



Analizzatore di stati logici

Andrea Malacrino
Marco Montagna
Emanuele Valpreda
Sistemi digitali integrati

Indice degli argomenti

- Introduzione al progetto	pagina 1
- Descrizione delle specifiche assegnate	pagina 3
- Descrizione qualitativa scelte progettuali	pagina 4
- Sampler	pagina 7
- Trigger generato	pagina 10
- Memory interface	pagina 13
- Codifica ASCII	pagina 22
- Decodifica ASCII	pagina 26
- UART TX	pagina 33
- UART RX	pagina 36
- PC interface	pagina 40
- MAIN FSM	pagina 47
- Test in laboratorio	pagina 56

▪ Introduzione al progetto

Durante il corso di sistemi digitali integrati ci è stato assegnato un progetto di laboratorio da svolgere, ovvero la realizzazione di un analizzatore logico ad 8 canali con possibilità di selezionare diverse frequenze di campionamento e trigger.

L'analizzatore logico è un dispositivo in grado di campionare un diverso numero di segnali di ingresso digitali, cioè con dinamica di ingresso limitata ai livelli logici tipicamente utilizzati nell'elettronica digitale. In questo progetto non ci preoccupiamo della dinamica d'ingresso, ovvero di condizionare i segnali di input con dei circuiti analogici che eventualmente forniscano anche una protezione da cortocircuiti, sovratensioni e altre possibili cause di malfunzionamento, ma solo della parte di salvataggio, elaborazione parziale e invio dei dati ad una periferica in grado di comunicare tramite il protocollo RS-232.

Abbiamo quindi progettato anche l'interfacciamento della nostra macchina con un oggetto esterno in grado di elaborare in post-processing i dati raccolti, senza preoccuparci della parte software dedicata alla connessione seriale o all'analisi off-line dei campioni.

L'analizzatore logico permette di visualizzare l'andamento temporale di più segnali contemporaneamente, con un numero di canali decisamente più grande rispetto ad un normale oscilloscopio, con una banda adatta all'utilizzo su circuiti che lavorino con frequenze elevate rispetto a quelli analogici, utilizzando segnalazioni tipicamente ad onda quadra. Non è quindi detto che sia necessario un vero e proprio campionamento del segnale con un convertitore A/D, in quanto solitamente è importante conoscere il livello logico dei segnali di interesse, le loro commutazioni e dipendenze da altri segnali o dallo stato della macchina e, soprattutto, la presenza di eventuali glitch che potrebbero causare malfunzionamenti. In realtà si utilizza comunque un ADC veloce con bassa risoluzione per poter osservare anche i tempi di salita e discesa dei segnali. Una ulteriore differenza rispetto ad un classico oscilloscopio è la possibilità di visualizzare segnali aperiodici, usando come condizione di trigger la salita/discesa/livello di più segnali in contemporanea.

Grazie a queste features è possibile usare l'analizzatore anche per decodificare gli stati di una macchina o verificare l'andamento dei segnali usati in un protocollo di handshake, oppure in una interfaccia SPI o ancora vedere come variano i segnali di controllo di una memoria. I possibili utilizzi si estendono al debug/test di praticamente tutta la parte digitale di un circuito elettronico.

In questa relazione viene descritto il nostro lavoro e le nostre scelte progettuali. Il circuito è stato diviso in una serie di blocchi più piccoli, ciascuno con una sua unità di controllo, che lavorano in modo dipendente tra di loro. Abbiamo quindi affrontato il problema di stabilire dei protocolli di interfaccia semplici a basso livello, dove richiesti, utilizzano delle semplici tecniche di handshake, ricorrendo anche all'utilizzo di semafori per far vedere alla control unit a livello più alto lo stato di alcuni segnali, non controllati direttamente da lei, che influenzano l'evoluzione della macchina nel suo insieme.

▪ Descrizione delle specifiche assegnate

L'analizzatore logico che ci è stato richiesto di progettare deve seguire alcune linee guida, elencheremo dapprima quelle generali:

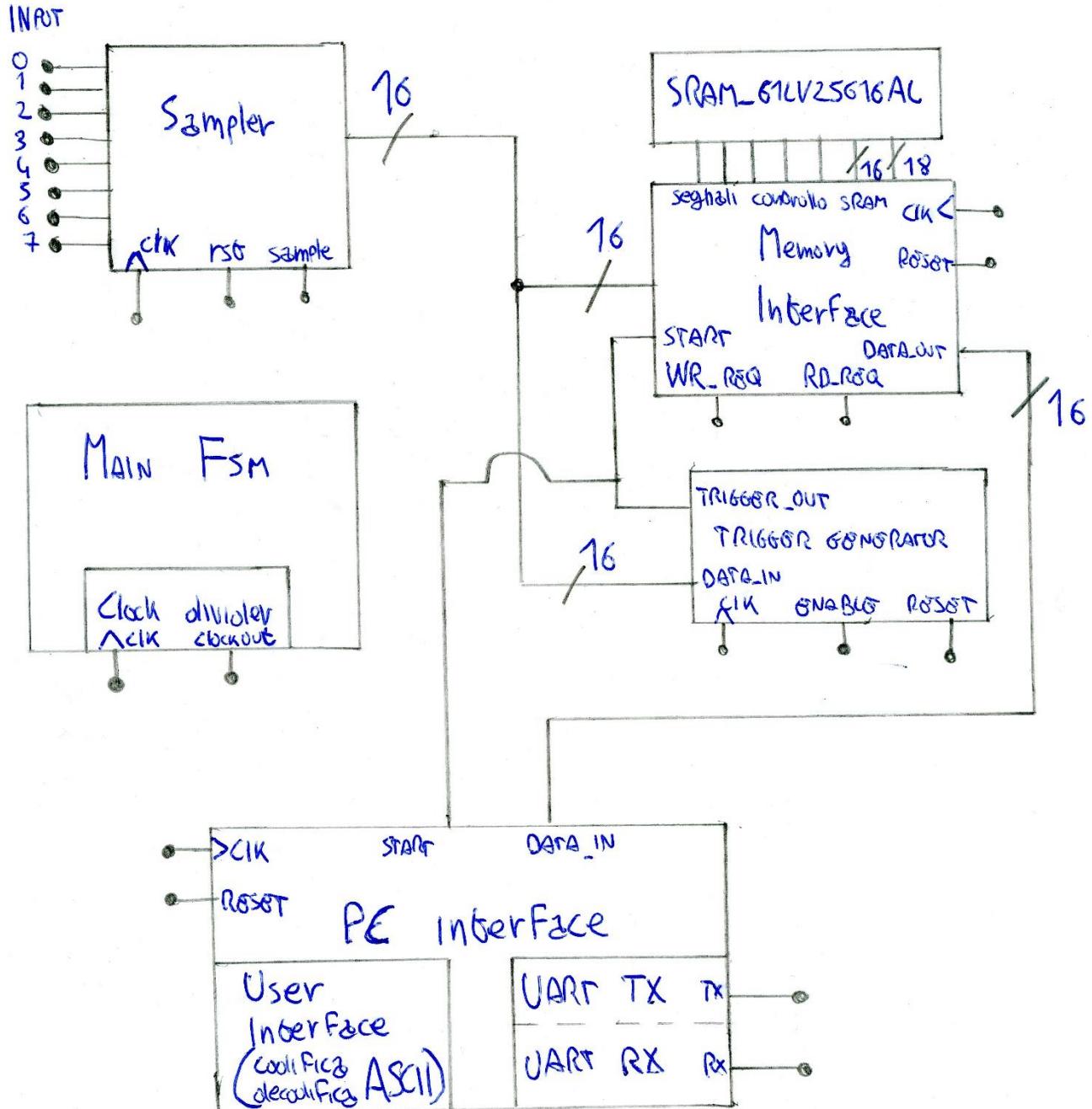
- 8 canali di ingresso digitali;
- Rilevazione dei glitch, sotto ipotesi che se ne verifichi al più uno sul tempo di campionamento considerato e che sia di interesse solo verificare che ne sia avvenuto almeno uno e non il numero totale, in caso di più commutazioni spurie;
- Possibilità di modificare la frequenza di campionamento, con le seguenti opzioni: 10MHz, 5MHz, 2.5MHz, 1.25MHz, 625kHz, 312.5kHz, 156.25kHz, 78.125kHz, 39.063kHz e 19.531kHz;
- Possibilità di modificare il pattern di trigger, indicante la combinazione dei valori logici degli ingressi che si vuole cercare;
- Salvataggio dei campioni e possibili glitch su una memoria SRAM da 256k e parallelismo a 16bit, da usare come buffer circolare;
- Campionamento continuo dei dati in ingresso e salvataggio di ulteriori 128k campioni dopo che si verifica la condizione di trigger: in questo modo metà della memoria contiene 128k campioni pre-trigger che permettono di vedere le evoluzioni passate dei segnali e 128k campioni che permettono invece di vedere le evoluzioni post-trigger;
- Utilizzo della linea seriale RS-232, insieme ad un programma fornитoci per la gestione della linea da PC, per inviare dal calcolatore all'analizzatore alcuni comandi come la frequenza di campionamento (si invia un numero che funge da prescaler), il pattern di trigger, il comando di start (che permette al circuito di sentire il trigger e iniziare a salvare i 128k campioni) e la richiesta di lettura (utilizzata per leggere tutta la memoria dell'analizzatore);
- Il circuito dell'analizzatore responsabile della gestione dell'interfaccia RS-232 deve inviare anche la conferma della corretta ricezione del comando oppure inviare un messaggio di errore in caso di ricezione errata o comando sbagliato/non definito;
- I caratteri inviati e ricevuti sulla linea seriale devono utilizzare la codifica ASCII;
- Il baud rate è fissato a 115200 con una tolleranza ammissibile del 3% sulla frequenza di trasmissione.

Le specifiche sono state discusse preliminarmente in aula, durante la lezione introduttiva, e affrontate in modo più approfondito in fase di progettazione.

- Descrizione qualitativa dei componenti del circuito e scelte progettuali**

Una volta capite sia le specifiche progettuali che la struttura RTL del circuito vista a lezione possiamo iniziare a fare delle considerazioni sui componenti da utilizzare e sul parallelismo dei BUS locali.

Innanzitutto stabiliamo la versione di riferimento del datapath dell'analizzatore logico, che è la seguente:



Gli ingressi e uscite sono puramente indicativi. La larghezza di una word della SRAM è 16bit ed è indirizzata con 18bit. UART TX e RX è connessa alla porta seriale RS-232 della board DE2. La Main FSM è la control unit a livello più alto ed il suo datapath è, ripetizione necessaria, il datapath dell'analizzatore.

Segue un'analisi preliminare dei vari blocchetti visti sopra, la descrizione dettagliata verrà fatta nei paragrafi successivi.

- Main Fsm: è l'unità di controllo che gestisce il funzionamento dell'intero circuito, che comanda i vari blocchetti tramite una serie di segnali di controllo. Tra le varie funzioni è responsabile di attivare il reset sincrono che viene inviato a tutto il circuito;
- Sampler: è il blocco in grado di campionare i dati provenienti dagli 8 ingressi digitali. Supponiamo di avere in ingresso già dei livelli logici validi. Solitamente si utilizzano in ingresso dei circuiti analogici di condizionamento e protezione più un comparatore, noi non ci preoccupiamo di questa parte. Inoltre questo blocco è composto da una unità fondamentale ripetuta otto volte: gli 8 canali sono uguali e indipendenti tra loro. Possiamo dunque progettare e testare (inizialmente) una sola unità e poi replicarla altre 7 volte;
- Contatore a tc variabile: è un contatore che, come dice il nome, permette di variare il terminal count. L'ampiezza degli impulsi dovuti al segnale di fine conta è la stessa di quella di un periodo di clock a 50MHz, il clock principale della macchina, la frequenza con cui questi impulsi vengono generati, che controlleranno il sampler e quindi decidono la frequenza di campionamento, verrà impostata dall'utente esterno;
- Trigger generator: blocco responsabile di verificare la condizione di trigger, cioè l'egualanza tra i dati campionati e il pattern impostato dall'utente. Quando questa condizione è verificata si attiva un segnale di start che viene sentito solo se è stato abilitato dall'utente;
- Memory interface: questo circuito si occupa del salvataggio dei campioni in uscita dal sampler nella SRAM montata sulla DE2, oltre che della lettura degli stessi. Lo scopo principale di questa unità è quello di rispettare il timing della SRAM e gestire gli spostamenti dei dati nella memoria. La memory interface contiene al suo interno un contatore per indirizzare la memoria e un registro per salvare l'indirizzo corrispondente alla configurazione degli ingressi che soddisfa la condizione di trigger;
- PC interface: è responsabile dell'interfacciamento dell'analizzatore con il mondo esterno. Gestisce la linea seriale RS-232 a livello di protocollo di comunicazione, codificando e decodificando i dati inviati e letti da essa. I comandi inviati dall'utente vengono elaborati dalla PC interface e tradotti in una serie di segnali interpretabili dalla main FSM dell'analizzatore logico. La PC interface comprende due sotto-blocchi: la UART, utilizzata per la trasmissione e la ricezione dei caratteri ASCII sulla linea seriale, e la User interface, responsabile della decodifica dei comandi inviati dall'utente e della codifica dei dati generati dall'analizzatore, anche questi in caratteri ASCII.

Analizzati i componenti a livello RTL (register transfer level) e disegnata una prima bozza dei vari datapath decidiamo che ci serviranno:

- Registri di diverso parallelismo;
- Shift register;
- Flip flop di tipo D;
- Contatori di diverso parallelismo con terminal count fisso e variabile;
- Multiplexer di diverso parallelismo e numero di ingressi;
- Comparatori;
- Sommatori;
- Porte TRI-STATE;
- Porte logiche come NOT, AND etc. per implementare alcune funzioni puramente combinatorie.

L'utilizzo e l'implementazione dei vari componenti scritti sopra sarà spiegata meglio nei paragrafi successivi, laddove vengano usati.

Il RESET in tutti i componenti dell'analizzatore è sincrono e attivo basso, tutti gli elementi di memoria sono sensibili ai fronti di salita del CLOCK.

L'unica unità sensibile al RESET asincrono è la Main FSM, che comanderà di conseguenza un RESET sincrono collegato al resto del circuito.

Iniziamo adesso a descrivere tutti i componenti, sviluppati seguendo il metodo di progettazione consigliatoci dal professore a lezione.

Abbiamo quindi definito un datapath ed un timing, per regolare il funzionamento di ogni blocco anche per quanto riguarda l'interfacciamento con gli altri circuiti dell'analizzatore. Poi abbiamo tracciato il pallogramma e definito una FSM. In seguito abbiamo scritto il VHDL ed eseguito tutte le simulazioni del caso, provvedendo a modificare datapath timing e pallogramma (senza fare modificare necessariamente tutte e tre le cose ma rispettando rigorosamente in quest'ordine), per poi correggere il codice e eseguire nuovamente le simulazioni.

Per una questione di forma vengono presentati prima le FSM e poi i timing, nel caso in cui il circuito analizzato sia semplice. Laddove serve invece una spiegazione più approfondita sulle scelte di progetto vengono invece mostrati e commentati prima i timing diagram.

Ci teniamo inoltre a precisare che alcuni pallogrammi presenti in questa relazione sono stati presi dalla State Machine View di QUARTUS per una semplice questione di estetica, sono infatti più puliti ed ordinati dei nostri e hanno la comodità di essere già pronti per essere inseriti in un documento, senza dover passare per uno scanner. Ovviamente tutti i pallogrammi ottenuti in questo modo sono risultati essere uguali ai nostri. Nel caso i pallogrammi ottenuti con QUARTUS fossero troppo confusionali abbiamo inserito le nostre copie finali di quelli utilizzati in fase di progettazione.

I datapath sono stati scritti a mano, mentre i timing sono stati creati con TimingEditor.

I programmi utilizzati per la creazione e test del codice hardware sono QUARTUS e MODELSIM, scelta imposta dalla board Altera DE2 utilizzata in laboratorio.

Abbiamo riportato i risultati delle simulazioni evidenziando alcune parti ritenute significative. Insieme a questa relazione si possono trovare allegati tutti i testbench utilizzati più il progetto QUARTUS finale.

▪ Sampler

Il sampler è un blocco a cui viene affidato il compito di campionare i segnali in ingresso ed al tempo stesso rilevare i glitch, ove presenti, e inviare i dati risultanti su un BUS locale a 16 bit, collegato con la memory interface e il trigger generator.

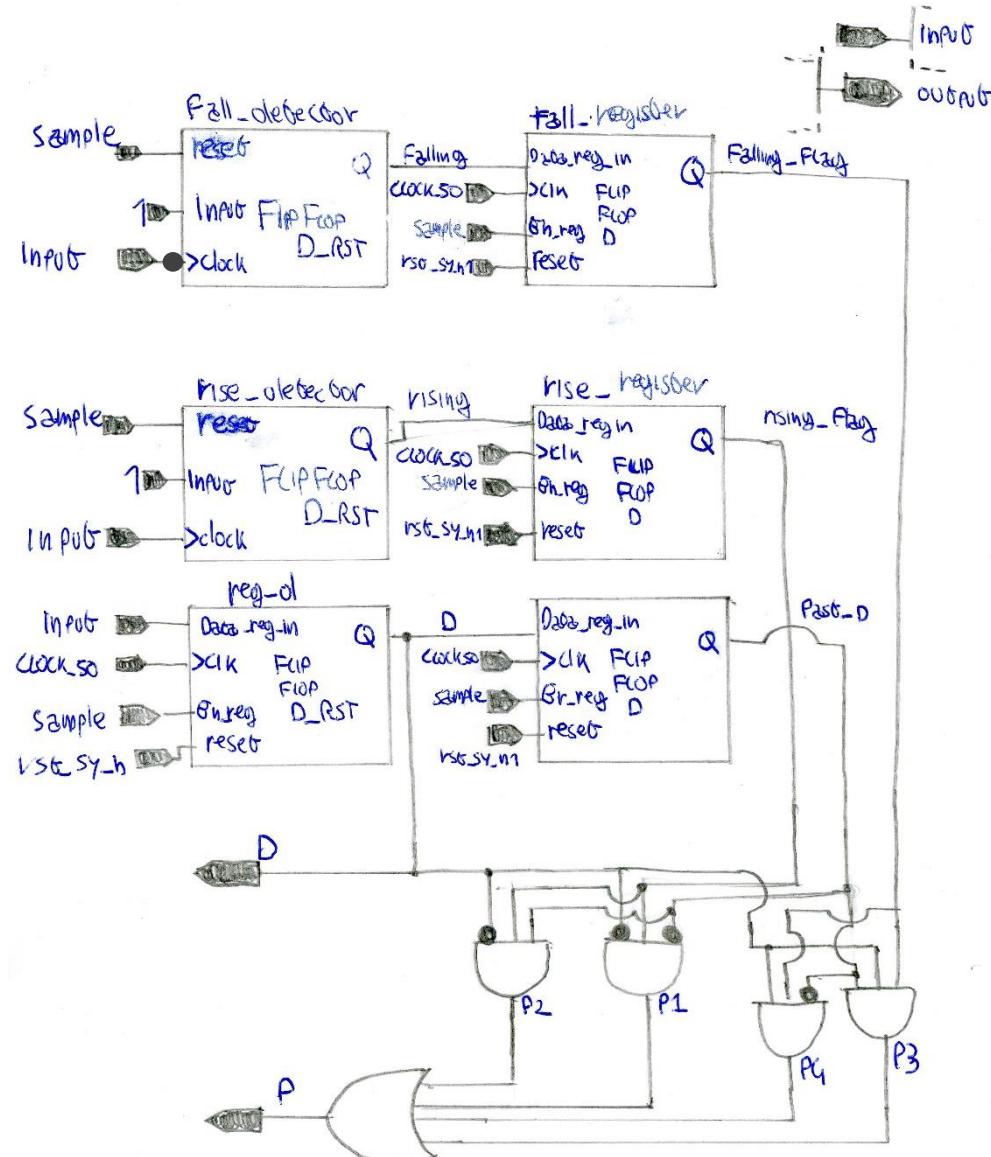
○ Datapath

Esso è composto da 8 unità uguali che lavorano in modo indipendente tra di loro e in parallelo, ma rispondono tutte agli stessi comandi provenienti dalla main FSM.

Ogni unità possiede una propria control unit ed un datapath, ha una porta di ingresso a cui viene collegato il segnale da campionare e due di uscita su cui vengono mandati i dati campionati e i glitch rilevati. Sono presenti anche altre 3 porte con i segnali di clock, reset sincrono e sample collegati.

Il funzionamento di questo circuito è abbastanza banale: si campionano gli ingressi tramite dei FLIP FLOP e si verificano sia il valore del segnale di ingresso all'istante di campionamento, sia la presenza di commutazioni spurie.

Lo schema del datapath è il seguente:

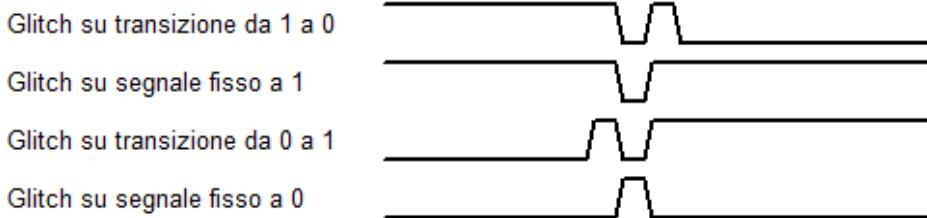


input è attivo basso, l'uscita di rise_detector viene asserita.

Il segnale di ingresso viene campionato dal FLIP FLOP reg_d per trovare il valore logico D, che rappresenta lo stato dell'input e viene propagato all'uscita ed al FLIP FLOP past_d, contenente il valore assunto dal dato al colpo di clock precedente.

Il segnale di input viene anche mandato al FLIP FLOP rise_detector e complementato al FLIP FLOP fall_detector.

Se si verifica un glitch mentre il dato è alto viene asserita l'uscita del FLIP FLOP fall_detector, se invece si ha un glitch mentre il segnale di



I FLIP FLOP vengono utilizzati per rilevare i fronti di salita e discesa del segnale di ingresso, il glitch viene rilevato confrontando le uscite **FALLING_FLAG** e **RISING_FLAG** con i valori di D e past D. Ci sono 4 tipi di

glitch che possono verificarsi, sono riassunti brevemente nell'immagine a lato.

Il comando **SAMPLE** oltre ad attivare i FLIP FLOP resetta anche quelli utilizzati per rilevare i fronti di salita o discesa dei glitch. In questo modo ad ogni nuova richiesta di campionamento le uscite vengono ripristinate ai valori di default, cioè 0, questo è necessario perché, come si può vedere, due FLIP FLOP particolari sono sempre abilitati.

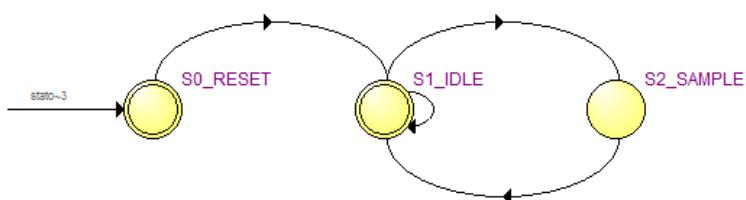
Le uscite di tutti i FLIP FLOP infine convergono in una serie di porte AND, con cui si verifica se è avvenuta una commutazione spuria durante la fase di campionamento. Il risultato viene mandato in una porta OR che rappresenta l'uscita del circuito **P**, indicante per l'appunto la presenza di un glitch, quando essa viene asserita.

Quando i FLIP FLOP di rising e falling vengono resettati, per il colpo di clock successivo al reset sono "ciechi": non sono in grado di sentire i glitch. Per risolvere questo problema avremmo dovuto implementare un sistema con il doppio degli elementi di memoria usati per rilevare i fronti di salita e discesa del segnale di ingresso, così che lavorino in modo complementare tra loro, ovvero che quando uno non è in grado di sentire il glitch l'altro invece può rilevarlo.

○ FSM e Timing Diagram

La macchina a stati del sampler è composta da 3 stati: uno di reset, uno di idle ed infine lo stato di sample, in cui si esegue il campionamento dei dati e la rilevazione dei glitch.

Di seguito possiamo vedere il pallogramma della FSM:



Il primo stato è **S0_RESET**, la macchina lo sente solo quando il segnale di reset viene negato in corrispondenza di un fronte positivo del clock. Il passaggio a **S0_RESET**, da qualsiasi altro stato avviene dunque

in modo sincrono, sia che esso venga pilotato dalla Main Fsm, sia che esso venga raggiunto nel caso in cui la macchina finisce in uno stato non previsto.

La macchina rimane nello stato di **S1_IDLE** fino a quando non riceve la richiesta di sample, dopodiché passa nello stato **S2_SAMPLE**, esegue il campionamento e ritorna in **S1_IDLE**. In sample si esegue il reset dei FLIP FLOP usati per rilevare i fronti di salita e discesa dovuti alla presenza di un glitch oppure ad una normale commutazione del segnale.

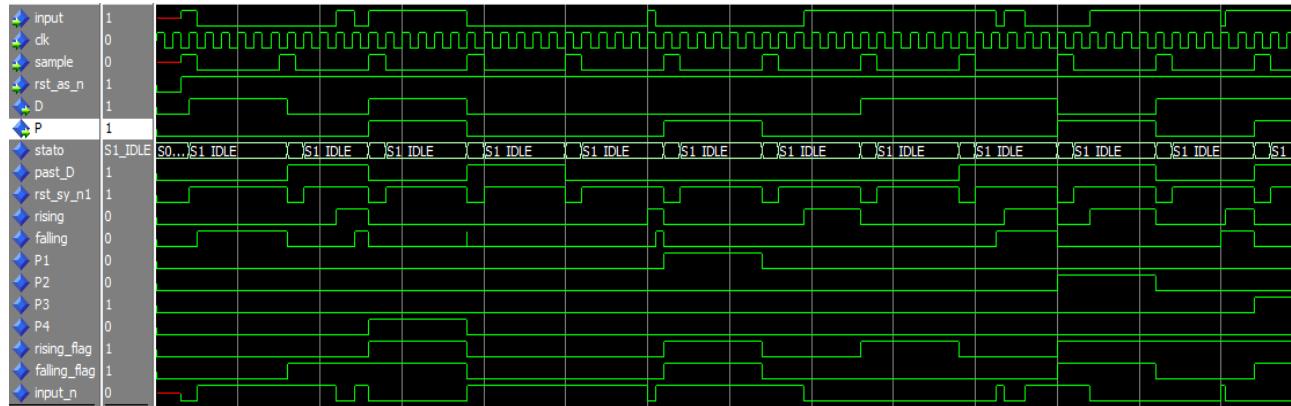
Il timing diagram sarà dunque il seguente:



Come si può vedere dal timing, dopo che **SAMPLE** è stato asserito, i FLIP FLOP campionano e in modo combinatorio dalle loro uscite si ottengono i segnali di dato **D** e glitch **P**. Se viene rilevato un glitch **P** viene asserito, mentre **D** viene asserito solo se **INPUT** è alto, mentre viene negato se **INPUT** è basso.

Abbiamo utilizzato direttamente i segnali che poi andremo ad utilizzare in VHDL. Il segnale di clock utizzato è **CLOCK_50**, cioè quello preso direttamente dalla DE2, per evidenziare che la macchina a stati e il datapath lavorano alla frequenza di 50MHz. In verità i segnali di **SAMPLE** dovrebbero arrivare, nel caso di massima frequenza di campionamento a 10MHz, ogni 5 colpi di clock.

Di seguito includiamo l'esito della simulazione modelsim.

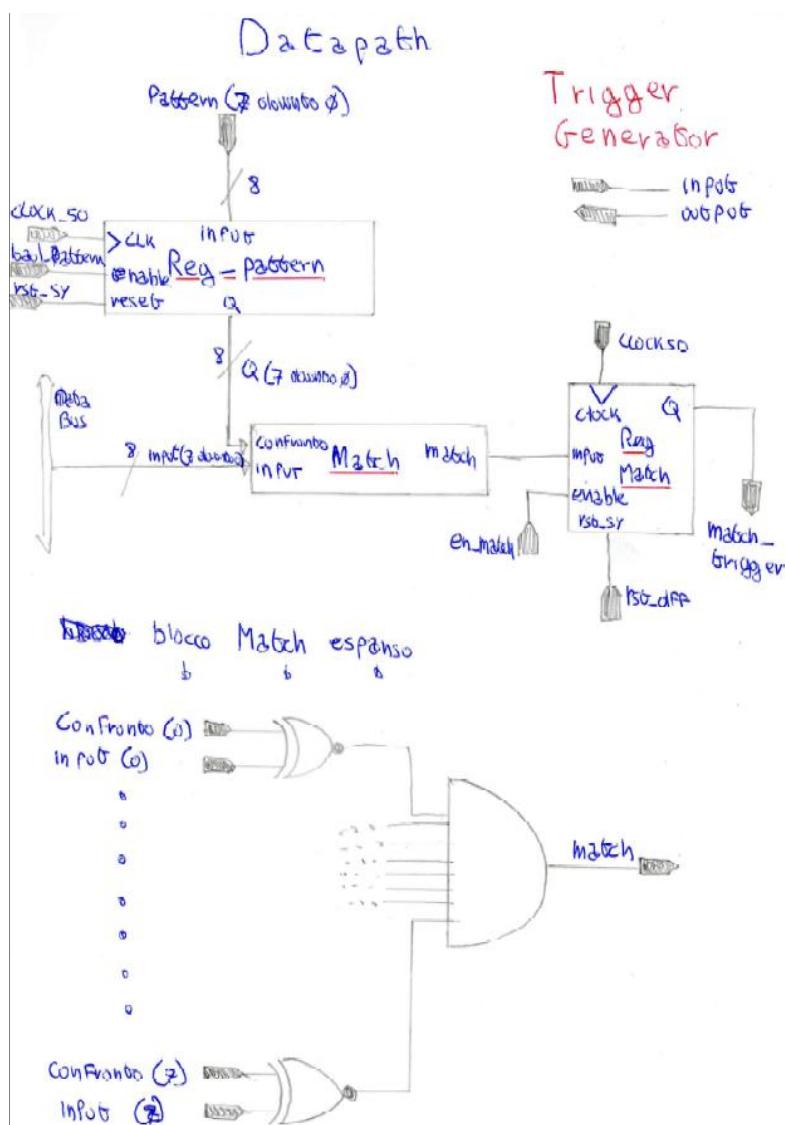


Come si può osservare dalla simulazione, un fronte sul segnale di ingresso fa commutare uno dei due FLIP FLOP rising o falling. Il dato viene campionato al primo fronte utile del clock dai FLIP FLOP collegati in cascata. Il segnale di glitch viene generato in modo combinatorio e mantenuto stabile, insieme al segnale di dato, fino al prossimo ciclo di campionamento, dove tutti gli elementi di memoria del circuito vengono resettati o sovrascritti.

