**ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH**

TRƯỜNG ĐẠI HỌC BÁCH KHOA

KHOA ĐIỆN – ĐIỆN TỬ

**BỘ MÔN ĐIỆN TỬ**

---------------o0o---------------

****

**LUẬN VĂN TỐT NGHIỆP**

**THIẾT KẾ PHẦN CỨNG XỬ LÝ HÀM BĂM KECCAK CHO GIẢI THUẬT SHA3**

**GVHD: TS. Trần Hoàng Linh**

**SVTH: Trần Công Tiến**

**MSSV: 1810580**

**TP. HỒ CHÍ MINH, THÁNG 4 NĂM 2022**

ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM

TRƯỜNG ĐẠI HỌC BÁCH KHOA Độc lập – Tự do – Hạnh phúc.

-----✩----- -----✩-----

Số: \_\_\_\_\_\_ /BKĐT

Khoa: **Điện – Điện tử**

Bộ Môn: **Điện Tử**

N**HIỆM VỤ LUẬN VĂN TỐT NGHIỆP**

1. HỌ VÀ TÊN : Trần Công Tiến MSSV: 180580

2. NGÀNH: **ĐIỆN TỬ - VIỄN THÔNG** LỚP : DD18DV08

1. Đề tài: Thiết kế phần cứng xử lý hàm băm Keccak cho giải thuật SHA3.
2. Nhiệm vụ (Yêu cầu về nội dung và số liệu ban đầu):

.............................................................................................................................................

.............................................................................................................................................

.............................................................................................................................................

.............................................................................................................................................

.............................................................................................................................................

.............................................................................................................................................

1. Ngày giao nhiệm vụ luận văn: ...............................
2. Ngày hoàn thành nhiệm vụ: ...................................

5. Họ và tên người hướng dẫn: Phần hướng dẫn

................................................................. .....................................

................................................................. .....................................

Nội dung và yêu cầu LVTN đã được thông qua Bộ Môn.

*Tp.HCM, ngày…... tháng….. năm 20*

**CHỦ NHIỆM BỘ MÔN NGƯỜI HƯỚNG DẪN CHÍNH**

**PHẦN DÀNH CHO KHOA, BỘ MÔN:**

Người duyệt (chấm sơ bộ):.......................

Đơn vị:......................................................

Ngày bảo vệ : ...........................................

Điểm tổng kết: .........................................

Nơi lưu trữ luận văn: ...............................

***LỜI CẢM ƠN***

***Trong thời gian làm luận văn và đề cương luận văn, em đã nhận được nhiều sự giúp đỡ đóng góp ý kiến từ thầy cô, gia đình và bạn bè.***

***Em xin gửi lời cám ơn chân thành đến TS. Trần Hoàng Linh, trưởng bộ môn Điện Tử - Trường Đại học Bách Khoa thành phố Hồ Chí Minh đã tận tình hướng dẫn, chỉ bảo em trong suốt thời gian làm luận văn và đề cương luận văn, anh Vương Tuấn Hùng – người trực tiếp theo sát em trong quá trình hoàn thành đề tài.***

***Em cũng xin gửi lời cảm ơn chân thành đến các thầy cô giảng viên trường Đại học Bách Khoa thành phố Hồ Chí Minh đã giảng dạy, truyền đạt những kiến thức từ đại cương cho đến chuyên ngành.***

***Em cũng xin gửi lời cảm ơn sâu sắc đến gia đình, bạn bè đã luôn ủng hộ và hỗ trợ em rất nhiều trong quá trình theo học tại đại học.***

***Em xin gửi lời chúc sức khỏe đến thầy cô, gia đình và bạn bè!***

*Tp. Hồ Chí Minh, ngày 23 tháng 12 năm 2021 .*

**Sinh viên**

**MỤC LỤC**

[1. GIỚI THIỆU 1](#_Toc23159)

[1.1 Tổng quan 1](#_Toc17931)

[1.2 Tình hình nghiên cứu trong và ngoài nước 1](#_Toc13131)

[1.3 Mục tiêu đề tài 1](#_Toc27395)

[2. NỘI DUNG LÝ THUYẾT ĐỀ TÀI 2](#_Toc22306)

[2.1 Hàm băm và một số ứng dụng của hàm băm 2](#_Toc25572)

[2.1.1 Hàm băm 2](#_Toc26996)

[2.1.2 Ứng dụng của hàm băm 3](#_Toc14258)

[2.2 Sponge Function 4](#_Toc18653)

[2.3 Bộ đệm Pad10\*1 7](#_Toc6045)

[2.4 Hàm hoán vị Keccak 8](#_Toc711)

[2.4.1 Biến đổi θ 11](#_Toc24697)

[2.4.2 Biến đổi ρ 12](#_Toc14721)

[2.4.3 Biến đổi π 13](#_Toc10492)

[2.4.4 Biến đổi χ 14](#_Toc14342)

[2.4.5 Biến đổi ι 14](#_Toc19174)

[2.4.6 Hàm hoán vị Keccak-p[b,] 16](#_Toc2302)

[2.4.7 Hàm hoán vị Keccak-f[b] 16](#_Toc1667)

[2.4.8 Hàm Keccak[c](N,d) 16](#_Toc21180)

[2.5 Thiết kế SHA-3 bằng thuật toán Keccak 17](#_Toc6952)

[3. GIẢI PHÁP THỰC HIỆN 17](#_Toc2124)

[3.1 Module Padding 19](#_Toc17556)

[3.2 Module Step-Mapping 20](#_Toc16334)

[3.3 Module Control 22](#_Toc19550)

[4. THIẾT KẾ VÀ THỰC HIỆN PHẦN CỨNG 23](#_Toc16549)

[4.1 Yêu cầu thiết kế 23](#_Toc6105)

[4.2 Phân tích thiết kế 23](#_Toc12371)

[4.3 Sơ đồ khối tổng quát 24](#_Toc7701)

[4.4 Sơ đồ khối chi tiết và nhiệm vụ, chức năng từng khối 24](#_Toc15090)

[4.4.1 Bộ đệm ngõ vào Buffer\_in 24](#_Toc20195)

[4.4.2 Bộ VSX 26](#_Toc15525)

[4.4.3 Bộ STA 28](#_Toc31499)

[4.4.4 Bộ ATS 29](#_Toc23427)

[4.4.5 Bộ biến đổi Transformation Round 30](#_Toc3857)

[4.4.6 Bộ Trunc 32](#_Toc15800)

[4.4.7 Khối điều khiển tín hiệu Control 34](#_Toc9730)

[4.4.8 Bộ đệm ngõ vào Buffer\_in\_2 38](#_Toc22708)

[4.4.9 Khối điều khiển tín hiệu Control\_2 38](#_Toc20651)

[5. THIẾT KẾ VÀ THỰC HIỆN PHẦN MỀM 42](#_Toc29457)

[5.1 Create\_Random\_Keccak.py 44](#_Toc32170)

[5.2 Hash\_Function.py 45](#_Toc5744)

[5.3 Compare.py 46](#_Toc14493)

[6. KẾT QUẢ THỰC HIỆN 47](#_Toc2565)

[6.1 Cách thức đo đạc, thử nghiệm 47](#_Toc13748)

[6.2 Số liệu đo đạc 47](#_Toc15334)

[7. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN 49](#_Toc4802)

[8. TÀI LIỆU THAM KHẢO 49](#_Toc584)

DANH SÁCH HÌNH

[Hình 2 - 1 Cấu trúc bọt biển 5](#_Toc19293)

[Hình 2 - 2 Giải thuật bọt biển 6](#_Toc6992)

[Hình 2 - 3 Hàm absorbing function 6](#_Toc6388)

[Hình 2 - 4 Hàm Squeezing function 7](#_Toc29056)

[Hình 2 - 5 Mảng trạng thái 3 chiều 9](#_Toc26022)

[Hình 2 - 6 Các thành phần của mảng trạng thái 10](#_Toc31040)

[Hình 2 - 7 Ký hiệu chiều x, y, z của mảng trạng thái 11](#_Toc29179)

[Hình 2 - 8 Biến đổi Theta 11](#_Toc8738)

[Hình 2 - 9 Biến đổi Rho 12](#_Toc25562)

[Bảng 2 - 1 Các thông số rotate trong Rho 12](#_Toc1179)

[Bảng 2 - 2 Các thông số rotate trong Rho khi w = 64 13](#_Toc6984)

[Hình 2 - 10 Biến đổi Pi 13](#_Toc11362)

[Hình 2 - 11 Biến đổi Chi 14](#_Toc31197)

[Bảng 2 - 3 Các thông số trong từng giải thuật SHA-3 17](#_Toc20097)

[Hình 3 - 1 Sơ đồ tổng quát hàm băm SHA-3 18](#_Toc6920)

[Bảng 3 - 1 So sánh giữa các thiết kế 19](#_Toc4365)

[Bảng 3 - 2 Thông số hậu tố trong từng giải thuật băm SHA-3 20](#_Toc11105)

[Hình 3 - 2 Sơ đồ khối StepMapping trong thiết kế [1] 21](#_Toc31151)

[Hình 3 - 3 Sơ đồ khối tổng quát trong thiết kế [3] 22](#_Toc16277)

[Hình 4 - 1 SHA3-core 23](#_Toc352)

[Hình 4 - 2 Sơ đồ khối tổng quát của thiết kế 24](#_Toc20406)

[Hình 4 - 3 Sơ đồ khối khối Buffer\_in 24](#_Toc32513)

[Bảng 4 - 1 Chân kết nối trong khối Buffer\_in 25](#_Toc25069)

[Bảng 4 - 2 Bảng số Byte tương ứng mỗi MODE 26](#_Toc26549)

[Hình 4 - 4 Sơ đồ khối VSX 27](#_Toc2019)

[Hình 4 - 5 Sơ đồ khối chi tiết khối VSX 27](#_Toc6964)

[Bảng 4 - 3 Chân kết nối trong khối VSX 28](#_Toc25443)

[Hình 4 - 6 Sơ đồ khối STA 28](#_Toc23285)

[Bảng 4 - 4 Chân kết nối trong khối STA 29](#_Toc18929)

[Hình 4 - 7 Sơ đồ khối ATS 29](#_Toc14739)

[Bảng 4 - 5 Chân kết nối trong khối ATS 30](#_Toc15327)

[Hình 4 - 8 Sơ dồ khối Transformation\_round 30](#_Toc27799)

[Hình 4 - 9 Sơ đồ chi tiết khối Transformation\_round 31](#_Toc29191)

[Bảng 4 - 6 Chân kết nối trong khối Transformation\_round 31](#_Toc8090)

[Hình 4 - 10 Bảng ROM lưu giữ giá trị round\_constant dùng trong Iota 32](#_Toc3879)

[Hình 4 - 11 Sơ đồ khối Trunc 33](#_Toc483)

[Bảng 4 - 7 Chân kết nối trong khối Trunc 33](#_Toc9977)

[Hình 4 - 12 Máy trạng thái khối Control 35](#_Toc15239)

[Bảng 4 - 8 Chân kết nối trong khối Control 35](#_Toc16224)

[Bảng 4 - 9 Các trạng thái trong khối Control 36](#_Toc18552)

[Bảng 4 - 10 Các tín hiệu ngõ ra tương ứng trong khối Control 37](#_Toc9144)

[Hình 4 - 13 Máy trạng thái khối Control\_2 39](#_Toc20750)

[Bảng 4 - 11 Chân kết nối trong khối Control\_2 40](#_Toc25533)

[Bảng 4 - 12 Các trạng thái trong khối Control 41](#_Toc5069)

[Bảng 4 - 13 Các tín hiệu ngõ ra tương ứng trong khối Control 41](#_Toc25265)

[Bảng 5 - 1 Các thông số trong hàm Keccak 43](#_Toc21116)

[Hình 5 - 1 Kết quả khi test chuỗi ngõ vào trên bằng Python 43](#_Toc16548)

[Hình 5 - 2 File Create\_Random\_Keccak.py 44](#_Toc1221)

[Hình 5 - 3 Kết quả khi chạy file Create\_Random\_Keccak.py 45](#_Toc25839)

[Hình 5 - 4 File Hash\_Function.py 46](#_Toc2828)

[Hình 5 - 5 File Compare.py 47](#_Toc20355)

[Hình 6 - 1 Kết quả tính toán hàm băm bằng phần cứng và phần mềm 48](#_Toc2857)

[Hình 6 - 2 Kết quả khi so sánh hai kết quả 48](#_Toc29895)

[Hình 6 - 3 Tần số của hệ thống 48](#_Toc31530)

[Hình 6 - 4 Kết quả tổng hợp phần cứng 49](#_Toc4991)

**DANH SÁCH BẢNG**

[Bảng 2 - 1 Các thông số rotate trong Rho 12](#_Toc11071)

[Bảng 2 - 2 Các thông số rotate trong Rho khi w = 64 13](#_Toc26107)

[Bảng 2 - 3 Các thông số trong từng giải thuật SHA-3 17](#_Toc28773)

[Bảng 3 - 1 So sánh giữa các thiết kế 19](#_Toc32561)

[Bảng 3 - 2 Thông số hậu tố trong từng giải thuật băm SHA-3 20](#_Toc30394)

[Bảng 4 - 1 Chân kết nối trong khối Buffer\_in 25](#_Toc21071)

[Bảng 4 - 2 Bảng số Byte tương ứng mỗi MODE 26](#_Toc12117)

[Bảng 4 - 3 Chân kết nối trong khối VSX 27](#_Toc21291)

[Bảng 4 - 4 Chân kết nối trong khối STA 28](#_Toc19478)

[Bảng 4 - 5 Chân kết nối trong khối ATS 29](#_Toc25298)

[Bảng 4 - 6 Chân kết nối trong khối Transformation\_round 31](#_Toc27629)

[Bảng 4 - 7 Chân kết nối trong khối Trunc 33](#_Toc14318)

[Bảng 4 - 8 Chân kết nối trong khối Control 35](#_Toc30387)

[Bảng 4 - 9 Các trạng thái trong khối Control 36](#_Toc21206)

[Bảng 4 - 10 Các tín hiệu ngõ ra tương ứng trong khối Control 36](#_Toc14027)

[Bảng 4 - 11 Các thông số trong hàm Keccak 38](#_Toc30661)

# GIỚI THIỆU

## Tổng quan

Trong lĩnh vực nghiên cứu về mã hóa (crypto), hàm băm là một trong những cốt lõi của lĩnh vực này. Hàm băm là một giải thuật nhằm sinh ra giá trị băm tương ứng với mỗi khối dữ liệu, ngõ ra của hàm băm là một chuỗi ký tự có độ dài cố định và chuỗi ký tự này là duy nhất gọi là “hash”. Vì những tính chất đặc biệt, hàm băm được ứng dụng nhiều, nhất là ở lĩnh vực mã hóa: chữ ký số, mã hóa lượng tử, …Một ví dụ về ứng dụng nổi tiếng của hàm băm hiện này là SHA2-256 sử dụng trong lĩnh vực Blockchain.   
 Một số hàm băm đã xuất hiện trước đây có thể kể đến như là MD2, MD4, MD5, SHA1, SHA2,… Trong đó 2 hàm băm khá thông dụng là SHA1. Tuy nhiên vào năm 2005, SHA1 đã bị tìm ra lỗi bảo mật và đến nay không còn đảm bảo an toàn nữa. Hiện nay vẫn chưa có cuộc tấn công mã hóa nào có thể phá được tính bảo mật của SHA2 nên hiện nay hàm băm này được sử dụng phổ biến.  
 SHA-3 được NIST phát hành vào ngày 5 tháng 8 năm 2015. Đây có lẽ là tiêu chuẩn hàm băm mới nhất cho đến hiện nay. SHA-3 là một tập con của họ nguyên thủy mật mã rộng hơn là Keccak. Thuật toán Keccak được đưa ra bởi Guido Bertoni, Joan Daemen, Michael Peeters và Gilles Van Assche. Keccak dựa trên cấu trúc bọt biển (sponge). Cấu trúc này cũng có thể được sử dụng để xây dựng các nguyên thủy mã hóa khác như các hệ mật mã dòng. SHA-3 cũng có các kích cỡ đầu ra tương tự như SHA-2 bao gồm: 224, 256, 384 và 512 bit.

