# Introduzione

Nello sviluppo di software per hardware simulato su Vivado capita, a volte, di voler utilizzare il modulo UART per effettuare comunicazioni, sia verso altri dispositivi piuttosto che all’utente. Nel caso di hardware simulato via software su Vivado, non ci risulta esista un sistema pratico per decodificare i segnali UART in uscita (TX) o entrata (RX). Per facilitare il debug della trasmissione di dati su UART, abbiamo quindi deciso di sviluppare un modulo che riesca a svolgere il processo di decodifica dei segnali RX e TX, in modo da rendere possibile l’interazione umana. Al fine di aumentare l’affidabilità della trasmissione su lunghe distanze, abbiamo, inoltre, sviluppato un modulo in grado di calcolare 4+1 bit aggiuntivi rispetto al semplice bit di parità, aprendo alla possibilità di effettuare oltre all’error detection anche l’error correction su errori a singolo bit e segnalazione di errore su errori a più bit. Il modulo UART\_0 è trasparente rispetto agli utenti finali della trasmissione, in modo da poter essere utilizzato senza necessariamente agire lato software per una corretta trasmissione di dati.

Anche se testato solo su Cortex M3 e AXI UART Lite, il modulo da noi sviluppato non dovrebbe, in linea teorica, presentare problemi di compatibilità con altre implementazioni, poiché indipendente dall’architettura ARM utilizzata.

# Analisi del sistema

Per comprendere meglio ciò che ci ha portati ad effettuare certe scelte progettuali, viene data una visione d’insieme del funzionamento della comunicazione seriale UART su dispositivi basati su ARM Cortex M3, AXI Interconnect ed AXI UART Lite.

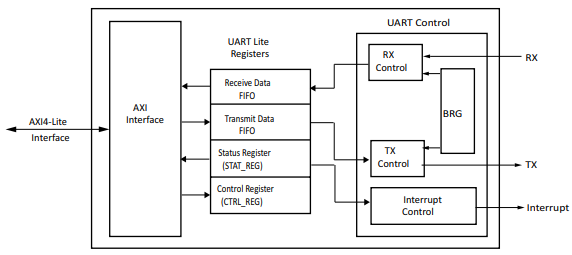


Figura 1: Schema a blocchi del modolo AXI UART Lite

Com’è possibile osservare dalla figura 1, il modulo consiste di 4 canali, 2 utilizzati per la comunicazione con il processore (etichettati “AXI4-Lite Interface” e “Interrupt”) e 2 per la comunicazione da e verso l’esterno (RX e TX). Nell’implementazione del dispositivo presa in esempio, i segnali RX e TX non sono direttamente esposti all’esterno del device, ma sono inviati al modulo DAP-Link con funzione di debugger e programmatore che si occupa di ritrasmetterli all’esterno. L’utilità di tale modulo ci risulta limitata in un ambiente simulato quale quello considerato. Il suo funzionamento non è stato quindi approfondito.

Il modulo può essere suddiviso in tre principali sotto-blocchi:

* AXI Interface: implementa le funzionalità della modalità slave del protocollo AXI4 Lite per permettere la comunicazione da e verso il processore di dati ed impostazioni.
* UART control: si occupa di generare e decodificare i segnali UART rispettivamente sulle linee TX ed RX. Il blocco BRG (baud rate generator) si occupa di generare il clock necessario a mantenere la velocità di trasmissione (baud rate) impostata. Il generatore di interrupt (interrupt control) genera due diversi segnali di interrupt (rising edge) per il processore: nel caso di nuovi dati presenti nella coda di ricezione (Receive data FIFO), utile nel caso si voglia evitare il polling della CPU per sondare la presenza di eventuali nuovi dati ricevuti dall’esterno (attraverso la linea RX) e nel caso la coda di trasmissione (Transmit data FIFO) non sia piena, per invitare la CPU ad inviare nuovi dati da trasmettere.
* UART Lite registers: sono i registri utili al funzionamento del modulo.

L’accesso ai registri è diretto e definito da offset di memoria rispetto all’indirizzo base del modulo.

