Implementazione Algoritmo DES su FPGA

Stefano Zenaro

Marzo 2022

Indice

1 Introduzione

Questo documento descrive come implementare l'algoritmo DES nel linguaggio di descrizione dell'hardware Verilog, come simulare i moduli principali e infine come programmare la Programmable Logic (FPGA) di Xilinx PYNQ-Z1 (xc7z020clg400-1) e come comandarla tramite il linguaggio di programmazione Python.

A fine documento saranno elencate le scelte progettuali.

Per brevità il seguente documento utilizzerà in modo intercambiabile il percorso dei file sorgente Verilog e il nome dei moduli definiti al loro interno: il nome dei moduli corrisponde al nome del file (senza estensione).

2 Algoritmo DES

L'algoritmo DES e' un algoritmo che, data una chiave da 64 bit (in questo caso implementata in hardware), si occupa di cifrare una sequenza di 64 bit restituendo in output altri 64 bit da cui risulta molto difficile risalire alla sequenza originale.

L'algoritmo DES consiste nel:

- Fare una permutazione iniziale (gestita dal modulo $src/IP.v$) dell'input. Con la permutazione si ottiene una nuova sequenza di 64 bit formata dagli stessi bit del messaggio da cifrare ma in un ordine diverso.
- Eseguire 16 round in cui si prende l'output del round precedente (o della permutazione iniziale) e si suddivide in due parti uguali da 32 bit (L_{n-1} e R_{n-1}).

n e' il numero del round attuale, $n-1$ e' il numero del round precedente (se $n-1=0$ significa che l'input proviene dal risultato ottenuto dalla permutazione iniziale)

Anche l'output di ogni round e' formato da due parti da 32 bit:

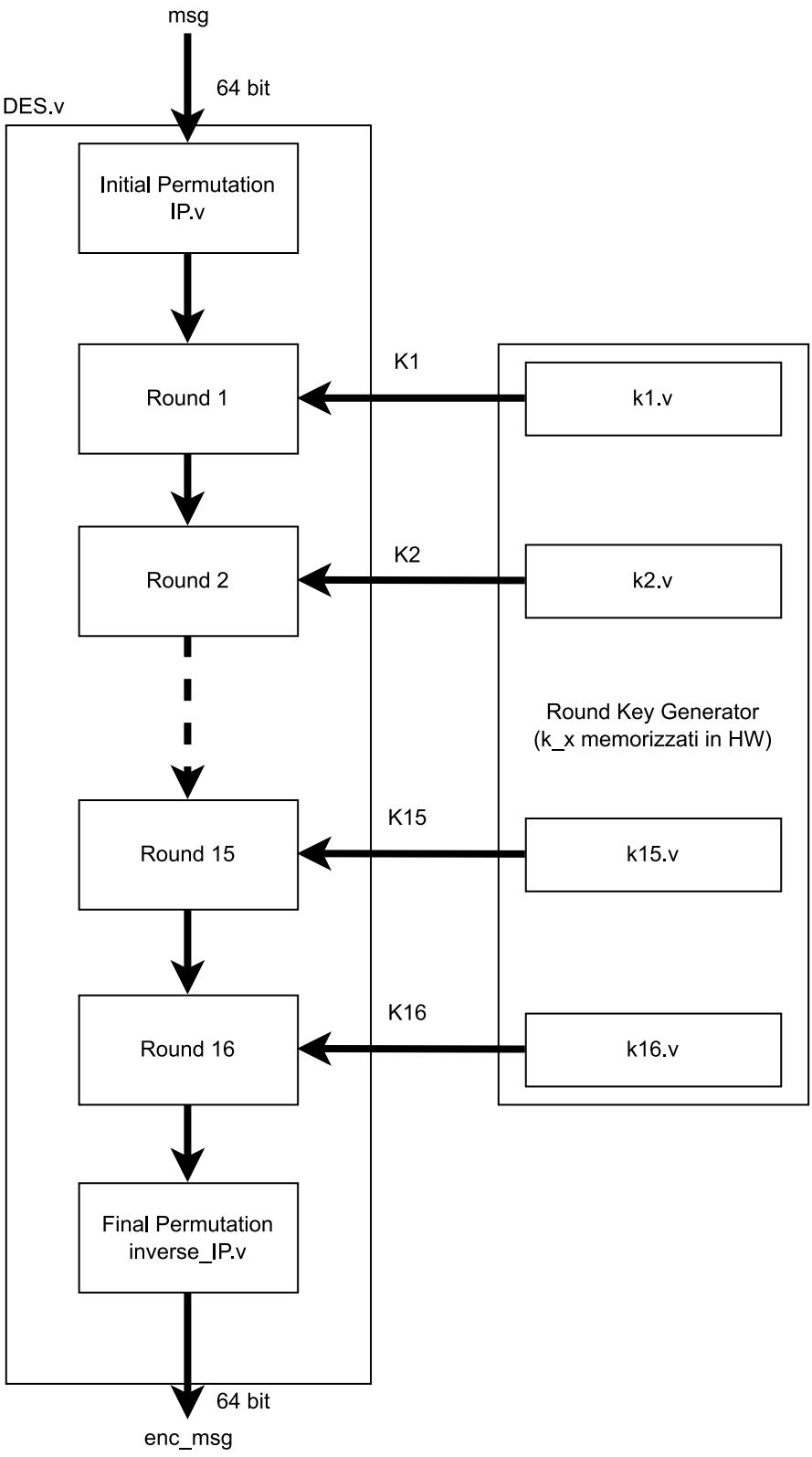
- La parte di sinistra e' uguale alla parte di destra del round precedente:

