Implementazione dell’algoritmo di cifratura Noekeon in VHDL

Davide Quarta

maggio 2, 2006

1. Panoramica generale su Noekeon

Noekeon è un algoritmo di cifratura a “blocchi” basato su una chiave di cifratura a 128 bit. Esso consiste nella ripetuta applicazione di un semplice algoritmo di trasformazione della parola da cifrare (denominato round) seguito da un’ ultima trasformazione in output.

Noekeon è stato concepito per essere un algoritmo di cifratura efficiente e sicuro implementabile su un insieme decisamente ampio di piattaforme; tuttavia la sua struttura particolarmente compatta e semplice lo rende particolarmente adatto per la realizzazione su piattaforme,come le Smart Cards, in cui le risorse sono scarse ma in cui comunque si voglia garantire un elevato grado di sicurezza.

La “semplicità” dell’algoritmo è dovuta al fatto che Noekeon può essere implementato usando esclusivamente operazioni booleane bit-wise e shift eventualmente ciclici. Un’ulteriore caratteristica particolarmente interessante è che anche l’algoritmo di decifratura può essere eseguito sullo stesso circuito su cui è stato implementato Noekeon; in questo modo in quei casi particolari in cui sia il cifratore che il decifratore siano richiesti “contestualmente” si ha un chiaro dimezzamento del codice/dimensione del circuito necessario.

Noekeon può funzionare in due modalità differenti che ne caratterizzano la velocità di risposta e il grado di sicurezza adottato.
La prima, che garantisce una maggiore protezione contro gli attacchi legati alla “debolezza” della chiave di cifratura adottata, necessita di un key schedule consistente nella conversione della chiave di cifratura a 128 bit (Cipher Key) nella Working Key (anch’essa di 128 bit) utilizzata poi durante la successiva evoluzione dell’algoritmo. La conversione da Cipher Key a Working Key avviene tramite l’applicazione di Noekeon alla Cipher Key ed utilizzando una Working key composta da soli ‘0’ (ciò rappresenta un’importante novità soprattutto nell’hardware dedicato in cui lo stesso circuito può essere riutilizzato sia per il key schedule che per la cifratura vera e propria).
Tuttavia è anche definita una modalità di funzionamento, vulnerabile agli attacchi related-key, in cui il key schedule non è applicato, chiamata di direct-key in quanto la working key e la cipher key coincidono.