▪ Trigger generator

Il trigger generator riceve in ingresso i dati campionati dal sampler e li confronta col pattern di trigger definito dall'utente. È un circuito molto semplice, composto da un registro in cui viene salvata la sequenza di uni e zeri che rappresenta il pattern di trigger cercato ed un insieme di porte logiche, EXNOR e AND che eseguono la comparazione tra dati in ingresso e uscita del registro appena descritto. Il risultato viene salvato in un FLIP FLOP e inviato alla MAIN FSM.

○ Datapath



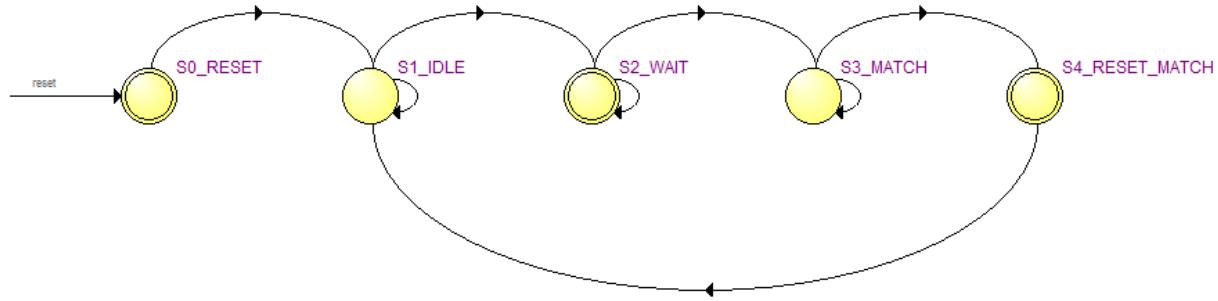
Il registro `reg_pattern` riceve in ingresso il segnale **PATTERN**, rappresentante la sequenza di trigger. L'uscita di tale registro è l'ingresso di un blocchetto chiamato `Match`, il cui interno è osservabile nella parte inferiore del grafico: è composto da una serie di porte logiche EXNOR, comparatori che verificano che gli ingressi ad esso collegati siano uguali, in tal caso l'uscita dell'EXNOR vale 1, altrimenti 0. All'uscita di questi 8 comparatori è collegato un AND che esegue la moltiplicazione di tutti i segnali di comparazione e asserisce un 1 logico nel caso in cui la combinazione degli input sia uguale a quella del pattern. L'uscita di questo AND, **MATCH**, è collegata all'ingresso di un FLIP FLOP che è sempre abilitato, quando la sequenza di trigger è stata impostata per la prima volta, e viene resettato dopo che si è verificata la condizione di trigger, se e solo se viene asserito un segnale dall'esterno, **TRIGGER_ACK**, che porta la control unit in uno stato in cui viene per l'appunto negato il segnale **RST_DFF**.

Il segnale **MATCH_TRIGGER** è collegato direttamente alla `memory_interface`, che una volta ricevuto asserisce il segnale **TRIGGER_ACK** per resettare il FLIP FLOP `reg_match` ed al tempo stesso comunica alla Main FSM dell'analizzatore, tramite un FLIP FLOP usato come semaforo, che si è verificata la condizione di trigger.

- **FSM e timing diagram**

Il controllo di questo blocco sarà composto verosimilmente da 5 stati: uno di reset, uno di idle, uno in cui si aspetta il verificarsi della condizione di trigger, uno in cui si attende che venga asserito il segnale di acknowledge del trigger ed infine uno stato in cui si resetti il FLIP FLOP in cui è salvato il trigger e si torni in idle.

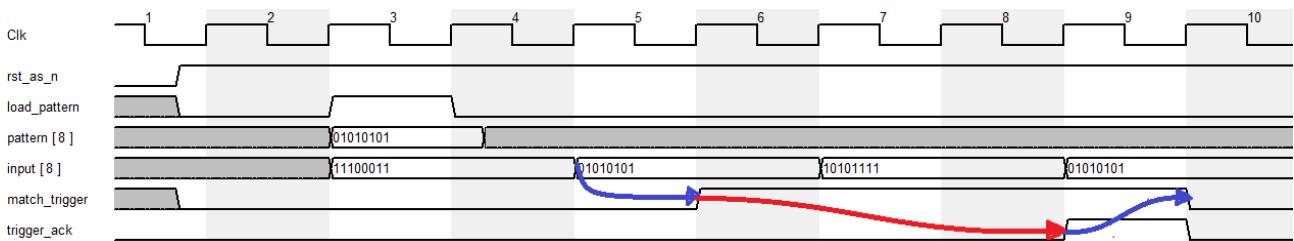
Di seguito possiamo osservare il palogramma della macchina a stati.



Abbiamo definito i seguenti stati:

- **S0_RESET**: stato in cui si resettano il registro ed il FLIP FLOP;
- **S1_IDLE**: stato in cui si attende l'arrivo del comando di abilitazione del trigger generator, in tal caso si salta in **S2_WAIT**, altrimenti si resta in **S1_IDLE**;
- **S2_WAIT**: stato in cui si attende che il confronto tra il pattern di trigger caricato e la configurazione degli input dia esito positivo. Se si verifica questa condizione si va in **S3_MATCH**, in caso contrario si resta in attesa in **S2_WAIT**;
- **S3_MATCH**: stato in cui si asserisce l'uscita **MATCH_TRIGGER**, indicante l'evento di trigger. L'uscita rimane alta finché non si riceve dall'esterno **TRIGGER_ACK**. Quindi si salta in **S4_RESET_MATCH** se tale segnale viene alzato a uno, altrimenti si resta nello stato corrente;
- **S4_RESET_MATCH**: si resetta il FLIP FLOP da cui esce il segnale **MATCH_TRIGGER**, riportandolo a zero. Da questo stato si torna direttamente in **S1_IDLE**

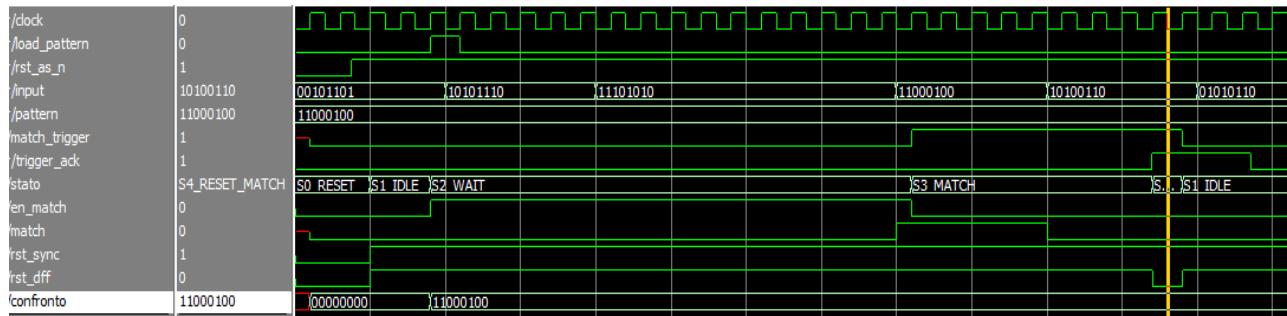
Di seguito possiamo osservare il timing diagram del circuito.



Il reset sincrono porta la macchina in uno stato noto di reset, quindi al primo colpo di clock utile si passa in **S1_IDLE** e si aspetta che venga asserito **LOAD_PATTERN** per caricare la sequenza di trigger. Dopo che questo segnale è stato asserito il circuito diventa sensibile agli 8 bit di input ed è in grado di rilevare la corrispondenza tra questi ed il pattern caricato. Quando viene rilevata la condizione di egualanza tra pattern e input viene asserito il segnale **MATCH_TRIGGER**, fino al momento in cui non viene ricevuto **TRIGGER_ACK**, che viene usato come segnale di acknowledge, quindi viene negato.

Di seguito i risultati della simulazione modelsim. Dove c'è il puntatore giallo è possibile vedere lo stato **S4_RESET_MATCH**, in cui si arriva quando **TRIGGER_ACK** viene asserito.

È possibile osservare anche i segnali di controllo del datapath visto sopra e lo stato in cui vengono commutati.



Come si può notare, il FLIP FLOP da cui esce **MATCH_TRIGGER** è abilitato per tutto il tempo trascorso dal caricamento del pattern di trigger, fino alla prima volta che si verifica l'evento di trigger. Il segnale **TRIGGER_ACK**, normalmente è pilotato dalla memory interface, ma in questo caso è stato pilotato a mano modificando la forma d'onda in ingresso al circuito.

▪ Memory interface

La memory interface ha il compito di gestire la scrittura dei dati in uscita dal sampler (8bit di dato e 8 bit di informazione sui glitch) e la lettura degli stessi della memoria SRAM_61LV25616AL montata sulla DE2. È una memoria RAM statica asincrona, quindi non è presente nessun tipo di clock e dovranno essere soddisfatti tutti i tempi dichiarati dal costruttore nei timing diagram. La SRAM è vista come un buffer circolare, una volta iniziata la scrittura si prosegue in modo sequenziale, per variare il modo in cui essa viene letta occorre cambiare l'indirizzo di partenza, salvato in un registro particolare, dove viene memorizzato l'indirizzo in corrispondenza dell'evento di trigger. La memory interface è in grado di fermarsi in maniera autonoma dopo che sono stati scritti 128k campioni nella memoria dopo il trigger, come da specifiche, segnalando tale condizione alla Main FSM.

Le richieste di scrittura e lettura vengono fatte dalla Main FSM, il segnale di trigger viene inviato direttamente dal trigger generator, i 16 bit di input provengono dal sampler (al cui stesso BUS locale è collegato il trigger generator) e i 16bit di dato letti vengono inviati alla PC interface attraverso un altro BUS locale.

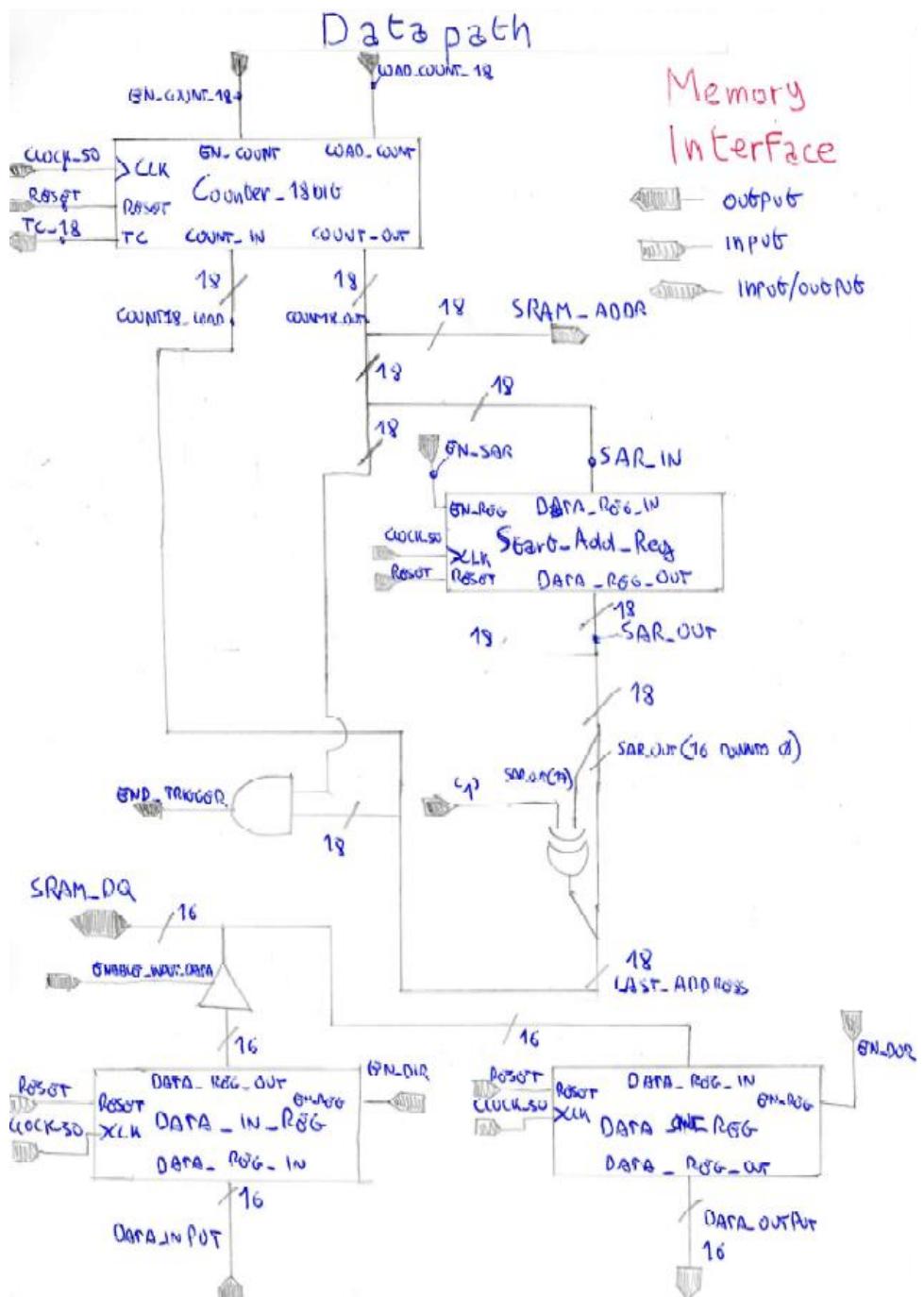
○ **Datapath**

Il datapath della memory interface comprende 2 registri da 16 bit per la memorizzazione dei dati in ingresso e uscita, 16 porte tri-state collegate all'uscita del registro contenente i dati di ingresso, per evitare conflitti in fase di lettura (anche se non si vede il tri-state è comandato da una porta AND collegata ad un controllo della FSM della mem. Int. ed al segnale **SRAM_OE_N**, come ulteriore protezione contro i conflitti), un registro da 18 bit per salvare l'indirizzo in corrispondenza dell'evento di trigger, un contatore da 18 bit utilizzato per indirizzare la memoria in modo sequenziale, una porta EXOR e una serie di porte AND usate per rilevare l'indirizzo finale del buffer circolare ed infine la memoria SRAM, non presente nel datapath in quanto viene vista come un blocco esterno ad esso, comandata anch'essa dalla FSM della memory interface.

L'uscita del contatore da 18 bit è collegata al BUS indirizzi della SRAM, ovvero **SRAM_ADDR**. La porta tri-state è collegata sull'uscita della memory interface, sul BUS dati locale che collega i registri di dato e uscita alla SRAM. Tale utilizzo del tri-state è perfettamente legittimo in quanto non verrebbe utilizzato on-chip, ma a livello di scheda.

La funzione dei registri di dato è per svincolarsi dal timing della SRAM: il registro dei dati di uscita si giustifica pensando che una volta letto il dato è possibile averlo disponibile "per sempre", senza dover comandare continuamente la SRAM, il registro di ingresso invece serve per mantenere stabile il dato da scrivere in memoria senza preoccuparsi di quello che succede sul BUS (comunque controllato), evitando conflitti che potrebbero inficiare il normale funzionamento del circuito.

L'uscita del contatore da 18 bit è collegata anche ad un registro chiamato SAR (start address register), la cui uscita a sua volta viene collegata ad una porta AND. Notare come tra la porta AND e l'uscita del SAR ci sia una porta EXOR collegata solamente al MSB dei 18bit in uscita dal SAR, in cui uno dei due ingressi è sempre a '1'. Lo scopo di questa porta è sommare 128k al valore dell'indirizzo, in modo che sia possibile verificare successivamente, tramite la porta AND, quando sono stati scritti 128k campioni dopo che si è verificata la condizione di trigger, ovvero quando è stata scritta mezza memoria. Il segnale così generato, **END_TRIGGER**, viene sentito se e solo se è attivo il segnale di controllo **read_flag**, asserito dopo che è stato rilevato lo start (ovvero condizione di trigger). È presente un ulteriore segnale chiamato **LAST_ADDRESS_OUT**, dato dall'AND tra **END_TRIGGER**. In seguito allo sviluppo della Main Fsm abbiamo aggiunto infatti un segnale di uscita, pilotato da **END_TRIGGER**, mandato in AND con un segnale di comando della control unit, **READ_FLAG_OUT**, attivo dopo che si è verificato l'evento di trigger. L'utilizzo di questo segnale sarà spiegato meglio nella descrizione della control unit della Main Fsm ed è necessario per il corretto funzionamento dell'analizzatore alla massima velocità di campionamento consentita.



○ FSM e timing diagram

Il compito principale della FSM della memory interface è gestire il timing della scrittura e lettura della memoria SRAM. Come prima cosa consideriamo il tempo massimo entro cui devono avvenire il ciclo di lettura e quello di scrittura. Non esistono vincoli sulla durata della lettura, non esiste un tempo massimo entro il quale dobbiamo fornire in uscita il dato, che poi dovrà essere codificato e inviato sulla linea seriale, un processo di durata molto più grande di quella di un ciclo di lettura.

Diverso è il discorso per la scrittura: la massima frequenza di campionamento del sampler è 10 MHz, questo vuol dire che il minimo numero di cicli del clock tra una scrittura e l'altra è 5. Bisognerà quindi essere in grado di effettuare correttamente tutte le operazioni di scrittura in meno di 5 colpi di clock, ovvero 100 ns. Per il momento ignoriamo eventuali problemi di sincronizzazione col resto del circuito, che affronteremo progettando la Main Fsm dell'analizzatore logico.

L'ingresso counter18_load è collegato ma non viene mai abilitato, il segnale LOAD_COUNT_18 è infatti sempre fisso a 0.

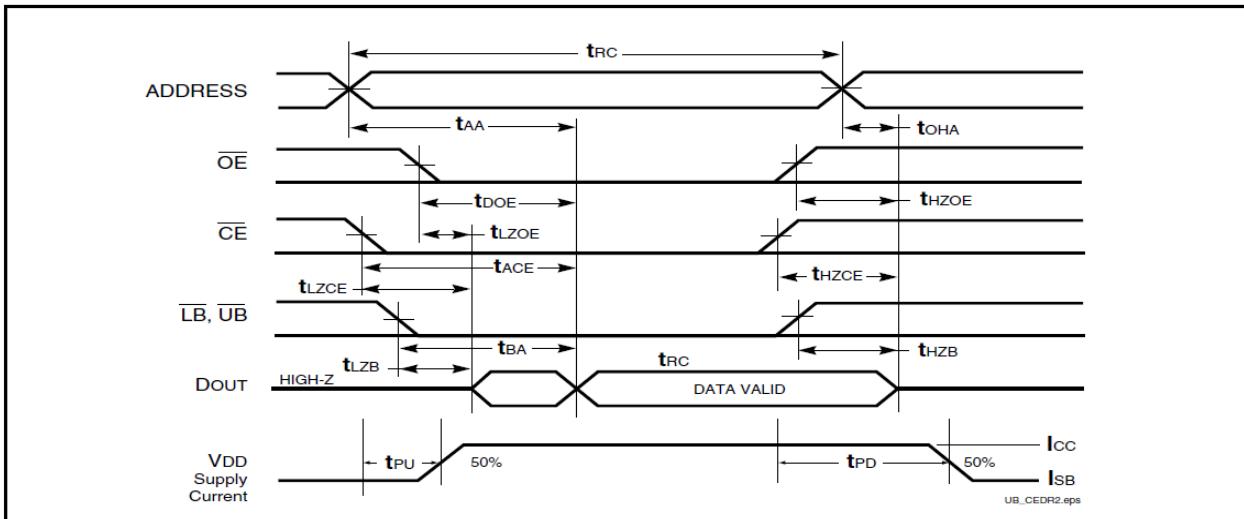
DATA_INPUT è collegato all'uscita del sampler, **DATA_OUTPUT** è collegato all'ingresso della PC interface.

Il clock del datapath è **CLOCK_50**, ovvero il clock a 50MHz derivato dalla DE2. Tutti gli elementi in questo blocco lavorano utilizzando questo segnale di clock, ad eccezione della SRAM che è asincrona.

Analizziamo quindi il timing diagram in scrittura e lettura della SRAM: notiamo come esistano diversi timing della scrittura, tutti più o meno con gli stessi tempi. Prendiamo come riferimento due timing che descrivono al meglio la nostra condizione di utilizzo della memoria.

Iniziamo considerando la lettura:

READ CYCLE NO. 2^(1,3)

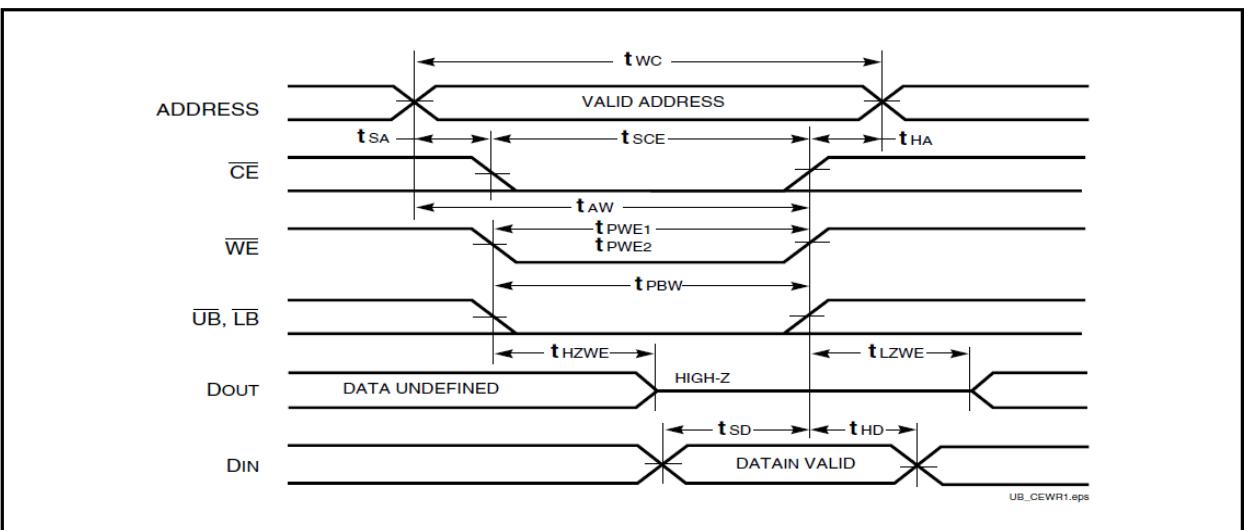


Decidiamo di negare CE, LB e UB insieme nello stesso ciclo di clock e poi OE nel ciclo successivo. Quindi manteniamo questi segnali bassi per un ulteriore ciclo, durante il quale i dati in un registro, ed infine li alziamo tutti insieme ed abilitiamo il contatore degli indirizzi per incrementarlo dopo un tempo superiore a t_{OHA} , t_{HZOE} , t_{HZCE} e t_{HZB} , in modo da evitare conflitti.

Il BUS indirizzi rimane stabile per il tempo indicato e il dato viene campionato dopo un tempo tale da considerarlo valido. Durante la lettura il tri-state visto nel datapath dovrà essere in alta impedenza per evitare conflitti.

Passiamo adesso alla scrittura:

WRITE CYCLE NO. 1 (\overline{CE} Controlled, \overline{OE} is HIGH or LOW) ⁽¹⁾



Dal timing diagram decidiamo di abbassare CE, UB, LB e WE tutti insieme, mantenerli bassi per due colpi di clock e poi tirarli su. I dati ed indirizzo andranno mantenuti stabili per un colpo di clock dopo che questi segnali sono stati portati alti. Inoltre dopo la richiesta di scrittura ci servirà un ciclo di clock per salvare i dati in uscita dal sampler nel registro dei dati in ingresso. Decidiamo di prendere un ciclo di clock per eseguire

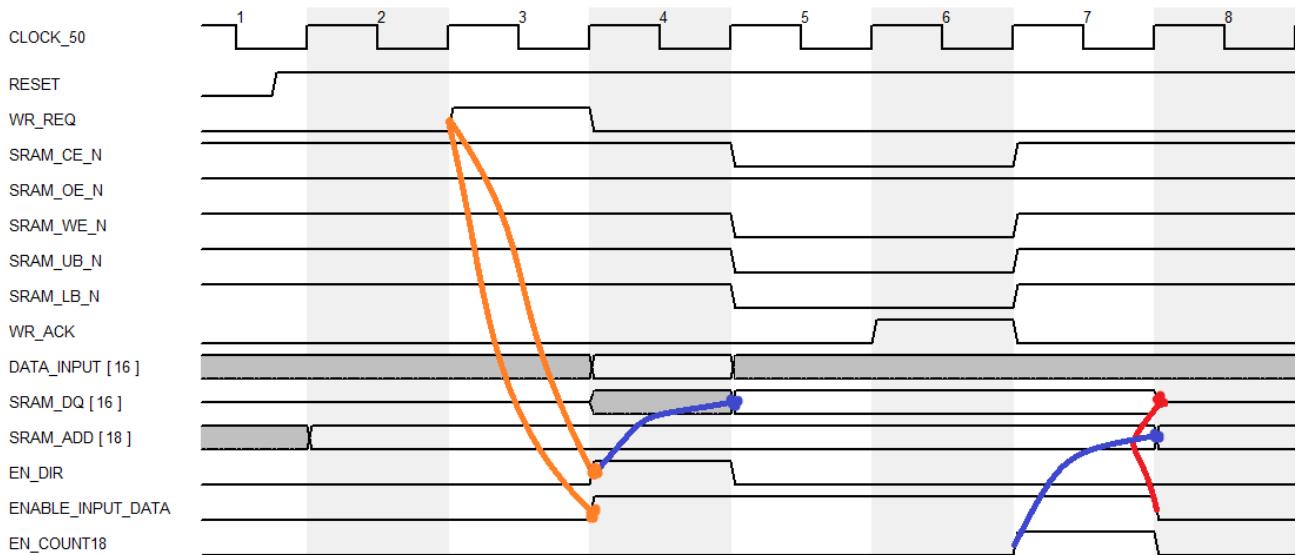
esclusivamente questa operazione. Il numero di cicli di clock totali è quindi 4 e siamo dunque sotto il tetto dei 5 colpi di clock definito prima.

Durante la scrittura il tri-state sarà sempre abilitato dalla FSM, siccome OE è alto e siamo in scrittura, non possono verificarsi conflitti.

Decidiamo anche di inserire tre segnali di acknowledge, pilotati dalla FSM per semplificare l'interfaccia col resto del circuito: **START_ACK**, **READ_ACK** e **WRITE_ACK**. Questi tre segnali indicano rispettivamente la ricezione corretta del segnale di start e salvataggio indirizzo iniziale, la ricezione del segnale di richiesta di lettura ed inizio della routine di lettura di tutta la memoria, la ricezione del segnale di richiesta di lettura e salvataggio del dato in memoria.

Abbiamo aggiunto anche un segnale di strobe, chiamato **OUTPUT_VALID**, che indica alla Main Fsm quando è presente un dato valido all'uscita della memory interface, inoltre affinchè la memoria esegua un nuovo ciclo di lettura, per evitare che il dato venga perso senza essere stato elaborato dal resto del circuito, è richiesto l'invio di un segnale **READ_READY**, che comunica che il dato è stato elaborato e può essere letto uno nuovo. Questo segnale è necessario che venga asserivo a partire dalla fine del primo ciclo di lettura.

Possiamo adesso definire il timing in lettura e scrittura e partiamo proprio da quest'ultimo:

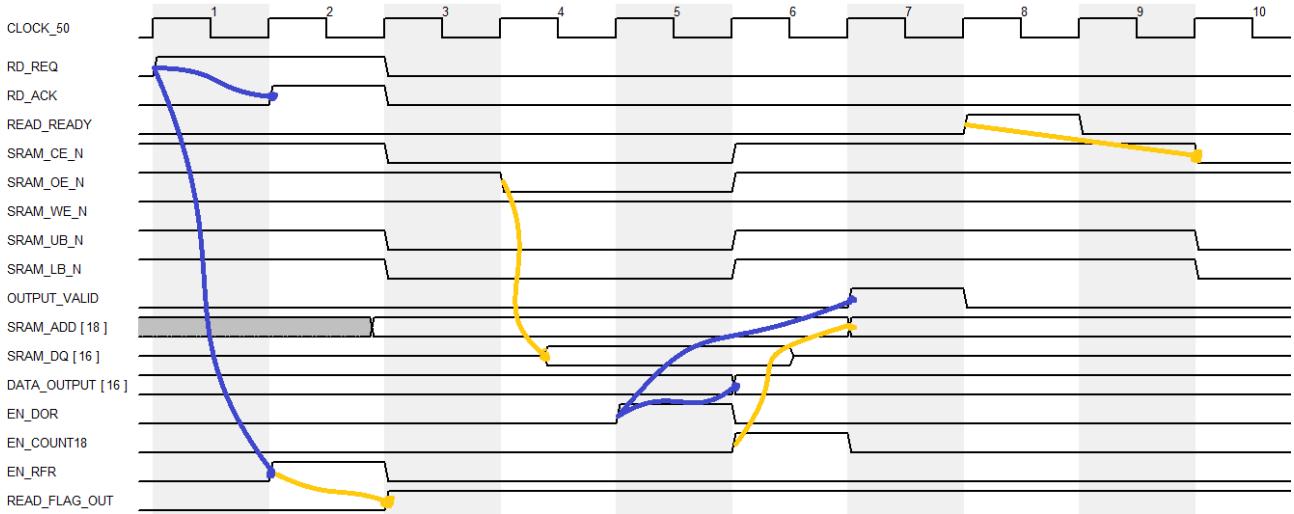


Come si può vedere il **WR_REQ**, quando viene asserito, implica che ci sia un dato valido in ingresso alla memory interface, che viene campionato portando **EN_DIR** e si ritroverà al colpo di clock successivo in ingresso alla SRAM, sul pin **SRAM_DQ**. Insieme al registro si abilita anche il tri-state con **ENABLE_INPUT_DATA**.

