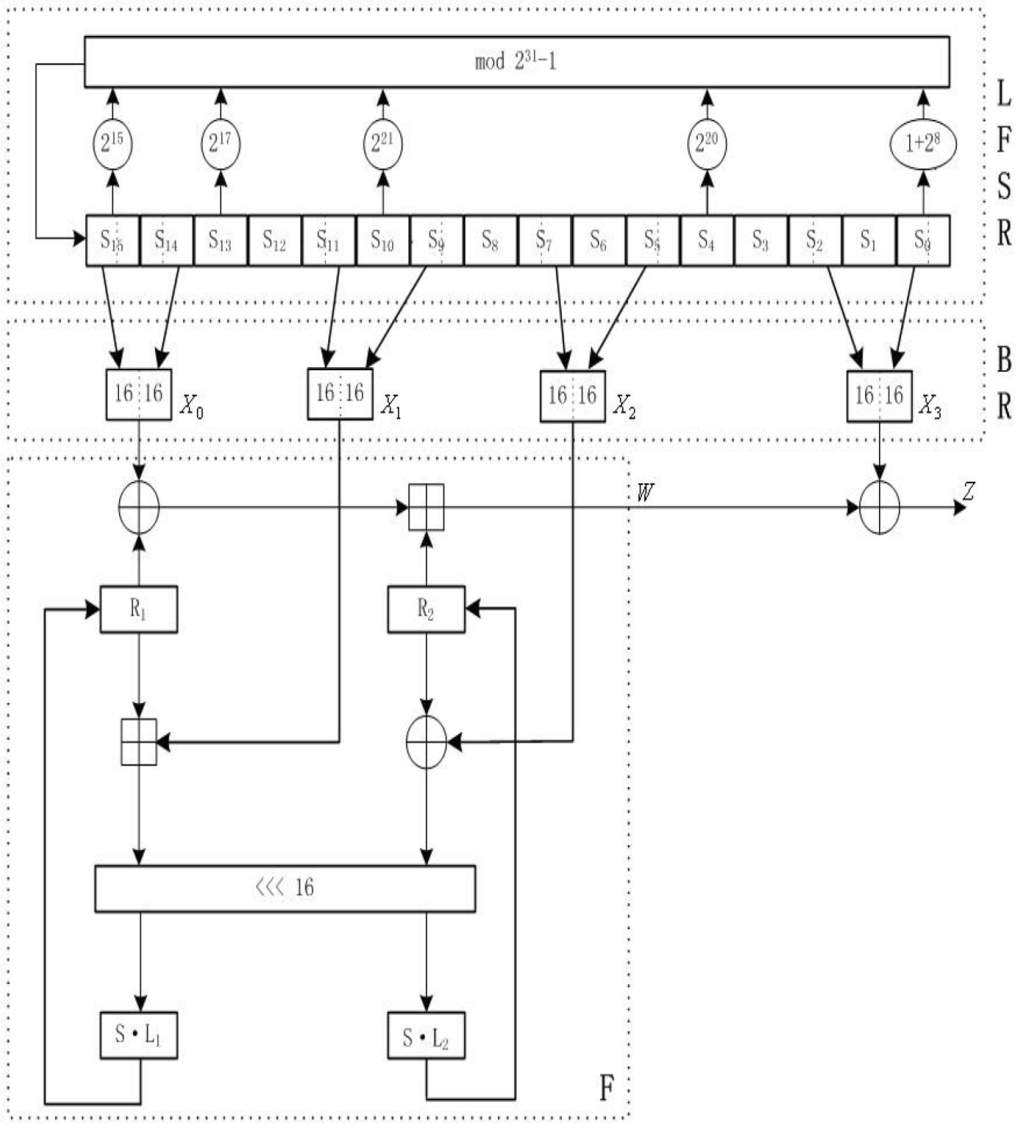
- Length of the input Key and IV (bytes)
- 16 byte for ZUC 128
- Can be made to 32 to inherit and implement for ZUC 256

Parallelism/Load cycles

- Parametrized clock cycles for loading input Key and IV
- Decrease hardware requirements by decreasing the need of loading into initial LFSR registers just in one cycle to multiple cycles



