**ОПИСАНИЕ АЛГОРИТМА КУЗНЕЧИК С РЕАЛИЗАЦИЕЙ НА ПЛИС**

**Шифрование**

Алгоритм реализован автоматным способом на языке Verilog в САПР ISE 14.7. При проектировании использовались такие информационные источники, как официальная документация ГОСТ[1]. Контрольные примеры взяты там же. Подробное описание алгоритма можно прочитать в статье на сайте - Криптографический алгоритм «Кузнечик» (ГОСТ Р 34.12-2015)[2]. Так же были разработаны свои таблицы для упрощения вычислений на аппаратном уровне. Также для лучшего понимания работы алгоритма можно обратиться к готовому коду, написанному на языке программирования python который можно посмотреть на github[3], но будьте внимательны, не весь данный код работает правильно. Из данной работы можно заимствовать логику L и S преобразований. Правильность можно проверить с помощью контрольных примеров на сайте официальной документации алгоритма.

Теперь к самому автомату. Он состоит из модуля S преобразования на 1 байт (он же модуль нелинейного преобразования), который преобразовывает поступающее число согласно таблице. Данную таблицу вы можете найти на сайте - Криптографический алгоритм «Кузнечик» (ГОСТ Р 34.12-2015).

*Таблица 1 – таблица для S преобразования*

|  |
| --- |
||

Немного поясним эту таблицу. Любое число можно представить в данной таблице. Ведь в ней приведены значения по модулю x^8 + x^7 + x^6 + x + 1. Т.е. по модулю 111000011. Реализация на Verilog сделана так, что значение из таблицы отдается мгновенно, когда приходит значение на таблицу. Таких таблиц нужно 16. На каждый байт исходного слова.

Также автомат включает модуль L преобразования. И он уже более интересный. Он включает в себя таблицу L преобразования… НОООО! Эту таблицу нужно изменить под нужды аппаратной реализации алгоритма. Для сокращения времени поиска необходимо создать 9 разных смещенных таблиц под каждый из коэффициентов 148, 32, 133, 16, 194, 192, 1, 251, 1, 192, 194, 16, 133, 32, 148, 1. Тут стоит объяснить, как вообще работает L преобразование. Сначала на модуль подается слово из 16 байт, каждый байт которого должен быть умножен на соответствующий коэффициент в поле Галуа. Но данный вид умножения является очень сложным и требует дополнительных вычислительных ресурсов и довольно много времени. Поэтому для удобства расчета была создана таблица линейного преобразования.

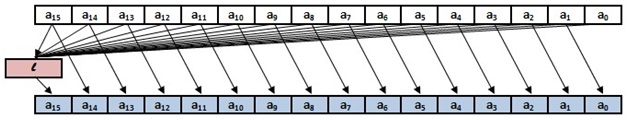
*Таблица 2 – таблица степеней двойки в поле гаула*

|  |
| --- |
| 1 2 4 8 16 32 64 128 195 69 138 215 109 218 119 238  31 62 124 248 51 102 204 91 182 175 157 249 49 98 196 75  150 239 29 58 116 232 19 38 76 152 243 37 74 148 235 21  42 84 168 147 229 9 18 36 72 144 227 5 10 20 40 80  160 131 197 73 146 231 13 26 52 104 208 99 198 79 158 255  61 122 244 43 86 172 155 245 41 82 164 139 213 105 210 103  206 95 190 191 189 185 177 161 129 193 65 130 199 77 154 247  45 90 180 171 149 233 17 34 68 136 211 101 202 87 174 159  253 57 114 228 11 22 44 88 176 163 133 201 81 162 135 205  89 178 167 141 217 113 226 7 14 28 56 112 224 3 6 12  24 48 96 192 67 134 207 93 186 183 173 153 241 33 66 132  203 85 170 151 237 25 50 100 200 83 166 143 221 121 242 39  78 156 251 53 106 212 107 214 111 222 127 254 63 126 252 59  118 236 27 54 108 216 115 230 15 30 60 120 240 35 70 140  219 117 234 23 46 92 184 179 165 137 209 97 194 71 142 223  125 250 55 110 220 123 246 47 94 188 187 181 169 145 225 1 |

