

Chapter 8

Enigma

8.1 Storia

La macchina "Enigma" è un dispositivo elettro-meccanico usato per le comunicazioni cifrate. Inventata nel 1918, fu inizialmente utilizzata in ambito commerciale; I primi modelli erano alimentati a batteria. Insieme ad una tastiera, il dispositivo era formato da una tavola contenente 26 lettere ognuna con una piccola lampadina posta dietro. Dal 1920 in poi, questo dispositivo fu adottato dall'esercito Tedesco e fu ampiamente sfruttato in seguito, durante la II Guerra Mondiale. Attraverso quest'ultima, era possibile cifrare informazioni sensibili e scambiare in completa sicurezza senza che potessero cadere in mano degli Alleati.

La sicurezza della macchina era resa possibile dalle impostazioni che venivano cambiate quotidianamente, basandosi su una chiave segreta

distribuita precedentemente.

Inoltre, altra caratteristica fondamentale per assicurare l'impossibilità della decriptazione, ogni qual volta si digitava un messaggio, un meccanismo interno della macchina faceva sì che le impostazioni stesse per la criptazione cambiassero.

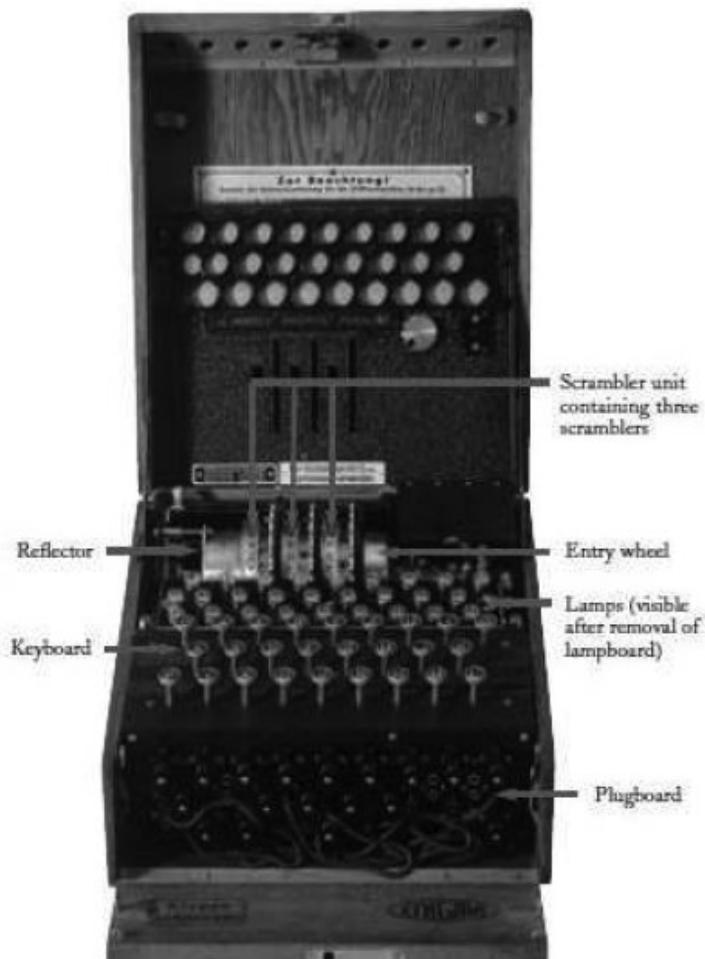
Il ricevente del messaggio criptato non doveva fare altro che usare le stesse identiche impostazioni adottate dal mittente per decriptare con successo il messaggio.

Diversi modelli della macchina Enigma sono stati prodotti, ma i modelli creati dall'esercito Tedesco erano i più complessi, implementando al loro interno una "plugboard".

L'iniziale avanzata delle forze Tedesche era reso possibile soprattutto grazie a questo enorme vantaggio: le trasmissioni radio erano sicure e impossibili da decriptare.

Questo fu vero fino al 1932, quando Marian Rejewski, matematica Polacca, riuscì a capire il funzionamento della plugboard attraverso la teoria delle permutazioni.

Grazie a quest'ultima e alle informazioni ottenute dalla spia francese Hans-Thilio Schmidt, che ottenne le chiavi segrete utilizzate quotidianamente nei mesi successivi, fu possibile fare breccia nel sistema di crittografia Tedesco e a ribaltare completamente l'andamento della guerra.



8.2 Funzionamento

Come altre macchine a rotore, la macchina "Enigma" è una combinazione di sistemi meccanici e elettrici.

I sistemi consistono in:

- **Tastiera:** formata da 26 lettere. Rendeva possibile inserimento delle lettere da criptare.
- **Rotori:** Un set di dischi rotanti disposti in maniera adiacente. Formano il cuore della macchina Enigma. Ogni rotore ospita 26

pin disposti su una faccia circolare del rotore e 26 collegamenti elettrici disposti sull'altra faccia. I pin e i contatti rappresentano l'alfabeto.

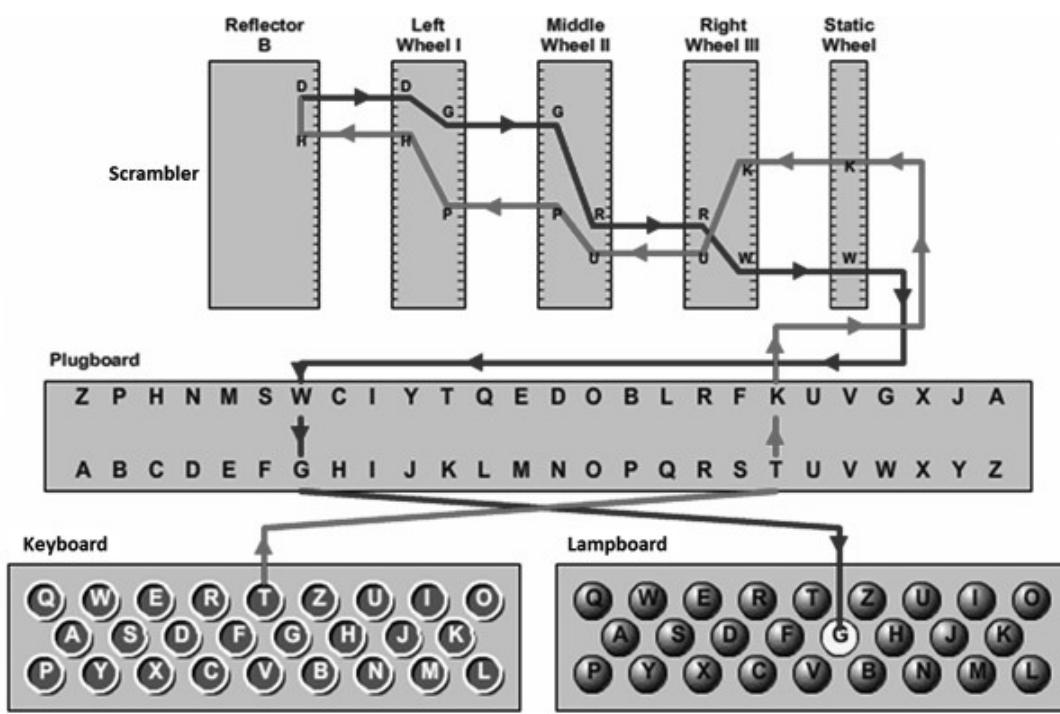
- **"Stepping":** Per evitare che fosse una semplice macchina che effettuava un semplice cifrario a sostituzione, ogni pressione dei tasti faceva sì che i rotori, attraverso questo sistema meccanico, si spostassero di un ventiseiesimo di una rotazione completa. Cio' ha fatto sì che la sostituzione crittografica fosse diversa ad ogni nuova posizione del rotore, producendo un cifrario a "sostituzione polialfabetica" più efficiente
- **Ruota d'ingresso:** La ruota d'ingresso collega la scheda plugboard al gruppo rotore. Nel caso in cui la plugboard non fosse presente, questa rotella collega invece la tastiera e la lampada al gruppo rotore.
- **Riflettore:** L'ultimo rotore prevede un riflettore. Mai implementato precedentemente da nessuna macchina a rotore del periodo, il riflettore collegava le uscite dell'ultimo rotore a due a due, reindirizzando la corrente attraverso gli altri rotori seguendo un percorso diverso. Il riflettore assicurava che Enigma sarebbe stato auto-reciproco; con due macchine configurate in modo identico, un messaggio poteva essere crittografato su uno e decrittografato sull'altro senza la necessità di un meccanismo ingom-

brante per passare tra le modalità crittografia e decrittografia.

