dadduni

UART TX... VERSIONE CORRETTA!

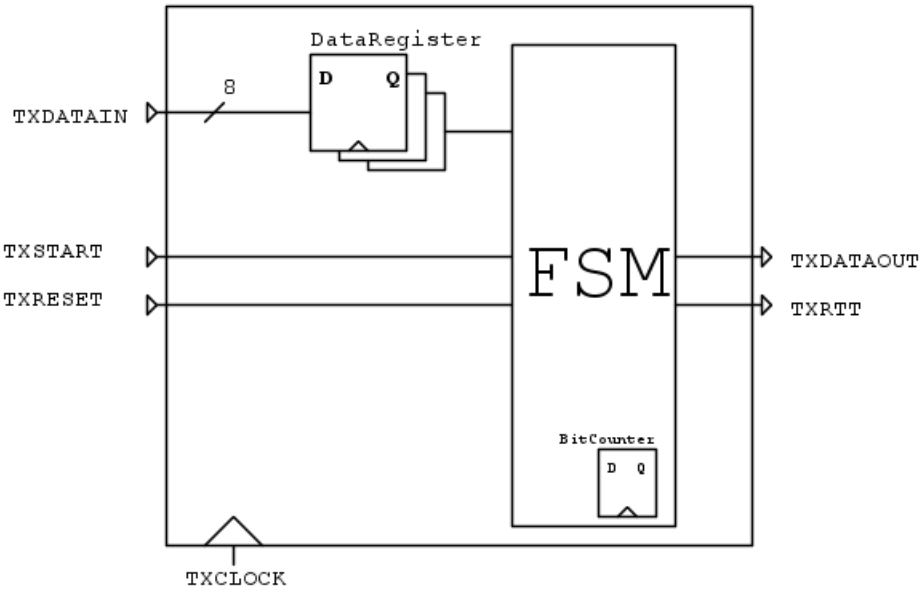
30 September 2018

Introduzione

Questo articoletto è una naturale prosecuzione del procedente presente sul mio blog. L'avevo già detto nel precedente: non ho tutte le competenze necessarie e l'esperienza adatta e infatti ho fatto una marea di errori. Tutto sommato funzionava anche ma fortunatamente Boiler (che non rigrazierò mai abbastanza) ha sottolineato gli errori e le criticità e mi ha dato modo di studiare vari aspetti che non avevo neanche preso in considerazione. Questa è una distorsione piuttosto comune in cui si casca, e io ci sono finito con mani e piedi alla grande: "non si sa quel che non si sa". Ho così deciso (previa conferma di Boiler) di scrivere una specie di errata corrige in cui riporto le spiegazioni che lui ha dato, e fare un confronto tra il codice "vecchio" e quello corretto, confrontando anche i circuiti risultati che vengono fuori dai due codici.
Disclaimer:
Non prendete quello che scrivo come farina del mio sacco: non sto dando spiegazioni a nessuno perchè non ne sono capace. Ho solo attinto da fonti diverse guidato da un utente infinitamente più esperto di me. Scrivo solo per correggere il tiro e magari far uscire una conversazione altrettanto interessante come la prima, lo faccio solo per imparare e non per insegnare. Se imparate da me imparate male: non vi conviene!
End Disclaimer

La entity

Nel precedente articolo ho sorvolato completamente questo aspetto che però non è da sottovalutare. Design digitale non è programmazione. Qui si scrive codice che non viene eseguito: si sta creando un componente e bisogna solo spiegare al sintetizzatore quali collegamenti vogliamo vengano effettuati. Quindi non può mancare uno schema "top level" in cui si prefissano almeno gli input e gli output del nostro componente e il numero di elementi di memoria e di elementi combinatori che ci saranno.



