TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH

KHOA ĐIỆN ĐIỆN TỬ

ĐỒ ÁN 1

NGÀNH CÔNG NGHỆ KỸ THUẬT MÁY TÍNH

THIẾT KẾ MODULE UART TRÊN FPGA

SVTH: **NGUYỄN THỊ NGỌC HÂN**

MSSV: 22119067

GVHD: **ThS. HUỲNH HOÀNG HÀ**

TP. HỒ CHÍ MINH – 05/2025

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM

Độc lập – Tự do – Hạnh phúc

----***----

TP. Hồ Chí Minh, ngày 25 tháng 05 năm 2025

NHIỆM VỤ ĐỒ ÁN 1

Họ và tên sinh viên: Nguyễn Thị Ngọc Hân **MSSV**: 22119067

Ngành: Công nghệ kỹ thuật máy tính **Lớp:** 22119CL2B

Giảng viên hướng dẫn: ThS. HUỲNH HOÀNG HÀ

Ngày nhận đề tài: **Ngày nộp đề tài:**

1. Tên đề tài: Thiết kế module UART trên FPGA

2. Các số liệu, tài liệu ban đầu:

3. Sản phẩm:

Mô phỏng module truyền và nhận UART trên FPGA (Xilinx)

TRƯỞNG NGÀNH **GIẢNG VIÊN HƯỚNG DẪN**

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM

Độc lập – Tự do – Hạnh phúc

----***----

TP. Hồ Chí Minh, ngày 25 tháng 05 năm 2025

PHIẾU NHẬN XÉT CỦA GIÁO VIÊN PHẢN BIỆN

NHIỆM VỤ ĐỒ ÁN 1

Họ và tên sinh viên: Nguyễn Thị Ngọc Hân **MSSV**: 22119067

Ngành: Công nghệ kỹ thuật máy tính **Lớp:** 22119CL2B

Giảng viên hướng dẫn: ThS. HUỲNH HOÀNG HÀ

Ngày nhận đề tài: **Ngày nộp đề tài:**

NHẬN XÉT

1. Về nội dung đề tài & khối lượng thực hiện:

.....

2. Ưu điểm:

.....

3. Khuyết điểm:

.....

4. Đánh giá loại:

.....

5. Điểm: ..... (Bằng chữ:)

.....

GIẢNG VIÊN PHẢN BIỆN

LỜI CẢM ƠN

Lời đầu tiên, em xin gửi lời cảm ơn chân thành đến thầy Huỳnh Hoàng Hà, người đã hướng dẫn và hỗ trợ em trong suốt quá trình thực hiện Đồ án 1 này. Thầy đã chia sẻ với em những kiến thức, hiểu biết và kinh nghiệm quý báu, giúp em có cái nhìn sâu sắc hơn về lĩnh vực nghiên cứu mà em đang thực hiện. Sự tận tâm và nhiệt huyết của thầy là nguồn động lực lớn lao, giúp em vượt qua những thách thức khó khăn trong quá trình làm Đồ án. Những ý kiến đóng góp và sự chỉ bảo của thầy đã giúp em hoàn thiện nội dung cũng như phương pháp nghiên cứu, từ đó nâng cao chất lượng đề tài.

Trong suốt quá trình thực hiện, em cũng không tránh khỏi những sai sót và thiếu sót. Em rất mong nhận được sự đóng góp của thầy để có thể hoàn thiện đề tài tốt hơn.

LỜI CAM ĐOAN

Đồ án 1 mang tên " Thiết kế module UART trên FPGA" là quá trình nghiên cứu của em thực hiện đề tài dưới sự chỉ dẫn của thầy Huỳnh Hoàng Hà. Mọi thông tin, dữ liệu và tài liệu mà em thực hiện đề tài tham khảo được trích dẫn trong đồ án này đều được ghi rõ nguồn gốc và tác giả theo quy tắc của trường. Em hiểu rằng bất kỳ hành vi gian lận nào trong quá trình thực hiện đồ án sẽ bị xử lý theo quy định của trường.

Người thực hiện Đồ án 1

(Ký và ghi rõ họ tên)

MỤC LỤC

DANH MỤC HÌNH

CHƯƠNG 1: GIỚI THIỆU

1.1 GIỚI THIỆU

Trong lĩnh vực điện tử và truyền thông, giao tiếp nối tiếp đóng vai trò quan trọng trong việc trao đổi dữ liệu giữa các thiết bị. UART (Universal Asynchronous Receiver/Transmitter) là một giao thức truyền thông nối tiếp không đồng bộ phổ biến, được sử dụng rộng rãi nhờ đơn giản, dễ triển khai và không yêu cầu tín hiệu đồng hồ chung giữa các thiết bị.

Trong môi trường công nghiệp với nhiều nhiễu điện từ, việc thiết kế module UART đáng tin cậy trở thành thách thức lớn. Đề tài này tập trung vào việc thiết kế và hiện thực module UART trên FPGA với khả năng chống nhiễu cao, sử dụng kỹ thuật lấy mẫu đa điểm và xác định giá trị bit dựa trên phương pháp bỏ phiếu (voting). Thiết kế này không chỉ đảm bảo truyền dữ liệu ổn định trong môi trường nhiễu mà còn tối ưu hóa tài nguyên FPGA.

1.2 MỤC TIÊU ĐỀ TÀI

Thiết kế module UART hoàn chỉnh bao gồm cả phần truyền (TX) và nhận (RX) dữ liệu trên FPGA.
Hiện thực cơ chế chống nhiễu cho module nhận thông qua phương pháp lấy mẫu đa điểm và xác định giá trị bit dựa trên xác suất.
Xây dựng hệ thống kiểm tra và đánh giá hiệu năng của module trong cả môi trường lý tưởng và có nhiễu.
Mô phỏng module trên Xilinx

1.3 GIỚI HẠN ĐỀ TÀI

Đề tài tập trung vào:

Thiết kế UART hoạt động ở các tốc độ baud phổ biến: 9600, 19200, 38400, 57600 và 115200 bps.

Mô phỏng trên Xilinx IDE

Giải pháp chống nhiễu tập trung vào phần nhận dữ liệu (RX).

1.4 PHƯƠNG PHÁP NGHIÊN CỨU

Phương pháp phân tích hệ thống: Phân tích yêu cầu và đặc tả kỹ thuật của giao thức UART.
Phương pháp mô hình hóa: Thiết kế module bằng ngôn ngữ Verilog HDL với kiến trúc phân tầng rõ ràng.
Phương pháp mô phỏng: Sử dụng Xilinx IDE để kiểm tra chức năng và thời gian của thiết kế.
Phương pháp đánh giá: So sánh hiệu năng giữa phương pháp lấy mẫu truyền thống và phương pháp cải tiến trong môi trường có nhiễu.

1.5 ĐỐI TƯỢNG VÀ PHẠM VI NGHIÊN CỨU

Đối tượng nghiên cứu:

Giao thức truyền thông UART không đồng bộ.

Kỹ thuật chống nhiễu trong truyền thông nối tiếp.

Ngôn ngữ mô tả phần cứng Verilog HDL.

Phạm vi nghiên cứu:

Thiết kế phần cứng số cho module UART TX/RX.

Cơ chế đồng bộ hóa tín hiệu không đồng bộ.

Giải pháp lấy mẫu đa điểm và xử lý nhiễu.

1.6 BỐ CỤC QUYỂN BÁO CÁO

Báo cáo được tổ chức thành các chương chính sau:

1. **Giới thiệu**: Tổng quan về đề tài, mục tiêu và phạm vi nghiên cứu.
2. **Cơ sở lý thuyết**: Các khái niệm về UART, giao tiếp không đồng bộ và kỹ thuật chống nhiễu.
3. **Thiết kế hệ thống**: Kiến trúc tổng thể và thiết kế chi tiết các module.
4. **Mô phỏng**: Kết quả mô phỏng, thực nghiệm.

CHƯƠNG 2: CƠ SỞ LÝ THUYẾT

2.1 GIỚI THIỆU VỀ UART

2.1.1 Khái niệm về UART

UART (Universal Asynchronous Receiver/Transmitter) là một giao thức truyền thông nối tiếp không đồng bộ, được triển khai dưới dạng khối chức năng phần cứng trong vi điều khiển, FPGA, hoặc các mạch tích hợp độc lập. Nó cho phép cấu hình định dạng dữ liệu và tốc độ truyền (baud rate) để thực hiện giao tiếp giữa các thiết bị

Truyền nhận không đồng bộ đa năng (UART) hoặc giao tiếp nối tiếp là một trong những giao thức giao tiếp đơn giản nhất giữa hai thiết bị. Nó truyền dữ liệu giữa các thiết bị bằng cách kết nối hai dây giữa các thiết bị, một dây là đường truyền trong khi dây kia là đường nhận. Dữ liệu truyền từng chút một kỹ thuật số dưới dạng bit từ thiết bị này sang thiết bị khác. Ưu điểm chính của giao thức truyền thông này là cả hai thiết bị không nhất thiết phải có cùng tần số hoạt động. Ví dụ, hai bộ vi điều khiển hoạt động ở các tần số xung nhịp khác nhau có thể giao tiếp với nhau một cách dễ dàng thông qua giao tiếp nối tiếp. Tuy nhiên, tốc độ bit được xác định trước được gọi là tốc độ truyền thường được đặt trong bộ nhớ flash của cả hai bộ vi điều khiển để cả hai thiết bị hiểu lệnh.

2.1.2 Truyền và nhận dữ liệu nối tiếp

Bộ truyền UART sẽ lấy từng byte dữ liệu và truyền các bit theo thứ tự tuần tự. Bộ thu ở phía bên kia (cũng là một UART) sẽ ghép lại các bit đó thành một byte hoàn chỉnh.

Việc truyền dữ liệu tuần tự qua một dây dẫn đơn thực tế tiết kiệm chi phí hơn so với truyền song song qua nhiều dây.

Giao tiếp giữa hai thiết bị UART có thể ở các chế độ: simplex, song công hoàn toàn (full-duplex) hoặc bán song công (half-duplex).

Simplex là kiểu giao tiếp một chiều, trong đó tín hiệu chỉ đi từ một UART đến UART còn lại, và không có cơ chế để UART nhận phản hồi lại tín hiệu.

Full-duplex là khi cả hai thiết bị có thể truyền và nhận dữ liệu đồng thời.

Half-duplex là khi các thiết bị luân phiên nhau truyền và nhận dữ liệu.

2.1.3 Cấu trúc của giao thức UART

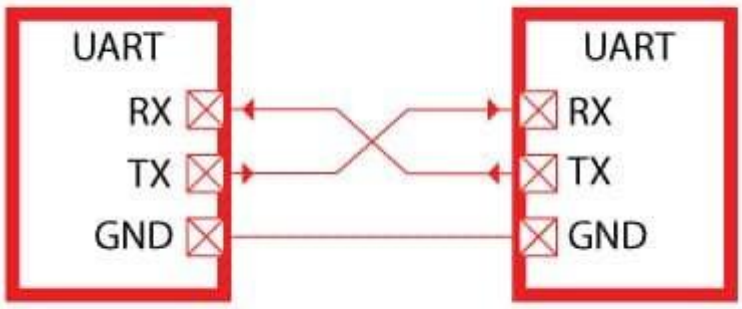
Một UART chứa một bộ tạo xung nhịp (clock generator). Điều này cho phép việc lấy mẫu (sampling) trong suốt một chu kỳ bit.
Nó cũng bao gồm các thanh ghi dịch đầu vào và đầu ra (input and output shift registers). Ngoài ra còn có khối điều khiển truyền và nhận dữ liệu (transmitting and receiving control), cùng với logic điều khiển đọc/ghi (read/write control logic).

Các thành phần tùy chọn khác của UART có thể bao gồm:

- Bộ đệm truyền hoặc nhận (transmit/receive buffers)
- Bộ nhớ đệm FIFO (FIFO buffer memory)

- Bộ điều khiển DMA (DMA controller)

2.1.4 Cách giao tiếp thông qua UART



Hình 1: Kết nối vật lý của chuẩn UART cơ bản

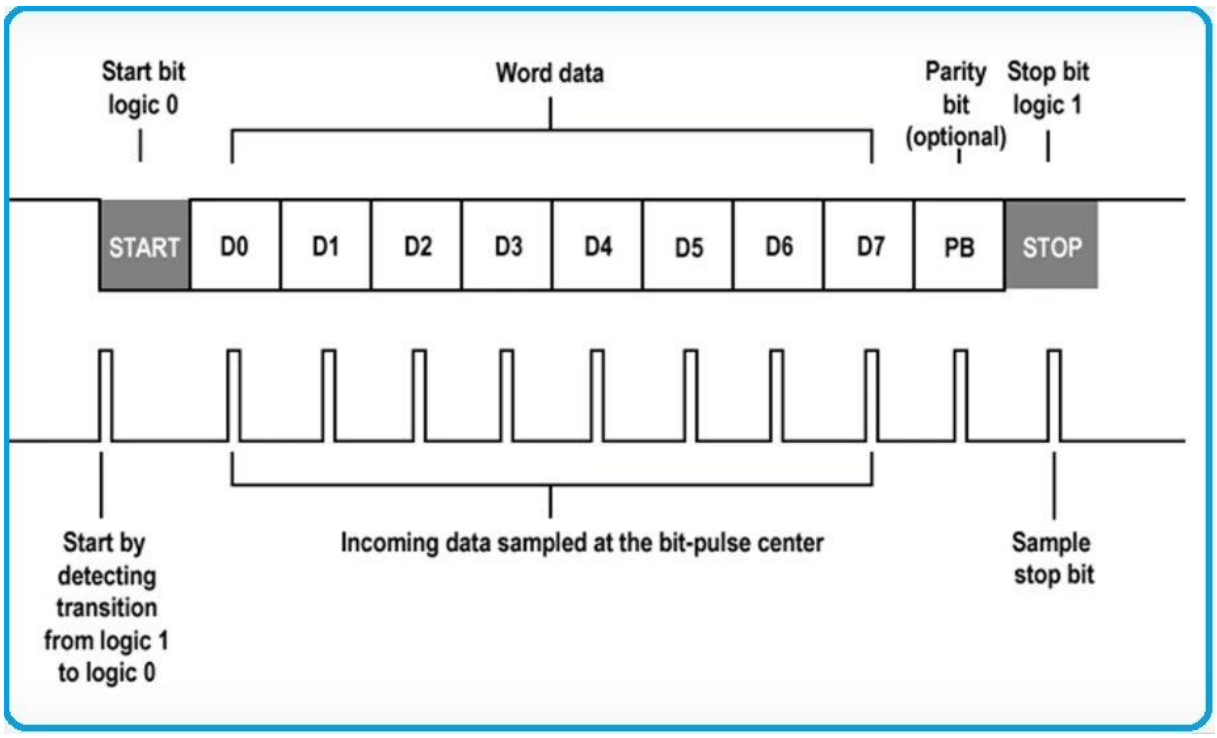
Cần hai module UART để giao tiếp trực tiếp với nhau. Ở một đầu, UART truyền chuyển đổi dữ liệu song song từ CPU thành dạng nối tiếp, sau đó truyền dữ liệu ở dạng nối tiếp đến UART thứ hai sẽ nhận dữ liệu nối tiếp và chuyển đổi trở lại thành dữ liệu song song. Dữ liệu này sau đó có thể được truy cập từ thiết bị nhận.