## Tình hình nghiên cứu trong và ngoài nước

Dù là một tiêu chuẩn hàm băm mới xuất hiện và hiện tại việc triển khai SHA3 thay

thế cho SHA2 là vẫn chưa được rộng rãi, các thiết kế phần cứng của SHA3 đã được triển khai từ lâu. Bởi vì tầm quan trọng và tiện ích của việc xử lý hàm băm trên phần cứng, nhiều thiết kế hàm băm này đã được đề xuất và thường thì có hai hướng chính trong đó một là tối ưu diện tích thiết kế phần cứng và hướng thứ hai là tăng tốc độ xử lý.

Trong thiết kế [4], nhóm tác giả đã đề xuất kiến trúc pipeline cho Keccak nhằm tăng tốc độ xử lý. Thiết kế được mô phỏng bằng Modelsim và được thực hiện trên Virtex -5.

Trong thiết kế [5], nhóm tác giả đã đề xuất kiến trúc cho Keccak giúp tiết kiệm được phần cứng đến 16% các tài nguyên cổng logic. Thiết kế hoạt động với tần số là 301.02MHz đồng thời throughput là 30.1Mbps/slice. Tuy nhiên thiết kế chỉ thực hiện với SHA3-512.

## Mục tiêu đề tài

Nội dung 1: Tìm hiểu lý thuyết về hàm băm và một số ứng dụng của hàm băm.

Nội dung 2: Tìm hiểu lý thuyết về hàm bọt biển “Sponge Function”.

Nội dung 3: Tìm hiểu lý thuyết về hàm hoán vị Keccak.

Nội dung 4: Thiết kế hàm băm SHA3 bằng thuật toán Keccak.

\* Hỗ trợ 4 đầu ra phổ biến là 224bit, 256bit, 384bit, 512bit theo chuẩn FIPS 202.

\* Hỗ trợ hai giải thuật SHAKE-256 và SHAKE-512 với ngõ ra không cố định.

# NỘI DUNG LÝ THUYẾT ĐỀ TÀI

**2.1 Hàm băm và một số ứng dụng của hàm băm**

**2.1.1 Hàm băm**

Hàm băm là giải thuật nhằm sinh ra các giá trị băm tương ứng với mỗi khối dữ liệu (có thể là một chuỗi ký tự, một đối tượng trong lập trình hướng đối tượng, …). Mỗi giá trị tạo ra là duy nhất nhưng về lý thuyết vẫn có thể xảy ra hiện tượng trùng khóa hay còn gọi là đụng độ. Người ta vẫn chấp nhận nó và cố gắng cải thiện giải thuật để giảm thiểu sự đụng độ đó. Hàm băm thường được dùng để xây dựng bảng băm nhằm giảm chi phí tính toán khi tìm một khối dữ liệu trong một tập hợp nhờ việc so sánh các giá trị băm nhanh hơn việc so sánh những khối dữ liệu có kích thước lớn.

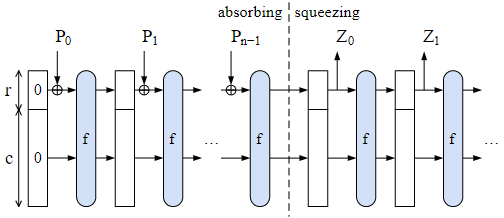
Hàm băm được ví như là trái tim của công nghệ mã hóa. Tầm quan trọng của hàm băm lần đầu tiên được nhận ra với việc phát minh ra mật mã khóa công khai (PKC) của Diffie và Hellman [Whitfield Diffie and Martin Hellman. New Directions in Cryptography] vào năm 1976 và trở thành một phần không thể thiếu của PKC từ đó. Thật không may, những tiến bộ gần đây trong phân tích mật mã đã bộc lộ những yếu điểm cố hữu trong hầu hết các hàm băm dẫn đến một cuộc nghiên cứu vào sâu hơn trong lĩnh vực này. Có hai phương pháp tiếp cận chính đó là cải tiến từ các công trình cũ hoặc là tiến hành xây dựng các giải thuật hàm băm khác [Cryptographic Hash Functions: Recent Design Trends and Security Notions]. Các hàm băm về cơ bản là hàm một chiều, chúng ánh xạ các đầu vào có giá trị tùy ý thành một chuỗi có độ dài cố định và đầu vào thường lớn hơn so với đầu ra do đó dẫn đến hiện tượng xung đột tức là hai giá trị ngõ vào cùng cho ra một giá trị hàm băm mà đã được nhắc đến ở trên. Một hàm băm tốt cần phải đảm bảo thỏa mãn các điều kiện sau:  
 - Tính toán nhanh với bất cứ thông tin ngõ vào nào.  
 - Không thể đảo ngược quá trình tạo ra giá trị băm đã cho.   
 Ví dụ: mã hash sử dụng giải thuật SHA-256 của đoạn mã vào “Hello” có giá trị là   
0x185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969, tuy nhiên việc có được giá trị hàm băm là chuỗi mã hex trên thì không thể suy ra được đoạn mã ngõ vào là “Hello”.  
 - Không thể tìm thấy được hai ngõ vào cùng cho ra một giá trị băm.  
 Ví dụ: Cũng như ví dụ trên, với cùng một chuỗi mã hex là 0x185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969 không thể tìm được chuỗi thứ hai nào khác cho cùng giá trị hash trên ngoài đoạn mã “Hello”.  
 - Chỉ một thay đổi nhỏ ở ngõ vào cũng sẽ tạo ra một giá trị băm hoàn toàn không liên quan đến giá trị băm cũ.  
 Ví dụ: hai đoạn mã sau “Hello” và “Helo” sẽ cho ra hai giá trị băm (sử dụng giải thuật SHA-256) lần lượt như sau:  
 0x185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969 và   
 0x375738319e86099fe081fabee238c40d6f038959da383c99ca3fe146e5cc8b7e.  
Hai mã hash trên hoàn toàn không liên quan gì với nhau mặc dù chỉ có một thay đổi nhỏ ở ngõ vào.  
 Từ đoạn mã “Hello” ban đầu sau quá trình băm thì trở thành một chuỗi số mà chúng ta nhìn vào sẽ không thể hiểu được. Điều này có vẻ giống với việc mã hóa (encrypt) nhưng khác biệt ở chỗ, ở mã hóa ta có thể giải mã được từ chuỗi số khó hiểu đó còn ở hàm băm thì không như tính chất đã trình bày ở trên. Điều này giúp cho hàm băm có ứng dụng đặc biệt trong công nghệ mã hóa và trở thành một phần không thể thiếu của công nghệ này.

**2.1.2 Ứng dụng của hàm băm**

Các hàm băm được ứng dụng trong nhiều lĩnh vực, chúng thường được thiết kế để phù hợp với từng ứng dụng. Ngoài ra, chúng cũng có thể được sử dụng như các hàm băm thông thường, để lập chỉ mục dữ liệu trong bảng băm, lấy đặc trưng của dữ liệu, phát hiện dữ liệu trùng lặp hoặc làm tổng kiểm tra để phát hiện lỗi các dữ liệu ngẫu nhiên.   
 - Trong khoa học máy tính, bảng băm là một cấu trúc dữ liệu sử dụng hàm băm để ánh xạ từ giá trị xác định, được gọi là khóa đến giá trị tương ứng. Do đó bảng băm là một bảng kết hợp. Hàm băm được sử dụng dụng để chuyển đổi từ khóa thành chỉ số (giá trị băm) trong mảng lưu trữ các giá trị tìm kiếm [wiki]. Bảng băm là một ứng dụng quan trọng của hàm băm.   
 - Hàm băm đảm bảo tính toàn vẹn và xác thực của thông tin. Việc so sánh các giá trị băm (digest) sau khi truyền đi có thể xác định được xem có bất kỳ thay đổi nào được thực hiện đối với dữ liệu ban đầu hay không. Một khi kiểm tra thấy được có sự thay đổi ở giá trị băm có nghĩa là thông tin đã có sự thay đổi (dựa vào tính chất của hàm băm) và ta sẽ biết được. Các hàm băm MD5, SHA-1 hoặc SHA-2 đôi khi được sử dụng trên các trang web hoặc diễn đàn để cho phép xác minh tính toàn vẹn cho các tệp được tải xuống. Phương pháp này thiết lập một chuỗi tin cậy miễn là các hàm băm được đăng trên một trang web đáng tin cậy - được xác thực bởi HTTPS. Sử dụng hàm băm mật mã và chuỗi tin cậy sẽ phát hiện các thay đổi độc hại đối với tệp. Các mã phát hiện lỗi không liên quan đến mật mã như kiểm tra dự phòng theo chu kỳ chỉ ngăn chặn các thay đổi không độc hại của tệp, vì một hành vi giả mạo có chủ đích có thể dễ dàng được tạo ra để có giá trị mã va chạm.  
 - Tạo và xác minh chữ ký số. Hầu như tất cả các lược đồ [chữ ký số](https://bitcoinvietnamnews.com/digital-signature-la-gi" \t "https://nacis.gov.vn/nghien-cuu-trao-doi/-/view-content/213743/_blank) đều yêu cầu tính toán bản tóm lược của thông điệp bằng các hàm băm mật mã. Điều này cho phép việc tính toán và tạo chữ ký được thực hiện trên một khối dữ liệu có kích thước tương đối nhỏ và cố định thay vì trên toàn bộ văn bản dài. Tính chất toàn vẹn thông điệp của hàm băm mật mã được sử dụng để tạo các lược đồ chữ ký số an toàn và hiệu quả.   
 - Việc sử dụng hàm băm để xác minh mật khẩu là hiệu quả hơn nhiều so với việc sử dụng dạng văn bản (plain text) hay là dạng mã hóa (encryption). Nếu ta sử dụng plaint text để lưu dữ mật khẩu, rõ ràng nếu như một hacker tìm được plain text đó, họ có thể dễ dàng biết được mật khẩu vì plaint text là ở dạng văn bản có thể đọc được. Nếu sử dụng encryption, thì như đã trình bày ở trên, nếu hacker tìm ra được khóa để có thể giải mã ngược lại thông điệp đó thì cũng sẽ biết được mật khẩu dẫn đến ta bị mất thông tin. Còn sử dụng hàm băm thì khác, nếu hacker có được giá trị băm thì cũng không thể nào truy ngược lại giá trị văn bản ban đầu bởi vì tính chất của hàm băm đo đó việc bảo mật mật khẩu sẽ an toàn hơn.

**2.2 Sponge Function**

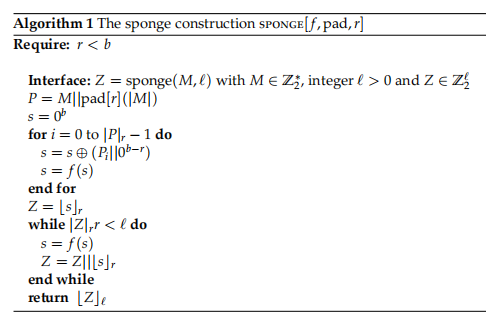
Sponge Function hay Sponge Construction là hàm bọt biển hay cấu trúc bọt biển, là một loại thuật toán mà có số trạng thái hữu hạn ở bên trong, nhận đầu vào có độ dài bất kỳ và tạo ra đầu ra có độ dài mà ta mong muốn. Hàm bọt biển khởi tạo cấu trúc bọt biển, là cấu trúc lặp đi lặp lại đơn giản, xây dựng một hàm tạo ra đầu ra có độ dài cố định dựa trên việc hoán vị một chuỗi độ dài cố định. Với giao diện này, một hàm bọt biển cũng có thể được sử dụng như một mật mã dòng (stream cypher) hoặc như là một hàm băm, do đó hàm bọt biển bao gồm một loạt các chức năng với các hàm băm và mật mã dòng. Cấu trúc bọt biển đóng vai trò quan trọng trong việc thiết kế thuật toán Keccak.



Hình 2 - 1 Cấu trúc bọt biển

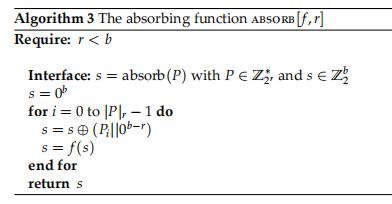
Cấu trúc bọt biển là một cấu trúc lặp đi lặp lại liên tục để xây dựng một hàm F với đầu vào tùy ý thành đầu ra có độ dài cố định dựa trên phép hoán vị độ dài cố định với độ dài cố định là b. Ở đây b được gọi là chiều rộng (width). Cấu trúc bọt biển hoạt động trên trạng thái b = r + c bit. Giá trị r được gọi là tốc độ bit (bitrate) và giá trị c là dung lượng (capacity). Các bit c cuối cùng của trạng thái không bao giờ bị ảnh hưởng trực tiếp bởi các khối đầu vào và không bao giờ được xuất ra trong giai đoạn ép.

Một hàm bọt biển hay cấu trúc bọt biển được xây dựng từ 3 thành phần chính:   
 \* Một mảng trạng thái bao gồm b bit trong đó gồm 2 phần là r (bitrate) và c (capacity) đã trình bày ở trên.  
 \* Phép hoán vị để biến đổi mảng trạng thái này thành mảng trạng thái khác.  
 \* Hàm đệm P để thêm các bit vào đầu vào để hình thành một mảng trạng thái có độ dài là bội số của b thì mới có thể đưa vào hàm bọt biển.



