“Stromchiffrierung mittels FPGA-Board Basys 3”

Projekt Arbeit
Vertiefung FPGA

Im Studiengang „Elektrotechnik“
An der Dualen Hochschule Baden-Württemberg Stuttgart

von

Robin Vogel und Furkan Cetinkaya

24.11.2020

Projektdauer	4 Wochen
Matrikelnummer, Kurs	(Furkan Cetinkaya) 9351417, TEL18Gr2
	(Robin Vogel) 3893394, TEL18Gr2
Firma	Thales Management & Services Deutschland GmbH, Ditzingen
Dozent	Klaus Gosger

Inhaltsverzeichnis

Tabbellenverzeichnis

Abkürzungsverzeichnis

FPGA	**Field Programmble Array**
IoT	**Internet of Things**
ECB	**Electronic Codebook**
CPU	**Central Processing Unit**
ROM	**Read Only Memory**
CLK	**Clock**
RST	**Reset**

1. Aufgabenstellung und Zielsetzung

Diese Arbeit umfasst ein studentisches Projekt, welches im Verlauf des Auswahlmoduls „Vertiefung FPGA“ bearbeitet wurde. Durchgeführt wurde dies in Zusammenarbeit von zwei Studierenden.

Das ausgewählte Projekt beschäftigt sich mit einer Stromchiffrierung (Kryptografie-Methode), die mittels eines Basys 3 Moduls realisiert werden soll.

Hierbei soll man einen Prozess definieren, welcher durch Starten des Programms den eingeschriebenen Klartext in einen Geheimtext verschlüsselt. Gleichzeitig soll dieser im Anschluss den verschlüsselten Text wieder entschlüsseln. Der Schlüssel soll dabei durch einen Zufallsgenerator generiert werden.

Eine erfolgreiche Ver- und Entschlüsselung ist erfolgt, wenn der eingeschriebene Klartext und der entschlüsselte Klartext identisch sind.

Dieses Projekt wird mit der Vorstellung und Abgabe dieser Ausarbeitung am 07.12.2020 beendet.

2. Einleitung Kryptografie

Im alltäglichen Menschenleben ist der Einsatz der Kryptografie allgegenwärtig. Beispielweise wird diese benötigt, um Passwörter für Online-Käufe über enorm große Netzwerke zu versenden. Bankserver oder andere im Security-Bereich arbeitende Knotenpunkte treten sehr häufig mit Kryptosystem in Kontakt. Zudem wird sie verwendet, um Informationen in einer IoT-verbundenen Welt zu übertragen oder auch Personen zu authentifizieren. Falls kryptografische Funktionen nicht mehr geregelt laufen können, würde das Alltagsleben wie man es kennt nicht mehr existieren. Von Banktransaktionen bis zum Internetverkehr, alle Prozesse kämen zum Erliegen und die Kommunikationssysteme würden kollabieren. Zu diesem Zeitpunkt würden alle wichtigen persönlichen aber auch staatlichen Informationen aufgedeckt und diese für Angreifer zu nutzen werden. Die Kryptografie ist ein essenzieller Weg, diese Schäden und Sicherheitslücken zu verhindern. Es verhilft Informationen von Sender auf sicherem Weg an Endverbraucher zu übermitteln. In einer zunehmend globalisierten Welt wie heute steigt die Nachfrage nach Cybersecurity und somit auch der Kryptografie, da durch die Vernetzung von allen Menschen durch ein gemeinsames Netz das Risiko für neue Angriffsmethoden steigt.

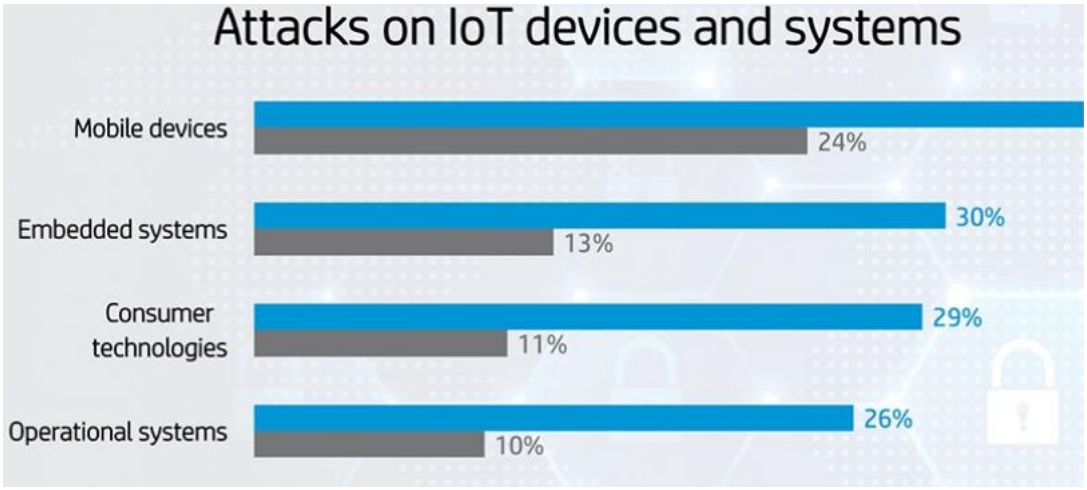

Abbildung_01: „Attacks on IoT devices and systems from 2014 to 2015“

3. Grundlagen der Kryptografie

Verschlüsselungen beschreiben Verfahren bzw. Algorithmen, welche sensiblen Daten mittels generierten Schlüssels einen sogenannten Klartext in einen nicht entzifferbaren Geheimtext umsetzen. Dieser Vorgang wird als das „Encryption“ bezeichnet. Ein Anwender kann somit nur durch das Besitzen eines solchen Schlüssels den Geheimtext entschlüsseln.

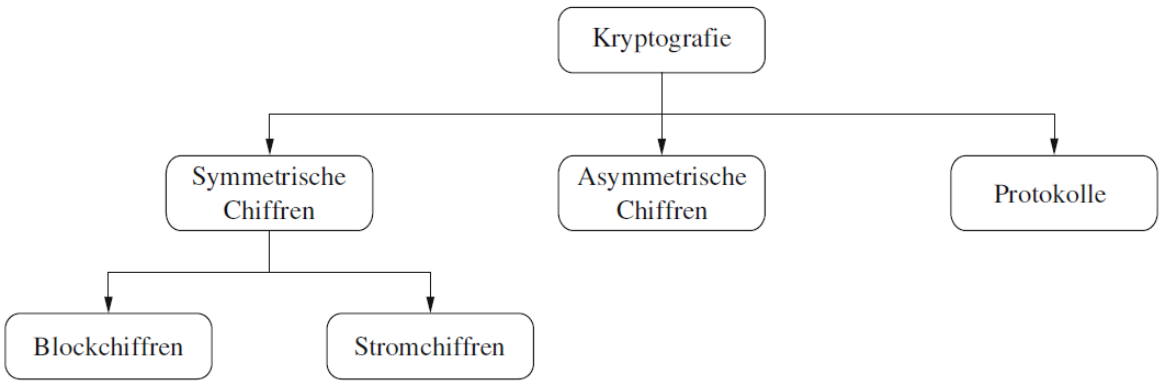

Abbildung_02: „Aufbau der Kryptografie“

Die Kryptografie stellt dem Anwender mehrere Verschlüsselungsverfahren zur Verfügung. Die Hauptgruppen werden unter der Charakteristik der Symmetrie untergeordnet. Diese werden in der Abbildung _01 als Symmetrische und Asymmetrische Chiffren gekennzeichnet.

