Progetto Finale di Reti Logiche

Giovanni De Lucia - Matricola n. 933206

Anno Accademico 2021/2022

POLITECNICO
MILANO 1863

Indice

1 Introduzione

Il mio obiettivo per questo progetto, oltre a creare un design che funzionasse senza errori e nel rispetto delle specifica sia in behavioral che in post synthesis, è stato quello di creare un componente in cui è facile distinguere la FSM che gestisce i segnali dalla FSM "interna" che rappresenta il codificatore convoluzionale fornito nelle specifiche.
Viste le dimensioni ridotte del progetto non è stato necessario un focus sull'area occupata (la FPGA consigliata dispone di 134600 LUT e 269200 Flip-Flop) né sulla riduzione del periodo di clock (fissato a 100ns). Ho puntato, quindi, alla scrittura di un codice semplice nell'ottica di semplificare la sostituzione del codificatore convoluzionale.

1.1 Interfaccia del componente

```
entity project_reti_logiche is
      port (
            i_clk     : in std_logic;
            i_rst     : in std_logic;
            i_start   : in std_logic;
            i_data    : in std_logic_vector(7 downto 0);
            o_address : out std_logic_vector(15 downto 0);
            o_done    : out std_logic;
            o_en      : out std_logic;
            o_we      : out std_logic;
            o_data    : out std_logic_vector (7 downto 0)
      );
end project_reti_logiche;
```

In particolare:

- il nome del modulo deve essere project_reti_logiche
- i_clk è il segnale di CLOCK in ingresso generato dal TestBench;
- i_rst è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START;
- i_start è il segnale di START generato dal Test Bench;
- i_data è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- o_address è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- o_done è il segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- o_en è il segnale di ENABLE da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- o_we è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0;
- o_data è il segnale (vettore) di uscita dal componente verso la memoria.

1.2 Elaborazione specifiche di progetto

Il progetto consiste nella codifica di W parole da 1 byte e nella scrittura in memoria dei risultati. Si tratta quindi di leggere W, il numero di parole da codificare presente all'indirizzo 0, e per ognuna di queste (presenti negli indirizzi successivi allo 0, in ordine) produrne la codifica e scriverla in memoria a partire dall'indirizzo 1000.
Il codificatore ha rate 1/2 ed è quello mostrato in figura 2. Dunque viene letta una paraola da 1 byte, serializzata in uno stream da 8 bit, ognuno di questi viene dato in input al codificatore che ne genera 2. Quindi si ottiene uno stream da 16 bit che va parallelizzato su due parole da 1 byte, esse sono le parole che andranno scritte in memoria a partire dall'indirizzo 1000. Il ciclo si ripete finché ci sono parole da 1 byte da codificare. Il componente è stato progettato in modo da poter codificare più flussi in sequenza sia con un reset tra essi sia con un nuovo START=1 dopo il segnale di DONE posto a 1 e riportato basso. La specifica del progetto è ben riassunta dal flowchart in figura 1.

