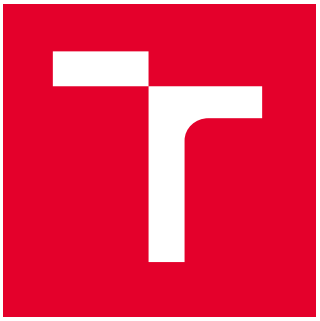


BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF MICROELECTRONICS

ÚSTAV MIKROELEKTRONIKY

DESIGN AND VERIFICATION OF USART PERIPHERAL

NÁVRH A VERIFIKACE PERIFÉRIE USART

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Artem Gumenyuk**
AUTOR PRÁCE

SUPERVISOR — **doc. Ing. Jana Drbohlavová, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Mikroelektronika**

Ústav mikroelektroniky

Student: Bc. Artem Gumenyuk ***ID:*** 228619

Ročník: 2 ***Akademický rok:*** 2024/25

NÁZEV TÉMATU:

Návrh a verifikace periférie USART

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s komunikačním protokolem USART (univerzální synchronní/asynchronní přijímač a vysílač) a s možností připojení periferií k mikroprocesoru pomocí sběrnice AMBA APB4.

Navrhněte periferní zařízení USART, které bude mít registry přístupné prostřednictvím protokolu AMBA APB4. Mezi požadavky, které musí tato periferie splňovat patří: podpora synchronního i asynchronního režimu, plný i poloviční duplex, dále možnost konfigurovat počet datových bitů, počet stop bitů, paritu a nastavení přenosové rychlosti a také vstupní a výstupní vyrovnávací paměť FIFO s připojením řadiče DMA, včetně podpory jednoho a více přenosů.

K dalším důležitým funkcím patří podpora maskovatelných přerušení při detekci chyby, plné nebo prázdné FIFO atd. a rovněž povolení ovladače pinů (DE) pro aktivaci externího transceiveru.

Poté vypracujte podrobnou funkční specifikaci periferie ve formátu EARS (The Easy Approach to Requirements Syntax).

Na základě specifikace implementujte periferii v jazyce SystemVerilog. Napište plán verifikace a pomocí UVM frameworku vytvořte verifikační prostředí pro ověření funkčnosti návrhu.

Výstupem práce by měl být plně verifikovaný a syntetizovatelný RTL se 100% pokrytím kódu a 100% pokrytím funkčnosti.

DOPORUČENÁ LITERATURA:

According to recommendations of the supervisor.

Termín zadání: 10.2.2025 ***Termín odevzdání:*** 26.5.2025

Vedoucí práce: doc. Ing. Jana Drbohlavová, Ph.D.

doc. Ing. Lukáš Fujcik, Ph.D.
předseda rady studijního programu

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Microelectronics**

Department of Microelectronics

Student: Bc. Artem Gumenyuk ***ID:*** 228619

Year of study: 2 ***Academic year:*** 2024/25

TITLE OF THESIS:

Design and Verification of USART Peripheral

INSTRUCTION:

Learn about the USART communication protocol (Universal Synchronous/Asynchronous Receiver and Transmitter) and the possibility of connecting peripherals to the microprocessor using the AMBA APB4 bus.
Design a USART peripheral that will have registers accessible via the AMBA APB4 protocol. The requirements that this peripheral must meet include support for synchronous and asynchronous modes, full and half duplex, as well as the ability to configure the number of data bits, number of stop bits, parity and baud rate settings, and input and output FIFO buffering with DMA controller connections, including support for single and multiple transfers. Other important features include support for maskable interrupts on error detection, full or empty FIFOs, etc., as well as pin driver (DE) enable for external transceiver activation.
Finally develop a detailed functional specification of the peripheral in EARS (The Easy Approach to Requirements Syntax) format. Based on the specification, implement the peripheral in the SystemVerilog language. Write a verification plan and use the UVM framework to create a verification environment to verify the functionality of the design. The output of the work should be a fully verified and synthesizable RTL with 100% code coverage and 100% functionality coverage.

RECOMMENDED LITERATURE:

According to recommendations of the supervisor.

Date of project specification: 10.2.2025 ***Deadline for submission:*** 26.5.2025

Supervisor: doc. Ing. Jana Drbohlavová, Ph.D.

doc. Ing. Lukáš Fujcik, Ph.D.
Chair of study program board

ABSTRACT

This master's thesis presents a theoretical introduction to the USART serial communication protocol, necessary hardware design parts, AMBA APB parallel communication protocol, and object-oriented verification using UVM and UVMF. The USART peripheral device is designed to be parameterizable and configurable. The design includes support for adjustable baud rates, selectable data word length, parity control, and flexible stop bit settings. Additional features include the ability to select synchronous or asynchronous communication mode, full or half duplex, DMA (Direct Memory Access) support, interrupt outputs, and others. The thesis results include a USART hardware design architecture, design specification, a verification environment architecture, verification flow, synthesis and coverage results.

KEYWORDS

Clocks, Universal Synchronous and Asynchronous Receiver-Transmitter (USART), Universal Asynchronous Receiver-Transmitter (UART), functional verification, UVM, UVMF, APB, serial communication, embedded systems, SystemVerilog

ABSTRAKT

Tato diplomová práce představuje teoretický úvod do sériového komunikačního protokolu USART, nezbytné části návrhu hardwaru, paralelní komunikační protokol AMBA APB a objektově orientovanou verifikaci s použitím UVM a UVMF. Periferní zařízení je navrženo tak, aby bylo parametrizovatelné a konfigurovatelné. Návrh zahrnuje podporu nastavitelných přenosových rychlostí, volitelnou délku datového slova, řízení parity a flexibilní nastavení stop bitů. Mezi další funkce patří možnost volby synchronního nebo asynchronního komunikačního režimu, plný nebo poloviční duplex, podpora DMA (Direct Memory Access), výstupy přerušení a ostatní. Výsledky práce zahrnují architekturu hardwarového návrhu USART, specifikaci návrhu, architekturu verifikačního prostředí, postup verifikace, výsledky syntézy a pokrytí.

KLÍČOVÁ SLOVA

Hodinové signály, Univerzální Synchronní a Asynchronní Přijímač-Vysílač (USART), Univerzální Asynchronní Přijímač-Vysílač (UART), funkční verifikace, UVM, UVMF, APB, sériová komunikace, vestavěné systémy, SystemVerilog

ROZŠÍŘENÝ ABSTRAKT

Tato práce prezentuje komplexní návrh, implementaci a verifikaci Univerzálního Synchronního a Asynchronního Přijímače-Vysílače (USART) kompatibilního s protokolem AMBA APB. Primárním cílem je vytvoření spolehlivého a efektivního komunikačního modulu, který umožňuje CPU komunikovat s externími zařízeními pomocí protokolů USART nebo UART. Funkční verifikace byla navržena s využitím objektově-orientovaného přístupu, moderní verifikační metodologie UVM, jejího rozšíření UVMF a s důrazem na metriky pokrytí pro odhalení skrytých chyb.

V moderní číslicové elektronice jsou trendem parametrizovatelné a softwarově konfigurovatelné návrhy. Navržený USART modul lze konfigurovat pro synchronní či asynchronní režim, což nabízí flexibilitu při integraci do systému. Parametrizace dovoluje provádět změny chování návrhu beze změny samotného kódu. Práce dále zkoumá optimalizaci spotřeby energie pomocí technik clock-gating a také jednovodičovou half-duplexní komunikaci pro konfigurace s minimálním počtem vodičů.

Univerzální Asynchronní Přijímač-Vysílač (UART) a jeho synchronní varianta USART je populární sériový protokol, který se využívá v aplikacích, kde vysoká propustnost není primárním požadavkem. Výhodou tohoto protokolu je jednoduchost, minimální počet vodičů potřebných pro komunikaci, nízká spotřeba a malá plocha návrhu. Implementace je vybavena mechanismy detekce chyb a odolnosti proti deviaci hodinových signálů během přijímání. Synchronní režim, využívající externí synchronizační hodinový signál, řeší problémy asynchronní komunikace, umožňuje přenos dat vyššími rychlostmi a zmenšuje pravděpodobnost vzniku chyby.

V rámci této práce byla vytvořena funkční specifikace návrhu s využitím metodologie pro psaní požadavků - EARS (The Easy Approach to Requirements Syntax). Také byla vytvořena registrová mapa, která popisuje registry uvnitř návrhu přístupné přes paralelní komunikační protokol APB (Advanced Peripheral Bus). Implementace vysílače a přijímače byla specifikována pomocí stavových automatů.

Práce se také zabývá univerzální verifikační metodologií UVM, vysvětlením jejích základních částí, mechanismů a popisem dalších nástrojů pro verifikaci, jako jsou SystemVerilog assertions. Návrh znovupoužitelného verifikačního prostředí byl proveden s ohledem na architekturu prostředí definovaného UVMF, což je klíčové pro důkladné otestování a odhalení skrytých chyb v návrhu. Výsledná část práce prezentuje architekturu hardwarového návrhu USART, jeho detailní specifikaci včetně návrhu banky registrů, vysílače a přijímače, a také registrovou mapu. Dále jsou uvedeny výsledky RTL syntézy, popis kontroly pomocí nástroju Lint a popis navrženého verifikačního prostředí založeného na UVMF, včetně inovativního přístupu k implementaci prediktoru. Prezentován je také verifikační postup a dosažené výsledky pokrytí kódu a funkčního pokrytí.

GUMENYUK, Artem. *Design and Verification of USART Peripheral*. Master's Thesis. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Microelectronics, 2025. Advised by doc. Ing. Jana Drbohlavová, Ph.D.

Author's Declaration

Author:	Bc. Artem Gumenyuk
Author's ID:	228619
Paper type:	Master's Thesis
Academic year:	2024/25
Topic:	Design and Verification of USART Peripheral

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I would like to sincerely thank my consultant, Ing. Miloš Juhás, for his invaluable guidance and expertise, which greatly contributed to the success on this project. I am also grateful to my advisor, doc. Ing. Jana Drbohlavová, Ph.D., for her mentorship and consulting support throughout this work.

Contents

List of Figures

List of Tables

Listings

Introduction

One of the most popular serial communication protocols in digital electronics is UART (Universal Asynchronous Receiver-Transmitter). This protocol is used in applications where high throughput is not the primary concern, valued for its simplicity and minimal number of wires. An evolution of this, the USART (Universal Synchronous and Asynchronous Receiver-Transmitter), introduces a synchronous mode of operation. Utilizing an external synchronization clock signal solves issues of asynchronous communication and allows data to be transferred at higher speeds. The USART device can be configured by the main device to operate in either synchronous or asynchronous mode, offering flexibility to system integration.

The integration of this peripheral into SoC (System on Chip) level relies on the standardized AMBA (Advanced Microcontroller Bus Architecture) APB (Advanced Peripheral Bus) bus protocol. This protocol is widely used for connecting low-bandwidth peripherals to the central processor.

This master's thesis covers the comprehensive design, implementation, and verification of a USART peripheral compatible with the AMBA APB protocol. The primary objective is to create a reliable and efficient communication module that lets the CPU communicate with external devices using either USART or UART protocols. Additionally, the thesis explores power optimization through clock-gating techniques. It also examines one-wire half-duplex communication in a configuration with a minimum number of wires.

Equally critical is the verification strategy: digital designs must be exercised to uncover corner-case bugs. Coverage metrics guide this process, quantifying how thoroughly functional requirements have been tested. The modern object-oriented methodologies UVM (Universal Verification Methodology) and UVMF (UVM Framework) are used in this thesis to create a reusable verification environment.

This thesis is structured into five main chapters:

1. **UART peripheral**: This chapter provides an overview of the UART and USART protocols, examines the components and techniques of UART controllers, and discusses their relevance in embedded systems.
2. **AMBA: Advanced Peripheral Bus**: Details the APB4 protocol, bus timing, and integration of memory-mapped peripherals.
3. **Universal Verification Methodology**: This chapter introduces the UVM, highlighting its principles and advantages in hardware verification. It outlines the structure of the UVM-based verification environment, the SystemVerilog assertions, and coverage in design verification.
4. **UVM Framework**: Explains the structure and benefits of the UVM Framework.

5. **Thesis Results**: The final chapter presents the results of this work, including the USART design architecture, design specification, FSM diagrams, register map, and the designed UVMF-based verification environment, as well as RTL synthesis metrics, coverage results, lint-check findings, and verification flow. The innovative approach to the predictor implementation is described in this section.

1 UART peripheral

UART is a widely used serial communication protocol. The main advantages of this protocol are simplicity, low power consumption, and minimal wiring required for communication. Frequent scenarios of using are control data sending, rarely reads and writes between transmitter and receiver [5].

1.1 UART serial protocol

The data frame shown in Fig.1.1 consists of a start bit, a number of data bits (usually 8 bits) starting from LSB, an optional parity bit, and 1, 1.5, or 2 stop bits.

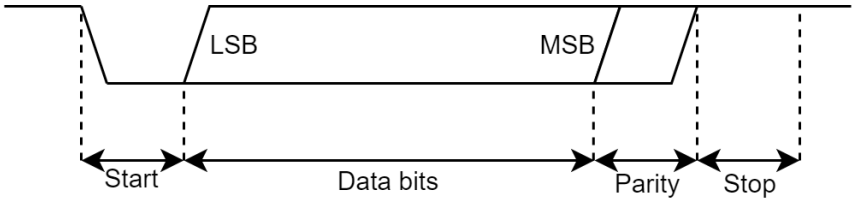


Fig. 1.1: UART frame format

Start and stop bits are used to define the beginning and end of the frame. Parity can be odd, even, sticky, or disabled. Its main objective is data protection. If odd parity is selected, a parity bit is '1' when a count of ones in data is odd; otherwise, parity is '0'. Even parity works similarly, but it's active for an even count of ones. When the receiver samples all data bits, it calculates parity for this data vector. If the parity bit in the frame differs from the calculated, data will not be accepted. In this case, the receiver can acknowledge to the transmitter or processor that the last received frame was invalid.

A peripheral component that will be implemented in this thesis has a configurable number of data bits, stop bits, and parity settings. The transmitter and receiver should be set at the same baud rate, that is, the number of bits transmitted or received per second. This is a mandatory condition for asynchronous communication between two devices. The designed peripheral component should support any of the most common baud rates up to 2MBd (2Mb/s).

1.2 UART controller

The main parts of UART controller are the transmitter, receiver, and baud rate generator, as shown in Fig. 1.2.

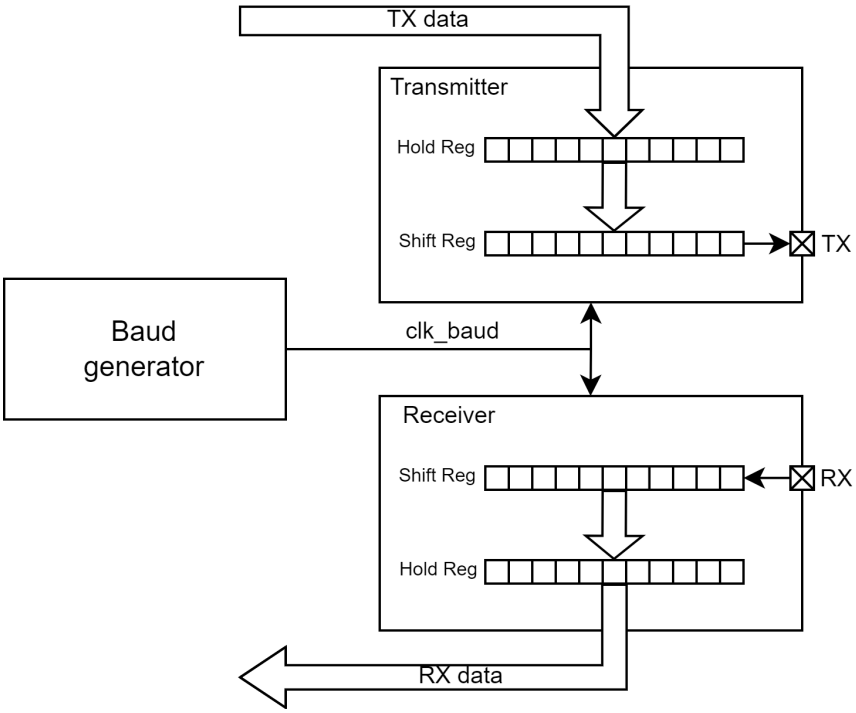


Fig. 1.2: UART controller block scheme (redrawn from [1])

Data to be transmitted are placed in the hold register or FIFO (First In First Out) register by the master component. When the transmitter is ready to start a new transfer, data from the hold register or FIFO are transferred to the shift register. Each bit of the UART frame is driven synchronously with the pulses from the baud rate generator [5].

The receiver waits for the start bit on the RX port. To recognize the start bit, the receiver samples at a higher frequency than the one set for UART communication. The clock signal from the baud rate generator often has a higher frequency than is needed for receiving and transmitting at a set baud rate.