К сожалению, нам нужно перестроить ее так, чтобы сразу получать результат умножения на коэффициент поступающего числа. Каждую из таких таблиц пришлось генерировать самостоятельно на языке программирования C++. Код приведен ниже

*Листинг 1 – код для преобразования таблиц, реализованный на С++*

|  |
| --- |
| #include <vector>  #include <iostream>  #include <fstream>  int main()  {  std::ofstream text("text.txt");  std::vector<int> array; //исходная таблица  std::vector<int> massiv;  int elem;  // смещенные таблицы  std::vector<int> massiv\_conv;  array = { 1, 2, 4, 8, 16, 32, 64, 128, 195, 69, 138, 215, 109, 218, 119, 238, 31, 62, 124, 248, 51, 102,  204, 91, 182, 175, 157, 249, 49, 98, 196, 75, 150, 239, 29, 58, 116, 232, 19, 38, 76, 152, 243, 37, 74,  148, 235, 21, 42, 84, 168, 147, 229, 9, 18, 36, 72, 144, 227, 5, 10, 20, 40, 80, 160, 131, 197, 73, 146,  231, 13, 26, 52, 104, 208, 99, 198, 79, 158, 255, 61, 122, 244, 43, 86, 172, 155, 245, 41, 82, 164, 139,  213, 105, 210, 103, 206, 95, 190, 191, 189, 185, 177, 161, 129, 193, 65, 130, 199, 77, 154, 247, 45, 90,  180, 171, 149, 233, 17, 34, 68, 136, 211, 101, 202, 87, 174, 159, 253, 57, 114, 228, 11, 22, 44, 88, 176,  163, 133, 201, 81, 162, 135, 205, 89, 178, 167, 141, 217, 113, 226, 7, 14, 28, 56, 112, 224, 3, 6, 12, 24,  48, 96, 192, 67, 134, 207, 93, 186, 183, 173, 153, 241, 33, 66, 132, 203, 85, 170, 151, 237, 25, 50, 100,  200, 83, 166, 143, 221, 121, 242, 39, 78, 156, 251, 53, 106, 212, 107, 214, 111, 222, 127, 254, 63, 126,  252, 59, 118, 236, 27, 54, 108, 216, 115, 230, 15, 30, 60, 120, 240, 35, 70, 140, 219, 117, 234, 23, 46,  92, 184, 179, 165, 137, 209, 97, 194, 71, 142, 223, 125, 250, 55, 110, 220, 123, 246, 47, 94, 188, 187,  181, 169, 145, 225, 1 };  for (int i = 0; i < array.size(); i++)  {  massiv.push\_back(i);  }  for (int i = 0; i < array.size(); i++)  {  massiv[array[i]] = i;  }  massiv[0] = 0;  // поиск смещения  elem = massiv[148]; // 148 – коэффициент умножения. именно его нужно менять для получения смещенных таблиц    /\*for (int i = 1; i < massiv.size(); i++)  {  std::cout << i << " " << massiv[i] << " " << (massiv[i] + elem) << " " << (massiv[i] + elem) % 255 << " " << array[(massiv[i] + elem) % 255] << "\n";  massiv\_conv.push\_back(array[(massiv[i] + elem) % 255]);  }\*/  for (int i = 0; i < 32 + 1; i++)  {  text << "array[" << i << "] = 8'd" << array[(massiv[i] + elem) % 255] << ";" << "\n";  }  } |