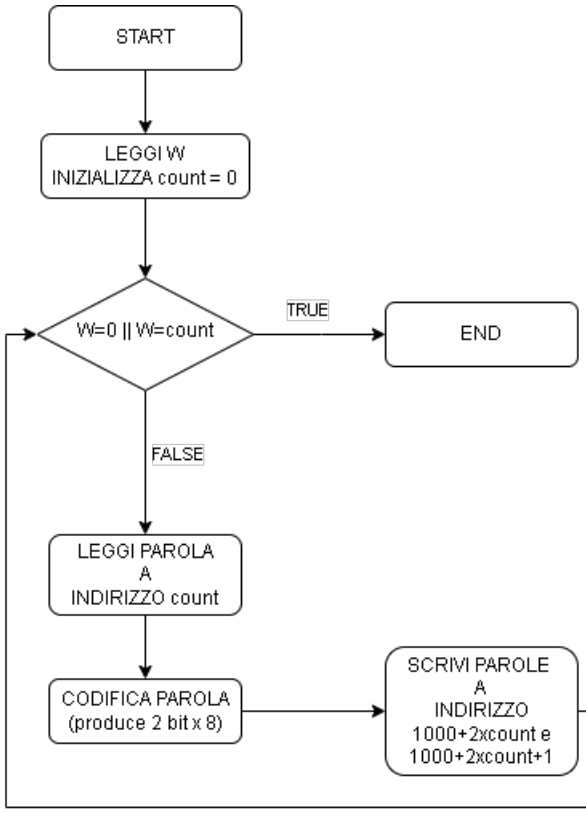


Figura 1

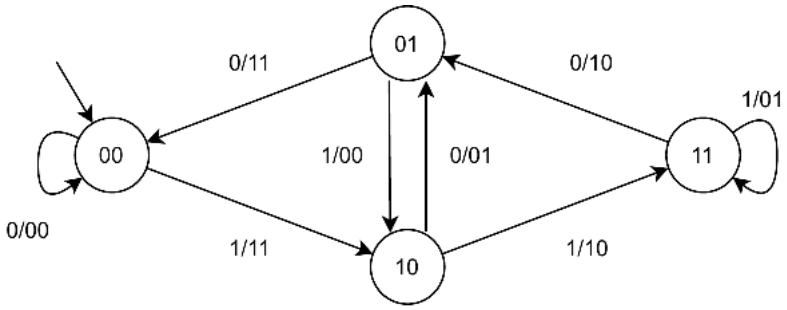


Figura 2: codificatore convoluzionale

1.3 Memoria: gestione e descrizione

La RAM istanziata dal Test Bench può essere vista come un vettore composto da 65536 (2^{16}) celle ognuna da 8 bit.
Si legge dalla RAM scrivendo l'indirizzo di lettura in `o_address` e abilitandola in lettura, cioè ponendo `o_en`=1 e `o_we`=0.
Si scrive in una cella della RAM scrivendo l'indirizzo in `o_address` e abilitandola in scrittura, cioè ponendo `o_en`=1 e `o_we`=1.
Come si può notare dalla figura 3, l'ultimo indirizzo utilizzato della memoria è $1001 + 2 * W$ che, dato $W_{max} = 255$ da specifica, vale al massimo 1511. La lettura richiede 1 ns (da Test Bench), per un funzionamento corretto, però, ho preferito inserire uno stato d'attesa suddividendola in due stati: "`read_req`" e "read"

