

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA KỸ THUẬT MÁY TÍNH



BÁO CÁO ĐỒ ÁN
MÔN: THIẾT KẾ HỆ THỐNG SỐ VỚI HDL
Giảng viên hướng dẫn: Ths. Ngô Hiếu Trường

ĐỀ TÀI:

**DESIGN EXPANDED 128 BIT DATA ENCRYPTION STANDARD
ALGORITHM (DES) USING HDL**

**THIẾT KẾ MỞ RỘNG THUẬT TOÁN MÃ HÓA
DỮ LIỆU DES 128 BIT BẰNG HDL**

Sinh viên thực hiện:

Nguyễn Đình Anh - 23520057

Nguyễn Hoàng Quốc Cường - 23520200

Bùi Tân Đạt - 23520244

Nguyễn Phạm Thiên Ân - 23520015

Lớp: CE213.Q12

LỜI CẢM ƠN

Trong suốt quá trình hoàn thành bài báo cáo môn **THIẾT KẾ HỆ THỐNG SỐ VỚI HDL**:

Chúng em xin chân thành gửi lời cảm ơn đến Trường Đại học Công nghệ Thông tin – Đại học Quốc gia TP.HCM đã tạo điều kiện thuận lợi với hệ thống cơ sở vật chất hiện đại và thư viện phong phú. Những nguồn tài liệu quý giá từ thư viện đã hỗ trợ chúng em rất nhiều trong việc tìm kiếm thông tin phục vụ cho bài báo cáo của mình.

Đặc biệt, chúng em xin gửi lời tri ân sâu sắc nhất đến thầy **Ngô Hiếu Trường** – giảng viên bộ môn, người đã tận tâm giảng dạy và hướng dẫn chúng em trong suốt quá trình học tập. Sự chia sẻ kiến thức quý báu và những bài học bổ ích từ thầy đã trang bị cho chúng em những kỹ năng cần thiết, giúp bài báo cáo này trở nên hoàn thiện hơn.

Nhóm chúng em cũng xin gửi lời cảm ơn đến các thành viên trong nhóm đã cùng nhau nỗ lực và hoàn thành bài báo cáo này. Chúng em đã cùng nhau học hỏi, trao đổi và phối hợp nhịp nhàng để hoàn thành bài báo cáo này một cách tốt nhất. Mỗi thành viên trong nhóm đều đã đóng góp những ý tưởng và kiến thức của mình để làm cho bài báo cáo thêm hoàn thiện.

Xin gửi lời cảm ơn đến các bạn đã quan tâm và theo dõi bài báo cáo của nhóm. Nhóm chúng em hy vọng bài báo cáo này sẽ mang lại những thông tin hữu ích cho các bạn.

Chúng em cũng ý thức rằng do còn thiếu kinh nghiệm và một số hạn chế về kiến thức, bài báo cáo này có thể chưa hoàn hảo. Rất mong nhận được những đóng góp ý kiến và nhận xét từ thầy, để chúng em có thể hoàn thiện hơn trong tương lai.

Cuối cùng, chúng em xin kính chúc thầy **Ngô Hiếu Trường** sức khỏe, hạnh phúc và thành công trong sự nghiệp giảng dạy cũng như trong cuộc sống!

Xin chân thành cảm ơn !

Thành phố Hồ Chí Minh, ngày 4 tháng 12 năm 2025

NHẬN XÉT CỦA GIẢNG VIÊN

PHÂN CÔNG NHIỆM VỤ

HỌ VÀ TÊN	MSSV	NHIỆM VỤ	MỨC ĐỘ HOÀN THÀNH
Nguyễn Đình Anh	23520057	Phân nhiệm vụ, làm phần roundkey, tổng hợp code, viết báo cáo chương 1, 3 ,4. Chính sửa báo cáo.	100%
Nguyễn Hoàng Quốc Cường	23520200	F Function, slide báo cáo, viết báo cáo chương 2	100%
Bùi Tân Đạt	23520244	Làm phần I_P, State machine, tạo test case, viết báo cáo chương 2, chỉnh sửa báo cáo.	100%
Nguyễn Phạm Thiên Ân	23520015	Làm phần I_P, Round key, tạo test case, Viết báo cáo chương 2	100%

MỤC LỤC

CHƯƠNG I: TỔNG QUAN VỀ ĐỀ TÀI	7
I. Giới thiệu đề tài.....	7
II. Mục tiêu nghiên cứu.....	7
III. Giới hạn đề tài.....	7
CHƯƠNG 2: THIẾT KẾ CHI TIẾT	8
I. Tổng quan về thiết kế (cấu trúc và sơ đồ khối thiết kế)	8
II. Chi tiết thiết kế của các khối.	10
1. Hoán vị khởi tạo IP.	10
2. Hàm mã hóa khóa F(R,K)	14
3. Khóa vòng	21
4. Hoán vị khởi tạo đảo IP_1	32
5. State machine	35
CHƯƠNG 3: KIỂM TRA THIẾT KẾ VÀ MÔ PHỎNG	40
I. Kiểm tra kết quả và mô phỏng chức năng.....	40
1. Pre-simulation.	40
2. Post-simulation.....	42
II. Số lượng chu kì.	43
III. Coverage code.....	44
1 Total coverage code.	44
2. Coverage code của từng file.....	44
CHƯƠNG 4: TỔNG KẾT VÀ ĐÁNH GIÁ QUÁ TRÌNH THỰC HIỆN	49
I. Tổng kết.	49
II. Quá trình thực hiện của nhóm.	49
1. Phân công nhiệm vụ	49
2. Quá trình thực hiện.....	49
CHƯƠNG 5: TÀI LIỆU THAM KHẢO	50

DANH MỤC BẢNG BIỂU VÀ HÌNH ẢNH

Hình 1: Lưu đồ thuật toán	8
Hình 2: Sơ đồ khối của thiết kế	9
Hình 3: Hoán vị khởi tạo IP	10
Hình 4: Hoán vị khởi tạo IP sau khi trừ 1	11
Hình 5: Sơ đồ lưu trữ trong Registers	12
Hình 6: Expansion Table.....	14
Hình 7: Substitution Box (Sbox).....	15
Hình 8: Minh họa sử dụng chuỗi 6bit để tìm giá trị trong bảng Sbox	15
Hình 9: Permutation Box	16
Hình 10: Chi tiết sơ đồ khối F(R,K).....	16
Hình 11: Expansion Table được điều chỉnh theo input của khối	17
Hình 12: Hiện thực code Expansion Table	18
Hình 13: Hiện thực code Sbox	19
Hình 14: Permutation Box dựa trên input của khối	20
Hình 15: Hiện thực Verilog Permutation Box	20
Hình 16: Hiện thực hàm f(R,K)	21
Hình 17: Sơ đồ thuật toán tính khóa vòng	22
Hình 18: Bảng hoán vị lựa chọn 1.....	23
Hình 19: Bảng hoán vị lựa chọn 1 dạng 0-127	23
Hình 20: Hoán vị lựa chọn 1 sau khi hiệu chỉnh.....	24
Hình 21: Bảng hoán vị lựa chọn 2.....	25
Hình 22: Bảng hoán vị lựa chọn 2 dạng 0-127	25
Hình 23: Bảng hoán vị lựa chọn 2 sau khi hiệu chỉnh	25
Hình 24: Sơ đồ khối kiến trúc phần cứng của Round Key.....	27
Hình 25: Hoán vị khởi tạo đảo IP^{-1}	33
Hình 26: Hoán vị khởi tạo đảo IP^{-1} sau khi trừ 1.....	34
Hình 27: Sơ đồ lưu trữ trong Registers	34
Hình 28: STATE ACTION TABLE	36
Hình 29: Giản đồ chuyển trạng thái	37
Hình 30: Kết quả tính toán lý thuyết của thuật toán.....	40
Hình 31: Các test case.	41
Hình 32: Kết quả mô phỏng tất cả test case.	42
Hình 33: Kết quả mô phỏng test case đầu tiên.....	42
Hình 34: Kết quả mô phỏng một số test case.....	43
Hình 35: Kết quả mô phỏng test case đầu tiên.....	43
Hình 36: Tổng coverage code là 97,28 %	44
Hình 37: Coverage code của từng file.....	48
Hình 38: Tổng kết nhiệm vụ	49

CHƯƠNG I: TỔNG QUAN VỀ ĐỀ TÀI

I. Giới thiệu đề tài

Trong thế giới hiện đại ngày nay, việc lưu trữ và mã hóa dữ liệu là điều hết sức quan trọng, nó có ảnh hưởng lớn đến sự tồn tại của 1 hệ thống hay 1 công ty và luôn được cập nhật liên tục. Hiện nay có rất nhiều thuật toán phức tạp và bảo mật để thực hiện được yêu cầu trên. Một trong số những thuật toán lâu đời và có độ bảo mật cao vào thời điểm những năm 90 đó là DES. Đây là một thuật toán mã hóa khối, nó xử lý từng khối thông tin của bản rõ có độ dài xác định và biến đổi theo những quá trình phức tạp để thành khối thông tin bản mã có độ dài không thay đổi. Trong trường hợp của DES, độ dài mỗi khối là 64 bit. DES cũng sử dụng khóa để cá biệt hóa quá trình biến đổi. Nhờ vậy chỉ khi biết khóa mới có thể giải mã được văn bản mã.

Đề tài trên được thực hiện dựa trên tiêu chuẩn FIPS 46 -3 , là tiêu chuẩn mã hóa được ban hành bởi cơ quan an ninh quốc gia Hoa kỳ CIA vào năm 1999, quy định về DES và triple data encryption algorithm (TDEA). Tiêu chuẩn này quy định các thuận toán mã hóa và các biện pháp bảo vệ thông tin, bao gồm việc mã hóa dữ liệu trong quá trình truyền tải và lưu trữ. Với bản tiêu chuẩn là 56 bit độ dài khóa, 8 bit còn lại dùng cho việc kiểm tra, thì nó được xem là không an toàn và thường được ít sử dụng trong các hệ thống hiện đại.

Việc sử dụng ngôn ngữ mô tả phần cứng verilog đạt hiệu quả hơn rất nhiều so với ngôn ngữ lập trình bình thường vì các thao tác được thực hiện một cách song song thay vì tuân tự.

II. Mục tiêu nghiên cứu

Vì những giới hạn về khóa và bảo mật kém đó, nhóm mình đã đề xuất phiên bản mở rộng với phiên bản 128 bit key và dữ liệu giúp tăng đáng kể không gian khóa và khả năng chống tấn công vét cạn. Cụ thể là hướng đến các mục tiêu như sau:

- Phân tích hoạt động của DES và hướng điểm yếu trong cấu trúc
- Cải thiện mức độ an toàn thông qua việc sử dụng nhiều khóa
- Đề xuất bản mở rộng 128 bits nhằm tăng độ phức tạp trong việc dò tìm khóa, đồng thời duy trì với cấu trúc DES truyền thống.
- Xây dựng chương trình mô phỏng Pre-simulation và Post-Simulation, qua đó đánh giá tính hiệu quả và độ an toàn của phương pháp.
- Hiểu được đầy đủ về tính song song của ngôn ngữ mô tả phần cứng verilog.

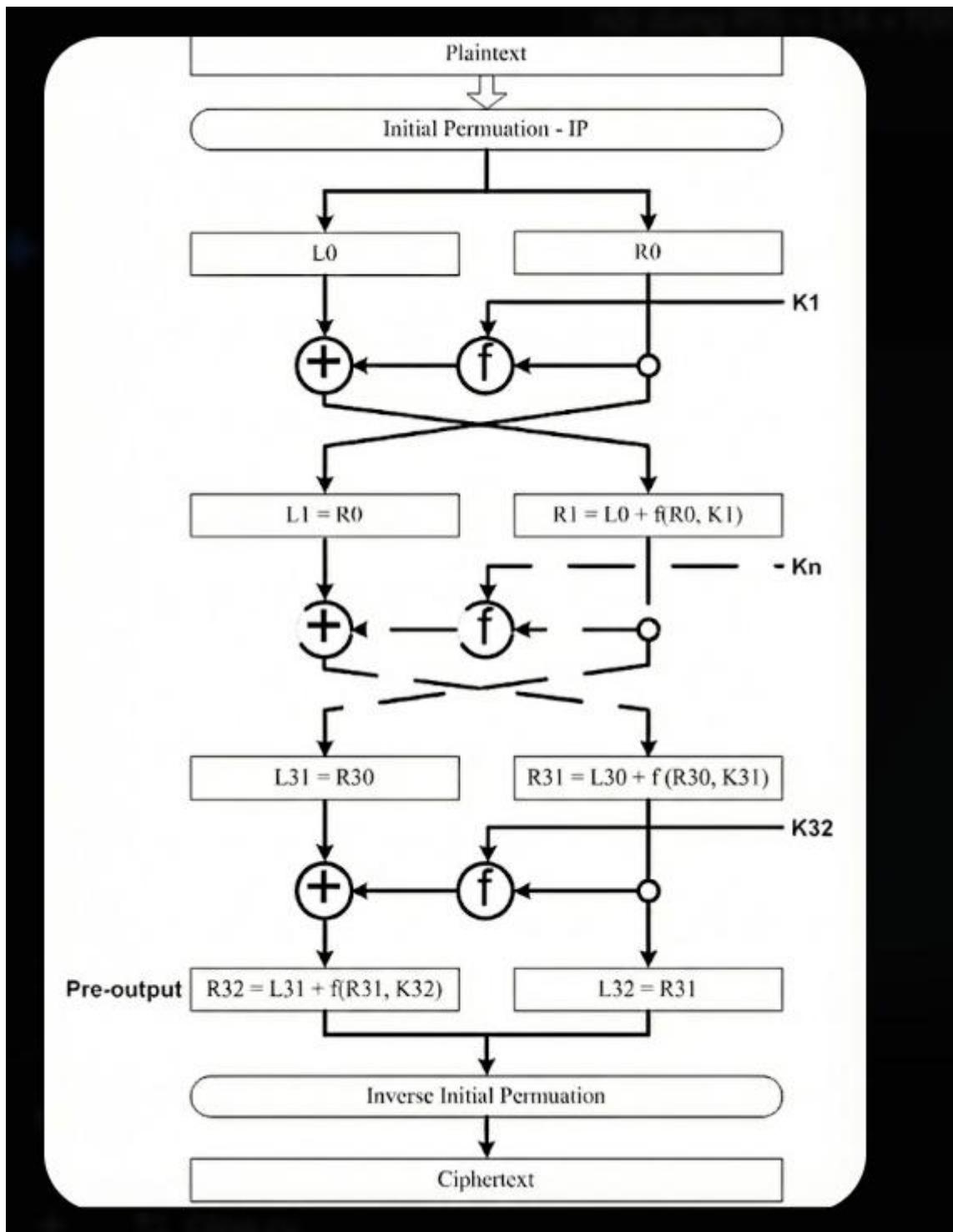
III. Giới hạn đề tài

Mặc dù đề tài tập trung nghiên cứu về thuật toán mã hóa khối, nhưng trong phạm vi thực hiện, một số giới hạn được đặt ra như sau:

- Đề tài chỉ nghiên cứu thuật toán DES và hướng mở rộng khóa, không đi sâu vào các thuật toán mã hóa khối hiện đại như AES hay các thuật toán mã hóa bất đối xứng.
- Việc mở rộng khóa lên 128bit chỉ mang tính nghiên cứu và mô phỏng bằng phần mềm, không nhằm xây dựng một tiêu chuẩn mã hóa mới có thể áp dụng trong công nghiệp.
- Quy mô thử nghiệm chỉ giới hạn trên các tập dữ liệu mẫu, không đánh giá hiệu năng trong môi trường mạng hay hệ thống lớn.
- Các phương pháp tấn công nâng cao như phân tích vi sai (Differential Cryptanalysis), phân tích tuyến tính (Linear Cryptanalysis) chỉ được nhắc đến ở mức lý thuyết mà không triển khai kiểm thử thực tế.