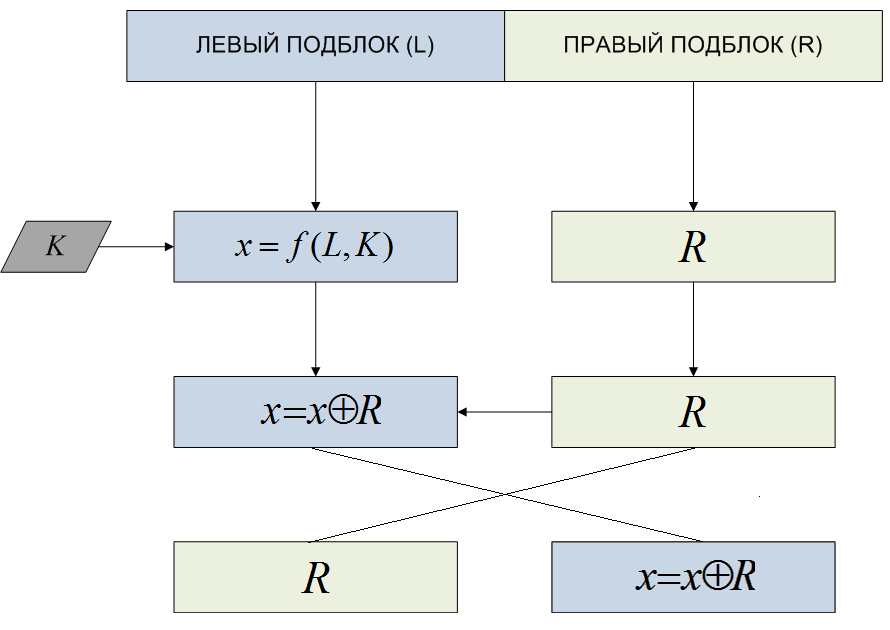
Также необходимо создать текстовый файл, в который будет записана измененная таблица. Таких таблиц необходимо создать 9 (под каждый из коэффициентов). Далее исходное слово сдвигается на 1 байт вправо, а на место левого байта(15 байта в [15:0] регистре) поступает XOR между всеми байтами, поступившими из таблиц L преобразования. 

***Рисунок 1 – L преобразование***

Все вышеперечисленное, а именно: преобразование через таблицы, а так же сдвиг и запись в первый (15 байт) результата XORа между всеми выходами таблиц. Все эти операции должны повториться 16 раз. После чего слово может быть выведено.

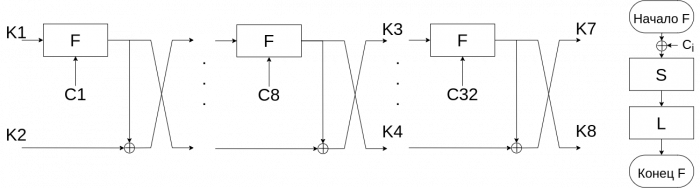
Следующий модуль будет включать в себя оба модуля, описанных раньше. Это будет модуль формирования итерационных ключей. В самом алгоритме шифрования используется как раз не входной ключ, который между прочим будет размером в 32 байта (в два раза больше слов, которые будут кодироваться, и итерационных ключей). Подалгоритм формирования ключей будет строиться на итерациях ячеек фейстеля.

Ячейка Фейстеля



***Рисунок 2 – Ячейка фейстеля***

Первая пара итерационных ключей будет состоять из основного ключа, поделенного пополам. Первый ключ будет равен левой части основного(первой части в 16 байт), второй соответственно второй части(вторые 16 байт). Последующий ключи будут получены после 8 итераций сети фейстеля на каждую пару ключей.