ZUC : Three physical layers of operation



- **Linear Feedback Shift Register**
  - 16 Register of 32 bit
  - mod(2^{31} -1) addition operation
- **Combinatorial logic block Bit Reorganization**
  - BRC_X0 , BRC_X1 , BRC_X2 and BRC_X3 (32 bits) are generated by using some combinational logic on latest LFSR values
- **Nonlinear function F**
  - F_R0 , F_R1 values are generated from BRC_X1 and BRC_X2 values combined with S boxes
  - W is evaluated from BRC_X0 , F_R0 and F_R1

ZUC: Components involved

✔ **LFSR_S(16)**
16 Registers of 32 bit each
updates at each clock cycle based upon some combinational logics and other components

✔ **F_R0 and F_R1**
(32 bit memory cells)

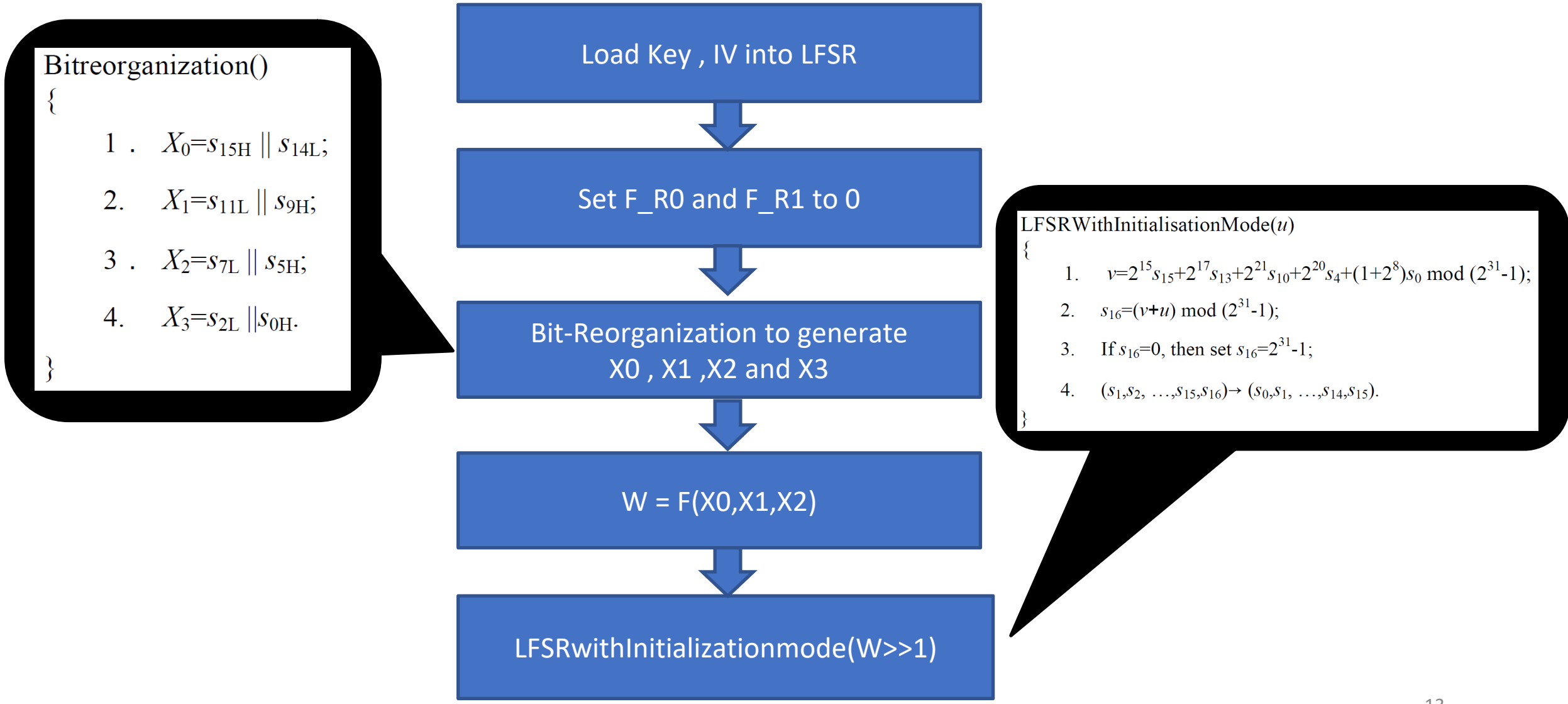
✔ **BRC_X**
4 of 32-bit words X0, X1, X2, X3
generated at Bit-Reorganization process from LFSR values combined with other logical operations
used to generate F_R values
BRC_X(3) is used for keystream generation