$$L_n = R_{n-1}$$

- La parte di destra e' formata eseguendo la funzione logica XOR tra la parte di sinistra del round precedente e il risultato di un calcolo effettuato in funzione della chiave e della parte di destra del round precedente.

$$R_n = L_{n-1} \oplus f(R_{n-1}, K_n)$$

- Fare una permutazione finale (gestita dal modulo $src/inverse\_IP.v$) dell'output del round 16. L'output della permutazione finale e' il messaggio cifrato.


msg
64 bit
DES.v
Initial Permutation
IP.v
Round 1
Round 2
Round 15
Round 16
Final Permutation
inverse_IP.v
64 bit
enc_msg
K1
K2
K15
K16
k1.v
k2.v
k15.v
k16.v
Round Key Generator
(k_x memorizzati in HW)

Per calcolare l'output di un round si utilizzano istanze dei seguenti moduli:

- $src/f.v$ che, presi in ingresso R_{n-1} e K_n, restituisce il risultato di $f(R_{n-1}, K_n)$
- $src/K\_selector.v$ che seleziona e restituisce K_n (output del modulo $src/kn.v$, dove n e' il numero del round corrente), valore che dipende da una computazione effettuata sulla chiave.

Siccome, in questo progetto, la chiave viene memorizzata in hardware e la sua computazione restituisce sempre gli stessi K_n il compito di computare i K_n e di creare i moduli Verilog $src/kn.v$ viene lasciato ad uno script scritto in Python ($utility/keys/gen\_kn.py$)

Il modulo "f" (ovvero il modulo che implementa la funzione generica $f(R_{n-1}, K_n)$) definito nel file $src/f.v$ si comporta in questo modo:

- Prende i 32 bit in ingresso di R_{n-1} e li estende duplicando alcuni bit e permutandoli attraverso il modulo $src/E.v$:
  $R\_E = E(R_{n-1})$
- L'output del modulo "E" viene messo in XOR con K_n:
  $R\_E\_XOR\_k = R\_E \oplus K_n = E(R_{n-1}) \oplus K_n$
- I 48 bit ottenuti dallo XOR vengono suddivisi in 8 blocchi da 6 bit. Ogni blocco da 6 bit entra in un modulo S_x (per $x = 1..8$).
  $B_1B_2B_3B_4B_5B_6B_7B_8 = E(R_{n-1}) \oplus K_n$

Queste funzioni S_x vengono nominate "S-Boxes". Ogni S Box viene definita nel suo modulo $src/Sx.v$ (per $x = 1..8$)

Il primo blocco di 6 bit entra in S_1, il secondo blocco entra in S_2, ...