Modalità di key schedule

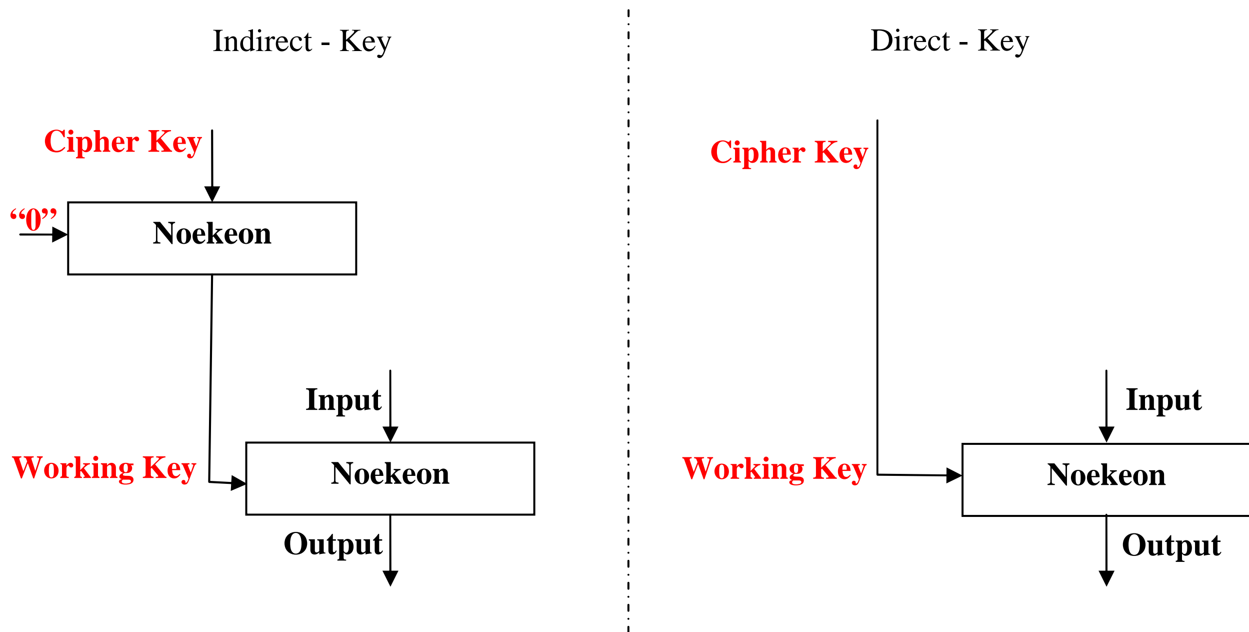


Fig. 1.1

2. Specifiche formali di Noekeon

- Stato e Cipher Key: L’input, l’output e la cipher key usate da Noekeon hanno un’interfaccia esterna costituita da un array monodimensionale composto da 16 byte numerati da 0 a 15 , tuttavia le varie trasformazioni intermedie non operano su nessuna delle tre componenti viste, bensì su di un risultato intermedio chiamato Stato (a) , caratterizzato da 4 parole di 32 bit denominate a[0], .. a[3]. Lo stato coincide con l’input prima dell’esecuzione dell’algoritmo di cifratura e l’output è estratto dallo stato al termine dell’algoritmo stesso.

- **Modularità di Noekeon:** Noekeon è un algoritmo di cifratura iterativo che consiste nell’applicazione ripetuta dello stesso round di trasformazione. Il numero di ripetizioni del Round è denotato da Nr.

- **Round :** La trasformazione dovuta al round può essere vista come la composizione di diverse trasformazioni. Usando una notazione pseudo-C si ha che:

```
Round(Key,State,Constant1,Constant2) {
      State[0] ^= Constant1;
      Theta(Key,State);
      State[0] ^= Constant2;
      Pi1(State);
      Gamma(State);
      Pi2(State);
}
```

(Dove ricordiamo che il simbolo ^ rappresenta uno XOR bit-wise).

Nella notazione proposta le “funzioni” (Round, Theta, Pi1, Gamma, Pi2) operano tutte sullo stato, mentre è fondamentale specificare il fatto che in ogni condizione di funzionamento sempre una tra

le due costanti è pari a zero. In particolare se stiamo utilizzando l'algoritmo di cifratura Constant2 sarà pari a zero, mentre se stiamo utilizzando l'algoritmo di decifratura si avrà che Constant1 sarà sempre nulla.

Come evidenziato dall'algoritmo presentato, ogni round può essere pensato come composto da cinque componenti distinte analizzabili e sintetizzabili separatamente:

- Processo di addizione delle costanti;
- Theta;
- Pi1;
- Gamma;
- Pi2;

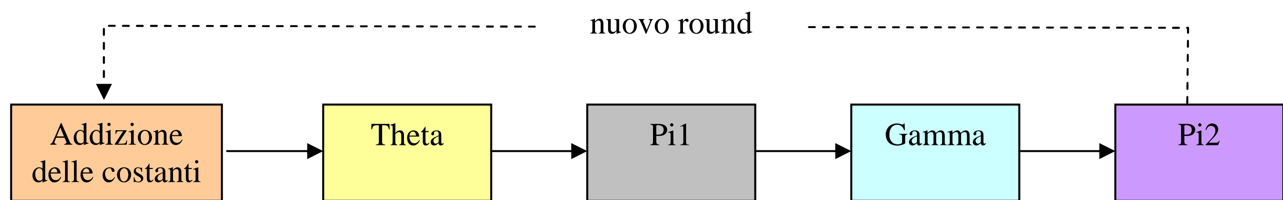


Fig. 2.1

- **Definizione ed addizione delle costanti :** Il processo di addizione delle costanti relative allo specifico round è utilizzato per rompere la simmetria tra le parole che costituiscono lo stato e tra i round stessi. Le costanti utilizzate durante l'algoritmo sono nella forma:

```
Roundct[i]= ('00','00','00',RC[i])
```