✔ **Ek_d Constant**
16 byte constant of unsigned Int
used with Key and IV while initializing LFSR

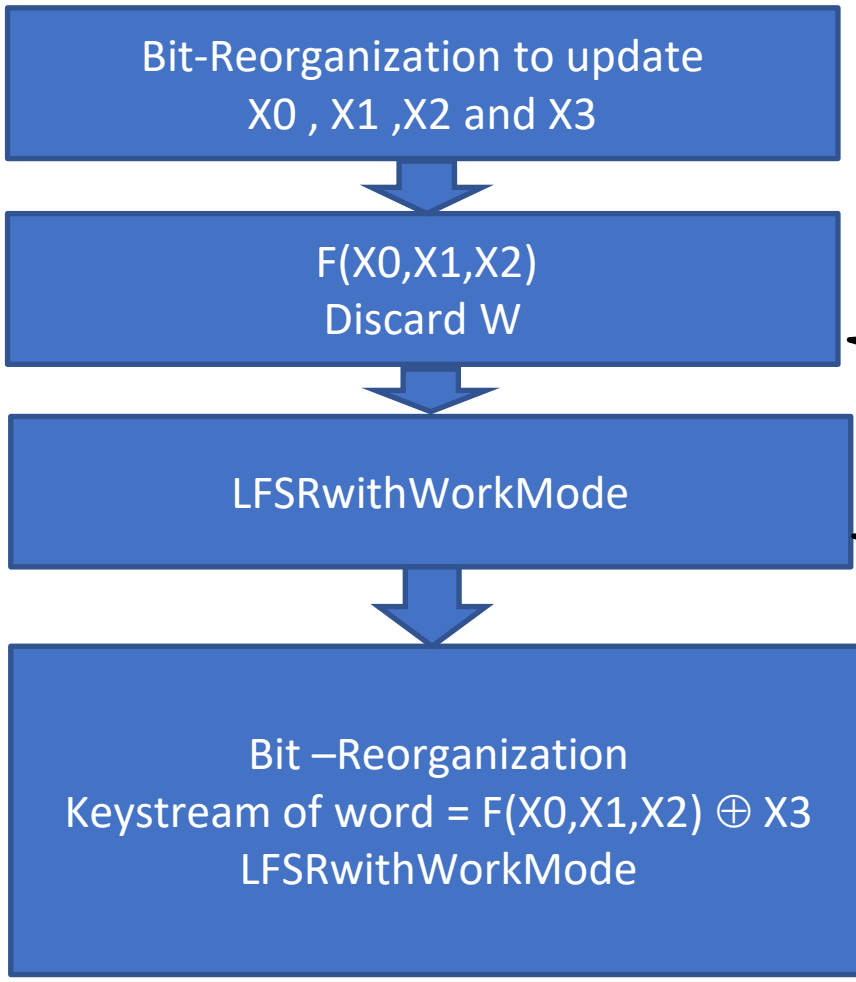
✔ **W**
output of non-linear function F
used in Initialization mode to update LFSR[15]
used in final keystream generation

✔ **S0 and S1 boxes constant**
The 32×32 S-box S is composed of 4 juxtaposed 8×8 S-boxes, i.e., S=(S0,S1,S2,S3)

ZUC Execution Stages : Initialization Stage



ZUC Execution Stage : Working Stage



F (X_0, X_1, X_2)
{

1. $W=(X_0 \oplus R_1) \boxplus R_2$;
2. $W_1= R_1 \boxplus X_1$;
3. $W_2= R_2 \oplus X_2$;
4. $R_1=S(L_1(W_{1L}\|W_{2H}))$;
5. $R_2=S(L_2(W_{2L}\|W_{1H}))$.