Thay vì tín hiệu không thể hiện rõ, bit truyền và nhận sử dụng tín hiệu start bit và stop bit cho các gói dữ liệu. Các start bit và stop bit này xác định phần đầu và phần cuối của các gói dữ liệu. Do đó, UART nhận biết khi nào nên bắt đầu và dừng đọc các bit.

UART nhận sẽ phát hiện start bit sau đó bắt đầu đọc các bit. Tần số cụ thể được sử dụng để đọc các bit đến được gọi là tốc độ truyền. Tốc độ truyền là một thước đo được sử dụng cho tốc độ truyền dữ liệu. Đơn vị được sử dụng cho tốc độ truyền là bit trên giây (bps). Để quá trình truyền dữ liệu thành công, cả UART truyền và nhận phải hoạt động ở tốc độ truyền gần như giống nhau. Tuy nhiên, nếu tốc độ truyền khác nhau giữa cả hai UART, chúng chỉ được chênh lệch 10%. UART nhận và truyền phải được định cấu hình để nhận các gói dữ liệu giống nhau.

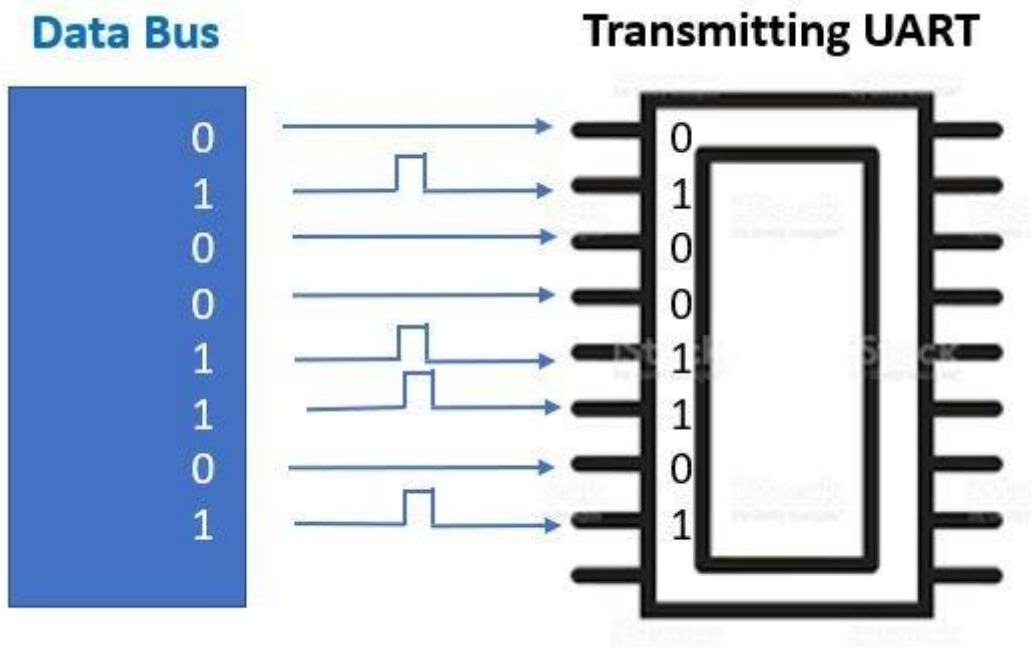
2.2 THÔNG SỐ CƠ BẢN VÀ KHUNG TRUYỀN DỮ LIỆU

2.2.1 Cách hoạt động của giao tiếp UART



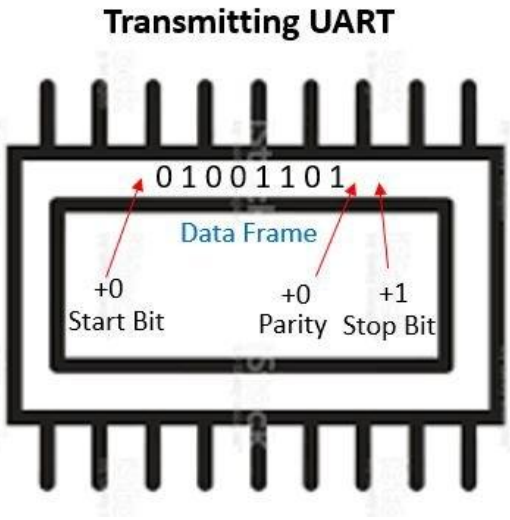
Hình 2: Cấu trúc khung dữ liệu UART

Đầu tiên, bus dữ liệu gửi dữ liệu đến UART truyền. UART truyền nhận dữ liệu từ bus dữ liệu. Bus dữ liệu được sử dụng để gửi dữ liệu từ một thiết bị khác như vi điều khiển, bộ nhớ hoặc CPU.



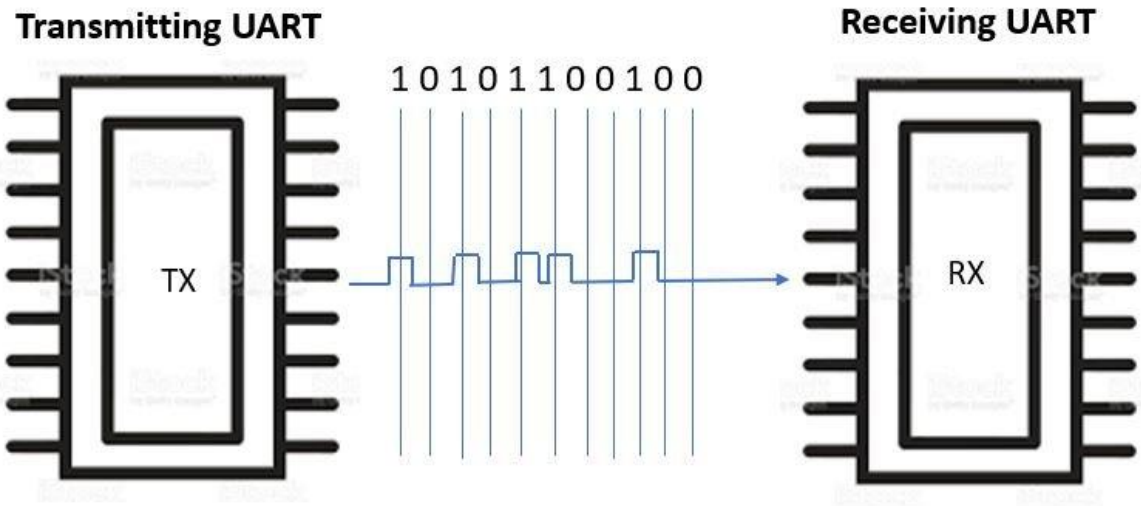
Hình 3: Truyền dữ liệu song song sang nối tiếp

Khi UART truyền nhận dữ liệu, nó xử lý dữ liệu bằng cách thêm bit bắt đầu và bit dừng. Điều này lần lượt tạo ra một gói dữ liệu.



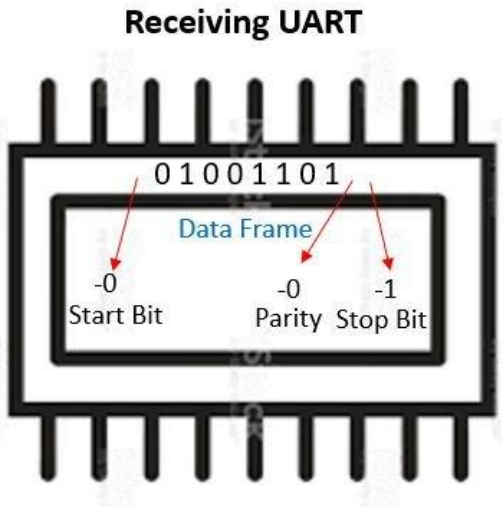
Hình 4: Data Frame trong gói truyền UART

Bước tiếp theo là truyền dữ liệu từ UART truyền đến UART nhận. Toàn bộ gói dữ liệu bao gồm gói dữ liệu, bit bắt đầu và dừng được gửi đến UART nhận một cách nối tiếp. Dòng dữ liệu được lấy mẫu ở tốc độ truyền được cấu hình sẵn bởi UART nhận**. Gói dữ liệu được xuất ra nối tiếp ở chân Tx. Sau đó, UART sẽ đọc gói dữ liệu từng chút một thông qua chân Rx của nó**



Hình 5: Quá trình truyền - nhận UART

Ở phía nhận, UART nhận loại bỏ start bit, parity bit và stop bit khỏi khung dữ liệu. Điều này là để chuyển đổi dữ liệu trở lại dạng ban đầu.



Hình 6: Data Frame trong gói nhận UART

2.2.2 Timing Diagram

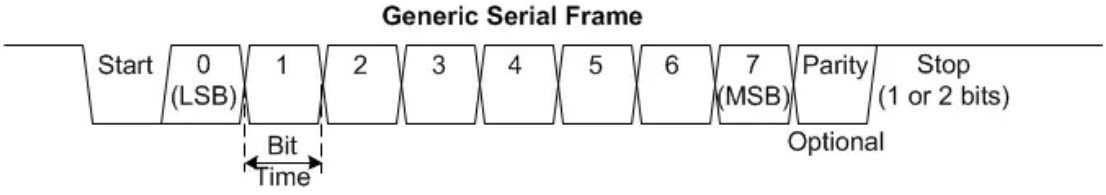
Start bit: Khi không truyền dữ liệu, đường truyền dữ liệu UART thường là điện áp cao. Để bắt đầu quá trình truyền, UART truyền chuyển từ điện áp cao sang điện áp thấp trong một chu kỳ đồng hồ. UART nhận sẽ phát hiện áp cao đến áp thấp chuyển đổi và bắt đầu đọc các bit ở tốc độ truyền chính xác.

Parity bit: Chẵn lẻ là độ lẻ hoặc đồng đều của một số. Bit chẵn lẻ có chức năng cho UART nhận biết nếu dữ liệu có thay đổi trong quá trình truyền. Các bit có thể thay đổi do điện từ, tốc độ truyền khác nhau hoặc truyền dữ liệu khoảng cách xa. UART đọc khung dữ liệu sau khi nhận được dữ liệu. Sau đó, nó đếm số bit và kiểm tra xem chúng là chẵn hay lẻ. Nếu bit parity là 0 thì nó là parity chẵn. Nếu bit là 1 thì đó là parity lẻ. Để UART biết rằng quá trình truyền không có lỗi, bit chẵn lẻ phải khớp với dữ liệu.

Stop bit: Trong thời lượng ít nhất hai bit, UART truyền dẫn đường truyền từ điện áp thấp sang điện áp cao.

Tóm lại, nguyên lí hoạt động của UART là:

- Dữ liệu trong giao tiếp nối tiếp được gửi bằng cách truyền các gói dữ liệu. Mỗi gói dữ liệu bao gồm start bit, byte dữ liệu (8 bit), parity bit (optional) và stop bit.
- Start bit được gửi trước khi gửi byte dữ liệu để thông báo cho thiết bị nhận bắt đầu ghi dữ liệu. Bit này hoạt động ở mức thấp, có nghĩa là bit chuyển sang mức thấp khi nó phải thông báo cho người nhận.
- Sau start bit, byte dữ liệu được gửi. 8 bit liên tiếp được truyền và thời gian của chúng phụ thuộc vào tốc độ truyền (baud rate).