Dove si nota che solo il byte meno significativo è diverso da '0'.
La varie costanti (da RC[1] a RC[Nr]) vengono calcolate in maniera ricorsiva seguendo la seguente procedura:

```
RC[0]= 0x80;
if (RC[i]&0x80 != 0)
      RC[i+1] = RC[i]<<1 ^ 0x1B
else
      RC[i+1] = RC[i]<<1;
```

se desiderato, le stesse costanti possono essere calcolate in ordine inverso:

```
RC[Nr]= 0xD4;
if (RC[i]&0x01 != 0)
      RC[i-1] = RC[i]>>1 ^ 0x8D
else
      RC[i-1] = RC[i]>>1;
```

(Ricordiamo che un'operazione del tipo `variabile>>x` corrisponde ad uno shift semplice verso destra di x posizioni)

- **Theta :** Theta corrisponde ad una trasformazione lineare che riceve in input la working key K ed opera sullo stato a attraverso tre fasi distinte:

- modifica delle parole dispari;

- addizione della working key;
- modifica delle parole pari.

```
Theta(k,a){
      temp = a[0]^a[2]; temp ^= temp>>>8 ^ temp<<<8;
      a[1] ^= temp;
      a[3] ^= temp;
      a[0] ^= k[0]; a[1] ^= k[1]; a[2] ^= k[2]; a[3] ^= k[3];
      temp = a[1]^a[3]; temp ^= temp>>>8 ^ temp<<<8;
      a[0] ^= temp;
      a[2] ^= temp;
}
```

L'operazione inversa di Theta può essere espressa come `Theta(k',a)` dove k' è la working key ottenuta applicando Theta a k con una chiave nulla (composta cioè da soli '0').
(Ricordiamo che un'operazione del tipo `variabile>>>x` corrisponde ad uno shift ciclico verso destra di x posizioni)

- **Operazioni di Shift :** I macroblocchi Pi1 e Pi2 eseguono shift ciclici con offset diversi su tre delle 4 parole che costituiscono lo stato. In particolare si ha che:

```
Pi1(a){ a[1] <<<= 1; a[2] <<<= 5; a[3] <<<= 2;}
Pi2(a){ a[1] >>>= 1; a[2] >>>= 5; a[3] >>>= 2;}
```

Si noti che le operazioni Pi1 e Pi2 sono l'una l'inversa dell'altra.

- Gamma : Gamma è una trasformazione non lineare che opera in relazione al seguente schema:

```
Gamma(a){
      a[1] ^= ~a[3]&~a[2];
      a[0] ^= a[2]& a[1];
      tmp = a[3]; a[3] = a[0]; a[0] = tmp;
      a[2] ^= a[0]^a[1]^a[3];
      a[1] ^= ~a[3]&~a[2];
      a[0] ^= a[2]& a[1];
}
```

Gamma ha l'effetto di operare indipendentemente su 32 quartetti di bit, chiamati boxes, ottenuti mediante i 4 bit di ciascuna parola costituente lo stato che sono nella stessa posizione. Per esempio, la box 9 sarà costituita dal nono bit di a[0], dal nono bit di a[1], dal nono bit di a[2] e dal nono bit di a[3]. Il valore della box 9 può essere quindi rappresentato mediante il valore esadecimale che corrisponde al valore dei bit presi nell'ordine 3, 2, 1, 0.