CHƯƠNG 2: THIẾT KẾ CHI TIẾT.

I. Tổng quan về thiết kế (cấu trúc và sơ đồ khối thiết kế)



Hình 1: Lưu đồ thuật toán.

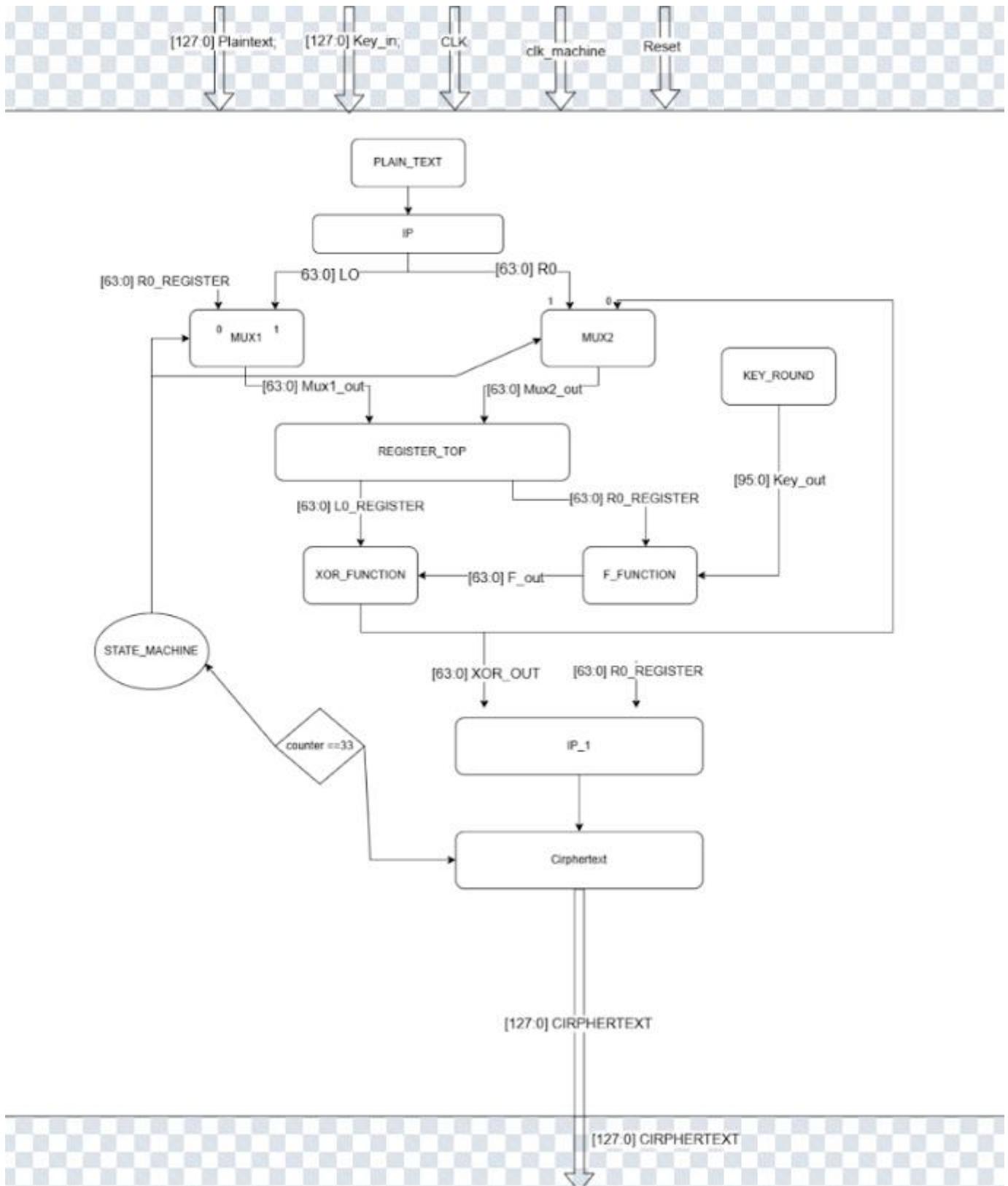
Nguyên tắc hoạt động chính:

- Kích thước khối & khóa : 128 bit
- Số vòng lặp 16 hoặc 32 (ở thiết này nhóm mình làm cả 16 và 32 vòng lặp)
- S Boxes: 16 khối (giữ nguyên 8 cũ và thêm 8 khối mới)
- Hoán vị ban đầu \rightarrow chia thành hai khối L₀, R₀ với mỗi khối 64 bit.
- Hàm h: Mở rộng R thành 96-bit, XOR với khóa vòng, thay thế qua S-box, hoán vị P, XOR với L, sau đó hoán đổi L và R.

- Hoán vị nghịch (IP^{-1}): Tạo bản mã cuối

Sinh khóa vòng:

- PC-1: Giảm từ 128-bit xuống 112-bit (loại bỏ bit chẵn lẻ).
- Chia thành C và D (mỗi 56-bit), xoay trái theo bảng dịch vòng.
- PC-2: Nén xuống 96-bit cho mỗi vòng.
- Lặp lại cho 32 vòng để tạo $K_1 \rightarrow K_{32}$.



Hình 2: Sơ đồ khói của thiết kế

II. Chi tiết thiết kế của các khối.

1. Hoán vị khởi tạo IP.

1.1 Tổng quan cách hoạt động

- Khối Hoán vị khởi tạo (Initial Permutation - IP) là bước xử lý đầu tiên đối với dữ liệu đầu vào. Dữ liệu đưa vào khối này là chuỗi **Plaintext 128 bit**. Mục tiêu của khối IP là xáo trộn vị trí các bit của Plaintext theo một quy tắc cố định trước khi dữ liệu đi vào các vòng lặp mã hóa chính.
- Quy tắc hoán vị được xác định cụ thể trong **Bảng 1.1** dưới đây. Bảng này bao gồm 128 giá trị số nguyên, biểu diễn vị trí của bit trong chuỗi dữ liệu gốc

Bảng 22									(Cột 8)
Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6	Cột 7			
Nửa đầu									
58	50	42	34	26	18	10			2
60	52	44	36	28	20	12			4
62	54	46	38	30	22	14			6
64	56	48	40	32	24	16			8
57	49	41	33	25	17	9			1
59	51	43	35	27	19	11			3
61	53	45	37	29	21	13			5
63	55	47	39	31	23	15			7
Nửa sau									
122	114	106	98	90	82	74			66
124	116	108	100	92	84	76			68
126	118	110	102	94	86	78			70
128	120	112	104	96	88	80			72
121	113	105	97	89	81	73			65
123	115	107	99	91	83	75			67
125	117	109	101	93	85	77			69
127	119	111	103	95	87	79			71

Hình 3: Hoán vị khởi tạo IP

- Đặc tả khối:

- Input: Chuỗi Plaintext 128 bit.
- Output: Chuỗi dữ liệu sau hoán vị 128 bit.
- Nguyên tắc: Mỗi ô trong bảng tương ứng với một vị trí của chuỗi đầu ra. Giá trị nằm trong ô đó chỉ định chỉ số (index) của bit trong chuỗi đầu vào cần được đưa vào vị trí này.

1.2 Chuẩn hóa chỉ số cho hiện thực Verilog

- Trong quá trình hiện thực hóa thuật toán bằng ngôn ngữ mô tả phần cứng Verilog, có sự khác biệt về quy ước đánh số so với mô hình lý thuyết:
 - Thuật toán(lý thuyết): Các bit được đánh số thứ tự từ 1 đến 128.
 - Verilog: Các vector và mảng sử dụng chỉ số bắt đầu từ 0 đến 127.
- Do đó, để đảm bảo tính chính xác khi ánh xạ các chân tín hiệu trong code, ta cần thực hiện bước hiệu chỉnh chỉ số bằng cách trừ đi 1 đơn vị ở tất cả các giá trị trong bảng hoán vị gốc. Kết quả thu được là bảng giá trị như sau:

Bảng_21								
Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6	Cột 7	(Cột 8)	
Nửa đầu (
57	49	41	33	25	17	9	1	
59	51	43	35	27	19	11	3	
61	53	45	37	29	21	13	5	
63	55	47	39	31	23	15	7	
56	48	40	32	24	16	8	0	
58	50	42	34	26	18	10	2	
60	52	44	36	28	20	12	4	
62	54	46	38	30	22	14	6	
Nửa sau								
121	113	105	97	89	81	73	65	
123	115	107	99	91	83	75	67	
125	117	109	101	93	85	77	69	
127	119	111	103	95	87	79	71	
120	112	104	96	88	80	72	64	
122	114	106	98	90	82	74	66	
124	116	108	100	92	84	76	68	
126	118	110	102	94	86	78	70	

Hình 4: Hoán vị khởi tạo IP sau khi trừ 1.

- Sau khi đã hiệu chỉnh chỉ số về dạng 0-127, dữ liệu đầu vào cần được tổ chức lại để phù hợp với cấu trúc của thuật toán. Cụ thể, khối dữ liệu 128bit sẽ được phân chia thành hai phần riêng biệt để nạp vào các thanh ghi đầu(Registers).Bảng dưới đây mô tả chi tiết cách ánh xạ từng vị trí bit từ dữ liệu đầu vào:

Bảng_23											
Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6	Cột 7	(Cột 8)				
Nửa đầu											
6	14	22	30	38	46	54	62				
4	12	20	28	36	44	52	60				
2	10	18	26	34	42	50	58				
0	8	16	24	32	40	48	56				
7	15	23	31	39	47	55	63				
5	13	21	29	37	45	53	61				
3	11	19	27	35	43	51	59				
1	9	17	25	33	41	49	57				
Nửa sau											
70	78	86	94	102	110	118	126				
68	76	84	92	100	108	116	124				
66	74	82	90	98	106	114	122				
64	72	80	88	96	104	112	120				
71	79	87	95	103	111	119	127				
69	77	85	93	101	109	117	125				
67	75	83	91	99	107	115	123				
65	73	81	89	97	105	113	121				

Hình 5: Sơ đồ lưu trữ trong Registers

- **Nửa đầu (64 bit):** Tương ứng với thanh ghi L0 (Left Register).
- **Nửa sau (64 bit):** Tương ứng với thanh ghi R0 (Right Register).

1.3 Hiện thực code

Dựa trên bảng ánh xạ địa chỉ và phân chia dữ liệu đã thiết lập ở các mục 1.2 và 1.3, khối Hoán vị khởi tạo (IP) được hiện thực bằng ngôn ngữ Verilog như sau:

```
module IP(Data_in, Reset, R0, L0);
    input [127:0] Data_in;
    input Reset;
    output [63:0] R0, L0;
    reg [127:0] Data_temp;

    always @(*) begin
        if (Reset)
            Data_temp = 128'b0;
        else begin
            Data_temp = {
                Data_in[6], Data_in[14], Data_in[22], Data_in[30], Data_in[38], Data_in[46], Data_in[54],
                Data_in[62],
```

```

    Data_in[4], Data_in[12], Data_in[20], Data_in[28], Data_in[36], Data_in[44], Data_in[52],
Data_in[60],
    Data_in[2], Data_in[10], Data_in[18], Data_in[26], Data_in[34], Data_in[42], Data_in[50],
Data_in[58],
    Data_in[0], Data_in[8], Data_in[16], Data_in[24], Data_in[32], Data_in[40], Data_in[48],
Data_in[56],
    Data_in[7], Data_in[15], Data_in[23], Data_in[31], Data_in[39], Data_in[47], Data_in[55],
Data_in[63],
    Data_in[5], Data_in[13], Data_in[21], Data_in[29], Data_in[37], Data_in[45], Data_in[53],
Data_in[61],
    Data_in[3], Data_in[11], Data_in[19], Data_in[27], Data_in[35], Data_in[43], Data_in[51],
Data_in[59],
    Data_in[1], Data_in[9], Data_in[17], Data_in[25], Data_in[33], Data_in[41], Data_in[49],
Data_in[57],
    Data_in[70], Data_in[78], Data_in[86], Data_in[94], Data_in[102], Data_in[110],
Data_in[118], Data_in[126],
    Data_in[68], Data_in[76], Data_in[84], Data_in[92], Data_in[100], Data_in[108],
Data_in[116], Data_in[124],
    Data_in[66], Data_in[74], Data_in[82], Data_in[90], Data_in[98], Data_in[106],
Data_in[114], Data_in[122],
    Data_in[64], Data_in[72], Data_in[80], Data_in[88], Data_in[96], Data_in[104],
Data_in[112], Data_in[120],
    Data_in[71], Data_in[79], Data_in[87], Data_in[95], Data_in[103], Data_in[111],
Data_in[119], Data_in[127],
    Data_in[69], Data_in[77], Data_in[85], Data_in[93], Data_in[101], Data_in[109],
Data_in[117], Data_in[125],
    Data_in[67], Data_in[75], Data_in[83], Data_in[91], Data_in[99], Data_in[107],
Data_in[115], Data_in[123],
    Data_in[65], Data_in[73], Data_in[81], Data_in[89], Data_in[97], Data_in[105],
Data_in[113], Data_in[121]
};

end
end

assign L0 = Data_temp[127:64];
assign R0 = Data_temp[63:0];
endmodule

```

Mô tả hiện thực code phía trên:

- Giao diện(Interface)
 - Input: Data_in [127:0] nhận dữ liệu Plaintext đầu vào; Reset dùng để khởi tạo trạng thái.
 - Output: L0 [63:0] và R0 [63:0] là hai khối dữ liệu đầu ra sau hoán vị.
- Logic xử lý:
 - Thứ tự các bit trong mã nguồn tuân thủ chính xác bảng tra cứu chỉ số đã xây dựng tại bảng.
 - Kết quả Data_temp cuối cùng được chia tách: 64 bit trọng số cao được gán cho L0 và 64

bit trọng số thấp được gán cho R0 làm đầu ra output cho bước kế tiếp của thuật toán.

2. Hàm mã hóa khóa F(R,K)

2.1 Tổng quan cách hoạt động

- Đầu tiên 64bit đoạn R được đánh số từ 1 tới 64 từ trái sang phải. Giá trị này sẽ được chuyển đổi qua bảng E (Expansion table) để chuyển thành một giá trị 96bit.