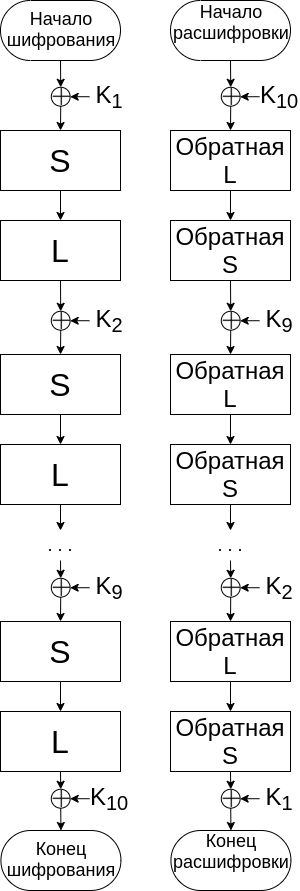
***Рис 3 - Сеть фейстеля***

Картинка не соответствует действительности(хотя основная суть правильная). Ошибка данного рисунка в том, что на вторую итерацию(вниз сети, если смотреть на рисунок 1) идет измененное значение первого ключа, но на вторую итерацию приходят измененный ключ(К2 нижний, идет вверх) и неизмененный(К1 верхний, идет вниз). Вторая ошибка данного рисунка состоит в том, что после итерации с константой C32 будут получены ключи К9 и К10. Также нужно сказать о константах. В некоторых источниках говориться о генерации констант по принципу L преобразовании. Но константы на то и константы, потому что их задают вручную. Константы представлены ниже.

*Таблица 3 – таблица констант*

|  |
| --- |
||

Следующий модуль – шифрование слов. Только в этой части начинается основной этап, который будет шифровать слова и выводить их из ПЛИС. Процедура шифрования слова несколько отличается от процедуры генерации ключей, хоть и очень на нее похожа. Чтобы зашифровать слово, нужно прогнать его через алгоритм, представленный ниже.



***Рисунок 3 – итерации шифрования и дешифрования***

Сам алгоритм шифрования довольно прост, если у вас есть готовые блоки S и L преобразования. Как видно на рисунке, слово должно 9 раз пройти через последовательность XOR с ключом Ki (где i – номер итерации), S-преобразование, L-преобразование. После 9 таких последовательностей необходимо еще раз провести XOR с ключом K10. Это будет концом шифрования слова и его можно будет выводить из ПЛИС.

Верхний модуль данного алгоритма лишь контролирует флаги и соединяет модуль генерации ключа с модулем(модулями) шифрования слов.

Данная реализация способна выдавать 128 битное зашифрованное слово каждые 200 тактов синхросигнала ПЛИС, что соответствует 64 Мбит в секунду, при условии, что частота ПЛИС равна 100МГц.

При необходимости можно распараллелить алгоритм на ПЛИС. При добавлении дополнительных блоков шифрования слова, скорость передачи данных будет соответствующе возрастать. Идеальным количеством блоков будет 16. Таким образом скорость передачи данных будет равна 1ГБиту в секунду, что подходит подо все используемые интерфейсы ethernet

**Дешифрование**

**Дешифрование происходит почти по тому же принципу, что и шифрование. Стоит сказать, что модуль генерации итерационных ключей и включающиеся в него модули S и L преобразования остаются в проекте неизменными. Теперь поговорим про измененные блоки. А именно блоки S\_revers и L\_revers обратного преобразования. А также модуль шифрования слов.**

**Начнем с модуля S\_revers обратного преобразования, так как он проще и строится на основании таблицы прямого S преобразования. Таблица была сгенерирована и подтверждена на одном из сайтов.**