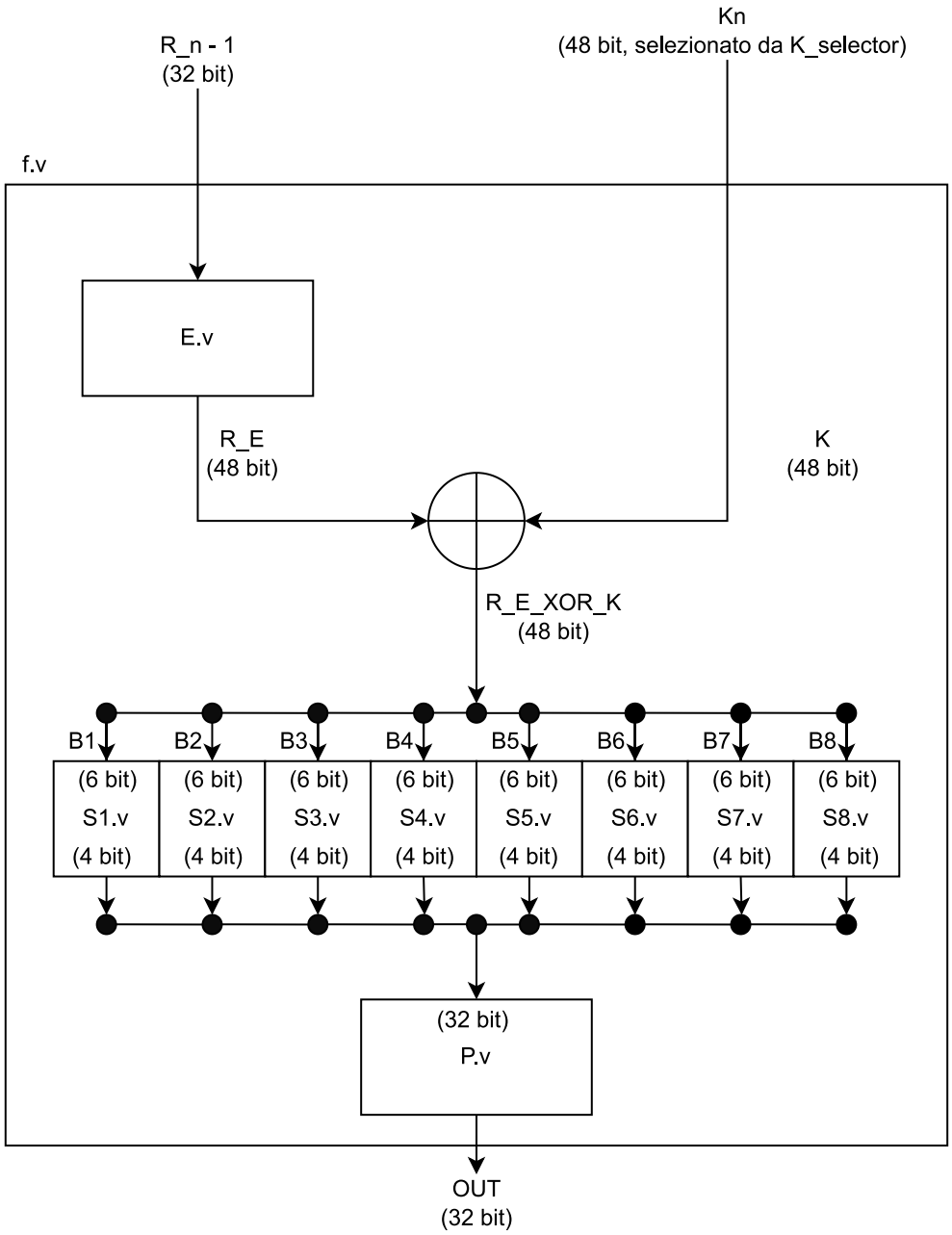
L'output di ogni blocco e' da 4 bit, quindi l'output complessivo degli otto blocchi e' da 32 bit.
$Sboxes(E(R_{n-1}) \oplus K_n) = S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$

- I 32 bit in output delle S-Boxes vengono permutati dal modulo $src/P.v$. L'output del modulo "P" e' l'output della funzione $f(R_{n-1}, K_n)$

In termini matematici:

$$f(R_{n-1}, K_n) = P(Sboxes(E(R_{n-1}) \oplus K_n))$$

Dove la funzione $Sboxes()$ e' una astrazione di ciò che avviene in ogni modulo S_x (per $x = 1..8$).



I Moduli K_n, le S-Boxes e i moduli che si occupano di permutare segnali ($IP, inverse\_IP, E, P$) sono stati generati mediante script Python, che risiedono nella cartella *utility*, per rimuovere la possibilità di errore umano durante la realizzazione di quei moduli.

3 Implementazione Verilog dell'Algoritmo DES

A causa delle limitazioni di bit di input/output della FPGA la macchina a stati e' stata realizzata per ricevere in ingresso il messaggio da cifrare in due stati differenti (vengono letti 32 bit alla volta).

Questo permette di impiegare 32 bit di messaggio e 2 di controllo anziché utilizzare 64 bit di messaggio.

La macchina a stati, implementata nel file $src/FSMD.v$ (poiché implementa FSM + Datapath), esegue queste operazioni:

- Resetta i registri interni quando entra in ingresso il segnale di reset rst impostato a 0.

I registri resettati servono per memorizzare il messaggio da cifrare, gli output della FSMD (memorizzazione effettuata della prima parte del messaggio, fine) e dei moduli intermedi.