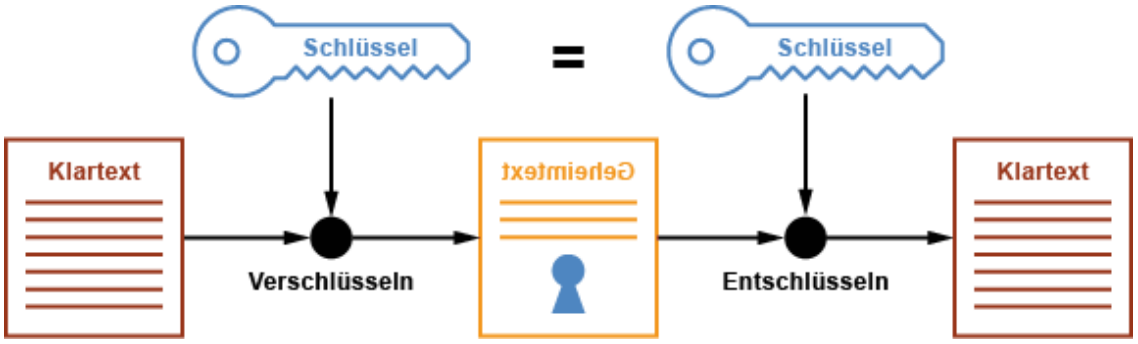

Abbildung_03: „Symmetrische Verschlüsselung“

Symmetrische Verfahren (Secret-Key-Verfahren) werden durch die Nutzung eines für beide Parteien bekannten Schlüssels charakterisiert. Prinzipiell wandelt dabei eine Partei den Klartext mithilfe des Schlüssels in Geheimtext um.

Daraufhin wird durch die gegenüberstehende Partei der Geheimtext mittels gleichen Schlüssels „decrypted“. Dieser Schlüssel muss von beiden Anwendern akzeptiert und angenommen werden. Prinzipiell bilden symmetrische Verfahren einen festen Bestandteil in meisten Kryptosystemen. Beispielen begegnet man im Alltag unter Verschlüsselungen von Dateien oder Laufwerken. Symmetrische Chiffren sind primär unter Block- und Stromchiffren gegliedert:

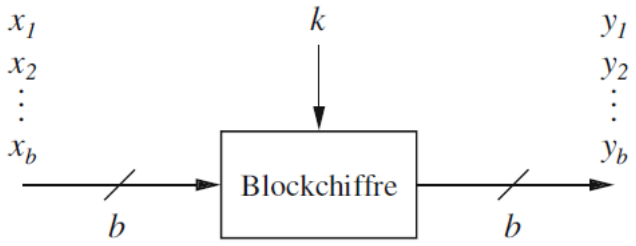

Abbildung_04: „Blockchiffren“

Die Blockchiffre verschlüsselt einen Block simultan mit dem gleichen Schlüssel. Während dessen kann jedes einzelne Bit auf die Verschlüsselung eines anderen im Block vorhandenen Teilnehmer haben. Wenn die Bitanzahl nicht einen ganzen Block füllen kann, wird dieser mit dem „Padding“ gefüllt. Die Blockbreite unterscheidet sich von 128 Bit (16 Byte) bis 64 Bit (8 Byte). Im Allgemeinen kann man die Blockchiffre in verschiedene Modi betreiben. Ein Beispiel stellt das ECB dar, welches jeden Block mit dem gleichen Schlüssel verschlüsselt. Dieser Betriebsmodus birgt das Risiko, dass bei gleichem Klartext gleiche Ergebnisse ausgegeben werden. Angreifer können durch Mustererkennung auf den Inhalt schließen.

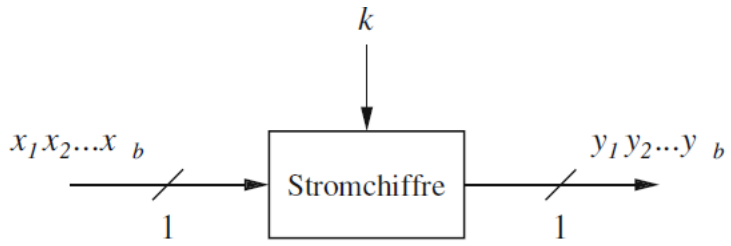

Abbildung_05: „Stromchiffren“

Die Stromchiffre verschlüsselt im Gegensatz zur Blockchiffre jedes Bit einzeln. Prinzipiell wird ein Schlüssel generiert und mit dem Klartext XOR-verknüpft.

Die generierte Folge wird als „Schlüsselstrom“ bezeichnet. Um den erzeugten Geheimtext zu entschlüsseln muss eine weitere XOR-Verknüpfung erfolgen. Der Schlüsselstrom muss dabei der Länge des Klartextes entsprechen. Bei der Stromchiffrierung kann man zwischen synchronen und asynchronen Systemen differenzieren. Synchrone Stromchiffren benutzen Schlüsselströme, welche nur von dem eigentlichen Schlüssel abhängig sind. Asynchrone hingegen beziehen sich auf den „Geheimtext“ und passen sich dementsprechend an. Stromchiffren sind klein und schnell, da der Schlüsselstrom vorausberechnet und zwischengespeichert werden kann. Zudem wird durch einen Bitfehler im Geheimtext mithilfe des bitweisen Verschlüsselns lediglich ein Bitfehler im Klartext angezeigt. Somit bieten sich Anwendungen mit geringerer Rechenleistung an. Nachteilig ist der Aufwand fürs Erstellen des Schlüsselstroms. Des Weiteren kann nicht einzeln entschlüsselt werden und der Anwender ist gezwungen beim Anwenden der Chiffre den ganzen Geheimtext zu entschlüsseln. Abschließend ist die Länge des Schlüsselstroms zu erwähnen, welche vor dem Verschlüsseln bekannt sein muss so wie die Vielfalt der vorhandenen Schlüssel.