Una rappresentazione di Gamma più compatta può essere ottenuta proprio attraverso le boxes definite in precedenza; in particolare si ha che:

Input (a3 a2 a1 a0)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Output (b3 b2 b1 b0)	7	A	2	C	4	8	F	0	5	9	1	E	3	D	B	6

- **Il cifratore :** Il cifratore consiste nella esecuzione di un numero prestabilito di round, seguito da un'ulteriore applicazione di Theta. In particolare si ha che:

```
Noekeon(WorkingKey,State) {
      For( i=0 ; i<Nr ; i++) Round(WorkingKey,State,Roundct[i],0);
      State[0] ^= Roundct[Nr];
      Theta(WorkingKey,State);
```

```
}
```

Il decifratore, invece è invece descritto dal seguente algoritmo:

```
InverseNoekeon(WorkingKey,State) {
Theta(NullVector,WorkingKey);
      For( i=Nr ; i>0 ; i--) Round(WorkingKey,State,0,Roundct[i]);
      Theta(WorkingKey,State);
      State[0] ^= Roundct[0];
}
```

Il numero dei round **Nr** è 16.
Nella modalità *indirect-key* la Working Key è derivata dalla Cipher Key tramite applicazione di Noekeon con una Working Key nulla (composta cioè da soli '0').
Nella modalità *direct-key* invece la Working Key coincide con la Cipher Key.
Si noti infine che in entrambe le modalità la Working Key non viene modificata durante l'esecuzione dei vari round.

3. Modellizzazione di Noekeon come macchina a stati

L'implementazione dell'algoritmo di cifratura Noekeon da me proposta sfrutta l'evidente modularità dell'algoritmo stesso.
L'identificazione dei "blocchi" suggerita nelle specifiche è stata la stessa da me adottata (Gamma, Theta, Pi1, Pi2); ciò ha consentito di realizzare un'architettura abbastanza lineare a discapito di implementazioni particolarmente creative (magari più efficienti), poiché si è privilegiata un'analisi dei moduli basata sul loro comportamento effettivo (trasformazioni lineari, non lineari, semplici shift…).
In questo modo si potrà andare a costruire un checker che implementi diverse tecniche di rivelazione dei guasti (replica, codici di parità …) a seconda della funzionalità implementata da ciascun modulo.
Si è inoltre deciso di utilizzare il paradigma delle FSM per modellizzare il comportamento del cifratore.

- **il process delta :** Questo processo ha lo scopo di determinare lo stato prossimo a partire dallo stato presente e dal vettore degli ingressi. Il processo è puramente combinatorio e può essere ben modellizzato attraverso un semplice costrutto `case`

  ```
  delta: process ( I, PS )
  begin
        case PS is
              when state0 =>
                    NS <= …
              …
              when stateN =>
                    NS <= …
              when others =>
                    --Exception code
                    NS <= …
        end case;
  end process;
  ```

- **il process lambda :** Questo processo, nel caso più generale, ha lo scopo di determinare l'uscita a partire dallo stato presente e dal vettore degli ingressi. Il processo è puramente combinatorio e produce come uscita un insieme di segnali temporanei Y che sono poi propagati all'uscita mediante il process `output`. La struttura più generale di lambda è la seguente:

```
lambda: process ( I, PS )
begin
      case PS is
            when state0 =>
                  Y <= …
            …
            when stateN =>
                  Y <= …
            when others =>
                  --Exception code
                  Y <= …
      end case;
end process;
```

- **il process state :** Il process `state` è semplicemente un registro parallelo – parallelo che, in corrispondenza di un fronte di clock memorizza in PS il valore dello stato prossimo NS. Inoltre, il processo si occupa di riportare la macchina nello stato di reset in corrispondenza del valore attivo del segnale RESET. Nel seguito è riportata la struttura del process `state` di una FSM con reset sincrono.

```
status: process ( CLK )
begin
      if( CLK'event and CLK = '1' ) then
            if( RESET = '1' ) then
                  PS <= reset_state;
            else
                  PS <= NS;
            end if;
      end if;

end process;
```