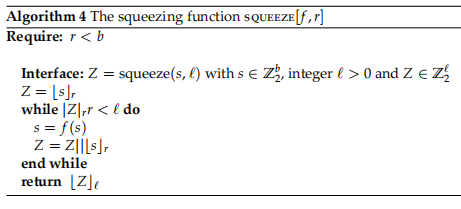
Hình 2 - 2 Giải thuật bọt biển

Đầu tiên, tất cả các bit của trạng thái được khởi tạo bằng không. Thông điệp đầu vào được đệm và cắt thành các khối r bit. Sau đó, quá trình xây dựng bọt biển được tiến hành theo hai giai đoạn: giai đoạn hấp thụ (absorbing) tiếp theo là giai đoạn ép (squeezing).  
 \* Trong giai đoạn hấp thụ, những khối r-bit đầu vào được XOR với khối r-bit trong trạng thái, xen kẽ với hàm hoán vị f. Khi tất cả đầu vào đều được xử lý, cấu trúc bọt biển sẽ chuyển sang giai đoạn ép. Figure 3 chỉ ra giải thuật cho giai đoạn hấp thụ.



Hình 2 - 3 Hàm absorbing function

\* Trong giai đoạn ép, r bit đầu tiên của trạng thái được trả về dưới dạng các khối đầu ra, xen kẽ với hàm hoán vị f, nếu như số lượng khối đầu ra lớn hơn r - do người dùng tùy ý chọn thì hoán vị f sẽ được sử dụng để tạo ra các bit tiếp theo. Figure 4 chỉ ra giải thuật cho giai đoạn ép.



Hình 2 - 4 Hàm Squeezing function

Với bộ ba thành phần: phép hoán vị f, phép đệm Pad, và bitrate r như đã trình bày ở trên, ta có thể xây dựng được một cấu trúc bọt biển.   
Thuật toán bọt biển SPONGE[f, Pad, r](N,d) được trình bày như sau:  
 **Input:** Chuỗi bit đầu vào N và số nguyên không âm d là chiều dài của output.  
 **Output:** Chuỗi bit Z trong đó len(Z) = d.  
 Các bước thực hiện như sau:  
 1. Đặt P = N || Pad(r, len(N)).  
 2. Đặt n = len(P) / r.  
 3. Đặt c = b - r.  
 4. Đặt được tách ra từ P = .  
 5. Đặt S = .  
 6. Cho i chạy từ 0 đến n-1, đặt S = .  
 7. Đặt Z là một chuỗi rỗng.  
 8. Đặt Z = Z || : Trong đó là cắt r bit trong chuỗi S.  
 9. Nếu d ≤ |Z|, ta sẽ trả về kết quả là , nếu không thì sẽ tiếp tục bước 10.  
 10. Đặt S = f(S), quay lại bước 8.

d quyết định chiều dài của ngõ ra chứ không ảnh hưởng đến giá trị và thứ tự các bit trong chuỗi của chúng. Thực tế thì d có thể là một số vô hạn và vòng lặp chỉ kết thúc khi đạt đến độ dài ngõ ra mong muốn.

**2.3 Bộ đệm Pad10\*1**

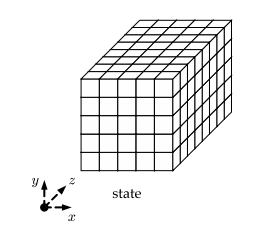
Ta đã biết được nguyên lý hoạt động của hàm bọt biển, nhận đầu vào có độ dài bất kỳ và cho kết quả ngõ ra có độ dài cố định. Để làm được điều đó, ta cần phải chia đầu vào thành các khối để xử lý. Trong khi ngõ vào có độ dài không cố định, ta cần phải có thêm một bộ đệm để thêm vào những bit bổ sung để tạo thành một khối có đủ độ dài để xử lý. Bộ đệm sử dụng trong Keccak là Multi-rate padding hay còn được ký hiệu là pad10\*1 được thực hiện khá đơn giản đó là thêm bit 1 sau chuỗi ngõ vào, sau đó là một chuỗi các bit 0 và kết thúc bằng một bit 1. Số bit 0 sẽ tùy thuộc vào độ dài của chuỗi đầu vào ban đầu mà tạo thành một chuỗi đầu vào sau khi qua bộ đệm có độ dài là bội số của giá trị bitrate r. Pad10\*1 thêm vào tối thiểu là 2 bit và tối đa là tối đa là b + 1 bit. Thuật toán Pad10\*1(x, m) được biểu diễn như sau:  
 **Input:** số nguyên dương x và số nguyên không âm m.  
 **Output:** chuỗi P mà m + len(P) là bội số của x.  
 Các bước thực hiện như sau:  
 1. Đặt j = ( - m -2 ) mod x.  
 2. Trả lại kết quả P = 1 || || 1.  
 Số bit 0 không cố định sẽ tạo ra được chiều dài P cần thiết để thực hiện tính toán trong thuật toán bọt biển.

**2.4 Hàm hoán vị Keccak**

Hàm băm thiết kế theo thuật toán Keccak sử dụng cấu trúc bọt biển như đã trình bày ở 2.2, cấu trúc này sử dụng các hoán vị lặp dựa trên nguyên lý lặp đi lặp lại các biến đổi tuyến tính và phi tuyến. Theo đó biến đổi phi tuyến cung cấp tính xáo trộn cho các bit được xử lý qua hàm vòng, còn biến đổi tuyến tính sẽ đảm đương nhiệm vụ khuếch tán rộng hơn tính xáo trộn này. Việc sử dụng đơn lẻ hai tính chất này sẽ không mang lại hiệu quả trong các thiết kế mật mã. Chúng chỉ mang lại hiệu quả khi được kết hợp với nhau. Keccak là hàm băm đã chiến thắng trong cuộc thi tuyển chọn hàm băm SHA-3 do NIST - tổ chức an ninh mật mã Hoa Kỳ tổ chức. Nguyên lý thiết kế của nó cũng dựa trên nguyên tắc trên. Hàm vòng của nó có dạng theo tiêu chuẩn FIPS 202 như sau:  
   
 Trong đó, tầng tuyến tính của nó là kết hợp bởi một số thành phần tuyến tính như phép biển đổi Theta (phép ), biến đổi Pi (phép ), biến đổi Rho (phép ) và phép biến đổi Iota (phép ). Còn biến đổi phi tuyến được đảm bảo bởi phép biến đổi Chi (phép χ). Mỗi phép biến đổi trên được gọi là một phép ánh xạ (step mapping). Chi tiết sẽ được trình bày ở 2.4.1 - 2.4.5.

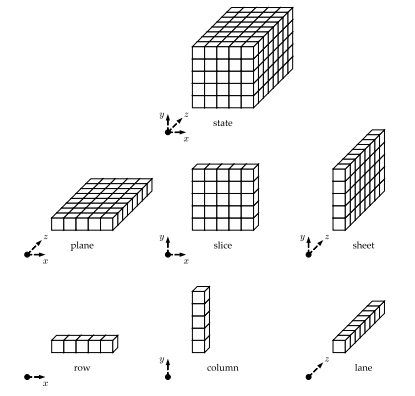
Trạng thái của một mảng các bit được cập nhật liên tục trong quá trình xử lý. Đối với phép hoán vị Keccak-p, trạng thái được biểu diễn bằng một chuỗi (S) hoặc là một mảng 3 chiều (A). Phép hoán vị Keccak-p gồm hai thông số đó là b là chiều rộng (width) đã trình bày ở trên và gọi là số lần lặp của phép hoán vị, ký hiệu là Keccak-p[b,]. Trong đó b có thể là một trong số các số sau {25,50,100,200,400,800,1600} và là bất kỳ số nguyên dương nào. Bản đặc tả thông số kỹ thuật trong bộ tiêu chuẩn SHA3 bao gồm thêm hai đại lượng khác liên quan đến b là b/25 và lần lượt ký hiệu là w và *l* trong đó ,  
*l* thuộc {0, 1, 2, 3, 4, 5, 6}.  
 \* Nếu S ký hiệu là một chuỗi biểu diễn trạng thái thì các bit của nó được đánh số từ 1 đến b-1, do đó   
 S = S[0]||S[1]||S[2]||…S[b-1]

Nếu biểu diễn trạng thái đó dưới dạng một mảng A 3 chiều có kích thước 5 x 5 x w bit khi đó các chỉ số thỏa mãn 0 ≤ x ≤ 4, 0 ≤ y ≤ 4, 0 ≤ z ≤ (w - 1). Trong đó, tích 5 x 5 x w = b chính là số bit của mảng A. Figure 5 biểu diễn mảng trạng thái A trong trường hợp b = 200 do đó w = 8.



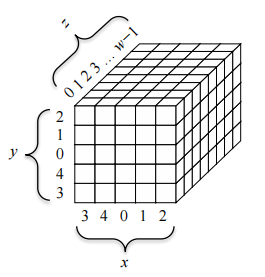
Hình 2 - 5 Mảng trạng thái 3 chiều

Mảng trạng thái A là một mảng 3 chiều trong đó các mảng con hai chiều được tạo thành từ A gồm các mảng *sheet, plane và slice* ; các mảng con một chiều được tạo thành từ A gồm các mảng *column, row và lane,* trong đó:  
 \* *sheet:* là mảng con hai chiều gồm b/5 bit theo trục tọa độ x cố định. *\* plane:* là mảng con hai chiều gồm b/5 bit theo trục tọa độ y cố định. *\* slice:* là mảng con hai chiều gồm 25 bit theo trục tọa độ z cố định. *\*column:* là mảng con một chiều gồm 5 bit theo trục tọa độ x và z cố định. *\* row:* là mảng con một chiều gồm 5 bit theo trục tọa độ y và z cố định. *\* lane:* là mảng con một chiều gồm b/25 bit theo trục tọa độ x và y cố định.



Hình 2 - 6 Các thành phần của mảng trạng thái

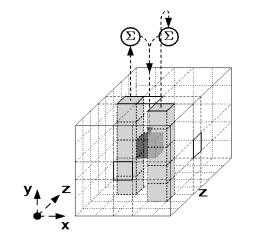
Để biểu diễn một chuỗi trạng thái từ mảng trạng thái 3 chiều, ta định nghĩa như sau:  
 Đối với mỗi cặp số nguyên (i,j) sao cho 0 ≤ i ≤ 4, 0 ≤ i ≤ 4, xác định chuỗi *lane*[i,j]:  
 *lane*[i,j] = A[i,j,0]|||A[i,j,1]||A[i,j,2]||…||A[i,j,w-1].  
 Đối với mỗi số nguyên i sao cho 0 ≤ j ≤ 4, xác định *plane*[j]:   
 *plane*[j] = *lane*[0,j]||*lane*[1,j]||*lane*[2,j]||*lane*[3,j]||*lane*[4,j].  
 Do đó,  
 S = *plane*[0]||*plane*[1]||*plane*[2]||*plane*[3]||*plane*[4].  
 Để biểu diễn một mảng trạng thái từ chuỗi trạng thái ta định nghĩa như sau: đối với mọi bộ ba (x,y,z) sao cho 0 ≤ x ≤ 4, 0 ≤ y ≤ 4, 0 ≤ z ≤ (w - 1), ta có:   
 A[x,y,z] = S[w(5y+x) + z]  
 Trong đó quy ước tọa độ (x,y) = (0,0) nằm ở trung tâm của *slice* như sau:



Hình 2 - 7 Ký hiệu chiều x, y, z của mảng trạng thái

Như đã trình bày ở phần trên, hàm vòng trong Keccak được biểu diễn bằng 5 phép ánh xạ trong đó ngoại trừ phép ι có thêm một ngõ vào thứ hai là , các phép còn lại có ngõ vào là một mảng trạng thái ba chiều và ngõ ra cũng là một mảng trạng thái ba chiều đã được biến đổi.

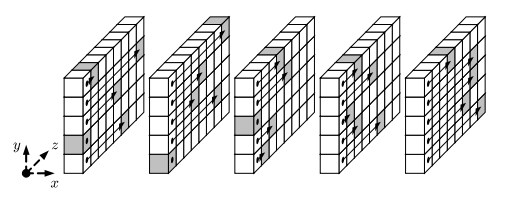
**2.4.1 Biến đổi θ**



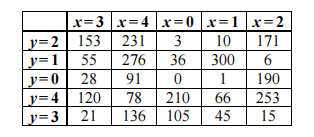
Hình 2 - 8 Biến đổi Theta

**Input:** Mảng trạng thái A  
 **Output:** Mảng trạng thái A’  
 Các bước biến đổi như sau:  
 1. Với tất cả các cặp (x, z) với 0 ≤ x ≤ 4 và 0 ≤ z ≤ (w-1)  
   
 2. Với tất cả các cặp (x, z) với 0 ≤ x ≤ 4 và 0 ≤ z ≤ (w-1)  
   
 3. Với tất cả các bộ ba (x, y, z) với 0 ≤ x ≤ 4, 0 ≤ y ≤ 4, 0 ≤ z ≤ (w-1)  
   
 Minh họa phép biển đổi θ như trong Figure 8.

**2.4.2 Biến đổi ρ**

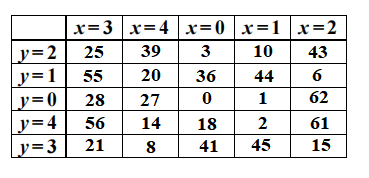


Hình 2 - 9 Biến đổi Rho

**Input:** Mảng trạng thái A  
 **Output:** Mảng trạng thái A’  
 Các bước biến đổi như sau:  
 1. Với tất cả z với 0 ≤ z ≤ (w-1), ta đặt A’[0, 0, z] = A[0, 0, z].  
 2. Đặt (x, y) = (1, 0).  
 3. Cho t chạy từ 0 đến 23:  
 a. Với tất cả z thỏa mãn 0 ≤ z ≤ (w-1) ta đặt   
 A’[x, y, z] = A[x, y, (z-(t+1)(t+2)/2) mod w]  
 b. Đặt [x, y] = [y, (2x + 3y) mod 5].  
 4. Trả lại kết quả A’.  
 Tác động của biến đổi ρ làm xoay các bit của từng *lane* theo 1 chiều dài gọi là *offset,* với việc phụ thuộc vào các tọa độ cố định của x và y trong *lane.* Tương ứng với từng bit trong *lane* tọa độ z được sửa đổi bằng cách cộng modulo các *offset* theo kích thước *lane.* 

Bảng 2 - 1 Các thông số rotate trong Rho

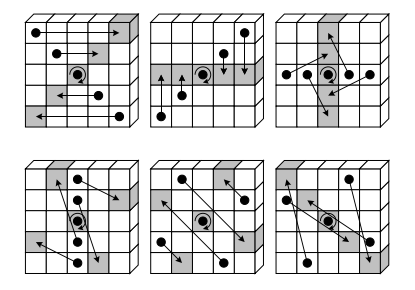
Trong hàm băm Keccak sử dụng để xây dựng SHA-3, w = 64 nên ta xây dựng được Bảng 2 - 2 chứa các thông số rotate sau:



Bảng 2 - 2 Các thông số rotate trong Rho khi w = 64

Minh họa phép biến đổi ρ với w = 8 được biểu diễn trong Figure 9 và các thông số *offset* được tính sẵn như trong Bảng 2 - 2.