1.2.1 Oversampling technique

Sampling at higher frequencies than baud rate frequency is called the oversampling technique. Usually, 8x and 16x oversampling are used, meaning the sampling clock should be 8 or 16 times faster than the communication frequency. The first step of this technique is to sample a first '0' bit at the RX line corresponding to the start bit. The auxiliary counter increments each sample after that. When the counter value equals 7 (in case of 16x oversampling) - the counter resets, and the receiver goes to the next state. If at least one of the samples from 0 to 7 was equal to '1', the receiver goes to the idle state. In this situation, a counter value of 7 means the last sample was obtained from the middle of the start bit. The counter

overflows after reaching the value of 15. The counter value of zero represents the start of the following data bit. Samples from the middle of the data bit, for example, samples 7, 8, and 9, are used to determine the value of the bit by the majority rule. The calculated value is moved to the shift register [6]. Oversampling by 8 allows to achieve a higher maximum baud rate (up to f_{CLK}[1]/8) but reduces receiver tolerance to timing deviations of data transfer [7]. Example of oversampling by 8 is shown in figure 1.3. Three middle samples are taken into account to calculate the result bit value. In this example, the first and second samples are evaluated as logic high and the third as logic low. Using the majority rule, the result value is calculated on logic high.

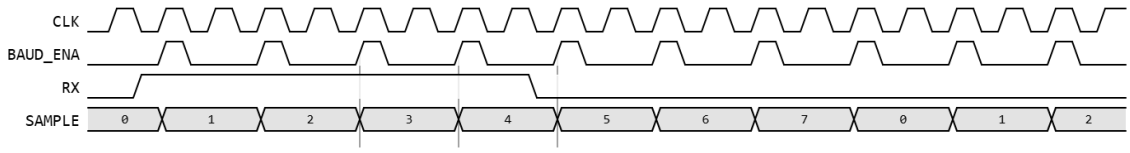


Fig. 1.3: Oversample input data by 8

1.2.2 Transmitter and receiver FIFO buffers

To minimize access frequency from the processor, UART controller can use FIFO buffers to hold data for the transmitter and from the receiver side. Both FIFOs can support the configurable trigger levels. When the number of data frames in the FIFO reaches the trigger level, UART peripheral can interrupt the main device. The main device will execute a subroutine that will read all data from the receive FIFO and process them or write new data to the transmitter FIFO.

1.2.3 UART communication modes

UART peripheral component can function in three duplex modes: full-duplex, half-duplex, and simplex. Full-duplex is a default two-wire communication, where data transmitting and receiving are possible in parallel. Half-duplex is a bi-directional single-wire communication, where only transmitting or receiving operation is possible simultaneously. The RX port is not used in half-duplex mode. In this configuration, the main component should select the transmit or receive mode by writing the appropriate control register. Simplex is a unidirectional single-wire communication, where only transmitting or only receiving is possible [7].

[1]CLK stands for clock

1.2.4 Communication errors

The following errors can occur during UART communication:

- Framing error. It occurs when UART hardware sees the logic '0' where the stop bit should be [8]. An example of an error UART frame with enabled parity bit and one stop bit set is shown in figure 1.4.
- Overrun error. Occurs when new data is received, but old data wasn't read from the hold register or FIFO buffer. After the overrun error occurs, the last data written to the data buffer will be re-written. The overrun error can occur in either the receiver or the transmitter.
- Parity error. It occurs when the calculated parity of newly received data isn't equal to the parity bit value in that frame.

Fig. 1.4: UART frame with framing error

1.2.5 UART interrupts

Interrupts are needed to notify the main component, usually the processor, that some predefined actions happened. Interrupt sources can be enabled or masked by writing to the interrupt enable register. All interrupt signals from the UART controller are combined to the single-bit interrupt signal using the OR function. The cause of the interruption can be found by reading the status register. The status register includes all possible interrupt source bits.

All communication errors described in the section 1.2.4 are sources of the interrupt. The interrupt can be generated if the transmitter's or receiver's FIFO buffer level is triggered. The transmitter's frame start and end can be a source of interruption. The receiver's frame end can also be a source of the interrupt if it's enabled.

1.2.6 Direct Memory Access support

DMA (Direct Memory Access) is a device that can transfer data from one address space to another directly, without the CPU (Central Processing Unit). This component allows to save CPU resources that would otherwise be used to move data. DMA can be configured to transfer data from a certain memory address space to the hold register or FIFO in the transmitter. Also, DMA can be used to transfer data from the receiver's hold register or FIFO to another memory space.

A peripheral device that supports DMA transfers can implement the hardware handshaking interface. This interface consists of a request signal from the periphery to the DMA controller and a clear signal from the DMA controller to the periphery. The request signal is used to ask a DMA controller to start the burst of data transfers. When the burst is finished, the DMA controller drives the "clear" signal.

1.2.7 Fractional baud rate generator

Two clocks must be generated—one for the transmitter and the other for the receiver. The clock generated for the transmitter shall have an accurate frequency and constant period. This will be implemented using a classic counter with a programmable overflow level. A signal pulse is generated when the counter reaches that level. The same counter generates another clock signal for the transmitter, but the waveform should have approximately 50% duty cycle. When synchronous mode is selected, this signal is asserted to the SCLK output pin.

The receiver uses the second clock signal output to sample input data on the RX port. The generated clock signal shall have 8 or 16 times higher frequency than the transmitter clock. This signal is used for oversampling, and it doesn't necessarily have to have a constant period, but the average frequency should be within the specified limits. If only the sample in the middle of the data bit and 10-bit frame is considered, a deviation of the receiver clock can be up to 5%. In the actual design, the deviation of the clock frequency of both sides of communication should be taken into account.

The receiver's clock frequency is a baud rate frequency multiplied by the oversampling factor. The main clock should be divided by a fractional number to generate a clock with such a frequency. A binary rate multiplier can be used for that purpose [9]. The main idea is to combine clock signals divided by powers of two, i.e., clk/2, clk/4, clk/8, etc., into the signal that will be used as a fractionally divided clock or to create such a clock. Creating this clock signal is shown in the figures 1.5 and 1.6. Signals from a0 to a3 are extracted from the divided clocks and selected using the division factor. These signals are combined into the out_ena signal using the OR operator. This enable signal is used to generate a divided clock clk_out using the clock gating technique.

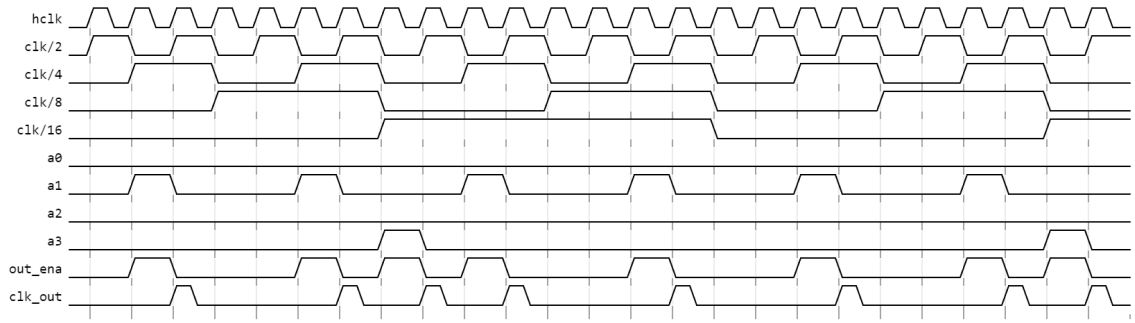


Fig. 1.5: Generation fractionally divided clock signal. Division factor: 5/16

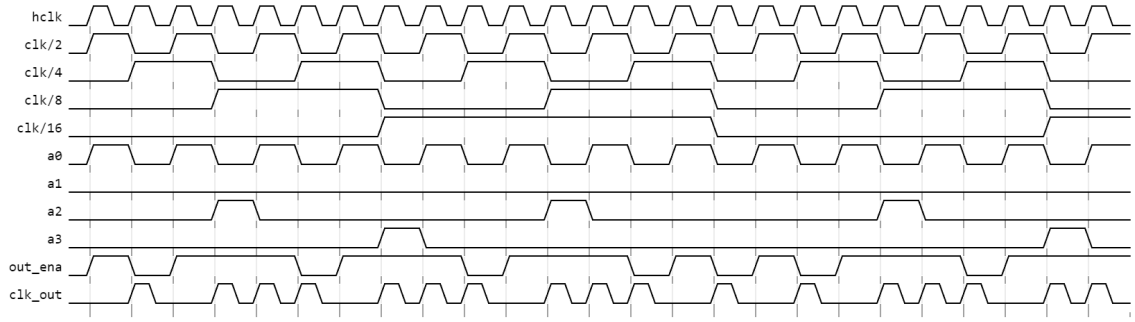


Fig. 1.6: Generation fractionally divided clock signal. Division factor: 11/16

1.3 USART: synchronous communication

The USART peripheral component can function asynchronous or synchronous. The SCLK pin should be connected to the other side device if synchronous mode is selected. SCLK is the clock signal from the baud rate generator, processed by the USART transmitter block. SCLK pulses every data bit starting from LSB to MSB (Most Significant Bit). No pulses are generated during start and stop bits. If receiving is enabled, the data should be sampled synchronously on the RX port. Reception is only possible concurrently with the transmission, similar to the SPI (Serial Peripheral Interface) communication protocol. Therefore, only full-duplex communication mode is available for synchronous mode. The full-duplex synchronous communication frame is shown in figure 1.7. If transmission is disabled, the clock is not generated, preventing reception. Because receiving is synchronous, no oversampling is needed, which means that a higher baud rate can be reached. Because of this, the transmitting device must adhere to the setup and hold time requirements [7].

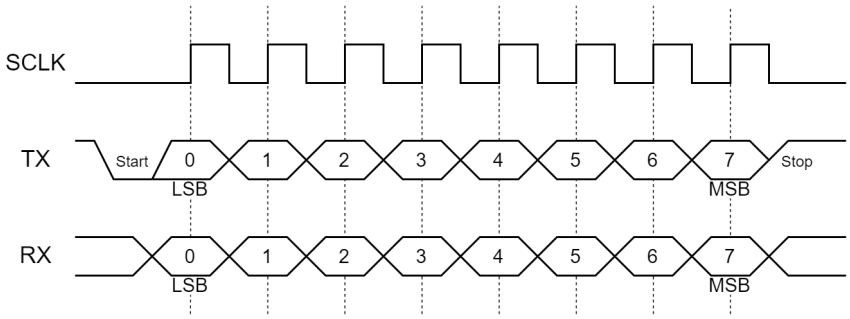


Fig. 1.7: USART synchronous communication frame. Data width = 8bit.

1.4 Clock gating technique

Clock gating is a technique for reducing dynamic power consumption. Up to 45% of the total power consumption in the digital design is associated with the clock tree. The main idea is to reduce the register toggle count. Fundamentally, it can be achieved in two ways: in the data path or in the clock path [10].

In the first option, the feedback loop is used when the data are not supposed to be updated in the register. When the multiplexer's selection signal selects the input data branch - these data will be loaded into the register on the next rising edge of the clock [10]. This method is suitable for the FPGA (Field Programmable Gate Array) because of the platform clock tree routing specialties. This method is shown in the figure 1.8.

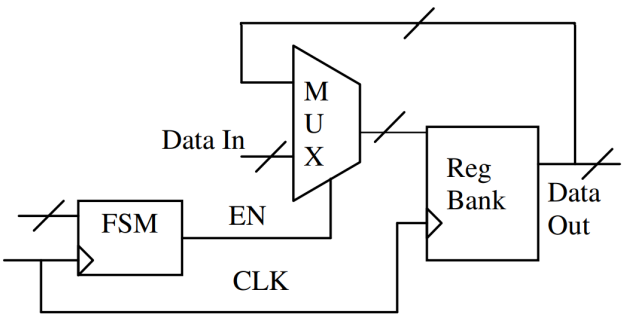


Fig. 1.8: Clock gating inserted into the data path. Reprinted with permission from [10].

The second option is to use latch-based clock gating cells. The load enable signal and the clock signal are the inputs of the clock gating cell. The output is a gated clock signal that is used instead of the standard clock of the selected registers. This method has the advantage of better power and area consumption [10]. This method

is suitable for the ASIC (Application Specific Integrated Circuit) and SoC (System on Chip) platforms. The method is shown in the figure 1.9.

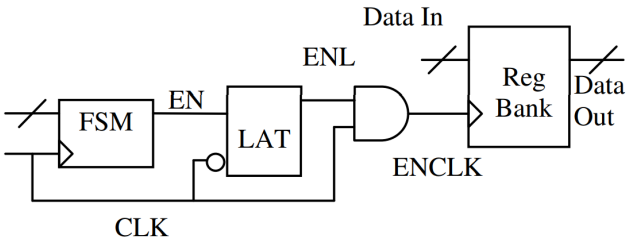


Fig. 1.9: Clock gating inserted into the clock path. Reprinted with permission from [10].

1.5 One-wire communication

One-wire or half-duplex communication allows both sides to receive and transmit, but not simultaneously. The analog part of the devices may include pull-up resistors and driving transistors, as shown in the figure 1.10. The pull-up resistors play a crucial role - preventing circuit shortcuts.

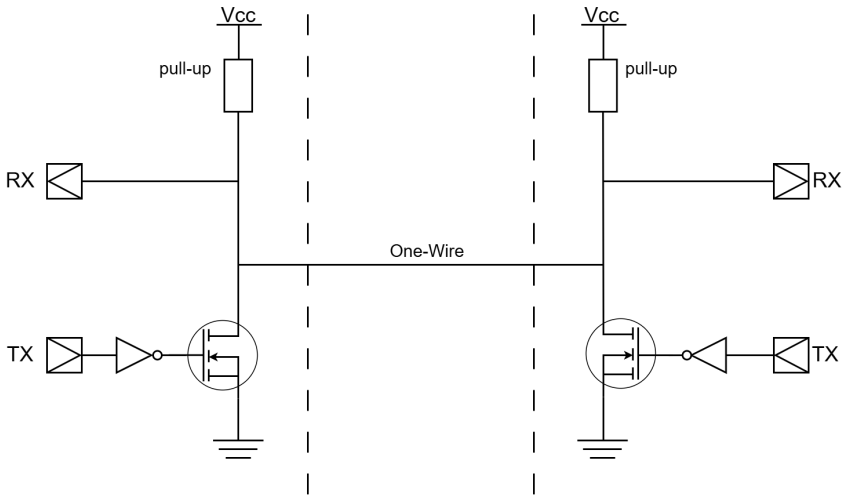


Fig. 1.10: One-wire USART connection. Reprinted with permission from [2].

When both transistors are closed, the voltage level on the wire corresponds to the logic high level. When at least one of the transistors is opened, the voltage level on the wire is pulled down to the logic low level. The main rule of such a communication mode is that only one side can transmit at a given time. In other words, only one of the transistors shall be opened at a time. Violation of this rule is called an overlapping error.

The transmitting device can detect an overlapping error. The receiver can function in the "overlapping check" mode. It shall compare the wire's actual value to the transmitter's unaffected value. The occurrence of this error indicates an issue in the frame ordering.

2 AMBA: Advanced Peripheral Bus

AMBA (Advanced Microcontroller Bus Architecture) APB is a low-power, parallel, not pipelined, synchronous bus protocol. APB is usually used in applications where a very high data transfer speed is not needed. The minimum transfer length is two clock cycles. Typical usage of the APB interface is accessing the peripheral devices registers. ARM processors use more complex communication protocols such as AXI or AHB as their main bus. Therefore, a bridge to the APB interface is needed [3].

2.1 Interface signals

To designate the sides of the communication interface, requester and completer keywords will be used. The requester is referred to as the central processor or the APB bridge. And APB peripherals are referred to as the completer [3]. An example of such a peripheral is the USART controller described in chapter 1. Certain signals listed in Table 2.1 may have parameterized widths, but the widths used in the implemented interface design will be specified explicitly.

Table 2.1: APB interface signals [3]

Signal	**Source**	**Width**	**Description**
PCLK	Clock generator	1	Clock. All APB signals are timed against the rising edge of PCLK.
PRESETNn	System bus reset	1	Reset. Active low APB domain system reset.
PADDR	Requester	32	Address. APB address bus.
PPROT	Requester	3	Protection type. Indicates the normal, privileged, or secure transfer. Specify data or instruction access.
PSELx	Requester	1	Select. Indicates that this completer is selected for transfer.
PENABLE	Requester	1	Enable. Indicates the second and subsequent transfer cycles.

PWRITE	Requester	1	Direction. Indicates read or write operation.
PWDATA	Requester	32	Write data.
PSTRB	Requester	4	Write strobe. Indicates which bytes should be updated during write operation.
PREADY	Completer	1	Ready. Used to extend data phase of APB transfer by completer.
PRDATA	Completer	32	Read data. Should be driven low when PWRITE is HIGH.
PSLVERR	Completer	1	Transfer error. Asserted HIGH by the completer to indicate an error condition on the bus.

2.2 APB transfers

There are two types of transfers: write and read. Each transfer has a setup phase and an access phase. The setup phase always lasts one clock cycle, and the access phase can last one or more clock cycles. The completer asserts the PREADY signal when it can accept the data at the next clock cycle. If data cannot be read immediately, the completer can insert wait states using the PREADY signal. Write transfer without wait states is shown in figure 2.1. Read transfer without wait states is shown in figure 2.2. APB bus enables changing only selected bytes for write operation using PSTRB signal. That could be useful when only some 32-bit register fields should be changed without performing read, edit, and write sequence [3].