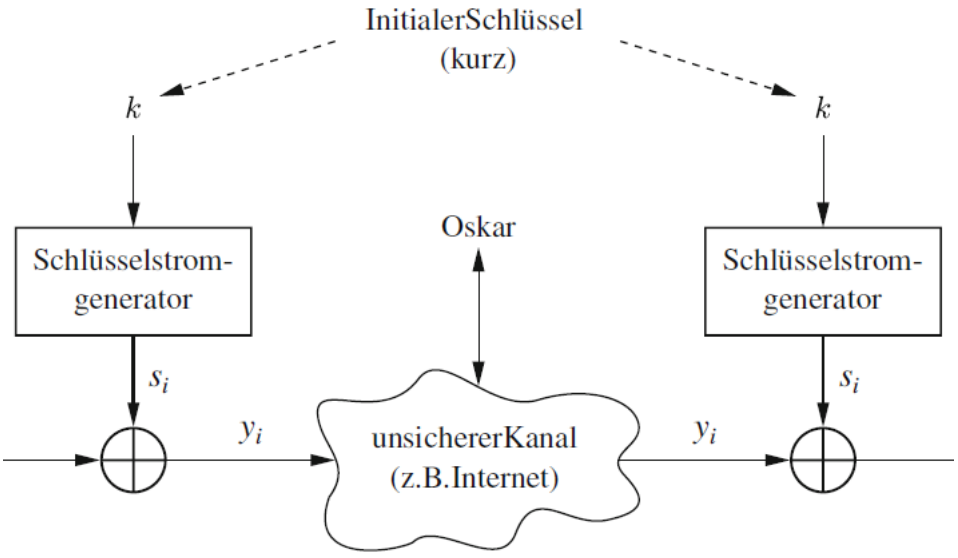

Abbildung_06: „Stromchiffren mit Zufallszahlen“

Um einen sicheren Schlüssel zu generieren wird in der Technik auf Zufallszahlen gesetzt. Diese bilden den Schlüsselstrom und müssen identisch an Sender und Empfänger anliegen, damit eine Ver- und Entschlüsselung erfolgen kann.

Vereinfacht lässt sich dies durch zwei identische zueinander komplementäre Zufallsgeneratoren bewerkstelligen. Diese müssen jeweils mit gleichem Startwert initialisiert werden, um eine identische Zufallsfolge zu produzieren. Die Hauptrolle des Schlüssels wird hierbei durch den Startwert symbolisiert.

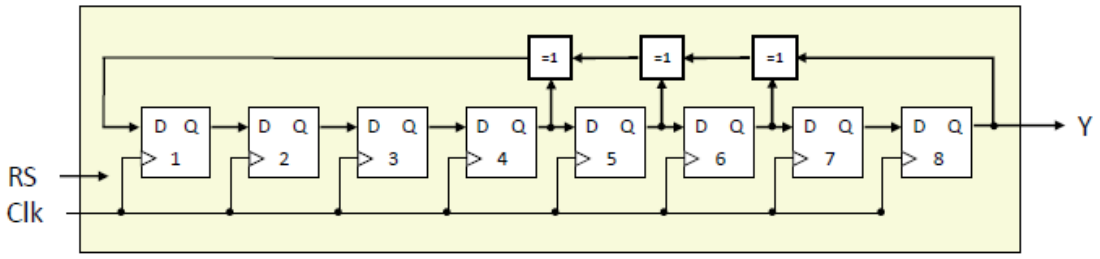

Abbildung_07: „Zufallsgenerator“

Der Zufallsgenerator ist mithilfe eines rückgekoppelten Schieberegisters realisierbar. Dieser besteht aus der jeweiligen Wortbreite (in diesem Beispiel 8-Bit). Hierbei werden die rückgekoppelten Signale Bit 8, Bit 6, Bit 5 und Bit 4 XOR-verknüpft. Damit der Generator anfängt Zufallszahlen zu produzieren müssen eine Clock und ein Startwert initialisiert sein. Durch den Reset kann man die Register auf den Initialzustand (d.h. Startwert) zurücksetzen.

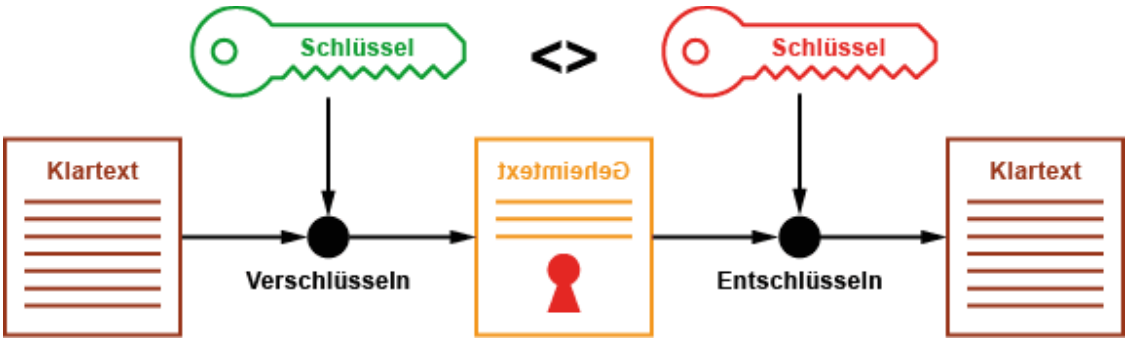

Abbildung_08: „Asymmetrische Verschlüsselung“

Asymmetrische Verschlüsselungen (Public-Key-Verfahren) werden mit zwei Schlüsseln betrieben. Der Schlüssel zum Verschlüsseln (Public-Key) ist dabei öffentlich und der Komplementäre (Private-Key) geheim. Die Schlüsselpaare müssen während dem Entschlüsseln bekannt sein.

Symmetrische Verfahren dominieren im Vergleich zu Asymmetrischen heutzutage den Markt. Dies liegt an der Komplexität der Asymmetrischen Verfahren, wodurch Implementierungsfehler entstehen können. Zudem können Asymmetrische Verschlüsselungsverfahren, welche auf Primfaktorenzerlegung und der Berechnung diskreter Logarithmen beruhen, durch einfache Quantencomputern gebrochen werden. Prinzipiell bieten diese dementsprechend mehr Angriffsfläche und Ansätze für Sicherheitslücken.

Die Effizienz der ausgewählten Chiffrierung ist in Soft- und Hardware unterschiedlich aufgefasst, wodurch in der Software wenige Taktzyklen bzw. CPU-Befehle für die Verschlüsselung eines Bits entscheidend sind und in der Hardware die benutzte Chipfläche für die logischen Gatter Bedeutung haben.

4. Vorüberlegung und Aufbau der Stromchiffre

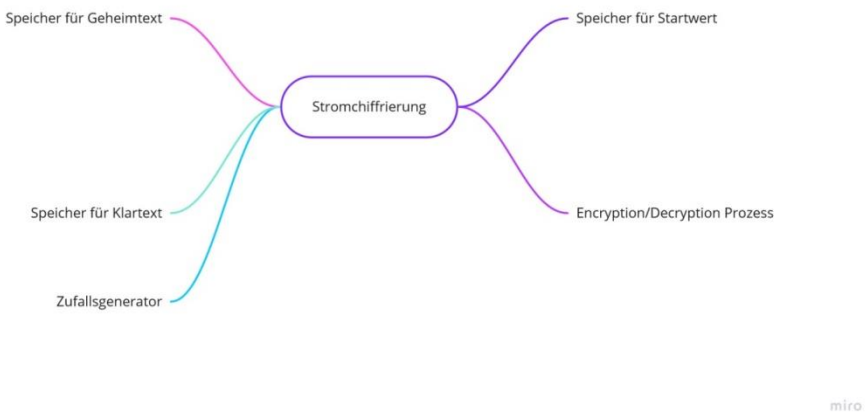

Abbildung_09: „Requirements für die Stromchiffrierung“

Die Stromchiffrierung muss verschiedene grundlegende Bausteine besitzen, um die Funktion zu garantieren: Einen Zufallsgenerator, einen Speicher für den Startwert, einen Speicher für den Klartext, einen Speicher für den Geheimtext und den Kern der Stromchiffrierung, den Hauptprozess der Ver- und Entschlüsselung.