Hình 7: Timing Diagram

- Sau khi byte dữ liệu được truyền đi, một bit chẵn lẻ (parity bit) sẽ được gửi. Bit này là tùy chọn và được sử dụng để kiểm tra lỗi dữ liệu trong quá trình truyền. Khi byte dữ liệu đang được truyền, số bit có mức cao (1) trong tổng số 8 bit của byte dữ liệu sẽ được đếm. Có hai loại kiểm tra chẵn lẻ có thể được áp dụng: chẵn (Even) hoặc lẻ (Odd).
- Ví dụ, nếu số lượng bit có mức cao là 3 (số lẻ), thì bit chẵn lẻ sẽ được đặt là 1. Trong kiểm tra chẵn (even parity), bit parity được đặt là 1 để tổng số bit "1" là 4 (chẵn). Trong kiểm tra lẻ (odd parity), bit parity được đặt là 0 để tổng số bit "1" là 3 (lẻ).
- Khi thiết bị nhận nhận được gói dữ liệu, nó cũng sẽ đếm số bit mức cao trong byte dữ liệu và so sánh với bit chẵn lẻ. Nếu bit chẵn lẻ xác nhận tính đúng đắn của dữ liệu, dữ liệu sẽ được xử lý tiếp. Nếu bit chẵn lẻ và số bit mức cao không khớp nhau, dữ liệu sẽ bị loại bỏ do bị lỗi và bộ đếm lỗi sẽ được tăng lên. Nếu bộ đếm lỗi vượt quá một ngưỡng nhất định, nên giảm tốc độ baud để tăng chất lượng dữ liệu, dù phải đánh đổi bằng việc giảm tốc độ truyền.
- Sau bit chẵn lẻ, một bit dừng (stop bit) sẽ được gửi để thông báo cho thiết bị nhận biết rằng gói dữ liệu đã kết thúc. Stop bit thường có độ dài tối thiểu 1 bit, nhưng trong một số cấu hình, có thể sử dụng 1.5 hoặc 2 bit để tăng thời gian nghỉ giữa các gói dữ liệu, giúp giảm tỷ lệ lỗi trong môi trường nhiễu hoặc khi tốc độ baud cao

2.2.3 Baud Rate

Giao tiếp giữa hai thiết bị thông qua giao thức UART diễn ra bằng cách truyền các bit. Tổng cộng có 8 bit được gửi liên tiếp nhau để truyền một byte. Mỗi bit có thể là mức logic thấp hoặc cao. Khoảng thời gian giữa hai bit được gọi là tốc độ baud (baud rate) hoặc tốc độ bit (bit rate). Tốc độ này phải được xác định trước ở cả hai thiết bị để thiết bị gửi có thể mã hóa dữ liệu thành các bit với khoảng thời gian cụ thể này, và thiết bị nhận có thể nhận các bit kế tiếp đúng thời điểm. Tốc độ baud được sử dụng phổ biến nhất là 9600 bit mỗi giây.

Tốc độ baud cao hơn (như 115200 bps) có thể làm tăng khả năng lỗi dữ liệu, đặc biệt trong môi trường nhiễu hoặc khi truyền qua khoảng cách xa do suy giảm tín hiệu và khó khăn trong việc đồng bộ. Để giảm lỗi, có thể sử dụng các kỹ thuật như lấy mẫu đa điểm hoặc giảm tốc độ baud

2.3 KỸ THUẬT CHỐNG NHIỄU TRONG HỆ THỐNG UART

2.3.1 Giới Thiệu

Giao thức UART (Universal Asynchronous Receiver-Transmitter) được sử dụng rộng rãi trong các hệ thống nhúng để truyền dữ liệu nối tiếp. Tuy nhiên, trong môi trường thực tế, nhiễu điện từ, dao động tín hiệu, hoặc bất đồng bộ đồng hồ có thể gây ra lỗi trong quá trình truyền và nhận dữ liệu. Để đảm bảo độ tin cậy, các kỹ thuật chống nhiễu là yếu tố quan trọng trong thiết kế UART. Phần này trình bày các phương pháp

chống nhiễu phổ biến và liên hệ với các kỹ thuật được áp dụng trong thiết kế UART hiện tại.

2.3.2 Các kỹ thuật chống nhiễu phổ biến trong UART

Đồng bộ hóa tín hiệu

Tín hiệu UART bất đồng bộ thường không đồng pha với đồng hồ hệ thống, dẫn đến nguy cơ metastability. Để giải quyết, tín hiệu đầu vào được đồng bộ hóa bằng cách sử dụng các thanh ghi nối tiếp (flip-flop) để căn chỉnh với miền đồng hồ. Kỹ thuật này giúp giảm thiểu lỗi do nhiễu bất đồng bộ và đảm bảo tín hiệu ổn định trước khi xử lý.

Lấy mẫu đa điểm

Nhiễu ngắn hạn (spike noise) có thể làm sai lệch giá trị của một bit dữ liệu. Để khắc phục, kỹ thuật lấy mẫu đa điểm (oversampling) được sử dụng, trong đó mỗi bit được lấy mẫu nhiều lần (thường từ 3 đến 16 lần) trong một chu kỳ bit. Giá trị bit cuối cùng được xác định dựa trên đa số mẫu hoặc tổng giá trị mẫu, giúp loại bỏ ảnh hưởng của nhiễu tạm thời.

Phát hiện và kiểm tra khung dữ liệu

Một khung UART bao gồm start bit, data bits, và stop bit. Nhiễu có thể làm sai lệch các bit này, dẫn đến lỗi khung. Kỹ thuật kiểm tra tính hợp lệ của start bit và stop bit được sử dụng để phát hiện khung dữ liệu bị lỗi. Nếu các bit này không thỏa mãn điều kiện (ví dụ: start bit không ở mức thấp hoặc stop bit không ở mức cao), khung dữ liệu sẽ bị từ chối, giảm thiểu lỗi do nhiễu.

Sử dụng bit kiểm tra lỗi

Để tăng cường phát hiện lỗi, một số hệ thống UART sử dụng bit chẵn lẻ (parity bit) hoặc mã kiểm tra (CRC) trong khung dữ liệu. Các bit này cho phép module nhận kiểm tra tính toàn vẹn của dữ liệu, phát hiện và loại bỏ các khung bị nhiễu.

Bộ lọc số

Nhiễu dạng xung ngắn có thể được loại bỏ bằng cách áp dụng bộ lọc số, chẳng hạn như bộ lọc bỏ phiếu đa số (majority voting). Bộ lọc này chỉ chấp nhận giá trị tín hiệu nếu nó ổn định qua nhiều chu kỳ đồng hồ, giúp tăng độ bền của hệ thống trong môi trường nhiễu cao.

2.3.3 Liên hệ với thiết kế UART hiện tại

Trong thiết kế UART được phát triển, một số kỹ thuật chống nhiễu đã được tích hợp để đảm bảo độ tin cậy của quá trình nhận dữ liệu. Cụ thể:

Đồng bộ hóa tín hiệu: Tín hiệu UART đầu vào được xử lý qua các giai đoạn đồng bộ hóa để căn chỉnh với đồng hồ hệ thống, giảm thiểu nguy cơ metastability.

Lấy mẫu đa điểm: Mỗi bit dữ liệu được lấy mẫu nhiều lần trong một chu kỳ bit, với giá trị bit được xác định dựa trên tổng các mẫu, giúp giảm lỗi do nhiễu ngắn hạn.

Kiểm tra khung dữ liệu: Hệ thống kiểm tra tính hợp lệ của start bit và stop bit, từ chối các khung dữ liệu không đạt yêu cầu, đảm bảo chỉ các dữ liệu đúng định dạng được xử lý.

Các kỹ thuật này phù hợp với môi trường có mức nhiễu trung bình, chẳng hạn như trong các ứng dụng nhúng hoặc giao tiếp với vi điều khiển.

CHƯƠNG 3: THIẾT KẾ GIAO THỨC TRUYỀN NHẬN DỮ LIỆU UART

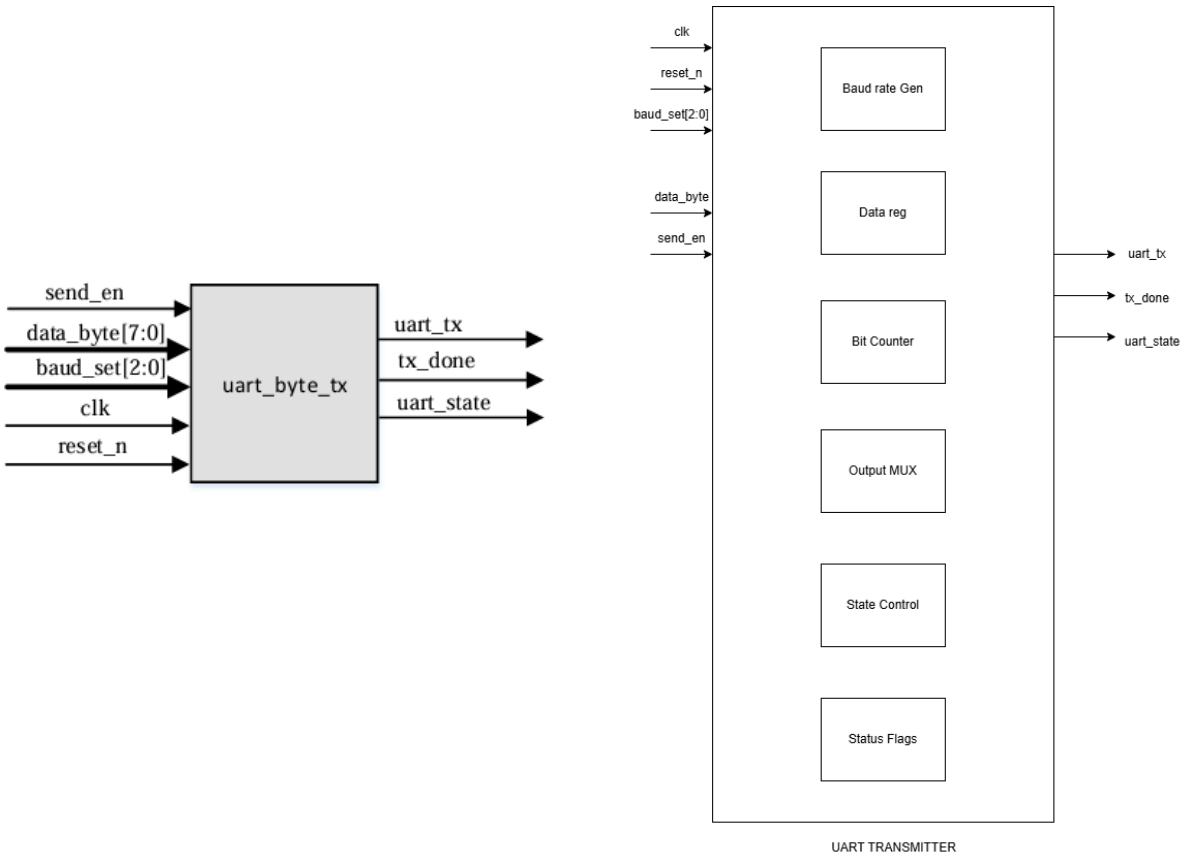
3.1 THIẾT KẾ MÔ-ĐUN TRUYỀN UART

3.1.1 Phân tích Yêu cầu và Nguyên lý

- Mô-đun truyền UART (Transmitter - TX) được thiết kế để gửi dữ liệu 8-bit qua giao tiếp UART bất đồng bộ, đáp ứng các yêu cầu sau:
- Giao diện: Nhận dữ liệu 8-bit (data_byte), tín hiệu kích hoạt (send_en), cài đặt tốc độ baud (baud_set); xuất tín hiệu UART (uart_tx), tín hiệu hoàn tất (tx_done), và trạng thái truyền (uart_state).
- Định dạng dữ liệu: 1 start bit (0), 8 data bits, 1 stop bit (1).
- Tốc độ baud: Hỗ trợ 9600, 19200, 38400, 57600, 115200, sử dụng đồng hồ 50 MHz.
- Tính ổn định: Lưu dữ liệu đầu vào vào thanh ghi để đảm bảo ổn định trong quá trình truyền.
- Nguyên lý hoạt động dựa trên việc phân tần đồng hồ 50 MHz để tạo đồng hồ tốc độ baud. Một bộ đếm bit điều khiển việc truyền tuần tự từng bit trong khung dữ liệu UART, đảm bảo thời gian chính xác.

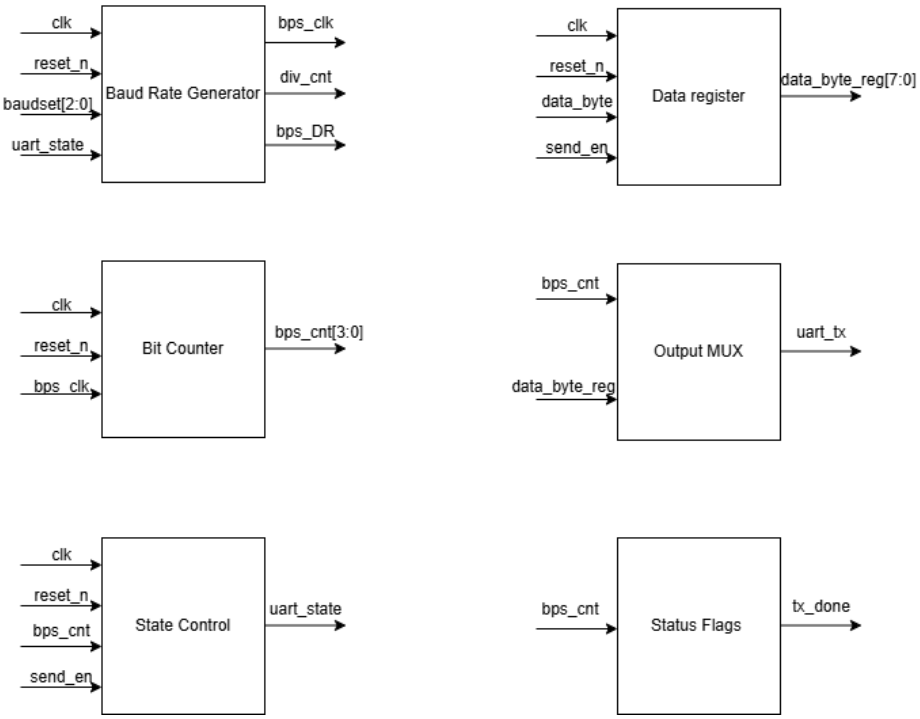
3.1.2 Sơ đồ khối

Sơ đồ khối của mô-đun truyền UART được mô tả như sau:



Hình 8: Block diagram uart transmitter

Trong đó, các khối module thành phần lần lượt như ở hình 9



Hình 9: Các khối module con bên trong UART truyền

UART Transmitter

Tổng quan hoạt động: UART Transmitter truyền dữ liệu 8-bit (data_byte) qua đường uart_tx dưới dạng khung UART (1 start bit, 8 data bits, 1 stop bit, trạng thái idle). Quá trình được điều khiển bởi đồng hồ 50 MHz (clk), tốc độ baud được chọn qua baud_set, và kích hoạt bởi send_en. Đầu ra tx_done báo hiệu hoàn tất truyền, và uart_state biểu thị trạng thái truyền.