64	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	33
32	33	34	35	36	37
36	37	38	39	40	41
40	41	42	43	44	45
44	45	46	47	48	49
48	49	50	51	52	53
52	53	54	55	56	57
56	57	58	59	60	61
60	61	62	63	64	1

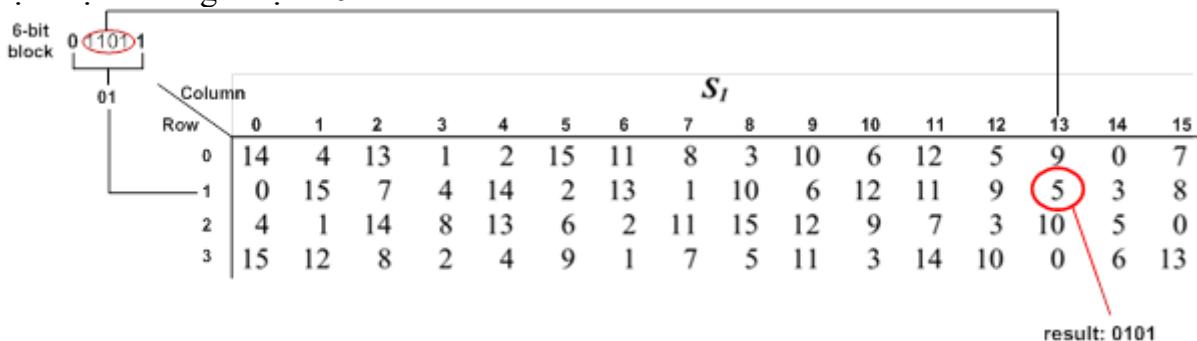
Hình 6: Expansion Table

- Sau đó chuỗi 96bit đó được **XOR** với **Khoá vòng** (phần 3), kết quả của phép XOR được chia làm 16 block được đánh số từ 1 đến 16 theo thứ tự từ trái sang phải mỗi block 6bit và được biến đổi thông qua các hàm lựa chọn riêng biệt là S1, S2, S3, S4, S5, S6, S7, S8,S9, S10, S11, S12, S13, S14, S1 và S16.

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S1/S9	0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
	1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
	2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
	3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S2/S10	0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
	1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
	2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
	3	13	18	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S3/S11	0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
	1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
	2	13	6	4	9	8	15	3	0	11	1	2	12	15	10	14	7
	3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S4/S12	0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
	1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
	2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
	3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S5/S13	0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
	1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
	2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
	3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S6/S14	0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
	1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
	2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
	3	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S7/S15	0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
	1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
	2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
	3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S8/S16	0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
	1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
	2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
	3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Hình 7: Substitution Box (Sbox)

- Sbox ở bản gốc có 8 loại nhưng đối với trường hợp 16 block trên, ta chỉ cần thêm 8 box nữa bằng cách tận dụng lại những Sbox đã có sẵn, VD: nội dung S9 giống S1, S10 giống S2,...
- Như đã nói chúng ta có 6bit ở trong 1 block, cách xác định được giá trị trong bảng bằng cách lấy tổ hợp bit đầu và bit cuối để chọn hàng, 4 bit còn lại để chọn cột. Cứ như vậy với 16 block ta được một chuỗi giá trị có 64bit



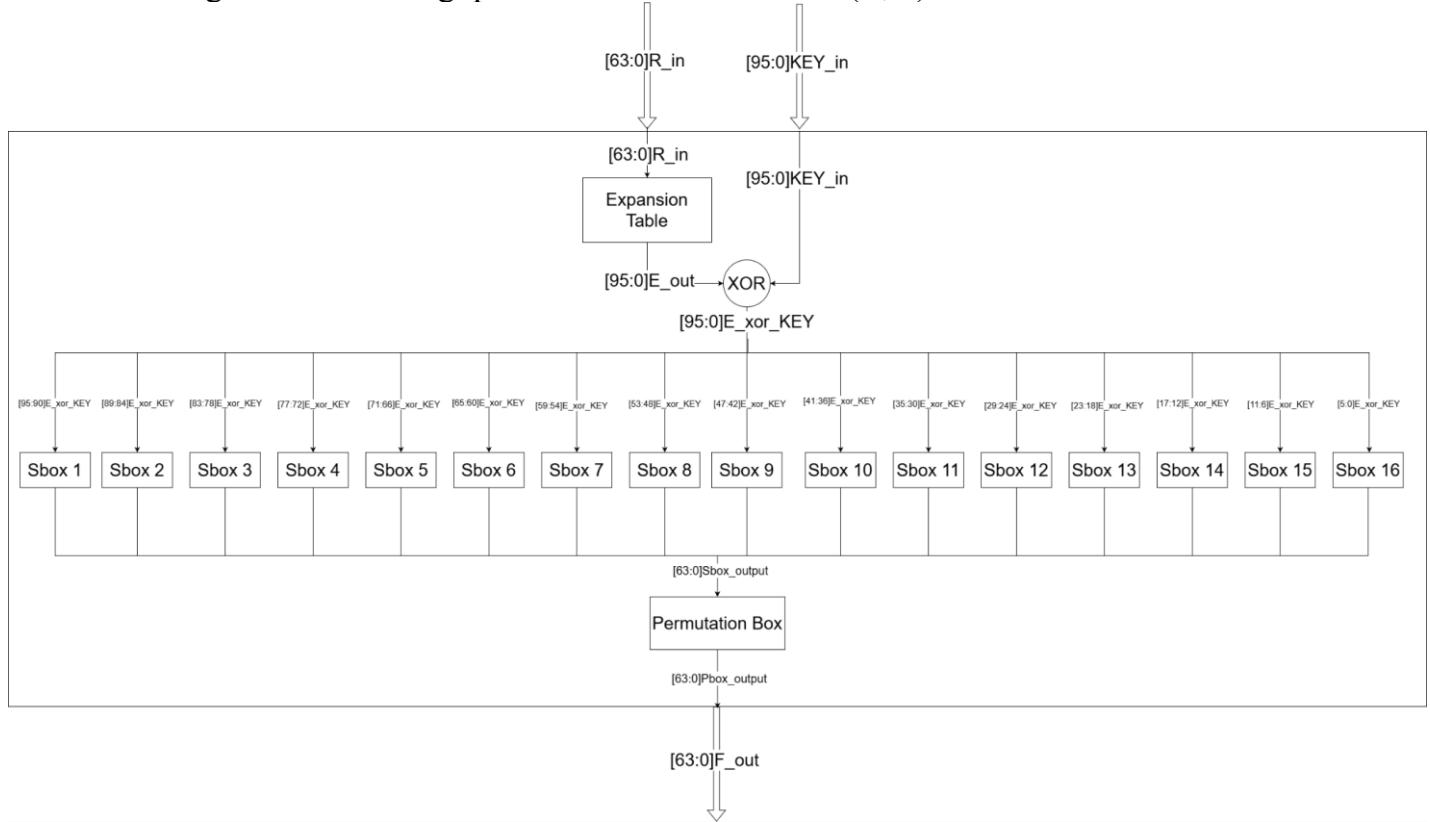
Hình 8.: Minh họa sử dụng chuỗi 6bit để tìm giá trị trong bảng Sbox

- Sau khi có được giá trị 64bit đó, ta tiếp tục đưa cho Pbox (Permutation Box) để tạo ra giá trị hàm f(R,K)

Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6	Cột 7	Cột 8
1->32							
16	7	20	21	29	12	28	17
1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9
19	13	30	6	22	11	4	25
33->64							
48	39	52	53	61	44	60	49
33	47	55	58	37	50	63	42
34	40	56	46	64	59	35	41
51	45	62	38	54	43	36	57

Hình 9: Permutation Box

- Để có thể xử lý được 64bit chúng ta phải mở rộng bảng 32bit cũ như trên.
- Sau khi giá trị 64bit thông qua hàm Pbox ta thu được $f(R, K)$.



Hình 10: Chi tiết sơ đồ khối $F(R, K)$

2.2 Hiệu thực code

- Ở phần hiệu thực hàm $f(R, K)$ này, chúng ta có tổng cộng 3 phần mở rộng thêm: Expansion Table, Substitution Box, Permutation Box.
- Để hiệu thực phần code, chúng ta dựa trên các bảng ở phần 2.1 và lưu ý cách đánh số trong bảng là từ trái sang phải, để hiệu thực code ta cần phải phân tích sang input của khối đó.

Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6
0	63	62	61	60	59
60	59	58	57	56	55
56	55	54	53	52	51
52	51	50	49	48	47
48	47	46	45	44	43
44	43	42	41	40	39
40	39	38	37	36	35
36	35	34	33	32	31
32	31	30	29	28	27
28	27	26	25	24	23
24	23	22	21	20	19
20	19	18	17	16	15
16	15	14	13	12	11
12	11	10	9	8	7
8	7	6	5	4	3
4	3	2	1	0	63

Hình 11: Expansion Table được điều chỉnh theo input của khối

```

module F_FUNCTION(R, Key,F_out);
input [95:0] Key;
input [63:0] R;
output [63:0] F_out;

wire [95:0]E_OUT;
wire [5:0] S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16;
wire [63:0] S_out;
E_MATRIX E(.R(R),
.E_OUT(E_OUT));

E_XOR_KEY e_xor_key(
.E(E_OUT),
.K(Key),
.S1(S1),
.S2(S2),
.S3(S3),
.S4(S4),
.S5(S5),

```

```

.S6(S6),
.S7(S7),
.S8(S8),
.S9(S9),
.S10(S10),
.S11(S11),
.S12(S12),
.S13(S13),
.S14(S14),
.S15(S15),
.S16(S16);

S_BOX s_box_1(.S1_in(S1),
    .S2_in(S2),
    .S3_in(S3),
    .S4_in(S4),
    .S5_in(S5),
    .S6_in(S6),
    .S7_in(S7),
    .S8_in(S8),
    .S_out(S_out[63:32]));

S_BOX s_box_2(.S1_in(S9),
    .S2_in(S10),
    .S3_in(S11),
    .S4_in(S12),
    .S5_in(S13),
    .S6_in(S14),
    .S7_in(S15),
    .S8_in(S16),
    .S_out(S_out[31:0]));

P_MATRIX p_matrix(.P_in(S_out), .P_out(F_out));

```

endmodule

Hình 12: Hiện thực code Expansion Table

- Đổi với Sbox:

```

module S_BOX(S1_in,S2_in,S3_in,S4_in,S5_in,S6_in,S7_in,S8_in,S_out);

input [5:0] S1_in,S2_in,S3_in,S4_in,S5_in,S6_in,S7_in,S8_in;
output reg [31:0] S_out;
// S BOX 1
always @(*)
begin
    case ({S1_in[5], S1_in[0]})
        2'b00: begin
            case (S1_in[4:1])
                4'd0: S_out[31:28] = 4'd14;

```

```

4'd1: S_out[31:28] = 4'd4;
4'd2: S_out[31:28] = 4'd13;
4'd3: S_out[31:28] = 4'd1;
4'd4: S_out[31:28] = 4'd2;
4'd5: S_out[31:28] = 4'd15;
4'd6: S_out[31:28] = 4'd11;
4'd7: S_out[31:28] = 4'd8;
4'd8: S_out[31:28] = 4'd3;
4'd9: S_out[31:28] = 4'd10;
4'd10: S_out[31:28] = 4'd6;
4'd11: S_out[31:28] = 4'd12;
4'd12: S_out[31:28] = 4'd5;
4'd13: S_out[31:28] = 4'd9;
4'd14: S_out[31:28] = 4'd0;
default: S_out[31:28]      = 4'd7;
endcase
end
2'b01: begin
    case (S1_in[4:1])
        4'd0: S_out[31:28] = 4'd0;
        4'd1: S_out[31:28] = 4'd15;
        4'd2: S_out[31:28] = 4'd7;
        4'd3: S_out[31:28] = 4'd4;
        4'd4: S_out[31:28] = 4'd14;
        4'd5: S_out[31:28] = 4'd2;
        4'd6: S_out[31:28] = 4'd13;
        4'd7: S_out[31:28] = 4'd1;
        4'd8: S_out[31:28] = 4'd10;
        4'd9: S_out[31:28] = 4'd6;
        4'd10: S_out[31:28] = 4'd12;
        4'd11: S_out[31:28] = 4'd11;
        4'd12: S_out[31:28] = 4'd9;
        4'd13: S_out[31:28] = 4'd5;
        4'd14: S_out[31:28] = 4'd3;
        default: S_out[31:28]      = 4'd8;
    endcase
end

```

.....

Hình 13: Hiện thực code Sbox

- Đôi với phần code này khá dài nên chỉ lấy 1 phần.

Bảng lưu theo chỉ số input

Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6	Cột 7	Cột 8
0->31							
48	57	44	43	35	52	36	47
63	49	41	38	59	46	33	54
62	56	40	50	32	37	61	55
45	51	34	58	42	53	60	39
32->63							
16	25	12	11	3	20	4	15
31	17	9	6	27	14	1	22
30	24	8	18	0	5	29	23
13	19	2	26	10	21	28	7

Hình 14: Permutation Box dựa trên input của khối

```

module P_MATRIX(P_in, P_out);
input [63:0] P_in;
output [63:0] P_out;

assign P_out = {
    P_in[48], P_in[57], P_in[44], P_in[43], P_in[35], P_in[52], P_in[36], P_in[47],
    P_in[63], P_in[49], P_in[41], P_in[38], P_in[59], P_in[46], P_in[33], P_in[54],
    P_in[62], P_in[56], P_in[40], P_in[50], P_in[32], P_in[37], P_in[61], P_in[55],
    P_in[45], P_in[51], P_in[34], P_in[58], P_in[42], P_in[53], P_in[60], P_in[39],
    P_in[16], P_in[25], P_in[12], P_in[11], P_in[3], P_in[20], P_in[4], P_in[15],
    P_in[31], P_in[17], P_in[9], P_in[6], P_in[27], P_in[14], P_in[1], P_in[22],
    P_in[30], P_in[24], P_in[8], P_in[18], P_in[0], P_in[5], P_in[29], P_in[23],
    P_in[13], P_in[19], P_in[2], P_in[26], P_in[10], P_in[21], P_in[28], P_in[7]
};

endmodule

```

Hình 15: Hiện thực Verilog Permutation Box

- Và khi ta có được các thành phần cần thiết, tiến hành xây dựng module Function để thực hiện tạo giá trị của hàm f(R,K)

```

module F_FUNCTION(R, Key,F_out);
input [95:0] Key;
input [63:0] R;
output [63:0] F_out;

wire [95:0]E_OUT;
wire [5:0] S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16;
wire [63:0] S_out;
E_MATRIX E(.R(R),
            .E_OUT(E_OUT));

```

```

E_XOR_KEY e_xor_key(
    .E(E_OUT),
    .K(Key),
    .S1(S1),
    .S2(S2),
    .S3(S3),
    .S4(S4),
    .S5(S5),
    .S6(S6),
    .S7(S7),
    .S8(S8),
    .S9(S9),
    .S10(S10),
    .S11(S11),
    .S12(S12),
    .S13(S13),
    .S14(S14),
    .S15(S15),
    .S16(S16));

S_BOX s_box_1(.S1_in(S1),
    .S2_in(S2),
    .S3_in(S3),
    .S4_in(S4),
    .S5_in(S5),
    .S6_in(S6),
    .S7_in(S7),
    .S8_in(S8),
    .S_out(S_out[63:32]));

S_BOX s_box_2(.S1_in(S9),
    .S2_in(S10),
    .S3_in(S11),
    .S4_in(S12),
    .S5_in(S13),
    .S6_in(S14),
    .S7_in(S15),
    .S8_in(S16),
    .S_out(S_out[31:0]));

P_MATRIX p_matrix(.P_in(S_out), .P_out(F_out));

```

endmodule

Hình 16: Hiện thực hàm f(R,K)

3. Khóa vòng

3.1 Tổng quan cách thức hoạt động

Cấu trúc Khóa đầu vào

- Độ dài: 128 bit.
- Số bit thực sự dùng: 112 bit.