Der Speicher für den Startwert beinhaltet den manuell eingespeicherten 8-Bit Wert, welcher den Zufallsgenerator aktiviert. Der Zufallsgenerator ist ein 8-Bit rückgekoppelter Schieberegister, welcher mithilfe von XOR-Verknüpfungen an den Ausgängen von den einzelnen FlipFlops arbeitet und somit aus dem Startwert die Zufallszahlen bzw den Schlüsselstrom generiert. Der Zufallsgenerator sollte mit Hilfe einer Clock synchronisiert und eines Resets auf den Initialzustand zurückgesetzt werden. Der Initialzustand ist der anliegende Startwert, welcher durch dessen Speicher am Eingang des Generators ständig anliegt. Vom Zufallsgenerator wird nun der Schlüsselstrom zur Verfügung gestellt, um den Verschlüsselungsvorgang zu starten. Der Speicher des Klartextes stellt uns nun den zu verschlüsselnden Text, welcher nun bitweise mithilfe der Verschlüsselungsfunktion verschlüsselt wird, zur Verfügung. Für Testversuche kann dieser Klartext manuell in den Speicher eingetragen werden, um sicherzustellen, dass alle Zeichen auch richtig ver-/entschlüsselt werden. Der Encryption/Decryption Prozess stellt die Hauptfunktion der Stromchiffre dar und ist folgend erklärt: Nachdem der Schlüsselstrom vom Zufallsgenerator erzeugt wurde steht dieser zum Verschlüsseln zur Verfügung. Anschließend wird der Klartext aus dem jeweiligen Speicher bitweise entzogen und mithilfe der Zufallsfolge durch eine einfache XOR-Verknüpfung verschlüsselt. Danach wird der Geheimtext in einen Speicher abgelegt. Sofern der Prozess der Verschlüsselung beendet wurde, folgt ununterbrochen die Entschlüsselung. Hierfür wird der Geheimtext dem „Zwischenspeicher“ entzogen und wieder bitweise mit dem Schlüssel XOR-verknüpft. Die Schwierigkeit dessen bestand darin, dass ein identischer Schlüssel anliegen musste. Dies wurde durch den Reset des Zufallsgenerators erreicht, da dadurch wieder der Initialzustand (Startwert) anliegt und somit der gleiche Schlüssel wieder kreiert wird. Während dem Prozesses der Decryption wird bitweise gleichzeitig der entschlüsselte Geheimtext, also der Klartext, in einen weiteren Speicherbereich abgelegt. Eine erfolgreiche Stromchiffrierung ist implementiert, wenn der entschlüsselte Klartext und der manuell eingespeicherte Klartext identisch sind. Dies kann mithilfe der Rückgabe eines definierten Wertes signalisiert werden. Wichtig ist es bei der Implementierung die Taktabhängigkeit zu betrachten, da bei Verzögerung eines Eingangssignals es zum nicht gewünschten Ergebnis führt.

5. Implementierung und Verifikation

Um die einzelnen verwendeten Komponenten darzustellen und eine Übersicht über die ganzen Prozesse zu bekommen wurden alle Bausteine und Funktionalitäten unter der hierarchisch übergeordneten Entiy „electricity cipher“ zusammengefasst. Die erwähnten Bauteile und Prozesse finden alle parallel und bitweise statt.

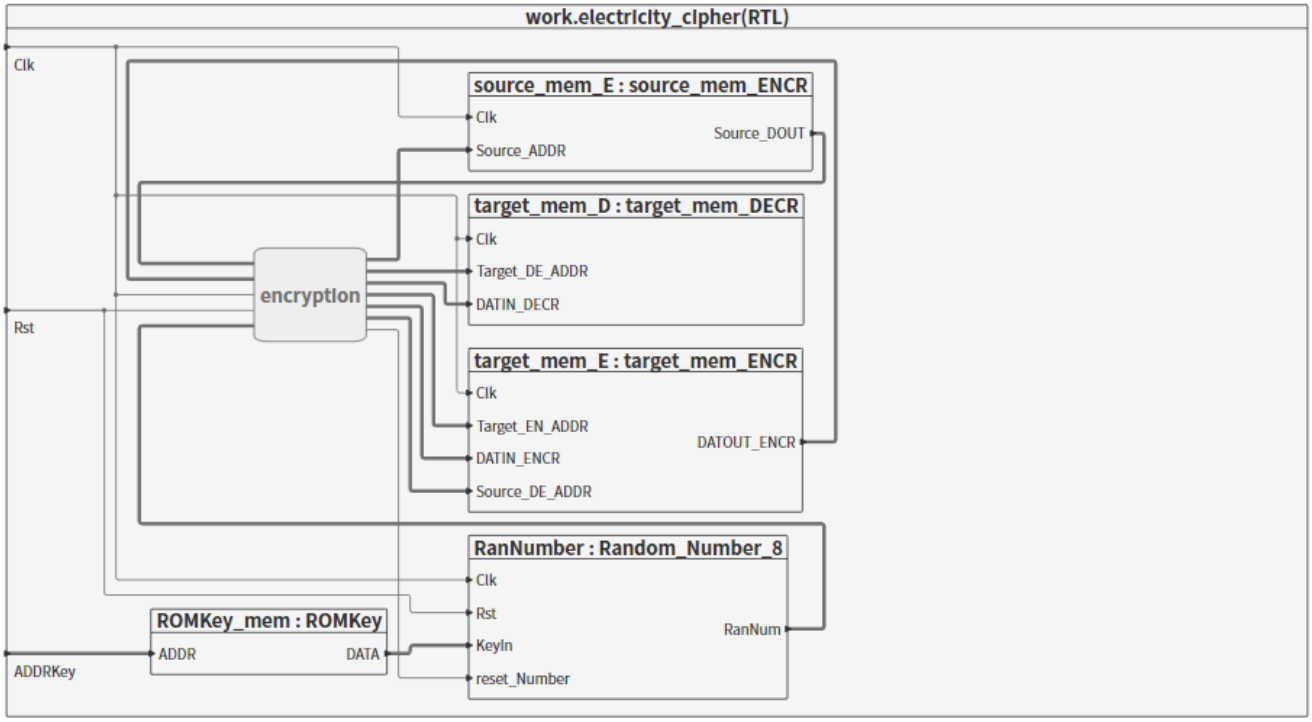

Abbildung_10: „Überblick Aufbau der Stromchiffrierung“

Die Abbildung zeigt, dass der ganze Prozess der Stromchiffrierung primär von drei Eingangssignal abhängig ist: Einem Clock-, einem Reset- und ein ADDRKey-Signal. Die interne Beschaltung funktioniert automatisch, sofern diese Signale anliegen. Als erstes wird durch den ADDRKey die Adresse des Startwertes aus der ROMKey, welche verschiedene Werte gespeichert hat, herausgesucht und als KeyIn bzw. DATA ausgegeben. Der ROMKey ist lediglich für die Versorgung des Startwerts zuständig.