Come si vede non sono specificate le implementazioni dei singoli blocchi e non si è entrati in nessun dettaglio. Però ci sono tutti i segnali necessari, i registri per fare lo storage dei dati e una grossa macchina a stati finiti (FSM) che pilota il tutto. Nel progetto ho deciso che il circuito non avrà al suo interno nessun divisore di clock, componente che sarà esterno e che si occuperà di calcolare il clock necessario per un dato baudrate. E' tutto sufficientemente autoesplicativo: ingresso parallelo dei dati a 8 bit, segnale di start, reset, clock, output seriale e ReadyToTrasfer (RTT) per una eventuale segnalazione esterna. Si nota una stranezza: dentro la macchina a stati c'è un contatore? Si, perchè un contatore è di per sè una macchina a stati fatta e finita: conta 2^n stati diversi e ha una legge di transizione da uno stato all'altro. Quindi la macchina a stati grande ha al suo interno una macchinina a stati più piccola che si occupa solo di contare il numero di bit uscenti. Volendo usare una sola macchina a stati "pura" bisognava creare uno stato per ogni bit da contare e non sarebbe cambiato nulla. Solo così è più intuitivo e malleabile in futuro. Da questo disegno estraiamo immediatamente la entity:

```
entity UART_TX is
    Generic(        bit_num: integer := 8    );
    Port (
        TXCLOCK: in std_logic;
        TXSTART: in std_logic;
        TXRESET: in std_logic;
        TXDATAIN: in std_logic_vector ( bit_num -1 downto 0 );
        TXDATAOUT: out std_logic;
        TXRTT: out std_logic
          );
end UART_TX;
```

Possiamo estrarre un'altra importante informazione ossia il numero di elementi di memoria. Sappiamo già in partenza che dobbiamo avere 3 registri: contatore numero di bit, buffer per i dati in ingresso e registro di stato della macchina a stati. Suggerisce Boiler di prestare ben attenzione a questo conteggio perchè funge da verifica sul codice scritto, alla fine del progetto guardando lo schematico dobbiamo avere esattamente il numero di registri previsto altrimenti c'è qualcosa che non è andato come avrebbe dovuto.

Nelle puntate precedenti...

Riporto velocemente il codice del precedente articolo. E' un codice testato e funzionante ma che non rispetta alcune regole di buona progettazione.

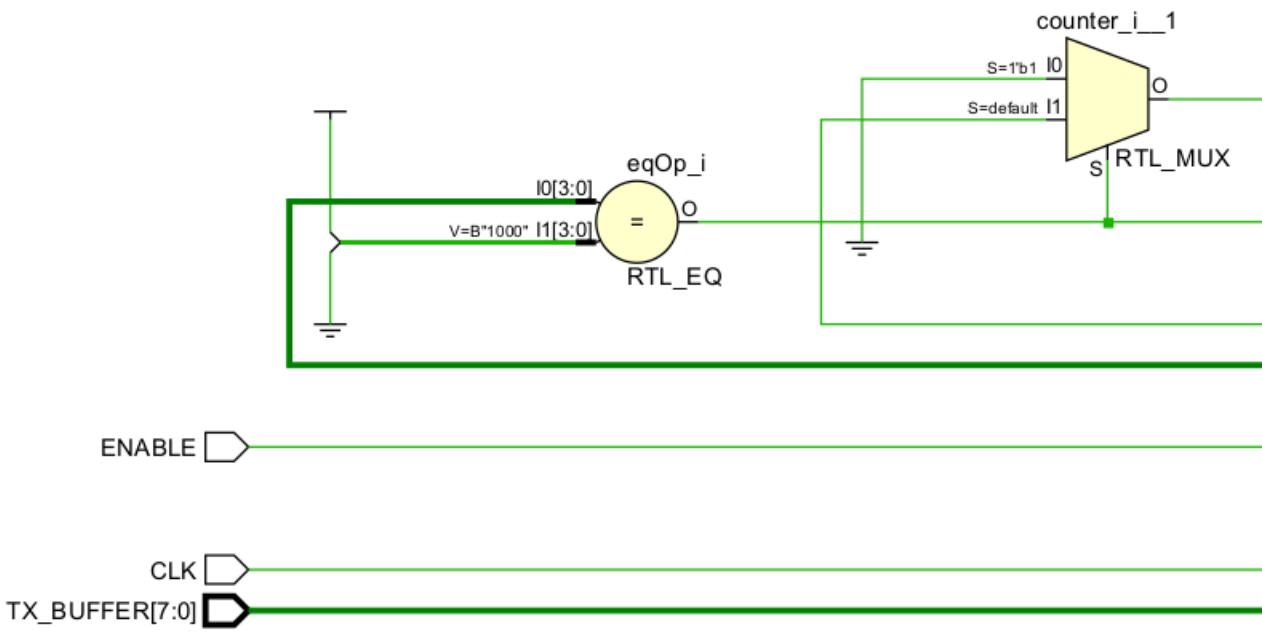
```
signal ready: std_logic:= '1';
signal buff: std_logic_vector( 7 downto 0) := (others => '0');
signal counter: std_logic_vector (3 downto 0) := "0000";
```