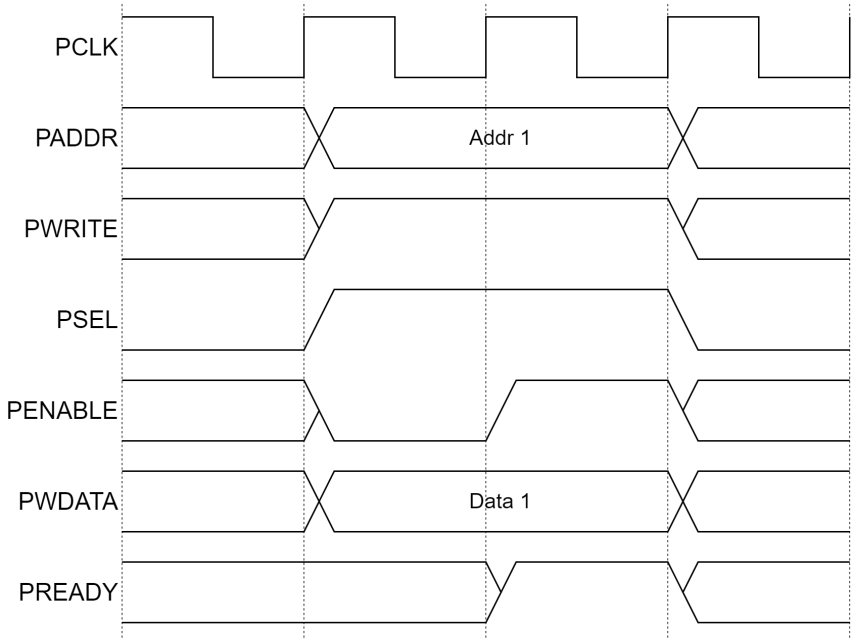


Fig. 2.1: APB write transfer [3]

When asserting PREADY high, the completer should assert a valid PRDATA signal. The completer commonly returns a zero data vector if the periphery is selected and the requester tries to read from a non-existing address.

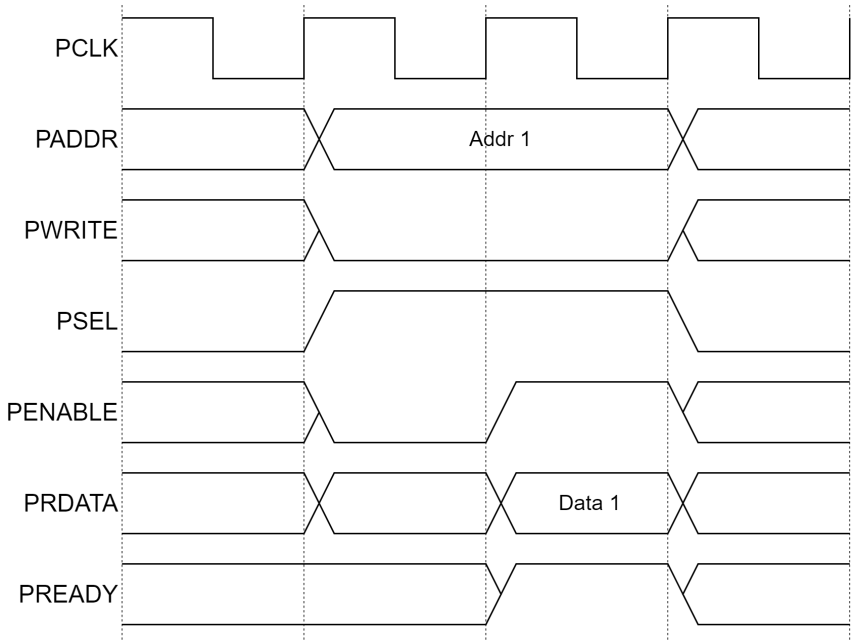


Fig. 2.2: APB read transfer [3]

3 Universal Verification Methodology

UVM is an object-oriented verification methodology released by the standards organization Accellera. This methodology uses the object-oriented capabilities of the SystemVerilog hardware description and verification language. The following sections describe these capabilities and the UVM parts required to create a verification environment. A basic understanding of concepts of OOP (Object-Oriented Programming) is expected.

3.1 OOP in SystemVerilog

Class is a SystemVerilog type that includes properties (data variables) and methods (functions and tasks). Object of class can be created using constructor function "new()". Classes don't have destructors because SystemVerilog has an automatic garbage collector that deallocates the object memory [11].

The inheritance OOP principle in SystemVerilog is implemented by the class extension mechanism using the "extends" keyword and "virtual" method qualifier. Methods declared as virtual can be rewritten in any of the subclasses. A subclass is a class that extends some existing class. An extended class called base class from the perspective of the subclass [11].

Polymorphism is also implemented by class extensions and virtual methods. Any class can be replaced by another class if they have the same base class and implement the same methods. That enables using the base class as some method argument type, with the possibility of calling this method and passing any of the subclasses objects as argument [11].

The abstraction principle is represented by class item qualifiers: protected and local. Unqualified class members are public and available to access through an object handle. Protected and local class members are only accessible inside the class. A protected member differs from a local member in that it can be inherited by any subclass [11].

In conclusion, SystemVerilog has almost all the OOP features of other object-oriented languages, such as Java or C++. UVM is based on the principles and possibilities described above.

3.2 UVM Structure

UVM itself is primarily a set of classes. These classes are used to build a UVM-based verification testbench. UVM base classes are organized into a hierarchy. The basic hierarchy of UVM base classes is shown in the figure 3.1.

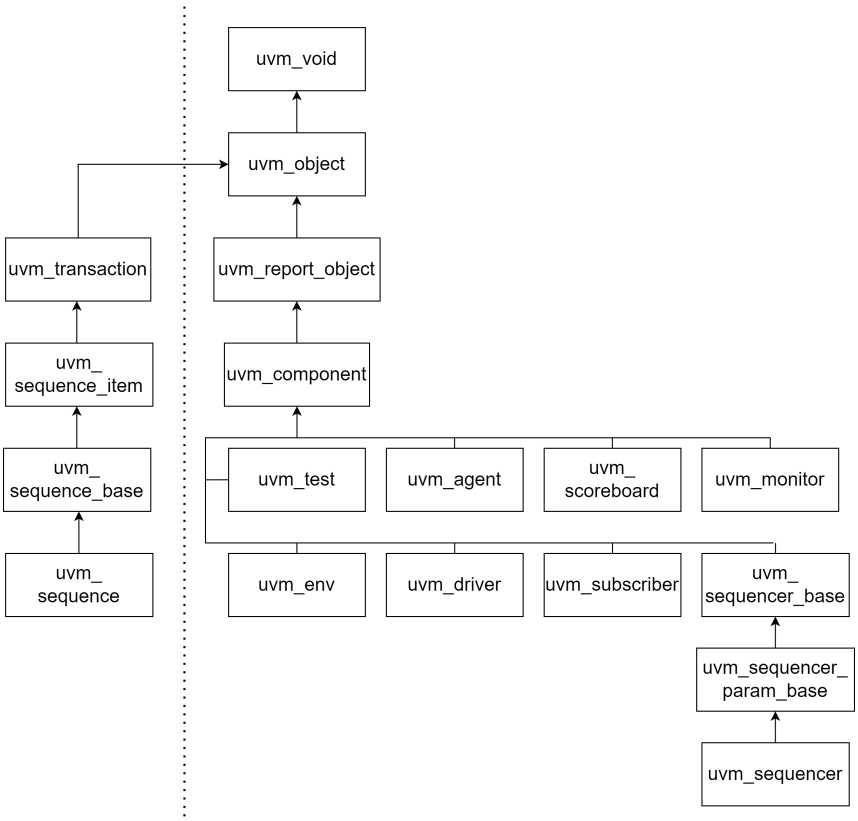


Fig. 3.1: UVM base classes diagram

All base classes can be divided into two parts: object-based and component-based. All "uvm_component" derived classes are static components that exist throughout the simulation's lifetime. Object-based classes can have reduced simulation lifetime; an example of such a class is "uvm_sequence" and its derived classes. Sequences are constructed manually and destructed automatically when the sequence ends.

To construct a UVM verification environment, the user shall derive his classes from UVM base classes. In addition to the classes shown in fig. 3.1, there are other base classes for register modeling, TLM (Transaction Level Modeling), DPI-C support, and other parts of the methodology.

3.3 UVM Components

Many UVM components are used to build a verification environment. This section will explain all the necessary components to create a basic UVM testbench.

3.3.1 UVM transaction

The transaction is a user-defined class that extends the "uvm_sequence_item" class. Transactions always represent signal interfaces that need to be driven or monitored. An example of the simplified USART transaction is shown in the listing 3.1. This transaction has random property declared on line 7 and non-random on lines 9 - 12. The agent uses the first group to drive the USART frame specified by their values. The second group transfers the data from monitoring the signal interface to analyzing components.

Data property members can be divided into the following types of information [4]:

- Control - a type of transfer (Read/Write), size.
- Payload - main data content.
- Configuration - error injection, alternative mode of operation.
- Analysis - fields that aid analysis.

Random properties usually should be constrained to be randomized on valid values. Constraint is declared on line 21.

Listing 3.1: User-defined UVM transaction example

```
1  class usart_transaction extends uvm_sequence_item;
2
3    `uvm_object_utils(usart_transaction)
4
5    // Group: Properties
6    // random
7    rand int unsigned tx_data;
8    // non-random
9    int unsigned rx_data;
10   bit           parity_err;
11   bit           framing_err;
12   bit           line_break;
13   // error injection
14   bit       do_line_break      = 0;
15   bit       par_err_inj        = 0;
16   bit       frm_err_inj_stop_0 = 0;
17   bit       frm_err_inj_stop_1 = 0;
18
19   // Group: Constraints
20   // max data width is c_DATA_MAX_WIDTH
21   constraint data_width_c {tx_data < (2**c_DATA_MAX_WIDTH);}
```

3.3.2 UVM sequence

Sequences are used to create test scenarios. User-defined sequences should be derived from the "uvm_sequence" class. This base class adds a new virtual method "body()" that should be rewritten to create a time-consuming test scenario. Transactions and other sequences can be created, randomized, and started within the body method of the sequence. The advantage of the sequence approach is the possibility of sequence layering. The top sequence associated with some test can call any of the testbench sequences. Sequences are divided into different levels: test, top environment, environment, and agent. The disadvantage of the sequence approach is the inability to directly access the testbench resources such as configuration, register model handle, and other agents and environments handles [4]. An example of the "body()" method of the USART sequence is shown in the listing 3.2. The transaction described in section 3.3.1 is created on line 5, started on line 6, randomized on line 8, and finished on line 11 of the shown method. Randomization error handling is on line 9.

Listing 3.2: User-defined UVM sequence example

```
1  virtual task body();
2      bit ok;
3
4      usart_transaction usart_txn;
5      usart_txn = usart_transaction::type_id::create("usart_txn");
6      start_item(usart_txn);
7
8      ok = usart_txn.randomize();
9      if(!ok) `uvm_fatal("USART_SEQ", "Randomization of usart_txn failed")
10
11     finish_item(usart_txn);
12 endtask
```

3.3.3 Agent

The agent is a component-based part of a testbench. It is the lowest component in the hierarchy of the testbench structure. Each agent is associated with some logical interface, such as APB, USART, or another [4]. It should include these verification components:

- Driver - converts transactions into pin-level activity, may receive a response from the DUT (Design Under Test) and convert it back on the transaction level.

- Sequencer - arbitrates sequence items between driver and sequences.
- Monitor - converts interface pin-level activity to transaction level.
- Agent configuration - defines what agent components will be constructed, hold the monitor and driver BFMs (Bus Functional Model) handles, and defines the agent behavior [4].

Other components that can be present in the agent are analysis. These components are connected to the monitor output to perform comparisons or collection procedures. Part of the simplified USART agent is shown in the listing 3.3. The user-defined agent is derived from the UVM agent base class.

Listing 3.3: User-defined UVM agent example

```
1  class usart_agent extends uvm_agent;
2    `uvm_component_utils(usart_agent)
3
4    typedef uvm_sequencer #(usart_transaction) usart_sqcr_t;
5
6    uvm_analysis_port #(usart_transaction) m_usart_ap;
7
8    usart_sqcr_t   m_usart_sequencer;
9    usart_driver   m_usart_drv;
10   usart_monitor  m_usart_mon;
11
12   function void build_phase(uvm_phase phase);
13     super.build_phase(phase);
14
15     m_usart_ap        = new("m_usart_ap");
16     m_usart_sequencer = usart_sqcr_t::type_id::create("m_usart_sequencer");
17     m_usart_drv       = usart_driver::type_id::create("m_usart_drv");
18     m_usart_mon       = usart_monitor::type_id::create("m_usart_mon");
19   endfunction
20
21   function void connect_phase(uvm_phase phase);
22     super.connect_phase(phase);
23     m_usart_drv.seq_item_port.connect(m_usart_sequencer.seq_item_export);
24     m_usart_mon.mon_ap.connect(m_usart_ap);
25   endfunction
26 endclass
```

Utilization macros are recommended for each object-based or component-based part of the UVM testbench. Using this macro, the "type_id" type is defined and allows the use of a "create()" static function. This function registers the class in the

UVM factory and makes it possible to override any component in the testbench if it is created this way [12].

The analysis port declared on line 6 is a component that works as an output port to communicate with other agents, environments, or analysis components on the transaction level. This port is connected to the monitor output port on line 24. Pin-level activity is sampled and converted on transactions by the monitor component. These transactions are distributed to the subscriber components for analysis. The sequencer and the driver are connected on line 23.

3.3.4 Environment

The environment is a top component for agents, predictors, scoreboards, function coverage collectors, and others. An environment can have sub-environments instantiated into it. It's highly recommended that reusable environments be designed on a per-protocol basis. This ensures that the environment is independent of other project-specific parts [13]. The environment that is, for example, created to verify conversion between AHB and APB buses can be instantiated in another project environment that has this bridge in the RTL (Register-Transfer Level) design.

The environment can have a monitor unrelated to any agent to perform transaction checking and coverage collection on the environment level. The primary function of the environment is to model behavior depending on the environment configuration, monitor DUT activity, check the correctness of the protocols activity, and collect coverage [13].

An example of the user-defined environment is shown in the listing 3.4. This environment includes the USART agent, APB agent, and USART analysis components: predictor, coverage collector, and the scoreboard. These classes have their handles declared on lines 4 - 8. Objects of these classes are created into the "build_phase()" method using the "create()" static function on lines 16 - 20. Finally, environment components are connected into "connect_phase()" method on lines 25 - 29. The analysis ports mechanism provides a connection.

Analysis ports send transactions to all connected analysis exports using the "write()" method. This method is not time-consuming; it means that transactions are sent to all subscribers in the same delta cycle [13].

Listing 3.4: User-defined UVM environment example

```
class usart_environment extends uvm_env;
  `uvm_component_utils(usart_environment)

  usart_agent         m_usart_agt;
  apb_agent           m_apb_agt;
  usart_predictor     m_usart_pred;
  usart_cov_collector m_usart_cov;
  usart_scoreboard    m_usart_sb;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    m_usart_agt  = usart_agent::type_id::create("m_usart_agt");
    m_apb_agt    = apb_agent::type_id::create("m_apb_agt");
    m_usart_pred = usart_predictor::type_id::create("m_usart_pred");
    m_usart_cov  = usart_cov_collector::type_id::create("m_usart_cov");
    m_usart_sb   = usart_scoreboard::type_id::create("m_usart_sb");
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    m_usart_agt.m_usart_ap.connect(m_usart_pred.usart_ana_exp);
    m_apb_agt.m_apb_ap.connect(m_usart_pred.apb_ana_exp);
    m_usart_agt.m_usart_ap.connect(m_usart_cov.ana_exp);
    m_usart_pred.exp_ap.connect(m_usart_sb.exp_ana_export);
    m_usart_agt.m_usart_ap.connect(m_usart_sb.actual_ana_export);
  endfunction: connect_phase
endclass
```

3.3.5 Test

The test class is a top component for environments, configuration objects, and other components. The user-defined test class is derived from the uvm_test base class. It provides the ability to select the test to execute using the UVM_TESTNAME command-line argument and allows you to use the UVM phasing mechanism, which will be described in section 3.4 [13].

In practice, a user-defined base test class is created to instantiate the top-level

environment, configure the properties of configuration objects, and establish connections between BFMs and agents. Specific user-defined tests are built upon this base class, calling the respective test sequences. Tests tailored to particular scenarios can modify the configurations set in the base test class.

3.3.6 Configuration

One of the most essential things to create reusable verification components is to make them as configurable as possible. For this purpose, configuration objects are one of the best choices [13].

Configuration objects are user-defined classes typically associated with a test, environment, or agent. Configuration properties can be divided into standard and specific.

Examples of the standard configuration parameters are:

- Active or passive mode. In active mode, the agent drives stimuli to the DUT. In passive mode, the agent samples bus activity, collects coverage, and makes checks.
- Enable/disable checks and coverage collection.

Examples of the specific configuration parameters can be:

- Number of bits in the frame.
- Bus communication speed.
- Enable/disable error injection.

Listing 3.5: User-defined configuration object example

```
1  class usart_config extends uvm_object;
2    `uvm_object_utils(usart_config)
3
4    // non-random properties
5    int unsigned clk_freq  = 8_000_000;
6    int unsigned frame_len = 10;
7    // random properties
8    rand int unsigned baud_rate = 115200;
9    // other
10   virtual usart_mon_bfm_t usart_mon_bfm; // USART virtual monitor BFM
11   virtual usart_drv_bfm_t usart_drv_bfm; // USART virtual driver BFM
12   usart_reg_model rm;                    // Register model handle
13   string usart_agt_activity = "ACTIVE";  // USART agent activity
14
15   ...
```

Methods such as "wait_for_number_of_clocks(int clocks)", which provides an interface between test parts and BFMs, can be defined in the configuration object. The top-level environment configuration object is a good place for this because it is accessible to all components under the top-level environment. Some configuration object properties may be defined as random to allow configuration randomization [4, 13].