- Bit Parity là các bit kiểm tra chẵn lẻ và không tham gia vào quá trình tạo khóa chính, các bit nằm ở vị trí 8, 16, 32, ..., 128 (bội số của 8).

3.2 Chuẩn hóa chỉ số

Trong quá trình phát triển mô hình lý thuyết và hiện thực hóa bằng ngôn ngữ mô tả phần cứng (Verilog) cho thuật toán thì có sự khác biệt về quy ước đánh dấu chỉ số cho các bit của dữ liệu đầu vào như sau:

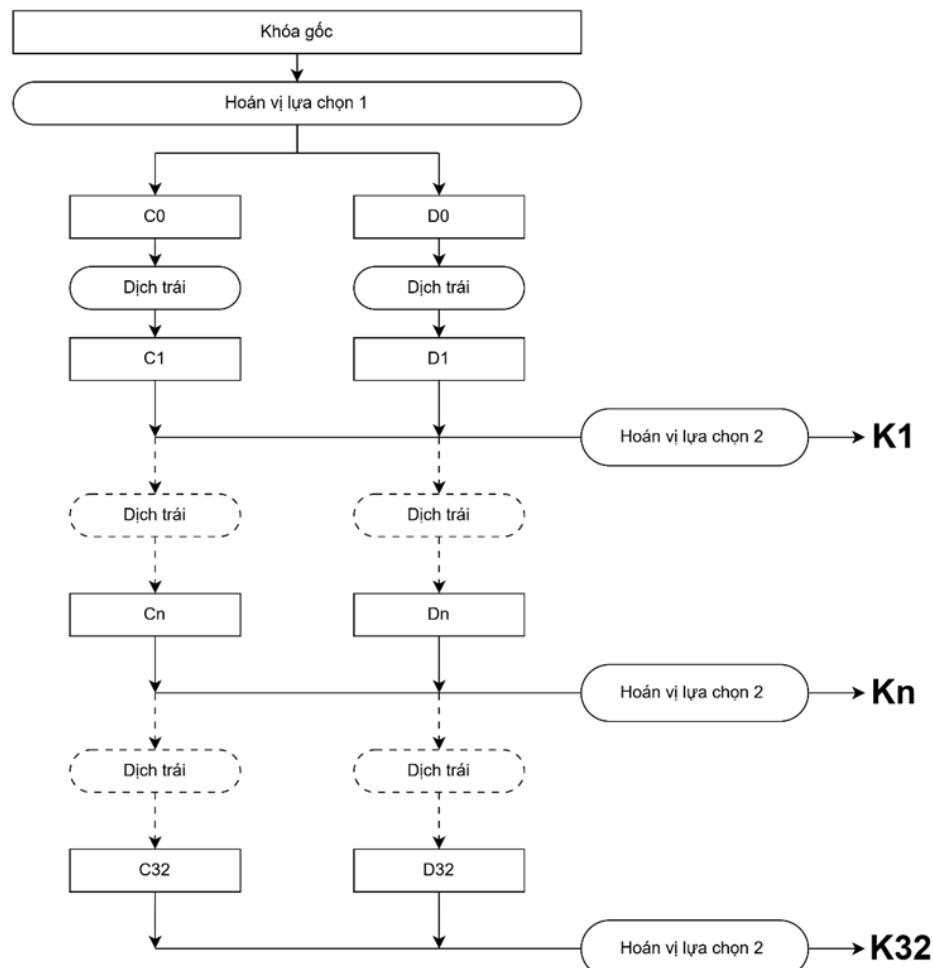
- Mô hình lý thuyết: Đánh số thứ tự từ 1 (MSB) đến 128 (LSB).
- Verilog: Mảng và vector sử dụng chỉ số từ 127 (MSB) đến 0 (LSB).

Do đó trong mô hình lý thuyết, ta cần hiệu chỉnh lại chỉ số trong bảng hoán vị gốc về dạng 0-127 bằng cách trừ đi 1 các giá trị trong bảng. Sau đó tiếp tục hiệu chỉnh lại chỉ số theo dạng 127-0 để tạo ra bảng hoán vị phù hợp cho chương trình Verilog.

3.3 Nguyên lý hoạt động

Quy trình tổng quát được thực hiện lần lượt qua 3 bước chính:

- Hoán vị lựa chọn 1 (PC-1): Chọn ra 112 bit từ khóa gốc và hoán vị lại vị trí các bit, sau đó chia thành hai nửa vào hai thanh ghi C0 (56 bit đầu) và D0 (56 bit sau).
- Dịch vòng trái: Tại mỗi vòng thứ n (1 đến 32) thì các thanh ghi Cn-1 và Dn-1 được dịch trái theo số bit quy định (1 hoặc 2 bit) để tạo ra giá trị cho thanh ghi Cn và Dn.
- Hoán vị lựa chọn 2 (PC-2): Tại mỗi vòng, Cn và Dn được ghép lại và hoán vị vị trí các bit để tạo ra khóa vòng Kn (96 bit).



Hình 17: Sơ đồ thuật toán tính khóa vòng

3.4 Các thành phần chi tiết

3.4.1 Hoán vị lựa chọn 1 (PC-1)

Dữ liệu đầu vào đã được đánh số thứ tự 1-128 sẽ được lựa chọn và sắp xếp như sau:

Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6	Cột 7
Nửa đầu						
57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4
Nửa sau						
121	113	105	97	89	81	73
65	122	114	106	98	90	82
74	66	123	115	107	99	91
83	75	67	124	116	108	100
127	119	111	103	95	87	79
71	126	118	110	102	94	86
78	70	125	117	109	101	93
85	77	69	92	84	76	68

Hình 18: Bảng hoán vị lựa chọn 1

Sau đó, hiệu chỉnh lại chỉ số về dạng 0-127 như sau:

Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6	Cột 7
Nửa đầu						
56	48	40	32	24	16	8
0	57	49	41	33	25	17
9	1	58	50	42	34	26
18	10	2	59	51	43	35
62	54	46	38	30	22	14
6	61	53	45	37	29	21
13	5	60	52	44	36	28
20	12	4	27	19	11	3
Nửa sau						
120	112	104	96	88	80	72
64	121	113	105	97	89	81
73	65	122	114	106	98	90
82	74	66	123	115	107	99
126	118	110	102	94	86	78
70	125	117	109	101	93	85
77	69	124	116	108	100	92
84	76	68	91	83	75	67

Hình 19: Bảng hoán vị lựa chọn 1 dạng 0-127

Để hiện thực hóa thuật toán bằng Verilog thì phải hiệu chỉnh lại chỉ số của dữ liệu đầu vào như sau:

Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6	Cột 7
Nửa đầu						
71	79	87	95	103	111	119
127	70	78	86	94	102	110
118	126	69	77	85	93	101
109	117	125	68	76	84	92
65	73	81	89	97	105	113
121	66	74	82	90	98	106
114	122	67	75	83	91	99
107	115	123	100	108	116	124
Nửa sau						
7	15	23	31	39	47	55
63	6	14	22	30	38	46
54	62	5	13	21	29	37
45	53	61	4	12	20	28
1	9	17	25	33	41	49
57	2	10	18	26	34	42
50	58	3	11	19	27	35
43	51	59	36	44	52	60

Hình 20: Hoán vị lựa chọn 1 sau khi hiệu chỉnh

Kết quả thu được sau khi key gốc được hoán vị lựa chọn 1 (PC-1) sẽ chia vào 2 thanh ghi C0 (nửa đầu) và D0 (nửa sau).

3.4.2 Dịch vòng trái

Từ kết quả sau khi hoán vị lựa chọn 1 (PC-1), các khóa vòng Kn (với n từ 1 đến 32) sẽ được tính theo nguyên tắc “Giá trị của khóa vòng thứ n được tính từ giá trị khóa vòng thứ $n-1$ ” như công thức:

$$K_n = PC_{-2}(C_n D_n)$$

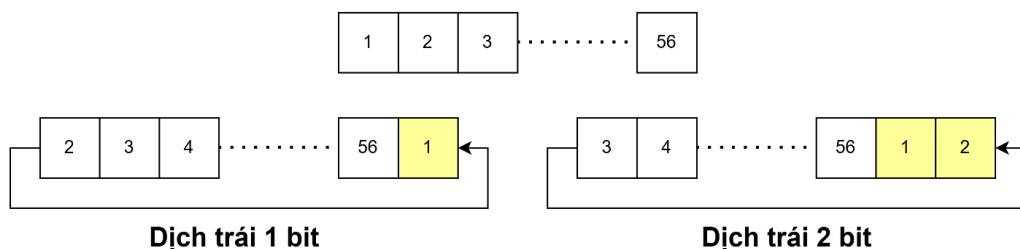
Trong đó: Cn và Dn được khởi tạo bằng cách dịch trái các giá trị Cn-1 và Dn-1 với số bit được quy định trong bảng sau:

Thứ tự lặp lại	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Số bit dịch trái	1	1	2	2	2	2	2	2	1	2	2	2	2	2	1	1	1	2	2	2	2	2	2	2	1	2	2	2	2	2	1	

Bảng 3.2.2: Bảng quy định số bit dịch trái khi tính khóa vòng

Theo bảng trên, ví dụ ta xét C5 và D5 có được bằng cách dịch trái 2 bit C4 và D4 hoặc C32 và D32 có được bằng cách dịch trái 1 bit C31 và D31. Dịch trái ở đây được hiểu là quay trái như minh họa sau:

C_{n-1} hoặc D_{n-1}



3.4.3 Hoán vị lựa chọn 2 (PC-2)

Dữ liệu hai thanh ghi Cn và Dn được ghép lại và đánh số thứ tự từ 1 (MSB) đến 112 (LSB) và được lưu

chọn và sắp xếp như sau:

Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6	Cột 7	Cột 8
14	17	11	24	1	5	3	28
15	6	21	10	23	19	12	4
26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40
51	45	33	48	44	49	39	56
34	53	46	42	50	36	29	32
70	73	67	80	57	61	59	84
71	62	77	66	79	75	68	60
82	64	72	63	83	76	69	58
97	108	87	93	103	111	86	96
107	101	89	104	100	105	95	112
90	109	102	98	106	92	85	88

Hình 21: Bảng hoán vị lựa chọn 2

Sau đó, hiệu chỉnh lại chỉ số về dạng 0-127 như sau:

Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6	Cột 7	Cột 8
13	16	10	23	0	4	2	27
14	5	20	9	22	18	11	3
25	7	15	6	26	19	12	1
40	51	30	36	46	54	29	39
50	44	32	47	43	48	38	55
33	52	45	41	49	35	28	31
69	72	66	79	56	60	58	83
70	61	76	65	78	74	67	59
81	63	71	62	82	75	68	57
96	107	86	92	102	110	85	95
106	100	88	103	99	104	94	111
89	108	101	97	105	91	84	87

Hình 22: Bảng hoán vị lựa chọn 2 dạng 0-127

Để hiện thực hóa thuật toán bằng Verilog thì phải hiệu chỉnh lại chỉ số của dữ liệu đầu vào như sau:

Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6	Cột 7	Cột 8
98	95	101	88	111	107	109	84
97	106	91	102	89	93	100	108
86	104	96	105	85	92	99	110
71	60	81	75	65	57	82	72
61	67	79	64	68	63	73	56
78	59	66	70	62	76	83	80
42	39	45	32	55	51	53	28
41	50	35	46	33	37	44	52
30	48	40	49	29	36	43	54
15	4	25	19	9	1	26	16
5	11	23	8	12	7	17	0
22	3	10	14	6	20	27	24

Hình 23: Bảng hoán vị lựa chọn 2 sau khi hiệu chỉnh

3.5 Kiến trúc phần cứng

3.5.1 Các thành phần chính

Khối hoán vị lựa chọn 1 (PC_1)

- Đầu vào: 128 bit của Key_in
- Đầu ra: 112 bit

Bộ chọn luồng dữ liệu (MUX1 và MUX2)

- Quyết định nguồn dữ liệu để tạo khóa vòng.
- Select = 1: Chọn dữ liệu mới từ PC_1 (C0, D0) để bắt đầu quá trình mã hóa.
- Select = 0: Chọn dữ liệu hồi tiếp (Feedback) từ kết quả dịch bit của vòng trước (Out_mux3, Out_mux4) để thực hiện vòng tiếp theo.

Thanh ghi trạng thái (Register_C0_D0)

- Lưu trữ giá trị hiện tại của hai nửa khóa C (56-bit) và D (56-bit).
- Cập nhật giá trị mới tại mỗi cạnh lên của xung nhịp Clk

Khối dịch bit (Shifters)

- Hai bộ dịch bit được thiết kế song song cho mỗi nửa Cn và Dn:
- SHIFT_LEFT_1: Dịch trái 1 bit.
- SHIFT_LEFT_2: Dịch trái 2 bit

Bộ chọn phép dịch (MUX 3 và MUX 4)

- Được điều khiển bởi tín hiệu select_mux_shift từ STATE_MACHINE.
- Chọn kết quả từ bộ dịch 1 bit hoặc 2 bit tùy thuộc vào quy định của vòng hiện tại

Khối hoán vị lựa chọn 2 (PC_2)

- Đầu vào: Nhận giá trị thanh ghi Cn và Dn sau khi đã dịch bit.
- Đầu ra: Chọn lọc và hoán vị để tạo ra khóa vòng Round_key [95:0].

Máy trạng thái (STATE_MACHINE)

- Xác định hệ thống đang ở vòng nào, từ đó xuất tín hiệu điều khiển select_mux_pc (nạp mới/lắp) và select_mux_shift (dịch 1 hoặc 2 bit).