}

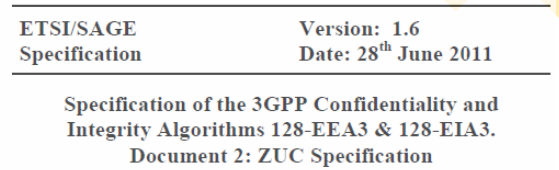
LFSRWithWorkMode()
{

1. $s_{16}=2^{15}s_{15}+2^{17}s_{13}+2^{21}s_{10}+2^{20}s_4+(1+2^8)s_0 \bmod (2^{31}-1)$;
2. If $s_{16}=0$, then set $s_{16}=2^{31}-1$;
3. $(s_1,s_2, \ldots,s_{15},s_{16}) \rightarrow (s_0,s_1, \ldots,s_{14},s_{15})$.

}

Workflow and learning

- Scala implementation guided by the ZUC specification document from Security Algorithms Group of Experts (SAGE).
  - Sequential implementation
  - Mutable vs. immutable Scala constructs. Started with Seq. Moved to ArrayBuffer
  - No Unsigned in Scala. So used BigInt instead of Int.
  - Using helper functions and learned various operators
    - Like >> vs >>> as per requirement.
  - Writing the test assertions into proper classes and individual test modules for unit testing.


ETSI/SAGE Specification
Version: 1.6
Date: 28th June 2011

Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification

The ZUC algorithm is the core of the standardised 3GPP Confidentiality and Integrity algorithms 128-EEA3 & 128-EIA3.

Workflow and learning

- Scala implementation guided by the ZUC specification document from Security Algorithms Group of Experts (SAGE).
  - Sequential implementation
  - Mutable vs. immutable Scala constructs. Started with Seq. Moved to ArrayBuffer
  - No Unsigned in Scala. So used BigInt instead of Int.
  - Using helper functions and learned various operators (Like >> vs >>> as per requirement).
  - Writing the test assertions into proper classes and individual test modules for unit testing.
- Tested : 4 vectors
  - Initialization mode
    - LFSR value
  - Post initialization mode
    - LFSR value, F_R value, W value
  - Output of keystream generated mode.
  - All the correct and expected intermediate and final values were extracted by compiling the c-code.
- After Scala implementation and testing for keystream generation of length = 1, we moved to Chisel implementation and testing

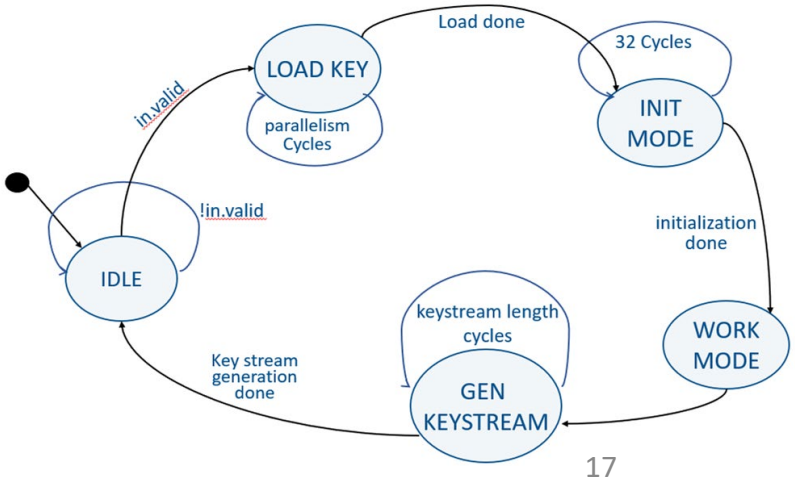
ETSI/SAGE Specification — Version: 1.6 — Date: 28th June 2011

Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3.
Document 2: ZUC Specification

The ZUC algorithm is the core of the standardised 3GPP Confidentiality and Integrity algorithms 128-EEA3 & 128-EIA3.