UVM configuration database API is used to make the verification component independent from another part of the testbench. BFMs handles are set to the configuration database from the HDL (DUT) part of the testbench and get from the verification component environment configuration object [4]. An example of the part of the user-defined configuration object is shown in the listing 3.5.

Variables inside of the agent's configuration objects can be changed directly. But instead of the direct change, they can be set from the parent environment using the "configure()" method. This is especially useful when the environment instantiates more agents with the same agent's configuration properties. An example is a USART verification IP environment. It has two agents: a TX agent and an RX agent. These agents have shared configuration fields, such as baud rate, data length, number of stop bits, etc.

3.3.7 UVM Register Abstraction Layer

UVM provides a number of base classes, which are used to create an object-oriented model for memory-mapped registers and memories on the abstraction register layer. These base classes should be extended to abstract the read/write operations to registers and memories in the DUT [13]. The register model consists of these layers:

- Field. Represents a contiguous set of bits inside of the register.
- Register. Collection of register fields.
- Memory. Represents a block of memory of a specified size.
- Block. Collection of registers and memories.
- Map. Locates the offset address of registers, memories, and blocks.

Each layer has a configuration method. The field has size, LSB position offset, and access type properties. The register has a string-type property that contains an HDL path to the register [4].

Registers in the register model have two access types: front-door and back-door. Back-door access uses an HDL path specified during register configuration to perform read/write operations directly on the hardware register. A Back-door read is used to monitor or check the register's value. Since back-door access doesn't generate any stimuli on the bus, it consumes no simulation time.

Front-door type operations use a physical protocol (for example, APB) to access the register. For that, "bus to reg" and "reg to bus" adapters shall be created as part of the bus agent. These adapters convert generic register transactions into protocol-specific transactions and vice versa [4].

The UVM register model library provides several built-in register and memory test sequences. These sequences are used for faster testing, code coverage and functional coverage increase. Examples of such sequences are the reset value check sequence, accessibility check sequence, walking one sequence, and back-door paths check.

In reality, the register model stores four values for each field:

- Desired. Used for holding the required value. Allows to write multiple fields and registers values by calling a single "update()" method.
- Mirrored. Used for holding the expected value. Updated by prediction.
- Reset. Used for holding the reset value. Set into the register model.
- Value. Used for randomization constraints.

Value is the only public property that can be accessed directly. Interaction with other properties is possible using these methods:

- Write and read. Front-door or back-door operations method. Respect the access policies.
- Poke and peek. Back-door access methods that ignore access policies.
- Predict. Used for explicit prediction of field or register value. Update mirrored value.
- Set and get. Does not affect the DUT. Updates desired field or register value.
- Randomize. Set a random value to the field or register. Updates desired field or register value. Constraints can be applied.
- Update. Call write or poke method if desired and mirrored values differ.
- Mirror. Reads field or register value from the DUT and updates mirrored value. Allows checking by comparing actual and mirrored values.

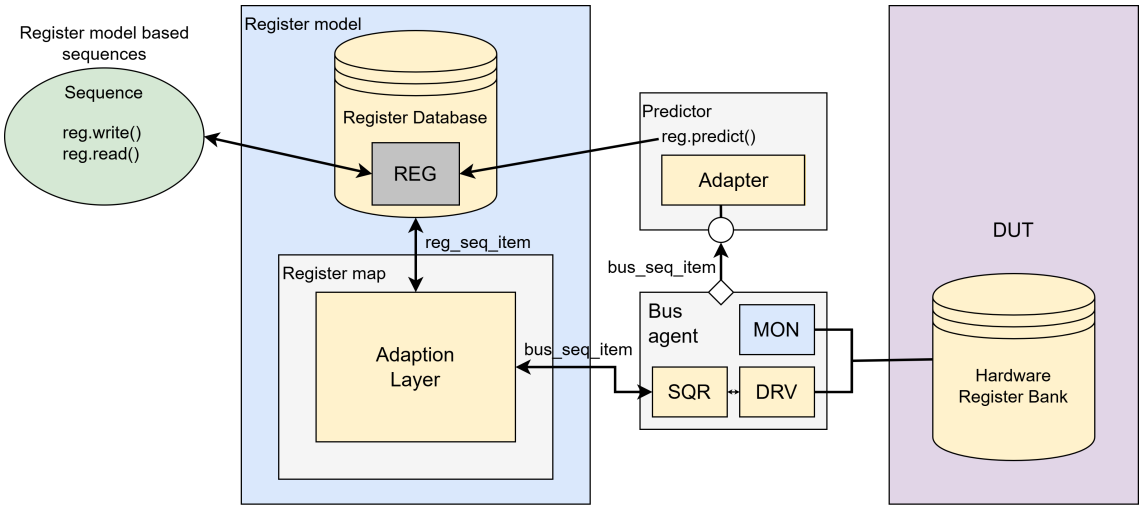


Fig. 3.2: Register model functionality block scheme (redrawn from [4])

A register model with an explicit prediction block scheme is shown in figure 3.2. Sequences can use the register model to perform read and write operations on the hardware registers. The register model will generate a register sequence item that will be converted to the bus-specific sequence item. This sequence item will be sent to the driver, who will convert it to the signal-level operation on the bus. This activity is monitored and converted on the transaction level by the monitor. The monitored sequence item will be sent to the predictor component, which will predict the value of the register. The predicted value will be written to the "mirrored" field of the register in the register model.

Register model generators are used to generate register models. A register model can consist of hundreds or thousands of registers with many small details and configurations, making it inefficient to create manually [13].

3.4 Phasing

The main idea of the UVM phasing mechanism is to divide the simulation flow into a certain number of steps and execute them consistently simultaneously for every test component. Components can be developed separately without explicit synchronization with other parts of the testbench. Phasing starts when the "run_test()" method is called from the static part of the testbench, often from the "HDL_TOP" [4].

Phases can be divided into three main categories: build, run, and cleanup. All UVM phases are shown in the figure 3.3. The "build_phase" and "connect_phases" were used in user-defined components described in sections 3.3.3 and 3.3.4. The

build and cleanup phases are not time-consuming; run phases are the only time-consuming phases within the simulation.

The purpose of the build phases is to construct, configure, and connect components. The run phase is a task that runs on all components in parallel, and it is used for stimulus generation. Reset, configure, main, and shutdown phases run parallel to the run phase. They should be called only from the test or the environment. Cleanup phases are used to extract metrics, check results, report them, and finish all actions in the simulation [4].

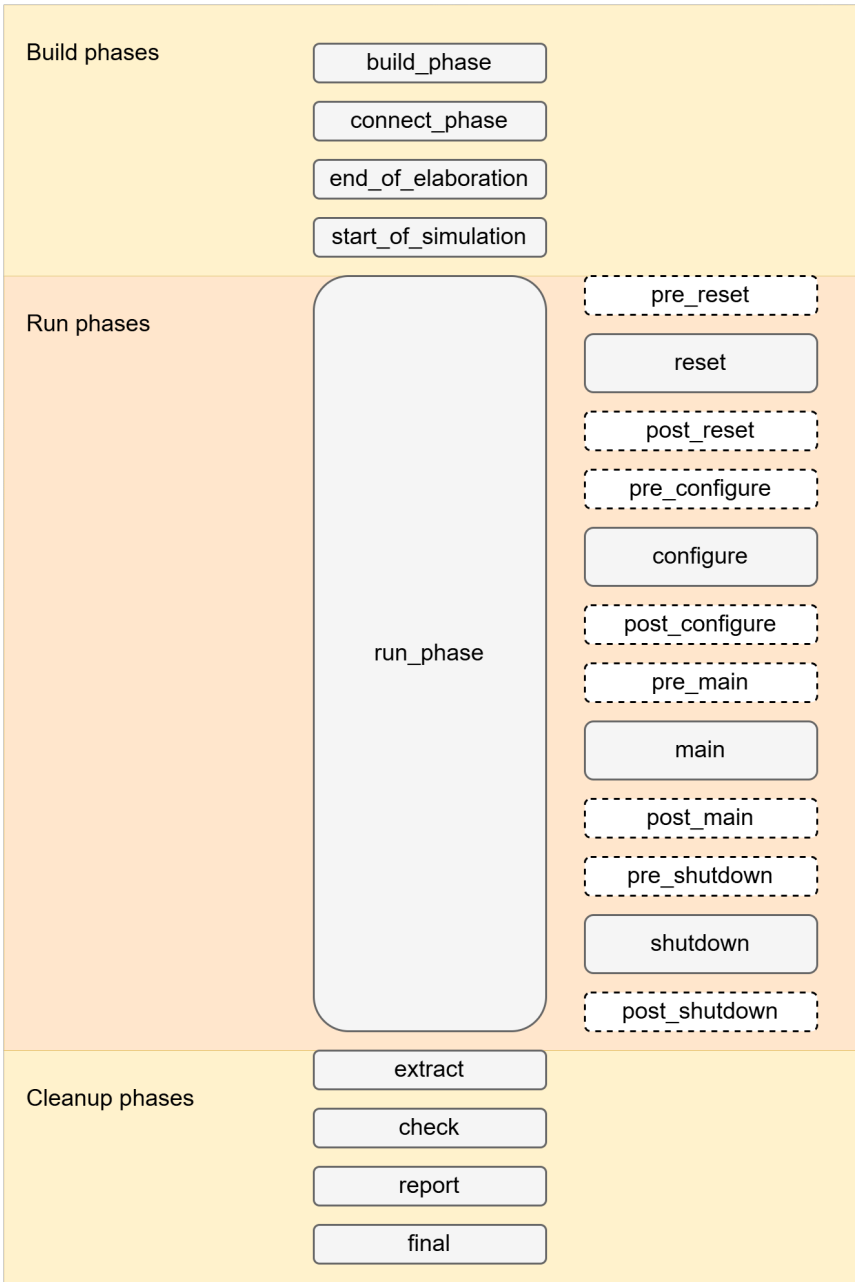


Fig. 3.3: UVM phases diagram (redrawn from [4])

3.5 UVM Testbench

A UVM testbench combines static components (modules and interfaces) with dynamic classes. Methodology defines a hierarchy as shown in the figure 3.4. DUT interacts with BFMs on a signal level in the static part of the testbench. BFMs convert signal-level activity on the transaction level (dynamic) and vice versa. Driver and monitor proxy components as parts of an agent communicate with other dynamic components of the testbench on the transaction level (using transaction objects).

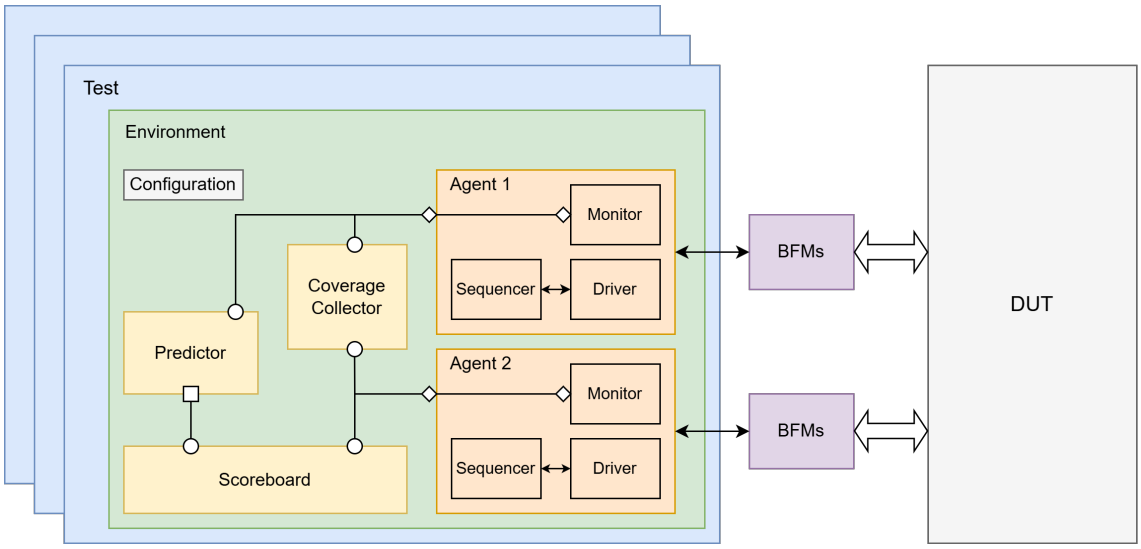


Fig. 3.4: UVM block-level testbench (redrawn from [4])

UVM-based verification is beneficial because it can be reused in the context of system-level integration. Well-designed verification components can be used on the higher-level testbench without changing code. The environment and its content sources should be copied into another project. Then, the copied environment should be instantiated within the system-level environment. The configuration of the block-level environment can be changed from the system-level test. Because of the extended DUT, active agents of the integrated environment are often switched to the passive mode.

These parts can often be reused after integration: environments, agents, analysis components, coverage collectors, configuration objects, register model, environment and agent level sequences, and assertions. Tests and test-level sequences are not reusable at the system level.

Dual-top architecture is one of the options for organizing a verification environment. The essence of this method is to separate everything associated with RTL and cycle-based signal activity from the object-based testbench. This architecture is commonly used in the industry, and Siemens's UVM framework is built on it.

3.6 SystemVerilog Assertions

Assertions are verification instruments used to validate the design's behavior on the signal level. They can also be used for formal verification and to provide functional coverage. During simulation, assertions check to see if a specific condition or sequence of events occurs. If the condition fails or a sequence is not completed correctly, an assertion may generate an error if required. In the context of the UVM-based verification, assertions are entirely separated from the object-oriented part of the testbench. Assertions are used to verify RTL design functionality, which is complicated to verify on the transaction level [11].

There are two types of assertions: immediate and concurrent. An immediate assertion is a simple check executed like a statement in a procedural block. Designers commonly use this type directly in the RTL code to verify their assumptions are correct. A concurrent assertion is a cycle-based verification instrument that spans over time and always does the following steps [11]:

1. Check if a condition or a sequence of conditions is true. After that, an assertion will activate.
2. When assertion is activated, a state of signal, condition, or sequence of conditions is checked.
3. Some action can be done if the assertion passed or failed (for example, print an error message).

Examples of the immediate and concurrent assertions are shown in the listing 3.6. A waveform, that clearly shows workflow of the concurrent assertion from this example is on the figure 3.5.

Listing 3.6: Immediate and concurrent assertion example

```
// immediate assertion
always @(posedge clock) begin
  assert !(write_ena && read_ena);
end

// concurrent assertion
req_sets_busy_a : assert property (
  @(posedge clock) disable iff(reset)
  REQ |-> !BUSY ##1 BUSY
);
```

If this concurrent assertion is applied to check REQ and BUSY signals stimuli from the figure 3.5, the following steps will be done:

1. Assertion is activated when the sampled value of REQ is 1. In the same cycle, assertion, check if the sample value of BUSY is 0.
2. Assertion check if the sampled value of BUSY is 1 in the next cycle. Assertion passed.
3. Assertion is activated again when the sampled value of REQ is 1. The assertion failed because the sampled value of BUSY was not 0.

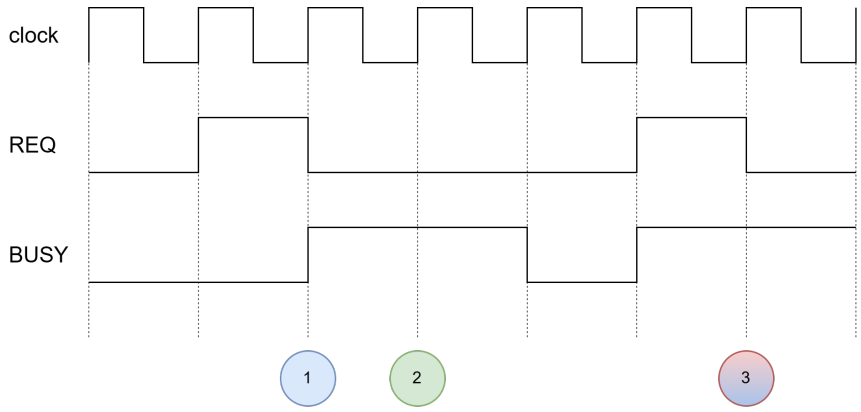


Fig. 3.5: Waveform for concurrent assertion example

There are a lot of system functions and syntax constructions used in SVA (SystemVerilog Assertions). Assertions are very powerful verification instruments. To make assertions reusable - properties are used. An example of a declared property is in the listing 3.7. The benefit of writing properties instead of the approach shown in the example 3.6 is the possibility to use them more times in the different assertions. A clocking source and disable criteria can be defined externally when the property is called to make it more reusable.

Listing 3.7: Property declaration example

```
1 property SIGNAL_CAN_INCREMENT_IF_COND_ACTIVE(COND, SIGNAL);
2   (SIGNAL == ($past(SIGNAL) + 1)) |-> COND;
3 endproperty
```

This property checks that when a signal is passed as argument "SIGNAL" increments, the signal passed as argument "COND" is high. A system function "$past," used in this property, returns a past sampled value of its argument.

An interface is a convenient way to integrate assertions into the verification environment. The interface can contain variables, models, properties, and assertions. Next, it should be bound to the RTL module.