3.5.2 Luồng dữ liệu

Khởi tạo

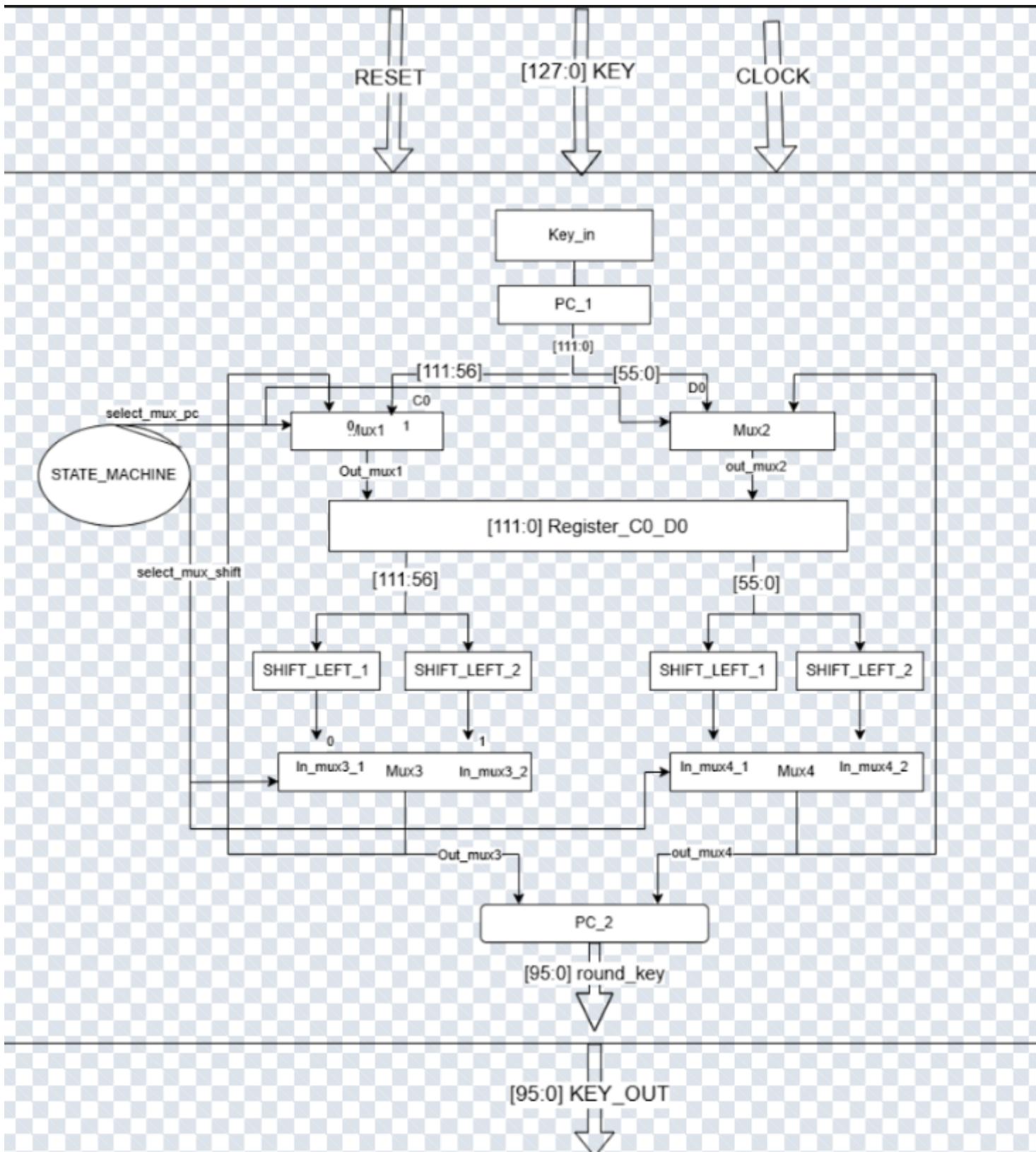
- Kích hoạt tín hiệu Reset, Counter về 0.
- Tại Counter = 1: STATE_MACHINE đặt select_mux_pc = 1. MUX1 và MUX2 cho phép dữ liệu từ PC_1 đi vào thanh ghi Register_C0_D0.

Vòng lắp tính toán

- Từ Counter = 2 đến 32 (và chu trình lắp lại):
- select_mux_pc chuyển về 0, đóng kín vòng lắp hồi tiếp.
- Dữ liệu từ Register đi qua các khối SHIFT.
- STATE_MACHINE kiểm tra Counter. Nếu thuộc các vòng cần dịch 1 bit, nó đặt select_mux_shift = 0. Nếu cần dịch 2 bit, nó đặt select_mux_shift = 1.
- Kết quả sau khi chọn (MUX3/4) được đưa trở lại đầu vào của Register (cho vòng sau) và đồng thời đưa vào PC_2.

Xuất kết quả

- Tại mỗi chu kỳ, PC_2 lấy dữ liệu đã dịch bit và tạo ra Key_out.



Hình 24: Sơ đồ khái kiến trúc phần cứng của Round Key

3.6 Hiện thực code

Dựa trên Kiến trúc phần cứng của Round Key, ta có các module sau:

Module ROUND_KEY

```
module ROUND_KEY(Key_in,Clk,Clk2, Reset, Key_out,Done, Counter );
  input [127:0] Key_in;
```

```

input Clk, Reset;
input Clk2;
output [95:0] Key_out;
output Done;
output [5:0] Counter;
wire [55:0] C0, D0;
wire [55:0] Out_mux1, out_mux2, Out_mux3, Out_mux4;
wire [55:0] Reg_out_C0, Reg_out_D0;
wire [55:0] In_mux3_1, In_mux3_2, In_mux4_1, In_mux4_2;

// ===== TIN HIEU MAY TRANG THAI =====

wire Select_mux_pc;
wire Select_mux_shift;

STATE_MACHINE STATE(.Clk(Clk2),
    .Reset(Reset),
    .Select_mux_pc_temp(Select_mux_pc),
    .Select_mux_shift_temp(Select_mux_shift),
    .Counter(Counter),
    .Pre_state(),
    .Next_state(),
    .Done(Done));

// ===== KHOI HOAN VI =====

PC_1 PC1(.Des_key_in(Key_in),
    .Reset(Reset),
    .C0(C0),
    .D0(D0));

// ===== MUX CHON PC 1 VA C1 HOAC D1 =====

MUX2_1 MUX1 (.In_a(C0), // select = 1 -> chon tin hieu tu PC_1 ( vong dau tien )
    .Select(Select_mux_pc),
    .In_b(Out_mux3),
    .Out(Out_mux1));

MUX2_1 MUX2 (.In_a(D0), // select = 1 -> chon tin hieu tu PC_1 ( vong dau tien )
    .Select(Select_mux_pc),
    .In_b(Out_mux4),
    .Out(out_mux2));

// ===== KHOI REGISTER LUU GIA TRI SAU MUX 1 VA MUX 2 =====

REGISTER_C0_D0 REGISTER(.C0(Out_mux1),
    .D0(out_mux2),
    .Clk(Clk));

```

```

    .Reset(Reset),
    .C0_out(Reg_out_C0),
    .D0_out(Reg_out_D0));
// ===== KHOI SHIFT LEFT 1, 2
=====

SHIFT_LEFT_1 SHIFT1(.C_in(Reg_out_C0), .C_out(In_mux3_1));
SHIFT_LEFT_2 SHIFT2(.C_in(Reg_out_C0), .C_out(In_mux3_2));

SHIFT_LEFT_1 SHIFT3(.C_in(Reg_out_D0), .C_out(In_mux4_1));
SHIFT_LEFT_2 SHIFT4(.C_in(Reg_out_D0), .C_out(In_mux4_2));

// ===== KHOI MUX 3 VA MUX 4 =====

MUX2_1 MUX3 (.In_a(In_mux3_2),
    .Select(Select_mux_shift), // select = 1 -> dich 2
    .In_b(In_mux3_1),
    .Out(Out_mux3));

MUX2_1 MUX4 (.In_a(In_mux4_2),
    .Select(Select_mux_shift), // select = 1 -> dich 2
    .In_b(In_mux4_1),
    .Out(Out_mux4));

PC_2 PC2 (.In({Out_mux3,Out_mux4}),
    .Round_key(Key_out));

endmodule

```

Module Hoán vị lựa chọn 1 (PC_1)

```

module PC_1(Des_key_in,Reset,C0,D0);

input [127:0] Des_key_in; // ===== 128 BIT KEY DAU VAO
=====
input Reset;
output reg [55:0] C0,D0; //===== 112 BIT KEY DAU RA
=====

always @(*)
begin
if(Reset)
begin
  C0 = 55'b0;
  D0 = 55'b0;
end
else
begin

```

```

C0 = {
    Des_key_in[71], Des_key_in[79], Des_key_in[87], Des_key_in[95], Des_key_in[103],
Des_key_in[111], Des_key_in[119],
    Des_key_in[127], Des_key_in[70], Des_key_in[78], Des_key_in[86], Des_key_in[94],
Des_key_in[102], Des_key_in[110],
    Des_key_in[118], Des_key_in[126], Des_key_in[69], Des_key_in[77], Des_key_in[85],
Des_key_in[93], Des_key_in[101],
    Des_key_in[109], Des_key_in[117], Des_key_in[125], Des_key_in[68], Des_key_in[76],
Des_key_in[84], Des_key_in[92],
    Des_key_in[65], Des_key_in[73], Des_key_in[81], Des_key_in[89], Des_key_in[97],
Des_key_in[105], Des_key_in[113],
    Des_key_in[121], Des_key_in[66], Des_key_in[74], Des_key_in[82], Des_key_in[90],
Des_key_in[98], Des_key_in[106],
    Des_key_in[114], Des_key_in[122], Des_key_in[67], Des_key_in[75], Des_key_in[83],
Des_key_in[91], Des_key_in[99],
    Des_key_in[107], Des_key_in[115], Des_key_in[123], Des_key_in[100], Des_key_in[108],
Des_key_in[116], Des_key_in[124]
};

D0 = {
    Des_key_in[7], Des_key_in[15], Des_key_in[23], Des_key_in[31], Des_key_in[39],
Des_key_in[47], Des_key_in[55],
    Des_key_in[63], Des_key_in[6], Des_key_in[14], Des_key_in[22], Des_key_in[30],
Des_key_in[38], Des_key_in[46],
    Des_key_in[54], Des_key_in[62], Des_key_in[5], Des_key_in[13], Des_key_in[21],
Des_key_in[29], Des_key_in[37],
    Des_key_in[45], Des_key_in[53], Des_key_in[61], Des_key_in[4], Des_key_in[12],
Des_key_in[20], Des_key_in[28],
    Des_key_in[1], Des_key_in[9], Des_key_in[17], Des_key_in[25], Des_key_in[33],
Des_key_in[41], Des_key_in[49],
    Des_key_in[57], Des_key_in[2], Des_key_in[10], Des_key_in[18], Des_key_in[26],
Des_key_in[34], Des_key_in[42],
    Des_key_in[50], Des_key_in[58], Des_key_in[3], Des_key_in[11], Des_key_in[19],
Des_key_in[27], Des_key_in[35],
    Des_key_in[43], Des_key_in[51], Des_key_in[59], Des_key_in[36], Des_key_in[44],
Des_key_in[52], Des_key_in[60]
};

end
end
endmodule

```

Module Hoán vị lựa chọn 2 (PC_2)

```

module PC_2(In, Round_key);
    input [111:0] In;
    output [95:0] Round_key;

    assign Round_key = {

```

```

// row 1
In[98], In[95], In[101], In[88], In[111], In[107], In[109], In[84],
// row 2
In[97], In[106], In[91], In[102], In[89], In[93], In[100], In[108],
// row 3
In[86], In[104], In[96], In[105], In[85], In[92], In[99], In[110],
// row 4
In[71], In[60], In[81], In[75], In[65], In[57], In[82], In[72],
// row 5
In[61], In[67], In[79], In[64], In[68], In[63], In[73], In[56],
// row 6
In[78], In[59], In[66], In[70], In[62], In[76], In[83], In[80],
// row 7
In[42], In[39], In[45], In[32], In[55], In[51], In[53], In[28],
// row 8
In[41], In[50], In[35], In[46], In[33], In[37], In[44], In[52],
// row 9
In[30], In[48], In[40], In[49], In[29], In[36], In[43], In[54],
// row 10
In[15], In[4], In[25], In[19], In[9], In[1], In[26], In[16],
// row 11
In[5], In[11], In[23], In[8], In[12], In[7], In[17], In[0],
// row 12
In[22], In[3], In[10], In[14], In[6], In[20], In[27], In[24]
};

endmodule

```

Module Thanh ghi trạng thái

```

module REGISTER_C0_D0(C0,Clk,Reset, D0,C0_out,D0_out);
    input [55:0] C0, D0;
    input Clk;
    input Reset;
    output reg [55:0] C0_out;
    output reg [55:0] D0_out;           //

    reg [111:0]Reg_temp ;           // DOC TAI CANH LEN
    always @(posedge Clk, posedge Reset)
    begin
        if(Reset)
            begin
                C0_out <=28'b0;
                D0_out <= 28'b0;
            end
        else
            begin
                C0_out <= Reg_temp[111:56];
            end
    end
endmodule

```

```

D0_out <= Reg_temp[55:0];
end
end

always @ (posedge Clk, posedge Reset)           // GHI TAI CANH XUONG
begin
if(Reset)
    Reg_temp <=0;
else
begin
    Reg_temp[111:56] <= C0;
    Reg_temp[55:0] <= D0;
end
end
endmodule

```

Module MUX2_1

```

module MUX2_1(In_a,Select, In_b, Out);
    input [55:0] In_a;
    input [55:0] In_b;
    output [55:0] Out;
    input Select;
    assign Out = (Select == 1'b1) ? In_a : In_b;
endmodule

```

Module Dịch trái 1 bit

```

module SHIFT_LEFT_1(C_in, C_out);
    input [55:0] C_in; //      56 bit key vao
    output [55:0] C_out ; //      56 bit sau khi dich
    assign C_out = {C_in[54:0], C_in[55]}; //
endmodule

```

Module Dịch trái 2 bit

```

module SHIFT_LEFT_2(C_in, C_out);
    input [55:0] C_in; //      56 bit key vao
    output [55:0] C_out ; //      56 bit sau khi dich
    assign C_out = {C_in[53:0], C_in[55:54]}; //
endmodule

```

4. Hoán vị khởi tạo đảo IP_1

4.1 Tổng quan cách hoạt động

- Đây là bước xử lý cuối cùng của thuật toán mã hóa DES mở rộng. Dữ liệu sau khi đi qua các vòng lặp và thực hiện phép tráo đổi 64bit cuối cùng (Swap) sẽ được đưa vào khối **Hoán vị khởi tạo đảo IP⁻¹**.
- Mục đích của khối này là thực hiện phép hoán vị ngược lại so với hoán vị khởi tạo (IP) ban đầu để khôi phục lại trạng thái bit cần thiết, tạo ra chuỗi **Ciphertext 128bit** (bản mã hóa) hoàn chỉnh. Quy tắc hoán vị được xác định theo bảng dưới đây:

Bảng_2							
Cột 0	Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6	Cột 7
40(LSB)	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25
Nửa sau							
104	72	112	80	120	88	128	96
103	71	111	79	119	87	127	95
102	70	110	78	118	86	126	94
101	69	109	77	117	85	125	93
100	68	108	76	116	84	124	92
99	67	107	75	115	83	123	91
98	66	106	74	114	82	122	90
97	65	105	73	113	81	121	89(MSB)