begin

```
process(CLK) begin
    if(rising_edge(CLK)) then
        if(ENABLE= '1' and ready='1') then
                -- carica start bit, prendi registo, porta basso ready
                TX_DATA_OUT <= '0';
                buff<= TX_BUFFER;
                ready <= '0';
        elsif(ready = '0') then
            if(counter = "1000") then
                 TX_DATA_OUT <= '1';
                 counter<= counter +1;
            elsif(counter= "1001") then
                counter <= "0000";
                ready <= '1';
                              else
                counter <= counter +1;
            --TX_DATA_OUT <= buff(7 - to_integer(unsigned(counter)));
            TX_DATA_OUT <= buff(to_integer(unsigned(counter)));
            end if;
        end if;
    end if;
end process;
```

Il problema è principalmente uno (ma non solo uno). Sono mischiati elementi sequenziali e combinatori in un unico processo che fa tutto. Il sintetizzatore è piuttosto bravo a "capire quello che volevo dire" e quindi questa volta ha interpretato il mio codice in maniera sensata capendo

che counter è un registro di stato di una macchina a stati **ma** per come è scritto il codice il processo viene elaborato al fronte di clock e quando scrivo "counter<= counter+1" il risultato è piuttosto incerto perchè nello stesso momento viene elaborato cosa assegnare all'uscita e gli viene assegnato. In una logica procedurale tipo il C questo non porterebbe problemi perchè viene prima calcolato il valore da assegnare e poi viene assegnato. Su una FPGA non esiste "elaborato" in un istante: ci sono blocchi combinatori e blocchi sequenziali. L'assegnazione ad un registro è un evento sequenziale invece la somma è un evento combinatorio e vanno fatti in processi differenti che tutelino queste differenze. Il secondo problema è che nel vecchio codice non esiste un solido stato di reset che porti la macchina ad uno stato definito e stabile, le assegnazioni iniziali sono state fatte con l'operatore := che è un assegnazione che al compilatore va bene (il compilatore lavora come il C) ma che potrebbe generare risultati incerti in fase di sintesi. Il risultato è il seguente. Click to zoom



schema vecchio.png

Si notano immediatamente un paio di cose:

- Ci sono registri counter e buffer: great! Il sintetizzatore ha capito cosa volevo dire

- Ma quanti mux ci sono? questo è il risultato dell'utilizzo del costrutto if invece del case. Non cambia nulla, solo "ordine" grafico.
- Quindi va tutto bene? Bhe, quasi. Vediamo la comparsa di due registri extra che non erano richiesti in uscita! Poco male, oltre ad ingrandire un po' il circuito il danno è minimo. Ma non erano richiesti e sono il risultato di un modo troppo poco rigoroso di impostare il circuito

Instanziare un registro

C'è un solo modo giusto per instanziare un registro in VHDL. Sembra scomodo ma pensandoci 5 secondi è in effetti l'unico modo ragionevole di farlo. Un Flip Flop ha due segnali: uno di ingresso e uno di uscita. Al colpo di Clock l'Uscita diventa uguale all'Ingresso. Tutto qui. Bisogna esclusivamente copiare in codice quanto appena detto ed è banale. E' vero che bisognerà dichiarare due segnali per ogni registro e questo può sembrare scomodo ma in effetti è così che funziona un FF ed è così che dobbiamo descriverlo, niente di più e niente di meno.

```
signal FFIn, FFOut: std_logic;
process( CLK ) begin
    if rising_edge(CLK) then
        FFOut<= FFIn;
    end if;
end process;
```

una macchina a stati... in più colpi

Questo titolo non l'ho inventato io, è su un famoso manuale di progettazione RTL in VHDL. Il motivo è che abbiamo due modi altrettanto corretti di descrivere una macchina a stati in VHDL e il tutto avviene fondamentalmente in 3 processi:

- aggiornare i registri al clock
- calcolare il next state partendo da present state e dagli ingressi
- calcolare l'output partendo da present state e ingressi (Moore o Mealy)

Questo può esser fatto in VHDL con 3 processi separati di cui uno sequenziale (Aggiornamento FF) e due combinatori e sarebbe il metodo in "3 colpi" (three shots). Dal momento che la cosa importante è tenere separati processi sequenziali da combinatori e che due processi combinatori convivono abbastanza pacificamente, possiamo raggrupparli e quindi avere solo due blocchi (two shots).

Il nuovo codice

I registri:

```
type state_type is (idle, start, stop, transmitting);
signal fsmxPS, fsmxNS: state_type;
signal BitCounterxPS, BitCounterxNS: integer range 0 to bit_num-1;
signal DataRegisterxPS, DataRegisterxNS: std_logic_vector( bit_num-1 downto 0 );
```

L'aggiornamento dei registri con il reset: il present state diventa il next state calcolato da un'altra parte

```
memory_sequential_process: process ( TXRESET, TXCLOCK ) begin
    if (TXRESET = '1') then --asyincronus reset
         DataRegisterxPS <= (others => '0');
         BitCounterxPS <= 0;
         fsmxPS <= idle;
     elsif rising_edge ( TXCLOCK ) then
         DataRegisterxPS <= DataRegisterxNS;
         BitCounterxPS <= BitCounterxNS;
         fsmxPS <= fsmxNS;
     end if;
end process;
```