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
entity ROMKey is
generic (
    L_BITS : natural; -- Addressbreite
    M_BITS : natural); -- Wortbreite
port (
    ADDR : in std_logic_vector(L_BITS-1 downto 0);
    DATA : out unsigned(M_BITS-1 downto 0));
end ROMKey;

architecture RTL of ROMKey is

type ROM_array is array(0 to 2**L_BITS - 1) of unsigned(M_BITS - 1 downto 0);

signal memory : ROM_array := ( "00101101" , "00101110" ,"00101100" ,"10101100" ,
    "01101101" , "00101111" , "00100100" , "00100011" , "11111100" , "01101101" ,
     "00101100" , "00101100" , "00101100" , "00101100" , "00101100" , "00111111",
     others => "00000001");

begin
DATA <= memory(to_integer(unsigned(ADDR)));
end RTL;
```

Abbildung_11: „ROM für den Startwert“

Der ROMKey ist mittels eines Arrays aufgebaut und kann in dem vorliegenden Fall zwischen 16 unterschiedlichen Startwerten differenzieren. Falls eine ungültige Adresse eingegeben wurde wird diese mittels der Zahlenfolge „00000001“ signalisiert. Der Zufallszahlengenerator erhält über das Signal Data seinen Startwert. Hierbei wird der zugehörige Wert der Adresse aus dem Array diesem zugewiesen.

```
entity Random_Number_8 is
    port(
        Clk : in std_logic;
        Rst : in std_logic;
        KeyIn : in unsigned (7 downto 0);
        reset_Number : in std_logic;
        RanNum : out unsigned (8 downto 1)
    );
end entity Random_Number_8;

architecture RTL of Random_Number_8 is
    signal RanNumReg : unsigned (8 downto 1);
begin
    process (Clk, Rst) begin
        if(Rst = '0') then
            RanNumReg <= KeyIn;
        elsif(reset_Number = '1')then
            RanNumReg <= KeyIn;
        elsif rising_edge(Clk) then
            for ii in 8 downto 2 loop
                RanNumReg(ii) <= RanNumReg(ii-1);
            end loop;
            RanNumReg(1) <= RanNumReg(8) XOR RanNumReg(6) XOR RanNumReg(5) XOR RanNumReg(4);
        end if;
    end process;
    RanNum <= RanNumReg;
end architecture RTL;
```

Abbildung_12: „Zufallszahlengenerator“

Der Baustein Random_Number_8 fungiert als Zufallszahlengenerator und generiert nun mithilfe des Startwertes von der ROMKey den Stromschlüssel. Eingänge des Zahlengenerators sind die Clock, der Reset, der Startwert, so wie der interne Reset. Der interne Reset muss existieren, da der Prozess sowohl die Ver- und Entschlüsselung automatisch durchführen muss. Damit bei der Entschlüsselung der gleiche Schlüssel vorliegt soll der Reset den Zahlengenerators auf seinen Initialzustand (Startwert) zurücksetzen. Dadurch wird eine identische Zufallszahlenfolge erzeugt, sofern der Startwert am ROMKey der gleiche ist. Der Zufallsgenerator ist mithilfe eines „process“, der Clock und Reset (low-active) abhängig ist, realisiert. Falls der Reset betätigt wird, stellt sich der Initialzustand ein und in den Schieberegistern steht der Startwert. Dasselbe geschieht, wenn der interne Reset (high-active) aktiviert ist und somit gleichzeitig bekannt ist, dass der Decryption-Vorgang begonnen hat. Wenn angenommen der Takt nun eine „rising-edge“ hat, kommen die 8 Schieberegister zum Einsatz und anschließend erfolgt die XOR-Verknüpfung der Bit 8, 6, 5, 4, welche in das erste Bit eingeschrieben werden und somit sich die erste Zufallszahl bilden kann. Der Output wird als „RanNum“ an den Hauptprozess der En-/Decryption weitergegeben.

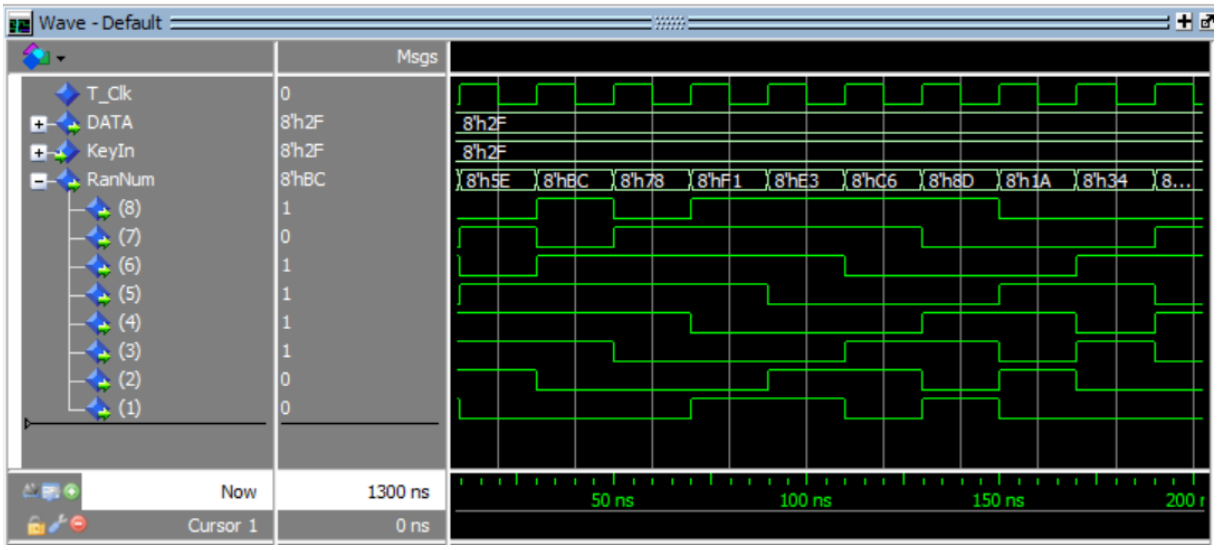

Abbildung_13: „Verifikation Zufallsgenerator“

In der „Verifikation Random_Number_8“ wird aufgezeigt, dass nach jeder positiven Taktflanke eine unterschiedlich unabhängige Zufallszahl ausgegeben wird.

Zudem ist zu erkennen, dass der Startwert stets konstant bleibt, wodurch man beim Reset des Generators eine identische Abfolge kreiert. Abschließend ist zu sagen, dass die Periodendauer der T_Clk von 25 ns genau eingehalten wurden und somit ein Zufallsgenerator erschaffen wurde, welcher erfolgreich einen Stromschlüssel generiert.

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity source_mem_ENCR is
    port (Clk    : in  std_logic;
          Source_ADDR : in  unsigned(7 downto 0);
          Source_DOUT : out unsigned(7 downto 0));
end source_mem_ENCR;

architecture RTL of source_mem_ENCR is


 type ROM_array is array(0 to 63) of unsigned (7 downto 0);

 -- Beispieltext
 signal memory_source_ENCR : ROM_array   := (

 1 => (to_unsigned(character'pos('A'), 8)), 2 => (to_unsigned(character'pos('A'), 8)), 3 => (to_unsigned(character'pos('A'), 8)),
 4 => (to_unsigned(character'pos('A'), 8)), 5 => (to_unsigned(character'pos('A'), 8)), 6 => (to_unsigned(character'pos(' '), 8)),
 7 => (to_unsigned(character'pos('A'), 8)), 8 => (to_unsigned(character'pos('A'), 8)), 9 => (to_unsigned(character'pos('A'), 8)),
 10 => (to_unsigned(character'pos('A'), 8)), 11 => (to_unsigned(character'pos('A'), 8)), 12 => (to_unsigned(character'pos('A'), 8)),

 others => (to_unsigned(character'pos('A'), 8)));

 begin
  process (CLK) begin

       Source_DOUT <= memory_source_ENCR(to_integer(Source_ADDR));

  end process;
end RTL;
```