- Attende che in ingresso sia presente la prima parte del messaggio da cifrare. La FSM riconosce che il messaggio e' pronto quando $ready\_part1$ e' posto a 1.
- La FSM manda in uscita il segnale $read\_part1$ indicando che la prima parte del messaggio da cifrare e' stata memorizzata.

  Dopo aver fatto questo attende che in ingresso sia presente la seconda parte del messaggio da cifrare. La FSM riconosce che il messaggio e' pronto quando $ready\_part2$ e' posto a 1.
- Effettua la permutazione iniziale sull'intero messaggio in input (ottenuto concatenando le due parti)

La permutazione viene effettuata dal modulo $src/IP.v$

- Prepara l'output della permutazione per il primo round suddividendolo in due meta' (L_{n-1} e R_{n-1})
- Esegue i 16 round di crittografia. Per ogni round:
  - Seleziona la K_n del round n (round attuale)

Viene utilizzato il modulo $src/K\_selector.v$

- Calcola L_n ed R_n:

$$L_n = R_{n-1}$$
$$R_n = L_{n-1} \hat{} f(R_{n-1}, K_n)$$

$f(R_{n-1}, K_n)$ viene calcolato dal modulo $src/f.v$

- Passa al round successivo incrementando n e memorizzando L in L_{n-1} e R in R_{n-1}

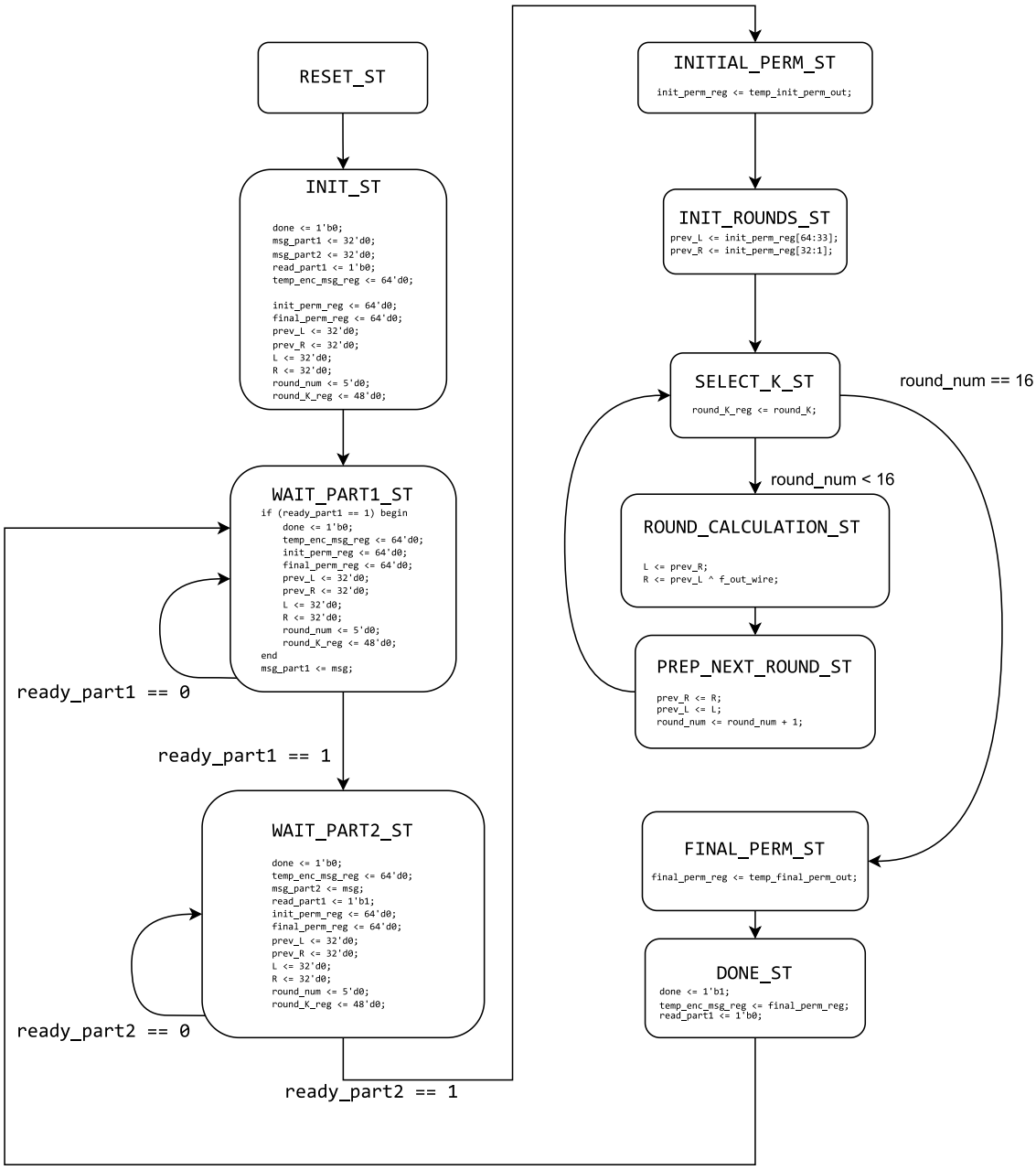
- Esegue la permutazione finale sull'output del round 16. (Dove l'output e' dato dalla concatenazione $R_{16} + L_{16}$, e non $L_x + R_x$ come per i round precedenti)