I registri disponibili sono:

* RX FIFO: coda a 16 livelli contenente i dati ricevuti. Si tratta di un registro read only, eventuali richieste di scrittura vengono ignorate;
* TX FIFO: coda a 16 livelli contenente i dati da inviare attraverso il modulo UART. Il registro è write only. Nel caso in cui si tenti di scrivere mentre il registro è pieno, viene generato un errore sul bus AXI Lite;
* Control register (CTRL\_REG): è un registro write only attraverso il quale è possibile abilitare o disabilitare l’uscita interrupt del modulo e svuotare i registri RX FIFO e TX FIFO attraverso comandi di reset;
* Status register: (STAT\_REG): è un registro read only che contiene lo stato dei registri FIFO e flag riguardanti eventuali errori. In particolare, è presente un bit per identificare un errore nel bit di parità in pacchetti ricevuti, un bit per identificare frame malformati, un bit per segnalare dati ricevuti e scartati a causa della coda di ricezione piena e 4 bit per segnalare lo stato delle code di ricezione e trasmissione.

Nel modulo AXI\_Uart\_Lite, la velocità di trasmissione, la dimensione e le caratteristiche del frame sono stabilite alla generazione del modulo, con apposito tool per personalizzare l’IP. Non sono, quindi, modificabili attraverso software a momento dell’esecuzione. Per tale motivo, la nostra scelta implementativa si è focalizzata su un pacchetto a 8 bit senza bit di parità e velocità di trasmissione di 115200 baud/s. Resta possibile modificare e adattare la nostra soluzione per altre velocità e formati di frame con minime modifiche.

Ulteriori informazioni riguardo le modalità di indirizzamento, programmazione, consumi di risorse post sintesi ed esempi riguardanti questo modulo possono essere trovate all’indirizzo: <https://www.xilinx.com/support/documentation/ip_documentation/axi_uartlite/v2_0/pg142-axi-uartlite.pdf>

Con l’aiuto dello schema a blocchi dell’implementazione del core ARM M3, illustriamo le modalità con cui il processore può gestire le comunicazioni UART.

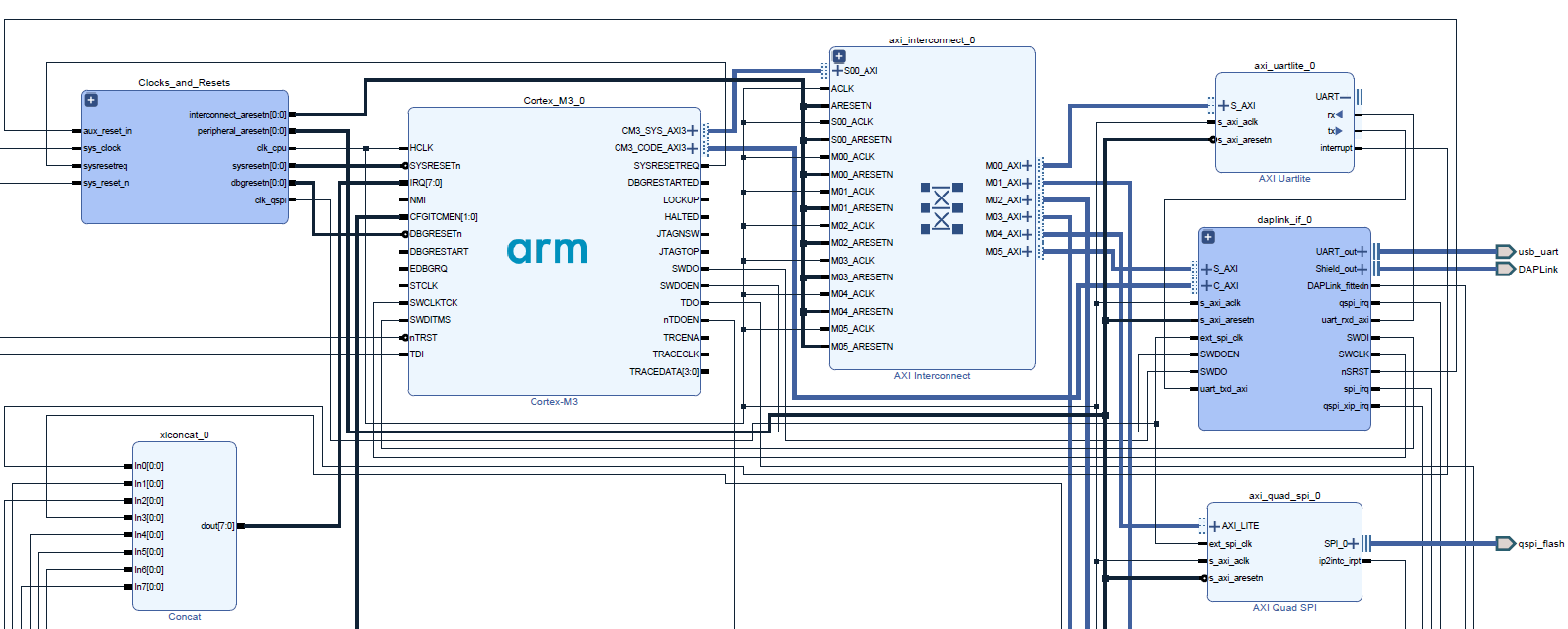


Figura 2: Estratto diagramma a blocchi dell'implementazione

Vi sono principalmente due modalità in cui il processore può leggere ed inviare dati al modulo UART: effettuando polling oppure sfruttando gli interrupt.