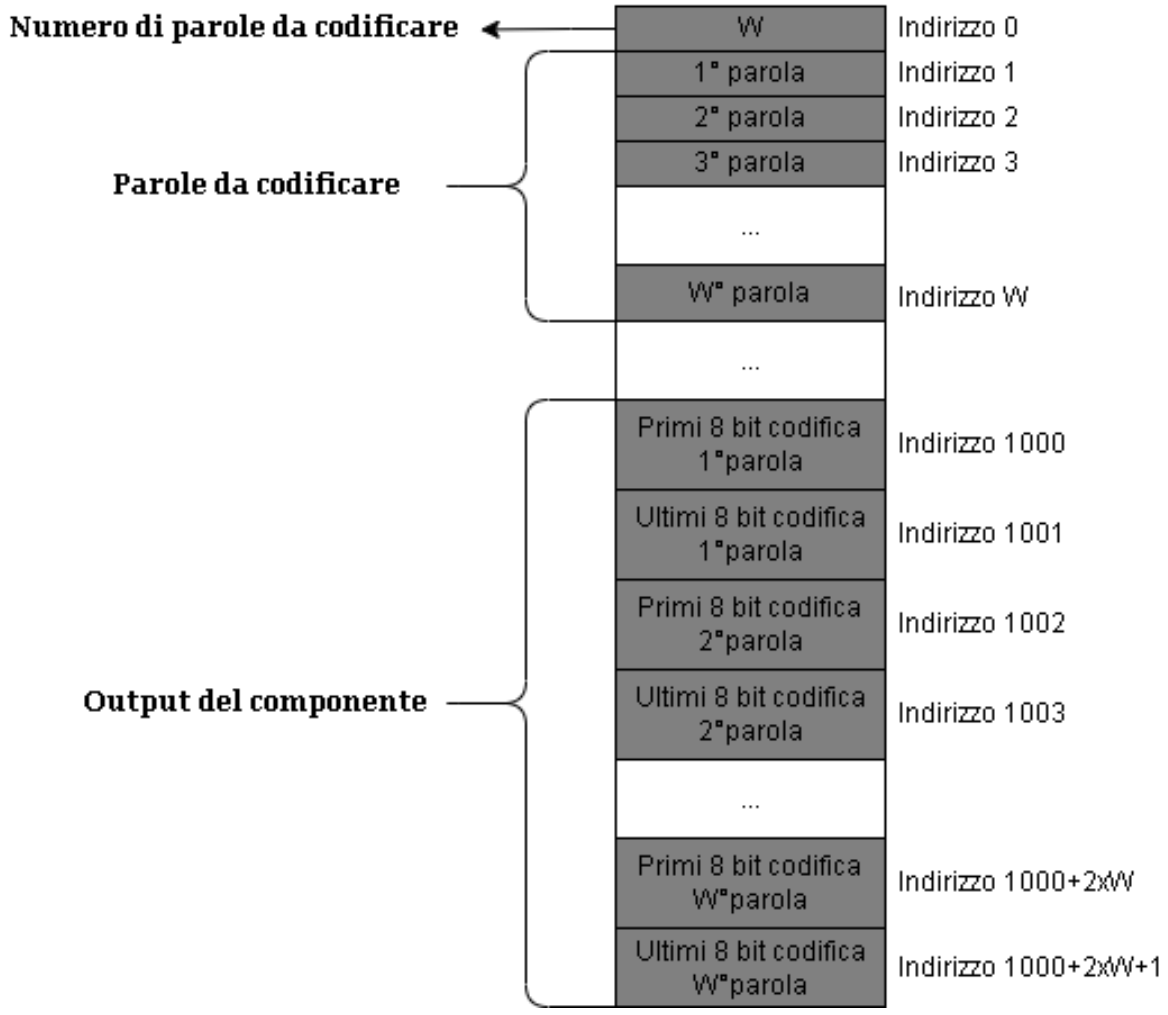


Figura 3: Rappresentazione indirizzi significativi memoria

2 Scelte progettuali: architettura e ottimizzazioni

2.1 Descrizione generale

Il componente hardware è una FSM con 18 stati. E' stata realizzata con due process. Un process è sensibile al clock e reset e aggiorna i valori dei registri e dello stato corrente ad ogni ciclo di clock. Un altro process si occupa della logica combinatoria ed è sensibile allo stato corrente. Ho usato diversi signal che descrivo nella sezione 2.2, per ognuno di questi c'è un signal `tmp_segnale` che sarà il valore che voglio assegnato al prossimo ciclo di clock. Ho scelto questo modo di assegnare/incrementare, ad esempio, i contatori per evitare l'inferimento di latch in post-synthesis. Quattro degli stati sono quelli del codificatore in figura 2. Si può facilmente cambiare algoritmo di codifica intervenendo soltanto su questi quattro stati se si assume che il nuovo codice convoluzionale abbia sempre rate 1/2 e che lo stato iniziale dello stesso sia S00. Questo è possibile perché ho utilizzato uno state "`state_curr_encoder`" in cui salvo il punto in cui sono arrivato nel codificatore.