Các khối và nguyên lý hoạt động:

1. **Baud Rate Generator**

Nguyên lý hoạt động:

- Nhận clk (50 MHz), reset_n, baud_set, và uart_state.
- Tạo tín hiệu bps_clk (đồng hồ tốc độ baud) bằng cách phân tần clk dựa trên baud_set.
- Giá trị phân tần bps_DR được chọn theo baud_set:
    - 0: 5207 (9600 baud), 1: 2603 (19200 baud), 2: 1301 (38400 baud), 3: 867 (57600 baud), 4: 433 (115200 baud).
    - Công thức: Tần số bps_clk = 50 MHz / (bps_DR + 1).
- Bộ đếm div_cnt tăng từ 0 đến bps_DR, reset về 0 khi đạt bps_DR, và tạo xung bps_clk = 1 khi div_cnt == 1 (chỉ khi uart_state cao).

Vai trò: Cung cấp đồng hồ bps_clk để đồng bộ việc truyền từng bit.

2. **Data Register**

Nguyên lý hoạt động:

- Nhận clk, reset_n, data_byte, và send_en.
- Khi send_en = 1, lưu data_byte vào thanh ghi data_byte_reg.
- Giữ nguyên data_byte_reg trong suốt quá trình truyền, reset về 0 khi reset_n = 0.

Vai trò: Lưu trữ dữ liệu 8-bit để đảm bảo dữ liệu không thay đổi trong khi truyền.

3. **Bit Counter**

Nguyên lý hoạt động:

- Nhận clk, reset_n, và bps_clk.
- Bộ đếm bps_cnt (4-bit) tăng từ 0 đến 11 mỗi khi bps_clk = 1, reset về 0 khi đạt 11 hoặc khi reset_n = 0.

Các giá trị bps_cnt tương ứng:

0: Trạng thái idle.

1: Start bit.

2-9: 8 bit dữ liệu (data_byte_reg[0-7]).

10: Stop bit.

11: Hoàn tất truyền.

Vai trò: Điều khiển thứ tự truyền các bit trong khung UART.

4. **Output MUX**

Nguyên lý hoạt động:

- Nhận clk, reset_n, bps_cnt, và data_byte_reg.
- Dựa trên bps_cnt, chọn bit để xuất ra uart_tx:

bps_cnt = 0: uart_tx = 1 (idle).

bps_cnt = 1: uart_tx = 0 (start bit).

bps_cnt = 2-9: uart_tx = data_byte_reg[0-7].

bps_cnt = 10: uart_tx = 1 (stop bit).

bps_cnt = 11 hoặc mặc định: uart_tx = 1.

Reset về uart_tx = 1 khi reset_n = 0.

Vai trò: Tạo khung UART trên đường uart_tx theo thứ tự do bps_cnt chỉ định.

5. **State Control**

Nguyên lý hoạt động:

- Nhận clk, reset_n, send_en, và bps_cnt.
- Đặt uart_state = 1 khi send_en = 1, báo hiệu bắt đầu truyền.
- Đặt uart_state = 0 khi bps_cnt = 11 (truyền hoàn tất).
- Giữ nguyên uart_state trong các trường hợp khác, reset về 0 khi reset_n = 0.

Vai trò: Quản lý trạng thái truyền (bật/tắt) để kích hoạt các khối khác.

6. **Status Flags**

Nguyên lý hoạt động:

- Nhận clk, reset_n, và bps_cnt.
- Xuất tx_done = 1 khi bps_cnt = 11, báo hiệu hoàn tất truyền một byte.
- Đặt tx_done = 0 trong các trường hợp khác, reset về 0 khi reset_n = 0.

Vai trò: Cung cấp tín hiệu phản hồi để hệ thống bên ngoài biết khi nào một byte đã được truyền xong.

Tương tác giữa các khối:

- Khởi động: Khi send_en = 1, State Control đặt uart_state = 1, kích hoạt Baud Rate Generator bắt đầu tạo bps_clk.
- Lưu dữ liệu: Đồng thời, Data Register lưu data_byte vào data_byte_reg.
- Tạo đồng hồ baud: Baud Rate Generator sử dụng baud_set để chọn bps_DR, đếm div_cnt và tạo xung bps_clk để đồng bộ.
- Điều khiển bit: Bit Counter sử dụng bps_clk để tăng bps_cnt, xác định vị trí bit trong khung UART (idle, start, data, stop).
- Xuất tín hiệu: Output MUX đọc bps_cnt và data_byte_reg để xuất từng bit ra uart_tx theo thứ tự: idle (1), start bit (0), 8 data bits, stop bit (1).
- Hoàn tất: Khi bps_cnt = 11, State Control đặt uart_state = 0 (tắt truyền), và Status Flags đặt tx_done = 1 để báo hiệu hoàn tất.

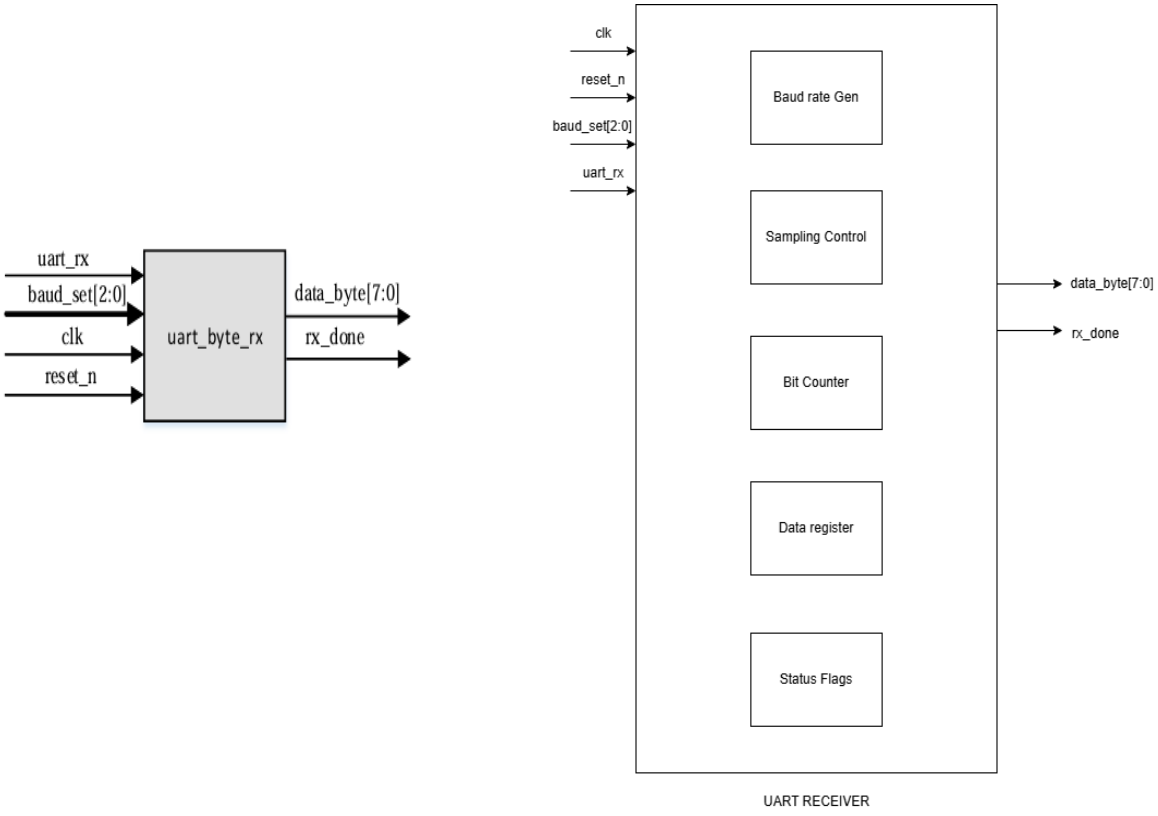
Hoạt động tổng thể:

- Khi send_en được kích hoạt, UART Transmitter bắt đầu truyền một khung UART (1 start bit, 8 data bits, 1 stop bit) qua uart_tx với tốc độ baud được chọn.
- Quá trình truyền kéo dài 10 chu kỳ bps_clk (từ bps_cnt = 1 đến bps_cnt = 10), sau đó tx_done bật để báo hiệu hoàn tất, và hệ thống trở về trạng thái idle (uart_tx = 1, uart_state = 0).

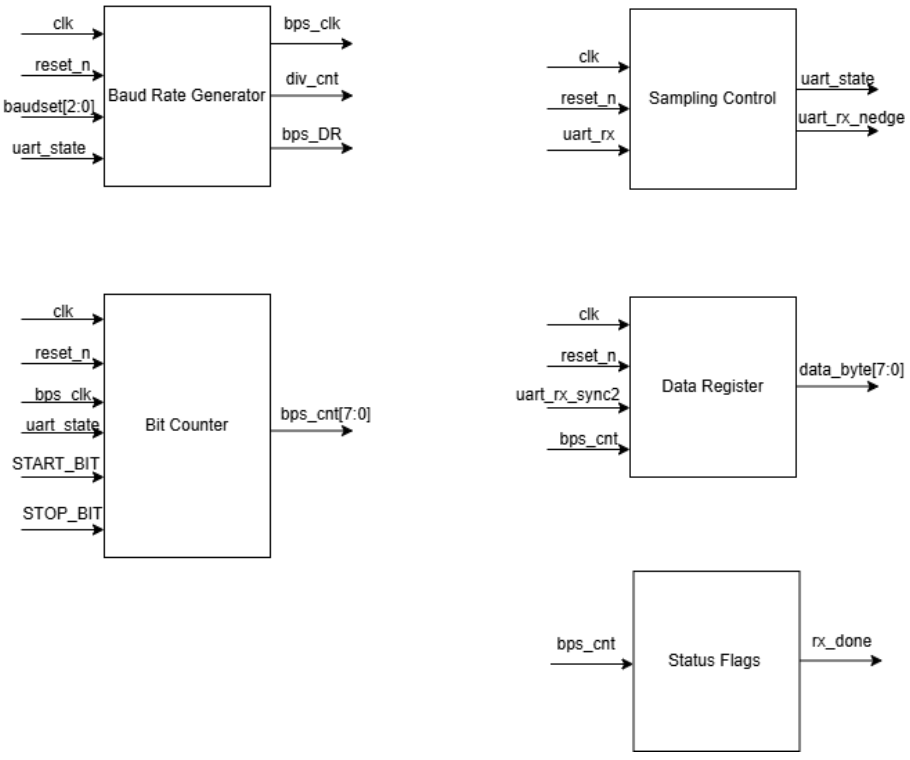
Source Code:

src_tx.rar

3.2 THIẾT KẾ MÔ-ĐUN NHẬN UART



Hình 10: Block diagram uart receiver



Hình 11: Các khối module con bên trong UART nhận

3.2.1 Lấy mẫu dữ liệu cơ bản

a) Nguyên lý lấy mẫu:

- Thông thường, dữ liệu được lấy mẫu ở giữa mỗi bit (xem Hình 12 trong tài liệu).

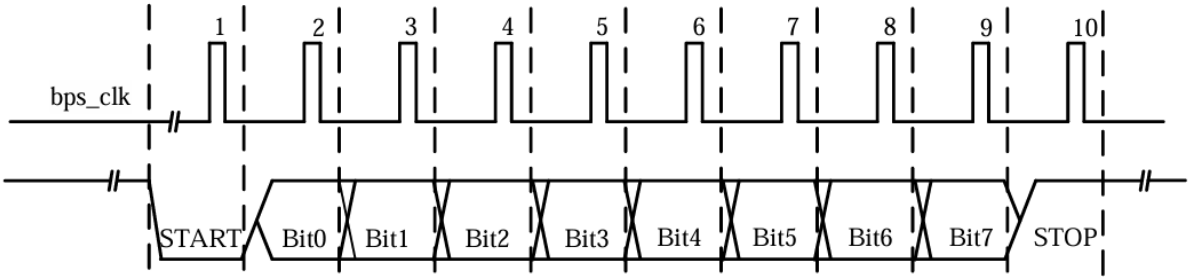


图 7-2 串口接收时序图

Hình 12: Sơ đồ thời gian nhận dữ liệu qua cổng nối tiếp

- Lý do: Điểm giữa của bit thường ổn định nhất, ít bị ảnh hưởng bởi nhiễu hoặc méo tín hiệu.

b) Hạn chế trong môi trường thực tế:

- Trong môi trường công nghiệp, nhiễu điện từ mạnh có thể làm sai lệch tín hiệu.
- Nếu chỉ lấy mẫu một lần, kết quả có thể không đáng tin cậy.

3.2.2 Phương pháp cải tiến

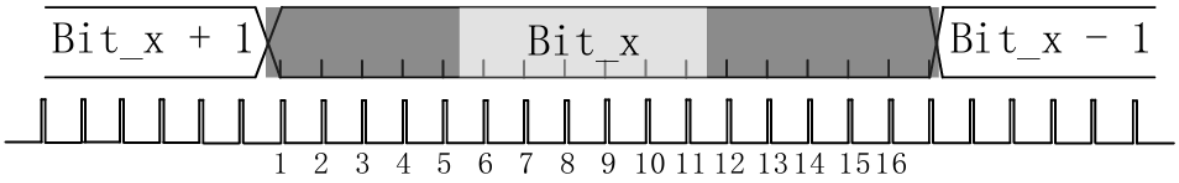


图 7-3 改进型串口接收方式示意图

Hình 13: Sơ đồ minh họa phương pháp nhận dữ liệu qua cổng nối tiếp cải tiến

Chia nhỏ bit:

- **Mỗi bit** dữ liệu được chia thành **16** đoạn nhỏ.
- Hai đoạn đầu và cuối (màu xám đậm) bị loại bỏ vì dữ liệu không ổn định khi chuyển đổi trạng thái.
- Đoạn giữa (màu xám nhạt) là vùng ổn định, được dùng để lấy mẫu.