Nel caso in cui il processore voglia trasmettere un byte sulla porta UART TX, dovrà inviare una richiesta di scrittura all’indirizzo del registro TX FIFO, passando attraverso AXI Interconnect. Tale procedimento può, però, avvenire solamente a patto che la coda di trasmissione sia vuota. Il processore deve prima assicurarsi di questo controllando lo status register del modulo AXI UART Lite. Nella modalità in polling, tale controllo è effettuato a intervalli regolari, inviando richieste di lettura del registro STAT\_REG e controllando che il flag di coda piena non sia attivo. Nel caso si scegliesse, invece, di sfruttare gli interrupt, il modulo UART segnalerebbe un interrupt, alzando l’apposita linea; a tal punto, il processore può controllare il registro di stato del modulo per sincerarsi della condizione del buffer e, successivamente inviare il dato da scrivere, evitando numerose interrogazioni a vuoto dello status register.

Il processo di ricezione dei dati è analogo a quello di trasmissione: una volta arrivato un pacchetto sulla linea UART RX e dopo eventuali controlli sul bit di parità, il byte ricevuto viene inserito nella coda di ricezione RX FIFO e l’apposito bit segnalante dati validi ricevuti nello status register viene portato ad 1.

Successivamente, è possibile procedere in due modi: effettuando polling o sfruttando gli interrupt.

Nel primo caso, il processore dovrà controllare periodicamente lo status register per controllare la presenza di nuovi byte nella coda di ricezione che dovrà leggere con un’ulteriore richiesta. Nel secondo caso, il processore effettuerà il controllo solamente allo scattare di un interrupt.

È importante notare che, anche sfruttando gli interrupt, il processore dovrà sempre controllare lo status register, poiché l’interrupt è unico, sia che sia generato dal processo di trasmissione sia che lo sia da quello di ricezione, e non è quindi possibile distinguerli a prescindere.

# Analisi delle alternative

Conclusa questa premessa, vengono di seguito illustrate le possibili alternative a quella effettivamente realizzata.

Lo scopo primario di questa soluzione è quello di fornire un modo semplice per la persona di decodificare quanto inviato e ricevuto in una comunicazione UART.

Questa operazione dovrebbe essere idealmente eseguita senza modificare in modo significativo l’implementazione hardware già esistente, questo porta ad eliminare ogni soluzione che modifichi direttamente il modulo AXI\_UART\_Lite. L’intenzione, di fatti, è quella di fornire uno strumento di decodifica per un’implementazione hardware già esistente, in modo che anche il software compilato per essa non debba subire modifiche.

Una tale operazione di decodifica può essere eseguita da un modulo connesso principalmente in tre diversi punti: è possibile collegarsi al bus AXI tra processore e AXI Interconnect, al bus tra AXI inteconnect e il modulo AXI UART Lite oppure alle linee RX e TX.

Va notato, però, che utilizzare il bus tra processore ed AXI Interconnect come fonte di dati per il modulo porterebbe inevitabilmente quest’ultimo ad un’aggiunta complessità data dalla necessità di filtrare tutte le richieste non destinate al modulo UART. Inoltre, se non attentamente progettato, il modulo potrebbe causare un sovraccarico del bus a causa delle capacità parassite aggiunte, minando il corretto funzionamento dell’intero sistema.

Un'altra possibilità risiede nel collegare il modulo di decodifica tra AXI Interconnect ed il modulo AXI UART Lite. Anche questa opzione è subottimale: risolve in parte il problema del filtraggio delle richieste, ma non elimina il rischio di sovraccaricare la linea. Inoltre, un approccio basato su AXI renderebbe molto difficoltosa la realizzazione dello scopo secondario di questo progetto: una correzione trasparente degli errori sul singolo bit nel pacchetto sfruttando la codifica Hamming.