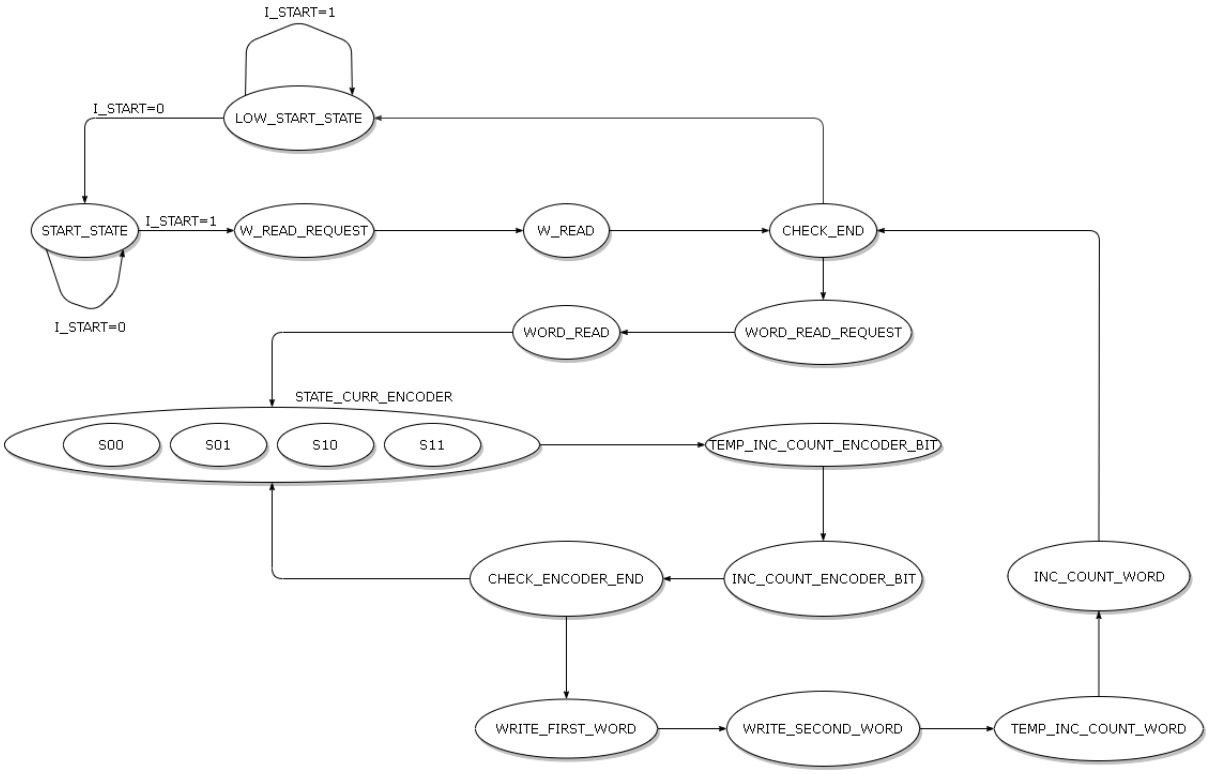


Figura 4: FSM semplificata
NOTA: `I_RST=1` conduce da ogni stato a `START_STATE`

2.2 Registri utilizzati

- **state_curr_encoder**: memorizza lo stato corrente del codificatore
- **o_encoder_data**: uscita del codificatore su 2 bit
- **i_word**: registro in cui salvo la parola letta dalla RAM e da codificare
- **encoded_word**: vettore di 16 bit in cui è salvata, man mano che la codifica procede, la doppia parola codificata
- **i_num**: è il registro in cui si salva W dopo la sua lettura
- **count_encoder_bit**: contatore dei bit della codifica, arriva a 8 poi si passa alla lettura e codifica della parola successiva
- **count_word**: conta le parole lette e codificate, arriva a W(=i_num) e poi termina la computazione

NOTA: I contatori partono da 0, non da 1

2.3 Stati della macchina

La macchina ottenuta non è quella minima per lo svolgimento di questo progetto ma per una maggiore suddivisione logica dei compiti da svolgere si è scelto di non ridurla ulteriormente. Nella tabella 1 si riporta una descrizione riassuntiva dei vari stati.

Stato	Descrizione
START_STATE	Stato di reset. Aspetta l'inizio della computazione
W_READ_REQUEST	Prima richiesta di lettura da RAM, pone o_en=1 e o_we=0 e o_address=0
W_READ	Dopo aver atteso un ciclo di clock posso concludere la lettura da RAM e salvare W in i_num
CHECK_END	Verifica se W appena letto è 0 o, nel caso di iterazioni successive se ho codificato tutte le parole (count_word=W). In tal caso viene posto o_done=1
WORD_READ_REQUEST	Richiedo la lettura della parola all'indirizzo count_word+1
WORD_READ	Dopo aver atteso un ciclo di clock conclude la lettura e salva la parola da codificare in i_word
STATE_CURR_ENCODER	E' la visione dall'esterno del codificatore. A questo stato, che può essere S00, S01, S10, S11 viene dato in input il bit indicato da count_encoder_bit della parola da codificare. Verrà aggiornato o_encoder_data con i 2 bit di output
TEMP_INC_COUNT_ENCODER_BIT	Aggiorna il registro a 16 bit che contiene la "doppia" parola codificata con i due bit usciti dal codificatore. Incrementa tmp_count_encoder_bit
INC_COUNT_ENCODER_BIT	Aspetta che i cambiamenti diventino effettivi (1 ciclo di clock)
CHECK_ENCODER_END	Controlla se ho finito la codifica della parola letta da RAM (count_encoder_bit = 8), se è così scrivo in memoria i risultati
WRITE_FIRST_WORD	Scrivo gli 8 bit più significativi della parola codificata all'indirizzo 1000+(2∗count_word)
WRITE_SECOND_WORD	Scrivo gli 8 bit meno significativi della parola codificata all'indirizzo 1000+(2∗count_word)+1
TEMP_INC_COUNT_WORD	Incremento tmp_count_word
INC_COUNT_WORD	Aspetto che il cambiamento di count_word diventi effettivo (1 ciclo di clock) poi vado in CHECK_END per verificare se ho codificato tutte le parole
LOW_START_END	Da specifica dopo la fine della computazione, prima che ne possa iniziare un'altra devo attendere che il segnare I_START vada LOW.