Lấy mẫu đa lần:

- Thay vì lấy mẫu một lần, **thực hiện 6 lần lấy mẫu trong vùng ổn định**.
- Quyết định trạng thái bit (0 hoặc 1) dựa trên số lần xuất hiện nhiều nhất:

  Ví dụ: Nếu kết quả là 1/1/1/1/0/1 → Bit = 1.

  Nếu kết quả là 0/0/1/0/0/0 → Bit = 0.
- Trường hợp đặc biệt: Nếu 1 và 0 xuất hiện đều nhau (3/3), tín hiệu được coi là không đáng tin cậy và không xử lý.

Ưu điểm:

- Tăng độ chính xác trong môi trường nhiễu.
- Giảm nguy cơ lỗi do nhiễu điện từ.

3.2.3 Phát Hiện và Đồng bộ hóa tín hiệu nhận

Mục đích: Đồng bộ hóa tín hiệu uart_rx với xung nhịp FPGA (clk) để tránh hiện tượng **metastability** (trạng thái không ổn định khi tín hiệu đầu vào thay đổi gần cạnh xung nhịp).

Sơ đồ (Hình 14):

Tín hiệu uart_rx đi qua hai flip-flop loại D (uart_rx_sync1 và uart_rx_sync2) để đồng bộ hóa.

Đầu vào: clk, reset, uart_rx.

Đầu ra: uart_rx_sync2 (tín hiệu đã đồng bộ).

```
always @(posedge clk or posedge reset) begin
    if (reset) begin
        uart_rx_sync1 <= 1'b0;
        uart_rx_sync2 <= 1'b0;
    end
    else begin
        uart_rx_sync1 <= uart_rx;
        uart_rx_sync2 <= uart_rx_sync1;
    end
end
```

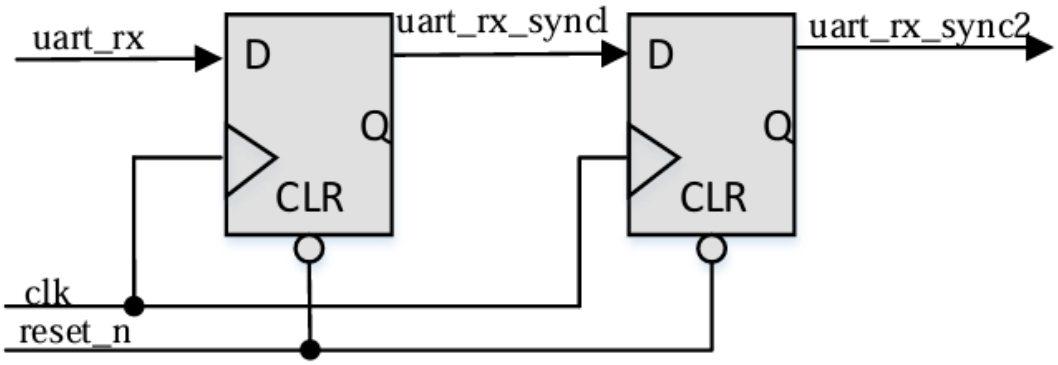


图 7-5 单 bit 信号同步示意图

Hình 14: Sơ đồ đồng bộ tín hiệu 1 bit

- Hai flip-flop được sử dụng để đồng bộ tín hiệu uart_rx với clk.
- uart_rx_sync1 lấy giá trị của uart_rx tại mỗi cạnh lên của clk.
- uart_rx_sync2 lấy giá trị của uart_rx_sync1, đảm bảo tín hiệu ổn định sau hai chu kỳ xung nhịp.
- Khi reset = 1, cả hai thanh ghi được đặt về 0.
- **Metastability:**

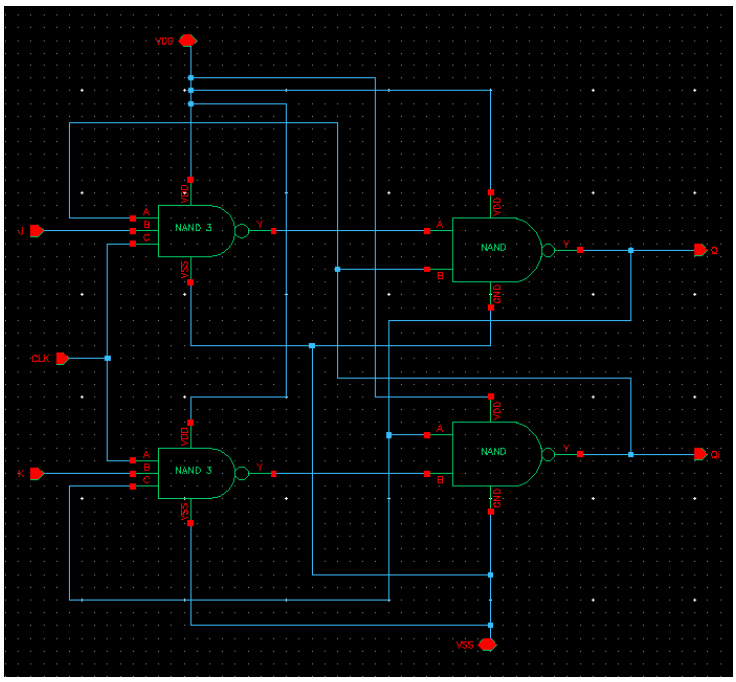
  Nếu uart_rx thay đổi gần cạnh lên của clk, flip-flop có thể rơi vào trạng thái không xác định, có nguy cơ gây glitch hoặc race condition.

  Sử dụng hai flip-flop giúp giảm xác suất xảy ra lỗi (Mean Time Between Failure - MTBF).

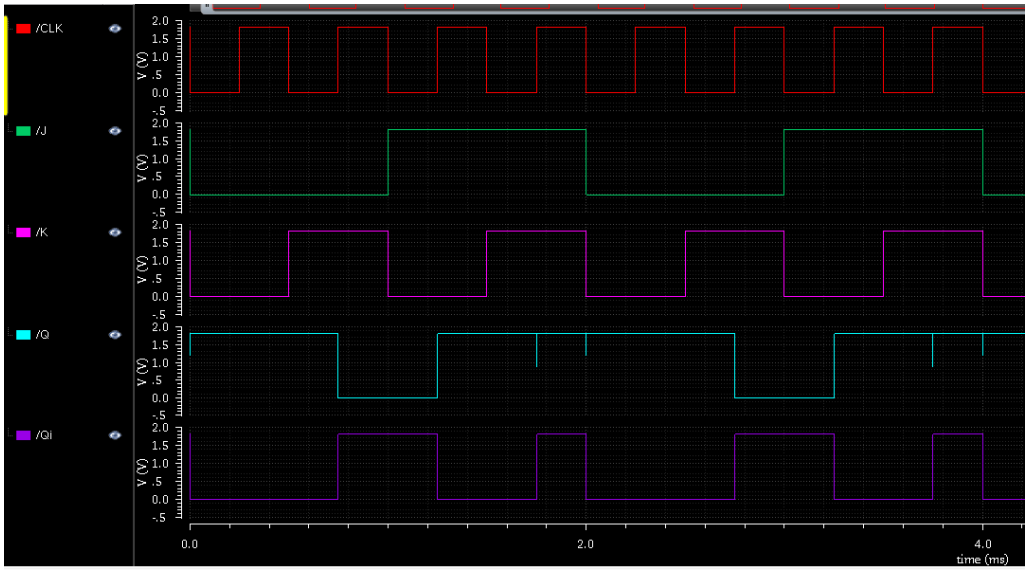
- **Glitch** là một xung ngắn không mong muốn xuất hiện ở đầu ra mạch số, thường do:

  Sự khác biệt thời gian lan truyền (delay mismatch) giữa các ngõ vào.
  Phản hồi không ổn định, đặc biệt trong mạch nhạy mức (level-sensitive).
  Mạch tổ hợp hoặc mạch có phản hồi như latch, nếu không kiểm soát tốt thời gian, dễ tạo ra glitch

  ➔ Glitch thường có độ rộng rất nhỏ (vài ns), và nếu mạch sau đọc nhầm giá trị trong lúc glitch xảy ra → gây lỗi logic.
- Race condition (trạng thái tranh chấp) là hiện tượng trong mạch số khi hai hoặc nhiều tín hiệu thay đổi gần như đồng thời, và đầu ra phụ thuộc vào thứ tự chính xác mà các tín hiệu đến.
  Ví dụ: Minh hoạ **Race condition** gây **glitch** trong mạch phản hồi
  Đây là một latch JK



Hình 15: Schematic của latch JK



Hình 16: Mô phỏng Latch JK, trường hợp Race condition gây glitch

Phân tích trường hợp race condition gây glitch trong hình 16

- Trong các trường hợp **Set, Reset, hoặc Giữ nguyên**, đầu ra Q được xác định **từ J và K độc lập với Q trước đó → không có phản hồi nội bộ → ổn định**.
- Nhưng trong **toggle**, đầu ra **phụ thuộc vào chính trạng thái Q trước đó**:

Nếu Q = 1 → toggle → phải thành 0.

Nhưng nếu latch mở lâu (CLK = 1 liên tục), phản hồi Q = 0 **vừa được sinh ra** sẽ lập tức **quay lại mạch tổ hợp**, khiến mạch toggle **nghĩ rằng Q mới là 0 → toggle lần nữa thành 1**.

Kết quả: Q → 0 → 1 trong thời gian rất ngắn → glitch thực sự.

3.2.4 Thiết kế phát hiện cạnh

Nguyên lý phát hiện cạnh

Trong giao tiếp UART, tín hiệu dữ liệu ban đầu ở trạng thái không hoạt động (idle) là mức cao (logic 1). Khi **bắt đầu truyền** một byte dữ liệu, luồng dữ liệu sẽ chuyển từ mức **cao xuống mức thấp** để tạo ra bit bắt đầu (start bit). Việc phát hiện chính xác thời điểm chuyển đổi này là vô cùng quan trọng để đồng bộ hóa quá trình nhận dữ liệu.

Để phát hiện sự chuyển đổi trạng thái này, chúng ta sử dụng kỹ thuật **phát hiện cạnh (edge detection)**. Có hai loại cạnh cần phát hiện trong các hệ thống số:

- **Cạnh lên (Rising edge)**: Khi tín hiệu chuyển từ 0 lên 1.
- **Cạnh xuống (Falling edge)**: Khi tín hiệu chuyển từ 1 xuống 0.

Trong giao thức UART, chúng ta đặc biệt quan tâm đến việc phát hiện cạnh xuống để xác định khởi đầu của một khung dữ liệu mới.

Mạch phát hiện cạnh hoạt động dựa trên nguyên lý so sánh trạng thái hiện tại với trạng thái trước đó của tín hiệu. Trong FPGA, chúng ta thực hiện việc này bằng cách:

- Sử dụng các thanh ghi D-flip-flop để lưu trữ giá trị tín hiệu tại các thời điểm khác nhau
- Tạo trễ thời gian bằng cách chuyển tín hiệu qua các thanh ghi liên tiếp
- So sánh đầu ra của các thanh ghi để xác định sự thay đổi trạng thái

Nhìn vào sơ đồ mạch phát hiện cạnh:

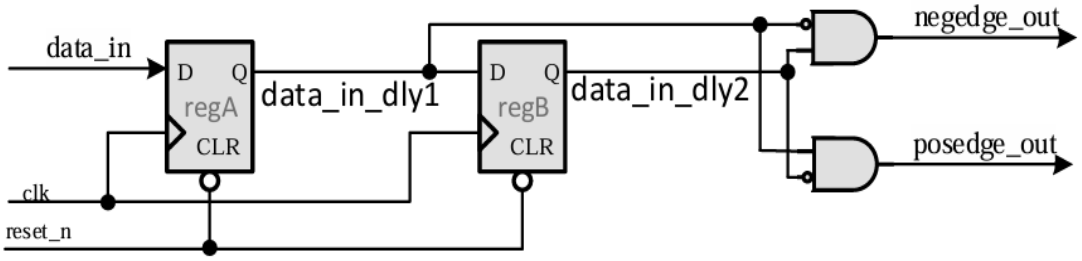
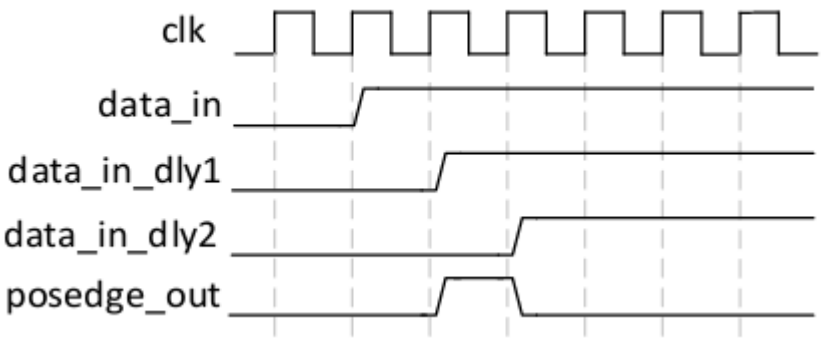


图 7-6 边缘检测原理

Hình 17: Schematic mạch phát hiện cạnh

Các thanh ghi được điều khiển bởi xung clock hệ thống, vì vậy mỗi thanh ghi sẽ tạo ra độ trễ một chu kỳ clock.