Abbildung_14: „ROM für Klartext“

Der Klartext ist in einem Speicherbaustein ROM_array hinterlegt, welcher durch ein 64-Bit Array realisiert wurde. Hierbei wurden lediglich zum Testen die ersten zwölf Zeichen mit dem Buchstaben A belegt. Der buchstabenweise eingegebene Quelltext wird hier jeweils als Asciisymbol verwertet, und somit die zugehörige Asciicodierung als Wert in das Array eingespeichert. Diese Asciicodierung ist in der später folgenden Verifikation als Hexadezimalzahl wiederzufinden. Die Eingangssignale sind eine Clock und das Source_ADDR, welches die Funktion hat bitweise den Klartext von Speicherplatz 1 bis 64 hinzuweisen. Das Source_DOUT dient hierbei als Ausgabewert des jeweilig eingespeicherten Klartextbits.

```
if(Source_EN_ADDR < "00111111") then
for i in 0 to 63 loop
       if rising_edge(Clk) then
         DOut <= RanNum xor DReg;
         Source_EN_ADDR <= Source_EN_ADDR + 1;
         Target_EN_ADDR <= Target_EN_ADDR + 1;
       end if;
   end loop;
end if;
```

Abbildung_15: „Verschlüsselung des Klartexts“

Mittels dieses Prozesses werden die 64-Bit abgearbeitet, bis es keine zu verschlüsselnden Bits mehr gibt (Grenze ist bei Source_ADDR < „63“). Bei steigender Taktflanke wird die Zufallszahl (Schlüsselstrom) mit Source_DOUT XOR-verknüpft und somit bitweise verschlüsselt. Anschließend wird simultan Source_ADDR und, in die der Geheimtext gespeicherte Adresse, Target_EN_ADDR erhöht.

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity target_mem_ENCR is
    port ( Clk    : in  std_logic;
           Target_EN_ADDR : in  unsigned (7 downto 0);
           DATIN_ENCR : in  unsigned (7 downto 0);
           DATOUT_ENCR : out unsigned (7 downto 0);
           Source_DE_ADDR : in  unsigned(7 downto 0));
end target_mem_ENCR;

architecture RTL of target_mem_ENCR is


 type RAM_array is array(0 to 63) of unsigned(7 downto 0);

signal memory_target_ENCR : RAM_array  := (others => "00000000");

 begin
  process (Clk) begin
    if(rising_edge(Clk))then
      memory_target_ENCR(to_integer(Target_EN_ADDR)) <= DATIN_ENCR;
      DATOUT_ENCR <=  memory_target_ENCR(to_integer(Source_DE_ADDR));
    end if;
  end process;
end RTL;
```

Abbildung_16: „Zielspeicher für den Geheimtext“

Der Speicherbaustein target_mem_ENR hinterlegt den Geheimtext. Die Eingangssignale sind die Clock, in die zu speichernde Adresse, der Dateninhalt aus der Verschlüsselung und die Source_DE_ADDR, welche grundsätzlich die gleiche Funktionalität wie die Source_ADDR aufweist.

An dem Ausgangssignal DATOUT_ENCR liegt der zu entschlüsselnde Geheimtext an. Der Aufbau ähnelt der des Speicherbausteins des Klartextes, da im Endeffekt die gleiche Funktion erfüllt werden soll.

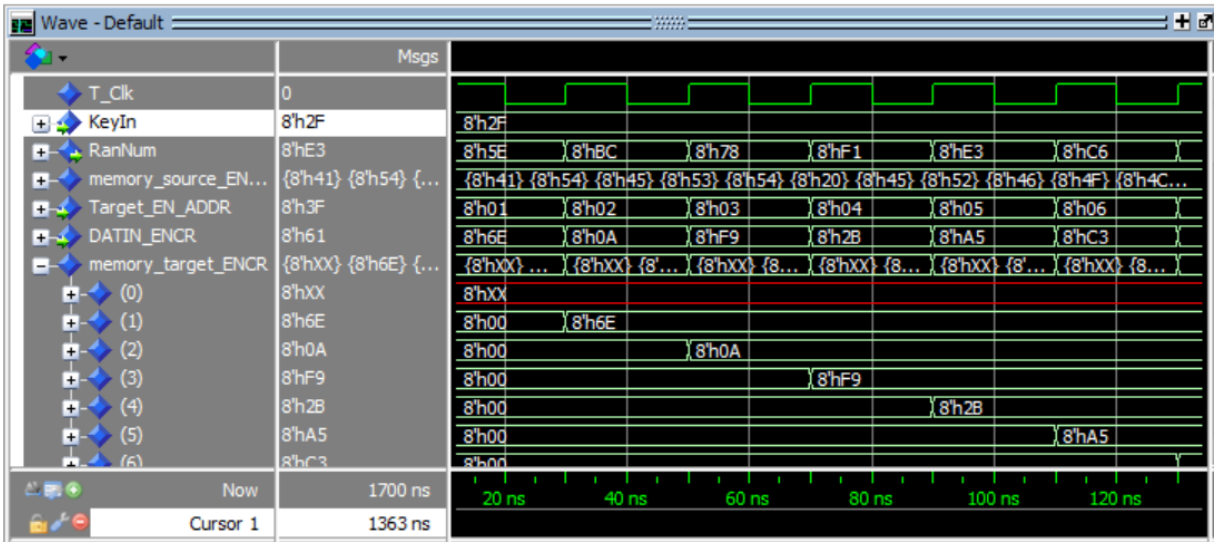

Abbildung_17: „Verifikation der Verschlüsselung“

In der Verifikation wird gezeigt, dass die Zufallszahlen erfolgreich mit dem Klartext verschlüsselt werden. Zu Beginn bestand der Klartext, welcher zu verschlüsseln war, lediglich aus einer Reihe an „A“s, welche als Asciicodierung die Hexadezimalzahl 41 ergibt. Zur Variation wurde in einem bestimmten Abschnitt des Klartextes (Adresse 1 bis 12) eine andere Zeichenfolge hinterlegt. Beispielhaft kann man den ersten Wert, der Adresse 0, des Schlüsselstroms entnehmen und mittels des ersten Klartext-Bits XOR-verknüpfen: „2F“ und „41“ ergeben in der Berechnung die Hexadezimalzahl „6E“. Diese wird nun in die target_memory_ENCR an erster Stelle abgespeichert (30ns). Intern wird nun der Geheimtext als DATOUT_ENCR zum Entschlüsseln freigegeben. Die Source_DE_ADDR fungiert als Quelltextadresse für die Entschlüsselung. Der Prozess der Entschlüsselung nutzt nun den ehemaligen Zielspeicher der Verschlüsselung als neuen Quellspeicher. Nun erfolgt ein komplementärer Vorgang, welcher 64-Bit abarbeitet. Der entschlüsselte Klartext wird in den letzten Speicherbaustein, den target_mem_DECR, abgelegt. Dieser wird mit synchron zu allen anderen Bauteilen geschalten. Der Target_DE_ADDR zeigt auf die jeweilige Speicheradresse in dem Baustein und der DATIN_DECR beinhaltet den entschlüsselten Klartext.