La creazione di un modulo esterno, che utilizzi solamente le linee TX ed RX porta numerosi vantaggi rispetto alle precedenti soluzioni: non è necessario effettuare alcun filtraggio dei dati letti, poiché sono esattamente i pacchetti che si intende decodificare; le linee UART sono più tolleranti rispetto a capacità parassite rispetto al bus AXI, inoltre, nel caso di una realizzazione fisica, sarebbe semplice aggiungere dei buffer per compensare tali carichi aggiuntivi. La realizzazione di un modulo totalmente indipendente dal sistema già presente permette di lasciare inalterati i segnali RX e TX, utilizzabili per altri scopi. Essendo un modulo esterno ed indipendente, inoltre, la portabilità verso altri dispositivi risulta notevolmente semplificata, necessitando solo dei due segnali RX e TX presenti in ogni comunicazione UART.

Di seguito è riassunto schematicamente uno scenario d’uso d’esempio:

TX

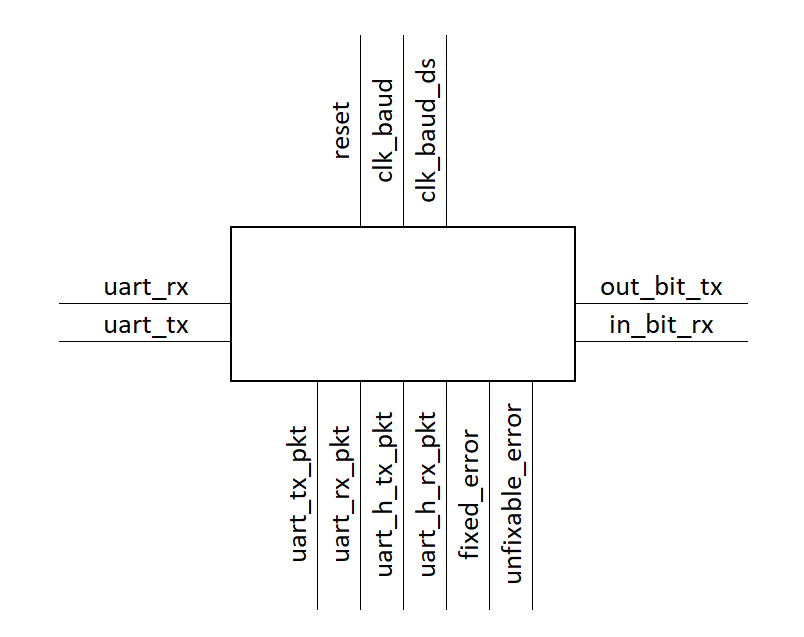
RX

RX

TX

A destra e a sinistra vi sono due microcontrollori (MC1, MC2), in mezzo una rappresentazione semplificata dei moduli UART\_0 sviluppati. La trasmissione parte da MC1, che raggiunge il modulo subito collegato. La comunicazione tra questo e il modulo UART\_0 all’altro estremo della comunicazione avviene aggiungendo bit di parità secondo lo schema Hamming. Il modulo di destinazione riceverà il pacchetto, correggerà (ove possibile) il pacchetto, o in alternativa segnalerà un problema del pacchetto all’esterno, causando un interrupt nel microcontrollore MC2. Configurando una precedenza sull’ordine degli interrupt, il segnale verrà intercettato prima che il pacchetto venga ricevuto, permettendo al ricevente di segnalare in tempo utile il re-invio del pacchetto errato al mittente MC1.

# Descrizione del modulo realizzato



Il modulo si attiva in uno dei seguenti casi:

* Ci sono bit in ingresso su uart\_rx a velocità clk\_baud
* Ci sono bit in ingresso su in\_bit\_rx a velocità clk\_baud\_ds

Nel primo caso, il pacchetto (per semplicità 8 bit, ma facilmente estendibile a parametro), riconosciuto dal bit iniziale a zero, viene inserito in un nuovo pacchetto, che contiene i bits secondo lo schema Hamming, di seguito riportato (con bit di parità finale su tutti i bit (compresi quelli di parità Hamming)):





Tale pacchetto viene riportato sul bus uart\_h\_tx\_pkt, mentre l’originale viene riportato sul bus uart\_rx\_pkt.