3.7 Coverage

Coverage is the metric that is used during the verification process. This metric includes measuring the exercised RTL code and verifying the design features identified in the test plan. Additionally, coverage helps measure the verification progress and completeness. Coverage metrics show what design parts were activated during the simulation process. Using this information, the input stimulus can be adjusted to improve the verification completeness. Coverage can be classified into two kinds: code coverage and functional coverage [14].

3.7.1 Code coverage

Code coverage is an implicit kind of coverage with the implementation origin of the source. This type of coverage is automatically extracted from the RTL code. One of the advantages of code coverage is that it describes the degree to which the source code has been activated during testing. The 100% code coverage doesn't guarantee that the design is bug-free. The main limitation of this type of coverage is that the order of code execution does not matter for the code coverage [14].

Another limitation of the code coverage is that it doesn't indicate exactly what functionality defined in the specification was tested. If code coverage collection is enabled, selected metrics will be saved in the unified format to the UCDB database during the behavioral simulation. Code coverage results can be analyzed, and some coverage excludes can be applied. Coverage excludes are convenient when some code cannot be covered but should be present in the RTL code.

3.7.2 Functional coverage

Functional coverage is an explicit kind of coverage with the implementation and specification origins of the source. This type of coverage shall be manually defined and implemented by the verification engineer. Each functional requirement shall have a defined relevant functional coverage. The 100% functional coverage result means that all specified functionality were tested. Functional coverage provides automatic requirements tracing during simulation. A disadvantage of the functional coverage is that it shall be implemented manually [14].

Unfortunately, 100% code and functional coverage itself doesn't guarantee the correctness of the design behavior. There are some reasons for that:

1. Functional specification incompleteness. Edge case behavior isn't specified, which means that functional coverage won't be probably defined.
2. Mistakes in functional coverage implementation.
3. Coverage measures the completeness of stimuli but doesn't check the correctness of the behavior.

4 UVM Framework

UVMF (Universal Verification Methodology Framework) is a verification framework based on the Universal Verification Methodology. The UVMF provides a standardized and structured approach to creating reusable verification components and testbenches. It includes a set of base classes derived from the UVM base classes, a testbench code generator, verification components, simulation scripts, usage examples, debug features, guidelines, and documentation [15].

There are benefits of UVMF:

- Standardized testbench. It promotes the compatibility, readability, and reusability of verification components created by different teams and companies. Standardization allows the automatic generation of the typical parts of the verification testbench, reducing development time and effort.
- Hiding the details. Starting with UVMF, instead of the UVM, is easier and faster because many implementation details are hidden.
- Scalability. Verification environments can be easily scaled from block-level to system-level using a code generator and abstract description of the environment in YAML description language. A code generator takes an integrated environment or agent as a black box with IO ports at signal or transaction levels.
- Debugging and error tracing. The UVMF includes many debug features and transaction-level tracing that help solve verification errors [15].

The UVMF python scripts can generate a testbench from YAML files. There are three types of YAML templates to complete: interface (agent-associated), environment, and bench. An example of the USART interface description is shown in the listing 4.1. The SystemVerilog/UVM code is generated from provided YAML files, which can be a good starting point in creating a verification environment [15].

This USART interface example shows the declaration of the input and output signal ports on lines 13-28 and the agent transaction class on lines 30-37. Additional transaction variable random constraints can be declared. Declared parameters and type definitions will be distributed throughout the generated agent. Transaction-level connections are generated implicitly. The user shall define the driver and monitor BFMs' behavior manually.

Listing 4.1: USART interface description in YAML example

```yaml
uvmf:
  interfaces:
    "usart":
      existing_library_component: "False"
      clock: "clock"
      reset: "reset"
      reset_assertion_level: "False"
      parameters:
        - name: "DATA_WIDTH"
          type: "int"
          value: "8"
      ports:
        - name: "tx"
          width: "1"
          dir: "output"
          reset_value: "1'b0"
        - name: "wdata"
          width: "DATA_WIDTH"
          dir: "input"
          reset_value: "'bz"
        - name: "rx"
          width: "1"
          dir: "input"
          reset_value: "1'b0"
        - name: "rdata"
          width: "DATA_WIDTH"
          dir: "output"
          reset_value: "'bz"
      transaction_vars:
        - name: "data"
          type: "bit [DATA_WIDTH-1:0]"
          isrand: "True"
          iscompare: "True"
        - name: "rw"
          type: "bit"
          isrand: "False"
          iscompare: "False"
```

5 Thesis Results

This chapter presents the key findings of the study. The results are grouped into three sections: hardware design architecture, verification environment architecture, and hardware design specification. All findings are visualized through tables and figures.

5.1 Hardware Design Architecture

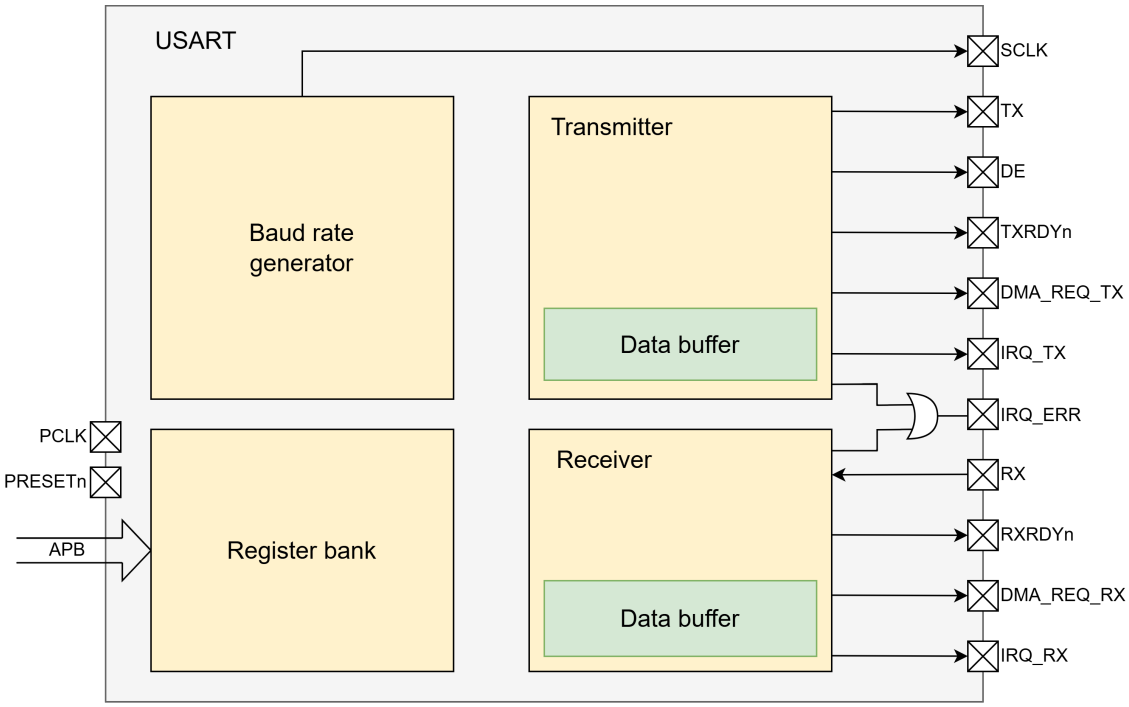


Fig. 5.1: USART hardware design architecture

The USART peripheral device architecture includes parts described in the chapter 1. The "PCLK" is the main clock in this design, and "PRESETn" is the main asynchronous reset.

The main device communicates with USART through the APB interface described in the chapter 2. Data should be stored in registers inside of the register bank module. Registers with RO (read-only) and RW (read-write) access rights should be accessible to read via the APB. Registers with WO (write-only) and RW (read-write) access rights should be accessible to write via the APB. Registers are accessible to other modules in the design.

To configure the baud rate generator, the "br_tx_cfg.div" and "br_rx_cfg.rate" registers shall be set to the corresponding values according to the equations 5.1 and 5.2.

$$div = \frac{f_{clk}}{baud\_rate}, \tag{5.1}$$

where f_{clk} is the main clock frequency and $baud\_rate$ is the desired baud rate.

$$rate = (\frac{baud\_rate \cdot OVER}{f_{clk}} \cdot \ 2^N) - 1, \tag{5.2}$$

where $OVER$ is the oversampling factor (8 or 16) and N is 20 for the 20-bit accumulator of the baud rate generator.

The baud rate generator generates a clock signal for transmitting and receiving purposes. The transmitter transmits data written to the data buffer, and the receiver receives data and writes it into the buffer. Data buffers are accessible through the APB interface. The architecture of the hardware design is shown in the figure 5.1.

Parametrization in the hardware description languages allows the creation of flexible, reusable, and easily maintainable designs. The USART peripheral device has the following hardware parameters:

- g_USART_CLK_GATING: enable/disable clock gating cells in the RTL.
- g_USART_ID_INSTANCE: USART ID of the instance (the read value of the id.ip_inst register).
- g_USART_TX_ENA: enable/disable transmitter instantiation.
- g_USART_RX_ENA: enable/disable receiver instantiation.
- g_USART_FIFO_DEPTH: FIFO buffer depth. The value of this parameter is used as a power of two to calculate the FIFO depth:
  0x0: 1 transfer
  0x1: 2 transfer
  0x2: 4 transfer
  0x3: 8 transfer
  ...

5.2 Hardware design specification

This section defines the design's register map, transmitter's and receiver's FSMs (Finite State Machine), and USART implemented features.

5.2.1 USART register map

All registers in the register map should be accessible through the APB interface. Register access rights are divided into RO (read-only), WO (write-only), and RW (read-write). Register fields can be volatile or non-volatile. The status of a volatile field is set by the RTL design, not by the requester via APB. A status register is an example of a volatile register. Write-only registers are volatile because they clear their value the next cycle after they are set. Each register is 32 bits long. Fields of registers can have any length. The register map is defined in the table B.1 in the appendix B.

5.2.2 USART register bank design

Register bank is divided into two main parts: write to the registers and read from the registers. Registers that have RW or WO access rights can be written through the APB. Write strobes are respected during the write operations. If APB access is detected, but some of the bus signals have an illegal value - PSLVERR will be set, and operation will be ignored. There are transfer error (PSLVERR) activation conditions:

- Unsupported value of the strobe signal (PSTRB).
- Operation on the unmapped address.
- Unaligned address (PADDR[1:0] != 0).
- Write to the read-only register.

5.2.3 Transmitter design

The transmitter's FSM model from the figure 5.2 describes transmitter behavior, state transitions, and outputs. This FSM is a Mealy type; the current state and the current inputs determine outputs. Input/output signals, configuration registers, and internal registers are described in the legend. There are features implemented as part of this FSM model:

- Asynchronous UART frame transmission.
- Synchronous USART frame transmission.
- Full-duplex and half-duplex transmission.
- Break character transmission.
- The sts.tx_busy flag control.
- The driver enable (DE) output control.
- Bits inverting feature.
- Configurable parity bit.
- Configurable number of data bits.

- Configurable number of stop bits.
- Transmission start and end interrupts set feature.
- Continuous transmission feature.

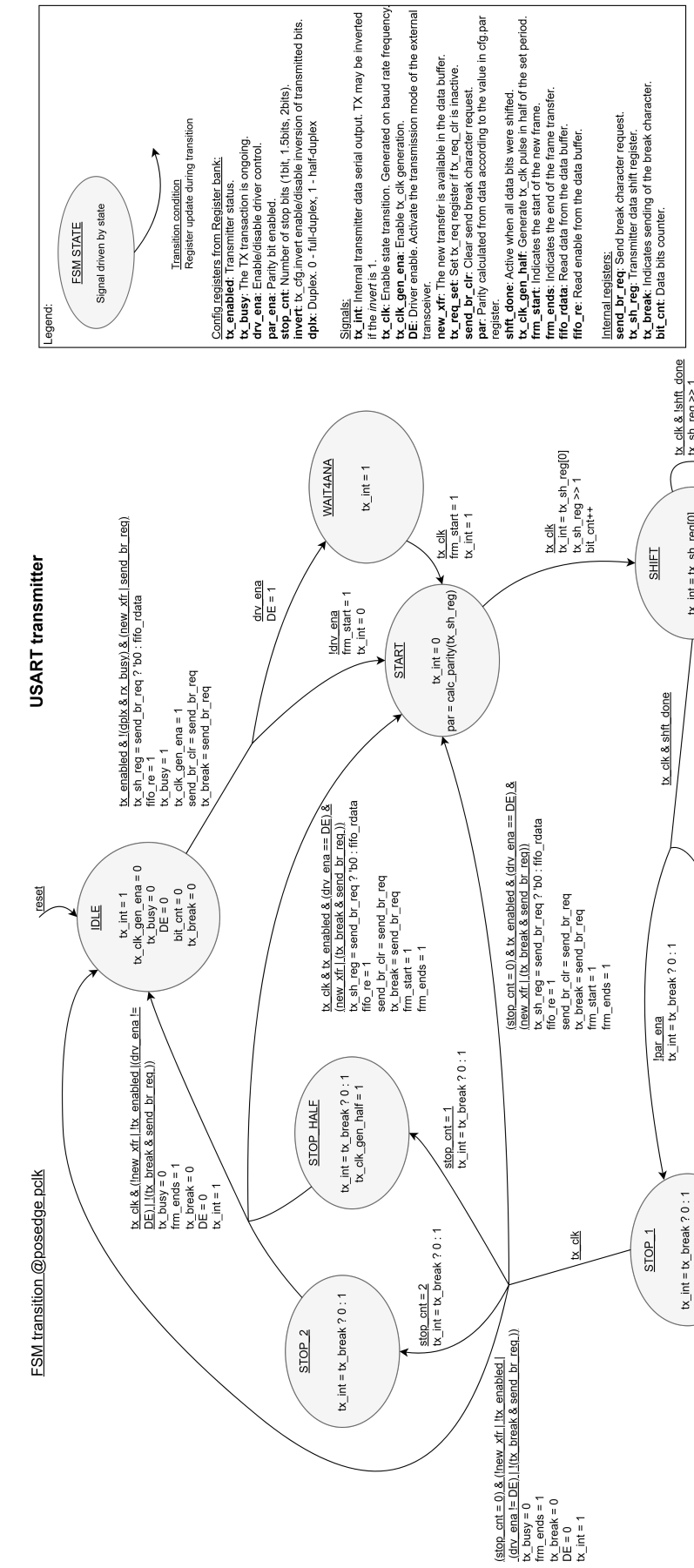


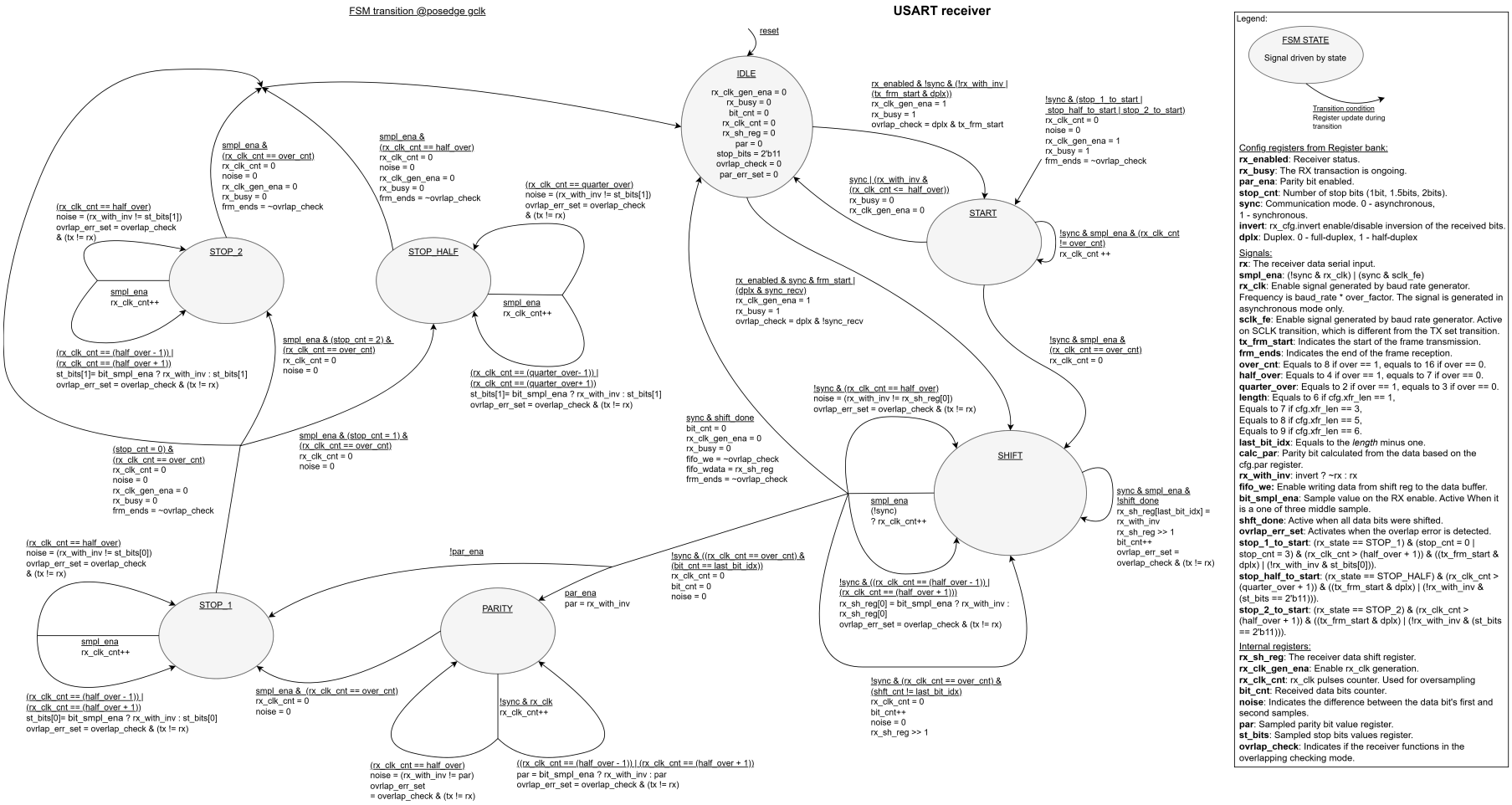
Fig. 5.2: Transmitter's FSM

5.2.4 Receiver design

The receiver's FSM model from figure 5.3 describes receiver behavior, state transitions, and outputs. This FSM is also a Mealy type. The model is designed so many signals change their values into transitions between states, not assigned from the state. As a result, the synthesized design will have more combinational logic but fewer flip-flop registers. This approach is conditioned by the need to minimize the number of logical states.