Hình 25: Hoán vị khởi tạo đảo IP⁻¹

4.2 Chuẩn hóa chỉ số cho hiện thực Verilog

- Để hiện thực hóa bảng hoán vị trên vào mã nguồn Verilog, ta cần thực hiện bước chuẩn hóa chỉ số tương tự như khối IP đầu vào.
- Bảng dưới đây thể hiện chi tiết các giá trị chỉ số sau khi đã chuẩn hóa, được dùng trực tiếp để gán tín hiệu trong lúc hiện thực code:

Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6	Cột 7	(Cột 8)
Nửa đầu (Dành cho bit đầu ra 0-63)							
39[LSB]	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25
32	0	40	8	48	16	56	24
Nửa sau (Dành cho bit đầu ra 64-127)							
103	71	111	79	119	87	127	95
102	70	110	78	118	86	126	94
101	69	109	77	117	85	125	93
100	68	108	76	116	84	124	92
99	67	107	75	115	83	123	91
98	66	106	74	114	82	122	90
97	65	105	73	113	81	121	89
96	64	104	72	112	80	120	88[MSB]

Hình 26: Hoán vị khởi tạo đảo IP⁻¹ sau khi trừ 1

Cột 1	Cột 2	Cột 3	Cột 4	Cột 5	Cột 6	Cột 7	Cột 8
Nửa đầu (Dành							
24		56	16	48	8	40	0
25		57	17	49	9	41	1
26		58	18	50	10	42	2
27		59	19	51	11	43	3
28		60	20	52	12	44	4
29		61	21	53	13	45	5
30		62	22	54	14	46	6
31		63	23	55	15	47	7
Nửa sau (Dành							
88		120	80	112	72	104	64
89		121	81	113	73	105	65
90		122	82	114	74	106	66
91		123	83	115	75	107	67
92		124	84	116	76	108	68
93		125	85	117	77	109	69
94		126	86	118	78	110	70
95		127	87	119	79	111	71

Hình 27: Sơ đồ lưu trữ trong Registers

4.3 Hiện thực code

- Dựa trên bảng ánh xạ địa chỉ và phân chia dữ liệu đã thiết lập ở các mục 4.2, khối Hoán vị khởi tạo đảo IP⁻¹ được hiện thực bằng ngôn ngữ Verilog như sau:

```
module IP_1(In, Reset, Ciphertext);
    input [127:0] In;
    input Reset;
```

```

output reg [127:0] Ciphertext;
always @(*)
begin
    if (Reset)
        begin
            Ciphertext = 128'b0;
        end
        else
            begin

                Ciphertext = {

                    In[88], In[120], In[80], In[112], In[72], In[104], In[64], In[96],
                    In[89], In[121], In[81], In[113], In[73], In[105], In[65], In[97],
                    In[90], In[122], In[82], In[114], In[74], In[106], In[66], In[98],
                    In[91], In[123], In[83], In[115], In[75], In[107], In[67], In[99],
                    In[92], In[124], In[84], In[116], In[76], In[108], In[68], In[100],
                    In[93], In[125], In[85], In[117], In[77], In[109], In[69], In[101],
                    In[94], In[126], In[86], In[118], In[78], In[110], In[70], In[102],
                    In[95], In[127], In[87], In[119], In[79], In[111], In[71], In[103],
                    In[24], In[56], In[16], In[48], In[8], In[40], In[0], In[32],
                    In[25], In[57], In[17], In[49], In[9], In[41], In[1], In[33],
                    In[26], In[58], In[18], In[50], In[10], In[42], In[2], In[34],
                    In[27], In[59], In[19], In[51], In[11], In[43], In[3], In[35],
                    In[28], In[60], In[20], In[52], In[12], In[44], In[4], In[36],
                    In[29], In[61], In[21], In[53], In[13], In[45], In[5], In[37],
                    In[30], In[62], In[22], In[54], In[14], In[46], In[6], In[38],
                    In[31], In[63], In[23], In[55], In[15], In[47], In[7], In[39]
                };
            end
        end
    endmodule

```

- Mô tả hiện thực code phía trên:

- Giao diện (Interface):
 - Input: In [127:0] nhận dữ liệu đầu vào từ vòng lặp cuối cùng (sau bước Swap); Reset dùng để khởi tạo trạng thái.
 - Output: Ciphertext [127:0] là chuỗi dữ liệu đầu ra cuối cùng.
- Logic xử lý:
 - Thứ tự các bit trong mã nguồn tuân thủ chính xác bảng tra cứu chỉ số đã xây dựng tại mục 4.2.
 - Kết quả cuối cùng được thực hiện bằng phép ghép bit {...} và gán trực tiếp cho biến ngõ ra Ciphertext. Khác với khối IP đầu vào, khối này không chia tách dữ liệu mà gộp lại thành một chuỗi 128bit duy nhất để xuất ra kết quả mã hóa.

5. State machine

- Khối điều khiển (Control Unit) được thiết kế dưới dạng một máy trạng thái hữu hạn (Finite State Machine - FSM) để điều phối toàn bộ hoạt động của bộ mã hóa. Máy trạng thái này sinh ra các tín hiệu điều khiển cho các bộ dồn kênh (Mux) và bộ dịch bit (Shifter) dựa trên chu kỳ hoạt động hiện tại.
- Hệ thống hoạt động dựa trên hai trạng thái chính:
 - **INIT:** Trạng thái khởi tạo và chờ tín hiệu bắt đầu.
 - **PROCESSING:** Trạng thái thực thi mã hóa, điều khiển việc sinh khóa và tính toán qua các vòng lặp dựa trên tín hiệu Counter.

5.1. Bảng hoạt động của các trạng thái (State Action)

- Bảng dưới đây mô tả chi tiết các điều kiện chuyển đổi trạng thái và các tín hiệu ngõ ra tương ứng:

State	Next State		Operation
	Condition	State	
INIT	Start = 1		Select_mux_pc_temp= 1'b0; // CHON TIN HIEU TU MUX Select_mux_shift_temp = 1'b0; // CHON QUAY TRAI 1 BIT Done = 0;
PROCESSING	Counter == 1	PROCESSING	Select_mux_pc_temp= 1'b1; // CHON TIN HIEU C0, D0
	(Counter >= 4 && Counter <=9) (Counter >= 20 && Counter <=25)		Select_mux_shift_temp = 1'b1; // KEY 3: CHON QUAY TRAI 2 BIT
	(Counter ==10) (Counter ==26)		Select_mux_shift_temp = 1'b0; // CHON QUAY TRAI 1 BIT
	(Counter >= 11 && Counter <=16) (Counter >= 27 && Counter <=32)		Select_mux_shift_temp = 1'b1;
	Counter ==33	INIT	Done = 1;

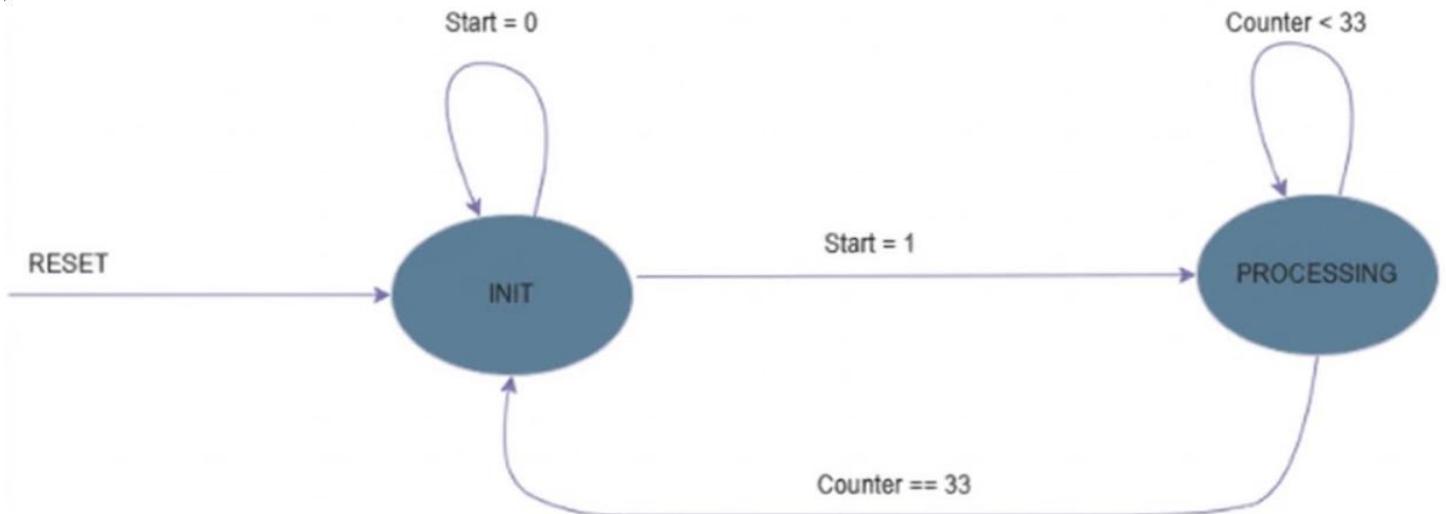
Hình 28: STATE ACTION TABLE

- Giải thích bảng:
 - Select_mux_pc_temp: Tín hiệu chọn đầu vào cho vòng lặp. Tại chu kỳ đầu (Counter = 1),

- tín hiệu này lên mức 1 để nạp dữ liệu gốc.
- Select_mux_shift_temp: Tín hiệu điều khiển số bit dịch trái. Tùy thuộc vào vòng lặp thứ mấy (dựa trên Counter), khóa sẽ được dịch 1 bit hoặc 2 bit theo đúng tiêu chuẩn thuật toán.
 - Done: Tín hiệu báo hiệu quá trình mã hóa hoàn tất khi Counter đạt giá trị 33.
 - Start: Tín hiệu báo hiệu bắt đầu quá trình mã hóa.

5.2 Lưu đồ chuyển trạng thái

- Giản đồ thể hiện sự chuyển đổi giữa trạng thái chờ (INIT) và trạng thái xử lý (PROCESSING) thông qua tín hiệu Start và bộ đếm Counter:



Hình 29: Giản đồ chuyển trạng thái

- Mô tả hoạt động:
 - Khi có tín hiệu Reset, hệ thống quay về trạng thái INIT.
 - Khi có tín hiệu Start = 1, hệ thống chuyển sang trạng thái PROCESSING.
 - Tại PROCESSING, hệ thống duy trì trạng thái này và tăng biến đếm Counter sau mỗi chu kỳ xung nhịp.
 - Khi Counter đạt giá trị 33 (hoàn thành đủ các vòng lặp mã hóa), hệ thống kích hoạt cờ xong và quay trở về trạng thái INIT.

5.3 Hiện thực code

- Dựa trên bảng hoạt động và lưu đồ đã thiết kế, khối STATE_MACHINE được hiện thực bằng ngôn ngữ Verilog:

```

module STATE_MACHINE(Clk,
    Reset,
    Select_mux_pc_temp,
    Select_mux_shift_temp,
    Counter,
    Pre_state,
    Next_state, Done);
  
```

```

input Clk, Reset;
output reg Select_mux_pc_temp,Select_mux_shift_temp;
output reg [5:0] Counter;
  
```

```

output reg Done;
localparam INIT = 0, PROCESSING = 1;
output reg Pre_state , Next_state;
// ===== COUNTER DEM SO LUONG CHU KI
always @(posedge Clk , posedge Reset) begin
  if(Reset)
    Counter <=0;
  /*else if (Counter==17)
    Counter <= 1;*/
  else
    begin
      if(Counter==33)
        Counter<=0;
      else
        Counter <= Counter + 1;
    end
  end

// ===== FINITE STATE MACHINCE
always @(posedge Clk, posedge Reset)
begin
  if(Reset)
    Pre_state <=INIT;
  else
    Pre_state <= Next_state;
end

always@(*)
begin
  case(Pre_state)
    INIT:
    begin
      Next_state = PROCESSING;
      Done = 0;
    end
    PROCESSING:
    begin
      Select_mux_pc_temp= 1'b0;          // CHON TIN HIEU TU MUX
      Select_mux_shift_temp = 1'b0;       // CHON QUAY TRAI 1 BIT
      Done = 0;                         // COUNTER 1 THI MOI BAT DAU GHI GIA
    end
  end
  TRI
  if(Counter == 1||Counter==17)
    Select_mux_pc_temp= 1'b1;           CHON TIN HIEU C0, D0
  else if((Counter >= 4 && Counter <=9) || (Counter >= 20 && Counter <=25))
    Select_mux_shift_temp = 1'b1;        KEY 3: CHON QUAY TRAI 2 BIT
  else if ((Counter ==10) || (Counter ==26) )
    Select_mux_shift_temp = 1'b0;
  else if ((Counter >= 11 && Counter <=16)|| (Counter >= 27 && Counter <=32))
    Select_mux_shift_temp = 1'b1;

```

```
else
begin
Select_mux_shift_temp = 1'b0;
if( Counter ==33)
begin
Next_state = INIT;
Done = 1;
end
else
Next_state = PROCESSING;
end
end
default: Next_state = INIT;
endcase
end
endmodule
```