- **il process output :** Il process `output`, come il process `state`, è semplicemente un registro parallelo – parallelo, dotato anch'esso di reset. La forma tipica di tale processo è pertanto la seguente

```
output: process ( CLK )
begin
      if( CLK'event and CLK = '1' ) then
            if( RESET = '1' ) then
                  U <= reset_state;
            else
                  U <= Y;
            end if;
      end if;

end process;
```

Tramite una tale implementazione l'uscita U viene aggiornata ad ogni fronte di salita del clock, rimanendo pertanto stabile durante la fase di definizione dei segnali interni e dello stato.

La FSM proposta ha un insieme degli stati di cardinalità pari a 6. In particolare si ha :

```
type STATUS is (RST, S0, S1, S2, S3, CPH);
```

dove:
- RST rappresenta lo stato di reset della macchina;
- S0 indica che è stata appena effettuata una trasformazione di tipo Theta
- S1 indica che è stata appena effettuata una trasformazione di tipo Pi1
- S2 indica che è stata appena effettuata una trasformazione di tipo Gamma

- S3 indica che è stata appena effettuata una trasformazione di tipo Pi2
- CPH è uno stato che indica la fine dell'algoritmo di cifratura per la parola in ingresso. Inoltre è fondamentale ricordare che CPH porta in stato di deadlock la macchina in quanto è necessario un impulso di reset per dare inizio alla cifratura di una nuova parola di ingresso.

Il ritardo della macchina così sintetizzata è dovuto a diverse componenti e, ricordando che:

- il numero di stati costituenti un round è 4 (S0,S1,S2,S3)
- il numero totale di round da eseguire è 16
- devo eseguire un'ultima trasformazione di tipo Theta alla fine dell'esecuzione dei 16 round
- il risultato temporaneo Y ha bisogno di un ulteriore ciclo di clock per essere riportato in uscita

si ha che la parola cifrata è disponibile in uscita dopo un numero di clock pari a 66 (= 4*16 + 1 + 1). Tale ritardo è di fondamentale importanza là dove si voglia realizzare una versione di Noekeon che cifri più parole consecutivamente. In tale caso sarà infatti sufficiente pilotare l'intero circuito attraverso lo stesso clock: basterà progettare un opportuno divisore di clock che comandi il buffer di ingresso delle parole da cifrare e contestualmente il comando di reset della FSM.

4. Implementazione di Noekeon su FPGA

La realizzazione su scheda di Noekeon impone la creazione di opportuni moduli VHDL che vadano a definire un sistema che implementi correttamente la funzionalità proposta secondo il paradigma delle FSM presentato nel paragrafo precedente. A tal proposito si è adottato un approccio di tipo bottom – up in cui si è partiti dalla definizione dei moduli base Theta, Gamma, Pi1 , Pi2 che sono pensati come entità "autonome" realizzati attraverso logica puramente combinatoria, per poi passare ad un livello di astrazione più alto in cui si è sviluppato un ulteriore modulo che implementa la logica di controllo necessaria per il coordinamento delle componenti base e che incorpora i `process delta, lambda, state , output` essenziali per la costituzione dello scheletro di una qualsiasi FSM.