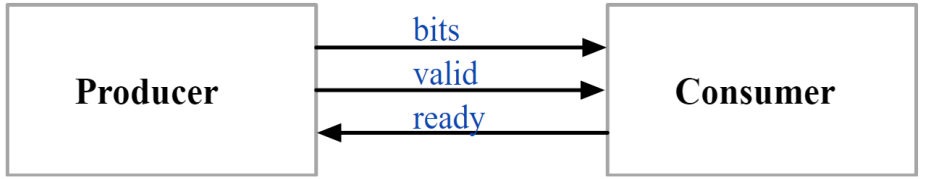
Workflow and learning

- Chisel implementation
  - Started with a basic sequential flow. Realized the need of FSM. Implemented one.
  - When to use Reg vs. when to use Wire in hardware
  - Using collections like Vec
  - Features for parametrization
    - Case class, object, IO class extending bundle and class extending module.
  - Ensuring no use of variables (“var”). Chisel constructs should be “val” since they are HW entities.
  - Improving tests using functional programming (zip, map, foreach)



Workflow and learning

- Chisel implementation
  - Started with a basic sequential flow. Realized the need of FSM. Implemented one.
  - When to use Reg vs. when to use Wire in hardware
  - Using collections like Vec
  - Features for parametrization
    - Case class, object, IO class extending bundle and class extending module.
  - Ensuring no use of variables (“var”). Chisel constructs should be “val” since they are HW entities.
  - Improving tests using functional programming (zip, map, foreach)
- After the system and its test were working for KS len=1, we parametrized and checked for KS length > 1.
- Added the feature of Decoupled IO
  - Use of “noenq” for indicating invalid data.
- Parametrized inputs to enable future inheritance.
- Introduced parallelism for input loading mechanism.



Scala Model verification

```
class ZUC_128_ModelTester extends FreeSpec with ChiselScalatestTester {

  "Initial LFSR value for the initialization mode. TC:1" in {...}
  "LFSR value for the POST initialization mode. TC:1" in {...}
  "F_R(1) value after the initialization mode. TC:1" in {...}
  "F_R(2) value after the initialization mode. TC:1" in {...}
  "W value after the initialization mode. TC:1" in {...}
  "Keystream generated. TC:1" in {...}
```

- Reference Model for verification is written in Scala
- Model has been tested to be golden by comparing values of each stage with the C – code provided with the document[2]
- Referred to four test vectors which covered mostly all possible corner cases for Input Keys and IV from document[3]
- Extracted intermediate values of all hardware components i.e. LFSR stages, F_R memory cells , W after non-linear function F ; for each test vector from compiled C –code and then verified with Scala model values
- Finally verified output KeyStream up-to Keystream length of 32

```
class ZUC128Tester extends FreeSpec with ChiselScalatestTester {

  def testZUC128(Key : Seq[UInt], Iv : Seq[UInt], keystreamlen : Int , outkeystream : Seq[SInt],Key_num : Int,parallelism :Int): Boolean = {...}

  "Hardware ZUC128 should generate correct keystream for Test Vector 1 (no parallelism)" in {...}
  "Hardware ZUC128 should generate correct keystream for Test Vector 2 (no parallelism)" in {...}
  "Hardware ZUC128 should generate correct keystream for Test Vector 3 (no parallelism)" in {...}
  "Hardware ZUC128 should generate correct keystream for Test Vector 4 (no parallelism)" in {...}
  "Hardware ZUC128 should generate correct keystream for Test Vector 1 (parallelism=2, HW requirement reduced, number of cycles increased by 2)" in {...}
  "Hardware ZUC128 should generate correct keystream for Test Vector 2 (parallelism=2, HW requirement reduced, number of cycles increased by 2)" in {...}
  "Hardware ZUC128 should generate correct keystream for Test Vector 3 (parallelism=2, HW requirement reduced, number of cycles increased by 2)" in {...}
  "Hardware ZUC128 should generate correct keystream for Test Vector 4 (parallelism=2, HW requirement reduced, number of cycles increased by 2)" in {...}
  "Hardware ZUC128 should generate correct keystream for Test Vector 1 (full parallelism; minimum HW requirement)" in {...}
  "Hardware ZUC128 should generate correct keystream for Test Vector 2 (full parallelism; minimum HW requirement)" in {...}
  "Hardware ZUC128 should generate correct keystream for Test Vector 3 (full parallelism; minimum HW requirement)" in {...}
  "Hardware ZUC128 should generate correct keystream for Test Vector 4 (full parallelism; minimum HW requirement)" in {...}
}
```