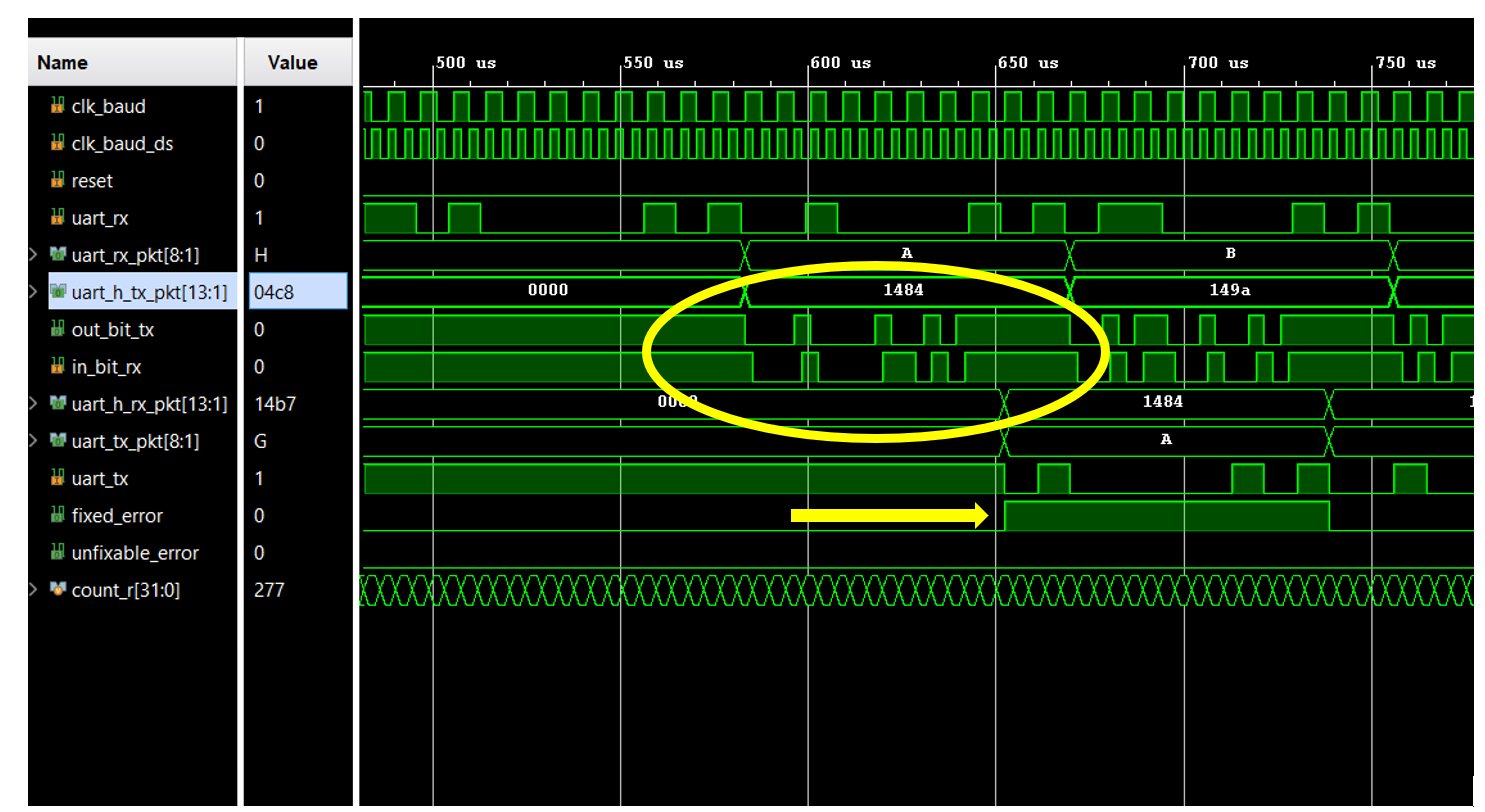
Nel secondo caso, il pacchetto (riportato anche sul bus uart\_h\_rx\_pkt), già nella forma con Hamming, viene in primo luogo decodificato, poi corretto (ove possibile) e infine segnalato il risultato. Chiamando P, lo XOR di tutti i bit del pacchetto ricevuto (13) e C, il conteggio dato dalle posizioni dei bit di parità che non corrispondono, si può utilizzare la seguente tabella per il procedimento di verifica di errori:

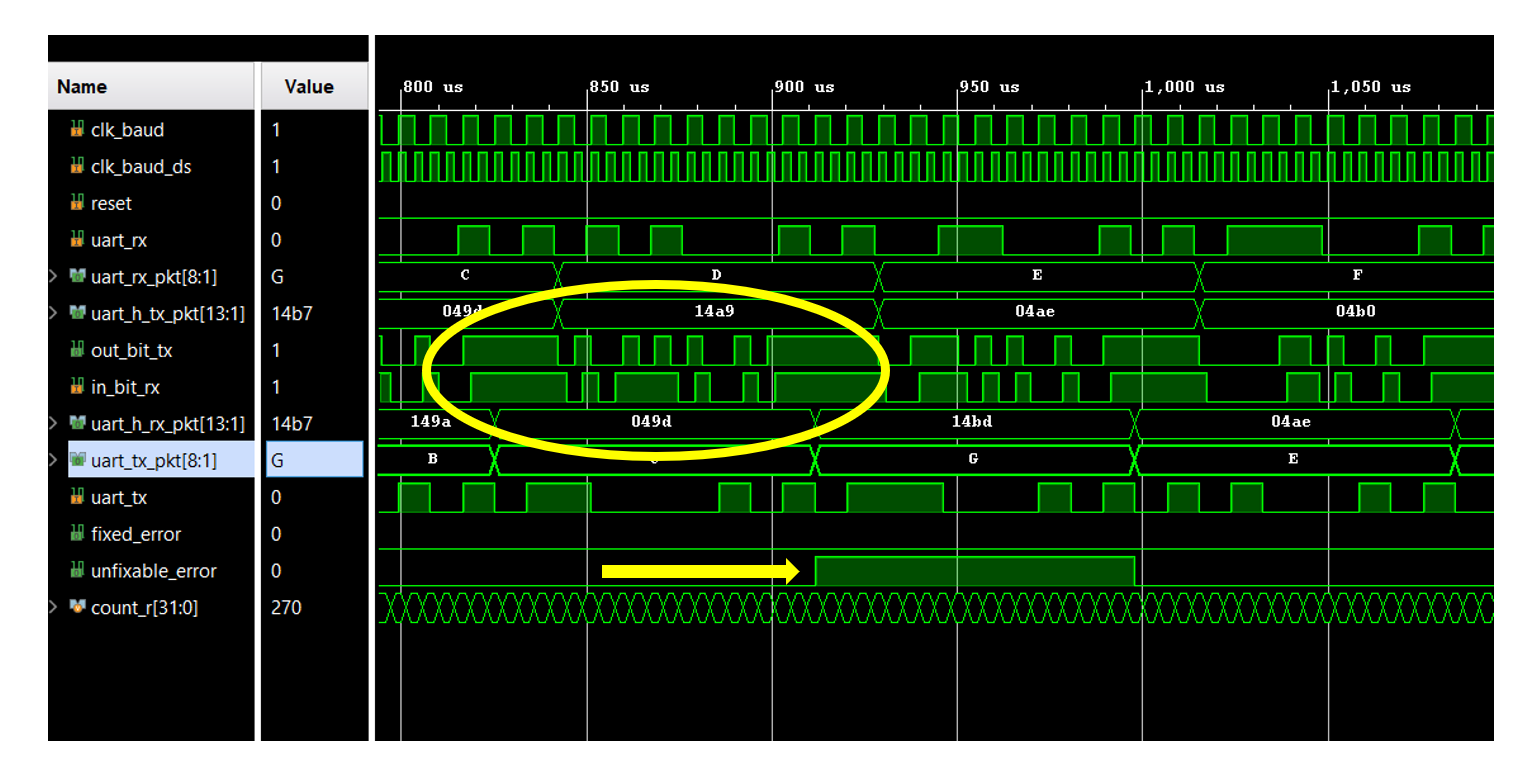
|  |  |  |
| --- | --- | --- |
| **C** | **P** | **esito** |
| = 0 | 0 | No errors |
| = 0 | 1 | Error on the bit 13. Can be fixed. |
| > 0 | 1 | Single bit error. Can be fixed |
| > 0 | 0 | Double error. Cannot be fixed. |

Come si può vedere dalle ultime due colonne della tabella, è necessario aggiungere il bit di parità finale per poter capire se è possibile correggere l’errore oppure no.

A seconda del tipo di errore, viene mandato un segnale all’esterno, per notificare l’utente e/o i dispositivi collegati dell’esito della procedura.

Il pacchetto viene infine trasmesso su uart\_tx e torna al mittente con lo stesso baud\_rate con cui invia i messaggi. I moduli in mezzo per evitare la formazione di code lavorano a doppio baud\_rate.