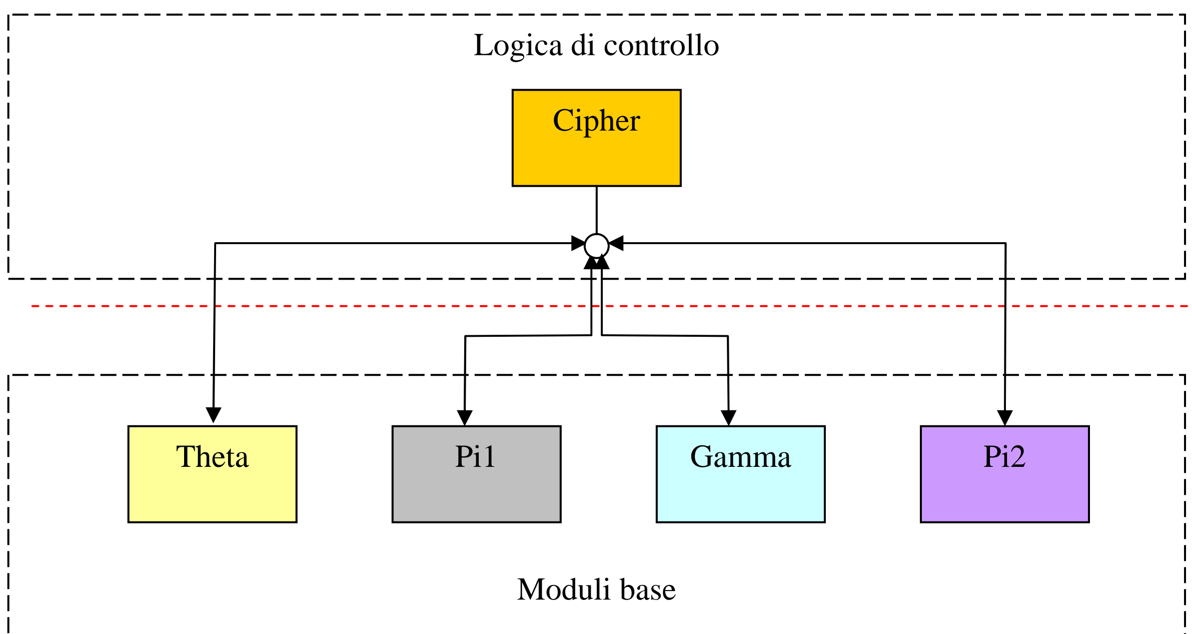
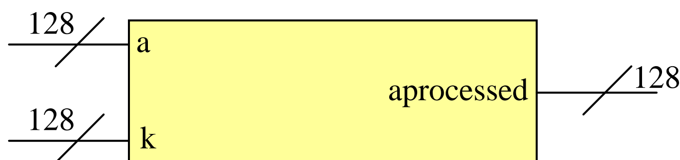


Fig 4.1

Come si può notare dalla figura, un approccio di tale genere porta alla costituzione di un sistema in cui le comunicazioni tra i moduli base avvengono esclusivamente in verticale attraverso l'unita di controllo denominata Cipher , e non in orizzontale come invece una sintesi "per round" avrebbe richiesto (vedi fig. 2.1).
Ciascun modulo base avrà quindi un opportuna interfaccia che consentirà la comunicazione con il modulo di livello superiore il quale gestirà il flusso di dati attraverso un canale di comunicazione condiviso costituito da un segnale interno di 128 bit denominato `currState`.
Un' analisi dettagliata di ciascuna componente può fornire ulteriori dettagli utili a comprendere le caratteristiche del sistema che si è definito.

- **Theta:** l'interfaccia del modulo Theta è composta da tre parole di 128 bit, due di input e una di output. In particolare le due parole in input non sono altro che lo stato a e la Working Key k, mentre l'output rappresenta il valore dello stato dopo la trasformazione lineare subita.



Usando una notazione VHDL tale componente sarà dunque descritta dal seguente template di instanziazione

```
component theta is
Port ( k : in std_logic_vector(0 to 127);
       a : in std_logic_vector(0 to 127);
       aprocessed : out std_logic_vector(0 to 127)
      );
end component;
```

- **Pi1 – Pi2 :** I blocchi Pi1 e Pi2 operano esclusivamente sullo stato a, producendo in output il risultato di semplici operazioni di shift.

Tali entità saranno quindi descritte quindi da uno stesso codice di instanziazione; a titolo di esempio si riporta il template riferito a Pi1.

```
component Pi1 is
Port ( a : in std_logic_vector(0 to 127);
       ashifted : out std_logic_vector(0 to 127)
      );
end component;
```

- **Gamma:** equivalentemente ai moduli Pi1 e Pi2, Gamma effettua le sue computazioni operando esclusivamente sullo stato a, producendo un risultato a_out avente le stesse dimensioni della parola di ingresso (128 bit).