Tabella 1: Tabella con descrizione degli stati

3 Testing

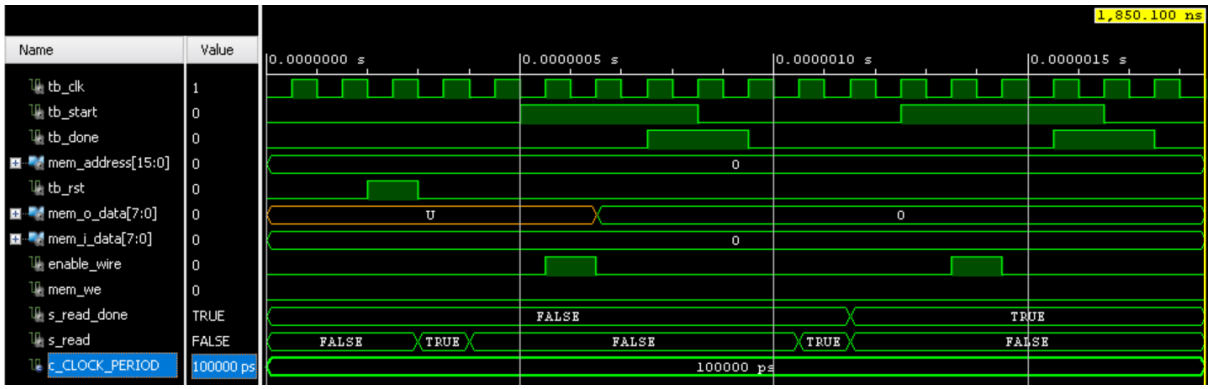
3.1 Test con generatore

La macchina è stata sottoposta ad un elevato numero di test sia in behavioral che in post-synthesis functional generati con un software scritto in python. Il programma genera un file con il contenuto della RAM e l'output atteso. Un test bench caricato in VIVADO si occupa di leggere questo contenuto, caricarlo nella RAM, far funzionare il componente e poi verificare e suddividere tra "PASSATI" e "NON PASSATI" i vari test uno dopo l'altro. Il numero elevato di test è pensato per garantire una copertura quasi totale dei casi, oltre ovviamente a verificare che il componente è in grado di gestire più flussi di computazione uno dopo l'altro sia con reset tra essi sia con un nuovo segnale di START.

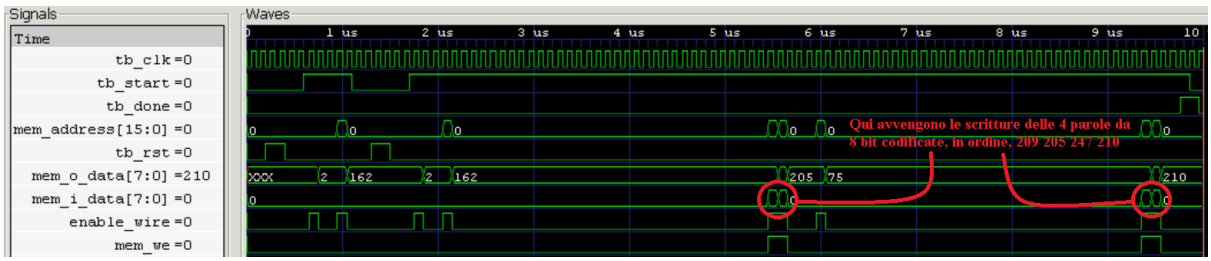
3.2 Corner case

I corner case a cui ho pensato e che ho testato sono sostanzialmente 4:

- **W=0** Non c'è nessuna parola da codificare, mi aspetto che il componente legga W si accorga che è 0 e termini. Nel caso test scelto ho due flussi consecutivi con W=0. In effetti come si nota dalla waveform l'indirizzo della RAM è sempre fisso a 0, non viene scritto nulla, vengono letti i due zeri e subito dopo viene alzato o_done.