Input/output signals, configuration registers, and internal registers are described in the legend. There are features implemented as part of this FSM model:

- Asynchronous UART frame reception.
- Synchronous USART frame reception.
- Full-duplex and half-duplex reception.
- Oversampling by 8 or 16 in asynchronous mode with three sampling points.
- Break character reception.
- The sts.rx_busy flag control.
- Start bit error detection.
- Bits inverting feature.
- Parity bit calculation and check. Parity error reporting.
- Configurable number of stop bits.
- Framing error detection.
- Overlapping error detection (works in half-duplex mode during transmission).
- Reception end - interrupt set feature.
- Continuous reception feature.



Config registers from Register bank:
rx_enabled: Receiver status.
rx_busy: The RX transaction is ongoing.
par_ena: Parity bit enabled.
stop_cnt: Number of stop bits (1bit, 1.5bits, 2bits).
sync: Communication mode. 0 - asynchronous,
1 - synchronous.
invert: rx_cfg.invert enable/disable inversion of the received bits.
dplx: Duplex. 0 - full-duplex, 1 - half-duplex

Signals:
rx: The receiver data serial input.
smpl_ena: (!sync & rx_clk) | (sync & sclk_fe)
rx_clk: Enable signal generated by baud rate generator. Frequency is baud_rate * over_factor. The signal is generated in asynchronous mode only.
sclk_fe: Enable signal generated by baud rate generator. Active on SCLK transition, which is different from the TX set transition.
tx_frm_start: Indicates the start of the frame transmission.
frm_ends: Indicates the end of the frame reception.
over_cnt: Equals to 8 if over == 1, equals to 16 if over == 0.
half_over: Equals to 4 if over == 1, equals to 7 if over == 0.
quarter_over: Equals to 2 if over == 1, equals to 3 if over == 0.
length: Equals to 6 if cfg.xfr_len == 1,
Equals to 7 if cfg.xfr_len == 3,
Equals to 8 if cfg.xfr_len == 5,
Equals to 9 if cfg.xfr_len == 6.
last_bit_idx: Equals to the *length* minus one.
calc_par: Parity bit calculated from the data based on the cfg.par register.
rx_with_inv: invert ? ~rx : rx
fifo_we: Enable writing data from shift reg to the data buffer.
bit_smpl_ena: Sample value on the RX enable. Active When it is a one of three middle sample.
shft_done: Active when all data bits were shifted.
ovrlap_err_set: Activates when the overlap error is detected.
stop_1_to_start: (rx_state == STOP_1) & (stop_cnt = 0 | stop_cnt = 3) & (rx_clk_cnt > (half_over + 1)) & ((tx_frm_start & dplx) | (!rx_with_inv & st_bits[0])).
stop_half_to_start: (rx_state == STOP_HALF) & (rx_clk_cnt > (quarter_over + 1)) & ((tx_frm_start & dplx) | (!rx_with_inv & (st_bits == 2'b11))).
stop_2_to_start: (rx_state == STOP_2) & (rx_clk_cnt > (half_over + 1)) & ((tx_frm_start & dplx) | (!rx_with_inv & (st_bits == 2'b11))).

Internal registers:
rx_sh_reg: The receiver data shift register.
rx_clk_gen_ena: Enable rx_clk generation.
rx_clk_cnt: rx_clk pulses counter. Used for oversampling
bit_cnt: Received data bits counter.
noise: Indicates the difference between the data bit's first and second samples.
par: Sampled parity bit value register.
st_bits: Sampled stop bits values register.
ovrlap_check: Indicates if the receiver functions in the overlapping checking mode.

Fig. 5.3: Receiver's FSM

5.3 Lint check

Lint is a tool for static verification of the HDL code. Linting doesn't require any stimulus; it works only with the RTL code. During the lining process, HDL code is verified for coding errors, synthesis and simulation mismatches, connectivity errors, etc.

The Siemens Questa Lint tool was used in this work. It provides a lot of checks based on industry best practices. The results are presented in text and graphical format. The lint tool was run right after the design phase. It found two combinational loops in the design, which, in the long run, helped save a lot of time during the verification phase. Also, it found arithmetic operations inside of the condition blocks. This problem was solved by separating the arithmetic operation into an independent process.

5.4 Synthesis Results

Logic synthesis, or RTL synthesis, is a process by which the abstract specification of the circuit behavior described in the HDL (Hardware Description Language) is turned into the design implementation. The design is implemented in the logical gates and technology-specific cells [16].

Cell count and type, area, timings, and power estimates are technology-specific and implementation-specific parameters. The synthesis results are presented in the number of gate cells in NAND equivalent specific for used technology and the number of ports in the table 5.1. Outputs are presented for different selected values of the hardware parameters. Hardware parameters are described in the section 5.2.

Table 5.1: USART synthesis results

Parameters	**Gate count**	**Port count**
FIFO_DEPTH=0 (single data buffer)	3620.25	106
FIFO_DEPTH=5 (32-word FIFO buffer)	10928.75	106
TX_ENA=0 (receiver only)	1807.5	101
RX_ENA=0 (transmitter only)	1746.5	102

5.5 Verification Environment Architecture

The verification environment is created using the UVM framework. The architecture of the verification environment is shown in the figure 5.4. The APB QVIP (Questa Verification Intellectual Property) is used as a requester side of communication in the

verification. It includes register model support, coverage, and sequences to perform read/write operations on the bus.

The "vip_usart_env" environment is developed as a USART verification IP. The "rx_agent" included in this environment is a passive agent that should receive and convert frames on the transaction level. The "tx_agent" should be able to send USART frames.

The "usart_ifc_env" includes other agents required in this project. The "usart_env" includes the "usart_ifc_env" and the "vip_usart_env" environments, and a scoreboard with a predictor. The predictor receives transactions from the APB QVIP, "tx_agent", and "rx_agent". Based on these transactions, the predictor can predict the operations the USART periphery performs. Also, the predictor predicts the values of the registers in the design. A scoreboard compares transactions from the predictor (expected) with transactions from the "rx_agent" (actual).

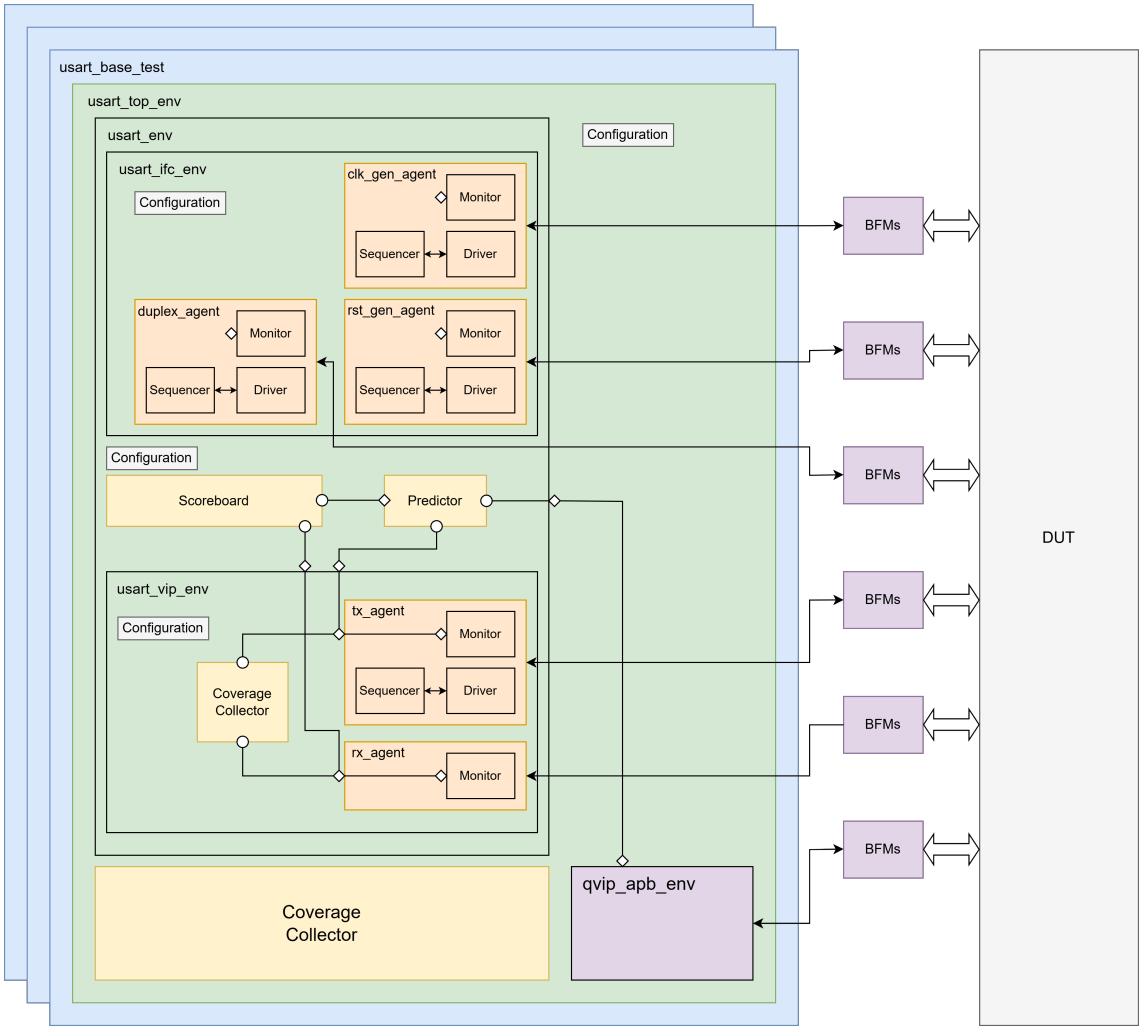


Fig. 5.4: USART verification environment architecture

5.5.1 Predictor Implementation

Predictor is a TLM-based verification analysis component. Inputs of this component are analysis exports. Transactions are transmitted to the predictor by calling the function "write(transaction_object)" inside the predictor class. When the expected transaction is ready to be sent to the scoreboard, the "write(expected_transaction)" function is called on the analysis port. This project's predictor includes three main TLM models: transmitter, receiver, and FIFO model.

First TLM input is associated with the register model "read/write notify callback". This callback is created and configured inside the register model. It monitors operations on the selected fields and creates a transaction object with information about the operation. The predictor model changes its state and makes predictions using it.

The second TLM input receives transactions from the USART VIP TX agent's monitor. These transactions are used to predict the RX hold register value and status bits associated with the RX part.

Predictor is a class; only non-time-consuming expressions are allowed inside it. It naturally interferes with the possibility of creating a timing-accurate model. Imagine this sequence of operations: three words are written to the TX hold register, and during the transmission of the first frame, the transmitter is disabled. That means the transmitter is expected to be disabled after the transmission ends, and the FIFO buffer shall be reset. The predictor without any timing control can't handle such cases. A simple predictor will immediately send all the words written to the hold register to the scoreboard. However, there will be mismatches in the case above.

Solution shown in the figure 5.5 is used to handle this problem. The workflow is described in the following steps:

1. When a new frame transmission shall be started, the event is triggered by the predictor model. The event is located inside the BFM module.
2. The process inside the BFM waits for the event trigger. When an event is triggered, the time measuring starts. Measured time corresponds with the expected frame duration.
3. At the end of the wait process, a transaction is created and sent back to the predictor via the analysis port.
4. Predictor model execute the "write()" function associated with this analysis port. If the FIFO model is not empty and other required conditions are met, the next transaction will be sent to the scoreboard, and the event will be triggered again.

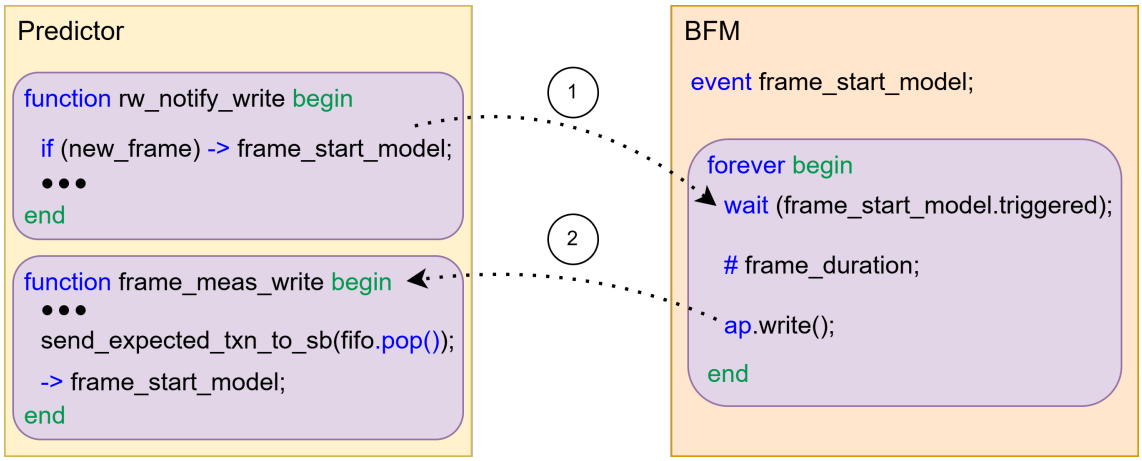


Fig. 5.5: Predictor timing control mechanism

This solution enters the timing control into the prediction model and solves many problems of the "predictor & scoreboard" verification strategy.

5.6 Verification Flow

This section describes the USART verification flow. The verification flow selected to verify the peripheral device is specific due to the use of the UVMF. On the other hand, the UVM-based verification flow is almost the same. The main advantage of the UVMF-based flow from the point of view of the steps is automatic verification environment generation. This feature significantly accelerates the verification process.

5.6.1 UVMF Environment Generation Phase

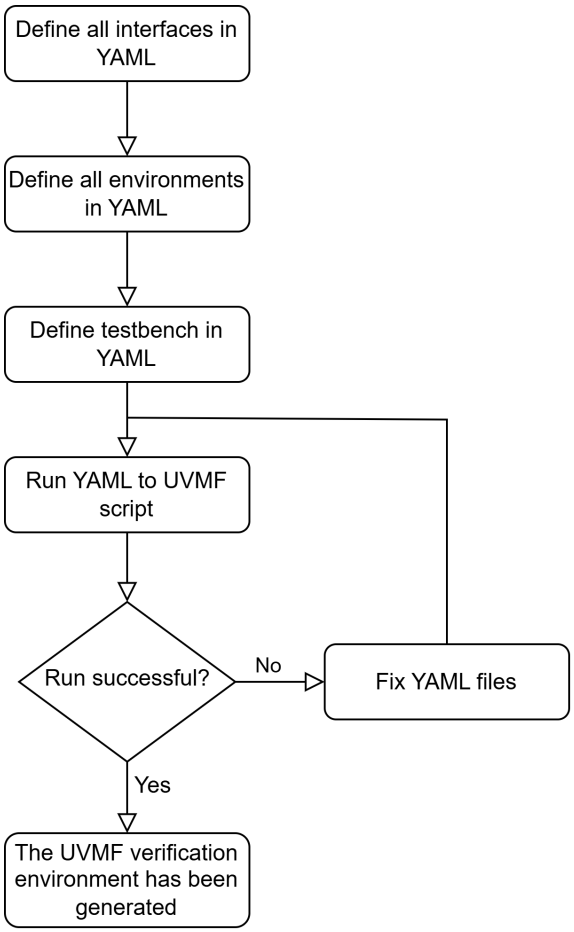


Fig. 5.6: Verification Flow: UVMF environment generation flowchart

As described in chapter 4, the UVMF includes the YAML to UVMF generation script. The YAML constructions are divided into three types: interface, environment, and testbench. This order is better to follow when defining the new verification environment. The script checks only some of the possible user mistakes. After the generation and bug fix step, the output directories shall be copied to the main verification directory. The UVMF environment generation phase is shown in the 5.6 figure.