English Translation:

Introduction

During the development of softwares for simulated hardware on Vivado, it sometimes happens that the module UART is chosen to carry out communications to other devices rather than to the user. If we consider an hardware simulated by software on Vivado, no practical system to decode inbound (RX) or outbound (TX) UART signals is observed. To simplify the data transmission debug on UART, we have decided to develop a module which is able to run the decodification process of the RX and TX signals; in this way human interaction is possible. To increase the reliability level of the long-distance transmission, we have also developed a module which can evaluate 4+1 bit more than the simple parity bit; as a consequence, in addition to the error detection, we can also carry out the error correction on errors on a individual bit and signal an error over errors with more bit. The UART\_0 module is transparent with respect to the final users of the transmission, so that it can be used without necessarily acting on the software side for correct data transmission.

Even if only tested on Cortex M3 and AXI UART Lite, the module developed by us should not theoretically have compatibility problems with other implementations, as it is independent of the ARM architecture used.

System Analysis

To better understand what led us to make certain design choices, an overview of the operation of UART serial communication on devices based on ARM Cortex M3, AXI Interconnect and AXI UART Lite is given.

As Figure 1 shows, the module consists of 4 channels, 2 of which are used for communication with the processor (labeled "AXI4-Lite Interface" and "Interrupt") and the remaining 2 for communication from and to the outside (RX and TX). In the implementation of the device taken as an example, the RX and TX signals are not directly exposed to the outside of the device, but are sent to the DAP-Link module with the function of debugger and programmer that takes care of retransmitting them externally. The usefulness of this module is limited to us in a simulated environment such as the one considered. Therefore, its function was not investigated.

The module can be divided into three main sub-blocks:

* AXI Interface: implements the slave mode functions of the AXI4 Lite protocol to allow communication to and from the data and settings processor.
* UART control: takes care of generating and decoding the UART signals on the TX and RX lines, respectively. The BRG (baud rate generator) block is responsible for generating the necessary clock to maintain the set transmission speed (baud rate). The interrupt generator (interrupt control) generates two different interrupt signals (rising edge) for the processor: in the case of new data available in the reception queue (Receive data FIFO), they are particularly useful, if you want to avoid polling the CPU and to probe the presence of any new data received from the outside (through the RX line) and if the transmission queue (Transmit data FIFO) is not full, to invite the CPU to send new data to be transmitted.
* UART Lite registers: are the registers useful for the functioning of the module.

Access to the registers is direct and defined by memory offset with respect to the module base address.

The available registers are:

* RX FIFO: 16-level queue containing received data. This is a read only register, any write requests are ignored;
* TX FIFO: 16-level queue containing the data to be sent through the UART module. The register is write only. If an attempt is made to write while the register is full, an error is generated on the AXI Lite bus;
* Control register (CTRL\_REG): it is a write only register through which you can enable or disable the interrupt output of the module and empty the RX FIFO and TX FIFO registers through reset commands;
* Status register: (STAT\_REG): it is a read only register that contains the status of the FIFO registers and flags any errors. In particular, there is a bit to identify an error in the parity bit in received packets, a bit to identify malformed frames, a bit to report received and discarded data due to the full reception queue and 4 bits to report the status of the reception and transmission queues

In the AXI\_Uart\_Lite module, the transmission speed, size and characteristics of the frame are established with a special tool to customize the IP when the module is generated. Therefore, they cannot be modified through software at the time of execution. For this reason, our implementation choice has focused on an 8-bit packet without parity bit and a transmission speed of 115200 baud / s. It remains possible to modify and adapt our solution for other frame rates and formats with minimal changes.

Further information regarding the methods of addressing, programming, post synthesis resource consumption and examples regarding this module can be found at:

Link

Wi the block diagram of the ARM M3 core implementation, we illustrate how the processor can manage UART communications.

There are mainly two ways in which the processor can read and send data to the UART module: by polling or by exploiting interrupts.

If the processor wants to transmit a byte on the UART TX port, it must send a write request to the TX FIFO register address, passing through AXI Interconnect. However, this procedure can only take place provided that the transmission queue is empty. The processor must first make sure of this by checking the status register of the AXI UART Lite module. In polling mode, this check is performed at regular intervals, by sending requests to read the STAT\_REG register and by checking that the queue full flag is not active. On the other hand, if you choose to exploit the interrupts, the UART module would signal an interrupt, raising the appropriate line; at this point, the processor can check the status register of the module to verify the buffer condition and, subsequently, send the data to be written, avoiding numerous empty queries of the status register.