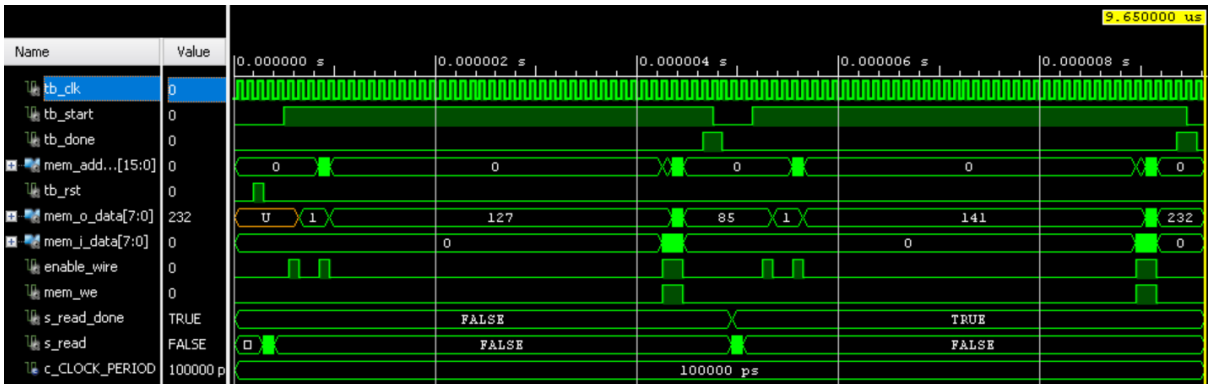


- **Reset asincrono** In caso di reset asincrono il componente reagisce letteralmente iniziando da capo la computazione, come ci si aspetterebbe. Ho preso il test bench fornito con le specifiche e ho aggiunto un reset durante la computazione. I valori scritti in memoria sono quelli giusti e ai giusti indirizzi



- **W=255** Il caso di W=W_{MAX}. Risulta difficile riuscire a visualizzare tutta la waveform in una immagine ma con il generatore di test e il test bench che ho usato, questo caso è stato ampiamente verificato e soddisfatto.

- **Multi-Start** Da specifica il componente deve essere in grado di codificare più flussi uno dopo l'altro senza un reset tra essi. Ho testato quindi il caso di multi-start.



4 Sintesi

4.1 Tool e FPGA

Il tool di sintesi utilizzato è "VIVADO 2016.4". La FPGA target consigliata è la Artix-7 FPGA xc7a200tfbg484-1.

4.2 Risultati della sintesi

Il componente sintetizzato non presenta errori segnalati da VIVADO. Gli unici warning presenti riguardano il segnale o_address per cui non vengono utilizzati i 5 bit più significativi.
Questo non dipende dall'implementazione ma dal fatto che l'indirizzo massimo a cui si accede, da specifica è $o\_address_{MAX} = 1000 + 2 * W_{MAX} + 1 = 1511$.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	95	0	134600	0.07
LUT as Logic	95	0	134600	0.07
LUT as Memory	0	0	46200	0.00
Slice Registers	54	0	269200	0.02
Register as Flip Flop	54	0	269200	0.02
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Figura 5: (parte di) report_utilization

```
Timing Report

Slack (MET) :           96.333ns  (required time - arrival time)
  Source:                 i_num_reg[3]/C
                            (rising edge-triggered cell FDCE clocked by clock  (rise@0.000ns fall@5.000ns period=100.000ns))
  Destination:            state_curr_reg[1]/D
                            (rising edge-triggered cell FDCE clocked by clock  (rise@0.000ns fall@5.000ns period=100.000ns))
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            100.000ns  (clock rise@100.000ns - clock rise@0.000ns)
  Data Path Delay:        3.516ns  (logic 0.999ns (28.413%)  route 2.517ns (71.587%))
```

Figura 6: (parte di) report_timing

4.3 Osservazioni sulla sintesi

Dalla tabella in figura 5 si può notare che non sono stati inferiti latch. Dalla tabella in figura 6 invece si può notare che si ha uno slack pari a $\approx 96.333ns$. Dunque, in realtà, il componente è in grado di funzionare ad una frequenza di clock f_{clk} maggiore di quella data da specifica che è pari a $\frac{1}{100ns} = 10$ MHz. In particolare posso approssimare un tempo minimo di clock pari a $T_{min} = 100ns - Slack + T_{RAM} \approx 4.667ns$ che corrisponde ad una frequenza di clock di circa 215 MHz

5 Conclusioni

In conclusioni si può affermare che il componente ha le seguenti caratteristiche:

1. Facile manutenzione, sostituzione ed integrazione dell'algoritmo di codifica
2. Codice semplice
3. Basso uso delle risorse della FPGA: non utilizza LATCH, utilizza lo 0.07% di LUT disponibili e lo 0.02% di FF (non latch) disponibili
4. Può funzionare fino a una frequenza di clock di circa 215 MHz