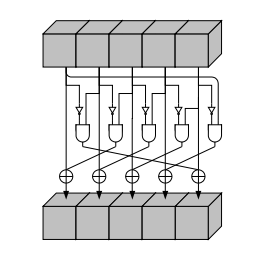
**2.4.3 Biến đổi π**



Hình 2 - 10 Biến đổi Pi

**Input:** Mảng trạng thái A  
 **Output:** Mảng trạng thái A’  
 Các bước biến đổi như sau:  
 1. Với tất cả bộ ba (x, y, z) thỏa mãn điều kiện 0 ≤ x ≤ 4, 0 ≤ y ≤ 4, 0 ≤ z ≤ (w-1), đặt:  
   
 2. Trả lại kết quả A’.  
 Biến đổi π thực chất là phép hoán vị các bit trên một *slice* của một trạng thái. Việc hoán vị này là giống nhau cho toàn bộ w *slice* trong mảng trạng thái. Như vậy có thể ghép tất cả *slice* này và thực hiện hoán vị *lane* trong khối trạng thái. Figure 10 mô tả tác động của biến đổi π lên các bit của mảng trạng thái.

**2.4.4 Biến đổi χ**



Hình 2 - 11 Biến đổi Chi

**Input:** Mảng trạng thái A  
 **Output:** Mảng trạng thái A’  
 Các bước biến đổi như sau:  
 1. Với tất cả những bộ 3 (x, y, z) thỏa mãn những điều kiện 0 ≤ x ≤ 4, 0 ≤ y ≤ 4,   
0 ≤ z ≤ (w-1), đặt:  
   
 2. Trả lại kết quả A’.  
 Figure 11 biểu diễn tác động của biến đổi χ đến mảng trạng thái A.

**2.4.5 Biến đổi ι**

Biến đổi ι chỉ tác động lên *lane* gốc, nghĩa là *lane* có tọa độ x = 0, y = 0. Bản chất của nó là cộng vào *lane* gốc các hằng số phụ thuộc vào chỉ số vòng của hoán vị. Phép biến đổi ι được tham số hóa bởi chỉ số vòng , . Trong phạm vi phép biến đổi ở thuật toán ι, tham số này xác định *l+1* bit của giá trị *lane* được gọi là hằng số vòng, và ký hiệu là RC. Mỗi bit của *l+1* bit được tạo ra bởi một hàm mà hàm này dựa trên một thanh ghi dịch tuyến tính có phản hồi. Hàm này ký hiệu là *rc* và được định nghĩa ở thuật toán bên dưới.

Thuật toán *rc(t)* **Input:** Số nguyên t  
 **Output:** bit *rc(t)* Các bước của thuật toán:  
 1. Nếu t mod 255 = 0, trả lại kết quả 1  
 2. Đặt R = 10000000  
 3. Cho *i* chạy từ 1 tới t mod 255, đặt:  
 3.1. R = 0||R  
 3.2. R[0] = R[0] ⊕ R[8]  
 3.3. R[4] = R[4] ⊕ R[8]  
 3.4. R[5] = R[5] ⊕ R[8]  
 3.5. R[6] = R[6] ⊕ R[8]  
 3.6. R = ]  
 4. Trả lại kết quả R[0]  
 Tác động của phép biến đổi ι là để biến đổi một vài bit của *lane[0,0]* phụ thuộc vào chỉ sổ của , còn lại 24 *lane* khác đều không bị ảnh hưởng bởi phép biến đổi ι. Biến đổi ι bao gồm việc thêm các hằng số vòng và hướng tới phá vỡ tính đối xứng. Các bit của hằng số vòng là khác nhau từ vòng này đến vòng kia và được lấy là đầu ra của LFSR(Linear-feedback shift register) có độ dài lớn nhất. Các hằng số này chỉ được thêm trong một *lane* của trạng thái. Do đó sự phá vỡ này sẽ được lan truyền thông qua θ và χ đối với tất cả *lane* của trạng thái.   
 Thuật toán ι  
 **Input:** Mảng trạng thái A và chỉ số vòng   
 **Output:** Mảng trạng thái A’  
 Các bước biến đổi như sau:  
 1. Với tất cả những bộ 3 (x, y, z) thỏa mãn những điều kiện 0 ≤ x ≤ 4, 0 ≤ y ≤ 4,   
0 ≤ z ≤ (w-1), đặt:  
   
 2. Đặt RC =   
 3. Cho *j* chạy từ 0 đến *l* ta đặt:  
   
 4. Với tất cả z thỏa mãn 0 ≤ z ≤ (w-1), ta đặt:  
   
 5. Trả lại kết quả A’.

**2.4.6 Hàm hoán vị Keccak-p[b,]**

Keccak-p[b,] là hàm hoán vị biến đổi một chuỗi bit ban đầu S có độ dài b bit thành một chuỗi S’ có độ dài b bit gồm hai thông số: b - chiều dài của chuỗi bit và là số vòng lặp trong phép biến đổi. Trong Keccak-p sử dụng hàm vòng Rnd như đã trình bày trên.  
   
 Phép hoán vị Keccak-p được trình bày như sau:  
 Input: Chuỗi bit S có độ dài b và số vòng lặp .  
 Output: Chuỗi bit S’ có độ dài b.  
 Các bước biến đổi như sau:   
 1. Biến đổi chuỗi S thành ma trận trạng thái 3 chiều A như đã trình bày ở 2.4.  
 2. Cho chạy từ 12 + 2*l* - đến 12 + 2*l* - 1, đặt A = Rnd(A,).  
 3. Chuyển đổi ma trận trạng thái 3 chiều A thành chuỗi S’ có độ dài b.  
 4. Trả lại kết quả S’.

**2.4.7 Hàm hoán vị Keccak-f[b]**

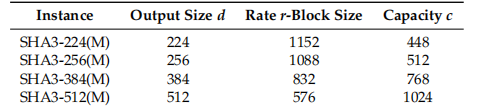
Cũng là một họ của phép hoán vị Keccak. Keccak-f chính là Keccak-p[b,] trong trường hợp = 12 + 2*l* :  
 Keccak-f[b] = Keccak-p[b,12 + 2*l*]  
 Keccak-f[1600,24] là cơ sở cho sáu hàm SHA-3 mà sẽ trình bày ở phần sau có thể viết ngắn gọn là Keccak-f[1600].

**2.4.8 Hàm Keccak[c](N,d)**

Như đã nhắc đến ở phần giới thiệu, Keccak được thiết kế dựa trên cấu trúc bọt biển do đó cần phải có những thành phần để xây dựng hàm bọt biển bao gồm mảng trạng thái, bộ đệm và phép hoán vị. Trong trường hợp b = 1600, bộ đệm sử dụng là Pad10\*1, và phép hoán vị là Keccak-f[1600] thì hàm bọt biển trở thành Keccak[c] chính là cấu trúc của thuật toán Keccak dùng để xây dựng các hàm SHA-3.   
 Keccak[c] = SPONGE[Keccak-f[1600], Pad10\*1, 1600 - c]  
 Keccak-c nhận đầu vào là N và d trong đó N là chuỗi bit đầu vào và d là độ dài của output.  
 Keccak[c](N,d) = SPONGE[Keccak-f[1600], Pad10\*1, 1600 - c](N,d)

**2.5 Thiết kế SHA-3 bằng thuật toán Keccak**

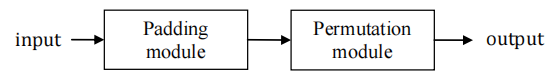
Như đã trình bày ở phần giới thiệu, Keccak là giải thuật đã giành chiến thắng trong cuộc thi thiết kế phần cứng cho giải thuật Secure Hash Algorithm -3 do NIST tổ chức. Cuộc thi này yêu cầu mỗi ứng viên phải đề xuất giải thuật hỗ trợ tối thiểu ngõ ra có 4 chiều dài khác nhau tùy thuộc vào chức năng của mỗi hệ thống. Họ SHA-3 gồm 4 hàm băm mã hóa chính : SHA3-224, SHA3-256, SHA3-384 và SHA3-512, và 2 hàm băm mở rộng: SHAKE128 và SHAKE256.  
 Với một đoạn thông điệp ký hiệu là M, bốn hàm băm mã hóa chính SHA-3 được định nghĩa bằng phép Keccak-[c] như đã trình bày ở 2.4.8 như sau:  
 SHA3-224(M) = Keccak-[448] (M||01, 224),  
 SHA3-256(M) = Keccak-[512] (M||01, 256),  
 SHA3-384(M) = Keccak-[768] (M||01, 384),  
 SHA3-512(M) = Keccak-[1024] (M||01, 512).  
Trong mỗi trường hợp, capacity c bằng hai lần giá trị d - chiều dài của ngõ ra. Chuỗi ngõ vào N chính là thông điệp M chèn thêm hai bit suffix 01 phía sau để phân biệt với 2 thuật toán mở rộng là SHAKE128 và SHAKE256.  
 SHAKE128(M,d) = Keccak-[256] (M||1111, d),  
 SHAKE256(M,d) = Keccak-[512] (M||1111, d).  
Khác với bốn hàm băm mã hóa chính của SHA-3 các hàm mở rộng này có độ dài output không cố định và phù thuộc vào lựa chọn của người sử dụng. Chuỗi ngõ vào N chính là thông điệp M chèn thêm bốn bit 1111 phía sau.



Bảng 2 - 3 Các thông số trong từng giải thuật SHA-3

# GIẢI PHÁP THỰC HIỆN

Từ những nội dung lý thuyết đã trình bày ở phần 2, kiến trúc cơ bản của hàm băm Keccak cho giải thuật SHA-3 được trình bày đơn giản như sau



Hình 3 - 1 Sơ đồ tổng quát hàm băm SHA-3

Ngõ vào trước tiên được đi qua khối Padding để tạo thành một khối có độ dài là bội số của bitrate r. Padding Module được xây dựng dựa trên thuật toán Pad10\*1 đã trình bày ở phần 2.3. Sau khi thực hiện Padding, ngõ ra của nó sẽ được đưa vào bộ hoán vị chính là module chính của giải thuật hàm băm Keccak chính là sự kết hợp giữa 5 hàm ánh xạ đã trình bày ở 2.4.1 - 2.4.5. Và sau đó kết quả được cắt ra theo đúng độ dài ngõ ra tương ứng và trả ra ngõ ra.  
 Một trong những thiết kế đáng để tham khảo đó là thiết kế [1]. Ở thiết kế này, tác giả đã đạt được thông lượng (throughput) khá ấn tượng nhờ việc sử dụng kỹ thuật pipeline để giảm thời gian tính toán tới hạn. Một thiết kế khác là [3], tác giả giới thiệu một thiết kế giúp tiết kiệm tài nguyên phần cứng bằng cách kết hợp các phép ánh xạ Rho (ρ), Pi (π), Chi (χ) lại từ đó giảm đi phần cứng xử lý, tần số hoạt động cũng khá cao tuy nhiên đánh đổi với thông lượng (throughput) khá thấp. Trong thiết kế [2], nhóm tác giả giới thiệu 5 kiến trúc khác nhau áp dụng kết hợp các kỹ thuật pipeline, subpipeline, unrolling đạt những kết quả ấn tượng trong đó tần số và thông lượng khá cao tuy nhiên tốn khá nhiều tài nguyên phần cứng so với [1]. Một số thiết kế khác đạt kết quả ấn tượng cũng được trình bảy theo bảng dưới, các thông số sử dụng trong bảng là sử dụng thiết bị của Xilinx.

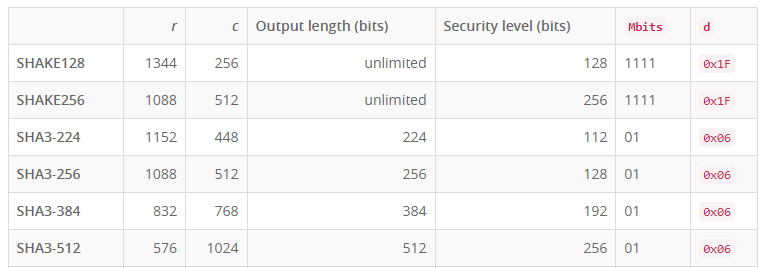
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Thiết kế | Thiết bị | Tần số (MHz) | Diện tích (Slices) | Throughput (Gbps) |
| [1] | Virtex-5 | 389 | 1702 | 18.7 |
| [2] | Virtex-6 | 344 | 1406 | 16.51 |
| [3] | Virtex-5 | 301 | 240 | 7.224 |
| [4] | Virtex-5 | 287.39 | 1388 | 11.5 |
| [5] | Virtex-5 | 312.98 | 1304 | 7.511 |

Bảng 3 - 1 So sánh giữa các thiết kế

Để thực hiện cứng hóa cho giải thuật hàm băm Keccak , em xin được trình bày 3 module chính đóng góp từ việc nhận giá trị ngõ vào, xử lý ngõ vào và các tín hiệu điều khiển quá trình xử lý của hệ thống trên cho đến việc xuất ra giá trị ngõ ra với độ dài tương ứng và một số module liên quan khác.  
 Việc ngõ vào có thể là một số 64bit, 128bit hay 1024bit … dẫn đến số lượng I/O port khi mô phỏng bằng các thiết bị FPGA sẽ không đáp ứng đủ. Trong thiết kế của em, thông điệp cần được băm là một chuỗi có độ dài là bội số của 128. Tại thời điểm cạnh tích cực của clock, một chuỗi 128 bit được đưa vào và theo lý thuyết là không giới hạn độ dài chuỗi ngõ vào. Tại chuỗi 128 cuối cùng, sẽ có một tín hiệu thông báo đây là khối cuối cùng để hệ thống tiến hành xử lý. Để thực hiện được điều đó, một bộ đệm được sử dụng để lưu giá trị từ chuỗi đầu tiên đến chuỗi cuối cùng. Khi tín hiệu ngõ vào đã được đưa vào hết, giá trị trong buffer sẽ được đi vào module Padding.

**3.1 Module Padding**

Padding là một phần không thể thiếu trong các thiết kế dựa trên cấu trúc bọt biển, nó thêm vào các bit bổ sung để tạo thành một khối hoặc là nhiều khối có độ dài cố định từ chuỗi ngõ vào. Module Padding sử dụng trong thiết kế là Pad10\*1 có nguyên lý hoạt động như đã trình bày ở 2.3. Nó sẽ chèn một bit 1 vào sau chuỗi ngõ vào và tiếp theo là chuỗi bit 0 và kết thúc là một bit 1.   
 Trước khi thực hiện Pad10\*1, thông điệp ngõ vào phải được chèn thêm các bit suffix như đã trình bày ở phần 2.5.