La permutazione viene effettuata dal modulo $src/inverse\_IP.v$

- Manda in uscita il messaggio cifrato, output della permutazione finale, sul segnale enc_msg e imposta il segnale $done$ a 1 per indicare che l'operazione e' conclusa.

Il segnale $read\_part1$ verrà resettato per poter riconoscere quando avverrà la memorizzazione della prossima prima parte del messaggio.

- A questo punto la FSM torna in attesa che in ingresso sia presente la prima parte del messaggio.


RESET_ST
INIT_ST
done <= 1'b0;
msg_part1 <= 32'd0;
msg_part2 <= 32'd0;
read_part1 <= 1'b0;
temp_enc_msg_reg <= 64'd0;
init_perm_reg <= 64'd0;
final_perm_reg <= 64'd0;
prev_L <= 32'd0;
prev_R <= 32'd0;
L <= 32'd0;
R <= 32'd0;
round_num <= 5'd0;
round_K_reg <= 48'd0;
WAIT_PART1_ST
if (ready_part1 == 1) begin
done <= 1'b0;
temp_enc_msg_reg <= 64'd0;
init_perm_reg <= 64'd0;
final_perm_reg <= 64'd0;
prev_L <= 32'd0;
prev_R <= 32'd0;
L <= 32'd0;
R <= 32'd0;
round_num <= 5'd0;
round_K_reg <= 48'd0;
end
msg_part1 <= msg;
ready_part1 == 0
ready_part1 == 1
WAIT_PART2_ST
done <= 1'b0;
temp_enc_msg_reg <= 64'd0;
msg_part2 <= msg;
read_part1 <= 1'b1;
init_perm_reg <= 64'd0;
final_perm_reg <= 64'd0;
prev_L <= 32'd0;
prev_R <= 32'd0;
L <= 32'd0;
R <= 32'd0;
round_num <= 5'd0;
round_K_reg <= 48'd0;
ready_part2 == 0
ready_part2 == 1
INITIAL_PERM_ST
init_perm_reg <= temp_init_perm_out;
INIT_ROUNDS_ST
prev_L <= init_perm_reg[64:33];
prev_R <= init_perm_reg[32:1];
SELECT_K_ST
round_K_reg <= round_K;
round_num == 16
round_num < 16
ROUND_CALCULATION_ST
L <= prev_R;
R <= prev_L ^ f_out_wire;
PREP_NEXT_ROUND_ST
prev_R <= R;
prev_L <= L;
round_num <= round_num + 1;
FINAL_PERM_ST
final_perm_reg <= temp_final_perm_out;
DONE_ST
done <= 1'b1;
temp_enc_msg_reg <= final_perm_reg;
read_part1 <= 1'b0;

4 Simulazione e test dei moduli Verilog

Per simulare i moduli Verilog si e' deciso di optare per il simulatore Icarus Verilog e per visualizzare le forme d'onda esportate in formato VCD GTKwave.

Per i test e' stato utilizzato il framework cocotb per il linguaggio di programmazione Python. Per lanciare i test con cocotb occorre eseguire il $Makefile$ nella cartella *tests* con il comando *make*.

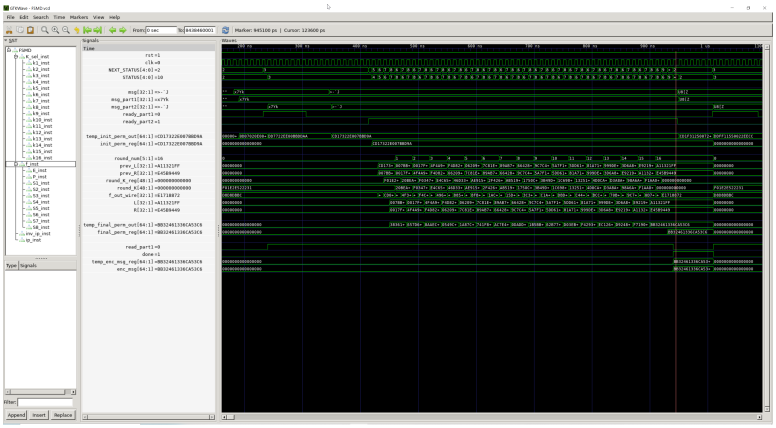
Se si desidera visualizzare le forme d'onda create dai test e' necessario eseguire singolarmente il test della FSMD (con il comando $make\ test\_FSMD$) e poi aprire il file VCD con GTKwave con il comando "$gtkwave < nomefile > .vcd$"

Inoltre nella cartella *tests* sono presenti delle semplici test bench in formato Verilog.