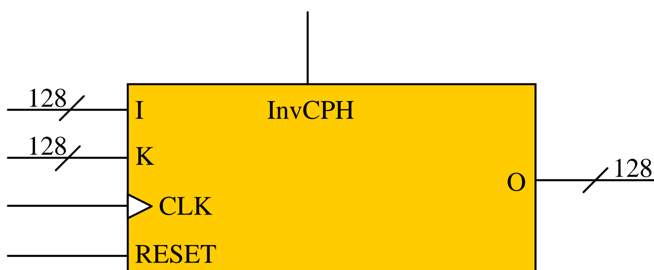
Per completezza di seguito è riportato anche il template di instanziazione di Gamma

```
component Gamma is
Port ( a : in std_logic_vector(0 to 127);
       a_out : out std_logic_vector(0 to 127)
      );
end component;
```

- **Cipher:** come già accennato in precedenza, tale componente costituisce l'elemento principale dell'intero sistema. Esso infatti incorpora una serie di funzionalità che permettono il controllo e la gestione dei flussi di dati con le entità del livello sottostante. In particolare esso è provvisto di una logica interna che consente:

  - la definizione delle nuove costanti necessarie all'inizio di ogni nuovo round;
  - la caratterizzazione della macchina a stati necessaria per il corretto funzionamento del sistema (par. 3);
  - la gestione del canale di comunicazione condiviso realizzato attraverso un segnale interno di 128 bit (`currState`).

  Va inoltre sottolineato il fatto che l'interfaccia del modulo Cipher coincide con la visione ai morsetti dell'utilizzatore finale. Una sua schematizzazione è presentata in figura:



I dati di input sono quindi rappresentati dalla parola da cifrare I, la Cipher Key K, il bit di selezione InvCPH, attivo alto, che consente di effettuare una operazione di decifratura sulla parola in ingresso anziché una classica operazione di cifratura.
Inoltre sono presenti un segnale di segnale di reset ed un segnale clock CLK, necessari rispettivamente per l'inizializzazione e l'evoluzione della FSM.
L'output è ovviamente costituito da una parola da 128 che rappresenta la parola cifrata/decifrata dopo l'esecuzione di Noekeon.

La descrizione in VHDL di tale entità è la seguente

```
entity cipher is
   port ( I: in std_logic_vector(0 to 127);
          K: in std_logic_vector(0 to 127);
          CLK: in std_logic;
          RESET: in std_logic;
          InvCPH: in std_logic; --if(0) then Cipher else Cipher^(-1)
          O: out std_logic_vector(0 to 127)
         );
end cipher;
```

Come sottolineato più volte, la comunicazione tra le varie componenti del sistema avviene attraverso il segnale interno `currState`. Tale segnale, in effetti, costituisce lo stato corrente a in ingresso a ciascun modulo.

```
signal currState : std_logic_vector(0 to 127);

Th: theta
    port map ( currK, currState, ThOUT);

P1: pi1
    port map ( currState, P1OUT);

G : gamma
    port map ( currState, GOUT);

P2: pi2
    port map ( currState, P2OUT);
```

In particolare, ricordando la particolare struttura del round (fig 2.1) e gli stati della FSM associati a ciascun modulo (par. 3), si ha che:

- `PS = RST` → `currState = I;`
- `PS = S0` → `currState = ThOUT;`
- `PS = S1` → `currState = P1OUT;`
- `PS = S2` → `currState = GOUT;`
- `PS = S3` → `currState = P2OUT;`
- `PS = CPH` → `currState = ThOUT;`

dove , con PS , si vuole indicare lo stato presente in cui si trova la macchina a stati caratterizzante il circuito, mentre l'ultima riga serve a garantire l'ulteriore applicazione di Theta al risultato intermedio al termine degli Nr round prima che il risultato finale venga propagato in uscita.