Bảng 3 - 2 Thông số hậu tố trong từng giải thuật băm SHA-3

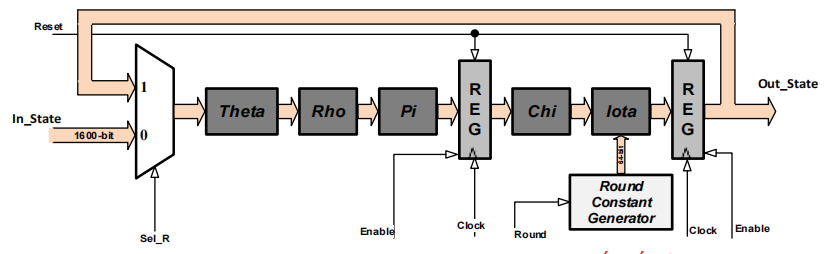
Bảng 3 - 1 trình bày các bit hậu tố suffix (Mbits) và hậu tố phân cách (d) được mã hóa từ các Mbits theo sau và độ dài của nó. Các Mbits và d được quyết định dựa trên loại thuật toán mà ta sử dụng. Vì vậy cần có một bộ mux để lựa chọn giá trị chèn vào thích hợp. Ở 6 giải thuật chính trong Keccak, ta cần phải dùng tín hiệu lựa chọn có độ dài là 3 bit. Mỗi tín hiệu lựa chọn quyết định một cặp số r và c tương ứng làm ngõ vào cho Module Padding. Trong các giải thuật SHA-3 thì SHAKE128 có độ dài bitrate r cao nhất nên em sẽ sử dụng một buffer có kích thước là 11x128 dể lưu trữ. Tùy thuộc vào tín hiệu lựa chọn mà độ dài của bitrate là khác nhau và buffer sẽ gửi thông báo khi độ dài lưu trữ trong buffer bằng với bitrate của thuật toán mà ta lựa chọn.   
 Ví dụ, khi thuật toán ta lựa chọn là SHA3-226 có r = 1152, c = 448 thì khi chuỗi ngõ vào thứ 9 được ghi vào buffer (9 = 1152/128), tín hiệu thông báo buffer đã đầy sẽ được xuất ra và chuỗi 1152bit đó sẽ được kết hợp với 448 bit 0 tạo thành một khối 1600 bit đưa vào hàm băm xử lý.   
 Ví dú trên là trong trường hợp xử lý đa khối, tức là khi buffer đầy, chuỗi ngõ vào vẫn chưa dừng lại. Chuỗi ngõ vào thứ nhất được đưa vào quá trình xử lý và chuỗi ngõ vào thứ hai lần lượt được đưa vào buffer cho đến khi có chuỗi cuối cùng. Trong trường hợp chuỗi cuối cùng đã xuất xong mà buffer vẫn chưa đầy, lúc này sẽ thực hiện chèn các bit hậu tố và Pad10\*1 đến khi buffer đầy và tiến hành xử lý.

Ở cuối module Padding là một thanh ghi có vai trò làm giảm đường tới hạn, áp dụng kỹ thuật pipeline giúp xử lý các chuỗi khối tiết kiệm được thời gian hơn.

**3.2 Module Step-Mapping**

Đây là module chính thực hiện quá trình xử ly. Thông điệp sau khi đã đi qua k hối Padding đã được chia thành nhiều khối, mỗi khối có độ dài là 1600bit bằng với số bit trong ma trận trạng thái ba chiều mà ta sử dụng trong phép hoán vị Keccak đã được giới thiệu trong phần 2.4. Một hàm vòng là sự kết hợp giữa năm hàm ánh xạ và được lặp lại 24 lần trước khi đi vào bước Squeezing của cấu trúc bọt biển.

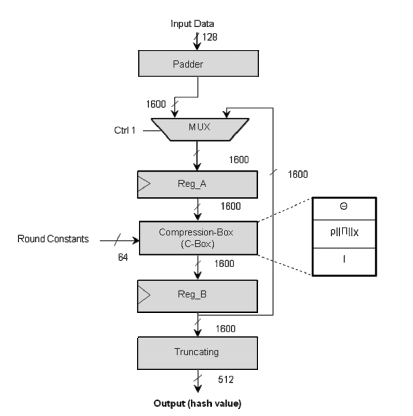
Một thiết kế đáng để tham khảo là thiết kế [1] khi tác giả đã áp dụng pipeline vào trong Module này và thu được kết quả là throughput rất lớn.



Hình 3 - 2 Sơ đồ khối StepMapping trong thiết kế [1]

Hai thanh ghi được chèn vào giữa bước Pi-Chi và cuối Iota để giảm đi thời gian tính toán tới hạn từ đó làm tăng tần số hoạt động của hệ thống.

Thiết kế [2] mang lại hiệu quả cao về diện tích vì tiêu thụ ít tài nguyên phần cứng có sơ đồ thiết kế cho module này như sau.



Hình 3 - 3 Sơ đồ khối tổng quát trong thiết kế [3]

Trong đề tài luận văn này, em sẽ cải tiến thiết kế giải thuật hàm băm Keccak bằng cách kết hợp pipeline như Hình 3 - 2 nhưng thanh ghi sẽ được đặt sau phép ánh xạ Theta và đồng thời gộp các phép ánh xạ Rho, Pi, Chi lại để tiết kiệm tài nguyên phần cứng hơn.

**3.3 Module Control**

Một module quan trọng khác nữa đó là module Control xử lý các tín hiệu điều khiển trong toàn hệ thống. Module Control sẽ cho ngõ ra là các tín hiệu điều khiển như: tín hiệu lựa chọn phép toán, tín hiệu thông báo đã bắt đầu quá trình băm, tín hiệu thông báo quá trình băm đã hoàn tất, tín hiệu đếm số vòng lặp trong phép hoán vị, … Lựa chọn thiết kế của em đó là xây dựng module Control là một máy trạng thái (FSM).

# THIẾT KẾ VÀ THỰC HIỆN PHẦN CỨNG

## Yêu cầu thiết kế

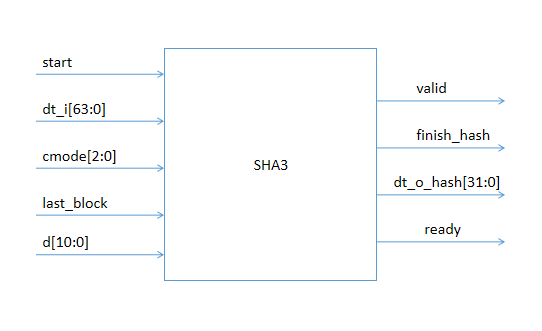
\* Thiết kế giải thuật Keccak hỗ trợ cho các hàm băm SHA3-224, SHA3-256, SHA3-384, SHA3-512 và hai hàm mở rộng SHAKE-128, SHAKE-256 dưới dạng IP (Intellectural Property) core để sử dụng cho nhiều dự án khác nhau.

\* Thiết kế đảm bảo tài nguyên tiết kiệm trên FPGA Altera Cyclone V.

\* Thiết kế đạt tốc độ cao.

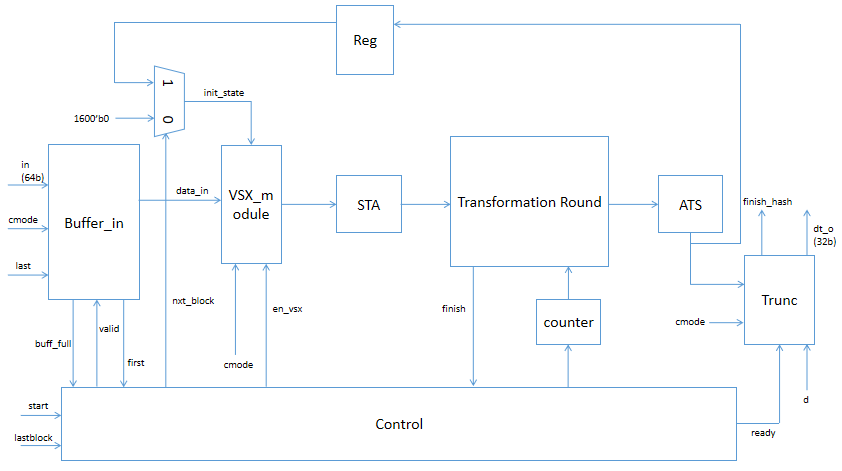
## Phân tích thiết kế

Phương pháp thiết được thực hiện dưới dạng các mô đun phần cứng trực quan, dễ thay đổi và chỉnh sửa.



Hình 4 - 1 SHA3-core

## Sơ đồ khối tổng quát

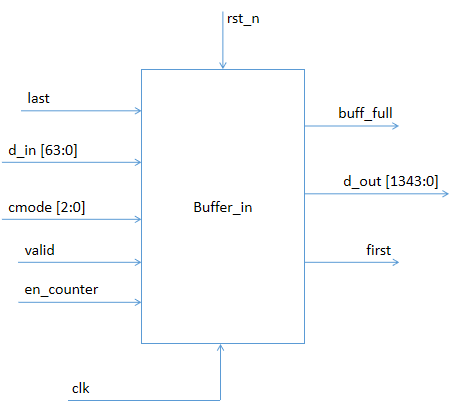


Hình 4 - 2 Sơ đồ khối tổng quát của thiết kế

## Sơ đồ khối chi tiết và nhiệm vụ, chức năng từng khối

**4.4.1 Bộ đệm ngõ vào Buffer\_in**

**4.4.1.1 Mô hình chi tiết**



Hình 4 - 3 Sơ đồ khối khối Buffer\_in

**4.4.1.2 Mô tả chân kết nối**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tín hiệu** | **Số bit** | **Ngõ vào** | **Ngõ ra** | **Mô tả** |
| last | 1 | x |  | Tín hiệu thông báo chuỗi 64bit d\_in đã là chuỗi cuối cùng thực hiện tính |
| d\_in | 64 | x |  | Ngõ vào dữ liệu của hàm băm, gồm 64bit |
| cmode | 3 | x |  | Lựa chọn MODE của hàm băm |
| valid | 1 | x |  | Cho phép Buffer nhận giá trị từ d\_in |
| en\_counter | 1 | x |  | Tín hiệu cho phép khối Counter thực hiện, trong này dùng để xét điều |
| buff\_full | 1 |  | x | Tín hiệu thông báo Buffer đã đầy |
| d\_out | 1344 |  | x | Ngõ ra dữ liệu gồm 1344bit của Buffer, tín hiệu này sẽ kết hợp với tín hiệu từ Init\_state tạo thành một khối 1600bit |
| first | 1 |  | x | Tín hiệu thông báo Block hiện tại là đầu tiên |

Bảng 4 - 1 Chân kết nối trong khối Buffer\_in

**4.4.1.3 Mô tả chức năng**

Bởi vì số lượng pin có hạng khi ta tổng hợp trên phần cứng của FPGA Cyclone V, ta không thể đọc cùng lúc một lượng lớn số bit ngõ vào được. Do đó em làm một bộ Buffer\_in có chức năng nhận giá trị từ ngõ vào (64bit) đến lúc đầy thì sẽ bật tín hiệu **buff\_full** hoặc khi có tín hiệu **last** thì sẽ dừng việc đọc lại. Trong trường hợp tín hiệu **last** ở mức cao mà buffer vẫn chưa đầy, thì buffer sẽ thực hiện việc padding với quy tắc pad10\*1 cho đến khi buffer đầy và tín hiệu **buff\_full** lúc đó sẽ được tích cực. Ngõ ra dữ liệu của Buffer\_in là một chuỗi gồm 1344 bit. Lý do xuất hiện số 1344 là bởi vì 1344 là số bit rate lớn nhất trong tất cả 4 giải thuật SHA3 và 2 giải thuật SHAKE. Tùy thuộc vào MODE mà ta sử dụng mà số bit chứa trong Buffer là đầy. MODE được thể hiện qua tín hiệu **cmode** như bảng dưới đây:

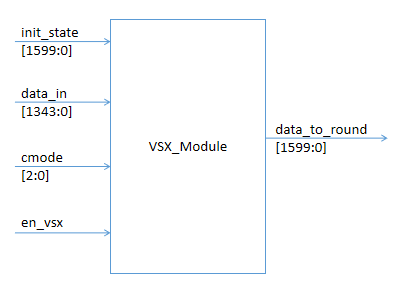
|  |  |  |
| --- | --- | --- |
| cmode[2:0] | MODE | Byte of Buffer |
| 0 | SHA3-224 | 144 |
| 1 | SHA3-256 | 136 |
| 2 | SHA3-384 | 104 |
| 3 | SHA3-512 | 72 |
| 4 | SHAKE-128 | 168 |
| 5 | SHAKE-256 | 136 |

Bảng 4 - 2 Bảng số Byte tương ứng mỗi MODE

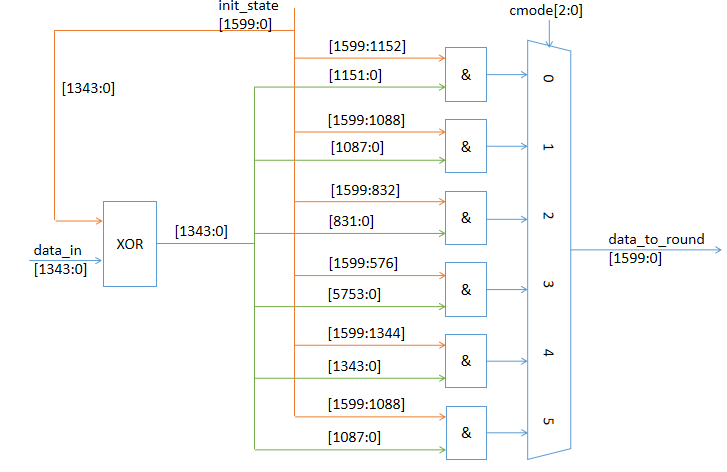
Theo lý thuyết của hàm bọt biển, tại vòng lặp đầu tiên, r bit rate sẽ kết hợp với c bit 0 của chuỗi trạng thái ban đầu 1600b’0. Còn tại những vòng lặp tiếp theo, c bit 0 sẽ được thay bằng c bit của chuỗi trạng thái trong vòng lặp trước đó. Bởi vậy, em đã thêm vào một tín hiệu **first** có chức năng thông báo cho bộ điều khiển biết rằng khối tính toán hiện tại là khối đầu tiên từ đó khối điều khiển sẽ điều khiển con MUX lựa chọn chuỗi trạng thái là 1600’b0 hay là chuỗi trạng thái ở vòng lặp trước đó để kết hợp với chuỗi trạng thái hiện tại để thực hiện tính toán hàm băm.