La richiesta di scrittura causa anche la negazione simultanea di tutti i segnali di controllo della SRAM coinvolti in questo processo, che è comandato da **SRAM_CE_N** (è questo segnale che causa l'effettiva scrittura dei dati in memoria sul suo fronte di salita).

Infine il fronte di salita su **SRAM_CE_N** comporta che **SRAM_DQ** vada di nuovo in alta impedenza e quindi la fine del ciclo di scrittura vero e proprio. Contemporaneamente si aggiorna l'indirizzo della SRAM abilitando il contatore con **EN_COUNT_18** e si manda in alta impedenza il tri-state negando **ENABLE_INPUT_DATA**.

Consideriamo adesso il timing della lettura:



Nel timing abbiamo supposto che al primo colpo di clock ci sia stato un fronte su **RD_REQ**, che comporta l'asserimento del segnale **RD_ACK**, ad indicare la ricezione del comando di richiesta di lettura della memoria. Nello stesso ciclo si abilita il flip flop che da in uscita il segnale **READ_FLAG_OUT**, usato per controllare lo stato della lettura. che a sua volta indica l'inizio della routine di lettura, con la negazione di **SRAM_CE_N**, **SRAM_UB_N** e **SRAM_LB_N**, inoltre adesso il valore dell'indirizzo **SRAM_ADD** diventa significativo ed è importante che venga mantenuto stabile per tutta la durata del processo.

Notare come **SRAM_DQ** non sia più in alta impedenza dopo che viene negato **SRAM_OE_N**. In verità il valore di **SRAM_DQ** non è subito valido, c'è un tempo di qualche ns in cui esso è completamente aleatorio e variabile, tuttavia ai fini del nostro timing, considerato che stiamo lavorando a 20ns e il dato viene campionato quasi un colpo di clock e mezzo dopo dal registro di uscita, possiamo ignorarlo.

Infine la salita del chip select e dell'output enable comportano la fine del ciclo di scrittura, l'asserimento di **OUTPUT_VALID** e il passaggio in alta impedenza di **SRAM_DQ**.

Il rilevamento del segnale **READ_READY**, comandato dalla Main Fsm, indica alla memory interface di eseguire un nuovo ciclo di lettura.

Adesso analizziamo la FSM, essa è composta da 15 stati di seguito riportati e descritti:

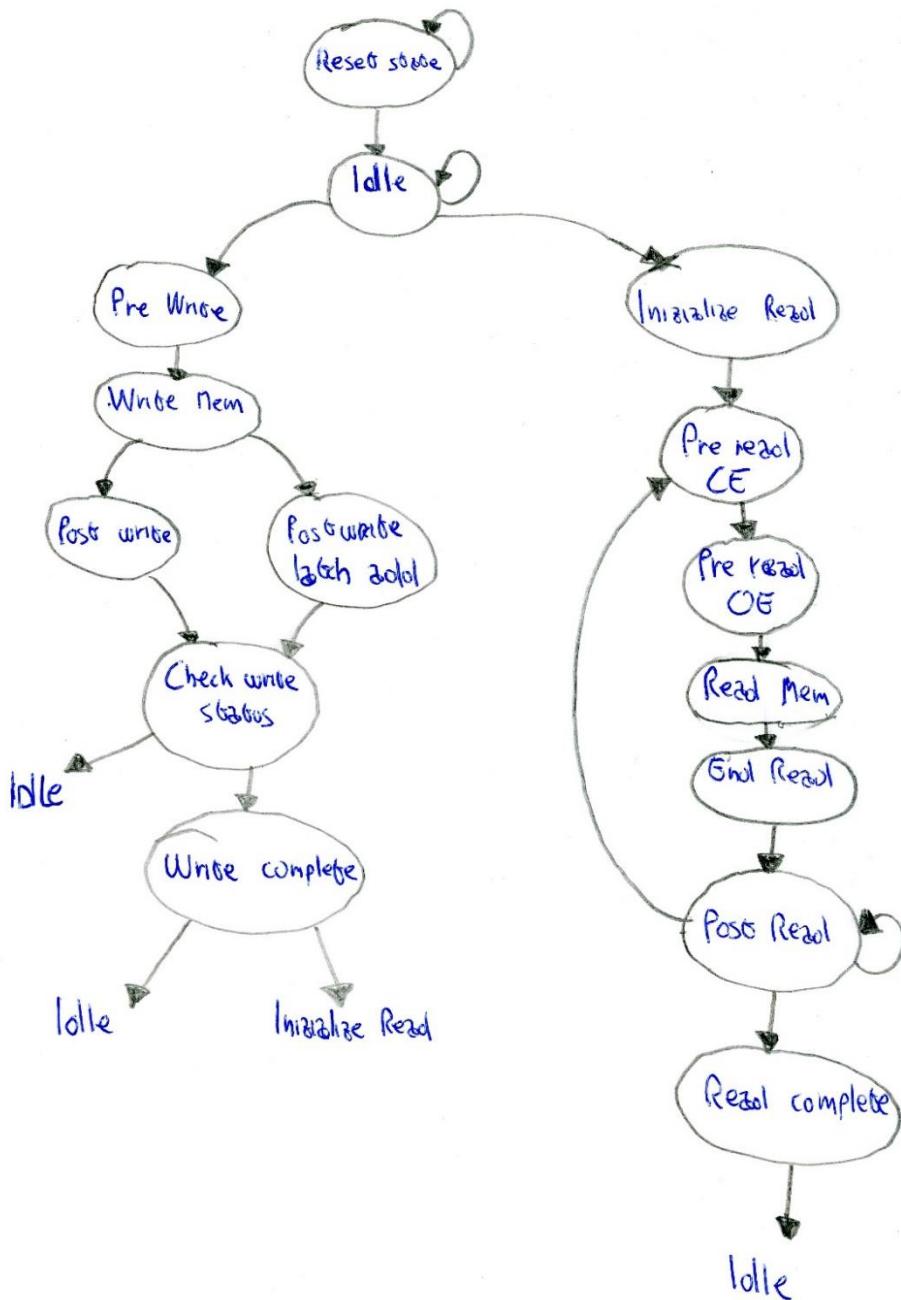
- **RESET_STATE**: stato in cui il circuito viene portato in modo sincrono quando si nega il **RESET** esterno, oppure in caso in cui esso finisce in uno stato proibito;
- **IDLE**: stato in cui la macchina ritorna una volta eseguite le operazioni di scrittura e lettura. Si controlla se sono arrivate richieste di lettura o scrittura e in caso affermativo si salta in **INIZIALIZE_READ** o **PRE_WRITE**;
- **PRE_WRITE**: stato in cui viene abilitato il registro **data_in_reg** (DIR) per salvare i dati provenienti dal sampler. Da qui si salta in **WRITE_MEM**;
- **WRITE_MEM**: stato in cui si comanda la scrittura della memoria SRAM secondo il timing visto sopra, si controlla inoltre se è stato ricevuto il segnale **START**, indicante l'evento di trigger. In tal caso si salta in **POST_WRITE_LACTH_ADD**, altrimenti si va in **POST_WRITE**;
- **POST_WRITE**: stato con gli stessi comandi di **WRITE_MEM**, in più viene asserito **WR_ACK** che indica che la richiesta di scrittura è andata a buon fine e verrà completata nel colpo di clock successivo. Lo stato successivo è **CHECK_WRITE_STATUS**;

- **POST_WRITE_LATCH_ADD**: stato identico a **POST_WRITE**, con l'eccezione che viene salvato l'indirizzo corrente nel registro **start_address_register** (SAR) e viene asserito un flag indicante la nuova variabile da controllare, ovvero se siano stati già salvati 128k campioni dopo la condizione di trigger. Come in **POST_WRITE**, si salta direttamente in **CHECK_WRITE_STATUS**;
- **CHECK_WRITE_STATUS**: stato in cui si abilita il segnale di incremento del contatore e si controlla la condizione attuale della macchina, ovvero se sono stati scritti 128k campioni dopo l'evento di trigger, in tal caso si salta nello stato **WRITE_COMPLETE**, altrimenti si torna in **IDLE** oppure si può eseguire una nuova scrittura in serie;
- **WRITE_COMPLETE**: stato in cui va la macchina dopo che la memoria è stata interamente scritta secondo specifiche, si asserisce **MEM_WRITTEN** e poi da qui si può andare in **IDLE** oppure in **INITIALIZE_READ**;
- **INITIALIZE_READ**: in questo stato si attiva il segnale **RD_ACK**, dopodiché si inizia il processo di lettura della memoria intera. Lo stato successivo è **PRE_READ_CE**;
- **PRE_READ_CE**: come visto precedentemente nel timing diagram, in questo stato si comandano dei segnali della SRAM, quindi si passa in **PRE_READ_OE**;
- **PRE_READ_OE**: si abbassa l'output enable e si salta in **READ_MEM**;
- **READ_MEM**: in questo stato si mantengono stabili i segnali visti sopra e si abilita il registro dei dati di uscita **data_output_register** (DOR). Da qui si va in **END_READ**;
- **END_READ**: stato in cui si alzano insieme tutti i segnali di controllo della SRAM e si abilita il segnale di incremento del contatore, che cambierà nel colpo di clock successivo, rispettando dunque le specifiche. Lo stato successivo è **POST_READ**;
- **POST_READ**: adesso la lettura del dato salvato in memoria è stata completata, si asserisce il segnale **OUTPUT_VALID** e si aspetta un segnale di acknowledge da parte della Main FSM, **READ_READY**, che autorizza la memory interface a caricare la word successiva. Si controlla prima se sono stati già lette tutte le locazioni della memoria, in tal caso si salta in **READ_COMPLETE**, dopodiché si controlla se **READ_READY** è stato asserito. In caso affermativo si ritorna in **PRE_READ_CE**, altrimenti si resta in **POST_READ**;
- **READ_COMPLETE**: in questo stato si asserisce il segnale **MEM_READ**, indicante che è stata completata la lettura della memoria, quindi si va in **IDLE**, in attesa di un nuovo comando di lettura, oppure di scrittura. Va fatto notare che per tutto il processo di lettura della memoria il circuito è insensibile al comando **WR_REQ**. Dopo che la lettura è stata eseguita interamente è possibile sentire di nuovo le richieste di scrittura.

Ricapitolando: la scrittura è composta da 4 stati e la lettura da 5. Siamo sicuri che non esistono vincoli temporali sull'esecuzione delle letture, che gestiamo tramite un meccanismo di strobe e acknowledge.

Affronteremo eventuali criticità rimaste nella fase di scrittura nella progettazione della Main Fsm.

Di seguito riportiamo il pallogramma della memory interface:



Riportiamo adesso gli esiti delle simulazioni eseguite con un testbench su MODELSIM in cui si verifica solamente la corretta gestione dei segnali di controllo, poiché il corretto interfacciamento con la SRAM è verificabile solamente con la DE2. Il test vero e proprio in laboratorio è stato fatto in seguito con successo.

Aggiungiamo per correttezza che il testbench dell'analizzatore logico completo è stato fatto utilizzando una SRAM fittizia, che qui non è stata usata.

Si possono notare le variazioni dei principali segnali di controllo descritti nelle righe precedenti.

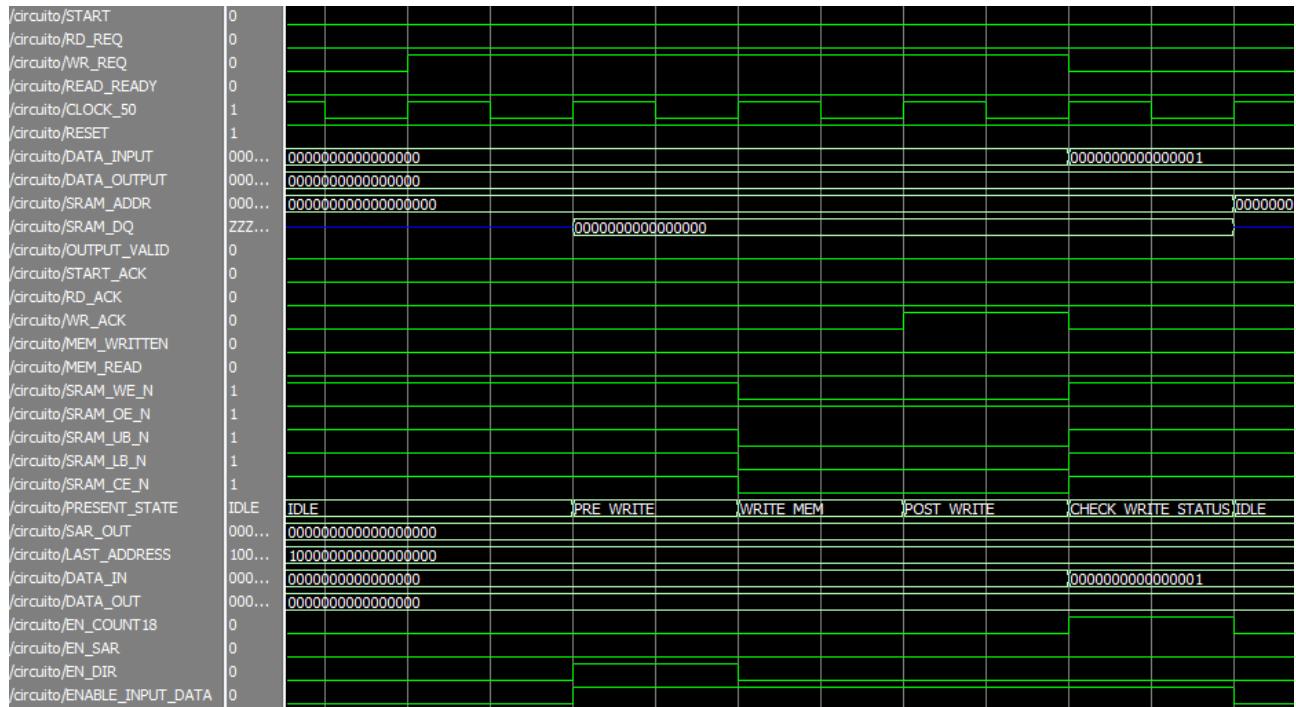


Figura 1 Ciclo di scrittura

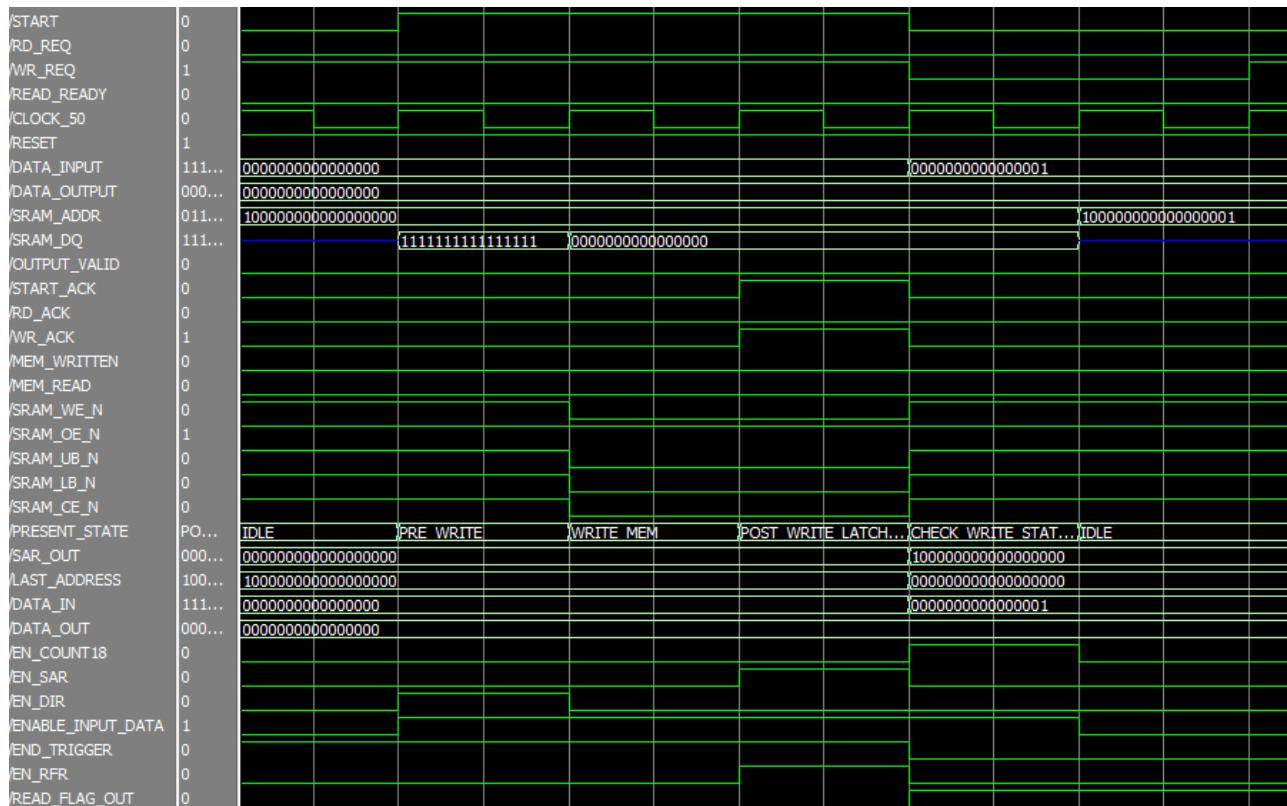


Figura 2 Condizione di trigger e salvataggio indirizzo di start

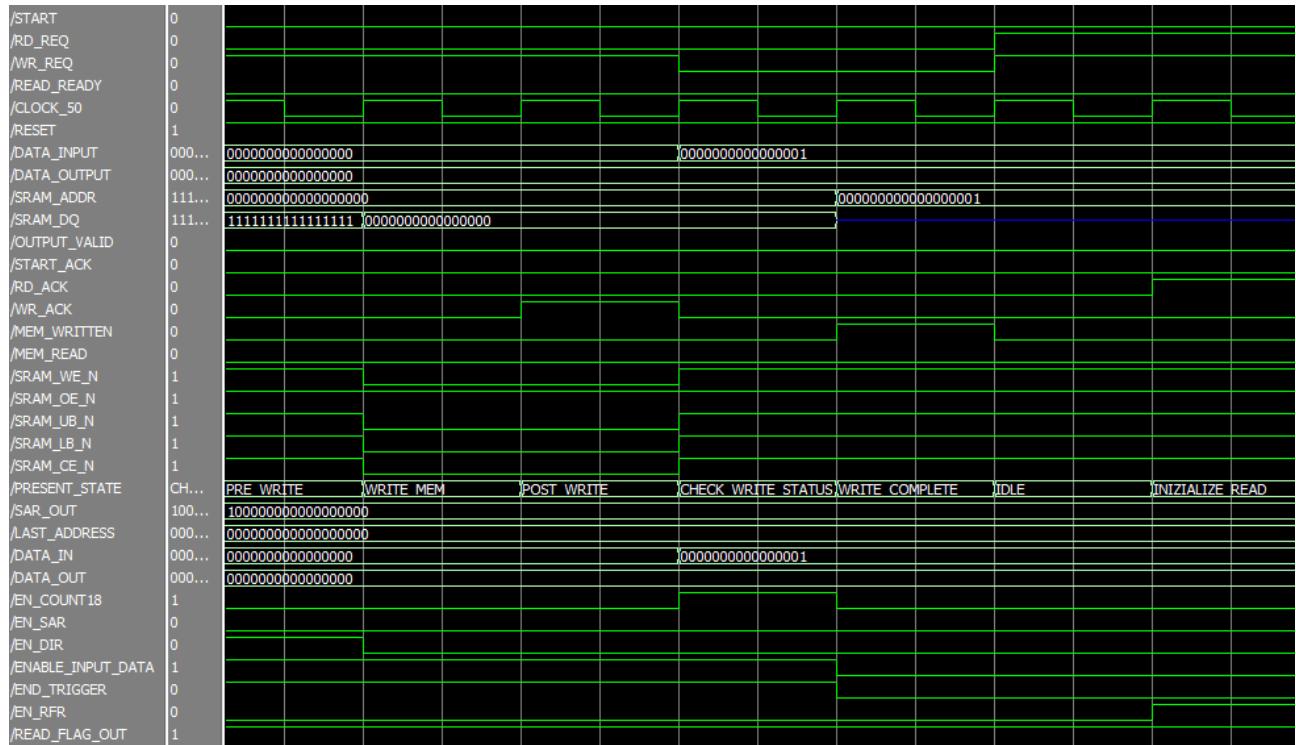


Figura 3 Completamento della scrittura ed inizio prima lettura

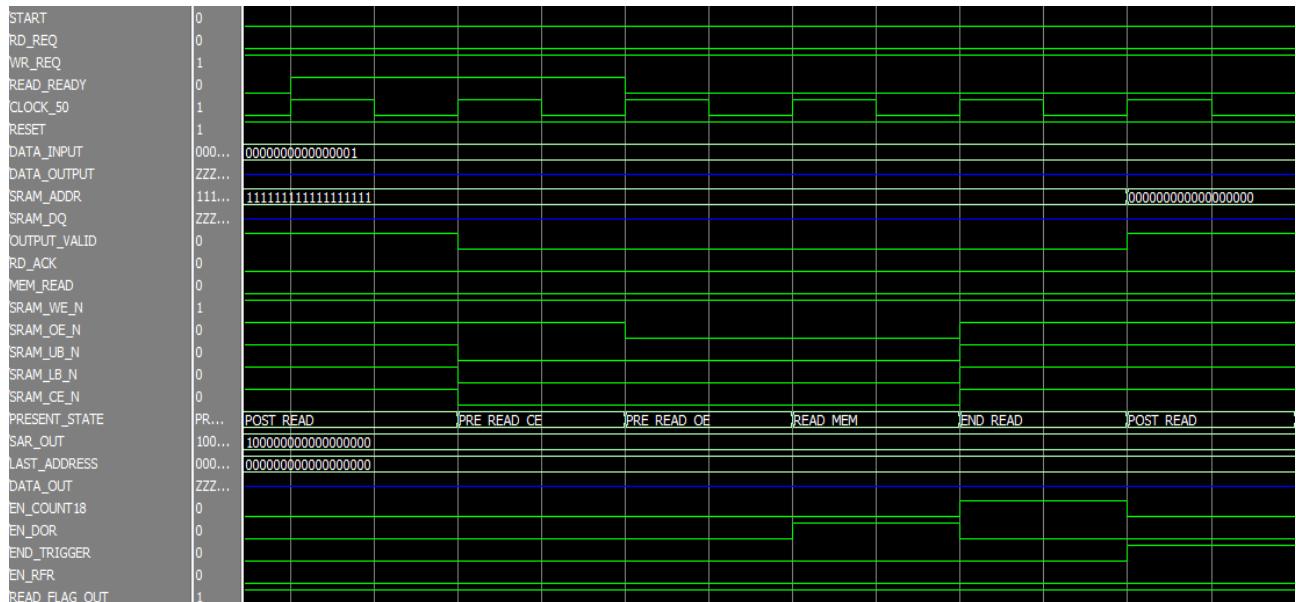


Figura 3 Ciclo di lettura

Notare come in lettura il tri-state che separa l'uscita del data_out_register e l'ingresso della SRAM, sia sempre in alta impedenza.

Il segnale **READ_FLAG_OUT** è asserito quando si è verificata la condizione di trigger ed è usato per sentire il segnale **END_TRIGGER**. Questi due segnali vengono usati per capire quando è stato eseguito un ciclo completo della memoria, ovvero quando sono stati scritti 128k campioni dopo il trigger (più i 128k antecedenti all'evento) e quando tutto il contenuto della memoria è stato letto.

▪ Introduzione Pc interface

La PC interface è un blocco composto di una serie di unità più piccole. Il datapath della PC interface è composto da 4 unità secondarie: codifica ASCII, decodifica ASCII, UART TX e UART RX.

La funzione svolta dalla PC interface è complessa. Deve gestire la ricezione dei comandi dalla linea seriale RS-232, interpretarli e comunicarli alla Main FSM dell'analizzatore logico, in modo che questa possa agire di conseguenza. Deve codificare i dati in uscita dalla memoria, interpretando correttamente i bit di dato e rispettivi bit di glitch, in modo che possano essere trasmessi sulla linea seriale sotto forma di caratteri ASCII. In più deve comandare correttamente la UART, composta dal trasmettitore e dal ricevitore, che operano con un baud rate di 115200 con tolleranza del 3%.

Prima di descrivere datapath, pallogramma e timing della PC interface e visualizzare l'esito delle simulazioni, analizziamo i componenti secondari uno ad uno, secondo la filosofia del divide et impera. Spezziamo quindi l'unità principale in una serie di sotto-blocchi, relativamente indipendenti tra loro, in modo da semplificare la ricerca della soluzione finale.

Questa unità è molto critica perché dal suo corretto funzionamento dipendono tutti gli altri componenti del circuito, inoltre essendo composta da molti parti diverse, diventa fondamentale lo studio del timing tra le varie interfacce, senza tralasciare quella con la Main FSM.

La PC interface comanda alcuni registri interni dei blocchi da cui essa è composta, come la codifica e decodifica ASCII. Questo ci ha permesso di spostare la complessità e la gestione dell'interfacciamento all'esterno, semplificandole di molto. Abbiamo infatti ridotto il numero degli stati nei blocchi a livello più basso.

1. Codifica ASCII

Questo blocco deve codificare i dati in ingresso in caratteri ASCII, quindi ogni coppia di bit in ingresso, dato e glitch viene codificata su 8bit che verranno poi inviati dalla UART TX sulla linea seriale. Deve anche generare i caratteri O e K, usati per comunicare al PC rispettivamente la ricezione corretta del comando inviato o l'errata ricezione/decodifica dello stesso, K viene inviato anche in caso di comando non riconosciuto. Bisogna anche inviare \n, cioè line feed, per poter visualizzare su una singola riga i valori degli 8 ingressi.

○ **Datapath**

L'ingresso di questo blocco è l'uscita della memory interface su cui viaggeranno i 16 bit di dato e glitch. Per come abbiamo progettato, i dati sono organizzati come dato_sample1, glitch_sample1, dato_sample2, glitch_sample2, dato_sample3, glitch_sample3, ..., dato_sample8, glitch_sample8. Nel momento in cui il dato è disponibile e stabile da parte della memory interface, il blocco di codifica riceve un segnale di load, chiamato **LOAD_DATI**, che permette di caricare questi 16 bit in un registro interno alla codifica ASCII .

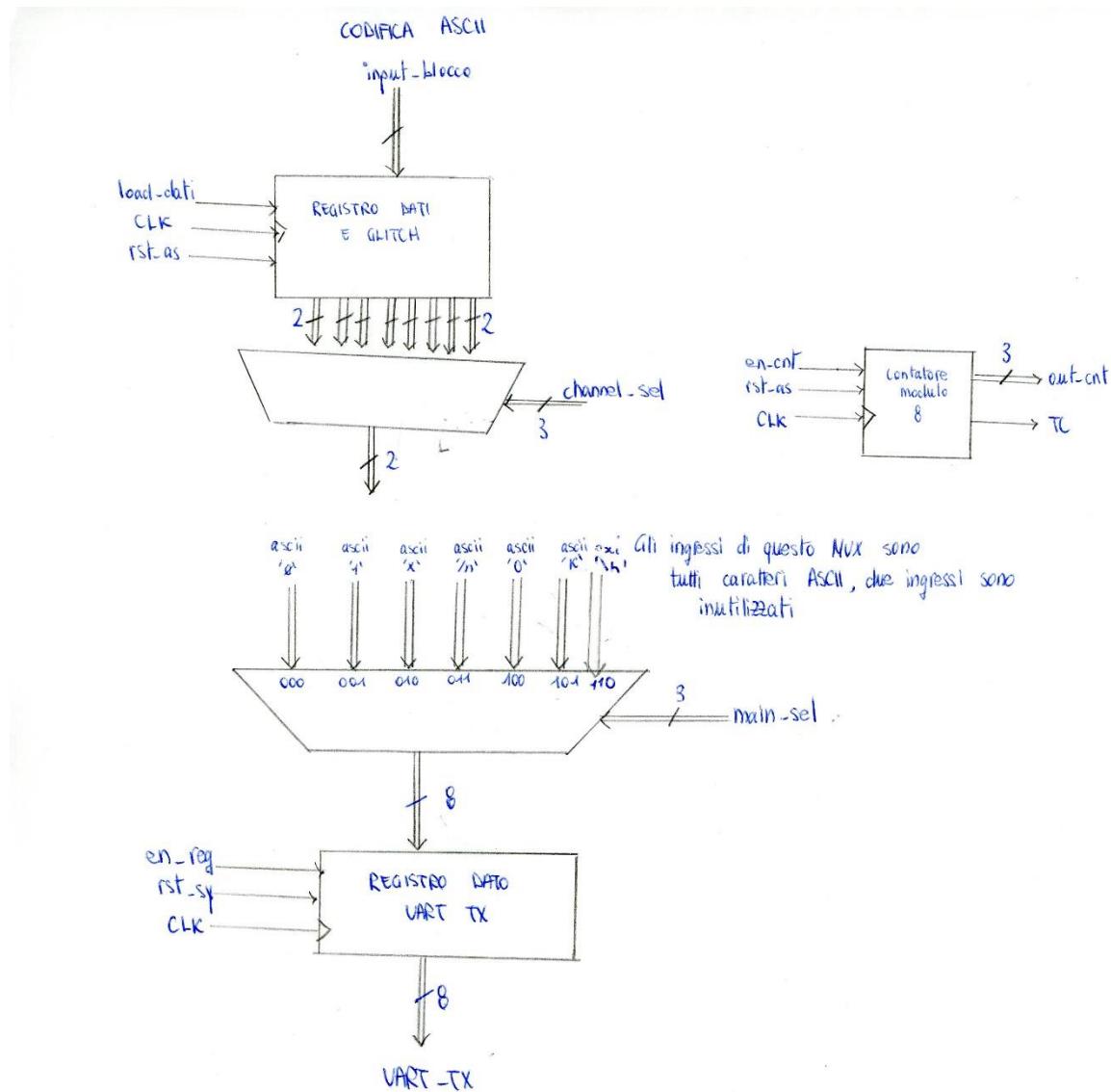
A questo punto si dovranno interpretare separatamente i bit di ogni sampler e il pacchetto di dati da 8 bit contenente la codifica ASCII del dato da inviare verrà poi mandato al trasmettitore della uart, che si occuperà poi di inviarlo sulla linea seriale.