- **Plugboard:** La plugboard consentiva un cablaggio variabile che poteva essere riconfigurato dall'operatore. Fu introdotto nelle versioni dell'esercito Tedesco nel 1928. La plugboard ha contribuito ad aumentare la forza crittografica e quindi all'efficienza della macchina Enigma, mettendo a disposizione più di 150 trilioni di impostazioni differenti.

L'utilizzo della macchina era molto semplice: Dopo aver impostato lo stato iniziale del macchinario (chiave giornaliera), bastava scrivere attraverso la tastiera un messaggio. Per ogni lettera premuta, una lampada si illuminava indicando una lettera differente. La lettera indicata dalla lampada veniva scritta da un secondo operatore, che si occupava di salvare il messaggio cifrato. La pressione di un tasto faceva sì che all'interno della macchina si spostasse uno o più rotore in modo tale che la lettera successiva venisse sostituita in "un altro modo", anche se quest'ultima fosse stata una lettera uguale alla precedente. Per ogni chiave premuta quindi, avveniva la rotazione di almeno un rotore, generalmente quello di destra, e meno spesso la rotazione degli altri due, restituendo così una sostituzione diversa per ogni lettera usata nel messaggio. Questo processo continuava finché il messaggio era completo. Il messaggio criptato veniva trasmesso in genere, tramite radio usando il codice morse, ad un altro operatore di un'altra macchina Enigma. Questo operatore non doveva fare altro che scrivere il testo cifrato e,

finchè le impostazioni della seconda macchina erano identiche a quelle della prima, per ogni chiave premuta avveniva la sostituzione inversa e il messaggio originale veniva recepito.



8.3 Progetto

Enigma, come detto in precedenza è una macchina molto complessa che nella sua versione completa prevede fino a 5 rotori. Per il progetto e la realizzazione su scheda si è pensato di apportare una semplificazione, ovvero di produrre una macchina a singolo rotore che però funzionasse come enigma.

Si è deciso di non prevedere la plugboard (presente nella macchina originale) poichè essa potrebbe essere facilmente riprodotta cambiando la configurazione della tastiera.

Il progetto quindi prevede un sistema che si occupa del prelievo dei dati da tastiera tramite il protocollo PS/2, dopodichè ci sarà la macchina progettata e poi un sistema che si occupa di mostrare i dati.

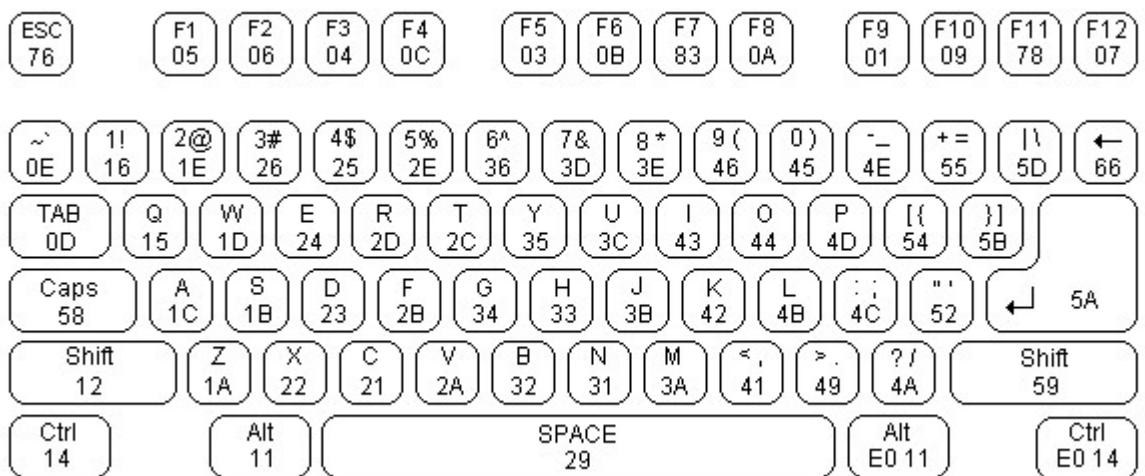
8.3.1 PS/2

Il protocollo PS/2 è un protocollo **sincrono seriale bidirezionale**. Quando le linee di dati e di clock sono alte, il bus è inattivo e la tastiera collegata può iniziare a trasmettere i dati; l'host può inibire la trasmissione in qualsiasi momento tirando la linea di clock in basso per 100 microsecondi. Il dispositivo genera sempre il segnale di clock e se l'host vuole comunicare può farlo tirando la linea di clock in basso (inibizione della trasmissione da parte del dispositivo), tirando la linea dati in basso e quindi rilasciando la linea di clock: questa è la richiesta di stato di invio, così facendo dice al dispositivo di iniziare a generare

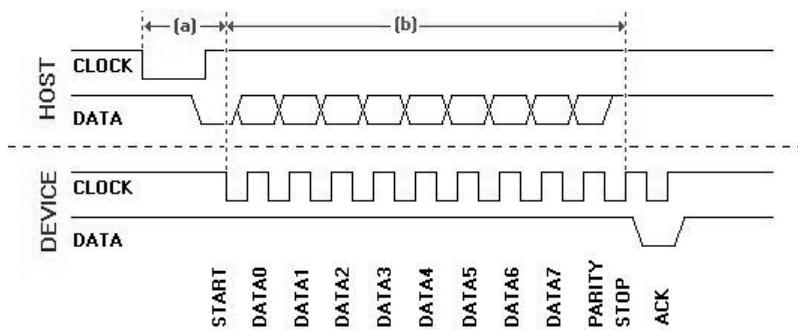
impulsi di clock. Un frame di dati è composto da 11 o 12 bit (a seconda della direzione dei dati):

- un bit di avvio (sempre basso)
- 8 bit di dati, prima LSB
- un bit di parità dispari
- un bit di stop (sempre alto)
- un bit di conferma (il dispositivo abbassa la linea dati quando trasmette dall'host al dispositivo)

Comunicazione da dispositivo a host: Il dispositivo controlla lo stato della linea di clock: se è alta inizia a trasmettere dati (la linea di clock deve essere continuamente alta per almeno 50 microsecondi prima che il dispositivo inizi a trasmettere). Il dispositivo genera gli impulsi di clock e i dati devono essere stabili sul fronte discendente del segnale di clock e cambiare dopo il fronte ascendente:

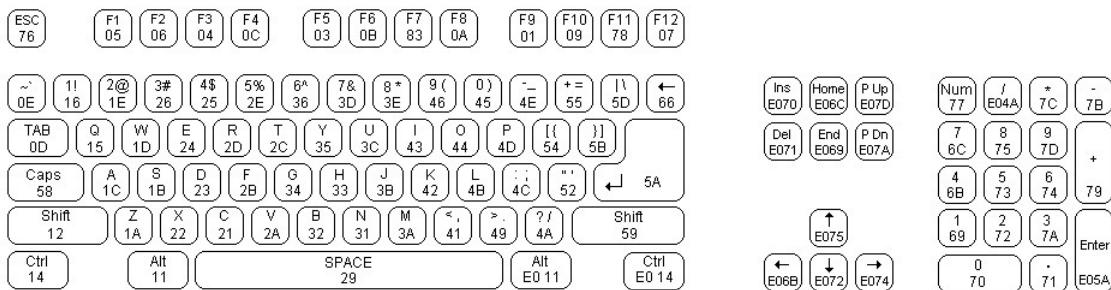


Comunicazione da host a dispositivo: Poiché il dispositivo genera sempre il segnale di clock, l'host deve mettere le linee di clock e dati nella richiesta di **stato di invio** tirando la linea di clock in basso per 100 microsecondi, tirando poi la linea dati in basso e successivamente tirando di nuovo la linea di clock in alto (a). Quando il dispositivo rileva questo stato, inizierà a generare impulsi di clock e clock nei bit di dati del frame (b): l'host cambia i dati quando il clock è basso e il dispositivo campiona la linea dati quando il clock è alto (questo è l'opposto di ciò che accade durante la comunicazione dal dispositivo all'host):



Dopo aver cronometrato il bit di arresto, il dispositivo tira la linea dati in basso per riconoscere i dati, quindi genera l'ultimo impulso di clock e rilascia dati e linee di clock. La comunicazione da host a dispositivo è utile per inviare comandi alla tastiera.