a) Phát hiện cạnh lên



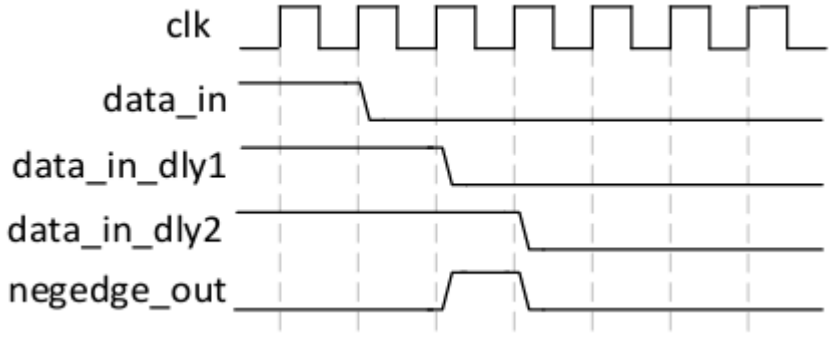
Hình 18: Waveform của nguyên lý phát hiện cạnh lên

Khi tín hiệu `data_in` chuyển từ 0 lên 1, quá trình diễn ra như sau:

1. Tại thời điểm t_0: `data_in` = 0, `data_in_dly1` (thanh ghi A) = 0, `data_in_dly2` (thanh ghi B) = 0
2. Tại thời điểm t_1: `data_in` chuyển từ 0 lên 1
3. Tại thời điểm t_2 (sau 1 xung clock): `data_in` = 1, `data_in_dly1` = 1, `data_in_dly2` = 0
    - Tại thời điểm này, `data_in_dly1 & !data_in_dly2` = 1 & !0 = 1 (phát hiện cạnh lên)
4. Tại thời điểm t_3 (sau 2 xung clock): `data_in` = 1, `data_in_dly1` = 1, `data_in_dly2` = 1
    - Tại thời điểm này, `data_in_dly1 & !data_in_dly2` = 1 & !1 = 0 (không có cạnh)

Như vậy, khi có cạnh lên, phép tính `data_in_dly1 & !data_in_dly2` sẽ cho kết quả 1 trong đúng một chu kỳ clock, tạo ra một xung báo hiệu sự xuất hiện của cạnh lên.

b) Phát hiện cạnh xuống



Hình 19: Waveform của nguyên lý phát hiện cạnh xuống

Tương tự, khi tín hiệu `data_in` chuyển từ 1 xuống 0:

1. Tại thời điểm t_0: `data_in` = 1, `data_in_dly1` = 1, `data_in_dly2` = 1

2. Tại thời điểm t_1: `data_in` chuyển từ 1 xuống 0
3. Tại thời điểm t_2 (sau 1 xung clock): `data_in` = 0, `data_in_dly1` = 0, `data_in_dly2` = 1
    - Tại thời điểm này, `!data_in_dly1 & data_in_dly2` = !0 & 1 = 1 (phát hiện cạnh xuống)
4. Tại thời điểm t_3 (sau 2 xung clock): `data_in` = 0, `data_in_dly1` = 0, `data_in_dly2` = 0
    - Tại thời điểm này, `!data_in_dly1 & data_in_dly2` = !0 & 0 = 0 (không có cạnh)

Phép tính `!data_in_dly1 & data_in_dly2` sẽ cho kết quả 1 chỉ trong chu kỳ clock ngay sau khi cạnh xuống xuất hiện.

c) Code triển khai phát hiện cạnh xuống cho UART

```
always@(posedge clk or posedge reset)
if(reset)begin
    uart_rx_reg1 <= 1'b0;// Reset thanh ghi thứ nhất về 0
    uart_rx_reg2 <= 1'b0;// Reset thanh ghi thứ hai về 0
end
else begin
    uart_rx_reg1 <= uart_rx_sync2;// Lưu giá trị hiện tại (đã đồng bộ) của tín hiệu UART
    uart_rx_reg2 <= uart_rx_reg1;// Lưu giá trị của thanh ghi thứ nhất (tức là giá trị trước đó)
end

    // Phát hiện cạnh xuống: hiện tại=0, trước đó=1
assign uart_rx_nedge = !uart_rx_reg1 & uart_rx_reg2;
```

Đoạn code trên thực hiện các chức năng:

1. Khai báo hai thanh ghi `uart_rx_reg1` và `uart_rx_reg2` để lưu trữ giá trị hiện tại và giá trị trước đó của tín hiệu UART
2. Mỗi khi có xung clock hoặc tín hiệu reset:
    - Nếu reset active, cả hai thanh ghi được đặt về 0
    - Nếu không, `uart_rx_reg1` lưu giá trị mới nhất của `uart_rx_sync2` (tín hiệu UART đã được đồng bộ), và `uart_rx_reg2` lưu giá trị trước đó của `uart_rx_reg1`
3. Tín hiệu phát hiện cạnh xuống `uart_rx_nedge` được tính bằng biểu thức: `!uart_rx_reg1 & uart_rx_reg2`, tức là `uart_rx_reg1` phải là 0 và `uart_rx_reg2` phải là 1 để phát hiện cạnh xuống

Lưu ý rằng tín hiệu `uart_rx_sync2` là tín hiệu UART đã được đồng bộ hóa với clock hệ thống để tránh hiện tượng không ổn định (metastability).

3.2.5 Thiết kế module tạo xung clock lấy mẫu

a) Tính toán tần số lấy mẫu

Trong giao tiếp UART, để đạt được độ chính xác và khả năng chống nhiễu cao, chúng ta thường không lấy mẫu tín hiệu dữ liệu với tần số baud rate, mà sử dụng tần số lấy mẫu cao hơn nhiều lần. Trong thiết kế này, chúng ta sử dụng tần số lấy mẫu bằng 16 lần tần số baud rate.

Ví dụ, với baud rate 9600, nghĩa là mỗi bit dữ liệu kéo dài 1/9600 = 104,167 µs. Với tần số lấy mẫu gấp 16 lần, mỗi bit dữ liệu sẽ được lấy 16 mẫu, và mỗi mẫu cách nhau 104,167/16 = 6,51 µs.

Với tần số clock hệ thống là 50MHz (chu kỳ 20ns), để tạo ra khoảng thời gian 6,51 µs giữa các mẫu, chúng ta cần đếm: 6,51 µs / 20 ns = 325,5 ≈
325 xung clock. Do chúng ta bắt đầu đếm từ 0, nên giá trị đếm tối đa là 324.

Bảng giá trị đếm được tính toán như sau:

波特率	波特率周期	采样时钟分频计数值	System_clk_period = 20 计数值
9600	104167ns	104167/ System_clk_period/16	325-1
19200	52083ns	52083/ System_clk_period/16	163-1
38400	26041ns	26041/ System_clk_period/16	81-1
57600	17361ns	17361/ System_clk_period/16	54-1
115200	8680ns	8680/ System_clk_period/16	27-1

Baud rate	Chu kỳ bit	Chu kỳ lấy mẫu	Giá trị đếm
9600	104167ns	6510.4ns	325-1 = 324
19200	52083ns	3255.2ns	163-1 = 162
38400	26041ns	1627.6ns	81-1 = 80
57600	17361ns	1085.1ns	54-1 = 53
115200	8680ns	542.5ns	27-1 = 26

Hình 20: Bảng tính toán giá trị lấy mẫu

b) Bộ chọn tốc độ baud

```
always@(posedge clk or posedge reset)
if(reset)
    bps_DR <= 16'd324;// Giá trị mặc định cho 9600
baud
else begin
    case(baud_set)
        0:bps_DR <= 16'd324;// 9600 baud
        1:bps_DR <= 16'd162;// 19200 baud
        2:bps_DR <= 16'd80;// 38400 baud
```

```
            3:bps_DR <= 16'd53;// 57600 baud
            4:bps_DR <= 16'd26;// 115200 baud
            default:bps_DR <= 16'd324;// Trường hợp mặc
định: 9600 baud
        endcase
end
```

Đoạn code trên thực hiện:

1. Định nghĩa một thanh ghi 16-bit `bps_DR` để lưu giá trị đếm tối đa cho bộ chia tần số
2. Khi reset, đặt giá trị mặc định tương ứng với baud rate 9600
3. Khi hoạt động bình thường, dựa vào giá trị của `baud_set` để chọn giá trị đếm phù hợp:
   - `baud_set = 0`: 9600 baud, `bps_DR = 324`
   - `baud_set = 1`: 19200 baud, `bps_DR = 162`
   - `baud_set = 2`: 38400 baud, `bps_DR = 80`
   - `baud_set = 3`: 57600 baud, `bps_DR = 53`
   - `baud_set = 4`: 115200 baud, `bps_DR = 26`
   - Các giá trị khác: mặc định 9600 baud, `bps_DR = 324`

c) Bộ tạo xung clock lấy mẫu

```
//counter
always@(posedge clk or posedge reset)
if(reset)
    div_cnt <= 16'd0;// Reset bộ đếm về 0
else if(uart_state)begin// Chỉ đếm khi đang trong
trạng thái nhận dữ liệu
    if(div_cnt == bps_DR)// Nếu đạt giá trị đếm tối
đa
        div_cnt <= 16'd0;// Reset bộ đếm về 0
    else
        div_cnt <= div_cnt + 1'b1;// Tăng bộ đếm lên
1
end
else
    div_cnt <= 16'd0;// Nếu không trong trạng thái
nhận, reset bộ đếm// bps_clk gen
always@(posedge clk or posedge reset)
if(reset)
    bps_clk <= 1'b0;// Reset tín hiệu clock lấy mẫu
về 0
else if(div_cnt == 16'd1)// Khi bộ đếm đạt giá trị 1
```

```
        bps_clk <= 1'b1;// Tạo xung 1
    else
        bps_clk <= 1'b0;// Đưa xung về 0
```

Đoạn code trên thực hiện hai nhiệm vụ chính:

1. **Bộ đếm chia tần số (div_cnt)**:
   - Đếm từ 0 đến giá trị `bps_DR` đã chọn
   - Chỉ đếm khi `uart_state = 1` (đang trong trạng thái nhận dữ liệu)
   - Reset về 0 khi đạt giá trị tối đa hoặc khi không trong trạng thái nhận
2. **Tạo xung clock lấy mẫu (bps_clk)**:
   - Tạo một xung cao (1) kéo dài một chu kỳ clock hệ thống
   - Xung được tạo ra khi `div_cnt = 1`
   - Tần số của `bps_clk` sẽ bằng tần số hệ thống chia cho (`bps_DR + 1`)

Cách làm này đảm bảo tạo ra một xung `bps_clk` ngắn (chỉ một chu kỳ clock) sau mỗi khoảng thời gian bằng với chu kỳ lấy mẫu đã tính toán.

d) Bộ đếm lấy mẫu

```
//bps counter
always@(posedge clk or posedge reset)
if(reset)
    bps_cnt <= 8'd0;// Reset bộ đếm về 0
else if(bps_cnt == 8'd159 | (bps_cnt == 8'd12 &&
(START_BIT > 2)))
    bps_cnt <= 8'd0;// Reset bộ đếm khi kết thúc
frame hoặc phát hiện lỗi bit bắt đầu
else if(bps_clk)
    bps_cnt <= bps_cnt + 1'b1;// Tăng bộ đếm khi có
xung clock lấy mẫu
else
    bps_cnt <= bps_cnt;// Giữ nguyên giá trị bộ đếm

always@(posedge clk or posedge reset)
if(reset)
    rx_done <= 1'b0;// Reset tín hiệu hoàn thành về 0
else if(bps_cnt == 8'd159)
    rx_done <= 1'b1;// Đặt tín hiệu hoàn thành khi đã
nhận đủ một frame
else
    rx_done <= 1'b0;// Giữ tín hiệu hoàn thành ở mức
thấp
```

Đoạn code này thực hiện:

1. **Bộ đếm lấy mẫu (bps_cnt)**:
    - Đếm từ 0 đến 159, tương ứng với 16 mẫu × 10 bit (1 start bit + 8 data bits + 1 stop bit)
    - Reset về 0 trong hai trường hợp:
        - Khi `bps_cnt = 159`: đã nhận đủ một frame dữ liệu
        - Khi `bps_cnt = 12` và `START_BIT > 2`: phát hiện lỗi bit bắt đầu (sẽ giải thích chi tiết sau)
    - Chỉ tăng bộ đếm khi có xung `bps_clk`
2. **Tín hiệu hoàn thành (rx_done)**:
    - Tạo một xung cao (1) khi `bps_cnt = 159`, báo hiệu đã nhận đủ một frame dữ liệu
    - Xung này kéo dài một chu kỳ clock hệ thống

Điều kiện `bps_cnt == 8'd12 && (START_BIT > 2)` là một cơ chế phát hiện lỗi thông minh:

- Sau khi lấy 6 mẫu từ bit bắt đầu (tại vị trí từ 6 đến 11), tổng giá trị được lưu trong `START_BIT`
- Nếu bit bắt đầu hợp lệ (mức thấp), tổng này phải nhỏ
- Nếu `START_BIT > 2`, nghĩa là có quá nhiều mẫu ở mức cao, bit bắt đầu không hợp lệ
- Trong trường hợp này, bộ đếm sẽ được reset và quá trình nhận bắt đầu lại

3.2.6 Thiết kế module nhận dữ liệu lấy mẫu

a) Nguyên lý lấy mẫu nâng cao

Trong môi trường công nghiệp có nhiễu điện từ mạnh, việc chỉ lấy một mẫu ở giữa mỗi bit dữ liệu không đủ đảm bảo độ tin cậy. Để tăng cường khả năng chống nhiễu, thiết kế này sử dụng phương pháp lấy nhiều mẫu và quyết định giá trị bit dựa trên đa số.