Per eseguirle occorre compilare i moduli e poi eseguire la simulazione:

```
iverilog -c FSMD_modules.txt
vvp a.out
```

A questo punto e' possibile visualizzare le forme d'onda aprendo il file VCD appena creato con GTKwave.



File VCD generato dal test del modulo FSMD.v

5 Programmare la scheda Xilinx PYNQ-Z1

Per programmare la scheda e' necessario utilizzare il software Xilinx Vivado per:

- Creare una Intellectual Property (IP) di tipo periferica AXI (Lite) e istanziarci all'interno il modulo FSMD.

Per controllare input e output della FSMD sono richiesti 7 registri

- Creare il Block Design per interfacciare la IP con la APU della scheda.

Vivado permette di collegare automaticamente le IP cliccando su "Run Block Automation" e "Run Connection Automation"

- Creare un wrapper HDL: questo wrapper al suo interno gestirà APU, IP (con all'interno la FSMD) e il modulo che si occupa di resettare la board.
- Generare il Bitstream (*system.bit*)
- Esportare il Bitstream e il Block Design (*system.tcl*)

Attraverso uno script Python e' possibile utilizzare il Bistream e il Block Design per programmare e controllare la FPGA.
I registri che verranno utilizzati dall'interfaccia AXI sono i seguenti:

Indirizzo	Nome registro	Nome segnale
0x00	slv_reg0	msg
0x04	slv_reg1	ready_part1
0x08	slv_reg2	ready_part2
0x0C	slv_reg3	read_part1
0x10	slv_reg4	done
0x14	slv_reg5	enc_msg_part1
0x18	slv_reg6	enc_msg_part2

La interfaccia della periferica AXI generata da Vivado deve essere modificata nei seguenti punti:

- Dopo la definizione dell'interfaccia del modulo bisogna dichiarare i wire per gestire gli output della FSMD.

```
wire read_part1;
wire done;
wire [31:0] enc_msg_part1;
wire [31:0] enc_msg_part2;
```

- Nel processo principale (*always @(posedge S_AXI_ACLK)*) bisogna far propagare i segnali di output.

  Modificare le seguenti righe:

```
else begin
if (slv_reg_wren)
```

  E aggiungere la propagazione:

```
else begin
    slv_reg3 <= 32'b0000_0000 + read_part1;
    slv_reg4 <= 32'b0000_0000 + done;
    slv_reg5 <= enc_msg_part1;
    slv_reg6 <= enc_msg_part2;
    if (slv_reg_wren)
```

- Sempre nel processo principale nel ramo default bisogna commentare la propagazione degli output:

```
slv_reg0 <= slv_reg0;
slv_reg1 <= slv_reg1;
slv_reg2 <= slv_reg2;
//  slv_reg3 <= slv_reg3;   <--- Comment these lines.
//  slv_reg4 <= slv_reg4; <--- Comment these lines.
//  slv_reg5 <= slv_reg5; <--- Comment these lines.
//  slv_reg6 <= slv_reg6; <--- Comment these lines.
```