```
if(Source_EN_ADDR > "00111100") then
reset_Number <= '0';

for i in 0 to 63 loop
       if rising_edge(Clk) then
         DATOUT_DECR <= RanNum xor DATOUT;
         Target_DE_ADDR <= Target_DE_ADDR + 1;
         Source_DE_ADDR <= Source_DE_ADDR + 1;
         end if;
   end loop;
end if;
 if(Source_EN_ADDR = "00111110") then
    reset_Number <= '1';
end if;
```

Abbildung_18: „Entschlüsselung des Geheimtexts“

Der Entschlüsselungsprozess wird gestartet, wenn das drittletzte Bit verschlüsselt ist. Dies hat den Grund, da durch die Verarbeitung der Bits eine Taktverzögerung von drei Taktflanken entsteht. Wenn die Source_ADDR beim drittletzten Bit ist muss der interne Reset am Zufallsgenerator aktiv sein, um die gleiche Zahlenfolge zu garantieren. Der interne Reset muss auf inaktiv gestellt werden, damit beim Entschlüsseln nicht als Schlüsselstrom lediglich der Startwert anliegt. Der „reset_Number“ muss nicht unbedingt nach dem drittletzten Bit aktiv sein, sondern kann beliebig später aktiviert werden. Jedoch muss dabei ebenfalls auf den Start der Entschlüsselung geachtet werden. Wichtig ist dabei nur, dass man ihn lediglich einmalig aktiviert. Der „loop“ ist dem Anfänglichen identisch und trägt die Werte in den letzten Speicherbaustein ein.

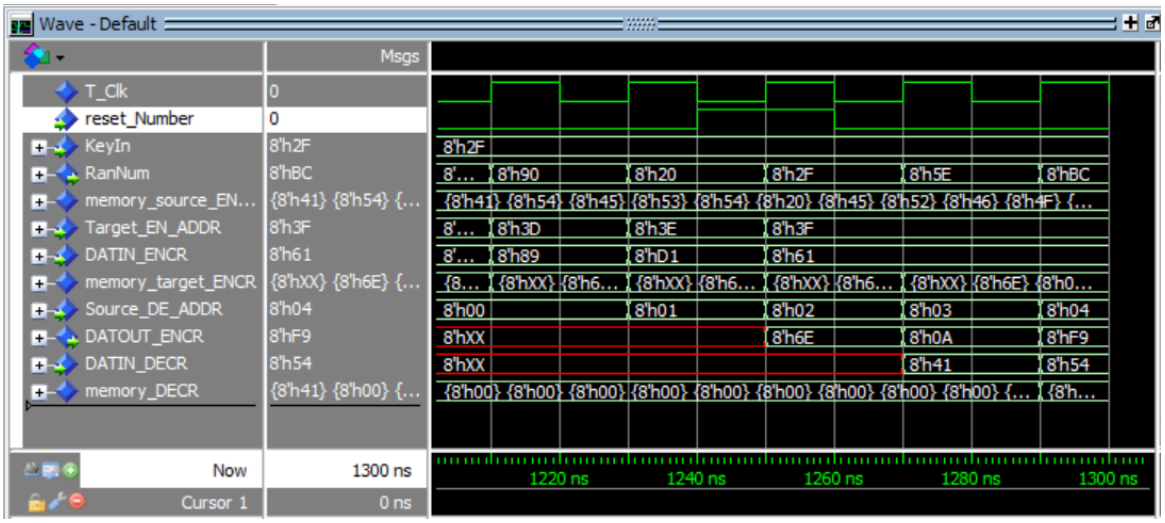

Abbildung_19: „Verifikation des internen Resets“

Aus der Verifikation ist gut ersichtlich, dass nach dem internen Reset (1240ns) der Initialzustand des Zahlengenerators übernommen (1250ns) und ein identischer Schlüsselstrom für die Entschlüsselung erzeugt wird. Anschließend ist mit dem Signal Datin_Decr die Abspeicherung in den Zielspeicher des Entschlüsselungsprozesses aufgezeigt.

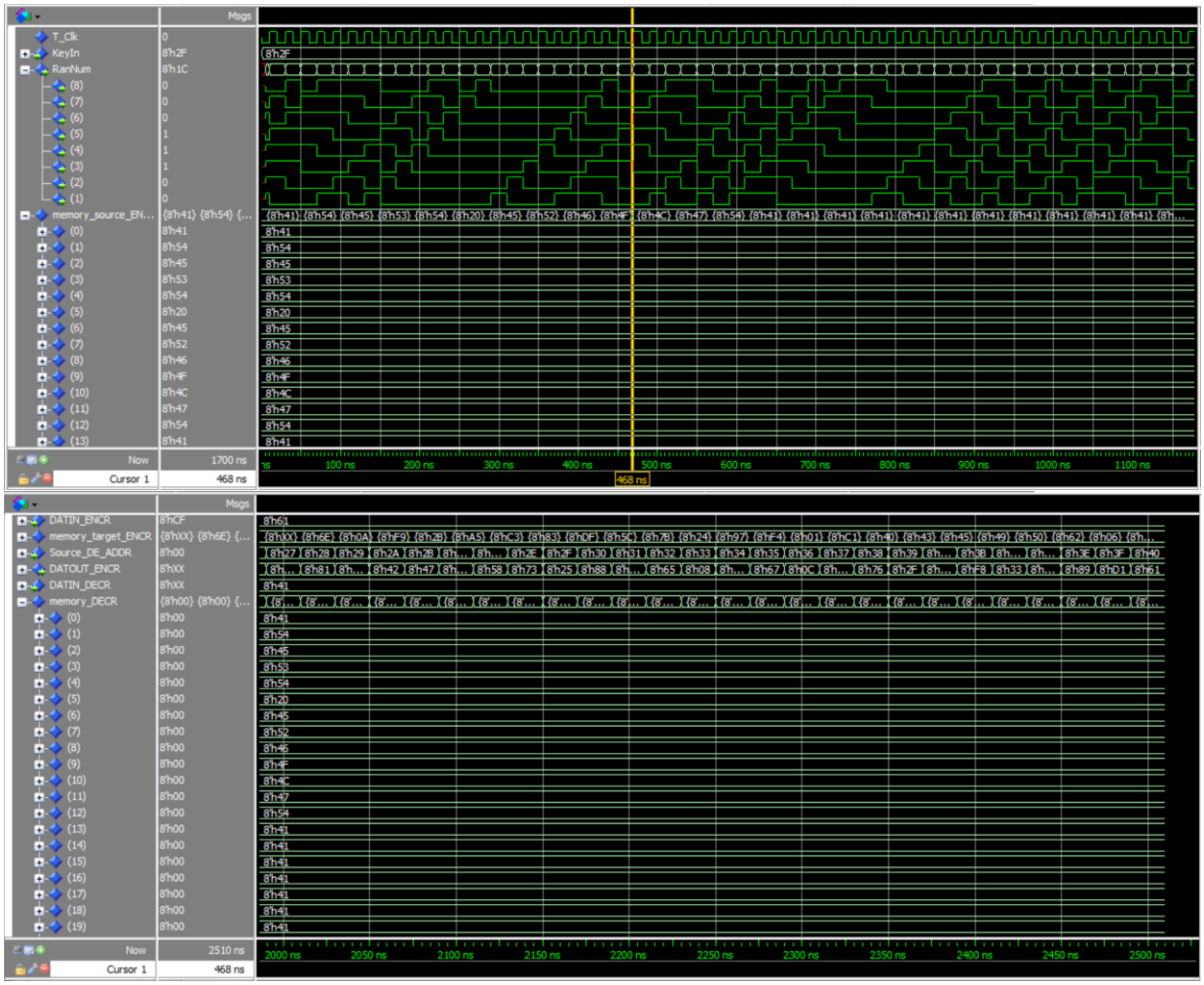

Abbildung_20: „Verifikation Klartext entspricht entschlüsseltem Klartext“

Die Simulation zeigt in der ersten Abbildung den eingespeicherten Klartext. Um zu überprüfen, ob die entschlüsselnden Daten den Klartext in der Source entsprechen, wurde als Test der Satz „ATEST ERFOLGTAAAA…“ ver- und entschlüsselt. Ersichtlich ist, dass im memory_DECR der identische Dateninhalt vorliegt und somit garantiert werden kann, dass die Stromchiffrierung erfolgreich durchgeführt werden kann.

6. Fazit