Interfacciamento di una tastiera PS/2: Una tastiera è una matrice di tasti che sono monitorati da un controller on-board, chiamato **encoder** della tastiera. Questo controller controlla quale tasto viene premuto o rilasciato e invia i dati corrispondenti all'host. I dati inviati dal controller all'host sono il **codice di scansione** (il controller esegue la scansione della tastiera per verificare la pressione dei tasti) del tasto che è stato premuto o rilasciato. Ci sono due tipi di codici: il codice make(di esecuzione) e il codice break(di interruzione). Un **codice make** viene inviato ogni volta che un tasto viene premuto o tenuto premuto; un **codice break** viene inviato quando viene rilasciato un tasto. Ogni tasto sulla tastiera ha il suo codice di marca e scansione unico in modo che l'host possa sapere che cosa è successo a un determinato tasto osservando il codice stesso. Tutti i codici di scansione costituiscono un **set di codici di scansione**: ci sono tre set di codici di scansione (set uno, due o tre), e tutte le tastiere moderne utilizzano di default il set di codici di scansione due:



Anche se la maggior parte dei codici di composizione sono lunghi un

byte, esistono alcuni codici di composizione estesi che sono composti da due o quattro byte (tutti questi codici di scansione iniziano con il byte 0xE0). Ogni volta che si preme un tasto, viene inviato all'host un codice di scansione. È importante notare che un codice di scansione corrisponde a un tasto fisico della tastiera e non è associato a un carattere di un particolare set di caratteri; spetta all'host tradurre il codice di scansione nel carattere corrispondente. Quando un tasto viene rilasciato, viene inviato all'host un codice di interruzione: il codice di interruzione è il codice di composizione preceduto da 0xF0 (il codice di interruzione dei tasti estesi è 0xE0, 0xF0 e il codice del tasto). Quando si tiene premuto un tasto, questo diventa **typematic** (questa è una funzione della tastiera che continua a ripetere un tasto finché viene tenuto premuto) e (dopo un breve periodo) la tastiera continuerà a inviare il suo codice di creazione finché non verrà rilasciato o finché non verrà premuto un altro tasto. I dati typematic non vengono memorizzati all'interno della tastiera: quando vengono premuti più tasti, solo l'ultimo diventa typematic e questa ripetizione si interrompe al suo rilascio, anche se altri tasti potrebbero essere ancora premuti. Al reset la tastiera esegue il cosiddetto BAT (Basic Assurance Test) e invia 0xAA in caso di esito positivo o 0xFC in caso di esito negativo (gli eventuali led della tastiera lampeggiano).

Lettura dei codici di scansione dalla tastiera: Il modo più semplice per leggere i dati inviati da una tastiera PS/2 è utilizzare gli **in-**

interrupt esterni. Tutti i microcontrollori consentono di attivare un interrupt quando si verifica un evento specifico su un pin esterno e la maggior parte di essi dispone di un controller di interrupt esterno dedicato. Durante la comunicazione tra dispositivo e host, l'host campiona la linea dati quando il clock è basso, quindi è facile impostare un trigger quando il pin collegato alla linea di clock diventa basso e leggere il codice di scansione dalla tastiera. L'ISR chiama semplicemente **keyboard-handler()** che legge il codice di scansione e poi chiama **keyboard-decode()**, che traduce il codice di scansione in un carattere (viene utilizzato un array come tabella di ricerca) e lo inserisce in un buffer FIFO:

8.3.2 Riflettore

Partiamo nella descrizione dei blocchi della macchina. Il riflettore è il componente più semplice poichè, a differenza del rotore, esso è statico. Tale componente serve a riflettere indietro il segnale facendone tornare ai rotorì e quindi applicare una nuova cifratura.

Progettualmente è una macchina combinatoria con dentro una ROM avente le associazioni del riflettore.

Di seguito riportiamo il codice del componente in vhdl.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
```

```
3 use ieee.numeric_std.all;
4 use work.datatypes.all;
5
6 entity Reflector is
7 Port (
8     clock: in std_logic;
9     reset: in std_logic;
10    data_in: in std_logic_vector(4 downto 0);
11    getLetter: in std_logic;
12    data_out: out std_logic_vector(4 downto 0)
13 );
14 end Reflector;
15
16 architecture Structural of Reflector is
17 component shifting_memory
18 generic (
19     mappatura: mappa := mappa_reflector
20 );
21 Port (
22     data_in: in std_logic_vector(4 downto 0);
23     clock: in std_logic;
24     reset: in std_logic;
25     shift: in std_logic;
```

```
26      data_out: out std_logic_vector(4 downto 0)
27  );
28 end component;
29
30 signal letterOut: std_logic_vector(4 downto 0);
31 begin
32 memory: shifting_memory
33 generic map(
34     mappa_reflector
35 )
36 port map(
37     data_in => data_in,
38     clock => clock,
39     reset => reset,
40     shift => '0',
41     data_out => letterOut
42 );
43 data_out <= letterOut when getLetter = '1' else
44     (others => '1');
45 end Structural;
```

La mappa riflettore è definita in un package chiamato **datatype**s ed ha le seguenti associazioni:

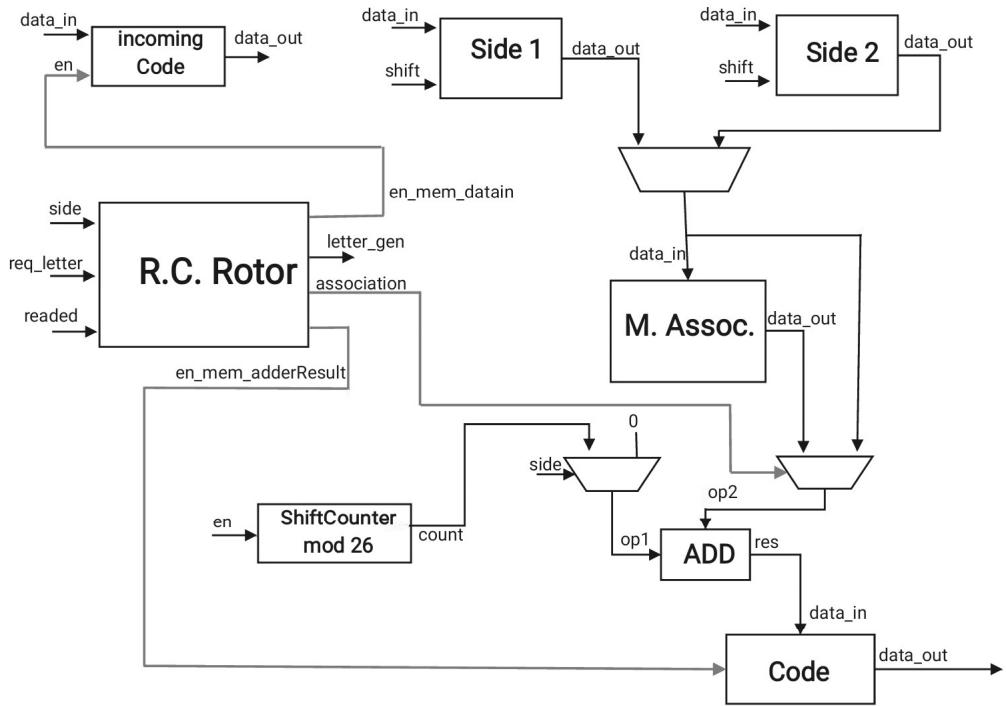
```
1 constant mappa_reflector: mappa(25 downto 0) := (
2     0 => (letter => a, connection => y),
3     1 => (letter =>b, connection =>r),
4     2 => (letter =>c, connection =>u),
5     3 => (letter =>d, connection =>h),
6     4 => (letter =>e, connection =>q),
7     5 => (letter =>f, connection =>s),
8     6 => (letter =>g, connection =>l),
9     7 => (letter =>h, connection =>d),
10    8 => (letter =>i, connection =>p),
11    9 => (letter =>j, connection =>x),
12    10 => (letter =>k, connection =>n),
13    11 => (letter =>l, connection =>g),
14    12 => (letter =>m, connection =>o),
15    13 => (letter =>n, connection =>k),
16    14 => (letter =>o, connection =>m),
17    15 => (letter =>p, connection =>i),
18    16 => (letter =>q, connection =>e),
19    17 => (letter =>r, connection =>b),
20    18 => (letter =>s, connection =>f),
21    19 => (letter =>t, connection =>z),
22    20 => (letter =>u, connection =>c),
23    21 => (letter =>v, connection =>w),
```