Descriveremo ora in che abbiamo gestito la codifica: l'uscita del registro in cui sono stati salvati i 16 bit in ingresso viene divisa in otto segnali di due bit l'uno, ognuno contenente l'informazione di dato e glitch per ogni canale. Questi otto segnali rappresentanti i dati inviati dagli otto sampler vengono gestiti tramite un multiplexer il cui selettore è l'uscita di un contatore che viene abilitato ogni volta la uart_tx invia il segnale **DONE_TX** che indica il completamento dell'operazione di invio del dato precedente.

Il dato su 8 bit da inviare, come detto precedentemente, potrà essere o 0 o 1 o X in ASCII: i dati 0 e 1 vengono inviati quando non c'è stato nessun glitch, la X viene inviata quando ho avuto un glitch, qualunque valore di dato fosse stato ricevuto. Oltre a questi dati possono essere inviati i comandi \n (feed line), \r, K e O. I primi due vengono inviati quando è arrivato il TC del contatore che indica che gli 8 canali sono stati codificati e inviati. Per quanto riguarda K e O il discorso è differente: K viene inviato quando è stato decodificato un comando errato, per cui il blocco di decodifica ASCII ha restituito in uscita un segnale di **FAIL**; il segnale O indica invece l'invio corretto di un comando.

Queste sette possibilità di codice ASCII inviabile vengono gestiti da un multiplexer, comandato dalla PC interface tramite **MAIN_SEL**, la cui uscita viene salvata in un registro che conterrà il dato da inviare alla uart tx, anch'esso abilitato dalla PC interface tramite **LOAD_COMANDO**. Quest'ultima, interpretando i due bit contenenti bit e dato inviati dalla codifica setterà il multiplexer su una certa porta; nel momento in cui il contatore dei canali restituisce in uscita il segnale TC allora verranno inviati in sequenza i dati in ASCII che rappresentano \n e \r. Per quanto riguarda i comandi K e O, dopo aver decodificato un comando inviato dall'utente, il selettore verrà impostato su una o l'altra porta in base alla correttezza del comando in modo da inviare su seriale questa risposta.

Il blocco di codifica restituisce in uscita un segnale, **DONE_CODIFICA**, che indica il fatto che un dato è pronto nel registro finale e può essere campionato dalla uart. Per eseguire una successiva codifica questo blocco attende l'invio completo del dato precedente.



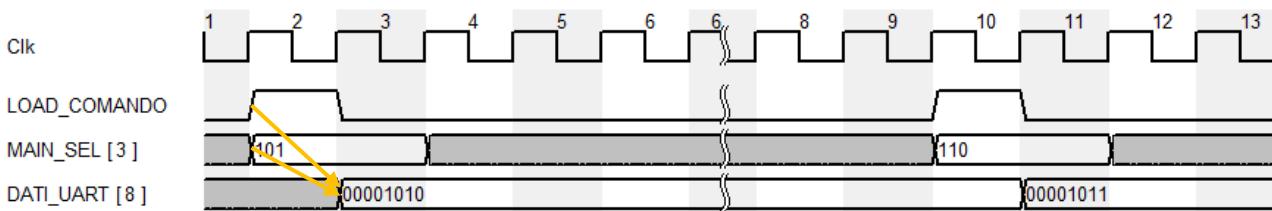
Il contatore è comandato dalla PC interface tramite il segnale **EN_CNT che**, quando viene portato a 1, indica l'invio dell'ultimo carattere relativo all'ottavo canale del sampler. Il valore di uscita del contatore ha quindi la funzione di indicare quale carattere di stato del sampler (1, 0 oppure x) viene inviato sulla linea seriale.

- **Timing diagram.**

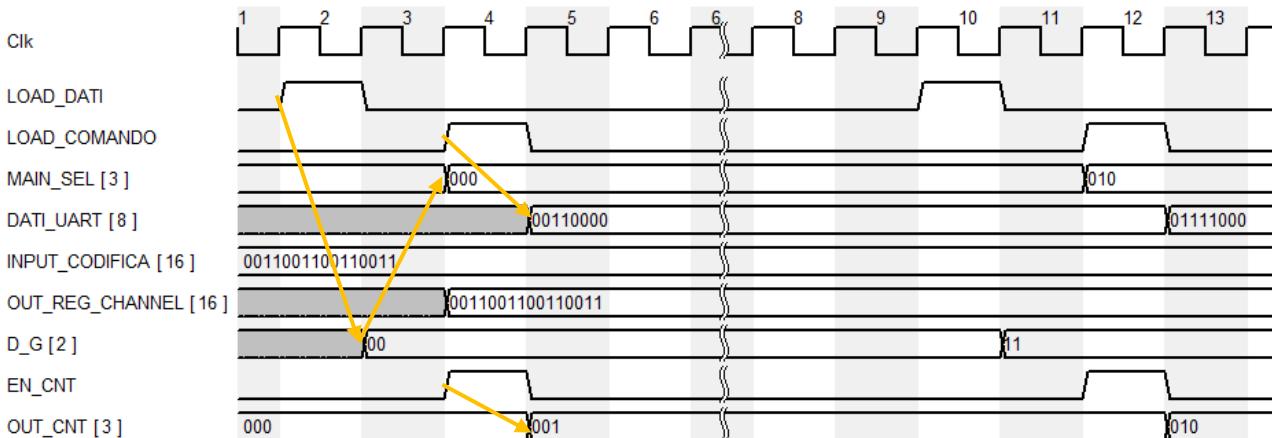
Analizziamo adesso i timing per capire come gestire la codifica ASCII dalla PC interface, considerando che alcuni segnali saranno pilotati da questo blocco ed altri da un'altra control unit a livello più alto.

Partiamo dal caso in cui vogliamo inviare un carattere, ad esempio \n in seguito all'invio di una O di una K.

Bisogna asserire **LOAD_COMANDO** dall'esterno e pilotare il segnale **MAIN_SEL** in modo da selezionare il carattere desiderato. Dopo due colpi di clock il dato codificato potrà essere campionato dal registro della uart.



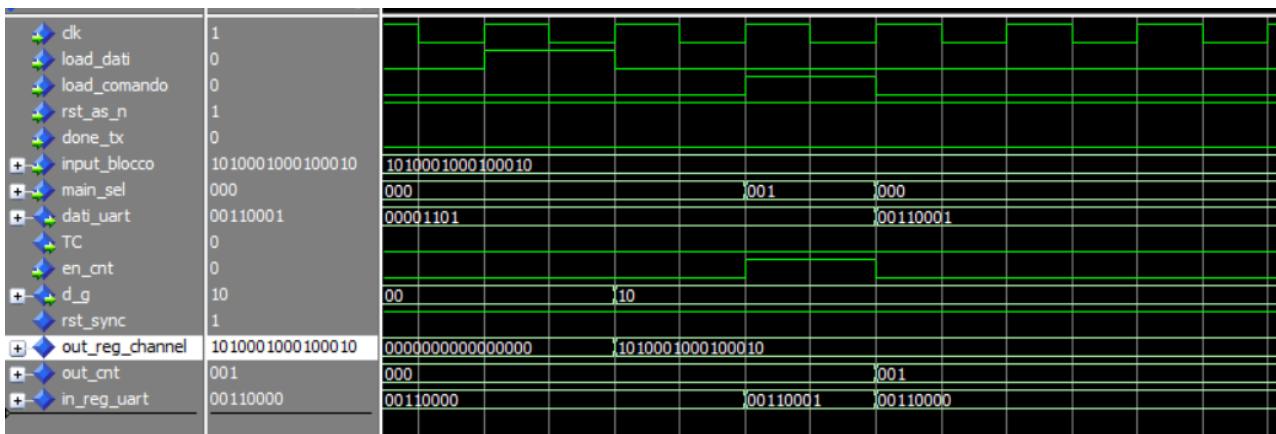
Consideriamo adesso il caso in cui si debba inviare una coppia di bit di dato e glitch, che andrà codificata in un 1, 0 oppure x.



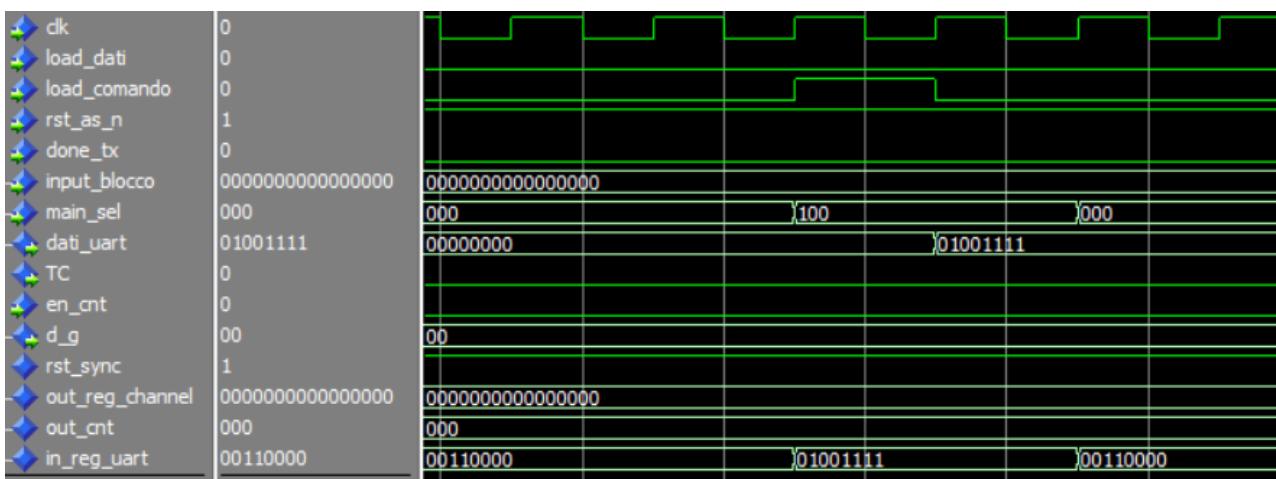
Una volta asserito **LOAD_DATI** si caricano i dati in uscita dalla memory interface nel registro interno della codifica ASCII. La word di 16 bit viene spezzata in 8 coppie dato glitch e analizzata. Dopo un periodo del clock, in base al valore di **D_G** si avrà un diverso valore di **MAIN_SEL** che piloterà il multiplexer collegato al registro dati della uart, il cui valore sarà valido al colpo di clock successivo, dopo che **LOAD_COMANDO** è stato asserito. Nel mentre il contatore interno della codifica seleziona la coppia di bit **D_G** successiva. Si rimane in questa configurazione fino a quando la PC interface non asserisce nuovamente **LOAD_COMANDO**, per dare il via ad una nuova codifica.

Questo ciclo continua finchè **TC** non viene asserito. Quando ciò si verifica si trasmette l'ultimo dato che si stava codificando, aspettando nella PC interface che **DONE_TX** venga portato a 1, e si inizia la trasmissione di \n e \r secondo le modalità viste nel primo timing.

Includiamo di seguito i risultati delle simulazioni. Nel primo grafico è possibile osservare la sequenza di codifica di una delle 8 coppie di bit dato/glitch.



Di sotto è invece possibile osservare in dettaglio l'evoluzione dei segnali quando si codifica un carattere, come ad esempio \r.



2. Decodifica ASCII

Il blocco di decodifica ascii è un componente della pc interface nella nostra macchina. Teoricamente avrebbe dovuto essere in un blocco separato - la user interface -, ma per una nostra scelta progettuale abbiamo deciso di inserirlo all'interno della pc interface, facendo in modo che quest'ultimo blocco citato si occupasse della trasmissione, della decodifica e codifica dei segnali inviati e ricevuti.

Il componente decodifica_ascii comunica agli altri componenti della pc interface tramite la CU di quest'ultima.

Il compito dell'elemento che stiamo descrivendo, come si intuisce dal nome, si occupa di interpretare e decodificare i segnali inviati dall'utente tramite terminale.

Da specifiche sappiamo che questi comandi hanno un formato differente: il comando F<n> indica l'invio della frequenza di campionamento, dove <n> è un numero compreso tra 0 e 9 e indica il prescaler della nostra frequenza di sample; il secondo comando è T<h><h> che indica l'impostazione, da parte dell'utente, del pattern del trigger generator. I numeri <h> sono in formato esadecimale.

Gli ultimi due segnali inviabili dall'esterno sono S e R: il primo dice alla macchina che, da quel momento in poi, dovrà cercare il match tra i dati in ingresso caricati sul bus e il trigger pattern settato precedentemente; il secondo, la R, indica che terminato il salvataggio in memoria dei dati si dovranno stampare i 128K campioni salvati prima e i 128K campioni salvati dopo il match da parte del trigger generator.

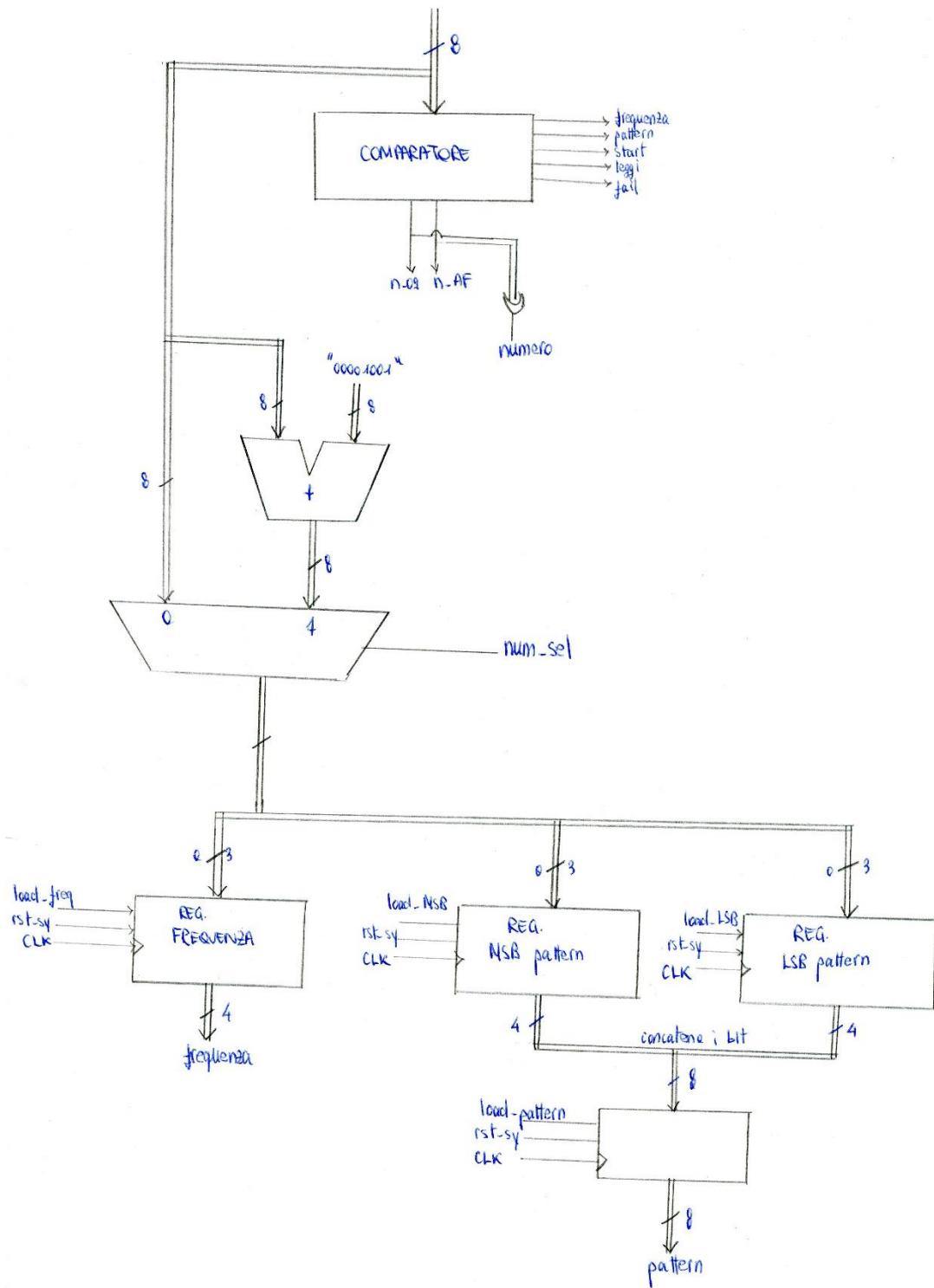
I dati saranno scritti dall'utente nella forma descritta qui sopra, ma verranno inviati lungo la linea seriale in formato ascii. Il blocco ricevitore della uart si occuperà di fornire gli 8 bit da interpretare al blocco di decodifica.

- **Datapath**

Il primo componente della decodifica è il comparatore, che si occupa di interpretare il dato e restituire in uscita dei flag in modo da capire in che modo interpretare i valori successivi. Il comparatore ci fornirà perciò in uscita i seguenti segnali: **matchF**, **matchT**, **matchS**, **matchR**, un segnale di **fail** che indica l'errore nell'invio del comando, numero compreso tra 0 e 9, numero compreso tra A e F e, per ultimo, l'uscita della porta OR a cui ingresso troviamo gli ultimi due segnali citati.

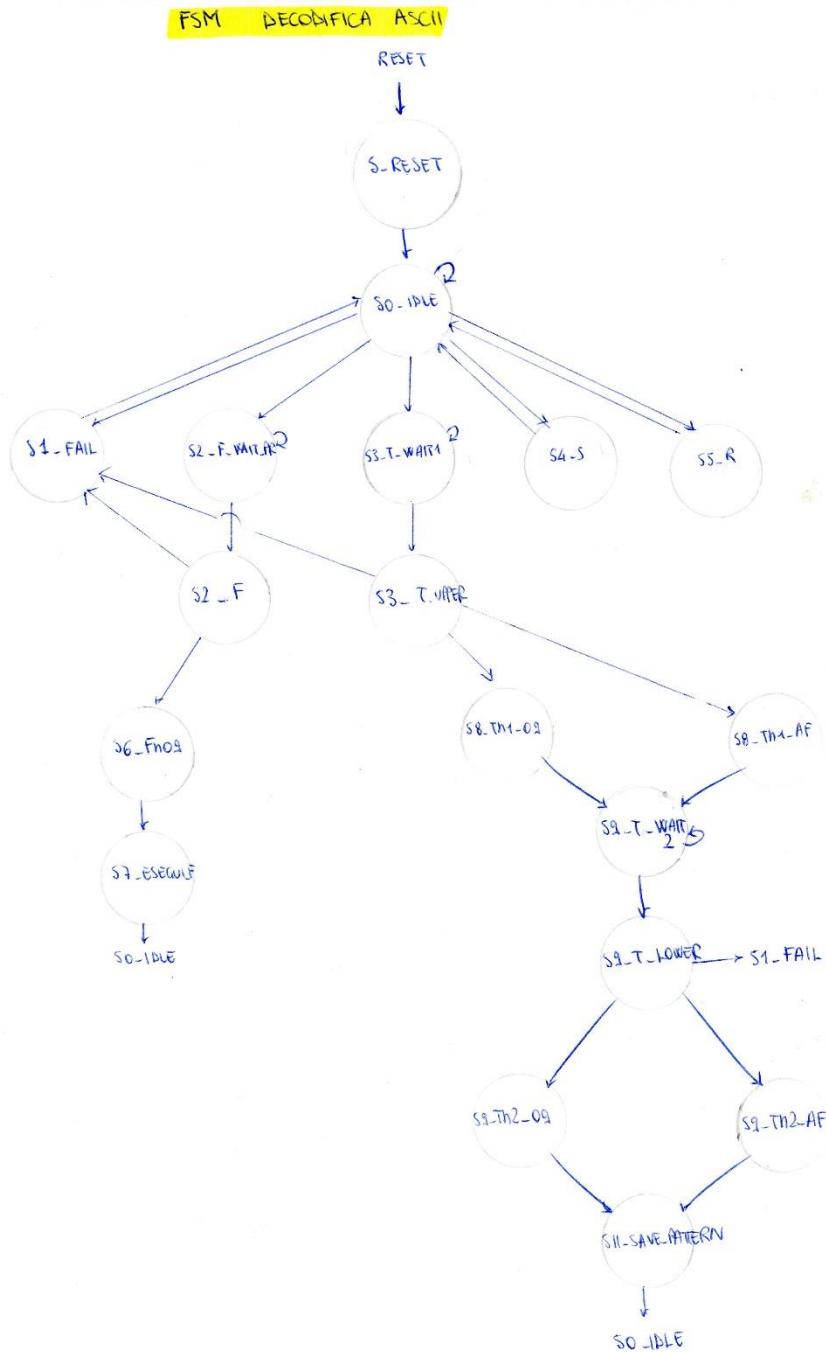
I comandi sono salvati all'interno di FLIP FLOP, le si comportano come flag per la nostra macchina. Dopo aver ricevuto una certa lettera la macchina prosegue nell'evoluzione tra gli stati: ad esempio se riceve una F, come valore successivo si aspetta un numero compreso tra 0 e 9. Per decodificare un numero compreso tra 0 e 9 basta prendere i primi quattro bit del dato in codifica ASCII (su 8 bit). Questo valore viene salvato all'interno di un registro la cui uscita andrà all'esterno del blocco. Quando si riceve il comando T la macchina si aspetta due numeri tra 0 e F, in esadecimale. I numeri tra 0 e 9 sono decodificati come spiegato precedentemente; per quanto riguarda i numeri compresi tra A e F, per averli in un formato binario interpretabile dalla macchina, vengono sommati a "00001001" da cui ricaveremo un numero tra 10 e 15. Il primo numero ricevuto verrà salvato all'interno del registro degli LSB, mentre il secondo dato viene salvato all'interno del registro degli MSB. Le uscite di questi ultimi due registri sono concatenate e rappresentano il pattern su 8 bit che verrà inviato al trigger generator. Gli ultimi due comandi inviabili, S e R, vengono semplicemente salvati in appositi FLIP FLOP che verranno resecati nel momento in cui verrà ricevuto l'acknowledge da parte dei blocchi che ricevono le uscite di questi due blocchi.

Di seguito illustriamo il datapath appena descritto.



- **FSM e timing diagram**

La decosifica ASCII è composta da un discreto numero di stati, che descriviamo brevemente in seguito, dopo aver visto il pallogramma.



- **S_RESET**: resettiamo tutti i componenti di memoria nel nostro blocco quando ricevo un reset dall'esterno.
Il reset inviato dall'esterno è asincrono;

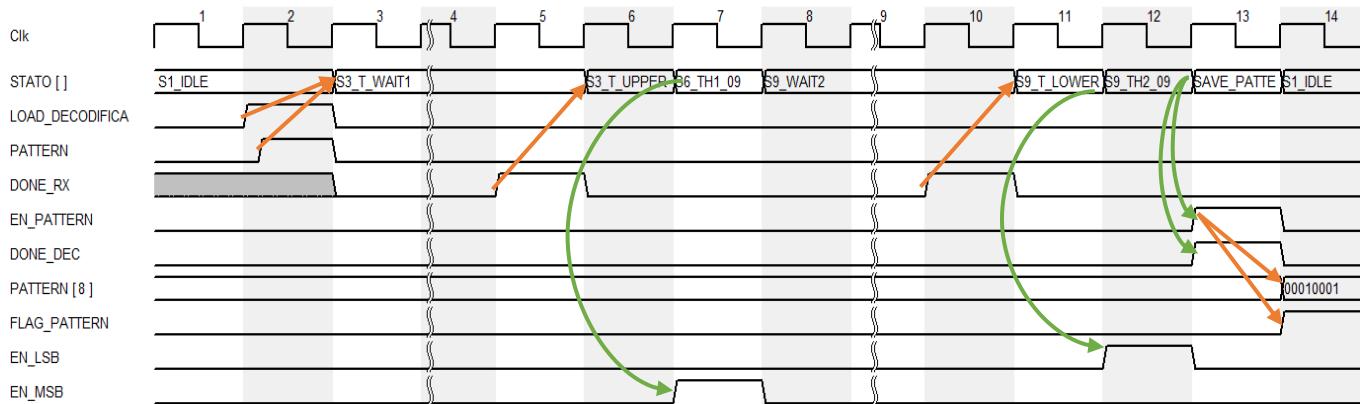
- **S0_IDLE**: è uno stato in cui aspetto che venga inviato il segnale di load della codifica che significa che posso decodificare il dato inviato dalla uart. In base ai segnali di uscita dal comparatore mi muoverò negli stati dei relativi comandi inviati dall'utente;

- S1_FAIL: quando ricevo un comando sbagliato entro in questo stato, la macchina invierà poi all'esterno il segnale 'K' per dire che c'è stato un errore nell'invio del comando, inoltre mando a 1 il segnale che indica la fine della decodifica del comando;
- S2_F: se entro in questo stato significa che è stata decodificata una F come comando, a questo punto come dato successivo mi aspetto un numero tra 0 e 9;
- S2_F_WAIT_PR: aspetto che arrivi il numero tra 0 e 9 che sarà il prescaler che mi servirà per settare la frequenza di campionamento della mia macchina;
- S3_T_WAIT1: ho ricevuto il comando 'T', aspetto il numero successivo che sarà in esadecimali, compreso tra 0 e F, questo dato verrà poi decodificato in modo che possa essere rappresentato su 4 bit in binario;
- S3_T_UPPER: controllo che il numero arrivato rientri nei parametri consentiti;
- S4_S e S5_R: questi due segnali sono analoghi, ricevo il comando 'S' o 'R' e abilito il flag che lo riguarda;
- S6_Fn09: il numero che ho ricevuto è compreso tra 0 e 9 e quindi corretto, setto il multiplexer in modo che passi il dato corretto;
- S7_ESEGUI_F: a questo punto, ricevuto il comando 'F' e il relativo prescaler, salvo questo valore all'interno del registro apposito abilitando l'enable e mando a 1 il segnale che indica la fine della decodifica;
- S8_Th1_09 e S8_Th1_AF: in entrambi i casi setto il selettore in base al primo numero che ho ricevuto per il trigger pattern salvando il dato del registro MSB (abilito il suddetto registro);
- S9_T_WAIT2: ho ricevuto il secondo numero del pattern e attendo l'arrivo del secondo prima di decodificarlo. Aspetto quindi che la uart mi invii il dato;
- S9_T_LOWER: controllo che il numero che mi è arrivato rientri nelle specifiche riguardanti questo valore;
- S9_Th2_09 e S9_Th2_AF: come negli stati S8, ma in questo caso salverò il dato decodificato all'interno del LSB;
- S11_SAVE_PATTERN: dopo aver ricevuto tutti i numeri che compongono il pattern effettuo una concatenazione degli MSB e degli LSB e salvo questo valore all'interno del registro che conterrà il mio trigger pattern da inviare al trigger generator.

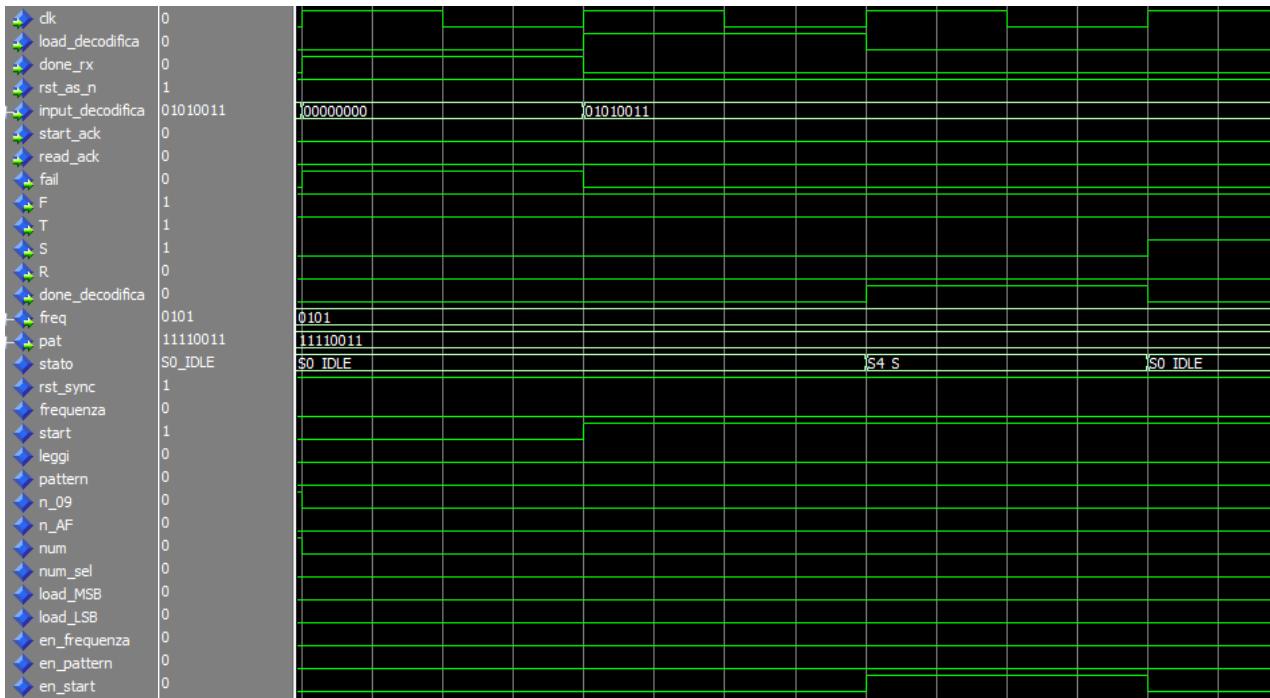
Includiamo il timing diagram studiato per ricevere e decodificare correttamente il comando che setta la frequenza.