CHƯƠNG 3: KIỂM TRA THIẾT KẾ VÀ MÔ PHỎNG

I. Kiểm tra kết quả và mô phỏng chức năng

kiểm tra kết quả :	L	R
1	00000000ff000000	af4e5baeda5d0aec
2	af4e5baeda5d0aec	7ec8c06a1e0129aa
3	7ec8c06a1e0129aa	f6dd74284813995c
4	f6dd74284813995c	40fe809bb81af787
5	40fe809bb81af787	95901b219b43f8a0
6	95901b219b43f8a0	d94ea54518b16abd
7	d94ea54518b16abd	f3bebc04119872e
8	f3bebc04119872e	6f714e6738f10ec4
9	6f714e6738f10ec4	98bb055daf9e6654
10	98bb055daf9e6654	46aedf9422316fc5
11	46aedf9422316fc5	6a16ce129f2c65a5
12	6a16ce129f2c65a5	a1b8fe8298d8b9f4
13	a1b8fe8298d8b9f4	70dcb32f49f64245
14	70dcb32f49f64245	523c9e2ee494f292
15	523c9e2ee494f292	8b1bb8f06f456ecf
16	8b1bb8f06f456ecf	b29bf6d528302a2d
.....		
32	abccae176ea062b	895a30c2c47b52d5

kết quả cuối cùng là 623982703e24bbc343ee9c7681e7b175

Hình 30: Kết quả tính toán lý thuyết của thuật toán

1. Pre-simulation.

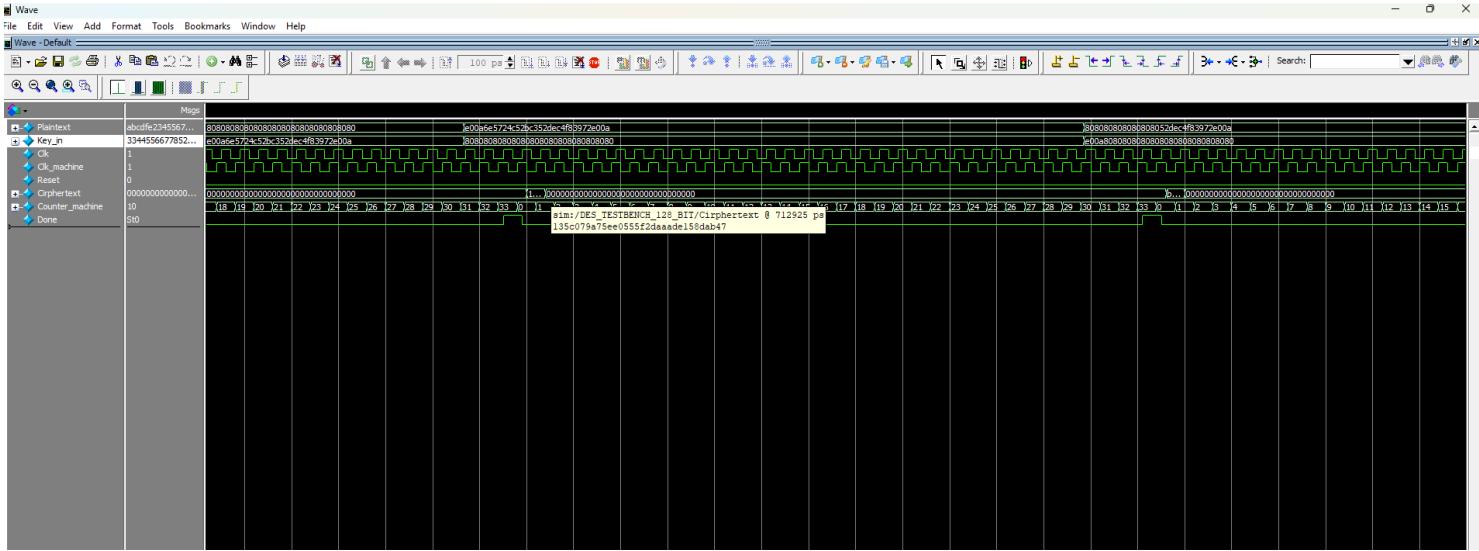
- Về yêu cầu của Pre-simulation thì chỉ cần chạy đúng logic mà không cần quan tâm đến mạch có tổng hợp được hay không, ở đây mình chỉ sử dụng 10 test case để mô phỏng chức năng của mạnh đúng hay không nhưng trong thực tế thì phải tới hàng triệu test case.
- Vì trong mạch có nhiều khối chức năng và được chia làm hai khối chính đó là khối điều khiển (state machine và khối dữ liệu), vì vậy tín hiệu điều khiển từ khối control phải có trước so với khối dữ liệu. Do đó, mình đã sử dụng 2 xung clock với cùng chu kì nhưng một clock sẽ lên 1 sớm hơn clock còn lại.

```

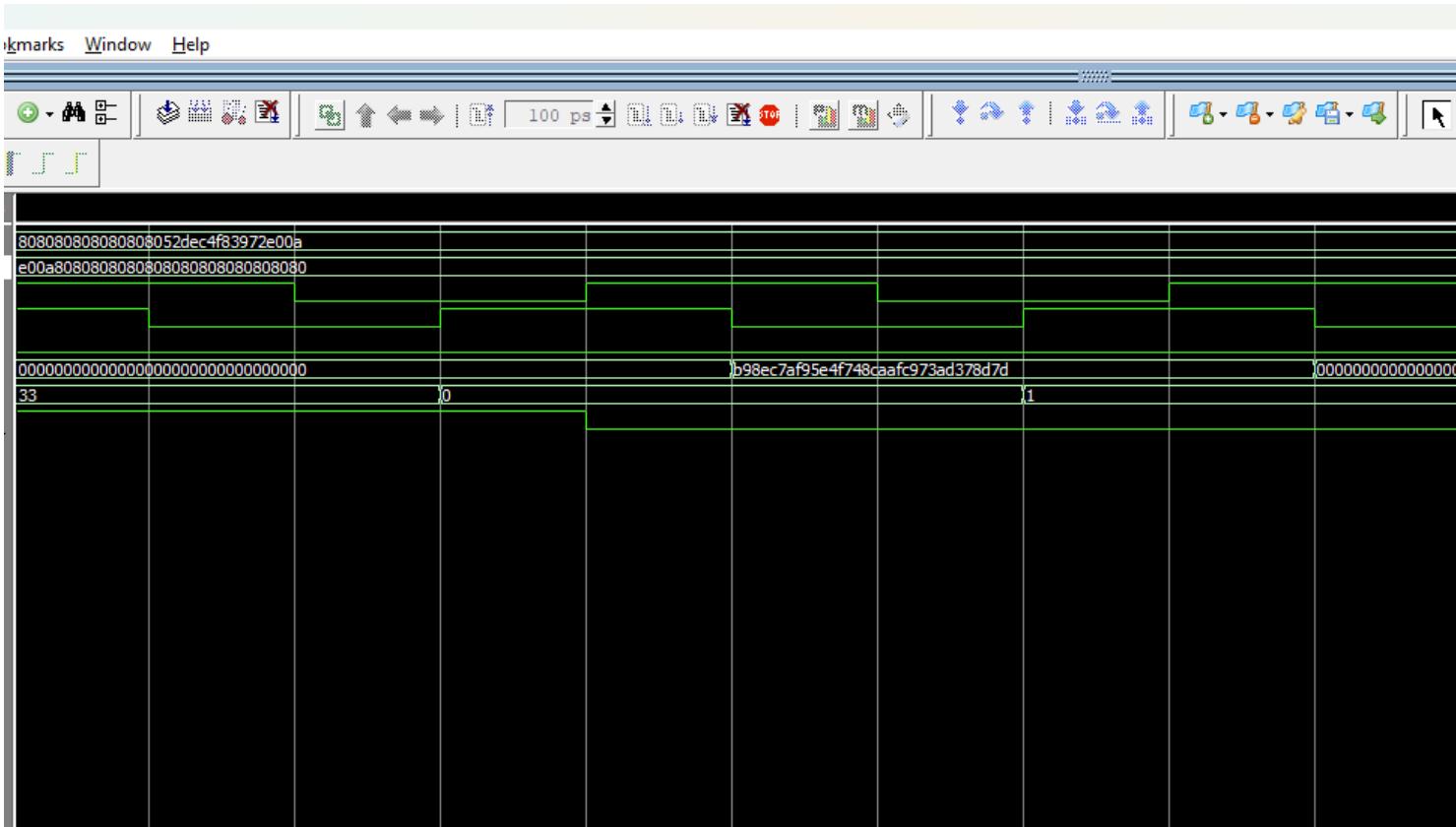
91     initial begin
92         // ===== test 1 =====
93         Reset = 1'b1;
94         Plaintext = 128'h80808080808080808080808080808080;
95         Key_in    = 128'hE00A6E5724C52BC352DEC4F83972E00A;
96         #13;
97         Reset = 1'b0;
98         //===== test 2 =====
99         #first_cycle;
100        Plaintext = 128'hE00A6E5724C52BC352DEC4F83972E00A;
101        Key_in    = 128'h80808080808080808080808080808080;
102        // ===== test 3 =====
103        #second_cycle;
104        Plaintext = 128'h8080808080808052DEC4F83972E00A;
105        Key_in    = 128'hE00A8080808080808080808080808080;
106        //===== test 4 =====
107        #second_cycle;
108        Plaintext = 128'h8080808080808052DEC4F83972E00A;
109        Key_in    = 128'hE00A6E57252DEC4F83972E00A0808080;
110        //===== test 5 =====
111        #second_cycle;
112        Plaintext = 128'h808080808080808052DEC4F83972E00A;
113        Key_in    = 128'hE00A6E5724C52BC3123456789abcdefd;
114        // ===== test 6 =====
115        #second_cycle;
116        Plaintext = 128'h0124356331234dfa52DEC4F83972E00A;
117        Key_in    = 128'h123456789aC52BC3123456789abcdefd;
118        // ===== Test 7 =====
119        #second_cycle;
120        Plaintext = 128'h8080808073241dfecab224F83972E00A;
121        Key_in    = 128'hE00A6E5724C52BC312cdefabbde03452;
122        // ===== Test 8 =====
123        #second_cycle;
124        Plaintext = 128'habcdfe2345567822cab224F83972E00A;
125        Key_in    = 128'hE00A6E5724C52BCaccdeff3344556677;
126        // ===== Test 9 =====
127        #second_cycle;
128        Plaintext = 128'habcdfe234556eeffdedbba1122334455;
129        Key_in    = 128'hE00A6E5724C52BCaccdeffdd99887766;
130        // ===== Test 10 =====
131        #second_cycle;
132        Plaintext = 128'habcdfe2345567112233445566778899a;
133        Key_in    = 128'h3344556677852BCaccdeff3344556677;
134        // =====
135        end

```

Hình 31: Các test case.

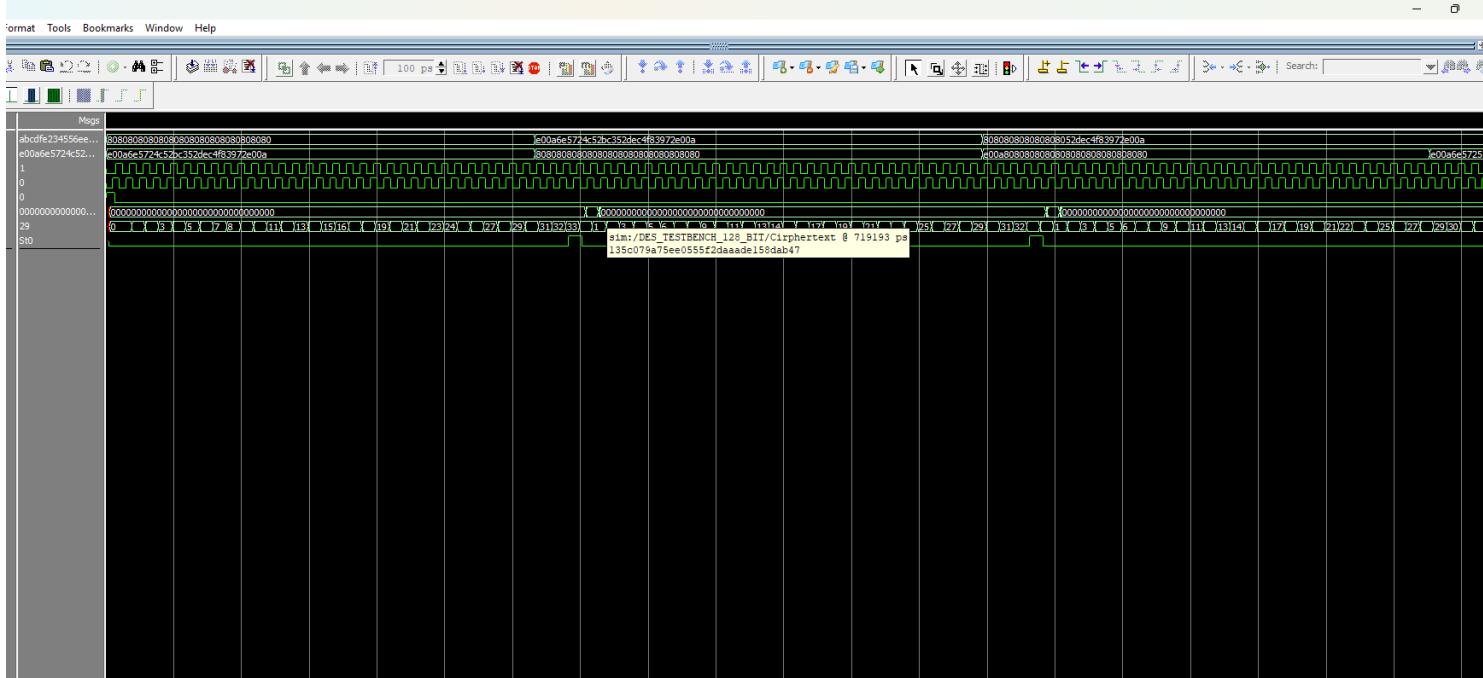


Hình 32: Kết quả mô phỏng tất cả test case.

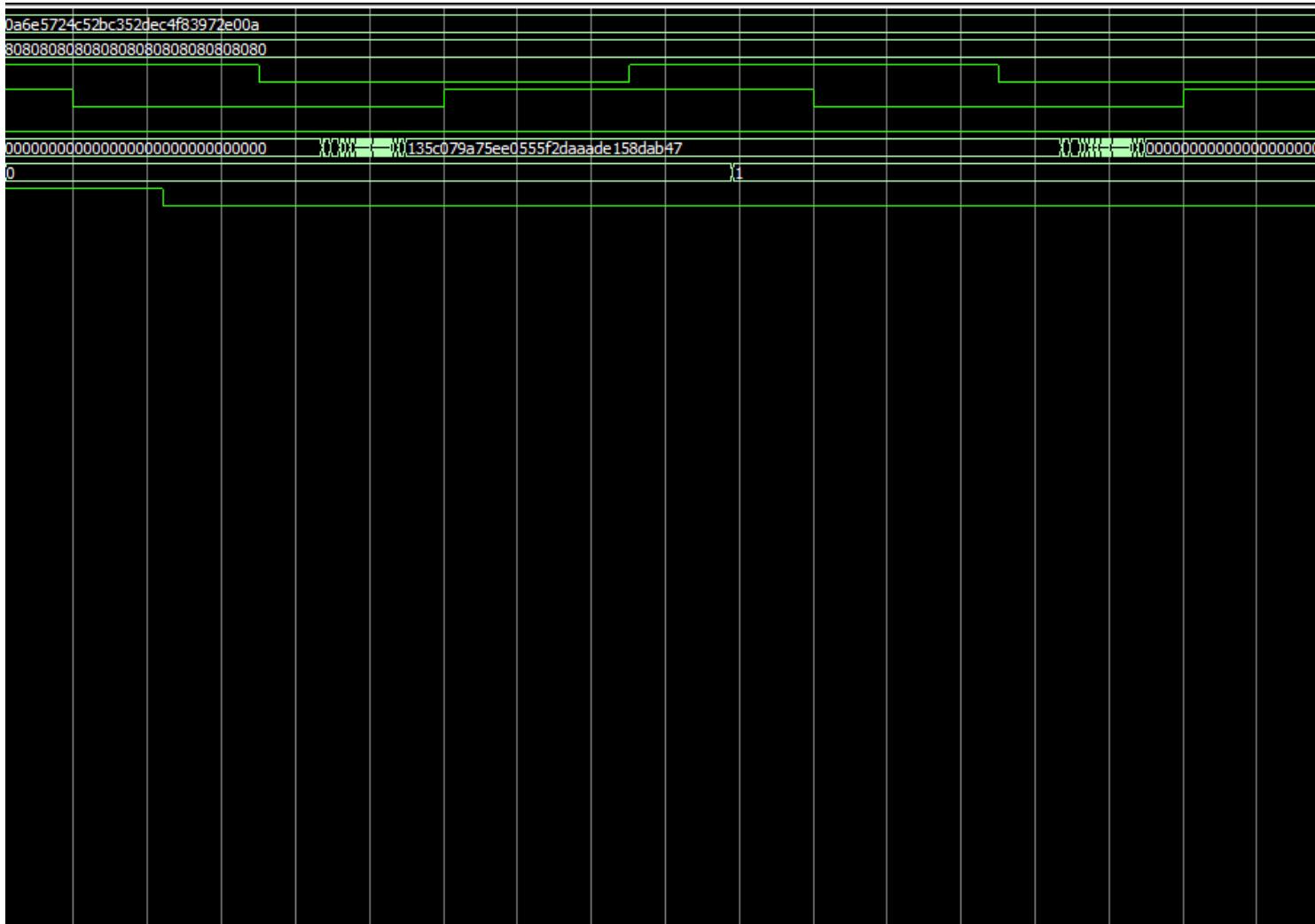


Hình 33: Kết quả mô phỏng test case đầu tiên.