```
24    22 => (letter =>w, connection =>v) ,  
25    23 => (letter =>x, connection =>j) ,  
26    24 => (letter =>y, connection =>a) ,  
27    25 =>(letter =>z, connection =>t)  
28 );
```

Le associazioni definite sopra sono state definite in base al documento ufficiale per le configurazioni dei rotori e dei riflettori.

8.3.3 Rotore

Il rotore è il componente di enigma forse più complesso da progettare ed astrarre, poichè nella realtà il processo meccanico che produce le rotazioni, cambia anche le associazioni lettera-lettera. Con il seguente schema architetturale si è cercato di riprodurre il più fedelmente possibile il meccanismo fisico di rotazione del rotore.



Chiaramente, come si può facilmente notare è una macchina complessa, schematizzata come parte operativa + parte di controllo.

Uno dei componenti più importanti nella parte operativa è la Shifting Memory. Essenzialmente che a comando è in grado si "shiftare" a destra le lettere, tale meccanismo simulerebbe la rotazione del rotore. Nel progetto sono state previste due tipi di Shifting Memory, una che prevede solo un alfabeto e l'altra che invece crea delle associazioni lettere-lettere.

La shifting memory è definita dal seguente codice:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
    
```

```
4 use work.datatypes.all;  
5  
6 entity shifting_memoryNoAssoc is  
7 generic (  
8     mappatura: configurazioneRotore :=  
9         mappa_rotore  
10    );  
11 Port (  
12     data_in: in std_logic_vector(4 downto 0);  
13     clock: in std_logic;  
14     reset: in std_logic;  
15     shift: in std_logic;  
16     data_out: out std_logic_vector(4 downto 0)  
17 );  
18  
19 end shifting_memoryNoAssoc;  
20  
21 architecture Behavioral of shifting_memoryNoAssoc  
22 is  
23 signal memoria: configurazioneRotore(25 downto 0)  
24     := mappatura;  
25 begin  
26 mem: process(clock)  
27 begin
```

```

24      if (rising_edge(clock)) then
25          if (reset = '1') then
26              memoria <= mappatura;
27          elsif (shift = '1') then
28              memoria(0) <= memoria(25);
29              for i in 1 to 25 loop
30                  memoria(i) <= memoria(i - 1);
31              end loop;
32          end if;
33      end if;
34  end process;
35 data_out <= memoria(to_integer(unsigned(data_in)))
36                      mod 26);
37 end Behavioral;

```

Per la distinzione associativa non associativa abbiamo due tipi nel pacchetto datatypes che definiscono le varie strutture:

Associativa

```

1 type connection is record
2     letter: std_logic_vector(4 downto 0);
3     connection: std_logic_vector(4 downto 0);
4 end record connection;
5 type mappa is array(natural range<>) of connection

```

;

Non-Associativa

```
1 type configurazioneRotore is array( natural range
<>) of std_logic_vector(4 downto 0);
```

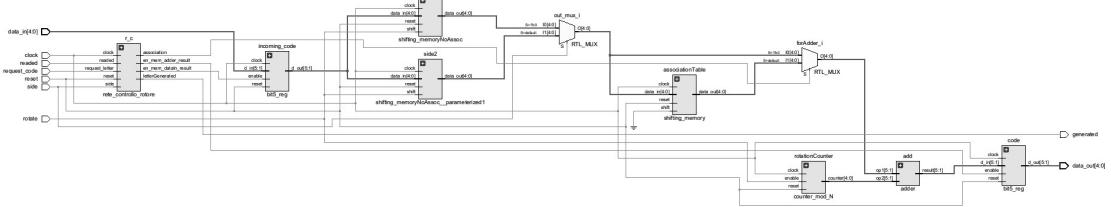
Per il rotore esistono due facce, la prima che prevede la configurazione del rotore, come è stata definita nel documento, l'altra invece prevede l'alfabeto ordinato.

Quando avviene lo shift entrambe le facce ruotano mentre la memoria che prevede le associazioni lettere-lettera no.

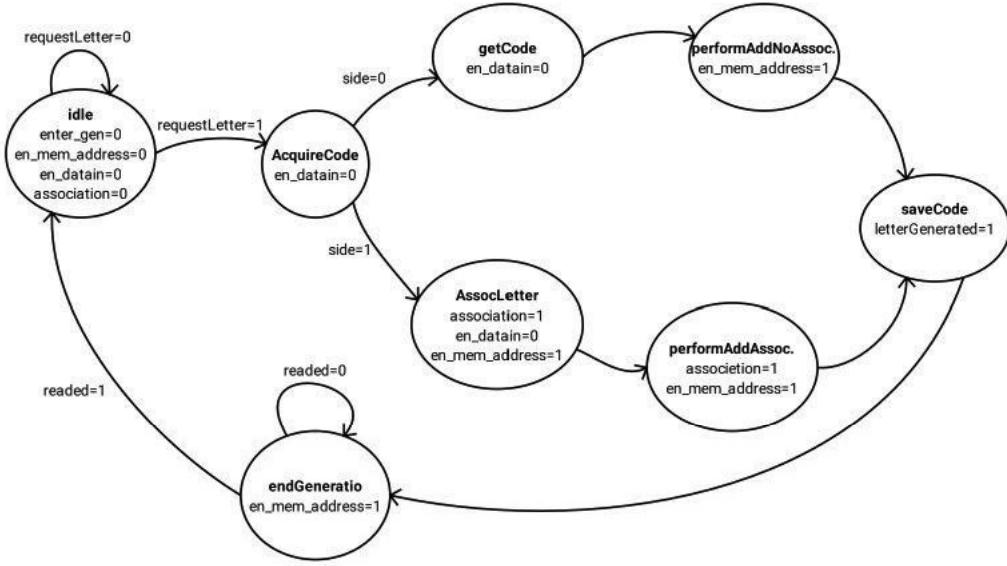
Inoltre nell'architettura è previsto anche un contatore che mi da l'indicazione di quanti shift ha fatto il rotore, indicazione che mi serve nel caso del side 2 per prevedere uno spiazzamento tra le lettere che mi produce l'addizionatore messo a valle del registro codice.

Il multiplexer messo a valle della memoria associativa è pilotato dal side.

Di seguito riportiamo il disegno architetturale restituito da Vivado.



Per quanto riguarda la rete di controllo, quella che gestisce il rotore ha il seguente automa:



Riportiamo una breve descrizione degli stati del seguente automa.

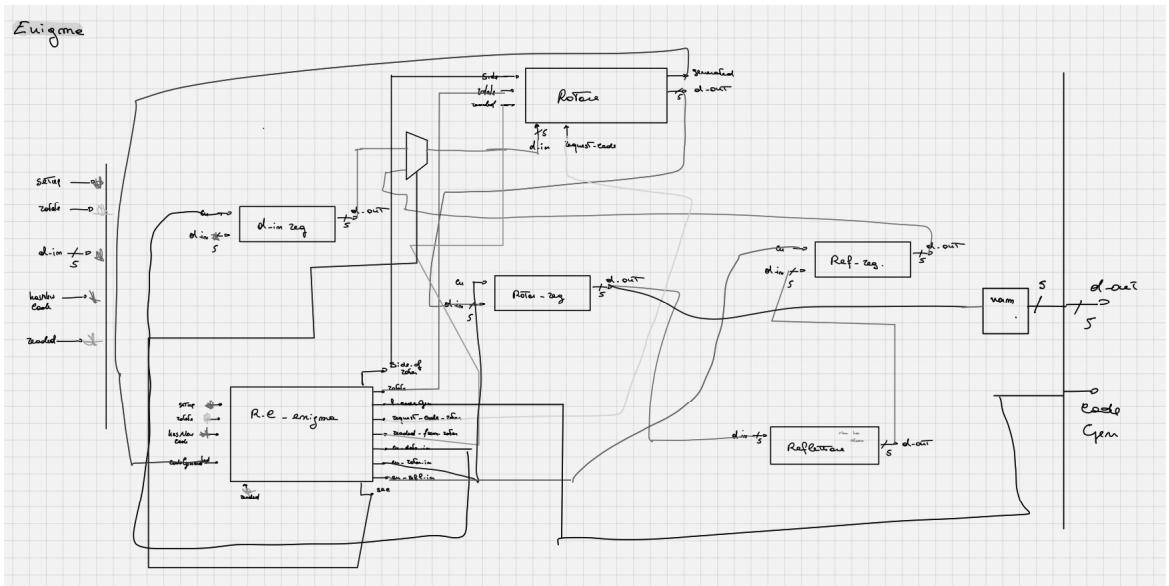
- **idle:** Stato iniziale dell'automa che non produce una nuova lettera crittografata, in questo stato aspettiamo che il sistema enigma "richieda" la lettera al sistema rotore.
- **AcquireCode:** In questo stato abbiamo ricevuto la richiesta di lettera e dunque prendiamo il codice in ingresso al rotore abilitando il registro del dato d'ingresso a memorizzare
- **getCode:** Stato raggiungibile nel caso siamo nella prima facciata del rotore, la quale non deve produrre nessuna associazione, quindi prenderà semplicemente il codice dalla shiftingMemory.
- **performAddNoAssoc:** In questo stato lasciamo all'addizionatore il tempo di produrre in uscita il valore vero e proprio che deve essere spiazzato in base al numero degli shift.