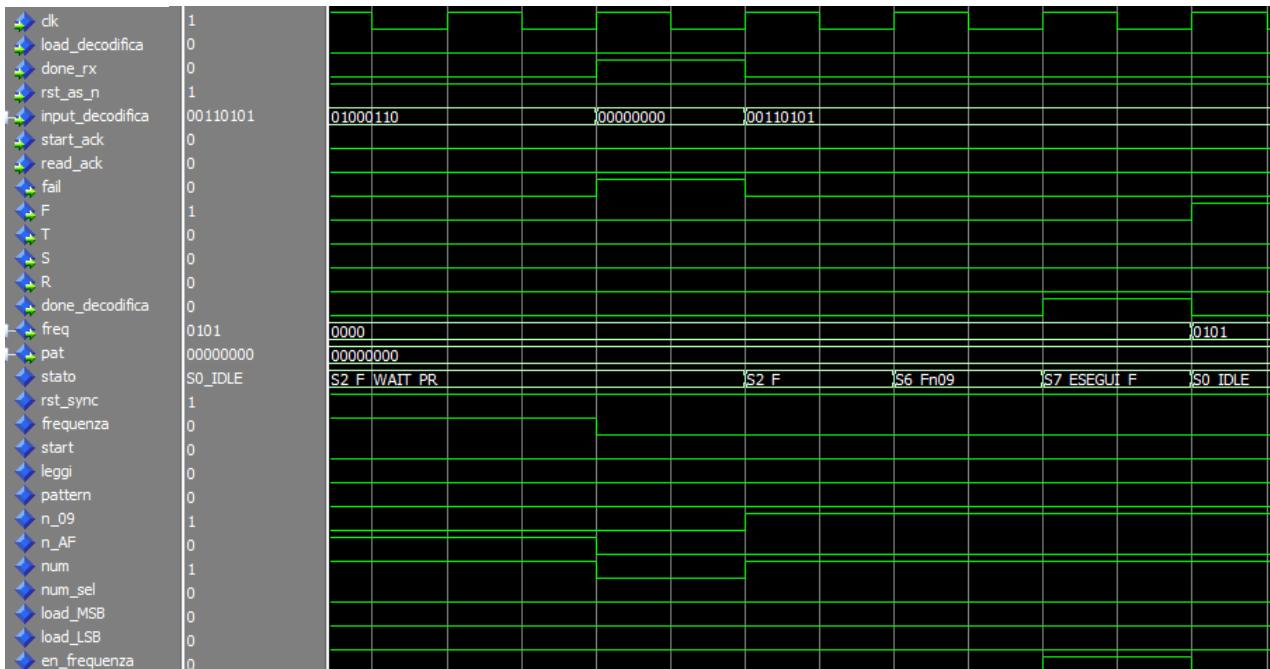
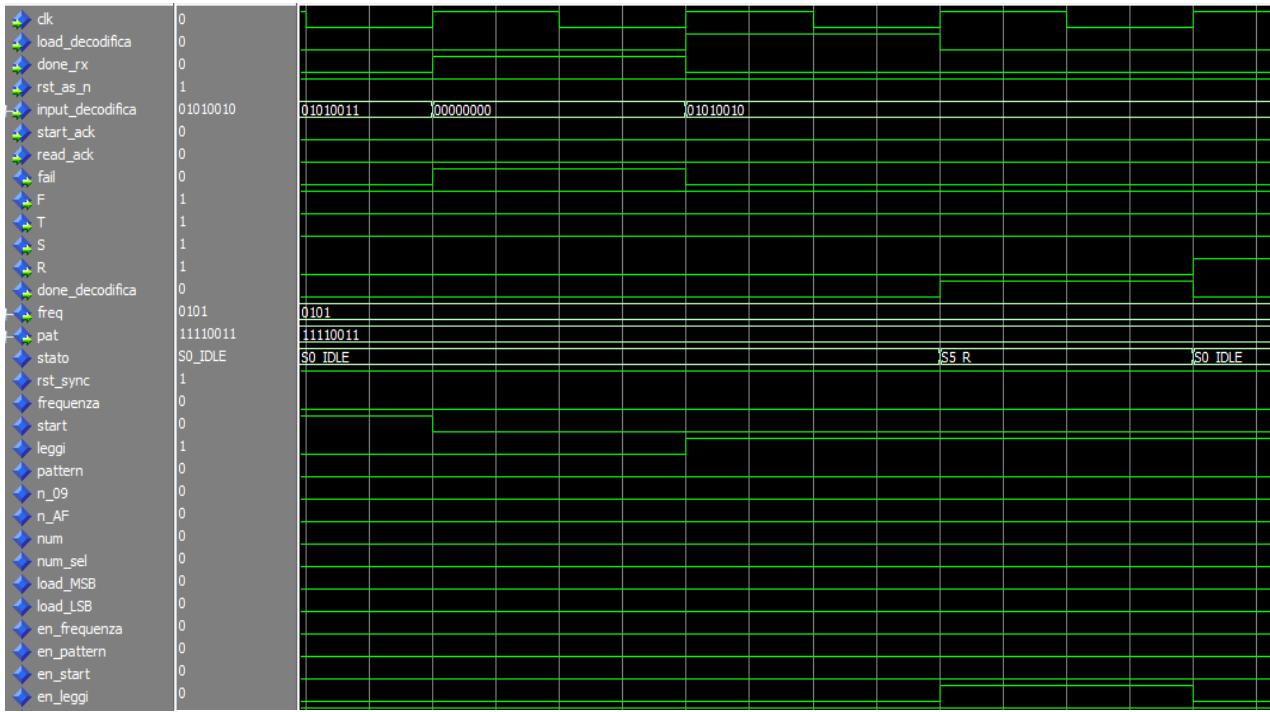
In questo timing possiamo invece vedere la decodifica del comando che setta il trigger e dei due caratteri in codice esadecimale che indicano il pattern di trigger. Abbiamo ipotizzato che sia stato inviato dal PC T11 per decidere che quali stati disegnare nel timing.



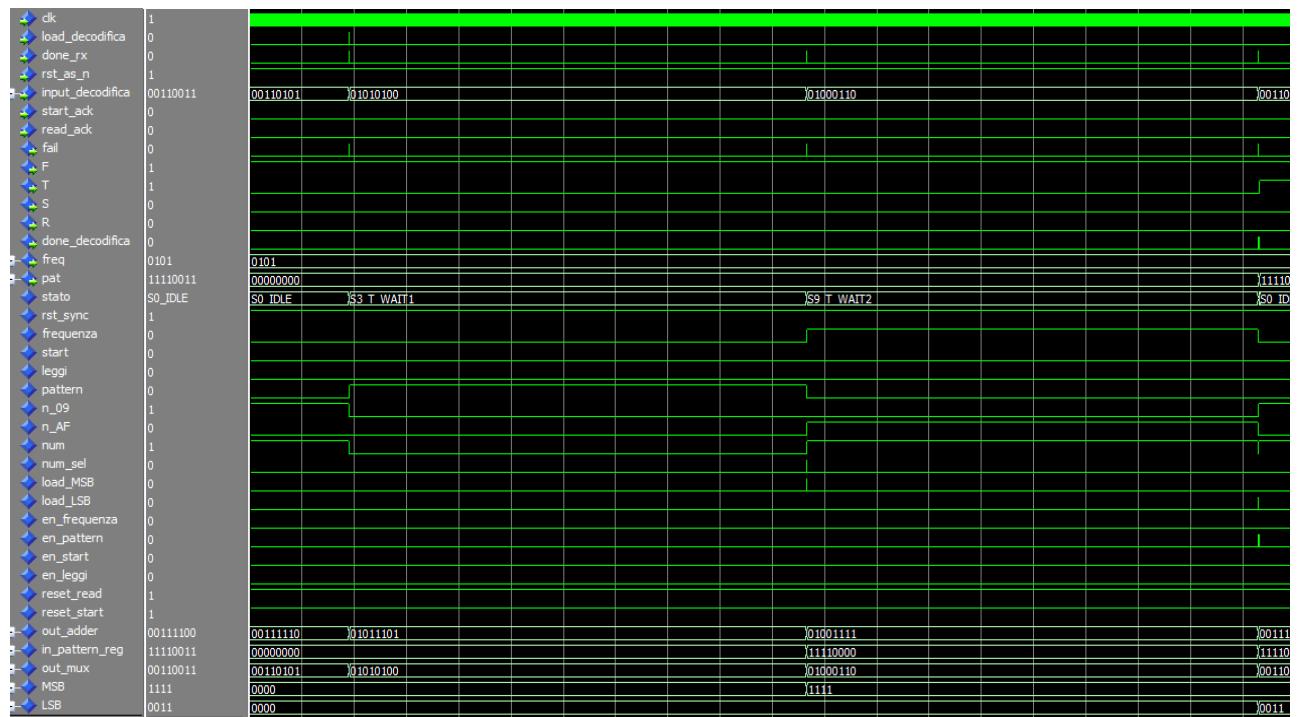
Includiamo adesso gli esiti delle nostre simulazioni.



Qui sopra è possibile osservare il rilevamento del comando di start, mentre nell'immagine sottostante si vede il rilevamento del comando di read.



Sopra è possibile osservare l'evoluzione dei segnali di comando e degli stati interni nel caso in cui sia stato ricevuto il comando F e poi un numero compreso tra 0 e 9. Sotto è possibile vedere la decodifica dei caratteri indicanti il pattern di trigger.



3. UART

Nel nostro progetto l'EIA RS-232 è il protocollo scelto, che definisce un'interfaccia per la realizzazione di una trasmissione seriale asincrona. Tale protocollo definisce come deve avvenire la trasmissione seriale del dato, la velocità di trasmissione del singolo bit e le caratteristiche di natura elettrica. In questo caso il frame scelto è pari a 8 bit, mentre la velocità di trasmissione è 115200 bps ($t_{bit} = 8.68\mu s$). Inoltre l'interfaccia RS-232 può operare anche in modalità full duplex.

Si ricorda che per asincrona si intende che non vi è alcun segnale di clock destinato a sincronizzare la trasmissione con la ricezione.

Lo standard RS-232 prevede che inizialmente la linea si trovi nello stato di riposo a livello logico '1' (idle), ma non appena vi sarà una transizione dal livello logico '1' a livello logico '0', inizierà una trasmissione. Il livello logico '0' è il bit di start, perciò esso durerà $8.68\mu s$, a cui seguiranno gli 8 bit di dato dal bit meno significativo (LSB) al più significativo (MSB) che dureranno $8.68\mu s$ ciascuno.

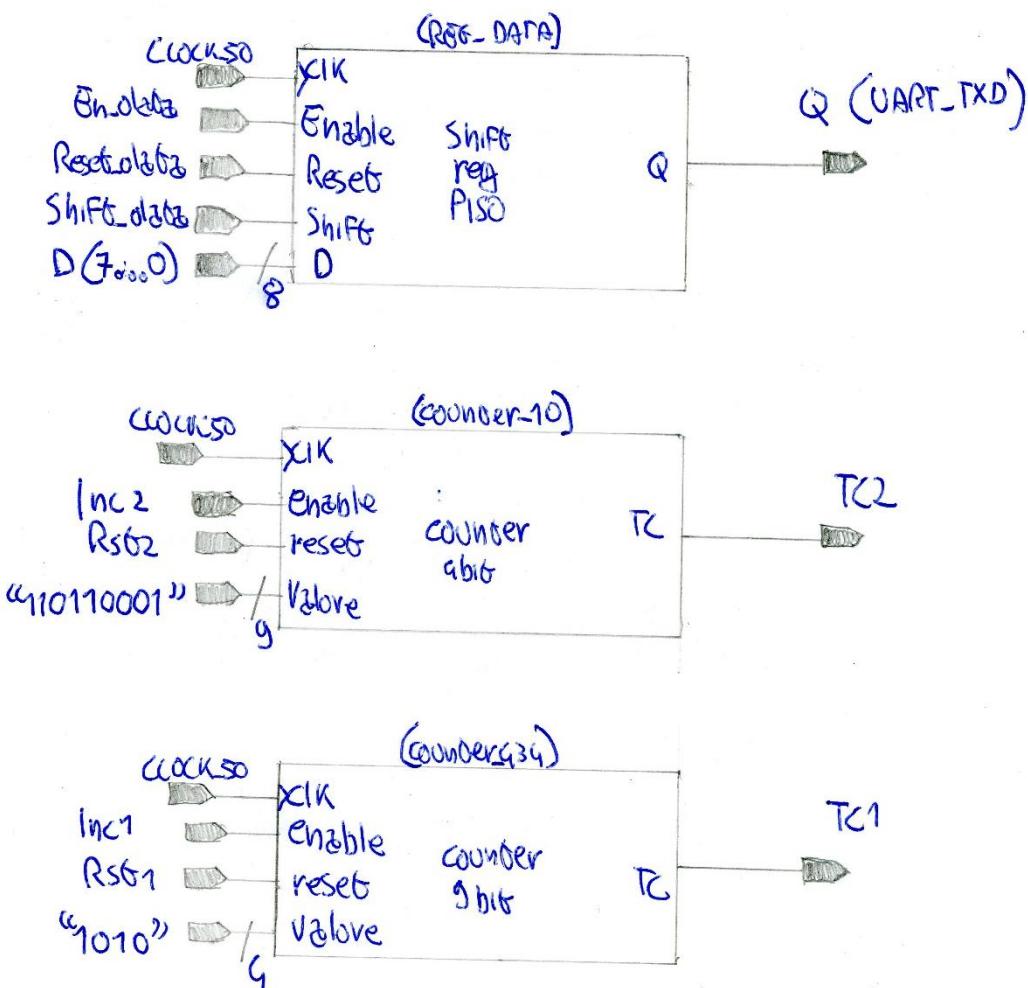
Seguirà infine un bit di stop al livello logico '1', che riporterà la linea in uno stato di idle.

Il dispositivo hardware che permette di convertire un flusso di dati da un formato parallelo a uno seriale asincrono, o viceversa, è lo UART. L'obiettivo era infatti progettare lo UART, che rispettasse il protocollo RS-232 per comunicare con il pc.

Per la UART descriveremo i datapath insieme alla control unit che li pilota, usando un approccio diverso rispetto a quello adottato finora.

a. UART TX

o **Datapath, FSM e timing diagram**



L'UART TX è composta da due contatori con TC variabile e uno shift_register parallel in serial out da 10 bit, dove il primo bit, a partire da sinistra, è lo stop bit, l'ultimo è lo start bit e i rimanenti sono i bit di dato. Il

trasmettitore parte inizialmente da uno stato di reset chiamato RESET_STATE, dove resetta il contenuto dello shift_register ponendo il suo contenuto con tutti ‘1’. Non appena viene asserito un segnale chiamato START il trasmettitore evolve in uno stato chiamato BEFORE_DATA_STATE, dove viene caricato il dato e ad esso viene concatenato lo start bit ‘0’. (Lo stop bit è già presente, poiché shift_register è stato inizializzato precedentemente). Successivamente si evolve in uno stato chiamato DATA_STATE nella quale si incrementa il primo contatore in modo tale che il TC1 arrivi ogni 8.68 μ s. Essendo il clock pari a 50 MHz e il baud rate voluto pari a 115200 si dovrà contare fino a $N = \frac{50000000}{115200} \approx 434$ prima che il singolo bit venga mandato fuori dallo shift_register tramite un comando di SHIFT.

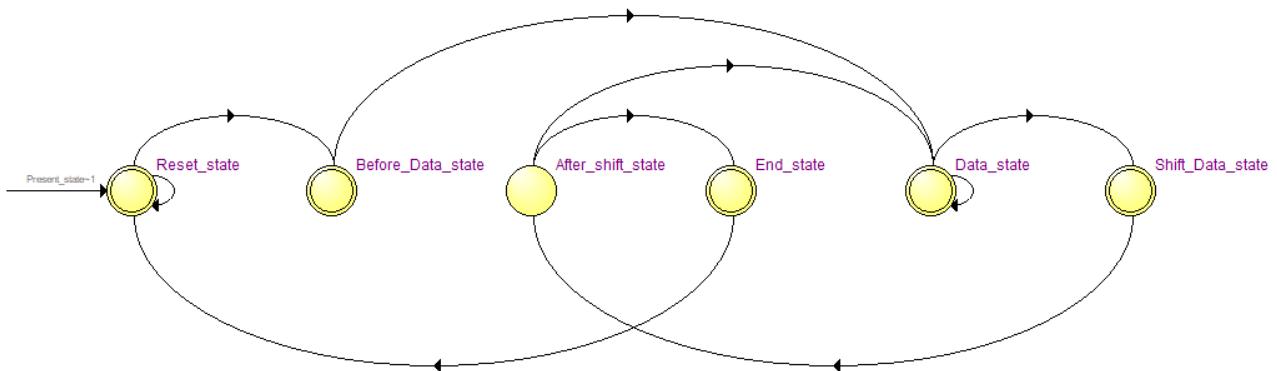
È importante garantire il rispetto della durata dei singoli bit, poiché l'unico elemento di sincronizzazione è dato dal fronte di discesa del bit di start. Infatti non appena viene asserito il TC1 si passa in uno stato chiamato SHIFT_DATA_STATE, nella quale verrà shiftato il primo bit di dato in uscita ed inoltre verrà incrementato un secondo contatore, che ci permetterà di capire quando la trasmissione è terminata.

Il trasmettitore, prima di controllare il TC2 del secondo contatore, evolve in uno stato chiamato AFTER_SHIFT_STATE, nella quale il secondo contatore si sarà effettivamente incrementato e il dato shiftato. In realtà in tutto ciò si perde un colpo di clock ogni 434, ma l'errore sul tempo di bit rimane comunque nelle specifiche del protocollo EIA RS-232.

Quando il TC2 sarà pari a ‘1’ la macchina evolverà in uno stato chiamato END_STATE, dove verrà asserito un bit di DONE che comunicherà la fine di una trasmissione.

Successivamente la macchina tornerà nella condizione di essere in grado di effettuare una nuova trasmissione.

Di seguito possiamo vedere il pallogramma ed i timing interni della uart tx relativi al primo bit ed ultimo bit trasmessi.



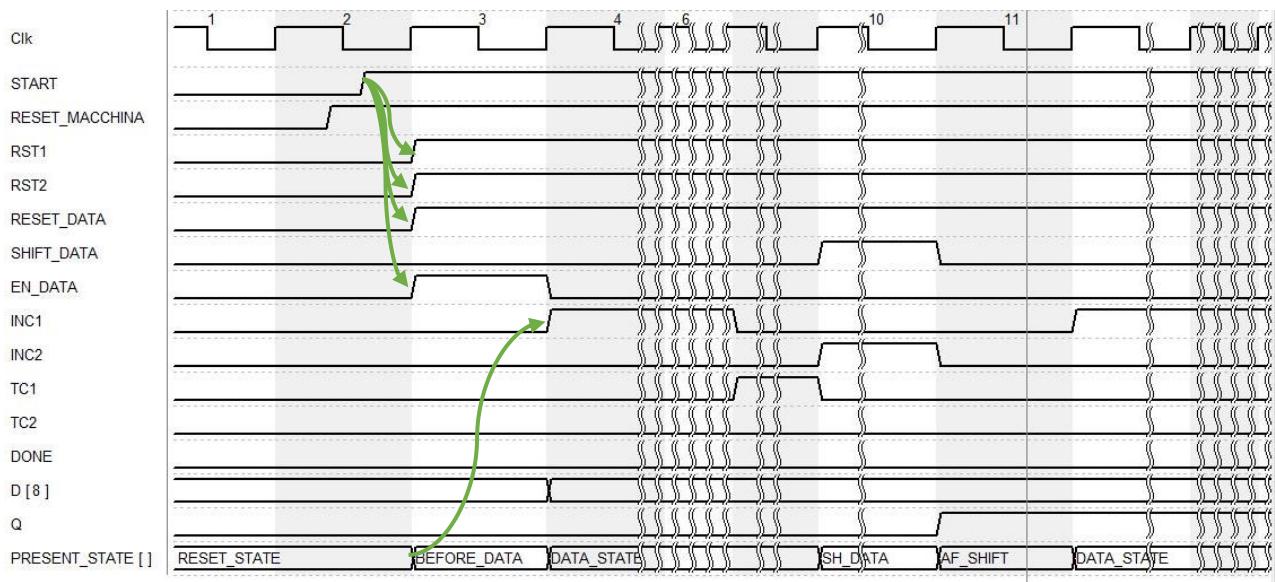


Figura 4 Primo bit trasmesso

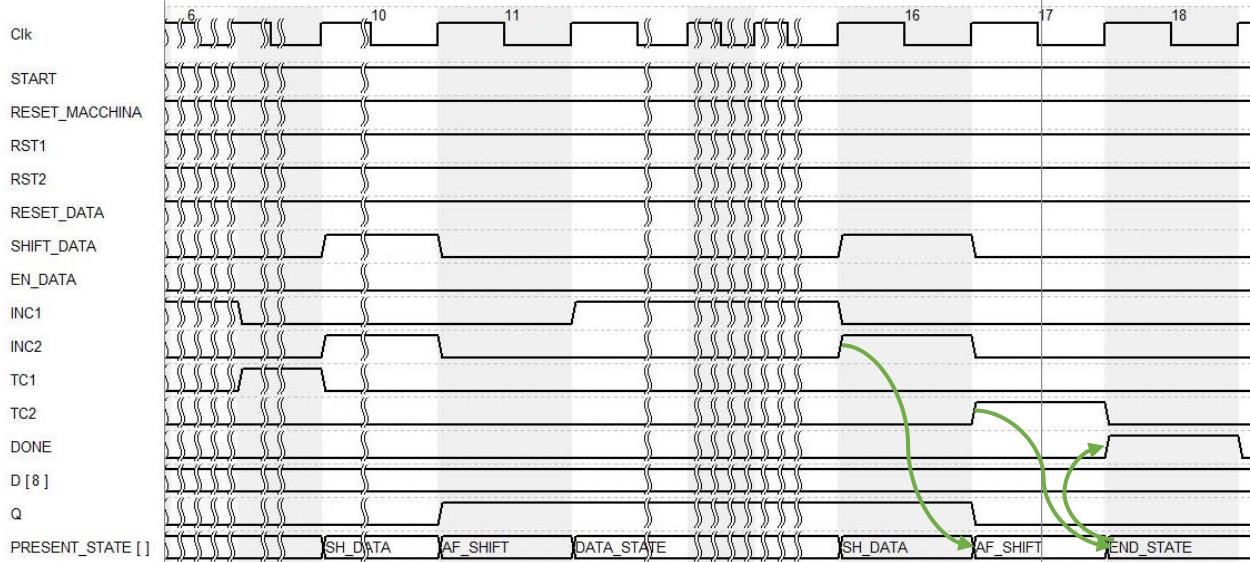
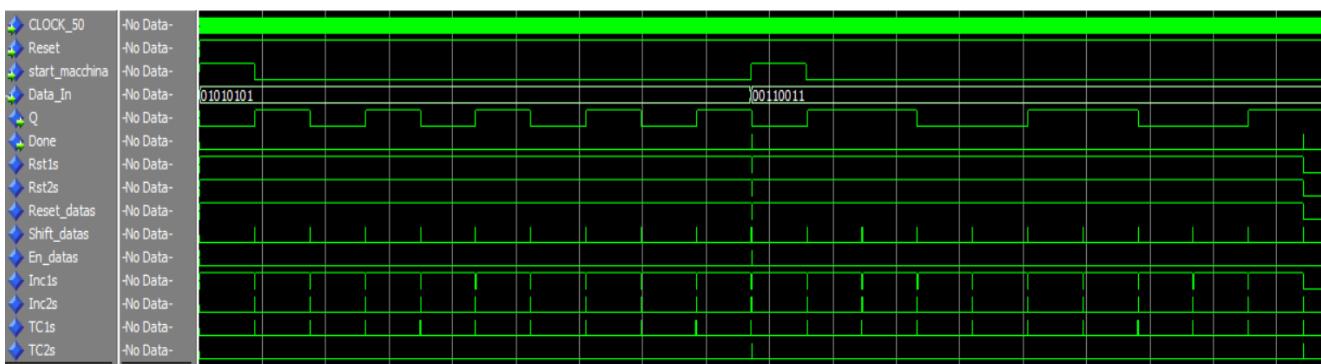


Figura 5 Ultimo bit trasmesso

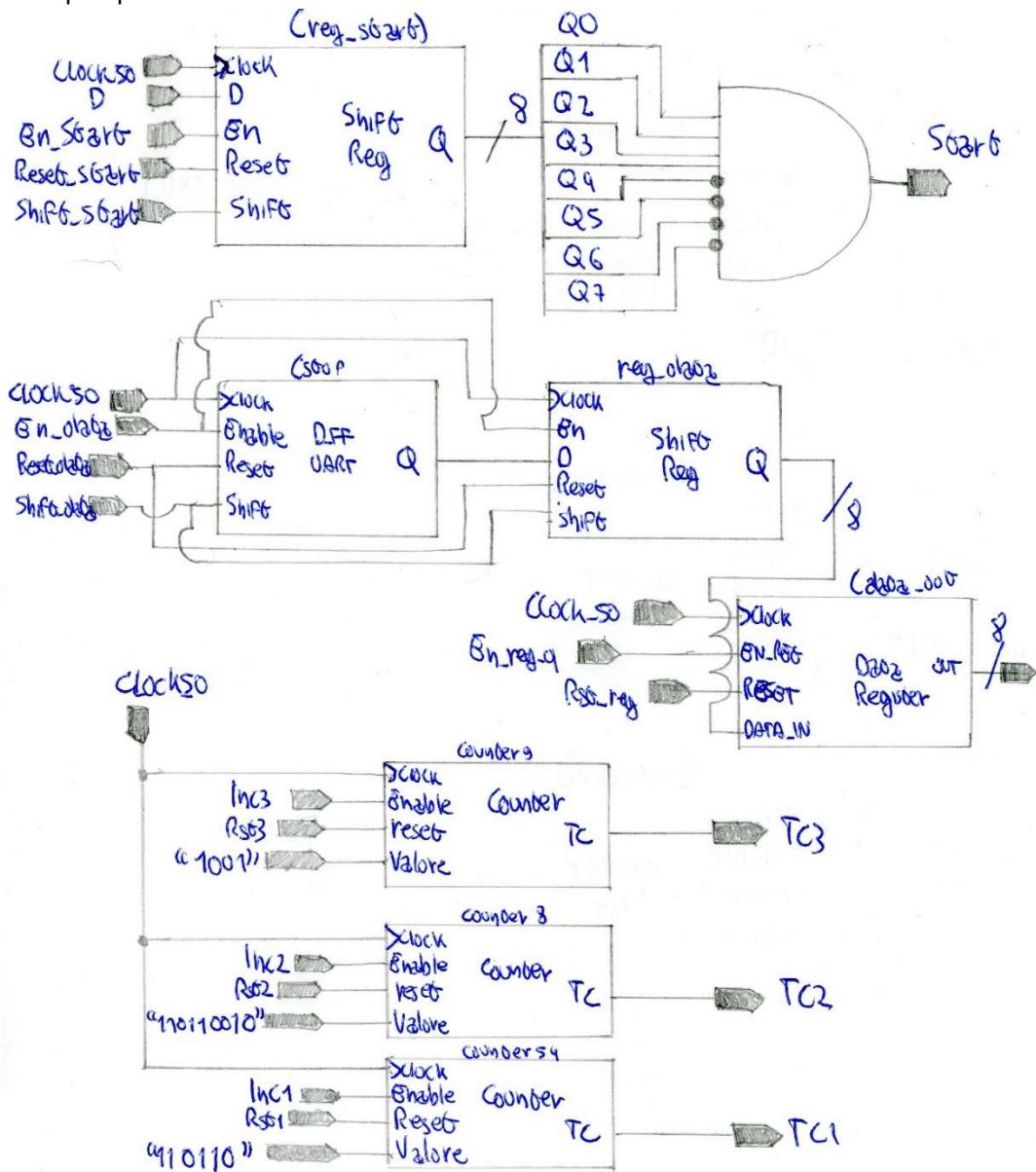
Laddove ci fosse da aspettare il terminal count dei contatori abbiamo inserito delle terminazioni, poiché non è di interesse pratico visualizzare tutti i colpi di clock tra uno shift e l'altro. Di seguito i risultati della simulazione che dimostra il funzionamento del trasmettitore nel caso di due caratteri trasmessi di seguito, in cui si possono osservare in particolare i terminal count dei contatori ed il comando di shift.



a. UART RX

o Datapath, FSM e timing diagram

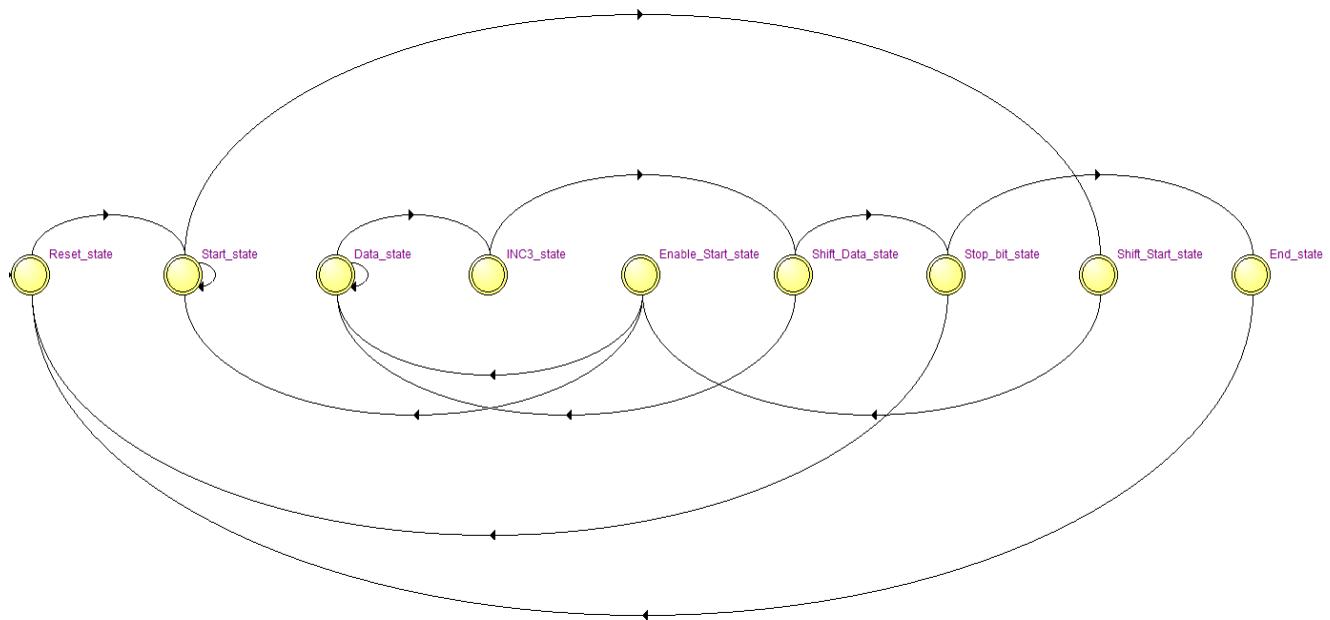
L'UART RX è composta da due shift_register a 8 bit, un data_register a 8 bit, tre contatori con TC variabile ed un D flip flop.