Die Stromchiffrierung konnte erfolgreich mittels des FPGA-Boards Basys 3 durchgeführt werden. Die Fehlerquellen umfassen mehrere Bereiche, welche in der Projektphase zu Hindernissen geführt haben: Zum einen haben die Prozesse der Ver- und Entschlüsselung gegenseitig interferiert und somit ein paralleles Abarbeiten des Klar- und Geheimtextes nicht ermöglicht. Aufgrund dessen kam es zu Teilüberlappungen, welche teilweise schon im Simulationsprogramm zu Endlosschleifen geführt haben. Des Weiteren war man gezwungen eine sequentielle Verarbeitungsweise auszuwählen, welche zu Taktverzögerungen geführt hat.

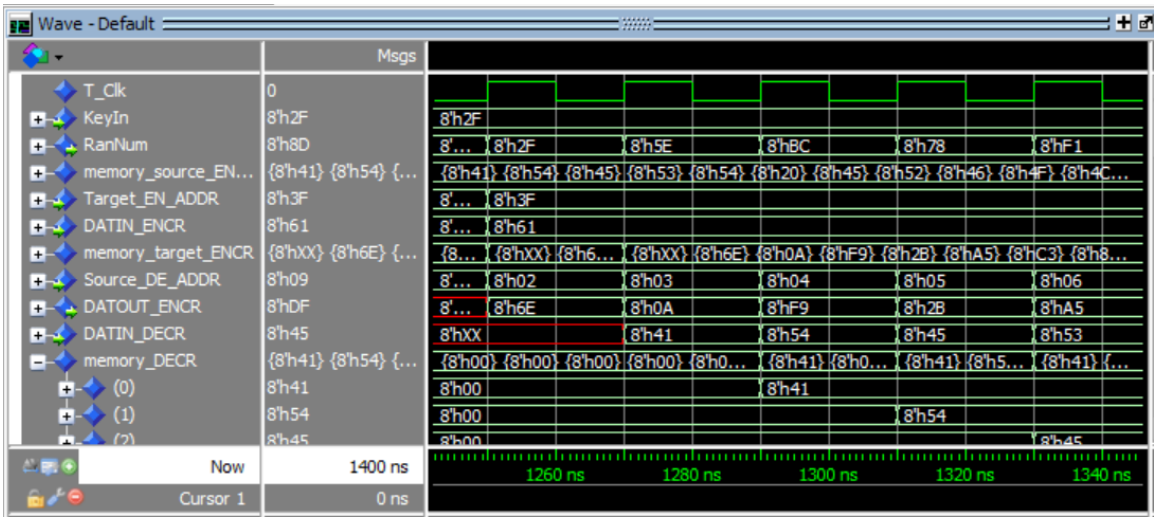

Abbildung_21: „Taktverzögerung durch sequentielle Prozesse“

In der Simulation ist ersichtlich, dass das Holen, Ver- oder Entschlüsseln und Abspeichern der Bits jeweils ein Takt erfordert. Beispielsweise wird hier die Zufallszahl „2F“(hex) mit dem verschlüsselten Geheimtextbyte „6E“ entschlüsselt (1270ns) und anschließend abgespeichert (1290ns). Durch diese Zeitverzögerung wird man, nicht nur vom Prozess aus gesehen verlangsamt, sondern auch das Timing für den internen Reset wird hierdurch zusätzlich erschwert. Sobald der Prozess der Entschlüsselung beginnt, müssen für die jeweiligen Werte, welche zu entschlüsselnd sind, die passenden Zufallszahlen anliegen. Kommt es hierbei zu einer nicht einbezogenen Verzögerung und es liegen nichtmehr die zugehörigen Werte zum Entschlüsseln an, schlägt die Entschlüsselung komplett fehl. Eine Erweiterungsmöglichkeit ist es die beiden Prozesse der Ver- und Entschlüsselung teilweise parallel ablaufen zu lassen, was den Gesamtprozess beschleunigen würde.

7. Quellen

[1] Vertiefung FPGA Skript von Klaus Gosger „VFS_Charts 2020_09_09“

[2] „https://www.electronicdesign.com/technologies/embedded-revolution/article/21127827/maxim-integrated-cryptography-why-do-we-need-it“

[3] „https://www.hs-osnabrueck.de/fileadmin/HSOS/Forschung/Recherche/Laboreinrichtungen_und_

Versuchsbetriebe/Labor_fuer_Digital_und_Mikroprozessortechnik/Buch/VHDL-UEbersicht.pdf“

[4] „https://tams.informatik.uni-hamburg.de/paper/2001/SA_Witt_Hartmann/cdrom/Text/Arbeit.pdf“

[5] „https://www.elektronik-kompendium.de/sites/net/1907041.htm“

[6] „https://www.elektronik-kompendium.de/sites/net/1910101.htm“

[7] „https://www.elektronik-kompendium.de/sites/net/1910111.htm“

[8] „https://www.elektronik-kompendium.de/sites/net/1911011.htm“

[9] „https://www.elektronik-kompendium.de/sites/net/1911041.htm“

[10] „Kryptografie verständlich“ von Chirstof Paar, Jan Pelzl [Buch]

[11] „Einführung in die Kryptografie 6.Auflage“ von Johannes Buchmann [Buch]

[12] „Kryptografie und IT-Sicherheit“ von Joachim Swoboda, Stephan Spitz, Michael Pramateftakis [Buch]