- **SaveCode:** Stato che permette alla memoria a valle di salvare il codice generato dall'adder e di avviare la macchina richiedente che il codice è stato generato.
- **AssocLetter:** Questo stato viene attivato nel caso stiamo nella seconda facciata del rotore, che richiede una associazione lettera-lettera (nella seconda facciata del rotore c'è l'alfabeto ordinato ed eventualmente shiftato). Dunque in questo stato diamo il tempo alla macchina di prelevare il codice dalla memoria associativa.
- **performAddAssoc:** In questo caso l'addizionatore fa passare il dato così com'è senza effettuare spiazzamenti.
- **endGeneration:** Stato di stallo che aspetta che il sistema richiedente abbia letto il dato. È stato progettato per evitare overwrite.

8.3.4 Enigma

Questo è il secondo livello del nostro sistema, ovvero la macchina enigma. Qui colleghiamo opportunamente le varie componenti che poi verranno gestite da una rete di Controllo che gestirà anche il protocollo di lettura dal rotore.

Lo schema dell'architettura è il seguente:



Lo schema è il classico schema ancora di una macchina complessa, composta cioè da parte operativa e parte di controllo.

Abbiamo il classico schema di enigma, e a valle un blocco che ci permette di "aggiustare il codice".

Difatti la macchina funziona in modulo 26, il codice che i componenti possono dare in out è su 5 bit, ovvero in un range da 0 a 32.

Il blocco di elaborazione è stato scritto in maniera behavioral poichè era facilmente interpretabile dal sintetizzatore in termini di componen-tistica.

Di seguito il codice:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use ieee.std_logic_unsigned.all;
4 use ieee.numeric_std.all;
```

```
5 use ieee.math_real.all;
6
7 entity modular_normalize is
8 Port (
9     data_in: in std_logic_vector(4 downto 0);
10    data_out: out std_logic_vector(4 downto 0)
11 );
12 end modular_normalize;
13
14 architecture Behavioral of modular_normalize is
15 signal normalize_result: std_logic_vector(4 downto
16 0);
17 constant subtractor: std_logic_vector(4 downto 0)
18 := "11010";
19 begin
20 norm: process(data_in)
21 begin
22 if (to_integer(unsigned(data_in)) >= 26) then
23     normalize_result <= data_in - subtractor;
24 else
25     normalize_result <= data_in;
26 end if;
27 end process;
```

```
26 data_out <= normalize_result;  
27 end Behavioral;
```

Riportiamo anche di seguito il codice per riprodurre l'architettura:

```
1 library IEEE;  
2 use IEEE.STD_LOGIC_1164.ALL;  
3  
4 entity enigma is  
5 Port (  
6     clock: in std_logic;  
7     reset: in std_logic;  
8     setup: in std_logic;  
9     rotate: in std_logic;  
10    data_in: in std_logic_vector(4 downto 0);  
11    data_out: out std_logic_vector(4 downto 0);  
12    hasNewCode: in std_logic;  
13    readed: in std_logic; —Segnale che mi avvisa  
14      che ho letto l'elemento, me lo da il livello  
15      inferiore  
16      code_generated: out std_logic  
17 );  
18 end enigma;
```

```
18 architecture Structural of enigma is
19 component rotore
20 Port (
21     side: in std_logic; —Segnale che mi indica
22         quale parte devo restituire
23     clock: in std_logic;
24     reset: in std_logic;
25     rotate: in std_logic;
26     readed: in std_logic; —Il livello inferiore
27         avvisa il rotore che ha letto e puo' andare
28         avanti
29     data_in: in std_logic_vector(4 downto 0);
30     request_code: in std_logic;
31     generated: out std_logic;
32     data_out: out std_logic_vector(4 downto 0)
33 );
34 end component;

35 component Reflector
36 Port (
37     clock: in std_logic;
38     reset: in std_logic;
39     data_in: in std_logic_vector(4 downto 0);
```

```
38     getLetter: in std_logic;
39     data_out: out std_logic_vector(4 downto 0)
40   );
41 end component;

42
43 component bit5_reg
44 Port (
45     clock: in std_logic;
46     reset: in std_logic;
47     enable: in std_logic;
48     d_in: in std_logic_vector(5 downto 1);
49     d_out: out std_logic_vector(5 downto 1)
50   );
51 end component;

52
53 component rete_controllo_enigma
54 Port (
55     clock: in std_logic;
56     reset: in std_logic;
57     setup_s: in std_logic;
58     rotate_s: in std_logic;
59     hasNewCode: in std_logic;
60     codeGenerated: in std_logic;
```

```

61      readed: in std_logic;
62      rotate_sign: out std_logic;
63      finaleCodeGenerated: out std_logic;
64      request_code_rotor: out std_logic;
65      readed_from_rotor: out std_logic;
66      en_mem_data_in: out std_logic;
67      en_mem_rotor: out std_logic;
68      en_mem_refl: out std_logic;
69      src: out std_logic; — 0 data, 1 reflector
70      sideOfRotor: out std_logic —Dice al sistema
        rotore quale lato guardare
71  );
72 end component;

73

74 component modular_normalize
75 Port (
76     data_in: in std_logic_vector(4 downto 0);
77     data_out: out std_logic_vector(4 downto 0)
78 );
79 end component;

80

81 signal side_of_rotor, rotor_generate,
        rotor_generated, rotor_rotate, memorize, rotor_read

```

```

    : std_logic := '0';

82 signal reg_out, out_of_rotor, out_of_reflector,
     forReg: std_logic_vector(4 downto 0);

83

84 signal memorize_data_in, memorizeRotor_data,
     memorizeReflector_data: std_logic := '0';

85 signal data_in_reg_out, reg_rotor_out,
     reg_reflector_out, forRotor: std_logic_vector(4
downto 0);

86 signal src: std_logic := '0';

87 signal normalizedLetter: std_logic_vector(4 downto
0);

88 begin

89

90 forRotor <= data_in_reg_out when src = '0' else
91
92         reg_reflector_out when src = '1' else
93         (others => '0');

94 normalizer: modular_normalize
95 port map(
96
97         data_in => reg_rotor_out,
98         data_out => normalizedLetter
99     );

```

```
99 rotore_sys: rotore
100 port map(
101     side => side_of_rotor ,
102     clock => clock ,
103     reset => reset ,
104     rotate => rotor_rotate ,
105     readed => rotor_read ,
106     data_in => forRotor ,
107     request_code => rotor_generate ,
108     generated => rotor_generated ,
109     data_out => out_of_rotor
110 );
111
112 ref: Reflector
113 port map(
114     clock => clock ,
115     reset => reset ,
116     data_in => reg_rotor_out ,
117     getLetter => '1' ,
118     data_out => out_of_reflector
119 );
120
121 codeReflectorReg: bit5_reg
```

```
122 port map(
123     clock => clock ,
124     reset => reset ,
125     enable => memorizeReflector_data ,
126     d_in => out_of_reflector ,
127     d_out => reg_reflector_out
128 );
129
130 codeRotorReg: bit5_reg
131 port map(
132     clock => clock ,
133     reset => reset ,
134     enable => memorizeRotor_data ,
135     d_in => out_of_rotor ,
136     d_out => reg_rotor_out
137 );
138
139 data_in_reg: bit5_reg
140 port map(
141     clock => clock ,
142     reset => reset ,
143     enable => memorize_data_in ,
144     d_in => data_in ,
```