**4.4.2 Bộ VSX**

**4.4.2.1 Mô hình chi tiết**



Hình 4 - 4 Sơ đồ khối VSX



Hình 4 - 5 Sơ đồ khối chi tiết khối VSX

**4.4.2.2 Mô tả chân kết nối**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tín hiệu** | **Số bit** | **Ngõ vào** | **Ngõ ra** | **Mô tả** |
| init\_state | 1600 | x |  | Trạng thái khởi tạo dùng để XOR với data\_in |
| data\_in | 1344 | x |  | Dữ liệu ngõ vào của khối VSX |
| cmode | 3 | x |  | Lựa chọn MODE |
| en\_vsx | 1 | x |  | Cho phép VSX\_Module hoạt động |
| data\_to\_round | 1600 |  | x | Chuỗi trạng thái ngõ ra 1600 bit |

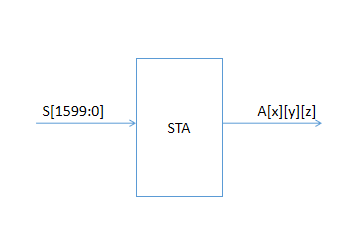
Bảng 4 - 3 Chân kết nối trong khối VSX

**4.4.2.3 Mô tả chức năng**

Chuỗi dữ liệu 1344 bit **data\_in** từ ngõ ra của khối Buffer\_in sẽ được đi qua khối VSX\_Module. Trong giải thuật Keccak, ta sẽ thực hiện trên một khối ma trận có độ dài cố định là 1600 bit. Để tạo thành một khối có độ dài cố định 1600bit, chuỗi dữ liệu ngõ vào (**data\_in)** phải kết hợp với một chuỗi trạng thái ban đầu **init\_state.** Chuỗi này là 1600 bit 0 hoặc là chuỗi trạng thái của khối tính toán tiếp theo như đã giải thích ở 4.4.1.3. VSX\_Module (Version Selection and XOR) là khối thực hiện nhiệm vụ lựa chọn MODE sau đó XOR **data\_in** với chuỗi 1600 bit **init\_state**. Tùy vào MODE mà số bit XOR với **init\_state** là khác nhau. Tín hiệu **cmode** làm tín hiệu lựa chọn cho bộ lựa chọn và sẽ cho giá trị 1600 bit tương ứng với MODE phép tính.

**4.4.3 Bộ STA**

**4.4.3.1 Mô hình chi tiết**



Hình 4 - 6 Sơ đồ khối STA

**4.4.3.2 Mô tả chân kết nối**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tín hiệu** | **Số bit** | **Ngõ vào** | **Ngõ ra** | **Mô tả** |
| S | 1600 | x |  | Chuỗi dữ liệu ngõ vào gồm chuỗi 1600bit |
| A | 1600 |  | x | Mảng trạng thái ngõ ra 3 chiều A[x][y][z] với x = 0..4 ; y = 0..4; z = 0..64 |

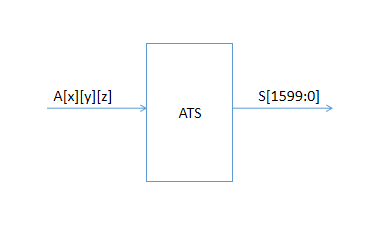
Bảng 4 - 4 Chân kết nối trong khối STA

**4.4.3.3 Mô tả chức năng**

Sau khi đưa ra được một chuỗi 1600 bit - ngõ ra của khối VSX\_Module, ta đã có được một chuỗi trạng thái gồm 1600 bit. Để biểu diễn chuỗi bit này dưới dạng ma trận trạng thái 3 chiều phục vụ cho việc tính toán hàm băm Keccak, ta cần một module để chuyển đổi nó. Module STA dùng để chuyển đổi chuỗi dữ liệu ngõ vào gồm 1600bit thành một mảng trạng thái 3 chiều để thực hiện các phép tính toán hoán vị bên trong giải thuật Keccak.

**4.4.4 Bộ ATS**

**4.4.4.1 Mô hình chi tiết**



Hình 4 - 7 Sơ đồ khối ATS

**4.4.4.2 Mô tả chân kết nối**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tín hiệu** | **Số bit** | **Ngõ vào** | **Ngõ ra** | **Mô tả** |
| A | 1600 | x |  | Mảng trạng thái ngõ vào 3 chiều A[x][y][z] với x = 0..4 ; y = 0..4; z = 0..64 |
| S | 1600 |  | x | Chuỗi dữ liệu ngõ ra gồm chuỗi 1600bit |

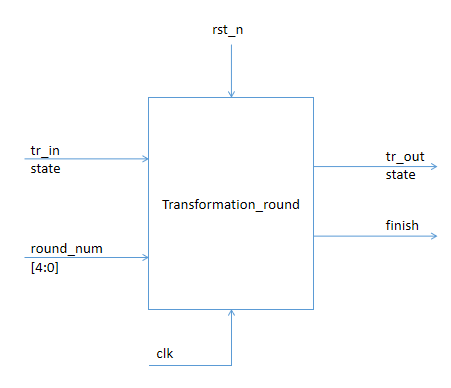
Bảng 4 - 5 Chân kết nối trong khối ATS

**4.4.4.3 Mô tả chức năng**

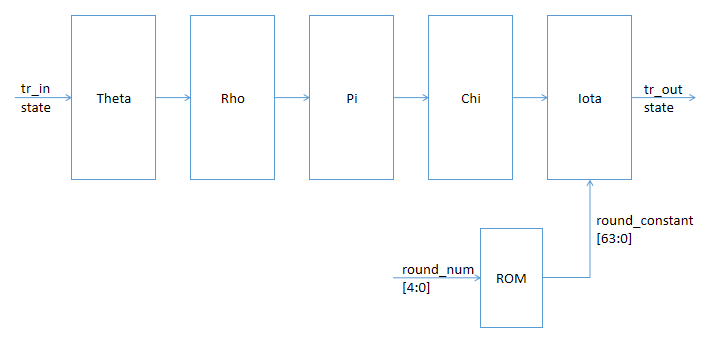
Module ATS dùng để chuyển đổi mảng trạng thái 3 chiều ngõ vào thành chuỗi dữ liệu 1600bit ngõ ra. Sau quá trình tính toán các hàm hoán vị, mảng trạng thái được chuyển về dạng chuỗi bit và xuất ra ngõ ra với số lượng bit phụ thuộc vào loại giải thuật SHA3 mà ta lựa chọn (**cmode**).

**4.4.5 Bộ biến đổi Transformation Round**

**4.4.5.1 Mô hình chi tiết**



Hình 4 - 8 Sơ dồ khối Transformation\_round



Hình 4 - 9 Sơ đồ chi tiết khối Transformation\_round

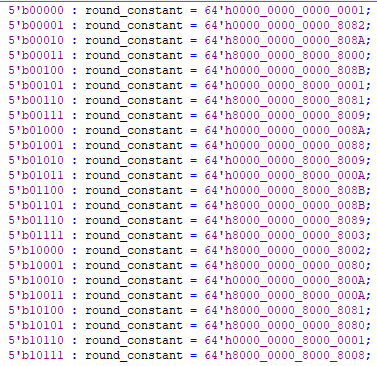
**4.4.5.2 Mô tả chân kết nối**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tín hiệu** | **Số bit** | **Ngõ vào** | **Ngõ ra** | **Mô tả** |
| tr\_in | 1600 | x |  | Ngõ vào dữ liệu của Transformation\_round là mảng trạng thái ba chiều có 1600 phần tử (5x5x64) |
| round\_num | 5 | x |  | Đếm số thự tự vòng lặp và tra vào bảng ROM kết quả tương ứng sử dụng trong phép hoán vị Iota |
| tr\_out | 1600 |  | x | Ngõ ra dữ liệu của Transformation\_round là mảng trạng thái ba chiều có 1600 phần tử (5x5x64) |
| finish | 1 |  | x | Tín hiệu thông báo đã kết thúc tính toán |

Bảng 4 - 6 Chân kết nối trong khối Transformation\_round

**4.4.5.3 Mô tả chức năng**

Module Transformation\_round thực hiện phép tính toán chính của Keccak, gồm 5 phép hoán vị Theta, Rho, Pi, Chi và Iota như trên. Việc thực hiện các phép hoán vị tác động lên mảng trạng thái 3 chiều và ngõ ra cũng là một mảng trạng thái 3 chiều. Giải thuật chi tiết của các phép hoán vị đã được trình bày trong mục 2.4.1 đến 2.4.5. Trong đó riêng phép hoán vị Iota (chỉ tác động lên lane gốc) cần thêm một hằng số vòng được cho sẵn trong bảng ROM và phụ thuộc vào thứ tự vòng hiện tại.

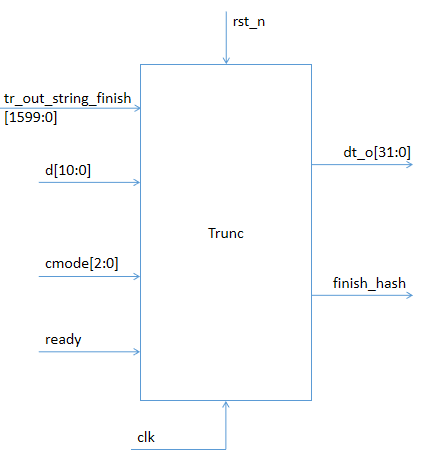


Hình 4 - 10 Bảng ROM lưu giữ giá trị round\_constant dùng trong Iota

Module Transformation\_round thực hiện tính toán 5 phép hoán vị và lặp lại 24 lần trong mỗi khối trạng thái 3 chiều 1600 bit. Tín hiệu **round\_num** có vai trò như một biến đếm, sau mỗi lần tính toán thì tín hiệu sẽ được tăng lên một nhờ bộ counter như trong hình 4 - 2. Bộ counter được điều khiển bằng tín hiệu **en\_counter** là một trong những ngõ ra của bộ điều khiển.

**4.4.6 Bộ Trunc**

**4.4.6.1 Mô hình chi tiết**



Hình 4 - 11 Sơ đồ khối Trunc

**4.4.6.2 Mô tả chân kết nối**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tín hiệu** | **Số bit** | **Ngõ vào** | **Ngõ ra** | **Mô tả** |
| Tr\_out\_string\_finish | 1600 | x |  | Chuỗi dữ liệu ngõ vào của Trunc |
| d | 11 | x |  | Độ dài output mong muốn (sử dụng cho SHAKE) là bội số 32 có độ dài tối đa là 1024 bit |
| cmode | 3 | x |  | Tín hiệu lựa chọn MODE |
| ready | 1 | x |  | Tín hiệu thông báo đã sẵn sàng cho việc xuất dữ liệu ngõ ra |
| dt\_o | 32 |  | x | Chuỗi dữ liệu ngõ ra |
| finish\_hash | 1 |  | x | Tín hiệu thông báo kết thúc giải thuật băm |

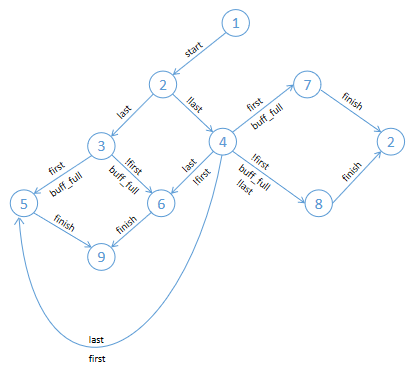
Bảng 4 - 7 Chân kết nối trong khối Trunc

**4.4.6.3 Mô tả chức năng**

Ngõ ra của khối Transformaton\_round là một ma trận trạng thái ba chiều sau đó được chuyển đổi thành chuỗi trạng thái 1600 bit nhờ module ATS. Mỗi giải thuật SHA3, SHAKE thì độ dài ngõ ra là khác nhau do đó để có thể xuất dữ liệu ra có độ dài tương ứng với loại giải thuật mà ta muốn tính hàm băm, ta cần phải có thêm một module để tách dữ liệu từ chuỗi dữ liệu 1600 bit đó. Module Trunc dùng để tách chuỗi dữ liệu là ngõ ra của phép tính toán hàm vòng thành các chuỗi bit tương ứng với từng MODE của hàm băm và đưa giá trị ngõ ra thông qua **dt\_o**. Ngõ ra của SHA3-224, SHA3-256, SHA3-384 và SHA3-512 lần lượt là 224, 256, 384 và 512 bit. Ngõ ra của SHAKE-128 và SHAKE-256 là không giới hạn. Cũng như module Buffer\_in đọc giá trị mỗi xung clock là 64 bit, số lượng pin trong phần cứng FPGA Cyclone V là có hạn nên ta không thể cho ngõ ra có độ dài quá lớn. Do đó em quyết định lấy bội chung nhỏ nhất của 224, 256, 384 và 512 là 32 để làm số lượng bit ngõ ra trong một chu kỳ clock. Điều này dẫn đến ngõ vào **d** là độ dài ngõ ra khi thực hiện giải thuật SHAKE-128 và SHAKE-256 cũng có ràng buộc là bội số của 32. Giới hạn của **d** là 2048 tương ứng với **d** có độ dài 10 bit. Khi số bit được đưa ra đã tương ứng với độ dài ngõ ra của hàm băm thì tín hiệu **finish\_hash** sẽ được tích cực thông báo kết thúc quá trình tính toán hàm băm Keccak.