5.6.2 Integration and Evaluation Phase

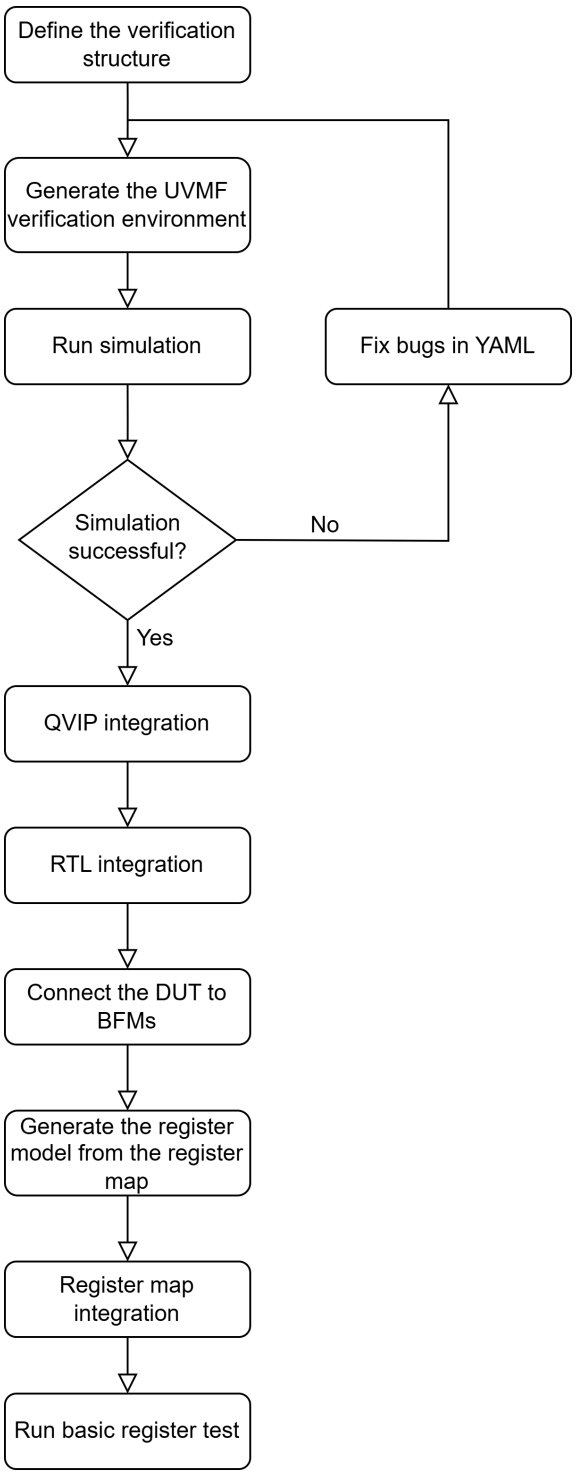


Fig. 5.7: Verification Flow: Integration and evaluation flowchart

After the UVMF project structure has been generated, the compilation shall be started to check that everything works. There can be different problems, such as

mistakes in the YAML, importation of the packages between parts of the testbench, problems with importing the UVM and UVMF packages, and others.

The process can be continued when the simulation runs successfully on all used simulators. The APB QVIP shall be configured and integrated into the testbench. This component connects to the DUT on a signal level and to the register model on the transaction level. When the operation is called on the register model register, the APB QVIP converts it on the APB-specific transaction and drives it on the signal level using an internal driver.

The RTL modules shall be included in the compilation files. After that, the top design module shall be instantiated inside of the testbench top. The design shall be connected to the global clock, reset, and all BFMs signals.

The next step is a register model generation. There are a lot of register model generators available. The register model for this project is generated from the XLSX file format. This register model replaces the standard UVMF generator output register model. The basic register test can be run after that step. A full flowchart of the evaluation and integration phase is shown in the figure 5.7.

5.6.3 Pre-testplan Phase

All components shall be integrated and evaluated by this moment. The next step is the implementation of the agent's BFMs (monitor and driver). The USART VIP consists of two agents: the transmit and receive agent. The BFMs are designed to operate in both synchronous and asynchronous modes. Behaviors of the BFMs depend on the values of the configuration properties. The configuration properties can be changed dynamically from any test or sequence.

Then, the agent's and environment's sequences shall be implemented. Sequences set the values of the transactions driven by the driver on the signal level afterward. Sequences are divided by type, for example, correct, framing error, parity error, or line break sequence.

After all agent and environment level sequences are done, the example test and example test sequence shall be created. The first step in the sequence is the agents and design configuration. The second step is calling sequences and performing the APB write and read operations using the register model. Results can be checked directly from the sequence. However, usually, result checking is provided by the predictor with the scoreboard or the assertions.

When the example test succeeds and BFM's behaviors are checked, the predictor shall be implemented. The implementation of the predictor includes the implementation of the USART and FIFO transaction-level models. The predictor receives the

transaction objects from the register model and the TX agent of the USART VIP. Detailed implementation of the predictor is described in the 5.5.1 subsection.

A test plan is written after it. Writing the test plan in that late step is not usual. This approach is used because of the knowledge of the implemented predictor possibilities and limitations. In other words, in this step, it's obvious what can be checked by the predictor, and the rest shall be checked by the assertions or direct checks from the sequences. A full flowchart of the pre-test plan phase is shown in figure 5.8.

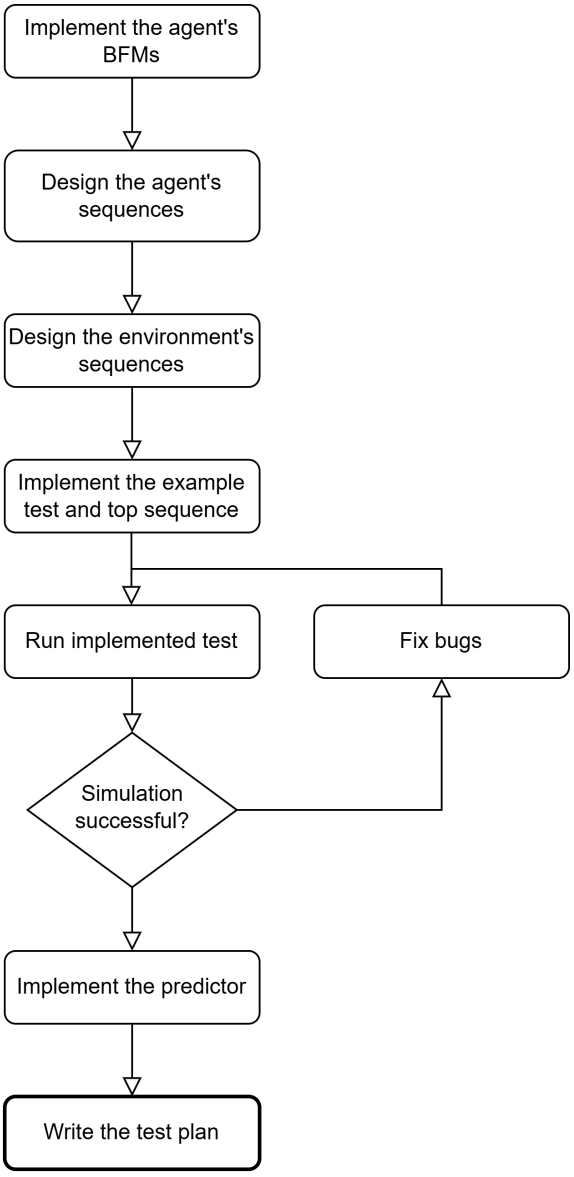


Fig. 5.8: Verification Flow: Pre-testplan flowchart

5.6.4 Post-testplan Phase

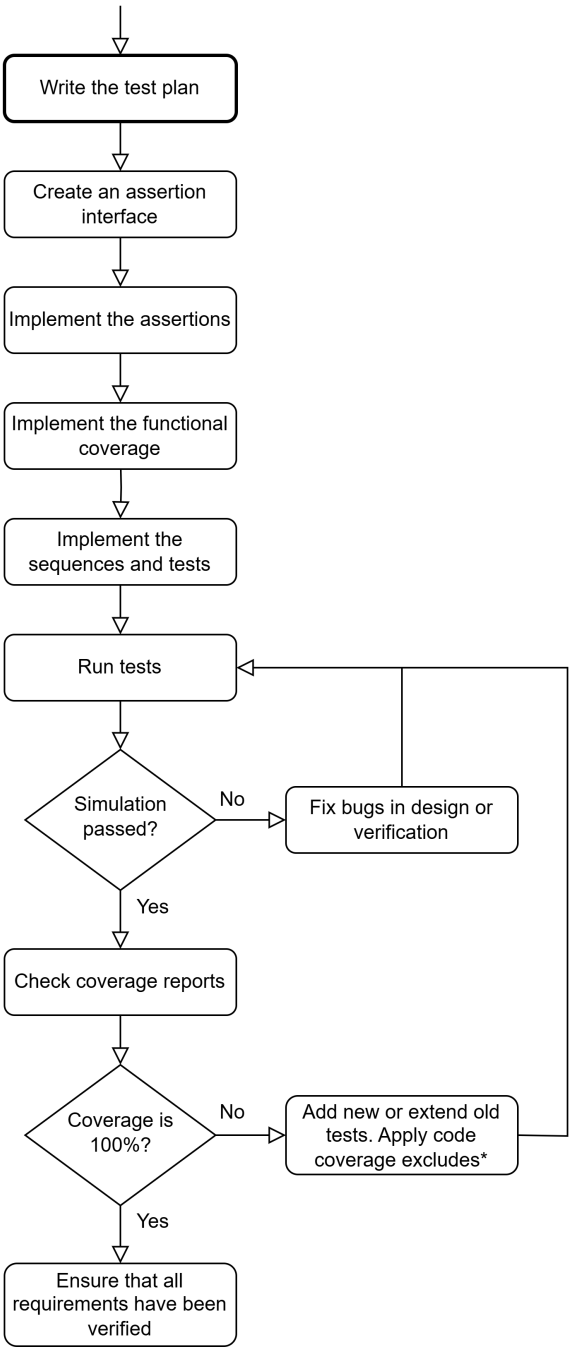


Fig. 5.9: Verification Flow: Post-testplan flowchart

The last phase of the verification process starts with creating the assertion interface. This interface is bound to the top design module. There are a lot of signals inside the interface that are connected to the internal registers and ports of the design using hierarchical paths.

Assertions are included in the interface. They are designed to check the design functionality that the predictor can't check. Assertions are described in detail in the section 3.6.

The next step is functional coverage. The coverage associated with the transactions shall be implemented in a separate class and integrated into the verification environment. The coverage that is associated with the signal activity shall be placed inside of the assertion interface in the form of the cover properties.

The last implementation step is a sequence and test design according to the test plan. When all tests are implemented, they shall be run with different randomization seeds. Then, errors shall be resolved by fixing the bugs inside the RTL design or the verification environment. A mismatch between specification and RTL implementation is the most frequent issue.

After resolving all errors, the coverage reports shall be analyzed. The incompleteness of the coverage can have two reasons: lack of stimuli (tests) or impossibility of coverage. The first reason can be solved by adding new tests or extending the existing ones. The second reason shall be analyzed in more detail. If some piece of code is unreachable on purpose, it shall be excluded from the coverage reports. Otherwise, the reachability can be checked using formal verification techniques, the design can be fixed, or coverage can be changed.

Simulations and coverage shall be complete for all supported combinations of the design parameters. The last step is to ensure that all hardware requirements have been verified. Figure 5.9 shows a flowchart of the post-test plan phase.

5.7 Verification Tests

Tests that were created to verify the USART design functionality are covered in this section. All tests are described in the table 5.2.

Table 5.2: USART verification tests

Test name	**Description**
apb_test	APB test. Perform read and write operations on different aligned and unaligned addresses. Write with legal and illegal strobes (APB PSTRB signal values). Execute operations with invalid PSEL and PENABLE values.

register_test	Register test. Check reset values of all registers defined in the RM (register map). Sequentially writes 1's and 0's in each bit of the register, checking that it is appropriately set or cleared, based on the field access policy. Writes all registers, then confirms the value was written using the back-door. Subsequently writes a value via the back-door and checks whether the corresponding value can be read through the address map [13].
baud_rate_sweep_test	Baud rate sweep test. Randomize and set all configuration registers. Transmit and receive synchronously and asynchronously on a randomized baud rate with a randomized oversampling factor. Randomly set and clear IENA (interrupt enable) bits during the test execution. Perform UVM RAL mirrors after each frame to check the actual register value against the predicted value.
async_full_duplex_test	Asynchronous full-duplex communication test. Randomize and set all configuration registers. Transmit and receive asynchronously on a randomized baud rate, oversampling factor, and IENA configuration.
sync_full_duplex_test	Synchronous full-duplex communication test. Randomize and set all configuration registers. Transmit and receive synchronously on a randomized baud rate, oversampling factor, and IENA configuration.
async_half_duplex_test	Asynchronous half-duplex communication test. Randomize and set all configuration registers. Transmit and receive asynchronously in half-duplex mode on a randomized baud rate, oversampling factor, and IENA configuration.

sync_half_duplex_test	Synchronous half-duplex communication test. Randomize and set all configuration registers. Transmit and receive synchronously in half-duplex mode on a randomized baud rate, oversampling factor, and IENA configuration.
ena_dis_full_dplx_test	TX/RX enable/disable full-duplex test. Randomize and set all configuration registers. Transmit and receive synchronously and asynchronously in full-duplex mode on a randomized baud rate, oversampling factor, and IENA configuration. Enable and disable the transmitter or receiver during the transmission or reception during the random part of the frame.
fifo_test	USART FIFO test. Perform writes and reads on TX and RX data buffers. Test to toggle with the fifo_ena configuration bit when FIFO is not empty. Test FIFO overrun. Perform simultaneous read and write. Randomize FIFO's trigger levels.
par_and_frm_err_test	Parity and framing error test. Randomize and set all configuration registers. Receive asynchronous frames with parity or framing error. Perform UVM RAL mirrors after each frame. Receive synchronous frames with parity or framing error to verify that these errors aren't checked in synchronous mode.
break_char_test	Break character test. Transmit break characters. Set break character request during ongoing transaction or when the transmitter is disabled. Receive a break character asynchronously.
oversampling_test	Oversampling test. Receive a short start bit. Receive a frame with randomly delayed bits. Randomized delay of each bit is in the range from 0% to 50%. Verify that receiver oversampling works correctly.

overlapping_test	Overlapping test. Verify the functionality of the receiver's overlapping mode in half-duplex mode.
reset_test	Software and hardware reset test. Activate the SW or HW reset in the randomized phase of the USART function.
fifo_reset_test	FIFO buffers reset test. Activate the TX and RX FIFO reset in all phases of the USART operation.
half_dplx_early_tx_rx_test	Half-duplex negative test. Start transmission while half-duplex receiving is ongoing. Start reception while half-duplex transmission is ongoing.
async_to_sync_recfg_test	Changing configuration during operation negative test. Change the synchronization mode while transmission/reception is ongoing.

5.8 Coverage Results

Coverage in hardware design verification is described in the section 3.7. There are covergroups created following the functional specification:

- cg_apb - APB operations, addresses, strobes, and other signals. Their combinations.
- cg_bd_gen_tx - Baud rate generator (TX part). Values of the configuration register fields and the "div" field. Combinations between them.
- cg_bd_gen_rx - Baud rate generator (RX part). Values of the configuration register fields and the "rate" field. Combinations between them.
- usart_tx_cg - The transmitter. Transmissions, hold register operations, transmitter state, configuration register fields, data buffer status, status register fields, control register fields, resets, operation modes, values of the frame parts, DMA-related signals, and interrupts. Combinations of selected parts.
- usart_rx_cg - The receiver. Receptions, hold register operations, receiver state, configuration register fields, data buffer status, status register fields, control register fields, resets, operation modes, values of the frame parts, DMA-related signals, and interrupts. Combinations of selected parts.

5.8.1 Code Coverage

Automatic code coverage exclusions were generated using the Siemens Covercheck tool. The tool uses formal verification methods to prove the unreachability of the selected code. These code coverage exclusions shall be reviewed before use. Also, manual code coverage exclusions can be applied if needed. Use of each manual exclusion shall be justified. Code coverage result before applied exclusions is 97.2%. Code coverage result with exclusions is 100%.

5.8.2 Functional Coverage

To reach full function coverage, many tests with different seeds and proper randomization shall be run. Final functional coverage of the USART peripheral device is 100%.

Conclusion

This master's thesis focused on the complete design, implementation, and verification of a Universal Synchronous and Asynchronous Receiver-Transmitter (USART) peripheral, fully compatible with the AMBA APB protocol. The main goal was to create a reliable, flexible, and efficient communication module that supports both UART and USART modes, making it suitable for integration into modern SoC designs.

The thesis began with an in-depth analysis of serial communication protocols, highlighting the strengths and typical applications of UART and USART. The hardware design was developed with attention to detail, resulting in a well-structured architecture. The transmitter and receiver modules were described using finite state machines, ensuring clear and predictable behavior. The design also incorporated advanced features such as clock-gating for power savings and support for one-wire half-duplex communication, which is useful in applications where minimizing the number of wires is important.

A comprehensive register map was created to provide a clear interface for software control and configuration of the peripheral. This register map is included in Appendix B, offering a structured overview of all configuration, status, and control registers.

The definition of functional requirements was a key part of the design process. These requirements were written using the EARS (Easy Approach to Requirements Syntax) methodology, which ensures clarity, consistency, and readability. The complete digital design specification, including all requirements written in the EARS format, is provided in "Attachment 1 functional_specification". This specification serves as a solid foundation for both implementation and verification.

Verification was a major focus of the thesis. A reusable and modular verification environment was built using the UVM and the UVM Framework. The verification process was carefully planned and executed to ensure that all functional requirements were thoroughly tested. The test plan, included as "Attachment 2 test_plan", details how each requirement is verified, which tests are used, and which covergroups are responsible for coverage. This structured approach made it possible to achieve 100% code and functional coverage, confirming that the design behaves as expected in all scenarios. The design also passed lint checks and was successfully synthesized, demonstrating its quality and readiness for SoC integration.