```
145      d_out => data_in_reg_out
146  );
147
148 controllo: rete_controllo_enigma
149 port map(
150     clock => clock ,
151     reset => reset ,
152     setup_s => setup ,
153     rotate_s => rotate ,
154     hasNewCode => hasNewCode ,
155     codeGenerated => rotor_generated ,
156     readed => readed ,
157     rotate_sign => rotor_rotate ,
158     finaleCodeGenerated => code_generated ,
159     request_code_rotor => rotor_generate ,
160     readed_from_rotor => rotor_read ,
161     en_mem_data_in => memorize_data_in ,
162     en_mem_rotor => memorizeRotor_data ,
163     en_mem_refl => memorizeReflector_data ,
164     src => src ,
165     sideOfRotor => side_of_rotor
166 );
167 data_out <= normalizedLetter ;
```

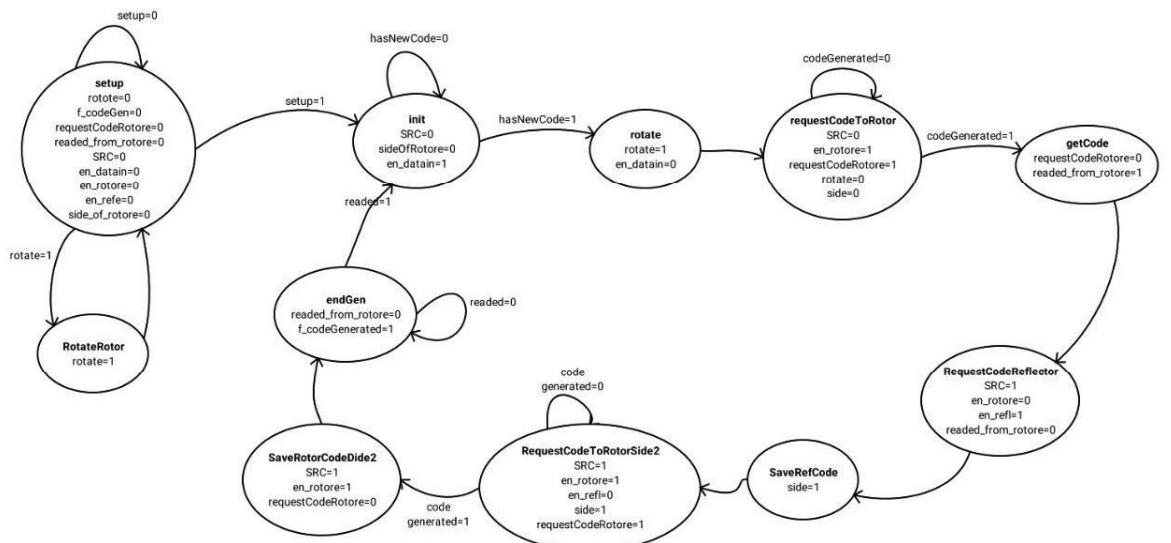
168

169 end Structural;

Abbiamo due fasi che attraversa la macchina:

- **Fase di andata:** In questa fase il codice di ingresso (la lettera in chiaro) impatta sulla prima faccia del rotore, e quest'ultimo genera un codice che a sua volta va ad impattare sul Riflettore
- **Fase di ritorno:** In questa fase il riflettore "riflette" il segnale alla seconda faccita del rotore, il quale poi genera un altro codice che sarà poi quello finale.

Ovviamente tali fasi sono scandite dalla rete di controllo che avrà il seguente automa:



Anche qui forniamo una breve descrizione degli stati:

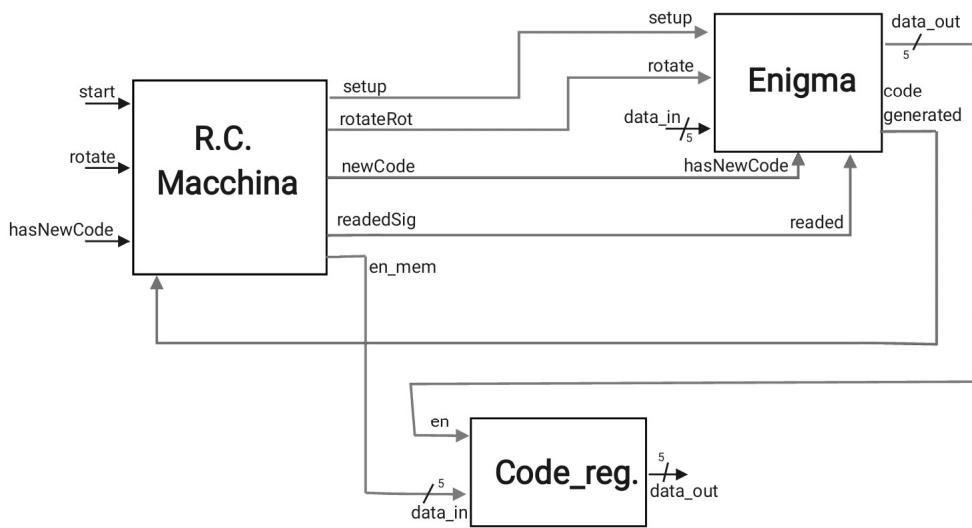
- **setup:** Stato di setup della macchina che mi permette di cambiare la configurazione iniziale del rotore, applicando delle ro-

tazioni

- **RotateRotor:** Questo stato da l'impulso al rotore di ruotare, cambiando la configurazione iniziale del rotore, inizialmente la configurazione è in posizione "A".
- **init:** Stato raggiungibile dopo la fase di setup che permette di avviare la macchina vera e propria.
In tale stato aspetto che ci sia un nuovo codice da codificare, andrebbe a simulare una pressione della tastiera.
- **rotate:** Dopo la pressione il rotore ruota passando alla prossima posizione.
- **requestCodeToRotor:** Questo stato permette di chiedere al rotore il codice di cifratura, è il primo passo della fase di andata.
- **getCode:** Una volta che il rotore genera il codice, tale codice deve essere salvato, questo stato permette tale salvataggio.
- **RequestCodeReflector:** Dopo il salvataggio del codice restituito dal rotore, questo codice impatta sul riflettore, il quale è una macchina combinatoria che restituirà subito il nuovo codice.
- **SaveRefCode:** Salviamo il codice restituito dal riflettore.
- **requestCodeToRotorSide2:** Passiamo alla fase di ritorno, dove il codice torna al rotore sulla seconda facciata, anche qui aspettiamo che il rotore ci risponda.

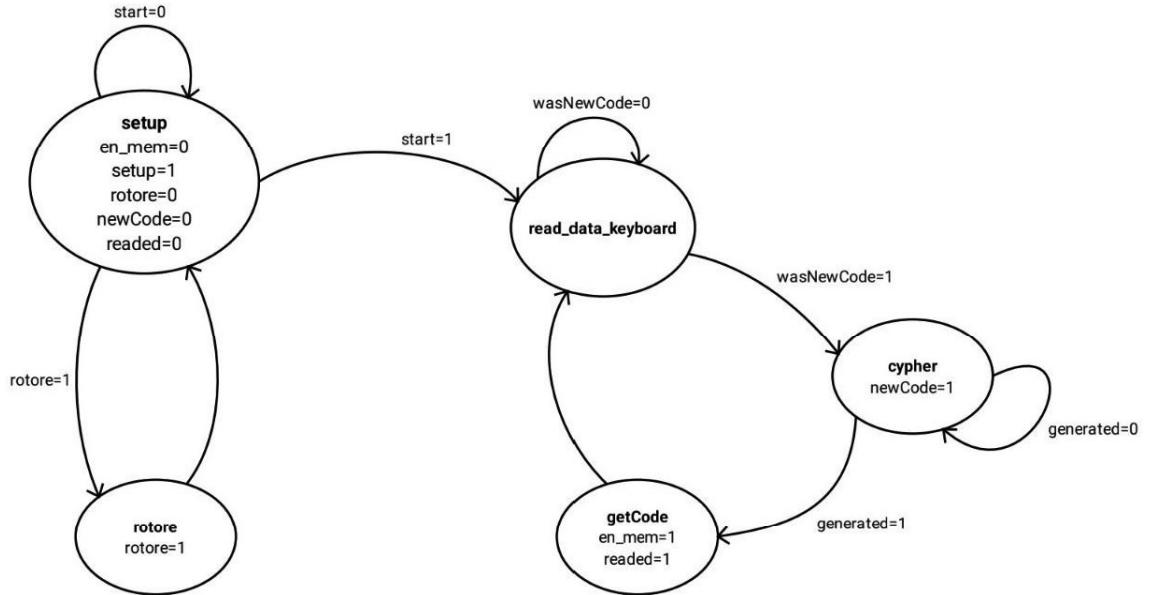
- **SaveRotorCodeSide2:** Quando il rotore genera il codice lo salviamo.
- **endGen:** Abbiamo generato il codice finale e quindi avvisiamo il sistema richiedente al livello 3 che abbiamo generato il codice. In questo stato aspettiamo anche che il sistema richiedente legga il codice.

Da progetto è stato deciso anche di aggiungere un terzo livello di progetto della macchina che è rappresentato da una architettura molto semplice:



La rete di controllo è usata solo per interfacciarsi con il progettoOn-board che usa la macchina crittografica.

Dunque la rete di controllo a questo livello sarà rappresentata dal seguente automa:

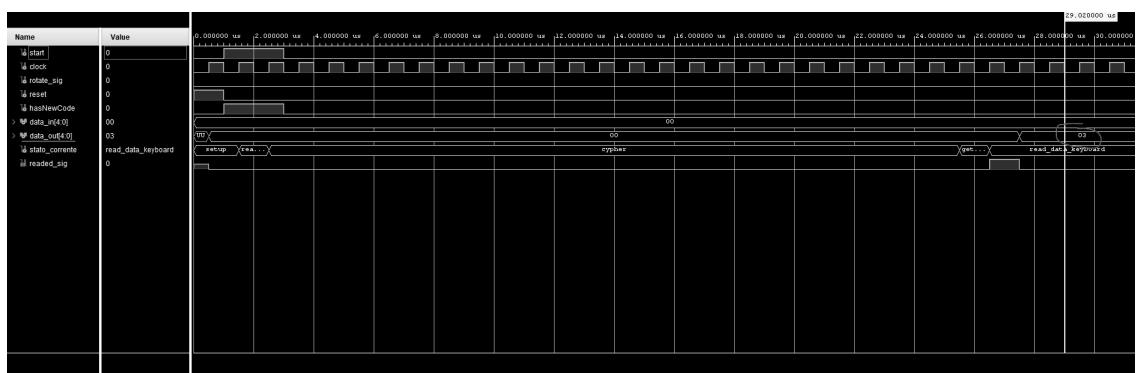


- **setup:** Anche qui ritroviamo la fase di setup della macchina che permette di ruotare il rotore.
- **rotate:** Fase che mi da l'impulso di rotazione del rotore.
- **read_data_keyboard:** Fase post-setup che mi permette di leggere il codice da tastiera, aspetto finchè non viene dato l'impulso che sia un nuovo codice da cifrare (Non deve essere per forza un'altra lettera).
- **cypher:** Fase di cifratura, dove aspettiamo che la macchina generi il codice di cifratura.
- **getCode:** Fase dove salviamo il codice generato ed avvisiamo la macchina che abbiamo letto.

Di seguito riportiamo una simulazione del funzionamento della macchina a questo progetto.

NB: Chiaramente la macchina funziona con dei codici e non delle lettere vere e proprie, dunque l'ingresso e l'uscita corrisponde alla posizione nell'alfabeto della lettera.

Come input alla macchina è stato dato il codice "00000" corrispondente alla lettera "A". L'output calcolato, in posizione "A" del rotore, è la lettera D codificata come "00011"



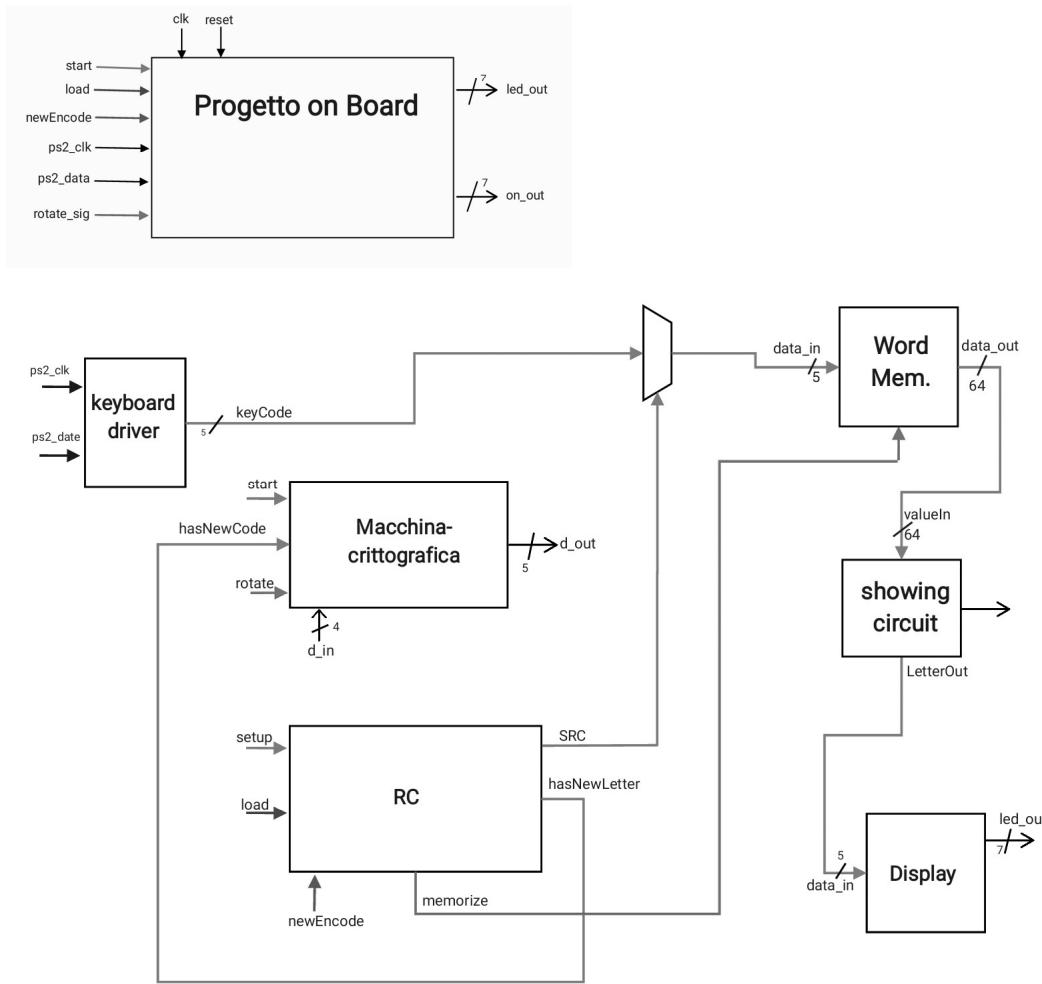
Come si può constatare, la macchina restituisce l'output atteso.

8.3.5 Sintesi su scheda

L'intera macchina è stata progettata in modo tale da essere più sintetizzabile possibile.

Per la sintesi è stato previsto un altro livello di progetto fatto da componenti di interfacciamento con le periferiche, ovvero tastiera e i display a 7 segmenti.

L'architettura è la seguente:



Il keyboardDriver ha al suo interno i componenti che implementano il protocollo PS/2 e un Encoder che codifica i codici a 8 bit facendoli diventare a 5 bit.

I componenti Display e ShowingCircuit implementano lo scanning circuit che è stato opportunamente modificato per funzionare per i nostri scopi.

Un componente nuovo per visualizzare opportunamente sugli 8 display è la word mem che è una memoria contenente le lettere della parola da mostrare.

In vhdl è stata implementata in modo comportamentale, di seguito il codice:

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4 use ieee.math_real.all;
5
6 entity word_mem is
7 Port (
8     clock: in std_logic;
9     reset: in std_logic;
10    memorize: in std_logic;
11    data_in: in std_logic_vector(4 downto 0);
12    data_out: out std_logic_vector(63 downto 0)
13 );
14 end word_mem;
15
16 architecture Behavioral of word_mem is
17 type memoria is array(0 to 7) of std_logic_vector
18 (7 downto 0);
19 signal addr: integer range 0 to 7 := 0;
20 signal memoria_interna: memoria := ((others => (
21         others => '1')));
```

```

20 signal data_out_tmp: std_logic_vector(63 downto 0)
21 ;
22 begin
23 mem: process(clock, reset, data_in, memorize)
24 begin
25 if(rising_edge(clock)) then
26   if(reset = '1') then
27     memoria_interna <= ((others => (others =>
28       '1'))));
29   addr <= 0;
30   elsif(memorize = '1') then
31     memoria_interna(addr) <= "000"&data_in;
32     addr <= addr + 1;
33     if(addr >= 7) then
34       addr <= 0;
35     end if;
36   else
37     memoria_interna(addr) <= "000"&data_in;
38   end if;
39   data_out_tmp <= memoria_interna(0) &
40   memoria_interna(1) & memoria_interna(2) &
41   memoria_interna(3) & memoria_interna(4) &