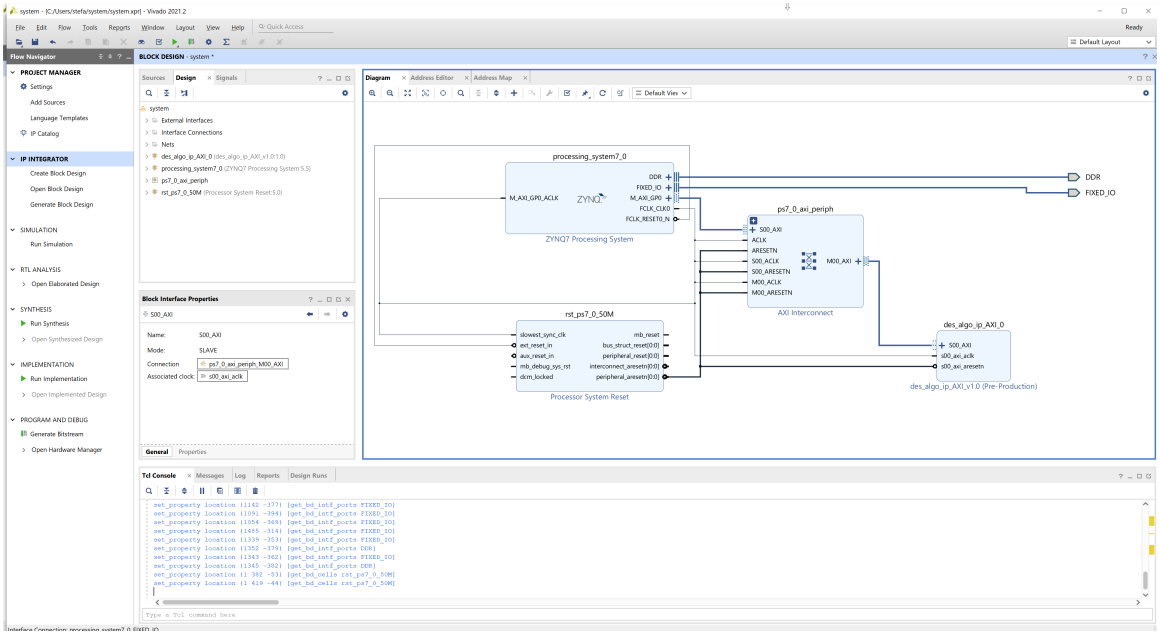
- Istanziare la FSMD nell'interfaccia dopo il commento "add user logic here":

```
FSMD fsmd(
    .clk(S_AXI_ACLK),
    .rst(S_AXI_ARESETN),

    .msg(slv_reg0),
    .ready_part1(slv_reg1[0]),
    .ready_part2(slv_reg2[0]),

    .read_part1(read_part1),
    .done(done),
    .enc_msg({enc_msg_part1, enc_msg_part2})
);
```

Il Block Design ottenuto integrando la nuova IP (con la FSMD) con la APU dovrebbe essere il seguente:



6 Controllare FPGA da Python

Per controllare la FPGA da Python e' possibile trovare lo script *des.py* nella cartella *dist* nel repository.

> Gli altri script Python presenti nella cartella *dist* sono file di test.

Prima di utilizzare lo script occorre generare i file *system.bit* e *system.tcl* con Xilinx Vivado: servono per programmare la Programmable Logic (FPGA) della board.

Lo script Python *des.py*:

- Riceve come argomento da linea di comando un messaggio da cifrare.
- Suddivide il messaggio in blocchi da 64 bit poiché l'algoritmo DES permette di cifrare 64 bit di dati.

> Se l'ultimo blocco, quello più a destra, e' composto da meno di 64 bit questo verrà esteso con NULL ($0x00$)

- Programma la FPGA con il file Bitstream *system.bit* (questo necessità del file contenente il block design: *system.tcl*)

- Ogni blocco viene predisposto per essere adatto all'interfaccia del modulo FSMD.v e viene mandato in input alla FPGA.

  Dopo aver ricevuto il blocco cifrato dalla FPGA questi vengono memorizzati in una lista finché la FPGA non ha terminato di cifrare tutti i blocchi.

- A fine cifratura i blocchi cifrati vengono mostrati a video all'utente sia in formato binario sia in formato esadecimale.

Per utilizzare lo script occorre:

- Copiare con il comando scp i file sulla board:

```
scp system.* <indirizzoIP_board>:~/
scp des.py <indirizzoIP_board>:~/
```

- Entrare in SSH sulla board (potrebbe essere necessario inserire la password di login):

```
ssh <indirizzoIP_board>
```

- Eseguire lo script Python passando come argomento il messaggio da cifrare:

```
sudo python3.6 des.py '<messaggio>'
```

E' necessario eseguire lo script Python con sudo per poter aver l'accesso alla programmazione e al controllo dell'hardware della FPGA.

Se si utilizzano le virgolette (") anziché gli apici (') occorre fare attenzione a caratteri interpretabili dalla shell (come ad esempio il dollaro) e eventualmente farne l'escape con la slash \

7 Scelte progettuali

7.1 Input suddiviso in due stati differenti

A causa delle limitazioni di bit di input/output della FPGA la macchina a stati e' stata realizzata per ricevere in ingresso il messaggio da cifrare in due stati differenti (vengono letti 32 bit alla volta).

Questo permette di impiegare 32 bit di messaggio e 2 di controllo anziché utilizzare 64 bit di messaggio.