*Таблица 3 – таблица обратного S преобразования*

|  |
| --- |
| 0xA5, 0x2D, 0x32, 0x8F, 0x0E, 0x30, 0x38, 0xC0, 0x54, 0xE6, 0x9E, 0x39, 0x55, 0x7E, 0x52, 0x91, 0x64, 0x03, 0x57, 0x5A, 0x1C, 0x60, 0x07, 0x18, 0x21, 0x72, 0xA8, 0xD1, 0x29, 0xC6, 0xA4, 0x3F, 0xE0, 0x27, 0x8D, 0x0C, 0x82, 0xEA, 0xAE, 0xB4, 0x9A, 0x63, 0x49, 0xE5, 0x42, 0xE4, 0x15, 0xB7, 0xC8, 0x06, 0x70, 0x9D, 0x41, 0x75, 0x19, 0xC9, 0xAA, 0xFC, 0x4D, 0xBF, 0x2A, 0x73, 0x84, 0xD5, 0xC3, 0xAF, 0x2B, 0x86, 0xA7, 0xB1, 0xB2, 0x5B, 0x46, 0xD3, 0x9F, 0xFD, 0xD4, 0x0F, 0x9C, 0x2F, 0x9B, 0x43, 0xEF, 0xD9, 0x79, 0xB6, 0x53, 0x7F, 0xC1, 0xF0, 0x23, 0xE7, 0x25, 0x5E, 0xB5, 0x1E, 0xA2, 0xDF, 0xA6, 0xFE, 0xAC, 0x22, 0xF9, 0xE2, 0x4A, 0xBC, 0x35, 0xCA, 0xEE, 0x78, 0x05, 0x6B, 0x51, 0xE1, 0x59, 0xA3, 0xF2, 0x71, 0x56, 0x11, 0x6A, 0x89, 0x94, 0x65, 0x8C, 0xBB, 0x77, 0x3C, 0x7B, 0x28, 0xAB, 0xD2, 0x31, 0xDE, 0xC4, 0x5F, 0xCC, 0xCF, 0x76, 0x2C, 0xB8, 0xD8, 0x2E, 0x36, 0xDB, 0x69, 0xB3, 0x14, 0x95, 0xBE, 0x62, 0xA1, 0x3B, 0x16, 0x66, 0xE9, 0x5C, 0x6C, 0x6D, 0xAD, 0x37, 0x61, 0x4B, 0xB9, 0xE3, 0xBA, 0xF1, 0xA0, 0x85, 0x83, 0xDA, 0x47, 0xC5, 0xB0, 0x33, 0xFA, 0x96, 0x6F, 0x6E, 0xC2, 0xF6, 0x50, 0xFF, 0x5D, 0xA9, 0x8E, 0x17, 0x1B, 0x97, 0x7D, 0xEC, 0x58, 0xF7, 0x1F, 0xFB, 0x7C, 0x09, 0x0D, 0x7A, 0x67, 0x45, 0x87, 0xDC, 0xE8, 0x4F, 0x1D, 0x4E, 0x04, 0xEB, 0xF8, 0xF3, 0x3E, 0x3D, 0xBD, 0x8A, 0x88, 0xDD, 0xCD, 0x0B, 0x13, 0x98, 0x02, 0x93, 0x80, 0x90, 0xD0, 0x24, 0x34, 0xCB, 0xED, 0xF4, 0xCE, 0x99, 0x10, 0x44, 0x40, 0x92, 0x3A, 0x01, 0x26, 0x12, 0x1A, 0x48, 0x68, 0xF5, 0x81, 0x8B, 0xC7, 0xD6, 0x20, 0x0A, 0x08, 0x00, 0x4C, 0xD7, 0x74 |

**Модуль L преобразования также претерпел изменения. Хотя таблицы остались такими же, какими и были. Изменения коснулись самого модуля преобразования. Данный модуль специфически изменил работу преобразования. В отличие от модуля прямого преобразования данный модуль сдвигает регистр на байт в лево(в сторону старшего байта). А также байты исходного регистра умножаются на коэффициенты со смещением. XOR происходит так же, как и в прямом преобразовании, но учитывая смещение.**

**Модуль шифрования слов изменен в соответствии с алгоритмом дешифрования слов Рисунок 3. Зашифрованное слово поступает на автомат и проходит через 9 итераций последовательности умножение на ключ(в зависимости от итерации начиная с последнего – 10), L\_revers, S\_revers. После 9 итерации слово XOR-ится с итерационным ключом с номером 1. После этого слово считается расшифрованным.**