```

```

memoria_interna(5) & memoria_interna(6) &
memoria_interna(7);

39 end if;

40 end process;

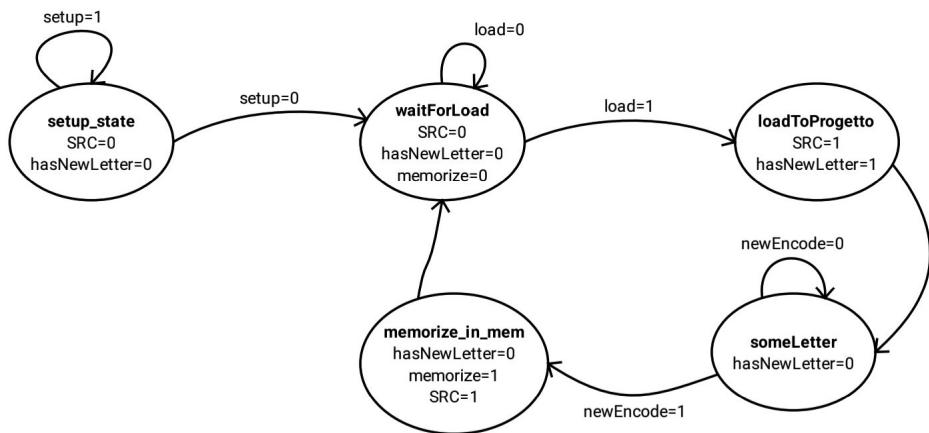
41 data_out <= data_out_tmp;

42 end Behavioral;

```

Anche a questo livello abbiamo una rete di controllo che scandisce le fasi del nostro sistema, ovvero quella di setup e di lavoro.