2. Post-simulation.



Hình 34: Kết quả mô phỏng một số test case.



Hình 35: Kết quả mô phỏng test case đầu tiên.

II. Số lượng chu kỳ.

Theo thuật toán, chúng ta cần 32 chu kỳ để hoàn thành 32 vòng, nhưng ở thiết kế của mình thì được

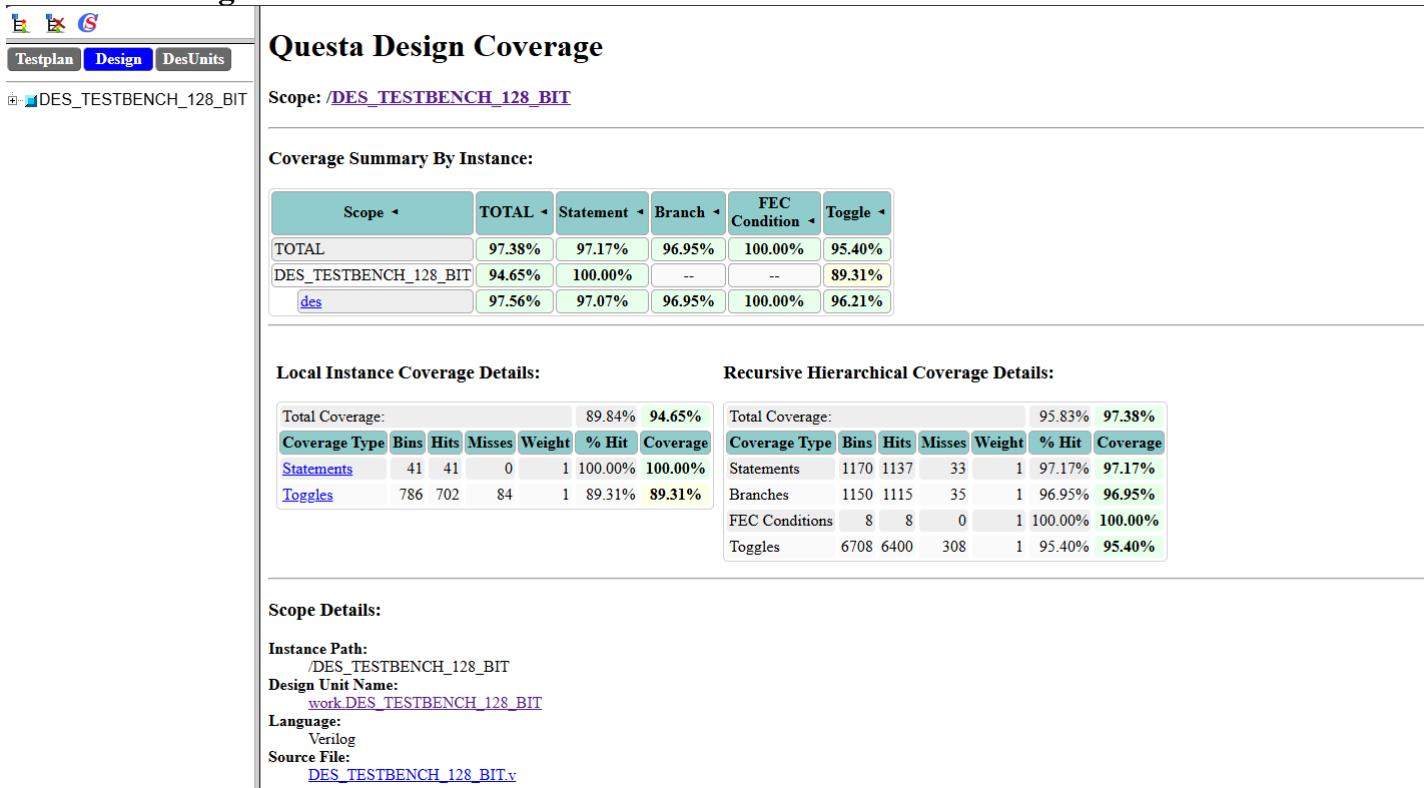
hiện thực theo cách như sau

- Chu kỳ đầu tiên (chu kỳ 0) là chu kỳ tạo sự ổn định cho mạch nên mình không làm gì
- Bắt đầu từ chu kỳ 1 thì mới thực hiện những vòng đầu tiên
- Vì cấu trúc thanh ghi được thiết kế theo dạng đọc tại cạnh lên, ghi tại cạnh xuống nên tại chu kỳ đầu chúng ta không thể đọc giá trị để tính toán được mà phải đợi tới cạnh lên chu kỳ thứ 2 (dữ liệu đã được ghi ở cạnh xuống chu kỳ thứ 1) thì mới có dữ liệu để đọc ra. Như vậy, Key1 , L1, R1 phải đến chu kỳ thứ 2 mới được tính toán xong và L32, R32, Key32 phải đến chu kỳ thứ 33 mới được tính toán xong.

-> Vậy tổng số chu kỳ để hoàn thành thuật toán là 34 chu kỳ.

III. Coverage code.

1 Total coverage code.



Hình 36: Tổng coverage code là 97,28 %

- Độ bao phủ là 97,28 % , tức là testbench đã quan được gần hết các nhánh trong Code RTL của thiết kế, còn một số trường hợp miss chủ yếu ở loại toggle pin và loại này thường rất nhiều và rất khó để test hết được.

2. Coverage code của từng file.

```

# File: DES_ALGORITHM_128BIT.v
# Enabled Coverage Active Hits Misses % Covered
# -----
# Stmtns 8 8 0 100.0
# Branches 6 6 0 100.0
# FEC Condition Terms 0 0 0 100.0
# FEC Expression Terms 0 0 0 100.0
# FSMs
# States 0 0 0 100.0
# Transitions 0 0 0 100.0
# Toggle Bins 2004 1873 131 93.4
#
# File: DES_TESTBENCH_128_BIT.v
# Enabled Coverage Active Hits Misses % Covered
# -----
# Stmtns 41 41 0 100.0
# Branches 0 0 0 100.0
# FEC Condition Terms 0 0 0 100.0
# FEC Expression Terms 0 0 0 100.0
# FSMs
# States 0 0 0 100.0
# Transitions 0 0 0 100.0
# Toggle Bins 786 702 84 89.3
#
# File: E_XOR_KEY.v
# Enabled Coverage Active Hits Misses % Covered
# -----
# Stmtns 1 1 0 100.0
# Branches 0 0 0 100.0
# FEC Condition Terms 0 0 0 100.0
# FEC Expression Terms 0 0 0 100.0
# FSMs
# States 0 0 0 100.0
# Transitions 0 0 0 100.0
# Toggle Bins 192 192 0 100.0
#
# File: F_FUNCTION.v
# Enabled Coverage Active Hits Misses % Covered
# -----
# Stmtns 0 0 0 100.0
# Branches 0 0 0 100.0
# FEC Condition Terms 0 0 0 100.0
# FEC Expression Terms 0 0 0 100.0
# FSMs
# States 0 0 0 100.0
# Transitions 0 0 0 100.0
# Toggle Bins 512 512 0 100.0
#
# File: IP.v
# Enabled Coverage Active Hits Misses % Covered
# -----
# Stmtns 5 5 0 100.0
# Branches 2 2 0 100.0
# FEC Condition Terms 0 0 0 100.0
# FEC Expression Terms 0 0 0 100.0
# FSMs
# States 0 0 0 100.0
# Transitions 0 0 0 100.0
# Toggle Bins 256 209 47 81.6

```

```

# File: IP_1.v
# Enabled Coverage Active Hits Misses % Covered
# -----
# Stmtns 3 3 0 100.0
# Branches 2 2 0 100.0
# FEC Condition Terms 0 0 0 100.0
# FEC Expression Terms 0 0 0 100.0
# FSMs
# States 0 0 0 100.0
# Transitions 0 0 0 100.0
# Toggle Bins 256 256 0 100.0
#
# File: MUX2_1.v
# Enabled Coverage Active Hits Misses % Covered
# -----
# Stmtns 1 1 0 100.0
# Branches 2 2 0 100.0
# FEC Condition Terms 0 0 0 100.0
# FEC Expression Terms 0 0 0 100.0
# FSMs
# States 0 0 0 100.0
# Transitions 0 0 0 100.0
# Toggle Bins 0 0 0 100.0
#
# File: MUX2_1_TOP.v
# Enabled Coverage Active Hits Misses % Covered
# -----
# Stmtns 1 1 0 100.0
# Branches 2 2 0 100.0
# FEC Condition Terms 0 0 0 100.0
# FEC Expression Terms 0 0 0 100.0
# FSMs
# States 0 0 0 100.0
# Transitions 0 0 0 100.0
# Toggle Bins 0 0 0 100.0
#

```

Transcript

```

# File: PC_1.v
# Enabled Coverage Active Hits Misses % Covered
# -----
# Stmtns 5 5 0 100.0
# Branches 2 2 0 100.0
# FEC Condition Terms 0 0 0 100.0
# FEC Expression Terms 0 0 0 100.0
# FSMs
# States 0 0 0 100.0
# Transitions 0 0 0 100.0
# Toggle Bins 224 201 23 89.7
#

```

```

# File: REGISTER_C0_D0.v
# Enabled Coverage Active Hits Misses % Covered
# -----
# Stmtns 9 9 0 100.0
# Branches 4 4 0 100.0
# FEC Condition Terms 0 0 0 100.0
# FEC Expression Terms 0 0 0 100.0
# FSMs
# States 0 0 0 100.0
# Transitions 0 0 0 100.0
# Toggle Bins 448 448 0 100.0
#
# File: REGISTER_TOP.v
# Enabled Coverage Active Hits Misses % Covered
# -----
# Stmtns 9 9 0 100.0
# Branches 4 4 0 100.0
# FEC Condition Terms 0 0 0 100.0
# FEC Expression Terms 0 0 0 100.0
# FSMs
# States 0 0 0 100.0
# Transitions 0 0 0 100.0
# Toggle Bins 512 512 0 100.0
#
# File: ROUND_KEY.v
# Enabled Coverage Active Hits Misses % Covered
# -----
# Stmtns 0 0 0 100.0
# Branches 0 0 0 100.0
# FEC Condition Terms 0 0 0 100.0
# FEC Expression Terms 0 0 0 100.0
# FSMs
# States 0 0 0 100.0
# Transitions 0 0 0 100.0
# Toggle Bins 1350 1327 23 98.2
#
# File: STATE_MACHINE.v
# Enabled Coverage Active Hits Misses % Covered
# -----
# Stmtns 21 21 0 100.0
# Branches 15 14 1 93.3
# FEC Condition Terms 4 4 0 100.0
# FEC Expression Terms 0 0 0 100.0
# FSMs
# States 0 0 0 100.0
# Transitions 0 0 0 100.0
# Toggle Bins 20 20 0 100.0
#

```

```

# File: S_BOX.v
#   Enabled Coverage          Active    Hits    Misses % Covered
# -----
#     Stmts                  520      520      0    100.0
#     Branches                544      544      0    100.0
#     FEC Condition Terms     0        0        0    100.0
#     FEC Expression Terms    0        0        0    100.0
#     FSMs                   100.0
#       States                0        0        0    100.0
#       Transitions            0        0        0    100.0
#     Toggle Bins              64      64      0    100.0
#
# File: XOR_FUNCTION.v
#   Enabled Coverage          Active    Hits    Misses % Covered
# -----
#     Stmts                  1        1        0    100.0
#     Branches                0        0        0    100.0
#     FEC Condition Terms     0        0        0    100.0
#     FEC Expression Terms    0        0        0    100.0
#     FSMs                   100.0
#       States                0        0        0    100.0
#       Transitions            0        0        0    100.0
#     Toggle Bins              0        0        0    100.0
#
# Total Coverage By File (code coverage only, filtered view): 98.7%

```

Hình 37: Coverage code của từng file.

CHƯƠNG 4: TỔNG KẾT VÀ ĐÁNH GIÁ QUÁ TRÌNH THỰC HIỆN

I. Tổng kết.

- Thuật toán trên là một trong những thuật toán mã hóa cơ bản trong mã hóa dữ liệu. Đồ án trên là một thiết kế ở mức mô phỏng nhưng qua đó cũng thấy được những điểm mạnh và yếu của thuật toán cũng như là sự vượt trội của ngôn ngữ verilog trong việc xử lý các tác vụ song song, từ đó tăng tốc độ thực thi của thuật toán.

II. Quá trình thực hiện của nhóm.

1. Phân công nhiệm vụ

HỌ VÀ TÊN	MSSV	NHIỆM VỤ	MỨC ĐỘ
Nguyễn Đình Anh	23520057	Phân nhiệm vụ, làm phần roundkey, tổng hợp code, viết báo cáo chương 1, 3, 4	100%
Nguyễn Hoàng Quốc Cường	23520200	F Function, slide báo cáo, viết báo cáo chương 2	100%
Bùi Tân Đạt	23520244	Làm phần I_P, State machine, tạo test case, viết báo cáo chương 2, chỉnh sửa báo cáo.	100%
Nguyễn Phạm Thiên Ân	23520015	Làm phần I_P, Round key, Tạo test case, Viết báo cáo chương 2	100%

Hình 38: Tổng kết nhiệm vụ

2. Quá trình thực hiện

- Buổi 1: Ngày 9/10/2025
- Buổi 2: Ngày 16/10/2025
- Buổi 3: Ngày 22/10/2025
- Buổi 4: Ngày 12/11/2025
- Buổi 5: Ngày 3/12/2025.

CHƯƠNG 5: TÀI LIỆU THAM KHẢO

<https://nguyenquanicd.blogspot.com/2017/08/background-thuat-toan-ma-hoa-va-giai-ma.html>

<https://nguyenquanicd.blogspot.com/2017/08/ip-core-loi-ip-ma-hoa-giai-ma-des.html>

https://www.researchgate.net/publication/337108251_Expanded_128-bit_Data_Encryption_Standard