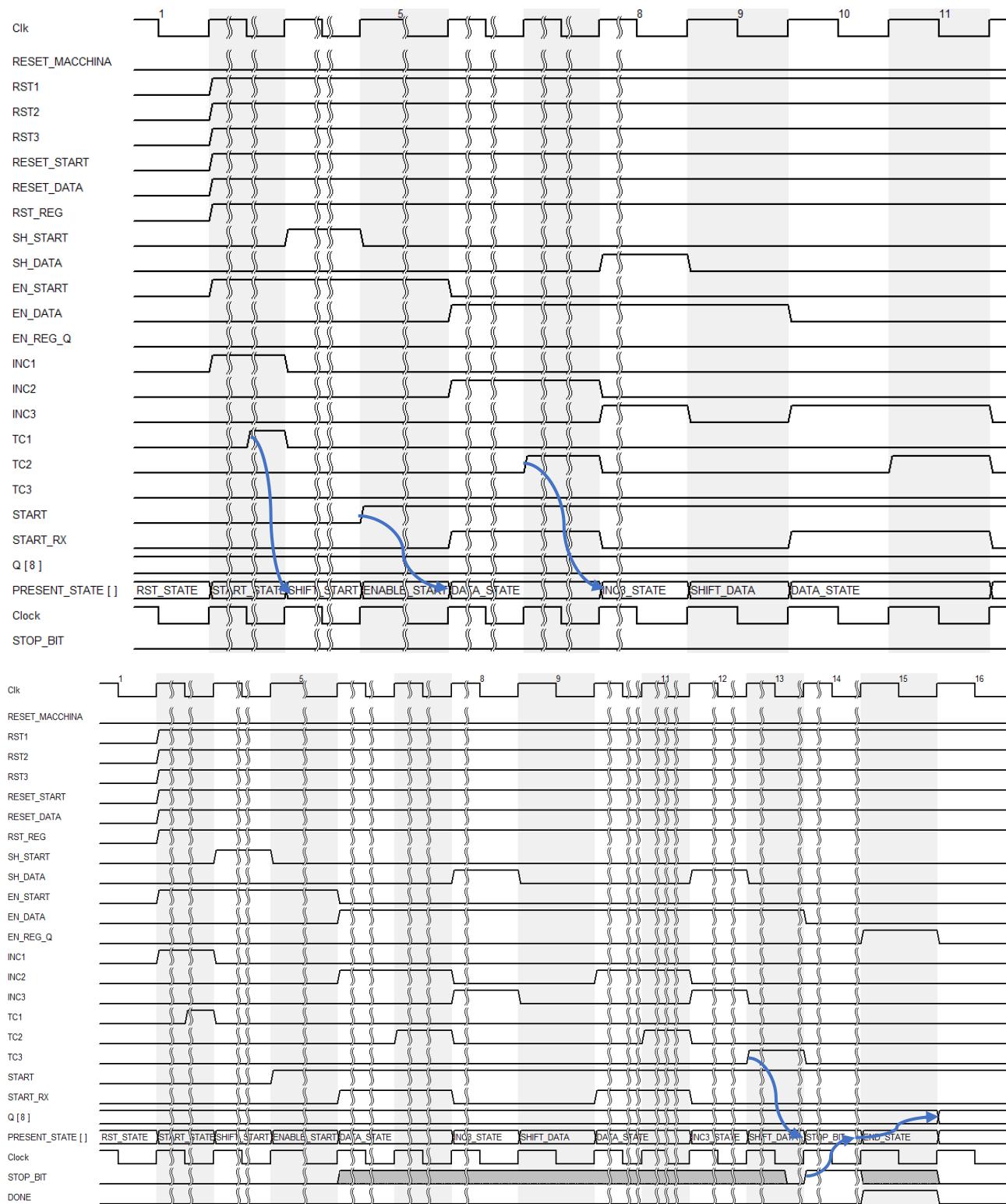
L'idea di massima nel ricevitore è quella di sovra campionare la linea, almeno 8 volte più veloce della velocità di trasmissione, in modo tale da non perdersi la transizione dal livello logico '1' al livello logico '0', poiché essa è l'unico elemento di sincronizzazione. Inoltre il campionamento in ricezione viene effettuato al centro di ciascun bit. Il ricevitore evolve nel seguente modo:

- **RESET_STATE:** stato nella quale tutto il datapath viene resettato, si va poi in **START_STATE**.
- **START_STATE:** stato nella quale viene abilitato il primo contatore con TC pari a $N = \frac{434}{8} \approx 54$. Questo contatore ci permetterà di sovra campionare la linea.
- **SHIFT_START_STATE:** in questo stato viene abilitato lo shift_register chiamato REG_START ed inoltre vengono fatti shiftare in modo seriale i bit presenti sulla linea. Questo shift_register, poiché viene asserito il comando di shift e di enable otto volte più veloce del baud rate della linea seriale, ci permetterà di notare il fronte di discesa della linea che identificherà l'inizio della recezione.

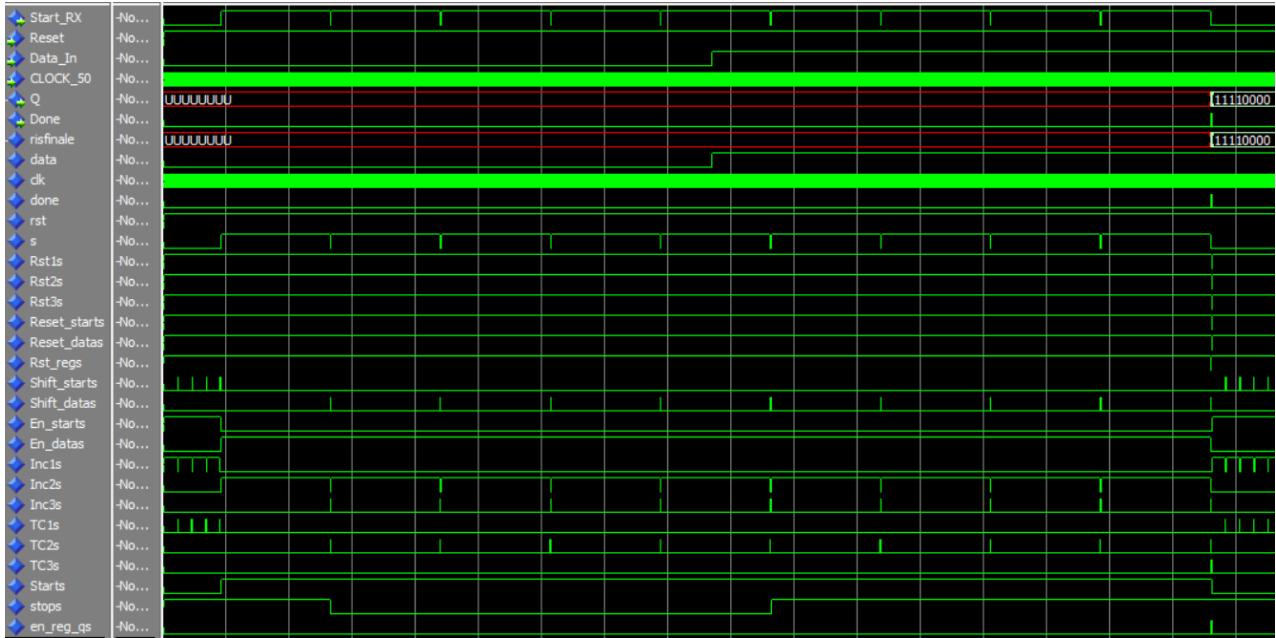
- **ENABLE_START_STATE**: in questo stato si verifica se la condizione di inizio trasmissione sia avvenuta, infatti viene controllato il contenuto del REG_START. Se al suo interno è presente “11110000” significa che vi è stato un fronte di discesa ed inoltre che ora ci troviamo al centro del bit di start.
- **DATA_STATE**: in **DATA_STATE** si abilita un secondo contatore modulo 434. Questo contatore ci permetterà di campionare ciascun bit al centro.
- **INC3_STATE**: in questo stato vengono abilitati il D flip flop e lo shift_register chiamati rispettivamente DFF_UART e REG_DATA. I bit vengono fatti shiftare in modo seriale ogni 434 colpi di clock, prima all'interno del D flip flop e poi all'interno dello shift_register. Il compito del D flip flop è quello di memorizzare alla fine della recezione lo stop bit, per tale motivo viene posto prima dello shift_register. Sempre in questo stato viene abilitato un contatore in modulo 8.
- **SHIFT_DATA_STATE**: in questo stato viene controllato io TC del terzo contatore, che ci permetterà di capire che la recezione degli 8 bit di dato è avvenuta.
- **STOP_BIT_STATE**: in tale stato viene controllato il bit di stop, che è l'uscita del DFF_UART.
- **END_STATE**: se il bit di stop è pari a '1' si evolverà in **END_STATE** che asserirà un bit di **DONE** per comunicare la corretta recezione. Inoltre viene abilitato un registro chiamato DATA_REGISTER in cui verranno salvati gli 8 bit di dato.



Di seguito abbiamo illustrato i timing diagram del primo bit ricevuto e dell'ultimo bit ricevuto.



Infine includiamo la simulazione eseguite col testbench, in cui si può osservare la prima ricezione di un carattere dopo il reset della uart.



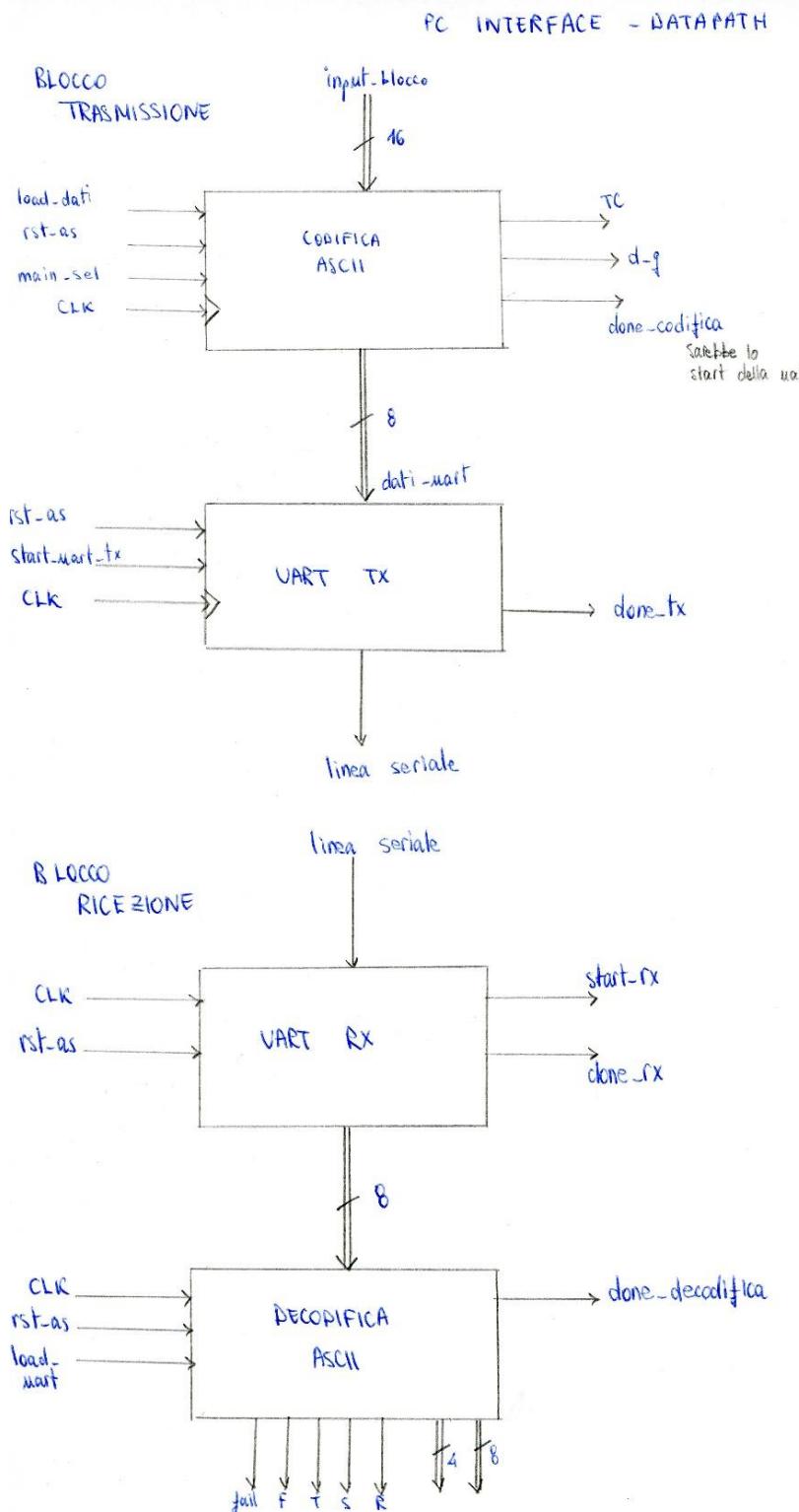
Il registro di uscita della uart rx è collegato ad un altro elemento di memoria, che non viene mai abilitato prima che le uscite di questo commutino assumendo dei valori logici validi diversi da undefined.

Si può notare la ricezione di ogni singolo bit del payload più i bit di start e stop, indicata dalle commutazioni di **start_rx**, indicante la ricezione di un singolo bit, e **shift_data**, indicante lo shift del dato ricevuto dal flip flop allo shift register.

▪ Pc interface

Abbiamo descritto tutti i blocchi interni alla PC interface e definito dei timing da rispettare. Possiamo quindi definire datapath e control unit della unità a livello più alto.

○ Datapath



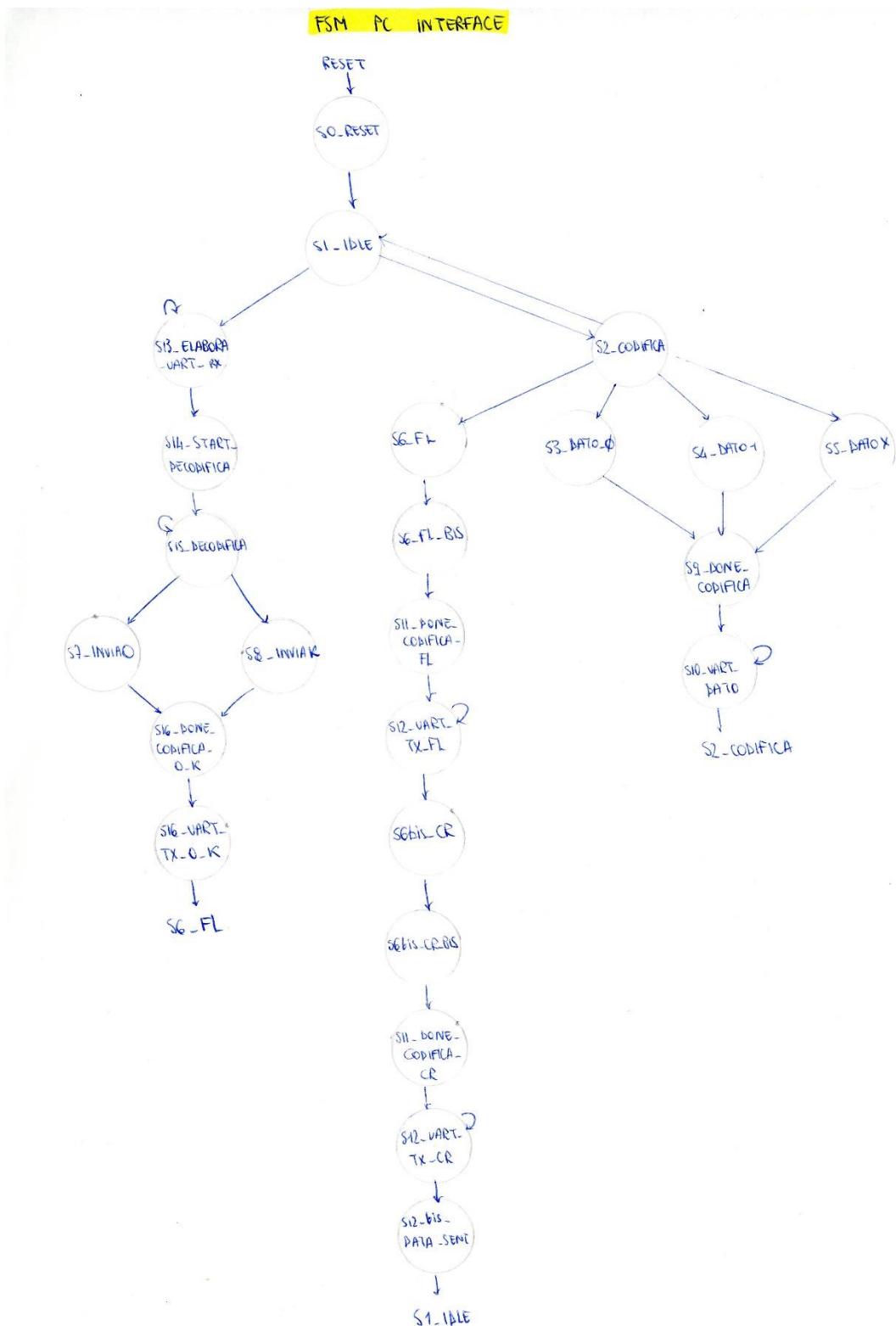
L'ingresso della codifica ASCII è collegato all'uscita della memory interface, mentre la sua uscita va alla UART_TX.

La UART_RX è collegata alla linea seriale e il dato ricevuto viene inviato al blocco di decodifica ASCII.

Nel datapath è presente anche un FLIP FLOP usato come semaforo per sentire il terminal count del contatore dentro la codifica ASCII. Quando questo segnale viene asserito l'uscita del FLIP FLOP va ad uno finché non viene resettato dalla control unit.

I segnali di ingresso e uscita della PC interface sono di fatto quelli visti prima nei punti precedenti, con l'aggiunta del segnale BLOCK_SENT, usato per segnalare alla Main Fsm dell'analizzatore l'invio di un pacchetto di dati all'esterno tramite la linea seriale.

- **FSM e timing diagram**



La PC interface è stata progettata a posteriori con in mente il timing dei blocchi che la compongono. Sono stati quindi inseriti degli stati vuoti su misura per poter rispettare tutti i vincoli temporali imposti dalle scelte di progetto effettuate in precedenza.

La PC interface ha il compito di verificare se è arrivato un dato dalla UART RX oppure se la Main Fsm vuole inviare dei dati all'esterno. La sua FSM dovrà pertanto avere degli stati di attesa in cui aspetta che si verifichino certe condizioni (ad esempio il segnale **START_RX** a 1) prima di saltare in uno stato in cui si eseguono tutte le operazioni del caso. La PC interface non pilota blocchi esterni e comunica con la Main Fsm asserendo dei flag in uscita, che vengono continuamente controllati da quest'ultima.

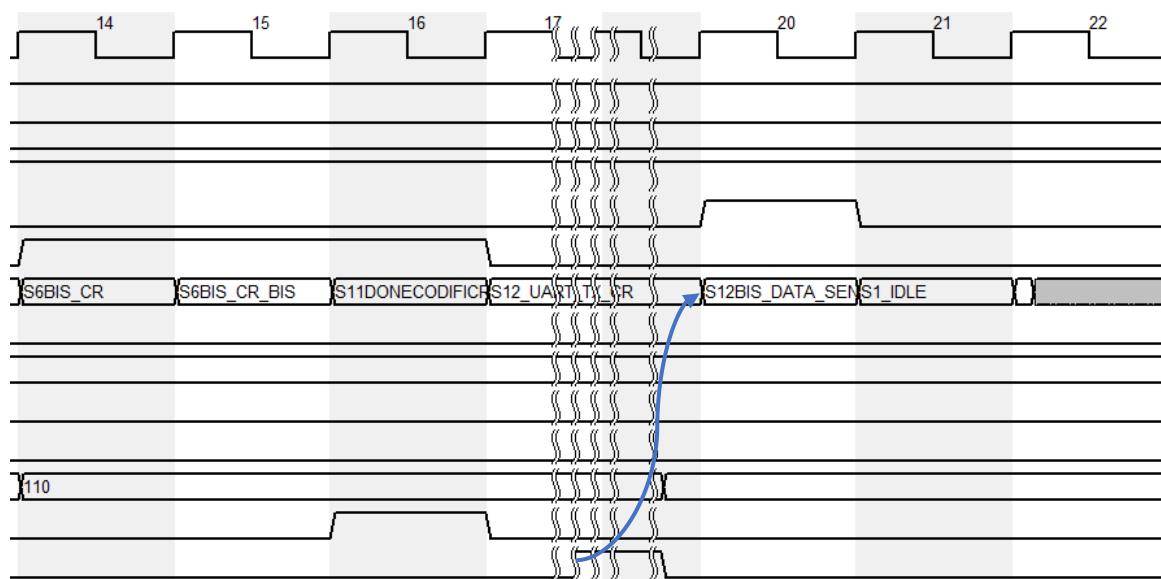
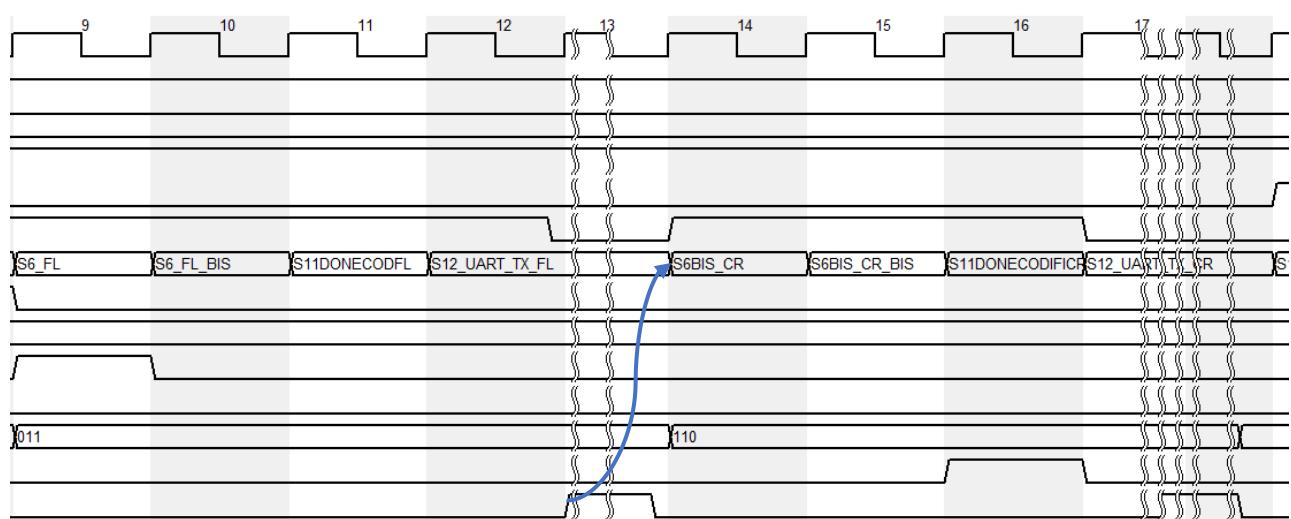
Spiegheremo le scelte progettuali direttamente nella descrizione degli stati della control unit, riportati di seguito:

- **S0_RESET**: in questo stato si abbassa il segnale di reset collegato al semaforo ed a tutti i componenti interni alla PC interface;
- **S1_IDLE**: qui si aspetta che venga comandato il caricamento di un dato asserendo **LOAD_DATI**, comandato dalla Main Fsm quando questa vuole inviare sulla linea seriale gli 8 caratteri di dato o glitch, oppure che venga rilevato l'inizio di una ricezione, segnalato con **START_RX**. Nel primo caso si salta in altrimenti **S2_CODIFICA** si va in **S13_ELABORA_UART_RX**;
- **S2_CODIFICA**: si verifica dal risultato della comparazione della coppia di bit dato/glitch, eseguita dalla codifica, quale carattere si deve codificare. A seconda che si debba inviare uno 0, un 1 oppure una x si va in **S3_DATO0**, **S4_DATO1**, **S5_DATOX**. Se **TC_SUM** è a 1, che significa che gli 8 caratteri relativi ai dati e glitch sono stati tutti inviati sulla linea seriale, si va in **S6_FL** e si inizia la sub-routine in cui si inviano \n e \r;
- **S3_DATO0**: si seleziona l'uscita del mux visto nella codifica in modo che dia in uscita la codifica dello 0, si abilitano il contatore che seleziona la coppia dato/glitch con **EN_CNT** ed il registro di uscita della codifica ASCII dove si salva il dato che si desidera codificare e inviare alla UART TX, poi si va in **S9_DONE_CODIFICA**;
- **S4_DATO1**: si seleziona l'uscita del mux visto nella codifica in modo che dia in uscita la codifica dell'1, si abilitano il contatore che seleziona la coppia dato/glitch con **EN_CNT** ed il registro di uscita della codifica ASCII dove si salva il dato che si desidera codificare e inviare alla UART TX, poi si va in **S9_DONE_CODIFICA**;
- **S5_DATOX**: si seleziona l'uscita del mux visto nella codifica in modo che dia in uscita la codifica della x, si abilitano il contatore che seleziona la coppia dato/glitch con **EN_CNT** ed il registro di uscita della codifica ASCII dove si salva il dato che si desidera codificare e inviare alla UART TX, poi si va in **S9_DONE_CODIFICA**;
- **S6_FL**: si resetta il semaforo usato per rilevare il terminal count della codifica ASCII, si seleziona il carattere \n tramite il mux interno al blocco appena citato e si asserisce **LOAD_COMMANDO** per iniziare il processo di codifica. Si abilita il registro di uscita della codifica ASCII e si va in **S6_FL_BIS**;
- **S6_FL_BIS**: stato vuoto inserito per rispettare il timing della codifica ASCII, si mantiene stabile il comando del mux visto nel punto precedente.
- **S6bis_CR**: si seleziona il carattere \n tramite il mux interno al blocco appena citato e si asserisce **LOAD_COMMANDO** per iniziare il processo di codifica. Si abilita il registro di uscita della UART RX e si va in **S6bis_CR**;
- **S6bis_CR_BIS**: stato vuoto inserito per rispettare il timing della codifica ASCII, si mantiene stabile il comando del mux visto nel punto precedente.
- **S7_INVIAO**: si asserisce **LOAD_COMMANDO**, si pilota il mux della codifica ASCII e si abilita il suo registro di uscita;
- **S7_INVIAO_BIS**: stato vuoto utilizzato per rispettare il timing della codifica ASCII. Da qui si va in **S16_DONE_CODIFICA_O_K**;
- **S8_INVIAK**: si asserisce **LOAD_COMMANDO** e si pilota il mux della codifica ASCII e si abilita il suo registro di uscita;

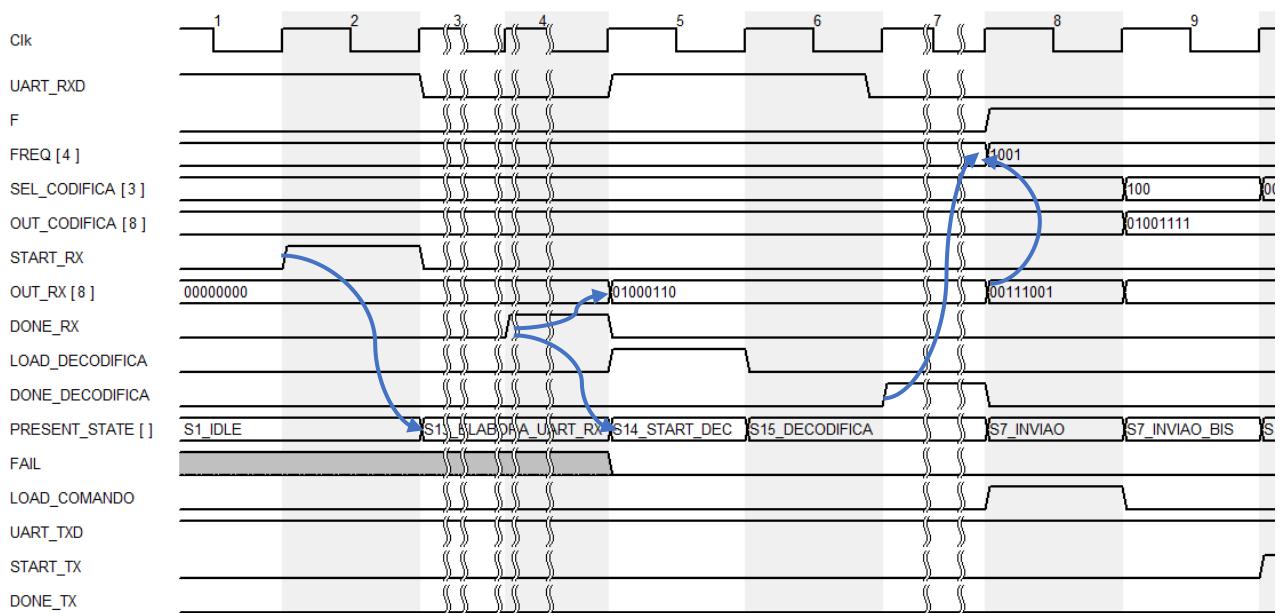
- S8_INVIAK_BIS: stato vuoto utilizzato per rispettare il timing della codifica ASCII. Da qui si va in S16_DONE_CODIFICA_O_K;
- S9_DONE_CODIFICA: si asserisce **START_TX** e si avvia la trasmissione del carattere. Da qui si va poi in S10_UART_TX_DATO;
- S10_UART_TX_DATO: si attende che la trasmissione sia terminata, controllando che venga portato a 1 **DONE_TX**, quindi si torna in S2_CODIFICA. In caso contrario si resta in questo stato;
- S11_DONE_CODIFICA_FL: si avvia la trasmissione di \n e si va in S12_UART_TX_FL;
- S11_DONE_CODIFICA_CR: si avvia la trasmissione di \r e si va in S12_UART_TX_CR;
- S12_UART_TX_FL: si aspetta che la trasmissione sia finita, controllando **DONE_TX**. Successivamente si salta in S6bis_CR;
- S12_UART_TX_CR: si aspetta che la trasmissione sia finita, controllando **DONE_TX**. Successivamente si salta in S12bis_DATA_SENT;
- S12bis_DATA_SENT: si asserisce **BLOCK_SENT** per indicare alla Main Fsm che gli 8 bit indicanti lo stato dei canali di ingresso più i due caratteri di terminazione \n e \r sono stati inviati. Si asserisce anche nel caso in cui sia stato inviato il carattere O oppure K. Si torna quindi in S1_IDLE;
- S13_ELABORA_UART_RX: si attende il completamento della ricezione, indicato da **DONE_RX**. Quando questo avviene si salta in S14_START_DECODIFICA;
- S14_START_DECODIFICA: si avvia la decodifica dei caratteri ricevuti;
- S15_DECODIFICA: si controlla l'esito della decodifica. Se il segnale è stato ricevuto correttamente si va in S7_INVIAO, altrimenti si va in S8_INVIAK;
- S16_DONE_CODIFICA_O_K: si avvia la trasmissione dei caratteri O o K, poi si va in S16_UART_TX_O_K;
- S16_UART_TX_O_K: si aspetta che la trasmissione sia finita e poi si va in S6_FL per inviare prima \n e poi \r.