Un'analisi di questo tipo fornisce quindi le basi per una concreta realizzazione dell'algoritmo proposto ma tuttavia una tale soluzione rende il circuito praticamente per nulla orientato alla tolleranza di eventuali guasti . Scopo del paragrafo successivo sarà quello di andare a modificare l'architettura del sistema per creare un sistema altamente tollerante ai guasti.

5. Creazione di un sistema tollerante ai guasti

Una versione preliminare della struttura di un sistema tollerante ai guasti può essere definita utilizzando il concetto di ridondanza dello spazio. Utilizzando un tale approccio difatti non si fa altro che replicare una o più volte la stessa funzione per poi analizzare la correttezza dei risultati conseguiti semplicemente andando a confrontare i risultati prodotti in uscita da ogni singolo modulo.

In particolare se si vuole esclusivamente andare ad individuare un eventuale comportamento anomalo, si può procedere duplicando la funzione di interesse per poi confrontare tra loro i due output così generati; ovviamente se vi è una differenza nei risultati ottenuti si può concludere che è occorso un errore in uno dei due moduli, senza tuttavia poter identificare l'eventuale modulo guasto.
Un approccio più dispendioso in termini di area occupata, ma certamente più efficace in quanto in grado non solo di rilevare un eventuale guasto, ma anche il particolare modulo affetto da disturbo prevede la triplicazione della funzione desiderata (TMR). In questo modo, nell'ipotesi che possa verificarsi solo un guasto alla volta, è possibile andare a determinare il modulo danneggiato attraverso dei confronti incrociati sugli output dei tre singoli moduli.

In prima analisi si è deciso di utilizzare una tecnica di tipo TMR per tollerare eventuali guasti all'interno del circuito. Ciò ha comportato la triplicazione delle funzioni Theta, Pi1, Gamma, Pi2 , oltre che all'aggiunta di un nuovo stato nella sintesi della FSM relativa, denominato ERR che chiaramente denota l'occorrenza di un eventuale errore nella fase computazionale dell'algoritmo.

Come si può notare dalla tabella seguente un tale approccio porta tuttavia ad un "esplosione" della dimensione del circuito su scheda. Ciò può far pensare che l'approccio TMR non sia il più corretto da adottare in un circuito che, per quanto strutturalmente non eccessivamente complicato, ciò nonostante opera su parole di dimensione abbastanza elevata (128 bit). Per tale ragione forse l'orientamento più appropriato è rappresentato da un sistema che faccia tolleranza ai guasti mediante tecniche di ridondanza dei dati. L'overhead dovuto all'aggiunto di un pacchetto ad esempio di 8 bit su di una parola di 128 bit si può considerare infatti trascurabile, inoltre il fatto che le funzioni implementate dai moduli base siano di tipo matematico fornisce una spinta ulteriore verso tale soluzione che, con la scelta di opportuni codici di controllo sui dati può risultare altamente tollerante e performante allo stesso tempo.

Utilizzazione del dispositivo privo di sistema per tolleranza ai guasti	
Number of Slices:	1036 out of 1920 53%
Number of Slice Flip Flops:	685 out of 3840 17%
Number of 4 input LUTs:	1839 out of 3840 47%
Number of bonded IOBs:	387 out of 173 223% (*)
Number of GCLKs:	4 out of 8 50%
WARNING:Xst:1336 - (*) More than 100% of Device resources are used	

Utilizzazione del dispositivo utilizzando tecnica TMR sui moduli base	
Number of Slices:	2367 out of 1920 123% (*)
Number of Slice Flip Flops:	1452 out of 3840 37%
Number of 4 input LUTs:	4327 out of 3840 112% (*)
Number of bonded IOBs:	387 out of 173 223% (*)
Number of GCLKs:	4 out of 8 50%
WARNING:Xst:1336 - (*) More than 100% of Device resources are used	