7.2 Moduli K_n

I segnali K vengono restituiti da moduli differenti per permettere facilmente la modifica della chiave memorizzata in hardware generando con lo script python *utility/keys/gen_kn.py* i file sorgente dei moduli Verilog.

Dopo aver passato una chiave allo script tramite parametro questo genererà i moduli Verilog.

Dopo averli generati e' sufficiente sovrascrivere i file sorgente presenti nella cartella src e riprogrammare l'FPGA per attuare la modifica della chiave.

7.3 Generazione moduli da utility

Come per i Moduli K_n, le S-Boxes e i moduli che si occupano di permutare segnali ($IP, inverse\_IP, E, P$) sono stati generati mediante script Python che risiedono nella cartella *utility* per rimuovere la possibilità di errore umano durante la realizzazione di quei moduli.

Tutte le permutazioni vengono descritte nelle documentazioni dell'algoritmo DES come matrici che devono essere lette da sinistra verso destra, dall'alto verso il basso: il primo bit in output corrisponde al bit numero x presente nella prima posizione della matrice, il secondo bit in output sarà il bit in posizione 2, ...

Esempio con la permutazione iniziale:

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

Il primo bit in output sarà il bit in posizione 58 in input, il secondo bit in output sarà il bit in posizione 50, ...

Gli indici degli input e degli output delle permutazioni partono da uno.

E' possibile trovare i file delle matrici delle permutazioni (memorizzate come una lista di valori), usati dallo script Python *utility/permutations/gen_permutations.py* per generare i moduli, nella cartella *utility/permutations*

Anche le S-Boxes vengono descritte come matrici ma in quel caso bisogna prendere la sequenza di 6 bit in input e leggere il primo e l'ultimo bit come l'indice della riga e i bit centrali come indice della colonna. L'elemento selezionato sarà l'output della S-Box.

Gli indici delle righe e delle colonne delle S-Boxes partono da zero.

Esempio con la prima S-Box:

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Se l'input fosse 0 = 000000, l'output si trova in posizione 00 = 0 come riga e 0000 = 0 come colonna: l'output sarà 14.

Se l'input fosse 2 = 000010, l'output si trova in posizione 00 = 0 come riga e 0001 = 1 come colonna: l'output sarà 4.

Se l'input fosse 38 = 100110, l'output si trova in posizione 10 = 2 come riga e 0011 = 3 come colonna: l'output sarà 8.

E' possibile trovare i file con le matrici delle S-Boxes, usati dallo script Python *utility/sboxes/gen_sboxes.py* per generare i moduli, nella cartella *utility/sboxes*

7.4 Implementazione interfaccia AXI

Per comunicare la APU e la Programmable Logic (FPGA) impiegano un bus standard basato sul protocollo AXI.

Nel protocollo AXI il dispositivo master (la APU) per leggere o scrivere dati verso il dispositivo slave (l'interfaccia AXI contenente la FSMD) utilizza il metodo input/output chiamato "Memory Mapped Input Output" che mappa i registri della periferica a zone di memoria: per leggere e scrivere su un registro il master deve conoscere l'indirizzo corrispondente al registro da controllare.

Per il progetto l'interfaccia AXI deve contenere 7 registri:

Indirizzo	Nome registro	Nome segnale
0x00	slv_reg0	msg
0x04	slv_reg1	ready_part1
0x08	slv_reg2	ready_part2
0x0C	slv_reg3	read_part1
0x10	slv_reg4	done
0x14	slv_reg5	enc_msg_part1
0x18	slv_reg6	enc_msg_part2

Ogni registro dell'interfaccia AXI e' da 32 bit.

- msg contiene la prima o la seconda parte del messaggio da cifrare ed essendo da 32 bit necessita di un solo registro

- ready_part1, indica che msg contiene la prima parte del messaggio, richiederebbe un solo bit ma il bus che si interfaccia con i registri e' da 32 bit, quindi e' necessario un registro da 32 bit
- ready_part2, indica che msg contiene la seconda parte del messaggio, richiederebbe un solo bit ma il bus che si interfaccia con i registri e' da 32 bit, quindi e' necessario un registro da 32 bit
- read_part1, indica che la FSM ha memorizzato la prima parte del messaggio, richiederebbe un solo bit ma il bus che si interfaccia con i registri e' da 32 bit, quindi e' necessario un registro da 32 bit
- done, indica che il messaggio intero e' stato cifrato, richiederebbe un solo bit ma il bus che si interfaccia con i registri e' da 32 bit, quindi e' necessario un registro da 32 bit
- enc_msg e' il messaggio cifrato ed essendo da 64 bit deve essere memorizzato in due registri da 32 bit: enc_msg_part1 e enc_msg_part2