L'automa sarà il seguente:



- **setup_state:** Stato di setup del sistema che ci permette di dare alla macchina i segnali di rotazione
- **waitForLoad:** Stiamo nella fase di lavoro, aspettiamo che l'utente ci dia l'ok per codificare la lettera
- **LoadToProgetto:** Questo stato carica nella macchina crit-

tografica la lettera scelta.

- **SameLetter:** Aspetto che l'utente mi dia il segnale di nuova crittazione
- **memorize_in_mem:** Stato che mi permette di memorizzare nella word_mem la lettera codificata.

Appurato che la macchina funziona e anche tutte le componenti, il progetto è stato sintetizzato sulla board.

Di seguito riportiamo la configurazione nel file dei constraint:

```
1 ### Clock signal
2 set_property -dict { PACKAGE_PIN E3      IOSTANDARD
   LVCMOS33 } [get_ports { clock }]; #
   IO_L12P_T1_MRCC_35 Sch=clk100mhz
3 create_clock -add -name sys_clk_pin -period 10.00
   -waveform {0 5} [get_ports {clock }];
4 [...]
5 ###7 segment display
6 set_property -dict { PACKAGE_PIN T10      IOSTANDARD
   LVCMOS33 } [get_ports { led_out[0] }]; #
   IO_L24N_T3_A00_D16_14 Sch=ca
7 set_property -dict { PACKAGE_PIN R10      IOSTANDARD
   LVCMOS33 } [get_ports { led_out[1] }]; #IO_25_14
   Sch=cb
```

```

8 set_property -dict { PACKAGE_PIN K16      IOSTANDARD
                      LVCMOS33 } [get_ports { led_out[2] }]; #IO_25_15
                      Sch=cc

9 set_property -dict { PACKAGE_PIN K13      IOSTANDARD
                      LVCMOS33 } [get_ports { led_out[3] }]; #
                      IO_L17P_T2_A26_15 Sch=cd

10 set_property -dict { PACKAGE_PIN P15     IOSTANDARD
                      LVCMOS33 } [get_ports { led_out[4] }]; #
                      IO_L13P_T2_MRCC_14 Sch=ce

11 set_property -dict { PACKAGE_PIN T11     IOSTANDARD
                      LVCMOS33 } [get_ports { led_out[5] }]; #
                      IO_L19P_T3_A10_D26_14 Sch=cf

12 set_property -dict { PACKAGE_PIN L18     IOSTANDARD
                      LVCMOS33 } [get_ports { led_out[6] }]; #
                      IO_L4P_T0_D04_14 Sch=cg

13 set_property -dict { PACKAGE_PIN H15     IOSTANDARD
                      LVCMOS33 } [get_ports { led_out[7] }]; #
                      IO_L19N_T3_A21_VREF_15 Sch=dp

14 set_property -dict { PACKAGE_PIN J17     IOSTANDARD
                      LVCMOS33 } [get_ports { anodes_out[0] }]; #
                      IO_L23P_T3_FOE_B_15 Sch=an[0]

15 set_property -dict { PACKAGE_PIN J18     IOSTANDARD
                      LVCMOS33 } [get_ports { anodes_out[1] }]; #

```

```

IO_L23N_T3_FWE_B_15 Sch=an [1]

16 set_property -dict { PACKAGE_PIN T9      IOSTANDARD
                      LVCMOS33 } [get_ports { anodes_out[2] }]; #

IO_L24P_T3_A01_D17_14 Sch=an [2]

17 set_property -dict { PACKAGE_PIN J14      IOSTANDARD
                      LVCMOS33 } [get_ports { anodes_out[3] }]; #

IO_L19P_T3_A22_15 Sch=an [3]

18 set_property -dict { PACKAGE_PIN P14      IOSTANDARD
                      LVCMOS33 } [get_ports { anodes_out[4] }]; #

IO_L8N_T1_D12_14 Sch=an [4]

19 set_property -dict { PACKAGE_PIN T14      IOSTANDARD
                      LVCMOS33 } [get_ports { anodes_out[5] }]; #

IO_L14P_T2_SRCC_14 Sch=an [5]

20 set_property -dict { PACKAGE_PIN K2      IOSTANDARD
                      LVCMOS33 } [get_ports { anodes_out[6] }]; #

IO_L23P_T3_35 Sch=an [6]

21 set_property -dict { PACKAGE_PIN U13      IOSTANDARD
                      LVCMOS33 } [get_ports { anodes_out[7] }]; #

IO_L23N_T3_A02_D18_14 Sch=an [7]

22 [ . . . ]

23 ##Buttons

24 #set_property -dict { PACKAGE_PIN C12      IOSTANDARD
                      LVCMOS33 } [get_ports { CPU_RESETN }]; #

```

```

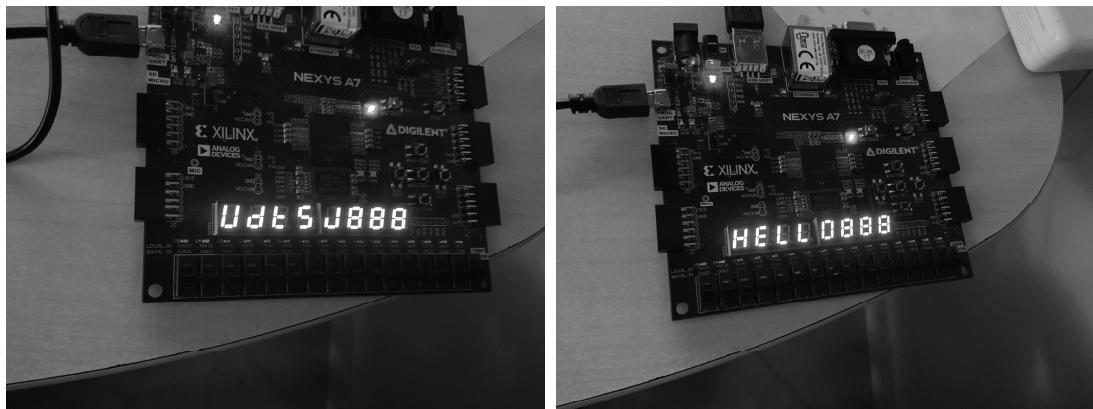
IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn
25 set_property -dict { PACKAGE_PIN N17 IOSTANDARD
LVCMOS33 } [get_ports { start }]; #
IO_L9P_T1_DQS_14 Sch=btnc
26 set_property -dict { PACKAGE_PIN M18 IOSTANDARD
LVCMOS33 } [get_ports { reset }]; #
IO_L4N_T0_D05_14 Sch=btnu
27 set_property -dict { PACKAGE_PIN P17 IOSTANDARD
LVCMOS33 } [get_ports { rotate_sig }]; #
IO_L12P_T1_MRCC_14 Sch=btndl
28 set_property -dict { PACKAGE_PIN M17 IOSTANDARD
LVCMOS33 } [get_ports { load }]; #
IO_L10N_T1_D15_14 Sch=bt(nr)
29 set_property -dict { PACKAGE_PIN P18 IOSTANDARD
LVCMOS33 } [get_ports { newEncode }]; #
IO_L9N_T1_DQS_D13_14 Sch=bt(nd)
30 [ . . . ]
31 ##/USB HID (PS/2)
32 set_property -dict { PACKAGE_PIN F4 IOSTANDARD
LVCMOS33 } [get_ports { ps2_clk }]; #
IO_L13P_T2_MRCC_35 Sch=ps2_clk
33 set_property -dict { PACKAGE_PIN B2 IOSTANDARD
LVCMOS33 } [get_ports { ps2_data }]; #

```

34 IO_L10N_T1_AD15N_35 Sch=ps2_data
 [. . .]

Per la simulazione post-sintesi è stato deciso di codificare la parola "HELLO" che per la codifica della macchina, restituisce la parola "UDTSJ" con la configurazione dei rotori ad A.

Dopodichè abbiamo decifrato la parola cirfrata riottenendo la parola "HELLO".



8.4 Future aggiunte

Rispetto agli altri progetti che provano ad imitare enigma online, questo è l'unico che cerca di imitarla meccanicamente e non attraverso un algoritmo, infatti sono state progettate le principali componenti di enigma a livello di blocchi elementari rendendo il lavoro del sintetizzatore più semplice.

Tra le futura aggiunte c'è sicuramente l'idea di aggiungere il sistema rotori che implementa 3 rotori almeno ed una rete che li gestisce in

modo che restituiscano il giusto codice.

Un'altra aggiunta sarebbe aggiustare il meccanismo di codifica modificando il PS/2 per far riconoscere quando la tastiera è idle e quando poi viene effettuata una pressione.