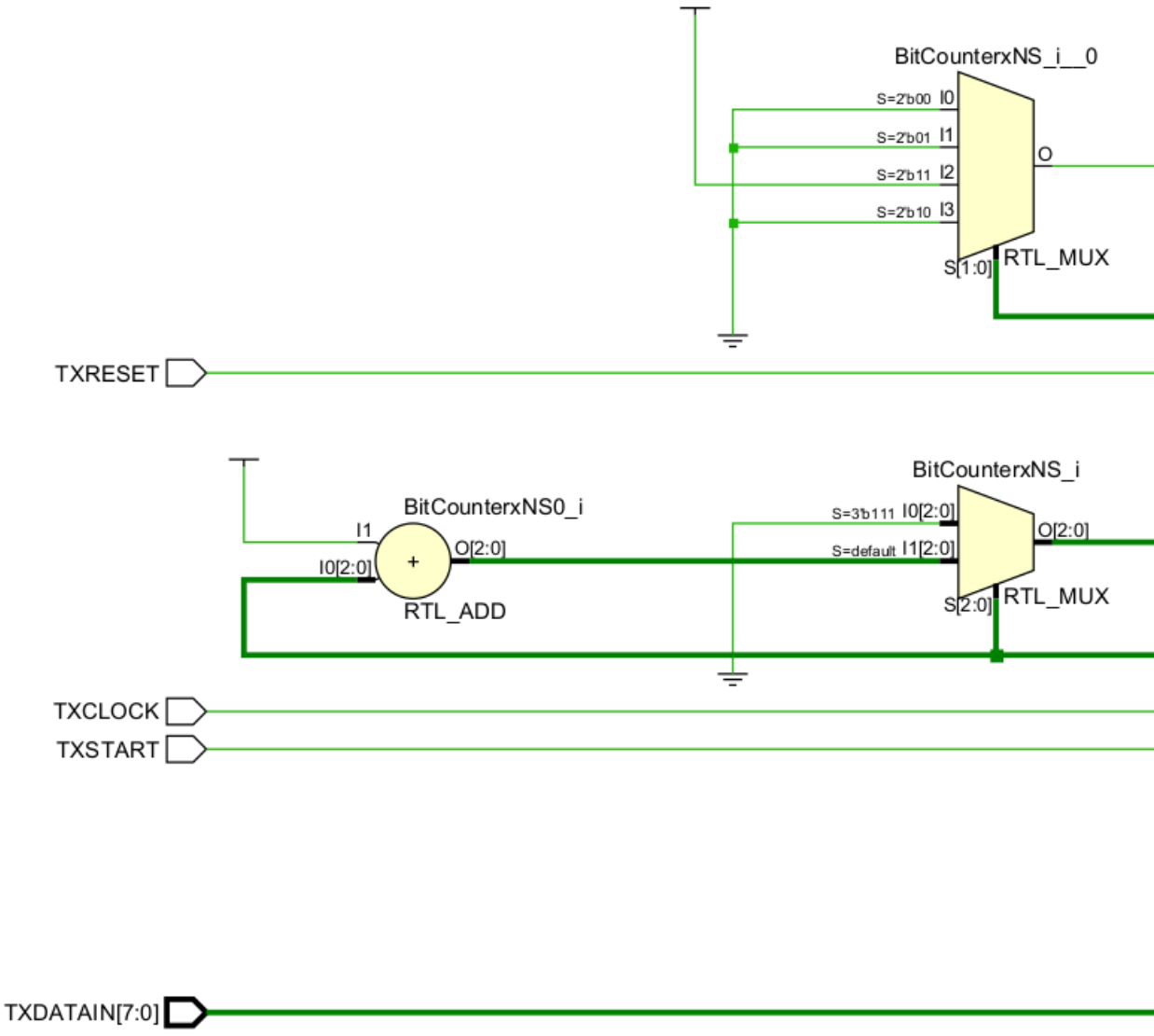
FSM in two shots. Ho usato il costrutto case così da avere anche in grafica i MUX fatti come si deve

```
fsm_process: process (fsmxPS, BitCounterxPS, DataRegisterxPS, TXDATAIN, TXSTART) beg
    fsmxNS <= fsmxPS;
    BitCounterxNS <= BitCounterxPS;
    DataRegisterxNS <= DataRegisterxPS;
            case fsmxPS is
        when idle =>
            TXDATAOUT <= '1';
            if TXSTART = '1' then
                fsmxNS <= start;
            end if;
                       when start =>
            TXDATAOUT <= '0';
            DataRegisterxNS <= TXDATAIN;
            fsmxNS <= transmitting;
                        when transmitting =>
            TXDATAOUT <= DataRegisterxPS( BitCounterxPS );
            if (BitCounterxPS = bit_num -1) then
                BitCounterxNS <= 0;
                fsmxNS <= stop;
            else
                BitCounterxNS <= BitcounterxPS +1;
            end if;
```

```
            when stop =>
                TXDATAOUT<= '1';
                fsmxNS <= idle;
            when others =>
                fsmxNS <= idle;
        end case;
    end process;
```

Risultato

Click to Zoom!


BitCounterxNS_i__0
S=2'b00 I0
S=2'b01 I1
S=2'b11 I2
S=2'b10 I3
O
S[1:0]
RTL_MUX
TXRESET
BitCounterxNS0_i
I1
I0[2:0]
+
O[2:0]
RTL_ADD
BitCounterxNS_i
S=3'b111 I0[2:0]
S=default I1[2:0]
O[2:0]
S[2:0]
RTL_MUX
TXCLOCK
TXSTART
TXDATAIN[7:0]

schema_nuovo.png

Non ci sono registri "strani" ma solo i 3 instanziati nel codice. L'uscita c'è un selettore che sceglie un solo bit all'interno del registro DataIn e questo selettore viene comandato dal bit counter, come è logico che sia! Quando boiler ha scritto il suo codice ha scelto di fare uno shift register, il dato in ingresso veniva registrato e poi shiftato via un colpo alla volta. Io ho scelto di tenere tutto il dato intatto e di leggere un bit diverso ogni volta. Non so se sia una scelta peggiore o al più uguale, ma mi piaceva fare un po di testa mia e sperimentare. Anche per questo è stato scritto il testbench e funziona a dovere. Nelle prossime puntate proverò un DDS e i suoi limiti e un ricevitore UART.

Davide