Chisel Verification

- Test 1 :Key (all 0) , IV (all 0) , KeyStream
- Test 2 :Key (all 1), IV(all 1), KeyStream
- Test 3: Key (random numbers) , KeyStream
- Test 4: Key(random numbers) , KeyStream
- Testing of idle state when no key and IV is loaded to Hardware is done
- Testing of decoupled Output KeyStream : valid when keystream is generated for keystream length of cycles
- Tested Loading of inputs in multiple cycles (parallelism) to reduce hardware requirement

Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 3: Implementor's Test Data

Future work

- Since the scope of this project limited the implementation of applications using the Stream Cipher, we intended to make the ZUC-128 Stream Cipher well enough to be used as a submodule without requiring changes for interfacing with any application module.
- Thus, this project paves way to other project ideas :
  - Implementation of confidentiality algorithm for encryption and decryption of message can be done as a future project just by plugging our ZUC 128 cipher as a hardware block.
  - Similarly, implementation of Integrity algorithm could be done.
  - Besides, ZUC 256 can be implemented by inheriting our ZUC 128 hardware model for more enhanced Keystream generation of 256-bit inputs of Key and IV.
  - Additionally, LFSR could be refactored to make a re-usable generator for easier use.

Advice for future students of the course

- Whenever the generator needs to work in different stages, for example, a counter in the system should work only after/before some state of the generator to be implemented, you should start with an FSM implementation. This would greatly reduce the possible difficulties that you might face otherwise.
- During chisel implementation, if tests are not passing due to some cycle mis-match then try debugging the enable and reset conditions of the counters implemented.
- Start with the project early because during later stages you might have some improvement ideas and they require your time. Also, do not hesitate to post your questions on Slack channel; you may get different perspectives from different replies.
- Prior Verilog knowledge could ease the learning process for this course.

Documentation

- Readme
- Well commented code

76 lines (49 sloc) | 7.08 KB

Raw Blame

ZUC-128 (3GPP LTE Stream Cipher)

This is a Chisel implementation of ZUC-128 (a 3GPP LTE Stream Cipher) used in 4G networks security.

Called also as Zu Chongzhi's cipher, ZUC-128 stream cipher is designed on the basis of 128-bit key. Including encryption algorithm 128-EEA3 and integrity algorithm 128-EIA3, it is mainly used for data encryption and integrity protection of mobile communication systems.

ZUC is a stream cipher that forms the heart of the 3GPP confidentiality algorithm 128-EEA3 and the 3GPP integrity algorithm 128-EIA3, offering reliable security services in Long Term Evolution networks (LTE). It is used for data encryption and integrity protection of mobile communication systems and provide message encryption and ID authentication.

In September 2011, ZUC-128 was approved as the LTE international standard cipher for 3GPP at the 53rd 3GPP Meeting for System Architecture Group held at Fukuoka, Japan which is compatible with 4G. Stream cipher ZUC plays a crucial role in the next generation of mobile communication as it has already been included by the 3GPP LTE-Advanced, which is a candidate standard for the 4G network.

Project details:

Algorithm implemented: `ZUC-128`

`ZUC` is a word-oriented stream cipher. It takes a 128-bit initial key and a 128-bit initial vector (IV) as input, and outputs a keystream of 32-bit words (where each 32-bit word is hence called a key-word). This keystream can be used for encryption/decryption.

The execution of ZUC has two stages: initialization stage and working stage. In the first stage, a key/IV initialization is performed, i.e., the cipher is clocked without producing output. The second stage is a working stage. In this stage, with every clock pulse, it produces a 32-bit word of output.

We have completed the Scala implementation along with necessary helper functions and testing with stages broken down into testing the input

References and links:

[1]. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 1: 128-EEA3 and 128-EIA3 Specifications.

[2]. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification.

[3]. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 3: Implementor's Test Data

[4].Link to the implementation: https://github.com/sakshi15108/CSE293Project

[5]. Documentation: https://github.com/sakshi15108/CSE293Project/blob/main/README.md

[6]. Part of initial theory obtained from: http://www.jcr.cacrnet.org.cn/EN/10.13868/j.cnki.jcr.000228

Thank you