Mỗi bit dữ liệu được chia thành 16 đoạn nhỏ (tương ứng với 16 mẫu). Tuy nhiên, không phải tất cả các mẫu đều có giá trị như nhau:

1. **Vùng chuyển tiếp** (đoạn 0-5 và 12-15): Đây là vùng gần với các ranh giới giữa các bit, dữ liệu có thể không ổn định do hiệu ứng chuyển tiếp. Các mẫu trong vùng này không được sử dụng.
2. **Vùng ổn định** (đoạn 6-11): Đây là vùng ở giữa mỗi bit, dữ liệu thường ổn định nhất. Chúng ta lấy 6 mẫu trong vùng này và sử dụng quy tắc đa số để xác định giá trị bit.

Giả sử chúng ta lấy 6 mẫu và kết quả là: 1, 1, 1, 1, 0, 1

- Có 5 mẫu là 1 và 1 mẫu là 0
- Đa số các mẫu là 1, vì vậy giá trị bit được xác định là 1

Tương tự, nếu kết quả lấy mẫu là: 0, 0, 1, 0, 0, 0

- Có 5 mẫu là 0 và 1 mẫu là 1
- Đa số các mẫu là 0, vì vậy giá trị bit được xác định là 0

Trong trường hợp số mẫu 0 và 1 bằng nhau (3-3), tín hiệu được coi là không tin cậy.

b) Phân tích vị trí lấy mẫu chi tiết

Thiết kế sử dụng biến `bps_cnt` để theo dõi vị trí lấy mẫu trong toàn bộ frame. Giá trị của `bps_cnt` tăng lên mỗi khi có xung `bps_clk`, tạo ra 160 vị trí từ 0 đến 159 cho toàn bộ frame (16 vị trí × 10 bit).

Đối với mỗi bit, 6 vị trí lấy mẫu trong vùng ổn định được phân bố như sau:

- **Bit bắt đầu**: vị trí 6, 7, 8, 9, 10, 11
- **Bit dữ liệu 0**: vị trí 22, 23, 24, 25, 26, 27
- **Bit dữ liệu 1**: vị trí 38, 39, 40, 41, 42, 43
- Và tiếp tục với khoảng cách 16 vị trí giữa các bit

Có thể thấy rằng vị trí đầu tiên của mỗi bit cách vị trí đầu tiên của bit trước đó đúng 16 đơn vị (ví dụ: 6 → 22 → 38).

c) Code triển khai lấy mẫu

```
always@(posedge clk or posedge reset)
if(reset)begin
    START_BIT <= 3'd0;// Reset bộ đếm bit bắt đầu
    data_byte_pre[0] <= 3'd0;// Reset các bộ đếm bit
dữ liệu
    data_byte_pre[1] <= 3'd0;
    data_byte_pre[2] <= 3'd0;
    data_byte_pre[3] <= 3'd0;
    data_byte_pre[4] <= 3'd0;
    data_byte_pre[5] <= 3'd0;
    data_byte_pre[6] <= 3'd0;
    data_byte_pre[7] <= 3'd0;
    STOP_BIT <= 3'd0;// Reset bộ đếm bit kết thúc
end
else if(bps_clk)begin// Chỉ thực hiện khi có xung
clock lấy mẫu
    case(bps_cnt)
        0:begin// Khởi tạo tất cả bộ đếm khi bắt đầu
frame mới
```

```
            START_BIT <= 3'd0;
            data_byte_pre[0] <= 3'd0;
            data_byte_pre[1] <= 3'd0;
            data_byte_pre[2] <= 3'd0;
            data_byte_pre[3] <= 3'd0;
            data_byte_pre[4] <= 3'd0;
            data_byte_pre[5] <= 3'd0;
            data_byte_pre[6] <= 3'd0;
            data_byte_pre[7] <= 3'd0;
            STOP_BIT <= 3'd0;
        end
// Lấy mẫu cho bit bắt đầu tại vị trí 6-11
        6 ,7 ,8 ,9 ,10,11:START_BIT <= START_BIT +
uart_rx_sync2;

// Lấy mẫu cho 8 bit dữ liệu, mỗi bit tại 6 vị trí

22,23,24,25,26,27:data_byte_pre[0]<=data_byte_pre[0]+
uart_rx_sync2;
        38,39,40,41,42,43:data_byte_pre[1] <=
data_byte_pre[1] + uart_rx_sync2;
        54,55,56,57,58,59:data_byte_pre[2] <=
data_byte_pre[2] + uart_rx_sync2;
        70,71,72,73,74,75:data_byte_pre[3] <=
data_byte_pre[3] + uart_rx_sync2;
        86,87,88,89,90,91:data_byte_pre[4] <=
data_byte_pre[4] + uart_rx_sync2;

102,103,104,105,106,107:data_byte_pre[5]<=data_byte_p
re[5]+uart_rx_sync2;

118,119,120,121,122,123:data_byte_pre[6]<=data_byte_p
re[6]+uart_rx_sync2;

134,135,136,137,138,139:data_byte_pre[7]<=data_byte_p
re[7]+uart_rx_sync2;

// Lấy mẫu cho bit kết thúc tại vị trí 150-155
        150,151,152,153,154,155:STOP_BIT <= STOP_BIT
+ uart_rx_sync2;
```

```
            default:// Giữ nguyên giá trị tại các vị trí
    khác
            begin
                START_BIT <= START_BIT;
                data_byte_pre[0] <= data_byte_pre[0];
                data_byte_pre[1] <= data_byte_pre[1];
                data_byte_pre[2] <= data_byte_pre[2];
                data_byte_pre[3] <= data_byte_pre[3];
                data_byte_pre[4] <= data_byte_pre[4];
                data_byte_pre[5] <= data_byte_pre[5];
                data_byte_pre[6] <= data_byte_pre[6];
                data_byte_pre[7] <= data_byte_pre[7];
                STOP_BIT <= STOP_BIT;
            end
        endcase
    end
```

Đoạn code này thực hiện các nhiệm vụ:

1. **Định nghĩa các biến tích lũy**:
   - `START_BIT`: biến 3-bit để tích lũy giá trị của bit bắt đầu
   - `data_byte_pre[0:7]`: mảng 8 biến 3-bit để tích lũy giá trị của 8 bit dữ liệu
   - `STOP_BIT`: biến 3-bit để tích lũy giá trị của bit kết thúc
2. **Khởi tạo biến tích lũy**:
   - Khi reset hoặc tại vị trí 0, tất cả biến được đặt về 0
3. **Quá trình lấy mẫu**:
   - Chỉ thực hiện khi có xung `bps_clk`
   - Tại các vị trí xác định (ví dụ: 6-11 cho bit bắt đầu), đọc giá trị của `uart_rx_sync2` và cộng vào biến tích lũy tương ứng
   - Với mỗi bit, thực hiện 6 lần lấy mẫu tại vùng ổn định
4. **Giữ nguyên giá trị tích lũy**:
   - Tại các vị trí không xác định (default), giữ nguyên giá trị của tất cả biến

3.2.7 Module phán đoán trạng thái dữ liệu

a) Nguyên lí phán đoán trạng thái dữ liệu

Như đã giới thiệu trong phần lý thuyết, để nhận một bit dữ liệu, chúng ta cần lấy mẫu 6 lần và chọn trạng thái xuất hiện nhiều lần nhất làm kết quả cuối cùng. Cụ thể, chỉ khi trạng thái đó xuất hiện nhiều hơn 3 lần trong 6 lần lấy mẫu, nó mới được coi là dữ liệu hợp lệ.

Việc phán đoán này có thể thực hiện bằng cách so sánh giá trị `data_byte_pre[n]` với một bộ so sánh số, hoặc đơn giản là lấy bit cao nhất của dữ liệu.

Giải thích cụ thể: Khi `data_byte_pre[n]` lần lượt là các giá trị nhị phân 011B, 010B, 100B, 101B (tương ứng với giá trị thập phân 3, 2, 4, 5), có thể nhận thấy các giá trị lớn hơn hoặc bằng 4 là 100B và 101B. Khi bit cao nhất là 1, tức giá trị tích lũy lớn hơn hoặc bằng 4, có thể kết luận giá trị thực của dữ liệu là 1. Khi bit cao nhất là 0, tức giá trị tích lũy nhỏ hơn hoặc bằng 3, có thể kết luận giá trị thực của dữ liệu là 0. Vì vậy, chỉ cần kiểm tra bit cao nhất là đủ.

b) Code triển khai

```
always@(posedge clk or posedge reset)
if(reset)
    data_byte <= 8'd0;
else if(bps_cnt == 8'd159)begin
    data_byte[0] <= data_byte_pre[0][2];
    data_byte[1] <= data_byte_pre[1][2];
    data_byte[2] <= data_byte_pre[2][2];
    data_byte[3] <= data_byte_pre[3][2];
    data_byte[4] <= data_byte_pre[4][2];
    data_byte[5] <= data_byte_pre[5][2];
    data_byte[6] <= data_byte_pre[6][2];
    data_byte[7] <= data_byte_pre[7][2];
end
```

Phân tích:

- Khối `always` này được kích hoạt bởi cạnh lên của xung clock hoặc tín hiệu reset
- Khi reset được kích hoạt, `data_byte` được đặt về 0
- Khi `bps_cnt` đạt giá trị 159 (kết thúc việc nhận một byte dữ liệu), các bit dữ liệu được lấy từ bit thứ 2 (bit cao nhất) của mỗi `data_byte_pre[n]`
- `data_byte_pre[n][2]` chính là bit thứ 2 (MSB) của giá trị tích lũy cho mỗi bit dữ liệu, xác định giá trị cuối cùng của bit đó

CHƯƠNG 4: TESTBENCH

4.1 MODULE TESTBENCH

Sau khi hoàn thành thiết kế, tiến hành kiểm thử chức năng thông qua mô phỏng. Trong file testbench, tín hiệu đầu vào được tạo ra bằng cách sử dụng đầu ra từ module truyền (uart_byte_tx). Vì vậy, file khai báo input này chỉ cần sửa đổi thông tin port, khởi tạo module này và kết nối đầu ra `uart_tx` của module truyền tới đầu vào `uart_rx` của module nhận. Một phần của file testbench sau khi sửa đổi như sau:

```
wire uart_tx_rx;
uart_byte_tx uart_byte_tx(
    .clk(clk),
    .reset_n(reset_n),
    .data_byte(data_byte_tx),
    .send_en(send_en),
    .baud_set(baud_set),
    .uart_tx(uart_tx_rx),
    .tx_done(tx_done),
    .uart_state(uart_state)
);

uart_byte_rx uart_byte_rx(
    .clk(clk),
    .reset_n(reset_n),
    .baud_set(baud_set),
    .uart_rx(uart_tx_rx),
    .data_byte(data_byte_rx),
    .rx_done(rx_done)
);
```

Phân tích:

- Khai báo một dây `uart_tx_rx` để kết nối giữa module truyền và module nhận
- Khởi tạo module truyền `uart_byte_tx` với các kết nối tương ứng, đầu ra `uart_tx` được kết nối tới dây `uart_tx_rx`
- Khởi tạo module nhận `uart_byte_rx` với các kết nối tương ứng, đầu vào `uart_rx` được kết nối từ dây `uart_tx_rx`
- Cấu hình baud rate giống nhau cho cả hai module thông qua `baud_set`

Sau khi thiết kế file testbench, mở phần mềm Modelsim, tạo dự án mới để mô phỏng chức năng. Kết quả mô phỏng hiển thị trong hình 7-9. Module truyền UART `uart_tx` lần lượt gửi 0xaa và 0x55, và module nhận UART `uart_rx` nhận dữ liệu

nối tiếp và chuyển đổi thành đầu ra song song với giá trị 0xaa và 0x55, hoàn toàn phù hợp với dữ liệu do module truyền gửi đi.

4.2 MODULE TOP

4.2.1 Tạo module top

Tạo một file mới `uart_rx_test.v` và thiết lập nó làm module thiết kế cấp cao nhất. Khởi tạo cả `uart_byte_rx` và `uart_byte_tx` trong module này. Mỗi khi module nhận UART nhận được một byte, nó tạo ra tín hiệu `rx_done`. Lúc này, gán tín hiệu `rx_done` cho `send_en`, và gửi lại dữ liệu vừa nhận được thông qua module truyền UART. Code của module kiểm thử nhận UART như sau:

```
module uart_rx_test(
    input clk,
    input reset_n,
    input uart_rx,
    output uart_tx
);

reg [7:0] data_byte_tx;
wire [7:0] data_byte_rx;
reg send_en;
wire tx_done;
wire rx_done;

always@(posedge clk or negedge reset_n)
if(!reset_n) begin
    data_byte_tx <= 8'd0;
    send_en <= 1'd0;
end
else if(rx_done) begin
    data_byte_tx <= data_byte_rx;
    send_en <= rx_done;
end
else begin
    data_byte_tx <= data_byte_tx;
    send_en <= 1'd0;
end

uart_byte_tx uart_byte_tx(
    .clk(clk),
    .reset_n(reset_n),
    .data_byte(data_byte_tx),
```

```
        .send_en(send_en),
        .baud_set(3'd0),
        .uart_tx(uart_tx),
        .tx_done(tx_done),
        .uart_state( )
    );

    uart_byte_rx uart_byte_rx(
        .clk(clk),
        .reset_n(reset_n),
        .baud_set(3'd0),
        .uart_rx(uart_rx),
        .data_byte(data_byte_rx),
        .rx_done(rx_done)
    );

    Endmodule
```