In summary, this thesis achieved all its objectives. The resulting USART peripheral is robust, flexible, and well-documented. The design is supported by a clear register map, a detailed requirements specification, and a comprehensive verification plan. All these supporting materials are included in the appendix and attachments.

The systematic approach to both design and verification ensures that the module can be confidently integrated into embedded systems and further developed in the future. This work provides a strong foundation for reliable serial communication in a wide range of digital applications.

Bibliography

[1] Bhadra, Dipanjan and Vij, Vikas S. and Stevens, Kenneth S., "A low power uart design based on asynchronous techniques," in *2013 IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2013, pp. 21–24.

[2] Microchip, "Usart in one-wire mode," 2018. [Online]. Available: https://www.microchip.com/content/dam/mchp/documents/OTH/ApplicationNotes/ApplicationNotes/AN2658-USART-In-One-Wire-Mode-00002658C.pdf

[3] ARM, "Amba apb protocol specification," February 2023. [Online]. Available: https://developer.arm.com/documentation/ihi0024/e/?lang=en

[4] Siemens Verification Methodology Team, "Uvm cookbook," 2018. [Online]. Available: https://verificationacademy.com/cookbook/uvm-universal-verification-methodology/

[5] Wang, Yongcheng and Song, Kefei, "A new approach to realize uart," in *Proceedings of 2011 International Conference on Electronic & Mechanical Engineering and Information Technology*, vol. 5, 2011, pp. 2749–2752.

[6] FastBitLab, "Stm32 usart lecture 7 - usart oversampling," *FastBit EBA*, Nov 2022. [Online]. Available: https://fastbitlab.com/usart-oversampling/

[7] STMicroelectronics, "Rm0390 reference manual," March 2021. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0390-stm32f446xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

[8] Infineon, "Universal asynchronous receiver transmitter (uart) component datasheet," May 2012. [Online]. Available: https://www.infineon.com/dgdl/Infineon-Component_UART_V2.20-Software%20Module%20Datasheets-v02_05-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e7f95451159

[9] Levido Andrew, "Binary rate multipliers," Oct 2024. [Online]. Available: https://circuitcellar.com/resources/quickbits/binary-rate-multipliers/

[10] Bhutada, Rani and Manoli, Yiannos, "Complex clock gating with integrated clock gating logic cell," in *2007 International Conference on Design Technology of Integrated Systems in Nanoscale Era*, 2007, pp. 164–169.

[11] IEEE Standards Association, "Ieee standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)*, pp. 1–1354, 2024.

[12] Keisuke Shimizu, “Uvm tutorial for candy lovers,” 2011. [Online]. Available: https://cluelogic.com/

[13] Systems Initiative, Accelera, “Uvm user guide,” May 2011. [Online]. Available: https://www.accellera.org/downloads/standards/uvm

[14] Gordon Allan, Gabriel Chidolue, Thomas Ellis, Harry Foster, Michael Horn, Peet James, Mark Peryer, “Coverage cookbook,” 2019. [Online]. Available: https://verificationacademy.com/cookbook/coverage/

[15] Verification Academy, Siemens, “Uvmf user guide,” 2024, included in the UVM Framework source distribution, available as a part of a compressed archive at the official website. [Online]. Available: https://verificationacademy.com/topics/uvm-universal-verification-methodology/uvmf/

[16] Wikipedia contributors, “Logic synthesis,” July 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Logic_synthesis&oldid=1236324508

Symbols and abbreviations

AMBA Advanced Microcontroller Bus Architecture

APB Advanced Peripheral Bus

ASIC Application Specific Integrated Circuit

BFM Bus Functional Model

CPU Central Processing Unit

DMA Direct Memory Access

DUT Design Under Test

EARS The Easy Approach to Requirements Syntax

FIFO First In First Out

FPGA Field Programmable Gate Array

FSM Finite State Machine

HDL Hardware Description Language

LSB Least Significant Bit

MSB Most Significant Bit

OOP Object-Oriented Programming

QVIP Questa Verification Intellectual Property

RTL Register-Transfer Level

SoC System on Chip

SPI Serial Peripheral Interface

SVA SystemVerilog Assertions

TLM Transaction Level Modeling

UART Universal Asynchronous Receiver-Transmitter

USART Universal Synchronous and Asynchronous Receiver-Transmitter

UVM Universal Verification Methodology

UVMF Universal Verification Methodology Framework

List of appendices

A Content of the electronic attachment

- /.....root of the attached archive
  - docs ..... project documentation
    - register_map.xlsm
    - functional_specification.pdf
    - test_plan.docx
  - rtl.....USART RTL source files
    - usart_top.sv
    - usart_tx.sv
    - usart_rx.sv
    - usart_regs.sv
    - usart_baud_rate_gen.sv
    - usart_fifo.sv
    - usart_cgc.sv
    - usart_pkg.sv
  - uvmf.....UVFM based verification environment
    - qvip ..... Questa Verification IP
    - project_benches...HDL part of verification environment (non-reusable part)
    - verification_ip.....HVL part of verification environment (reusable part)
    - yaml.....UVMF generator's input files

B USART register map

Table B.1: USART register map

Register name	Addr	Field name	Description	Range	Access type
cfg	0x00		USART configuration register		
		xfr_len	USART transfer length. 0: 5bits 1: 6bits 2: Reserved 3: 7bits 4: Reserved 5: 8bits 6: 9bits 7: Reserved	[2:0]	RW
		stop_cnt	Number of stop bits. 0: 1bit 1: 1.5bits 2: 2bits 3: Reserved	[4:3]	RW
		par_ena	Parity bit enable. 0: disabled 1: enabled	[5]	RW
		par	Parity bit configuration. 0: odd 1: even 2: sticky 0 3: sticky 1	[7:6]	RW
		dplx	USART duplex configuration. 0: full-duplex 1: half-duplex	[8]	RW
		sync	Asynchronous/Synchronous mode. 0: Asynchronous 1: Synchronous	[9]	RW
		dma_mode	DMA transfer mode. 0: Single DMA transfers. TXRDYn becomes active if TX FIFO is empty and deactivates otherwise. RXRDYn becomes active if RX FIFO is not empty, deactivates otherwise. 1: Block DMA transfers. TXRDYn becomes active if TX FIFO reaches the trigger level from the top. RXRDYn becomes active if RX FIFO reaches the trigger level from the bottom.	[10]	RW

		fifo_ena	Enable/disable FIFO functionality. 0: RX/TX FIFOs are disabled 1: RX/TX FIFOs are enabled	[11]	RW
		clk_pol	SCLK output clock signal polarity. Has effect only in the synchronous mode. 0: Steady low value on SCLK pin outside transmission window. 1: Steady high value on SCLK pin outside transmission window.	[12]	RW
		clk_pha	SCLK output clock signal phase. Has effect only in the synchronous mode. 0: The first SCLK transition is the first data sample edge. 1: The second SCLK transition is the first data sample edge.	[13]	RW
tx_cfg	0x04		USART transmitter configuration register		
		dma_ena	Enable/disable transmitter DMA requests. 0: No DMA requests 1: DMA request is generated when the data hold register is empty	[0]	RW
		drv_ena	Enable/disable driver control. 0: driver enable output is always low 1: driver enable (DE) output is - high when data is being transmitted - low when data is not being transmitted	[1]	RW

		invert	Enable/Disable inversion of transmitted bits. 0: inversion is disabled 1: all bits are inverted, includes start and stop bits	[2]	RW
		trig_lvl	Transmitter FIFO trigger level. A value greater than the maximum defined by g_FIFO_DEPTH will be wrapped.	[31:16]	RW
rx_cfg	0x08		USART receiver configuration register		
		over	Oversampling mode. 0: oversampling by 16 1: oversampling by 8	[0]	RW
		dma_ena	Enable/disable receiver DMA requests. 0: No DMA requests 1: DMA request is generated when new data are received.	[1]	RW
		invert	Enable/Disable inversion of received bits. 0: inversion is disabled 1: all bits are inverted, including start and stop bits	[2]	RW
		trig_lvl	Receiver FIFO trigger level. A value greater than the maximum defined by g_FIFO_DEPTH will be wrapped.	[31:16]	RW
br_tx_cfg	0x0C		Baud rate generator TX configuration register		

		div	Baud rate generator division constant. Values for 50MHz clock frequency: 0xA2C2 : 1200 baud/s 0x1458 : 9600 baud/s 0x01B2 : 115200 baud/s 0x0186 : 128 kbaud/s 0x00C3 : 256 kbaud/s 0x0061 : 512 kbaud/s 0x0032 : 1 Mbaud/s 0x0019 : 2 Mbaud/s	[15:0]	RW
br_rx_cfg	0x10		Baud rate generator RX configuration register		
		rate	Binary rate multiplier coefficient. Values for 50MHz clock frequency and 16 oversampling: 0x00192 : 1200 baud/s 0x00C94 : 9600 baud/s 0x096FE : 115200 baud/s 0x0A7C5 : 128 kbaud/s 0x14F8A : 256 kbaud/s 0x29F16 : 512 kbaud/s 0x51EB7 : 1 Mbaud/s 0xA3D70 : 2 Mbaud/s	[19:0]	RW
ctrl	0x14		USART control register		
		reset	Write 1 to reset the USART.	[0]	WO
		tx_ena	Write 1 to enable the USART transmitter.	[1]	WO
		tx_dis	Write 1 to disable the USART transmitter.	[2]	WO
		rx_ena	Write 1 to enable the USART receiver.	[3]	WO
		rx_dis	Write 1 to disable the USART receiver.	[4]	WO
tx_ctrl	0x18		USART transmitter control register		
		fifo_reset	Reset the transmitter FIFO buffer.	[0]	WO

		send_br	This bit is used to send a break character. If data transmitting is ongoing, a break character will be sent after this frame is finished.	[1]	WO
rx_ctrl	0x1C		USART receiver control register		
		fifo_reset	Reset the receiver FIFO buffer.	[0]	WO
		sync_recv	Receive frame in synchronous mode. SCLK will be generated, but data will not be transmitted. Works only in synchronous half-duplex mode.	[1]	WO
tx_hr	0x20		USART transmitter data hold register		
		hold_reg	Transmitter data hold register. If g_USART_FIFO_DEPTH != 0, the data will be written into the FIFO buffer.	[8:0]	RW
rx_hr	0x24		USART receiver data hold register		
		hold_reg	Receiver data hold register. If g_USART_FIFO_DEPTH != 0, the data will be read from the FIFO buffer.	[8:0]	RO
sts	0x28		USART status register		
		tx_enabled	Current USART transmitter status. 0: transmitter disabled 1: transmitter enabled	[0]	RO
		rx_enabled	Current USART receiver status. 0: receiver disabled 1: receiver enabled	[1]	RO
		tx_req	New data to transmit can be written.	[2]	RO
		rx_req	New received data are availible to be read.	[3]	RO
		tx_busy	TX transaction in ongoing.	[4]	RO

	rx_busy	RX transaction in ongoing.	[5]	RO
	rx_ovr_err	RX overrun error flag. HW sets when the new data is received, but old data wasn't read from the hold register or receiver FIFO buffer. FW clears by writing 1 to a relevant bit in the ists_clr register.	[6]	RO
	tx_ovr_err	TX overrun error flag. HW sets when the new data is received through APB, but the transmitter hold register or FIFO buffer is already full. FW clears by writing 1 to a relevant bit in the ists_clr register.	[7]	RO
	par_err	Parity error flag. HW sets when the calculated parity of newly received data isn't equal to the parity in that frame. FW clears by writing 1 to a relevant bit in the ists_clr register.	[8]	RO
	frm_err	Framing error flag. HW sets where the stop bit should be when detecting the low level. HW sets when the high level is detected where the start bit should be in synchronous mode. FW clears by writing 1 to a relevant bit in the ists_clr register.	[9]	RO
	ovrlap_err	Overlap error flag. HW sets when received value of the bit is not equal to the transmitted value in half-duplex mode. FW clears by writing 1 to a relevant bit in the $ists_clrregister$.	[10]	RO

		lin_br	Line break flag. HW sets when the break character is detected on the RX port. FW clears by writing 1 to a relevant bit in the ists_clr register.	[11]	RO
		tx_trig	TX FIFO level triggered flag. HW sets when the number of words in the TX FIFO is equal or less than the trigger level. HW clears when the number of words in the TX FIFO is greater than the trigger level.	[12]	RO
		rx_trig	RX FIFO level triggered flag. HW sets when the number of new frames in the RX FIFO is equal or greater than the trigger level. HW clears when the number of new frames in the RX FIFO is less than the trigger level.	[13]	RO
fifo_sts	0x2C		USART FIFOs status register		
		tx_cnt	Number of frames in TX FIFO.	[15:0]	RO
		rx_cnt	Number of frames in RX FIFO.	[31:16]	RO
iena	0x30		USART interrupt enable register		
		rx_ovr_err	RX overrun error interrupt enable.	[0]	RO
		tx_ovr_err	TX overrun error interrupt enable.	[1]	RO
		par_err	Parity error interrupt enable.	[2]	RO
		frm_err	Framing error interrupt enable.	[3]	RO
		ovrlap_err	Overlap error interrupt enable.	[4]	RO
		lin_br	Line break interrupt enable.	[5]	RO
		tx_trig	TX FIFO trigger interrupt enable.	[6]	RO
		rx_trig	RX FIFO trigger interrupt enable.	[7]	RO
		tx_start	Transmission starts interrupt enable.	[8]	RO
		tx_ends	Transmission ends interrupt enable.	[9]	RO
		rx_ends	Reception ends interrupt enable.	[10]	RO

iena_clr	0x34		USART interrupt enable register clear		
		rx_ovr_err	Clear RX overrun error interrupt enable. FW uses W1C operation to clear relevant iena bit.	[0]	WO
		tx_ovr_err	Clear TX overrun error interrupt enable. FW uses W1C operation to clear relevant iena bit.	[1]	WO
		par_err	Clear parity error interrupt enable. FW uses W1C operation to clear relevant iena bit.	[2]	WO
		frm_err	Clear framing error interrupt enable. FW uses W1C operation to clear relevant iena bit.	[3]	WO
		ovrlap_err	Clear overlap error interrupt enable. FW uses W1C operation to clear relevant iena bit.	[4]	WO
		lin_br	Clear line break interrupt enable. FW uses W1C operation to clear relevant iena bit.	[5]	WO
		tx_trig	Clear TX FIFO trigger interrupt enable. FW uses W1C operation to clear relevant iena bit.	[6]	WO
		rx_trig	Clear RX FIFO trigger interrupt enable. FW uses W1C operation to clear relevant iena bit.	[7]	WO
		tx_start	Clear transmission starts interrupt enable. FW uses W1C operation to clear relevant iena bit.	[8]	WO

		tx_ends	Clear transmission ends interrupt enable. FW uses W1C operation to clear relevant iena bit.	[9]	WO
		rx_ends	Clear reception ends interrupt enable. FW uses W1C operation to clear relevant iena bit.	[10]	WO
iena_set	0x38		USART interrupt enable register set		
		rx_ovr_err	Set RX overrun error interrupt enable. FW uses W1S operation to set relevant iena bit.	[0]	WO
		tx_ovr_err	Set TX overrun error interrupt enable. FW uses W1S operation to set relevant iena bit.	[1]	WO
		par_err	Set parity error interrupt enable. FW uses W1S operation to set relevant iena bit.	[2]	WO
		frm_err	Set framing error interrupt enable. FW uses W1S operation to set relevant iena bit.	[3]	WO
		ovrlap_err	Set overlap error interrupt enable. FW uses W1S operation to set relevant iena bit.	[4]	WO
		lin_br	Set line break interrupt enable. FW uses W1S operation to set relevant iena bit.	[5]	WO
		tx_trig	Set TX FIFO trigger interrupt enable. FW uses W1S operation to set relevant iena bit.	[6]	WO

		rx_trig	Set RX FIFO trigger interrupt enable. FW uses W1S operation to set relevant iena bit.	[7]	WO
		tx_start	Set transmission starts interrupt enable. FW uses W1S operation to set relevant iena bit.	[8]	WO
		tx_ends	Set transmission ends interrupt enable. FW uses W1S operation to set relevant iena bit.	[9]	WO
		rx_ends	Set reception ends interrupt enable. FW uses W1S operation to set relevant iena bit.	[10]	WO
ists_clr	0x3C		USART interrupt status clear register		
		rx_ovr_err	Clear RX overrun error flag. FW uses W1C operation to clear relevant sts bit.	[0]	WO
		tx_ovr_err	Clear TX overrun error flag. FW uses W1C operation to clear relevant sts bit.	[1]	WO
		par_err	Clear parity error flag. FW uses W1C operation to clear relevant sts bit.	[2]	WO
		frm_err	Clear framing error flag. FW uses W1C operation to clear relevant sts bit.	[3]	WO
		ovrlap_err	Clear overlap error flag. FW uses W1C operation to clear relevant sts bit.	[4]	WO
		lin_br	Clear line break flag. FW uses W1C operation to clear relevant sts bit.	[5]	WO
id	0xFC		ID register		

	rev_minor	Minor revision number starting from 0 incremented when existing feature is updated.	[7:0]	RO
	rev_major	Major revision number starting from 1 incremented when the new feature is implemented.	[15:8]	RO
	ip_inst	Instance identifier.	[19:16]	RO
	ip_type	Unique IP identifier.	[31:20]	RO