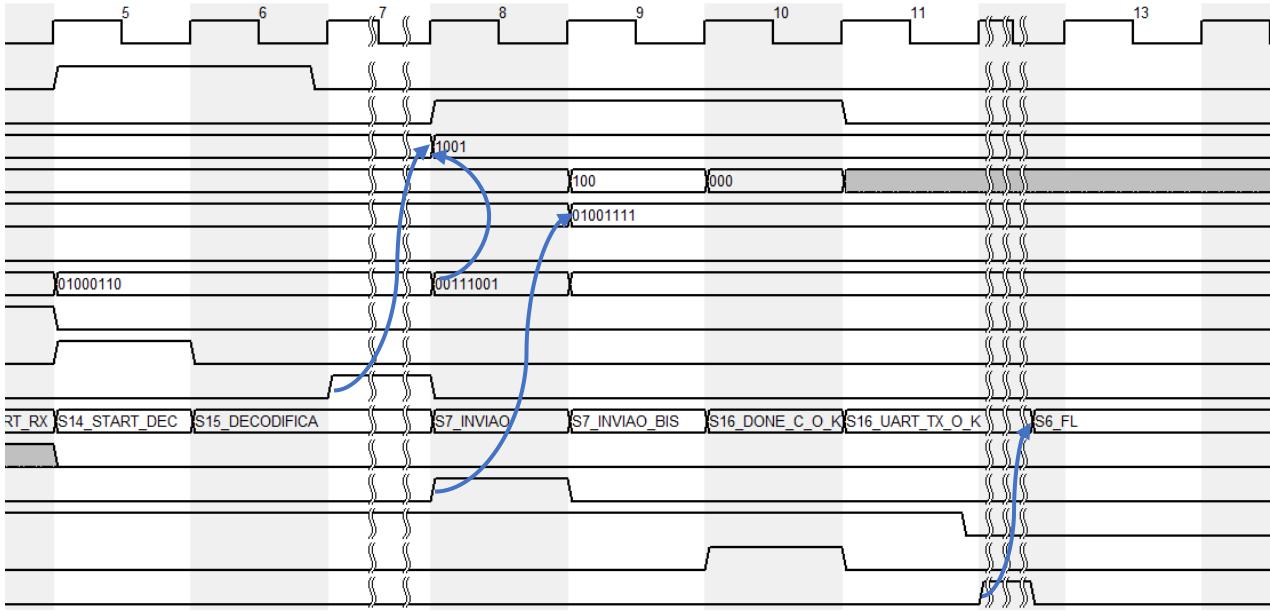
Di seguito è possibile vedere il timing diagram della PC interface in trasmissione.





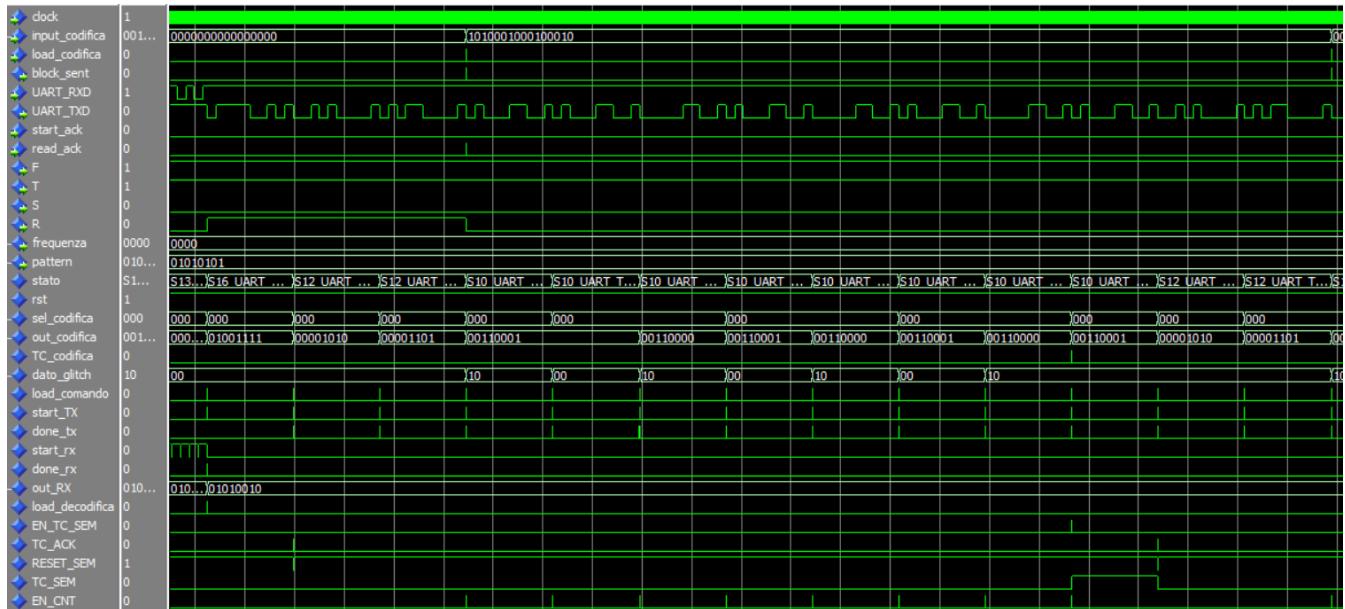
Mentre questo è il timing in ricezione





Passiamo ora all'esito delle simulazioni su MODELSIM. Abbiamo testato la PC interface in trasmissione e ricezione, ottenendo i seguenti risultati.

In questo screen è possibile osservare la trasmissione degli 8 caratteri associati ai canali dell'analizzatore logico. Si possono anche notare i segnali di controllo, omessi nel timing fornito qui sopra, ma descritti nell'analisi degli stati della FSM.



Qui sotto si può vedere la ricezione del comando Te del valore del pattern di trigger.

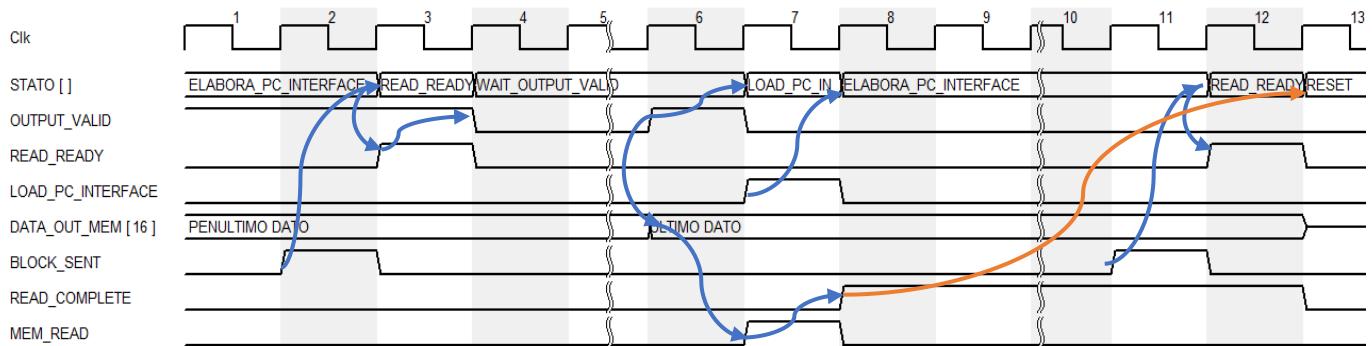
	clock	1														
	rst_as	1														
	input_codifica	000...	0000000000000000													
	load_codifica	0														
	block_sent	0														
	UART_RXD	1														
	UART_TXD	0														
	start_ack	0														
	read_ack	0														
	F	1														
	T	1														
	S	0														
	R	0														
	frequenza	0000	0000													
	pattern	010...	01010101													
	stato	S1...	S13 ELABORA UART RX	S14 START DECODIFICA	S15 DECODIFICA	S7 INVIAO	S7 INVIAO BIS	S16 DONE CODIFICA O K	S16 UART TX O K							
	rst	1														
	sel_codifica	000	000			100										
	out_codifica	010...	00001101				01001111									
	TC_codifica	0														
	dato_glitch	00	00													
	load_comando	0														
	start_TX	0														
	done_tx	0														
	start_rx	0														
	done_rx	0														
	out_RX	010...	00000000	01010011												
	load_decodifica	0														
	done_decodifica	0														

▪ Main FSM

La Main FSM è la macchina a stati finiti che gestisce il controllo a livello più alto dell'analizzatore logico. Essa è l'unica unità del circuito in grado di sentire il reset asincrono proveniente dall'esterno. Il datapath della Main Fsm è costituito da tutti i blocchi visti finora più il clock divider, usato per generare diverse frequenze di campionamento per il sampler. Sono stati aggiunti due FLIP FLOP usati come semafori: il primo per interfacciare correttamente il comando di start, pilotato dal trigger generator e sentito dalla memory interface e dalla Main Fsm, il secondo per eliminare dei problemi di timing nella lettura del segnale **MEM_READ**, sempre da parte della Main Fsm.

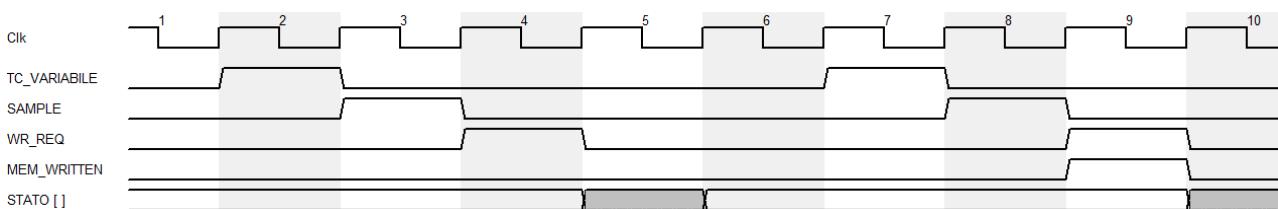
Per progettare la Main FSM non abbiamo studiato un timing globale, ma una serie di timing più piccoli in modo da far combaciare tutti i vincoli temporali dei vari blocchetti coinvolti di volta in volta. Nella prima fase, quella di scrittura, si utilizza la memory interface, il sampler, il clock divider, il trigger generator e la PC interface, quindi tutto il circuito. Questa è la fase più delicata perché deve funzionare a tutte le possibili frequenze di campionamento. Nella seconda fase invece sono coinvolti solamente PC interface e memory interface. Questa fase viene gestita usando praticamente due segnali di strobe, **OUTPUT VALID** e **READ_READY**, con cui questi blocchi si regolano il flusso di dati tra di loro.

Di seguito possiamo vedere l'evoluzione della Main FSM nella seconda fase, in particolare il meccanismo appena descritto, nel caso in cui si stia eseguendo anche l'ultima lettura:



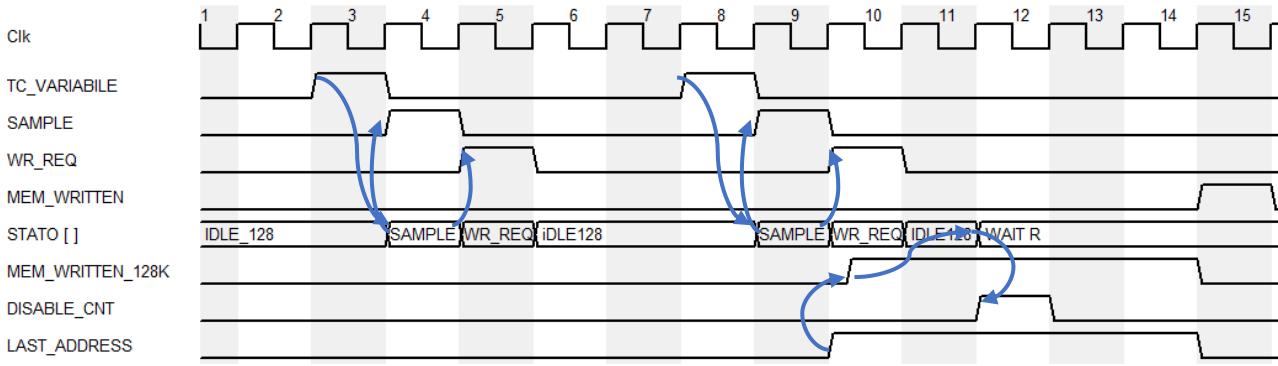
Abbiamo evidenziato con la freccia rossa come, una volta verificato in **READ_READY** che il flag **READ_COMPLETE** è attivo alto si vada direttamente in **RESET**

Per quanto riguarda la prima fase, dove tutti i componenti lavorano insieme, abbiamo dovuto affrontare dei problemi di interfacciamento con la memory interface in fase di scrittura, più precisamente nel rilevare il segnale **MEM_WRITTEN**, risolto mandando fuori da quest'ultima il segnale **LAST_ADDRESS**, che viene asserito quando la memoria deve scrivere l'ultimo dato (ha appena scritto il penultimo) e viene sentito solo quando si sta lavorando con il **PRESCALER** impostato a 0. Il **PRESCALER** viene mandato negato in AND con **LAST_ADDRESS** per ottenere il segnale **MEM_WRITTEN_10MHz**, che verrà poi controllato in uno stato specifico della FMS. Di seguito abbiamo tracciato il timing in cui evidenziamo il problema. Lo stato colorato di grigio è quello in cui si vuole controllare lo stato di **MEM_WRITTEN**.



Non solo **MEM_WRITTEN** viene perso, ma viene anche inviata una nuova richiesta di scrittura mentre la memoria non è in grado di sentirla.

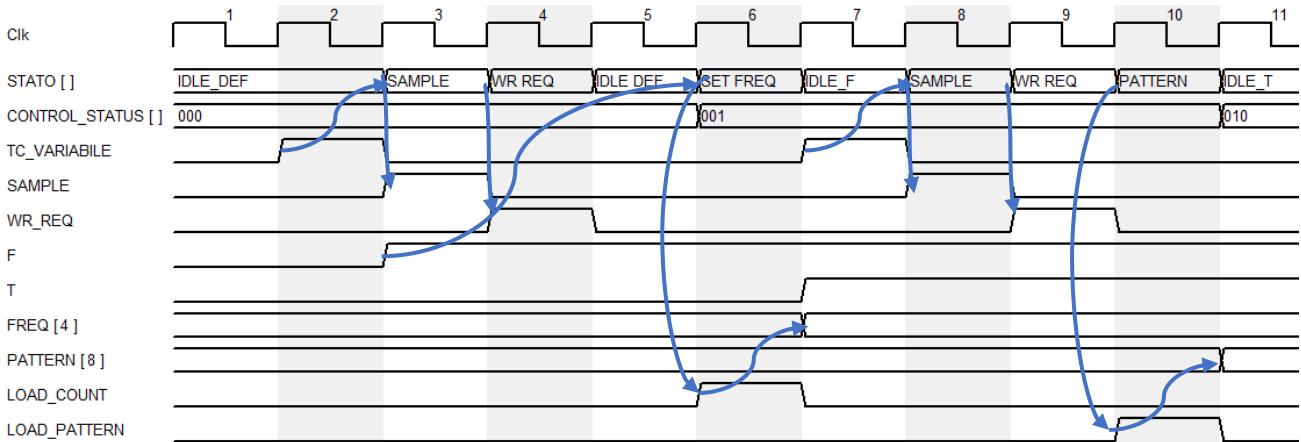
Questo ulteriore segnale di controllo si è rivelato necessario studiando il timing tra i vari componenti perché, per come sono stati progettati sampler e memory interface, quest'ultima con la gestione interna degli indirizzi di scrittura, accadrà che il segnale **MEM_WRITTEN**, asserito per un colpo di clock quando si finisce nello stato **WRITE_COMPLETE**, non verrà sentito dalla Main Fsm, poiché nel frattempo si troverà nello stato **S2_SAMPLE**. Avevamo detto che la scrittura nella memory interface richiede 4 cicli di clock e non crea nessun problema. Invece in questo caso particolare, solo quando si lavora alla frequenza di campionamento di 10MHz, è necessario mandare fuori in anticipo dalla memory interface il segnale che indica la scrittura completa della memoria, anche se questa deve ancora scrivere l'ultimo indirizzo. Stiamo di fatto anticipando il segnale **MEM_WRITTEN** di 5 colpi di clock, in modo che questo venga sentito prima che venga generato il successivo segnale di sample, in modo da poter spegnere il clock divider e interrompere il campionamento.



In questo timing è possibile osservare l'evoluzione temporale dei comandi di controllo in qualsiasi stato di IDLE. Il comportamento è sempre lo stesso, cambiano solo le condizioni per cui si salta da uno stato di IDLE all'altro.

Va precisato che questo vincolo temporale non è legato alla nostra scelta di far durare il ciclo di scrittura 4 colpi di clock, ma a come è stata pensata la memory interface. Abbiamo spostato la gestione degli indirizzi ed il controllo dello stato della scrittura e della lettura al suo interno per semplificare il comando ad alto livello. Questo problema però è stato risolto in modo piuttosto agevole usando solamente due porte AND, una interna alla Main Fsm e l'altra dentro alla memory interface. Avremmo anche potuto aggiungere un FLIP FLOP per tenere il segnale attivo alto fino a quando sarebbe stato necessario, ma avremmo dovuto aumentare i segnali di controllo interni, questa soluzione è dunque più economica.

Riportiamo anche un timing diagram con i passaggi tra i vari stati di IDLE:



Inoltre è stato inserito un registro da 3 bit, **control_status_register**, in cui viene salvata una sequenza di 3 bit indicante lo stato attuale della macchina che, per come è stata pensata da noi, utilizza sempre due stati per gestire campionamento e scrittura, ma si appoggia a diversi stati di IDLE, in cui deve poi necessariamente

tornare. Questo registro è necessario per avere un segnale di stato stabile che debba essere continuamente pilotato dalla macchina ad ogni colpo di clock.

Di default, prima che vengano inviati frequenza prescaler e start, l'analizzatore campiona i canali di ingresso e li scrive in memoria alla massima frequenza di campionamento. In seguito è possibile variare tutti i parametri inviando i comandi corretti dal PC.

La sequenza corretta con cui si inviano i comandi è la seguente: prima si invia F<n>, con n compreso tra 0 e 9 indicante il prescaler, poi si invia T<h><l>, dove h e l indicano rispettivamente i canali di ingresso da 1 a 4 e da 5 a 8 (h e l in esadecimale, quindi da 0 ad F), poi si invia il comando S ed infine R.

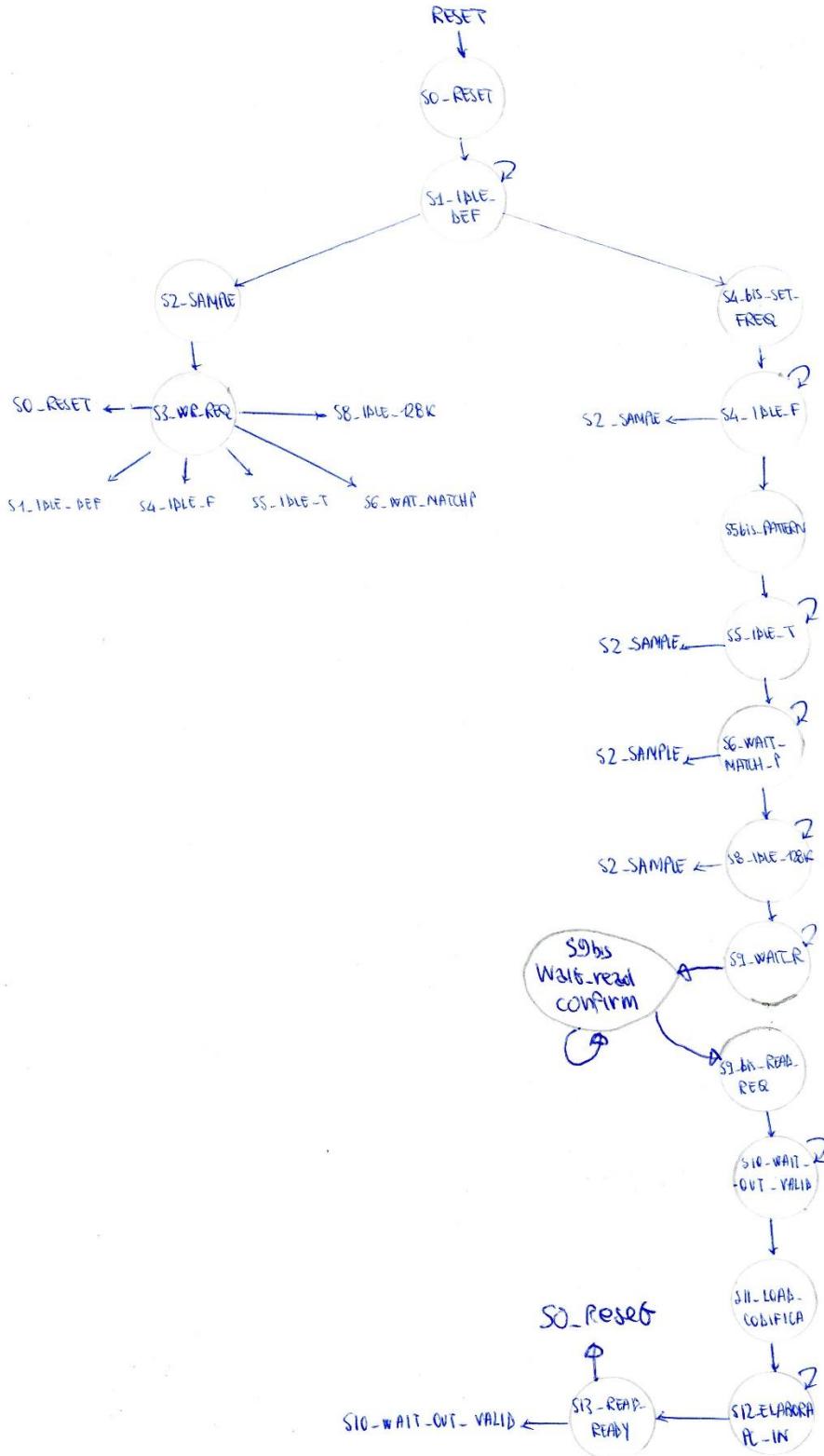
- **FSM e timing diagram**

- **S0_RESET:** questo è l'unico stato dell'analizzatore logico sensibile al reset asincrono. Quando si va in questo stato si pilota un secondo segnale di reset che viene sentito in modo sincrono dal resto del circuito;
- **S1_IDLE_DEF:** questo è lo stato di IDLE di default. Si campiona alla massima frequenza di campionamento possibile, aspettando che **TC_VARIABILE** vada a 1 per saltare in **S2_SAMPLE**. Se nel frattempo viene rilevato **Fm** a 1, che significa che l'utente esterno ha inviato il comando F seguito dal prescaler, si va in **S4bis_SET_FREQ**. Il flag **Fm** rimane attivo fino al reset del circuito ed è sentito solo in questo stato qui. Se si decide di cambiare la frequenza di campionamento occorre dunque azionare il reset ;
- **S2_SAMPLE:** ci si può arrivare in **S2_SAMPLE** da cinque stati di idle diversi (**S1_IDLE_DEF**, **S4_IDLE_F**, **S5_IDLE_T**, **S6_WAIT_MATCH_P**, **S8_IDLE_128K**). In questo stato si asserisce **SAMPLE** per avviare il campionamento e si salta in **S3_WR_REQ**;
- **S3_WR_REQ:** In questo stato si invia la richiesta di scrittura alla memory interface, portando a 1 **WR_REQ**, quindi a seconda del valore assunto da **CONTROL_STATUS** si torna in uno stato di IDLE diverso (000=**S1_IDLE_DEF**, 001= **S4_IDLE_F**, 010=**S5_IDLE_T**, 011=**S6_WAIT_MATCH_P**, 100=**S8_IDLE_128K**);
- **S4bis_SET_FREQ:** in questo stato si asseriscono **EN_CNT** e **LOAD_CNT**, che rispettivamente abilitano e caricano il **PRESCALER** in uscita dalla PC interface il **clock_divider** (in realtà già abilitato), poi si salta in;
- **S4_IDLE_F:** in questo stato si aggiorna lo status register, pilotando **status_mux** con l'apposito segnale di selezione **STATUS_SEL** = 001 e si asserisce **EN_STATUS_REG**. Da qui se **TC_VARIABILE** va a 1 si salta in **S2_SAMPLE**, se invece viene ricevuto il comando **Tm** più pattern di trigger, quindi **Tm** è alto, si va in **S5bis_PATTERN**, altrimenti si resta in questo stato;
- **S5bis_PATTERN:** si imposta **LOAD_PATTERN** a 1, caricando quindi il pattern di trigger nel trigger generator e si salta in **S5_IDLE_T**. Una volta che il pattern è stato caricato il trigger generator porterà a 1 il segnale **MATCH_TRIGGER** ogniqualvolta ci sia una corrispondenza tra **PATTERN_M** e il suo ingresso **INPUT_TRIGGER**. Tale segnale va sentito se e solo se è stato ricevuto il comando di start S ed è stato settato a 1 il flag **Sm**. Mandiamo quindi in AND **Sm** e **MATCH_TRIGGER**, da cui otteniamo il segnale **MATCH_PATTERN**, in modo da poter rilevare correttamente quando è il momento di andare nello stato in cui aspetteremo che vengano scritti in memoria i 128k campioni dopo l'evento di trigger;
- **S5_IDLE_T:** in questo stato si aggiorna lo status register, impostando **STATUS_SEL** = 010 e si asserisce **EN_STATUS_REG**. Se **TC_VARIABILE** va a 1 si salta in **S2_SAMPLE**, se invece viene ricevuto il comando S, quindi **Sm** è alto, si va in **S6_WAIT_MATCH_P**, altrimenti si resta in questo stato;
- **S6_WAIT_MATCH_P:** si aggiorna lo status register, pilotando **status_mux** in modo da avere **STATUS_SEL** = 011 e si asserisce **EN_STATUS_REG**. Se **TC_VARIABILE** è a 1 si salta in **S2_SAMPLE**.

In questo stato si aspetta che **MATCH_PATTERN** venga asserito. Se ciò accade si va in **S8_IDLE_128K**, altrimenti si resta in questo stato;

- **S8_IDLE_128K**: in questo stato si arriva dopo che si è verificata la condizione di trigger. Si aggiorna lo status register, pilotando **status_mux** in modo da avere **STATUS_SEL** = 100 e si asserisce **EN_STATUS_REG**. Se **TC_VARIABILE** è a 1 si salta in **S2_SAMPLE**. In questo stato si aspetta che vengano scritti 128k campioni e che quindi termini la fase di campionamento e scrittura a livello globale. Si controllano due segnali, **MEM_WRITTENe** **MEM_WRITTEN_10MHz**. Va precisato che questi due segnali non potranno mai essere attivi alti contemporaneamente. Se uno dei due segnali è a 1 si va in **S9_WAIT_R**;
- **S9_WAIT_R**: in questo stato si porta a 1 per disabilitare il **clock_divider** e si attende che il flag **Rm** indicante la richiesta di lettura della memoria venga asserito. Se ciò si verifica si va in **S9bis_READ_REQ**, altrimenti si resta in **S9bis_WAIT_READ_CONFIRM**;
- **S9bis_WAIT_READ_CONFIRM**: prima di iniziare ad inviare i dati sulla linea seriale bisogna aspettare che la PC interface invii al PC i caratteri O\n\r, per indicare la corretta ricezione del comando di lettura. Lo scopo di questo stato è aspettare che finisca la trasmissione di questi caratteri, controllando se viene asserito **BLOCK_SENT**, che come visto nella PC interface, indica la fine di un ciclo in cui sono stati inviati 3 o 10 caratteri sulla linea seriale. Se si verifica questa condizione si va in **S9bis_READ_REQ**, altrimenti si resta in questo stato;
- **S9bis_READ_REQ**: si invia la richiesta di scrittura alla memory interface mettendo **READ_REQ** a 1 e poi si va in **S10_WAIT_OUT_VALID**;
- **S10_WAIT_OUT_VALID**: richiestain questo stato si aspetta che la memory interface legga un dato dalla SRAM e asserisca **OUTPUT_VALID**. In questo caso si va in **S11_LOAD_CODIFICA**, altrimenti si rimane in **S10_WAIT_OUT_VALID**;
- **S11_LOAD_CODIFICA**: si asserisce **LOAD_PC_INTERFACE** e si va in **S12_ELABORA_PC_IN**;
- **S12_ELABORA_PC_IN**: si aspetta che venga asserito **BLOCK_SENT**, che indica la fine della trasmissione degli 8 caratteri indicanti lo stato dei canali di ingresso al momento del campionamento, più \n\r. Quando ciò accade si va in **S13_READ_READY**, altrimenti si aspetta in questo stato che la UART TX finisca di inviare tutto;
- **S13_READ_READY**: si asserisce **READ_READY** per far caricare alla memory interface il dato successivo dalla SRAM e si va in **S10_WAIT_OUT_VALID**.

Di seguito illustriamo il pallogramma della Main FSM:



Descritto il funzionamento della macchina passiamo adesso alla simulazione.