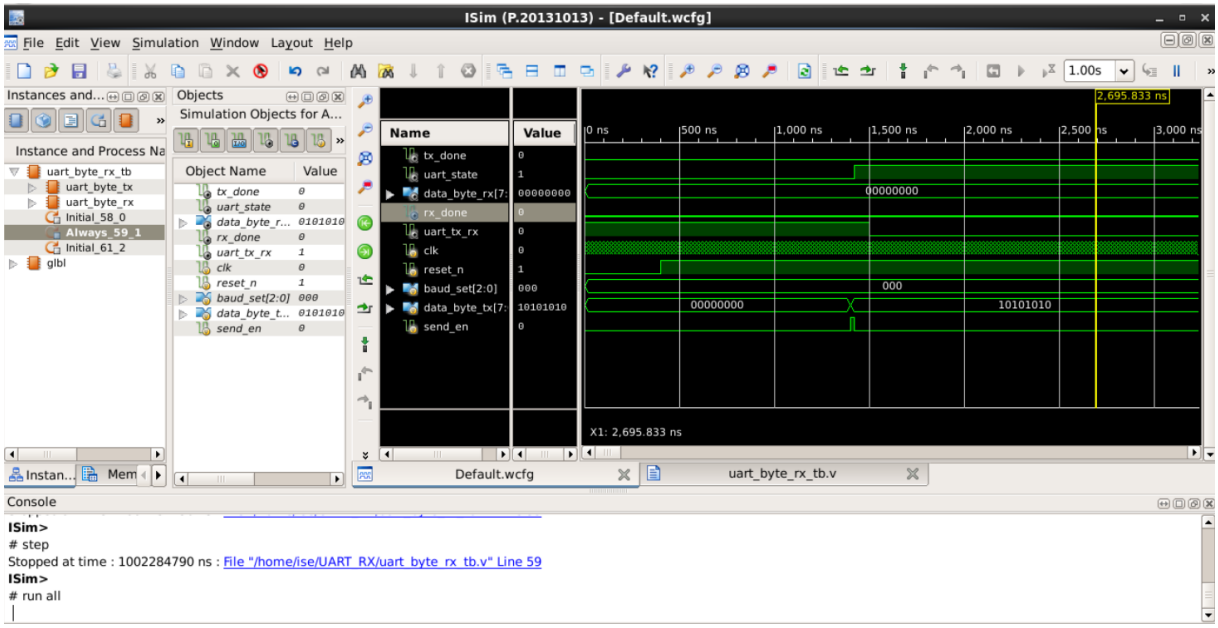
Phân tích:

- Module `uart_rx_test` có cổng vào `clk`, `reset_n`, `uart_rx` và cổng ra `uart_tx`
- Khai báo các thanh ghi và dây cần thiết:
  - `data_byte_tx`: thanh ghi 8 bit lưu dữ liệu cần truyền
  - `data_byte_rx`: dây 8 bit nhận dữ liệu từ module nhận
  - `send_en`: thanh ghi điều khiển bắt đầu truyền
  - `tx_done`: dây báo hiệu hoàn thành truyền
  - `rx_done`: dây báo hiệu hoàn thành nhận
- Khối `always` xử lý logic chính:
  - Nếu `reset_n` không kích hoạt, khởi tạo `data_byte_tx` và `send_en` về 0
  - Nếu `rx_done` kích hoạt (đã nhận xong một byte), gán dữ liệu nhận được `data_byte_rx` cho `data_byte_tx` để chuẩn bị gửi lại, và kích hoạt `send_en`
  - Trường hợp khác, giữ nguyên `data_byte_tx` và đặt `send_en` về 0
- Khởi tạo module truyền `uart_byte_tx`:
  - Đầu vào `data_byte` kết nối tới `data_byte_tx`
  - Đầu vào `send_en` kết nối tới thanh ghi `send_en`
  - Đầu vào `baud_set` đặt cố định là 3'd0 (tốc độ 9600 baud)
  - Đầu ra `uart_tx` kết nối tới cổng ra `uart_tx` của module kiểm thử
- Khởi tạo module nhận `uart_byte_rx`:
  - Đầu vào `uart_rx` kết nối tới cổng vào `uart_rx` của module kiểm thử
  - Đầu vào `baud_set` đặt cố định là 3'd0 (tốc độ 9600 baud)
  - Đầu ra `data_byte` kết nối tới dây `data_byte_rx`

- Đầu ra `rx_done` kết nối tới dây `rx_done`

Thiết kế này tạo ra một mạch vòng lặp (loop-back) UART: Khi dữ liệu được nhận qua `uart_rx`, nó được xử lý và ngay lập tức được gửi trả lại qua `uart_tx`. Cơ chế này rất hữu ích để kiểm tra chức năng nhận và truyền của UART trên thực tế.

4.2.2 Waveform



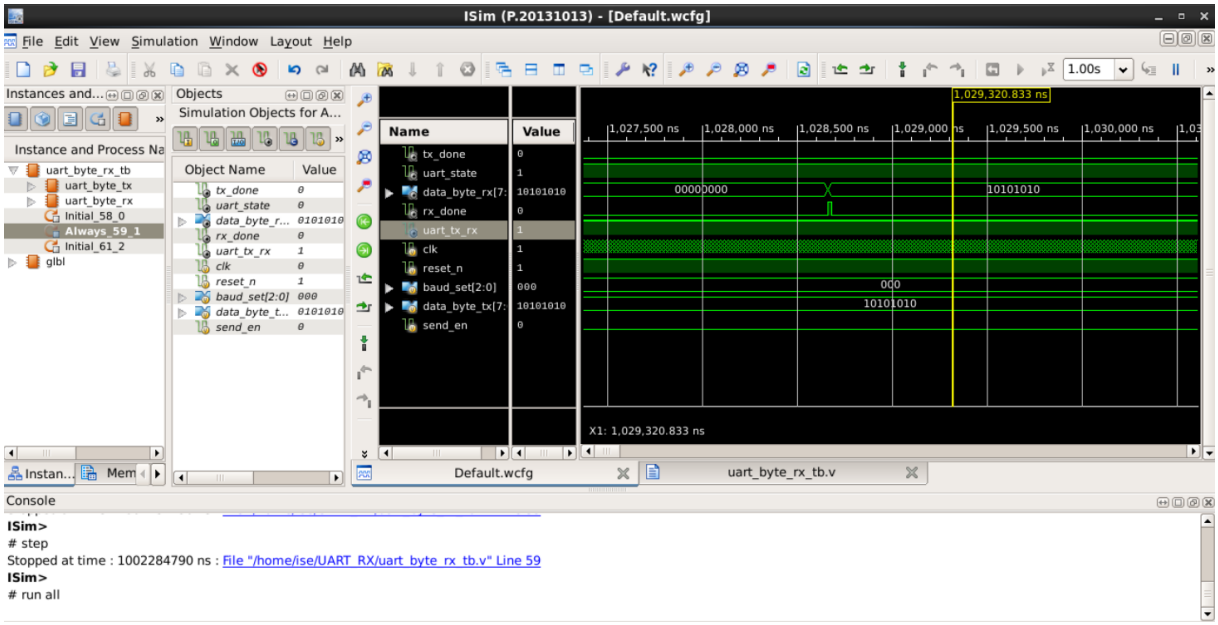
Hình 21: Waveform trong khoảng thời gian 0 – 3000ns

1. Phân tích các tín hiệu trong khoảng thời gian từ 0 – 3000ns:

- Cấu hình ban đầu:
  - baud_set[2:0] = 000 (baud rate 9600 - chậm nhất)
  - reset_n = 1 (hệ thống đã được reset và hoạt động bình thường)
  - clk chạy liên tục với chu kỳ 20ns (50MHz)
- Quá trình truyền byte đầu tiên (10101010):
  - Tại khoảng 1400ns:
    - data_byte_tx[7:0] = 10101010 (10101010 được load vào)
    - send_en có 1 xung ngắn để bắt đầu truyền
    - uart_state = 1 (TX module bắt đầu hoạt động)
    - uart_tx_rx (đường truyền UART **):** Bắt đầu ở mức HIGH (idle), sau đó kéo xuống LOW vì truyền Start bit (bit 0)
- Quá trình nhận dữ liệu:
  - Module RX nhận tín hiệu từ uart_tx_rx
  - data_byte_rx[7:0] ban đầu = 00000000
  - Sau khi nhận xong: data_byte_rx[7:0] = 10101010 (0xAA)
  - rx_done sẽ có xung báo nhận xong (không hiện rõ trong khung thời gian này)
- Đặc điểm thời gian:
  - Với baud_set = 0, mỗi bit UART mất khoảng 104.16µs (9600 baud)

- Tuy nhiên trong hình waveform chỉ hiển thị đến 3000ns, chưa đủ để thấy hoàn thành 1 byte

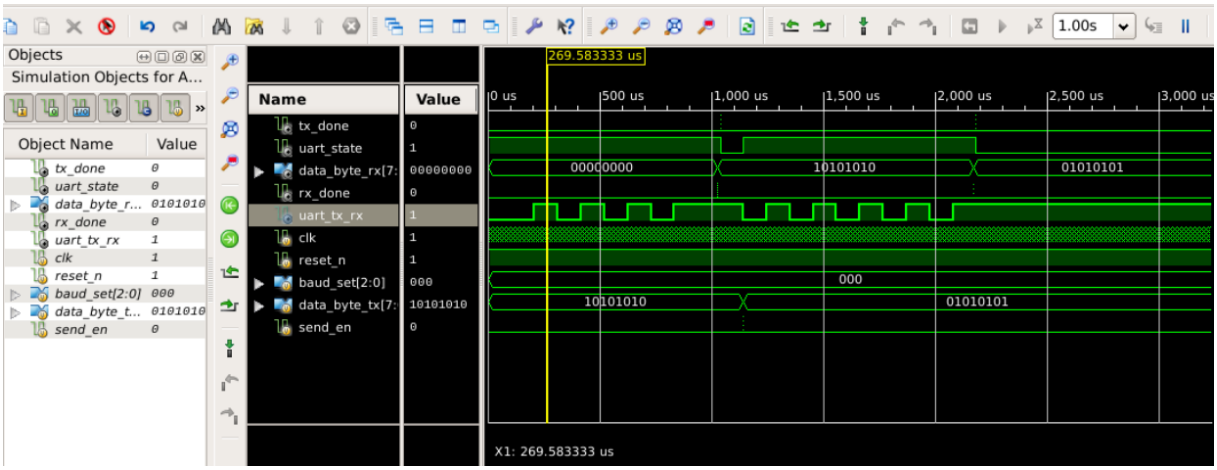
- Trạng thái hệ thống khi đến 3000ns:
  - tx_done = 0 (chưa hoàn thành truyền)
  - Hệ thống đang trong quá trình truyền bit Start



Hình 22: Waveform trong khoảng thời gian 1.027.000ns đến 1.030.000ns

2. Phân tích các tín hiệu trong khoảng thời gian từ 1.027.000ns đến 1.030.000ns:

- Thời điểm 1.028.500ns:
  - Đây là cuối quá trình truyền byte đầu tiên (0xAA)
  - Thời gian truyền 1 byte với baud rate 9600: ~1.04ms (10 bits × 104.16µs/bit)
- Trạng thái các tín hiệu:
  - uart_tx_rx (đường truyền):
    - uart_tx_rx (đường truyền): Đang ở mức HIGH (stop bit hoặc idle state)
    - Chuẩn bị kết thúc frame hiện tại
  - data_byte_rx[7:0] = 10101010:
    - Module RX đã nhận thành công byte 10101010
    - Giá trị này đã được decode đúng từ tín hiệu UART
  - rx_done:
    - Chuyển từ 0 lên 1 trong 1 chu kì clk để báo hiệu hoàn thành nhận
- Chuẩn bị cho byte thứ hai:
  - data_byte_tx[7:0] = 10101010, sẽ sớm chuyển thành 01010101
  - Theo testbench, byte thứ hai sẽ được gửi sau khi tx_done = 1 (không hiển thị trong hình này)
- Trạng thái hệ thống:
  - uart_state = 1: TX vẫn đang active (chưa hoàn thành hoàn toàn)
  - tx_done = 0: Chưa báo hoàn thành truyền
  - Hệ thống đang trong giai đoạn chuyển tiếp



Hình 23: Toàn bộ quá trình truyền 2 byte trong testbench

3. Phân tích các tín hiệu trong Toàn bộ quá trình truyền 2 byte:

- Byte đầu tiên (0xAA = 10101010):
  - Thời gian: ~500µs - 1500µs
    - data_byte_tx[7:0] = 10101010 (0xAA)
    - send_en có xung ngắn tại ~500µs để bắt đầu truyền
    - uart_state = 1 trong suốt quá trình truyền
    - uart_tx_rx hiển thị chuỗi bit UART: Start(0) + 8 data bits + Stop(1)
  - Kết quả nhận:
    - data_byte_rx[7:0] thay đổi từ 00000000 → 10101010 (nhận đúng 0xAA)
    - rx_done có xung báo hoàn thành nhận
- Byte thứ hai (0x55 = 01010101):
  - Thời gian: ~2000µs - 3000µs
    - data_byte_tx[7:0] chuyển thành 01010101
    - send_en có xung thứ hai để bắt đầu truyền byte mới.
    - uart_state lại = 1 cho quá trình truyền thứ hai.
    - uart_tx_rx hiển thị pattern bit khác (đảo ngược so với 0xAA).
  - Kết quả nhận:

data_byte_rx[7:0] thay đổi từ 10101010 → 01010101 (nhận đúng 0x55)

- Đặc điểm thời gian:
  - Khoảng cách giữa 2 lần truyền: ~500µs (cho tx_done và chuẩn bị byte mới).
  - Thời gian truyền mỗi byte: ~1000µs (phù hợp với baud rate 9600).
  - tx_done có xung ngắn sau mỗi lần truyền xong.
- Đánh giá hoạt động:
  - Module TX hoạt động đúng: truyền đúng 2 byte theo thứ tự.
  - Module RX hoạt động đúng: nhận chính xác cả 2 byte.
  - Timing đúng chuẩn UART với baud rate đã set.
  - Testbench chạy thành công.

Source code:

src_rx.rar

TÀI LIỆU THAM KHẢO

[1] Microcontrollerslab.com, *UART Communication Introduction, Working, Frame Format, Applications*.

Link: https://microcontrollerslab.com/uart-communication-working-applications/

[2] Xiaomeige Electronics, *高云开发板FPGA逻辑设计与验证教程V1.2 (Gaoyun Development Board FPGA Logic Design and Verification Tutorial V1.2)*.