**4.4.7 Khối điều khiển tín hiệu Control**

**4.4.7.1 Sơ đồ trạng thái của Module Control**



Hình 4 - 12 Máy trạng thái khối Control

**4.4.7.2 Mô tả chân kết nối**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tín hiệu** | **Số bit** | **Ngõ vào** | **Ngõ ra** | **Mô tả** |
| **start** | 1 | x |  | Tín hiệu bắt đầu thực hiện tính toán hàm băm, dữ liệu ngõ vào bắt đầu đi vào |
| **last** | 1 | x |  | Tín hiệu thông báo chuỗi dữ liệu ngõ vào tính toán hàm băm là chuỗi cuối cùng |
| **first** | 1 | x |  | Tín hiệu thông báo khối tính toán hàm băm là khối đầu tiên |
| **buff\_full** | 1 | x |  | Tín hiệu thông báo bộ đệm Buffer đã đầy |
| **finish** | 1 | x |  | Tín hiệu thông báo kết thúc tính toán hàm băm |
| **valid** | 1 |  | x | Tín hiệu cho phép buffer nhận giá trị |
| **nxt\_block** | 1 |  | x | Tín hiệu điều khiển mux nhận giá trị của khối tiếp theo |
| **en\_vsx** | 1 |  | x | Tín hiệu cho phép module VSX thực hiện |
| **en\_counter** | 1 |  | x | Tín hiệu cho phép module counter thực hiện |
| **ready** | 1 |  | x | Tín hiệu thông báo hoàn thành các bước tính toán |

Bảng 4 - 8 Chân kết nối trong khối Control

**4.4.7.3 Mô tả các trạng thái của Module Control**

|  |  |
| --- | --- |
| **THHT** | **Mô tả** |
| **S1** | Trạng thái ban đầu |
| **S2** | Đã nhận tín hiệu start, tín hiệu **valid = 1** để nhận giá trị từ input S7 và S8 khi nhận tín hiệu **finish** sẽ quay về S2 |
| **S3** | Nếu đã là khối cuối cùng thì **valid = 0** để không nhận giá trị nữa. Đồng thời **en\_vsx = 1** để thực hiện phép toán |
| **S4** | Nếu chưa phải khối cuối cùng thì **valid = 1** để nhận tín hiệu đến khi buffer đầy, **en\_vsx** vẫn tiếp tục vẫn bật |
| **S5** | Khi **buffer\_full = 1** và khối tính toán đang là khối đầu tiên **first = 1** thì **nxt\_block = 0** (để nhận giá trị 1600'b0 thay vì **init\_state** của khối tiếp theo) đồng thời tính hiệu **en\_counter = 1** để thực hiện việc đếm số vòng thực hiện |
| **S6** | Khi **buffer\_full = 1** và khối tính toán ko là khối đầu tiên **first = 0** thì **nxt\_block = 1** (để nhận giá trị là ngõ ra trước đó) đồng thời tính hiệu **en\_counter = 1** để thực hiện việc đếm số vòng thực hiện |
| **S7** | Tương tự S5 nhưng trạng thái trước đó của nó là S4 (chưa là khối cuối) |
| **S8** | Tương tự S6 nhưng trạng thái trước đó của nó là S4 (chưa là khối cuối) |
| **S9** | Trạng thái kết thúc, khối cuối đã được tính toán xong, **ready = 1** |

Bảng 4 - 9 Các trạng thái trong khối Control

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TTHT** | **Các tín hiệu ngõ ra** | | | | |
| **valid** | **nxt\_block** | **en\_vsx** | **en\_counter** | **ready** |
| **S1** | 0 | 0 | 0 | 0 | 0 |
| **S2** | 1 | 0 | 0 | 0 | 0 |
| **S3** | 0 | 0 | 1 | 0 | 0 |
| **S4** | 1 | 0 | 1 | 0 | 0 |
| **S5** | 0 | 0 | 1 | 1 | 0 |
| **S6** | 0 | 1 | 1 | 1 | 0 |
| **S7** | 0 | 0 | 1 | 1 | 0 |
| **S8** | 0 | 1 | 1 | 1 | 0 |
| **S9** | 0 | 0 | 0 | 0 | 1 |

Bảng 4 - 10 Các tín hiệu ngõ ra tương ứng trong khối Control

**4.4.7.4 Mô tả chức năng**

Module Control là module thực hiện nhiệm vụ điều khiển các tín hiệu điều khiển trong phần cứng của giải thuật Keccak. Ở trong thiết kế của em, sử dụng máy trạng thái kiểu Moore để thiết kế lên Module Control này. Các trạng thái được liệt kê như trong Bảng 4 - 9 với các giá trị ngõ ra tương ứng trong Bảng 4 - 10.

Các Module chính hỗ trợ việc thực hiện phần cứng giải thuật Keccak đã được trình bày từ mục 4.4.1 đến 4.4.7. Tiếp theo, em xin đề xuất một số cải tiến để tăng hiệu suất xử lý của phần cứng.

Theo lý thuyết hàm băm, chuỗi đầu vào có thể là một chuỗi có độ dài bất kỳ. Trong khi đó các giải thuật SHA3 và SHAKE thực hiện tính toán theo từng khối có độ dài cố định. Vì vậy nếu độ dài ngõ vào lớn hơn độ dài mà một khối tính toán có thể tính được thì sẽ được tách làm 2, 3, 4, … khối và thực hiện lần lượt từng khối như vậy với khối sau là **init\_state** của khối trước đó. Ví dụ, để thực hiện tính hàm băm với giải thuật SHA3-512 (r = 576), nếu độ dài của ngõ vào nhỏ hơn hoặc bằng 574 bit thì sẽ thực hiện tính toán với 1 khối (vì thuật toán padding10\*1 thêm tối thiểu là 2 bit 1) còn nếu độ dài của ngõ vào lớn hơn 574 thì buộc ta phải tính toán với nhiều hơn một khối. Sau đây là một thiết kế mà em đề xuất giúp (**keccak\_core\_ad\_NOPL**) làm cải thiện hiệu suất của phần cứng so với thiết kế đã trình bày ở trên (**keccak\_core\_NOPL**).

**4.4.8 Bộ đệm ngõ vào Buffer\_in\_2**

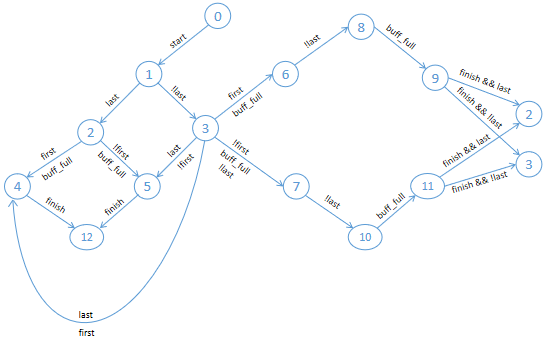
Như thiết kế phần cứng trên, Buffer\_in sẽ nhận giá trị từ ngõ vào mỗi chu kỳ là 64 bit và đến khi đầy sẽ dừng việc đọc lại. Nếu tín hiệu **last** chưa được tích cực, Buffer sẽ reset về 0 và chờ đến khi các khối tính toán xong để bắt đầu việc đọc cho khối tiếp theo. Điều này dẫn đến một sự lãng phí về thời gian. Trong lúc các khối đang thực hiện việc tính toán, Buffer có thể đọc giá trị từ ngõ vào và dự trữ trong đó. Đến lúc các khối phía sau thực hiện tính toán xong, thì dữ liệu của khối tiếp theo đã có sẵn trong Buffer và ta không cần phải tốn thêm thời gian để đọc từ ngõ vào điều này tiết kiệm rất nhiều thời gian xử lý vì với giải thuật SHA3-512 thì r = 576 tương ứng với việc tiết kiệm được 9 chu kỳ trong tổng thời gian tính toán cho mỗi khối và đối với khối SHAKE-128 thì con số đó lên tới 21 chu kỳ.

Thời gian đọc dữ liệu từ ngõ vào lớn nhất là của SHAKE-128 tương ứng với 21 chu kỳ. Điều này vẫn đảm bảo việc tính toán của hàm vòng Transformation\_round vẫn đúng vì để thực hiện hàm vòng này cần phải 24 chu kỳ.

**4.4.9 Khối điều khiển tín hiệu Control\_2**

Module Control là khối thứ hai được điều chỉnh để đáp ứng với ý tưởng trên. Ở thiết kế **keccak\_core\_NOPL**, ta cần 9 trạng thái để biểu diễn hoạt động và sự thay đổi của các tín hiệu điều khiển thì ở **keccak\_core\_ad\_NOPL**, số lượng trạng thái là 13 để đáp ứng với yêu cầu đặt ra.

**4.4.9.1 Sơ đồ trạng thái của Module Control**



Hình 4 - 13 Máy trạng thái khối Control\_2

**4.4.9.2 Mô tả chân kết nối**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tín hiệu** | **Số bit** | **Ngõ vào** | **Ngõ ra** | **Mô tả** |
| **start** | 1 | x |  | Tín hiệu bắt đầu thực hiện tính toán hàm băm, dữ liệu ngõ vào bắt đầu đi vào |
| **last** | 1 | x |  | Tín hiệu thông báo chuỗi dữ liệu ngõ vào tính toán hàm băm là chuỗi cuối cùng |
| **first** | 1 | x |  | Tín hiệu thông báo khối tính toán hàm băm là khối đầu tiên |
| **buff\_full** | 1 | x |  | Tín hiệu thông báo bộ đệm Buffer đã đầy |
| **finish** | 1 | x |  | Tín hiệu thông báo kết thúc tính toán hàm băm |
| **valid** | 1 |  | x | Tín hiệu cho phép buffer nhận giá trị |
| **nxt\_block** | 1 |  | x | Tín hiệu điều khiển mux nhận giá trị của khối tiếp theo |
| **en\_vsx** | 1 |  | x | Tín hiệu cho phép module VSX thực hiện |
| **en\_counter** | 1 |  | x | Tín hiệu cho phép module counter thực hiện |
| **ready** | 1 |  | x | Tín hiệu thông báo hoàn thành các bước tính toán |

Bảng 4 - 11 Chân kết nối trong khối Control\_2

Các tín hiệu trong Module Control\_2 cũng tương tự như trong Control.

**4.4.9.3 Mô tả các trạng thái của Module Control\_2**

|  |  |
| --- | --- |
| **THHT** | **Mô tả** |
| **S0** | Trạng thái ban đầu |
| **S1** | Đã nhận tín hiệu start, tín hiệu **valid** **= 1** để nhận giá trị từ input S7 và S8 khi nhận tín hiệu **finish** sẽ quay về S2 |
| **S2** | Nếu đã là khối cuối cùng thì **valid = 0** để không nhận giá trị nữa. Đồng thời **en\_vsx = 1** để thực hiện phép toán |
| **S3** | Nếu chưa phải khối cuối cùng thì **valid = 1** để nhận tín hiệu đến khi buffer đầy, en\_vsx vẫn tiếp tục vẫn bật |
| **S4** | Khi **buffer\_full = 1** và khối tính toán đang là khối đầu tiên **first = 1** thì **nxt\_block = 0** (để nhận giá trị 1600'b0 thay vì init\_state của khối tiếp theo) đồng thời tính hiệu **en\_counter = 1** để thực hiện việc đếm số vòng thực hiện |
| **S5** | Khi **buffer\_full = 1** và khối tính toán ko là khối đầu tiên **first = 0** thì **nxt\_block = 1** (để nhận giá trị là ngõ ra trước đó) đồng thời tính hiệu **en\_counter = 1** để thực hiện việc đếm số vòng thực hiện |
| **S6** | Tương tự S4 nhưng trạng thái trước đó của nó là S3 (chưa là khối cuối) |
| **S7** | Tương tự S5 nhưng trạng thái trước đó của nó là S3 (chưa là khối cuối) |
| **S8** | Nếu **last = 0** (chưa phải khối cuối cùng) thì vẫn reset buffer và tiếp tục nhận dữ liệu từ ngõ vào của khối tiếp theo |
| **S9** | Khi **buffer\_full = 1** S9 sẽ xét tín hiệu **finish** và **last** để chuyển sang trạng thái S2 hoặc S3 tương ứng |
| **S10** | Nếu **last = 0** (chưa phải khối cuối cùng) thì vẫn reset buffer và tiếp tục nhận dữ liệu từ ngõ vào của block tiếp theo (tương tự S8 nhưng trạng thái trước đó của nó là S7) |
| **S11** | Khi **buffer\_full = 1** ,S11 sẽ xét tín hiệu finish và last để chuyển sang trạng thái S2 hoặc S3 tương ứng (tương tự S9 nhưng trạng thái trước đó của nó là S10) |
| **S12** | Trạng thái kết thúc, khối cuối đã được tính toán xong, **ready = 1** |

Bảng 4 - 12 Các trạng thái trong khối Control

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TTHT** | **Các tín hiệu ngõ ra** | | | | |
| **valid** | **nxt\_block** | **en\_vsx** | **en\_counter** | **ready** |
| **S0** | 0 | 0 | 0 | 0 | 0 |
| **S1** | 1 | 0 | 0 | 0 | 0 |
| **S2** | 0 | 0 | 1 | 0 | 0 |
| **S3** | 1 | 0 | 1 | 0 | 0 |
| **S4** | 0 | 0 | 1 | 1 | 0 |
| **S5** | 0 | 1 | 1 | 1 | 0 |
| **S6** | 0 | 0 | 1 | 1 | 0 |
| **S7** | 0 | 1 | 1 | 1 | 0 |
| **S8** | 1 | 1 | 1 | 1 | 0 |
| **S9** | 0 | 1 | 1 | 1 | 0 |
| **S10** | 1 | 1 | 1 | 1 | 0 |
| **S11** | 0 | 1 | 1 | 1 | 0 |
| **S12** | 0 | 0 | 0 | 0 | 1 |

Bảng 4 - 13 Các tín hiệu ngõ ra tương ứng trong khối Control

# THIẾT KẾ VÀ THỰC HIỆN PHẦN MỀM

Yêu cầu đặt ra cho phần mềm: Thể hiện được các giải thuật sẽ được thực hiện trong phần cứng, hỗ trợ trong việc tính toán, sửa lỗi và kiểm tra các test của phần cứng, tạo file testbench phục vụ cho việc kiểm tra. Phần mềm cũng sẽ hỗ trợ được việc tính toán của tất cả các hàm băm trong giải thuật SHA3 dựa trên thuật toán Keccak.

Em lựa chọn ngôn ngữ Python để thiết kế cho phần mềm để phần mềm đơn giản sử dụng. Giải thuật Keccak trên phần mềm được tham khảo từ nhóm kỹ sư Guido Bertoni, Joan Daemen, Michael Peeters, và Gilles Van Assche. Nhiều thông tin sẽ được đề cập ở website [Keccak Team](https://keccak.team/index.html).

Dưới đây là đoạn code ngắn để test cho giải thuật Keccak của nhóm kỹ sư trên. Đầu vào là chuỗi gồm có 32 kí tự mã Hex tương ứng với độ dài theo bit là 128 bit.

Các thông số đưa vào hàm Keccak bao gồm:

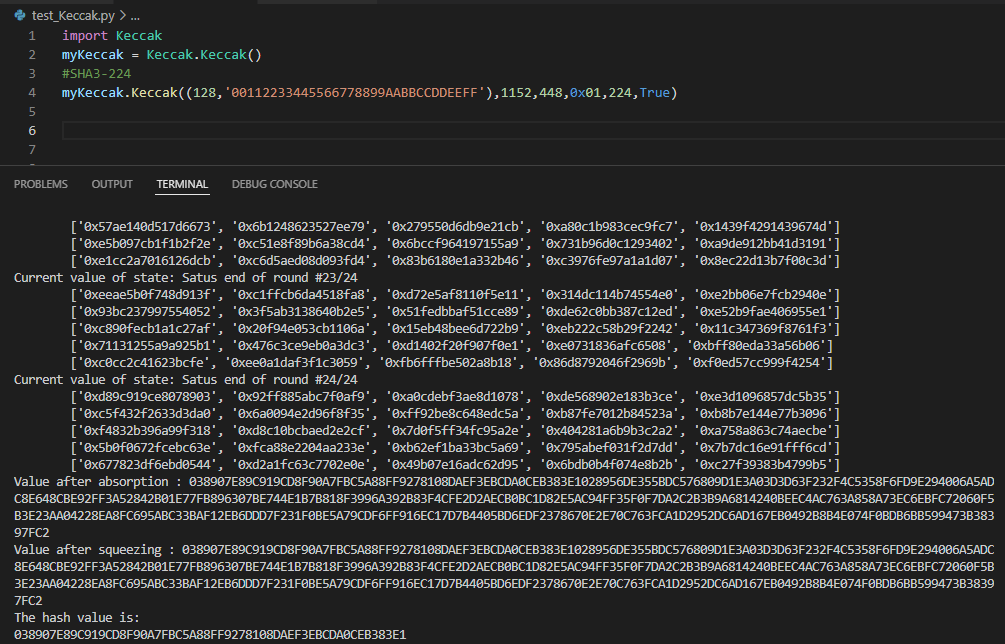
Keccak((in\_length), in\_string), r\_length, c\_length, suffix, out\_length, verbose)

Trong đó

|  |  |
| --- | --- |
| Thông số | Mô tả |
| in\_length | Chiều dài tính theo bit của chuỗi dữ liệu cần hash |
| in\_string | Chuỗi dữ liệu cần hash |
| r\_length | Chiều dài bitrate (tùy thuộc vào MODE đã mô tả trong bảng 3 - 2) |
| c\_length | Chiều dài capacity (tùy thuộc vào MODE đã mô tả trong bảng 3 - 2) |
| suffix | Giá trị hậu tố (đã mô tả trong bảng 3 - 2), giá trị mặc định là 0x01 biểu diễn cho giải thuật Keccak, 0x06 là SHA3 và 0x1F là SHAKE |
| out\_length | Chiều dài tính theo bit của giá trị ngõ ra |
| verbose | Cho phép in chi tiết các bước tính toán ra terminal |

Bảng 5 - 1 Các thông số trong hàm Keccak

Dưới đây là kết quả khi thử với chuỗi ngõ vào là : “00112233445566778899AABBCCDDEEFF”, giải thuật Keccak trả về kết quả như bên dưới.