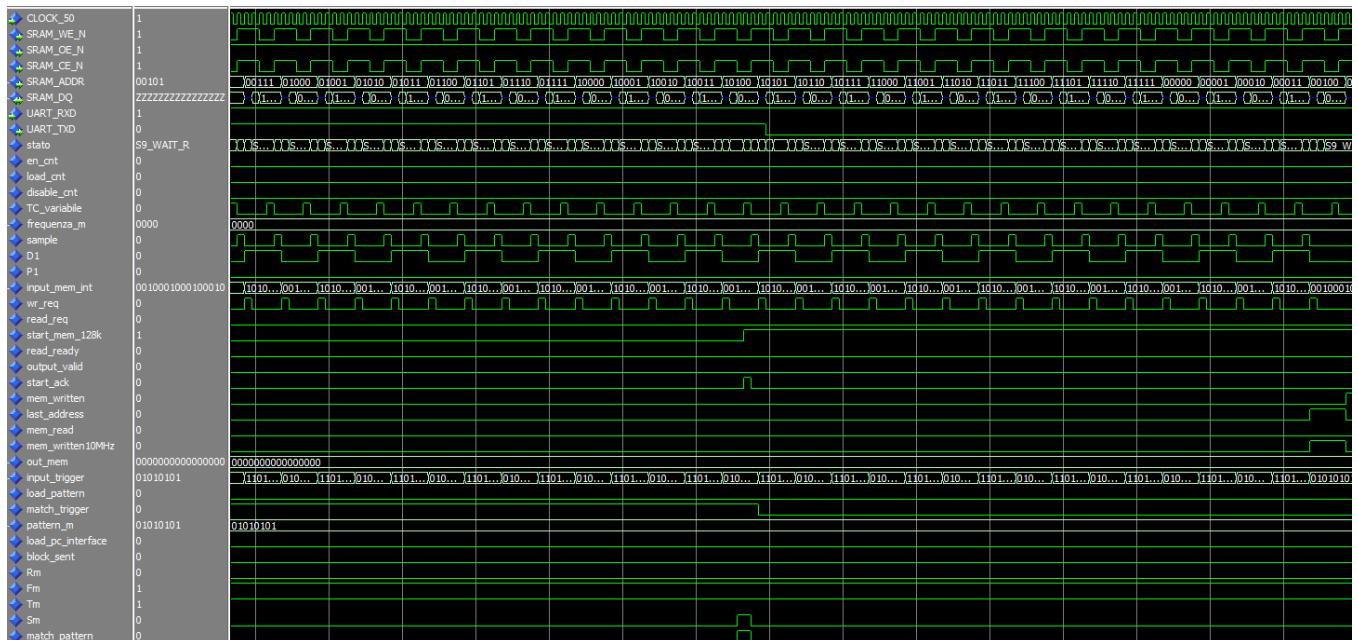
Abbiamo simulato l'analizzatore logico utilizzando una SRAM fittizia con 32 locazioni di memoria e parallelismo 16bit, modifica in modo da assomigliare il più possibile, per quanto riguarda la gestione dei comandi di lettura e scrittura, a quella montata sulla DE2.

Abbiamo generato il segnale di ingresso usando un'onda quadra a 5MHz collegata al canale uno, tutti gli altri canali sono stati fissati a 1 o a 0.

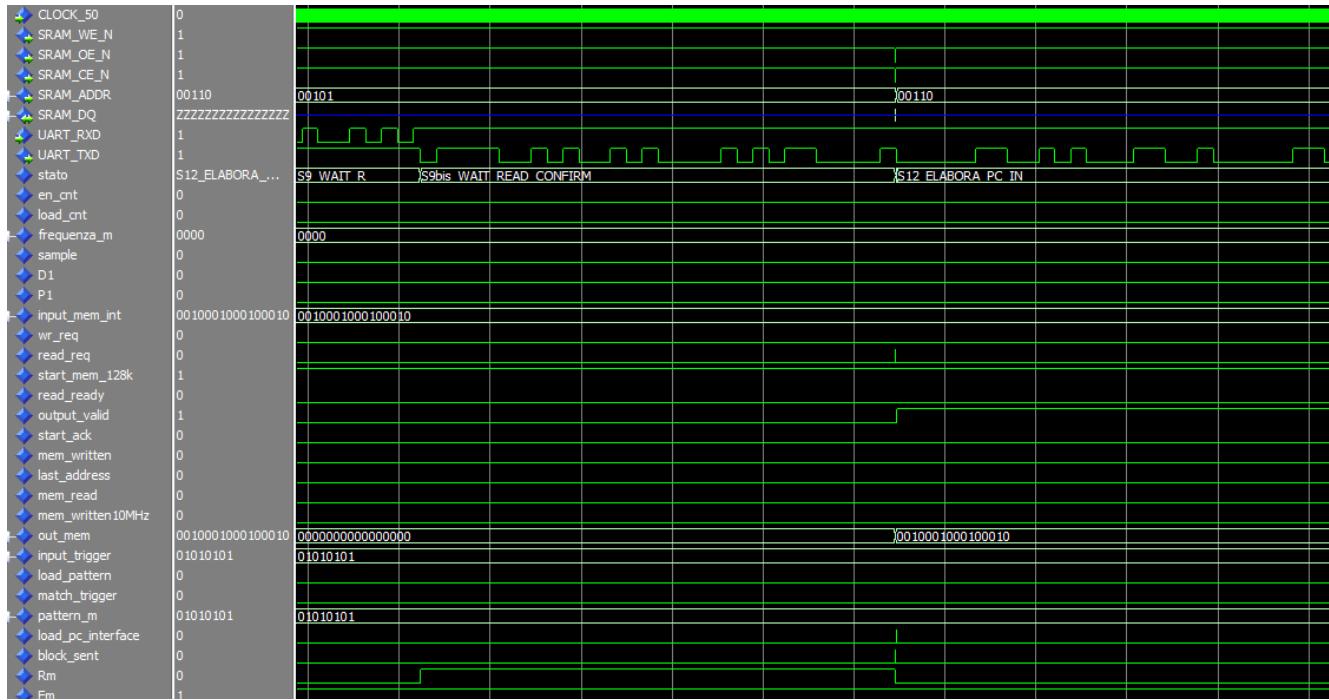
Abbiamo impostato la massima frequenza di campionamento e inviato un pattern di trigger adatto. Quindi abbiamo inviato il segnale di start che ha abilitato la generazione del segnale di trigger. Di seguito si può vedere l'abilitazione di tale segnale, concomitante in questo con l'evento di trigger, a causa della coincidenza fortuita con l'onda quadra inviata sul canale 1.

È possibile notare il segnale di acknowledge **START_ACK** pilotato dalla memory interface che resetta **MATCH_PATTERN**.

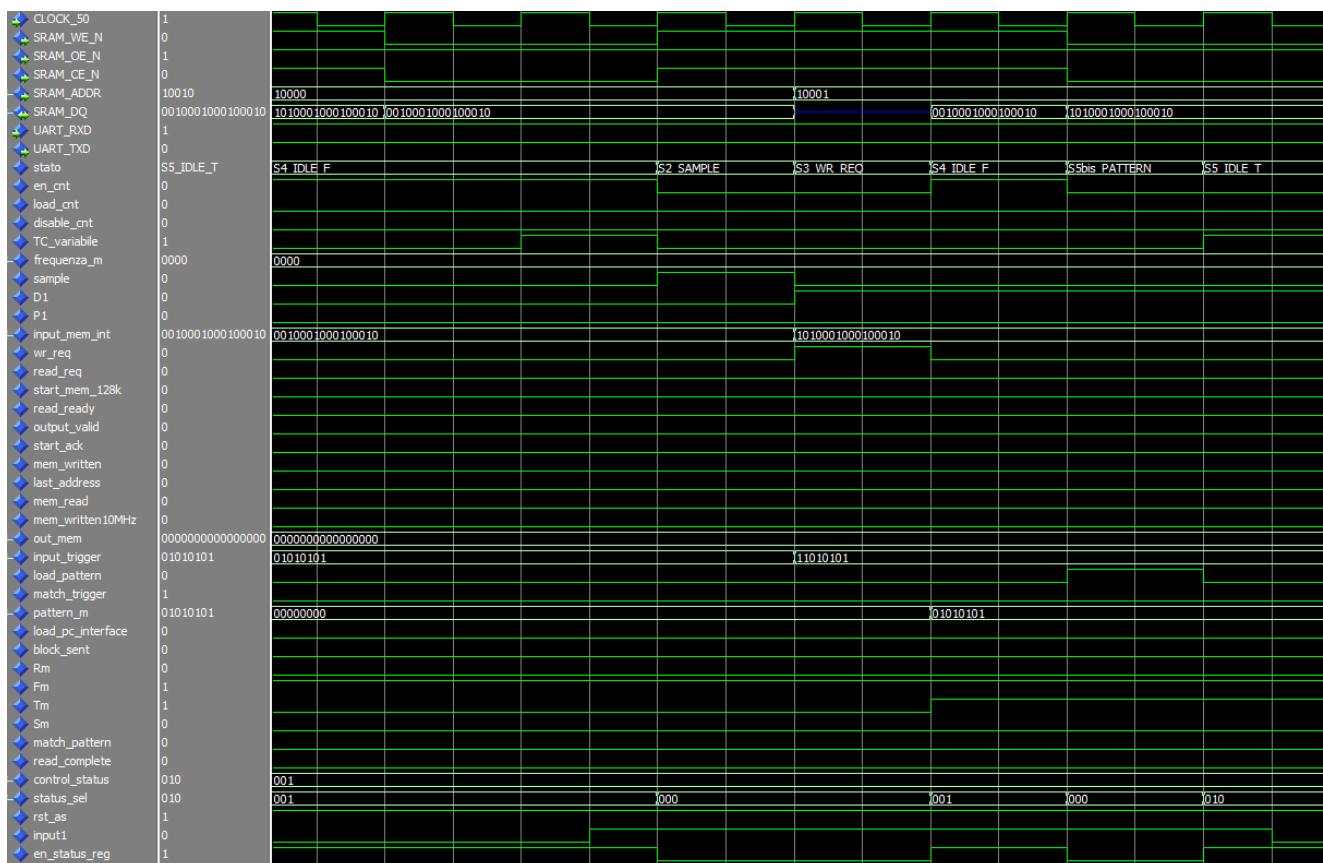
Si può anche notare come l'analizzatore salvi correttamente ancora 16 campioni, ovvero metà della memoria, prima di interrompere la scrittura e andare nello stato di attesa del comando di lettura.



Di seguito mostriamo l'evoluzione del circuito una volta arrivata la richiesta di lettura. **RM** viene rilevato alto a 1, poi viene resettato, quindi il segnale **READ_REQ** viene asserito e la memory interface inizia a caricare i dati dalla memoria. Notare come in **S9bis_WAIT_READ_CONFIRM** la uart tx invii al PC i caratteri **O\n\r** in seguito alla corretta ricezione del comando **R**.



Infine mostriamo come, tra un campionamento e l'altro, venga settato il pattern di trigger e caricato dentro il trigger generator. Si può notare anche in questo screen l'evoluzione del registro control status.

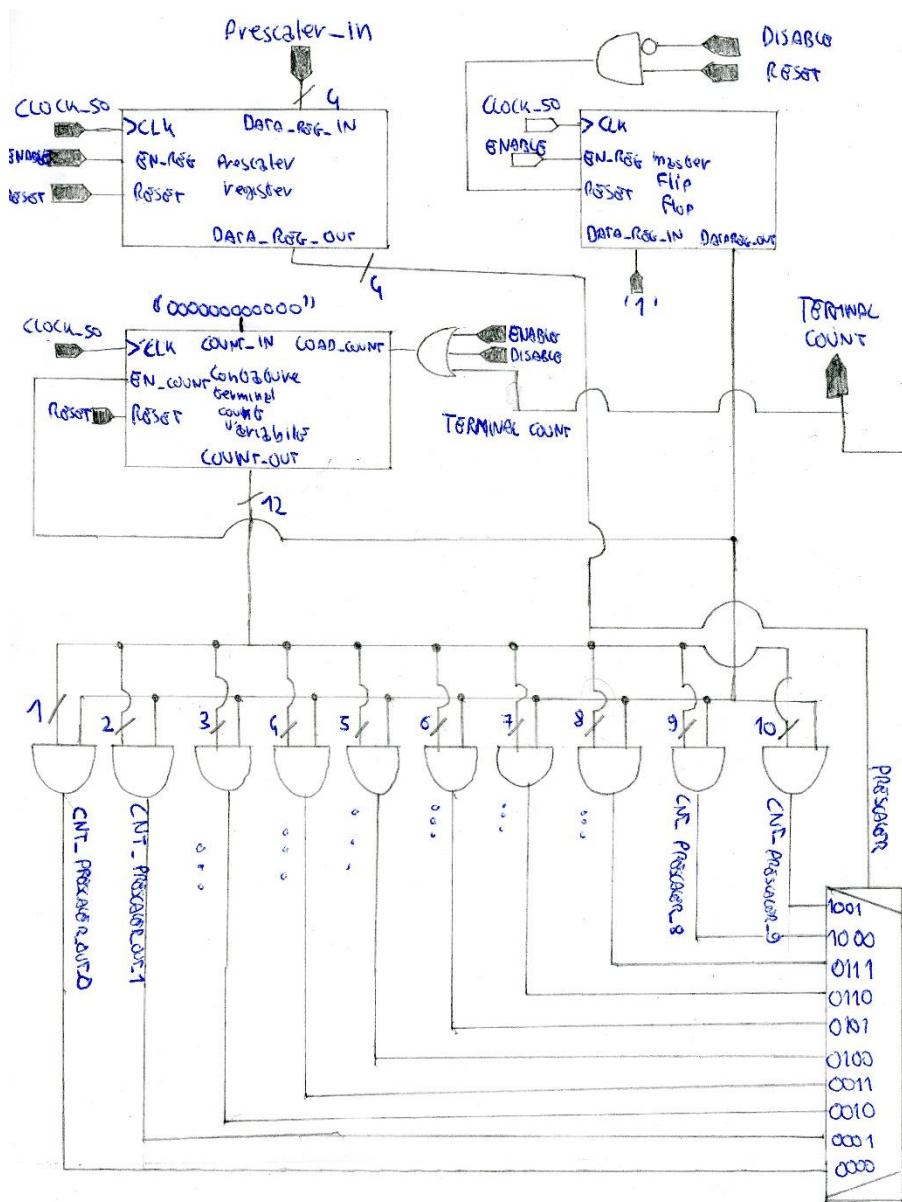


• Clock divider

Il clock_divider è un sotto-sistema composto da due registri, un contatore da 12bit, una serie di porte AND e un MUX per selezionare il terminal count desiderato. Viene utilizzato per stabilire l'istante corretto in cui abilitare il sampler, in base al clock da 50MHz ed il prescaler inserito dall'utente per scegliere la frequenza di campionamento desiderata.

Non dispone di una propria control unit, poiché viene controllato direttamente dalla Main FSM ed il suo scopo è quello di fornire degli impulsi di ampiezza pari ad un periodo del clock. Il tempo tra due impulsi rappresenta la frequenza ottenuta dividendo il clock per un prescaler, definito come 2^{pr} , $0 \leq pr \leq 9$. Dunque la frequenza con cui viene originato questo impulso è $freq = \frac{10MHz}{2^{pr}}$.

○ Datapath



Analizziamo il datapath:

[errata corrige: il segnale di ENABLE non entra tramite un OR in LOAD_COUNT. Esiste un segnale specifico chiamato LOAD_COUNT asserito dall'esterno]

Il clock divider viene pilotato dal segnale **ENABLE**, che attiva il contatore da 12bit e dal segnale **LOAD_COUNT** che abilita il registro da 4bit in cui si salva il valore del **PRESCALER**, in uscita dalla PC interface. Quindi l'uscita di questo registro controlla il MUX che seleziona l'uscita di una delle 10 porte AND (con ingressi multipli). I valori del **PRESCALER** validi sono compresi tra 0 e 9. Se viene inviato un valore superiore a 9 si utilizza la frequenza di default che è 10MHz.

Il segnale in uscita, cioè **TERMINAL_COUNT** viene inviato in loop e viene utilizzato per resettare il contatore in modo sincrono.

In questo modo abbiamo ottenuto tutte le frequenze di campionamento desiderate, che abbiamo poi misurato con l'oscilloscopio, verificando una tolleranza molto bassa della frequenza del segnale

TERMINAL_COUNT, intesa come l'inverso dell'intervallo temporale tra due impulsi successivi, rispetto a quella richiesta dalle specifiche.

I segnali di **ENABLE** e **DISABLE** vanno tenuti alti per un colpo di clock per abilitare o disabilitare la generazione di questi impulsi con la frequenza indicata dal **PRESCALER**, tuttavia possono anche essere tenuti alti per un tempo indefinito senza che questo possa inficiare il funzionamento del circuito. Il segnale di **DISABLE** resetta anche il contatore.

Le AND hanno questi ingressi, omessi nel datapath per non complicare troppo il disegno del circuito, l'uscita viene inviata al MUX, rappresentata dal vettore **CNT_PRESCALER_OUT**:

CNT_PRESCALER_OUT(0)<=MASTER_ENABLE AND CNT3_OUT(2); **-10 MHZ**

CNT_PRESCALER_OUT(1)<=MASTER_ENABLE AND CNT3_OUT(3) AND CNT3_OUT(0); **-5MHZ**

CNT_PRESCALER_OUT(2)<=MASTER_ENABLE AND CNT3_OUT(4) AND CNT3_OUT(1) AND CNT3_OUT(0); **-2.5MHZ**

CNT_PRESCALER_OUT(3)<=MASTER_ENABLE AND CNT3_OUT(5) AND CNT3_OUT(2) AND CNT3_OUT(1) AND CNT3_OUT(0); **--1.25MHZ**

CNT_PRESCALER_OUT(4)<=MASTER_ENABLE AND CNT3_OUT(6) AND CNT3_OUT(3) AND CNT3_OUT(2) AND CNT3_OUT(1)
AND CNT3_OUT(0); **--625KHZ**

CNT_PRESCALER_OUT(5)<=MASTER_ENABLE AND CNT3_OUT(7) AND CNT3_OUT(4) AND CNT3_OUT(3) AND CNT3_OUT(2) AND CNT3_OUT(1) AND CNT3_OUT(0); **--312.5KHZ**

CNT_PRESCALER_OUT(6)<=MASTER_ENABLE AND CNT3_OUT(8) AND CNT3_OUT(5) AND CNT3_OUT(4) AND CNT3_OUT(3) AND CNT3_OUT(2) AND CNT3_OUT(1) AND CNT3_OUT(0); **--156.25KHZ**

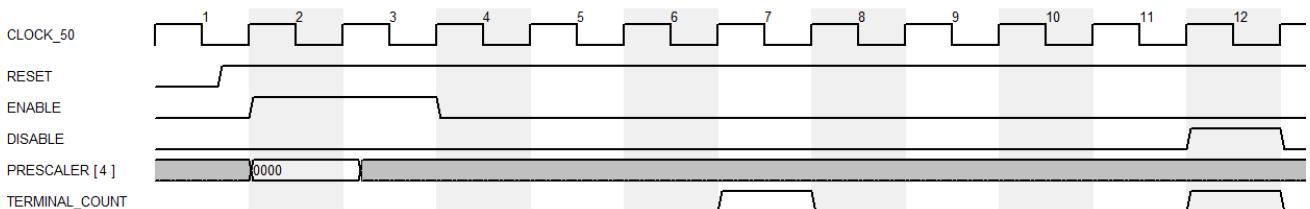
CNT_PRESCALER_OUT(7)<=MASTER_ENABLE AND CNT3_OUT(9) AND CNT3_OUT(6) AND CNT3_OUT(5) AND CNT3_OUT(4) AND CNT3_OUT(3) AND CNT3_OUT(2) AND CNT3_OUT(1) AND CNT3_OUT(0); **--78.125KHZ**

CNT_PRESCALER_OUT(8)<=MASTER_ENABLE AND CNT3_OUT(10) AND CNT3_OUT(7) AND CNT3_OUT(6) AND CNT3_OUT(5) AND CNT3_OUT(4) AND CNT3_OUT(3) AND CNT3_OUT(2) AND CNT3_OUT(1) AND CNT3_OUT(0); **--39.063KHZ**

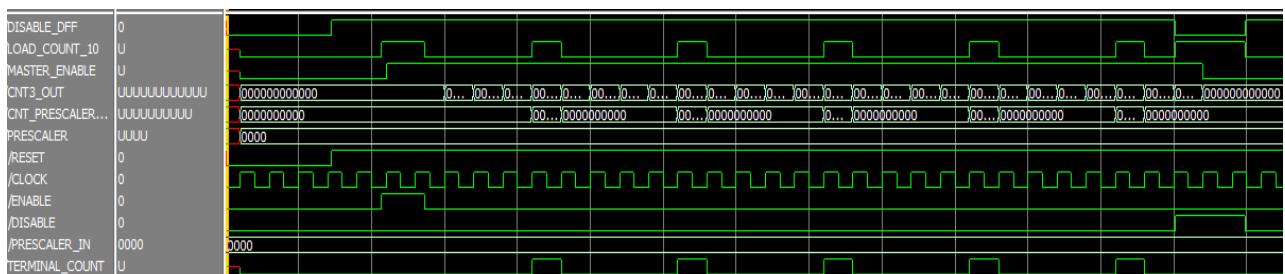
CNT_PRESCALER_OUT(9)<=MASTER_ENABLE AND CNT3_OUT(11) AND CNT3_OUT(8) AND CNT3_OUT(7) AND CNT3_OUT(6) AND CNT3_OUT(5) AND CNT3_OUT(4) AND CNT3_OUT(3) AND CNT3_OUT(2) AND CNT3_OUT(1) AND CNT3_OUT(0); **--19.531KHZ**

- **Timing diagram**

Il timing diagram è piuttosto semplice: si deve asserire **ENABLE** ed inviare simultaneamente un valore del prescaler compreso tra 0 e 9.



Questo timing è valido ovviamente per tutti i possibili valori del **PRESCALER**, cambia solo il tempo tra due fronti successivi del **TERMINAL_COUNT**. Di seguito riportiamo anche l'esito della simulazione eseguita su MODELSIM con lo stesso valore del **PRESCALER**.



▪ Test in laboratorio

Riportiamo brevemente le modalità con cui abbiamo eseguito i test in laboratorio sulla DE2 dei vari componenti dell'analizzatore.

1. UART TX: abbiamo collegato l'oscilloscopio ad un pin del connettore seriale della DE2 e abbiamo verificato sia la trasmissione corretta degli 8 bit di payload, più start bit e stop bit, oltre che al tempo di bit, in modo da verificare che stessimo rispettando il baud rate assegnatoci. Per generare la sequenza da inviare sulla linea seriale e gestire il comando di inizio trasmissione abbiamo usato gli switch presenti sulla board. In seguito abbiamo collegato la linea seriale alla porta COM del PC e tramite il programma Terminal abbiamo verificato la corretta trasmissione (e ricezione da parte del PC) dei caratteri 1, 0, O, K, x, \n, \r, sia inviandone uno singolarmente, sia inviandoli in modo continuo, per escludere la possibilità che potessero verificarsi errori di framing;
2. UART RX: abbiamo collegato la DE2 al PC e abbiamo usato lo stesso programma del punto precedente in trasmissione. Abbiamo verificato la corretta ricezione del carattere visualizzando i bit su cui era codificato tramite i LED montati sulla scheda;
3. UART RX in loop con UART TX: infine abbiamo collegato internamente i pin UART_TXD e UART_RXD e, comandando i blocchi usando gli switch, abbiamo verificato il corretto funzionamento osservando lo stato dei LED, indicanti gli 8 bit di dato ricevuti;
4. TRIGGER_GENERATOR: per testare questo blocco abbiamo semplicemente collegato l'uscita di un contatore al suo ingresso, impostando il pattern di trigger tramite gli switch sulla board, per poi verificare tramite i LED lo stato del segnale **MATCH_TRIGGER**, che indica il rilevamento della condizione di trigger;
5. CLOCK_DIVIDER: per testarne il funzionamento abbiamo usato gli switch per pilotare i segnali di abilitazione, disabilitazione e prescalet. Tramite l'oscilloscopio abbiamo verificato la corretta generazione del terminal count alla frequenza desiderata, secondo le modalità descritte nel punto dedicato a questa unità;
6. Memory interface: abbiamo innanzitutto ridotto la dimensione del buffer circolare da 256k a 32 locazioni, per poi generare i segnali di **WR_REQ**, **RD_REQ** e **START** usando i pulsanti. Tramite la macchina a stati interna abbiamo potuto fare in modo che ad una pressione del pulsante corrispondesse una sola richiesta di lettura, scrittura o start. Infatti essendo il clock interno a 50MHz, una sola pressione del pulsante collegato a **WR_REQ** avrebbe riempito interamente la memoria. Abbiamo inserito i 16 bit di dato usando gli switch ed infine abbiamo letto il contenuto, confrontandolo con i dati immessi. Non abbiamo rilevato nessun errore e abbiamo potuto verificare il corretto funzionamento del buffer circolare. La scrittura avviene con le modalità e le tempistiche descritte nel punto relativo a questa unità. Abbiamo quindi verificato che tutti i vincoli temporali imposti dalla SRAM venissero rispettati;
7. PC interface: abbiamo collegato il PC alla DE2 e usato lo stesso software dei punti precedenti. Abbiamo verificato la corretta ricezione di tutti i comandi, visualizzando sui LED lo stato dei flag indicanti il set della frequenza ed il valore del prescaler, il set del trigger con gli 8 bit indicanti il pattern, lo start ed infine la richiesta di lettura. Abbiamo potuto anche testare l'invio dei caratteri O e K (seguiti dai caratteri \n e \r), in seguito all'inserimento di comandi corretti o sbagliati. Prima di riuscire a far funzionare correttamente la PC interface abbiamo avuto dei problemi dovuti ai ritardi di propagazione dei segnali che hanno, di fatto, impedito di ottenere lo stesso comportamento del circuito visto durante i testbench. Questo perché con MODELSIM non è possibile verificare i ritardi combinatori e di propagazione, la simulazione viene infatti eseguita ad "eventi" (commutazione di un segnale, etc) e non propagando i segnali lungo il circuito.

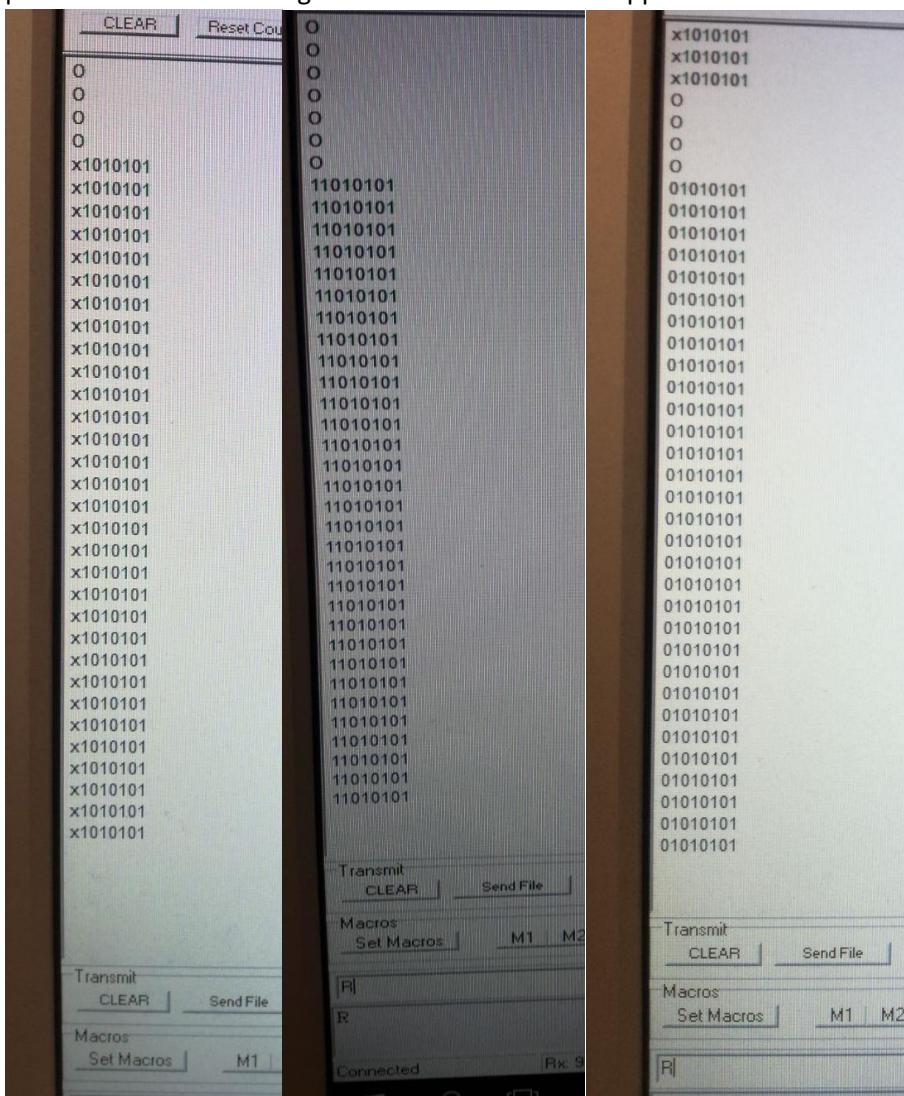
Abbiamo deciso di eliminare la control unit della codifica ASCII e pilotarla interamente tramite la PC interface.

Abbiamo quindi aggiunto due segnali di controllo nella PC interface, semplificando di molto l'interfacciamento tra questi due blocchi.

8. Analizzatore logico: per testare il circuito finale abbiamo collegato la DE2 al PC come descritto nei punti precedenti, abbiamo impostato 7 degli 8 canali di ingresso ad 1 o 0 fissi, alternando zeri e uni per verificare che stessimo scrivendo e leggendo effettivamente i dati dalla memoria.

Abbiamo quindi collegato il canale 1 dell'analizzatore logico ad un generatore di funzioni arbitrarie (PICOSCOPE 2608B) dal quale abbiamo inviato onde quadre col duty cycle del 100% oppure dello 0% per verificare il corretto campionamento degli 1 e degli 0.

In seguito abbiamo disegnato le 4 forme d'onda ricalcanti i 4 tipi di glitch visti nel paragrafo del sampler e abbiamo verificato la corretta rilevazione degli stessi. Riportiamo di seguito alcune foto per far vedere come vengono visualizzati i dati sull'applicazione Terminal.



La dimensione del buffer circolare è di 32 locazioni per 16 word. Abbiamo infatti mantenuto lo stesso parallelismo (necessario per il funzionamento del circuito) ma ridotto il numero di campioni salvati e visualizzati, in modo da poter testare agevolmente l'analizzatore. In questo caso non abbiamo riscontrato nessun problema, il circuito ha funzionato correttamente fin da subito.