The data reception process is similar to the transmission one: once a packet has arrived on the UART RX line and after any checks on the parity bit, the received byte is inserted in the RX FIFO reception queue and the appropriate bit indicating valid data received in the status register is brought to 1.

Thereafter, you can proceed in two ways: by polling or by exploiting interrupts.

In the first case, the processor will have to periodically check the status register in order to detect the presence of new bytes in the reception queue which it will have to read with a further request. In the latter case, the processor will check only when an interrupt is triggered.

It is important to notice that, even by exploiting interrupts, the processor must always check the status register; whether it is generated by the transmission process or by the receiving one, the interrupt is unique, so it is not possible to distinguish them beforehand.

Alternative analysis

Given our premises, possible alternatives to the one actually implemented are illustrated below.

The primary purpose of this solution is to provide an easy way for the person to decode what is sent and received in a UART communication.

This operation should ideally be performed without significantly modifying the existing hardware implementation; this leads to the elimination of any solution that directly modifies the AXI\_UART\_Lite module. The intention, in fact, is to provide a decoding tool for an existing hardware implementation, so that even the software compiled for it does not have to undergo any changes.

Such a decoding operation can be performed by a module connected mainly by three different points: it is possible to connect to the AXI bus between the processor and AXI Interconnect, to the bus between AXI inteconnect and the AXI UART Lite module or to the RX and TX lines.

It should be noted, however, that using the bus between the processor and AXI Interconnect as a data source for the module would inevitably lead to the latter adding complexity due to the need to filter all requests not intended for the UART module. Furthermore, if not carefully designed, the module could cause a bus overload due to the added parasitic capabilities, undermining the correct functioning of the entire system.

Another possibility is to connect the decoding module between AXI Interconnect and the AXI UART Lite module. This option is also suboptimal: it partially solves the problem of request filtering, but does not eliminate the risk of overloading the line. Furthermore, an AXI-based approach would make it very difficult to realize the secondary purpose of this project: a transparent correction of single-bit errors in the packet using Hamming encoding.

The creation of an external module, which uses only the TX and RX lines, brings numerous advantages over the previous solutions: it is not necessary to perform any filtering of the data read, since they are exactly the packets to be decoded; UART lines are more tolerant of parasitic capacitance than the AXI bus, and, in the case of a physical implementation, it would be easy to add buffers to compensate for these additional loads. The realization of a module totally independent from the already present system allows to leave the RX and TX signals unchanged, which can be used for other purposes. Furthermore, being an external and independent module, portability to other devices is considerably simplified, requiring only the two RX and TX signals present in each UART communication.

An example usage scenario is graphically summarized below:

On the right and left there are two microcontrollers (MC1, MC2), in the middle a simplified representation of the developed UART\_0 modules. The transmission starts from MC1, which reaches the immediately connected module. The communication between this and the UART\_0 module at the other end of the communication takes place by adding parity bits according to the Hamming scheme. The target module will receive the packet, correct (where possible) the packet, or alternatively report a packet problem externally, causing an interrupt in the MC2 microcontroller. By configuring a precedence on the interrupt order, the signal will be intercepted before the packet is received, allowing the receiver to signal in good time the re-sending of the wrong packet to the sender MC1.

Description of the designed module

The module is activated in one of the following cases:

* There are bits in on uart\_rx at clk\_baud speed
* There are bits coming into in\_bit\_rx at clk\_baud\_ds speed

In the first case, the packet (8 bits for simplicity, but extendable toa certain parameter), recognized by the initial bit at zero, is inserted into a new packet, which contains the bits according to the Hamming scheme, shown here below (with parity bit final on all bits (including Hamming parity)):

This packet is reported on the uart\_h\_tx\_pkt bus, while the original is reported on the uart\_rx\_pkt bus.

In the second case, the packet (also reported on the uart\_h\_rx\_pkt bus), already in the form with Hamming, is first decoded, then corrected (where possible) and finally reports the result. By calling P, the XOR of all the bits of the received packet (13) and C, the count given by the positions of the parity bits that do not match, the following table can be used for the error checking procedure:

As can be seen from the last two columns of the table, it is necessary to add the final parity bit in order to understand whether it is possible to correct the error.

Depending on the type of error, a signal is sent outside to notify the user and / or connected devices of the outcome of the procedure.

The packet is finally transmitted on uart\_tx and returns to the sender with the same baud\_rate with which it sends messages. The modules in between to avoid the formation of queues work at double baud\_rate.