Hình 5 - 1 Kết quả khi test chuỗi ngõ vào trên bằng Python

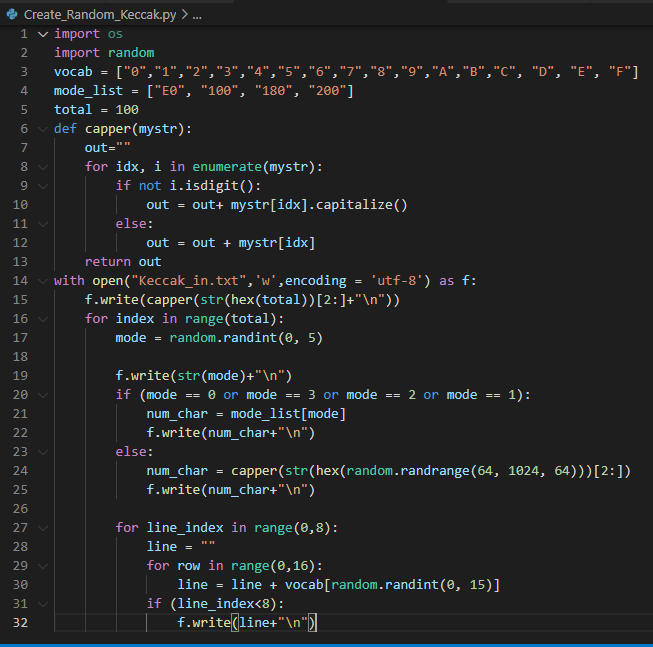
Kết quả này em đã kiểm định với nhiều website tính hàm băm khác ([Online Tools (emn178.github.io)](https://emn178.github.io/online-tools/), [New tab (leventozturk.com)](https://leventozturk.com/engineering/sha3/) ) và cũng cho ra kết quả tương tự.

Để phục vụ cho việc tính toán và kiểm tra với số lượng testbench lớn hơn. Em đã tự thiết kế một vài hàm với các chức năng như sau:

\* Tạo file testbench ngẫu nhiên (Create\_Random\_Keccak.py)  
 \* Nhận ngõ vào là file Create\_Random\_Keccak.py, thực hiện tính toán hàm băm và trả kết quả ra file Keccak\_out\_check.txt.  
 \* So sánh kết quả khi thực hiện bằng phần mềm (python) và khi thực hiện bằng phần cứng (Modelsim) và in kết quả kiểm tra ra màn hình.

**5.1 Create\_Random\_Keccak.py**

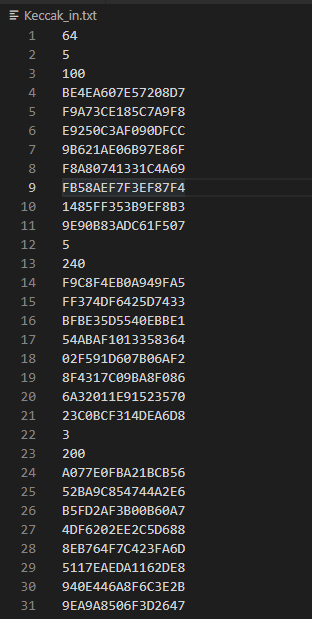
File Create\_Random\_Keccak.py tạo ra ngẫu nhiên một file Keccak\_in.txt là dữ liệu vào cho cả thiết kế phần cứng và phần mềm để đảm bảo việc kiểm tra được chính xác.



Hình 5 - 2 File Create\_Random\_Keccak.py

Kết quả sau khi chạy file này sẽ xuất ra một file Keccak\_in.txt như sau:  
 \* Dòng đầu tiên là số lượng test (dưới dạng mã HEX) mà ta dùng để tính toán hàm băm SHA3.  
 \* Dòng tiếp theo là cmode - MODE mà ta sử dụng: 0, 1, 2, 3, 4 và 5 lần lượt tương ứng với SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE-128 và SHAKE-256.  
 \* Dòng tiếp theo là chiều dài của dữ liệu ngõ ra (tính theo BIT, hiển thị dưới dạng HEX). Ở những giải thuật SHA3-224, SHA3-256, SHA3-384 và SHA3-512 có thể không cần giá trị ngõ vào này tuy nhiên phần cứng của em hỗ trợ thêm hai giải thuật là SHAKE-128 và SHAKE-256 nên cần phải có ngõ vào là số bit ở ngõ ra.  
 \* 8 dòng tiếp theo là dữ liệu cần được tính hàm băm.

Để việc đọc dữ liệu từ phần cứng dễ dàng hơn, em đã quy định cấu trúc file Keccak\_in.txt như trên và các khối dữ liệu cần tính hàm băm đều có chiều dài là 512bit.

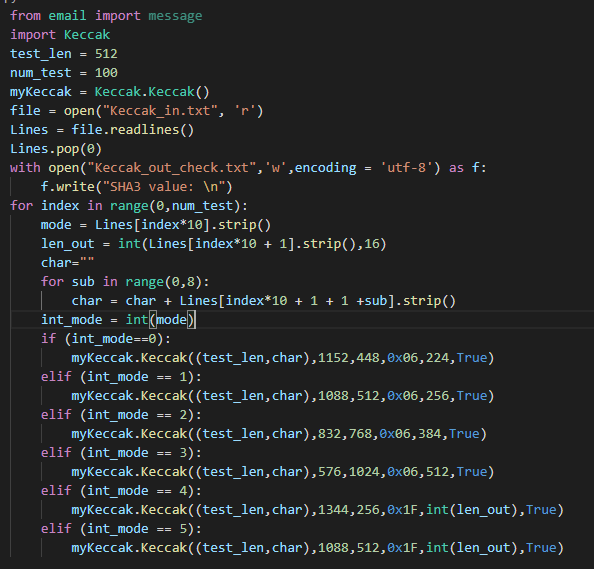


Hình 5 - 3 Kết quả khi chạy file Create\_Random\_Keccak.py

Hiện tại, em thực hiện tạo ra 100 test trong một file Keccak\_in.txt (giá trị “total” = 100). Giá trị trong file Keccak\_in.txt được đọc vào từ file Hash\_Function.py ở phần mềm và Module get\_input.sv ở phần cứng.

**5.2 Hash\_Function.py**

Hash\_Function.py là file chính thực hiện việc đọc dữ liệu từ file đầu vào là Keccak\_in.txt, thực hiện việc tính toán dựa trên hàm Keccak có sẵn từ nhóm kỹ sư Keccak sau đó tính ra giá trị hàm băm và in kết quả ra file Keccak\_out\_check.txt.

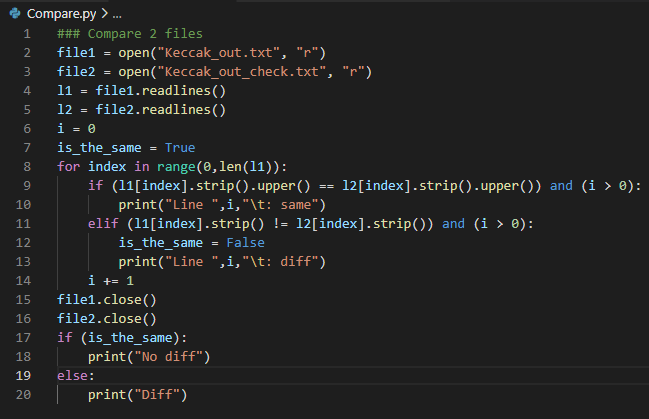


Hình 5 - 4 File Hash\_Function.py

Dựa trên giá trị MODE đọc được từ file Keccak\_in.txt mà ở file này, thực hiện tính toán với các thông số r, c, suffix, … tương ứng.

**5.3 Compare.py**

File cuối cùng là Compare.py thực hiện chức năng so sánh giá trị ngõ ra trong file Keccak\_out.txt (là ngõ ra của phần cứng) và file Keccak\_out\_check.txt (là ngõ ra của phần mềm). Kết quả hiện thị ra terminal.



Hình 5 - 5 File Compare.py

# 6. KẾT QUẢ THỰC HIỆN

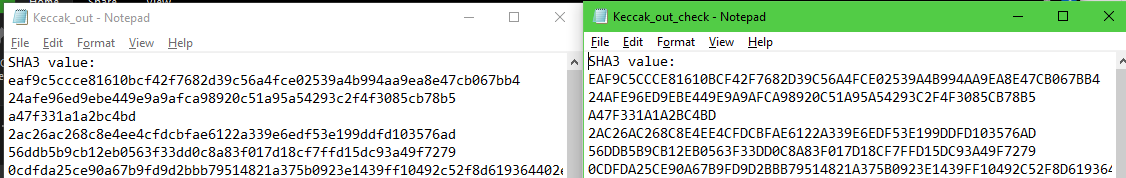
**6.1 Cách thức đo đạc, thử nghiệm**

Thiết kế phần cứng sau khi hoàn tất được thử nghiệm bằng chương trình mô phỏng phần cứng và phần mềm. Em sử dụng Modelsim để thiết kế và mô phỏng phần cứng, Python để thiết kế và mô phỏng phần mềm. Sau đó tổng hợp kết quả và đối chiếu để kiểm tra tính đúng đắn.

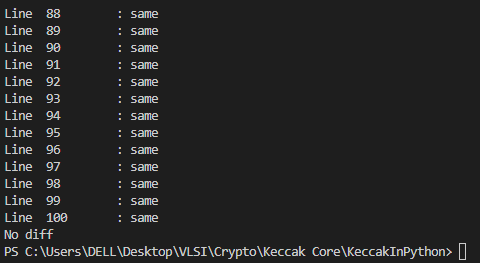
Để đo đạc các thông số về tài nguyên phần cứng, em sử dụng Quartus Prime 21.1 và thông số của kit FPGA 5CGXFC7C7F23C8 dòng Cyclone V của Intel.

**6.2 Số liệu đo đạc**

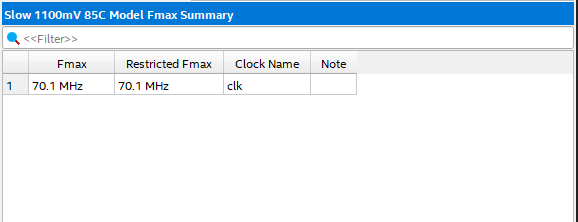
Kết quả mô phỏng cho thấy thiết kế chạy chính xác các MODE đã đề ra của SHA3 và SHAKE.



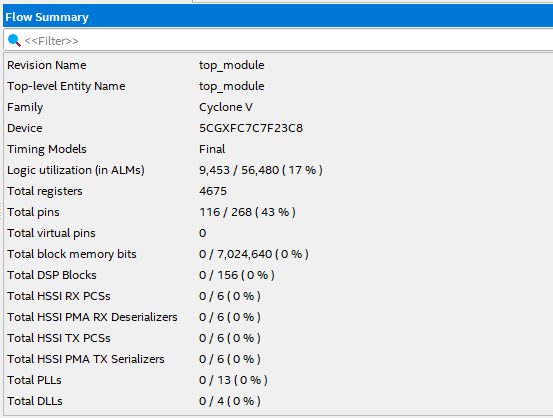
Hình 6 - 1 Kết quả tính toán hàm băm bằng phần cứng và phần mềm



Hình 6 - 2 Kết quả khi so sánh hai kết quả



Hình 6 - 3 Tần số của hệ thống



Hình 6 - 4 Kết quả tổng hợp phần cứng

# 7. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

# 8. TÀI LIỆU THAM KHẢO

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | [1] | G.-P. M. G. T. George S. Athanasiou, “HIGH THROUGHPUT PIPELINED FPGA IMPLEMENTATION OF THE NEW SHA-3”. | | [2] | J. H.-Y. S. S. A. C. Ming Ming Wong, “A New High Throughput and Area Efficient SHA-3 Implementation”. | | [3] | D.-e.-S. k. A. A. Alia Arshad, “Compact Implementation of SHA3-512 on FPGA”. | | [4] | H. M. B. B. M. M. Fatma Kahri, “High Speed FPGA Implementation of Cryptographic KECCAK Hash Function Crypto-Processor”. | | [5] | F. E. M. A. F. Assad, “An optimal hardware implementation of the KECCAK hash function on virtex-5 FPGA”. | | [6] | M. A. M. R. Kashif Latif, “Efficient Hardware Implementations and Hardware Performance Evaluation of”. | | [7] | P. K. N. S. C. K. George Provelengios, “FPGA-Based Design Approaches of Keccak Hash Function”. | | [8] | S. M. Atefeh Gholipour, “High-Speed Implementation of the KECCAK Hash Function on FPGA”. | | [9] | R. C. Magnus Sundal, “Efficient FPGA Implementation of the SHA-3 Hash Function”. | | [10] | K. I. T. M. T. K. Thuong Nguyen Dat, “Implementation of high speed hash function Keccak on GPU”. | | [11] | J. D. M. P. G. V. A. Guido Berton, “Cryptographic sponge functions”. | | [12] | J. D. M. P. G. V. A. Guido Berton, “The Keccak SHA-3 submission”. | | [13] | J. D. M. P. G. V. A. Guido Berton, “Keccak sponge function family main document”. | | [14] | J. D. M. P. G. V. A. Guido Berton, “The Keccak reference”. | | [15] | N. I. o. S. a. Technology, “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions”. | |